University of Victoria
ECE466 Project Report:
Hardware-Software Co-simulation

Name: Kirk D’Mello

Student number: V00832871

Submitted on: August 6,2018

Table of Contents

Problem description and solution outline

For this project the task was to implement handshaking functionality and hardware acceleration of the multiplication for a Diffie-Hellman key exchange. The goal of this project was to speed up this key exchange through hardware implementation since the multiplication was done in software which is inefficient. The first step of this was to replace the timed waits found in the original code to a handshake procedure. This allows for more versatile and robust communications since it can adapt. The second part of this project was to translate the multiplication code into hardware with it's own datapath and controller. To test this solution a test demo is performed which should produce the following results to verify proper functionality.

```
*** 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
```

HW-SW handshaking protocol

The hardware-software handshaking is accomplished with the sc_signal values “hw_mult_done” and “hw_mult_enable” which are outputs for the hardware and software respectively/ By lowering and raising these signals in a certain order the hardware and software are able to communicate and perform the multiplication seamlessly. This handshake is illustrated below in the timing diagram.

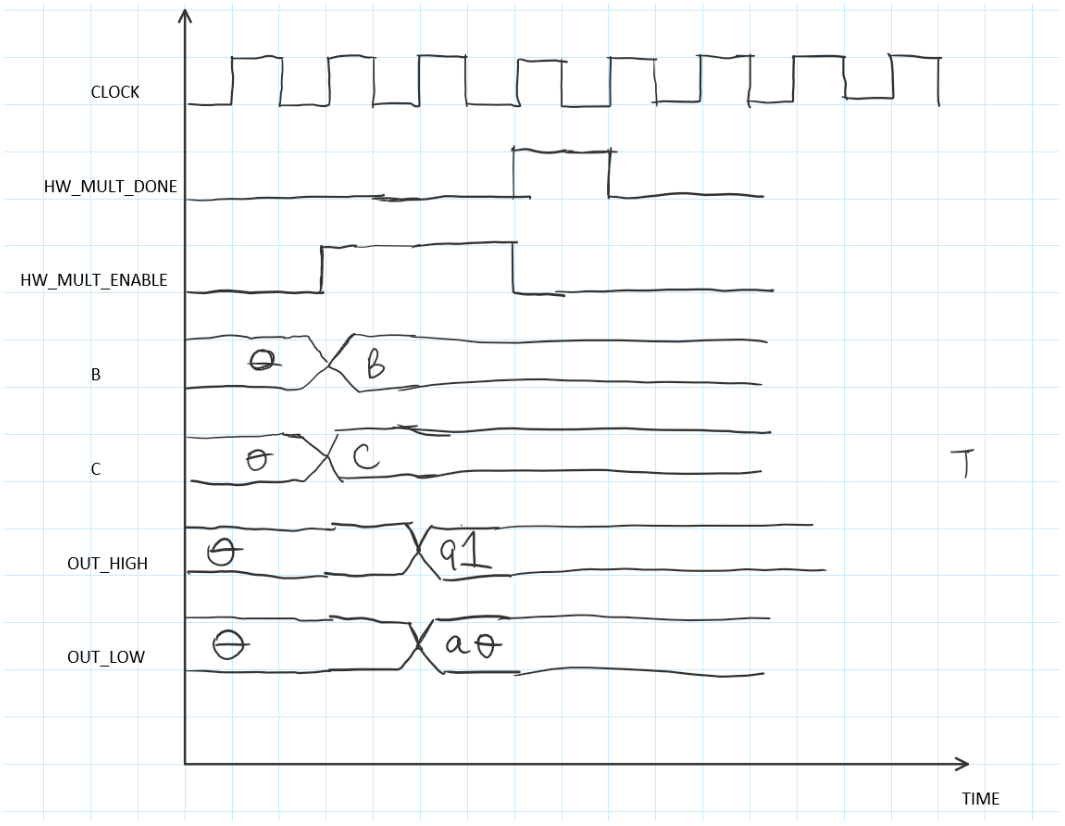


Figure 1: Timing diagram of the handshake protocol.

In this protocol the hardware waits until the software raises the hw_mult_enable signal then it performs the multiplication on it’s operands B&C. Once complete it raises the hw_mult_done signal which signals for the software to deassert the enable signal which then allows the hardware to lower the done signal, finishing the handshake. I implemented this protocol with a Moore machine consisting of four states, S0_WAIT, S1_EXECUTE, S2_OUTPUT and S3_FINISH. The finite state machine diagram can be seen below in figure 2.

