Face Recognition using Approximate Arithmetic

Marso, Karol ; Nannarelli, Alberto

Publication date:
2018

Document Version
Publisher's PDF, also known as Version of record

Citation (APA):
Marso, K., & Nannarelli, A. (2018). Face Recognition using Approximate Arithmetic. DTU Compute. DTU Compute Technical Report-2018, Vol.. 02

Technical University of Denmark

Face Recognition using Approximate Arithmetic

Technical Report

Author:
Karol Marso
Supervisor:
Alberto Nannarelli

This work was done as part of the coursework in "Design of Arithmetic Processors and Digital Systems Laboratory" under the supervision of A. Nannarelli

Abstract

Face recognition is image processing technique which aims to identify human faces and found its use in various different fields for example in security. Throughout the years this field evolved and there are many approaches and many different algorithms which aim to make the face recognition as effective as possible. The use of different approaches such as neural networks and machine learning can lead to fast and efficient solutions however, these solutions are expensive in terms of hardware resources and power consumption. A possible solution to this problem can be use of approximate arithmetic. In many image processing applications the results do not need to be completely precise and use of the approximate arithmetic can lead to reduction in terms of delay, space and power consumption. In this paper we examine possible use of approximate arithmetic in face recognition using Eigenfaces algorithm.

1 Introduction

Face recognition is an image processing technique which aims to identify persons based on their faces. Applications of face recognition can range from various security applications (e.g. unlocking the mobile phone with a face) to search for people based on their face in online databases. The whole process of face recognition is rather difficult task and requires a lot of computations. The first face recognition techniques were introduced in 1960s and were mostly semi-automated. The face recognition was mostly based on identifying specific features such as hair color, lips, or eye color. The face recognition can be divided into two basic steps. In the first step the face recognition algorithm must identify a face and in the second step algorithm extracts important features used to identify the face [6].

The algorithms for face recognition can be divided into 12 categories:

- Geometric feature based methods
- Template based methods
- Correlation based methods
- Matching pursuit based methods
- Singular value decomposition based methods
- The dynamic link matching methods
- Illumination invariant processing methods
- Support vector machine approach
- Karhuen-Loeve expansion based methods
- Feature based methods
- Neural networks based algorithms
- Model based methods

The details of these categories are beyond the scope of this paper and are explained in depth in [6]. From now on we focus mostly on Eigenfaces face recognition method which is Karhuen-loeve expansion based method.

As mentioned the face recognition methods are often computationally complex operations and performing them takes a lot of computational power and energy. Moreover, the face recognition is more often applied on devices which require lower power consumption such as mobile phones. In [9] the authors are examining low-power computations using mobile GPU to recognize faces. The results show that even though mobile GPU uses more power compared to mobile CPU it performs operations much faster and in total uses less energy then mobile CPU.

Another approach to reduce the energy consumption can be use of approximate arithmetic. The face recognition uses multiple different types of arithmetic operations which can be made much more efficient when using approximate arithmetic.

In this paper we examine possible approximate arithmetic units that can be implemented in order to reduce complexity (Section 2). In Section 3 we briefly explain Eigenfaces algorithm. In Section 4 we show the results of the simulation of Eigenfaces algorithm and we compare the differences between exact and approximate implementation. Section 5 describes hardware implementation and results of the testing on a hardware. Section 6 concludes the paper.

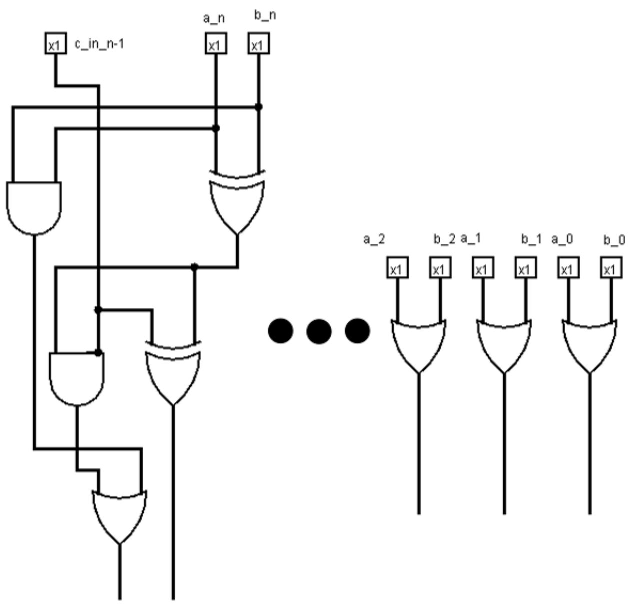


Figure 1: Scheme of the lower or adder

2 Approximate Arithmetic Operations

When we use full precision arithmetic operations, many times it is needed to spend many resources in terms of scale, delay and power consumption. On the other hand approximate arithmetic operations may produce wrong result however, they can lead to substantial cost savings and many application do not need fully precise results. In this section we examine some of the possible implementations of approximate arithmetic operations.

2.1 Lower OR Adder

One of the simplest implementations of the approximate adder is a Lower OR Adder. N-bit Lower OR Adder consists of i bits of ORs in the least significant part and j bit precise adder in the most significant part [11].

The equations below show error characteristics of the adder. Equation 1 defines the error probability of the adder, equations 2 and 3 define maximal and minimal error, equations 4 and 5 state the mean error and mean square error of the adder.

$$P(error) = 1 - (3/4)^i \tag{1}$$

$$MAX_{error} = 2^{(i-1)} - 1 \tag{2}$$

$$MIN_{error} = -2^{(i-1)} \tag{3}$$

$$ME = -0.25 \tag{4}$$

$$MSE = 2^{(2*i-4)} \tag{5}$$

The total gate count of the adder is described by Equation 6.

$$Gate_{Count} = Gate_{Precise}(j) + (i+1) \tag{6}$$

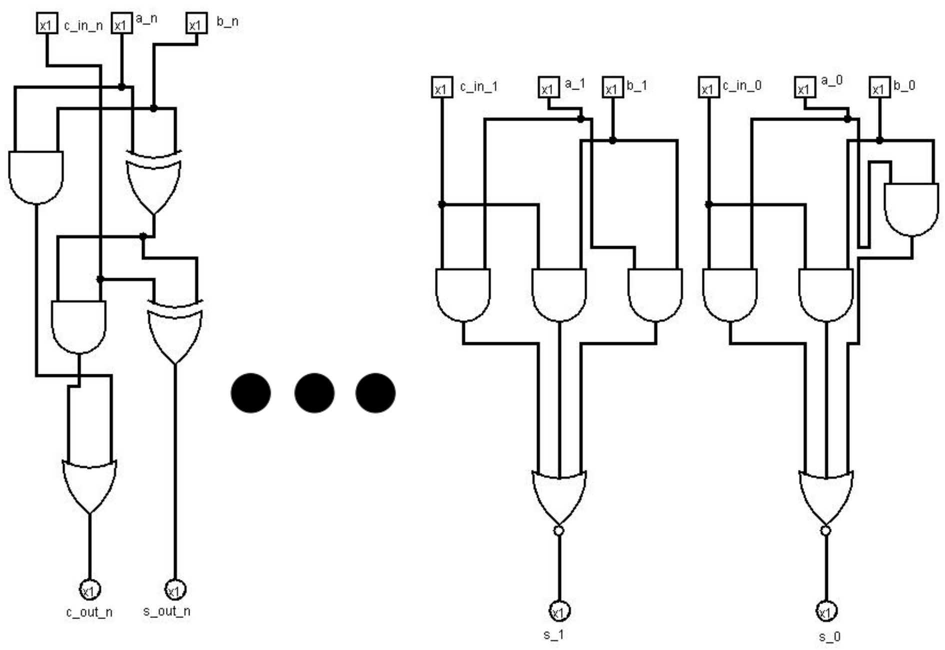


Figure 2: Design of approximate mirror adder proposed in approximation 2

2.2 Mirror Adder

In [4] authors propose simplification of the mirror adders which is one of the most common implementations of full adders.

