Efficient Hardware Algorithms for the FPGA

(FPGA 向けの効率的なハードウエアアルゴリズム)

by

Xin Zhou

A dissertation submitted
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy
in Information Engineering

Under Supervision of
Professor Koji Nakano

Department of Information Engineering,
Graduate School of Engineering,
Hiroshima University

September, 2016

Summary

Field-Programmable Gate Array (FPGA) is a programmable silicon device designed to be configured by the customer using hardware description language after manufacturing. In the past, FPGAs are used for lower speed and complexity designs due to the lack of internal logic resources and low frequency of the FPGA. Today's FPGAs can easily run at high frequency and have unprecedented logic density. Furthermore, embedded processors, DSP slices, block RAMs are embedded in the FPGA. Also, ability of parallel processing is one of the most important features that separate FPGA from the conventional microprocessor.

In order to improve the processing speed, multicore processors are widely used in many application domains such as general purpose computation, digital signal processing, and image processing. Embedded multicore processors represented by FPGA has lately attracted considerable attention for their potential computation ability and power consumption. By partitioning the algorithm into servel independent parts, multicore processors can perform all parts concurrently. If the algorithm is hard to be parallelized, we can also improve the processing speed considerably by employing multicore processor to perform the same algorithm for different data sets.

Hough transform is a technique to find shapes in images such as lines, circles, ellipses, etc. In this dissertation, we have presented implementations of the Hough transform on the FPGA for extracting lines and circles. The Hough transform defines a mapping from an image into a parameter space represented by an accumulate array. For each edge point of the image, the mapping adds a vote to corresponding elements in the accumulate array. Therefore, the elements that are voted intensively represent associated parameters of detected shapes. The first contribution of this dissertation is to

present an efficient implementation of the Hough transform on the FPGA. In our implementation, we partition the parameter space and the voting operation is performed in parallel by an efficient usage of DSP slices and block RAMs. As far as we know, there is no previously published work that fully utilizes DSP slices and block RAMs for the Hough transform. The experimental results show that our FPGA implementation attains a speed-up factor of more than 300 over the sequential implementation on the CPU by using 178 DSP slices and 180 block RAMs. However, this implementation needs to accept the coordinates of edge points as input. Also, since identified lines are obtained just by thresholding after voting, incorrect lines are also detected. Hence, the second contribution of this dissertation is to present an improved FPGA implementation of the Hough transform. The improved FPGA implementation processes all pixel data given in raster scan order, and the usage of DSP slices reduces. Also, maximum filters are used to obtain the correct lines after voting operation. The improved implementation uses only 90 DSP slices and 181 block RAMs and attains a speed-up factor of more than 38 over the sequential implementation on the CPU. Next, gradient-based Hough transform is one of the efficient improvements to the Hough transform for line detection, where the gradient direction and magnitude of each pixel are used to reduce the number of useless votes for obtaining more precise lines. The third contribution of this dissertation is to present an efficient implementation of the gradient-based Hough transform on the FPGA. This implementation uses only 13 DSP slices and runs 309 times faster over the sequential implementation on the CPU. Furthermore, comparing with other FPGA implementations, the performance of our FPGA implementations is better. On the other hand, the Hough transform can be also used to extract circles. The fourth contribution of this dissertation is to present an efficient implementation of the Hough transform for

circles detection on the FPGA, that uses only one-dimensional parameter spaces. Our implementation uses 398 DSP slices and 309 block RAMs and runs in 181.812MHz. According to the experimental results, our implementation attains a speed-up factor of approximately 189 over the sequential implementation on the CPU.

FPGA is also desired hardware device for general purpose computation. The Greatest Common Divisor (GCD) computation is widely used in computer systems for cryptography, data security and other important algorithms. Most of the time of these computer systems is consumed for computing the GCDs of very large integers. In this dissertation, the fifth contribution is to propose an efficient processor core that executes the Euclidean algorithm computing the GCD of two large numbers in an FPGA by using only one DSP slice and one block RAM. Since the proposed processor core is compactly designed and uses very few resources, we have succeeded in implementing more than one thousand processor cores in an FPGA. The experimental results have shown that our implementation of 1280 GCD processor cores runs 3.8 times faster than the best GPU implementation and 316 times faster than a sequential implementation on the CPU.

Data compression is one of the most important task in the area of computer engineering. LZW algorithm is one of the most famous dictionary-based compression and decompression algorithms. Since dictionary tables are created by reading input data one by one, LZW compression and decompression are hard to parallelize. The sixth contribution of this dissertation is to present a hardware architecture of LZW compression and decompression, respectively. Since the proposed modules of LZW compression and decompression use very few FPGA resources, we have succeeded in implementing 24 modules of LZW compression and 34 modules of LZW decompression in an FPGA, respectively. The experimental results show that, our implementation of 24 LZW

compression modules attains a speed-up factor of 23.51 times faster than a sequential implementation on a single CPU, while our implementation of 34 LZW decompression modules attains a speed-up factor of 64.39 times faster than a sequential implementation on the CPU.

Contents

List of Figures

Chapter 1

Introduction

1.1 Background and Motivation

Recently, the improvements in speed of the microprocessor is slowing down since the heat generation and size constraints of transistor become significant problems. On the other hand, the Field-Programmable Gate Array has been widely used in various fields for the high performance, ability of parallel processing, programmable features and low price of it.

The FPGA is an integrated circuit designed to be configured by a designer after manufacturing, that differs from Application Specific Integrated Circuits (ASCIs) which are designed for specific applications. It contains an array of programmable logic blocks called CLB (Configurable Logic Block), and the reconfigurable interconnects allow the blocks to be inter-wired in different configurations. In the past, FPGAs are used for lower speed and complexity designs due to the lack of internal logic resources and low frequency of the FPGA. Recent FPGAs can easily run at high frequency and have unprecedented logic density. Furthermore, embedded processors, DSP slices, block

RAMs are embedded in the FPGA that are make a higher performance and a broader application. Since any logic circuits can be embedded in an FPGA, it can be used for parallel computing which is one of the most important features that separate FPGA from the conventional microprocessor.

In order to improve the processing speed, multicore processors are widely used in many application domains such as general purpose computation, digital signal processing, and image processing. Especially Embedded multicore processors represented by FPGA has lately attracted considerable attention for their potential computation ability and power consumption. By partitioning the algorithm into several independent parts, multicore processors can perform all parts concurrently. If the algorithm is hard to be parallelized, we can also improve the processing speed considerably by employing multicore processor to perform the same algorithm for different data sets.

1.2 Contributions

1.2.1 Implementations of the Hough transform algorithm on the FPGA

Hough transform is a technique to find shapes in images. In particular, it has been utilized to extract lines, circles, ellipses and arbitrary shapes. In this dissertation, we have presented implementations of the Hough transform on the FPGA for extracting lines and circles. The Hough transform defines a mapping from an image into a parameter space represented by an accumulate array. For each edge point of the image, the mapping adds a vote to corresponding elements in the accumulate array. Therefore, the elements that are voted intensively represent associated parameters of detected shapes.

In the implementation of the Hough transform algorithm for extracting lines, we partition the parameter space and the voting operation is performed in parallel by using the DSP slices and block RAMs of the FPGA. The same architecture can also be easily implemented in other hardware device to obtain high performance of the Hough transform. First, for the voting operation of the Hough transform, our FPGA implementation attains a speed-up factor of more than 300 over the sequential implementation on the CPU by using 178 DSP slices and 180 block RAMs. However, the implementation needs to accept the coordinates of edge points as input. Also, since identified lines are obtained just by thresholding after voting, similar to lines in the input image but incorrect lines are also detected. Then, we improve the implementation to process pixel data given in raster scan order, and the number of used DSP slices becomes approximately half. Also, 3×3 maximum filters are used to obtain more precise lines after voting operation. The experimental result show that this implementation uses only 90 DSP slices and 181 block RAMs and attains a speed-up factor of more than 38 over the sequential implementation on the CPU. Next, as one of the efficient improvements to the Hough transform for line detection, we present an efficient architecture of gradient-based Hough transform, where the gradient direction and magnitude of each pixel are used to simplify the voting operation and reduce the usage of the FPGA resources. This implementation uses only 13 DSP slices and runs 309 times faster over the sequential implementation on the CPU. Furthermore, comparing with other FPGA implementations, the performance of our FPGA implementations is better. On the other hand, we present an efficient implementation of the Hough transform algorithm that uses only one-dimensional parameter spaces for circles detection on the FPGA. Our implementation uses 398 DSP slices and 309 block RAMs and runs in 181.812MHz. According to

the experimental results, our implementation attains a speed-up factor of approximately 189 over the sequential implementation on the CPU.

1.2.2 Implementation of the Euclidean algorithm on the FPGA

FPGA is also desired hardware device for general purpose computation. The Greatest Common Divisor (GCD) computation is widely used in computer systems for cryptography, data security and other important algorithms. Most of the time of these computer systems is consumed for computing the GCDs of very large integers. In this dissertation, we have proposed an efficient processor core that executes the Euclidean algorithm computing the GCD of two large numbers in an FPGA by using only one DSP slice and one block RAM. Since the proposed processor core is compactly designed and uses very few resources, we have succeeded in implementing more than one thousand processor cores in an FPGA. The experimental results have shown that our implementation of 1280 GCD processor cores runs 3.8 times faster than the best GPU implementation and 316 times faster than a sequential implementation on the CPU.

1.2.3 Implementations of the LZW compression and decompression algorithms on the FPGA

Data compression is one of the most important task in the area of computer engineering. It is always used to improve the efficiency of data transmission and save the storage of data. Data compression includes two basic methods, lossy compression and lossless compression. LZW algorithm is one of the most famous dictionary-based lossless compression and decompression algorithms. Since dictionary tables are created by reading input data one by one, LZW compression and decompression are hard to parallelize. In

this dissertation, we present a hardware architecture of LZW compression and decompression, respectively. Since the proposed modules of LZW compression and decompression use very few FPGA resources, we have succeeded in implementing 24 modules of LZW compression and 34 modules of LZW compression in an FPGA, respectively. The experimental results show that, our implementation of LZW compression attains a speed-up factor of 23.51 times faster than a sequential implementation on a single CPU, while our implementation of LZW decompression attains a speed-up factor of 64.39 times faster than a sequential implementation on the CPU.

1.3 Dissertation Organization

The doctoral dissertation is organized as follows. In Chapter 2, we show the details of the FPGA and the embedded resources of it. We show efficient implementations of the Hough transform for extracting lines and circles in Chapter 3. In Chapter 4, we propose a hardware binary Euclidean algorithm for computing GCD of two very large numbers and implement it on the FPGA. Chapter 5 presents an efficient implementation of LZW compression and decompression algorithms, respectively. Finally, Chapter 6 concludes this dissertation.

Chapter 2

FPGA

We show the architecture and the embedded resources of the FPGA in this chapter. Since Xilinx Virtex-6 and Virtex-7 FPGAs are used to evaluate the performance of our implementations in this dissertation, we also show the main resources of them such as the embedded DSP48E1 slices and block RAMs which are used in our implementations.

2.1 Architecture of FPGA

Field-Programmable Gate Arrays (FPGAs) are programmable semiconductor devices that contains an array of Configurable Logic Blocks (CLBs) [54, 59] that can be inter-wired by reconfigurable interconnects. Slice Registers and Slice LUTs (Look-Up-Tables) are the main hardware resources in CLB, that are used to implement sequential and combinatorial logics. Recent FPGA architecture consists of an array of CLBs, I/O pads, DSP slices [51, 58], block RAMs [52, 60], and routing channels as shown in Figure 2.1, where the embedded block RAMs and DSP slices make a higher performance and a broader application. Since most of recent FPGAs produced by principal vendors equip

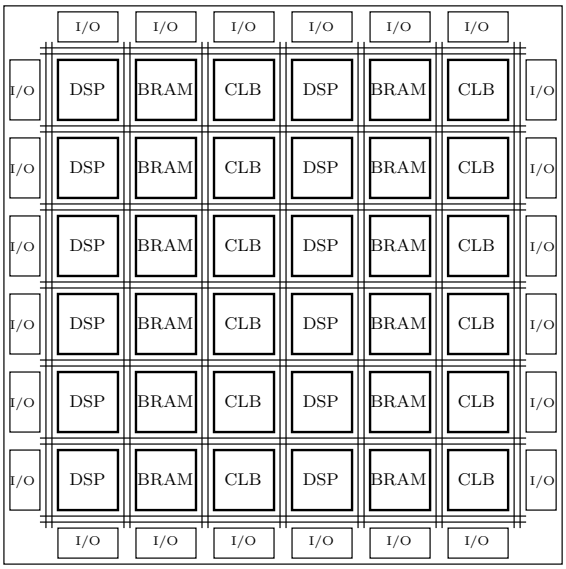


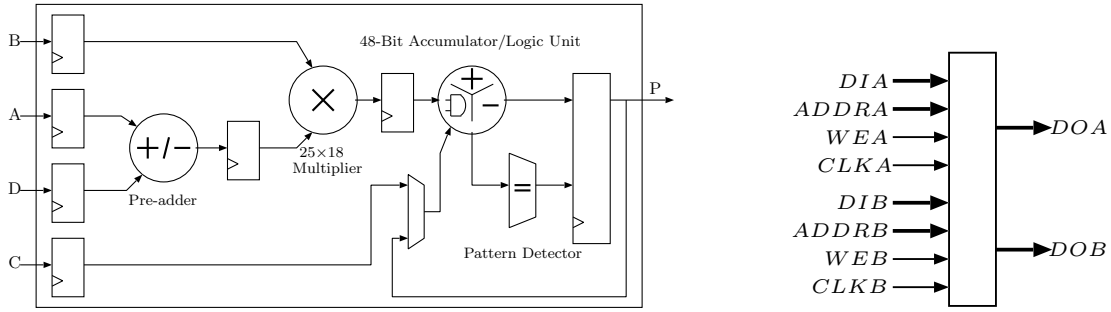
Figure 2.1: The architecture of FPGA

embedded DSP slices and block RAMs, one of the most important key techniques for accelerating computation using FPGAs is an efficient usage of DSP slices and block RAMs. In this dissertation, all of our FPGA implementations are proposed by using the DSP slices and block RAMs efficiently to obtain high performance. Hence, we show the details of the DSP slice and block RAM in the following.

2.2 DSP48E1 slice

DSP48E1 slices are the embedded DSP slices of Virtex-6 and Virtex-7 family FPGAs. The basic architecture of DSP48E1 slice is illustrated in Figure 2.2(a). DSP48E1 slices are equipped with a 25-bit pre-adder, a 25-bit by 18-bit two's complement multiplier, 48-bit multiplexers, an optional logic unit, a pattern detector, etc. We show some details of the embedded resource of DSP48E1 slice as follows.

Pre-adder: Port A and D feed pre-adder. By controlling the behavior of the pre-adder,



(a) The DSP48E1 slice

(b) The dual-port block RAM

Figure 2.2: The DSP slice and block RAM in Xilinx FPGAs

it can dynamically compute the values of D-A, A, D, D+A, etc. The A and D data inputs can optionally be registered one or two times to highly pipelined architecture for different applications.

Multiplier: The embedded multiplier of DSP slice has two input ports. The output of the pre-adder and Port B feed to the multiplier, where the B data input and the output of the pre-adder can optionally be registered up to two times and one time, respectively.

ALU: The Arithmetic Logic Unit (ALU) can be configured as three-input adder/subtractor or two-input logic unit. The output of the multiplier and port C are connected to inputs of the ALU. By controlling the behavior of the ALU, we can dynamically perform different addition/subtraction computations and logic operations between the inputs of the ALU. The obtained result of the ALU is then connected to the register P.

Pattern detector: The pattern detector at the output of the DSP48E1 slice provides support for convergent rounding, overflow/underflow, block floating point, and support for accumulator terminal count. More specifically, the pattern detector can detect if the output of the DSP48E1 slice matches a pattern as qualified by a predefined mask.

The DSP slice includes dedicated buses for cascading. Hence, the DSP slice also supports cascading multiple DSP48E1 slices for applications requiring wide math func-

tions and complex arithmetic without the use of general FPGA logic resources. By employing the functionality of the DSP slice, we can implement complex applications on the FPGA with better performance and reduce the usage of FPGA resources, comparing with the implementation using general FPGA logic resources.

2.3 Block RAM

The block RAM is an embedded dual-port memory supporting synchronized read and write operations as illustrated in Figure 2.2(b). The block RAM can be configured as 36kbit dual-port block RAMs, FIFOs, or two 18kbit dual port RAMs. The dual port block RAMs have two sets of ports operated independently. Two sets of ports are:

Port Set A: *ADDRA* (ADDRess A), *DOA* (Data Output A), *DIA* (Data Input A)

Port Set B: *ADDRB* (ADDRess B), *DOB* (Data Output B), *DIB* (Data Input B)

In read operation of Port Set A, the element in address *ADDRA* is output from *DOA* after the rising clock edge. In write operation of Port Set A, the data given to *DIA* is written to the element in address *ADDRA* of the block RAM at the rising clock edge. Read/write operations of Port Set B are the same as Port Set A. Port Set A and Port Set B work independently. In the block RAMs in the target devices of this dissertation, read/write operations can be configured as either RF (Read First) mode or WF (Write First) mode. In the RF mode, if reading and writing operations are performed to the same address, reading operation is performed before the writing operation. Hence the reading data is the data before writing data. On the other hand, in the WF mode, since the writing performed before the reading, the reading data is the updated data. However, when a dual port is used, there is a restriction that if read and write operation to the same address are performed for each port, the setting of block RAMs must be RF [52, 60].

Chapter 3

Implementations of the Hough transform algorithm on the FPGA

In this chapter, we show efficient implementations of the Hough transform for extracting lines and circles, respectively. In the implementation of the Hough transform algorithm for extracting lines, the parameter space is partitioned, and the voting operation is performed in parallel by using the DSP slices and block RAMs of the FPGA. First, we present an architecture for the voting operation of the Hough transform algorithm and implement it on the FPGA. Our FPGA implementation uses 178 DSP slices and 180 block RAMs and attains runs over 300 faster than the sequential implementation on the CPU. However, the implementation needs to accept the coordinates of edge points as input. Also, since identified lines are obtained just by thresholding after voting, similar to lines in the input image but incorrect lines are also detected. Then, we improve the implementation to process pixel data given in raster scan order, and the number of used DSP slices reduces approximately half. The revised implementation uses only 90 DSP slices and 181 block RAMs, and runs over 38 times faster than the sequential

implementation on the CPU.

Next, as one of the efficient improvements to the Hough transform for line detection, we present an efficient architecture of gradient-based Hough transform, where the gradient direction and magnitude of each pixel are used to reduce the useless votes for obtaining more precise straight lines. The experimental results show that this implementation use only 13 DSP slices and runs 309 times faster over the sequential implementation on the CPU.

On the other hand, we present an efficient implementation of the Hough transform algorithm that uses only one-dimensional parameter spaces for circles detection on the FPGA. Our implementation runs 189 times faster than the sequential implementation on the CPU.

3.1 Efficient implementations of the Hough transform algorithm for extracting lines

Hough transform is a technique to find shapes in images [20] such as lines, circles, ellipses. The Hough transform defines a mapping from an image into a parameter space represented by an accumulate array. The parameter space is defined by parameterizing detected shapes. Based on each edge point of the image, the mapping adds a vote to corresponding elements in the accumulate array. The elements that are increased represent associated parameters based on detected shapes. Therefore, the elements that are voted intensively correspond to the parameters of shapes in the image space. In this section, we show two implementations of the Hough transform using DSP slices and block RAMs on the FPGA.

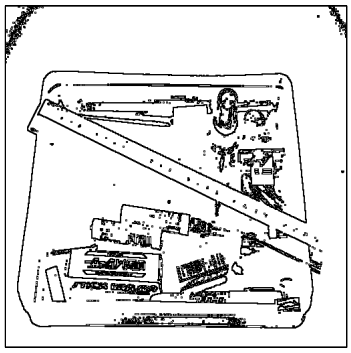
3.1.1 Introduction

The Hough transform can be used to extract straight lines in a binary image [11]. The idea of this method is to exploit the duality between points of a line and parameters of that line. A point in the image is represented by a curve in the parameter space and lines of collinear points intersect in the parameter space at one point. These intersections are counted in an array of accumulators that quantizes the parameter space appropriately. In the followings, we call this counting to the accumulators *voting*. More specifically, for each edge point (x, y) in a 2-dimensional image, the voting is performed along a curve $\rho = x\cos\theta + y\sin\theta$ $(0 \leq \theta < 180)$. Possible lines can be detected by searching points that are voted intensively. Figure 3.1 shows an example of straight line detection using the Hough transform. For an input image (Figure 3.1(a)), the binary edge image (Figure 3.1(b)) is obtained by the edge detector such as Sobel filter. The result of voting to the parameter space is shown in Figure 3.1(d). In this figure, darker points show points that are voted intensively, that is, represent probable lines. According to the result of voting, the principal lines are detected (Figure 3.1(c)).

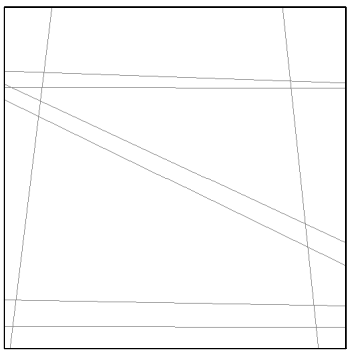
The first contribution of this chapter is to present an implementation of the Hough transform on the FPGA. The first idea of the implementation is an efficient usage of DSP slices and block RAMs for FPGAs. The second idea is to partition the voting space in the Hough transform and the voting operation is performed in parallel. We describe the ideas of our FPGA implementation as follows.

Voting Space Partitioning: Polar coordinate voting space (θ, p) is partitioned and arranged into block RAMs. This enables us to perform voting operations in parallel. Also, the function of dual-port of block RAMs are fully used to accumulate the voting value instantly.

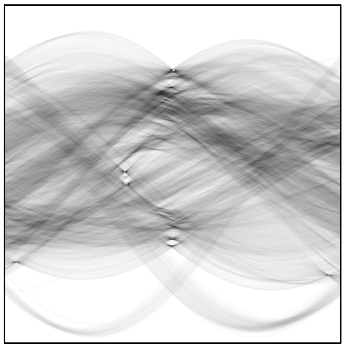
(a) Input image



(b) Binary edge image by Sobel filter



(c) Line detection using the Hough transform



(d) Hough parameter space

Figure 3.1: Example of straight line detection using the Hough transform

Efficient Usage of DSP slices: DSP slices are used to compute $x\cos\theta$ and $y\sin\theta$ in parallel for each edge pixel (x, y). We compute $x\cos\theta$ and $y\sin\theta$ for θ such that $0 \leq \theta < 90$ instead of computing them for θ such that $0 \leq \theta < 180$. Also, we avoid the computation of the values of $\cos\theta$ and $\sin\theta$ by pre-loading them in the DSP slices.

Fully Pipelined Architecture: We take into account a layout of DSP slices and block RAMs in Virtex-6 FPGA architecture, and design our Hough transform architecture as a fully pipelined one. For example, in the Virtex-6 FPGA XC6VLX240T has 768 DSP48E1 slices arranged in 8 columns of 96 adjacent DSP48E1 slices. Neighboring DSP48E1 slices are connected directly through pipeline registers. Our Hough transform

architecture uses 2 columns to compute $x\cos\theta$ and $y\sin\theta$ each, and uses a pipeline technique to maximize the clock frequency.

Using these ideas, our architecture for the Hough transform uses 178 DSP48E1 slices and 180 block RAMs with 18kbits that work in parallel. As far as we know, there is no previously published work that full utilizes DSP slices and block RAMs for the Hough transform. Roughly speaking, a conventional sequential implementation performs $180m$ voting operations for m edge points. Our architecture performs voting operations in parallel, and outputs identified lines in $m + 97$ clock cycles. Since $180m$ voting operations are performed using 178 DSP48E1 slices, the lower bound of the computing time is m clock cycles. Hence, our implementation is close to optimal. We have implemented our architecture on a Virtex-6 family FPGA XC6VLX240T-1, that runs in 245.519MHz. For example, Figure 3.1(b) includes 33232 edge points. The circuit can perform the Hough transform in 135.75μs and the software implementation on the CPU performs in $37.10ms$. Also, if all the points of an image of size 512×512 are edge points, it takes 1068.11μs to output the results, where the software implementation takes $359.27ms$ to output the results. In other words, our FPGA implementation runs over 300 times faster than the sequential implementation on the CPU.

However, the implementation needs to accept the coordinates of edge points as input. Since pixel data of input images from digital video cameras are generally input in raster scan order, the requirement might not to be versatile. Also, since identified lines are obtained just by thresholding after voting, similar to lines in the input image but incorrect lines are also detected. Then, the second contribution of this chapter is to present an improved FPGA implementation of the Hough transform. One of the main different points from the previous implementation is that the improved FPGA implementation

processes pixel data in raster scan order and outputs the identified lines. Therefore, the voting time is a fixed clock cycles corresponding to the size of the image. Compared to the previous implementation above, the number of used DSP slices becomes approximately half. Our new idea that differs from the previous implementation above includes:

(i) More Efficient Usage of DSP slices: DSP slices are used $x\cos\theta$ and $y\sin\theta$ in parallel for each edge pixel (x, y). We compute $x\cos\theta$ and $y\sin\theta$ for θ such that $0 \leq \theta < 90$ instead of computing them for θ such that $0 \leq \theta < 180$. In addition, since pixel data are input in raster scan order, we use the fact that the value of y in a certain row is not change. When pixels in a certain row y are processed, we pre-compute $(y+1)\sin\theta$ for θ such that $0 \leq \theta < 90$ in the next row. According to the above, compared with the previous implementation, the number of used DSP slices is reduced to approximately half. More specifically, the improved implementation uses only 1 DSP slice to compute $y\sin\theta$, and uses 1 columns to compute $x\cos\theta$.

(ii) More precise line detection: In previous implementation, the straight lines are output such that the number of votes exceeds a certain threshold value. However, the output includes many mistaken lines due to the discretization error in voting. Therefore, after voting operation, to obtain more precise straight lines, we apply 3×3 maximum filters for the voted results.

For the image shown in Figure 3.1(b) of size 512×512, the software implementation performs the Hough transform in $41.408ms$. On the other hand, the improved FPGA implementation performs in $1.065ms$. Hence, the improved FPGA implementation attains a speed-up factor of more than 38 over the sequential implementation on the CPU.

Many hardware algorithms for FPGA implementation of the Hough transform for

lines have been proposed in past. In the existing researches, they introduced incremental Hough transform [45, 5, 10], CORDIC [24, 8], and hybrid-log arithmetic [30] to the computation of Hough transform. Since most of recent FPGAs produced by principal vendors equip embedded DSP slices [49, 50, 4], one of the most important key techniques for accelerating computation using FPGAs is an efficient usage of DSP slices and block RAMs.

3.1.2 Hough transform algorithm

The main purpose of this section is to review Hough transform algorithms for straight lines. Suppose that we have an image of size $n \times n$. We assume that $n \times n$ pixels are arranged in two dimensional xy-space such that the origin is in the center of the image as illustrated in Figure 3.2. Hence, both coordinates x and y take integers in the range

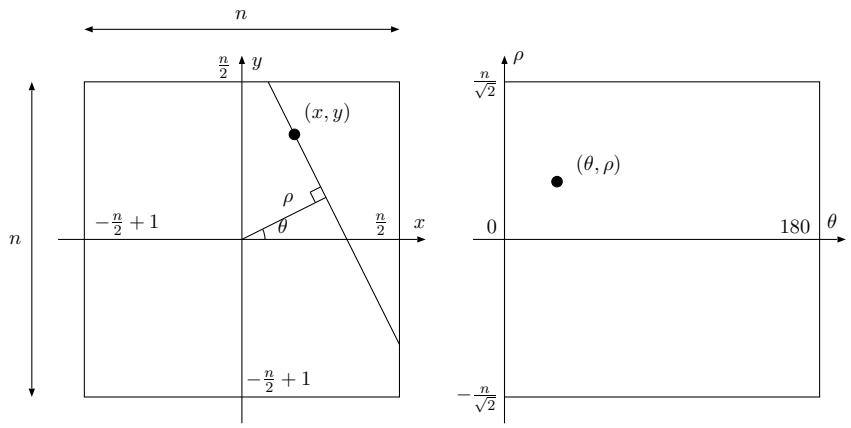


Figure 3.2: Two dimensional Spaces xy and $\theta\rho$ used in the Hough transform

$[-\frac{n}{2}+1, \frac{n}{2}]$. A pixel (x,y) $(-\frac{n}{2}+1 \leq x,y \leq \frac{n}{2})$ in the xy-space is converted to a curve in the $\theta\rho$-space by the following formula: $\rho = x\cos\theta + y\sin\theta \quad (0 \leq \theta < 180)$. Clearly, the double inequality $-\frac{n}{\sqrt{2}} < \rho \leq \frac{n}{\sqrt{2}}$ is satisfied. The values of θ and ρ can also be obtained

geometrically. Suppose that we draw a line going through the origin with angle θ as illustrated in Figure 3.2. For such lines, we can draw the orthogonal line going through a pixel (x, y). The value of ρ corresponds to the distance to the line. In other words, a point (θ, ρ) of $\theta\rho$-space corresponds to a line of xy-space.

The key idea of the Hough transform is to vote in $\theta\rho$-space only for every edge pixel in the xy-space. Let $(x_0, y_0), (x_1, y_1), \ldots, (x_{k-1}, y_{k-1})$ be the k pixels in xy-space. Suppose that the coordinates of edge points edge points are given, the Hough transform is spelled out as follows:

[Straight Forward Hough Transform]

for $i \leftarrow 0$ to $k - 1$

  for $\theta \leftarrow 0$ to 179

    begin

      $\rho \leftarrow x_k \cos\theta + y_k \sin\theta$

      $v[\theta][\rho] \leftarrow v[\theta][\rho] + 1$

      output (θ, ρ) if $v[\theta][\rho] = threshold$

    end

For simplicity, we assume that the value of ρ is automatically rounded to an integer. In the Straight Forward Hough Transform, for each point (x, y), the values of $x\cos\theta$ and $y\sin\theta$ are computed for $\theta = 0, 1, \ldots, 179$. If $v[\theta][\rho]$ is storing a large value, many edge pixels in the input pixels lie in the line in xy-space corresponds to a point (θ, ρ) in $\theta\rho$-space.

