University of Victoria

Department of Electrical and Computer Engineering

ECE 466 - Project Report

Kyle Cathers

V00851761

August 6th, 2019

1.0 Problem and Solution Description

The Dieffe-Hellman key exchange is a software algorithm which allows for a secure generation of cryptographic keys between two parties over a public medium. The protocol requires the use of many software functions to operate. This report focuses on the task of converting the given 32-bit multiplication function (NN_DigitMult) into a hardware equivalent model using SystemC. The hardware multiplication must communicate with the rest of the software algorithm using a simple handshaking protocol, and must be implemented through structural hardware blocks such as adders and comparators.

To implement the hardware module, a finite state machine was added to allow for the handshaking protocol consisting of 'wait', 'execute', 'output', and 'finish' states. The software multiplication algorithm was converted to a combinational circuit using the basic hardware components connected at the top level. The 'execute' state sends the two operands to the combinational circuit through two additional output ports, and receives the results through two additional input ports. The 'output' and 'finish' states then complete the data transaction with the software module in order to send the results used for the key exchange.

2.0 SW-HW Handshaking Protocol

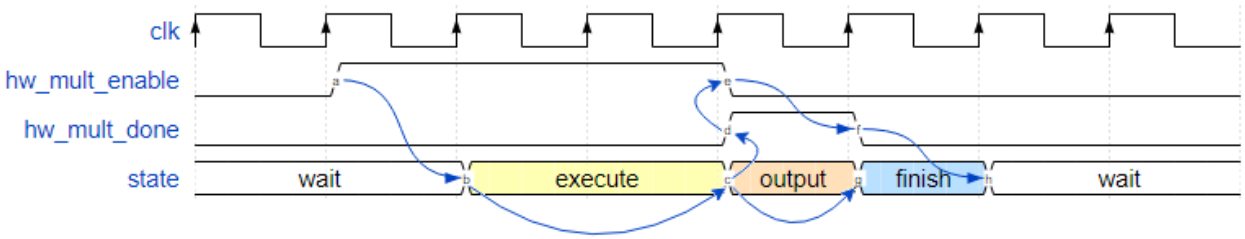

Figure 1: SW-HW Handshaking Timing Diagram

To allow for communication between the hardware multiplier module and the SW module containing the key exchange algorithm, a handshaking protocol was required. An enable signal coming from the SW algorithm, and a done signal coming from the HW multiplier was added to allow for such a protocol. The protocol begins with the SW module calling the NN_DigitMult function with two inputs (multiplication operands). NN_DigitMult then asserts the 'enable' signal (point 'a' in figure 1), which is then sent to the HW module. The HW module then begins the multiplication process (point b), sends the result back, and asserts its 'done' signal (point d) upon completion (point c). The SW module then (asynchronously) detects the done signal, reads the result, and deasserts it's enable signal to the multiplier (point e). Once the multiplier detects that the 'enable' signal has been deasserted, it deasserts the 'done' signal (point f) and resets back to its default waiting mode (point h).

