

École Normale Supérieure
de Lyon



Stanford University

Designing arithmetic units for adaptive computing with PAM-Blox

Nicolas Boullis

Second year MIM internship
June to September 2000
under Oskar Mencer and Michael J. Flynn

Résumé en Français

Cette partie, en Français, sert à la fois de résumé et d'introduction au rapport qui suit, et qui a été rédigé en Anglais.

Motivations

Depuis son origine, l'informatique est en quête de puissance. Actuellement, les gains en puissance sont obtenus en augmentant la finesse de gravure des processeurs. Cependant, des limites physiques vont finir par être atteintes (on ne peut pas descendre au-delà de la taille des atomes). Il faut donc s'y préparer, et trouver d'autres solutions pour continuer à gagner en puissance. Bien sûr, il est toujours possible d'augmenter la taille des processeurs, mais c'est aux dépens du prix, du taux de puces ratées et de l'encombrement.

De nombreuses solutions existent, ou sont actuellement étudiées. Il y a, entre autres, les réseaux système à haut débit, l'informatique quantique, ou les transistors mono-électron. Une autre solution pourrait être l'usage de circuits reconfigurables.

En effet, pour un usage donné, il est souvent possible de construire un coprocesseur dédié, qui permet de traiter le problème très efficacement. Malheureusement, il semble impossible de posséder tous les circuits adaptés à tous les problèmes imaginables. Une solution est donc un circuit reconfigurable, qui peut simuler n'importe lequel de ces coprocesseurs dédiés.

Contexte

Le projet PAM, décrit au chapitre 3 de ce rapport, va dans ce sens. Ce projet a permis la création de la carte Pamette, qui sert de coprocesseur reconfigurable, et des environnements de programmation PamDC, PAM-Blox et StReAm.

Pour mon stage de deuxième année du MIM (magistère informatique et modélisation), je suis allé passer onze semaines à l'université de Stanford (en Californie).

J'ai travaillé sur PAM-Blox, avec une carte Pamette. La carte Pamette contient des FPGAs de la série XC4000 de Xilinx ; ces FPGAs sont décrits au chapitre 2.

Le but de mon stage était de concevoir de nouvelles unités arithmétiques.

Travail effectué

Dans un premier temps, j'ai dû concevoir un multiplieur entier. Cette partie m'a demandé un certain temps, nécessaire pour m'habituer aux outils dont je disposais (principalement PamDC, et les FPGAs). La plus grosse difficulté, après la prise en main des outils, a été la gestion des nombres signés. La conception de ce multiplieur est décrite au chapitre 4.

Dans un second temps, j'ai eu à concevoir un diviseur. C'est à ce moment là que me sont apparues les faiblesses de PamDC et PAM-Blox, dont le manque total de contrôle sur le routage. C'est aussi à ce moment là que j'ai découvert quelques erreurs dans la gestion de la chaîne de retenue par PamDC.[1] La conception de ce diviseur est décrite au chapitre 5.

Enfin, il m'a été demandé de concevoir une configuration capable de calculer la fonction logarithme. Aucune des solutions trouvées ne me convenant complètement, j'ai préféré choisir une solution plus simple qui me semblait plus efficace. J'ai ainsi conçu un approximateur générique capable de calculer n'importe quelle fonction (ayant tout de même quelques propriétés, dont le fait d'être de classe $\mathcal{C}^2$ et bornée) à la précision voulue. Bien entendu, j'ai alors dû travailler sur la précision de cet approximateur, prouvant sa précision, en fonction de quelques paramètres. La conception de cet approximateur « générique », et son utilisation pour calculer la fonction logarithme népérien sont décrites au chapitre 6.

Conclusion

Tout d'abord l'approximateur « générique » doit encore pouvoir être amélioré. En particulier, il doit être possible d'utiliser des approximations polynomiales par morceaux de degré supérieur. Cependant, le choix d'une bonne approximation polynomiale sur un intervalle donné est beaucoup moins facile que pour une approximation affine. Si, pour une fonction donnée, on sait, par des méthodes analytiques, trouver une bonne approximation polynomiale de haut degré, ceci semble difficile à automatiser. Devra-t-on donc perdre le caractère générique ?

Il reste aussi beaucoup de travail pour le prochain portage de PamDC et PAM-Blox (StReAm ne devrait pas être affecté) pour la nouvelle série de FPGAs de Xilinx : la série Virtex.

Enfin, une restructuration de PAM-Blox pourrait s'avérer utile. En particulier, il me semble qu'il serait bon de diminuer l'usage qui est fait des « classes paramétrées » (mieux connues sous le nom anglais de *templates*), qui imposent que les paramètres soient connus à la compilation, et posent des problèmes pour une compilation séparée. Il me semble qu'il serait aussi bon de séparer le constructeur d'une classe (représentant une unité arithmétique) de la fonction qui connecte ses entrées. De même, la fonction qui connecte la (les) sortie(s) pourrait être séparée de la fonction qui « construit » l'unité sur le FPGA.

Malgré ces quelques critiques, PAM-Blox, et le projet PAM au grand complet, me semblent avoir un avenir prometteur, et m'ont fourni un sujet de stage qui m'a réellement passionné.

[1] J'ai réussi à trouver et corriger les erreurs en question ; les modifications devraient être incorporées dans la nouvelle version de PamDC. Merci à Mark Shand d'avoir accepté mes modifications.

Aknowledgments

Many people contributed, directly or indirectly, to this internship. Here are a few of them whom I would especially like to thank:

- Michael J. Flynn for accepting me as an intern in his group during the summer, and for his kindness, his help and his bike;
- Oskar Mencer for suggesting me to work with him on PAM-Blox;
- the staff of the Computer Architecture and Arithmetic Group at Stanford University, and especially Hossam Fahmy, Martin Morf and Patrick Hung, for welcoming me during 11 weeks;
- Jean-Michel Muller and Florent Dupont de Dinechin for helping me to get into contact with Michael J. Flynn, and giving me some advice and help when I had problems;
- the Free Software Foundation, and all the developers of free software, for GNU Emacs, GNU make, gcc, and lots of other free softwares;
- Jean Vuillemin, Mark Shand and Laurent Moll for the PAM project;
- Jean Duprat for his course on computer architecture;
- Stanford architects because the university is very beautiful;
- and my girlfriend, Sylvie Boldo for being available whenever I needed, and for reviewing my report (the remaining errors are mine).

Contents

Introduction **1**

1 Basic computer arithmetic **3**
1.1 How to code numbers . . . 3
1.1.1 Coding natural integers . . . 3
1.1.2 Coding signed integers . . . 3
1.1.3 Coding real numbers . . . 5
1.2 Usual hardware designs . . . 5
1.2.1 Half adder and full adder . . . 5
1.2.2 Adding and subtracting integers . . . 5
1.2.3 Converting signed integers . . . 6

2 The Xilinx XC4000 series FPGAs **7**
2.1 General information about FPGAs . . . 7
2.2 Xilinx XC4000 series FPGAs . . . 7
2.2.1 The core of the CLB . . . 7
2.2.2 The fast carry chain . . . 8
2.2.3 Wiring . . . 8

3 Pamette, PamDC and PAM-Blox **9**
3.1 Using FPGAs as versatile coprocessors . . . 9
3.2 Creating configurations . . . 9
3.2.1 PamDC . . . 9
3.2.2 PAM-Blox . . . 10
3.2.3 StReAm . . . 10

4 Designing an array multiplier **11**
4.1 Method used . . . 11
4.2 Design . . . 11
4.3 Handling signed numbers . . . 13
4.4 Implementation . . . 15

5 Designing an array divider **17**
5.1 Method used . . . 17
5.2 Design . . . 18
5.3 Implementation . . . 19

6 Designing a generic approximator **21**
6.1 A simple idea . . . 21
6.2 Accuracy . . . 22
6.3 Speed and space . . . 23
6.3.1 Encodings and widths of numbers . . . 23
6.3.2 Space needed . . . 24
6.3.3 Speed achieved . . . 24
6.4 Computing the natural logarithm . . . 24
6.5 Implementation . . . 25
6.6 Comparison with other schemes . . . 25

Conclusion **27**

Appendices **29**

A Internals of Xilinx XC4000 series FPGAs **31**

B Accuracy of the generic approximator **35**
B.1 Method error . . . 35
B.2 Rounding error . . . 38

Bibliography **39**

List of Figures

1.1 Half adder . . . 5
1.2 Full adder . . . 5
1.3 A 4-bit adder . . . 6

3.1 An example of StReAm code . . . 10

4.1 Decimal multiplication . . . 11
4.2 Binary multiplication . . . 11
4.3 4-bit times 5-bit multiplier . . . 12

5.1 Decimal division . . . 17
5.2 Binary division . . . 17
5.3 Binary non-restoring division . . . 18
5.4 Main block used in the divider . . . 18
5.5 Restoring block used in the divider . . . 18
5.6 Restoring array divider . . . 19

6.1 Approximating function . . . 21
6.2 Internal layout of the generic approximator . . . 24

A.1 A (slightly) simplified block diagram of XC4000 series CLB . . . 31
A.2 Connections between the carry logic and the function generators . . . 32
A.3 Internal diagram of the carry logic . . . 33
A.4 Wires and connections between the CLBs . . . 34

List of Tables

1.1 Different encodings for signed integers 4

6.1 Fixed point encodings used in the approximator 23
6.2 Space comparison for several designs 26
6.3 Speed comparison for several designs 26

Introduction

Computer designers have always tried to build more and more powerful computers. Currently, power is achieved by shrinking the transistors in a processor. Unfortunately, because of the size of the atoms which constitute the matter, there will be a limit to this process. Hence, we have to find other solutions to continue to increase the computational power.

There are many solutions to increase the computational power. Some solutions, such as the use of high-speed system networks, are rather well-known. Other ones, such as quantum computing, are still quite mysterious. Another emerging solution is adaptive computing.

This solution is based on the fact that, for a given problem, we can often build a dedicated coprocessor that is very efficient for this problem. Unfortunately, it looks impossible to have all the dedicated coprocessors for all the problems that we may have to solve.

That is why we may want to have a configurable coprocessor, that can be configured into any coprocessor that we may need. This is what we call adaptive computing. The project PAM, described in chapter 3 is a project of adaptive computing.

The Pamette card, which can be used as a configurable coprocessor, is part of the PAM project. There are several programming tools for the Pamette card, such as PamDC, PAM-Blox and StReAm. All of these (the Pamette card and the three programming tools) are described in chapter 3.

The core of the Pamette card is made of four FPGAs from Xilinx XC4000 series. These FPGAs are described in chapter 2.

During my second year MIM internship, I spent eleven weeks in Stanford University (California). There, I worked with a Pamette card. I had to design a few new arithmetic units for PAM-Blox.

The arithmetic units that I designed are an array multiplier, an array divider, and a logarithm unit. They are described in chapters 4, 5 and 6.

Chapter 1

Basic computer arithmetic

This chapter explains a few notions of computer arithmetic that can be useful to understand the following designs. Other algorithms and hardware designs can be found in [1].

1.1 How to code numbers

Inside a computer, the only known values are booleans. Booleans can be used to code any pair of values, such as true/false, 0/1 or black/white ... We will now consider that the two coded values are 0 and 1. A bit is a boolean coding 0 or 1.

1.1.1 Coding natural integers

The simplest solution to code integers is to use the so-called unary encoding scheme: the number x is encoded by x ones, surrounded by zeros. This scheme is very inefficient because the encoding of the number x needs a size of $\mathcal{O}(x)$.

A much better solution is to use the binary encoding: if $x = \sum_{i=0}^{n-1} x_i 2^i$ with $\forall i \in [0, n-1]$, $x_i \in \{0, 1\}$, then x is encoded by the bits $x_{n-1}x_{n-2}\dots x_2x_1x_0$. Hence, the number 1234 is coded by 10011010010, 0000010011010010, or 00000000000000000000000010011010010.

We see that, with n bits, we can code integers from 0 to $2^n - 1$. So, we can code the number x with a size of $\mathcal{O}(\log_2(x))$ or more.

