Privacy Preserving Computations Accelerated using FPGA Overlays

A Dissertation Presented
by

Xin Fang

to

The Department of Electrical and Computer Engineering

in partial fulfillment of the requirements
for the degree of

Doctor of Philosophy

in

Computer Engineering

Northeastern University
Boston, Massachusetts

August 2017

To my family.

Contents

List of Figures

List of Tables

Acknowledgments

I want to express my appreciation to those who have supported me during the process of my Ph.D. study at Northeastern University. Ph.D. study is a marathon which requires the determination and endeavors to face the challenge in research and every life. Without the help and encouragement from them, I cannot think of finishing this journey while enjoying it. So I thank them at the beginning of this dissertation.

First and foremost, I want to thank my research advisor Prof. Miriam Leeser who I have been working with for the last six years. Several projects are successfully finished with the help of her, including open-source variable precision floating point library, side-channel analysis and countermeasure and privacy-preserving computation using FPGA overlays. Looking back, I am deeply thankful for all the time that Prof. Leeser spends on my projects. She is very knowledgeable in many areas and so helpful that she does not hold back any things to help me improve. She comes up with insightful ideas during every one-on-one meeting. She is always there to help me. Words are powerless to express my gratitude to her.

I am very thankful that Prof. Stratis Ioannidis is my co-advisor for this project. He is an expert on secure function evaluation and has made significant contributions to the garbled circuit platforms including GraphSC and applications in data mining. Exploring the area where nobody has been reached before is sometimes confusing and he is always there to help. He is invaluable to me and to this project and I can not imagine this project without him.

I want to thank Prof. Yunsi Fei for being my committee member. She is my co-advisor for side-channel analysis and countermeasure project for about two years. She is an expert on hardware security and I enjoy the timing working with her. Thank you for the guidance and support.

I also want to express my appreciation to Dr. Nina Taft who is a scientist at Google. She is an expert on secure function evaluation and gave me many valuable suggestions during my proposal which makes my work after the proposal easier.

I would like to thank Kartik Nayak and Xiao Wang from University of Maryland, College Park for discussing with me about the FlexSC platform for this project.

I want to say thank you to my colleagues and friends along the way for their encouragement and help: Chao Liu, Kai Huang, Ben Drozdenko, Mahsa Bayati, Leiming Yu, Saoni Mukherjee, Chao Luo, Zhen Hang Jiang, Pei Luo, Yu Han, Liwei Zhang and all my friends who are invaluable to me during the journey.

Most importantly, I want to say thank you to my beloved parents, my mother Lianhong Shan and my father Xianjin Fang. They not only raise me up with their unconditional love but also encourage and support me along the way. I am so lucky to have you as my parents. Thank you for everything.

Abstract of the Dissertation

Privacy Preserving Computations Accelerated using FPGA Overlays

by

Xin Fang
Doctor of Philosophy in Electrical and Computer Engineering
Northeastern University, August 2017
Dr. Miriam Leeser, Advisor

Secure Function Evaluation (SFE) has recently received considerable attention due to the massive collection and mining of personal data over the Internet, but remains impractical due to its large computational cost. Garbled Circuits (GC) is a protocol for implementing SFE which can evaluate any function that can be expressed as a Boolean circuit and obtain the result while keeping each party's input private. Recent advances have led to a surge of garbled circuits implementations and applications in software to secure evaluation of a variety of different tasks. Due to the high computational complexity in garbled circuits, these implementations are inefficient and therefore GC is not widely used, especially for large problems. This research investigates, implements and evaluates secure computation generation using a heterogeneous computing platform featuring FPGAs. Unlike traditional FPGA design, overlay architecture on FPGAs is adopted since the SFE problem is too large to map to a single FPGA. The system leverages hardware acceleration to tackle the scalability and efficiency challenges inherent in SFE. To that end, we designed and implemented a generic, reconfigurable architecture as a coarse-grained FPGA overlay. On the host side, tools include SFE problem generator, parser and automatic host code generation tool are provided. Compared with tailored approaches that are tied to the execution of a specific SFE structure, and require full reprogramming of an FPGA with each new execution, our design allows re-purposing an FPGA to evaluate different SFE tasks without the need for reprogramming, and fully explores the parallelism for any GC problem. Our system demonstrates significant speedup compared with existing software platforms.

Chapter 1

Introduction

1.1 Garbled Circuits

Garbled Circuits (GC) originate from Andrew Yao's paper [3] which introduced a two party Secure Function Evaluation (SFE) protocol. Through Yao's protocol, two parties can jointly evaluate a function over private inputs, learning only the final outcome of this computation. In particular, neither party learns anything about the other party's input, other than what can be inferred from the outcome of the evaluation. Thus the computation is accomplished without sacrificing privacy. Yao's GC protocol can be extended to multiple party SFE and attain the above properties for the secure evaluation of any function that can be represented as a Boolean circuit.

The statistical analysis of data pertaining to human subjects, e.g. through experimentation, survey, monitoring, etc, has a long history in academic disciplines including sociology and behavioral economics. Internet companies like Google, Amazon and Facebook routinely monitor and explore a broad array of behavioral information from their users. The massive collection of user data is of considerable business value to online companies for targeted advertising and personalized product recommendations [4]. It is also beneficial to society: detecting medical emergencies or the spread of disease [5], accessing political opinions [6], assessing terrorist threats [7], etc. On the other hand, the massive collection and analysis of behavioral data has given rise to significant privacy concerns. Privacy threats have been extensively documented by researchers [8, 9, 10, 11, 12, 13, 14, 15] as well as the popular press [4, 16], and have drawn the attention of consumer advocacy groups, legislative bodies, and the general public. Such concerns are only likely to further increase with the emergence of the "Internet of Things", as wearable devices and home automation sensors connected to the Internet proliferate.

Secure function evaluation is the solution. SFE will guarantee the privacy of sensitive data while still providing the functionality needed and are needed where data privacy is obligatory. The implementations of SFE include but are not limited to secure data mining, secure IoT systems, online auctions, financial transactions, genomic computation, etc.

Particularly in secure data mining, a series of recent research efforts [17, 18, 19, 20, 21, 22] have attempted to address the privacy problems through cryptographic means and through SFE. SFE allows one party to evaluate any desirable function over private data from multiple owners while revealing only the final results and obtaining no information about any data. SFE can thus enable, e.g., a data analyst, or a statistician to conduct a study of sensitive data without jeopardizing the privacy of the participants.

However, challenges exist. SFE over private data comes at a considerable additional computational cost compared with execution in the clear. Any algorithm to be executed in SFE, especially GC, needs to be highly efficient and scalable. The function to be evaluated is first converted to a binary circuit which is "garbled" in such a way that an evaluator of the circuit learns only the values of its output gates. Prior work has made positive steps in this direction, showing that a variety of important data mining algorithms [17, 18, 19] can be computed using Yao's garbled circuits in a parallel fashion. Execution of this circuit is subsequently parallelized, e.g., over threads [18] or across a cluster of machines [19]. Nevertheless, further speed-up is needed for garbled circuits.

1.1.1 An Example: Computing Average Blood Pressure

Here is the an example of evaluating a function derived from a real life problem. A researcher wants to find the average blood pressure within a group of people in a public health project. The key computation which needs each people's blood pressure reading is the sum operation. In order to use the data securely without letting the researcher knows the exact reading for each person, garbled circuits can be applied in this situation. Table 1.1 shows the timing information of doing a sum of 1000 32-bit values. Using Java in clear mode, running this function takes 12 us. However, using FlexSC, it takes 2 seconds, which is more than 1 million times slower than the clear mode. However, clear mode suffers from the privacy issue since each individual has to reveal their own data. Our system can accelerate the operation to 300 ms, which is much faster than using FlexSC, and brings such secure function evaluation into the realm of the possible.

Table 1.1: Garbled Circuit Time Information

Sum 1000 32-bit Value	Time Information
Clear Mode	12 us
FlexSC	2s
Our System	300 ms

1.2 Heterogeneous Reconfigurable Computing

The heterogeneous reconfigurable computing means using FPGAs as an accelerator inside traditional data center for speedup. This has been demonstrating to have better performance, energy efficiency, overall cost, compared with traditional multi-core CPUs or other heterogeneous computing platforms such as GPUs for same applications. FPGAs in the cloud is becoming a trend in both industry and academia [23, 24, 25]. However, there are lots of concerns and obstacles that hamper FPGAs in the data center being used for a broader domain of applications. Problems still remain such as what kind of operation is suitable for FPGA acceleration, how to manage the interface between FPGAs and the host, how to divide hardware and software for different problems, tools for mapping a problem onto hardware. In this research, we evaluate the performance of the system, exploit different hardware architecture of accelerating SFE on FPGAs for better performance, and improve host communication, synchronization and control.

We accelerate garbled circuit generation using an FPGA overlay architecture on a heterogeneous computing platform for garbled circuits. This approach provides a new hardware architecture for garbled circuit generation and tackles the performance bottleneck. The results of this research can also act as a guide for general heterogeneous computing platforms featuring FPGAs. An FPGA overlay architecture consists of a circuit design on FPGA fabric and the user circuit is mapped onto that overlay circuit. Since garbled circuits problem sizes are often too large to map onto FPGAs and the operations are limited by two garbled circuits cores (Garbled Circuit AND and XOR gates), the overlay architecture is perfect for mapping different SFE problems. Results show significant speedup compared with existing software platforms for various of garbled circuit problems.

1.3 Contributions

The goal of this research is to accelerate garbled circuit generation using an FPGA coarse-grained overlay architecture in a heterogeneous computing platform. We provide hardware architecture and software tools for any garbled circuit and still maintain small communication overhead between hardware and software. Specifically, the contributions of this dissertation are:

- We design and develop a heterogeneous reconfigurable computing platform for acceleration of garbled circuit generation. We provide a complete workflow to map any garbled circuit problem to the garbled circuit overlay cells on FPGAs. On the hardware side, unlike the tailored approaches that are tied to the execution of a specific SFE structure, which requires full reprogramming of an FPGA with each new execution, our design allows re-purposing an FPGA to evaluate different SFE tasks without the need for reprogramming, and fully explores the parallelism for any GC problem. Host side tools include SFE problem generator, parser and automatic tools for host code generation. Those tools will generate the host program based on the structure of a problem so that it can be accelerated to the most. We also provide analytical tools to show the different characteristics of a problem. Results demonstrate significant speedup compared with current realizations of GC. This is the first heterogeneous computing system using FPGAs for accelerating garbled circuit problems.

- For reconfigurable hardware, we design and implement this system for secure function evaluation using FPGA overlay architecture for the execution of arbitrary garbled circuit topology. This overlay architecture supports 1) the mapping of large problems in GC and 2) all kinds of GC problems. We solve the problem when the amount of computation required to evaluate a garbled circuit for an application is too large to fit in a single FPGA, due to the extremely high computation complexity. Tailored approaches that are tied to the execution of a specific SFE structure, and require full reprogramming of an FPGA with each new execution, cannot be applied efficiently to new types of SFE problems. In our system, FPGA is programmed only once for all garbled circuit problems. Wiring and instantiation are determined at execution time through writing to registers and memory by the host, and this allows us to support many different problems. This overlay architecture is scalable and it enables the users to avoid the long design and deploy time on FPGAs for new problems compared with traditional FPGA system. The overhead for re-purposing the FPGA is kept very low, which is simply transferring the initial data to device memory.

- Software tools include layer extractor, problem parser and host code generator. Eventually, the tools will generate host code and enable users who do not have specific knowledge of FPGAs in heterogeneous computing platform to accelerate any garbled circuit operation. We integrate our implementation with FlexSC [26] which uses ObliVM [27] as the backend for any garbled circuit operation. ObliVM is a privacy preserving computation framework that allows one or multiple organizations to perform secure data analysis without disclosing their private data. The following tools will analyze the problem and extract the parallelism within it which later can be mapped to our FPGA overlay architecture for best performance.

- We simulate the performance of the overlay architecture for different GC problems [28, 29] and later map the whole garbled circuit generation onto a heterogeneous computing platform featuring a Stratix V FPGA. We tackle different aspects of performance bottlenecks and alleviate them, such as the levels of parallelism using different numbers of FPGA overlay cells, optimize the host to FPGA communication pattern via PCIe, hybrid memory system, etc. We compare the performance of the improvement for various problems with different sizes in our system.

1.4 Remainder of the Dissertation

Chapter 2 covers background information including garbled circuit systems and garbled circuits generation and evaluation. It also explains the general heterogeneous computing platform featuring FPGAs, including ProceV board from Gidel. Related work is discussed in Section 2.4. Our design methodology is presented in Chapter 3. We will demonstrate the methodology of how we tackle the garbled circuit problem from simulation to heterogeneous reconfigurable system. We alleviate the bottleneck in the system to improve performance. Experiments and the corresponding results are in Section 4. The dissertation ends in Chapter 5 with the conclusion and future research directions.

Chapter 2

Background

In this chapter, we first introduce background on garbled circuits including the terminologies and communication protocols. Next is a brief description of the SHA-1 algorithm adopted by our implementation. The FPGA architecture, overlay architecture and heterogeneous computing platform using FPGAs are described the following background. This chapter ends with related work in the garbled circuit research area.

2.1 Garbled Circuits

Yao's protocol (a.k.a. *garbled circuits*) [30] is a generic cryptographic protocol for secure function evaluation using cryptographic primitives. It enables two mistrusting parties jointly evaluate a function over their own private inputs. The two parties is called the garbled and the evaluator. At the end of the computation, neither of them will know each other's input and only know the output of the function.

An SFE protocol should have the following characteristics. Validity is the first one. Given the same input, any method that realizes an SFE protocol should produce the same result as the insecure one. This guarantees the correct functionality for any problem under SFE protocol.

The second characteristic is privacy. An SFE protocol should prevent a party from learning about the input data from a different party. Notice that this definition of privacy does not include situations where other participants may guess the input based on the final output. For example, if the operation is a bitwise AND operation, and one party has an input of '1'. If the output is '1', then he/she can easily deduce that the other party also holds a '1' value. Otherwise, the output will not be '1'. The situation above does not affect the definition of privacy.

Last, for sharing the output among parties, there also should be a mechanism to distribute it, which is the characteristic of fairness. Semi-honest adversary model will guarantee that one cannot deny or fake the output to the other party (or parties) by modifying or declining to do so.

2.1.1 Garbled Circuits Overview

In the variant we study here (adapted from [31, 32, 18]), Yao's protocol runs between (a) a set of private input owners (e.g., Google's users), (b) an Evaluator, (e.g., a data analyst working for Google), that wishes to evaluate a function over the private inputs, and (c) a third party called the Garbler, that facilities and enables the secure computation. Formally, let n be the number of input owners, and let $x_i \in \{0,1\}^*$ denote the private input of individual i, $1 \leq i \leq n$, represented as a binary string. Finally, let $f : (\{0,1\}^*)^n \to \{0,1\}^*$ be the function that the Evaluator wishes to compute over the private data. The protocol satisfies the following property: at the conclusion of the protocol, the Evaluator learns *only* the value $f(x_1, x_2, \ldots, x_n)$ and nothing else about $x_1, \ldots, x_n$, while the Garbler learns nothing.

A critical assumption behind Yao's protocol is that the function f can be expressed as a *Boolean circuit*, and, more specifically, as a directed acyclic graph (DAG) of AND and XOR gates.[1] The structure of the circuit – and, thus, the function to be computed – is known to all participants: e.g. the circuit could be computing the sum or the maximum among all inputs x_i.

Overall, Yao's protocol consists of three phases:

1. **Garbling Phase.** During the garbling phase, the Garbler prepares (a) a set of encrypted (i.e., "garbled") truth tables for each binary gate in the circuit, as well as (b) a set of random strings, termed *keys*, one for each possible binary value in the string set representing the inputs. At the conclusion of this phase, the Garbler sends to the Evaluator the garbled truth tables; each such table is referred to as a "garbled gate", and all gates together constitute the "garbled circuit".

2. **Oblivious Transfer Phase.** Subsequently, the Evaluator, Garbler, and the input owners engage in a proxy oblivious transfer [33, 34, 31]. Through this, the Evaluator retrieves the input keys from the Garbler that correspond to the true input binary values held by the owners. Oblivious transfer ensures that, although the Evaluator learns the correct keys, the clear-text input values are never revealed to either the Garbler or the Evaluator.

[1] Recall that any Boolean circuit can be represented using only ANDs and XORs.

3. **Evaluation Phase.** Finally, the Evaluator uses these input keys to "evaluate" the gates of the circuit, effectively decrypting the garbled gates. The Evaluator has access to a copy of the circuit used by the Garbler; this is due to the fact that the function being computed is known to all parties. The Evaluator implements similar parallelism to the garbler; gates are decrypted in a breadth-first manner. Each such decryption reveals a new key that allows the Evaluator to ungarble/decrypt subsequent gates connected to it. Ungarbling the output gates reveals the value $f(x_1, \ldots, x_n)$.

The above three phases are illustrated in Fig. 2.1. The execution flow (as well as the opportunity for parallelism) is determined by the circuit representing function f. Both the "garbling" of the gates, that occurs at the Garbler, and the "ungarbling/evaluation", that occurs at the Evaluator, are computationally intensive tasks; these are precisely the operations that we propose to implement using FPGAs. For the sake of completeness, we describe the garbling and evaluation phases in more detail below. As the proxy oblivious transfer is not as computationally intensive as the other two phases and does involve hardware acceleration, we do not describe it in detail; we refer the interested reader to [33, 34, 31] for a formal description and implementation.

The Evaluator wishes to evaluate a function f, represented as a binary circuit of AND and XOR gates, over private user inputs $x_1, x_2, \ldots, x_n$. In Phase I, the Garbler "garbles" each gate of the circuit, outputting (a) a "garbled circuit", namely, the garbled representation of every gate in the circuit representing f, and (b) a set of keys, each corresponding to a possible value in the string representing the inputs $x_1, \ldots, x_n$. In Phase II, through proxy oblivious transfer, the Evaluator learns the keys corresponding to the true user inputs, while the Garbler learns nothing. In the final phase, the Evaluator uses the keys as input to the garbled circuit to evaluate the circuit, ungarbling the gates in breadth-first order. At the conclusion of Phase III, the Evaluator learns $f(x_1, \ldots, x_n)$.

2.1.2 Garbling Phase

We now describe how gates are garbled in Yao's protocol. As illustrated in Fig. 2.2, each binary gate in the DAG representing the circuit is associated with three wires: two input wires and one output wire. At the beginning of the garbling phase, the Garbler associates two random strings, $k^0_{w_i}$ and $k^1_{w_i}$, with each wire w_i in the circuit. Intuitively, each $k^b_{w_i}$ is an encoding of the bit-value $b \in \{0, 1\}$ that the wire w_i can take. For each gate g, with input wires (w_i, w_j) and output wire w_k, the Garbler computes the following four cyphertexts, one for each pair of values $b_i, b_j \in \{0, 1\}$:

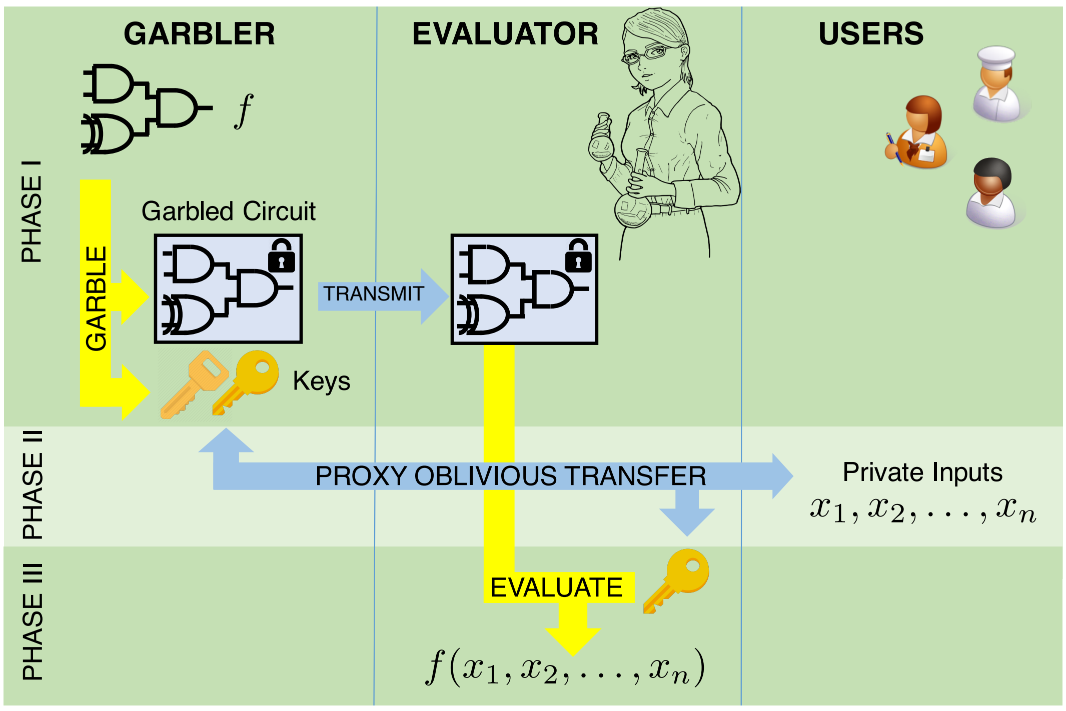


Figure 2.1: Yao's Protocol

$$Enc_{(k_{w_i}^{b_i},k_{w_j}^{b_j},g)}(k_{w_k}^{g(b_i,b_j)}) = SHA(k_{w_i}^{b_i} \| k_{w_j}^{b_j} \| g) \oplus k_{w_k}^{g(b_i,b_j)}, \tag{2.1}$$

where SHA represents the hash function, $\|$ indicates concatenation, g is an identifier for the gate, and $\oplus$ is the XOR operation. The "garbled" gate is then represented by a random permutation of these four ciphertexts. An example of a garbled AND gate is illustrated on Fig. 2.2. Observe that, given the pair of keys $(k_{w_i}^0, k_{w_j}^1)$ it is possible to successfully recover the key $k_{w_k}^1$ by decrypting $c = Enc_{(k_{w_i}^0,k_{w_j}^1,g)}(k_{w_k}^1)$ through[2]:

$$Dec_{(k_{w_i}^0,k_{w_j}^1,g)}(c) = SHA(k_{w_i}^{b_i} \| k_{w_j}^{b_j} \| g) \oplus c. \tag{2.2}$$

On the other hand, the other output wire key, namely $K_{w_k}^0$, cannot be recovered. More generally, it is worth noting that the knowledge of (a) the ciphertexts, and (b) keys $(k_{w_i}^{b_i}, k_{w_j}^{b_j})$ for some inputs b_i and b_j yields *only* the value of key $k_{w_k}^{g(b_i,b_j)}$; no other input or output keys of gate g can be recovered.

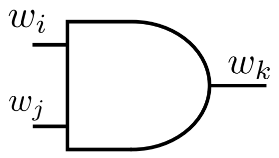
2.1.3 Evaluation Phase

Having described how gates are garbled, we turn our attention to how the garbled circuit is evaluated. The output of the garbling process is (a) the garbled gates, each comprising a random permutation of the four ciphertexts representing each gate, and (b) the keys $(k_{w_i}^0, k_{w_i}^1)$ for every wire w_i in the circuit. At the conclusion of the first phase, the Garbler sends all garbled gates to the Evaluator. It also provides the correspondence between the garbled value and the real bit-value for the circuit-output wires (the outcome of the computation): if w_k is a circuit-output wire, the pairs $(k_{w_k}^0, 0)$ and $(k_{w_k}^1, 1)$ are given to the Evaluator. Finally, the Garbler discards all random wire keys *except for* the keys corresponding to input wires (i.e., wires at the first layer of the circuit).

To transfer the garbled values of the input wires, the Garbler engages in a proxy oblivious transfer with the Evaluator and the users, so that the Evaluator obliviously obtains the garbled-circuit input value keys $k_{w_i}^b$ corresponding to the actual bit b of input wire w_i. Proxy OT ensures that (a) the Garbler does not learn the user inputs and that (b) the Evaluator can only compute the function on these inputs alone.

