|  |
| --- |
| 임베디드시스템설계(001) |
| Term Project Report |
| Hardware acceleration of Black Scholes put option pricing model using Monte Carlo simulation |

|  |
| --- |
| Team 5: 김다훈, 류기민, 장재희  2014-6-19 |

Contents

[변경 내역 3](#_Toc390984789)

[1. Specification 4](#_Toc390984790)

[1.1 Name 4](#_Toc390984791)

[1.2 Objective 4](#_Toc390984792)

[1.3 Application 4](#_Toc390984793)

[1.4 Input 4](#_Toc390984794)

[1.5 Output 5](#_Toc390984795)

[2. Implementation 5](#_Toc390984796)

[2.1 FPGA 5](#_Toc390984797)

[2.1.1 Gaussian Random Number Generator 5](#_Toc390984798)

[2.1.2 Present\_vals 6](#_Toc390984799)

[2.1.3 Math 7](#_Toc390984800)

[2.1.4 Host\_itf 8](#_Toc390984801)

[2.1.5 Counter 8](#_Toc390984802)

[2.1.6 Monte Carlo 9](#_Toc390984803)

[2.2 Linux device driver 9](#_Toc390984804)

[2.2.1 Write Operation 9](#_Toc390984805)

[2.2.2 Read Operation 9](#_Toc390984806)

[2.2.3 Ioctl Operation 10](#_Toc390984807)

[2.3 JNI Library 10](#_Toc390984808)

[2.3.1 Write Operation 10](#_Toc390984809)

[2.3.2 Read Operation 10](#_Toc390984810)

[2.3.3 IOControl Operation 10](#_Toc390984811)

[2.4 Android application 10](#_Toc390984812)

[2.5 Serial communication program 11](#_Toc390984813)

[3. Result 12](#_Toc390984814)

[3.1 Simulation 12](#_Toc390984815)

[3.1.1 MATLAB simulation 12](#_Toc390984816)

[3.1.2 MODELSIM simulation 12](#_Toc390984817)

[3.2 FPGA test result 13](#_Toc390984818)

[참고문헌 14](#_Toc390984819)

# 변경 내역

|  |  |  |  |
| --- | --- | --- | --- |
| 날짜 | 변경자 | 버전 | 변경 사항 |
| 6.18 | 장재희 | 1.0 | 보고서 틀 작성, 보고서 초안 작성 |
| 6.19 | 류기민 | 1.1 | FPGA 구현 내용 작성 |
| 6.19 | 김다훈 | 1.2 | Device driver 구현 내용 작성 |
| 6.19 | 장재희 | 1.3 | Software 구현 내용 작성 |
| 6.19 | 장재희 | 1.4 | 보고서 포맷 통일, 글 교정 |
| 6.19 | 류기민 | 1.5 | 시뮬레이션 및 실제 테스트 결과 첨부. |

# Specification

## Name

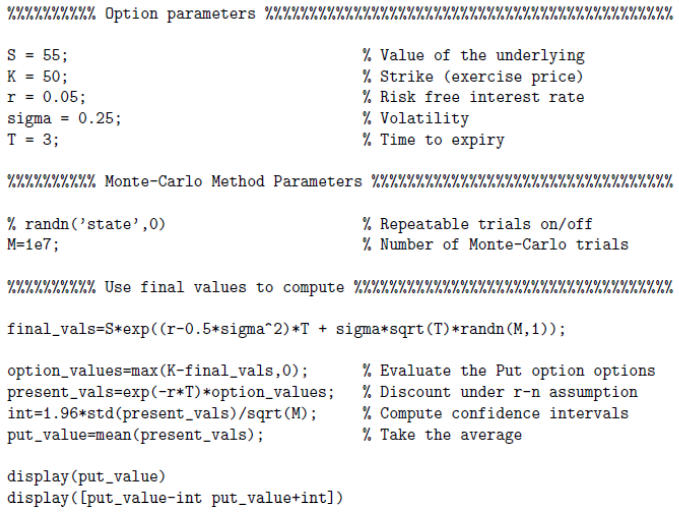
Monte Carlo simulation accelerator

## Objective

Hardware acceleration of an application using FPGA on HBE-SMIII-SV210

## Application

Black Scholes put option pricing model using Monte Carlo simulation.  
Computing equation and parameters are specified below.



