University of Victoria

ECE 466

Project Report

2019 Summer Semester

Diffie-Hellman Key Exchange

Multiplier Hardware-Software Co-simulation

Matthew Wegener

V00826848

mwegener@uvic.ca

Submitted to:

Dr. Daler Rakhmatov

Table of Contents

[Introduction 3](#_Toc15858508)

[Project Tasks [1] 3](#_Toc15858509)

[Hardware-Software Handshaking Protocol 4](#_Toc15858510)

[Hardware Handshaking 5](#_Toc15858511)

[Software Handshaking 5](#_Toc15858512)

[Hardware Multiplier 6](#_Toc15858513)

[Structural Description 6](#_Toc15858514)

[Controller Description 7](#_Toc15858515)

[Results 8](#_Toc15858516)

[Future Improvements 8](#_Toc15858517)

[Recommendations 9](#_Toc15858518)

[References 9](#_Toc15858519)

[Appendix A: Code 9](#_Toc15858520)

List of Figures

[Figure 1: Handshaking Protocol 4](#_Toc15858521)

[Figure 2: Handshaking Timing Diagram 4](#_Toc15858522)

[Figure 3: Hardware Structure Block Diagram 6](#_Toc15858523)

[Figure 4: Hardware Controller State Diagram 7](#_Toc15858524)

# Introduction

The project uses a Diffie-Hellman (DH) key exchange protocol, originally written in C and distributed as a part of the NetBench benchmark suite [1]. The purpose of the project is to use SystemC and model one of its computationally intensive functions, NN\_DigitMult, as a hardware module [1]. On the course website the following 7 files have been made available to assist in this task: Makefile, dhdemo.cpp, dh\_sw.h, dh\_sw.cpp, dh\_hw\_mult.h, dh\_hw\_mult.cpp, digit.h [1]. With the provided files, they can be compiled using the Makefile and the SystemC executable main.x can be run. The output looks like this:

\*\*\*Agreed Key: 09 2a f1 41 e2 93 61 d5

\*\*\*Agreed Key: 64 30 94 c5 da d2 f6 da 49 6d 67 f1 16 55 b3 ea ee a2 c0 30 2b b5 4f 05 9e a4 58 ac 97 3b b9 a0 25 b7 56 fe 82 73 bb 22 d4 31 36 60 7f 41 e9 47 97 b9 5e 27 99 3e 73 f0 28 da b5 25 da e4 61 84

[1]

In the project files there are two control signals exchanged between software and hardware: enable and done. Currently, software and hardware are synchronized using only enable and explicit timing delays. The delays used are 100 ns for computation and 10 ns for communication. The enable signal is generated by software to enable hardware multiplication, and the done signal is generated by hardware to indicate that multiplication has been completed.

# Project Tasks [1]

The first task is to replace timed waits with the enable-done handshaking protocol in both hardware software. For handshaking to occur, first the hardware should wait for enable signal to be asserted. Once enable has been asserted, the hardware should execute the multiplication followed with outputting the result and asserting the done signal. The hardware should deassert done only if enable has been deasserted. To implement this a clocked input needs to be added in the hardware and make it a CTHREAD; and in the hardware module an FSM with 4 states:

1. *WAIT:* Wait for the enable signal to be asserted.
2. *EXECUTE:* Multiply the two inputs (using the multiplication code as is).
3. *OUTPUT:* Write to the module's output ports, assert the done signal.
4. *FINISH:* Check if enable is deasserted; if so, deassert done.

The second task is to design the datapath and controller of the hardware multiplier. First, extract the multiplication code inside the EXECUTE state and convert it to the structural description. Then, split the EXECUTE state into as many states as necessary to control the datapath. This makes the datapath controller becomes "embedded" into the handshaking FSM. In the final design implantation, it should produce the same output as the original code without any timed waits.

# Hardware-Software Handshaking Protocol

Part of the hardware-software development is the implementation of a handshaking protocol. The protocol utilizes two signals: enable and done. Hardware controls the done signal and software controls the enable signa, figure 1 shows how the protocol occurs.