We will show that, it is sufficient to compute these values for $\theta = 0, 1, \ldots, 90$. From the addition theorem of trigonometric functions, we have

$$\rho = x\cos(180 - \theta) + y\sin(180 - \theta)$$

$$= \quad -x\cos(\theta) + y\sin(\theta). \tag{3.1}$$

Using Formula (3.1), the Hough transform can also be done by partitioning the range $[0, 179]$ of θ into two ranges $[0, 89]$ and $[90, 179]$. Also, we avoid going through array v for finding elements larger than a threshold. Thus, our new Hough transform, called the Circuit-oriented Hough Transform, is spelled out as follows:

[Circuit-oriented Hough Transform]

for $i \leftarrow 0$ to $k - 1$

begin

for $\theta \leftarrow 0$ to 89 do

begin

$\rho \leftarrow x_k \cos\theta + y_k \sin\theta$

$v[\theta][\rho] \leftarrow v[\theta][\rho] + 1$

output (θ, ρ) if $v[\theta][\rho] =$ *threshold*

end

for $\theta \leftarrow 1$ to 90 do

begin

$\rho \leftarrow -x_k \cos(\theta) + y_k \sin(\theta)$

$v[180 - \theta][\rho] \leftarrow v[180 - \theta][\rho] + 1$

output (θ, ρ) if $v[\theta][\rho] =$ *threshold*

end

end

Recall that the FPGA implementation of the Circuit-oriented Hough transform is improved to process all pixels of the image in raster scan order, and maximum filters are applied to obtained more precise straight lines. Let (x, y) be the pixel in xy-space, and let $p[x][y]$ be the value of the pixel such that $p[x][y] = 1$ if a pixel (x, y) is an edge

pixel and $p[x][y] = 0$ if a pixel (x, y) is a non-edge pixel. The Circuit-oriented Hough Transform for the improved FPGA implementation, is spelled out as follows:

[Circuit-oriented Hough Transform for the improved implementation]

for $y \leftarrow -\frac{n}{2} + 1$ to $\frac{n}{2}$

  for $x \leftarrow -\frac{n}{2} + 1$ to $\frac{n}{2}$

    if $p[x][y] = 1$

      begin

        for $\theta \leftarrow 0$ to 89 do

          begin

            $\rho \leftarrow x \cos\theta + y \sin\theta$

            $v[\theta][\rho] \leftarrow v[\theta][\rho] + 1$

          end

        for $\theta \leftarrow 1$ to 90 do

          begin

            $\rho \leftarrow -x \cos(\theta) + y \sin(\theta)$

            $v[180 - \theta][\rho] \leftarrow v[180 - \theta][\rho] + 1$

          end

      end

In the following sections, we show the FPGA implementation of the Circuit-oriented Hough transform and the improved FPGA implementation, respectively.

3.1.3 FPGA architecture for the Hough transform

This section describes our FPGA architecture for the Hough transform using DSP slices and block RAMs in Xilinx Virtex-6 Family FPGA XC6VLX240T-1 [53].

Figure 3.3 illustrates our architecture for the Hough transform. We use 178 DSP

blocks $X_1, X_2, \ldots X_{89}$ and $Y_1, Y_2, \ldots, Y_{89}$. For each θ ($0 \leq \theta \leq 90$) X_θ and Y_θ compute $x_k \cos\theta$ and $y_k \cos\theta$ for given x_k and y_k, respectively. Since $x_k \cos 0 = x_k$, $x_k \cos 90 = 0$, $y_k \sin 0 = 0$, and $y_k \cos 90 = y_k$, DSP blocks X_0, X_{90}, Y_0, and Y_{90} are not necessary. Using an adder and a subtractor for each pair X_θ and Y_θ, $\rho_\theta = x_k \cos\theta + y_k \cos\theta$ and $\rho_{180-\theta} = -x_k \cos\theta + y_k \cos\theta$ are computed. We also use 180 block RAMs $V_0, V_1, \ldots V_{179}$ to store the voting value. Address ρ of each V_θ ($0 \leq \theta \leq 179$) is used to store the value of $v[\theta][\rho]$.

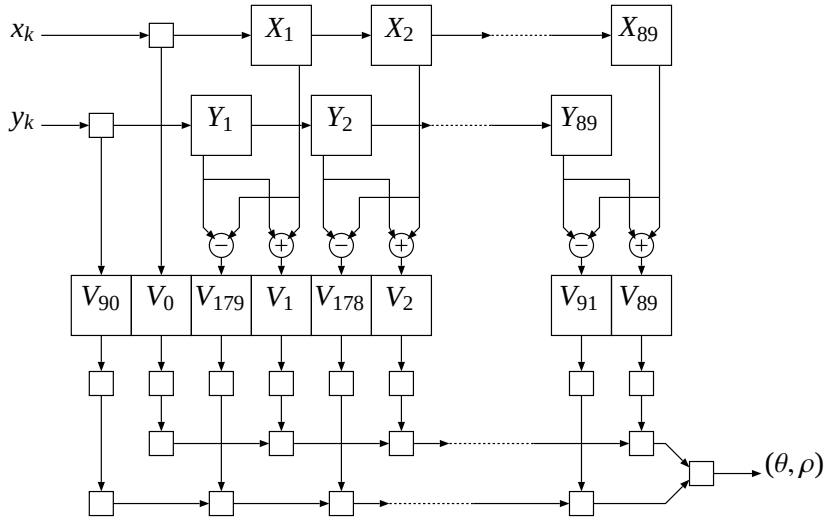


Figure 3.3: The outline of our FPGA architecture for the Hough transform

To minimize the delay between registers, DSP blocks $X_1, \ldots, X_{90}$ are connected in a pipeline fashion as illustrated in Figure 3.3. Each X_θ has a register to store the value of x_k. In every clock cycle, the value is transferred from X_θ to $X_{\theta+1}$. Similarly, DSP blocks $Y_0, Y_1, \ldots, Y_{90}$ are connected in a pipeline fashion.

Figure 3.4 illustrates two DSP blocks X_θ and Y_θ with an adder and subtractor to compute ρ. In X_θ, the value of x_k is loaded in an internal register. Also, the value of $\cos\theta$ is pre-computed. Note that the value of $\cos\theta$ used in X_θ is a fixed value. The

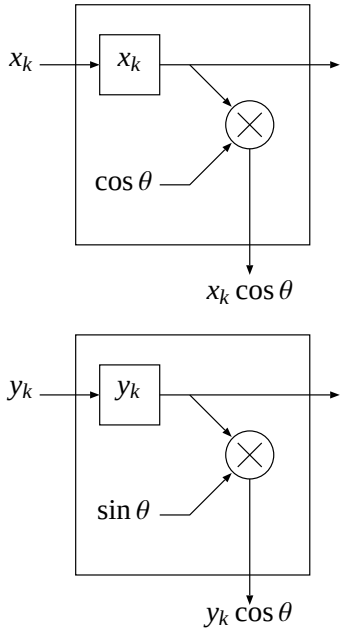


Figure 3.4: Two DSP blocks X_θ and Y_θ with an adder and subtractor to compute ρ

product of x_k and $\cos\theta$ is computed in a multiplier of the DSP block X_θ. Similarly, the value of $\sin\theta$ used in Y_θ is a fixed value and the product of y_k and $\sin\theta$ is computed in a multiplier of the DSP block Y_θ.

In the Virtex-6 FPGA XC6VLX240T, that is our target device, has DSP48E1 blocks are arranged in 8 columns of 96 adjacent DSP48E1 blocks. Neighboring DSP48E1 blocks are connected directly through pipeline registers. Our Hough transform architecture uses 2 columns to compute $x_k \cos\theta$ and $y_k \sin\theta$ each, and uses a pipeline technique to maximize the clock frequency (Figure 3.5).

Figure 3.6 illustrates the architecture of V_θ using a block RAM. A block RAM in the FPGA is dual port architecture. Xilinx Virtex-6 Family has 18Kbit dual-port block RAMs, which have two sets of ports operated independently. Two sets of ports are:

Port Set A: *ADDRA* (ADDRess A), *DOA* (Data Output A), *DIA* (Data Input A)

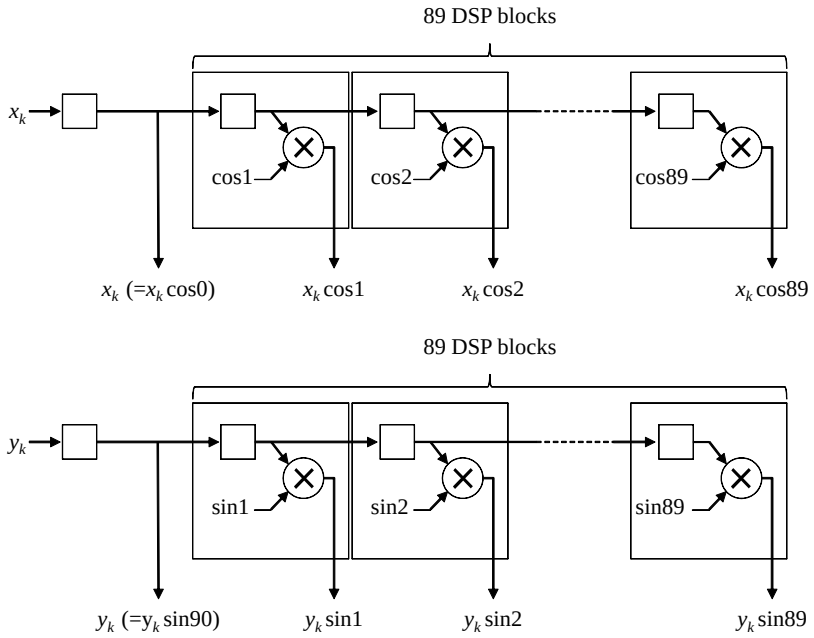


Figure 3.5: Pipeline architecture to compute $x_k \cos\theta$ and $y_k \sin\theta$ with DSP blocks

Port Set B: *ADDRB* (ADDRess B), *DOB* (Data Output B), *DIB* (Data Input B).

Let $M[i]$ denote a data of address i of the block RAM. In read operation of Port Set A, $M[ADDRA]$ is output from *DOA* after the rising clock edge. In write operation of Port Set A, the data given to *DIA* is written in $M[ADDRA]$ at the rising clock edge. Read/write operations of Port Set B are the same as Port Set A. Port Set A and Port Set B work independently. In the block RAMs in the target device of this work, read/write operations can be configured as either RF (Read First) mode or WF (Write First) mode. In the RF mode, if reading and writing operations are performed to the same address, reading operation is performed before the reading operation. Hence the reading data is the data before writing data. On the other hand, in the WF mode, since the writing performed before the reading, the reading data is the updated data. However, when a dual port is used, there is a restriction that if read and write operation to the same address

are performed for each port, the setting of block RAMs must be RF as mentioned in Chapter 2.

We use the block RAM to store the values of $v[\theta][\rho]$ ($-\frac{n}{\sqrt{2}} < \rho \leq \frac{n}{\sqrt{2}}$). Let $v_\theta[i]$ denote the data of address i in block RAM V_θ. Since ρ is given to it ADDRA, $v_\theta[\rho]$ is output from *DOA* after the rising clock edge as illustrated in Figure 3.6. After that, $v_\theta[\rho] + 1$ is computed and it is given to *DOB*. Since ρ is given to *ADDB*, $v_\theta[\rho] + 1$ is written in $v_\theta[\rho]$. In other words, $v_\theta[\rho] \leftarrow v_\theta[\rho] + 1$ is performed. At that time, according to the restriction stated in the above, since the same value of ρ may be input continuously, the setting of block RAMs must be RF. Namely, when the same value of ρ is input continuously, the former voted value is not read from the block RAM. To avoid this situation, we use an additional register to store the latest voted value and if the same value of ρ is input continuously, the stored value is used instead of the value read from the block RAM.

In the same time, a comparator is used to determine if $v_\theta[\rho] + 1 = \mathit{threshold}$. If so, the value of ρ is written in a register. After that, a pair (θ, ρ) is written into a next register. The pair (θ, ρ) represents a probable line. It moves toward the output of the circuit using series of shift registers one by one shown in Figure 3.3. In order to reduce the number of clock cycles necessary to move data to the output, we use two series of shift registers. One is used for output data of $V_0, \ldots, V_{89}$. The other is used for output data of $V_{90}, \ldots, V_{179}$. Therefore, the number of clock cycles necessary to move data to the output is reduced to at most 90 clock cycles.

The choice of data precision is guided by the implementation cost in terms of area, simplicity of design, speed and power consumption. Higher precision will lead to less quantization error in the final implementation. On the other hand, lower precision will

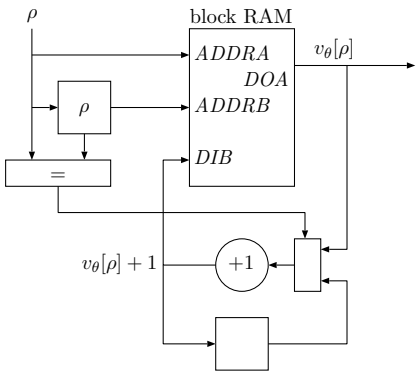


Figure 3.6: A block RAM V_θ to store $v[\theta][\rho]$

produce more compaction and faster designs with less power consumption. A trade-off choice needs to be made depending on the given application and available FPGA resources.

In our work, in order to minimize chip space and computation time, short fixed point representation of numbers are used. Considering the structure of DSP blocks and block RAMs, we choose the data presentation in our implementation, as follows. The data format of inputs that are pairs of coordinates x_k and y_k are 10bit two's complement integer each. Also, the data format of $\cos\theta$ and $\sin\theta$ is 16bit fixed point number, which consists of 1bit sign, 1bit integer and 14bit fraction based on two's complement. On the other hand, the data format of ρ is 10bit two's complement integer. The data format of the voted value is 18bit integer. Namely, the number of the vote is at most $2^{18} - 1$. Since the range of the value of θ is 0 to 180, the data format of θ is 8bit integer.

3.1.4 Improved FPGA architecture for the Hough transform

This section describes the improved FPGA architecture for the Hough transform that uses approximately half of DSP slices and processes all pixels in raster scan order.

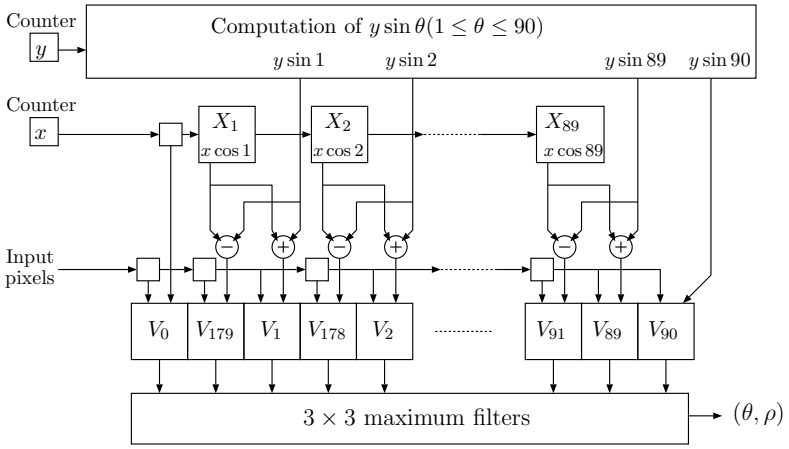


Figure 3.7: The outline of the improved FPGA architecture for the Hough transform

Figure 3.7 illustrates the outline of our FPGA architecture for the Hough transform. Whenever each input pixel is given, the two counters for x and y increment appropriately. We use 89 DSP slices $X_1, X_2, \ldots X_{89}$. For each θ ($0 \le \theta \le 90$), X_θ computes $x \cos\theta$. Since $x \cos 0 = x$ and $x \cos 90 = 0$, DSP slices X_0 and X_{90} are not necessary. Also, we use a module to compute $y \sin\theta$ ($1 \le \theta \le 90$). Using an adder and a subtractor for $x \cos\theta$ and $y \sin\theta$, $\rho_\theta = x \cos\theta + y \sin\theta$ and $\rho_{180-\theta} = -x \cos\theta + y \sin\theta$ are computed. We also use 180 block RAMs $V_0, V_1, \ldots, V_{179}$ to store the voting value. Address ρ of each V_θ ($0 \le \theta \le 179$) is used to store the value of $v[\theta][\rho]$. After voting, to obtain identified straight lines, we use 3×3 maximum filters. These filters simultaneously work row by row.

As the same as the implementation of previous section, to minimize the delay between registers, DSP slices $X_1, \ldots, X_{90}$ are connected in a pipeline fashion as illustrated in Figure 3.7. Each X_θ has a register to store the value of x. In every clock cycle, the value is transferred from X_θ to $X_{\theta+1}$. The DSP slice X_θ is the same as illustrated in Fig-

ure 3.4. In X_θ, the value of x is loaded in an internal register. Note that the value of $\cos\theta$ used in X_θ is a fixed value. The product of x and $\cos\theta$ is computed in a multiplier of the DSP slice X_θ. Also, the improved Hough transform architecture uses 1 column to compute $x\cos\theta$, and uses a pipeline technique to maximize the clock frequency as show in Figure 3.5.

On the other hand, to compute $y\sin\theta$ $(1 \leq \theta \leq 90)$ we use the fact that the value of y in a certain row is not change since pixel data are input in raster scan order. Therefore, when pixels in a certain row y are processed, we pre-compute $(y+1)\sin\theta$ for θ such that $0 \leq \theta < 90$ in the next row and store them into the registers. In the next row $y+1$, the computed values of $(y+1)\sin\theta$ are used. Figure 3.8 illustrates our architecture to compute $y\sin\theta$. We use a look-up-table using a block RAM to compute $\sin\theta$. and a DSP slice to compute a product of y and $\sin\theta$. Also, we utilize two series of registers, called banks. One is used to pre-compute the values of $y\sin\theta$ for the next row. The other is used to output the already computed $y\sin\theta$ for the current processing row. To compute the values of $\sin\theta$ we successively generate the value of $\theta = 90, 89, 88, \ldots, 2, 1$ by a counter. By inputting them to the look-up-table, the values of $\sin\theta$ are obtained. Using the DSP slice, the products of $y\sin\theta$ are computed. Note that the values of $y\sin\theta$ is for the next row. Therefore, the value of y is incremented in advance. The obtained values are successively input to a bank. In each bank, registers are cascaded shown in the figure. The values shift one by one until all the values are input to the bank. When pixels in a row are finished, the banks are switched.

Let $v_\theta[i]$ denote the data of address i in block RAM V_θ. Since ρ is given to it ADDRA, $v_\theta[\rho]$ is output from *DOA* after the rising clock edge as illustrated in Figure 3.6, that is the same with the implementation of the previous section. After that,

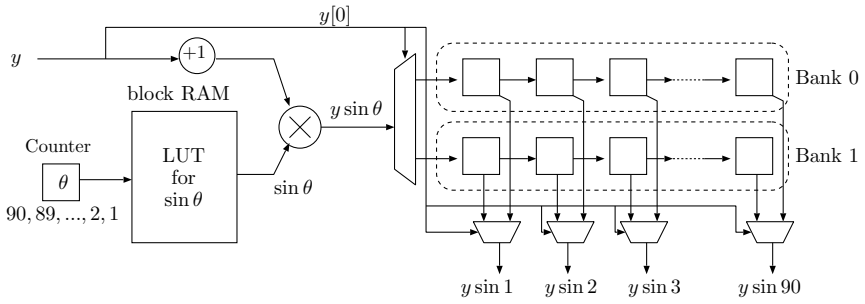


Figure 3.8: Architecture of computing $y \sin\theta$ with one DSP slice

$v_\theta[\rho] \leftarrow v_\theta[\rho] + 1$ is performed. Also, we use an additional register to store the latest voted value and if the same value of ρ is input continuously, the stored value is used instead of the value read from the block RAM. Recall that the improved implementation processes all pixels in raster scan order. Note that the above voting process is performed when the input value is an edge pixel. Namely, when it is a non-edge pixel, the voting process is not performed.

In the following, when all the voting operations are completed, we utilize 3×3 maximum filters to output the final correct identified straight lines. The maximum filter is defined as the maximum of all pixels within a local region of an image. In here, for each value in the voting space, this filter copies the largest value from a 3×3 region to it. In the voting process, the vote concentrates to each point (θ, ρ) corresponding to a line in the original image. However, it also concentrates to around the point.

Figure 3.9 illustrates our architecture to perform a 3×3 maximum filter to the voted results. Since the voted values in the same ρ can be obtained from $V_0, V_1, \ldots, V_{179}$, this architecture works row by row in a pipeline fashion. To perform a 3×3 maximum filter to each value in a certain row, it is concurrently read from $V_0, V_1, \ldots, V_{179}$. After that, using comparators, local maxima of each 3 neighboring votes in the row are obtained.

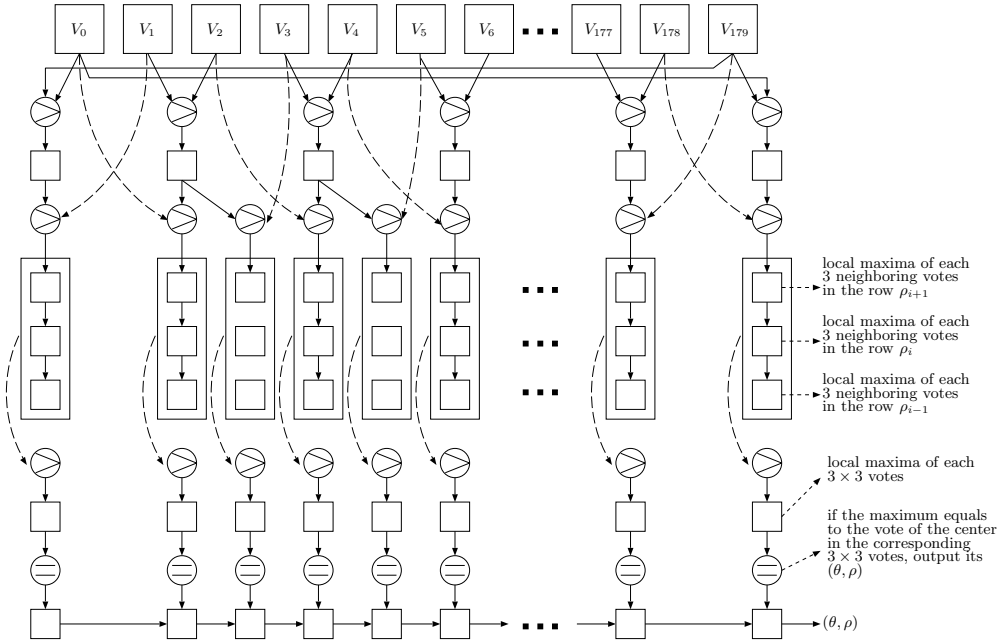


Figure 3.9: Pipeline architecture of 3×3 maximum filters

These local maxima are input shift registers. After local maxima in the 3 rows are computed, local maxima of each 3×3 votes are obtained by computing maxima from corresponding 3 values. If the maximum equals to the original value of the center in the corresponding 3×3 votes, its (θ, ρ) that represents a probable line is input to the shift registers and output through the registers.

In the improved implementation of the Hough transform, the data format of inputs are values (0 or 1) of all pixels in the image, these values are input in raster scan order. The coordinates (x, y) which are necessary to compute ρ are generated by the counter as shown in Figure 3.7. In order to minimize chip space and computation time, short fixed point representation of numbers are used. Considering the structure of DSP slices and block RAMs, we choose the data presentation in our implementation, as follows. The data format of inputs that are values of pixels $p[x][y]$ are 1bit binary number. The

data format of $\cos\theta$ and $\sin\theta$ is 16bit fixed point number, which consists of 1bit sign, 1bit integer and 14bit fraction based on two's complement. On the other hand, the data format of ρ is 10bit two's complement integer. The data format of the voted value is 18bit integer. Namely, the number of the vote is at most $2^{18} - 1$. Since the range of the value of θ is 0 to 180, the data format of θ is 8bit integer.

3.1.5 Experimental Results

We have implemented and evaluated our proposed architectures of the Hough transform on the Xilinx Virtex-6 FPGA XC6VLX240T-1. For the purpose of estimating the speed up of our implementations, we have also implemented a conventional software approach of the Hough transform using GUN C. We have used Intel Xeon X7460 (2.66GHz) and 128GB memory to run the sequential algorithm for the Hough transform.

In the evaluation of our original implementation that processes only edge points and outputs lines by only thresholding, Table 3.1 shows the experimental results using Xilinx ISE 13.1. In this implementation, to reduce the delay of the circuit, some pipeline registers are inserted into between circuit elements. It takes 3 clock cycles to compute the values of ρ for given coordinates x_k and y_k. Also, 4 clock cycles are necessary to output a pair (θ, ρ) that represents a probable line. Moreover, the number of clock cycles necessary to move data to the output is reduced to at most 90 clock cycles. Hence, this circuit can output identified lines represented by (θ, ρ) in $m + 97$ clock cycles, i.e., $\frac{m+97}{245.519}\mu$s. For example, Figure 3.1(b) includes 33232 edge points. The circuit performs the Hough transform in 135.75μs, where the software implementation on the CPU takes $37.10ms$. Also, if all the points of an image of size $512 \times 512 = 262144$ are edge points, it takes $1.068ms$ to complete to output the results, and the software implementation

takes 359.27ms. Of course, it is not possible that all points are edge points, however, this fact guarantees that this implementation for any 512×512 image terminates in less than 1.068ms. Therefore, our original implementation attains a speed-up factor of more than 300 over the sequential implementation on the CPU.

Table 3.1: Performance evaluation of the proposed architectures for Hough transform

	original work	improved work
DSP48E1 slices (out of 768)	178 (23.1%)	90 (11.1%)
18Kbit block RAMs (out of 832)	180 (21.6%)	181 (21.7%)
Slices (out of 301440)	14493 (4.81%)	40487 (13%)
Clock frequency [MHz]	245.519	247.525

On the other hand, in the evaluation of the improved implementation that processes all pixels in raster scan order and outputs lines using 3×3 maximum filters, Table 3.1 shows the experimental results. To compute $y \sin\theta$ for $(1 \leq \theta \leq 90)$ in the first row, 94 clock cycles are necessary. It takes 3 clock cycles to compute the values of ρ for given x and the precomputed $y \sin\theta$. Also, 4 clock cycles are necessary to vote to the Hough space. Furthermore, to perform vote for each V_θ, the number of clock cycles necessary to move data from the leftmost register to the rightmost register as shown in Figure 3.7 is 91. Since all of the points in the binary image are input into the improved implementation, the voting operations are performed for an $n \times n$ image in $n^2 + 192$ clock cycles. After voting, $\sqrt{2}n + 187$ clock cycles are necessary to output identified straight lines with 3×3 maximum filters. Hence, in total, $n^2 + \sqrt{2}n + 379$ clock cycles, i.e., $\frac{n^2+\sqrt{2}n+379}{247.525}\mu$s are necessary to perform the Hough transform for an $n \times n$ image. Thus, our circuit completes the Hough transform for an 512×512 image in 1.065ms. For the image shown in Figure 3.1(b) of size 512×512, the software implementation performs the

Hough transform in 41.08ms. Therefore, our improved FPGA implementation attains a speed-up factor of more than 38 over the sequential implementation on the CPU. If all points of the image are edge points, the improved FPGA implementation runs over 300 times faster than the sequential implementation on the CPU.

Table 3.2: Comparison with related works for Hough transform

	Karabernou [24]	Deng [8]	Lee [30]	original work	improved work
Device	XC4010EPC84	XC4010XL	Virtex 4	XC6VLX240T-1	XC6VLX240T-1
Logic blocks	205 CLBs	333 CLBs	314 CLBs	14493 Slices	40487 Slices
DSP slices	—	—	—	178 DSP48E1s	90 DSP48E1s
Frequency	23.166MHz	40MHz	132MHz	245.519MHz	247.525MHz
Throughput	10.368Mpixel/s	0.623Mpixel/s	32.768Mpixel/s	245.428Mpixel/s	246.219Mpixel/s

There are a number of literatures reported to implement Hough transform for lines using the FPGA shown in Section 3.1.1. Performances such as device, logic blocks, DSP slices, frequency and throughput are compared in Table 3.2. It is difficult to directly compare to other works because utilized FPGAs and supported size of images differ. Considering the throughput, however, it is clear that the performance of our FPGA implementations are better than that of other works. In addition, although the improved implementation takes more time than our previous work to perform Hough transform, the number of DSP slices are less than the original implementation, and the result is filtered.

3.1.6 Concluding remarks

We have presented two architectures of the Hough transform for the straight lines using DSP slices and block RAMs in the Virtex-6 Family FPGA. The original FPGA im-

plementation of the Hough transform uses 178 DSP48E1 slices and 180 18kbit block RAMs. The implementation results show that the our original implementation runs at 245.519MHz, and performs the Hough transform for a image with m edge points in $m + 97$ clock cycles. On the other hand, we improved the original implementation to process all pixels of image in raster scan order and reduce the usage of DSP slices. Also, maximum filters are applied to obtain more precise lines after the voting operation. The experimental results show that the improved implementation uses 91 DSP48E1 slices and 181 18kbit block RAMs. The improved implementation runs at 247.525MHz, and performs the Hough transform for a binary image of size $n \times n$ in $n^2 + \sqrt{2}n + 379$ clock cycles. Compared to the conventional CPU implementation of the Hough transform, our implementations achieve a sufficient speed-up.

3.2 Efficient implementation of the Gradient-based Hough transform algorithm for extracting lines

The gradient-based hough transform is an improvement of the original Hough transform. It is utilized to reduce substantially the computation quantity and make the detection more accurate using gradient information. We show an efficient implementation of the gradient-based Hough transform for straight lines detection using a Xilinx Virtex-7 FPGA in this section.

3.2.1 Introduction

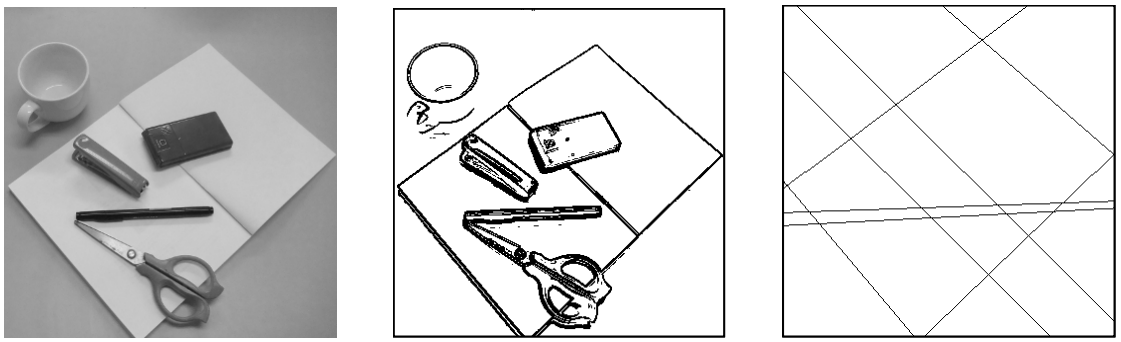
The Hough transform can be used to extract straight lines in a binary image [11]. As mentioned in the previous section, the idea of this method is to exploit the duality be-

tween points of a line and parameters of that line. More specifically, for each edge point (x, y) in a 2-dimensional image, the voting is performed along a curve $\rho = x\cos\theta + y\sin\theta$ $(0 \leq \theta < 180)$. Possible lines can be detected by searching points that are voted intensively. For an input image (Figure 3.10(a)), the binary edge image (Figure 3.10(b)) is obtained by the edge detector such as Sobel filter. We can see that the normal Hough transform performs well basing on the pure edge image. The result of voting to the parameter space is shown in Figure 3.11(a). In this figure, darker points show points that are voted intensively, that is, represent probable lines. According to the result of voting, the principal lines are detected (Figure 3.10(c)).