Having the garbled inputs, the Evaluator can "evaluate" each gate, by decrypting each ciphertext of a gate in the first layer of the circuit by applying equation (2.2): only one of these

[2]Note that the above encryption scheme is *symmetric*, as Enc,Dec are the same function.



b_i	b_j	$g(b_i, b_j)$	Garbled value
0	0	0	$Enc_{(k^0_{w_i}, k^0_{w_j}, g)}(k^0_{w_k})$
0	1	1	$Enc_{(k^0_{w_i}, k^1_{w_j}, g)}(k^1_{w_k})$
1	0	1	$Enc_{(k^1_{w_i}, k^0_{w_j}, g)}(k^1_{w_k})$
1	1	1	$Enc_{(k^1_{w_i}, k^1_{w_j}, g)}(k^1_{w_k})$

Figure 2.2: A Garbled AND Gate

decryptions will succeed[3], revealing the key corresponding to the output of this gate. Each output key revealed can subsequently be used to ungarble/evaluate any gate that uses it as an input. The evaluator can thus proceed ungarbling gates in breadth first order over the DAG until finally obtaining the keys of gates at the last layer of the circuit. Using the table mapping these keys to bits, the Evaluator can learn the final output.

We note that, in Yao's protocol, communication costs are dominated by the transfer of the garbled gates from the Garbler to the Evaluator. The oblivious transfer occurs only at the beginning of the evaluation phase to transfer inputs (the first layer of the circuit); as such, its communication cost is typically several orders of magnitude smaller than the Garbler to Evaluator transfer.

2.1.4 Optimization

Several improvements over the original Yao's protocol have been proposed recently, that lead to both computational and communication cost reductions. These include the point-and-permute [35], row reduction [31], and Free-XOR [36] optimizations, all of which we implement in our design. Free-XOR in particular significantly reduces the computational cost of garbled XOR gates: XOR gates do not need to be encrypted and decrypted, as the XOR output wire key is computed through an XOR of the corresponding input keys. In addition, the free-XOR optimization fully eliminates communication between the Garbler and the Evaluator for XORs: no ciphertexts need to be communicated between them for these gates. Our implementation takes advantage of all of these

[3]This can be detected, e.g., by appending a prefix of zeros to each key $k^b_{w_k}$, and checking if this prefix is present upon decryption.

optimizations; as a result, the circuit for computing AND gate, illustrated in Fig. 3.6, differs slightly from the AND gate garbling algorithm outlined above.

2.2 SHA-1 Algorithm

Secure Hash Algorithm 1, known as SHA-1 [37] is a cryptographic hash function published by National Institute of Standards and Technology (NIST). It has an output size of 160 bits known as the message digest. The block size is 512 bits, and 80 rounds are needed for one SHA-1 operation.

The applications of SHA-1 crypto operation include message authentication code, digital signature, etc, which ensures the data integrity for data sharing over the internet. The reason is that a hash function will ensure that the same input will always provide the same message digest; it is impossible to find two messages with the same digest and derive the message input from the digest; finally, a small change will generate a totally different digest which is known as the avalanche effect.

There are 80 iterations for SHA-1 operations. Each operation is the same and will update five variables representing the current state of the cipher. With different message input and constant round value, the digest will be unique for different message inputs.

There are two popular modes defined for block cipher operations. The simplest encryption mode is the Electronic Codebook (ECB) mode. The message is divided into several same size blocks, each of which is encrypted or hashed separately. Each message digest is independent of any other parts. The second mode is Cipher Block Chaining (CBC), where each block of message input is XORed with the output of a previously computed message digest before entering a new hash primitive. This is more secure compared with the ECB mode with the tradeoff of not being able to perform parallelization.

In our implementation of the garbled circuits protocol, the input bit width is less than the largest bit width allowed for one SHA-1 encryption. ECB mode with one hash primitive is suitable. There are several alternate computation algorithms that yield the same results. An implementation of the standard SHA-1 algorithm can be obtained from the open-course projects [38].

Here is a discussion of the security of the system using SHA-1. SHA-1 is not considered safe for attacks like brute-force attack, collision attack, and side channel attack. Collision attack means that attackers can forge two distinct documents with the same SHA-1 hash within a practical amount of time. Collision attack is becoming applicable as the computing power of computers increases every year observed by Moore's law. Side channel attack is a way of using side channel information while a cipher is running and later analyzing the secret information along with other known information. These attacks are very powerful and practical, and includes power traces, electrical magnetic signals, fault attacks, etc. Even the newest Hash algorithm Keccak (Standard SHA-3 Scheme) is also vulnerable [39, 40, 41, 42]. While SHA-1 itself is no longer considered secure, we claim that using SHA-1 is safe for garbled circuit because for each new operation, the system will generate new random variables for garbled value and keys which cannot be deducted by the previously observed information or any collision attack attempt. This makes any attack meaningless even it is successful.

2.3 Field-Programmable Gate Array

2.3.1 FPGA Architecture

Field Programmable Gate Array (FPGA) is a hardware architecture which can be configured after manufacturing. The application of FPGAs varies from ASIC prototyping, consumer electronics, signal processing, wireless communication, high-performance computing, aerospace and defense, medical devices, security, etc. Circuits can be described using Hardware Description Languages (HDL) or higher level languages using high-level synthesis. FPGAs are a good platform for application acceleration because of their flexibility, low development time and cost compared with Application Specific Integrated Circuits (ASICs) and low power consumption compared with Graphics Processing Units (GPUs) and CPUs.

The basic components of FPGAs are programmable logic blocks (Adaptive Logic Modules in Altera FPGAs and Configurable Logic Blocks in Xilinx FPGAs), massive programmable interconnections between those logic blocks, Digital Signal Processor (DSP) blocks, clocking, I/O, memory, etc. The LookUp Table (LUT) in programmable logic blocks contains memory cells to implement small logic functions.

For Altera which we target, each ALM supports up to eight inputs, contains four one bit registers, two dedicated full adders, a carry chain, a register chain and a 64-bit LUT mask. Fig. 2.3 shows the micro-architecture of one ALM in Stratix V family devices. Also, Stratix V devices contain two types of memories: enhanced Memory Logic Array Blocks (MLABs) with 640 bits and M20K memory blocks with 20K bits. MLABs are enhanced memory blocks that are configured from dual purpose Logic Array Blocks (LABs) and are optimized for implementation of shift registers for DSP applications, wide shallow FIFO buffers, and filter delay lines. M20Ks are the dedicated memory resources on Stratix V FPGAs. The M20K blocks are ideal for larger memory while providing a

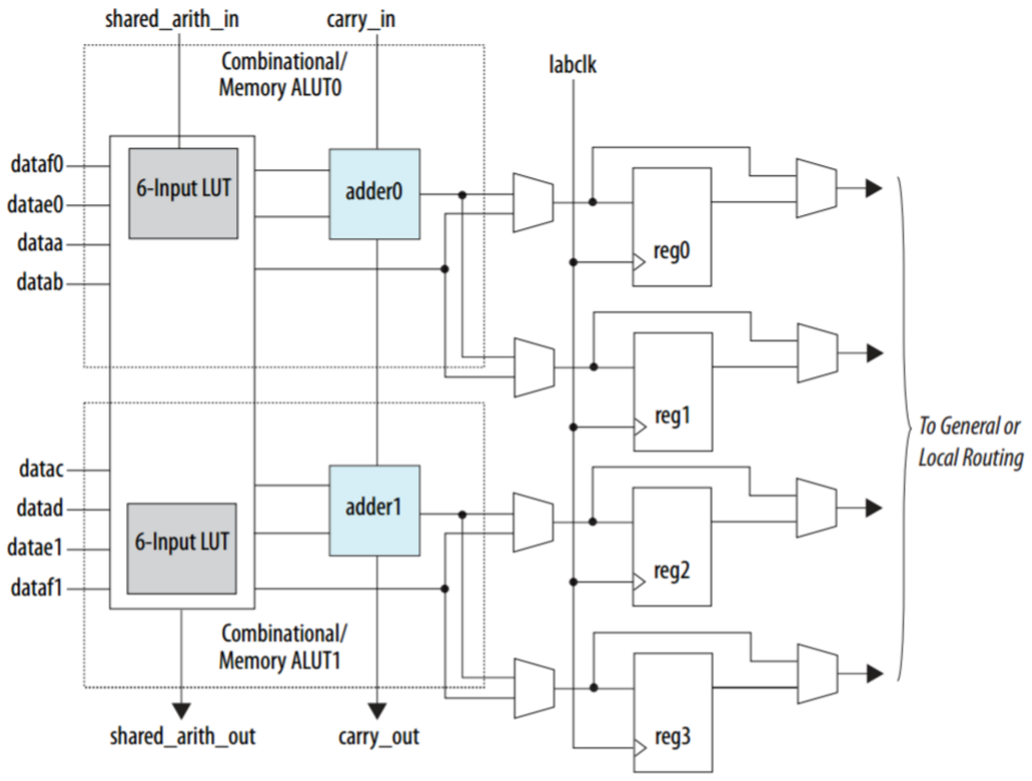


Figure 2.3: ALM Architecture for Stratix V Family FPGAs [1]

large number of independent ports and supporting more complicated memory modes than MLABs.

2.3.2 FPGA Overlays

As FPGAs have become denser and capable of holding a large number of gate equivalents, there has been an increased interest in FPGA overlay architectures [43, 44, 45, 46, 47, 48, 49]. An FPGA overlay consists of two parts: (1) a circuit design implemented on the FPGA fabric using the usual design flow, and (2) a user circuit mapped onto that overlay circuit. Fig. 2.4 is show the overlay architecture on FPGAs.

Overlays are in general used for two purposes. The first is to create FPGA designs that are independent of the specific structures on a particular FPGA and therefore to make designs portable, or, in other words, able to be mapped to FPGAs from different vendors and to different devices from the same vendor. This class of FPGA overlay designs [43, 44] creates basic FPGA structures, such as Look-Up Tables (LUTs) and routing, built on top of those provided in silicon on the target

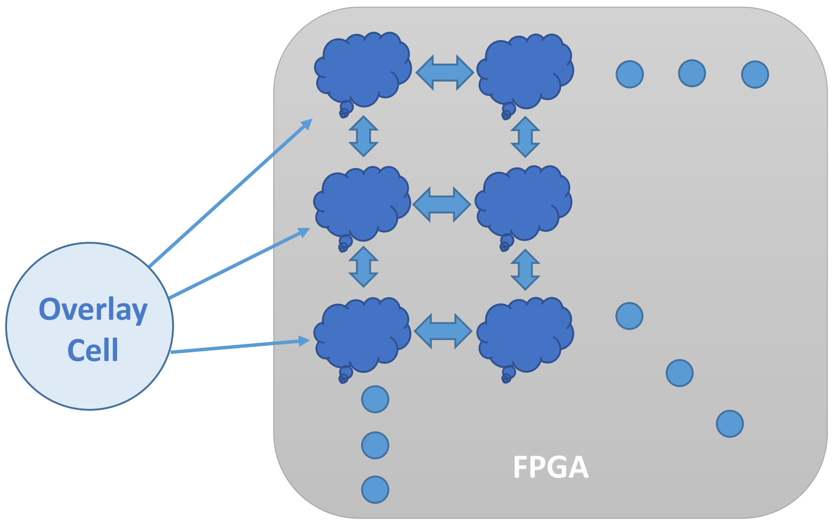


Figure 2.4: FPGA Overlay Architecture

FPGA chip. The second purpose is to reduce the amount of time to translate a design to an FPGA implementation.

Components of a circuit follow a generic structure, an overlay approach that does not reprogram FPGAs "from scratch", but simply *reroutes* connections between elementary components leads to important efficiency improvements. Building tradition architecture on FPGAs offer a great deal of reconfigurability and flexibility; however, this comes at the cost of programming. Generating designs that run efficiently on FPGAs can be challenging for the end user. In addition, the *compilation* process for a high-end FPGA design can take several hours. Here *compilation* refers to the complete set of steps from the specification (in a hardware description language (HDL) or high-level language) to generating a bit stream to download to the FPGA. These steps include synthesis, place-and-route and bit stream generation for the target FPGA. A big advantage of implementing the garbler (and evaluator) as an overlay architecture is that it eliminates the lengthy place and route times incurred when using an FPGA. Different pieces of the same problem, as well as different problems, can easily be mapped to the overlay without incurring this expense. Examples of this style of FPGA overlay architecture include Network on a Chip (NoC) overlays [45, 46] and instruction set extension overlays [47]. In these cases, structures are built on the FPGA and the overlay architecture provides flexible routing among them. Overlay architecture enables us to get rid of those traditional FPGA deployment steps for every new problem. Instead, using overlay architecture, the overhead is simply loading input to the memory on FPGAs and providing different control signals to feed the overlay cells. In chapter 3, we discuss the feasibility of mapping GC using the second type of FPGA overlays.

2.3.3 Heterogeneous Computing Platform using FPGAs

There are emerging heterogeneous computing platform using FPGAs in recent years. Here is an introduction of those platforms. Later we will show the ProceV Board which we use in this project.

FPGA as an accelerator in computer clusters can be traced back to the beginning of this century. In October 2004, the Cray XD1 range incorporated Xilinx Virtex-II Pro FPGAs for application acceleration. With 12 CPUs in a chassis, and up to 12 chassis installable in a rack, XD1 systems can hold multiple 144-CPUs in multirack configurations. The operating system used on the XD1 is a customized version of Linux, and the machine's load balancing and resource management system is an enhanced version of Sun Microsystems.

Later some popular platforms includes QP[50] and Novo-G[51]. QP (for "Quadro Plex") is developed at National Center for Supercomputing Applications at University of Illinois Urbana-Champaign. It includes 16 AMD CPUs and two types of accelerators: FPGAs from Xilinx and GPUs from NVIDIA. They are connected via InfiniBand and Ethernet and this system is very flexible since there are both FPGAs and GPUs. The Novo-G board is a reconfigurable computing cluster based at the University of Florida. The system features 96 Altera Stratix-V FPGAs in 24 networked servers. It is based on PCI Express FPGA cards by Gidel which we will discuss on 2.3.4.

Recent years saw the emergence of FPGAs in large heterogeneous computing platform. Internet companies including Microsoft, Amazon, Baidu, etc are involved in it. Microsoft Catapult project[23, 24] demonstrates an evolving system from one FPGA to multiple FPGAs per node for accelerating Bing searches and streaming data processing, etc. Amazon Web Services (AWS) cloud EC2 F1 server[25] let users run customizable FPGA hardware for their needs, featuring two different instance types. Under the AWS cloud infrastructure, users will have the necessary tools as a Hardware Development Kit (HDK) for better portability and user's designs will be registered as Amazon FPGA Images (AFI) for future reuse.

2.3.4 ProceV Board

We use the ProceV board from Gidel as our platform. This board is also widely used in Novo-G cluster. Fig. 2.5 shows the components of the ProceV board. It is a Stratix V FPGA-based platform along with two DDR3 external memories each of which can support 8GB. It provides high-speed communication between host and FPGA via a PCIe generation 3 bus which makes the system suitable for high-performance computing and low latency networking projects. There are 8 lanes on board each of which supports 8 Giga transmissions per second. Thus the throughput of this interface is 7.88 GB/s in each direction. The ProceV system enables high-bandwidth computation and networking, and unique flexibility to achieve diverse algorithm architectures. It is supported by Gidel's ProcWizard software and IP, which can shorten the development time. Altera's Stratix V FPGA provides high capacity and high speed for many architectures.

2.4 Related Work

Research on secure function evaluation using garbled circuits can be classified into three categories. First is the optimization of protocols and reduction of overhead at the algorithmic level. Researchers in this area focus on improving the algorithms of garbled circuits via cryptographic and mathematical deduction, reducing the size of the transmitted data, exploiting different variation of protocols for different adversary modules, etc. The second category is the development of platforms to implement garbled circuits, which includes algorithm libraries, custom compilers, and new languages. Garbled circuits remained purely theoretical until Fairplay [52] proved its feasibility in 2004. Ever since then, there have been more than a dozen platforms developed for garbled circuit applications. The focus of the third category is the accelerations of garbled circuits via specific hardware, including GPUs, FPGAs, and ASICs, etc.

2.4.1 Garbled Circuit Algorithm Research

Although garbled circuits were proposed nearly three decades ago [53, 30], it is only in the last few years that the research community has made progress in improving their efficiency, bringing their application closer to practicality. Several improvements over the original protocol have been proposed. Point-and-permute [35] is an optimization method to reduce evaluation time by adding a selection bit after ciphertexts, which ensures that only one rather than all four ciphertexts are needed for decryption. Row reduction [31] reduces the size of the transmitted garbled table

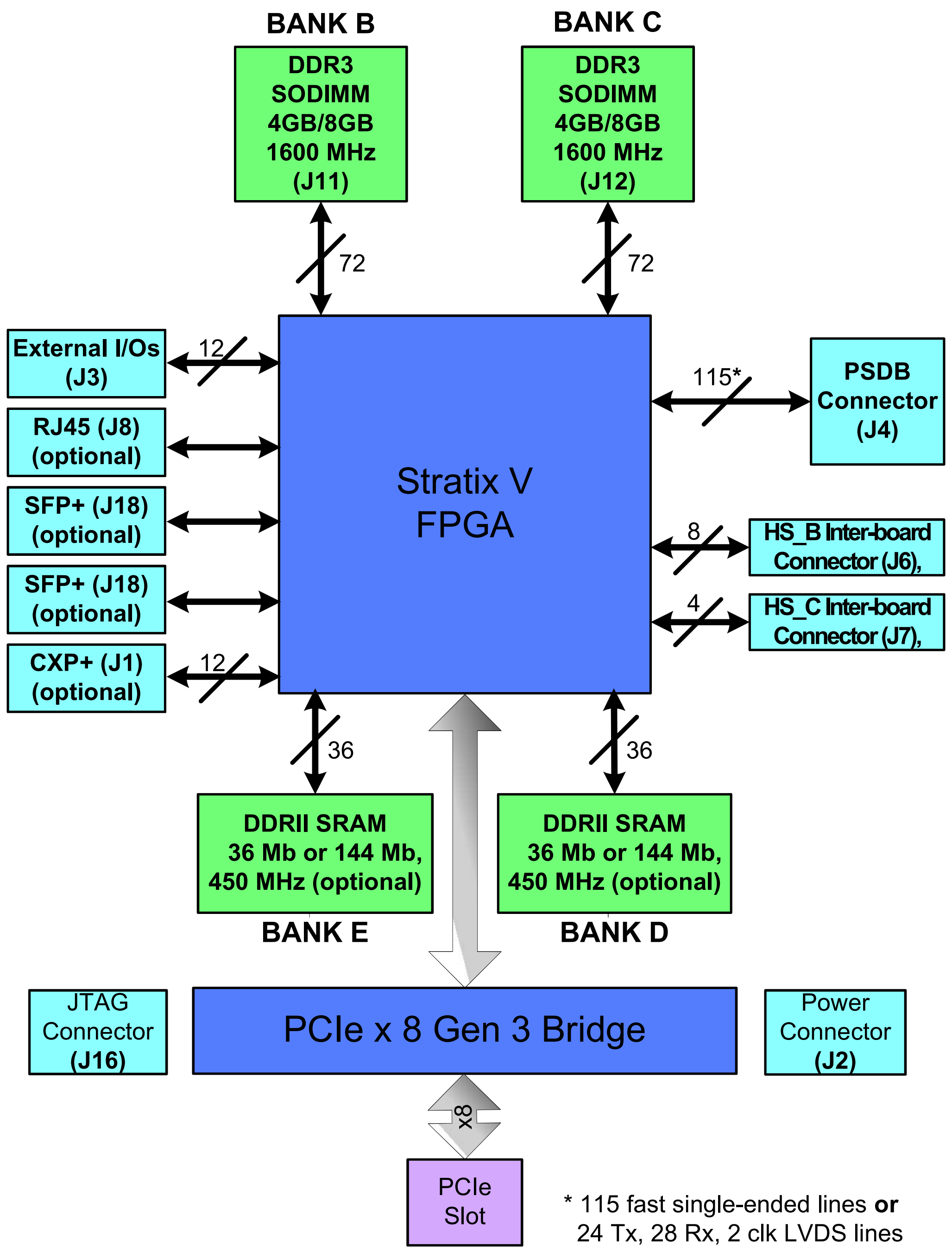

BANK B
DDR3 SODIMM 4GB/8GB 1600 MHz (J11)
BANK C
DDR3 SODIMM 4GB/8GB 1600 MHz (J12)
72
72
External I/Os (J3)
12
RJ45 (J8) (optional)
SFP+ (J18) (optional)
SFP+ (J18) (optional)
CXP+ (J1) (optional)
12
Stratix V FPGA
115*
PSDB Connector (J4)
8
HS_B Inter-board Connector (J6),
4
HS_C Inter-board Connector (J7),
36
36
DDRII SRAM 36 Mb or 144 Mb, 450 MHz (optional)
BANK E
DDRII SRAM 36 Mb or 144 Mb, 450 MHz (optional)
BANK D
JTAG Connector (J16)
PCIe x 8 Gen 3 Bridge
Power Connector (J2)
x8
PCIe Slot
* 115 fast single-ended lines or 24 Tx, 28 Rx, 2 clk LVDS lines

Figure 2.5: ProceV Block Diagram [2]

by 25% by enabling one of the garbled value to be always zero instead of generating it randomly. The Free-XOR [36] optimization is a big step toward the wide adoption of garbled circuits for real world problems, especially large ones. It proves that all the garbled circuit XOR operations are "free", meaning without the involvement of any encryption. Garbling with a fixed-key AES [54] will garble efficiently while still supporting row reduction and free-XOR. The two halves make a whole method [55] further reduces the number of cryptographic operations per AND gate to two, while still being compatible with free-XOR gates. The disadvantage is that for the evaluation step, two cryptographic operations are needed per AND gate. Clearly, research at this algorithmic level establishes the mathematical models via proofs, a cornerstone of garbled circuits.

2.4.2 Garbled Circuit Implementation

Based on the protocol improvements and optimizations at the algorithmic level, there has been a surge of programming frameworks for garbled circuits which can be divided into three categories.

The first is the garbled circuit arithmetic library for general purpose programming language such as Java. FastGC [56] is the first library-based garbled circuit platform introduced in 2011. Users can define the components in the library in an optimized way and also add new operations easily into the library. However, library-based approaches have no global optimization and require manual adjustment. Memory management also gets complicated for large problems. VMCRYPT [57] is a fully customized Java-based library which has a very small memory footprint, uses no disk storage and supports dynamically constructing and deconstructing circuits. It is also completely transparent to developers. The garbage collection overhead of VMCrypt caused by instantiating a new object for each gate is solved with less memory by [58]. ObliVM [27] is a privacy-preserving computation programming framework written in Java which can reduce the development process and effort, while still maintaining competitive performance for large problems like data mining, graph algorithms, and genomic data analysis. It has several different optimization levels of the garbled circuits algorithm, including the regular garbled circuits, offline and the half AND GC. ABY [59] is a mixed-protocol programming framework to evaluate operations using homomorphic encryption for arithmetic circuits (i.e. additions and multiplications) and garbled circuit operations for Boolean circuits. GraphSC [19] is a framework that supports graph-based algorithms including data mining and machine learning and also can execute in parallel with small communication overhead.

Second is the custom compiler for garbled circuits. Fairplay [52] is a compiler based platform from 2004 which can transform a high-level procedural language SFDL (Secure Function Definition Language) into a circuit description language SHDL (Secure Hardware Description Language). The generator and evaluator can then process the SHDL circuit with garbled circuits. TASTY [60] is a compiler for both homomorphic encryption and garbled circuits. It allows the user to automatically generate, benchmark and compare the performance with a semi-honest model. The first compiler with the scalability to billions of gates in malicious adversary mode is introduced in [61]. It can generate large circuits with fewer resources and can take advantage of the parallelism of the malicious model of Shelat and Shen's protocol [62]. PCF (Portable Circuit Format) [63] is a compiler that can optimize the program with the goal of producing a smaller circuit. A portable interpreter is responsible for loading PCF program and execution for different SFE security models. In this approach, it is not necessary to unroll loops until the protocol runs, leading to more compact representations which can scale to arbitrary circuit size. CBMC-GC [64] is the first ANSI C compiler for SFE which translates C programs as an input into equivalent Boolean circuits. Frigate [65] in 2016 is an efficient compiler and fast circuit interpreter for secure function evaluation with large improvement on compilation time, interpretation time and execution time.