3.0 HW Handshaking Protocol

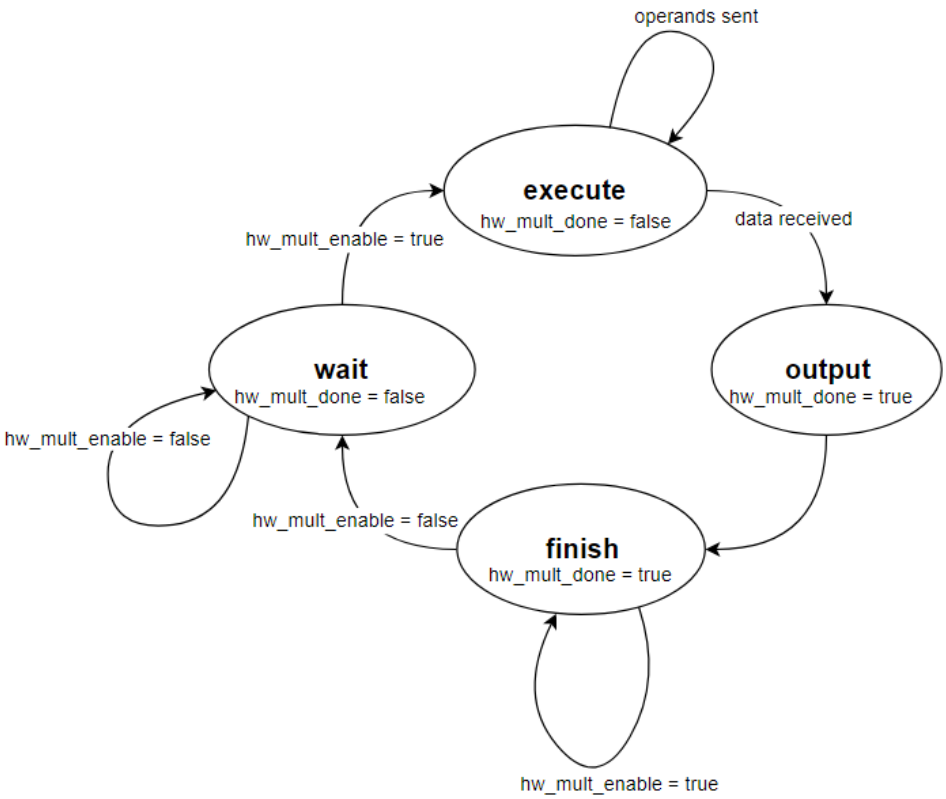

Figure 2: Hardware Handshaking Finite State Machine

The handshaking protocol requires the implementation of a finite state machine (figure 2 above), within the hardware module, using four states: wait, execute, output, finish. The wait state is the default state of the HW module when the enable signal from software is deasserted. Upon the SW module starting the hardware module by asserting the enable signal, the FSM changes to the execute state. The main HW module then sends the operands to the multiplication circuit, and waits for the results. The FSM then switches to output mode once the circuit result is received, sends the result to SW and asserts the done signal. Next clock cycle the FSM switches to its final state, finish, and waits for the SW to deassert the enable signal, thereby acknowledging that the transaction is complete. The done signal is then deasserted and the FSM resets back to its wait state for the next multiplication.

4.0 SW Handshaking Protocol

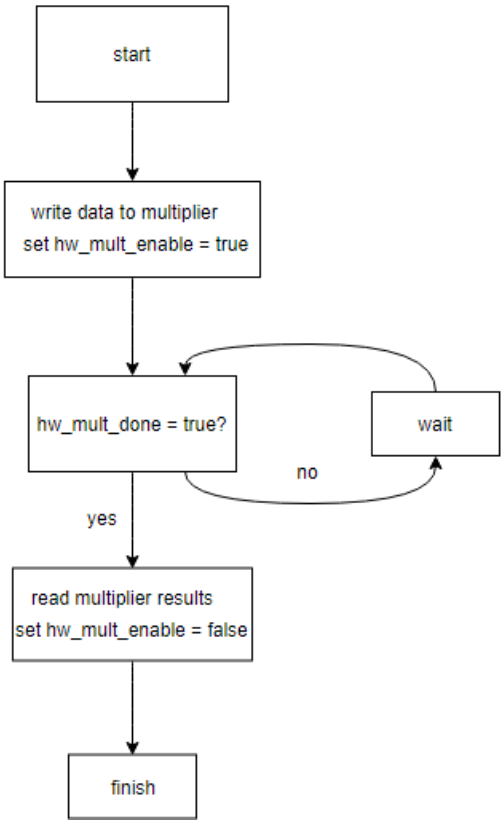

Figure 3: SW Handshaking Flowchart

To allow for the handshaking protocol to be implemented, a few changes had to be added to the NN_DigitMult function within the SW module. The function was modified to wait for the done signal from HW to assert (after the enable signal is asserted). The NN_DigitMult then receives the results from HW and deasserts its enable signal, thus completing the multiplication process for the key exchange. This is a necessary modification from the original code which uses timed waits to simulate the multiplication delay. Handshaking replaces this simulated delay with a much more accurate model.

5.0 HW Multiplier Datapath

The datapath for the HW multiplication was implemented through structural subcomponents which model more basic hardware circuits such as adders, multiplexers, and comparators. To approach the software to hardware conversion, each line of the provided software code was analyzed and converted into its corresponding circuit individually. For example, if-statements were implemented using a combination of comparators (to evaluate the condition) and multiplexers (to act on the result of the condition).

