VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

EVOLUTIONARY DESIGN FOR CIRCUIT APPROXIMATION

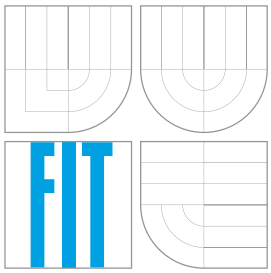
DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. PETR DVOŘÁČEK

BRNO 2015

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

EVOLUČNÍ NÁVRH PRO APROXIMACI OBVODŮ

EVOLUTIONARY DESIGN FOR CIRCUIT APPROXIMATION

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. PETR DVOŘÁČEK

VEDOUCÍ PRÁCE
SUPERVISOR

Prof. Ing. LUKÁŠ SEKANINA, Ph.D.

BRNO 2015

Abstrakt

V posledních letech klademe stále větší důraz na energetickou úspornost integrovaných obvodů. Můžeme vytvořit aproximační obvody, které nesplňují specifikovanou logickou funkci, a které jsou cíleně navrženy ke snížení velikosti, doby odezvy a příkonu. Tyto nepřesné obvody lze využít v mnoha aplikacích, kde lze tolerovat chyby, obzvláště v aplikacích ve zpracování signálů a obrazu, počítačové grafiky a strojového učení. Tato práce popisuje evoluční přístup k návrhu aproximačních aritmetických obvodů a dalších složitějších obvodů. Díky paralelnímu výpočtu fitness byl evoluční návrh osmibitových násobiček urychlen až 170 krát oproti standardnímu přístupu. Pomocí inkrementální evoluce byly vytvořeny různé aproximační aritmetické obvody. Vyvinuté obvody byly použity v různých typech detektorů hran.

Abstract

In recent years, there has been a strong need for the design of integrated circuits showing low power consumption. It is possible to create intentionally approximate circuits which don't fully implement the specified logic behaviour, but exhibit improvements in term of area, delay and power consumption. These circuits can be used in many error resilient applications, especially in signal and image processing, computer graphics, computer vision and machine learning. This work describes an evolutionary approach to approximate design of arithmetic circuits and other more complex systems. This text presents a parallel calculation of a fitness function. The proposed method accelerated evaluation of 8-bit approximate multiplier 170 times in comparison with the common version. Evolved approximate circuits were used in different types of edge detectors.

Klíčová slova

aproximace, přibližné počítání, evoluční návrh obvodů, kartézské genetické programování, měření chyby, optimalizace obvodů

Keywords

approximation, approximate computing, evolutionary circuit design, cartesian genetic programming, error measurement, circuit optimization

Citace

Petr Dvořáček: Evolutionary Design for Circuit Approximation, diplomová práce, Brno, FIT VUT v Brně, 2015

Evolutionary Design for Circuit Approximation

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pana Lukáše Sekaniny. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Petr Dvořáček
January 16, 2017

Poděkování

Rád bych poděkoval prof. Lukáši Sekaninovi za trpělivost, vedení a odbornou pomoc při řešení této práce.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In year 1965, Gordon E. Moore, co-founder of Intel company, wrote an article where he predicted that the number of transistors in integrated circuits increases two times every two years. He believed that this trend would remain unchanged for ten years. After fifty years when integrated circuits lithography is 14 nm and when mobile devices become popular, we deal with different requirement than density of integration – energy efficiency. The question is how should we design energetic efficient circuits?

Approximate computing seems to be a promising solution. It is based on the assumption that we can create efficient and energy-saving systems in the trade of higher error of the application. It was introduced a number of applications which can tolerate errors [5]. Human senses, especially sight and hearing, may not recognize the errors. This fact has been utilizing in multimedia data compression for many years. The principle can be also applied in signal filtration or computer vision. Another error tolerance can be observed in datamining in which is impossible to obtain provably optimal result. Furthermore, we can find it in the applications in which human expects errors and tolerates them; for example in weather forecasting or classification.

Approximate systems can be designed for both software (e.g. fast inverse square root calculation $f(x) = 1/\sqrt{x}$ [3]) and hardware at all levels of abstraction. Scientists presented approximate sequential and arithmetic circuits (e.g. SRAM [7], adders [26], multipliers [11], or comparators [14]) and more complex circuits (e.g perceptron [15], fast Fourier transformation or discrete cosine transformation [23]). Moreover, matrix processor was designed with a programming language for approximate computing. Nevertheless, the most widely used approximation is the interpretation of floating point numbers which represents real numbers [1].

In this work, I deal with evolutionary design of approximate circuits. This name refers to digital circuits which do not meet its logical function specification in order to reduce the size, the power or the latency. Evolutionary design allows us to create various innovative solutions which can be better than conventional solutions designed by a man. An example can be evolutionary designed antenna which was used in several NASA missions, or optical system design or analog circuits [17]. In [13] was shown that we can evolutionary design and optimize digital circuits too. In [16], [20], [21] was found that we can used evolutionary algorithms in the approximate circuit design.

The goal of this thesis is to continue on previous research in the field of evolutionary design of approximate circuits [16], [20], [21]. In work [16] was proposed a fitness function which evaluated each training vector separatly. This thesis presents a new parallel calculation of the fitness which allows to significantly accelerate the entire evolutionary design of

approximate circuits. Optimised algorithm is utilized in the design of approximated 8-bit multipliers and adders. Then the applicability of these circuits is demonstrated in the edge detection.

This work is structured as follows. The basic principles of digital circuits are described in the 2nd chapter (what circuits are made of, how to design them and what are parameters for assessment). Chapter 3 presents approximate circuits and their conventional design. Chapter 4 deals with evolutionary designs for the circuit approximation using Cartesian genetic programming . In chapter 5, edge detection is explained. Chapter 6 propose different accelerations of the evaluation and presents parallel computation of fitness. Chapter 7 analyses the found approximate circuits and also presents a new chart for an arithmetic error. Chapter 8 describes how to use approximate circuits in the edge detectors. Chapter 9 summarises the results and concludes this work.

Chapter 2

Digital Circuits

The gates are the simplest digital circuits implementing Boolean functions. Each gate have one or more inputs according to which the output responses. Inputs and outputs are modeled as digital signals acquiring just two discrete values which we call 1 and 0, in the other words, true and false (pravda, nepravda). However in the reality, gates work at an analog level where the signal is conditioned to voltage and to current [24].

Three basis gates are two-input AND gate computing conjunction, two-input OR realizing disjunction and one-input NOT which implements negation. With the using these three gates, we can evaluate any complex Boolean function (e.g. adders, multipliers, filters, parity checker). Combinational circuits is a part of the digital circuits whose outputs depend only on the given input. However, the sequential circuits depend on the state caused by previous input combinations. In the other words, sequential circuits have memory and they can „remember“ their state.

Traditional method to design simple combinational circuits is to specify them by a truth table in which for each input combination is assigned an output combination. For example, the multiplexer has three inputs thus it has 2^3 (i.e. 8) input combinations as shown in table 2.1. This table can be expressed by Boolean algebra in disjunctive normal form (abbreviated DNF) which can be minimized by DNF optimization algorithms. For multiplexer, we have an expression in the form

$$Z = \overline{S} \cdot A + S \cdot B. \tag{2.1}$$

In the next step, this expression can be converted to the corresponding combinational circuit, see Figure 2.1. For automated design of complex digital circuits, there are a number of methods that are implemented in the hardware design software.

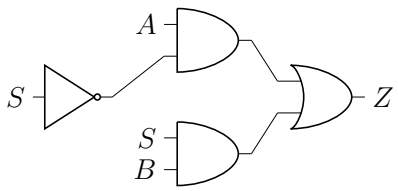


Figure 2.1: The multiplexer on the gate level abstraction

S	A	B	Z
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

Table 2.1: Truth table for the multiplexer

Integrated circuits are composed of one or more gates and they are set one one flat piece called chip. In the past, these circuits were divided into classes according to the amount of gates: small-scale integration (containing $1 - 20$ gates), medium-scale ($20 - 200$ gates) and large ($200 - 200000$ gates). Nowadays, integrated circuits have more than 50 millions transistors and we classify them as VLSI very large-scale integration [24].

The intentionally designed integrated circuits performing certain application (e.g. microcontrollers) are called ASIC, Application Specific Integrated Circuit. These circuits are optimized for specific application in all aspects. The main disadvantage is their cost price. A compromise might be to design a circuit for reconfigurable chips [24].

Reconfigurable circuits FPGA, Field-Programmable Gate Array, are composed of matrix of configurable logical blocks, CLB, which are consisted of smaller blocks called slices. Each slice has built in function generator, multiplexer, gates, registers or carry-look ahead logic. Function generators are in the other words look up tables that are built from SRAM, they can be utilized as memory or a shift register [17]. The circuit design is realized by specialized designing tools which generates configuration data for FPGA.

2.1 Digital Circuits Parameters

In the design of digital circuits, we try to optimise several parameters such as area, latency or power consumption. Moreover during the measurement, we have to select the circuit abstraction level. It is true that if we choose a lower level of abstraction then we obtain more accurate results. In addition, we must take into account the target technology for which the circuits are developed. For ASICs, we use different metrics than for FPGAs.

2.1.1 Area

The area of the circuit can be defined by the total amount of used gates or transistors by which is the circuit composed of. For example, we can design the circuit only with NAND gates due to the Boolean algebra rules and the total amount of NAND gates can be one of the metrics. If we want to be more precise, we can determinate total relative area of gates λ for VLSI integrated circuits. These parameters for each gate are listed in the table 2.2.

The more complex problem is to determinate the area of the circuit considering the information of connection wires. We can determinate the length of each connection or the number of their crossing. Due to the its complexity, I ignored all the wires in this work.

2.1.2 Latency

The delay of digital circuits can be defined as a critical path of the circuit. The critical path can be imagined as the maximal amount of visited nodes from the input to the output in the directed oriented graph representing the circuit. Inputs and outputs of the graph are in correspondence with inputs and outputs of the circuit. The nodes of the graph represents the gates of the circuit. The latency is labelled as Δ in this work.

The size of gates is in the inverse proportion to their speed. For example, inverter is smaller and faster than XNOR gate which is faster and slower. The latency can be defined by the gates composed of NAND logic [24] or the transistors.

Logical effort is one of the possible metric for measurement of the circuit delay composed of the transistors [2]. Logical effort of the gate is defined as a ratio between the input capacitance (the amount of electrical charge on the input wire) of inverter, which is needed

	NAND logic		VLSI		Logical effort		
	Gates	Latency	Transistors	λ	A	B	Avg.
NOT	1	1	2	24	1,00		1,00
NAND	1	1	4	32	1,29	1,35	1,32
NOR	4	3	4	32	1,67	1,63	1,65
AND	2	2	6	40	1,81	1,71	1,76
OR	3	2	6	40	2,09	1,99	2,04
XOR	4	3	10	72	1,50	1,60	1,55
XNOR	5	4	12	72	2,76	2,77	2,77

Table 2.2: The table shows us the number of transistors, relative area λ of gates in VLSI (for the standard of vsclib) and logical effort for input A and input B for the each gate. Taken from the publication [2]. In addition, here is demonstrated the total amount of gates and their latency in the NAND logic.

to transfer to its output, and the input capacitance of the measured gate which is needed to transfer to the its output. In the other words, it can be defined as relation $g = C_b/C_{inv}$ where C_b is the input capacitance of the gate and C_{inv} is the input capacitance of the inverter. Logical effort of the circuit is highly depended on the topology of the circuit and manufacturing technology of the transistors (if it is nMOS or pMOS transistors). The values of logical effort for chosen gates are listed in the table 2.2.

2.1.3 Power Consumption

The power consumption depends on the area of the circuit, manufacturing technology of transistors and the character of input data. Transistor?transistor logical circuits consisting of bipolar transistors are used rarely in the systems nowadays. Their successors are uniporal logical circuits which are built by CMOS technology which tries to reduce disadvantages of TTL. The main advantage is that CMOS transistors have very low static power consumption than TTL. The dynamic power consumption exhibits when transistors flips the power. It can be expressed as

$$P_{dyn} = C_{dyn} f U^2, \tag{2.2}$$

where C_{dyn} is the drain capacitance. Frequency of switching is labelled as f and U represents power voltage. It is obvious that the decrease of power voltage leads to the significant decrease of power consumption in CMOS. Nevertheless in modern lithography (65 nm and less), the share of static power consumption increases thus it should not be ignored. We can define it as

$$P_{stat} = U I_{leak}, \tag{2.3}$$

where I_{leak} is residual current and U is an input voltage [17].

Measuring of the power consumption is time expensive because it is necessary to simulate the circuit. There are numerous of the circuit simulators. Unfortunately, the most of them are commercial. From the non-commercial ones, we can use SIS which is sadly outdated and inaccurate. Thus, the perfect and accurate method is to measure power consumption in a real environment.

2.2 Design of the Arithmetical Circuits

The arithmetical circuits implement calculations in a binary system. We denote n-bit number as

$$b_{n-1}b_{n-2}\dots b_1b_0, \tag{2.4}$$

where $b_i \in \{0,1\}$ and n is the given number of bits. The most left digit refers to the most significant bit (MSB) and the most right is the least significant bit (LSB). The value in decimal base can be obtained by the formula

$$B = \sum_{i=0}^{n-1} b_i 2^i. \tag{2.5}$$

In the next text in this chapter, $+$ stands for a conjunction, $\cdot$ stands for a disjunction and $\oplus$ is an exclusive conjunction.

2.2.1 Addition

The adder is one of the most used arithmetical operation in the circuits and in the computers. The simplest adder is a half adder which adds two one bit operands x and y, and it give us a sum bit s and a carry bit c. It can be denoted as formulae

$$s = x \oplus y = x \cdot \overline{y} + \overline{x} \cdot y \tag{2.6}$$

$$c_{out} = x \cdot y \tag{2.7}$$

In the case of addition of three bits, we have a full adder which is similar to the half adder however it has a carry input c_{in}. The full adder can be defined by formulae

$$s = x \oplus y \oplus c_{in} \tag{2.8}$$

$$c_{out} = x \cdot y + x \cdot c_{in} + y \cdot c_{in} \tag{2.9}$$

Carry Ripple Adder

If we have several full adders constructed by formulae 2.8 and 2.9, then we can easily connect them in order to create a multiple bit adder. We create it in a such way that we connect output carry c_{out} of one adder to the input carry c_{in} of the other adder, on the least significant adder is c_{in} set to logical 0, see the Figure 2.2. This adder has problem with the delay caused by the transferring the carry value from the LSB to MSB. Therefore on Figure 2.2, the adder has delay 7Δ for the output s_3, Δ denotes the delay on the gate level of abstraction.

