國立清華大學
NATIONAL TSING HUA UNIVERSITY

Department of Computer Science and Information Engineering

IIS5008 Hardware Security Programming Assignment 3 Trusted Execution Environment of Multi-core System

黃偉祥 X1136010

Lecturer: Andy, Yu-Guang Chen

A report of IIS5008 course Program Assignment 3
Institute of Information Security

May 29, 2025

Declaration

I, Wei-Xiang Wong, of the Department of Computer Science and Information Engineering, National Tsing Hua University, confirm that this is my own work, figures, code snippets, artworks, and illustrations in this report are original and have not been taken from any other person's work, except where the works of others have been explicitly acknowledged, quoted, and referenced. I understand that failing to do so will be considered a case of plagiarism. Plagiarism is a form of academic misconduct and will be penalized accordingly.

I give my consent to having a copy of my report shared with future students as an example.

Wei-Xiang Wong May 29, 2025

Contents

1 About

1.1 Stochastic Computing (SC)

Stochastic Computing (SC) is a type of digital computation where values are represented probabilistically, and operations are performed using simple logic gates, such as AND, MUX, and XNOR. SC has recently regained attention due to its hardware friendliness, making it attractive for low-power, fault-tolerant, and edge computing applications.Alaghi et al. (2018)

In SC, a value is not represented in binary form, but rather through a bit-stream. Each bit in the stream does not indicate a fixed position; instead, it represents an instance of a probabilistic event. SC doesn't prioritize precision but rather focuses on delivering acceptable results under conditions of high fault tolerance, ultra-low hardware requirements, and resource-constrained environments.

1.1.1 Types of Stochastic Computing

- **Unipolar:** Encodes values between 0 and 1, commonly used for probabilistic multiplication.
- **Bipolar:** Uses symmetric encoding from -1 to 1, allowing it to handle both positive and negative values, making it suitable for neural network applications.

1.2 Compilation

To compile the **main.cpp**, execute the following commands in your terminal:

```
source /usr/cad/synopsys/CIC/pa_virtualizer.cshrc
source /home/tools/others/setup_systemc.csh 2.3.1
g++ -I$SYSTEMC_HOME/include -L$SYSTEMC_HOME/lib-linux64 main.cpp
    -lsystemc -o sim.o
```

> **Note:** Need to include the **sc_function_bipolar.h** and **error_injection.h** in **main.cpp**

1.3 Execution Example

Execute the simulation program **sim.o** to test the header files **sc_function_bipolar.h** and **error_injection.h**. A sample output is shown below:

```
[tsx1136010@linuxcad30 PA3]$ ./sim.o

        SystemC 2.3.1-Accellera --- Apr 24 2025 21:30:55
        Copyright (c) 1996-2014 by all Contributors,
        ALL RIGHTS RESERVED
******************************
Start stochastic test
Input 1 = 0.01
Input 2 = -0.2
SC Result: -0.00204003 int = -2190330
Exact Result: -0.002 int = -2147352
Error: 4.00285e-05 int = 42978
******************************

******************************
Start error test
Original value = 20000
Converted value (after attack): 11990
******************************
```

Output Analysis

- **Stochastic Test**:
  - Input values: 0.01 and -0.2
  - Computed error: 4.00285×10^{-5} (0.2% relative error)
- **Error Injection Test**:
  - Original value: 20000
  - Injected error: 40.05% reduction (20000 $\rightarrow$ 11990)

1.4 How to simulation

After source the above commands, type in **pct&** under directory **PA3**, then choose **Open Project** and select **PA3.xml** and open it.