This coding scheme is much more efficient than the previous one, that is why we will now only consider it.

Usually, we will use a fixed size to code integers. For example, in current processors, integers can usually be coded with 8, 16, 32 or 64 bits.

1.1.2 Coding signed integers

Now that we know how to code unsigned integers, we may want to code signed integers.

Sign-extension encoding

Sign-extension is surely the simplest way to encode signed integers. To encode a number x, you put a bit equal to 0 if $x \geq 0$ or 1 if $x \leq 0$, and then you encode the unsigned integer $|x|$.

Biased encodings

Biased encodings are also very simple: to code a number x in the interval $[-z, 2^n - 1 - z]$, you simply code the unsigned number $x + z$. The number z is called the bias.

Two's complement encoding

This encoding is a little more complex, and uses the modulus operation. To code a number x in the interval $[-2^{n-1}, 2^{n-1} - 1]$, you code the unsigned number $x \bmod 2^n$.

One's complement encoding

This encoding is probably the most difficult to explain. Positive integers are encoded with a zero followed by their usual coding. Negative integers are encoded with a one followed by the bit-complemented coding of their absolute values.

Examples

The table below shows a few integers encoded in 8 bits, with sign-extension, biased (with a bias of 128 and 100), two's complement, and one's complement encodings:

Number	sign-extension	biased (128)	biased (100)	two's complement	one's complement
50	00110010	10110010	10010110	00110010	00110010
0	00000000 or 10000000	10000000	01100100	00000000	00000000 or 11111111
-50	10110010	01001110	00110010	11001110	11001101

Table 1.1: Different encodings for signed integers

Qualities and shortcomings of each encoding

The sign-extension and one's complement encodings have the shortcoming that 0 has two different representations whereas, with the other encodings, each number has only one representation.

For positive integers, the sign-extension, two's complement and one's complement encoding have the quality of being the same as the unsigned encoding. This is not true for biased encodings.

Taking the opposite of a number is very easy with sign-extension and one's complement encodings, a little more difficult with two's complement encoding, and generally even more difficult with biased encodings.

Making additions is the easiest with two's and one's complement encodings; it is more difficult with biased encodings and sign-extension encoding.

Signed multiplications and divisions are easy with sign-extension encodings, more difficult with two's and one's complement encoding, and even more difficult with biased encodings.

After these comparisons, it appears that no choice is really obvious. Everything depends on the way these integers will be used. Current processors generally use two's complement encoding for integers, biased encoding for the exponent of floating point numbers, and sign-extension encoding for the mantissa.

1.1.3 Coding real numbers

Real numbers may be the most difficult to code. Generally, we only code a few rational numbers, except in formal calculus system, which are beyond the scope of this document.

Fixed point encoding

We can code a number $x = k \cdot 2^n$, where k and n are integers, only with a code of k if n is known.

For example, if we decide that $n = -4$, and if we use two's complement encoding, 2.5 is coded by 0101000, and -2.5 by 1011000.

Floating point encoding

We can also code a number $x = k \cdot 2^n$, where k and n are integers, by coding both k and n.

I did not need this kind of encoding during my internship, so I won't describe it any further.

1.2 Usual hardware designs

In this section, I will first present two very useful components: the half adder and the full adder. Most computer arithmetic designs use one or both of them.

Then, we will describe simple designs to add or subtract integers, and to convert signed integers from and to different types of encodings.

1.2.1 Half adder and full adder

The internal hardware of a half adder cell and a full adder cell are described in figures 1.1 and 1.2. Each input or output of each cell is a single bit.

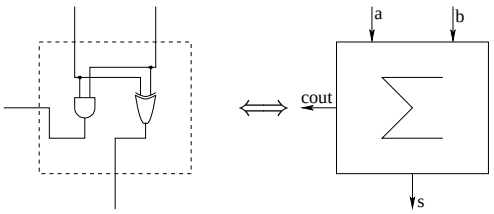


Figure 1.1: Half adder

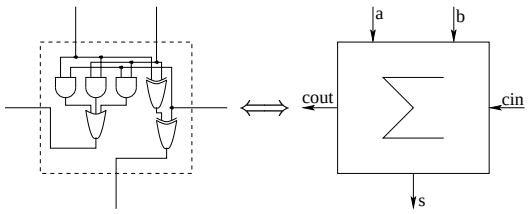


Figure 1.2: Full adder

The half adder counts the number of ones on its inputs (a and b), and outputs its result, in binary, on cout and s, cout being the most significant bit. The full adder counts the number of ones on its inputs (a, b and cin), and outputs its result, in binary, on cout and s.

Hence, the half adder is also called 2-2 counter, and the full adder is called 3-2 counter.

The half adder is simply a full adder with its carry input (cin) set to zero.

1.2.2 Adding and subtracting integers

We can use these two cells to build adders for unsigned integers. You can see such an adder in figure 1.3.

With this adder, we have $\sum_{i=0}^{3} \mathrm{a}_i \cdot 2^i + \sum_{i=0}^{3} \mathrm{b}_i \cdot 2^i = \sum_{i=0}^{4} \mathrm{s}_i \cdot 2^i$.

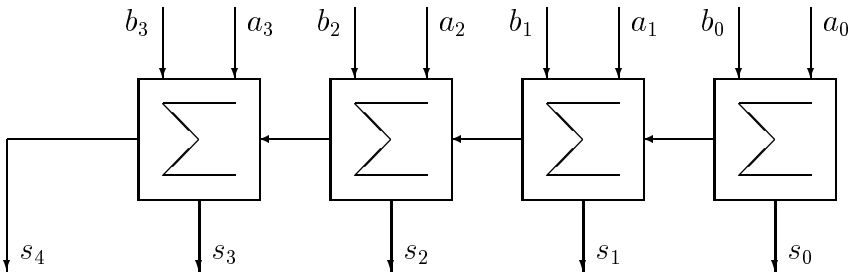


Figure 1.3: A 4-bit adder

To compute the sum of two n-bit two's complement integers, we can reuse this design and ignore the most significant bit of the output. Hence, we perform an addition modulo 2^n. As soon as the result stands in the corresponding interval $\left[-2^{n-1}, 2^{n-1}-1\right]$, it is a correct two's complement result thanks to the equality $a \bmod 2^n + b \bmod 2^n = (a+b) \bmod 2^n$.

We can compute the opposite of a two's complement number by inverting each bit and then adding 1 to the result. So we can perform a subtraction by inverting each bit of the second number, and replacing the half adder of the design with a full adder whose carry input is set to 1.

1.2.3 Converting signed integers

Converting integers from two's complement to biased and back can easily be done with an adder or a subtracter. Converting from sign-extension to one's complement can be done by complementing all the bits (except the sign bit) of negative integers. Converting integers from one's complement to two's complement can be done by adding or subtracting 1 to negative integers.

We can perform any other conversion with only these few ones.

Chapter 2

The Xilinx XC4000 series FPGAs

During my internship, I used Xilinx XC4020E FPGAs. So, I will first present what FPGAs are. Then I will present some specific data about this FPGA.

2.1 General information about FPGAs

The name FPGA stands for "Field Programmable Gate Array". Hence, FPGAs are programmable electronic components.

"Field Programmable" means that their configuration is stored in a RAM ("Random Access Memory"). We can configure them any number of time, but they "forget" their configuration whenever the power is off.

They are made of an array of identical configurable elementary blocks. The connections between these blocks are also configurable.

The blocks are made of boolean function generators. The number of function generators in each block depends on the FPGA. These function generators are simply lookup tables.

A lookup table is a mere read-only memory. The inputs are considered as an address, and the lookup table outputs the value stored in that address.

So, we can map any function with the proper number of inputs and outputs to a given function generators. Hence, a 4-input NAND gate is exactly as complex as a 4-input XOR gate. (This is not true in CMOS VLSI, where XOR gates are much more complex than NAND gates.)

2.2 Xilinx XC4000 series FPGAs

We can now study more specifically the series of FPGAs that we used. In these FPGAs, the blocks are called CLBs (for "Configurable Logic Block").

We will first study the main part of the CLB: the function generators. Then, we will study a facility that is specific to this series of FPGAs: the fast carry chain. We will then finish this section with a brief description of the wiring facility.

The explanations provided in this section are quite simplified. You can find complex schematics in appendix A, and full explanations in [2] and [3] (which deals specifically with the carry chain).

2.2.1 The core of the CLB

Each CLB of these FPGAs is mainly made of 3 boolean function generators, and 2 registers.

Two function generators (called `F` and `G`) have 4 inputs and 1 output. The third one (called `H`) has only 3 inputs and 1 output. Moreover, each of the outputs of `F` and `G` can be connected directly to an input of `H`.

The input of each register can be connected to any output of a function generator. Both registers share the same clock, but each one can either invert the clock signal or not.

The CLB has 4 outputs, called `X`, `Y`, `XQ` and `YQ`. `XQ` and `YQ` are connected to the outputs of the registers; `X` can be connected to the output of `F` or `H`; `Y` can be connected to the output of `G` or `H`.

You can refer to the diagram in appendix A for more details.

2.2.2 The fast carry chain

Let us remember figure 1.3. We can see that each carry output depends on the previous one. Hence, we have a chain of dependencies. This is called a carry chain, and we generally want to avoid it.

A few solutions to avoid these carry chain are explained in [1], and allow to compute all the carries in a logarithmic time instead of a linear time.

On Xilinx XC4000 series FPGAs, there is some extra hardware available, designed to compute quickly the carries (but using the unoptimized method). This dedicated carry logic is so fast that usual speed-up methods are almost useless, even for 32-bit adders.

Moreover, the hardware of the dedicated carry logic does not use the hardware of the generic function generators, but is directly connected to the `F` and `G` function generators. Hence, we can save some hardware by using it.

This dedicated carry logic is very powerful and versatile. It can be used for all kinds of adders, subtracters, incrementers and decrementers. We can even use them to build some hardware that can perform both additions and subtractions, depending on a configuration bit.

You can refer to the diagram in appendix A, and to [3], to figure out what exactly this carry chain can do.

2.2.3 Wiring

There are a lot of wires available in the FPGAs, designed to connect the CLBs.

There are some "single" wires (that are one CLB long), "double" wires (that are two CLB long), "long" wires (that run over half the FPGA) and, on XC4000X series, "quad" wires (that are four CLB long).

Each wire can be connected either to another wire (to build a longer wire) or to an input or output of a CLB.

The rules telling what connections are available are quite complex, because of the internal layout of the FPGA. You can refer to appendix A for a diagram showing all the wires and connections available.

Because of this huge complexity, one may not want to connect the wires "by hand". That is why this process is generally fully automated.

Chapter 3

Pamette, PamDC and PAM-Blox

3.1 Using FPGAs as versatile coprocessors

The PAM project (PAM standing for Programmable Active Memory) was born in 1987, at INRIA, under the leadership of Jean Vuillemin.

The goal of this project was to use several FPGAs as a powerful reconfigurable coprocessor[1].

Hence, in 1992, they built the DECPeRLe-1 card, that was used, for example, to solve Laplace heat equation (this configuration held the record for the highest raw compute power), and to decrypt some RSA encrypted messages.

More recently, another card, called Pamette was built. It is made of 5 FPGAs from Xilinx XC4000 series. The first one is used as a PCI interface; the other ones are available for the user.

During my internship, I used a Pamette card, with four XC4020E user FPGAs.

3.2 Creating configurations

Having some very powerful and versatile hardware is nevertheless not enough. We have to know how to use it. And especially, we need to be able to configure it.

Three development tools are available to build some configurations for the Pamette card: PamDC, PAM-Blox and StReAm. Let us study each of them.

3.2.1 PamDC

PamDC, created by Hervé Touati and Mark Shand allows us to build configurations for Xilinx XC3000 and XC4000 series FPGAs. It is fully described in [4].

PamDC, whose name stands for PAM Design Compiler, is part of the PAM project. It is composed of:

- a library of C++ classes, that are used to describe a configuration in C++, and output a `.dcf` file;
- a few tools to convert the created `.dcf` file to some other formats understood by Xilinx's back-end tools, such as XNF, EDIF and BLIF.