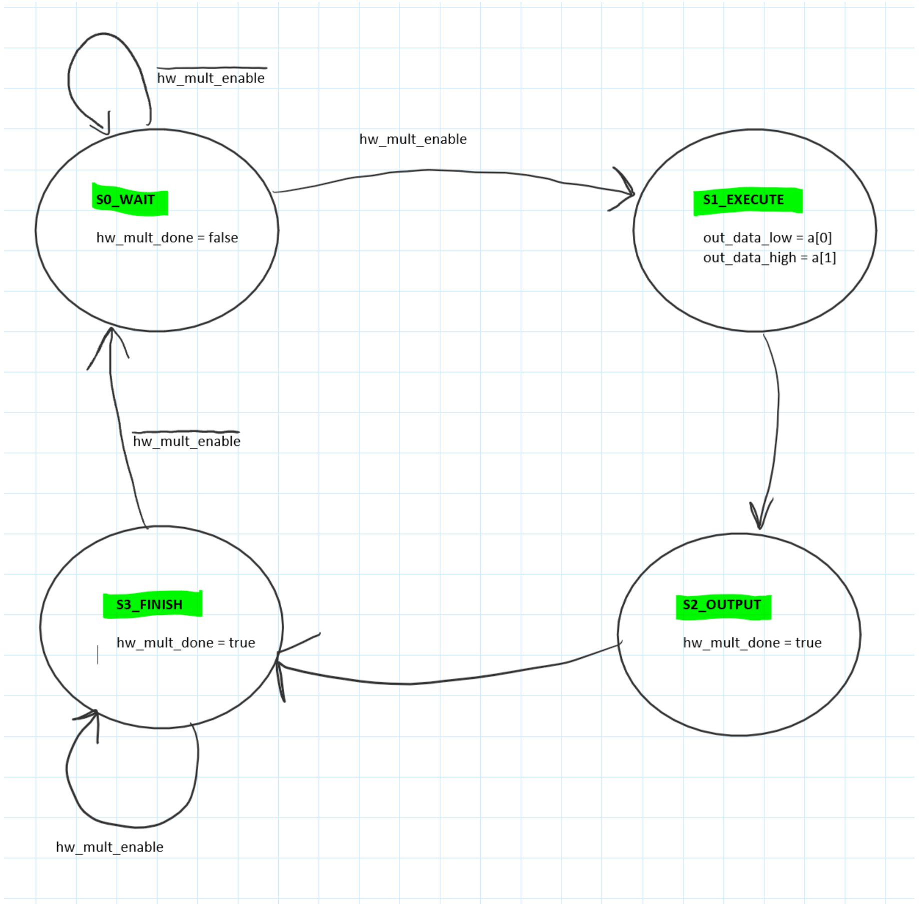
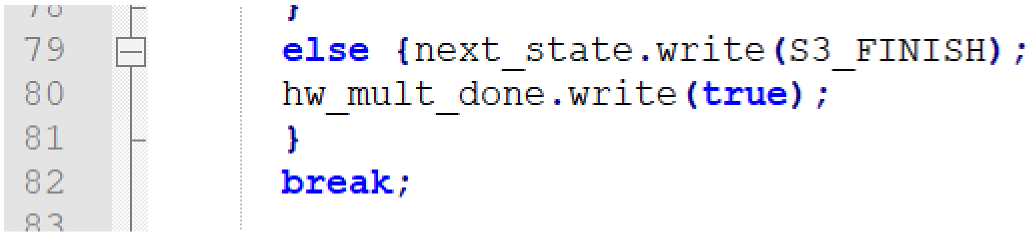


Figure 2: Moore machine for handshaking protocol.

One reason I chose to go with the Moore machine was because at first I tried using a mealy machine but the controller was unable to get out of the finish state because the enable signal would oscillate and cause it to get stuck in the finish state. I suspect that this was happening because before the state register function gets called on the positive clock edge the finish state gets activated again and it takes the else condition where the next state is written to as S3_FINISH and the done signal is re-asserted. This else branch is shown below:



```
        }
        else {next_state.write(S3_FINISH);
        hw_mult_done.write(true);
        }
        break;
```

Figure 3. Else branch in my mealy implementation.

I switched to a Moore machine by adding a two functions to dh_hw_mult.cpp (See appendix A). The first one was state_diagram() which consisted of a switch statement that would assign the correct next state based on the current state as well as the inputs. The next function was state_output(), this function was responsible for taking care of the done signal and the actual multiplication based on the current state. The path for the software can be seen in the flowchart shown as figure 4 below.

