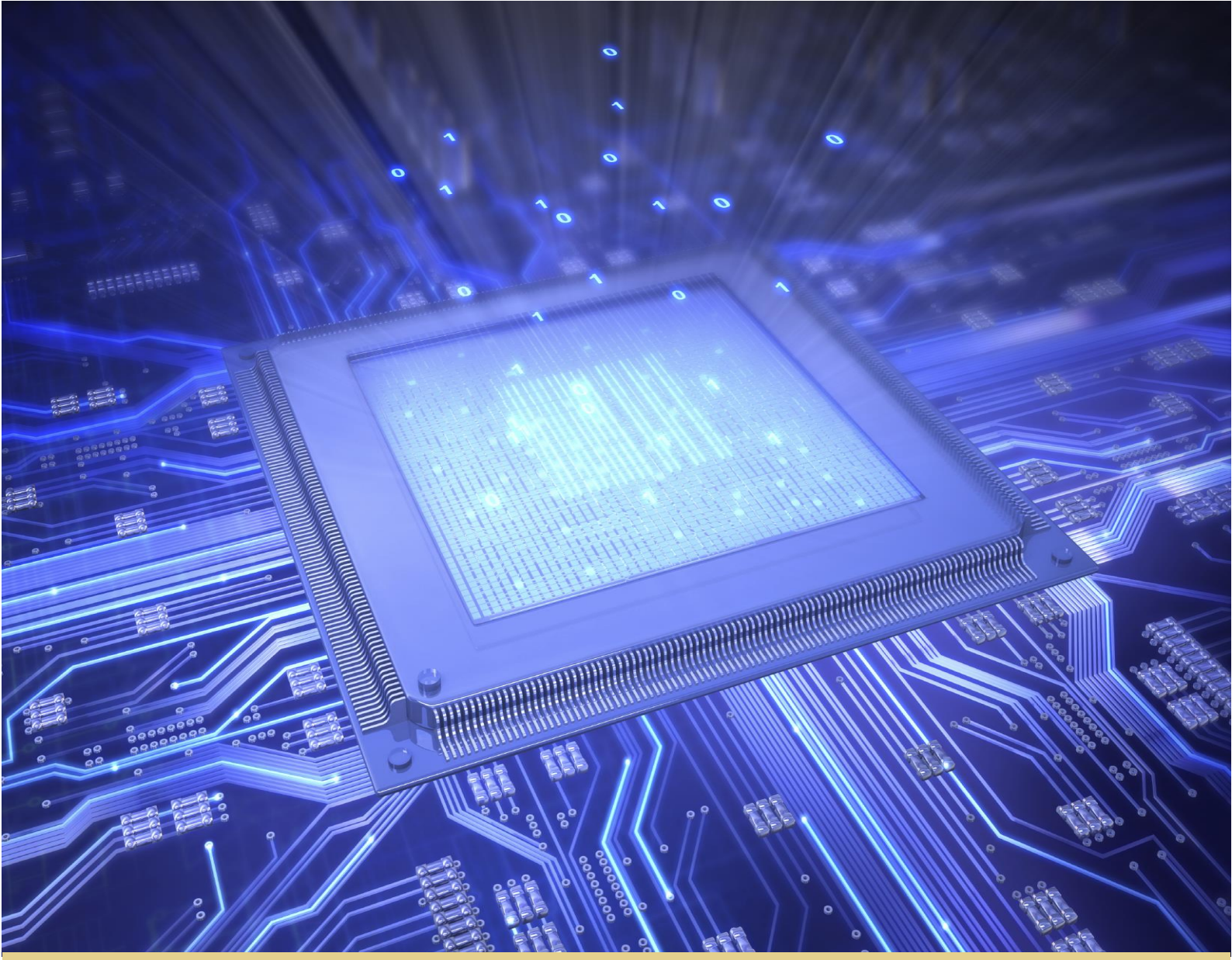


Hugo Filipe Costa Fernandes

Probabilistic Computing on FPGA with NIOS II Soft-Processor

Tese de Mestrado em Engenharia Electrotécnica e de Computadores
09/2015

· U C ·

UNIVERSIDADE DE COIMBRA

Probabilistic Computing on FPGA with NIOS II Soft-Processor

Hugo Filipe Costa Fernandes

Department of Electrical and Computer Engineering
University of Coimbra

This dissertation is submitted for the degree of
Master of Science

October 2015

Começo por agradecer ao meu orientador, Professor Jorge Lobo, pela disponibilidade constante ao longo do ano, pelo acompanhamento e pelas sugestões de trabalho. Quero também agradecer ao Rui Duarte, pelos conselhos que se provaram ser de grande valor para o meu trabalho. Sem esquecer também os meus colegas de laboratório José e Alves e Awis Aslam que ajudaram com opiniões e esclarecimento de dúvidas. Agradeço também a todos os meus amigos e família pelo apoio, em especial aos meus pais e à Catarina pela companhia e amor que sempre me acompanhou neste percurso universitário.

Abstract

This work explores ways of implementing the computations required for Bayesian inference in hardware. Probabilistic computations, such as Bayesian inference, easily over load standard Von Neumman architecture computers, leading to slow computations. To solve this, the European BAMBI FET project takes a bottom-up approach by proposing a theory and hardware implementation of probabilistic computation. Within this project a Bayesian algebra and the use of Rational Functions with Non-Negative Coefficients (RFNCs) was developed as a possible solution for Bayesian inference. The focus of this work is to map this onto computing devices using reconfigurable logic (FPGAs). We explore the implementation space available with current FPGA technology by taking advantage of the inherent parallelism and flexibility associated with these devices. Using the Nios-II soft-processor as test bed, we investigate the trade-offs of different configurations based on a Generic Bayesian Gate and the use of modified arithmetic operators which support Bayesian Algebra. Thorough tests were done concerning clock cycle, resource usage, maximum frequency and power consumption. Our results show that there are viable and interesting solutions that can be mapped to future devices dedicated to Bayesian inference.

Resumo

O trabalho aqui apresentado explora formas de implementar computação Bayesiana em hardware. Computações probabilísticas, tais como inferência Bayesiana, facilmente sobrecarregam computadores baseados em arquitetura Von Newmman, resultando na redução do seu desempenho. Para resolver este problema, o projeto europeu BAMBI FET tem uma abordagem estratificada propondo uma implementação teoria e física de computação probabilística. Inserido neste projeto, álgebra Bayesiana e o uso de funções racionais de coeficiente não negativo (RFNCs) foi desenvolvido como uma possível solução para inferência Bayesiana. O foco deste trabalho passa por mapear esta inferência para dispositivos de computação usando logica reconfigurável (FPGA). O espaço de implementação disponível com tecnologia reconfigurável atual é explorado tirando vantagem da sua flexibilidade e paralelismo associado. Usando o processador Nios II como plataforma de testes, investigámos as consequências de diferentes configurações baseadas num Operador Genérico Bayesiano e o uso de operadores aritméticos normais modificados para álgebra Bayesiana. Os nossos resultados mostram que existem soluções viáveis e interessantes que podem ser usadas em dispositivos futuros, dedicados a inferência Bayesiana.

Table of contents

List of figures

List of tables

Chapter 1

Introduction

1.1 Motivation

Probabilistic computations easily overload standard Von Neumman architecture computers, leading to slow computations. Reconfigurable logic devices (FPGAs) have been used in some probabilistic applications [8–10] providing significant performance gains compared to standard Von Neumman architectures. However, despite these performance gains, these solutions were purposely designed for the specific application and, so far, no general solution has been proposed. The BAMBI FET project takes a bottom-up approach into this problem in order to find a solution.

It has been proposed that the brain is a probabilist inference machine [11], this can be extended to living organisms in general. A recent study [12] suggests that at a sub-cellular level, biochemical cascades of cell signalling can perform the necessary probabilistic computations. This can be a starting point to pursue possible hardware implementations to support the computations required by Bayesian inference.

Since FPGAs provide a flexible platform to implement custom hardware, these will be used to experiment with different designs of basic computing blocks for probabilistic computing.

1.2 Objectives

This thesis focusses mainly on exploring the implementation space of Bayesian computing on reconfigurable logic devices. Using the Nios processor as a test-bed, we will test the feasibility of some Bayesian computation solutions by making a detailed analysis of the clock cycle performance, resource usage, maximum frequency and energy consumption. The

key objective is to provide insight on the best solutions to perform the required computations taking into account the problem characteristics and available resources.

1.3 Related Work

In previous works, FPGAs have already been used for Bayesian inference. In [8] a full-system prototype of a scalable Bayesian network learning algorithm is presented. A computing method based on dynamic Bayesian learning network is proposed in [13]. In [14], a probabilistic neural network (PNN) is proposed and developed to decode motor cortical ensemble recordings in rats. In [10] a binary LNS-based naïve Bayes inference engine for spam control. In [9] a Bayesian pixel-based segmentation algorithm is proposed. However, these were not generic solutions. In the BAMBI project, we follow a bottom-up approach towards a future probabilistic computer. A Bayesian algebra was developed and, in this work, we rely on a standard numeric coding to perform the computations. A preliminary work [7] addressed this approach and an implementation of a generic gate was presented but not thoroughly tested. Within BAMBI other solutions are being pursued that rely on coding values with stochastic bit streams [15, 16].

1.4 Main Contributions

Based on the Bayesian algebra, developed within the BAMBI project to address Bayesian inference problems, the main contribution of this work are:

1. Several implementations that support the required computations. These span the available solutions within reconfigurable logic.

2. Quantitative test results concerning power, clock cycle, frequency, and resource usage performance.

3. Analysis of the trade-offs and insights for the more suitable solutions.

1.5 Dissertation Outline

Following the Introduction, Chapter 2 introduces the basic concepts of the Bayesian algebra and the Generic Bayesian Gate that were the starting point for the hardware implementation. The next chapter provides basic background on reconfigurable logic and the test platform being used. Chapter 4 describes and explains the implemented designs in hardware In Chapter

5 presents quantitative performance test results and a qualitative analysis. In the final chapter, we present the conclusions and provide some insight for future work.

Chapter 2

Bayesian Algebra and Bayesian Gates

In this chapter we will present some background on Bayesian inference and how it can be formulated using odd ratios to compute the solution more efficiently. This will set the basic operations that need to be supported by the hardware implementation. The focus on this dissertation is to explore the implementation space trade-offs available with current technology.

2.1 Bayesian Inference

Living organisms have the ability to reason with uncertain and incomplete information. This probabilistic behaviour is an interesting aspect that could be explored. To replicate it, Bayesian approaches have been proposed as solutions [17].

The novel work by Droulez in [12] proposes that, at a sub-cellular level, biochemical cascades of cell signalling can perform probabilist computations. Also, Bayesian inference on subjective models is similar to the computation of Rational Functions with Non-negative Coefficients (RFNCs). Based on this, we will give a brief insight into this process.

The structure of the general subjective model considered is:

$$P(S \wedge F \wedge K) = \underbrace{P(S \wedge F)}_{prior} \times \underbrace{P(K|S \wedge F)}_{likelihood} \tag{2.1}$$

where unknown variables are divided into a set of searched S and a set of free F variables, with the set of observations being represented by K. Knowing a given set of observations k (in lower-case), the posterior distribution function over the relevant variable S is then given by:

$$P(S|k) = \frac{\sum_F P(S \wedge F) \times P(k|S \wedge F)}{\sum_S \sum_F P(S \wedge F) \times P(k|S \wedge F)} \tag{2.2}$$

where the denominator on the right side is a normalization constant and $\sum P(S|k) = 1$. If the state spaces of (S) and (F) are very large, this could prove very hard to compute.

A possible solution for this is to determine the posterior distribution using odd ratios.

It is shown that the posterior distribution of the inference can be completely specified by a vector of values as odd ratios (y) and the variables used are coefficients of prior distributions (a) and input vectors (x):

$$\forall s \in \{0,...,n_s-1\}, \forall f \in \{0,...,n_F-1\} : a_{s,f} = \frac{P([S=s][F=f])}{P([S=0][F=0])} \tag{2.3}$$

$$\forall s \in \{0,...,n_s-1\}, \forall f \in \{0,...,n_F-1\} : x_{s,f} = \frac{P(k|[S=s][F=f])}{P(k|[S=0][F=0])} \tag{2.4}$$

$$\forall s \in \{0,...,n_s-1\} : y_s = \frac{P([S=s]|k)}{P([S=0]|k)} = \frac{\sum_f a_{s,f} \times x_{s,f}}{\sum_f a_{0,f} \times x_{0,f}}, \quad y_i \in [0,\infty[; \tag{2.5}$$

where the state $[S=0]$ is used as a reference or default state, $a_{s,f}$ and $a_{0,f}$ are prior distributions and, $x_{s,f}$ and $x_{0,f}$ are input vectors. These are simplified sets of equations, a more detailed explanation is presented in [12, Section 4.2].

Essentially, if these odd ratios are real finite positive numbers, the solution can be obtained with simple algebra in the form of three mathematical operations: addition, multiplication and division.

Consider a simple example from the classical "sprinkler" Bayesian network. We have three binary variables R, S, G corresponding to the predicates "it has been Raining", "the Sprinkler was turned on" and "the Grass is wet". The joint distribution may be decomposed as $P(R,S,G) = P(R) \times P(S|R) \times P(G|R,S)$. We are interested in the probability that it rained knowing that the grass is either wet or dry: $P(R|G=g)$. We know G, are searching for R and S is a free variable. In odd ratios, this becomes:

$$y_r = \frac{P([R=True]|g)}{P([R=False]|g)} = \frac{\sum_s (a_{r,s} \times x_{r,s})}{\sum_s (a_{0,s} \times x_{0,s})} = \frac{((a_{1,0} \times x_{1,0}) + (a_{1,1} \times x_{1,1}))}{(1 + a_{0,1} \times a_{0,1})} \tag{2.6}$$

where 0 and 1 are used as a compact notation for *False* and *True*, y_r is the odds of "it has been raining" knowing that the grass is wet (G=True), $a_{i,i}$ are prior distributions and $x_{i,i}$ are input vectors.

In the next section, we present the implemented algebra in more detail.

2.2 Bayesian Algebra

As shown above, when dealing with odd ratios as finite positive numbers, usual algebra can be applied to compute the solution. This could also be seen as an extension of Boolean algebra, as proposed in [18], named Bayesian algebra.

However, to be seen as an extension of Boolean algebra, special cases must be added to the regular algebra in order to take into account the *unusual* odds (0 and ∞) corresponding to the logical values True(1) and False(0).

The next set of equations show how probabilities p and odd ratios r relate to each other using a binary random variable B as an example:

$$\begin{aligned} p &= P([B=1]), && p \in [0,1] \\ 1-p &= P([B=0]) \\ r &= \frac{P([B=1])}{P([B=0])}, && r \in [0,\infty[\\ p &= \frac{r}{1+r} \\ r &= \frac{p}{1-p} \end{aligned}$$

where p is the probability of a variable B being 1. This relation between the two representations can also be seen in Table 2.1, which gives a few examples.

p	r	Symbol	Comment
1	∞	T	True
0.999	999		
0.875	7		
0.75	3		
0.5	1	U	Uniform
0	0	F	False

Table 2.1 Relation of the distributions on a binary random variable given in both probability values $p \in [0,1]$ and odd ratios $r \in [0,\infty[$.

To exemplify the relation between Boolean algebra and its proposed extension Bayesian algebra, we have Table 2.2 with the equivalent *unusual* odds represented as *False*(0) and *True*(∞).

Next, we will introduce a probabilistic gate that can implement all these operations in a single component.

	Boolean Algebra	BayesianAlgebra	
OR	$F+F=F$	$0+0=0$	Addition
	$T+T=T$	$\infty+\infty=\infty$	
	$T+F=T$	$\infty+0=\infty$	
AND	$F\times F=F$	$0\times 0=0$	Multiplication
	$T\times T=T$	$\infty\times\infty=\infty$	
	$T\times F=F$	$\infty\times 0=0$	
NOT	$not\ F=T$	$0^{-1}=\infty$	Inverse
	$not\ T=F$	$\infty^{-1}=0$	

Table 2.2 Boolean algebra and the equivalent for Bayesian algebra.

2.3 Generic Bayesian Gate

In Boolean algebra, a NAND gate is seen as an universal block that can be used to create any logic circuit. For Bayesian algebra, the equivalent is the Generic Bayesian Gate, also known as GUT[1]. It is proposed as an extension of the Boolean algebra in [18] and [19]. The symbol is shown in Figure 2.1. The output, in odd ratios, is a function of three input variables, the input order being irrelevant within each gate thanks to the commutative property:

$$g(x,y,z)=\frac{x+y+z}{1+x\times y\times z} \tag{2.7}$$

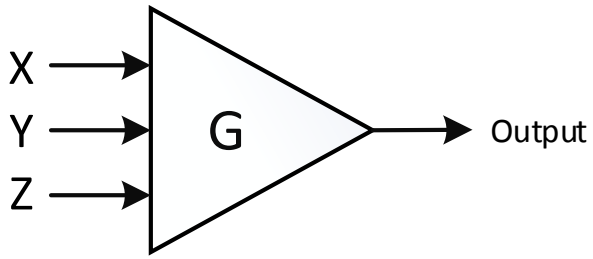


Fig. 2.1 Generic Probabilistic Gate Symbol.

