8/2/2016

Shaun Aiken

V00838297

Hardware Multiplier

For Diffie-Hellman key exchange

Problem description and solution outline

The goal of the project is to implement a multiplier in System C such that it could be synthesized into a hardware implementation. The logic of the hardware implementation is provided through function that operates as software, and this function will be replaced by the hardware implementation.

The framework for the implementation was provided through a software module, a module that contains the software multiplier, and the demo implementation that links them together. The given code contains several hard-coded delays to simulate a possible hardware implementation.

The first step in the project was to remove all timed delays from the hardware and software modules, and implement a handshaking protocol between the two modules. The protocol is to require the software to signal the hardware that the data is ready, the hardware computes the solution and then flags that it is done, then the software removes its enabling flag, and finally the hardware clears its flag that signals completion. At this time, the hardware module is still using the software multiplication implementation.

The second stage is to provide a clock signal to the hardware module to drive its operation. A simple state machine is to be implemented to handle the handshaking and control the multiplication logic. A second state machine will be necessary to control the actual multiplication flow, so as to keep the handshaking separate from the multiplication.

Finally, the software implementation of the multiplier is to be converted to hardware. The individual hardware units, such as adders, shifters, and multipliers, will need to be described and wired. The control logic for the multiplier will be completed at this time.

# SW-HW handshaking protocol timing diagram

The following timing diagram shows the handshaking occurring at 160ns and 340ns.

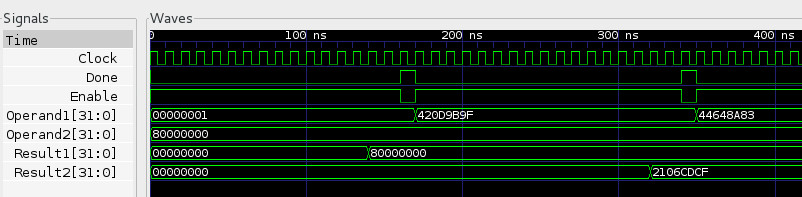


Figure - SW-HW handshaking timing diagram.

State diagram of the handshaking protocol for HW

The hardware module has an overall state machine that implements handshaking with the software, and triggers the multiplication logic. The actual multiplication is driven by a second state machine.



Figure - State diagram for the Hardware control, which implements handshaking through the hw\_mult\_enable and hw\_mult\_done ports.



Figure - Multiplier state diagram - Shown as an addendum, the multiplier is triggered by the state machine in Figure 2.

Flow chart of the handshaking protocol for SW



Figure - Hardware-software handshaking protocol

# Block diagram of the HW multiplier datapath

Shown below in Figure 5 is the datapath for the multiplication logic, as based on the software implementation that was originally a part of the dh\_sw.cpp function NN\_DigitMult().



Figure - Shown are the hardware components, their type in italics, and the named signals that connect each component.

State diagram of the HW multiplier controller

The hardware controller initializes into the S0\_WAIT state. The controller remains in the S0\_WAIT state until the hw\_mult\_enable input is asserted. The state machine moves to the S1\_EXECUTE state, and asserts the mult\_enable signal. The controller remains in the S1\_EXECUTE state until the mult\_done state is asserted. The state machine then moves to the S2\_OUTPUT state and asserts the hw\_mult\_done output. The controller remains in the S2\_OUTPUT state until the hw\_mult\_enable input is deasserted. The controller moves to the S3\_FINISH state, and then moves to the S0\_WAIT state.



Figure - Hardware control

The multiplier controller initializes into the MS0\_WAIT state and remains there until the mult\_enable signal is asserted. The multiplier controller moves to the MS1\_RUN state, which contains multiple wait() commands to facilitate the multiplication logic. NOTE: The MS1\_RUN state could properly be divided up into individual states; The divisions would correspond to each wait(). After moving through the MS1\_RUN state, the controller moves to the MS2\_DONE state. The multiplier controller remains in the MS2\_DONE state until the mult\_enable signal is deasserted, whereupon it moves to the MS0\_WAIT state.