[1] A coprocessor is a component designed to replace the processor for some specific tasks.

The configurations are described in terms of logic equations, registers, and placement constraints. On the opposite, we have no control over the wiring, which is fully performed by Xilinx's back-end tools.

Hence, it is a very low level description, suitable for a computer architect, but not for an end-user.

3.2.2 PAM-Blox

PAM-Blox, created by Oskar MENCER, is described in [5]. It is a collection of C++ classes, that relies on PamDC. Hence, it does not allow either any control on placement.

It allows to describe a configuration in a much higher-level way. There are some classes available for the main arithmetic units. For example, there are some classes to build adders, multipliers, dividers and CORDIC units.

However, this tool is not really suitable for an end-user, as every class for a piece of hardware needs to be declared, initialized, connected and placed.

3.2.3 StReAm

StReAm, also from Oskar MENCER, is described in [6]. It is also a collection of C++ classes; it relies on Pam-Blox.

This tool defines some very high level classes, such as `HWint`, which stands for hardware integer. Moreover, thanks to C++'s operator overloading, we can use these hardware integers very naturally, in additions, subtractions, multiplications...

This tool is designed for some special architectures: those who have to perform the same computation on lots of inputs. Hence, the generated design receives a stream of data in input, and outputs another stream of data. As soon as there is a lot of data, only the throughput (time between two results) is important. The latency (time between an input and the corresponding output) is meaningless.

Hence, each PAM-Blox unit should be able to run at 33MHz (which is the speed of the PCI bus), and may be pipelined for that.

You can find an example of StReAm code in figure 3.1. This piece of code computes the IDCT ("Inverse Discrete Cosine Transform").

```
const int NUM_BLOCK_INPUTS=8;
const int NUM_BLOCK_OUTPUTS=8;
const int BITS = 14;
const int COMP_MODE=PARALLEL;

const int coef[8] = {16069, 15137, 13623,
          11585, 11585, 3196, 6270, 9102};

HWint<BITS> t[18];

IDCT::build() {

t[0] =((in[0]+in[4])*coef[4]+256)>>9;
t[1] =((in[0]-in[4])*coef[4]+256)>>9;
t[2] =(in[2]*coef[6]-in[6]*coef[1]+256)>>9;
t[3] =(in[2]*coef[1]+in[6]*coef[6]+256)>>9;
t[4] =(in[1]*coef[0]+in[7]*coef[5]+256)>>9;
t[5] =(in[1]*coef[5]-in[7]*coef[0]+256)>>9;
t[6] =(in[3]*coef[2]+in[5]*coef[7]+256)>>9;
t[7] =(in[3]*coef[7]-in[5]*coef[2]+256)>>9;

t[8] =(t[0]+t[3]+2)>>2;
t[9] =(t[1]+t[2]+2)>>2;
t[10]=(t[1]-t[2]+2)>>2;
t[11]=(t[0]-t[3]+2)>>2;

t[0]=(t[4]+t[6]+2)>>2;
t[1]=(t[5]+t[7]+1)>>1;
t[2]=(t[4]-t[6]+1)>>1;
t[3]=(t[5]-t[7]+2)>>2;

t[4]=(((t[1]+t[2]+1)>>1)*coef[3]+8192)>>14;
t[5]=(((t[1]-t[2]+1)>>1)*coef[3]+8192)>>14;

 out[0]=(t[8]+t[0]+4)>>3;
 out[1]=(t[9]+t[4]+4)>>3;
 out[5]=(t[10]+t[5]+4)>>3;
 out[3]=(t[11]+t[3]+4)>>3;
 out[7]=(t[8]-t[0]+4)>>3;
 out[6]=(t[9]-t[4]+4)>>3;
 out[2]=(t[10]-t[5]+4)>>3;
 out[4]=(t[11]-t[3]+4)>>3;
}
```

Figure 3.1: An example of StReAm code

Chapter 4

Designing an array multiplier

4.1 Method used

Let us first remember how we learned multiplications at school. Figures 4.1 and 4.2 show the same multiplication, in decimal and in binary.

		1	2	3
×			4	5
		6	1	5
	4	9	2	
	5	5	3	5

Figure 4.1: Decimal multiplication

						1	1	1	1	0	1	1		
					×		1	0	1	1	0	1		
						1	1	1	1	0	1	1		
					0	0	0	0	0	0	0			
				1	1	1	1	0	1	1				
			1	1	1	1	0	1	1					
		0	0	0	0	0	0	0						
	1	1	1	1	0	1	1							
1	0	1	0	1	1	0	0	1	1	1	1	1		

Figure 4.2: Binary multiplication

We use a method very close to this method in our hardware design. We have the following equality:

$$\left(\sum_{i=0}^{n-1} a_i 2^i\right) \cdot \left(\sum_{j=0}^{m-1} b_j 2^j\right) = \sum_{i=0}^{n-1} \sum_{j=0}^{m-1} a_i b_j 2^{i+j}$$

Hence, we have to add $n \cdot m$ numbers. These numbers are only 1-bit wide, but have different significances.

4.2 Design

How to add these $n \cdot m$ bits? Let us remember that we can use a full adder as a counter. Hence, a full adder can take three bits of equal weight in input, and output one bit of the same weight, and one bit of the next higher weight. We can also do the same with a half adder (which takes only two input bits).

Moreover, the product of two bits is the logical AND of them. (You can check this with the two truth tables.)

Hence, we get the design shown in figure 4.3, for a 4-bit times 5-bit multiplier.

This design repeats the same pattern in the center. Only the edges differ from that pattern. Hence, this design is very easy to scale for smaller or larger multipliers.

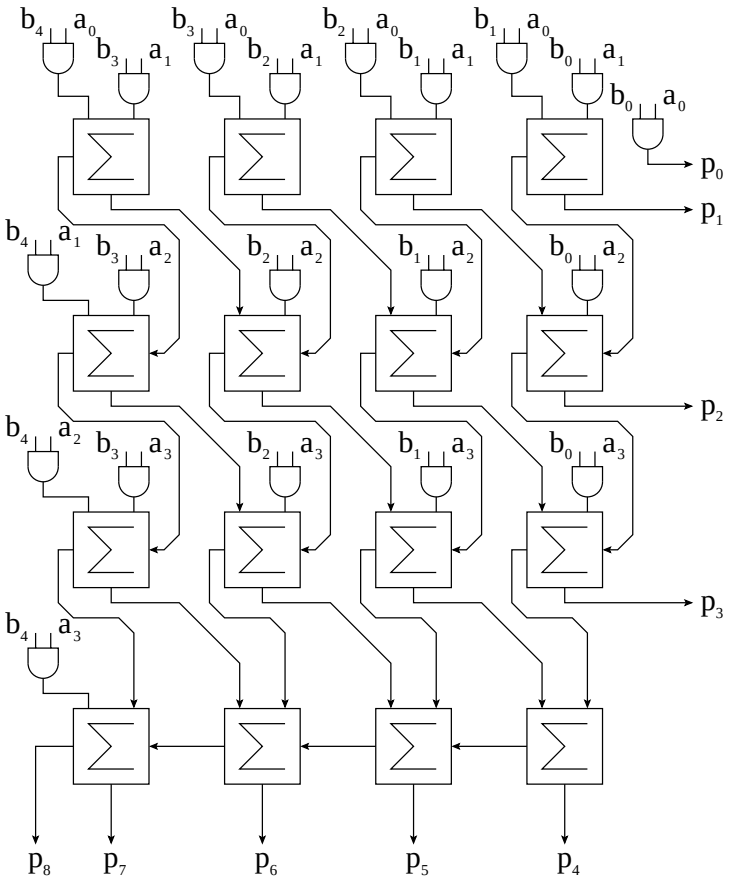


Figure 4.3: 4-bit times 5-bit multiplier

Moreover, it uses only small wires in internal; hence, it is really easy to map it to a FPGA, and there is no wiring problem for Xilinx's back-end tool.

The main pattern and the right-most pattern (one full adder and one AND gate) have four inputs and two outputs. Hence, each can be mapped in a single CLB.

The top pattern (one half adder and two AND gates) also has four inputs and two outputs. Hence, it can also be mapped to a single CLB.

The left-most pattern (one full adder and two AND gates) has five inputs and two outputs. Hence, we cannot map it to a single CLB.[1] So, we map this pattern to two CLBs: one for an AND gate, and one for all the remaining hardware.

Last, the bottom pattern looks like an adder. Hence, we can use the dedicated carry chain to compute it.[2] So, we could use half a CLB for each cell. But, doing so, the wires from the previous rows to the last row would not be straightforward. So, it is better to use one CLB for each cell.

Hence, we get a total design such that a n-bit times m-bit multiplier fits in a $n \times m$ rectangle. This kind of multiplier is called an array multiplier.

With this design, there is no dependency between several cells in the same row (except the last row). So, whatever the width of the multiplier may be, the signal can go through 5 rows in less than 30 ns. Hence, this design is pipelined every 5 rows.

There is a carry chain in the last row. So, we have some dependencies, and each cell may be unable to compute its outputs before the previous one (which is on the right) has finished. Hence, the signal needs more time to go through the last row (from the inputs of the rightmost cell to the outputs of the leftmost one) than it needs to go through the other cells. Hence, the last stage of the pipeline is smaller than the other ones.

4.3 Handling signed numbers

Let us remember how we learned to multiply signed integers at school. We learned a simple rule. The absolute value of the product is the product of the absolute values of the factors. The sign of the product is + if both factors have the same sign, and - in the other cases.

Hence, multiplying two sign-extended numbers is straightforward: we need the unsigned multiplier to compute the absolute value of the product, and a XOR gate to compute the sign of the product.

Multiplying two two's complement integers is not so straightforward. If a and b are n-bit and m-bit two's complement integers, we have $a = -a_{n-1}2^{n-1} + \sum_{i=0}^{n-2} a_i 2^i$ and $b = -b_{m-1}2^{m-1} + \sum_{j=0}^{m-2} b_j 2^j$.

[1] We might want to use the H function generator for one AND gate, and use the F and G for the outputs, but, as there is no direct connection from H to F and G, we would need three unregistered outputs, and we have only two.

[2] We could use it elsewhere, but it would slow down the design, without making it smaller.