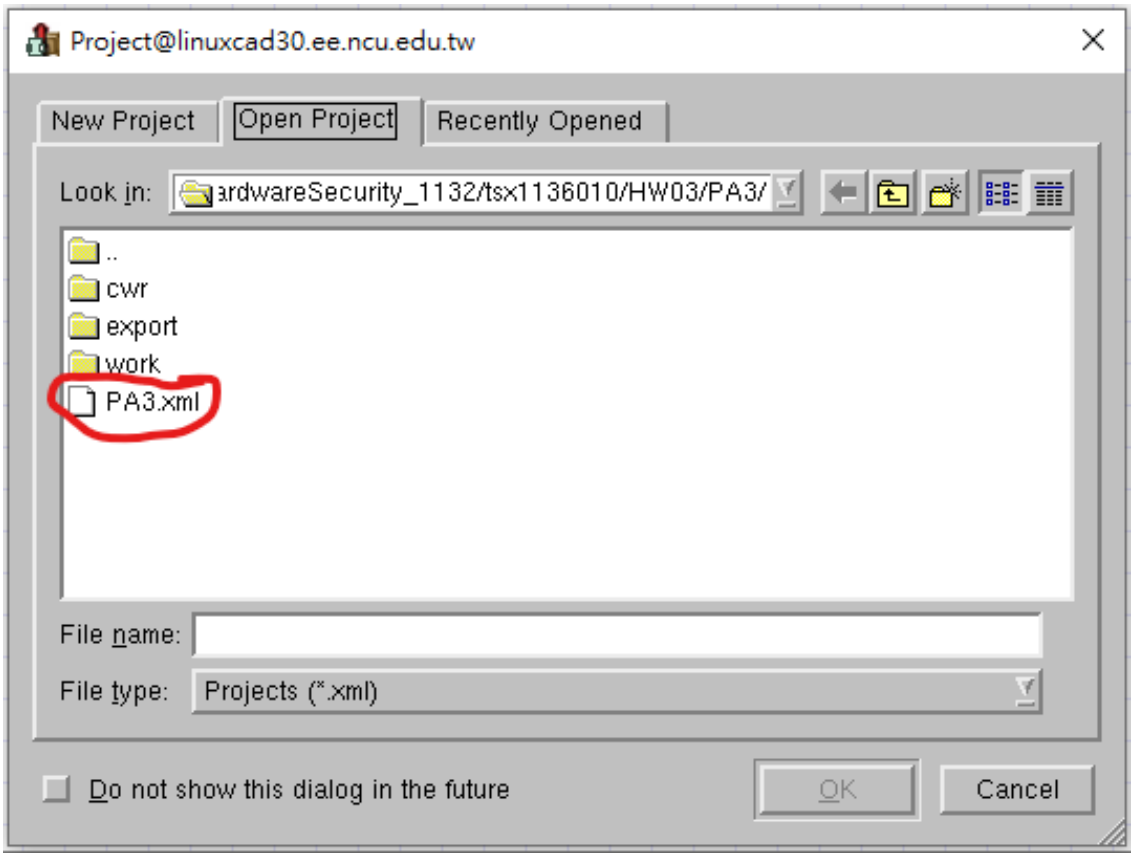

Figure 1.1: Open project

After that click the green button to start simulation the project(LeNet).

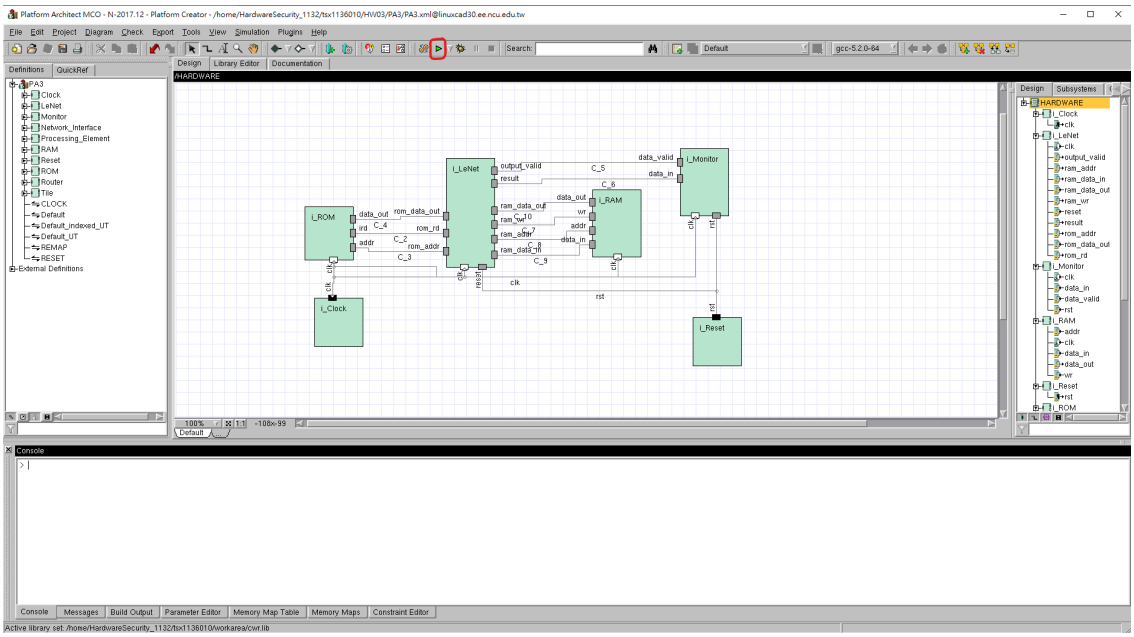

Figure 1.2: Start simulation

Then you can get your result that shown as the figure below.