Authors propose three approximations which help to reduce the complexity of the adder. The first approximation is based on the fact that the sum of operands usually results in

$$sum = \overline{c_{out}} \tag{7}$$

This is true for 6 out of 8 cases and produces the error in 3 cases of sum and 1 error in cout. Another approximation that authors are stating is that carry out is in 6 cases out of 8 is either A or B and can be used interchangeably to produce similar results.

$$c_{cout} = A \tag{8}$$

The last possible approximation authors state is to use one operand as the sum and the other as carry out.

$$sum = A \tag{9}$$

$$c_{out} = B \tag{10}$$

In this case the sum is correct in 4 out of 8 cases and carry out is correct in 6 out of 8 cases.

3 Eigenfaces Face Recognition

Eigenfaces treat the face recognition as two dimensional recognition problem. This algorithm was first proposed in [7]. The goal is to use information theory applied to face encoding by extracting the most relevant information about the face. This means that we need to compute eigenvectors of the covariance matrix of the set of face images. Eigenvectors are used to generate eigenfaces which consist of weighted sums from a small collection of characteristic images for every face. Face recognition is then performed by comparing the face with these eigenfaces. The algorithm can be described in the steps below:

- Initialization - acquiring the training set of faces and calculation of the eigenfaces.
- Classify the weight pattern as category of a face.

The computation is based on the assumption that similar faces will not be randomly distributed in the image space which is based on Karhunen-Loeve expansion which searches for the best distribution of the face images around the image space.

At first we need to compute average of the images for every pixel ψ from M images.

$$\psi = \frac{1}{M}\sum_{i=1}^{M}\gamma_i \tag{11}$$

We then compute the difference between the original picture and the eigenface which is called error ϕ. This error is then used to compute covariance matrix C.

$$\phi = \gamma_i - \psi_i \tag{12}$$

$$C = \frac{1}{M}\sum_{n=1}^{M}\phi_n * \phi_n^t = A * A^T \tag{13}$$

The resulting matrix C is however, of size N^2 by N^2 which makes it a computationally difficult task. The solution is to solve much smaller M by M matrix problem. From the analysis in [10] we get a much simpler way to compute the difference between Eigenface and the original picture when we just need to compute linear combination of the M training faces to form the eigenface.

$$u_l = \sum_{k=1}^{M} v_{lk} * \phi_k \qquad l = 1, ..., M \tag{14}$$

When classifying images we have to compute vector of weights called w_k. This vector is based on eigenface u, pixels of the classified face and error ϕ of the eigenface.

$$w_k = u_k^T(\gamma - \phi) \qquad k = 1, ..., M' \tag{15}$$

The computed weights form a vector $\Omega^T = [w_1, w_2, ..., w'_M]$. This vector is used to compute error $e_k = ||\Omega - \Omega_k||$ where Ω_k describes a *kth* face class. The face belongs to a class if minimal error of e_k is below a certain threshold Θ_e.

4 Eigenfaces using Approximate Arithmetic

We are using a simplified version of the algorithm however, it offers a lot of space to experiment with approximate adders due to its simplicity in order to make the whole implementation much more efficient. We especially aim to reduce the complexity of the arithmetic operations mostly in terms of approximate arithmetic.

To create an eigenface we need to compute the average as shows pseudocode below

The other part of the algorithm is the scoring where we compare a sample with computed ghost face. It is described in pseudocode below

We changed the whole code a bit and decided to use absolute value instead of multiplication.

In our experiments we tested three different variations of adders with different imprecise bit widths. For the part that generates ghosts, we used 14-bit adder due to the maximal possible sum of the values. For the scoring part we used the bit width of 21. For precise part we used

Algorithm 1 Training part of the algorithm

```
for i = 0 to M do
  for j = 0 to image_size do
    average[j] = average[j] + image[i][j]
  end for
end for=0
  for i = 0 to image_size do
    average[i] = average[i]/M
  end for=0
```

Algorithm 2 Scoring part of the algorithm

```
for i = 0 to M do
  for j = 0 to image_size do
    distance = sample[j] - image[i][j]
    error[j] = error[j] + (distance*distance)
  end for
end for
for i = 0 to M do
  error[j] = error[j]/(image_width * image_height)
end for=0
```

only simple carry ripple adders. In Table 1 and Table 2 we can see the gate count difference between adders. The inexact part length of 0 means that the adder is precise.

As we can see, out of all the adders Mirror adder approximation 3 presented in [4] uses the least amount gates.

For ghost generation we changed the addition using three different approximate adders. We were testing our solution on approximate bit sizes of up to 7 bits with bit width of 14. In Figure 3 we can see ghosts generated using different approximate adders with different length of approximate parts.

We generated 8 different ghosts using multiple configurations of the adder. Table 3 shows the average error from generating 8 ghosts. We present only results from the adders with at least 4 inexact bits, because they offer much better area savings.

The worst average error is about 20 %, however, the savings in gate count can be up to 50% with mirror adder approximation 3.

We made two experiments on scoring. In our first experiment we tested different combinations of approximate adders and identified the sample face. Because there were too many possible combinations we only used Mirror adder approximation 3 and Lower OR Adder which

Inexact part length	LOA	Mirror adder 2	Mirror adder 3
0	70	70	70
1	67	69	65
2	63	68	60
3	59	67	55
4	55	66	50
5	51	65	45
6	47	64	40
7	43	63	35

Table 1: Gate count of the adders used in ghost generation

Inexact part length	LOA	Mirror adder 2	Mirror adder 3
0	105	105	105
1	102	104	100
2	98	103	95
3	94	102	90
4	90	101	85
5	86	100	80
6	82	99	75
7	78	98	70
8	74	97	65

Table 2: Gate count of the adders used for scoring

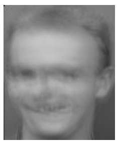
Inexact part length	LOA	Mirror adder 2	Mirror adder 3
4	3.54 %	3.57 %	3.48 %
5	4.60 %	4.76 %	4.43 %
6	7.94 %	8.20 %	7.56 %
7	19.93 %	18.28 %	19.00 %

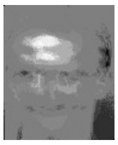
Table 3: Average error in generated ghosts

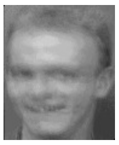
could save the most space and the tests on Ghost generation showed that the error was quite small.

The Scoring algorithm was successful in all cases with the imprecise part with size of up to 7 bits. However, when we were using imprecise adder with the size of 8 bits, in some cases it identified a sample as a different face. These cases occurred when the ghost was generated using 7 bit LOA adder and 8 bit imprecise adder. Tables 4 and 5 show the average errors in scoring using Mirror adder approximation 3 and ghosts generated using Mirror adder approximation 3 and Lower OR adder.