There are many improvements to the Hough transform for line detection [21]. One of the efficient improvements is using gradient information [39]. The idea of the method is to utilize gradient direction and magnitude. It is based on the fact that if a given point happens to indeed be on a line, **(i)** The local direction of the gradient gives approximately the same direction of the actual line. **(ii)** The gradient magnitude at the pixel is higher than that of other points not lying on lines. Using these ideas, we reduce the number of useless votes by limiting the range of votes with the local gradient direction, and weight voted values proportional to the local gradient magnitude to enhance the votes of pixels on lines. In the following, the straight forward Hough transform shown in Section 3.1.2, is called *conventional Hough transform*, and the Hough transform using gradient information is called *gradient-based Hough transform* to distinguish them easily. In our implementation, we use the Sobel filter, which is used in edge detection algorithms [38] to obtain the gradient information. Figure 3.11 (b) shows the resulting Hough space based on the above ideas. Compared with that of the conventional Hough transform, we can see that votes are limited to the several parts that are darker points in

the figure. Actually, these correspond to real lines in the image and it is easy to find that useless votes are reduced.

The third contribution of this chapter is to present an efficient architecture for the gradient-based Hough transform. Our implementation uses 13 DSP48E1 slices, 180 block RAMs with 36kbits and 8 block RAMs with 18kbits. We have implemented our architecture on a Virtex-7 XC7VX485T-2. Our proposed circuit runs at 260.061MHz, and the voting operations are performed for an $n \times n$ gray-scale image in $n^2+2n+44$ clock cycles. After the voting operation, our circuit outputs the identified lines in $\sqrt{2}n + 188$ clock cycles. Therefore, our circuit can perform the gradient-based Hough transform in $n^2 + (\sqrt{2} + 2)n + 232$ clock cycles.



(a) Input image

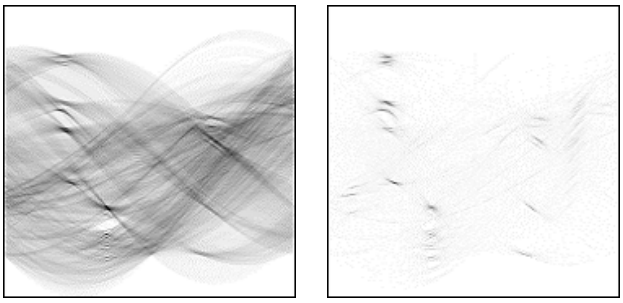
(b) Binary edge image by Sobel filter

(c) Line detection using the Hough transform

Figure 3.10: Example of straight lines detection using Hough transform

3.2.2 Gadient-based Hough transform algorithm

In the gradient-based Hough transform, lines detection is performed for a gray-scale image, not a binary image. To obtain the gradient information for a gray-scale image, we use the Sobel filter. The Sobel filter is applied on the image for approximating the



(a) Conventional (b) Gradient-based

Figure 3.11: Hough parameter spaces of the conventional Hough transform and gradient-based Hough transform

vertical and horizontal derivatives using a couple of 3×3 convolutions G_x and G_y:

$$G_x = \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix} \otimes I, G_y = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} \otimes I, \tag{3.2}$$

where I represents the input image and $\otimes$ denotes the 2-dimensional convolution operation. The two results convolved by G_x and G_y are approximations of the gradient for horizontal and vertical of the image, respectively. At each pixel in the image, the resulting gradient approximations can be combined to obtain the gradient magnitude using the formula:

$$G = \sqrt{{G_x}^2 + {G_y}^2}. \tag{3.3}$$

We can also compute the gradient direction θ' using

$$\theta' = \tan^{-1}(\frac{G_y}{G_x}). \tag{3.4}$$

Based on the gradient direction obtained by the above, we vote to the parameter space. However, there is an error between local gradient direction and the direction of actual

lines due to the quantization error. Therefore, voting operation is performed not only to the angle obtained by the gradient direction, but also angles in the vicinity of it. In our implementation, we introduce weighted values w that depends on the angle as follows:

$$w(\theta - \theta') = \begin{cases} 2^{\lambda - |\theta - \theta'|} & |\theta - \theta'| \leq \lambda \\ 0 & \text{otherwise.} \end{cases} \tag{3.5}$$

To be suitable for the compact FPGA implementation, we use the weights as power-of-two numbers. Also, the voting range is limited to $[-\lambda, +\lambda]$ instead of the range $[0, 179]$ in the conventional Hough transform. The gradient-based Hough transform is spelled out as follows:

[Gradient-based Hough Transform]

for $y \leftarrow -\frac{n}{2} + 1$ to $\frac{n}{2}$

  for $x \leftarrow -\frac{n}{2} + 1$ to $\frac{n}{2}$

    Compute G and θ' for $p[x][y]$

    for $\theta \leftarrow \theta' - \lambda$ to $\theta' + \lambda$ do

      begin

        if $\theta < 0$ then $\theta \leftarrow \theta + 180$

        $\rho \leftarrow x \cos\theta + y \sin\theta$

        $v[\theta][\rho] \leftarrow v[\theta][\rho] + G \cdot w(\theta - \theta')$

      end

Simply speaking, the gradient-based Hough transform votes for each pixel of the gray-scale image with a weighted value $G \cdot w(\theta - \theta')$ which is proportional to the gradient magnitude. The parameter space will be sharpened by such voting operations that make the accuracy higher. Our implementation for the computation of the gradient direction and magnitude is a pipelined architecture. In the following section, we show the efficient implementations of the gradient-based Hough Transform on the FPGA.

3.2.3 FPGA architecture for the gradient-based Hough transform

This section describes our FPGA architecture for the gradient-based Hough transform using DSP slices and block RAMs in Xilinx Virtex-7 Family FPGA XC7VX485T-2 as the target device [56]. Figure 3.12 illustrates the outline of our architecture. The details are described as follows.

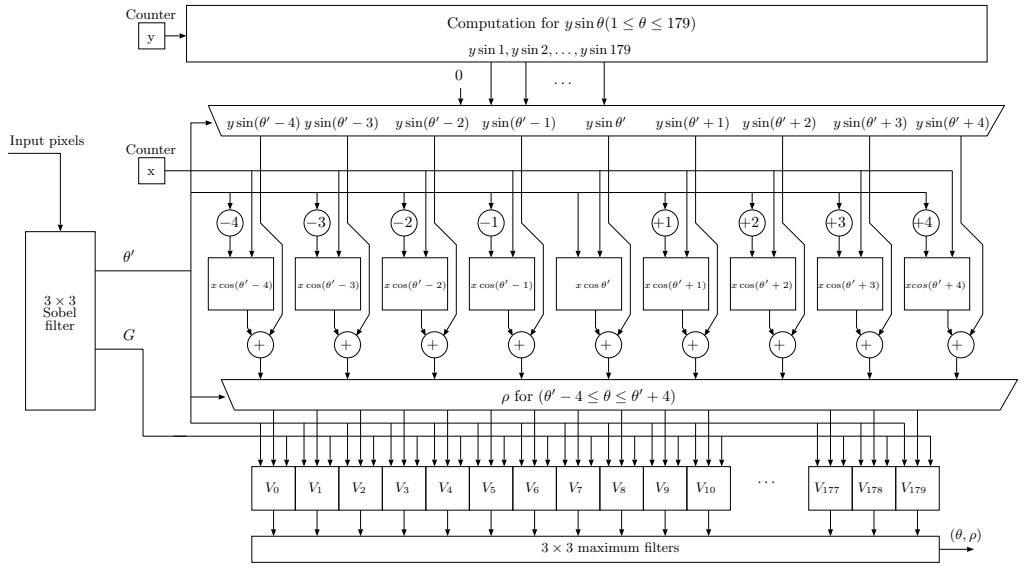
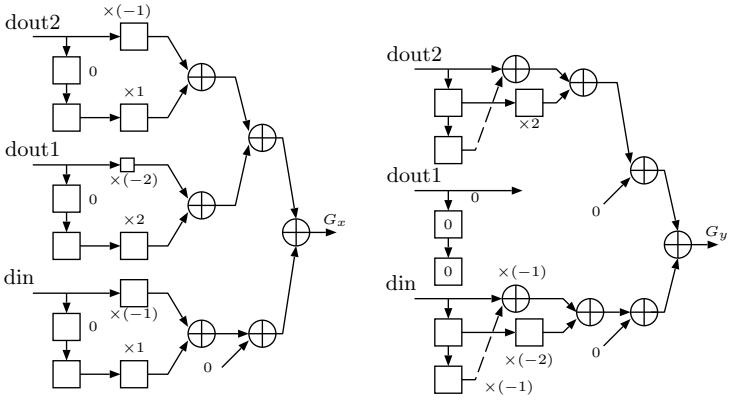


Figure 3.12: The outline of our FPGA architecture for the gradient-based Hough transform ($\lambda = 4$)

[Structure for the computation of gradient information]

In our architecture, we use a 3×3 Sobel filter to obtain gradient information. Since we assume that input pixels are given to the circuit in the raster scan order, we use a two-lines buffer with block RAMs to provide pixels in each 3×3 subimage to the filter. Our circuit computes the horizontal and vertical derivative approximations using combinational circuits as shown in Figure 3.13, where *dout2*, *dout1* and *din* represent

pixel values in the three lines of the input image, respectively.



(a) G_x (b) G_y

Figure 3.13: Structure for the computation of G_x and G_y

As mentioned before, the algorithm needs the gradient direction and magnitude as shown in Section 3.2.2. The formula includes the computation of the square root and the inverse tangent. Since it is difficult to compute them directly on the circuit, we use the CORDIC IP provided by Xilinx [57]. The CORDIC IP provides a hardware module that is fully pipelined architecture and available easily on the FPGA.

[Structure for the computation of ρ and voting operation]

Given the gradient direction θ' and magnitude G of each pixel are obtained with a pipelined architecture, the circuit computes ρ and performs voting operation. Whenever the gradient magnitude G and the gradient direction θ' of each pixel are given, the two counters for x and y increment appropriately.

We use DSP slices and block RAMs to compute $x\cos(\theta' - \lambda), \ldots, x\cos(\theta' + \lambda)$. The detail of each circuit that computes $x\cos\theta$ is shown in Figure 3.14. The circuit consists of one DSP slice and one block RAM. Using the block RAM as a look-up-table, $\cos\theta$ is computed and the DSP slice computes the product of x and $\cos\theta$. We note that in

our implementation, since two circuits can share the two block RAMs for the look-up-table with the dual port, we use $2\lambda + 1$ DSP slices and $\lceil \frac{2\lambda+1}{2} \rceil$ block RAMs to compute $x\cos(\theta' - \lambda), \ldots, x\cos(\theta' + \lambda)$. For simplicity, the sharing is omitted in Figure 3.12 though it seems that every circuit that computes $x\cos\theta$ has one block RAM.

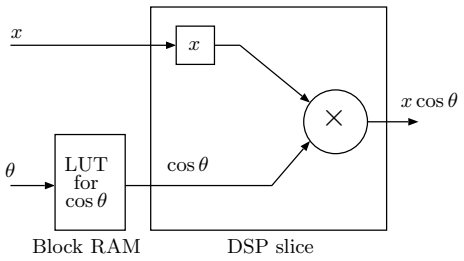


Figure 3.14: A DSP slice and a block RAM to compute $x\cos\theta$

Also, to compute $y\sin\theta$ $(1 \leq \theta \leq 179)$, we use the same architecture as illustrated in Figure 3.8 of Section 3.1.4. We use the fact that the value of y does not change while processing pixels are in a certain row. Therefore, when pixels in a row y are processed, we pre-compute the values of $(y + 1)\sin\theta$ for $(1 \leq \theta \leq 179)$ in the next row and store them into the registers, that differs from the value of $(1 \leq \theta \leq 90)$ as shown in Figure 3.8. When pixels in the next row $y + 1$ are processed, the values are used. We utilize a block RAM as a look-up-table to compute $\sin\theta$, and one DSP slice to compute a product of y and $\sin\theta$. Also, we use two series of registers, called banks. One is used to pre-compute the values of $y\sin\theta$ for the next row. The other is used to output the already computed $y\sin\theta$ for the current processing row. To compute the values of $\sin\theta$, we successively generate the value of $\theta = 179, 178, ...2, 1$ by a counter. By inputting them to the look-up-table, the values of $\sin\theta$ are obtained. The products of $y\sin\theta$ are computed using a DSP slice. Note that the values of $y\sin\theta$ are for the next row. Therefore, the value of

y is incremented in advance. The obtained values are successively input to a bank of registers. In each bank, registers are connected in cascade. The values shift one by one until all the values are input to the bank. When pixels in a row are finished, the banks are switched.

Figure 3.15 illustrates the architecture of V_θ using a block RAM. Given gradient direction θ' and magnitude G, the voted value $G \cdot w(\theta - \theta')$ is computed, where θ is a constant value in each V_θ. Since in our implementation the value of $w(\theta - \theta')$ is power of two shown in Section 3.2.2, it can be computed with a subtractor and a bit shifter. A block RAM in Xilinx Virtex-7 Family FPGA is dual port architecture, has two sets of ports operated independently.

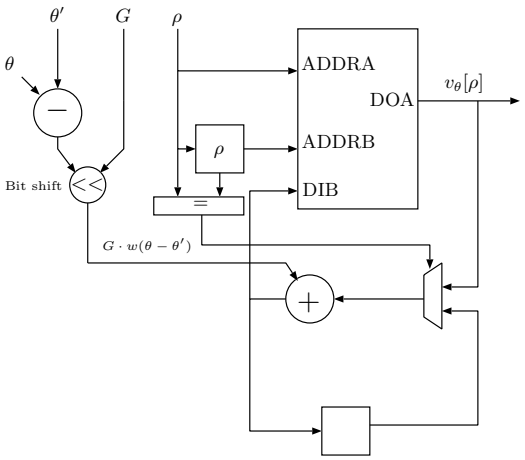


Figure 3.15: A block RAM V_θ to store $v[\theta][\rho]$

We use the block RAM to store the values of $v[\theta][\rho]$ ($-\frac{n}{\sqrt{2}} < \rho \leq \frac{n}{\sqrt{2}}$). Let $v_\theta[i]$ denote a data of address i of the block RAM V_θ. Since ρ is given to its *ADDRA*, $v_\theta[\rho]$ is output from *DOA* after the rising clock edge as illustrated in Figure 3.15. After that, $v_\theta[\rho] + G \cdot w(\theta - \theta')$ is computed and it is given to *DIB*. Since ρ is given to *ADDB*, $v_\theta[\rho] + G \cdot w(\theta - \theta')$ is written in $v_\theta[\rho]$. In other words, $v_\theta[\rho] \leftarrow v_\theta[\rho] + G \cdot w(\theta - \theta')$

is performed. At that time, according to the restriction stated in Section 2.3, since the same value of ρ may be input continuously, the setting of block RAMs must be RF. Namely, when the same value of ρ is input continuously, the former voted value is not read from the block RAM. To avoid this situation, we use an additional register to store the latest voted value and if the same value of ρ is input continuously, the stored value is used instead of the value read from the block RAM.

After all the voting operations are completed, we utilize the maximum filters as shown in Figure 3.9, to output the correct identified straight lines. In here, for each value in the voting space, the filter copies the largest value from a 3×3 region to it. Since the voted values in the same ρ can be obtained from $V_0,\ V_1, \ldots,\ V_{179}$, this architecture works row by row in a pipeline fashion. All values in a certain row is concurrently read from $V_0,\ V_1, \ldots,\ V_{179}$. After that using comparators, local maxima of each 3 neighboring votes in the row are obtained. These local maxima are input to shift registers. After local maxima in the 3 rows are computed, local maxima of each 3×3 votes are obtained by computing maxima from corresponding 3 values. If the maximum equals to the original value of the center in the corresponding 3×3 votes, its (θ, ρ) that represents a probable line is input to the shift registers and output through the registers.

In this implementation, the data format of inputs are 8-bit integer of all pixels in the gray-scale image and these values are input in raster scan order. The coordinates (x, y) which are used to compute ρ are appropriately generated by the counters. In order to minimize chip space and computation time, short fixed point representation of numbers is used. The data format of inputs that are values of pixels $p[x][y]$ is 8-bit integer. The data format of $\cos\theta$ and $\sin\theta$ is 16-bit fixed point number, which consists of 1-bit sign, 1-bit integer and 14-bit fraction based on two's complement. On the other hand, the data

formats of gradient magnitude G and gradient direction θ' are 12-bit and 8-bit integers, respectively. The data format of ρ is 10-bit two's complement integer. Since the range of the value of θ is 0 to 179, the data format of θ is 8-bit integer. The data format of the voted value is 24-bit integer.

3.2.4 Experimental Results

We have implemented the proposed architecture for the gradient-based Hough transform and evaluated it on the Xilinx Virtex-7 FPGA XC7VX485T-2. Table 3.3 shows the experimental results using Xilinx ISE 14.1.

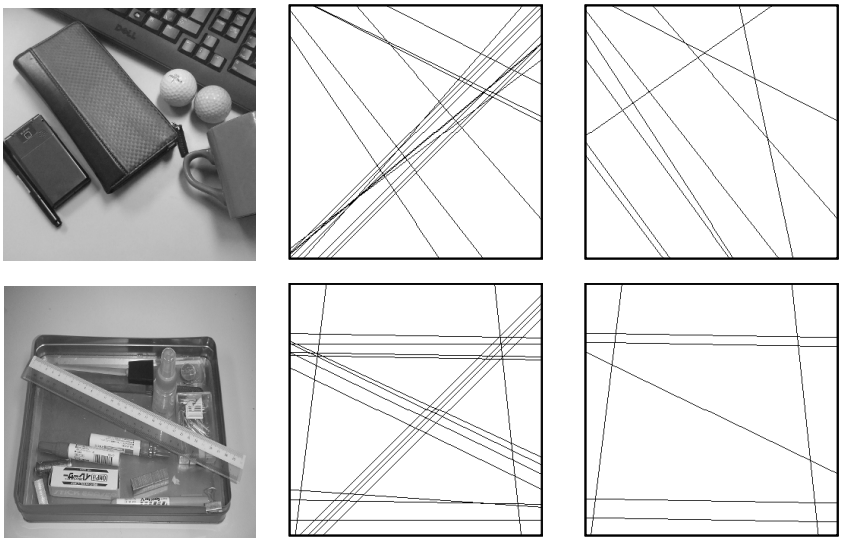
Table 3.3: Performance evaluation of the proposed architecture for the gradient-based Hough transform

DSP48E1 slices (out of 2800)	13 (1%)
36Kbit block RAMs (out of 1030)	180 (17%)
18Kbit block RAMs (out of 2060)	8 (1%)
Slices (out of 607200)	80181 (13%)
Clock frequency [MHz]	260.061

In our implementation, the voted range of the gradient-based Hough transform shown in Section 3.2.2 is set to $\lambda = 4$, that is for local gradient direction θ', we perform the voting operation to the range $\theta' - 4 \leq \theta \leq \theta' + 4$. The range was obtained by our experiments. The range is enough to extract lines because the error between the angle of lines obtained by the Sobel filter and the actual angle is small [7].

Figure 3.16 shows the result of lines detection for the conventional Hough transform and the gradient-based Hough transform. Compared with the result of the conventional

Hough transform, we can see that the gradient-based Hough transform obtained more correct lines and exclude the inexistent lines.



Original gray-scale image | Lines detection by the conventional Hough transform | Lines detection by the gradient-based Hough transform

Figure 3.16: Comparison between conventional and gradient-based Hough transform algorithms

In our implementation, the circuit can work in fully pipelined fashion. Namely, input pixels can be provided to the circuit clock by clock in raster scan order. To reduce the delay of the circuit, some pipeline registers are inserted into between circuit elements. It takes $2n + 44$ clock cycles to complete voting from the first input pixel is given to its voting is finished. Since the input image consists of n^2 pixels, the voting operations are performed in $n^2 + 2n + 44$ clock cycles. After voting, $\sqrt{2}n + 188$ clock cycles are necessary to output identified straight lines with 3×3 maximum filters. Therefore, in total, $n^2 + (\sqrt{2} + 2)n + 232$ clock cycles, i.e., $\frac{n^2+(\sqrt{2}+2)n+232}{260.061}\mu s$ are necessary to perform

the gradient-based Hough transform. If an input image of size 1000×1000 is given, our circuit can detect straight lines in $3.859ms$.

We have also implemented a software approach of the gradient-based Hough transform using GNU C. We have used Intel Xeon X7460 running in 2.66GHz and 128GB memory to run the sequential algorithm for the gradient-based Hough transform. For the image shown in Figure 3.10(a) whose size is 333×333, the software implementation can perform the gradient-based Hough transform in $133.519ms$. On the other hand, our circuit can perform it in $431.660\mu s$. Therefore, our FPGA implementation attains a speed-up factor of more than 309 over the sequential implementation on the CPU.

Table 3.4: Comparison with related works for Hough transform

	Deng [8]	Lee [30]	Karabernou [24]	Our work
Hough transform	Conventional	Conventional	Gradient-based	Gradient-based
Device	XC4010XL	Virtex 4	XC4010EPC84	XC7VX485T-2
Logic blocks	333 CLBs	314 CLBs	205 CLBs	80181 Slices
DSP slices	—	—	—	13 DSP48E1s
Frequency	40MHz	132MHz	23.166MHz	260.061MHz
Throughput	0.623Mpixel/s	32.768Mpixel/s	10.368Mpixel/s	263.979Mpixel/s

There are a number of literatures reported to implement Hough transform for lines using the FPGA shown in Section 3.1.1. Algorithms, that is conventional or gradient-based, and performances such as device, logic blocks, DSP slices, frequency and throughput are compared in Table 3.4. It is clear that the performance of our FPGA implementation is better than that of other works.

3.2.5 Concluding remraks

We have presented an efficient implementation of the gradient-based Hough transform for gray-scale images using DSP slices and block RAMs in the Virtex-7 Family FPGA. We have implemented the circuit using 13 DSP48E1 slices, 180 block RAMs with 36Kbits and 8 block RAMs with 18Kbits on the Virtex-7 Family FPGA XC7VX485T-2. The experimental results show that the architecture runs in 260.061MHz and for an $n \times n$ gray-scale image, our circuit can perform in $n^2 + (\sqrt{2} + 2)n + 232$ clock cycles, i.e., $\frac{n^2+(\sqrt{2}+2)n+232}{260.061}\mu s$, including the computation of gradient information.

3.3 Efficient implementation of the one-dimensional Hough transform algorithm for extracting circles

The Hough transform can be used to find circles in images. The conventional Hough transform for extracting circles needs three-dimension space, that is too costly. In this section, we show an efficient FPGA implementation of the Hough transform algorithm that uses only one-dimensional parameter spaces for circles detection on a Xilinx Virtex-7 FPGA. Our implementation uses 398 DSP48E1 slices and 309 block RAMs with 18kbits. The experimental results show that our implementation runs at 181.812MHz. For an edge image of size 400×400, our implementation attains a speed-up factor of approximately 189 over the sequential implementation on the CPU.

3.3.1 Introduction

The Hough transform defines a mapping from an image into a parameter space represented by an accumulate array. Let us consider circle detection using the Hough trans-

form. A circle can be defined by the three parameters, its center coordinate (x_c, y_c) and the radius r. Therefore, $O(N^3)$ space is necessary to store the parameter space, where n is the size of each dimension of the parameter space. Moreover, it takes $O(N^3)$-time to vote for each edge point and search intensive elements in the accumulate array. Recent FPGAs (Field Programmable Gate Arrays) have embedded DSP48E1 slices and block RAMs. The DSP slices are equipped with a multiplier, adders, logic operators, etc [58]. The block RAM is an embedded memory supporting synchronized read and write operations, and can be configured as a 36Kbit or two 18Kbit dual port RAMs [60]. The key technique for accelerating the algorithm is an efficient usage of DSP slices and block RAMs. However, in the conventional Hough transform algorithm for circles detection, even the state-of-the-art FPGA such as the Xilinx Virtex-7 series FPGAs can not handle the $O(N^3)$ space without off-chip memories. The parameter space decomposition is used to reduce the parameter space. Many of methods based on the Hough transform that use two-dimensional parameter spaces [22, 25] and one-dimensional parameter spaces [17] have been proposed. Specifically, in the one-dimensional Hough transform algorithm [17], the x-coordinate of center, y-coordinate of center, and radius are detected in series. In each detection, one-dimensional parameter spaces is used in the same way as the Hough transform. Moreover, various hardware algorithms for circle detection have been proposed. These existing researches use the template matching [43, 46] and the Hough transform algorithms [23, 16, 12]. Shafer *et al.* proposed an FPGA implementation to detect the iris position [43]. However, it detects only one circle in an image. Jen *et al.* proposed an FPGA implementation to detect circles using any three nonlinear pixels to form a circle [23]. However, because of the huge size of parameter spaces, this method uses off-chip memories. Elhossini *et al.* proposed a

pipelined FPGA architecture for circles detection [12]. Four specific radii are fixed due to the limitations of on-chip memories on the FPGA.

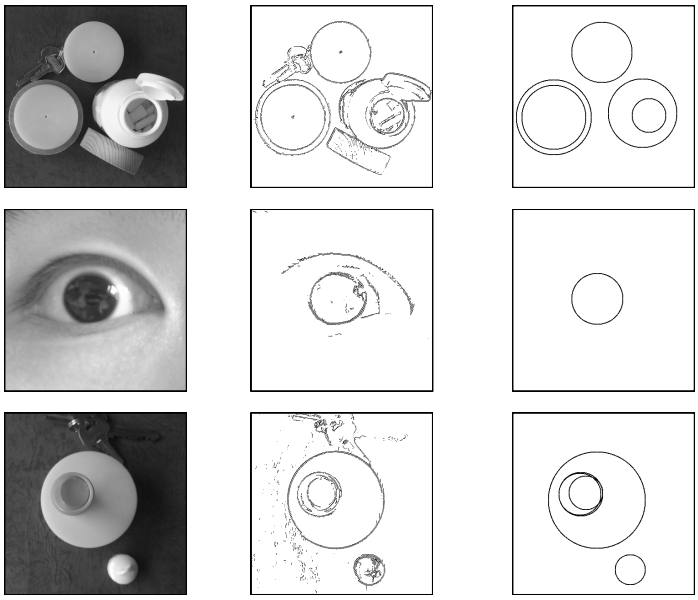
The fourth contribution of this chapter is to present an efficient FPGA implementation of the one-dimensional Hough transform algorithm for circles detection [17]. Our ideas include:

One-Dimensional Parameter Spaces: In the algorithm [17], since only one-dimensional parameter spaces are used, this algorithm is implemented using the block RAMs on the FPGA. Additional off-chip memories are not necessary.

Voting Space Partitioning: The parameter spaces for *x*- and *y*-coordinates of center candidates are partitioned into multiple block RAMs that are voted in parallel. The voting operations of radius for each center candidate is also concurrently performed using multiple block RAMs.

Efficient Usage of DSP slices: DSP slices are used to merge the partitioned voting spaces for *x*- and *y*-coordinates of center candidates in a pipelined fashion. Furthermore, DSP slices are used to compute the Euclidean distance between each center candidate and edge points.

The one-dimensional Hough transform algorithm consists of the following four steps: (1) *x*-coordinates of center candidates of circles are detected by voting midpoints of every two edge points in each row. (2) *y*-coordinates of center candidates of circles are detected by voting midpoints of every two edge points in each column. (3) Center candidates are listed from *x*- and *y*-coordinates of center candidates. (4) For each center candidate, radii of the circles are detected. Also, detected circle candidates are checked whether the candidate is a true circle by voting the Euclidean distances between each center candidate and every edge point.



(a) Input image (b) Binary edge image (c) Detected circles

Figure 3.17: Example of circles detection using the one-dimensional Hough transform algorithm

Figure 3.17 shows an example of circles detection using this method. For an input image (Figure 3.17(a)), its edge image (Figure 3.17(b)) is obtained using the edge detector such as Canny edge detector [6]. Figure 3.17(c) draws detected circles using the one-dimensional Hough transform algorithm.

We have implemented the one-dimensional Hough transform algorithm on a Xilinx Virtex-7 XC7VX485T-2. Our new architecture uses 398 DSP48E1 slices and 309 block RAMs with 18Kbits. Our proposed circuit runs in 181.812MHz. For a binary image of size 400×400, our circuit performs the circles detection in at most 970434 clock cycles, i.e., $5337.568\mu s$. Our implementation attains a speed-up factor of approximately 189 over the sequential implementation on the CPU.

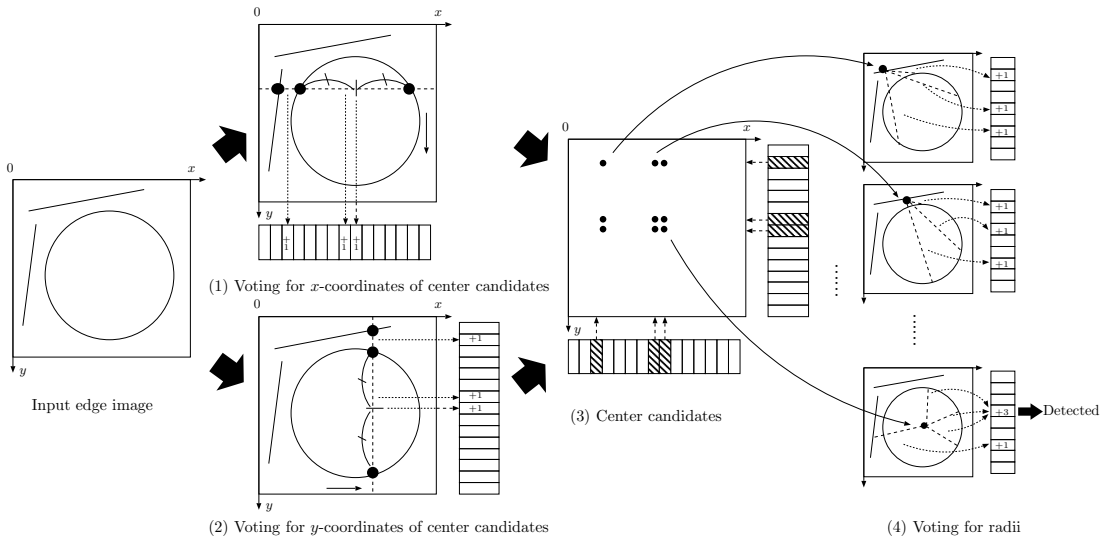


Figure 3.18: The outline of the one-dimensional Hough transform algorithm for circle detection

3.3.2 One-dimensional Hough transform algorithm for circles detection

The main purpose of this section is to show the one-dimensional Hough transform algorithm for circles detection. Figure 3.18 illustrates an outline of this algorithm. The detail of this algorithm is shown as follows.

