FPGA of Acceleration of Stochastic Simulation

A Design Project Report

Presented to the School of Electrical and Computer Engineering of

Cornell University

In Partial Fulfillment of the Requirements for the Degree of

Master of Engineering, Electrical and Computer Engineering

Submitted by

Tian Gao (tg293)

MEng Field Advisor: Bruce Robert Land

Degree Date January 2014

Abstract

Master of Engineering Program

School of Electrical and Computer Engineering

Cornell University

Design Project Report

Project Title: FPGA of Acceleration of Stochastic Simulation

Author: Tian Gao

Abstract:

This project is designed to implement a stochastic algorithm on an FPGA system to accelerate a biological simulation. A fast national disease spread prediction algorithm is needed to anticipate the tendency of an infectious disease to spread in a given geography. Using a traditional computing system, the simulation would require large execution time as the model size grows. The project explores methods for implementing such an algorithm on an FPGAsystem. By utilizing the parallel features of an FPGA, we may find a way to run the whole simulation in real time. In this project, a MATLAB simulation is firstly designed for a hardware compatible algorithm for an SIR model, which is a basic disease spread model. Then, three kinds of FPGAs are used to implement the algorithm. Finally, some verification is applied to validate the result. The project has successfully implemented a hardware compatible algorithm for an SIR model on an FPGA system.

Executive Summary

The project is designed to parallelize a stochastic algorithm for solving an SIR model so that it can be implemented on an FPGA system. It explores a method to implement the parallelizable stochastic algorithm on an FPGA, where anaccurate random number generator is easier to build and all the computations can be executed at the same time. In this way, we reduce the complexity in time so we can actually simulate more individuals in real time, which contributes to making a large scale stochastic simulation system.

The MATLAB simulation was tested before the algorithm is actually implemented on the board. To make the algorithm compatible with hardware, we use a unified clock for every execution which makes the system discrete instead of continuous. This implementation fits the FPGA system well, where a universal clock is used to trigger the registers.

After validating the algorithm on MATLAB, we implement the first version on a TerasicDE2 board. The parameters of the SIR model are set in MATLAB code and a random network is generated in MATLAB. The associated Verilog code is automatically written by file print function in MATLAB. The seeds of hardware random number generator are randomly assigned in MATLAB as well. Therefore,every time the MATLAB code is run, there is a new network and a new random number generator set generated.

The algorithm executes successfully on the DE2 board, generating the results on a VGA screen. We expanded the algorithm to two other boards: DE2-115 and DE2i-150 for a larger scale of network. The maximum of individuals we can simulate on DE2i-150 is 140 individuals with 14 connections in average for each individual. We also have done some verificationbetween the hardware implementation vs. software simulation and the random number generator. The project goals are completeand potential remains for future development.

CONTENTS

1 INTRODUCTION

As the performance of the computing systems improves, researchers are trying to predict complicated, real life situations using stochastic algorithms. However, it is becoming more difficult to execute a large network in serial processors because of the large scale of data and nodes to compute. Thus, parallelization tends to be the new solution to solve such stochastic simulations. An FPGA system provides an excellent platform to explore such parallel solutions. Instead of using traditional CPUs as the main processors, An FPGA system executes the simulation using compiled HDL code running on FPGA hardware using register and logic elements. In this way, increasing scale of the simulation leads to a tradeoff between FPGA area and simulation execution time. Evena relatively large network with many nodes can be computed on an FPGA system in real time, as long as the FPGA has enough logic elements.

In this project, we explore a way to implement a stochastic simulation;an SIR model on an FPGA system. The Verilog code is automatically generated by MATLAB code, by which parameters of the model are easily modified and a whole new random network is built for every MATLAB run. The simulation on the FPGA system can execute with the initial seeds for every random number generator, which would give out the exact result for every run, or it can be based on the seeds determined by time, which would result in a different output for the same network. The output is displayed on a VGA screen, showing the figure of infected individuals vs. time.

2 DESIGN AND IMPLEMENTATION

2.1 SIR MODEL

There are many different algorithms for stochastic models. We choose the SIR model for the simulation becausethe SIR model (representing Susceptible, Infectious and Recovered) is a basic, direct model that is widely used for modeling large scale disease outbreaks. It is a good model for initial investigation and is compatible with hardware design.

In the SIR model, initially, all individuals are susceptible which means they can be infected by other individuals. Then, several individuals are set to be infectious at the beginning of the simulation. Each individual has its own network, providing interconnection with other individuals in the simulation. Then, with these connections, the disease spreads. A susceptible individual connected to an infectious one could become infected. Also, the infectious individual could recover from the disease. In this particular model, the recovered individual cannot be infected again. By observing the number of individuals of each group, we can see the trends of the disease, which also provides validation. For the simulation, if our results are correct, the number figure should be like the following graph, where blue dots represent susceptible individuals, green represents infectious individuals and red represents recovered individuals.

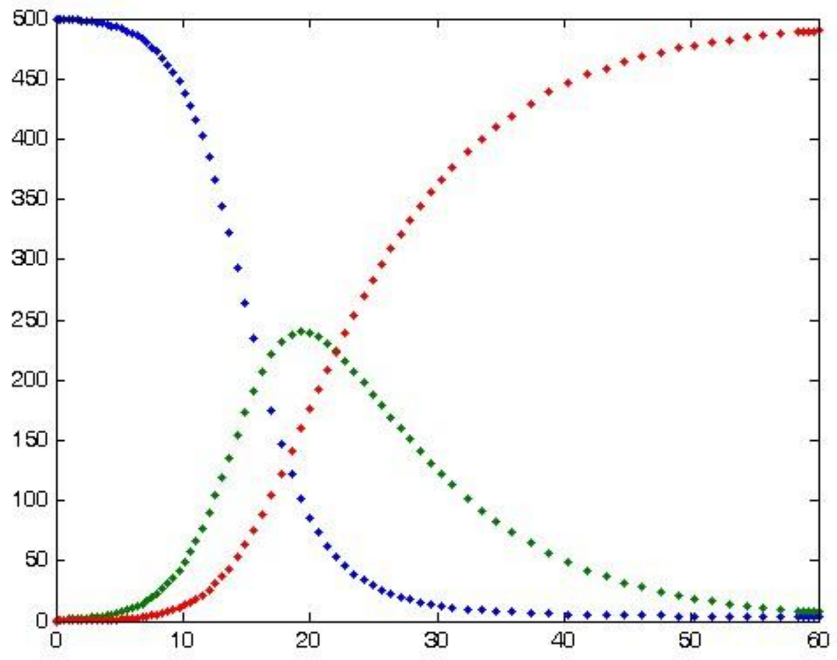


Figure 1Theoretical SIR model result, blue for S, green for I, and red for R

2.2 Discrete SIR Model Algorithm

The SIR model could be solved mathematically under some constraints. However, we need to simulate it on hardware to achieve a more realistic process. Thus, we need to figure out a hardware compatible algorithm for the SIR model.

Instead of treating the model as a continuous one, we decided to make it discrete which is how things work in hardware. That means, there is a minimum time step in the whole system. For every time step, there is a small possibility that an infected individual infects a susceptible one that is connected to it, or that the individual recovers. If this time step becomesinfinitely small, as the infect/recover probability approaches zero, the model becomes continuous.

In fact, if the time step is small enough, which means the infect/recover probability of each time step is small enough, we can take this discrete model as a continuous one with small error. If the error is acceptable (the probability of two individuals changing states in one time step is small), we can treat the model as a valid model.

2.3 Random Network Generation

To simulate the model, we need to build a network first. The network implies the relations between individuals. Two individuals can either be only connected or unconnected which means they can infect each other or not. The network is a bidirectional graph because if an individual can infect another, then it can be infected by it.

In this project, we use MATLAB to generate the random network. First the parameters; individual number and average connections per individual, are set at simulation start. Next we generate connections based on the parameters. We randomly choose two different individuals. If they are already connected, this pair of individuals is skipped. Otherwise we build a connection between them. In this way, we could generate a network with a totalconnection number of individual number multiplying average connections per individual. Then we generate a graph based on the connections we build and save it in the matrix.

2.4 Software Simulation

To test the algorithm, we simulate it on MATLAB before we actually implement it on the FPGA. After we generate the network, we randomly select several individuals (the number is set as parameters by users) to be infected initially. Then we start the simulation. We have two arrays to store the states of the individual; the current states and the next states. For each individual, we check whether it is currently infected. If it is, we search for every individual that is connected to it, generate a random number from 0 to 1, and compare the number to the probability. If the number is within the probability, we set the next state of the individual as infected. Also, if the individual is currently infected, we generate another random number to see whether it would recover next cycle. After going through all the individuals in the network, the current state of the individuals is updated as next state. This process will repeat until there are no infected individuals and this ends the simulation.

2.5 Hardware Implementation

2.5.1 SIR Cell

In the hardware implementation, each individual is abstracted as a block which contains a state machine representing the state of the individual. The state machine has two bits, one represents whether it is infectious, another represents whether is susceptible. There is an input signal for the cell that implies whether there is a source that would infect it next cycle. If there is, and the cell is susceptible, the infectious bit would set to one on the next cycle. Also the susceptible would clear. There’s another input signal as the recover signal, which is similar as the infectious signal, triggering the cell from infected to recover.

2.5.2 Random Number Generator

The core part of this design is the random number generator (RNG) which determines every random process. The performance of the RNG is the key to get a valid result. In our design, we choose a 63-bit linear feedback shift register as the RNG. The RNG is based on *“A special-purpose processor for the Monte Carlo simulation of ising spin systems”*by A. Hoogland, J. Spaa, B. Selman and A. Compagner but modified to use 63 bit shift register. The RNG generates a 64-bit random number every 4 cycles.

2.5.3 Block for Each Individual

For each individual, a hardware block is designed as follows

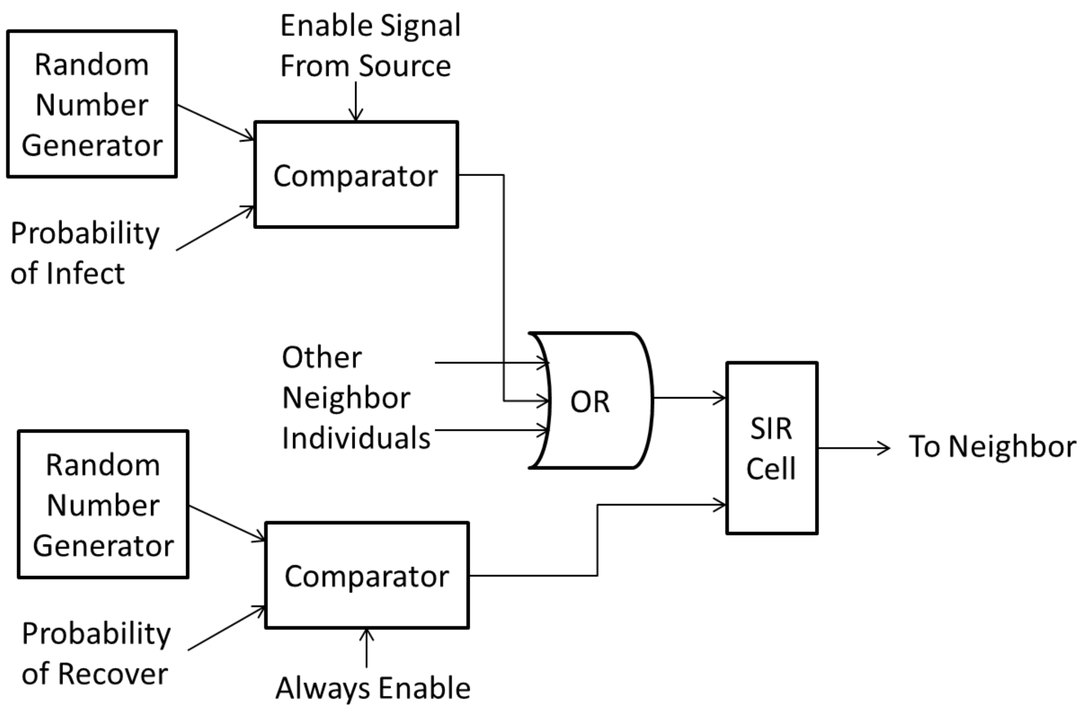


Figure 2 Schematic of Hardware for Each Individual

For each direction of every connection, there is a RNG rolling the dice. The output of the RNG and a fixed value which is the probability value set in MATLAB are going through a comparator. The comparator's enable signal is controlled by the source individual. If the source individual is infected and the RNG generates a number that is within the probability, the comparator will output a positive signal. Since each individual could be infected by any individual that is connected to it, all the infectious signals are OR together to generate the infectious input of SIR cell. There is a similar structure for recovery. The slightly difference is that the comparator is always enabled because only when the SIR cell is in the infected state, will it read the recover signal.