The first test show rather interesting results. While until imprecise size of 7 bits the error is relatively small, the imprecise length of 8 produces huge errors. On the other hand the imprecise part of 6 bits introduces relatively small errors. We therefore decided to test the configuration of Lower OR adder for ghost generation with imprecise size of 7 and Mirror adder approximation with imprecise part bit size of 6. The tests showed that of all 24 testing samples every single was recognized correctly. Overall the image recognition for scoring with up to 7


(a) Ghost using exact adder


(b) Ghost generated with LOA with 7 inexact bits


(c) Ghost generated using mirror adder approximation 2 [4]
with 5 imprecise bits

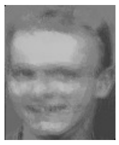

(d) Ghost generated using mirror adder approximation 3 [4] with 6 imprecise bits

Figure 3: Ghosts generated using different imprecise adders

Training part imprecise bits	Scoring part imprecise bits				
	4	5	6	7	8
4	0.76%	2.68%	0.76%	1.14%	112.26%
5	1.53%	1.92%	0.38%	3.46%	114.61%
6	0.73%	3.32%	4.79%	4.05%	109.96%
7	4.15%	1.03%	4.49%	12.4%	115.91%

Table 4: Average error of the Scoring algorithm using Mirror adder approximation 3 for ghost generation and for Scoring

Training part imprecise bits	Scoring part imprecise bits				
	4	5	6	7	8
4	0.38%	0.38%	3.46%	4.23%	110.38%
5	0.00%	1.16%	2.71%	12.79%	113.17%
6	1.83%	1.09%	6.95%	21.97%	105.86%
7	1.71%	1.37%	3.09%	16.49%	121.30%

Table 5: Average error of the Scoring algorithm using Lower or adder for ghost generation and Mirror adder approximation 3 for scoring

bits of inexact length the detection was always correct. However, images generated with 7 bit Lower Or Adder were identified wrongly in 10% of the cases. On the other hand, the success rate was still 90%.

5 Hardware Implementation

The simulations showed us that imprecise arithmetic can be used for face recognition with high success rate. We then implemented and tested our solution on the actual hardware. Our face recognition system consists of the two parts - training part and scoring part.

The training is performed using a C program on a PC. We created a training set made of 400 photos categorized as 40 faces which are then forming 40 ghosts. The pictures were obtained from [12]. The size of the pictures is 92x112. The scoring part of the algorithm was implemented in the FPGA board Digilent Nexys 4 DDR [13]. The results of simulations showed us that we can use higher number of imprecise bits and still have high success rate of the face recognition. In our implementation, we were using Lower Or Adder with 6 imprecise bits and bit width of 9 bits for subtraction (computing the difference between the sample and a ghost) and the second adder with 7 imprecise bits with bit width of 21.

5.1 Accelerator Architecture

To compare our results we created a precise implementation using structures similar to tensors. The top architecture of the accelerator is described in Figure 4.

The PC is connected to the accelerator using a USB cable and the communication between PC and FPGA is done using UART. Further details of the communication and control flow of the accelerator are of less importance and are not further discussed in the paper.

The accelerator unit is controlled as a standard digital system with a controller (FSM) sending control signals to the datapath. The datapath has an architecture of tensors as is shown in Figure 5. It computes the error by performing subtractions between ghost pixel and sample pixel, then creates the absolute value of the subtraction. The result is then added with

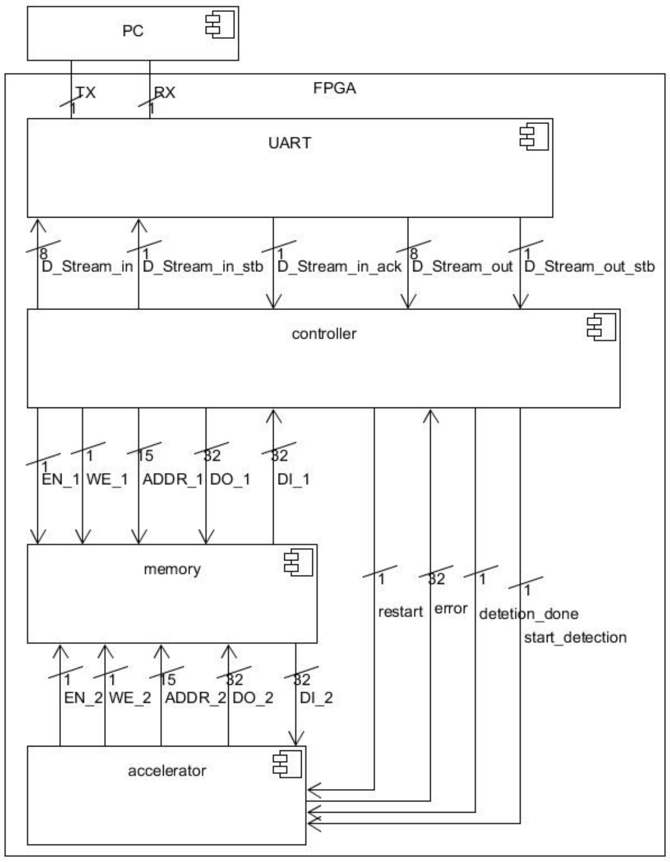


Figure 4: Architecture of the accelerator

the error matrix of four errors. These four errors are in the end added together and returned as a result. Figure 5 shows the whole process.

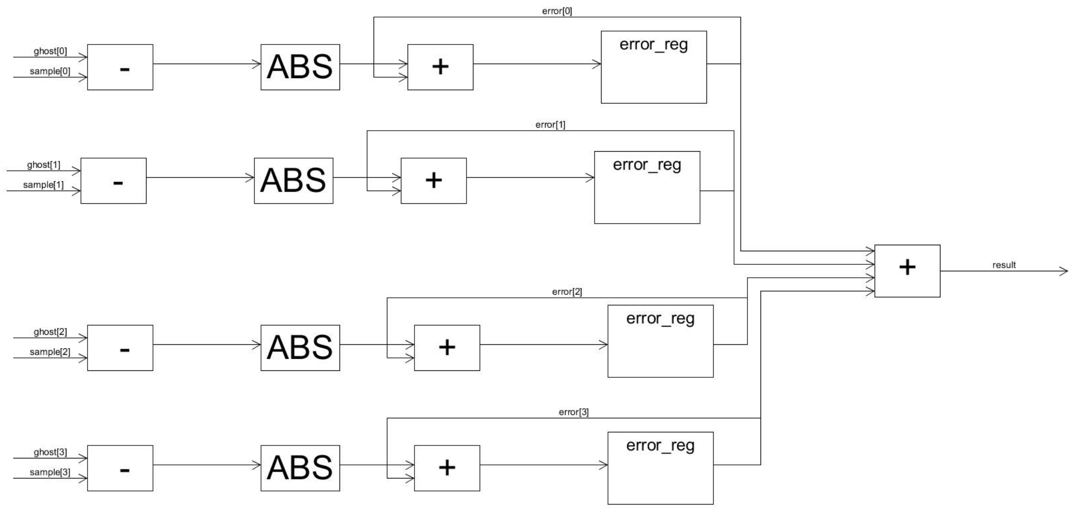


Figure 5: Architecture of the tensor

For the subtraction we are using 2‘s complement representation. The main advantage of this code is that that the subtraction can be easily done using addition. If the number is negative, we convert it by inverting all bits and adding one. Because of this we need to use 8 incrementers (one before subtraction to convert the number into negative before performing subtraction and one to make the absolute value of the number if the result of the subtraction is a negative number). When using approximate arithmetic we are excluding these incrementers, thus we are introducing more errors into our design, but we are also making the overall area smaller.