## Input

S(value of the underlying), K(Strike-exercise price), r(risk free interest rate), sigma(Volatility), T(time to expiry), M(number of Monte-Carlo trials)

## Output

put\_value, range of put\_value

# Implementation

## FPGA

주어진 칩은 36개의 한정된 곱셈기를 가지고 있는데, 이 때, 연산을 병렬화시켜 더 많이 빨리 계산 할 수 있도록 **곱셈기를 적게 사용하는 계산 모델**을 만드는 것이 중점적인 설계 목표였다. 또한, floating point보다 fixed point로 표현된 값들의 연산속도가 더 *빠르므로* **fixed point 자료형**을 사용했다. Fixed point의 경우 소수를 표현할 수 없으므로, **원래의 값에 어떤 상수가 곱해진 값을 계산**해주었다. 즉, 예를 들자면 최종적으로 나온 put\_value의 값이 2.5인데, 원래 값의 100배를 곱한 250을 구해준다는 것이다.

이렇게 원래 값에 어떤 상수가 곱한 상태로 계산을 하게 되면, 이 상수로 나눠주는 연산이 필요하게 된다. 그런데 나눗셈을 하게 되면 많은 resource를 소모하게 되고 시간도 오래 걸린다. 반면 나눗셈 대신 bit shift를 사용하면 그만큼 시간과 resource에서 이득을 얻을 수 있을 것이라고 생각하였다. 따라서 나눗셈 대신 shift로 원래의 값을 얻을 수 있고, 정확도와 속도 및 resource간에 존재하는 trade off를 고려하여 위에서 이 상수를 **256()**으로 정했다.

## Gaussian Random Number Generator

Normal Distribution을 구하기 위해 **Central Limit Theorem**을 이용했다. Central Limit Theorem에 의하면, 평균 M, 표준편차 를 가지는 임의의 확률분포 에 대해 은 평균 M, 표준편자 을 가지는 Gaussian로 근사한다. 따라서 어떤 확률 분포를 만든 후 이것을 적절히 더하고 나누는 과정을 통해 Normal Distribution을 구할 수 있다.

이 프로젝트에서는 **12bit LFSR로 10bit의 Uniform Distribution**을 만들었다. 12bit LFSR은 -2048 ~ 2047까지 0을 제외한 값을 가질 수 있으며, 충분히 규칙적이지 않은 순서로 값을 얻을 수 있다. 구간의 크기가 충분히 크기 때문에 이 때의 평균은 0이라고 할 수 있다. 이 때 값이 보다 무작위한 것처럼 보이도록 10bit만을 선택했다.

한 LFSR에서 나온 연속된 8개의 값을 고르더라도, 각각의 분포는 Uniform Distribution에 같은 평균, 같은 표준편차를 가지고 있다. 이들을 더하면 평균 0의 Gaussian Distribution을 구할 수 있다. Gaussian Random Number값에 256이 곱해지려면[[1]](#footnote-1) 표준편차는 256이어야 하고, 8개를 더한 표준 편차는 큰 값이기 때문에, 이들을 256이 되도록 나누어주어야 하였다. 이 때 **비용이 큰 나눗셈 대신, shift와 덧셈 뺄셈 등의 비용이 작은 연산들의 합으로 근사**하였고 그 결과 표준편차가 256.6 정도인 Normal Distribution을 구했다. 생성된 random number는 present\_vals모듈로 전달된다.

## Present\_vals

최종적으로 계산되는 값들이 바로 present\_vals값 들이다. 식을 최종적으로 정리하면 다음과 같은 것을 볼 수 있다.

즉, 가 보다 클 때, present\_vals은 두 수의 차가 되고, 그렇지 않으면 0이 된다. 이 때, 초기에 주어지는 파라미터들로 인해서 모두 상수이다. 이 상수들을 다음과 같이 명명하였다.

이 상수들은 컴퓨터에서 보내준 데이터를 토대로 **안드로이드에서 미리 계산을 해서 256이 곱해진 상태로 FPGA로 전달된다**. 이 상수들을 이용하여 계산한 결과를 평균과 분산계산을 위해 math모듈로 넘겨주었다.