By duplicating this structure, we can generate a SIR model network. When some of the SIR cells are set to infected, the simulation will start.

2.5.4 Adder Tree

To observe the result of the simulation, we need to sum up the number of infected individuals every several time step. Since a ripple adder is to slow for this application especially when the individual numbers are high, we implemented an adder tree to sum the data. In addition, we pipelined the adder tree to achieve a smaller cycle time.

2.5.5 MATLAB to Verilog

Since we want to build different networks and assign different seeds for the RNG, we have to make the design more flexible. In this project, we used a MATLAB program to generate Verilog with the file writing function in MATLAB. There are five parameters of the model that we can change for a different simulation: probability of infection, probability of recovery, total individuals, average connections for each individual, and number of initially infected individuals.

The MATLAB program will generate a network based on the given parameters, then set the initial condition for the network. With the network established, the MATLAB program will write a Verilog file for the hardware implementation. The Verilog code will have hardware connections that describe the network generated in MATLAB. The seeds for random number generators are also randomly generated by MATLAB.

MATLAB also builds the adder tree based on the actual number of individual in the simulation.

By using MATLAB to generate the Verilog code, the program is much more flexible and easy to modify. Anytime we need a new set of parameters or want a new network or random seed, we only need to change MATLAB parameters and generate a new code. This simplifies the often difficult task of modifying Verilog for each simulation change.

2.6 Output Method

To observe the result of the simulation, we need to sum up the infected individuals every several time steps. The interval between additions can be changed by the switches on the FPGA board.To display the results, we used a VGA screen. During the simulation, the sums are saved in SRAM. After saving 640 results (the width of the screen), the simulation stops. The curve of the results will be displayed on VGA screen only the simulation completes. The VGA controller is from Jordan Crittenden.

2.7 Timing Issue

There are several timing issues that we need to consider for the design. First, a universal clock is required for the FPGA to work. The FPGA provides a 50MHz internal clock which is a good baseline frequency for testing. However, there are parts in our design that require a slower clock. The VGA control module needs a 25MHz clock, which is provided by an Altera Phase Lock Loop design that is included in the Altera IP library. The adder tree and data storage do not requirehigh speed clocks, but prescalar to trigger the register is not a good design because that may cause the latency and edge conflict. To avoid these issues, we implemented a counter inside the adder and storage module. By counting a number of values before executing a data add and store, we are triggering the execution on the real clock edge, but also get the slower frequency.If future experiments require higher clock speed, we need to use a PLL to increase the frequency for our main clock.

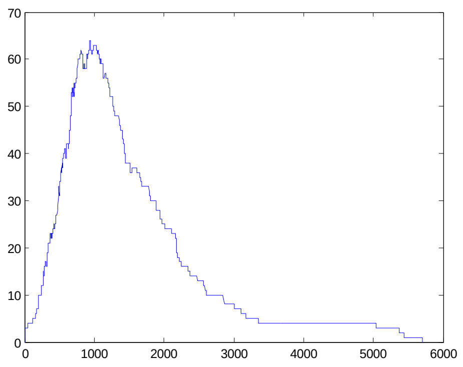
3 Tests and Results

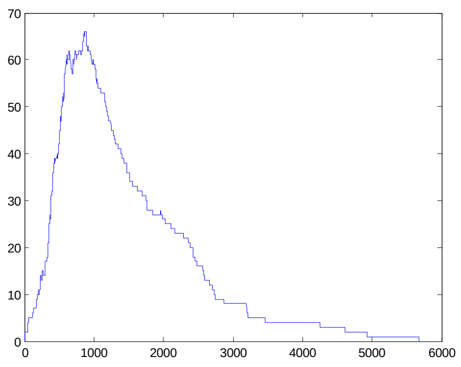
3.1 Control on Board

We control basic simulation function with user accessible devices on the FPGA board. There are two buttons to restart the simulation: one is to restart the simulation with the initial seeds fixed by MATLAB, the other is to restart the simulation with another set of seeds, selected at random. To fit the output graph to the screen, we need to modify the time scale determined by the SRAM add and store function. By setting switches, we can control the counter frequency for the SRAM, thus controlling storage speed for the results.

3.2 Simulation Results

Before actually implementing the algorithm on hardware, we tried our algorithm on a MATLAB simulation. As we discussed above, we can set five parameters to explore the SIR model. As an initial experiment, we set the P_Infecting = P_Recovering = 0.001 with a network of 100 people and 10 average connections with two individuals initially infected. We simulate using these settings, four times. The following figures show the relation between number of infected individuals to simulation cycles.





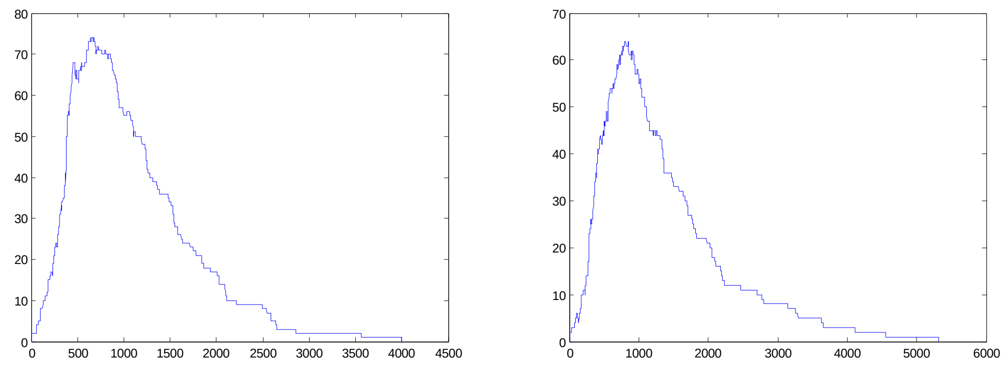


Figure 3 number of infected individuals vs cycles, P=0.001, 100 individuals with 10 average connections

As we can see, the output curve looks similar: a sharp rising and a relatively low falling, which matches the theoretical result for the SIR model we discussedin section 2.1. However, the curve is not smooth because the individual number is relatively low. If we increase the individual number to 1000, similar results are shown in figure 4:

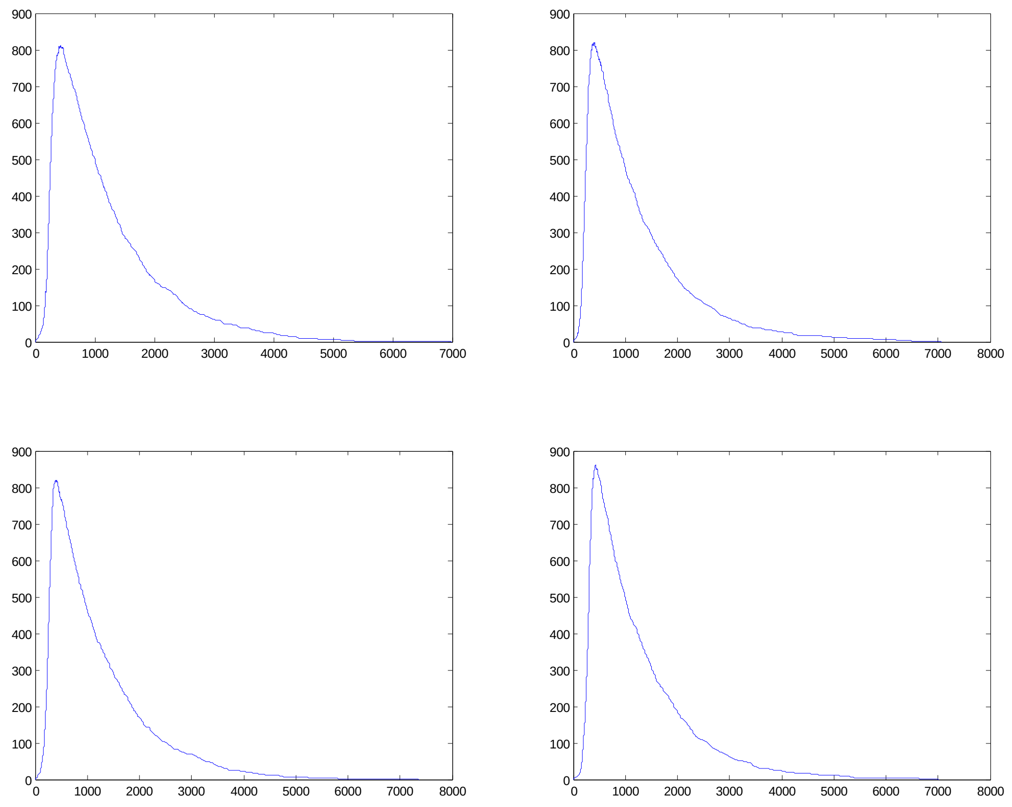


Figure 4 number of infected individual vs cycles, P=0.001, 1000 individuals with 10 average connections

This result looks smoother than the result with 100 individuals while it also illustrates characteristic features of an SIR model where the number of infected individuals rises quickly and falls relatively slowly.

If we double the probability of infection and recovery to 0.002, which could be approximately considered as double the time scale, we should expect the similar curve with half cycle number. The actual simulation is as follows:

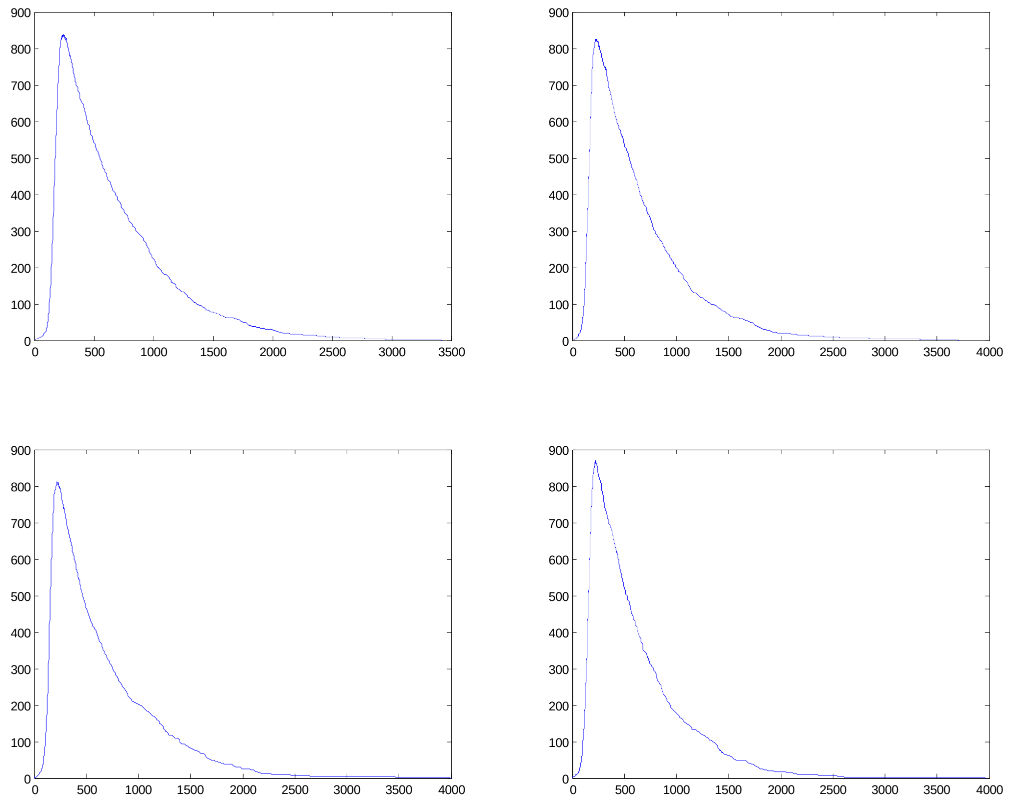


Figure 5 number of infected individuals vs cycles, P=0.002, 1000 individuals with 10 average connections

Basically we can say the time axis decreases by half which is our expectation.

