UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE ENGENHARIA DE COMPUTAÇÃO

LEANDRO MATEUS GIACOMINI ROCHA

# Evaluation of Arithmetic Operators for the MPPA-256 Manycore Architecture

Monografia apresentada como requisito parcial para a obtenção do grau de Bacharel em Engenharia de Computação.

Orientadora: Prof. Dra. Fernanda Lima Kastensmidt
Co-Orientadora: Prof. Dra. Katell Morin-Allory

Porto Alegre
2016

# 1 Resumo Estendido

## 1.1 Introdução

Este capítulo apresenta um resumo estendido em português sobre o Trabalho de Final de Estudos, escrito em inglês, realizado na França durante a participação do Programa de Intercâmbio Brafitec com a obtenção do duplo diploma. A motivação do trabalho assim como as etapas de desenvolvimento do mesmo são apresentadas brevemente para que se tenha uma compreensão geral texto.

### 1.1.1 Contexto do trabalho

Este trabalho de conclusão de curso em microeletrônica, decorrente da co-tutela firmada entre as universidades Institut Polytechnique de Grenoble – École Nationale de Physique, Électronique et Matériaux (Phelma) e Universidade Federal do Rio Grande do Sul, foi realizado no período de seis meses na empresa KALRAY. Esta empresa tem atuação na área de microeletrônica voltada para desenvolvimento de microprocessadores manycore de alto desempenho e baixo consumo. O principal produto desenvolvido pela empresa é o processador MPPA-256 que contém 256 núcleos de processamento funcionando a 400 MHz. Baseando-se no massivo poder de processamento paralelo com núcleos VLIW e com uma frequência de operação mais baixa se comparada aos processadores mais comuns disponíveis no mercado, este processador é extremamente eficiente energeticamente uma vez que possui uma relação de taxa teórica de pico de computação por unidade de energia em torno de 25 GFLOPS/W.

### 1.1.2 Motivação

Em processadores atuais mais populares, como aqueles pertencentes à família x86, a lógica de controle do processador é bastante engenhosa devido à retrocompatibilidade de instruções e à natureza superescalar desses processadores. Mecanismos de predição de desvios e escalonadores dinâmicos de instruções são exemplos de módulos de hardware consideravelmente complexos e, por consequência, possuem área consideravelmente grande em relação ao tamanho de cada núcleo de processamento. Assim, melhorias aportadas à construção da lógica de controle são mais propícias a surtirem efeitos consideráveis sobre consumo de energia e área utilizada

do que modificações sobre a lógica das unidades operacionais.

No processador MPPA-256 cada núcleo trabalha com um conjunto de instruções VLIW, logo algumas tarefas de otimização de execução – como a alocação de unidades funcionais – são realizadas pelo compilador. Logo, a lógica de controle do processador torna-se mais simples e, consequentemente, as unidades operacionais como somadores, multiplicadores, entre outros possuem uma representatividade maior na área e no consumo energético de cada núcleo.

A partir dessa premissa, a empresa percebeu a necessidade de explorar a implementação de diversas arquiteturas de operadores aritméticos na tecnologia empregada, buscando melhorias no desempenho do processador. Haja visto a grande quantidade de arquiteturas existentes na literatura e os diversos requerimentos e restrições impostos para cada operador aritmético no âmbito de sua utiliação, torna-se necessário a criação de uma biblioteca de hardware com circuitos aritméticos descritos em linguagem HDL, testados e sintetizados e de fácil acesso às características destes a fim de melhorar e simplificar o processo de decisão do projeto do processador quanto à parte operativa.

## 1.2 Trabalho Desenvolvido

Dado que existem diversas arquiteturas de somadores e multiplicadores definidas na literatura, é necessário efetuar um estudo sobre o estado da arte a fim de avaliar e comparar cada circuito. Dessa forma, a primeira etapa do trabalho desenvolvido remete a uma revisão teórica sobre os circuitos aritméticos assim como apresenta sucintamente todas as arquiteturas estudadas sobre as quais foram extraídos os dados da síntese. Nos anexos, encontram-se análises mais aprofundadas sobre área e timing para algumas arquiteturas selecionadas.

Visando a avaliação completa de cada arquitetura estudada, foi definido um conjunto de métricas, ao qual deu-se o nome de Quality of Results (QoR), comum tanto aos somadores quanto aos multiplicadores. Nesse conjunto são listados diversos dados característicos de cada circuito, no qual frequências de operação, área utilizada e potência dissipada correspondem aos fatores mais importantes.

Para implementar e comparar os resultados da síntese de cada arquitetura, algumas ferramentas foram desenvolvidas visando a simplificação e automatização desse processo. Quanto à implementação dos circuitos, foi desenvolvido um framework para flexibilizar a geração de códigos VHDL assim como a geração dos testbenches associados a esses circuitos. Também, diversos scripts foram desenvolvidos para que a síntese e a extração dos dados de cada circuito ocorresse de modo automático. Para catalogação dos dados, foi desenvolvido uma ferramenta de acesso ao banco de dados

criado, permitindo ao usuário, de forma simples, visualizar e comparar arquiteturas para certos critérios informados.

### 1.2.1 Gerador de Código VHDL

A descrição de circuitos em linguagem HDL pode ser uma tarefa longa e muito suscetível a erros. Além disso, certas arquiteturas são definidas a partir de modelos recursivos que cujos parâmetros de recursão dependem do tamanho dos dados de entrada. Essas arquiteturas são extremamente difíceis de descrever genericamente em VHDL e normalmente resultam em circuitos ineficientes.

Para resolver esse problema, foi desenvolvido um *framework* em linguagem Python que contém os elementos básicos da lógica combinacional propostos pela linguagem VHDL. Todos os elementos da linguagem VHDL – fios, operadores lógicos, entidades, etc – são representados através de classes nesse *framework*.

A vantagem desse framework reside justamente em resolver as complexidades inerentes a um algoritmo qualquer que descreve um circuito antes que este seja modelado através de uma linguagem HDL. Para descrever um circuito, o usuário cria um modelo em Python do algoritmo desejado usando os elementos disponibilizados pelo *framework* e as funcionalidades primitivas da linguagem Python. Assim, o resultado final é uma descrição VHDL completamente sintétizavel com todos os elementos do circuito contidos em um único arquivo. Vale ressaltar que esse arquivo não faz uso de recursos adicionais da linguagem VHDL tais como *generic*, *for*, etc, pois todas as dependências referentes a tamanho de dados e quantidade de instâncias de um componente dentro de um design são resolvidas pelo *framework*.

Adicionalmente, esse *framework* suporta a geração automática de testbenches em linguagem SystemVerilog conforme solicitação do usuário. A escolha da linguagem para tais testes foi feita baseada no ambiente utilizado dentro da empresa. O usuário deve informar uma expressão compatível com a linguagem do teste que represente a função computada pelo circuito assim como as entradas e as saídas a serem analisadas. Ao final, são gerados dois arquivos onde um deles é responsável pela interface do módulo de teste e o outro contém as instanciações e testes necessários para a verificação do circuito segundo a expressão fornecida.

### 1.2.2 *Scripts* de automatização do fluxo

Dado que a quantidade de circuitos a serem analisados é bastante grande, foi necessário automatizar o processo de geração dos arquivos HDL, verificação da corretude dos circuitos, sintetização da *netlists* e extração das características de cada arquitetura. Para tanto, diversos *scripts* foram desenvolvidos para que cada processo

ocoresse automática e independentemente.

Após a síntese de cada circuito com a ferramenta Cadence Encouter RTL Compiler ©, vários arquivos são gerados com características de potência, área, frequência, entre outros. A partir desses arquivos, as características de cada circuito são extraídas automaticamente através de um *script* e armazenadas em um arquivo CSV para ser uitlizado na análise de dados e atualização do banco de dados.

### 1.2.3 Ferramentas de análise de dados

Com os arquivos CSV criados com as características dos circuitos, duas abordagens são propostas para análise de dados. A primeira abordagem é voltada para compreender o a variação do QoR de cada arquitetura para diferentes constraints. Para tanto, alguns *scripts* em Matlab foram desenvolvidos com a capacidade de gerar gráficos de comparação a partir de uma interface gráfica projetada. Os gráficos utilizados na seção de resultados deste trabalho foram gerados com esses *scripts*.

Já a segunda abordagem remete à criação de um *software* responsável pelo gerenciamento de uma base de dados com as características de síntese extraídas para cada circuito. Essa ferramenta é importante pois permite escolher rapidamente qual arquitetura é mais adaptada para certos requisitos (área, tamanho de dados, etc) através de buscas parametrizadas pelo usuário. Apesar de ter sido desenvolvida para os circuitos aritméticos considerados nesse trabalho, o software é facilmente adaptável para suportar outros circuitos.

### 1.2.4 Análise dos Resultados

Após a síntese e extração das características de cada arquitetura estudada, alguns gráficos comparando a área ocupada, frequência máxima de utilização e potência estática dissipada são apresentados. As análises apresentadas referem-se apenas aos piores casos, isto é, aos circuitos com a maior largura de dados. Com esses gráficos é possível verificar experimentalmente as vantagens e as desvantagens de cada arquitetura. Ainda, é apresentada uma conclusão sobre os circuitos mais adaptados ao processador MPPA-256.

## 1.3 Conclusão

Sabe-se que hoje os processadores manycore são essenciais para obter sistemas de alta performance já que oferecem um poder de processamento paralelo elevado além de mitigar as restrições físicas do processo de fabricação de circuitos integrados, como a limitação da frequência. Para o processador de baixo consumo

MPPA-256, em particular, foi imprescindível estudar o comportamento das arquiteturas dos operadores aritméticos para encontrar formas de otimização do circuito como um todo considerando três aspectos principais – frequência, área e consumo.

A partir dos resultados obtidos do trabalho realizado, percebeu-se que a margem de otmização para somadores é muito pequena já que a abordagem atual apresenta-se mais eficiente em relação às arquiteturas estudadas. Entretanto, o mesmo não é válido para multiplicadores, onde há uma margem considerável para melhorias devido às otimizações arquiteturais apresentadas. Ainda, o sistema construído para armazenamento e consulta às características dos circuitos provou-se eficiente e extensível para outros tipos de circuito.

**Abstract**

Multi-core processors became the most suitable solution to increase the computer performance as the current manufacturing technology is reaching its limits and the unconstrained frequency increasing is no longer possible since it is directly linked to the dissipated power. In this context, there a highly performant and energy-aware many-core processor – the MPPA-256 – developed by Kalray. Consequently, this work describes my internship at Kalray where I explored and analyzed some arithmetic operators' architectures aiming for this type of processor. These arithmetic components are present in almost all processor subsystems from the arithmetic core to the memory addressing, thus its importance to observe which architectures offer the best solutions based on their speed, area and leakage power. The structure of this work is composed as follows: firstly, it is presented the context on which the many-core processors are inserted as well as a theoretical study about the most common adders and multipliers designs. Then, the adopted methodology is described with the developed tools to support perform all analysis followed by the obtained results comparing all studied architectures. Finally, some considerations are made about these results succeeded by some commentaries about the whole internship.

**Keywords:** Many-core processors; arithmetic circuits; low power circuits.

# Glossary

**ALU** Arithmetic and Logic Unit.

**CGS** Constant Group Size.

**CMOS** Complementary metal oxide semi-conductor.

**CSV** Comma-Separated Values.

**EDA** Electronic Design Automation.

**FA** Full adder.

**GFLOPS** Giga Floating-Point Operations per Second.

**HPC** High-Perfomance Computing.

**IEEE** Institute of Electrical and Electronics Engineers.

**LSB** Least Significant Bit.

**MFLOPS** Mega Floating-Point Operations per Second.

**MPPA** Multi-purpose processor array.

**MSB** Most Significant Bit.

**ORM** Object-Relational Mapping.

**PG** Propagate-Generate functions.

**QoR** Quality of Results.

**RDBMS** Relational Database Management System.

**SQL** Structured Query Language.

**TDP** Thermal Design Power.

**VGS** Variable Group Size.

**VHDL** VHSIC Hardware Description Language.

**VLIW** Very Long Instruction Word.

# Contents

# List of Figures

# List of Tables

# Acknowledgement

Firstly, I would like to thank my tutor Nicolas Brunie for all the patience and guidance he offered me during my internship as well as for the opportunity to work on a challenging subject. Here I express my gratitude to Vincent Ray and Jean-Philippe Barbiero for all the support they have given me and the new ideas they have brought to discussion. I am also grateful to my Phelma advisor Katell Morin-Allory for all the help she has given me not only during my internship but also during my stay in France.

I would like to extend my thanks to the people at Kalray for creating such a nice and fun environment to work in.

I am grateful to my parents who forged me into who I am and for the support they have always given me to pursue my dreams.

# 1 Introduction

Historically, the processor throughput was directly tied to the chip frequency and it always increased as new manufacturing technologies at the silicon level were developed, following Moore's Law. The problem, however, is that the computing requirements increase faster than the technology evolution. One recurrent solution is to parallelize the computation process both in hardware and software. Thus, a great number of cores inside a processor can meet those requirements at a lower frequency than that necessary for a single core processor, resulting in a simpler, less power-hungry design. For high throughput applications such as scientific computing, video broadcasting and cloud computing, the trend is to use many-core processors due to their massive parallelism that is well adapted to current processing algorithms.

Since the processor main purpose is to perform computation as fast as it may be able to, a special attention should be given to arithmetic operators since they have a major role in the processor throughput. These circuits are present not only in the arithmetic and logic units but also in several processor components, such as those that control the program execution and memory address calculation. In many-core processors such as the Kalray MPPA-256 which uses simpler cores – without branching prediction logic, for example – these operators have an even bigger impact because the ratio compute/control logic is greater than the one found in more complex processors such as the Intel x86 family. Besides, there is a close tie between the pipeline length of a processor and the size of the embedded multipliers because the latter represents some of the largest and deepest combinatorial blocks due to the number of logic levels necessary to implement them.

Thus, this work targets a study and exploration of arithmetic operators used as basic computing blocks in multicore processors realized at the enterprise Kalray S.A. This report is divided as follows: it presents a context of the many-core processors and the challenges associated to these processors, followed by a theoretical presentation of the most common architectures for adders and multipliers. Then, the methodology of work is presented as well as the tools developed during the internship to allow the analysis and comparison of the implementation of such operators. Consequently, it shows a comparative study of speed, area and leakage power of these architectures. Finally, there are some final considerations about the developed work as well as the knowledge and skills gathered through the internship.

# 2 The context of many-core processors

Currently, huge data-centers and high-performance computing (HPC) clusters play a major role in everyone's lives, directly or indirectly, mainly fueled by the constant increase of the number of internet users and science breakthroughs that try to explain phenomena that are not yet completely understood by the human being, for example. These computer arrays – formed by thousands of processors – have to be constantly upgraded to meet the performance requirements of applications such as weather prediction – often used to create action plans for natural disasters – and oil fields exploration.

Achieving such high-performance figures is an extensive field of research. Due to the inherent complexity linked to the conception and construction of such systems, there are numerous challenges to overcome. According to Kogge et al. [Kog+08, p.2], some of them are:

1. The capability of such system to remain functional;
2. The problems linked to conceiving algorithms that fully explore the resources offered by the available hardware;
3. Storage limits to deal with a high quantity of data; and
4. The ever-increasing need of power to drive such systems.

Until now, the processor performance increased as the transistor shrank, consequence of the expanding number of transistors in the chip according to Moore's Law, and the dissipated power decreased as result of the lower operating voltages. Despite the advantages of lower source voltages, the higher frequencies and the more prominent effects of static power[1] applied to a huge amount of transistors result on a higher thermal design power (TDP[2]). Due to the rise in costs of producing energy for such computing centers and difficulty of building suitable structures that can efficiently manage the generated heat, current trend is to use energy-aware processors to mitigate this problem.

The problem, however, is to create a good throughput/energy ratio. As stated by Singh et al. [Sin+13, p.1], the current trend is to move towards multicore/many-core

[1] Side effect of the CMOS oxide reduced thickness

[2] The maximum amount of power generated by a processor that must be dissipated by the cooling system.

architectures on which the frequency can be reduced to some extent while increasing the number of parallel jobs. Since the dynamic power dissipation in a silicon chip is linearly proportional to the frequency and to the square of the operation voltage, reducing both while replicating the number of functional units may indeed be the best choice.

One of the current promising design is the MPPA-256 many-core processor, presented by De Dinechin et al. [De +13, p.1]. The very long instruction word (VLIW) approach – allowing up to five instructions to be executed in parallel – combined with the 256 cores divided into 16 clusters results in highly parallelized dataflow architecture. Due to the simpler micro-architecture, each core is capable of computing up to 800 MFLOPS[3] at a lower frequency compared to Intel x86 processors, for example. Consequently, this many-core processor achieves an exceptional ratio of 25 GFLOPS per watt. Figure 2.1 shows the MPPA-256 processor architecture. The center of the chip is composed by the 16 processing clusters with the input/output control circuits surrounding it.

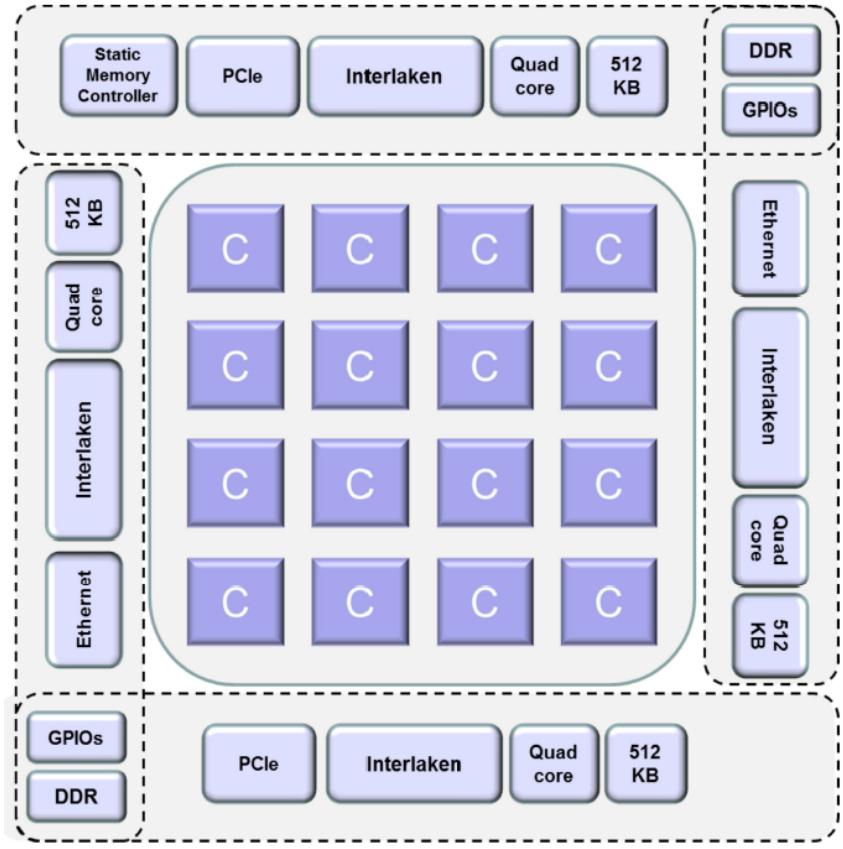


Figure 2.1: Structural view of MPPA-256 architecture [De +13, p.1]

[3]Metric for measuring computer performance based on the number of executed operations in the IEEE-764 Double Precision standard

# 3 Arithmetic Circuits

Many of the current computer applications have an arithmetic-intensive nature such as error checking, encryption and so on. Thus, it very likely that almost every task executed by the processor will somehow traverse an arithmetic component, either directly or indirectly, such as the arithmetic and logic units (ALU), memory address computation block, etc. Designing and optimizing these arithmetic blocks may be a challenging task in order to respect all speed, area and power constraints.

## 3.1 Binary Adders

To motivate the study of the impact that binary adders have in the global circuit, Deschamps, Bioul, and Sutter [DBS06, p.289] states that "[t]wo-operand addition is a primitive operation included in practically all arithmetic algorithms. As a consequence, the efficiency of an arithmetic circuit strongly depends on the way the adders are implemented". This simple arithmetic circuit can further be arranged in such way that it can form multiplier blocks, divisors, and other mathematical operations.

In digital circuits, the binary addition operation is performed by the full adder (FA) cell whose logic function is described in the truth table 3.1.1 where $A$, $B$ are the input operands, $C_{in}$ is the carry-in, $Sum$ is the result of the operation and $C_{out}$ is the output which indicates when there is an overflow in the number representation. Alternatively, a half adder cell can be used. It has rather similar logic functions, although it does not have a carry-in input.

| $A$ | $B$ | $C_i$ | $Sum$ | $C_{out}$ |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 0 |
| 0 | 1 | 0 | 1 | 0 |
| 0 | 1 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 1 | 0 | 1 |
| 1 | 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 1 | 1 |

Table 3.1.1: Full adder truth table

From the truth table above, one possible gate implementation of the full adder is shown in Figure 3.1.1. Although this represents a gate-level representation of the FA,

many alternative implementations can be found at both gate and transistor levels, each one with a specific target, like smaller area, less static power, etc. For this work, the standard full adder cell will be considered as the one shown below since it evaluates in parallel the influence of each entry on the result of the carry-out, resulting in a faster architecture with a smaller critical path, despite the bigger area.

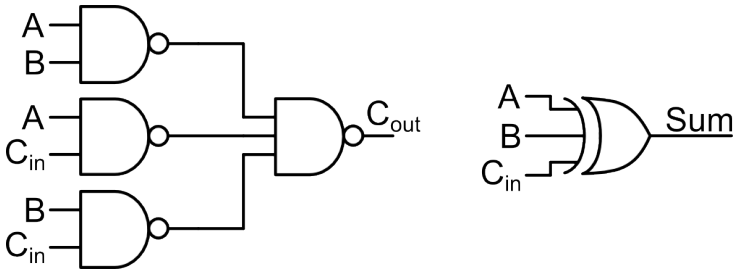


Figure 3.1.1: Full adder cell

### 3.1.1 Ripple Carry Adder

Usually, all arithmetic operations are performed over more than one single bit and, to do so, it is necessary to group full adders to operate over a large bit-width. The simplest approach to accomplish this task is to use the addition scheme used in the paper-and-pen method. The ripple carry adder corresponds to the binary version of this method, sequentially calculating the sum and the carry-out of each bit. From Figure 3.1.2 it is straightforward to observe that the carry-out of the first bit must be passed – or rippled – through all FA cells, until the leftmost bit.

