

# HPS/FPGA Based MD5 Decryption SoC

## COE838: Systems-on-Chip Design Final Project

### 1. Objectives

The purpose of this project is to design a SoC for MD5 Decryption<sup>1</sup> using a HPS/FPGA system. Students will:

- Analyze the provided IP core and its associated VHDL test-benches to obtain hardware characteristics
- Create Avalon MM slaves for interfacing the IP core to a HPS/FPGA SoC.
- Code an HPS application to perform hashing and statistics gathering for 32 MD5 engines executing concurrently (running on a Yocto Linux OS).

Further details are provided below.

### 2. MD5 Algorithm Overview

MD5 is a cryptographic algorithm most commonly used to verify data integrity, especially during data transmissions. An MD5 decryption engine accepts a variable-length message, applies a series of hashing functions, and outputs a 128-bit *digest* (i.e. hashed value message). For the purpose of this lab, we will input a fixed message length consisting of 16 32-bit values (i.e. 512-bit message) to avoid the need for implementing a "padding" algorithm in the software application. The general MD5 algorithm<sup>2</sup> is presented in Fig 1.

As observed in Fig. 1, the MD5 decryption algorithm includes two constant arrays, K and s, and a 512-bit input message represented by the matrix M. Four variables (a0, b0, c0, and d0) are initialized and used to track final digest outputs during computation. The main loop found in Fig. 1 represents the hashing function applied to the input message which employs various arithmetic and logical operations using the two constant arrays and four variables. Once computation has finished, the four 32-bit variables are concatenated and output as the MD5 decrypted digest message.

The VHDL core that invokes this MD5 algorithm has been provided to you in the course directory `/coe838/project/md5/rtl`. This core consists of 32 MD5 engines that compute hashes concurrently. However, these engines may run serially depending on how the software application is coded. Therefore it is the student's responsibility to understand the MD5 VHDL design hierarchy, the RTL provided, and how it corresponds directly to the algorithm of Fig. 1.

The next sections provide details pertaining to the architecture and design specifications for implementing the MD5 Decryption SoC's hardware and software.

---

<sup>1</sup> RTL adapted from Howard Mao's Verilog MD5 Cracker core

<sup>2</sup> [online] Slight variation from the algorithm presented in MD5 Article: <http://en.wikipedia.org/wiki/MD5>

```

//Note: All variables are unsigned 32 bit and wrap modulo 2^32 when calculating
var int[64] s, K
//s specifies the per-round shift amounts
s[ 0..15] := { 7, 12, 17, 22, 7, 12, 17, 22, 7, 12, 17, 22, 7, 12, 17, 22 }
s[16..31] := { 5, 9, 14, 20, 5, 9, 14, 20, 5, 9, 14, 20, 5, 9, 14, 20 }
s[32..47] := { 4, 11, 16, 23, 4, 11, 16, 23, 4, 11, 16, 23, 4, 11, 16, 23 }
s[48..63] := { 6, 10, 15, 21, 6, 10, 15, 21, 6, 10, 15, 21, 6, 10, 15, 21 }

//K constants
K[ 0.. 3] := { 0xd76aa478, 0xe8c7b756, 0x242070db, 0xc1bdceee }
K[ 4.. 7] := { 0xf57c0faf, 0x4787c62a, 0xa8304613, 0xfd469501 }
K[ 8..11] := { 0x698098d8, 0x8b44f7af, 0xffff5bb1, 0x895cd7be }
K[12..15] := { 0x6b901122, 0xfd987193, 0xa679438e, 0x49b40821 }
K[16..19] := { 0xff61e2562, 0xc040b340, 0x265e5a51, 0xe9b6c7aa }
K[20..23] := { 0xd62f105d, 0x02441453, 0xd8a1e681, 0xe7d3fbc8 }
K[24..27] := { 0x21e1cde6, 0xc33707d6, 0xf4d50d87, 0x455a14ed }
K[28..31] := { 0xa9e3e905, 0xfcefa3f8, 0x676f02d9, 0x8d2a4c8a }
K[32..35] := { 0xffffa3942, 0x8771f681, 0x6d9d6122, 0xfde5380c }
K[36..39] := { 0xa4bbee44, 0x4bdecfa9, 0xf6bb4b60, 0xbebfbc70 }
K[40..43] := { 0x289b7ec6, 0xeaal27fa, 0xd4ef3085, 0x04881d05 }
K[44..47] := { 0xd9d4d039, 0xe6db99e5, 0x1fa27cf8, 0xc4ac5665 }
K[48..51] := { 0xf4292244, 0x432aff97, 0xab9423a7, 0xfc93a039 }
K[52..55] := { 0x655b59c3, 0x8f0ccc92, 0xffefff47d, 0x85845dd1 }
K[56..59] := { 0x6fa87e4f, 0xfe2ce6e0, 0xa3014314, 0x4e0811a1 }
K[60..63] := { 0xf7537e82, 0xbd3af235, 0x2ad7d2bb, 0xeb86d391 }

//Initialize variables:
var int a0 := 0x67452301 //A
var int b0 := 0xefcdab89 //B
var int c0 := 0x98badcfe //C
var int d0 := 0x10325476 //D

//Process the message in successive 512-bit chunks:
for each 512-bit chunk of message
    break chunk into sixteen 32-bit words M[j], 0 ≤ j ≤ 15

//Initialize hash value for this chunk:
var int A := a0    var int B := b0
var int C := c0    var int D := d0

//Main loop:
for i from 0 to 63
    if 0 ≤ i ≤ 15 then
        F := (B and C) or ((not B) and D)
        g := i
    else if 16 ≤ i ≤ 31
        F := (D and B) or ((not D) and C)
        g := (5*i + 1) mod 16
    else if 32 ≤ i ≤ 47
        F := B xor C xor D
        g := (3*i + 5) mod 16
    else if 48 ≤ i ≤ 63
        F := C xor (B or (not D))
        g := (7*i) mod 16
    dTemp := D
    D := C
    C := B
    B := B + lefttrotate((A + F + K[i] + M[g]), s[i])
    A := dTemp
end for
//Add this chunk's hash to result so far:
a0 := a0 + A
b0 := b0 + B
c0 := c0 + C
d0 := d0 + D
end for

var char digest[16] := a0 append b0 append c0 append d0 //(Output is in little-endian)

//lefttrotate function definition
lefttrotate (x, c)
    return (x << c) binary or (x >> (32-c));