In the SIR model, if the recoveryprobability is much more than the infection probability, it is possible that the disease would not spread. To test this feature, we set the recoveryprobability five times larger than the infectionprobability, with a network of 100 individuals and 10 average connections.

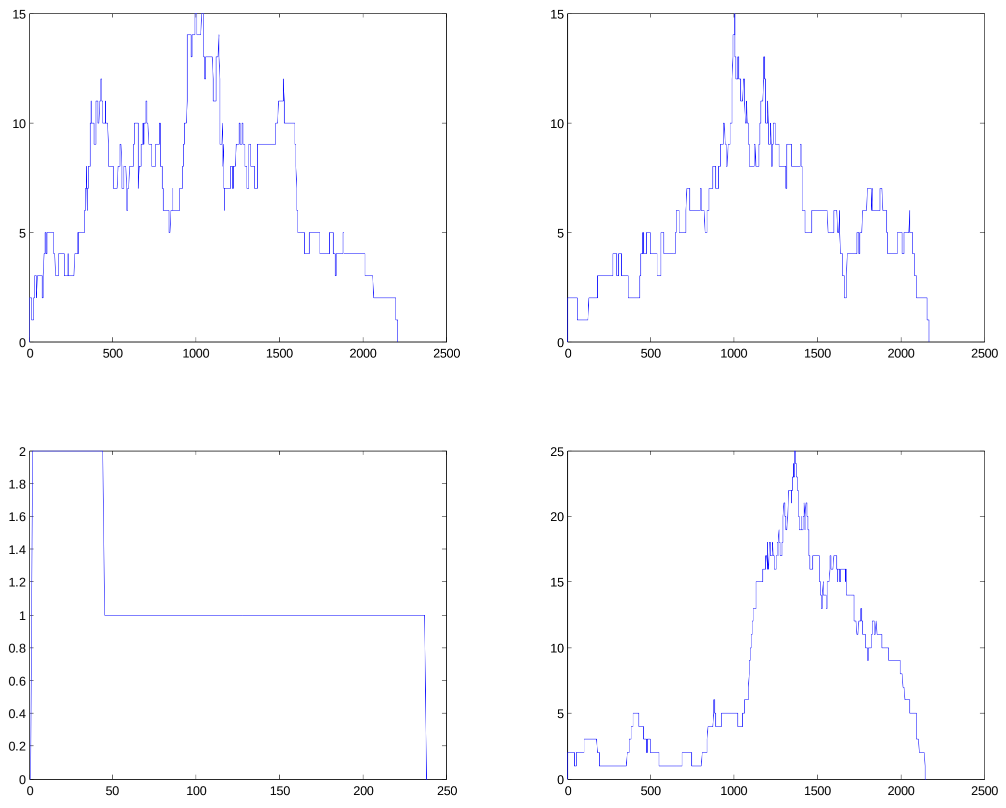


Figure 6 number of infected individual vs cycles, P_Recovery = 0.005, P_Infection=0.001, 100 individuals with 10 average connections

The third result shows that the disease has not spread at all.

The MATLAB simulation results match the theoretical result well, which suggests that the discrete algorithm is valid when the time step is small enough. By verifying our initial ideas with MATLAB simulations, we next implementedthe algorithm on the FPGA.

3.3 FPGA Simulation

With the same parameters for the network, the MATLAB simulation and the FPGA simulation should yield similar results, especially when the number ofindividuals in the simulation is relatively large. We began by implementing the same parameters on MATLAB and FPGA (P_Infecting = P_Recovering = 0.001, Individual Number = 100, Average Connection = 10, Initial Infected Individuals = 2), and the results are as follows:

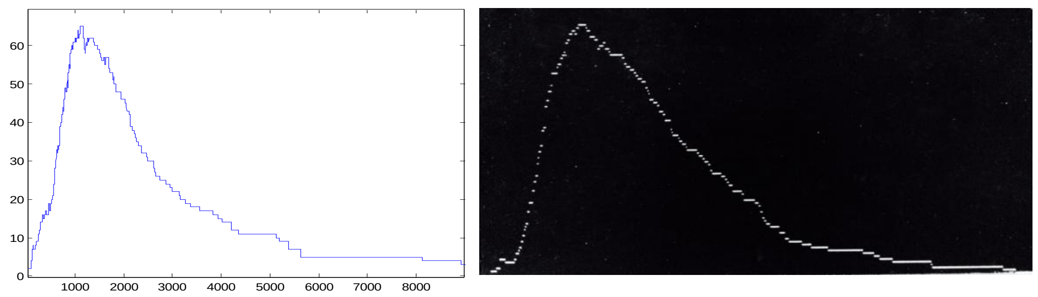


Figure 7 Comparison between MATLAB simulation result (left, and FPGA simulation result (right), P=0.001, 100 individuals with 10 average connections

As we can see from the figures, the results from MATLAB simulation and FPGA are similar with the same graphical shapes, with a steep initial rising followed by a slow decay in the number of infected individuals. However, similar results do not validate the FPGA version completely.

3.4 VERIFICATION

To verify the simulation, we need three more steps. First, with the same RNG, FPGA and MATLAB should yield exactly same results.This comparison would validate the Verilog code generation technique in MATLAB. Second, testing of the hardware RNG is required, especially compared to the MATLAB rand() function which is adequate for the stochastic simulation. Finally if we repeat the simulation on FPGA with the same seeds for all RNGs, the result should not change at all, proving the stability of the hardware simulation.

We implemented a 64-bit XOR shift hardware RNG on MATLAB, so we can use the exact same RNG in MATLAB and the FPGA, which establishes the foundation for the first step of verification. To run the same simulation on MATLAB and FPGA, we saved the entire network and the initial seeds for every RNG for hardware implementation and we use the same conditions for software simulation. In this way, every step should be exactly the same so the results should be identical.

However, it is difficult to tell difference between two similar graphs. A better way is to reduce the total individual number, making the result special, unique and easy to identify.

We used 10 individuals for verification, and the MATLAB/FPGA simulation yielded the identical results as follows:

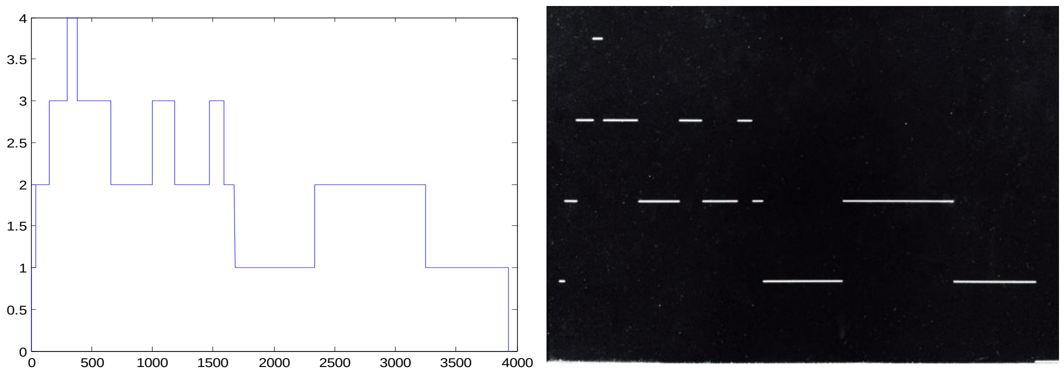


Figure 8 number of infected individualsvs cycles on MATLAB simulation (left) and FPGA simulation (right) with exactly same network and random number generator with same seeds

This test shows that the hardware simulation runs exactly as we expected so the implementation and the sampling is correct.

Random number generator is the most vital part of a stochastic simulation. It determines the quality of the simulation. An XOR linear feedback shift register is a common solution for RNG in hardware because it is relatively inexpensive (in terms of FPGA resources) and fast. In this case, we need to prove that it is good enough for the simulation.

We used two methods to test the RNG: A chi-square test and Aserial-correlation test. Since we used rand() function in MATLAB forgenerating the seeds for hardware RNG and software simulation, we basically assume that the rand() function is good enough for the whole simulation. Therefore, we just need to compare the hardware

RNG and the rand() function. Once the RNG is as good as the rand() function, it is good enough for hardware simulation.

For chi-square test, we repeated the test 50 times which generated 5000 random numbers. If the confidence is more than 90%, our test passed. Less than 90% would indicate a failed test. If in one test, one of the RNG shows a better result, we add a good to it.

For s-c test, we used lag=1 and lag=5, and the confidence is also 90%.

The results are as follows:

	Chi-Square			**S-C lag = 1**			**S-C lag = 5**		
	pass	fail	good	pass	fail	good	pass	fail	good
RNG	46	4	30	46	4	27	49	1	27
Rand()	44	6	20	41	9	25	45	5	23

Table 1 verification test comparison for hard random number generator and rand() function in MATLAB

From the chart, we can observe that hardware RNG shows an even better performance than rand() function in MATLAB. Consequently, the RNG is sufficient for our simulation.

The last thing we need to check is the stability. Since the results should remain the same if the seeds for RNGs are the same, we canrepeat the simulation and check for stability. In this case, we sampled the number of infected individual in the first 2000 steps as a flag value and observe this value between simulations. The results showed that the flag values stayed the same for every simulation, which proved the stability of the simulation.

4 Evaluation

Speed and area are the two most significant factors we should pay attention to in this project. We transferred the algorithm from software to hardware to make it faster, so we firstly analyze the speed difference between the MATLAB program and FPGA simulation.

4.1 Speed

In the FPGA design, the speed of execution is determined by the total cycle number and clock frequency. The cycle number varies from network to network and the clock frequency is limited by the hardware time constraint. If the cycle number is fixed and the clock frequency is set, the execution time can be calculated as

$$T_{execution} = \frac{Cycle\ \ Number}{Frequency} \times 4$$

The factor 4 is required because we generate one random number in four cycles.This implies that we can update the entire network every four cycles.

First we need to analyze the cycle number of the execution. As expected, cycle number is affected by connection number, individual number and the probability settings. Since we've proved that the MATLAB simulation yields the same result as the FPGA simulation, we can run our test in MATLAB, which makes it easier to obtain result data.

For the comparison, we can obtain accurate execution time in the MATLAB simulation. For the same network, we obtain the execution time in MATLAB and calculate the time it would take in the FPGA based on cycle number. This would give us a comparison between software and hardware.

Admittedly, MATLAB is relatively slow software for testing algorithms, but we can use MATLAB as a rough guide for comparison of software and hardware implementations.

4.1.1 Time vsProbability

Firstly, we will observe the impact of Probability. We fixed the network parameter at500 individuals, 20 average connections per individual and 2 initial infected individuals. Next we sweep the Probability for both infected and recovery from 0.00025 to 0.04. For every Probability we repeat the simulation with the same parameters 20 times and get the average execution time and cycle number. In the figure, X axis is probability.