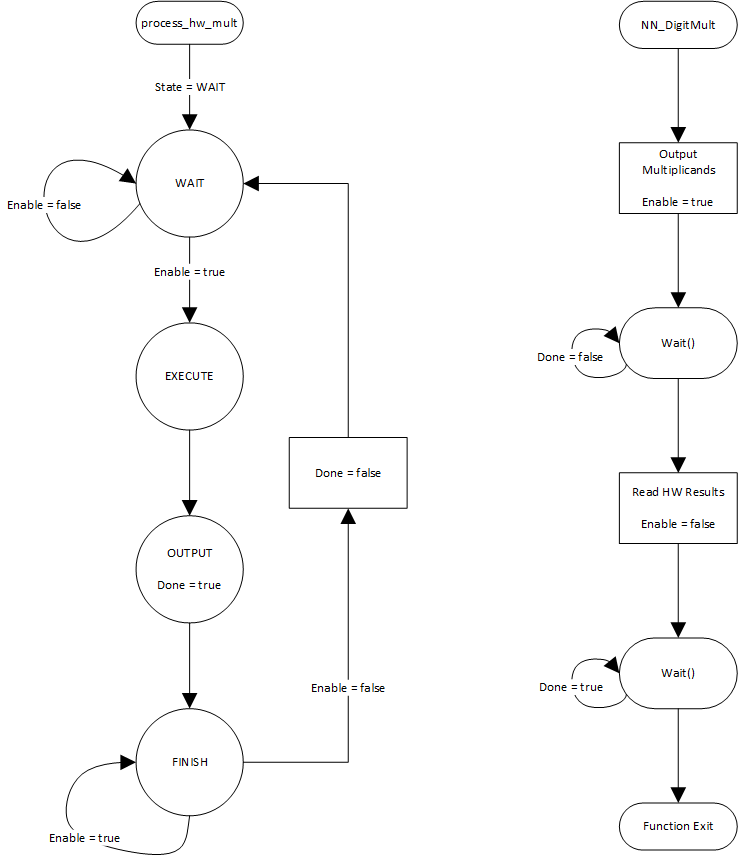


Figure 1: Handshaking Protocol

When the DH algorithm is operating, the hardware waits for the multiplication enable from the software. Once enable has been asserted, the software waits for the hardware done signal to be asserted. After the done signal has been asserted, the hardware will deassert done and the cycle continues for the remaining multiplications. The timing diagram of this interaction can be observed in figure 2 below. The details of the implementation will be discussed in following sections.

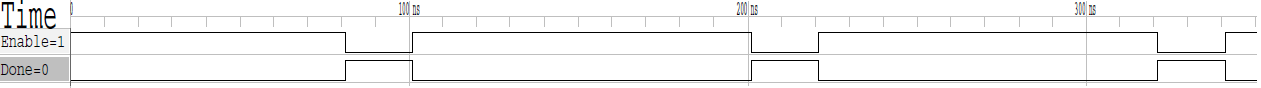


Figure 2: Handshaking Timing Diagram

## Hardware Handshaking

Hardware handshaking is part of the controller implementation. There are three states where the handshaking occurs: WAIT, OUTPUT, and FINISH. A code snippet of these states are provided below:

|  |
| --- |
| case WAIT: //wait for enable to be asserted  if (hw\_mult\_enable.read()){  exec = LOAD;  state = EXECUTE;  }  break;  case OUTPUT: //write to output port & assert done  out\_data\_low.write(a0\_out.read());  out\_data\_high.write(a1\_out.read());  hw\_mult\_done.write(true);  state = FINISH;  break;  case FINISH: //check for enable deassert -> deassert done  if(!(hw\_mult\_enable.read())){  hw\_mult\_done.write(false);  state = WAIT;  }  break; |

While in the WAIT state, the hardware waits until the software asserts the enable signal. Then it does the multiplication (discussed in a later section) before it enters the OUTPUT state. In the output state, it writes to the output variable then asserts a done signal. Moving into the FINISH state, the hardware waits for the software to deassert enable before it deassert the done signal.