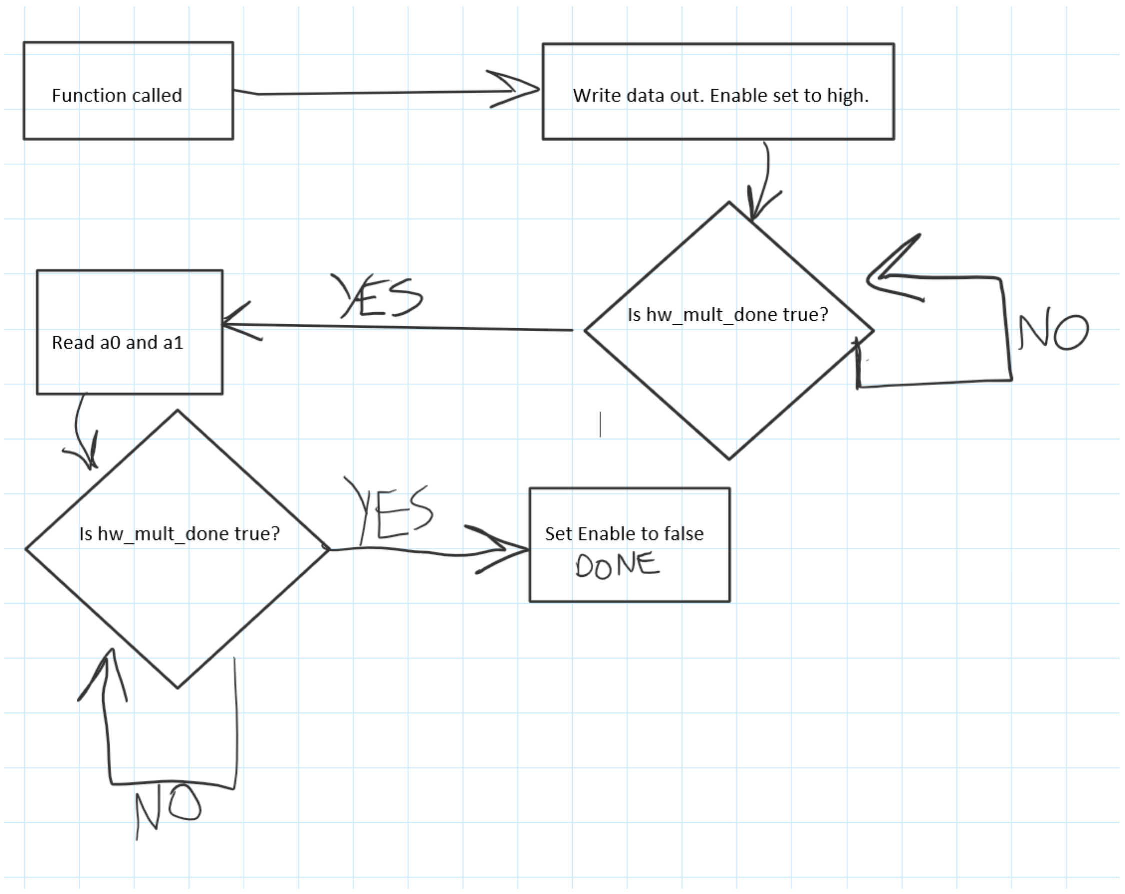


Figure 4: Software flowchart

As shown, the software flow is very linear with only two looping actions when it waits for the signal hw_mult_done to be asserted.

HW multiplier and datapath

Due to time constraints I was unable to implement the datapath for the HW multiplier after planning it out. In this section I'll talk about my proposed datapath as well as a possible Moore finite state machine.

Shown below is the datapath I came up with:

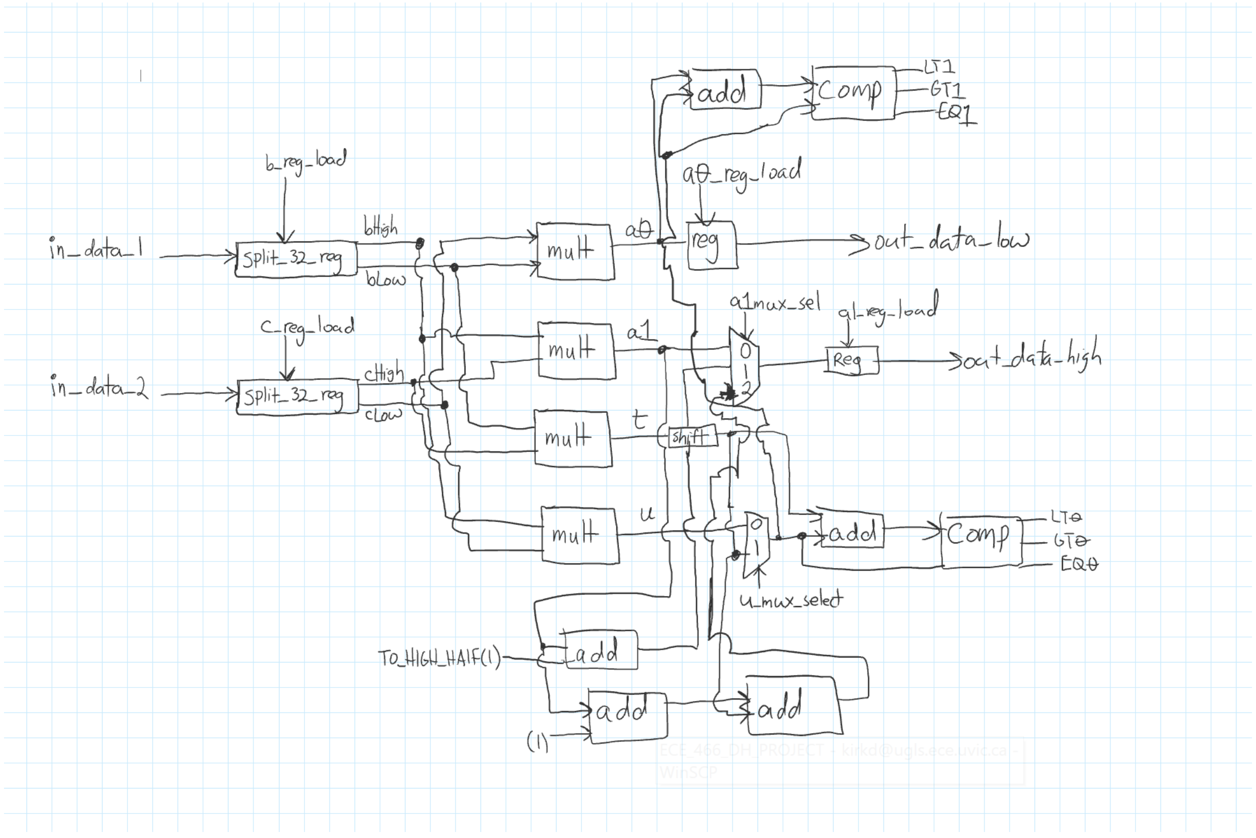


Figure 5: Datapath for hardware multiplication

This design requires 32-bitsplit registers, 16-bit by 16-bit multipliers, adders, 32-bit wide shift registers, 3x32 multiplexers and comparators. I have created these necessary modules and defined them in modules.h. The finite state machine to control this datapath is shown below in figure 6.