The truth table for this probabilistic gate, shown in Table 2.3, demonstrates that the output can be divided into two types: Boolean and Non-Boolean. The Non-Boolean cases can be derived from the main function (Eq. 2.7) by computing its limits.

[1]The name comes from the use of the three symbols (G U T) which in addition to names of inputs and outputs can specify a basic application (False can be generated from $G(T,T,T)$).

Some of these and other limits are shown here, representing a variety of arithmetic functions:

$$g(x,y,F) = x+y \tag{2.8}$$

$$g(x,y,T) = \frac{1}{x \times y} \tag{2.9}$$

$$g(x,U,F) = \frac{1}{x} \tag{2.10}$$

$$g(x,F,F) = x \tag{2.11}$$

They can also be combined to produce other functions:

$$g(U,T,g(x,y,T)) = x \times y \tag{2.12}$$

the same function can also be represented with symbols, as shown in Figure 2.2.

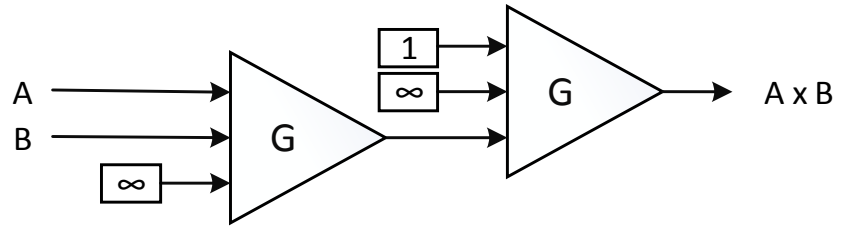


Fig. 2.2 Example of combination of two GUTs to produce a multiplication of two values (the same function is presented as Equation 2.12).

The logic of combining multiple GUTs to produce a desired output will be explained in the next section.

2.4 GUT Trees

Just like logic circuits that use NAND gates, GUTs can be used in a similar way as ternary trees that cascade into a single output. Theoretically, there is no limit for the scalability of this design. In this work, we considered two types of GUT Trees: (1) Static and (2) Dynamically Generated. In Figure 2.3 we can see an example of a Static GUT Tree where all the inputs of each gate derive from other gates with the exception of the top layer. This means that the set and arrangement of inputs defines the calculation being made, or in other words, its "software" configurable.

	X	Y	Z	Output (Odd Ratio)	Type of Operation
NAND	F	F	F	F (0)	Boolean
	F	F	T	T (∞)	
	F	T	T	T (∞)	
	T	T	T	F (0)	
	F	F	z	z	Non-Boolean
	F	T	z	T (∞)	
	T	T	z	F (0)	
	F	y	z	$y+z$	
	T	y	z	$\frac{1}{y \times z}$	
	x	y	z	$\frac{x+y+z}{1+x \times y \times x}$	

Table 2.3 GUT truth table in odd ratios, and its NAND gate equivalent (assuming (X,Y) as inputs and Z as a *True*(∞) constant). The order of the values is irrelevant, so the inputs represent all the different combinations possible. Lower-case letters represent real finite positive numbers.

The next equations show how the number of GUTs (N) and the number of inputs (M) scales with the amount of layers (L) in a Static GUT tree:

$$N = \sum_{i=0}^{L-1} 3^i \tag{2.13}$$

$$M = 3^L \tag{2.14}$$

The Dynamically Generated GUT Tree lacks the flexibility of Static GUT Trees but uses less gates. In Dynamically Generated GUT Trees inputs and constants can enter the Tree in any layer and thus avoiding the use of extra gates. An example of Dynamically Generated GUT Tree can be seen in Figure 2.4, with its corresponding function bellow (variables represented with lower case letters):

$$\text{Output} = \frac{1}{(a+b) \times (\frac{1}{c}+d)} = G(T, G(F,a,b), G(F, G(T,U,c), d)) \tag{2.15}$$

In Chapter 4 we present some of the implementation challenges of these designs that include the scaling of resources along with the respective data transfer delays.

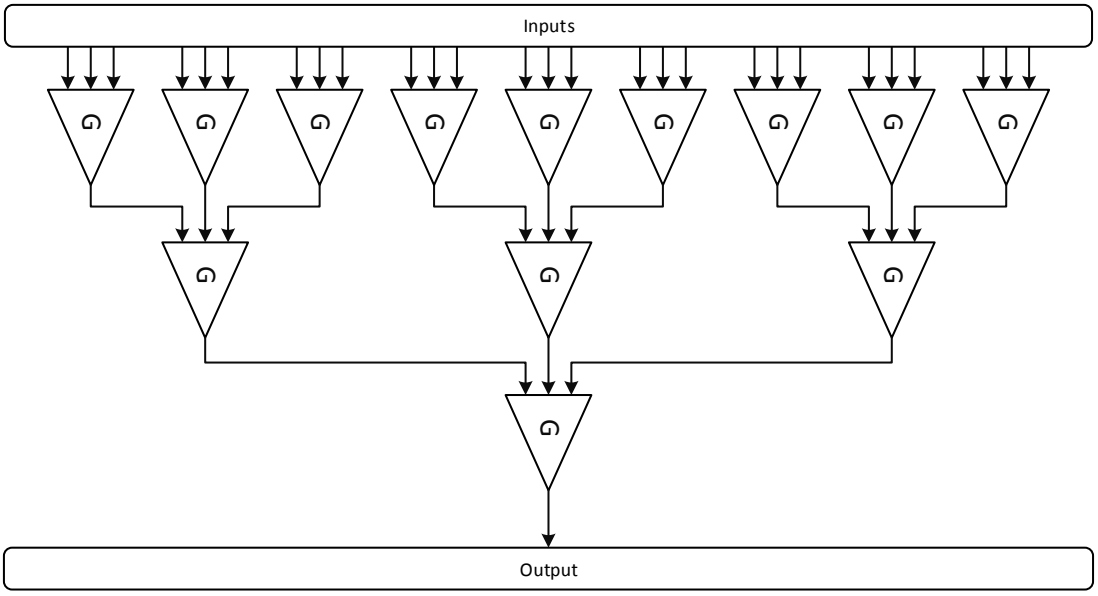


Fig. 2.3 Static GUT Tree where all the inputs are present at the top and there is only a single output

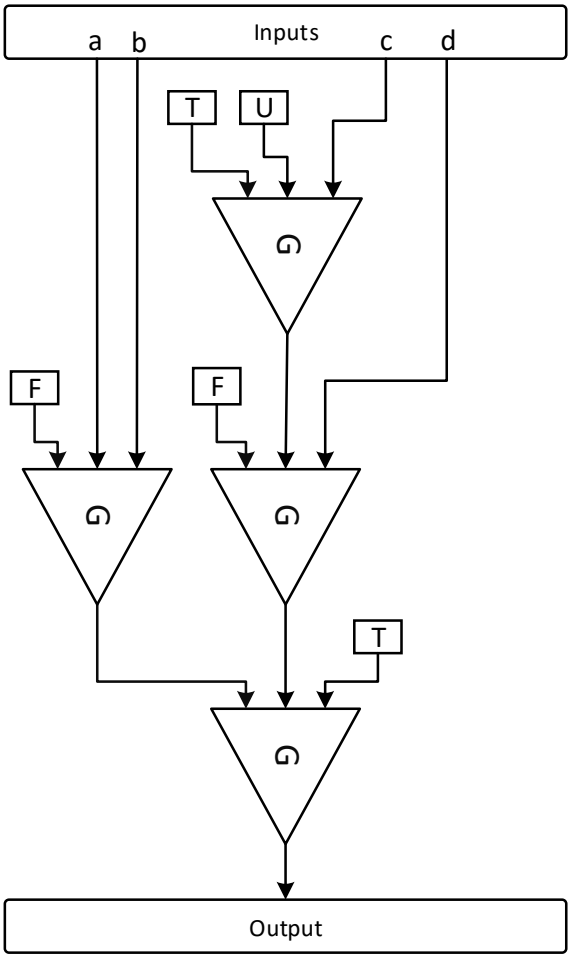


Fig. 2.4 Dynamically Generated GUT Tree where the use of constants (marked by rectangles) and inputs at any layer avoids the use of extra gates.

Chapter 3

Implementation Background

In this chapter, we present an hierarchical overview of the hardware, software and tools used. The hardware is based on the Altera Cyclone IV FPGA and is provided by Terasic as development boards. The Nios II processor, is a processor that can be integrated inside the FPGA and will be used as a test-bed, feeding inputs into different Bayesian accelerator designs. The tools are all provided with Quartus II Design Suite and accomplish different tasks like development, debugging and programming of the final designs.

3.1 Field-Programmable Gate Arrays

3.1.1 Introduction

Field-Programmable Gate Arrays (FPGAs) are devices that provide flexible implementations of digital circuits. Their flexibility comes from an array of Logic Elements (LEs) and other specialised blocks such as embedded memories and multipliers that can interconnect using routing resources. This makes FPGAs the element of choice for rapid prototyping and implementation of complete systems. Figure 3.1 shows the component layout on a FPGA device.

3.1.2 Development Boards and Device Architecture

Development boards provide power and communication support for the main chip, as well as peripherals that expand the its capabilities. In this work, the demonstration and evaluation of the design will primarily focus on Cyclone IV Architecture using both DE0-Nano and DE2-115 development boards from Terasic (Figures 3.2a and 3.2b). However, resource consumption in DE4 (Stratix IV) and DE5-Net (Stratix V) boards will also be presented.

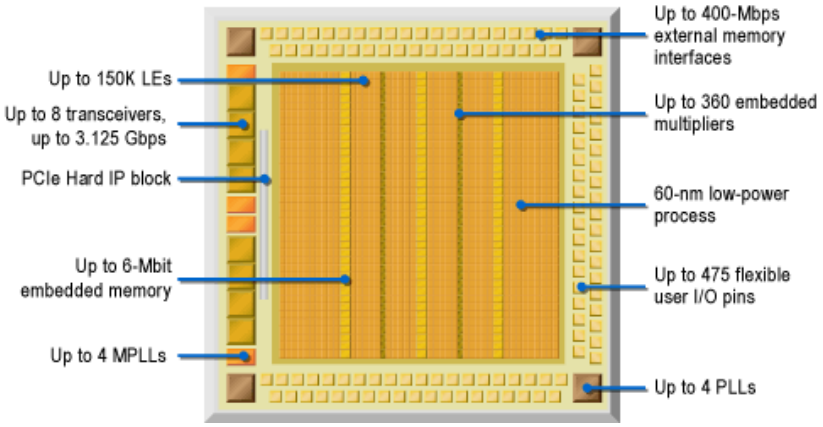
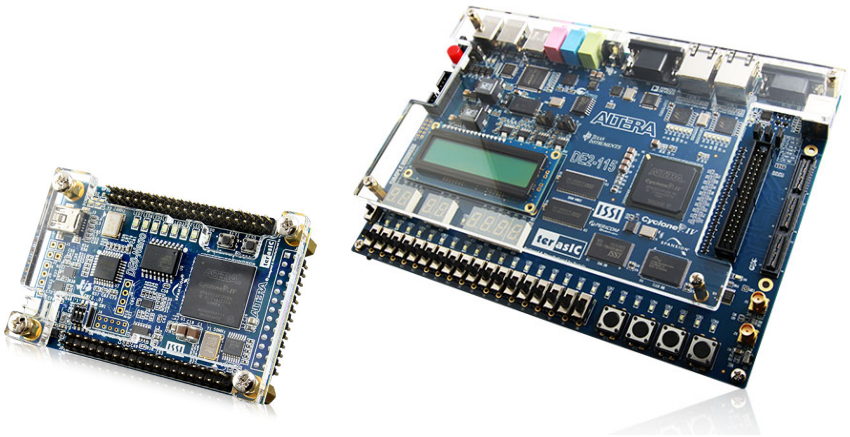


Fig. 3.1 FPGA components (source: [1]).

To give a visual idea of the logic area available in both Cyclone IV development boards, a comparison is shown in Figure 3.3.

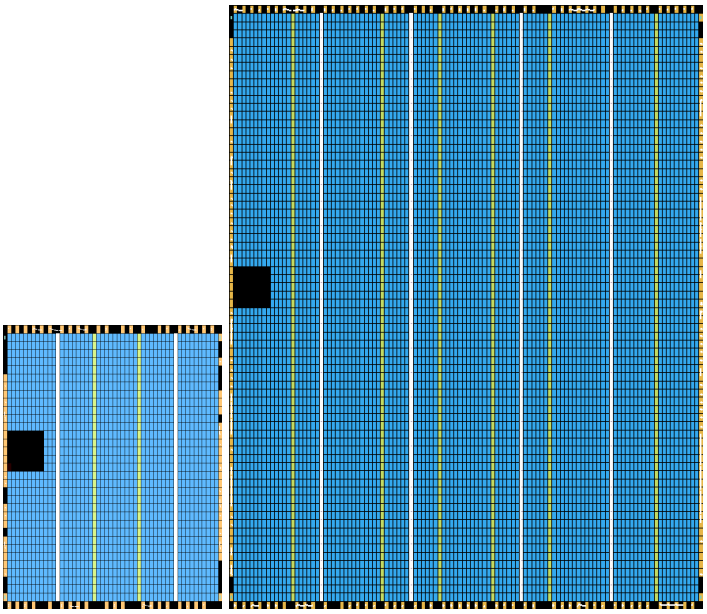


(a) DE0-Nano (b) DE2-115

Fig. 3.2 Development boards used in our work (from [2]).

3.1.3 Logic Elements and Logic Array Blocks

Logic Elements are the smallest units of logic in the Cyclone IV architecture [3, Chapter 2]. Its main components are a Look-Up Table (LUT) and a Register. Each LE has a Normal Mode (Figure 3.6a) and an Arithmetic Mode (Figure 3.6b). The first is suitable for general logic applications and combinational functions. The second is used to implement adders, counters, accumulators and comparators. The circuit of an LE is depicted in Figure 3.4. A Logic Array Block (LAB) controls groups of LEs and provides chain connections between LEs. A LAB structure is shown in Figure 3.5.



(a) EP4CE22 (b) EP4CE115

Fig. 3.3 Area comparison of the two used versions of Cyclone IV FPGA.

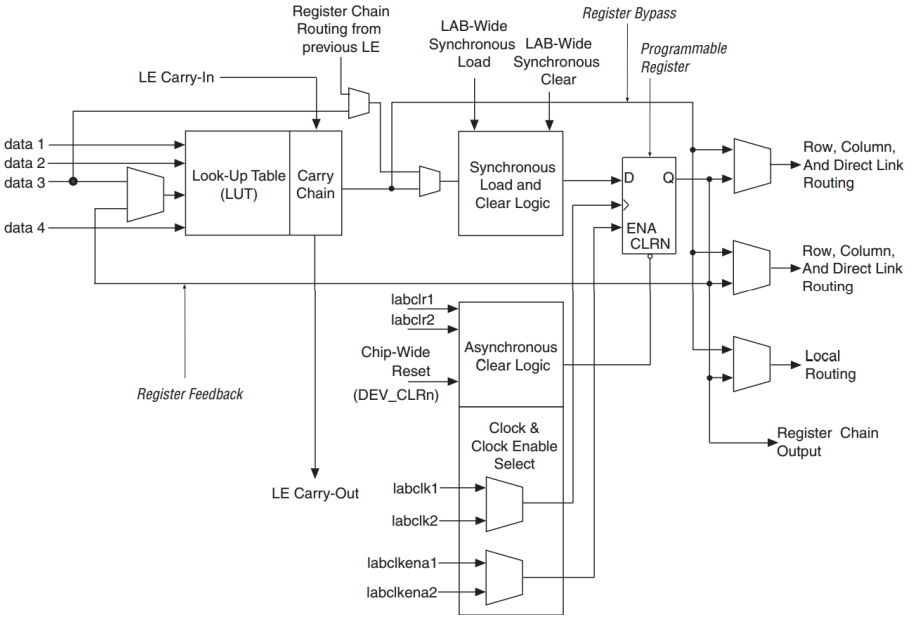


Fig. 3.4 Block diagram of a Cyclone IV LE (From [3])

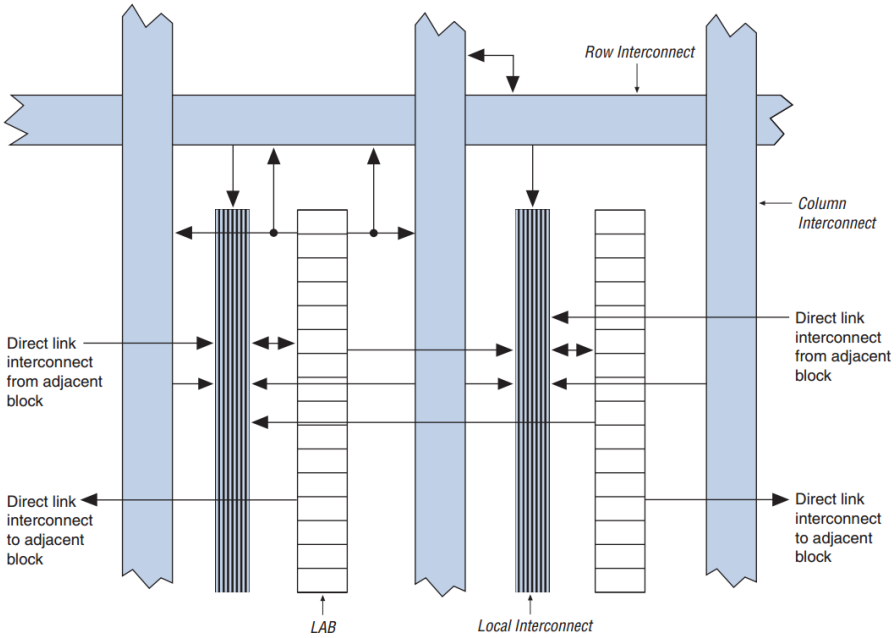


Fig. 3.5 Cyclone IV Device LAB Structure (From [3]).