## Software Handshaking

The software handshaking primarily occurs in the NN\_DigitMult function. The function takes in the operands as parameters and writes them to the hardware input ports. Then it asserts the enable signal for the hardware to begin exestuation. After asserting the enable, the software waits for the hardware to assert the done signal. Once deasserted, the software reads in the results and deasserts the enable and then waits for the hardware to deassert done. This allows the software to resume its process. In code snippet below of NN\_DigitMult, it can be seen where the software handshaking occurs:

|  |
| --- |
| void dh\_sw::NN\_DigitMult (NN\_DIGIT a[2], NN\_DIGIT b, NN\_DIGIT c){  out\_data\_1.write(b);  out\_data\_2.write(c);  hw\_mult\_enable.write(true);  //Wait until hardware mult is complete  while(!hw\_mult\_done.read()) wait();  a[0] = in\_data\_low.read();  a[1] = in\_data\_high.read();  hw\_mult\_enable.write(false);  //loop till hardware replies by dessarting done signal  while(hw\_mult\_done.read()) wait();  } |

# Hardware Multiplier

As part of the second task a structural implementation and controller for it was implemented in SystemC. The controller was broken into 4 main states with the execute having 3 sub states. As for the structural aspect, it comprised of modules to simulate actual hardware. Both the hardware description and controller are discussed more in depth in the following sections.

## Structural Description

The hardware structure implemented in SystemC uses modules and SC\_Methods to simulate the actual IC component. The implementation can be observed in figure 3 with what components and signals were used.