The adders used for subtraction and addition are synthesized by Vivado tool when using full precision and with imprecise adder we are using full adders for higher bits and OR gates in lower imprecise bits. The architecture of the Lower OR Adder is shown in Figure 1.

5.2 Test Results

We used a collection of 40 different ghosts (categories) and we were identifying 40 different samples. When using full precision we were capable of successful identification of 35 out of 40 faces. When using imprecise arithmetic we were capable to successfully identify 37 out of 40 faces.

Because we were testing our solution on an FPGA we could not achieve expected savings and results from synthesis are not considered as relevant for our paper.

6 Conclusion

In our paper we tested the possibilities of approximate arithmetic in face recognition using Eigenfaces. Theoretical simulations show that we can reduce the overall area of the arithmetic units by using the approximate arithmetic while still having high success rate.

One of the main problems of our solution is that the approximate arithmetic is static. Instead of static approximate adders we could use configurable adders such as the one proposed in [5]. Also relatively high delay of the carry-ripple adder used in the implementation could be reduced by using more effective adders, like the prefix adder. On the other hand this would lead to increase of the area of whole adder.

References

[1] Chu, Pong P. RTL hardware design using VHDL: coding for efficiency, portability, and scalability. John Wiley & Sons, 2006, Chapter 12.3.1

[2] A VHDL UART for communicating over a serial link with an FPGA, http://www.cl.cam.ac.uk/research/dtg/attarchive/facedatabase.html

[3] Mahdiani, Hamid Reza, et al. "Bio-inspired imprecise computational blocks for efficient VLSI implementation of soft-computing applications." IEEE Transactions on Circuits and Systems I: Regular Papers 57.4 (2010): 850-862.

[4] Gupta, Vaibhav, et al. "IMPACT: imprecise adders for low-power approximate computing." Proceedings of the 17th IEEE/ACM international symposium on Low-power electronics and design. IEEE Press, 2011..

[5] Kahng, Andrew B., and Seokhyeong Kang. "Accuracy-configurable adder for approximate arithmetic designs." Proceedings of the 49th Annual Design Automation Conference. ACM, 2012.

[6] Vijayakumari, V. "Face recognition techniques: A survey." World journal of computer application and technology 1.2 (2013): 41-50.

[7] Turk, Matthew A., and Alex P. Pentland. "Face recognition using eigenfaces." Computer Vision and Pattern Recognition, 1991. Proceedings CVPR'91., IEEE Computer Society Conference on. IEEE, 1991.

[8] Mittal, Sparsh. "A survey of techniques for approximate computing." ACM Computing Surveys (CSUR) 48.4 (2016): 62. APA

[9] Cheng, Kwang-Ting, and Yi-Chu Wang. "Using mobile GPU for general-purpose computing–a case study of face recognition on smartphones." VLSI Design, Automation and Test (VLSI-DAT), 2011 International Symposium on. IEEE, 2011. APA

[10] Turk, Matthew, and Alex Pentland. "Eigenfaces for recognition." Journal of cognitive neuroscience 3.1 (1991): 71-86.

[11] Mahdiani, Hamid Reza, et al. "Bio-inspired imprecise computational blocks for efficient VLSI implementation of soft-computing applications." IEEE Transactions on Circuits and Systems I: Regular Papers 57.4 (2010): 850-862.

[12] The database of faces, http://www.cl.cam.ac.uk/research/dtg/attarchive/facedatabase.html,

[13] Nexys 4 DDR reference, https://reference.digilentinc.com/reference/programmable-logic/nexys-4-ddr/start

Appendices

A Introduction

In the next sections we are describing the technical details of the implementation of the hardware accelerator.

In our accelerator we are implementing scoring part of the Eigenfaces algorithm [7]. The training is performed on a PC.

In this document we are describing top architecture of the design synthesized in the FPGA (Appendix B) communication between PC and FPGA and a controller of the accelerator (Appendix C), memory (Appendix D) and the accelerator (Appendix E).

B Top Module of the Hardware Accelerator

The hardware accelerator consists of these main parts:

- PC - PC is the computer to which the accelerator is connected
- FPGA
- UART - interface between PC and FPGA
- Controller - controller of the accelerator
- Memory
- Accelerator

The Figure 6 shows the architecture of the top module of the hardware accelerator

C Controller and Communication Between FPGA and PC

For the communication between PC and FPGA we are using UART provided by [2]. We are using baud rate of 115200 and COM4 port. On the PC we are running python scripts that send the sample and ghost to the FPGA. The pictures must be PGM P2 files and with the size of 92x112.

C.1 Python Script Functions

The paragraphs below are describing the Python functions that are responsible for the communication between PC and FPGA

setUpCom Function setUpCom is responsible for opening a USB port for the serial communication.

closeCom Function closes the communication at the USB port

send_ghost Sends the ghost to the FPGA. Ghost must be first read as byte stream from the PGM file.