```

Fig. 1: MD5 Hashing Algorithm (assuming Little Endian)

### 3. MD5 SoC Architecture Overview

The overall HPS/FPGA SoC MD5 architecture for this project is presented in Fig. 2. The MD5 SoC consists of a HPS running an MD5 hash controller application. The HPS software communicates with a series of Avalon Memory Mapped (MM) Slave interfaces to provide input data, obtain output digests, and send/receive control signals to configure, monitor, and initiate calculations in all 32 engines of the MD5 core. The HPS software must also record the total number of hashes computed, total execution time, and overall hash rate for all 32 MD5 engines.

The Avalon MM Slave Interface provides the hardware needed to support hardware/software communication between the HPS and MD5 core. However, it is important that all slaves be correctly configured with the MD5 core to function correctly. Therefore students must develop a SoC containing these slaves, a HPS and a clock source, and map the MD5 core accordingly.

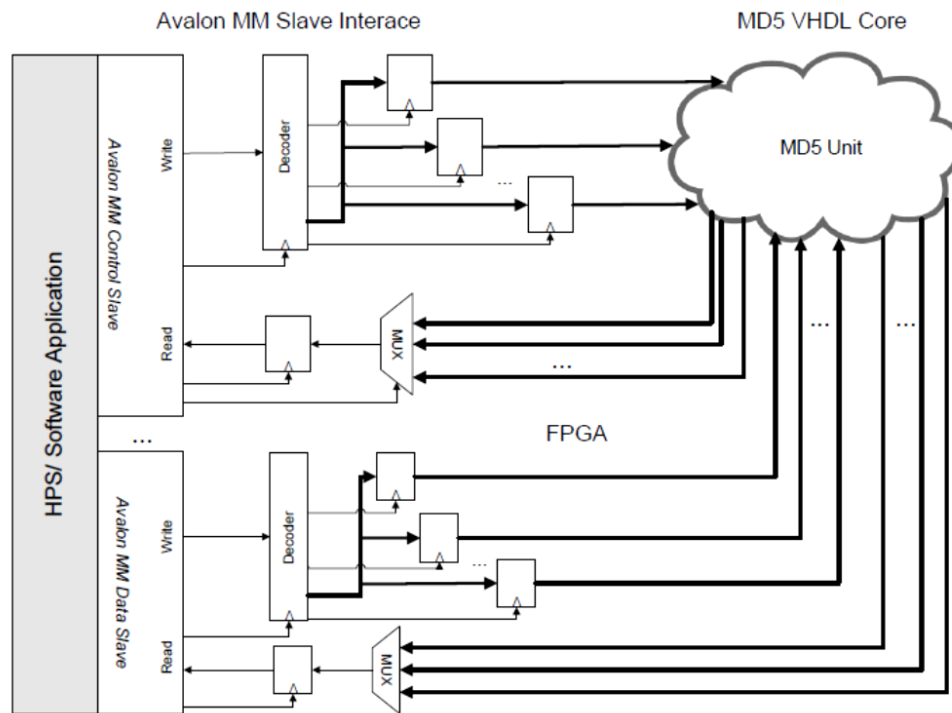


Fig. 2: HPS/FPGA MD5 SoC Architecture

### 4. Design Specifications

#### The Hardware

##### 4.1 MD5 Core Logic

The VHDL for the MD5 core may be found in the course directory */coe838/project/md5/rtl*. The top-level entity for the MD5 core is the file *md5\_group.vhdl*. You will need to go through each individual VHDL file to determine the design hierarchy and component functionality with respect to the overall design. In particular, you will need to understand how:

- an engine is configured for hashing
- a 512-bit message (M) is sent to an engine
- hashing is initiated in an engine
- an engine's completion signal is received
- an engine's 128-bit output digest message is read (once complete)