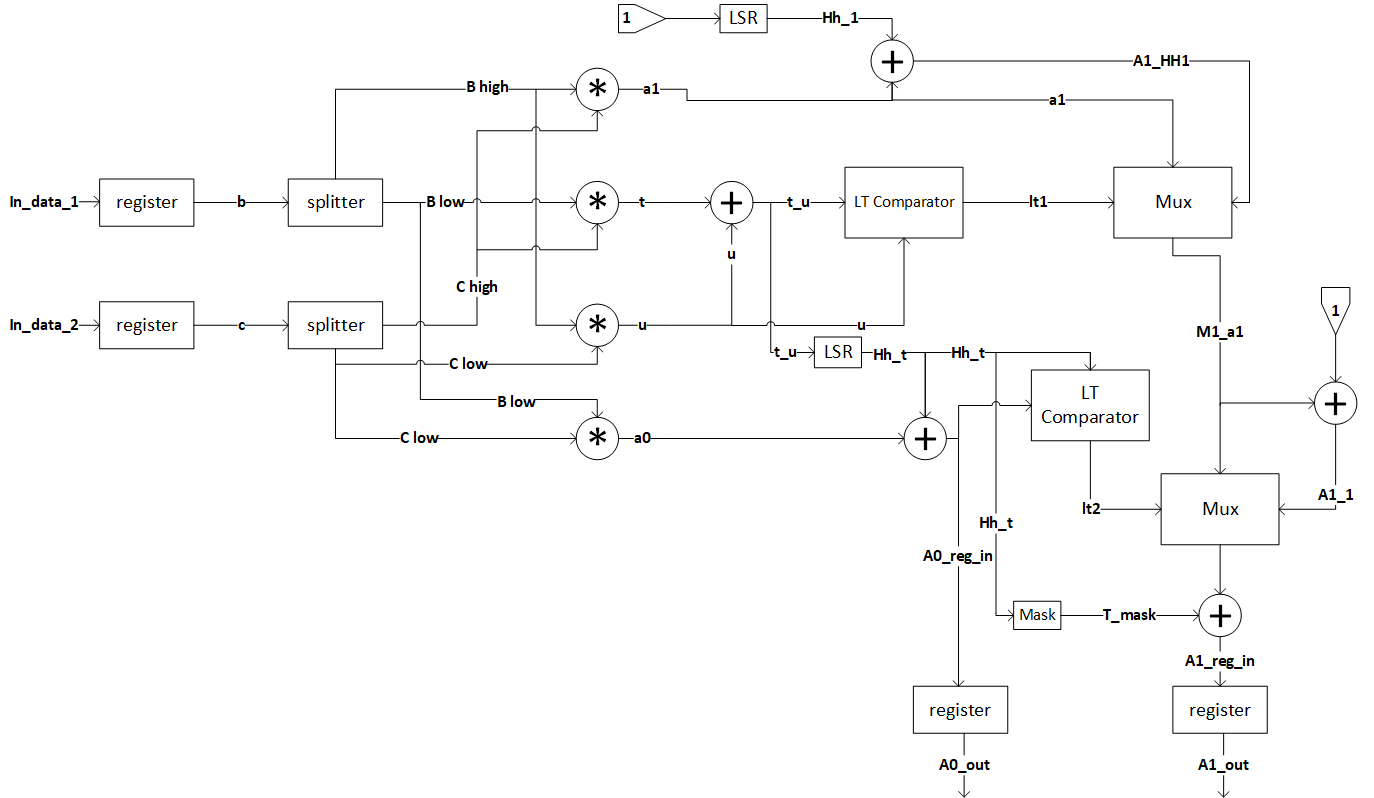


Figure 3: Hardware Structure Block Diagram

Multiplexers and LT Comparators were used as hardware substitutes the software if statement. The comparator would act as the select bit for mux sending it the lt signal. The lt signal would evaluate true if input1 was less than input2. If lt was false, it would be as if the if statement fell through. For example in the following snippet, if lt was true the mux would select the a[1] += TO\_HIGH\_HALF (1); otherwise it would select the unaltered a[1].

|  |
| --- |
| if ((t += u) < u) a[1] += TO\_HIGH\_HALF (1); |

Registers were used to synchronise the input and output of the data with the controller. Left shift registers (LSRs) were used to shift the input to the upper bytes replacing the TO\_HIGH\_HALF in code. Mask was a module that would output only the high half of the input signal where the SystemC implemented HIGH\_HALF macro. The splitter took the input signal and split it into the high half and low half as two different signals of 16-bits each. The multipliers took two 16-bit inputs and output a 32-bit result. Each of the modules were connected using the signals as seen in figure 3. The signals In\_data\_1 & In\_data\_2 are inputs from the software and A0\_out & A1\_out are the results sent from the hardware multiplication sent back to software.

## Controller Description

The controller developed in two steps: a simple implementation for task 1; and a structural implementation for task 2. Task 1 only required a single state for EXECUTE since it only had the unmodified multiplication code from NN\_DigitMult inside it. The controller for the structural implementation utilizes the states presented in the Hardware Handshaking section along with the 3 additional states for EXECUTE shown in figure 3.