Figure - Multiplier control

SystemC code of the HW multiplier

The multiplier implementation exists in three parts. "dh\_hw\_components.h" describes the hardware used in the multiplier. "dh\_hw\_mult.h" instantiates all of the required hardware and signals, connects the ports, and initializes as needed. "dh\_hw\_mult.cpp" contains the state machines for both the handshaking, and the multiplier.

/\*

\* dh\_hw\_components.h

\*

\* ELEC466 Project

\* Shaun Aiken

\* V00838297

\*/

#ifndef DH\_HW\_COMPONENTS\_H

#define DH\_HW\_COMPONENTS\_H 1

#include "digit.h"

#include "systemc.h"

/\*

\* COMPONENT TEMPLATE

SC\_MODULE (COMPONENT) {

sc\_in<> input;

sc\_out<> output;

void func(void) {

}

SC\_CTOR(COMPONENT) : init{

SC\_(func);

sensitive << input;

}

};

\*/

/\*

\* Multiplier

\*/

SC\_MODULE (multiplier) {

sc\_in<NN\_HALF\_DIGIT> in1;

sc\_in<NN\_HALF\_DIGIT> in2;

sc\_out<NN\_DIGIT> out;

void mult\_func() {

out.write(in1.read() \* in2.read());

}

SC\_CTOR(multiplier) {

SC\_METHOD(mult\_func);

sensitive << in1 << in2;

}

};

/\*

\* Adder

\*/

SC\_MODULE (adder) {

sc\_in<NN\_DIGIT> in1;

sc\_in<NN\_DIGIT> in2;

sc\_out<NN\_DIGIT> sum;

void func(void) {

sum.write(in1.read() + in2.read());

}

SC\_CTOR(adder) {

SC\_METHOD(func);

sensitive << in1 << in2;

}

};

/\*

\* Shift input left by half - logical shift

\*/

SC\_MODULE (half\_shift\_left) {

sc\_in<NN\_DIGIT> input;

sc\_out<NN\_DIGIT> output;

void func(void) {

output.write(input.read() << NN\_HALF\_DIGIT\_BITS);

}

SC\_CTOR(half\_shift\_left) {

SC\_METHOD(func);

sensitive << input;

}

};

/\*

\* Shift input right by half - logical shift

\*/

SC\_MODULE (half\_shift\_right) {

sc\_in<NN\_DIGIT> input;

sc\_out<NN\_DIGIT> output;

void func(void) {

output.write(input.read() >> NN\_HALF\_DIGIT\_BITS);

}

SC\_CTOR(half\_shift\_right) {

SC\_METHOD(func);

sensitive << input;

}

};

/\*

\* Constants - output is based on sel value

\* 0: 1 shifted left by NN\_HALF\_DIGIT\_BITS

\* 1: 1

\*/

SC\_MODULE (const\_mem) {

sc\_in<sc\_uint<1> > sel;

sc\_out<NN\_DIGIT> output;

void func(void) {

switch (sel.read()) {

case 0:

output.write(1 << NN\_HALF\_DIGIT\_BITS);

break;

case 1:

output.write(1);

break;

}

}

SC\_CTOR(const\_mem) {

SC\_METHOD(func);

sensitive << sel;

}

};

/\*

\* 2 Input Multiplexer

\*/

SC\_MODULE (mux\_2) {

sc\_in<NN\_DIGIT> input0;

sc\_in<NN\_DIGIT> input1;

sc\_in<sc\_uint<1> > mux;

sc\_out<NN\_DIGIT> output;

void func(void) {

switch (mux.read()) {

case 0:

output.write(input0.read());

break;

case 1:

output.write(input1.read());

break;

}

}

mux\_2(sc\_module\_name name) : sc\_module(name) {

SC\_METHOD(func);

sensitive << input0 << input1 << mux;

}

SC\_HAS\_PROCESS(mux\_2);