Step 1: We compute midpoints of every two edge points on each row. After that, the x-coordinates are voted to a one-dimensional accumulate array. Namely, the element that corresponds to the x-coordinate of each midpoint is incremented by one. This operation is performed for every row in the input image. If there is a circle, the voting to the x-coordinate of its center is concentrated since a circle is symmetrical to the vertical bisector through its center. To cope with quantization error, after voting, we use a maximum filter for the voted values. In here, for each value in the accumulate array,

this filter copies the value if it is the maximum within a local range, otherwise, the filter outputs zero. After that, the top λ largest elements are extracted as x-coordinates of center candidates of circles to detect multiple circles.

Step 2: In the same way as Step 1, we compute midpoints of every two edge points on each column. After that, the y-coordinates are voted to a one-dimensional accumulate array for every column. If there is a circle, the voting to the y-coordinate of its center is concentrated since a circle is symmetrical to the horizontal bisector through its center. The voted values are filtered by the maximum filter. After filtering, the top λ largest elements are obtained as y-coordinates of center candidates of circles.

In the following FPGA implementation, we perform Steps 1 and 2 in parallel since these steps can be executed independently.

Step 3: We list center candidates that are all combinations from x- and y-coordinates obtained in the above steps. Since each step obtains λ coordinates, λ^2 center candidates are listed in total.

Step 4: For each center candidate, the Euclidean distances between the center candidate and all edge points are computed and the distances are voted to a one-dimensional accumulate array as radii of circles. If a center candidate is a center of true circle in the image, an element that corresponds to the radius is intensely voted. Therefore, we verify whether the center candidate and each radius represent a true circle using the voted value. In digital images, it is known that the number of pixels on the circumference of circles with radius r is $4\sqrt{2}r$ [29]. However, practical circles may be broken or disturbed. Therefore, for each radius, we determine whether the center candidate and its radius represent a true circle using the threshold $f \times 4\sqrt{2}r$, where f is a threshold factor that is a constant value within the range $(0, 1]$. If the voted value of radius is larger than

the threshold, the circle is verified as a true circle. The validation for each radius, circles with the center candidate are extracted.

For each center candidate, the above operation is performed. In our FPGA implementation, this operation is performed in parallel and circles are detected.

3.3.3 FPGA architecture for the one-dimensional Hough transform

This section describes our FPGA architecture of the one-dimensional Hough transform algorithm. The input data for our implementation is given as two edge lists consisting of coordinates of edge pixels in row- and column-major order, respectively. These edge lists can be obtained by edge detection easily, and each list is stored into a block RAM.

Structure for voting operation of center candidates

Figure 3.19 shows our architecture for the voting operation of x-coordinates of center candidates, where p is the number of voting modules. Namely, up to p voting operations are concurrently performed. We utilize one series of p shift-registers that transfer data in the left-to-right direction. In order to compute the midpoints of any two edge points on each row, we read them from the block RAM that stores the edge list in row-major order. To give all pairs of x-coordinates of which y-coordinates are identical for each row, we input x-coordinates which have the same y-coordinate to the register and transferred them with shift-registers one by one. If all or p x-coordinates which have the same y-coordinate are transferred to the registers, then all x-coordinates which have the same y-coordinate are continuously read out from the block RAM and broadcast to pair with the x-coordinates in the registers. If the number of edge points whose y-coordinates are identical is larger than p, the above operation is repeated. For each given

pair of x-coordinates, the coordinate of the midpoint is computed using an adder and a 1-bit right-shifter that divides by two.

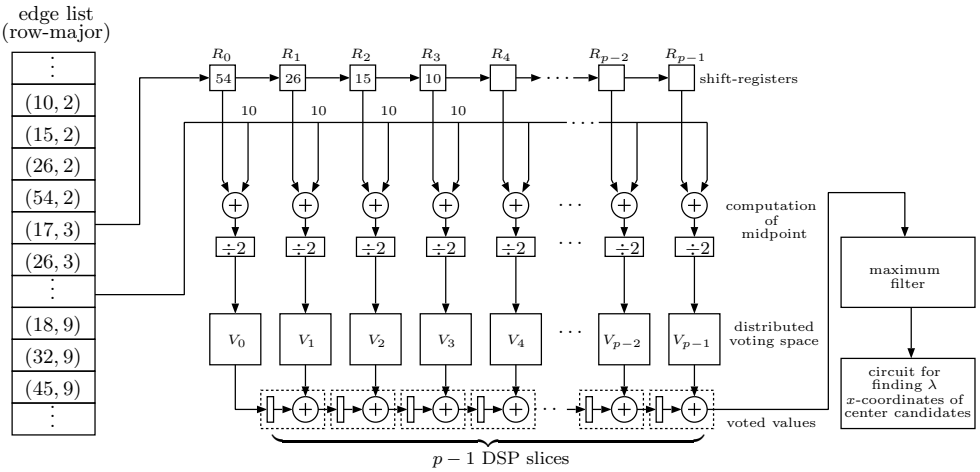


Figure 3.19: Architecture of voting for x-coordinates of center candidates

After that, the x-coordinates of midpoints are voted to the block RAM. Namely, an element that corresponds to the x-coordinate in the block RAM is incremented by one. We use p block RAMs V_k ($0 \leq k \leq p-1$) to store the voted values and at most p midpoints are concurrently voted. Figure 3.20 illustrates the architecture of V_k using a block RAM, that is the same with the architecture shown in Figure 3.6. The block RAM is utilized in dual port mode, where port set A and B are operated for read and write operation, respectively.

In the block RAMs on the target device of this work, read/write operations can be configured as either RF (Read First) mode or WF (Write First) mode. As mentioned in Section 2.3, in the dual port mode, there is a restriction that if read and write operation to the same address are performed for each port, the setting of block RAMs must be RF [60].

We use the block RAM to store the values of $v_k[x]$ ($0 \leq x < n$), where the size of

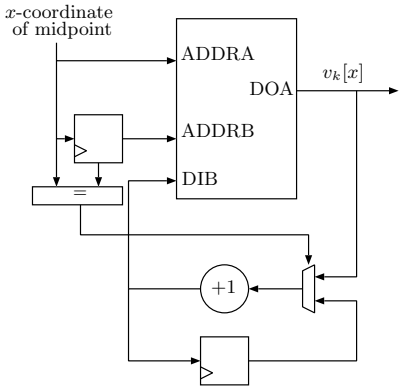


Figure 3.20: A block RAM to store the voted values

image is $n \times n$ and x is an x-coordinate of midpoints. Let $v_k[x]$ denote a data of address x in the block RAM V_k. Since x is given to its *ADDRA*, $v_k[x]$ is output from *DOA* after the rising clock edge as illustrated before. After that, $v_k[x] + 1$ is computed and it is given to *DOB*. Since x is given to *ADDRB*, $v_k[x] + 1$ is written in $v_k[x]$. According to the restriction stated in the above, since the same value of x may be input continuously, we used an additional register to store the latest voted value and if the same value of x is input continuously, the stored value is used instead of the value read from the block RAM.

After the voting operations, we combine the voted values stored in p block RAMs. We read every element in each block RAM one by one. These values are added and transferred left-to-right for each clock cycle to compute the sum of each element with $p - 1$ registers and $p - 1$ adders. To optimize the circuit resources, we use a cascaded DSP slice for each pair of register and adder.

Structure for finding the center candidates of circles

A maximum filter is used to cope with the quantization error for the voted values

above. After filtering, the x-coordinates to which the λ largest values correspond are obtained as x-coordinates of center candidates of circles. Each voted value is input to maximum filter for each clock cycle. For each value, the filter verifies whether it is the maximum comparing with its neighboring 2 values. If it is the maximum in the local range, the filter copies the largest value, otherwise, the filter outputs it as zero. Therefore, the largest value in the local range and zero are alternately output after filtering.

Figure 3.21 illustrates the structure that finds the x-coordinates of center candidates. An array of registers and comparators are used to obtain the largest λ values. Every register is initialized to zero. The filtered values are continuously input to the left-most register for each clock cycle. Each register of c_i ($0 \leq i < \lambda$) compares the value with its left register. If this register has smaller value, the value of its left register is then transferred to it. If this register has larger value, it will compare the value with its right register, and if the register has larger value, this value is transferred to its right neighboring register. All the values of registers are transferred in parallel. Since input values are obtained through the maximum filter, the input values are given at more than one clock cycle intervals. Hence, the larger values will be gradually transferred to the right side through the registers. Finally, the top λ largest values are stored in the registers. Namely, the x-coordinates that have the top λ largest values are chosen to be the x-coordinates of center candidates of circles.

Similarly, using another circuit whose structure is the same as the above, the voting operation for y-coordinates of center candidates are performed with edge lists stored in column-major order as input. Also, y-coordinates with the top λ largest values are obtained as the y-coordinates of center candidates. Every x- and y-coordinates of centers candidates are combined to construct λ^2 center candidates.

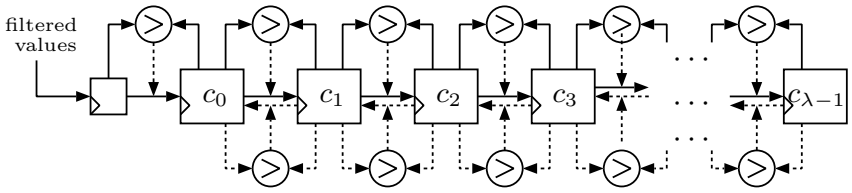


Figure 3.21: Architecture for finding λ x-coordinates of center candidates

Structure for the voting operation of radius

The voting operation of radius is performed for each center candidate in parallel (Figure 3.22). Since λ^2 center candidates are obtained, we use λ^2 voting spaces to store the voted values. For each center candidate, the Euclidean distances between the center candidate and all edge points are computed. The circuit runs in a pipeline fashion for coordinates of edge points that are given for each clock cycle. The computation of the Euclidean distances include the computation of the square root. Since it is difficult to compute them directly on the circuit, we use the CORDIC IP provided by Xilinx that provides a hardware module that is fully pipelined architecture [57]. Since the computed distances are voted to a block RAM for each center candidate, totally, λ^2 block RAMs M_k ($0 \leq k < \lambda^2$) are used. The architecture of M_k is the same as shown in Figure 3.20.

Structure for verifying true circles

Finally, we verify whether the center candidates are true circles. If a center candidate is that of true circle in the image, an element of the block RAM that corresponds to the radius is intensely voted. A maximum filter is also used to cope with the quantization error for the voted values. If the value is the maximum in a local range, the filter copies

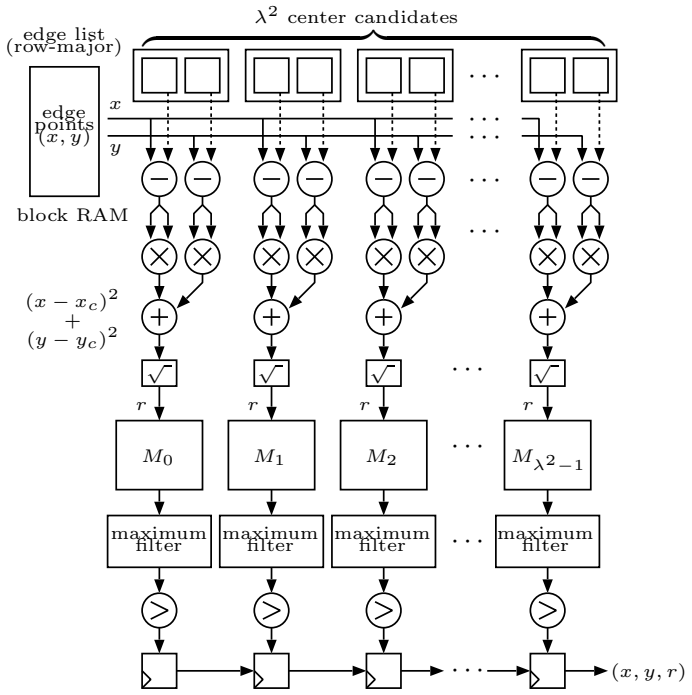


Figure 3.22: Architecture of voting for radius

the largest value, otherwise, the filter output it as zero. After filtering, each radius is verified whether it is the radius of a true circle by comparing its voted value with the threshold $f \times 4\sqrt{2}r$ (Section 3.3.2). All the values of $f \times 4\sqrt{2}r$ are precomputed to store in a block RAM that is used as a Look-up-table. If it is larger than the threshold, the center candidate and the radius that represent a circle is input to the shift registers and output through the registers.

3.3.4 Experimental results

We have implemented the proposed architecture for circles detection and evaluated it on the Xilinx Virtex-7 FPGA XC7VX485T-2 [55]. For our implementation, 398 DSP48E1 slices, 309 block RAMs with 18Kbit and 20452 slices of the FPGA are used. The FPGA

with the architecture proposed in this paper works in 181.812MHz.

In our implementation, the x- and y-coordinates of edge list stored in row- or column-major are 9-bit integer. The voted values of x- and y-coordinate of center candidates are set to be 17-bit integer. The number p of voting modules for x- and y-coordinates of center candidates is set to be 100. The number λ of x- or y-coordinates of center candidates is set to be 10, therefore, 100 center candidates are constructed. The data format of the voted values for radius is 13-bit integer.

Since the latency of our architecture depends on the input image, we suppose that all pixels of input image of size $n \times n$ are edge points. Let p be the number of voting modules for x- or y-coordinates of center candidates and λ be the number of x- or y-coordinates of center candidates. For simplicity, we assume that n is a multiple of p. For a certain row or column i, it takes $\frac{n}{p}(p + n + 6)$ clock cycles to complete the voting operations of row or column i. Therefore, it takes $\frac{n^2}{p}(p + n + 6)$ clock cycles to complete the voting operations for all rows or columns. After that, it takes $p + n + \lambda + 4$ clock cycles to find x- or y-coordinate of center candidates. The combination of every x- and y-coordinates of center candidates takes 1 clock cycle. The voting operation for radius takes $n^2 + 11$ clock cycles, and it takes $\frac{n}{2} + \lambda^2 + 8$ clock cycles to output all detected true circles. Finally, our circuit totally takes $\frac{n^3}{p} + \frac{(2p+6)n^2}{p} + \frac{3}{2}n + p + \lambda^2 + \lambda + 24$ clock cycles to implement the one-dimensional Hough transform.

For estimating the speed up of our FPGA implementation, we have also implemented a software approach of the one-dimensional Hough transform using GNU C. We have used Intel Xeon X7460 running in 2.66GHz and 128GB memory to run the sequential one-dimensional Hough transform algorithm. For the image above, the software implementation can perform the one-dimensional Hough transform for circles de-

tection in 1008.658ms. On the other hand, our circuit can perform it in 970434 clock cycles, i.e., 5337.568μs. Therefore, our FPGA implementation attains a speed-up factor of approximately 189 over the sequential implementation on the CPU.

Table 3.5: Comparison with related works for Hough transform

	Shafer [43]	Tokunaga [46]	Jen [23]
Base algorithm	Template matching	Template matching	Hough transform
Device	Altera EP4SGX530	Xilinx XC4025E	Altera Stratix 1S25
Memory	Int. 6.75Mbit	—	Int. 1.6Mbit & Ext
Frequency	159MHz	—	—
Throughput	9.362Mpixel/s	0.0512Mpixel/s	0.01524Mpixel/s

	Geninatti [16]	Elhossini [12]	Our work
Base algorithm	Hough transform	Template matching	Hough transform
Device	Xilinx Spartan 3	Xilinx Virtex-4	Xlinix XC7VX485T-2
Memory	Ext. 1Mbit	Int. 256Kbit	Int. 5.4Mbit
Frequency	—	27MHz	181.812MHz
Throughput	12.32Mpixel/s	14.4Mpixel/s	29.976Mpixel/s

There are a number of literatures reported to implement circles detection using the FPGA shown in Table 3.5, where Int. means internal (on-chip) and Ext. means external (off-chip). It is difficult to directly compare to other works because utilized FPGAs and supported size of images differ. Considering the throughput, our implementation compares favorably with other works. The deficiencies of these existing researches such as detecting only one circle or using off-chip memories are not existing in our implementation. Our implementation detects multiple circles with variable radii using

only the block RAMs on the FPGA.

3.3.5 Concluding remarks

We have presented an efficient implementation of the one-dimensional Hough transform using 398 DSP slices, 309 block RAMs with 18Kbits on the Virtex-7 Family FPGA XC7VX485T-2. The architecture runs in 181.812MHz and for an image of size 400×400 that all pixels are edge points, our circuit performs the one-dimensional Hough transform in 970434 clock cycles, i.e., 5337.568μs which theoretically attains a speed-up factor of approximately 189 over the sequential implementation on the CPU.

Chapter 4

Implementation of the Euclidean algorithm on the FPGA

In this chapter, we show a processor core that executes Euclidean algorithm computing the GCD (Greatest Common Divisor) of two large numbers in an FPGA. The proposed processor core uses only one DSP slice and one block RAM, that is called GCD processor core. Since the proposed GCD processor core is compactly designed based on FDFM approach (Few DSP slices and Few block Memories) and uses very few resources, we have succeeded in implementing 1280 GCD processor cores in a Xilinx Virtex-7 family FPGA XC7VX485T-2. The experimental results show that the performance of this FPGA implementation using 1280 GCD processor cores is 0.0904μs per one GCD computation for two 1024-bit integers. Quite surprisingly, it is 3.8 times faster than the best GPU implementation and 316 times faster than a sequential implementation on the Intel Xeon CPU.

4.1 Introduction

The GCD (Greatest Common Divisor) computation is widely used in computer systems for cryptography, data security and other important algorithms. Most of the time of these computer systems is consumed for computing the GCDs of very large integers. Therefore, it is an important task of accelerating the GCD computation. However, arithmetic operations on integers numbers exceeding 64 bits cannot be performed directly by a conventional 64-bit CPUs as its instruction set support integers of at most 64-bit in length. It is an efficient way to implement the arithmetic operations on large integers using hardware device such as FPGA, VLSI or GPU.

Recent FPGAs have embedded DSP48E1 slices and block RAMs. The Xilinx Virtex-7 series FPGAs have DSP slices equipped with a multiplier, adders, logic operators, etc [58]. The DSP slice also has pipeline registers between operators to reduce the propagation time. The block RAM is an embedded memory supporting synchronized read and write operations, and can be configured as a 36Kbit or two 18Kbit dual port RAMs [60]. They are widely used in consumer and industrial products for accelerating processor intensive algorithms [35, 36, 3, 18]. Since the continuing decline in the ratio of FPGA price to performance and its programmable features, FPGA is suitable for a hardware implementation of general purpose computing. The main contribution of this chapter is to present an efficient processor core that executes the Euclidean algorithm computing the GCD of two large integers using an FPGA. The proposed processor core is designed based on the *FDFM (Few DSP slices and Few block Memories) approach* [2]. The key idea of the FDFM approach is to use few DSP slices and few block RAMs for constituting a processor core. We must note that the FDFM approach has some advantages. First, despite the main circuit occupies most of hardware resources

of the FPGA, we can also implement the necessary hardware algorithm in the FPGA using remaining few resources as shown in Figure 4.1 (1). On the other hand, we can implement multiple FDFM processors working in parallel if enough hardware resources are available as illustrated in Figure 4.1 (2). In this paper, we also employ the FDFM approach to implement parallel GCD computation in the FPGA. For example, in this paper, we propose a processor core for GCD computation of 1024-bit, 2048-bit, 4096-bit, and 8192-bit integers, that uses only one DSP slice and one block RAM. We implement one processor core in the FPGA, and the frequency of the FPGA is over 380MHz, that is extremely high. If only one proposed GCD processor core is implemented in the FPGA for computing one GCD of 1024-bit, 2048-bit, 4096-bit, and 8192-bit integers, it takes 73.12μs, 253.35μs, 915.78μs, and 3614.91μs, respectively. In other words, single GCD processor core has competitive performance. Since the proposed GCD processor core uses very few resources of FPGA, we can implement more than one thousand identical processor cores in an FPGA, that all processor core work are paralleled to execute bulk GCD computation. The pairwise GCD computation that computes all pairs of integers in a set, can be used to evaluate the performance of the implementation of thousand processor cores.

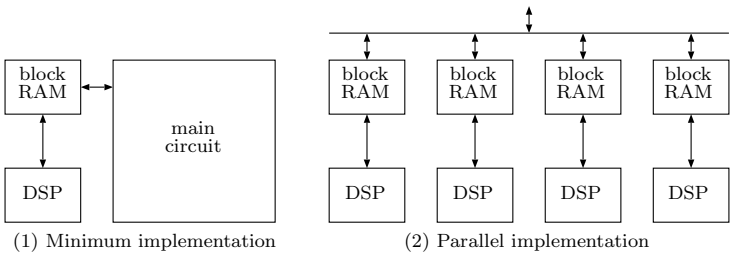


Figure 4.1: Advantages of our FDFM approach

One of the applications for benchmarking pairwise GCD computation is breaking

weak RSA keys. RSA [41] is one of the most well-known public-key cryptosystems widely used for secure data transfer. RSA cryptosystem has an encryption key open to the public. An encryption key includes a modulus n called *an RSA modulus* such that $n = pq$ for two distinct large prime numbers p and q. If the values of p and q are available, the encrypted message can be easily converted to the original message. Thus, the safety of RSA cryptosystem relies on the difficulty of factoring RSA modulus n of two large prime numbers p and q. Suppose that we have a set of many RSA encryption keys collected from the Web. If some of RSA moduli in encryption keys are generated by inappropriate implementation of a random prime number generator, they may reuse the same prime number. We call the keys sharing a prime number as *weak RSA keys*. If two RSA moduli share a prime number, they can be decomposed by computing the GCD of these two moduli. It is well known that the GCD can be computed very easily by Euclidean algorithms [27]. Hence, we can compute the GCDs of all pairs of RSA moduli in the Web to find the RSA keys that sharing the same prime number. In this paper, pairwise GCD computation for RSA moduli is used to measure the performance of the proposed GCD processor core based on FDFM approach. We have succeeded in implementing 1792 GCD processor cores in a Xilinx Virtex-7 family FPGA XC7VX485T-2. However, when the circuit of 1792 GCD processor cores is operated on the FPGA device, this circuit becomes unstable because the number of used resources of FPGA is too close to the maximum available resourses. Finally, we implement 1280 GCD processor cores in the FPGA, that compute the GCDs of all pairs of RSA moduli that are stored in an off-chip DDR3 memory MT8JTF12864HZ-1G6G1. Our implementation of 1280 GCD processor cores computes one GCD of two 1024-bit RSA moduli in expected 0.0904μs.

Several hardware implementations for computing the GCD on FPGAs have been presented [9, 28]. However, they just implemented Binary Euclidean algorithm to compute the GCD using programmable logic blocks as it is. Hence, they can support the GCD computation for numbers with very few bits. On the other hand, several previously published papers have presented GPU implementations of Binary Euclidean algorithm in CUDA-enabled GPUs. Fujimoto [13] has implemented Binary Euclidean algorithm using CUDA and evaluated the performance on GeForce GTX285 GPU. The experimental results show that the GCDs for 131072 pairs of 1024-bit numbers can be computed in 1.431932 seconds. Hence, his implementation runs 10.9μs per one 1024-bit GCD computation. Scharfglass *et al.* [42] have presented a GPU implementation of Binary Euclidean algorithm. It performs the GCD computation of all 199990000 pairs of 20000 RSA moduli with 1024 bits in 2005.09 seconds using GeForce GTX 480 GPU. Thus, their implementation performs each 1024-bit GCD computation in 10.02μs. Later, White [48] has showed that the same computation can be performed in 63.0 seconds on Tesla K20Xm. It follows that it computes each 1024-bit GCD in 3.15μs. Quite recently, Fujita *et al.* have presented new Euclidean algorithm called Approximate Euclidean algorithm and implemented it in the GPU [14]. Approximate Euclidean algorithm performs perform each 1024-bit GCD computation in 0.346μs on GeForce GTX 780Ti and 28.6μs on Intel Xeon X7460 (2.66GHz) CPU. Our implementation of 1280 GCD processor cores in Xilinx VC707 evaluation board [61] equipped with FPGA XC7VX485T-2 performs one 1024-bit GCD computation in 0.0904μs which is 3.8 times faster than the GPU and 316 times faster than the CPU.

4.2 Euclidean algorithms for computing GCD

This section review classical Euclidean algorithm and Fast Binary Euclidean algorithm for computing the GCD of two numbers X and Y. We then show Hardware Binary Euclidean algorithm by modifying Fast Binary Euclidean algorithm, that is implemented in an FPGA.

Let $\texttt{GCD}(X,Y)$ denote the GCD of X and Y. For any odd integer X and even integer Y, $\texttt{GCD}(X,Y) = \texttt{GCD}(X,\frac{Y}{2})$ holds. Also, for any even integers X and Y, $\texttt{GCD}(X,Y) = 2 \times \texttt{GCD}(\frac{X}{2},\frac{Y}{2})$ holds, and so we can obtain a factor of 2 in the GCD of X and Y very easily.

For simplicity, we assume that both inputs X and Y are odd and $X \geq Y$ holds. Based on the fact, it should have no difficulty to modify all GCD algorithms shown in this paper to handle even input numbers. Let $\texttt{swap}(X,Y)$ denote a function to exchange the values of X and Y. We can write a standard Euclidean algorithm for computing the GCD of X and Y as follows:

[Original Euclidean algorithm]

```
gcd(X,Y){
    do {
        X ← X mod Y; //X < Y always holds
        swap(X,Y); //X > Y always holds
    } while(Y ≠ 0)
    return(X);
}
```

Since $X \geq Y$ holds, modulo computation is performed and X will store the value of X mod Y, which is less than Y. After that, $\texttt{swap}(X,Y)$ is executed and $X > Y$ always

holds. The same operation is repeated until $Y = 0$ and X stores the GCD of input integers X and Y. However, modulo computation used in Original Euclidean algorithm is costly. So, Binary Euclidean algorithm which does not execute it, is often used to compute the GCD efficiently:

[Binary Euclidean algorithm]

gcd(X,Y){

    do {

        if(X is even) $X \leftarrow \frac{X}{2}$;

        else if(Y is even) $Y \leftarrow \frac{Y}{2}$;

        else $X \leftarrow \frac{X-Y}{2}$;

        if($X < Y$) `swap`(X, Y);

    } while($Y \neq 0$)

    return(X);

}

If X (or Y) is even, then X (or Y) is halved to remove the least significant bit of X (or Y) which is 0. If both X and Y are odd, X-Y is computed. Since result of subtraction of two odd numbers is even, $\frac{X-Y}{2}$ is performed to remove the least significant bit of $X - Y$, then X will store the value of $\frac{X-Y}{2}$. If $X < Y$ holds, `swap`(X, Y) is performed, then $X \geq Y$ always holds. Note that the Binary Euclidean algorithm removes one 0 bit from the least significant bit of X (or Y) and $\frac{X-Y}{2}$ in each iteration of the do-while loop. We can reduce the number of iterations of the do-while loop by removing consecutive 0 bits. Let `rshift`(X) be a function returning the number obtained by removing consecutive 0 bits from the least significant bit of X. For example, if $X = 11010100$ in binary system, then `rshift`(X) = 110101 in binary notation. Using swap and rshift functions, we can write the Fast Binary Euclidean algorithm as follows:

[Fast Binary Euclidean algorithm]

gcd(X,Y){

do {

$X \leftarrow$ `rshift`$(X - Y)$;

if$(X < Y)$ `swap`(X, Y);

} while $(Y \neq 0)$

return(X);

}

In each iteration of the do-while loop, at least one 0 bit is removed from X (or Y). Hence, for any input numbers, the number of iteration of the do-while loop in Fast Binary Euclidean algorithm is no larger than that in the Binary Euclidean algorithm. However, we need to read all bits of X and Y to exchange them if we implement function `swap` as it is. Also, `rshift` function needs a large barrel shifter. Hence, we should avoid direct implementations of these functions in the FPGA. Instead of function `rshift`(X), we implement function `rshift`$_k(X)$, which removes at most k consecutive 0 bits from the least significant bit of X. In other words, if X has at most k consecutive 0 bits from the least significant bit, all of them can be removed in one iteration of do-while loop by executing `rshift`$_k(X)$. If X has more than k consecutive 0 bits, then k 0 bits from the least significant bit are removed, and `rshift`$_k(X)$ is repeated until X is odd. For example, `rshift`$_2$(1101,1000)=11,0110 and `rshift`$_2$(1101,1010)=110,1101 hold. Using `rshift`$_k$, we can describe the Hardware Euclidean-based GCD algorithm as follows:

[Hardware Binary Euclidean algorithm]

gcd(X,Y){

do {

if(X is even) $X \leftarrow$`rshift`$_k(X)$;

```
        else if (Y is even) Y ←rshift_k(Y);
        else if(X ≥ Y) X ←rshift_k(X − Y);
        else Y ←rshift_k(Y − X); // X < Y
    } while(X ≠ 0 and Y ≠ 0)
    if(X ≠ 0) return(X);
    else return(Y);
}
```

Note that operation $\texttt{rshift}_k$ may return an even number. Hence, one of X or Y can be an even number. If this is the case, either $X \leftarrow \texttt{rshift}_k(X)$ or $Y \leftarrow \texttt{rshift}_k(Y)$ is executed until both of them are odd. Hence, both X and Y are odd, whenever $\texttt{rshift}_k(X - Y)$ is executed. Thus, the argument of $\texttt{rshift}_k$ is always even and the least significant bit is 0 when it is executed.

Table 4.1 shows the average number of iterations of the do-while loop 1024-bit RSA moduli for each values of k of $\texttt{rshift}_k$. Note that $k = \infty$ corresponds to Fast Binary Euclidean algorithm, which performs `rshift` function that removes all consecutive 0 bits. Clearly, the number of iterations is smaller for large k. In our implementation, we use a multiplier embedded in DSP slice to compute $\texttt{rshift}_k$ for $k = 17$ instead of using logic resources of the FPGA. Hence, we can reduce the number of CLBs and implement more GCD processor cores in an FPGA. Since the subtraction of two very large numbers X and Y returning a result which has more than 17 consecutive 0 bits from the least significant bit is a very rare case, $\texttt{rshift}_{17}$ of our implementation and ideal $\texttt{rshift}_k(k = \infty)$ of the Fast Binary Euclidean algorithm has almost the same number of iterations as shown in the table.