Carry Lookahead Adders

The problem with delay in the carry ripple adder can be fixed with utilizing carry lookahead adders (CLA). All we need is a calculation of the value c_{in} in the relation 2.8, so we can create a circuit according to chart on Figure 2.3. *Carry Lookahead Logic* is a circuit that determinates bit c_{in} for each bit. For the given bit j, the circuit predicts it by knowing when j-th bit generates the carry bit and when it propagates. The generating of carry bit happens when

$$g_j = x_j \cdot y_j. \tag{2.10}$$

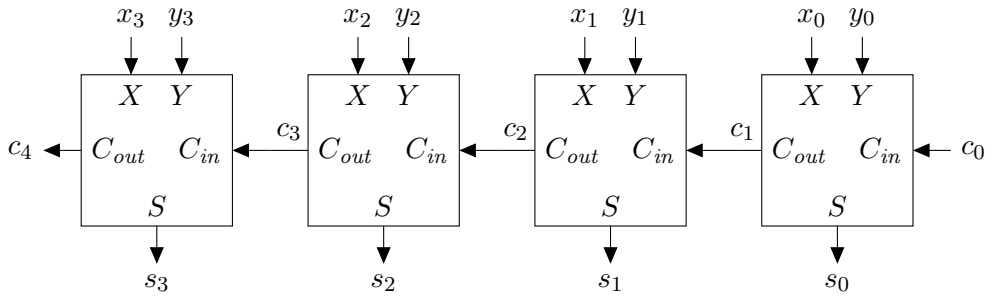


Figure 2.2: Construction of the carry ripple adder with full adders. The value of c_0 is set to logical 0.

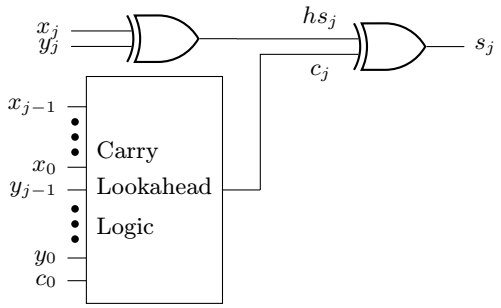


Figure 2.3: Diagram of the Adder Using Carry Lookahead Logic

Furthermore, we need the relation for the propagating of carry bit for j bit

$$p_j = x_j + y_j. \tag{2.11}$$

Neither generating nor propagating is not dependent on previous carries. The actual carry c_j can be written as a generalized expression $c_{j+1} = g_j + p_j \cdot c_j$. We expand this expression for the first four carry bits by the Formula 2.12. If we use gates with more than two inputs, then we can obtain CLA circuit with only 3Δ latency.

$$\begin{aligned} c_1 &= g_0 + p_0 \cdot c_0 \\ c_2 &= g_1 + p_1 \cdot g_0 + p_1 \cdot p_0 \cdot c_0 \\ c_3 &= g_2 + p_2 \cdot g_1 + p_2 \cdot p_1 \cdot g_0 + p_2 \cdot p_1 \cdot p_0 \cdot c_0 \\ c_3 &= g_3 + p_3 \cdot g_2 + p_3 \cdot p_2 \cdot g_1 + p_3 \cdot p_2 \cdot p_1 \cdot g_0 + p_3 \cdot p_2 \cdot p_1 \cdot p_0 \cdot c_0 \end{aligned} \tag{2.12}$$

In the case of two-input gates, we use different *carry lookahead* networks for the carry prediction. The most known is Kogge-Stone adder [10] which is named by its authors. It is an adder with time complexity of $\mathcal{O}(\log n)$ where n is the bitwidth of the operands. Carry values are calculated in parallel in the trade of used gates. The principle is in the utilizing of group calculations of Formulae 2.13 with that we create logical network as shown on the Figure 2.4.

$$\begin{aligned} p_{i,j} &= p_{i,k} p_{k-1,j} \\ g_{i,j} &= g_{i,k} + p_{i,k} g_{k-1,j} \end{aligned} \tag{2.13}$$

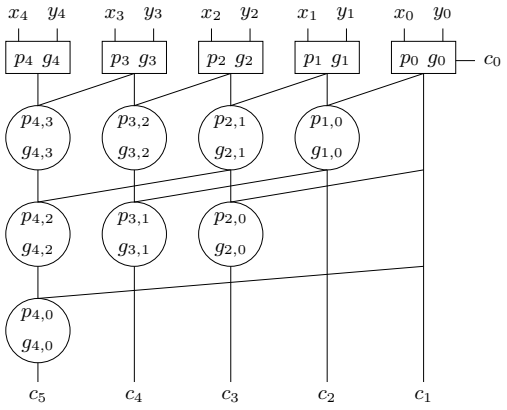


Figure 2.4: Schema of Kogge-Stone 5-bit adder with the logic of generating carries. Rectangles are calculations of the first iteration defined by 2.10 and 2.11. Circles represents the results from the formulae 2.13.

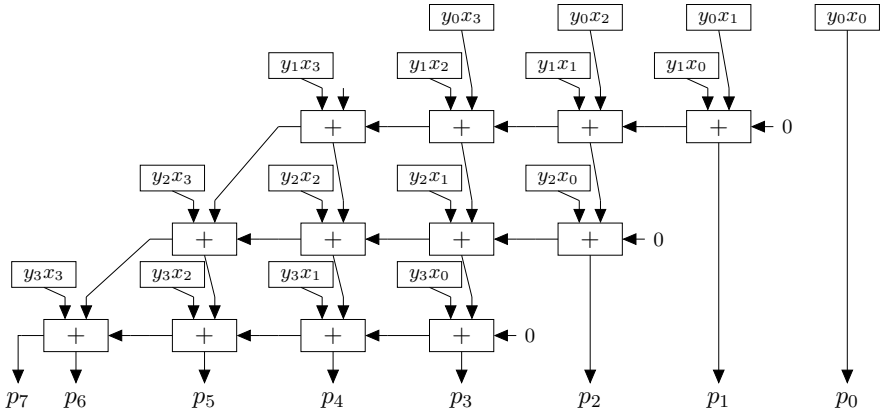


Figure 2.5: Connection of 4-bit Multiplier

We have a lot of other carry look ahead adders which work on the similar idea. For example Sklansky, Bret-Kung or more. These adders may have larger latency than Kogge-Stone, however they are composed of less gates. After all, they can be fast enough and can be much cheaper.

2.2.2 Subtraction

Binary subtraction is similar to binary addition. Subtracters can be realized as the adders with the difference that input values X or Y (exclusively) is inverted and that the input c_0 is set to the logical 1 [24].

2.2.3 Multiplication

One of the simplest method for designing the multiplier is to use the algorithm which is taught in elementary schools (standard long multiplication; shift and add). Its principle is based on the evaluation of partial products. For the building of multiplier, we need adders

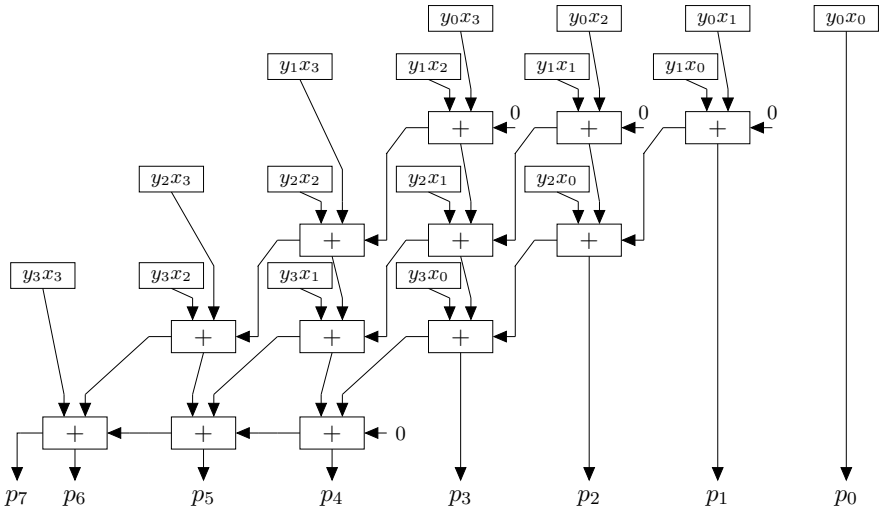


Figure 2.6: Optimized Connection of 4-bit Multiplier.

and logical gates AND. The interconnection is realized as we can see on the Figure 2.5 where $p_7p_6 \dots p_0$ is the result. The main disadvantage is in the large delay. The critical path is a passage between the input node y_0x_1 or y_1x_0 and the output p_7. Lets assume that the gate AND and the half adder have latency 1Δ and the full adder has latency 2Δ, then the delay of this multiplier is 16Δ. This delay increases with the increasing length of the operands.

The delay can be reduced by using so called carry save adders [24]. Each row is summed without carry values. However, the carries are added in the next partial product in the next step. The last step is realized by the carry ripple adder. The example of the connection is shown on the Figure 2.6. The total delay of this multiplier is 11Δ.

The last mentioned method performs seven sequential addition by the carry save adders. This number can be reduced by using so called Wallace Tree [25]. We accelerate the resulting multiplier and furthermore we reduce the area. The 8-bit multiplier using Wallace Tree can be built as shown on the Figure 2.7. It is obvious that using this method achieves a parallelization in which we need only six carry save adders and the multiplier has smaller latency.

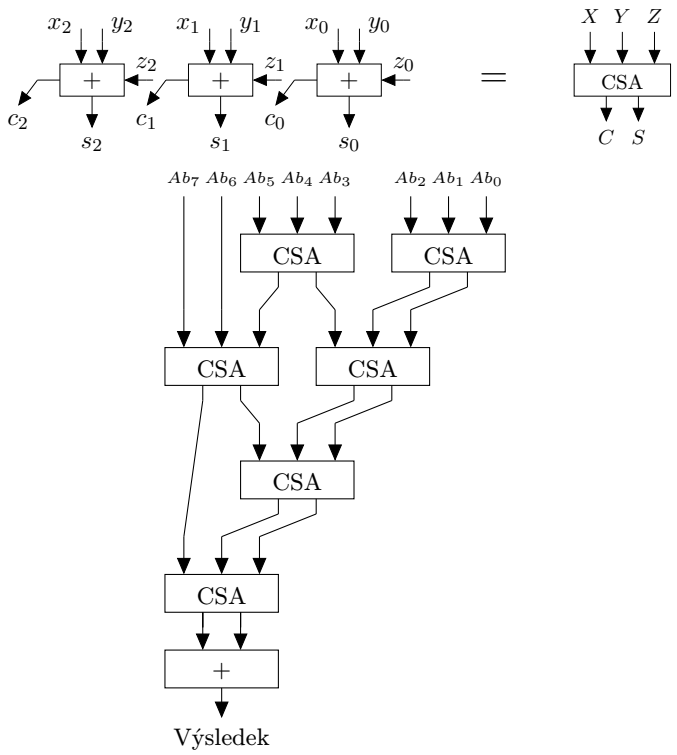


Figure 2.7: Scheme of multiplier using Wallace Tree. CSA means carry save adder.

Chapter 3

Approximate Circuits

Two main streams were formed to approximate circuits during the development. The first one is the approximation based on the electronic parameters modification (e.g. decreasing input power voltage or the working frequency). This method was shown in [7] in which the authors decreased the voltage in SRAM memories in order to decrease the power consumption. The second one is the function approximation. It is focused on the changes of the circuit specification in order to decrease the power consumption, the latency or the area. This work is dedicated to functional approximation.

In the dissertation thesis [6], functional approximation is defined as a transfer from fully specified logical function to partial. Let we have a Boolean function that have n inputs in the form $f : B^n \to B$ where $B = \{0, 1\}$. Boolean specification for n inputs is the function $g : B^n \to \{0, 1, -\}$ where $-$ means that the value of the function g is not specified for the given input combination. We say that g is completely specified if and only if that each output combination takes values 0 or 1. Specification is incompletely if at least one of them takes the value $-$. In the other words, input space $B^n = \{0, 1\}^n$ of the specified function g can be divided into three classes. The first one is the set of the input combinations for which is output 1. The second one is the set of the input combinations for which is output 0. The last one is the set of the input combinations for which is output $-$. By these classes, we can create their characteristic functions g_{on}, g_{off} and g_{dc}. Function $g_{dc} : B^n \to \{0, 1\}$ has the output 1 if and only if g has output with value $-$. Functions g_{on} and g_{off} are defined similarly by the first class or by the second class. Functional approximation is a transfer γ from the complete specification g to partial specification $\overline{g}$. After the applying approximation, we find its quality. The quality can be defined as a the number of the cases for which outputs of $\overline{g}$ differs with the outputs of g. In the addition, we try to reduce the amount of Boolean operators in the function. Thus, the function $\overline{f}$ received from the specification $\overline{g}$ should have less operators than in the function f. At the end from the approximated function $\overline{f}$, we can implement the approximate circuit by the given conventional circuit design technique.

In the section 3.1, error measurements are described metrics without which we could not have done this work. In the next section, the conventional proposes of the approximate circuit design are listed.

3.1 Error Metrics for Approximate Circuits

Error metrics are the most important things in case of approximate circuits. They compare the quality of the approximate solution *approx* to the fully functional solution *reference*. Their i-th input vector of the combinations is denoted as $approx_i$ respective $reference_i$. The number of primary inputs of the circuit is defined as n_i and the number of primary outputs is denoted as n_o.

