Imprecise Arithmetic for Low Power Signal Processing

Tobias N. Jeppe

Kongens Lyngby 2012

Technical University of Denmark
Department of Informatics and Mathematical Modelling
Building 321, DK-2800 Kongens Lyngby, Denmark
Phone +45 45253351, Fax +45 45882673
reception@imm.dtu.dk
www.imm.dtu.dk

Summary

This project will investigate imprecise arithmetic operations, which can result in circuits dissipating significant less power at the expense of errors tolerated by many applications in signal and image processing. The project will especially investigate addition and multiplication (the most common operators in signal processing), and the error that imprecise circuits introduce. Different implementation will be considered, evaluating their delay, power, area and error characteristics

Preface

This report was prepared at Department of Informatics and Mathematical Modelling, the Technical University of Denmark in partial fulfilment of the requirements for acquiring the Master degree in engineering.

The report deals with different aspects of imprecise arithmetic in addition and multiplication implementation. The main focus is on acquiring data on different implementation and find the scheme with the best balance between accuracy and power consumption.

This report is a summery of collected data from imprecise addition and multiplication schemes implemented in the spring and summer of 2012 during the master thesis, with the emphasis on error and power consumption.

Lyngby, August 2012

Tobias N. Jeppe

Contents

Chapter 1

Introduction

In some application the accuracy of a calculation is less important and the urge to save power drive people to approximate a result via software or hardware. This is specially the case with wearable gadgets such as media players, mobile phones ect. Decoding sound and images error free is important for the best result, but as most sound, images and videos are received/stored compressed, some loss of information has already been applied. Introducing errors to an already error-prone sound, image or video has little effect on the experience, especially if the receiver do not notice the error. In the case of unnoticeable errors, it is possible to save power by reducing the power consumption via software or use hardware with lower power consumption. This thesis is based on reducing the power consumption of hardware, that calculates an imprecise result which is good enough.

In January 2011 the MIT press released the article "The surprising usefullness of sloppy arithmetic" [LH11]. It describes how an algorithm commonly used in object-recognition, to separate foreground and background, where all numeric results were infused with a random error. Errors between ±1% would generate errors in around 14 pixels out of million, unnoticeable by the human eye. Knowing that an object-recognition algorithm for static pictures '*is considered good if it's right about half the times*', the errors introduced by using sloppy arithmetic to separate the foreground and background is minimal. The article also suggest that sloppy arithmetic can be used when interacting with humans, being the movement of a mouse pointer as the human "interface" intuitive compensate for the movement error or calculating 3D graphic. The great thing about sloppy arithmetic is the size of the circuit needed, which can be considerably smaller.

Even though a certain calculation precision is only obtainable using floating point numbers, the principle of sloppy arithmetic is pursued in the integer domain. In [MP10], the switching activity is reduced by freezing the least significant part of the input arguments to the adder and multiplier, to either 0 or a random number. The result clearly display a relation ship between the amount of input freezing and the power saving and error. The more bit frozen the bigger the error and power saving. The experiment were performed on custom hardware, designed for 1D filtering, but only to the degree of controlling the input registers.

What if fully sloppy circuits were to be used, meaning that the precision of the result always were questionable? This is pursued in the technical report [AN11], which is the foundation for this thesis. The report describes addition and multiplication with sloppy circuits. The idea being that the least significant part of the addition is calculated or more correct estimated without regards for carry information. The most significant part is calculated error free, with the exception of the missing carry from the least significant part. For multiplication, the "least significant" partial product is generated sloppy. It is shown that a sloppy adder can perform image smoothing, sharpening and edge-detection with some introduction of errors. Combining a sloppy multiplier and adder into a Multiplier and Accumulation Circuit (MAC), applied to a Inverse Discrete Cosinus Transformation (IDCT) application, shows power saving of 17%, a delay reduction of 17% and area saving of 10%, without doubling the error of the error free implementation.

This thesis will build on [AN11] and investigate imprecise arithmetic in depth. The report will describe imprecise adders and multiplier schemes. The imprecise schemes errors will be formalised as functions based on their performance and their accuracy in image processing will be investigated. Their performance will be compared in terms of area, delay and power, at a point where they perform identically, error wise. The investigation into sloppy arithmetic, will hopefully give a better understanding and knowledge of sloppy arithmetic and schemes, together with finding the best compromise between the performance parameters: Accuracy and power reduction.

1.1 Report Layout

Chapter 1 gives an superficial introduction to imprecise arithmetic and it uses and ends with a report layout describing how the report is constructed and the assumptions used in the report.
Chapter 2 starts the report with a short introduction to power dissipation and ways to reduce this. Then it will defined the errors used to measure the performance of imprecise schemes. The image processing routines is introduces, which is used to test the imprecise schemes together with a description of the tools used to synthesis and obtain the power dissipation of the different schemes. The chapter ends with the test pictures.
Chapter 3 introduces imprecise adders. Starting with a description of the respectable imprecise schemes. The statistically accuracy and the error introduced in image processing is investigated. A short discussion sums up the findings. The schemes are then implemented at a common precision and their area, delay and power consumption are compared to that of an error free implementation. The implementation result is then discussed and a conclusion summarising the error and implementation.
Chapter 4 introduces the imprecise multipliers. Start with a description of the respectable imprecise schemes. The statistically accuracy and the error introduced in image processing is investigated. A short discussion sums up the findings. The schemes are then implemented at a common precision and their area, delay and power consumption are compared to that of an error free implementation. The implementation result is then discussed and a conclusion summarising the error and implementation.
Chapter 5 introduces the Multiply and ACcumulation (MAC) units. The MAC's is comprised from the best performing imprecise multiplier and adders. The error it introduces in image processing is investigated. Based on their combined error, their implementation is compared by area, delay and power against an error free scheme. The third part end by discussion the result and summarising with a conclusion.
Chapter 6 concludes the report, summarises the findings for imprecise adders, imprecise multiplier and imprecise MAC's.

1.2 How to Read the Report

- Figures, tables and images which refers to a destination as "3.4" is placed in chapter 3 and is the fourth of its kind in that chapter.
- Figures, tables and images which refers to a destination as "E.3.4" is placed in Appendix E, section 3 and is the fourth of its kind in that appendix.
- [abc12] refer to other publications. The full list of references is collected in the Bibliography, page 79.

1.3 Project Assumptions

For data generation the following image processing has been applied: Transformation (IDCT) is described in 2.4.1. Smoothing uses a 5x5 Gaussian filter, described in appendix 2.4.2. Edge-detection is performed by sobel's algorithm, described in appendix 2.4.3.
The number system used in this report is two's complement system, if nothing else is mentioned. All adders used in this thesis is build as CPA using the two staged CLA as a basis, with a width of 32bit. All multipliers in the report is square 16 bit multipliers with a 32bit result, using the recoder scheme NRP3a [ZH03, p. 33]. A comparison between using Booth and NRP3a as recoding scheme can be found in 4.1.

CHAPTER 2

Background

2.1 Power Dissipation

The power dissipation of a CMOS circuit can be approximated by equation 2.1.

$$\mathrm{P}_{\mathrm{TOTAL}} = \underbrace{\sum_{i=1}^{N} \left(V_{DD}^2 C_{Li} + E_i^{int}\right) a_i f_{clk}}_{\text{Dynamic}} + \underbrace{\sum_{i=1}^{N} V_{DD} I_i^{leak}}_{\text{Static}} \tag{2.1}$$

For the dynamic part V_{DD} is the supply voltage, C_{Li} is the capacitive load connected to the gate output, E_i^{int} is the short-circuit current and the power dissipated by switching an internal node, a_i is the cell's switching activity and f_{clk} is the circuits clock frequency. The static contribution I_i^{leak} is the cell's leakage. The dynamic contribution is by far the largest in the 90 nm cell library used in this thesis, but static power must be accounted in deep sub-micro CMOS technologies as its contribution approaches the dynamic contribution. Equation 2.1 suggest a couple of ways to reduce the power consumption of a circuit. E^{int} and I^{leak} are technology dependent and cannot be changd without chancing the cell library. C_L which is basically the gates fan out can be changed, but is a low level optimization process primarily left to programs. The dynamic power dissipation is linear dependent on the switching activity and clock frequency. If power saving is of interest, the clock frequency has already been chosen as low as possible and can only be further lowered if a task can be performed with fewer clock cycles. Reducing the switching activity can be achieved by chancing the implementation schemes or disabling part of the logic [MP10]. Lowering V_{DD} reduces both the dynamic and the static power dissipation, but unfortunately also increases the circuit delay.

2.1.1 Reduce Switching Activity

An example of this is the reduction of switching activity in an adder caused by freezing the least significant part of the input. The logic used to calculate the least significant part of the result is kept in a steady state, as the adders input do not change, the switching activity of this logic is 0, while the unfrozen part of the adder still has a switching activity [MP10]. Another way to reduce switching activity is introducing pipe-lining. Each pipe-lining register acts as a signal barrier, placing a register after a circuit which is prone to introduce race conditions will prevent the race condition to carry over the register. Clock gating is another way to reduce the switching activity. The clock network in modern system uses a considerable amount of power, turning off the clock tree not alone saves the power dissipation of the clock tree, is also stops all switching activity for the logic that were provided with a clock from the given clock tree. Clock gating is primarily used on bigger logic blocks, which is being used frequently.

2.1.2 Voltage Scaling

Voltage scaling can be applied to a circuit if there is enough slack. Slack being the time from the circuit delay up to the timing constrains. Voltage scaling is a particular effective way of reducing the power dissipation as it influence both the dynamic and static part. The dynamic power dissipation is decreased quadratically while the static part is linear decreased.
If a circuit which barely obey the timing constraints placed upon it is pipelined with a single register at the exact middle of the circuit timing wise, the critical path of each part is half of the original, disregarding the setup, hold and propagation time for the pipeline register. Normally this is used to increase the frequency, giving a higher throughput, but if the frequency is kept the voltage can be lowered, still obeying the original circuits timing constrain. From the data provided in [MP10], a 30% decrease in supply voltage increases the delay with 50% but also decreases the power dissipation with 50%. Pipe-lining introduce latency, but for most application these can be hidden by other operations.

In this thesis, imprecise arithmetic is used to decrease the delay, thereby making it possible to lower the power supply voltage and reduce the power dissipation.

2.1.3 Reducing Clock Frequency

Reducing the clock also reduces a gates switching activity over time. In many cases where power is a factor the clock frequency is set as low as possible, reducing it further demands that the application to uses fewer clock cycles. As an example, a multiplier and adder is replaced by a MAC. For a sum function, a series of multiplications and additions are needed to obtain the final result. Each multiplication followed by an addition which uses 2N operations. Using a MAC which accumulates all multiplications uses N operations, half the time. This makes it possible to reduce the frequency and still meet the deadline. The frequency reduction can be calculated as

$$F_{new} = F_{org} \times \left(\frac{\text{Non MAC code}}{\text{Total code}} + \frac{\frac{\text{MAC code}}{\text{Total code}}}{2} \right) \tag{2.2}$$

Introducing a slower clock make it possible to scale the supply voltage as swell.

2.1.4 Leakage Power

The static power dissipations is low compared to the dynamic power consumption, but as the cell technology moves to sub-micron CMOS technology, the difference between the static and dynamic power dissipation decreases. Lowering the supply voltage is one way to reduce the leakage power, but this increases the circuit delay. Reducing the amount of gate area means implementing new schemes and this affect everything, which makes it less interesting for leakage reduction. Power gating is a schemes that turns off the power supply for certain logic blocks. As no power is delivered to the logic block, it cannot use any power. As the circuits which turns off the power supply is rather big to support the power needed and the turn on time is large, it is a scheme only applied to larger logic blocks which is seldomused.

2.2 Errors in Imprecise Schemes

The error caused by an imprecise calculation is given in equation 2.3. ϵ is the difference between the correct *result* and the result generated by the imprecise implementation, $result_{imprecise}$. The error, ϵ, for a single calculation is important for that specific calculation, but is not a manageable size for comparing different imprecise schemes. The reason being that it only describes a single calculation error leaving out all others possible number combinations and errors hereof. Comparing between different imprecise schemes, being adders or multipliers, the average error - $\bar{\epsilon}$, average absolute error - $|\bar{\epsilon}|$, min - ϵ_{min} and maximum error - ϵ_{max} and maximum absolute error - $|\epsilon|_{max}$, is far more usable as benchmarking tools, as they describe the overall performance of an imprecise scheme.

$$result = result_{imprecise} + \varepsilon \tag{2.3}$$

$\bar{\epsilon}$ describes the average error, defined in equation 2.4. $\bar{\epsilon}$ indicates if positive or negative errors are dominating and describes the average error which can be expected if many arbitrary numbers are added or multiplied separately together by an imprecise scheme. The error can be miss leading for a single operation, as positive and negative errors can cancel each other out given the impression of an imprecise scheme being error free.

$$\bar{\epsilon} = \frac{1}{z^2} \sum_{i=0}^{z} \sum_{j=0}^{z} O(i,j) - O_{imprecise}(i,j) \tag{2.4}$$

O being a multiplication or addition operation, $z = 2^{width} - 1$ describing the maximum value of the input, thereby given the error for an exhausted combination of all possible input.

$|\bar{\epsilon}|$ describes the average absolute error, defined in equation 2.5. $|\bar{\epsilon}|$ describes the size of error which can be expected if two arbitrary numbers are multiplied or added together by an imprecise scheme. It do not describe the sign of the expected error, but the distance between the exact and imprecise result.

$$|\bar{\epsilon}| = \frac{1}{z^2} \sum_{i=0}^{z} \sum_{j=0}^{z} |O(i,j) - O_{imprecise}(i,j)| \tag{2.5}$$

O being a multiplication or addition operation, $z = 2^{width} - 1$ describing the maximum value of the input, thereby given the error for an exhausted combination of all possible input.

ϵ_{min} describes the biggest negative error possible for an imprecise scheme to produce, defined in equation 2.6. ϵ_{max} describes the biggest positive error an imprecise scheme can produce, defined in equation 2.7. $|\epsilon|_{max}$ is the biggest error an imprecise can produced, being positive or negative, defined in equation 2.8.

$$\epsilon_{min} = min\left(O(i,j) - O_{imprecise}(i,j)\right) \qquad i,j \in \{0,\ldots,z\} \tag{2.6}$$

$$\epsilon_{max} = max\left(O(i,j) - O_{imprecise}(i,j)\right) \qquad i,j \in \{0,\ldots,z\} \tag{2.7}$$

$$|\epsilon|_{max} = max(\epsilon_{max}, -\epsilon_{min}) \tag{2.8}$$

2.3 Errors in Image Processing Caused by Imprecise Addition and Multiplication

Image processing are used to investigate the performance of imprecise adders and multipliers in applications. The imprecise adders and multipliers performance is expressed by the error they introduce in the final image compared to the same image processed on an error free platform. The error is the difference between the same positioned pixel value, defined in equation 2.9.

$$\epsilon^p_{(i,j)} = Pixel(i,j) - Pixel_{imprecise}(i,j) \tag{2.9}$$

But as the error $\epsilon^p_{(i,j)}$ only describes the error for a single pixel and not the entire image, a more general error definition is needed. The average image error - $\bar{\epsilon}^p$ and average absolute image error - $|\bar{\epsilon}|^p$ describes the average errors in the image and maximum absolute error - $|\epsilon|^p_{max}$ describes the biggest error in the image. The errors do not describe the application or the image alone, but the image processed by the application with the given imprecise adders and multipliers. Change one of the factors and the error can change more or less.

$\bar{\epsilon}^p$ describes the average pixel error in an image, defined in equation 2.10. The error indicates if positive or negative pixel errors are dominating. The error can be misleading, as positive and negative pixel errors can cancel each other out, given the impression that no pixel errors occurs. For pictures this means that $\bar{\epsilon}^p$ describes the image intensity changes, if $\bar{\epsilon}^p$ is positive the image are lighter, if negative the image are darker than the error free.

$$\bar{\epsilon}^p = \frac{1}{Width \times Height} \sum_{i=0}^{Width} \sum_{j=0}^{Height} \epsilon^p_{(i,j)} \tag{2.10}$$

$|\bar{\epsilon}|^p$ describes the average error per pixel, defined in equation 2.11. $|\bar{\epsilon}|$ describes the size of error which can be expected per pixel in the image. It do not describe the sign of the expected error, but the distance between the exact and imprecise pixel. If $|\bar{\epsilon}| \approx \bar{\epsilon}^p$ the prevailing pixel error is positive, $|\bar{\epsilon}| \approx -\bar{\epsilon}^p$ the prevailing pixel error is negative.

$$|\bar{\epsilon}|^p = \frac{1}{Width \times Height} \sum_{i=0}^{Width} \sum_{j=0}^{Height} |\epsilon^p_{(i,j)}| \tag{2.11}$$

$|\epsilon|^p_{max}$ describes the maximum pixel error in the picture, defined in equation 2.12. The average pixel error in the image is a good indicator for the image quality, but it do not describe how the errors are distributed, $|\epsilon|^p_{max}$ describes the biggest error in the image, this do not describe the error distribution either, but limits the error to a specific size.

$$|\epsilon|^p_{max} = max(|\epsilon^p_{(i,j)}|) \tag{2.12}$$

2.4 Image Processing Application

To test the performance of the different imprecise adder and multiplier schemes different image processing applications were used. Inverse Discrete Co-sinus Transformation - IDCT, found in one form or another in image and video applications. Image smoothing blurs the image and can be used to suppress pixel errors in an image. Edge-detection is used to distinguish element in object recognition systems.

2.4.1 IDCT Application

The Inverse Discrete Co-sinus Transformation - IDCT, is the inverse of DCT. DCT is used to transform a pixel block into an equivalent frequencies block. As high frequencies is less pronounced for the human eye they are removed making is possible to compress the image without loosing obvious quality. IDCT is used to transform a frequencies block into a pixel block.

DCT and IDCT is calculated by first taking the transform along one dimension and repeating the operation along the other direction, as explained in [KA06, p. 397]. By matrix terminology DCT is calculated by equation 2.13 and IDCT is calculated by EQ 2.14.

$$F = APA^T \tag{2.13}$$
$$P = A^T F A \tag{2.14}$$

P being the pixel block, F the frequency block and A is the transformation matrix given in equation 2.15.

$$A_{i,j} = \begin{cases} \sqrt{\frac{1}{8}} cos\frac{(2j+1)i\pi}{2N}, & i = 0, j = 0, 1, \ldots, 6, 7 \\ \sqrt{\frac{2}{8}} cos\frac{(2j+1)i\pi}{2N}, & i = 1, 2, \ldots, 7, j = 0, 1, \ldots, 6, 7 \end{cases} \tag{2.15}$$

For testing purposes, DCT is performed with floating points numbers and saved as integer numbers. To test how the IDCT application performed with imprecise adder and multiplier schemes, which is all integer based, the transformation matrix A, is represented as fixed point numbers. The transformation matrix contains only decimal numbers, which cannot be represented by integer numbers. Representing A as fixed point numbers by scaling the decimal part of the number, circumvents this. A is scaled $\frac{1}{2^{16}}$, meaning that the LSB representing $\frac{1}{2^{16}}$ the second LSB representing $\frac{1}{2^{15}}$ and so on, the values is basically left shifted 16 places. The frequency and pixel values are stored as integers, equivalent to fixed point numbers with a scaling factor of $\frac{1}{2^0}$.

Calculating P as in equation 2.14 with the transformation matrix scaled $\frac{1}{2^{16}}$ requires a multiplier, with an input width of atleast 16 bit for the first part $A^T X$, to represent the transformation matrix correct. The second part $(A^T X)A$ requires a multiplier with an input width of at least 24 bit, 16 bit $\times$ 8 bit. As this exceeds the multiplier width for this rapport, the first matrix multiplication ,AX, is right shifted 16 places, giving it a fixed point scaling of $\frac{1}{2^0}$. Now the second matrix multiplication can be performed on a multiplier with an input width of 16 bit. The result of $(A^T X)A$ is again a fixed point number with a scaling of $\frac{1}{2^{16}}$, because of the multiplication with the transformation matrix and is there for right shifted to again represent an integer value, for the pixels. This makes IDCT performed on integer hardware with the values in the transformation matrix scaled $\frac{1}{2^{16}}$, calculated as in equation 2.16.

$$P = (((A^T F) \times \tfrac{1}{2^{16}})A) \times \tfrac{1}{2^{16}} \tag{2.16}$$

It should be noted, that the performance between IDCT calculated with floating point numbers and as described in equation 2.16 are around $|\bar{\epsilon}|^p = 0.2$, given in table 2.1. The errors is rather consistent for all pictures, using the floating point method or integers method, which indicated that the application are picture independent to a certain degree. Figure 2.1 shows the test picture baboon

	$\lvert\bar{\epsilon}\rvert^p$				
IDCT using	Baboon	Barbara	Goldhill	Lena	Peppers
Floating point	0.58	0.57	0.56	0.57	0.57
Exact Integer	0.80	0.78	0.77	0.79	0.78

Table 2.1: The difference PSNR between IDCT calculated with Floating point numbers and 16 bit integers, using fixed point for the transformation matrix

	$\lvert\bar{\epsilon}\rvert^p_{(\text{Floating point}-\text{Fixed point})}$				
Image Smoothing	Baboon	Barbara	Goldhill	Lena	Peppers
Floating point - fixed point	0.03	0.03	0.03	0.03	0.02

Table 2.2: The difference between image smoothing with floating point and integer numbers

inverse transformed back from its DCT representation, using floating points numbers and fixed point numbers. As the data suggest in table 2.1 there are no visual difference between using floating point and integer for the IDCT.

