Design of Digital Platforms

Steven Janssens, Pieter Maene en Kristof Mariën

December 15, 2012

1 Overall Architecture

After having written a software implementation of the Montgomery algorithm in *C*, the conclusion was that this was too slow for a real implementation. Therefore, in the second part of this course, we had to build a hardware coprocessor that should significantly increase speed. The final goal of this project was to build RSA encryption and decryption.

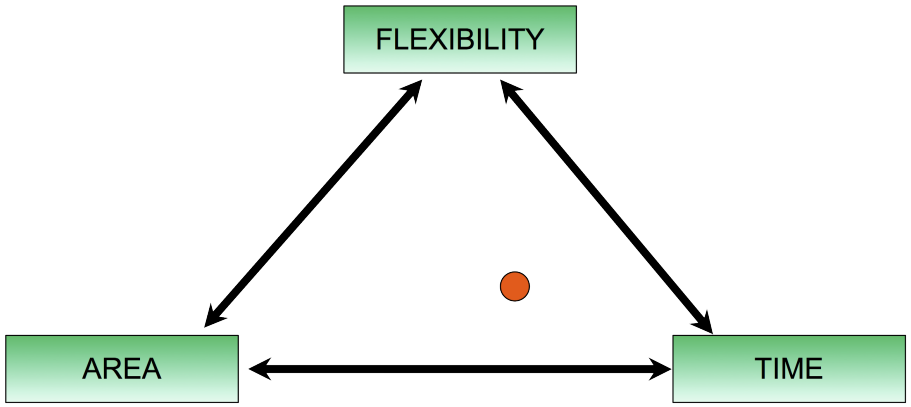

Figure 1: Situation

As you can see in Figure 1, we decided to optimize our implementation towards time usage. Due to inherent design tradeoffs, this meant we had to sacrifice both flexibility and area. We made this choice because the only operation required by the RSA algorithm is an exponentation. So, we opted to build a coprocessor that was specifically designed to do a fast exponentiation.

This hardware component could then be used in the software to implement the RSA encryption and decryption. This resulted in a two-way communication channel between the hardware and the software using the available I/O pins. To pass data back and forth, a memory-mapped implementation was chosen.

2 HW/SW Boundaries

The RSA algorithm is a basic and very often used algorithm for encryption and decryption of data. So it is important that the encryption and decryption is as fast as possible. The algorithm is standardized and will not change in the future. The only part of it that will change is the size of the inputs. Because of these facts a full hardware implementation was chosen.

From the previous section we know that a Montgomery multiplication implemented in *C* takes about 1 second and a Montgomery exponentiation uses a lot of multiplications. So the multiplication must be done in hardware. Our hardware implementation of the multiplication takes about 1300 cycles. Assuming a clock frequency of about 5 MHz, this means a multiplication in hardware takes about 0,26 µs.

So the only choice that was left, was to do the exponentiation in C and call every time the Montgomery multiplication in the hardware or write the exponentiation in *Gezel*. Doing the exponentiation in C would cause a lot of overhead to transfer the data from the software to the hardware and back. So the benefits to write the exponentiation in *Gezel* were quite obvious.

3 HW/SW Interface

For the communication between the hardware and the software a 2-way-handshake method is used. For passing the data from the software to the hardware the following procedure is followed. The software takes care of saving the data on the right places in the memory. If that is done, $P0$ is set to ins_write_data to notify the hardware the data is available. The hardware immediately set $P1$ to one. After that the software loops (with an empty while loop), until the hardware sets $P1$ back to zero. If the software detects that the hardware has returned to an idle state, it notifies the detection to the hardware with an ins_ack on $P0$.

To pass the data from the software to the hardware, all the values are located after each other. So all the values form one block. In the hardware it was set how long each value is and so all the values are again separated.