Table 4.1: The average number of iterations of the do-while loop for 1024-bit RSA moduli

	Hardware Binary Euclidean									Fast Binary
k	1	2	3	4	5	6	7	8	17	Euclidean
number of iterations	1445.8	964.3	827.0	772.0	747.1	735.3	729.7	726.8	724.0	723.9

4.3 A GCD processor core for large integers

This section shows a *GCD processor core*, which computes the GCD of two very large numbers based on the Hardware Binary Euclidean algorithm. Our GCD processor uses only one 18k-bit block RAM and one DSP slice in the FPGA. The 18k-bit block RAM is configured as a simple dual-port memory [60] with ports A and B of width 36 bits and 18 bits, respectively. Figure 4.2 illustrate the configuration of the 18k-bit block RAM used in our GCD processor core. Two large numbers X and Y of Hardware Binary Euclidean algorithm are stored as 18-bit words. If each of them has 1024 bits, it is stored in $\lceil \frac{1024}{18} \rceil = 57$ words. Let $X_{56}X_{55}\cdots X_0$ denote 57 words representing X such that $X = \sum_{i=0}^{56} X_i \times 2^{18i}$ holds. Similarly, let $Y_{56}Y_{55}\cdots Y_0$ denote the words representing Y. Since the operations $\texttt{rshift}_{17}(X - Y)$ and $\texttt{rshift}_{17}(Y - X)$ of Hardware Binary Euclidean algorithm are executed for computing the GCD of X and Y, we want to read X and Y simultaneously. Hence, port A of the block RAM is configured as read-only 36-bit mode. On the other hand, since the result of operation $\texttt{rshift}_{17}(X - Y)$ (or $\texttt{rshift}_{17}(Y - X)$) is overwritten to one of X or Y, the port B is configured as write-only 18-bit mode.

Reading: Since port A of the block RAM is configured as read-only 36-bit mode, the block RAM is a 512×36-bit memory for port A. We can read 36-bit data $X_iY_i(0 \leq$

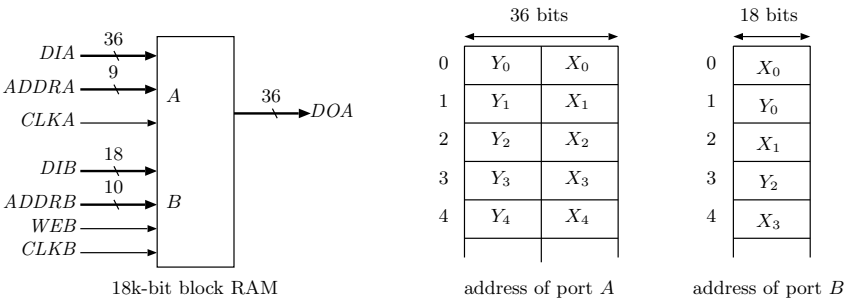


Figure 4.2: A 18k-bit block RAM and the memory configuration

$i \leq 56$) from address i using port A for performing the operation $\texttt{rshift}_{17}(X - Y)$ (or $\texttt{rshift}_{17}(Y - X)$).

Writing: Since port B is configured as write-only 18-bit mode, the block RAM is a 1024×18-bit memory for port B. We can write 18-bit data X_i in address $2i$ or and Y_i in address $2i + 1$ ($0 \leq i \leq 56$) using port B. In other words, the result of $\texttt{rshift}_{17}(X - Y)$ (or $\texttt{rshift}_{17}(Y - X)$) can be overwitten to X (or Y).

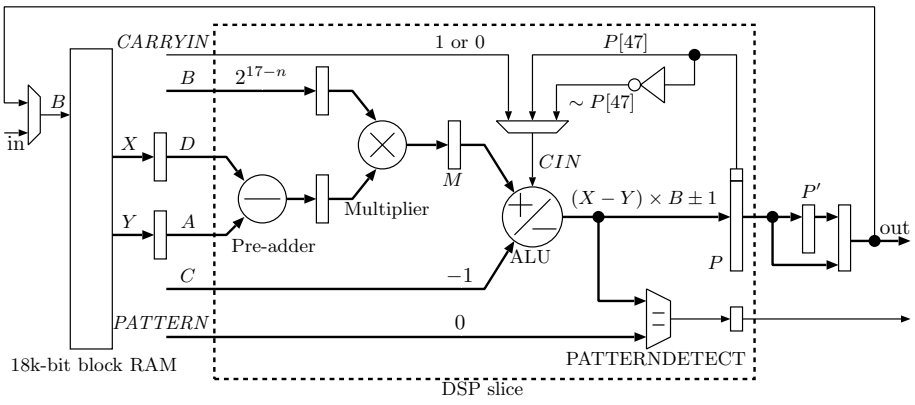


Figure 4.3: The architecture of a GCD processor

The DSP slice in our GCD processor core uses a pre-adder, a multiplier and a three input ALU (Arithmetic Logic Unit) as illustrated in Figure 4.3. Suppose that $X \geq Y$ holds. We briefly show how to use the DSP slice for executing the operation

$\texttt{rshift}_{17}(X - Y)$ of Hardware Binary Euclidean algorithm. The 36-bit data $X_iY_i(0 \leq i \leq 56)$ is read from the block RAM one by one, and is connected to the pre-adder of DSP slice. The operation $X - Y$ needs to be executed from the least significant bit of large numbers X and Y. Thus, the pre-adder is used to compute $X_i - Y_i$ for each 36-bit data X_iY_i one by one from X_0Y_0. Since $X_i - Y_i$ is computed one by one, if $X_0 - Y_0 < 0$ holds, we need to borrow from the higher bit which is in the next word $X_1 - Y_1$. In other words, $X_1 - Y_1 - 1$ needs to be computed for 36-bit data X_1Y_1, and we call -1 *borrow*. Let b_0 denote the borrow from $X_0 - Y_0$, and let $b_i(1 \leq i \leq 55)$ denote the borrow from $X_i - Y_i - b_{i-1}$. We note that $X_0 - Y_0$ needs to be computed for X_0Y_0, and $X_i - Y_i - b_{i-1}$ needs to be computed for $X_iY_i(1 \leq i \leq 56)$. However, we can not compute the borrow using the pre-adder because it has only two input ports. Thus, we first perform the shift operation to remove the consecutive 0 bits from the least significant bit using the multiplier. The multiplier performs the operation $(X_i - Y_i) \times 2^{17-n}$ for each $X_i - Y_i(0 \leq i \leq 56)$ one by one, where $n(1 \leq n \leq 17)$ is the number of consecutive 0 bits from the least significant bit of $X - Y$. If $X - Y$ has more than 17 consecutive 0 bits from the least significant bit, n has the value 17. For example, if $X_0 - Y_0 = (11, 0010, 1\underline{000}, \underline{0000}, \underline{0000})$, that is, $X - Y$ has 11 consecutive 0 bits from the least significant bit. The multiplier computes $(X_0 - Y_0) \times 2^{17-11} = (1100, 101\underline{0}, \underline{0000}, \underline{0000}, \underline{00}00, 0000)$. We note that the 11 bits consecutive 0 bits are on the right of the 17-th bit of $(X_0 - Y_0) \times 2^6$. For other $n(1 \leq n \leq 17)$, the n bits consecutive 0 bits are also on the right of the 17-th bit of $(X_0 - Y_0) \times 2^{17-n}$. We use this feature to remove the consecutive 0 bits in the following. Next, since we suppose that $X \geq Y$ holds, ALU computes $(X_i - Y_i) \times 2^{17-n} - b_{i-1}$. Otherwise, if $X < Y$ holds, ALU can also compute $-(X_i - Y_i) \times 2^{17-n} - b_{i-1} = (Y_i - X_i) \times 2^{17-n} - b_{i-1}$. In other words, the computation of $(X - Y)$ and $(Y - X)$ can be switched dynami-

cally by controlling the behavior of ALU, and the borrow is computed after the shift operation using the ALU. For example, suppose that $X \geq Y$ and $X_0 < Y_0$ hold, and $X_0 - Y_0 = (11, 0010, 1\underline{000}, \underline{0000}, \underline{0000})$. The multiplier computes $(X_0 - Y_0) \times 2^{17-11} = (1100, 101\underline{0}, \underline{0000}, \underline{0000}, \underline{0}000, 0000)$, where the 11 consecutive 0 bits are all on the right of the 17-th bit of $(X_0 - Y_0) \times 2^6$ as we shown above. Then, ALU outputs $(X_0 - Y_0) \times 2^6$ as it is. To remove 11 consecutive 0 bits from the least significant bit of $X - Y$, we retain the higher $18 - 11 = 7$ bits from the 17-th bit of $(X_0 - Y_0) \times 2^6$, which is (110,0101). On the other hand, suppose that $X_1 - Y_1 = (01, 1011, 0100, 1011, 0100)$, the multiplier also computes $(X_1 - Y_1) \times 2^6 = (0110, 110\underline{1}, \underline{0010}, \underline{1101}, \underline{0}000, 0000)$. Since there is a borrow from $X_0 - Y_0$, ALU computes $(X_1 - Y_1) \times 2^6 - 1 = (0110, 110\underline{1}, \underline{0010}, \underline{1100}, \underline{111}1, 1111)$. We note that the higher 18 bits of $(X_1 - Y_1) \times 2^6 - 1$ is equal to $X_1 - Y_1 - 1$. Since only 7 bits of $X_0 - Y_0$ are retained, we need to pick up 11 bits from the least significant bit of $X_1 - Y_1 - 1$ to restructure the first 18-bit word of $\texttt{rshift}_{17}(X - Y)$. Hence, we pick up 11 consecutive bits on the right of the 17-th bit of $(X_1 - Y_1) \times 2^6 - 1$, which is (100,1011,0011). Then, we concatenate 11 bits data (100,1011,0011) of $X_1 - Y_1 - 1$ with 7 bits data (110,0101) of $X_0 - Y_0$ to restructure the first word of $\texttt{rshift}_{17}(X - Y)$, which is (10,0101,1001,1110,0101). Also, the other words of $\texttt{rshift}_{17}(X - Y)$ can be obtained in the same way. The configuration of DSP slice is described as follows:

Pre-adder: The pre-adder of DSP slice has 25-bit port D and 30-bit port A. The 36-bit output of the block RAM are connected to the pre-adder via a pipeline register. X is given to port D, and Y is given to port A. The remaining bits of the ports are padded with 0. The pre-adder of DSP slice can compute $D - A$, A and D by controlling its behavior, in other words, the pre-adder outputs $X - Y$, Y or X optionally. For example, to perform the operation $X - Y$, the subtraction $X_i - Y_i$ is performed for each 36-bit data

X_iY_i one by one, and the output of pre-adder is connected to the multiplier.

Multiplier: The embedded multiplier has two input ports, where one accepts an 18-bit two's complement operand from port B via a pipeline register, the other one accepts an 25-bit two's complement operand from the pre-adder via a pipeline register. We use the multiplier to perform the multiplication between the result of pre-adder and value of port B, where B has the value $2^k (0 \leq k \leq 17)$ in our implementation. Thus, the operations $(X_i - Y_i) \times B$, $X_i \times B$, and $Y_i \times B$ can be executed using multiplier. In other words, shift operation can be executed for $X - Y$, X, and Y. The output of multiplier is connected to ALU(Arithmetic logic unit) via a pipeline register M as shown in the Figure 4.3.

ALU: The ALU (Arithmetic Logic Unit) has three input ports, that are connected to register M, input port C of DSP slice, and port CIN, respectively. The most significant bit of register P, the negation of the most significant bit of register P and port $CARRYIN$ are connected to port CIN of ALU. Port CIN can select one of the three values by controlling its behavior. The ALU can performs several operations such as $M+C+CIN$ and $-M-C-CIN-1$. In our implementation, C is configured as the value -1. Since M is connected to the output of multiplier, we can control the behavior of the ALU dynamically for computing $(X_i - Y_i) \times B + CIN - 1$ if $X \geq Y$ holds, and computing $-(X_i - Y_i) \times B - CIN = (Y_i - X_i) \times B - CIN$ if $X < Y$ holds, where CIN is used as the borrow corresponding to the subtraction of previous 36-bit data $X_{i-1}Y_{i-1}$. The value computed by ALU is then connected to register P.

Pattern detector: The pattern detector can determine that the value of register P matches a pattern or not, as qualified by a mask. The mask is used as enable signals for pattern detector. More specifically, if a certain bit of mask is set to "0", the corresponding bit of $PATTERN$ and P is compared. Otherwise, the comparison of the corresponding bits

is not performed. The value of port $PATTERN$ is configured as 0.

Using the block RAM and the functionality of DSP slice, we can perform Hardware Binary Euclidean algorithm without fabric barrel shifter and multiplexers that are used in the preliminary verison of this paper. We show how each operation in Hardware Binary Euclidean algorithm can be performed. Let $x_{1023}x_{1022}\cdots x_0$ denote 1024 bits representing X such that $x_{17}x_{16}\cdots x_0$ represents X_0. Similarly, let $y_{1023}y_{1022}\cdots y_0$ denote 1024 bits representing Y such that $y_{17}y_{16}\cdots y_0$ represents Y_0.

X **is even:** The number X is write to the block RAM word by word. Thus, the condition can be determined by reading the least significant bit of X_0 when X is input into the block RAM.

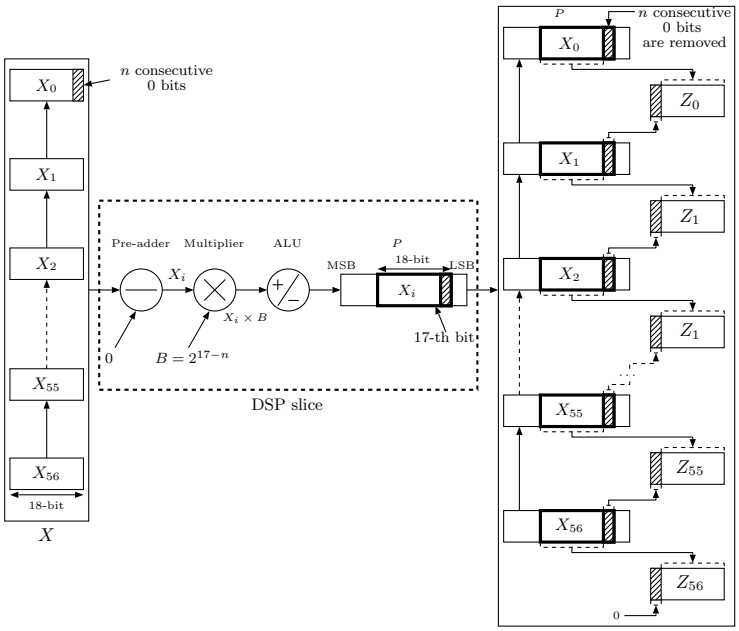


Figure 4.4: The outline of $\text{rshift}_{17}(X)$

$X \leftarrow$ **rshift$_{17}$(**X**):** If X is even, function $\texttt{rshift}_{17}(X)$ is executed to remove the consecutive 0 bits from the least significant bit of X. Suppose that we need to compute Z=$\texttt{rshift}_{17}(X)$. Let $Z_{56}Z_{55}\cdots Z_0$ denote 57 words representing Z and show how

$\texttt{rshift}_{17}(X)$ is computed as the flow shown in Figure 4.4. All words of X are sequentially read from the block RAM beginning with X_0 and then processed one by one in a pipelined order. $X_i (0 \leq i \leq 56)$ is given to the pre-adder of DSP slice. The pre-adder outputs X_i as it is. Also, we must obtain the number of consecutive 0 bits from the least significant bit of X_0 to execute shift operation using the multiplier. Let $\delta = \delta_{17}\delta_{16} \cdots \delta_0$ denote the result of logic prefix-or operation of X_0. The operation $\delta_i \leftarrow x_i \vee \delta_{i-1} (1 \leq i \leq 17)$ is performed, where $\delta_0 = x_0 = 0$ since X is even. For example, suppose that $X_0 = (11, 0010, 1011, 1011, 0000)$, where the number n of consecutive 0 bits of X is 4. We have $\delta = (11, 1111, 1111, 1111, 0000)$. Note that except the consecutive 0 bits from the least significant bit, the other bits all have the value 1. Let $\lambda = \lambda_{17}\lambda_{16} \cdots \lambda_0$ denote the result of exclusive-or operation of δ. The operation $\lambda_i \leftarrow \delta_i \oplus \delta_{i-1} (1 \leq i \leq 17)$ is performed, where $\lambda_0 = \delta_0 = 0$ holds. For the δ shown above, $\lambda = (00, 0000, 0000, 0001, 0000)$ holds. The only one bit that has the value 1 indicates that there are 4 consecutive 0 bits from the least significant bit of X_0. Then, the inverse of λ which has the value $(00, 0010, 0000, 0000, 0000)$, is configured as the value of port B to perform shift operation using the multiplier of DSP slice. We note that if X has $n (0 < n \leq 17)$ consecutive 0 bits, the value of B will be 2^{17-n}. Otherwise, $B = 2^0$ holds. In the case of executing operation $X \leftarrow \texttt{rshift}_{17}(X)$, pre-adder directly outputs X_0 to the multiplier. The product of $X_0 \times 2^{17-n}$ is then computed by the multiplier. Similarly, for other words of X, $X_i \times 2^{17-n} (1 < i \leq 56)$ are also computed one by one in the same way. We note that the consecutive 0 bits of X_0 are always on the right of 17-th bit from the least significant bit of $X_0 \times 2^{17-n}$. In the example above, since $n = 4$ holds, $X_0 \times 2^{17-4} = (110, 0101, 0111, 0110, 0000, 0000, 0000, 0000)$, where the 4 consecutive 0 bits from the least significant bit of X_0 are all on the right of 17-th

bit of $X_0 \times 2^{13}$. The resulting value of $X_0 \times 2^{13}$ is then transferred to ALU via register M. The ALU outputs $M + C + CIN$, where port C is configured as a constant -1. CIN is used as borrow of subtraction of $X - Y$ which is not needed for executing $\texttt{rshift}_{17}(X)$, thus CIN is set to 1. Therefore, ALU outputs the resulting value $X_0 \times 2^{13}$ to register P. We then retain higher $18 - 4 = 14$ bits from 17-th bit of P, that is (11,0010,1011,1011). In other words, the 4 consecutive 0 bits from the least significant bit of X_0 are removed. Since 4 consecutive 0 bits are removed from X_0, we must pick n bits from its next word X_1 of X to restructure the new word Z_0 of $Z = \texttt{rshift}_{17}(X)$. Suppose that $X_1 = (01, 1101, 0010, 0011, 1011)$, the same operation is performed for X_1, and $X_1 \times 2^{17-4} = (011, 1010, 0100, 011\underline{1}, \underline{011}0, 0000, 0000, 0000)$ will be stored in register P in the next clock cycle since the architecture is pipelined. Similarly, 4 bits from the least significant bit of X_1 are also on the right of 17-th bit of P. Thus, we can easily pick 4 bits from the least significant bit of X_1 that are store on the right of 17-th bit of P, and then concatenate with retained 14 bits of X_0 to restructure the new word $Z_0 = (10, 1111, 0010, 1011, 1011)$ of $Z = \texttt{rshift}_{17}(X)$. As shown in Figure 4.4, since $X_{56}X_{55} \cdots X_0$ are input one by one, $Z_{56}Z_{55} \cdots Z_0$ can be computed one by one and then transferred to the block RAM to overwrite the old X. We say that $X \leftarrow \texttt{rshift}_{17}(X)$ is executed such that n consecutive 0 bits from the least significant bit of X are removed. If input X has more than 17 consecutive 0 bits from the least significant bit, the function $\texttt{rshift}_{17}(X)$ is repeated until X is odd. Also, if input Y is even, the same operation is performed for Y.

$X \geq Y$: The condition $X \geq Y$ can be determined by comparing X and Y from the most significant bit. More specifically, X and Y are compared from the words X_{56} and Y_{56}. The words $X_{56}Y_{56}$ are read from the block RAM concurrently, then are connected to port

D and A of DSP slice, respectively. We always assume that $X \geq Y$ holds, thus, the pre-adder computes $X_{56} - Y_{56}$, and the resulting value is input to multiplier. The port B is configured as 2^{17} in this case. Thus, multiplier computes $(X_{56}-Y_{56})\times 2^{17}$. However, since B is 18-bit two's complement, the most significant bit of B is sign bit. Hence, if $B = 2^{17}$, the operation $(X_{56}-Y_{56})\times(-2^{17})$ is computed by multiplier, and the resulting value is then transferred to ALU. The ALU outputs the value to register P as it is. Clearly, the value of $X_{56}-Y_{56}$ is left shifted by 17 bits, and is stored in register P from 34-th bit to 17-th bit. If $X_{56} > Y_{56}$ holds, the most significant bit of P have the value 1 since $(X_{56}-Y_{56})\times(-2^{17})$ is computed by the multiplier. We determine the condition $X \geq Y$ if the most significant bit $P[47]$ of P has the value 0. However, the value $X_{56} - Y_{56}$ may be 0 if $X_{56} = Y_{56}$ holds. Thus, we use the pattern detector to determine that 18 bits in $P[34{:}17]$ of register P are all 0 or not. If $X_{56} = Y_{56}$, $P[34 : 17] = 0$ holds and the detector outputs the value 1. We need to compare the next words $X_{55}Y_{55}$ to determine the condition $X \geq Y$. It takes 3 clock cycles to determine the condition $X_{56} = Y_{56}$ from the words $X_{56}Y_{56}$ are input to the DSP slice, because three-stage pipeline registers are used as shown in Figure 4.3. And in most of cases, we can determine the condition $X \geq Y$ by comparing the words $X_{56}Y_{56}$. Hence, we start to execute the operation $\texttt{rshift}_{17}(X - Y)$ one clock cycle after the words $X_{56}Y_{56}$ are input to DSP slice. More specifically, we start the execution of $\texttt{rshift}_{17}(X - Y)$ from words X_0Y_0 without waiting the determination of the condition $X \geq Y$, which we will show in operation $X \leftarrow \texttt{rshift}_{17}(X - Y)$. If $X_{56} = Y_{56}$ is determined after 3 clock cycles, we terminate the execution of $\texttt{rshift}_{17}(X - Y)$, and restart to compare the next words $X_{55}Y_{55}$ to determine the condition $X \geq Y$.

$X \leftarrow \texttt{rshift}_{17}(X - Y)$: Suppose that we need to compute $Z = \texttt{rshift}_{17}(X - Y)$. Let $Z_{56}Z_{55}\cdots Z_0$ denote 57 words representing Z. As mentioned above, if we execute the

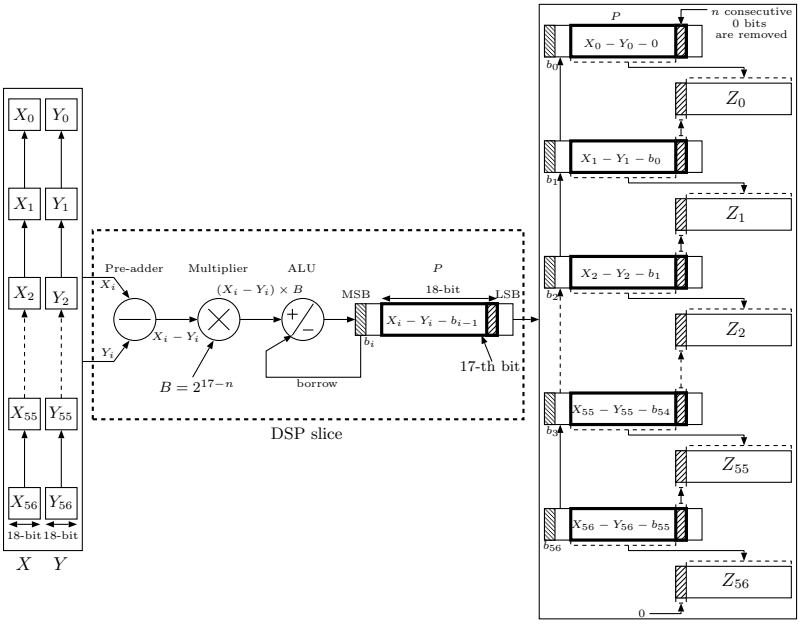


Figure 4.5: The outline of $\texttt{rshift}_{17}(X - Y)$

function $\texttt{rshift}_{17}(X - Y)$ after the condition $X \geq Y$ is determined which takes 3 clock cycles, that is, any operation can be performed in 3 clock cycles for each iteration of do-while loop of the Hardware Binary Euclidean algorithm. Fortunately, we do not need to wait for the determination of the condition $X \geq Y$. In our implementation, all words of X and Y are read from the block RAM one by one beginning with X_0Y_0, one clock cycle after $X_{56}Y_{56}$ are read from the block RAM to determine the condition of $X \geq Y$. Thus, $X_0 - Y_0$ is computed by pre-adder since we assume that $X \geq Y$ always holds. The resulting value of $X_0 - Y_0$ is input to the multiplier, then $(X_0 - Y_0) \times 2^{17-n}$ is computed by the multiplier, where n is the number of consecutive 0 bits from the least significant bit of $X - Y$. Since determination of the condition $X \geq Y$ is executed one clock cycle earlier than function $\texttt{rshift}_{17}(X - Y)$, we can dynamically control the behavior of the ALU depending on the determination of the condition $X \geq Y$. More specifically, the result of $X_{56} - Y_{56}$ is obtained one clock cycle earlier than $(X_0 - Y_0) \times 2^{17-n}$

is accepted by ALU. Hence, if $X_{56} > Y_{56}$ holds, we control the behavior of ALU to compute $(X_0 - Y_0) \times 2^{17-n} + CIN - 1$. If $X_{56} < Y_{56}$ is determined, $-(X_0 - Y_0) \times 2^{17-n} - CIN$ is computed by the ALU. The selection of CIN depends on the borrow of subtraction of words $X_i Y_i (0 \leq i \leq 56)$, and we can also dynamically select the value of CIN by controlling its behavior. For example, if $X \geq Y$ holds, we select CIN as 1 to compute $(X_0 - Y_0) \times 2^{17-n} + 1 - 1 = (X_0 - Y_0) \times 2^{17-n}$. If $X < Y$ holds, we select CIN as 0 to compute $-(X_0 - Y_0) \times 2^{17-n}$. Then, the result of the ALU is stored to register P. Hence, by checking the most significant bit $P[47]$ of register P, we can obtain the borrow of the subtraction $X_0 - Y_0$. Suppose that $X \geq Y$ is determined. If $X_0 \geq Y_0$, $P[47] = 0$ holds, otherwise, $P[47] = 1$ holds. In the same way, $(X_1 - Y_1) \times 2^{17-n} + CIN - 1$ is computed by ALU in the next clock cycle, We select the value of CIN as the negation of $P[47]$ as the borrow from $X_0 - Y_0$. Thus, if $X_0 \geq Y_0$, $(X_1 - Y_1) \times 2^{17-n} + 1 - 1 = (X_1 - Y_1) \times 2$ is computed. Otherwise, $(X_1 - Y_1) \times 2^{17-n} - 1$ is computed. Next, we briefly show how to obtain the word Z_0 of $Z = \texttt{rshift}_{17}(X - Y)$ is computed as shown in Figure 4.5. Suppose that $X_0 \geq Y_0$ holds. Since the result of $X_0 - Y_0$ is shifted by $17 - n$ bits and stored in P, the n consecutive 0 bits from the least significant bit of $X_0 - Y_0$ are on the right of 17-th bit of P. Hence, we retain $18 - n$ bits on the left of 17-th bit of P to store in a register. In other words, the n consecutive 0 bits from the least significant bit of $X_0 - Y_0$ stored on the right of 17-th bit of P are removed. Also, $X_1 - Y_1$ is shifted by $17 - n$ bits and stored in P. Similarly, the n bits from the least significant bit of $X_1 - Y_1$ are stored on the right of 17-th bit of P. Then, we can easily pick up n bits from the least significant bit of $X_1 - Y_1$ to concatenate with higher $18 - n$ bits of $X_0 - Y_0$ to restructure the new word Z_0 as shown in Figure 4.5. The same operation is executed for all words $X_i Y_i (0 \leq i \leq 56)$ in a pipelined order. Hence, the words $Z_{56} Z_{55} \cdots Z_0$ can be obtained

one by one and are then written back to the block RAM to overwrite the old X.

$X \neq 0$: We use a register to store the current number of bits of X. If operation $X \leftarrow \texttt{rshift}_{17}(X)$ or $X \leftarrow \texttt{rshift}_{17}(X - Y)$ is executed, we rewrite the value of this register. We determine the condition $X \neq 0$ if the number of bits of X is not 0.

Let us briefly confirm that the GCD processor core can execute Hardware Binary Euclidean algorithm. By controlling the behavior of pre-adder, multiplier and ALU of DSP slice, we can compute $\texttt{rshift}_{17}(X - Y)$, $\texttt{rshift}_{17}(Y - X)$, $\texttt{rshift}_{17}(X)$ and $\texttt{rshift}_{17}(Y)$ without multiplexers and barrel shifter that use resources of FPGA. The resulting value can be written to the block RAM to overwrite X or Y. The conditions "X is even" and "Y is even" can be determined when X_0 and Y_0 are written in the block RAM. The condition "$X \geq Y$" can be determined by checking X and Y from the MSB (Most Significant Bit). More specifically, if $X_{56} > Y_{56}$ holds, "$X \geq Y$" is determined. We execute the $\texttt{rshift}_{17}(X - Y)$ without waiting the determination of the condition "$X \geq Y$", because the condition "$X \geq Y$" can be determined by comparing the words X_{56} and Y_{56} in most of the cases. However, if $X_{56} = Y_{56}$, we terminate the execution of $\texttt{rshift}_{17}(X - Y)$, and then read and compare X_{55} with Y_{55}. During the computation of Hardware Binary Euclidean algorithm, the number of bits of X and Y is decreased. For example, if X_{56} and Y_{56} both decrease to 0, the next iteration of the do-while loop of Hardware Binary Euclidean algorithm is only performed for words $X_iY_i(0 \leq i < 56)$. We use registers to store the current numbers of bits of X and Y. If the number of bits is 0, we terminate the algorithm.

4.4 Implementation of Hierarchical GCD cluster with DDR3 Memory

This section presents a hierarchical parallel architecture that we call hierarchical GCD cluster using an off-chip DDR3 memory equipped in Xilinx VC707 evaluation board [61]. The proposed GCD processor core is compactly designed based on the FDFM approach. We use only one DSP slice, one block RAM and a few CLBs to implement the processor core. Therefore, single proposed FDFM GCD processor core is clocked at high frequency and provides high performance that we show in the next section. On the other hand, by employing multiple proposed FDFM GCD processors, the computing time reduces considerably. Since the proposed GCD processor is designed based on the FDFM approach and uses very few FPGA resources, we have succeeded in implementing more than one thousand proposed GCD processor cores working in parallel in the FPGA, thus, it makes sense to use multiple servers. Each server controls more than one hundred GCD processor cores. The hierarchical GCD cluster consists of multiple GCD clusters, each of which involves multiple GCD processor cores as illustrated in Figure 4.6. A single central server controls local servers, each of which maintains GCD processor cores in the same GCD cluster.