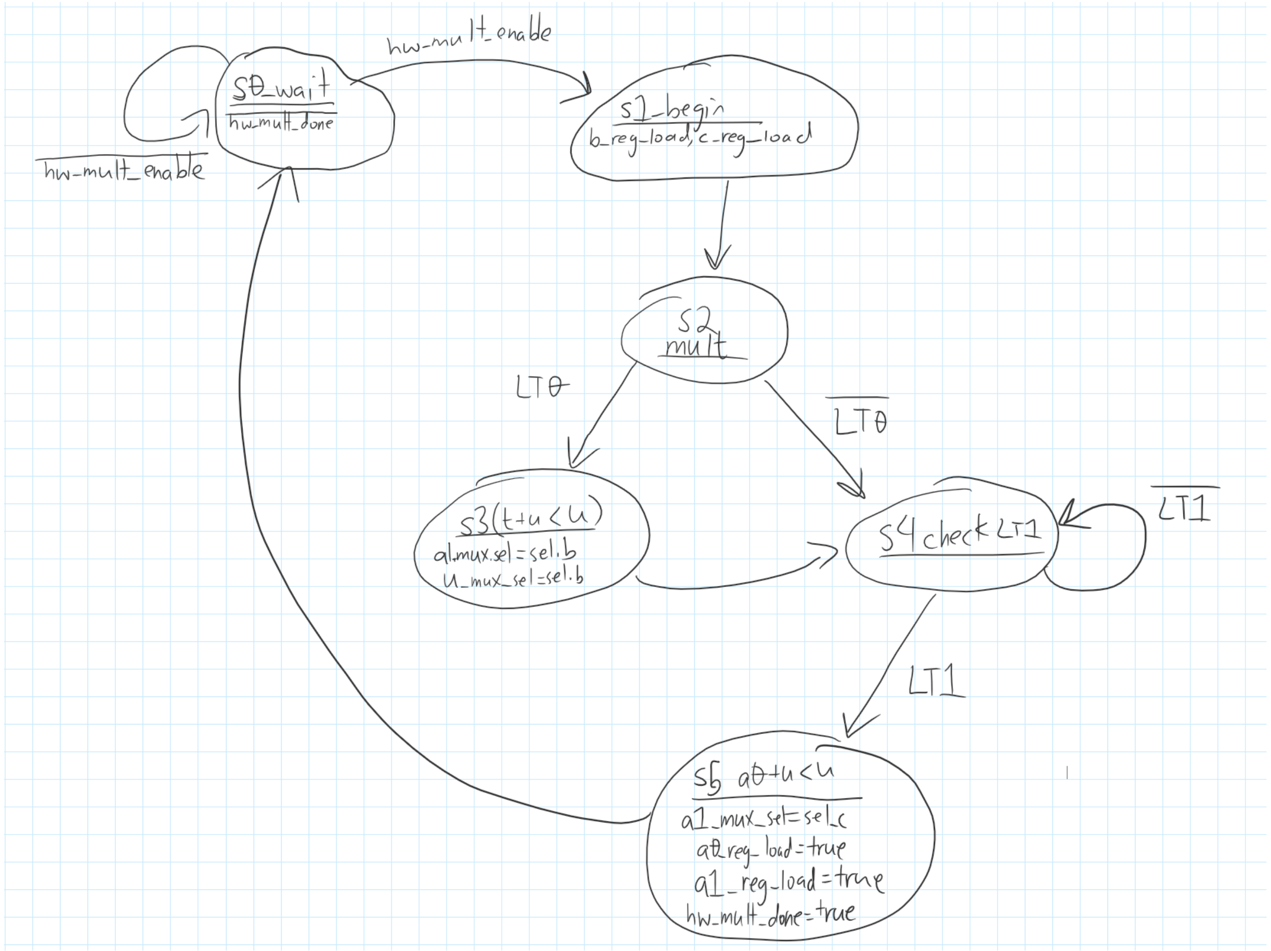


Figure 6: Moore machine for HW datapath.

The overall plan for this datapath was that once ‘b’ and ‘c’ were loaded the multipliers would automatically compute a[0], a[1], ‘t’ and ‘u’. At this point the four values would be shifted, added and compared triggering the two comparators. Depending on the output of these comparators the appropriate value for a[1] would be chosen through the multiplexor and signal “a1_mux_sel”. In the final state “s5” the outputs are loaded onto the data out lines and the done signal is asserted. Since I was unable to test this there are probably mistakes.