2.4.2 Image Smooting Application

Image smoothing blurs a picture by passing a mask over the picture for each pixel. The mask, which in this case covers 5x5 pixel, redefines the pixel value in the middle as a sum of the neighbouring pixels times the corresponding mask values divided by the total weight of the mask. Figure 2.2 shows the Gaussian mask used in this application. The mask total weight is 159, which needs to be divided. As division is expensive and 159 is a bit far from 128 and 256 which is division factors that can be achieved by a right shift. The weights of the mask is changes to fixed point number, which makes it possible to incorporate the division into the weight of the mask, without using division. This gives the mask in figure 2.3. The fixed point scaling is choose to be $\frac{1}{2^{16}}$, meaning that the LSB of the masks integer values represent a values of $\frac{1}{2^{16}}$. The pixel values is defined to an integer value between 0 and 255. As the mask represent a fixed point number with a scaling $\frac{1}{2^{16}}$, the sum is right shifted 16 positions, given the end result. The difference between image smoothing calculated with floating point and fixed point numbers is given in table 2.2. By visual inspection the difference between floating point and fixed point is unnoticeable for the human eye as seen in figure 2.4.

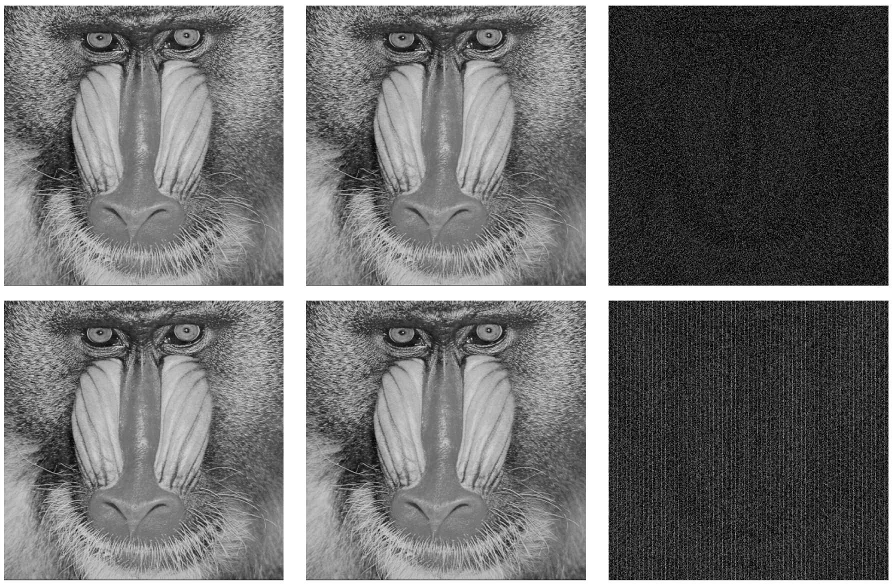


Figure 2.1: Visual result of floating point transformation (top) and integer transformation (bottom). Original picture to the left, DCT-IDCT transformation in the middle. Error map of $|\bar{\epsilon}|^p$ to the right, magnified 60 times.

$\frac{1}{159}\times$

2	4	5	4	2
4	9	12	9	4
5	12	15	12	5
4	9	12	9	4
2	4	5	4	2

Figure 2.2: Gaussian mask used for smoothing filter

$\frac{2}{159}$	$\frac{4}{159}$	$\frac{5}{159}$	$\frac{4}{159}$	$\frac{2}{159}$
$\frac{4}{159}$	$\frac{9}{159}$	$\frac{12}{159}$	$\frac{9}{159}$	$\frac{4}{159}$
$\frac{5}{159}$	$\frac{12}{159}$	$\frac{15}{159}$	$\frac{12}{159}$	$\frac{5}{159}$
$\frac{4}{159}$	$\frac{9}{159}$	$\frac{12}{159}$	$\frac{9}{159}$	$\frac{4}{159}$
$\frac{2}{159}$	$\frac{4}{159}$	$\frac{5}{159}$	$\frac{4}{159}$	$\frac{2}{159}$

Figure 2.3: Gaussian mask used for smoothing filter, for fixed point operation

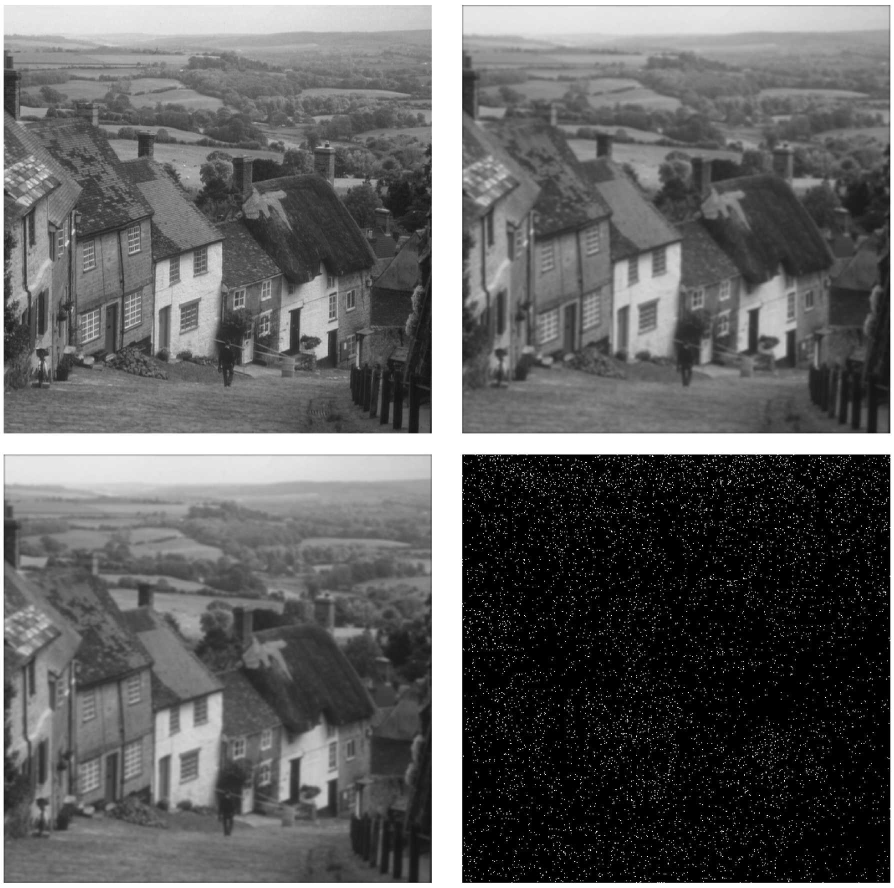


Figure 2.4: Visual result of image smoothing performed by floating point (top right) and integer (bottom left). Original image (top left), Error map of $|\bar{\epsilon}|^p$ between the two (bottom right), magnified 255 times.

$$\mathrm{M}_y = \begin{array}{|c|c|c|}\hline -1 & 0 & 1 \\ \hline -2 & 0 & 2 \\ \hline -1 & 0 & 1 \\ \hline \end{array} \qquad \mathrm{M}_x = \begin{array}{|c|c|c|}\hline -1 & -2 & -1 \\ \hline 0 & 0 & 0 \\ \hline 1 & 2 & 1 \\ \hline \end{array}$$

Figure 2.5: Gaussian mask used for smoothing filter

2.4.3 Edge-detection Application

Edge-detection comprises of four steps, Smoothing, Enhancement, Detection and Localization. The smoothing step suppresses as much noise as possible. The enhancement step applies a filter which enhances the edges in the image. The detection step provides a threshold for what pixels are noise and which are describing an edge. Localization determines the exact location of the edge. In this test application, smoothing is provided by the floating point implementation described in section 2.4.2. Enhancement is provided by the Sobel operation described underneath. Detection and Location is not used in this application.

After applying image smoothing to the image the enhancement step is performed as follows. As with image smoothing, a mask redefines the pixel value in its mid as a sum of the neighbouring pixels times the corresponding mask values. In the case of the sobel edge detection, there are two masks which covers 3x3 pixel, which absolute combined value gives the end pixel result. The Sobel masks is given in figure 2.5, one enhances the vertical edges and one enhances the horizontal lines, together they enhances the diagonal edges.

All values in the schemes are all integers. The Sobel mask weights are {-2,-1,0,1,2} and the pixel values from the image are in the range {0, 1,. . . , 254, 255}. The end result can be bigger than 255, as is the highest values a pixel value can take, this only affects the images and not the calculated error. In figure 2.6 the original picture with applied edge-detection can be seen. As it is a pure integer scheme, an error free integer application can produce a perfect result.

2.5 Implementation and Synthesis

Synopsys Design Vision (V. G-2012.06 May 30, 2012) is used to synthesize the VHDL implementation of the different schemes. No timing constrains is places on the synthesis, only dynamic power and leakage constraints which is set to zero, given the circuit with the smallest power dissipation possible. The simulated circuit power dissipation, area and delay are all obtained through Design

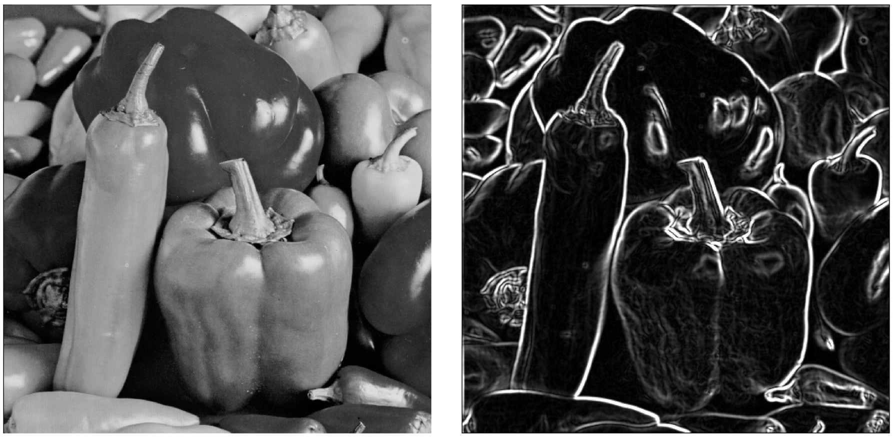


Figure 2.6: Visual result of edge detection for test picturepeppers

Vision.
Synopsys Chronologic VCS (V. D-2010.06_ Full64) is used to simulate the synthesized gate level design, obtaining the switching activity based on the actual gate delay. All simulation is done at 100MHz. The test vector used is comprise of a random set together with the test vectored recorded performing IDCT on the test pictures.
A 90 nm library of standard cells with the nominal supply voltage $V_{dd} = 1V0$ is used.

2.6 Test Pictures

The test pictures used are Baboon, Barbara, Goldhill, Lena and Peppers in 2.7. They can be found at `http://en.pudn.com/downloads187/sourcecode/graph/detail878292_en.html`.. They are all gray scale image, with the pixel values between 0 and 255, with 0 being black and 255 white.

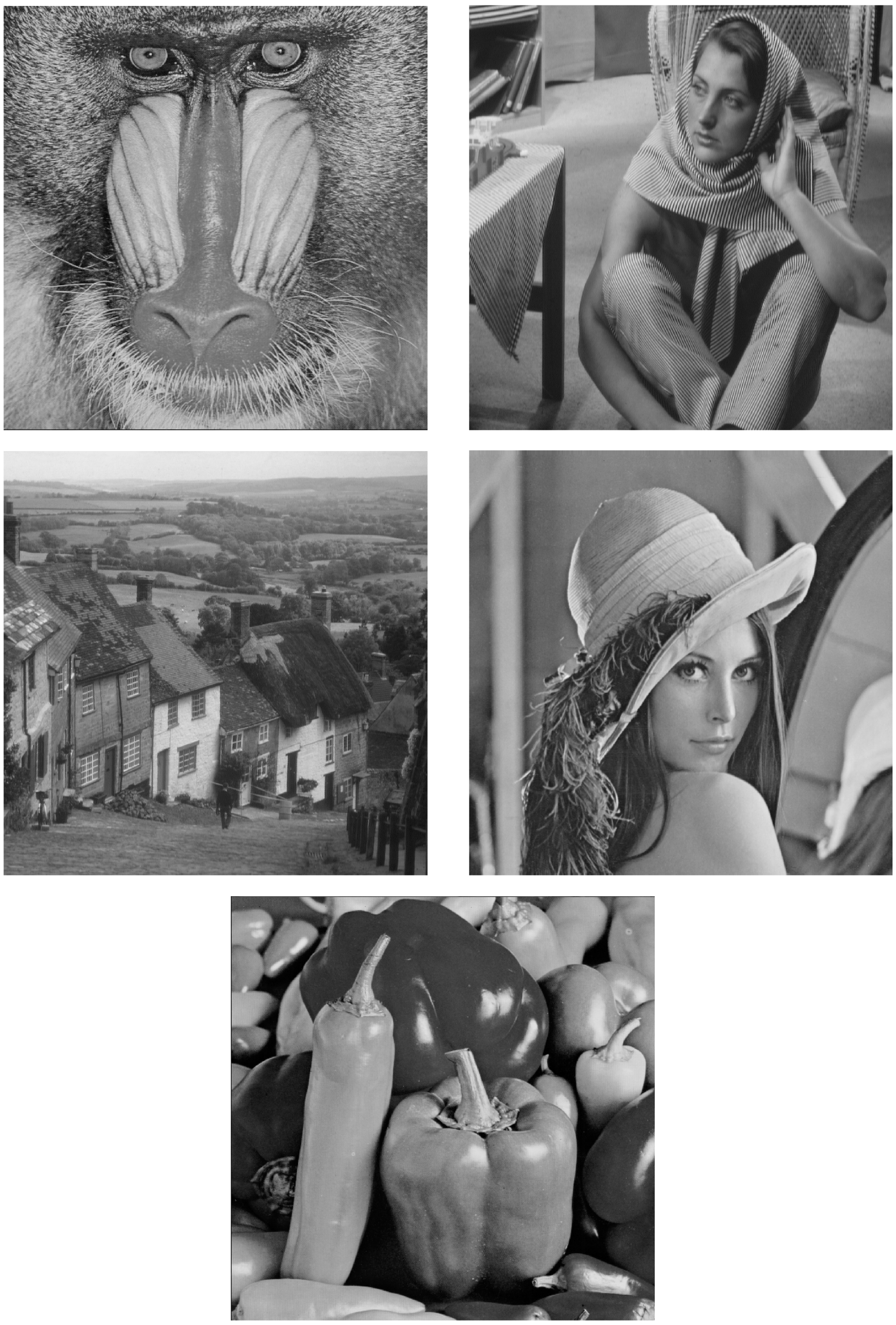


Figure 2.7: Test pictures used for image processing. From to left, Baboon, Barbara, Goldhill, Lena and Peppers

Chapter 3

Addition

The addition schemed learned in the second school year, applies to all non redundant number system, not only the decimal system, Radix-10. Addition in different radix systems all works in the same way and can be generalized as described in algorithm 1. Add the least significant digit of the two numbers together plus the carry, if they exceed the radix for the given system, create a carry for next least significant digit. Repeat until there are no more digits and carry bits are generated.

Algorithm 1 General addition scheme

```
$C \leftarrow 0, \qquad I \leftarrow 0$
**for** $C > 0$ & $Radix^i < max(A, B)$ **do**
    $T \leftarrow A_i + B_i + C$
    **if** $T > Radix$ **then**
        $C \leftarrow 1$
    **else**
        $C \leftarrow 0$
    **end if**
    $S_i \leftarrow T \ modulus \ Radix$
    $i \leftarrow i + 1$
**end for**
```

As an example of addition in different Radix's see figure 3.1, where addition is performed on decimal, octal and binary numbers.

As can been seen by algorithm 1, the carry is propagated through the whole addition and is basically the deciding factor when choosing adder implementing, given a timingconstraint.

3.1 Imprecise Addition Schemes

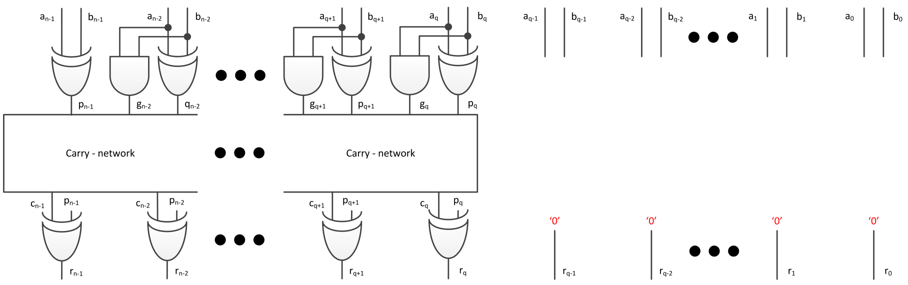
There are lots of ways to design an imprecise adder. You can make one which always gives the result 1, independent of the input. This of course would not been seen as an adder circuit by many people, but in theory it is an imprecise adder circuit, just one which uses very little power and in most cases gives a very big error. In this chapter commonly known imprecise adders and new proposed adders are presented. The most significant part of the adder is calculated error free, with the least significant part estimated by an imprecise adder circuit. The width of least significant part is denoted q and is presented as ADDER$_q$.

3.1.1 Input Truncation Scheme (Trunc)

This scheme is one of the most basic imprecise addition schemes and is widely used where the least significant part of an addition has no or little interest. The scheme do not try to retain any information of the least significant part of the two input arguments and set their values to zeros. Figure 3.2 describes the implementation. As no carry is generated by the least significant part and the output is set to zero, all logic is removed from this. The reduction of the carry-network, restricting it to the most significant part, reduces the adders delay, as the carry network always are the critical path. Figure 3.3 shows the difference between an exact adder and an input truncated adder with an imprecise width = 4, Trunc$_4$.

	Radix-10	Radix-8	Radix-2
A :	3 2 7	3 2 7	1 1 0
B :	7 5 4 +	7 5 4 +	0 1 1 +
C :	1 0 1 - +	1 1 1 - +	1 1 0 - +
S :	1 0 8 1 =	1 3 0 3 =	1 0 0 1 =

Figure 3.1: Addition of decimal, octal and binary numbers.



a_{n-1} b_{n-1} a_{n-2} b_{n-2} a_{q+1} b_{q+1} a_q b_q a_{q-1} b_{q-1} a_{q-2} b_{q-2} a_1 b_1 a_0 b_0
p_{n-1} g_{n-2} q_{n-2} g_{q+1} p_{q+1} g_q p_q
Carry - network
Carry - network
c_{n-1} p_{n-1} c_{n-2} p_{n-2} c_{q+1} p_{q+1} c_q p_q
'0' '0' '0' '0'
r_{n-1} r_{n-2} r_{q+1} r_q r_{q-1} r_{q-2} r_1 r_0

Figure 3.2: Input trunk scheme

Exact

A	:	0	0	1	1	0	1	0	1	
B	:	1	0	1	1	1	0	1	0	+
C	:	0	0	1	0	0	0	0	-	+
S	:	1	1	1	0	1	1	1	1	=

Trunck$_4$

A	:	0	0	1	1	0	1	0	1	
B	:	1	0	1	1	1	0	1	0	+
C	:	0	0	1	-	-	-	-	-	+
S	:	1	1	1	0	0	0	0	0	=

Figure 3.3: Different between Exact and Trunk addition. n=8, q=4

3.1.2 Freeze0.5 Scheme

The Freeze 0.5 is proposed in [MP10]. And looks identical to the input trunk scheme except that it changes the most significant bit of the truncated part to a logical 1. In [MP10] this is done by freezing the least significant part of an error free adder by disabling the shifting capability of the adders input register, taking advantage of the low leakage power. The reducing in power consumption is only from a reduction in switching activity, but makes it possible to place any number in the frozen least significant part of the adder. Theoretical, freezing the input also reduced the critical path. The implementation of a hardware implemented Freeze0.5 scheme is given in figure 3.4. Figure 3.5 shows the difference between an exact adder and the Freeze0.5$_4$.