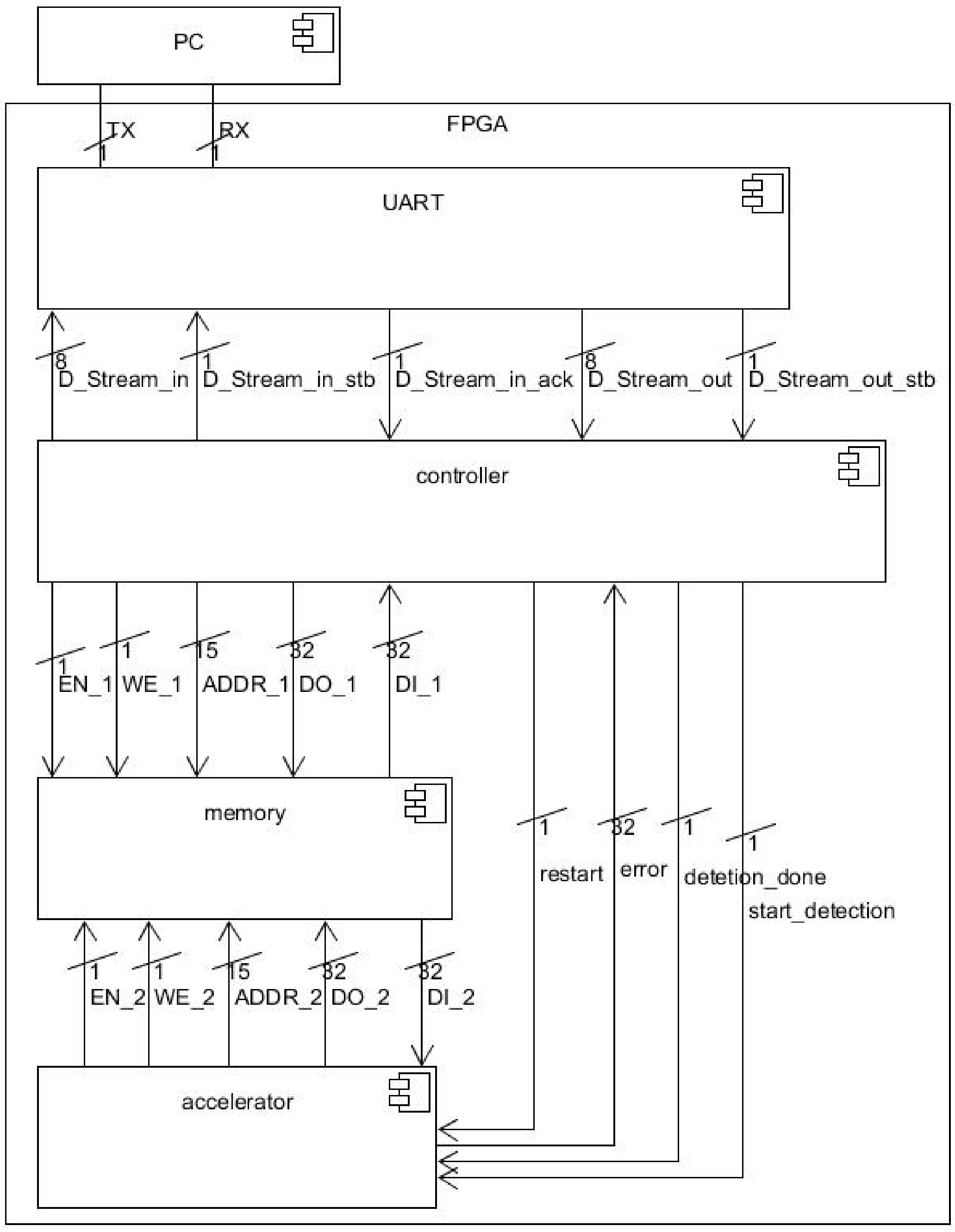


Figure 6: Architecture of the top part of the hardware accelerator

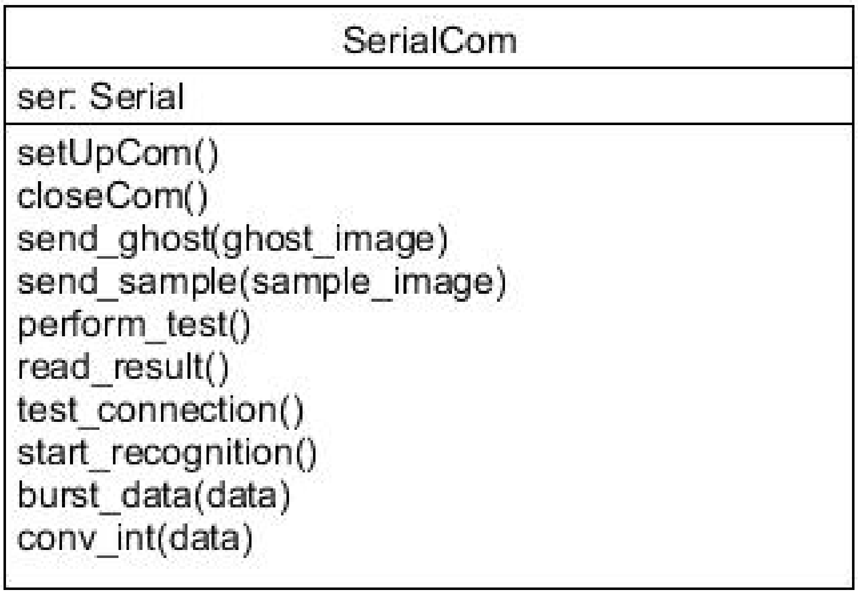


Figure 7: Class diagram of the SerialCom class used for communication between the FPGA and PC

send_sample Sends the sample to the FPGA. Sample must be first read as byte stream from the PGM file.

perform_test Serves for the debugging purposes and returns last four bytes of the sample.

test_connection Sends a test byte to the FPGA. In case of succes prints message "Success!!!" to the command line. In case of "Failure!!!" to the command line.

read_result used only for the debugging purposes. Reads the result wrote in the result register.

start_recognition Sends a command to the FPGA to start the recognition. Returns byte array in the big endian format (data[0] is LSB and data[3] is the MSB.

burst_data Sends the byte stream to the FPGA. Used to send the data in send_ghost and send_sample functions.

The Figure 7 shows the class diagram of the SerialCom class, which encapsulates functions responsible for communication between FPGA and the PC. The Figure 8 shows the flow of the face recognition. First we need to load bytes from a PGM file (not provided by the SerialCom class functions), then we store the file in the FPGA, next we load the sample in a similar way and store it into the FPGA. Then we run the function start_recognition which returns the absolute error of the scoring algorithm.

C.2 Accelerator Controller

To control the communication between the FPGA and PC we are using module called memory_controller. It is connected to the memory, accelerator and UART as can be seen in Figure

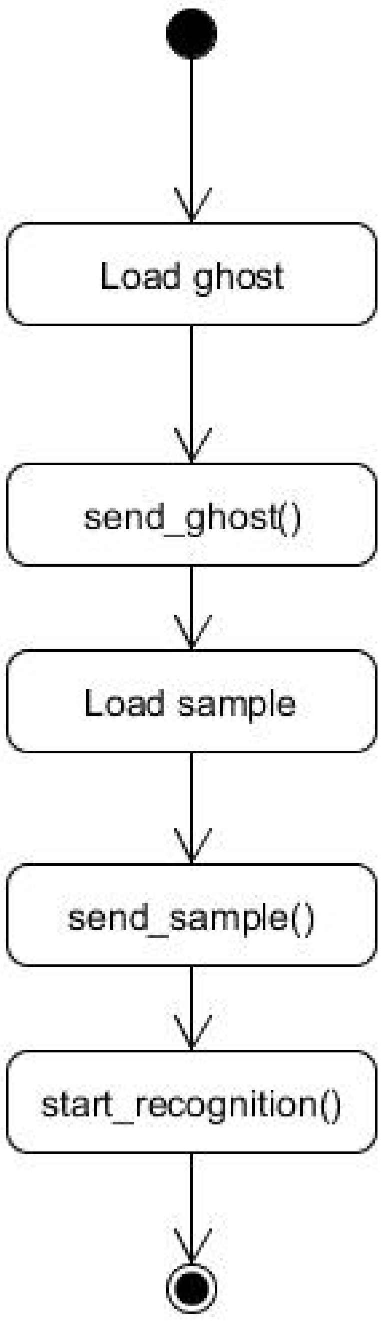


Figure 8: Acitivity diagram of the face recognition

6. To control the FPGA we are using several control codes which are in a form of a byte. The control codes are showed in the list below:

- s - hexadecimal value of 73 - when sent, FPGA expects the PGM file of size 92x112 which is in form of a byte array with size of 10304. Stores the picture as a sample(in the memory from address 8192).
- g - hexadecimal value of 67, when sent, FPGA expects the PGM file of size 92x112 which is in form of a byte array with size of 10304. Stores the picture as the ghost (in the memory from address 0).
- c - hexadecimal value of 63, when sent, FPGA returns last four bytes of the sample (address 10767).
- t - hexadecimal value of 74, when sent, FPGA returns test character, which has the same value (74h).
- e - hexadecimal value of 65, when sent, FPGA returns error stored in the error register, only used for debugging purposes.
- a - hexadecimal value of 61, when sent, FPGA starts the face recognition and returns the absolute error.