P of I and R	*0.00025*	*0.0005*	*0.00075*	*0.001*	*0.002*	*0.003*	*0.004*
MATLAB Exec T(ms)	37198	15467	10530	7800	3845	2753	1958
Cycle Number	30550	13796	9829	7282	3656	2581	1786

Table 2 MATLAB Execution time and number of total cycles with different probability

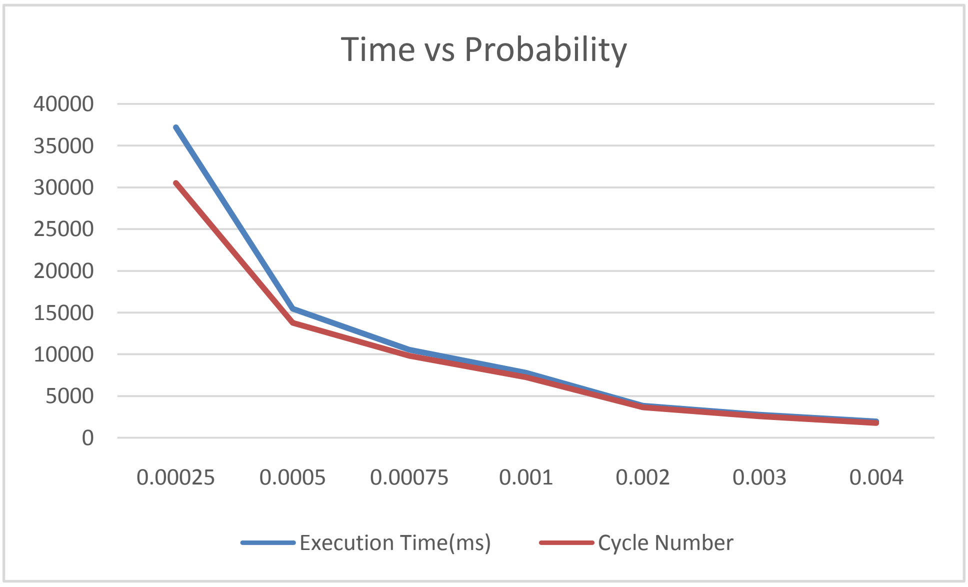


Figure 9 MATLAB Execution time and number of total cycles with different probability

As we can see in the chart, the execution time in MATLAB has a similar tendency with cycle number, which is in proportional with execution time in hardware.

Actually, if we divide two rows from the table and normalize that, we can get the figure below where Y axis is normalized time ratio between software and hardware.

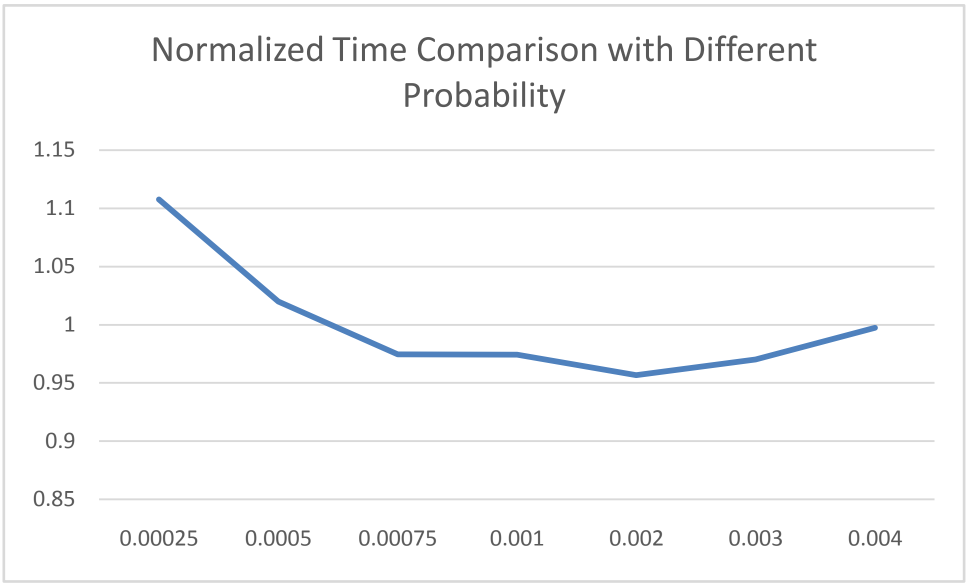


Figure 10normalized time comparison between software and hardware with different probability

We can see that the fluctuation is really small, basically within 10%. This result implies that the probabilitydoes not have a huge impact on the difference between software and hardware. This is not difficult to understand because the change of Probability only increases the cycle number. For software simulation, the time for each cycle approximately stays the same. In other words, the absolute value of Probability won't affect the difference between software simulation and hardware simulation.

4.1.2 Time vs Individual Number

Next we can explore the impact of individuals. We fix the Probability of infection and recovery at 0.0005. Also we set the average connection number at 15. Then we sweep the individual number from 100 to 1000. In fact, for the large scale simulation, it is intuitive that the average connection number for each individual is fixed because, as in real life, one person closely related to a certain number of people, independent of population size. In the figure, X axis is Individual number.

	100	*200*	*300*	*400*	*500*	*750*	*1000*
MATLAB Exec T(ms)	4157	6419	9142	11879	14024	21887	29898
Cycle Number	12243	12521	13527	14028	13789	15743	16717

Table 3 MATLAB Execution time and number of total cycles with different individuals

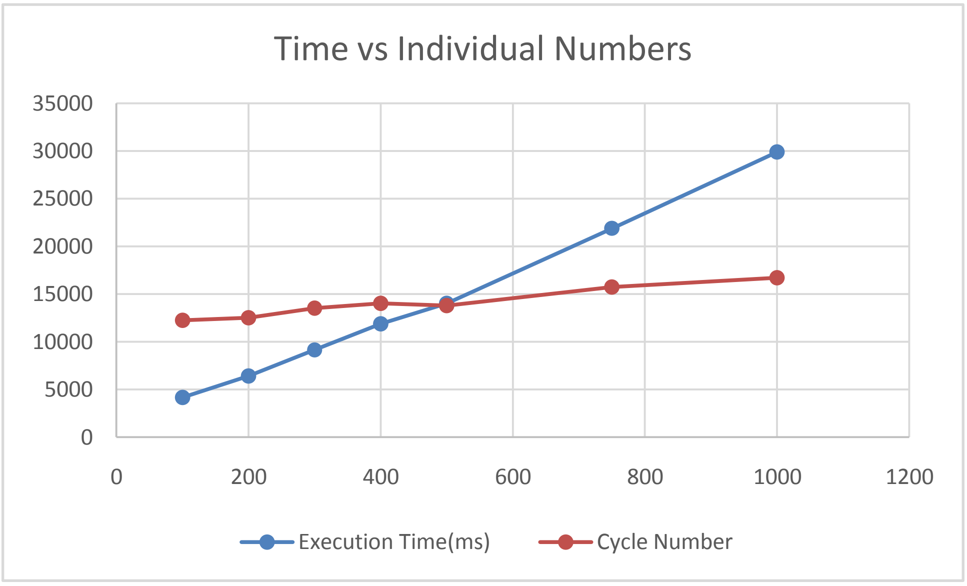


Figure 11 MATLAB Execution time and number of total cycles with different individuals

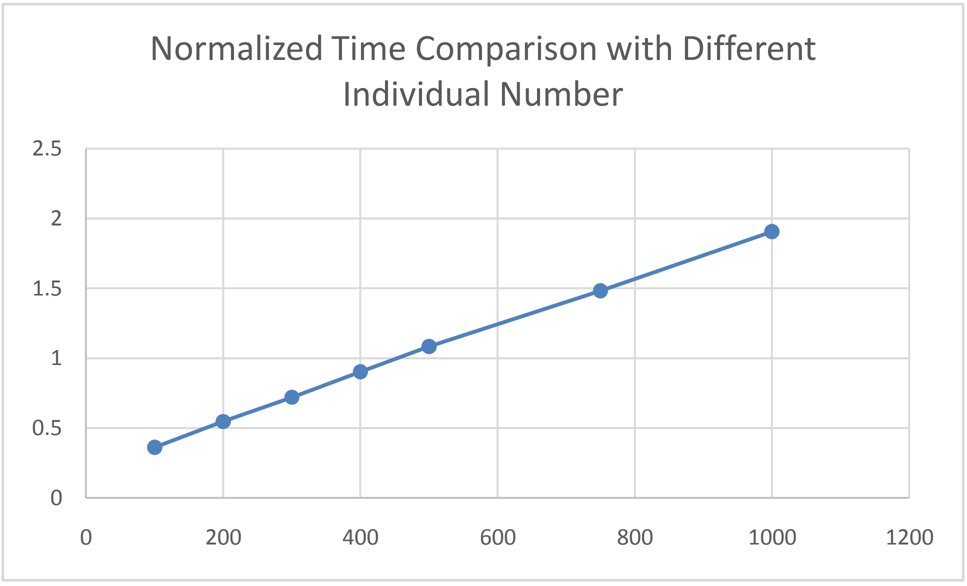


Figure 12 normalized time comparison between software and hardware with different individuals

As we can see in the chart, with individual numbers growing, the speed difference between software and hardware changes significantly. The software execution time increases linearly. However, cycle number is relatively stable. This means, while the scale of the network is increasing, the speed of hardware simulation basically stays the same, but the software simulation slows down. The ratio of the execution time between software and hardware increases linearly. This is because, in software, each cycle takes more time to execute, but in hardware, the computation is in parallel, limited only by the resources available in the FPGA.

4.1.3 Time vs Average Connection Number

Next, we could change the average connection number with the same Probability 0.0005 and the same total individual number 500. We will increase the average connection number per individual from 10 to 40. In the figure, X axis is the average connection number.

	10	*15*	*20*	*25*	*30*	*35*	*40*
MATLAB Exec T(ms)	14149	15265	16224	18244	18689	20179	20772
Cycle Number	15501	15057	14314	15374	14078	13565	12940

Table 4 MATLAB Execution time and number of total cycles with different average connections

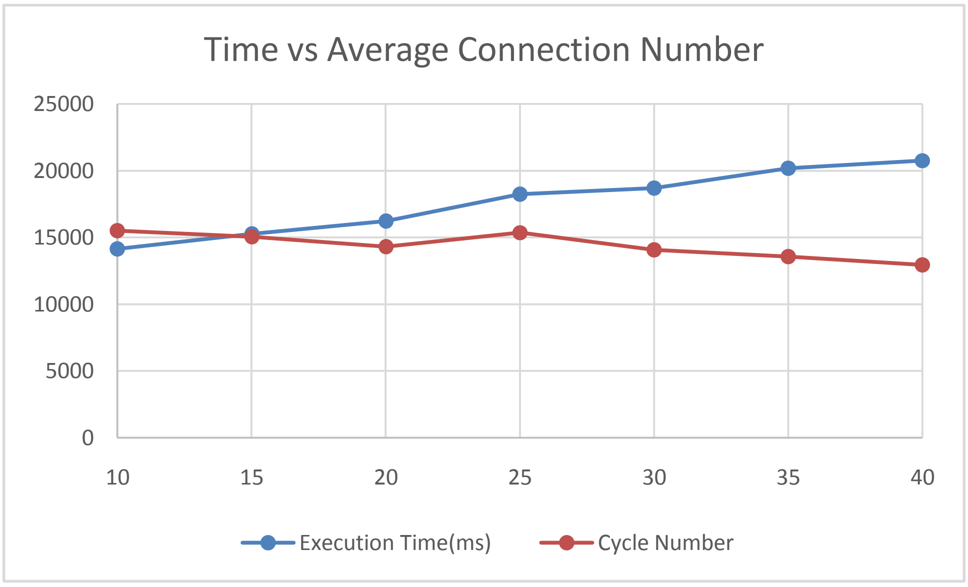


Figure 13 MATLAB Execution time and number of total cycles with different individuals

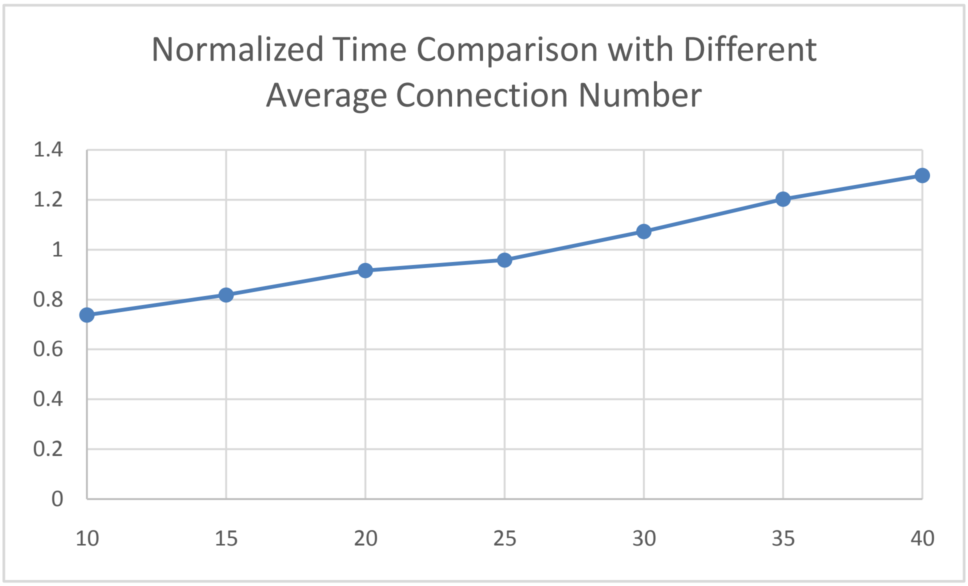


Figure 14 normalized time comparison between software and hardware with different average connections

With the average connection number increasing, the execution time of software increases. This is mainly due to the increasing time of each cycle. However, the cycle number decreases because with more connections, the disease spread faster and it takes less time to finish. The hardware simulation shows a better tendency in time aspect with increasing average connection number.

