

**MASTER**

**Energy efficient multi-granular arithmetic in a coarse-grain reconfigurable architecture**

Louwers, S.T.

*Award date:*  
2016

[Link to publication](#)

**Disclaimer**

This document contains a student thesis (bachelor's or master's), as authored by a student at Eindhoven University of Technology. Student theses are made available in the TU/e repository upon obtaining the required degree. The grade received is not published on the document as presented in the repository. The required complexity or quality of research of student theses may vary by program, and the required minimum study period may vary in duration.

**General rights**

Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain

# Energy Efficient Multi-Granular Arithmetic in a Coarse-Grain Reconfigurable Architecture

Stef Louwers

30 May 2016

## Abstract

*Coarse-Grain Reconfigurable Architectures* (CGRAs) are a class of architectures that can be dynamically adapted to match an application, similar to *Field Programmable Gate Arrays* (FPGAs). Unlike FPGA systems, which can be programmed at the gate level, CGRA systems can be programmed as a network of higher level operations such as addition and multiplication. By being configurable at a coarser granularity, these systems are much more energy efficient than an FPGA, but this comes at a loss of adaptability.

In a CGRA system, the width of the configurable operation units is traditionally a difficult design decision. If this width is too narrow, not all operations are natively possible, and software support is required to calculate larger operations. On the other hand, if the width is too wide, energy is wasted on the computation of unnecessary operand bits.

One way of solving this issue is by designing the operation circuits such that several such units can be combined efficiently to form a single, bigger arithmetic unit. Each operation performed by the application can then be computed by a combined arithmetic unit of the exact width required by the application. Computing wide operations this way is not as efficient as a native wide circuit, but the upside of this approach is that narrower operations can be performed much more efficiently than in the alternative design. We call this concept *Multi-Granular Arithmetic*.

In this report, we investigate the details of performing common arithmetic operations in a multi-granular setting in the context of the BLOCKS CGRA architecture. For the operations of addition, accumulation, multiplication, and multiply-accumulation, we show that the multi-granular design is feasible, with a very modest efficiency cost for wide operations, and substantial efficiency gains for narrow operations. Using a silicon synthesis-toolflow analysis, we demonstrate the ability to perform a narrow multiplication at an energy cost 15 times lower than the native alternative under realistic conditions, with an energy cost of a factor 1.5 for performing the matching wide multiplication.

# Contents

<b>1</b>	<b>Introduction</b>	<b>6</b>
<b>2</b>	<b>Background</b>	<b>8</b>
2.1	Flexibility versus Energy Efficiency . . . . .	9
2.2	Signal Processing . . . . .	10
2.3	State of the Art . . . . .	11
2.3.1	ASIC . . . . .	11
2.3.2	FPGA . . . . .	11
2.3.3	CPU . . . . .	12
2.3.4	DSP . . . . .	12
2.3.5	VLIW . . . . .	12
2.3.6	SIMD . . . . .	13
2.3.7	GPU . . . . .	13
2.4	CGRA . . . . .	13
2.4.1	PipeRench . . . . .	14
2.4.2	MATRIX . . . . .	15
2.4.3	RAW Machine . . . . .	15
2.5	Multi-Granular Arithmetic . . . . .	15
2.6	BLOCKS Architecture . . . . .	17

<b>3 Multi-Granular Arithmetic</b>	<b>19</b>
3.1 Frugal Arithmetic . . . . .	19
3.2 Multi-Granular Arithmetic . . . . .	21
3.3 Composing Operations . . . . .	23
3.4 Multi-Granular Arithmetic in BLOCKS . . . . .	26
<b>4 Experimental Methodology</b>	<b>27</b>
4.1 Verilog Structure . . . . .	27
4.2 Tools . . . . .	28
4.3 Results . . . . .	29
<b>5 Addition</b>	<b>34</b>
5.1 Addition Algorithms . . . . .	34
5.1.1 Ripple-Carry Adder . . . . .	35
5.1.2 Carry-Lookahead Adder . . . . .	36
5.1.3 Carry-Select Adder . . . . .	38
5.2 Multi-Granular Addition . . . . .	39
5.2.1 Base Algorithms . . . . .	40
5.2.2 Composition Algorithms . . . . .	40
5.3 Multi-Granular Adder Configurations . . . . .	41
5.3.1 Ripple-Carry Composition, Ripple-Carry Base . . . . .	42
5.3.2 Ripple-Carry Composition, Carry-Lookahead Base . . . . .	45
5.3.3 Carry-Select Composition, Ripple-Carry Base . . . . .	48
5.3.4 Carry-Select Composition, Carry-Lookahead Base . . . . .	50
5.4 Comparison . . . . .	52
5.5 Conclusions . . . . .	57

<b>6 Accumulation</b>	<b>58</b>
6.1 Accumulation Algorithms . . . . .	58
6.1.1 Adder-Accumulator . . . . .	58
6.1.2 Carry-Save Accumulator . . . . .	59
6.2 Multi-Granular Accumulation . . . . .	61
6.2.1 Ripple-Carry Accumulator . . . . .	61
6.2.2 Carry-Accumulation . . . . .	62
6.3 Multi-Granular Accumulator Configurations . . . . .	64
6.3.1 Ripple-Carry Composition, Carry-Lookahead Base . . . . .	65
6.3.2 Ripple-Carry Composition, Carry-Save Base . . . . .	68
6.3.3 Carry-Accumulate Composition, Carry-Lookahead Base .	71
6.3.4 Carry-Accumulate Composition, Carry-Save Base . . . . .	74
6.4 Comparison . . . . .	77
6.5 Conclusions . . . . .	82
<b>7 Multiplication</b>	<b>83</b>
7.1 Multiplication Algorithms . . . . .	83
7.1.1 Signed Multiplication . . . . .	84
7.1.2 Output Formats . . . . .	85
7.2 Multi-Granular Multiplication . . . . .	86
7.2.1 Signed Multi-Granular Multiplication . . . . .	87
7.2.2 Half-Width Multi-Granular Multiplication . . . . .	88
7.2.3 Partial Product Addition . . . . .	89
7.2.4 Adder Tree . . . . .	90
7.2.5 Accumulator . . . . .	93
7.3 Multi-Granular Multiplier Configurations . . . . .	95
7.3.1 Single-Cycle Multiplier with Adder Tree . . . . .	95
7.3.2 Dual-Cycle Multiplier with Adder Tree . . . . .	99

7.3.3	Single-Cycle Multiplier with Accumulator . . . . .	102
7.3.4	Dual-Cycle Multiplier with Accumulator . . . . .	105
7.3.5	Standalone Multiplier . . . . .	108
7.4	Comparison . . . . .	111
7.5	Conclusions . . . . .	117
<b>8</b>	<b>Multiply-Accumulation</b>	<b>118</b>
8.1	Multiply-Accumulation Algorithms . . . . .	118
8.2	Multi-Granular Multiply-Accumulation . . . . .	119
8.2.1	Multiply-Accumulation with Accumulator . . . . .	119
8.2.2	Distributed Multiply-Accumulator . . . . .	119
8.3	Multiply-Accumulate Configurations . . . . .	120
8.3.1	Single-Cycle Accumulator-based MAC . . . . .	123
8.3.2	Dual-Cycle Accumulator-based MAC . . . . .	128
8.3.3	Distributed Multiply-Accumulator . . . . .	133
8.4	Comparison . . . . .	138
8.5	Conclusions . . . . .	143
<b>9</b>	<b>Design for the Application</b>	<b>144</b>
9.1	Granularity Model . . . . .	144
9.2	Interconnect . . . . .	145
9.3	Comparing Architectures . . . . .	146
9.4	Improving the Model . . . . .	146
<b>10</b>	<b>Conclusions and Future Work</b>	<b>147</b>
10.1	Optimising Code for Multi-Granular Architectures . . . . .	148
10.2	Interconnect Considerations . . . . .	149
10.3	Future Work . . . . .	151

<b>A Benchmark Results</b>	<b>162</b>
A.1 Addition . . . . .	163
A.1.1 Ripple-Carry Composition, Ripple-Carry Base . . . . .	164
A.1.2 Ripple-Carry Composition, Carry-Lookahead Base . . . . .	167
A.1.3 Carry-Select Composition, Ripple-Carry Base . . . . .	170
A.1.4 Carry-Select Composition, Carry-Lookahead Base . . . . .	173
A.2 Accumulation . . . . .	176
A.2.1 Ripple-Carry Composition, Carry-Lookahead Base . . . . .	177
A.2.2 Ripple-Carry Composition, Carry-Save Base . . . . .	179
A.2.3 Carry-Accumulate Composition, Carry-Lookahead Base .	181
A.2.4 Carry-Accumulate Composition, Carry-Save Base . . . . .	184
A.3 Multiplication . . . . .	186
A.3.1 Single-Cycle Multiplier With Adder Tree . . . . .	187
A.3.2 Dual-Cycle Multiplier With Adder Tree . . . . .	189
A.3.3 Single-Cycle Multiplier With Accumulator . . . . .	192
A.3.4 Dual-Cycle Multiplier With Accumulator . . . . .	194
A.4 Multiply-Accumulation . . . . .	196
A.4.1 Single-Cycle MAC With Accumulator . . . . .	198
A.4.2 Dual-Cycle MAC With Accumulator . . . . .	200
A.4.3 Distributed Multiply-Accumulator . . . . .	203

# Chapter 1

## Introduction

When designing a low-power information-processing embedded device, there is a trade-off between the ability to redesign the implemented functionality after production, and the energy required to power the device. Systems based on FPGAs can be reprogrammed as a response to updated designs, and as such are very suitable for applications where future functionality updates are expected, but use a considerable amount of energy in order to provide this flexibility. In contrast, custom-designed integrated circuits — ASICs — are by definition very close to optimal in terms of energy efficiency, but provide very limited capabilities to accommodate updates. If an application demands both high energy efficiency and the ability to reconfigure functionality in the field, few good architectural options are available to suit this use case.

A *Coarse-Grain Reconfigurable Architecture* (CGRA) tries to fill the void between the flexibility of an FPGA, and the energy efficiency of an ASIC. It achieves this by being configurable at a coarser granularity than an FPGA: where an FPGA consists of individually configurable bit-level operations, a CGRA system contains units such as 8-bit or 16-bit operations that can be field-programmed in a way similar to an FPGA. Instead of bit-level operations, these configurable units implement relatively complex operations such as addition and multiplication.

One of the challenges in designing a CGRA system is selecting the basic *word size*, which is the bit-width of the registers, the data communicated over the interconnect network, and the operand size of arithmetic and logic operations. If this word size is too large, a considerable portion of the transmitted and computed bits are superfluous, which means energy spent on computing these bits is wasted. On the other hand, if the word size is too small, large computational operations have to be performed in software over the course of several cycles, which is even less energy efficient. This problem can only be avoided to a limited degree by analysing the application, as applications generally process data consisting of a mix of different bit-sizes; for example, an application might deal in both 8-bit and 32-bit units of data, and perform arithmetic operations on both of them.

The BLOCKS design is a CGRA architecture that aims to store, transport and process data using the bit-size required by the application. It accomplishes this by internally using words of some small width — say, 8-bits wide — as the size of registers and interconnect systems, and composing multiple of such words together into broader pieces of data as demanded by the application. Likewise, arithmetic and logic operations are implemented as functional units that take words of this word-size as input. Operations on larger inputs are implemented by composing several such smaller functional units, a construction we call *multi-granular arithmetic*.

In this report, we will explore techniques for constructing multi-granular compositions for several commonly used arithmetic and logic operations; for example, we describe the construction of a 32-bit adder as a composition of four 8-bit adders. In this analysis, we focus on designs that are a good fit for the BLOCKS architecture; however, our results can also be applied to other contexts.

After introducing the problem context and relevant background in Chapter 2, we formally introduce the notion of multi-granular arithmetic in Chapter 3. After a consideration of the consequences of the BLOCKS architecture for our multi-granular designs, we follow with a description of the experimental setup in Chapter 4. We analyse in detail the multi-granular implementation of the operations of *addition*, *accumulation*, *multiplication*, and *multiply-accumulation*, respectively described in Chapters 5, 6, 7 and 8. In Chapter 9, we summarise the previous four chapters by constructing a model that can be used to determine the optimal design for an envisioned set of applications. We conclude with Chapter 10, in which we summarise our main results, and give an overview of work that remains to be done in this area.

# Chapter 2

## Background

Embedded systems in medical patient monitoring devices monitor the health of (hospitalised) patients. Often the need for monitoring continues after the patients have left the hospital bed, so it should be possible for the patients to carry these systems on their body in order to improve their quality of life. The sensors of these monitoring devices generate a large amount of data that needs to be processed. One approach would be to transmit this data to an external server for processing; however, this is not feasible within the severely limited energy budget of these wearable devices. It would also limit the freedom of movement of the patient, as they have to stay near an access point. Thus ideally, the data must be processed on a battery powered processor that the patient can wear on his or her body.

This requires a low-power processor, and because the algorithms required for these applications are computation-intensive, the processor must also be energy efficient. Therefore, one might suggest to develop an ASIC, as they are very energy efficient. The algorithms to analyse and process the sensor data, however, are in active development and improvements are to be expected in the foreseeable future. This means that developing an ASIC is not feasible, as this chip might not be compatible with improved versions of the algorithms. Also, the relatively long and expensive design process of an ASIC slows down the development process of these monitoring devices. For some of these algorithms, such as EEG and ECG processing, there is currently a trend towards adaptive algorithms, where the algorithm adapts its structure depending on characteristics of each individual patient. This makes an ASIC even more unsuitable, as it is impossible to develop an ASIC for each individual patient. So in order to make it possible to update the devices, and to support adaptive algorithms, the processor must be more flexible than an ASIC can provide.

An FPGA is very flexible, as it offers bit-level reconfiguration of the datapaths and operations. This way any digital circuit can be created. This gives a lot of flexibility, but this flexibility reduces the performance per watt due to increased overhead costs, which limits battery life.

These patient monitoring devices process data from a lot of sensors in parallel, which means that there is parallelism available that we can exploit. There is both data level parallelism [10] (which can benefit from an *Single Instruction, Multiple Data* (SIMD) architecture), as well as instruction level parallelism [5], which benefit from a *Very Long Instruction Word* (VLIW) architecture. However, the ratio between data and instruction level parallelism might change during the development of the algorithms. These architectures all have some strong points, but it would be ideal if they all can be combined in one architecture that has some of the flexibility of an FPGA, and can gradually change from a VLIW to an SIMD architecture, with an ASIC-like performance per watt.

Such an architecture would allow rapid application development, and a short time to market with reduced costs because the same chip can be used for development and production, while the focus on energy efficiency should ensure that the battery life of the product is as long as possible. Because of its flexibility, this design allows reuse for related (and possibly unrelated) application domains.

## 2.1 Flexibility versus Energy Efficiency

Flexibility is the ability of an architecture to efficiently adapt to new or changing applications and algorithms. This can be roughly defined as the number of applications that can be executed at modal performance on a given architecture. A more formal model for flexibility would support strong objectively comparisons of the flexibility of different architectures, but this is outside the scope of this report.

There is a trade-off between the flexibility of an architecture on one hand, and the energy efficiency of said architecture on the other hand. Adding flexibility will require more power, as more flexibility means that there is more choice, and each choice requires extra logic. These extra costs for flexibility can decrease the energy efficiency by multiple orders of magnitude compared to a completely dedicated circuit [30].

Energy efficiency could be defined as the energy needed by an architecture to run a certain application. This number can easily be determined by benchmarking the application, but to allow a fair comparison between applications, this number should be normalised. Therefore, we will normalise the energy efficiency to *energy per operation*.

The defined metrics can be used to quantify possible architectures, and determine which architectures are Pareto-optimal on the flexibility and energy efficiency trade-off curve.

## 2.2 Signal Processing

Embedded systems in medical patient monitoring devices tend to use a lot of algorithms that can be classified as signal processing algorithms, which share many properties that can be used to optimise the processor architecture. Signal processing applications typically have several forms of parallelism available that can be exploited. For instance, these algorithms often use vector functions, map and reduce functions, or complex functions that can be expressed in terms of other “basic” operations.

The availability of multiple parallel data streams in the algorithm is called *Data Level Parallelism* (DLP) [10]. This form of parallelism can be exploited by processing several data streams in parallel, where each processing element is executing the same operation, each on a different data stream.

Another form of parallelism is called *Instruction Level Parallelism* (ILP) [5]. In this form of parallelism, multiple instructions are available for execution at the same time, because these instructions do not have any “data-dependencies” between them, i.e. the result of these operations are not needed as the input of the other operations. This form of parallelism can be exploited by executing multiple operations in parallel. In contrast to DLP, these operations do not have to be the same.

The vector and reduce functions frequently use accumulate operations, and thus benefit from hardware support for add-accumulate and multiply-accumulate operations. For example, the dot product is defined as  $\mathbf{A} \cdot \mathbf{B} = \sum_{i=1}^n A_i B_i$ , and maps exactly on an  $n$ -stage multiply-accumulator.

Complex functions are functions that consist of several “basic” operations. Where basic operations can be calculated directly, complex functions require several computation steps; for example Cooley-Tukey’s FFT-algorithm [6] computes the term  $X_k = \sum_{n=0}^{N-1} x_n e^{-\frac{2\pi i}{N} nk}$ . Calculations such as these benefit from an architecture that is able to chain these operations efficiently.

An architecture targeted at signal processing applications should ideally combine all these optimisations. So it should be able to exploit both DLP and ILP. Additionally, it should support add-accumulate and multiply-accumulate operations efficiently, and be able to combine and chain basic operations to form complex functions.

The application domain that we will consider consists of signal processing applications, primarily targeted at the EEG and ECG domains. Additionally, applications from the following domains will be considered part of the application domain: vision and image processing, telecommunications, machine learning and linear algebra. This broad list of applications underlines that we strive to develop a very flexible architecture, capable of executing very different workloads in an energy efficient fashion.

## 2.3 State of the Art

We can classify existing architectures by the point they occupy on the flexibility/energy efficiency scale. The most important groups are displayed in Figure 2.1. We briefly discuss the properties of these architectures in the remainder of this section.

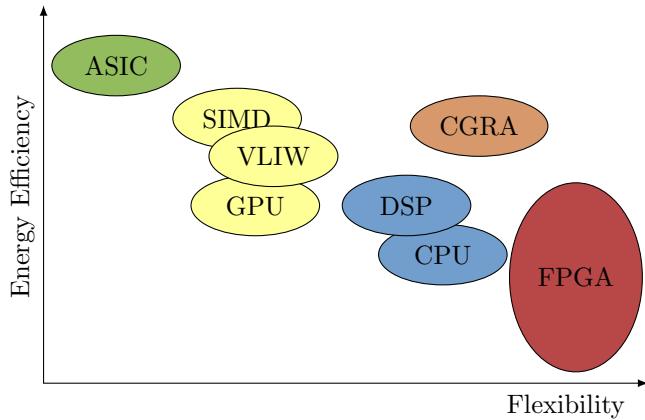


Figure 2.1: Flexibility versus energy efficiency trade-off for state-of-the-art architectures.

### 2.3.1 ASIC

*Application-Specific Integrated Circuit* (ASIC) chips are chips designed for a specific application. They are designed to do one task, and do them well. This gives them a very high energy efficiency, and also a good performance. But as they are designed for a single task, they are not flexible at all. Often these chips are not able to run any other application. The custom design also gives these chips high development costs.

Because our application domain demands an architecture where the application logic can be reconfigured during the lifetime of the product, ASICs cannot be used as an architecture for our application domain.

### 2.3.2 FPGA

The *Field-Programmable Gate Array* (FPGA) is on the other end of the spectrum. These chips are designed such that they can implement every possible logic circuit, making them very flexible. This is possible because this architecture is reconfigurable at the bit-level. However, all these configuration options have their impact on the energy efficiency of the chip. The energy efficiency can differ greatly between application running on an FPGA. If an FPGA just used to implement regular adders and multipliers, its efficiency is very low.

For algorithms that can benefit from (complex) custom logic circuits, this efficiency can be boosted as these circuits can be implemented at a very low level on an FPGA. Thus, the high flexibility comes at a cost of configuration overhead, but the adaptation to the application can boost the overall energy efficiency to a higher level than a CPU.

Our application domain uses a lot of signal processing, addition and multiplication, which are operations that require a great deal more energy when implemented in an FPGA than they would as implemented in a dedicated circuit [8, 16, 24]. This makes FPGAs a poor fit for our application domain, as a lot of energy is wasted on inefficient implementations of these basic constructs.

### 2.3.3 CPU

A *Central Processing Unit* (CPU) is also quite flexible, as it can be programmed to execute any program. Because they are commonly available, they are cheap and available in many variations, varying from high performance to low power. They are also easy to program.

However, the general purpose character of these chips makes it impossible to use accelerators that are targeted at the application domain, and often multiple instructions are needed for a calculation that could have been done in a single instruction with dedicated hardware. Similarly, the fact that each unit of processed data needs to pass through a register file before being available for further processing introduces inefficiency that is ideally avoided. For this reason, while CPUs are a usable architecture for our problem domain, there are many opportunities for a more optimised design.

### 2.3.4 DSP

A *Digital Signal Processor* (DSP) is a type of CPU optimised for digital signal processing. Often they have more parallel memory interfaces in order to provide a high memory bandwidth. They are also equipped with specialised accelerators frequently used in signal processing applications, such as multiply-accumulate units.

This architecture is already quite suited for our application domain, as this contains mostly signal processing applications. However, DSPs are not equipped to exploit the DLP and ILP available in our application domain, and also are unable to use spatial layouts and software pipelining to save energy.

### 2.3.5 VLIW

The *Very Long Instruction Word* (VLIW) architecture takes advantage of *Instruction Level Parallelism* (ILP). It has large instruction words, that can control multiple functional units in parallel. Where an instruction on a traditional

CPU encodes a single operation, a VLIW instruction encodes multiple operations; each instruction encodes operations for each execution unit of the device. This allows the compiler or programmer to exploit the available instruction level parallelism explicitly.

For our application domain, we do not want to be limited to a pure VLIW architecture, as we also expect some DLP that we would like to exploit. However, the concept of wide instructions that allow the execution of multiple parallel execution paths certainly applies to our application domain, and makes a design property that should ideally be incorporated in any architecture serving our application domain.

### 2.3.6 SIMD

The *Single Instruction, Multiple Data* (SIMD) architecture takes advantage of *Data Level Parallelism* (DLP). Each instruction is executed by multiple ALUs on different data elements; this is sometimes called a vector processor. This architecture excels at parallel processing multiple data streams with the same instructions, such as vector operations. If not all parallelism is used, its efficiency drops quickly.

The SIMD architecture does not offer enough flexibility for our application domain, as it is only able to exploit DLP, and our application domain also contains ILP that we would like to exploit. And when no DLP is available in a code section, the efficiency of a SIMD is very poor. Like VLIW, the SIMD concept is one that should be present in our ideal architecture.

### 2.3.7 GPU

A GPU is a special case of an SIMD, called SIMT (*Single Instruction, Multiple Threads*), and is mostly focused on raw parallel processing power, and energy efficiency subordinately. GPUs have large register files, extremely high memory bandwidth and support runtime scheduling. As our focus is on energy efficiency, this architecture is not suited for our application domain.

## 2.4 CGRA

A *Coarse Grain Reconfigurable Architecture* (CGRA) is an architecture for a type of processor that can be reconfigured at runtime, much like an FPGA. However, where an FPGA is configured at the gate level, a CGRA is configurable at the *Functional Unit* (FU) level. A functional unit is a generalised ALU; however, its exact definition differs between CGRA architectures. For example, some functional units also contain registers or an instruction decoder. Changing the reconfiguration point from the gate level to the functional unit level keeps

a lot of the flexibility of an FPGA to construct compute platforms specialised for a given application, but gives a significantly smaller configuration overhead.

Some CGRA designs enable the creation of a spatial layout for an application. That means the functional units are configured at the start of the application or even at the start of a loop, and execute (as much as possible) the same instruction, while the data flows through the compute network. The major energy advantage is that the functional unit keep computing the exact same instruction for many cycles, reducing accesses to the instruction memory and keeping the toggling of control signals to a minimum.

Furthermore, by separating the *Instruction Decoders* (ID) from the functional units, it is possible to group the functional units into SIMD- and VLIW-like structures. In general SIMD exploits *Data Level Parallelism* (DLP), while VLIW exploits *Instruction Level Parallelism* (ILP). How much DLP and ILP is present heavily depends on each particular application. Because a CGRA can be reconfigured, the right mix of SIMD and VLIW structures can be chosen for each application to achieve a high energy efficiency, as illustrated in Figure 2.2.

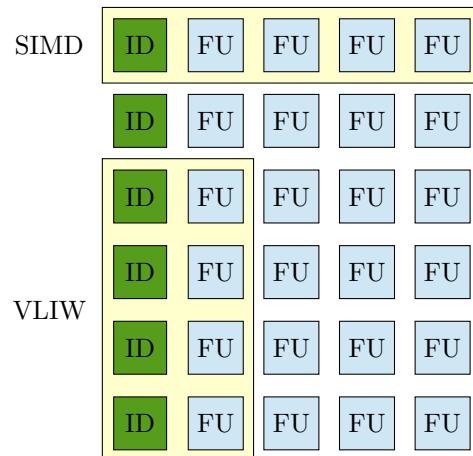


Figure 2.2: A mapping of an SIMD and VLIW instruction on a CGRA architecture.

In the literature, the term CGRA is used for a wide range of architectures, and thus many different definitions exist. Hartenstein [13] gives an overview of the research done on coarse-grain reconfigurable computing in 2001, and Kim [14] updates this overview in 2011. Some notable CGRA architectures are PipeRench [11], MATRIX [17] and RAW Machines [27]. These architectures will briefly be discussed in the following sections.

#### 2.4.1 PipeRench

The PipeRench [11] architecture is focused on creating configurable computational pipelines. These pipelines are based on a chain of *Processing Elements* (PE), which contain an ALU and a register file. The instructions for these PEs

are loaded statically at the configuration stage. The output of each PE can go to the next stage, either directly to the PE in the same column, or to any other PE in the next stage via a interconnect network, or it can repeat the same stage.

There is a carry chain between the PEs on the same stage, which allows larger operations to be constructed from multiple PEs. However no details are given on how this can be done, and what the impact is on the energy usage.

#### 2.4.2 MATRIX

The MATRIX [17] architecture is a multi-granular array of 8-bit *Basic Functional Units* (BFU). Each BFU can serve as instruction memory, data memory, register file or ALU. The interconnect is organised in three levels: 12 nearest neighbours, length four bypass connections, and global interconnect lines spanning entire rows or columns. The BFUs are also connected by a carry chain that allows wide-word addition operations. Wider multiply operations are also possible, but the techniques are not explicitly mentioned. Each BFU is equipped with a dual-cycle output 8-bit multiplier.

#### 2.4.3 RAW Machine

The RAW Machine [27] architecture consists of a  $4 \times 4$  grid of identical tiles. Each tile consists of a 32-bit ALU, instruction and data memory, a register file and a programmable switch to communicate with the other tiles. These switches connect the tiles in horizontal and vertical lines, and how packages are routed depends on the switching program. This can both be a static schedule, or a dynamic schedule with data-dependencies.

The ALUs are multi-granular, in the sense that they are equipped with vector operations that can process 8, 16, and 32 bit operations. There is no support to form larger operations by combining multiple tiles.

### 2.5 Multi-Granular Arithmetic

In many signal processing applications, not all operations need the same number of bits. Often many of the operations performed are short-width, such as an 8-bit multiplication; and only few operations use the full 32 or 64 bits available on many architectures. So it is a waste of energy to force a uniform bit-width on all operations, as this will often be too wide. Or, if a narrow operation-width is used in an architecture, software support is needed to calculate the additional bits of a wide operation, resulting in extra cycles and computational overhead.

As applications often only need 8-bit or 16-bit operations, energy efficiency and performance could be increased compared to a traditional 32-bit or 64-bit

architecture if it is possible to compute operations on other granularities. Switching entirely to an 8-bit or 16-bit architecture is also not practical, because most applications use some larger bit-width calculations (among others to address the memory). Figure 2.3 indicates that in a 32-bit architecture, often not all available bits are used for the calculations, for example when processing the 8-bit RGB colour values of a pixel.

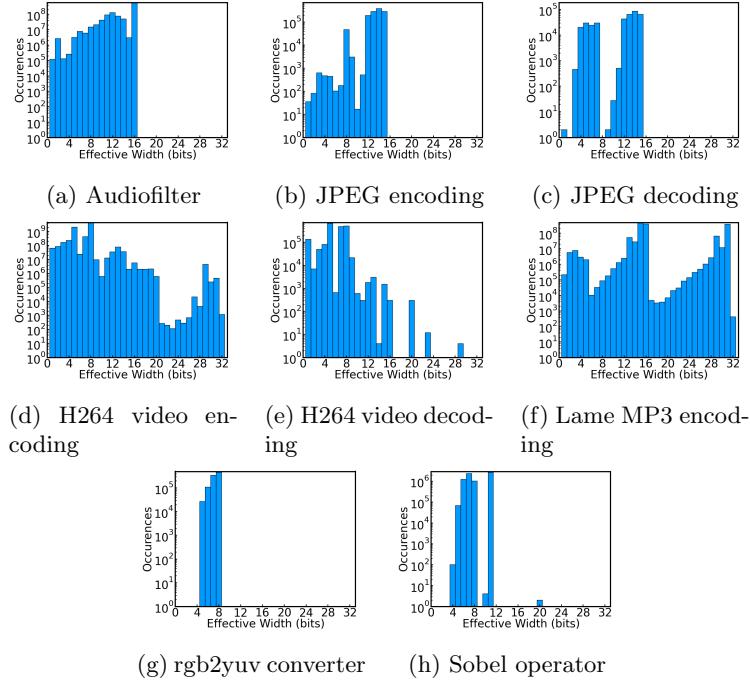


Figure 2.3: These graphs show the effectively used bit-width for the operands of multiplications in several applications, as determined by benchmarking on a 32-bit OpenRISC.

As silicon area is becoming cheaper [25], one solution is to just place multiple arithmetic units on the chip, each with another width, e.g. an 8-bit, 16-bit and 32-bit arithmetic unit [2]. This might save some energy in the actual arithmetic calculation, but the interconnect and register files still have to be as large as the largest supported width. As the actual calculations only use a small part of the total energy budget, this small improvement in the cost of arithmetic operations is easily outweighed by the increased logistic overhead this creates.

For multiplication, twin multipliers [21] are another attempt to find the right granularity for operations. In a twin multiplier, it is possible to compute either one operation that uses the full available width, or two operations in parallel, each half of the available width wide. This is done by disconnecting some of the wires inside the multiplier, effectively separating the upper and the lower part of the multiplier. This gives some power savings, as the unused parts of the multiplier can be disabled, and doubles the potential throughput for operations that are half as wide as the multiplier. However, multipliers are quadratic in size, so even with this improvement, still half of the multiplier is

idle when performing half width multiplications; extending this algorithm to a quad multiplier is not feasible, as the standard ALU interface is unable to supply enough input operands.

We can also approach this bottom-up, starting from small arithmetic units, and combining multiple of these blocks, or *Functional Units* (FU), to form larger operations. This gives the programmer the freedom to choose, for each operation, the width that he needs. We call this concept *Multi-Granular Arithmetic*.

The *BLOCKS* architecture, described in the next section, uses multi-granular arithmetic as a basic principle. In the remaining chapters of this report, we will develop several multi-granular arithmetic algorithms, and investigate the trade-off between computational (energy) efficiency and flexibility as a function of the coarseness of the operation building block size, in particular for add and multiply, and their combinations.

## 2.6 BLOCKS Architecture

The *BLOCKS* [29] architecture is currently being developed in an attempt to create an energy efficient architecture for the signal processing applications described in Section 2.2. *BLOCKS* is a CGRA, that consists of many *Functional Units* (FU), that can perform logic and arithmetic operations. Additionally there are *Instruction Decode* (ID) with *Instruction Fetch* (IF) units that communicate with the instruction memory, *Register Files* (RF) and *Load-Store units* (LS) that can both access a local memory, and the global shared data memory. These blocks are connected by a reconfigurable, mesh type, interconnect network. An abstract overview of this design is given in Figure 2.4.

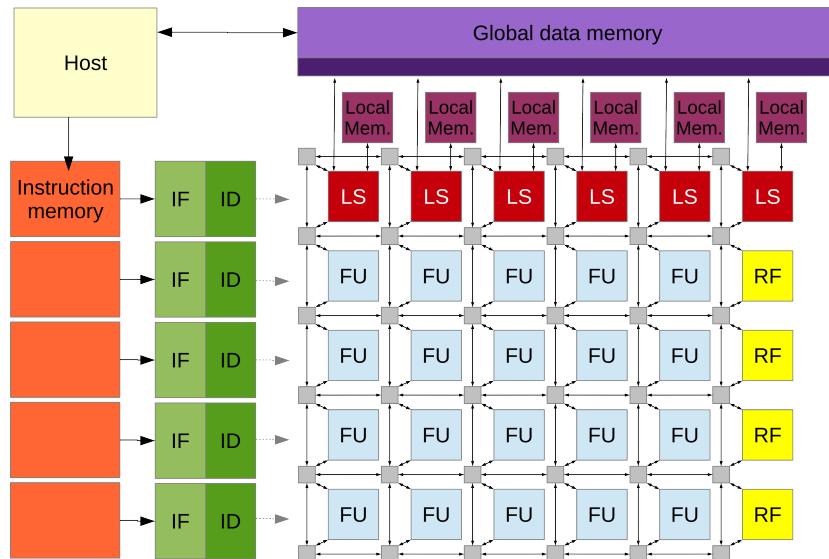


Figure 2.4: An abstract overview of the BLOCKS architecture.

The functional units can be connected in order to perform larger, multi-word operations. The interconnect can be reconfigured to route the result of a functional unit to the input of another functional unit, or to a register file. This allows for the creation of spatial layouts, where multiple functional units are connected to compute more complex operations, bypassing the register files.

Not only the interconnect is reconfigurable; this also holds for the other elements in the architecture, mainly the functional units and instruction decoders. As discussed in the previous section, multiple functional units can be configured to work together in order to perform operations at different granularities; for example, two 8-bit adders can work together to perform an 16-bit addition. The instruction decoders are configurable, such that they are able to decode instructions that are applicable to the configured functional unit layout. This can be SIMD-like, but also VLIW-like or even form other configurations, such as a tree that filters an input array to a single result, as is illustrated in Figure 2.5.

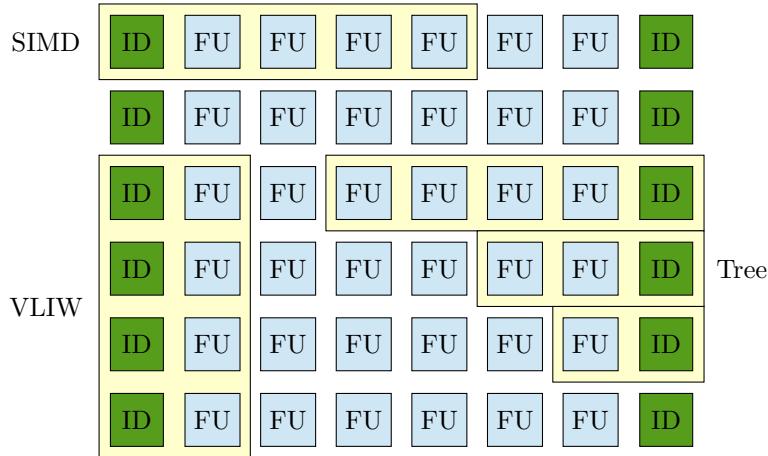


Figure 2.5: SIMD, VLIW and a tree mapped to the BLOCKS functional unit grid.

As functional units can be combined to form larger operations, the base size of a functional unit should not be very large; the operations in a functional unit could be 8 or 16 bit wide. This allows the efficient execution of both large and small operations. For the remainder of this report, we will look at the design — and in particular the multi-granular properties — for the functional units for the BLOCKS architecture.

## Chapter 3

# Multi-Granular Arithmetic

Most modern processor architectures have a fixed-width ALU data-path. This has many advantages: it is easy to implement and use, it has a small area footprint and it is fast. And as long as the implemented width is actually used, it is also energy efficient. However, many applications perform a lot of operations that operate on data that is narrower than the width of the datapath of the architecture, as is illustrated in Figure 2.3. So for many applications, energy is wasted on operations that are too wide.

In this chapter we discuss how arithmetic units can operate on different operation sizes by combining multiple smaller functional units, using a technique called multi-granular arithmetic.

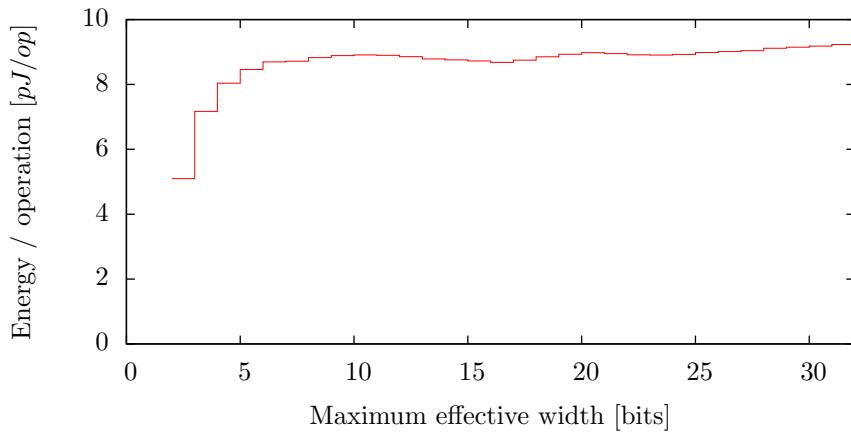
### 3.1 Frugal Arithmetic

For many applications running on general purpose hardware, there is a mismatch between the width of the operations as they are supported by the hardware, and the effectively used bits in the calculations required by the application, as illustrated in Figure 2.3. This is caused by the inflexibility of existing architectures to adapt the width of their operations to the needs of the application, as most architectures only support operations with a fixed width. For example, when an 8-bit calculation is required by the application while executing on a 32-bit general purpose processor, this operation has to be extended to 32 bit, and after calculating the result, only the lower 8 bits of the result are used; the upper 24 bits of the result are simply ignored.

This mismatch leads to wasted energy, as a part of the energy is used to calculate unused bits. If the width of the operations was flexible instead of fixed, we could reduce the amount of energy that is wasted on unnecessary large operations, because it would then be possible to only calculate the bits that are required by the application. We call this concept *frugal arithmetic*.

Several solutions are already available to make the size of the performed arithmetic calculations a better match to the size of the arithmetic operation that are required by the application. Here we discuss several of these solutions.

Most processor architectures support only a single width for their operations; for example, operations such as addition can be performed by a piece of circuitry that performs a 32-bit addition, and not in any other way. Applications can perform narrower operations, such as an addition of 16-bit numbers, by performing a 32-bit additions for which only the least significant 16 bits of the result are used. If an application has to perform larger operations, e.g. a 64-bit operation on a 32-bit architecture, this has to be done in software; multiple smaller operations have to be executed sequentially in order to produce the desired result. Performing wide operations in software in this way comes at a high cost, both in terms of energy efficiency and performance. Operations narrower than the native width of the processor are not any slower than operations using the full width, but they also use nearly the same amount of energy as full width operations, as can be seen in Plot 3.1.



Plot 3.1: Energy used to perform multiplications of increasing width using a 32-bit multiplier.

An alternative architecture could be made by implementing multiple versions of the arithmetic unit on the same chip, where each unit has a different width. Say a chip has 8-, 16-, 32- and 64-bit arithmetic units, then it is able to compute all operations up-to 64-bit with at most 50% unused capacity; the other arithmetic units could be power-gated. Of course this requires extra chip area, but as silicon area is cheap [25], this is a reasonable price to pay. However, the actual arithmetic units only use a fraction of the total energy budget of modern processor architectures [18]; a large part of the energy is used for the registers, memories, and data-paths. With this approach, these parts of the architecture cannot easily be made smaller than the maximum supported width, as they all need to support the largest arithmetic unit, resulting in a very limited reduction in energy usage.

This could be solved if multiple versions of the datapath and register files are added, however this generates even more overhead. Somehow the different arithmetic units should be able to communicate, thus if the interconnect and register files are available in multiple widths, extra mechanisms must be added to facilitate this communication; adding too much overhead to the system.

Själander et al. [21] introduce the concept of twin-multipliers. Twin-multipliers can perform both normal multiplications, or a single multiplication of half the width with reduced energy usage, or two parallel multiplications that are half as wide as the multiplier. They have added some extra logic to disable and disconnect parts of the multiplication circuit, such that these two independent multiplications can take place at the same time, and unused parts can be disabled. Figure 3.1 shows a twin multiplier, where the parts that perform the two half width multiplications are shaded different shades of grey. This design is able to save energy and improve throughput compared to a standard multiplier. However, the throughput is only doubled; half of the circuit is idle when performing half width multiplication. It is also unable to perform multiplications of arbitrary size.

$$\begin{array}{ccccccccccccccccccccc}
 & \\
 y_7 & y_6 & y_5 & y_4 & y_3 & y_2 & y_1 & y_0 & & & & & & & & & & & & \\
 x_7 & x_6 & x_5 & x_4 & x_3 & x_2 & x_1 & x_0 & \times & & & & & & & & & & & & \\
 \hline
 & p_{7,0} & p_{6,0} & p_{5,0} & p_{4,0} & p_{3,0} & p_{2,0} & p_{1,0} & p_{0,0} & & & & & & & & & & & & \\
 & p_{7,1} & p_{6,1} & p_{5,1} & p_{4,1} & p_{3,1} & p_{2,1} & p_{1,1} & p_{0,1} & & & & & & & & & & & & \\
 & p_{7,2} & p_{6,2} & p_{5,2} & p_{4,2} & p_{3,2} & p_{2,2} & p_{1,2} & p_{0,2} & & & & & & & & & & & & \\
 & p_{7,3} & p_{6,3} & p_{5,3} & p_{4,3} & p_{3,3} & p_{2,3} & p_{1,3} & p_{0,3} & & & & & & & & & & & & \\
 & p_{7,4} & p_{6,4} & p_{5,4} & p_{4,4} & p_{3,4} & p_{2,4} & p_{1,4} & p_{0,4} & & & & & & & & & & & & \\
 & p_{7,5} & p_{6,5} & p_{5,5} & p_{4,5} & p_{3,5} & p_{2,5} & p_{1,5} & p_{0,5} & & & & & & & & & & & & \\
 & p_{7,6} & p_{6,6} & p_{5,6} & p_{4,6} & p_{3,6} & p_{2,6} & p_{1,6} & p_{0,6} & & & & & & & & & & & & \\
 & p_{7,7} & p_{6,7} & p_{5,7} & p_{4,7} & p_{3,7} & p_{2,7} & p_{1,7} & p_{0,7} & & & & & & & & & & & & + \\
 \hline
 s_{15} & s_{14} & s_{13} & s_{12} & s_{11} & s_{10} & s_9 & s_8 & s_7 & s_6 & s_5 & s_4 & s_3 & s_2 & s_1 & s_0 & & & & 
 \end{array}$$

Figure 3.1: Partial products of two 4-bit multiplications in an 8-bit twin-multiplier.

None of the currently existing solutions are able to perform operations of arbitrary width in an energy efficient manner. We would like to have this for the BLOCKS architecture, so something new has to be developed.

### 3.2 Multi-Granular Arithmetic

Frugal arithmetic can be implemented in a very flexible manner using an FPGA; any operation can easily be constructed on any desired operation width. Unfortunately, this flexibility comes at a hefty price: implementations of complicated

operations such as addition or multiplication are much less energy efficient when implemented on an FPGA than they would be when implemented as a dedicated circuit. Moreover, we do not need the full flexibility provided by an FPGA to implement frugal arithmetic; our architecture only needs to construct typical ALU operations, such as addition, subtraction, multiplication, comparison, bit-level operations, et cetera.

Many of these ALU operations have the property that wide-word versions of the operation can be implemented as the composition of multiple narrow-word versions of the operation. For example, a 16-bit bit-wise AND operation can be implemented as a concatenation of two 8-bit bit-wise AND operations:

$$\begin{aligned} a \wedge_{16} b &= ((a_1 \ll 8) \oplus a_0) \wedge_{16} ((b_1 \ll 8) \oplus b_0) \\ &= ((a_1 \wedge_8 b_1) \ll 8) \oplus (a_0 \wedge_8 b_0) \end{aligned}$$

Using this structure, many ALU operations can be implemented in arbitrary sizes, assuming sufficient hardware implementations of small versions of these operations are available. We call this concept *multi-granular arithmetic*.

With this in mind, we can design an architecture that supports frugal arithmetic by applying multi-granular arithmetic to implement wide-word operations. For each ALU operation, this architecture would contain several circuits implementing a small-width version of that operation; large-width operations would be constructed as a multi-granular composition, using a reconfigurable datapath to implement the operation at the exact width required.

This architecture implements frugal arithmetic in a reasonably efficient way. Operations can be performed in any width that is a multiple of the size of the natively implemented operation blocks. As a consequence, each operation is performed at a width at most  $w - 1$  bits more than the application demands, where  $w$  is the width of the native operation blocks. The benefit of this approach is that energy wasted on computing unnecessary bits is likewise limited to a number proportional to  $w$ .

Of course, implementing large operations as a multi-granular composition does come at a cost. In general, a multi-granular composition of smaller blocks makes a circuit that is not as efficient as a circuit optimised for the exact overall width; as a consequence, implementing large operations in this way incurs an energy efficiency penalty. The size of this penalty depends on the width of the native blocks, as larger compositions generally have a larger overhead.

In this architecture, the width of the operation blocks can be chosen freely; however, this choice has consequences for energy efficiency. There is a trade-off between the efficiency gain when performing small operations on one hand, and the penalty when performing large operations on the other hand. Ideally, this block size is chosen such that the application does not perform many operations substantially smaller than the chosen block size. In practice, 8-bit or 16-bit blocks will probably be efficient choices for many applications.

### 3.3 Composing Operations

In order to compose large operations from smaller computational blocks, several of these blocks have to be combined to form a larger, multi-granular, operation. For some operations it is trivial to support multi-granular operations, while other operations require more thought or specialised algorithms.

To do this, we decomposed numbers wider than the block size into word-sized blocks. For a two-word wide number  $a$ , that uses  $2w$ -bits, we denote this as  $a_1 \ll w \oplus a_0$ . In this notation,  $a_0$  contains the lower word, while  $a_1$  contains the upper word. The bit-shift operation  $\ll$  is used to indicate the significance of each word, while the  $\oplus$  symbol is used to indicate that these words together form a larger number. As this bit-shifting only occurs with a multiple of the block size, the interconnect is able to perform this shifting, by routing the result to another computational block. Similarly, the  $\oplus$  operation is only used to concatenate disjoint, word-aligned, partial numbers, which means that the larger number can be constructed by concatenating the words together.

To illustrate the decomposition of the operations discussed below, we will use two 16-bits inputs  $a = a_1 \ll 8 \oplus a_0$  and  $b = b_1 \ll 8 \oplus b_0$ , thus  $a_0$  and  $b_0$  contains the lower 8 bits, and  $a_1$  and  $b_1$  the upper 8 bits. The block-size used in these examples is 8-bit.

Trivial are the bit-level logic operations: AND, OR, NOT, NAND, NOR, XOR, etc. These operations are easy because all calculations are bit-wise, with no dependencies on the other bits in the input. These operations can be executed on several, unmodified, parallel computational blocks and the result can be obtained by concatenating the results. Both Equation 3.1, where  $\bullet_x$  is used as a symbol for an arbitrary logic operation with a width of  $x$  bits, and Figure 3.2 illustrate this.

$$\begin{aligned} a \bullet_{2w} b &= ((a_1 \ll w) \oplus a_0) \bullet_{2w} ((b_1 \ll w) \oplus b_0) \\ &= ([a_1 \bullet_w b_1] \ll w) \oplus [a_0 \bullet_w b_0] \end{aligned} \quad (3.1)$$

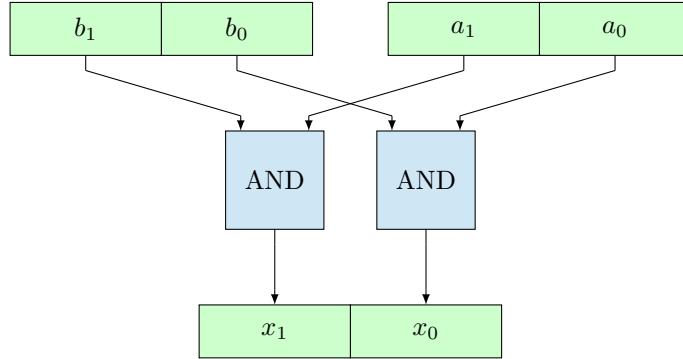


Figure 3.2: Two 8-bit computational blocks performing an AND operation on two 16-bit inputs.

Bitshift operations are also not difficult; they can even benefit from the multi-granular approach, as bitshifting by a multiple of the block size can be handled for free by the interconnect network. If the amount of bits to be shifted is not a multiple of the word-size, then each computational block has to shift a block individually, as shown in Equation 3.2; the only difference with a dedicated shift operation is that each block of input data has to be distributed to two computational blocks, as is illustrated in Figure 3.3.

$$\begin{aligned} a \ll n &= ((a_1 \ll w) \oplus a_0) \ll n \\ &= [(a_1 \ll n + a_0 \gg (w - n)) \bmod 2^w] \ll w \oplus [(a_0 \ll n) \bmod 2^w] \end{aligned} \quad (3.2)$$

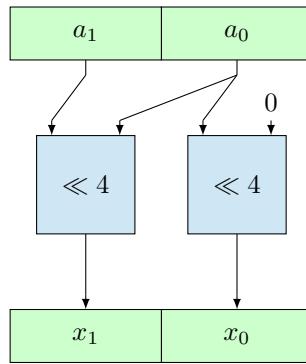


Figure 3.3: Two 8-bit computational blocks performing a left-bit-shift of 4 bits on a 16-bit input.

Add and subtract operations are not as easy, as these operations have dependencies between the individual bits in the form of carry bits. Equation 3.3 shows how a  $2w$ -bit wide addition can be composed into two  $w$ -bit wide additions. The term  $(a_0 +_w b_0 +_1 c_i) \text{ div } 2^w$  that is added to the most significant adder is also known as the carry-out produced by the least significant adder. Thus, in order to combine multiple computational blocks to perform larger additions, a carry chain is needed between these blocks, as illustrated in Figure 3.4.

$$\begin{aligned} a +_{2w} b +_1 c_i &= ((a_1 \ll 2) \oplus a_0) +_{2w} ((b_1 \ll w) \oplus b_0) +_1 c_i \\ &= [a_1 +_w b_1 +_1 ((a_0 +_w b_0 +_1 c_i) \text{ div } 2^w)] \ll w \oplus [(a_0 +_w b_0 +_1 c_i) \bmod 2^w] \end{aligned} \quad (3.3)$$

Multiplication scales quadratically with respect to the number of input bits, so dividing a larger operation over multiple computational block will require a quadratic amount of these blocks. Equation 3.4 shows how the multiplication  $a \times b$ , with both  $a$  and  $b$  of width  $2w$  can be composed of operations of size  $w$ . The mapping to of these operations to multi-granular computational blocks is shown in Figure 3.5.

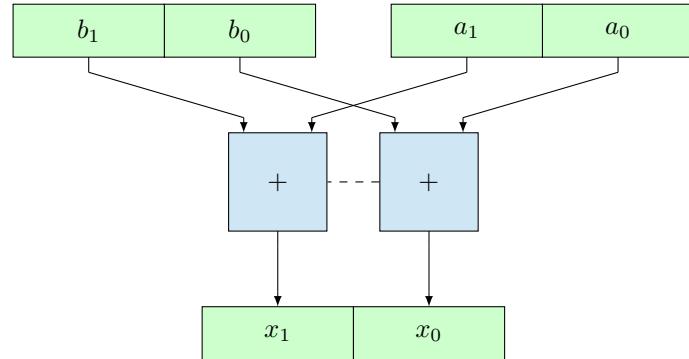


Figure 3.4: Two 8-bit computational blocks performing addition on two 16-bit inputs. The dotted line connects the carry-out of the right block to the carry-in of the left block.

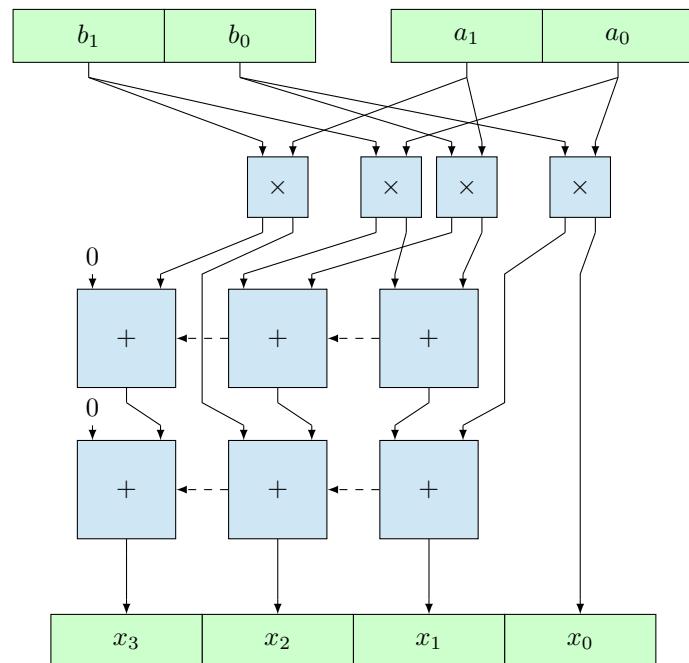


Figure 3.5: Ten 8-bit computational blocks together performing full-width (e.g. 32-bit output) multiplication on two 16-bit inputs. The first four blocks perform multiplication operations, and the other six blocks, configured as two 24-bit adders, add the results from the multipliers.

$$a \times_{2w} b = (a_0 \times_w b_0) + (a_1 \times_w b_0) \ll w + (a_0 \times_w b_1) \ll w + (a_1 \times_w b_1) \ll 2w \quad (3.4)$$

For both addition and multiplication there exists an accumulation variant, where the results of subsequent operations are added together. Equation 3.5 shows two equations that add-accumulate and multiply-accumulate can solve.

$$\begin{aligned} r_{add} &= a + b + c + d + e + f + \dots \\ r_{multiply} &= (a \times b) + (c \times d) + (e \times f) + \dots \end{aligned} \quad (3.5)$$

### 3.4 Multi-Granular Arithmetic in BLOCKS

In order to find a suitable design for multi-granular arithmetic in the BLOCKS architecture, we need to determine the constraints and requirements which are imposed by the BLOCKS architecture. The main focus of the BLOCKS architecture is on energy efficiency and flexibility, and we will use these optimisation criteria as a starting point for the constraints and requirements for the arithmetic algorithms.

To minimise routing and compiler complexity, and to maximise flexibility, the functional units in the BLOCKS architecture are homogeneous; that is to say, all functional units implement the same behaviour.

In order to combine several of these functional units into a larger operation, the BLOCKS architecture contains an interconnect network, connecting the different functional units. Because the details of the exact designs of the BLOCKS interconnect are not yet known, we analyse optimal designs for various interconnect configurations.

Within the scope of this report, we will assume that it is at least possible to use a carry-chain to pass a carry-bit to an adjacent functional unit; we will also assume that the interconnect network allows us to connect the output of a functional unit to the input of another functional unit. However, the interconnect network will also dissipate some energy, and thus a decomposition algorithm that uses the interconnect less frequently has an advantage above another algorithm that uses the interconnect more often.

In this report, we develop methods for multi-granular addition in Chapter 5 and multi-granular accumulation in Chapter 6. In Chapter 7 multi-granular multiplication is discussed, and finally, in Chapter 8, we cover multi-granular multiply-accumulation.

## Chapter 4

# Experimental Methodology

In the following four chapters, we will elaborate on the multi-granular operations discussed in Section 3.3. We want to compare different designs and investigate for each design the trade-off between computational energy efficiency and flexibility as a function of the coarseness of the operation building block size.

In order to compare and quantify energy usage and area usage of different hardware designs, a model of the proposed functional units is needed. We modelled these functional units using the Verilog hardware description language and benchmarked these models for different frequencies to gain the energy and area numbers. In this chapter, we will describe how the Verilog is structured, which tools are used, and how the results can be interpreted.

### 4.1 Verilog Structure

In order to compare different designs, we need energy and area measurements for the different implementations of the functional units. To provide these numbers, the functional units have to be modelled and benchmarked.

In order to perform a certain operation, one or more functional units are needed. We want to measure the energy that is needed to perform this operation; however, we are only interested in the energy used by the functional units itself — not the energy dissipated by the interconnect, instruction fetching and distribution or memory accesses. The same hold for the area usage; we want to measure only the area used for the functional units.

It is also important that the individual functional units are not merged together into a single large operation by the synthesis tool. To ensure this does not happen, all control signals and interconnect routing is handled by the test-bench, which is not synthesised together with the functional units. Without this precaution, the synthesizer might decide to optimise away large parts of some functional units in cases where it can predict parts of the input. Only the

carry wires are routed directly between the functional units, as this signal is important for the timing constraints. The delay of the interconnect routing can be ignored in this simulation and added to the delay of the functional units (or the interconnect is added as an extra pipeline stage).

The functional units are modelled using the Verilog HDL. As is illustrated in Figure 4.1, the Verilog code consists of a “main” module and a “testbench” module. The main module contains the functional units that are needed for the particular simulation, and the inputs and outputs of the functional units are routed to the testbench. The testbench takes the role of both instruction fetching, interconnect network and memory. So it sends opcodes to the functional units, it provides the input data and takes care of the interconnect routing such that result from one functional unit is routed to the input of another functional unit. After the calculations are completed, the testbench also verifies the results.

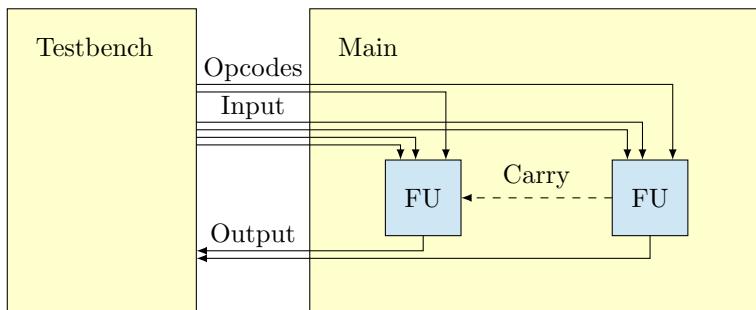


Figure 4.1: Model to test the functional units with a testbench.

The functional units decode the opcode, and instruct the right modules — adder, multiplier, etc. — to do their work, and connect the result from the right module to the output register of the functional unit. This is illustrated in Figure 4.2. The exact modules that are present in each variation of the functional unit varies.

The functional units are not internally pipelined; there is only an output register. However, for operations whose output is larger than the available output ports on the functional unit, the unit may take multiple cycles to fully transmit its result. Pipelining the designs is certainly possible, but this is left as future work.

## 4.2 Tools

The Verilog implementation of the functional units with the accompanying main modules are compiled and synthesised with the Cadence RTL/RC Compiler [4], using a low power library from TSMC’s 40nm CMOS technology [26].

The toolflow used to benchmark our designs is visualised in Figure 4.3. The Cadence RTL/RC Compiler is first used to compile and synthesise the Verilog code containing the functional units and main module for a specified clock frequency; no place and route is done. Then the synthesised code is simulated with the

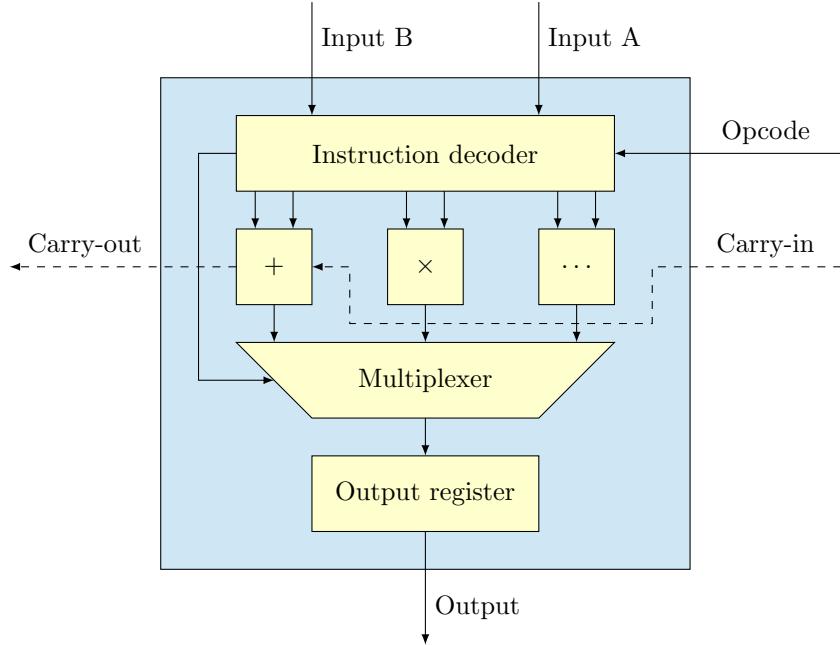


Figure 4.2: A simplified model of a functional unit as implemented in Verilog.

provided testbench to both verify correctness of the design and generate usage patterns. These usage patterns are used to obtain a good estimate on the power usage. Finally, the tools generate reports on power and area usage.

### 4.3 Results

The testbench simulation will provide us with power and area numbers, and we also know the operating frequency and interconnect usage. This is used, together with the number of operations in the testbench, to normalise the power numbers to energy per operation.

These numbers can be plotted, so it is possible to compare several configurations. An example is shown in Plots 4.2 and 4.3. These pages each contain three graphs, where each graph gives a different view on the same configurations. The first graph shows the energy per operation, expressed in  $pJ/op$ , versus the operation frequency, and each line represents a configuration. In this case, the configurations are a native 32-bit adder that serves as a baseline, and three adders each constructed from combining smaller functional units. The second plot also shows energy usage, but now relative to the native baseline. The last plot contains area, in  $\mu m^2$ , versus frequency.

For each design, we will provide several plots. In the first plot, we will depict the behavior of the design implementing a 32-bit operation, constructed using

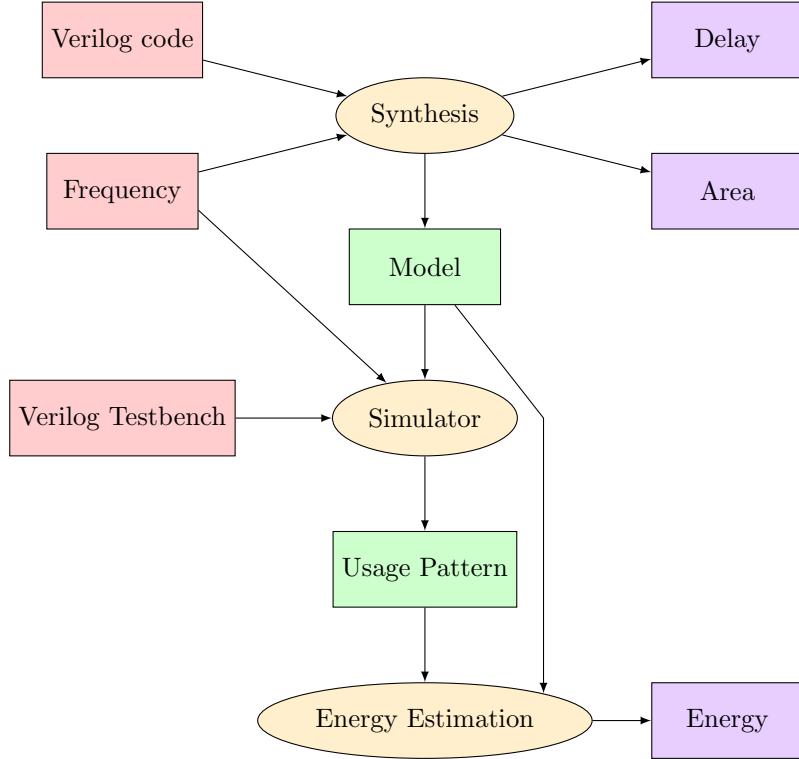


Figure 4.3: The toolflow used to benchmark our designs.

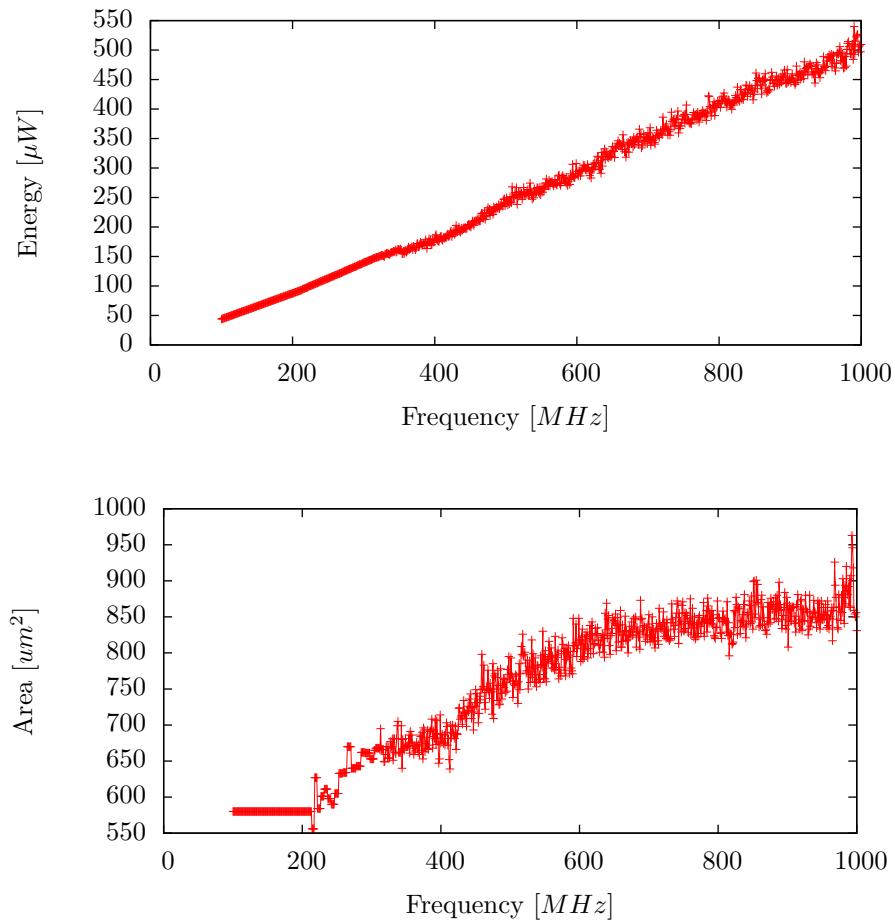
8-bit, 16-bit, and 32-bit functional units. In the second plot, we will measure the behavior of the same three granularities when used to implement an 8-bit operation. In all plots, we will also provide a “native” implementation to serve as a baseline, which is implemented by whatever the Verilog library contains as the standard algorithm for the relevant operation; for example, the native baseline for addition is the Verilog “+” operator. Of these plots, the first illustrates the performance penalty incurred by implementing a wide operation as a composition of several smaller functional units, as opposed to using an optimal width-specific implementation. Conversely, the second plot depicts the performance gained in those cases where a narrow operation is implemented using a narrow functional unit, as opposed to using the unnecessarily large implementation.

At the end of each chapter, we will also compare the various designs with each other (and the baseline). We compare the results for both 32-bit operations as well as 8-bit operations, in order to determine the costs of composing wide operations from smaller functional units, as well as the gain, compared to the native 32-bit baseline, when smaller functional units are used.

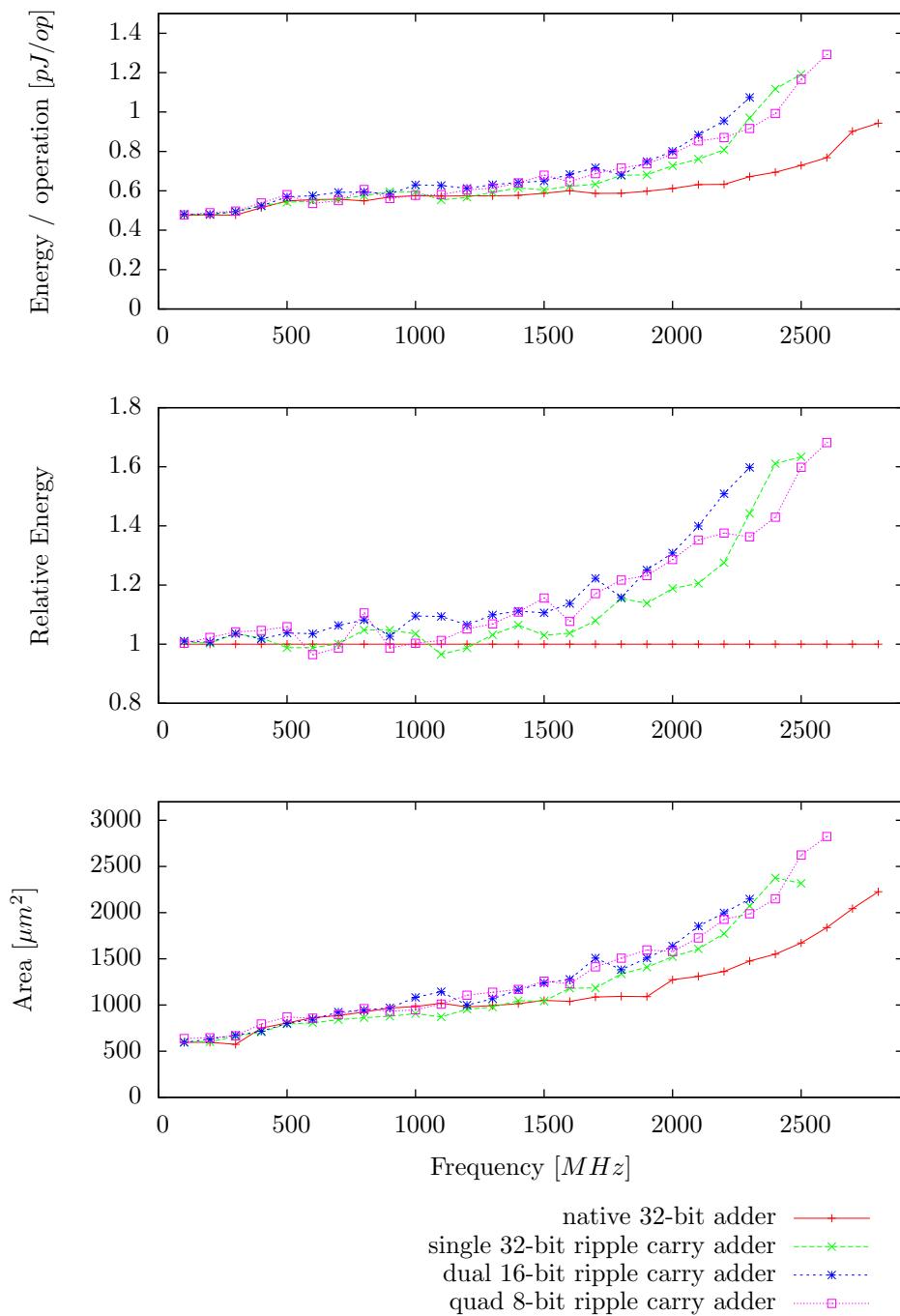
Unfortunately, the synthesis tool is unable to create the optimal design, as this task is NP-hard. Instead, the synthesis tool uses deterministic heuristics to produce a good design in a reasonable amount of time. However, this can lead to benchmark results that can substantially differ between slightly different

designs, as illustrated in Plot 4.1, which shows the energy and area numbers for a simple ripple-carry adder which was benchmarked from 100 to 1000 MHz, in 1 MHz increments. Where an ideal synthesis tool would produce a smooth line, this is clearly not the case here.