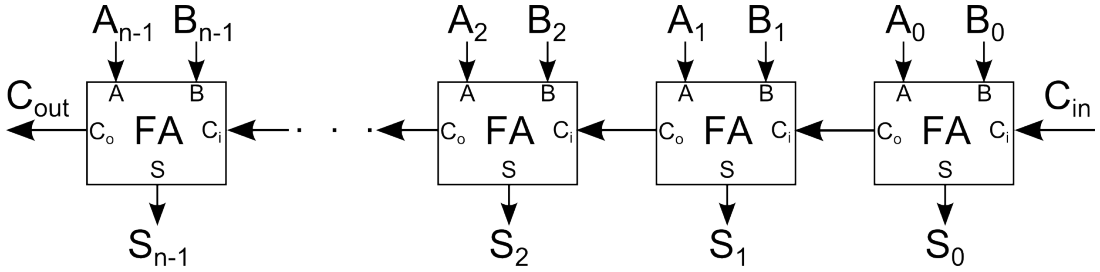


Figure 3.1.2: Structure of a $n$-bit ripple carry adder

Since there is no additional logic to compute the result besides the full adders, this architecture offers the smallest area for a given operand size if compared with any other architecture without timing aspects. To analyze the timing performance of this adder, let us suppose that each logic gate has a delay D. From the previous section, the full adder critical path corresponds to the carry-out logic and it is composed by two logic gates. The first adder will take $2D$ time units to compute the carry. The second adder will have to wait the delay imposed from the first cell and, then, it will spend $2D$ time units to compute its own carry-out. Thus, if this process is extended to $n$-bit inputs, the

total delay at the last cell can be calculated as:

$$T_{delay} = \overbrace{2D}^{C_{O_{n-1}}} + \overbrace{2D}^{C_{O_{n-2}}} + \cdots + \overbrace{2D}^{C_{O_0}} = n \times 2D \tag{3.1.1}$$

From this equation, it is clear, that as the bit-width increases, the time taken to reach the result increases linearly due the fact that each bit depends upon the result of the previous bit. The required area to accommodate such adder may be calculated as follows. Let us calculate the area in terms of number of logic gates, regardless their differences in drive-loading capabilities, number of inputs, etc. So, from the full adder cell proposed in the previous section, it is easy to identify that its area is equal to five logic gates. Then, the cell area occupied by this adder increases linearly with the bit-width of the inputs, as can be seen in the equation below for a $n$-bit inputs, assuming that the area of each logic gate is equal to $A$:

$$T_{area} = \overbrace{5A}^{FA_{n-1}} + \overbrace{5A}^{FA_{n-2}} + \cdots + \overbrace{5A}^{FA_0} = n \times 5A \tag{3.1.2}$$

### 3.1.2 Carry Look-Ahead Adder

Despite its small area, the ripple-carry adder does not scale well in timing as the operand sizes increases, due to the nature of its carry propagation path. Since all the operands' bits are present at the adder input at a given time, some properties can be extracted from the binary addition. Thus, from the full adder truth table it is possible to determine that:

1. When the two inputs are equal to one, a carry-out is generated regardless the value of the carry-in. Thus, it can be mapped into a *carry generate* function that is defined as the logic AND between the operands.

2. When one input is equal to zero and the other is equal to one, the carry-in will be propagated to the carry-out. Thus, it can be mapped into a *carry propagate* function that is defined as the logic XOR between the operands.

The carry look-ahead adder family is based on the functions above and can dramatically reduce the time to complete a sum. Adopting $G_i$ for the generate function for the $i^{th}$ bit and $P_i$ the propagate function for the $i^{th}$ bit, Katz [Kat94, p.264] shows that "sum and carry-out can be expressed in terms of the carry generate and carry propagate functions":

$$S_i = \overbrace{A_i \oplus B_i}^{P_i} \oplus C_i = P_i \oplus C_i \tag{3.1.3}$$

$$C_{i+1} = A_iB_i + A_iC_i + B_iC_i \tag{3.1.4a}$$
$$= A_iB_i + C_i(A_i + B_i) \tag{3.1.4b}$$
$$= A_iB_i + C_i(A_i \oplus B_i) \tag{3.1.4c}$$
$$= G_i + C_iP_i \tag{3.1.4d}$$

The transformation made through Equation 3.1.4b and 3.1.4c is possible because when $A_i$ and $B_i$ are equal to one, the result will be correctly evaluated from the generate function, so there is no need to have both sides of an OR gate with a true input. These functions can then be arranged in a recursive fashion in order to parallelize the evaluation of each input carry which must arrive to the full adder cells. Supposing that the first carry will be given with the inputs, the carry-in for each adder cell can be calculated as follows, for a n-bit inputs:

$$c_0 = C_{in} \tag{3.1.5a}$$
$$C_1 = G_0 + P_0C_0 \tag{3.1.5b}$$
$$C_2 = G_1 + P_1C_1 = G_1 + P_1G_0 + P_1P_0C_0 \tag{3.1.5c}$$
$$\vdots$$
$$C_{n-1} = G_{n-2} + P_{n-2}C_{n-2} = G_{n-2} + P_{n-2}G_{n-3} + P_{n-2}P_{n-3}C_{n-3} = \ldots \tag{3.1.5d}$$

If the functions above were to be implemented as is, it is straightforward to observe that the number of necessary logic gates increases considerably for larger inputs. According to Katz [Kat94, p.265], the bigger the inputs, the more carries will have to be generated and, consequently, OR logic gates with many inputs will be necessary, which imposes a technological problem to build such gates.

Due to the inherent recursive characteristic of the carry look-ahead, this problem can be addressed by using parallel prefix computation approach since all carries “can be computed as a chain of prefix operations” [Kno01, p.278]. This solution leads to a new family of adders on which the computation time has a logarithmical complexity.

#### 3.1.2.1 Brent-Kung Adder

In order to present the new architecture based on a parallel prefix computation, Brent and Kung [BK82, p.261] introduced a new operator "*o*" which is defined in Equation 3.1.6. The final values of the carry propagate and carry generate functions for the $i^{th}$ bit are given $G_i$ and $P_i$ whose definition is found in Equation 3.1.6. The identifiers $g_i$ and $p_i$ denote, respectively, the carry generate and carry propagate functions whose result is

obtained directly from the $i^{th}$ bit of each input operand. It is out of the scope of this work to prove that this set of equations is mathematically correct.

$$(g, p)o(\hat{g}, \hat{p}) = (g + (p\hat{g}), p\hat{p}) \tag{3.1.6}$$

$$(G_i, P_i) = \begin{cases} (g_1, p_1) & \text{if } i = 1 \\ (g_i, p_i)o(G_{i-1}, P_{i-1}) & \text{if } 2 \leq i \leq n \end{cases} \tag{3.1.7}$$

Given that this operator is associative, these functions do not need to be calculated sequentially. This operator can be mapped to a hardware component – henceforth called black cell – whose logic functions are defined in Equations 3.1.8 and 3.1.9. The Brent-Kung adder is based on the fusion of two trees composed by black and white cells – the latter only transmits the results from the previous level to the next one.

$$P_{out} = P_{in}\widehat{P_{in}} \tag{3.1.8}$$

$$G_{out} = G_{in} + (P_{in}\widehat{G_{in}}) \tag{3.1.9}$$

The first tree is constructed from its leaves down to its root, similarly to a binary tree. At the first level, generate and propagate functions of each pair of bits formed by the inputs are computed. Then, at each level, a black cell is inserted, combining a pair of results generated on the previous level until the tree is completely generated up to the last bit. From the properties of the binary tree, it is straightforward to observe that all carries whose position is a power of two will have their final values computed.

Regarding the construction of the second tree, a more complex approach is used. In this case, the tree is constructed from the root to its leaves. The root of this tree is located at the power of two nearest to the center. Then, the tree follows the same algorithm as the first tree. To illustrate this operation, Figure 3.1.3 shows the complete carry tree for a 16-bit wide adder.

From Figure 3.1.3, two properties might be extracted. From this adder design, the number of levels is equal to $2\log_2 n - 1$, which means that the computation time increases logarithmically with the size of the operands. Further, the fanout[1] of each cell is always constant and equal to two, allowing the use of gates with smaller drives[2]. Brent and Kung [BK82, p.262] affirm that the area occupied by this design is quasi-linear, that is, it increases proportionally to $n \times \log n$.

[1] Common notation to refer to the total capacitance that the logic port has connected to its output.

[2] Drive is defined as the capability of a cell to supply the necessary current to charge its total output capacitance.

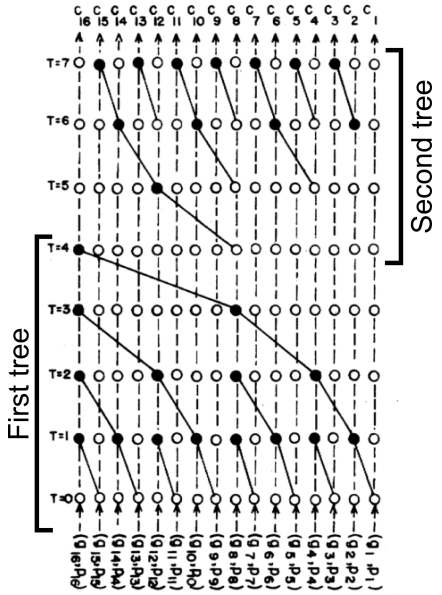


Figure 3.1.3: Carry tree of a Brent-Kung adder adapted from [BK82, p.263]

#### 3.1.2.2 Kogge-Stone Adder

As stated above, the carry look-ahead adder family is defined through a set of recursive logic equations to calculate as soon as possible the carry-in of each full adder cell. Since Equation 3.1.5 has a linear recurrence characteristic and all their input values are well-defined as consequence of the existent PG functions applied to each pair of input bits, they can be solved using the idea of recursive doubling introduced by Kogge and Stone [KS73, p.787]. This design is relatively similar to the Brent-Kung adder shown in the previous section since it also uses black and white cells although their interconnections are slightly different.

As shown in Figure 3.1.4, this architecture seeks to calculate each carry value as soon as possible while keeping a constant fanout for both black and white processors. Consequently, the necessary lateral wiring increases the wiring capacitance, demanding the insertion of buffers to mitigate this problem [Kno01, p.278]. In this design, the parallelism level of carry computation is at its maximum. Theoretically, the Kogge-Stone adder represents the fastest binary adder architecture, with a temporal complexity of $O(\log(n))$; however the massive hardware replication and high quantity of interconnections induce routing problems such as multiple metal layers and higher capacitance due to the number of grouped wires, degrading the final timing.

Among the most common variations of this design, there are those that consider higher-radix processors – more than two inputs per cell, thus reducing the number of cells – and those based on a sparse tree, calculating fewer carries that are used as inputs for other adders in a mixed-architecture scheme. If only the connections and processors with a red line were left in Figure 3.1.4, this architecture would illustrate a

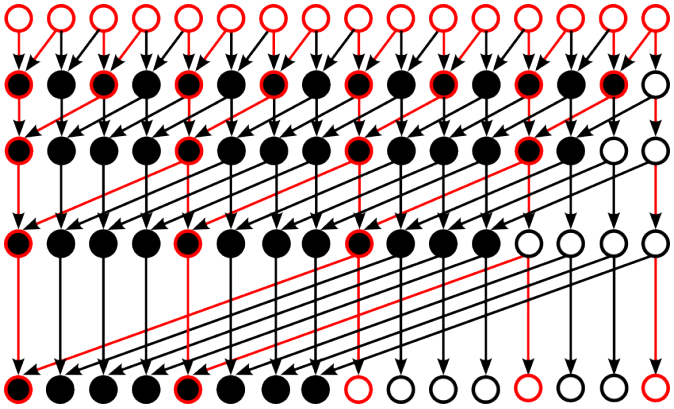

Figure 3.1.4: Carry tree of a Kogge-Stone adder

sparsity-4 Kogge-Stone adder which consumes far less area than the classic version with a timing penalty.

### 3.1.3 Carry Select Adder

The carry select adder mixes the block-based approach with the conditional sum principle in order to pre-compute slices of the final result, thus reducing the time taken to perform the operation. This adder relies on the duplication of each adder block in order to compute the two carry-in possibilities. Then, a multiplexer selects which block output will be used as the adder output. Consequently, the critical path becomes the logic to select the output in addition to the propagation delay of the first adder block whose carry-out will be the control signal driving the selection circuit, as seen in Figure 3.1.5.

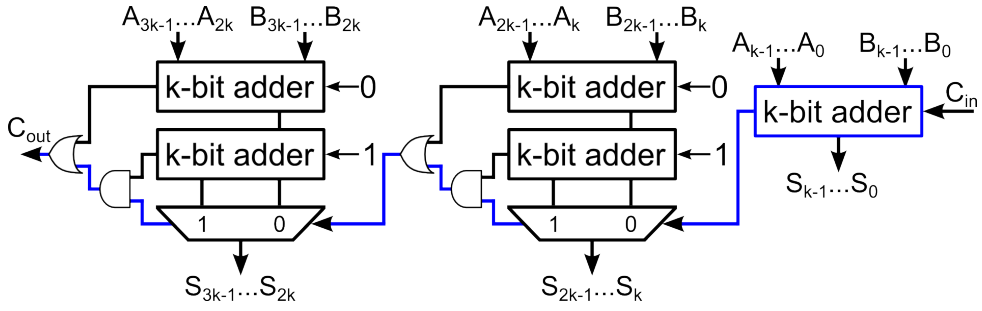


Figure 3.1.5: Structure of a fixed-group size carry-select adder

Tyagi [Tya93, p. 1163] shows that a carry-select adder performs the computation in a time proportional to $O(n^{1/2})$ which is much faster than the linear complexity found in the ripple-carry adder. Variable block sizes offer better performance when compared to designs with fixed block sizes, especially if the gate delays of the targeted technology are well-known in advance. In this case, each block size can be adjusted according to the length of the selection circuit, calculated from the adder start up to the beginning of

the current block, matching both propagation times – ripple-carry group and selection chain – and effectively reducing the computation delay.

### 3.1.4 Carry Skip Adder

Based on the ripple-carry structure, the carry-skip adder proposes a chain division into small ripple-carry groups where the critical path might be broken. Considering that all inputs arrive at the same time, the propagate function associated to each pair of bits that compose each group stage can be calculated. Clearly, the group carry-in will be transmitted to the carry-out only if all group stages propagate the signal. Thus, a group propagate function may be defined as a logical AND of the propagate function of each stage.

To better understand the carry-skip architecture, consider a 12-bit adder with 4-bit ripple carry groups as shown in Figure 3.1.6. Each group has a group propagate function "P" that is determined from its inputs. If, for example, the function P of the second group is true, the carry-in of the third group will be equal to the carry-out of the first group, thus bypassing the carry computation of the second group. This process can be extended for any adder size of arbitrary group sizes.

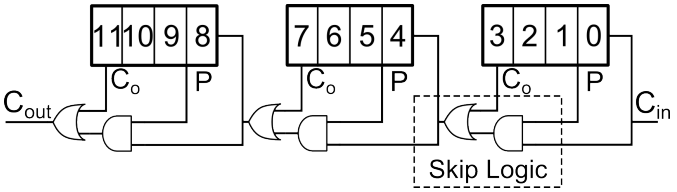


Figure 3.1.6: Structure of a carry-skip adder

Parhami [Par00, p.109] shows that a 32-bit carry-skip adder with optimal block size is approximately 2.5 times faster than a ripple carry adder of the same size. The block size has an important impact on the adder timing. To better explore this architecture, Deschamps, Bioul, and Sutter [DBS06, p.301] proves that variable block sizes can increase the performance by a factor of $\sqrt{2}$.

## 3.2 Parallel Binary Multipliers

The number of applications designed for digital image and signal processing has been increasing in the last years. It is widely known that these applications are heavily based on multipliers as, for example, they use digital filters to obtain the desired result. Besides, with the adoption of high resolution images and signals, the arithmetic operations have to be completed as fast as possible to satisfy the requirements. These multiplications are normally performed either through a software algorithm based on successive additions or through a multiplication-dedicated hardware (multiplier). This

chapter presents some well-known architectures of the former, detailing the characteristics of each one.

In order to understand the binary multiplication, let's assume two numbers, $A$ and $X$, that are, respectively, $m$- and $n$-bit wide and $P_0, P_1 \ldots P_{n-1}$ the partial products of $A$ times $X$, like the pencil-and-paper method. These partial products are generated from the logical operation AND between the multiplicand $A$ and each bit of the multiplier $X$, that is, the partial product will be equal to $A$ when the current bit of $X$ is one and it will be equal to zero when the current bit of $X$ is zero. Next, these partial products are aligned according to their bit-weight, shifted one position to the left at a time. Then, these products are summed accordingly, obtaining the desired result. The partial products and their bits can be represented with a dot diagram to better understand the design of the multiplier. Each dot represents a bit of a vector regardless its value. Figure 3.2.1 shows a simple dot diagram for a 16-bit multiplication of the algorithm described above where each row corresponds to a partial product.

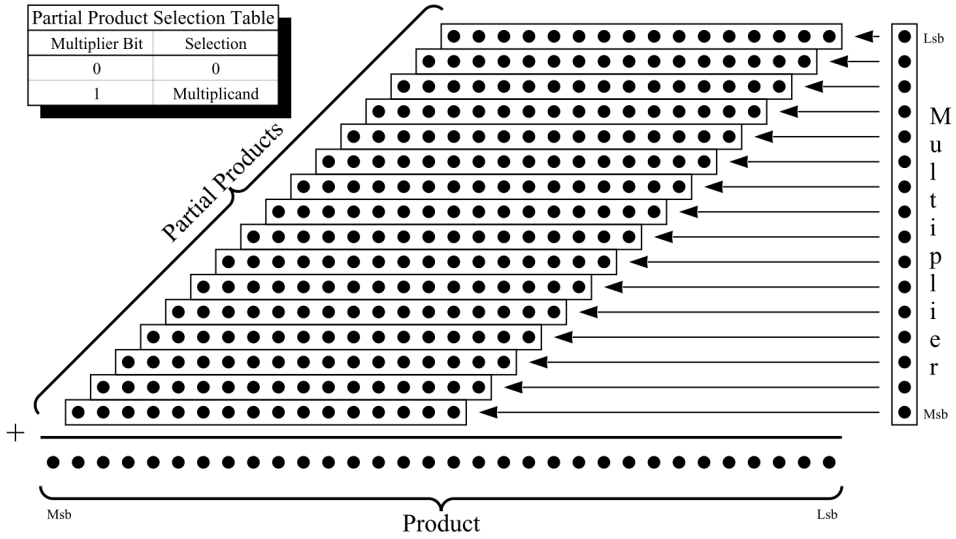


Figure 3.2.1: Example of a classic binary multiplication [Bew94, p.14]

### 3.2.1 Unsigned multipliers

The unsigned multipliers considered in this section do not use any encoding for the operands. Consequently, their structure is solely conceived regarding the disposition of the adder cells in order to sum the various partial products. These multipliers are based on trees of adders and perform the operation in one clock cycle. This approach is only possible due to the advances on the fabrication technology which is currently small enough to fit a complete multiplier into a chip without occupying most of its area.

#### 3.2.1.1 Wallace Multiplier

Following the shift-add algorithm exposed above, it can be seen that sometimes many binary numbers have to be added at the same time. The main problem of this method is the carry propagation in each addition which has a highly negative impact on the circuit timing. To address this issue, Wallace [Wal64, p.14] suggests an adder tree structure that uses a redundant representation called carry-save that operates without carry propagation, resulting in a much faster multiplier. Wallace's multiplier latency is logarithmically proportional to the number of partial products.

In the classic carry-save scheme, the tree is composed of half and full adders, always generating two outputs per compression level until there are only two rows of partial products to sum. At the end, a carry propagate adder has to be inserted to transform the redundant representation into a single number. Considering this scheme, the algorithm to generate such tree is as follows:

1. Take any group of three bits with the same weight and sum them using a full adder. If there are more bits of the same weight, group them with either a full or a half adder.
2. The sum bit will have the same weight – let's say $i$ – as the adder cell inputs while the carry-out bit will have a higher weight, let's say $i + 1$.
3. If there is only a single bit left for a current weight, transfer it to the next level.
4. Repeat the steps above until there are no more than two bits left for any weight.

Alternatively, the tree might contain 4:2 compressors which are based on the different latency of the full adder outputs to further improve the multiplier timing. This cell compresses four bits into two bits. One particularity of this design resides on the use of fast carry-in and carry-out to interconnect side-by-side multiple cells. Figure 3.2.2 shows the architecture of such cell. Assuming that the XOR gate delay is higher than the one of the NAND gate, the latency of this component is equal to three XOR gates whose path is shown in blue in Figure 3.2.2.

The classic algorithm is represented in Figure 3.2.3 for a $8 \times 8$ multiplier. Each blue rectangle represents either a full or a half adder, generating bits for the next compression level in the tree. From this representation, it is clear that the tree critical path is composed by four full adders since there are four compression levels before the carry propagating adder, each containing at most one full adder in its critical path. This tree can be optimized to reduce the number of half adders, though the carry propagate adder size is increased.