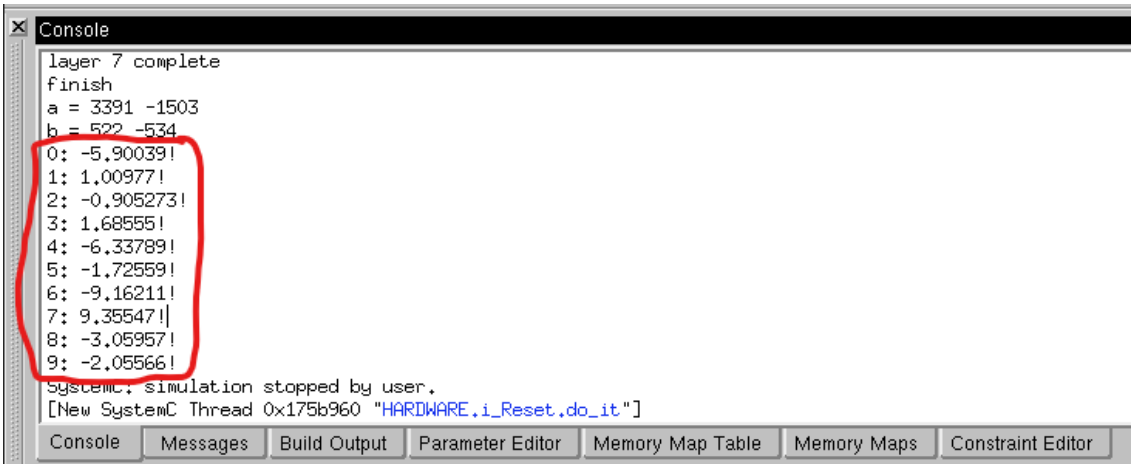

Figure 1.3: Simulation result

1.5 To use Stochastic computing

In the **Processing_Element.cpp**, line 292 change **SC_EN = true;** will use stochastic computing at layer 6, if set to **false** will use **binary computing**.

```
275         if(layer == 6){//Dense
276             for( int i = 0; i < 84 ; i++ ){
277                 ofmap[i] = 0;
278                 for( int j = 0; j < 120 ; j++ ){
279                     //normalised  normalize_sc_int16(); 9080 for a, 950 for b in all layer
280                     // 4500 for a, 540 for b in layer 6
281                     float a = ifmap[j]/4500.0f;
282                     float b = weight[(120+1)*i+j]/540.0f;
283                     a_max = max(static_cast<int>(ifmap[j]),a_max);
284                     a_min = min(static_cast<int>(ifmap[j]),a_min);
285                     b_max = max(static_cast<int>(weight[(120+1)*i+j]),b_max);
286                     b_min = min(static_cast<int>(weight[(120+1)*i+j]),b_min);
287                     // denormalize_float()
288                     // cout<<"a="<<ifmap[j]<<" nor="<<a<<" b="<<weight[(120+1)*i+j]<<" nor="<<b<<endl;
289                     sc_int<16> sc_result = (bipolar_mul(a, b, 2048)*4500.0f*540.0f)/1024;
290                     sc_int<16> bc_result = (ifmap[j] * weight[(120+1)*i+j])/1024;
291                     // cout<<"Real result="<<bc_result<<" sc_result="<<sc_result<<endl;
292                     SC_EN = true;
293                     if(SC_EN)ofmap[i] = ofmap[i] + sc_result;
294                     else ofmap[i] = ofmap[i] + bc_result;
295
296                 }
297                 if((ofmap[i] + weight[(120+1)*i+120]) >= 0)
298                     ofmap[i] = ofmap[i] + weight[(120+1)*i+120];
299                 else
300                     ofmap[i] = 0;
301             }
302         }
```