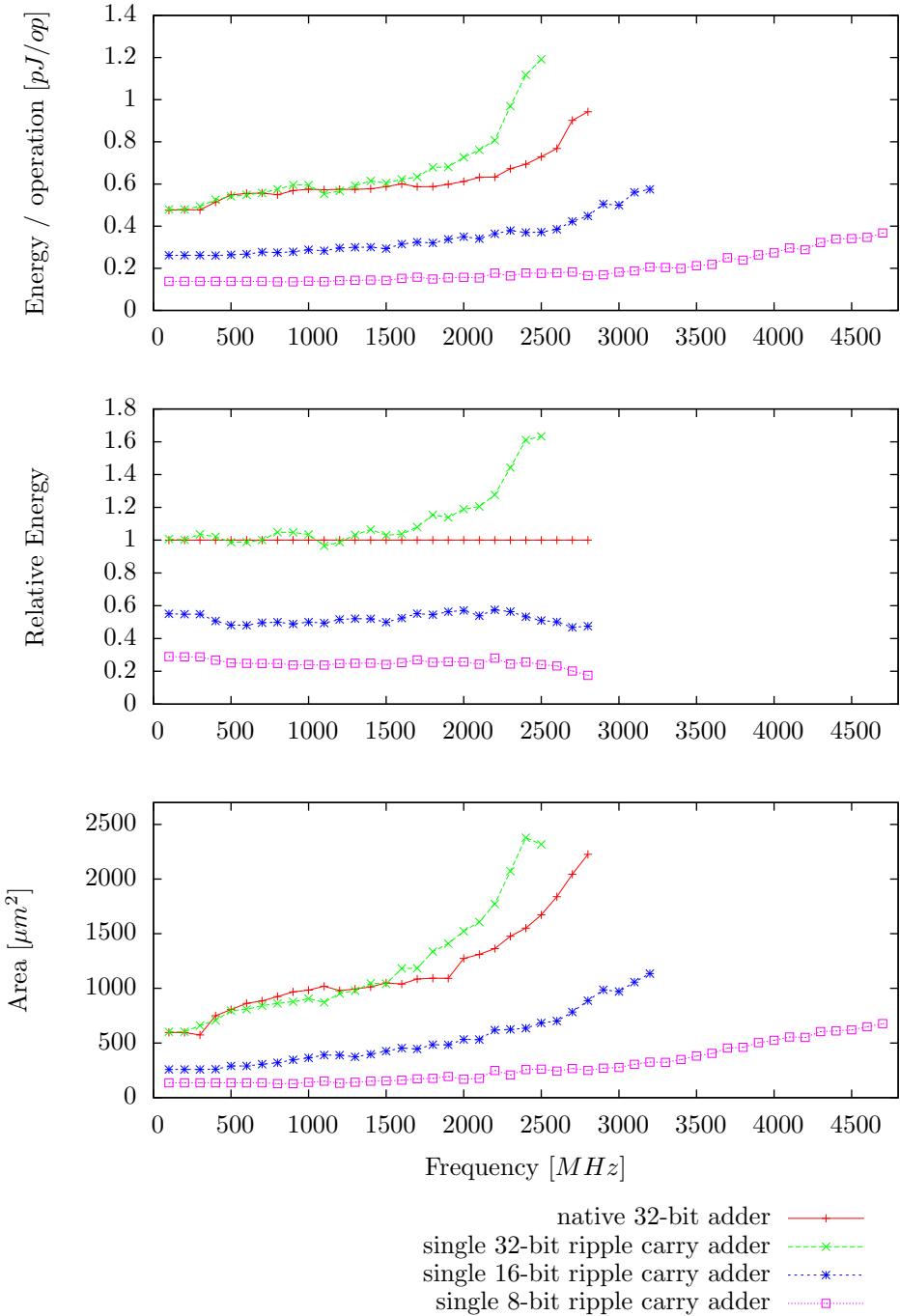
In addition to these plots, we will also analyse, where applicable, the delay and area in a more mathematical manner, using the  $\mathcal{O}$ -notation [12]. This notation shows how fast the delay or area will grow when certain parameters are changed, such as the width of a functional unit, denoted as  $w$ , or the width of the operation, which is expressed as a multiple  $n$  of the functional unit width. For example, a concatenation of  $n$  separate  $w$ -bit ripple-carry adders implements a  $(w \times n)$ -bit ripple carry adder, with a delay of  $\mathcal{O}(w \times n)$ .



Plot 4.1: Unpredictability in benchmark results due to synthesis heuristics.



Plot 4.2: Example plot measuring energy efficiency and area usage of four different 32-bit adder implementations.



Plot 4.3: Example plot measuring energy efficiency and area usage of four different 8-bit adder implementations. The relative energy graph is truncated at the highest frequency achieved by the baseline implementation.

# Chapter 5

## Addition

Addition is the operation that adds two numbers, the inputs  $a$  and  $b$ , to form the sum  $s = a + b$ . In our implementations, both inputs have to be the same width, which has to be a multiple of the block size used by the architecture. The width of the output  $s$  is the same as the width of both inputs.

### 5.1 Addition Algorithms

In digital logic, natural number are typically encoded as a sequence of bits representing a binary encoding of said number. The *sum* of two  $n$ -bit sequences  $a$  and  $b$  representing the natural numbers  $A$  and  $B$ , then, is the  $n$ -bit sequence  $s$  representing the natural number  $A + B$ .

Unlike operations such as bitwise AND, the  $i$ -th bit of the sum  $A + B$  does not only depend on the  $i$ -th bit of inputs  $a$  and  $b$ ; its value is also affected by the less-significant bits of  $a$  and  $b$ . If  $A_{0..i}$  and  $B_{0..i}$  are the natural-number interpretations of the least  $i$  significant bits of  $a$  and  $b$ , the  $i$ -th bit of the sum  $s$  is equal to  $(a_i + b_i + 1) \bmod 2$  if  $A_{0..i} + B_{0..i} \geq 2^i$ , and equal to  $(a_i + b_i) \bmod 2$  otherwise. The bit describing whether  $A_{0..i} + B_{0..i} \geq 2^i$  is called the *i-th carry bit*, denoted  $c_i$ .

The structure of carry bits makes addition a more complicated operation than operations such as bitwise AND. Naive implementations of the computation of the  $i$ -th carry bit result in a delay proportional to  $i$ , with all the performance consequences this implies. For this reason, many different addition algorithms exist, that vary primarily in the way these carry bits are calculated.

The basic building block for all commonly used adders is the *half adder*, which takes two input bits ( $a$  and  $b$ ), and produces a sum bit ( $s$ ) and a carry bit ( $c_{out}$ ), where  $a + b = s + 2 \times c_{out}$ . Two half adders and an or-gate can be combined to form a *full adder*, depicted in Figure 5.1. A full adder takes three input bits ( $a, b, c_{in}$ ) and produces a sum bit ( $s$ ) and a carry bit ( $c_{out}$ ), where

$a + b + c_{in} = s + 2 \times c_{out}$ . Based on this structure, a full adder can be used to compute the bits  $s_i$  and  $c_{i+1}$  based on input bits  $a_i$  and  $b_i$  as well as the previous carry bit  $c_i$ .

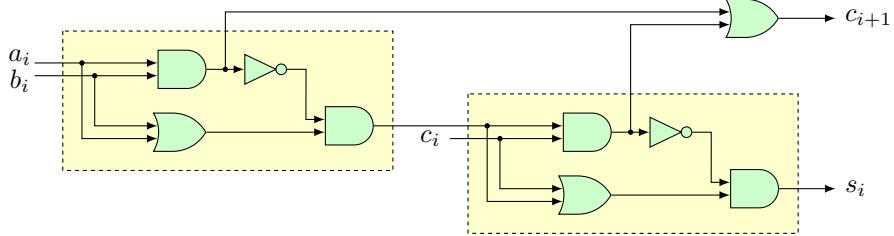


Figure 5.1: Implementation of a full adder based on two half adders.

There are many algorithms available to make  $n$ -bit adders with full adders as a basic building block. Parhami [20] and Koren [15] give a comprehensive overview of the algorithms that have been developed. We will give a quick introduction to the algorithms that are used in this report.

### 5.1.1 Ripple-Carry Adder

The most basic adder algorithm is the *Ripple-Carry Adder* (RCA). This algorithm simply consists of a chain of full adders through which the carry ripples. This results in the naive way of computing  $c_i$  described above.

By connecting the carry-out of the first full adder to the carry-in of the second full adder, and the carry-out of the second full adder to the carry-in of the next full adder, and so on, a larger adder can be constructed, as is shown in Figure 5.2.

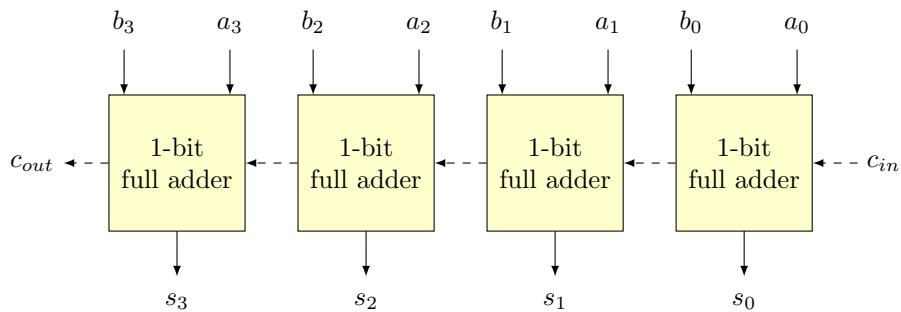


Figure 5.2: Schematic design of a ripple-carry adder.

Due to its simplicity, the ripple-carry adder has a very low gate count, which scales with  $\mathcal{O}(n)$  for an  $n$ -bit adder, and thus has a low area usage. The downside of the ripple-carry adder is that one of the inputs for each full adder consists of the output of the previous full adder. Because the carry has to “ripple” through all of its full adders this way, the delay of the ripple-carry adder does not scale

well: an  $n$ -bit adder incurs a delay of  $\mathcal{O}(n)$ . Because the gate count is low, this adder is energy efficient at low frequencies, where the higher delay is no issue.

### 5.1.2 Carry-Lookahead Adder

The *Carry-Lookahead Adder* (CLA) tries to have the carry-information available as soon as possible. It does this by modifying the full adder to compute *generate*  $g_i$  and *propagate*  $p_i$  bits for each pair of input bits  $a_i$  and  $b_i$ , as is illustrated in Figure 5.3. These bits indicate whether the addition will generate a carry bit, or if it will propagate the carry bit if it receives a high carry-in. Then a *Carry-Lookahead Generator* (CLG) can be used to generate the carry bits, based on the generate and propagate bits, as is illustrated in Figure 5.4.

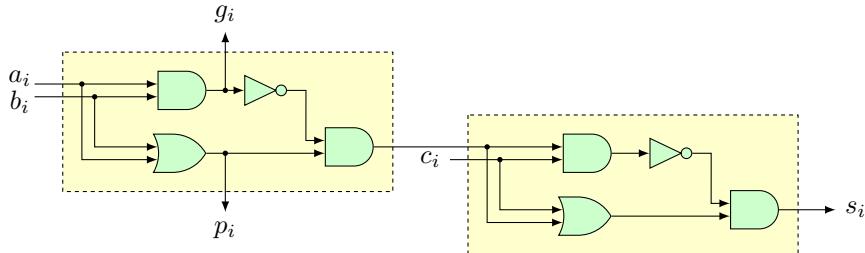


Figure 5.3: Reduced full adder implementation, which outputs generate  $g_i$  and propagate  $p_i$  bits.

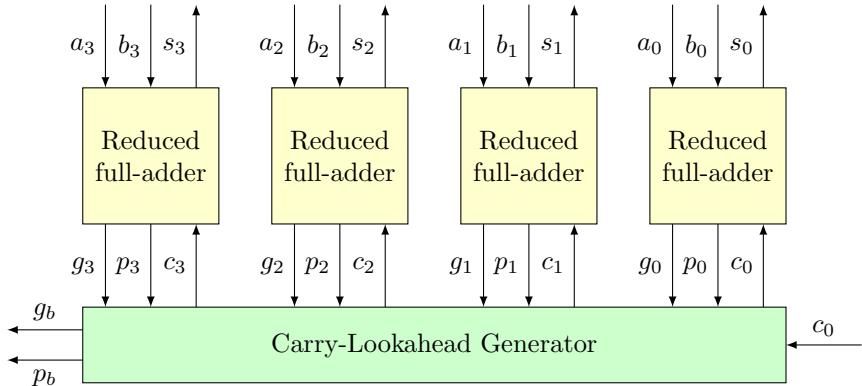


Figure 5.4: A schematic representation of a Carry-Lookahead Generator or CLG with four full adders.

The CLG calculates the carry bits based on the recursive Formula 5.1. As this formula rapidly becomes more complex when expanded, as shown in Formula 5.2, it is not common to use this for more than four bits. Instead, the CLGs are nested. Each CLG produces its own generate and propagate bits,  $g_{out}$  and  $p_{out}$ . Another, higher level, CLG can then produce the carry bits for the lower level CLGs, as is illustrated in Figure 5.5. If desired, the generate and propagate bits of the top-level CLG can be converted to a “normal” carry-out bit, as defined in Formula 5.3.

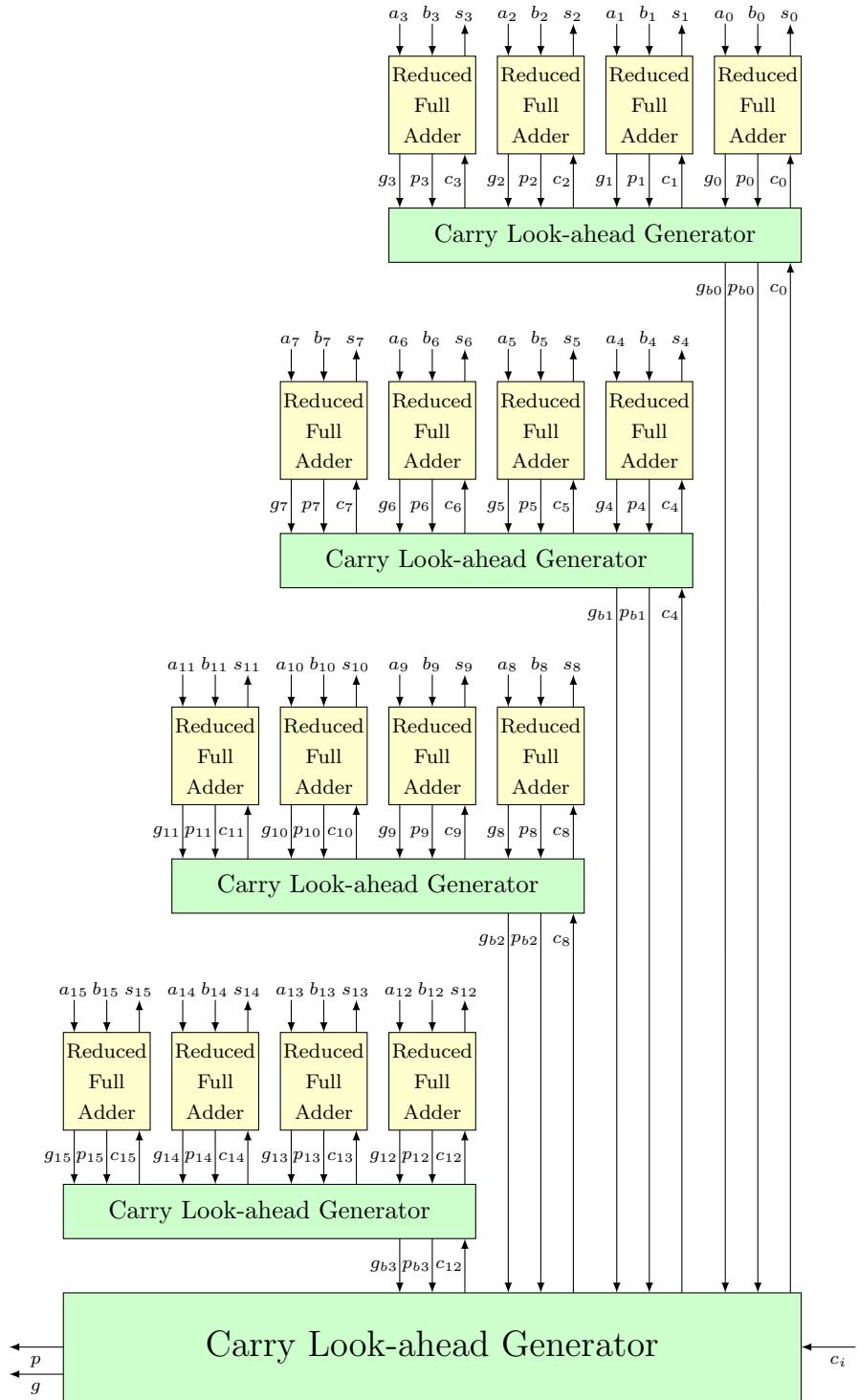


Figure 5.5: A schematic representation of a 16-bit Carry-Lookahead Adder (CLA).

$$c_{i+1} = g_i + (p_i \cdot c_i) \quad (5.1)$$

$$\begin{aligned} c_1 &= g_0 + p_0 \cdot c_0 \\ c_2 &= g_1 + p_1 \cdot g_0 + p_1 \cdot p_0 \cdot c_0 \\ c_3 &= g_2 + p_2 \cdot g_1 + p_2 \cdot p_1 \cdot g_0 + p_2 \cdot p_1 \cdot p_0 \cdot c_0 \\ g_{out} &= g_3 + p_3 \cdot g_2 + p_3 \cdot p_2 \cdot g_1 + p_3 \cdot p_2 \cdot p_1 \cdot g_0 \\ p_{out} &= p_3 \cdot p_2 \cdot p_1 \cdot p_0 \end{aligned} \quad (5.2)$$

$$c_{out} = g_{out} + p_{out} \cdot c_{in} \quad (5.3)$$

The design of this adder has a better delay than the ripple-carry adder, at the expense of more gates. The delay grows logarithmic;  $\mathcal{O}(\log n)$  for an  $n$ -bit adder. The required area grows with  $\mathcal{O}(n \log n)$ . For low speed adders, where the reduced delay makes no difference, the extra gates will increase energy usage compared to a ripple-carry adder. While at higher frequencies, due to the lower delay, smaller, more energy efficient gates can be used which reduces the energy usage compared to a ripple-carry adder.

### 5.1.3 Carry-Select Adder

The *Carry-Select Adder* (CSA) is an attempt to speed up the calculations by not waiting for the carry to ripple through the full adders, but instead divide the work in blocks and calculate each block twice: once while assuming the carry-in for that block will be 0, and the second time assuming the carry-in will be 1. Once the input carry arrives, a multiplexer selects the right result (both carry-out and sum). This is illustrated in Figure 5.6.

This algorithm has to be used in conjunction with another algorithm; it is a method to speed up the other algorithm, which we will call the base algorithm. This base algorithm is used to create the blocks for this algorithm. It can be used together with the ripple-carry adder and the carry-lookahead adder. The expected gain when used with a ripple-carry adder is larger as this algorithm has a worse base delay than the carry-lookahead adder algorithm.

The size of the adder blocks can be chosen freely, either uniform among all blocks, or varying in size. When all blocks are uniform, a good block size for an  $n$ -bit adder (assuming equal delay for a single addition bit and a multiplexer) is  $\lfloor \sqrt{n} \rfloor$ . When variable-sized blocks are used, the ideal size of each block is such that the delay of that block is the same as the delay that the carry-in needs to reach that block.

The required number of gates for this algorithm depends on the base algorithm that is used, but in general it uses roughly twice as many gates as the base algorithm. The delay for an  $n$ -bit carry-select adder with uniform sized blocks

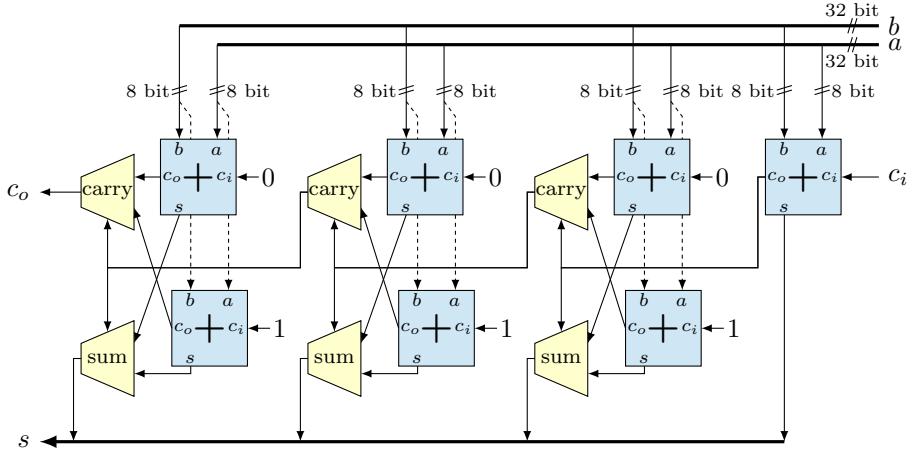


Figure 5.6: Schematic design of a quad 8-bit carry-select adder, using seven adder blocks.

is  $\mathcal{O}(A_{\frac{n}{w}} + w)$ , where  $A_x$  is the delay of an  $x$ -bit base adder, and  $w$  is the block size. The delay can be improved when blocks of increasing size are used [23].

## 5.2 Multi-Granular Addition

In order to construct a multi-granular addition operation, the calculation has to be divided among several functional units, depending on the width of the operation and the width of the functional units. An  $(n \times w)$ -bit addition can be decomposed into  $n$  mostly-independent  $w$ -bit additions; the only communication that needs to happen between these additions consists of the carry information. If each functional unit is equipped with a carry-in and carry-out port in addition to the regular two inputs and the output, then it is possible to chain several functional units by connecting the carry ports to perform larger additions. Figure 5.7 shows how four 8-bit functional units can be connected to form a large adder, but by disconnecting the carry chain, the functional units are also able to perform individual calculations.

The carry-in and carry-out interface is the simplest way to connect the different functional units. It is compatible with all algorithms that we have discussed, and it requires very little support from the interconnect as only local, one bit wide, connections are required. Other carry-interconnect techniques are also possible; for example, a multi-granular version of the carry-lookahead algorithm would require a more extensive flow of carry information between functional units. However, we will only consider the carry-in/carry-out interface in this paper; other carry-interconnect mechanisms are left as future work.

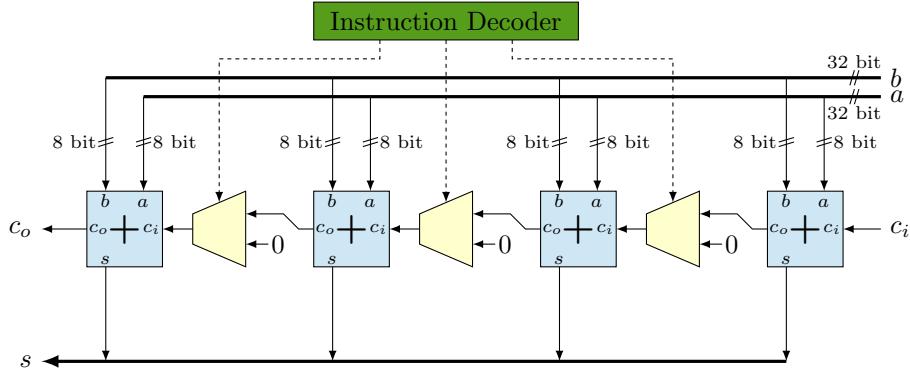


Figure 5.7: Schematic design of a quad functional unit adder connected with an RCA.

### 5.2.1 Base Algorithms

As a base algorithm, placed inside the functional units, several algorithms are possible. We will examine both the ripple-carry adder and carry-lookahead adder algorithms.

The ripple-carry adder is small and simple, but slow. This is expected to be efficient at low clock speeds, as the slowness of the adder is no issue there. At higher clock speeds, the high delay becomes problematic and will drive the energy and area requirements up.

The carry-lookahead adder is faster, but the CLGs required for this speed improvement will also use energy and area, while this does not pay off at lower frequencies. At higher frequencies, the increased speed pays off as this reduces the area and energy growth needed to accommodate the higher clock frequencies.

The carry-select adder is not a suitable algorithm inside a functional unit. This algorithm operates on several (larger) blocks, and inside a relative small functional unit are not enough bits available to effectively form multiple blocks.

### 5.2.2 Composition Algorithms

For the carry propagation between the functional units, we will consider the ripple-carry adder and carry-select adder. The ripple-carry adder is simple and requires little support from the interconnect. It is enough to connect the carry-out of a functional unit to the carry-in of one of the adjacent functional units, and this can be done using a simple, one-bit wide connection between adjacent functional units.

The carry-select adder is a block based algorithm, as discussed before. We will use a functional unit as a block, which means that the entire carry-select algorithm, i.e. selecting the right results based on the carry from the previous block, takes place in the interconnect. Two blocks calculating the same part with a different carry-in are thus mapped to two functional units, where afterwards the interconnect selects the results from the functional unit that had the correct carry-in. The advantage is that the functional units can work in parallel to increase the maximum clock frequency, at the cost of increased energy and area usage.

The carry-lookahead adder is not a suitable algorithm to connect different functional units. Where the interconnect support for the carry-select adder is limited to routing the signals, supporting carry-lookahead adder would require extensive support from the interconnect: the CLGs have to be inlined into the interconnect somewhere. This is impractical as it limits the maximum width to a predetermined maximum size that is supported by these CLGs. It also makes the structure of the functional units and the interconnect less heterogeneous which complicates both synthesis as well as compiler support.

### 5.3 Multi-Granular Adder Configurations

The combination of two base algorithms with two compositions gives us a total of four different designs for multi-granular addition. In this section, we analyze each of these designs in detail, and compare them to a baseline; as a baseline, we used a native, fixed 32-bit adder that is implemented using the Verilog “+” operator, which produces an adder that is optimised for the given operating frequency.

For both the ripple-carry composition and the carry-select composition, the composition of a single functional unit consists of just that one functional unit. As a consequence, these configurations produce identical system designs, and thus produce the same benchmark results, depicted in Plots 5.2 and 5.4.

### 5.3.1 Ripple-Carry Composition, Ripple-Carry Base

The adder that consists of a ripple-carry compositional chain of functional units each implementing a ripple-carry adder is called the *Ripple-carry composition with Ripple-carry base Adder* or *RRA*.

In this design, each  $w$ -bit functional unit has a chain of  $w$  full adders, where the carry-in of the functional unit is connected to the carry-in of the first full adder, and the carry-out of the last full adder produces the carry-out of the functional unit. The different functional units are also connected using a ripple-carry chain. This means that the composition as a whole simply functions as a large ripple-carry adder.

This configuration has the lowest number of gates among all analyzed designs, at the expense of a high delay because the critical path of this algorithm goes through all the full adders. As such, at low clock speeds it performs quite well as delay is not really an issue there. However, it does not scale very well. When higher clock speeds are required or the number of bits become too large, the gates must be scaled in order to meet the timing requirements, thus increasing energy usage.

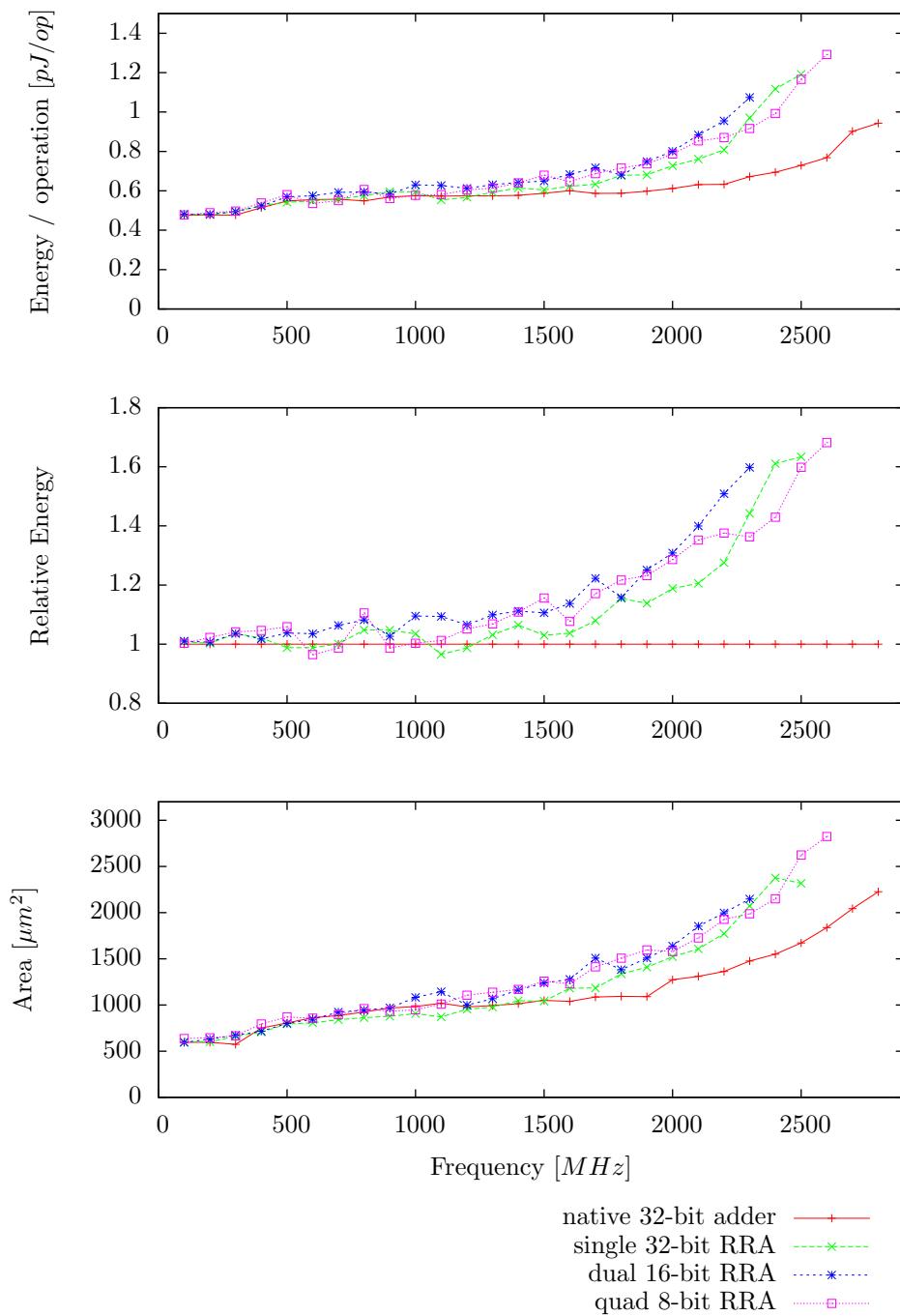
The delay of this design is linear, both in the width  $w$  of a single functional unit, as in the number of words  $n$  used to perform a single larger operation. That gives a total delay of  $\mathcal{O}(n \times w)$ . This adder requires  $n$  functional units for an  $(n \times w)$ -bit wide operation.

To measure the performance of the ripple-carry composition with ripple-carry base adder, we benchmarked the execution of a 32-bit addition as implemented by this adder, using either 8-bit, 16-bit, or 32-bit wide functional units. We compared this to a baseline implemented by the native 32-bit adder produced by Verilog's "+" operator. There measurements are summarised in Plot 5.1.

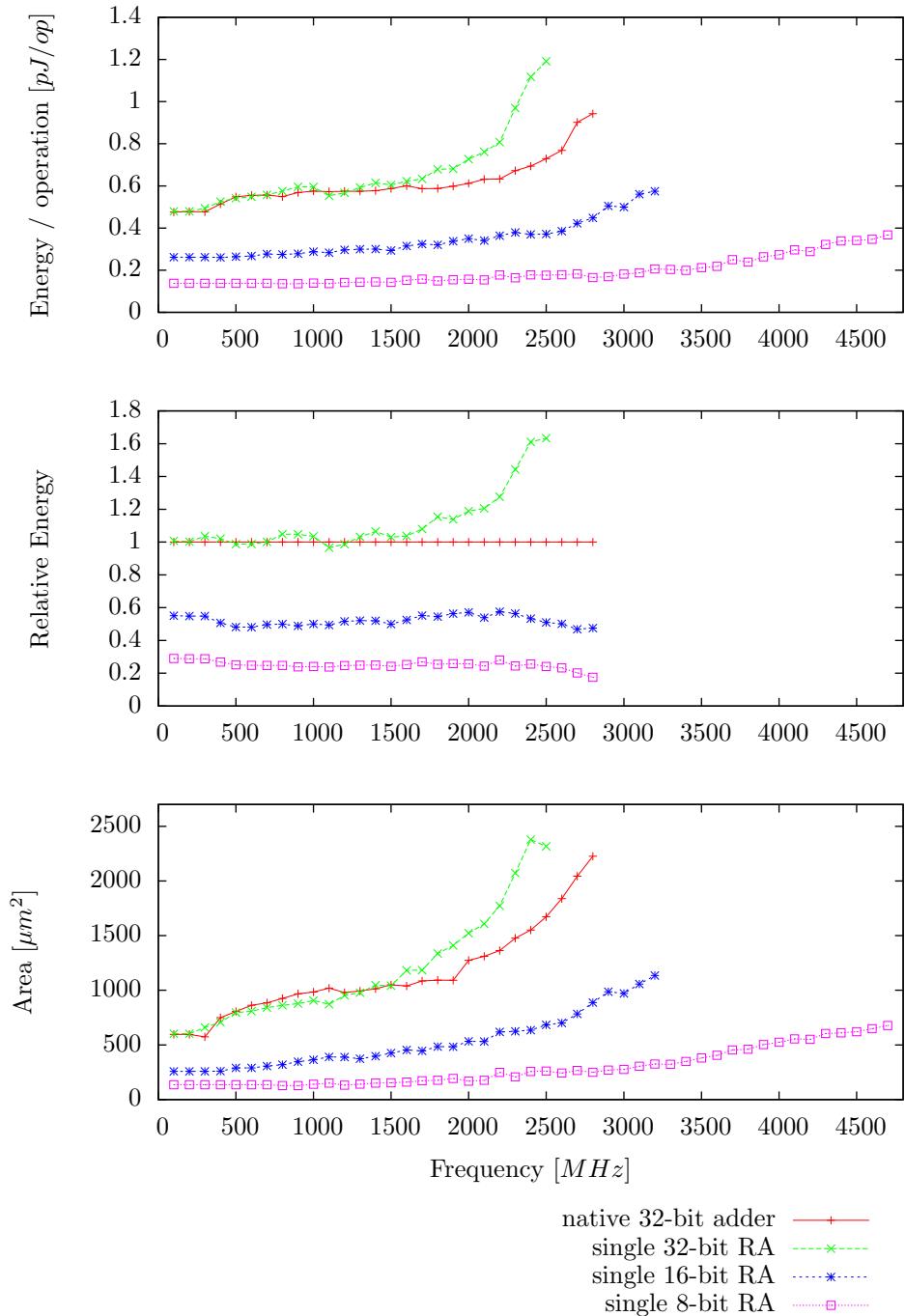
Up to 1500 MHz, the blocks size used for the decomposition of the ripple-carry adder has little effect on the energy and area usage for 32-bit additions, as the three different configurations are close together. Up to this point, the energy per operation is also fairly constant and the energy overhead is less than 15% compared to the baseline.

For higher frequencies, the energy per operation increases rapidly. The reason for this is not that the decomposition is expensive, but that the ripple-carry adder in general is the bottleneck here, as the single 32-bit RRA and quad 8-bit RRA have similar energy efficiency and area usage.

Plot 5.2 shows that, when performing 8-bit additions, the 8-bit adder performs the best. Energy usage can be as low as 0.14 pJ per addition, saving 70% from the 0.48 pJ per operation used by the baseline at low frequencies, and up to 80% at higher frequencies. When 16-bit operations are used, 45% energy savings are possible.



Plot 5.1: 32-bit addition on a ripple-carry composition with ripple-carry base adder (RRA).



Plot 5.2: 8-bit addition on a single ripple-carry base adder (RA).

### 5.3.2 Ripple-Carry Composition, Carry-Lookahead Base

The configuration consisting of functional units with carry-lookahead adders connected using a ripple-carry adder chain is called *Ripple-carry composition with carry-Lookahead base Adder* (RLA).

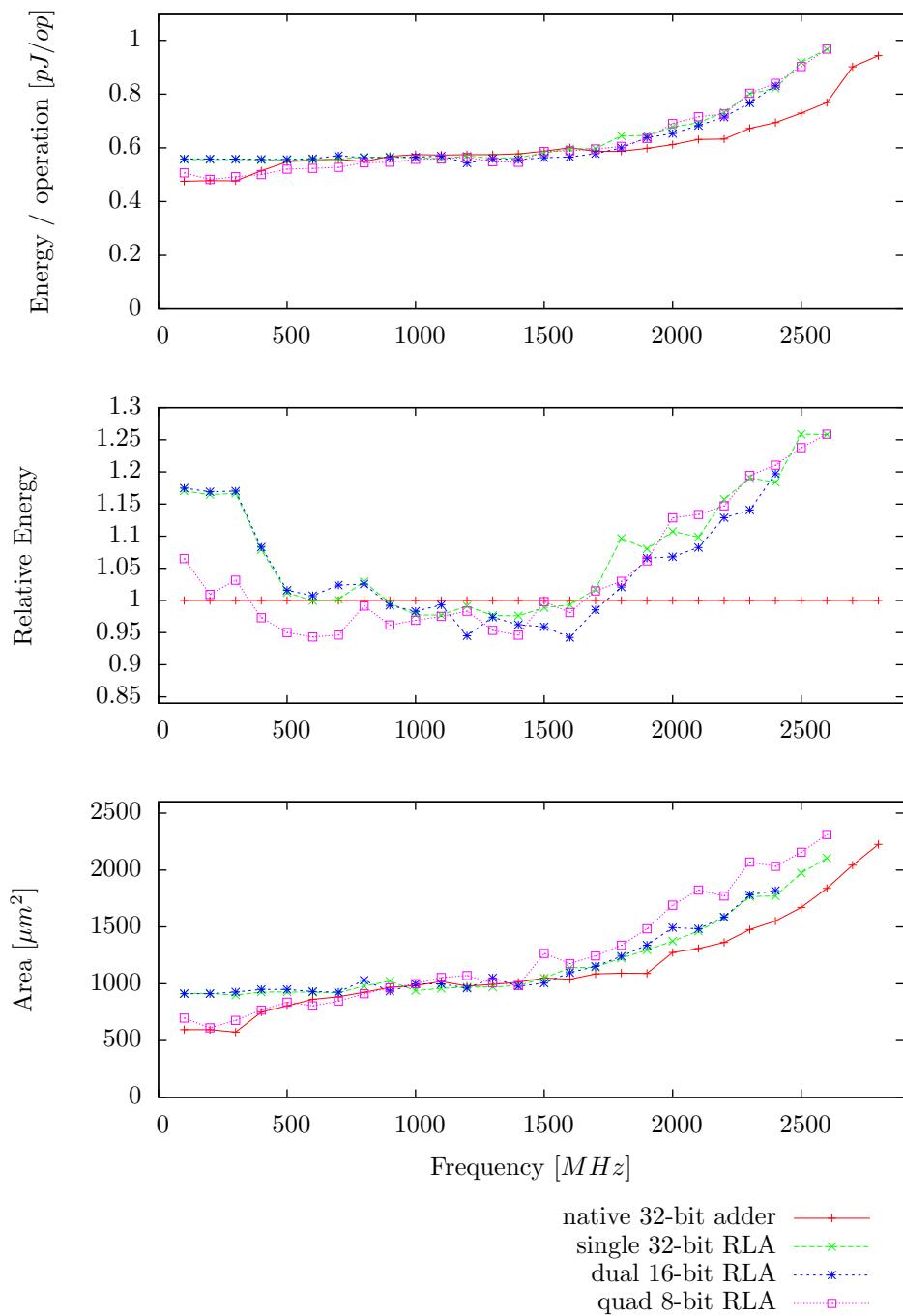
As discussed before, the input-to-output delay of a  $w$ -bit carry-lookahead adder is  $\mathcal{O}(\log w)$ . However, the carry-in to carry-out delay is constant,  $\mathcal{O}(1)$ , if the generate and propagate bits from the highest level carry-lookahead generator are available, which will be the case for all functional units except the first. Formula 5.3 can then be used to calculate the carry-out based on the carry-in. With a ripple between the functional units, which has a delay of  $\mathcal{O}(n)$ , the total delay for  $n$  functional units of each  $w$ -bit wide (thus calculating an  $(n \times w)$ -bit addition) will be  $\mathcal{O}(n + \log w)$ . This configuration will use  $n$  functional units.

This configurations looks promising: the carry propagation delay of the carry-lookahead adder is very low which should result in a low penalty when multiple functional units are chained, even when the slow ripple carry adder is used for the interconnect. The fast carry-lookahead adder ensures that the delay of a single functional unit is also low. However, compared to a simple ripple-carry adder, the extra logic used in the CLGs will also dissipate some energy and use extra area. For higher clock frequencies, this pays off as the reduced delay is worth the extra overhead, but at lower clock frequencies, the lower delay is not used and the energy and area overhead are wasted.

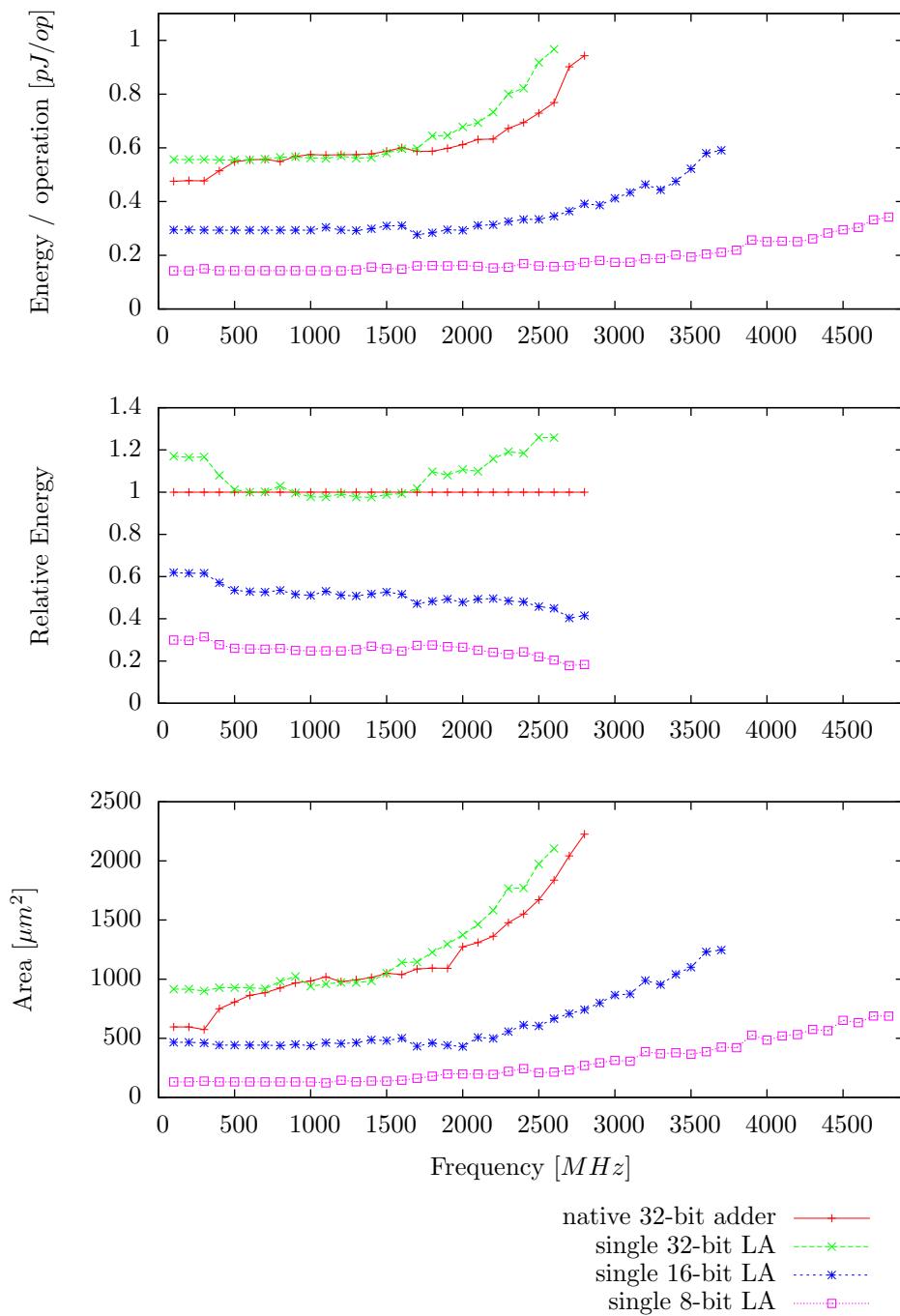
This can also been seen in Plot 5.3, especially the configurations with larger (16 and 32 bit) functional units show a high energy overhead at the lower frequencies, up to 400 MHz, compared to the baseline. The quad 8-bit functional unit configuration behaves more like a ripple-carry adder, and thus has a lower energy overhead. Up to 1700 MHz, the configurations perform as good or even better than the baseline. Starting at 1700 MHz, the energy per operation and area usage increases compared to the baseline, up to 25% overhead at 2500 MHz.

The decomposition seems to have a positive effect on the performance of this adder. The configuration with four 8-bit functional units performs better than the other two variants. This is because the ripple-carry adder is more energy efficient, and for a short ripple just as fast as the carry-lookahead generator.

When performing 8-bit additions, Plot 5.4 shows similar results to the ripple-carry composition with ripple-carry base adder: when performing small calculations, smaller functional units are better. This can lead to an energy reduction of 75% to 80% compared to the baseline for 8-bit functional units, and 40% to 60% for 16-bit functional units.



Plot 5.3: 32-bit addition on a ripple-carry composition with carry-lookahead base adder (RLA).



Plot 5.4: 8-bit addition on a single carry-lookahead base adder (LA).

### 5.3.3 Carry-Select Composition, Ripple-Carry Base

The combination of a ripple-carry adder inside the functional units and a carry-select adder to combine multiple functional units is called *carry-Select composition with Ripple-carry base Adder* (SRA).

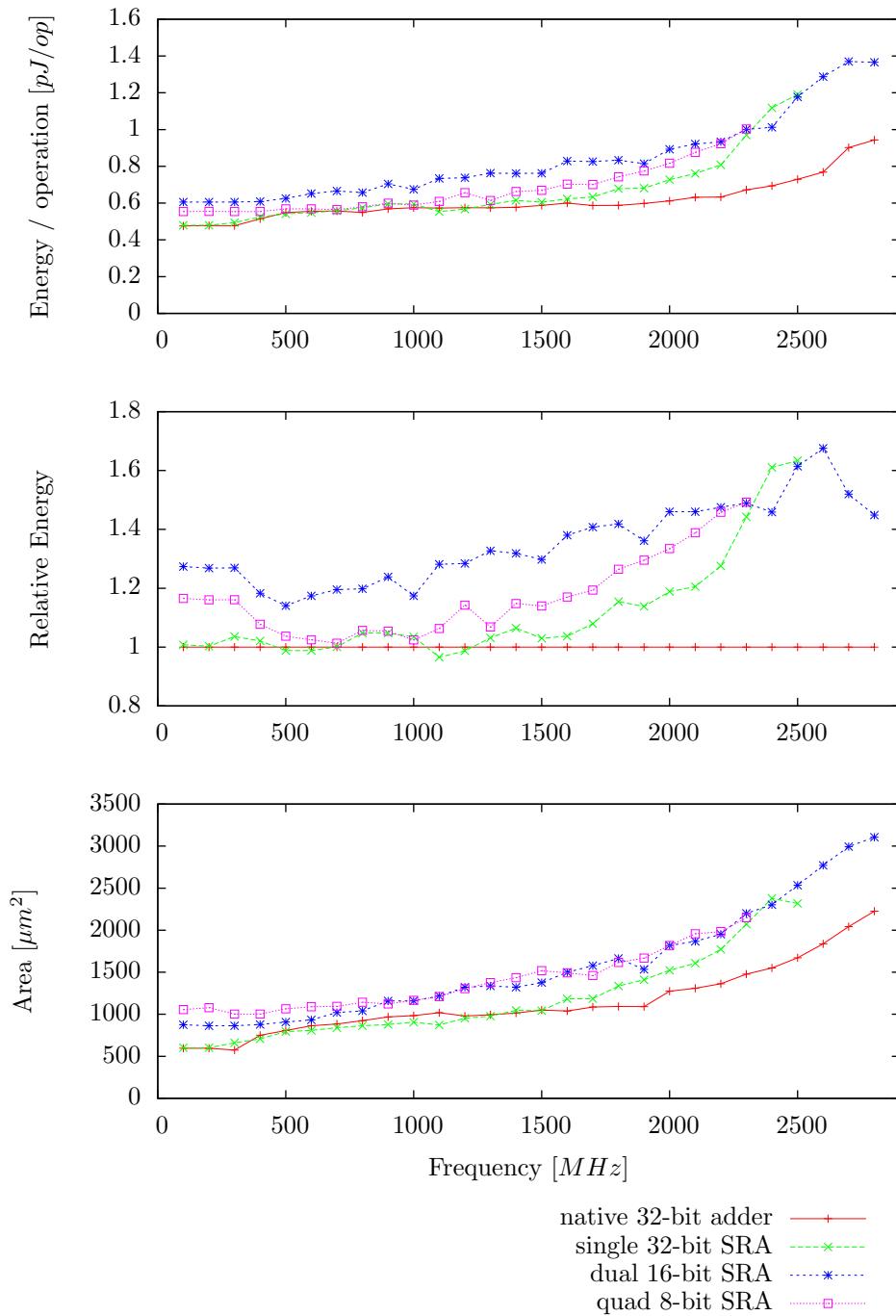
This design calculates all blocks other than the first twice, which results in an increased area and energy usage. The advantage is that the propagation delay is reduced compared to the ripple-carry interconnect, because after the delay of a single functional unit, all results are available and only the right answers have to be chosen. Thus the delay of this design is  $\mathcal{O}(w)$  for the  $w$ -bit functional units, plus  $\mathcal{O}(n)$  for the propagation of  $n$  functional units, for a total delay of  $\mathcal{O}(n + w)$ . This design uses  $2n - 1$  functional units.

The area usage as shown in Plot 5.5 is almost twice that of the baseline for the quad 8-bit configuration, and 1.5 times for the 16-bit configuration. This is as expected, as the 8-bit configuration is four words wide, which requires  $2 \times 4 - 1 = 7$  functional units, and the 16-bit configuration requires  $2 \times 2 - 1 = 3$  functional units.

The energy per operation of the dual 16-bit configuration is 15–30% higher compared to the baseline for frequencies lower than 1500 MHz. This is not unexpected, as 50% more work is done to speed up the calculations. However the quad 8-bit configuration is more energy efficient, having about 5–15% overhead compared to the baseline. As this configuration has more blocks, the carry-select algorithm works more effectively, also the sorter carry chains inside the functional units reduce the amount of toggling as the carry ripples through the full adders.

In addition to these extra area and energy costs, a more comprehensive interconnect is required. The interconnect must be able to route the signals to a destination that depends on another signal: the carry-out of the previous functional unit. Where the interconnect overhead of the previous configurations could be neglected because this was a local connection only, this is no longer the case for this configurations because the carry lines also have to be switched depending on the result of another functional unit. We nonetheless analyse the performance of this design without considering the overhead generated by the interconnect, because details of the interconnect are unfortunately not yet available.

The carry-select composition of a single ripple-carry functional unit is identical to the ripple-carry composition of a single ripple-carry functional unit. As a consequence, the benchmark results of the computation of 8-bit additions are identical to those discussed in Section 5.3.1. As depicted in Plot 5.2, this yields energy savings of up to 80% for 8-bit additions, and 45% for 16-bit additions.



Plot 5.5: 32-bit addition on a carry-select composition with ripple-carry base adder (SRA).

### 5.3.4 Carry-Select Composition, Carry-Lookahead Base

The configuration that consists of the carry-lookahead adder inside the functional units and where the carry-select adder forms the interconnect will be called *carry-Select composition with carry-Lookahead base Adder* (SLA).

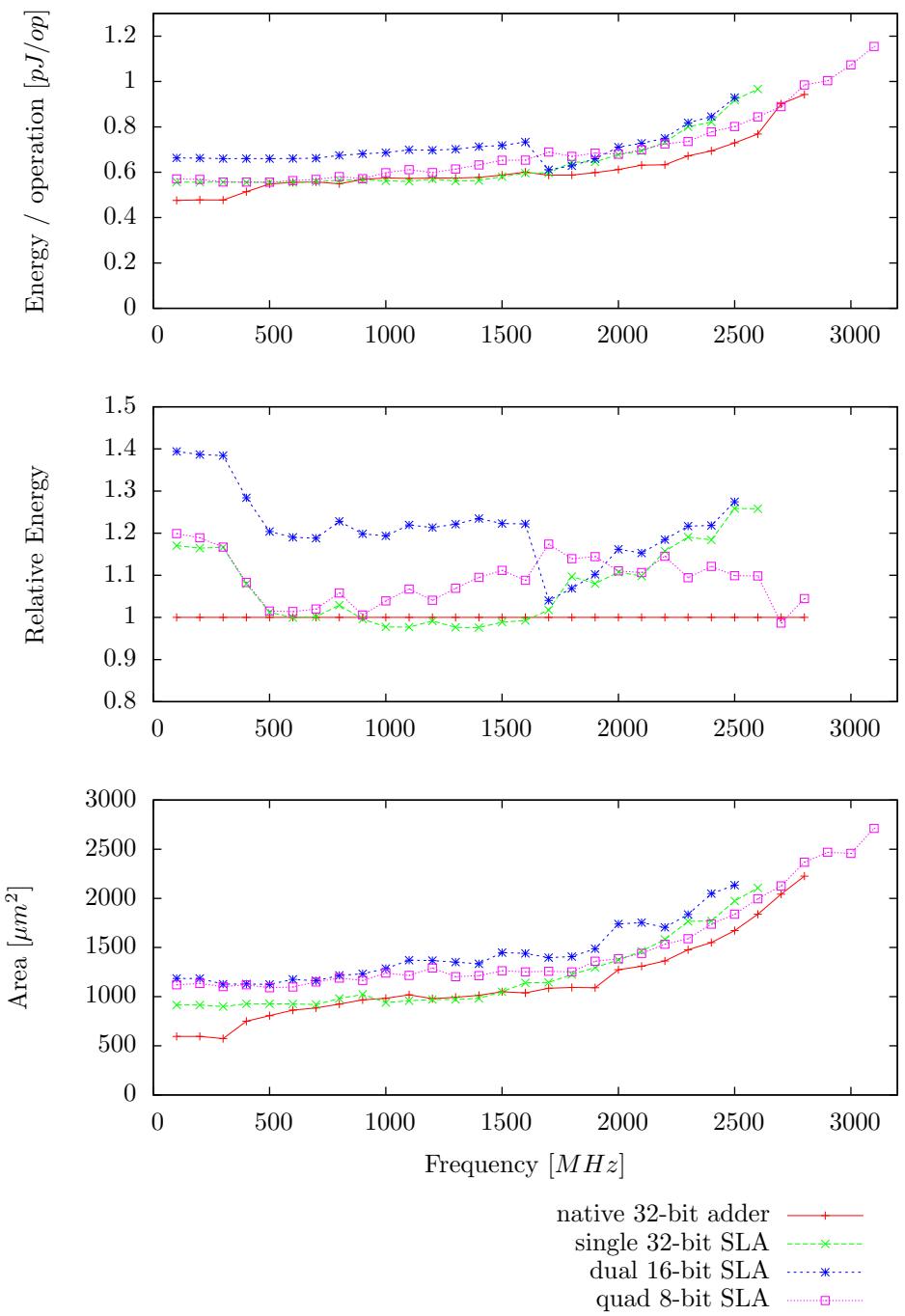
The carry-select adder is again used to connect the functional units, where each block in the carry-select algorithm is mapped to a single functional unit. Inside each functional unit there is a carry-lookahead adder which uses generate and propagate bits and a carry-lookahead generator to quickly produce the carries. Both the base and interconnect algorithms are the fastest algorithm that we will consider, but they also have their extra overhead costs in terms of energy and area.

A single functional unit with a carry-lookahead adder has an input-to-output delay of  $\mathcal{O}(\log w)$ , and the carry-in to carry-out delay is  $\mathcal{O}(1)$ . For a  $(n \times w)$ -bit addition, the carry-select algorithm has a delay equal to the input-to-output delay of a single functional unit, plus  $n$  times a constant carry selection delay. Combining this gives a total delay of  $\mathcal{O}(n + \log w)$  for this configuration. Note that the already fast carry-in to carry-out propagation of the carry-lookahead adder is not exploited by the carry-select algorithm.

The benchmark results shown in Plot 5.6 show some surprising results regarding the energy overhead, which varies from no overhead to 40% overhead compared to the baseline. Especially the dual 16-bit configuration shows some strange results, which seems to be the synthesis tool that is switching to another optimisation algorithm at 1700 MHz. From 500 to 1000 MHz, the single 32-bit and the quad 8-bit configuration are quite energy efficient, comparable to the baseline. At lower and higher frequencies, the overhead of these configurations increases. The quad 8-bit adder is the only configuration that can achieve clock speeds above 2500 MHz, up to 3000 MHz.

All configurations use more area than the baseline, varying from 20% extra area, up to twice the area for the dual 16-bit configuration at low frequencies, which then quickly drops to around 40%.

Again, the single-functional-unit configuration of this design is identical to the configuration discussed in Section 5.3.2. Plot 5.4 shows 70%–80% energy savings for 8-bit functional units, and 40%–60% savings for 16-bit functional units.



Plot 5.6: 32-bit addition on a carry-select composition with carry-lookahead base adder (SLA).

## 5.4 Comparison

We have seen four different multi-granular addition designs in the previous section — ripple-carry composition with ripple-carry base (RRA), ripple-carry composition with carry-lookahead base (RLA), carry-select composition with ripple-carry base (SRA), and carry-select composition with carry-lookahead base (SLA) — and discussed them individually. Multi-granular functional adder units can be used to perform wide additions at a higher cost than a native implementation of the appropriate width, but perform narrow additions at a higher efficiency than a generic-width native adder. When designing an architecture that can perform both wide and narrow additions efficiently, there is a trade-off between the cost savings when performing narrow additions, and the penalty when performing wide additions.

To estimate how this trade-off applies to the different adder designs we have analysed in this chapter, we model a use case where the architecture needs to be able to perform 8-bit and 32-bit additions efficiently. We analyse and compare how each of the above four adder designs behaves when performing either an 8-bit or an 32-bit addition; to furthermore investigate how the block size of the functional units affects this behaviour, we perform these analyses using 8-bit and 16-bit functional units.

In Table 5.1 we compare the behaviour of the four adder designs using a theoretical analysis. This comparison shows that the RRA design behaves very poorly in terms of delay when computing wide additions, independent of the used block size; however, it has an area usage that scales better than all the other designs. The remaining three designs all have much more favourable delay properties, at the cost of some area. Of these three, the RLA design uses the least number of functional units, which suggests it is the most attractive design.

Plots 5.7 and 5.8 depict the behaviour of different configurations performing 32 bit addition, constructed from 8-bit and 16-bit functional units respectively.

For the 8-bit configurations, we see that the cost of implementing a 32-bit addition in a multi-granular way is very modest for all four designs considered; only at very high frequencies does the penalty become significant. For those cases, the RLA design is the most energy efficient for frequencies up to 1900 MHz; starting at 2000 MHz, SLA takes over the leading position, reaching speeds of up to 3100 MHz. Regarding area, the ripple-carry composition adders are close to the baseline, while the carry-select adders use substantially more area.

Design	Delay	Area	FUs
RRA	$\mathcal{O}(n \times w)$	$\mathcal{O}(n \times w)$	$n$
RLA	$\mathcal{O}(n + \log w)$	$\mathcal{O}(n \times w \log w)$	$n$
SRA	$\mathcal{O}(n + w)$	$\mathcal{O}(n \times w)$	$2 \times n - 1$
SLA	$\mathcal{O}(n + \log w)$	$\mathcal{O}(n \times w \log w)$	$2 \times n - 1$

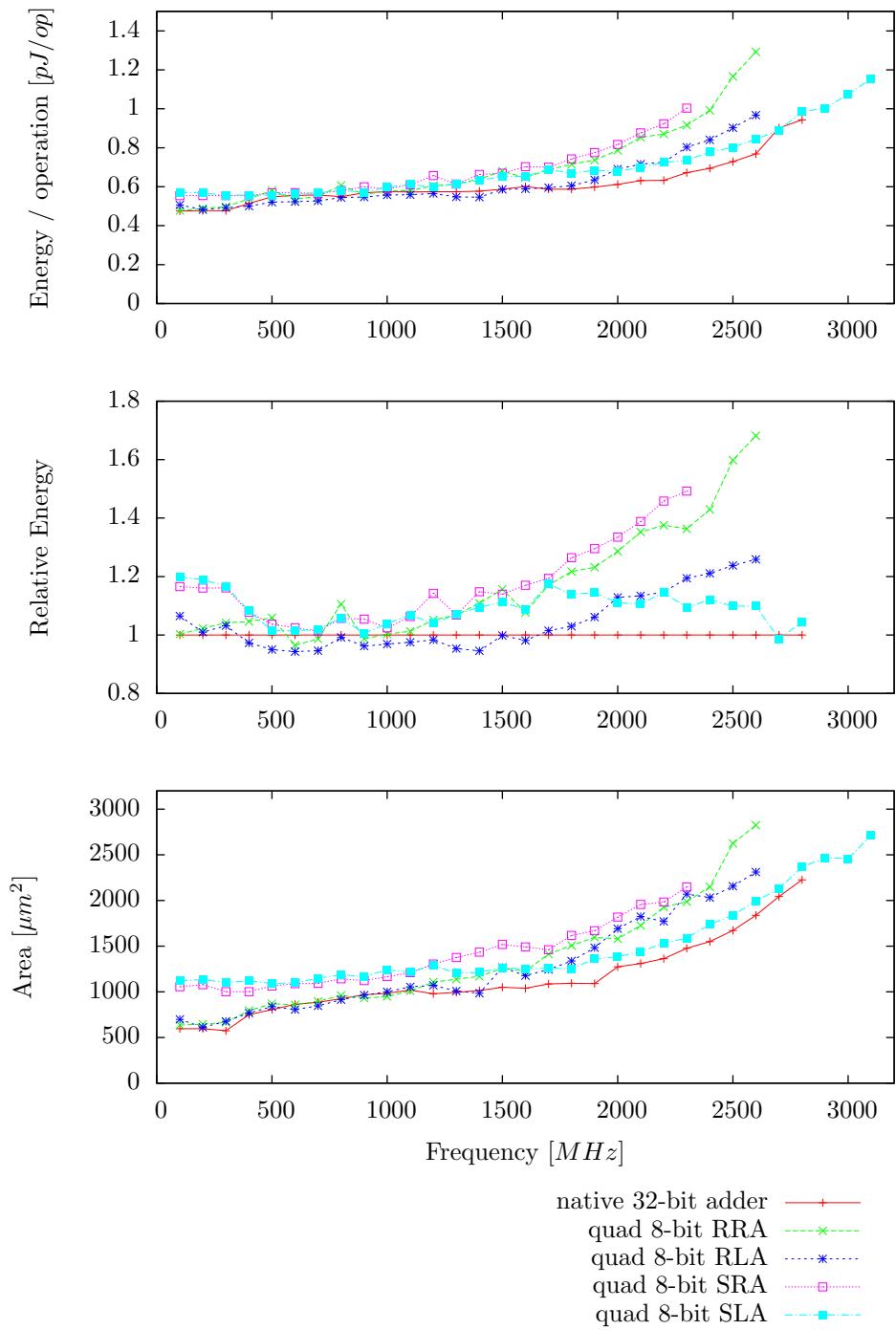
Table 5.1: Scalability for different adder designs implementing an  $(n \times w)$ -bit addition using  $w$ -bit wide functional units.

For the 16-bit configurations, the differences between the different designs are more pronounced. Much like the 8-bit case, the RLA design incurs a very limited efficiency penalty relative to the baseline; but the penalty for the other three designs is more notable. This generalisation is broken for frequencies less than 500 MHz; there the RRA design has the best performance, whereas the RLA design has 20% overhead. Similar to the 8-bit case, the SLA adder performs well at high frequencies; but unlike the 8-bit case, it never overtakes the performance of the RLA adder; nor does it reach very high frequencies.

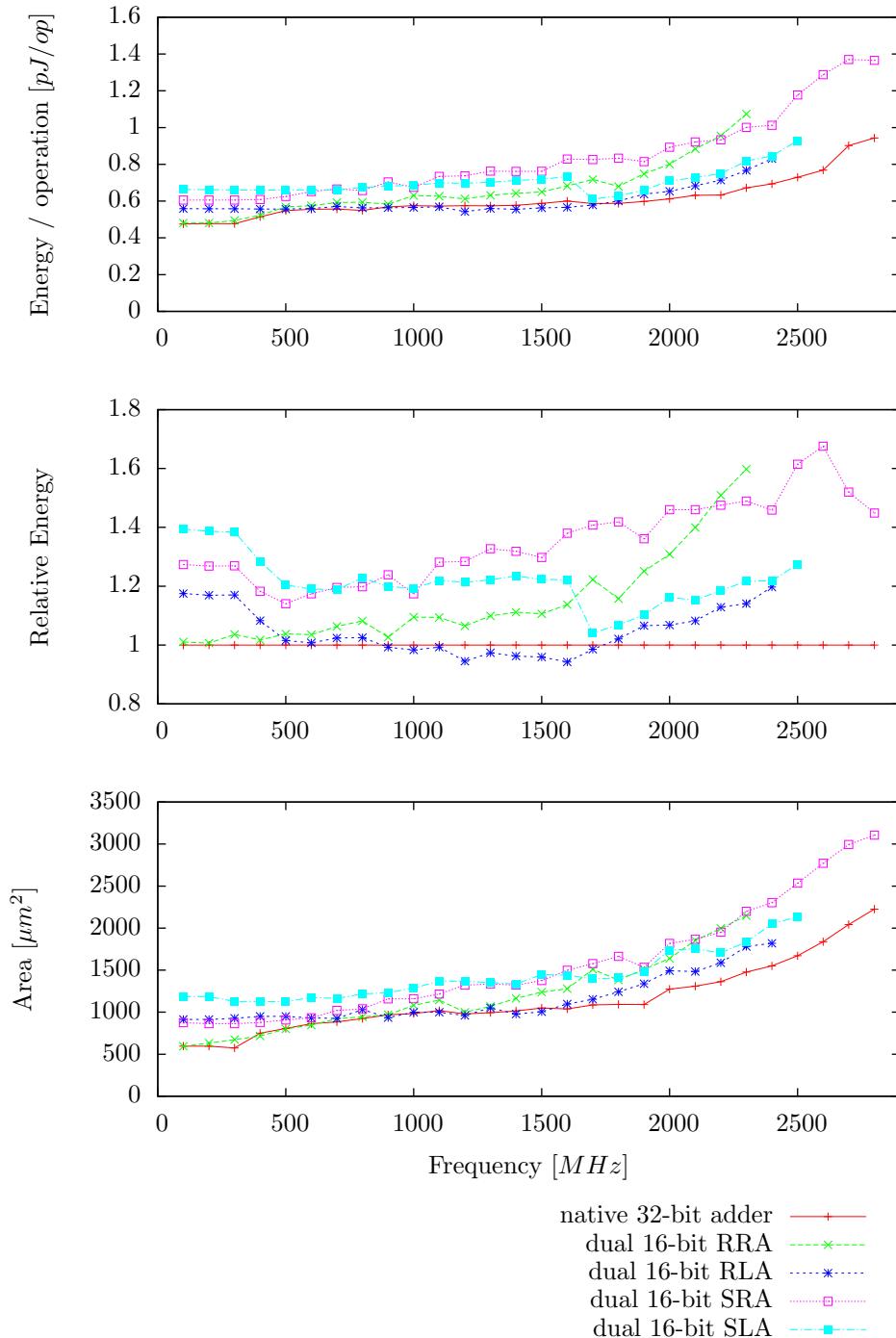
The performance of the different configurations when performing an 8-bit addition is depicted in Plot 5.9. Because an 8-bit addition can be performed by a single functional unit, no composition algorithm applies to these configurations; therefore, this plot shows only the ripple-carry adder and the carry-lookahead adder designs, for both 8-bit and 16-bit functional units.

When performing 8-bit additions using 8-bit functional units, both adder designs have similar performance. Energy usage is only 30–20% of the baseline, while area is also significantly lower: 20% of the baseline at 100 MHz, and only 12% at 2800 MHz, the highest operating frequency of the baseline. These functional units can operate up to 4800 MHz.