4.1.4 The Actual Acceleration

The discussion above is based on an ideal situation that we have a large enough FPGA which can hold as many individuals as we want. However, in this project, due to the limited resources of the FPGA the largest simulation is 140 individuals with 14 average connections per individual.

With these parameters and a probability of 0.0005, we run the simulation 20 times and convert the cycle time to actual execution time with a 100MHz clock, which is the best we can achieve on DE2i-150 without any timing issues, we can get the comparison graph as follows

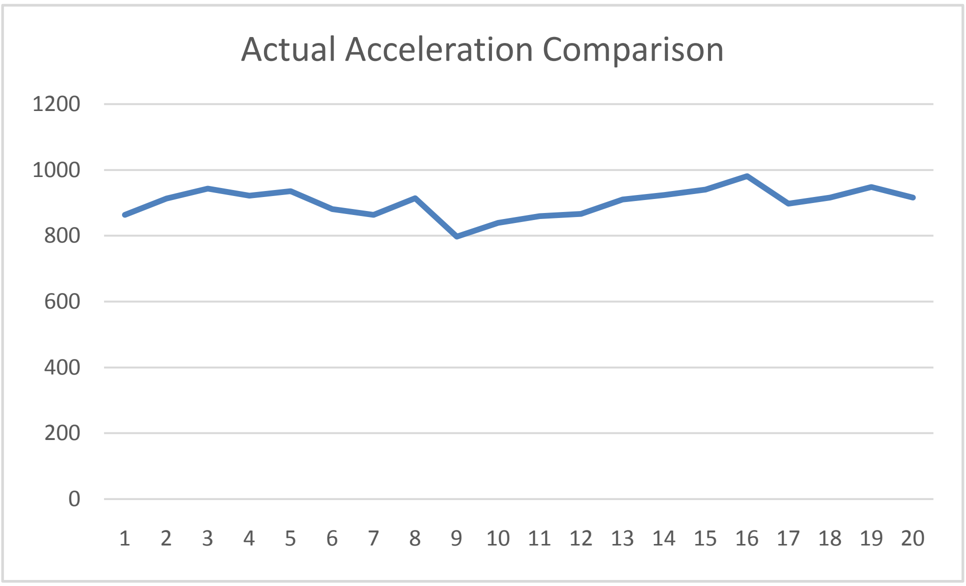


Figure 15 speed comparison between FPGA simulation and MATLAB simulation

As we can see in the figure, the acceleration is basically between 800 and 1000 times. This is a good result for hardware implementation. As we have discussed above, with the scale of the network increasing, we can definitely get a better acceleration with a parallel computing method.

4.2 AREA

Basically, the FPGA converts the complexity from time to area. Area is an important issue we need to analyze because it limits the scale of our simulation. In our design, the most costly part is the random number generator. So the area of the whole design is mainly determined by the total RNG number, which is the total connection number plus individual number. Each individual also has a small state machine and some basic logic for the state machine, requiring the use of some logic elements. Finally, there are some basic parts to complete the design, such as PLL, VGA controller, and adder. These will require a fixed number of logic elements.

4.2.1 Different Individual Number with Same Connections

First we want to observe the impact of individual numbers, so we tested some parameters with identical connection numbers but different individual numbers. We compiled four times with the same parameters using different network and take the average number. For area, we use the logic elements requiredwithin FPGA.

Individual#/Average Con#	*12/10*	*15/8*	*20/6*	*24/5*	*30/4*
Network 1	8005	8779	8954	9734	10605
Network 2	8212	8851	9372	9668	10249
Network 3	8072	8641	9233	9670	10673
Network 4	8142	8990	9580	9597	10112
Average Area	8108	8815	9285	9667	10410

Table 5 number of logic elements required in FPGA for different individuals but same total connections

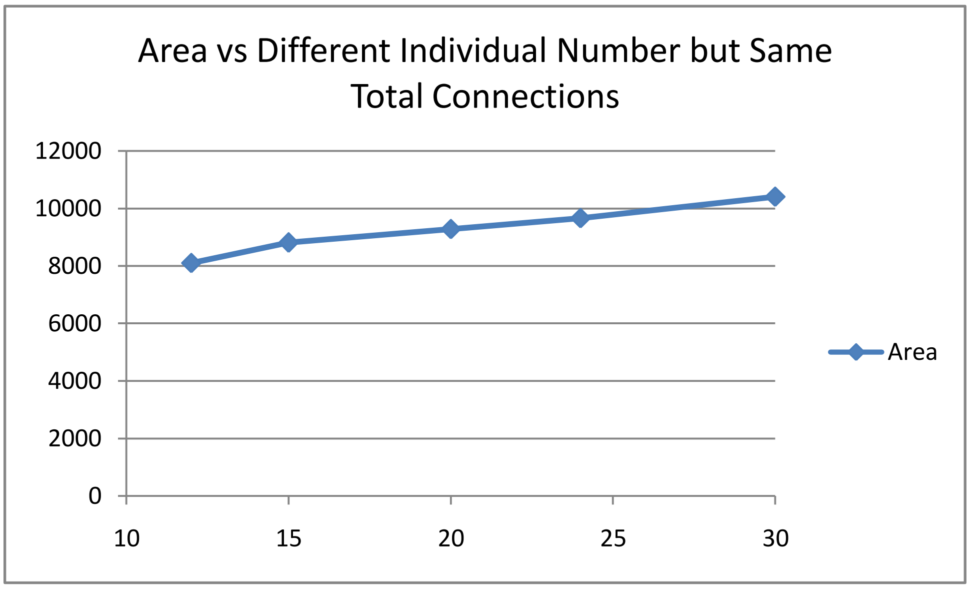


Figure 16 number of logic elements required in FPGA for different individuals but same total connections

From the figure we can see that the area is approximately linear to the individual number with the same total connection number which is our expectation. We can also roughly calculate that each individual would take about 100 logic elements.

4.2.2 Different Connection Number with Same Individual Number

Next, we fixed the individual number to 20, and sweep the average connection number from 4 to 10.

Individual#/Average Con #	*20/4*	*20/6*	*20/8*	*20/10*
Network 1	6872	9163	11591	14363
Network 2	6592	9579	11800	14297
Network 3	6943	9370	12078	14295
Network 4	6730	9509	11591	14088
Average Area	6784	9405	11765	14261

Table 6 number of logic elements required by FPGA for the same individuals but different average connections

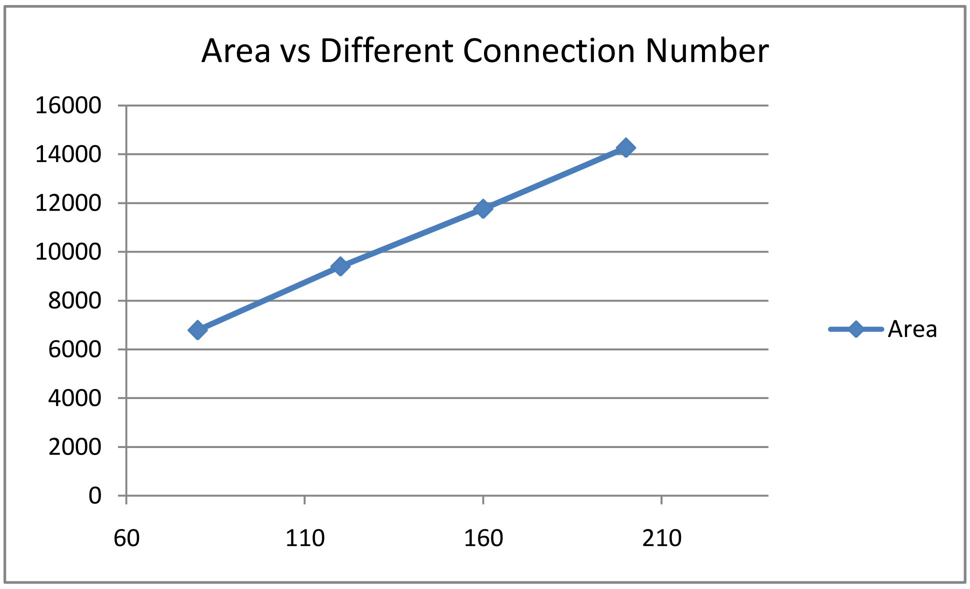


Figure 17 number of logic elements required by FPGA for the same individuals but different average connections

From the figure we can see, the area is in linear with the total connection number using identical individual numbers. We can roughly calculate that each connection takes 60-65 logic elements.

4.2.3 The Final Result in FPGA

In our final design, we used DE2i-150 which has about 150k logic elements. Our maximum implementation is 140 individuals with 14 average connections per individual. In our data above, it should take about 140*14*62.5 + 140*100 = 136k elements. The actual number of logic elements required is 146k. The error is mainly due to the rough estimation for an individual/connection, but also probably because of the relatively large network. A large network may cause a deep adder tree which would lead to a larger use of logic elements.

4.3 SUMMARY

In summary, probability would not greatly impact execution time differences between software and hardware a lot, so probability changes have little impact on acceleration. As individual number and average connection number grow, the better hardware simulation performs, relative to software execution time. Based on this, we can confidently anticipate that with larger scale of network, the hardware simulation would have an even fasterspeed compared to software simulation.

The downside of hardware simulation is the area. The number of logic elements basically grows linearly with total connections. So with the scale of the network increasing, we requirea larger FPGA, or a series of FPGAs. In DE2i-150, we can simulate 140 individuals with 14 average connections per individual.

5 CONCLUSION

In this project, we designed a FPGA based device to simulate SIR model, which is about 1000 times faster than it is in MATLAB. The Verilog code is automatically generated in MATLAB and the parameters are set in MATLAB as well. The use of Verilog code generation provides excellent design flexibility. A well designed hardware random number generator is introduced to ensure the simulation is valid. The hardware simulation is verified for solving SIR model compared to theoretical method. For future development, it is predicted that the FPGA model would give an even better performance when the model scales to even larger devices or a network of FPGA devices.

6 ACKNOWLEDGEMENT

I would like to thank Professor Bruce Land who is the advisor for me MEng project. His advice and help truly makes the project easier for me to accomplish. He also provided many ideas to improve and validate the design. Without his help, the project would be painful and difficult, and I might not be able to finish it in time.

Also, I would like to thank Joseph Skovira and Professor Dave Schneider who help me a lot during the weekly meeting. They provided me information that I'm not familiar with so I can actually think in that region and try to study related knowledge.

At last, I really want to thank Cornell University for provided me such a good research environment that I can sit down and think. It's a great experience studying here and I sincerely the university continued success.

