**FPGA-CPU Co-Processing for GF(2) Matrix-Matrix Product**

**Submitted in partial fulfillment of the degree of**

**Bachelor of Technology**

**By**

**Abhishek Bhowmick**

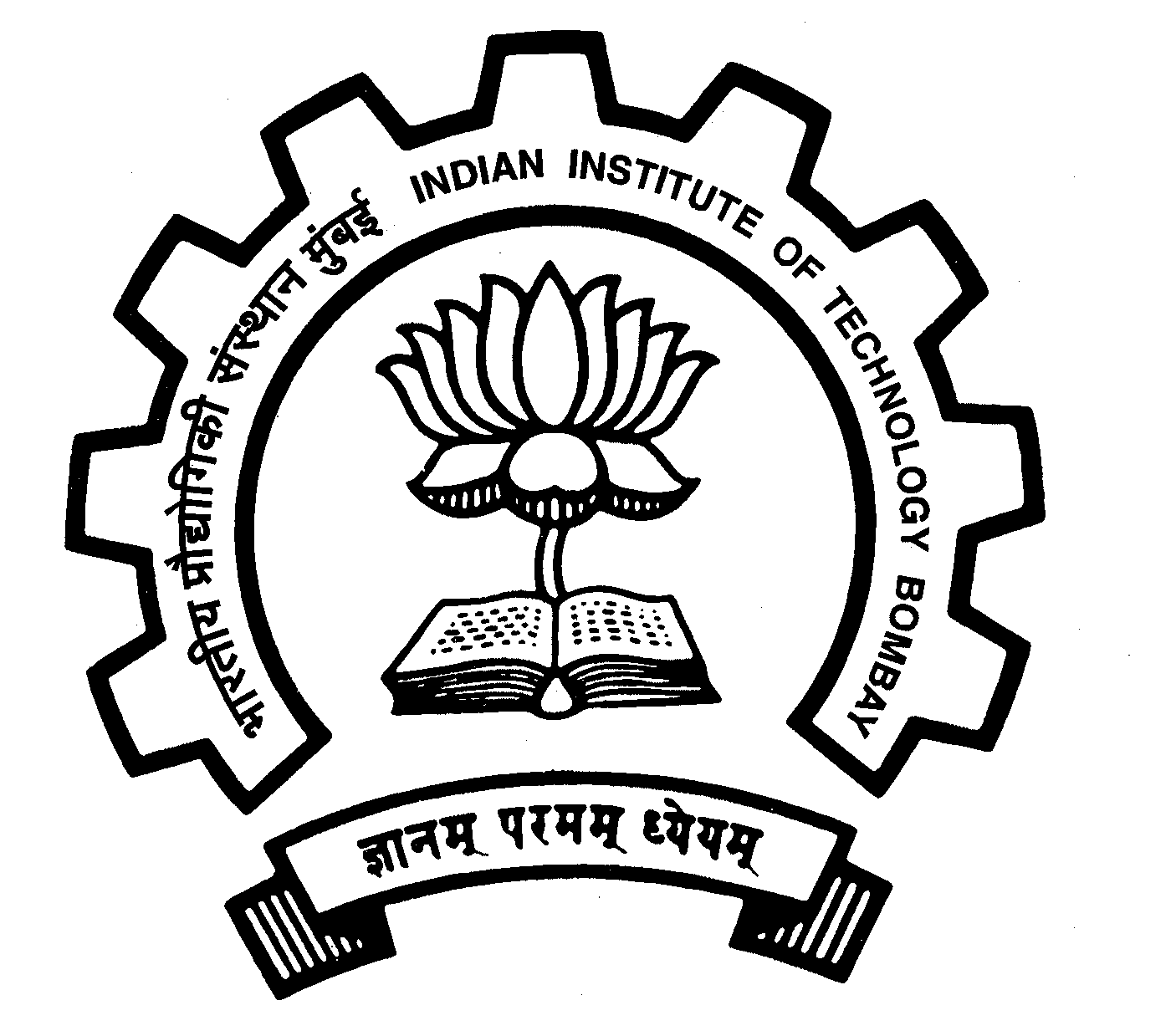
**Roll No. 09007015**

**Udit Jalan**

**Roll No. 09D07017**

**Supervisor:**

**Prof. Sachin Patkar**



**Department of Electrical Engineering**

**Indian Institute of Technology, Bombay**

**2012**

**Approval Sheet**

This report entitled “FPGA-CPU Co-Processing for GF(2) Matrix-Matrix Product” by Abhishek Bhowmick, Roll no 09007015 and Udit Jalan, Roll no. 09D07017, is approved for the degree of Bachelor of Technology.

Examiner

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Supervisor

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Date:\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Place:\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Declaration**

I declare that this written submission represents my ideas in my own words and where others' ideas or words have been included, I have adequately cited and referenced the original sources. I also declare that I have adhered to all principles of academic honesty and integrity and have not misrepresented or fabricated or falsified any idea/data/fact/source in my submission. I understand that any violation of the above will be cause for disciplinary action by the Institute and can also evoke penal action from the sources which have thus not been properly cited or from whom proper permission has not been taken when needed.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(Signature)

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(Name of the student)

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(Roll No.)

Date: \_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(Signature)

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(Name of the student)

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(Roll No.)

Date: \_\_\_\_\_\_\_\_\_\_

**Acknowledgement**

We would like to thank our guide, Prof. Sachin Patkar, Electrical Engineering Department, IIT Bombay for his constant guidance and valuable inputs given during the course of this project. We have gained a lot of insight through our various discussions with him.

We would further like to thank Vinay B.Y. Kumar, PhD student in the Electrical Engineering Department for his tremendous help during the course of this project. Without the long discussions and suggestions from him, this project would not have been possible.

The work presented in this report has been done in collaboration with Yatish Turakhia, Dual Degree student of the department of Electrical Engineering.

Abhishek Bhowmick

09007015

Udit Jalan

09D07017

**ABSTRACT**

Hardware accelerators are increasing becoming popular to meet the ever increasing demand for computational power. Reconfigurable computing using FPGA-CPU co-processing has shown promise to provide increased performance and lower power consumption as compared to general purpose processors and GPUs. However the lack of open source API libraries to provide communication between the host CPU and the FPGA has hindered the penetration of FPGAs as general purpose accelerators. Recently SIRC was introduced by Microsoft Research which provides an open source API for communication and synchronization between the host PC and the FPGA. In this project we have explored the use of FPGA-CPU co-processing for GF(2) Matrix-Matrix Multiplication. Two designs have been proposed, synthesized and verified for correctness using a Xilinx Vertex 5 FPGA on the popular XUPV5 board. We have also explored the use of a few High Level Sythesis tools like c-to-verilog and implemented designs for Discrete wavelet transform and Discrete Fourier Transform.

**CONTENTS**

1. [Introduction](#h.riwrhe4jdiw3) 7
2. [High Level Synthesis](#h.e91os1eixr4) 11
3. [FPGA-PC Communication Using SIRC](#h.s4wxjjlrixvc) 13
4. [Matrix Multiplication over GF(2)](#h.57oh8hmsfx8e) 16
5. [Hardware Design of GF(2) Matrix-Matrix Product](#h.vjf8dskzoa9a) 18
6. [Steps to run the project (rename)](#h.lgc5etmv470p) 26
7. [Documentation of Source Code](#h.yj8x8j21odk0) 28
8. [Conclusion and Future Work](#h.qef4dpdiasxj) 40
9. References 43

**Chapter 1**

# **Introduction**

The demand for increased computational power has been ever increasing. We are already in the era of petascale computing and several research efforts are underway to reach exascale. Several scientific problems have been identified which require computational power of the order of exascale or even beyond. Some examples include weather forecasting, nuclear power plant simulations, and quantum physics.