};

/\*

\* 3 Input Multiplexer

\*/

SC\_MODULE (mux\_3) {

sc\_in<NN\_DIGIT> input0;

sc\_in<NN\_DIGIT> input1;

sc\_in<NN\_DIGIT> input2;

sc\_in<sc\_uint<2> > mux;

sc\_out<NN\_DIGIT> output;

void func(void) {

switch (mux.read()) {

case 0:

output.write(input0.read());

break;

case 1:

output.write(input1.read());

break;

case 2:

output.write(input2.read());

break;

case 3:

// Fall through to default - error case

default:

output.write(MAX\_NN\_DIGIT);

break;

}

}

SC\_CTOR(mux\_3) {

SC\_METHOD(func);

sensitive << input0 << input1 << input2 << mux;

}

};

/\*

\* Input splitter - split inputs into high half and low half

\*/

SC\_MODULE (input\_splitter) {

sc\_in<NN\_DIGIT> in0, in1;

sc\_out<NN\_HALF\_DIGIT> in0\_low, in0\_high, in1\_low, in1\_high;

void func(void) {

in0\_low.write(in0.read() & MAX\_NN\_HALF\_DIGIT);

in0\_high.write((in0.read() >> NN\_HALF\_DIGIT\_BITS) & MAX\_NN\_HALF\_DIGIT);

in1\_low.write(in1.read() & MAX\_NN\_HALF\_DIGIT);

in1\_high.write((in1.read() >> NN\_HALF\_DIGIT\_BITS) & MAX\_NN\_HALF\_DIGIT);

}

SC\_CTOR(input\_splitter) {

SC\_METHOD(func);

sensitive << in1 << in0;

}

};

/\*

\* Memory unit - copy input to output on rising clock if enable is set

\*/

template <class T>

SC\_MODULE (mem\_unit) {

sc\_in<T> input;

sc\_in<bool> enable;

sc\_out<T> output;

void func(void) {

while(1) {

if (enable.read()) {

output.write(input.read());

}

wait();

}

}

SC\_CTOR(mem\_unit) :

input("input"), enable("enable"), output("output") {

SC\_THREAD(func);

sensitive << enable;

}

};

#endif

/\*

\* dh\_hw\_mult.h

\*

\* ELEC466 Project

\* Shaun Aiken

\* V00838297

\*/

#ifndef \_DH\_HW\_MULT\_H\_

#define \_DH\_HW\_MULT\_H\_ 1

#include "systemc.h"

#include "digit.h"

#include "dh\_hw\_components.h"

enum ctrl\_states {S0\_WAIT, S1\_EXECUTE, S2\_OUTPUT, S3\_FINISH};//, S98\_INIT, S99\_INIT};

enum mult\_states {MS0\_WAIT, MS1\_RUN, MS2\_DONE};

SC\_MODULE (dh\_hw\_mult) {

sc\_in\_clk clk;

sc\_in<bool> hw\_mult\_enable;

sc\_in<NN\_DIGIT> in\_data\_1;

sc\_in<NN\_DIGIT> in\_data\_2;

sc\_out<NN\_DIGIT> out\_data\_high;

sc\_out<NN\_DIGIT> out\_data\_low;

sc\_out<bool> hw\_mult\_done;

// Signals

sc\_signal<ctrl\_states> state, next\_state;

sc\_signal<mult\_states> mult\_state;

sc\_signal<bool> mult\_enable;

sc\_signal<bool> mult\_done;

sc\_signal<NN\_HALF\_DIGIT> in0\_low, in0\_high, in1\_low, in1\_high;

sc\_signal<NN\_DIGIT> mult\_a0\_out, mult\_a1\_out, mult\_u\_out, mult\_t\_out;

sc\_signal<sc\_uint<1> > a0\_in\_mux, t\_in\_mux, u\_in\_mux;

sc\_signal<sc\_uint<2> > a1\_in\_mux;