A VHDL testbench file is included in the */rtl* folder called *md5\_group\_tb.vhdl* simulating 32 engines running concurrently. The testbench *md5\_unit\_tb.vhdl* is provided as well to visualize the simulation of a single engine in the MD5 core. These testbenches may be run using ModelSim to obtain an in-depth understanding of how engines input messages, compute hashes, and receive output digests. A tutorial on running ModelSim with the *md5\_unit\_tb.vhdl* testbench is provided in the Appendix of this document. A sample functional waveform is presented in Fig. 3.

When browsing through the */rtl* folder, you may notice that several .mif files are included in the design. Mif files provide an easy way to implement memory in Altera-based design. Since there are several "constant" values in the MD5 algorithm (i.e. the K and S arrays), ROM's with predefined values were included in the design as *krom.mif* and *srom.mif* respectively. Similarly, since an engine's input message (M) is 512-bits and must be computed in 16 steps, 16 32-bit numbers are written to each engine's dedicated RAM (*mram.mif*) prior to execution, and are read one at a time during computation of the main loop (see algorithm of Fig. 1). Be sure to include all mif based .qip files in your project before compiling.

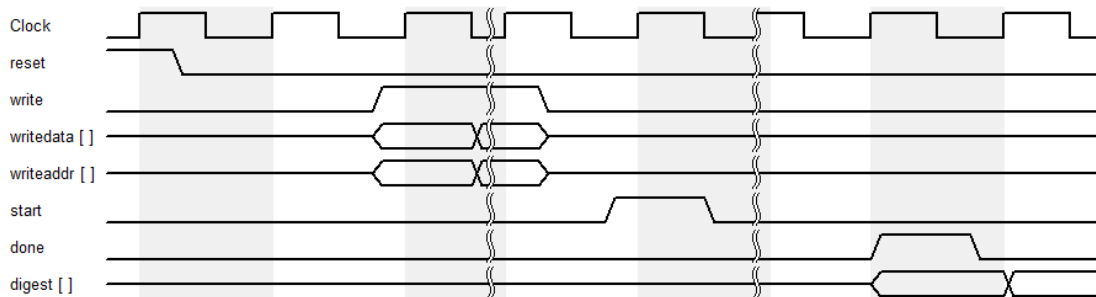


Fig. 3: Example Functional Waveform for one MD5 Engine (*md5\_group.vhd*)

## 4.2 Avalon Memory Mapped (MM) Slaves & SoC Design

The **lightweight** HPS-to-FPGA bridge and AXI bus must be used in your design. This bridge and bus supports a maximum datawidth of 32-bits. Therefore, given that there are 32 engines within an MD5 core, the control signals on a 32-bit bus must be shared and monitored concurrently.