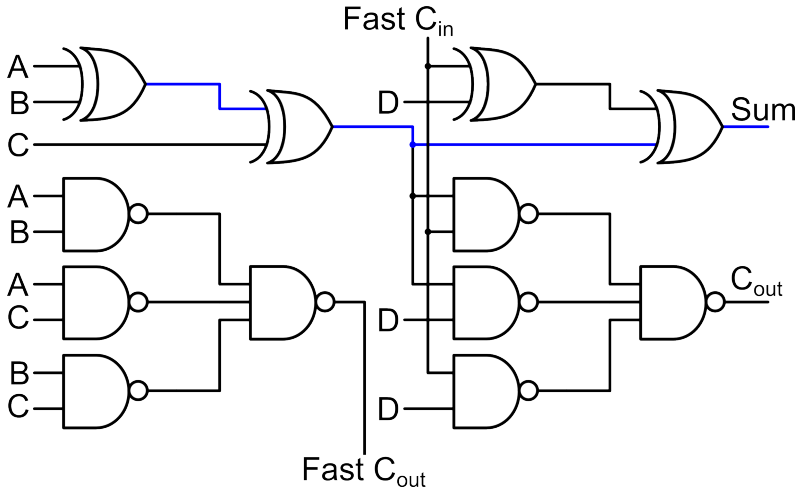


Figure 3.2.2: Schematic of a 4:2 compressor cell

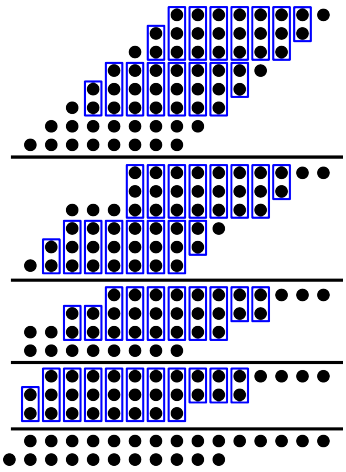

Figure 3.2.3: $8 \times 8$ Wallace multiplier reduction tree

#### 3.2.1.2 Dadda Multiplier

The Dadda multiplier uses a slightly different approach to distribute the adder cells in a carry-save fashion to sum the partial products. This multiplier reduces the number of operands in each stage using as few as possible half and full adders. Dadda [Dad65, p.122] proposes an algorithm on which the maximum number of summands in each stage follows a geometric progression whose common ratio is defined in function the compression ratio of the available adder cells. For instance, this ratio is equal to $^3/_2$ when the largest adder cell is a full adder since it compresses three bits into two, resulting in the following sequence, considering only the integer part of the result:

$$\overbrace{2}^{\text{Stage n}}, \underbrace{3}_{\text{Stage n-1}}, \overbrace{4}^{\text{Stage n-2}}, \underbrace{6}_{\text{Stage n-3}}, \overbrace{9}^{\text{Stage n-4}}, \underbrace{13}_{\text{Stage n-5}} \ldots \tag{3.2.1}$$

Figure 3.2.4 shows how this algorithm works in order to create the compression

tree for an $8 \times 8$ multiplier. Each dot represents a bit and the blue boxes represent either a full adder or a half adder. In the first level of compression, the longest column has eight bits, thus it has to be reduced to six bits to obey the progression shown above. For the subsequent levels, the same approach is used. Compared to the Wallace multiplier, this scheme uses less adders, resulting in a 26% smaller compression tree for an $8 \times 8$ multiplier. However, the final propagating adder size grows from 11 to 14 bits.

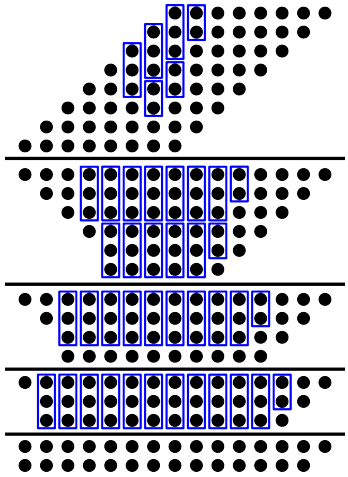

Figure 3.2.4: $8 \times 8$ Dadda multiplier reduction tree

### 3.2.2 Signed Multipliers

Most of the arithmetic operations are performed using signed numbers, generally represented using two's complement. This representation has been vastly studied and its theory is out of the scope of this work. The main difference between signed and unsigned multipliers is the partial products generation because most significant bits of both operands have a sign value. Once the partial products are correctly generated, they may be added using the algorithms for unsigned multipliers presented earlier.

#### 3.2.2.1 Booth Multiplier

A first approach to address the signed multiplication was the Booth algorithm on which "binary numbers of either sign may be multiplied by a uniform process which is independent of any foreknowledge of the sign of these numbers" [Boo51, p.1]. In this technique both the multiplicand and the multiplier are represented in two's complement. The partial products are generated by analyzing a group of two bits of the multiplier, which may reduce the number of these summands.

Despite the benefits of this algorithm, it was not well adapted to the hardware implementation. Therefore, Macsorley [Mac61, p.74] proposes an optimization of this algorithm by analyzing groups of 3 bits that, in two's complement, results in the multiples

$\{0, \pm M, \pm 2M\}$ of the multiplicand. For the first 3-bit window, the least significant bit will always be zero while the upper bits will come from the multiplier's least significant bits. The remaining windows are left-shifted twice from last group, thus the LSB from the current window will be the same as the MSB from the previous group. If the number of bits of the multiplier is odd, a zero may be inserted into the MSB of the last group to fill the analysis window. Consequently, the number of generated partial products will be equal to $\lceil n/2 \rceil$ if $n$ is taken as the size of the multiplier. This encoding is illustrated in Table 3.2.1.

| $X_{2j+1}$ | $X_{2j}$ | $X_{2j-1}$ | Partial Product Result |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1×Multiplicand |
| 0 | 1 | 0 | 1×Multiplicand |
| 0 | 1 | 1 | 2×Multiplicand |
| 1 | 0 | 0 | -2×Multiplicand |
| 1 | 0 | 1 | -1×Multiplicand |
| 1 | 1 | 0 | -1×Multiplicand |
| 1 | 1 | 1 | 0 |

Table 3.2.1: Modified-Booth codification of partial products

To avoid creating partial products with the size of the final product due to the sign extension, Bewick [Bew94, p.138] proposes a technique on which all partial products are supposed to be negative and, due to the properties of two's complement representation, only three extra bits are needed for the first partial product and two bits for the others. For all the partial products except the first, the most significant bit (MSB) is a constant equal to one followed by the sign extension bit. For the first row, the MSB is the sign extension followed twice by its complement. The sign extension computation is calculated as follows:

1. If the partial product is equal to zero (the first and the last value of Table 3.2.1), the sign extension bit is equal to one.

2. For the other cases, the sign bit is calculated from a XNOR gate whose inputs are $X_{2j+1}$ and the most significant bit of the multiplicand. This procedure ensures that the right sign will be propagated according to the sign of the multiplicand and the sign of the desired partial product.

As a further optimization, Farooqui and Oklobdzija [FO98, p.II-262] propose a pre-calculation of all sign bits, leading to a constant $"101010\ldots01011"$, thus removing the constant from the partial products. As a secondary effect, the first partial product is reduced to only a single sign extension bit, without its complements. Then, this constant

is considered as a new partial product with its least significant bit (LSB) aligned to the sign extension bit of the first partial product.

However, these techniques do not address the problem inherent to the subtraction in two's complement where a constant one has to be added to correct the result. Själander and Larsson-Edefors [SL08, p.2] introduce a solution where the impact of this constant is pre-calculated over the LSB of each partial product and a new single-bit term is added to the right of the LSB of the next partial product as shown in Figure 3.2.5. The pre-computation functions are defined from the equations below. To simplify the representation, they are not optimized for the CMOS inverted logic which could further reduce the delay:

$$p_{LSBi} = y_0(x_{2i-1} \oplus x_{2i}) \tag{3.2.2}$$

$$a_i = x_{2i+1}(\overline{(x_{2i-1} + x_{2i})} + \overline{(x_{2i-1} + y_{LSB})} + \overline{(x_{2i} + y_{LSB})}) \tag{3.2.3}$$

| | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | $y_7$ | $y_6$ | $y_5$ | $y_4$ | $y_3$ | $y_2$ | $y_1$ | $y_0$ |
| | | | | | | | | $x_7$ | $x_6$ | $x_5$ | $x_4$ | $x_3$ | $x_2$ | $x_1$ | $x_0$ |
| | | | | | | | $\overline{p_{80}}$ | $p_{70}$ | $p_{60}$ | $p_{50}$ | $p_{40}$ | $p_{30}$ | $p_{20}$ | $p_{10}$ | $p_{LSB0}$ |
| | | | | | $\overline{p_{81}}$ | $p_{71}$ | $p_{61}$ | $p_{51}$ | $p_{41}$ | $p_{31}$ | $p_{21}$ | $p_{11}$ | $p_{LSB1}$ | | |
| | | | $\overline{p_{82}}$ | $p_{72}$ | $p_{62}$ | $p_{52}$ | $p_{42}$ | $p_{32}$ | $p_{22}$ | $p_{12}$ | $p_{LSB2}$ | | | | |
| | $\overline{p_{83}}$ | $p_{73}$ | $p_{63}$ | $p_{53}$ | $p_{43}$ | $p_{33}$ | $p_{23}$ | $p_{13}$ | $p_{LSB3}$ | | | | | | |
| 1 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | $a_3$ | | $a_2$ | | $a_1$ | | $a_0$ | |
| $s_{15}$ | $s_{14}$ | $s_{13}$ | $s_{12}$ | $s_{11}$ | $s_{10}$ | $s_9$ | $s_8$ | $s_7$ | $s_6$ | $s_5$ | $s_4$ | $s_3$ | $s_2$ | $s_1$ | $s_0$ |

Figure 3.2.5: Modified-Booth multiplier with sign extension and LSBs pre-calculation [SL08, p.2]

#### 3.2.2.2 Baugh-Wooley Multiplier

Similarly to the Booth multiplier, this scheme also considers both multiplicand and multiplier to be informed in two's complement representations, although all the partial products generated by the Baugh-Wooley algorithm are positive, resulting in a simpler hardware [BW73, p.1045]. To understand this algorithm, let's assume $A$ and $X$ as the multiplicand and the multiplier, respectively, of arbitrary sizes. Thus, Equations (3.2.4)

and (3.2.5) show the mathematical representation of both numbers:

$$A = -a_{m-1}2^{m-1} + \sum_{i=0}^{m-2} a_i 2^i \tag{3.2.4}$$

$$X = -x_{n-1}2^{n-1} + \sum_{i=0}^{n-2} x_i 2^i \tag{3.2.5}$$

$$P = A \times X \tag{3.2.6}$$

$$P = x_{n-1}a_{m-1}2^{n+m-2} + \sum_{i=0}^{n-2}\sum_{j=0}^{m-2} x_i a_j 2^{i+j} - x_{n-1}2^{n-1}\sum_{i=0}^{m-2} a_i 2^i - a_{m-1}2^{m-1}\sum_{i=0}^{n-2} x_i 2^i \tag{3.2.7}$$

Equation (3.2.7) is obtained by applying the multiplication distributive property to Equation (3.2.6). Consequently, there are two subtractions to be performed which leads to a heterogeneous and less efficient design. Baugh and Wooley [BW73, p.1046] show that this issue can be addressed with the negation property in the two's complement representation. Thus, after some substitutions and algebraic operations, the subtraction terms can be transformed into the pattern of Equation (3.2.8) to use only adders, resulting in a homogeneous circuit.

$$2^{n-1}\left(-2^m + 2^{m-1} + \bar{x}_{n-1}2^{m-1} + x_{n-1} + \sum_{i=0}^{m-1} x_{n-1}\bar{y_i}2^i\right) \tag{3.2.8}$$

Reordering the partial products, Hatamian and Cash [HC86, p.511] propose a very regular structure to dispose the partial products as seen in Figure 3.2.6. This scheme generates $n + m$ summands to be further reduced by a carry-save tree. Compared to the Booth multiplier, the number of partial products is duplicated, although the hardware is simpler since there is neither analysis nor encoding of the input data.

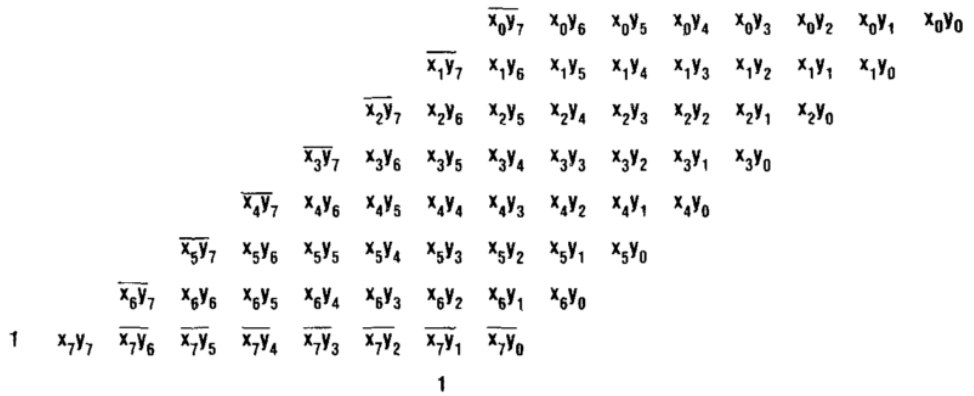


Figure 3.2.6: Partial products layout for the Baugh-Wooley algorithm [HC86, p.512]

# 4 Methodology

Once all the target architectures are studied, it is necessary to implement and synthesize them to compare to the current solution used in the MPPA-256 processor. This analysis is based on the Quality of Results (QoR) obtained from each one of these architectures. The QoR considered in this work is a performance indicator that comprehends mainly the speed, area and power consumption of each circuit. It may be extended to number of logic gates and other indicators that may give an idea about the routing challenge of a given circuit.

Thus, some tools were developed to support the conception of such architectures as well as to extract and classify the synthesis data. These designs are synthesized under specific data sizes and clock periods constraints to standardize the comparison of architectures of the same type.

## 4.1 Design parameters

To investigate the performance and characteristics of every studied architecture, some parameters have to be set to define the constraints of each design. Since this work targets the MPPA-256 manycore processor, the input and output data sizes were defined after an analysis of the processor functionalities that incorporates any of the arithmetic circuits discussed here. The adders data sizes were based on the regular design of an arithmetic and logic unit as well as the counters distributed along the processor to control program execution, memory addresses, etc. The multipliers sizes, however, were chosen based on the double precision floating-point multiplication.

It is widely known that the area and leakage power are related to the speed at which the circuit operates. Then, given the specification of the processor and the limits of the technology, an operating frequency range was defined for each type of circuit to better understand the impact of latency constraint on the QoR of each architecture. Since higher frequencies correspond to smaller periods, the frequency ranges were defined in function of the latter through a logarithmic-spaced vector. Consequently, the results are more fine-grained as the lower bound range is reached.

## 4.2 Manual crafting of arithmetic designs

To start the exploration of arithmetic architectures, the initial designs were described manually, using all the resources embedded in the VHDL language such as generic data sizes and code generation loops. Despite the limitations associated to the language, this approach is straightforward for simple designs, such as the most basic adders targeted by this work.

## 4.3 The VHDL code generator for automatic design generation

As seen earlier, certain architectures follow a recursive mathematical equation. This property is not easily described in VHDL when the data sizes are generic. Even though the latest version of the VHDL language supports recursive-defined designs, the available interpreters integrated into the current electronic design automation (EDA) tools do not process efficiently this type of design. Consequently, it leads to a poorer design result, normally having a worse timing and bigger area if compared to a similar non-recursive design.

To address this problem, a VHDL code generator – called VHDLGen – was developed. With this tool, the design definition complexity is moved to another working space, with a higher abstraction level, on which this task is more easily accomplished. For example, recursive architectures algorithms can be completely solved before the code generation. Consequently, the files created by this generator are considered to have a "flat VHDL description" since all components' instances and specification are in the same file in such way that the synthesis tool does not need to perform any processing on the VHDL code. The main drawback of removing all generics from the VHDL code is the need of many files to describe the same design for different data sizes. Figure 4.3.1 shows the generation flow of a given architecture to be implemented.

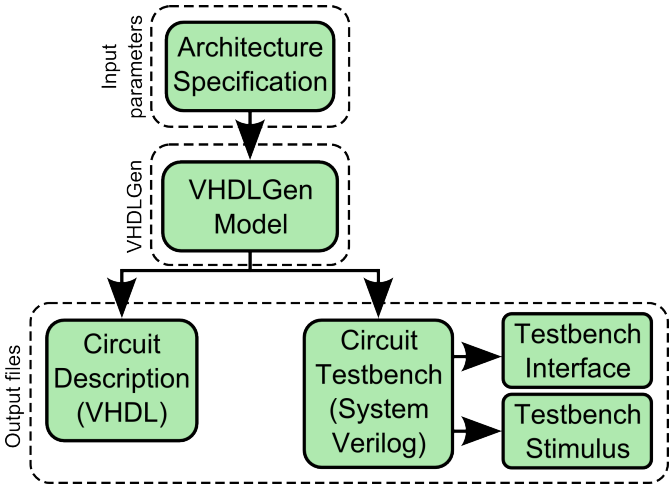


Figure 4.3.1: Circuit generation flow of the VHDLGen

The generator was developed using the objected-oriented language Python. This approach makes easier the construction of a modular system which is less cumbersome to extend. Figure 4.3.2 shows a simplified version of the system class diagram with all the basic structures to construct a proper VHDL design. Here, all attributes were suppressed for legibility and the testbench functionality is not shown since it is an additional system module.

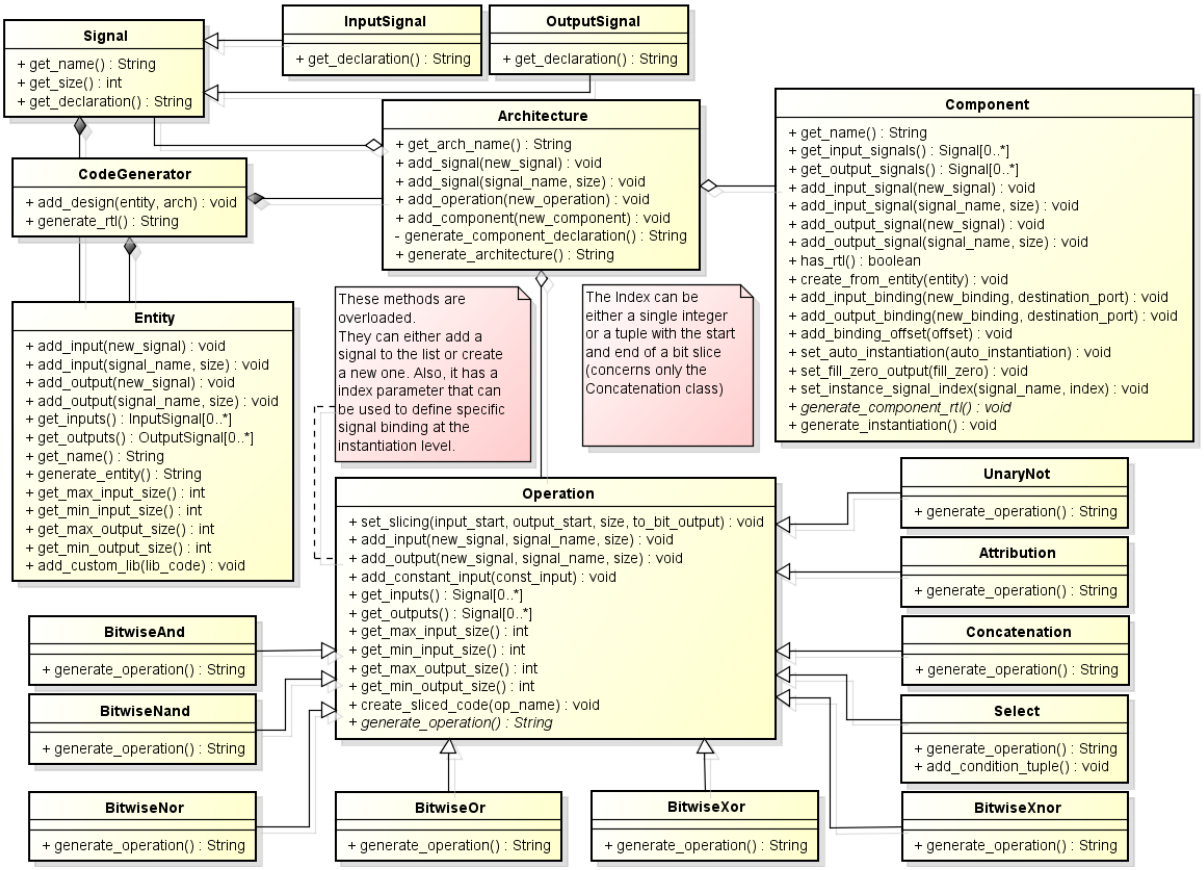


Figure 4.3.2: Simplified class diagram of the VHDL code generator

Each design has an entity – with its inputs and outputs ports and its name – and an architecture on which the functionalities are described. To the architecture object, it can be added intermediary signals, components already defined – for example, the full adder cell – and elementary bitwise operations supported by the VHDL syntax.

Besides, VHDLGen is capable of generating the necessary test bench files which implement the design verification. It requires a list of input and output signals as well as the operations that should be performed with these signals to obtain the desired result. These operations should be described using a SystemVerilog syntax due to the nature of the output files. The verification language was chosen based on the environment on which the designs will be tested as well as on its object-oriented characteristic that makes it easier to write complex tests.

The testbench interface file is a wrapper module written in SystemVerilog, allow-

ing the communication between a VHDL-described circuit and a SystemVerilog test module. It is responsible for translating and transmitting the data from one module to the other. Differently, the testbench stimuli file embeds a high-level description of the operation implemented by the design, used as the reference model, as well as a random data generator. Consequently, it creates the input data fed to the circuit, then it reads and evaluates the data gathered from the design through the interface.

Figure 4.3.3 illustrates the communication flow among these files. The stimuli are generated and then translated before being sent to the circuit through the simulation bus offered by the verification tool. Then, the circuit reacts to the new inputs and sends back the output though the bus. These outputs are translated back and then compared to the reference model to check the correct behavior.

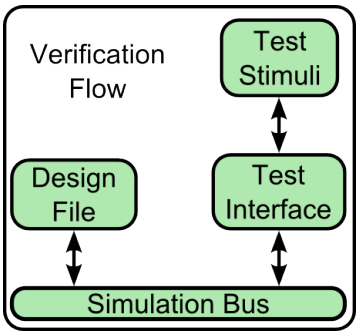


Figure 4.3.3: Communication among modules in the verification step

## 4.4 Automatic circuit verification, synthesis and data extraction

With the code generator in place and with the architectures described for the pre-selected data sizes, it is necessary to ensure the design correctness. Due to high number of designs to verify, it is necessary to develop a system to automatically verify the circuit based on the test bench files generated from the VHDLGen code generator. To do so, a Bash script is used to compile and execute the test bench together with a Ruby script to parallelize the verification. The algorithm adopted for this approach is as follows:

1. List all available designs and launch the Bash script for all of them;
2. For each design, compile the test bench files and the circuit description into the correct libraries;
3. The SystemVerilog stimuli file generates random input values and feeds them to the circuit inputs. The comparison result between the circuit and the verification model is captured by the Bash script

Once the design is guaranteed to be functional, it can be synthesized to obtain the circuit QoR. The Bash script developed for this task creates a list of the designs to synthesize for either a specific clock frequency or a list of clock constraints. For each pair design file-clock period, the script calls the synthesis tool – in our case, Cadence Encounter RTL Compiler – to compile and map the circuit into technology gates. Subsequently, with all circuits synthesized for their frequency range, another script parses all the generated reports, gathering the relevant information. From the filtered data, a plain-text file is filled respecting the comma-separated values (CSV) standard.

## 4.5 Data analysis

### 4.5.1 MATLAB for architecture characterization

After the CSV files generation, it is necessary to perform a deep analysis of the implemented architectures to understand how they evolve relatively to their data sizes and clock constraints. Graphs have proven to be the most straightforward way to do such analysis. Based on these assumptions, a MATLAB script was developed to process these files and then generate graphics that allows comparisons of architectures.