We show how the hierarchical GCD cluster is used to execute pairwise GCD computation for RSA moduli. The DDR3 memory consists of 8 banks. Each bank has a memory array that can be used to store lots of moduli. Suppose that we have a lot of moduli collected from the Web and all moduli are divided into two sets. We store two sets of moduli to two different banks of DDR3 memory for simplifying the address/control circuit. Our goal is to compute all pairs of moduli using the hierarchical GCD cluster

in an FPGA. For this purpose, we partition all moduli of each set into groups with m moduli each. FPGA picks one group from each set and sends them to the central server, respectively. Let $N = \{n_0, n_1, \ldots n_{m-1}\}$ and $N' = \{n'_0, n'_1, \ldots n'_{m-1}\}$ denote two groups of m moduli each that the central server in the FPGA has received. The hierarchical GCD cluster computes $\gcd(n_i, n'_j)$ for all pairs of i and j ($0 \leq i, j \leq m-1$), and reports the GCDs larger than 1.

Next, we will show how the hierarchical GCD cluster computes the GCDs of N and N' using GCD clusters. Each group of m moduli is partitioned into b blocks of k moduli each, where $m = bk$. Let $N_i = \{n_{ik}, n_{ik+1}, \ldots, n_{(i+1)k-1}\}$ and $N'_i = \{n'_{ik}, n'_{ik+1}, \ldots, n'_{(i+1)k-1}\}$ ($0 \leq i \leq b-1$) be two sets of k moduli in the i-th groups of sets N and N', respectively. Each cluster is assigned a task to compute the GCDs of all pairs X ($\in N_i$) and Y ($\in N'_j$) for a pair i and j ($0 \leq i, j \leq b-1$). For this purpose, all moduli in N_i and in N'_j are copied from the block RAM in the central server to that in the local server of a GCD cluster. After the local server receives all moduli, the cluster starts computing the GCDs of all pairs X ($\in N_i$) and Y ($\in N'_j$). The local server then picks a pair X and Y and copies them to the block RAM of a GCD processor. Upon completion of the copy, the GCD processor starts computing the GCD of X and Y by the Hardware Binary Euclidean algorithm. This procedure is repeated for all GCD processors. If a GCD processor terminates the GCD computation, the local server sends a new pair to it. In this way, the GCDs of all pairs in N_i and N'_j are computed by a GCD cluster. When a GCD cluster completes the computation of all GCDs of a given pair of two groups, the central server picks a new pair i and j and sends all moduli in N_i and in N'_j to the local server. The same operation is repeated until the GCDs of all pairs N and N' are computed.

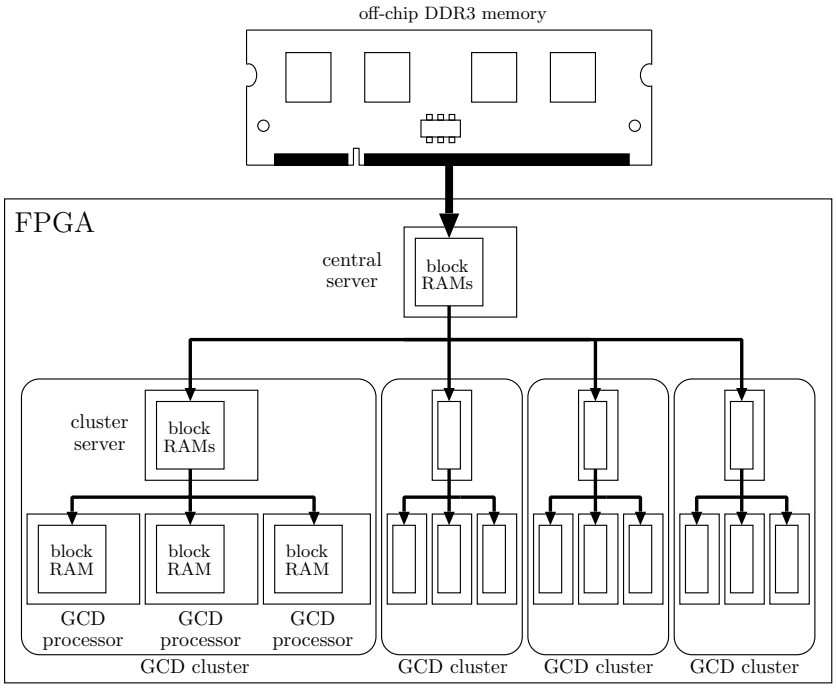


Figure 4.6: The architecture of the Hierarchical GCD cluster

4.5 Experimental results

We have implemented a GCD processor core for computing the GCD of 1024-bit, 2048-bit, 4096-bit, and 8192-bit integers in Xilinx Virtex-7 XC7VX485T-2. Table 4.2 shows the implementation results. Slice Registers and Slice LUTs (Look-Up-Tables) are hardware resources in CLB (Configurable Logic Block) [59], which are used to implement sequential logics. The proposed GCD processor is compactly designed based on FDFM approach. More specifically, we use only one DSP slice to perform subtraction and shift operation for very large numbers and use one block RAM to store the computed result instead of using lots of CLBs. Therefore, the proposed FDFM GCD processor is clocked at over 380MHz and provides a high performance. Calculated simply, single proposed FDFM GCD processor core computes one GCD of two 1024-bit, 2048-bit,

4096-bit and 8192-bit moduli in expected 73.12μs, 253.35μs, 915.78μs and 3614.91μs.

We control the behavior of the embedded ALU of the DSP slice to perform $X - Y$ or $Y - X$ dynamically instead of using multiplexers. Also, we use the embedded multiplier of the DSP slice to perform the shift operation instead of the barrel shifter that uses a lot of FPGA logic resources. Since these mechanisms simplify the circuit of the proposed processor, the frequency of the proposed FDFM processor is over 380MHz that is very high.

Table 4.2: Implementation results of one GCD processor for 1024-bit, 2048-bit, 4096-bit, and 8192-bit moduli

	Slice Registers	Slice LUTs	DSP slices	18k-bit block RAMs	Clock cycles for computing one GCD	Clock Frequency (MHz)
Available	607200	303600	2800	2060		
1024-bit	179	163	1	1	28006.1	383.00
2048-bit	185	174	1	1	98198.5	387.60
4096-bit	191	178	1	1	359131.4	392.16
8192-bit	197	188	1	1	1381328.5	382.12

First, the simulation of pairwise GCD computation for 1024-bit RSA moduli without DDR3 memory is performed. In our implementation, a GCD cluster with a local cluster with eight 18k-bit block RAMs and 128 GCD processor cores are used. Since four 18k-bit block RAMs can store $\lfloor \frac{4\times1024}{57} \rfloor = 71$ moduli with 1024 bits, each GCD cluster computes the GCDs of $71 \times 71 = 5041$ pairs of blocks stored in block RAMs. Hence, each GCD processor computes the GCDs for expected $\frac{5041}{128} = 39.4$ pairs of 1024-bit moduli. Also we arranged 64 block RAMs to the central server. Since a block of moduli is stored in four block RAMs, we can think that the central server has $8 \times 8 = 64$ pairs of blocks. Thus, each cluster computes the GCDs for moduli in expected $\frac{64}{14} = 4.5$ pairs

of blocks since we have succeeded in implementing 14 clusters in an FPGA. Table 4.3 shows the implementation results of clusters of our work. Since a cluster server uses eight 18k-bit block RAMs, each GCD cluster with 128 GCD processors involves 128 + 8 = 136 block RAMs. In this paper, the implementation of the hierarchical GCD cluster with 14 GCD clusters and the central server, uses $14 \times 128 = 1792$ DSP slices and $14 \times 136 + 64 = 1968$ block RAMs. Due to the overhead for the connection between the central server and GCD clusters, the clock frequency is decreased to 207.04MHz. The used block RAMs of the implementation with 14 clusters are close to the available number.

Table 4.3: Implementation results of the GCD cluster and the hierarchical GCD cluster for 1024-bit moduli

	Slice Registers	Slice LUTs	DSP slices	18kb block RAMs	Clock Frequency
Available	607200	303600	2800	2060	(MHz)
one cluster	23414	20598	128	136	327.87
hierarchical clusters	325987	272127	1792	1968	207.04

We have evaluated the number of clock cycles to compute all GCDs of $71 \times 71 = 5041$ pairs of 1024-bit moduli by one GCD cluster. For this purpose, we have used RSA moduli generated by OpenSSL Toolkit. By performing the simulation, one cluster with 128 processors takes 1157789 clock cycles to compute the GCDs of 5041 pairs. If a GCD cluster is clocked at 207.04MHz as shown in Table 4.3, the expected computing time is $1157789/207.04\text{MHz} = 5.592\text{ms}$. Also, it takes about $71 \times 2 \times 57 = 8094$ clock cycles to transfer a pair of two blocks involving 71 moduli each and this overhead is negligible. Since up to 14 clusters can be implemented theoretically, we can expect that

the GCDs of $5041 \times 14 = 70574$ pairs can be computed in the same time. Therefore, we say that one GCD can be computed in expected $5.592\text{ms}/70574 = 0.0792\mu\text{s}$.

Table 4.4: Implementation results of hierarchical GCD clusters for 1024-bit, 2048-bit, 4096-bit, and 8192-bit moduli

	Slice Registers	Slice LUTs	DSP slices	18kb block RAMs	Clock Frequency (MHz)	Average Time (μs)	Number of clusters
Available	607200	303600	2800	2060			
1024-bit	235486	206955	1280	1424	250.00	0.0904	10
2048-bit	220697	204460	1152	1424	250.00	0.3422	9
4096-bit	230636	213670	1152	1568	250.00	1.2537	9
8192-bit	244621	226521	1152	1568	250.00	4.7895	9

Next, for measuring the performance of GCD computation accurately, we implement the hierarchical GCD cluster to compute all pairs of moduli stored in an off-chip DDR3 memory MT8JTF12864HZ-1G6G1 [33] equipped in VC707 evaluation board [61]. Unfortunately, if the used resources of FPGA is close to the available number, the circuit of FPGA becomes unstable and can not compute the results correctly when it is actually operated in the evaluation board. According to the experimental results, 10 clusters can be implemented in the FPGA clocked at 250MHz for pairwise GCD computation of 1024-bit RSA moduli. In other words, 1280 GCD processor cores can be implemented in FPGA XC7VX485T-2 equipped in VC707 evaluation board, and works in parallel to compute GCDs of all pairs of 1024-bit RSA moduli stored in the off-chip DDR3 memory.

We use the built-in CORE Generator software of Xilinx Vivado design suite 2015.1 to generate a DDR3 memory interface core in the FPGA to control the write and read operations of the DDR3 memory. The DDR3 memory consists of 8 banks. Each bank

has a $2^{14} \times 2^{10}$ memory array, of which each element has 64-bit. In other words, each bank of the DDR3 memory can store up to $(2^{14} \times 2^{10} \times 64\text{bits})/1024\text{bits} = 1048576$ 1024-bit RSA moduli. The DDR3 memory runs in 500MHz that is 2 times faster than the FPGA. Moreover, the DDR3 memory offers high-speed data transfers on the rising and falling edges of the clock of it. Hence, the DDR3 memory can perform $500\text{MHz}/250\text{MHz} \times 2 = 4$ times write or read operations in one clock cycle of the FPGA. Hence, we can read $4 \times 64\text{bits} = 256\text{bits}$ data from DDR3 memory in one clock cycle of the FPGA. Suppose that we have a lot of 1024-bit RSA moduli collected from the Web, we divide all moduli into two sets and store them to two different banks of the DDR3 memory. We partition all moduli of each set into groups with $71 \times 8 = 568$ moduli each. FPGA picks one group from each set and sends them to the central server, respectively. More specifically, we send read commands to the DDR3 memory interface core for reading a 1024-bit modulus. Then, the interface core performs the read operation of the DDR3 memory and the modulus is transferred to FPGA after a few clock cycles. The obtained 1024-bit modulus is then stored to the block RAMs of central server as 18-bit words in 57 clock cycles, and we read the next 1024-bit modulus at the same time. The same operation is repeated until two groups of 568 moduli are stored in the central server. Moreover, the interface core processes a refresh operation to maintain the data of the DDR3 memory in refresh interval, and other operations of DDR3 memory must wait for the refresh operation. By implementing the hierarchical GCD cluster with 1280 processor cores in the FPGA, we have that it takes 7294417 clock cycles compute the GCDs of $568 \times 568 = 322624$ pairs, where 71646 clock cycles for transferring 568×2 1024-bit moduli from DDR3 memory to the central server of the FPGA is included. Comparing with the total clock cycles for computing the GCDs of 322624 pairs of 1024-bit moduli,

the clock cycles for transferring moduli from DDR3 memory to central server is negligible. Moreover, after all moduli of central server are transferred to the clusters, we can read the next two groups with 568 moduli each from DDR3 memory while the GCD computation of the clusters is still being performed. In other words, the operation of transferring moduli from DDR3 memory to central server can be overlapped. Hence, we note that the transfer time is not significant. Since the hierarchical GCD cluster runs in 250MHz, the computing time is 7294417/250MHz = 29.178ms. Therefore, we say that one GCD can be computed in 29.178ms/322624 = 0.0904μs. For performing pairwise GCD computation of 2048-bit, 4096-bit, and 8192-bit moduli, we have succeeded in implementing the hierarchical GCD cluster that has 9 clusters in the FPGA, where the frequency of FPGA is also 250MHz. The implementation results of hierarchical clusters and computing time for one GCD of 1024-bit, 2048-bit, 4096-bit, and 8192-bit moduli is also shown in Table 4.4. The hierarchical GCD cluster is designed based on FDFM GCD processors that are compact and use very few FPGA resources. One of the advantage of the FDFM approach is that we can implement multiple FDFM processors working in parallel to reduce the computing time if enough hardware resources are available. Comparing with single FDFM GCD processor core, the computing time of the hierarchical GCD cluster for one GCD reduces considerably by employing more than one thousand FDFM GCD processor cores.

According to the implementation results as shown in Table 4.4, the hierarchical GCD cluster computes one GCD of two 8192-bit moduli in 4.7895μs, that is 52.98 times slower than the time for computing one GCD of two 1024-bit moduli. We show the reason of the large difference. Since the large input numbers are stored in the block RAM as 18-bit words and processed word by word, if the width of the input numbers

increases, the number of iterations of the do-while loop of the Hardware Binary Euclidean algorithm will increase. Also, the clock cycles for performing each iteration of the do-while loop will increase. Hence, the proposed GCD processor takes more clock cycles for computing one GCD of larger numbers. For example, as shown in Table 4.2, single proposed processor takes 1381328.5 clock cycles for computing one GCD of two 8192-bit moduli, that is 49.32 times more than that for computing one GCD of two 1024-bit moduli. This is the main reason for the large difference of the computing time for 1024-bit and 8192-bit moduli. Recall that each 1024-bit and 8192-bit modulus is stored in the block RAM as 57 and 456 18-bit words, respectively. Hence, the central server and cluster server take more time for transferring the 8192-bit moduli than that for 1024-bit moduli. However, since the data transfer is overlapped with the GCD computation, the transfer time does not significantly affect the large difference of the computing time for 1024-bit and 8192-bit moduli. Moreover, the number of clusters for 8192-bit moduli is 9, that is less than that for 1024-bit moduli. Based on the reasons above, one GCD of two 8192-bit moduli is computed 52.98 times slower than one GCD of two 1024-bit moduli in the implementation of hierarchical GCD cluster.

4.6 Concluding remarks

We have presented an efficient processor core for computing GCDs of very large numbers. Since the processor is designed based on the FDFM approach, each processor core uses only one DSP slice and one 18k-bit block RAM. We implement the hierarchical GCD cluster with 1280 processor cores in Xilinx FPGA XC7VX485T-2. The implementation with 1280 processor cores executes pairwise GCD computation for 1024-bit RSA moduli stored in an off-chip DDR3 memory on Xilinx VC707 evaluation board.

The experimental results shows that our implementation of 1280 GCD processor cores computes one GCD of two 1024-bit RSA moduli in 0.0904μs including the time of data transferring from off-chip DDR3 memory to FPGA. It is 3.8 times faster than the best GPU implementation and 316 times faster than a sequential implementation on the Intel Xeon CPU.

Chapter 5

Implementations of the LZW compression and decompression algorithms on the FPGA

LZW compression algorithm is one of the most famous compression algorithms. In this chapter, we first show an efficient hardware architecture for accelerating LZW compression using an FPGA. Since the proposed module of LZW compression is compactly designed, we have succeeded in implementing 24 identical modules in an FPGA. On the other hand, we present a hardware LZW decompression algorithm and implement it in the FPGA. Also, we have succeeded in implementing 34 LZW decompression modules which works in parallel on the FPGA. According to the experimental results, the implementation of 24 modules of LZW compression attains a speed-up factor of 23.51 times faster than a sequential software implementation on a single CPU. The implementation of 34 LZW decompression runs up to 64.39 times faster than a sequential implementation on the CPU.

5.1 Introduction

Data compression is one of the most important tasks in the area of computer engineering. It is always used to improve the efficiency of data transmission and save the storage of data. Data compression includes two basic methods, lossy compression and lossless compression. Lossy compression uses the fact that human are not sensitive to some frequency ingredients of image or sound. Some information of the original data are discarded in lossy compression. Thus, the decompressed data are not identical to the original data. On the other hand, lossless compression preserves all information of the original data. In other words, the decompression of lossless compression creates exactly the same data with the original data.

Some famous compression algorithm are proposed such as LZ77 [62], LZ78 [63] and LZW [47]. LZ77 algorithm uses two buffers such as dictionary buffer and preview buffer. Dictionary buffer includes the processed data and preview buffer stores the pending data. In LZ77 algorithm, the longest string of preview buffer matching to the string of dictionary buffer is converted to a code that corresponds to the index of dictionary buffer. However, it is not suitable to hardware implementation since it needs a large dictionary buffer and preview buffer. LZ78 algorithm creates a dictionary table and finds the longest matched string in the dictionary table. If there is no matched string in the dictionary table, it outputs the index of dictionary table and the last character of the unmatched string. LZW algorithm is a variant of LZ78 algorithm that outputs only the index of matched string of dictionary table. In our implementation, we focus on LZW compression which is used in Unix utility “compress” and in GIF image format. LZW compression is included in TIFF standard [1], which is widely used in the area of commercial digital printing. The LZW compression algorithm converts an input string

of characters into a series of codes using a dictionary that maps strings into codes. In LZW compression in TIFF standard, characters are 8-bit unsigned integers representing intensity levels of gray-scale image, and codes are 12-bit unsigned integers. Since dictionary tables are created by reading input data one by one, LZW compression and decompression are hard to parallelize. One of the main goals of this chapter is to develop an efficient hardware architecture of LZW compression and implement it in an FPGA. Furthermore, suppose that a high-definition image or video is compressed to a file once, and stored to the server of a commercial organization to be accessed by users in different regions or countries. The compressed file is transferred to users through the network, and then is decompressed locally. Hence, decompression is performed more frequently than compression. Also, the other goal of this chapter is to present an efficient hardware architecture of LZW decompression in the FPGA.

Recent FPGAs have embedded block RAMs. As illustrated in Figure 2.2, the Xilinx Virtex-7 family FPGAs have block RAMs, each of which is an embedded dual-port memory supporting synchronized read and write operations, and can be configured as a 36k-bit or two 18k-bit dual port RAMs [60]. Since FPGA chips maintain relatively low price and its programmable features, it is suitable for a hardware implementation of image processing method to a great extent. They are widely used in consumer and industrial products for accelerating processor intensive algorithm.

Numerous implementations of variety of LZW compression and decompression on FPGAs or VLSIs have been proposed to accelerate the computation. LZRW3 data compression core [19] is designed by Helion technology. This data compression core uses LZRW3 algorithm that is a variant of LZ77 algorithm. This core provides a maximum compression throughput of 172MBytes/s and a maximum decompression throughput

of 180.75MBytes/s in Xilinx Virtex-5 FPGA. Navqi *et al.* [37] implemented a variant of LZW algorithm in Xilinx Virtex-2 FPGA, where only one fixed-length dictionary table is used. This implementation provides the maximum compression and decompression throughputs of 87.5MBytes/s and 160MBytes/s in Xilinx Virtex-2 FPGA. Several implementations of data compression are proposed based on PDLZW(Parallel Dictionary LZW) algorithm that is a variant of LZW algorithm [31, 32, 40]. Instead of one variable-length table used in LZW algorithm, multiple fixed-length tables are used in PDLZW algorithm to accelerate the speed of data compression. Lin implemented the PDLZW algorithm in a VLSI that provides a maximum compression throughput of 33.33MBytes/s and a maximum decompression throughput of 45.5MBytes/s [31]. Lin *et al.* also proposed a two stage hardware architecture that combines PDLZW and AH(Adaptive Huffman) algorithm and implement it in a VLSI [32]. By decreasing the number of parallel dictionaries, this implementation provides a maximum compression throughput of 125MBytes/s and a maximum decompression throughput of 83MBytes/s. In these hardware implementations, the LZW compression algorithms is modified to be suitable for hardware implementation. However, these modified algorithms sacrifice compression ratio and provide worse performance than the original LZW compression algorithm. On the other hand, there is some research for accelerating the computation of LZW compression using GPUs (Graphics Processing Units) [15, 44], multiprocessor [26] and cluster systems [34]. However, as far as we know, there is no hardware implementation of the original LZW compression and decompression algorithms since it is not easy to implement them.

The first contribution of this chapter is to present an efficient hardware LZW compression algorithm and to implement it in an FPGA. In general, the original LZW

compression uses a dictionary table which stores variable-length strings. On the other hand, in our implementation, we use a pointer-character table efficiently. Each pair of pointer and character corresponds to a string. Characters are 8-bit unsigned integers and pointers are 12-bit unsigned integers. Since the creation of dictionary depends on input string of characters, there are $2^{12} \times 2^8 = 2^{20}$ possible combinations of pointer and character. Moreover, each pair of pointer and character corresponds to an index of the dictionary, where the index is 12-bit unsigned integers if dictionary size is 4096. If the pointer-character table is implemented in a straightforward way, the table needs $2^{20} \times 12\text{bits} = 1.5\text{MBytes}$. Therefore, we have reduced the size of the pointer-character table to 32KBytes using a hash table. In the proposed architecture, we efficiently use dual-port block RAMs embedded in the FPGA to implement a hash table that is used as the dictionary. Using independent two ports of the block RAM, reading and writing operations for the hash table are performed simultaneously. Additionally, we can read eight values in the hash table in one clock cycle by partitioning the hash table into eight tables. The proposed module of LZW compression in Virtex-7 family FPGA uses 104 slice registers, 346 slice LUTs and 18 block RAMs with 18k-bit each, where the frequency of FPGA is 179.99MHz. Since the compression ratio is data dependent, the throughput of our implementation for LZW compression differs for input data. When the compression ratio of input file is lower, that means the size of compressed file is larger, the throughput time is shorter. On the other hand, when the compression ratio of input file is higher, the throughput time is longer. According to the experimental results, the compression throughput of the proposed module is 118.73MBytes/s while the compression ratio (original image size : compressed image size) is 1.43:1. On the other hand, the compression throughput is 86.79MBytes/s while the compression ratio

is 36.72:1. Furthermore, since the proposed module of LZW compression uses a few FPGA resources, we have succeeded in implementing 24 identical modules in an FPGA, where the frequency is 163.35MHz and each module has independent input/output ports to work in parallel. Hence, the implementation of 24 proposed modules attains a speed up factor that surpasses 23.51 times over a sequential algorithm on a single CPU. The number of available input/output ports of the targeted FPGA is the bottleneck of our FPGA implementation of multiple modules. In addition, assuming that the limitation of the number of input/output ports is ignored, we show that at most 110 proposed modules can be arranged in the FPGA. It is a theoretical result, but the result shows the compactness of the proposed architecture.

On the other hand, the second contribution of this chapter is to present an efficient hardware LZW decompression algorithm and to implement it in an FPGA. In general, LZW decompression uses a dictionary table which stores variable-length strings. However, in our hardware algorithm we use two tables, pointer table p and character table C_f which store a single value in each entry. The algorithm consists of three steps and these steps are concurrently executed efficiently using the dual-port block RAMs. The proposed module of hardware LZW decompression algorithm in Virtex-7 family FPGA uses 278 slice registers, 307 slice LUTs and 13 block RAMs with 18k-bit, where the frequency of FPGA is 301.02MHz. The running time of proposed module attains a speed up factor that surpasses 2.16 times over a sequential algorithm on a single CPU. Since the decompression throughput is data dependent, according to the experimental results, the decompression throughput of our module is about 280.17MBytes/s while the compression ratio of input file is extremely high. Even in the worst condition, the decompression throughput of proposed module is about 143.54MBytes/s. Also, since

the proposed module uses very few resources, we have succeeded in implementing 34 modules in an FPGA, where all modules works in parallel clocked at 263.92MHz. The implementation of 34 proposed LZW decompression modules attains a speed up factor that surpasses 64.39 times over a sequential algorithm on a single CPU. Similarly, we implement multiple modules in the FPGA, where we assume that the limitation of the number of input/output ports is ignored. We show that 150 LZW decompression modules can be arranged to the FPGA.

5.2 LZW compression and decompression algorithms

The main purpose of this section is to review LZW compression and decompression algorithms. For details of the algorithms, the interested reader may refer to Section 13 in [1].

The LZW (Lempei-Ziv-Welch) [47] lossless data compression algorithm converts an input string of characters into a series of codes using a dictionary table that maps strings into codes. If the input is an image, characters may be 8-bit unsigned integers. It reads characters in an input image string one by one and adds an entry in a dictionary table. In the same time, it writes an output series of codes by looking up the dictionary table. Let $X = x_0x_1 \cdots x_{n-1}$ be an input string of characters and $Y = y_0y_1 \cdots y_{m-1}$ be an output string of codes. For simplicity, we assume that an input string is a string of 4 characters a, b, c and d. Let C be a dictionary table, which determines a mapping of a code to a string, where codes are non-negative integers. Initially, $C(0) = a$, $C(1) = b$, $C(2) = c$ and $C(3) = d$. By operation AddTable, new code is assigned to a string. For example, if AddTable(cb) is executed after initialization of C, we have $C(4) = cb$.

The LZW compression algorithm finds the longest prefix Ω of the current input that

is already added in the dictionary table, and outputs the code of Ω. Let x be the following character of Ω. Since $\Omega \cdot x$ is not in the dictionary table, it is added to the dictionary, where "$\cdot$" denotes the concatenation of string/character. The same procedure is repeated from x. Let $C^{-1}(\Omega)$ denote the index of C where Ω is stored. For example, if $C(3) = d$, then $C^{-1}(d) = 3$. The LZW compression algorithm is described as follows:

[LZW compression algorithm]

$\Omega \leftarrow x_0$;

for $i \leftarrow 1$ to $n - 1$ do

  if($\Omega \cdot x_i$ is in C)

    $\Omega \leftarrow \Omega \cdot x_i$;

  else

    Output($C^{-1}(\Omega)$); AddTable($\Omega \cdot x_i$); $\Omega \leftarrow x_i$;

Output($C^{-1}(\Omega)$);

Table 5.1 shows the compression flow of an input string "*cbcbcbcda*". First, $\Omega \leftarrow x_0 = c$ is executed. Next, since $\Omega \cdot x_1 = cb$ is not in C, $C^{-1}(c) = 2$ is output and cb is added in the dictionary, then we have $C(4) = cb$. Also, $\Omega \leftarrow x_1 = b$ is performed. It should have no difficult to confirm that 214630 is output by this algorithm.

Table 5.1: LZW compression flow for input string $X = cbcbcbcda$

i	0	1	2	3	4	5	6	7	8	-
x_i	c	b	c	b	c	b	c	d	a	-
Ω	-	c	b	c	cb	c	cb	cbc	d	a
S	-	cb(4)	bc(5)	-	cbc(6)	-	-	$cbcd$(7)	da(8)	-
Y	-	2	1	-	4	-	-	6	3	0

Next, let us discuss implementations of dictionary table C. The following operations for a string Ω of characters and the following character x must be supported for LZW compression. **(i)** determine if $\Omega \cdot x_i$ is in C. **(ii)** return the value of $C^{-1}(\Omega)$. **(iii)** perform AddTable($\Omega \cdot x_i$). A straightforward implementation of the dictionary table C, which uses an array such that i-th ($i \geq 0$) element stores $C(i)$. However, since the lengths of strings in C are variable, the straightforward implementation of dictionary C is not efficient. All values of $C(i)$ may be accessed to compute $C^{-1}(\Omega)$, We can use an associative array with keys $C(i)$ and values i, which can be implemented by a balanced binary tree or a hash table. But, these operations take more than $O(|\Omega|)$ time. If the compression ratio is high, Ω may be a long string. Hence, it is not a good idea to use a conventional associative array to implement C.

In this section, we use a pointer-character table to implement the dictionary table C as shown in Table 5.2. In this table, a pointer $p(j)$ and a character $c(j)$ are stored for each code j. Also, a back-pointer $q(j, x)$ for every code j and character x is used. Back-pointer table q can be implemented using an associative array which we will discuss later. We can obtain a string $C(j)$ by traversing p until we reach NULL. More specifically, $C(j)$ can be obtained from p and c by the following definition:

$$C(j) = \begin{cases} c(j) & \text{if } p(j) = \text{NULL} \\ C(p(j)) \cdot c(j) & \text{otherwise} \end{cases} \tag{5.1}$$

For example, in Table 5.2, we have $C(6) = C(4) \cdot c = C(2) \cdot bc = cbc$. A back-pointer $q(j, x)$ takes value k if $p(k) = j$ and $c(k) = x$. If there exists no k such that $p(k) = j$, then $q(j, k) = \text{NULL}$. This is used to perform the three operations above efficiently.