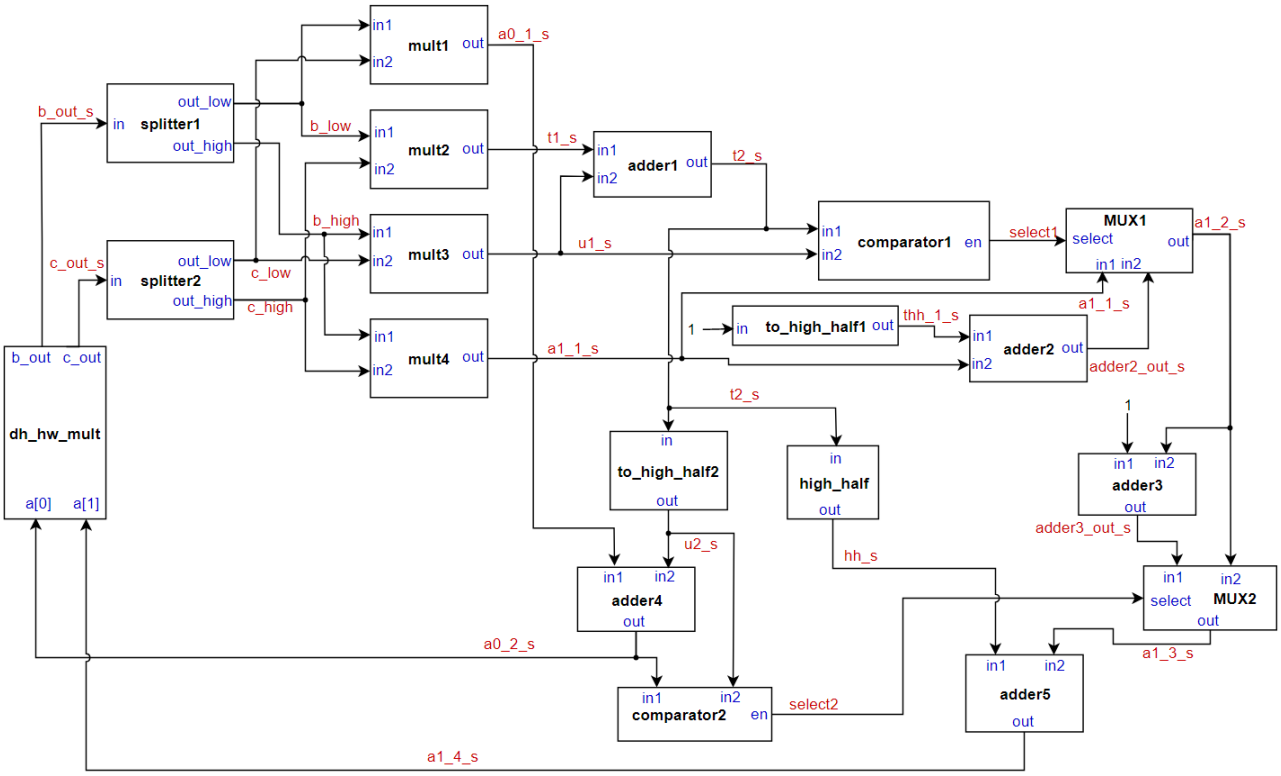

Figure 4: Hardware Multiplier Block Diagram

A block diagram of the multiplier circuit can be seen in figure 4 above. The circuit connects to the dh_hw_mult main module through the b_out_s and c_out_s signals carrying the input operands, and the a0_2_s and a1_4_s signals carrying the results. Note that a0_x_s/a1_x_s corresponds to the x'th stage of the a[0]/a[1] signal being updated (a[0] is updated twice, a[1] is updated 4 times).

The multiplication begins with the operands being split into their upper and lower halves with the splitter modules, which are then passed to 4 different multipliers (operating on 16 bit data, resulting in 32 bit outputs). The first if-statement from the SW code ("if ((t += u) < u)") is implemented using adders 1 and 2, to_high_half1, comparator1, and mux1. The second if-statement ("if ((a[0] += u) < u)") is implemented using to_high_half2, high_half, mux2, comparator2, and adders 2-5. The if-statements operate by evaluating both the results of being true and false concurrently, then selecting the correct result (using the multiplexor) based on the result of the comparator. For example if the first if-statement is true, then a[1] = a[1] + to_high_half(1), but if false a[1] is unchanged. Therefore both versions of a[1] are fed into mux1, with the selection being controlled by the comparator's condition evaluation. The final results from adder 5 and adder 4 are sent back to dh_hw_mult to complete the multiplication.

The original software code and its corresponding hardware components can be seen below in table 1.