For a more user-friendly experience, MATLAB offers a graphical user interface which might be used to create interactions between the user and the computing script. Using this functionality, the script shows the user which types of architectures are currently available. This feature is important since there is no sense in comparing different architecture types.

Further, the user is able to choose whether or not individual plots will be generated for each architecture, looking for a better comprehension of the design such as the cell area growth rate complexity. The last analysis feature in this script is the comparison of architectures of the same data size for a given characteristic such as timing slack[1], cell area, etc.

### 4.5.2 Database manager as a tool for the decision-taking process

The combination of various architectures of different data sizes synthesized for different clock constraints form a vast catalog of possibilities that may help the decision-taking process of which architecture should be used to implement a given functionality. However, this catalog is only useful when the information is properly classified, allowing a quick analysis. From this principle, a system based on a MySQL server central

[1] Difference between the required time needed by the circuit to execute its operation and the time given to do so.

database to store the data with a front-end application capable of updating and querying this database is proposed.

Storing the gathered data in a MySQL database has many advantages since it is a relational database management system (RDBMS), that is, the database is composed of tables of rows where each row has a unique key. With this approach, tables can be linked to – or have relationships with – other tables, thus the relational concept. Besides, MySQL has support for transaction-based data processing which ensures the data integrity because data modifications are only committed if there was no error during the operation.

As the information to be stored is well-defined, the normalization rules may be applied, trying to obtain the best relational structure which maximizes performance while preventing data inconsistency and minimizing redundancy [Ken83, p.1]. The relational model of this database is shown in Figure 4.5.1.

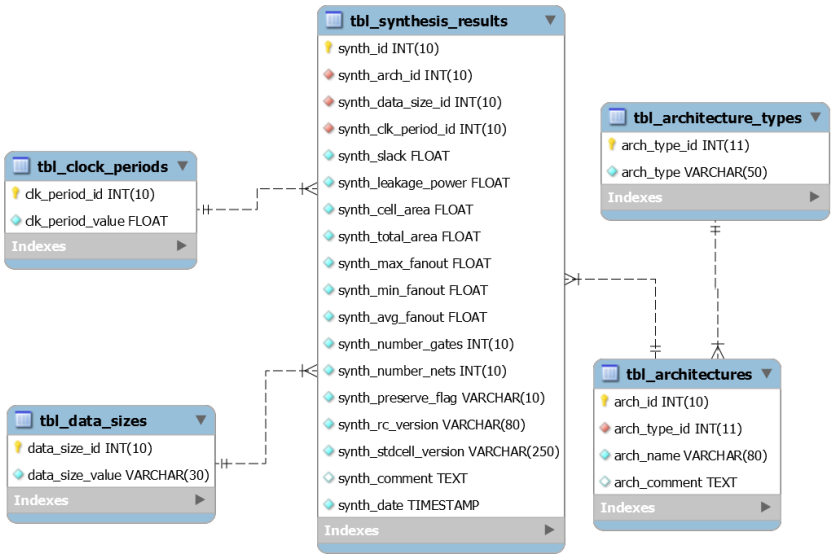


Figure 4.5.1: Entity-relationship model of the MySQL database

Despite the advantages of using a RDBMS, it is cumbersome to manually write SQL statements to insert registries into the database. This problem was solved by developing a database manager in Python which is called DBManager. From the CSV files generated from the synthesis data extraction, this program is able to insert the gathered data into the database and it also offers an interface to inquire the MySQL server about results that respects some parameters defined by the user.

To better integrate Python with MySQL, the SQLAlchemy framework was used since it maps each table of a database into a Python class where the attributes correspond exactly to the table fields. This approach, also known as Object-Relational Mapping (ORM), has many advantages, especially regarding structure changes in the database which are easily integrated into its corresponding class without having to

rewrite all the query and insertion procedures. Consequently, this framework is able to create automatically SQL statements, which is very useful and time-saving when many tables are linked and the expected result is obtained from the union of such datasets.

The class diagram shown in Figure 4.5.2 represents the simplified application core, without the user interfaces and the details of the data access layer. The DATABASEENGINE class follows a composite design pattern since it is responsible to handle the connection to the database through the SQLEngine as well as to create a new session for every transaction. For every query or insertion operation, a new session is created, then processed and closed by the SESSIONMANAGER class. Even though more transactions can be grouped, the chosen approach was considered the best choice since it is safer regarding the concurrent access despite the possible performance degradation.

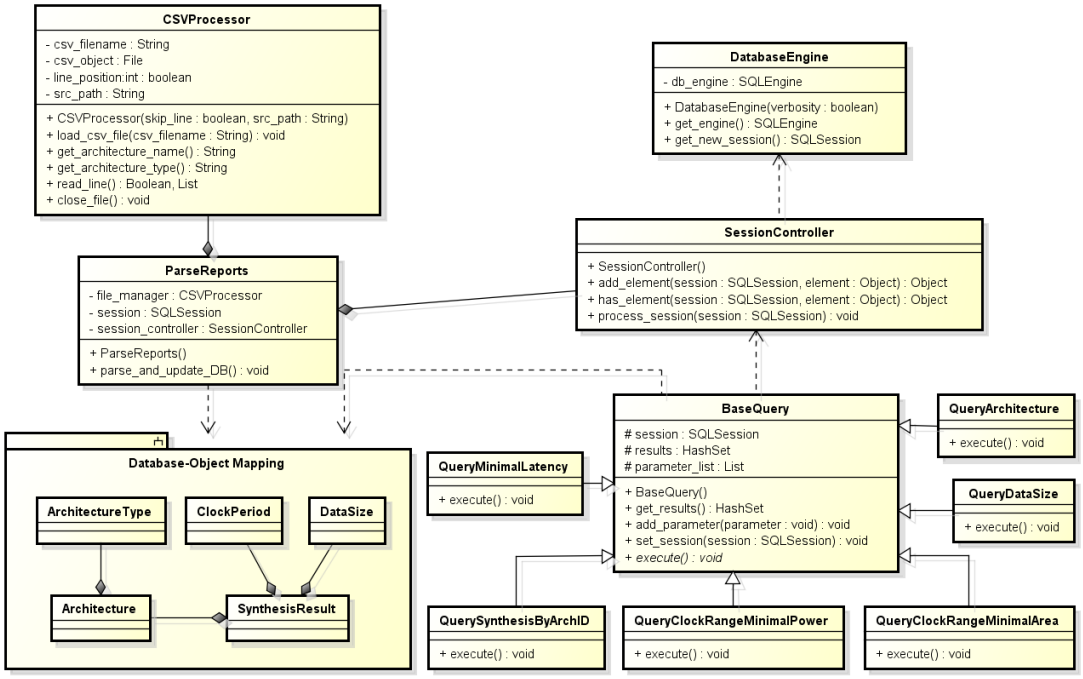


Figure 4.5.2: DBManager simplified class diagram

All the processes of reading and processing the CSV files are performed by the CSVPROCESSOR class whose outputs are then inserted into the database from within the PARSEREPORTS class. Once this data is registered into the tables in the MySQL server, any kind of queries can be executed. For the purposes of this work, the proposed queries shown in the previous diagram satisfy the system requirements. Due to the adopted development methodology, it is clear that this system could be extended withou great effort to store other circuits' characteristics.

# 5 Experimental results and analysis

Once the adders and multipliers studied in Sections 3.1 and 3.2 were implemented and synthetized using standard cells of a CMOS 28nm technology, the QoR evaluation of the architectures could be performed. Thus, this section makes an analysis of timing, cell area and leakage power of each design, comparing it to other architectures of the same type. For all design hereby considered, the Cadence® Encounter RTL Compiler™ was used as the synthesis tool. Further, it presents briefly the advantages and drawbacks of some architectures which stand out for having a distinct characteristic.

Due to the huge amount of data generated from all the architectures and data sizes, only the worst-case scenarios are presented in this section. A more detailed analysis of the studied architectures can be found in the appendices, focusing in those designs with a group-based approach.

## 5.1 Analysis and comparison of adder architectures

To have a reference model to compare the performance between the studied architectures and the current implemented adders in the MPPA processor, the latter is modeled as a simple addition offered by VHDL which is, then, automatically synthetized by the synthesis tool. Thus, it is not possible to define exactly to which architecture it refers to as the synthesis tool adapts it to meet all imposed constraints. Before any analysis, it is necessary to explain some legends in the graphs:

- For adders with the CGS (constant group size) acronym, their groups have a fixed width of 4 full adder cells.
- For adders with the VGS (variable group size) acronym, their groups have a variable width following a linear progression with a constant ratio of 0.25 and an initial group size of two adders.
- For adders with the PG (propagate-generate) acronym, the propagate and generate functions are built-in in order to accelerate the carry computation.

To start this study, Figure 5.1.1 shows the timing evolution of some selected architectures. Despite the equally-spaced clock period values in the graph, they represent a logarithmic-spaced vector in a semi-log graph for better comprehension. All synthesis data considered for these adders refer to 64-bit wide inputs, representing the processor

biggest adder. Observing this figure, we see that the ripple-carry adder does not meet its critical path timing even at the largest clock period considered. Clearly, this adder is very inefficient for bi adders due to its linear dependency to the data size. It is worth noting that with simple modifications, the carry-skip adder leads to a much faster design than its ripple-carry counterpart.

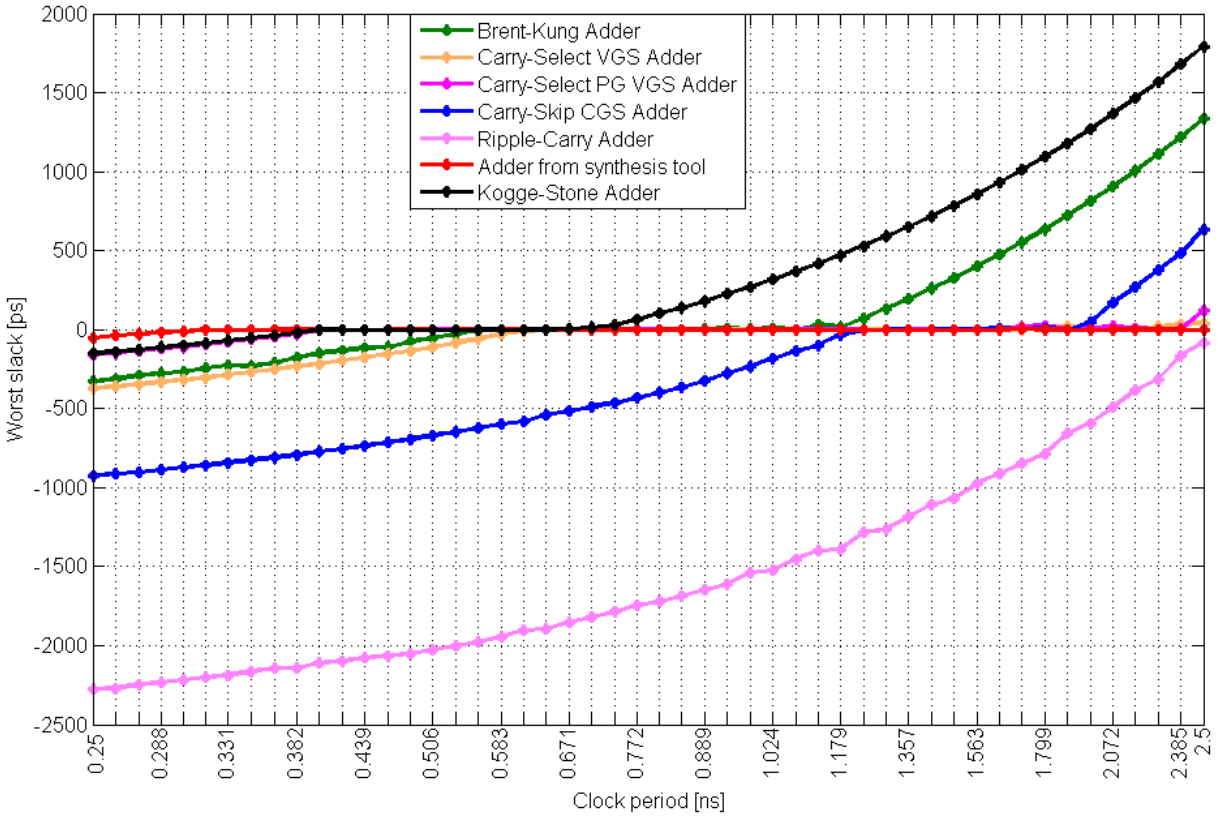


Figure 5.1.1: Timing comparison of 64-bit adders

As expected, the Kogge-Stone adder is faster than almost all architectures due to its massive parallelization to compute each carry. Surprisingly, the same timing was achieved with a carry-select adder embedded with PG functions to accelerate the output selection. We believe the reason behind this fact is the logical functions of the black cells that cannot be optimized to the negative CMOS logic, forcing inverters between logic levels. With the PG functions, the variable group size carry is 34% faster than the version with simple ripple-carry groups which represents a great performance gain. Despite all efforts, no proposed architecture was faster than the automatically generated adder. This fact might come from the technology-optimized adder designs that are already built in the synthesis tool.

Since one of the goals is to compare the area of all architectures, the comparison shown in Figure 5.1.2 refers to the cell area without considering the routing area. The latter is only obtained after the place and route of each architecture and that depends on the wiring congestion on its neighborhood. Firstly, the ripple-carry area does not

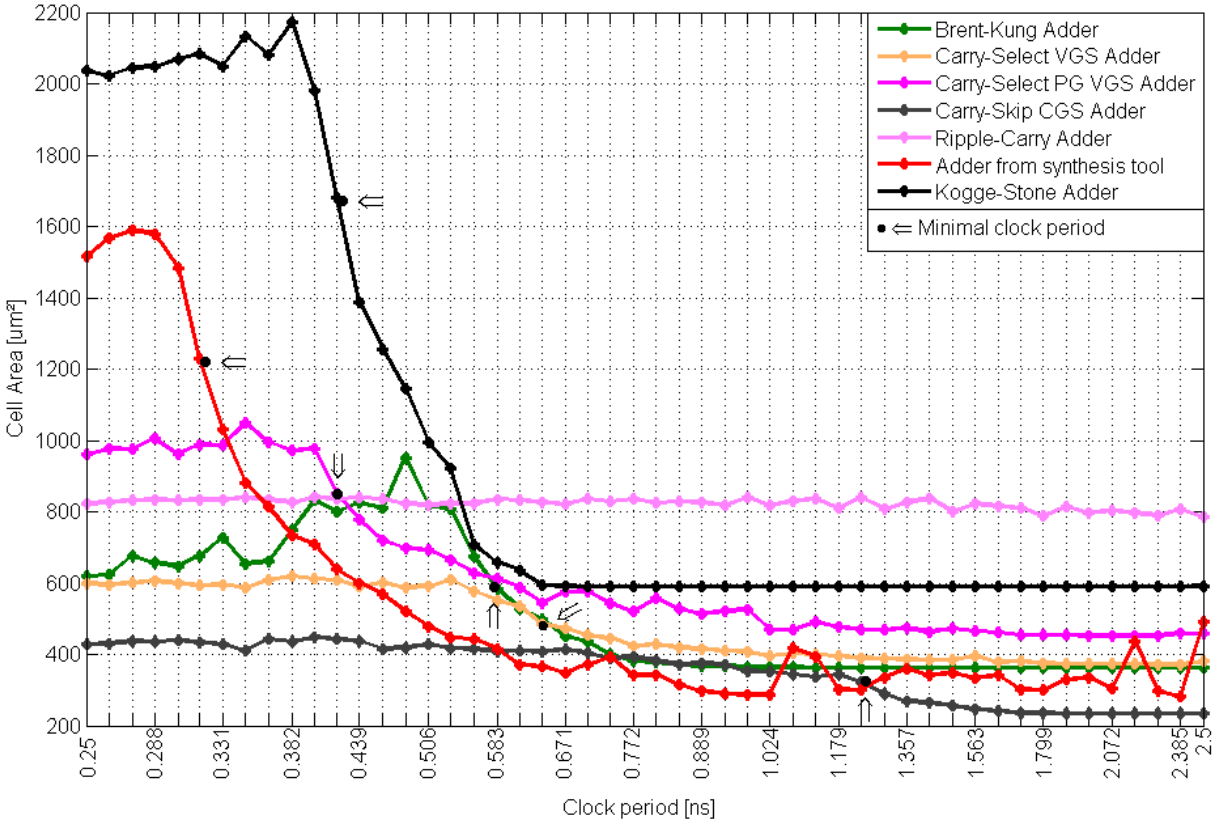


Figure 5.1.2: Cell area comparison of 64-bit adders

have any real significance since it corresponds to a design that does not meet the timing requirement. This area growth is justified as the synthesis tool tries to increase the output drive of each logic port trying to achieve the requirements, leading to bigger logic components. This fact can be observed in all architectures, especially when the latency constraints exceed the design capabilities (point marked with an arrow in the graph).

Clearly, the carry-skip adder presents a huge performance gain if compared to the ripple-carry design. At its minimum clock period, the carry-skip adder has a cell area that is almost 62% smaller and it represents the best design for low frequencies, even better than the one automatically generated from the synthesis tool. To illustrate how speed normally represents a high area cost, let's analyze the carry-select adders. The timing improvement of the design with the PG functions comes with an area increase of 75% compared to that used by the carry-select adder with ripple-carry groups at its minimal clock period.

Observing the parallel-prefix tree adder class, a clear difference appears between the design conceived for the smallest area and deepest logic tree (Brent-Kung) and the one conceived for maximum speed regardless the surface (Kogge-Stone). By reducing the critical path time by 28%, the Kogge-Stone adder needs a cell area that is almost three times bigger than the area occupied by the other prefix tree adder.

For power comparison, only leakage power will be considered because it is an

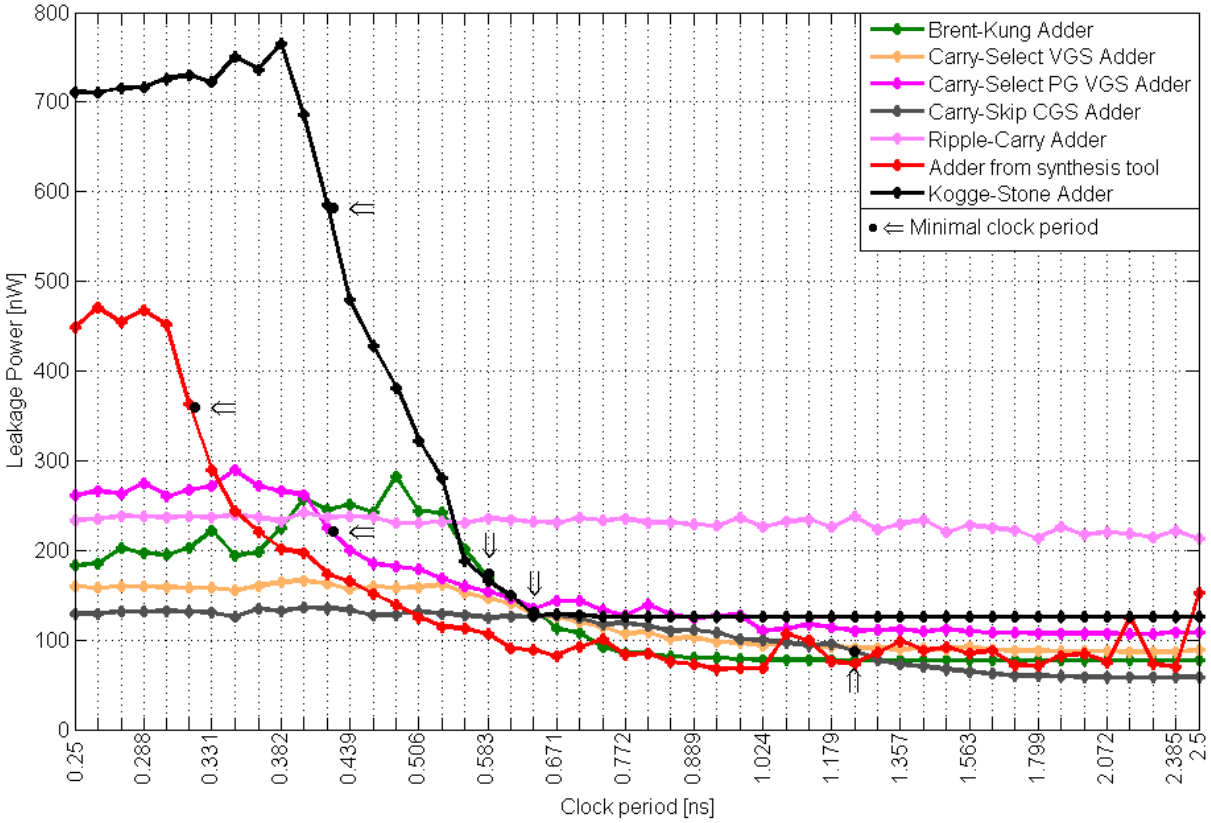


Figure 5.1.3: Leakage power comparison of 64-bit adders

intrinsic characteristic to each logic gate. Since the dynamic consumption is, by default, based on statistical approach of the input signals commuting, it is not addressed by this work. The synthesis tool considers the static power of each logic gate as well as a simple wire model to create an estimation of the interconnection power loss.

Generally, bigger designs have bigger leakage power because the latter is directly linked to the gate count and size, as can be seen from Figure 5.1.3. Thus, comparing both carry-select adders at their minimum clock periods, we observe that the one with PG functions has power consumption 77% higher than the traditional approach with ripple-carry groups. Also, it is interesting to note that the gain of 164 ps that the Kogge-Stone adder has over the Brent-Kung adder increases the static power by a factor of almost 3.5.

## 5.2 Analysis and comparison of multipliers architectures

The multipliers were divided into two categories since signed and unsigned designs differ due to their refined logic to manage the number signs. Since all multipliers need a final adder with carry propagation to compute the final result, this adder is automatically designed by the synthesis tool because the results shown in Section 5.1 show that this adder outperforms all designs as the timing constraints become harder to

meet. Following the same worst-case approach as for the adders, the input operands have 54 bits since they represent the mantissas' multiplication in the double precision floating point unit. The legends with the text "4:2 compressor" refer to designs using these compressors as the preferred cells in addition to the half and full adders.

### 5.2.1 Unsigned multipliers

Using the same previous approach with the automatically generated design as the reference model, Figure 5.2.1 shows the timing evolution of the studied architectures. The first fact to note is that multipliers are much slower than simple adders due to the amount of partial products to sum.