Because the state machine of the memory controller would be too complicated we are only showing state CMD, which is responsible for handling the commands. The Figure 9 shows the part of the ASM chart with CMD state.

D Memory

The memory is a simple two port RAM similar to the RAM in [1]. The protocol for communication is shown in the diagram in the Figure 10

When reading from the memory we need to send a valid address and activate signal EN. In the next cycle memory D_OUT signal contains valid data from the memory. When writing to a memory we have to activate signal WE, EN, send the valid address and send the valid data into input D_IN.

E Accelerator

The accelerator module is responsible for performing scoring algorithm of the eigenfaces. It reads four bytes (1 address in the memory) from the ghost picture and from the sample. The datapath consists of two adders. The first adder is a 9 bit adder which does a subtraction. For this we are using 2‘s complement. In the precise architecture we have to invert the number and add 1 to it before subtraction and in order to create absolute value of the subtraction we have to again invert the number and add 1. In the imprecise version we are omitting this addition, thus introducing an error on one hand, but reducing the total area and complexity of the circuit on the other hand. The second adder is 32 bit adder for precise architecture and 21 bit adder for the imprecise architecture.

The Equation 16 shows the computation of the error for each byte. The result is always stored in a matrix of four bytes - four errors. These errors are in the end summed and sent from the accelerator as an absolute error. Because we are using image of the same size, the absolute error does not need to be divided.

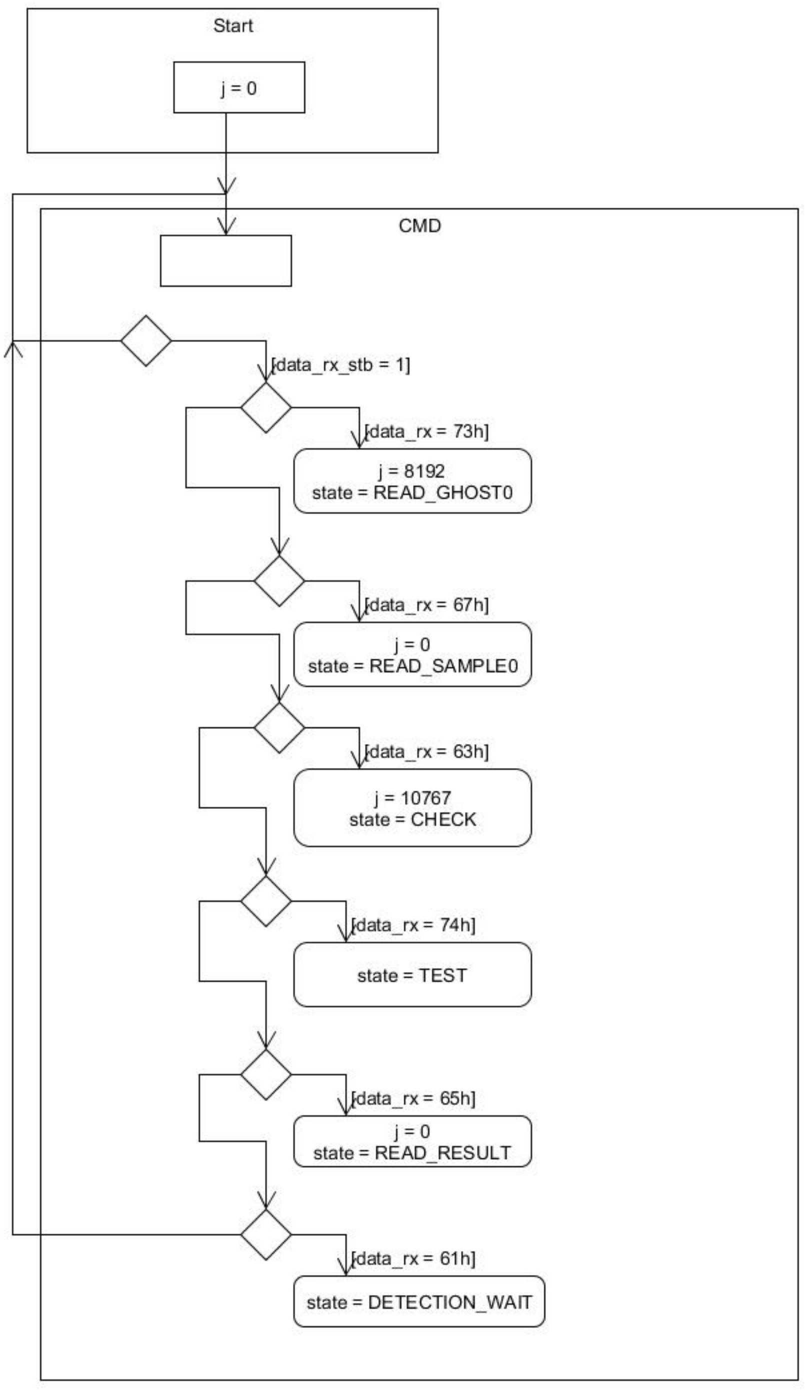


Figure 9: ASM chart with the CMD state

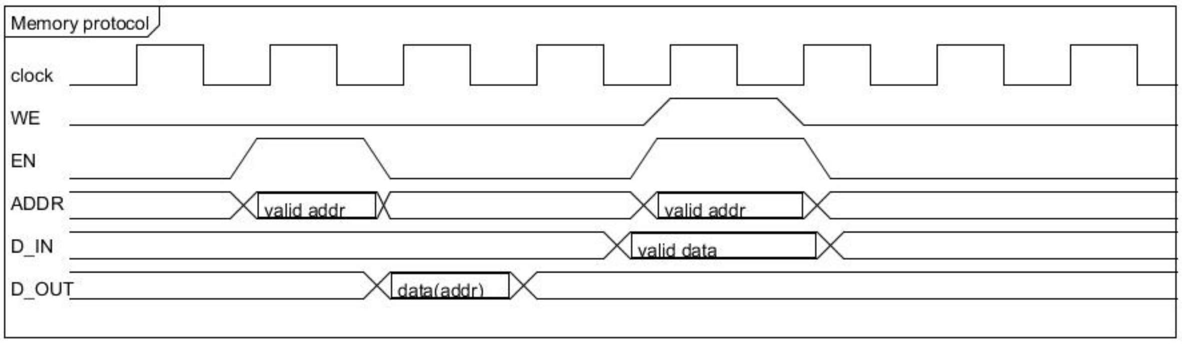


Figure 10: Protocol of the memory communication

$$err_i = err_i + |ghost_i - sample_i| \tag{16}$$

The ASMD chart in the Figures 11 and 12 shows the flow of the accelerator. Because of the complexity of the chart, it is split into two halves.