sc\_signal<NN\_DIGIT> a0\_mux\_out, a1\_mux\_out, t\_mux\_out, u\_mux\_out;

sc\_signal<NN\_DIGIT> a0\_add\_out, a1\_add1\_out, a1\_add2\_out, t\_add\_out;

sc\_signal<NN\_DIGIT> t\_shift\_left\_out, t\_shift\_right\_out;

sc\_signal<sc\_uint<1> > constants\_sel;

sc\_signal<NN\_DIGIT> constants\_out;

sc\_signal<NN\_DIGIT> a0\_out, a1\_out, u\_out, t\_out;

sc\_signal<bool> a0\_en, a1\_en, u\_en, t\_en;

multiplier mult\_a0, mult\_a1, mult\_u, mult\_t;

adder a0\_plus\_u, a1\_plus\_shift\_t, a1\_plus\_const, t\_plus\_u;

const\_mem constants;

mem\_unit<NN\_DIGIT> a0, a1, u, t;

mux\_2 a0\_mux, u\_mux, t\_mux;

mux\_3 a1\_mux;

half\_shift\_left t\_shift\_left;

half\_shift\_right t\_shift\_right;

input\_splitter splitter;

// Prototypes

void state\_advance();

void state\_control();

void multiplier\_control();

//void do\_mult(); //Software implementation - retain for testing handshaking

// Constructor

SC\_CTOR (dh\_hw\_mult) :

clk("clk"), hw\_mult\_enable("hw\_mult\_enable"), in\_data\_1("in\_data\_1"), in\_data\_2("in\_data\_2"),out\_data\_high("out\_data\_high"), out\_data\_low("out\_data\_low"), hw\_mult\_done("hw\_mult\_done"), mult\_a0("mult\_a0"), mult\_a1("mult\_a1"), mult\_u("mult\_u"), mult\_t("mult\_t"), a0\_plus\_u("a0\_plus\_u"), a1\_plus\_shift\_t("a1\_plus\_shift"), a1\_plus\_const("a1\_plus\_const"), t\_plus\_u("t\_plus\_u"), constants("constants"), a0("a0"), a1("a1"), u("u"), t("t"), a0\_mux("a0\_mux"), u\_mux("u\_mux"), t\_mux("t\_mux"), a1\_mux("a1\_mux"), t\_shift\_left("t\_shift\_left"), t\_shift\_right("t\_shift\_right"), splitter("splitter") {

SC\_THREAD(state\_advance)

// Assigns next\_state signal to state signal

sensitive << clk.pos();

dont\_initialize();

SC\_THREAD(state\_control);

// Overall hardware controller

sensitive << state << hw\_mult\_enable << mult\_done;

dont\_initialize();

SC\_THREAD(multiplier\_control)

// Multiplier controller

sensitive << clk.pos() << mult\_enable;

dont\_initialize();

// Initialization

out\_data\_low.initialize(0);

out\_data\_high.initialize(0);

hw\_mult\_done.initialize(false);

state.write(S0\_WAIT);

next\_state.write(S0\_WAIT);

mult\_state.write(MS0\_WAIT);

mult\_enable.write(false);

mult\_done.write(false);

a0\_in\_mux.write(0);

a1\_in\_mux.write(0);

t\_in\_mux.write(0);

u\_in\_mux.write(0);

a0\_en.write(0);

a1\_en.write(0);

u\_en.write(0);

t\_en.write(0);

constants\_sel.write(0);