The first basis metric is the number of input combinations for which the circuit works incorrectly.

$$err_{sum} = \sum_{i=1}^{2^{n_i}} \begin{cases} 1 & reference_i \neq approx_i \\ 0 & reference_i = approx_i \end{cases} \tag{3.1}$$

The similar and more accurate metric is the number of wrongly set bits err_{bits}. The j-th bit of the vector $approx_i$ or $reference_i$ is identfied as $approx_{i,j}$ or $reference_{i,j}$. This metric is suitable for the design of not-arithmetic approximate circuits.

$$err_{bits} = err_{shd} = \sum_{i=1}^{2^{n_i}} \sum_{j=1}^{n_o} reference_{i,j} \oplus approx_{i,j}. \tag{3.2}$$

For approximated arithmetical circuits, we can utilize more suitable metric which is based on the sum of absolute differences from the reference solution. It can be a sum of absolute error values of the approximate solution. It is defined by formula

$$err_{sad} = \sum_{i=1}^{2^{n_i}} |reference_i - approx_i|. \tag{3.3}$$

The average error is defined as the sum of the errors divided by the amount of input combinations. The main disadvantage is in its sensitiveness to large values. This problem might be solved by using median.

$$err_{avg} = \frac{err_{sad}}{2^{n_i}} = \frac{1}{2^{n_i}} \sum_{i=1}^{2^{n_i}} |reference_i - approx_i| \tag{3.4}$$

Standard deviation and variance determines how error values are distributed and deviated from the average value. The variance is the average of the squared distances from the mean value. The deviation is the mean squared variance and therefore, it is calculated according to the equation

$$err_{\sigma} = \sqrt{\frac{1}{2^{n_i}} \sum_{i=1}^{2^{n_i}} (reference_i - err_{avg})^2} \tag{3.5}$$

The next metric is maximal error of approximate solution defined by formula 3.6. It gives us the worst case of the working circuit. Similarly, we can set minimal error for which the circuit works incorrectly as seen on the formula 3.7.

$$err_{max} = \max_{\forall i \in \langle 1 \ldots 2^{n_i} \rangle} |reference_i - approx_i|. \tag{3.6}$$

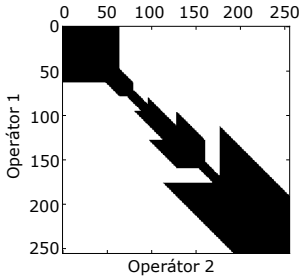


Figure 3.1: The example of error dispersion. The black colour means that there is an error for the input combinations. The white colour means the correct functionality for the given input combinations.

$$err_{min} = \min_{\forall i \in \langle 1 \ldots 2^{n_i} \rangle} \begin{cases} |reference_i - approx_i| & reference_i \neq approx_i \\ undef & reference_i = approx_i \end{cases} \tag{3.7}$$

The metrics above, for example err_{max} or err_{avg} can be transferred to their percentual values. It is a tradition that the error is expressed by these values in the case of conventionally designed approximate circuits. Percentages can be obtained in a such way that we divided the error with the maximal value of the arithmetic circuit:

$$err_{max\%} = \frac{err_{max}}{2^{n_o} - 1} \tag{3.8}$$

$$err_{avg\%} = \frac{err_{avg}}{2^{n_o} - 1} \tag{3.9}$$

$$err_{sum\%} = \frac{err_{sum}}{2^{n_o} - 1} \tag{3.10}$$

The relative error adds the information about the magnitude of the error. It can be obtained in the way that the targeted output is divided by the reference value.

$$err_{relative} = \frac{|reference_i - approx_i|}{reference_i}. \tag{3.11}$$

Note that the relative error may be bigger than 1.

The next metrics can be various statistical values, for example, finding the median or determining the quartiles. These values might be visualised by box plots. Another statistical value is so called modus which is the most frequent error.

The errors might be pictured into a chart in the case of larger approximate circuits. On the Figure 3.1, we can see the graph of the error dispersion which is taken from [14]. It is an illustration of the input combinations for which the circuit works incorrectly.

3.2 Conventional Designs for the Approximate Circuits

3.2.1 Assigning a Constant to the Output

One of the most basic approximation is to assign a constant to the output [15]. The main disadvantage might be its error rate. In some cases however, it still might be minimal error.

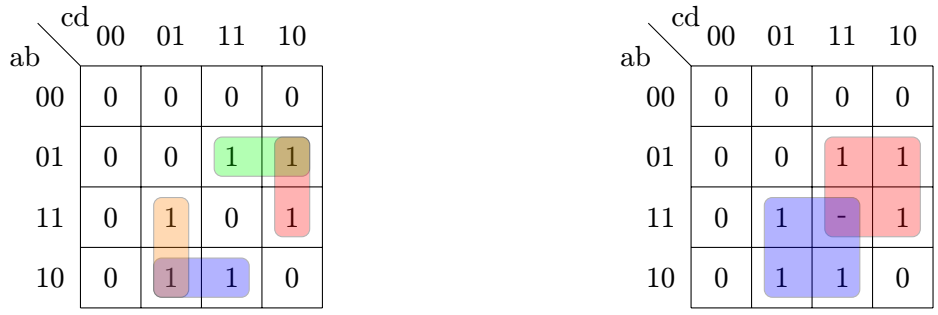


Table 3.1: Logical Function Approximation by Karnaugh Map. Complete specification is given on the left, the approximated specification is on the right. Colour differs each implicant.

For the best results, the constant is chosen as the most likely value. Lets have a data stream which gives us data in the Gaussian distribution. Then the most likely value is the mean one. The error is highly depended on the deviation of the distribution. Lets have a Gaussian distribution which changes its mean value time to time. We can improve this approximation method by the assigning the mean value at the regular intervals.

Lets have an arithmetical circuit performing operation $\circ$ which can be applied on two values with the same bitwidth w. Then the output constant is the mean value from the all input combinations. The formula can be defined as

$$avg = \frac{\sum_{i=1}^{2^w} \sum_{j=1}^{2^w} i \circ j}{2^{w+1}}, \tag{3.12}$$

in the denominator is the amount of all input combinations and in the numerator is the sum of the operation results for all of the input combinations.

Similarly, we can assign median or modus. It is obvious that in the case of addition the median is same to its average value. However in the case of multiplication, the average value is $avg = 12$ and the median values is $median = 8$.

The different way is to copy the input value to its output. This is suitable for the addition of large numbers with the small numbers. For example, we can just copy the 32-bit number when we add the 32-bit number with 2-bit one [15].

3.2.2 Ignoring the Least Significant Bits

This technique is utilized in the floating point arithmetic. In the case of integers, we can ignore the least significant bits and use the adders or the multipliers connected to the most significant bits [15].

3.2.3 Karnaugh Maps for the Approximation

Karnaugh maps are used for the minimization of logical functions. This technique transfers the truth table to two dimensional space which enables to better identify the Boolean neighbouring. The neighbour cells differs just in the one bit and the cells can be rotated.

The map is not a table. It is a torus. The coordinates are represented by Gray code, not the binary.

Minimization is performed in the such way that we create the Karnaugh map by the truth table. Then on the map, we search areas from which are used for the AND expression. Their sum defines the logical function. We select the areas to be the largest and to be as less as possible. The areas should satisfy the condition to contain all the ones and not to contain any zero (or vice versa). In the approximation of the circuit, we can break this condition, if the solution leads to the function with less operations [18]. An example of the approximation is pictured on the table 3.1. The result is a logical function in conjunctive (or disjunctive) normal form.

3.2.4 Arithmetical Approximate Circuits Blocks

The method is based on the building blocks such as full binary adders, see previous chapter. We can use approximate version of the carry as defined in

$$c_{approx} = (x \oplus y)c + xy, \tag{3.13}$$

we obtain an adder with smaller area because the adder has less functions and the expression $x \oplus y$ is already calculated and needed for the sum result.

The error happens only once in sixteen cases $err_{rate} = \frac{1}{16}$ and maximal error is $err_{max} = 1$. These approximate adders may be connected as seen on figure 2.2. With the connection of the blocks, the error propagates and its size depends on the number of approximate and fully functional adders in the solution. This method was described in the article [26]. The main advantage of this adder is that it contains one AND gate less and it doesn't need three input OR but the only two input. The delay of approximate circuit is the same as the delay of fully functional adder.

In [11], the multiplier design was presented. It requires smaller multipliers and adders. Firstly the approximate multiplier is created by the Karnaughovy map as shown on the table 3.2. Standard multiplication of numbers $3 * 3$ has four bits on the output (and it has value 1001); in the approximate multiplier we can assign only three bits (with value 111). The error occurs in one case out of sixteen ($err_{rate} = \frac{1}{16}$) and its magnitude is $err_{max} = 2$. Approximate circuit has less gates as shown on the figure 3.2. The area decreased to 30% as well as its critical path. The main advantage of this method is the ability to recover from the error. We know that the error happens when we multiply $3 * 3$ and it has size of 2. Thus, we can create another circuit, which adds number 2 to the result of the approximate multiplier when both operands are set to number 3.

The result multiplier is utilized as a building block for larger multipliers. On figure 3.2, the structure of larger multiplier is shown. It is consisted of four approximate multipliers and three adders. The error rate err_{rate} grows with the size of the multiplier. However the average error $err_{avg\%}$ grows slightly. This phenomena is given in the table 3.3.

ab \ cd	00	01	11	10
00	000	000	000	000
01	000	001	011	010
11	000	011	111	110
10	000	010	110	100

Table 3.2: Modified Karnaugh Map for Approximate 2-bit Multiplier. The modified value is highlighted by pink colour. Taken from [11].

Bitwidth	err_{rate}	$err_{avg\%}$	$err_{max\%}$
2	0.0625	1.39%	22.22%
4	0.19	2.60%	22.22%
8	0.46	3.25%	22.22%
12	0.675	3.31%	22.22%
16	0.81	3.32%	22.22%

Table 3.3: The Errors of the Approximate Multipliers. Taken from [11].

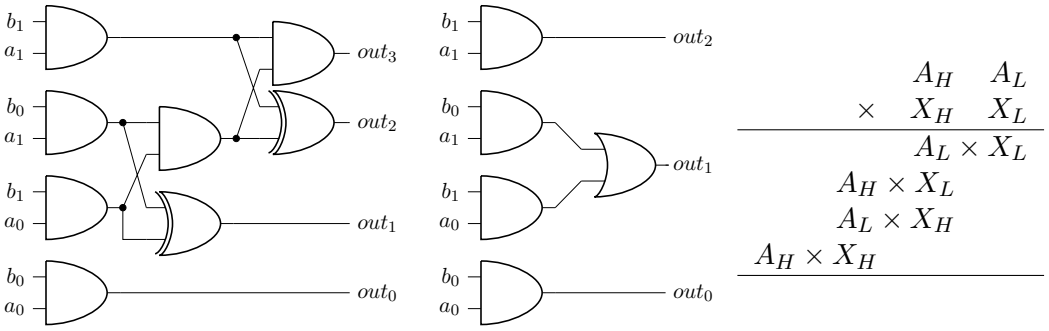


Figure 3.2: The Fully Functional (on left) and Approximate Multiplier (on the middle). On the right, the building larger multipliers is shown. Převzato z [11].

3.2.5 SALSA

One of the algorithm for automatic design of approximate circuit is SALSA (Systematic Logic Synthesis of Approximation Circuits). It is a conventional iterative method which creates the approximate circuit from the given fully functional circuit and the given error. Algorithm works as follows:

- For each primary input of the circuit, the set of gates is selected in a such way that the quality function is inactive for those gates.
- By those gates, the circuit is approximated.
- The approximation continues to next primary input.

SALSA contains several acceleration and heuristic techniques, it searches for input-output dependences [23]. Nevertheless, the whole implementation is not open source and we do not know what kind of quality function authors used.

3.2.6 ABACUS

ABACUS (A Technique for Automated Behavioral Synthesis of Approximate Computing Circuits) is an automatic method for the synthesis of approximate circuits on the behavioural level of design [15].

It is an algorithm which utilizes greedy search. Each combinational approximate circuit can be represented as an abstract syntax tree, AST. In the each iteration of the algorithm, the greedy search transforms the AST by several transform operations such as simplification of data types, moving the operation on the MSB, or assigning a constant to the output. Furthermore, it may transfer fully functional arithmetic circuits to simpler version e.g. addition might be performed with logical operation OR or the multiplication might be done with the shifting. On the AST level, this algorithm transforms the subtrees. For example expression $y_i * x_i + y_j * x_j$ replaces with $(y_i + y_j) * x_i$ in order to reduce the number of operation.

By this stochastic algorithm, the set of approximate circuits is created. For each circuit, the fitness value is assigned. The fitness function is

$$fitness = w_1 * accuracy + w_2 * power + w_3 * area, \tag{3.14}$$

where is the condition that $w_1 > w_2 > w_3$. The *accuracy* of the result circuit is simulated on training sets. The algorithmic search ends if the circuit with larger error ratio was found or if the number of iterations was exhausted.

Chapter 4

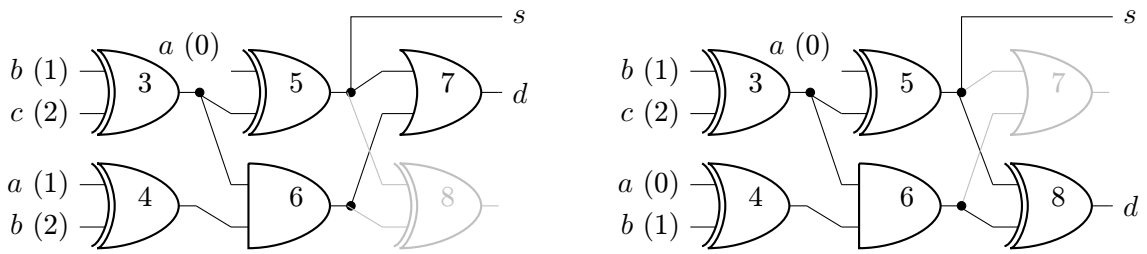
Evolutionary Approximation by Cartesian Genetic Programming

Cartesian genetic programming, CGP, was proposed for the circuit design by Julian Miller in the article [13] in 1999. This algorithm fits into evolutionary algorithms which are based on Darwin's theory of evolution and various Neodarwinism theories. The main driving force of the evolution is natural selection. The individuals above the average quality have larger probability to have an offspring. Therefore, their genetic information occurs in the next generations more often than the information of lower quality individuals. By the time, we obtain an individual with better abilities than the individuals on the beginning of the evolution. Evolutionary algorithms work similarly.