7 REFERENCE

[1] A. Hoogland, J. Spaa, B. Selman and A. Compagner, A special-purpose processor for the Monte Carlo simulation of ising spin systems, Journal of Computational Physics, Volume 51, Issue 2, August 1983, Pages 250-260

[2] Skyler Schneider, VGA control module, http://people.ece.cornell.edu/land/courses/ece5760/StudentWork/ss868/ss868_VGA_Controller.zip

[3] "Compartmental models in epidemiology", Wikipedia, http://en.wikipedia.org/wiki/Compartmental_models_in_epidemiology

[4] DE2i-150 specification on Terasic website, http://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English&CategoryNo=11&No=529&PartNo=1

[5] Altera PLL IP, http://www.altera.com/literature/ug/altera_pll.pdf

8 APPENDIX

8.1 MATLAB CODE

```
clc;clear;
%%%%%%%%%%%Parameters Are HERE!!!!!!!!!
P=0.001;  %Probability that an infected person get another
P_Recover=0.001;   %Probability of recovering
Avr_Con_Num=14;   %average of connections of one person
Infected_Num=2;   %the infected people number when start
People_Num=140; %Total number of people(sample)
%%%%%%%%%%%Parameters Done!!!!!!!!
Log_People_Num = ceil(log2(People_Num));
char16='0123456789ABCDEF';
Ptemp=P;
P_Recovertemp=P_Recover;
charP=[];
charP_Recover=[];
InitInfected=zeros(People_Num,1);
Individual=zeros(People_Num,2); %graph, including Num of people and
their connections
Col_Num_Max=2;
fori=1:1:People_Num           %build graph(connections)
Individual(i,1)=i;
end
for Relation=1:1:People_Num*Avr_Con_Num/2 %build relation
flag=0;
while (flag==0)
       temp1=ceil(rand()*People_Num);
       temp2=ceil(rand()*People_Num);
if (temp1~=temp2)
fori=2:1:Col_Num_Max
if Individual(temp1,i)==temp2
break; %if reduplicate, break for
elseif Individual(temp1,i)==0 %valid relation
Individual(temp1,i)=temp2;
for j=2:1:Col_Num_Max %non-direction link
if Individual(temp2,j)==0
Individual(temp2,j)=temp1;
flag=1;
break;
```

```
elseif j==Col_Num_Max
                        Individual=[Individual,zeros(People_Num,1)];
Col_Num_Max=Col_Num_Max+1;
Individual(temp2,j+1)=temp1;
flag=1;
break;
end
end
break;
elseifi==Col_Num_Max
                 Individual=[Individual,zeros(People_Num,1)];
Col_Num_Max=Col_Num_Max+1;
end
end
end
end
end

Individual = [Individual,zeros(People_Num,1)];  % add a colomn of zeroat
the end

fori=1:1:4      %create 16_bit char for Probability
Ptemp=Ptemp*16;
charP=[charP,char16(floor(Ptemp)+1)];
Ptemp=Ptemp-floor(Ptemp);
P_Recovertemp=P_Recovertemp*16;
charP_Recover=[charP_Recover,char16(floor(P_Recovertemp)+1)];
P_Recovertemp=P_Recovertemp-floor(P_Recovertemp);
end
i=1;
whilei<=Infected_Num
getInf=ceil(People_Num*rand());
ifInitInfected(getInf,1)==0
InitInfected(getInf,1)=1;
i=i+1;
end
end
fid=fopen('SIR.v','wt');
fid_simu = fopen('simu.txt','wt');
%-----------------------------------------------------
% print relation in the txt file
%-----------------------------------------------------
fori=1:1:People_Num
```

```
    j=1;
while (Individual(i,j)~=0)
fprintf(fid_simu,'%i ',Individual(i,j));
       j=j+1;
end
if (InitInfected(i,1) == 0)
fprintf(fid_simu,'0\n');
else
fprintf(fid_simu,'-1\n');
end
end
%-----------------------------------------------------
%module SIR(clk,Reset_All,Reset_I,S,I);
%-----------------------------------------------------
fprintf(fid,'module SIR(clk,Reset_All,Reset_I,S,I);\n');
fprintf(fid,'\tinput wire clk,Reset_All,Reset_I;\n');
fprintf(fid,'\toutput wire [%i:1]S;\n',People_Num);
fprintf(fid,'\toutput wire [%i:1]I;\n',People_Num);
fprintf(fid,'\twire Reset = Reset_All | Reset_I;\n');
fprintf(fid,'\twirestate_in;\n');
fprintf(fid,'\tassignstate_in=0;\n');
fprintf(fid,'\twire [15:0]Constin_en;\n');
fprintf(fid,'\twire [15:0]Constin_dis;\n');
fprintf(fid,'\tassignConstin_en=15''h%s;\n',charP);
fprintf(fid,'\tassignConstin_dis=15''h%s;\n',charP_Recover);

fori=1:1:People_Num
fprintf(fid,'\twireen%i;\n',i);
fprintf(fid,'\twireresult_dis%i;\n',i);
fprintf(fid,'\twire [63:1]seed_dis%i;\n',i);
fprintf(fid,'\twire [15:0]rand_dis%i;\n',i);
ifInitInfected(i,1)==0
fprintf(fid,'\tSIR_cellSIR%i (clk, en%i, result_dis%i, Reset, S [%i],
I[%i]);\n',i,i,i,i,i);
else
fprintf(fid,'\tSIR_cellSIR%i (clk, 1, result_dis%i, Reset, S [%i],
I[%i]);\n',i,i,i,i);
end
fprintf(fid,'\trand63
rand63_dis%i(.rand_out(rand_dis%i), .seed_in(seed_dis%i), .state_in(s
tate_in), .clock_in(clk), .reset_in(Reset_All));\n',i,i,i);
```

```
fprintf(fid,'\tComparator
compdis%i(.clk(clk),.rst(Reset),.Constin(Constin_dis),.Variablein(ran
d_dis%i),.en(I[%i]),.Result(result_dis%i));\n',i,i,i,i);
fprintf(fid,'\tassignseed_dis%i=63''h',i);
temp_char = char16(ceil(rand()*8));
fprintf(fid,'%c',temp_char);
fprintf(fid_simu,'%c',temp_char);
for j=1:1:15    %assign seeds
temp_char = char16(ceil(rand()*16));
fprintf(fid,'%c',temp_char);
fprintf(fid_simu,'%c',temp_char);
end
fprintf(fid_simu,'\n');
fprintf(fid,';\n');
for j=2:1:Col_Num_Max
target = Individual(i,j);
if target == 0
break;
end
fprintf(fid,'\twire result_en%i_to_%i;\n',i,target);
fprintf(fid,'\twire [63:1]seed_en%i_to_%i;\n',i,target);
fprintf(fid,'\twire [15:0]rand_en%i_to_%i;\n',i,target);
fprintf(fid,'\tassign seed_en%i_to_%i=63''h',i,target);
temp_char = char16(ceil(rand()*8));
fprintf(fid,'%c',temp_char);
fprintf(fid_simu,'%c',temp_char);
for k=1:1:15    %assign seeds
temp_char = char16(ceil(rand()*16));
fprintf(fid,'%c',temp_char);
fprintf(fid_simu,'%c',temp_char);
end
fprintf(fid_simu,'\n');
fprintf(fid,';\n\trand63
rand63_en%i_to_%i(.rand_out(rand_en%i_to_%i), .seed_in(seed_en%i_to_%
i), .state_in(state_in), .clock_in(clk), .reset_in(Reset_All));\n',i,
target,i,target,i,target);
fprintf(fid,'\tComparator
compen%i_to_%i(.clk(clk),.rst(Reset),.Constin(Constin_en),.Variablein
(rand_en%i_to_%i),.en(I[%i]),.Result(result_en%i_to_%i));\n',i,target
,i,target,i,i,target);
end

end
```

```
%--------------------------------------------------------
% Relations
%--------------------------------------------------------
fori=1:1:People_Num    %relation
fprintf(fid,'\tassignen%i=1''b0',i);
    j=2;
while j<=Col_Num_Max&& Individual(i,j)~=0
fprintf(fid,' | result_en%i_to_%i',Individual(i,j),i);
        j=j+1;
end
fprintf(fid,';\n');
end
fprintf(fid,'endmodule\n');
%--------------------------------------------------------
% module SIR_cell(clk, StoIEn, ItoREn, Reset, S, I);
%--------------------------------------------------------
fprintf(fid,'moduleSIR_cell (clk, StoIEn, ItoREn, Reset, S, I);\n');
fprintf(fid,'\toutputreg S,I;\n');
fprintf(fid,'\tinput wire clk,StoIEn,ItoREn,Reset;\n');
fprintf(fid,'\talways @ (posedgeclk)\n');
fprintf(fid,'\tbegin\n');
fprintf(fid,'\t\tif (Reset)\n');
fprintf(fid,'\t\tbegin\n');
fprintf(fid,'\t\t\tS<=1''b1;\n');
fprintf(fid,'\t\t\tI<=1''b0;\n');
fprintf(fid,'\t\tend\n');
fprintf(fid,'\t\telse\n');
fprintf(fid,'\t\tbegin\n');
fprintf(fid,'\t\t\tif (StoIEn&& S)\n');
fprintf(fid,'\t\t\tbegin\n');
fprintf(fid,'\t\t\t\tS<=1''b0;\n');
fprintf(fid,'\t\t\t\tI<=1''b1;\n');
fprintf(fid,'\t\t\tend\n');
fprintf(fid,'\t\t\tif (ItoREn&& I)\n');
fprintf(fid,'\t\t\tbegin\n');
fprintf(fid,'\t\t\t\tI<=1''b0;\n');
fprintf(fid,'\t\t\tend\n');
fprintf(fid,'\t\tend\n');
fprintf(fid,'\tend\n');
fprintf(fid,'endmodule\n');
%--------------------------------------------------------
%module Comparator (clk,rst,Constin,Variablein,en,Result);
%--------------------------------------------------------
```

```
fprintf(fid,'module Comparator
(clk,rst,Constin,Variablein,en,Result);\n');
fprintf(fid,'\tinput wire [15:0] Variablein;\n');
fprintf(fid,'\tinput wire [15:0] Constin;\n');
fprintf(fid,'\tinput wire clk,rst,en;\n');
fprintf(fid,'\toutputreg Result;\n');
fprintf(fid,'\treg [1:0]state;\n');
fprintf(fid,'\tregtempResult;\n');
fprintf(fid,'\talways @ (posedgeclk)\n');
fprintf(fid,'\tbegin\n');
fprintf(fid,'\t\tif (rst)\n');
fprintf(fid,'\t\tbegin\n');
fprintf(fid,'\t\t\tstate<=2''b00;\n');
fprintf(fid,'\t\t\tResult<=1''b0;\n');
fprintf(fid,'\t\t\ttempResult<=1''b0;\n');
fprintf(fid,'\t\tend\n');
fprintf(fid,'\t\telse\n');
fprintf(fid,'\t\tbegin\n');
fprintf(fid,'\t\t\tcase (state)\n');
fprintf(fid,'\t\t\t2''b00:\n');
fprintf(fid,'\t\t\t\tbegin\n');
fprintf(fid,'\t\t\t\t\tResult<= (en &&Variablein<Constin);\n');
fprintf(fid,'\t\t\t\tend\n');
fprintf(fid,'\t\t\tendcase\n');
fprintf(fid,'\t\t\tstate<= state + 1''b1;\n');