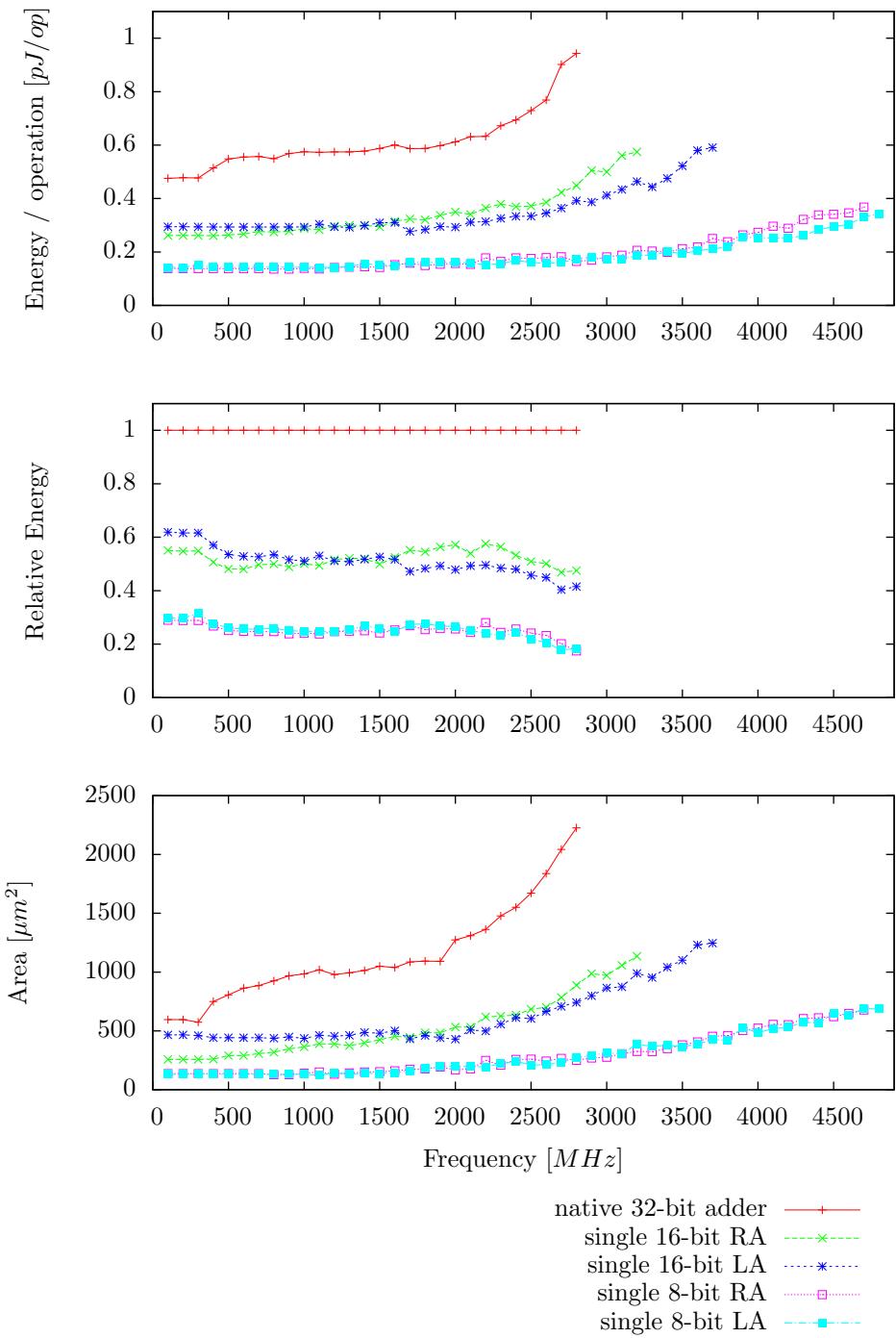
For the 16-bit functional units, energy usage is also similar for both algorithms. The ripple-carry adder performs slightly better up to 1100 MHz, and afterwards the carry-lookahead adder is more energy efficient; but in both cases, the difference is small. The energy usage is between 40% and 60% of the energy usage of the baseline. The area usage of the ripple-adder is lower than the carry-lookahead adder up to 1600 MHz; for higher frequencies, their area usage is comparable, but significantly lower than the baseline: only one third at 2800 MHz.



Plot 5.7: 32-bit addition using 8-bit functional units.



Plot 5.8: 32-bit addition using 16-bit functional units.



Plot 5.9: 8-bit addition using 8-bit and 16-bit functional units.

## 5.5 Conclusions

Based on the results analysed in the previous section, we can draw the following conclusions.

If high performance is not required and thus low clock speeds can be used, then the ripple-carry adder can be considered inside the functional units. At higher clock speeds, the carry-lookahead adder performs much better, and thus is to be preferred. Even when low frequencies can be used, the carry-lookahead adder might perform better when the decreased delay is exploited to apply voltage scaling, which can significantly reduce energy consumption. This is however not part of this investigation and considered future work.

For the interconnect between the functional units, the ripple-carry algorithm generally performs better, especially when combined with the carry-lookahead adder inside the functional units, unless really high frequencies are required; then the carry-select adder performs better. However it is not necessary to choose at design time. If the interconnect is capable of performing the carry-selection for the carry-select algorithm, then the compiler can construct either a ripple-carry composition or a carry-select composition at compile time: it can either choose to use the carry-select algorithm when high performance is necessary, or fall back to ripple-carry when possible to save energy. This gives the architecture two operating modes: a low performance and low power mode, and a high performance but high power mode.

The overhead costs for dividing a addition over multiple functional units is minimal; computing the addition this way incurs only a very limited energy efficiency penalty compared to a native implementation. This also means that large additions can efficiently be computed even on processors with small functional units.

The remaining operations that are discussed in the following chapters, accumulation, multiplication and multiply-accumulation, all use addition as a basic building block to form larger operations. In order to reduce the design space, we will use the ripple-carry composition with carry-lookahead base adder (RLA) configuration there, as this configuration generally performs the best.

# Chapter 6

## Accumulation

Accumulation is defined as repetitive addition. Instead of adding two numbers, as is done with regular addition, an arbitrary amount of numbers can be added. The sum  $s = a_0 + a_1 + a_2 + \dots + a_{n-1}$  can be calculated efficiently using the accumulation operation, for arbitrary values of  $n$ . The inputs  $a_i$ , with  $0 \leq i < n$ , are called summands.

### 6.1 Accumulation Algorithms

Accumulation is the operation of computing the sum of a large number of terms. An accumulation procedure begins with the execution of an *initialisation* step, in which the value of some sum register is set to zero. After initialisation, zero or more *accumulation* steps are performed, each supplying a single term to be added to the sum; this typically takes the form of an accumulation instruction executed in a loop. After accumulation proper, a *finalisation* step may be necessary to produce the final sum, depending on the accumulation algorithm used.

#### 6.1.1 Adder-Accumulator

One way to perform accumulation is to simply repeatedly perform an add instruction. The initialisation step consists of setting a register to zero, and each accumulation steps adds one term to this register. No separate finalisation step is necessary; the register contains the computed sum after performing the last accumulation step.

In the BLOCKS architecture based on functional units with internal registers, accumulation can be implemented efficiently by storing the intermediate result in a register inside the functional unit, as illustrated in Figure 6.1. In our

analysis of this design described later in this chapter, we implement this accumulator based on the carry-lookahead adder, or CLA, as concluded in Chapter 5.

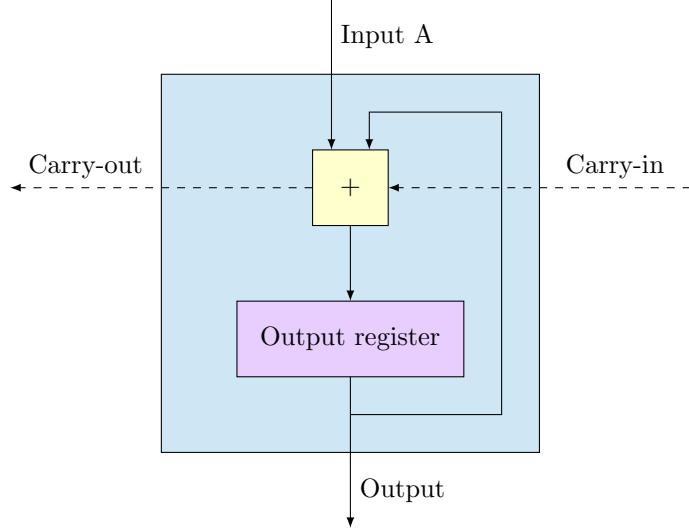


Figure 6.1: A simplified model of a functional unit implementing an adder-accumulator.

### 6.1.2 Carry-Save Accumulator

When performing accumulation, we are not interested in the results after each addition step; only during the finalisation step does the operation need to produce a usable sum. This means that the intermediate accumulated sum can be stored in a format that optimises the efficiency of the accumulation steps.

The *carry-save format* [15] is such a construction. The carry-save format represents an  $n$ -bit number as a pair of two  $n$ -bit numbers, whose sum is the value of the encoded number. This construction means that there are multiple different encodings of the same number; for this reason, this format is often called the *redundant format* in the literature. This format can be used to add terms to an accumulation register at excellent efficiency.

When adding an  $n$ -bit number in carry-save format to an  $n$ -bit number in regular binary format, one is effectively computing the sum of three numbers. For each of these  $n$  bits, one can compute a two-bit number between 0 and 3 that represents the sum of the two carry-save bits and the regular-number bit, as depicted in Figure 6.2.

Each digit in this  $n$ -digit sequence can be computed using a full adder using only bits from the input term and the accumulation register as input. A full adder, as described in Section 5.1, with input bits  $a$  and  $b$  and carry-in bit  $c$ , computes sum bit  $s$  and carry-out bit  $t$  such that  $a+b+c = s+2 \times t$ ; this yields the two-bit number above. As a consequence, the computation above can be

$$\begin{array}{r}
 01010011 \\
 11001001 \\
 +10011111 \\
 \hline
 21022123
 \end{array}$$

Figure 6.2: First step of a carry-save addition, summing three  $n$ -bit numbers to a sequence of  $n$  2-bit numbers.

implemented using an array of independent full adders, and thus have constant delay.

Each digit in this  $n$ -digit sequence is composed of the sum of a sum-bit  $s$ , and a carry-out bit  $t$  shifted one position to the left. As a consequence, the sequence can be interpreted as the sum of an  $n$ -bit number consisting of the concatenation of the  $s$ -bits, and an  $n$ -bit number shifted one position to the left consisting of the concatenation of the  $t$ -bits, as illustrated in Figure 6.3.

$$\begin{array}{r}
 =21022123 \\
 \hline
 01000101 \\
 10011011
 \end{array}$$

Figure 6.3: Interpretation of the result of a carry-save addition as two  $n$ -bit numbers.

This result, produced by the carry-save addition, is very nearly in carry-save format again. One complication is the most significant bit of the left-shifted output word; this bit represents a value of  $2^n$ , and therefore is too large to fit in the accumulation register. In other words, the most significant bit of the left-shifted output word of the carry-save addition functions as the *carry-out* of the accumulation step.

Conversely, the carry-save addition result has one missing bit at the location of the least significant bit of the left-shifted output word. Storing an arbitrary bit in this location has the effect of adding this bit to the number represented by the carry-save format; which means this missing bit provides an opportunity for the accumulation step to accept a *carry-in* bit. The logistics of the carry-in and carry-out bits are depicted in Figure 6.4.

$$\begin{array}{r}
 =21022123 \\
 \hline
 01000101 \\
 \text{carry-out} \leftarrow \boxed{1} 0011011 \boxed{\phantom{0}} \leftarrow \text{carry-in}
 \end{array}$$

Figure 6.4: Carry-in and carry-out in carry-save additions.

Taken together, these three pieces form a construction that can compute an accumulation step very efficiently. This process takes as input the contents of a register pair in carry-save format, an input number in the regular binary integer encoding, and a carry-in bit; and produces the sum of these three inputs in the

form of a carry-save format pair, as well as a carry-out bit. Because all of these output bits are computed using just a full adder, this accumulation step can be computed using very little delay. A summary of the carry-save accumulation procedure is illustrated in Figure 6.5.

$$\begin{array}{r}
 01010011 \quad \text{carry-save register} \\
 11001001 \\
 +10011111 \quad \text{input term} \\
 \hline
 =21022123 \\
 01000101 \\
 \text{carry-out} \leftarrow \boxed{1} \boxed{0011011} \boxed{1} \leftarrow \text{carry-in} \\
 \downarrow \\
 \text{to carry-save register}
 \end{array}$$

Figure 6.5: Summary of a carry-save accumulation procedure.

A downside of the carry-save addition algorithm lies in the fact that the accumulated sum is not immediately available after accumulating the last summand term. In order to transform the pair of registers making up the carry-save format into a number encoded in the usual way, these two registers need to be added using a regular addition algorithm. This results in added area usage to accomodate the final regular adder; it also means that a carry-save accumulation procedure requires an extra cycle after accumulating the last summand to perform the final addition. Furthermore, an  $n$ -bit carry-save accumulator requires two  $n$ -bit registers to store the accumulated sum, whereas a simple adder-accumulator needs only a single  $n$ -bit register.

Figure 6.6 shows how the carry-save adder can be used to form an accumulator. Note that both the carry-save adder as well as the carry-lookahead adder use the same carry in and carry out pins, and thus these two adders cannot be used in the same cycle.

## 6.2 Multi-Granular Accumulation

We constructed two distinct designs in order to perform accumulation in a multi-granular way. We present these designs in the remainder of this section.

### 6.2.1 Ripple-Carry Accumulator

Similar to a ripple-carry adder, a basic multi-granular accumulator can be constructed simply by connecting multiple accumulators together in a ripple-carry chain. We call this construction a *ripple-carry accumulator*, as depicted in Figure 6.7.

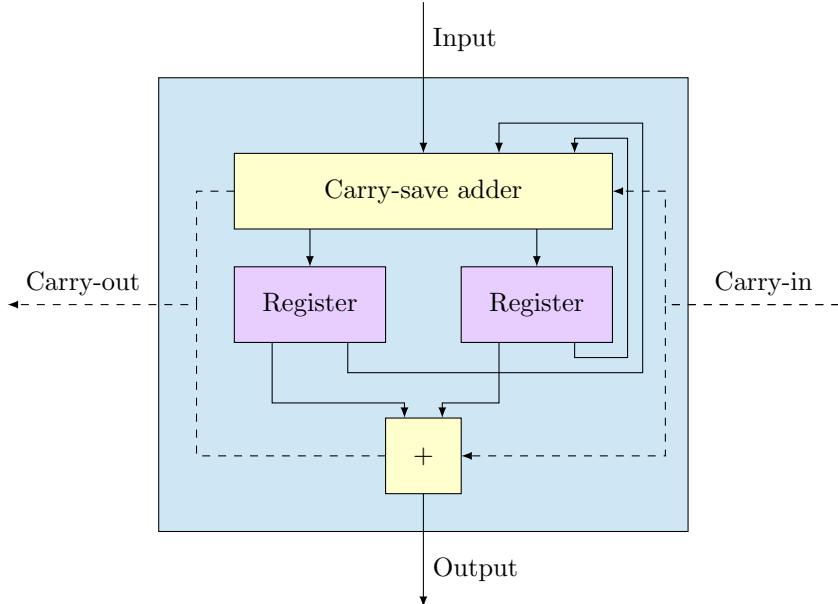


Figure 6.6: Simplified representation of a carry-save accumulator.

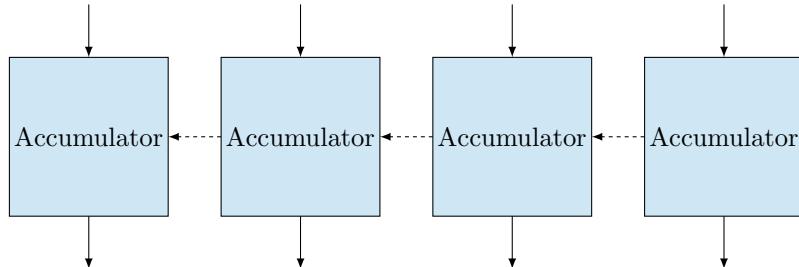


Figure 6.7: Construction of a  $4w$ -bit ripple-carry accumulator.

### 6.2.2 Carry-Accumulation

Instead of using the carry-chain network, we could accumulate the carry-out bits generated by each accumulator in an additional *carry-accumulation register* inside these functional units. A consequence of this design is that after all the accumulation steps are completed, a few more cycles are needed to assemble the final result, as is illustrated in Figure 6.9.

At the end of the accumulation procedure, each accumulator holds the sum of all accumulated carry bits inside its carry-accumulation register; in order to compute the final result of the accumulation, these registers have to be added to the output words whose values they represent. This can be done by adding the carry-accumulation register of each accumulator to the sum of the next-most-significant accumulator, as depicted in Figure 6.8.

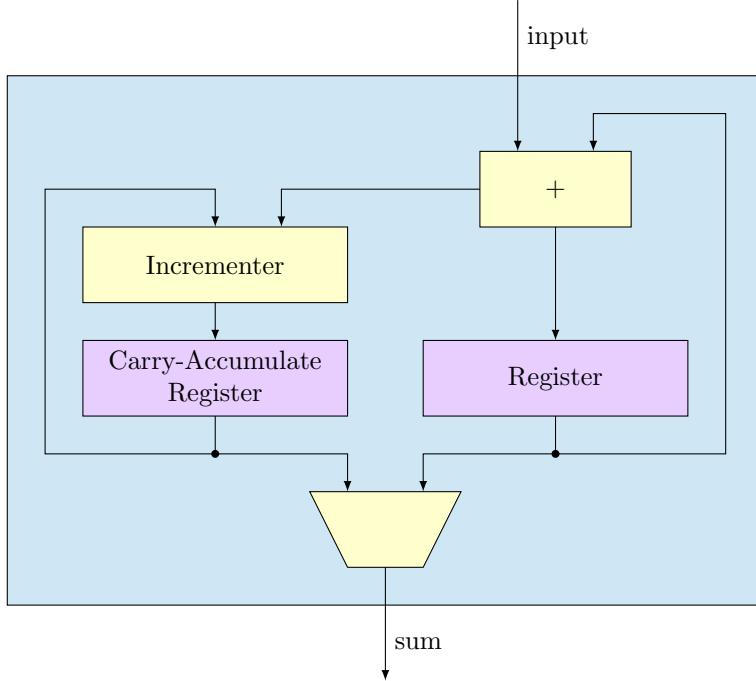


Figure 6.8: Schematic overview of a carry-accumulate functional unit.

This algorithm has the advantage that the different accumulating functional units do not have to be physically close to each other, which is required when a ripple-carry interconnect is used, as the interconnect network for the carry propagation is only locally connected to ensure a low delay.

This design also has some disadvantages: an extra register is required to store the overflow carry bits, and extra cycles are required to process the stored carries after the accumulation is completed. These stored carries have to “ripple” through the used functional units, so an accumulation that is  $n$  functional units wide, requires  $n$  extra cycles to propagate the carries.

Additionally, the amount of accumulation steps that can be executed is limited, to ensure that the carry accumulation register does not overflow. How many accumulate steps can be processed before the overflow happens depends on the width of the register. An  $r$ -bit carry-accumulate register allows  $2^r - 1$  accumulate steps before an overflow can happen. This register should be the same size or smaller than the width of a functional unit, as otherwise this register could not be propagated in a single cycle to the next functional unit.

Luckily this limit can be resolved by propagating, and thus clearing, the carry registers to the next functional unit. In contrast to the final propagation, can this be done by all functional units in parallel, as the newly created carry bit can again be stored in the accumulate register.

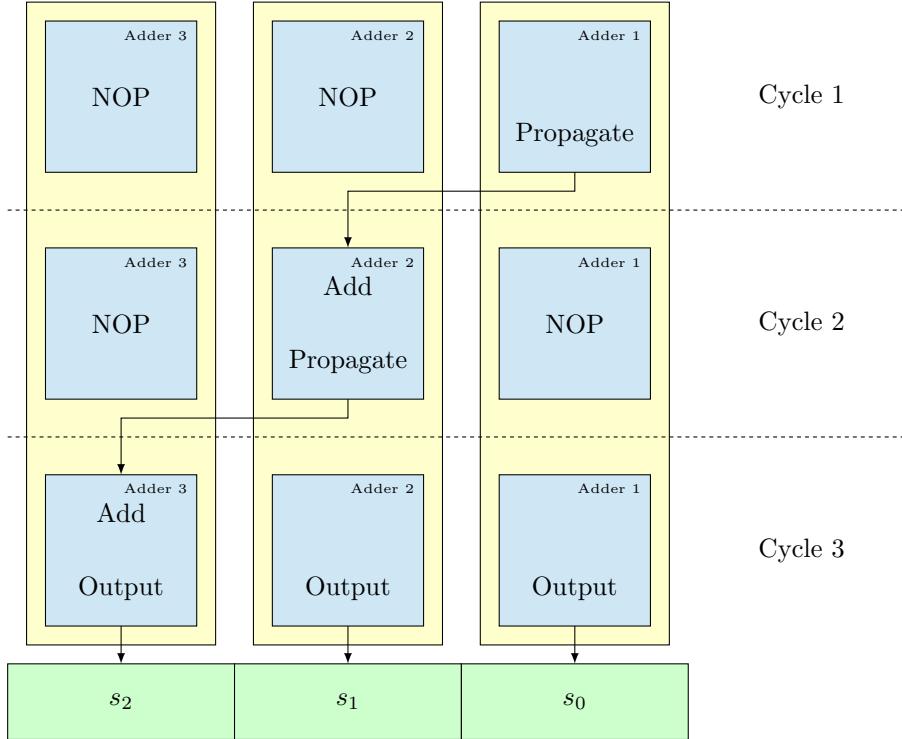


Figure 6.9: A  $(3 \times w)$ -bit carry-accumulation accumulator takes 3 cycles to propagate accumulated carries and produce the final result.

Cycle 1: Adder 1 outputs its accumulated-carry register.  
 Cycle 2: Adder 2 accumulates its input, and afterwards outputs its updated accumulated-carry register.  
 Cycle 3: Adder 3 accumulates its input, and afterwards outputs its sum register. Adders 1 and 2 output their sum registers without doing anything else.

### 6.3 Multi-Granular Accumulator Configurations

We can combine the carry-save and carry-lookahead adders both with a ripple-carry interconnect and with the carry-accumulation technique, leading to four combinations. These combinations will be discussed in this section.

As a baseline we used a native 32-bit accumulator. This accumulator consists of an adder implemented with the Verilog “+” operator, and a 32-bit register to store the intermediate result.

To benchmark our designs, we perform summations with 100 summands, and we count each accumulation step as a single operation. Thus, when we speak of energy per operation, we mean energy per accumulate step.

### 6.3.1 Ripple-Carry Composition, Carry-Lookahead Base

The accumulator built from a carry-lookahead adder inside the functional units and a ripple-carry composition is called *Ripple-carry composition with carry-Lookahead base Accumulator* (RLAC).

This design is very similar to the ripple-carry composition with carry-lookahead base adder as discussed in Section 5.3.2. The main difference is that, instead of sending the result of each addition to the output, it is written to the accumulate register, and instead of using two inputs for the addition, only one input is used; the second operand comes from the accumulate register. As the carry-lookahead adder has performance similar to the baseline adder when performing 32-bit additions, we expect the same here.

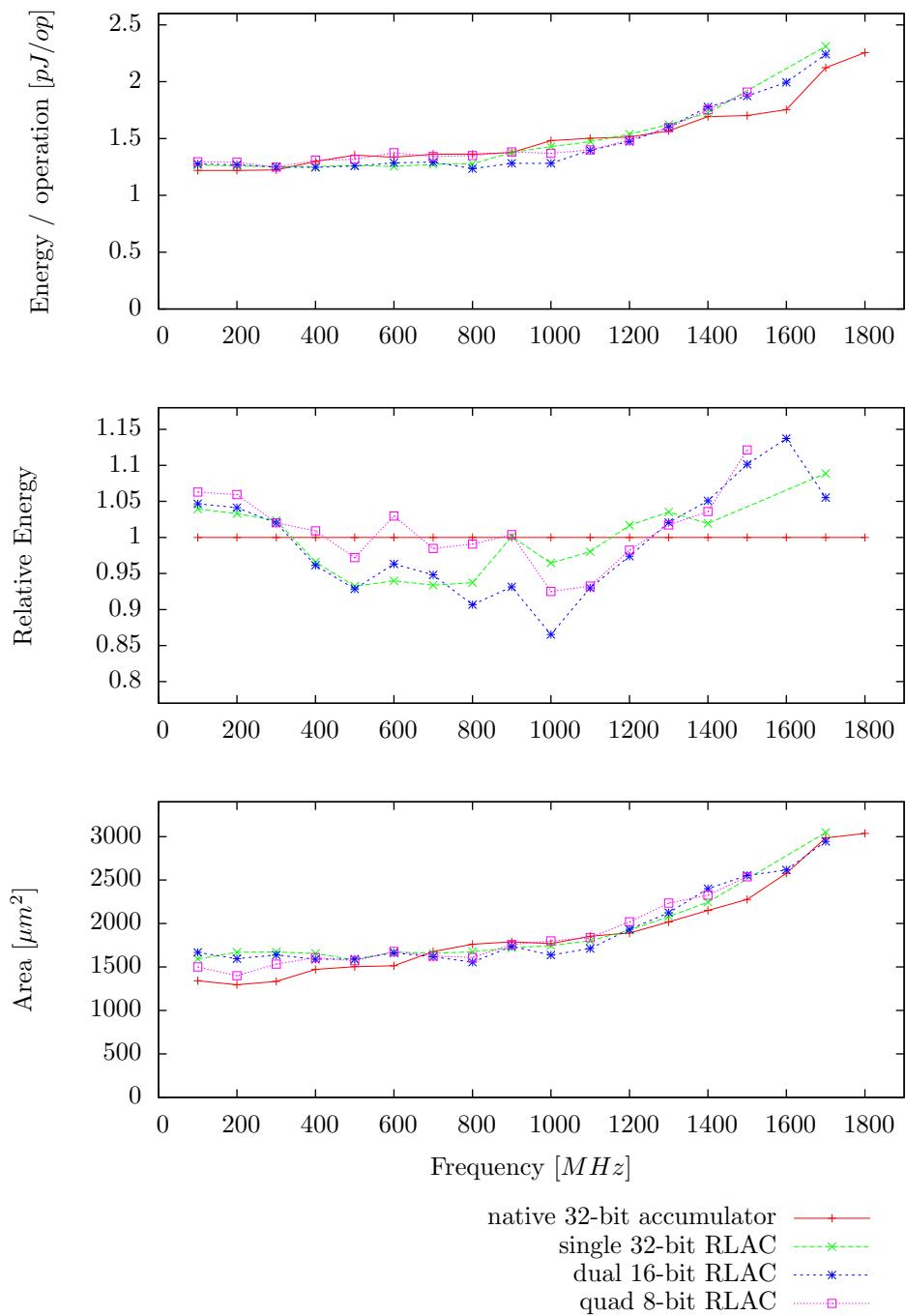
Plot 6.1 shows exactly this. This accumulator performs 5–15% better than the baseline regarding energy efficiency from 400 MHz until 1200 MHz. For lower and higher frequencies, this design performs 5–10% worse than the baseline, which seems to be due to the carry-lookahead adder; for the low frequencies, the ripple-carry adder might be a better choice, as discussed in Section 5.5.

Regarding the multi-granular separation, we see a slightly odd pattern, although not very distinct: the dual 16-bit configuration generally performs the best. However the difference is small, and could partially be due to the unpredictability of the synthesis tools (as described in Section 4.3), so we could say that the multi-granular separation has no large impact on this design.

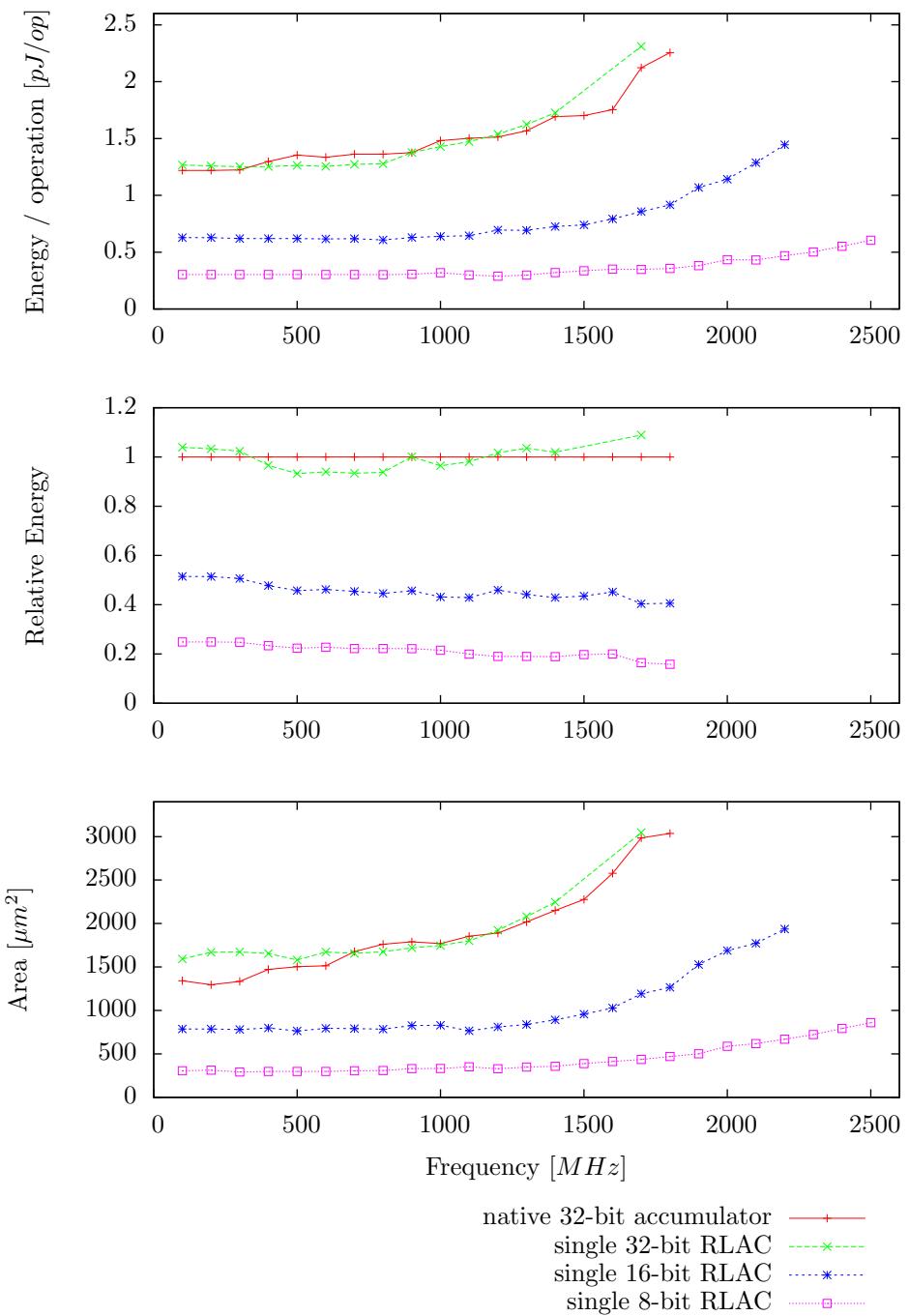
The area requirements for this design are slightly higher than the baseline for the most part of the frequency spectrum, with the exception of the 700–1100 MHz range, where the area usage of all configurations is slightly lower.

In Plot 6.2 we compare 8-bit accumulation performed by a 32-bit functional unit, a 16-bit functional unit and an 8-bit functional unit, all with this design. Next to that we also have the 32-bit baseline accumulator. Here we see the real benefit of the multi-granular approach, as the 8-bit configuration uses only 15–25% of the energy of the baseline, and the 16-bit configuration uses 40–50% of the energy of the baseline for 8-bit accumulation.

When we look at area, the difference is almost as large: the 8-bit functional unit uses  $471 \mu\text{m}^2$  at 1800 MHz, while the baseline requires  $3036 \mu\text{m}^2$ ; more than a factor 6 difference.



Plot 6.1: 32-bit accumulation on a ripple-carry composition with a carry-lookahead accumulator (RLAC).



Plot 6.2: 8-bit accumulation on a ripple-carry composition with a carry-lookahead accumulator (RLAC).

### 6.3.2 Ripple-Carry Composition, Carry-Save Base

The combination of a carry-save adder in the functional units together with a ripple-carry composition is called a *Ripple-carry composition with carry-Save base Accumulator* (RSAC).

In this design, each functional unit contains, in addition to the carry-lookahead adder, a carry-save adder and an extra register to save the intermediate results in the carry-save format. This way the accumulation steps can be done using the cheap and fast carry-save adder, and only the final addition, to produce the accumulation result, has to be done by the carry-lookahead adder.

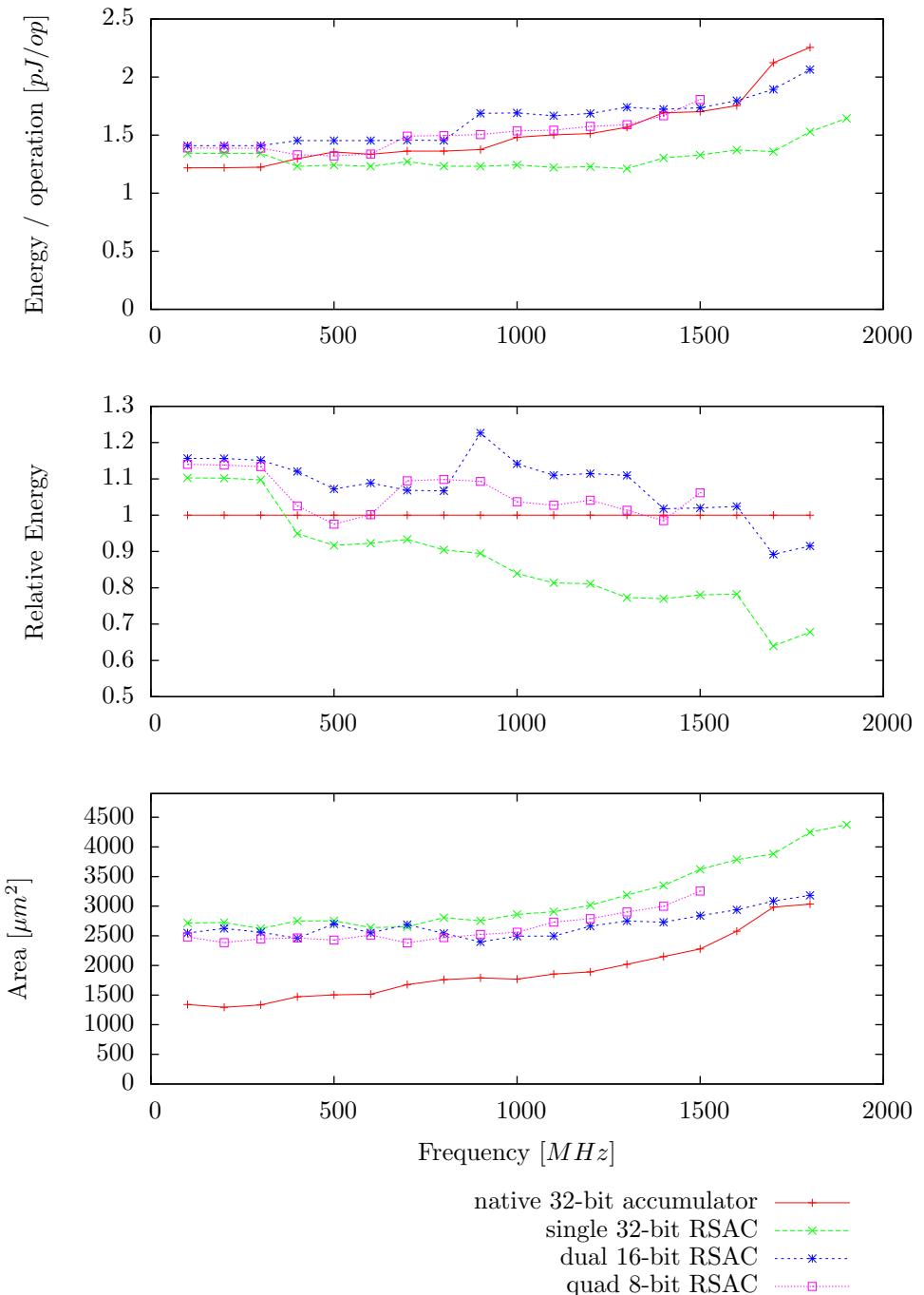
As extra components are added — the carry-save adder and a second register — it is expected that the area used by this design is higher than the baseline. The extra register will also increase energy usage, while the carry-save adder is more energy efficient than the carry-lookahead adder. The difference between the two addition algorithms will become more pronounced as the operating frequency increases, so we expect the performance, compared to the baseline, to improve as the frequency increases.

Plot 6.3 shows the benchmark results for 32-bit accumulations, and we can see that the area is nearly twice as large as the baseline for all configurations at low frequencies, but the area of these designs is constant until 1100 MHz, and after that it only grows slowly. At 1700 MHz, the area of the dual 16-bit configuration and the baseline are almost the same.

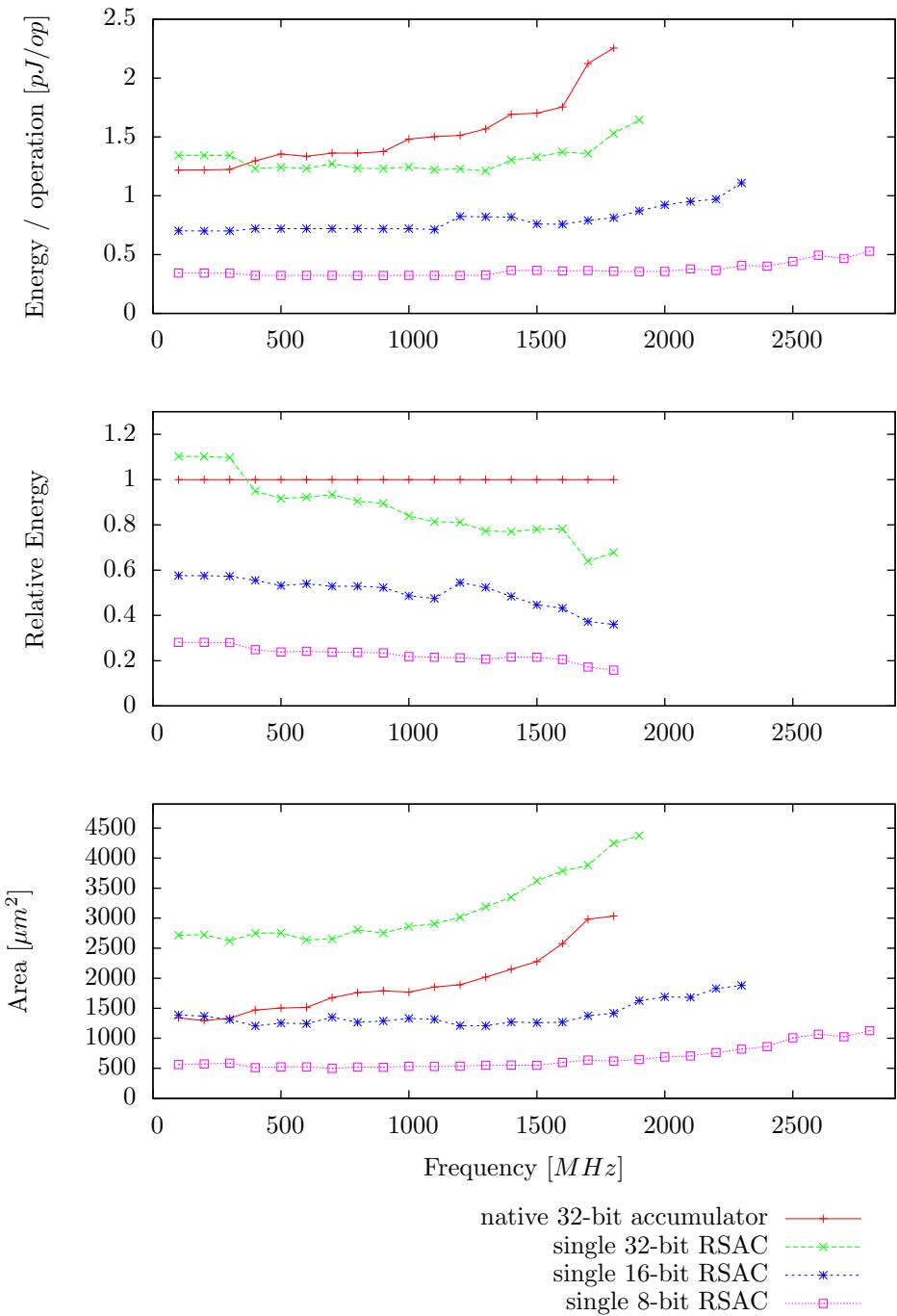
The energy usage is 10–15% higher than the baseline at 100 MHz. After that point, the quad 8-bit and dual 16-bit configurations slowly move towards the baseline, while the energy usage of the 32-bit configuration remains stable at around 1.21 pJ/operation, until it uses only 65% of the energy of the baseline at 1700 MHz.

The benchmark results for 8-bit accumulation, depicted in Plot 6.4, show that the energy per operation of the 8-bit and 16-bit functional units is very constant: the 8-bit configuration starts at 0.34 pJ per operation, and grows to 0.36 pJ per operation at 2000 MHz. This results in a relative energy usage of 20–25% compared to the baseline. The 16-bit configuration uses 0.70 pJ per operation, 40–60% of the energy that the baseline uses.

The area plot shows nothing surprising: the area of the single 8-bit configuration is half as large as the 16-bit configuration, which again is half as large as the 32-bit configuration.



Plot 6.3: 32-bit accumulation on a ripple-carry composition with a carry-save accumulator (RSAC).



Plot 6.4: 8-bit accumulation on a ripple-carry composition with carry-save accumulator (RSAC).

### 6.3.3 Carry-Accumulate Composition, Carry-Lookahead Base

When the carry chain is exchanged for a carry-accumulate register, combined with a carry-lookahead adder, we have created a *carry-Accumulate composition with carry-Lookahead base Accumulator* (ALAC).

The functional units in this multi-granular accumulator are not directly connected, as there is no carry chain. Instead, each functional unit is equipped with an extra register that can accumulate the produced carry bits, and when all summands are accumulated, this register is added to the next functional unit.

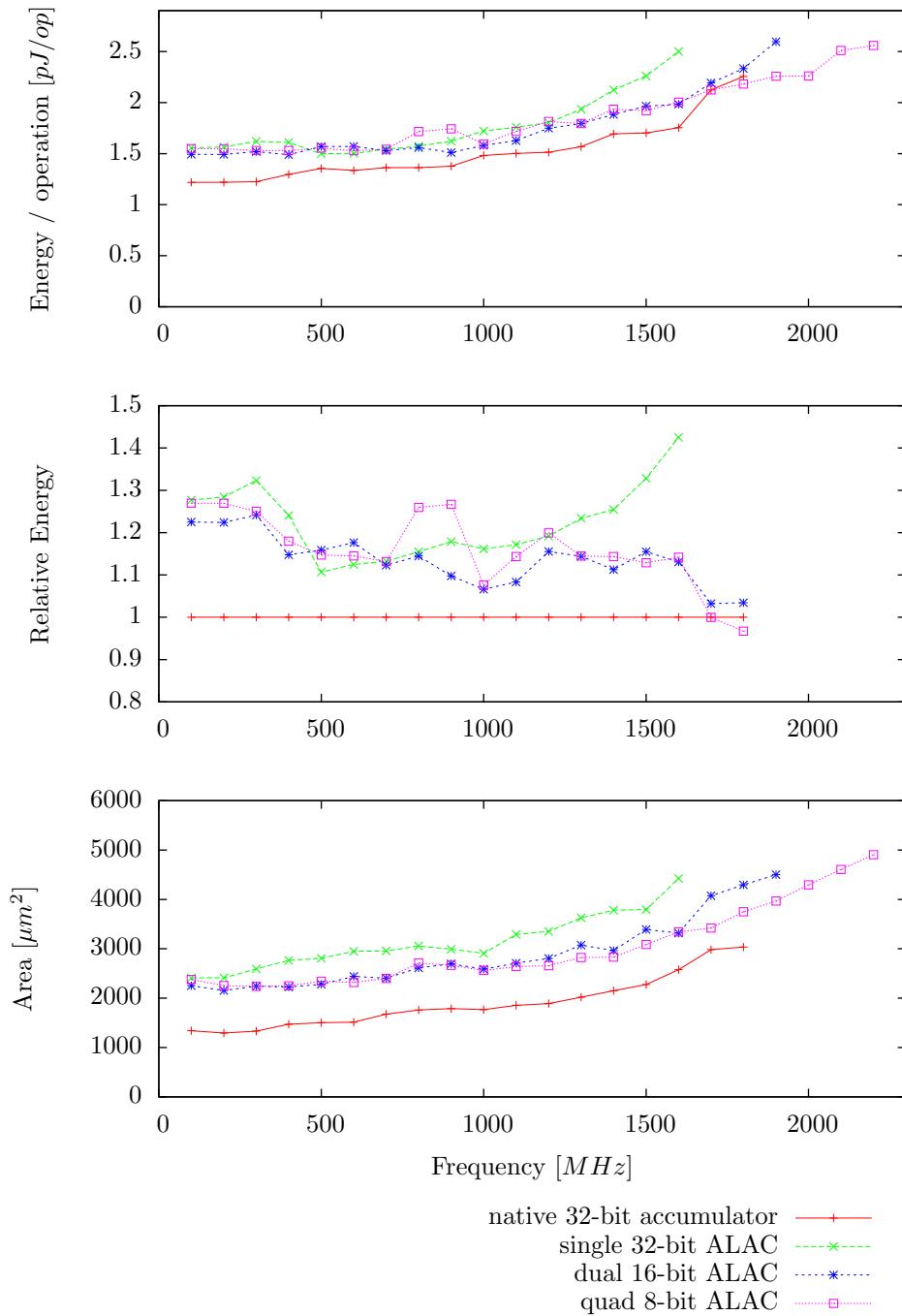
This means that two extra components are added: first an extra register that accumulates the carry bits. In our implementation this register is as wide as the functional unit, but this is not required. This register could be made smaller to save some energy and area, but this limits the amount of accumulation steps that can be executed before the carry register has to be propagated to the next functional unit. Secondly, an incrementer is added to the functional units. This module takes the value from the carry register, and adds the carry-out bit produced by the carry-lookahead adder.

In Plot 6.5, we see that this design is not particularly energy efficient: the energy usage is 20–30% higher than the baseline at 100–300 MHz, and is up to 1600 MHz more than 10% higher for the quad 8-bit and dual 16-bit configurations, and the single 32-bit configuration has even 40% overhead compared to the baseline. This design also does not perform very well in terms of area usage; it is about 50% larger than the baseline.

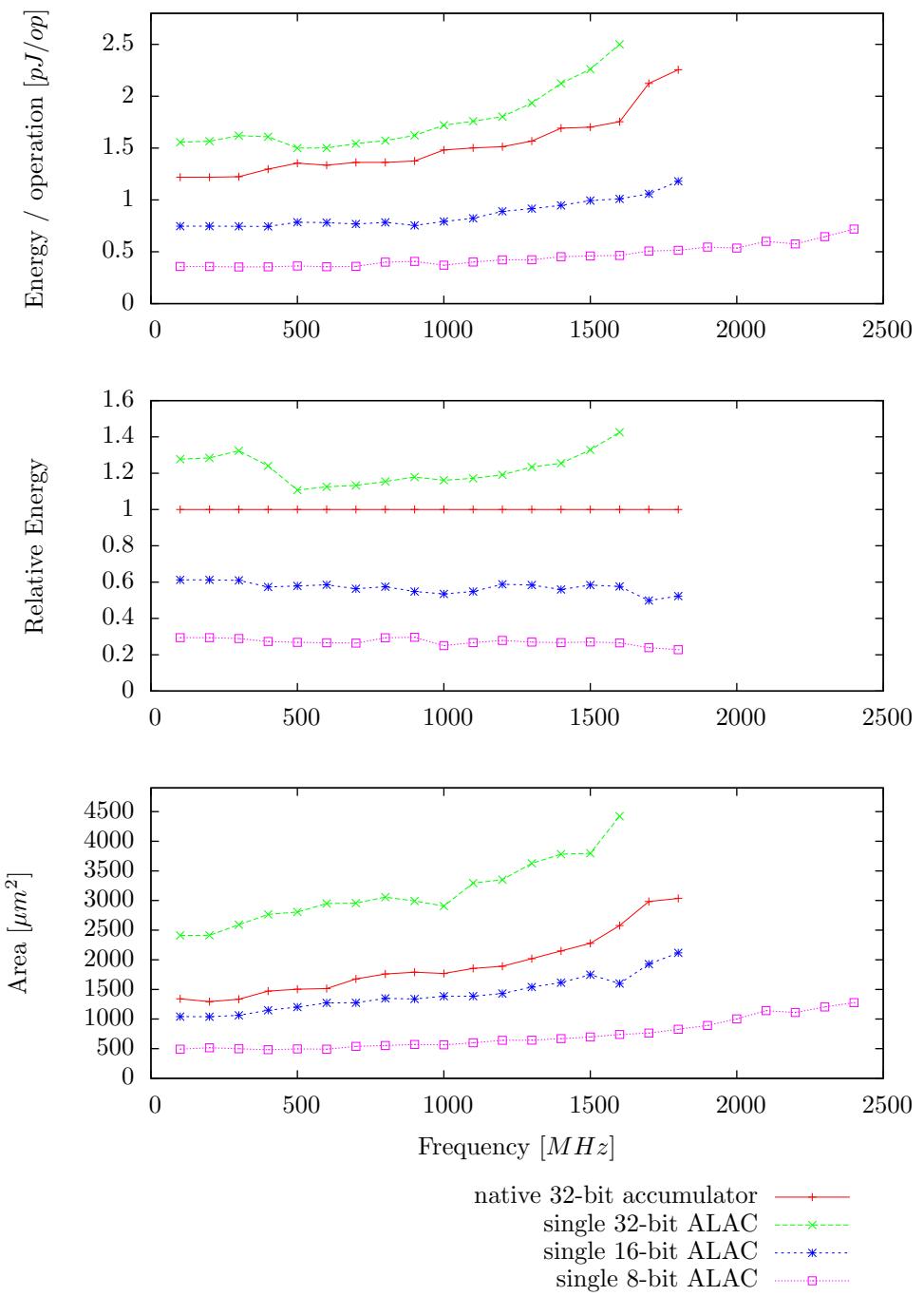
However, this design does seem to benefit from the multi-granular separation; the quad 8-bit accumulator is the smallest, and most energy efficient configuration. This configuration is also able to reach the highest operation speeds, up to 2200 MHz, where the baseline stops at 1800 MHz.

The 8-bit accumulations shown in Plot 6.6 display a rather energy efficient 8-bit functional unit, using only 20–25% of the energy per operation of the baseline, while the 16-bit configuration uses about 60% of the energy.

So the advantage of this design is not energy or area efficiency, but flexibility. These functional units do not have to be physically close to each other, which might be useful when the upper and lower part of the inputs come from different parts in the grid.



Plot 6.5: 32-bit accumulation on a carry-accumulate composition with a carry-lookahead accumulator (ALAC).



Plot 6.6: 8-bit accumulation on a carry-accumulate composition with a carry-lookahead accumulator (ALAC).

### 6.3.4 Carry-Accumulate Composition, Carry-Save Base

The last accumulator design uses both the carry-accumulate technique, as well as the carry-save technique. This together forms the *carry-Accumulate composition with carry-Save base Accumulator* (ASAC).

By combining these two techniques, the total number of registers in this design becomes four: two registers to save the intermediate accumulation results in carry-save format, one output register, and a register to store the accumulated carry bits. It also contains a carry-lookahead adder, a carry-save adder, and an incrementer. This makes this a large design in terms of area usage. Due to the large number of components, it is expected that the energy usage is also high.

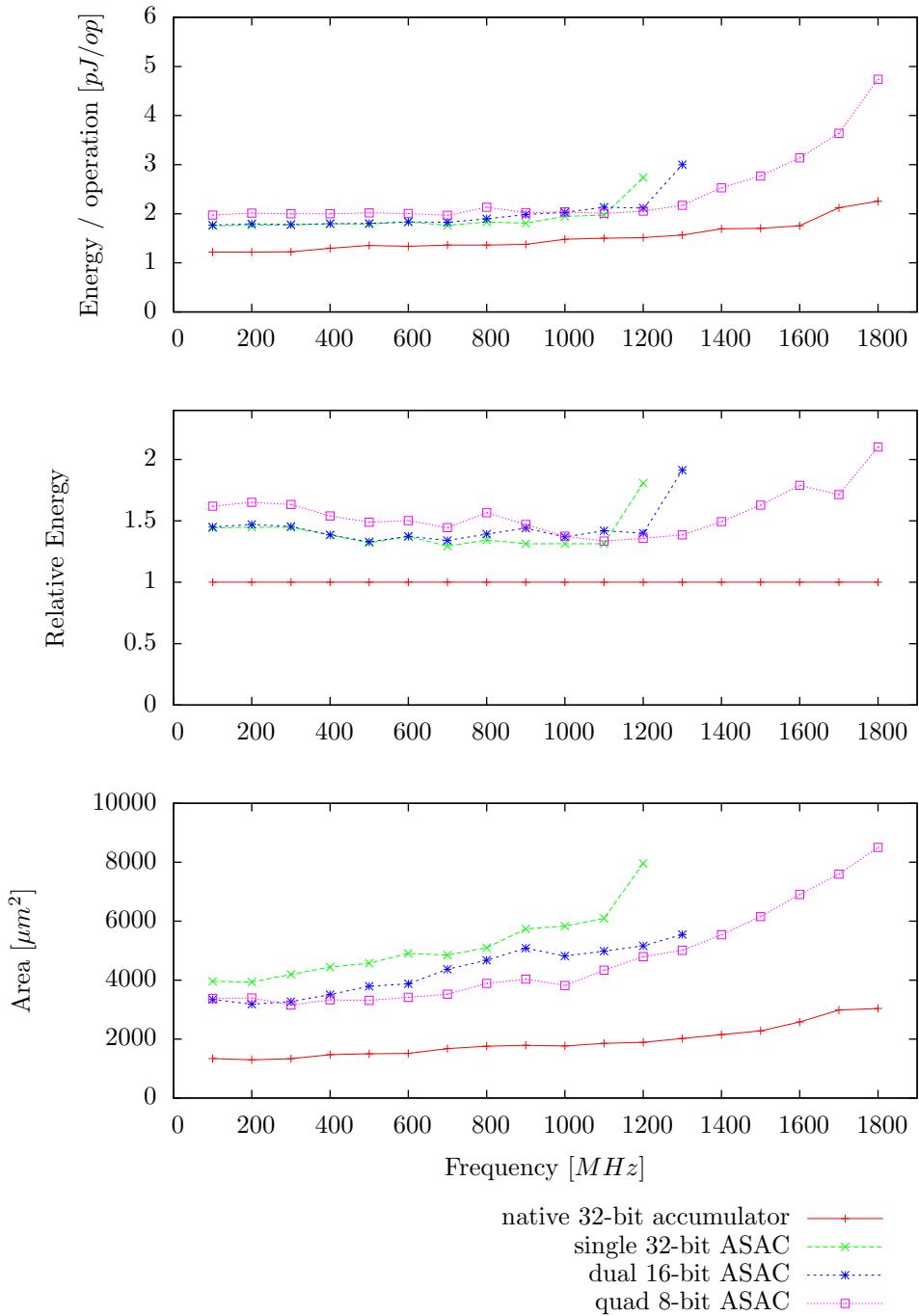
After benchmarking, Plot 6.7 confirms these assumptions. The lowest energy overhead compared to the baseline is 30%, and large parts of all three configuration are 40–60% worse than the baseline.

Also the area usage of this design is huge: the smallest configuration, quad 8-bit, uses  $3157 \mu\text{m}^2$  of area at 300 MHz, almost 2.5 times the area of the baseline, which uses  $1334 \mu\text{m}^2$ . The largest configuration, single 32-bit, uses  $4193 \mu\text{m}^2$ , 3.1 times the baseline.

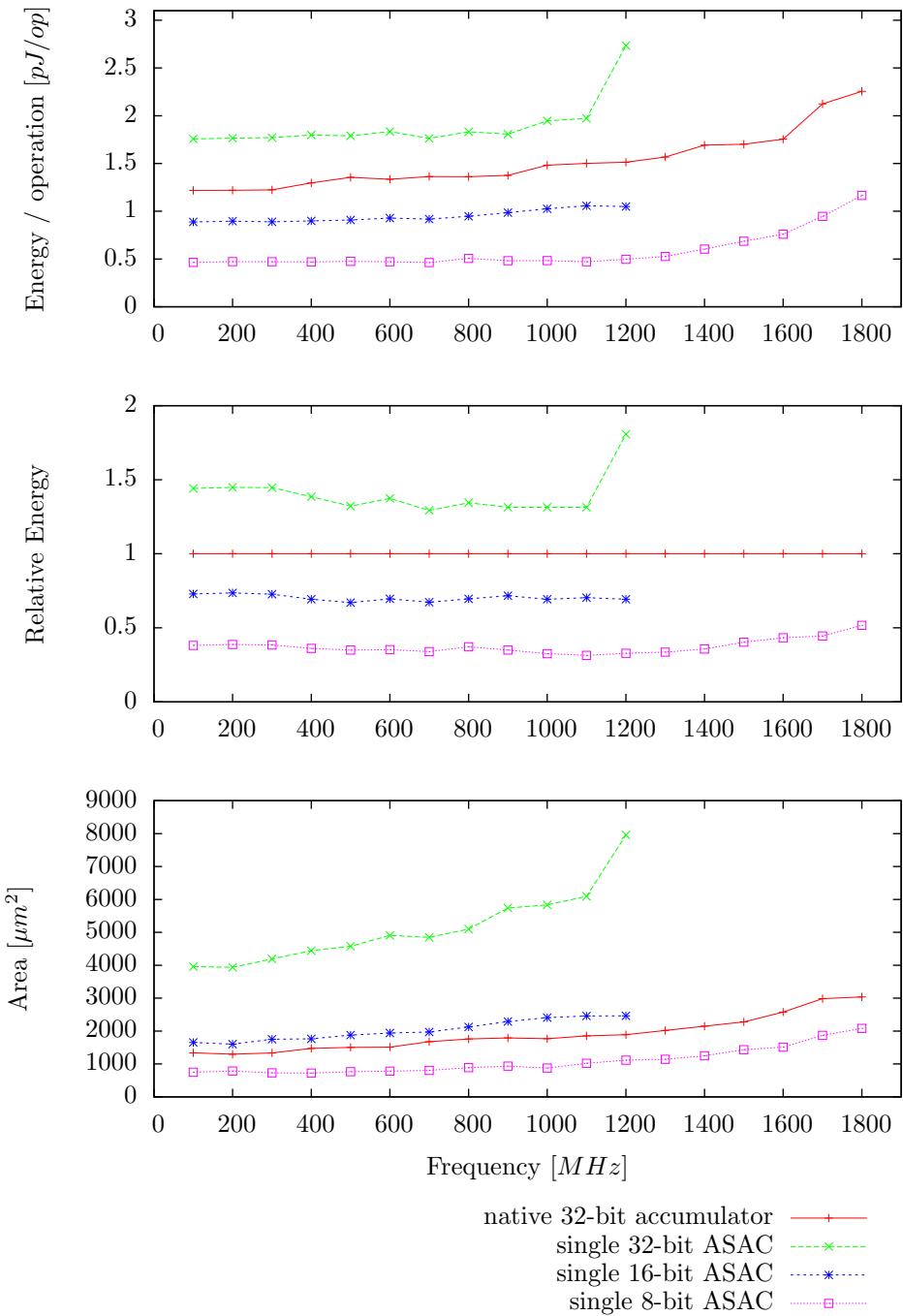
When looking at 8-bit accumulation in Plot 6.8, we see that the 16-bit configuration performs only slightly better than the baseline; this configuration uses around 70% of the energy of the baseline. The 32-bit configuration uses 40% more energy, and the 8-bit functional unit still uses about 40% of the energy of the baseline. Looking at area, we see that only the 8-bit configuration is smaller than the 32-bit native baseline. The 16-bit functional unit is slightly larger, and the 32-bit functional unit was, as discussed before, more than three times as large.

This design has the advantage that the functional units that are combined to form a larger addition do not have to be close together, as there is no carry chain needed between these units. However, the carry-save adder was supposed to save energy, but the overhead from the extra registers was too much in this design.

On the other hand, this design was made using a very simple functional unit, only containing the needed modules and registers. If, for example, each functional unit should also be able to work as a register file, then the required registers for this accumulator would already be available, and the extra costs of this design would be reduced.



Plot 6.7: 32-bit accumulation on a carry-accumulate composition with a carry-save accumulator (ASAC).



Plot 6.8: 8-bit accumulation on a carry-accumulate composition with carry-save accumulator (ASAC).

## 6.4 Comparison

We have seen two variants for the internal adder, and two techniques to make a larger accumulator based on smaller functional units, for a total of four designs — ripple-carry composition with carry-lookahead base (RLAC), ripple-carry composition with carry-save base (RSAC), carry-accumulate composition with carry-lookahead base (ALAC), and carry-accumulate composition with carry-save base (ASAC). These designs were discussed in the previous section, and compared to a 32-bit native Verilog accumulator, implemented using the “+” operator and an internal accumulate register. Just like we did for addition in Section 5.4, we will compare these designs against each other, and for reference again to the baseline, in order to determine which design has the least overhead when performing 32-bit accumulation, and which design has the highest gain during 8-bit accumulation.

Plots 6.9 and 6.10 show the performance of 32-bit wide accumulation, using 8-bit and 16-bit functional units respectively.

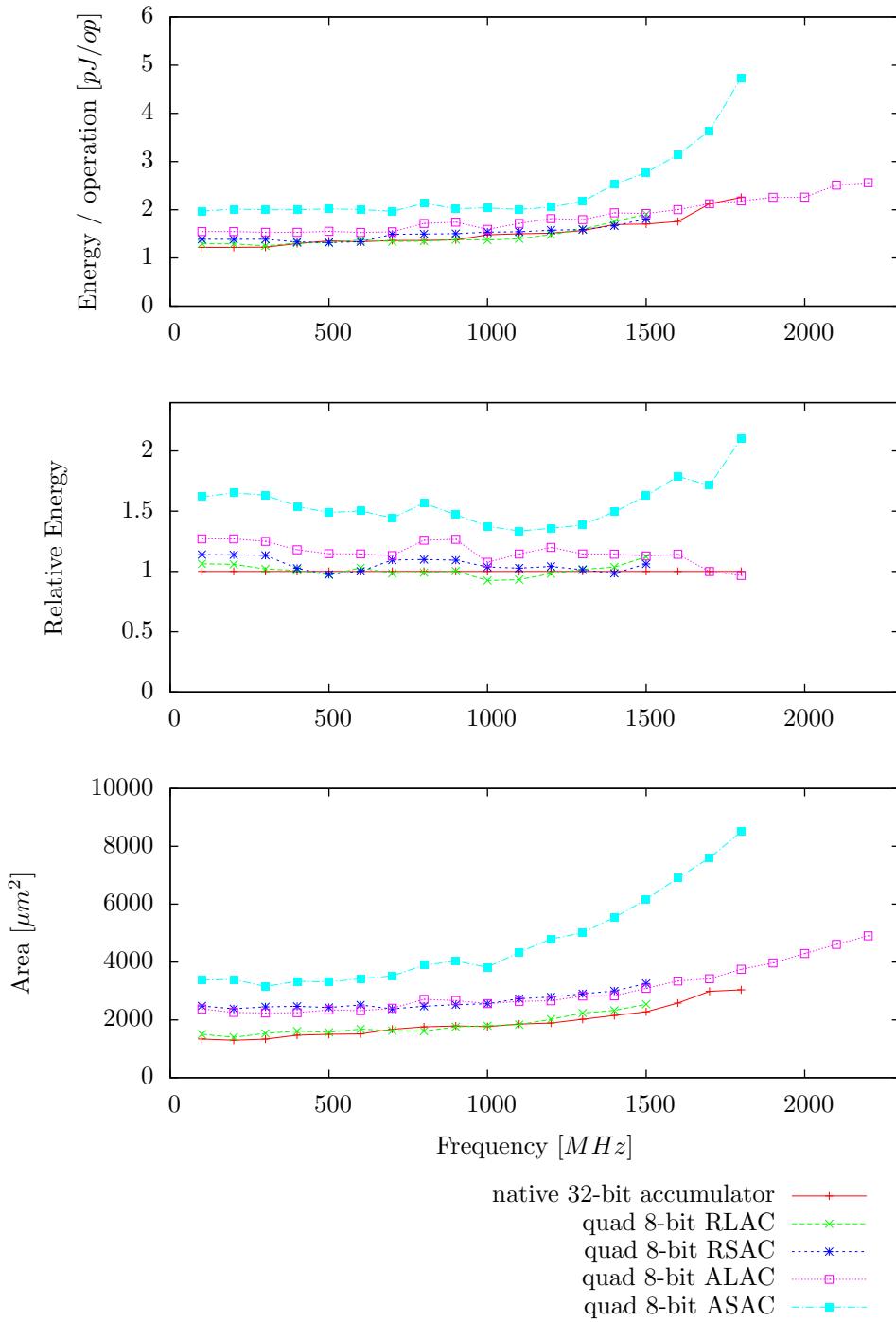
For the 8-bit configurations, we see that both designs based on the ripple composition have similar energy efficiency as the baseline; less than 5% overhead for the RLAC and less than 15% overhead for the RSAC design. On the contrary, the two designs based on the carry-accumulator use substantially more energy: 15–25% more for the ALAC design, and 35–70% more for the ASAC design. Regarding area: the RLAC design uses the same amount of area as the baseline, while the RSAC and ALAC designs use up to twice as much. The ASAC design uses the most area: around three times as much as the baseline.

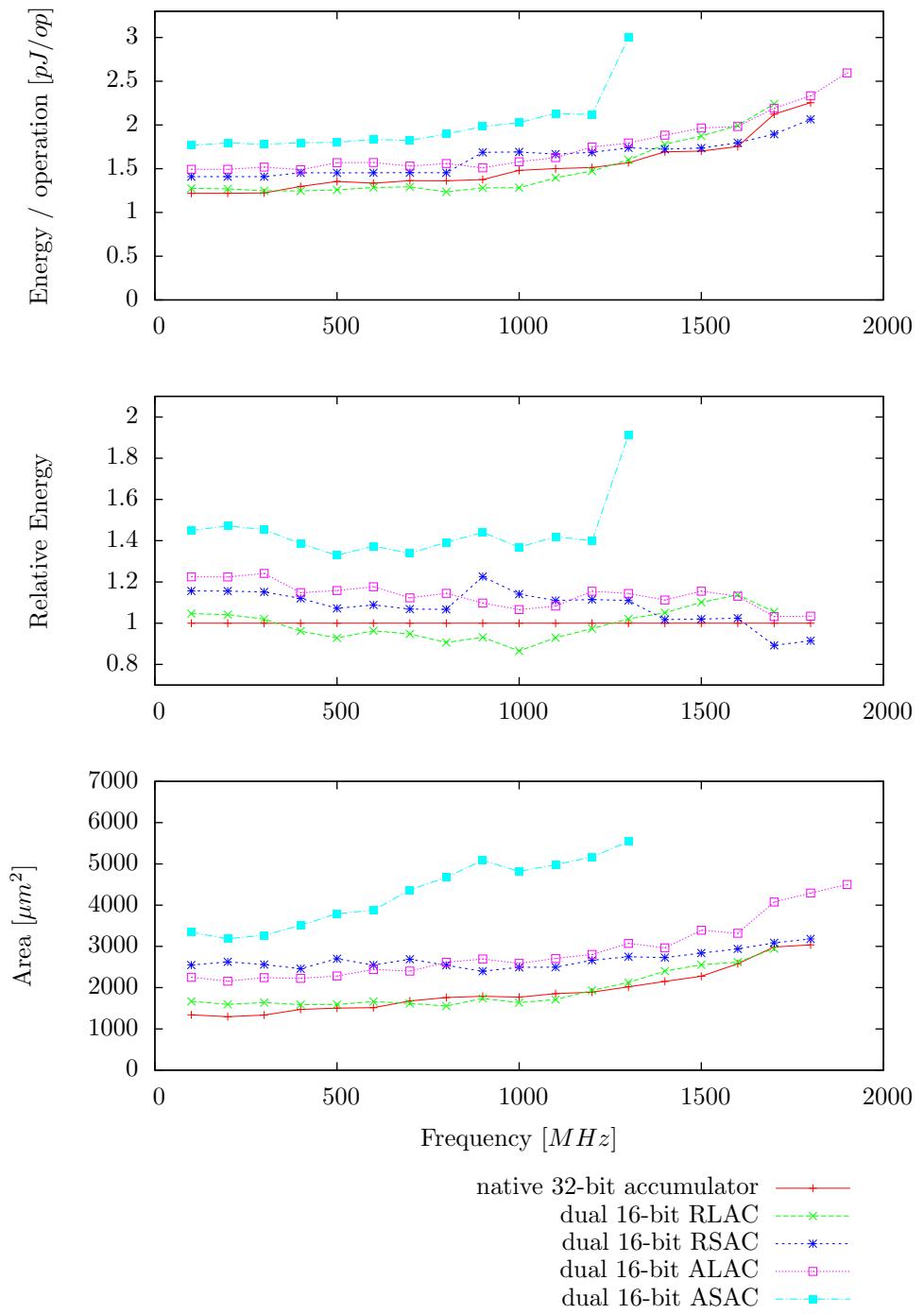
The 16-bit configurations show a similar result: the RLAC is still the best design, with a similar energy and area efficiency as the baseline; and the ASAC design has the worst performance, with 40% energy overhead and twice the area of the baseline. The other two designs, RSAC and ALAC, are now closer together, with about 10–20% energy overhead and 50% area overhead.

Plots 6.11 and 6.12 show how much performance can be gained when 8-bit accumulations are performed on 8-bit and 16-bit functional units respectively.