The third is new languages specifically for garbled circuits. One example is SHDL of Fairplay [52]. However, it largely follows the standard of the C and Pascal programming languages. The L1 intermediate language [66] implements mixed protocol SFE. It supports secret shares, homomorphic encryption, and garbled circuits. WYSTERIA [67] supports mixed mode which can switch between normal and secure computation, but hiding lower level details and is not limited to two parties.

2.4.3 Garbled Circuit Acceleration

Acceleration of garbled circuits has become a hot research area in the SFE field. Researchers use different parallel models and hardware platforms for better speed up. These platforms include CPU/MIPS, GPUs, and FPGAs. This sections shows the current stages and the comparison.

Table 2.1 shows the comparison for different methods in the hardware acceleration area. The standards include parallelism exploitation, customized circuit or general circuit and the ease of translating a problem to a specific architecture. Compared with current state-of-art, our approach is the only one that has all three desirable properties.

Table 2.1: Comparison among Different Approaches

Property	ObliVM	GPUs	TinyGarble	MIPS	Standard FPGA	Ours
Parallel	N	Y	Y	N	Y	Y
Customized Circuits	N	N	Y	N	Y	Y
Fast SW Transition	Y	Y&N	N	Y	N	Y

2.4.3.1 CPU/MIPS

JustGarble [54] shows that using AES-NI (Advanced Encryption Standard New Instruction) can garble and evaluate a circuit much faster than the traditional software method. Intel AES-NI is a new encryption instruction set that improves AES operations in the Intel Xeon and Core processor family. TinyGarble [68] uses techniques from hardware design to implement GCs as sequential circuits and then optimizes these designs. The circuits can be optimized to reduce the non-XOR operations using traditional high-level synthesis tools and simulation. In the technology library, the area of an XOR gate is set to 0 while the area of non-XOR a non-zero value. The offline circuit synthesis will provide a ready-to-use circuit description for any garbled circuit platform. They also propose a 32-bit MIPS architecture specifically implemented to support for Private Function-SFE. However, there is only a proof-of-concept using the Hamming Distance example. GarbledCPU [69] is a MIPS-based general purpose sequential processor which enables the high-level description of garbled circuits in hardware. Problems to be evaluated securely are compiled to MIPS assembler and then run securely on their garbled MIPS processor. The goal of this project is to fabricate the MIPS core; FPGAs are used for prototyping the design. Using MIPS assembly code to represent the problem being evaluated alleviates the problem of lengthy FPGA place and route cycles.

2.4.3.2 GPUs

Recent studies have also used GPUs for hardware implementations of garbled circuits. Fastplay [70] has Fairplay as the frontend which provides SHDL as the input and uses a GPU architecture to accelerate ECC related arithmetic operations and achieve a 35 to 40 times acceleration over a serial implementation. [71] implements a protocol based on cut-and-choose of garbled circuits for malicious situation using GPUs. [72] implements free-XOR, pipeline, and OT extension on GPUs which exploit some of the embarrassingly parallel nature of many tasks. It shows the difference between implementations on Single Instruction Multiple Data (SIMD) architecture of GPUs and on Multiple Instruction Multiple Data (MIMD) architectures of multi-core CPUs.

2.4.3.3 FPGAs

[73] describes the first FPGA implementation of GC. Their implementation, while generic and able to support a wide range of hardware implementations, implements one encryption core and there is no parallelism. Two FPGA-based prototypes are proposed. A system-on-chip with access to a hardware cryptographic accelerator core and a stand-alone hardware implementation targeting ASICs. Another implementation involving FPGAs is the modified platform using MIPS discussed above which use FPGAs as a proof-of-concept. However, they use many fewer encryption cores and running code on a MIPS processor creates an extra level of overhead. The FPGA is not used as efficiently as in our proposed implementation. Our architecture uses much more parallelism than other FPGA implementations of garbling. For starters, we implement four SHA-1 cores in hardware for each AND and XOR gate, while others use one encryption core serially [73]. In addition, we implement as many garbled AND gates as we can keep busy at the same time, and implement garbled circuits directly on top of an efficient overlay. Details are provided in section 3.

Chapter 3

System Design Methodology

This chapter will introduce the methodology of designing garbled circuit generation on FPGAs from scratch. Our aim is to present an FPGA coarse-grained overlay architecture for a heterogeneous computing system along with corresponding software for general garbled circuit problems. We will show the tools to support mapping different kinds of garbled circuit problems onto the hardware architecture and leverage the interaction between hardware and software while maintaining small communication and memory access overhead.

This chapter contains three main parts: garbled circuit generation system, software structure, and hardware architecture.

Section 3.1 shows our garbled circuit generation system as a whole.

Section 3.2 discusses the software structure needed for working with the hardware architecture. Depending on different assumptions and characters of the hardware architecture, the software configurations for generating host code will be different. The system performance will also vary. Corresponding results with each proposed architecture will be presented in Chapter 4.

For the hardware architecture, we elaborate the way that we tackle this new challenge on FPGAs, which includes three sections: the design and simulation of garbled circuit generation in Section 3.3, garbled circuit hardware in the heterogeneous system design methodology using ProceV board in Section 3.4, and different levels of architecture improvement for best performance in Section 3.5. We first tackle this problem via simulation-based hardware design and later move to the heterogeneous system. For the real system, we show the different levels of bottlenecks which hamper the performance and give solutions. Different architecture exploration is conducted so that the best performance is achieved for general garbled circuit problems.

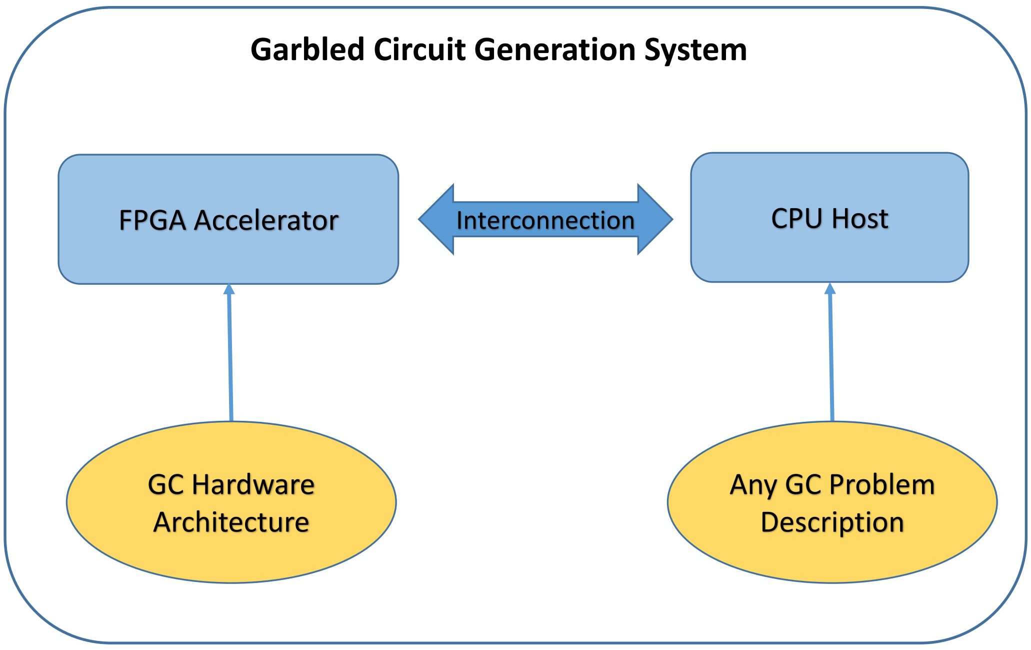


Figure 3.1: A Garbled Circuit Generation System

3.1 Garbled Circuit Generation System

Our FPGA acceleration of GC generation works as follows. We start with a function f the user wishes to evaluate securely and generate garbled circuit tables and garbled circuit values based on garbled circuit protocol. This is done by 1) translating the function to a Boolean circuit using an existing software platform and providing commands to the FPGA to garble the function, 2) accelerate the garbling by making use of an FPGA overlay architecture. We also improve the performance of the system by addressing different bottlenecks. We will show briefly how the system works as a whole and later describe each part of the hardware and software separately.

Fig. 3.1 shows the high-level structure of a heterogeneous system using FPGAs. In general, any heterogeneous system has the host side and the accelerator works in coordination with the host. There is an interconnection between the accelerator and the host.

The FPGA acts as the accelerator providing customized hardware circuits for a specific application. The benefit of using FPGAs can be time performance improvement, reduced total power consumption, increased energy efficiency, reduced system cost, etc. The bitstream for the hardware architecture will be programmed on to the FPGA from the host.

The host side of the heterogeneous system acts as the controller for the accelerator and works with the corresponding hardware architecture. It takes responsibility for configuration of the hardware, transmission of the initial data, start of operations, collecting the hardware results, etc. The low-level accelerator driver supplied for the FPGA, application interface, and operating systems make those operations possible.

The interconnection between the FPGA and host is the communication channel of all the data and control signals. There are several ways of implementing the interface including Intel QuickPath Interconnect (QPI), AXI bus, PCI Express, etc. Those interfaces aim to have low latency and memory coherency. For QPI and AXI bus, FPGA is usually in-socket, meaning the scope of the system is relatively small. This can reduce latency, have good memory coherency, cache coherency and shared memory modules. On the other hand, for the PCIe interface, the system can be on a large scale which makes the wide utilization of FPGAs as an accelerator in the data center possible. The latest Generation 5 of PCIe can provide up to 3.9GB/s throughput for a single lane. The lower communication delay cost of PCIe will make large HPC problems using FPGAs more widespread.

As for garbled circuits, there are some differences compared with the traditional workflow:

First, the computational complexity of garbled circuits is too high for mapping to any FPGA directly. For example, one simple garbled circuit AND gate takes about 3K Adaptive Logic Modules (ALM) on an Altera FPGA. If there are thousands of Boolean gates including AND gates even for a very small problem, the total resources needed will exceed well beyond the limit of any FPGAs off the shell. Even multiple FPGA solutions might not apply without dividing the problem into evenly smaller ones. Moreover, even if a solution might be feasible, the problem becomes the total redesign time for a specific problem of garbled circuits. For each new problem, the hours needed for designing a new hardware implementation will be very large.

Second, reducing unnecessary access between host and hardware can boost the total performance and the communication pattern of the host and the FPGA is not the same either. FPGA implementations such as video processing, wireless communication, has a distinct boundary between hardware and software operation with regards to time. This avoids frequent synchronization between hardware and software. For example, video processing is essentially streaming processing which deals with a flow of pixel data. While doing the operations, the host does not have to intervene until all the required data finishes processing. Once the start address of one frame is acquired by the hardware, the flow of data will be taken care of via buffers and the datapath for the specific functionality, like filter, edge detection, etc. Similar situations apply to the physical layer of the transceiver in wireless communications, where each stage such as preamble detection, modulation, demodulating can be applied either using hardware or software. The intermediate data can be buffered from/to host to/from FPGA, making the separation of operations viable.

In the case of garbled circuits, the challenges are the granularity of the operation and data access pattern. The granularity means that in order to support different kinds of garbled circuit problems, the hardware architecture has to be as general as possible. Later we will show that any structure of garbled circuit problem can be implemented with only two garbled gates, garbled circuit AND gate and garbled circuit XOR gate, which shows the possibility of realizing large problems using these two smaller garbled circuit Boolean operations. As a result, for this data access pattern, the host has to control the feeding of the initial data, intermediate address, the operation stages, etc. More importantly, the address of the intermediate data is not contiguous. It is a major drawback since the data can not be accessed as a stream of data. Neither PCIe nor DDR interfaces can be used for the best performance with respect to the throughput. Later discussion shows the latency of data transmission is also the bottleneck.

Besides the two differences above, there are also some factors which are the constraint of a specific heterogeneous computing platform, in our case ProceV board and the corresponding software development tools and device drivers. Some can be crucial to performance such as the clock frequency of the FPGAs, the data transmission clock frequency, the size and the method data can be shared between FPGA and the host, and the API for using specific operations through the device driver. Those can vary among different platforms which affect the final performance.

3.2 Software Structure

This section will discuss the software structure of the heterogeneous computing system for general garbled circuit problems. Fig. 3.2 shows the full software structure and workflow for generating the host side code for any garbled circuit problem. The following sections address problem generation, problem parser, layer extractor, and code generation tools.

3.2.1 Problem Generation

A garbled circuit problem is defined as a function to be evaluated using garbled circuit scheme to protect the privacy of input values. A batch of operations for hardware overlays is a group of Boolean operations within one layer and it can be mapped onto the hardware overlay architecture. The type of a batch can be either AND or XOR operation. The information of each

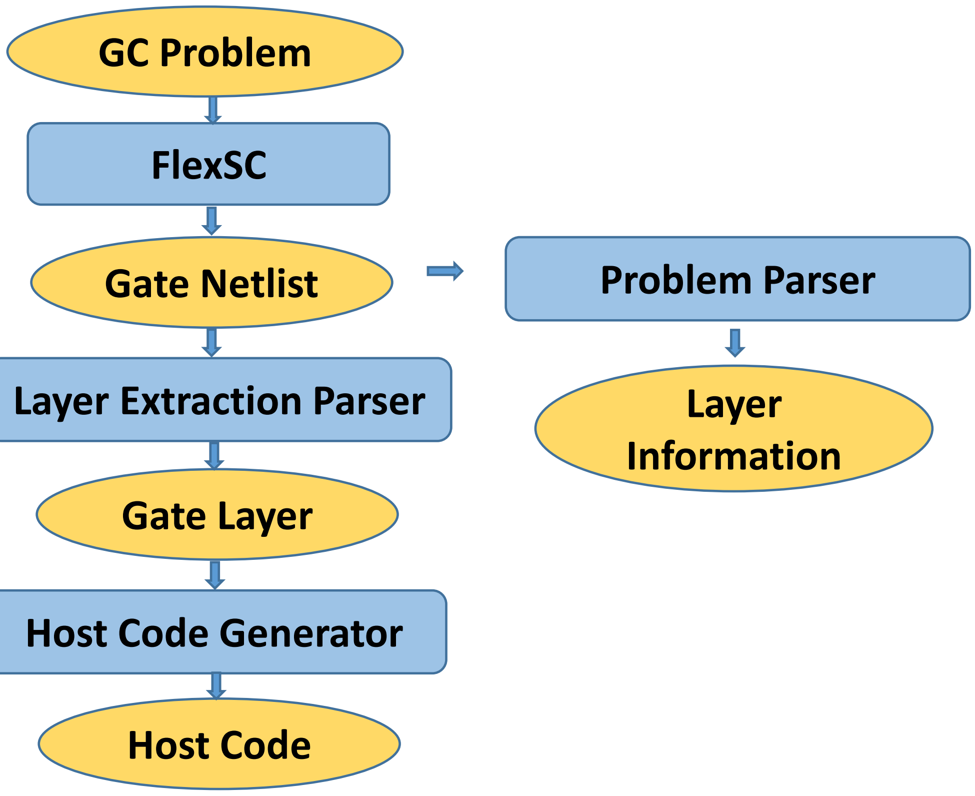


Figure 3.2: Software Workflow

batch of operations includes the input and output wire IDs for each Boolean operation, total number of Boolean operations in the batch and the AND gate ID for each AND operation. Each layer can be separated into several batches of operations, which makes garbled circuit using FPGA overlay cells possible for large problems.

We use FlexSC [26] as the software platform to generate the Boolean circuit representing any garbled circuit operation. Problems are easy to implement using FlexSC compare with other platforms such as FastGC, TinyGarble etc. FlexSC uses ObliVM as the backend for any garbled circuit operation. ObliVM is a privacy preserving computation framework that allows one or multiple organizations to perform secure data analysis without disclosing their private data. FlexSC provides an arithmetic library so that if users use the function within the library, the function will call the garbled circuit protocol backend realized inside ObliVM while running.

We build different types of garbled circuit problems based on the library from FlexSC. After modifying the platform of FlexSC, specifically within the ObliVM backend, we can get the garbled circuit gate list information with or without the value on each wire. We need the version with the value on each wire for verification purposes and the one without values on each wire for the Boolean operation of the problem. Here are the details.

First is garbled circuit gate list with value on each wire. This is helpful for generating the testbench so that while doing hardware design, the operation runs exactly the same as that of the software. Appendix A is an example of the generated garbled circuit values.

Second is the circuit generation mode to generate the structure of the circuit without the value of each wire. This mode generates the garbled circuit operations for later layer extractor and host code generation. The value on each wire is not important, since for every operation the input value will be a random sequence, and only the Boolean structure stays the same. Each Boolean gate is represented by the wire IDs and operation type. Here is an example of the millionaire's question. Suppose two millionaires want to compare their wealth without revealing the value of their wealth to each other. We assume that the wealth of each millionaire can be represented in 8 bits, then the problem becomes comparing two 8 bit values. Here is how the operation is represented using FlexSC. The geq function will return true if x is greater or equal than y and false otherwise. The *sub* and *not* functions inside are further implemented in other functions which are eventually realized by garbled AND and XOR operations.

```
public T geq(T[] x, T[] y) {
    assert (x.length == y.length) : "bad input";
    T[] result = sub(x, y);
    return not(result[result.length - 1]);
}
```

Here are the generated Boolean operations gate list from FlexSC. The form is wireID1 AND/XOR wireID2 = wireID3. The Boolean sequence is the same as the order performed in FlexSC.

```
8 XOR 17 = 18
9 XOR 17 = 19
10 XOR 17 = 20
11 XOR 17 = 21
12 XOR 17 = 22
13 XOR 17 = 23
14 XOR 17 = 24
15 XOR 17 = 25
0 XOR 26 = 27
18 XOR 26 = 28
0 XOR 28 = 29
27 AND 28 = 30
26 XOR 30 = 31
1 XOR 31 = 32
19 XOR 31 = 33
1 XOR 33 = 34
32 AND 33 = 35
31 XOR 35 = 36
2 XOR 36 = 37
20 XOR 36 = 38
2 XOR 38 = 39
37 AND 38 = 40
36 XOR 40 = 41
3 XOR 41 = 42
21 XOR 41 = 43
3 XOR 43 = 44
42 AND 43 = 45
41 XOR 45 = 46
4 XOR 46 = 47
22 XOR 46 = 48
4 XOR 48 = 49
47 AND 48 = 50
46 XOR 50 = 51
5 XOR 51 = 52
23 XOR 51 = 53
5 XOR 53 = 54
52 AND 53 = 55
51 XOR 55 = 56
6 XOR 56 = 57
```

```
24 XOR 56 = 58
6 XOR 58 = 59
57 AND 58 = 60
56 XOR 60 = 61
7 XOR 61 = 62
25 XOR 61 = 63
7 XOR 63 = 64
62 AND 63 = 65
61 XOR 65 = 66
64 XOR 17 = 67
```

The type for XOR is wireID_1 XOR wireID_2 = wireID_3; type for AND is wireID_1 AND wireID_2 = wireID_3. Notice that any problem can be expressed as a Boolean circuit can be represented using only Boolean AND and XOR gates. Note this is a small example. The number of gates within each problem can be hundreds of thousands.

3.2.2 Layer Extractor

After we get the complete gate list description for a problem, the list goes through a layer extractor. We process the gate list in breadth first order and the layer extractor organizes the Boolean gates for this procedure.

There are two levels of parallelism for garbled circuit problem, which is the parallelism in each garbled circuit AND and XOR operation and in each layer of Boolean operations. We represented Boolean circuit gate list as a Direct Acyclic Graph (DAG), and each layer of the circuit that can be garbled in parallel is extracted. The CPU is responsible for mapping individual gates to their realization on the FPGA. Our FPGA architecture implements as many AND and XOR operations in parallel as can be kept busy. The CPU can transmit the AND operations or XOR operations as a batch of Boolean operations in a layer, and assign them to overlay cell. When a layer is completed, the FPGA signals this information to the CPU which then transmits the next batch of the circuit, until the circuit has been fully garbled. More details of the FPGA architecture are given in Section 3.3 and Section 3.4.

We use Python to construct the layer scheduler which implements breadth first search on the graph and the output is the layer information. We also separate out the primary input values where the wire ID is not the output of any gate.

We use 8-bit millionaire's problem to demonstrate its functionality. Fig. 3.3 shows the visualization which provides the user a straightforward way of monitoring this function layer by

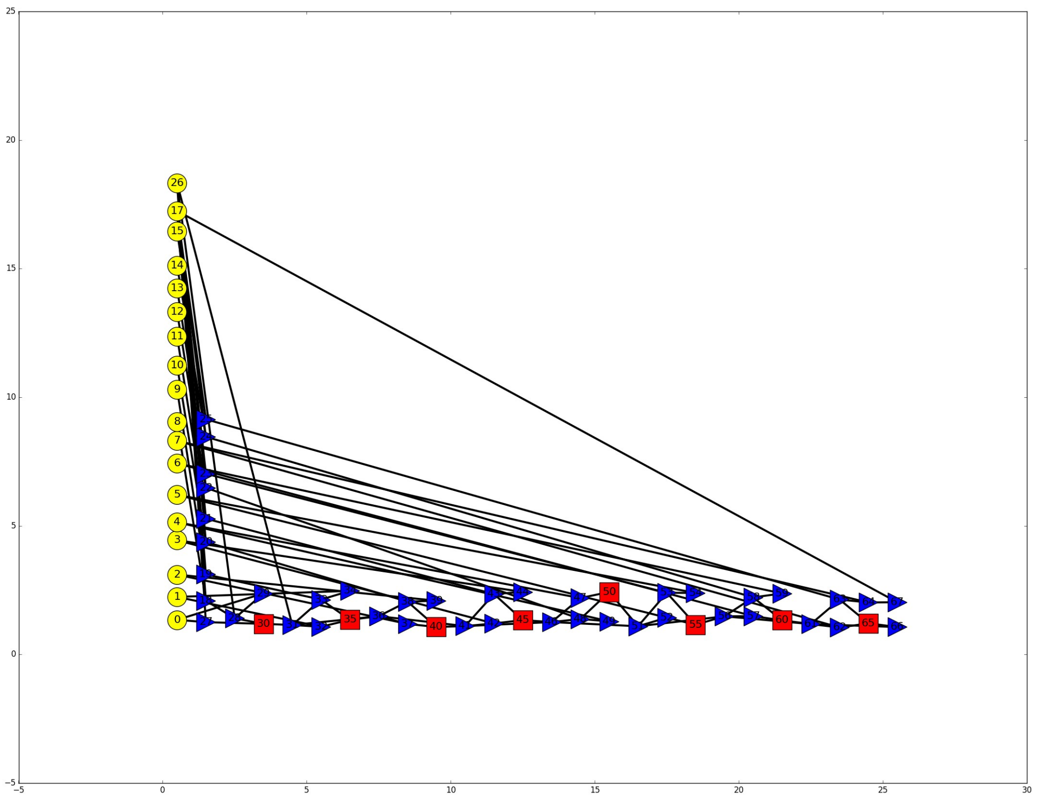


Figure 3.3: Layer Extractor Output Visualization

layer. X axis is the layer number, and the y-axis is the gate ID. The yellow circle represents the input wire ID; the red square represents the Boolean AND operation; the blue triangle represents the Boolean XOR operation. This figure demonstrates the dependency of each layer on the DAG. Based on garbled circuit AND and XOR operation, we can map the operation onto the generic architecture on FPGAs for any problem. Notice that this is a small problem, large problems have more gate within each layer. The layer extraction is a prepossessing step for each problem and only needs to be performed once per problem, independent of input values.