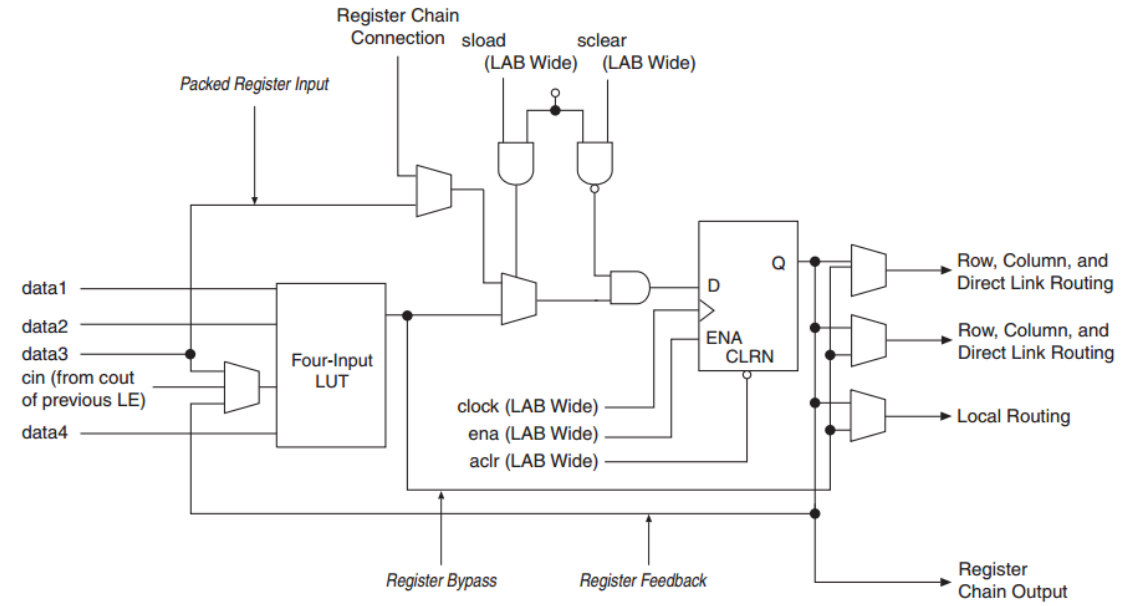
3.1.4 Embedded Memory

To address data storage needs, the FPGA device provides specialized memory blocks arranged in columns next to the LEs. These blocks can provide different functions, such as, RAM, shift registers, ROM, FIFO buffers, etc... [3, Chapter 3]. When creating embedded systems with external SDRAM memory, a good implementation of these internal memory blocks is to use them as cache memory because they boost the system performance considerably.

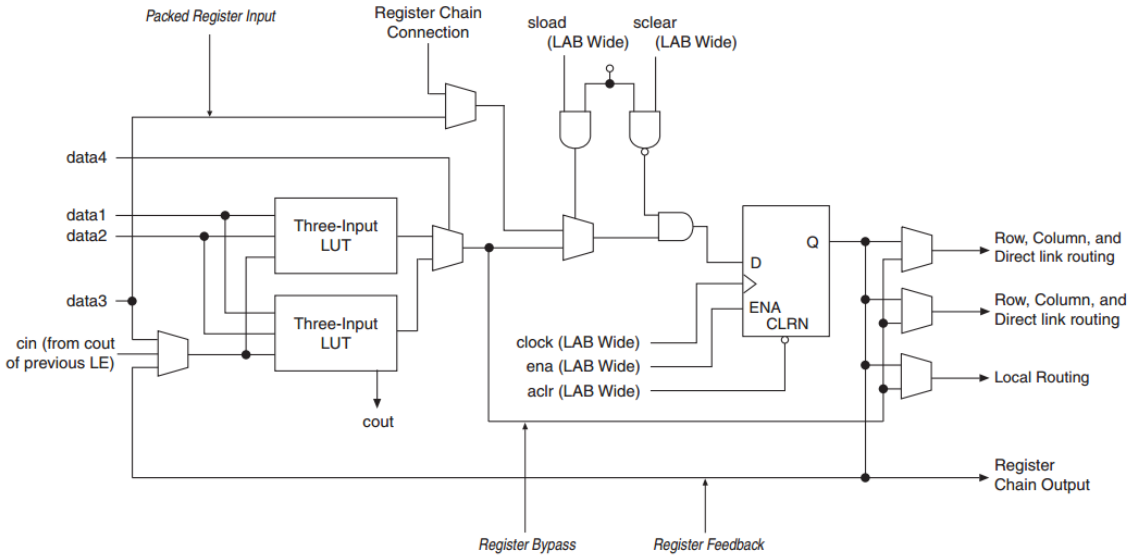
3.1.5 Embedded Multipliers

Embedded multipliers[1] are dedicated blocks used for Digital Signal Processing (DSP) [3, Chapter 4]. In Cyclone IV FPGA architectures, each embedded multiplier block is composed of two 9 x 9 Multipliers, with one multiplier stage, I/O registers and interfaces. They can operate independently or jointly as an 18 x 18 multiplier. For multiplications larger than 18 x 18, embedded blocks can be cascaded together at the cost of increased latency. The structure of an 18 x 18 bit Multiplier block is shown in Figure 3.7. Although the use of these blocks is optional, when it comes to multiplications, they are more area and power efficient than regular LE blocks. In the Stratix V architecture for example, this embedded multiplier is called an DSP block.

[1] Also known as DSP blocks.



(a) Normal Mode.



(b) Arithmetic Mode.

Fig. 3.6 The two modes of the LE (From [3]).

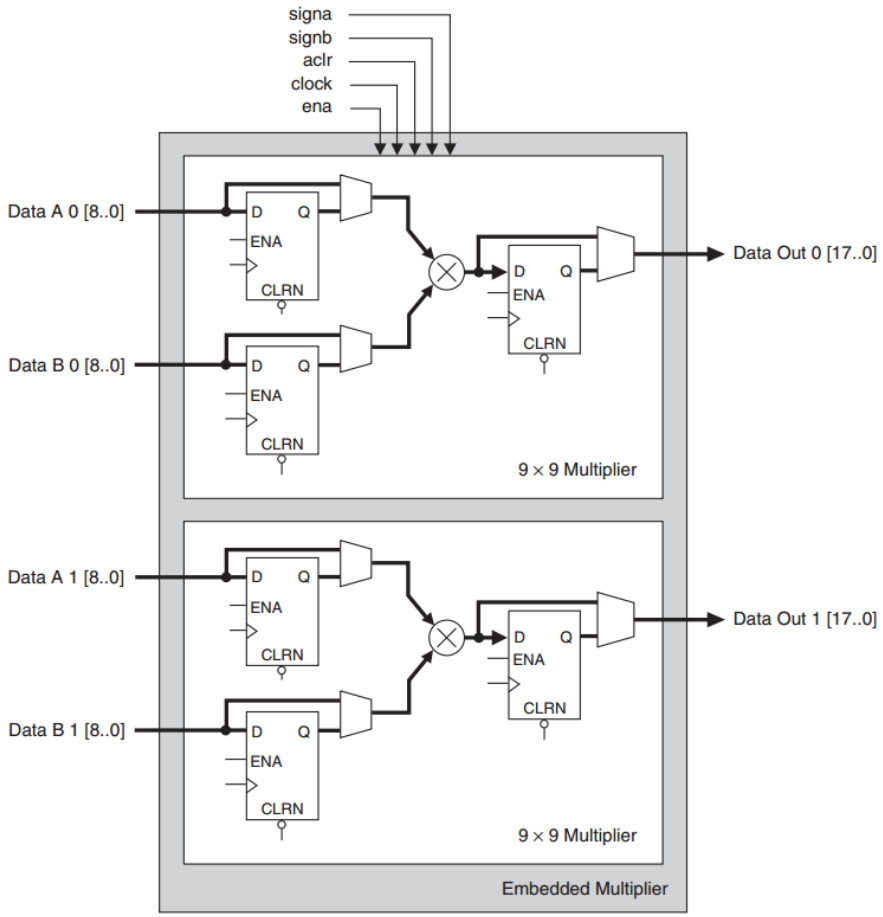


Fig. 3.7 18 x 18 bit Embedded Multiplier Block (From [3, Chapter 4]).

3.2 Nios II System

3.2.1 Introduction

The Nios II [20] is a soft processor that can be instantiated on Altera FPGA devices. It is a general-purpose RISC processor with 32-bit instruction words and datapath, it also includes integer only ALU and 32 general purpose registers. The Nios II processor and its associated components are easily instantiated by using Qsys system integration tool and Quartus II software. A set of peripherals are included and can be added to the system from the Intellectual Property (IP) Catalog in the Qsys tool. These include: timers, serial communication interfaces, general-purpose I/O, SDRAM controllers, etc. Custom peripherals can be created using hardware description languages such as VHDL or Verilog. An example processor is shown in Figure 3.8.

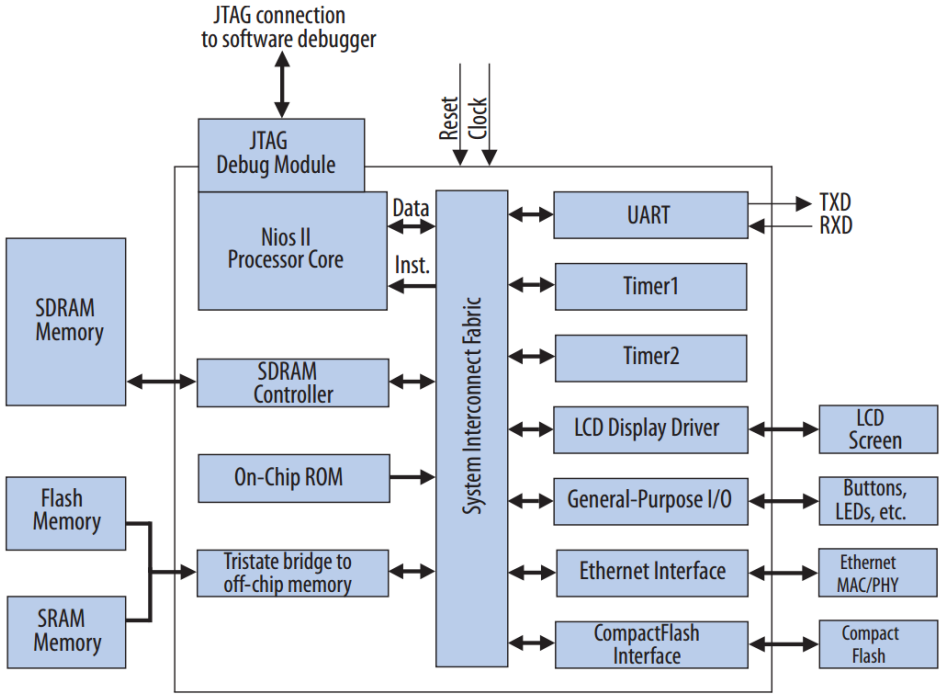


Fig. 3.8 Example of a Nios II Processor System (from [4])

3.2.2 Processor Variants

The Nios II has three core processor variants: Economy (Nios II/e), Standard (Nios II/s)[2] and Fast (Nios II/f). The cores provide different capabilities which the designer can choose from. For instance, the Economy core provides a low footprint solution for simpler applications that do not require processing power. In contrast, the Fast core is designed for high performance and is able to run a full-featured operating system. Their main characteristics are depicted in Table 3.1.

Feature	Nios II/e	Nios II/s	Nios II/f
Objective	Minimal core size	Small core size	Fast execution speed
LE	< 700	< 1400	< 2400
ALMs	< 350	< 700	< 1200
DMIPS/MHz	0.15	0.74	1.16
f_{max}	200 MHz	165 MHz	185 MHz
Cache	-	512 bytes to 64KB	512 bytes to 64KB
Pipelined Memory	-	Yes	Yes
Instruction Cache	-	Yes	Yes
Branch Prediction	-	Yes	Yes
Hardware Multiply	-	Yes	Yes
Hardware Divide	-	Yes	Yes
Barrel Shifter	-	-	Yes
Data Cache	-	-	Yes
Branch Prediction	-	-	Yes

Table 3.1 Main features of the three Nios II Processor Cores.

3.2.3 Memory Types

When creating a Nios II embedded system, memory must be added to store the program and data. From the memory types available [21, Chapter 7], the most commonly used with Nios are on-chip memory (using embedded memory blocks) and external SDRAM. The on-chip memory provides an high performance but relatively low capacity storage solution. The SDRAM however, provides high capacity at the cost of latency. The use of these two memory types can be combined in several ways to increase performance. In this work, SDRAM memory is used as the main storage for the Nios II program. Using a Direct Memory Access (DMA) interface, other components can access the memory as well.

[2]Only available in Quartus II v14.0 release and earlier.

3.2.4 Custom Instructions

The Nios II Custom Instructions (CIs) [5] are customizable logic blocks next to the Arithmetic Logic Unit (ALU) in the processor's datapath (Figure 3.9). It allows the Nios processor core to meet the needs of a particular application by converting time critical software algorithms into custom hardware logic blocks. CIs provide an easy way to experiment with hardware-software trade-offs at any point in the design process.

In our work, CIs were used to implement Bayesian gates and test different approaches. One of the problems with this implementation is that Nios II Custom Instructions can only have up to two 32 Bit inputs. To solve this issue, Direct Memory Access (DMA) was given to the CI in order to expand its capabilities(more on this topic will be presented in Chapter 4).

Figure 3.10 shows an example that illustrates the interface signals needed to run a CI. In this example of a variable clock cycle instruction, the beginning of the operation is controlled by the processor with the *start* signal and the end by the CI with the *done* signal along with the result.

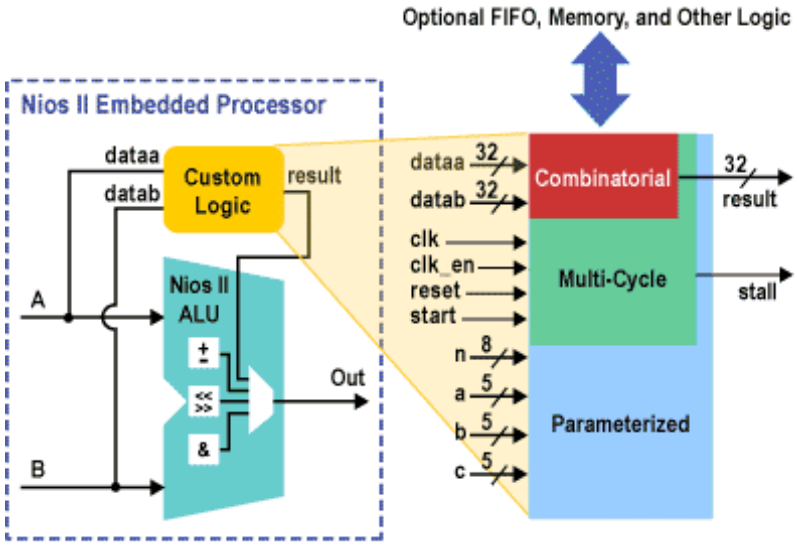


Fig. 3.9 Nios II Custom Instruction Layout (from Altera)

3.3 Software Development Tools

The tools used in this work are provided by Altera within Quartus II development environment [22]. The following list shows a brief description of them:

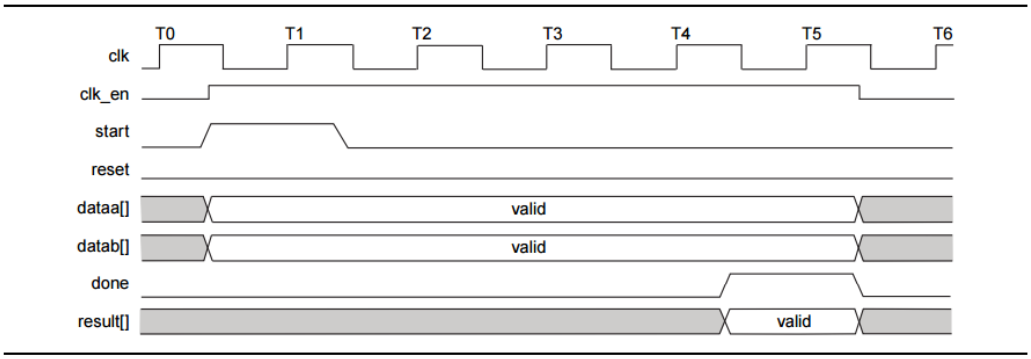


Fig. 3.10 Interface Signals for a Multi-Cycle Custom Instruction (from [5])

Quartus II

Altera Quartus II is a programmable logic device design software produced by Altera. It manages the elements of a project and enables analysis, synthesis and programming of Hardware Description Language designs into FPGA devices.

Qsys

Qsys is an integration tool that simplifies the task of defining and integrating customized IP components as subsystems of a higher-level entity. One of its main features is handling the internal data bus and control signals of an embedded system like the Nios II and its peripherals.