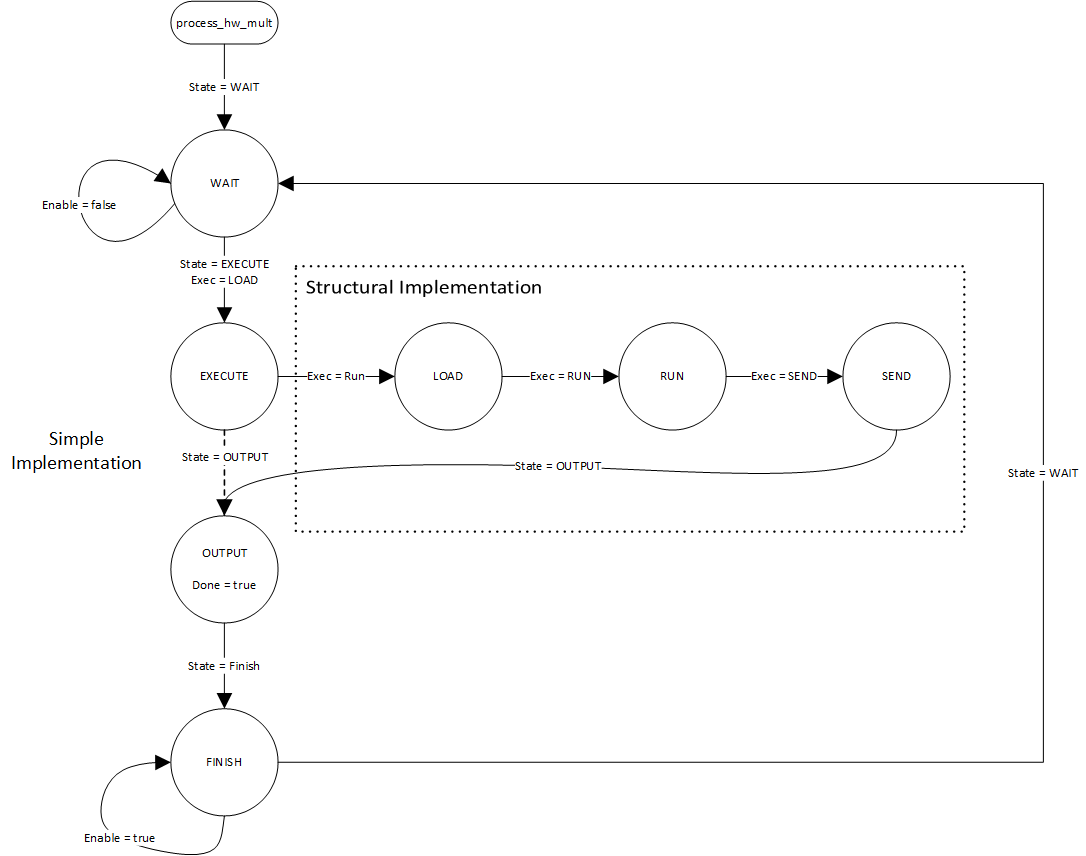


Figure 4: Hardware Controller State Diagram

The three EXECUTE states (LOAD, RUN, SEND) implementation can be seen in the code snippet below:

|  |
| --- |
| case EXECUTE: //do multiplication  switch(exec){  case LOAD:  b\_load.write(true);  c\_load.write(true);  a0\_load.write(false);  a1\_load.write(false);  exec = RUN;  break;  case RUN:  b\_load.write(false);  c\_load.write(false);  a0\_load.write(true);  a1\_load.write(true);  exec = SEND;  break;  case SEND:  a0\_load.write(false);  a1\_load.write(false);  state = OUTPUT;  break;  }  break; |

The LOAD state tells the registers to take in the values provided the software while also ensuring the output registers are not prematurely sending values back to the software. It then transitions to the RUN state where the inputs are propagating through the hardware structure. Also, during this time in the RUN state, the output registers are free to take in values that are ready to be sent back to the software. Moving to the SEND state, the output registers are now closed to send any values to ensure the proper values are read by the software while handshaking in the OUTPUT state.

# Results

The SystemC implementation overall had the correct result which was:

\*\*\*Agreed Key: 09 2a f1 41 e2 93 61 d5

\*\*\*Agreed Key: 64 30 94 c5 da d2 f6 da 49 6d 67 f1 16 55 b3 ea ee a2 c0 30 2b b5 4f 05 9e a4 58 ac 97 3b b9 a0 25 b7 56 fe 82 73 bb 22 d4 31 36 60 7f 41 e9 47 97 b9 5e 27 99 3e 73 f0 28 da b5 25 da e4 61 84

The time in which to complete all multiplications was 180044021 ns, equivalent to 180.04 ms, while total time to create waveform traces and output the agreed key was 10s for the simulation to come to a full stop. No errors were encountered in the process of generating the agreed key.

# Future Improvements

For further refinement of the system additional aspects could be simulated with hardware. The Division algorithm is quite costly as a software implementation and further operation optimizations could occur if it is pipelined through hardware. Another point of potential optimization could be for the hardware to output its values into a fifo as results are completed so the software can then take from it when its ready. This could aid in parallelization of the code while not taking to much of a performance cut.

# Recommendations

A key recommendation for the project would to have a set lab time for access to the lab. At points throughout the day, other course would have a session and those not in that course would be asked to leave the lab. Another prospect with having a lab component, the project could be a little more complex using aspects of scheduling for power and cycle efficiency as part of it.

# References

|  |  |
| --- | --- |
| [1] | D. Rakhmatov, "Project," [Online]. Available: https://www.ece.uvic.ca/~daler/courses/ece466/. [Accessed 1 August 2019]. |

# Appendix A: Code

Full source code for the project as been provided via email.