In order to meet this need for increased computational capabilities, multicore processors have emerged as a possible solution since technological constraints make frequency scaling quite prohibitive. Multiprocessor System on Chip (MPSoC) systems are indeed becoming increasing popular. However for several important applications, general purpose processors are hardly sufficient and dedicated hardware accelerators are needed. Hardware accelerators, simply put, are specialized hardware units which provide significantly better performance than general purpose processors for specific applications. The characteristics of the application are quite critical for extracting increased performance from these accelerators.

Two most popular and widely used hardware accelerators are FPGAs (Field Programmable Gate Array) and GPUs (Graphics Processing Units). GPUs offer thousands of programmable cores which work in parallel. Further GPUs provide a very high memory bandwidth. Also NVIDIA has provided extensive API which make using GPUs easy to use and thus there has been an enormous penetration of GPUs in applications requiring hardware acceleration. FPGAs consist of a large number of Configurable Logic Blocks with programmable interconnects. Basically FPGAs can be “programmed” by the user to create an integrated circuit using a Hardware Description Language like verilog or VHDL. Thus any integrated circuit can be built on an FPGA and the large resources on the FPGA can provide enormous parallelism. Advantages of FPGAs and GPUs for specific applications have been well studied in literature (eg [1],[3],[4]). As compared to GPUs, FPGAs offer greater flexibility and can give better performance and significantly lower power consumption in several applications. As compared to ASICs (application specific integrated circuits) FPGAs offer very low development costs.

Although FPGAs show the promise of better performance at lower power, their application has been limited as compared to GPUs mainly because of the difficulty in hardware design for FPGAs and the lack of open source APIs for communication and synchronization between the CPU and the FPGAs. More precisely, a user looking to accelerate a particular application needs to build a custom interface for communication between his host machine (CPU) and the acceleration platform (FPGA). Only after this basic functionality has been established that the work on the actual application can begin. This is a huge development cost and users are often not willing to bear it. Some proprietary solutions do exist but they are often too expensive and provide poor support. In contrast, CUDA and OpenCL languages with their supporting tools make GPUs easy to use.

Recently an open source project, Simple Interface for Reconfigurable Computing (SIRC) [5], was introduced. It provides an easy to use communication and synchronization API. It comprises of two interfaces - one on the hardware side (on the FPGA) and another on the host machine (CPU). This enables the user C++ code running on the CPU of the host machine to communicate with the user hardware blocks synthesized on the FPGA. Another similar open source tool RIFFA [6] has also been introduced recently.

Figure 1 shows the overall block diagram of how a general purpose accelerator using an FPGA would look like. The user program code would first be compiled by a standard compiler. Next a profiler would go through the program to identify the portions of the software code which are highly parallel and whose performance can be improved by moving to hardware. Next a high level synthesis tool (HLS) would be used to convert the identified software blocks to a synthesizable Hardware description language (HDL) . Alternatively, an IP core library can be used which consists of highly optimized HDL codes of certain standard blocks and whenever such a block is identified in software (maybe by tags placed by the programmer) , it is replaced by the hardware block from the library. Finally the hardware design would be mapped onto a FPGA and the rest of the software code would run on the host PC. The PC and the FPGA would need to communicate and synchronize in order to correctly execute the user application. A system like SIRC is extremely helpful in this part.

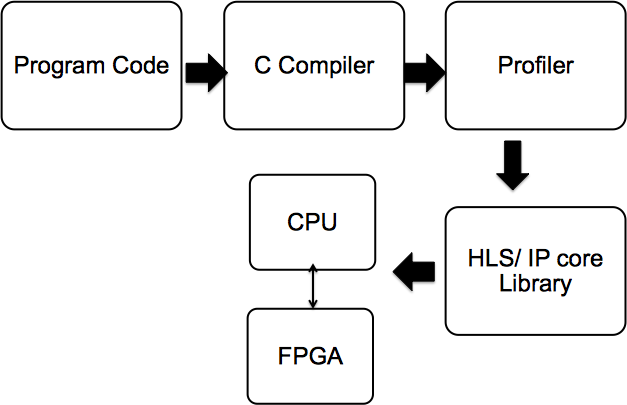


Figure 1: Block diagram of a general purpose accelerator using FPGAs

The aim of the present study is to explore the use of FPGAs in hardware acceleration. We have employed SIRC to provide communication and synchronization between the PC and the FPGA. We have also explored the use of a high level synthesis tool (c-to-verilog) to convert user C code into synthesizable verilog for some applications. Next we have picked a particular application of dense GF(2) matrix-matrix multiplication and have build a complete system which takes 2 large GF(2) matrices and uses the FPGA to compute the product in a blockwise fashion and return the result to the PC.

**Organization of this report**

This dissertation is split into a total of 8 chapters. We outline briefly the contents of these chapters here.

**Chapter 1** serves as an introduction of the work to follow and motivates the work done in this project.

**Chapter 2** provides an overview of High Level Synthesis. It also includes a brief discussion about the modules that we generated and tested using various online open-source RTL core generators.

**Chapter 3** provides a brief overview of SIRC, the communication API for FPGA-CPU co-processing.

**Chapter 4** provides a brief overview of finite field arithmetic over GF(2). It further describes the computation of dense matrix-matrix product over GF(2) and using block computations to create a large matrix matrix product.

**Chapter 5** describes the Hardware design of the GF(2) matrix-matrix product. Two designs have been implemented and each is described in detail.

**Chapter 6** lists down the steps to run the project (Detailed procedure of synthesizing and running our project).

**Chapter 7** provides a documentation of the relevant parts of source code that should be modified by user while developing new applications.

**Chapter 8** concludes the report. It also describes ideas for possible future work. Further some limitations of the current system have also been discussed.

**Chapter 2**

# **High Level Synthesis**

High level synthesis is the process by which logic described in a high level language such as C is converted to hardware specification at the Register Transfer Level (RTL). The output of this stage allows engineers to optimize and verify their designs, following which the RTL design is sent as input to the logic synthesis stage, which creates the final gate level implementation.

Several high level synthesis tools exist in the market, a very popular tool being the BlueSpec compiler from Bluespec, Inc. However, we are interested in open source options, that can be provided for use to small research groups and individuals. Examples of such tools are C-to-verilog [7], LegUp [11] and PandA [12]. Of these, C-to-verilog is a free C to RTL (VHDL or Verilog) compiler with the source code available as well. The resulting code can be synthesized into either an FPGA or ASIC and can be easily integrated into system level design tools such as Xilinx EDK.

This tool supports all primitives of the C language except structs, function pointers, standard C library function calls such as printf, malloc and features such as recursive functions, data types like float/double etc. A full list of synthesizable and non-synthesizable features can be found on the project webpage. The tool also automatically generates test-benches that can be used to verify the module using an RTL simulator like IVerilog or Modelsim.

**Discrete Wavelet Transform**

For a sample run of the tool and testing with our FPGA-CPU communication API, we implemented a discrete wavelet transform module, which is provided as an example on the project website. A pipelined version of this module was synthesized on a Virtex 5 FPGA on the XUPV5 board, with the data for the module generated internally using the provided testbench and the results transferred to the PC using the SIRC API.

**SPIRAL Core Generator**

The SPIRAL project allows us to generate customized soft cores in RTL Verilog for DSP applications through an online interface that can be synthesized on a Xilinx FPGA []. Again, an iterative DFT core generated through the project interface was synthesized and tested on our Xilinx Virtex 5 FPGA. The testbench generated by the project allowed us to verify the operation of a 64-point DFT core, the input being a sequence of 64 complex words. The core consumes two words in each cycle, with each word further represented as a pair of real and imaginary 32-bit fixed point words. Thus, a total of four 32-bit words need to be fed in each cycle which necessitates the use of four input channels. A group of 4 input(output) FIFO queues feeding in(out) data from(to) the BRAM buffer of the SIRC API to(from) the DFT module will enable operation of the module from application software running on host CPU.