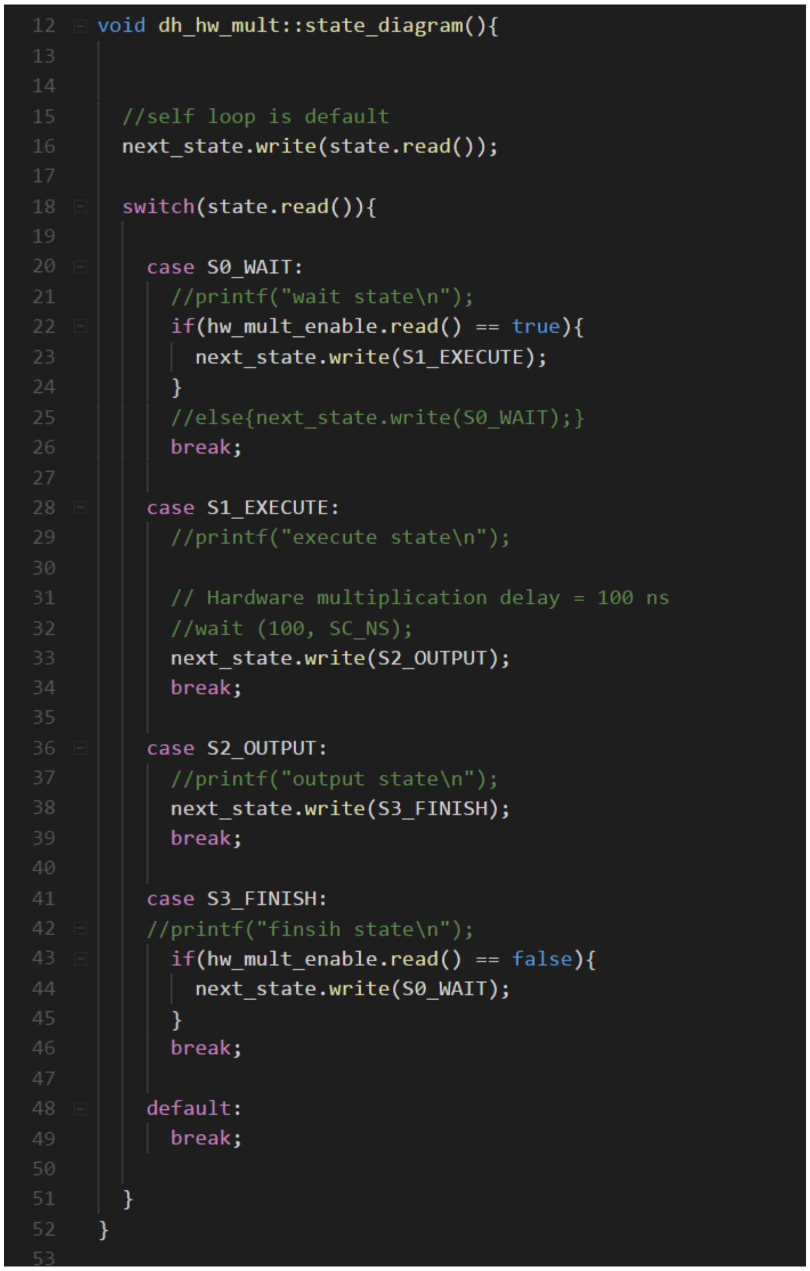
Conclusion and Recommendation

After issuing the ‘make all’ command and running main.x I’m able to verify that the output agrees with the original output, showing that my implementation of the handshake indeed works. This however still uses the original code for the multiplication since my code lacks a datapath. I will continue to try and get the datapath and hardware to work. I would recommend myself to try and clean up the datapath and get rid of any redundancies to ensure a cleaner and easier implementation.