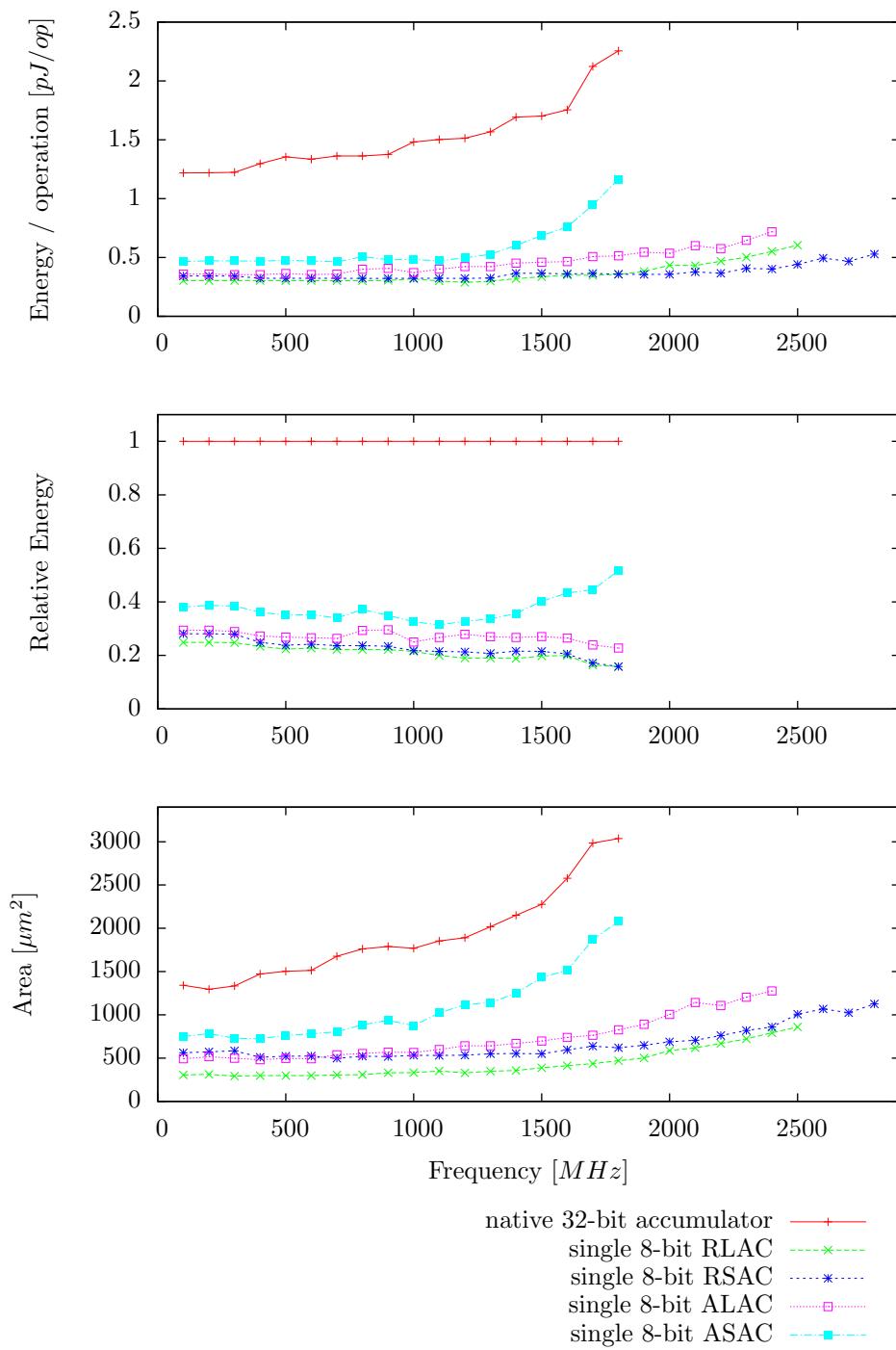
When 8-bit functional units are used, the RLAC design again has the best energy and area properties: the energy usage for these 8-bit additions is down to 25–15% compared to the baseline, while using less than 20% of the area. The RSAC design has a slightly worse performance, both in terms of energy and area, although it is able to operate at higher frequencies. The ALAC design uses 20–30% of the energy of the baseline, while the ASAC still uses 35–45% of the energy.

For 16-bit functional units, the results are again similar, although the numbers are higher: RLAC is still the most efficient design, using 40–50% energy compared to the baseline. The RSAC design is second, closely followed by the ALAC design, and the ASAC design again has the worst performance, using around 75% of the energy of the baseline.

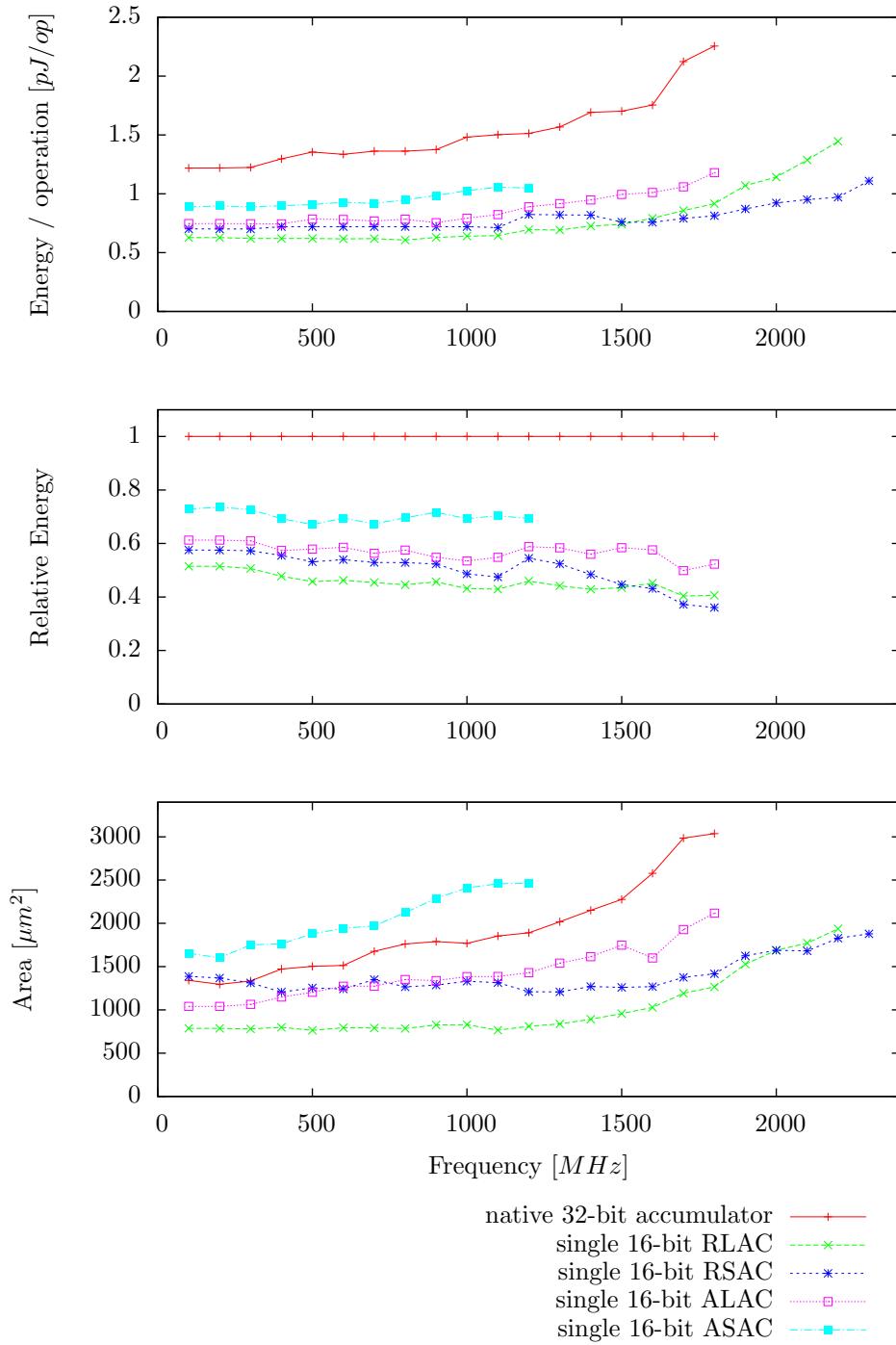




Plot 6.10: 32-bit accumulation using 16-bit functional units.



Plot 6.11: 8-bit accumulation using 8-bit functional units.



Plot 6.12: 8 bit accumulation using 16-bit functional units.

## 6.5 Conclusions

Based on the results from the previous two sections, we can draw the following conclusions.

The two designs with the ripple-carry interconnect both perform quite well, while the designs with the carry-accumulator composition are (very) expensive in both energy and area. Unless the added flexibility from the carry-accumulator is needed, this is a bad design. If one does need to use a design with a carry-accumulator, then it is preferred to be used in combination with the carry-lookahead adder.

When the ripple-carry interconnect is used, then the best adder algorithm depends on the granularity and operating frequency. When high frequencies are used, especially with the 32-bit functional units, then the ripple-carry composition with carry-save base accumulator performs the best, and is even preferred over the baseline accumulator. When multi-granular designs are used with functional units of 8-bit or 16-bit, then the ripple-carry composition with carry-lookahead base accumulator preforms better, although the difference is sometimes small.

The decomposition into smaller, multi-granular, functional units does not have much effect on the energy usage for these accumulators. Only the single 32-bit ripple-carry composition with carry-save base accumulator is significantly more energy efficient than the other two configurations. The two designs with carry-accumulate composition even profit from the multi-granular decomposition, as the smaller blocks have a lower delay, and thus higher frequencies can be achieved.

In conclusion, the ripple-carry interconnect is by far the best composition algorithm for accumulation, and inside the functional units the carry-lookahead generally performs slightly better, although the difference with carry-save is not very large.

# Chapter 7

## Multiplication

The multiplication operation multiplies two operands,  $a$  and  $b$ , to form the product  $p = a \times b$ . We will consider only multiplications where the two operands have the same width, but the discussed algorithms also work for operands of different sizes. The product  $p$  can be both full-width (having a width equal to the sum of the widths of both operands), or half-width, which is the width of the largest factor.

### 7.1 Multiplication Algorithms

Several algorithms have been developed for binary multiplication; most notable are the Wallace tree [28], the Dadda multiplier [7]. A more recently developed algorithm is the *High Performance Multiplier* or HPM algorithm by Eriksson et al. [9], which is based on the Dadda multiplier, but has a more regular structure which makes the place and route phase easier and more efficient.

Because of its energy efficiency properties, we will use the HPM algorithm in the investigations in this chapter. Most results in this chapter generalise to related algorithms such as the Wallace or Dadda multipliers.

When  $A$  and  $B$  respectively are  $n$ -bit and  $m$ -bit numbers in the usual binary encoding — that is, when  $A = \sum_{i=0}^{n-1} (a_i \ll i)$  and  $B = \sum_{i=0}^{m-1} (b_i \ll i)$  — then their product  $A \times B$  consists of the sum  $\sum_{i=0}^{n-1} \sum_{j=0}^{m-1} ((a_i \times b_j) \ll (i + j))$ . The HPM multiplier, as well as simpler variants such as the Wallace and Dadda multipliers, perform a multiplication by first computing these *partial products*  $pp_{i,j} = (a_i \times b_j) \ll (i + j)$ , and then adding all these partial products using some form of adder network. This conceptual procedure is illustrated in Figure 7.1.

To implement this procedure, the HPM multiplier consists of three components: a *partial product generator*, a *reduction tree*, and a *final adder*. The partial product generator is the component that computes the  $n \times m$  partial product

$y_7$	$y_6$	$y_5$	$y_4$	$y_3$	$y_2$	$y_1$	$y_0$	$\times$							
$x_7$	$x_6$	$x_5$	$x_4$	$x_3$	$x_2$	$x_1$	$x_0$								
$p_{7,0}$	$p_{6,0}$	$p_{5,0}$	$p_{4,0}$	$p_{3,0}$	$p_{2,0}$	$p_{1,0}$	$p_{0,0}$								
$p_{7,1}$	$p_{6,1}$	$p_{5,1}$	$p_{4,1}$	$p_{3,1}$	$p_{2,1}$	$p_{1,1}$	$p_{0,1}$								
$p_{7,2}$	$p_{6,2}$	$p_{5,2}$	$p_{4,2}$	$p_{3,2}$	$p_{2,2}$	$p_{1,2}$	$p_{0,2}$								
$p_{7,3}$	$p_{6,3}$	$p_{5,3}$	$p_{4,3}$	$p_{3,3}$	$p_{2,3}$	$p_{1,3}$	$p_{0,3}$								
$p_{7,4}$	$p_{6,4}$	$p_{5,4}$	$p_{4,4}$	$p_{3,4}$	$p_{2,4}$	$p_{1,4}$	$p_{0,4}$								
$p_{7,5}$	$p_{6,5}$	$p_{5,5}$	$p_{4,5}$	$p_{3,5}$	$p_{2,5}$	$p_{1,5}$	$p_{0,5}$								
$p_{7,6}$	$p_{6,6}$	$p_{5,6}$	$p_{4,6}$	$p_{3,6}$	$p_{2,6}$	$p_{1,6}$	$p_{0,6}$								
$p_{7,7}$	$p_{6,7}$	$p_{5,7}$	$p_{4,7}$	$p_{3,7}$	$p_{2,7}$	$p_{1,7}$	$p_{0,7}$	+							
$s_{15}$	$s_{14}$	$s_{13}$	$s_{12}$	$s_{11}$	$s_{10}$	$s_9$	$s_8$	$s_7$	$s_6$	$s_5$	$s_4$	$s_3$	$s_2$	$s_1$	$s_0$

Figure 7.1: Conceptual breakdown of a multiplication. Each combination of input bits  $x_i$  and  $y_j$  yields a partial product bit  $p_{i,j} = \text{AND}(x_i, y_j)$ , and these partial products are added together into the final product.

bits involved in a multiplication of  $n$ -bit and  $m$ -bit numbers. For unsigned multiplication<sup>1</sup>, the partial product generator uses an array of AND-gates to compute the partial product bits, computed as  $pp_{i,j} = \text{AND}(a_i, b_j)$ .

The HPM multiplier's *reduction tree* implements a carry-save adder, described in Section 6.1.2, to add the  $n \times m$  partial product bits; the output of this reduction tree consists of the complete sum in carry-save format. The reduction tree of the HPM multiplier is structured in such a way that it forms a regular pattern, which uses only  $\mathcal{O}(\log(n+m))$  time for the reduction step of a multiplication of  $n$ -bit and  $m$ -bit numbers.

The reduction tree produces the result of the multiplication in carry-save format; to convert this into usable form, a regular addition is necessary as a final step of the multiplication procedure. Any adder algorithm can be used as this final adder; as discussed in Chapter 5, in the context of this report, we will use a carry-lookahead adder for this purpose, as this adder is fast and energy efficient.

### 7.1.1 Signed Multiplication

The HPM multiplier can perform signed multiplications by following the Baugh–Wooley algorithm [1] or Booth's recoding [3] for two's complement multiplication. Both variants are discussed in detail by Själander [22], and the HPM tree based on the Baugh–Wooley algorithm is the most energy efficient algorithm. For an  $n$ -bit multiplier, this means that the most significant bit of the first  $n-1$  partial product rows, and all bits except the most significant bit of the last

---

<sup>1</sup>We will discuss signed multiplication in Section 7.1.1.

partial product are inverted. Additionally, a 1 is added to the  $n$ -th column, and to the  $2n - 1$ -th column, as illustrated in Figure 7.2.

$$\begin{array}{r}
 & y_7 & y_6 & y_5 & y_4 & y_3 & y_2 & y_1 & y_0 \\
 & x_7 & x_6 & x_5 & x_4 & x_3 & x_2 & x_1 & x_0 \times \\
 \hline
 \mathbf{1} & \overline{p_{7,0}} & p_{6,0} & p_{5,0} & p_{4,0} & p_{3,0} & p_{2,0} & p_{1,0} & p_{0,0} \\
 & \overline{p_{7,1}} & p_{6,1} & p_{5,1} & p_{4,1} & p_{3,1} & p_{2,1} & p_{1,1} & p_{0,1} \\
 & \overline{p_{7,2}} & p_{6,2} & p_{5,2} & p_{4,2} & p_{3,2} & p_{2,2} & p_{1,2} & p_{0,2} \\
 & \overline{p_{7,3}} & p_{6,3} & p_{5,3} & p_{4,3} & p_{3,3} & p_{2,3} & p_{1,3} & p_{0,3} \\
 & \overline{p_{7,4}} & p_{6,4} & p_{5,4} & p_{4,4} & p_{3,4} & p_{2,4} & p_{1,4} & p_{0,4} \\
 & \overline{p_{7,5}} & p_{6,5} & p_{5,5} & p_{4,5} & p_{3,5} & p_{2,5} & p_{1,5} & p_{0,5} \\
 & \overline{p_{7,6}} & p_{6,6} & p_{5,6} & p_{4,6} & p_{3,6} & p_{2,6} & p_{1,6} & p_{0,6} \\
 \mathbf{1} & p_{7,7} & \overline{p_{6,7}} & \overline{p_{5,7}} & \overline{p_{4,7}} & \overline{p_{3,7}} & \overline{p_{2,7}} & \overline{p_{1,7}} & \overline{p_{0,7}} \\
 \hline
 s_{15} & s_{14} & s_{13} & s_{12} & s_{11} & s_{10} & s_9 & s_8 & s_7 & s_6 & s_5 & s_4 & s_3 & s_2 & s_1 & s_0
 \end{array} +$$

Figure 7.2: Illustration of the partial products for an 8-bit two's complement multiplication.

### 7.1.2 Output Formats

The result of a multiplication with an  $n$ -bit input and an  $m$ -bit input is  $n + m$ -bit wide, and this is called their full-width output. However, often the result is truncated to  $\max(n, m)$ -bits, especially in the special case where  $n = m$ . So with this half-width output, the output of the multiplication has the same width as the input. We will look at both half-width and full-width multiplication.

For a full-width output, there are again two options to output the results from the functional unit to the interconnect network. As the interconnect network is only  $n$  bits wide, the  $2n$ -wide result cannot be outputted at once over a single output port. So either two output ports are used such that the result can be written to the interconnect network in a single cycle, which we call the *single cycle* multiplier design; or the output is written to the network in two cycles using a single output port, which is called a *dual cycle* multiplier design.

The final addition step of a multiplication adds two terms that together form the result product in carry-save format. A multiplication unit could output its result in this carry-save format, skipping the final addition; if the interconnect network is fast enough, then this can be used to add the results of the multiplication to other results, using carry-save adders. However this does require twice as many output ports, i.e. two or four ports, depending on whether full-width output is used. As this option depends heavily on the interconnect, and this part of the design is not yet known, this design variation is considered future work.

## 7.2 Multi-Granular Multiplication

We would like to make a multi-granular multiplier, i.e. a multiplier capable of computing multiplications in different bit-widths. As we did not want to support arbitrary widths for efficiency reasons, the widths will be a multiple of the block size  $w$ . So a full-width multiplier with inputs of width  $n \times w$  and  $m \times w$ , for arbitrary values of  $n$  and  $m$ , produces a result of width  $(n + m) \times w$ .

We split the inputs  $a$  and  $b$  into respectively  $n$  and  $m$  blocks of size  $w$ , formally defined in Equation 7.1. Now a multiplier of width  $w$  can compute the partial product  $p_{i,j} = a_i \times b_j$ , of which the result is  $2w$  bits wide.

$$\begin{aligned} a &= \sum_{i=0}^{n-1} a_i \ll iw \\ b &= \sum_{j=0}^{m-1} b_j \ll jw \end{aligned} \tag{7.1}$$

With these partial products, it is possible to compute wider multiplications by computing multiple of these partial products, shifting the results to the right place, and adding them together. Equation 7.2 shows how this works for a  $2w$  wide multiplication, and Figure 7.3 gives a graphical representation of this equation. Equation 7.3 generalises this for a multiplication of  $(n \times w)$ -bit and  $(m \times w)$ -bit terms.

$$\begin{aligned} a \times b &= (a_0 + (a_1 \ll w)) \times (b_0 + (b_1 \ll w)) \\ &= (a_0 \times b_0) + (a_1 \times b_0) \ll w + (a_0 \times b_1) \ll w + (a_1 \times b_1) \ll 2w \end{aligned} \tag{7.2}$$

$$\begin{aligned} a \times b &= \left( \sum_{i=0}^{n-1} (a_i \ll (i \times w)) \right) \times \left( \sum_{j=0}^{m-1} (b_j \ll (j \times w)) \right) \\ &= \sum_{i=0}^{n-1} \sum_{j=0}^{m-1} (a_i \times b_j) \ll ((i + j) \times w) \end{aligned} \tag{7.3}$$

Figure 7.4 gives an example of a multi-granular decomposition of a full-width unsigned multiplication. The inputs are both 8-bit, which are multiplied using 4-bit functional units to produce a 16-bit number. In this example we calculate  $90 \times 60 = 5400$ ; or in binary  $0101\ 1010 \times 0011\ 1100 = 0001\ 0101\ 0001\ 1000$ . To do this, first the four partial products are computed, then these numbers are shifted to the right position according to Equation 7.3, and then these partial products are added together to form the final result.

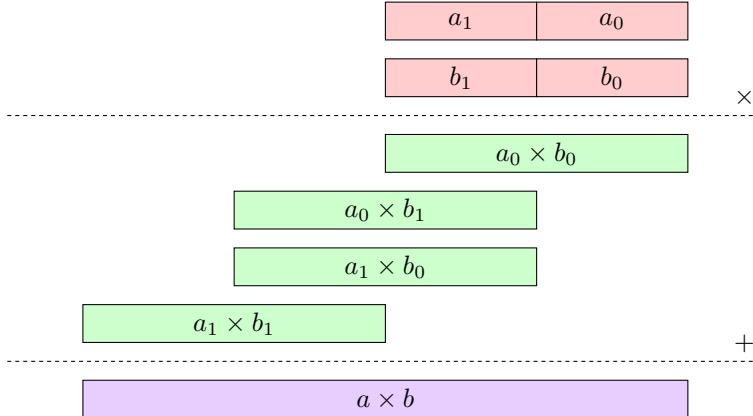


Figure 7.3: A  $2w$ -bit multiplication is decomposed into four  $w$ -bit multiplications, as described in Equation 7.2.

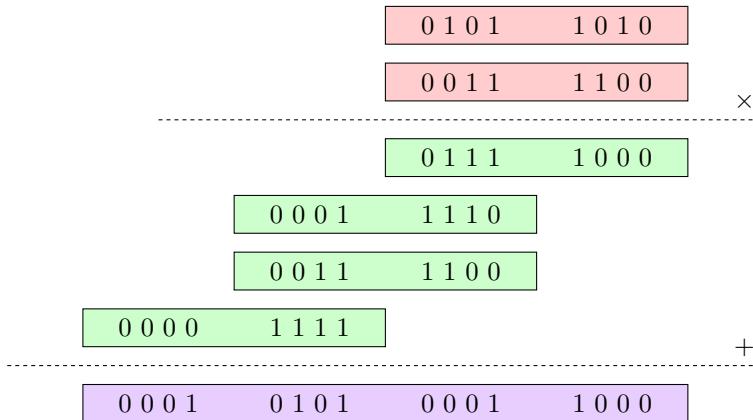


Figure 7.4: Example of an 8-bit full-width unsigned multiplication of  $90 \times 60$  decomposed into 4-bit multiplications.

### 7.2.1 Signed Multi-Granular Multiplication

The decomposition described above works for unsigned multiplication, as the wider multiplication can be decomposed into smaller computations that are themselves unsigned multiplications. However this is not the case for two's complement signed multiplication as shown in Figure 7.5: the most significant bits of each row, except the last, are negated, the bottom row, again except the most significant bit, is negated, and a one is added to columns  $w$  and  $2 \times w - 1$ , for a  $w$ -bit wide multiplication. Note that in Section 7.1.1, the most significant output bit was negated, and here we added 1 to the last column. This has the same result, but in this solution all adjustments to support signed operations are in the partial product array, which makes the decomposition for multi-granular operations easier.

$$\begin{array}{cccccccc|cccccccc}
& y_7 & y_6 & y_5 & y_4 & & y_3 & y_2 & y_1 & y_0 \\
& x_7 & x_6 & x_5 & x_4 & & x_3 & x_2 & x_1 & x_0 \\
\hline
\mathbf{1} & \overline{p_{7,0}} & p_{6,0} & p_{5,0} & p_{4,0} & | & p_{3,0} & p_{2,0} & p_{1,0} & p_{0,0} \\
& \overline{p_{7,1}} & p_{6,1} & p_{5,1} & p_{4,1} & | & p_{3,1} & p_{2,1} & p_{1,1} & p_{0,1} \\
& \overline{p_{7,2}} & p_{6,2} & p_{5,2} & p_{4,2} & | & p_{3,2} & p_{2,2} & p_{1,2} & p_{0,2} \\
& \overline{p_{7,3}} & p_{6,3} & p_{5,3} & p_{4,3} & | & p_{3,3} & p_{2,3} & p_{1,3} & p_{0,3} \\
& \overline{p_{7,4}} & p_{6,4} & p_{5,4} & p_{4,4} & | & p_{3,4} & p_{2,4} & p_{1,4} & p_{0,4} \\
& \overline{p_{7,5}} & p_{6,5} & p_{5,5} & p_{4,5} & | & p_{3,5} & p_{2,5} & p_{1,5} & p_{0,5} \\
& \overline{p_{7,6}} & p_{6,6} & p_{5,6} & p_{4,6} & | & p_{3,6} & p_{2,6} & p_{1,6} & p_{0,6} \\
\hline
\mathbf{1} & p_{7,7} & \overline{p_{6,7}} & \overline{p_{5,7}} & \overline{p_{4,7}} & | & \overline{p_{3,7}} & \overline{p_{2,7}} & \overline{p_{1,7}} & \overline{p_{0,7}} \\
s_{15} & s_{14} & s_{13} & s_{12} & s_{11} & | & s_{10} & s_9 & s_8 & s_7 & s_6 & s_5 & s_4 & s_3 & s_2 & s_1 & s_0
\end{array}
+$$

Figure 7.5: Illustration of the multi-granular decomposition for an 8-bit two's complement multiplication into four 4-bit multiplications.

The red lines show how this multiplication can be decomposed into four 4-bit wide multiplication. However, due to the negations and added ones, not all of these component multiplication are themselves either signed or unsigned multiplications. In general, we can identify four deviations from the plain unsigned multiplication: negate the bottom row, negate the most significant (left) row, add one to column  $w$ , and add one to column  $2w - 1$ . These four deviations occur in six different configurations: the left-top block, the left-bottom block, a general left (not top or bottom) block, a bottom (but not left) block, a center block (where nothing is negated or added), and a special case if only a single multiplier is used, and all deviations have to be applied. A summary of these deviations, and when they have to be applied, is given in Table 7.1.

It is also possible to perform unsigned multiplication with this multiplier, by disabling all these deviations. This is already happening for the multipliers that are not on the left or bottom row of a signed multi-granular multiplication, thus support for unsigned multiplications is always included.

### 7.2.2 Half-Width Multi-Granular Multiplication

Until now, we assumed the multiplication result was full-width, i.e. twice as wide as the inputs (or the sum of the input widths, in case they are asymmetric). If only half-width output is required, parts of the calculations can be skipped; Figure 7.6 shows this for a  $2w$ -bit multiplication. This half-width multiplication still requires three out of the four multipliers; the main performance gain is in the reduced number of adders needed to add the results of all these partial products.

<i>Block type</i>	Negate left row	Negate bottom row	Add 1 to column $w$	Add 1 to column $2w - 1$
Left-top	Yes	No	Yes	No
Left-middle	Yes	No	No	No
Left-bottom	Yes	Yes	No	Yes
Bottom	No	Yes	No	No
Middle	No	No	No	No
Single FU signed	Yes	Yes	Yes	Yes
Unsigned	No	No	No	No

Table 7.1: Deviations for the multipliers depending on their position in the multi-granular decomposition.

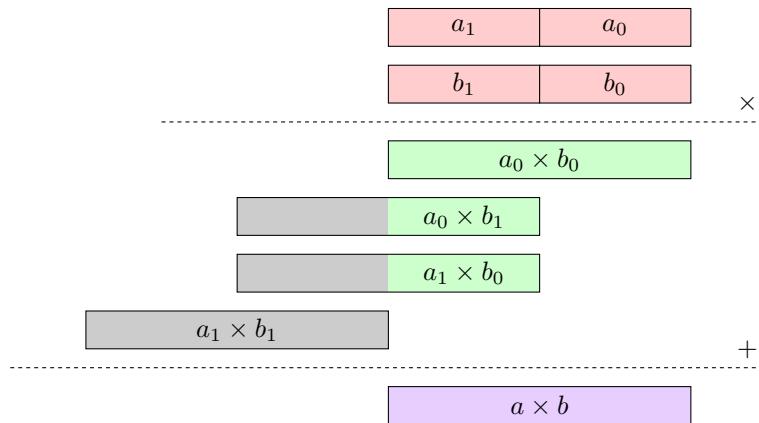


Figure 7.6: A  $2w$ -bit half-width multiplication is decomposed into three  $w$ -bit multiplications. The fourth  $w$ -bit multiplication cannot contribute to the half-width output.

### 7.2.3 Partial Product Addition

As described in the sections above, a multi-granular multiplication operation can be decomposed into several smaller multiplications, whose outputs have to be added into the final product. This final product consists of a concatenation of a number of  $w$ -bit wide words; moreover, each partial product consists of two concatenated  $w$ -bit wide words shifted a multiple of  $w$  bits. As a consequence, this addition can be decomposed into the addition of several  $w$ -bit terms for each  $w$ -bit word in the final product.

As a first approximation, then, the final-product addition consists of computing one independent sum of terms per output word. Specifically, when computing the final sum of an  $(nw \times nw)$ -wide multiplication, the sum for output word  $i$  consists of the sum of  $2i + 1$  words for  $i < n$ , or  $2(2n - i) - 1$  word for  $i \geq n$ . However, each of these sums may also produce several carry bits, which each have to be added into the sum of the next output word. As a consequence, the computation of these output-word sums are not fully independent, which complicates the addition process.

Depending on the type of multipliers used for computing the partial products, the partial products may become available in *dual-cycle* form. In this case, the multipliers output their result over the course of two subsequent cycles. Specifically, for the dual-cycle multiplier design, the most-significant word of the partial product becomes available to the interconnect network one cycle later than the least-significant partial product word. This logistical feature complicates the design of the partial product addition scheme.

We have developed two schemes for adding the partial products into the final product: the *adder tree* and *accumulator* designs. We describe these different designs in detail in the following two sections.

#### 7.2.4 Adder Tree

One possible way of adding all partial-product output words into a final product is to construct a conceptually-combinatorial adder network for each word in the final product. By adding two words into a single intermediate word whenever possible, one gets a binary tree with a logarithmic depth that ultimately results in a single output word, as illustrated in Figure 7.7. By furthermore adding delay units to this network as necessary, this results in a fully pipelined design, that has a throughput of one multiplication per cycle when single-cycle multipliers are used. When using dual-cycle multipliers, the output bandwidth of the multipliers limits this to one multiplication per two cycles.

How these delays are implemented depends on the architecture, interconnect and situation; sometimes it is possible to delay the functional units that provide the inputs, such that no delay is necessary. It could be that the functional units are able to produce the output a cycle later, or that the interconnect is capable of stalling the output for a cycle before passing it on to the next functional unit. If this is both not possible, a dedicated delay unit could be added to the network, consisting of a single register, that just outputs its input a cycle later. And finally, if all other options are unavailable, the intermediate result could be stored temporarily in a register file or another functional unit could be used to perform the identity operation, i.e. just output the input, just as the delay unit would do. In our benchmarks, we have used the delay unit to simulate delayed signals, but Appendix A provides some numbers of the amount of delay units used in all configurations, such that is it possible to perform a back-of-the-envelope calculation for power and energy numbers in case another delay method is used.

The multiplier design used to calculate the partial products has influence on the structure of the adder trees; if dual-cycle multipliers are used, then some of the partial products are one cycle later available than other partial products. This results in a adder tree that is slightly different than the adder tree used to accumulate the partial products from single-cycle multipliers.

Moreover, as a consequence that the dual-cycle multipliers use two cycles to perform the multiplication, the total multiplier now has a throughput of one multiplication per two cycles. Therefore, we can use the adder units for two

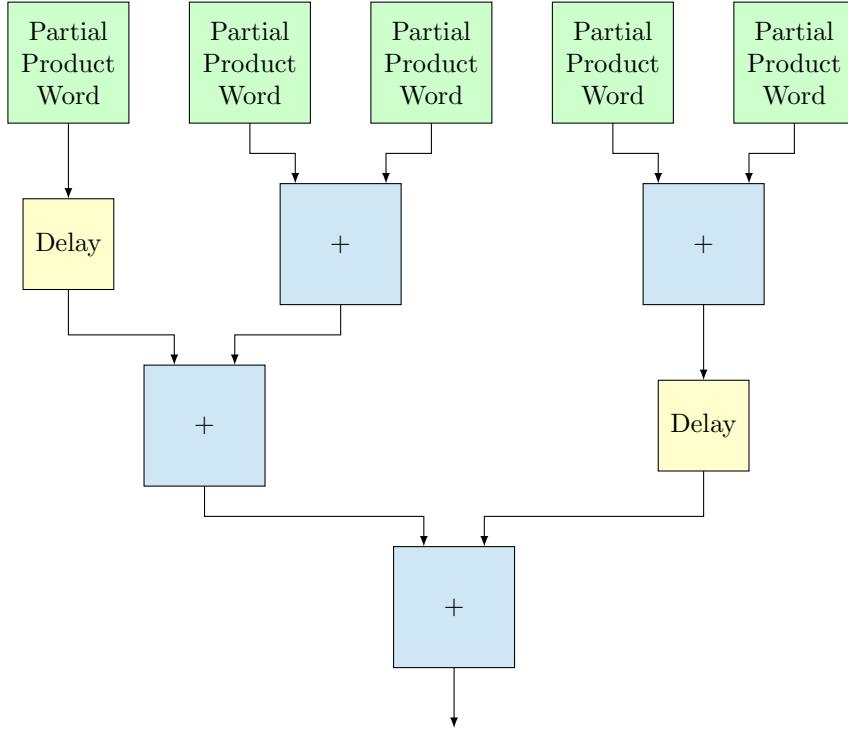


Figure 7.7: An adder tree takes  $\lceil \log_2 5 \rceil = 3$  cycles to add 5 partial product words.

consecutive cycles in an adder tree. This leads to a design that only uses half the number of adder units for the addition, and consequently, has to send less partial products through the interconnect. Figure 7.8 shows an example of such an adder tree, that accumulates the partial results of five dual-cycle multipliers, using two adder units; two of these partial results are available in the first cycle, and three partial products are produced in the second cycle.

As already mentioned in the previous section, we cannot consider each column individually, as these adders all produce a carry-out that has to be consumed by an adder in the next column. For the columns in lower half of the full-width output, this is no problem as each consecutive column requires more adders, however in the upper half of the output, the number of adders required decreases, and thus not enough adders are available to consume all the carry bits.

This can be resolved by adding more adders, such that all carry bits can be consumed. This solution is easy and requires no further adaptation to the functional units, but the disadvantage is of course that more functional units are required to form the adder tree; the number of adders in each cycle can never decrease. This means that the last column has the same amount of adders as the middle column.

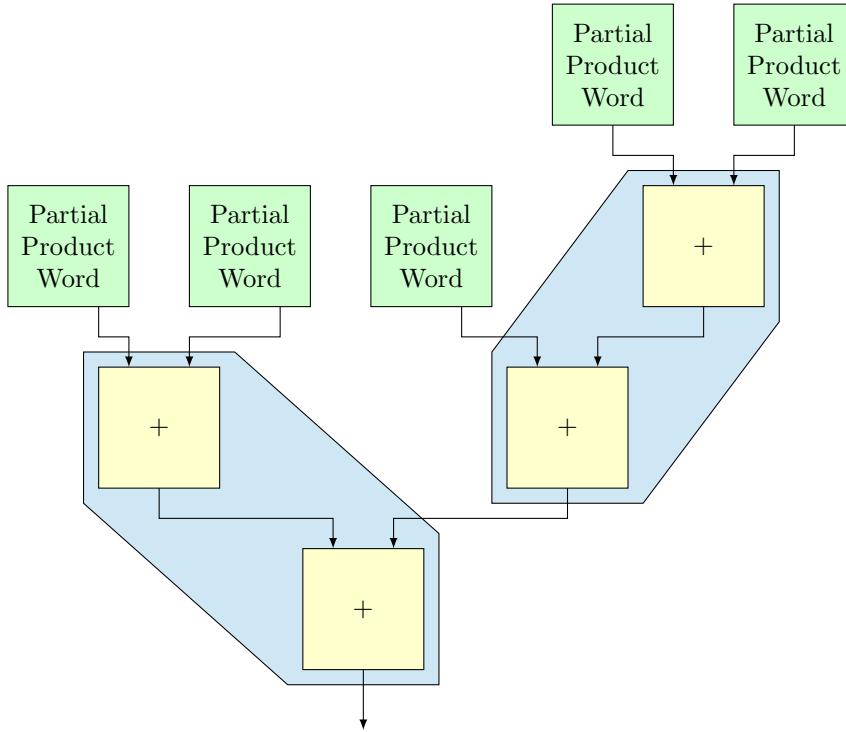


Figure 7.8: An adder tree uses only two adders for two cycles per multiplication to add 5 partial products generated by dual-cycle multipliers.

With some modifications, it is possible that adders can consume not one, but two carries. However, these adders also produce two carry bits if all inputs are high. So the carry chain between the functional units has to be two bits wide. Fortunately, adders that accept two pairs of carries still produce only two carry bits. If we extend the interconnect and functional units with these carry pairs, each cycle in the upper half of the adder tree now only requires at least one adder, thus reducing the number of adders required for multiplications. This is especially effective when wide full-width multiplications are frequently used; in particular, for multiplications of width two this adaptation does not change the number of functional units needed.

As these extra adders only have to be added in the upper half of the output, this only has effect when full-width multiplications are performed; half-width multiplications simply do not calculate these columns. As the dual carry-in adders require quite some changes in the interconnect and adder algorithm, and they only are advantageous for wide and full-width multiplications, we decided to simply use more adders to solve this problem, and leave this as future work.

Finally, Figure 7.9 shows an example of a full-width single-cycle adder tree. This multiplier requires four multiplier units, six adder units, and four delay units (of which two might be removed if the least significant bits can be written

directly to an output register). The six adders are combined into two  $3w$ -bit wide adders. This multiplier is fully pipelined, with a pipeline depth of three.

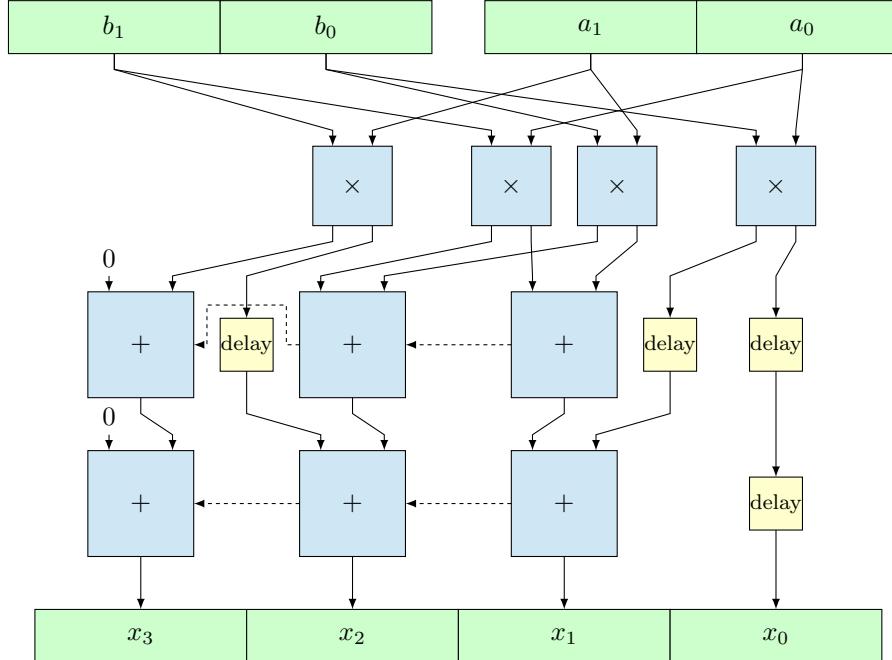


Figure 7.9: An example adder tree for  $2w \times 2w$  wide full-width single-cycle multiplication.

### 7.2.5 Accumulator

Another way to accumulate the partial products is by using an accumulator as discussed in Chapter 6, preferably the ripple-carry interconnect with carry-lookahead base accumulator, as this design generally has the best performance, but this does not matter for the concepts discussed in this section. In contrast with the adder tree, this design is not pipelined; it accumulates the partial results over the course of several cycles. An example of this design is given in Figure 7.10.

In this design, it is easy to connect the carry chain, as only a single accumulator is present for each column. Also no delays are needed, although it is possible to replace the least significant accumulator with a delay, as this accumulator “accumulates” only a single number and thus produces no carry out.

The partial products produced by the multipliers have to be routed to the right accumulator by the interconnect, i.e. the interconnect has to perform the shift operations from Formula 7.3. This requires an interconnect that is capable of runtime reconfiguration, as the outputs of the multipliers have to be routed to a different accumulator each cycle.

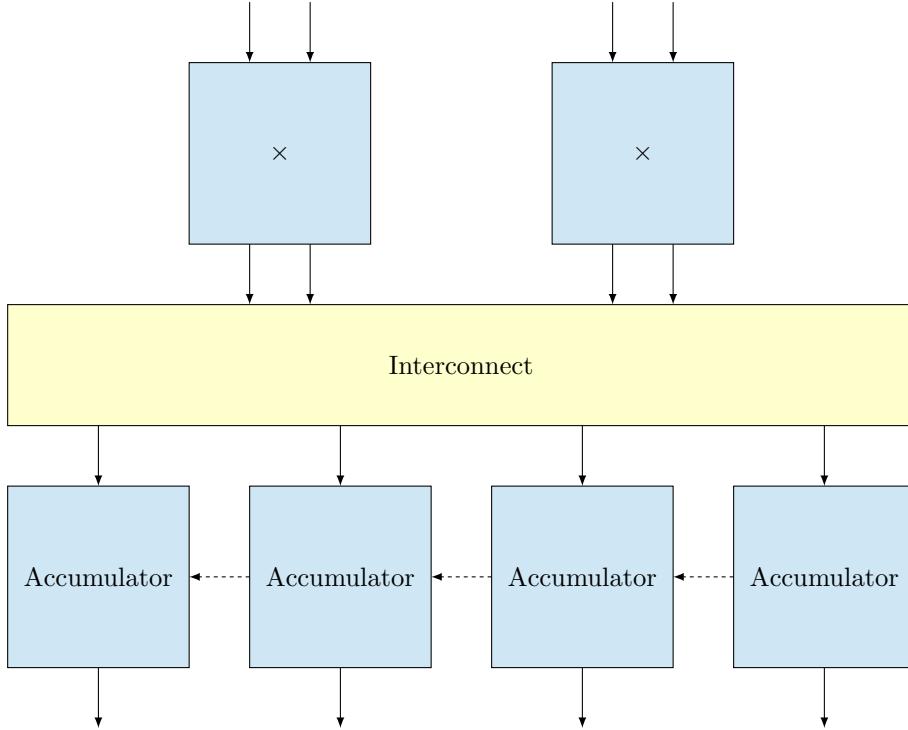


Figure 7.10: An example multiplier with accumulator for  $2w \times 2w$  wide full-width multiplication.

One might have noted that the examples in Figure 7.10 only contain two multipliers, while a  $2w \times 2w$  wide multiplication requires four multipliers. This is because the limiting factor in this design is the input bandwidth of the accumulators. Figure 7.11 shows the scheduling of the four partial multiplications on the two multipliers, taking three cycles. Only in the first cycle both multipliers are active; in the following two cycles only one multiplier is active, as both results have to be consumed by the same accumulators. Finding an optimal schedule can become somewhat complex for wide multiplications, but that is a task for the compiler and outside the scope of this report, as the compiler can also take other requirements into account, such as availability of the interconnect.

It is possible to increase the bandwidth of the accumulators by placing one or more adders in front of the accumulator input, basically creating a hybrid implementation between this design, and the previous adder tree design. Each layer of adders does increase the delay by one cycle, but doubles the available bandwidth to the accumulators, making this technique only viable for wide multiplications. Ideally, these extra adders are only placed in the middle of the output, as the highest bandwidth is required there, but as this technique also introduces the problem of the carry chain as discussed in the previous section, so the adder chain has to continue to the last column, making this technique most profitable on half-width multiplications.

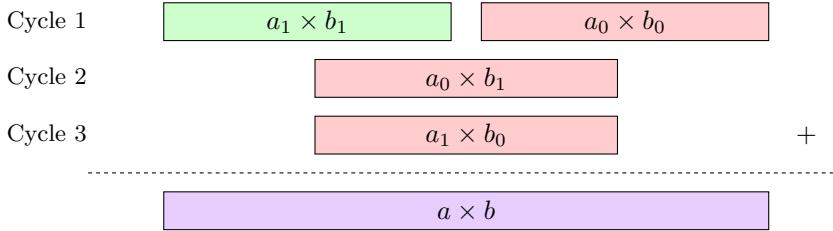


Figure 7.11: The time schedule used to add the partial products of a  $2w \times 2w$  wide full-width multiplication, generated by two single-cycle multipliers. The red partial products are generated by multiplier one, and the green partial product is generated by multiplier two.

## 7.3 Multi-Granular Multiplier Configurations

For multiplication, we have created two multiplier designs: a single-cycle multiplier and a dual-cycle multiplier. Additionally, we have two techniques to add the partial products: the adder tree and the accumulator. In the first four sections, we will discuss these four combinations, and in the last section, we will discuss both multiplier variants when used standalone; i.e. when no partial products have to be accumulated. In each section, we will look at the results of two variants of multiplication: full-width multiplication and half-width multiplication.

### 7.3.1 Single-Cycle Multiplier with Adder Tree

The multi-granular multiplier made from single-cycle multipliers and an adder tree is called the *Single-cycle Multiplier with adder Tree* (STM).

For full-width  $(n \times w)$ -wide multiplication on  $w$ -wide functional units, this design consists of  $n^2$  single-cycle multipliers and an adder tree consisting of  $3 \times (n^2 - n)$  adders. A half-width  $(n \times w)$ -wide multiplication consists of  $\frac{1}{2} \times (n^2 + n)$  single-cycle multipliers and an adder tree consisting of  $n^2 - n$  adders. The delay of both multipliers is the same:  $\lceil \log_2(2n - 1) \rceil + 1$ . Table 7.2 gives an overview of these formulas, and the delay and number of used multipliers and adders for 1-, 2-, 4- and 8-word wide multiplications.

The number of needed functional units increases quadratic with the size of the multiplications; making this design not very suitable for  $n > 4$ . However, with 8-bit functional units, this gives already  $4 \times 8 = 32$  bit wide multiplications, and with 16-bit functional units, multiplications up to 64-bit can be performed with reasonable numbers of functional units.

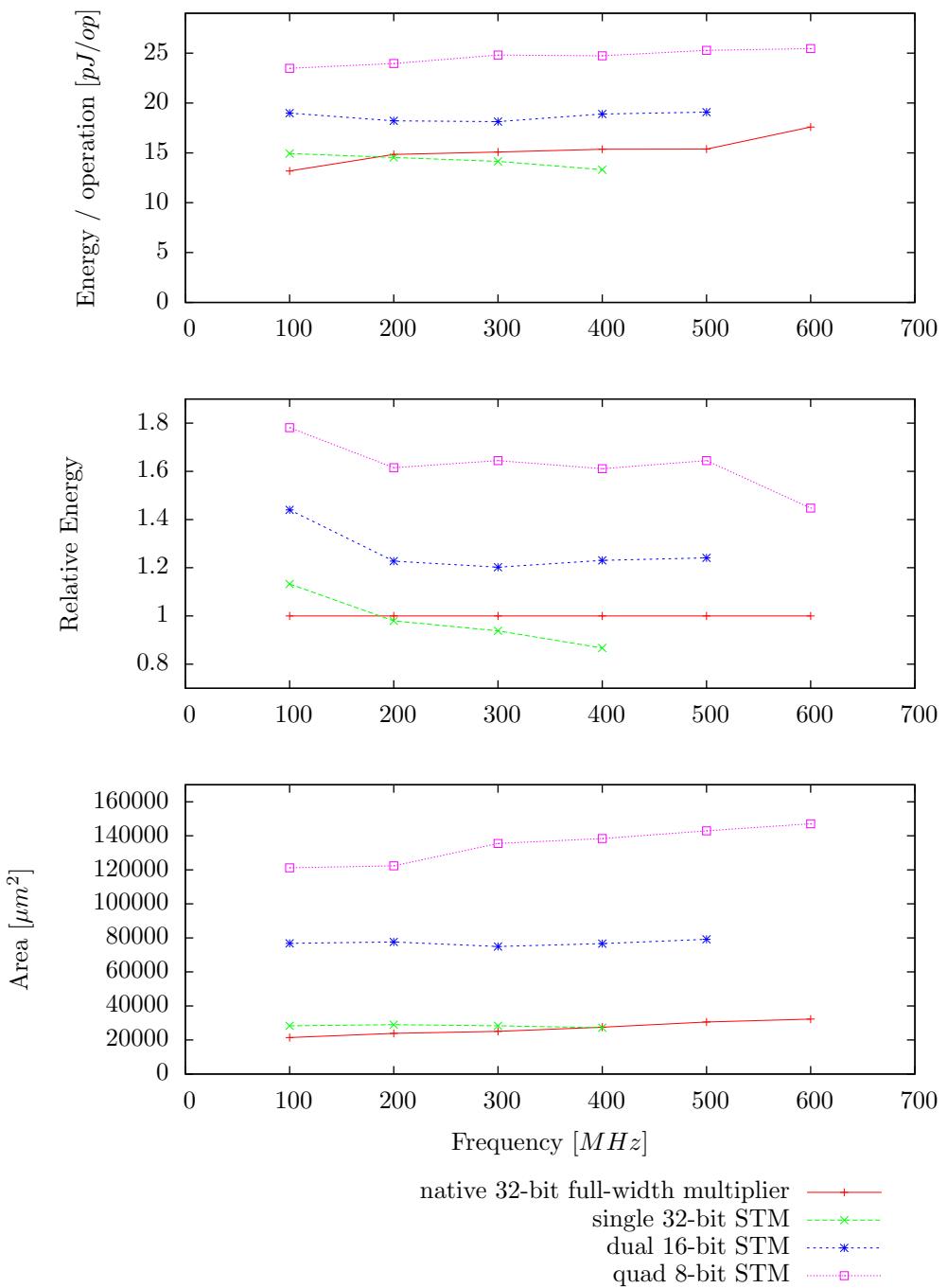
The benchmark results for full-width multiplication with the STM multiplier are depicted in Plot 7.1. These plots show that the single 32-bit wide multiplier performs quite similar to the native 32-bit full-width baseline multiplier, both in

terms of energy and area. When 10 16-bit functional units (4 multipliers and 6 adders) are used to form a single 32-bit multiplier, the energy usage is increased by 20%. Because each functional unit can perform both multiplications as well as additions, the required area is increases with a factor 3 or 4, depending on the operating frequency. When 8-bit functional units are used, 52 of these units (16 multipliers and 36 adders) have to be combined to perform a 32-bit wide multiplication, resulting in a 60% more energy usage, and 5 or 6 times the area requirements.

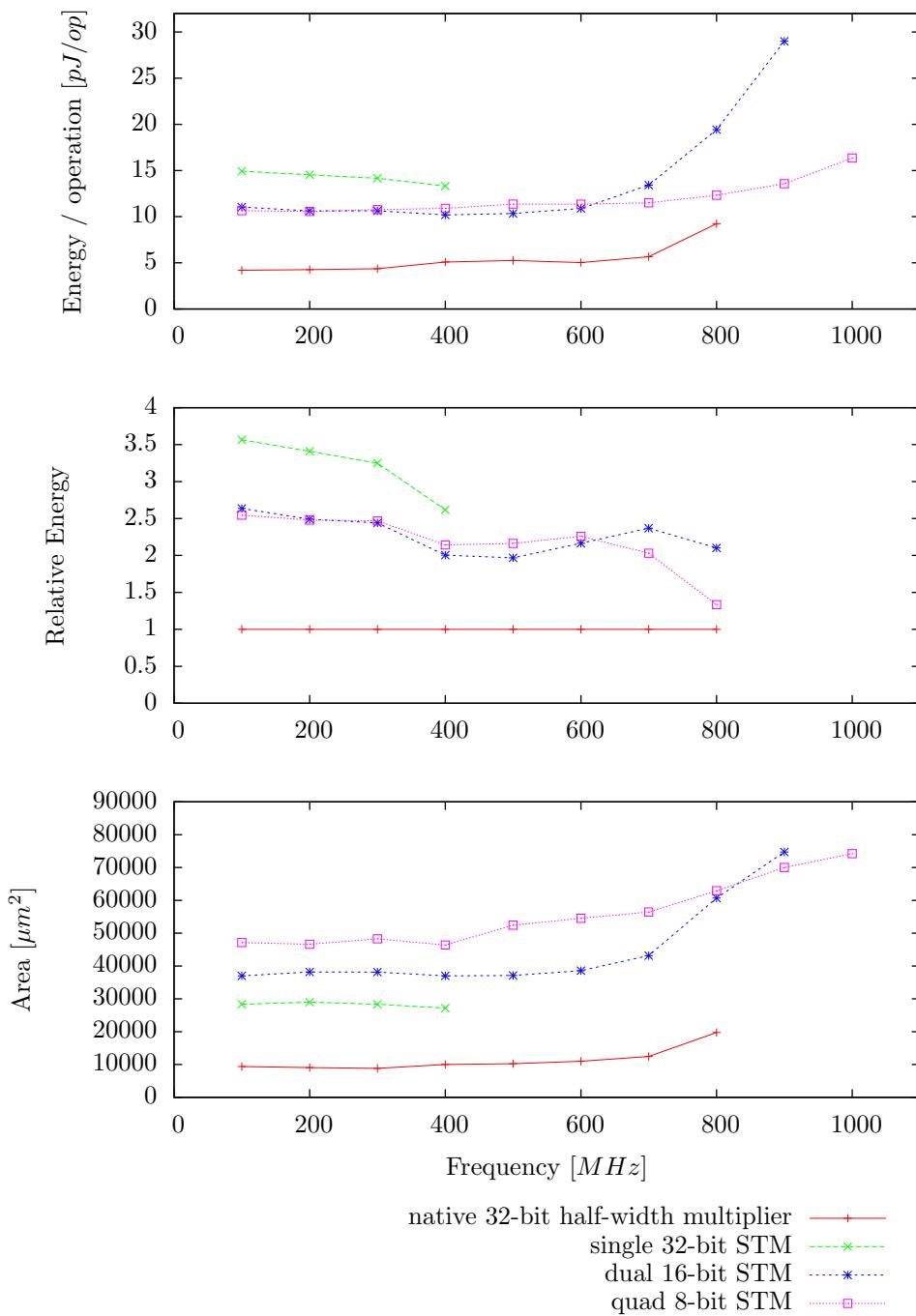
Plot 7.2 shows the results for half-width multiplication. As our functional units can only perform full-width multiplications, the single 32-bit configuration, which performed nearly the same as the full-width baseline, now uses 2.5–3.5x more energy per operation than the half-width baseline; even more than the configurations with 8- and 16-bit wide functional units. These configuration use 2–2.5x more energy than the baseline, but these designs are able to reach higher frequencies as these are pipelined, using 3 or 4 cycles. The area usage is 3–5 times higher than the baseline for these configurations.

width	full-width		half-width		delay
	multipliers	adders	multipliers	adders	
1	1	0	1	0	1
2	4	6	3	2	3
4	16	36	10	12	4
8	64	168	36	56	5
$n$	$n^2$	$3(n^2 - n)$	$\frac{1}{2}(n^2 + n)$	$n^2 - n$	$\lceil \log_2(2n - 1) \rceil + 1$

Table 7.2: Number of multipliers and adders used for STM configurations of several sizes, and their delay.



Plot 7.1: Single-cycle 32-bit full-width multiplication with adder tree.



Plot 7.2: Single-cycle 32-bit half-width multiplication with adder tree.

### 7.3.2 Dual-Cycle Multiplier with Adder Tree

A dual-cycle multiplier, combined with an adder tree, is called *Dual-cycle Multiplier with adder Tree* (DTM).

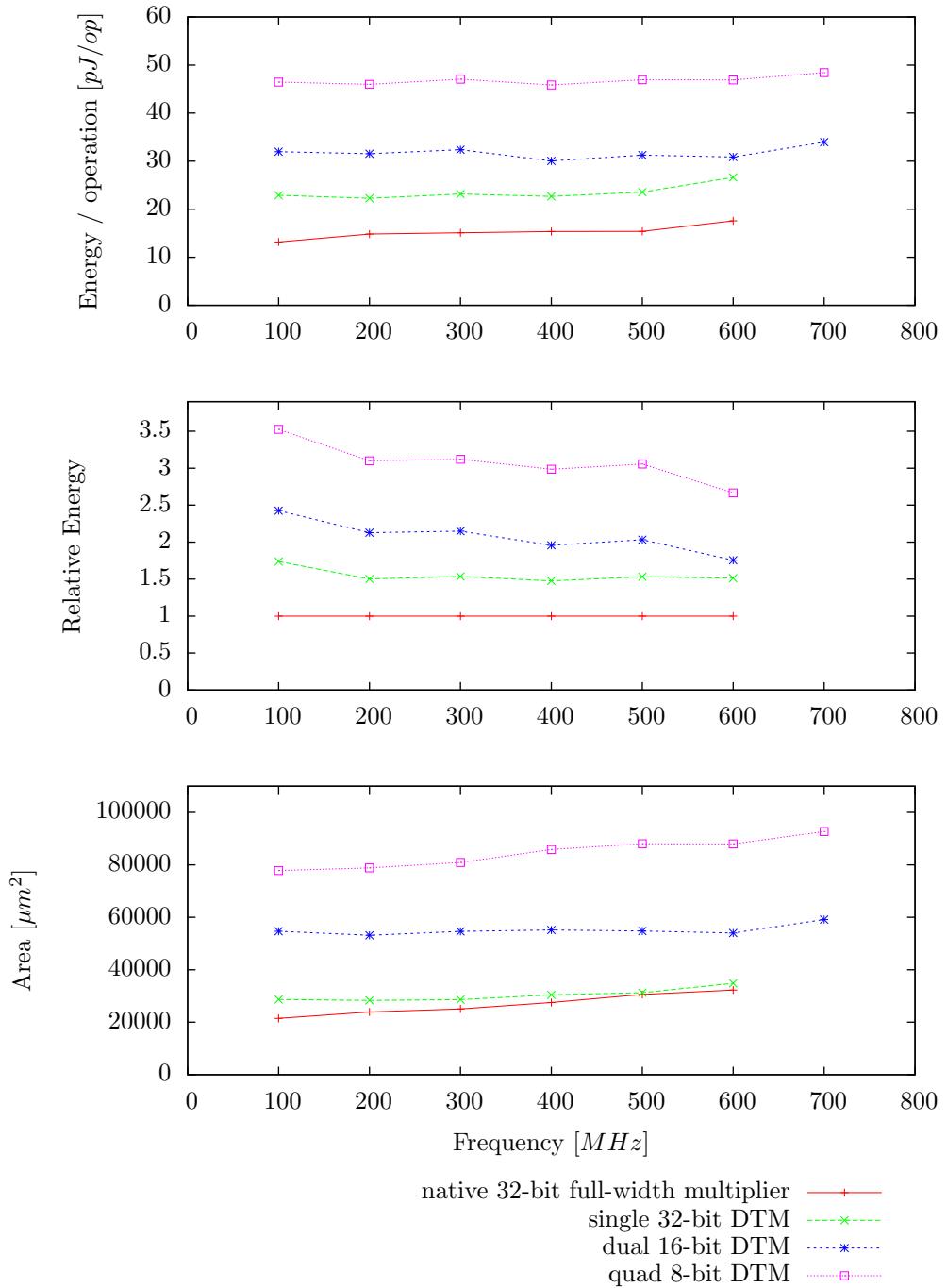
In this design, the multipliers use not one, but two cycles to produce the output. In the first cycle, the lower half of the output is produced, and the second cycle the upper half. This has implications for the adder tree design, as not all inputs are available at the same time, making the tree more irregular. However, the number of adders and multipliers is identical to the previous design.

Luckily the impact on the delay is limited; this design uses one cycle more than the design from the previous section. The impact is more severe on the throughput; where the previous design could produce one result each cycle, this design requires two cycles for a single multiplication.

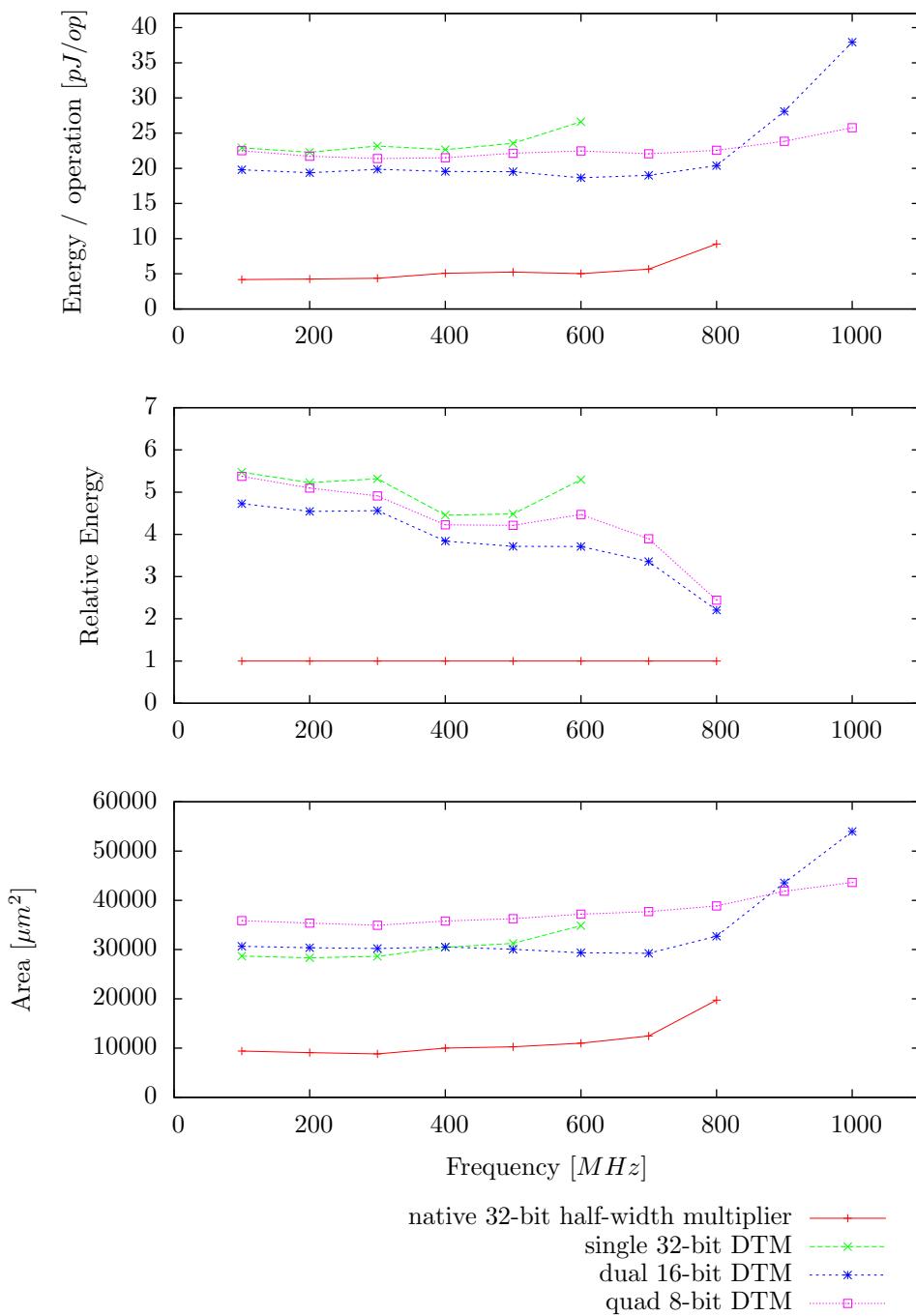
The benefit of this dual-cycle multiplier is that the interconnect can be simpler; only one port is required where the single-cycle variant would require dual-ported functional units, where both ports are individually routable.

We start with the full-width results in Plot 7.3. The single 32-bit configuration uses 1.5 times the energy of the baseline, while using slightly more area. The dual 16-bit configuration uses two times the energy and 2–2.5 times the area. When a 32-bit multiplier is constructed from 8-bit functional units, around 3 times the energy of the baseline is used, while 3–4 times the area is used. All energy per operation curves are almost flat, meaning that the designs scale quite well to higher frequencies, up to 700 MHz.

Plot 7.4 shows the benchmark results for half-width multiplications. Notable is that all three configurations perform nearly the same: 3.5–5.5 times the energy of the baseline, and use 3–4 times the area of the baseline. Also interesting is the order of the configurations: the single 32-bit configuration uses the most energy, closely followed by the quad 8-bit. The dual 16-bit configuration is the most energy efficient configuration. This is because these functional units can only perform full-width multiplications, and thus the single 32-bit configuration calculates the full-width result, and then just ignores the upper half, resulting in a higher energy usage.



Plot 7.3: Dual-cycle 32-bit full-width multiplication with adder tree.



Plot 7.4: Dual-cycle 32-bit half-width multiplication with adder tree.

### 7.3.3 Single-Cycle Multiplier with Accumulator

A single-cycle multiplier with an accumulator is called the *Single-cycle Multiplier with Accumulator* (SAM).

For an  $(n \times w)$ -wide multiplication on  $w$ -wide functional units, this design consists of several multipliers, and a  $((2n - 1) \times w)$ -wide accumulator for full-width multiplication. Half-width multiplication requires an  $((n - 1) \times w)$ -wide accumulator. The least significant  $w$ -bits do not have to be accumulated, as this consists of just a single result.

This number of multipliers in this design is limited by the number of input ports on the accumulator; it would be a waste of area if the multipliers can produce results faster than the accumulator can process them. As each multiplier produces two results, and each accumulator functional unit accepts a single  $w$ -bit wide value, at most  $\lceil \frac{n}{2} \rceil$  multipliers are used for half-width multiplication, and  $n$  multipliers for full-width multiplication. The number of cycles required for multiplications of different widths, and the actually used number of multipliers and accumulators are given in Table 7.3.

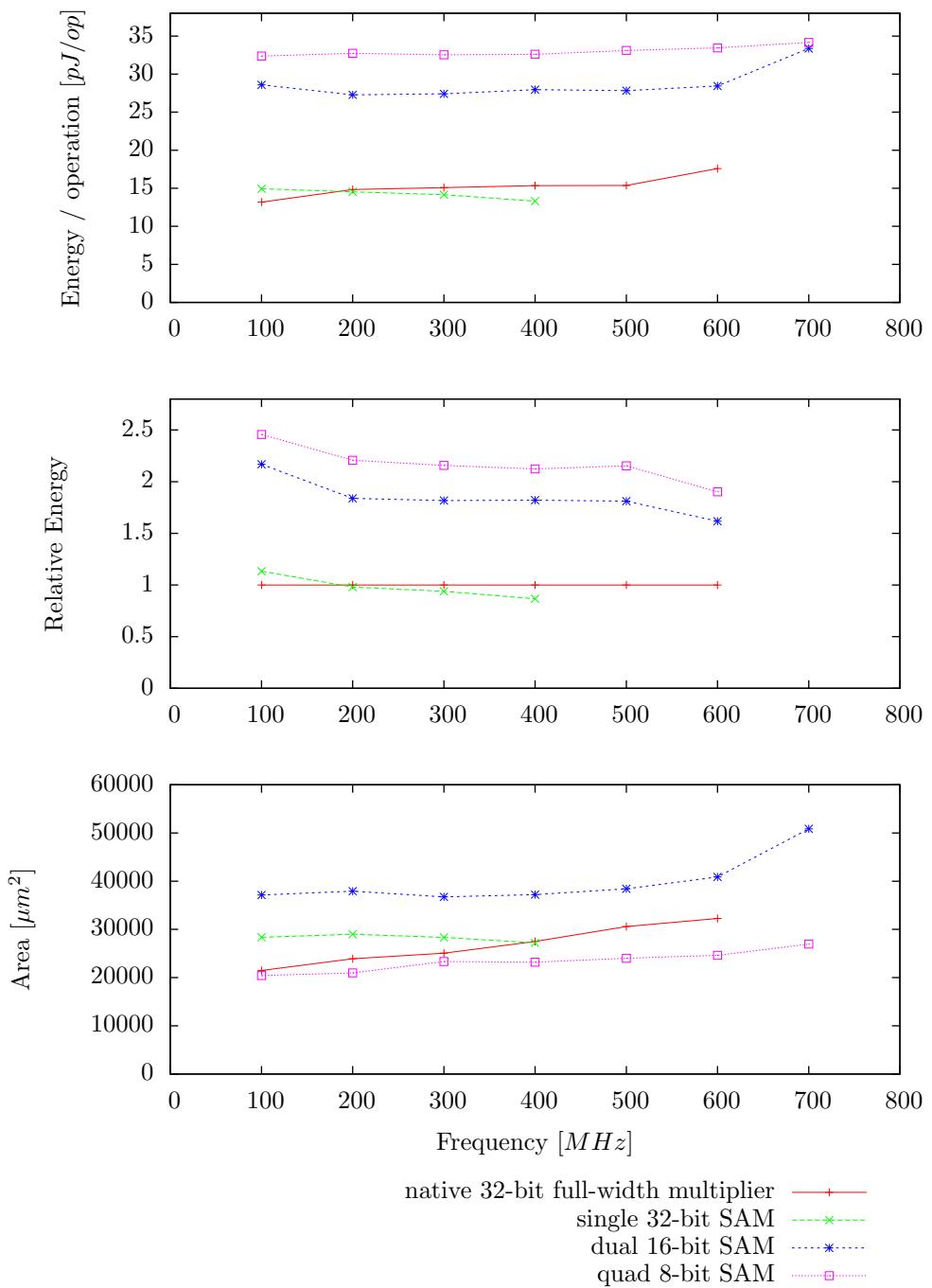
The full-width multiplication results are shown in Plot 7.5. The single 32-bit configuration is the most energy efficient configuration, nearly the same as the baseline, as this configuration does not use the accumulator because there is only a single result. The dual 16-bit configuration uses about 1.8 times the energy of the baseline, while the quad 8-bit configuration uses slightly more than twice the energy of the 32-bit baseline multiplier.