Eclipse

Nios II Build Tools for Eclipse is a set of plug-ins based on the Eclipse framework. It provides a consistent development platform for Nios II embedded processor systems. In this case this tool is used to create and upload C/C++ programs to the SDRAM.

SignalTap

SignalTap II is logic analyser editor that allows the designer to debug a design in real-time. SignalTap II allows the capture of specific nodes of within an FPGA to a waveform file where it can be later reviewed.

Modelsim

Modelsim is a simulation environment for Hardware Description Languages such as VHDL and Verilog. The simulation can be performed using the graphical user interface, or automatically using scripts.

PowerPlay

PowerPlay is power analyser tool that gives the designer the ability to estimate power consumption from an early design concept through design implementation. As the designer provides more details, the power estimation accuracy improves.

Chapter 4

Design Space Exploration

In this chapter we present custom instruction hardware implementations for GUTs with both Floating and Fixed Point representations, as well as, independent Bayesian operators. Trees of both Floating Point GUT and Bayesian operators were also made to test the scalability. An emulation in software that mimics the operation of a hardware based GUT was initially done to allow us to compare the performance across different processor architectures, including an i7 laptop processor.

4.1 Exploring Implementation Space

4.1.1 GUT Emulated in Software

To test the implementation of a hardware accelerator, we must first check the efficiency of a software based solution. Using C code, a small function was created to emulate the behaviour of a Generic Bayesian Gate, we can call it Soft-GUT from now on. This function was planned to compare the changes in latency of the Soft-GUT across different CPU architectures. The implementation of the function follows the GUT truth table 2.3. The pseudo-code in Algorithm 1 shows the logic.

Although it might seem simple, a few challenges arise from working with representations of infinity in floating point arithmetic. For instance, in C language, to create the floating point constant that represents True(∞) we had to use the following code:

```
const float True = 1./0; // Infinite/True number
```

Because of that, the function must pre-process the inputs to be able to distinguish between False(0) and True(∞) from other finite positive floating point numbers. The code implemented is presented in detail in Appendix B.

Algorithm 1 Soft GUT Emulation

```
1:  function SOFT_GUT(x, y, z)
2:      if x = FALSE and y = FALSE and z = FALSE then
3:          return FALSE
4:      else if x = FALSE and y = FALSE and z = TRUE then
5:          return TRUE
6:
7:          (...)
8:
9:      else if x = FALSE and y = Number and z = Number then
10:         return y + z
11:     else if x = TRUE and y = Number and z = Number then
12:         return 1/(y×z)
13:     else
14:         return (x+y+z)/(1+x×y×z)
```

4.1.2 Custom Instructions

To implement different types of Bayesian Gates using Nios II Custom Instructions as hardware accelerators, we had to create a controller that made the interface between the Nios II and the hardware acceleration block (Fig. 4.1). Another function that this controller had to execute was Direct Memory Access (DMA), this enabled the use of more that two inputs, which was a previous limitation of the CI interface signals.

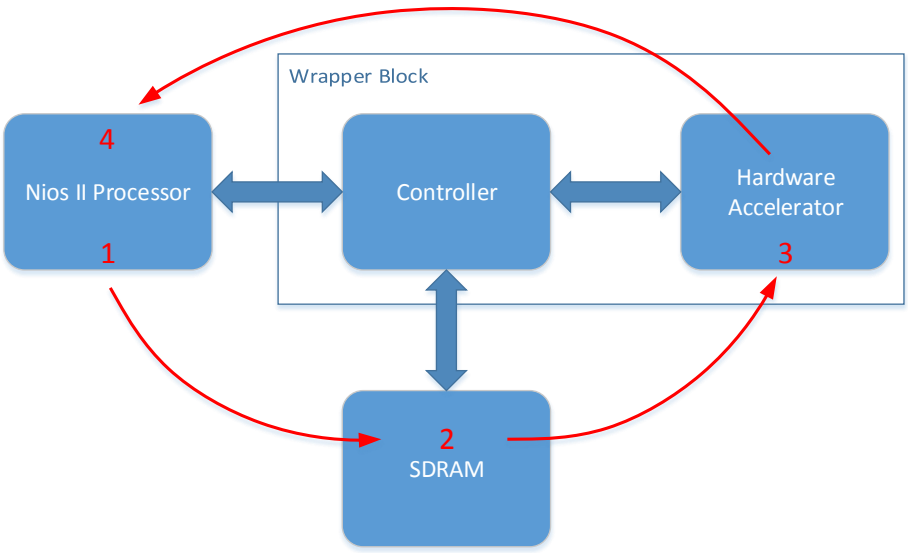


Fig. 4.1 Diagram of the CI design.

The DMA module of the controller has an Avalon Memory-Mapped Master [23] and uses Burst Transfers [6]. These can execute multiple data transfers as a unit, rather than treating every word independently. This type of transfer is best used when there is a need to read/write large blocks of data and there are some delays in the interface, which is the case of SDRAM. Templates can be found at Altera Website [23]. Figure 4.2 shows an example of a burst transfer interface.

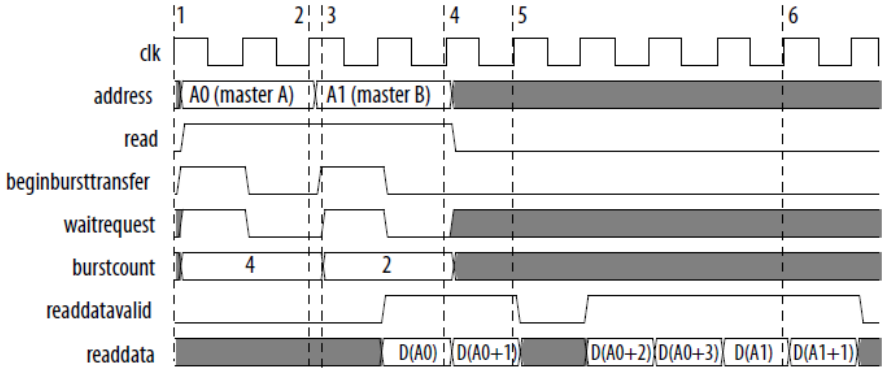


Fig. 4.2 Burst transfer example (Source: [6]).

4.2 Single Gates

4.2.1 GUT using Floating Point

From results obtained in a previous work [7], it was obvious that a major problem with this gate implementation was resource consumption. This was mainly due to the number of IP cores used for operations with floating point numbers.

This new design aimed at solving the resource problem using a dedicated controller instead. This would do two tasks: (1) the controller would give a combinational output whenever possible and (2) for the rest of the cases it would multiplex the arithmetic operations in time to save resources. Doing so, the amount of IP components dropped to just three, one for each type of arithmetic operation. Shown in Figure 4.3 is an example of time multiplexing for the GUT main function (Equation 2.7).

The Finite State Machine of the controller is shown in Figure 4.4 with the arithmetic cases in the middle. In the case of combinational[1] output the state sequence would be: $IDLE->START->DONE->IDLE$, thus avoiding any other clock delays.

[1] Although it is possible to be completely combinational, it needed a couple clock cycles due to the use of a FSM, input/output registers and CI interface.

Clock Cycle	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
Start																																
ADD		A+B								"A+B" + C								"AxBxC" + 1														
MULT		AxB							"AxB" x C																							
DIVIDE																										"A + B + C" / "ABC + 1"						
Done																																

Fig. 4.3 Example of time multiplexed arithmetic operations using Floating Point Megafunction blocks that are reused for resource optimization.

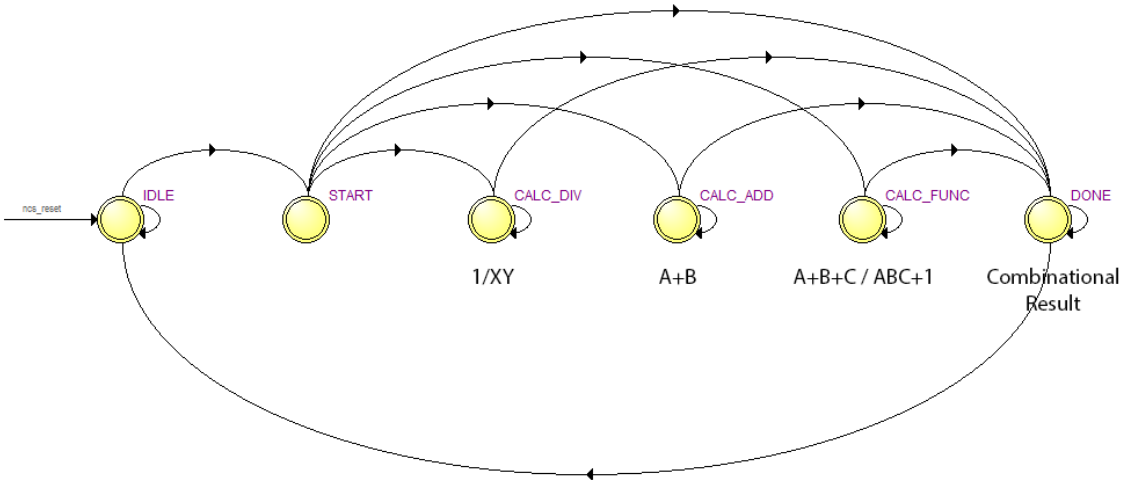


Fig. 4.4 Overview of the Finite State Machine of the controller as generated by the State Machine Viewer in Quartus II.

Depending on the state sequence, the number of clocks cycles may vary significantly. As shown in Figure 4.5, only the last set of inputs that represent $y+z$, $\frac{1}{y \times z}$ and $^{(x+y+z)}/_{(1+x \times y \times z)}$, require more than one clock cycle.

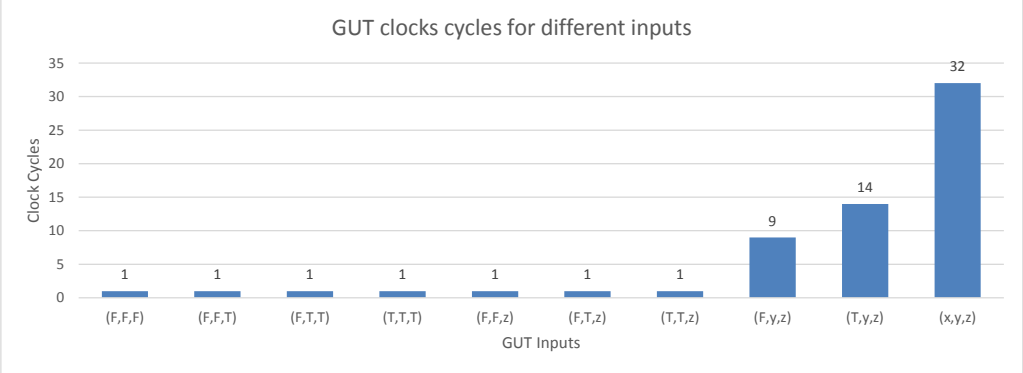


Fig. 4.5 Floating Point GUT execution clock cycles for different inputs.

The overall schematic of the GUT with the respective components is represented in Figure 4.6, showing the controller and its three floating point arithmetic blocks (Addition, Multiplication and Division).

As shown earlier, the GUT design needed DMA access to overcome the limitations of CI interface. Figure 4.7 gives an overview of the design in which the major components are represented.

4.2.2 GUT using Fixed Point

An alternative GUT design was also implemented using *probability values* instead of *odd ratios*. Because these two representations are equivalent, we implemented both in order to compare their performance in FPGAs.

Fixed Point representation has a fractional part and an integer part. To represent probability values, we can use just the fractional part of the Fixed Point representation. This allows us to maximize the resolution between the limits of probability values (0 and 1).

Similar to Integer representation, each bit in Fixed Point has a fixed value:

$$\underbrace{\underbrace{...2^3+2^2+2^1+2^0}_{\text{Integer Part}}+\underbrace{2^{-1}+2^{-2}+2^{-3}...}_{\text{Fractional Part}}}_{\text{Fixed Point Representation}}$$

The equivalent solution for the Fixed-Point variant can be derived from the main GUT function (Equation 2.7) by converting it into probability values. The final result is the

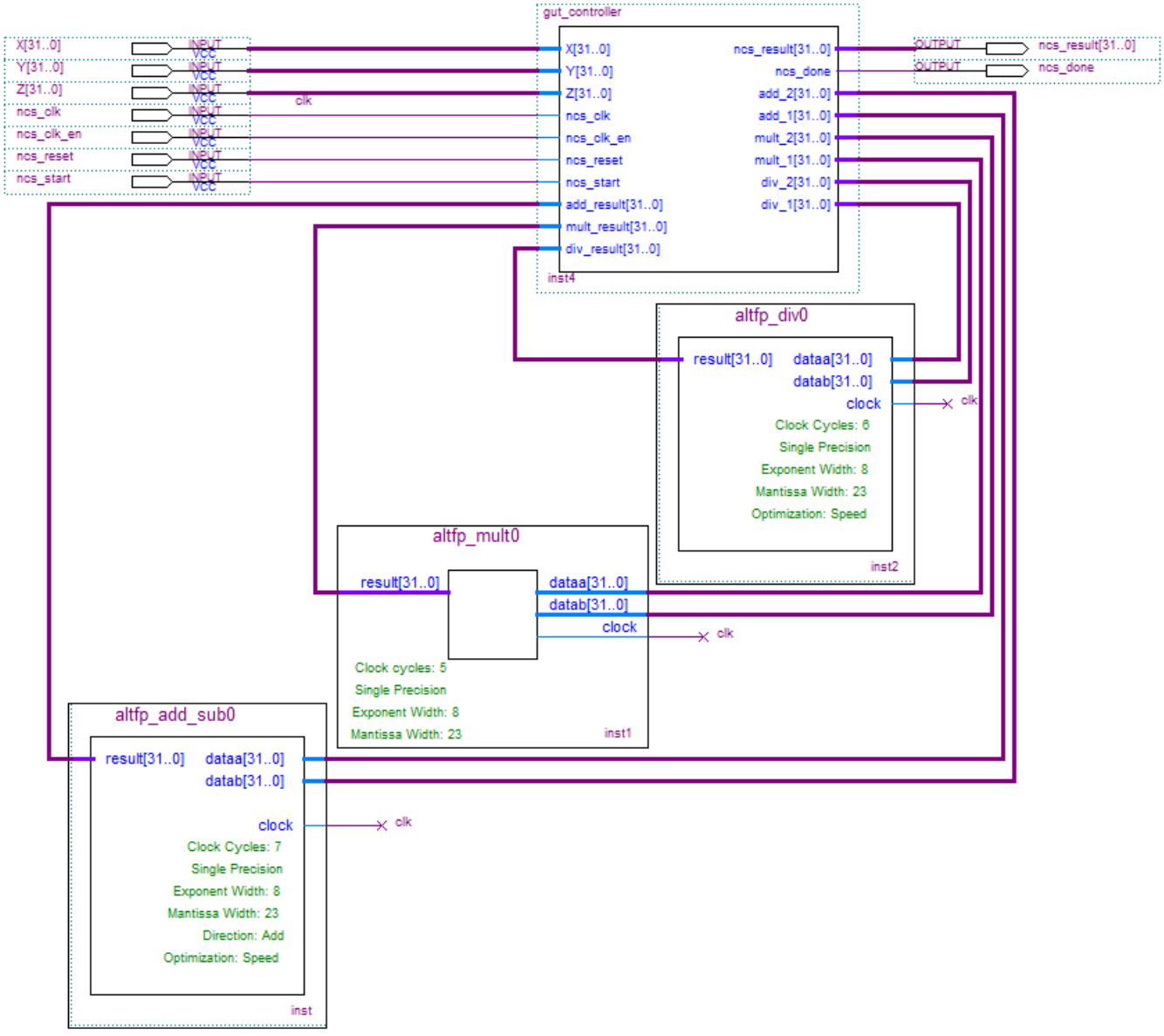


Fig. 4.6 Block Diagram/Schematic of the Floating Point GUT in Quartus II.