4 Performance Metrics

5 Synthesis Results

After completing the design of the RSA algorithm in the hardware-software co-design, you might want to put it on a FPGA. Before actually putting the program on it, it is usefull to know apriori how many slices, flipflops and other hardware components you will need. The outcome of the analysation of the code for a specific *Xilinx* FPGA (Virtex4) is shown in table 2. As can be seen, this design can be implemented on the specified FPGA because there are no conflicts in the amount of hardware needed.

Slices	16153 (63%)
Slice Flip Flops	12370 (24%)
4 Input LUTs	30723 (60%)
Bonded IOBs	2 (0.5%)
GCLKs	1 (3%)
Maximum Frequency	11.679 MHz

Table 1: Synthesis Results

6 Test Strategy

Since the RSA algorithm is a quite complex one, a test strategy has to be used. Otherwise it will be very difficult to see where errors occur. Because our code is completely implemented in hardware, the only possibility to give an output is by using display-statements. Further it can be easy to use finish statements in the *Gezel* code to stop the simulation of the hardware immediately. However, this statement can only be used once and sometimes this is not a possibility because the result only comes out after a few iterations. This problem can be solved by using a separate sfg that is used when the result is completed or by using the same display statement in the cycle after the result is ready. But the first thing to test, is whether the inputs are correct. If the values given from the software are not matching the data in the hardware, the result will be quite random and the right result will never be given. Afterwards, the algorithm itself can be debugged.

The RSA algorithm mainly consists of a Montgomery exponentiation, which repeatedly computes Montgomery multiplications. By splitting the complex problem up into small problems, it is also easier to test. So first the Montgomery multiplication will be tested by using several testvectors and see whether

the multiplication is correct. It is best to work with small vectors of 8 bit for instance, because then it is easier to go through all individual steps and errors can be found easier. When the multiplication works for small numbers, it can also be tested on big numbers. If this works properly, the higher complexity of the Montgomery exponentiation can be tested. And finally, the encryption and decryption can be tested in serial by first doing the encryption on the given message and afterwards performing the decryption on this cipher-text. If the result of this decryption is exactly the message originally sent, the RSA algorithm works properly.

For debugging the exponentiation, after checking the correctness of the inputs, the first Montgomery multiplication for $\tilde{x}$ will also displayed. Only if this value is equal to the value coming out of the *Magma calculator*, the further debugging can begin. For easier debugging, the whole exponentiation algorithm was implemented in *Maple*. So each step could be analyzed and errors could be seen quicker. To measure the amount of iterations, a display of the counter can be used. This counter begins at the maximum wanted iterations and decrements always by one before the next iteration is started. When the counter comes to zero, the loop must not be executed and the last Montgomery multiplication can be started. By using the *Maple* file, the outcome of the for-loop is known and this can be controlled first. Afterwards, there is only one Montgomery multiplication left. The outcome of this multiplication is also the output of the Montgomery exponentiation. When using encryption, the cipher-text of the *Magma calculator* must be the result and for the decryption, the original message must be the result.

To check if the result of the encryption or decryption is well transferred to the software, the result has to be sent back to the hardware. Only here the display statements can be used. To control this connection between hardware and software, displays can be used to see which data is sent. When the complete result is transferred to the *C* code, this result has to be read in back to the *Gezel*, where it can be read.

7 Further Optimizations

The hardware and the software of this project can still be optimized. About 12000 bits register are used now, but this number can be reduced. Also the speed of our software can be optimized to write some *assembly*.

To reduce the number of registers, the Montgomery datapath must be changed. Now the multiplication is calculated and saves in a register and during one cycle the value is set on an output signal. Then the exponentiation saves the result in a register and passes it immediately to another Montgomery multiplication. This way there are 'duplicate' registers in the hardware. This can be optimized by rewriting the Montgomery multiplication so it keeps the output longer on the output so the exponentiation doesn't need registers to save the result of the multiplication.

Another optimization for the register count is to calculate the exponentiation inside the decoder datapath. Doing so, the results can overwrite the start values of our exponentiation. But the code will be harder the debug and read.

In the *C* code some instructions take multiple cycles to execute, and this can be reduced by replacing the instructions with more efficient *assembly*.