## Math

exponential의 계산은 **Taylor Series를 이용한 근사**를 이용했다. Exponential 계산을 해줄 때, exponent의 값도 256배가 곱해져 있고, 계산 값에도 256이 곱해져야 한다. 근사의 대상은 exp(exponent/256)이다. 따라서 다음과 같은 식으로 256이 곱해진 exponential 계산식의 값을 구할 수 있다.

256 \* (1+(exponent/256)+ (exponent/256)^2/2+(exponent/256)^3/6)

= 256 + exponent + exponent^2/256/2 + exponent^3/256^2/6

= 256 + exponent + exponent ^2 >> 8 /2 + exponent^3 >> 16 / 6

그런데 이 근사는 exponent의 값이 -256 ~ 256 사이의 값을 가질 때 유효하고, 그 외의 구간에서는 결과의 정확도가 떨어지는 것을 발견하였다. 따라서 이것을 극복하기 위해 다음과 같은 방법을 고안하였다.

모든 exp(x)는

( -256 < y < 256, k는 정수, x + 512k = y)

처럼, exp(y)와 exp(512\*k)의 곱으로 나타낼 수 있다. 이 때 x의 범위가 정도이므로 인 k에 대해 exp(512\*k)를 미리 계산의 parameter로 둘 수 있다. 또한, exp(512\*k)는 정해진 상수이므로 곱셈대신 이를 이진수로 계산해 exp(y)의 exponential 근사식에 **있는 shift와 함께 연산해주는 방식을 통해 곱셈기를 줄여줄 수 있었다**. 예를 들면 exponential 근사식의 3차항에 있는 1/6 같은 경우 1/6을 분모가 최대한 2의 거듭제곱에 가깝게 되도록 분자 분모에 170을 곱해 170/1020으로 만들었다. 이렇게 하면, 170을 2의 거듭제곱의 합으로 표현할 수 있고, 나누기 1020 대신 시프트 10으로 바꿔줄 수 있기 때문에 리소스를 아낄 수 있다.

Math 모듈은 이러한 근사를 이용하여 전달받은 present\_val의 합과 제곱의 합(분산 계산 시 이용)을 계산해 output으로 내보낸다. 이 때 평균과 interval을 바로 계산하지 않고, 안드로이드로 전달하여 계산하였다. 이를 통해 **resource를 줄이고 더 많은 병렬화를 꾀할 수 있었다.**

## Host\_itf

기존에 주어진 host.itf에서 모든 모듈들에 관한 변수들을 모두 제거하고, FPGA에서 present\_val의 합과 제곱의 합을 써주기 위한 레지스터, FPGA가 사용할 상수들을 안드로이드에서 계산하여 저장하기 위한 레지스터, 그리고 안드로이드로부터 명령을 전달받기 위한 mode register와 안드로이드에 상태를 전달해주기 위한 status register를 각각 추가했다.

|  |  |  |
| --- | --- | --- |
|  | 0 | 1 |
| Mode  (input) | **STOP**  (계산 정지, 모듈 초기화) | **START**  (계산 시작) |
| Status  (output) | **WORKING**  (계산 중) | **FINISHED**  (계산이 끝남) |

## Counter

정해진 연산횟수만큼 각 계산 모듈들에게 연산을 시키기 위해 만든 모듈이다. Status는 0으로 초기화 되어있는데, host\_itf의 mode bit이 1이고 count가 연산횟수보다 작을 때, counter를 증가시킨다. Count가 주어진 연산횟수에 도달하면 host\_itf의 status bit을 1로 바꾸고,, 안드로이드가 계산결과를 읽어가길 기다린다. 안드로이드로부터 stop signal이 들어오면 count를 초기화한다.

## Monte Carlo

위에서 언급했던 최적화 작업을 통해 Normal Generator, Present\_val, Math의 모듈 집합 하나당 사용하는 곱셈기를 6개까지 줄일 수 있었다. 보드의 곱셈기가 36개였으므로 이 모듈 집합을 **최대 6개까지 병렬화**할 수 있게 되었다. 구현시에는 5개의 계산모듈집합을 통해 총 연산횟수를 나누어 계산하였고, 모듈 집합이 한 개일 때에 비해서 **훨씬 빠른 속도로 계산할 수 있게 되었다.** 따라서 이 모듈 집합들을 하나의 모듈인 monte carlo로 다시 만들었다.