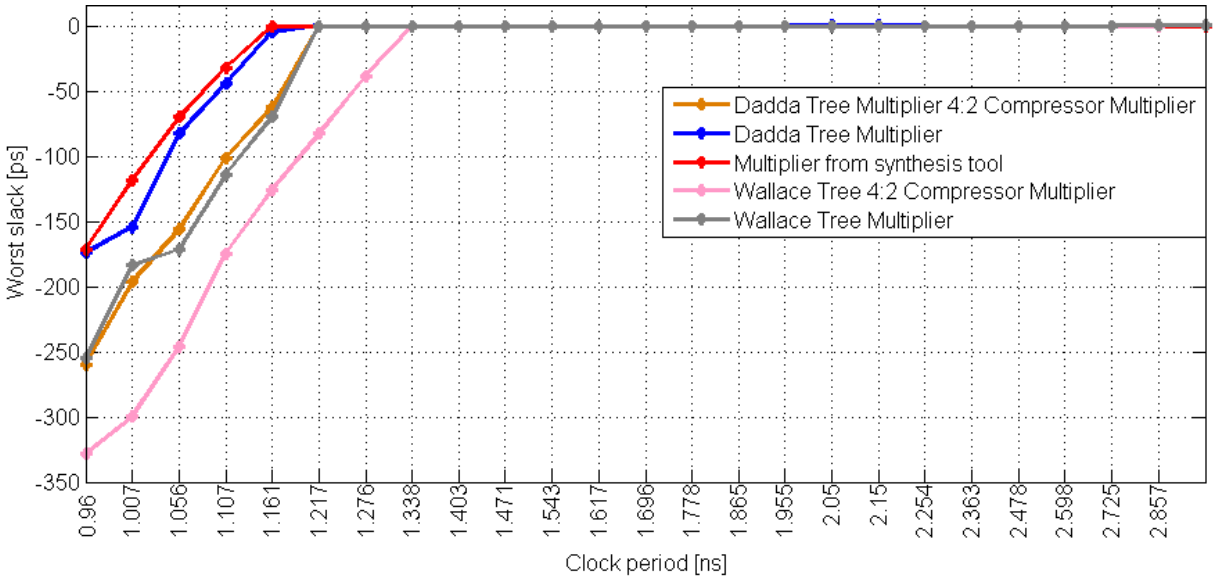


Figure 5.2.1: Timing comparison of $54 \times 54$ unsigned multipliers

As expected, the multipliers based on the Dadda tree are faster than those based on the Wallace tree because the reduced number of adder cells decreases considerably the critical path. Differently from the adders, the reference model is only 4% faster than the original Dadda multiplier without the 4:2 compressors.

The speed gain of the reference model reflects on the design cell area. Being 4% faster leads to a design almost 12% larger than our best multiplier, proved by Figure 5.2.2. Note that both Dadda and Wallace multipliers have the same minimal clock period, although the latter occupies an area 55% larger than the area of the former. Clearly, the use of 4:2 compressors degrades both timing and area, contrary to what was expected. We believe that this fact may be linked to the absence of this cell type in the technology standard cells, thus the need to group other logic cells to create such function without the proper timing match.

The artifacts found in the middle of the graph refer to a change in the synthesis

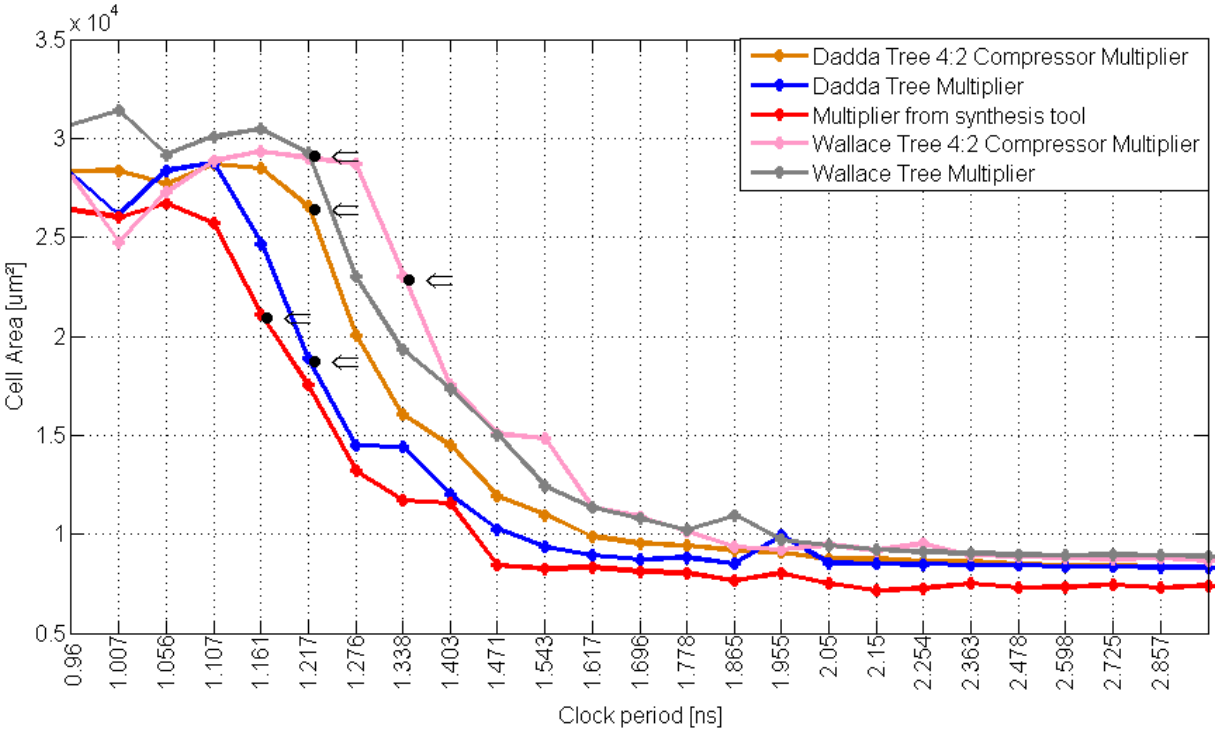


Figure 5.2.2: Cell area comparison of $54 \times 54$ unsigned multipliers

tool mapping strategy, restructuring the logic functions to use other logic gates as well as increasing or decreasing the cell output drive. The thresholds of such changes are inherent to each tool-technology pair and their analysis is out-of-scope of this work.

Regarding the leakage power of such multipliers, the same behavior is found (see Figure 5.2.3). The multiplier generated by the synthesis tool is 56 ps faster at the expense of increasing the static power by 11% compared to the classic Dadda multiplier. Although, comparing those two multipliers at the minimal clock period where both architectures have a non-negative timing slack, the Dadda multiplier has a leakage around 8% higher. Comparing both versions of the Dadda multipliers, the one with 4:2 compressors has a leakage power around 44% higher than its counterpart.

### 5.2.2 Signed multipliers

Since both Baugh-Wooley and Booth techniques are methods to generate partial products for the signed multiplications, we used them with all the adder trees used in the unsigned multipliers in order to analyze which tree is best adapted for which architecture. Thus, Figure 5.2.4 shows the timing evolution of all the possible designs combinations for this multiplier type. For the first time, one of the proposed designs – Dadda multiplier with Booth algorithm – achieves the same speed as the synthesis tool's multiplier. One first interesting conclusion from this graph is that the Booth algorithm represents the best scheme for this type of multiplier, either with the Dadda or the Wallace tree.

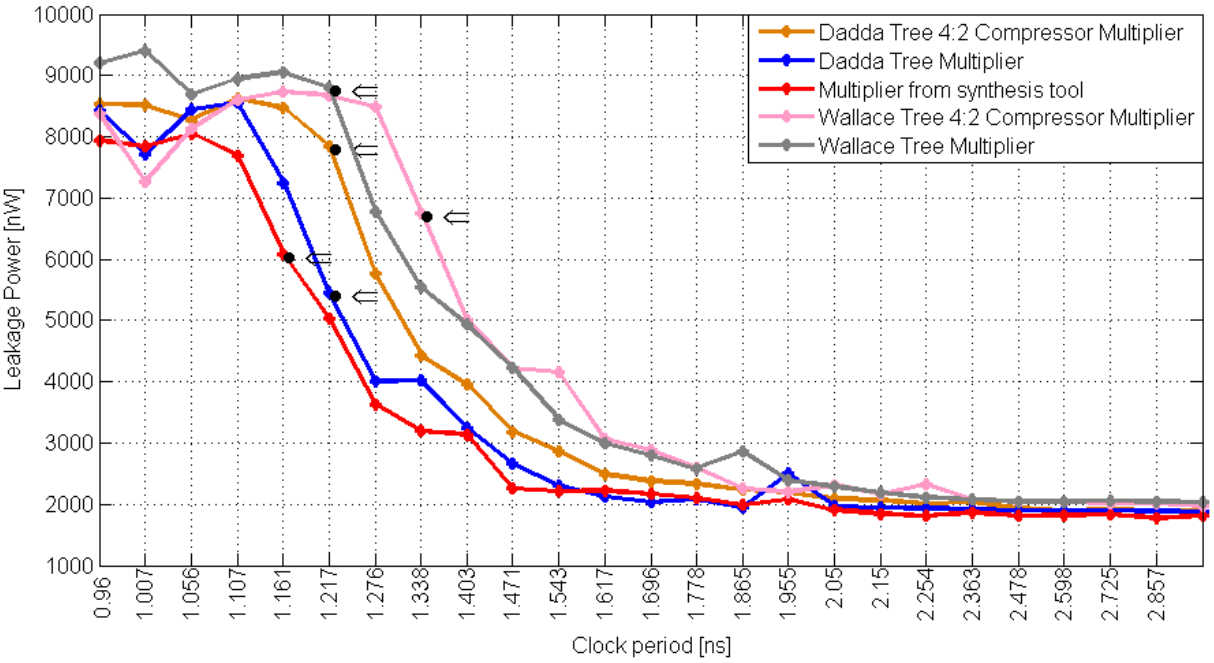


Figure 5.2.3: Leakage power comparison of $54 \times 54$ unsigned multipliers

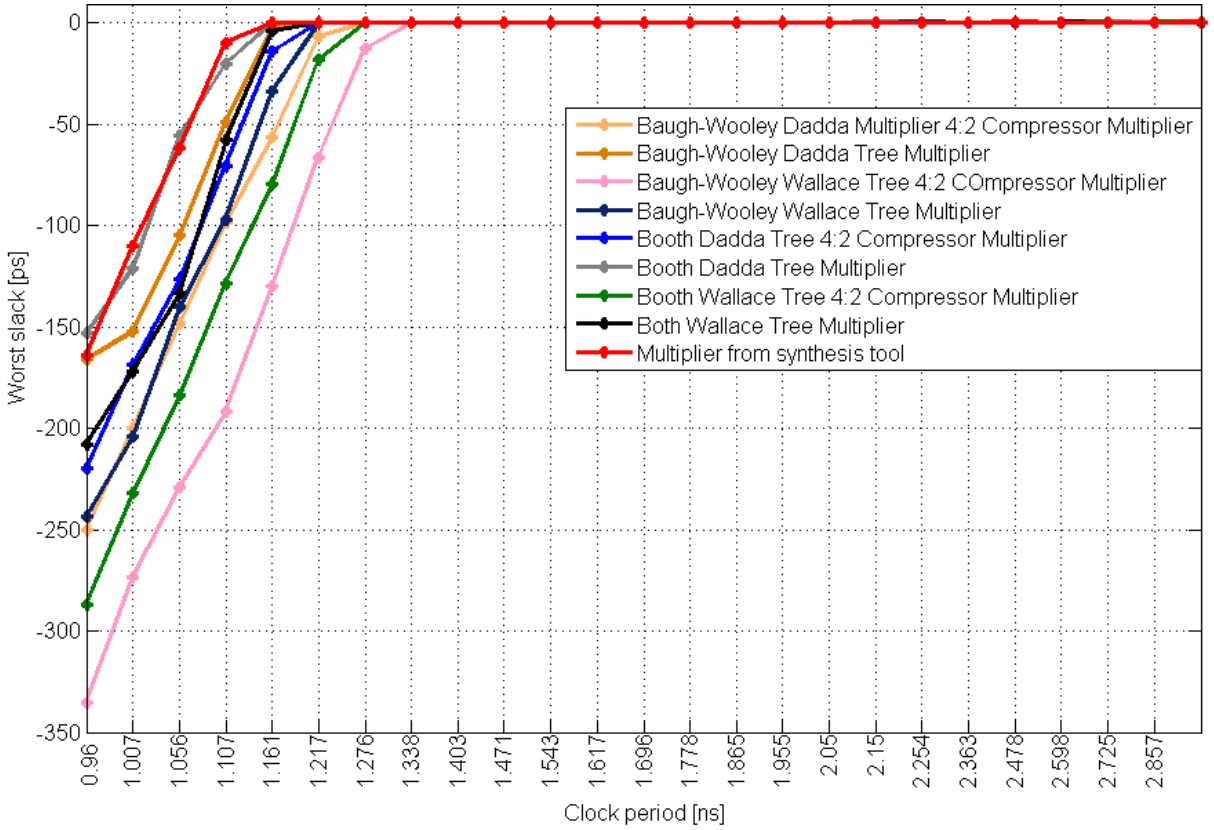


Figure 5.2.4: Timing comparison of $54 \times 54$ signed multipliers

Looking at the maximum speed data points of all architectures in Figure 5.2.5, the cell area of the automatically generated design is around 3% larger than our best design. Also, it is interesting to observe all the other designs have a larger area than the fastest

ones. For instance, the Dadda multiplier implementing the Baugh-Wooley algorithm at its maximum speed (1.217 ns) is almost 30% larger than the one implementing the Booth algorithm at the same clock period, despite the additional hardware to analyze and encode the inputs.

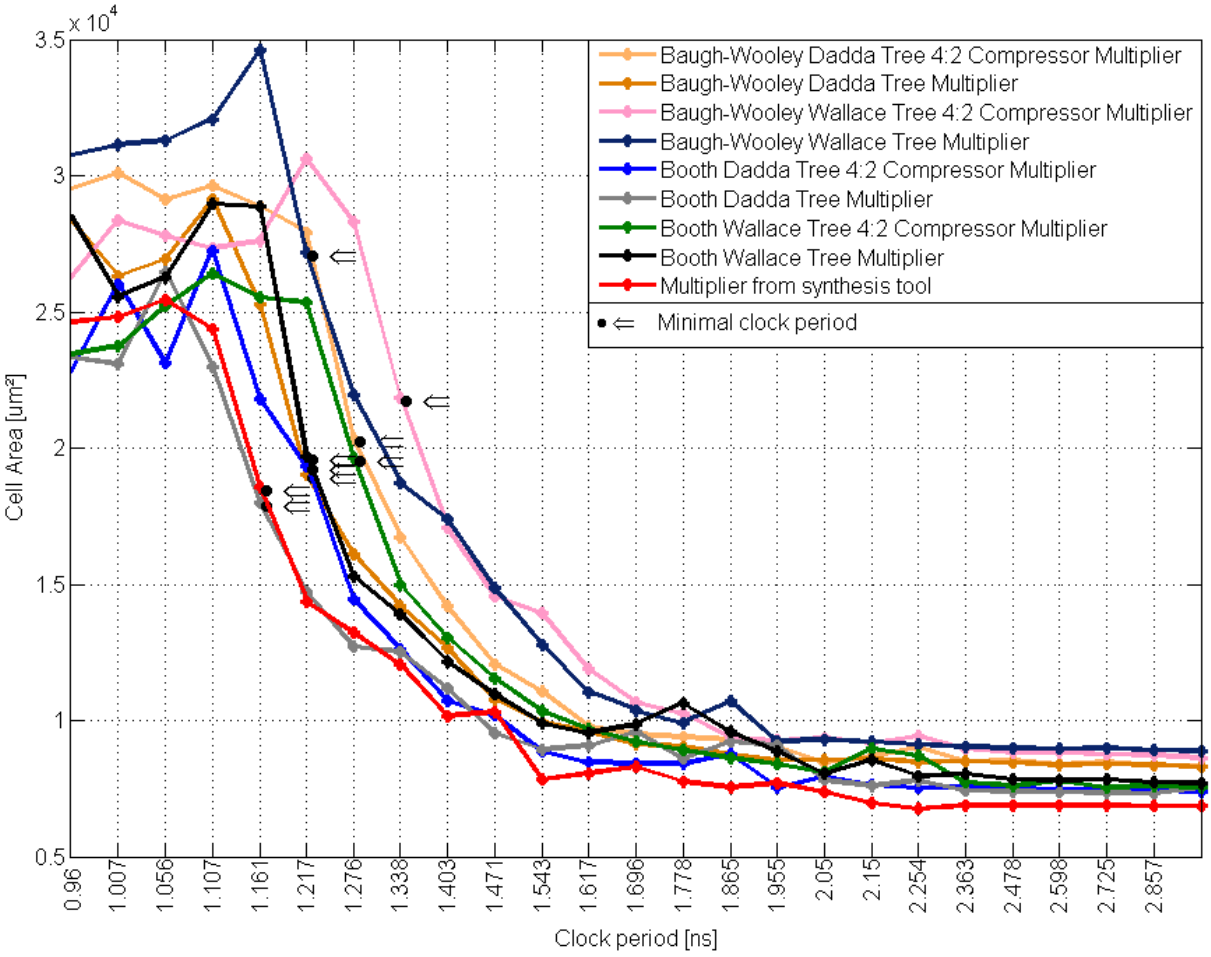


Figure 5.2.5: Cell area comparison of $54 \times 54$ signed multipliers

As a result of a smaller design, the Dadda-Booth multiplier shows a reduction of 7% for the static power comparing to the reference model (see Figure 5.2.6). If the clock period is increased by only 4% (to 1.217 ns), the leakage power of the Dadda-Booth multiplier is reduced by 21% which is an interesting trade-off for the sake of an energy-aware design. Again, all designs using 4:2 compressors proved inefficient in relation to those using only full and half adders.

An intriguing conclusion is that the fastest unsigned multipliers have higher leakage power and cell area in comparison to the signed multipliers. One possible reason for this fact is that the reference model uses a Booth multiplier with additional logic to process the unsigned numbers since its characteristics are similar to those of the best signed multiplier.

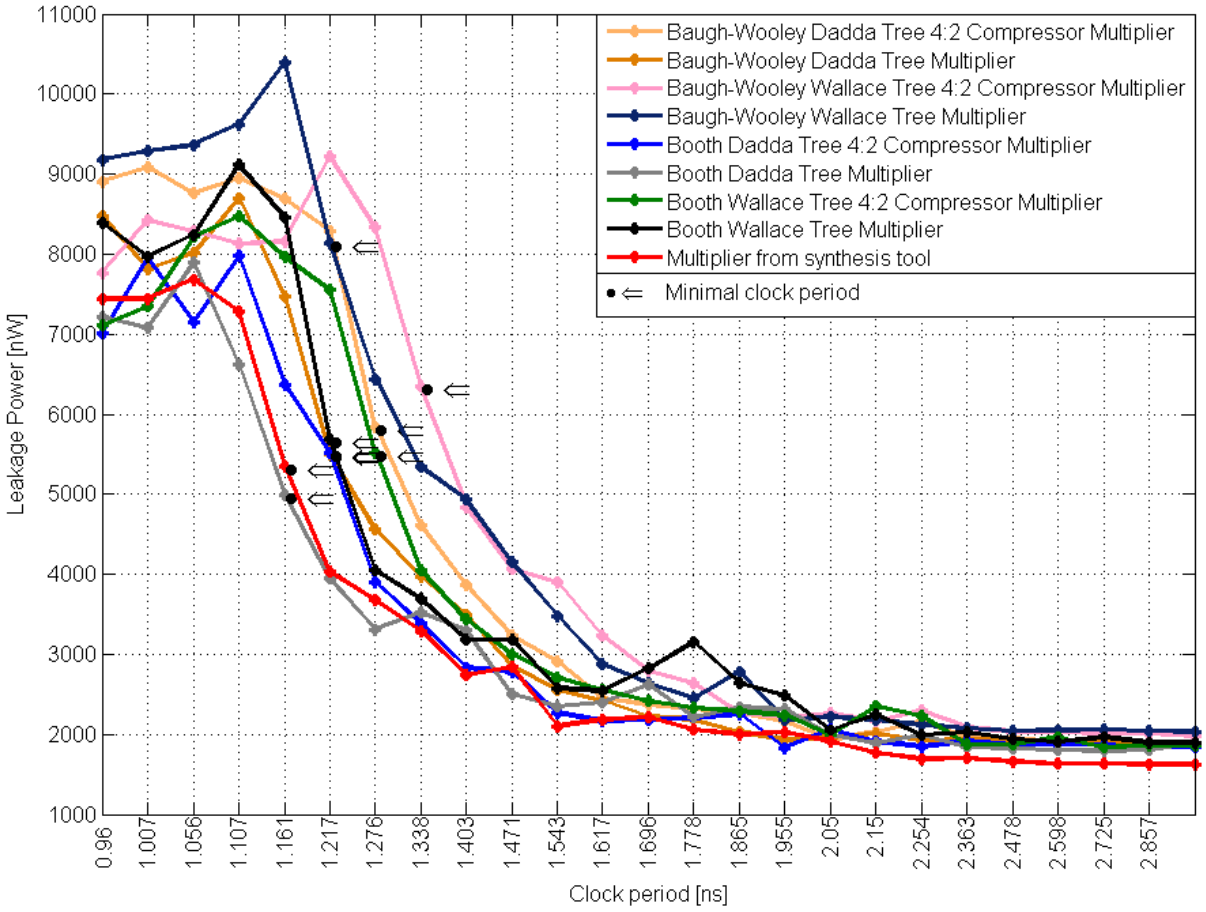


Figure 5.2.6: Leakage power comparison of $54 \times 54$ signed multipliers

### 5.2.3 Further comparison between signed and unsigned multipliers

In the previous sections, the widest input data size considered was 54 bits (the mantissa length in double precision floating point representation). However, to have both signed and unsigned multipliers in the same design which are capable of performing operations in such size, it is necessary to have multipliers with inputs 55-bit wide as a result of the zero needed in the most significant bit to have a unsigned multiplication.

So, the reference model and the Booth-Dadda multiplier (the best architecture so far) were synthetized for inputs 54-bit and 55-bit wide in order to extract and compare the QoR of both architectures supporting signed and unsigned operations. Looking at the cell area comparison of these designs in Figure 5.2.7, it is clear that the Booth-Dadda has a considerable area reduction if compared to the reference model when the widest inputs are considered.