The area usage of the 32-bit configuration is, as expected, the same as the baseline, but more interesting is the quad 8-bit configuration, as this uses almost the same area as the baseline. This is because this design is not pipelined, but uses the same accumulators and multipliers in multiple cycles. The 16-bit configuration uses the most area, as this design uses fewer cycles to compute the result, and thus has less reuse in the functional units.

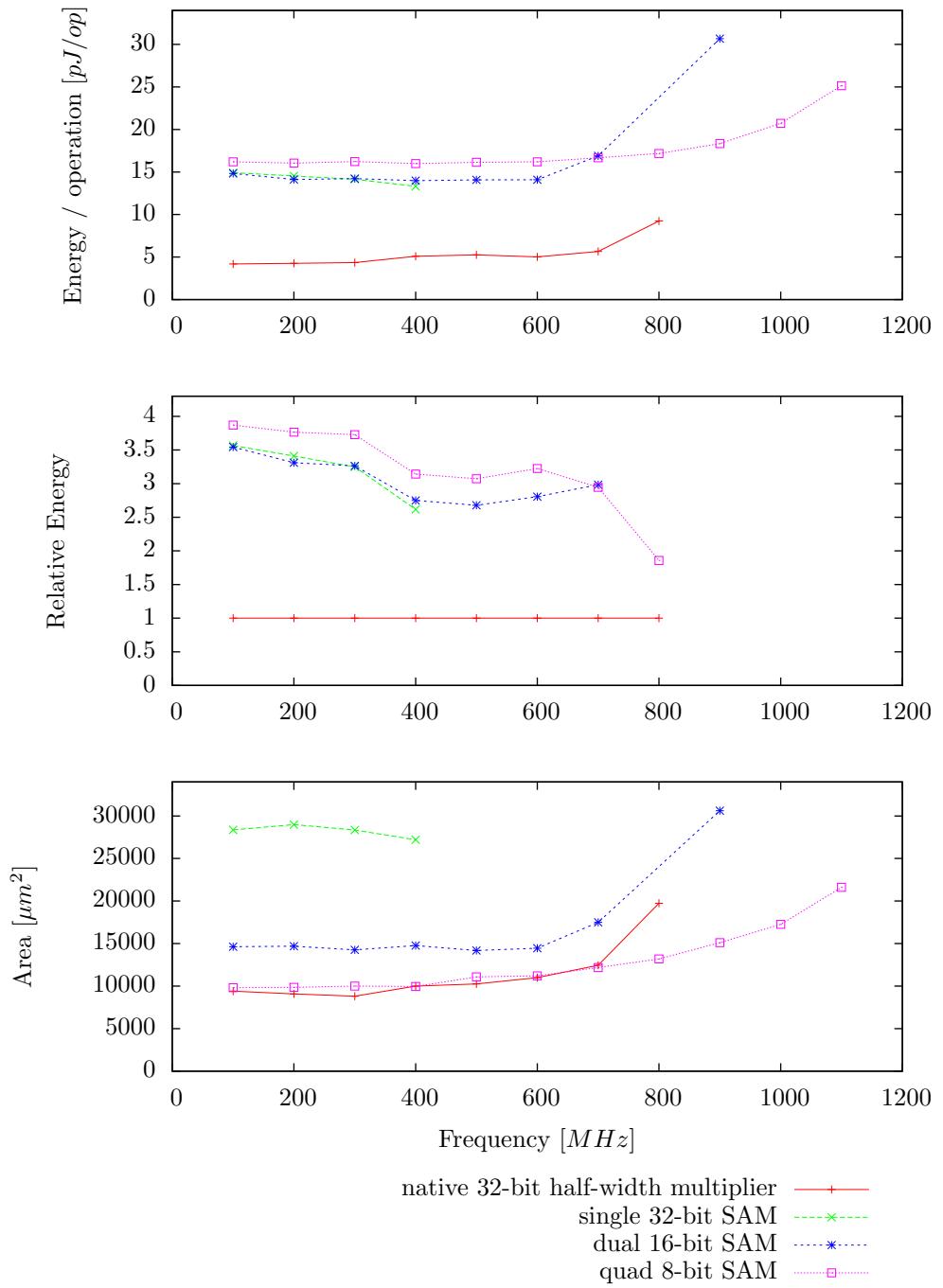
Plot 7.6 shows the energy and area usage of half-width 32-bit multiplications using this design. The energy usage of all three designs is nearly the same, around 3-3.5 times the energy usage of the baseline. The area usage of the quad 8-bit configuration is the same as the area usage of the baseline, and the dual 16-bit configuration uses slightly more area. The single 32-bit multiplier is much larger, as this is in fact a full-width multiplier.

width	full-width		half-width		delay
	multipliers	accumulators	multipliers	accumulators	
2	2	3	1	1	5
4	3	7	2	3	9
8	6	15	4	7	16

Table 7.3: Number of cycles required to calculate  $n \times w$  wide multiplications using  $w$ -bit wide functional units using the SAM algorithm.



Plot 7.5: Single-cycle 32-bit full-width multiplication with accumulator.



Plot 7.6: Single-cycle 32-bit half-width multiplication with accumulator.

### 7.3.4 Dual-Cycle Multiplier with Accumulator

If we combine the dual-cycle multiplier with an accumulator to accumulate the partial products, we have the *Dual-cycle Multiplier with Accumulator* (DAM) design.

This design consists — just as the single-cycle variant from the previous section — of a  $(2n - 1) \times w$ -wide accumulator for full-width multiplication and a  $((n - 1) \times w)$ -wide accumulator for half-width multiplication.

As these multipliers take not one but two cycles to produce their results, more multipliers are used and the delay is increased, compared to the single-cycle variant. This is also shown in Table 7.4, together with the number of used multipliers and accumulators.

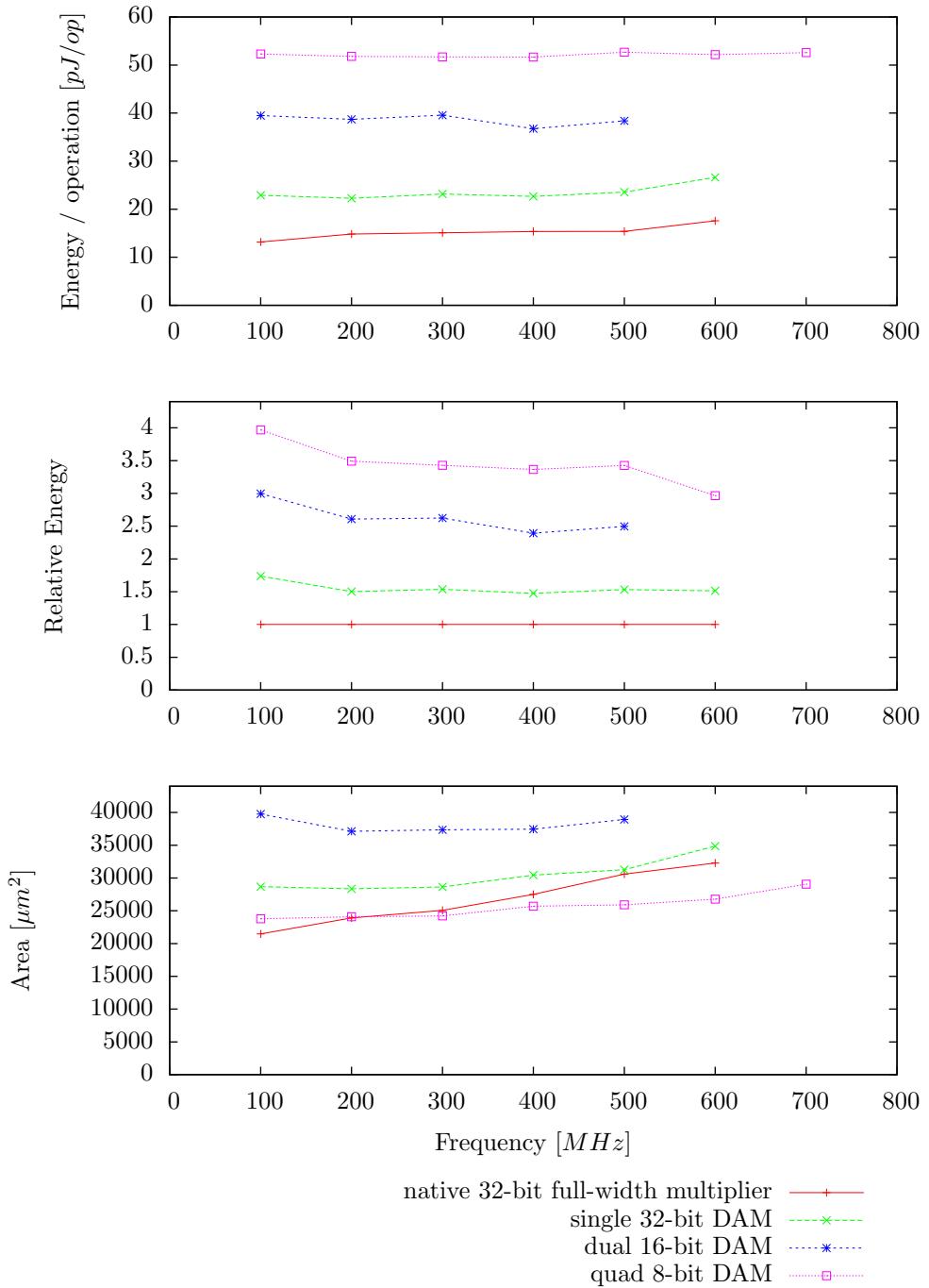
The full-width benchmark results are shown in Plot 7.7. This plot clearly shows an increase of energy usage as the functional units become smaller; the single 32-bit uses 1.5 times the energy of the baseline, the dual 16-bit around 2.5 times, and the quad 8-bit uses 3.5 times the energy of the baseline. The quad 8-bit and single 32-bit multipliers use the same amount of area as the baseline, while the dual 16-bit uses 1.5–2 times the amount of area of the baseline.

Plot 7.8 shows the benchmark result for half-width multiplication with the dual-cycle multiplier with accumulator. All these designs use a lot more energy than the baseline; 5–6.5 times as much. However at higher frequencies, the 8-bit and 16-bit configurations perform better; at 800 MHz, they use only 2.5 times as much energy as the baseline, and both designs are capable of performing up to 1000 MHz.

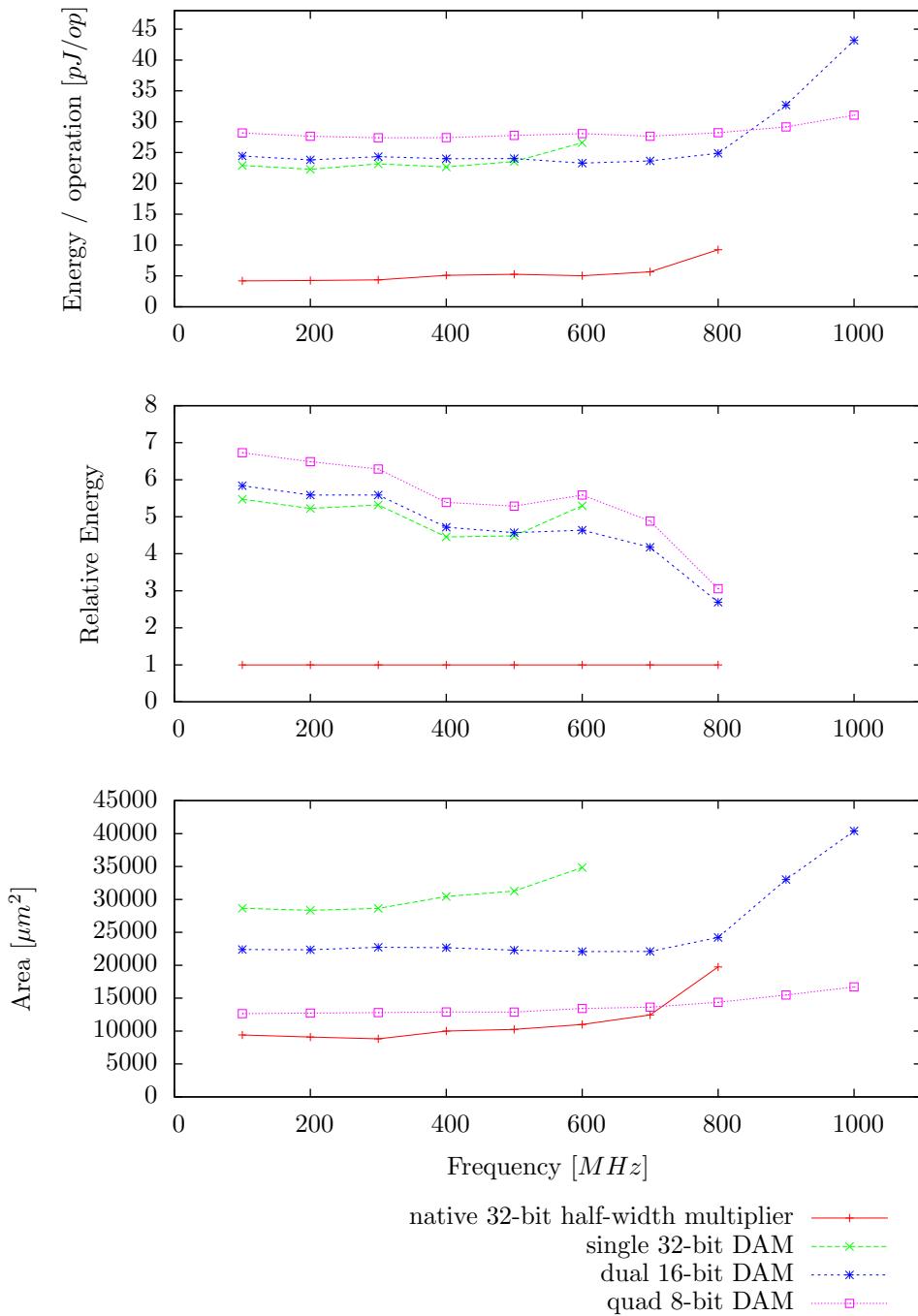
The area usage has a similar structure as the previous design; the 8-bit configuration uses the same amount of area as the baseline, the 16-bit configuration uses twice as much, and the 32-bit configuration uses three times as much area, as this is actually a full-width multiplier.

width	full-width		half-width		delay
	multipliers	accumulators	multipliers	accumulators	
2	2	3	2	1	6
4	4	7	3	3	10
8	7	15	6	7	21

Table 7.4: Number of multipliers, adder and cycles required to calculate  $n \times w$  wide multiplications using  $w$ -bit wide functional units using the DAM algorithm.



Plot 7.7: Dual-cycle 32-bit full-width multiplication with accumulator.



Plot 7.8: Dual-cycle 32-bit half-width multiplication with accumulator.

### 7.3.5 Standalone Multiplier

The design with only a single multiplier is called *Standalone Single-cycle Multiplier* (SSM) or *Standalone Dual-cycle Multiplier* (DSM), depending on the type of multiplier that is used.

As this design uses only a single multiplier, no adder scheme is needed to accumulate all the results. And because our multipliers are only capable of performing full-width multiplication, we will not show the half-width plots here as they are identical.

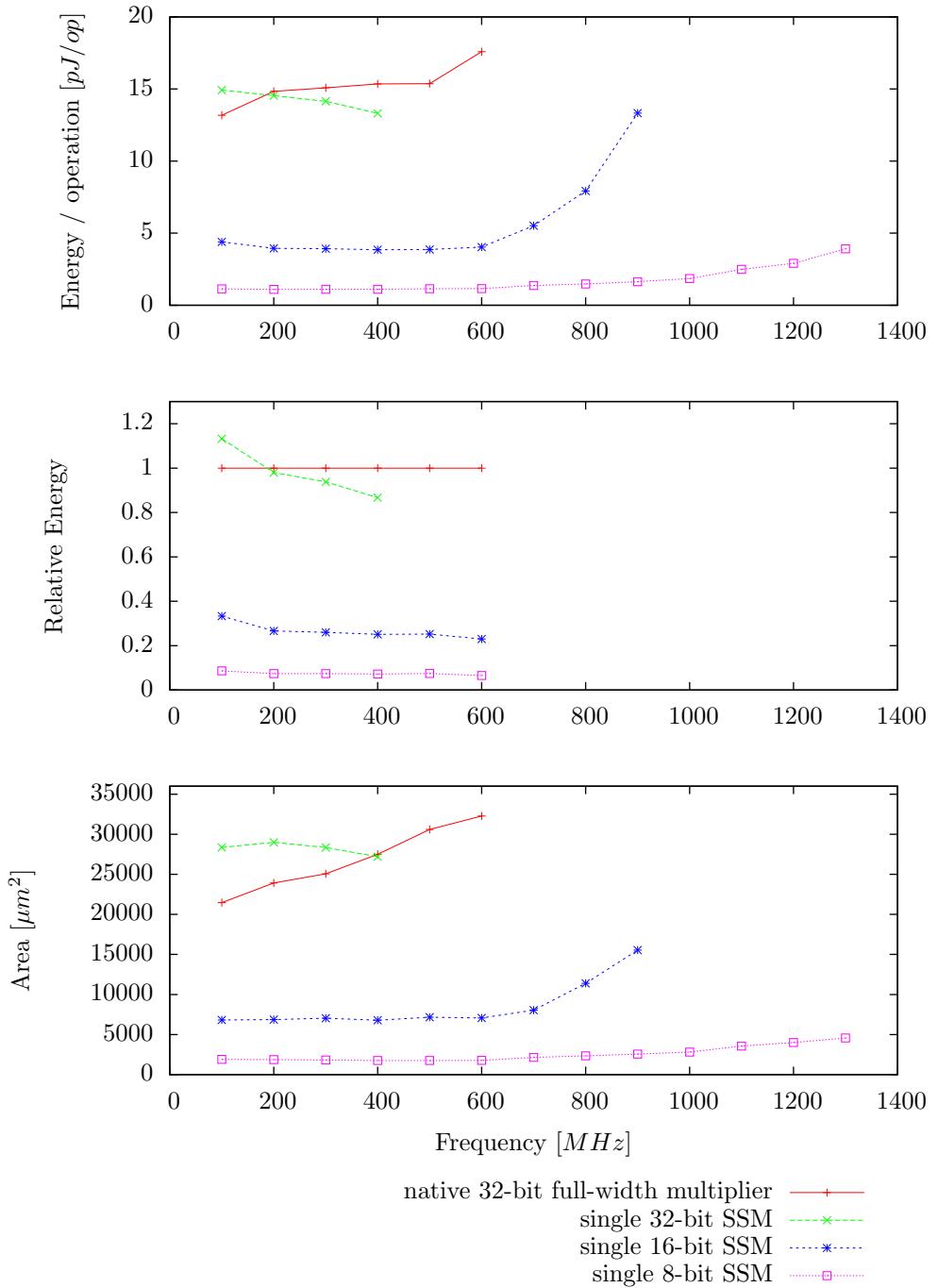
Where the previous designs showed the penalty of the added flexibility of the multi-granular designs, here we will look at the benefits of this technique; how much energy and area can be saved when smaller operations are performed.

Plot 7.9 shows the results of full-width 8-bit multiplication using one single-cycle multiplier. Just as in the previous sections, the 32-bit multiplier performs the same as the baseline. The 16-bit functional unit saves a factor 3–5, depending on the operating frequency, compared to the baseline, and the 8-bit multiplier is even able to save a factor 15 from the energy usage of the baseline. The 16-bit and 8-bit configurations are also able to achieve higher operating frequencies, up to 900 MHz and 1300 MHz respectively.

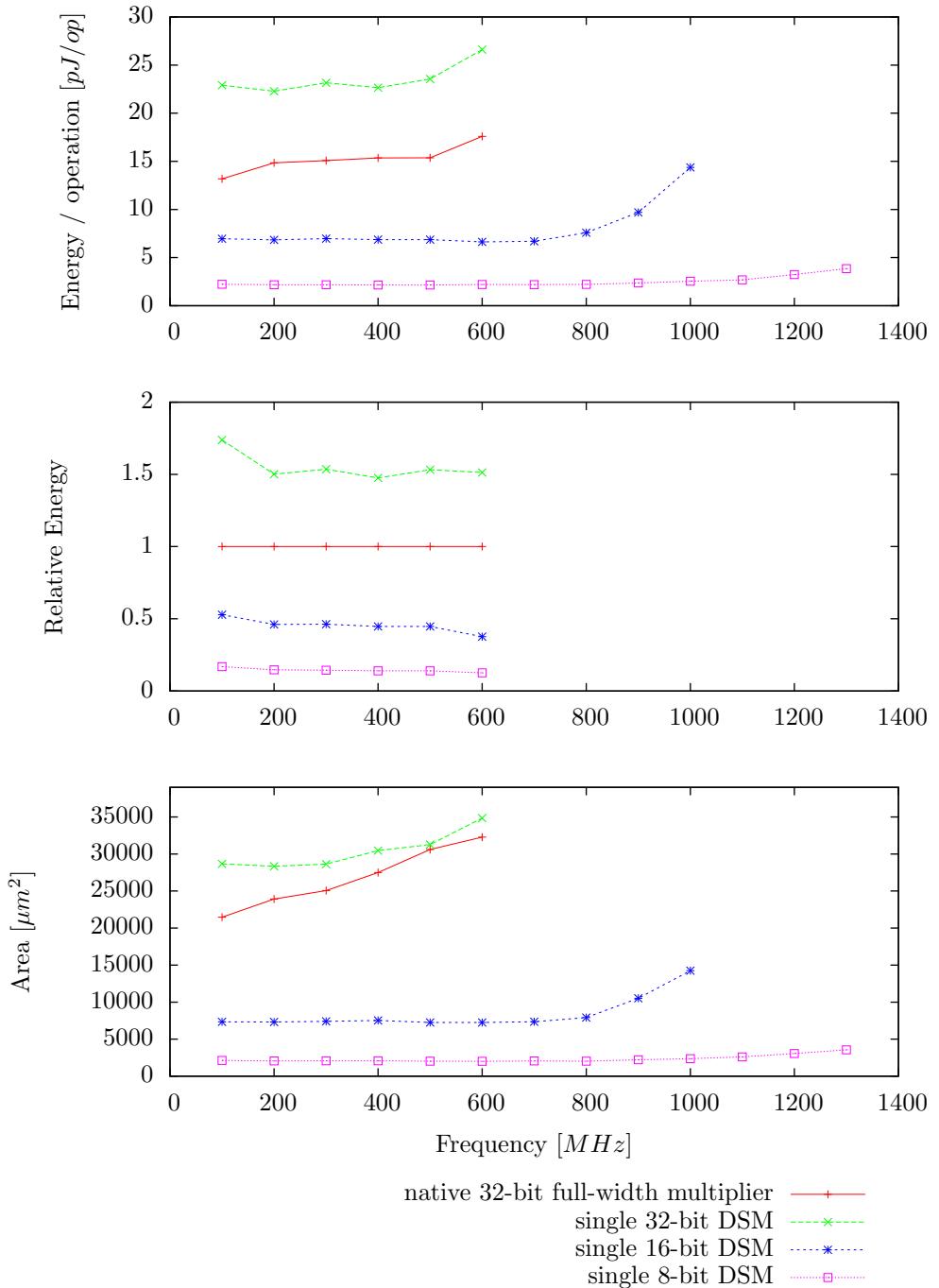
The area usage of these smaller configurations is also significantly less than the baseline: the 16-bit variant uses 25% and the 8-bit variant uses only 5% of the area of the baseline.

The dual-cycle multipliers in Plot 7.10 also show large energy savings, although the savings here are not as good as for the single-cycle multiplier. The 32-bit multiplier uses 1.5 times as much energy as the baseline, while the 16-bit functional unit is able to save roughly half of the energy that the baseline uses, and the 8-bit multiplier uses only 16–12% of the energy of the baseline, saving a factor 8. These multipliers are able to operate at 1000 MHz and 1300 MHz, for the 16-bit and 8-bit variant respectively.

The savings in area are comparable to the savings for the single-cycle multipliers: the 16-bit multiplier uses 4.5 times less area, and the 8-bit multiplier uses 6% of the area of the baseline.



Plot 7.9: Single-FU single-cycle 8-bit full-width multiplication.



Plot 7.10: Single-FU dual-cycle 8-bit full-width multiplication.

## 7.4 Comparison

We have discussed two multiplier types, the single-cycle multiplier and the dual-cycle multiplier. Additionally, we have introduced two techniques to add all the partial products together, the adder tree and the accumulator. Together, they form four designs — single-cycle multiplier with adder tree (STM), dual-cycle multiplier with adder tree (DTM), single-cycle multiplier with accumulator (SAM), and dual-cycle multiplier with accumulator (DAM) — which are discussed individually in the previous section. In this section, we will compare these designs in order to determine the trade-off between the overhead when performing 32-bit multiplication on one hand, and the energy and area savings when performing 8-bit multiplications on the other hand.

First, we compare the different configurations when performing 32-bit full-width multiplication in Plots 7.11 and 7.12. Both show the same relative behaviour of the designs: the STM design has the lowest energy usage, followed by the SAM, DTM and finally the DAM. The energy overhead of the STM configurations is 1.4–1.7 times the baseline when 8-bit functional units are used, and 1.5–2.4 times the baseline for the 16-bit functional units. However, this design uses the most area: 4–6 times as much as the baseline.

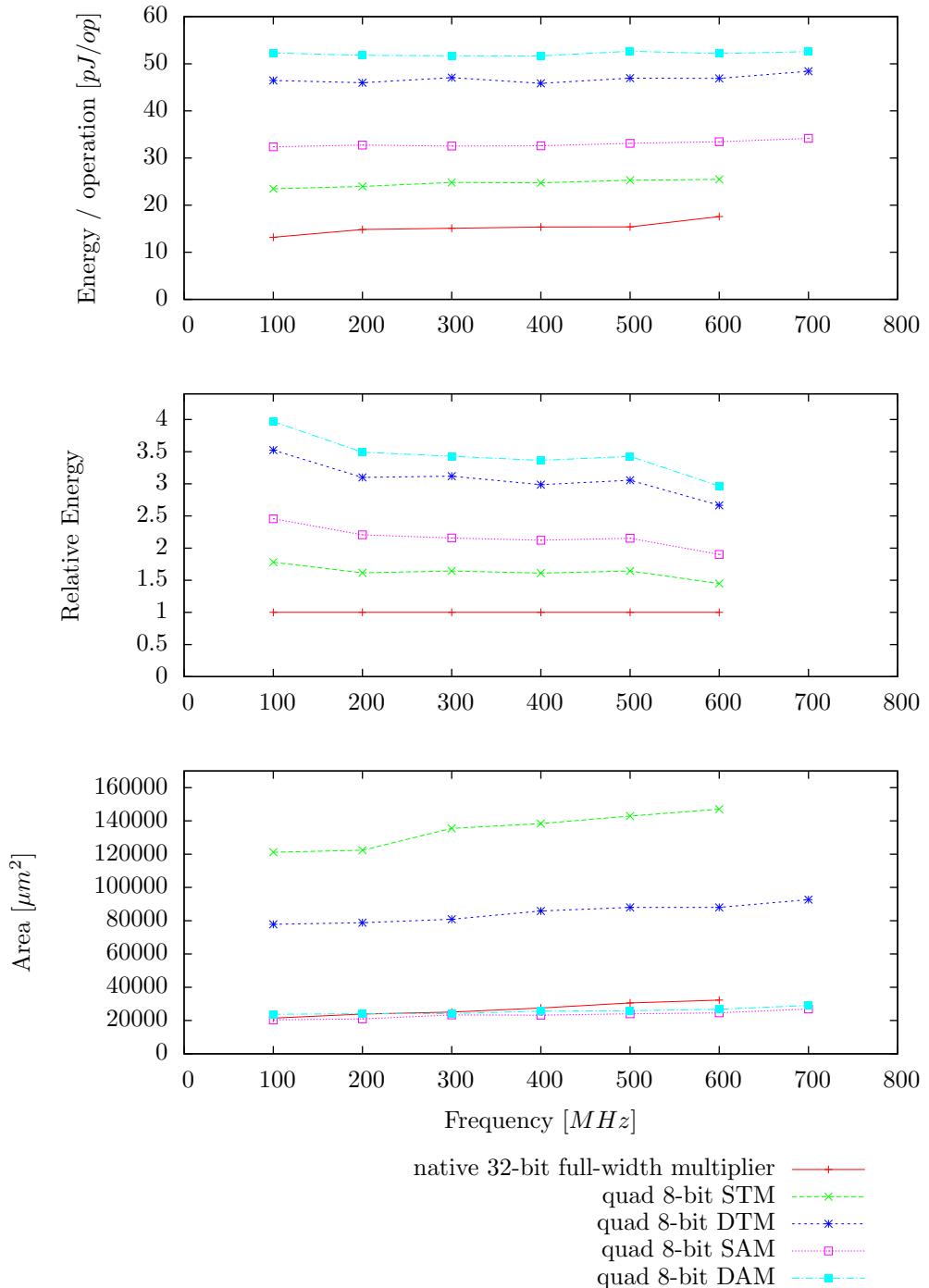
The same results hold for half-width 32-bit wide multiplication, as depicted in Plots 7.13 and 7.14: the STM design uses a factor 1.5–2.5 more energy than the baseline when 8-bit functional units are used, and a factor 2–2.5 for 16-bit functional units. Note that here a half-width baseline multiplier is used, which is roughly 3 times as energy efficient as the native full-width baseline.

For the 8-bit multiplication, only the full-width output is considered because our functional units are unable to compute half-width results more efficiently than full-width results. As the interconnect scheme also does not affect the results, as discussed in Section 7.3.5, we are left with only four configurations: single-cycle and dual-cycle multipliers for both 8-bit and 16-bit granularity.

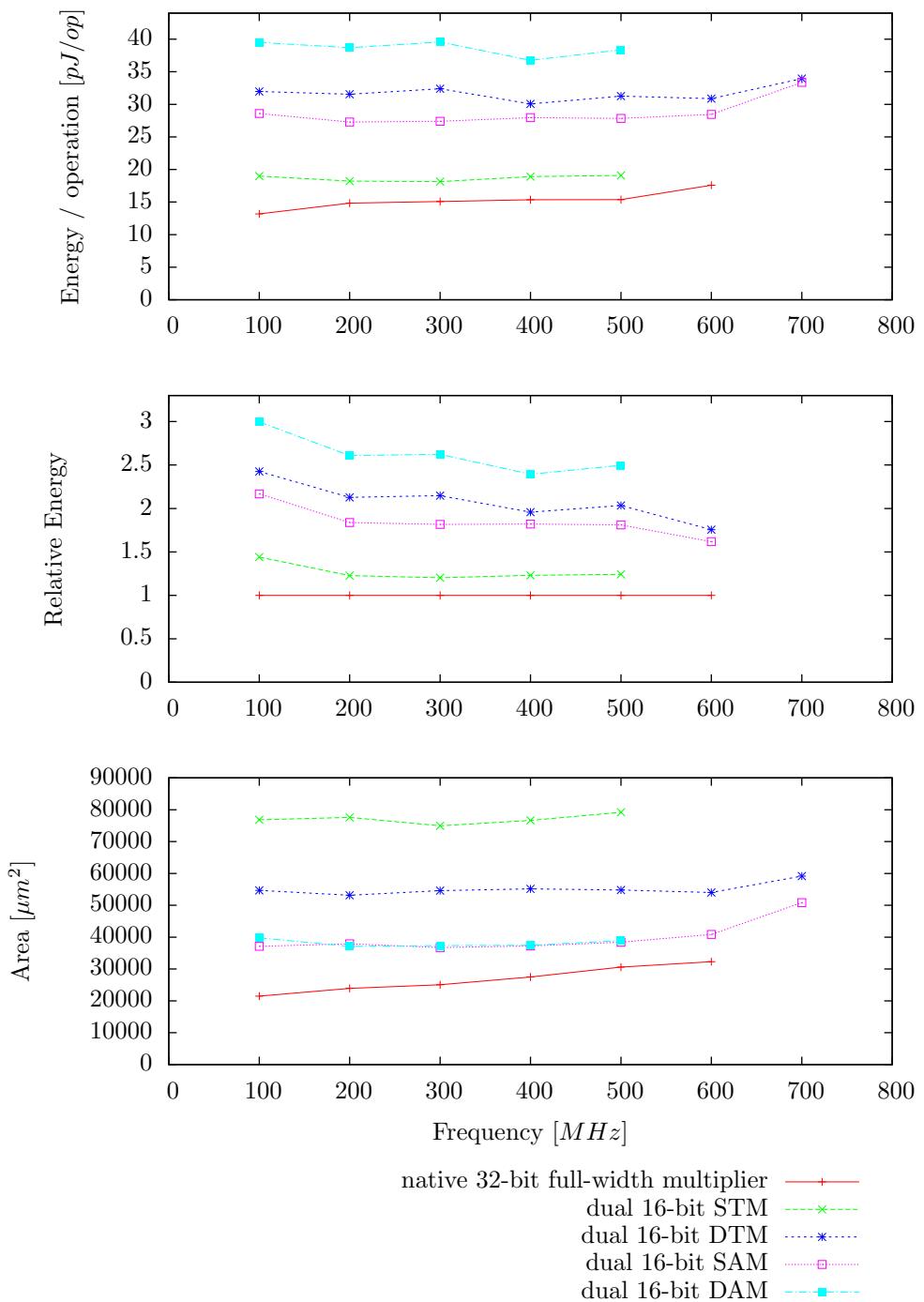
For both 8-bit and 16-bit functional units, the single-cycle multipliers are the most energy efficient, while area usage is nearly identical for both designs. The 8-bit functional unit can perform 8-bit multiplication using only 6–8% of the energy the baseline would need, while the 16-bit functional unit requires roughly 25% of the energy of the baseline.

This shows that the single-cycle multipliers perform better than the dual-cycle multipliers, which is not unexpected as the dual-cycle multipliers needs more registers in order to store the upper half of the result. The single-cycle multiplier, however, puts a higher burden on the interconnect as it needs two output ports. As the energy usage of the interconnect is not taken into account, more measurements are needed before we can form a definitive verdict.

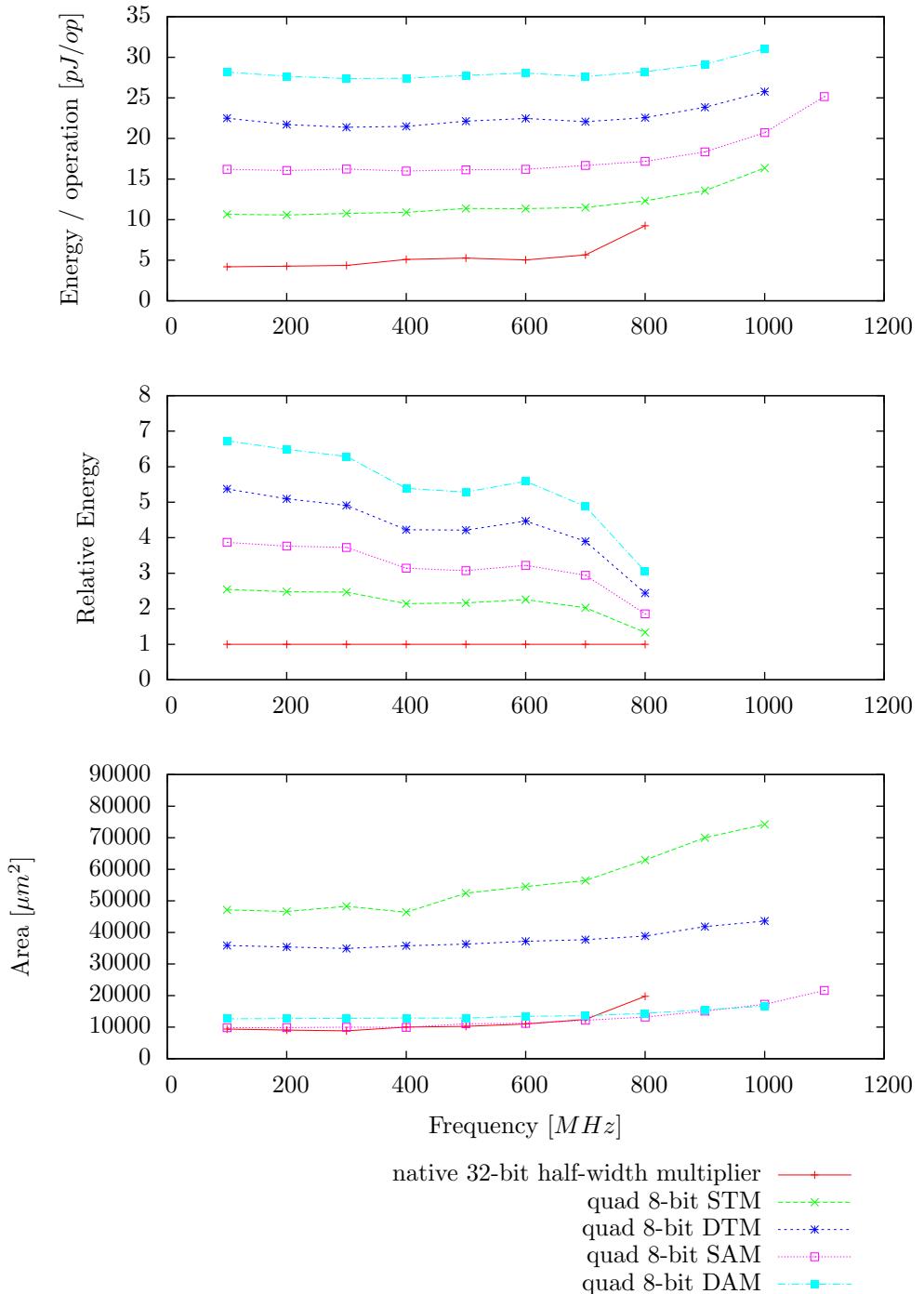
The adder tree performs better than the accumulator in terms of energy, while the accumulator uses far less functional units and thus area. These measurements were taken on a fully pipelined adder tree; if the application does not need to perform a multiplication each cycle, the tree will not perform as well.



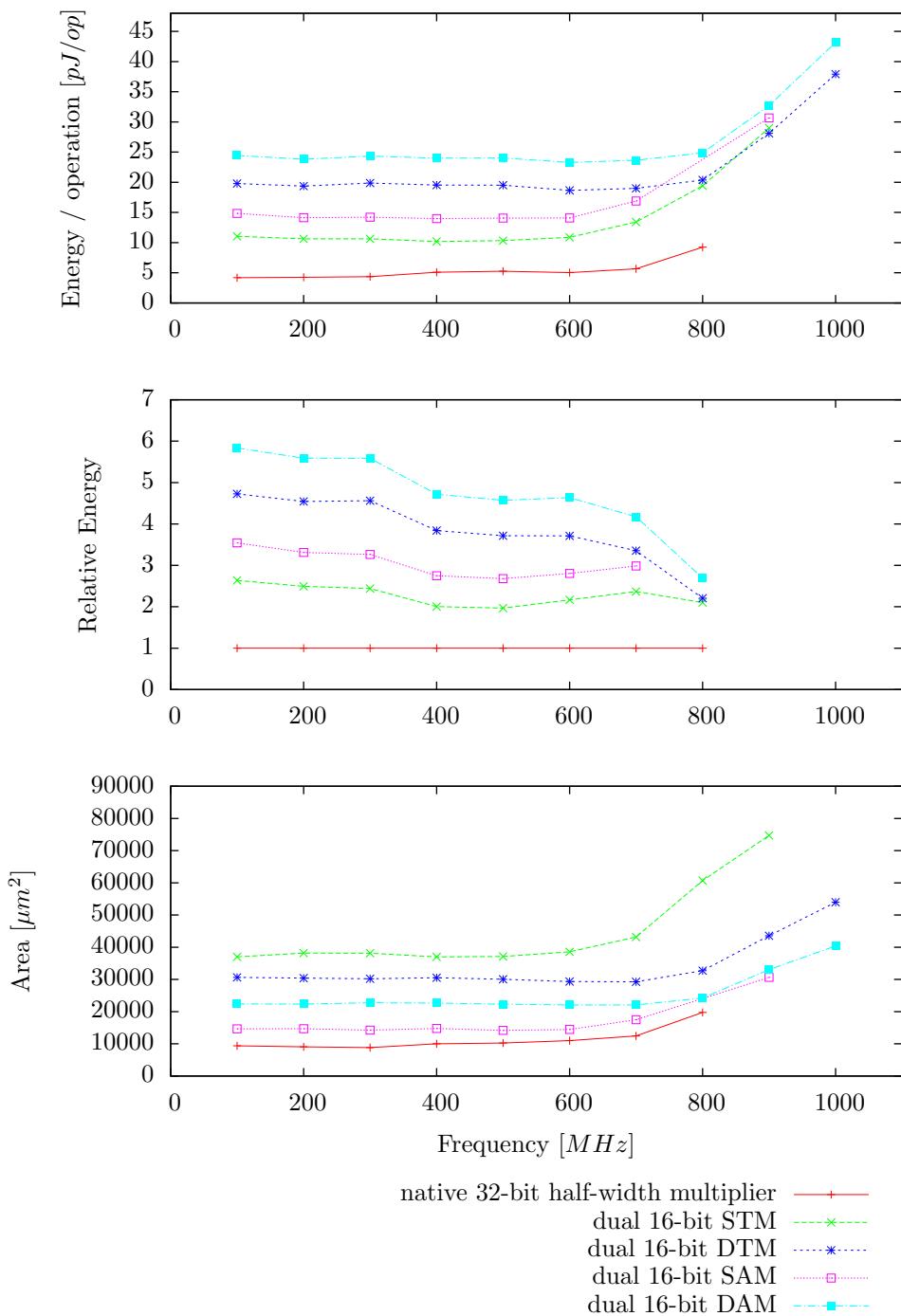
Plot 7.11: 32-bit full-width multiplication using 8-bit functional units.



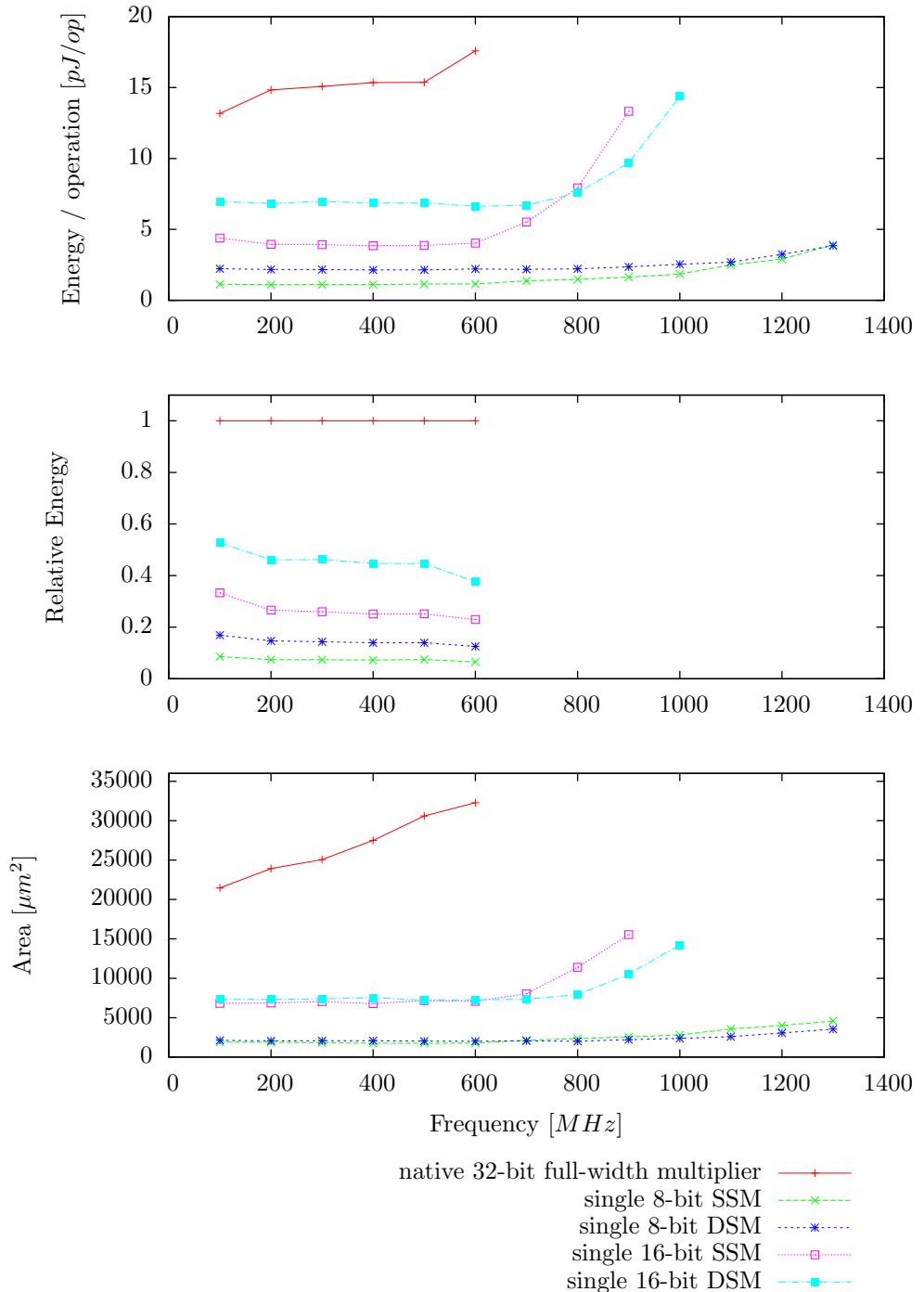
Plot 7.12: 32-bit full-width multiplication using 16-bit functional units.



Plot 7.13: 32-bit half-width multiplication using 8-bit functional units.



Plot 7.14: 32-bit half-width multiplication using 16-bit functional units.



Plot 7.15: 8-bit full-width multiplication using 8-bit and 16-bit functional units.

## 7.5 Conclusions

After comparing and reviewing all the designs in the previous two sections, we can draw the following conclusions.

The single-cycle multiplier performs the best, although with a higher impact on the interconnect than the dual-cycle multiplier. Which multiplier is best in a certain architecture depends on the costs of adding a second output port to the functional units. However this is certainly worth considering.

The choice of accumulation technique depends more on the application; if many multiplications have to be performed then the adder tree is the way to go. This technique has a throughput of one multiplication per one or two cycles, depending on the chosen multiplier. If a single or few multiplications have to be performed, it might be more efficient to use the accumulator, which is not pipelined and uses far less area. Fortunately, this choice can be made compile-time, and thus can be optimised for the individual application.

It is also possible to form a hybrid between the adder tree and the accumulator, by placing one or more adders in front of the accumulators, where each adder increases the input bandwidth of the accumulator. This is especially effective on the central accumulators, as they need to accumulate the most partial products. This can reduce the number of cycles required to perform a multiplication, at the expense of using more functional units. We do not consider this configuration in the context of this report, and this is left as future work.

As all functional units can currently only perform full-width operations, even if only half-width results are needed. If half-width operations are common, it might be worth investigating options to disable the upper part of the multiplier in order to save some energy. Again, we did not consider this here, and thus this is left as future work.

The multi-granular decomposition is not for free for multiplication, but the gain in case smaller width operations are used is also huge. The overhead costs, in case a 32-bit multiplication is composed from 8-bit functional units, increase the energy per operation with a factor 1.5–3.5 compared to the baseline, depending on the multiplier design used. When 8-bit multiplications are performed, the gain can be a factor 15.

Often, when a half-width multiplication is performed, it is sufficient to perform a full-width multiplication at half the width; e.g. instead of a  $32 \times 32 = 32$ -bit multiplication, a  $16 \times 16 = 32$ -bit multiplication is adequate, which is more energy efficient. Similarly, when a big number is multiplied by a small number, then, for example, instead of a  $32 \times 32 = 32$ -bit multiplication, a  $32 \times 8 = 32$ -bit multiplication can be performed, which uses significantly less energy.

So to conclude, the choice of multiplier depends on the used interconnect network, and which accumulation technique is most suitable depends on the application, and can be decided at compile time.

# Chapter 8

## Multiply-Accumulation

Multiply-accumulation, sometimes abbreviated as MAC, is the combination of both multiplication and addition; the product of many multiplication is accumulated in order to calculate a sum of the form  $x = \sum_{i=0}^N a_i \times b_i$ .

### 8.1 Multiply-Accumulation Algorithms

Multiply-accumulation consists, just as add-accumulation, of three phases: initialisation, accumulation, and finalisation. A simple implementation of this operation consists of an accumulator which is initialised in the initialisation phase, followed by the accumulation phase where a multiplier computes the products of all the operand pairs, which are then added together by the accumulator, and finally, in the finalisation phase, the accumulator produces the result.

In Chapter 6, we introduced the accumulator in order to make repeated addition more efficient. One of these accumulation algorithms was the carry-save adder, which uses a redundant data format to store the intermediate accumulation results, known as the carry-save format. The HPM-multiplier introduced in Chapter 7 also produces a result in the carry-save format, after which an adder is needed to perform the final addition to produce the multiplication result.

It is possible to combine those two techniques: instead of performing an addition after the multiplier produces its result in carry-save format, this carry-save result could directly be added to the intermediate register using a carry-save adder. This saves computations, as the carry-save adder is very efficient, but it does require double registers to save the intermediate result in the carry-save format.

## 8.2 Multi-Granular Multiply-Accumulation

In order to perform multi-granular multiply-accumulate operations, several techniques are possible. Here we will discuss two of those methods: an extension of the multiplier with accumulator from the previous chapter, and a multiply-accumulator that uses the technique described in the previous section; keeping the intermediate results as long as possible in the carry-save format.

### 8.2.1 Multiply-Accumulation with Accumulator

This design, based on the multiplier with accumulator algorithm as described in Section 7.2.5, uses the accumulators for both the partial product accumulation as well as the accumulation of multiple products.

This design deviates from the multiply-accumulator on two points: the first word now also needs an accumulator, such that multiple products can be accumulated there. Secondly, the accumulators should not initialise before each multiplication, but only at the start of the accumulation, and the result can be retrieved after all accumulations are completed.

This design required multiple cycles per multiplication step. Just like the multiplier, this design can be speed up by placing extra adders in front of the accumulators, doubling there input capacity. Where this was limited with the normal multiplier because each extra layer also added a cycle delay, this is less an issue here because this extra cycle is only needed after all accumulation steps are completed. This can be done up the the point where it becomes a multiplication tree with accumulator at the bottom, processing a single multiplication every cycle.

### 8.2.2 Distributed Multiply-Accumulator

The distributed multiply-accumulator works as long as possible in the carry-save format, in order to reduce the delay and save energy.

To do this, the result of the HPM-multiplier, which is in carry-save format, is directly accumulated to the accumulation register using a carry-save adder. As there are four input values, two from the multiplier and two from the register, two carry-save adders are needed to perform the addition. This results in the production of two carry-out bits, which are send to an incrementer that adds these overflow bits to an overflow register, as shown in Figure 8.1. This way, the multiply-accumulators can operate independently of each other, and only at the end of the accumulation they need to communicate to produce the final result. This reduces interconnect usage, routing complexity and gives more scheduling freedom.

The overflow register is not able to store an unlimited amount of carry-out bits. Thus, if many products have to be accumulated this register has to be “emptied”. This can be done by sending the content of the overflow register to another accumulator. This has to be done every  $2^o - 1$  cycles, where  $o$  is the width of the overflow register, which can be as wide as the functional unit, or smaller.

When it is time to produce the final result, the functional units start to work as accumulators, where the partial results in each column are accumulated. When only a single functional units holds a value in each column, the overflow bits are also propagated to the next column and the final result is produced by adding the two parts of the carry-save format. An example schedule is given in Figure 8.2, but this can be done in many ways; the exact order of these operations is best left to the scheduler of the compiler, as the compiler can take additional constraints into account, such as other parallel operations that require access to the interconnect.

This design can operate without any carry chains, however, this does delay the final accumulation phase by a couple of cycles. Switching to “carry-chain mode” for the accumulation phase speeds this up, as this removes some of the order restrictions, if the interconnect criteria and placement of the functional units allow this.

### 8.3 Multiply-Accumulate Configurations

Based on the two techniques described in the previous sections, we form three different configurations. The multiply-accumulation with accumulator design has two variants: one with the single-cycle multipliers, and the second with the dual-cycle multipliers. The distributed multiply-accumulator only has one variant.

A multiply-accumulation operation consists of multiple steps; the number of terms to be added. In order to provide a fair comparison between the designs, all benchmarks discussed here are performed with 100 multiply-accumulate steps.

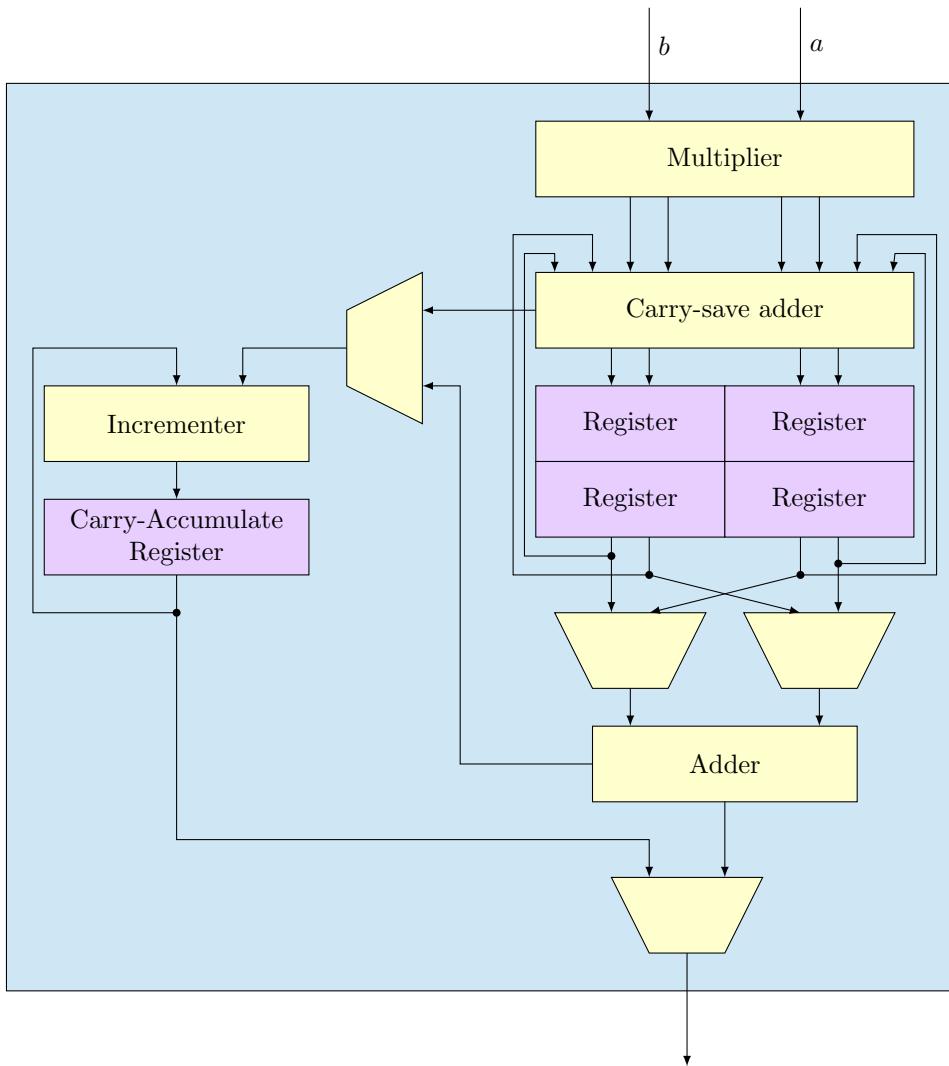


Figure 8.1: Schematic overview of the modules in a distributed multiply-accumulator functional unit.

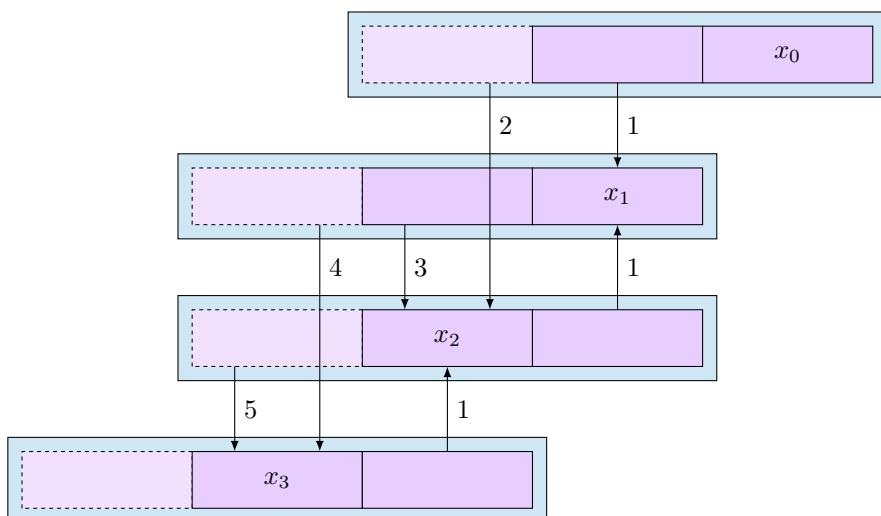


Figure 8.2: A possible schedule for accumulating the final result for a distributed multiply-accumulator. The numbers at the arrow are cycle numbers, and register  $x_i$  contains part  $i$  of the final result.

### 8.3.1 Single-Cycle Accumulator-based MAC

The first design we discuss here consists of single-cycle multipliers, where the partial products are added using accumulators. This design is called *Single-cycle Accumulator-based Multiply-Accumulator* (SAMAC).

Plot 8.1 shows the benchmark results for full-width multiply-accumulation on 32-bit wide inputs. Our single 32-bit implementation already performs slightly better than the baseline, probably because our implementation uses a separated multiplier and accumulator, making it a two cycle pipeline, whereas the native multiply-accumulator is a single-cycle implementation. This is also visible in the area usage, which is 3–5 times higher for the single 32-bit SAMAC compared to the baseline.

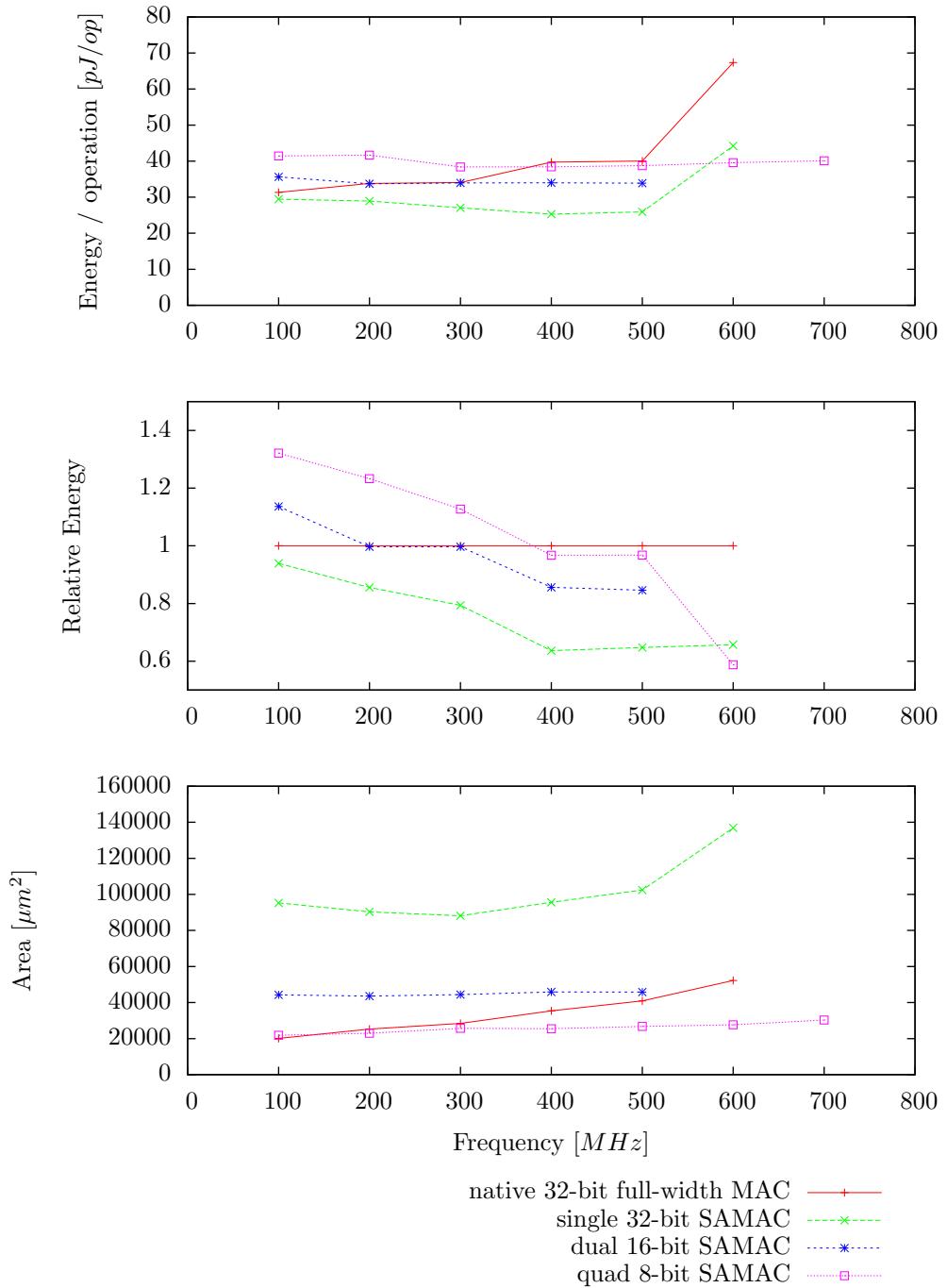
The dual 16-bit configuration performance is similar to the baseline, using 35 pJ/op over the entire frequency domain, while the quad 8-bit uses somewhat more energy, 40 pJ/op. At 100 MHz, this means an increase of about 30% energy usage compared to the baseline. At 600 MHz, the quad 8-bit is still at 40 pJ/op, while the baseline is at 70 pJ/op.

Note that the 32-bit configuration uses the most area, followed by the dual 16-bit and finally the single 8-bit uses the least area. This is because the configurations with smaller functional units reuse the same multipliers for multiple partial products, as discussed in Section 7.2.5, and thus also use more cycles.

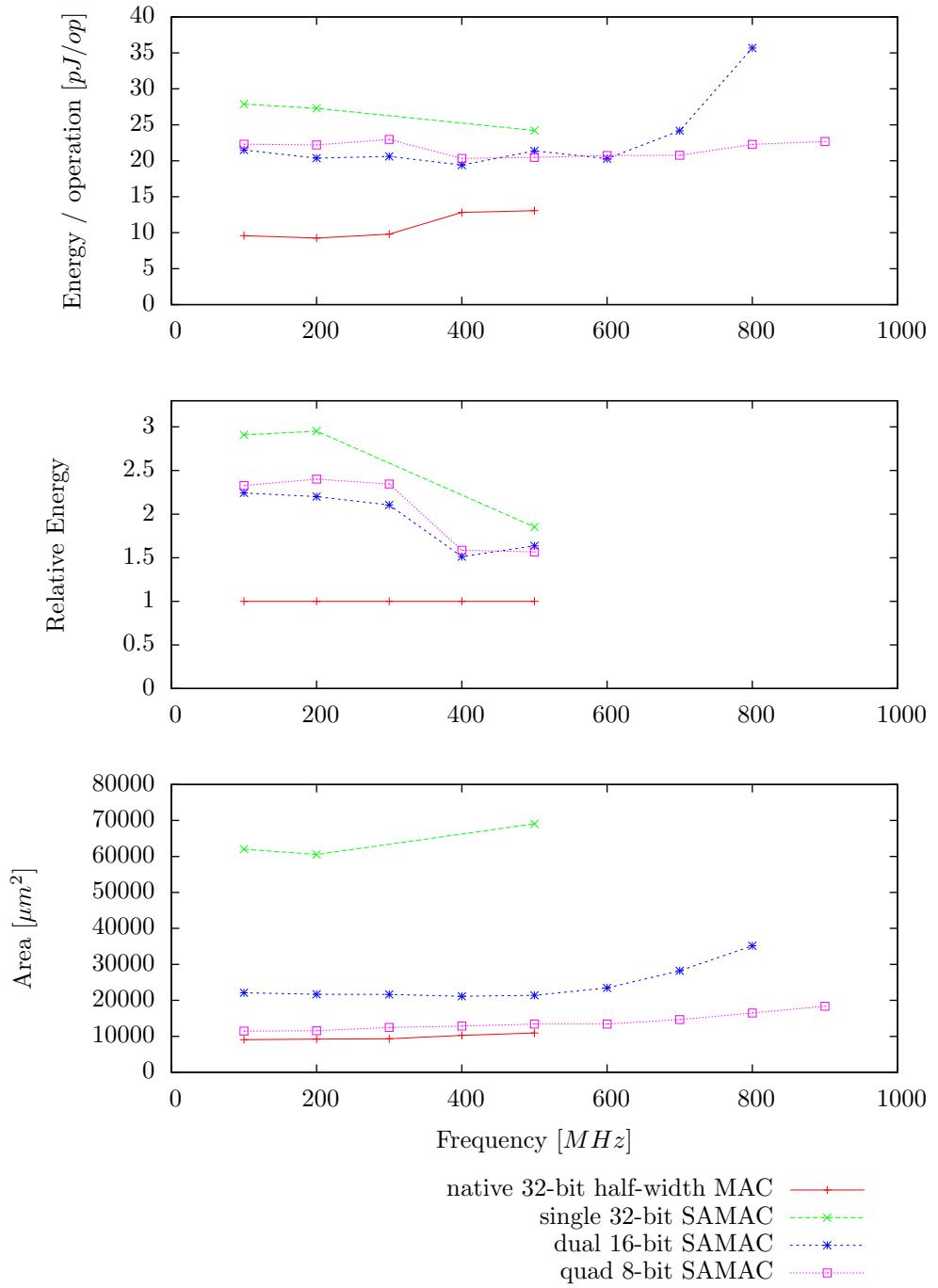
The results for 32-bit half-width multiply-accumulation are in Plot 8.2. Compared to the full-width operations from the previous plot, the dual 16-bit and quad 8-bit configurations use almost half the amount of energy, while the energy of the single 32-bit configuration stays nearly the same. However the baseline is able to reduce the energy from 30 pJ/op to 10pJ/op, resulting in an energy usage of 1.5–3 times the baseline.

When performing 8-bit full-width multiply-accumulation, as shown in Plot 8.3, the field changes in favour of the smaller functional units. The 16-bit functional units use 11% of the energy of the baseline, while the 8-bit functional units use only 3% of the energy of the baseline, saving a factor 33. Additionally, the 8-bit configuration is able to operate at much higher frequencies, up to 1100 MHz.

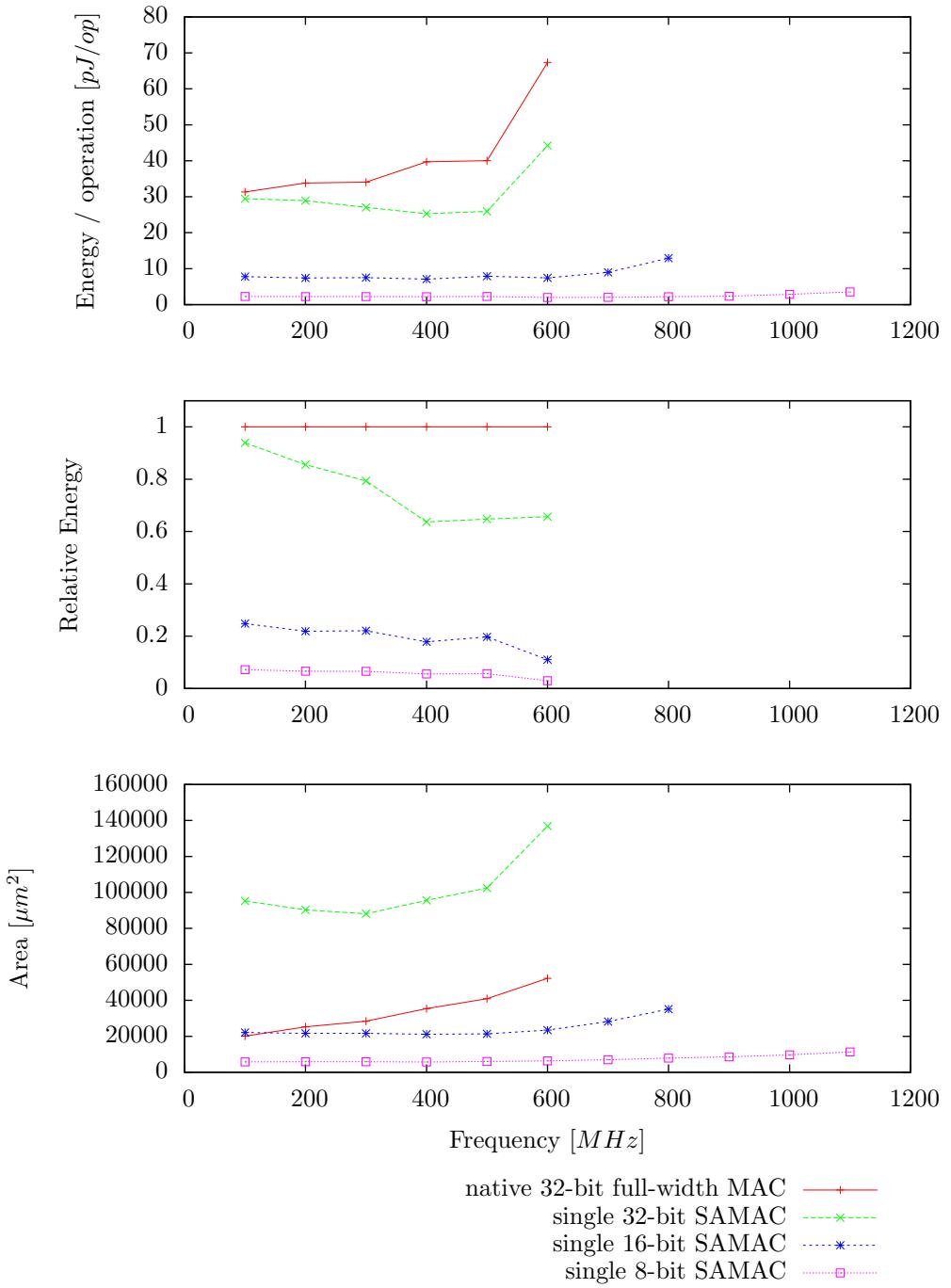
The benchmark results for half-width 8-bit multiply-accumulation are shown in Plot 8.4. Again the smaller functional units perform better than the baseline and the 32-bit configuration. The 16-bit configuration uses 50–70% of the energy of the baseline, while the 8-bit configuration uses 13–18% of the energy of the baseline.