3.2.3 Problem Parser

As shown in Fig. 3.2, we also take gate netlist to problem parser for layer information. Problem parser analyzes gate, wire and layer information for each problem. Knowing the information will enable us to build the hardware architecture for garbled circuit problems. Fig. 3.4 shows the

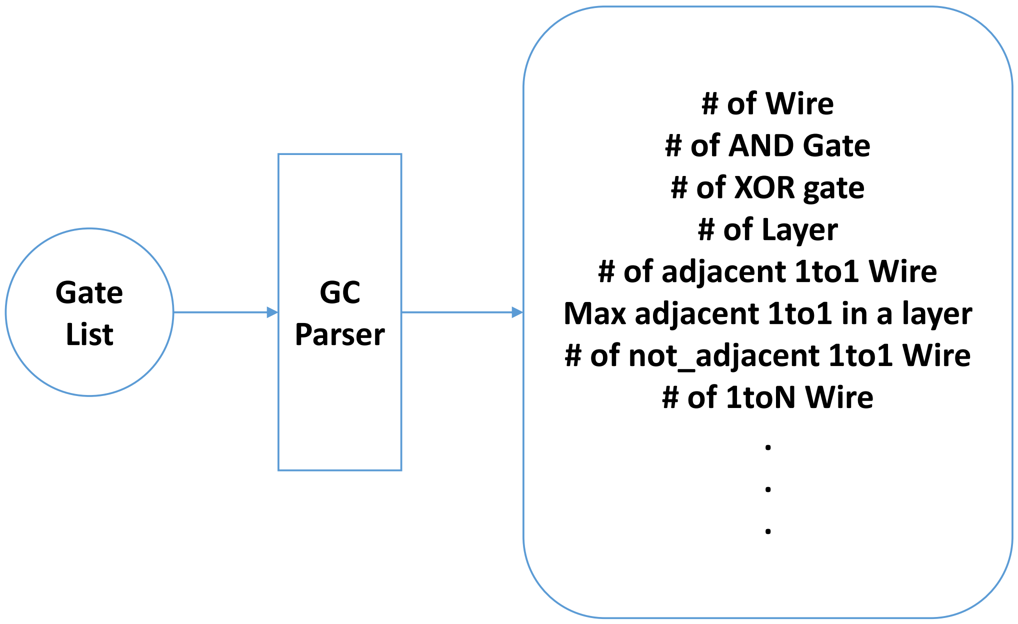


Figure 3.4: Garbled Circuit Problem Parser

structure of the problem parser.

For the output, some are general information: the total number of wires represent the total memory locations that we need for a specific problem; total AND gates and XOR gates show the amounts of those two operations; the number of layers reflects the level of dependency among all the Boolean operations.

Other output is information about separating wires into different groups. This is beneficial for different hardware memory hierarchy designs. For example, the adjacent 1-to-1 wire means a wire is only connected to one input after it was computed and also the connected gate is in the next layer. This gives us an opportunity to use local memory to accelerate storage and also reuse the location after the next layer is finished. Also, there are 1-to-1 wires but not in the adjacent layer and 1-to-n wires where one output is used multiple times.

3.2.4 Host Code Generation

For heterogeneous computing systems, we developed the tools to automatically generate the host code based on any garbled circuit operation. The input to it is the output of the layer extractor containing the gate list with layer information. Here is the functionality of code generation.

- This tool generates host code based on different hardware architectures obeying different acceleration policies. The host has to work with the hardware to provide the right functionality, such as the number of AND and XOR overlay cells on FPGAs and the memory systems. Translation of the address for each wire is based on the specific memory policy on the hardware.
- The host code contains the necessary initialization buffer for different problems and also comes with selection of debugging mode
- It rearranges the order within each layer and decides a good order to send AND and XOR operations to the FPGA.
- Separation main function into a group of smaller ones to avoid exceed heap size problem

Using this generator, users can general host code for any garbled circuit problem. The structure of host code is described in Appendix B.

3.3 Simulation of Garbled Circuit Generation

This is the first section of garbled circuit hardware architecture design methodology. We provide the computing unit for garbled circuit operation on FPGAs and make sure the functionality is correct. An overlay architecture is applied on FPGAs and we use on-chip Block RAM for data storage. We elaborate on each part of the hardware and also estimate of the speedup compared with software in Chapter 4.

3.3.1 FPGA Overlay Architecture

The complete design of the overlay architecture using Block RAM on FPGAs, shown in Fig. 3.5, includes garbled circuit XOR and AND overlay cells, Block RAM, a FIFO for communicating the garbled table and outputs with the CPU, and a workload dispatcher and data controller.

Garbled circuit AND and XOR overlay cells are the computational components which are mapped to FPGA fabric in a coarse grained manner. BRAM stores the initial, intermediate data and the final result in the form of a garbled table for each Boolean gate and garbled value for each wire. The wire ID is also used as the memory address. The workload dispatcher and data controller is

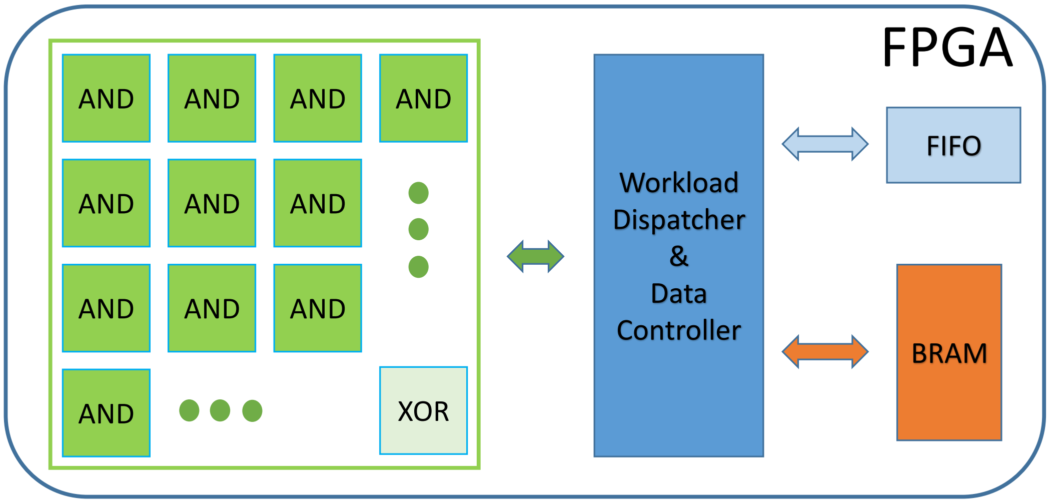


Figure 3.5: Overlay Architecture for Garbled Circuit

a finite state machine that controls the whole system. The FIFO enables communication from the FPGA to the host. The values transmitted include the garbled table for each AND operation and the garbled values of the outputs of the operation. The garbled table values are written to the FIFO when each AND garbling completes.

The following sections introduce the architecture of garbled circuit AND overlay cell, XOR overlay cell, embedded memory and workload dispatcher and controller.

3.3.2 Garbled Circuit AND Overlay Cell

The AND overlay cells required for garbled circuit generation are much more complicated than single bit operations. Each wire of the AND operation is represented as 80 bits in our implementation. A basic garbling AND operation implements the functionality described in Section 2.1 and shown in Fig. 2.2. Each line is implemented according to Equation 2.1. This implementation requires four SHA-1 cores, using 512-bit values derived from the garbled inputs and additional information. The SHA-1 core inside the garbled circuit AND overlay cell is based on an open source SHA-1 core [38]. Our garbled AND operation requires 82 clock cycles on the FPGA and uses 3070 ALMs and 3750 one bit registers on a Stratix V FPGA.

Fig. 3.6 shows the hardware architecture of garbled AND overlay cell. K_0^0, K_1^0 and K_2^0 are three garbled value representing value 0 on the wire for an AND operation. R is the global

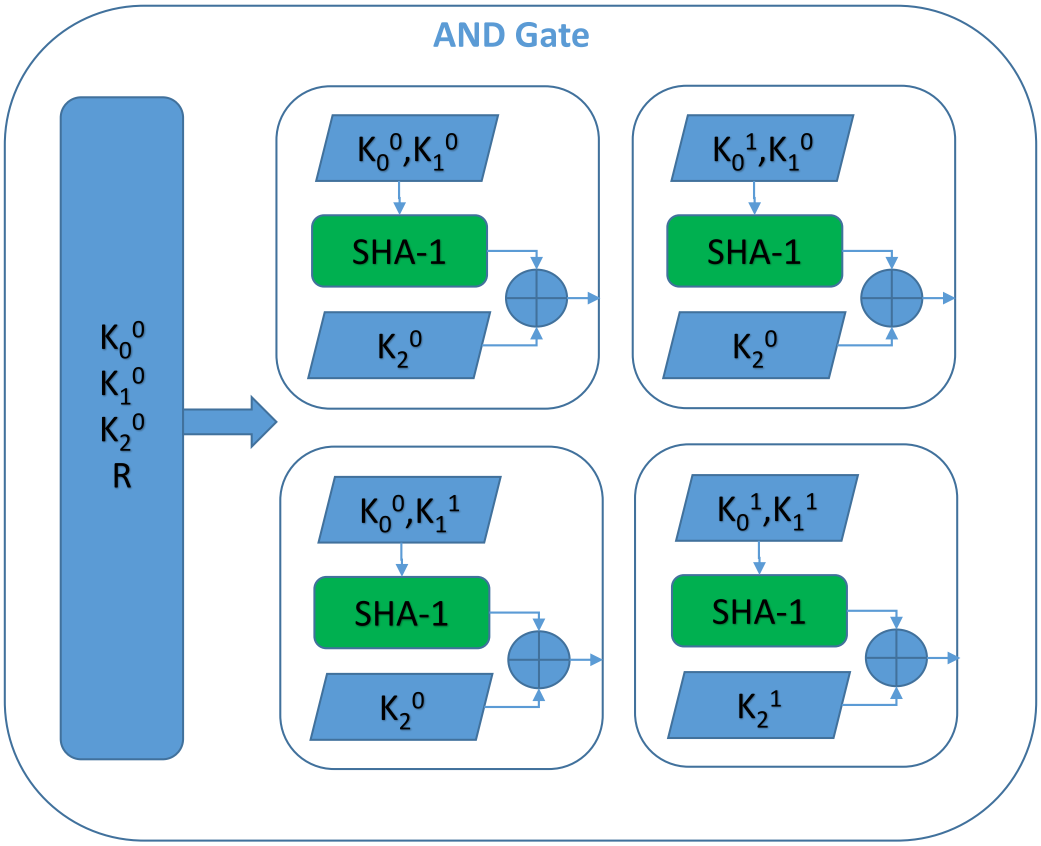


Figure 3.6: A Garbled AND Overlay Cell

variable based on which the cipher can get the garbled value representing value 1. For any wire i, the equation is always $K_i^0 \oplus K_i^1 = R$. This figure shows a basic Garbled Circuit AND operation without applying the garbled table space saving optimization method. Clearly, there are four combinations of the operations for any two pairs of input garbled value, which are K_0^0, K_1^0 and K_2^0; K_0^1, K_1^0 and K_2^0; K_0^0, K_1^1 and K_2^0; K_0^1, K_1^1 and K_2^1. The operation is to compute the message digest the concatenation of the two input values and then XOR with the corresponding garbled value of the output. There are four output values, each of which is in one entry of the garbled table.

We use FlexSC [26] as the software platform to generate garbled circuits. The proposed system should behave exactly the same as what FlexSC can provide. The optimization method that FlexSC applies is row reduction and can save only three instead of four entries inside any garbled circuit AND operation table. They achieve this by making one of the garbled table entries a value zero and randomizing the other three entries. The entry is decided by the garbled operation of two inputs for each AND operation. So is the value of K_2^0.

Fig. 3.7 shows the optimized operation which uses the row reduction method for a garbled AND operation that we implement. The optimized version of the garbled AND overlay cell can reduce the size of the garbled table by 25%, which is also the saving of transmission size. The module still needs four SHA-1 primitives. There are two arbitrators for all four output of the SHA-1 operations. The first arbitrator will decide the sequence of the result and pick one of them which XORs with the other three. The second arbitrator will rearrange the sequence of those three values and store them into the garbled table. Note that these arbitrators do not introduce any latency to the system. Also, four SHA-1 modules are always running in parallel. In Chapter 4, we will verify the functionality via the testbench generated directly from FlexSC and show the timing information.

3.3.3 Garbled Circuit XOR Overlay Cell

Garbled Circuit XOR overlay cell benefits from the so-called “free” XOR protocol. A free XOR gate consists of 80-bit plaintext XOR operations. For any garbled circuit operation, it is guaranteed that using free XOR protocol will have the same validity and privacy as using standard cryptographic primitives. This has been proven [74] under the assumption of the semi-honest model. Using this free XOR protocol, the XOR computation is much smaller than garbled circuit AND.

Fig. 3.8 shows the structure of XOR overlay cell on FPGAs. The computation is $K_2^0 = K_0^0 \oplus K_1^0$ and $K_2^1 = K_2^0 \oplus K_1^0 \oplus R$. It is clear that XOR operation is a combinational datapath and is very suitable for hardware. Compared with software implementation using a for loop, the hardware

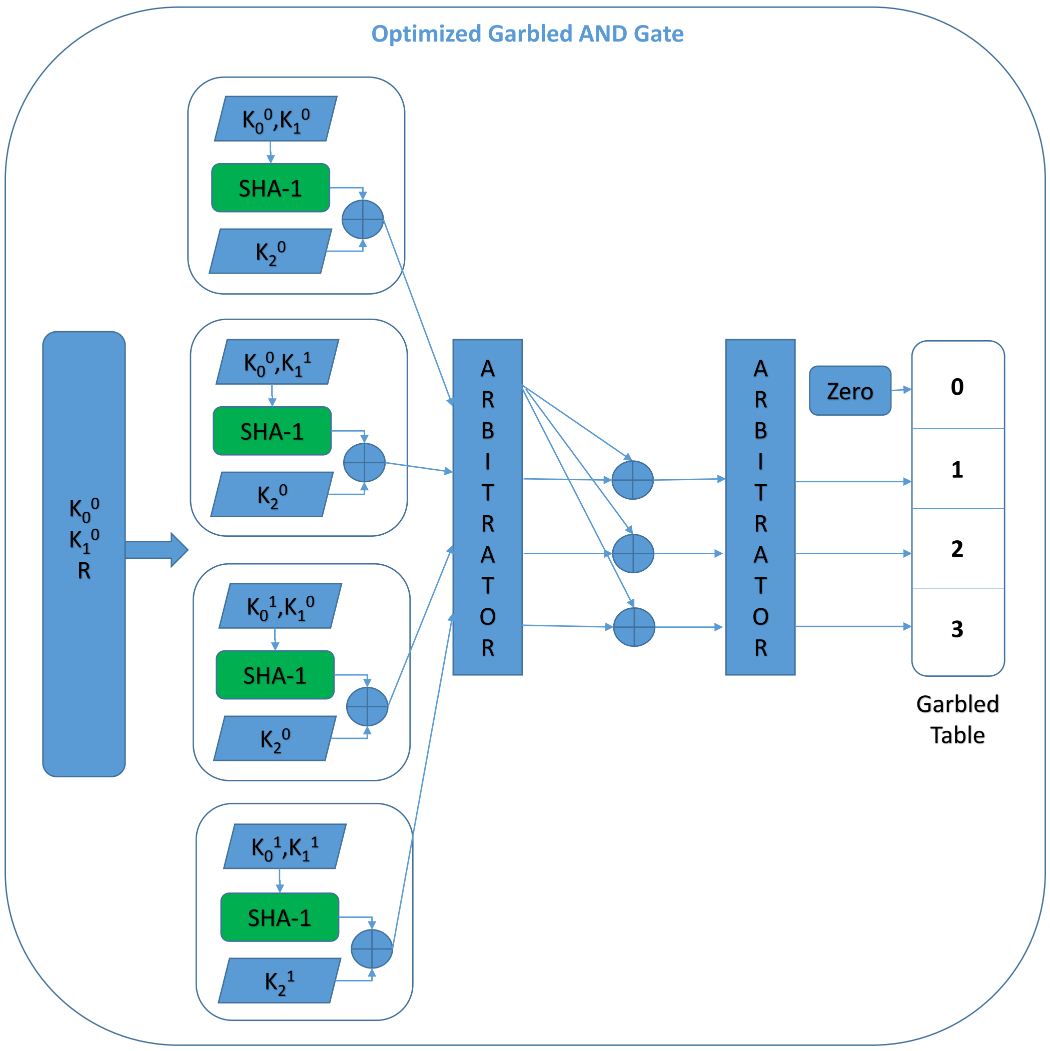


Figure 3.7: Optimized Garbled AND Overlay Cell

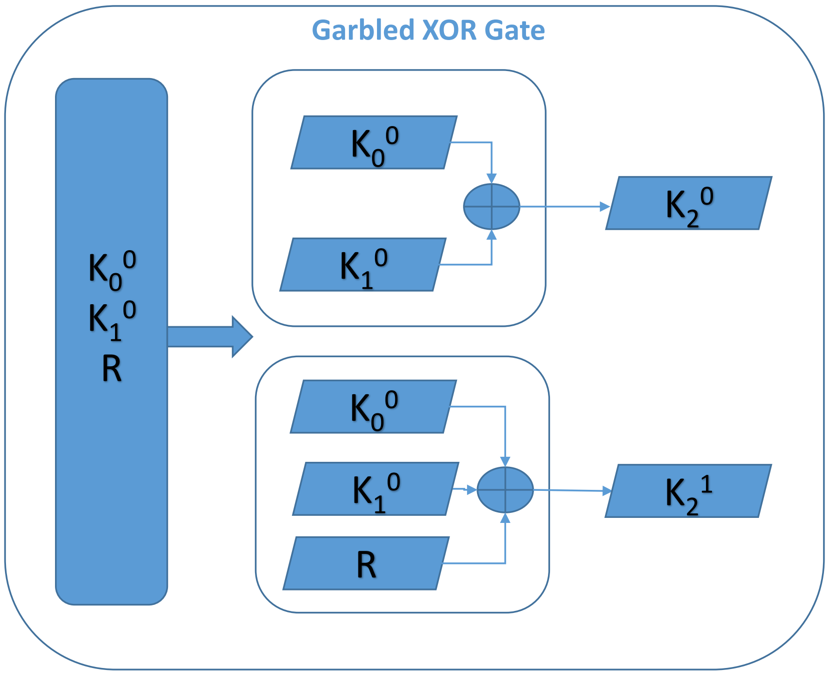


Figure 3.8: A Garbled XOR Overlay Cell

overlay cell also saves a considerable amount of computing time.

3.3.4 Embedded Memory

We support communication between gates by storing all inputs and outputs inside memory. We first explore embedded memory on the FPGA chip for simulation purposes. For Altera Stratix FPGAs, there are two types of embedded memories: one is enhanced Memory Logic Array Blocks (MLAB) and the other is the M20K block memory. Each MLAB is implemented by one Logic Array Block (LAB) containing 10 adaptive logic modules (ALM). M20K block memory is dedicated memory resources and is suitable for providing a large amount of data storage. The tools determine which type of memory is used and the user can set the memory policy.

We use Block Random Access Memory (BRAM) to store the garbled values for each wire (i.e. every input and output for every gate). We treat all of the on-chip memory as one monolithic sequential memory device. The memory is 80 bits wide and implemented with one read and write port. The unique wire ID in the Boolean circuit corresponds to the memory location. This monolithic memory simplifies our design since no decision making is required in determining where to find

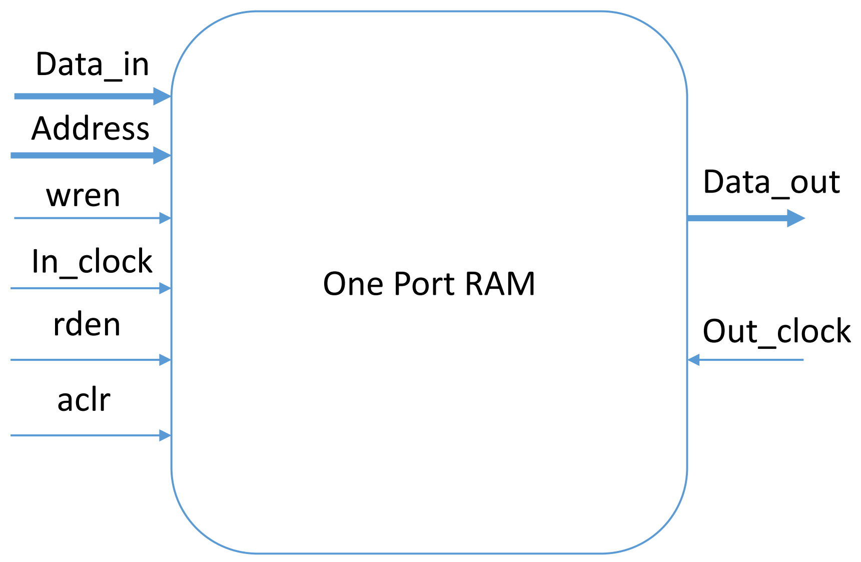


Figure 3.9: A Standard RAM Interface

inputs or where to store outputs.

Fig. 3.9 shows the interface of 1-port RAM MegaCore IP function. MegaCore from the Altera IP library provides the predefined modules with multiple configurations available for users. Users do not need to redesign the functionality. This guarantees a standard interface, increases the robustness of the system, and optimizes for Altera FPGA devices.

For inputs in the 1-port RAM IP core, data input and write enable are specifically for the write operation; port read enable is for the read operation; others ports such as input clock, address and asynchronous clear are shared by both operations. The number of data values can be configurable based on the number of wire within each garbled circuit problem. Output data will be available once the RAM is finished reading. The synthesis process decides the physical implementation of how this IP will be mapped onto Altera FPGAs.

3.3.5 Workload Dispatcher and Data Controller

The workload dispatcher and data controller is responsible for fetching garbled values from the memory based on input or output address, pushing the input data to the unoccupied garbled circuit AND or XOR overlay cell, writing back the result to the memory after each operation finishes.

The design of this module is vital since the timing performance is decided by the state defined within it. In other words, it decides not only the total resource utilization for hardware architecture but also the timing of interact with the host. Fig. 3.10 shows the timing information of the workload dispatcher and the controller. In details, it implements the following steps:

1. Determine the type of the next batch of operations sent from the host
2. Read values in BRAM for input values and forward to a vacant gate
3. When an AND or XOR operation is finished, write the output garbled value to the corresponding location in BRAM
4. For an AND gate, the controller is also responsible for pushing the gate ID and garbled table entries to the FIFO for transmission to the host
5. At the end of garbled circuit generation, read the garbled output value(s) from the wire ID(s) and push them to the FIFO for transmission back to the host

In order to achieve the maximum performance as a heterogeneous computing system, we implement in hardware the maximum number of AND and XOR overlay cells that can be kept busy, taking the latency of the AND operations and the availability of the BRAM for reading into account. In our current design, this results in 43 AND gates. We implement a single XOR gate, as the computation has one cycle latency which makes it always available and additional XORs will not help the performance. Fig. 3.11 shows the timeline for garbled circuit AND operation. The read from and write to BRAM are both one clock cycle latency; garbled circuit AND takes 82 clock cycle; reset takes one. Our overlay architecture with BRAM implements 43 AND cells. Based on the timeline, if there are more AND operation the 44th gate will run on the first overlay cell since it has already completed. In our implementation, XOR operation in a layer will be computed after all the ANDs have started. There may be contention for writing BRAM if XOR and AND operations complete at the same time and this contention is handled by the workload dispatcher. Performance results for this design are given in Section 4.3.

Fig. 3.12 shows the timeline for garbled circuit AND operation. An XOR gate has four cycles of latency total, two for reading inputs, one for computing XOR and one for writing the output.