The Booth-Dadda multiplier has roughly the same area for both input sizes as the introduction of the new partial product for the 55-bit inputs – totalling 28 PPs – does not have a significant impact on the compression tree. Extending the analysis, Figure 5.2.8 shows the comparison of leakage power for both architectures. Again,

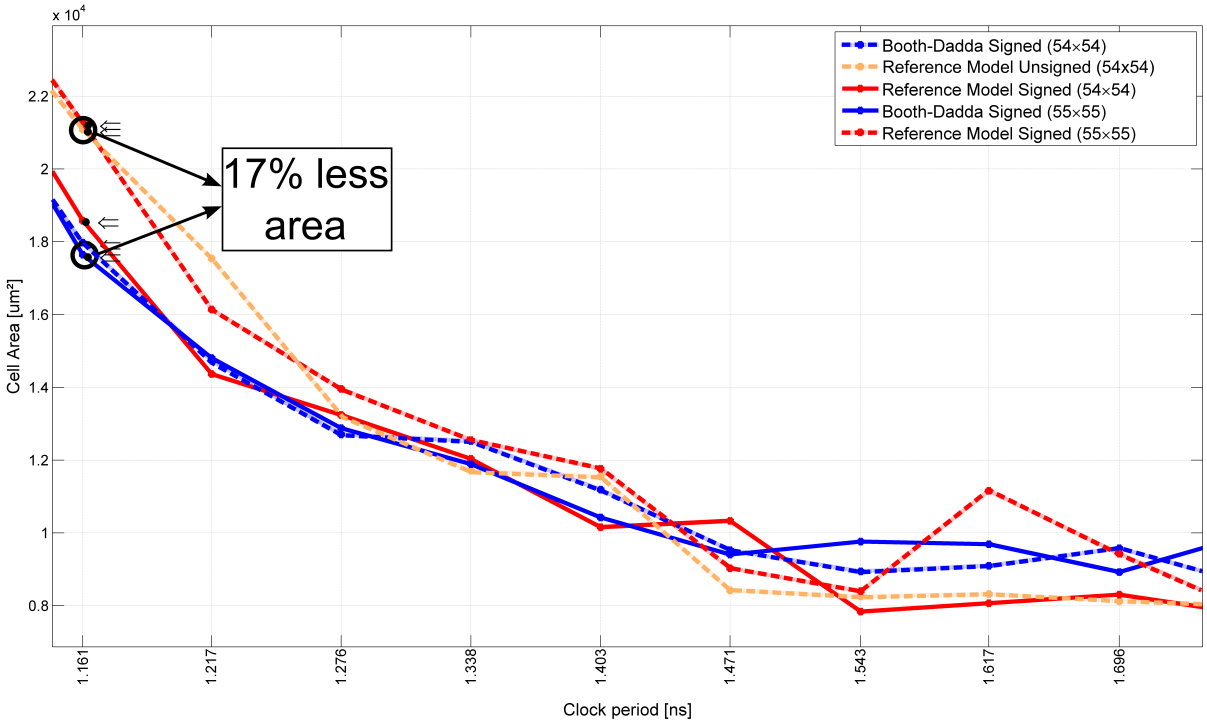


Figure 5.2.7: Cell area comparison for multiple sizes of signed and unsigned multipliers

the Booth-Dadda design outperforms the reference model by a significative amount, justifying the adoption of the design with the widest inputs considering the obtained benefits regarding all available architectures.

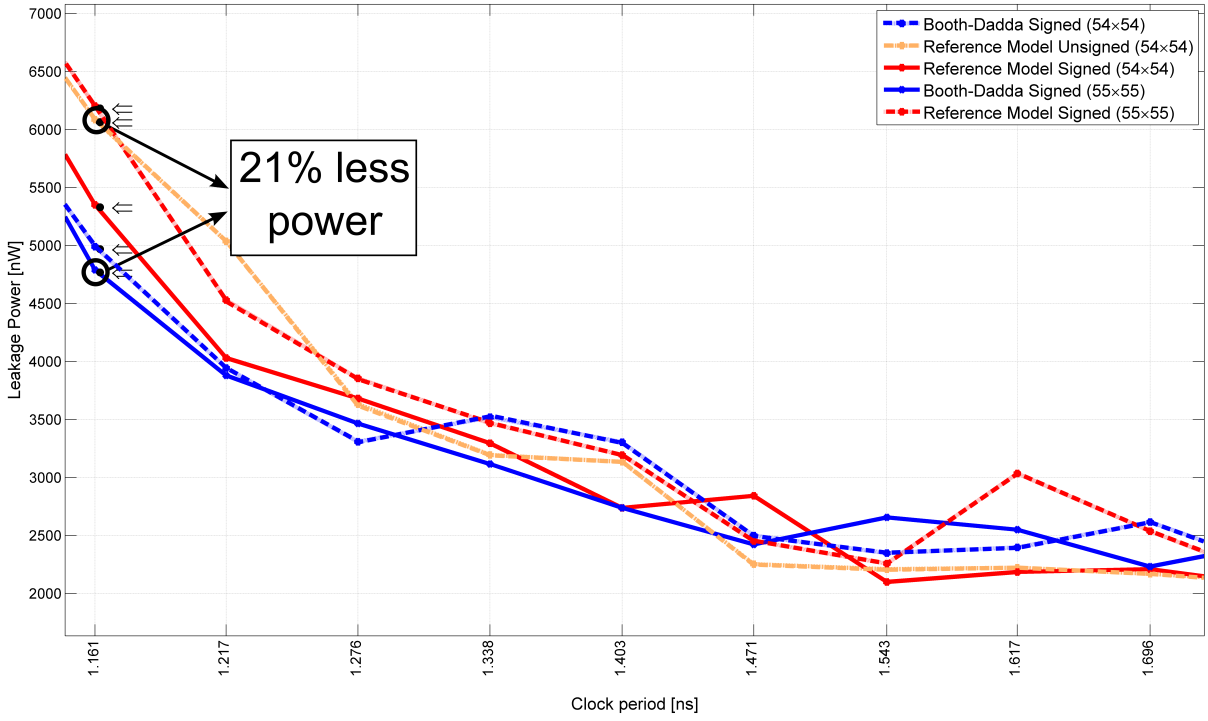


Figure 5.2.8: Leakage power comparison for multiple sizes of signed and unsigned multipliers

# 6 Task Scheduling

At the beginning of the internship, there was a very important period where I learned about the development environment that I would work with. During this period, I have learned how to use a distributed revision control system (Git) to keep track of the activities I was doing. Besides, I became aware of the synthesis and verification flows already existent with their respective directory structures. Soon after this familiarization, I started to research about binary adders and the most common architectures that can be implemented using standard cells for a given technology.

Initially, it was easy to manually describe the first adders because their definition was straightforward to implement. However, the more I learned about adders, the more difficult it was to implement them. Consequently, we had the idea of developing a code generator that could handle these difficulties in a higher level of abstraction. Since I had no idea how to build such generator, it took a long time to do so because I had to define the system requirements before implementing it in a language I had no prior knowledge, though it has a short learning curve. Python was the chosen programming language mainly because it is interpreter-based, which facilitates debugging, and due to embedded support for data structure such as lists and dictionaries.

As the work progressed, the scripts for synthesis, verification and data extraction were updated as needed in order to automatize the flow. With the gathered data, it was necessary to organize them in a more comprehensible way, thus the development of the MATLAB script to generate graphs that facilitate the comparison of the implemented architectures. This script allowed me to further investigate optimizations in those circuits. However, we decided to create a database to store such results and develop an interpreter for it so it would be easier to choose which architecture was better suited for a specific constraint set, behaving like a component library with an easy access.

When my grasp of the adders was advanced enough, I started reading about multiplier architectures. Since their structures are more complicated than those of the adders, I took a longer time to start implementing them. As I was learning about new architectures, I implemented those that seemed prosperous for our purpose. All the while, I started a draft of my report in order not to forget the specific details of each operator.

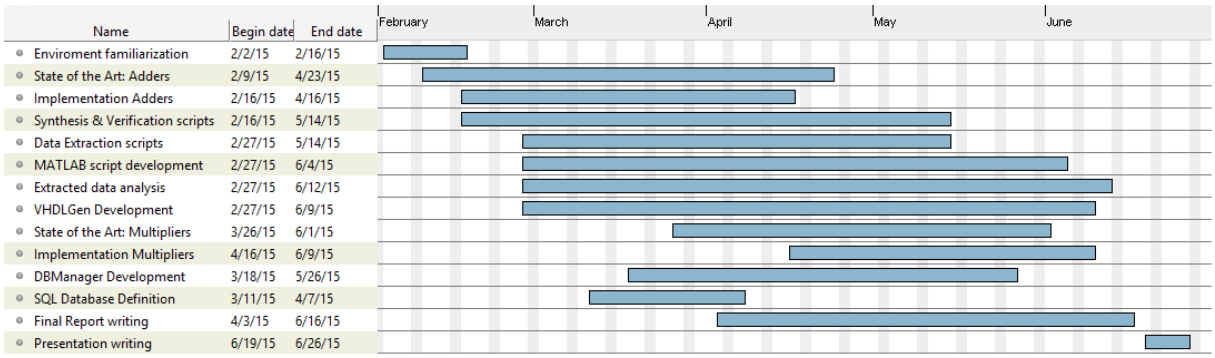


Figure 6.1: Gantt Diagram of executed tasks

# 7 Conclusion

Many-core processors represent a consolidated choice to mitigate the physical constraints imposed by the manufacturing process of integrated circuits and the power consumption in order to achieve a higher computing capacity. Thus, it is important to optimize the arithmetic operators inside these processors to achieve better performance. The study presented by this work shows how different architectures can have a significant impact on the chip and, consequently, how to better adapt the requirements based on each design characteristic to achieve the best performance.

From the presented results, we saw that the current hardware selection – automatically generated by the synthesis tool – to perform addition is better in almost all test frequencies. However, the same is not true for the multipliers since the current synthesis tool designware can be outperformed by custom architectures that can be designed from standard cells. From the results shown, one of the proposed multiplier architectures can save up to 17% of cell area in comparison to a synthesis tool-based architecture. This fact illustrates the usefulness of a database with such information to help deciding which design should be used upon the imposed constraints. Furthermore, the proposed database system proved itself very useful to gather together the information and it could be easily extended to support any kind of circuit.

This study was done during an intership at Kalray. The designs considered in this work have presented great challenges. This work demanded a great amount of skills, either in arithmetic circuits as well as in programming skills. Besides, it constantly required a keen mind to perform deeper theoretical and empirical analysis to process the data gathered through the experiments.

# Bibliography


[Bew94] G. W. Bewick. “Fast Multiplication : Algorithms and Implementation”. PhD thesis. Stanford University, 1994, p. 170.

[BK82] R. P. B. Brent and H. T. Kung. “A Regular Layout for Parallel Adders”. In: *IEEE Transactions on Computers* C-31.3 (1982), pp. 260–264. ISSN: 0018-9340.

[Boo51] A. D. Booth. “A signed binary multiplication technique”. In: *Quarterly Journal of Mechanics and Applied Mathematics* 4.2 (1951), pp. 236–240.

[BW73] C. Baugh and B. Wooley. “A Two’s Complement Parallel Array Multiplication Algorithm”. In: *IEEE Transactions on Computers* C-22.12 (1973), pp. 1045–1047. ISSN: 0018-9340.

[Dad65] L. Dadda. “Some Schemes for Parallel Multipliers”. In: *Colloque sur l’Algèbre de Boole* (1965).

[DBS06] J.-P. Deschamps, G. J. A. Bioul, and G. D. Sutter. *Synthesis of Arithmetic Circuits (FPGA, ASIC and Embedded Systems)*. John Wiley & Sons, Inc, 2006, p. 556.

[De +13] B. D. De Dinechin et al. “A clustered manycore processor architecture for embedded and accelerated applications”. In: *2013 IEEE High Performance Extreme Computing Conference, HPEC 2013* (2013).

[FO98] A. A. Farooqui and V. G. Oklobdzija. “General data-path organization of a MAC unit for VLSI implementation of DSP processors”. In: *ISCAS ’98. Proceedings of the 1998 IEEE International Symposium on Circuits and Systems (Cat. No.98CH36187)* 2 (1998), pp. 260–263.

[HC86] M. Hatamian and G. Cash. “A 70-MHz 8-bit x 8-bit parallel pipelined multiplier in 2.5-$\mu$m CMOS”. In: *IEEE Journal of Solid-State Circuits* 21.4 (1986), pp. 505–513. ISSN: 0018-9200.

[Kat94] R. H. Katz. *Contemporary logic design*. 1st. Redwood City, CA, USA: Benjamin-Cummings Publishing Co., 1994. ISBN: 0805327134.

[Ken83] W. Kent. “A simple guide to five normal forms in relational database theory”. In: *Communications of the ACM* 26.2 (1983), pp. 120–125. ISSN: 00010782.

[Kno01] S. Knowles. “A family of adders”. In: *Proceedings 15th IEEE Symposium on Computer Arithmetic. ARITH-15 2001* (2001), pp. 277–284.

[Kog+08] P. Kogge et al. “ExaScale Computing Study : Technology Challenges in Achieving Exascale Systems”. In: *Government PROcurement* TR-2008-13 (2008), p. 278.

[KS73] P. M. Kogge and H. S. Stone. “A Parallel Algorithm for the Efficient Solution of a General Class of Recurrence Equations”. In: *IEEE Transactions on Computers* C-22.8 (1973), pp. 786–793. ISSN: 0018-9340.

[Mac61] O. Macsorley. “High-Speed Arithmetic in Binary Computers”. In: *Proceedings of the IRE* 49.1 (1961).

[Par00] B. Parhami. *Computer Arithmetic: Algorithms and Hardware Designs*. 1st ed. New York, New York, USA: Oxford University Press, 2000. ISBN: 0-19-512583-5.

[Sin+13] A. K. Singh et al. “Mapping on multi many core systems: Survey of current and emerging trends”. In: *Design Automation Conference (DAC), 2013 50th ACM / EDAC / IEEE* (2013), pp. 1–10.

[SL08] M. Själander and P. Larsson-Edefors. *The Case for HPM-Based Baugh-Wooley Multipliers*. Tech. rep. 08. Göteborg, Sweden: Chalmers University of Technology, 2008, p. 16.

[Tya93] A. Tyagi. “A reduced-area scheme for carry-select adders”. In: *IEEE Transactions on Computers* 42.10 (1993), pp. 1163–1170. ISSN: 0018-9340.

[Wal64] C. S. Wallace. “A Suggestion for a Fast Multiplier”. In: *IEEE Transactions on Electronic Computers* EC-13.1 (1964), pp. 14–17.

# A Detailed analysis of adder architectures

In this appendix, some adder architectures will be further analyzed from the gathered synthesis data. This analysis is more fine-grained since in considers the particularities of each design. For all designs, the chosen data size will be equal to the same selected before, thus only 64-bit adders will be considered.

## A.1 Sparsity impact in Kogge-Stone adders

As studied in Section 3.1.2.2, some modifications can be made to the Kogge-Stone prefix tree structure in order to reduce the place and routing challenges as well as the occupied area. These alterations refer to the tree sparsity, that is, how many carries will be calculated from the tree. A sparsity-2 tree, for instance, will compute one carry out of two, leading to an adder group of two adders at the tree end. Thus, Figure A.1.1 shows the timing of three different trees. The sparsity-1 tree refers to the complete Kogge-Stone prefix tree. As expected, there is a time penalty associated to the sparsity since it uses a group of ripple-carry adders at the tree end to compute the carries which were not calculated previously.

From Figure A.1.2 it is possible to see some advantages of using sparse trees. If the penalty of 4% in timing (19 ps) is tolerable, the sparsity-2 tree shows a good balance between speed and area since it is 12% smaller than the sparsity-1 tree at the same speed. Trees with higher sparsity values does not seem to be a good choice since the adder chain at the end becomes large enough to have a huge penalty in timing. Consequently, it forces the synthesis tool to increase the cells' drives trying to meet the clock constraint which results in a much larger design. This fact is illustrated by the sparsity-4 tree.

Extending this analysis to power leakage of these designs, the same behavior is found, as can be seen in Figure A.1.3. In this case, the sparsity-2 tree at its minimal clock period consumes almost 20% less power than the sparsity-1 tree at the same speed, which is an interesting fact when power is a major concern. If the clock constraint is loosened this difference is more pronounced. When all three curves are flat, the sparsity-2 tree consumes around 40% less than the fastest tree and 16% more than the slowest tree.

Thus, if speed is the principal factor in the designs' QoR, the sparsity-1 tree is clearly the best choice as it is the fastest design obtained. However, if we are willing to

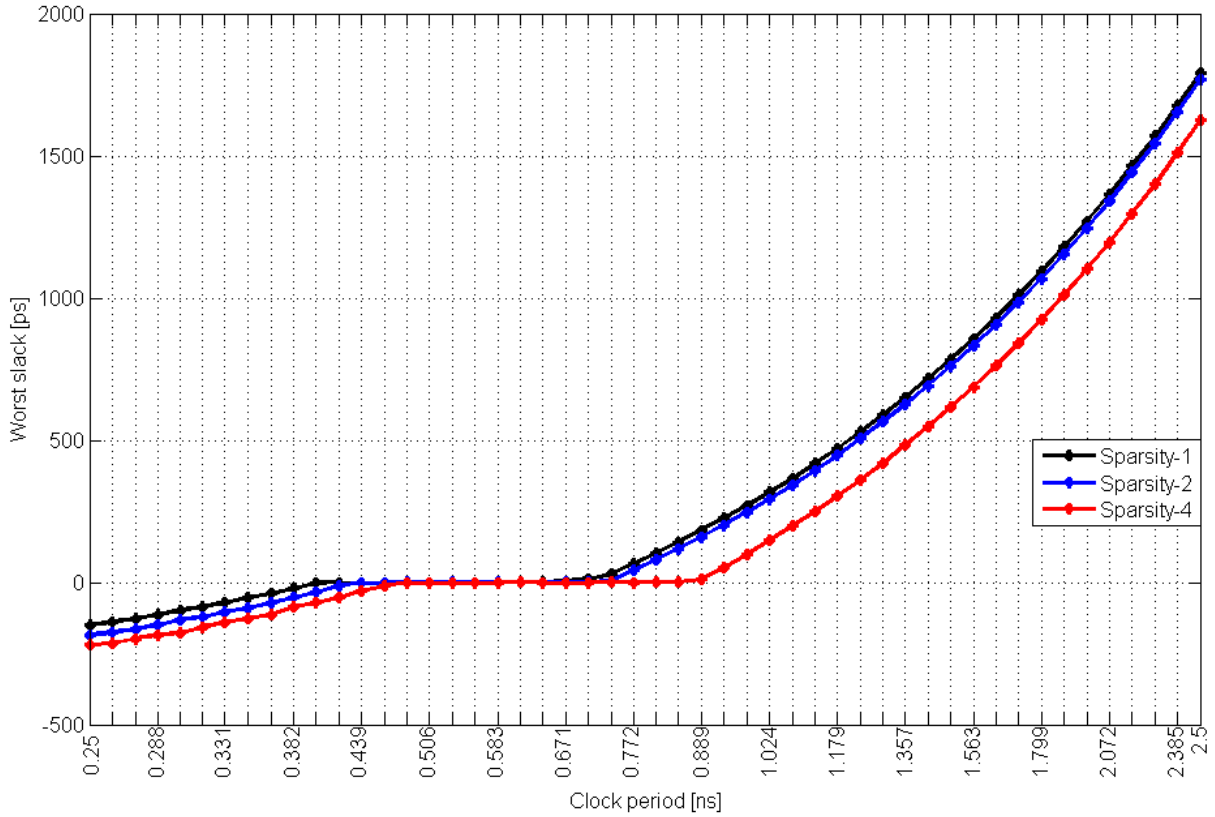


Figure A.1.1: Timing comparison of 64-bit Kogge-Stone adders for different sparsity values

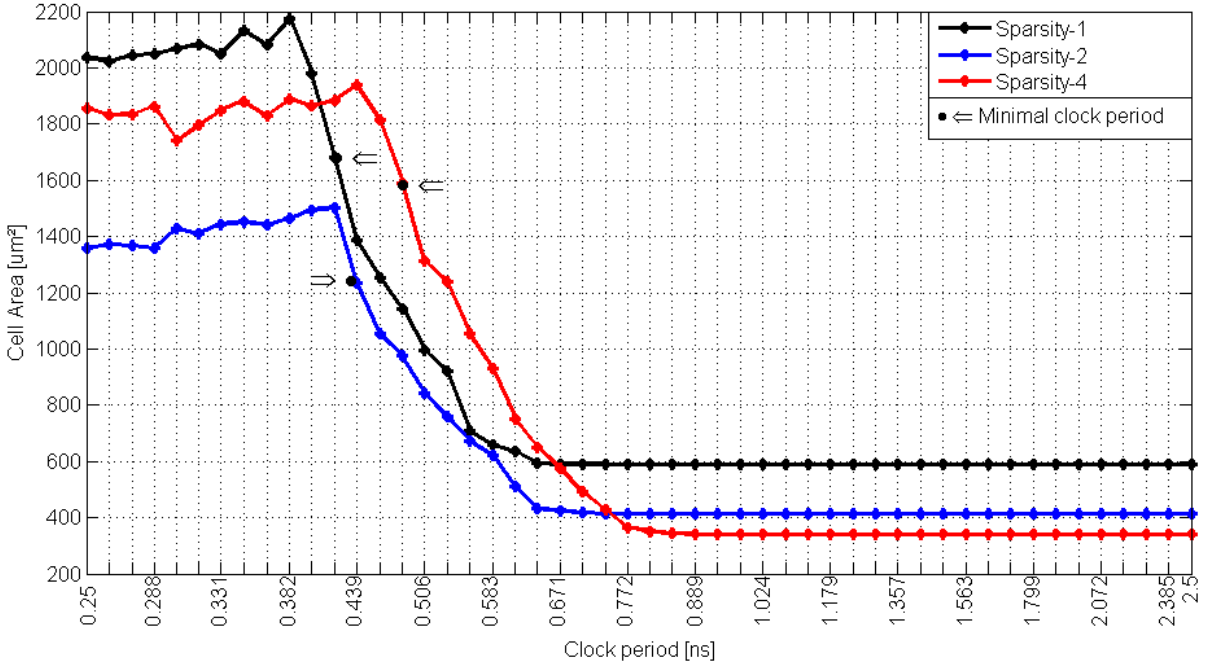


Figure A.1.2: Cell area comparison of 64-bit Kogge-Stone adders for different sparsity values

make a small speed sacrifice, the sparsity-2 tree is, by far, the best choice since it has considerable gains in area and power leakage both in higher and lower frequencies.