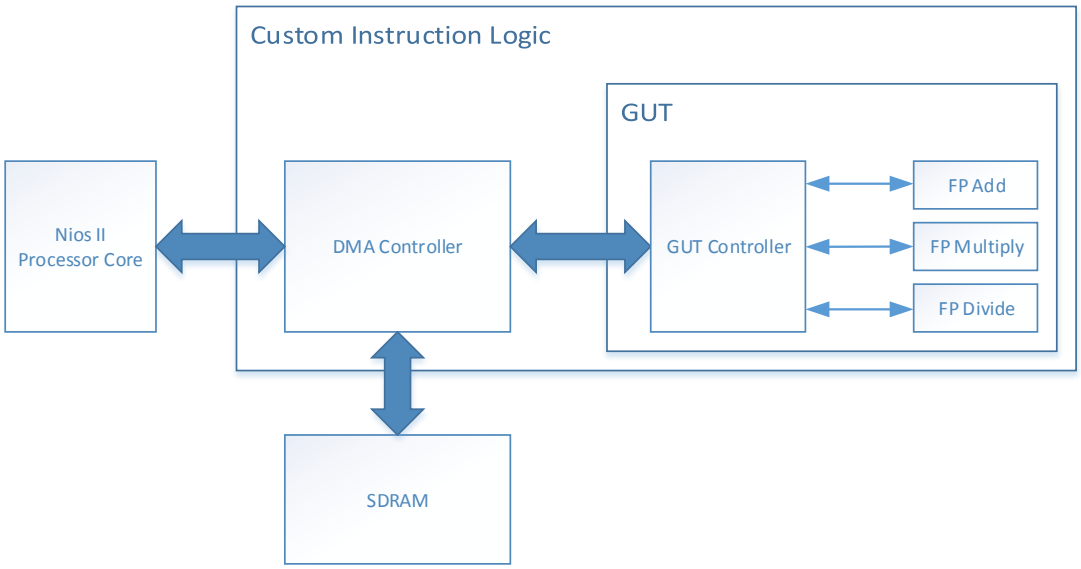


Fig. 4.7 Overview of the Floating Point GUT implemented as a Custom Instruction.

following function:

$$g'(p,q,r) = \frac{p+q+r-2(pq+qr+pr)+3pqr}{1-pq-qr-pr+3pqr} \tag{4.1}$$

this function was directly implemented using a Fixed-Point package for VHDL, which can be downloaded at [24]. The segment of code that implements this Equation is shown bellow in VHDL language (A complete code can be seen in detail in Appendix B.2.):

```
p(-1 downto -Q_SIZE) <= to_ufixed(pp(Q_SIZE-1 downto 0), -1, -Q_SIZE);
q(-1 downto -Q_SIZE) <= to_ufixed(qq(Q_SIZE-1 downto 0), -1, -Q_SIZE);
r(-1 downto -Q_SIZE) <= to_ufixed(rr(Q_SIZE-1 downto 0), -1, -Q_SIZE);
frac   <= to_slv(result(-1 downto -Q_SIZE)) when result < 1 else (
   others => '1');
-- calculations
pq <= resize( p * q, pq);
qr <= resize( q * r, qr);
pr <= resize( p * r, pr);
pqr3 <= resize( integer(3) * pq * r, pqr3);
sum_p_q_r <= resize( p + q + r, sum_p_q_r);
pq_qr_rp_2 <= resize( integer(2) * (pq + qr + pr), pq_qr_rp_2);
dividend <= resize( sum_p_q_r + pqr3 - pq_qr_rp_2   , dividend);
divisor <= resize( pqr3 + integer(1) - pq - qr - pr , divisor);
result <= resize( dividend/divisor , result);
```

Just like in integer operations, when we multiply two numbers, the number of resolution bits increases by a factor of two. This created a problem when implementing the function where the resolution of the number kept increasing. Moreover, synthesis limitations in Quartus prevented us from using more than 64 Bit resolution. To overcome this, the calculations where rounded to a fixed number of Bits after each operation (function *resize()*) and, as consequence, the output lost some precision. Because we wanted maximum resolution ranging from one to zero in decimal, the inputs were only the fraction part of the representation.

The resolution chosen was 16 Bits, this allowed the use of the default CI interface instead of DMA because it has a total of 64 Bits for inputs. An input of *0xFFFF* was considered True(1). Also, because there were no registers in this implementation we selected a Combinational type of CI (Figure 3.9) , which is the simplest.

4.2.3 Floating Point Add, Multiply and Divide with Special Cases

Another possible implementation of Bayesian Algebra, using Floating Point representation, could be the use of Addition, Multiplication and Division operators from common algebra as an alternative Arithmetic Logic Unit (ALU). Using Floating Point IP Cores from Altera connected with controller blocks for exception handling, we were able to create simple components that implemented Bayesian Algebra. In Figure 4.8 we have an example of Bayesian Algebra addition, or *BA_ADD* for short notation.

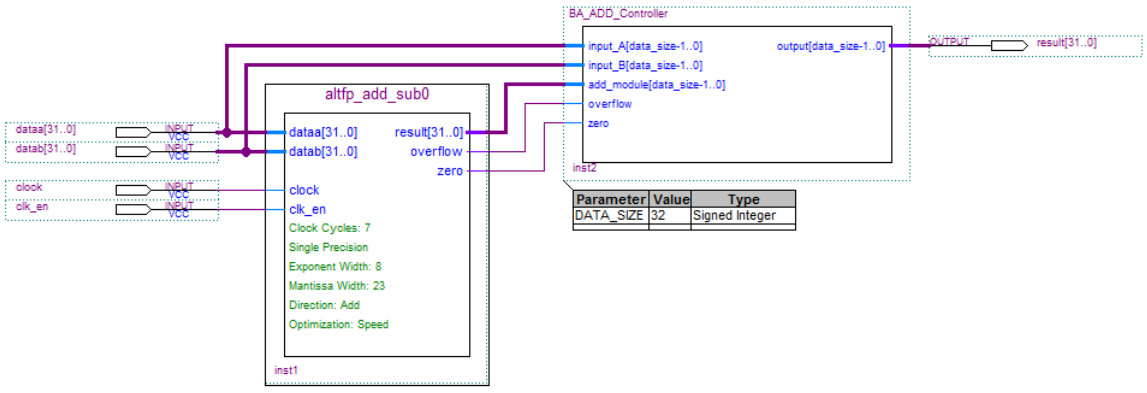


Fig. 4.8 Block schematic diagram of the Bayesian Algebra Addition operator with special cases implemented by the controller.

Joining gates for addition, multiplication, and division into a custom instruction creates an the alternative ALU dedicated to Bayesian Algebra. This design can be seen in Figure 4.9.

The Controller selects the Bayesian operation and waits a given number of clock cycles, which depends on the type of operation. When the operation is complete, the controller

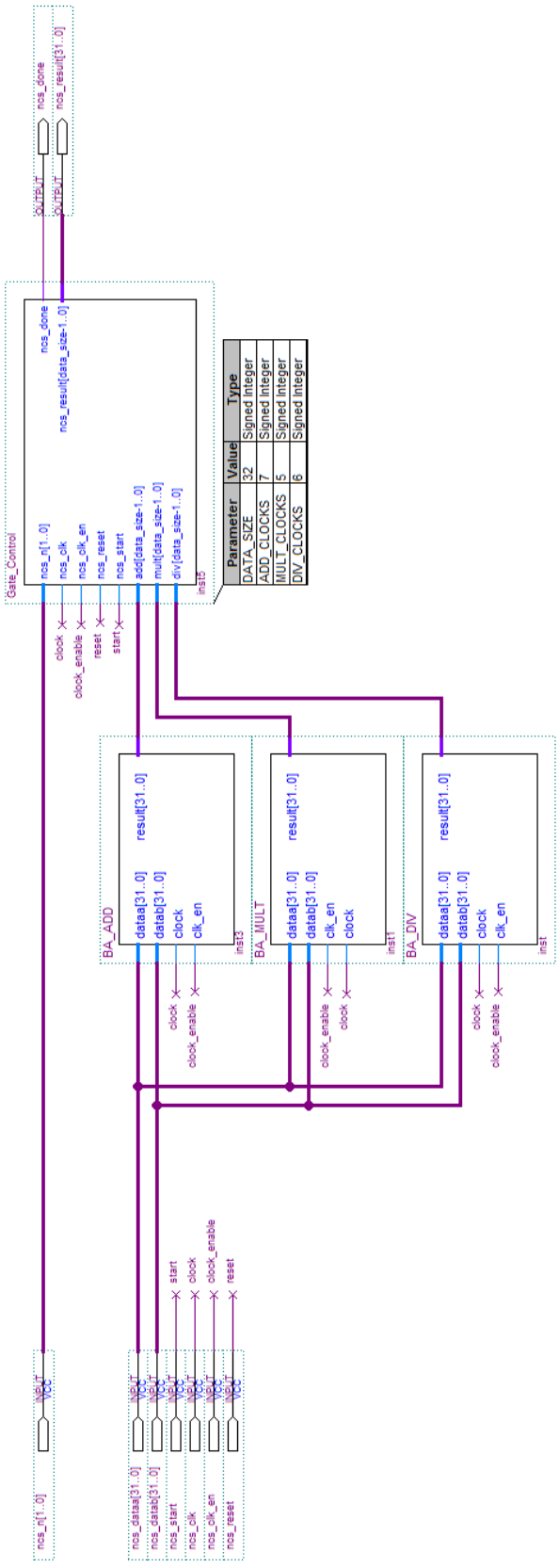


Fig. 4.9 Block Schematic Diagram of Bayesian Algebra functions implemented as a Custom Instruction. Only one operation is allowed at any given time. The *ncs_n* signal selects the desired function.

asserts the *ncs_done* flag along with the result. The signal responsible for the selection of the operation is the *ncs_n*, which is an optional signal from the CI interface and its basically an index.

Just like a common processor (Figure 3.9 shows the Nios II ALU), these operations cannot run simultaneously, which can be a limitation. As an example, we can calculate the same output as the Dynamically Generated GUT Tree, seen previously in chapter 2, using the algebraic form instead of the GUT expression:

$$\frac{1}{(a+b)\times(\frac{1}{c}+d)} = G(T,G(F,a,b),G(F,G(T,U,c),d)) \tag{4.2}$$

this renders two additions, one multiplication and two divisions with Bayesian operators.

4.3 Assemblies of Gates

4.3.1 Static GUT Tree

In Chapter 2 we presented the concept of a Static GUT Tree, where the top layer receives all the inputs and the output is given by the last GUT in the tree.

To apply this into a CI, we took advantage of the DMA Controller to load the values into the gate tree. Due to data bus limitations, these inputs do not arrive instantaneously, so the solution was loading each GUT as soon as there was at least three inputs available. This enabled the GUTs to start operating as soon as data was available. In Figure 4.10 we have a simple example that illustrates this concept. From the DMA Controller, data is provided in *dataX*, *dataY* and *dataZ* input signals. The controller then gives the ID of the appropriate gate using *gut_select* and a start command using *gut_start* signal. When a GUT has finished the *ncs_done* goes high, and the GUT that receives the results waits for all the previous GUTs to finish. Both DMA controller and GUT Tree assembly component can be seen in Figure 4.11.

Another option with GUT Trees would be running in pipeline mode. This design would be difficult to implement as a Custom Instruction because multiple operations could be in transit at the same time. However, assuming that input values can arrive instantaneously, or at least in a few clock cycles, a new calculation could start as soon as the first layer finishes. The latency would depend on the number of layers in the Tree, and the throughput on the number of clock cycles of the worst case input.

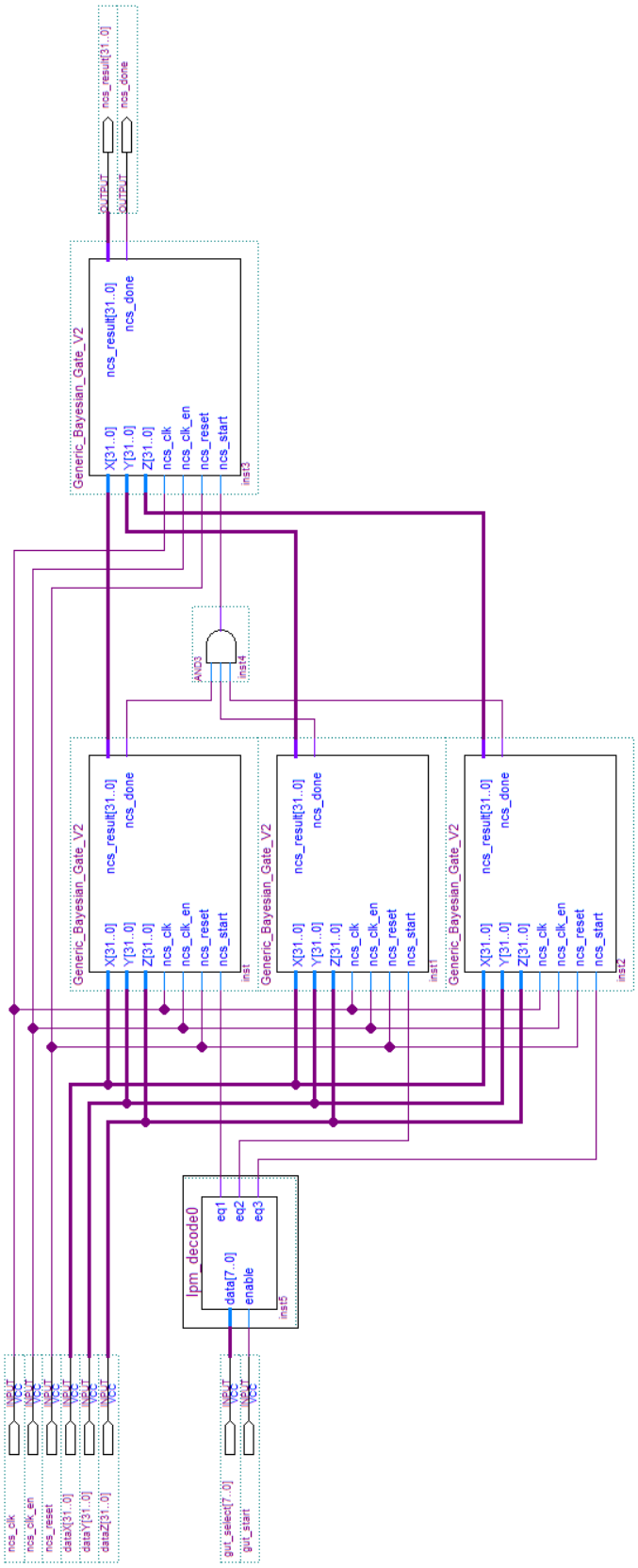


Fig. 4.10 Dynamically Generated GUT Tree, where the inputs in the first layer are loaded sequentially by the DMA Controller.

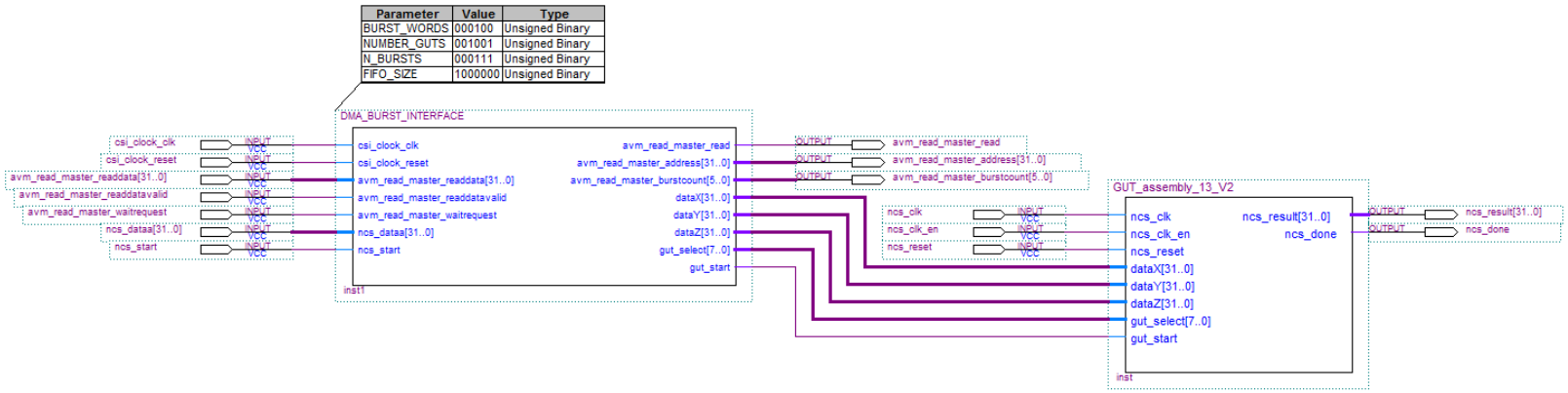


Fig. 4.11 Block schematic diagram of the DMA controller and GUT Tree assembly viewed from Quartus II.

4.3.2 Dynamically Generated GUT Tree

For the case of Dynamically Generated Trees, the implementation can be efficiently achieved using a parsing and generating tool [7]. This tool converts a string that represents the desired implementation, for instance 'GGTXYGTXYGUTGUTGUTGUTGUTGUTGUGTXYGTXY', into a VHDL code that can be compiled in Quartus. Figure 4.12 shows the RTL of a GUT Tree implemented using this tool.

4.3.3 Bayesian Algebra Trees

Bayesian Algebra Trees are similar to Dynamically Generated GUT Trees, but instead of using generic gates, they use the specific Bayesian operator directly. This tree implements add, multiply or divide functions as required by the problem. In the case of FPGAs, these use less resources than GUT Trees due to the fact that they do not implement anything more than the essential. Figure 4.13 depicts an example of this tree of operators.