**Chapter 3**

# **FPGA-PC Communication Using SIRC**

SIRC [1] (Simple Interface for Reconfigurable Computing) provides an API that allows communication and synchronization of a host CPU with an FPGA (over Ethernet). It abstracts away all the low level hardware and software engineering effort required before project development. In this chapter, we discuss the operation of this API. One of the major advantages of SIRC is that it is consistent across multiple FPGA platforms, thus allowing code reuse.

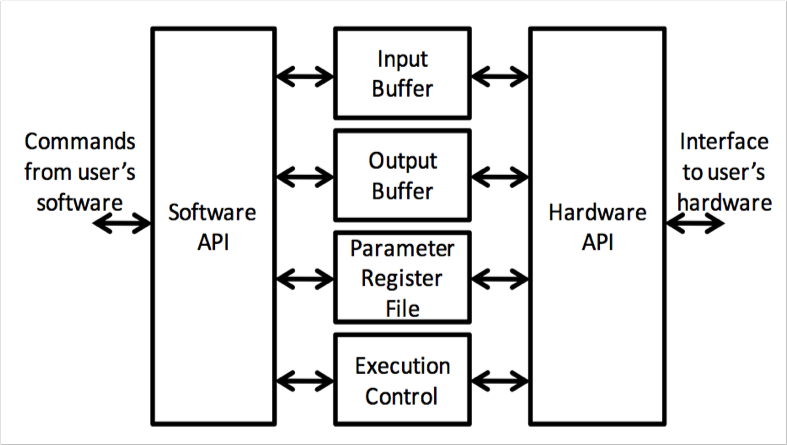


Fig 2 : SIRC software and hardware APIs

The user’s software is a code written in C++ running on a host CPU and the hardware API is a set of primitives that interface with the Verilog-based circuit that was generated by High Level Synthesis and mapped onto the FPGA. The operation of the system occurs in a master-slave arrangement where the C++ software on CPU is the master and the verilog circuit on the FPGA is the slave.

Following are the events that occur to complete the communication routine.

Master :

1. User software sends data to the buffers of the SIRC buffers on the FPGA through Ethernet, by

calling *sendWrite*() from the software API.

2. On complete transfer of data, it signals the hardware API to start operation on the supplied data, calling *sendRun*().

3. It waits (implemented by *waitDone*() )for a completion signal from the hardware API, and then retrieves the data from the SIRC output buffers, by calling *sendRead*().

Slave :

1. Hardware side waits for signal from software side to start executing on data.

2. Operate on the data loaded from input buffers and load the results into output buffers.

3. Signal the software side that execution is complete and return to idle state.

The functions *sendWrite(), sendRun(), waitDone()* and *sendRead()* can be replaced by a single call *sendWriteandRun()*, which is preferred because firstly, it does away with the requirement of *sendRead()* to specify the size of the data to be read from output buffers, and secondly, it offers better performance, by avoiding some overhead latencies. This has also been observed by us in one of our sample applications involving GF2 multiplication, where the communication round-trip time halves when *sendWriteandRun()* is used instead of the four separate function calls. However, the four functions can help us for debugging and analysis purposes.

In the beginning, the software side has exclusive read/write access to the SIRC buffers. When it signals the hardware to execute once data transfer has been completed, the exclusive read/write access is transferred to the hardware circuit on the FPGA. Thus, the two sides of the API can never simultaneously access the register file and IO buffers of the SIRC interface. This may seem to indicate a lack of support for pipelined applications, that would ideally need both reading from input buffers and writing to output buffers at the same time. However, it is possible to declare and instantiate multiple instances of the SIRC API, and thus support overlapped IO and streaming applications.

In the default setup, the user logic circuit on the FPGA communicates with the SIRC memory and register file using a request/acknowledge protocol, which has been described in detail in the documentation provided in the installation directory. However, using this protocol requires the handling of a lot of interfacing signals by the user logic. This requires extra effort on the part of application developer to satisfy the timing/control constraints while reading/writing from memory. We have abstracted away this interface using parameterizable input and output FIFOs. Thus, the user circuit need only read from the input FIFO and write to the output FIFO, which is much simpler than supporting the interface protocol of the SIRC hardware API. The FIFOs take the width and depth as parameters and introduce a small overhead when filling in/writing out data from/to SIRC memory.

**Chapter 4**

# **Matrix Multiplication over GF(2)**

A Galois field or a finite field is, as the name suggests, a field with a finite number of elements. GF(2) is the smallest finite field with just two elements. The two elements are usually represented as 0 and 1. Finite fields are important because of their applications in areas such as information/coding theory, error correction, cryptography etc.

The arithmetic over finite fields differs from regular arithmetic. In particular, addition and multiplication over GF(2) is performed in accordance with the tables shown below:

|  |  |  |
| --- | --- | --- |
| **+** | 0 | 1 |
| 0 | 0 | 1 |
| 1 | 1 | 0 |

|  |  |  |
| --- | --- | --- |
| X | 0 | 1 |
| 0 | 0 | 0 |
| 1 | 0 | 1 |

It is easy to observe that addition over GF(2) is same as the logical XOR operation and multiplication over GF(2) is same as the logical AND operation. Thus Galois field operations are well suited for FPGA implementation since a large number of AND and XOR gates can be synthesized on the FPGA thus obtaining large amounts of parallelism.

Dense Matrix-matrix multiplication is an important primitive in computational linear algebra. The input matrices can be fairly large and so first they are divided into blocks of a fixed size (say 32x32). We first describe the scheme used to multiply 2 blocks (ie 2 32x32 matrices). After that we would use this block multiplication to compute the product of 2 large matrices. In order to multiply 2 blocks A and B, we look at the columns of A and rows of B. Let the ith column of A be denoted by ai and the ith row of B be denoted by bi. The product of the 2 matrices A and B is simply the sum of the outer products ai\*biT. Figure 3 pictorially shows the multiplication scheme. Note that in GF(2) addition is same as XOR, therefore, the xor operator is used. Also note that the outer product ai\*biT would be a matrix.

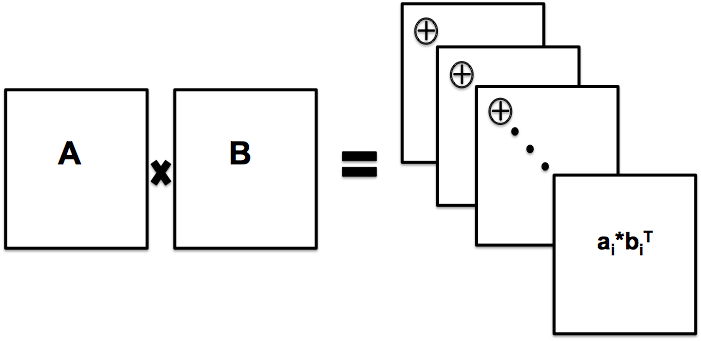
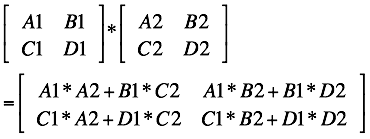


Figure 3: Multiplication of 2 GF(2) blocks

In order to compute the product of 2 large GF(2) matrices, we can simply regard each block as a number and use the usual rules of matrix multiplication to compute the matrix product. Here the product of the 2 numbers (which are actually blocks) is computed by the method described above. For example,



where A1, B1, C1, D1 are the block matrices obtained from matrix A and A2, B2, C2, D2 are block matrices obtained from matrix B. Thus it is fairly straightforward to compute the matrix product of a large matrix if the matrix product of each block can be computed.