We implement operation AddTable($\Omega \cdot x_i$) for dictionary C by performing operation AddTable(j,x_i) for the pointer-character table. If AddTable(j,x_i) is performed, a new entry k with $p(k) = j$ and $c(k) = x_i$ is added to the pointer-character table. In other

Table 5.2: A pointer-character table and a back-pointer table to implement dictionary table C

j	0	1	2	3	4	5	6	7	8	9
$p(j)$	NULL	NULL	NULL	NULL	2	1	4	6	3	0
$c(j)$	a	b	c	d	b	c	c	d	a	-
$q(j,a)$	NULL	NULL	NULL	8	NULL	NULL	NULL	NULL	NULL	NULL
$q(j,b)$	NULL	NULL	4	NULL	NULL	NULL	NULL	NULL	NULL	NULL
$q(j,c)$	NULL	5	NULL	NULL	6	NULL	NULL	NULL	NULL	NULL
$q(j,d)$	NULL	NULL	NULL	NULL	NULL	7	NULL	NULL	NULL	NULL
$C(j)$	a	b	c	d	cb	bc	cbc	$cbcd$	da	-

words, the value k is written in $q(j, x_i)$ of back-pointer table. Using the back-pointer table, we can rewrite LZW compression algorithm as follows:

[LZW compression algorithm with the back-pointer table]

$j \leftarrow c^{-1}(x_0)$;

for $i \leftarrow 1$ to $n-1$ do

  if($q(j, x_i) \neq$ NULL)

    $j \leftarrow q(j, x_i)$;

  else

    Output(j); AddTable(j, x_i); $j \leftarrow c^{-1}(x_i)$;

Output(j);

We show how Table 5.2 is created. First, $j \leftarrow c^{-1}(x_0) = 2$ is executed. Next, since $q(j, x_1) = q(2, b)$ is NULL, Output(2) and AddTable(2,b) are executed. Then, the pointer-character table has new entry $p(4) = 2$ and $c(4) = b$. Also, the value 4 is stored in $q(2, b)$, and operation $j \leftarrow c^{-1}(b) = 1$ is executed. In the next iteration of the for-loop,

since $q(1, c)$ is NULL, Output(1) and AddTable(1,c) are executed. The pointer-character table has new entry $p(5) = 1$ and $c(5) = c$, and the value 5 is added in $q(1, c)$. Similarly, we can confirm that a series of codes 214630 is output by this algorithm. We will show the implementation of this algorithm using the back-pointer table q afterwards.

Next, let us show LZW decompression algorithm. Let C be *the code table*, such that $C(0) = a$, $C(1) = b$, $C(2) = c$, and $C(3) = d$. Also, let $C_1(i)$ denote the first character of code i. For example, $C_1(4) = c$ if $C(4) = cb$. Similarly to LZW compression, the LZW decompression algorithm reads a string Y of codes one by one and adds an entry of a code table. At the same time, it writes a string X of characters. The LZW decompression algorithm is described as follows:

[LZW decompression algorithm]

Output($C(y_0)$);
for $i \leftarrow 1$ to $n - 1$ do
  if(y_i is in C)
    Output($C(y_i)$); AddTable($C(y_{i-1}) \cdot C_1(y_i)$);
  else
    Output($C(y_{i-1}) \cdot C_1(y_{i-1})$); AddTable($C(y_{i-1}) \cdot C_1(y_{i-1})$);

Table 5.3 shows the decompression process for a code string 214630. First, $C(2) = c$ is output. Since $y_1 = 1$ is in C, $C(1) = b$ is output and AddTable(cb) is performed. Hence, $C(4) = cb$ holds. Next, since $y_2 = 4$ is in C, $C(4) = cb$ is output and AddTable(bc) is performed. Thus, $C(5) = bc$ holds. Since $y_3 = 6$ is not in C, $C(y_2) \cdot C_1(y_2) = cbc$ is output and AddTable(cbc) is performed. The reader should have no difficulty to confirm that *cbcbcbcda* is output by this algorithm.

Since the length of strings in C are variable, it is difficult to implement this algorithm on hardware as it is. Therefore, we propose a new LZW decompression algorithm for

Table 5.3: Code table C and the output string for 214630

i	0	1	2	3	4	5
y_i	2	1	4	6	3	0
C	-	cb(4)	bc(5)	cbc(6)	$cbcd$(7)	da(8)
X	c	b	cb	cbc	d	a

hardware implementation without such dictionary table.

We assume that input characters are selected from an alphabet (or a set) with k characters $\alpha(0), \alpha(1), \ldots, \alpha(k-1)$. We use $k = 4$ characters $\alpha(0) = a$, $\alpha(1) = b$, $\alpha(2) = c$, and $\alpha(3) = d$, when we show examples as before. Let $Y = y_0 y_1 \cdots y_{m-1}$ denote the compressed string of codes obtained by the LZW compression algorithm.

Before showing the LZW decompression for hardware implementation, we define several notations. We define pointer table p using code table Y as follows:

$$p(i) = \begin{cases} \text{NULL} & \text{if } 0 \leq i \leq k-1 \\ y_{i-k} & \text{if } k \leq i \leq k+m-1 \end{cases} \tag{5.2}$$

We can traverse pointer table p until we reach NULL. Let $p^0(i) = i$ and $p^{j+1}(i) = p(p^j(i))$ for all $j \geq 0$. In other words, $p^j(i)$ is the code where we reach from code i in j pointer traversing operations. Let $L(i)$ be an integer satisfying $p^{L(i)}(i) = \text{NULL}$ and Let C_f be the character table defined as follows:

$$C_f(i) = \begin{cases} \alpha(i) & \text{if } 0 \leq i \leq k-1 \\ C_f(p(i)) & \text{if } k \leq i \leq k+m-1 \end{cases} \tag{5.3}$$

It should have no difficulty to confirm that $C_f(i)$ represents the first character of $C(i)$, and $L(i)$ is the length of $C(i)$. Using C_f and p, we can define the value of $C(i)$ in following. If $0 \leq i \leq k-1$, $C(i) = C_f(i)$. On the other hand, if $k \leq i \leq k+m-1$, $C(i) = C_f(p^{L(i)-1}(i)) \cdot C_f(p^{L(i)-2}(i)+1) \cdot C_f(p^{L(i)-3}(i)+1) \cdots C_f(p^0(i)+1)$.

Table 5.4: The values of p, L, C_f and C if $Y = 214630$

i	0	1	2	3	4	5	6	7	8	9
$p(i)$	NULL	NULL	NULL	NULL	2	1	4	6	3	0
$C_f(i)$	a	b	c	d	c	b	c	c	d	a
$L(i)$	1	1	1	1	2	2	3	4	2	-
$C(i)$	a	b	c	d	cb	bc	cbc	$cbcd$	da	-

Table 5.4 shows the value of p, C_f, L, and C for $Y = 214630$. According to the table, we can obtain the decompressed string. Figure 5.1 shows an example of obtaining the decompression string of code $y_3 = 6$, that is $C(6)$, from the table. For code $y_3 = 6$, we first read $p(6) = 4$ from table p. Also, we read $C_f(6 + 1) = c$ from table C_f that corresponds to the last character of $C(6)$. Since the obtained pointer 4 is not NULL, we continue the traversing of table. Next, $p(4) = 2$ and $C_f(4 + 1) = b$ are read from tables p and C_f, respectively. Finally, pointer $p(2)$ is read out, and we stop the traversing operation for code y_3 because $p(2)$ is NULL. Also, $C_f(2) = c$ is read out as the first character of the string corresponding to code y_3. We note that each character of string corresponding to a code is obtained in reverse order.

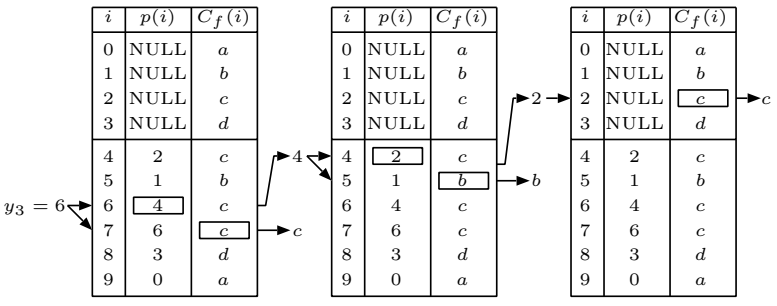


Figure 5.1: An example of traversing tables p and C_f

We are now in position to show hardware LZW decompression. This algorithm can

be done in three steps as follows:

[Hardware LZW decompression algorithm]

Step 1: Update tables p and C_f.

Step 2: Compute partially-reversal strings of X and L using p and C_f.

Step 3: Reorder decompression string X.

In Step 1, we initialize the values of $p(i)$, $C_f(i)$ for each i ($0 \leq i \leq k-1$). After that, we compute the values of $p(i)$ and $C_f(i)$ for each i ($k \leq i \leq k+m-1$). The details of Step 1 are spelled out as follows:

[Step 1 of hardware LZW decompression algorithm]

for $i \leftarrow 0$ to $k-1$ do

$p(i) \leftarrow$ NULL; $C_f(i) \leftarrow \alpha(i)$;

for $i \leftarrow k$ to $k+m-1$ do

$p(i) \leftarrow y_{i-k}$; $C_f(i) \leftarrow C_f(y_{i-k})$;

In Step 2 of hardware LZW decompression algorithm, for each compressed code y_i ($0 \leq i \leq m-1$) of Y, $C^r(y_i)$ is read from table C_f by traversing pointer table p, where $C^r(i)$ denotes a string obtained by reversing $C(i)$. At the same time, the length of string $L(i)$ is also computed. By traversing table C_f with table p, the reversed string is read and temporally stored to an output buffer for each character. Let o denote a table for storing characters of concatenation of strings $C^r(y_0) \cdot C^r(y_1) \cdots C^r(y_{m-1})$. For example, if $C(7) = abc$, in Step 2, we have $C^r(7) = cba$ and $L(7) = 2$. The details of Step 2 of hardware LZW decompression algorithm are shown as follows:

[Step 2 of hardware LZW decompression algorithm]

$addr \leftarrow 0$

for $i \leftarrow 0$ to $m-1$ do

$j \leftarrow y_i$; $L(i) \leftarrow 0$;

while($p(j) \neq$ NULL)

$o(addr) \leftarrow C_f(j+1)$; $j \leftarrow p(j)$;

$L(i) \leftarrow L(i)+1$; $addr \leftarrow addr+1$;

$o(addr) \leftarrow C_f(j)$; $L(i) \leftarrow L(i)+1$; $addr \leftarrow addr+1$;

In Step 3 of hardware LZW decompression algorithm, a concatenated string of $C^r(y_0) \cdot C^r(y_1) \cdots C^r(y_{m-1})$ stored in output buffer o is arranged in corrected order, that is, $C(y_0) \cdot C(y_1) \cdots C(y_{m-1})$. Each ordered character is output one by one. The algorithm code of Step 3 is shown as follows:

[Step 3 of hardware LZW decompression algorithm]

$addr \leftarrow 0$;

for $i \leftarrow 0$ to $m-1$ do

$l \leftarrow L(i)$;

while($l > 0$)

Output($o(addr+l-1)$); $l \leftarrow l-1$;

$addr \leftarrow addr + L(i)$;

By sequentially executing Step 1, Step 2, and Step 3, LZW decompression can be performed. In addition, the execution of these steps can be overlapped. More specifically, after an execution for an input code in each step is completed, the execution for the code in the next step can be started. Figure 5.2 illustrates a process of the above execution for an input compressed code $Y = y_0 y_1 \cdots y_{m-1}$. We will show the FPGA implementation of the hardware LZW decompression algorithm after showing the implementation of LZW compression. In our FPGA implementation of LZW decompression,

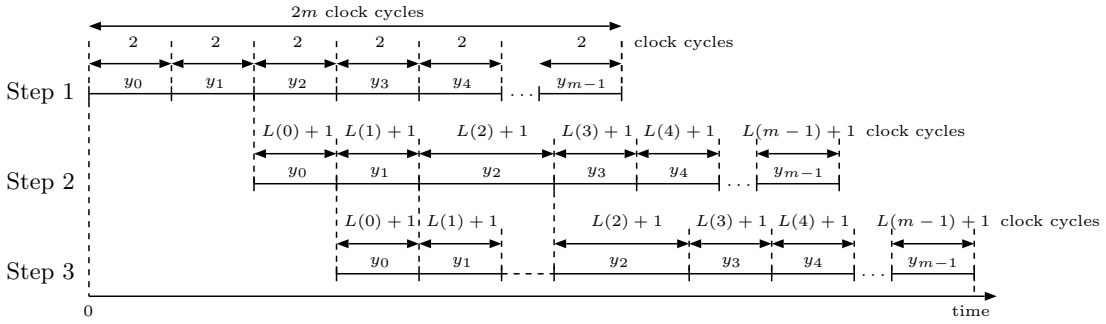


Figure 5.2: Process of our LZW decompression hardware for an input compressed code $Y = y_0 y_1 \cdots y_{m-1}$

we use block RAMs of FPGA to implement the pointer table p, character table C_f, and output buffer o. In the utilized FPGA, the block RAMs can be configured as a dual-port block RAM. Since dual-port block RAM has two set of ports that work independently, the writing and reading operations of these tables can be performed concurrently. Using the block RAMs efficiently, we realizes the overlapped execution of the three steps.

5.3 TIFF image file

In our implementation, we focus on the compression of an image into a TIFF image file, and the decompression of LZW compressed data in a TIFF image file. We assume that a TIFF image file contains a gray-scale image with 8-bit depth, that is, each pixel has intensity represented by an 8-bit unsigned integer. Since each of RGB or CMYK color planes can be handled as a gray scale image, it is obvious to modify gray scale image TIFF compression for color image compression. For further details on a TIFF image file, the interested reader may refer to [1].

We show how image data in a TIFF image file is compressed. Since every pixel

has an 8-bit intensity level, we can think that an input string of an integer in the range $[0, 255]$. Hence, codes from 0 to 255 are assigned to these integers. Code 256 (ClearCode) is reserved to clear the code table. Code 257 (EndOfInformation) is used to specify the end of the data. Thus, AddTable operations assign codes to strings from code 258. While the entry of the code table is less than 512, codes are represented as 9-bit integer. After adding code table entry 511, we switch to 10-bit codes. Similarly, after adding code table entry 1023 and 2037, 11-bit codes and 12-bit codes are used, respectively. As soon as code table entry 4094 is added, ClearCode is output. After that, the code table is re-initialized and AddTable operations use codes from 258 again. The same procedure is repeated until all pixels are converted into codes. After the code for the last pixel is output, EndOfInformation is written out. We can think that a code string is separated by ClearCode, We call each of them *a code segment*. Except the last one, each code segment has $4094 - 257 + 1 = 3838$ codes. The last code segment may have codes less than that.

5.4 FPGA architecture for LZW compression

This section describes our FPGA architecture of the LZW compression algorithm with back-pointer table using block RAMs in Xilinx Virtex-7 FPGA. We use Xilinx Virtex-7 Family FPGA XC7VX485T-2 as the target device [55].

In this section, we show the implementation of the back-pointer table q for TIFF LZW compression. Recall that codes have 12 bits and characters have 8bits, thus, we can implement q as a table that has $2^{12} \times 2^8 = 2^{20}$ entries. Also, the value of each entry $q(j, x)$ has 12 bits. Hence, the back-pointer table needs $2^{20} \times 12\text{bits} = 1.5\text{MBytes}$. However, the straightforward implementation has large overhead due to the cache miss.

In other words, most entries of the table are not used. Thus, we use a hash table to implement the back-pointer table q.

In the proposed FPGA implementation, we use a hash table that is suitable for FPGA implementation. The hash table consists of 1024 buckets $B_s (0 \leq s \leq 1023)$ and each bucket B_s has 8 entries $e_{s,0}, e_{s,1}, \ldots, e_{s,7}$. To implement this hash table, we use two tables, *number table* and *data table*. Let $|B_s|$ denote the number of values stored in bucket B_s. Each element of the number table stores $|B_s|$. Also, the data table stores values of back-pointers. The table is partitioned into 8 tables, each of which stores one of the 8 entries. Each entry stores 12-bit pointer j, 8-bit character x and 12-bit back-pointer $q(j, x)$. Figure 5.3 illustrates the structure of the hash table.

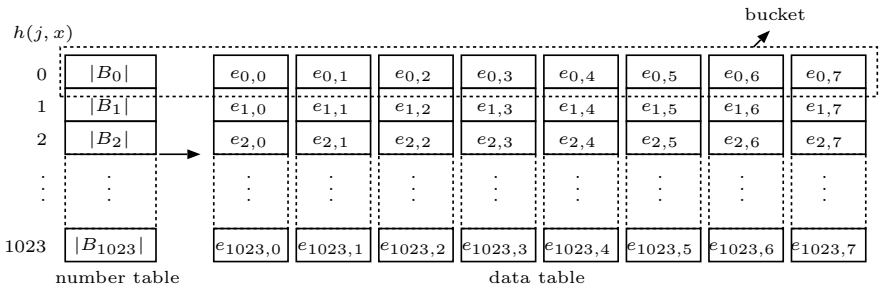


Figure 5.3: The arrangement of hash table

Let $h(j, x)$ be a hash function returning a 10-bit number, where pointer j is 12 bits and character x is 8 bits. We can use the following hash function h to specify a 10-bit number.

$$h(j, x) = ((j << 4) \oplus (j >> 6) \oplus (x << 1)) \wedge 0\text{x}3\text{FF} \tag{5.4}$$

Using this hash funtion, we select a bucket in address $h(j, x)$ and store the value of back-pointer in one of the eight entries in the bucket. However, when the bucket may be full, that is, eight values are already stored in the bucket. If this is the case, called *conflict*,

the current value of each address $(h(j, x) + i) \wedge$ 0x3FF is read for $i = 1, 2, \ldots$ until a bucket that has unused entries is found. We can easily find whether the bucket B_s is full or not by referring $|B_s|$ in the number table. Regarding the size of the hash table, since the total size of the hash table is 8192 and at most 3837 elements are added, conflict may occur, but it is clear that the hash table can store all data.

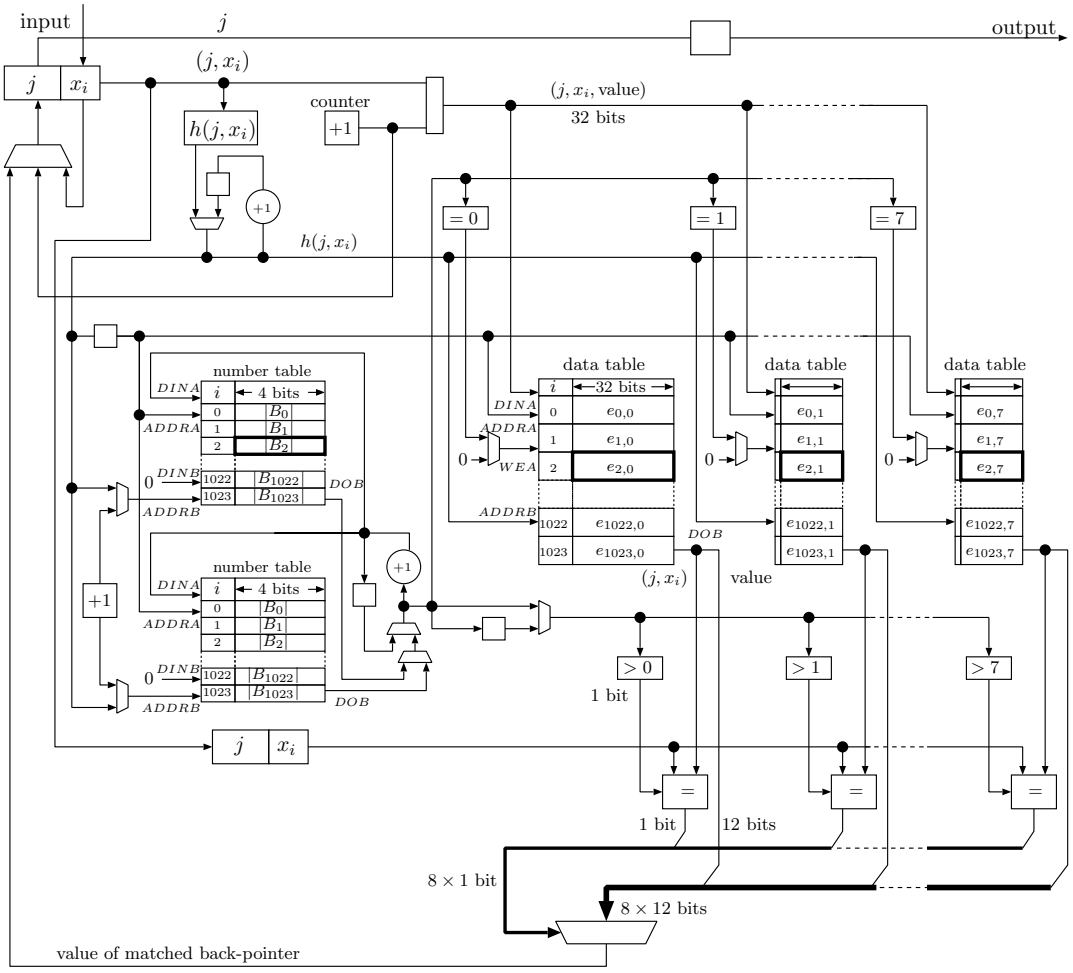


Figure 5.4: The outline of our FPGA architecture for LZW compression algorithm

Figure 5.4 shows the proposed architecture of LZW compression. In our implementation, characters in an input string are input one by one and the corresponding compressed codes are output.

In the LZW compression, it is necessary to find whether a value of back-pointer is already stored or not. Since the data table is partitioned into 8 tables, we read 8 values at the same time. Therefore, given an address of bucket from the hash function, we can find whether a value that includes the back-pointer is stored or not without checking eight entries in the bucket one by one.

On the other hand, the number table consists of 1024 entries with 4 bits that represent the number of used entries in each bucket. For example, if an element in the number table is 0, we can find that the corresponding bucket does not have any values. Using the number table, we can simply determine an element whether it is already stored or not. Recall that we need to initialize all entries in the hash table whenever compression for each segment is finished, that is, ClearCode is output. Since each entry represents the number of used entries in each bucket, we set each entry to zero without clearing the data tables.

In the proposed architecture, we perform LZW compression algorithm shown in the previous section. The main part of the architecture is the hash table as described in the above. There are three operations for the hash table, (i) *initialize operation*, (ii) *find operation,* and (iii) *add operation*. We not that in the LZW algorithm, delete operation that removes a value stored in the table is not necessary. We show the details of these operations, as follows.

Initialize operation: The initialize operation is to clear all entries in the hash table. We need to initialize all entries in the hash table whenever compression for each segment is finished, that is, ClearCode is output. As shown in the above, we clear only the number table to initialize the hash table. However, the next characters cannot be input during the initialization. Therefore, in the proposed architecture, we use two number

tables and switch them in turn whenever ClearCode is output. More specifically, one of them is used to perform LZW compression, while the other is initialized. Since every segment excluding the last segment has 3838 codes and the last code is ClearCode, it takes at least 3838 clock cycles to process a segment. Since the number table has 1024 entries, the initialized operation can be performed while another segment is processed.

Find operation: Given pointer j and character x, this operation is to find whether an entry that includes a back-pointer $q(j, x)$ is stored in the hash table and if it is already stored, outputs the back-pointer $q(j, x)$. This operation corresponds to "$q(j, x_i) \neq$ NULL", "$j \leftarrow q(j, x_i)$", and "Output(j)" in the algorithm shown in the previous section. In the operation, first, we obtain the address of the hash table by computing $h(j, x)$. This address shows bucket $B_{h(j,x)}$ that has 8 entries. After that, we find whether a back-pointer $q(j, x)$ is stored in $B_{h(j,x)}$. As shown in the above, we can simultaneously read eight values in a bucket and the number of values in a bucket is read from the number table to read valid data. Since each entry in the hash table has the values of j and x, we can find it by comparing j and x read from the hash table with input values j and x. Therefore, we can check at most 8 entries in $B_{h(j,x)}$ at the same time. After comparing, if $q(j, x)$ is found, output it. Otherwise, we check whether $B_{h(j,x)}$ is full or not. If $|B_{h(j,x)}| < 8$, that is, $B_{h(j,x)}$ is not full, we can find $q(j, x)$ does not exist in the hash table and output NULL. If not, we perform the above operation for bucket $B_{(h(j,x)+1)\wedge 0\text{x}3\text{FF}}$ for $i = 1, 2, \ldots$ until we find whether $q(j, x)$ is stored or not.

Add operation: This operation is to add pointer j, character x, and back-pointer $q(j, x)$ to the hash table. It is performed in AddTable operation as shown in the algorithm of the previous section. Indeed, it is performed after the find operation. More specifically, this operation is used when $q(j, x)$ does not exist in the hash table and we

already know an entry in which $q(j, x)$ can be stored. The entry to be stored locates in the bucket which was referred last in the find operation. Therefore, according to the result of the find operation, we add j, x and $q(j, x)$ to the hash table and increment the corresponding number of stired values in the number table.

In order to implement the hash table, we use block RAMs configured as dual-port mode [60] as illustrated in Figure 2.2. Each of the number table consists of one 18k-bit block RAMs. Also, two 18k-bit block RAMs are assigned to one of the 8 tables in the data table. Since we use two tables for the number table, eighteen 18k-bit block RAMs are used in total. For the number table, its dual-port is used as reading port and writting port. They are used to perform the find and add operations, respectively. The reading port is to refer the number of stored values. The writing port is to clear all the values to zero for the initialize operation and to increment the number of stored values located in address $h(j, x)$ for the add operation. On the other hand, for the data table, we also use the dual-port as reading port and writing port for each. The reading port is to refer the number of stored values in the find operation. The writing port is to add a value of back-pointer for the add operation. To reduce the clock cycles, we always suppose that for input string of characters $x_0, x_1, \ldots, x_{n-1}$, the condition $q(j, x_i) = \text{NULL}$ is satisfied. More specifically, if $q(j, x_i) = \text{NULL}$, the next value of j for x_{i+1} is $c^{-1}(x_i)(= x_i)$, which shows that the next value of j depends only on x_i when the condition is true. Using this, we can continuously input characters unless the condition $q(j, x_i) = \text{NULL}$ is not satisfied. When the condition is not satisfied, we need to wait to input the next character.

5.5 FPGA architecture for LZW decompression

This section describes our FPGA architecture of the hardware LZW decompression algorithm using block RAMs in Xilinx Virtex-7 FPGA.

In thissection, we focus on the decompression of LZW-compressed data in a TIFF image file. Figure 5.5 shows the proposed architecture of LZW decompression. In our implementation, the LZW-based module decompresses all codes in a segment that are given one by one. In order to implement pointer table p, character table C_f, code buffer and output buffer o, the block RAMs are used. The block RAMs are configured as dual-port mode [60] as shown in Figure 2.2. A dual-port block RAM has two sets of ports which work independently. We use these two port to perform executions in three steps described in Section 5.2 in parallel. For table p, as shown in the previous section, since the values of $p(i)$ $(0 \leq i \leq 255)$ are initialized to NULL and codes 256 and 257 are reserved codes, we do not actually use the block RAM in that range to reduce the memory resources as illustrated in Figure 5.6. Instead of use of the block RAM, the module checks the value of the address. Namely, if the address is in $[0, 255]$, NULL is output. Otherwise the value of $p(i)$ is read from the block RAM. For the same reason, table C_f do not use the block RAM in the range. Each value of $C_f(i)$ $(0 \leq i \leq 255)$ is initialized to an alphabet $\alpha(i)$. From the target application, we can assume that $\alpha(i) = i$ $(0 \leq i \leq 255)$ since the alphabets correspond to pixel values. If the address is in $[0, 255]$, the value of the address is just output.

We can obtain a string of each code by traversing tables p and C_f. To store the characters, an output buffer o is used. Output buffer o is also configured as dual-port block RAMs. Since the characters of corresponding string of a code is reversely read out from table C_f and then written to the output buffer for each character in reversed order,

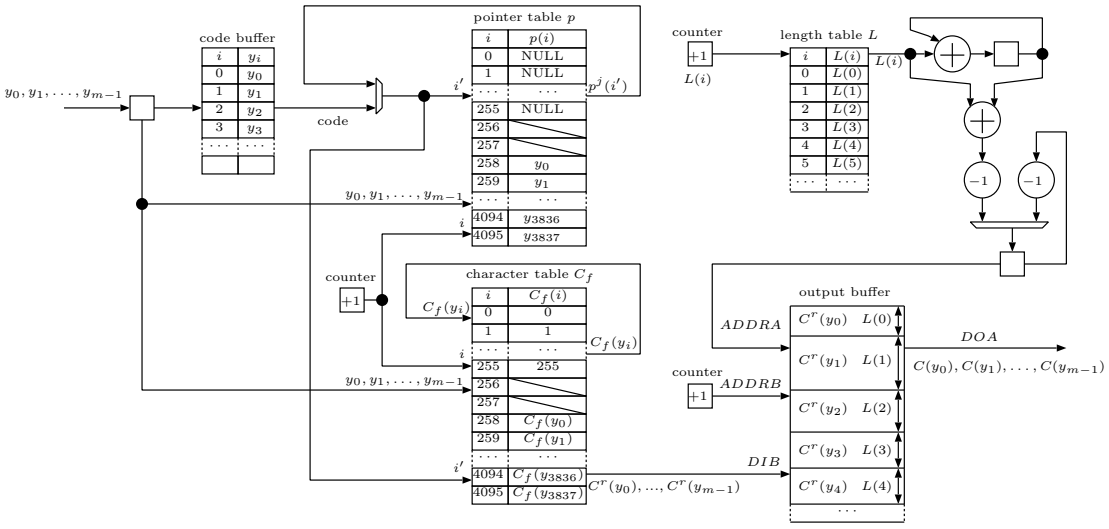


Figure 5.5: The outline of our FPGA architecture for hardware LZW algorithm

we use table L to store the length of the string to reverse it in the following step. Finally, we read the characters from output buffer and reverse it with the length. Indeed, it is not necessary to store all the values of L since the executions of three steps described in Section 5.2. Therefore, table L is configured as a FIFO (First-In-First-Out). As the same reason, we use a FIFO, which is also composed of the block RAMs, to temporally store input codes called code buffer. For the reader's benefit, the behavior of the proposed architecture for each step is described next.

Step 1: In Step 1, for tables p and C_f, one port set of the dual-port block RAMs is used to perform the updating operation as described in the algorithm in Section 5.2, respectively. The table update is executed for given compressed codes y_i $(0 \leq i \leq m-1)$ one by one. If an input code $y_i \leq 257$ holds, it is unnecessary to update both tables since the elements in p and C_f are constant values for $i \leq 257$. Otherwise, if $y_i \geq 258$, table p is updated by writing y_i to $p(i+258)$. The update for table p can be easily done since y_i is stored to an element at address i in the block RAM as illustrated in Figure 5.6. Also, the

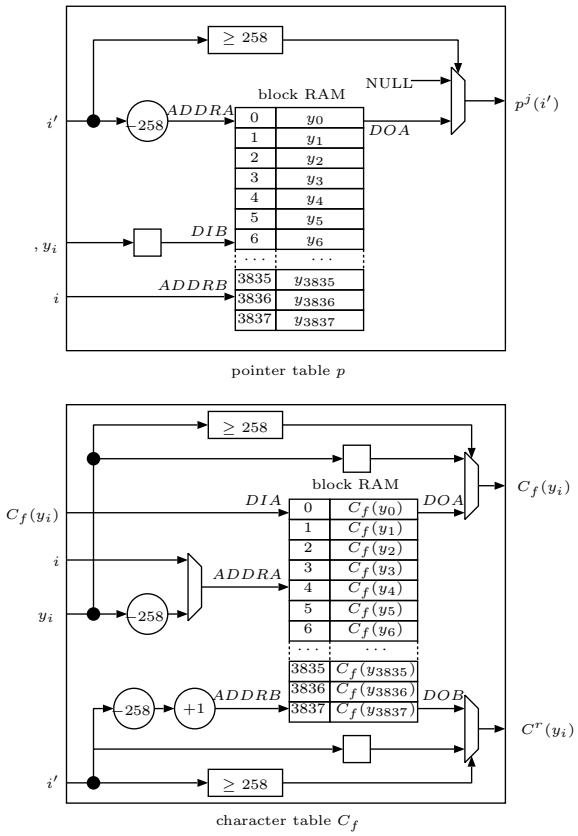


Figure 5.6: Dual-port block RAM and memory configurations of tables p and C_f

update operation for table C_f is performed. It takes 2 clock cycles to read a value stored at $C_f(y_i - 258)$ and write it to $C_f(i)$. The above operations are repeatedly executed for each input code. Since the update operations for both tables can be executed at the same time, it takes two clock cycles for each input code. Since m codes are input, in total, $2m$ clock cycles are necessary to perform Step 1. Recall that all each code segment has 3838 codes except the last one. For each code segment that has 3838 codes, table p and C_f are full if the update operations for all codes of one code segment are performed. The update operation is terminated until all codes of this segment are decompressed. For the last code segment, if code 257 (EndOfInformation) is reached, the update operation is

terminated until all codes of the last code segment are decompressed.