Table 1: Corresponding SW to HW Conversion Components

NN_DigitMult SW Code	Corresponding HW Components
bHigh = (NN_HALF_DIGIT)HIGH_HALF(b)	splitter1
bLow = (NN_HALF_DIGIT)LOW_HALF(b)	splitter1
cHigh = (NN_HALF_DIGIT)HIGH_HALF(c)	splitter2
cLow = (NN_HALF_DIGIT)LOW_HALF(c)	splitter2
a[0] = (NN_DIGIT)bLow * (NN_DIGIT)cLow	mult1
t = (NN_DIGIT)bLow * (NN_DIGIT)cHigh	mult2
u = (NN_DIGIT)bHigh * (NN_DIGIT)cLow	mult3
a[1] = (NN_DIGIT)bHigh * (NN_DIGIT)cHigh	mult4
t = t + u	adder1
if (t < u) a[1] = a[1] + TO_HIGH_HALF(1)	comparator1, to_high_half1, adder2, mux1
u = TO_HIGH_HALF(t)	to_high_half2
a[0] = a[0] + u	adder4
if (a[0] < u) a[1] = a[1] + 1	comparator2, adder3, comparator2
a[1] = a[1] + HIGH_HALF(t)	adder5, high_half

6.0 HW Multiplier Controller

Note that since the hardware is implemented as a purely combinational circuit, the multiplication calculation is modelled without an explicit controller. However, this implementation could be changed to include registers to split the multiplication process into multiple clock cycles, allowing for a more realistic multiplication model. Currently the HW is combined into one stage, completing in one clock cycle with no registers. Therefore the use of a controller is unnecessary.

7.0 HW Multiplier Code

The SystemC code for the subcomponents of the HW datapath can be seen below (figures 5-9).

```
SC_MODULE(half_mult) {
    sc_in<NN_HALF_DIGIT> in_data_1;
    sc_in<NN_HALF_DIGIT> in_data_2;
    sc_out<NN_DIGIT> out_data;


    void process_half_mult() {
        out_data.write((NN_DIGIT)(in_data_1.read()) * (NN_DIGIT)(in_data_2.read()));
    }

    SC_CTOR(half_mult) {
        SC_METHOD(process_half_mult);
        sensitive << in_data_1 << in_data_2;
    }
};
```

Figure 5: Code for Half-Digit Multiplier Components (mult1/2/3/4)

```
SC_MODULE(adder) {
    sc_in<NN_DIGIT> in_data_1;
    sc_in<NN_DIGIT> in_data_2;
    sc_out<NN_DIGIT> out_data;


    void process_adder() {
        out_data.write(in_data_1.read() + in_data_2.read());
    }

    SC_CTOR(adder) {
        SC_METHOD(process_adder);
        sensitive << in_data_1 << in_data_2;
    }
};
```

Figure 6: Code for Splitter Components (splitter1/2)

```
SC_MODULE(adder) {
    sc_in<NN_DIGIT> in_data_1;
    sc_in<NN_DIGIT> in_data_2;
    sc_out<NN_DIGIT> out_data;


    void process_adder() {
        out_data.write(in_data_1.read() + in_data_2.read());
    }

    SC_CTOR(adder) {
        SC_METHOD(process_adder);
        sensitive << in_data_1 << in_data_2;
    }
};
```

Figure 7: Code for Adder Components (adder1-5)

```
SC_MODULE(MUX) {
    sc_in<NN_DIGIT> in_data1;
    sc_in<NN_DIGIT> in_data2;
    sc_in<bool> select;
    sc_out<NN_DIGIT> out_data;

    void process_MUX() {
        if (select) {           // select = 1
            out_data.write(in_data1.read());
        } else {                // select = 0
            out_data.write(in_data2.read());
        }
    }

    SC_CTOR(MUX) {
        SC_METHOD(process_MUX);
        sensitive << in_data1 << in_data2 << select;
    }
};
```

Figure 8: Code for Multiplexer Components (mux1/2)

```
SC_MODULE(comparator) {
    sc_in<NN_DIGIT> in_data1;
    sc_in<NN_DIGIT> in_data2;
    sc_out<bool> en;

    void process_comparator() {
        if (in_data1.read() < in_data2.read()) {
            en.write(1);
        } else {
            en.write(0);
        }
    }

    SC_CTOR(comparator) {
        SC_METHOD(process_comparator);
        sensitive << in_data1 << in_data2;
    }
};
```

Figure 9: Code for Comparator Components (comparator1/2)

These components are then instantiated in the dh_demo file at the same hierarchical level as the main HW and SW components (as seen in figure 4). Creating these files as individual components allows for easier code readability and design as the details of each component is hidden from view at the top level. The I/O ports (shown in blue text in figure 4) of each component are then connected using sc_signals (shown in red text in figure 4). Note that the '1' constants being fed into to_high_half1 and adder3 are implemented using sc_signals initialized to '1' (rather than as constants) due to the constraints of ports in SystemC.