3.1.3 OR- and XOR-tail Scheme

Both OR and XOR tail schemes are proposed in [AN11]. The addition schemes replaces the least significant part with a parallel OR or XOR network, their implementation shown in 3.6. The schemes reduced the complexity of a precise addition by removing the least significant part of the carry-network, thereby reducing the critical path. They different from the Trunc and Freeze0.5 scheme by retaining some of the information in the least significant part of the two

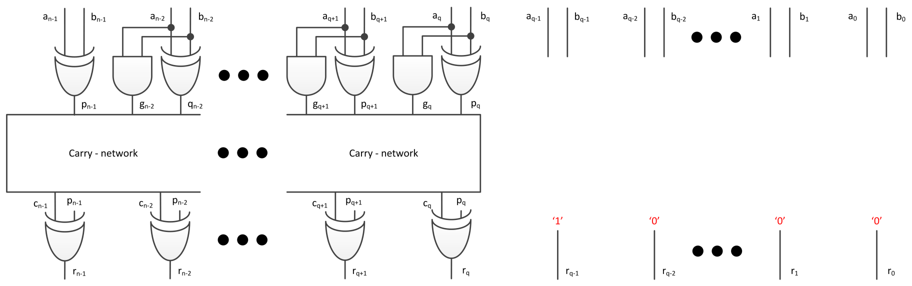


Figure 3.4: Freeze 0.5 scheme

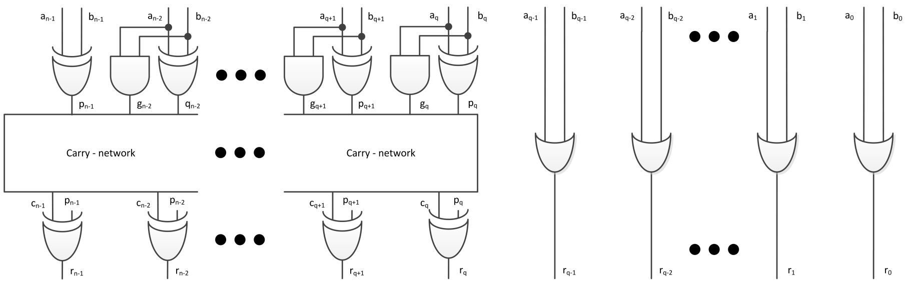
```
              Exact                                      Freeze0.5₄
A :   0 0 1 1   0 1 0 1             A :   0 0 1 1   0 1 0 1
B :   1 0 1 1   1 0 1 0  +          B :   1 0 1 1   1 0 1 0  +
C : _ 0 0 1 0 _ 0 0 0 -  +          C : _ 0 0 1 - _ - - - -  +
S :   1 1 1 0   1 1 1 1  =          S :   1 1 1 0   1 0 0 0  =
```

Figure 3.5: Different between Exact and Freeze0.5 addition. n=8, q =4

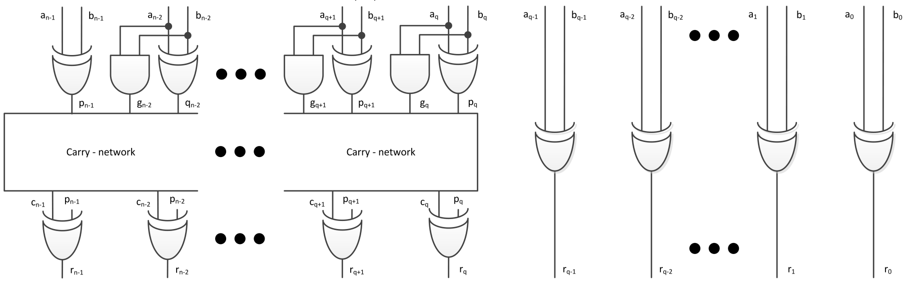
input arguments. The reduction of area reduces the static power consumption together with the switching activity of the missing circuit, compared to that of an error free adder. As the imprecise part do not have carry network, the critical path is restricted to the precise part. Figure 3.7 shows the difference between an exact adder and the OR-tail$_4$ and XOR-tail$_4$.

3.1.4 Carry-one Scheme

Different form the other schemes presented the Carry-one scheme only cripples the lest significant part of the carry-network instead of removing it completely. The carry-network is crippled in the least significant part, so it only generate a carry to the following bit. Basically replacing the carry-network with parallel half adders, where the sum$_i$ and carry$_{i-1}$ are OR'ed together. One advantage of this scheme is that it is possible to preserve more information than the OR- and XOR-tail schemes, but getting the same delay reduction. The implementation is illustrated in figure 3.8. It is the only scheme where the imprecise part of the adder can influence the error free part, with a pseudo carry. The area is only reduced slightly, but the critical path is reduced the same as the other schemes. Q is a some what miss leading denotation here, as some of the data from the imprecise part can move over to the error free part, as the last carry bit of the imprecise part is OR'ed with the LSB of the error free parts result. The logic


a_{n-1} b_{n-1} a_{n-2} b_{n-2} a_{q+1} b_{q+1} a_q b_q a_{q-1} b_{q-1} a_{q-2} b_{q-2} a_1 b_1 a_0 b_0
p_{n-1} g_{n-2} q_{n-2} g_{q+1} p_{q+1} g_q p_q
Carry - network
Carry - network
c_{n-1} p_{n-1} c_{n-2} p_{n-2} c_{q+1} p_{q+1} c_q p_q
r_{n-1} r_{n-2} r_{q+1} r_q r_{q-1} r_{q-2} r_1 r_0

(a) OR-tail scheme


a_{n-1} b_{n-1} a_{n-2} b_{n-2} a_{q+1} b_{q+1} a_q b_q a_{q-1} b_{q-1} a_{q-2} b_{q-2} a_1 b_1 a_0 b_0
p_{n-1} g_{n-2} q_{n-2} g_{q+1} p_{q+1} g_q p_q
Carry - network
Carry - network
c_{n-1} p_{n-1} c_{n-2} p_{n-2} c_{q+1} p_{q+1} c_q p_q
r_{n-1} r_{n-2} r_{q+1} r_q r_{q-1} r_{q-2} r_1 r_0

(b) XOR tail scheme

Figure 3.6: OR- and XOR-tail implementation, both retaining some information through the imprecise part

```
            Exact                             OR4                               XOR4
A :   0 0 1 1   0 1 1 1          A :   0 0 1 1   0 1 1 1          A :   0 0 1 1   0 1 1 1
B :   1 0 1 1   1 0 1 0 +        B :   1 0 1 1   1 0 1 0 +        B :   1 0 1 1   1 0 1 0 +
C : _ 0 0 1 0 _ 0 0 0 - +        C : _ 0 0 1 - _ - - - - +        C : _ 0 0 1 - _ - - - - +
S :   1 1 1 0   1 1 1 1 =        S :   1 1 1 0   1 1 1 1 =        S :   1 1 1 0   1 1 0 1 =
```

Figure 3.7: Different between Exact and OR/XOR-tail addition. n=8, q =4

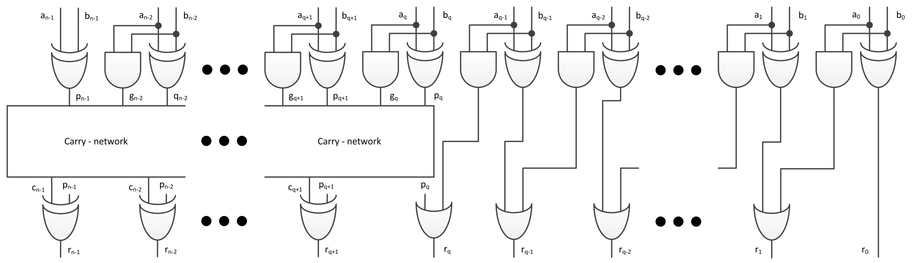


Figure 3.8: Carry-one scheme

		Exact								
A	:	0	0	1	1	0	1	1	1	
B	:	1	0	1	1	1	0	1	0	+
C	:	0	0	1	0	0	0	0	-	+
S	:	1	1	1	0	1	1	1	1	=

		Carry-one$_4$								
A	:	0	0	1	1	0	1	1	1	
B	:	1	0	1	1	1	0	1	0	+
C	:	0	0	1	0	0	1	0	0	+
S	:	1	1	1	0	1	1	0	1	=

Figure 3.9: Different between Exact and Carry-on addition. n=8, q = 4

in the imprecise part of Carry-one and the soft separation of the imprecise and error free part makes a full carry chain from 2^0 to 2^n possible, over multiple additions. Figure 3.9 shows the difference between an exact adder and the Carry-one$_4$.

3.2 Errors Generated by Imprecise Adders

The four performance parameters for an imprecise adder are in alphabetic order Area, Delay, Error and Power, the smaller the better. In this chapter the error is investigated.

3.2.1 Statistical Error

Table 3.1 describes the error functions for the imprecise addition schemes presented in chapter 3.1. The error functions describes how the error change depending on the width, q, of the imprecise part. The average error - $\bar{\epsilon}$, average absolute error -$|\bar{\epsilon}|$, minimum error -ϵ_{min}-, maximum error -ϵ_{max} and maximum absolute error -$|\epsilon|_{max}$ is defined in section 2.2. All errors growth exponential with respect to q.

Type	$\bar{\epsilon}$	$\lvert\bar{\epsilon}\rvert$	ϵ_{min}	ϵ_{max}	$\lvert\epsilon\rvert_{max}$
Trunc$_q$	-2^q+1	2^q-1	$2-2^{q+1}$	0	$2^{q+1}-2$
Freeze 0.5$_q$	$1-2^{q-1}$	$< 2.025^{i-1}$	$-\left(3-\frac{4}{2^q}\right)2^{q-1}$	2^{q-1}	$\left(3-\frac{4}{2^q}\right)2^{q-1}$
OR-tail$_q$	$\frac{1}{4}-2^{q-2}$	$2^{q-2}-\frac{1}{4}$	$1-2^q$	0	2^q-1
XOR-tail$_q$	$\frac{1}{2}-2^{q-1}$	$2^{q-1}-\frac{1}{2}$	$2-2^{q+1}$	0	$2^{q+1}-2$
Carry-one$_q$	$\frac{1}{4}-2^{q-2}$	$2^{q-2}-\frac{1}{4}$	$-2^q \times \sum_{i=1}^{\frac{q+1}{2}} \frac{2}{2^{2i}-1}$	0	$2^q \times \sum_{i=1}^{\frac{q+1}{2}} \frac{2}{2^{2i}-1}$

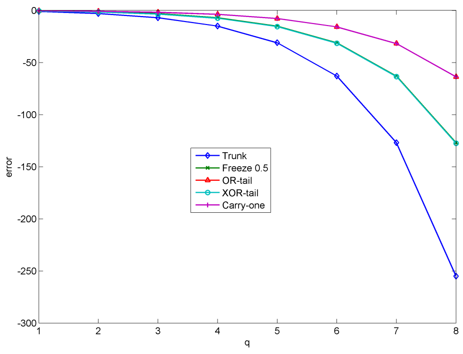
Table 3.1: Statistical derived error functions for different imprecise addition schemes, found by exhausting simulation. Blue indicated the lowest error function. Red indicated $|\bar{\epsilon}|$ for Freeze0.5 given as an upper limit.

Freeze 0.5 stands out as it is the only imprecise adder which can generate a positive error, all other schemes has a maximum error of 0. All schemes have a negative $\bar{\epsilon}$, even Freeze 0.5. This means that for all schemes $\bar{\epsilon} = -|\bar{\epsilon}|$ except for Freeze 0.5, for which $\bar{\epsilon} \approx -|\bar{\epsilon}|$ holds. OR-tail is the best performing scheme, for $\bar{\epsilon}$, $|\bar{\epsilon}|$ and ϵ_{max} it performs equivalent to Carry-one, but outperforms it for ϵ_{min} and $|\epsilon|_{max}$. $|\bar{\epsilon}|$ for Freeze0.5 is given as an upper limit, as the exact function could not be found, it is indicated red in table 3.1. A graphical representation of the error function can be seen in figure 3.10.

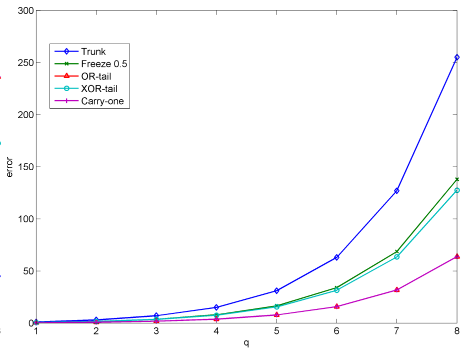
3.2.2 Transformation Error

The error represent the difference between IDCT calculated with an error free adder and an imprecise adder scheme. The error are averaged over the test pictures. $|\bar{\epsilon}|^p$ and $|\epsilon|^p_{max}$ is defined in section 2.3.

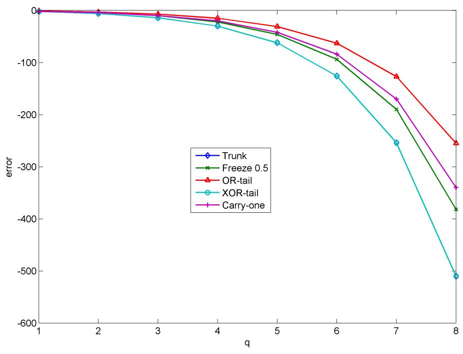
Figure 3.11a shows the average absolute error, $|\bar{\epsilon}|^p$, of each pixel. Trunc, Freeze0.5 and OR/XOR-tail produces equivalent errors for an imprecise width under 16,$q \leq 16$, the reason for this is the hard separation of the error free and imprecise part of the adder schemes, which prohibits data smaller than 2^q to carry upwards. As the IDCT transformation scheme uses a fix point scaling of $\frac{1}{2^{16}}$, which right shift 16 positions for obtaining the final integer pixel value, data with a value under 2^{16} are discarded. For $q > 16$ the imprecise part of the addition scheme starts to be a represented in the final pixel value. The error generated by Trunc, Freeze0.5 and OR/XOR-tail starts to distinguish from each other, with OR-tail as the best performing of the four and Trunc as the worst. Carry-one has a soft separation of its precise and its imprecise part, which allows data generated in the imprecise part to be carried upward into the precise part. This property makes Carry-one the best performing of the imprecise addition



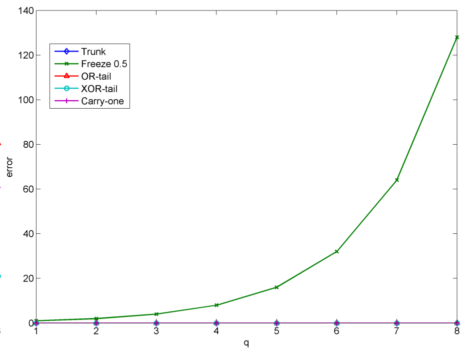
(a) $\bar{\epsilon}$



(b) $|\bar{\epsilon}|$

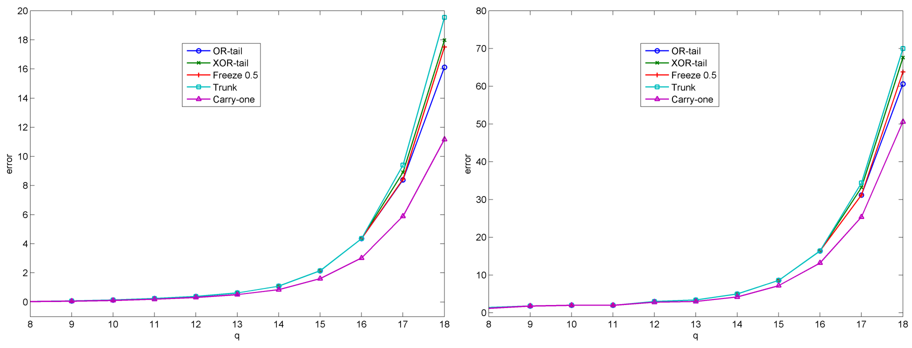


(c) ϵ_{min}



(d) ϵ_{max}

Figure 3.10: Graphical representation of the error functions presented in 3.1



(a) $|\bar{\epsilon}|^p$ **(b)** $|\epsilon|^p_{MAX}$

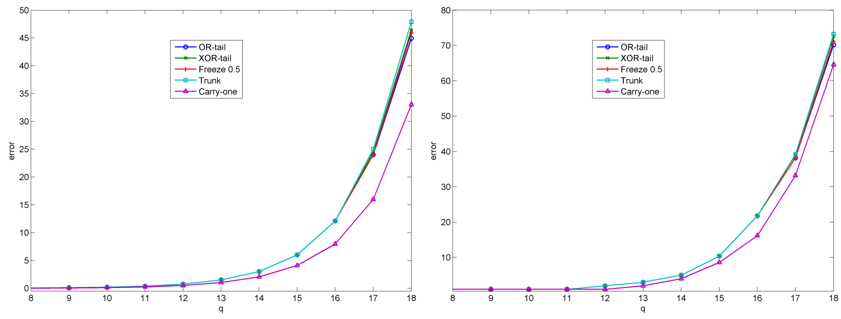
Figure 3.11: Errors generated using imprecise addition compared to error free addition performing IDCT

schemes.

Figure 3.11b shows the maximum absolute error, $|\epsilon|^p_{max}$. Despite the fact that $|\bar{\epsilon}|^p$ indicate that an imprecise addition scheme could be used without a performance impact for $q \leq 10$, $|\epsilon|^p_{max}$ tells a different story as $|\epsilon|^p_{max} = 2$ for $q \leq 10$. Again the error generated by the Carry-one is equal or better than the other proposed schemes.

3.2.3 Image Smoothing Error

The error represents the difference between two pictures after they have been smoothed, one using error free and one using imprecise addition. $|\bar{\epsilon}|^p$ and $|\epsilon|^p_{max}$ is defined in section 2.3. The error are averaged over the test pictures. The image smoothing application is described in section 2.4.2.Figure 3.12a shows the average absolute error, $|\bar{\epsilon}|^p$. As with transformation, image smoothing uses a fix point scalin of $\frac{1}{2^{16}}$, which right shift 16 positions for obtaining the final integer pixel value, data with a value under 2^{16} are discarded. Given the same error for Trunc$_{<16}$, Freeze0.5$_{<16}$, OR-tail$_{<16}$ and XOR-tail$_{<16}$. $|\bar{\epsilon}|^p \approx -\bar{\epsilon}^p$ which, indicates that majority of errors are negative and the amount of positive errors produced are to small to have any influence. The error graphs is similar to those found performing IDCT, but scale differently. The error scaling is most likely produced by the difference in schemes, as smoothing sums 25 entries compared to 8 for IDCT. As the amount of entries in the sum function grows the lack of a full carry chain get more apparent. Figure 3.12b shows the maximum absolute error $|\epsilon|^p_{max}$. Again the error shape seems similar to that of the IDCT, even the



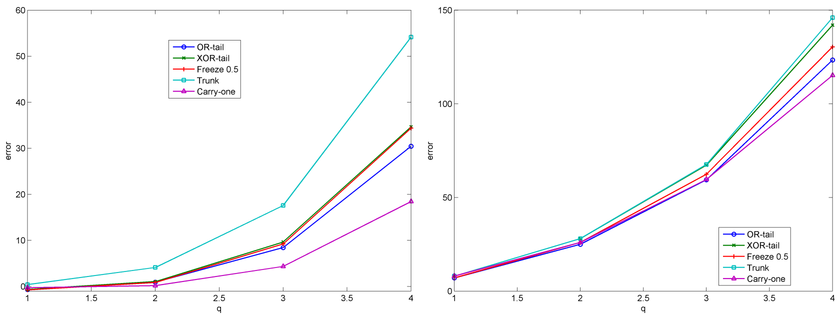
(a) $|\bar{\epsilon}|^p$ **(b)** $|\epsilon|^p_{max}$

Figure 3.12: Errors generated using imprecise addition compared to error free addition performing image smoothing

scaling, but are much more closely packed together. Carry-one performs as good or better than all others schemes for all q's, Trunc, Freeze0.5 and OR/XOR-tail still performance equivalent for $q \leq 16$, and ranks OR-tail, Freeze0.5, XOR-tail and Trunc for $q > 16$ for both $|\bar{\epsilon}|^p$ and $|\epsilon|^p_{max}$.

3.2.4 Edge-detecting Error

Edge detecting is performed without fixed point arithmetic, which distinguishes it from the transformation and image smoothing. $|\bar{\epsilon}|^p$ and $|\epsilon|^p_{max}$ is defined in section 2.3. The error are averaged over the test pictures. See section 2.4.3 for detailed on edge detection algorithm. Figure 3.13a shows a completely different average error graph than that of the transformation and image smoothing application. This can be reasoned with the two different types of number format used, integer operations vs fixed point numbers. All errors seems to be positive, which contradicts the derived error functions in table 3.1. The positive errors are contributed the edge detection algorithm in conspiracy with the imprecise adder schemes. Carry-one performs the best, then OR-tail, XOR-tail and Freeze 0.5 has similar performance and Trunc which generates the biggest errors. In figure 3.13b, the $|\epsilon|^p_{max}$ is shown, which changes ranking to Carry-one, OR-tail, Freeze 0.5 where XOR-tail and Trunc display similar performance.