**Chapter 5**

# **Hardware Design of GF(2) Matrix-Matrix Product**

As was discussed in the previous chapter, to implement the matrix-matrix product over GF(2), we first need to implement the block matrix product. Once this is done, it is straightforward to compute the matrix-matrix product for large matrices. The following discussion assumes block size of 32x32. However, it should be noted that the discussion is equally valid for other block sizes. There is no hard constraint on what the block size should be.

To compute the product of two GF(2) 32x32 matrices, we use 2 building blocks -

1. 32x32 GF(2) Transpose block

2. Outer Product Accumulation block

For the sake of completeness, we briefly describe the functioning of these blocks. The details of the design can be seen in [10].

As was described in the previous chapter, the product of the 2 matrices A and B is simply the sum of the outer products ai\*biT where ai is the ith column of A and bi is the ith row of B. So to

compute the product we first take the transpose of A so that the rows and columns of A get swapped. This will help us later.

Figure 4 shows the block diagram of the GF(2) transpose block. As can be seen from the figure, the block is essentially a collection of shift registers. For the first 32 clock cycles the input (rows of A) is fed and a down shift is performed on each clock cycle. So at the end of 32 clock cycles the entire matrix A has been read into the shift registers. The output is immediately available after that and can be read in the next 32 clock cycles. In each cycle a left circular shift is performed. Thus it is clear that such a design would take 1 row of A as an input for each of the first 32 clock cycles and output the columns of A in the next 32 clock cycles. Thus we have computed the transpose of the matrix. (It should be noted that we store each matrix block as an array of 32 bit words. So each entry of the array represents a row of the block. Thus all the entries of each row are accessible together but all entries of each column are not accessible together. Therefore the need to create the transpose of the matrix)

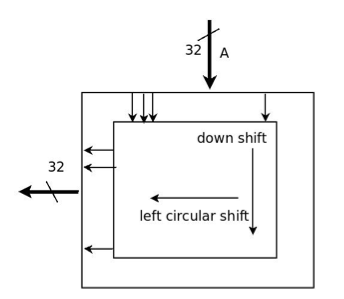


Figure 4: 32 × 32 GF(2) Transpose

Once we have the columns of A (from rows of AT) we have to compute the outer product using a column of A (ie row of AT) and a row of B. This product is simply the AND operator. Once the outer product is computed, it needs to be accumulated. Accumulation here simply involves taking XOR of each entry. Figure 5 shows the block diagram of the outer product computation and accumulation block. This block reads the inputs ( a row of AT and a row of B) in each clock cycle for the first 32 clock cycles and keeps accumulating the results. In the next 32 clock cycles the output (result of the block matrix product) can be read.

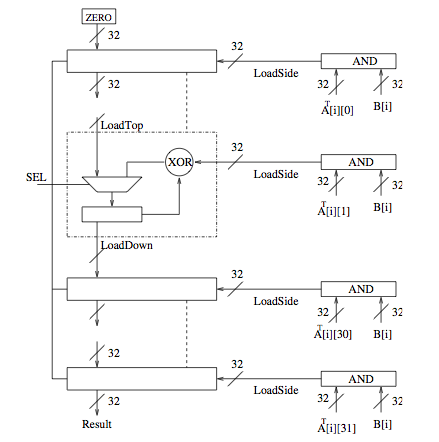


Figure 5: Outer Product Accumulation

Now we describe the 2 designs for large dense matrix matrix product over GF(2) which we have implemented during the course of this project. Both use the basic building blocks described above to accomplish the job.

**DESIGN A:**

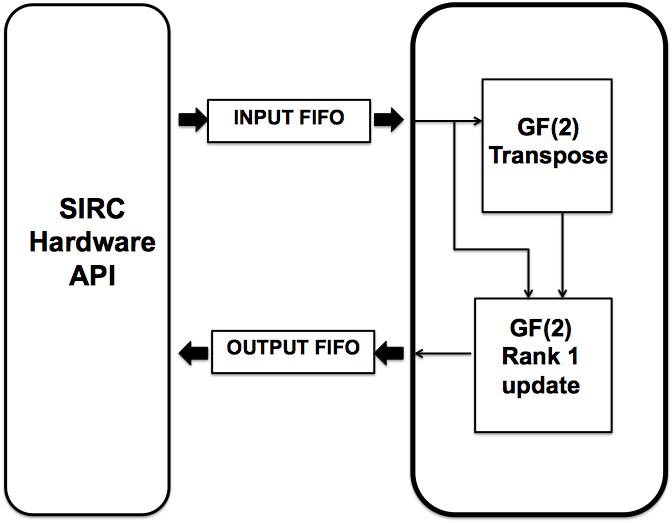


Figure 6: Block Diagram of the hardware in Design A

In this design, the hardware consists of just the 2 blocks - GF(2) Transpose and GF(2) rank 1 update (same as outer product accumulation which was described previously). Input and output FIFOs are created as described in chapter 3 to feed in the multiplier blocks with a new input each clock cycle.

The operation proceeds as follows:

1. The user c++ code sends the input data (block A and block B) to the FPGA over ethernet by calling the function sendWriteAndRun in the c++ code

2. The data is stored in the input memory on the FPGA defined in the SIRC part of the hardware and the userRun signal is asserted

3. The input FIFOs are now filled with the data from the input memory (Note that each word in the input FIFO is 32 bits wide and represents a row of the block)

4. For the next 32 clock cycles, the GF(2) transpose block reads the 1st 32 words of the input FIFO (This is basically the block A)

5. In the next 32 clock cycles the GF(2) transpose block gives the output AT. This is read into the GF(2) rank1 update block. Simultaneously the GF(2) rank1 update block also reads the next 32 words from the input fifo (this is basically the block B)

6. The next 32 clock cycles are spent in writing the output data from the GF(2) rank1 update block to the output FIFO.

7. Finally the data is copied from the output FIFO to the output memory in the SIRC part of the hardware and the userRunClear signal is asserted which signals SIRC to send the data back to the PC.

The above steps describe how the product of 2 matrix blocks is computed. In order to compute the product of two large matrices, we successively compute block products and keep accumulating the results appropriately. Thus the hardware on the FPGA is simply the one required to compute the product of 2 matrix blocks. The software is responsible for using this block to compute the product of 2 large matrices

The pseudo code for the software side is given below:

INPUT: 2 NxN matrices M1 and M2

OUTPUT: 1 NXN matrix M3

for i = 0 to N/32 do

for j =0 to N/32 do

for k = 0 to N/32 do

A = M1[i][k]

B = M2[k][j]

C = computeBlockProduct(A,B)

M3[i][j] = M3[i][j] ^ C

end

end

end

Note that in the above pseudo code, Mx[i][j] represents the (i,j)th 32x32 block of matrix Mx

**DESIGN B:**

Design A suffers from the disadvantage that it involves too many calls to the hardware block. In particular the block multiplier on the FPGA is called O(n3) times where n = N/32. On each call we waste time in the roundtrip time latency (=65usec) and the time to send the data over ethernet. This is highly unoptimal as we send each block of B n times incurring a communication cost each time. It turns out that this roundtrip latency is the bottleneck and so we use Design B which does much better.

In this design, in each iteration in the software we send a block of M1 (say A) and a row of blocks of M2 (say B1, B2 , … Bn) . The product of A with each of B1, B2 , … Bn is computed and the resultant matrices C1, … Cn are returned.

The pseudo code on the software side is as given below:

INPUT: 2 NxN matrices M1 and M2

OUTPUT: 1 NXN matrix M3

for i = 0 to N/32 do

for j =0 to N/32 do

A = M1[i][j]

B1 = M2[j][1]

B2 = M2[j][2]

.