Depending on the architecture of your data I/O Avalon MM slave(s), data may only write or read from one engine at a time (at 32-bit datawidths). Thus a minimum of two Avalon MM slaves are required – one for control and one for data I/O. Since the output digest of an MD5 engine is 128-bits and must be sent to the HPS on a 32-bit bus, the digest must be written back to the HPS in 4 segments. Similarly, the 512-bit input message must be sent in 16 32-bit segments to an MD5 engine's MRAM.

## Software

### 4.3 MD5 Controller Application

The MD5 application will be coded in C, compiled, and executed on the HPS running a Yocto Linux OS. The application must abide by the MD5 core's control signal requirements as presented in Fig. 3 and VHDL testbenches. Messages sent or received from the core must be 32-bit numbers to maintain compatibility with the *lwh2f* bridge. Note that all MD5 core data complies with the *alt\_write\_word()*, *alt\_read\_word()* etc API's of type unsigned 32bit integers (*uint32\_t*).

The 512-bit messages sent to the core may be randomly generated or determined statically in advance. The digests received by the HPS from the MD5 core should be compared to the correct expected digest to ensure correct functionality.

The application must also keep track of the total hashes computed by the MD5 core, the total execution time, and the overall hash rate once complete. The program should also keep track of the correct hashes received, ensuring that it is 100% correct.

Students are to also develop a serial and parallel version of this code (i.e. serial - compute one engine at a time for a given specified time, parallel – allow the 32 engines to compute concurrently and monitor all computations for a given specified time). One .c application should only be coded for both versions. Therefore students are encouraged to use preprocessor flags in their code.

## 5. What to Hand In

This project is due week 12 at the beginning of your lab session. You are expected to deliver the following:




- You are to write a report based on your MD5 SoC design using the IEEE paper format. The final report should be 10-15 pages, not including code and appendices. The following specifications should also be followed:
  - Avoid the cutting and pasting of Figures. Only use pictures and diagrams of your own, and label accordingly.
  - Ensure that you reference as necessary using the IEEE format.
  - Use a suitable font, preferably Times New Roman size 11 or 12 and single line spacing. Single or double column is optional.
  - The pages must be letter size, with 1.0” top, bottom, left and right margins.
- The report must include the following sections:
  - **Abstract**
  - **Introduction**
  - **Past Work/ Review** – *include relevant information on your prior work, along with previous and relevant work on the subject you are implementing (other academic or industry works etc).*
  - **Methodology** – *describe the method you used to design your project from a general perspective. Describe the MD5 algorithm invoked in VHDL. What functionality does your SoC provide? What are the main components? etc. Be sure to use diagrams.*
  - **Design** – *give details based on your methodology for the various components, modules, functions etc you implemented in your project. Again, diagrams are useful.*
  - **Experimental Results** – *describe your results in words, diagrams, discuss serial vs parallel version, logic utilization, performance (frequency, hash rates etc).*
  - **Conclusion**
  - **References**
  - **Appendix** – *include all your .h/c and VHDL files here, including any screenshots necessary of your QSys system, terminal execution etc*
- Ensure that the Ryerson University title page is included, dated and signed with your report attached.
- Students are to present a working demo of the MD5 SoC, displaying software execution on the HPS/FPGA system prototype. You may also be quizzed during the demo to test your knowledge of the topics covered and your implementation methods for the project.

**\*Bonus\*** – create a \*.c application which performs all the steps of MD5 decryption. Create a pure HPS system (using Quartus II and QSys) to run the \*.c application on. Compare performance, logic utilization and frequency to the HPS/FPGA SoC implemented (both serial and parallel approaches). Explain and contrast these results in your report stating its significance.

## A. Appendix

### A.1. Running ModelSim

Once your design is synthesized and has finished compiling in Quartus, you may run the provided testbenches in ModelSim. By using a testbench, stimulus may be input to your Device Under Test (DUT, i.e. your system) so that you may observe the outputs and determine if the system is functioning correctly. We will use the *md5\_unit\_tb.vhdl* testbench in this example, which is provided for the *md5\_unit.vhdl* DUT. Use the testbench to analyze the control signals and verify that the correct output hash has been generated by the DUT. Similarly, you may also use this message input and 128-bit digest output to verify the correctness of your HPS software during hash generation and retrieval.