Monte Carlo 모듈은 Counter로부터 Status를 읽어와 계산을 수행한다. 그리고 5개의 모듈에서 나온 present\_val값들의 합과 제곱의 합을 합치기 위해 sum\_merge라는 모듈을 추가해주었다. 계산을 마치고 안드로이드가 데이터를 가져가서 stop signal을 보내면 각 math 모듈들은 계산한 값들을 0으로 초기화하고 다음 샘플들에 대한 연산을 준비한다.

## Linux device driver

### Write Operation

JNI로부터 시뮬레이션에 사용할 상수들을 입력 받고, 이를 FPGA에 전달한다. 앞의 FPGA 절에서도 설명한 것처럼 FPGA로 전달할 값들은 2.1.2에서 언급한 상수들을 fixed point로 나타낸 값이다. 이 값들은 JNI로부터 ‘|’로 연결된 문자열로 전달된다.

### Read Operation

FPGA로부터 Option Value의 합(sum)과 제곱합(square\_sum), 그리고 계산의 종료 여부(status)를 받아온다. 이 때 status가 1이면 시뮬레이션이 종료된 것으로, Android Application에서 반복적으로 Read Operation을 수행하며 status가 1이 될 때까지 계속 확인하는 방법으로 시뮬레이션의 종료 여부를 확인할 수 있다.

### Ioctl Operation

위의 Write Operation으로 FPGA에 전달한 값들을 이용하여 시뮬레이션 로직의 모드(mode)를 변경한다. Start Mode인 경우에는 시뮬레이션을 계속하며, Stop Mode인 경우에는 FPGA 내부의 sum과 square\_sum을 0으로 초기화한다.

Android Application에서는 Ioctl를 이용해 Stop Mode로 변경하여 초기화를 진행한 후, Write Operation으로 상수를 입력하고, 다시 Ioctl로 Start Mode로 변경하여 시뮬레이션을 시작한다. 시뮬레이션의 완료 여부는 Read Operation으로 확인하며, sum, square\_sum으로 시뮬레이션의 결과값을 구한다.

## JNI Library

### Write Operation

/dev/montecarlo에 값을 입력한다. 이 때 입력하는 값이 string이므로 jstring을 이용해 전달받는다. 이 때 이 값이 java String이므로 C style string으로 변환시키기 위한 GetStringUTFChars 함수를 사용하였다.

### Read Operation

/dev/montecarlo로부터 값을 읽어온다. 이 때 읽어오는 값이 char\*이므로 이를 jstring으로 변환시켜주어야 한다. 이 때 NewStringUTF 함수를 이용하였다. host\_itf의 status를 읽는 것 또한 이 함수를 사용한다.

### IOControl Operation

/dev/montecarlo에 host\_itf의 mode를 변경하는 값을 입력하는 함수이다.

## Android application

안드로이드에서는 컴퓨터로부터 ‘S|K|R|sigma|T|m’의 포맷으로 전송받은 데이터를 파싱하고 보드에서 필요한 상수계산을 미리 마친다. 이 때 오차를 최대한 줄이기 위해 모두 double로 받아온 다음 계산한 결과만 int로 casting하였으며 이는 다음과 같다.

|  |
| --- |
| // data parsing  // S|K|R|Sigma|T|m  //tmp = "a|b|c";  String inputdata[] = tmp.split("\\|");    double S = Double.parseDouble(inputdata[0]);  double K = Double.parseDouble(inputdata[1]);  double r = Double.parseDouble(inputdata[2]);  double sigma = Double.parseDouble(inputdata[3]);  double T = Double.parseDouble(inputdata[4]);  double M = Math.pow(10.0, Double.parseDouble(inputdata[5]));    int k\_ert = (int) (K \* Math.exp((-1.0) \* r \* T) \* 256.0);  int sigma\_sqrt\_t = (int) (sigma \* Math.sqrt(T) \* 256.0);  int s\_e05\_sigma\_t = (int) (S \* Math.exp(-0.5 \* Math.pow(sigma, 2.0) \* T) \* 256.0); |