In CGP, the individual is represented by acyclic oriented graph. It is possible to use regular oriented graph for sequent circuit design. The nodes are arranged into the matrix with n_r rows and n_c columns. Each node represents an operation from the finite set of operations Γ. In some implementations of CGP, it is necessary to set maximal arity n_a of the operation f such that $f \in \Gamma$. This arity gives the amount of inputs of the node. In the case of circuit design, the nodes are posed as gates with binary arity. However, it is possible to use gates with bigger arity [17]. Furthermore, we have to define l-back parameter l which gives us the interconnection rate between the columns of the graph. For $l = 1$, the interconnection is minimal because the nodes are connected just between the neighbouring columns. For $l = n_c$, the interconnection is maximal because the column might be connected to any previous column. Moreover, we have the set the amount of primary inputs n_i and primary outputs n_o of the graph. These values represents the inputs and outputs of the resulting circuit.

The connection of the circuit is coded into chromosome of constant length. Each node is coded by tuple $(i_1, i_2, \ldots, i_{n_a}, \alpha)$ where α is a code of the function from Γ. Values i_1, i_2 to i_{n_a} are indexes to the node from previous column or it is the index to primary input of the circuit. Furthermore, Chromosome contains n_o genes (values) which denotes the connection of primary outputs. These values are nodes of the graph or primary inputs. An example of this coding is shown on figure 4.1. We can also see that coding is redundant and some of primary inputs may not be used in the final solution.

Evolution then finds the interconnection of the graph in order to satisfy the given specification. On th beginning, the initial population is created either randomly or using some heuristic, e.g. fully functional circuit. In each generation, the parent is selected which is the individual with the best fitness value. The parent is cloned and the clones are mutated.



`[[1,2,3][0,1,3]`
`[0,3,3][3,4,1]`
`[5,6,2][6,1,3][5,7]]`

a	b	c	s	d
0	0	0	0	0
0	0	1	1	1
0	1	0	1	0
0	1	1	0	1
1	0	0	1	1
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Fitness = 12

`[[1,2,3][0,1,3]`
`[0,3,3][3,4,1]`
`[5,6,2][6,1,3][5,8]]`

a	b	c	s	d
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Fitness = 16

Figure 4.1: Coding in CGP, chromosome and example of the mutation.

Mutants with a parent are the new population. This variant of the generating population is called evolution strategy $(1 + \lambda)$ where λ means the amount of offspring and 1 represents the parent. The individual with the highest fitness is selected as a parent. However, if we have two individuals with the same and the highest fitness value then we select the one who was not the parent in the previous generation. Evolution is ended by finding the individual with sufficient quality or by the exhaustion of the numbers of generations n_e.

The mutation provides us the variation of the nodes interconnection or their operations. Parameter h gives us the number of genes of the chromosome which are going to be mutated. The mutations is neutral if the change has no influence on the fitness value of the individual. This happens in two situations. Either we apply mutation on the inactive node which has no influence on the candidate solution (so called phenotype), or the phenotype changes and it has the same fitness as parent has. The opposite of neutral mutations are adaptive mutations.

Cartesian genetic programming was successfully utilized in the design of combinational circuits, image filters, classifiers, or even in the design of simply programs [17].

4.1 Multiobjective Optimization

The design of approximate circuits may be formulated as multiobjective optimization problem in which exists several exclusive criteria such as area, latency, power consumption and error metrics. In multiobjective optimisation, there is not the best solution which would cover all the criteria. Lets imagine a human who designs circuits. Will he or she design

them based on area, latency or the error? The situations, in which human has to decide by a single parameter, happens rarely.

Nevertheless, we can create a Pareto-front which takes into account all the criteria. We say that the solution x Pareto-dominate the solution y if $\forall i \in N : f_i(x) \leq f_i(y) \wedge \exists j \in N : f_i(x) < f_i(y)$ where f_i is the criterion and $N = \{1 \dots m\}$ is the set of criteria $|N| = m$ and the goal is to minimize f_i. Let P is a set of the solutions, then $x \in P$ is Pareto optimal if and only if there is no other solution $y \in P$ such that y Pareto dominates x and also solution x is in the Pareto front Φ ($x \in \Phi$).

4.2 Fitness Functions for the Approximate Computing

In the fitness function design, we need a complete specification for fully functional circuits. For each input vector, 2^{n_i} in total, is assigned an input vector of size n_o. For each input combination is defined the output value. Therefore, we need to specify $n_o * 2^{n_i}$ input combinations (in bits).

Fitness value of the candidate circuit C_{obt} basically corresponds to the number of incorrect responses of the reference circuit C_{ref}. We calculate the Humming distance between each output vector obt_i of the candidate solution and the reference output vector ref_i. The humming distance of two binary strings of the same size is the amount of bits in which these strings differ. The fitness value *fit* is summed up Hamming distances for all input combination. This fitness function may be implemented by the formula

$$fit = \sum_{i=1}^{2^{n_i}} \sum_{j=1}^{n_o} ref_{i,j} \oplus obt_{i,j}, \tag{4.1}$$

which is nearly equivalent to Formula 3.2.

We have many ways how to implement fitness function for the approximate circuits. In this section, fitness functions for the approximate circuit design are described.

4.2.1 Sum of Humming Distances

One of the basic method is to to create approximate circuits by the progressive reduction of the search space. The graph of CGP is defined as follows: $n_c \in \{n_{g\_AC} | 0 < n_{g\_AC} < n_{g\_PC}\}$, $n_r = 1, l = n_c$, where n_{g_PC} is amount of the gates of fully specified circuit [16]. This method fails if we want to use more number of the rows n_r.

Therefore, more sophisticated method is to restrict the fitness function to the maximal number of the used gates of the candidate approximate circuit n_{g_AC} (i.e. required number of the gates) such that $0 < n_{g\_AC} < n_{g\_PC}$. The fitness function is calculated as

$$fit_{SHD} = \begin{cases} \sum_{i=1}^{2^{n_i}} \sum_{j=1}^{n_o} ref_{i,j} \oplus obt_{i,j} & n_g \leq n_{g\_AC} \\ \infty & \text{else} \end{cases}. \tag{4.2}$$

If candidate solution is consisted with n_g such that $n_g > n_{g_{AC}}$, then we set its fitness to the worst in order to avoid its selection for parent.

4.2.2 Weighted Sum of Humming Distances

The previous method is not suitable for the approximate circuits realizing arithmetical operations. It may find a circuit which gives us the correct solution in the most of cases,

however in the error, its functionality may be below the average. In the other words, the circuit may have less incorrectly specified bits err_{bits} (formula 3.2), however its average absolute error err_{sad} may be larger (formula 3.3). The previous method (SHD) would select the circuits with larger absolute error err_{sad}.

We introduce so called priority of the outputs. Its principle is that the weight is set for each output bit. Thus, it means that the largest priority has the MSB. It leads to a greater probability that the error will happen on LSBs more often than on MSBs. For the weighting of SHD, we apply the formula

$$fit_{weighted\ SHD} = \begin{cases} \sum_{i=1}^{2^{n_i}} \sum_{j=1}^{n_o} w_j * (ref_{i,j} \oplus obt_{i,j}) & n_g \leq n_{g\_AC} \\ \infty & \text{else} \end{cases}, \tag{4.3}$$

where each output j is multiplied by its weight w_j. In the implementation, it is better to use bit-shifting than the multiplication because of it is executed faster.

4.2.3 Sum of Absolute Differences

For the design of approximate circuits, it might be better to use fitness function used for the symbolic regression. The problem of the symbolic regression is stated as a fitting mathematical equation to reference data with the given accuracy. In the case of circuit design, the fitness value is the sum of the error of the vector point of view. The value can be obtained by the formula

$$fit_{SAD} = \begin{cases} \sum_{i=1}^{2^{n_i}} |ref_i - obt_i| & n_g \leq n_{g\_AC} \\ \infty & \text{jinak} \end{cases}, \tag{4.4}$$

in which reference vector ref_i and candidate vector obt_i are their numerical values The main principle was described in the article [16].

4.3 Evolutionary Approximation of Complex Arithmetical Circuits

This section describes the different methods of utilizing CGP for the arithmetic circuit design which are suitable for approximate computing. The main problem of CGP is in the scalability of its use. The time needed for the evaluation of the candidate solution grows with added inputs and the number of generations for the finding the suboptimal solution grows too.

A few publications solves this problem. In the article [20], CGP is initiated by fully functional multiplier, which is approximated by the different fitness functions, see subsection 4.3.1. The article [21] describes the progressive evolution. The gates are decremented each evolution run in order to obtain the Pareto front. The method is described in subsection 4.3.2.

4.3.1 Evolutionary Approximation by Different Error Metrics

CGP is initiated by fully functional arithmetic circuit, e.g. multiplier, see 2.2.3. Designer select the error size K which would be the maximal error of the targeted circuit. The

CGP run will be separated into two phases. In the first phase, we try to approximate the adder on the error, so the fitness value f_1 is near to value K. The fitness function of f_1 is implemented by the error metrics which can be seen in 3.1. For example, the suitable metric is maximal error err_{max}, average error err_{avg} or the relative error $err_{relative}$. The second phase starts, when we obtain the circuit with the required error K or the value which is near to the required error, $K \pm \varepsilon$. In this part, the optimization by second criterion begins. It might be the amount of used gates, the delay, or even a different error metric. If we include fitness function of f_1 into the calculation of f_2, then the first optimization criterion will not get worse during the evolution. There are several ways how to assign fitness functions for f_1 and f_2.

In the original article [20], authors created three different scenarios. In the first scenario, they approximated fully specified multiplier to value K, which was the maximal error under the restriction to have the smallest average error. In the other words, fitness value f_1 was calculated as err_{max} by formula 3.6 and fitness value f_2 was calculated as err_{avg} by formula 3.4. In the second scenario, they created approximate multipliers with the smallest area. The fitness value of f_1 was evaluated as maximal error err_{max} and the second fitness f_2 gave us the number of used gates. Third scenario was similar to the previous one. The fitness value of f_1 was evaluated as average error err_{avg} and the second fitness f_2 gave us the number of used gates.

It was shown that for 8-bit multiplier with the maximal error of $err_{max\%} = 10$, they reduced the power supply by 96% [20].

4.3.2 Evolutionary Circuit Approximation by Heuristic Initialization

The article [21] proposes the method of heuristic seeding for the design of approximate circuits. Utilized algorithm CGP is bit different to the algorithm described in this chpater. Primary outputs of CGP may be connected to logical constants 0 and 1. This small change might be radical in case of the approximate circuits design.

Heuristic initialization is done as follows. Lets have the complete specified circuit C_{ref} containing n_g gates and we want to design an approximate circuit consisting of $n_g - 1$ gates. We create $2n_g$ circuits in a such way, that each gate of C_{ref} is replaced by a wire which connects the node output with the first (or second) input. The set of circuits having one less gate is created. Then, the circuits are evaluated by fitness function e.g. err_{sad}, see 3.3. The CGP run is then initiated by the circuit with the smallest error.

For the circuits containing larger amount of gates, this heuristic initiation might be slow. We can replace k gates instead of the one. These k gates are randomly removed and we create N of those circuits.

In the original article, the authors propose two scenarios. The first one is on the principle of incremental evolution. From the completely specified circuit, we create the approximate circuit by the heuristic. This circuit has $n_g - 1$ gates and we optimise it by CGP. The optimized circuit is then used for the creation of new circuit consisting $n_g - 2$ gates. Gradually, we obtain Pareto front of the approximate circuits with the best functionality for the given amount of gates. In the second scenario, authors initiate CGP by the best circuits of different size i.e. $n_g - 1, n_g - 2, \ldots, 2, 1$. The approximation of the circuit consisting $n_g - 2$ gates doesn't matter on the circuit consisting $n_g - 1$ gates, but it depends on the reference circuit C_{ref}.

In the result, authors obtained an approximate solution of 25 bit median containing 221 components. Random initialization fails in the complex problems like that.

Chapter 5

Edge Detection

Edge detection in images is one of the most common operation in computer vision. It is utilized for the analysis, image segmentation, scene reconstruction, the objects tracking or even in biological and medical programs. The goal of the edge detection is to find the neighbouring pixels in the image which have significantly different brightness. The difference has constant gradient direction in its surrounding, see Figure 5.1. The gradient direction can be defined as the first derivation that is the largest in the direction perpendicular to the edge [27].

Nowadays, we have a lot of edge detectors. We can use algorithms based on the knowledge of mathematical analysis, neural networks, fuzzy logic or even evolutionary algorithms [17]. This chapter describes only two common edge detectors in which we can use approximate components. The section 5.1 presents the edge detection by Sobel operator. The next section 5.2 describes Canny edge detector.

5.1 First Derivation Methods

The computation of gradient in a discrete picture can be performed by convolution kernels (operators). The most common kernels are Sobel operators, Roberts operators, and Prewitt operators, see tables 5.1 and 5.2. Each operator differs in the convolution mask values giving the weight of point for gradient calculation. Furthermore, they differs in the mask size, for example Roberts operator uses 2x2 sized mask, however, Sobel and Prewitt operators have

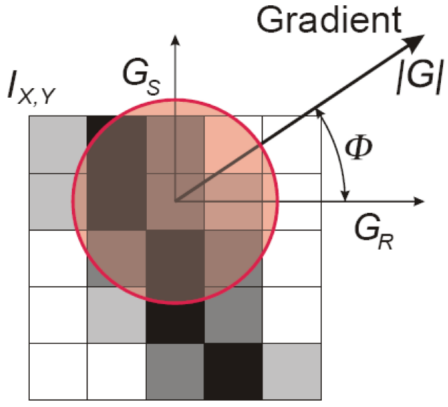


Figure 5.1: Graphical representation of the edges and their gradient[27].

1	1	1	1	1	0	1	0	-1	0	-1	-1
0	0	0	1	0	-1	1	0	-1	1	0	-1
-1	-1	-1	0	-1	-1	1	0	-1	1	1	0

Table 5.1: Prewitt operators, from left for angle 0°, 45°, 90° a 135°.