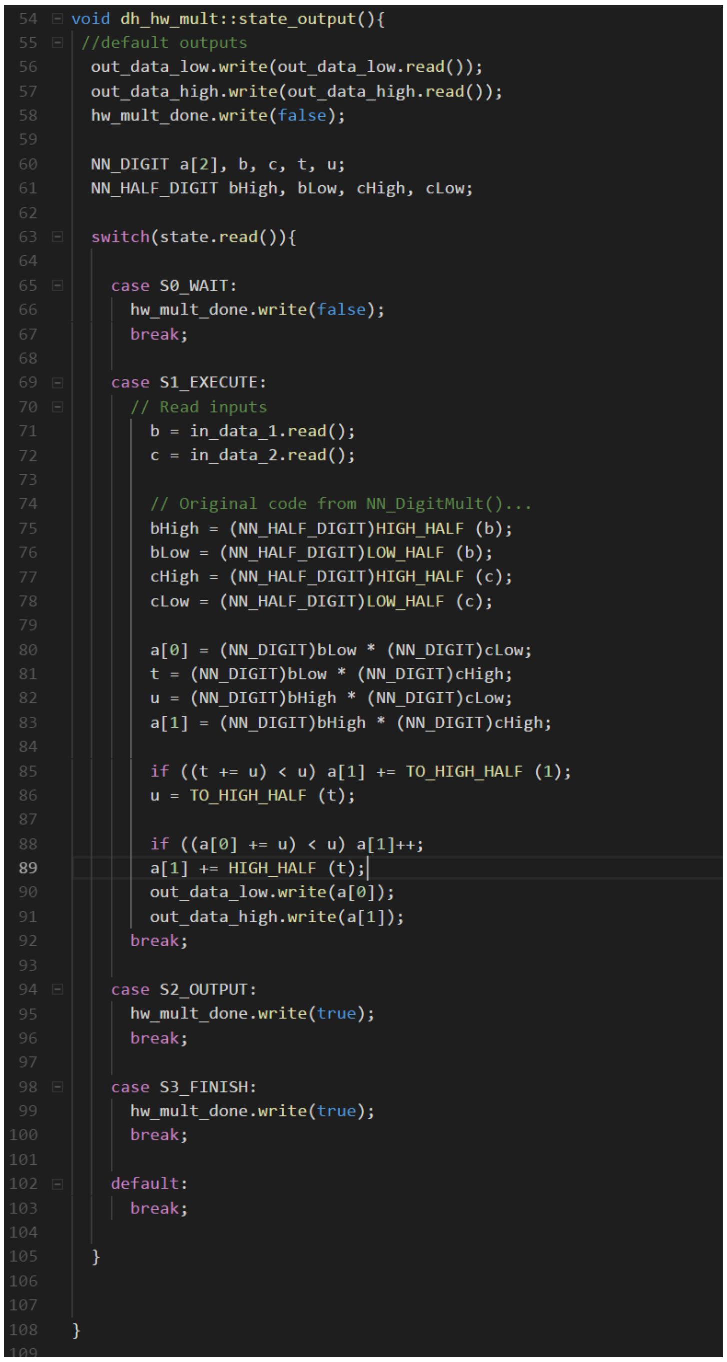
Appendix A

Code snippets of HW and SW communicating

From “dh_hw_mult.cpp”

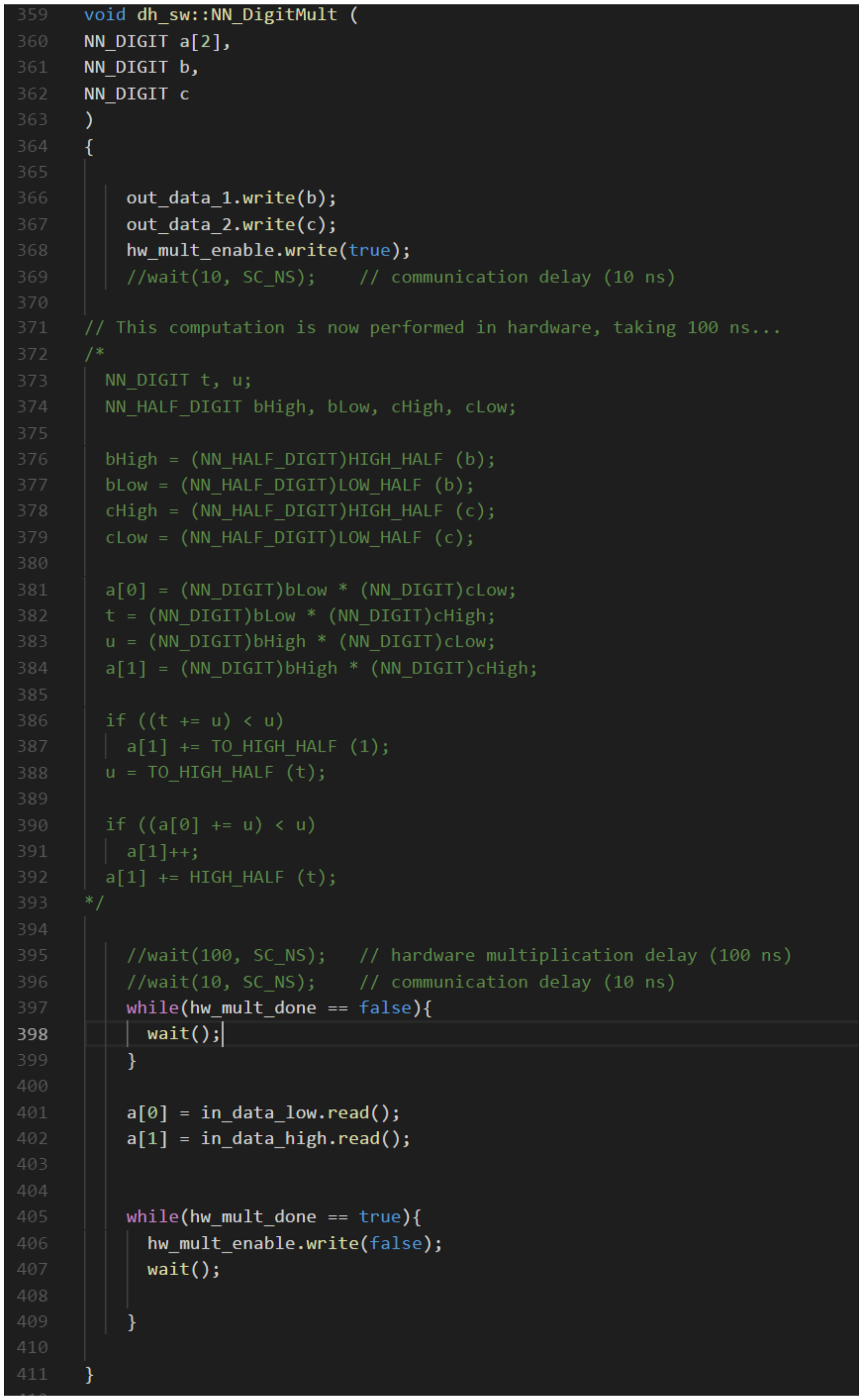


```cpp
void dh_hw_mult::state_diagram(){


  //self loop is default
  next_state.write(state.read());

  switch(state.read()){

    case S0_WAIT:
      //printf("wait state\n");
      if(hw_mult_enable.read() == true){
        next_state.write(S1_EXECUTE);
      }
      //else{next_state.write(S0_WAIT);}
      break;

    case S1_EXECUTE:
      //printf("execute state\n");

      // Hardware multiplication delay = 100 ns
      //wait (100, SC_NS);
      next_state.write(S2_OUTPUT);
      break;

    case S2_OUTPUT:
      //printf("output state\n");
      next_state.write(S3_FINISH);
      break;

    case S3_FINISH:
    //printf("finsih state\n");
      if(hw_mult_enable.read() == false){
        next_state.write(S0_WAIT);
      }
      break;

    default:
      break;

  }
}