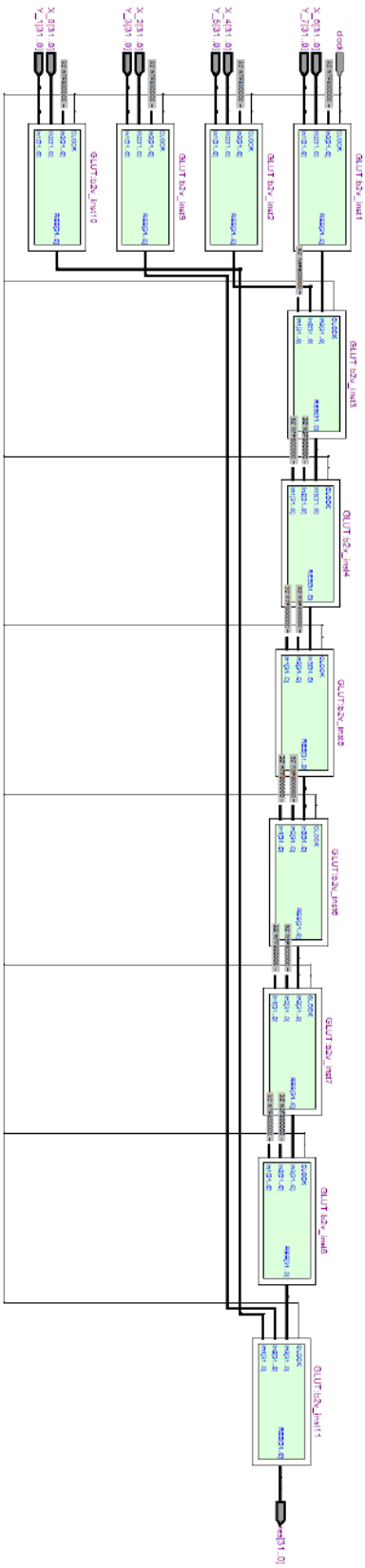


Fig. 4.12 Example of an Dynamically Generated GUT Tree generated using the parser tool (From [7]).

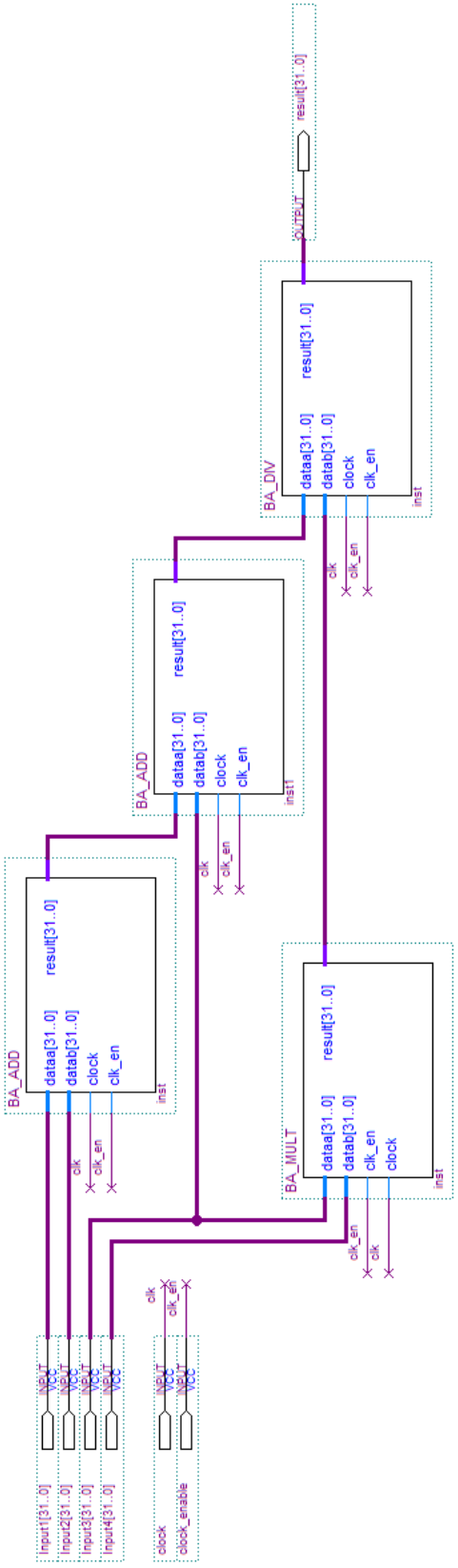


Fig. 4.13 Block Schematic Diagram of a Bayesian Algebra Tree using different operations in order to execute a pre-selected function.

Chapter 5

Results

In this chapter we present some results relative to the implementation of the designs described previously. These results are based on timing, area and power efficiency, as well as, maximum frequency. In addition, we present an overall analysis of the implemented system and some conclusions.

5.1 Clock Cycle Performance

5.1.1 Soft-GUT

To have an idea of what kind of performance can be achieved using a software based solution, we ran the Soft-GUT in different processor architectures. The code used to count the elapsed time can be seen bellow:

```
char buf[40];
clock_t exec_t1, exec_t2;
// get system time before starting the process
exec_t1 = times(NULL);
for (i=0; i<samples; i++){
        Soft_GUT(x,y,z);
}
// get system time after finishing the process
exec_t2 = times(NULL);
gcvt(((double)exec_t2-(double)exec_t1) / (alt_ticks_per_second()),
   10, buf);
alt_putstr(buf); //print the result
```

In this case, we compared the time it took to run a given amount of samples using different Nios II cores and also an Intel Core i7-3630QM Processor(with a similar code). Because

these architectures run at considerable different clock frequencies, these were compared using the number of clock cycles needed to complete a single operation.

Some functions were also added to the cores to observe their impact in performance, these include:

1. A Floating Point Custom Instruction provided by Altera.
2. Dedicated Multipliers/Dividers for integer numbers

The Nios II Economic core, being the most simple, has no option for dedicated Multipliers/-Dividers.

The clock cycle performance is shown in Figure 5.1, where it is visible the performance gap between a Nios II Economic core and the Intel i7 CPU. Probably due to multiple levels of cache and other hardware based optimizations present in high-end processors.

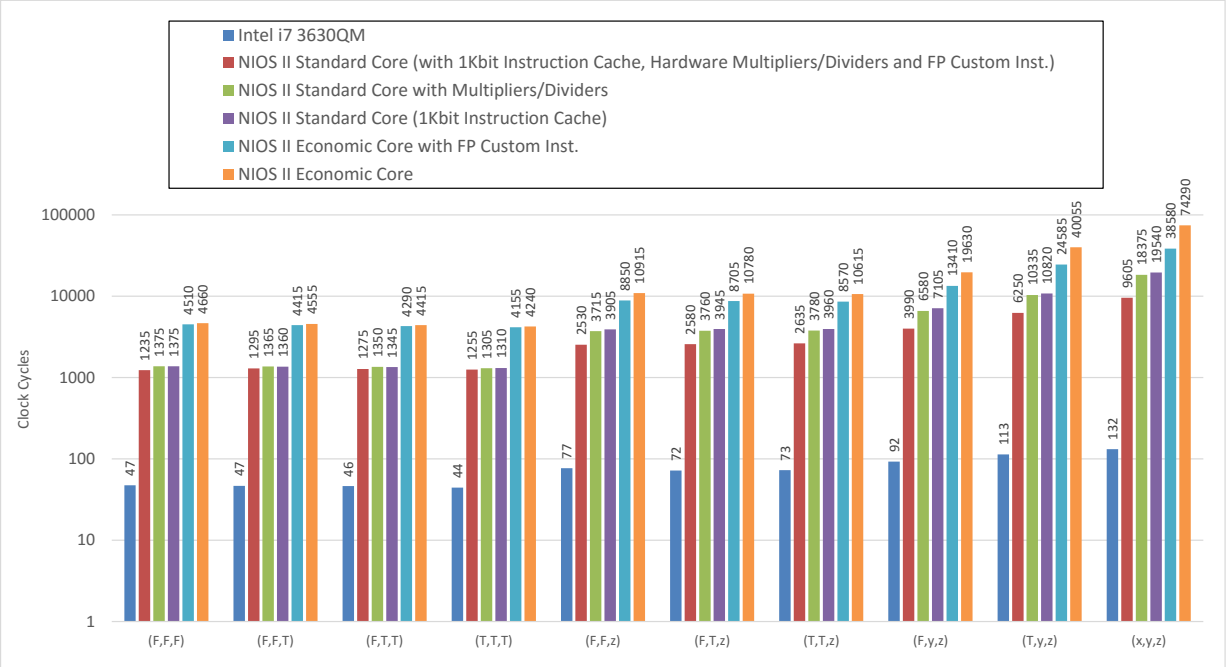


Fig. 5.1 Comparison of the Software based GUT using different processor architectures. The graphic uses logaritmic scale on the Y-Axis.

5.1.2 Custom Instructions

For the Floating Point GUT itself, we already know the clock cycle performance (Figure 4.5), it is a consequence of the logic design implemented. But there are other factors that

contribute to the global number of clock cycles. In order to compare these influences, tests were made using three different GUT designs with both Economic and Standard cores. The results are presented in Figure 5.2 where the Standard core shows better results mainly due to the use of Instruction Cache. We can also see the number of clock cycles required for Static GUT Trees using 2 and 3 layers. All these results have Best and Worst cases which depend on the type of inputs. A combinational output like *G(F,F,F)* being the Best case and an arithmetic case like *G(x,y,z)* being the Worst case.

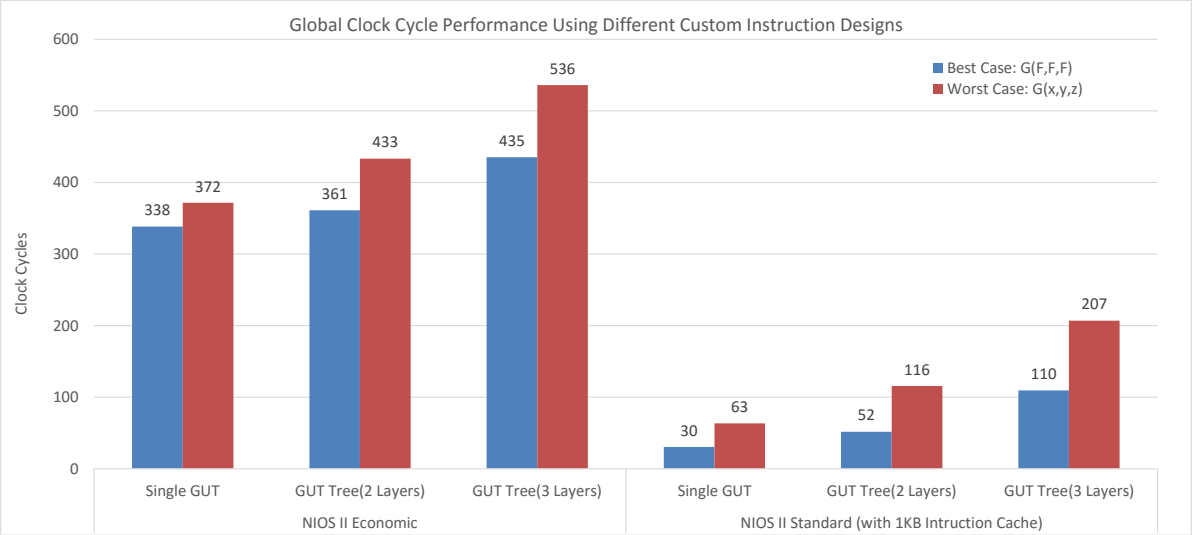


Fig. 5.2 Performance comparison of 3 implementations of the GUT and Static GUT Tree. On the left side using a Nios Economic core and, on the right side, using a Standard core with memory cache enabled.

5.2 GUT and Bayesian Algebra Resource Usage

The resources available in each development board depend on the FPGA chip being used. Here, we present the resource usage of different components for some commonly available development boards. Table 5.1 presents the resource usage on a Cyclone IV, Table 5.2 on a Stratix IV and Table 5.3 on a Stratix V.

To have an idea of how many of these elements can be fitted in each development board, we applied these values to the limits of each FPGA chip. The result are visible in Figure 5.3 where we can see that the DE4 with Stratix IV allows up to 128 Floating Point GUTs and 113 Fixed Point GUTs. On the DE5 development board with Stratix V chip, we see that the number of GUTs decreases compared with the DE4, this is due to differences in the architecture and how hardware multipliers are used.

	Total Combinational Functions	Dedicated Logic Registers	Memory Bits	Embedded Multiplier 18x18-Bit
Floating Point GUT	2479	1407	4608	8
Fixed Point GUT (16 Bit)	1634	64	0	5
BA_ADD	835	320	233	0
BA_MULT	852	493	0	0
BA_DIV	239	227	4642	8
DMA Controller	253	177	2048	0

Table 5.1 Resource consumption of the main components on a Cyclone IV FPGA Architecture.

	Total Combinational Functions	Dedicated Logic Registers	Memory Bits	Embedded Multiplier 18x18-Bit
Floating Point GUT	1684	1488	4700	8
Fixed Point GUT (16 Bit)	1542	237	0	9
BA_ADD	557	359	0	0
BA_MULT	609	478	0	0
BA_DIV	167	282	4608	8

Table 5.2 Resource consumption of some components on a Stratix IV FPGA Architecture.

	Total Combinational Functions	Dedicated Logic Registers	Memory Bits	DSP Blocks
Floating Point GUT	1684	1627	4700	5
Fixed Point GUT (16 Bit)	1579	64	0	5
BA_ADD	572	359	0	0
BA_MULT	614	478	0	0
BA_DIV	174	422	4608	5

Table 5.3 Resource consumption of some components on a Stratix V FPGA Architecture.

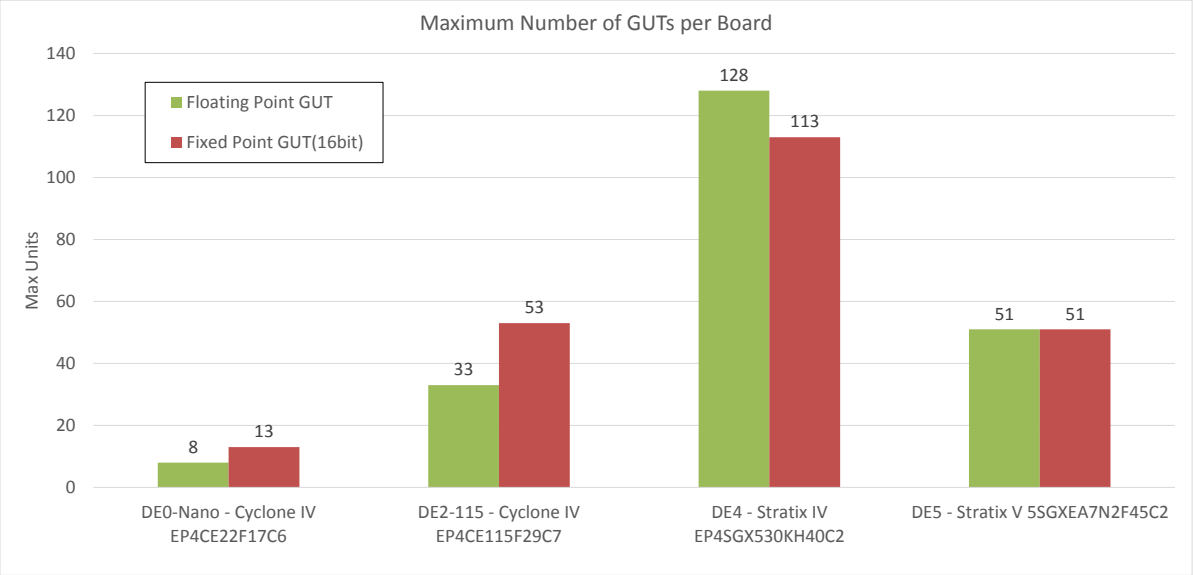


Fig. 5.3 Maximum number of GUTs on different development boards.

Relative to individual Bayesian Algebra operators, the maximum number of each type per board is presented in Figure 5.4. Basically, this gives an idea of the limits but the resource usage of a combination of these three elements will depend on the Bayesian function being solved. The use of the BA_DIV component is limited by the number of Embedded Multipliers. Also note that the Floating Point GUT follows the same values has BA_DIV, which means that the use of hardware multipliers in division is the main limiting factor.

5.3 Maximum Frequency

In this section the maximum frequency is presented relative to the current implementation of the components, however, these were not completely constrained with Time Quest Timing Analyser Tool present in Quartus II. This means that if those signals paths were constrained the fitter would reallocate components to fit the needs of a specific frequency. Taking this into account, these tests are still valid because they were made in a uniform pattern. The values presented in Table 5.4 might be substantially improved by constraining the signals paths.

On another note, there were some technical difficulties regarding the final design of the Fixed Point GUT. The compilation with the Timing Analyser Tool did not return any valid results on maximum frequency. However, early tests showed that it could run at about 5 MHz. In section 5.5 we will present details on why we had this low performance.

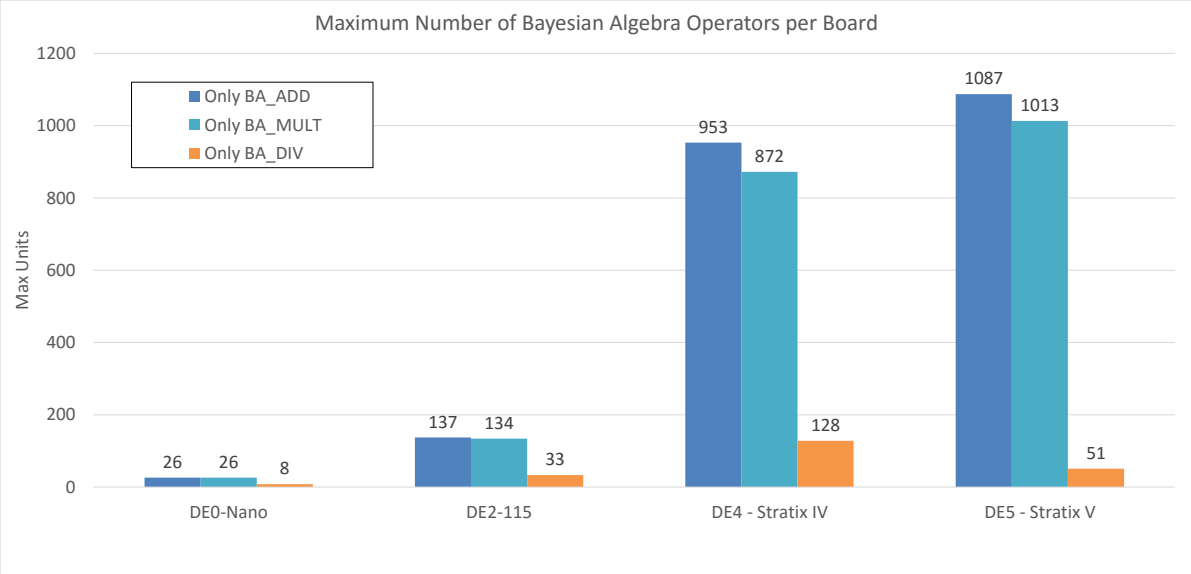


Fig. 5.4 Maximum number of Bayesian Algebra Operators using Floating Point representation on different development boards. Because Addition and Multiplication can be made with logic elements only, Emebeded Multipliers (DSPs) were reserved for Division. The graphic does not take into account combinations with different types of gates due to this being highly depended of the final application.

