ELEC 466/568
SystemC Project

Daler N. Rakhmatov

Including material (with modifications) from:

http://cares.icsl.ucla.edu/NetBench
F. Vahid, UC-Irvine

Goal

- Start from a purely software implementation of an application example
  - **Diffie-Hellman** key exchange (NetBench v1.1.0)
- Convert computationally intensive function **NN_DigitMult** into a hardware module
  - Design the datapath
  - Design the controller
- Connect hardware and software using a simple handshaking protocol
  - Use **enable** and **done** signals
- Simulate mixed hardware-software implementation to verify functional correctness

Diffie-Hellman key exchange I

- The Diffie-Hellman key exchange protocol allows two users to exchange a secret key over an insecure medium without any prior secrets
- The protocol has two public system parameters: *prime* **p** and *generator* **g**
  - Parameter **p** is a prime number
  - Parameter **g** is an integer less than **p**, with the following property:
    - For every number **n = 1, 2, …, p–1**, there is a power **k** of **g** such that $\mathbf{n \equiv g^k \bmod p}$
- So, suppose Alice and Bob want to agree on a shared secret key using the Diffie-Hellman protocol…

Diffie-Hellman key exchange II

- First, Alice generates a random **private** value **a**, and Bob generates a random **private** value **b**
  - Both **a** and **b** are integers
- Second, they derive their **public** values using parameters **p** and **g** and their private values:
  - Alice's public value is $\mathbf{A \equiv g^a \bmod p}$
  - Bob's public value is $\mathbf{B \equiv g^b \bmod p}$
- Third, they exchange their public values
- Fourth, Alice computes $\mathbf{B^a \bmod p \equiv (g^b)^a \bmod p}$, and Bob computes $\mathbf{A^b \bmod p \equiv (g^a)^b \bmod p}$
  - Since $\mathbf{(g^b)^a \bmod p \equiv (g^a)^b \bmod p \equiv k}$, Alice and Bob now have a shared secret key **k**

Some definitions and macros

```
typedef unsigned short int UINT2;
typedef unsigned int UINT4;
...

typedef UINT4 NN_DIGIT;
typedef UINT2 NN_HALF_DIGIT;
...

#define NN_DIGIT_BITS 32
#define NN_HALF_DIGIT_BITS 16
#define MAX_NN_DIGIT 0xffffffff
#define MAX_NN_HALF_DIGIT 0xffff
...

#define LOW_HALF(x) ((x) & MAX_NN_HALF_DIGIT)
#define HIGH_HALF(x) (((x) >> NN_HALF_DIGIT_BITS) & MAX_NN_HALF_DIGIT)
#define TO_HIGH_HALF(x) (((NN_DIGIT)(x)) << NN_HALF_DIGIT_BITS)
...
```

NN_DigitMult

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

Hardware multiplier: dh_hw_mult.h

```
SC_MODULE (dh_hw_mult) {

   sc_in <bool> hw_mult_enable;
   sc_in <NN_DIGIT> in_data_1, in_data_2;
   sc_out <NN_DIGIT> out_data_low, out_data_high;
   sc_out <bool> hw_mult_done;

   void process_hw_mult();

   SC_CTOR (dh_hw_mult)  {
       SC_THREAD (process_hw_mult);
       sensitive << hw_mult_enable;
   }

};
```

dh_hw_mult.cpp

```
void dh_hw_mult::process_hw_mult() {
   NN_DIGIT a[2], b, c, t, u;
   NN_HALF_DIGIT bHigh, bLow, cHigh, cLow;

   for (;;) {
     if (hw_mult_enable.read() == true) {
        b = in_data_1.read();    c = in_data_2.read();
        // Original code from NN_DigitMult()
        ...

        // Hardware multiplication delay = 100 ns
        wait (100, SC_NS);
        // Write outputs
        out_data_low.write(a[0]);    out_data_high.write(a[1]);
     }
     wait();            // wait for a change of hw_mult_enable
   }

}
```