1	2	1	2	1	0	1	0	-1	0	-1	-2
0	0	0	1	0	-1	2	0	-2	1	0	-1
-1	-2	-1	0	-1	-2	1	0	-1	2	1	0

Table 5.2: Sobel operators, from left for angle 0°, 45°, 90° a 135°.

size 3x3.

The size of gradient $|G|$, so called the edge magnitude, can be obtained by a convolution of two operators which are perpendicular to each other, for example in vertical direction G_x and for horizontal direction (G_y). The result value of the gradient is a distance of the vector of these two values, see formula 5.1. In that case, the gradient is calculated in Euclidean space and it might be time and resource consuming. For hardware implementation, it is better to use simplified version based on Manhattan metric, see 5.2.

$$|G| = \sqrt{G_x^2 + G_y^2} \tag{5.1}$$

$$|G| = |G_x| + |G_y| \tag{5.2}$$

The edge detection consists two steps:

1. Selecting the areas of the edges, i.e. areas with high change of the brightness intensity.
2. The edge detection by thresholding

An example of the edge detector can be seen on the Figure 5.2. The gradient is calculated by Sobel operator for angles 0° and 90°. We can see that the detector do not make the edges thinner. Moreover, the edges are discontinuous, this can be seen on bottom left on the picture [27].

Figure 5.2: On left, the reference image. In the middle, the image after the application of Sobel operators. On right, the result image after thresholding where $T = 42$.

5.2 Canny Edge Detector

The Canny edge detector was proposed by John Canny in year 1986. He described in his work, the detector has to satisfy three parameters; the quality, accuracy and uniqueness. The quality of the detector means that all of the edges must be found and the detector does not detect imaginary edges. The accuracy means that the edges have to be as close as possible - the detector has to prevent the discontinuity of the edges. The detected edge is unique if and only if it is selected only once.

John Canny designed an algorithm which satisfies these requirements. For two dimensional imagine, the algorithm is implemented by four steps:

1. Noise elimination
2. The calculation of gradient magnitude
3. The calculation of gradient direction and edges thinning
4. Double thresholding with hysteresis

Output of each step is shown on figure 5.3.

The first step is a fundamental step because the image noise reduction affects the result of the next part of algorithm. We remove the noise by convolution with Gauss filter having kernel of size 5x5. The kernel values are calculated by formula

$$Gauss(x, y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}}. \tag{5.3}$$

Convolution matrix may have larger size for more intensive blurring of the source image. In this work, I used the most common version of the Gauss kernel (thus the kernel of size 5x5). Furthermore, we will use integer implementation which is more suitable for hardware. We change the first part of Gaussian function (the part before e) which normalises the curve to have its integral equal to one. The Gauss function, we can multiply by an integer and we obtain a kernel which is suitable for the convolution on the integer level. Furthermore, we can do convolution by shifting bits.

The second step calculates the magnitude of the edge. In the most cases, this is done by Sobel operators, see the previous section.

The third step thins the found edges by their local extremes. For each magnitude, we need to select the gradient direction. We have four main directions, each of them has an angle 0°, 45°, 90° or 135°. The direction is calculated by formula

$$\Theta = \tan^{-1}(G_x/G_y). \tag{5.4}$$

Figure 5.3: The steps of Canny Edge Detector. From the left side, reference image, blurred image, magnitude, edge thinning and image after hysteresis.

The edge is than thinned by using the normal of the edge direction and then neighbouring magnitudes are compared.

The last step is double thresholding and hysteresis. The pixels having greater magnitude than higher threshold T_H are automatically selected as edges. The pixels having greater magnitude than lower threshold T_L are selected edges if they are neighbouring with another pixel which is selected as edge. In the other words, this step removes weak edges which are not connected to strong edges.

Chapter 6

Acceleration of Evaluation of Approximated Candidate Solution

Cartesian Genetic Programming is time consuming algorithm, thus this chapter deals with its acceleration, especially of the candidate solution evaluation. By utilizing profile tool `GNU gprof`, we can find out in which function the algorithm spends the most of it time. For this purpose, CGP parameters were set to $n_r = 1$, $n_c = 80$, $l = 40$, $\Gamma = \{$BUF, NOT, AND, OR, XOR, NAND, NOR, XNOR$\}$, $\lambda = 4$, $h = 1$, $n_e = 5 \cdot 10^6$ for the desing of three bit, four bit, five bit and six bit adders. Population was initiated randomly. Evaluation of the circuit `cgp_eval()` was using 64-bit simulation which evaluated unused gates. The fitness values `calc_fitness()` were calculated by SHD method using function `zeroscount()` implemented by lookup table. The determining of used gates, `used_nodes()`, was accomplished after the evaluation after finding the exact solution.

In the results of experiment, table 6.1, we can see taht the most of time algorithm spends in the simulation of candidate solution `cgp_eval()`. In one simulation, we need to evaluate 2^{n_i} input vectors for all of the gates. Therefore, the time of simulation grows exponentially with the growth of the primary inputs. This feature is proved by the values in the table.

6.1 Parallel Simulation

Partial solution is to utilize parallel simulation. We will use the fact that we can save 64 bits into the integer. Then we can obtain the result of 64 combinations in only one simulation. Thus we speed up the algorithm 64 times [17]. Moreover, we can utilize SIMD

Adder	3-bit 6 inputs		4-bit 8 inputs		5-bit 10 inputs		6-bit 12 inputs	
`cgp_eval()`	74.10 %	38.7 s	87.63 %	152.0 s	91.01 %	620.2 s	92.24 %	2440.1 s
`zeroscount()`	3.55 %	1.8 s	5.07 %	8.8 s	6.01 %	41.1 s	5.93 %	157.0 s
`calc_fitness()`	1.24 %	0.6 s	1.54 %	2.6 s	1.52 %	10.3 s	1.73 %	45.6 s
`used_nodes()`	18.30 %	9.5 s	4.79 %	8.3 s	1.15 %	7.8 s	0.00 %	0.0 s
`cgp_mutate()`	2.13 %	1.1 s	0.78 %	1.3 s	0.29 %	1.9 s	0.12 %	3.2 s
`main()`	0.78 %	0.4 s	0.30 %	0.5 s	0.10 %	0.6 s	0.08 %	2.1 s

Table 6.1: The time spend in the functions in CGP algorithm for adders with different bitwidth using tool GNU `gprof`.

coprocessor which is the part of the most of processors. This unit has 128-bit registers and it has effective instruction *popcnt* which is used for the determining the number of bits in the register.

6.2 Phenotype Simulation and Skipping Neutral Mutations

The next method is to not to simulate whole graph of CGP. We will ignore the gates which are used in the final candidate solution (phenotype). The finding of unused nodes brings some time overhead, however in the case of approximation, we need to find the amount of gates of each candidate solution [22].

In the article [8], authors presented the way of acceleration in which some neutral mutations are skipped. If a inactive gene is mutated then the candidate solution has the same fitness as its parent. Note that the inactive genes are not used in the phenotype. Moreover in [8], they modified the mutation operators. For example, they mutated the candidate solution until the active gene is mutated.

6.3 JIT Compilation of Chromosome and Fitness Function

In pipelined instruction processing, jump instructions are problematic. During the interpretation of the circuit, jump instructions are used and they lead to the slowdown of algorithm. One of the acceleration is to pre-compile the simulation, i.e. Just-in-time (JIT) compilation of chromosome. The iteration of the CGP graph with jump instructions is done only once during the JIT compilation of candidate circuit. To achieve acceleration efficiency, the compiled chromosome has to be run many times because the compilation has some time overhead. The suitability of JIT depends on the size of CGP graph and the size of training vectors [22].

Nevertheless, the previous work [22] compiles only the graph. We can compile it with fitness function. This work extends online CGP generator [19] which generates chromosome compiler. I added fitness function compiler, however, it was needed to write it in the operation code (directly in the binary) in order to achieve the most effective solution. For these purposes, the reference instruction set of Intel x64 was used [12] with the reverse engineering of compiled parts of code. The reverse engineering was done by the programs `objdump`.

6.3.1 Experimental Evaluation of JIT compilations

Experiments were done on a graph $n_c = 80$, $n_r = 1$, $l = 40$ with the function set $\Gamma = \{$BUF, NOT, AND, OR, XOR, NAND, NOR, XNOR$\}$. Fitness function calculated SAD and its results were saved in the extra array. In the compiled version of fitness function, the processor stack was used (hypothesis - in order to have the values in the cache memory). The goal of CGP was to design 3-bit, 4-bit, 5-bit and 6-bit adders and 4-bit, 5-bit and 6-bit multipliers Byly navrhovány tří, čtyř, pěti a šesti bitové sčítačky a čtyř, pěti a šesti bitové násobičky for $n_e = 10^6$ generations. The CGP runs were seeded with the same circuit and same random seeds in order to achieve the most precisely results for comparison. From ten runs, the average value of the evaluations per second was selected as the result. The number of evaluations per second was obtained using Formula

$$eps = \frac{\lambda * n_e}{t}, \tag{6.1}$$

Task	TV	n_o	Interpretation	JIT compilation	JIT compilation with fitness function
3+3 adder	1	4	**588 077**	327 128	259 358
4+4 adder	4	5	132 500	**151 369**	139 292
5+5 adder	16	6	30 939	86 055	**103 971**
6+6 adder	64	7	7 847	34 638	**71 532**
4x4 multiplier	4	8	108 528	**120 736**	107 512
5x5 multiplier	16	10	27 799	69 156	**88 449**
6x6 multiplier	64	12	7 061	23 917	**59 988**

Table 6.2: Comparison of acceleration methods based on JIT compilations depending on the number of training vectors TV and the given task. The measured values are given in the number of evaluations per second by formula 6.1.

where t is the time spend in the evolution. The measurement was done on processor Intel Core i3-2350M with 2.3 GHz.

The experimental results are shown on the table 6.2. We can see that the JIT compilation of chromosome is worth if we have four 64 bit training vectors . Furthermore if we have sixteen training vectors, the JIT compilation with fitness is worth more than without fitness precompilation.

6.4 Vector Calculation of SAD

If we will use parallel simulation, then it would be good to implement an effective fitness function which is complex in the case of SAD method. The response of parallel simulation of the circuit in CGP is n_o 64-bit vectors, see 6.1. These vectors are arranged from the MSB to LSB. For the calculation of the fitness value fit_{SAD} see formula 4.4, we can perform the bit transposition. In other words, we transform the n_o 64-bit vectors to their 64 integer values. By those values, we can easily calculate the fitness value fit_{SAD}. This principle was used in the works [16], [20] and [21].

The most problematic part of the current approach is in the transposition, because it requires a lot of bit shifting, arithmetic and logical opeartions and even jump instructions which disallow the instructions pipelining. Furthermore, we need to allocate a memory for the output response. These all things lead to slow down of the algorithm.

Acceleration can be done by the interpretation of logical circuits on the vector level. For parallel calculation of SAD, we can use 64-bit registers and boolean functions in the way they were used in parallel simulation of candidate solution. The calculation is based on the interpretation of subtraction and absolute value by boolean functions.

The subtractor is realised as follows. Firstly during the initialization of CGP, we invert the reference output of the circuit specification C_r. Secondly during the evaluation, the inverted reference output C_r^- is added up to the output o of the candidate circuit C_o. The adding operation is implemented as an interpretation of the carry ripple adder on bit level. In the first iteration, the sum value of the output s_0 and carry value c_0 are calculated by the Boolean formulae 6.2 and 6.3 which are equal to half adder. Then we gradually obtain their j-th values by formulae 6.4 and 6.5 which realize full adder.

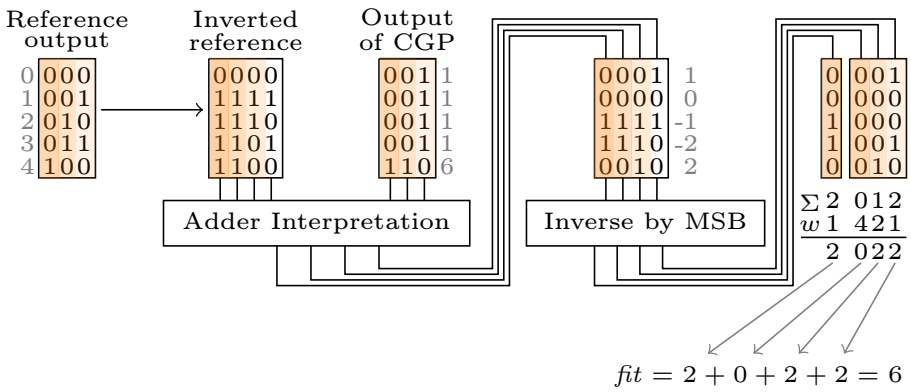


Figure 6.1: Paralel calculation of fitness value SAD on 5bit vectors, where Σ means *popcnt* and w is the weight of the vector. The grey coloured numbers represent the numerical values of transposed vector.

$$s_0 = C_r^-(0) \oplus C_o(0) \tag{6.2}$$
$$c_0 = C_r^-(0) \cdot C_o(0) \tag{6.3}$$
$$s_j = C_r^-(j) \oplus C_o(j) \oplus c_{j-1} \tag{6.4}$$
$$c_j = C_r^-(j) \cdot C_o(j) + (C_r^-(j) \oplus C_o(j)) \cdot c_{j-1} \tag{6.5}$$

The problematic is also absolute value calculation. After the subtraction, some values are positive and some are negative. On MSB s_{n_o} is saved the flag of negative value by which the rest vectors are inverted $s_j = s_j \oplus s_{n_o}, j \neq n_o$.

Now, we have to calculate the sum of those differences. Each vector represents each binary base w, e.g. 1, 2, 4, 8, 16 etc which is calculated by $w = 2^j$. In the each vector j, we calculate the amount of set bits by *popcnt* and then the value is multiplied by the binary base. At the end, we sum up those values and we have to add the MSB into the sum aswell without adding a weight. Ha ha, complicated, isnt it? If someone actually reads this text, contact me. Back to the text. In the other words, we multiply each output vector with their binary base. Then we sum them up and at the end we add the MSB values into the sum. The MSB values are needed to add because of the negative/positive arithmetic. Anyway the obtaining of absolute value can be written as formula 6.6.