The code for the finite state machine of the dh_hw_mult module can be seen in figure 10 below. The FSM is implemented as a switch statement using an enumerated type variable holding the current state of the machine.

```
void process_hw_mult() {
    NN_DIGIT a[2], b, c;

    // states for handshaking
    enum ctrl_state { WAIT, EXECUTE, OUTPUT, FINISH };
    ctrl_state state = WAIT;

    hw_mult_done.write(false);

    while (1) {
        switch (state) {
        case WAIT:      // wait for enable signal to be asserted
            if (hw_mult_enable.read() == true) {
                state = EXECUTE;
            }
            break;

        case EXECUTE:   // multiply two inputs
            // Read inputs
            b = in_data_1.read();
            c = in_data_2.read();
```

```
            // send data out to be multiplied
            b_out.write(b);
            c_out.write(c);
            wait();

            // receive multiplied results
            a[0] = a0.read();
            a[1] = a1.read();

            state = OUTPUT;
            break;

        case OUTPUT:    // write to output ports of module, assert done signal
            hw_mult_done.write(true);
            out_data_low.write(a[0]);
            out_data_high.write(a[1]);
            state = FINISH;
            break;

        case FINISH:    // check if enable is deasserted; if so, deassert done
            if (hw_mult_enable.read() == false) {
                hw_mult_done.write(false);
                state = WAIT;
            }
            break;
        }
        wait();
    }
};
```

Figure 10: SystemC Code for the HW Finite State Machine (dh_hw_mult)

8.0 SW-HW Communication Code

```
/* Computes a = b * c, where b and c are digits.

   Lengths: a[2].
 */
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 for HW multiplier to finish
    do {
        wait();
    } while (hw_mult_done.read() == false);

    // read results from HW multiplier
    a[0] = in_data_low.read();
    a[1] = in_data_high.read();

    // deassert HW multiplier enable
    hw_mult_enable.write(false);
    wait();
}
```

Figure 11: SystemC Code of the Modified Software Multiply Module

As mentioned in section 4.0, the software code for the NN_DigitMult function provided was modified to allow for the replacement of timed waits (simulating HW multiplication delay time) with a handshaking protocol. The function shown in figure 11 can be seen as the other 'end' of the FSM of the hardware multiplier (figure 10). The function starts by sending the data received from the main algorithm (b and c) to the dh_hw_mult module, and asserting enable. The function then waits for input from the hardware (done signal), reads from the input ports, deasserts the enable signal, and suspends. The a[0] and a[1] result signals are then used by the rest of the key exchange algorithm. This process is also illustrated in the flowchart in figure 3.

9.0 Recommendations

As mentioned in section 6, the structural hardware multiplication circuit could be modified to split the process into smaller, simpler states by use of registers holding relevant state data. For example, the first four lines of code in table 1 could correspond to the first stage, the next four lines the second stage, the next three lines (related to the first if-statement) the third stage, and the last three lines (related to the second if-statement) the fourth stage. These stages would then hold their resultant data in a series of 'stage' registers, and pass the data onto the next stage each clock cycle, similar to a CPU pipeline. As most multipliers in devices such as processors require multiple clock cycles to execute, this modification could result in a more realistic model.

The components making up the multiplication circuit could also possibly be optimized to improve speed. For example, the half digit multipliers could be converted to a more explicit multiplication circuit rather than using the built in '*' operator from c++. As well, the if-statements are implemented by calculating both the results of a true or false condition, for example the "if (t < u) a[1] = a[1] + TO_HIGH_HALF(1)" line always calculates a[1] + TO_HIGH_HALF(1) even if the "t < u" condition is false. Instead this could be modified to only calculate the code (possibly using an extra enable signal) within the statement once the condition from the comparator is evaluated true. This would help to reduce the computational requirements of the circuit, possibly improving performance and power consumption.