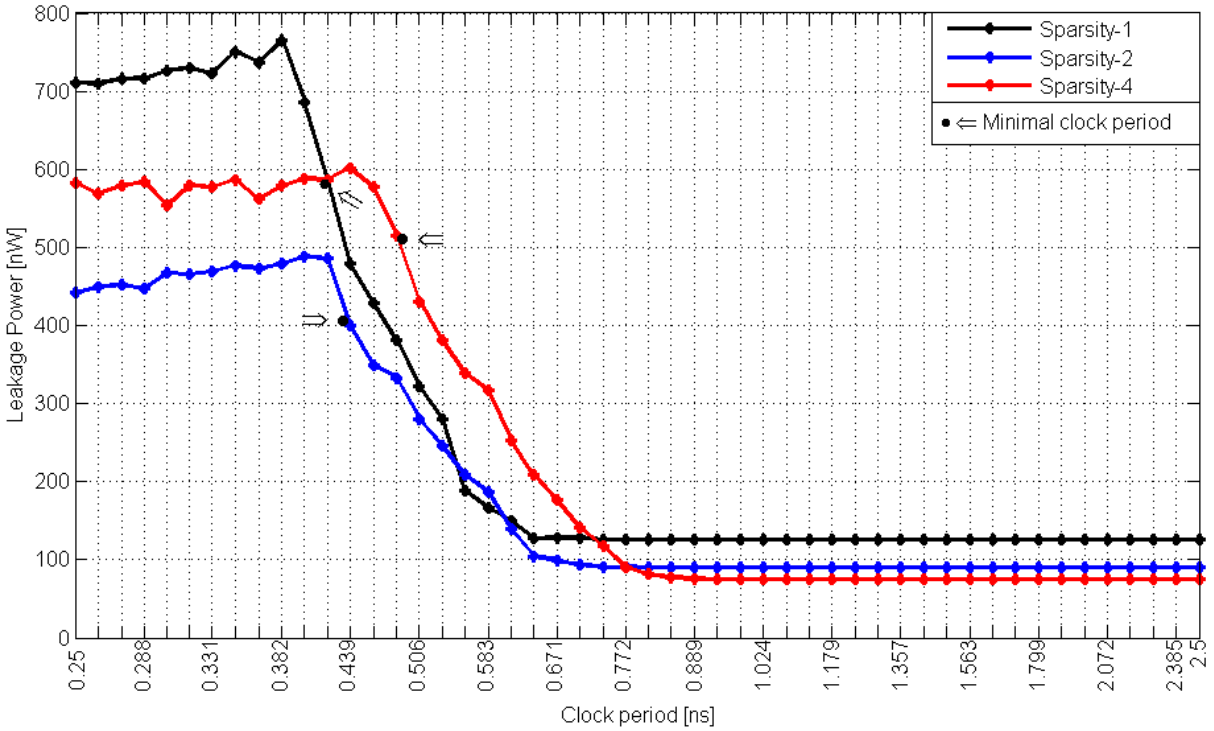


Figure A.1.3: Leakage power comparison of 64-bit Kogge-Stone adders for different sparsity values

## A.2 Group size impact in carry-skip adders

From the experimental data shown in Section 5.1, we observed that the carry-skip adders have shown a great speed increase with little logic overhead. Consequently, it motivates the study of the group size impact in the architecture QoR. For this analysis, only constant group sizes are considered due to the difficulty of implementing an organ-pipe structure for the variable block sizes. The theory about variable-sized groups will not be addressed by this work. For the constant-sized groups, let us use Figure A.2.1 as a base to develop the theory.

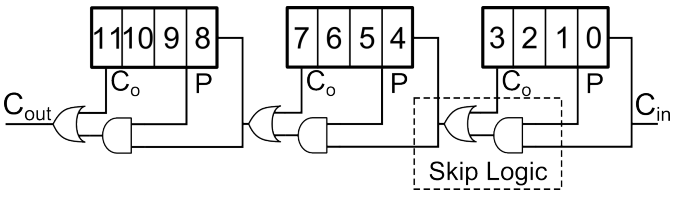


Figure A.2.1: Structure of a carry-skip adder

Consider a $n$-bit carry-skip adder composed of $k$ bits groups. From the adder structure, we observe that the first and last groups will propagate the carry from the first to the last adder of their respective groups. For the intermediary groups, the carry will be calculated by the skip logic. Thus, if we take $T_{FA}$ as the full adder delay (equal to

2D) and $T_{SL}$ as the time to traverse the skip logic (equal to 2D), we have:

$$T_{Total} = \overbrace{(k-1)T_{FA}}^{\text{First group}} + D + (n/k - 2)T_{SL} + \overbrace{(k-1)T_{FA}}^{\text{Last group}} \tag{A.2.1}$$

where $D$ corresponds to the delay of the OR gate to which the first group is connected. Taking the derivative of this function and finding its maximum and minimum, the optimal group size for a $n$-bit adder is given by the expression:

$$k_{opt} = \sqrt{\frac{n}{2}} \tag{A.2.2}$$

For a 64-bit adder, this result is approximately 5,6. Thus, several group sizes were tested to observe the QoR variations to verify this theoretical analysis. Figure A.2.2 shows the timing evolution for each group size. The best architecture is the 4-bit group size which is 9% faster than the 6-bit group size which is the closest group size to the one obtained from the equations above. Interestingly, the 2-bit group shows that there is a lower threshold for the group size from which the logic overhead becomes dominant and, consequently, it impacts negatively on the design timing.

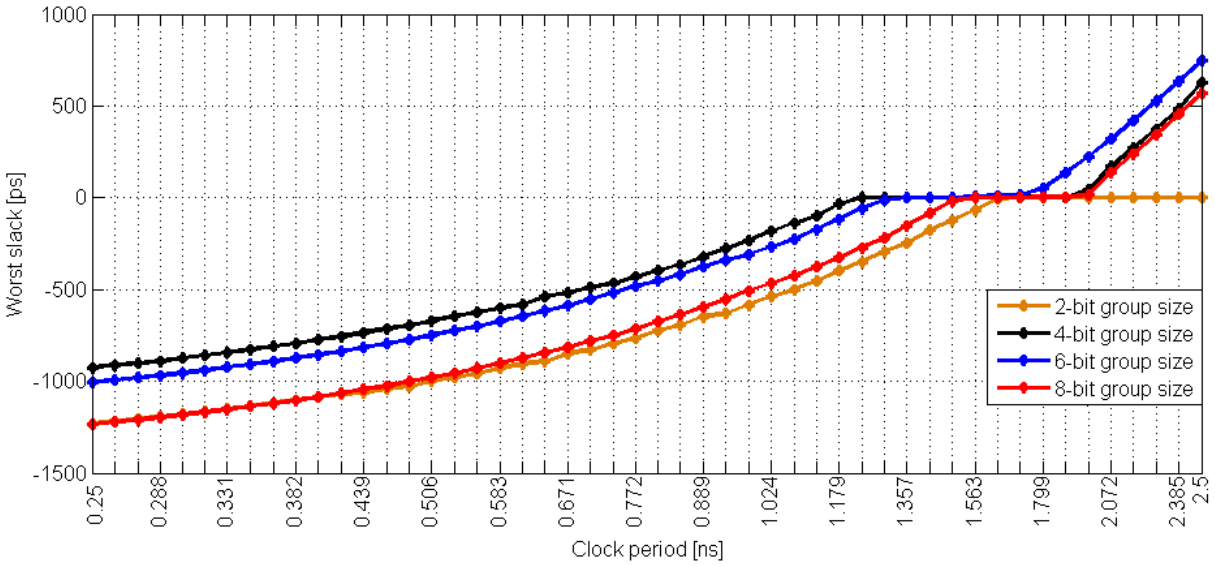


Figure A.2.2: Timing comparison of 64-bit carry-skip adders with constant group sizes

Analyzing the adder area from Figure A.2.3, it is clear to observe that the skip logic represents a great overhead for the 2-bit group scheme, demanding higher cell drives to meet the timing constraints, thus leading to a very large and inefficient design. On the other hand, the 6-bit group proves to be a very area-efficient design, occupying 11% less area than the fastest adder when synthetized at the minimal clock period of the former. Besides, at higher periods, it still outperforms both 2- and 8-bit group schemes in terms of area. However, this design has a small disadvantage if compared

to the 8-bit group adder when the curves become flat.

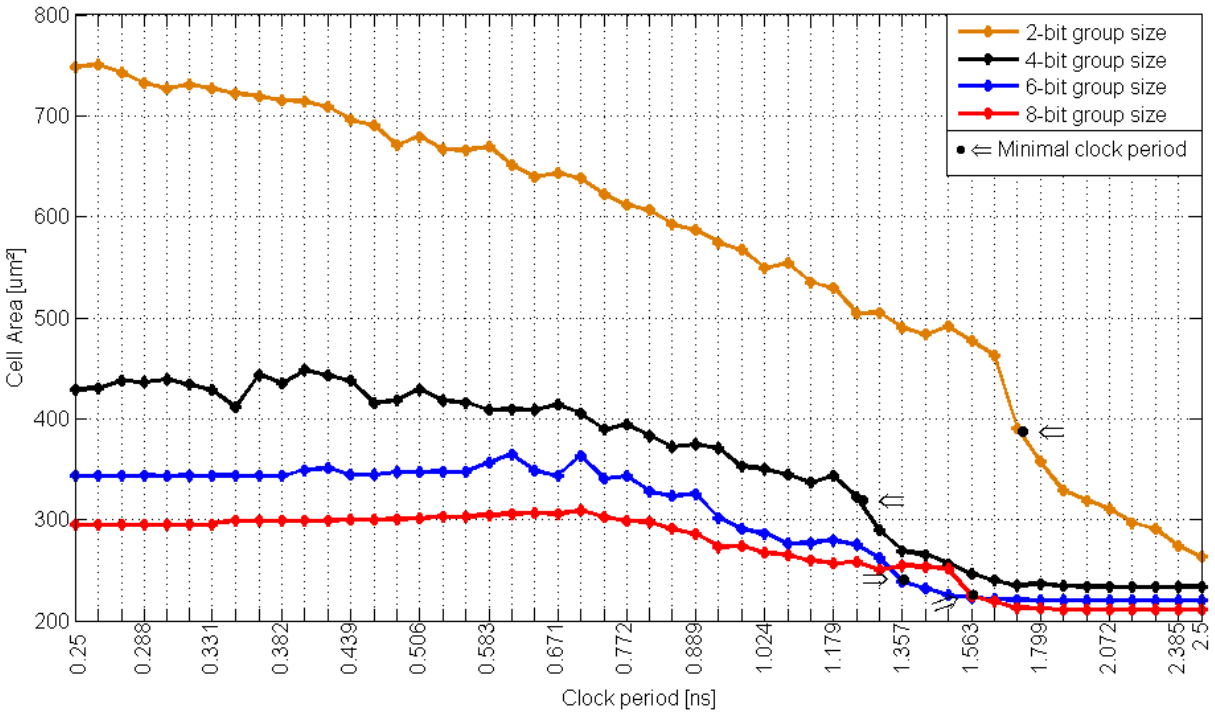


Figure A.2.3: Cell area comparison of 64-bit carry-skip adders with constant group sizes

As a direct consequence of the bigger cells, the 2-bit group adder consumes much more than the other schemes, as it can be seen in Figure A.2.4. Again, the 6-bit group adder has better results than the 4-bit group design in the frequency range where the timing constraints are met, especially at its maximum speed where it consumes 16% less than its counterpart. Besides, it presents an almost negligible difference in leakage power from the 8-bit scheme.

It was proved earlier that this architecture vastly better than the ripple-carry adder considering their respective results in the context of data sizes and technology used in this work. Clearly, the detailed study of group sizes offers a bigger variety of possibilities that can be used in the decision-taking process of the architecture selection based on the obtained QoR. Despite being the fastest scheme considered in this section, the 4-bit group size adder is outperformed by the 6-bit scheme since the latter has a smaller area and lower power leakage in its available frequency range. Another interesting conclusion that is not clearly found is that this architecture has a lower bound considering the group size, proved by the 2-bit group size data shown in these graphs.

## A.3 Group size impact in carry-select adders

The fact of being one of the fastest adder architectures without having a prefix tree is deeply tied to the group size used in the carry-select adder. Thus, to investigate the

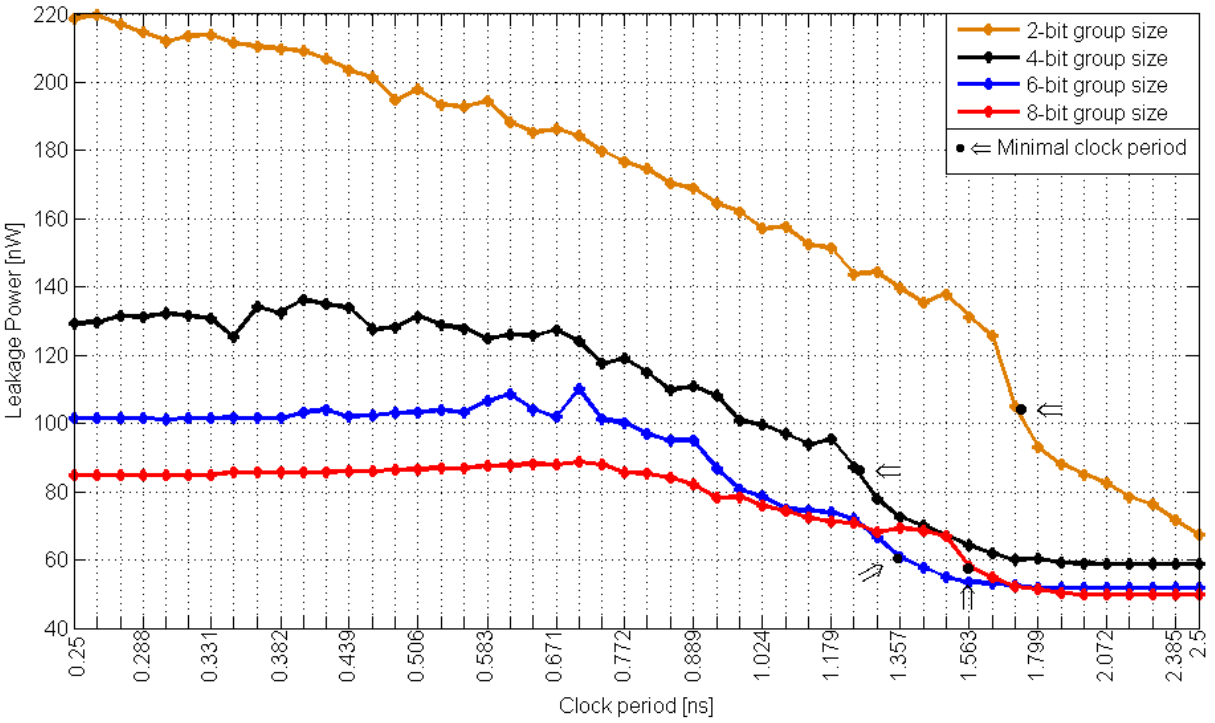


Figure A.2.4: Leakage power comparison of 64-bit carry-skip adders with constant group sizes

QoR evolution of this architecture in function of the chosen group sizes, two approaches are used: one based on fixed-size groups and another based on variable-sized groups following a linear progression.

### A.3.1 Constant group size approach

To obtain the optimal block size for a carry-select adder, let us take Figure A.3.1 to develop our theory. Analyzing this design and assuming that all groups have the same size, all sum slices will have finished their computation at the moment the first group calculates its result and start propagating the carry-out. Afterwards, this carry has to be propagated through the select logic in order to choose the right result for each group.

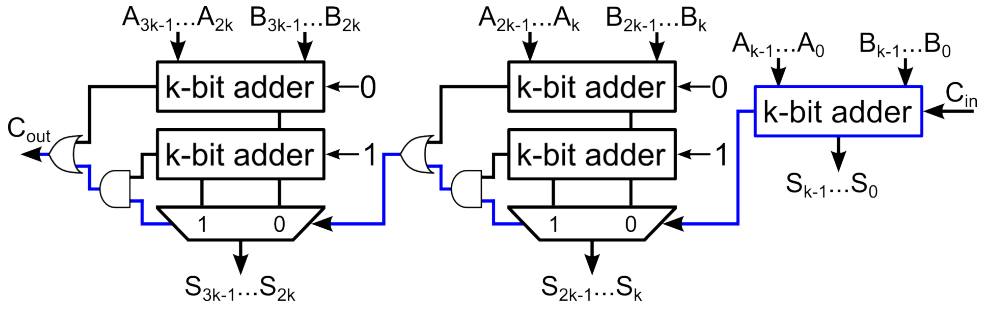


Figure A.3.1: Structure of a fixed-group size carry-select adder

Thus, assuming a $n$-bit carry-select adder divided into $k$-bit groups, its computation timing is given by Equation (A.3.1). The full adder delay ($T_{FA}$) is considered to be equal to 2D which corresponds to the same value of the select logic $T_{SL}$. Deriving this equation and equaling it to zero leads us to find the optimal block size, given by Equation (A.3.2). Substituting Equation (A.3.2) back in Equation (A.3.1), we observe that the timing complexity of this design is proportional to $\sqrt{n}$. In this case, we assumed that the carry select logic delay is equal to the multiplexer delay, thus only one multiplexer (output selection of the last group) is considered in the critical path.

$$T_{Total} = \overbrace{(k-1)T_{FA}}^{\text{Group delay}} + (n/k - 1)T_{SL} + \overbrace{D}^{\text{Mux}} \tag{A.3.1}$$

$$k_{opt} = \sqrt{n} \tag{A.3.2}$$

Theoretically, for a 64-bit adder, the optimal group size is equal to 8. However, due to timing model simplifications, Figure A.3.2 shows that this group size does not represent the best choice. The 2-bit group is 38% faster than the theoretical best group. In fact, as the group size becomes smaller, the adder speed increases around 13% comparing group-to-group sizes.

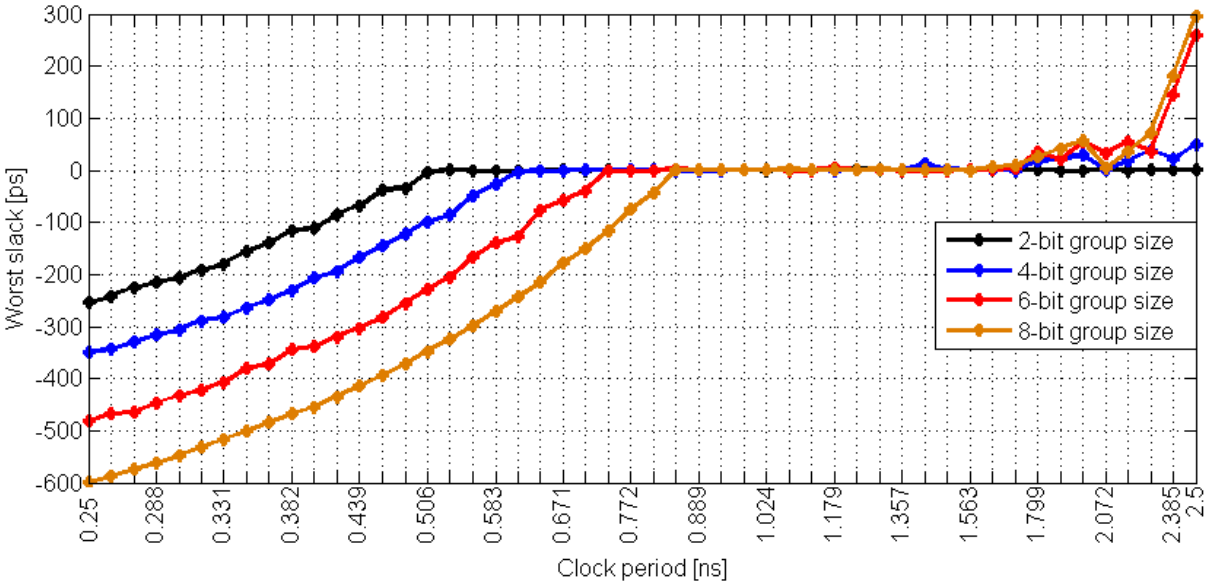


Figure A.3.2: Timing comparison of 64-bit carry-select adders with constant group sizes

As expected, reducing the group size increases the overhead caused by the carry select logic which leads to a larger design, as can be seen in Figure A.3.3. The 2-bit group scheme is clearly not area-efficient for lower frequencies as it occupies 14% more area than the second largest design. On the other hand, as the clock period is reduced (especially after 800ps), this design outperforms the other schemes because

the synthesis tool does not increase uncontrollably the cells' drive to meet the timing requirements. It is worth noting that the other designs – in their functional clock period range – do not present a great area difference among themselves (less than 5% for lower frequencies).

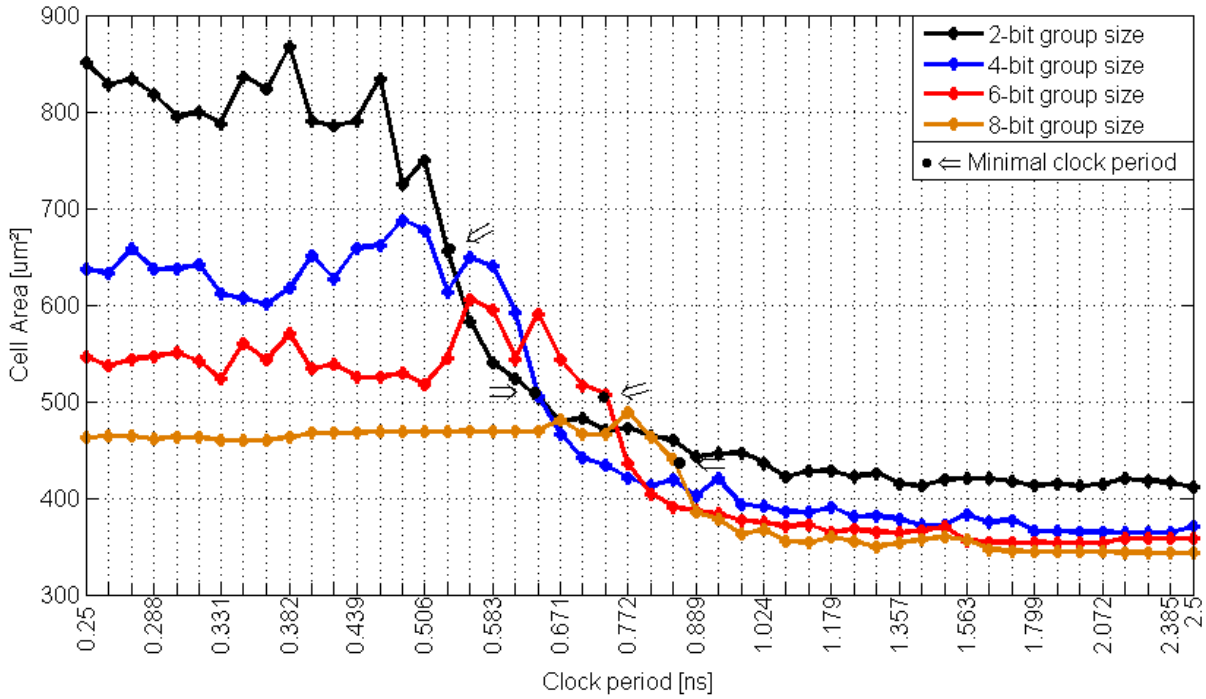


Figure A.3.3: Cell area comparison of 64-bit carry-select adders with constant group sizes

From Figure A.3.4 we observe that, despite being 18% faster, the 2-bit group design has to consume 25 % more power than the 4-bit group design at its maximum speed. Comparing the 4-bit and 8-bit schemes at their minimal clock period, the former consumes 12% more than the latter as a direct consequence of being 24% faster which, depending upon the system requirements, can be an interesting trade-off. Finally, the most inefficient design is the 6-bit group size since it is not the fastest adder and, in almost all its frequency, it is more power-hungry than the other designs.