Bn = M2[j][n]

C1,C2, ... Cn = computeBlockProduct(A,B1,B2, Bn)

M3[i][1] = M3[i][1] ^ C1

.

M3[i][n] = M3[i][n] ^ Cn

end

end

Note that in the above pseudo code, Mx[i][j] represents the (i,j)th 32x32 block of matrix Mx

The hardware part of design B is very similar to design A except that now we want to perform block multiplication for n pairs of blocks (A and B1, A and B2 , … A and Bn) instead of just 1 pair as in design A. This can be easily accomplished by extending the finite state machine to successively calculate the matrix products and store them onto the output FIFO. The details of the code would be discussed in Chapter 7.

Now it can be clearly seen that in this design the hardware block is called only O(n2) times (contrast with O(n3) times of design A). Of course now the hardware block takes longer to execute as compared to the hardware block in design A. However this time is much smaller than the rountrip latency and thus overall we get a large improvement. It turns out that the bottleneck in this design is now the ethernet speed.

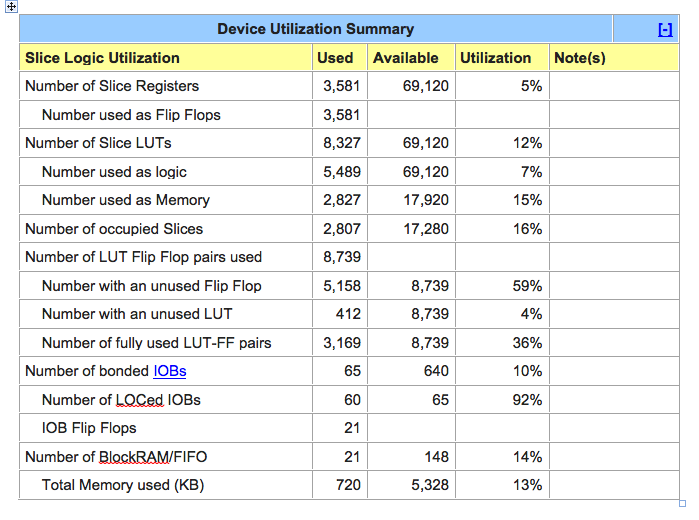
**RESULTS**

**Design A**

For multiplication of two 1024 x 1024 matrices, total execution time = 3.2 sec .

Note that the total execution time includes the time for writing the input data to SIRC buffers, filling input FIFO from memory, computation, writing results to output FIFO, filling output memory from output FIFO, reading in data from buffer over Ethernet in addition to the roundtrip latency of 65 us for every operation.

The time spent in the user hardware part is 203 cycles, which is equivalent to 1.2 us at a clock frequency of 167 MHz.



**Design B**

With block reuse, the total execution time is 0.33 sec.

Note that the total execution time includes the time for writing the input data to SIRC buffers, filling input FIFO from memory, computation, writing results to output FIFO, filling output memory from output FIFO, reading in data from buffer over Ethernet in addition to the roundtrip latency of 65 us for every operation.

**Discussion of the results:**

• Communication is the major bottleneck.

• At 450 Mbps, ~75 usec required for transfer of 33 blocks of 32 bit x 32 bit size. => for 2\*32\*32 such transfers, we would require ~153 msec.

• In addition to ethernet time, ~62usec required by SIRC protocol for each transfer. ~127 msec for 2\*32\*32 transfers.

• ~280msec out of ~330 msec spent in communication. Major bottleneck!

**Chapter 6**

# **Steps to run the project**

This chapter gives a step by step tutorial of how to run the GF2 matrix multiplication example as described in the previous chapter. This implementation follows the design B where the input is one block of matrix A followed by 32 blocks of matrix B, each block being 32\*32 in size.

A lot of the steps are the same as those outlined in the file README\_v1.1.pdf which outlines the method of creating and running a project [9]. We will continuously refer to this document while listing down the steps for running our project.

We have used a 64-bit Windows OS and 64 bit versions of Xilinx ISE Project Navigator, CORE Generator and IMPACT. The FPGA platform we use is the Xilinx XUPV5 board, which houses a Virtex 5 FPGA.

**Hardware Implementation**

1. Follow steps I and II as shown in the README file.

2. Right click on the system.v module in the “Design -> Hierarchy” window and click on ‘Remove Source’. We will replace this with our own system module.

3. Right click in the “Design -> Hierarchy” window and click on ‘Add source’. Navigate to the

location where the project files have been stored and select the system\_32\_word.v module

(GF2files\system\_32\_word.v). Once added, right click on the module and select ‘Set as Top

Module’.

4. In the same way, remove the module SimpleTestModuleOne.v and replace with

SimpleTestModuleOne\_largefifo\_ip\_and\_op.v

(GF2files\SimpleTestModuleOne\_largefifo\_ip\_and\_op.v).

5. Follow step III as in the README file.

6. Follow instructions h-k in step IV. In instruction l.ii.3, enter the read width according to the size of the matrix multiplier block that has been implemented, ie N in case of a NxN multiplier block. In our implementation, we have used a read width of 32 as the matrix blocks are of size 32x32 (thirty two 32-bit words), and hence set this field equal to 32. The user may use other values such as N = 64, 128, 256 if the corresponding multiplier blocks have been defined to take NxN blocks as input (to exploit parallelism with more number of PEs). Note that this field accepts a maximum value of 256 only.

7. Follow the remaining steps till step m. In step n.ii, set Memory Size Write Width = 32 and the

Memory Size Write Depth equal to one fourth times the Memory Size Read Depth. Complete

the remaining instructions till IV.o.

8. Now, we will add the source files for the 32x32 multiplier logic circuit. Right click on the

“Design -> Hierarchy” window and click on “Add source”. Select the files *gfm\_pac.v*,

*gfm\_mac\_pack.v*, *gfmm\_rank1up\_comp.v, gfmm\_rank1up\_ci.v* from the project folder

(GF2files\GFMMR1U). Similarly add the files *gfm\_transpose.v, gfm\_transpose\_comp.v,*

*gfm\_transpose\_ci.v* and *srxy.v* from the GFMT folder (GF2files\GFMT)

9. Next compile and synthesize the circuit as outlined in the README.

**Software Implementation**

Load the example software project (SIRC Installation Directory\SWSrc....) and replace the source file with the file provided in the folder GF2files (GF2fies\eth\_SW\_Example.cpp). Build and run this project.

**Chapter 7**

# **Documentation of Source Code**

In this chapter, we provide a walkthrough of the source code. For any application, the user needs to modify only three files, namely system.v (modified module contained in the file system\_32\_word.v of our project folder), SimpleTestModuleOne.v (modified module contained in the file SimpleTestModuleOne\_largefifo\_ip\_and\_op.v of our project folder) and eth\_SW\_Example.cpp (modified with the same name in our project folder).

In the following sections, we will highlight and explain only the relevant sections of code and indicate which parts need to be modified for different applications. The following documentation is supposed to supplement the explanation already provided in the source code.

**I.** **system\_32\_word.v**

**a. # module system parameters**

|  |
| --- |
| **parameter INMEM\_USER\_BYTE\_WIDTH = 4,**  **parameter OUTMEM\_USER\_BYTE\_WIDTH = 4,**  **parameter INMEM\_USER\_ADDRESS\_WIDTH = 11,**  **parameter OUTMEM\_USER\_ADDRESS\_WIDTH = 11,**  **parameter MAC\_ADDRESS = 48'hAAAAAAAAAAAA** |

The parameters IN\_MEM\_USER\_BYTE\_WIDTH and OUTMEM\_USER\_BYTE\_WIDTH represent the size of the words that are input to the user logic circuit.Currently, the value is 4 as we read/write 32 bit (byte width = 4) words. The parameters INMEM\_USER\_ADDRESS\_WIDTH and OUTMEM\_USER\_ADDRESS\_WIDTH represent the size of the address space for the input and output BRAM blocks that have been generated using CORE Generator. The value 11 indicates an address space of 211 = 2048 elements.