Step 2: We will show how to obtain partially-reversed strings $C^r(y_i)$ $(0 \leq i \leq m-1)$ with table p and C_f updated in Step 1. In the following, we use another port set of the dual-port block RAMs of tables p and C_f, respectively. As shown in the algorithm of Step 2 in the previous section. for each input code $y_i (0 \leq i \leq 3837)$, we traverse tables p and output characters of $C^r(y_i)$ in table C_f. Since it takes one clock cycle to read an element in tables p and $C^r(y_i)$, respectively, two clock cycles are necessary to output a character in $C^r(y_i)$. However, the access to tables p and C_f can be performed currently. We can overlap the access for an input code y_i with that for the next code y_{i+1}. Therefore, when the length of $C^r(y_i)$ is $L(i)$, we can output a string $C^r(y_i)$ in $L(i) + 1$ clock cycles. The characters of $C^r(y_i)$ are stored into an output buffer o one by one. Also, $L(i)$ is counted at the same time. After outputting the characters of $C^r(y_i)$, and $L(i)$ is stored to table L which is composed of a dual-port block RAM. Since it takes $L(i) + 1$ clock cycles to output for each input code y_i, Step 2 is performed in $\Sigma_{i=0}^{m-1}(L(i) + 1)$ clock cycles in total.

Step 3: In Step 3, partially-reversed strings $C^r(y_0)$, $C^r(y_1)$,..., $C^r(y_{m-1})$, stored in output buffer o in Step 2 is reordered to the uncompressed strings $C(y_0)$, $C(y_1)$,..., $C(y_{m-1})$. Since the length of each string is known from $L(i)$, each character can be read reversely from output buffer o one by one. Each operation for an input code y_i can be started after $C^r(y_i)$ is stored to output buffer o, that is, $L(i)$ is stored into table L. It takes $L(i) + 1$ clock cycles to perform the operation for a code y_i since one clock cycle for reading $L(i)$ and L(i) clock cycles for reversely reading $C(y_i)$ are necessary.

Let us consider the overlapped execution among the three steps as illustrated in Figure 5.2. Recall that Step 1 can be performed in 2 clock cycles for each input code.

"Crafts" "Flowers" "Graph"

Figure 5.7: Three gray scale images with 4096×3072 pixels used for experiments

The operation for an input code y_i ($0 \leq i \leq m-1$) in Step 2 can be performed after the operation for the next code y_{i+1} in Step 1 is finished. Also, the execution time for each y_i is at least 2 clock cycles since $L(i) + 1 \geq 2$. Therefore, the execution of Step 2 can be started 4 clock cycles later after the first code y_0 is given in Step 1 and performed continuously. In Step 3, the operation for an input code y_i can be performed after the operation for y_i in Step 2. Namely, the operation for y_i in Step 3 can be executed when the operation for y_j ($y_j \geq i+1$) in Step 2. Therefore, in Step 3, the execution sometimes waits for the execution in Step 2. Let us consider the longest case for computing time that an input data obtained by compressing data whose elements are the same value is given. For example, when a string $X = 0, 0, 0, \ldots$ is compressed, the compressed data is $Y = 0, 258, 259, \ldots$. The length $L(i)$ of each uncompressed string $C(y_i)$ can be represented as $L(i) = i + 1$ ($0 \leq i \leq m-1$) since the lengths are incremented by one for each code. Since $L(i+1) = L(i) + 1$ in this case, $L(i) < L(i+1)$ always holds. Therefore, the execution for y_i in Step 3 can be performed when that for y_{i+1} is performed concurrently. Moreover, the execution for each y_i in Step 3 waits for one clock cycle. In such case, it takes $\Sigma_{i=0}^{m-1}(L(i)+2) = m(m+5)/2$ clock cycles. Since the execution of Step 2 can be started 4 clock cycles later after the first code y_0 is given in

Step 1 and Step 3 can be started 2(= $L(0) + 1$) clock cycles later after the execution of Step 2 is started, Step 3 can be started 6 clock cycles later after the first code y_0 is given in Step 1. Therefore, in such case, it takes $m(m + 5)/2 + 6$ clock cycles to perform the LZW decompression in total.

5.6 Experimental results

This section shows the implementation results of the implementations of LZW compression and decompression algorithms in the FPGA.

First, to evaluate the performance of the proposed architecture of LZW compression, we have implemented it in VC707 board [61] equipped with the Xilinx Virtex-7 FPGA XC7VX485T-2. The experimental results of the implementation is shown in Table 5.5. We also use Intel Core CPU i7-4790 (3.6GHz) to evaluate the running time of sequential LZW compression. We have used three gray scale images with 4096×3072 pixels as shown in Figure 5.7, which are converted from JIS X 9204-2004 standard color image data. Table 5.6 shows the time of compression on CPU and FPGA and the compression ratio (original image size : compressed image size). The image "Graph" has high compression ratio since it has large areas with similar intensity levels. The image "Crafts" has small compression ratio since it has small details.

In the CPU implementation, we use the back-pointer table q of which size is $2^{20} \times$ 12bits = 1.5MBytes without a hash table. The table is composed of a two-dimensional array of $2^{12} \times 2^8$ elements. Each element has 32 bits that can be used to store the back-pointer $q(j, x)$. Since this table is not a hash table, no conflict occurs in the sequential implementation on the CPU. After ClearCode is output, we need to initialize the table. However, it is too costly to clear all elements in the table. Therefore, we use the time-

stamp technique as follows. Since the value of each $q(j, x)$ has 12 bits, the remaining 20 bits are used as a time stamp. The time stamp takes value from 0 to $2^{20} - 1 = 1048575$. Initially, the time stamp is 0 and incremented whenever ClearCode is output. When a back-pointer $q(j, x)$ is added in the table, the current time stamp is written with it. Using the time stamp, we can determine if the value stored in each entry is valid. When the time stamp is incremented 1048575 times, it is set to 0 and all entries of the table has to be reinitialized. However, in the compression for most images, the number of code segments is less than 1048575. Hence, in the sequential implementation on the CPU, for three images utilized in the experiment, we can perform the sequential algorithm without reinitializing the table.

Table 5.6 shows the time of compression on CPU and FPGA and the compression ratio. In our implementation on FPGA, to save the usage of block RAMs of FPGA, we use the hash table to implement the back-pointer table. Since we partition the data table of the hash table into 8 tables, we can check 8 elements of back-pointer table concurrently. For example, when "Crafts", "Flowers" and "Graph" are compressed, the average number per code segment of accessing the hash table, that is the average number of performing the add operation, is 3853.7, 3848.2 and 3916.4, respectively. Since the add operation is performed 3837 times per code segment excluding the last segment, we note that the conflict occurs. On the other hand, in the CPU implementation, there is no conflict because the hash table is not used. As shown in Table 5.6, for only one proposed module of LZW compression, the results show that implementation on FPGA is not faster than the implementation on the CPU. For example, in our FPGA implementation of single proposed module, it takes 18191909 clock cycles to compress image "Crafts", i.e., $\frac{18191909}{179.99\text{MHz}} = 101.07ms$. It takes 19426610 clock cycles to

compress image “Flowers”, i.e., $\frac{19426610}{179.99\text{MHz}} = 107.93ms$. To compress image “Graph”, it takes 24886071 clock cycles, i.e., $\frac{24886071}{179.99\text{MHz}} = 138.26ms$. However, since the proposed module uses very few FPGA resources, we have succeeded in implementing 24 identical LZW compression modules in an FPGA, where the frequency is 163.35MHz and each module has independent input/output (I/O) ports. Simply calculated, for image “Crafts”, our implementation with 24 modules runs up to 23.51 times faster than sequential LZW compression on a single CPU. As shown in Table 5.5, the implementation of 24 proposed modules uses 555 I/O ports, where more than one hundred of the remaining ports are dedicated to perform other communication protocols and can not be used as general I/O ports. Therefore, the number of available I/O ports of the FPGA is the bottleneck of our implementation. For a theoretical interest, we implement much more modules in the FPGA by assuming that the limitation of the number of I/O ports is ignored. The experimental results show that up to 110 proposed modules can be arranged in the FPGA. This implementation may not be used actually, but according to the results, for example, our proposed implementation with 110 modules for the image “Crafts” attains a speed-up factor of 89.63 over the sequential LZW compression on the CPU.

Table 5.5: Implementation results of one module and multiple modules of LZW compression algorithm

number of modules	1	24	110	Available
Slice Registers	104 (0.02%)	3120 (0.51%)	15977 (2.63%)	607200
Slice LUTs	346 (0.11%)	7782 (2.56%)	37249 (12.27%)	303600
18K-bit block RAMs	18 (0.87%)	432 (20.97%)	1980 (96.12%)	2060
I/O	26 (3.71%)	555 (79.29%)	—	700
Clock frequency [MHz]	179.99	163.35	135.87	—

For gray scale image “Graph” which has high compression ratio with 4096×3072 pixels, the proposed module of LZW compression compresses $4096 \times 3072 \times 1$Byte original data in $138.26ms$, that is, the compression throughput of the proposed module is $\frac{4096\times3072\times1\text{Byte}}{138.26ms} = 86.79$MBytes/s. On the other hand, for gray scale image “Crafts” which has low compression ratio, the compression throughput is $\frac{4096\times3072\times1\text{Byte}}{101.07ms} = 118.73$MBytes/s.

Table 5.6: Computing time (milliseconds) of LZW compression for three images

images	compression ratio	time of CPU	time of FPGA	Speedup ratio
“Crafts”	1.43:1	109.10	101.07	1.08:1
“Flowers”	1.72:1	93.60	107.93	0.87:1
“Graph”	36.72:1	46.79	138.26	0.34:1

Table 5.7: Implementation results of one module and multiple modules of hardware LZW decompression algorithm

number of modules	1	34	150	Available
Slice Registers	278 (0.05%)	9537 (1.57%)	40642 (6.69%)	607200
Slice LUTs	307 (0.1%)	10361 (3.41%)	45194 (14.89%)	303600
18K-bit block RAMs	13 (0.63%)	442 (21.46%)	1950 (94.66%)	2060
I/O	26 (3.71%)	564 (80.57%)	—	700
Clock frequency [MHz]	301.02	263.92	245.4	—

Next, we have also implemented the proposed architecture for hardware LZW decompression algorithm and evaluated it in VC707 board [61]. The experimental results of the implementation is shown in Table 5.7. We also use Intel Xeon CPU E5-2430 (2.2GHz) to evaluate the running time of sequential LZW decompression for the three gray scale images above. Table 5.8 shows the time of decompression on CPU and

FPGA. In LZW decompression on CPU, the operation of creating dictionary tables occupies most of the computing time. In our implementation on FPGA, the operation of creating tables is performed independently, and writing characters to output buffer and reading characters from output buffer are paralleled, hence, the operation of outputting characters occupies most of the time. As shown in Table 5.8, for only one proposed module, the results show that implementation on FPGA is 2.16 times faster than the implementation on the CPU. For example, in our FPGA implementation of one proposed module, it takes 19674631 clock cycles to decompress image "Crafts", i.e., $\frac{19674631}{301.02\text{MHz}} = 65.36ms$. It takes 18339574 clock cycles to decompress image "Flowers", i.e., $\frac{18339574}{301.02\text{MHz}} = 60.924ms$. To decompress image "Graph", it only takes 12892927 clock cycles, i.e., $\frac{12892927}{301.02\text{MHz}} = 42.831ms$. Hence, for gray scale image "Graph" which has high compression ratio with 4096×3072 pixels, the LZW decompression module outputs 4096×3072×1 Bytes of original data in $42.83ms$. Therefore, the decompression throughput of module is $\frac{4096\times3072\times1\text{Bytes}}{42.831ms} = 280.17\text{MBytes/s}$. Since the decompression throughput depends on input data, the decompression throughput can be even better if the compression throughput of input file is higher. Suppose that in the worst case for computing time, 4096×3072 input codes are given, all of which corresponding strings include 1 character. Since it takes 2 clock cycles to decompress each code that includes only 1 character, all the codes can be decompressed in $4096 \times 3072 \times 2 = 25165824$ clock cycles, i.e., $\frac{25165824}{301.02\text{MHz}} = 83.602ms$. More specifically, the minimum decompression throughput of proposed module is $\frac{4096\times3072\times1\text{Byte}}{83.602ms} = 143.54\text{MByte/s}$. Since the proposed FPGA module uses a few resources of the FPGA, we have succeeded in implementing 34 LZW decompression modules in a FPGA, where each module has independent I/O port. Since input characters are transferred to the proposed module every two clock

cycles, two modules can share one set of input port. Our implementation with 34 modules runs up to 64.39 times faster than sequential LZW decompression on a single CPU. Similarly, we also have succeeded in implementing 150 proposed modules, where we assume that the limitation of the number of I/O ports is ignored. The experimental results show that 150 proposed modules can be arranged in the FPGA.

Table 5.8: Computing time (milliseconds) of LZW decompression for three images

images	compression ratio	time of CPU	time of FPGA	Speedup ratio
“Crafts”	1.43:1	141.534	65.36	2.16:1
“Flowers”	1.72:1	127.136	60.924	2.08:1
“Graph”	36.72:1	75.901	42.831	1.77:1

5.7 Concluding remarks

We have presented hardware architectures of LZW compression and decompressing, and implement them in a Virtex-7 family FPGA XC7VX485T-2, respectively. According to the implementation results, the implementation of 24 modules of LZW compression attains a speed-up factor of 23.51 times faster than a sequential software implementation on a single CPU. On the other hand, the implementation of 34 modules of LZW decompression runs up to 64.39 times faster than the software implementation on the CPU.

Chapter 6

Conclusions

Ability of parallel processing is one of the most important features of FPGA. Moreover, recently, embedded multicore processors represented by FPGA has lately attracted considerable attention for their potential computation ability and power consumption. In this dissertation, we have presented several efficient FPGA implementations on the FPGA.

In Chapter 3, we have presented an efficient FPGA implementation of the Hough transform for lines detection on the Xilinx Virtex-6 FPGA XC6VLX240T-1. Our FPGA implementation uses 178 DSP slices and 180 block RAMs, and runs over 300 times faster than the sequential implementation on the CPU (Intel Xeon X7460 2.66GHz) for processing an 512×512 binary image of which all points are edge points. Then, we improved the proposed implementation to process pixel data given in raster scan order, and the usage of DSP slices reduces. Also, maximum filters are used to obtain the correct lines after voting operation. The improved implementation used only 90 DSP slices and 181 block RAMs and attains a speed-up factor of more than 38 over the sequential implementation on the CPU for processing an 512×512 binary image

with 33232 edge points. If all the points of an $512 \times 512 = 262144$ image are edge pints, our improved implementation also runs over 300 times faster than the sequential implementation on the CPU. Next, as one of the efficient improvements to the Hough transform for line detection, we presented an efficient FPGA implementation of the gradient-based Hough transform, where the gradient direction and magnitude of each pixel are used to reduce the useless voting operation to obtain more precise straight lines. The implementation of the gradient-based Hough transform uses only 13 DSP slices and runs 309 times faster over the sequential implementation on the CPU for processing an 333×333 gray scale image. On the other hand, we presented an efficient implementation of the Hough transform for circles detection on the Xilinx Virtex-7 FPGA XC7VX485T-2, that uses only one-dimensional parameter spaces. Our implementation for circles detection uses 398 DSP slices and 309 block RAMs, and attains a speed-up factor of 189 over the sequential implementation on the CPU.

In Chapter 4, we proposed an efficient processor core that executes the Euclidean algorithm computing the GCD of two large numbers in Xilinx Virtex-7 FPGA XC7VX485T-2, that uses only one DSP slice and one block RAM. Since the proposed GCD processor core uses very few resources, we have succeeded in implementing 1280 GCD processor cores in the FPGA. The implementation of 1280 GCD processor cores runs 3.8 times faster than the best GPU implementation and 316 times faster than a sequential implementation on the CPU.

In Chapter 5, we proposed a hardware architecture of LZW compression and decompression, respectively. We have succeeded in implementing 24 modules of LZW compression and 34 modules of LZW decompression in an Xilinx Virtex-7 FPGA XC7VX485T-2, respectively. According to the experimental results, our implementation of 24 LZW

compression modules runs 23.51 times faster than a sequential implementation on the CPU (Intel Core CPU i7-4790 3.6GHz). On the other hand, our implementation of 34 LZW decompression modules runs 64.39 times faster than a sequential implementation on the same CPU.

References

[1] Adobe Developers Association. *TIFF Revision 6.0*, June 1992.

[2] Y. Ago, Y. Ito, and K. Nakano. An FPGA implementation for neural networks with the fdfm processor core approach. *International Journal of Parallel, Emergent and Distributed Systems*, 28(4):308–320, 2013.

[3] Y. Ago, K. Nakano, and Y. Ito. A classification processor for a support vector machine with embedded DSP slices and block RAMs in the FPGA. In *Proc. of IEEE 7th International Symposium on Embedded Multicore Socs (MCSoC)*, pages 91–96, 2013.

[4] Altera Corp. *Stratix V Device Handbook*, 2012.

[5] H. Bessalah, S. Seddiki, F. Alim, and M. Bencherif. On line mode incremental Hough transform implementation on Xilinx FPGA's. In *Proc. of the 8th conference on Signal, Speech and image processing*, pages 176–179, 2008.

[6] J. Canny. A computational approach to edge detection. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, (6):679–698, 1986.

[7] E. R. Davies. Circularity – a new principle underlying the design of accurate edge orientation operators. *Image and Vision Computing*, 2(3):134–142, 1984.

[8] D. D. S. Deng and H. ElGindy. High-speed parameterisable Hough transform using reconfigurable hardware. In *Proc. of the Pan-Sydeny area workshop on Visual information processing*, volume 11, pages 51–57, 2001.

[9] R. Devi, J. Singh, and M. Singh. VHDL implementation of GCD processor with built in self test feature. *International Journal of Computer Applications*, 25(2):50–54, July 2013.

[10] O. Djekoune and K. Achour. Incremental Hough transform: an improved algorithm for digital device implementation. *Real-Time Imaging*, 10(6):351–363, 2004.

[11] R. O. Duda and P. E. Hart. Use of the Hough transformation to detect lines and curves in pictures. *Communications of the ACM*, 15(1):11–15, 1972.

[12] A. Elhossini and M. Moussa. Memory efficient FPGA implementation of Hough transform for line and circle detection. In *Electrical & Computer Engineering (CCECE), 2012 25th IEEE Canadian Conference on*, pages 1–5. IEEE, 2012.

[13] N. Fujimoto. High throughput multiple-precision GCD on the CUDA architecture. In *International Symposium on Signal Processing and Information Technology*, pages 507–512, Dec 2009.

[14] T. Fujita, K. Nakano, and Y. Ito. Bulk GCD computation using a GPU to break weak RSA keys. In *International Parallel and Distributed Processing Symposium Workshops*, May 2015.

[15] S. Funasaka, K. Nakano, and Y. Ito. Fast LZW compression using a GPU. In *in Proc. of International Symposium on Computing and Networking*, pages 303–308, 2015.

[16] S. R. Geninatti, J. I. B. Benítez, M. Calvio, N. G. Mata, and J. G. Luna. FPGA implementation of the generalized Hough transform. In *Reconfigurable Computing and FPGAs, 2009. ReConFig'09. International Conference on*, pages 172–177. IEEE, 2009.

[17] A. Goneid, S. El-Gindi, and A. Sewisy. A method for the Hough transform detection of circles and ellipses using a 1-dimensional array. In *Systems, Man, and Cybernetics, 1997. Computational Cybernetics and Simulation., 1997 IEEE International Conference on*, volume 4, pages 3154–3157. IEEE, 1997.

[18] K. Hashimoto, Y. Ito, and K. Nakano. Template matching using DSP slices on the FPGA. In *Proc. of International Symposium on Computing and Networking*, pages 338–344, 2013.

[19] Helion Technology. *LZRW3 Data Compression Core for Xilinx FPGA*, October 2008.

[20] P. V. C. Hough. Method and means for recognizing complex patterns. U.S. Patent 3,069,654, 1962.

[21] J. Illingworth and J. Kittler. A survey of the Hough transform. *Computer Vision, Graphics, and Image Processing*, 44(1):87–116, 1988.

[22] D. Ioannou, W. Huda, and A. F. Laine. Circle recognition through a 2D Hough transform and radius histogramming. *Image and vision computing*, 17(1):15–26, 1999.

[23] J. R. Jen, M. C. Shie, and C. Chen. A circular Hough transform hardware for industrial circle detection applications. In *Industrial Electronics and Applications, 2006 1ST IEEE Conference on*, pages 1–6. IEEE, 2006.

[24] S. M. Karabernou and F. Terranti. Real-time FPGA implementation of Hough transform using gradient and CORDIC algorithm. *Image and Vision Computing*, 23(11):1009–1017, 2005.

[25] H.-S. Kim and J.-H. Kim. A two-step circle detection algorithm from the intersecting chords. *Pattern recognition letters*, 22(6):787–798, 2001.

[26] S. T. Klein and Y. Wiseman. Parallel Lempel Ziv coding. *Discrete Applied Mathematics*, 146(2):180–191, 2005.

[27] D. E. Knuth. *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*. Addison-Wesley, 1997.

[28] S. D. Kohale and R. W. Jasutkar. Power optimization of GCD processor using low power Spartan 6 FPGA family. *International Journal of Conceptions on Electronics and Communication Engineering*, 2(1):1–6, June 2014.

[29] Z. Kulpa. On the properties of discrete circles, rings, and disks. *Computer graphics and image processing*, 10(4):348–365, 1979.

[30] P. Lee and A. Evagelos. An implementation of a multiplierless Hough transform on an FPGA platform using hybrid-log arithmetic. In *Proc. of Real-Time Image Processing 2008*, volume 6811, pages 68110G–1, 2008.

[31] M. Lin. A hardware architecture for the LZW compression and decompression algorithms based on parallel dictionaries. *Journal of VLSI signal processing systems for signal, image and video technology*, 26(3):369–381, 2000.

[32] M. Lin, J. Lee, and G. E. Jan. A Lossless Data Compression and Decompression Algorithm and Its Hardware Architecture. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 14(9):925–936, 2006.

[33] Micron Inc. *1GB, 2GB, 4GB (x64, SR) 204-Pin DDR3 SODIMM*, May 2015.

[34] M. K. Mishra, T. K. Mishra, and A. K. Pani. Parallel Lempel-Ziv-Welch (PLZW) technique for data compression. *International Journal of Computer Science and Information Technologies*, 3(3):4038–4040, 2012.

[35] K. Nakano and E. Takamichi. An image retrieval system using FPGAs. *IEICE Transactions on Information and Systems*, E86-D(5):811–818, May 2003.

[36] K. Nakano and Y. Yamagishi. Hardware n choose k counters with applications to the partial exhaustive search. *IEICE Transactions on Information and Systems*, E88-D(7), 2005.

[37] S. Navqi, R. Naqvi, R. A. Riaz, and F. Siddiqui. Optimized RTL design and implementation of LZW algorithm for high bandwidth applications. *PRZEGLAD ELEKTROTECHNICZNY (Electrical Review)*, 4:279–285, 2011.

[38] M. S. Nixon and A. S. Aguado. *Feature Extraction and Image Processing*. Academic Press, second edition, 2008.

[39] F. O'Gorman and M. Clowes. Finding picture edges through collinearity of feature points. *Computers, IEEE Transactions on*, C-25(4):449–456, 1976.

[40] S. Prakash, M. Purohit, and A. Raizada. A novel approach of speedy-highly secured data transmission using cascading of PDLZW and arithmetic coding with cryptography. *International Journal of Computer Applications*, 57(19), 2012.

[41] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21:120 – 126, 1978.

[42] K. Scharfglass, D. Weng, J. White, and C. Lupo. Breaking weak 1024-bit RSA keys with CUDA. In *Internatinal Conference of Breaking weak 1024-bit RSA keys with CUDA*, pages 207 – 212, Dec 2012.

[43] J. L. Shafer, H. Ngo, and R. W. Ives. Using an FPGA to accelerate pupil isolation in iris recognition. In *Signals, Systems and Computers (ASILOMAR), 2010 Conference Record of the Forty Fourth Asilomar Conference on*, pages 1774–1777. IEEE, 2010.

[44] K. Shyni and K. V. M. Kumar. Lossless LZW data compression algorithm on CUDA. *IOSR Journal of Computer Engineering*, pages 122–127, 2013.

[45] S. Tagzout, K. Achour, and O. Djekoune. Hough transform algorithm for FPGA implementation. *Signal Processing*, 81(6):1295–1301, 2001.

[46] Y. Tokunaga and T. Inoue. A method for circular pattern recognition in a binary image and its implementation onto an FPGA. *IEICE TRANSACTIONS on Fundamentals of Electronics, Communications and Computer Sciences*, 82(2):246–254, 1999.

[47] T. A. Welch. A technique for high-performance data compression. *IEEE Computer*, 17(6):8–19, June 1984.

[48] J. R. White. *PARIS: A PARALLEL RSA-PRIME INSPECTION TOOL*. PhD thesis, California Polytechnic State University - San Luis Obispo, June 2013.

[49] Xilinx Inc. *Virtex-4 FPGA User Guide(v2.6)*, 2008.

[50] Xilinx Inc. *Virtex-5 FPGA User Guide(v5.2)*, 2009.

[51] Xilinx Inc. *Virtex-6 FPGA DSP48E1 Slice User Guide (v1.3)*, 2011.

[52] Xilinx Inc. *Virtex-6 FPGA Memory Resources User Guide (v1.6)*, 2011.

[53] Xilinx Inc. *Virtex-6 Family Overview (v2.4)*, 2012.

[54] Xilinx Inc. *Virtex-6 FPGA Configurable Logic Block User Guide (v1.2)*, 2012.

[55] Xilinx Inc. *7 Series FPGAs Configuration User Guide*, 2013.

[56] Xilinx Inc. *7 Series FPGAs Overview (v1.14)*, 2013.

[57] Xilinx Inc. *LogiCORE IP CORDIC v6.0*, 2013.

[58] Xilinx Inc. *7 Series DSP48E1 Slice User Guide (v1.8)*, 2014.

[59] Xilinx Inc. *7 Series FPGAs Configurable Logic Block User Guide (v1.7)*, 2014.

[60] Xilinx Inc. *7 Series FPGAs Memory Resources User Guide (v1.11)*, 2014.

[61] Xilinx Inc. *VC707 Evaluation Board for the Virtex-7 FPGA*, Sept. 2015.

[62] J. Ziv and A. Lempel. A Universal Algorithm for Sequential Data Compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977.

[63] J. Ziv and A. Lempel. Compression of Individual Sequences via Variable-Rate Coding. *IEEE Transactions on Information Theory*, 24(5):530–536, 1978.

Acknowledgment

First and foremost, I would like to show my deepest gratitude to my supervisor, Professor Koji Nakano for his continuous encouragement, advice and support. He is a respectable and resourceful scholar. Without his enlightening instruction and patience, I can not complete my Ph.D. study. As a supervisor, he taught me skills and enlightened in my future research career.

I shall express sincere appreciation to my thesis committee members, Associate Professor Yasuaki Ito and Professor Takio Kurita for reviewing my dissertation.

I would also like to express my thanks to Assistant Professor Daisuke Takafuji for his support and continuous guidance in every stage of my study. I shall extend my thanks to all members of computer system laboratory. They were always very keen to help.

I would express thanks to all the faculty members of the Department of Information Engineering of Hiroshima University. I also would like to express thanks to the Japanese Government for financial support to complete my study in Japan.

I wish to express my thanks to my family who always supported me. I also would like to express my heartiest thanks to my wife Fang Hu for waiting me four years to complete my study. Last, I would express my special thanks to my unborn child since you encouraged me.

List of publications

Journals

[J-1] Xin Zhou, Norihiro Tomagou, Yasuaki Ito, and Koji Nakano, Implementations of the Hough Transform on the Embedded Multicore Processors, International Journal of Networking and Computing, Vol. 4, No. 1, pp. 174-188, January 2014.

[J-2] Xin Zhou, Koji Nakano, and Yasuaki Ito, Efficient Implementation of FDFM Approach for Euclidean Algorithms on the FPGA, International Journal of Networking and Computing, to appear.

International Conferences

[C-1] Xin Zhou, Yasuaki Ito, and Koji Nakano, An Efficient Implementation of the Hough Transform using DSP slices and block RAMs on the FPGA, Proc. of the IEEE 7th International Symposium on Embedded Multicore SoCs (MCSoC), pp. 85-90, September 2013.

[C-2] Xin Zhou, Yasuaki Ito, and Koji Nakano, An Efficient Implementation of the Gradient-based Hough Transform using DSP slices and block RAMs on the FPGA, Proc. of International Parallel and Distributed Processing Symposium Workshops, pp. 762-770, May 2014.

[C-3] Xin Zhou, Yasuaki Ito, and Koji Nakano, An Efficient Implementation of the One-Dimensional Hough Transform Algorithm for Circle Detection on the FPGA, Proc. of International Symposium on Computing and Networking, pp. 447-452, December 2014.

[C-4] Xin Zhou, Koji Nakano, and Yasuaki Ito, Parallel FDFM Approach for Computing GCDs Using the FPGA, Proc. of International Conference on Parallel Processing and Applied Mathematics (PPAM 2015, LNCS 9573), pp. 238-247, September 2015.

[C-5] Xin Zhou, Yasuaki Ito, and Koji Nakano, An Efficient Implementation of LZW Decompression in the FPGA, Proc. of International Parallel and Distributed Processing Symposium Workshops, pp. 599-607, May 2016.