fprintf(fid,'\t\tend\n');
fprintf(fid,'\tend\n');
fprintf(fid,'endmodule\n');
%---------------------------------------------------
%module rand63(rand_out, seed_in, state_in, clock_in, reset_in);
%---------------------------------------------------
fprintf(fid,'module rand63(rand_out, seed_in, state_in, clock_in,
reset_in);\n');
fprintf(fid,'\toutput wire [15:0] rand_out ;\n');
fprintf(fid,'\tinput wire state_in ;\n');
fprintf(fid,'\tinput wire clock_in, reset_in ;\n');
fprintf(fid,'\tinput wire [63:1] seed_in; \n');
fprintf(fid,'\treg [4:1] sr1, sr2, sr3, sr4, sr5, sr6, sr7, sr8, sr9, sr10,
sr11, sr12, sr13, sr14, sr15, sr16;\n');
fprintf(fid,'\tparameterreact_start=1''b0 ;\n');
fprintf(fid,'\tassignrand_out = {sr1[3], sr2[3], sr3[3], sr4[3],sr5[3],
sr6[3], sr7[3], sr8[3],sr9[3], sr10[3], sr11[3], sr12[3],sr13[3],
sr14[3], sr15[3], sr16[3]} ;\n');
```

```
fprintf(fid,'\talways @ (posedgeclock_in)\n');
fprintf(fid,'\tbegin\n');
fprintf(fid,'\t\tif (reset_in)\n');
fprintf(fid,'\t\tbegin\n');
fprintf(fid,'\t\t\tsr1 <= seed_in[4:1] ;\n');
fprintf(fid,'\t\t\tsr2 <= seed_in[8:5] ;\n');
fprintf(fid,'\t\t\tsr3 <= seed_in[12:9] ;\n');
fprintf(fid,'\t\t\tsr4 <= seed_in[16:13] ;\n');
fprintf(fid,'\t\t\tsr5 <= seed_in[20:17] ;\n');
fprintf(fid,'\t\t\tsr6 <= seed_in[24:21] ;\n');
fprintf(fid,'\t\t\tsr7 <= seed_in[28:25] ;\n');
fprintf(fid,'\t\t\tsr8 <= seed_in[32:29] ;\n');
fprintf(fid,'\t\t\tsr9 <= seed_in[36:33] ;\n');
fprintf(fid,'\t\t\tsr10 <= seed_in[40:37] ;\n');
fprintf(fid,'\t\t\tsr11 <= seed_in[44:41] ;\n');
fprintf(fid,'\t\t\tsr12 <= seed_in[48:45] ;\n');
fprintf(fid,'\t\t\tsr13 <= seed_in[52:49] ;\n');
fprintf(fid,'\t\t\tsr14 <= seed_in[56:53] ;\n');
fprintf(fid,'\t\t\tsr15 <= seed_in[60:57] ;\n');
fprintf(fid,'\t\t\tsr16 <= {1''b0,seed_in[63:61]} ;\n');
fprintf(fid,'\t\tend\n');
fprintf(fid,'\t\telse\n');
fprintf(fid,'\t\tbegin\n');
fprintf(fid,'\t\t\tsr1 <= {sr1[3:1], sr16[3]^sr15[3]} ;\n');
fprintf(fid,'\t\t\tsr2 <= {sr2[3:1], sr16[3]^sr1[4]}  ;\n');
fprintf(fid,'\t\t\tsr3 <= {sr3[3:1], sr1[4]^sr2[4]}  ;\n');
fprintf(fid,'\t\t\tsr4 <= {sr4[3:1], sr2[4]^sr3[4]}  ;\n');
fprintf(fid,'\t\t\tsr5 <= {sr5[3:1], sr3[4]^sr4[4]}  ;\n');
fprintf(fid,'\t\t\tsr6 <= {sr6[3:1], sr4[4]^sr5[4]}  ;\n');
fprintf(fid,'\t\t\tsr7 <= {sr7[3:1], sr5[4]^sr6[4]}  ;\n');
fprintf(fid,'\t\t\tsr8 <= {sr8[3:1], sr6[4]^sr7[4]}  ;\n');
fprintf(fid,'\t\t\tsr9 <= {sr9[3:1], sr7[4]^sr8[4]}  ;\n');
fprintf(fid,'\t\t\tsr10 <= {sr10[3:1], sr8[4]^sr9[4]}  ;\n');
fprintf(fid,'\t\t\tsr11 <= {sr11[3:1], sr9[4]^sr10[4]}  ;\n');
fprintf(fid,'\t\t\tsr12 <= {sr12[3:1], sr10[4]^sr11[4]}  ;\n');
fprintf(fid,'\t\t\tsr13 <= {sr13[3:1], sr11[4]^sr12[4]}  ;\n');
fprintf(fid,'\t\t\tsr14 <= {sr14[3:1], sr12[4]^sr13[4]}  ;\n');
fprintf(fid,'\t\t\tsr15 <= {sr15[3:1], sr13[4]^sr14[4]}  ;\n');
fprintf(fid,'\t\t\tsr16 <= {sr16[3:1], sr14[4]^sr15[4]}  ;\n');
fprintf(fid,'\t\tend\n');
fprintf(fid,'\tend\n');
fprintf(fid,'endmodule\n');
%----------------------------------------------------
```

```
%module Infected_Adder_Tree(clk,rst,I,sumI)
%----------------------------------------------------
Adder_Tree = [];
Adder_Num = ceil(People_Num/2);
Tree_Depth = 0;
while (Adder_Num>1)
Adder_Tree = [Adder_Tree,Adder_Num];
Adder_Num = ceil(Adder_Num/2);
Tree_Depth = Tree_Depth + 1;
end
fprintf(fid,'moduleInfected_Adder_Tree(clk,rst,I,sumI);\n');
fprintf(fid,'\tinputclk,rst;\n');
fprintf(fid,'\tinput [%i:1]I;\n',People_Num);
fprintf(fid,'\toutput wire [%i:0]sumI;\n',Log_People_Num);
fprintf(fid,'\twire [%i:0]sumI_temp;\n',Log_People_Num);
fprintf(fid,'\treg [%i:0]sumI_r;\n',Log_People_Num);
fprintf(fid,'\treg [%i:1]Itemp;\n',People_Num);
fprintf(fid,'\tassignsumI = sumI_r;\n');
fori = 1:1:Tree_Depth
for j = 1:1:Adder_Tree(i)
fprintf(fid,'\treg [%i:0]adder_%i_%i;\n',i,i,j);
end
end
if (Tree_Depth ~= 0)
fprintf(fid,'\tassignsumI_temp = adder_%i_1 +
adder_%i_2;\n',Tree_Depth,Tree_Depth);
else
fprintf(fid,'\tassignsumI_temp = Itemp;\n');
end
fprintf(fid,'\talways @ (posedgeclk)\n');
fprintf(fid,'\tbegin\n');
fprintf(fid,'\t\tItemp[%i:1]<=I[%i:1];\n',People_Num,People_Num);
fori = 1:1:Tree_Depth
for j = 1:1:Adder_Tree(i)
if (i==1)       %%First line
if (2*j<=People_Num)
fprintf(fid,'\t\tadder_%i_%i<=
Itemp[%i]+Itemp[%i];\n',i,j,(2*j-1),2*j);
else
fprintf(fid,'\t\tadder_%i_%i<= Itemp[%i];\n',i,j,(2*j-1));
end
else
if (2*j<=Adder_Tree(i-1))
```

```
fprintf(fid,'\t\tadder_%i_%i<=
adder_%i_%i+adder_%i_%i;\n',i,j,i-1,(2*j-1),i-1,2*j);
else
fprintf(fid,'\t\tadder_%i_%i<= adder_%i_%i;\n',i,j,i-1,(2*j-1));
end
end
end
end
fprintf(fid,'\t\tsumI_r<= sumI_temp;\n');
fprintf(fid,'\tend\n');

fprintf(fid,'endmodule\n');
%---------------------------------------------------
%module
VGA_DISPLAY(VGA_CAL_CLK,VGA_RST,Coord_X,Coord_Y,WriteAddrX,WriteAddrY
,WriteInBits,sumI,IsReadFlag)
%---------------------------------------------------
fprintf(fid,'module
VGA_DISPLAY(VGA_CAL_CLK,VGA_RST,Coord_X,Coord_Y,WriteAddrX,WriteAddrY
,WriteInBits,sumI,IsReadFlag,counter_fix);\n');
fprintf(fid,'\tinput VGA_CAL_CLK;\n');
fprintf(fid,'\tinput VGA_RST;\n');
fprintf(fid,'\tinput [9:0]Coord_X;\n');
fprintf(fid,'\tinput [8:0]Coord_Y;\n');
fprintf(fid,'\toutputreg [9:0]WriteAddrX;\n');
fprintf(fid,'\toutputreg [8:0]WriteAddrY;\n');
fprintf(fid,'\toutputreg [15:0]WriteInBits;\n');
fprintf(fid,'\toutputregIsReadFlag;\n');
fprintf(fid,'\tinput [10:0]sumI;\n');
fprintf(fid,'\tinput [7:0]counter_fix;\n');
fprintf(fid,'\treg Start;\n');
fprintf(fid,'\treg [7:0]counter;\n');
fprintf(fid,'\talways @ (posedge VGA_CAL_CLK or posedge VGA_RST)\n');
fprintf(fid,'\tbegin\n');

fprintf(fid,'\t\tif (VGA_RST)\n');
fprintf(fid,'\t\tbegin\n');
fprintf(fid,'\t\t\tIsReadFlag<=1''b0;\n');
fprintf(fid,'\t\t\tWriteAddrX<=Coord_X[9:0];\n');
fprintf(fid,'\t\t\tWriteAddrY<=Coord_Y[8:0];\n');
fprintf(fid,'\t\t\tWriteInBits<=16''h0000;\n');
fprintf(fid,'\t\t\tStart<=1''b0;\n');
fprintf(fid,'\t\t\tcounter<=8''b0;\n');
fprintf(fid,'\t\tend\n');

fprintf(fid,'\t\telse\n');
fprintf(fid,'\t\tbegin\n');
fprintf(fid,'\t\t\tif (counter >= counter_fix)\n');
fprintf(fid,'\t\t\tbegin\n');
fprintf(fid,'\t\t\t\tif (Start==1''b0)\n');
fprintf(fid,'\t\t\t\tbegin\n');
fprintf(fid,'\t\t\t\t\tStart<=1''b1;\n');
fprintf(fid,'\t\t\t\t\tWriteAddrX<=10''b0;\n');
fprintf(fid,'\t\t\t\t\tWriteAddrY<=9''d479-{sumI,2''b0};\n');
fprintf(fid,'\t\t\t\t\tWriteInBits<= 16''hFF00;\n');
fprintf(fid,'\t\t\t\tend\n');
fprintf(fid,'\t\t\t\telse if (WriteAddrX>=10''d639)\n');
fprintf(fid,'\t\t\t\tbegin\n');
fprintf(fid,'\t\t\t\t\tIsReadFlag<=1''b1;\n');
fprintf(fid,'\t\t\t\tend\n');
fprintf(fid,'\t\t\t\telse if (WriteAddrX[0]==1''b0)\n');
fprintf(fid,'\t\t\t\tbegin\n');
fprintf(fid,'\t\t\t\t\tWriteAddrX<=WriteAddrX+1;\n');
fprintf(fid,'\t\t\t\t\tWriteAddrY<=9''d479-{sumI,2''b0};\n');
fprintf(fid,'\t\t\t\t\tWriteInBits<=
(WriteAddrY==9''d479-{sumI,2''b0})? 16''hFFFF : 16''hFF00;\n');
fprintf(fid,'\t\t\t\tend\n');
fprintf(fid,'\t\t\t\telse\n');
fprintf(fid,'\t\t\t\tbegin\n');
fprintf(fid,'\t\t\t\t\tWriteAddrX<=WriteAddrX+1;\n');
fprintf(fid,'\t\t\t\t\tWriteAddrY<=9''d479-{sumI,2''b0};\n');
fprintf(fid,'\t\t\t\t\tWriteInBits<= 16''h00FF;\n');
fprintf(fid,'\t\t\t\tend\n');
fprintf(fid,'\t\t\t\tcounter<= 8''b0;\n');
fprintf(fid,'\t\t\tend\n');
fprintf(fid,'\t\t\telse\n');
fprintf(fid,'\t\t\tbegin\n');
fprintf(fid,'\t\t\t\tcounter<= counter + 1''b1;\n');
fprintf(fid,'\t\t\tend\n');
fprintf(fid,'\t\tend\n');

fprintf(fid,'\tend\n');
fprintf(fid,'endmodule');

fclose(fid);
fclose(fid_simu);
```

```
%---------------------------------------------------
%SIRTest.v
%---------------------------------------------------
fid=fopen('SIRTest.v','wt');
fprintf(fid,'`timescale 1ns/1ns\n');
fprintf(fid,'`include "SIR.v"\n');
fprintf(fid,'moduleSIRTest;\n');
fprintf(fid,'\tregclk,rst;\n');
fprintf(fid,'\twire [%i:1]S;\n',People_Num);
fprintf(fid,'\twire [%i:1]I;\n',People_Num);
fprintf(fid,'\tSIR SIR1(.clk(clk),.Reset(rst),.S(S),.I(I));\n');
fprintf(fid,'\talways\n');
fprintf(fid,'\tbegin\n');
fprintf(fid,'\t\t#10 clk=~clk;\n');
fprintf(fid,'\tend\n');
fprintf(fid,'\tinitial\n');
fprintf(fid,'\tbegin\n');
fprintf(fid,'\t\tclk=1''b0;\n');
fprintf(fid,'\t\t#5 rst=1''b1;\n');
fprintf(fid,'\t\t#10 rst=1''b0;\n');
fprintf(fid,'\tend\n');
fprintf(fid,'endmodule\n');
fclose(fid);
```