Figure 1.4: To use stochastic computing

1.6 To change error rate

In the **error_injection.h**, line 21 change the **error_rate** to target error rate.

```
17  int* error_injection(int input1, int bit_stream_length) {
18      // random_device rd;
19
20      // mt19937 g(rd());
21      float error_rate = 0.2;
22      // srand( time(NULL) );
23      vector<int> thresholds(bit_stream_length);
24      uniform_int_distribution<> dis(1, bit_stream_length);
```

Figure 1.5: To change error rate

2 LeNet

2.1 PE Arrangement & Layout

The layout of the **Processing Elements(PE)** are showed in the **Figure 2.1**

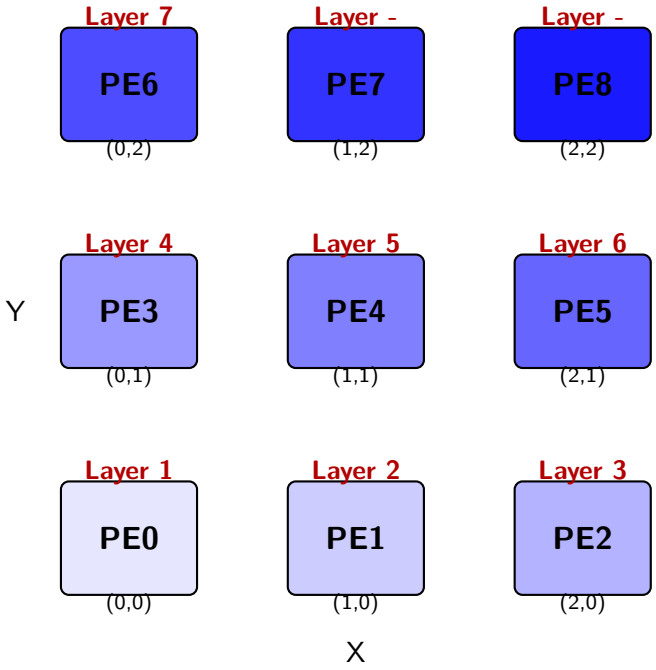

Figure 2.1: Processor Element Layout

In the **Tile.h** file, it define each PE coordination and responsible layer for **LeNet**.

```
// Tile Configuration (name, x, y, layer)
m_tile_0_0 ( "m_tile_0_0", 0, 0, 1 )
m_tile_1_0 ( "m_tile_1_0", 1, 0, 2 )
m_tile_2_0 ( "m_tile_2_0", 2, 0, 3 )
m_tile_0_1 ( "m_tile_0_1", 0, 1, 4 )
m_tile_1_1 ( "m_tile_1_1", 1, 1, 5 )
m_tile_2_1 ( "m_tile_2_1", 2, 1, 6 )
m_tile_0_2 ( "m_tile_0_2", 0, 2, 7 )
m_tile_1_2 ( "m_tile_1_2", 1, 2, 0 )
m_tile_2_2 ( "m_tile_2_2", 2, 2, 0 )
```

Listing 2.1: PE & Tile Configuration

The following **Table 2.1** shows the configuration of **Processing Elements** (PEs), including their **positions** in a 2D grid and the neural network **layer** each one is responsible for. Unassigned PEs are marked with a dash.

Table 2.1: PE Configuration

PE Element	Position	Responsible Layer
PE0	(0,0)	Layer 1
PE1	(1,0)	Layer 2
PE2	(2,0)	Layer 3
PE3	(0,1)	Layer 4
PE4	(1,1)	Layer 5
PE5	(2,1)	Layer 6
PE6	(0,2)	Layer 7
PE7	(1,2)	-
PE8	(2,2)	-

As shown above, the **3x3 grid** maps specific PEs to corresponding layers of **LeNet**. **PE7** and **PE8** are currently unassigned, which could allow for future expansion or redundancy.