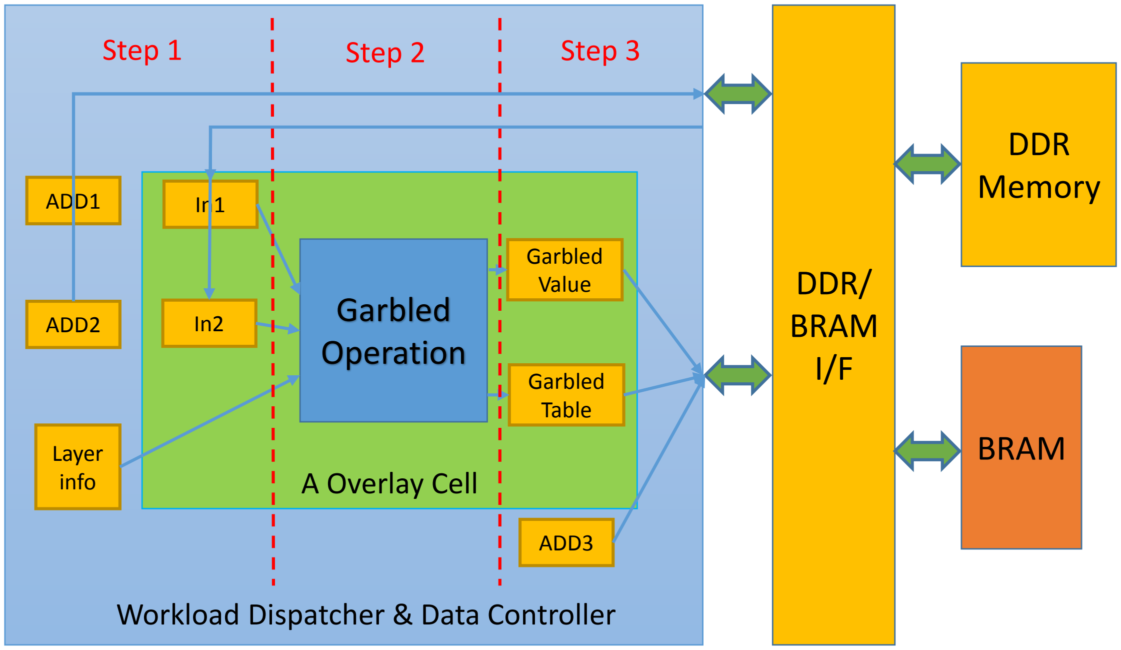


Figure 3.10: Workload Dispatcher and Data Controller Timing information

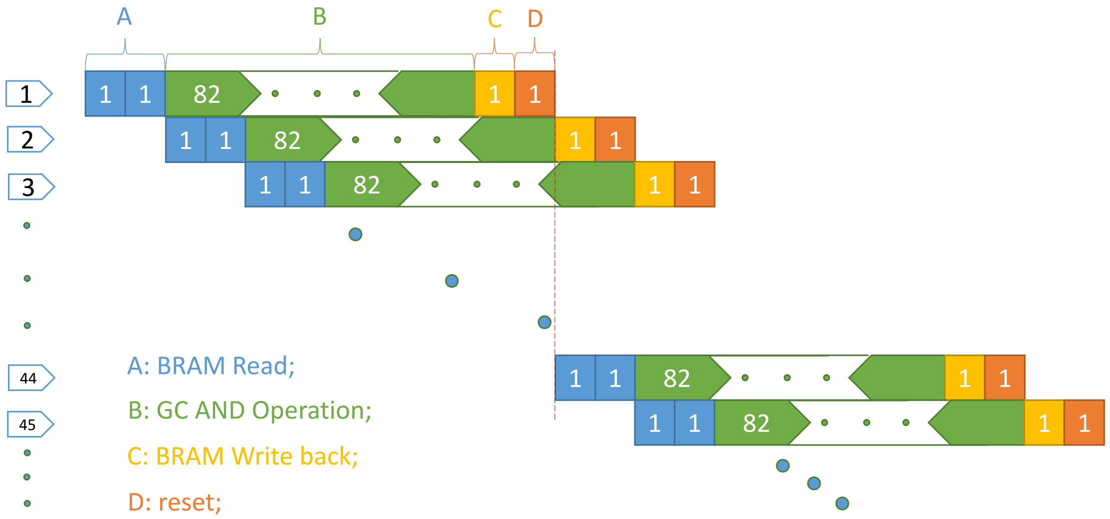


Figure 3.11: Garbled Circuit AND Gate Operations Sequence

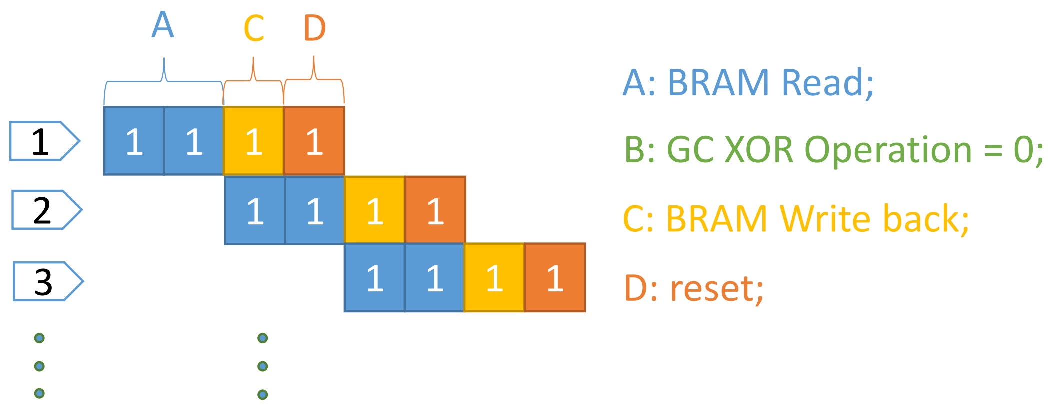


Figure 3.12: Garbled Circuit XOR Gate Operations Sequence

3.4 Heterogeneous Computing System using ProceV Board

After having verified the correctness and validity of using FPGA overlay cells to accelerate garbled circuit Boolean operation, we redesign the hardware architecture to fit in a heterogeneous computing system, specifically, the ProceV board from Gidel Inc. With the support of 8-lane PCIe Generation 3 interface between host and FPGA, the Stratix V FPGAs can perform at its best potential. The device drivers provided by the development kits guarantee the communication between hardware and host. Off-chip DDR memory will allow us to garble large problems. Fig. 3.13 shows the heterogeneous computing system with off-chip DDR memory and PCIe interface.

We will discuss several topics in the following sections. One is the difference between the FPGA hardware only system described in Section 3.3 and the heterogeneous system with hardware and software. In this section, we focus on PCIe interfaces, DDR memory and workload dispatcher and data controller. In Section 3.5, we demonstrate the methodology of improving the performance of the whole system, not only hardware architecture but also the adjustments needed in the software structure.

3.4.1 Communicating between Host and FPGA

The communication channel of the host and FPGA is through PCIe interface. It is a common high-speed serial connection for connecting peripheral boards to CPUs. On the host side,

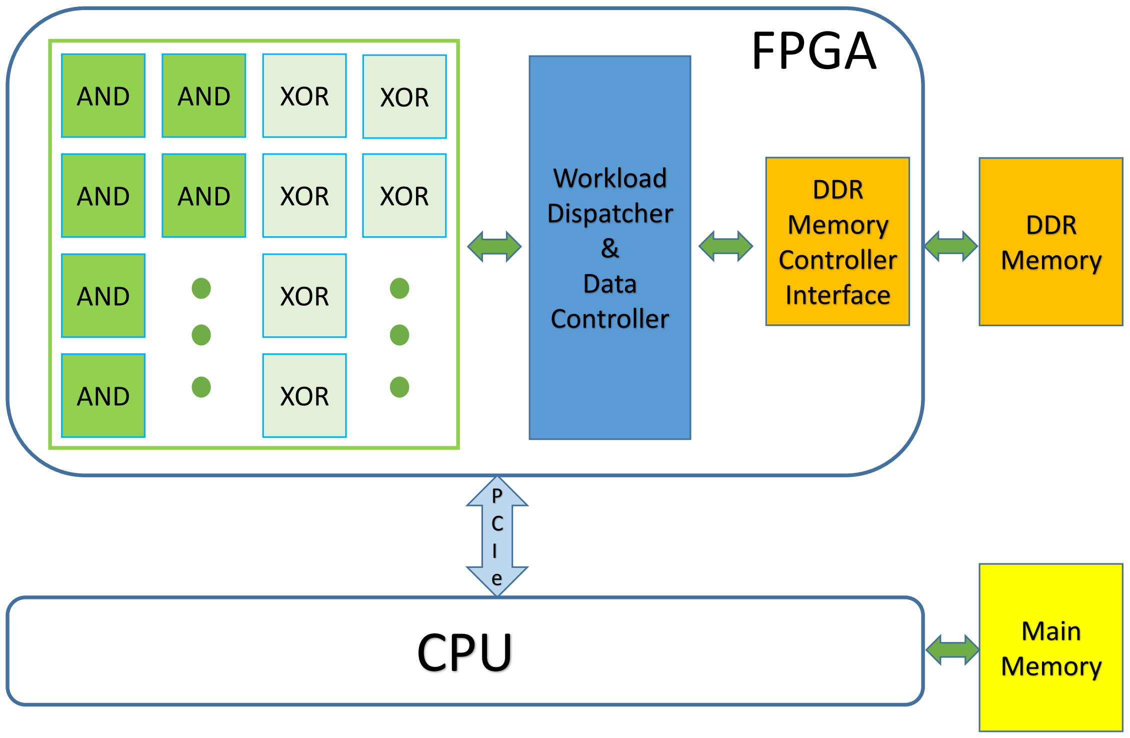

FPGA
AND
AND
XOR
XOR
AND
AND
XOR
XOR
AND
XOR
AND
XOR
Workload Dispatcher & Data Controller
DDR Memory Controller Interface
DDR Memory
PCIe
CPU
Main Memory

Figure 3.13: Heterogeneous Computing System with DDR Memory

the device driver enables the host to transfer data through PCIe in a bi-directional way. On the hardware side, the interface module connecting to the PCIe physical interface is responsible for the data transfer based on the host commands.

There are two modes for the host to transmit the information. First is accessing to DDR memory on board described in Section 3.4.2 and second is accessing the registers allocated inside the FPGA in Section 3.4.3.

3.4.2 Accessing DDR Memory On Board

The ProceV board features 8GB DDR SODIMM memory which is much larger than the embedded RAM on FPGAs. Here are the differences of using DDR compared with BRAM.

- The first difference is the speed of accessing memory. For BRAM, the maximum speed can be 600MHz and only takes one clock cycle for reads and writes. However, the clock frequency is constrained by that of the whole system. On the ProceV board, the fastest speed of the system clock is 200MHz. For on board DDR memory, the ProceV environment provides the specific interface to interact with the memory. Specifically, the system will handle both Direct Memory Access (DMA) transfer and user controlled access to DDR memory. DMA is responsible for the transfer of the initial data from host memory to DDR on the ProceV board. FPGA hardware will later take control of accessing the data via the DDR memory controller interface.
- The second difference is the size of two memories. Size is the pitfall of using embedded memory for storing all the data. The Quartus RAM IP core's largest address space is 65536 which is very limited with 80-bit data. Also, we need to store three entries for each garbled table in memory. Thus the total address space is $65536/4 = 16384$. We can not even map a two 64-bit multiplication operation based on the problem size. This is the limitation of using embedded memory. Our purpose is to map all kinds of garbled circuits on the hardware architecture, and for large problems, the BRAM is not large enough to store all the initial and intermediate data.
- Third difference is the interface. For BRAM, users can use predefined memory IP cores which have many configurations mode to choose from. Compared to the DDR memory access from the ProceV board, the workload dispatch and data controller can access the on board DDR memory via DDR interface shown in Fig. 3.14.

As mentioned above, our aim is to provide a general hardware architecture and software structure for any garbled circuit problems. The limited size of Block RAM inside FPGA is not large enough for the initial data and intermediate data including garbled value and garbled table for most problems. So the next approach is to store all the values onto DDR memory.

For DMA access, the software API is provided by Gidel to utilize the effective DMA mode. These are divided into three steps: preparation, usage, and closing.

- For the preparation step, users will first allocate a memory space in the host memory for a user defined size to be transmitted, the start location of this space, the channel number, the direction of the transmission, etc. A memory handle is returned after this step.
- For the usage step, The handle will be fed into a function which runs the DMA supported by the driver. Before calling DMA transmission, the host can write any initial data into the allocated memory. Gidel also provides other useful functions such as checking the completeness of DMA, remove the memory handler, etc.
- For the closing step, the host unlocks this memory location and release the system resources.

For DDR memory access from user's customized module, the provided interface is shown in Fig. 3.14. It is a single port interface with two groups of input signals for read and write operations. The address is 32 bit wide and the data width can be customized for different widths. The latency of the DDR access is about 170 ns for one read and about 180 ns for one write operation as we measure. The timing information varies a little since there is DDR memory refresh every now and then. This latency result is captured thought real signals via the Altera tool signalTap while the system is running on FPGAs. Port lclk is the clock of the local bus which can be configured while generating the hardware interface. The frequency options for this clock are 100MHz, 125MHz, 150MHz, and 200MHz. This is different from another clock for the module which directly connects to PCIe which has the maximum frequency 300MHz.

The interface changes and the increased latency of DDR memory access make the hardware architecture different from the architecture described in Section 3.3. Here are some design changes implemented to support changing from BRAM to DDR memory in a heterogeneous system.

- It affects the optimal number of AND and XOR overlay cells on the hardware. Compared with standalone FPGAs, the heterogeneous system is different in many ways, including the above-mentioned memory access latency and different clock frequency of both the HW-SW interface clock and the local bus clock. The principle of deciding the number of overlay cells is

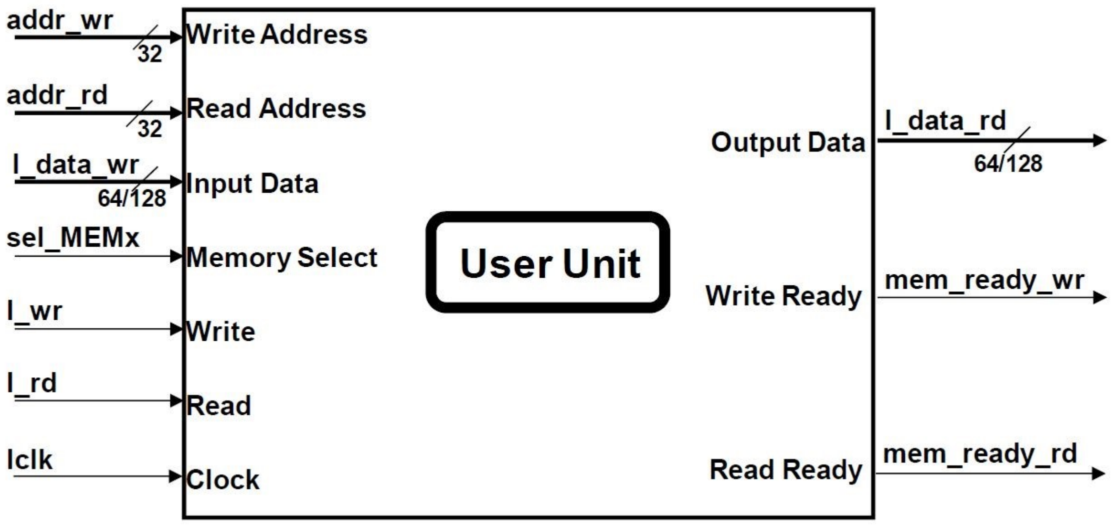


Figure 3.14: ProceV Board DDR Memory Interface

to use the smallest number of garbled circuit AND and XOR gates while keeping the memory interface and PCIe interface as busy as possible. We also try a different number of AND and XOR overlay cell combinations.

- It also affects the address translation while doing code generation from the host side. For the BRAM memory system, the width of the BRAM is configurable within the IP core, which means that one address can represent 80 bits, so each wire id will directly represent one address. In contrast, the width of the DDR memory is 32 bits. This means it needs 3 addresses for one garbled value of one wire ID. For alignment purposes, we allocate 4 DDR addresses for one garbled value. Thus one garbled circuit AND gate will take 16 DDR addresses in order to store the garbled value of the output wire and also three entries of the garbled table.

- The bottleneck of the system changes from FPGA computation bound to the memory and I/O bound. The latency of the garbled circuit AND and XOR operation will stay the same no matter what the memory and I/O latencies are. The memory and I/O latencies change from 1 clock cycle (If 200MHz, 5 ns) to 170 to 180 ns for each DDR access. The memory interface is the new bottleneck of the system and the software should accommodate the access pattern for better performance.

3.4.3 Accessing On-Chip Registers

Not only the initial data but also the two input and one output address of each garbled circuit AND/XOR gate needs to be transferred via the PCIe interface. Each gate will perform the corresponding operations on the values of specific addresses. The host parses the structure of each garbled circuit problem, according to which addresses are sent to the hardware side. Using this mechanism, one fixed hardware architecture can support many different garbled circuit problems. Assume there are m AND gates and n XOR gates, each gate will need in total three address registers which can be written by the host. Thus the total number of registers will be $3 * (m + n)$.

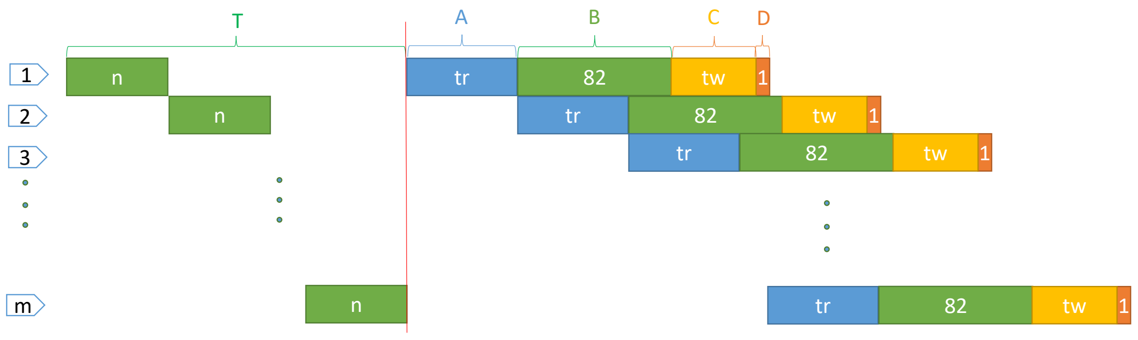
For the host, the application driver contains the register type which represents the I/O register in the design. In other words, a register is a member of the application driver class with direct access to the hardware architecture. Based on the directions, there will be three types of registers: write register, read register and read&write register. For example, the write register can only be written to by the host but can not be read. The host first maps the register to the physical address on the ProceV board and later writes to the register. There are also register array types as a member of the application driver class which the host can access. Similar to register, the host can access the register inside the array via the corresponding index.

The actual time for the host writing to one register on hardware is about 50 ns. The timing information is captured via SignalTap while the garbled circuit is running. So for three addresses of each Boolean gate, the data transmission time is around 150 ns. Along with the transmission of layer information in each batch of operations, the total latency could be even larger. Time overlapping method is provided in Section 3.5. More timing information will be presented in chapter 4.

3.4.4 Workload Dispatcher and Data Controller

For the workload dispatcher and data controller, states in the FSM are added according to the timing information of the memory access pattern. The system using BRAM will only wait for one clock cycle for either read or write; however, for DDR memory, memory access operation is not finished until the rise of the complete flag. Also, data concatenation is needed for DDR memory because of the DDR interface provided.

Here are the steps for the basic version of workload dispatcher and controller for the heterogeneous computing system. Fig. 3.15 shows the time line of one batch of Boolean operations. Once the current batch has finished, the host will send the next batch. Improved version is discussed in Section 3.5.



A: Memory Read (Time tr);

B: GC AND Operation (Time 82);

C: Memory Write back (Time tw);

D: Reset (Time 1);

T: Operation Info Transition (Time n*m)

Figure 3.15: Coarse Granularity CPU and FPGA Communication

1. Get a batch of Boolean operations from the host via input registers.
2. Start the FSM based on the operations; specifically fetch the data and feed the value to the input of the overlay cells.
3. When one Boolean operation is finished, write the output of the garbled gate to the correct location in DDR. For AND gate, three entries of the garbled table are also written back to the DDR in the next memory address of the garbled circuit output.
4. At the end of garbling, raise the flag for CPU to transmit the next batch of operations. Repeat until host finishes sending the operations within all the layers.

3.4.5 AND and XOR Overlay Cells

Switching from BRAM to DDR memory for data storage affects the number of AND and XOR overlay cells because of the change of timing information. For example, the time of one XOR Boolean operation which includes accessing the memory three times takes longer than the host is needed for transferring the wire information of one XOR operation. That means increasing the number of XOR overlay cells within the hardware will increase the performance of the whole system.

We tried different numbers of AND and XOR overlay cell combination in our experiments and the results are presented in Chapter 4.

3.5 Architecture Improvement

In this section, we focus on the different sources of bottleneck that hamper the total performance of the garbled circuit generation system. Each section will describe a bottleneck of the standard hardware architecture described in Section 3.4 and the disadvantages. We then discuss methods to increase the performance. The architectural improvements include a hybrid memory hierarchy, improved host to FPGA communication, and optimizations to the overlay architecture especially the number of AND and XOR overlay cells.

3.5.1 Hybrid Memory Hierarchy

For a hybrid memory hierarchy, we are seeking to use both Block RAM and DDR memory for better performance. The character of the two memories is obvious: Block RAM is within the FPGA chip and is faster but small in size; DDR memory is large but slow. We can take advantage of the benefit of each memory, while at the same time avoid the shortcomings. In a CPU memory hierarchy, the purpose of using cache architecture is to reduce the time for data access. Modern CPU shares three level caches and with DDR memory, and the total data access time is much lower than using DDR memory alone. These hierarchies also keep the size of the cache close to the CPU as small as possible to reduce the expense.

Fig. 3.16 shows the improvement of our hardware architecture using both the BRAM inside the FPGA and the DDR memory on board. We define the rules by which garbled values are stored in BRAM via a register allocation policy for BRAM. Values for the rest of the wires will be stored on DDR. This architecture supports many different register allocation policies by the software.

For the hardware architecture, we configure the workload dispatcher and data controller to accommodate this hybrid memory hardware architecture. The controller monitors the flag of the address provided from the host and decides which to interact with, either the Block RAM or DDR interface. We use the last bit of the address location as the flag, and if the flag bit is zero, the location is DDR memory address; otherwise, it is a BRAM address. There is no need to provide an extra register flag for memory type. The remaining 31-bit address (besides the flag bit) is more than enough to represent the address domain and this saves one register write communication time for

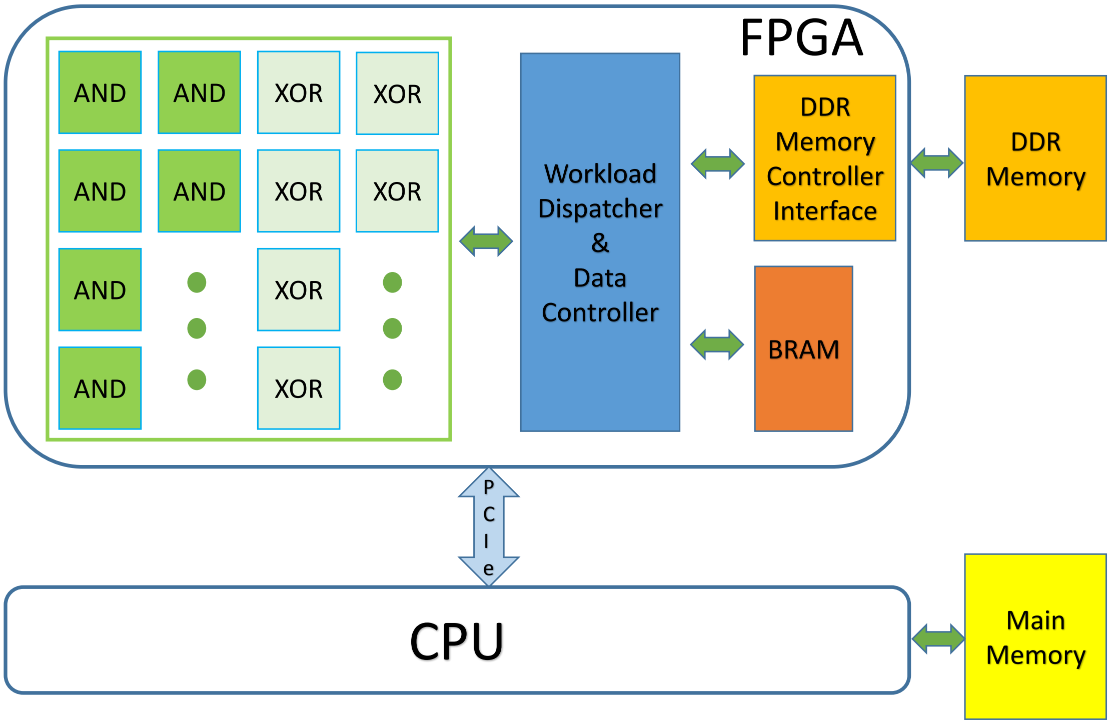


Figure 3.16: Hybrid Memory System

each wire. For each Boolean gate, the saving time is around 150 ns, which is considerable, especially for large problems.