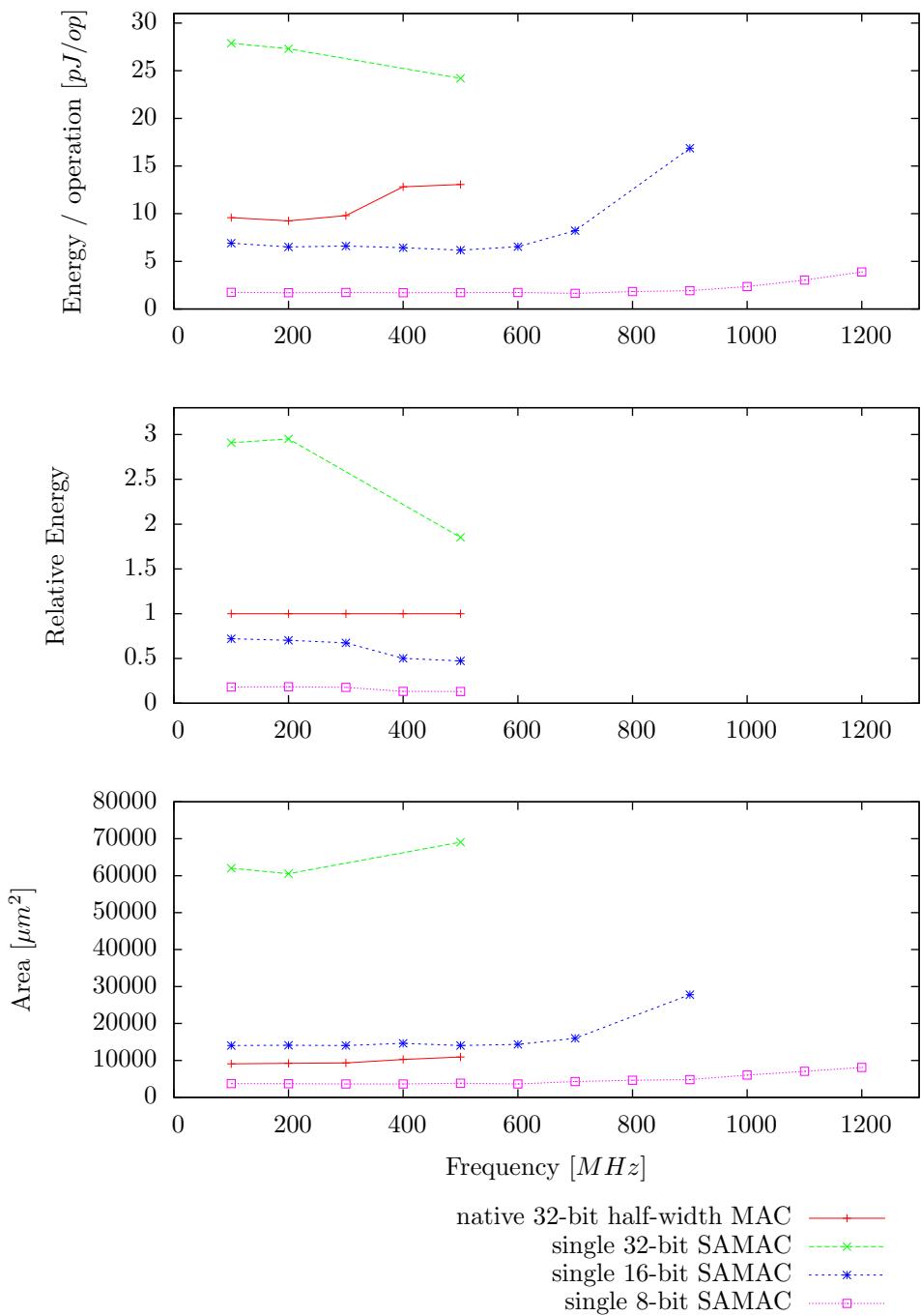
Plot 8.1: Single-cycle 32-bit full-width accumulator-based multiply-accumulation.



Plot 8.2: Single-cycle 32-bit half-width accumulator-based multiply-accumulation.



Plot 8.3: Single-cycle 8-bit full-width accumulator-based multiply-accumulation.



Plot 8.4: Single-cycle 8-bit half-width accumulator-based multiply-accumulation.

### 8.3.2 Dual-Cycle Accumulator-based MAC

When a dual-cycle multiplier is used in combination with an accumulator to perform multiply-accumulate operations, then this design is called *Dual-cycle Accumulation-based Multiply-Accumulator* (DAMAC).

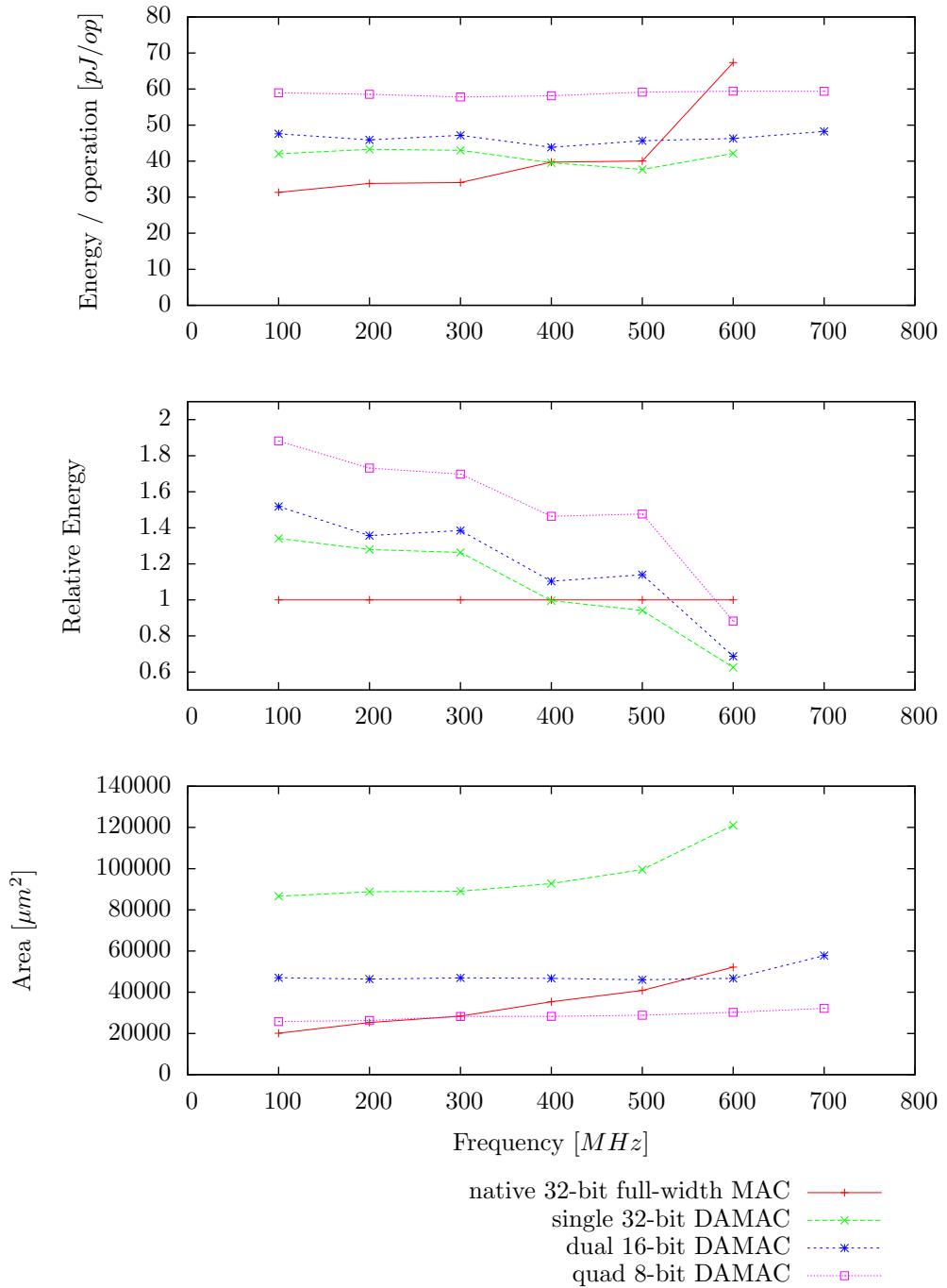
Just like the previous design, this design is based on the multiplier with accumulator from Section 7.2.5. However, this design uses dual-cycle multipliers, which is mainly an advantage for the interconnect, as these multipliers only have a single output port.

Plot 8.5 shows the results for full-width 32-bit multiply-accumulation. The energy per operation is not very dependent on the operating frequency, and lies between 40 and 60 pJ/op. The baseline is a bit more energy efficient with 30 pJ/op at 100 MHz, but at 600 MHz, the baseline uses 70 pJ/op; more than all other configurations. The single 32-bit configuration is the most efficient, followed by the dual 16-bit and the quad 8-bit configurations.

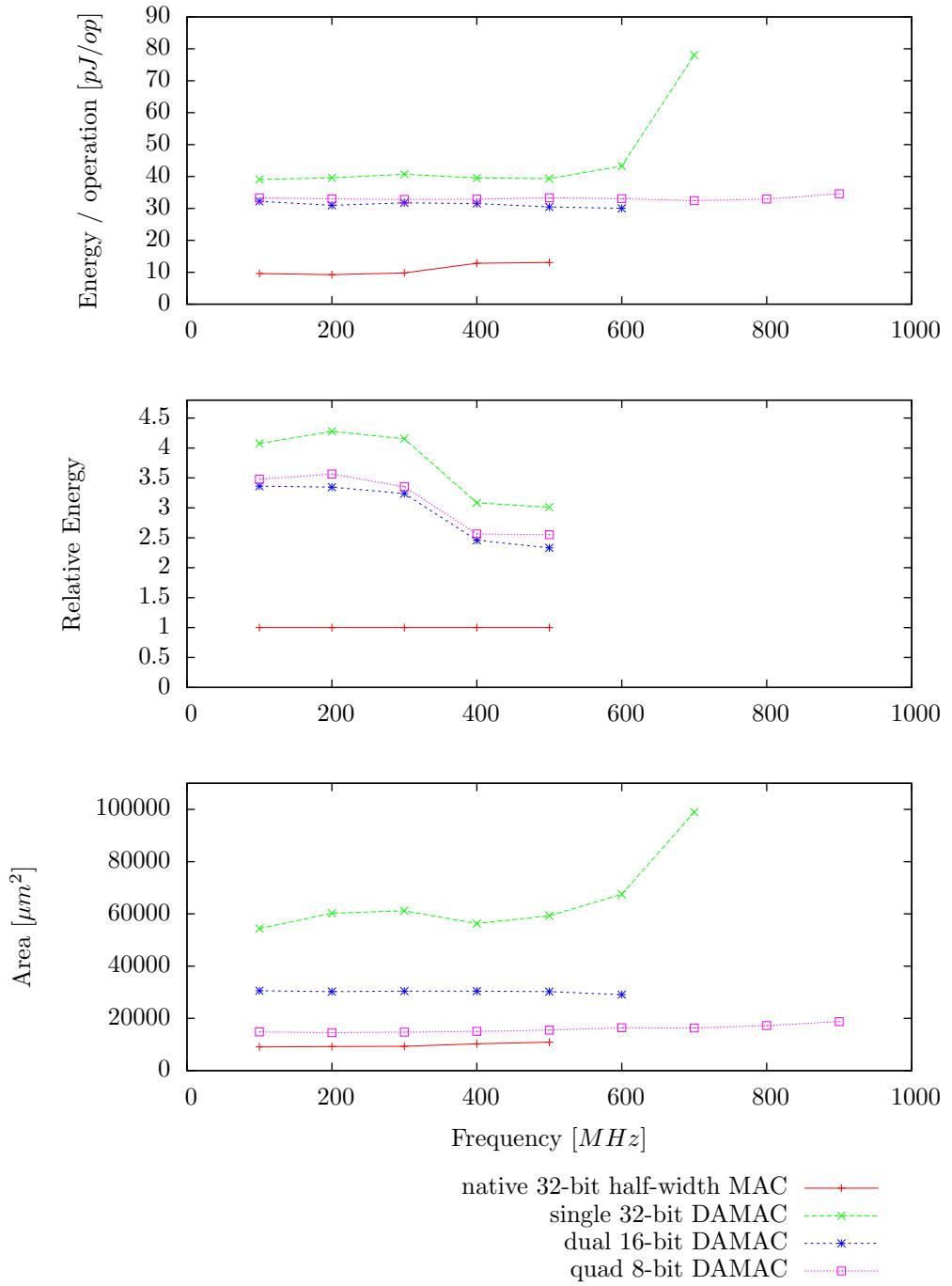
The results for half-width 32-bit multiply-accumulation are in Plot 8.6. The single 32-bit configuration uses almost the same amount of energy as the full-width case, as the functional unit is only capable of full-width multiplication; the only difference with the full-width design is that the upper half of the accumulator can be excluded. The smaller the functional units become, the less unused calculations are performed, and that is why the 16-bit and 8-bit configurations are close to each other in terms of energy. The area usage of the 8-bit configuration is not much more than the baseline; the 16-bit configuration uses 3 times as much area, and the 32-bit configuration uses 5–6 times as much area as the baseline.

When smaller, 8-bit, products are accumulated, the results are different. The full-width results are shown in Plot 8.7. The results for the single 32-bit configuration are the same as in the first plot; about the same energy usage, and a much higher area usage. The 16-bit configuration uses only 20–40% of the energy of the baseline, and the 8-bit configuration is even more energy efficient, using 6–13% of the energy of the baseline, while using only 10–30% of the area of the baseline.

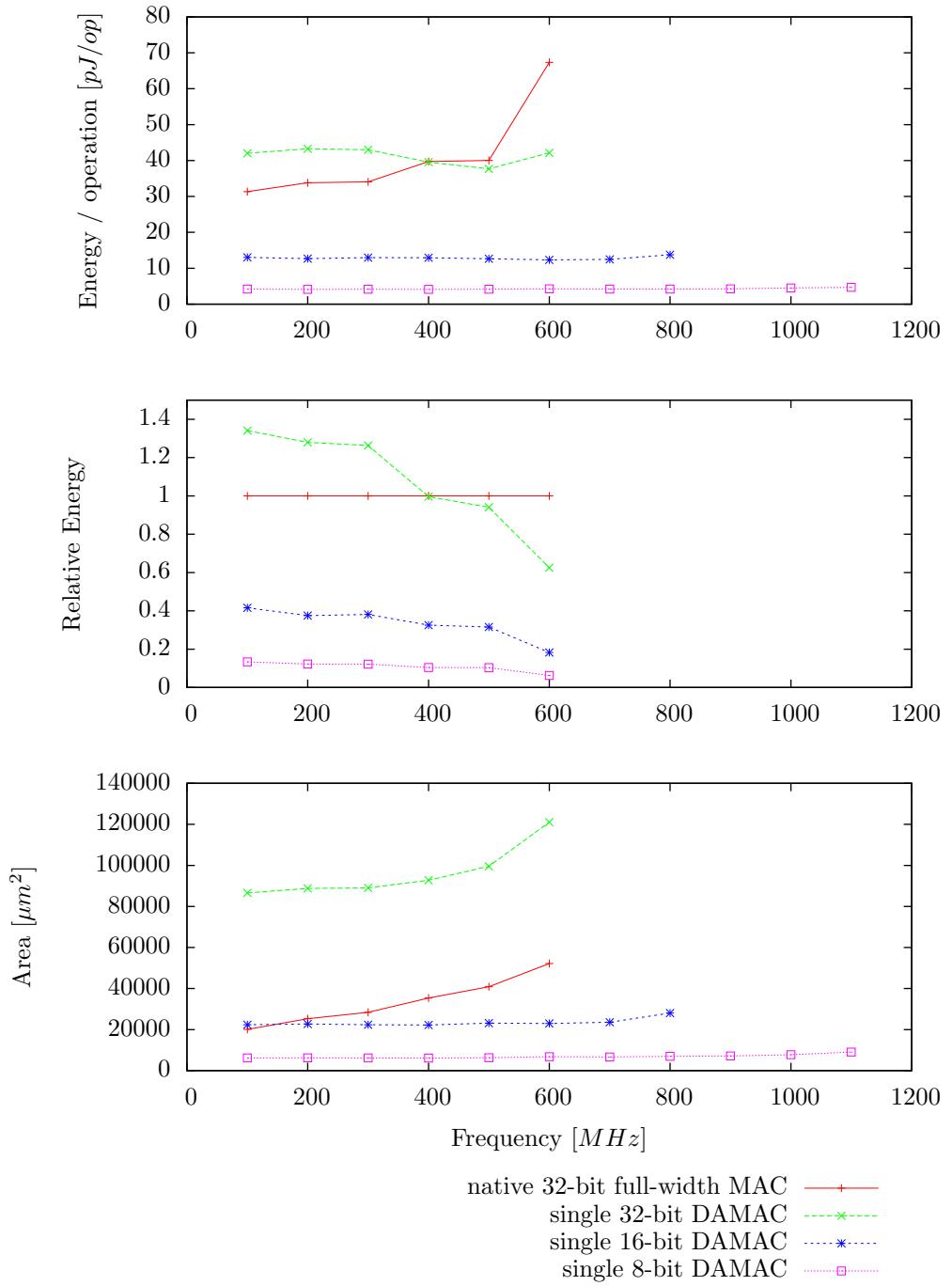
Finally, Plot 8.8 shows the results for half-width 8-bit multiply-accumulation. Just like in the 32-bit case, the single 32-bit configuration uses a lot more energy than the baseline; 3–4 times as much. The 16-bit functional units perform similar to the baseline, although they are able to operate at a much higher frequency, up to 900 MHz. The 8-bit configuration uses 30–40% of the energy of the baseline, and is able to achieve operating speeds up to 1300 MHz, without a significant increase in energy usage.



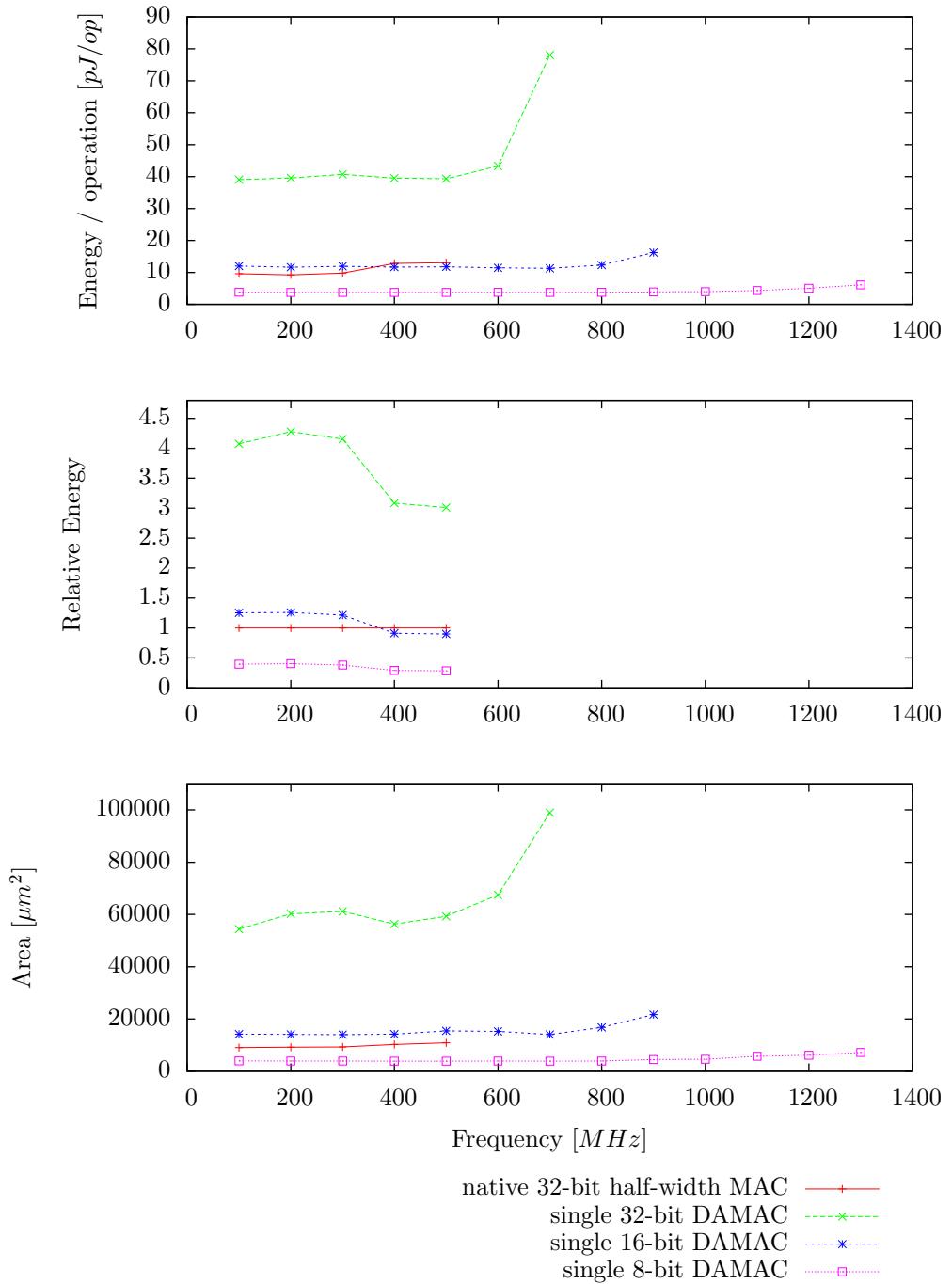
Plot 8.5: Dual-cycle 32-bit full-width accumulator-based multiply-accumulation.



Plot 8.6: Dual-cycle 32-bit half-width accumulator-based multiply-accumulation.



Plot 8.7: Dual-cycle 8-bit full-width accumulator-based multiply-accumulation.



Plot 8.8: Dual-cycle 8-bit half-width accumulator-based multiply-accumulation.

### 8.3.3 Distributed Multiply-Accumulator

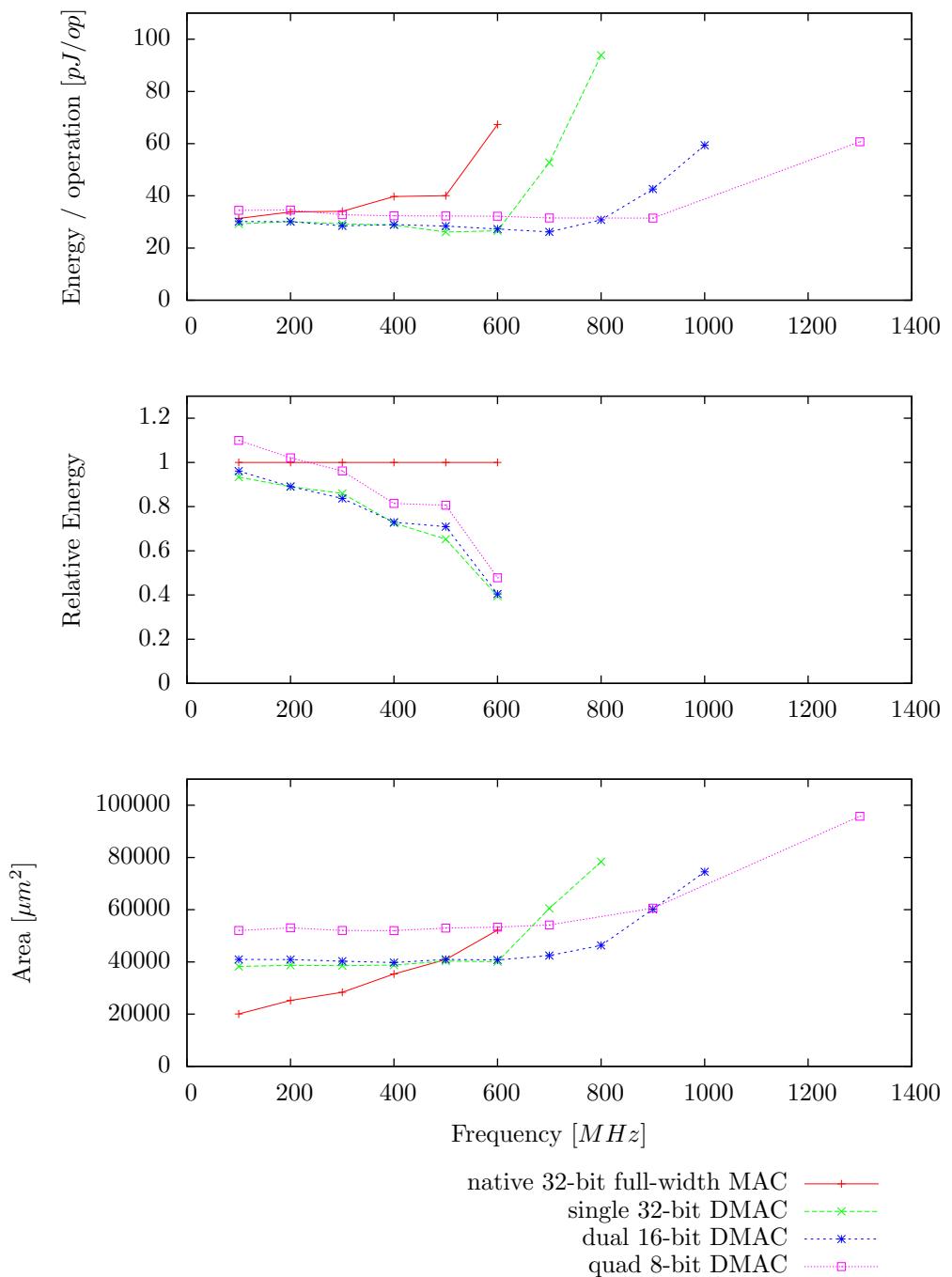
The last design to discuss is the *Distributed Multiply-Accumulator* (DMAC), as discussed in Section 8.2.2.

Again, we start with the full-width 32-bit multiply-accumulation results in Plot 8.9. This plot shows that the chosen granularity has little effect on the energy efficiency; up to 600 MHz, the quad 8-bit configuration consumes only about 10% more energy than the other two configurations, which are very close to each other. At higher clock frequencies, the energy usage of the single 32-bit configuration increases quickly, while the quad 8-bit configuration stays nearly constant at 30 pJ/op up to 900 MHz. The area usage is a bit higher than the baseline at lower frequencies, but the baseline area increases quickly, crossing the 32-bit and 16-bit configurations at 500 MHz.

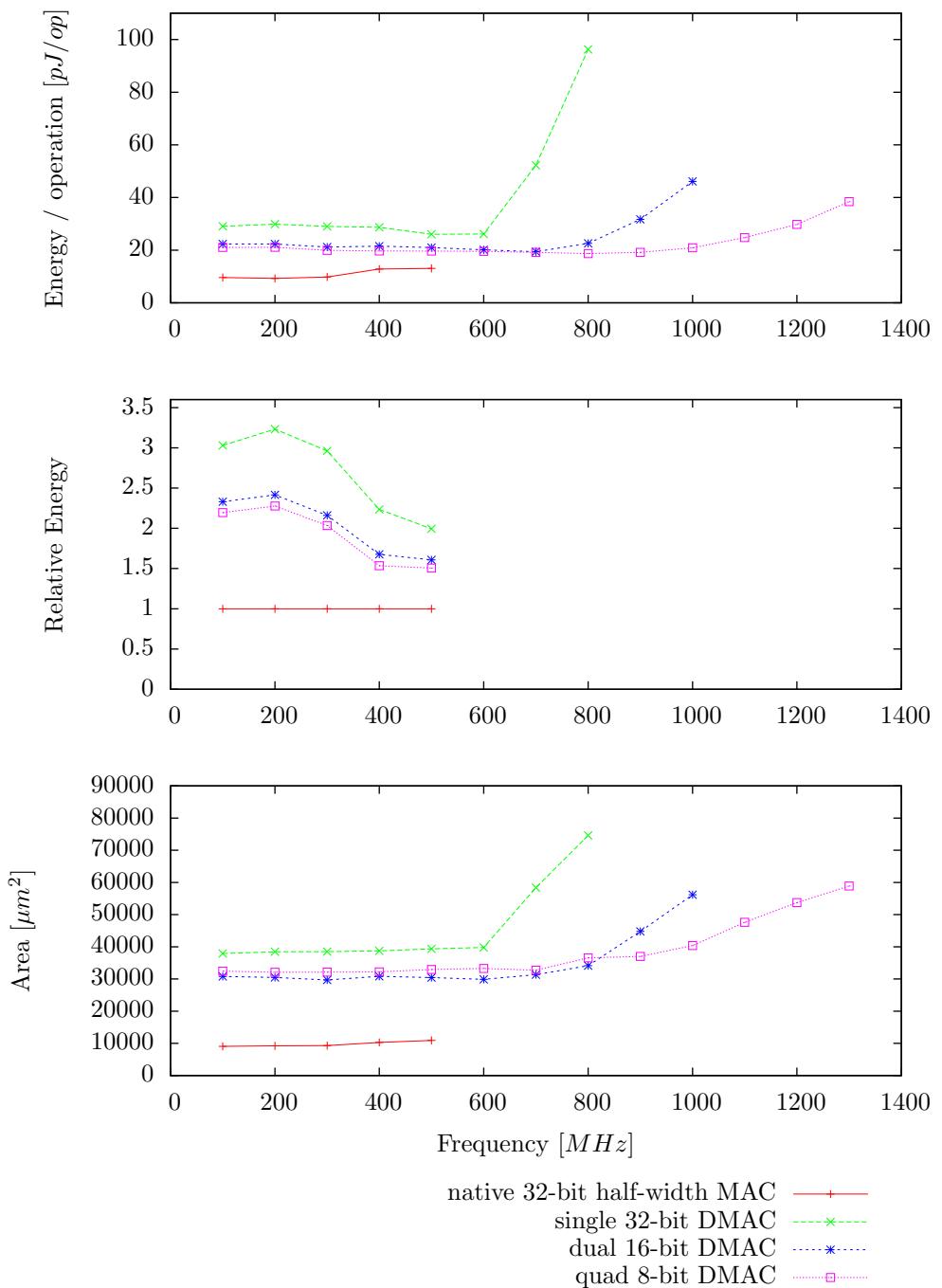
The half-width 32-bit results are shown in Plot 8.10. The difference between the 32-bit configuration and the 16-bit and 8-bit configurations is now more pronounced, to the advantage of the smaller functional units; the quad 8-bit configuration is now the most energy efficient configuration, as this configuration has the least unused calculated bits. The quad 8-bit configuration is also able to reach the highest operating frequencies, up to 1300 MHz. At 600 MHz, the dual 16-bit and quad 8-bit configurations use 50% more energy than the baseline; the 32-bit configuration uses twice the energy of the baseline. All configurations use 3–4 times as much area as the baseline.

The full-width 8-bit multiply-accumulation results are visualised in Plot 8.11. The 32-bit configuration is slightly better than the baseline in terms of energy per operation. The 16-bit configuration uses 10–25% of the energy of the baseline, while being able to operate at up to 1100 MHz, and using 20–50% of the area. The 8-bit functional units perform even better, using 3–7% of the energy of the baseline, while being able to operate up to 1400 MHz without significant increase in energy per operation: 2.2 pJ/op at 100–1000 MHz, and 4.3 pJ/op at 1400 MHz. All this while using far less area; only 6–20% of the area of the baseline.

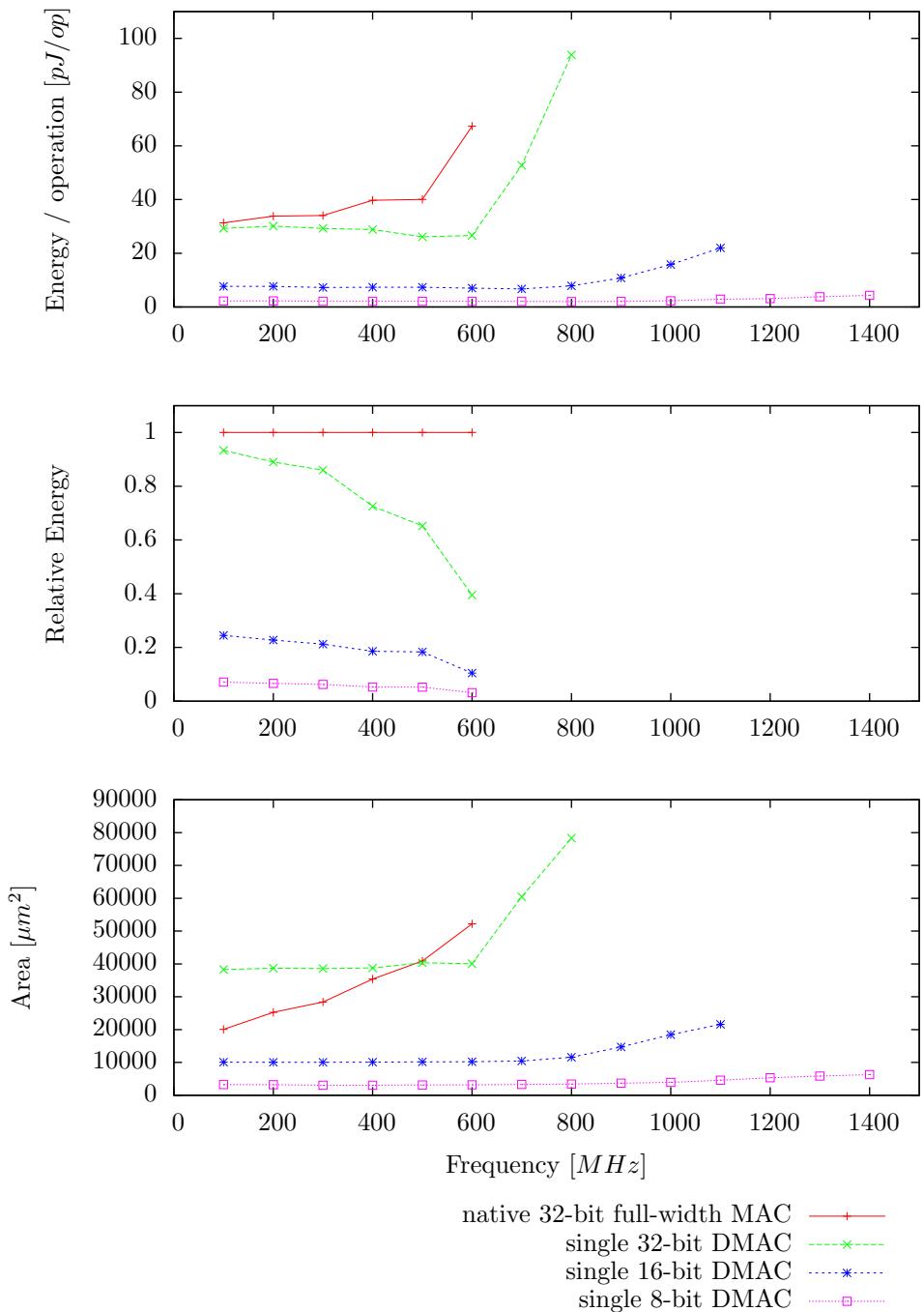
The half-width 8-bit multiply-accumulation results are shown in Plot 8.12. The relative performance of the three designs is the same as the previous, full-width, configuration, as each functional unit can only calculate full-width results. And as such, the relative performance compared to the baseline is worse than for the full-width configuration; the 32-bit configuration uses 2–3 times as much energy as the baseline, the 16-bit configuration uses 50–80% of the energy of the baseline, and the 8-bit configuration uses 15–20% of the energy of the baseline.



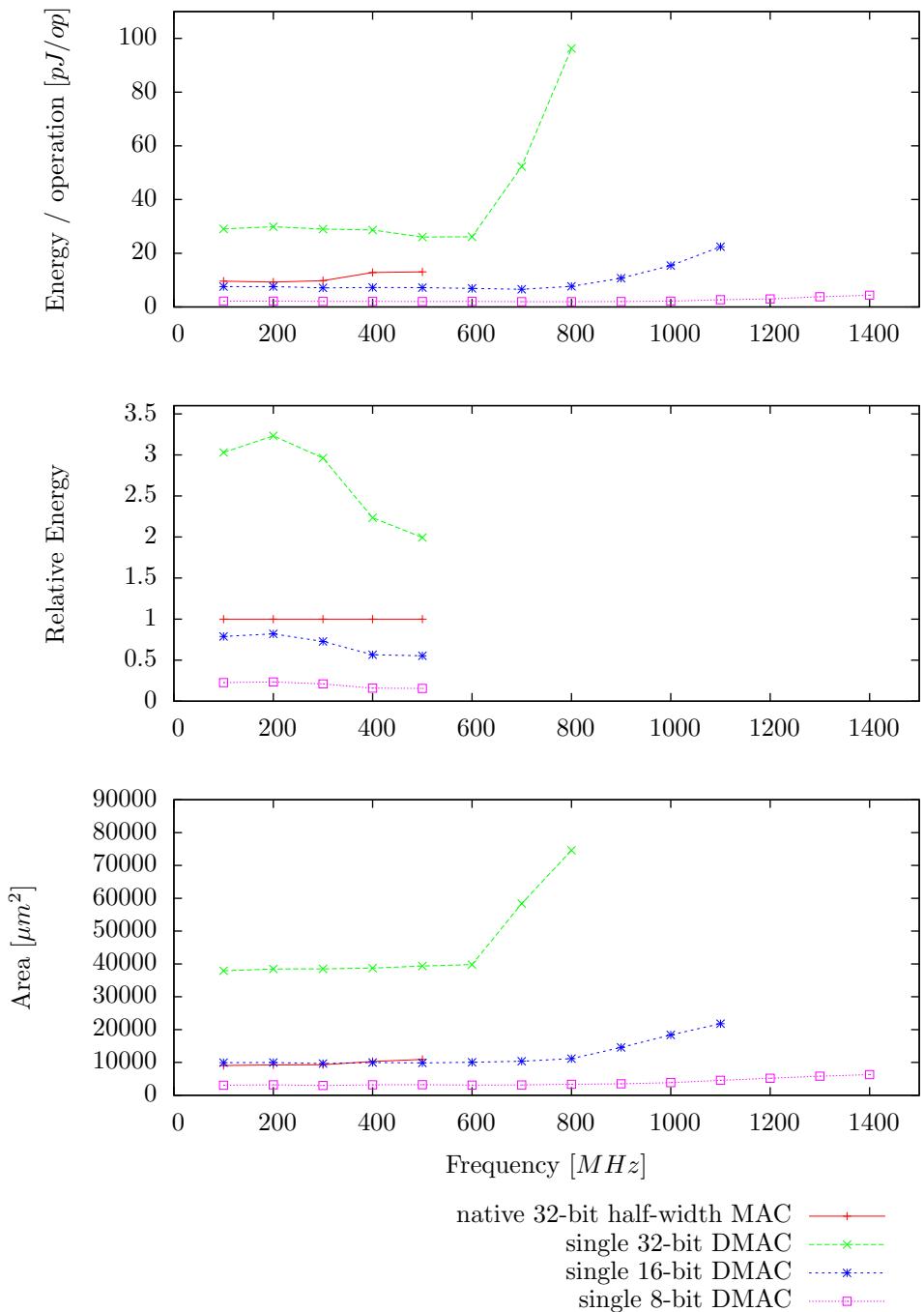
Plot 8.9: 32-bit full-width distributed multiply-accumulator.



Plot 8.10: 32-bit half-width distributed multiply-accumulator.



Plot 8.11: 8-bit full-width distributed multiply-accumulator.



Plot 8.12: 8-bit half-width distributed multiply-accumulator.

## 8.4 Comparison

Now that we have discussed all designs — the single-cycle accumulator-based multiply-accumulator (SAMAC), the dual-cycle accumulator-based multiply-accumulator (DAMAC) and the distributed multiply-accumulator (DMAC) — individually, it is time to compare them against each other. First we compare the full-width and half-width 32-bit multiply-accumulation in order to determine the overhead that the multi-granular decomposition entails. Secondly, we have a comparison of full-width and half-width 8-bit multiply-accumulation to learn how much can be gained from the multi-granular design.

The full-width 32-bit multiply-accumulation results are shown in Plot 8.13. We see that the energy efficiency depends in the first place on the used design, and secondly on the granularity; the DMAC designs have the best energy efficiency, followed by the SAMAC and DAMAC designs respectively. The distributed multiply-accumulators can achieve the highest frequencies, up to 1300 MHz.

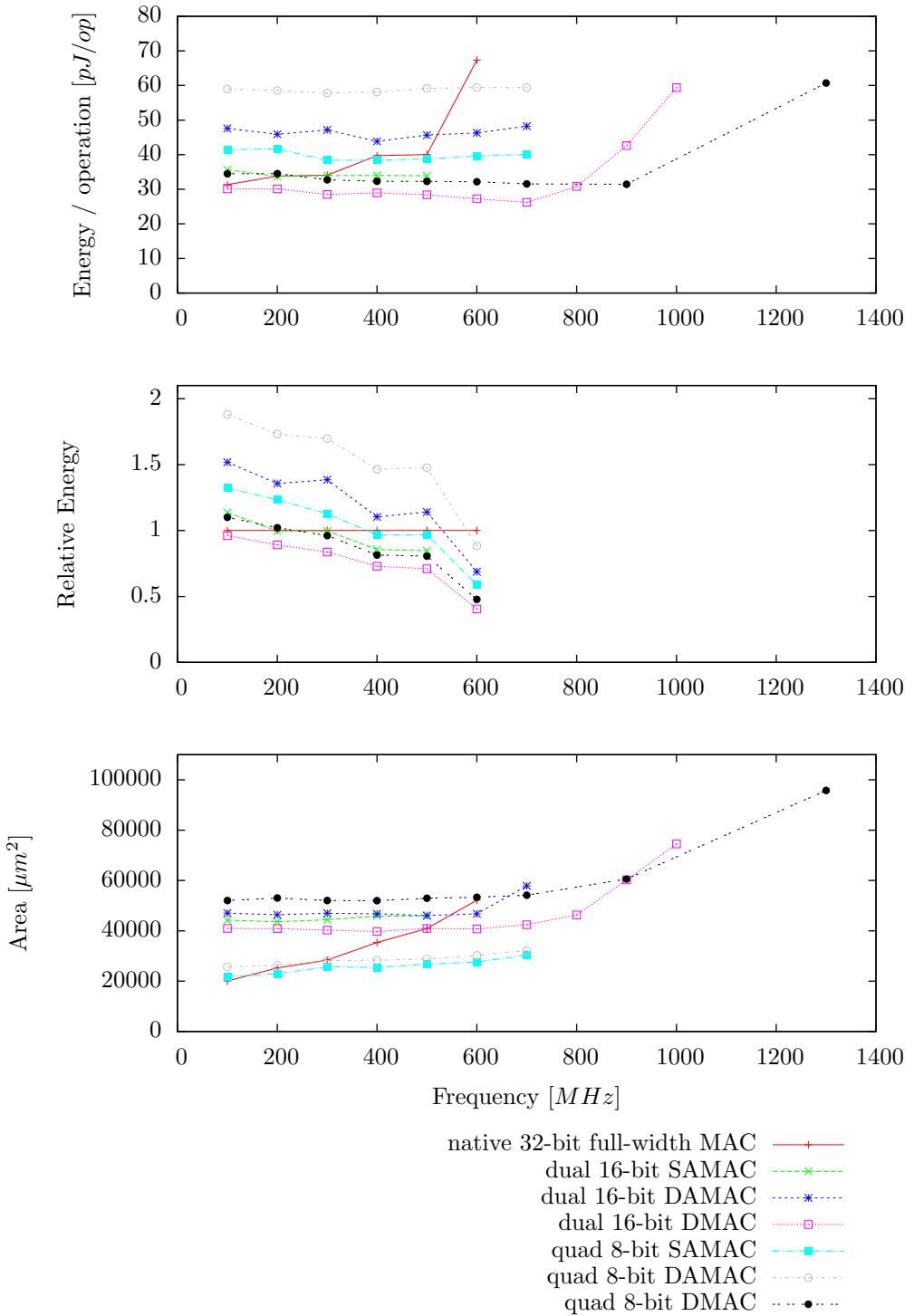
For half-width 32-bit multiply-accumulation, as shown in Plot 8.14, the chosen granularity does not seem to have an impact on the energy efficiency; both 8-bit and 16-bit units have similar performance. Also the DMAC and SAMAC designs have a similar energy usage, while the DAMAC designs uses more energy.

The results for 8-bit full-width multiply-accumulation are shown in Plot 8.15. Here we see that the 8-bit functional units all use less energy and area than the 16-bit variants, which is not unexpected. The energy efficiency of DMAC and SAMAC is similar, using 3–7% of the energy of the baseline with 8-bit functional units, and 10–25% for 16-bit functional units. Both DMAC configurations, however, are able to reach higher operating frequencies and use less area.

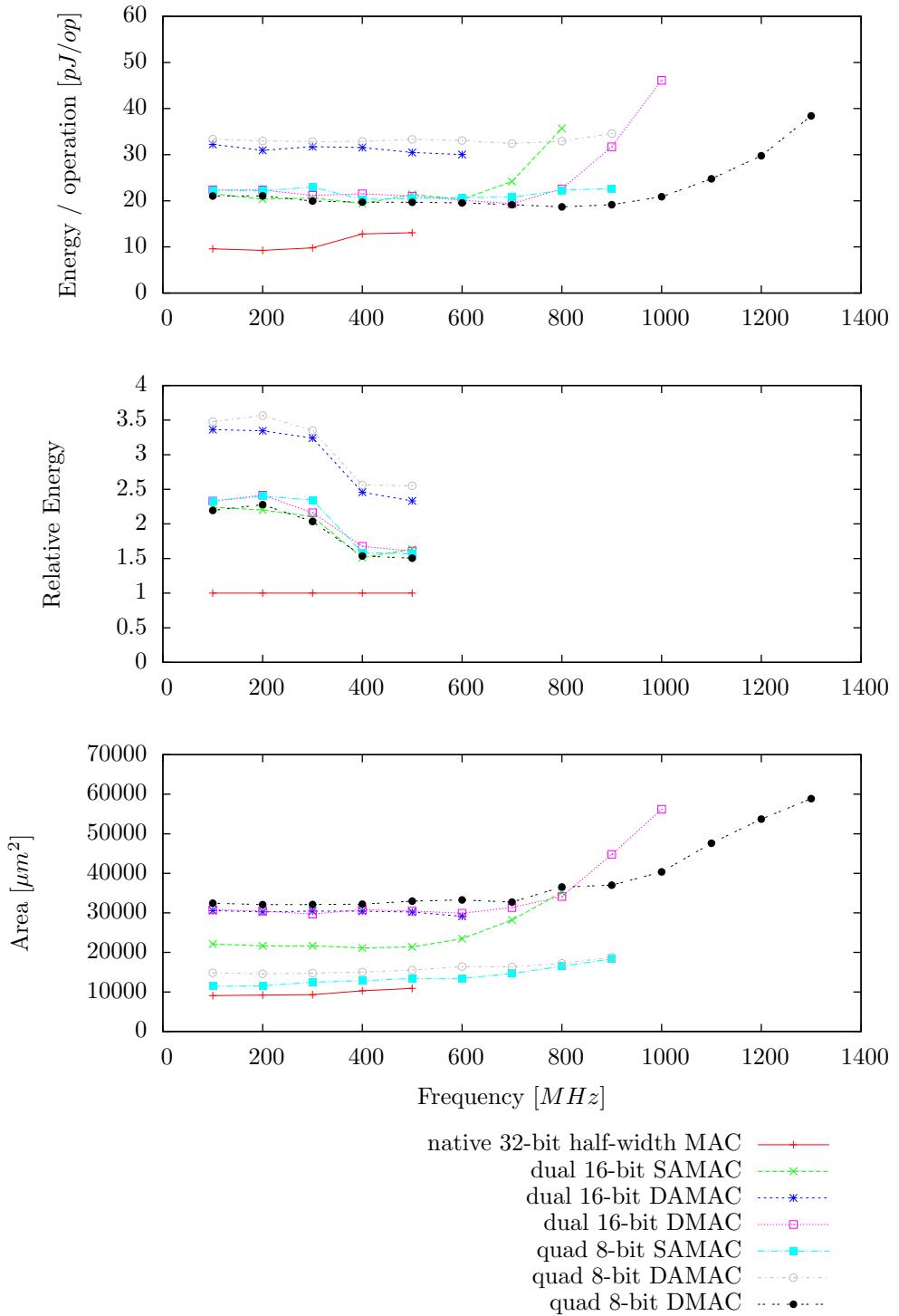
The half-width 8-bit multiply-accumulation results are shown in Plot 8.16. Just as for the full-width case, the 8-bit configurations perform significantly better than the 16-bit functional units. As our multipliers are unable to perform half-width multiplication, the energy gain is not as large as for the full-width case: the best designs, DMAC and SAMAC, still use 15–20% compared to the baseline for 8-bit functional units, and 50–80% for the 16-bit functional units. The area usage of all 8-bit configurations is nearly identical, while for the 16-bit designs, the DMAC variant is significantly smaller than the other two designs.

Plot 8.15 shows the results for quad 8-bit full-width multiply-accumulation. Again, the DMAC uses the least amount of energy per operation, followed by the SAMAC, which uses about 20% more energy; the DAMAC uses 70% more energy than the DMAC. The energy efficiency of the DMAC comes at a large cost in terms of area usage, which is twice as much as the other two designs.

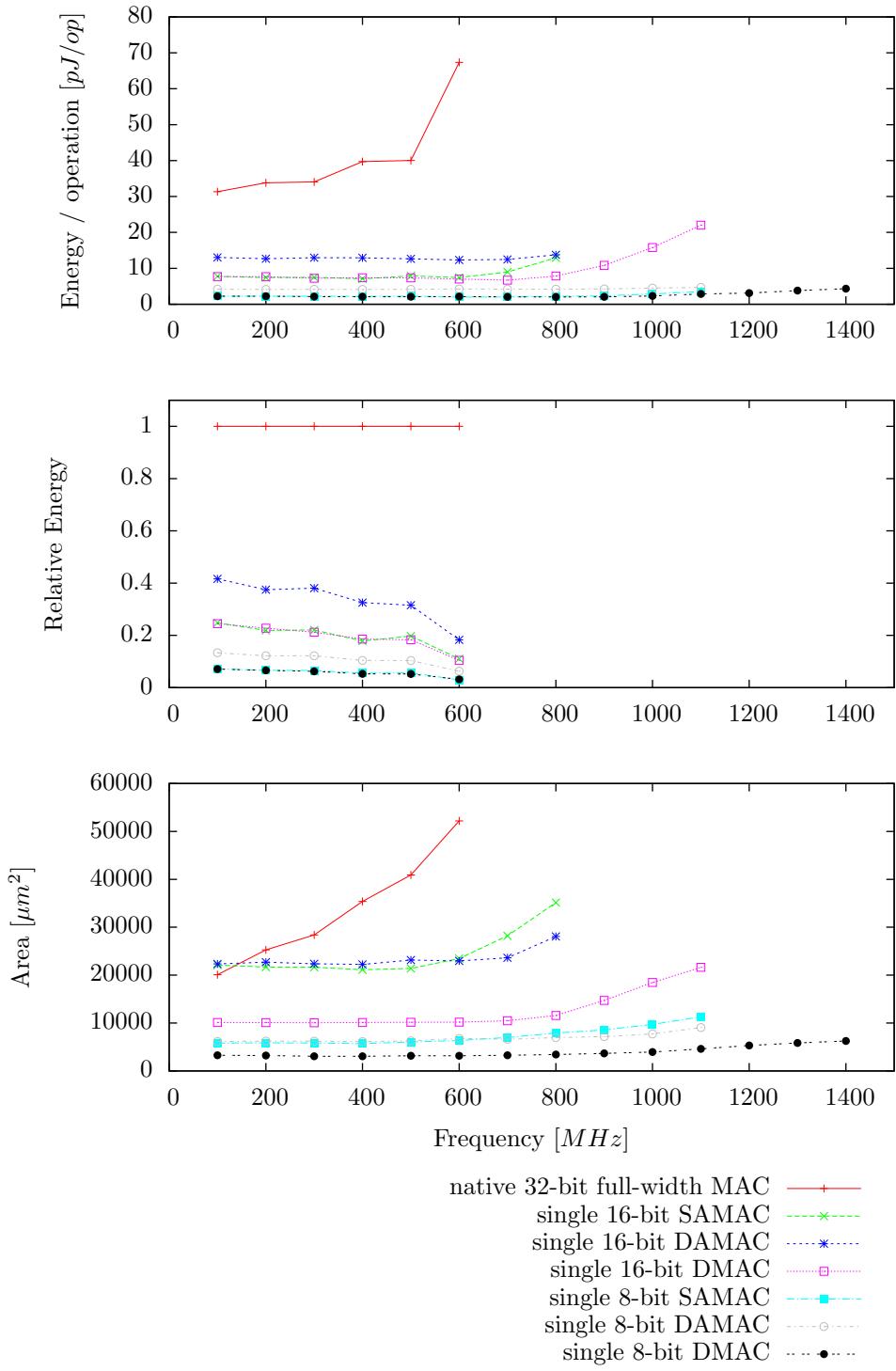
Finally, Plot 8.16 shows the half-width multiply-accumulation results for the quad 8-bit configurations. The DMAC and SAMAC designs perform similar in terms of energy, although the DMAC design is able to reach a much higher operation frequency; 1300 MHz versus 900 MHz for the other two designs. The DAMAC design uses 50% more energy per operation than the other two designs. However the DMAC design uses 2–3 times as much area as the other two designs.



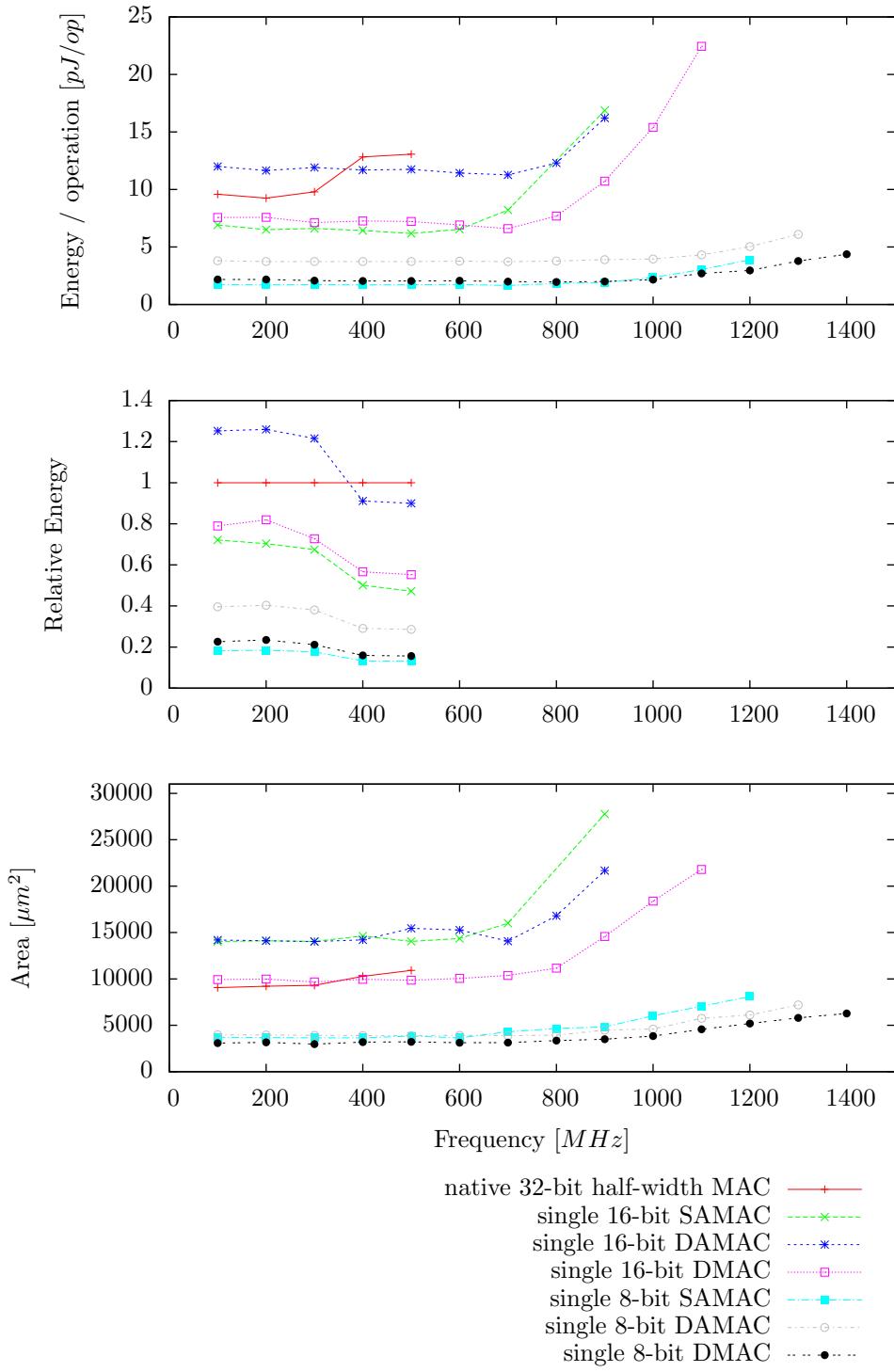
Plot 8.13: 32-bit full-width multiply-accumulation using 8-bit and 16-bit functional units.



Plot 8.14: 32-bit half-width multiply-accumulation using 8-bit and 16-bit functional units.



Plot 8.15: 8-bit full-width multiply-accumulation using 8-bit and 16-bit functional units.



Plot 8.16: 8-bit half-width multiply-accumulation using 8-bit and 16-bit functional units.

## 8.5 Conclusions

Based on the analysis of the benchmark results in the previous two sections, we can draw some conclusions and determine which design is to be preferred.

Looking at energy per operation, the distributed multiply-accumulator is a good choice. In all tested configurations, it performed either the best, or was very close to the best design. Additionally, the load on the interconnect is low.

Single-cycle accumulator-based multiply-accumulation is a good second choice; it performs similar to the distributed multiply-accumulator when it comes to energy usage per operation for half-width operation, and it required about 10–20% more energy per operation for the full-width operations.

The dual-cycle accumulator-based multiply-accumulation uses the most energy, but requires only a single output connection to the interconnect.

The distributed multiply-accumulator has a high throughput — 1 multiplication per cycle — but it does require a few extra cycles to accumulate all results: twice the word-width for half-width operations and four times the word-width for full-width operations, making it expensive for accumulations of only a few products.

The multipliers-accumulators that are based on the multiplier with accumulator design require multiple cycles per multiplication step. This makes this multiply-accumulator rather slow if many products have to be accumulated.

The advantage of the multiply-accumulators with accumulator is that no further changes are needed to the functional units; it is constructed using only multiplier and accumulator units from the previous chapters. The throughput of these multiply-accumulators can be increased by placing adders in front of the accumulators; each layer of adders halves the number of cycles, while adding a single cycle to the final accumulation phase, as the depth of the accumulator is increased. It is even possible to increase the throughput to more than one multiplication per cycle, if the number of available functional units allow this, and the input values can be supplied fast enough.

The multi-granular design has a positive impact on the distributed multiply-accumulator, as this design is capable of operating at higher frequencies. Both multiply-accumulation with accumulator designs have an increased energy usage. This penalty is 1.5–2 times the baseline for the full-width multiplication, and 2–4 times the baseline for half-width multiplications. The gain from the smaller functional units is again large: when 8-bit multiply-accumulation can be performed on 8-bit functional units, then the gain compared to the baseline is a factor 30 for the single-cycle accumulator-based MAC and the distributed multiply-accumulator, and a factor 15 for the dual-cycle accumulator-based MAC.

# Chapter 9

## Design for the Application

In the previous chapters, the individual operations — addition, accumulation, multiplication, and multiply-accumulation — were discussed. For each operation, several alternative implementations are introduced, and each implementation was evaluated for 8-bit, 16-bit and 32-bit wide functional units. All these configurations where benchmarked for 32-bit, 16-bit and 8-bit wide operations. Although the 16-bit wide operations were not presented in the previous chapters, their energy and area number are available in Appendix A.

Some configurations performed better than the native baseline, while some other configurations used more energy and/or area than the baseline. Thus we cannot simply conclude that multi-granular arithmetic operations are the solution for every problem, or that they should be avoided at all costs. This depends on the application; if all operations in an application have to be 32-bits wide, then it does not make sense to use multi-granular operations when designing an architecture for that application. On the other hand, if most operations are 8-bit or 16-bit, with only a few 32-bit wide operations, then a multi-granular architecture can save a lot of energy.

However, most applications are somewhere in between these two extremes, making it not so easy to decide what would be the best architecture right away. To support the quest for the most energy efficient architecture, some instruments are developed to sort the configurations for a specific application.

### 9.1 Granularity Model

Ideally, all operations are executed at exactly the width that is needed for the operands. However, supporting all operation sizes has high overhead costs. Therefore, we need to balance the overhead costs of smaller block sizes against the costs of operations that are performed wider than needed. In order to decide what granularity to use, we need to develop a model of the application, or

application domain, containing the distribution of operations, and their required operating width.

This distribution of operations can be obtained by profiling the target applications, or by analysing the applications. However it is important that this is done on an application that is already optimised to use the lowest possible number of bits; i.e. if it is known that an operand never uses more than 8 bit, then this should be modelled as such in order to accurately determine the potential gains of a multi-granular architecture.

This distribution can be used to quickly estimate the energy usage for the application on a certain configuration, by mapping each operation to the most energy efficient operation that is available in the chosen configuration. The energy usage of the best operation is multiplied by the number of times that the operation occurs in the application. By adding these energy numbers for all operations, the energy usage for the application on the used hardware configuration is estimated.

Let  $\mathbb{O}$  be the set of possible operations that can be performed by our proposed architectures. For application  $a$ , we can define  $u_a(o)$  as the number of times that application  $a$  executes operation  $o \in \mathbb{O}$ . Additionally, for architecture configuration  $c$  we define  $e_c(o)$  as the energy that operation  $o \in \mathbb{O}$  uses when executed on architecture  $c$ . With this, we can define the function  $E(a, c)$  as the energy used to execute application  $a$  on architecture configuration  $c$ , as shown in Formula 9.1.

$$E(a, c) = \sum_{o \in \mathbb{O}} u_a(o) \times e_c(o) \quad (9.1)$$

As said, the set of operations and their counts  $O_a$  has to be determined by benchmarking or analysing the application. The energy number for the operations on different architecture configurations  $e_{o,c}$  are available in Appendix A for the operations and architectures, including the different granularities, discussed in this report.

## 9.2 Interconnect

However, this model does not take the interconnect into account. As the choice for the interconnect is left open, the energy usage for the interconnect has to be estimated using some other means.

If we define the energy that is used to pass a single bit through the interconnect of architecture configuration  $c$  as  $i_c$ , then it is possible to estimate the energy used by the interconnect for operation  $o$  by multiplying  $i_c$  with the number of bits  $p_c(o)$  that pass through the interconnect to perform operation  $o$  on architecture configuration  $c$ . The values for  $p_c(o)$  are also given in Appendix A for the operations and configurations that are discussed in this report. Combined with the previous version, the model can be extended to Equation 9.2.

$$E'(a, c) = \sum_{o \in \mathbb{O}} u_a(o) \times (e_c(o) + i_c \times p_c(o)) \quad (9.2)$$

## 9.3 Comparing Architectures

The energy number can be computed for different configurations. By comparing these numbers, one can easily determine the most energy efficient configuration for the application.

It also determines the most energy efficient granularity, as the granularity is one of the parameters — possibly the most important — of the configuration. It will also provide an insight into the performance of the different implementations of the operations, as they all have different trade-offs.

## 9.4 Improving the Model

The current model is a first-order approximation; many aspects of the architecture and application are not taken into account. This can be improved on by adding new aspects to the model if more precision is required, at the cost of a more complicated model. The following list of additions can be considered to improve this model.

The interconnect network is not completely ignored, but it is extremely simplified, as it only measures the number of bits passing through the interconnect. It does not measure the carry bits differently, these are just counted as single bits. It also abstracts from the distance the data has to travel; in many interconnect networks, the energy used highly depends on the distance that the data has to travel [19].

The aspect of time is also left out, as this depends on many factors: the number of available functional units, the amount of parallelism available in the application, the scheduling of the operations on these functional units — basically a compiler — and the number of register files. This also means that the time required to execute an application is not known, and thus we cannot determine the throughput of the architecture.

Also ignored are the other aspects of the architecture, such as memory accesses, which use energy and delay the execution of the application, and the energy used by the register files. Additionally, instruction fetching and decoding is ignored; some arithmetic algorithms require cycle-time reconfiguration of the operations executed by the functional units (for example the multiplier with accumulator from Section 7.3.3), while others can load an instruction once, and use it for many cycles (for example the multiplier tree from Section 7.3.1).

This does not make the previous model useless — it can still be used to quickly determine which architecture configurations and granularities are interesting — but it should only be used as a tool to compare the configurations and determine the next steps, not to estimate the actual energy usage of the chip.

## Chapter 10

# Conclusions and Future Work

In this report, we have developed multiple techniques for multi-granular arithmetic operations for addition, add-accumulation, multiplication, and multiply-accumulation. Using these algorithms, it is possible to perform operations than are a multiple of the word-size of the architecture wide, by combining multiple operation blocks together. Using these techniques, energy is no longer wasted on operations that are unnecessary large.

Some energy is wasted on the composition of these smaller operation blocks, and as such, there is a trade-off between the energy that can be saved when executing narrow operations on one hand, and energy that is needed to perform wider operation on the other hand. In this report, for all developed algorithms, we examined the energy savings during the execution of 8-bit wide operations, and the energy overhead when performing 32-bit wide operations.

In general, the concept of multi-granular arithmetic is feasible: it is possible to compose wider operations without too much overhead, while narrower operations could be performed with significantly less energy per operation.

For addition, the ripple-carry interconnect combined with the carry-lookahead adder inside the functional units performed the best. This design has nearly no overhead when 8-bit functional units where composed to perform 32-bit wide operations, while 8-bit operation could be performed with 80% less energy compared to a native 32-bit baseline.

Accumulation again performed the best with the ripple-carry interconnect, while both the carry-lookahead algorithm as well as the carry-save algorithm performed well as base algorithm; the choice between those two depends on the required operating frequency. Composing a 32-bit accumulator from 8-bit functional units results in a 5% overhead compared to a native baseline, while a single 8-bit functional unit can save 85% compared to a 32-bit native baseline.

Multiplication is not as straightforward; both the adder tree design as well as the accumulator design have their strong points. Luckily, this choice can be left to the compiler, which can choose compile-time which design is most suitable for each situation. While multiplication does use 40% more energy for overhead costs to compose a 32-bit multiplication from 8-bit functional units, the gains for smaller operations are also larger: an 8-bit multiplication can save 94% of the energy compared to a native 32-bit baseline.

The last operation we have discussed is multiply-accumulation. For this operation, most of the overhead costs can be moved to the finalisation phase, which is only executed once for each accumulation series. The distributed multiply-accumulator design was able to compose a 32-bit multiply-accumulator from 8-bit functional units with negligible overhead costs, whilst saving 97% when 8-bit multiply-accumulate operations are performed compared to a native 32-bit baseline.

The highest penalty that the multi-granular design technique imposes on the architecture is the added cycles to perform multiplication, resulting in an increased delay before the results are available. Whether this is imposes a performance impact on the application depends on the available parallelism and data dependencies of the application.

The BLOCKS architecture is very suited for multi-granular arithmetic operations, as it consists of a large grid of small functional units, which can be composed to form arbitrary operations by means of the reconfigurable interconnect; such as operations that are wider than the word size.

## 10.1 Optimising Code for Multi-Granular Architectures

In this report, we investigated what the costs would be to compose a wide operation from smaller functional units. In particular, we looked at 32-bit operations composed from 8-bit and 16-bit functional units. While it is certainly good that our architectures are able to perform those 32-bit wide operations efficiently, most operations in a typical program do not really need such wide operations.

A typical program written in any high-level programming language will most likely use whatever data-types are generally available on the architecture, such as the generic `int` data-type. On desktop machines, this most likely translates to 32-bit or 64-bit wide variables, which makes sense, because this is the only data-width that a typical desktop CPU can process. However a multi-granular architecture can perform these operations at the width that is actually required, if the compiler is able to extract this width information.

For example, if a variable that fits in an 11 bits wide number would be multiplied by the constant number 5, then ideally on an architecture with 8-bit functional units, this should result in a multiplication that is 16-bits times 8-bits wide, resulting in a 16-bits number, as the resulting number is at most 14

bits wide. Multiplication requires a quadratic amount of work in terms of the input operand size — it is linear in the size of each input operand  $w_a$  and  $w_b$ :  $\mathcal{O}(w_a \times w_b)$ . Therefore, this multiplication is roughly 8 times as energy efficient as a native half-width 32-bit wide multiplication.

Therefore, this multi-granular design will only reduce the energy usage of an application if operations are executed at their minimal required width; if each operation is executed in 32-bit, there is not much use for a multi-granular architecture, and an architecture that is not based on this multi-granular technique will be more energy efficient.

These optimisations could be made manually, but this is tedious work; a better approach would be to adapt the compiler such that it takes these considerations into account. The compiler should have enough bookkeeping in order to pick the optimal width for each operation. The compiler can determine the maximum number of bits after performing an operation if the width of the inputs is known. For input widths  $w_a$  and  $w_b$ , the maximum width after addition is  $\max(w_a, w_b) + 1$ , while the maximum width after multiplication is  $w_a + w_b$ . However, it cannot always determine the width of a variable, therefore, the programmer also has to specify the width of variables where possible, e.g. it could specify the width of input variables, or the maximum width of a variable after (multiply)-accumulation.

Without these optimisations on the application side, a multi-granular architecture will not perform substantially better, or even worse, than a non-multi-granular architecture.

## 10.2 Interconnect Considerations

In this report, we decided to keep the choice for the design of the interconnect open, and thus the exact features, and the costs in terms of energy, are not known yet. Here we will discuss the features that might be relevant, or worth considering, for an interconnect of a multi-granular architecture.