	Slow 1200mV 85C Model	**Slow 1200mV 85C Model**
Floating Point GUT	64.99 MHz	73.2 MHz
Fixed Point GUT (16 Bit)	5 MHz*	5 MHz*
BA_ADD	175.05 MHz	196.73 MHz
BA_MULT	187.23 MHz	211.33 MHz
BA_DIV	125.02 MHz	140.86 MHz

Table 5.4 Maximum Frequency of Bayesian Gates on Cyclone IV Architecture (Designs not constrained).

5.4 Power Consumption

For power consumption, tests were made on the Floating Point GUT in various configurations using Modelsim and PowerPlay Tool from Quartus. To achieve this, components had to be compiled in Quartus to incorporate signal delays and transition times. Then, the generated simulation files were added to Modelsim and tested with a testbench. The simulation generated a Value Change Dump (.vcd) file, which is basically a waveform file, that was read by the power analyser.

Power consumptions did not include DMA or the Nios II processor and all ran at 50 MHz clock frequency. The tests made were based on a single GUT either as a Custom Instruction or Pipelined. To verify the scalability of the power consumption we ran the tests again, but with a tree of thirteen GUTs also implemented as custom instruction or pipelined. Idle times were added to simulate the behaviour of a custom instruction. In the case of pipeline, the GUT ran continuously. The same was done for the GUT Tree. A better observation of the power consumption can be seen in Figures 5.5 and 5.6.

			Total Thermal Power Dissipation	Core Dynamic Thermal Power Dissipation	Core Static Thermal Power Dissipation	I/O Thermal Power Dissipation
Single GUT	Custom Instruction	Best Case	125.60	11.75	77.89	35.96
		Worst Case	127.01	12.03	77.88	37.10
	Pipelined	Best Case	125.96	11.42	77.89	36.64
		Worst Case	142.35	13.58	77.84	50.92
13 GUT Tree	Custom Instruction	Best Case	262.36	112.36	101.57	48.43
		Worst Case	256.65	116.19	101.38	50.08
	Pipelined	Best Case	256.20	110.56	101.55	44.08
		Worst Case	288.76	135.73	101.27	51.76

Table 5.5 Power consumption of four Floating Point GUT implementations. The values present in this table are in milliwatts (mW).

5.5 Overall Analysis

Beginning with the Soft-GUT, the results showed that high-end CPUs use considerably less clock cycles than the Nios II to execute the same function. In addition, an i7 CPU can run at more than 3 GHz clock frequency which is considerably better than the Nios II is capable of. Nevertheless, the purpose of Nios in this work was serving as test-bench and the final application may be an embedded system without processor.

Another alternative design tested here was the GUT with Fixed Point representation. Although it is combinational, the output of this design approach was not the best. As seen in Figure 5.7, it lacks resolution and deviates from the desired output in some cases. Also, being heavily combinational gave it very low maximum frequency. Attempts to solve this issue

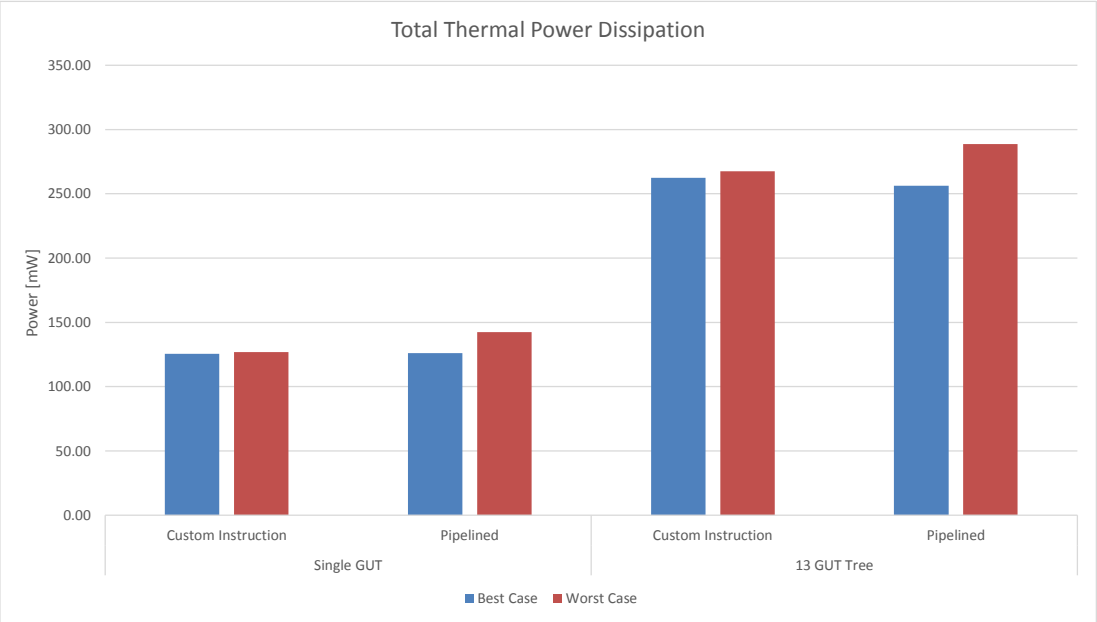


Fig. 5.5 Total Thermal Power Dissipation compared in 4 GUT implementations.

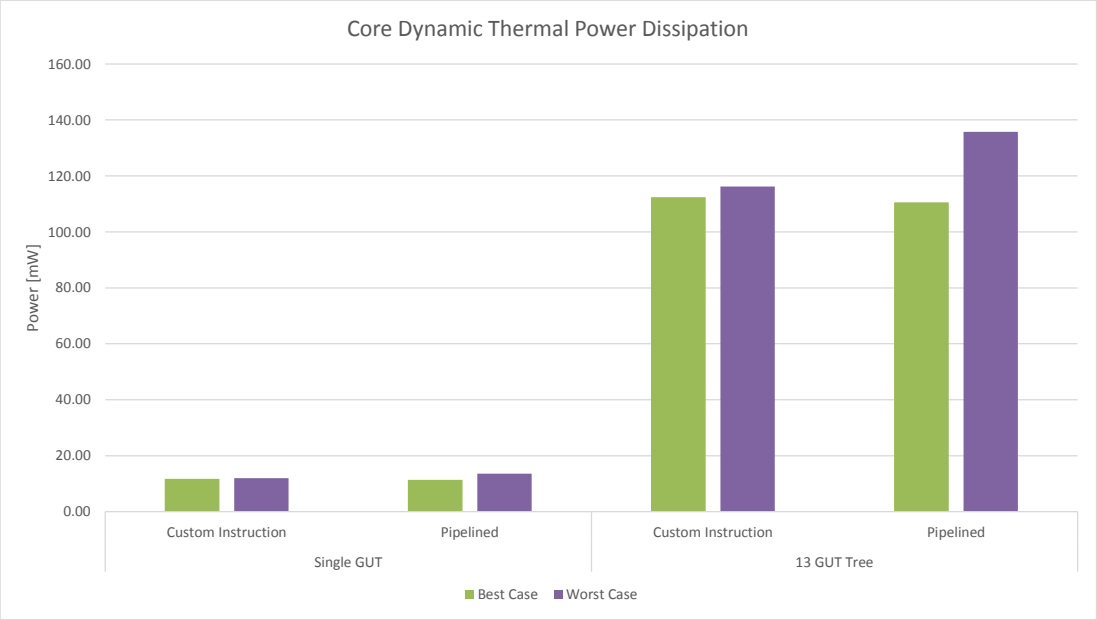


Fig. 5.6 Core Dynamic Thermal Power Dissipation compared in 4 GUT implementations.

with registers, did not improve the results. The Fixed Point GUT was not further pursued due to the problems mentioned above, representing probabilities values with odd ratios in Floating Point representation proved to be a better approach. In the end, the Fixed Point GUT was not considered as a viable design approach.

Fixed Point GUT as Nios II Custom Instruction **Processor: NIOS II Standard**
Freq: 50 Mhz

	Decimal					Hexadecimal			
Input	**p**	**q**	**r**	**Output**	**Ideal**	**X**	**Y**	**Z**	**Output**
(F,F,F)	0	0	0	0	0	0x0000	0x0000	0x0000	0x0000
(F,F,T)	0	0	1	1	1	0x0000	0x0000	0xFFFF	0xFFFF
(F,T,T)	0	1	1	1	1	0x0000	0xFFFF	0xFFFF	0xFFFF
(T,T,T)	1	1	1	0	0	0xFFFF	0xFFFF	0xFFFF	0x0000
(F,F,z)	0	0	0.0273	0.0273	0.0273	0x0000	0x0000	0x0700	0x0700
(F,T,z)	0	1	0.0273	1	1	0x0000	0xFFFF	0x0700	0xFFFF
(T,T,z)	1	1	0.0273	0.0011	0	0xFFFF	0xFFFF	0x0700	0x0049
(F,y,z)	0	0.3125	0.0273	0.3255	0.32555	0x0000	0x5000	0x0700	0x5356
(T,y,z)	1	0.3125	0.0273	0.9874	0.9874	0xFFFF	0x5000	0x0700	0xFCC5
(x,y,z)	0.4375	0.3125	0.0273	0.5552	0.5552	0x7000	0x5000	0x0700	0x8E1F

Fig. 5.7 Outputs generated by Fixed Point GUT (16-Bit) as a Custom Instruction. The values were extracted from the Nios II Console in Eclipse. The result from the input (T,T,z) deviated from the one expected.

In this work we can observe that for an embedded system, the dedicated hardware accelerator can reduce up to two orders of magnitude the number of clock cycles, while maintaining very low power consumption. Even a tree of thirteen GUTs will stay under 300 mW of power consumption. Also, in the Single GUT implementation with DMA access results show clock cycle efficiencies between 30 and 63 clocks per instruction, which is better than the i7 CPU.

For an embedded system where resources are very limited, the best option would be the alternative ALU, with Add, Multiply and Divide operators as custom instructions. This would give the embedded system Bayesian algebra compatibility without DMA or external interfaces. The drawback would depend on the interface overhead of the Custom Instruction.

If the solution to a Bayesian Inference problem requires an extensive number of operations, trees become the best option. A big tree can perform multiple operations at the same time due to the inherent parallelism of the design. If resources are more limited and the application is always the same, a Dynamically Generated Tree will save some resources.

If the inference problem changes frequently, Static GUT Trees are the best option because the inputs will define the behaviour of the tree. This however, will consume more resources because the design will consist of a full ternary tree.

In general, the best solution depends on the final application and resources available.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

In this work, we explored the implementation space of Bayesian computing on reconfigurable logic. The implemented solutions and performance tests provided some insights on the trade-offs, and allowed us to draw some conclusions.

Using the NIOS-II soft processor as a base, custom instruction hardware implementations were made for GUTs with both Floating and Fixed Point representations, as well as, independent Bayesian operators. Trees of both Floating Point GUT and Bayesian operators were also made to test the scalability.

We concluded that the Fixed Point GUT is feasible but the preliminary tests show no significant advantages over Floating Point. Moreover, it has limited precision and dynamic range.

For an embedded processor with limited resource options, a Floating Point ALU modified for Bayesian algebra can provide support for the probabilistic computations required while maintaining minimal resource footprint.

A Single GUT, functioning as dedicated hardware accelerator for an embedded processor, can reduce up to two orders of magnitude the number of clock cycles required for the same Bayesian computation in software.

If the inference problem changes frequently, a Static GUT Tree provides flexibility and a generic solution for Bayesian computation. While a Dynamically Generated GUT or a tree of Bayesian algebra operators can reduce the number of resources but loses the flexibility.

6.2 Future Work

Having performed this batch of tests, a future work should address an integrated software-hardware solution to run Bayesian inference problems. Since we used a soft-processor, this is basically a software work to break down the inference problem to the size of the implemented hardware block. This could done by having a compilation chain between the ProBt framework [25] and the reconfigurable logic hardware. This would make the bridge from the Bayesian problem concept to a final computing implementation.

Concerning the implemented solutions, data transfers might also be improved with tightly coupled memory or shared memory dedicated for Bayesian computations. Reducing the number of clock cycles required. Future work could apply this to a robotic system if the problem can be mapped to a single GUT Tree that can fit on the device. This would be a true embedded custom solution without the overhead of software processor and data transfers.

References

[1] Altera. Altera website: https://www.altera.com/, August 2015.

[2] Terasic. Terasic website: http://www.terasic.com.tw/en/, August 2015.

[3] Altera. *Cyclone IV Device Handbook*, April 2014.

[4] Altera. *Nios II Classic Processor Reference Guide*, April 2015.

[5] Altera. *Nios II Custom Instruction User Guide*, January 2011.

[6] Altera. *Avalon Interface Specifications*, March 2015.

[7] Miguel Garcia Almeida. Exploring different implementations of probabilistic computations on fpgas. Master's thesis, University of Coimbra, 2014.

[8] Narges Bani Asadi, Teresa H. Meng, and Wing H. Wong. Reconfigurable computing for learning bayesian networks. In *Proceedings of the 16th International ACM/SIGDA Symposium on Field Programmable Gate Arrays*, FPGA '08, pages 203–211, New York, NY, USA, 2008. ACM.

[9] R. Bannister, D. Gregg, S. Wilson, and A. Nisbet. FPGA implementation of an image segmentation algorithm using logarithmic arithmetic. In *Circuits and Systems, 2005. 48th Midwest Symposium on*, pages 810–813, 2005.

[10] M.N. Marsono, M.W. El-Kharashi, and F. Gebali. Binary LNS-based naive Bayes inference engine for spam control: noise analysis and FPGA implementation. *IET Computers & Digital Techniques*, 2(1):56–62, 2008.

[11] Pierre Bessiere Christian Laugier and Roland Siegwart. *Probabilistic Reasoning and Decision Making in Sensory-Motor Systems*. Springer-Verlag, 2008.

[12] BAMBY. D2.1: Abstract probabilistic model of biochemical cascades. December 2014.

[13] Mingjie Lin and Yaling Ma. Base-calling in DNA pyrosequencing with reconfigurable Bayesian network. In *Reconfigurable Computing and FPGAs, 2009. ReConFig '09. International Conference on*, pages 95–100, 2009.

[14] Fan Zhou, Jun Liu, Yi Yu, Xiang Tian, Hui Liu, Yaoyao Hao, Shaomin Zhang, Weidong Chen, Jianhua Dai, and Xiaoxiang Zheng. Field-programmable gate array implementation of a probabilistic neural network for motor cortical decoding in rats. *Journal of Neuroscience Methods*, 185(2):299 – 306, 2010.

[15] R. Laurent M. O. Abdallah R. Lehy M. Faix, E. Mazer and J. Lobo. Cognitive computation: from bayesian models to bayesian machines. In *14th IEEE Int. Conference on Cognitive Informatics and Cognitive Computing (ICCI*CC15), Beijing, China, July 2015*.

[16] J. F. Ferreira R. Duarte, J. Lobo and J. Dias. Synthesis of bayesian machines on fpgas using stochastic arithmetic. In *2nd International Workshop on Neuromorphic and Brain-Based Computing Systems (NeuComp 2015), associated with DATE2015, Design Automation Test Europe 2015, March 2015.*, 2015.

[17] Joao Filipe Ferreira, Jorge Lobo, Pierre Bessiere, Miguel Castelo-Branco, and Jorge Dias. A bayesian framework for active artificial perception. *IEEE TRANSACTIONS ON CYBERNETICS*, 43(2):699–711, April 2013.

[18] Pierre Bessiere. Probabilistic algebra and generic bayesian gates. BAMBI internal documentation, 2014.

[19] Pierre Bessiere and Jacques Droulez. Bayesian gates baysian algebra calculus tree. BAMBI internal documentation, 2014.

[20] Altera. *Nios II Classic Software Developer's Handbook*, May 2015.

[21] Altera. *Embedded Design Handbook*, July 2011.

[22] Altera. *Quartus II Handbook Volume 1: Design and Synthesis*, May 2015.

[23] Altera. Avalon memory-mapped master templates, August 2015.

[24] EDA Industry Working Groups. http://www.eda.org/fphdl/.

[25] Pierre Bessiere, Emmanuel Mazer, Kamel Mekhnacha, and Juan Manuel Ahuactzin. *Bayesian Programming*. Chapman & Hall/CRC Press, December 2013.