On the software side, the problem parser will analyze the layer and wire information based on the register allocation policy before generating two groups of wires IDs for BRAM and DDR memory. Based on this information, the host code generator generates the locations of the memory based on its type. We have implemented two register allocation policies for assigning the wire IDs. The first one is the directly-used policy and the other is the most-frequently-used policy. We describe both of the policies and their time-saving analysis as follows.

For the directly-used policy, we define that the directly-used wire will be stored in BRAM rather than in off-chip memory. We select the specific wire ID by analyzing the characteristics of all the wires and pick the wire ID based on two criteria: (1) the wire is used only once after it is generated and (2) the Boolean gate which uses this wire ID is in the adjacent layer. In Chapter 4, we show the number of wires that meet this requirement for different problems and analyze the minimum space to be allocated on BRAM.

The advantage of using this policy is that the BRAM can be reused compared with DDR memory, where the latency is around 170 ns for the read operation and 180 ns for write, the access of BRAM is only 5 ns when using the 200 MHz local clock frequency. The second benefit is that space can be reused once the value is used by the next layer's Boolean operation. For the design, careful consideration is needed for the addresses that can be reused. We can not write the BRAM address to the previous layer's address since the address space is still used for the rest of operation in the current layer. The 1-to-1 output wires of the current layer should be written to another BRAM space. As a solution, we separate the BRAM into two halves and write the output value in switched mode. For all the operations in each layer, if the value on the first half is being read, the second half will store the output values and vice versa. This is a ping pong buffer realized in Block RAM. This mechanism guarantees the value on the address to be written is already used and it can save lots of memory space.

For the most-frequently-used policy, we define that the wires that are most frequently used are stored in BRAM to shorten the total memory access time. Different from the directly-used policy, this one is based on the fact that for those wire ID whose value is accessed multiple times, using one address of BRAM could saving more DDR access time. For a 1-to-n wire ID, the timing saving is N times the time difference between DDR Access and BRAM Access. However, the tradeoff of using this policy is that space is not immediately available for reuse, which considering the limited amount of locations on BRAM, could be a concern. Prepossessing can analyze each wire's behavior and check the exact time that the BRAM space allocated to it can be reused. In other words, There is a potential improvement of doing register life time analysis. Once the last access for a specific wire ID is finished, the location can be reused. We first sort the wire ID based on the number of accesses which is N and put the wire IDs with larger N in BRAM. Once BRAM can not fit more 1-to-n wires, the rest of the wire ID with smaller N will be translated to the addresses in DDR Memory. Both of these policies are compared with each other and we present the timing results in Chapter 4.

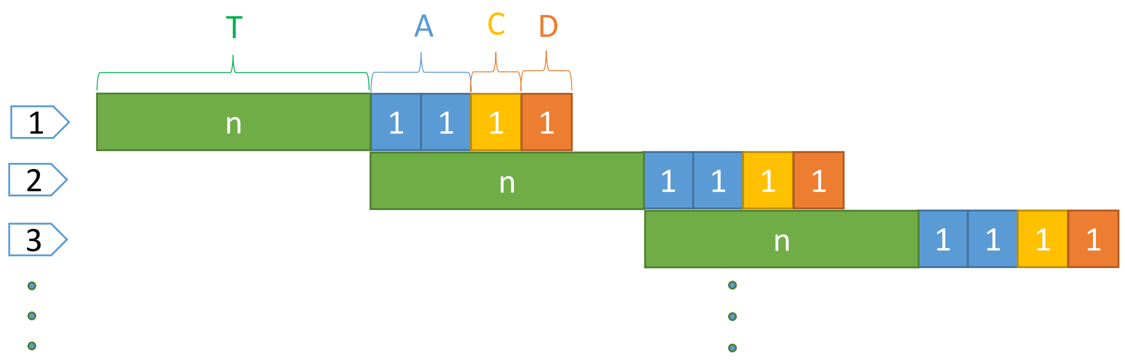
3.5.2 Host to FPGA Communication

Host to FPGA communication contributes major delay for our garbled circuit heterogeneous reconfigurable system as described in Section 3.4. First, the address access pattern of garbled circuit problem is not contiguous so we cannot use burst mode for streaming of data; second, address information has to be sent by the host on-the-fly in order to support any garbled circuit problem over FPGA overlays.

These characteristics of garbled circuit computation make the data transmission between host and FPGA a major bottleneck of the system. The base design proposed in Section 3.4.1 suffers drawbacks brought by the communications on the PCIe interface. The host has to wait until a batch of operations is finished before sending the next batch. The total time includes not only the data transmission and garbled circuit operation but also the synchronization steps between hardware architecture and the host.

There are three principles of improving the performance of communication between host and FPGA. First is to avoid unnecessary checking signals; second is to reduce the number of registers for each batch of operations sent from the host; third is to implement pipeline methodology and increase the fine granularity of information sent from the host for time overlap between data transmission and garbled circuit operation. A detailed explanation is as follows.

Our first principle of avoiding unnecessary checking signals means that the host does not have to wait before sending the next batch of operations. Fig. 3.17 shows the situation. In the heterogeneous system using BRAM alone, the timing it takes for the host to send the information of one XOR gate is longer than the actual operation which includes fetching the input data, doing XOR operation and writing back to BRAM. Based on this fact, the host does not have to check the finish flag of XOR operation and can keep sending the gate information of XOR operation one by one. However, if DDR memory is involved in the system, this optimization does not apply. If the main clock is 300 MHz and the local clock is 200 MHz, the host has to wait for the finish for one batch



A: BRAM Read;

B: GC XOR Operation = 0;

C: BRAM Write back;

D: reset;

T: Operation Info Transition

Figure 3.17: Data transmission for XOR Operation without Check using BRAM

before sending another one due to the synchronization between the host and FPGA. The solution is to increase the number of XOR overlay cell on hardware.

Second is to reduce the number of registers the host needs to write to for each batch of operations. It is realized by concatenating values from several registers into one register. We use two address registers to represent three addresses of information for each gate. The total width of 2 addresses is $2register * 32bits/register = 64bits$ and the actually bit-width for each address location in our design is $\lfloor 64/3 \rfloor = 21$ bits. Besides the flag bit representing the memory type, there will be 20 bits for a real address which is about one million wires. We demonstrate the feasibility of using this optimization method for problems less than one million wires. Fig. 3.18 shows this optimization.

The third method is to increase the fine granularity within a batch of operations. Fig. 3.19 shows the timeline after using the pipeline methodology. We design the communication mode which can overlap the transmission of one Boolean gate information and Boolean operation. Overlay cells can start working as soon as the transmission of a new Boolean operation is finished. So the granularity of the operation changes from a batch of operations to one Boolean operation. At the

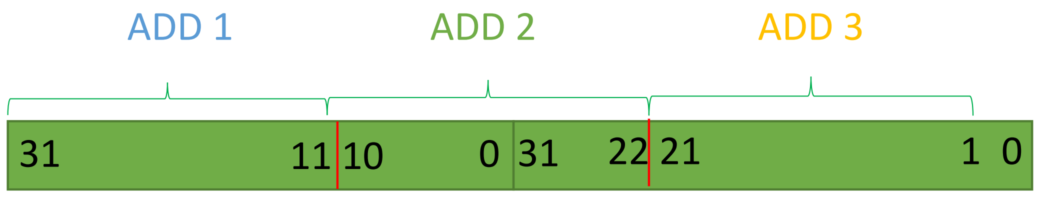


Figure 3.18: Reduce of Number of Registers

same time, the host does not have to change any thing. In order to recognize the new operation, we check if a specific register changes its value. For AND, it is gate id. For XOR, it is the address of output wire ID. Since this ID is the last to be written to the hardware. Note that, if using the directly-used policy for choosing the address for BRAM, the output address can be reused. So the workload dispatcher and data controller cannot recognize a new operation simply via the change of this address. Our solution is to use a flipped bit within the output address so that the controller knows the new operation.

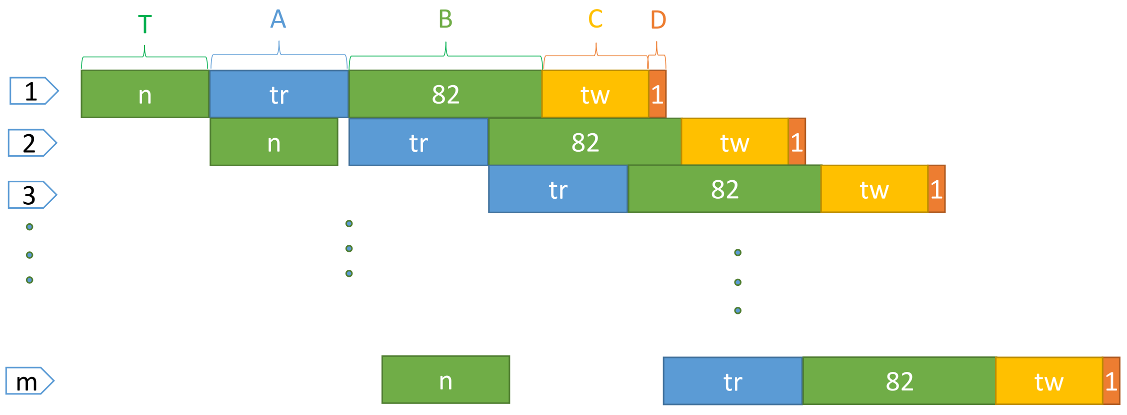
3.5.3 AND and XOR Overlay Cells

In order to find the optimal number of AND and XOR overlay cells for the system, the system's memory hierarchy is a key factor. Here are some situations which affect the number of overlay cells:

1. For using only BRAM, the number of XOR overlay cells is one and the number of AND overlay cells is 43.

2. If we choose the workload dispatcher and data controller without pipelining, the optimal number of AND/XOR gates can not be decided before experiments on the real system, no matter what type of memory we choose.

3. If we use the workload dispatcher and data controller with the pipeline, the optimal number of AND/XOR gates is the larger the better. Timing results are shown in Chapter 4.



A: Memory Read (Time tr);

B: GC AND Operation (Time 82);

C: Memory Write back (Time tw);

D: Reset (Time 1);

T: Operation Info Transition (Time n)

Figure 3.19: Fine Granularity CPU and FPGA Communication

3.5.4 System Parameters

There are also many system parameters that affect the total performance. To improve the system performance, we increase the total clock frequency of our overlay architecture from 100MHz, 150MHz to 200MHz. The interface clock frequency changed from 200MHz to 300MHz.

Other potential improvement methods are not supported by the infrastructure of ProceV. For example, the register size for the host to access is limited to 32 bits. If we could write a larger size register for each access, the data transmission over PCIe could be faster. Also, the DDR memory interface cannot read and write at the same time which also limits the performance provided by the overlay.

Chapter 4

Experiments and Results

This chapter applies the methodologies discussed in Chapter 3 and shows the corresponding results. We give the step by step system improvement for both hardware architecture and software structure. Especially we focus on solving the bottlenecks of the data transmission, garbled operation, and synchronization between hardware and software.

In this chapter, we first summarize the workflow of the whole system in Section 4.1. Then we summarize the problems we are targeting and show the characteristics of each one in Section 4.2. In Section 4.3, we show the simulation results for the hardware architecture presented in Section 3.3. In Section 4.4, the results of using a heterogeneous computing system are included. Also, we explore the architectural improvement method mentioned in Section 3.5 and discuss the results.

4.1 System Workflow

Our heterogeneous system consists of a host PC and target FPGA card for acceleration as shown in Fig. 4.1. This figure highlights the tasks performed on both CPU and FPGA side.

Hardware on the FPGA side can map any garbled circuit problem thanks to the overlay architecture mentioned in Chapter 3. Users do not have to redesign and follow the traditional FPGA design workflow such as synthesis, place and route, program, etc. All the user has to do is to program the FPGA ahead of time using a generated bit stream, in our case the SRAM Object File(.sof). An sof file can configure the SRAM based Altera device via the programmer. The software thus will have the ability to utilize the architecture via different generation patterns. We explore different hardware architectures, focusing on PCIe communications and memory hierarchy. The hardware platform is well tested and very robust.

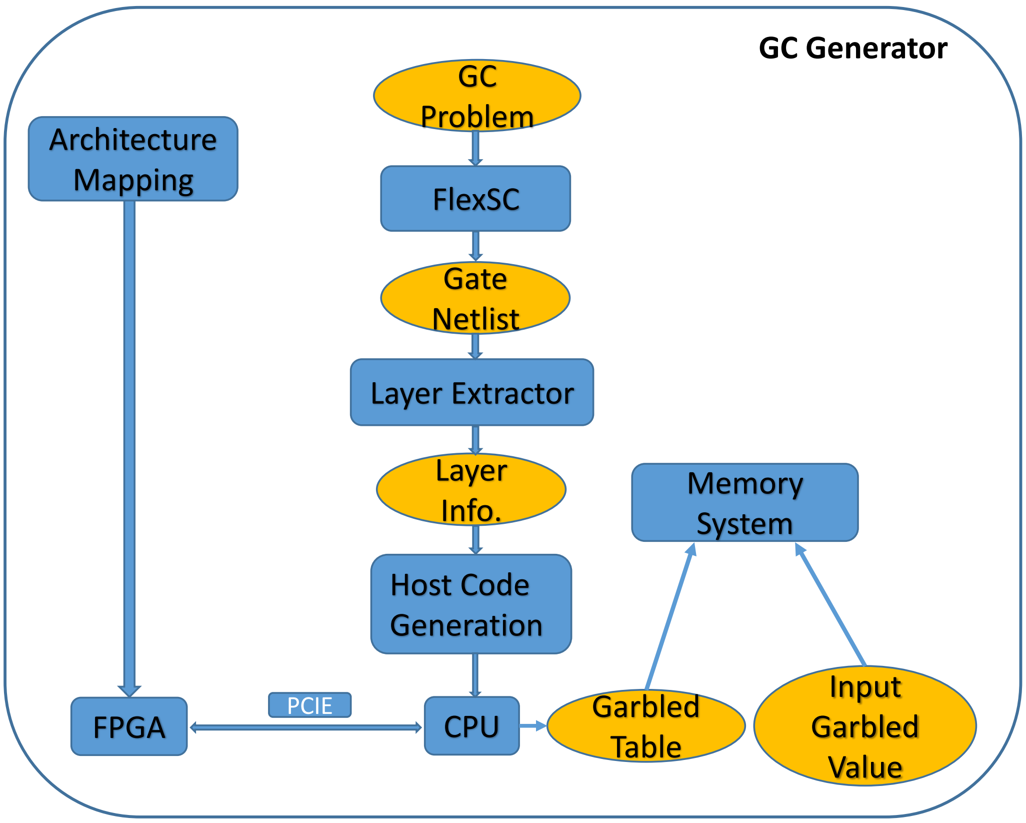


Figure 4.1: System of Garbled Circuits

Table 4.1: Problem Switching Time

Our Workflow		Traditional Workflow
Hardware Architecture	Software Generation	Hardware Design
One Time Compile less than one hour	Minutes to hours depending on size	Every problem takes days to weeks

We compare the reuse time by measuring the compile time of our overlay architecture and the traditional FPGA workflow. For compile time, the overlay architecture can be reused without recompilation, while the traditional FPGA workflow has to design whole circuits based on the new problem and recompile even if a small change is made. Table 4.1 reflects this discussion.

For software workflow, we make use of FlexSC based on ObliVM [75] as the software framework that allows developers without any cryptography expertise to convert algorithms expressed in a high-level language to GC expression. FlexSC is a Java platform for the garbled circuit implementation for various arithmetic operations. It enables easier access to the garbled circuit for a general software engineer. Specifically, it realizes an arithmetic library whose backend is realized using the protocol of garbled circuit provided by ObliVM.

Also, it supports several modes of the garbled circuit. For example, regular garbled circuit mode comes with row reduction optimization which saves 25% generation and transmission size option, etc. We modify FlexSC to generate the operations performed which will be stored for later usage. After this stage, the output file with all the Boolean operations for this problem will be available for further translation.

After the layer information is generated, we have the following tools to rearrange the gates within the same layer to better fit the architecture and communication pattern for this heterogeneous platform. Also, for each problem, we can automatically generate the corresponding host code written in C plus plus. We only need one time bit programming for FPGAs meaning that the FPGA acceleration core can fit any garbled circuit problem and the only difference is the host code. The host is responsible for initial data transmission through Direct Memory Access (DMA), controlling the garbling operations and sending corresponding signals and addresses.

What is sent to the FPGA for each layer is in the form of batches. One batch of operations includes the number of gates in the layer, the input and output wire IDs for the layer, and for the AND gates the gate ID. If one layer does not fit on FPGA, it can be separated into several batches, which allows us to handle large problems. Layer 0 requires values for the inputs, which are 80-bit random values generated for each possible input value, i.e. $k^0_{w_i}$ and $k^1_{w_i}$. These strings are generated using a random number generator for each input wire w_i.

The results of the garbler include the garbled table values and the garbled values of the outputs stored in memory. These later are sent to the host CPU. The garbler also provides input keys, based on the input data from each user sent to the garbler via oblivious transfer, as described in Section 2.1.

4.2 Problem Analysis

In order to check the speedup compared with FlexSC, which runs exactly the same protocol as our system, we choose various problems and check the garbled circuit generation time within both platforms. We also use the problem parser to obtain useful information for those problems. This information is helpful for analyzing the characteristics of each problem and also is useful for choosing the parameters while designing our system.

Problems we choose include the millionaire's problem, addition, hamming distance, multiplication, sorting, sum, and matrix multiplication. We choose different sizes of these problems to check scalability.

We develop the problem parser to shows many aspects of each problem, including general gate information and layer information. There is also information about wire ID for classification using two different register allocation policies, such as 1-to-1 wires to be used in the next layer for the directly-used policy and 1-to-N wires for the most-frequently-used one. We show the analysis results in Tables 4.2, 4.6, 4.7 and 4.8 and discuss these further below.

4.3 Simulation Results

We first show simulation results for the hardware architecture for garbled circuits presented in Section 3.3. The memory for storing garbled values and garbled tables is BRAM on the FPGA. The circuit consists of 43 garbled AND gates, 1 XOR and the workload dispatcher and data controller. We focus on two parts in this section. One is the verification of the circuit in Section 4.3.1 and the second is performance in Section 4.3.2.

4.3.1 Testbench Generation

We build the testbench from FlexSC inputs to check the correctness. We design the hardware along with the test which includes the test of different numbers of Boolean operation within

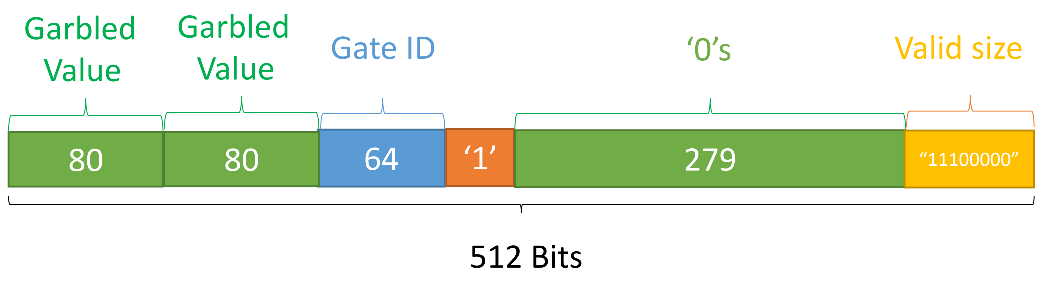


Figure 4.2: SHA-1 Message Padding

a batch of operations and across batches.

There are four SHA-1 cores in our design and the block size of SHA-1 is 512 bits. The SHA-1 input has to follow the standard for correct functionality. Here is the situation for the garbled circuit AND gate. For each garbled value in two wires within one AND gate, the bitwidth for each wire is 80 bits. The gate id is 64 bits. Based on the SHA-1 standard, the format is as shown in Fig. 4.2.

4.3.2 Performance Results

We use FlexSC to generate the Boolean circuit representation fed into the FPGA. We also use it to run our experiments. We compare our results with FlexSC to validate our designs and also compare run times to show speed up. This design is not fully working in hardware so the experimental results we provide are estimates based on the design tools and placed and routed circuits. We present fully operational hardware in Section 4.4. The maximum frequency achievable for this overlay architecture is 200 MHz.

We compare the number of clock cycles between FlexSC and our approach. We precisely monitor the clock cycle count purely for the computation of garbled circuit. We sum the clock cycle times for the XORs and ANDs to provide the computing time on the FPGAs. Note that these operations are performed serially on the CPU, but in parallel on the FPGA.

The problems that we garble are millionaire's problem, addition, Hamming Distance (HD), multiplication and sorting. The size of these problems is shown in Table 4.2. To show the flexibility of our design, different problems have different numbers of bits of input. The millionaire's problem

Table 4.2: Size of the Examples

Problem	# of AND Gate	# of XOR gate	# of layers	Max # of AND gate in One Layer
Millionaire (2)	2	11	7	1
Addition (6)	6	24	18	1
Hamming Distance (10)	20	90	22	5
A * B (8)	120	352	57	64
A * B (12)	276	816	89	144
Sorting (10*4)	848	4683	278	32

Table 4.3: Clock Cycle Comparison

Problem (Input Size in bit)	Our Approach (Clock Cycle)	FlexSC (Clock Cycle)
Millionaire (2)	$1.9 * 10^2$	$1.1 * 10^6$
Addition (6)	$5.6 * 10^2$	$1.7 * 10^6$
Hamming Distance (10)	$1.2 * 10^3$	$4 * 10^6$
A * B (8)	$4.4 * 10^3$	$3 * 10^7$
A * B(12)	$7.8 * 10^3$	$6.1 * 10^7$
Sorting (10*4)	$1.1 * 10^4$	$1.4 * 10^8$

uses 2 bits for each person; the adder is 6-bit wide. Hamming Distance takes two 10-bit values. We garble two multipliers, one with 8-bit inputs and one with 12-bit inputs. Sorting orders the sequence of two groups of digits each of which has ten 4-bit integers. Remember this design requires that the entire problems fit on a single FPGA and in embedded memory which limits the size of problems we can handle.

Table 4.3 compares clock cycles of our FPGA design with FlexSC. The FPGA implementation requires about 10^4 times fewer clock cycles than FlexSC and demonstrates the advantage of implementing garbling using FPGAs. The software platform runs on a computer with Intel Core i7-2640 CPU at 2.80 GHz; the FPGA design runs at 200 MHz. Taking this into consideration, Table 4.4 presents the expected speedup. Our approach is two to three orders of magnitude faster. In addition, we expect the FPGA implementation to consume much less power.

Table 4.5 shows the resource utilization for our FPGA overlay system. The system uses 75% of the available logic in ALMs and about half of the M20K for BRAM and FIFO.

Table 4.4: Speedup compared with FlexSC

Problem	Speedup
Millionaire (2 bits)	422
Addition (6 bits)	222
Hamming Distance (10 bits)	243
A * B (8 bits)	498
A * B (12 bits)	571
Sorting (10*4 bits)	929

Table 4.5: Resource Utilization

Module	ALMs	M20Ks	1-bit Register
One AND	3,070	0	3,750
One XOR	40	0	81
BRAM	0	1,060	0
FIFO	510	280	404
Whole Design	176,893/234,720 75.4%	1,340/2,560 52.3%	215,308

4.4 Heterogeneous Computing System Results

After completing the simulation, we switch to the ProceV board from Gidel as our target platform for the evaluation of a heterogeneous computing system. The ProceV board is a Stratix V FPGA-based platform with 16+GB external memory. It provides high-speed communication between host and FPGA via a PCIe*8 generation 3 bus which makes the system suitable for high-performance computing and low latency networking projects. The ProceV system is supported by Gidel's ProcWizard software and IP. The Altera Stratix V FPGA on board provides high capacity and high speed for many designs and contains 234K Adaptive Logic Modules (ALM) and 52M memory bits. PCIe generation 3 doubles the data rate compared with generation 2 with 8 Giga transfers per second (GT/s) per lane. The ProceV board, with 8 lanes, will provide about 7.88 Gb/s throughput. This high throughput benefits the data transfer between the host CPU and the Stratix FPGA on board. This system provides APIs for developers to access the device driver through the host.