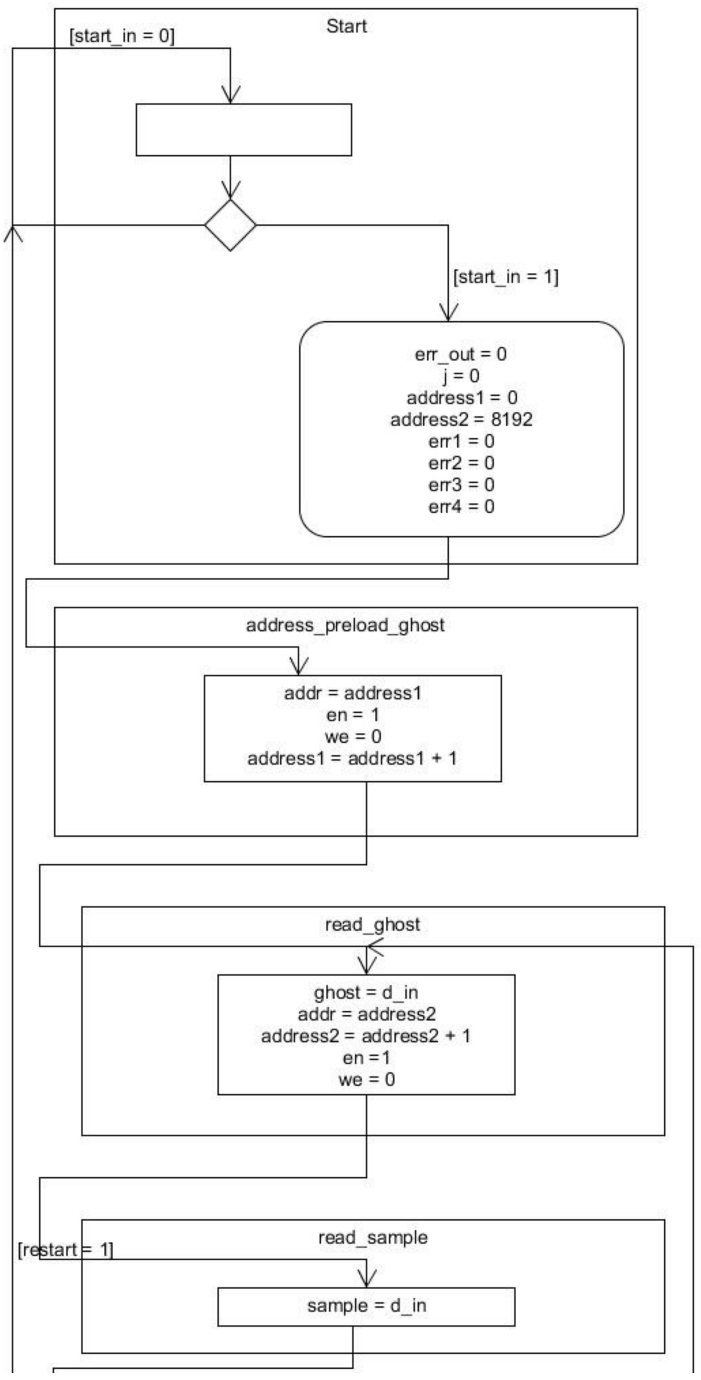


Figure 11: ASMD chart of the accelerator part 1

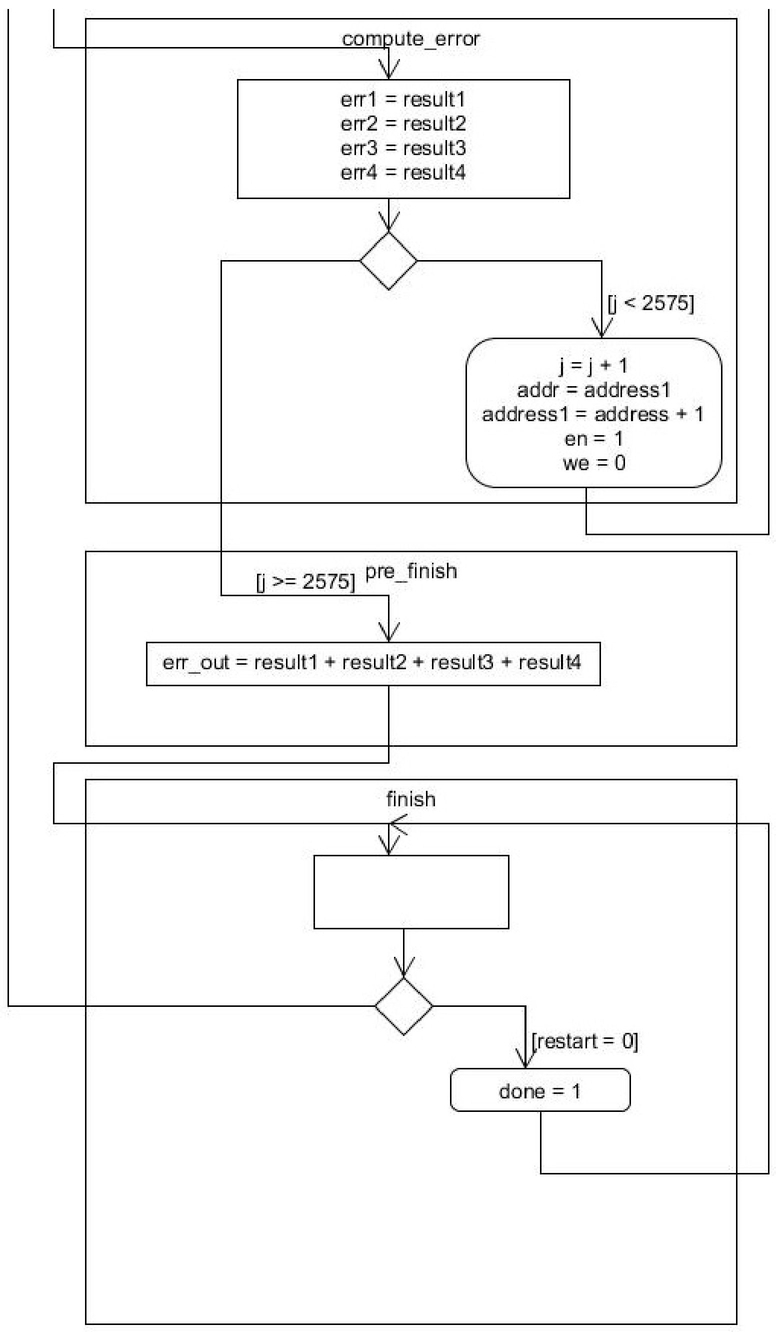


Figure 12: ASMD chart of the accelerator part 2

The architecture of the accelerator consists of the FSM and of the datapath which is similar to tensors. It is shown in the Figures 13 and 14