Software module: dh_sw.h

```
...
SC_MODULE (dh_sw) {
  sc_in<bool> hw_mult_done;
  sc_in<NN_DIGIT> in_data_low, in_data_high;
  sc_out<NN_DIGIT> out_data_1, out_data_2;
  sc_out<bool> hw_mult_enable;

  void process_sw();

  SC_CTOR (dh_sw)   {
     SC_THREAD (process_sw);
     sensitive << hw_mult_done;
  }
  ...
  void NN_DigitMult (NN_DIGIT [2], NN_DIGIT, NN_DIGIT);
  ...
};
```

dh_sw.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)

   // Multiplication is now performed in hardware...
   wait(100, SC_NS);   // hardware multiplication delay (100 ns)
   wait(10, SC_NS);    // communication delay (10 ns)

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

   hw_mult_enable.write(false);
   wait(10, SC_NS);    // communication delay (10 ns)
}
...
```

Main program: dhdemo.cpp

```
int sc_main () {
   sc_signal <bool> enable, done;
   sc_signal <NN_DIGIT> operand1, operand2, result1, result2;
   enable.write(false);    done.write(false);

   dh_sw DH_SW("DH_Software");
   DH_SW.out_data_1 (operand1);                 // operand1 to hardware
   DH_SW.out_data_2 (operand2);                 // operand2 to hardware
   DH_SW.in_data_low (result1);                 // result1 from hardware
   DH_SW.in_data_high (result2);                // result2 from hardware
   DH_SW.hw_mult_enable (enable);               // enable hardware
   DH_SW.hw_mult_done (done);                   // hardware done

   dh_hw_mult DH_HW_MULT("DH_Hardware_Multiplier");
   DH_HW_MULT.in_data_1 (operand1);             // operand1 from software
   DH_HW_MULT.in_data_2 (operand2);             // operand2 from software
   DH_HW_MULT.out_data_low (result1);           // result1 to software
   DH_HW_MULT.out_data_high (result2);          // result2 to software
   DH_HW_MULT.hw_mult_enable (enable);          // enable hardware
   DH_HW_MULT.hw_mult_done (done);              // hardware done

   sc_start();    return(0);
}
```

Correct 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
```

Things to do I

- Replace timed waits with **enable-done** handshaking protocol in both HW (dh_hw_mult) and SW (dh_sw)
- Example: handshaking in HW
  - HW should wait for **enable** signal to be asserted by SW
  - Once **enable** has been asserted, HW should perform multiplication
  - Then, HW should output the result and assert **done**
  - HW should deassert **done**, but only if **enable** has been deasserted by SW

Things to do II

- To implement handshaking in HW, you need:
  - Add a clock input to HW and make it a CTHREAD
  - Code a simple FSM with 4 states:
    - WAIT - wait for enable signal to be asserted
    - EXECUTE - multiply two inputs (use multiplication code as is)
    - OUTPUT - write to output ports of module, assert done signal
    - FINISH - check if enable is deasserted; if so, deassert done
- To implement handshaking in SW, you need to modify **NN_DigitMult** (dh_sw.cpp)
  - Do NOT feed any clocks to SW!
  - Correct program output does not necessarily mean correct protocol implementation
- There must be **NO** **<u>timed</u> waits** in the final code!

Things to do III

- Design HW datapath and controller
  - Extract multiplication code inside the EXECUTE state and convert it to the structural description using registers, multiplexers, shifters, adders, multipliers, etc…
  - Split the EXECUTE state into as many states as needed to control your datapath
    - Your controller becomes "embedded" into the handshaking FSM
    - Alternatively, you can separate the controller and the handshaking FSM into two communicating state machines
- Submit electronic and hard copies before deadline
  - Submit your **report** in ELEC 466 drop-box
  - **Undergraduate** students: submit your code via ELEC 466 CourseSpaces webpage
  - **Graduate** students: email your code to daler@ece.uvic.ca

Marking

- Project marking scheme
  - Correct output = 25%
  - Correct SystemC code = 50%
  - Project report = 25%
    - See ELEC 466/568 website for report guidelines
- **Extra credit:** 5% of the overall course mark
  - Once your hardware multiplier works, apply the same design steps to create the hardware divisor (NN_DigitDiv)
  - Email your new design files as a separate submission