// Wiring

a1.input(a1\_mux\_out);

a1.output(a1\_out);

a1.enable(a1\_en);

a0.input(a0\_mux\_out);

a0.output(a0\_out);

a0.enable(a0\_en);

u.input(u\_mux\_out);

u.output(u\_out);

u.enable(u\_en);

t.input(t\_mux\_out);

t.output(t\_out);

t.enable(t\_en);

splitter.in0(in\_data\_1);

splitter.in1(in\_data\_2);

splitter.in0\_low(in0\_low);

splitter.in0\_high(in0\_high);

splitter.in1\_low(in1\_low);

splitter.in1\_high(in1\_high);

mult\_a0.in1(in0\_low);

mult\_a0.in2(in1\_low);

mult\_a0.out(mult\_a0\_out);

mult\_a1.in1(in0\_high);

mult\_a1.in2(in1\_high);

mult\_a1.out(mult\_a1\_out);

mult\_u.in1(in0\_high);

mult\_u.in2(in1\_low);

mult\_u.out(mult\_u\_out);

mult\_t.in1(in0\_low);

mult\_t.in2(in1\_high);

mult\_t.out(mult\_t\_out);

a0\_plus\_u.in1(a0\_out);

a0\_plus\_u.in2(u\_out);

a0\_plus\_u.sum(a0\_add\_out);

a1\_plus\_const.in1(a1\_out);

a1\_plus\_const.in2(constants\_out);

a1\_plus\_const.sum(a1\_add1\_out);

a1\_plus\_shift\_t.in1(a1\_out);

a1\_plus\_shift\_t.in2(t\_shift\_right\_out);

a1\_plus\_shift\_t.sum(a1\_add2\_out);

t\_plus\_u.in1(t\_out);

t\_plus\_u.in2(u\_out);

t\_plus\_u.sum(t\_add\_out);

a0\_mux.input0(mult\_a0\_out);

a0\_mux.input1(a0\_add\_out);

a0\_mux.mux(a0\_in\_mux);

a0\_mux.output(a0\_mux\_out);

a1\_mux.input0(mult\_a1\_out);

a1\_mux.input1(a1\_add1\_out);

a1\_mux.input2(a1\_add2\_out);

a1\_mux.mux(a1\_in\_mux);

a1\_mux.output(a1\_mux\_out);

u\_mux.input0(mult\_u\_out);

u\_mux.input1(t\_shift\_left\_out);

u\_mux.mux(u\_in\_mux);

u\_mux.output(u\_mux\_out);

t\_mux.input0(mult\_t\_out);

t\_mux.input1(t\_add\_out);

t\_mux.mux(t\_in\_mux);

t\_mux.output(t\_mux\_out);

t\_shift\_left.input(t\_out);

t\_shift\_left.output(t\_shift\_left\_out);

t\_shift\_right.input(t\_out);

t\_shift\_right.output(t\_shift\_right\_out);

constants.sel(constants\_sel);

constants.output(constants\_out);

}

};

#endif /\* end \_DH\_HW\_MULT\_H\_ \*/

/\*

\* dh\_hw\_mult.cpp

\*

\* ELEC466 Project

\* Shaun Aiken

\* V00838297

\*/

#include "systemc.h"

#include "digit.h"

#include "dh\_hw\_mult.h"

/\* Expected output:

\*\*\* 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

\*/

/\* Software implementation - retain for testing handshaking

void dh\_hw\_mult::do\_mult() {

NN\_DIGIT a[2], b, c, t, u;

NN\_HALF\_DIGIT bHigh, bLow, cHigh, cLow;

// 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);

// Write outputs

std::cout << "sw out " << a[1] << " " << a[0] << endl;

// out\_data\_low.write(a[0]);

// out\_data\_high.write(a[1]);

}

\*/

void dh\_hw\_mult::state\_advance() {

for (;;) {

// Debug output

//std::cout << "state\_advance" << endl;

/\*If we want to add a reset

if (reset.read() == SC\_LOGIC\_1) state.write(S0\_BEGIN);

else

if (next\_state.read() != state.read()) {

std::cout << "Curr: " << state.read() << "Next: " << next\_state.read() << endl;

\*/

state.write(next\_state.read());

/\* } \*/

wait();

}

}

void dh\_hw\_mult::state\_control() {

for(;;) {

// Debug output

//std::cout << "state\_control" << endl;

switch(state.read()) {

case S0\_WAIT:

// Debug output

//std::cout << "WAIT - (" << sc\_time\_stamp() << ") HW EN: " << hw\_mult\_enable.read() << " HW DN: " << hw\_mult\_done.read() << " M EN: " << mult\_enable.read() << " M DN: " << mult\_done.read() << endl;

// Control

if (hw\_mult\_enable.read() == true) {

next\_state.write(S1\_EXECUTE);

}

else {

next\_state.write(S0\_WAIT);

}

// Action

// No action

break;

case S1\_EXECUTE:

// Debug output

//std::cout << "EXECUTE - (" << sc\_time\_stamp() << ") HW EN: " << hw\_mult\_enable.read() << " HW DN: " << hw\_mult\_done.read() << " M EN: " << mult\_enable.read() << " M DN: " << mult\_done.read() << endl;

// Control

if (mult\_done.read() == true) {

next\_state.write(S2\_OUTPUT);

}

// Action

mult\_enable.write(true);

//dh\_hw\_mult::do\_mult(); // Software multiplier

break;

case S2\_OUTPUT:

// Debug output

//std::cout << "OUTPUT - (" << sc\_time\_stamp() << ") HW EN: " << hw\_mult\_enable.read() << " HW DN: " << hw\_mult\_done.read() << " M EN: " << mult\_enable.read() << " M DN: " << mult\_done.read() << endl;

//std::cout << in\_data\_1.read() << " " << in\_data\_2.read() << " " << out\_data\_high.read() << " " << out\_data\_low.read() << endl;

// Control

if (hw\_mult\_enable.read() == true) {

next\_state.write(S2\_OUTPUT);

}

else {

next\_state.write(S3\_FINISH);

}

//Action

mult\_enable.write(false);

hw\_mult\_done.write(true);

break;

case S3\_FINISH:

// Debug output

//std::cout << "FINISH - (" << sc\_time\_stamp() << ") HW EN: " << hw\_mult\_enable.read() << " HW DN: " << hw\_mult\_done.read() << " M EN: " << mult\_enable.read() << " M DN: " << mult\_done.read() << endl;

// Control

next\_state.write(S0\_WAIT);

// Action

hw\_mult\_done.write(false);

break;

default:

break;

}

wait();

}

}

void dh\_hw\_mult::multiplier\_control() {

while(1) {

// Debug output

//std::cout << "mult\_control (" << mult\_state.read() << ")" << endl;

switch (mult\_state.read()) {

case MS0\_WAIT:

if (mult\_enable.read()) {

mult\_state = MS1\_RUN;

}

break;

case MS1\_RUN:

// mux' should all be set to the multiplier input

// constants should be set to shift 1 to left half

a0\_in\_mux.write(0);

a1\_in\_mux.write(0);

t\_in\_mux.write(0);

u\_in\_mux.write(0);

constants\_sel.write(0);

wait();

// Debug output

// std::cout << in0\_high.read() << " " << in1\_high.read() << " High" << endl;

// std::cout << in0\_low.read() << " " << in1\_low.read() << " Low" << endl << endl;

// Enable copy of values from the multipliers

a0\_en.write(1);

a1\_en.write(1);

u\_en.write(1);

t\_en.write(1);

wait();

// Debug output

// std::cout << "post mult - a0:LL a1:HH u:HL t:LH" << endl;

// std::cout << "a0 " << a0\_out.read() << " a1 " << a1\_out.read() << endl;

// std::cout << "t " << t\_out.read() << " u " << u\_out.read() << endl << endl;

// Stop updating a0, a1, u, and update t with adder output

a0\_en.write(0);

a1\_en.write(0);

u\_en.write(0);

t\_en.write(0);

wait();

t\_in\_mux.write(1);

wait();

t\_en.write(1);

wait();

// Debug output

// std::cout << "t = t + u" << endl;

// std::cout << "a0 " << a0\_out.read() << " a1 " << a1\_out.read() << " " << endl;

// std::cout << "t " << t\_out.read() << " u " << u\_out.read() << endl << endl;