(a) $\bar{\epsilon}^p \approx |\bar{\epsilon}|^p$ (b) $|\epsilon|^p_{max}$

Figure 3.13: Errors generated using imprecise addition compared to error free addition performing edge detection

3.2.5 Error Discussion

IDCT and image smoothing both uses fixed point arithmetic with the same scale factor. Their error curves for $|\bar{\epsilon}|^p$ looks similar but with a different scaling. The biggest error of the two occurs in image smoothing and is likely generated by the larger amount of entries to be summed together. The $|\epsilon|^p_{max}$ curves are similar both in shape and scaling. Using fixed point number representation some of the errors is removed when truncating, which is clearly seen in both IDCT and smoothing for imprecise adders with an imprecise width under 16, where errors generate by Trunc, Freeze0.5 and OR/XOR-tail are the same, as information from the imprecise part of the schemes never contributed to the end value. When the imprecise part of the schemes is directly a part of the final result, for $q > 16$, they ranked as table 3.1 suggested, with the exception of Carry-one. The edge detection uses integer operations and is more sensitive for error in the least significant part of the logic, this can clearly be seen in figure 3.13, where $|\bar{\epsilon}|^p > 15$, for all schemes with a $q \geq 4$, where transformation and image smoothing ha $|\bar{\epsilon}|^p < 1$. Transformation, smoothing and edge detection do not give the same impression as the statistically derived error functions. The reason for this is that the error in table 3.1 is based on a single addition and not a summarising, which is heavily used in transformation, smoothing and edge detection. OR/XOR-tail, Trunc and Freeze 0.5 all have a distinct error free part and an imprecise part with no overlap. Carry-one has a precise part which is overlapped by the imprecise, lest significant part. The overlap can deliver a single carry from the imprecise part into the error free part, basically adding a delayed carry, which unfortunately can be masked by the precise result, giving a slight performance advantage summarizing over many numbers.

	Trunk_{14}	Freeze 0.5_{14}	OR-tail_{14}	XOR-tail_{14}	Carry-One_{15}
IDCT $\lvert\bar{\epsilon}\rvert^q$	1.09	1.09	1.09	1.09	1.61
IDCT $\lvert\epsilon\rvert^q_{max}$	5.0	5.0	5.0	5.0	7.2

Table 3.2: Chosen q's for addition schemes together with errors average over the test images

It is clear from the different errors generated by the IDCT and image smoothing versus that of edge-detection, that not all application can be executed on the same hardware, with the expectation of the same error generation.

The imprecise adders big weakness is its lack of carry-chain, which blocks carry through from the imprecise part, Carry-one solves to some degree this problem and is shown to be superior when summering over many entries. The performance of the OR-tail is superior forsingleadditions.

3.3 Imprecise Adder Implementation

The four performance parameters for an imprecise adder are in alphabetic order Area, Delay, Error and Power, the smaller the better. In in the previous section the error was investigated, in this section area, delay and power dissipation is investigated.

To compare the different schemes implementation when generating a similar error, each scheme were tuned to maximize the width of the imprecise part, produced $|\bar{\epsilon}|^p \leq 2$ when calculating IDCT. The imprecise additions schemes were applied to a 32bit adder, implemented as 32 bit two-level carry-lookahead adder, as described in [MDETL04, p 75]. The synthesise tools and conditions is described in section 2.5.

Table 3.2 summarise the width of the imprecise adders and the error generated by calculating IDCT, collected from section 3.2.2.

3.3.1 Area and Delay Comparison

The area and delay are specific for the implementation and do not change with changes in the applied data, the area data is located in table 3.3. The XOR-tail is the largest of the simple imprecise adder schemes, using 65% of the error free implementation, the OR-tail is a close second and it is apparent how big a

	Area $[\mu m^2]$					
	Error-Free	OR-Tail$_{14}$	XOR-tail$_{14}$	Trunk$_{14}$	Freeze 0.5$_{14}$	Carry-One$_{15}$
Area	1471	873	949	796	796	1035
Ratio$_{Errorfree}$	1.00	.59	.65	.54	.54	0.70

Table 3.3: Area of imprecise addition schemes, compared to an error free implementation.

	Delay $[ns]$					
	Error-Free	OR-Tail$_{14}$	XOR-tail$_{14}$	Input Trunk$_{14}$	Freeze 0.5$_{14}$	Carry-One$_{15}$
Dalay	2.04	1.59	1.59	1.59	1.59	1.59
Ratio$_{Errorfree}$	1.00	.78	.78	.78	.78	.78

Table 3.4: Delay of imprecise addition schemes, compared to an error free implementation.

XOR gates is compared to a OR gate. Both Trunk and Freeze0.5 uses 54% of the error free implementation which is the minimum size for an imprecise adder with an imprecise width of 14, as the imprecise parts output gates is bounded to either logic 0 or 1. The Carry-one uses the most area, 70%, even though it has the widest imprecise part. This is due to the complexity of its imprecise part, which uses multiple gates for generating the output.
The delay data is located in table 3.4. The error free adder has a delay of 2.04 nS. OR-tail$_{14}$, XOR-tail$_{14}$, Trunk$_{14}$, Freeze0.5$_{14}$ and Carry-one$_{15}$ all have the same delay of 1.59 nS, 22% faster than the error free adder. This is because the critical path in the adder is the carry-network, which for all is A[17] $\rightarrow$ Result[31].

3.3.2 Power Comparison

The test vectored is the input values performing IDCT on different pictures, together with a random set of test vectored. Table 3.5 shows the simulated power consumption. Trunk$_{14}$ and Freeze0.5$_{14}$ consumes the least power, only

	Power $[\mu W]$						
	Error free	$\text{EF}_{Trunk_{14}}$	OR-tail_{14}	XOR-tail_{14}	Trunk_{14}	Freeze0.5_{14}	Carry-one_{15}
Random	133	76	73	78	71	71	80
Barboon	118	65	60	67	60	60	68
Barbara	108	60	56	62	55	55	63
Goldhill	112	62	58	63	57	57	65
Lena	107	60	56	61	55	55	63
Peppers	110	61	57	63	56	56	64
Average	115	64	60	66	59	59	67
Ratio_{EF}	1.00	0.56	0.52	0.57	0.51	0.51	0.58

Table 3.5: Transformation - Power consumption of different pictures on different addition schemes $[\mu W]$

using 51% of the error free implementation, this was expected as no logic is present in its imprecise part. OR-tail_{14} uses 52% of the EF implementation. For the imprecise part of the OR-tail holds that the switching activity is kept to a minimum, as the output is feed back as input. The OR gates will generate a high output when presented with a high input and will at most switch ones, giving a minimum of switching activity and power consumption. XOR-tail is not bounded by the same input output feed back as the OR-tail, as the output is dependent on both input. This gives a higher switching activity in more complex gates and uses 57% of the EF implementation. Carry-one has the highest power consumption of all the imprecise schemes using 58% of the EF implementation, this can be contributed to the more complex gate array in the imprecise part and a higher switching activity. The $\text{EF}_{Trunk_{14}}$ describes the power consumption of the error free implementation with the least significant part of the input frozen, basically giving the same scheme as Trunk_{14} but on an error free implementation. $\text{EF}_{Trunk_{14}}$ only uses 56% of the EF implementation and it is evident that the static power consumption only contributes very little to the total power numbers.

3.3.3 Implementation Discussion

For all implementations goes that a the imprecise implementation uses less area, creates a shorter critical path and have a smaller power consumption than the error free adder. If a bigger error could be tolerated, the width of the imprecise part could be widend, giving an even bigger savings in area, delay and power consumption. All schemes shows equivalent delay characteristic only leaving two performance criteria: area and power. Area and power wise Trunk and Freeze0.5 wins as they uses the lowest area and power. The frozen EF implementation $EF_{trunk_{14}}$ is a potential competitor, as it besides having a low power consumption, also can act as precise adder.

3.4 Conclusion for Imprecise Addition

None of the presented imprecise addition schemes manage to have a balanced $\bar{\epsilon}$. OR-tail is the most statistical accurate adder, with Carry-one as a close second only differentiating them self from each other by $|\epsilon|_{max}$. For image smoothing and transformation the errors introduced by the imprecise adder schemes is the same up to an imprecise width of 16, except for Carry-one. This it because both schemes uses a right shift of 16 bit at the end, making the precise part of the adder scheme the only contributing part to the end result. With an imprecise width of over 16, the imprecise part contributes directly to the end result, OR-tail with the best outcome, but at this width the error is considerable. For edge-detecting which is an integer algorithm, the difference between the imprecise adders are substantial, as the imprecise part contributes to the end result. Again Carry-one is the best performing imprecise adder. OR-tail is marginally better than XOR-tail and Freeze0.5 and Trunc is the worst performing, producing an error three times that of Carry-one. The imprecise addition scheme Carry-one, outperforms the other schemes in all three applications by being able to transfer a carry from the imprecise to the precise part, thereby saving more information than any other schemes. To compare the different additions schemes area, delay and power consumption, the biggest imprecise width of each scheme with a performance of $|\bar{\epsilon}|^p \leq 2$ for transformation were implemented and synthesised. Carry-one had an imprecise with of 15, while the others had a with of 14. The imprecise additions schemes were applied to a 32bit two stage CLA. All schemes came out with a 22% lower delay, making it possible to save $\approx$ 22% power by a $\approx$ 10% decrease in operating voltage, still keeping the same timing constrains. As the width of the imprecise part of the adders schemes are almost the same, the area of them is directly comparable to the complexity of the imprecise part. As Trunc and Freeze0.5 do not have any logic in their imprecise part, they have the smallest area and saves 46%, Carry-one has the biggest area as it has the most complex imprecise part only saving 30%. Trunc$_{14}$ and Freeze0.5$_{14}$ have a power reduction of 49%, closed followed by OR-tail, saving 48%. The reason for the low power consumption of the OR-tail is its self reinforcing production of '1' when the output is reapplied to its input, making the output only switch once. Carry-one$_{15}$ had a high power consumption and only saved 42%, closed followed by XOR-tail$_{14}$ with 43%. The error free addition scheme were applied the same test vectors as the Trunc_{14}, the last 14 least significant bits zeroed, there by performing as Trunc_{14}. It had a power reduction of 44% by not using its full precision, using less power than XOR-tail_{14} and Carry-one_{15}.

Given the highly different error sizes for different application with the same imprecise addition scheme, it is fair to say that one-size does not fit all. Carry-one had the over all best performance, but also the highest power consumption of the imprecise adders, still manages to save 42% power, 55% with applied voltage scaling. But the title of: king of the hill, goes to the error free implementation, as by freezing its input, a 44% reduction in power consumption can be achieved and altering the amount of frozen bits, would make it perform with many differentapplications.

CHAPTER 4

Multiplication

Multiplication can be expressed in its the general form as equation 4.1. Where y is the multiplier, z the multiplicand and the result is the product between the two.

$$result = y \times z \tag{4.1}$$

With small or pleasing numbers this can be done by mental arithmetic. But as the numbers gets funnier, mental arithmetic becomes hard and papers are normally taken to aid, if not machines. Even though most people jumps a couple of steps, the main way to multiply numbers is the one taught in 5'th school year, which can be described as the sum function 4.2.

$$result = \sum_{j=0}^{|y|<Radix^j} O(y_j) \times Z \times Radix^j \tag{4.2}$$

Even though it was taught in the decimal system, Radix-10, the structure in which to multiply two numbers together holds for most number systems.

An example of equation 4.2, two decimal numbers $325_{10} \times 954_{10}$, 325_{10} being the multiplier and 954_{10} the multiplicand is being multiplied. As $y < 10^3, j = \{0, 1, 2\}$ the sum function is executed over 3 iterations. The multiplication is described in equation 4.3, where the sum function is unrolled.

$$\begin{aligned}
325_{10} \times 954_{10} = \quad & O(5_{10}) \times 954_{10} \times 10^0 : & 4770_{10} \qquad (4.3a)\\
& +O(2_{10}) \times 954_{10} \times 10^1 : & 19080_{10} \qquad (4.3b)\\
& +O(3_{10}) \times 954_{10} \times 10^2 : & 286200_{10} \qquad (4.3c)\\
& & = 310050_{10} \qquad (4.3d)
\end{aligned}$$

A further example of this method works with other than the decimal system, the binary, Radix-2, numbers $01101_2(13_{10}) \times 01011_2(11_{10})$, where 01101_2 is the multiplier and 01011_2 is the multiplicand is being multiplied. As $y < 2^4, j = \{0, 1, 2, 3\}$ the sum function is executed over 4 iterations. The multiplication is described in EQ 4.4, where the sum function is unrolled.

$$\begin{aligned}
01101_2 \times 01011_2 = \quad & O(1_2) \times 01011_2 \times 2^0 : & 01011_2 \qquad (4.4a)\\
& +O(0_2) \times 01011_2 \times 2^1 : & 000000_2 \qquad (4.4b)\\
& +O(1_2) \times 01011_2 \times 2^2 : & 0101100_2 \qquad (4.4c)\\
& +O(1_2) \times 01011_2 \times 2^3 : & 01011000_2 \qquad (4.4d)\\
& & = 10001111_2 \qquad (4.4e)
\end{aligned}$$

$01101_2(13_{10}) \times 01011_2(11_{10}) = 010001111_2(143_{10})$ is the correct result. Normally when operating with logic, the number of iterations of the sum function is fixed to the with of the multiplier word. Even though the sum function, EQ 4.2, is a serial function, it can be unrolled and executed in parallel. The parallel execution style is used in this report. The radix chosen for this thesis is Radix-4, which requires a recoder, see appendix 4.1 for more information.

4.1 Precise Radix-4 Multiplier

The general parallel multiplier scheme consist of 4 main components: Recoder, Partial Product Generator (PPG), Partial Product Reducer (PPR) and a Carry

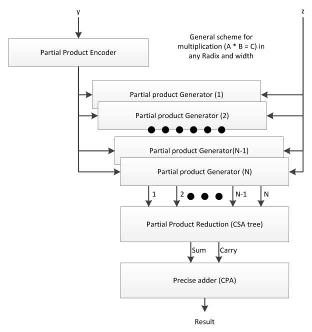

y
z
Partial Product Encoder
General scheme for multiplication (A * B = C) in any Radix and width
Partial product Generator (1)
Partial product Generator (2)
Partial product Generator(N-1)
Partial product Generator (N)
1
2
N-1
N
Partial Product Reduction (CSA tree)
Sum
Carry
Precise adder (CPA)
Result

Figure 4.1: General multiplier scheme

Propagate Adder (CPA). Equation 4.2 is used as a reference to described the different components.

- The Recoder transforms the binary numbers y into a Radix-4 number set, $O(y_j)$

- The Partial Product Generator, multiplies the multiplicand width a pre-defined Radix-4 digit set creating the product $O(y_j) \times z$

- The Partial Product Recucer and CPA is equivilent to the summering function, $\sum$, giving the final result

The missing $Radix^j$ is a simple right shift when representing in binary numbers and can be thought of as keeping alignment, as it will not change the generation of the partial product. The Recoder, PPG, PPR an CPA, is connected as seen in FIG 4.1. Each component has a specific job in the multiplier and can be created with different function as well as gate combination. Chancing one component, being function or gate wise, can alter the accuracy, area and power consumption of the entire multiplier, therefore each component and sub components has been kept as standard as possible, as not to favour or optimize a scheme over another. The change of component is demonstrated in the following sections, where two different recoder schemes is being introduced where everything else is identical. In the following sections the main component of a multiplier will be explained i more detail, together with the implementation.

y_{2j+1}	y_{2j}	y_{2j-1}	$O(y_j)$
0	0	0	0
0	0	1	z_j
0	1	0	z_j
0	1	1	$2z_j$
1	0	0	$-2z_j$
1	0	1	$-z_j$
1	1	0	$-z_j$
1	1	1	-0

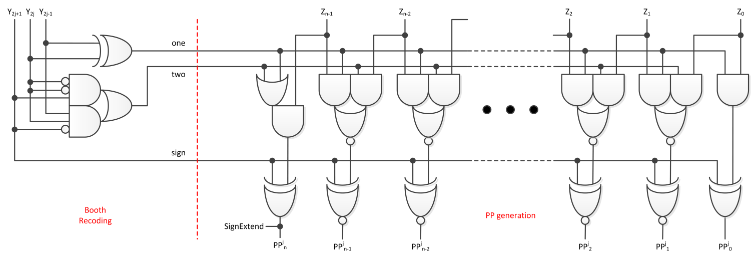
Table 4.1: Recoding of Booth digit set

y_{2j+1}	y_{2j}	y_{2j-1}	$O(y_j)$
0	0	0	0
0	0	1	z_j
0	1	0	z_j
0	1	1	$2z_j$
1	0	0	$-2z_j$
1	0	1	$-z_j$
1	1	0	$-z_j$
1	1	1	0

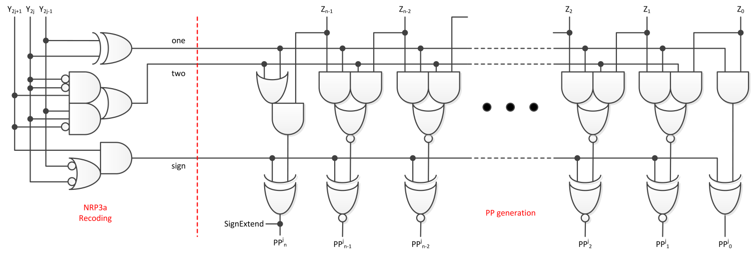
Table 4.2: Recoding of NPR3A digit set

4.1.1 Recode

The recoding component takes the multiplier input, Y, and generates a set of signals, which the partial product generator (PPG) uses to generate the partial products from the multiplicand, Z. How the partial product are generated depends on the recoding scheme, an its digit sets. The most simple digit set is that of the Radix-2, with the digit set {0,1}. An example of this is equation 4.4e. Using a radix-4 recoding scheme only requires half the partial product of a radix-2 scheme, why a radix-4 or higher is favoured for area saving. Generating partial product form a Radix-4 digit set or higher radix can require pre addition, something the delay will suffers under. In this project only radix-4 multipliers are used. The best known radix-4 recoding digit set is that of Booth [MDETL04, p. 197], with the digit set {-2,-1,-0,0,1,2}, recoded described in TAB 4.1. Implementation in FIG 4.2a. The recoding scheme NPR3a, proposed in [ZH03, p. 33, NPR3a], recodes Y into the digit set {-2,-1,0,1,2}, which avoids the generation of -0, recode description in TAB 4.2. See FIG 4.2b for NPR3a implementation. Avoiding the generation of -1, reduces the switching activity in PPC and CLA, but does introduce more area and delay in the recoder.



(a) Booth implementation



(b) NRP3a implementation

Figure 4.2: Implementation of Recoding and PP generation

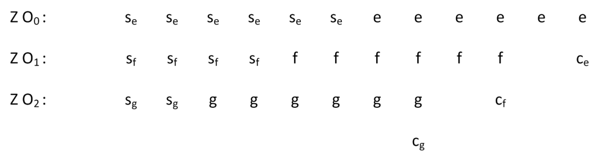
Booth and NRP3a represent their digit set with tree signals $two_j, one_j, sign_j(c_j)$, see FIG 4.2a and FIG 4.2b. The Booth and NRP3a recoded is interchangeable, which makes it possible to change between the two recoders without have to change the PPG, PPR or CPA.

4.1.2 Partial Product Generator (PPG)

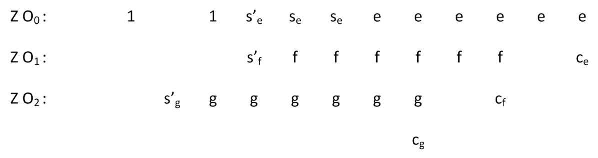
The partial product generator, generate the products which is accumulated in PPR to two bit vectors which is added together in the CPA given the multipliers result. Each partial product (PP_j) is generates as a product of the recorders digit set $(O(y_j))$ and the multiplicand, Z. Equation 4.5 describes the partial product generated by a Radix-4 recoding scheme. The shift 4^j is added to keep the right alignment between partial products.

$$PP_j = O(y_j) \times Z \times 4^j \tag{4.5}$$

The partial product $Z_j \times \{0, 1\}$ is easily generated. The $Z_j \times 2$ is generated with a left shift. The partial product $Z_j \times \{-2, -1, -0\}$ is generated by taking their respectable positive and inverting their bit values adding one. But instead of doing the addition before hand, the partial product is kept in a redundant number format, EQ 4.6. Giving the PPC the partial product in a redundant


Z O0: se se se se se se e e e e e e
Z O1: sf sf sf sf f f f f f f ce
Z O2: sg sg g g g g g g cf
cg

(a) Partial product generated, for a 6x6 multiplier



(b) Reduced complexity of the partial product generated, for a 6x6 multiplier

Figure 4.3: Shape of Radix-4 partial production generation

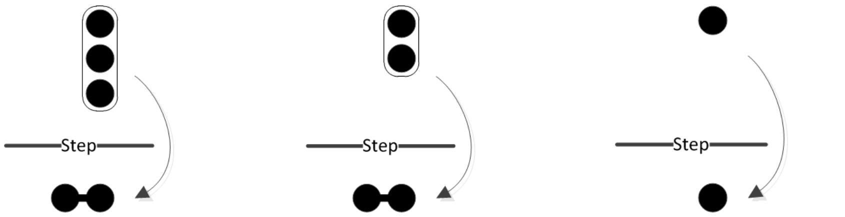
number format, hides the addition delay between the reduction of partial products. The one needed to be added is defined as a carry C_j and corresponds to the signal $Sign_j$. This gives the partial product generated the shapes in FIG 4.3a. The partial product generator implementation is given in FIG 4.2.

$$PP_j = (Sing_j \oplus (Z \times one_j + 2Z \times two_j)) \times 4^j, C_j \times 4^j \tag{4.6}$$

Because Booth, NRP3a and the two's complement system, the partial products is signed, which means each partial product is sign extended to the width of the results length. To reduce the numbers of bit needed to represent a given partial product, the scheme given in [MDETL04, p. 202] is used, which turns FIG 4.3a into FIG 4.3b.