In order to decompose addition, some form of carry propagation is required. The simple version of the carry-chain support the ripple-carry algorithms as discussed in Section 5.1.1 and has a good performance, and is easy to implement. It is sufficient to make a single-bit wide connect between adjacent functional units; these signals do not have to be routed. If one wants to use the carry-select adder as discussed in Section 5.1.3, then this carry-chain interconnect has to be extended such that it can select the right carry-bit and result based on another carry bit. Other carry-interconnect mechanisms, such as a carry-lookahead interconnect, might also work, but this is left as future work.

For multiplication, we have discussed two base multipliers in Section 7.1.2: the single-cycle and the dual-cycle variant. The choice between these two depends on the capabilities of the interconnect; if each functional unit only has a single output port, then the dual-cycle multiplier has to be used. The single-cycle

variant — which was more energy efficient, and has a lower delay — required two output ports to the interconnect, in order to produce the whole, full-width, product in a single cycle.

In order to support the multiplier algorithm based on the accumulator, as discussed in Section 7.2.5, the interconnect must be able to send the result of the multipliers to another accumulator each cycle. This implies that the interconnect must support runtime reconfiguration of the data-paths, in contrast to a static configuration-time configured interconnect.

As discussed in Section 7.2.4, wide full-width multipliers based on the adder tree algorithm requires a few extra adders in order to accumulate the carry bits. This can be circumvented if adders can accept the carry-in bit from not one, but two functional units. However, as this can lead to the production of two carry-out bits, the carry interconnect network should no longer transport a single bit, but a pair of two bits between the functional units. Together with some adjustments to the adder circuit, this can lead to a reduction of the number of adders needed for wide full-width multiplications.

Sometimes, the result of an operation is not directly needed, but one or two cycles later. This happens in the adder tree multiplier as discussed in Section 7.2.4, but it could also happen during the construction of other complex operations. These values could temporary be saved in a register file, until they are needed for further processing, of a functional unit could be used to perform the “identity” operation, i.e. just forward the input, to delay the result a cycle. However it might be more energy efficient if these values could be delayed by a single cycle by the interconnect, or by a dedicated delay unit, such that no functional unit has to be wasted, and a trip to the expensive register file can be spared.

If one wants to use the carry-save format, as discussed in Section 6.1.2, as a transport mechanism between different functional units, then the interconnect has to be able to transport results in carry-save format. This was not investigated in this report, as it is expected that most of the energy for this algorithm would be used in the interconnect; the carry-save adders itself are very energy efficient. If an implementation of an interconnect with dual-width connections is available, or at least some numbers on the energy usage of this interconnect, then this adder technique can be reevaluated.

Once a choice has been made on the exact form of the interconnect, based on these and other considerations, the benchmarks from this report should be reevaluated in order to provide more accurate energy and area numbers. The benchmark results from this report are useful to choose the interconnect, however they are not generally applicable or comparable against other architectures.

## 10.3 Future Work

In this report, we have investigated how elementary arithmetic operations, and operations that are frequently used in the signal processing domain, can be composed from small building blocks in a multi-granular way. However, we did not cover all operations that might be needed for our application domain. Likewise, we discussed some of the straightforward design variations, but many more variations are possible. Here we will look at some ideas that we think are interesting for a follow-up research.

We have discussed addition and multiplication, but for there inverse operations, subtraction and division, we did not yet develop a multi-granular variant. In order to make a full-featured multi-granular architecture, these operations should also be added. Other operations that might be interesting are square root, modulo and negation.

Another feature that might be of interest is floating point support. As it is already possible to make a small adder for the exponent, and a multiplier for the significant, the open problem here is combining these two parts, and applying correct rounding to the result. The significant and exponent can both be a multiple of the chosen block size, allowing great flexibility in the trade-off between energy usage, precision and range.

In the conclusions of Chapter 7, we already mentioned that it is possible to form a hybrid between the adder tree and the accumulator, and the same can be done for the multiply-accumulator that is based on the accumulator scheme. More research on this topic might be useful, such that the compiler can choose the right variant in each situation.

As multiplication is often performed as a half-width operation, i.e. the upper half of the result is not used. As our multipliers are currently unable to perform half-width multiplications, energy is wasted on calculating this upper part. As this happens  $n + m - 1$  times for an  $(nw \times mw)$ -wide half-width multiplication using  $w$ -bit wide functional units, the performance gain can be substantial.

If the interconnect is equipped with the ability to output results in carry-save format, then more research on this topic is needed. Adders can be made in carry-save format, but also the multipliers can send out the partial products in this format, removing the final adder step on the partial products.

For the composition of adders, we only investigated two carry-propagation techniques: ripple-carry and carry-select. Other techniques might give better results, such as the carry-lookahead algorithm.

The current functional units are not pipelined. For small functional units (e.g. 8-bit wide), this does not seem necessary. However, for implementations with somewhat wider functional units (such as 16-bit wide), that should operate at high clock frequencies, pipelining might help to reduce the delay of the multiplier circuit.

We did not use all tools available to optimise our circuits to the lowest possible energy usage, as our goal was to compare the different designs, and not to find the absolute lowest energy number. If these designs are to be produced on an actual chip, several other optimisation steps should be considered, such as voltage scaling if low clock speeds are acceptable, and power or clock gating to disable unused functional units and unused parts of a functional unit (such as the multiplier when the functional unit is configured for addition). We also did not perform the place and route step; as our HPM multiplier is known for its regular layout, the results might be better than the synthesis tools currently have estimated.

# List of Figures

2.1	Flexibility versus energy efficiency trade-off for state-of-the-art architectures. . . . .	11
2.2	A mapping of an SIMD and VLIW instruction on a CGRA architecture. . . . .	14
2.3	These graphs show the effectively used bit-width for the operands of multiplications in several applications, as determined by benchmarking on a 32-bit OpenRISC. . . . .	16
2.4	An abstract overview of the BLOCKS architecture. . . . .	17
2.5	SIMD, VLIW and a tree mapped to the BLOCKS functional unit grid. . . . .	18
3.1	Partial products of two 4-bit multiplications in an 8-bit twin-multiplier. . . . .	21
3.2	Two 8-bit computational blocks performing an AND operation on two 16-bit inputs. . . . .	23
3.3	Two 8-bit computational blocks performing a left-bit-shift of 4 bits on a 16-bit input. . . . .	24
3.4	Two 8-bit computational blocks performing addition on two 16-bit inputs. The dotted line connects the carry-out of the right block to the carry-in of the left block. . . . .	25
3.5	Ten 8-bit computational blocks together performing full-width (e.g. 32-bit output) multiplication on two 16-bit inputs. The first four blocks perform multiplication operations, and the other six blocks, configured as two 24-bit adders, add the results from the multipliers. . . . .	25
4.1	Model to test the functional units with a testbench. . . . .	28
4.2	A simplified model of a functional unit as implemented in Verilog.	29

4.3	The toolflow used to benchmark our designs.	30
5.1	Implementation of a full adder based on two half adders.	35
5.2	Schematic design of a ripple-carry adder.	35
5.3	Reduced full adder implementation, which outputs <i>generate</i> $g_i$ and <i>propagate</i> $p_i$ bits.	36
5.4	A schematic representation of a Carry-Lookahead Generator or CLG with four full adders.	36
5.5	A schematic representation of a 16-bit Carry-Lookahead Adder (CLA).	37
5.6	Schematic design of a quad 8-bit carry-select adder, using seven adder blocks.	39
5.7	Schematic design of a quad functional unit adder connected with an RCA.	40
6.1	A simplified model of a functional unit implementing an adder-accumulator.	59
6.2	First step of a carry-save addition, summing three $n$ -bit numbers to a sequence of $n$ 2-bit numbers.	60
6.3	Interpretation of the result of a carry-save addition as two $n$ -bit numbers.	60
6.4	Carry-in and carry-out in carry-save additions.	60
6.5	Summary of a carry-save accumulation procedure.	61
6.6	Simplified representation of a carry-save accumulator.	62
6.7	Construction of a $4w$ -bit ripple-carry accumulator.	62
6.8	Schematic overview of a carry-accumulate functional unit.	63
6.9	A $(3 \times w)$ -bit carry-accumulation accumulator takes 3 cycles to propagate accumulated carries and produce the final result.	64
7.1	Conceptual breakdown of a multiplication. Each combination of input bits $x_i$ and $y_j$ yields a partial product bit $p_{i,j} = \text{AND}(x_i, y_j)$ , and these partial products are added together into the final product.	84
7.2	Illustration of the partial products for an 8-bit two's complement multiplication.	85

7.3	A $2w$ -bit multiplication is decomposed into four $w$ -bit multiplications, as described in Equation 7.2. . . . .	87
7.4	Example of an 8-bit full-width unsigned multiplication of $90 \times 60$ decomposed into 4-bit multiplications. . . . .	87
7.5	Illustration of the multi-granular decomposition for an 8-bit two's complement multiplication into four 4-bit multiplications. . . . .	88
7.6	A $2w$ -bit half-width multiplication is decomposed into three $w$ -bit multiplications. The fourth $w$ -bit multiplication cannot contribute to the half-width output. . . . .	89
7.7	An adder tree takes $\lceil \log_2 5 \rceil = 3$ cycles to add 5 partial product words. . . . .	91
7.8	An adder tree uses only two adders for two cycles per multiplication to add 5 partial products generated by dual-cycle multipliers. . . . .	92
7.9	An example adder tree for $2w \times 2w$ wide full-width single-cycle multiplication. . . . .	93
7.10	An example multiplier with accumulator for $2w \times 2w$ wide full-width multiplication. . . . .	94
7.11	The time schedule used to add the partial products of a $2w \times 2w$ wide full-width multiplication, generated by two single-cycle multipliers. The red partial products are generated by multiplier one, and the green partial product is generated by multiplier two. . . . .	95
8.1	Schematic overview of the modules in a distributed multiply-accumulator functional unit. . . . .	121
8.2	A possible schedule for accumulating the final result for a distributed multiply-accumulator. The numbers at the arrow are cycle numbers, and register $x_i$ contains part $i$ of the final result. . . . .	122

# List of Plots

3.1	Energy used to perform multiplications of increasing width using a 32-bit multiplier. . . . .	20
4.1	Unpredictability in benchmark results due to synthesis heuristics. . . . .	31
4.2	Example plot measuring energy efficiency and area usage of four different 32-bit adder implementations. . . . .	32
4.3	Example plot measuring energy efficiency and area usage of four different 8-bit adder implementations. The relative energy graph is truncated at the highest frequency achieved by the baseline implementation. . . . .	33
5.1	32-bit addition on a ripple-carry composition with ripple-carry base adder (RRA). . . . .	43
5.2	8-bit addition on a single ripple-carry base adder (RA). . . . .	44
5.3	32-bit addition on a ripple-carry composition with carry-lookahead base adder (RLA). . . . .	46
5.4	8-bit addition on a single carry-lookahead base adder (LA). . . . .	47
5.5	32-bit addition on a carry-select composition with ripple-carry base adder (SRA). . . . .	49
5.6	32-bit addition on a carry-select composition with carry-lookahead base adder (SLA). . . . .	51
5.7	32-bit addition using 8-bit functional units. . . . .	54
5.8	32-bit addition using 16-bit functional units. . . . .	55
5.9	8-bit addition using 8-bit and 16-bit functional units. . . . .	56

6.1	32-bit accumulation on a ripple-carry composition with a carry-lookahead accumulator (RLAC). . . . .	66
6.2	8-bit accumulation on a ripple-carry composition with a carry-lookahead accumulator (RLAC). . . . .	67
6.3	32-bit accumulation on a ripple-carry composition with a carry-save accumulator (RSAC). . . . .	69
6.4	8-bit accumulation on a ripple-carry composition with carry-save accumulator (RSAC). . . . .	70
6.5	32-bit accumulation on a carry-accumulate composition with a carry-lookahead accumulator (ALAC). . . . .	72
6.6	8-bit accumulation on a carry-accumulate composition with a carry-lookahead accumulator (ALAC). . . . .	73
6.7	32-bit accumulation on a carry-accumulate composition with a carry-save accumulator (ASAC). . . . .	75
6.8	8-bit accumulation on a carry-accumulate composition with carry-save accumulator (ASAC). . . . .	76
6.9	32-bit accumulation using 8-bit functional units. . . . .	78
6.10	32-bit accumulation using 16-bit functional units. . . . .	79
6.11	8-bit accumulation using 8-bit functional units. . . . .	80
6.12	8 bit accumulation using 16-bit functional units. . . . .	81
7.1	Single-cycle 32-bit full-width multiplication with adder tree. . . . .	97
7.2	Single-cycle 32-bit half-width multiplication with adder tree. . . . .	98
7.3	Dual-cycle 32-bit full-width multiplication with adder tree. . . . .	100
7.4	Dual-cycle 32-bit half-width multiplication with adder tree. . . . .	101
7.5	Single-cycle 32-bit full-width multiplication with accumulator. . . . .	103
7.6	Single-cycle 32-bit half-width multiplication with accumulator. . . . .	104
7.7	Dual-cycle 32-bit full-width multiplication with accumulator. . . . .	106
7.8	Dual-cycle 32-bit half-width multiplication with accumulator. . . . .	107
7.9	Single-FU single-cycle 8-bit full-width multiplication. . . . .	109
7.10	Single-FU dual-cycle 8-bit full-width multiplication. . . . .	110

7.11	32-bit full-width multiplication using 8-bit functional units.	112
7.12	32-bit full-width multiplication using 16-bit functional units.	113
7.13	32-bit half-width multiplication using 8-bit functional units.	114
7.14	32-bit half-width multiplication using 16-bit functional units.	115
7.15	8-bit full-width multiplication using 8-bit and 16-bit functional units.	116
8.1	Single-cycle 32-bit full-width accumulator-based MAC.	124
8.2	Single-cycle 32-bit half-width accumulator-based MAC.	125
8.3	Single-cycle 8-bit full-width accumulator-based MAC.	126
8.4	Single-cycle 8-bit half-width accumulator-based MAC.	127
8.5	Dual-cycle 32-bit full-width accumulator-based MAC.	129
8.6	Dual-cycle 32-bit half-width accumulator-based MAC.	130
8.7	Dual-cycle 8-bit full-width accumulator-based MAC.	131
8.8	Dual-cycle 8-bit half-width accumulator-based MAC.	132
8.9	32-bit full-width distributed multiply-accumulator.	134
8.10	32-bit half-width distributed multiply-accumulator.	135
8.11	8-bit full-width distributed multiply-accumulator.	136
8.12	8-bit half-width distributed multiply-accumulator.	137
8.13	32-bit full-width MAC using 8-bit and 16-bit functional units.	139
8.14	32-bit half-width MAC using 8-bit and 16-bit functional units.	140
8.15	8-bit full-width MAC using 8-bit and 16-bit functional units.	141
8.16	8-bit half-width MAC using 8-bit and 16-bit functional units.	142

# Bibliography

- [1] Charles R Baugh and Bruce A Wooley. A two's complement parallel array multiplication algorithm. *IEEE Transactions on Computers*, 100(12):1045–1047, 1973.
- [2] Manish Bhardwaj, Rex Min, and Anantha P Chandrakasa. Quantifying and enhancing power awareness of VLSI systems. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 9(6):757–772, 2001.
- [3] Andrew D. Booth. A Signed Binary Multiplication Technique. *The Quarterly Journal of Mechanics and Applied Mathematics*, IV(2), 1951.
- [4] Cadence Design Systems. Cadence RTL/RC Compiler. [http://cadence.com/products/ld/rtl\\_compiler/pages/default.aspx](http://cadence.com/products/ld/rtl_compiler/pages/default.aspx). Accessed: 2016-01-15.
- [5] Timothy J. Callahan and John Wawrzynek. Instruction-Level Parallelism for Reconfigurable Computing. In *Field-Programmable Logic and Applications From FPGAs to Computing Paradigm*. Springer, 1998.
- [6] James W Cooley and John W Tukey. An Algorithm for the Machine Calculation of Complex Fourier Series. *Mathematics of computation*, 19(90):297–301, 1965.
- [7] Luigi Dadda. Some Schemes For Parallel Multipliers. *Alta frequenza*, 34(5):349–356, 1965.
- [8] Lanping Deng, Kanwaldeep Solti, and Chairali Chakrabarti. Accurate models for estimating area and power of FPGA implementations. In *Acoustics, Speech and Signal Processing, 2008. ICASSP 2008. IEEE International Conference on*, 2008.
- [9] H. Eriksson, P. Larsson-Edefors, M. Sheeran, M. Själander, D. Johansson, and M. Schölin. Multiplier Reduction Tree with Logarithmic Logic Depth and Regular Connectivity. In *Proceedings of IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 4–8, May 2006.
- [10] Jose Fridman. Sub-word parallelism in digital signal processing. *Signal Processing Magazine, IEEE*, 17(2):27–35, 2000.
- [11] Seth Copen Goldstein, Herman Schmit, Matthew Moe, Mihai Budiu, Srihari Cadambi, R. Reed Taylor, and Ronald Laufer. PipeRench: A

- Co/Processor for Streaming Multimedia Acceleration. *SIGARCH Comput. Archit. News*, 27(2):28–39, May 1999.
- [12] Ronald Graham, Donald Knuth, and Oren Patashnik. *Concrete Mathematics: A Foundation for Computer Science*. Addison-Wesley, 1994.
  - [13] Reiner Hartenstein. A decade of reconfigurable computing: a visionary retrospective. In *Proceedings of the conference on Design, automation and test in Europe*, pages 642–649. IEEE Press, 2001.
  - [14] Yoonjin Kim and Rabi N. Mahapatra. *Trends in CGRA*, chapter 2, pages 7–44. CRC Press, 2011.
  - [15] Israel Koren. *Computer arithmetic algorithms*. Universities Press, second edition, 2002.
  - [16] Fei Li, Deming Chen, Lei He, and Jason Cong. Architecture evaluation for power-efficient FPGAs. In *Proceedings of the 2003 ACM/SIGDA eleventh international symposium on Field programmable gate arrays*, 2003.
  - [17] E. Mirsky and A. DeHon. MATRIX: a reconfigurable computing architecture with configurable instruction distribution and deployable resources. In *FPGAs for Custom Computing Machines, 1996. Proceedings. IEEE Symposium on*, pages 157–166, Apr 1996.
  - [18] K. Natarajan, H. Hanson, S. W. Keckler, C. R. Moore, and D. Burger. Microprocessor pipeline energy analysis. *Low Power Electronics and Design*, 2003.
  - [19] Partha Pratim Pande, Cristian Grecu, Michael Jones, Andre Ivanov, and Resve Saleh. Performance evaluation and design trade-offs for network-on-chip interconnect architectures. *Computers, IEEE Transactions on*, 54(8):1025–1040, 2005.
  - [20] Behrooz Parhami. *Computer arithmetic: algorithms and hardware designs*. Oxford University Press, second edition, 2010.
  - [21] M. Själander, H. Eriksson, and P. Larsson-Edefors. An efficient twin-precision multiplier. In *Computer Design: VLSI in Computers and Processors, 2004. ICCD 2004. Proceedings. IEEE International Conference on*, pages 30–33, Oct 2004.
  - [22] M. Själander and P. Larsson-Edefors. High-Speed and Low-Power Multipliers Using the Baugh-Wooley Algorithm and HPM Reduction Tree. In *Proceedings of IEEE International Conference on Electronics, Circuits and Systems (ICECS)*, pages 33–36, September 2008.
  - [23] James E Stine. *Digital computer arithmetic datapath design using verilog HDL*. Springer Science & Business Media, 2012.
  - [24] S. Suresh, S.F. Beldianu, and S.G. Ziavras. FPGA and ASIC square root designs for high performance and power efficiency. In *Application-Specific Systems, Architectures and Processors (ASAP), 2013 IEEE 24th International Conference on*, pages 269–272, June 2013.

- [25] Michael B. Taylor. Is Dark Silicon Useful?: Harnessing the Four Horsemen of the Coming Dark Silicon Apocalypse. In *Proceedings of the 49th Annual Design Automation Conference*, DAC '12, pages 1131–1136, New York, NY, USA, 2012. ACM.
- [26] TSMC. TSMC's 40nm CMOS technology. <http://www.tsmc.com/english/dedicatedFoundry/technology/40nm.htm>. Accessed: 2016-01-15.
- [27] E. Waingold et al. Baring it all to software: Raw machines. *Computer*, 30(9):86–93, Sep 1997.
- [28] C.S. Wallace. A Suggestion for a Fast Multiplier. *Electronic Computers, IEEE Transactions on*, EC-13(1):14–17, Feb 1964.
- [29] Mark Wijtvliet and Luc Waeijen. BLOCKS. <http://cgra.nl>. Accessed: 2016-01-15.
- [30] Ning Zhang and Bob Brodersen. The cost of flexibility in systems on a chip design for signal processing applications. *University of California, Berkeley, Tech. Rep*, 2002.

## Appendix A

# Benchmark Results

This appendix shows a listing of raw benchmark data, which can be useful to redo calculations or estimate new energy numbers when an interconnect network is known, as discussed in Chapter 9.

There are two types of tables in this appendix: the first lists the pipeline depth and delay, interconnect usage and the number of functional and delay units used in each configuration; the second lists the energy per operation and area usage for each configuration and frequency.

The first two columns,  $w$  and  $n$ , define the configuration. The column  $w$  contains the width of a single functional unit, while the column  $n$  gives the width of the operation, in a multiple of  $w$ ; i.e. the operation width  $v$  equals  $v = w \times n$ . Note that the result is twice as wide for full-width multiplications.

The pipeline depth is the time in cycles between the start of the calculations, and the final result. The pipeline delay is the time between the start of the first operation and the start of the second operation; a configuration with a pipeline delay of 1 is said to be “fully pipelined”.

The interconnect usage gives the number of bits send through the interconnect to complete the calculation. This includes supplying the inputs to the first functional units, retrieving the results from the last functional units, and carry connections (they are counted as 1 bit).

The number of functional units units gives an idea of the size and scaling properties of an algorithm. The number of delays shows the number of intermediate results that have to be stored temporarily; this can be for one or multiple cycles.

The benchmark results can also be obtained in CSV format, for automated processing. They are attached to the pdf-file and can be obtained by (right)-clicking here: [benchmark-results.csv](#), or they can be downloaded from the page <http://files.steflouwers.nl/multi-granular-results.csv>.

## A.1 Addition

Table A.1: Benchmark results for Addition Baseline (Chapter 5).

$w$	$n$	pipeline depth	pipeline delay	interconnect	FUs	delays
8	1	1	1	26	1	1
16	1	1	1	50	1	1
32	1	1	1	98	1	1

Table A.2: Benchmark results for Addition Baseline (Chapter 5).

$w$	$n$	freq.	energy/op	area	freq.	energy/op	area
8	1	100	0.14 pJ	129 $\mu m^2$	200	0.14 pJ	129 $\mu m^2$
		300	0.14 pJ	129 $\mu m^2$	400	0.14 pJ	129 $\mu m^2$
		500	0.14 pJ	129 $\mu m^2$	600	0.14 pJ	129 $\mu m^2$
		700	0.14 pJ	129 $\mu m^2$	800	0.14 pJ	129 $\mu m^2$
		900	0.14 pJ	129 $\mu m^2$	1000	0.14 pJ	129 $\mu m^2$
		1100	0.14 pJ	135 $\mu m^2$	1200	0.14 pJ	137 $\mu m^2$
		1300	0.15 pJ	134 $\mu m^2$	1400	0.14 pJ	148 $\mu m^2$
		1500	0.15 pJ	142 $\mu m^2$	1600	0.14 pJ	143 $\mu m^2$
		1700	0.15 pJ	155 $\mu m^2$	1800	0.16 pJ	190 $\mu m^2$
		1900	0.15 pJ	181 $\mu m^2$	2000	0.16 pJ	180 $\mu m^2$
		2100	0.16 pJ	186 $\mu m^2$	2200	0.17 pJ	204 $\mu m^2$
		2300	0.16 pJ	202 $\mu m^2$	2400	0.16 pJ	188 $\mu m^2$
		2500	0.15 pJ	207 $\mu m^2$	2600	0.15 pJ	215 $\mu m^2$
		2700	0.15 pJ	228 $\mu m^2$	2800	0.18 pJ	246 $\mu m^2$
		2900	0.16 pJ	234 $\mu m^2$	3000	0.16 pJ	257 $\mu m^2$
		3100	0.16 pJ	267 $\mu m^2$	3200	0.17 pJ	289 $\mu m^2$
		3300	0.18 pJ	333 $\mu m^2$	3400	0.17 pJ	308 $\mu m^2$
		3500	0.17 pJ	313 $\mu m^2$	3600	0.22 pJ	385 $\mu m^2$
		3700	0.22 pJ	381 $\mu m^2$	3800	0.24 pJ	414 $\mu m^2$
		3900	0.2 pJ	406 $\mu m^2$	4000	0.21 pJ	408 $\mu m^2$
		4100	0.21 pJ	401 $\mu m^2$	4200	0.25 pJ	478 $\mu m^2$
		4300	0.24 pJ	481 $\mu m^2$	4400	0.25 pJ	497 $\mu m^2$
		4500	0.28 pJ	536 $\mu m^2$	4600	0.28 pJ	572 $\mu m^2$
		4700	0.32 pJ	638 $\mu m^2$	4800	0.3 pJ	557 $\mu m^2$
		4900	0.32 pJ	599 $\mu m^2$	5000	0.32 pJ	622 $\mu m^2$
16	1	100	0.26 pJ	260 $\mu m^2$	200	0.26 pJ	260 $\mu m^2$
		300	0.26 pJ	260 $\mu m^2$	400	0.26 pJ	260 $\mu m^2$
		500	0.26 pJ	260 $\mu m^2$	600	0.26 pJ	270 $\mu m^2$
		700	0.27 pJ	273 $\mu m^2$	800	0.29 pJ	310 $\mu m^2$
		900	0.28 pJ	346 $\mu m^2$	1000	0.3 pJ	425 $\mu m^2$
		1100	0.3 pJ	429 $\mu m^2$	1200	0.3 pJ	434 $\mu m^2$

*Continued on next page*

Table A.2: Benchmark results for Addition Baseline (Chapter 5).

*Continued from previous page*

$w$	$n$	freq.	energy/op	area	freq.	energy/op	area
		1300	0.3 pJ	$441 \mu m^2$	1400	0.3 pJ	$429 \mu m^2$
		1500	0.3 pJ	$455 \mu m^2$	1600	0.3 pJ	$451 \mu m^2$
		1700	0.3 pJ	$454 \mu m^2$	1800	0.29 pJ	$450 \mu m^2$
		1900	0.3 pJ	$464 \mu m^2$	2000	0.31 pJ	$467 \mu m^2$
		2100	0.31 pJ	$481 \mu m^2$	2200	0.31 pJ	$477 \mu m^2$
		2300	0.32 pJ	$480 \mu m^2$	2400	0.34 pJ	$572 \mu m^2$
		2500	0.33 pJ	$554 \mu m^2$	2600	0.34 pJ	$592 \mu m^2$
		2700	0.32 pJ	$570 \mu m^2$	2800	0.34 pJ	$654 \mu m^2$
		2900	0.36 pJ	$691 \mu m^2$	3000	0.38 pJ	$772 \mu m^2$
		3100	0.4 pJ	$859 \mu m^2$	3200	0.43 pJ	$963 \mu m^2$
		3300	0.45 pJ	$992 \mu m^2$	3400	0.49 pJ	$1080 \mu m^2$
		3500	0.49 pJ	$1024 \mu m^2$	3600	0.6 pJ	$1322 \mu m^2$
32	1	100	0.48 pJ	$596 \mu m^2$	200	0.48 pJ	$596 \mu m^2$
		300	0.48 pJ	$574 \mu m^2$	400	0.51 pJ	$750 \mu m^2$
		500	0.55 pJ	$806 \mu m^2$	600	0.56 pJ	$863 \mu m^2$
		700	0.56 pJ	$885 \mu m^2$	800	0.55 pJ	$925 \mu m^2$
		900	0.57 pJ	$968 \mu m^2$	1000	0.58 pJ	$984 \mu m^2$
		1100	0.57 pJ	$1019 \mu m^2$	1200	0.57 pJ	$979 \mu m^2$
		1300	0.57 pJ	$994 \mu m^2$	1400	0.58 pJ	$1014 \mu m^2$
		1500	0.59 pJ	$1050 \mu m^2$	1600	0.6 pJ	$1039 \mu m^2$
		1700	0.59 pJ	$1086 \mu m^2$	1800	0.59 pJ	$1093 \mu m^2$
		1900	0.6 pJ	$1091 \mu m^2$	2000	0.61 pJ	$1274 \mu m^2$
		2100	0.63 pJ	$1310 \mu m^2$	2200	0.63 pJ	$1363 \mu m^2$
		2300	0.67 pJ	$1477 \mu m^2$	2400	0.69 pJ	$1551 \mu m^2$
		2500	0.73 pJ	$1672 \mu m^2$	2600	0.77 pJ	$1838 \mu m^2$
		2700	0.9 pJ	$2043 \mu m^2$	2800	0.94 pJ	$2226 \mu m^2$

### A.1.1 Ripple-Carry Composition, Ripple-Carry Base

Table A.3: Benchmark results for Ripple-Carry Composition, Ripple-Carry Base Adder (Section 5.3.1).

$w$	$n$	pipeline depth	pipeline delay	interconnect	FUs	delays
8	1	1	1	26	1	1
8	2	1	1	51	2	1
8	4	1	1	101	4	1
16	1	1	1	50	1	1
16	2	1	1	99	2	1
32	1	1	1	98	1	1

Table A.4: Benchmark results for Ripple-Carry Composition, Ripple-Carry Base Adder (Section 5.3.1).

$w$	$n$	freq.	energy/op	area	freq.	energy/op	area
8	1	100	0.14 pJ	$136 \mu m^2$	200	0.14 pJ	$136 \mu m^2$
		300	0.14 pJ	$136 \mu m^2$	400	0.14 pJ	$136 \mu m^2$
		500	0.14 pJ	$136 \mu m^2$	600	0.14 pJ	$136 \mu m^2$
		700	0.14 pJ	$136 \mu m^2$	800	0.14 pJ	$128 \mu m^2$
		900	0.14 pJ	$128 \mu m^2$	1000	0.14 pJ	$140 \mu m^2$
		1100	0.14 pJ	$151 \mu m^2$	1200	0.14 pJ	$132 \mu m^2$
		1300	0.14 pJ	$142 \mu m^2$	1400	0.14 pJ	$151 \mu m^2$
		1500	0.14 pJ	$153 \mu m^2$	1600	0.15 pJ	$160 \mu m^2$
		1700	0.16 pJ	$172 \mu m^2$	1800	0.15 pJ	$177 \mu m^2$
		1900	0.15 pJ	$193 \mu m^2$	2000	0.16 pJ	$169 \mu m^2$
		2100	0.15 pJ	$177 \mu m^2$	2200	0.18 pJ	$249 \mu m^2$
		2300	0.16 pJ	$208 \mu m^2$	2400	0.18 pJ	$257 \mu m^2$
		2500	0.18 pJ	$260 \mu m^2$	2600	0.18 pJ	$243 \mu m^2$
		2700	0.18 pJ	$266 \mu m^2$	2800	0.16 pJ	$250 \mu m^2$
		2900	0.17 pJ	$270 \mu m^2$	3000	0.18 pJ	$276 \mu m^2$
		3100	0.19 pJ	$305 \mu m^2$	3200	0.21 pJ	$325 \mu m^2$
		3300	0.2 pJ	$323 \mu m^2$	3400	0.2 pJ	$349 \mu m^2$
		3500	0.21 pJ	$381 \mu m^2$	3600	0.22 pJ	$406 \mu m^2$
		3700	0.25 pJ	$454 \mu m^2$	3800	0.24 pJ	$461 \mu m^2$
		3900	0.26 pJ	$503 \mu m^2$	4000	0.27 pJ	$524 \mu m^2$
		4100	0.3 pJ	$555 \mu m^2$	4200	0.29 pJ	$552 \mu m^2$
		4300	0.32 pJ	$605 \mu m^2$	4400	0.34 pJ	$611 \mu m^2$
		4500	0.34 pJ	$621 \mu m^2$	4600	0.35 pJ	$648 \mu m^2$
		4700	0.37 pJ	$678 \mu m^2$			
8	2	100	0.25 pJ	$266 \mu m^2$	200	0.25 pJ	$266 \mu m^2$
		300	0.25 pJ	$283 \mu m^2$	400	0.25 pJ	$288 \mu m^2$
		500	0.26 pJ	$300 \mu m^2$	600	0.27 pJ	$297 \mu m^2$
		700	0.27 pJ	$331 \mu m^2$	800	0.27 pJ	$345 \mu m^2$
		900	0.29 pJ	$362 \mu m^2$	1000	0.29 pJ	$372 \mu m^2$
		1100	0.3 pJ	$391 \mu m^2$	1200	0.32 pJ	$436 \mu m^2$
		1300	0.32 pJ	$435 \mu m^2$	1400	0.31 pJ	$432 \mu m^2$
		1500	0.32 pJ	$454 \mu m^2$	1600	0.31 pJ	$469 \mu m^2$
		1700	0.32 pJ	$510 \mu m^2$	1800	0.34 pJ	$556 \mu m^2$
		1900	0.34 pJ	$508 \mu m^2$	2000	0.33 pJ	$529 \mu m^2$
		2100	0.36 pJ	$545 \mu m^2$	2200	0.33 pJ	$537 \mu m^2$
		2300	0.35 pJ	$590 \mu m^2$	2400	0.38 pJ	$667 \mu m^2$
		2500	0.39 pJ	$689 \mu m^2$	2600	0.41 pJ	$788 \mu m^2$
		2700	0.49 pJ	$944 \mu m^2$	2800	0.48 pJ	$949 \mu m^2$
		2900	0.53 pJ	$1080 \mu m^2$	3000	0.51 pJ	$1015 \mu m^2$
		3100	0.6 pJ	$1191 \mu m^2$	3200	0.65 pJ	$1207 \mu m^2$
		3300	0.67 pJ	$1250 \mu m^2$			
8	4	100	0.48 pJ	$637 \mu m^2$	200	0.49 pJ	$645 \mu m^2$
		300	0.5 pJ	$666 \mu m^2$	400	0.54 pJ	$795 \mu m^2$

*Continued on next page*

Table A.4: Benchmark results for Ripple-Carry Composition, Ripple-Carry Base Adder (Section 5.3.1).

*Continued from previous page*

<i>w</i>	<i>n</i>	freq.	energy/op	area	freq.	energy/op	area
		500	0.58 pJ	$870 \mu m^2$	600	0.54 pJ	$857 \mu m^2$
		700	0.55 pJ	$896 \mu m^2$	800	0.61 pJ	$961 \mu m^2$
		900	0.56 pJ	$932 \mu m^2$	1000	0.58 pJ	$949 \mu m^2$
		1100	0.58 pJ	$1010 \mu m^2$	1200	0.6 pJ	$1106 \mu m^2$
		1300	0.61 pJ	$1139 \mu m^2$	1400	0.64 pJ	$1169 \mu m^2$
		1500	0.68 pJ	$1258 \mu m^2$	1600	0.65 pJ	$1229 \mu m^2$
		1700	0.69 pJ	$1413 \mu m^2$	1800	0.72 pJ	$1507 \mu m^2$
		1900	0.74 pJ	$1597 \mu m^2$	2000	0.79 pJ	$1580 \mu m^2$
		2100	0.85 pJ	$1726 \mu m^2$	2200	0.87 pJ	$1928 \mu m^2$
		2300	0.92 pJ	$1987 \mu m^2$	2400	0.99 pJ	$2150 \mu m^2$
		2500	1.17 pJ	$2625 \mu m^2$	2600	1.29 pJ	$2824 \mu m^2$
16	1	100	0.26 pJ	$258 \mu m^2$	200	0.26 pJ	$258 \mu m^2$
		300	0.26 pJ	$258 \mu m^2$	400	0.26 pJ	$260 \mu m^2$
		500	0.26 pJ	$290 \mu m^2$	600	0.27 pJ	$290 \mu m^2$
		700	0.28 pJ	$307 \mu m^2$	800	0.27 pJ	$319 \mu m^2$
		900	0.28 pJ	$347 \mu m^2$	1000	0.29 pJ	$364 \mu m^2$
		1100	0.28 pJ	$390 \mu m^2$	1200	0.3 pJ	$389 \mu m^2$
		1300	0.3 pJ	$374 \mu m^2$	1400	0.3 pJ	$397 \mu m^2$
		1500	0.29 pJ	$425 \mu m^2$	1600	0.32 pJ	$454 \mu m^2$
		1700	0.32 pJ	$445 \mu m^2$	1800	0.32 pJ	$484 \mu m^2$
		1900	0.34 pJ	$483 \mu m^2$	2000	0.35 pJ	$533 \mu m^2$
		2100	0.34 pJ	$532 \mu m^2$	2200	0.36 pJ	$619 \mu m^2$
		2300	0.38 pJ	$625 \mu m^2$	2400	0.37 pJ	$635 \mu m^2$
		2500	0.37 pJ	$684 \mu m^2$	2600	0.39 pJ	$701 \mu m^2$
		2700	0.42 pJ	$784 \mu m^2$	2800	0.45 pJ	$888 \mu m^2$
		2900	0.5 pJ	$986 \mu m^2$	3000	0.5 pJ	$971 \mu m^2$
		3100	0.56 pJ	$1057 \mu m^2$	3200	0.57 pJ	$1135 \mu m^2$
16	2	100	0.48 pJ	$595 \mu m^2$	200	0.48 pJ	$632 \mu m^2$
		300	0.49 pJ	$673 \mu m^2$	400	0.52 pJ	$716 \mu m^2$
		500	0.57 pJ	$800 \mu m^2$	600	0.57 pJ	$847 \mu m^2$
		700	0.59 pJ	$923 \mu m^2$	800	0.59 pJ	$944 \mu m^2$
		900	0.58 pJ	$970 \mu m^2$	1000	0.63 pJ	$1082 \mu m^2$
		1100	0.63 pJ	$1144 \mu m^2$	1200	0.61 pJ	$999 \mu m^2$
		1300	0.63 pJ	$1069 \mu m^2$	1400	0.64 pJ	$1164 \mu m^2$
		1500	0.65 pJ	$1239 \mu m^2$	1600	0.68 pJ	$1278 \mu m^2$
		1700	0.72 pJ	$1509 \mu m^2$	1800	0.68 pJ	$1383 \mu m^2$
		1900	0.75 pJ	$1509 \mu m^2$	2000	0.8 pJ	$1639 \mu m^2$
		2100	0.88 pJ	$1853 \mu m^2$	2200	0.95 pJ	$1997 \mu m^2$
		2300	1.07 pJ	$2148 \mu m^2$			
32	1	100	0.48 pJ	$602 \mu m^2$	200	0.48 pJ	$602 \mu m^2$
		300	0.49 pJ	$660 \mu m^2$	400	0.53 pJ	$707 \mu m^2$
		500	0.54 pJ	$795 \mu m^2$	600	0.55 pJ	$808 \mu m^2$
		700	0.56 pJ	$841 \mu m^2$	800	0.58 pJ	$863 \mu m^2$
		900	0.6 pJ	$878 \mu m^2$	1000	0.6 pJ	$906 \mu m^2$

*Continued on next page*

Table A.4: Benchmark results for Ripple-Carry Composition, Ripple-Carry Base Adder (Section 5.3.1).

*Continued from previous page*

$w$	$n$	freq.	energy/op	area	freq.	energy/op	area
		1100	0.55 pJ	$872 \mu m^2$	1200	0.57 pJ	$954 \mu m^2$
		1300	0.59 pJ	$978 \mu m^2$	1400	0.61 pJ	$1046 \mu m^2$
		1500	0.6 pJ	$1042 \mu m^2$	1600	0.62 pJ	$1182 \mu m^2$
		1700	0.63 pJ	$1185 \mu m^2$	1800	0.68 pJ	$1337 \mu m^2$
		1900	0.68 pJ	$1410 \mu m^2$	2000	0.73 pJ	$1523 \mu m^2$
		2100	0.76 pJ	$1607 \mu m^2$	2200	0.81 pJ	$1772 \mu m^2$
		2300	0.97 pJ	$2072 \mu m^2$	2400	1.12 pJ	$2378 \mu m^2$
		2500	1.19 pJ	$2317 \mu m^2$			

### A.1.2 Ripple-Carry Composition, Carry-Lookahead Base

Table A.5: Benchmark results for Ripple-Carry Composition, Carry-Lookahead Base Adder (Section 5.3.2).

$w$	$n$	pipeline depth	pipeline delay	interconnect	FUs	delays
8	1	1	1	26	1	1
8	2	1	1	51	2	1
8	4	1	1	101	4	1
16	1	1	1	50	1	1
16	2	1	1	99	2	1
32	1	1	1	98	1	1

Table A.6: Benchmark results for Ripple-Carry Composition, Carry-Lookahead Base Adder (Section 5.3.2).

$w$	$n$	freq.	energy/op	area	freq.	energy/op	area
8	1	100	0.14 pJ	$131 \mu m^2$	200	0.14 pJ	$131 \mu m^2$
		300	0.15 pJ	$138 \mu m^2$	400	0.14 pJ	$131 \mu m^2$
		500	0.14 pJ	$131 \mu m^2$	600	0.14 pJ	$131 \mu m^2$
		700	0.14 pJ	$131 \mu m^2$	800	0.14 pJ	$131 \mu m^2$
		900	0.14 pJ	$131 \mu m^2$	1000	0.14 pJ	$131 \mu m^2$
		1100	0.14 pJ	$122 \mu m^2$	1200	0.14 pJ	$145 \mu m^2$
		1300	0.15 pJ	$132 \mu m^2$	1400	0.16 pJ	$139 \mu m^2$
		1500	0.15 pJ	$137 \mu m^2$	1600	0.15 pJ	$145 \mu m^2$
		1700	0.16 pJ	$162 \mu m^2$	1800	0.16 pJ	$179 \mu m^2$
		1900	0.16 pJ	$199 \mu m^2$	2000	0.16 pJ	$199 \mu m^2$

*Continued on next page*

Table A.6: Benchmark results for Ripple-Carry Composition, Carry-Lookahead Base Adder (Section 5.3.2).

*Continued from previous page*

<i>w</i>	<i>n</i>	freq.	energy/op	area	freq.	energy/op	area
		2100	0.16 pJ	198 $\mu m^2$	2200	0.15 pJ	194 $\mu m^2$
		2300	0.16 pJ	220 $\mu m^2$	2400	0.17 pJ	243 $\mu m^2$
		2500	0.16 pJ	208 $\mu m^2$	2600	0.16 pJ	214 $\mu m^2$
		2700	0.16 pJ	231 $\mu m^2$	2800	0.17 pJ	270 $\mu m^2$
		2900	0.18 pJ	292 $\mu m^2$	3000	0.17 pJ	313 $\mu m^2$
		3100	0.17 pJ	306 $\mu m^2$	3200	0.19 pJ	386 $\mu m^2$
		3300	0.19 pJ	369 $\mu m^2$	3400	0.2 pJ	378 $\mu m^2$
		3500	0.19 pJ	366 $\mu m^2$	3600	0.2 pJ	386 $\mu m^2$
		3700	0.21 pJ	426 $\mu m^2$	3800	0.22 pJ	420 $\mu m^2$
		3900	0.26 pJ	527 $\mu m^2$	4000	0.25 pJ	484 $\mu m^2$
		4100	0.25 pJ	519 $\mu m^2$	4200	0.25 pJ	531 $\mu m^2$
		4300	0.26 pJ	575 $\mu m^2$	4400	0.28 pJ	564 $\mu m^2$
		4500	0.3 pJ	651 $\mu m^2$	4600	0.3 pJ	632 $\mu m^2$
		4700	0.33 pJ	689 $\mu m^2$	4800	0.34 pJ	688 $\mu m^2$
8	2	100	0.27 pJ	319 $\mu m^2$	200	0.28 pJ	312 $\mu m^2$
		300	0.26 pJ	262 $\mu m^2$	400	0.26 pJ	262 $\mu m^2$
		500	0.26 pJ	262 $\mu m^2$	600	0.26 pJ	274 $\mu m^2$
		700	0.28 pJ	338 $\mu m^2$	800	0.27 pJ	368 $\mu m^2$
		900	0.28 pJ	375 $\mu m^2$	1000	0.28 pJ	372 $\mu m^2$
		1100	0.29 pJ	401 $\mu m^2$	1200	0.29 pJ	416 $\mu m^2$
		1300	0.3 pJ	418 $\mu m^2$	1400	0.3 pJ	437 $\mu m^2$
		1500	0.3 pJ	453 $\mu m^2$	1600	0.31 pJ	461 $\mu m^2$
		1700	0.3 pJ	477 $\mu m^2$	1800	0.29 pJ	450 $\mu m^2$
		1900	0.3 pJ	482 $\mu m^2$	2000	0.31 pJ	492 $\mu m^2$
		2100	0.31 pJ	543 $\mu m^2$	2200	0.31 pJ	533 $\mu m^2$
		2300	0.33 pJ	630 $\mu m^2$	2400	0.35 pJ	674 $\mu m^2$
		2500	0.34 pJ	732 $\mu m^2$	2600	0.34 pJ	688 $\mu m^2$
		2700	0.37 pJ	774 $\mu m^2$	2800	0.37 pJ	775 $\mu m^2$
		2900	0.39 pJ	820 $\mu m^2$	3000	0.4 pJ	886 $\mu m^2$
		3100	0.39 pJ	886 $\mu m^2$	3200	0.44 pJ	965 $\mu m^2$
		3300	0.49 pJ	1070 $\mu m^2$	3400	0.52 pJ	1105 $\mu m^2$
		3500	0.47 pJ	1069 $\mu m^2$			
8	4	100	0.51 pJ	698 $\mu m^2$	200	0.48 pJ	612 $\mu m^2$
		300	0.49 pJ	678 $\mu m^2$	400	0.5 pJ	768 $\mu m^2$
		500	0.52 pJ	838 $\mu m^2$	600	0.52 pJ	806 $\mu m^2$
		700	0.53 pJ	847 $\mu m^2$	800	0.54 pJ	913 $\mu m^2$
		900	0.55 pJ	964 $\mu m^2$	1000	0.56 pJ	1000 $\mu m^2$
		1100	0.56 pJ	1054 $\mu m^2$	1200	0.57 pJ	1071 $\mu m^2$
		1300	0.55 pJ	1008 $\mu m^2$	1400	0.55 pJ	985 $\mu m^2$
		1500	0.59 pJ	1267 $\mu m^2$	1600	0.59 pJ	1177 $\mu m^2$
		1700	0.6 pJ	1245 $\mu m^2$	1800	0.61 pJ	1338 $\mu m^2$
		1900	0.64 pJ	1483 $\mu m^2$	2000	0.69 pJ	1691 $\mu m^2$
		2100	0.72 pJ	1824 $\mu m^2$	2200	0.73 pJ	1771 $\mu m^2$
		2300	0.8 pJ	2070 $\mu m^2$	2400	0.84 pJ	2032 $\mu m^2$

*Continued on next page*

Table A.6: Benchmark results for Ripple-Carry Composition, Carry-Lookahead Base Adder (Section 5.3.2).

*Continued from previous page*

w	n	freq. energy/op area			freq. energy/op area		
		2500	0.9 pJ	2156 $\mu m^2$	2600	0.97 pJ	2312 $\mu m^2$
16	1	100	0.29 pJ	466 $\mu m^2$	200	0.29 pJ	466 $\mu m^2$
		300	0.29 pJ	461 $\mu m^2$	400	0.29 pJ	442 $\mu m^2$
		500	0.29 pJ	442 $\mu m^2$	600	0.29 pJ	442 $\mu m^2$
		700	0.29 pJ	442 $\mu m^2$	800	0.29 pJ	438 $\mu m^2$
		900	0.29 pJ	448 $\mu m^2$	1000	0.29 pJ	437 $\mu m^2$
		1100	0.3 pJ	463 $\mu m^2$	1200	0.29 pJ	455 $\mu m^2$
		1300	0.29 pJ	463 $\mu m^2$	1400	0.3 pJ	487 $\mu m^2$
		1500	0.31 pJ	480 $\mu m^2$	1600	0.31 pJ	501 $\mu m^2$
		1700	0.28 pJ	432 $\mu m^2$	1800	0.28 pJ	461 $\mu m^2$
		1900	0.3 pJ	441 $\mu m^2$	2000	0.29 pJ	430 $\mu m^2$
		2100	0.31 pJ	507 $\mu m^2$	2200	0.31 pJ	499 $\mu m^2$
		2300	0.33 pJ	556 $\mu m^2$	2400	0.33 pJ	612 $\mu m^2$
		2500	0.33 pJ	605 $\mu m^2$	2600	0.35 pJ	666 $\mu m^2$
		2700	0.36 pJ	708 $\mu m^2$	2800	0.39 pJ	742 $\mu m^2$
		2900	0.39 pJ	799 $\mu m^2$	3000	0.41 pJ	865 $\mu m^2$
		3100	0.43 pJ	874 $\mu m^2$	3200	0.46 pJ	989 $\mu m^2$
		3300	0.44 pJ	954 $\mu m^2$	3400	0.48 pJ	1041 $\mu m^2$
		3500	0.52 pJ	1101 $\mu m^2$	3600	0.58 pJ	1231 $\mu m^2$
		3700	0.59 pJ	1246 $\mu m^2$			
16	2	100	0.56 pJ	913 $\mu m^2$	200	0.56 pJ	913 $\mu m^2$
		300	0.56 pJ	927 $\mu m^2$	400	0.56 pJ	950 $\mu m^2$
		500	0.56 pJ	950 $\mu m^2$	600	0.56 pJ	932 $\mu m^2$
		700	0.57 pJ	926 $\mu m^2$	800	0.56 pJ	1031 $\mu m^2$
		900	0.56 pJ	935 $\mu m^2$	1000	0.57 pJ	999 $\mu m^2$
		1100	0.57 pJ	999 $\mu m^2$	1200	0.54 pJ	961 $\mu m^2$
		1300	0.56 pJ	1051 $\mu m^2$	1400	0.56 pJ	978 $\mu m^2$
		1500	0.56 pJ	1005 $\mu m^2$	1600	0.57 pJ	1097 $\mu m^2$
		1700	0.58 pJ	1153 $\mu m^2$	1800	0.6 pJ	1242 $\mu m^2$
		1900	0.64 pJ	1339 $\mu m^2$	2000	0.65 pJ	1493 $\mu m^2$
		2100	0.68 pJ	1483 $\mu m^2$	2200	0.71 pJ	1587 $\mu m^2$
		2300	0.77 pJ	1782 $\mu m^2$	2400	0.83 pJ	1819 $\mu m^2$
32	1	100	0.56 pJ	915 $\mu m^2$	200	0.56 pJ	915 $\mu m^2$
		300	0.56 pJ	900 $\mu m^2$	400	0.56 pJ	927 $\mu m^2$
		500	0.55 pJ	927 $\mu m^2$	600	0.56 pJ	926 $\mu m^2$
		700	0.56 pJ	920 $\mu m^2$	800	0.56 pJ	981 $\mu m^2$
		900	0.57 pJ	1023 $\mu m^2$	1000	0.56 pJ	940 $\mu m^2$
		1100	0.56 pJ	959 $\mu m^2$	1200	0.57 pJ	972 $\mu m^2$
		1300	0.56 pJ	971 $\mu m^2$	1400	0.56 pJ	984 $\mu m^2$
		1500	0.58 pJ	1050 $\mu m^2$	1600	0.6 pJ	1140 $\mu m^2$
		1700	0.6 pJ	1145 $\mu m^2$	1800	0.64 pJ	1226 $\mu m^2$
		1900	0.65 pJ	1296 $\mu m^2$	2000	0.68 pJ	1374 $\mu m^2$
		2100	0.69 pJ	1463 $\mu m^2$	2200	0.73 pJ	1582 $\mu m^2$
		2300	0.8 pJ	1767 $\mu m^2$	2400	0.82 pJ	1771 $\mu m^2$

*Continued on next page*

Table A.6: Benchmark results for Ripple-Carry Composition, Carry-Lookahead Base Adder (Section 5.3.2).

*Continued from previous page*

$w$	$n$	freq.	energy/op	area	freq.	energy/op	area
		2500	0.92 pJ	1974 $\mu m^2$	2600	0.97 pJ	2105 $\mu m^2$

### A.1.3 Carry-Select Composition, Ripple-Carry Base

Table A.7: Benchmark results for Carry-Select Composition, Ripple-Carry Base Adder (Section 5.3.3).

$w$	$n$	pipeline depth	pipeline delay	interconnect	FUs	delays
8	1	1	1	26	1	1
8	2	1	1	51	2	1
8	4	1	1	101	4	1
16	1	1	1	50	1	1
16	2	1	1	99	2	1
32	1	1	1	98	1	1

Table A.8: Benchmark results for Carry-Select Composition, Ripple-Carry Base Adder (Section 5.3.3).

$w$	$n$	freq.	energy/op	area	freq.	energy/op	area
8	1	100	0.14 pJ	136 $\mu m^2$	200	0.14 pJ	136 $\mu m^2$
		300	0.14 pJ	136 $\mu m^2$	400	0.14 pJ	136 $\mu m^2$
		500	0.14 pJ	136 $\mu m^2$	600	0.14 pJ	136 $\mu m^2$
		700	0.14 pJ	136 $\mu m^2$	800	0.14 pJ	128 $\mu m^2$
		900	0.14 pJ	128 $\mu m^2$	1000	0.14 pJ	140 $\mu m^2$
		1100	0.14 pJ	151 $\mu m^2$	1200	0.14 pJ	132 $\mu m^2$
		1300	0.14 pJ	142 $\mu m^2$	1400	0.14 pJ	151 $\mu m^2$
		1500	0.14 pJ	153 $\mu m^2$	1600	0.15 pJ	160 $\mu m^2$
		1700	0.16 pJ	172 $\mu m^2$	1800	0.15 pJ	177 $\mu m^2$
		1900	0.15 pJ	193 $\mu m^2$	2000	0.16 pJ	169 $\mu m^2$
		2100	0.15 pJ	177 $\mu m^2$	2200	0.18 pJ	249 $\mu m^2$
		2300	0.16 pJ	208 $\mu m^2$	2400	0.18 pJ	257 $\mu m^2$
		2500	0.18 pJ	260 $\mu m^2$	2600	0.18 pJ	243 $\mu m^2$
		2700	0.18 pJ	266 $\mu m^2$	2800	0.16 pJ	250 $\mu m^2$
		2900	0.17 pJ	270 $\mu m^2$	3000	0.18 pJ	276 $\mu m^2$
		3100	0.19 pJ	305 $\mu m^2$	3200	0.21 pJ	325 $\mu m^2$
		3300	0.2 pJ	323 $\mu m^2$	3400	0.2 pJ	349 $\mu m^2$

*Continued on next page*

Table A.8: Benchmark results for Carry-Select Composition,  
Ripple-Carry Base Adder (Section 5.3.3).

*Continued from previous page*

<i>w</i>	<i>n</i>	freq.	energy/op	area	freq.	energy/op	area
		3500	0.21 pJ	$381 \mu m^2$	3600	0.22 pJ	$406 \mu m^2$
		3700	0.25 pJ	$454 \mu m^2$	3800	0.24 pJ	$461 \mu m^2$
		3900	0.26 pJ	$503 \mu m^2$	4000	0.27 pJ	$524 \mu m^2$
		4100	0.3 pJ	$555 \mu m^2$	4200	0.29 pJ	$552 \mu m^2$
		4300	0.32 pJ	$605 \mu m^2$	4400	0.34 pJ	$611 \mu m^2$
		4500	0.34 pJ	$621 \mu m^2$	4600	0.35 pJ	$648 \mu m^2$
		4700	0.37 pJ	$678 \mu m^2$			
8	2	100	0.28 pJ	$419 \mu m^2$	200	0.28 pJ	$417 \mu m^2$
		300	0.28 pJ	$433 \mu m^2$	400	0.28 pJ	$433 \mu m^2$
		500	0.28 pJ	$433 \mu m^2$	600	0.28 pJ	$433 \mu m^2$
		700	0.28 pJ	$419 \mu m^2$	800	0.29 pJ	$440 \mu m^2$
		900	0.3 pJ	$441 \mu m^2$	1000	0.3 pJ	$448 \mu m^2$
		1100	0.31 pJ	$462 \mu m^2$	1200	0.31 pJ	$450 \mu m^2$
		1300	0.32 pJ	$483 \mu m^2$	1400	0.35 pJ	$538 \mu m^2$
		1500	0.35 pJ	$536 \mu m^2$	1600	0.36 pJ	$546 \mu m^2$
		1700	0.36 pJ	$585 \mu m^2$	1800	0.33 pJ	$577 \mu m^2$
		1900	0.34 pJ	$614 \mu m^2$	2000	0.38 pJ	$594 \mu m^2$
		2100	0.37 pJ	$691 \mu m^2$	2200	0.39 pJ	$724 \mu m^2$
		2300	0.42 pJ	$777 \mu m^2$	2400	0.42 pJ	$778 \mu m^2$
		2500	0.4 pJ	$774 \mu m^2$	2600	0.47 pJ	$945 \mu m^2$
		2700	0.45 pJ	$922 \mu m^2$	2800	0.49 pJ	$1134 \mu m^2$
		2900	0.51 pJ	$1139 \mu m^2$	3000	0.51 pJ	$1134 \mu m^2$
		3100	0.53 pJ	$1187 \mu m^2$	3200	0.52 pJ	$1206 \mu m^2$
		3300	0.6 pJ	$1347 \mu m^2$	3400	0.6 pJ	$1334 \mu m^2$
		3500	0.73 pJ	$1527 \mu m^2$	3600	0.74 pJ	$1596 \mu m^2$
8	4	100	0.55 pJ	$1056 \mu m^2$	200	0.55 pJ	$1078 \mu m^2$
		300	0.55 pJ	$1002 \mu m^2$	400	0.55 pJ	$1002 \mu m^2$
		500	0.57 pJ	$1065 \mu m^2$	600	0.57 pJ	$1090 \mu m^2$
		700	0.56 pJ	$1094 \mu m^2$	800	0.58 pJ	$1142 \mu m^2$
		900	0.6 pJ	$1123 \mu m^2$	1000	0.59 pJ	$1166 \mu m^2$
		1100	0.61 pJ	$1210 \mu m^2$	1200	0.66 pJ	$1305 \mu m^2$
		1300	0.61 pJ	$1376 \mu m^2$	1400	0.66 pJ	$1434 \mu m^2$
		1500	0.67 pJ	$1518 \mu m^2$	1600	0.7 pJ	$1492 \mu m^2$
		1700	0.7 pJ	$1461 \mu m^2$	1800	0.74 pJ	$1617 \mu m^2$
		1900	0.77 pJ	$1669 \mu m^2$	2000	0.82 pJ	$1818 \mu m^2$
		2100	0.88 pJ	$1957 \mu m^2$	2200	0.92 pJ	$1983 \mu m^2$
		2300	1 pJ	$2148 \mu m^2$			
16	1	100	0.26 pJ	$258 \mu m^2$	200	0.26 pJ	$258 \mu m^2$
		300	0.26 pJ	$258 \mu m^2$	400	0.26 pJ	$260 \mu m^2$
		500	0.26 pJ	$290 \mu m^2$	600	0.27 pJ	$290 \mu m^2$
		700	0.28 pJ	$307 \mu m^2$	800	0.27 pJ	$319 \mu m^2$
		900	0.28 pJ	$347 \mu m^2$	1000	0.29 pJ	$364 \mu m^2$
		1100	0.28 pJ	$390 \mu m^2$	1200	0.3 pJ	$389 \mu m^2$
		1300	0.3 pJ	$374 \mu m^2$	1400	0.3 pJ	$397 \mu m^2$

*Continued on next page*

Table A.8: Benchmark results for Carry-Select Composition,  
Ripple-Carry Base Adder (Section 5.3.3).

*Continued from previous page*

<i>w</i>	<i>n</i>	freq.	energy/op	area	freq.	energy/op	area
16	2	1500	0.29 pJ	$425 \mu m^2$	1600	0.32 pJ	$454 \mu m^2$
		1700	0.32 pJ	$445 \mu m^2$	1800	0.32 pJ	$484 \mu m^2$
		1900	0.34 pJ	$483 \mu m^2$	2000	0.35 pJ	$533 \mu m^2$
		2100	0.34 pJ	$532 \mu m^2$	2200	0.36 pJ	$619 \mu m^2$
		2300	0.38 pJ	$625 \mu m^2$	2400	0.37 pJ	$635 \mu m^2$
		2500	0.37 pJ	$684 \mu m^2$	2600	0.39 pJ	$701 \mu m^2$
		2700	0.42 pJ	$784 \mu m^2$	2800	0.45 pJ	$888 \mu m^2$
		2900	0.5 pJ	$986 \mu m^2$	3000	0.5 pJ	$971 \mu m^2$
		3100	0.56 pJ	$1057 \mu m^2$	3200	0.57 pJ	$1135 \mu m^2$
		100	0.61 pJ	$876 \mu m^2$	200	0.61 pJ	$863 \mu m^2$
32	1	300	0.61 pJ	$863 \mu m^2$	400	0.61 pJ	$879 \mu m^2$
		500	0.63 pJ	$907 \mu m^2$	600	0.65 pJ	$934 \mu m^2$
		700	0.67 pJ	$1021 \mu m^2$	800	0.66 pJ	$1041 \mu m^2$
		900	0.7 pJ	$1158 \mu m^2$	1000	0.68 pJ	$1160 \mu m^2$
		1100	0.73 pJ	$1216 \mu m^2$	1200	0.74 pJ	$1321 \mu m^2$
		1300	0.76 pJ	$1337 \mu m^2$	1400	0.76 pJ	$1322 \mu m^2$
		1500	0.76 pJ	$1377 \mu m^2$	1600	0.83 pJ	$1499 \mu m^2$
		1700	0.83 pJ	$1578 \mu m^2$	1800	0.83 pJ	$1663 \mu m^2$
		1900	0.81 pJ	$1534 \mu m^2$	2000	0.89 pJ	$1818 \mu m^2$
		2100	0.92 pJ	$1867 \mu m^2$	2200	0.93 pJ	$1953 \mu m^2$
		2300	1 pJ	$2200 \mu m^2$	2400	1.01 pJ	$2302 \mu m^2$
		2500	1.18 pJ	$2535 \mu m^2$	2600	1.29 pJ	$2772 \mu m^2$
		2700	1.37 pJ	$2995 \mu m^2$	2800	1.37 pJ	$3106 \mu m^2$
		100	0.48 pJ	$602 \mu m^2$	200	0.48 pJ	$602 \mu m^2$
		300	0.49 pJ	$660 \mu m^2$	400	0.53 pJ	$707 \mu m^2$
		500	0.54 pJ	$795 \mu m^2$	600	0.55 pJ	$808 \mu m^2$
		700	0.56 pJ	$841 \mu m^2$	800	0.58 pJ	$863 \mu m^2$
		900	0.6 pJ	$878 \mu m^2$	1000	0.6 pJ	$906 \mu m^2$
		1100	0.55 pJ	$872 \mu m^2$	1200	0.57 pJ	$954 \mu m^2$
		1300	0.59 pJ	$978 \mu m^2$	1400	0.61 pJ	$1046 \mu m^2$
		1500	0.6 pJ	$1042 \mu m^2$	1600	0.62 pJ	$1182 \mu m^2$
		1700	0.63 pJ	$1185 \mu m^2$	1800	0.68 pJ	$1337 \mu m^2$
		1900	0.68 pJ	$1410 \mu m^2$	2000	0.73 pJ	$1523 \mu m^2$
		2100	0.76 pJ	$1607 \mu m^2$	2200	0.81 pJ	$1772 \mu m^2$
		2300	0.97 pJ	$2072 \mu m^2$	2400	1.12 pJ	$2378 \mu m^2$
		2500	1.19 pJ	$2317 \mu m^2$			

#### A.1.4 Carry-Select Composition, Carry-Lookahead Base

Table A.9: Benchmark results for Carry-Select Composition, Carry-Lookahead Base Adder (Section 5.3.4).

$w$	$n$	pipeline depth	pipeline delay	interconnect	FUs	delays
8	1	1	1	26	1	1
8	2	1	1	51	2	1
8	4	1	1	101	4	1
16	1	1	1	50	1	1
16	2	1	1	99	2	1
32	1	1	1	98	1	1

Table A.10: Benchmark results for Carry-Select Composition, Carry-Lookahead Base Adder (Section 5.3.4).

$w$	$n$	freq.	energy/op	area	freq.	energy/op	area
8	1	100	0.14 pJ	131 $\mu m^2$	200	0.14 pJ	131 $\mu m^2$
		300	0.15 pJ	138 $\mu m^2$	400	0.14 pJ	131 $\mu m^2$
		500	0.14 pJ	131 $\mu m^2$	600	0.14 pJ	131 $\mu m^2$
		700	0.14 pJ	131 $\mu m^2$	800	0.14 pJ	131 $\mu m^2$
		900	0.14 pJ	131 $\mu m^2$	1000	0.14 pJ	131 $\mu m^2$
		1100	0.14 pJ	122 $\mu m^2$	1200	0.14 pJ	145 $\mu m^2$
		1300	0.15 pJ	132 $\mu m^2$	1400	0.16 pJ	139 $\mu m^2$
		1500	0.15 pJ	137 $\mu m^2$	1600	0.15 pJ	145 $\mu m^2$
		1700	0.16 pJ	162 $\mu m^2$	1800	0.16 pJ	179 $\mu m^2$
		1900	0.16 pJ	199 $\mu m^2$	2000	0.16 pJ	199 $\mu m^2$
		2100	0.16 pJ	198 $\mu m^2$	2200	0.15 pJ	194 $\mu m^2$
		2300	0.16 pJ	220 $\mu m^2$	2400	0.17 pJ	243 $\mu m^2$
		2500	0.16 pJ	208 $\mu m^2$	2600	0.16 pJ	214 $\mu m^2$
		2700	0.16 pJ	231 $\mu m^2$	2800	0.17 pJ	270 $\mu m^2$
		2900	0.18 pJ	292 $\mu m^2$	3000	0.17 pJ	313 $\mu m^2$
		3100	0.17 pJ	306 $\mu m^2$	3200	0.19 pJ	386 $\mu m^2$
		3300	0.19 pJ	369 $\mu m^2$	3400	0.2 pJ	378 $\mu m^2$
		3500	0.19 pJ	366 $\mu m^2$	3600	0.2 pJ	386 $\mu m^2$
		3700	0.21 pJ	426 $\mu m^2$	3800	0.22 pJ	420 $\mu m^2$
		3900	0.26 pJ	527 $\mu m^2$	4000	0.25 pJ	484 $\mu m^2$
		4100	0.25 pJ	519 $\mu m^2$	4200	0.25 pJ	531 $\mu m^2$
		4300	0.26 pJ	575 $\mu m^2$	4400	0.28 pJ	564 $\mu m^2$
		4500	0.3 pJ	651 $\mu m^2$	4600	0.3 pJ	632 $\mu m^2$
		4700	0.33 pJ	689 $\mu m^2$	4800	0.34 pJ	688 $\mu m^2$
8	2	100	0.29 pJ	423 $\mu m^2$	200	0.32 pJ	500 $\mu m^2$
		300	0.31 pJ	458 $\mu m^2$	400	0.31 pJ	484 $\mu m^2$
		500	0.31 pJ	484 $\mu m^2$	600	0.31 pJ	482 $\mu m^2$
		700	0.31 pJ	474 $\mu m^2$	800	0.31 pJ	475 $\mu m^2$

Continued on next page

Table A.10: Benchmark results for Carry-Select Composition, Carry-Lookahead Base Adder (Section 5.3.4).

*Continued from previous page*

<i>w</i>	<i>n</i>	freq.	energy/op	area	freq.	energy/op	area
		900	0.3 pJ	$477 \mu m^2$	1000	0.3 pJ	$488 \mu m^2$
		1100	0.3 pJ	$476 \mu m^2$	1200	0.31 pJ	$500 \mu m^2$
		1300	0.32 pJ	$535 \mu m^2$	1400	0.32 pJ	$563 \mu m^2$
		1500	0.31 pJ	$520 \mu m^2$	1600	0.31 pJ	$538 \mu m^2$
		1700	0.33 pJ	$611 \mu m^2$	1800	0.31 pJ	$577 \mu m^2$
		1900	0.33 pJ	$571 \mu m^2$	2000	0.32 pJ	$595 \mu m^2$
		2100	0.33 pJ	$628 \mu m^2$	2200	0.33 pJ	$681 \mu m^2$
		2300	0.33 pJ	$642 \mu m^2$	2400	0.36 pJ	$750 \mu m^2$
		2500	0.38 pJ	$688 \mu m^2$	2600	0.38 pJ	$690 \mu m^2$
		2700	0.4 pJ	$756 \mu m^2$	2800	0.42 pJ	$785 \mu m^2$
		2900	0.42 pJ	$860 \mu m^2$	3000	0.44 pJ	$991 \mu m^2$
		3100	0.46 pJ	$959 \mu m^2$	3200	0.48 pJ	$1051 \mu m^2$
		3300	0.5 pJ	$1200 \mu m^2$	3400	0.53 pJ	$1287 \mu m^2$
		3500	0.57 pJ	$1303 \mu m^2$	3600	0.66 pJ	$1403 \mu m^2$
		3700	0.64 pJ	$1477 \mu m^2$	3800	0.69 pJ	$1606 \mu m^2$
8	4	100	0.57 pJ	$1121 \mu m^2$	200	0.57 pJ	$1134 \mu m^2$
		300	0.56 pJ	$1102 \mu m^2$	400	0.56 pJ	$1121 \mu m^2$
		500	0.56 pJ	$1091 \mu m^2$	600	0.56 pJ	$1099 \mu m^2$
		700	0.57 pJ	$1150 \mu m^2$	800	0.58 pJ	$1189 \mu m^2$
		900	0.57 pJ	$1166 \mu m^2$	1000	0.6 pJ	$1240 \mu m^2$
		1100	0.61 pJ	$1217 \mu m^2$	1200	0.6 pJ	$1291 \mu m^2$
		1300	0.61 pJ	$1203 \mu m^2$	1400	0.63 pJ	$1214 \mu m^2$
		1500	0.65 pJ	$1264 \mu m^2$	1600	0.65 pJ	$1252 \mu m^2$
		1700	0.69 pJ	$1259 \mu m^2$	1800	0.67 pJ	$1252 \mu m^2$
		1900	0.68 pJ	$1360 \mu m^2$	2000	0.68 pJ	$1386 \mu m^2$
		2100	0.7 pJ	$1440 \mu m^2$	2200	0.72 pJ	$1534 \mu m^2$
		2300	0.74 pJ	$1590 \mu m^2$	2400	0.78 pJ	$1738 \mu m^2$
		2500	0.8 pJ	$1838 \mu m^2$	2600	0.84 pJ	$1995 \mu m^2$
		2700	0.89 pJ	$2127 \mu m^2$	2800	0.99 pJ	$2367 \mu m^2$
		2900	1 pJ	$2468 \mu m^2$	3000	1.07 pJ	$2458 \mu m^2$
		3100	1.15 pJ	$2712 \mu m^2$			
16	1	100	0.29 pJ	$466 \mu m^2$	200	0.29 pJ	$466 \mu m^2$
		300	0.29 pJ	$461 \mu m^2$	400	0.29 pJ	$442 \mu m^2$
		500	0.29 pJ	$442 \mu m^2$	600	0.29 pJ	$442 \mu m^2$
		700	0.29 pJ	$442 \mu m^2$	800	0.29 pJ	$438 \mu m^2$
		900	0.29 pJ	$448 \mu m^2$	1000	0.29 pJ	$437 \mu m^2$
		1100	0.3 pJ	$463 \mu m^2$	1200	0.29 pJ	$455 \mu m^2$
		1300	0.29 pJ	$463 \mu m^2$	1400	0.3 pJ	$487 \mu m^2$
		1500	0.31 pJ	$480 \mu m^2$	1600	0.31 pJ	$501 \mu m^2$
		1700	0.28 pJ	$432 \mu m^2$	1800	0.28 pJ	$461 \mu m^2$
		1900	0.3 pJ	$441 \mu m^2$	2000	0.29 pJ	$430 \mu m^2$
		2100	0.31 pJ	$507 \mu m^2$	2200	0.31 pJ	$499 \mu m^2$
		2300	0.33 pJ	$556 \mu m^2$	2400	0.33 pJ	$612 \mu m^2$
		2500	0.33 pJ	$605 \mu m^2$	2600	0.35 pJ	$666 \mu m^2$

*Continued on next page*

Table A.10: Benchmark results for Carry-Select Composition, Carry-Lookahead Base Adder (Section 5.3.4).