Hence, we have:

$$\begin{aligned}
a \cdot b &= \left(-a_{n-1}2^{n-1} + \sum_{i=0}^{n-2} a_i 2^i\right)\left(-b_{m-1}2^{m-1} + \sum_{j=0}^{m-2} b_j 2^j\right) \\
&= a_{n-1}b_{m-1}2^{n+m-2} - a_{n-1}2^{n-1}\sum_{j=0}^{m-2} b_j 2^j - b_{m-1}2^{m-1}\sum_{i=0}^{n-2} a_i 2^i + \left(\sum_{i=0}^{n-2} a_i 2^i\right)\left(\sum_{j=0}^{m-2} b_j 2^j\right) \\
&= a_{n-1}b_{m-1}2^{n+m-2} + \sum_{i=0}^{n-2}\sum_{j=0}^{m-2} a_i b_j 2^{i+j} \\
&\quad -a_{n-1}\left(\sum_{j=0}^{n-2} 0 \cdot 2^j + \sum_{j=0}^{m-2} b_j 2^{j+n-1} + 0 \cdot 2^{m+n-2} + 0 \cdot 2^{m+n-1}\right) \\
&\quad -b_{m-1}\left(\sum_{i=0}^{m-2} 0 \cdot 2^i + \sum_{i=0}^{n-2} a_i 2^{i+m-1} + 0 \cdot 2^{m+n-2} + 0 \cdot 2^{m+n-1}\right)
\end{aligned}$$

Then we have, modulo 2^{n+m}:

$$\begin{aligned}
a \cdot b &= a_{n-1}b_{m-1}2^{n+m-2} + \sum_{i=0}^{n-2}\sum_{j=0}^{m-2} a_i b_j 2^{i+j} \\
&\quad +a_{n-1}\left(\sum_{j=0}^{n-2} 1 \cdot 2^j + \sum_{j=0}^{m-2} \bar{b_j} 2^{j+n-1} + 1 \cdot 2^{m+n-2} + 1 \cdot 2^{m+n-1} + 1\right) \\
&\quad +b_{m-1}\left(\sum_{i=0}^{m-2} 1 \cdot 2^i + \sum_{i=0}^{n-2} \bar{a_i} 2^{i+m-1} + 1 \cdot 2^{m+n-2} + 1 \cdot 2^{m+n-1} + 1\right) \\
&= a_{n-1}b_{m-1}2^{n+m-2} + \sum_{i=0}^{n-2}\sum_{j=0}^{m-2} a_i b_j 2^{i+j} \\
&\quad +a_{n-1}\left(1 \cdot 2^{n-1} + \sum_{j=0}^{m-2} \bar{b_j} 2^{j+n-1} + 1 \cdot 2^{m+n-2} + 1 \cdot 2^{m+n-1}\right) \\
&\quad +b_{m-1}\left(1 \cdot 2^{m-1} + \sum_{i=0}^{n-2} \bar{a_i} 2^{i+m-1} + 1 \cdot 2^{m+n-2} + 1 \cdot 2^{m+n-1}\right) \\
&= a_{n-1}b_{m-1}2^{n+m-2} + \sum_{i=0}^{n-2}\sum_{j=0}^{m-2} a_i b_j 2^{i+j} \\
&\quad +\left(a_{n-1}2^{n-1} + \sum_{j=0}^{m-2} a_{n-1}\bar{b_j} 2^{j+n-1} + a_{n-1}2^{m+n-2} + a_{n-1}2^{m+n-1}\right) \\
&\quad +\left(b_{m-1}2^{m-1} + \sum_{i=0}^{n-2} \bar{a_i} b_{m-1} 2^{i+m-1} + b_{m-1}2^{m+n-2} + b_{m-1}2^{m+n-1}\right) \\
&= \sum_{i=0}^{n-2}\sum_{j=0}^{m-2} a_i b_j 2^{i+j} + \sum_{j=0}^{m-2} a_{n-1}\bar{b_j} 2^{j+n-1} + \sum_{i=0}^{n-2} \bar{a_i} b_{m-1} 2^{i+m-1} \\
&\quad +a_{n-1}2^{n-1} + b_{m-1}2^{m-1} + a_{n-1}b_{m-1}2^{n+m-2} + (a_{n-1} + b_{m-1})\left(2^{n+m-2} + 2^{n+m-1}\right)
\end{aligned}$$

The first three terms that we get are very close to what we have for an unsigned multiplier. Only the terms $a_{n-1}2^{n-1}$, $b_{m-1}2^{m-1}$, and the two most significant bits really differ.

Hence, we only need to modify slightly the unsigned multiplier to compute signed multiplications. The new signed multiplier fits in the same rectangle that the unsigned one.

4.4 Implementation

First, the implementation of the unsigned array multiplier was quite straightforward. Pipelining it was also quite easy.

Then, I wanted to extend it to handle two's complement signed integers. With the previous result, I could build a multiplier with two's complement signed inputs. And, after a few simplifications of this signed design, I got a multiplier able to receive any combination of unsigned or two's complement signed integers as inputs. The output is unsigned if both inputs are unsigned; else it is two's complement signed.

With all these design, the size of the design and the output depends only on the width of the inputs, not whether they are signed.

Next, I wanted to handle sign-extended integers, which looked very easy. Unfortunately, it is not so easy, as the sizes of the design and the output vary whether the inputs are signed or unsigned. Because of some technical reasons (including the use of C++ `templates`), I could not manage to handle sign-extended numbers.

Chapter 5

Designing an array divider

5.1 Method used

Let us first remember how we learned divisions at school. Figures 5.1 and 5.2 show the same division, in decimal and in binary.

```
    1 2 3 4 | 5 6
-   1 1 2   | 2 2
      1 1 4
    -   1 2
          2
```

Figure 5.1: Decimal division

```
      1 0 0 1 1 0 1 0 0 1 0 | 1 1 1 0 0 0
-       1 1 1 0 0 0         | 1 0 1 1 0
          1 0 1 0 1 0
        - 0 0 0 0 0 0
          1 0 1 0 1 0 0
        -   1 1 1 0 0 0
              1 1 1 0 0 1
            - 1 1 1 0 0 0
                      1 0
          -   0 0 0 0 0 0
                    1 0
```

Figure 5.2: Binary division

This method "by hand" is very close to the restoring division algorithm: we try each subtraction; if the result is positive or zero, we keep it and write 1 to the quotient; if the result is negative, we restore the previous value and write 0 to the quotient.

There is another method, slightly more efficient, called the non-restoring algorithm: after each operation, we look at the result; if the result is positive or zero, we write a 1 to the quotient and the next operation is a subtraction; if the result is negative, we write a 0 to the quotient and the next operation is an addition. We always keep the result of an operation.

At the end, if the remainder is positive or zero, we write a last 0 to the quotient; if it is negative, we write a 1 to the quotient, and we add the divisor to the remainder. (This is a kind of restoring stage.)

Figure 5.3 shows the same division, performed "by hand" with a non-restoring algorithm.

This method gives the correct result thanks to the equality $R - 2^{k+1}D + 2^kD = R - 2^kD$. Each time we get a positive result, it is the same as with the restoring algorithm. When we get a negative result, the previous subtraction and the next addition are equivalent to a canceled subtraction and a subtraction. Hence, we can prove that we get the same result that with the restoring algorithm.

	1	0	0	1	1	0	1	0	0	1	0	1	1	1	0	0	0
-		1	1	1	0	0	0	0	0	0	0	1	0	1	1	0	
			1	0	1	0	1	0	0	1	0						
		-	1	1	1	0	0	0	0	0	0						
				-	1	1	0	1	1	1	0						
			+	1	1	1	0	0	0	0	0						
					1	1	1	0	0	1	0						
				-	1	1	1	0	0	0	0						
										1	0						
					-	1	1	1	0	0	0						
					-	1	1	0	1	1	0						
					+	1	1	1	0	0	0						
										1	0						

Figure 5.3: Binary non-restoring division

5.2 Design

To describe in a simpler way the whole divider, we have to define a block that will be used in the divider. This block is shown in figure 5.4. It contains one full adder and one XOR gate. We also need a block for the restoring stage. It is shown in figure 5.5. It contains one full adder and one multiplexer.[1]

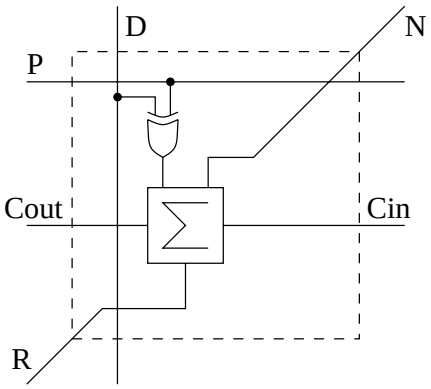


Figure 5.4: Main block used in the divider

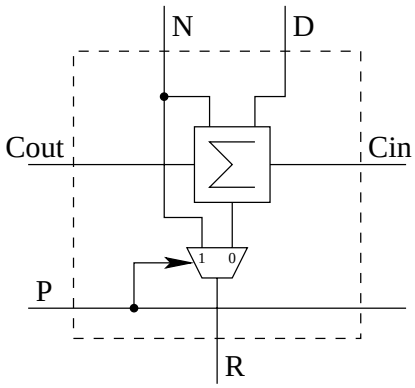


Figure 5.5: Restoring block used in the divider

With several main blocks, we can build an adder/subtracter which, depending on the value in the P input, adds or subtract the value from the D input to the value from the N input. The result is send to the R output.

With several restoring blocks, we can build the restoring stage: if the P input (which is the same for all the cells) equals one, the value on the R outputs equals the value on the N inputs; else, the value on the R outputs equals the some of the value on the N inputs and the value on the D inputs.

Hence, we can build a full non-restoring array divider. An example of diagram for such a divider is shown in figure 5.6.

Thanks to the dedicated carry chain, each cell fits in half a CLB. Hence, a divider with a d-bit divisor and a q-bit quotient fits in a $\lceil \frac{d+1}{2} \rceil \times (q+1)$ rectangle. For that, because of how the dedicated carry chain is built, we have to invert the P input of each block.

[1] A multiplexer is a component with two data inputs, one control input and one output. Depending on the value of the control input, the output is equal to one data input or the other one.

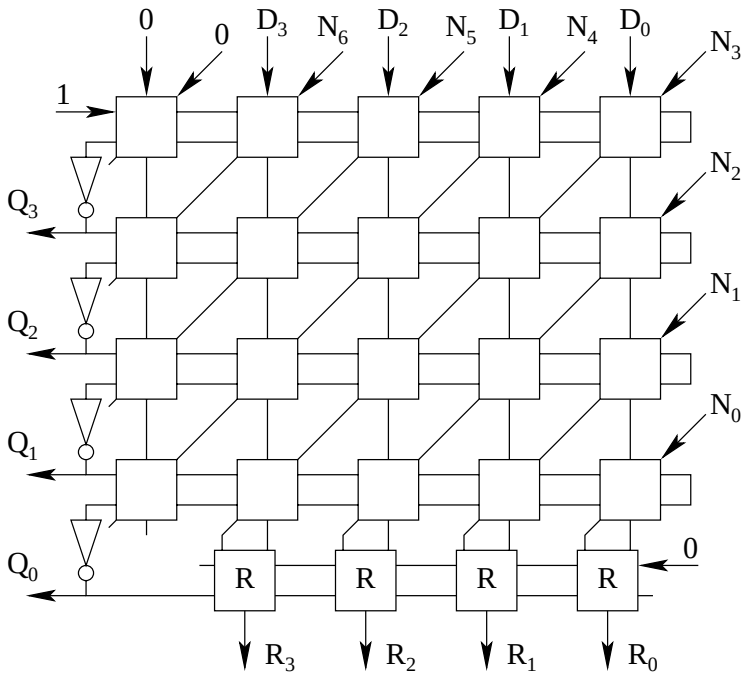


Figure 5.6: Restoring array divider

This divider works only if the quotient fits in q bits. Else, the result is unpredictable. The only solution is to ask for a larger quotient, and pad the dividend (on the left) with zeros.

5.3 Implementation

Implementing the combinational (non-pipelined) version of this divider was quite straightforward, after I understood how it really worked.

Then, I wanted to pipeline it. Here I got my first problems. Some outputs have a very high fan-out[2]; for example all the P inputs in the same row are driven by the same output. Because of this, Xilinx's back-end tools add some buffers. Unfortunately, the choices made by these tools look quite unpredictable. Hence, it is very difficult to guess the delay of a signal.

So, we have to make a pessimistic assumption for the delays to pipeline the design. Then, for large designs (8-bit divisors seems to be a correct limit), we have to add registers to every row of the design. So we would need 4 registers per CLB, but we have only 2. So the design has to be bigger.

The next problem appears while trying to handle signed numbers. We need a convention for the sign of the remainder. According to [7], the usual convention is that the sign of the remainder is the same as the sign of the dividend. Hence, handling signed numbers should be very easy with sign-extended numbers: we would have to compute the division with the absolute value; then the sign of the remainder is the sign of the dividend, and the sign of the quotient is the "product" of the signs of the dividend and divisor.

Then, the sizes of the design and of the outputs does not only depend with the sign of the

[2]This means that they are connected to many inputs.

inputs. So, because of some technical reasons (including the use of C++ `templates`), I could not implement it.

Handling two's complement numbers is really more difficult, and was not implemented either.

Chapter 6

Designing a generic approximator

After the multiplier and the divider, I was asked to design some hardware to compute logarithms. I chose to compute it on the interval $[1, 2)$, in fixed point arithmetic.

A few designs, such as Wong and Goto's method and shift-and-add methods are described in [8].

None of these method seemed very appropriate to calculations in FPGAs. Wong and Goto's method looked too complex because it needs several multipliers and lookup tables; shift-and-add methods seemed to slow, as each stage needs the full result of the previous stage to start.