2.2 Processing_Element.cpp

Each Processing Element(PE) follow a state machine pattern with the following states:

Algorithm 1 Processing Element State Machine

Initial State: `state = 0`

1: **State 0:** Read Data and Weights
- Layer-dependent operation
- Layer 1 reads:
  - Input data (external)
  - Weight values
- Other layers read weight values only

2: **State 1:** Read Intermediate Data
- Receives `ifmap` from previous PE
- Source depends on current layer

3: **State 2:** Computation Phase
- Performs calculation:
  - `ifmap` $*$ `weight` for Convolution & Dense
  - ReLu, Maxpooling, Flatten
- Stores results to `ofmap` buffer

4: **State 3:** Data Propagation
- Sends `ofmap` to:
  - `Data_out` interface
  - Next PE in pipeline (except Layer 7)
- Layer 7 sends directly to result buffer

5: **State 4:** Idle State
- Low-power waiting state
- Ready for next task assignment

Table 2.2: Summary of Processing Element State Machine

State	Phase	Description
0	Data Load	▪ **Layer 1**: Input + Weights ▪ **Layers 2-7**: Weights only
1	Data Fetch	▪ Receives `ifmap` ▪ Source: Previous PE
2	Computation	▪ Conv & Dense ▪ ReLU & MaxPool & Flatten
3	Data Route	▪ **Layer 1-6**: $\rightarrow$ Next PE ▪ **Layer 7**: $\rightarrow$ Result
4	Idle	▪ Low-power mode ▪ Ready state

2.3 LeNet.cpp

The weight loading process & wait PE done computation follows a state machine pattern with the following states:

Algorithm 2

1: **State 0:** Read input data and weight
   - PE_0 reads input from ROM addresses: 44426 to 45209
   - PE_0 reads weights from ROM addresses: 0 to 155
   - Counter increments from 0 to 940, then transitions to State 1
   - PE_0 reads from counter ≥ 1 until counter $= 940$

2: **State 1:** Read weight data
   - ROM addresses: 156 to 2571
   - Counter increments from 0 to 2416, then transitions to State 2
   - PE_2 reads weights from counter ≥ 1 until counter $= 2416$

3: **State 2:** Read weight data
   - ROM addresses: 2572 to 33411
   - Counter increments from 0 to 30840, then transitions to State 3
   - PE_4 reads weights from counter ≥ 1 until counter $= 30840$

4: **State 3:** Read weight data
   - ROM addresses: 33412 to 43575
   - Counter increments from 0 to 10164, then transitions to State 4
   - PE_5 reads weights from counter ≥ 1 until counter $= 10164$

5: **State 4:** Read weight data
   - ROM addresses: 43576 to 44425
   - Counter increments from 0 to 850, then transitions to State 5
   - PE_6 reads weights from counter ≥ 1 until counter $= 850$

6: **State 5:** All weight data loaded $\rightarrow$ transition to State 6

7: **State 6:** Wait for all layer to complete computation and PE_6 output result to Monitor

Table 2.3: LeNet Processing States Summary

State	Operation	ROM Addresses	Counter	Processing Element
0	Read input+weights	44426-45209 (input) 0-155 (weights)	0→940	PE_0
1	Read weights	156-2571	0→2416	PE_2
2	Read weights	2572-33411	0→30840	PE_4
3	Read weights	33412-43575	0→10164	PE_5
4	Read weights	43576-44425	0→850	PE_6
5	All weights loaded	-	-	-
6	Wait for completion	-	-	PE_6

Note: State in PE and LeNet is separate, and each PE compute calculation depends on its PE state.

2.4 LeNet Architecture

The architecture of the provided **LeNet** is shown in the **Table 2.4**. Included **Convolution** layer(ReLu),**Max Pooling** layer, Max Pooling layer(flatten), **Dense** layer.

Table 2.4: LeNet Architecture Summary

Layer	Type	Input Dim.	Output Dim.	Key Features
1	Convolution	28×28×1	24×24×6	5×5 kernel ReLU activation
2	Max Pooling	24×24×6	12×12×6	2×2 pooling (stride 2)
3	Convolution	12×12×6	8×8×16	5×5 kernel ReLU activation
4	Max Pooling	8×8×16	4×4×16	2×2 pooling (stride 2) Flatten to 256
5	Dense (FC)	256	120	Fully connected ReLU activation
6	Dense (FC)	120	84	Fully connected ReLU activation
7	Output	84	10	Fully connected 10-class output