4.1.3 Partial Product Reducer (PPR)

The PPR takes the partial product generated and reduced them into two bit vectors. A carry propagate adder uses the two bit vectors to calculating the final result. The generated partial products FIG 4.3b, is reduced using (3,2] counters and (2,2] counters. In practise a (3,2] is a full adderand a (2,2] is a half adder. The partial product is reduced in steps, where each counter, "transforms" the columns into rows and unused bits are move to the next step, as displayed in FIG 4.4. Figure 4.5 shows how the partial product from a 6x6 bit multiplier is reduced to two bit vectors. The reduction used in this project follows a very strict method and are parted into two stages: the general stage where atleast one column has more than 3 entries & the final stage where there are at most 3 entries in all columns, see algorithm 2 and 3 for general and final reduction steps.



(a) (3,2] CSA reduction step **(b)** (2,2] CSA reduction step **(c)** Transfer bit to next step

Figure 4.4: Reduction by (3,2] & (2,2] counters

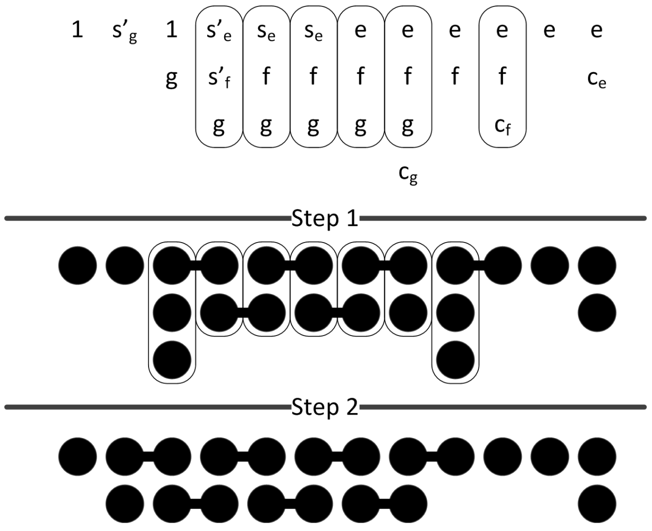


Figure 4.5: The reduction of partial products from a 6x6 bit multiplier into two bit vectors.

Algorithm 2 General method, used when atleast one column height > 3

```
$i \leftarrow \#first\ non\ empty\ column$
**for** $i < culumns$ **do**
    **if** $column_i height > 2$ **then**
        **for** $column_i height > 2$ **do**
            Reduce with (3,2]
        **end for**
        **if** $column_i height > 1$ **then**
            Reduce with (2,2]
        **end if**
    **end if**
    $i \leftarrow i + 1$
**end for**
```

Algorithm 3 Final methods, used when the columns height ≤ 3

```
$i \leftarrow \#first\ non\ empty\ column$
**for** $i < culumns$ **do**
    **if** $column_i height > 2$ **then**
        Reduce with (3,2]
    **end if**
    **if** $column_{(i-1)} height < 1$ **then**
        Reduce with (2,2]
    **end if**
    $i \leftarrow i + 1$
**end for**
```

4.1.4 Carry Propagate Adder

There are many types of CPA adders. The one used here is a 32 bit two-level carry-lookahead adder, as described in [MDETL04, p 75]. If the CLA is implemented with the correct amount of look-ahead levels it should have a logarithmic delay.

4.1.5 Area, Delay and Power Comparison

The synthesise tools and conditions is described in section 2.5. The multiplier tested is both squared 16bit parrallel Radix-4 multipliers, the difference is their recoding scheme. Booth and NRP3a is both described in SEC 4.1.1.

From table 4.3, it is clear to see that the collected area of the multiplier using NRP3a instead of Booth can be disregarded, NRP3a uses a higher area as expected, but the area increased is under 1%.

	Area $[\mu m^2]$		
	Booth	NRP3a	Difference
Recoder	190	214	12%
PP Generate	2381	2381	0%
PP Reducer	3149	3146	0%
CLA	1143	1143	0%
Total	6863	6886	<1%

Table 4.3: Area comparison between Booth and NRP3a

TAB 4.4 shows that the collected delay rises 12 % using NRP3a instead of Booth as recoder. This is a substantial increase, which is a catastrophe in modern high speed systems. It do seem that the delay is contained to the Recoder and the PPG, which could make the delay arbitrary to multiplier size. Meaning bigger multiplier will be less affected by using NRP3a that Booth as a recorder.

The power numbers speaks for them self, there are no difference between using NRP3A and Booth recoder. Both Booth and NRP3A are excellent recorders, they both have a small impact on the collected area. Booth is a faster recoder and is there for to perfer in an error free implementation, as both recorders show similar power dissipation numbers.

The NRP3a is being used in all imprecise schemes in this project, as it has the lowest power consumption. The NRP3a digit sets avoids the -0 generation,

	Delay	*[ns]*	
	Booth	NRP3a	Difference
Recoder	0.42	0.73	74 %
PP Generate	0.29	0.35	20 %
PP Reducer	0.95	0.95	0 %
CLA	1.45	1.45	0 %
Total	3.11	3.48	12 %

Table 4.4: Delay comparison between Booth and NRP3a, through the critical path

	Power	*[mW]*	
Picture	Booth	NRP3a	Difference
Random	1.935	1.939	<1 %
Barboon	1.906	1.904	<1 %
Barbara	1.856	1.855	<1 %
Goldhill	1.891	1.885	<1 %
Lena	1.863	1.861	<1 %
Peppers	1.884	1.878	<1 %
Average	1.889	1.887	<1 %

Table 4.5: Power dissipation of Booth and NRP3a for different pictures

which makes it more ideal for imprecise multiplier schemes where the carry signal is left out, as only -1 and -2 will be generated imprecise.

Because of an error in my VCS scripts I got incorrect power numbers. I got a 22% power saving using NRP3a rather than Booth as recoder. This was expected from the result from data given in table 2.14 in [ZH03]. Because of this, NRP3a is used in the project instead of Booth, as the error were discovered quite late in theproject.

4.2 Imprecise Multiplier Schemes

In this section the imprecise multiplier schemes is presented. The schemes change the way the partial products are generated. As the partial product generated changes shape, so will the PPR and CPA, influencing the area, delay, error and power consumption of the scheme. The imprecise multipliers are based on a parallel Radix-4 scheme described in section 4.1, which uses the NRP3a scheme for recoding.

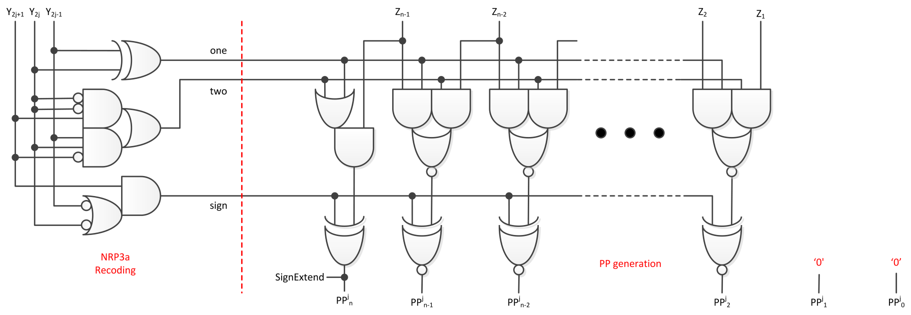


Figure 4.6: Implementation of the imprecise TEPPE$_2$ scheme, with a truncating width of 2

4.2.1 Truncating Each Partial Products End Scheme (TEPPE)

The Trunk Each Parital Product End scheme (TEPPE) as the name suggest truncates the generation of each partial product. All partial products are generated leaving out the least significant part, this reduced the logic needed generating the partial product and subsequent the logic in the compression tree and CPA. Depending on the width chosen, more or less of the least significant part of each partial product is truncated out. C_j is moved, $C_{j.teppe} = C_j \times 2^q$, towards the most significant part to accommodate the generation of minus products, {-2, -1}. The implementation can be seen in figure 4.6. The width of the truncation is given as q, TEPPE$_q$.

4.2.2 Hybrid 2 Scheme (H2)

The Hybrid 2 scheme (H2) generates it partial product in a hybrid fashion. The hybrid part consist of the PP being generated with two different digit set, a full and a crippled. The different set can be seen in table 4.6. The least significant part of the partial product is generated using a crippled version of the original NRP3a, with the digit set {-2,0,2}. A new recoding is not necessary as the NRP3a generated the correct signals already. q describes the width of the partial product which is generated by the crippled digit set. See figure 4.7a for implementation.
The generation of $PP_0^j = sign_j$, which is the same signals as C_j. C_j represent the same value as PP_0^j in this implementation and they are added together, $C_J + PP_0^J = 2 \times C_J$. A complexity reduction is achieved by leave out PP_0^j, but letting C_j represent double it original value, $C_{j.h2} = 2 \times C_j$. This gives the implementation in figure 4.7b. The generation of the hybrid part uses less logic,

y_{2j+1}	y_{2j}	y_{2j-1}	$O(y_j)$	$O(y_j)_{hybrid}$
0	0	0	0	0
0	0	1	z_j	0
0	1	0	z_j	0
0	1	1	$2z_j$	$2z_j$
1	0	0	$-2z_j$	$-2z_j$
1	0	1	$-z_j$	-0
1	1	0	$-z_j$	-0
1	1	1	0	0

Table 4.6: Recoding of NPR3A digit set, with hybrid generation

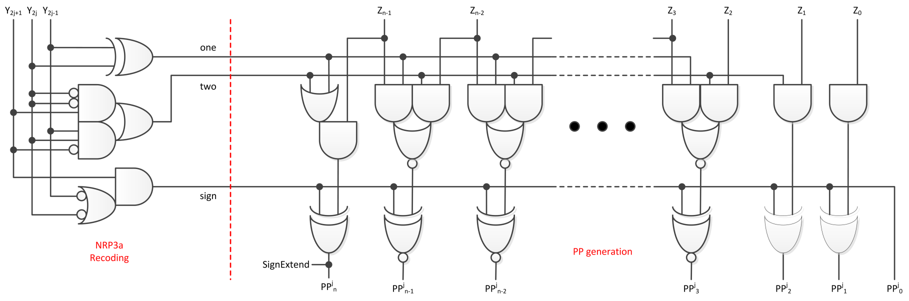
but the partial product width = n, given almost the same size compression tree and CPA as a precise implementation.

4.2.3 Leave out LowLow Scheme (LLL)

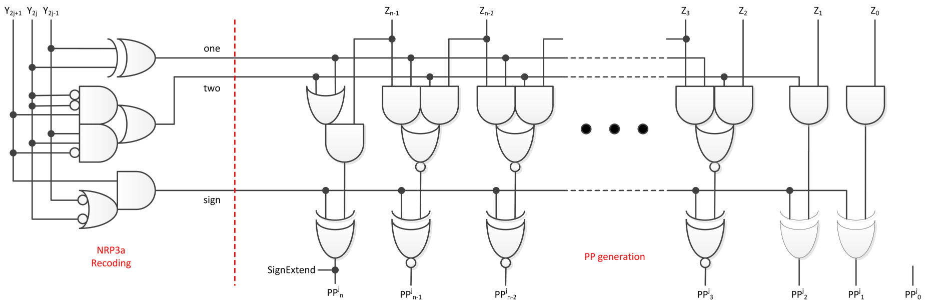
It is generally known that multiplication between two words which width exceeds the available multiplier hardware, can be parted up in smaller multiplications bites and combined. Given a square 8 bit multiplier and the wish to multiply the two 16 bit words A and B, the scheme illustrated in figure 4.8 will do just that. Part up the two big numbers in to a high and a low part. Multiply A_{Low} with B_{Low}, A_{High} with B_{Low}, A_{Low} with B_{High} and A_{High} with B_{High}, align them correctly and add them together will give the product A B. In the LLL scheme the A_{Low} x B_{Low} product is left out in the final summering, illustrated in figure 4.9. Instead of using 3 small multiplier to incorporated the scheme, the product generated by A_{Low} x B_{Low} is left out generating the partial product. q $\in \{2, 4, 6, 8...\}$ describes as the width of the low part. The partial product generated without A_{Low} x B_{Low} is illustrated in figure 4.10, the carry bit, C_j, is preserved.

4.2.4 Truncating Scheme (R4T)

The implementation is the same as used for R4TEPPE in section 4.2.1, with the exception that each partial product is truncated differently if truncated at all. All bits generated in the partial products PP_j which represents a value in the collected scheme under 2^q is truncated away. See figure for a description 4.11a. C_j is preserved and moved, $C_{j.R4T} = C_j \times 2^{(q-j)}, j * 2 < q$, towards the most significant part to accommodate the generation of minus products, {-2, -1}.



(a) Naive implementation



(b) Optimized implementation, works by applying $C_{j.h2} = 2 \times C_j$

Figure 4.7: Implementation of Hybrid 2 scheme, $q = 3$. Top is the naive implemented and bottom is the optimized implementation.

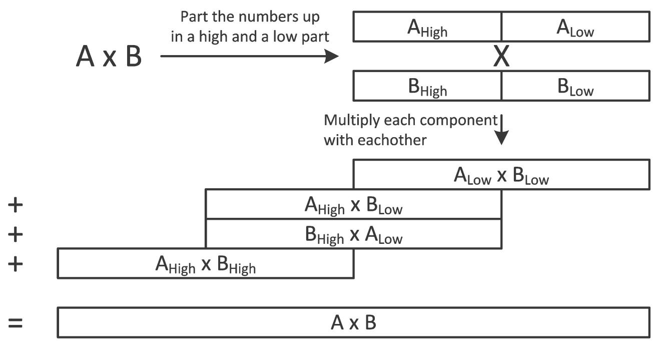


Figure 4.8: How to multiply big number on small hardware

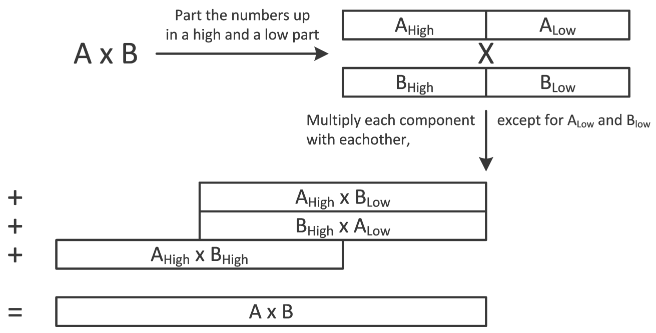

A x B
Part the numbers up in a high and a low part
A_High
A_Low
X
B_High
B_Low
Multiply each component with eachother,
except for A_Low and B_low
+ A_High x B_Low
+ B_High x A_Low
+ A_High x B_High
= A x B

Figure 4.9: LLL scheme

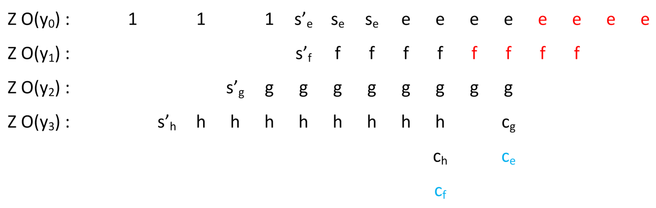
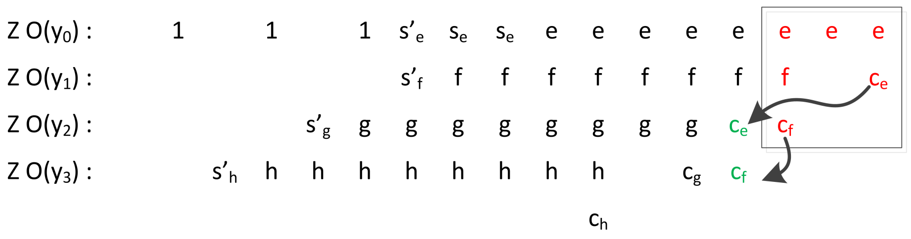
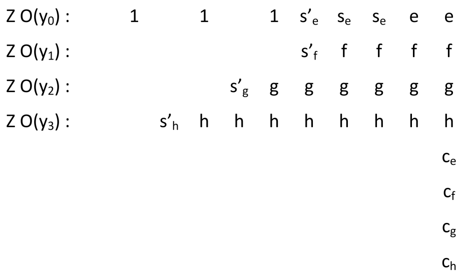

Z O(y0) : 1 1 1 s'e se se e e e e e e e e
Z O(y1) : s'f f f f f f f f f
Z O(y2) : s'g g g g g g g g g
Z O(y3) : s'h h h h h h h h h cg
ch ce
cf

Figure 4.10: The partial product generated by the LLL scheme, Red indicates removed bit, as they represents A_{Low} x B_{Low}. Blue represents carry bit which has been preserved and moved. Multiplier width = 8, scheme size = 4


Z O(y0) : 1 1 1 s'e se se e e e e e e e e
Z O(y1) : s'f f f f f f f f f ce
Z O(y2) : s'g g g g g g g g g ce cf
Z O(y3) : s'h h h h h h h h h cg cf
ch

(a) Partial product generation where red indicated truncated bits and green represents preserved carry bits, C_j


Z O(y0) : 1 1 1 s'e se se e e
Z O(y1) : s'f f f f f
Z O(y2) : s'g g g g g g g
Z O(y3) : s'h h h h h h h h h
ce
cf
cg
ch

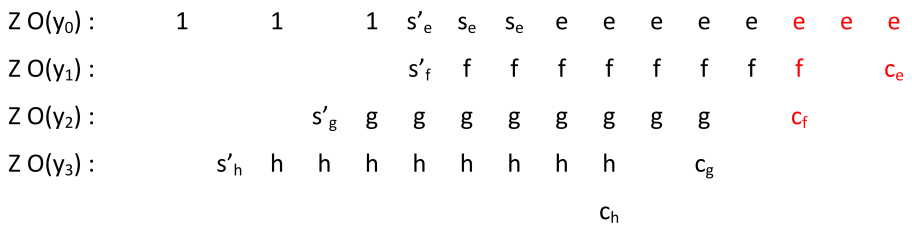
(b) Partial products generated and the preserved $C_j.R4T$ which creates an unreasonable high column

Figure 4.11: R4T scheme. Top the general partial product generation for 8x8bit multiplier $q = 3$, bottom displays the an unreasonable high column for a 8x8 multiplier with a$q = 6$

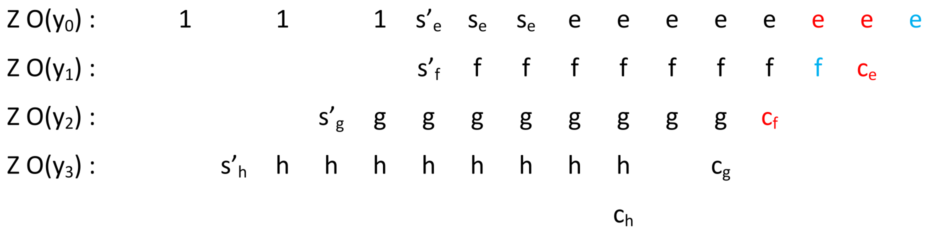
The truncation can build up $C_{0..n-1}$, which can become a problem with a high q as column height is increased dramatically. When the column height increases to much the delay reducing the partial product is increased together with the power consumption. an example of this is given in figure 4.11b, where instead of a reduction of 5 rows, the reduction is increase to 8 rows. The truncated bits creates a shorter CPA and some what smaller partial product reduction logic, but also one which can be slower because of the extra rows which needs to be compressed.

4.2.5 Hybrid 1 Scheme (H1)

The Hybrid 1 scheme use the same partial product generation as the Hybrid 2 scheme presented in section 4.2.2, the only difference is that the imprecise part of each partial product varies, if present at all. Figure 4.12a describes how the partial product is parted up in a precise and imprecise generation of PP. All bits generated in the partial products PP_j which represents a value in the collected



(a) Partial product generated, where the imprecise part of the partial product is marked red



(b) The generation of the partial product is optimized by reduction, red represent imprecise generated partial product and blue bit which is not generated

Figure 4.12: H1 schemes. Top the naive implementation, bottom the optimized version. Both 8x8 bit multiplier, $q = 3$

scheme under 2^q are imprecise encoded. The optimizing scheme for the partial production in section 4.2.2, for the R4H2 scheme holds for the R4H1 as well. Using the optimization will give the partial product generated in figure 4.12b.

4.2.6 Truncating Normal Carry Bits Scheme (TNCB)

The R4TNCB scheme is equivalent to that of the R4T scheme, with the exception that R4TNCB truncated the carry bits C_j as well, not preserving their information. This reduces the amount of bits needed to be processed by the compression tree, it especially solves the problem for preserving the Carry bits - which created high columns, as seen in figure 4.11b. An example of the partial products generated by TNCB is shown in picture 4.13.