I wanted to try something both simpler and faster.

6.1 A simple idea

Let us consider a function F that is $\mathcal{C}^2$ [1] on the interval $[0, 1]$. Let us consider that we want to compute it on the interval $[0, 1)$, and that $\forall x \in [0, 1], F(x) \in [0, 1)$.

We will cut this interval $[0, 1)$ into several smaller intervals, and use an affine approximation on each interval. So we need two lookup tables to store the coefficients of the local affine approximation, a multiplier and an adder.

We cut the $[0, 1)$ interval into the $n = 2^m$ intervals $\left[\frac{i}{n}, \frac{i+1}{n}\right)$ $(i \in [0, n-1])$. Then, the approximating function f is the function which is continuous on $[0, 1]$, affine on each interval $\left[\frac{i}{n}, \frac{i+1}{n}\right]$ $(i \in [0, n-1])$, and such that $\forall i \in [0, n], f\left(\frac{i}{n}\right) = F\left(\frac{i}{n}\right)$. An example is shown on figure 6.1.

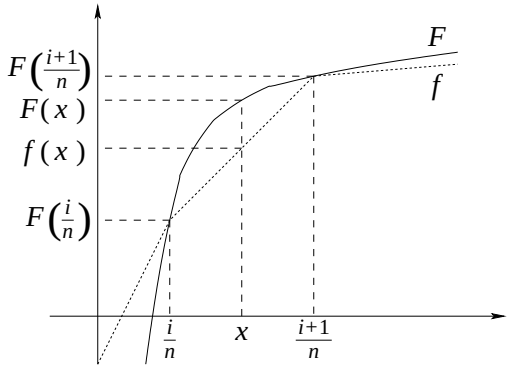


Figure 6.1: Approximating function

[1] $\mathcal{C}^2$ functions are functions that can be derived twice, and whose second derivative is continuous.

Assuming that $x \in [0,1)$, we can compute this function with the following algorithm:

$i = \lfloor n \cdot x \rfloor$
$y = n \cdot x - i$
$v_0[i] = F\left(\frac{i}{n}\right)$
$v_1[i] = F\left(\frac{i+1}{n}\right)$
$c_0[i] = v_0[i]$
$c_1[i] = v_1[i] - v_0[i]$
$a = y \cdot c_1[i]$
$b = a + c_o[i]$
$f(x) = b$

Hence, we have $f(x) = y \cdot F\left(\frac{i+1}{n}\right) + (1-y) \cdot F\left(\frac{i}{n}\right)$.

The values $v_0[i]$, $v_1[i]$, $c_0[i]$ and $c_1[i]$ ($i \in [0, n-1]$) are pre-computed, and $c_0[i]$ and $c_1[i]$ are stored in two lookup tables. The FPGA computes i (which is equal to the first m fractional bits of x) and y (which is equal to the following bits of x). Then it looks up $c_0[i]$ and $c_1[i]$, and it computes a, b and $f(x)$.

The values $c_0[i]$ and $c_1[i]$ are stored with k fractional bits, and a and b are computed with k fractional bits, whereas $f(x)$ is rounded to l fractional bits (with $l < k$). (For tips on choosing k for a given l, see the section "Computing the natural logarithm".)

Frow now on, $\widetilde{\alpha}$ will stand for the value actually computed in the FPGA for α. Because of roundings, we will generally have $\widetilde{\alpha} \neq \alpha$. Moreover, $\widetilde{f}$ will stand for the function actually computed; so we have $\widetilde{f}(x) = \widetilde{f(x)}$.

All the values that are not exact are rounded to the nearest (with rounding to $+\infty$ tiebreaking rule), with a few exceptions explained later.

6.2 Accuracy

Let us first compare the function F with its piecewise affine approximation f. After calculations shown in appendix B (page 35), we get the following result:

$$\boxed{\forall x \in [0,1), \left|F(x) - f(x)\right| \leq 2^{-(2m+3)} \sup_{z \in [0,1]} \left|F''(z)\right|}$$

We can then compare the function f with the function $\widetilde{f}$ we get from the FPGA (these values may differ due to roundings). The calculations shown in appendix B lead to the following result:

$$\boxed{\forall x \in [0,1), \left|f(x) - \widetilde{f}(x)\right| < \left(1 - \frac{1}{4}\right) 2^{-(k-1)} + 2^{-(l+1)}}$$

We want to have a total error smaller than 1 u.l.p.[2], as this is often done for such designs. This means that we want that $\forall x \in [0,1), \left|F(x) - \widetilde{f}(x)\right| < 2^{-l}$. To ensure this property, we should have the following inequality:

[2] u.l.p. stands for "unit in the last place", which is the weight of the least significant bit

$$\boxed{2^{-(2m+3)} \cdot \sup_{z\in[0,1]} \left|F''(z)\right| + \left(1-\frac{1}{4}\right) 2^{-(k-1)} \leq 2^{-(l+1)}}$$

6.3 Speed and space

6.3.1 Encodings and widths of numbers

Let us first remember that all the number are encoded in fixed point. $\widetilde{f}(x)$ is encoded with l fractional bits (*i.e.* with an implicit exponent of $-l$), whereas all the other numbers are encoded with k fractional bits (*i.e.* with an implicit exponent of $-k$).

We know that $\forall x \in [0,1), F(x) \in [0,1)$. So, we can slightly modify the rounding mode to have $\forall i \in [0, n-1]$, $\widetilde{v_0[i]} \in \left[0, 1-2^{-k}\right]$ and $\widetilde{v_1[i]} \in \left[0, 1-2^{-k}\right]$. Then, we also have $\widetilde{c_0[i]} \in \left[0, 1-2^{-k}\right]$ and $\widetilde{c_1[i]} \in \left[-\left(1-2^{-k}\right), 1-2^{-k}\right]$. Hence, we will encode $\widetilde{c_0[i]}$ with k bits, and $\widetilde{c_1[i]}$ with $k+1$ bits (except if we can be sure that $\widetilde{c_1[i]} \geq 0$ so we can use only k bits).

Then we have $\widetilde{a} \in \left[-\left(1-2^{-k}\right), 1-2^{-k}\right]$, and $\widetilde{b} \in \left[0, 1-2^{-k}\right]$ [3]. And, because of the last rounding (which is a rounding to the nearest to l fractional bits), we should have $\widetilde{f}(x) \in [0,1]$. But we can change slightly the rounding mode to get $1-2^{-l}$ instead of 1. This is correct because, if we get 1, then $|F(x)-1| < 2^{-l}$, but $F(x) < 1$, so $1-2^{-l} < F(x) < 1$ and then $\left|F(x) - \left(1-2^{-l}\right)\right| < 2^{-l}$.

We can now choose how to encode (for width and sign) each value in fixed point. The results are shown in table 6.1.

Number	sign	width	implicit exponent
$\widetilde{c_0[i]}$	unsigned	k	$-k$
$\widetilde{c_1[i]}$	2's complement	$k+1$	$-k$
$\widetilde{a}$	2's complement	$k+1$	$-k$
$\widetilde{b}$	unsigned	k	$-k$
$\widetilde{f}(x)$	unsigned	l	$-l$

Table 6.1: Fixed point encodings used in the approximator

Let p be the number of fractional bits of the input. (We assume that $p > m$.) Generally, we have $p \simeq l$.

As $x \in [0,1)$, x is encoded as a fixed point number with a p-bit unsigned mantissa, and an implicit exponent of $-p$.

Then i is a m bit unsigned integer, and y is a fixed point number with a $p-m$ bit unsigned mantissa, and an implicit exponent of $-(p-m)$.

[3] Proving this property of $\widetilde{b}$ is not difficult but not either straightforward. We have to use the definitions of $c_0[i]$, $c_1[i]$, a and b, and to consider the roundings errors.

6.3.2 Space needed

We can now try to figure out how wide the whole design will be. This whole design contains one lookup table with 2^m numbers of $k+1$ bits, one lookup table with 2^m numbers of k bits, one $(k+1)$-bit times $(p-m)$-bit multiplier, one $(k+1)$-bit adder, and a l-bit rounding unit.

If $m \geq 4$, then the two lookup tables fit in rectangles of size $\frac{2^{m-3}+1-2\cdot(m \bmod 2)}{3} \times (k+1)$ and $\frac{2^{m-3}+1-2\cdot(m \bmod 2)}{3} \times k$. (If $m < 4$ the rectangles are of size $1 \times (k+1)$ and $1 \times k$.)

The multiplier fits in a $(p-m) \times (k+1)$ rectangle; the adder in a $1 \times (k+1)$ rectangle, and the rounding unit in a $1 \times l$ rectangle.

Hence, the whole design fits in a $\left(2 \cdot \frac{2^{m-3}+1-2\cdot(m \bmod 2)}{3} + (p-m) + 2\right) \times (k+1)$ rectangle, as shown in figure 6.2, assuming that $m \geq 4$.

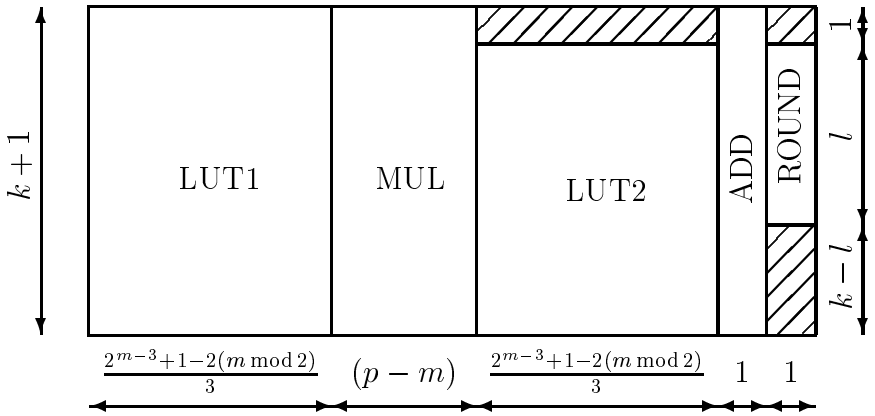


Figure 6.2: Internal layout of the generic approximator

6.3.3 Speed achieved

The delay of both lookup tables is $1 + \max\left(0, \lceil \frac{m-5}{6} \rceil\right)$ cycles. Both lookups can be performed in parallel.

The delay of the multiplier is $1 + \left\lfloor \frac{p-m+\left\lceil \frac{8(k-2)}{35} \right\rceil}{5} \right\rfloor$ cycles.

The delay of the adder is 1 cycle, as the delay of the rounding unit.

Hence, the total delay of the approximator is:

$$\boxed{\max\left(0, \left\lceil \frac{m-5}{6} \right\rceil\right) + \left\lfloor \frac{p-m+\left\lceil \frac{8(k-2)}{35} \right\rceil}{5} \right\rfloor + 4}$$

6.4 Computing the natural logarithm

We should keep in mind that our goal was to compute the natural logarithm in the interval $[1, 2)$. So, we can choose $F(x) = \ln(1+x)$. This function fulfills all the requirements. Then, we have

$F'(x) = \frac{1}{1+x}$, $F''(x) = -\frac{1}{(1+x)^2}$. Hence, $\sup_{z\in[0,1]} \left|F''(z)\right| = 1$.

Let us assume that we want the input and output with q bits. So, we have $l = q$ and $p = q - 1$. We now have to choose m and k.

Because of the term 2^m in the space needed by the design, we want to choose the smallest m possible. We must have $2^{-(2m+3)} \cdot \sup_{z\in[0,1]} \left|F''(z)\right| < 2^{-(l+1)}$, so we choose $m = \left\lfloor \frac{l}{2} \right\rfloor$.

If l is even, we have $m = \frac{l}{2}$. We have to choose k such that $\left(1 - \frac{1}{4}\right) 2^{-(k-1)} \leq 2^{-(l+1)} - 2^{-(l+3)}$. So, we choose $k = l + 2$.