Appendix A

Tutorials

This is a quick tutorial that will show how to create a basic Nios II system with a few basic peripherals. It is expected that the reader has basic knowledge of the Quartus II software environment. This tutorial was created for Quartus II v15 Web Edition which includes Nios II EDS. The development board can either be a DE0-Nano or a DE2-115.

The following list includes some documentation available online:

- Nios II Classic Processor Reference Guide
- Nios II Classic Software Developer's Handbook
- Nios II Hardware Development Tutorial
- Nios II Software Tutorial
- Nios II Performance Benchmarks
- Nios II Core Implementation Details
- Nios II Custom Instruction User Guide
- Nios II Instruction Set Reference

A.1 Creating a Nios System in Quartus II/Qsys

1. Start the **Quartus II** software.
2. Select **New Project Wizard** on the splash screen, or in the **File** menu, click **New Project Wizard**.
3. Once the Dialog box appears, click **Next**, set a **working directory** and **name** for your project. The remaining settings can be left as default. Click **Finish**.
4. Now that the project is created you can open **Qsys** on the **Tools** menu.
5. In **Qsys**, go to the **File** menu and click **Save As**. Choose an appropriate name for your Nios system and click **Save**.
6. Use the **IP Catalog** window on the upper left side to add the following components (**Double Click** on the name to add component):

   ⇒ On-Chip Memory (Options: Memory size 32768 bytes; data width 32 bit)

   ⇒ Nios II Processor (Options: Nios II/e.)

   ⇒ JTAG UART (Options: Default.)

   ⇒ Interval Timer (Options: Default.)

   ⇒ System ID Peripheral (Options: Default.)

   ⇒ PIO (Options: Default.)

7. In the Connections column select the **instruction_master** signal, inside the Nios II Processor, and connect it to **debug_mem_slave** and **s1** in the On-Chip Memory.
8. In the same way, select the **data_master** signal in the processor and connect it to all the peripherals, including On-Chip Memory).
9. Connect the **clk** signal from **Clock Source** to all the other components.
10. Also connect **IRQ** signals to the processor (the IRQ column is far to the right and may be hidden).
11. Now go to **System** menu and select **Assign Base Addresses** and **Create Global Reset Network**.

12. Now we need to export the PIO (Parallel I/O) interface **external_connection** out of the Qsys component. Click **Double-click to export** on the export column inside System Contents.

13. Go to the **Nios Processor** component and select **Vectors** tab. On **Reset Vector Memory** and **Exception Vector Memory** select **On-ChipMemory.s1**. Click **Finish** or **Close** the component window.

14. If there are still error messages, go to **System** menu, and run **Assign Base Addresses**, **Assign Interrupt Numbers** and **Create Global Reset Network** again. If the error messages persist try to check the components related to the error message.

    Note: Most of the time is just a forgotten parameter or a bad connection. The final setup should look something like Figure A.1.

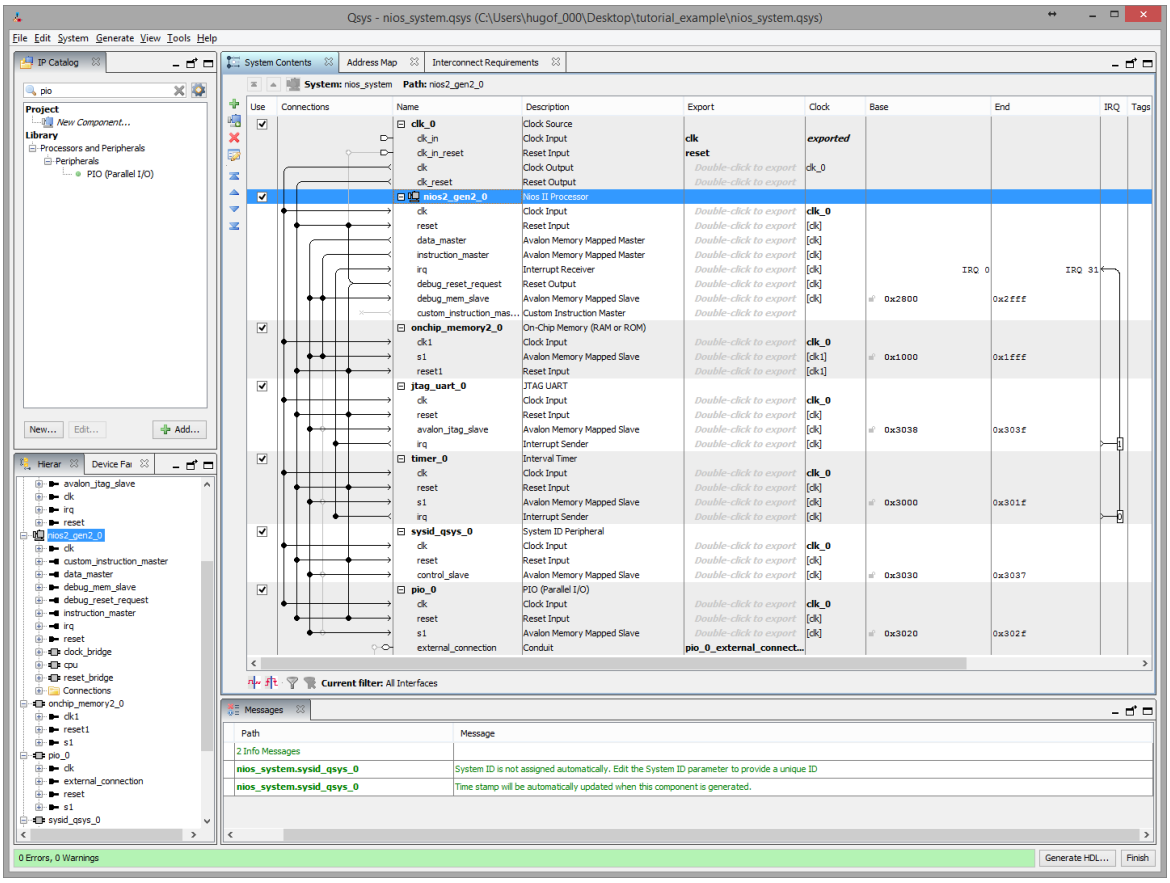


Fig. A.1 Nios system in Qsys

15. On top menu bar, select **Generate > Generate HDL...** and, on the Dialog box, click **Generate**.

16. The component has been created. Now open Quartus and add the new (**<name_nios_system>.qip**) file to the project. Go to **Project** menu tab and select **Add/Remove Files in Project...**. The Dialog box appears. Add the file **<name_nios_system>.qip** located in **<project directory>/<name_nios_system>/synthesis** and in the **Settings** Dialog box, click **Add** and then **OK** at the bottom.

17. In Quartus, create a new **Block Diagram/Schematic File** and save it in the main project directory.

18. Double click anywhere inside the Block Diagram design window. The component may not be listed but we can add the **.bsf** component file by clicking on the button with "three dots". Look for the (**.bsf**) block file, it should be in **<project directory>/<qsys component>**. Click **OK** after adding the new component and place it inside the design space.

19. Double click again anywhere in the space and add the following pins **VCC**, an **INPUT** and an **OUTPUT**.

20. Change the name of the input pin to "**CLOCK_50**", and the name of the output pin to "**LED[7..0]**".

21. Connect the **VCC** pin to **reset_n**, **CLOCK_50** to **clk_clk** and **LED[7..0]** to **pio_0_external**. Save the project.

22. Go to **Assignments > Device** and select the chip that matches the one in the development board. If prompted to remove the pins assignments, select **No**. Click **Ok** to accept the changes.

23. Now we need to import pin assignments to the project. This can be done using a (**.qsf**) file, which can be found online, with all the pin connections of the development board. If you have the file go to **Assignments > Import Assignments...**.

24. Compile the Project.

25. To **upload** the project into the board you the **USB-Blaster** cable. If you do not have the **drivers** installed, connect the board to your computer (if not already done), and open **Device Manager**. Look for **Other Devices** and select **USB-Blaster** with right mouse button, and then Update Drivers. Use a manual search, and include the Quartus II folder.

26. In Quartus, go to **Tools > Programmer**. In the dialog box click **Hardware Setup...** and then in **Currently selected hardware > USB-Blaster**. Close the dialog box.

27. Click on the **Auto Detect** button and Select the **EP4CE22** or the **EP4CE115** device.

28. Add/Change the file **.sof** located at **<project directory>/output_files**. Tick the **Program/Verify** box, and click **Start**. The FPGA is now programmed with the Nios system.

A.2 Running a Programs on NIOS II using Eclipse IDE.

This a quick tutorial on how to write, compile and upload a Nios II program.

1. Open your compiled Nios II project in Quartus II, at the menu bar on top select **Tools > Nios II Software Build Tools for Eclipse**. Note: If this is the first time you open Eclipse, select a Workspace folder for your programs.

2. Go to **File** menu and select **New > Nios II Application and BSP from Template**.

3. On the new dialog box, select the **SOPC** file **(.sopcinfo)** located in the project folder created in Quartus. Note: This file contains information and characteristics of the Nios System required to compile the programs. Also, the path must not contain any spaces.

4. Give the project a name and select a **Project Template**, for example the "Hello World Small". Click **Finish**.

5. In the **Project Explorer** window, right mouse click on <project_name>_bsp and select **Nios > Generate BSP**. Right mouse click again and select **Build Project**. Right click on the <project_name> folder and also select Build Project.

6. Now the program is ready to be uploaded. Go to **Run** menu on top and select **Run Configurations...**. Double click on **Nios II Hardware**, select the current project and click on **Target Connection** tab. Click on **Refresh Connections** and tick all the boxes bellow **except Disable "Nios II Console" view**.

7. Click Run. The output should be visible in the **Nios II Console > "Hello from Nios II"**.

A.3 Adding SDRAM Memory to the System.

Development boards have external memories which Nios can use to store data. To used this memory a controller must be added and external pins configured. Also, SDRAM memory requires compensated clocks due to the long path between FPGA chip and the memory chip. This is solved using a PLL compensated clock.

1. Open your Nios II project in Quartus and then open Qsys. Select the .qsys file that has the Nios System.

2. Either remove or deactivate the On-chip Memory component.

3. Go to **IP Catalog** and find **SDRAM Controller** under memory interfaces and controllers. Leave all the settings as default an click finish.

4. With the component added, go to **System** menu and select **Create Global Reset Network** and **Assign Base Addresses**.

5. Connect the Clock Input to the source and Avallon Slave signal (**s1**) to both **data** and **instruction** masters from Nios II Processor. Note: This controller runs at the same clock as the processor, only the memory needs a compensated clock.

6. Now Export the Wire Conduit. The SDRAM will be connected to these wires in Quartus.

7. Select the processor and update the **vectors** in the vector tab.

8. Hit the **Generate HDL...** button in the lower right corner. Close Qsys after generating the new files.

9. In the Quartus II window open the Block Diagram File with the Qsys component. Remove the old component and add the new one from the project folder.

10. Now the new component has extra pins that will connect to the SDRAM. To find which pins we need to add, open the Pin Planner in Assignments menu and scroll down to find **DRAM_*** nodes. Note: If you look closely you will find a match between the names of those nodes and the ones on the component.

11. Right click on the Nios component and select **Generate Pins for Symbol Ports**. Now the names of the pins must be changed to match the ones in the Pin Planner.

12. The clock for the memory can now be added, double click on the working space and add an **OUTPUT** pin. Give the name **DRAM_CLK**.

13. Now we need to add the PLL that will generate the compensated clock. Open the **IP Catalog** in Tools and find the **ALTPLL** component. Give it a new variation name. The **MegaWizzard** will open, change the frequency to **50 MHz**, in Inputs/Lock remove **areset** and **locked output**.

14. Skip to **Output Clocks** tab and select **Use this clock**, make sure you add -3ns to the phase shift. Click Finish and do not forget to add a symbol file for Quartus in the files list.

    Note: Check if Quartus has created the PLL with the same name as the processor, this will give an error in the compilation report. Select the pll, go to Properties and change the name to something else.

15. Connect the CLOCK_50 and DRAM_CLK pins to the PLL. Compile the Project.

Appendix B

Schematics and Code

B.1 GUT Software Function

```
float gut_function(float Data[])
{
unsigned int *ptr1,*ptr2,*ptr3, i = 0, j = 0;
unsigned int count_F = 0, count_T = 0, count_var = 0;
unsigned int data_valid[3] = {0,0,0};
float data_buffer[3] = {0,0,0};
unsigned int check_exponent[3];
// Pointers
ptr1 = &Data[0];
ptr2 = &Data[1];
ptr3 = &Data[2];
// Exponent Isolation
check_exponent[0] = *ptr1 & SEL_EXPONENT;
check_exponent[1] = *ptr2 & SEL_EXPONENT;
check_exponent[2] = *ptr3 & SEL_EXPONENT;
// Counts how many False, True and Variables in the input
for(i=0;i<3;i++){
        // SEL_EXPONENT happens to be equal to Infinite in binary
           float
        if(check_exponent[i] == SEL_EXPONENT){
                count_T++;
        }
        else if(check_exponent[i] == F)
                count_F++;
        else{
                data_valid[i] = 1;
                count_var++;
```

```
        }
}
//Puts the values in order
if (count_var > 0){
        for (i=0;i<3;i++){
                if (data_valid[i] != 0){
                        data_buffer[j] = abs(Data[i]);
                        j++;
                }
        }
}
// Truth Table
if (count_F == 3 && count_T == 0 && count_var == 0)
        return F;
else if (count_F == 2 && count_T == 1 && count_var == 0)
        return T;
else if (count_F == 1 && count_T == 2 && count_var == 0)
        return T;
else if (count_F == 0 && count_T == 3 && count_var == 0)
        return F;
else if (count_F == 2 && count_T == 0 && count_var == 1)
        return data_buffer[0];
else if (count_F == 1 && count_T == 1 && count_var == 1)
        return T;
else if (count_F == 0 && count_T == 2 && count_var == 1)
        return F;
else if (count_F == 1 && count_T == 0 && count_var == 2)
        return (data_buffer[0] + data_buffer[1]);
else if (count_F == 0 && count_T == 1 && count_var == 2)
        return (1 / (data_buffer[0] * data_buffer[1]));
else if (count_F == 0 && count_T == 0 && count_var == 3)
        return ((data_buffer[0] + data_buffer[1] + data_buffer[2])/(1
        + (data_buffer[0] * data_buffer[1] * data_buffer[2])));
else{
        printf("␣ERROR:(%d,%d,%d)\n␣",count_F, count_T, count_var);
        return 0;
}
}
```

B.2 GUT in VHDL using Fixed Point Representation

```
-- Fixed Point GUT
```

```
--
--                [p + q + r - 2(pq + qr +pr) + 3pqr]
-- g'(p,q,r) =  ------------------------------------
--                    [1 - pq - qr - pr + 3pqr]

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

library ieee_proposed;
use ieee_proposed.fixed_pkg.all;
use ieee_proposed.fixed_float_types.all;

entity GUT_Fixed is
generic(
Q_SIZE: natural:= 16;  --fraction
I_SIZE: natural:= 8);  --extra integers for calculations
port(
pp,qq,rr: in std_logic_vector(Q_SIZE-1 downto 0);
frac: out std_logic_vector(Q_SIZE-1 downto 0));
end entity GUT_Fixed;

architecture structure of GUT_Fixed is
signal p,q,r: ufixed (I_SIZE-1 downto -Q_SIZE) := (others => '0');
signal pq,qr,pr: ufixed (I_SIZE-1 downto -Q_SIZE) := (others => '0');
signal pqr3: ufixed (I_SIZE-1 downto -Q_SIZE) := (others => '0');
signal sum_p_q_r: ufixed (I_SIZE-1 downto -Q_SIZE) := (others => '0');
signal pq_qr_rp_2: ufixed (I_SIZE-1 downto -Q_SIZE) := (others => '0');
signal dividend: ufixed (I_SIZE-1 downto -Q_SIZE) := (others => '0');
signal divisor: ufixed (I_SIZE-1 downto -Q_SIZE) := (others => '0');
signal result: ufixed (I_SIZE-1 downto -Q_SIZE) := (others => '0');
begin
 p(-1 downto -Q_SIZE) <= to_ufixed(pp(Q_SIZE-1 downto 0), -1, -Q_SIZE);
 q(-1 downto -Q_SIZE) <= to_ufixed(qq(Q_SIZE-1 downto 0), -1, -Q_SIZE);
 r(-1 downto -Q_SIZE) <= to_ufixed(rr(Q_SIZE-1 downto 0), -1, -Q_SIZE);
 frac   <= to_slv(result(-1 downto -Q_SIZE)) when result < 1 else (
    others => '1');
 -- calculations
 pq <= resize( p * q, pq);
 qr <= resize( q * r, qr);
 pr <= resize( p * r, pr);
 pqr3 <= resize( integer(3) * pq * r, pqr3);
 sum_p_q_r <= resize( p + q + r, sum_p_q_r);
 pq_qr_rp_2 <= resize( integer(2) * (pq + qr + pr), pq_qr_rp_2);
```

```
  dividend <= resize( sum_p_q_r + pqr3 - pq_qr_rp_2    , dividend);
  divisor <= resize( pqr3 + integer(1) - pq - qr - pr , divisor);
  result <= resize( dividend/divisor , result);
end structure;
```