```



```
void dh_hw_mult::state_output(){
 //default outputs
  out_data_low.write(out_data_low.read());
  out_data_high.write(out_data_high.read());
  hw_mult_done.write(false);

  NN_DIGIT a[2], b, c, t, u;
  NN_HALF_DIGIT bHigh, bLow, cHigh, cLow;

  switch(state.read()){

    case S0_WAIT:
      hw_mult_done.write(false);
      break;

    case S1_EXECUTE:
      // Read inputs
        b = in_data_1.read();
        c = in_data_2.read();

        // Original code from NN_DigitMult()...
        bHigh = (NN_HALF_DIGIT)HIGH_HALF (b);
        bLow = (NN_HALF_DIGIT)LOW_HALF (b);
        cHigh = (NN_HALF_DIGIT)HIGH_HALF (c);
        cLow = (NN_HALF_DIGIT)LOW_HALF (c);

        a[0] = (NN_DIGIT)bLow * (NN_DIGIT)cLow;
        t = (NN_DIGIT)bLow * (NN_DIGIT)cHigh;
        u = (NN_DIGIT)bHigh * (NN_DIGIT)cLow;
        a[1] = (NN_DIGIT)bHigh * (NN_DIGIT)cHigh;

        if ((t += u) < u) a[1] += TO_HIGH_HALF (1);
        u = TO_HIGH_HALF (t);

        if ((a[0] += u) < u) a[1]++;
        a[1] += HIGH_HALF (t);
        out_data_low.write(a[0]);
        out_data_high.write(a[1]);
      break;

    case S2_OUTPUT:
      hw_mult_done.write(true);
      break;

    case S3_FINISH:
      hw_mult_done.write(true);
      break;

    default:
      break;

  }


}
```

From “dh_sw.cpp”



```cpp
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(10, SC_NS);    // communication delay (10 ns)

// This computation is now performed in hardware, taking 100 ns...
/*
  NN_DIGIT t, u;
  NN_HALF_DIGIT bHigh, bLow, cHigh, cLow;

  bHigh = (NN_HALF_DIGIT)HIGH_HALF (b);
  bLow = (NN_HALF_DIGIT)LOW_HALF (b);
  cHigh = (NN_HALF_DIGIT)HIGH_HALF (c);
  cLow = (NN_HALF_DIGIT)LOW_HALF (c);

  a[0] = (NN_DIGIT)bLow * (NN_DIGIT)cLow;
  t = (NN_DIGIT)bLow * (NN_DIGIT)cHigh;
  u = (NN_DIGIT)bHigh * (NN_DIGIT)cLow;
  a[1] = (NN_DIGIT)bHigh * (NN_DIGIT)cHigh;

  if ((t += u) < u)
    a[1] += TO_HIGH_HALF (1);
  u = TO_HIGH_HALF (t);

  if ((a[0] += u) < u)
    a[1]++;
  a[1] += HIGH_HALF (t);
*/

    //wait(100, SC_NS);   // hardware multiplication delay (100 ns)
    //wait(10, SC_NS);    // communication delay (10 ns)
    while(hw_mult_done == false){
      wait();
    }

    a[0] = in_data_low.read();
    a[1] = in_data_high.read();


    while(hw_mult_done == true){
      hw_mult_enable.write(false);
      wait();

    }

}
```