3 My Design

3.1 Implementation of Bipolar Stochastic Computation

Algorithm 3 Bipolar Stochastic Multiplication (Flat Pseudocode)

function BipolarMul(x, y, N)
  If $x = 0$ or $y = 0$, return 0
  Generate threshold list $T = \{1, 2, \ldots, N\}$
  Shuffle T randomly
  Compute probability $P_x = \dfrac{x+1}{2}$
  Initialize counter $C_x = 0$
  For each i from 1 to N:
    Compute $b_x[i] = 1$ if $P_x \cdot N \geq T[i]$, else 0
    Update $C_x = C_x + b_x[i]$
  Compute reconstructed value $x_{\text{rec}} = 2 \cdot \dfrac{C_x}{N} - 1$
  Shuffle T again
  Compute probability $P_y = \dfrac{y+1}{2}$
  Initialize counter $C_y = 0$
  For each i from 1 to N:
    Compute $b_y[i] = 1$ if $P_y \cdot N \geq T[i]$, else 0
    Update $C_y = C_y + b_y[i]$
  Compute reconstructed value $y_{\text{rec}} = 2 \cdot \dfrac{C_y}{N} - 1$
  Initialize result counter $C_z = 0$
  For each i from 1 to N:
    Compute $b_z[i] = \neg(b_x[i] \oplus b_y[i])$
    Update $C_z = C_z + b_z[i]$
  Compute $P_z = \dfrac{C_z}{N}$
  Compute final result $z = 2 \cdot P_z - 1$
  Return z
end function

x is **input1**, **y** is **input2**, **N** is **bit_stream_length**. Multiplication in bipolar using **XNOR** to b_x and b_y then count the number of one in final bit stream b_z and map back to [-1,1]. **z** is the final result of **x*y**.

3.2 Implementation of Error injection

Algorithm 4 Error Injection in Bitstream

function ErrorInjection(x, N)
  Initialize error rate $error\_rate \leftarrow 0.2$
  Initialize threshold list $T \leftarrow \{1, 2, \ldots, N\}$
  Shuffle T randomly
  Compute probability $P_x = \dfrac{x + 32767}{2 \cdot 32767}$
  Initialize counter $C_x = 0$
  For each i from 1 to N:
    Compute $b_x[i] = 1$ if $P_x \cdot N \geq T[i]$, else 0
    Update $C_x = C_x + b_x[i]$
  Compute reconstructed value $x_{\text{rec}} = 2 \cdot 32767 \cdot \dfrac{C_x}{N} - 32767$
  Shuffle T again for error injection
  For each i from 1 to $N \cdot error\_rate$:
    Compute $atk \leftarrow T[i] - 1$
    Flip $b_x[atk]$
  Recompute the number of ones after error injection:
  Initialize $C_x = 0$
  For each i from 1 to N:
    Update $C_x = C_x + b_x[i]$
  Compute reconstructed value after attack $x_{\text{rec}} = 2 \cdot 32767 \cdot \dfrac{C_x}{N} - 32767$
  Convert reconstructed value to integer: $result \leftarrow \text{round}(x_{\text{rec}})$
  Initialize bitstream array bin of size 16
  For each i from 1 to 16:
    Compute $bin[i] = (result >> i)\&1$
  Return bin
end function

Use **32767** to normalize the **x** because **INT16_MAX** is 32767, use the similar method in bipolar stochastic computing to get the bit stream and flip the bit randomly depends on the error rate and the shuffled threshold. Return the error injected 16 bits of binary bit stream, e.g. the error injected value is 15 then will return [0000000000001111].

4 Comparison BC & SC

4.1 Experiments setting

Each result uses the same fixed random seed (1136010). **Stochastic Computing(SC)** is used in the **layer 6**, and **normalize** the **ifmap** with **4500.0f** and **normalize** the **weight** with **540.0f**.The value used to perform normalization will be discuss in the **Conclusion** section. The accuracy is calculated as:

$$\text{Accuracy} = 1 - \frac{|x - y|}{x},$$

where x is the predicted value before error injection and y is the predicted value after error injection.

4.2 Result

Table 4.1: Error Injection Ratio vs Accuracy of BC and SC

Error Injection Ratio (%)	Result of BC	Result of SC	Avg of SC
0.00	14.86	14.92	14.76
5.00	13.04	13.25	13.08
10.00	12.52	12.75	11.88
15.00	11.15	10.13	9.61
20.00	10.05	9.36	8.60
25.00	9.90	9.32	7.26
30.00	8.20	8.41	5.72
35.00	6.18	5.98	4.20
40.00	5.91	5.63	2.90
45.00	3.49	3.75	0.94
50.00	2.05	1.42	−0.01