### A.3.2 Variable group size approach

As stated in Section A.3.1, the adder timing depends on the group size since it will determine how long will be the ripple-carry chain and how much additional logic will be necessary to select the result. This approach can be optimized if different group sizes are allowed in order to match both adder group and the carry select timing. To do so, the timing model used here will consider that both full adder and the carry select logic have the same delay, although this hypothesis is rarely true.

If the first group is composed of two full adders, the second group has to have

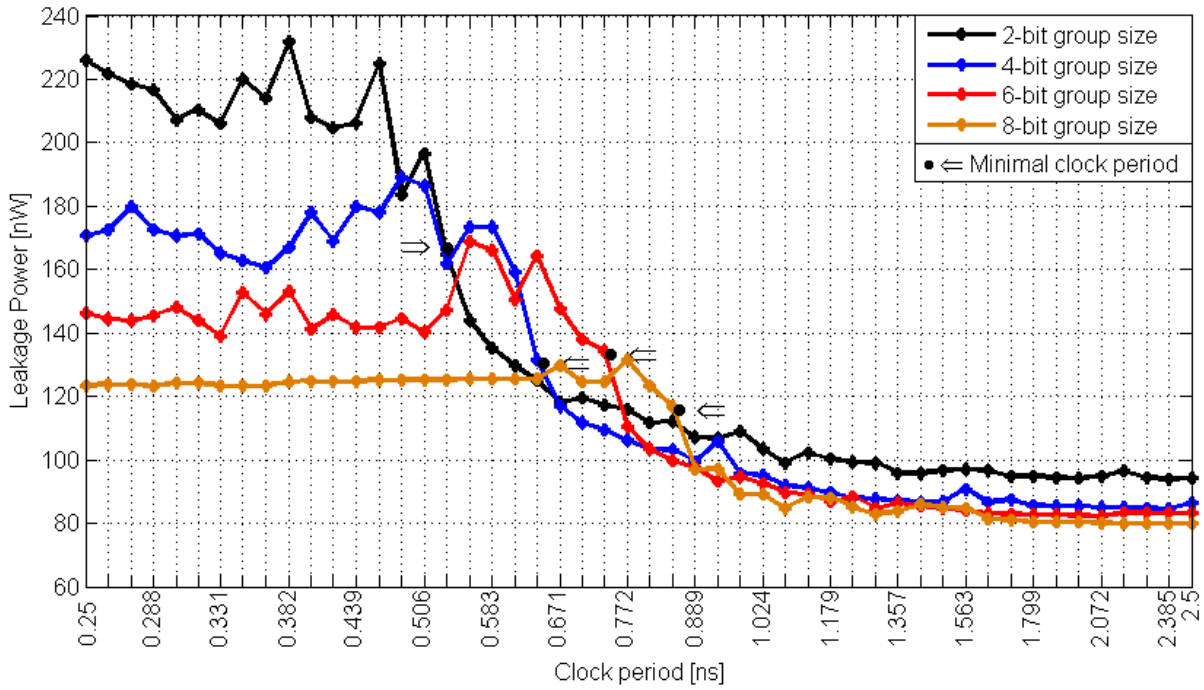


Figure A.3.4: Leakage power comparison of 64-bit carry-select adders with constant group sizes

the same size since the carry select logic is located after the second group. The size of the third group is increased by one to match the timing path, that is, the time of the two first full adders plus the additional delay of the select logic. The fourth group will have a carry-chain of four bits and so on. If we apply this technique to a 64-bit adder, the total delay will be equal to 22D while with optimal group size this delay would be equal to 29D.

However, it is hard to match the cells' timing as it depends on intrinsic gate delays as well as their load and driving capabilities. With this idea, some empirical growth rates were used on a linear progression with the first group size equal to two full adders. As these rates might generate real values, only the integer part of each progression element is taken as the group size.

Figure A.3.5 shows the timing evolution for the selected growth rates. Firstly, we observe that our simplified timing model is not valid. It is possible to deduce that the full adder and the select logic have different delays as the smaller the progression ration, the faster the circuit is. As we halved the ratio from 0.5 to 0.25, the circuit speed increased 17%. However, the speed increase is less pronounced (8%) as the ratio is halved again from 0.25 to 0.125.

Interestingly, the area variation from ratio-to-ratio is smaller in this approach than that observed for the fixed-group size seen earlier (see Figure A.3.6). In the frequency range where all designs meet the timing requirements, the area gap between the largest and smallest design does not pass 6%. Also, the fastest design occupies only 11%

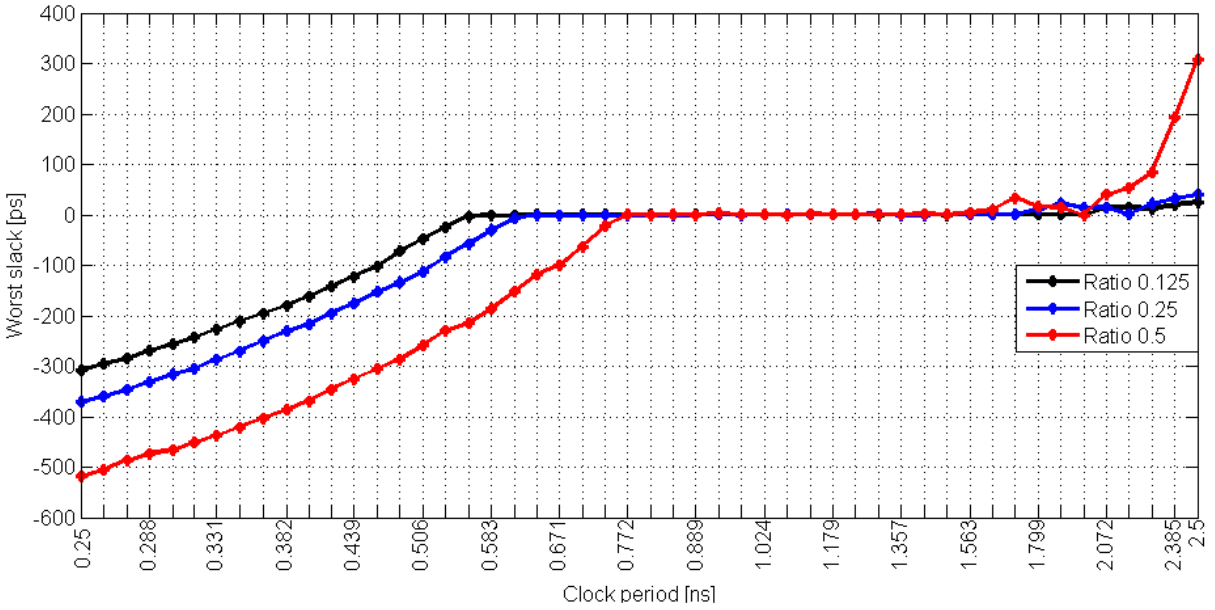


Figure A.3.5: Timing comparison of 64-bit carry-select adders with variable group sizes

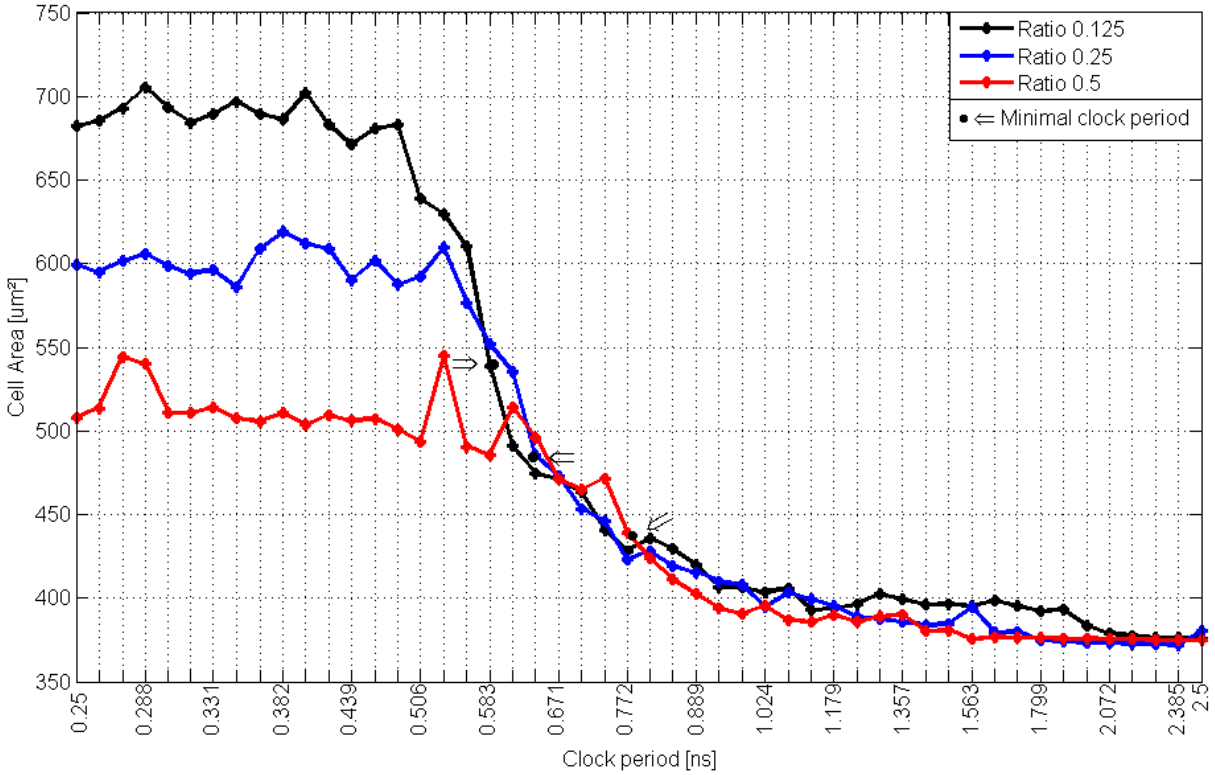


Figure A.3.6: Cell area comparison of 64-bit carry-select adders with variable group sizes

more area than the design with a 0.25 growth ratio.

From Figure A.3.7 we can analyze how leakage power varies with different growth ratios. Here, the artifacts created in the design by the synthesis tools are more visible, especially at the speed limit of each one of the tested designs. Also, the circuit from the

0.125 ratio consumes only 8% than the one obtained from the ratio 0.25 when both are at their minimal clock period. Hence, we can conclude that the best variation rate is the 0.125 due to the QoR of this design as it greatly increases the speed with acceptable penalties in area and power consumption.

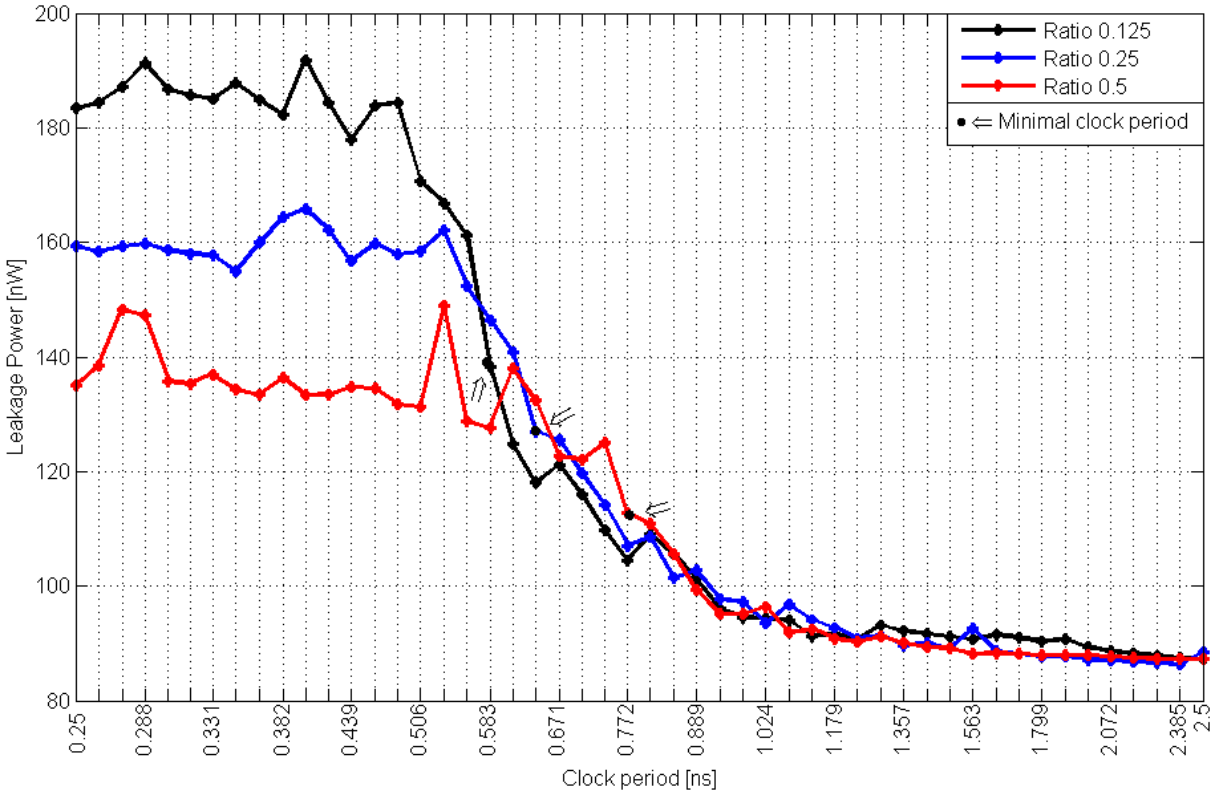


Figure A.3.7: Leakage power comparison of 64-bit carry-select adders with variable group sizes

# B Description of an Arithmetic Architecture in VHDLGen

To understand exactly how architectures are described using the VHDLGen framework, a simple example is shown in B.1. This example illustrates a carry save adder that compresses three inputs into two outputs without carry propagation. For this design, both input and output sizes are defined by the user when the script is executed. The object constructor (given by the function `__init__`) sets the inputs (`data1_i`, `data2_i`, `data3_i`) and outputs (`data_sout`, `data_cout`) that will define the entity of the RTL design according to the specified sizes. The function `gen_adder` instantiates all the components (full adders) and connect them together to produce the expected outputs.

In line 96, a special function is called as it is needed to build the testbench to validate the design. The first expression ("`s0+s1+s2`") represents which operations will be performed upon the inputs in the order specified previously. As this design produces two outputs to represent the sum, an additional operation needs to be performed upon the outputs (expression "`s0+s1`") in order to obtain a result that may be compared against the output of the first expression. Thus, the VHDLGen core will be able to create the necessary testbench to satisfy both expressions, ensuring the design correctness.

```
1  from gen_code_generator import *
2  from gen_entity import *
3  from gen_custom_fa import *
4  from gen_full_adder_stdcell import *
5  import math
6  import sys
7
8  class CarrySave(CodeGenerator):
9    def __init__(self, size_D1 = 32, size_D2 = 32, size_D3 = 32, size_SOUT = 33,
       size_COUT = 33, use_custom_fa = "false"):
10     CodeGenerator.__init__(self)
11     self.entity = Entity("adder_multioperand_arch", True)
12     self.entity.add_custom_lib("library lib_mppa_common_unit_netlist_fe;")
13     self.entity.add_custom_lib("use lib_mppa_common_unit_netlist_fe.all;")
14     self.use_custom_fa = use_custom_fa.lower()
15     if (self.use_custom_fa == "true"):
16       aux_name = "_custom_fa"
```

```
17      else:
18        aux_name = ""
19      self.arch = Architecture("arch_carry_save%s" % aux_name, self.entity.get_name())
20      self.add_design(self.entity, self.arch)
21      self.signals_list = []
22
23      # Define the interface
24      self.data1_i = InputSignal(signal_name = "data1_i", signal_size = size_D1)
25      self.data2_i = InputSignal(signal_name = "data2_i", signal_size = size_D2)
26      self.data3_i = InputSignal(signal_name = "data3_i", signal_size = size_D3)
27      self.data_sout = OutputSignal(signal_name = "data_sout", signal_size = size_SOUT
        )
28      self.data_cout = OutputSignal(signal_name = "data_cout", signal_size = size_COUT
        )
29      self.entity.add_input(signal_name = "clk")
30      self.entity.add_input(signal_name = "reset")
31      self.entity.add_input(self.data1_i)
32      self.entity.add_input(self.data2_i)
33      self.entity.add_input(self.data3_i)
34      self.entity.add_output(self.data_sout)
35      self.entity.add_output(self.data_cout)
36
37      #Compare inputs/outputs with binding signals to see how many instantiations
        should be done
38      self.max_input_size = self.entity.get_max_input_size()
39      self.min_input_size = min([input_signal.get_size() for input_signal in self.
        entity.get_inputs()
40                  if (input_signal.get_name() != "cin") and (input_signal.get_name()
        != "reset")
41                    and (input_signal.get_name() != "clk")])
42
43    def get_architecture_name(self):
44      return self.arch.get_name()
45
46    def get_entity_name(self):
47      return self.entity.get_name()
48
49    def get_inputs(self):
50      return self.entity.get_inputs()
51
52    def get_outputs(self):
53      return self.entity.get_outputs()
54
55    def set_entity_name(self, entity_name):
56      self.entity.set_name(entity_name)
57      self.arch.set_entity_name(entity_name)
58
```

```
59    def gen_adder(self):
60      # Internal signals
61      signal_sout = Signal("s_sout", self.max_input_size)
62      signal_cout = Signal("s_cout", self.max_input_size)
63      self.arch.add_signal(signal_sout)
64      self.arch.add_signal(signal_cout)
65
66      # Check whether use the FA defined by the EDA tool or use custom FA
67      if (self.use_custom_fa == "true"):
68        adderCell = CustomFullAdder()
69      else:
70        adderCell = FullAdder()
71      adderCell.add_input_binding(self.data1_i, "A")
72      adderCell.add_input_binding(self.data2_i, "B")
73      adderCell.add_input_binding(self.data3_i, "CI")
74      adderCell.add_output_binding(signal_sout, "SUM")
75      adderCell.add_output_binding(signal_cout, "CO")
76      self.arch.add_component(adderCell)
77
78      # Concatenation operation
79      concat1 = Concatenation()
80      concat1.add_input(signal_sout, index=(self.max_input_size-1, 0))
81      concat1.set_slicing(signal_sout.get_size(), signal_sout.get_size(), self.
        data_sout.get_size())
82      concat1.add_constant_input("0-s0")
83      concat1.add_output(self.data_sout)
84
85      concat2 = Concatenation()
86      concat2.add_input(signal_cout, index=(self.max_input_size-1, 0))
87      concat2.set_slicing(signal_cout.get_size(), signal_cout.get_size(), self.
        data_cout.get_size())
88      concat2.add_constant_input("s0-0")
89      concat2.add_output(self.data_cout)
90
91      # Add chains
92      self.arch.add_operation(concat1)
93      self.arch.add_operation(concat2)
94
95      # Create the bench
96      self.add_tuple_test([self.data1_i, self.data2_i, self.data3_i], "s0+s1+s2", [
        self.data_sout, self.data_cout], "s0+s1!== result0")
97
98  # Gen Adder
99  if __name__ == "__main__":
100   # Set the parameters
101   if len(sys.argv) > 1:
102     use_custom_fa = sys.argv[1]
```

```
103     data1_width = int(sys.argv[2])
104     data2_width = int(sys.argv[3])
105     data3_width = int(sys.argv[4])
106     data_sout_width = int(sys.argv[5])
107     data_cout_width = int(sys.argv[6])
108
109   else:
110     sys.exit(1)
111
112   csAdder = CarrySave(data1_width, data2_width, data3_width, data_sout_width,
        data_cout_width, use_custom_fa)
113   csAdder.gen_adder()
114   csAdder.set_architecture_typename("adder_multioperand")
115   rtl_file_name = "adder_multioperand.%s_%s-%s-%s-%s-%s.vhd" % (csAdder.
        get_architecture_name(), data1_width, data2_width, data3_width, data_sout_width,
         data_cout_width)
116   rtl_file = open(rtl_file_name, 'w')
117   rtl_code = csAdder.generate_rtl()
118   rtl_file.write(rtl_code)
119   rtl_file.close()
120
121   bench_if_code = csAdder.generate_interface_bench()
122   bench_if_file_name = "tb_mppa_adder_multioperand.%s_%s-%s-%s-%s-%s.sv" % (csAdder.
        get_architecture_name(), data1_width, data2_width, data3_width, data_sout_width,
         data_cout_width)
123   bench_if_file = open(bench_if_file_name, 'w')
124   bench_if_file.write(bench_if_code)
125   bench_if_file.close()
126
127   bench_stim_code = csAdder.generate_stimulus_bench()
128   bench_stim_file_name = "tb_mppa_adder_multioperand.%s_tests_%s-%s-%s-%s-%s.sv" % (
        csAdder.get_architecture_name(), data1_width, data2_width, data3_width,
        data_sout_width, data_cout_width)
129   bench_stim_file = open(bench_stim_file_name, 'w')
130   bench_stim_file.write(bench_stim_code)
131   bench_stim_file.close()
```

Listing B.1: Carry Save architecture description in VHDLGen