If l is odd, we have $m = \frac{l-1}{2}$. We have to choose k such that $\left(1 - \frac{1}{4}\right) 2^{-(k-1)} \leq 2^{-(l+1)} - 2^{-(l+2)}$. So, we choose $k = l + 3$.

Moreover, the logarithm is an increasing function. So, through the rounding used, we can force $\forall i \in [0, n-1], \widetilde{v_1[i]} \geq \widetilde{v_0[i]}$, while preserving $\left|v_0[i] - \widetilde{v_0[i]}\right| < 2^{-k}$ and $\left|v_0[i] - \widetilde{v_0[i]}\right| < 2^{-k}$. Then, we deduce $\forall i \in [0, n-1], \widetilde{c_1[i]} \geq 0$. Hence, all the numbers used are positive, and there is no need to store or compute sign bits. So, the design fits in a $\left(2 \cdot \frac{2^{m-3} + 1 - 2(m \bmod 2)}{3} + (p - m) + 2\right) \times k$ rectangle.

We can also force $\forall i \in [0, n-2], \widetilde{v_1[i]} = \widetilde{v_0[i+1]}$. Then we can prove that the computed function $\widetilde{f}$ is also increasing, which is an interesting property.

6.5 Implementation

First, I had to design lookup tables. Lookup tables whose index is 5-bit wide or narrower were straightforward, thanks to the internal structure of the FPGA used. Building larger lookup tables was a little more annoying, as multiplexers were required, and needed to be connected correctly.

Then, implementing this piecewise affine approximator was not really difficult, by using designs for the multiplier and the lookup tables.

I tried to use the inheritance offered by C++ as much as possible. So I defined 4 classes:

- `SimiliPoly` that computes polynomials (whose coefficients are given as inputs and may vary),
- `PiecewisePoly` that computes a piecewise polynomial function,
- `PiecewiseAffineApprox` that computes the piecewise affine approximation of a given function,
- `PiecewiseAffineLog` that computes the natural logarithm, using this method, with an error smaller than 1 u.l.p.

Hence, all the classes can be re-used. For example, one could re-use `PiecewiseAffineApprox` to compute sines, cosines, exponentials, square roots, or almost any other function, or could re-use `PiecewisePoly` to build higher-degree piecewise polynomial approximations, or `SimiliPoly` to compute a polynomial (with constant coefficients).

6.6 Comparison with other schemes

We may now want to compare this design with a few classic designs, such as a full lookup table (all the values are stored) and a CORDIC unit (which is a "shift-and-add" method), for different data widths.

data width	full lookup	piecewise affine approximation	CORDIC unit
4	$1 \times 4 = 4$	$5 \times 6 = 30$	$4 \times 9 = 36$
8	$5 \times 8 = 40$	$7 \times 10 = 70$	$8 \times 15 = 120$
12	$85 \times 12 = 1020$	$13 \times 14 = 182$	$12 \times 21 = 252$
16	$1365 \times 16 = 21840$	$31 \times 18 = 558$	$16 \times 27 = 432$
24	$349525 \times 24 = 8388600$	$355 \times 26 = 9230$	$24 \times 39 = 936$

Table 6.2: Space comparison for several designs (in CLBs)

data width	full lookup	piecewise affine approximation	CORDIC unit
4	1	4	4
8	2	5	8
12	2	6	12
16	3	7	16
24	4	9	24

Table 6.3: Speed comparison for several designs (in clock cycles)

The results are shown in tables 6.2 and 6.3.

For all data widths, the full lookup is the fastest method. And, for widths larger than 4 bits, the new approximator is faster than the CORDIC unit.

As for size, the full lookup table is the smallest design for data widths smaller than 9. Then, for data width between 9 and 15 included, the piecewise affine approximator is the smallest design. For data widths larger than 15, the CORDIC design is the smallest.

Hence, for (very) small data widths, the full lookup is obviously the best design. Moreover, we can have errors as small as 0.5 u.l.p., if the values are pre-computed using multi-precision algorithms.

For big data width, the only reasonable choice is the CORDIC unit, since the sizes of the other designs grow exponentially with the data width.

But the FPGA that we used for our tests had a size of 28×28, which make impossible to have data widths larger than 16 (even with the CORDIC unit). For such widths, the piecewise affine approximator is a good choice.

Conclusion

During my internship, I found and corrected a few bugs in PamDC's handling of the carry chain. Thanks to Mark SHAND, my modifications should be included in the next release of PamDC.

This internship was really interesting, and I really enjoyed to learn how the FPGAs work. I also really enjoyed designing the "generic" approximator, and proving its accuracy.

As for the generic approximator, I think that there is still some work to do. I think that it should be possible to get smaller designs with higher-degree polynomial piecewise approximators. Unfortunately, it is not so easy to find a good high-degree polynomial approximation that an affine approximation.

PAM-Blox also requires much work, as well as PamDC. Moreover, getting everything to work with Xilinx's new FPGA series (the Virtex series) will require some additional work.

Unfortunately, I think that the current version of PAM-Blox have a few shortcomings that need to be corrected. Especially, it uses a lot of C++ TEMPLATEs. Even if these TEMPLATEs look like a good feature, it is a trap: every parameter must be known at compile time, so it is impossible to compute a parameter at run time. Moreover, these TEMPLATEs seem to be very difficult to compile, because of the way C and C++ are compiled. You can find more information about this difficulty in [9].

Another shortcoming is the way a class in PAM-Blox behaves. There are three methods that should be defined for every unit. The first method is the class constructor, which also connects the input of the design. The second method, called OUT, connects the output of the hardware, and builds it. The third method, called PLACE places the unit in the FPGA. I think that it would be much more convenient if the constructor, the connecting method, the building method and the placement method could be separate.

Despite these two little shortcomings, working on PAM-Blox was really interesting, and I hope that PAM-Blox will have a good future.

Appendices

Appendix A

Internals of Xilinx XC4000 series FPGAs

The schematics in this appendix where extracted from [2], and describe the internals of Xilinx XC4000 series FPGAs.

Figure A.1 shows the internals of the CLBs. The hardware for the carry chain, and to use the function generators as RAM are not shown.

On figure A.2 you can see how the dedicated carry chain is connected with the standard function generators; on figure A.3, you can see how the dedicated carry chain really works.

Last, on figure A.4, you can see the wires and connections available in each CLB. The carry chain does not use these routing resources. It is wired vertically either down to up or both ways.

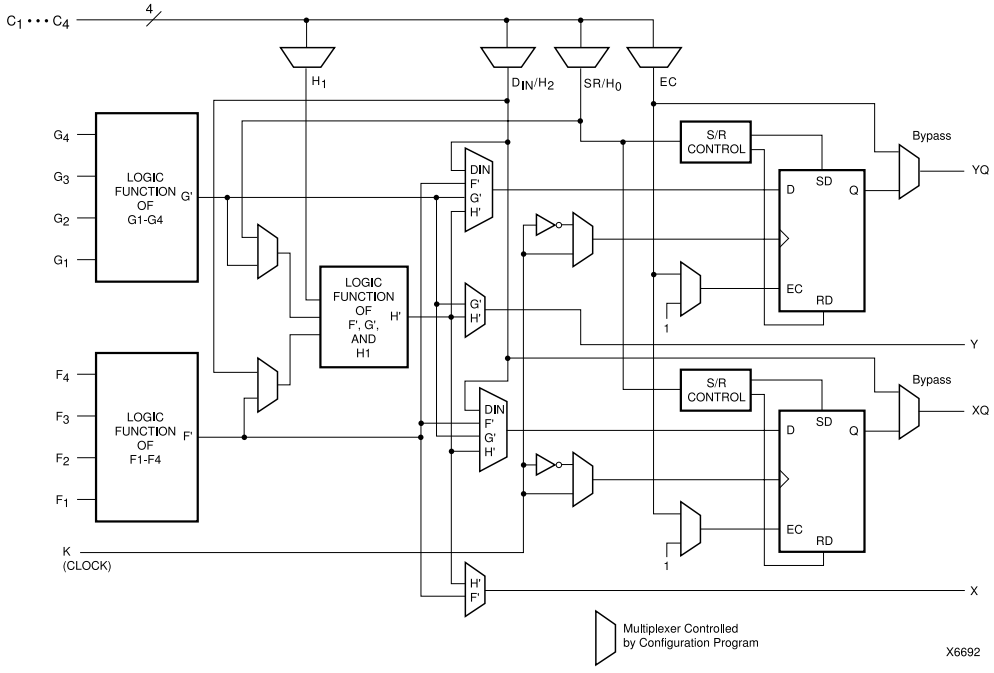


Figure A.1: A (slightly) simplified block diagram of XC4000 series CLB

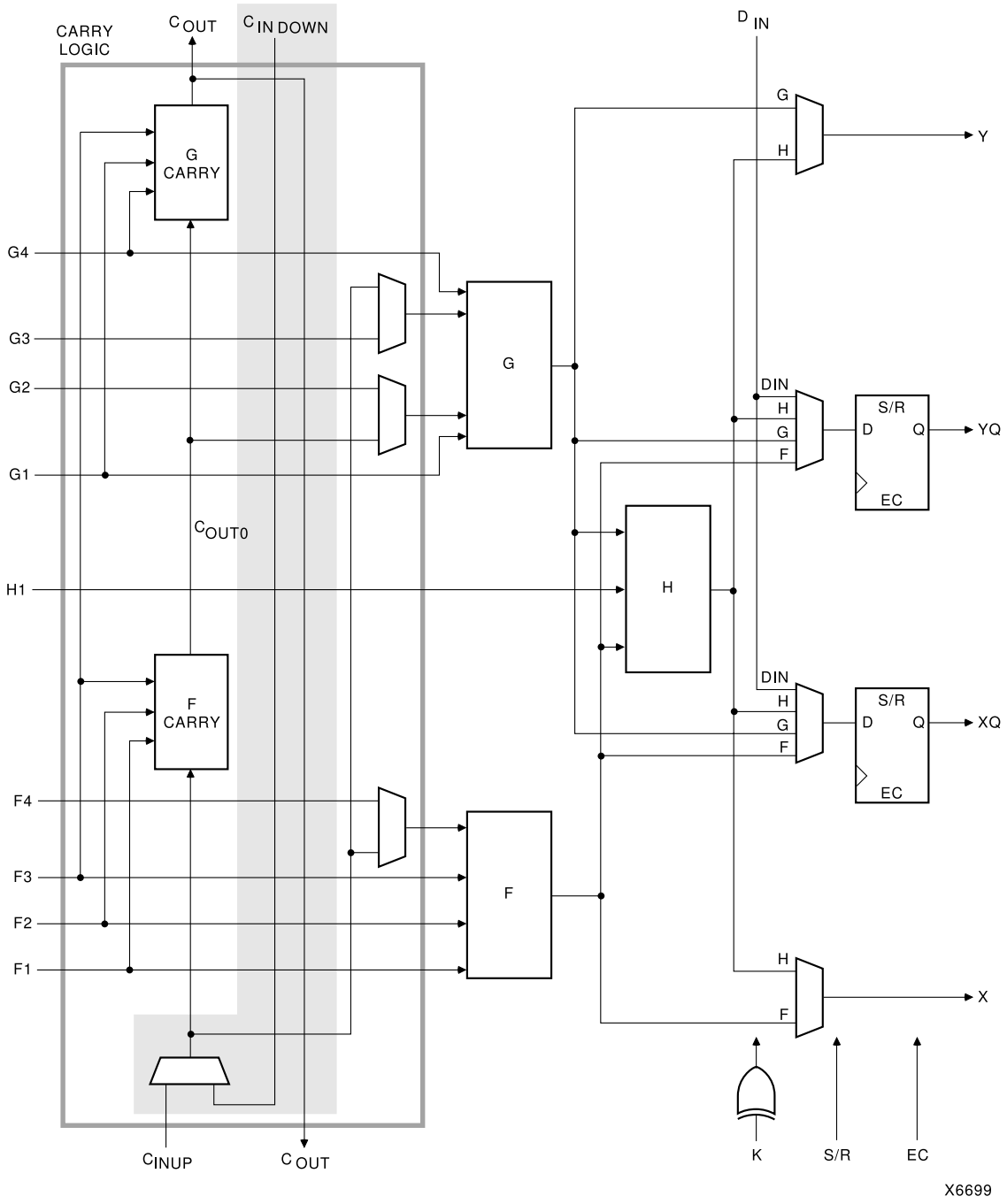


Figure A.2: Connections between the carry logic and the function generators (the shaded area is available on XC4000E only)