The MAC\_ADDRESS is used for identifying the hardware block by the software on CPU. The MAC address value will differ for different SIRC Hardware API instantiations.

These are the only modifications that need to be made system.v file.

**II.** **SimpleTestModuleOne\_largefifo\_ip\_and\_op.v**

**a. # module SimpleTestModuleOne parameters**

|  |
| --- |
| **parameter INMEM\_BYTE\_WIDTH = 4,**  **parameter OUTMEM\_BYTE\_WIDTH = 4,**  **parameter INMEM\_ADDRESS\_WIDTH = 11,**  **parameter OUTMEM\_ADDRESS\_WIDTH = 11** |

These parameters are the default values of the *SimpleTestModuleOne* module. Since this module is instantiated in *system* module, the proper parameters will override the default parameter values of this module.

**b. # module SimpleTestModuleOne body**

|  |
| --- |
| **localparam IDLE = 0;**  **localparam READING\_IN\_PARAMETERS = 1;**  **localparam RUN = 2;**  **localparam COMPUTE =3;**  **localparam COMPUTE2 = 4;**  **localparam FILLOUTPUTFIFO = 5;**  **localparam OUTPUTRESULT =6;**  **localparam DISPLAYCOUNT = 7;**  **reg [2:0] currState;**  **reg [4:0] blockCount;** |

The variable *currState* stores the current state of the FSM of the user logic after integration with SIRC interface. The various localparam constants are used to indicate the various state of the FSM. The variable *blockCount* is a counter for the number of block multiply operations that have been completed for each block of the matrix A.

|  |
| --- |
| **reg [12:0] lastPendingReads;**  **wire [12:0] currPendingReads;**  **wire [((INMEM\_BYTE\_WIDTH \* 8) - 1):0] inputFifoDataOut;**  **wire [((OUTMEM\_BYTE\_WIDTH \* 8) - 1):0] outputFifoDataOut;**  **wire inputFifoEmpty;**  **wire outputFifoEmpty;**  **wire [12:0] inputFifoCount;**  **wire [12:0] outputFifoCount;**  **wire infifoRead;**  **wire outfifoRead;**  **wire inFifoReadDataValid;**  **reg [31:0] countNumberOfCycles;** |

The variables *lastPendingReads* and *currPendingReads* denote the number of elements left to be read in the last and current cycle respectively. Variables *inputFifoDataOut* and *outputFifoDataOut* are the read ports of the input and output FIFOs respectively. Variables *inputFifoEmpty* and *outputFifoEmpty* indicate whether the FIFOs are empty. Variables *inputFifoCount* and *outputFifoCount* denote the number of elements currently in the input and output FIFOs. Wires *infifoRead* and *outfifoRead* act as switched to enable reading from the head of input and output FIFO respectively.

The variable *countNumberOfCycles* is a counter that has been introduced to increment every cycle of total runtime. This counter allows us to determine the total number of cycles elapsed for an entire process, it can also be used to determine the number of cycles spent in each state of the FSM.

|  |
| --- |
| **localparam N=32;**  **reg [N-1:0] a\_;**  **reg [N-1:0] a;**  **reg [2:0] nn;**  **wire [N-1:0] result;**  **reg clk\_en;**  **reg collect\_result;**  **reg start;**  **wire done;**  **reg [5:0] counter, counter\_;**  **wire WriteOutputFifo;**  **reg xyz;** |

These variables are to be connected to the ports of the GF2 Block Transpose module (GFMT). The register *a* is the input to the module while the result is available on *result*. The result from the module is available one clock cycle after the variable *collect\_result* goes high.

|  |
| --- |
| **reg [N-1:0] a1\_;**  **wire [N-1:0] a1;**  **reg [N-1:0] b1\_;**  **wire [N-1:0] b1;**  **reg [2:0] nn1;**  **wire [N-1:0] result1;**  **reg clk\_en1;**  **reg collect\_result1;**  **reg start1;**  **wire done1;**  **reg [5:0] counter1, counter1\_;**  **reg xyz1;** |

Similarly, these variables are meant to be connected to the ports of the GF2 rank 1 update module (GFMM). The result *result1* due to inputs *a1* and *b1* is available one clock cycle after *collect\_result1* goes high.

|  |
| --- |
| **RUN:**  **begin**  **countNumberOfCycles <= countNumberOfCycles + 1;**  **if((currPendingReads + inputFifoCount < 1056) && inputDone == 0) begin**  **inputMemoryReadReq <= 1;**  **end**  **else begin**  **inputMemoryReadReq <= 0;**  **end**    **if(inputMemoryReadReq == 1 && inputMemoryReadAck == 1 && inputMemoryReadAdd != lengthMinus1[(INMEM\_ADDRESS\_WIDTH - 1):0])**  **begin**  **inputMemoryReadAdd <= inputMemoryReadAdd + 1;**  **end**  **else if(inputMemoryReadReq == 1 && inputMemoryReadAck == 1 && inputMemoryReadAdd == lengthMinus1[(INMEM\_ADDRESS\_WIDTH - 1):0])**  **begin**  **inputDone <= 1;**  **end**    **if(inputMemoryReadReq == 0 && inputMemoryReadAdd == lengthMinus1[(INMEM\_ADDRESS\_WIDTH - 1):0])**  **begin**  **currState<=COMPUTE;**  **end**  **lastPendingReads <= currPendingReads;**  **end** |

This stage reads the data from the BRAM into the input FIFO. The value on the RHS in the condition : **currPendingReads + inputFifoCount < 1056**, has to be the value that is the maximum number of words that are sent as input. The FSM moves to the COMPUTE1 state after this.

|  |
| --- |
| **COMPUTE:**  **begin**  **countNumberOfCycles <= countNumberOfCycles + 1;**  **start<=1;**  **clk\_en <= 1;**  **if(start == 1)**  **begin**  **counter\_ <= counter\_ + 1;**  **end**    **if(counter == 31)**  **begin**  **nn <= 4;**  **collect\_result <= 1;**  **currState<=COMPUTE2;**  **nn1<=3;**  **end**    **counter <= counter\_;**  **a <= inputFifoDataOut;**  **end** |

This stage is responsible for computing the transpose of the 32x32 block. The *clk\_en* is set to high to enable operation of the matrix transpose block. The input *a* is fed the 32 bit words from the input FIFO. After 32 cycles, which is maintained by *counter*, the transpose block starts producing the words of the transpose matrix at its output, which are fed to the *a1* input of the rank 1 update block. *xyz* is set high one cycle after *collect\_result*, which indicates result of transpose is valid.

|  |
| --- |
| **COMPUTE2:**  **begin**  **countNumberOfCycles <= countNumberOfCycles + 1;**  **if(collect\_result==1) begin**  **xyz <= 1;**  **end**  **start1<=1;**  **clk\_en1 <= 1;**  **if(start1 == 1)**  **begin**  **counter1\_ <= counter1\_ + 1;**  **end**    **if(counter1 == 30)**  **begin**  **nn1 <= 4;**  **collect\_result1 <= 1;**  **currState<=FILLOUTPUTFIFO;**  **clk\_en <= 0;**  **end**    **counter1 <= counter1\_;**  **end** |