이를 FPGA로 전달하고, FPGA에서 계산이 끝나면 이를 읽어와서 다음과 같이 interval을 계산하였다.

|  |
| --- |
| double put\_value = sum / 256.0 / M;  double interval = 1.96 \* Math.sqrt((square\_sum / Math.pow(256.0, 2.0)) / M - Math.pow(put\_value, 2.0)) / Math.sqrt(M); |

계산이 끝나면 이를 컴퓨터로 전송해주고, 다음 데이터 전송을 기다린다.

## Serial communication program

lab8에서 제공된 sample\_port.c를 customize해서 구현하였다. 프로그램은 세 가지 state로 나뉘어 동작한다. 포트 세팅이 끝나면 처음에 프로그램은 WRITE\_STATE에 진입하여 안드로이드로 데이터를 전송하고 전송이 끝나면 READ\_STATE로 진입한다. READ\_STATE에서는 안드로이드로부터 데이터를 기다린다. 이 때 데이터 전송의 종료는 ‘!’로 구분된다. 전송이 끝나면 전송한 데이터 내역을 data.out에 저장한다. data전송이 이것이 처음에 입력 받은 line\_num만큼 진행되면 프로그램이 종료된다.

# Result

## Simulation

시뮬레이션은 다음과 같은9개의 샘플을 가지고 MATLAB, MODELSIM에서 진행했고, 이 결과를 토대로 실제 하드웨어에서 연산한 값의 타당성을 고려하는데 사용했다.

|  |
| --- |
| **Input Sample (9 samples)**  – 55|50|0.05|0.25|3|7  – 34|35|0.05|0.55|5|7  – 34|35|0.05|0.55|5|8  – 24|21|0.05|0.25|5|7  – 24|21|0.05|0.25|5|8  – 24|21|0.35|0.05|5|7  – 50|60|0.01|0.02|5|7  – 60|55|0.02|0.05|5|7  – 60|55|0.02|0.15|5|7 |