The whole process of parallel simulation is shown on the picture 6.1.

$$fit = popcnt(s_{n_o}) + \sum_{j=0}^{n_o-1} 2^j * popcnt(s_{n_o} \oplus s_j) \tag{6.6}$$

6.4.1 Experimental Evaluation of Vector Calculation

We examine three different implementations of fitness functions. The first method (A1) calculates fitness value of SAD by the transposition. The second method (A2) utilizes vector calculations of SAD. The third method (A3) precompiled the vector calculation of SAD, i.e. it is JIT compilation of method A2. Acceleration techniques are compared by their *eps* values, which are defined by formula 6.1.

		Evaluations per seconds					
		With neutral mutations			Without neutral mutations		
Multiplier	$n_e * 10^6$	A1 [*eps*]	A2 [*eps*]	A3 [*eps*]	A1 [*eps*]	A2 [*eps*]	A3 [*eps*]
4x4	10	29 917	**230 149**	177 970	54 912	**277 923**	226 222
5x5	5	6 558	124 336	**142 373**	12 052	168 688	**186 781**
6x6	1	1 386	41 468	**85 696**	2 683	61 673	**129 541**
7x7	0,1	303	10 600	**38 142**	602	17 644	**53 930**
8x8	0,01	66	2 432	**11 325**	135	4 782	**23 190**

Table 6.3: The comparison of implemented fitness functions SAD. Method A1 computes it by transposition. Method A2 utilizes vector calculation. Method A3 precompiles the method A2.

For each method, algorithm used these acceleration: not evaluating unused nodes and precompiled 64-bit parallel simulation of the circuit. For each multiplier and each acceleration technique, ten independent runs were evaluated and their average *esp* value was taken as the result. The CGP parameters were chosen as follows: $\Gamma = \{$BUF, INV, AND, OR, XOR, NAND, NOR, XNOR$\}$, $n_r = 1$, $n_c = 40$, $l = 40$, $\lambda = 4$, $h = 2$. The experiment was performed on 4-bit, 5-bit, 6-bit, 7-bit and 8-bit multipliers with the specific number of generations n_e. The experiment was done on the processor Intel Core i3-2350M, 2.3 GHz.

The results of the experiment are shown on the table 6.3. We can see that parallel computation (A2) is more efficient than the original method (A1). For larger multipliers (5x5 and more), the compiled fitness function (A3) is more suitable. This fact was observed in the previous experiment. The design of 8-bit multiplier was speed up more than **170** times against to the original method (A1).

Moreover, we can see the the acceleration if we skip the neutral mutations. This method, however, is highly depended on the search space and the amount of mutated genes.

Chapter 7

Evolution of Approximate Arithmetic Circuits

The chapter deals with the design and evaluation of the arithmetical circuits. For the evolutionary design, the algorithm presented in the chapter 4.3.2 was used. The candidate solution simulation was accelerated by techniques described in the previous chapter.

7.1 Design of Inaccurate Adders and Their Evaluation

CGP was initiated by fully functional adder, which was constructed by Kogge-Stone algorithm which was described in the previous chapter 2.2.1. The settings of CGP was choose as follows: Γ = {BUF, INV, AND, OR, XOR, NAND, NOR, XNOR}, $n_r = 13$, $n_c = 7$ which is equal to the delay of fully functional Kogge-Stone adder, $l = 7$, $\lambda = 4$, $h = 4$, $n_e = 500\,000$. The number of fully functional adder is $n_g = 73$. This value was decremented after 50 runs of CGP. For the mapping of the adder consisting 73 gates into the graph with nodes 7×13, the ALAP (as late as possible) scheduling algorithm was used.

The evolutionary algorithm found the implementation of fully functional adder with the less gates ($n_g = 62$) which had less gates than the initiated solution. The properties of approximated adders are shown on the table 7.1. We can see that the relative delay of incorrect adders can be worse than fully functional solution, however the relative area has a falling trend.

7.2 Design of Inaccurate Multipliers and Their Evaluation

The initialization of CGP was done by fully functional multiplier constructed by Wallace Tree, which was described in the chapter 2.2.3. The amount of gates was $n_g = 330$ in the fully functional solution. This value was decremented after 50 CGP runs. The settings of CGP was chosen as follows: Γ = {BUF, INV, AND, OR, XOR, NAND, NOR, XNOR}, $n_r = 1$, $n_c = n_g + 3$, $l = n_c$, $\lambda = 4$, $h = 0.05 n_c$, $n_e = 500\ 000$. After the start of the algorithm, it was needed two weeks to obtain all the results. The properties of each multiplier can be compared in the table 7.2.

7.3 Dispersion Error Plot for Arithmetic Circuit

In the tables 7.1 and 7.2 are shown the statistical data about the error. It is better to picture the error for the user in a such way that he would know when the error happens and what is its size. Now, we will study the plot shown in chapter 3. The plot gives us the cases when the error happens. For arithmetical circuits however, we can add the information about the error size. For this propose, we introduce the third dimension defined by colour.

The dispersion error plot can be implemented as follows. We know that grey colour can be encoded into 256 bits. In the other words, we have 256 colours which can be ordered by their highlight. For all input combination from the inaccurate circuit, we obtain the error size. Then, we order the error size and we determine 256 quantiles which are equal to the highlight value. In the result, the bright colours determine small error and the dark errors mean greater error. For each input combination, we will select its quantile and it is pictured in the plot. The resulting plots are on the pictures 7.1 and 7.2 where we can see the error progression.

Amount of Gates	Relative area	Relative delay	Error MIN	Q1	Q2	Q3	MAX
1	24	1,00	1	16	33	55	128
6	320	8,85	1	8	17	28	64
12	624	10,31	1	5	10	17	64
18	848	9,49	1	4	9	17	64
24	1 120	9,80	1	3	7	13	64
30	1 312	15,07	1	3	7	15	64
36	1 648	12,16	1	3	7	15	64
42	1 024	8,74	1	4	10	23	64
43	2 416	12,05	1	1	1	2	3
48	2 520	13,69	1	1	1	1	1
54	2 784	13,63	1	1	1	1	1
60	2 888	14,62	1	1	1	1	1
62	3 272	14,05	0	0	0	0	0

Table 7.1: The Properties of Evolutionary Approximated Adders. The calculation of relative area and relative delay was based on VLSI values given on the table 2.2.

Amount of Gates	Relative area	Relative delay	Error MIN	Q1	Q2	Q3	MAX
1	40	1,76	1	2 437	4 781	7 373	16 600
30	1 464	18,60	1	467	1 078	1 874	5 888
60	3 136	36,83	1	237	495	838	3 714
90	4 712	41,43	1	135	285	487	5 257
120	6 040	34,99	1	84	178	305	924
150	7 672	42,80	1	60	127	214	739
180	9 232	44,89	1	36	76	129	2 390
210	10 872	51,15	1	21	41	70	244
240	12 712	48,81	1	10	21	35	106
270	14 048	54,11	1	6	11	17	48
300	15 672	52,15	1	2	3	7	18
319	16 760	51,03	2	2	2	2	2
330	16 752	56,66	0	0	0	0	0

Table 7.2: The Properties of Evolutionary Approximated Multipliers. The calculation of relative area and relative delay was based on VLSI values given on the table 2.2.

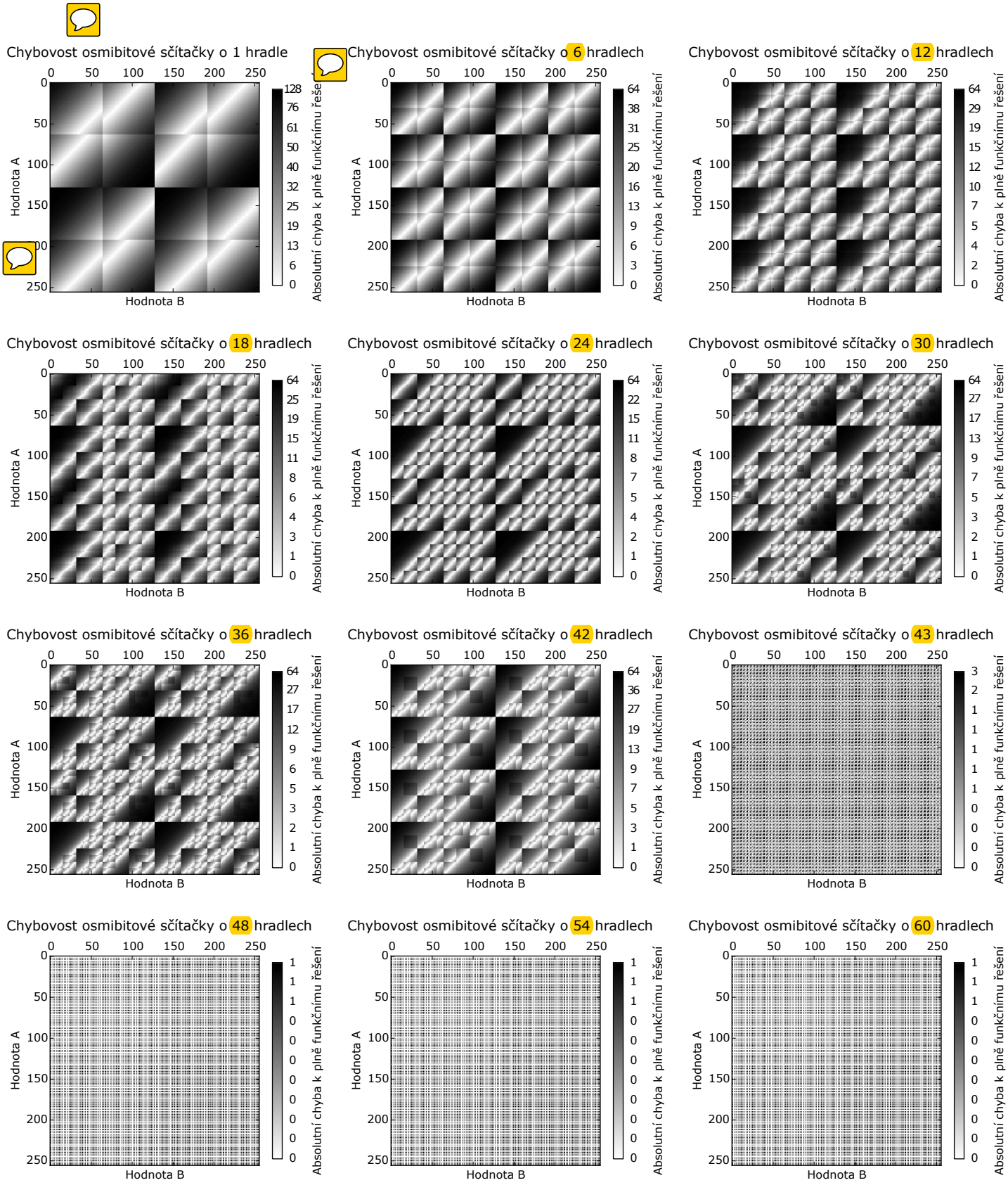


Figure 7.1: Error Dispersion Plots of Developed Approximate Adders

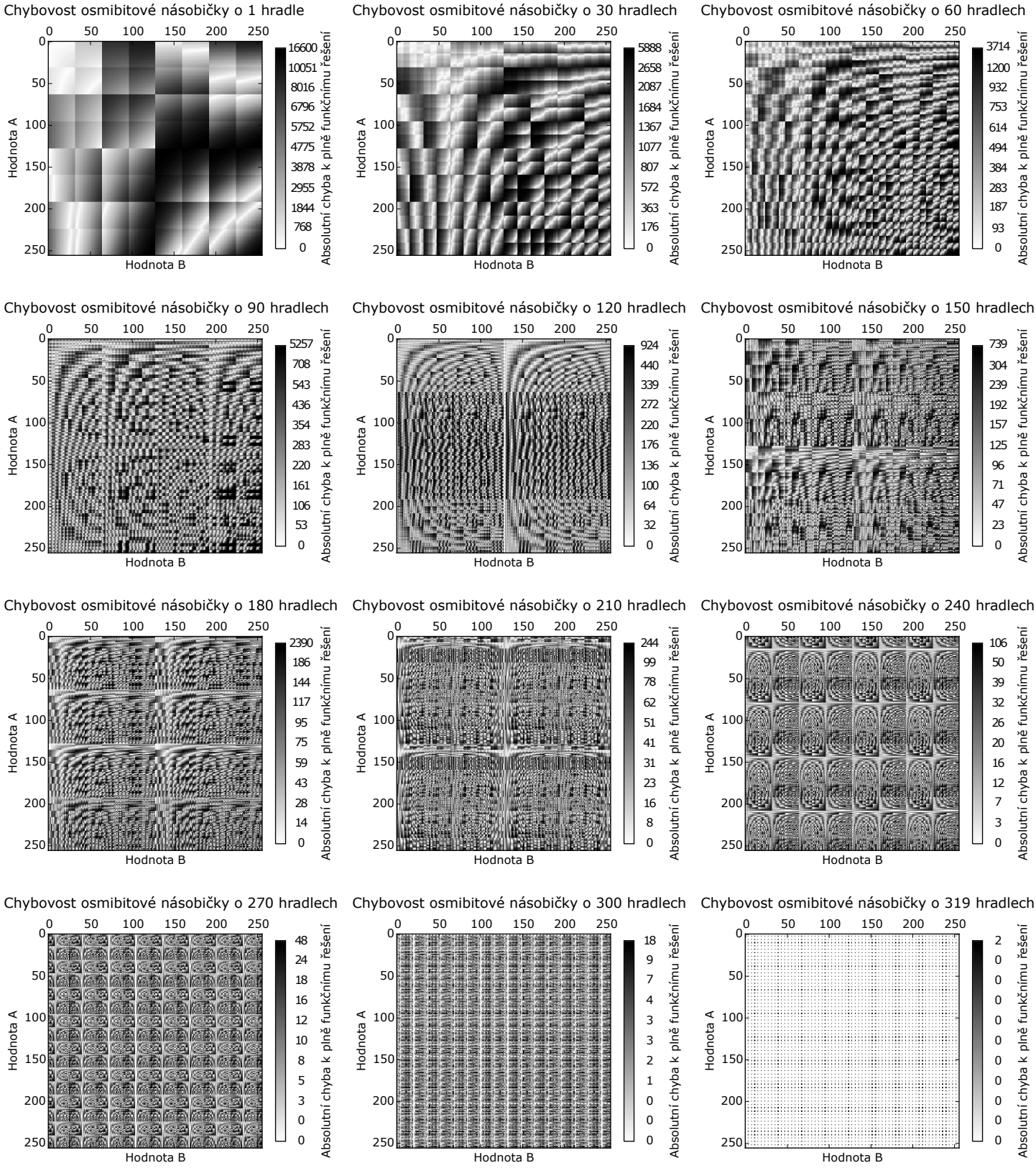


Figure 7.2: Error Dispersion Plots of Developed Approximate Multipliers

Chapter 8

Utilizing Approximate Adders in the Edge Detectors

This chapter describes utilization of designed inaccurate circuits in edge detectors. The first part shows us the utilization of approximate adders in Sobel edge detector. And the second part is about the approximation in Canny Edge detector.