This stage is responsible for the rank 1 update. For 32 clock cycles which is counted by counter1, the module *gfmm\_rank1up\_ci* receives the output of *gfmm\_transpose\_ci* at its *a1* input and the *inputFifoDataRead* at its *b1* input. After the end of 32 cycles, the *collect\_result1* signal is pulled high, one cycle after which the *result1* output is valid. This delay of 1 cycle after *collect\_result1* goes high is ensured by *xyz1.*

|  |
| --- |
| **FILLOUTPUTFIFO :**  **begin**  **countNumberOfCycles <= countNumberOfCycles + 1;**  **if(collect\_result1==1) begin**  **xyz1 <= 1;**  **collect\_result1 <=0;**  **end**  **else if(outputFifoCount%32==0 && outputFifoCount>0)**  **begin**  **blockCount <= blockCount + 1;**  **if(blockCount==31) begin**  **currState <= OUTPUTRESULT;**  **outputMemoryWriteReq <= 0;**  **outputMemoryWriteAdd <= 0;**  **outputMemoryWriteData <= 0;**  **end**  **else**  **begin**  **currState <=COMPUTE2;**  **clk\_en <= 1;**  **counter1 <=0;**  **counter1\_ <=0;**  **nn1<=3;**  **xyz1<=0;**  **collect\_result1<=0;**  **end**  **end**  **end** |

This stage fills the output FIFO with the result of the *gfmm\_rank1up\_ci* block. Since the output FIFO is to be filled with N/32 blocks, where N is the total size of the input matrix row, the counter *blockCount* maintains the count of number of blocks that have been written. After each block that has been filled into the output FIFO, the state goes back to the COMPUTE2 stage where the rank 1 update takes in the next block from matrix B as input and produces the corresponding output. At the end of 32 such block updates, the operation is complete and the FSM moves to the OUTPUTRESULT state.

|  |
| --- |
| **OUTPUTRESULT : begin**  **…**  **if(outputMemoryWriteAdd == 11'b01111111111) begin**  **currState <= DISPLAYCOUNT;**  **end**  **…**  **end** |

The OUTPUTRESULT state is where we write the results from the output FIFO into the output memory. Once the complete FIFO has been written, (check by **outputMemoryWriteAdd == 11'b0111111111**), the FSM moves to DISPLAYCOUNT stage where we just write back the *countNumberofCycles* variable, which helps us to determine the total number of cycles taken.

|  |
| --- |
| **DISPLAYCOUNT : begin**  **outputMemoryWriteAdd <= 11'b10000000000;**  **…**  **end** |

Write the *countNumberofCycles* variable at the memory address just after the last word of the result. This stage also asserts the *userRunClear* signal which indicates to the API that computation is complete and result is ready to be read.

|  |
| --- |
| **assign outfifoRead = ((outputMemoryWriteReq == 0) || (outputMemoryWriteReq == 1 && outputMemoryWriteAck == 1)) && (outputFifoEmpty == 0) && (currState == OUTPUTRESULT);**  **assign infifoRead = ((counter\_ >= 6'd1)&&(counter\_ <= 6'd32)&&(currState == COMPUTE))||((xyz==1)&&(counter1\_ <= 6'd32)&&(currState == COMPUTE2));**  **assign WriteOutputFifo = (xyz1==1)&&(currState == FILLOUTPUTFIFO);**  **assign b1 = inputFifoDataOut;**  **assign a1 = result;** |

The signal *outfifoRead*, which is the read enable of the output FIFO is set high when currently there are no write requests to the output memory or a previous write request to memory has been acknowledged, in addition to the conditions that output FIFO is not empty and we are in the state OUTPUTRESULT. The signal *infifoRead* is asserted when the FSM is either in state COMPUTE1 or COMPUTE2, and there are words left to be read from the input FIFO. The signal *WriteOutputFifo* goes high one clock cycle after the signal *collect\_result1* is set, thus allowing data to be written into the output FIFO when result from the rank 1 update module is ready. Ports *a1* and *b1* are the input ports of rank 1 update module, which fetch their values from the transpose module and input FIFO respectively.

**# module instantiations**

|  |
| --- |
| **FIFO #(.WIDTH((INMEM\_BYTE\_WIDTH \* 8)),.LOGDEPTH(11))inFIFO(**  **…**  **FIFO #(.WIDTH((OUTMEM\_BYTE\_WIDTH \* 8)),.LOGDEPTH(11))outFIFO(**  **…**  **gfm\_transpose\_ci gfm\_transpose\_ci(**  **…**  **gfmm\_rank1up\_ci gfmm\_rank1up\_ci(**  **...** |

These are the instantiations of the modules for the user logic. Modules inFIFO and outFIFO are parametrizable FIFO queues. The depth in our example is 211 = 2056 and the entry width is IN/OUTMEM\_BYTE\_WIDTH \* 8 = 32 bits. The modules *gfm\_transpose\_ci* and *gfmm\_rank1up\_ci* are the 32x32 matrix transpose and rank 1 update modules respectively.

**III. eth\_SW\_Example**

eth\_SW\_Example.cpp (GF2files\software\eth\_SW\_Example.cpp) is the application software that runs on the CPU and makes the calls to SIRC API functions.

|  |
| --- |
| **int main(int argc, char\* argv[]){**  **//Test harness variables**  **ETH\_SIRC \*SIRC\_P;**  **uint8\_t FPGA\_ID[6];**  **bool FPGA\_ID\_DEF = false;**  **int a;**  **uint32\_t numOps = 0;**  **uint32\_t numOpsReturned;**  **uint32\_t tempInt;**  **uint32\_t artificialStopPoint;**  **uint32\_t driverVersion = 0;**  **int sizewrite=(N/32)\*(N/32);**  **double \*writeTime=new double[sizewrite];** |

We construct an instance SIRC\_P of the class ETH\_SIRC which represents the hardware block. The variable *numOps* indicates the number of words to be written/read to/from the SIRC buffers. *sizeWrite* is the number of total operations that need to be carried out. writeTime is a construct used for measuring the time spent by re in various sections of the code.

|  |
| --- |
| **//Input buffer**  **uint32\_t \*inputValues;**  **//Output buffer**  **uint32\_t \*outputValues;** |

*inputValues* and *outputValues* are the arrays which are to be written/read to/from the SIRC memory buffers respectively. They are defined to be of type uint32\_t as we want to transmit 32 bit words.

|  |
| --- |
| **uint32\_t\*\* matA = new uint32\_t\*[N];**  **uint32\_t\*\* matB = new uint32\_t\*[N];**  **uint32\_t\*\* matC = new uint32\_t\*[N];**  **uint32\_t\* countervalues = new uint32\_t[(N/32)\*(N/32)];**  **int s = 0;**  **for(int i = 0; i < N; ++i){**  **matA[i] = new uint32\_t[N/32];**  **matB[i] = new uint32\_t[N/32];**  **matC[i] = new uint32\_t[N/32];**  **}** |

Declare and allocate memory for the input matrices A and B and the output matrix C. The variable *countervalues* keeps a track of the number of cycles spent by the FPGA for every operation. It is d by the counter *countNumberofCycles* in the hardware code.

|  |
| --- |
| **cout << "Matrix A is \n\n" ;**  **for(int i=0;i<N;i++){**  **for(int j=0;j<N/32;j++){**  **matA[i][j] = 0x00000008;**  **matC[i][j] = 0x00000000; // initialize matrix C also**  **}**  **}**  **cout << "Matrix B is \n\n" ;**  **for(int i=0;i<N;i++){**  **for(int j=0;j<N/32;j++){**  **if(j==(i/32)) matB[i][j] = 0x80000000 >> (i%32);**  **else matB[i][j] = 0x00000000 ;**  **}**  **}** |

Initialize matrices A and B.

|  |
| --- |
| **params.maxInputDataBytes = 1<<13; //2\*\*17 128KBytes**  **params.maxOutputDataBytes = 1<<13; //2\*\*13 8KBytes** |

These reflect the buffer sizes in the hardware design.

|  |
| --- |
| **if(!SIRC\_P->sendWriteAndRun(0,4\*numOps,(uint8\_t \*)inputValues,4000,(uint8\_t \*) outputValues,4\*(numOps-32)+4, &outputLength)){** |

*sendWriteAndRun* is the command that combines write, run, wait and read commands. 4\*numOps is the number of bytes to be sent over Ethernet while writing. 4\*(numOps-32)+4 is the number of bytes to be read once the operation is complete. numOps = 33x32 words, ie that 32 words for each block transmitted and there are 33 blocks sent. Now we need to read back 32 blocks each of size 32 words, this multiplied by 4 gives the number of bytes to be read. 4 bytes are for reading the variable *countNumberofCycles* into the array *countervalues*.