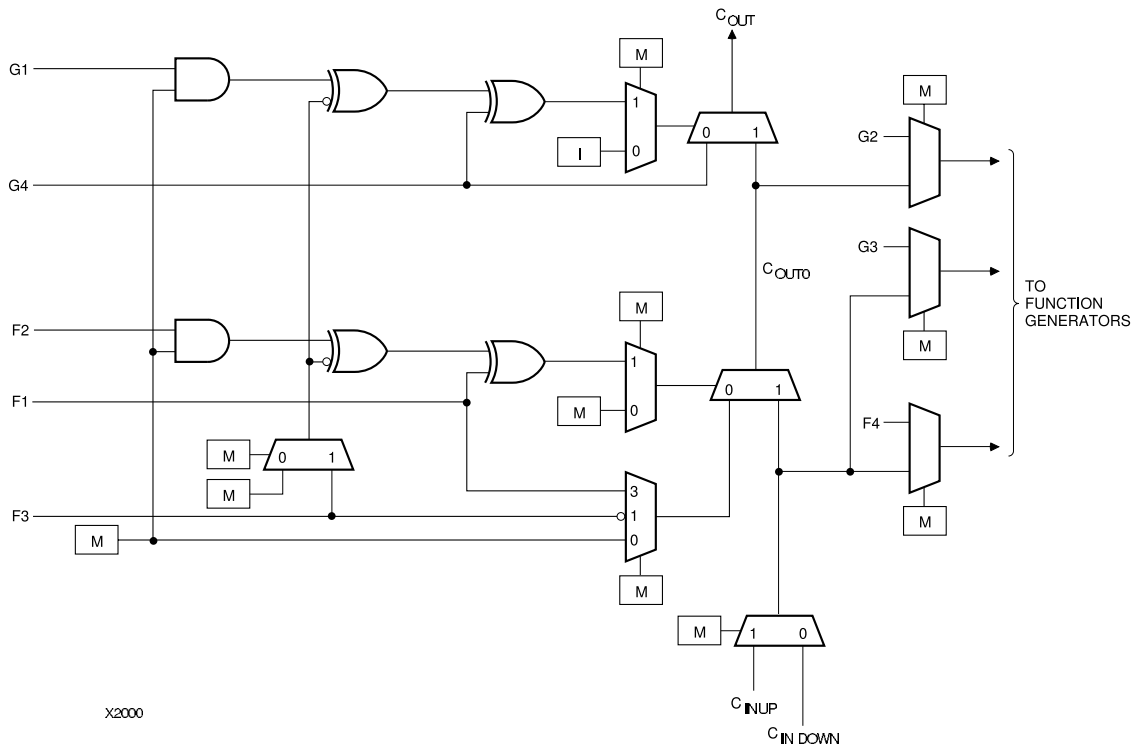


Figure A.3: Internal diagram of the carry logic

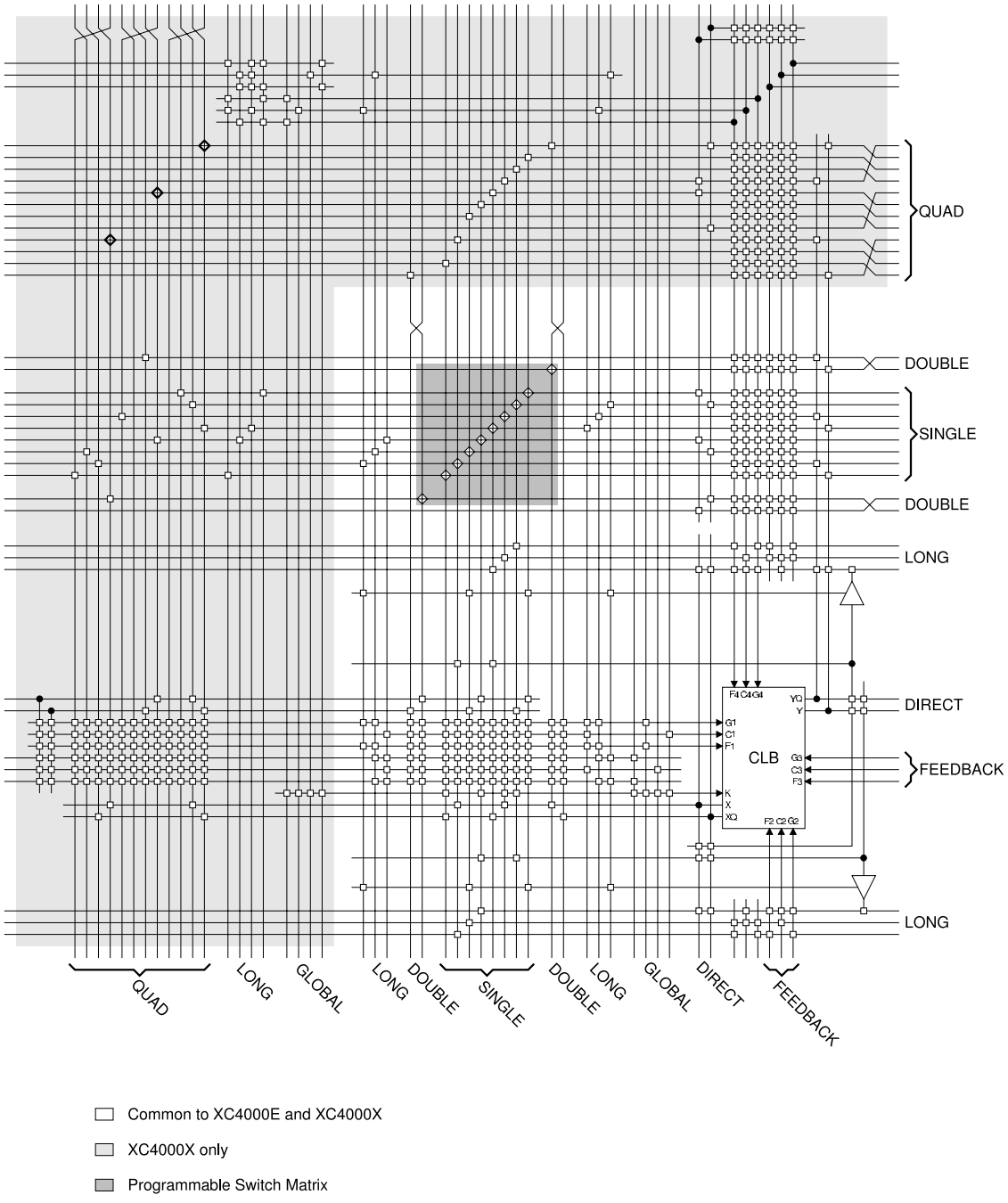


Figure A.4: Wires and connections between the CLBs

Appendix B

Accuracy of the generic approximator

This appendix proves the accuracy of the approximator presented in chapter 6 (page 21). All the notations used here were explained in that chapter.

B.1 Method error

In this section, we try to find a bound to the "method" error $|F(x) - f(x)|$.

Let us remember the definition of f:

$$f(x) = y \cdot F\left(\frac{i+1}{n}\right) + (1-y)F\left(\frac{i}{n}\right)$$

Provided the definition of y, we have:

$$f(x) = (n \cdot x - i)\, F\left(\frac{i+1}{n}\right) + (i+1-n \cdot x)\, F\left(\frac{i}{n}\right)$$

Let $\varepsilon(x) = F(x) - f(x)$. So, we have:

$$\varepsilon(x) = F(x) - (n \cdot x - i)\, F\left(\frac{i+1}{n}\right) - (i+1-n \cdot x)\, F\left(\frac{i}{n}\right)$$

Then we can compute:

$$\begin{aligned}
\varepsilon(x) &= F(x) - (n \cdot x - i)\, F\left(\frac{i+1}{n}\right) - (i+1-n \cdot x)\, F\left(\frac{i}{n}\right) \\
&= (n \cdot x - i)\, F(x) + (i+1-n \cdot x)\, F(x) \\
&\quad - (n \cdot x - i)\, F\left(\frac{i+1}{n}\right) - (i+1-n \cdot x)\, F\left(\frac{i}{n}\right) \\
&= (n \cdot x - i)\left(F(x) - F\left(\frac{i+1}{n}\right)\right) + (i+1-n \cdot x)\left(F(x) - F\left(\frac{i}{n}\right)\right) \\
&= (i+1-n \cdot x)\int_{\frac{i}{n}}^{x} F'(y)\,\mathrm{d}y - (n \cdot x - i)\int_{x}^{\frac{i+1}{n}} F'(y)\,\mathrm{d}y \\
&= n\left(\frac{i+1}{n} - x\right)\int_{\frac{i}{n}}^{x} F'(y)\,\mathrm{d}y + n\left(x - \frac{i}{n}\right)\int_{x}^{\frac{i+1}{n}} -F'(y)\,\mathrm{d}y
\end{aligned}$$

We can now compute separately each of these two integrals:

$$\begin{aligned}
\int_{\frac{i}{n}}^{x} F'(y)\ \mathrm{d}y &= \int_{\frac{i}{n}}^{x} 1 \cdot F'(y)\ \mathrm{d}y \\
&= \left[\left(y - \frac{i}{n}\right) F'(y)\right]_{\frac{i}{n}}^{x} - \int_{\frac{i}{n}}^{x} \left(y - \frac{i}{n}\right) F''(y)\ \mathrm{d}y \\
&= \left(x - \frac{i}{n}\right) F'(x) - \int_{\frac{i}{n}}^{x} \left(y - \frac{i}{n}\right) F''(y)\ \mathrm{d}y
\end{aligned}$$

$$\begin{aligned}
\int_{x}^{\frac{i+1}{n}} -F'(y)\ \mathrm{d}y &= \int_{x}^{\frac{i+1}{n}} -1 \cdot F'(y)\ \mathrm{d}y \\
&= \left[\left(\frac{i+1}{n} - y\right) F'(y)\right]_{x}^{\frac{i+1}{n}} - \int_{x}^{\frac{i+1}{n}} \left(\frac{i+1}{n} - y\right) F''(y)\ \mathrm{d}y \\
&= -\left(\frac{i+1}{n} - x\right) F'(x) - \int_{x}^{\frac{i+1}{n}} \left(\frac{i+1}{n} - y\right) F''(y)\ \mathrm{d}y
\end{aligned}$$

So we have:

$$\begin{aligned}
\varepsilon(x) &= n\left(\frac{i+1}{n} - x\right)\left(\left(x - \frac{i}{n}\right) F'(x) - \int_{\frac{i}{n}}^{x} \left(y - \frac{i}{n}\right) F''(y)\ \mathrm{d}y\right) \\
&\quad + n\left(x - \frac{i}{n}\right)\left(-\left(\frac{i+1}{n} - x\right) F'(x) - \int_{x}^{\frac{i+1}{n}} \left(\frac{i+1}{n} - y\right) F''(y)\ \mathrm{d}y\right) \\
&= n\left(\frac{i+1}{n} - x\right)\left(x - \frac{i}{n}\right) F'(x) - n\left(x - \frac{i}{n}\right)\left(\frac{i+1}{n} - x\right) F'(x) \\
&\quad - n\left(\frac{i+1}{n} - x\right) \int_{\frac{i}{n}}^{x} \left(y - \frac{i}{n}\right) F''(y)\ \mathrm{d}y - n\left(x - \frac{i}{n}\right) \int_{x}^{\frac{i+1}{n}} \left(\frac{i+1}{n} - y\right) F''(y)\ \mathrm{d}y \\
&= -n\left(\frac{i+1}{n} - x\right) \int_{\frac{i}{n}}^{x} \left(y - \frac{i}{n}\right) F''(y)\ \mathrm{d}y - n\left(x - \frac{i}{n}\right) \int_{x}^{\frac{i+1}{n}} \left(\frac{i+1}{n} - y\right) F''(y)\ \mathrm{d}y
\end{aligned}$$

Now, let η_2 be the value $\sup_{z \in [0,1]} \left|F''(z)\right|$. Hence, $\forall z \in [0,1], |F''(z)| \leq \eta_2$.