The first part of the chapter deals with the hardware implementation. Then, we chose the elements which are going to be approximated. At the end, we evaluate the approximated detectors with the fully functional solution.

8.1 Inaccurate Sobel Edge Detector

In the chapter 5 was described that the process of Sobel edge detection can be split into two phases: highlighting areas of high intensity change and following thresholding. The first part gives us more information about the detected edges magnitude than the second part in which the information is lost due to thresholding. Therefore, we will focus only on the first part.

8.1.1 Hardware Implementation and Approximation

Convolution by Sobel kernels can be realized by 8-bit adders and subtractors. It is not worth to utilize multipliers because the mask uses uses values 0, 1, 2 and their negations. The multiplication realized by 0 constant is always 0 because zero is an absorbent number. The constant 1 is neutral for multiplication, thus we do not need to multiply it. The constant 2 multiplication can be realized by shifting bits. Their negative values can be realized by the subtractor.

For hardware, it is better to calculate magnitude by formula $|G| = |G_x| + |G_y|$. Absolute values $|G_x|$ and $|G_y|$ can be obtained by bits inversion by MSB and the summing of MSBs into the magnitude can be ignored. The resulting circuit is shown on the figure 8.1.

In the developed circuit, we approximate the fast Kogge-Stone adders which do the convolution. Subtraction with the gradient calculation will stay accurate.

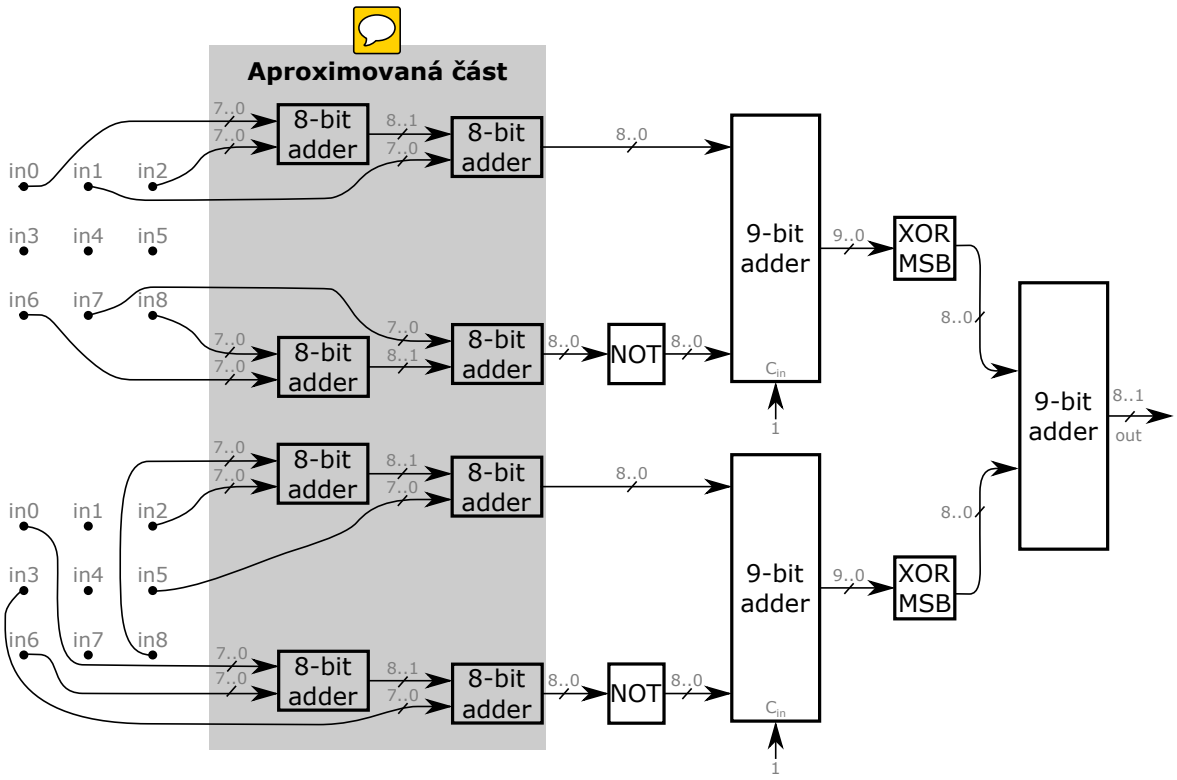


Figure 8.1: Sobel Edge Detector in Hardware Design

8.1.2 Experimental Evaluation

The results of approximated solution are pictured on the figure 8.2. We can see that we can approximate adders to 43 gates, where human eye doesn't see the error. In the adders using less gates, we observe the error easily, e.g. the incorrect edges are detected on Lenna's shoulder.

By the results of table 8.1, the fully functional implementation consists 897 gates and its relative area is 41 716 with relative latency 65,16. If we change the inaccurate adders with approximated version consisting 43 gates, we decrease the area to 34 868. Relative latency is reduced to 61,13. Therefore, we decreased the area by 16, 4% and increased it speed by 9% and the error of detection is insignificant.

Circuit	Number of use	Number of gates	Relative area	Relative delay
8-bit adder	8	62	3 272	14,05
9-bit NOT	9	18	432	1,00
9-bit adder (with Cin)	3	79	3 708	17,24
9-bit XOR	2	18	648	1,55
Sum Σ	-	897	41 716	*65,13

Table 8.1: Parameters of Hardware Implementation of Accurate Sobel Edge Detector.
* Critical path.

(a) 1 gate (b) 6 gates (c) 12 gates

(d) 18 gates (e) 24 gates (f) 30 gates

(g) 36 gates (h) 42 gates (i) 43 gates

(j) 48 gates (k) 54 gates (l) 62 gates (fully functional)

Figure 8.2: Results of Approximated Sobel Edge Detector

8.2 Inaccurate Canny Edge Detector

In the chapter 5 was described that Canny Edge detector is algorithm which works as follows. Firstly the noise is eliminated then the magnitudes are calculated. Then the detection of local extremes is done and the end double thresholding is done with hysteresis.

8.2.1 Propose of the Approximation

Firstly, we have to decided which part of algorithm will be approximated.

The article [9] proposes an hardware approximation of Gauss filter not using multiplication. The approximation of convolution by Sobel kernels was already shown. In hysteresis, we can not apply multipliers or adders. By exclusion method, the local extremes detection was left.

Operation tan^{-1} is hard to implement in hardware. The angles can be calculated as a vector from the vertical (G_V) and horizontal gradient (G_H). For these proposes, we find out the approximate angle of vertical and horizontal parts as shown 8.3.

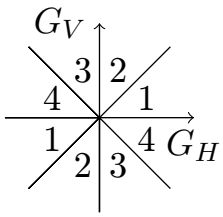


Figure 8.3: Approximate angles of edges used for the edge angle calculation.

Process of thinning is based on the obtaining the magnitudes of neighbouring pixel which are on the edge angle normal. If the neighbouring magnitudes are smaller than the magnitude of the pixel one, then the pixel is in the local maximum. The calculation of normal can be realized by weighting the neighbours. The first part of local extreme detection for the first direction is shown by formulae 8.1 and 8.2, where upper indexes are the pixel coordination in the image. If both equations are valid, then it is a local extreme. We can change the equations to formulae 8.3 and 8.4 that utilizes multiplications which can be approximated.

$$|G^{x-1,y-1}|\frac{G_V^{x,y}}{G_H^{x,y}} + |G^{x,y-1}|(1 - \frac{G_V^{x,y}}{G_H^{x,y}}) \leq |G^{x,y}| \tag{8.1}$$

$$|G^{x-1,y-1}|\frac{G_V^{x,y}}{G_H^{x,y}} + |G^{x,y+1}|(1 - \frac{G_V^{x,y}}{G_H^{x,y}}) \leq |G^{x,y}| \tag{8.2}$$

$$|G^{x-1,y-1}|G_V^{x,y} - |G^{x,y-1}|(G_V^{x,y} - G_H^{x,y}) \leq |G^{x,y}|G_H^{x,y} \tag{8.3}$$

$$|G^{x-1,y-1}|G_V^{x,y} - |G^{x,y+1}|(G_V^{x,y} - G_H^{x,y}) \leq |G^{x,y}|G_H^{x,y} \tag{8.4}$$

The hardest part is hardware realization of double thresholding with hysteresis, for which is need an extra memory with the size of image. To simplify the problem, we will assume that the canny edge detection will be implemented on software level with the CPU containing approximate multiplier.

8.2.2 Design of Inaccurate Multipliers by Partial Specification

In the previous chapter, we showed the design of multiplier by complete specification (S1) in which we checked all of the 2^{16} possible input/output combinations. In this section, I will approximate them by partial specification (S2) and thus we will define only some input/output combinations of the multiplier.

It is obvious that the multiplier doesn't use all of the input combinations in the thinning process. This is proved by Figure 8.4. This plot shows us the usages of combinations and by this plot we will define the partial specification.

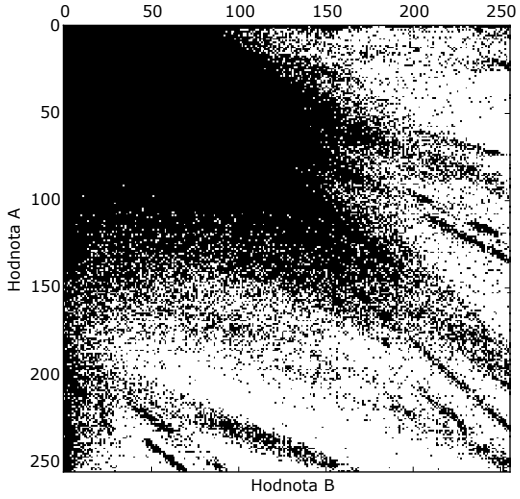


Figure 8.4: The usage of input combination of multiplier in thinning process in Canny edge detection. Black colour shows us if the combination was used for multiplication. The values were taken from several pictures.

In order to have fair comparison of S2, we set the CGP run with the parameters which were used for the design of multiplier by complete specification S1. So, the initialization of algorithm CGP was done by fully functional multiplier, which was constructed by Wallace tree [25]. The amount of gates of fully functional multiplier was $n_g = 330$. This value was decremented after 50 runs of CGP. The CGP settings was chosen as follows: $\Gamma = \{$BUF, INV, AND, OR, XOR, NAND, NOR, XNOR$\}$, $n_r = 1$, $n_c = n_n + 3$, $l = n_c$, $\lambda = 4$, $h = 0.05 n_c$, $n_g = 500\ 000$. The designed circuits were compared with multipliers designed by complete specification.

8.2.3 Experimental Evaluation

In this section, we will compare the relative error of multiplier designed by complete specification (S1) and partial specification (S2) with the error of the detected images.

On the figure 8.6, we can see the results of the detection by inaccurate multipliers developed by method S1. We can see that the edges are thinned correctly until the multiplier consisting 210 gates. In the detection by multipliers having 180 gates, we can see the incorrectly thinned area under the chimney.

In the detailed analysis, we find out that both methods S1 and S2 behave differently. This statement is captured by the plot on figure 8.5. For example the edge detection is worse for method S2 by multipliers having $n_g = 217 \ldots 230$. However in $n_g = 15 \ldots 85$, S2 multipliers are better even they have worse approximated multipliers.