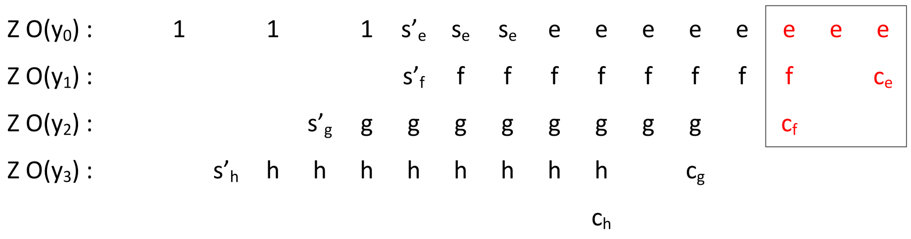

Z O(y0) : 1 1 1 s'e se se e e e e e e e e
Z O(y1) : s'f f f f f f f f f ce
Z O(y2) : s'g g g g g g g g g cf
Z O(y3) : s'h h h h h h h h h cg
ch

Figure 4.13: R4TNCB scheme The red partial product representing bit which is not generated, notice the lack of C_j preservation. 8x8bit multiplier with $q = 3$

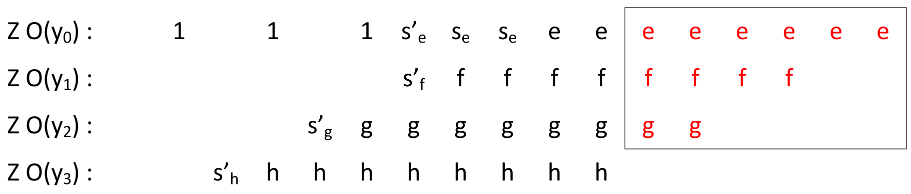

Z O(y0) : 1 1 1 s'e se se e e e e e e e e
Z O(y1) : s'f f f f f f f f f
Z O(y2) : s'g g g g g g g g g
Z O(y3) : s'h h h h h h h h h

Figure 4.14: R4TMCB scheme. The red part of the partial product is not generated. notice the total lact of carry bits C_j. 8x8bit multiplier with $q = 6$

4.2.7 Truncating Missing Carry Bits Scheme (TMCB)

The TMCB scheme is equivalent to that of the R4T scheme, with the exception that R4TMCB leaves out all carry bits C_j. As TNCB this reduces the amount of bits needs to be processed by the compression tree, it especially solves the problem when preserving the Carry bits - which created high columns, as seen in figure 4.11b. An example of the PP generation for the R4TMCB scheme is shown in picture 4.14,

4.3 Errors Generated by Imprecise Multipliers

In this chapter the error generated by the imprecise multipliers are investigated. The statical errors are found by exhausting simulation. The error generated via application is investigated through IDCT, image smoothing and edge-detection.

4.3.1 Statistical Error for Imprecise Multipliers

The error function are derived from statistically observations based on exhausting simulation. q for 1 to 8 are exhaustively simulated for 2^{32} combinations and the error observed are converted into functions for comparison. The error of TEPPE, H2 and TMCB are not only affected by q, but also the multipliers width, as each partial product is generates with an error. The error functions for the imprecise multipliers is given in table 4.7. See section 2.2 for error definition.

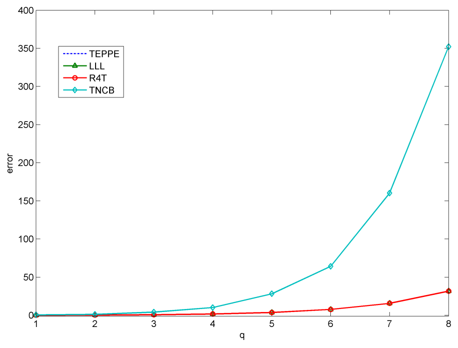
The average error, $\bar{\epsilon}$, can be divided into 4 groups. H1, H2 which have an average error of zero, independent of q. TEPPE, LLL and R4T which has the same medium error growth. TNCB which has a high average error growth. TMCB having a biased and high negative error growth. In figure 4.15a TEPPE, LLL, R4T and TNCB error development is displayed.
The average absolute error, $|\bar{\epsilon}|$, is divided into 3 groups. R4T and H1 with similar error generation and TNCB all showing low error growth. TEPPE, H2 and LLL showing the highest error growth and TMCB which is biased with medium growth rate. Figure 4.15b shows R4T, H1 and TNCB as the lowest growing error functions. Figure 4.15c shows just how much LLL grows compared to the other schemes.It looks like all others errors converts against the same limited growth.
The maximum absolute error, $|\epsilon|_{max}$, can be parted into 4 distinct groups. As seen in figure 4.15d. R4, H1 and TNCB displays the same maximum error development over q. LLL has a high growth rate, but manageable at low q'es. TMCB start biased, but grows only very little. TEPPE and H2 Starts biased and grows fast, even though LLL would exceed their maximum error at a higher q.

4.3.2 Transformation Error

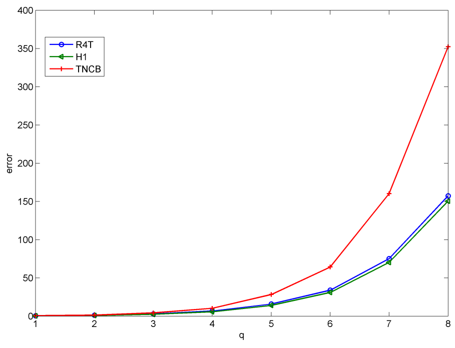
The IDC-Transformation is described in section 2.4.1. There are given two algorithm scenarios, one where A is the multiplier and B the multiplicand(AxB) and visa versa (BxA). B generally being the number with he smallest magnitude, given A and B represents integers. As the algorithm returns an integer number, from a scaling of $\frac{1}{2^{16}}$, an error which has direct influence on the end result can be generated for the following schemes TEPPE$_{p>2}$, H2$_{p>2}$, LLL$_{q>8}$, R4T$_{q>16}$, H1$_{q>16}$, TNCB$_{q>16}$ and TMCB$_{q>16}$. The generated error size is confined to the multiplier scheme and not the picture content.The errors are average over the five test pictures presented in section 2.6. The error is given as $\bar{\epsilon}^p$ which indication the average intensity change in the image. $|\bar{\epsilon}|^p$ gives the average error

Type	$\bar{\epsilon}$	$\lvert\bar{\epsilon}\rvert$	ϵ_{min}	ϵ_{max}	$\lvert\epsilon\rvert_{max}$
TEPPE$_q$	$\frac{1}{4} - 2^{q-3}$	$< 2.045^{q+12}$	$-21845 \times (2^q - 1)$	$-\epsilon_{min}$	$-\epsilon_{min}$
H2$_q$	0	$< 2.015^{q+12}$	$-21845 \times (2^q - 1)$	$-\epsilon_{min}$	$-\epsilon_{min}$
LLL$_q$	$\frac{1}{4} - 2^{q-3}$	$< 3^q$	$-\frac{(2^q-1)^2}{3}$	$-\epsilon_{min}$	$-\epsilon_{min}$
R4T$_q$	$\frac{1}{4} - 2^{q-3}$	$< 2.5 \times 2.2^q$	$-\sum_{q=1}^{q} \lceil \frac{q}{2} \rceil \times 2^{q-1}$	$-\epsilon_{min}$	$-\epsilon_{min}$
H1$_q$	0	$< 0.3 \times 2.2^q$	$-\sum_{q=1}^{q} \lceil \frac{q}{2} \rceil \times 2^{q-1}$	$-\epsilon_{min}$	$-\epsilon_{min}$
TNCB$_q$	$\left(\lceil \frac{q}{2} \rceil + \frac{q-3+2 \bmod (q+1)}{4}\right) 2^{q-2} + \frac{1}{4}$	$\bar{\epsilon}$	0	$-\sum_{q=1}^{q} \lceil \frac{q}{2} \rceil \times 2^{q-1}$	ϵ_{max}
TMCB$_q$	$< -8192.5 - 2.1^q$	$< \bar{\epsilon}$	$< -21846 - 2.2^{i+1}$	0	$-\epsilon_{min}$

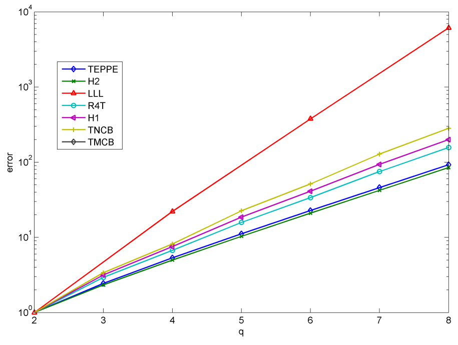
Table 4.7: Statistical derived error functions for imprecise adders. Red entries is not exact but a upper bound for the error



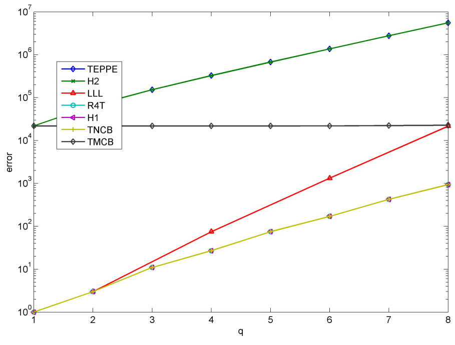
(a) Average error growth of TEPPE, LLL, R4T and TNCB over q. TEPPE, LLL and R4T is overlaying



(b) Average absolute error growth of R4T, H1 and TNCB over q. R4T & H1 is overlaying

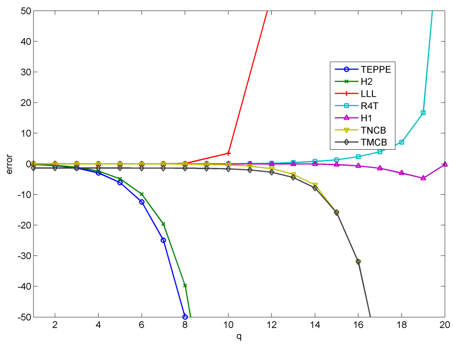


(c) Average absolute error growth of all schemes, TMCB is displayed as |TMCB|. All error growth has been normalised. Notice the logarithmic y axes

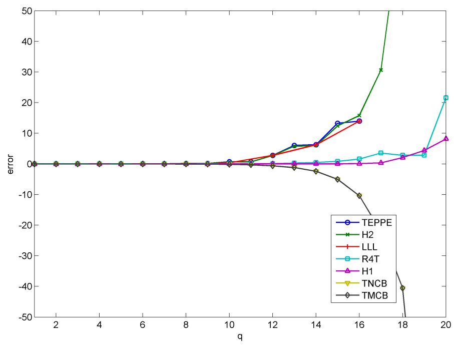


(d) Maximum absolute error growth for all schemes over q. Notice the logarithmic y axes. TEPPE and H2 over lays, so do R4T, H1 and TNCB

Figure 4.15: Graphs based on error functions in table 4.7 for imprecise multipliers



(a) $\bar{\epsilon}^p$ for IDCT, AxB



(b) $\bar{\epsilon}^p$ for IDCT, BxA

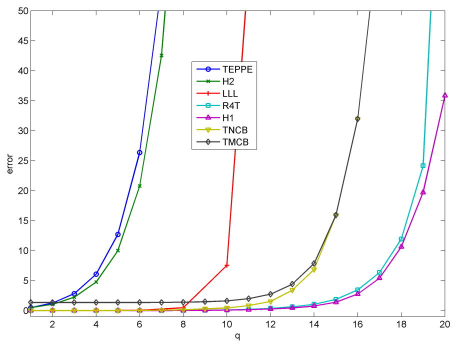
Figure 4.16: $\bar{\epsilon}^p$ for IDCT

of each pixel in the image. $|\epsilon|^p_{max}$ describes the biggest pixel error in the image, the error definition is given in section 2.3.

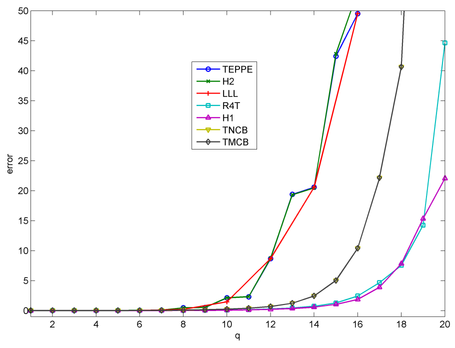
The average error, $\bar{\epsilon}^p$, displayed in figure 4.16a for AxB and 4.16b for BxA, clearly displays the difference between calculation the transformation with input order AxB or BxA. The argument order BxA being the one which generally generates the smallest error over q. For both AxB and BxA there are some similarities between schemes, TEPPE and H2, TNCB and TMCB and in some sense R4T and H1, have a similar error curve. TEPPE and H2 are especially affected by the order of the arguments, the argument order even negates the errors signs. R4T and H1, seems to be the ones which is least effected by the order of the input to the multipliers, TNCB and TMCB close after. LLL has the highest error growth rate for AxB over q compared the others schemes, but have a similar growth rate of TEPPE and H2 with reverse argument order. TNCB and TMCB produces a darker image, with a negative error, H1 produces the smallest error and seems to weave around zero. LLL and R4T generates a lighter picture, with a positive error. TEPPE and H2 goes from producing a rather negative error for AxB to a small positive error for BxA. The graphs which are discontinues generate unstable errors, give errors bigger than the width of the multiplier or adder.

The average absolute error, $|\bar{\epsilon}|^p$, displayed in figure 4.17b for AxB and 4.17b for BxA. As for $\bar{\epsilon}^p$, the error generated is greatly affected by the size of the input. TEPPE and H2 being the most affected. Again LLL has a high error growth rate for AxB but follows that of TEPPE and H2 for BxA. The errors compared for both argument orders shows that except for TNCB and TMCB, which only produces negative errors, all scheme produce both negative and positive errors. The graphs which are discontinues generate unstable errors, give errors bigger than the width of the multiplier or adder.

The maximum absolute error, $|\epsilon|^p_{max}$, shown in figure 4.18 for both AxB

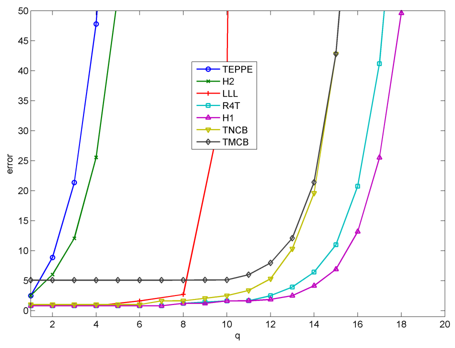


(a) $|\bar{\epsilon}|^p$ for IDCT, AxB

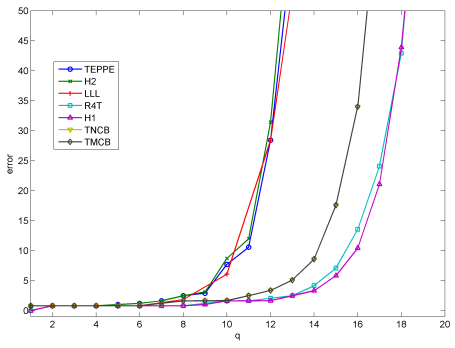


(b) $|\bar{\epsilon}|^p$ for IDCT, BxA

Figure 4.17: $|\bar{\epsilon}|^p$ for IDCT



(a) $|\bar{\epsilon}|^p_{max}$ for IDCT, BxA



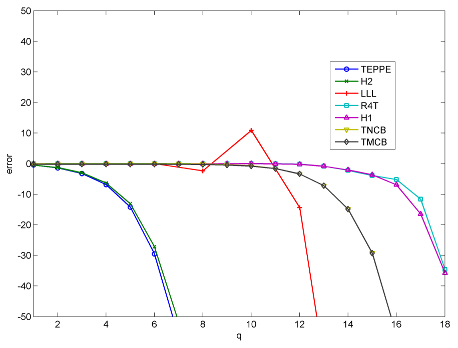
(b) $|\bar{\epsilon}|^p_{max}$ for IDCT, AxB

Figure 4.18: $|\bar{\epsilon}|^p_{max}$ forIDCT

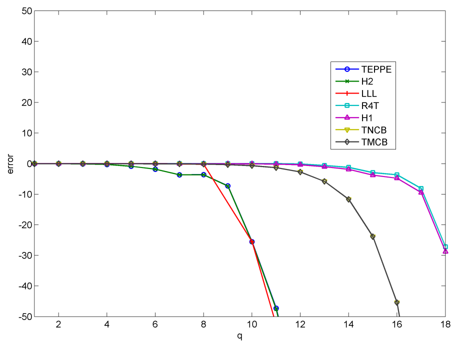
and BxA. $|\epsilon|^p_{max}$ seems to follow the same tendencies observed in $\bar{\epsilon}^p$ and $|\bar{\epsilon}|^p$, where the order has great influence on the errors generate. Again TEPPE, H2 and LLL, TNCB and TMCB and R4T and H1 have similar error growth rates for AxB. TEPPE, H2 and LLL generating the biggest error, then TNCB and TMCB, R4T and H1 having the smallest. R4T and H1 are the least affected by the order of the arguments.

4.3.3 Smoothing Error

The image smoothing is described in section 2.4.2. There are given two algorithm scenarios, one where A is the multiplier and B the multiplicand(AxB) and



(a) $\bar{\epsilon}^p$ for smooth, AxB



(b) $\bar{\epsilon}^p$ for smooth, BxA

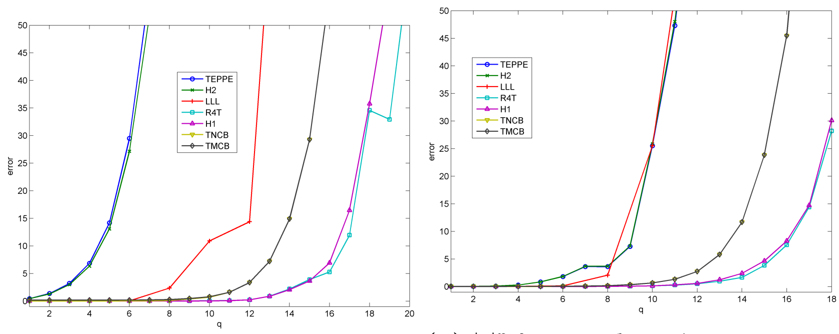
Figure 4.19: $\bar{\epsilon}^p$ for smooth, TNCB and TMCB cover each other

visa versa (BxA). B generally being the number with he smallest magnitude, given A and B represents integers. As the algorithm returns an integer number, from a scaling of $\frac{1}{2^{16}}$, the introducing of a direct error into the end result can be generated by the following schemes $\text{TEPPE}_{p>2}$, $\text{H2}_{p>2}$, $\text{LLL}_{q>8}$, $\text{R4T}_{q>16}$, $\text{H1}_{q>16}$, $\text{TNCB}_{q>16}$ and $\text{TMCB}_{q>16}$. The generated error size is confined to the multiplier scheme and not the picture content.The errors are averaged over the five test pictures presented in section 2.6.The error is given as $\bar{\epsilon}^p$ which indicates the average intensity change in the image. $|\bar{\epsilon}|^p$ gives the average error of each pixel in the image. $|\epsilon|^p_{max}$ describes the biggest pixel error in the image, the error definition can be found in 2.3

The average error, $\bar{\epsilon}^p$, development over q is shown in 4.19. R4T, H1, TNCB and TMCB seems almost unaffected by the order of the arguments, but performs slightly better with BxA. TEPPE and H2 is affected the most where the input order BxA is al lot better than AxB. As the only one LLL seems to behave best with the argument order AxB. All except LLL seems to make the picture darker, with general negative errors for both argument ordering.

The average absolute error, $|\bar{\epsilon}|^p$, over q is displayed in 4.20. TEPPE and H2 are most affected by the order of the arguments and they have the biggest error growth.

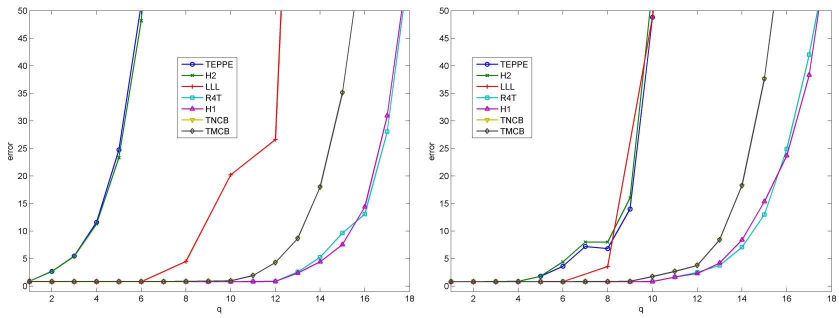
The maximum absolute error, $|\bar{\epsilon}|^p_{max}$, over q is displayed in 4.21. The input argument BxA have the lowest error generation for all multiplier schemes except LLL which favours the AxB order. TEPPE and H2 errors growth fast, R4T and H1 has the lowest growing error, followed closely by TNCB and TMCB.



(a) $|\bar{\epsilon}|^p$ for smooth, AxB.