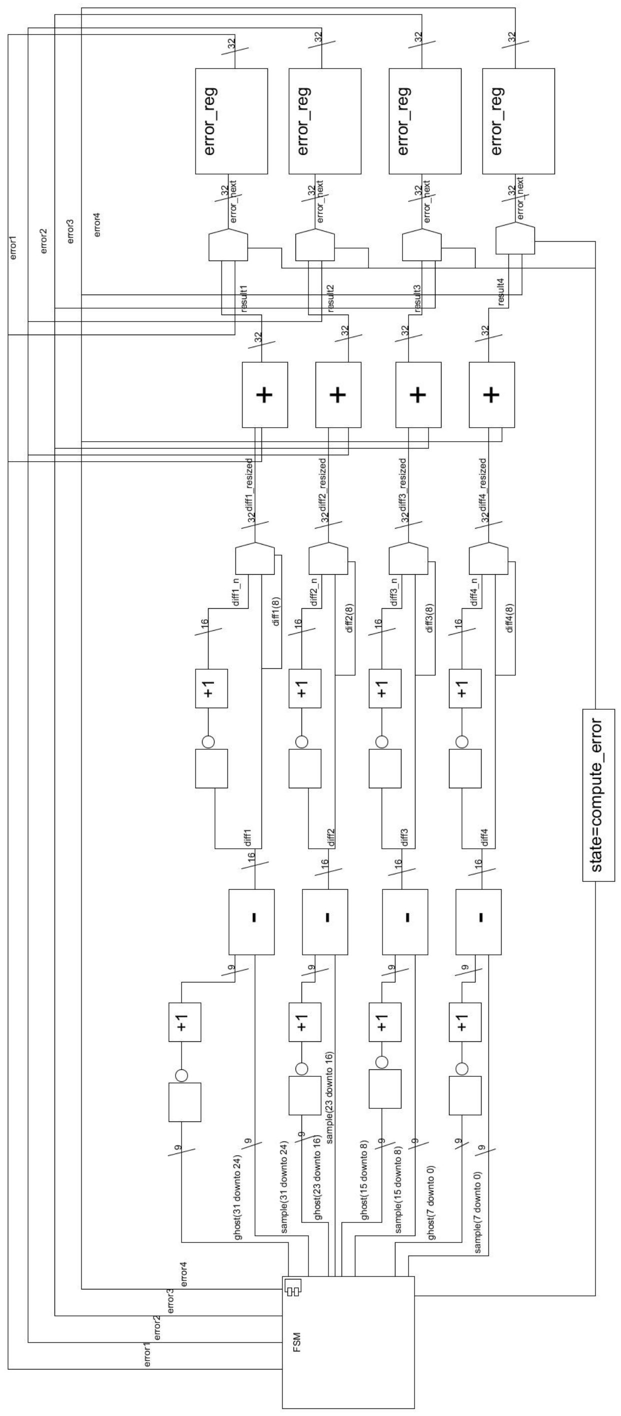


Figure 13: Datapath and FSM with precise arithmetic

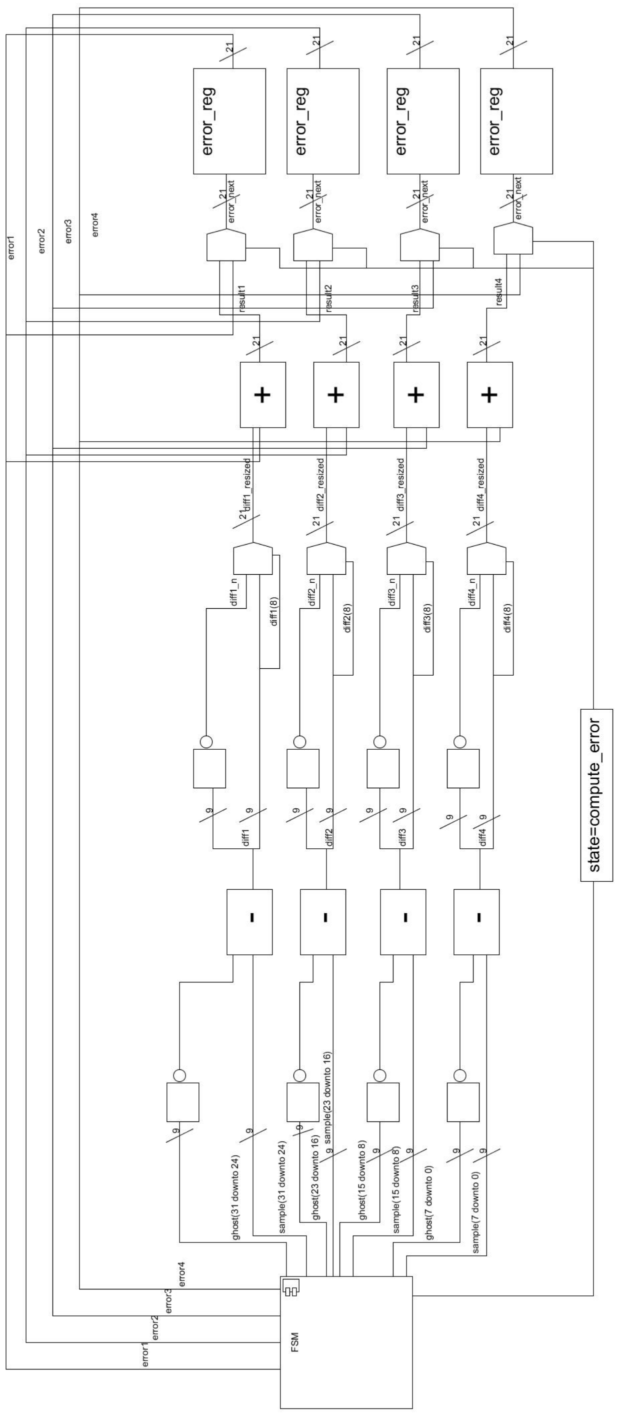


Figure 14: Datapath and FSM with imprecise arithmetic

E.1 Lower OR Adder

In our accelerator we are using Lower Or Adder as proposed in [11]. The idea is to use the precise adder for the higher order bits and simple or gates for the lower bits. For the higher bits we are using a full adder however, more efficient adder (e.g. prefix adder) can also be used. The Figure 15 shows the general architecture of the Lower OR Adder.

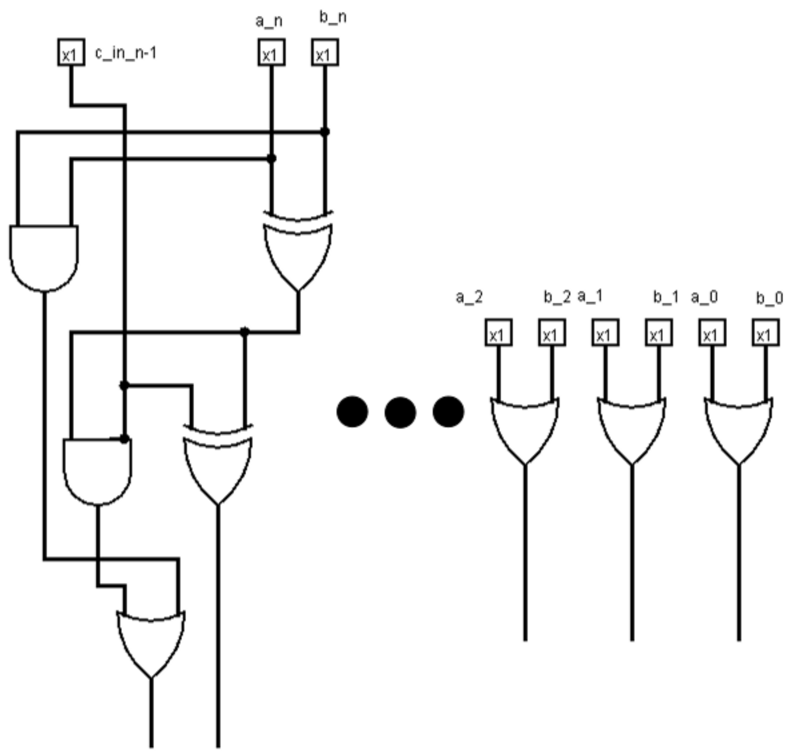


Figure 15: Lower Or adder