1. To run the testbench, in Quartus go to “Tools” – “Run Simulation Tool” – “RTL Simulation”. Wait for ModelSim to launch. Next, we must compile the design again in ModelSim.
2. In ModelSim, go to “Compile” – “Compile”. A window will pop up. Browse (using the folder up button) to your folder which contains the VHDL design files. Highlight the project’s VHDL files and select compile. Assuming your design is correct, the files will compile. Also make sure you have compiled the actual testbench file. Select “Done” once complete.
3. In ModelSim, select “Simulate” – “Start Simulation”. Expand the “work” folder and highlight *md5\_unit\_tb.vhdl*. This VHDL should show up as *work.md5unit\_tb* in the “Design Unit(s)” field. Press OK.
4. Next you will see various messages appear in your “Transcript” window with new windows appearing for the simulation. Highlight the “UUT” field in the “sim-Default” window. Right-click the field and select “Add to” – “Wave” – “All Items in Region”. This will open a waveform window.
  - To add more details to the waveform (i.e. test a specific module), you may expand the UUT module and sort through the component hierarchy, highlighting the component of interest and selecting “Add to” – “Wave” – “All Items in Region”.
5. Press the  (run –all) icon. Press stop after one second. Press  Black magnifying glass to view the entire simulation. Use the  Zoom icon to zoom into a section. It is recommended to start from a "reset" signal assertion, and follow the logic until a "done" signal is reached. This trace represents one digest input, its MD5 algorithmic computation, and its 128-bit digest output when complete. Continue to view and explore the simulate data using the waveform editor toolbar.
  - To end the simulation, go to the main window and select "Simulate" - "End Simulation".
6. Each time you have finished simulating or re-adjust your VHDL files, you must compile (in ModelSim) and simulate again, repeating the steps presented in this section.

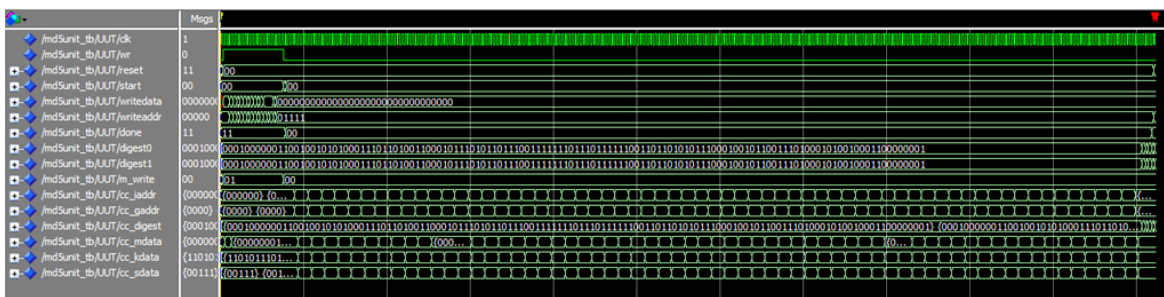


Fig. A.1: ModelSim Simulation Example

## A.2. Analyzing a System Using Quartus II

### A.2.1 Performance Analysis

To view Quartus II's performance analysis, you must first compile the design by setting the appropriate top-level entity. In this case, you may choose to include or not include the HPS-based design, i.e. the speed of the processor will dictate overall speed in the HPS/FPGA SoC, whereas not including the HPS will reflect the maximum frequency of the hardware prototyped (on the FPGA, i.e. *md5\_group.vhdl*). This will also apply to the other analysis parameters found in the next section (A.2.2).

A compilation report will generate when your design is compiled. If you cannot find this report, press CTRL-R. In the Compilation Report's "Table of Contents" window select "TimeQuest Timing Analyzer" – "Slow 1100mV 85°C Model" and select FMax Summary. A report will show. The frequency value represents the maximum frequency that may be achieved by the top-level entity design at 85°C. Now select "Slow 1100mV 0°C Model" and select FMax Summary. The slowest of these two frequencies represents the **overall** maximum achievable system frequency that the device may obtain (on the FPGA).

\*Note that performance for your system will also include the total hashes generated, execution time, and overall hash rate. As you vary execution time, does your hash rate change? Why or why not?

### A.2.2 Logic Utilization Analysis

Select the top-level entity and compile the design to generate a Compilation Report. In the left pane under "Table of Contents", expand the folders "Fitter" – "Resource Section" and double click "Resource Usage Summary". This will give you an outline of how many resources were used from the total available FPGA resource count. In particular, look at the rows pertaining to total ALMs needed, Total LABs, Combinational LUTs, Dedicated Logic Registers, and I/O pins. These FPGA components are usually of interest when assessing hardware area and logic utilization.