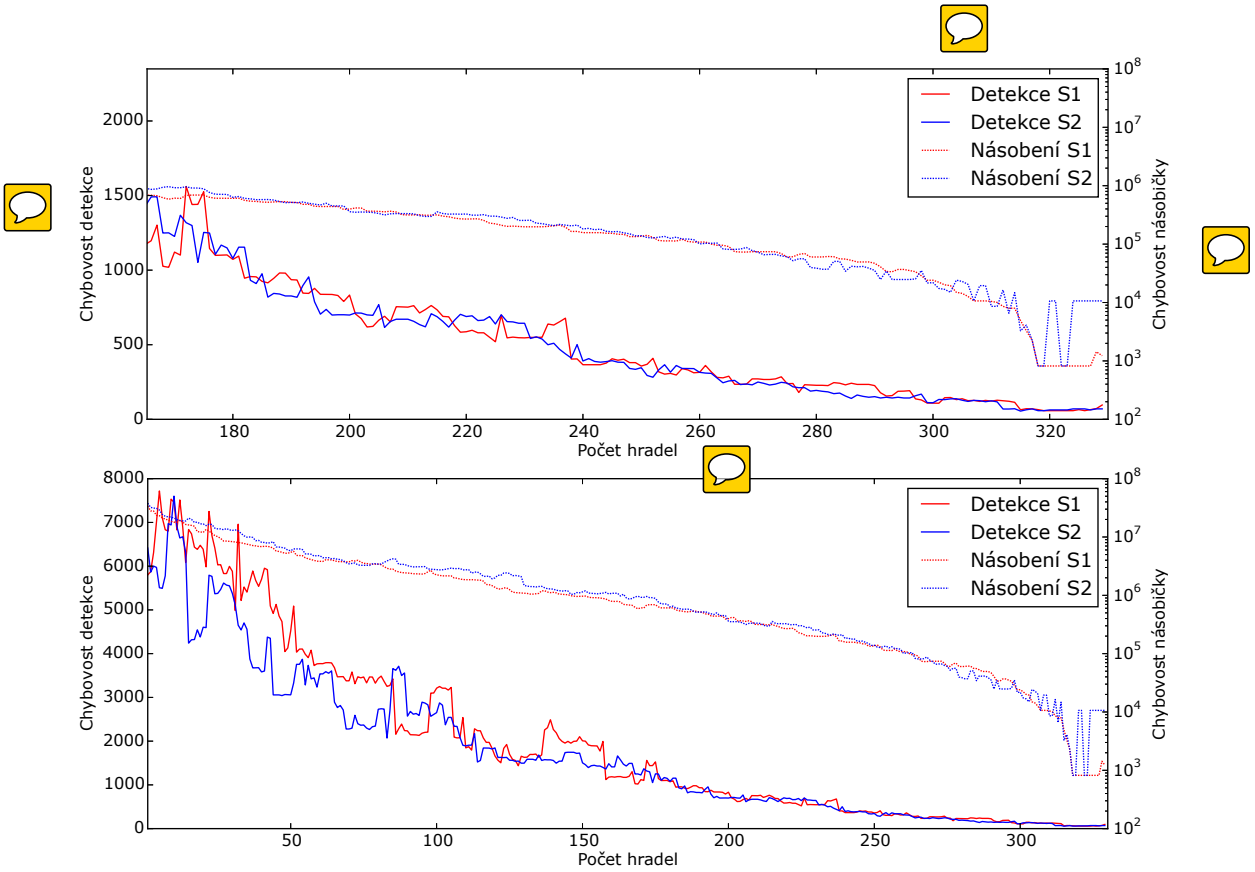
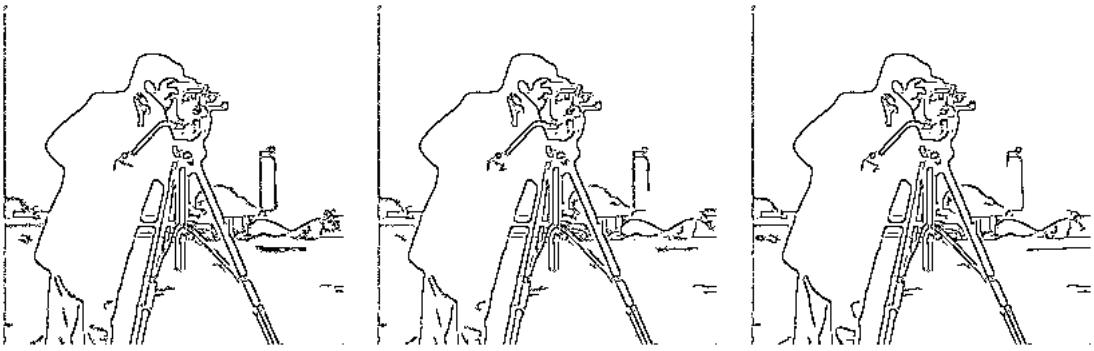


Figure 8.5: Comparison of the Error Size of Approximate Multipliers with the Error in Edge Detection. Error of the multiplier was calculated by formula 4.4. The error of the detection was calculated by incorrectly detected edges towards a fully functional solution.

The main advantage of S2 is in the speed evaluation of candidate solutions, because we do not need to evaluate all the fitness cases which are 2^{16} but only the part of them $31594 \simeq 2^{15}$. The speed up of method S2 is twice faster than method S1.

(a) 1 gate (b) 30 gates (c) 60 gates

(d) 90 gates (e) 120 gates (f) 150 gates



(g) 180 gates (h) 210 gates (i) 240 gates

(j) 270 gates (k) 300 gates (l) 330 gates (fully functional)

Figure 8.6: Approximate Edge Thinning by Using Inaccurate Multipliers where $\sigma = 1, 5$, $T_L = 15$ and $T_H = 30$.

Chapter 9

Conclusion

This work explained the main principles of digital circuit, their parameters and their conventional design methods. The work showed conventional construction of approximate circuits and their evaluation. The work extended the ways of evolutionary approximations by Cartesian genetic programming.

The work presented a new way of fitness value calculation by weighted Hamming distances. This function can be utilized in non-arithmetical combinational circuits in which the output has priority, e.g. coders, multiplexers.

The evaluation of fitness function by sum of absolute differences was accelerated. The evaluation was speed up more than 170 times in case of eight bit multiplier thanks to parallel vector calculation and its inclusion into JIT compilation.

Accelerated CGP was used for the design of 8-bit approximate circuits - adders and multipliers. These adders were replaced in the correctly working convolution in Sobel edge detector. The circuit, realizing the approximate edge detection, has 16% smaller area and it was 9% faster while the error of detection was insignificant for human sight.

Approximate multipliers were used in Canny edge detector for the gradient direction calculation. The work showed two different ways how to evaluate multipliers serving for edge detection. The first way evaluated multiplier by fully specification, and the second one evaluated by partial specification. The second variant allows us to approximate circuits even faster depending on the number of withdrawn training vectors. In the experimental comparison these proposes, we saw that the multipliers designed by partial specification can detect edges more correctly even they are worse approximated.

In addition, the work presented a new type of error plot for inaccurate arithmetical circuit. This plot captures when the error happens and its size.

In the further work, we can focus on the problem of CGP scalability by decomposition or by the dynamic fitness function. We can research for another application which are error resilient. This thesis can continue in the hardware realisation of the approximate circuits and measure their power consumption.

The part of this work was presented on student conference Excel@FIT 2015 in which out of 72 works, it received the award of sponsor company TESCAN, the fifth place in the category of scientific article and the third place in the category for innovative potential.

Bibliography

[1] IEEE Standard for Floating-Point Arithmetic. *IEEE Std 754-2008*, Aug 2008: s. 1–70.

[2] Vsclib Standard Cell Library. 2013.
URL
http://www.vlsitechnology.org/html/cells/vsclib013/lib_gif_index.html

[3] Blinn, J.: Floating-point tricks. *IEEE Computer Graphics and Applications*, ročník 17, č. 4, Jul 1997: s. 80–84, ISSN 0272-1716.

[4] Canny, J.: A Computational Approach to Edge Detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Nov 1986: s. 679–698, ISSN 0162-8828.

[5] Chippa, V.; Chakradhar, S.; Roy, K.; aj.: Analysis and characterization of inherent application resilience for approximate computing. In *50th ACM / EDAC / IEEE Design Automation Conference*, May 2013, ISSN 0738-100X, s. 1–9.

[6] Choudhory, M.: *Approximate logic circuits: Theory and application*. Dizertační práce, Rice University, 2011.

[7] Emre, Y.; Chakrabarti, C.: Quality-Aware Techniques for Reducing Power of JPEG Codecs. *J. Signal Process. Syst.*, Dec 2012: s. 227–237, ISSN 1939-8018.

[8] Goldman, B.; Punch, W.: Reducing Wasted Evaluations in Cartesian Genetic Programming. In *Genetic Programming*, Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2013, ISBN 978-3-642-37206-3, s. 61–72.

[9] Hsiao, P.-Y.; Chen, C.-H.; Chou, S.-S.; aj.: A parameterizable digital-approximated 2D Gaussian smoothing filter for edge detection in noisy image. In *IEEE International Symposium on Circuits and Systems*, 2006, s. 1–4.

[10] Kogge, P. M.; Stone, H. S.: A Parallel Algorithm for the Efficient Solution of a General Class of Recurrence Equations. *IEEE Transactions on Computers*, Aug 1973: s. 786–793, ISSN 0018-9340.

[11] Kulkarni, P.; Gupta, P.; Ercegovac, M.: Trading Accuracy for Power with an Underdesigned Multiplier Architecture. In *24th International Conference on VLSI Design (VLSI Design)*, Jan 2011, ISSN 1063-9667, s. 346–351.

[12] Ludloff, C.; Mocko, M.; Lopes, A.; aj.: X86 Opcode and Instruction Reference tables. URL http://ref.x86asm.net/geek64.html

[13] Miller, J. F.: *Cartesian Genetic Programming*. Natural Computing Series, Springer Berlin Heidelberg, 2011, ISBN 978-3-642-17309-7.

[14] Monajati, M.; Fakhraie, S.; Kabir, E.: Approximate Arithmetic for Low-Power Image Median Filtering. *Circuits, Systems, and Signal Processing*, 2015: s. 1–29, ISSN 0278-081X.

[15] Nepal, K.; Li, Y.; Bahar, R.; aj.: ABACUS: A technique for automated behavioral synthesis of approximate computing circuits. In *Design, Automation and Test in Europe Conference and Exhibition (DATE), 2014*, March 2014, s. 1–6.

[16] Sekanina, L.; Vašíček, Z.: Approximate Circuits by Means of Evolvable Hardware. In *2013 IEEE International Conference on Evolvable Systems (ICES)*, Proceedings of the 2013 IEEE Symposium Series on Computational Intelligence (SSCI), IEEE Computer Society, 2013, ISBN 978-1-4673-5847-7, s. 21–28.

[17] Sekanina, L.; Vašíček, Z.; Růžička, R.; aj.: *Evoluční hardware: Od automatického generování patentovatelných invencí k sebemodifikujícím se strojům*. Edice Gerstner, Academia, 2009, ISBN 978-80-200-1729-1, 328 s.

[18] Shin, D.: *Techniques For Design and Synthesis of Approximate Digital Circuits for Error-tolerant Applications*. Dizertační práce, University of Southern California, 2012.

[19] Vašíček, Z.: Online CGP generator. 2012.
URL http://www.fit.vutbr.cz/~vasicek/cgp/

[20] Vašíček, Z.; Sekanina, L.: Evolutionary Design of Approximate Multipliers Under Different Error Metrics. In *17th IEEE Symposium on Design and Diagnostics of Electronic Circuits and Systems*, IEEE Computer Society, 2014, ISBN 978-1-4799-4558-0, s. 135–140.

[21] Vašíček, Z.; Sekanina, L.: Evolutionary Approach to Approximate Digital Circuits Design. *IEEE Transactions on Evolutionary Computation*, in press, ISSN 1089-778X.

[22] Vašíček, Z.; Slaný, K.: Efficient Phenotype Evaluation in Cartesian Genetic Programming. In *Proc. of the 15th European Conference on Genetic Programming*, Springer Verlag, 2012, ISBN 978-3-642-29138-8, s. 266–278.

[23] Venkataramani, S.; Sabne, A.; Kozhikkottu, V.; aj.: SALSA: Systematic logic synthesis of approximate circuits. In *Design Automation Conference (DAC), 2012 49th ACM/EDAC/IEEE*, IEEE Computer Society, June 2012, ISSN 0738-100X.

[24] Wakerly, J. F.: *Digital Design: Principles and Practices*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., třetí vydání, 2000, ISBN 0130555207.

[25] Wallace, C.: A Suggestion for a Fast Multiplier. *IEEE Transactions on Electronic Computers*, Feb 1964: s. 14–17, ISSN 0367-7508.

[26] Yazdanbakhsh, A.; Thwaites, B.; Park, J.; aj.: Methodical Approximate Hardware Design and Reuse. In *Workshop on Approximate Computing Across the System Stack (WACAS)*, 2014, s. 1–7.

[27] Zemčík, P.; Španěl, M.; Beran, V.; aj.: *Zpracování obrazu – skripta*. Vysoké učení technické v Brně, 2011.

Appendix A

CD Content

Attached CD contains all used source codes, literature and results. It is structured as follows:

- `bib` used literature, bibliography
- `results` result of the work
  - `canny_line_surpression` approximated thinning by Canny edge detector
  - `dumb_adders` approximated adders
  - `dumb_mults` approximated multipliers by full specification (S1)
  - `dumber_mults` approximated multipliers by partial specification (S2)
  - `sobel_detection` edge detection by approximated Sobel detector
- `src` source codes
  - `canny_detector` approximated Canny edge detector
  - `cgp_sad` CGP where fitness was calculated as SAD
  - `cgp_shd` CGP where fitness was calculated as SHD
  - `cgp_wshd` CGP where fitness was calculated as weighted SHD
  - `data` benchmarks
  - `sobel_detector` approximation of Sobel detector
  - `sobel_detector_thold` approximation of Sobel detector with thresholding
- `tex` Czech technical text in TEX

Appendix B

Usage of the Implemented Program

Evolutionary Circuit Design by CGP

Directories

- `/src/cgp_shd`
- `/src/cgp_wshd`
- `/src/cgp_sad`
- `/src/cgp_wsad`

Installation

- `make` chromosome is interpreted
- `make jit` chromosome is precompiled
- `make jitfit` chromosome with fitness function is precompiled

Run

```
./cgp specification.txt [-r rows_count] [-c columns_count] [-l level_back]
[-n maximal_amount_of_gates] [-m mutated_genes_count] [-p population_size]
[-g number_of_generations] [-x filename_of_initial_population] [-s seed]
```

The run has only one required parameter which is the filename containing the circuit specification `specifikace.txt`. The default values are set as follows: `-r 1 -c 40 -l 40 -n 40 -m 1 -p 5 -g 5000000 -s 0`. The seed value is assigned to the actual time. For the successful run with assigned initial population (parameter -x which is not obligated), the file has to contain a chromosome. An example of the chromosomes are in the folder `/src/chromozomy`.

Approximated Sobel Edge Detector

Directories

- `/src/sobel_detector`
- `/src/sobel_detector_thold`

Installation

```
make
```

Run

```
./sobel input_image.png adder.txt threshold
```

The file with approximated adder requires specific syntax. The examples of these files can be found in the folder `/results/dumb_adders`. For successful run, it is required to run the program with all the parameters. However in case of `/src/sobel_detector`, threshold can be omitted. Example:

```
./sobel /src/data/lena.png /results/dumb_adders/20.log 42
```

Approximated Canny Edge Detector

Directory

```
/src/canny_detector
```

Installation

```
make
```

Run

```
./canny input_image.png sigma lower_thold higher_thold multiplier.txt
```

The file with approximated multiplier requires specific syntax. The examples of these files can be found in the folders `/results/dumb_mults` or `/results/dumber_mults`. For successful run, it is required to run the program with all the parameters. Example:

```
./canny /src/data/lena.png 2.0 10 20 /results/dumb_mults/42.log
```