The structure of this section is that first we show the analysis results of all the problems that we tackle using this ProceV board in section 4.4.1 and then show the software and hardware improvement via performance results in section 4.4.2.

Table 4.6: Gate Information for Problems

Problem	Layer #	AND #	XOR #	Gate #	AND %	Gates/layer
6-bit adder	17	6	24	30	20.0	1.8
10-bit HD	22	20	90	110	18.2	5.0
30-bit HD	27	60	270	330	18.2	12.2
50-bit HD	32	100	450	550	18.2	17.2
8-bit multi	57	120	352	472	25.4	8.3
16-bit multi	121	496	1472	1968	25.2	16.3
32-bit multi	249	2016	6016	8032	25.1	32.3
64-bit multi	505	8128	24320	32448	25.0	64.3
10 4-bit sorting	278	848	4638	5486	15.5	19.7
5 5 4-bit m_mult	25	3900	11600	15500	25.2	620.0
10 10 4-bit m_mult	27	7526	22489	30015	25.1	1111.7
5 5 8-bit m_mult	57	15800	47200	63000	25.1	1105.3
10 10 8-bit m_mult	57	127200	380800	508000	25.0	8912.3
20 20 4-bit m_mult	37	254400	761600	1016000	25.0	27459.5

4.4.1 Problem Analysis

This section shows the different problems that we have been working on for speedup on performance. The problems include addition, hamming distance, multiplication, sorting and matrix multiplication.

Table 4.6 shows the layer information for those problems. For example, 6-bit adder means the addition operation of two 6-bit integers; 10-bit HD means the hamming distance between two 10-bit value; 32-bit multi means the multiplication between two 32-bit integer; 10 4 sorting means to sort 10 value each has 4 bits; 10 10 8 m_mult mean two matrix multiplication each has 10 rows by 10 column and each value is 8 bits. Layer number and gate number reflect the size of each problem. The computational complexity of XOR gates is relatively low because of the free XOR policy. AND gates are the most computation intensive part of the operation. From the table, we can see the percentage of AND operations varies from 18.2% to 25.4%. The rest of the operations are XORs. Gates per layer show the average number of gates per layer which do not have data dependency and this gives us an estimate of the size of the layers and the problems. From this table, we can see that our largest problems reach one million garbled circuit operations and there can be several thousand independent gates within each layer. Also, the largest number of layers is not necessarily from the largest problem.

Table 4.7 shows the wire information for all the problems. Each wire corresponds to one memory location. The more wires, the more memory needed for storing the values of the complete the operation. The problems we choose have a wide range of wire numbers from several dozen to one

Table 4.7: Wire Information for Problems

Problem	Wire	A wire	B gate	C gate	D gate	max E	wire/layer
6-bit adder	42	12	0	12	0	1	2.5
10-bit HD	140	55	0	50	5	7	6.4
30-bit HD	420	147	0	163	11	22	15.6
50-bit HD	700	293	0	269	24	37	21.9
8-bit multi	495	296	0	247	49	64	8.7
16-bit multi	2015	1232	0	1007	225	256	16.7
32-bit multi	8127	5024	0	4063	961	1024	32.6
64-bit multi	32639	20288	0	16319	3969	4096	64.6
10 4-bit sorting	5717	2968	0	2136	832	40	20.6
5 5 4-bit m_mult	16175	9700	0	8350	1350	2000	647.0
10 10 4-bit m_mult	31472	18768	0	16051	2717	3809	1165.6
5 5 8-bit m_mult	64375	39400	0	32850	6550	8000	1129.4
10 10 8-bit m_mult	517500	317600	0	263400	54200	64000	9078.9
20 20 4-bit m_mult	1050800	635200	0	541600	93600	128000	28400.0

A: 1-to-1 wire; B: gate with both input the 1-to-1 wire; C: gate with one 1-to-1 wire from adjacent layer; D: gate with one 1-to-1 wire NOT from adjacent layer; E: 1-to-1 wire in a layer.

million wires. There are also different types of wires such as 1-to-1 wire and 1-to-N wire. For 1-to-1 wires, this includes 1-to-1 wires where the output of one gate is immediately used in the following layer, and 1-to-1 wires where the output is not used in the following layer. The former type is a column C gate and the latter is column D. We choose the former type in column C gate of 1-to-1 wire for good temporal locality for directly-used policy in the hybrid memory system. One finding is that there is no gate with 2 1-to-1 wires, which means at least one of the two inputs are used in other operations. The maximum number of 1-to-1 wires in a layer means the size of memory needed for directly-used policy for full speedup.

Table 4.8 shows the information of the percent of each type of wire. Percent A shows the number of 1-to-1 wires among all the wires and based on the table, multiplication, sorting, matrix multiplication has about 60% of 1-to-1 wires. We can see that most of the 1-to-1 wires are used in the next layer represented in Percent B column. This makes the directly-used policy a good fit for many garbled circuit problems with much less memory space required compared with the most-frequently-used policy. Percent C column shows the percent of the 1-to-1 wire to be used in the next layer among all the wires.

Table 4.8: Wire Percent for Problems

Problem	Percent A	Percent B	Percent C
6-bit adder	28.57%	100.0%	28.6%
10-bit HD	39.29%	90.9%	35.7%
30-bit HD	35.00%	110.9%	38.8%
50-bit HD	41.86%	91.8%	38.4%
8-bit multi	59.80%	83.4%	49.9%
16-bit multi	61.14%	81.7%	50.0%
32-bit multi	61.82%	80.9%	50.0%
64-bit multi	62.16%	80.4%	50.0%
10 4-bit sorting	51.92%	72.0%	37.4%
5 5 4-bit m_mult	59.97%	86.1%	51.6%
10 10 4-bit m_mult	59.63%	85.5%	51.0%
5 5 8-bit m_mult	61.20%	83.4%	51.0%
10 10 8-bit m_mult	61.37%	82.9%	50.9%
20 20 4-bit m_mult	60.45%	85.3%	51.5%

Percent A: Percent of 1-to-1 wire in all wires;

Percent B: 1-to-1 wire to be used in the next layer in all 1-to-1 wires;

Percent C: 1-to-1 wire to be used in the next layer in all wires

4.4.2 Performance Results

Here we show step by step improvement of performance using different levels optimizations mentioned in Chapter 3. Eventually, we come up with one hardware architecture and the corresponding software structure for general garbled circuit problems. We use problems of addition, hamming distance, multiplication and sorting for hardware and software exploration. At the end, we summarize the speedup compared with software.

First, we put all the garbled values on the DDR memory and build the system up using 3 AND overlay cells and 1 XOR overlay cell. We also monitor the FlexSC running time for each problem. Table 4.9 shows the speed-up can get as low as 1.16 which is relatively the same as software. Speed-up is the division between software time and our system's running time. The reason is the data access of each Boolean operation is not contiguous and the DDR interface can not apply burst mode to that. Another reason is the long synchronization time between the host and the FPGA. This design sets the main clock to 200 MHz which is the interface clock responsible for data transmission between host and FPGA, and local clock to 125MHz which is the frequency the workload dispatcher and data controller works at.

We then increase the number of AND gates to 5 and 10 respectively and the speed up is shown in Table 4.10. Total speed-up is the division between software time and our system's running

Table 4.9: 3 AND 1 XOR Overlay Cells Realization for Garbled Circuit

Problem	SW Time (ms)	Ours (us)	Speed-up
6-bit adder	2.06	109	18.90
10-bit HD	2.53	368	6.88
30-bit HD	4.08	1067	3.82
50-bit HD	6.46	1757	3.68
8-bit multi	9.22	1534	6.01
16-bit multi	14.54	6267	2.32
32-bit multi	33.76	24720	1.37
64-bit multi	153.13	100788	1.52
10 4-bit sorting	21.12	18192	1.16

Table 4.10: Increase Number of AND Overlay Cells

Problem	5 AND Overlay (us)	Speed-up	10 AND Overlay (us)	Total Speed-up
6-bit adder	78	26.41	76	27.11
10-bit HD	260	9.73	257	9.84
30-bit HD	765	5.33	741	5.51
50-bit HD	1282	5.04	1210	5.34
8-bit multi	1098	8.40	1058	8.71
16-bit multi	4280	3.40	4218	3.45
32-bit multi	17406	1.94	17056	1.98
64-bit multi	71068	2.15	69858	2.19
10 4-bit sorting	12605	1.68	12375	1.71

time. Results show that there are performance increases by increasing the number of AND gates. But the increase in not obvious from 5 to 10. Later results show that the time is largely consumed by the PCIe communication between host and FPGA and the host has to write one batch of operations before the hardware can start running. This is based on 200MHz for the main clock and 125MHz for the local clock. All architectures have one XOR overlay cell.

We look into the transmission time of each XOR operation and find that it is larger than the XOR operation time. Additional speed-up is the additional speed-up compared with the version with xor check and total speed-up is the division between software time and our system's running time. This means that we can remove the synchronization steps and let the host keep sending XOR gates. For all the XOR operations within one layer, the host can send them one by one via PCIe before sending batches of AND operations. Table 4.11 shows the results of removing XOR operation check from the host. This contributes largely to the speedup as the table shows.

Next, we used a hybrid memory system for speeding up the data access time. We use registers and later BRAM for applying the directly-used policy. Tables 4.12 and 4.13 shows the

Table 4.11: Results for Removing Host XOR Operation Check

Problem	10 AND w/o xor check (us)	Additional Speed-up	Total Speed-up
6-bit adder	60	1.30	34.33
10-bit HD	99	2.63	25.56
30-bit HD	216	3.43	18.89
50-bit HD	365	3.32	17.70
8-bit multi	428	2.47	21.54
16-bit multi	1420	2.97	10.24
32-bit multi	4924	3.46	6.86
64-bit multi	18673	3.74	8.20
10 4-bit sorting	2770	4.47	7.62

Table 4.12: Directly-Used Policy using Register and DDR Hybrid Memory

Problem	10AND + Hybrid Memory 1 (us)	Time Save vs DDR	Total Speed-up
6-bit adder	54	10.00%	57.22
10-bit HD	87	12.10%	29.08
30-bit HD	181	16.20%	22.54
50-bit HD	292	20.00%	22.12
8-bit multi	371	13.30%	24.85
16-bit multi	1264	11.00%	11.50
32-bit multi	4278	13.10%	7.89
64-bit multi	15419	17.40%	9.93
10 4-bit sorting	2243	19.00%	9.42

Hybrid Memory 1 consists of registers on FPGA and DDR on Board

speedup improvement compared with software. These tables also show the percentage of time saved compared with using DDR memory alone. The register has a slightly better speed-up since the register does not have any delay for accessing the values, while for the BRAM, there is one clock cycle delay for accessing the data.

We also implement the most-frequently-used policy using hybrid memory which is BRAM on-chip memory and DDR off-chip memory. We try three large problems and the results are shown in Table 4.14. Policy comparison means the speed-up comparison between two policies which if it is larger than 1, it means using most-frequently-used policy is faster. For 32-bit multiplication and 10 4-bit sorting, most-frequently-used policy performs slightly better than directly-used policy. However, it is the opposite for bigger problems like 64-bit multiplication. Based on the analysis above, large problems tend to have more wires and if the BRAM can not reuse many locations like the directly-used policy does, there will be limitations of BRAM space for even large problems for the most-frequently-used policy. The register lifetime analysis is needed for reusing the BRAM

Table 4.13: Directly-Used Policy using BRAM and DDR Hybrid Memory

Problem	10AND + Hybrid Memory 2 (us)	Speed-up
6-bit adder	54	57.2
10-bit HD	88	28.8
30-bit HD	193	21.1
50-bit HD	302	21.4
8-bit multi	380	24.3
16-bit multi	1284	11.3
32-bit multi	4208	8
64-bit multi	15945	9.6
10 4-bit sorting	2292	9.2

Hybrid Memory 2 consists of BRAM on FPGA and DDR on Board

Table 4.14: Most-Frequently-Used Policy

Problem	10AND + Hybrid Memory 2 (us)	Policy Comparison
32-bit multi	4384	1.04
64-bit multi	15648	0.98
10 4-bit sorting	2425	1.06

locations in the future.

We also check the influence of different local clock frequencies. We use a page-ranking algorithm realized in GraphSC [19] which is a platform of implement graphic operations using ObliVM as the backend. GraphSC implements three steps of solving page ranking problems, which is scatter, gather and apply. This facilitates any software programmer to use the garbled circuit protocol on graph operations. We implement page ranking with two, three, and four lines of information where each line represents either an edge or a vertex. Table 4.15 shows that increasing the local clock helps the total performance of the system. 4% to 10% for changing from 100 to 125 MHz and 18% to 21% for changing from 125 to 200 MHz.

We now switch to 200MHz as local clock and 300MHz as the main clock. One problem is that since the main clock increases and the simplification of the synchronization method for XOR operation, the time of transmitting a XOR operand is no longer larger than the XOR operation time.

Table 4.15: Influence of Clock Frequency of Hardware

Problem	sw (us)	100MHz (us)	Speedup	125MHz	Speedup	200MHz	Speedup
2 PR	465895	42762	10.9	41044	11.35	34786	13.39
3 PR	601691	73069	8.23	66409	9.06	55186	10.9
4 PR	667604	93314	7.15	90087	7.41	74620	8.95

10 AND 1 XOR Overlay Cells using the Directly-Used Policy

Table 4.16: Influence of Number of Gates

Gates Number	Time	Speedup Compared with SW	Speedup Improvement
5XOR 5AND	18677	8.20	-
10XOR 10AND	14888	10.29	1.25
15XOR 15AND	12252	12.50	1.22

64-bit multiplication problem. 300 MHz main clock and 200 MHz local clock.

Table 4.17: Using 2 Address Registers for 3 Addresses

Problem	1 Reg as 1 address	2 Regs as 3 addresses	Improvement	Total Speedup
2 pr	41044	37358	1.1	12.47
3 pr	66409	58587	1.13	10.27
4 pr	90087	7.83	1.06	7.83

10AND and 10XOR overlay cells; 300 MHz main clock and 200 MHz local clock.

So we can not apply xor without synchronization stages between the host and FPGA. The solution is to use multiple XORs to improvement the total performance. Table 4.16 shows the results of using 5 AND and 5 XOR overlay cells; 10 AND and 10 XOR; 15 AND and 15 XOR for the speedup. The speedup improvement shows that the increase from changing from 5 to 10 is 1.25 times and changing from 10 to 15 is 1.22 times.

We also check the speedup of using 2 register addresses for 3 addresses in Table 4.17. We use page-ranking examples and the results show 1.06 to 1.13 speedup improvement compared with the method of using 1 register for 1 address. However, as we mentioned in Table 4.17 this only applies to relatively small problems because of the valid bitwidths it supports.

In Table 4.18, we show all the problems that we tackle for the purpose of accelerating garbled circuit protocols using ProceV as the heterogeneous reconfigurable system. Speed-up is the division between software time and our system's running time. These results have applied the following optimizations: (1) 15 AND overlay cells and 15 XOR overlay cells; (2) Hybrid memory system with the directly-used policy; (3) Maximum 300 MHz main clock frequency for PCIe interface and maximum 200 MHz local clock frequency; (4) Pipelining synchronization between the host and FPGA. We show significant speedup compared with software platform FlexSC for all the problems.

Table 4.18: Speedup Results

Problem	sw (ms)	Time (us)	Speedup
6-bit adder	2.06	45	45.78
10-bit HD	2.53	80	31.63
30-bit HD	4.08	171	23.86
50-bit HD	6.46	259	24.94
8-bit multi	9.22	293	31.47
16-bit multi	14.54	949	15.32
32-bit multi	33.76	3308	10.21
64-bit multi	153.13	12252	12.50
10 4-bit sort	21.12	2339	9.03
5 5 4-bit m_mult	60.66	5830	10.40
10 10 4-bit m_mult	220.81	11286	19.56
5 5 8-bit m_mult	203.86	24128	8.45
10 10 8-bit m_mult	1060.63	170895	6.21
20 20 4-bit m_mult	2170.88	340698	6.37

Chapter 5

Conclusion and Future work

5.1 Conclusion

This dissertation demonstrates a heterogeneous reconfigurable computing system using FPGA overlay architecture for general garbled circuit operations. This is the first heterogeneous computing system using FPGAs for garbled circuit problems so far. This system lets the user implement and accelerate their application without knowing any knowledge of either hardware development or secure function evaluation protocol by providing a complete workflow to transfer any garbled circuit problem onto it. We demonstrate the benefit of using this system by showing significant speedup compared with existing software platforms.

For the hardware architecture on FPGA, unlike the tailored approaches that are tied to the execution of a specific SFE structure which requires full reprogramming of an FPGA with each new execution, our design uses a coarse grained overlay architecture and enables the evaluation of different SFE tasks without the need for reprogramming.

The host side workflow includes garbled circuit problem generator, problem parser, and host code generation tools which can be configurable for different hardware architectures. These tools explore the parallelism for any GC problem and generate the host program based on the structure of the problem. We also provide analytical tools to show the different characteristics of a problem.

We explore the bottlenecks while working on this heterogeneous reconfigurable computing system using ProceV board and tackle them using different solutions. This exploration can give future researchers valuable data and vision for improving their own heterogeneous systems.

This system is the first heterogeneous computing system using FPGAs for accelerating general garbled circuit problems and this research makes possible the wider adoption of using garbled circuit schemes in the future.

5.2 Future Work

Future work of this project contains several aspects. First is the further improvement of the existing heterogeneous system in Section 5.2.1. Section 5.2.2 talks about expanding the scale of the system to a cluster of nodes, each of which features one ProceV board. The third is to choose other heterogeneous system using FPGAs, discussed in Section 5.2.3.

5.2.1 Performance Improvement within one Node

First is the further improvement of the performance of the heterogeneous system using one ProceV board on one node. As analyzed in this dissertation, the performance suffers from the interconnection delay between host and FPGAs, especially PCIe communication and DDR memory access.

One direction is to expand the overlay cell library to abstract more complicated computational patterns using Boolean AND and XOR operations. The current work uses garbled circuit AND and XOR overlay cells as two components of the hardware architecture library, which guarantees that any problem that can terminate in finite time can map onto those two overlays. This fine grain of operation suffers from the DDR access delay in every Boolean operation. Based on Table 4.7 and 4.8, we already know that there are many 1-to-1 wires which are to be used in the next layer. Even if we use embedded RAM for storing those data using directly-used policy, there is still memory access time for that wire.

One possible solution is to build other overlay cells which consist of the Boolean operations across two layers. For example, there are several Boolean functions which are $E = A \oplus B$, $F = C \oplus D$ and $G = E \oplus F$. If the output wire E of function $E = A \oplus B$ is only used in the next layer function $G = E \oplus F$, we can combine the three functions into $G = (A \oplus B) \oplus (C \oplus D)$. By doing this, we save the time of storing the garbled value E to the memory either on-chip or off-chip. This improves the timing performance for this operation. Also, each XOR operation in this function can be replaced by AND operation, making total 6 variations (Notice that equation $G = (A \wedge B) \oplus (C \oplus D)$ and $G = (A \oplus B) \oplus (C \wedge D)$ are the same in hardware architecture). If there are many operation patterns among those 6 variations in a garbled circuit problem, total time saving could be considerable. This also requires redesigning the problem parser and running time will benefit from extra preprocessing time which is a good tradeoff. One more thing is that since there is no gate with both input wires from the previous adjacent layer, there will be at least one of the two operations needed to be stored in memory for future use. In the previous example, at least one of two wires E and F has to be written back to memory.

5.2.2 Map Garbled Circuit Problem onto Multiple Nodes

Second is to separate a large problem into several small problems which can be computed independently through several host nodes each with its own ProceV board. This enables the expansion of the size of the problems into even larger data mining problems, such as page ranking with more nodes using GraphSC and eventually provide a large, scalable, efficient platform for privacy preserving computation.

In detail, the communication within each node in the data center is still PCIe interface. However, the connections between the hosts can have several options, such as Ethernet interface, Interlaken networking, etc. Ethernet interface is a standard connection for connecting the host in today's data center. Interlaken networking interface can be applied for direct FPGA to FPGA communication, which could potentially avoid the high latency communication channel through PCIe and Ethernet interface. The challenge of multiple nodes is the separation of the problems and data storage. Since we want to separate large problems into smaller tasks suitable for one node, clear boundaries should be defined between different tasks. Also, for each task, there should be less dependency between each other. In other words, the task should be low in coupling and high in cohesion. For data storage, each host along with FPGA have their own memory space and data should not be shared unless necessary. In the future, universal memory space could alleviate the data sharing challenge for the garbled circuit, but this requires infrastructure support. The future work will be continued thanks to the NSF funding for building a "Massively Scalable Secure Computation Infrastructure Using FPGAs" [19, 17, 76].

5.2.3 Other Platform Exploration for Garbled Circuit

Another potential research direction includes exploring other platforms for better performance and scalability. For example, Amazon launched the AWS EC2 F1 instance in April this year (2017) which provides the instance of using one or multiple FPGAs as an accelerator within its own Amazon Web Services (AWS) cloud infrastructure. F1 instances feature Intel Boardwell E5 processor with up to 976 GB of memory, up to 4 TB of SSD storage and also one to eight Xilinx

UltraScale+ VU9P FPGAs and this provides plenty of resources for the exploration of even larger garbled circuit problems. There is 64 GB memory on a 288 bitwide four DDR4 interface and the connection between host and FPGA is PCIe x16. These parameters are better than the ProceV board which features 8 GB on board memory with DDR3 interface and PCIe x8 lane for the host to FPGA connection. For instances with more than one FPGA, there is dedicated PCIe fabric which allows FPGAs to share the same memory address space and communicate with each other at up to 12 GB/s in each direction. Given the support of faster DDR and PCIe interface and customized size of register access from the host to FPGA, those bottlenecks of our current heterogeneous reconfigurable system using ProceV board can be further alleviated.