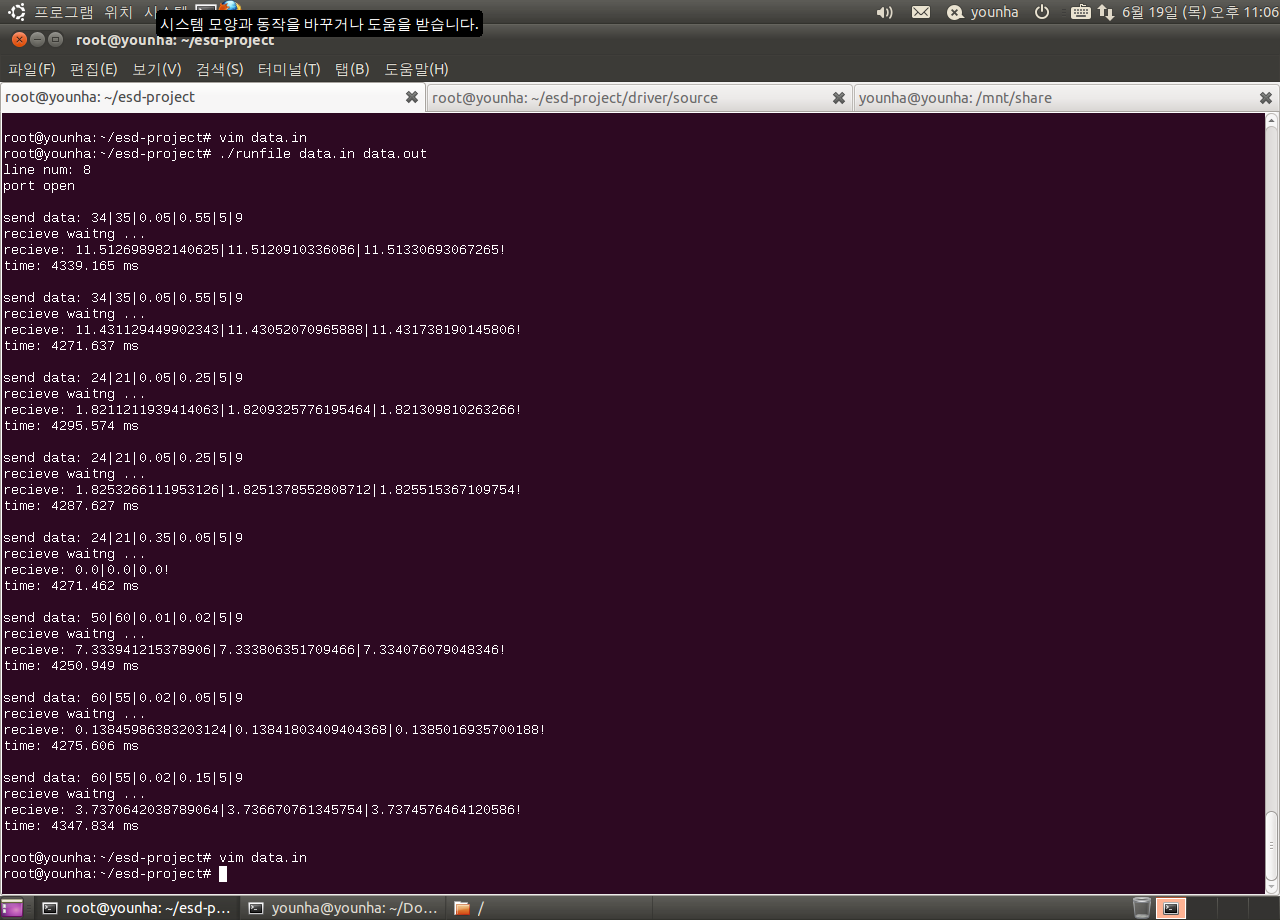
### MATLAB simulation

Put\_Value를 구할 때, MATLAB으로 구현하여 계산한 값이 매번 Gaussian Random Number Generator의 값들이 변하므로 Put\_Value의 값도 변하기는 하지만 주어진 계산식을 제일 정확하게 계산한다. 그에 따라 MATLAB시뮬레이션을 통해 계산한 값을 Put\_Value의 참값이라 가정하고, 모델심 결과와 실제 하드웨어의 값과 비교하는 척도로 사용했다. 시뮬레이션 결과는 부록 엑셀에 첨부한다.

### MODELSIM simulation

모델심 시뮬레이션을 통해 계산모듈이 얼마나 정확한지 시뮬레이션 해볼 수 있었다. 실제 시뮬레이션 결과 Put\_Value의 값들이 모두 Matlab의 값을 기준으로 3% 이내의 오차를 가지는 결과를 얻었다. 또한 interval의 값 역시 대부분이 3%이내의 오차를 가지고 있고, 원래 값이 작은 interval에 대해서는 오차가 크게 나타났다.

## FPGA test result[[2]](#footnote-2)



FPGA에 sample들을 입력해 Put\_Value값을 계산하는데 성공했다. 하지만 오차가 위의 두 시뮬레이션에 비해 매우 크게 나타났다. 이는 우리가 구현한 Random Generator가 알고리즘 상의 이유로 시간에 dependent한, 일종의 주기를 가지는 sample들을 생성하기 때문이다. 이런 시간에 대한 dependency가 얼마나 되는지 실험을 하기 위해 기존 샘플에서 m을 모두 9로 바꿔 FPGA에서 연산을 수행해봤다. 그 결과 평균적으로 수십 퍼센트의 오차를 가지고 있던 결과들이 10퍼센트대의 오차를 가지게 되었다. 즉, sample들이 많을수록 Random Number Generator의 시간에 대한 dependency를 줄여 보다 정확한 값을 얻을 수 있었다. 하지만 이 문제는 근본적으로 Random Number Generator의 알고리즘을 고쳐서 해결하는 것이 바람직할 것이다.

# 참고문헌

Ward, Roy William, and Timothy Christopher Anthony Molteno. Table of linear feedback shift registers. Electronics Group, University of Otago, 2012.

1. 2.1 [↑](#footnote-ref-1)
2. 검증은 함께 첨부한 team5\_simul\_data.xlsx를 참조 [↑](#footnote-ref-2)