(b) $|\bar{\epsilon}|^p$ for smooth, BxA. TEPPE and H2 cover each other

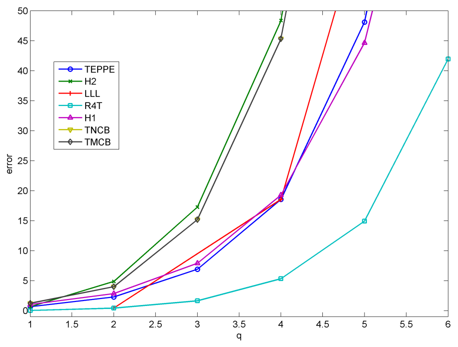
Figure 4.20: $|\bar{\epsilon}|^p$ for smooth, TNCB and TMCB cover each other



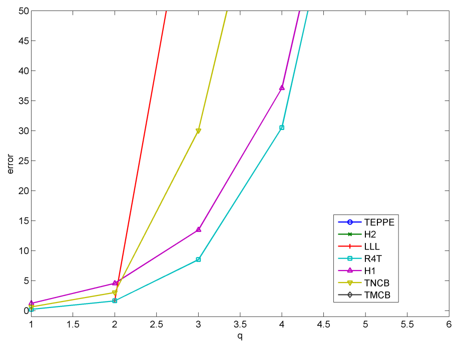
(a) $|\bar{\epsilon}|^p_{max}$ for smooth, AxB

(b) $|\bar{\epsilon}|^p_{max}$ for smooth, BxA

Figure 4.21: $|\bar{\epsilon}|^p_{max}$ for smooth, TNCB and TMCB cover each other



(a) $\bar{\epsilon}^p$ for edge-detection, AxB



(b) $\bar{\epsilon}^p$ for edge-detection, BxA

Figure 4.22: $\bar{\epsilon}^p$ for edge-detection

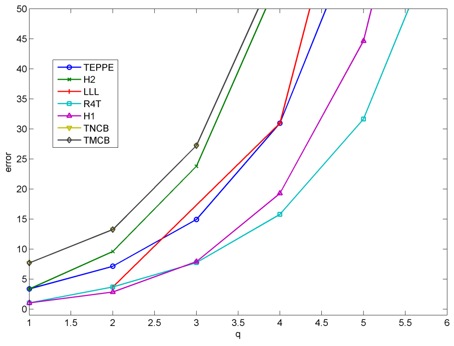
4.3.4 Edge-detection Error

The edge-detection is described in section 2.4.3. There are given two algorithm scenarios, one where A is the multiplier and B the multiplicand(AxB) and visa versa (BxA). A generally being the number with the smallest magnitude, given A and B represents integers. Independent of q, all schemes has the possibility to generate a direct error in the final result. The generated error size is confined to the multiplier scheme and not the picture content.The errors are averaged over the five test pictures presented in section 2.6.The error is given as $\bar{\epsilon}^p$ which indicates the average intensity change in the image. $|\bar{\epsilon}|^p$ gives the average error of each pixel in the image. $|\epsilon|^p_{max}$ describes the biggest pixel error in the image, the error definition can be found in section 2.3.

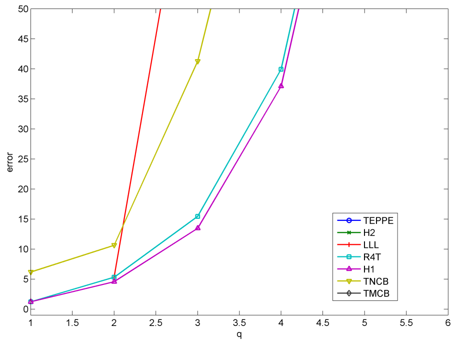
The average error, $\bar{\epsilon}^p$, over q is illustrated in figure 4.22. All schemes produces a general brighter picture than the error free multiplier, generating positive errors. All schemes performs better with the AxB ordering of the multiplier arguments. The errors produced by TEPPE, H2 and TMCB using BxA are all over 200 and not displayed in 4.22b. As all errors are direct errors, even a small change of q alternating the end result substantial. R4T performs the best no matter the input argument order and is the only one which is usable for $q \leq 5$. For AxB, H2, TNCB and TMCB performs similar but the worst, $q \leq 3$ is usable. LLL, TEPPE and H1 performs better, but only slightly, as $q \leq 4$ is usable.

The average absolute error, $|\bar{\epsilon}|^p$, over q is illustrated in figure 4.23. The curves are similar as those in 4.22 for $\bar{\epsilon}^p$ but are scales slightly higher. This means that most errors are positive but some negative errors occurs.

The average absolute error, $|\bar{\epsilon}|^p_{max}$, over q is illustrated in figure 4.23. $|\bar{\epsilon}|^p_{max}$ follows $\bar{\epsilon}^p$ with a positive scaling between 2 and 4 times that of $|\bar{\epsilon}|^p$. Meaning that the average absolute error is around half to a quarter of the maximum error in the picture.

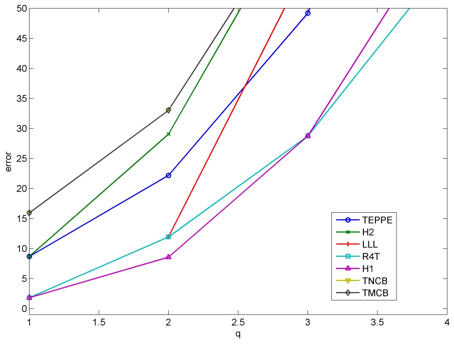


(a) $|\bar{\epsilon}^p|$ for edge-detection, AxB

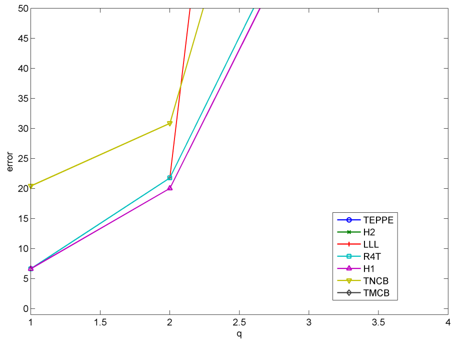


(b) $|\bar{\epsilon}^p|$ for edge-detection, BxA

Figure 4.23: $|\bar{\epsilon}^p|$ for edge-detection



(a) $|\bar{\epsilon}|^p_{max}$ for edge-detection, AxB



(b) $|\bar{\epsilon}|^p_{max}$ for edge-detection, BxA

Figure 4.24: $|\bar{\epsilon}|^p_{max}$ for edge-detection

4.3.5 Error Discussion

For all applications and imprecise multipliers it is clear that using the argument with the smallest magnitude as the multiplier and the largest magnitude as the multiplicand gives the best result. For short, this means that the smallest number recodes the fewest partial products with the biggest value. This makes sense, as fewer imprecise partial product is generated, the less error is introduced into the final result. The magnitude of the arguments to the multiplier especially affected TEPPE, H2 in IDCT, image smoothing and edge-detecton and TMCB especially in edge-detection. This is not a quality wanted in a multiple, as it applies extra work on the programmer to create the most precise program, an error free multiplier will give the same result no matter the order of the input. The faster an error grows the harder it will be to find a good q for a given scheme and precision, given this TEPPE, H2 and LLL are not preferable when choosing an IDCT precision, as their error grow the fastest. In smoothing and edge-detection, given some schemes start biased and some start generating errors late, they almost all have the same growth, except for LLL which error seems to grow with twice the speed. The statistically error functions given in section 4.3.1 is not applicable directly to any application, the reason being that they do not take error accumulation into account. For instance $\bar{\epsilon}$ for TEPPE$_q$, LLL$_q$ and R4T$_q$ are exactly the same over q, but there are no evidence in this when applied to the IDCT, image smoothing or edge-detection application. The bests imprecise multipliers seen from the error prospective is R4T and H1, TNCB as a close third. Their performance are stable over q given a smooth curve in all application. They are easy to adjust to a wanted error size, as their error do not leap uncontrollable. Changing the order of the input only change the error size a little, which means that they introduce a bounded error in each partialproduct.

4.4 Imprecise Multiplier Implementation

To compare the different schemes implementations when generating similar error, each scheme were tuned to maximize its imprecise part, produced $|\bar{\epsilon}|^p \leq 2$ when calculating IDCT. As there are two error curves depending on the order of the input argument to the multiplier, the worst is chosen to limit the error of a worst-case picture. Table 4.8 summarise the size of the imprecise multiplier scheme and the error generated by calculating IDCT, collected from section 4.3.2. The error free multiplier implementation is a square 16bit multiplier with NRP3A recoding, as described in 4.1. The different imprecise multiplier schemes are applied to the same frame work. The synthesise tools and conditions is described in section 2.5.

Scheme	$TEPPE_2$	$H2_2$	LLL_8	$R4T_{15}$	$H1_{15}$	$TNCB_{12}$	$TMCB_{11}$
IDCT $\|\bar{\epsilon}\|^p$	1.255	1.04	0.48	1.87	1.41	1.53	1.99
IDCT $\|\epsilon\|^p_{max}$	8.85	6.00	2.67	10.98	25.52	5.31	6.00

Table 4.8: The size of the imprecise multiplier schemes given as q, together with $|\bar{\epsilon}|^q$ and $|\epsilon|^q_{max}$ average over the test pictures

	Area $[\mu m^2]$							
	EF	$TEPPE_2$	$H2_2$	LLL_8	$R4T_{15}$	$H1_{15}$	$TNCB_{12}$	$TMCB_{11}$
Recoder	214	204	206	209	205	210	206	206
PPG	2381	2118	2241	1829	1272	2066	1651	1758
PPR	3146	2758	2953	2472	1866	2953	2205	2238
CPA	1143	1028	1104	896	650	1104	686	839
Total	6886	6109	6504	5405	3993	6331	4748	5041
$Ratio_{EF}$	1.00	0.84	0.94	.78	.58	.92	.69	.73

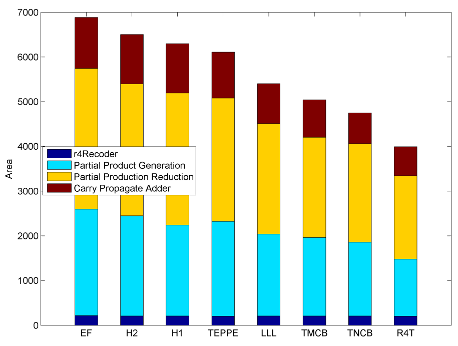
Table 4.9: The area of the multiplier schemes divided into main component, compared to a precise implementation.

4.4.1 Area Comparison

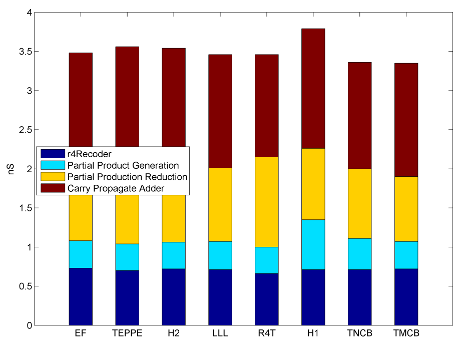
The area of the different implementations can be seen in table 4.9. The area is divided into the main part of the multiplier. As all schemes uses the same recoder, this area is similar. The partial product generation is where the different schemes differ, the compression tree and CPA is based on the partial product size and shape and will therefore vary with the schemes. $R4T_{15}$ uses the smallest area then $TNCB_{12}$, with 58% and 69% of the error free implementation. The reason for $R4T_{15}$ "big" PPC is the preservation of carry bits. $H2_2$ and $H1_{15}$ is identical in terms of recoder, PPR and CPA, only the area of their partial product differ. Non of them saves allot of area, as they uses 94% and 92% of the EF implementation. In figure 4.25a is the graphical representation of table 4.9. It is clear to see that the area of the recoder do no change.

4.4.2 Delay comparison

See table 4.10 and figure 4.25b for graphical representation of delay numbers. The delay are divided into the main components of the multiplier and are



(a) Graphical representation of area use for the different imprecise multiplier implementation, divided into multiplier components



(b) Graphical representation of the delay through the different imprecise multiplier implementation, divided into multiplier components

	Delay [*nSec*]							
	EF	$TEPPE_2$	$H2_2$	LLL_8	$R4T_{15}$	$H1_{15}$	$TNCB_{12}$	$TMCB_{11}$
Recoder	0.73	0.70	0.51	0.71	0.66	0.71	0.71	0.72
PPG	0.35	0.34	0.56	0.36	0.34	0.63	0.40	0.35
PPR	0.95	0.88	0.96	0.94	1.15	0.91	0.89	0.83
CLA	1.45	1.64	1.44	1.44	1.31	1.53	1.36	1.45
Total	3.48	3.56	3.53	3.46	3.46	3.86	3.36	3.35
$Ratio_{EF}$	1.00	1.02	1.01	0.99	0.99	1.11	0.96	0.96

Table 4.10: Delay of each multiplier schemes through its critical part, [nS]

recorded for the critical path. The delay through each component are more or less the same. As the same hardware are to be found in the Recoder and PPG the almost equal delay through these are expected. The time through the PPR is almost identical, except for R4T and TMCB. R4T's long delay through PPR is caused by the excessive column height, generated by preserving the carry bit used for generating the products {-2,-1}. TMC's small delay through PPR, caused by the total lack of carry bits. The only serial delay in the scheme are caused by the error free CPA and seems to compensate the delay through the preceding parts. Even though the critical path varies through the main components the total delay is more or less the same. H1 is the only scheme which has a delay which lies away from the error free implementation with a 11%. The reason is contributed the partial product generation which apparently, to optimize the power dissipation, introducing slow gates.

	Power [$mWatt$]								
	Error free	EF_{TEPPE_2}	$TEPPE_2$	$H2_2$	LLL_8	$R4T_{15}$	$H1_{15}$	$TNCB_{12}$	$TMCB_{11}$
Random	1.939	1.820	1.693	1.846	1.551	1.142	1.615	1.364	1.477
Barboon	1.904	1.795	1.669	1.811	1.521	1.119	1.587	1.344	1.456
Barbara	1.855	1.740	1.624	1.761	1.483	1.091	1.531	1.310	1.418
Goldhill	1.855	1.772	1.651	1.790	1.507	1.107	1.563	1.329	1.438
Lena	1.861	1.746	1.630	1.766	1.487	1.093	1.536	1.314	1.421
Peppers	1.878	1.766	1.644	1.782	1.502	1.102	1.555	1.324	1.433
Average	1.882	1.773	1.652	1.793	1.508	1.109	1.563	1.331	1.441
$Ratio_{EF}$	1.00	0.94	0.88	0.95	0.80	0.59	0.83	0.71	0.77

Table 4.11: Power consumption of different pictures on different schemes [mW], worst case

4.4.3 Power Comparison

See table 4.11 for power numbers. The test vectors contains the same information as calculation IDCT for the test pictures. The random generated test vectors consumes more power than calculation IDCT, this can be contributed to the similarity between test vectored calculating IDCT, reducing switching activity. All imprecise schemes uses less power than the error free implementation. $R4T_{15}$ uses 59%, $H2_2$ 92% of the error free implementation. EF_{TEPPE_2} is an error free multiplier where the two least significant bits of the multiplicand input has been frozen to 0. The scheme will perform similar to $TEPPE_2$ regarding the error generation. It places itself between the precise implementation and $TEPPE_2$ power wise, saving 6% power. Freezing the input of a multiplier do not stop all switching activity further down the logic, as generating the partial product can inverse the frozen input. In figure 4.25 the power dissipation for the different part of the multiplier is show. The reduction of the partial product dissipates most power, then CPA, PPG and the recoder consumes the least power.

4.4.4 Implementation discussion

The delay of the multiplier is rather even, but H1 has a very slog path through PPG, this is reasoned to be caused by the lack of timing constrains and emphasis

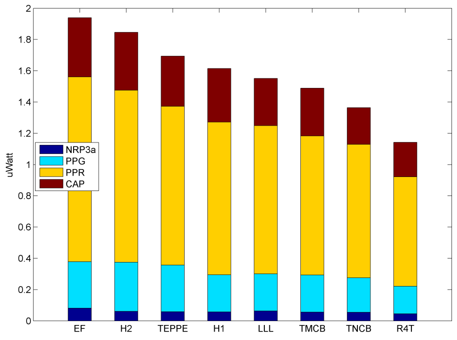


Figure 4.25: Graphical representation of power dissipation for each component in the different imprecisemultipliers

on low power synthesised. The area and power consumption do follow each other and to some degree the size of the imprecise scheme for TEPPE, R4T, TNCB and TMCB. R4T have the smallest power consumption followed by TNCB and TMCB, as they uses similar schemes this is not uprising. H2 and H1 which tries to estimate the imprecise part of the partial product loses on area and power to the other schemes, as the generation of the partial products is only $\frac{1}{3}$ of the area and power and $\frac{1}{10}$ of the delay for the EF circuit.

4.5 Conclusion for Imprecise Multipliers

The statistical generated errors indicated that TMCB were never going to produce usable numbers for any application as its average error started at 8000 for $q \leq 1$ and only grow. But in transformation, image smoothing it showed good performance. This is a good exampled of how the different imprecise schemes behaves a bit irrational depending which application they performs. Chancing the order of the inputs also changed the error generated, the most effected schemes were TEPPE and H2, where R4T and H1 seems the least sensitive to the arguments input order. Except for LLL, all imprecise multipliers performed the best when the smallest input were the multiplier and the biggest the multiplicand. This input order gives the fewest generated partial product, as the multiplier input recodes the digit set for the partial product. Depending the application the error were prevailing negative or positive. One solution size imprecise multiplier do not fit all application, which is evident comparing transformation, image smoothing and edge-detection. The same size error is not achievable by the same schemes in all tree application, not even for the smallest possible imprecise schemes. For all application goes that R4T have an excellent

behaviour in the sense that the errors grow slowly, making it easy to find a participial size schemes, which fit the error size wanted. There is no big difference between the different implementation delay except for $H1_{15}$, which delay has increased 11% from precise implementation. Area wise $H2_2$ are the biggest, closely followed $H1_{15}$, of the imprecise multiplier schemes, this is not surprising as their partial product are almost the same size as the precise implementation. their reduction of logic generating the partial product is not sufficient to compensate for the large partial product compared to the other schemes. There is a big difference between the difference schemes power dissipation, $TEPPE_2$ and $H2_2$ saves 10%, LLL_8 and $TMCB_{12}$ saves 20%, TNCB 30% and $R4T_{15}$ saves 40%, all performing IDCT with the same error loft.
The multipliers can be parted into two groups. TEPPE, H2, LLL, TMCB all have mixed precise and imprecise values in their generated partial products, meaning that precise generate values is added with an imprecise generate value with the same weight. The other group being R4T and TNCB which do not mix precise and imprecise generates partial product. R4T is clearly the best imprecise multiplier closely followed by TNCB when comparing the trades off between error andpower.

Chapter 5

Multiply and Accumulate

The Multiply and ACcumulate (MAC) scheme is present in most if not all DSP chips in one form or another. It is a piece of hardware which has been created to reduce the amount of operations needed to perform most signal processing tasks. Many signal processing algorithms has a major part which can be written in the form $\sum a \times b$, the MAC performing exactly this operation. A MAC is a multiplier where the result from its previous calculation is reintroduced through a register into the PPR. As the compression tree reduces the partial product in parallel, the extra partial product being the previous result, do not contribute substantial to the overall delay. Figure 5.1 shows a MAC, using a (3,2] Carry Save Adder (CSA), for introduction of the previous result.This MAC construction is used when testing the imprecise multipliers and adder schemes together.

5.1 Errors Generated by Imprecise MAC

MAC - Combining Exact Multiplier with Imprecise Addition Schemes. Image application suffers under the hard separation of the precise and imprecise part of the adder. When combining an imprecise adder scheme with an exact multiplier into a MAC, see 5.1, most adder schemes performances rise substantially. The performance improvement is contributed the CSA32 in the adders

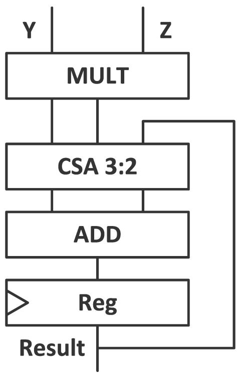


Figure 5.1: A MAC with the adder in the critical path

path. The CSA32 blurs the border of the precise and imprecise part of the imprecise adder, as it "incorporates" a carry chain with a width of one, allowing data from the imprecise part to influence the precise part. Figure 5.2 and 5.3 shows $|\bar{\epsilon}|^p$ for transformation and image smoothing for standalone adder and MAC, both featuring an imprecise addition scheme. Except for Trunc which performs worse, all other imprecise additions schemes performs better in a MAC. Both image smoothing and edge-detection favours Freeze0.5 with a good margin to the next. Freeze0.5 only generate half the error in a MAC as Carry-one, the best standalone imprecise adder. Figure 5.4 shows $|\bar{\epsilon}|^p$ for edge-detection for standalone imprecise adders and MAC's with imprecise adders. Depending on the width of the imprecise adder, edge-detection favours OR-/XOR-tail upto a width of 4 and above Trunc perform better. For edge detection Carry-one and Freeze0.5 performs worse than their standalone counter part, directly contrary to the image smoothing and transformation application where they performed the best.

MAC - Combining Imprecise Multiplier Scheme with Imprecise Addition Schemes. Given the multiplier conclusion in section 4.5, TEPPE, H2, LLL, H1 and TMCB are avoided when generating MAC's, as their accuracy or power properties are undesirable, leaving the imprecise schemes R4T, TNCB. As the MAC is a combination of an imprecise multiplier scheme and an imprecise addition scheme there is two variables which can impact performance, the width of the imprecise part of the adder and the size of the imprecise part of the multiplier. In the following sections, behaviour of the different compositions of imprecise schemes are investigated. When describing the different imprecise multipliers performance in the previous chapter, the order of the input were investigated as well, this is left out in this chapter to make it more readable. To guaranteed a certain performance the input with the worst performance were chosen.