Appendix A

Example of Garbled Circuit with Garbled Values

Here is an example of generating the garbled circuit with garbled value from FlexSC generated using Millionaire problems with 4-bit input. This information is useful for constructing the testbench for the hardware design. Each wire is represented as 10 8-bit value and the total is 80 bits. The type of wires includes input, AND and XOR operations. In the generated output, garbled value on each wire is composed of 10 8-bit values which make the total size 80 bits. In this example, Alice is the garbled circuit generator with four-bit input, which is false, false, true false. The sequence ”-87 -78 81 -34 84 -68 87 -121 -27 116” represents the false value of Alice’s input bit 0. The type of the Boolean gates is either AND or XOR with its own gate ID. Each gate consists three GCSignal wire ID, which represent the location of garbled value stored in memory. In this example, the first XOR gate has input ID 2 and 101, output ID 1175. Those generated details become the testbench with which we verify the correctness of hardware design.

```
Input of Alice:
0(false)
-87 -78 81 -34 84 -68 87 -121 -27 116
1(true)
0 -90 115 12 -99 -3 42 126 22 -19
2(false)
-55 65 -103 -107 -27 -77 62 -81 -121 -112
3(false)
-54 -115 -102 -42 15 103 -42 118 -76 36
----------------------------------------
****************************************
```

```
 XOR GATE ID: 1
 GCSignal ID: 2
 25 -6 54 115 14 100 1 12 -125 106
 GCSignal ID: 101
-73 110 76 -65 -58 28 -128 125 16 38
 GCSignal ID: 1175
-82 -108 122 -52 -56 120 -127 113 -109 76
 *****************************************
 *****************************************
 XOR GATE ID: 2
 GCSignal ID: 2
 25 -6 54 115 14 100 1 12 -125 106
 GCSignal ID: 103
 122 73 -127 -48 -66 -26 -50 23 10 -58
 GCSignal ID: 1176
 99 -77 -73 -93 -80 -126 -49 27 -119 -84
 *****************************************
 *****************************************
 XOR GATE ID: 3
 GCSignal ID: 2
 25 -6 54 115 14 100 1 12 -125 106
 GCSignal ID: 105
 20 -78 -1 44 82 -108 50 -54 -5 59
 GCSignal ID: 1177
 13 72 -55 95 92 -16 51 -58 120 81
 *****************************************
 *****************************************
 XOR GATE ID: 4
 GCSignal ID: 2
 25 -6 54 115 14 100 1 12 -125 106
 GCSignal ID: 107
-102 95 20 50 9 -10 53 34 79 81
 GCSignal ID: 1178
-125 -91 34 65 7 -110 52 46 -52 59
 *****************************************
 *****************************************
 XOR GATE ID: 5
 GCSignal ID: 2
 25 -6 54 115 14 100 1 12 -125 106
 GCSignal ID: 93
-87 -78 81 -34 84 -68 87 -121 -27 116
 GCSignal ID: 1180
-80 72 103 -83 90 -40 86 -117 102 30
 *****************************************
 *****************************************
 XOR GATE ID: 6
 GCSignal ID: 2
 25 -6 54 115 14 100 1 12 -125 106
 GCSignal ID: 1175
-82 -108 122 -52 -56 120 -127 113 -109 76
 GCSignal ID: 1181
-73 110 76 -65 -58 28 -128 125 16 38
 *****************************************
 *****************************************
 XOR GATE ID: 7
 GCSignal ID: 93
-87 -78 81 -34 84 -68 87 -121 -27 116
 GCSignal ID: 1181
-73 110 76 -65 -58 28 -128 125 16 38
 GCSignal ID: 1182
 30 -36 29 97 -110 -96 -41 -6 -11 82
 *****************************************
 *****************************************
 GCGenz And Gate ID is : 1
 GCsignal a :
-80 72 103 -83 90 -40 86 -117 102 30
 GCsignal b :
-73 110 76 -65 -58 28 -128 125 16 38
 GCsignal res :
-38 57 -93 -18 94 107 75 13 50 85
 *****************************************
 *****************************************
 XOR GATE ID: 8
 GCSignal ID: 2
 25 -6 54 115 14 100 1 12 -125 106
 GCSignal ID: 1183
-38 57 -93 -18 94 107 75 13 50 85
 GCSignal ID: 1184
-61 -61 -107 -99 80 15 74 1 -79 63
 *****************************************
 *****************************************
 XOR GATE ID: 9
 GCSignal ID: 95
 25 92 69 127 -109 -103 43 114 -107 -121
 GCSignal ID: 1184
-61 -61 -107 -99 80 15 74 1 -79 63
 GCSignal ID: 1185
-38 -97 -48 -30 -61 -106 97 115 36 -72
```

```
 ****************************************
 ****************************************
 XOR GATE ID: 10
 GCSignal ID: 1176
 99 -77 -73 -93 -80 -126 -49 27 -119 -84
 GCSignal ID: 1184
-61 -61 -107 -99 80 15 74 1 -79 63
 GCSignal ID: 1186
-96 112 34 62 -32 -115 -123 26 56 -109
 ****************************************
 ****************************************
 XOR GATE ID: 11
 GCSignal ID: 95
 25 92 69 127 -109 -103 43 114 -107 -121
 GCSignal ID: 1186
-96 112 34 62 -32 -115 -123 26 56 -109
 GCSignal ID: 1187
-71 44 103 65 115 20 -82 104 -83 20
 ****************************************
 ****************************************
 GCEva And Gate ID is : 2
 GCsignal a :
-38 -97 -48 -30 -61 -106 97 115 36 -72
 GCsignal b :
-96 112 34 62 -32 -115 -123 26 56 -109
 GCsignal res :
-38 57 -93 -18 94 107 75 13 50 85
 ****************************************
 ****************************************
 XOR GATE ID: 12
 GCSignal ID: 1184
-61 -61 -107 -99 80 15 74 1 -79 63
 GCSignal ID: 1188
-38 57 -93 -18 94 107 75 13 50 85
 GCSignal ID: 1189
 25 -6 54 115 14 100 1 12 -125 106
 ****************************************
 ****************************************
 XOR GATE ID: 13
 GCSignal ID: 97
-55 65 -103 -107 -27 -77 62 -81 -121 -112
 GCSignal ID: 1189
 25 -6 54 115 14 100 1 12 -125 106
 GCSignal ID: 1190
```

```
-48 -69 -81 -26 -21 -41 63 -93 4 -6
 *****************************************
 *****************************************
 XOR GATE ID: 14
 GCSignal ID: 1177
 13 72 -55 95 92 -16 51 -58 120 81
 GCSignal ID: 1189
 25 -6 54 115 14 100 1 12 -125 106
 GCSignal ID: 1191
 20 -78 -1 44 82 -108 50 -54 -5 59
 *****************************************
 *****************************************
 XOR GATE ID: 15
 GCSignal ID: 97
-55 65 -103 -107 -27 -77 62 -81 -121 -112
 GCSignal ID: 1191
 20 -78 -1 44 82 -108 50 -54 -5 59
 GCSignal ID: 1192
-35 -13 102 -71 -73 39 12 101 124 -85
 *****************************************
 *****************************************
 GCEva And Gate ID is : 3
 GCsignal a :
-48 -69 -81 -26 -21 -41 63 -93 4 -6
 GCsignal b :
 20 -78 -1 44 82 -108 50 -54 -5 59
 GCsignal res :
-38 57 -93 -18 94 107 75 13 50 85
 *****************************************
 *****************************************
 XOR GATE ID: 16
 GCSignal ID: 1189
 25 -6 54 115 14 100 1 12 -125 106
 GCSignal ID: 1193
-38 57 -93 -18 94 107 75 13 50 85
 GCSignal ID: 1194
-61 -61 -107 -99 80 15 74 1 -79 63
 *****************************************
 *****************************************
 XOR GATE ID: 17
 GCSignal ID: 99
-54 -115 -102 -42 15 103 -42 118 -76 36
 GCSignal ID: 1194
-61 -61 -107 -99 80 15 74 1 -79 63
```

```
 GCSignal ID: 1195
 9 78 15 75 95 104 -100 119 5 27
 ****************************************
 ****************************************
 XOR GATE ID: 18
 GCSignal ID: 1178
-125 -91 34 65 7 -110 52 46 -52 59
 GCSignal ID: 1194
-61 -61 -107 -99 80 15 74 1 -79 63
 GCSignal ID: 1196
 64 102 -73 -36 87 -99 126 47 125 4
 ****************************************
 ****************************************
 XOR GATE ID: 19
 GCSignal ID: 99
-54 -115 -102 -42 15 103 -42 118 -76 36
 GCSignal ID: 1196
 64 102 -73 -36 87 -99 126 47 125 4
 GCSignal ID: 1197
-118 -21 45 10 88 -6 -88 89 -55 32
 ****************************************
 ****************************************
 GCEva And Gate ID is : 4
 GCsignal a :
 9 78 15 75 95 104 -100 119 5 27
 GCsignal b :
 64 102 -73 -36 87 -99 126 47 125 4
 GCsignal res :
-38 57 -93 -18 94 107 75 13 50 85
 ****************************************
 ****************************************
 XOR GATE ID: 20
 GCSignal ID: 1194
-61 -61 -107 -99 80 15 74 1 -79 63
 GCSignal ID: 1198
-38 57 -93 -18 94 107 75 13 50 85
 GCSignal ID: 1199
 25 -6 54 115 14 100 1 12 -125 106
 ****************************************
 ****************************************
 XOR GATE ID: 21
 GCSignal ID: 2
 25 -6 54 115 14 100 1 12 -125 106
 GCSignal ID: 1197
```

```
-118 -21 45 10 88 -6 -88 89 -55 32
 GCSignal ID: 1200
-109 17 27 121 86 -98 -87 85 74 74
 *****************************************
```

Appendix B

Interface Between Host and ProceV Board

Here are the steps of how to construct the host code for the heterogeneous reconfigurable system using ProceV:

1. Initialize an object of driver D
2. Create handle H for the memory M allocated for DMA transition
3. Fill the initial values into memory M by randomly generated value
4. Run DMA through the function of driver: D→runDMA(H)
5. Write global register such as R value via driver D
6. For each layer:
   write in out registers for each AND and XOR overlays
   write registers for control such as layer information
7. Iterate until all the layer are processed

List of Acronyms

AES	Advanced Encryption Standard
ALM	Adaptive Logic Module
ASIC	Application-Specific Integrated Circuit
AWS	Amazon Web Services
BRAM	Block Random Access Memory
CBC	Cipher Block Chaining
CPU	Central Processing Unit
DAG	Direct Acyclic Graph
DDR	Double Data Rate
DMA	Direct Memory Access
DMA	Direct Memory Access
DSP	Digital Signal Processor
ECB	Electronic Codebook
ECC	Elliptic Curve Cryptographic
FPGA	Field Programmable Gate Array
GC	Garbled Circuits
GPU	Graphics Processing Unit

HDK	Hardware Development Kit
HDL	Hardware Description Language
IP	Intellectual Property
LAB	Logic Array Blocks
LUT	Look Up Table
MIMD	Multiple Instruction Multiple Data
MLAB	Memory Logic Array Blocks
NoC	Network on a Chip
PCIe	Peripheral Component Interconnect Express
SDRAM	Synchronous Dynamic Random-Access Memory
SFDL	Secure Function Definition Language
SFE	Secure Function Evaluation
SHA	Secure Hash Algorithm
SHDL	Secure Hardware Description Language
SIMD	Single Instruction Multiple Data
SODIMM	Small Outline Dual In-line Memory Module
SOF	SRAM Object File

Bibliography

[1] Altera, "Stratix V Device Handbook," https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/hb/stratix-v/stx5_core.pdf.

[2] Gidel, "ProceV," http://gidel.com/ProceV.html.

[3] A. Yao, "How to generate and exchange secrets," in *Foundations of Computer Science, 1986., 27th Annual Symposium on*, 1986, pp. 162–167.

[4] J. Angwin, "The web's new gold mine: Your secretsa journal investigation finds that one of the fastest-growing businesses on the internet is the business of spying on consumers; first in a series," *Wall Street Journal*, 2010.

[5] M. Ramos-Casals, P. Brito-Zerón, B. Kostov, A. Sisó-Almirall, X. Bosch, D. Buss, A. Trilla, J. H. Stone, M. A. Khamashta, and Y. Shoenfeld, "Google-driven search for big data in autoimmune geoepidemiology: Analysis of 394,827 patients with systemic autoimmune diseases," *Autoimmunity reviews*, 2015.

[6] L. A. Adamic and N. Glance, "The political blogosphere and the 2004 US election: divided they blog," in *Proceedings of the 3rd international workshop on Link discovery*. ACM, 2005, pp. 36–43.

[7] S. Ressler, "Social network analysis as an approach to combat terrorism: past, present, and future research," *Homeland Security Affairs*, vol. 2, no. 2, pp. 1–10, 2006.

[8] M. Kosinski, D. Stillwell, and T. Graepel, "Private traits and attributes are predictable from digital records of human behavior," *Proceedings of the National Academy of Sciences*, vol. 110, no. 15, pp. 5802–5805, 2013.

[9] S. Salamatian, A. Zhang, F. du Pin Calmon, S. Bhamidipati, N. Fawaz, B. Kveton, P. Oliveira, and N. Taft, "How to hide the elephant-or the donkey-in the room: Practical privacy against statistical inference for large data," in *GlobalSIP*, 2013.

[10] U. Weinsberg, S. Bhagat, S. Ioannidis, and N. Taft, "Blurme: Inferring and obfuscating user gender based on ratings," in *RecSys*, 2012.

[11] S. Bhagat, I. Rozenbaum, and G. Cormode, "Applying link-based classification to label blogs," in *WebKDD*, 2007.

[12] A. Mislove, B. Viswanath, K. P. Gummadi, and P. Druschel, "You are who you know: Inferring user profiles in Online Social Networks," in *WSDM*, 2010.

[13] J. Otterbacher, "Inferring gender of movie reviewers: exploiting writing style, content and metadata," in *CIKM*, 2010.

[14] D. Rao, D. Yarowsky, A. Shreevats, and M. Gupta, "Classifying latent user attributes in twitter," in *SMUC*, 2010.

[15] A. Narayanan and V. Shmatikov, "Robust de-anonymization of large sparse datasets," in *OAKLAND*, 2008.

[16] J. Wortham, "Facebook and privacy clash again," *The New York Times May*, vol. 6, 2010.

[17] V. Nikolaenko, U. Weinsberg, S. Ioannidis, M. Joye, D. Boneh, and N. Taft, "Privacy-preserving ridge regression on hundreds of millions of records," in *IEEE S & P*, 2013.

[18] V. Nikolaenko, S. Ioannidis, U. Weinsberg, M. Joye, N. Taft, and D. Boneh, "Privacy-preserving matrix factorization," in *ACM CCS*, 2013.

[19] K. Nayak, X. S. Wang, S. Ioannidis, U. Weinsberg, N. Taft, and E. Shi, "GraphSC: Parallel secure computation made easy," in *IEEE S & P*, 2015.

[20] M. Beye, Z. Erkin, and R. L. Lagendijk, "Efficient privacy preserving k-means clustering in a three-party setting," in *IEEE International Workshop on Information Forensics and Security*. IEEE Press, 2011.

[21] W. Du, Y. S. Han, and S. Chen, "Privacy-preserving multivariate statistical analysis: Linear regression and classification," in *4th SIAM International Conference on Data Mining (SDM 2004)*. SIAM, 2004.

[22] Y. Huang, L. Malka, D. Evans, and J. Katz, "Efficient privacy-preserving biometric identification," in *NDSS*, 2011.

[23] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray *et al.*, "A reconfigurable fabric for accelerating large-scale datacenter services," in *Computer Architecture (ISCA), 2014 ACM/IEEE 41st International Symposium on*. IEEE, 2014, pp. 13–24.

[24] A. M. Caulfield, E. S. Chung, A. Putnam, H. Angepat, J. Fowers, M. Haselman, S. Heil, M. Humphrey, P. Kaur, J.-Y. Kim *et al.*, "A cloud-scale acceleration architecture," in *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*. IEEE, 2016, pp. 1–13.

[25] "Amazon ec2 f1 instances," https://aws.amazon.com/ec2/instance-types/f1/, accessed: 2017-06.

[26] "FlexSC," https://github.com/wangxiao1254/FlexSC, accessed: 2017-08-01.

[27] C. Liu, X. S. Wang, K. Nayak, Y. Huang, and E. Shi, "Oblivm: A generic, customizable, and reusable secure computation architecture," in *IEEE S & P*, 2015.

[28] X. Fang, S. Ioannidis, and M. Leeser, "Secure function evaluation using an fpga overlay architecture," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 2017, pp. 257–266.

[29] ——, "Garbled circuits for preserving privacy in the datacenter," in *International Workshop on Heterogeneous High-performance Reconfigurable Computing (H2RC'16)*, 2016.

[30] A. Yao, "How to generate and exchange secrets," in *FOCS*, 1986.

[31] M. Naor, B. Pinkas, and R. Sumner, "Privacy preserving auctions and mechanism design," in *1st ACM Conference on Electronic Commerce*, 1999.

[32] V. Nikolaenko, U. Weinsberg, S. Ioannidis, M. Joye, D. Boneh, and N. Taft, "Privacy-preserving ridge regression on hundreds of millions of records," in *IEEE S&P*, 2013.

[33] M. O. Rabin, "How to exchange secrets by oblivious transfer," Aiken Computation Laboratory, Harvard University, Tech. Rep. TR-81, 1981.

[34] S. Even, O. Goldreich, and A. Lempel, "A randomized protocol for signing contracts," *Commun. ACM*, vol. 28, no. 6, 1985.

[35] D. Beaver, S. Micali, and P. Rogaway, "The round complexity of secure protocols," in *Proceedings of the twenty-second annual ACM symposium on Theory of computing*. ACM, 1990, pp. 503–513.

[36] V. Kolesnikov and T. Schneider, "Improved Garbled Circuit: Free XOR Gates and Applications," in *ICALP*, 2008.

[37] P. FIPS, "180-4," *Federal Information Processing Standards Publication, Secure Hash*, 2012.

[38] J. Strömbergson, "SHA1 core," https://github.com/secworks/sha1.

[39] X. Fang, P. Luo, Y. Fei, and M. Leeser, "Balance power leakage to fight against side-channel analysis at gate level in fpgas," in *IEEE ASAP 2015 Conference*, 2015.

[40] ——, "Leakage evaluation on power balance countermeasure against side-channel attack on fpgas," in *High Performance Extreme Computing Conference (HPEC), 2015 IEEE*. IEEE, 2015, pp. 1–6.

[41] P. Luo, Y. Fei, X. Fang, A. A. Ding, D. R. Kaeli, and M. Leeser, "Side-channel analysis of mac-keccak hardware implementations." *IACR Cryptology ePrint Archive*, vol. 2015, p. 411, 2015.

[42] P. Luo, Y. Fei, X. Fang, A. A. Ding, M. Leeser, and D. R. Kaeli, "Power analysis attack on hardware implementation of mac-keccak on fpgas," in *ReConFigurable Computing and FPGAs (ReConFig), 2014 International Conference on*. IEEE, 2014, pp. 1–7.

[43] A. Brant and G. G. Lemieux, "ZUMA: An open FPGA overlay architecture," in *Field-Programmable Custom Computing Machines (FCCM), 2012 IEEE 20th Annual International Symposium on*. IEEE, 2012, pp. 93–96.

[44] T. Wiersema, A. Bockhorn, and M. Platzner, "Embedding fpga overlays into configurable systems-on-chip: Reconos meets zuma," in *ReConFigurable Computing and FPGAs (ReConFig), 2014 International Conference on*. IEEE, 2014, pp. 1–6.

[45] N. Kapre, N. Mehta, M. Delorimier, R. Rubin, H. Barnor, M. J. Wilson, M. Wrighton, and A. Dehon, "Packet switched vs. time multiplexed FPGA overlay networks," in *Field-Programmable Custom Computing Machines, 2006. FCCM'06. 14th Annual IEEE Symposium on*. IEEE, 2006, pp. 205–216.

[46] N. Kapre and J. Gray, "Hoplite: Building austere overlay NoCs for FPGAs," in *Field Programmable Logic and Applications (FPL), 2015 25th International Conference on*. IEEE, 2015, pp. 1–8.

[47] D. Koch, C. Beckhoff, and G. G. Lemieux, "An efficient FPGA overlay for portable custom instruction set extensions," in *Field Programmable Logic and Applications (FPL), 2013 23rd International Conference on*. IEEE, 2013, pp. 1–8.

[48] A. K. Jain, S. A. Fahmy, and D. L. Maskell, "Efficient overlay architecture based on dsp blocks," in *Field-Programmable Custom Computing Machines (FCCM), 2015 IEEE 23rd Annual International Symposium on*. IEEE, 2015, pp. 25–28.

[49] A. K. Jain, D. L. Maskell, and S. A. Fahmy, "Are coarse-grained overlays ready for general purpose application acceleration on fpgas?" in *Proceedings of IEEE International Conference on Pervasive Intelligence and Computing*. IEEE, 2016.

[50] M. Showerman, J. Enos, A. Pant, V. Kindratenko, C. Steffen, R. Pennington, W.-m. Hwu *et al.*, "Qp: A heterogeneous multi-accelerator cluster," in *Proc. 10th LCI International Conference on High-Performance Clustered Computing*, 2009.

[51] A. George, H. Lam, and G. Stitt, "Novo-G: At the forefront of scalable reconfigurable supercomputing," *Computing in Science and Engineering*, vol. 13, no. 1, pp. 82–86, 2011.

[52] D. Malkhi, N. Nisan, B. Pinkas, and Y. Sella, "Fairplay—a secure two-party computation system," in *Proceedings of the 13th Conference on USENIX Security Symposium - Volume 13*, ser. SSYM'04, 2004.

[53] A. C.-C. Yao, "Protocols for secure computations," in *FOCS*, vol. 82, 1982, pp. 160–164.

[54] M. Bellare, V. T. Hoang, S. Keelveedhi, and P. Rogaway, "Efficient garbling from a fixed-key blockcipher," in *IEEE Symposium on Security and Privacy (SP)*, 2013.

[55] S. Zahur, M. Rosulek, and D. Evans, "Two halves make a whole," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2015, pp. 220–250.

[56] Y. Huang, D. Evans, J. Katz, and L. Malka, "Faster secure two-party computation using garbled circuits," in *USENIX Security*, 2011.

[57] L. Malka, "Vmcrypt: modular software architecture for scalable secure computation," in *Proceedings of the 18th ACM conference on Computer and communications security*. ACM, 2011, pp. 715–724.

[58] W. Henecka and T. Schneider, "Faster secure two-party computation with less memory," in *Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security*. ACM, 2013, pp. 437–446.

[59] D. Demmler, T. Schneider, and M. Zohner, "Aby-a framework for efficient mixed-protocol secure two-party computation." in *NDSS*, 2015.

[60] W. Henecka, S. Kögl, A.-R. Sadeghi, T. Schneider, and I. Wehrenberg, "TASTY: tool for automating secure two-party computations," in *CCS*, 2010.

[61] B. Kreuter, A. Shelat, and C.-H. Shen, "Billion-gate secure computation with malicious adversaries," in *USENIX Security*, 2012.

[62] C.-h. Shen *et al.*, "Two-output secure computation with malicious adversaries," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2011, pp. 386–405.

[63] B. Kreuter, B. Mood, A. Shelat, and K. Butler, "PCF: A portable circuit format for scalable two-party secure computation," in *USENIX Security*, 2013.

[64] A. Holzer, M. Franz, S. Katzenbeisser, and H. Veith, "Secure two-party computations in ansi c," in *CCS*, 2012.

[65] B. Mood, D. Gupta, H. Carter, K. Butler, and P. Traynor, "Frigate: A validated, extensible, and efficient compiler and interpreter for secure computation," in *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2016, pp. 112–127.

[66] A. Schropfer, F. Kerschbaum, and G. Muller, "L1 - an intermediate language for mixed-protocol secure computation," in *35th Annual Computer Software and Applications Conference (COMPSAC)*. IEEE Computer Society, 2011.

[67] A. Rastogi, M. A. Hammer, and M. Hicks, "Wysteria: A programming language for generic, mixed-mode multiparty computations," in *2014 IEEE Symposium on Security and Privacy*. IEEE, 2014, pp. 655–670.

[68] E. M. Songhori, S. U. Hussain, A.-R. Sadeghi, T. Schneider, and F. Koushanfar, "Tinygarble: Highly compressed and scalable sequential garbled circuits," in *IEEE S & P*, 2015.

[69] E. M. Songhori, S. Zeitouni, G. Dessouky, T. Schneider, A.-R. Sadeghi, and F. Koushanfar, "Garbledcpu: a mips processor for secure computation in hardware," in *Proceedings of the 53rd Annual Design Automation Conference*. ACM, 2016, p. 73.

[70] S. Pu, P. Duan, and J.-C. Liu, "Fastplay-a parallelization model and implementation of smc on cuda based gpu cluster architecture." *IACR Cryptology ePrint Archive*, vol. 2011, p. 97, 2011.

[71] T. K. Frederiksen, T. P. Jakobsen, and J. B. Nielsen, "Faster maliciously secure two-party computation using the gpu," in *International Conference on Security and Cryptography for Networks*. Springer, 2014, pp. 358–379.

[72] N. Husted, S. Myers, A. Shelat, and P. Grubbs, "GPU and CPU parallelization of honest-but-curious secure two-party computation," in *CSAC*, 2013.

[73] K. Järvinen, V. Kolesnikov, A.-R. Sadeghi, and T. Schneider, "Garbled circuits for leakage-resilience: Hardware implementation and evaluation of one-time programs," in *Cryptographic Hardware and Embedded Systems, CHES 2010*. Springer, 2010, pp. 383–397.

[74] V. Kolesnikov and T. Schneider, "Improved garbled circuit: Free xor gates and applications," in *Proceedings of the 35th International Colloquium on Automata, Languages and Programming, Part II*, ser. ICALP '08, 2008, pp. 486–498.

[75] C. Liu, X. S. Wang, K. Nayak, Y. Huang, and E. Shi, "Oblivm: A programming framework for secure computation," in *Security and Privacy (SP), 2015 IEEE Symposium on*, 2015, pp. 359–376.

[76] V. Nikolaenko, S. Ioannidis, U. Weinsberg, M. Joye, N. Taft, and D. Boneh, "Privacy-preserving matrix factorization," in *ACM CCS*, 2013.