// Stop updating t, conditionally switch a1 mux and update a1, update u

t\_en.write(0);

wait();

// Debug output

// std::cout << "t: " << t\_out.read() << " u: " << u\_out.read() << endl;

if (t\_out.read() < u\_out.read()) {

a1\_in\_mux.write(1);

wait();

a1\_en.write(1);

wait();

a1\_en.write(0);

}

u\_in\_mux.write(1);

wait();

u\_en.write(1);

wait();

// Debug output

// std::cout << "if t < u then a1 += 00010000 - regardless, u = t<<16" << endl;

// std::cout << "a0 " << a0\_out.read() << " a1 " << a1\_out.read() << " " << endl;

// std::cout << "t " << t\_out.read() << " u " << u\_out.read() << endl << endl;

// Stop updating u, switch a0 mux, and allow a0 to update

u\_en.write(0);

a0\_in\_mux.write(1);

wait();

a0\_en.write(1);

wait();

// Debug output

// std::cout << "a0 += u" << endl;

// std::cout << "a0 " << a0\_out.read() << " a1 " << a1\_out.read() << " " << endl;

// std::cout << "t " << t\_out.read() << " u " << u\_out.read() << endl << endl;

// Stop updating a0, if a0 < u then switch the constant to add to a1 and en a1

// regardless, add the high half of t

a0\_en.write(0);

if (a0\_out.read() < u\_out.read() ) {

a1\_in\_mux.write(1);

constants\_sel.write(1);

wait();

a1\_en.write(1);

wait();

a1\_en.write(0);

wait();

}

a1\_in\_mux.write(2);

wait();

a1\_en.write(1);

wait();

// Debug output

// std::cout << "t shifted right " << t\_shift\_right\_out.read() << endl;

// std::cout << "if a0 < u then a1 += 1 - regardless, a1 += t>>16" << endl;

// std::cout << "a0 " << a0\_out.read() << " a1 " << a1\_out.read() << " " << endl;

// std::cout << "t " << t\_out.read() << " u " << u\_out.read() << endl << endl;

// stop updating a1 and signal that multiply is done

a1\_en.write(0);

out\_data\_low.write(a0\_out);

out\_data\_high.write(a1\_out);

wait();

mult\_done.write(true);

mult\_state = MS2\_DONE;

// Debug output

// std::cout << "done - should be same as above" << endl;

// std::cout << "a0 " << a0\_out.read() << " a1 " << a1\_out.read() << " " << endl;

// std::cout << "t " << t\_out.read() << " u " << u\_out.read() << endl << endl;

break;

case MS2\_DONE:

if (mult\_enable.read() == false) {

mult\_done.write(false);

mult\_state.write(MS0\_WAIT);

}

break;

default:

a0\_in\_mux.write(0);

a1\_in\_mux.write(0);

t\_in\_mux.write(0);

u\_in\_mux.write(0);

constants\_sel.write(0);

mult\_state.write(MS0\_WAIT);

break;

}

wait();

}

}

SystemC code of the SW function communicating with HW multiplier

The given code contained a function with the software implementation of the multiplier, as well as several timed wait statements to simulate communication delays with a hardware module. This function did not actually use the hardware module. The code below has the multiplication removed, and uses the hardware logic to compute the multiplication.

/\*

\* The following code is in dh\_sw.cpp and communicates with the hardware

\* multiplier to multiply two NNDIGIT length numbers.

\*/

void dh\_sw::NN\_DigitMult (

NN\_DIGIT a[2],

NN\_DIGIT b,

NN\_DIGIT c

) {

out\_data\_1.write(b); // Set up multiplicands

out\_data\_2.write(c);

hw\_mult\_enable.write(true);// Send Enable to multiplier and wait for Done

wait();

a[0] = in\_data\_low.read(); // Get product

a[1] = in\_data\_high.read();

hw\_mult\_enable.write(false); // Clear Enable and wait for !Done

wait();

}

Recommendations

None at this time.