Moreover, $i = \lfloor n \cdot x \rfloor$, so $i \leq n \cdot x < i + 1$, and $\frac{i}{n} \leq x < \frac{i+1}{n}$. Hence, we have $x - \frac{i}{n} \geq 0$ and $\frac{i+1}{n} - x \geq 0$.

Hence, we can compute a bound of $\varepsilon(x)$:

$$\begin{aligned}
|\varepsilon(x)| &\leq n\left(\frac{i+1}{n}-x\right)\left|\int_{\frac{i}{n}}^{x}\left(y-\frac{i}{n}\right)F''(y)\,\mathrm{d}y\right| \\
&\quad +n\left(x-\frac{i}{n}\right)\left|\int_{x}^{\frac{i+1}{n}}\left(\frac{i+1}{n}-y\right)F''(y)\,\mathrm{d}y\right| \\
&\leq n\left(\frac{i+1}{n}-x\right)\int_{\frac{i}{n}}^{x}\left(y-\frac{i}{n}\right)\left|F''(y)\right|\,\mathrm{d}y \\
&\quad +n\left(x-\frac{i}{n}\right)\int_{x}^{\frac{i+1}{n}}\left(\frac{i+1}{n}-y\right)\left|F''(x)\right|\,\mathrm{d}y \\
&\leq n\left(\frac{i+1}{n}-x\right)\int_{\frac{i}{n}}^{x}\left(y-\frac{i}{n}\right)\cdot\eta_2\,\mathrm{d}y+n\left(x-\frac{i}{n}\right)\int_{x}^{\frac{i+1}{n}}\left(\frac{i+1}{n}-y\right)\cdot\eta_2\,\mathrm{d}y \\
&\leq n\left(\frac{i+1}{n}-x\right)\cdot\eta_2\int_{\frac{i}{n}}^{x}\left(y-\frac{i}{n}\right)\,\mathrm{d}y+n\left(x-\frac{i}{n}\right)\cdot\eta_2\int_{x}^{\frac{i+1}{n}}\left(\frac{i+1}{n}-y\right)\,\mathrm{d}y \\
&\leq n\left(\frac{i+1}{n}-x\right)\cdot\eta_2\int_{0}^{x-\frac{i}{n}}y\,\mathrm{d}y+n\left(x-\frac{i}{n}\right)\cdot\eta_2\int_{x-\frac{i+1}{n}}^{0}-y\,\mathrm{d}y \\
&\leq n\left(\frac{i+1}{n}-x\right)\cdot\eta_2\left[\frac{y^2}{2}\right]_{0}^{x-\frac{i}{n}}+n\left(x-\frac{i}{n}\right)\cdot\eta_2\left[-\frac{y^2}{2}\right]_{x-\frac{i+1}{n}}^{0} \\
&\leq \frac{n}{2}\left(\frac{i+1}{n}-x\right)\cdot\eta_2\left(x-\frac{i}{n}\right)^2+\frac{n}{2}\left(x-\frac{i}{n}\right)\cdot\eta_2\left(\frac{i+1}{n}-x\right)^2 \\
&\leq \frac{n}{2}\left(\frac{i+1}{n}-x\right)\left(x-\frac{i}{n}\right)\left(x-\frac{i}{n}+\frac{i+1}{n}-x\right)\cdot\eta_2 \\
&\leq \frac{n}{2}\left(\frac{i+1}{n}-x\right)\left(x-\frac{i}{n}\right)\frac{1}{n}\cdot\eta_2 \\
&\leq \frac{1}{2}\left(\frac{i+1}{n}-x\right)\left(x-\frac{i}{n}\right)\cdot\eta_2
\end{aligned}$$

As $\left(\frac{i+1}{n}-x\right)+\left(x-\frac{i}{n}\right)=\frac{1}{n}$, we have $\left(\frac{i+1}{n}-x\right)\left(x-\frac{i}{n}\right)\leq\left(\frac{1}{2n}\right)^2$.

This is true because, if $\alpha+\beta=S$, then $\alpha=\frac{S}{2}+\left(\alpha-\frac{S}{2}\right)$ and $\beta=S-\alpha=\frac{S}{2}-\left(\alpha-\frac{S}{2}\right)$. So, $\alpha\cdot\beta=\left(\frac{S}{2}\right)^2-\left(\alpha-\frac{S}{2}\right)^2\leq\left(\frac{S}{2}\right)^2$.

So, we have:

$$|\varepsilon(x)|\leq\frac{1}{8n^2}\cdot\eta_2$$

Hence, we have the following result:

$$\boxed{\left|F(x)-f(x)\right|\leq 2^{-(2m+3)}\sup_{z\in[0,1]}\left|F''(z)\right|}$$

B.2 Rounding error

We assume that $v_0[i]$ and $v_1[i]$ are computed accurately enough to have an error smaller than 1 u.l.p. Hence, $\left|v_0[i] - \widetilde{v_0[i]}\right| < 2^{-k}$ and $\left|v_1[i] - \widetilde{v_1[i]}\right| < 2^{-k}$. It is always possible to ensure this by using a multiprecision algorithm.

Both values $c_0[i]$ and $c_1[i]$ are computed exactly. So, $\widetilde{c_0[i]} = \widetilde{v_0[i]}$ and $\widetilde{c_1[i]} = \widetilde{v_1[i]} - \widetilde{v_0[i]}$. a is computed exactly, and then rounded to the nearest. So, $\left|\widetilde{a} - y \cdot \widetilde{c_1[i]}\right| \leq 2^{-k+1}$. The value b is also computed exactly. Hence, $\widetilde{b} = \widetilde{a} + \widetilde{c_0[i]}$. $\widetilde{f(x)}$ is $\widetilde{b}$ rounded to the nearest. Hence, $\left|\widetilde{f(x)} - \widetilde{b}\right| \leq 2^{-(l+1)}$.

Now, we can compute $f(x) - \widetilde{f(x)}$:

$$
\begin{aligned}
f(x) - \widetilde{f(x)} &= b - \widetilde{f(x)} \\
&= b - \widetilde{b} + \widetilde{b} - \widetilde{f(x)} \\
&= \left(b - \widetilde{b}\right) - \left(\widetilde{f(x)} - \widetilde{b}\right) \\
&= \left(a + c_0[i] - \widetilde{a} - \widetilde{c_0[i]}\right) - \left(\widetilde{f(x)} - \widetilde{b}\right) \\
&= \left(a - \widetilde{a} + c_0[i] - \widetilde{c_0[i]}\right) - \left(\widetilde{f(x)} - \widetilde{b}\right) \\
&= \left(y \cdot c_1[i] - y \cdot \widetilde{c_1[i]} + y \cdot \widetilde{c_1[i]} - \widetilde{a} + c_0[i] - \widetilde{c_0[i]}\right) - \left(\widetilde{f(x)} - \widetilde{b}\right) \\
&= \left(y\left(c_1[i] - \widetilde{c_1[i]}\right) - \left(\widetilde{a} - y \cdot \widetilde{c_1[i]}\right) + c_0[i] - \widetilde{c_0[i]}\right) - \left(\widetilde{f(x)} - \widetilde{b}\right) \\
&= \left(y\left(c_1[i] - \widetilde{c_1[i]}\right) + c_0[i] - \widetilde{c_0[i]}\right) - \left(\widetilde{a} - y \cdot \widetilde{c_1[i]}\right) - \left(\widetilde{f(x)} - \widetilde{b}\right) \\
&= \left(y\left(v_1[i] - v_0[i] - \widetilde{v_1[i]} + \widetilde{v_0[i]}\right) + v_0[i] - \widetilde{v_0[i]}\right) - \left(\widetilde{a} - y \cdot \widetilde{c_1[i]}\right) - \left(\widetilde{f(x)} - \widetilde{b}\right) \\
&= \left(y\left(v_1[i] - \widetilde{v_1[i]}\right) - y\left(v_0[i] - \widetilde{v_0[i]}\right) + v_0[i] - \widetilde{v_0[i]}\right) - \left(\widetilde{a} - y \cdot \widetilde{c_1[i]}\right) - \left(\widetilde{f(x)} - \widetilde{b}\right) \\
&= y\left(v_1[i] - \widetilde{v_1[i]}\right) + (1-y)\left(v_0[i] - \widetilde{v_0[i]}\right) - \left(\widetilde{a} - y \cdot \widetilde{c_1[i]}\right) - \left(\widetilde{f(x)} - \widetilde{b}\right)
\end{aligned}
$$

So, we can find a bound to the rounding error:

$$
\begin{aligned}
\left|f(x) - \widetilde{f(x)}\right| &\leq \left|y\left(v_1[i] - \widetilde{v_1[i]}\right)\right| + \left|(1-y)\left(v_0[i] - \widetilde{v_0[i]}\right)\right| + \left|\widetilde{a} - y \cdot \widetilde{c_1[i]}\right| + \left|\widetilde{f(x)} - \widetilde{b}\right| \\
&\leq y\left|v_1[i] - \widetilde{v_1[i]}\right| + (1-y)\left|v_0[i] - \widetilde{v_0[i]}\right| + \left|\widetilde{a} - y \cdot \widetilde{c_1[i]}\right| + \left|\widetilde{f(x)} - \widetilde{b}\right| \\
&< y \cdot 2^{-k} + (1-y) \cdot 2^{-k} + \left|\widetilde{a} - y \cdot \widetilde{c_1[i]}\right| + \left|\widetilde{f(x)} - \widetilde{b}\right| \\
&< 2^{-k} y + 2^{-k}(1-y) + 2^{-(k+1)} + 2^{-(l+1)} \\
&< 2^{-k} + 2^{-(k+1)} + 2^{-(l+1)} \\
&< \left(1 - \frac{1}{4}\right) 2^{-(k-1)} + 2^{-(l+1)}
\end{aligned}
$$

Hence, we have the following result:

$$
\boxed{\left|f(x) - \widetilde{f(x)}\right| < \left(1 - \frac{1}{4}\right) 2^{-(k-1)} + 2^{-(l+1)}}
$$

Bibliography

[1] Behroz Parhami,
Computer Arithmetic — Algorithms and Hardware Designs,
Oxford University Press,
2000.

[2] Xilinx,
XC4000E and XC4000X Series Field Programmable Gate Arrays,
`http://www.xilinx.com/partinfo/4000.pdf`,
1999.

[3] Bernie New,
Using the Dedicated Carry Logic in XC4000E,
`http://www.xilinx.com/xapp/xapp013.pdf`,
1996.

[4] Hervé Touati and Mark Shand,
PamDC: a C++ Library for the Simulation and Generation of Xilinx FPGA Designs,
`http://www.research.digital.com/SRC/pamette/Download.html`,
1999.

[5] Oskar Mencer, Martin Morf and Michael J. Flynn,
Pam-Blox: High Performance FPGA Design for Adaptive Computing,
IEEE Symposium on FPGAs for Custom Computing Machines (FCCM),
Napa Valley, 1998.

[6] Oskar Mencer, Heiko Hübert, Martin Morf and Michael J. Flynn,
StReAm: Object-Oriented Programming of Stream Architectures using PAM-Blox,
IEEE Symposium on FPGAs for Custom Computing Machines (FCCM),
Napa Valley, 2000.

[7] Schlomo Wasser and Michael J. Flynn,
Introduction to Arithmetic for Digital Systems Designers,
Holt, Rinehart and Winston,
1982.

[8] Jean-Michel Muller,
Elementary Functions — Algorithms and Implementation,
Birkhäuser,
1997.

[9] Steve Oualline,
Practical C++ Programming,
O'Reilly & Associates,
1997.