**Chapter 8**

# **Conclusion and Future Work**

In conclusion, the work done in the course of this project comprised of the following:

* Literature survey on recent trends in High Performance Computing and hybrid CPU-FPGA computing. In particular, we looked at several papers using FPGAs/GPUs for hardware acceleration in a variety of applications to better understand the key application characteristics which make performance enhancement by using hardware-software co-processing possible. The major bottlenecks in using FPGA-CPU based co-processing for general purpose hardware acceleration were identified.
* Study and implementation of algorithms on FPGA using high-level synthesis tools such as C-to-Verilog [7] and SPIRAL [8]. A high level synthesis tool C-to-verilog was used to generate synthesizable verilog code for a discrete wavelet transform application. The code was synthesized and verified for correctness. Further Discrete Fourier Transform was synthesized and verified by using verilog code generated using SPIRAL.
* Study and implementation of GF(2) matrix-matrix multiplication using SIRC for hybrid CPU-FPGA application. An entire system has been built and tested which can take 2 input matrices, partition them into blocks of size 32x32 and then use block multiplication on the FPGA to compute the matrix-matrix product and return the result which is accumulated on the host PC.
* The codes implemented during the course of this project can serve as templates for a variety of application to implement in future. The report has been written in a tutorial fashion with detailed walkthroughs of important parts of the code so as to speed up future development using these tools in the lab.

**8.1 Future Work**

* The current design uses blocks of size 32x32. Future work should implement the same design with larger block sizes (say 128x128). This would increase the amount of parallelism which is being exploited and thus give better speedups.
* Further exploit parallelism to implement several 32\*32 matrix- matrix multiplication blocks. The current design uses only one 32x32 block multiplication hardware. It would be interesting to look at how multiple such units can be utilized to extract better performance.
* Explore the use of a data streaming type architecture with SIRC. For example two sets of hardware interfaces can be created and each is associated with a software interface (running on separate threads). This can help to overlap the I/O with the execution time. When one hardware interface is being used to write data into the FPGA from the PC, the other one can be used to write the output (coming from the execution of the last input) from the FPGA to the PC. This is not possible with just one hardware interface.
* Explore the benefits of hardware acceleration for other applications and domains:

–Sparse Matrix-Vector Product. Gaussian elimination

–LU decomposition/ Cholesky decomposition

–Coding theory: RS/ LDPC/ Expander decoding

–Cryptography: DES, Rijndael etc

–Signal and Image Processing: Wavelet transform, filtering etc

–Algorithms: Monte Carlo, BlackScholes, PageRank etc

* More detailed study and comparison of performance with benchmarks.
* Build on a strong library foundation for FPGA-based accelerated computing of applications for programmability and reusability.

**8.2 Limitations**

* Communication over Gigabit Ethernet: too slow for data-intensive applications. Communication latency is the major bottleneck in the current design. One way forward would be to use a PCI Express based communication which will likely be released by SIRC in a future release.
* Cannot simultaneously read and write to FPGA. However as explained in the third point under future work, a streaming architecture can be built using two hardware interfaces which would allow simultaneous read and write.
* High programming costs for low level HDLs. The open source high level synthesis tools are not most optimum. This leads to huge development costs which can often be prohibitive. Further currently we use separate tool flows and programming for hardware and software. Need for a single programming model.
* Need better tools for programming and analysis of hybrid computing. Need to know where cycles are being spent!

**References**

[1] Che, Shuai, et al. "Accelerating compute-intensive applications with GPUs and FPGAs." *Application Specific Processors, 2008. SASP 2008. Symposium on*. IEEE, 2008.

[2] Asanovic, Krste, et al. *The landscape of parallel computing research: A view from Berkeley*. Vol. 2. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, 2006.

[3] Leong, Philip HW. "Recent trends in FPGA architectures and applications."*Electronic Design, Test and Applications, 2008. DELTA 2008. 4th IEEE International Symposium on*. IEEE, 2008.

[4] “FPGA coprocessing evolution: Sustained performance approaches peak performance,” Altera White Paper, 2009.

[5] Eguro, Ken. "SIRC: An extensible reconfigurable computing communication API." *Field-Programmable Custom Computing Machines (FCCM), 2010 18th IEEE Annual International Symposium on*. IEEE, 2010.

[6] Jacobsen, Matthew, Yoav Freund, and Ryan Kastner. "RIFFA: A reusable integration framework for fpga accelerators." *Field-Programmable Custom Computing Machines (FCCM), 2012 IEEE 20th Annual International Symposium on*. IEEE, 2012.

[7] Rotem, Nadav, et al. "C-to-Verilog." *Automating circuit design.[HTML](http://c-to-verilog. com)* (2010).

[8] Milder, Peter, et al. "Computer generation of hardware for linear digital signal processing transforms." *ACM Transactions on Design Automation of Electronic Systems (TODAES)* 17.2 (2012): 15.

[9] SIRC User Manual <http://www.drsparrows.com/wp-content/uploads/pdfs/SIRC_README_v1.0.1.pdf>

[10] Saxena, Prateek et al. “Hardware-Software Scalable Architectures for Gaussian Elimination over GF(2) and Higher Galois Fields”

[11] Canis, Andrew, et al. "LegUp: high-level synthesis for FPGA-based processor/accelerator systems." *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays*. ACM, 2011.

[12] Bertels, Koen, et al. "Hartes: hardware-software codesign for heterogeneous multicore platforms." *Micro, IEEE* 30.5 (2010): 88-97.

[13] Gregerson, Anthony, et al. "Advances in architectures and tools for FPGAs and their impact on the design of complex systems for particle physics."*Typical Workshop on Electronics for Particle Physics (TWEPP)*. 2009.

[14] Chatziparaskevas, Georgios, Andreas Brokalakis, and Ioannis Papaefstathiou. "An FPGA-based parallel processor for Black-Scholes option pricing using finite differences schemes." *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2012*. IEEE, 2012.

[15] Yang, Depeng, G. Peterson, and Husheng Li. "High performance reconfigurable computing for Cholesky decomposition." *Symposium on Application Accelerators in High Performance Computing (SAAHPC), Urbana*. 2009.

[16] Kapre, Nachiket, and André DeHon. "Accelerating spice model-evaluation using FPGAS." *Field Programmable Custom Computing Machines, 2009. FCCM'09. 17th IEEE Symposium on*. IEEE, 2009.

[17] Todman, Timothy J., et al. "Reconfigurable computing: architectures and design methods." *IEE Proceedings-Computers and Digital Techniques* 152.2 (2005): 193-207.

[18] Yang, Depeng, et al. "Performance comparison of cholesky decomposition on GPUs and FPGAs." *Proceedings of the Symposium Application Accelerators in High Performance Computing (SAAHPC'10)*. 2010.

[19] Cui, Lintao, Jing Chen, and Bryan Hu. "Acceleration of Multi-Agent Simulation based on FPGA."