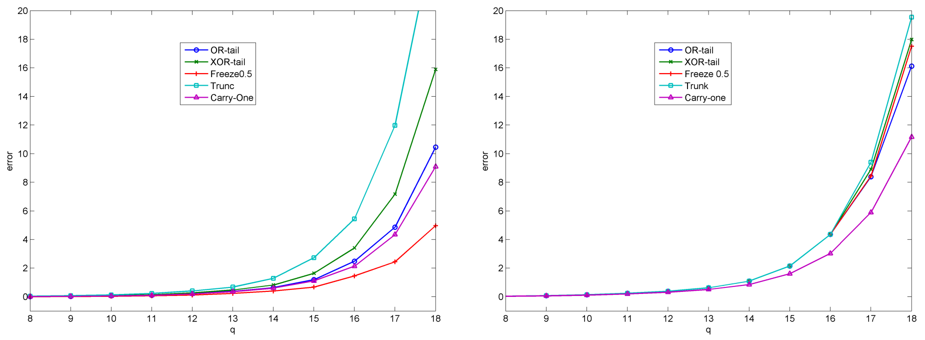


Figure 5.2: $|\bar{\epsilon}|^p$ for IDCT performed on MAC (Right) and separate multiplier+adder (Left). Both with exact multipliers and imprecise adder schemes

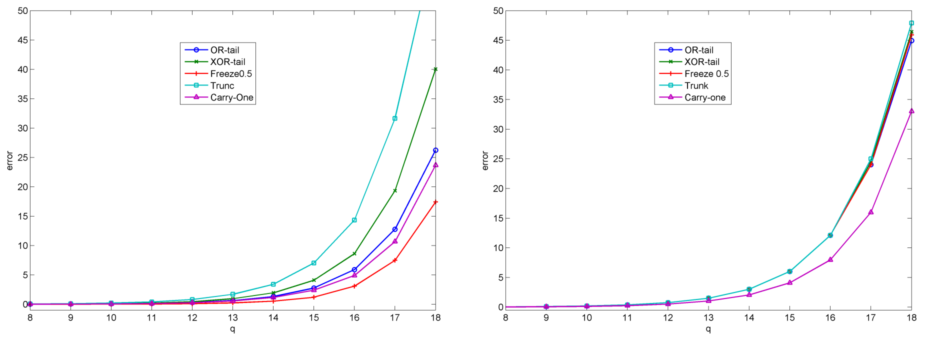


Figure 5.3: $|\bar{\epsilon}|^p$ for image smoothing performed on MAC (Right) and separate multiplier+adder (Left). Both with exact multipliers and imprecise adder schemes

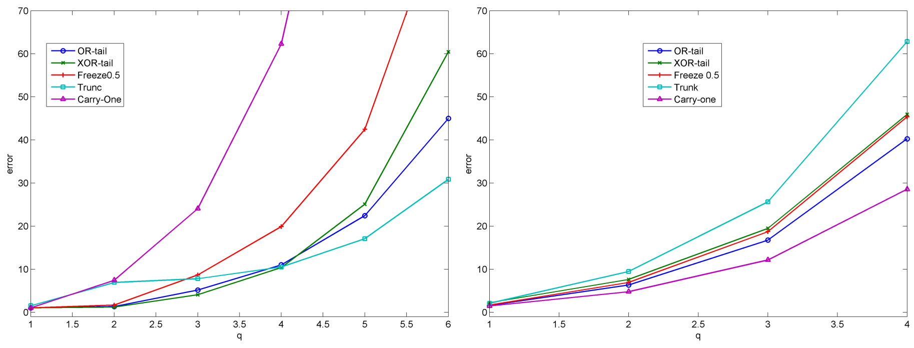


Figure 5.4: $|\bar{\epsilon}|^p$ for Edge-detection performed on MAC (Right) and separate multiplier+adder (Left). Both with exact multipliers and imprecise adder schemes. Notice the different scaled x-axle

5.1.1 Transformation Error

In figure 5.5 the $|\bar{\epsilon}|^p$ for transformation is shown for different combinations of imprecise multiplier schemes and imprecise additions schemes. The size of the imprecise part of the multiplier is kept steady and the width of the imprecise part of the adders varies. For both multiplier schemes goes that their partial product are truncated up to a certain value. This means than if the imprecise part of the adder scheme is smaller that the imprecise part of the multiplier, the different adder schemes will return the same result and basically behave as an error free adder. This effect is seen in the graph for $q \leq 14$. R4T and TNCB each favours a different imprecise adder. For R4T the OR-tail and TNCB the Freeze0.5. For both schemes goes that the imprecise adder actually reduces the error as it get wider, but only to a certain point. This means that a symbiotic relationship between imprecise multiplier and adder is achieved, compensating for each others shortcomings.

5.1.2 Image Smoothing Error

In figure 5.6 the $|\bar{\epsilon}|^p$ for image smoothing is shown for different combinations of imprecise multiplier schemes and imprecise additions schemes. The size of the imprecise part of the multiplier is kept steady and the width of the imprecise part of the adders varies. As for the transformation error, adders with an imprecise width less than the truncation width of the multiplier schemes, performs as a precise adder. R4T and TNCB both favours Freeze0.5 as their imprecise counter part. TNCB actually has a substantial performing boost using the Freeze0.5,

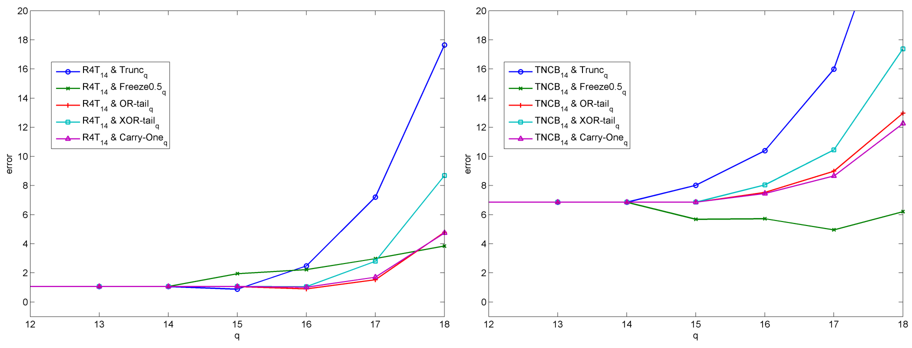


Figure 5.5: $|\bar{\epsilon}|^p$ for IDCT, for fixed width R4T$_{14}$ on the left and TNCB$_{14}$ on right with running Trunc$_q$, Freeze0.5, OR-/XOR-tail$_1$ and Carry-one$_q$

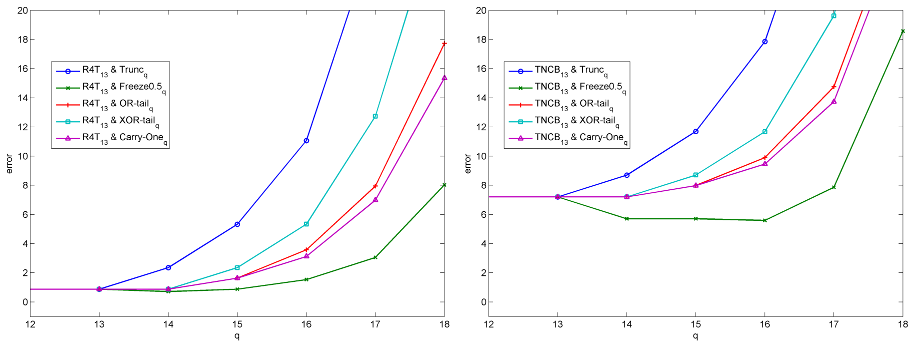


Figure 5.6: $|\bar{\epsilon}|^p$ for image smoothing, for fixed width R4T$_{13}$ on the left and TNCB$_{13}$ on right with running Trunc$_q$, Freeze0.5, OR-/XOR-tail$_1$ and Carry-one$_q$

reducing $|\bar{\epsilon}|^p$ with 1.

5.1.3 Edge-detection Error

In figure 5.7 the $|\bar{\epsilon}|^p$ for edge-detection is shown for different combinations of imprecise multiplier schemes and imprecise additions schemes. The size of the imprecise part of the multiplier is kept steady and the width of the imprecise part of the adders are varied. As seen with a separate multiplier and adder the errors is provoked instantaneous and grows fast. As with IDCT R4T favours the OR-tail and the TNCB favours the Freeze0.5 scheme. The combination

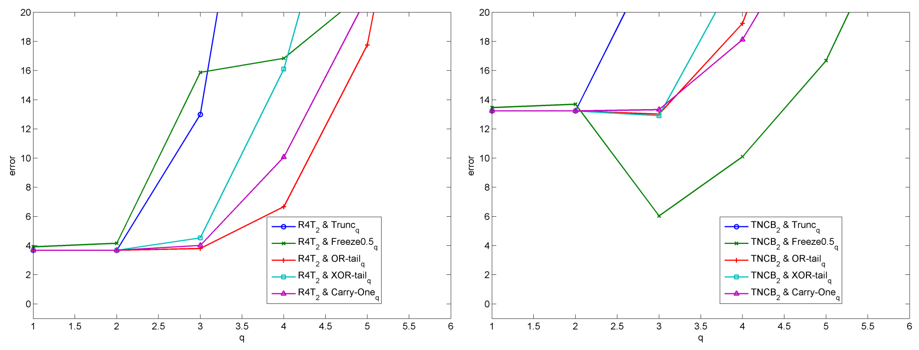


Figure 5.7: $|\bar{\epsilon}|^p$ for edge-detection, for fixed width R4T$_1$ on the left and TNCB$_{11}$ on right with running Trunc$_q$, Freeze0.5, OR-/XOR-tail$_1$ and Carry-one$_q$

of TNCB$_2$ and Freeze0.5$_3$ almost performs as good as R4T$_2$ and Or-tail$_3$, the reason can be that the constant adding of one by the Freeze0.5 scheme makes up for the missing carry bits in TNCB.

5.1.4 Error Discussion

As for the separate imprecise multiplier and adder schemes, the MAC do not give a single solution for IDCT, image smoothing or edge-detection. The extra dimension combining an imprecise multiplier with an imprecise adder, makes the error more complex. Fortunately only two imprecise adders seems to performe above the rest, being Freeze0.5 and OR-tail scheme. TNCB always performs best with Freeze0.5. R4T either performs best with Freeze0.5 or OR-tail and even performs quite bad with Freeze0.5 in edge-detection. The performance is application-, multiplier- and adder-dependent.

5.2 MAC Implementation

The error free multiplier implementation is a square 16bit multiplier with NRP3A recoding, as described in 4.1. The different imprecise multiplier schemes are applied to the same frame work. The imprecise addition schemes were implemented as a 32 bit two-level carry-lookahead adder, as described in [MDETL04, p 75]. The synthesise tools and conditions is described in section 2.5.
As the final addition in the imprecise multiplier, section 4.4.3 only uses a frac-

tion of the total power consumption of the total scheme, the imprecise adder is chosen on the basic of the imprecise multiplier. The size of the imprecise multiplier is chosen first, then the imprecise adder. The imprecise adder is custom fitted for the imprecise multiplier.
Two imprecise MAC's are pitched against an error free implementation. The MAC being R4T$_{15}$OR-tail$_{17}$ and TNCB$_{12}$Freeze0.5$_{16}$. As R4T$_{15}$ is a truncation scheme which truncated all data less that 2^{16}, the OR-tail is truncated to this width as well.

5.2.1 Area Comparison

Table 5.1 summarises the area usage for an error free MAC, the two imprecise MAC's R4T$_{15}$OR-tail$_{17}$ and TNCB$_{12}$Freeze0.5$_{16}$ and the area of an error free multiplier and adder. The area saving using an imprecise scheme is quite large, 45% for the smallest and 32% for the largest implementation. The area for a precise separate adder and multiplier exceed that of the error free MAC. The reason for the MAC's comparison with a multiplier and adder, is that their operations is needed if a MAC is not present in the system. The area for the CPA in the error free MAC exceeds that of the exact multiplier, the reason being that the introduction of CSA32 and the old result requires more entries that the PPR alone.

5.2.2 Delay Comparison

Table 5.2 summarises the delay of the MAC's. The delay through the error free MAC is increased by the delay through the CSA32 and lower entry values for the CPA compared to the error free multiplier, creating a longer critical path through the CPA. The imprecise MAC's are 6% faster than the error free MAC and around 10% slower than an error free multiplier.

5.2.3 Power Comparison

Table 5.3 summarises the power dissipation of the MAC's together with an exact multiplier and adder. The two imprecise MAC's used 54% and 66% of the error free implementation. They both use less power than if a separate error free multiplier and adder were to be used instead. Table 5.4 shows the power dissipation for the main components in the MAC, and their relation to the error free implementation. It is especially the final addition which uses significant

	Area [μm^2]					
	Error free MAC	$R4T_{15}OR\text{-}tail_{17}$	$TNCB_{12}Freeze0.5_{16}$	Error free Multiplier	Error free adder	Multiplier & Adder
Recoder	214	205	206	214		
PPG	2382	1272	1651	2381		
PPC	3147	1866	2205	3146		
CSA32	789	440	499			
CPA	1471	638	715	1143	1471	
Total	8003	4421	5277	6886	1471	8357
Ratio	1.0	0.55	0.66			1.05

Table 5.1: Area comparison between exact MAC implementation, MAC comprised of $R4T_{15}$ & $OR\text{-}tail_{17}$, MAC comprised by $TNCB_{12}$ & $Freeze0.5\text{-}tail_{16}$ and an exact multiplier and adder.

	Delay [ns]				
	Error free MAC	$R4T_{15}OR\text{-}tail_{17}$	$TNCB_{12}Freeze0.5_{16}$	Error free Multiplier	Error free adder
Recoder	0.73	0.72	0.72	0.73	
PPG	0.35	0.34	0.35	0.35	
PPC	0.85	1.03	0.99	0.95	
CSA32	0.23	0.26	0.26		
CLA	2.10	1.46	1.47	1.45	2.04
Total	4.05	3.81	3.80	3.48	2.04
Ratio	1.0	0.94	0.94		

Table 5.2: Delay comparison between exact MAC implementation, MAC comprised of $R4T_{15}$ & $OR\text{-}tail_{17}$, MAC comprised by $TNCB_{12}$ & $Freeze0.5\text{-}tail_{16}$ and an exact multiplier and adder.

	Power			[mW]		
	Error free MAC	$R4T_{15}OR\text{-}tail_{17}$	$TNCB_{12}Freeze0.5_{16}$	Error free Multiplier	Error free adder	Multiplier & Adder
Random	2.473	1.341	1.625	1.939	0.113	2.052
Baboon	2.400	1.284	1.566	1.904	0.118	2.022
Barbara	2.323	1.248	1.528	1.855	0.108	1.963
Goldhill	2.366	1.266	1.548	1.855	0.112	1.967
Lena	2.328	1.250	1.529	1.861	0.107	1.968
Peppers	2.353	1.260	1.542	1.878	0.110	1.988
Average	2.373	1.275	1.556	1.882	0.113	1.995
Ratio	1.0	0.54	0.66			0.84

Table 5.3: Power comparison between exact MAC implementation, MAC comprised of $R4T_{15}$ & OR-tail$_{17}$, MAC comprised by $TNCB_{12}$ & Freeze0.5-tail$_{16}$ and an exact multiplier and adder.

less power in the imprecise MAC compared to the error free MAC. For both imprecise MAC's, the smallest reduction in power dissipation is to be found the partial product generation.

5.3 Conclusion of Imprecise MAC

Given the many variables in creating a MAC with an imprecise multiplier and adder, an exact conclusion is almost impossible. Beside the error which changes with the application, each imprecise multiplier can be combined with each imprecise adder, both changing their error depending on the size of the precise and imprecise part and the interaction between multiplier and adder. This makes it very hard to come up with a set of imprecise multiplier and adder that works for all application. For the IDCT, the best imprecise MAC were a $R4T_{15}$ multiplier combined with a OR-tail$_{16}$ adder. Its delay were reduced by 6%, area and power dissipation with 45% for picture error which is unnoticeable for the human eye. This combination of size and imprecise schemes will not work for either image smoothing or edge-detection, which makes it very application specific.

	Power [mW]				
	EF MAC	$R4T_{15}OR\text{-}tail_{17}$		$TNCB_{12}Freeze0.5_{16}$	
	Power	Power	Ratio	Power	Ratio
Recode	0.0776	0.0438	0.56	0.0535	0.69
PPG	0.2888	0.1735	0.60	0.2177	0.75
PPR	1.1163	0.6690	0.60	0.8255	0.56
CSA32	0.3666	0.1929	0.53	0.2254	0.61
CPA	0.5245	0.1973	0.38	0.2343	0.47
Total	2.373	1.275	0.54	1.556	0.66

Table 5.4: Power comparison between exact MAC implementation, MAC comprised of $R4T_{15}$ & OR-$tail_{17}$, MAC comprised by $TNCB_{12}$ & Freeze0.5-$tail_{16}$, on a componentbasic

Chapter 6

Conclusion and Future Work

Imprecise adder, multiplier and MAC schemes has been proposed and their error investigated. Error functions for the proposed imprecise adders and multipliers has been found or been bounded. IDCT, image smoothing and edge-detection is used to investigate the error generate by imprecise adders, multipliers and MAC's running applications. The area, delay and power dissipation were obtained by gate list simulation at a common arithmetic performance making it possible to compare the error/power trade-offs, between the different schemes.

The errors that the imprecise addition schemes introduced were limited and has a nice error function which made them predictable. For the imprecise addition schemes it was found that a reduction in area, delay and power dissipation are achievable if an unnoticeable error is allowed performing IDCT. With an average maximum error per pixel of no more than 2 in the processed picture, a 49% power reduction, 22% delay reduction and an area reduction of 46% are achievable using Trunc_{14} or Freeze0.4_{14}. Using a precise addition scheme, freezing the input and allowing the same error, reduced the power dissipation with 44%, a higher power reduction than some of the proposed imprecise addition schemed. As the amount of inputs which can be frozen can be dynamical changed, the approach is more usable in a situation where a wide range of applications is used, demanding different precision.

The errors generated by some of the imprecise multiplication schemes is some what irrational at times. Multiplier schemes which mixed precise and imprecise generated partial product is the worst performing and were most affected by the size and order of the argument input. In generally using the smallest argument as the multiplier and the biggest as the multiplicand gives the smallest error. Allowing a small and unnoticeable error in the final picture, IDCT can be performed by the imprecise multiplier $R4T_{1}5$ with a 42% area reduction and 41% reduction in power dissipation. The delay stayed the same for most imprecise implementation and increased for one. Freezing some of the multipliers multiplicand input do reduce the power dissipation, but only with 6%.

The MAC is a combination of a multiplier and an adder, which accumulated the performed multiplications. The combination of an imprecise multiplier and an imprecise adder is only proposed for the best performing schemes. For some combinations of imprecise multiplier and imprecise adder the performance improves the more imprecise the addition scheme is. This improvement caused by a symbiotic relationship between the imprecise multiplier and adder is limited to a certain point where the error start to increase again. The imprecise MAC is implemented as the best performing combination of an imprecise multiplier and an imprecise adder, $R4T_{15}OR\text{-}tail_{17}$. Allowing a small error in the final picture, IDCT can be performed by an imprecise MAC. The best implemented imprecise MAC achieved a 6% reduction in delay, 45% reducing in area, and 46% reduction of power dissipation.

The biggest problem with the imprecise schemes is that one scheme does not fit all application. The schemes which performs excellent in one application both error and power wise is bound to perform adequate or not at all for other applications.

Future Work. It would be interesting to see if the MAC's performance would increase if introducing the previous result in the top of the compression tree instead of the bottom.
As one imprecise scheme cannot cover all application adequately, it could be interesting to investigate the possibility of an imprecise multiplier with adjustable precision, in the same lines as an exact adder which can freeze the input depending on the precision needed.
With integer calculation it is hard to give an upper bound for the error of each calculation, it could therefore be interesting to investigate a floating point implementation which has a maximum error of 1% per operation.

Bibliography

[AN11] Alberto Nannarelli. Slobby addition and Multiplication. Technical report, Dept.Informatics and Mathematical Modelling of DTU, 2011.

[KA06] Khalid Asyood. *Introduction to Data Compression*. Morgan Kaufmann Publishers, 2006.

[LH11] Larry Hardesty. The surprising usefulness of sloppy arithmetic. *MIT News Office*, January 2011. Available at http://web.mit.edu/newoffice/2010/fuzzy-logic-0103.html.

[MDETL04] Milos D. Ercegovac and Thomas Lang. *Digital Arithmetic*. Morgan Kaufmann Publishers, 2004.

[MP10] Alberto Nannarelli Marco Re Pietro Albicocco Massimo Petricca, Gian Carlo Cardarilli. Degrading precision arithmetic for low power signal processing. *IEEE*, 2010.

[ZH03] Zhijun Huang. High-Level Optimization Techniques for Low-Power Multiplier Design. Technical report, Univercity of California, Los Angeles, 2003.