8.2 TOP MODULE

```
// --------------------------------------------------------------------
// -- Stochastic Simulation --
// -- Cornell MEng Project --
// -- By TianGao --
// -- Fall 2013 --
// -- Advisor: Bruce Land --
// -- on board DE2i-150 --
// --------------------------------------------------------------------

module DE2_TOP
	(
		////////////////////	Clock Input	////////////////////
		CLOCK_50,				//	50 MHz
		//////////////////// Push Button	/////////////////////
		KEY,
    ///////////////////    SWITCH     //////////////////////
    SW,
    ///////////////////      HEX     ///////////////////////////
    HEX0,
    HEX1,
    HEX2,
    HEX3,
    HEX4,
    HEX5,
    HEX6,
    HEX7,
    //////////////////////    SSRAM     ////////////////////////
    SRAM_BE,
    SRAM0_CS_N,
    SRAM1_CS_N,
    SRAM_ADSC_N,
    SRAM_GW_N,
    SRAM_ADSP_N,
    SRAM_ADV_N,
    SRAM_CLK,
    SRAM_OE_N,
    SRAM_WE_N,
    SRAM_DQ,
    SRAM_ADDR,
    ////////////////// VGA       ///////////////////////////
    VGA_CLK,                         //    VGA Clock
    VGA_HS,                          //    VGA H_SYNC
    VGA_VS,                          //    VGA V_SYNC
    VGA_BLANK,                       //    VGA BLANK
    VGA_SYNC,                        //    VGA SYNC
    VGA_R,                           //    VGA Red[9:0]
    VGA_G,                           //    VGA Green[9:0]
    VGA_B                            //    VGA Blue[9:0]
);

/////////////////////// Clock Input          ///////////////////////
input                  CLOCK_50;                    //    50 MHz
//////////////////////// Push Button    ////////////////////////
input       [3:0]KEY;
//////////////////////    SWITCH     ///////////////////////////
input       [17:0]SW;
//////////////////////      HEX        //////////////////////////////
output      [6:0] HEX0;
output      [6:0] HEX1;
output      [6:0] HEX2;
output      [6:0] HEX3;
output      [6:0] HEX4;
output      [6:0] HEX5;
output      [6:0] HEX6;
output      [6:0] HEX7;
///////////////////////      SRAM        //////////////////////////////
output    [3:0]SRAM_BE;
output                SRAM0_CS_N;
output                SRAM1_CS_N;
output                SRAM_ADSC_N;
output                SRAM_GW_N;
output                SRAM_ADSP_N;
output                SRAM_ADV_N;
output                SRAM_CLK;
output                SRAM_OE_N;
output                SRAM_WE_N;
inout      [31:0] SRAM_DQ;
output    [19:0]      SRAM_ADDR;
/////////////////////// VGA              ////////////////////////////
output                VGA_CLK;                    //    VGA Clock
output                VGA_HS;                     //    VGA H_SYNC
output                VGA_VS;                     //    VGA V_SYNC
output                VGA_BLANK;                  //    VGA BLANK
output                VGA_SYNC;                   //    VGA SYNC
output    [7:0]VGA_R;                       //    VGA Red[9:0]
output    [7:0]VGA_G;                       //    VGA Green[9:0]
output    [7:0]VGA_B;                       //    VGA Blue[9:0]

/////////////////////////////////////////////////////////////////////
// CLOCK
wire      CLK_MAIN;

// SRAM
assign    SRAM_BE         =    4'b0000; // Byte enable
assign    SRAM0_CS_N    =    1'b0;           // chip 0 enable
assign    SRAM1_CS_N    =    1'b0;       // chip 1 disable
assign    SRAM_ADSC_N =    1'b0;       // SRAM Controller Address Status
assign     SRAM_GW_N      =    1'b1;       // SRAM global Write Enable
//assign  SRAM_ADSP_N =    1'b1;      // SSRAM Processor Address Status;
assign    SRAM_ADV_N    =    1'b1;       // burst address advance
assign    SRAM_OE_N      =    1'b0;       // SRAM output Enable
```

```
//assign     SRAM_ADDR     =    19'b0;
assign    SRAM_CLK       =    CLOCK_50;
//assign  SRAM_DQ          =     16'hzzzz;

wire [31:0]     mSEG7_DIG;
reg    [31:0]    Cont;
wire        VGA_CTRL_CLK;
wire        AUD_CTRL_CLK;
wire [7:0]       mVGA_R;
wire [7:0]       mVGA_G;
wire [7:0]       mVGA_B;
wire [19:0]     mVGA_ADDR;              //video memory address
wire [9:0]    Coord_X, Coord_Y;   //display coods
wire        DLY_RST;

assign    TD_RESET      =     1'b1;      //     Allow 27 MHz input
//assign  AUD_ADCLRCK     =     AUD_DACLRCK;
//assign  AUD_XCK            =     AUD_CTRL_CLK;


Reset_Delay              r0    (      .iCLK(CLOCK_50),.oRESET(DLY_RST)    );

VGA_Audio_PLL           p1    (
.areset(~DLY_RST),.inclk0(CLOCK_50),.c0(VGA_CTRL_CLK),.c2(VGA_CLK)      );

MAIN_PLL                        p2 ( .areset(~DLY_RST),.inclk0(CLOCK_50),.c0(CLK_MAIN) );
VGA_Controller           u1    (     //     Host Side
                                          .iCursor_RGB_EN(4'b0111),
                                          .oAddress(mVGA_ADDR),
                                          .oCoord_X(Coord_X),
                                          .oCoord_Y(Coord_Y),
                                          .iRed(mVGA_R),
                                          .iGreen(mVGA_G),
                                          .iBlue(mVGA_B),
                                          //     VGA Side
                                          .oVGA_R(VGA_R),
                                          .oVGA_G(VGA_G),
                                          .oVGA_B(VGA_B),
                                          .oVGA_H_SYNC(VGA_HS),
                                          .oVGA_V_SYNC(VGA_VS),
                                          .oVGA_SYNC(VGA_SYNC),
                                          .oVGA_BLANK(VGA_BLANK),
                                          //     Control Signal
```

```
                                        .iCLK(VGA_CTRL_CLK),
                                        .iRST_N(DLY_RST)  );



    assign    mVGA_R = {Coord_X[0]? SRAM_DQ[0]:SRAM_DQ[15], 7'b0} ;
    assign    mVGA_G = {Coord_X[0]? SRAM_DQ[1]:SRAM_DQ[14], 7'b0};
    assign    mVGA_B = {Coord_X[0]? SRAM_DQ[2]:SRAM_DQ[13], 7'b0} ;
    //assign    mVGA_R = {SRAM_DQ[15:11],3'b0} ;
    //assign    mVGA_G = {SRAM_DQ[10:6],3'b0};
    //assign    mVGA_B = {SRAM_DQ[5:0],2'b0} ;
    //SIR MODULE
    wire [8:0]sumI;
    wire SIR_CLK,SIR_RST,SIR_RESET_I;
    wire ADDER_CLK;
    wire [150:1]SIR_S;
    wire [150:1]SIR_I;

    //VGA stuff

    wireIsReadFlag;                    //whether finish writing, 0 for not read, 1 for read
    reg Start;
    wire [9:0]WriteAddrX;              //SRAM X address for writing
    wire [8:0]WriteAddrY;              //SRAM Y address for writing
    wire [15:0]WriteInBits;        //16bit temporary write in SRAM
    wire VGA_CAL_CLK;
    //wire [7:0]counter_fix = 8'b00100000;    //VGA prescalar
    wire [7:0]VGA_counter_fix = {SW[7:1],1'b1};



    //Here is my code
    //assign SIR_CLK = SW[10];
    assign SIR_CLK = CLK_MAIN;
    assign ADDER_CLK = CLK_MAIN;
    assign VGA_CAL_CLK = CLK_MAIN;
    wire [15:0]temp_out;

/*  HexDigit HD0
    (
        .segs      (HEX0),
        .num       (temp_out[3:0])
    );
    HexDigit HD1
```

```
(
	.segs	(HEX1),
	.num	(temp_out[7:4])
);
HexDigit HD2
(
	.segs	(HEX2),
	.num	(temp_out[11:8])
);
HexDigit HD3
(
	.segs	(HEX3),
	.num	(temp_out[15:12])
);*/

RST_DELAY RST0		//delay the reset signal to avoid glitch
(
	.clk(VGA_CAL_CLK),
	.rst_in(~KEY[0]),
	.rst_out(SIR_RST)
);
RST_DELAY RST1
(
	.clk(VGA_CAL_CLK),
	.rst_in(~KEY[1]),
	.rst_out(SIR_RESET_I)
);
Infected_Adder_TreeADDR1(ADDER_CLK,SIR_RST,SIR_I,sumI);
SIR SIR1(
			.clk(SIR_CLK),
			.Reset_All(SIR_RST),
			.Reset_I(SIR_RESET_I),
			.S(SIR_S),
			.I(SIR_I),
			//.temp_out(temp_out)
		);
VGA_DISPLAY VGA_DIS(
					.VGA_CAL_CLK(VGA_CAL_CLK),
					.VGA_RST(SIR_RST | SIR_RESET_I),
					.Coord_X(Coord_X),
					.Coord_Y(Coord_Y),
					.WriteAddrX(WriteAddrX),
					.WriteAddrY(WriteAddrY),
                                        .WriteInBits(WriteInBits),
                                        .sumI(sumI),
                                        .IsReadFlag(IsReadFlag),
                                        .counter_fix(VGA_counter_fix)
                                    );

    assign SRAM_WE_N         = IsReadFlag? 1'b1 : 1'b0;
    assign SRAM_DQ[15:0] = IsReadFlag? 16'hzzzz :                    //if reading, let read
                                    WriteInBits;          //if writing, write 16 bits
    assign SRAM_ADDR         = IsReadFlag ?
                                                  {Coord_X[9:1],Coord_Y[8:0]}:          //
if it's reading, get 6bits X and 9bits Y because memory has 16bits
                                                  {WriteAddrX[9:1], WriteAddrY};
    assign SRAM_ADSP_N      = IsReadFlag ? 1'b0 : 1'b1;


endmodule //top module

module RST_DELAY
(
    clk,
    rst_in,
    rst_out
);
    inputclk;
    inputrst_in;
    outputregrst_out;
    reg [19:0] counter;
    always @ (posedgeclk)
    begin
        if (rst_in == 1'b0)
        begin
            counter<= 20'b0;
            rst_out<= 1'b0;
        end
        else
        begin
            counter = counter + 1'b1;
            if (counter == {1'b1,19'b0})
            begin
                rst_out<= 1'b1;
            end
        end
    end
endmodule
```

8.3 How To Use

Open the code generation.m in MATLAB. Set the parameters on the top and run. The MATLAB code will generate a SIR.v file in the folder. Copy the SIR.v file and paste it into the folder of the FPGA design. Compile the design in Quartus and download it in the FPGA.

KEY0 is reset with initialized seed and KEY1 is reset with random seed. The output is a standard VGA signal from board. Connect that to a VGA screen.