Table 4.2: Error Injection Ratio vs Accuracy of BC and SC

Error Injection Ratio (%)	Accuracy of BC (%)	Accuracy of SC (%)	Avg Accuracy of SC(%)
0.00	100.00	99.63	99.10
5.00	87.74	89.18	88.03
10.00	84.25	85.78	79.96
15.00	75.03	68.17	64.65
20.00	67.60	62.95	57.86
25.00	66.61	62.71	48.85
30.00	55.17	56.61	38.49
35.00	41.61	40.23	28.29
40.00	39.77	37.87	15.36
45.00	23.46	25.24	6.30
50.00	13.80	9.53	0.00
55.00	0.00	0.00	0.00

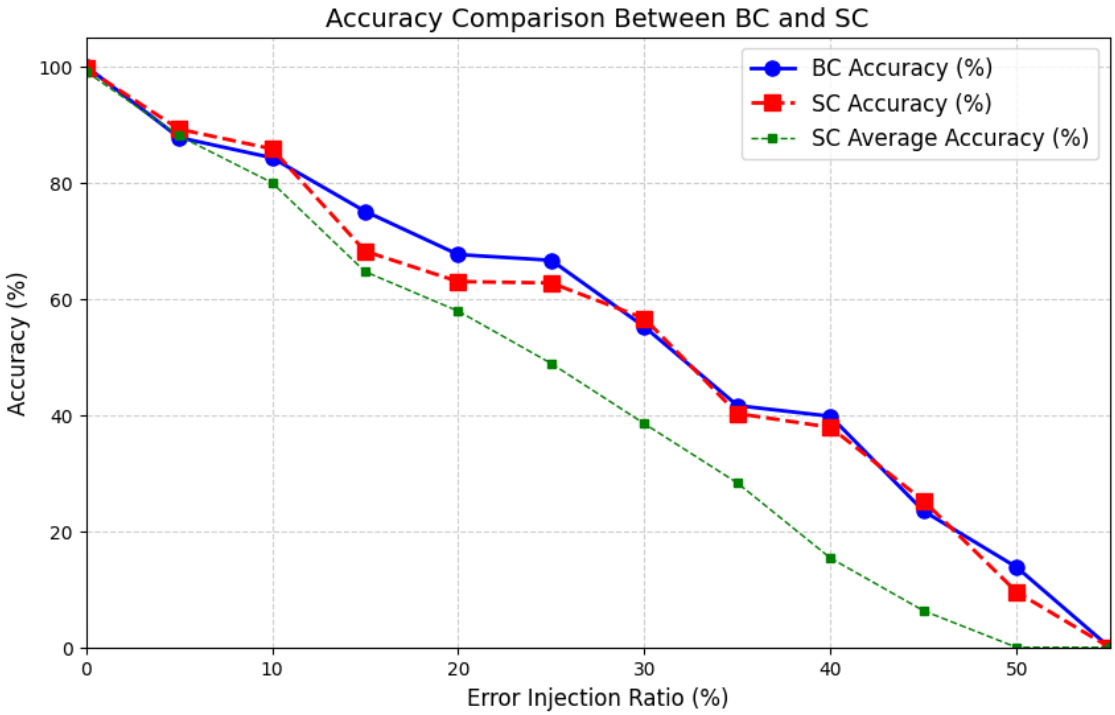

Figure 4.1: Enter Caption

5 Conclusion

Bit stream length is important, the longer bit stream length can have more accurate on the final result of stochastic computing.

The number to use for **normalization** is also important, if the value range is [-100,100] but use 32767 to normalize, then will higher chance that after map to [0,1] will not able to separate the different of 10 and 100. Because both are very small after normalized, so the probability will get very near to 0.5 any way.

E.g. : $10/32767 \simeq 0$ and $100/32767 \simeq 0$, after map to [0,1] probability will become very close to 0.5, this means any value will get a 0.5 probability of 1 in SC, not able to differential them.

So I first calculate the **maximum** and **minimum** number for layer 6(ifmap and weight separately) in the normal condition(binary computing mode with 0% error injection). Then I use this largest absolute value to do the normalize for ifmap and weight, after that de-normalize the result back to binary number.

I think that the document of PA3 is not very clear, luckyly TA explains on eeclass and summarize the discussion to the PA3 document.

References

Alaghi, A., Qian, W. and Hayes, J. P. (2018), 'The promise and challenge of stochastic computing', *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* **37**(8), 1515–1531.