*Continued from previous page*

<i>w</i>	<i>n</i>	freq.	energy/op	area	freq.	energy/op	area
16	2	2700	0.36 pJ	708 $\mu m^2$	2800	0.39 pJ	742 $\mu m^2$
		2900	0.39 pJ	799 $\mu m^2$	3000	0.41 pJ	865 $\mu m^2$
		3100	0.43 pJ	874 $\mu m^2$	3200	0.46 pJ	989 $\mu m^2$
		3300	0.44 pJ	954 $\mu m^2$	3400	0.48 pJ	1041 $\mu m^2$
		3500	0.52 pJ	1101 $\mu m^2$	3600	0.58 pJ	1231 $\mu m^2$
		3700	0.59 pJ	1246 $\mu m^2$			
32	1	100	0.66 pJ	1185 $\mu m^2$	200	0.66 pJ	1185 $\mu m^2$
		300	0.66 pJ	1127 $\mu m^2$	400	0.66 pJ	1129 $\mu m^2$
		500	0.66 pJ	1125 $\mu m^2$	600	0.66 pJ	1176 $\mu m^2$
		700	0.66 pJ	1162 $\mu m^2$	800	0.67 pJ	1217 $\mu m^2$
		900	0.68 pJ	1232 $\mu m^2$	1000	0.69 pJ	1286 $\mu m^2$
		1100	0.7 pJ	1371 $\mu m^2$	1200	0.7 pJ	1367 $\mu m^2$
		1300	0.7 pJ	1352 $\mu m^2$	1400	0.71 pJ	1334 $\mu m^2$
		1500	0.72 pJ	1448 $\mu m^2$	1600	0.73 pJ	1440 $\mu m^2$
		1700	0.61 pJ	1397 $\mu m^2$	1800	0.63 pJ	1408 $\mu m^2$
		1900	0.66 pJ	1487 $\mu m^2$	2000	0.71 pJ	1739 $\mu m^2$
		2100	0.73 pJ	1754 $\mu m^2$	2200	0.75 pJ	1705 $\mu m^2$
		2300	0.82 pJ	1836 $\mu m^2$	2400	0.85 pJ	2048 $\mu m^2$
		2500	0.93 pJ	2135 $\mu m^2$			

## A.2 Accumulation

Table A.11: Benchmark results for Accumulation Baseline — 100 steps (Chapter 6).

$w$	$n$	pipeline depth	pipeline delay	interconnect	FUs	delays
8	1	100	100	808	1	0
16	1	100	100	1616	1	0
32	1	100	100	3232	1	0

Table A.12: Benchmark results for Accumulation Baseline — 100 steps (Chapter 6).

$w$	$n$	freq.	energy/op	area	freq.	energy/op	area
8	1	100	0.29 pJ	$285 \mu m^2$	200	0.29 pJ	$279 \mu m^2$
		300	0.29 pJ	$283 \mu m^2$	400	0.29 pJ	$293 \mu m^2$
		500	0.29 pJ	$285 \mu m^2$	600	0.29 pJ	$285 \mu m^2$
		700	0.29 pJ	$274 \mu m^2$	800	0.29 pJ	$289 \mu m^2$
		900	0.29 pJ	$298 \mu m^2$	1000	0.29 pJ	$297 \mu m^2$
		1100	0.26 pJ	$302 \mu m^2$	1200	0.27 pJ	$317 \mu m^2$
		1300	0.29 pJ	$325 \mu m^2$	1400	0.31 pJ	$319 \mu m^2$
		1500	0.31 pJ	$354 \mu m^2$	1600	0.33 pJ	$371 \mu m^2$
		1700	0.32 pJ	$366 \mu m^2$	1800	0.34 pJ	$365 \mu m^2$
		1900	0.37 pJ	$432 \mu m^2$	2000	0.36 pJ	$443 \mu m^2$
		2100	0.39 pJ	$504 \mu m^2$	2200	0.4 pJ	$559 \mu m^2$
		2300	0.45 pJ	$585 \mu m^2$	2400	0.43 pJ	$623 \mu m^2$
		2500	0.47 pJ	$649 \mu m^2$	2600	0.52 pJ	$676 \mu m^2$
16	1	100	0.6 pJ	$648 \mu m^2$	200	0.6 pJ	$633 \mu m^2$
		300	0.6 pJ	$620 \mu m^2$	400	0.6 pJ	$637 \mu m^2$
		500	0.6 pJ	$628 \mu m^2$	600	0.59 pJ	$631 \mu m^2$
		700	0.64 pJ	$659 \mu m^2$	800	0.62 pJ	$745 \mu m^2$
		900	0.62 pJ	$751 \mu m^2$	1000	0.63 pJ	$783 \mu m^2$
		1100	0.63 pJ	$813 \mu m^2$	1200	0.68 pJ	$752 \mu m^2$
		1300	0.7 pJ	$846 \mu m^2$	1400	0.71 pJ	$846 \mu m^2$
		1500	0.75 pJ	$899 \mu m^2$	1600	0.76 pJ	$936 \mu m^2$
		1700	0.8 pJ	$1041 \mu m^2$	1800	0.9 pJ	$1227 \mu m^2$
		1900	0.98 pJ	$1354 \mu m^2$	2000	1.04 pJ	$1516 \mu m^2$
		2100	1.06 pJ	$1485 \mu m^2$	2200	1.2 pJ	$1595 \mu m^2$
32	1	100	1.22 pJ	$1341 \mu m^2$	200	1.22 pJ	$1295 \mu m^2$
		300	1.22 pJ	$1334 \mu m^2$	400	1.3 pJ	$1472 \mu m^2$
		500	1.35 pJ	$1503 \mu m^2$	600	1.33 pJ	$1514 \mu m^2$
		700	1.36 pJ	$1677 \mu m^2$	800	1.36 pJ	$1761 \mu m^2$
		900	1.38 pJ	$1790 \mu m^2$	1000	1.48 pJ	$1769 \mu m^2$
		1100	1.5 pJ	$1854 \mu m^2$	1200	1.51 pJ	$1890 \mu m^2$
		1300	1.57 pJ	$2019 \mu m^2$	1400	1.69 pJ	$2150 \mu m^2$

*Continued on next page*

Table A.12: Benchmark results for Accumulation Baseline — 100 steps (Chapter 6).

<i>Continued from previous page</i>						
<i>w</i>	<i>n</i>	freq.	energy/op	area	freq.	energy/op
		1500	1.7 pJ	2277 $\mu m^2$	1600	1.75 pJ
		1700	2.12 pJ	2984 $\mu m^2$	1800	2.26 pJ

### A.2.1 Ripple-Carry Composition, Carry-Lookahead Base

Table A.13: Benchmark results for Ripple-Carry Composition, Carry-Lookahead Base Accumulator — 100 steps (Section 6.3.1).

<i>w</i>	<i>n</i>	pipeline depth	pipeline delay	interconnect	FUs	delays
8	1	100	100	808	1	0
8	2	100	100	1716	2	0
8	4	100	100	3532	4	0
16	1	100	100	1616	1	0
16	2	100	100	3332	2	0
32	1	100	100	3232	1	0

Table A.14: Benchmark results for Ripple-Carry Composition, Carry-Lookahead Base Accumulator — 100 steps (Section 6.3.1).

<i>w</i>	<i>n</i>	freq.	energy/op	area	freq.	energy/op	area
8	1	100	0.3 pJ	306 $\mu m^2$	200	0.3 pJ	313 $\mu m^2$
		300	0.3 pJ	293 $\mu m^2$	400	0.3 pJ	298 $\mu m^2$
		500	0.3 pJ	299 $\mu m^2$	600	0.3 pJ	298 $\mu m^2$
		700	0.3 pJ	305 $\mu m^2$	800	0.3 pJ	308 $\mu m^2$
		900	0.31 pJ	331 $\mu m^2$	1000	0.32 pJ	333 $\mu m^2$
		1100	0.3 pJ	352 $\mu m^2$	1200	0.29 pJ	330 $\mu m^2$
		1300	0.3 pJ	349 $\mu m^2$	1400	0.32 pJ	358 $\mu m^2$
		1500	0.34 pJ	389 $\mu m^2$	1600	0.35 pJ	413 $\mu m^2$
		1700	0.35 pJ	436 $\mu m^2$	1800	0.36 pJ	471 $\mu m^2$
		1900	0.38 pJ	502 $\mu m^2$	2000	0.43 pJ	588 $\mu m^2$
		2100	0.43 pJ	619 $\mu m^2$	2200	0.47 pJ	670 $\mu m^2$
		2300	0.5 pJ	723 $\mu m^2$	2400	0.55 pJ	793 $\mu m^2$
		2500	0.61 pJ	859 $\mu m^2$			
	2	100	0.64 pJ	697 $\mu m^2$	200	0.65 pJ	675 $\mu m^2$

*Continued on next page*

Table A.14: Benchmark results for Ripple-Carry Composition, Carry-Lookahead Base Accumulator — 100 steps (Section 6.3.1).

*Continued from previous page*

<i>w</i>	<i>n</i>	freq.	energy/op	area	freq.	energy/op	area
8	4	300	0.62 pJ	691 $\mu m^2$	400	0.62 pJ	639 $\mu m^2$
		500	0.63 pJ	664 $\mu m^2$	600	0.62 pJ	702 $\mu m^2$
		700	0.63 pJ	761 $\mu m^2$	800	0.65 pJ	786 $\mu m^2$
		900	0.64 pJ	807 $\mu m^2$	1000	0.63 pJ	772 $\mu m^2$
		1100	0.61 pJ	817 $\mu m^2$	1200	0.66 pJ	855 $\mu m^2$
		1300	0.72 pJ	956 $\mu m^2$	1400	0.72 pJ	885 $\mu m^2$
		1500	0.75 pJ	964 $\mu m^2$	1600	0.84 pJ	1058 $\mu m^2$
		1700	0.87 pJ	1236 $\mu m^2$	1800	0.93 pJ	1359 $\mu m^2$
		1900	1.03 pJ	1384 $\mu m^2$	2000	1.09 pJ	1521 $\mu m^2$
		100	1.3 pJ	1500 $\mu m^2$	200	1.29 pJ	1400 $\mu m^2$
16	1	300	1.25 pJ	1533 $\mu m^2$	400	1.31 pJ	1609 $\mu m^2$
		500	1.32 pJ	1571 $\mu m^2$	600	1.37 pJ	1679 $\mu m^2$
		700	1.34 pJ	1626 $\mu m^2$	800	1.35 pJ	1610 $\mu m^2$
		900	1.38 pJ	1758 $\mu m^2$	1000	1.37 pJ	1800 $\mu m^2$
		1100	1.4 pJ	1838 $\mu m^2$	1200	1.49 pJ	2020 $\mu m^2$
		1300	1.6 pJ	2235 $\mu m^2$	1400	1.75 pJ	2325 $\mu m^2$
		1500	1.91 pJ	2538 $\mu m^2$			
		100	0.63 pJ	787 $\mu m^2$	200	0.63 pJ	787 $\mu m^2$
		300	0.62 pJ	780 $\mu m^2$	400	0.62 pJ	799 $\mu m^2$
		500	0.62 pJ	765 $\mu m^2$	600	0.62 pJ	794 $\mu m^2$
16	2	700	0.62 pJ	791 $\mu m^2$	800	0.61 pJ	785 $\mu m^2$
		900	0.63 pJ	826 $\mu m^2$	1000	0.64 pJ	830 $\mu m^2$
		1100	0.64 pJ	766 $\mu m^2$	1200	0.7 pJ	811 $\mu m^2$
		1300	0.69 pJ	839 $\mu m^2$	1400	0.73 pJ	892 $\mu m^2$
		1500	0.74 pJ	957 $\mu m^2$	1600	0.79 pJ	1029 $\mu m^2$
		1700	0.86 pJ	1191 $\mu m^2$	1800	0.92 pJ	1266 $\mu m^2$
		1900	1.07 pJ	1529 $\mu m^2$	2000	1.14 pJ	1689 $\mu m^2$
		2100	1.29 pJ	1772 $\mu m^2$	2200	1.44 pJ	1936 $\mu m^2$
		100	1.28 pJ	1668 $\mu m^2$	200	1.27 pJ	1595 $\mu m^2$
		300	1.25 pJ	1639 $\mu m^2$	400	1.25 pJ	1589 $\mu m^2$
32	1	500	1.26 pJ	1591 $\mu m^2$	600	1.29 pJ	1662 $\mu m^2$
		700	1.29 pJ	1617 $\mu m^2$	800	1.24 pJ	1555 $\mu m^2$
		900	1.28 pJ	1738 $\mu m^2$	1000	1.28 pJ	1637 $\mu m^2$
		1100	1.4 pJ	1712 $\mu m^2$	1200	1.47 pJ	1934 $\mu m^2$
		1300	1.6 pJ	2123 $\mu m^2$	1400	1.78 pJ	2401 $\mu m^2$
		1500	1.87 pJ	2553 $\mu m^2$	1600	1.99 pJ	2617 $\mu m^2$
		1700	2.24 pJ	2944 $\mu m^2$			

*Continued on next page*

Table A.14: Benchmark results for Ripple-Carry Composition, Carry-Lookahead Base Accumulator — 100 steps (Section 6.3.1).

*Continued from previous page*

$w$	$n$	freq.	energy/op	area	freq.	energy/op	area
		1300	1.62 pJ	$2080 \mu m^2$	1400	1.73 pJ	$2246 \mu m^2$
		1700	2.31 pJ	$3047 \mu m^2$			

### A.2.2 Ripple-Carry Composition, Carry-Save Base

Table A.15: Benchmark results for Ripple-Carry Composition, Carry-Save Base Accumulator — 100 steps (Section 6.3.2).

$w$	$n$	pipeline depth	pipeline delay	interconnect	FUs	delays
8	1	101	101	808	1	0
8	2	101	101	1717	2	0
8	4	101	101	3535	4	0
16	1	101	101	1616	1	0
16	2	101	101	3333	2	0
32	1	101	101	3232	1	0

Table A.16: Benchmark results for Ripple-Carry Composition, Carry-Save Base Accumulator — 100 steps (Section 6.3.2).

$w$	$n$	freq.	energy/op	area	freq.	energy/op	area
8	1	100	0.34 pJ	$562 \mu m^2$	200	0.34 pJ	$573 \mu m^2$
		300	0.34 pJ	$585 \mu m^2$	400	0.32 pJ	$511 \mu m^2$
		500	0.32 pJ	$523 \mu m^2$	600	0.32 pJ	$524 \mu m^2$
		700	0.32 pJ	$498 \mu m^2$	800	0.32 pJ	$522 \mu m^2$
		900	0.32 pJ	$518 \mu m^2$	1000	0.32 pJ	$533 \mu m^2$
		1100	0.32 pJ	$531 \mu m^2$	1200	0.32 pJ	$535 \mu m^2$
		1300	0.32 pJ	$551 \mu m^2$	1400	0.37 pJ	$553 \mu m^2$
		1500	0.37 pJ	$550 \mu m^2$	1600	0.36 pJ	$597 \mu m^2$
		1700	0.36 pJ	$638 \mu m^2$	1800	0.36 pJ	$621 \mu m^2$
		1900	0.36 pJ	$648 \mu m^2$	2000	0.36 pJ	$689 \mu m^2$
		2100	0.38 pJ	$706 \mu m^2$	2200	0.37 pJ	$761 \mu m^2$
		2300	0.41 pJ	$819 \mu m^2$	2400	0.4 pJ	$861 \mu m^2$
		2500	0.44 pJ	$1008 \mu m^2$	2600	0.49 pJ	$1068 \mu m^2$
		2700	0.47 pJ	$1024 \mu m^2$	2800	0.53 pJ	$1127 \mu m^2$
		8	2	$100$	$0.69 \mu m^2$	$1239 \mu m^2$	$200$

*Continued on next page*

Table A.16: Benchmark results for Ripple-Carry Composition, Carry-Save Base Accumulator — 100 steps (Section 6.3.2).

*Continued from previous page*

<i>w</i>	<i>n</i>	freq.	energy/op	area	freq.	energy/op	area
8	4	300	0.69 pJ	$1223 \mu m^2$	400	0.66 pJ	$1101 \mu m^2$
		500	0.66 pJ	$1067 \mu m^2$	600	0.66 pJ	$1095 \mu m^2$
		700	0.65 pJ	$1135 \mu m^2$	800	0.66 pJ	$1254 \mu m^2$
		900	0.66 pJ	$1146 \mu m^2$	1000	0.74 pJ	$1110 \mu m^2$
		1100	0.75 pJ	$1167 \mu m^2$	1200	0.74 pJ	$1220 \mu m^2$
		1300	0.75 pJ	$1204 \mu m^2$	1400	0.76 pJ	$1304 \mu m^2$
		1500	0.76 pJ	$1382 \mu m^2$	1600	0.77 pJ	$1504 \mu m^2$
		1700	0.81 pJ	$1559 \mu m^2$	1800	0.82 pJ	$1607 \mu m^2$
		1900	0.84 pJ	$1587 \mu m^2$	2000	0.87 pJ	$1675 \mu m^2$
		2100	0.95 pJ	$1694 \mu m^2$			
16	1	100	1.39 pJ	$2480 \mu m^2$	200	1.39 pJ	$2384 \mu m^2$
		300	1.39 pJ	$2448 \mu m^2$	400	1.33 pJ	$2464 \mu m^2$
		500	1.32 pJ	$2428 \mu m^2$	600	1.34 pJ	$2514 \mu m^2$
		700	1.49 pJ	$2380 \mu m^2$	800	1.5 pJ	$2470 \mu m^2$
		900	1.5 pJ	$2521 \mu m^2$	1000	1.54 pJ	$2560 \mu m^2$
		1100	1.54 pJ	$2731 \mu m^2$	1200	1.58 pJ	$2789 \mu m^2$
		1300	1.59 pJ	$2903 \mu m^2$	1400	1.67 pJ	$2998 \mu m^2$
		1500	1.81 pJ	$3256 \mu m^2$			
	2	100	0.7 pJ	$1388 \mu m^2$	200	0.7 pJ	$1369 \mu m^2$
		300	0.7 pJ	$1312 \mu m^2$	400	0.72 pJ	$1206 \mu m^2$
		500	0.72 pJ	$1254 \mu m^2$	600	0.72 pJ	$1242 \mu m^2$
		700	0.72 pJ	$1351 \mu m^2$	800	0.72 pJ	$1266 \mu m^2$
		900	0.72 pJ	$1287 \mu m^2$	1000	0.72 pJ	$1331 \mu m^2$
32	1	1100	0.71 pJ	$1315 \mu m^2$	1200	0.82 pJ	$1210 \mu m^2$
		1300	0.82 pJ	$1208 \mu m^2$	1400	0.82 pJ	$1271 \mu m^2$
		1500	0.76 pJ	$1260 \mu m^2$	1600	0.76 pJ	$1269 \mu m^2$
		1700	0.79 pJ	$1375 \mu m^2$	1800	0.81 pJ	$1418 \mu m^2$
		1900	0.87 pJ	$1625 \mu m^2$	2000	0.92 pJ	$1689 \mu m^2$
		2100	0.95 pJ	$1682 \mu m^2$	2200	0.97 pJ	$1828 \mu m^2$
		2300	1.11 pJ	$1879 \mu m^2$			
		100	1.41 pJ	$2547 \mu m^2$	200	1.41 pJ	$2624 \mu m^2$
		300	1.41 pJ	$2560 \mu m^2$	400	1.45 pJ	$2460 \mu m^2$

*Continued on next page*

Table A.16: Benchmark results for Ripple-Carry Composition, Carry-Save Base Accumulator — 100 steps (Section 6.3.2).

*Continued from previous page*

$w$	$n$	freq.	energy/op	area	freq.	energy/op	area
		1100	1.22 pJ	$2906 \mu m^2$	1200	1.23 pJ	$3016 \mu m^2$
		1300	1.21 pJ	$3190 \mu m^2$	1400	1.3 pJ	$3350 \mu m^2$
		1500	1.33 pJ	$3622 \mu m^2$	1600	1.37 pJ	$3786 \mu m^2$
		1700	1.36 pJ	$3882 \mu m^2$	1800	1.53 pJ	$4247 \mu m^2$
		1900	1.64 pJ	$4374 \mu m^2$			

### A.2.3 Carry-Accumulate Composition, Carry-Lookahead Base

Table A.17: Benchmark results for Carry-Accumulate Composition, Carry-Lookahead Base Accumulator — 100 steps (Section 6.3.3).

$w$	$n$	pipeline depth	pipeline delay	interconnect	FUs	delays
8	1	100	100	808	1	0
8	2	101	101	1624	2	0
8	4	103	103	3256	4	2
16	1	100	100	1616	1	0
16	2	101	101	3248	2	0
16	4	103	103	6512	4	2
32	1	100	100	3232	1	0
32	2	101	101	6496	2	0
32	4	103	103	13024	4	2

Table A.18: Benchmark results for Carry-Accumulate Composition, Carry-Lookahead Base Accumulator — 100 steps (Section 6.3.3).

$w$	$n$	freq.	energy/op	area	freq.	energy/op	area
8	1	100	0.36 pJ	$492 \mu m^2$	200	0.36 pJ	$516 \mu m^2$
		300	0.35 pJ	$498 \mu m^2$	400	0.35 pJ	$483 \mu m^2$
		500	0.36 pJ	$497 \mu m^2$	600	0.36 pJ	$492 \mu m^2$
		700	0.36 pJ	$539 \mu m^2$	800	0.4 pJ	$554 \mu m^2$
		900	0.41 pJ	$570 \mu m^2$	1000	0.37 pJ	$565 \mu m^2$
		1100	0.4 pJ	$599 \mu m^2$	1200	0.42 pJ	$642 \mu m^2$
		1300	0.42 pJ	$643 \mu m^2$	1400	0.45 pJ	$670 \mu m^2$

*Continued on next page*

Table A.18: Benchmark results for Carry-Accumulate Composition, Carry-Lookahead Base Accumulator — 100 steps (Section 6.3.3).

*Continued from previous page*

<i>w</i>	<i>n</i>	freq.	energy/op	area	freq.	energy/op	area
8	2	1500	0.46 pJ	698 $\mu\text{m}^2$	1600	0.47 pJ	739 $\mu\text{m}^2$
		1700	0.51 pJ	764 $\mu\text{m}^2$	1800	0.51 pJ	827 $\mu\text{m}^2$
		1900	0.54 pJ	891 $\mu\text{m}^2$	2000	0.54 pJ	1004 $\mu\text{m}^2$
		2100	0.6 pJ	1143 $\mu\text{m}^2$	2200	0.58 pJ	1109 $\mu\text{m}^2$
		2300	0.65 pJ	1204 $\mu\text{m}^2$	2400	0.72 pJ	1276 $\mu\text{m}^2$
8	4	100	0.72 pJ	1015 $\mu\text{m}^2$	200	0.72 pJ	1061 $\mu\text{m}^2$
		300	0.71 pJ	1009 $\mu\text{m}^2$	400	0.71 pJ	1027 $\mu\text{m}^2$
		500	0.72 pJ	1001 $\mu\text{m}^2$	600	0.71 pJ	1013 $\mu\text{m}^2$
		700	0.72 pJ	1032 $\mu\text{m}^2$	800	0.79 pJ	1155 $\mu\text{m}^2$
		900	0.82 pJ	1163 $\mu\text{m}^2$	1000	0.74 pJ	1177 $\mu\text{m}^2$
		1100	0.8 pJ	1223 $\mu\text{m}^2$	1200	0.85 pJ	1257 $\mu\text{m}^2$
		1300	0.83 pJ	1269 $\mu\text{m}^2$	1400	0.9 pJ	1296 $\mu\text{m}^2$
		1500	0.9 pJ	1368 $\mu\text{m}^2$	1600	0.93 pJ	1483 $\mu\text{m}^2$
		1700	1.01 pJ	1625 $\mu\text{m}^2$	1800	1.05 pJ	1603 $\mu\text{m}^2$
		1900	1.05 pJ	1697 $\mu\text{m}^2$	2000	1.1 pJ	1902 $\mu\text{m}^2$
		2100	1.19 pJ	2030 $\mu\text{m}^2$	2200	1.25 pJ	2160 $\mu\text{m}^2$
		2300	1.33 pJ	2383 $\mu\text{m}^2$	2400	1.41 pJ	2592 $\mu\text{m}^2$
16	1	100	1.55 pJ	2379 $\mu\text{m}^2$	200	1.55 pJ	2255 $\mu\text{m}^2$
		300	1.53 pJ	2239 $\mu\text{m}^2$	400	1.53 pJ	2249 $\mu\text{m}^2$
		500	1.55 pJ	2345 $\mu\text{m}^2$	600	1.53 pJ	2318 $\mu\text{m}^2$
		700	1.54 pJ	2400 $\mu\text{m}^2$	800	1.72 pJ	2710 $\mu\text{m}^2$
		900	1.74 pJ	2666 $\mu\text{m}^2$	1000	1.59 pJ	2562 $\mu\text{m}^2$
		1100	1.72 pJ	2641 $\mu\text{m}^2$	1200	1.82 pJ	2657 $\mu\text{m}^2$
		1300	1.79 pJ	2824 $\mu\text{m}^2$	1400	1.94 pJ	2832 $\mu\text{m}^2$
		1500	1.92 pJ	3089 $\mu\text{m}^2$	1600	2 pJ	3342 $\mu\text{m}^2$
		1700	2.12 pJ	3421 $\mu\text{m}^2$	1800	2.18 pJ	3751 $\mu\text{m}^2$
		1900	2.26 pJ	3968 $\mu\text{m}^2$	2000	2.26 pJ	4295 $\mu\text{m}^2$
16	2	100	1.49 pJ	2251 $\mu\text{m}^2$	200	1.49 pJ	2156 $\mu\text{m}^2$
		300	1.52 pJ	2242 $\mu\text{m}^2$	400	1.49 pJ	2225 $\mu\text{m}^2$
		500	1.57 pJ	2281 $\mu\text{m}^2$	600	1.57 pJ	2440 $\mu\text{m}^2$
		700	1.53 pJ	2403 $\mu\text{m}^2$	800	1.56 pJ	2615 $\mu\text{m}^2$
		900	1.51 pJ	2694 $\mu\text{m}^2$	1000	1.58 pJ	2588 $\mu\text{m}^2$
		1100	1.63 pJ	2707 $\mu\text{m}^2$	1200	1.75 pJ	2805 $\mu\text{m}^2$

*Continued on next page*

Table A.18: Benchmark results for Carry-Accumulate Composition, Carry-Lookahead Base Accumulator — 100 steps (Section 6.3.3).

*Continued from previous page*

$w$	$n$	freq.	energy/op	area	freq.	energy/op	area
16	4	1300	1.79 pJ	$3075 \mu m^2$	1400	1.88 pJ	$2964 \mu m^2$
		1500	1.97 pJ	$3392 \mu m^2$	1600	1.98 pJ	$3318 \mu m^2$
		1700	2.19 pJ	$4076 \mu m^2$	1800	2.33 pJ	$4294 \mu m^2$
		1900	2.6 pJ	$4502 \mu m^2$			
32	1	100	3.22 pJ	$5055 \mu m^2$	200	3.22 pJ	$4836 \mu m^2$
		300	3.23 pJ	$4926 \mu m^2$	400	3.22 pJ	$5478 \mu m^2$
		500	3.38 pJ	$5684 \mu m^2$	600	3.3 pJ	$5635 \mu m^2$
		700	3.3 pJ	$6033 \mu m^2$	800	3.34 pJ	$6023 \mu m^2$
		900	3.24 pJ	$6130 \mu m^2$	1000	3.38 pJ	$6173 \mu m^2$
		1100	3.49 pJ	$6530 \mu m^2$	1200	3.73 pJ	$6651 \mu m^2$
		1300	3.86 pJ	$6924 \mu m^2$	1400	4.02 pJ	$6799 \mu m^2$
		1500	4.17 pJ	$6995 \mu m^2$	1600	4.25 pJ	$7291 \mu m^2$
		1700	4.5 pJ	$8058 \mu m^2$	1800	4.97 pJ	$8888 \mu m^2$
32	2	100	1.56 pJ	$2411 \mu m^2$	200	1.57 pJ	$2412 \mu m^2$
		300	1.62 pJ	$2592 \mu m^2$	400	1.61 pJ	$2765 \mu m^2$
		500	1.5 pJ	$2807 \mu m^2$	600	1.5 pJ	$2950 \mu m^2$
		700	1.54 pJ	$2956 \mu m^2$	800	1.57 pJ	$3057 \mu m^2$
		900	1.62 pJ	$2992 \mu m^2$	1000	1.72 pJ	$2906 \mu m^2$
		1100	1.76 pJ	$3293 \mu m^2$	1200	1.8 pJ	$3351 \mu m^2$
		1300	1.93 pJ	$3628 \mu m^2$	1400	2.12 pJ	$3781 \mu m^2$
		1500	2.26 pJ	$3796 \mu m^2$	1600	2.5 pJ	$4423 \mu m^2$
32	4	100	3.12 pJ	$4851 \mu m^2$	200	3.14 pJ	$5029 \mu m^2$
		300	3.25 pJ	$5131 \mu m^2$	400	3.19 pJ	$5597 \mu m^2$
		500	3.01 pJ	$5774 \mu m^2$	600	3.04 pJ	$5958 \mu m^2$
		700	3.11 pJ	$5692 \mu m^2$	800	3.11 pJ	$5687 \mu m^2$
		900	3.23 pJ	$6000 \mu m^2$	1000	3.4 pJ	$5856 \mu m^2$
		1100	3.5 pJ	$6430 \mu m^2$	1200	3.64 pJ	$7400 \mu m^2$
		1300	3.84 pJ	$7022 \mu m^2$	1400	4.19 pJ	$7738 \mu m^2$
		1500	4.43 pJ	$7672 \mu m^2$	1600	5.2 pJ	$8913 \mu m^2$

#### A.2.4 Carry-Accumulate Composition, Carry-Save Base

Table A.19: Benchmark results for Carry-Accumulate Composition, Carry-Save Base Accumulator — 100 steps (Section 6.3.4).

$w$	$n$	pipeline depth	pipeline delay	interconnect	FUs	delays
8	1	100	100	808	1	0
8	2	101	101	1624	2	0
8	4	103	103	3256	4	2
16	1	100	100	1616	1	0
16	2	101	101	3248	2	0
16	4	103	103	6512	4	2
32	1	100	100	3232	1	0
32	2	101	101	6496	2	0
32	4	103	103	13024	4	2

Table A.20: Benchmark results for Carry-Accumulate Composition, Carry-Save Base Accumulator — 100 steps (Section 6.3.4).

$w$	$n$	freq.	energy/op	area	freq.	energy/op	area
8	1	100	0.46 pJ	750 $\mu m^2$	200	0.47 pJ	784 $\mu m^2$
		300	0.47 pJ	730 $\mu m^2$	400	0.47 pJ	723 $\mu m^2$
		500	0.47 pJ	765 $\mu m^2$	600	0.47 pJ	781 $\mu m^2$
		700	0.46 pJ	804 $\mu m^2$	800	0.51 pJ	886 $\mu m^2$
		900	0.48 pJ	936 $\mu m^2$	1000	0.48 pJ	876 $\mu m^2$
		1100	0.47 pJ	1022 $\mu m^2$	1200	0.5 pJ	1117 $\mu m^2$
		1300	0.53 pJ	1143 $\mu m^2$	1400	0.6 pJ	1248 $\mu m^2$
		1500	0.69 pJ	1433 $\mu m^2$	1600	0.76 pJ	1514 $\mu m^2$
		1700	0.95 pJ	1869 $\mu m^2$	1800	1.17 pJ	2081 $\mu m^2$
8	2	100	0.93 pJ	1502 $\mu m^2$	200	0.94 pJ	1617 $\mu m^2$
		300	0.94 pJ	1553 $\mu m^2$	400	0.94 pJ	1436 $\mu m^2$
		500	0.95 pJ	1444 $\mu m^2$	600	0.94 pJ	1579 $\mu m^2$
		700	0.93 pJ	1623 $\mu m^2$	800	1.01 pJ	1768 $\mu m^2$
		900	0.96 pJ	1763 $\mu m^2$	1000	0.95 pJ	1756 $\mu m^2$
		1100	0.93 pJ	1949 $\mu m^2$	1200	0.97 pJ	2082 $\mu m^2$
		1300	1.04 pJ	2222 $\mu m^2$	1400	1.2 pJ	2509 $\mu m^2$
		1500	1.28 pJ	2664 $\mu m^2$	1600	1.51 pJ	3149 $\mu m^2$
		1700	1.84 pJ	3711 $\mu m^2$	1800	2.36 pJ	4026 $\mu m^2$
8	4	100	1.97 pJ	3376 $\mu m^2$	200	2.01 pJ	3397 $\mu m^2$
		300	2 pJ	3157 $\mu m^2$	400	2 pJ	3330 $\mu m^2$
		500	2.02 pJ	3312 $\mu m^2$	600	2.01 pJ	3419 $\mu m^2$
		700	1.97 pJ	3523 $\mu m^2$	800	2.13 pJ	3890 $\mu m^2$
		900	2.02 pJ	4034 $\mu m^2$	1000	2.04 pJ	3820 $\mu m^2$
		1100	2.01 pJ	4340 $\mu m^2$	1200	2.05 pJ	4796 $\mu m^2$
		1300	2.17 pJ	5013 $\mu m^2$	1400	2.53 pJ	5538 $\mu m^2$

*Continued on next page*

Table A.20: Benchmark results for Carry-Accumulate Composition, Carry-Save Base Accumulator — 100 steps (Section 6.3.4).

*Continued from previous page*

$w$	$n$	freq.	energy/op	area	freq.	energy/op	area
16	1	1500	2.77 pJ	$6157 \mu m^2$	1600	3.14 pJ	$6904 \mu m^2$
		1700	3.64 pJ	$7601 \mu m^2$	1800	4.74 pJ	$8501 \mu m^2$
		100	0.89 pJ	$1654 \mu m^2$	200	0.9 pJ	$1602 \mu m^2$
		300	0.89 pJ	$1751 \mu m^2$	400	0.9 pJ	$1765 \mu m^2$
		500	0.91 pJ	$1879 \mu m^2$	600	0.93 pJ	$1942 \mu m^2$
		700	0.92 pJ	$1971 \mu m^2$	800	0.95 pJ	$2125 \mu m^2$
	2	900	0.99 pJ	$2289 \mu m^2$	1000	1.03 pJ	$2410 \mu m^2$
		1100	1.06 pJ	$2458 \mu m^2$	1200	1.05 pJ	$2464 \mu m^2$
		100	1.77 pJ	$3342 \mu m^2$	200	1.79 pJ	$3186 \mu m^2$
		300	1.78 pJ	$3269 \mu m^2$	400	1.8 pJ	$3510 \mu m^2$
		500	1.8 pJ	$3792 \mu m^2$	600	1.83 pJ	$3877 \mu m^2$
		700	1.83 pJ	$4368 \mu m^2$	800	1.9 pJ	$4676 \mu m^2$
16	4	900	1.98 pJ	$5086 \mu m^2$	1000	2.03 pJ	$4818 \mu m^2$
		1100	2.13 pJ	$4984 \mu m^2$	1200	2.12 pJ	$5164 \mu m^2$
		1300	3 pJ	$5547 \mu m^2$			
		100	3.77 pJ	$7660 \mu m^2$	200	3.82 pJ	$7517 \mu m^2$
		300	3.79 pJ	$7549 \mu m^2$	400	3.83 pJ	$7582 \mu m^2$
		500	3.85 pJ	$8186 \mu m^2$	600	3.94 pJ	$8705 \mu m^2$
		700	3.93 pJ	$8821 \mu m^2$	800	4.03 pJ	$9219 \mu m^2$
32	1	900	4.18 pJ	$10023 \mu m^2$	1000	4.31 pJ	$10580 \mu m^2$
		1100	4.46 pJ	$10645 \mu m^2$	1200	4.55 pJ	$11086 \mu m^2$
		1300	6.53 pJ	$12219 \mu m^2$			
		100	1.76 pJ	$3959 \mu m^2$	200	1.77 pJ	$3935 \mu m^2$
		300	1.77 pJ	$4193 \mu m^2$	400	1.8 pJ	$4444 \mu m^2$
		500	1.79 pJ	$4576 \mu m^2$	600	1.83 pJ	$4908 \mu m^2$
32	2	700	1.76 pJ	$4848 \mu m^2$	800	1.83 pJ	$5099 \mu m^2$
		900	1.81 pJ	$5741 \mu m^2$	1000	1.95 pJ	$5835 \mu m^2$
		1100	1.97 pJ	$6093 \mu m^2$	1200	2.73 pJ	$7960 \mu m^2$
		100	3.5 pJ	$8337 \mu m^2$	200	3.53 pJ	$7794 \mu m^2$
		300	3.54 pJ	$8142 \mu m^2$	400	3.6 pJ	$8734 \mu m^2$
		500	3.59 pJ	$8939 \mu m^2$	600	3.67 pJ	$9678 \mu m^2$
32	4	700	3.56 pJ	$9605 \mu m^2$	800	3.69 pJ	$10986 \mu m^2$
		900	3.66 pJ	$11783 \mu m^2$	1000	3.9 pJ	$12400 \mu m^2$
		1100	3.98 pJ	$12360 \mu m^2$	1200	5.02 pJ	$14999 \mu m^2$
		100	7.48 pJ	$17223 \mu m^2$	200	7.54 pJ	$17141 \mu m^2$
		300	7.56 pJ	$17924 \mu m^2$	400	7.67 pJ	$18464 \mu m^2$
		500	7.67 pJ	$19304 \mu m^2$	600	7.86 pJ	$20725 \mu m^2$
		700	7.6 pJ	$20927 \mu m^2$	800	7.85 pJ	$21584 \mu m^2$
		900	7.95 pJ	$24417 \mu m^2$	1000	8.44 pJ	$25331 \mu m^2$
		1100	8.56 pJ	$27201 \mu m^2$	1200	10.61 pJ	$30558 \mu m^2$

### A.3 Multiplication

Table A.21: Benchmark results for Full-Width Multiplication Baseline (Chapter 7).

$w$	$n$	pipeline depth	pipeline delay	interconnect	FUs	delays
8	1	1	1	32	1	0
16	1	1	1	64	1	0
32	1	1	1	128	1	0

Table A.22: Benchmark results for Full-Width Multiplication Baseline (Chapter 7).

$w$	$n$	freq.	energy/op	area	freq.	energy/op	area
8	1	100	0.83 pJ	1479 $\mu m^2$	200	0.84 pJ	1507 $\mu m^2$
		300	0.87 pJ	1452 $\mu m^2$	400	0.89 pJ	1482 $\mu m^2$
		500	0.87 pJ	1435 $\mu m^2$	600	0.87 pJ	1598 $\mu m^2$
		700	1.03 pJ	1496 $\mu m^2$	800	1.17 pJ	1930 $\mu m^2$
		900	1.18 pJ	1915 $\mu m^2$	1000	1.33 pJ	2090 $\mu m^2$
		1100	1.41 pJ	2208 $\mu m^2$	1200	1.74 pJ	2975 $\mu m^2$
		1300	2.32 pJ	3589 $\mu m^2$			
16	1	100	3.17 pJ	5501 $\mu m^2$	200	3.19 pJ	5456 $\mu m^2$
		300	3.21 pJ	5415 $\mu m^2$	400	3.59 pJ	5987 $\mu m^2$
		500	3.82 pJ	6062 $\mu m^2$	600	3.91 pJ	6156 $\mu m^2$
		700	4.04 pJ	6707 $\mu m^2$	800	4.89 pJ	8355 $\mu m^2$
		900	9.8 pJ	13591 $\mu m^2$			
32	1	100	13.18 pJ	21471 $\mu m^2$	200	14.84 pJ	23916 $\mu m^2$
		300	15.08 pJ	25058 $\mu m^2$	400	15.36 pJ	27500 $\mu m^2$
		500	15.37 pJ	30601 $\mu m^2$	600	17.59 pJ	32276 $\mu m^2$

Table A.23: Benchmark results for Half-Width Multiplication Baseline (Chapter 7).

$w$	$n$	pipeline depth	pipeline delay	interconnect	FUs	delays
8	1	1	1	24	1	0
16	1	1	1	48	1	0
32	1	1	1	96	1	0

Table A.24: Benchmark results for Half-Width Multiplication Baseline (Chapter 7).

$w$	$n$	freq.	energy/op	area	freq.	energy/op	area
8	1	100	0.26 pJ	$633 \mu m^2$	200	0.26 pJ	$608 \mu m^2$
		300	0.26 pJ	$601 \mu m^2$	400	0.26 pJ	$614 \mu m^2$
		500	0.26 pJ	$608 \mu m^2$	600	0.26 pJ	$608 \mu m^2$
		700	0.26 pJ	$598 \mu m^2$	800	0.26 pJ	$631 \mu m^2$
		900	0.26 pJ	$599 \mu m^2$	1000	0.27 pJ	$671 \mu m^2$
		1100	0.28 pJ	$733 \mu m^2$	1200	0.31 pJ	$707 \mu m^2$
		1300	0.39 pJ	$868 \mu m^2$	1400	0.35 pJ	$849 \mu m^2$
		1500	0.41 pJ	$993 \mu m^2$	1600	0.5 pJ	$1221 \mu m^2$
		1700	0.55 pJ	$1365 \mu m^2$	1800	0.48 pJ	$1284 \mu m^2$
		1900	0.59 pJ	$1485 \mu m^2$	2000	0.77 pJ	$1719 \mu m^2$
		2100	0.84 pJ	$1890 \mu m^2$			
16	1	100	1.06 pJ	$2511 \mu m^2$	200	1.04 pJ	$2435 \mu m^2$
		300	1.05 pJ	$2333 \mu m^2$	400	1.07 pJ	$2353 \mu m^2$
		500	1.09 pJ	$2302 \mu m^2$	600	0.98 pJ	$2795 \mu m^2$
		700	1.24 pJ	$2582 \mu m^2$	800	1.51 pJ	$2949 \mu m^2$
		900	1.59 pJ	$3081 \mu m^2$	1000	1.85 pJ	$3846 \mu m^2$
		1100	2.63 pJ	$5109 \mu m^2$	1200	3.4 pJ	$6123 \mu m^2$
32	1	100	4.19 pJ	$9406 \mu m^2$	200	4.26 pJ	$9089 \mu m^2$
		300	4.36 pJ	$8822 \mu m^2$	400	5.09 pJ	$9998 \mu m^2$
		500	5.25 pJ	$10258 \mu m^2$	600	5.02 pJ	$10993 \mu m^2$
		700	5.66 pJ	$12458 \mu m^2$	800	9.24 pJ	$19744 \mu m^2$

### A.3.1 Single-Cycle Multiplier With Adder Tree

Table A.25: Benchmark results for Full-Width Single-Cycle Multiplier With Adder Tree (Section 7.3.1).

$w$	$n$	pipeline depth	pipeline delay	interconnect	FUs	delays
8	1	1	1	32	1	0
8	2	3	1	180	10	4
8	4	4	1	830	52	9
16	1	1	1	64	1	0
16	2	3	1	356	10	4
32	1	1	1	128	1	0

Table A.26: Benchmark results for Full-Width Single-Cycle Multiplier With Adder Tree (Section 7.3.1).

$w$	$n$	freq.	energy/op	area	freq.	energy/op	area
8	1	100	1.14 pJ	$1909 \mu m^2$	200	1.1 pJ	$1873 \mu m^2$
		300	1.11 pJ	$1821 \mu m^2$	400	1.11 pJ	$1757 \mu m^2$
		500	1.14 pJ	$1761 \mu m^2$	600	1.15 pJ	$1776 \mu m^2$
		700	1.37 pJ	$2156 \mu m^2$	800	1.48 pJ	$2341 \mu m^2$
		900	1.64 pJ	$2557 \mu m^2$	1000	1.85 pJ	$2819 \mu m^2$
		1100	2.49 pJ	$3573 \mu m^2$	1200	2.91 pJ	$4010 \mu m^2$
		1300	3.91 pJ	$4565 \mu m^2$			
8	2	100	5.88 pJ	$20045 \mu m^2$	200	5.85 pJ	$19833 \mu m^2$
		300	6 pJ	$20554 \mu m^2$	400	6.05 pJ	$20184 \mu m^2$
		500	6.31 pJ	$22791 \mu m^2$	600	6.33 pJ	$23584 \mu m^2$
		700	6.54 pJ	$25040 \mu m^2$	800	6.98 pJ	$28168 \mu m^2$
		900	8.06 pJ	$31029 \mu m^2$	1000	9.42 pJ	$36391 \mu m^2$
8	4	100	23.48 pJ	$121211 \mu m^2$	200	23.96 pJ	$122340 \mu m^2$
		300	24.8 pJ	$135493 \mu m^2$	400	24.73 pJ	$138379 \mu m^2$
		500	25.28 pJ	$142896 \mu m^2$	600	25.46 pJ	$147045 \mu m^2$
16	1	100	4.39 pJ	$6823 \mu m^2$	200	3.95 pJ	$6869 \mu m^2$
		300	3.92 pJ	$7037 \mu m^2$	400	3.85 pJ	$6794 \mu m^2$
		500	3.87 pJ	$7171 \mu m^2$	600	4.04 pJ	$7075 \mu m^2$
		700	5.52 pJ	$8050 \mu m^2$	800	7.93 pJ	$11395 \mu m^2$
		900	13.32 pJ	$15535 \mu m^2$			
16	2	100	18.98 pJ	$76823 \mu m^2$	200	18.21 pJ	$77602 \mu m^2$
		300	18.13 pJ	$74949 \mu m^2$	400	18.9 pJ	$76635 \mu m^2$
		500	19.08 pJ	$79208 \mu m^2$			
32	1	100	14.93 pJ	$28359 \mu m^2$	200	14.54 pJ	$28984 \mu m^2$
		300	14.15 pJ	$28347 \mu m^2$	400	13.31 pJ	$27199 \mu m^2$

Table A.27: Benchmark results for Half-Width Single-Cycle Multiplier With Adder Tree (Section 7.3.1).

$w$	$n$	pipeline depth	pipeline delay	interconnect	FUs	delays
8	1	1	1	24	1	0
8	2	3	1	96	5	3
8	4	4	1	390	22	8
16	1	1	1	48	1	0
16	2	3	1	192	5	3
32	1	1	1	96	1	0

Table A.28: Benchmark results for Half-Width Single-Cycle Multiplier With Adder Tree (Section 7.3.1).

$w$	$n$	freq.	energy/op	area	freq.	energy/op	area
8	1	100	1.14 pJ	$1909 \mu m^2$	200	1.1 pJ	$1873 \mu m^2$
		300	1.11 pJ	$1821 \mu m^2$	400	1.11 pJ	$1757 \mu m^2$
		500	1.14 pJ	$1761 \mu m^2$	600	1.15 pJ	$1776 \mu m^2$
		700	1.37 pJ	$2156 \mu m^2$	800	1.48 pJ	$2341 \mu m^2$
		900	1.64 pJ	$2557 \mu m^2$	1000	1.85 pJ	$2819 \mu m^2$
		1100	2.49 pJ	$3573 \mu m^2$	1200	2.91 pJ	$4010 \mu m^2$
		1300	3.91 pJ	$4565 \mu m^2$			
8	2	100	3.24 pJ	$10052 \mu m^2$	200	3.22 pJ	$9852 \mu m^2$
		300	3.21 pJ	$9942 \mu m^2$	400	3.21 pJ	$9674 \mu m^2$
		500	3.27 pJ	$10151 \mu m^2$	600	3.31 pJ	$9934 \mu m^2$
		700	3.5 pJ	$11603 \mu m^2$	800	3.84 pJ	$12301 \mu m^2$
		900	4.02 pJ	$13155 \mu m^2$	1000	4.59 pJ	$15351 \mu m^2$
		1100	6.12 pJ	$20639 \mu m^2$	1200	7.35 pJ	$21642 \mu m^2$
8	4	100	10.65 pJ	$47148 \mu m^2$	200	10.57 pJ	$46639 \mu m^2$
		300	10.76 pJ	$48293 \mu m^2$	400	10.9 pJ	$46385 \mu m^2$
		500	11.36 pJ	$52458 \mu m^2$	600	11.35 pJ	$54507 \mu m^2$
		700	11.49 pJ	$56437 \mu m^2$	800	12.32 pJ	$62926 \mu m^2$
		900	13.57 pJ	$69995 \mu m^2$	1000	16.35 pJ	$74225 \mu m^2$
16	1	100	4.39 pJ	$6823 \mu m^2$	200	3.95 pJ	$6869 \mu m^2$
		300	3.92 pJ	$7037 \mu m^2$	400	3.85 pJ	$6794 \mu m^2$
		500	3.87 pJ	$7171 \mu m^2$	600	4.04 pJ	$7075 \mu m^2$
		700	5.52 pJ	$8050 \mu m^2$	800	7.93 pJ	$11395 \mu m^2$
		900	13.32 pJ	$15535 \mu m^2$			
16	2	100	11.04 pJ	$36998 \mu m^2$	200	10.62 pJ	$38190 \mu m^2$
		300	10.63 pJ	$38126 \mu m^2$	400	10.19 pJ	$36987 \mu m^2$
		500	10.34 pJ	$37135 \mu m^2$	600	10.88 pJ	$38585 \mu m^2$
		700	13.41 pJ	$43154 \mu m^2$	800	19.42 pJ	$60713 \mu m^2$
		900	28.99 pJ	$74709 \mu m^2$			
32	1	100	14.93 pJ	$28359 \mu m^2$	200	14.54 pJ	$28984 \mu m^2$
		300	14.15 pJ	$28347 \mu m^2$	400	13.31 pJ	$27199 \mu m^2$

### A.3.2 Dual-Cycle Multiplier With Adder Tree

Table A.29: Benchmark results for Full-Width Dual-Cycle Multiplier With Adder Tree (Section 7.3.2).

$w$	$n$	pipeline depth	pipeline delay	interconnect	FUs	delays
8	1	2	2	32	1	1
8	2	4	2	180	7	5
8	4	5	2	830	34	9

*Continued on next page*

Table A.29: Benchmark results for Full-Width Dual-Cycle Multiplier With Adder Tree (Section 7.3.2).

*Continued from previous page*

$w$	$n$	pipeline depth	pipeline delay	interconnect	FUs	delays
16	1	2	2	64	1	1
16	2	4	2	356	7	5
32	1	2	2	128	1	1

Table A.30: Benchmark results for Full-Width Dual-Cycle Multiplier With Adder Tree (Section 7.3.2).

$w$	$n$	freq.	energy/op	area	freq.	energy/op	area
8	1	100	2.23 pJ	$2125 \mu m^2$	200	2.18 pJ	$2065 \mu m^2$
		300	2.17 pJ	$2070 \mu m^2$	400	2.14 pJ	$2076 \mu m^2$
		500	2.15 pJ	$2031 \mu m^2$	600	2.21 pJ	$2015 \mu m^2$
		700	2.19 pJ	$2060 \mu m^2$	800	2.22 pJ	$2019 \mu m^2$
		900	2.36 pJ	$2212 \mu m^2$	1000	2.54 pJ	$2370 \mu m^2$
		1100	2.69 pJ	$2599 \mu m^2$	1200	3.24 pJ	$3071 \mu m^2$
		1300	3.85 pJ	$3550 \mu m^2$			
8	2	100	11.33 pJ	$15038 \mu m^2$	200	10.87 pJ	$15245 \mu m^2$
		300	10.89 pJ	$15295 \mu m^2$	400	11.05 pJ	$15267 \mu m^2$
		500	11.27 pJ	$16060 \mu m^2$	600	11.22 pJ	$16616 \mu m^2$
		700	11.28 pJ	$16569 \mu m^2$	800	11.47 pJ	$17470 \mu m^2$
		900	12.14 pJ	$17808 \mu m^2$	1000	13.07 pJ	$20081 \mu m^2$
8	4	100	46.46 pJ	$77821 \mu m^2$	200	45.99 pJ	$78836 \mu m^2$
		300	47.05 pJ	$80884 \mu m^2$	400	45.85 pJ	$85844 \mu m^2$
		500	46.96 pJ	$88039 \mu m^2$	600	46.89 pJ	$88012 \mu m^2$
		700	48.43 pJ	$92692 \mu m^2$			
16	1	100	6.96 pJ	$7340 \mu m^2$	200	6.84 pJ	$7318 \mu m^2$
		300	6.97 pJ	$7400 \mu m^2$	400	6.86 pJ	$7523 \mu m^2$
		500	6.87 pJ	$7248 \mu m^2$	600	6.62 pJ	$7260 \mu m^2$
		700	6.7 pJ	$7351 \mu m^2$	800	7.58 pJ	$7918 \mu m^2$
		900	9.69 pJ	$10526 \mu m^2$	1000	14.37 pJ	$14241 \mu m^2$
16	2	100	31.97 pJ	$54693 \mu m^2$	200	31.56 pJ	$53169 \mu m^2$
		300	32.39 pJ	$54622 \mu m^2$	400	30.06 pJ	$55160 \mu m^2$
		500	31.25 pJ	$54803 \mu m^2$	600	30.88 pJ	$54006 \mu m^2$
		700	33.95 pJ	$59160 \mu m^2$			
32	1	100	22.91 pJ	$28665 \mu m^2$	200	22.28 pJ	$28343 \mu m^2$
		300	23.16 pJ	$28634 \mu m^2$	400	22.66 pJ	$30447 \mu m^2$
		500	23.55 pJ	$31263 \mu m^2$	600	26.61 pJ	$34853 \mu m^2$

Table A.31: Benchmark results for Half-Width Dual-Cycle Multiplier With Adder Tree (Section 7.3.2).

$w$	$n$	pipeline depth	pipeline delay	interconnect	FUs	delays
8	1	2	2	24	1	1
8	2	3	2	96	4	1
8	4	5	2	390	16	6
16	1	2	2	48	1	1
16	2	3	2	192	4	1
32	1	2	2	96	1	1

Table A.32: Benchmark results for Half-Width Dual-Cycle Multiplier With Adder Tree (Section 7.3.2).

$w$	$n$	freq.	energy/op	area	freq.	energy/op	area
8	1	100	2.23 pJ	$2125 \mu m^2$	200	2.18 pJ	$2065 \mu m^2$
		300	2.17 pJ	$2070 \mu m^2$	400	2.14 pJ	$2076 \mu m^2$
		500	2.15 pJ	$2031 \mu m^2$	600	2.21 pJ	$2015 \mu m^2$
		700	2.19 pJ	$2060 \mu m^2$	800	2.22 pJ	$2019 \mu m^2$
		900	2.36 pJ	$2212 \mu m^2$	1000	2.54 pJ	$2370 \mu m^2$
		1100	2.69 pJ	$2599 \mu m^2$	1200	3.24 pJ	$3071 \mu m^2$
		1300	3.85 pJ	$3550 \mu m^2$			
8	2	100	6.38 pJ	$8559 \mu m^2$	200	6.3 pJ	$8472 \mu m^2$
		300	6.27 pJ	$8301 \mu m^2$	400	6.26 pJ	$8323 \mu m^2$
		500	6.26 pJ	$8244 \mu m^2$	600	6.31 pJ	$8409 \mu m^2$
		700	6.21 pJ	$8346 \mu m^2$	800	6.37 pJ	$8646 \mu m^2$
		900	6.62 pJ	$9061 \mu m^2$	1000	7.02 pJ	$9987 \mu m^2$
		1100	7.95 pJ	$11488 \mu m^2$	1200	9.11 pJ	$13242 \mu m^2$
		1300	11.21 pJ	$15019 \mu m^2$			
8	4	100	22.5 pJ	$35884 \mu m^2$	200	21.72 pJ	$35376 \mu m^2$
		300	21.39 pJ	$34929 \mu m^2$	400	21.49 pJ	$35790 \mu m^2$
		500	22.13 pJ	$36280 \mu m^2$	600	22.46 pJ	$37199 \mu m^2$
		700	22.06 pJ	$37719 \mu m^2$	800	22.56 pJ	$38857 \mu m^2$
		900	23.85 pJ	$41856 \mu m^2$	1000	25.76 pJ	$43621 \mu m^2$
16	1	100	6.96 pJ	$7340 \mu m^2$	200	6.84 pJ	$7318 \mu m^2$
		300	6.97 pJ	$7400 \mu m^2$	400	6.86 pJ	$7523 \mu m^2$
		500	6.87 pJ	$7248 \mu m^2$	600	6.62 pJ	$7260 \mu m^2$
		700	6.7 pJ	$7351 \mu m^2$	800	7.58 pJ	$7918 \mu m^2$
		900	9.69 pJ	$10526 \mu m^2$	1000	14.37 pJ	$14241 \mu m^2$
16	2	100	19.78 pJ	$30660 \mu m^2$	200	19.36 pJ	$30392 \mu m^2$
		300	19.87 pJ	$30220 \mu m^2$	400	19.54 pJ	$30526 \mu m^2$
		500	19.52 pJ	$30066 \mu m^2$	600	18.66 pJ	$29359 \mu m^2$
		700	18.99 pJ	$29259 \mu m^2$	800	20.38 pJ	$32711 \mu m^2$
		900	28.11 pJ	$43551 \mu m^2$	1000	37.92 pJ	$53978 \mu m^2$
32	1	100	22.91 pJ	$28665 \mu m^2$	200	22.28 pJ	$28343 \mu m^2$
		300	23.16 pJ	$28634 \mu m^2$	400	22.66 pJ	$30447 \mu m^2$

*Continued on next page*

Table A.32: Benchmark results for Half-Width Dual-Cycle Multiplier With Adder Tree (Section 7.3.2).

*Continued from previous page*

$w$	$n$	freq.	energy/op	area	freq.	energy/op	area
		500	23.55 pJ	$31263 \mu m^2$	600	26.61 pJ	$34853 \mu m^2$

### A.3.3 Single-Cycle Multiplier With Accumulator

Table A.33: Benchmark results for Full-Width Single-Cycle Multiplier With Accumulator (Section 7.3.3).

$w$	$n$	pipeline depth	pipeline delay	interconnect	FUs	delays
8	1	3	2	40	2	1
8	2	5	4	160	5	1
8	4	9	8	616	10	1
16	1	3	2	80	2	1
16	2	5	4	312	5	1
32	1	3	2	160	2	1

Table A.34: Benchmark results for Full-Width Single-Cycle Multiplier With Accumulator (Section 7.3.3).

$w$	$n$	freq.	energy/op	area	freq.	energy/op	area
8	1	100	2.85 pJ	$3731 \mu m^2$	200	2.67 pJ	$3765 \mu m^2$
		300	2.75 pJ	$3721 \mu m^2$	400	2.68 pJ	$3644 \mu m^2$
		500	2.71 pJ	$3705 \mu m^2$	600	2.72 pJ	$3694 \mu m^2$
		700	3.16 pJ	$4245 \mu m^2$	800	3.23 pJ	$5120 \mu m^2$
		900	3.39 pJ	$5141 \mu m^2$	1000	4.05 pJ	$5632 \mu m^2$
		1100	4.83 pJ	$7063 \mu m^2$	1200	5.64 pJ	$8859 \mu m^2$
		1300	6.35 pJ	$9161 \mu m^2$			
8	2	100	10.13 pJ	$9807 \mu m^2$	200	9.88 pJ	$9849 \mu m^2$
		300	10.18 pJ	$9997 \mu m^2$	400	9.88 pJ	$9957 \mu m^2$
		500	10.44 pJ	$11084 \mu m^2$	600	10.49 pJ	$11199 \mu m^2$
		700	10.98 pJ	$12178 \mu m^2$	800	11.26 pJ	$13188 \mu m^2$
		900	12.28 pJ	$15102 \mu m^2$	1000	14.05 pJ	$17255 \mu m^2$
		1100	16.87 pJ	$21602 \mu m^2$			
8	4	100	32.38 pJ	$20422 \mu m^2$	200	32.74 pJ	$20989 \mu m^2$
		300	32.54 pJ	$23346 \mu m^2$	400	32.6 pJ	$23201 \mu m^2$
		500	33.11 pJ	$24008 \mu m^2$	600	33.46 pJ	$24624 \mu m^2$
		700	34.17 pJ	$26952 \mu m^2$			

*Continued on next page*

Table A.34: Benchmark results for Full-Width Single-Cycle Multiplier With Accumulator (Section 7.3.3).

*Continued from previous page*

$w$	$n$	freq.	energy/op	area	freq.	energy/op	area
16	1	100	8.49 pJ	$14637 \mu m^2$	200	8.04 pJ	$14689 \mu m^2$
		300	8.08 pJ	$14267 \mu m^2$	400	8.14 pJ	$14764 \mu m^2$
		500	8.28 pJ	$14189 \mu m^2$	600	8.22 pJ	$14451 \mu m^2$
		700	9.74 pJ	$17499 \mu m^2$	900	17.32 pJ	$30622 \mu m^2$
16	2	100	28.58 pJ	$37125 \mu m^2$	200	27.28 pJ	$37926 \mu m^2$
		300	27.41 pJ	$36734 \mu m^2$	400	27.96 pJ	$37224 \mu m^2$
		500	27.84 pJ	$38432 \mu m^2$	600	28.45 pJ	$40871 \mu m^2$
		700	33.37 pJ	$50847 \mu m^2$			
32	1	100	25.49 pJ	$62248 \mu m^2$	200	26.74 pJ	$57556 \mu m^2$
		300	25.11 pJ	$57841 \mu m^2$	400	25.32 pJ	$59369 \mu m^2$
		500	25.02 pJ	$63286 \mu m^2$	600	41.94 pJ	$106194 \mu m^2$

Table A.35: Benchmark results for Half-Width Single-Cycle Multiplier With Accumulator (Section 7.3.3).

$w$	$n$	pipeline depth	pipeline delay	interconnect	FUs	delays
8	1	1	1	24	1	0
8	2	5	4	88	2	1
8	4	9	8	328	5	1
16	1	1	1	48	1	0
16	2	5	4	176	2	1
32	1	1	1	96	1	0

Table A.36: Benchmark results for Half-Width Single-Cycle Multiplier With Accumulator (Section 7.3.3).

$w$	$n$	freq.	energy/op	area	freq.	energy/op	area
8	1	100	1.14 pJ	$1909 \mu m^2$	200	1.1 pJ	$1873 \mu m^2$
		300	1.11 pJ	$1821 \mu m^2$	400	1.11 pJ	$1757 \mu m^2$
		500	1.14 pJ	$1761 \mu m^2$	600	1.15 pJ	$1776 \mu m^2$
		700	1.37 pJ	$2156 \mu m^2$	800	1.48 pJ	$2341 \mu m^2$
		900	1.64 pJ	$2557 \mu m^2$	1000	1.85 pJ	$2819 \mu m^2$
		1100	2.49 pJ	$3573 \mu m^2$	1200	2.91 pJ	$4010 \mu m^2$
		1300	3.91 pJ	$4565 \mu m^2$			
8	2	100	4.76 pJ	$3731 \mu m^2$	200	4.68 pJ	$3765 \mu m^2$
		300	4.76 pJ	$3721 \mu m^2$	400	4.7 pJ	$3644 \mu m^2$
		500	4.72 pJ	$3705 \mu m^2$	600	4.68 pJ	$3694 \mu m^2$

*Continued on next page*

Table A.36: Benchmark results for Half-Width Single-Cycle Multiplier With Accumulator (Section 7.3.3).

*Continued from previous page*

$w$	$n$	freq.	energy/op	area	freq.	energy/op	area
8	4	700	5.11 pJ	$4245 \mu m^2$	800	5.38 pJ	$5120 \mu m^2$
		900	5.57 pJ	$5141 \mu m^2$	1000	6.46 pJ	$5632 \mu m^2$
		1100	7.89 pJ	$7063 \mu m^2$	1200	9.3 pJ	$8859 \mu m^2$
		1300	10.36 pJ	$9161 \mu m^2$			
		100	16.2 pJ	$9807 \mu m^2$	200	16.05 pJ	$9849 \mu m^2$
		300	16.23 pJ	$9997 \mu m^2$	400	15.98 pJ	$9957 \mu m^2$
16	1	500	16.14 pJ	$11084 \mu m^2$	600	16.2 pJ	$11199 \mu m^2$
		700	16.68 pJ	$12178 \mu m^2$	800	17.17 pJ	$13188 \mu m^2$
		900	18.33 pJ	$15102 \mu m^2$	1000	20.72 pJ	$17255 \mu m^2$
		1100	25.16 pJ	$21602 \mu m^2$			
		100	4.39 pJ	$6823 \mu m^2$	200	3.95 pJ	$6869 \mu m^2$
16	2	300	3.92 pJ	$7037 \mu m^2$	400	3.85 pJ	$6794 \mu m^2$
		500	3.87 pJ	$7171 \mu m^2$	600	4.04 pJ	$7075 \mu m^2$
		700	5.52 pJ	$8050 \mu m^2$	800	7.93 pJ	$11395 \mu m^2$
		900	13.32 pJ	$15535 \mu m^2$			
		100	14.84 pJ	$14637 \mu m^2$	200	14.12 pJ	$14689 \mu m^2$
32	1	300	14.22 pJ	$14267 \mu m^2$	400	13.99 pJ	$14764 \mu m^2$
		500	14.07 pJ	$14189 \mu m^2$	600	14.09 pJ	$14451 \mu m^2$
		700	16.9 pJ	$17499 \mu m^2$	900	30.67 pJ	$30622 \mu m^2$
32	2	100	14.93 pJ	$28359 \mu m^2$	200	14.54 pJ	$28984 \mu m^2$
		300	14.15 pJ	$28347 \mu m^2$	400	13.31 pJ	$27199 \mu m^2$

### A.3.4 Dual-Cycle Multiplier With Accumulator

Table A.37: Benchmark results for Full-Width Dual-Cycle Multiplier With Accumulator (Section 7.3.4).

$w$	$n$	pipeline depth	pipeline delay	interconnect	FUs	delays
8	1	4	3	40	2	1
8	2	6	5	162	5	1
8	4	10	9	622	11	1
16	1	4	3	80	2	1
16	2	6	5	314	5	1
32	1	4	3	160	2	1

Table A.38: Benchmark results for Full-Width Dual-Cycle Multiplier With Accumulator (Section 7.3.4).

$w$	$n$	freq.	energy/op	area	freq.	energy/op	area
8	1	100	3.81 pJ	$3981 \mu m^2$	200	3.75 pJ	$3992 \mu m^2$
		300	3.73 pJ	$3925 \mu m^2$	400	3.71 pJ	$4199 \mu m^2$
		500	3.73 pJ	$3960 \mu m^2$	600	3.8 pJ	$3940 \mu m^2$
		700	3.77 pJ	$3931 \mu m^2$	800	3.8 pJ	$4062 \mu m^2$
		900	3.92 pJ	$4259 \mu m^2$	1000	4.05 pJ	$5015 \mu m^2$
		1100	4.31 pJ	$5272 \mu m^2$	1200	4.9 pJ	$7149 \mu m^2$
		1300	5.46 pJ	$7235 \mu m^2$			
8	2	100	14.67 pJ	$10428 \mu m^2$	200	14.43 pJ	$10393 \mu m^2$
		300	14.39 pJ	$10592 \mu m^2$	400	14.5 pJ	$10665 \mu m^2$
		500	14.69 pJ	$10936 \mu m^2$	600	14.62 pJ	$11095 \mu m^2$
		700	14.49 pJ	$11592 \mu m^2$	800	14.75 pJ	$12208 \mu m^2$
		900	15.28 pJ	$12930 \mu m^2$	1000	16.05 pJ	$14195 \mu m^2$
8	4	100	52.29 pJ	$23779 \mu m^2$	200	51.8 pJ	$24082 \mu m^2$
		300	51.7 pJ	$24214 \mu m^2$	400	51.67 pJ	$25703 \mu m^2$
		500	52.68 pJ	$25899 \mu m^2$	600	52.16 pJ	$26777 \mu m^2$
		700	52.58 pJ	$29054 \mu m^2$			
16	1	100	10.25 pJ	$15911 \mu m^2$	200	10.06 pJ	$15666 \mu m^2$
		300	10.23 pJ	$15977 \mu m^2$	400	10.09 pJ	$16042 \mu m^2$
		500	10.19 pJ	$14206 \mu m^2$	600	9.89 pJ	$14211 \mu m^2$
		700	9.93 pJ	$14127 \mu m^2$	800	10.86 pJ	$15521 \mu m^2$
		900	13.45 pJ	$21953 \mu m^2$	1000	17.99 pJ	$26721 \mu m^2$
16	2	100	39.5 pJ	$39748 \mu m^2$	200	38.7 pJ	$37130 \mu m^2$
		300	39.55 pJ	$37354 \mu m^2$	400	36.75 pJ	$37444 \mu m^2$
		500	38.37 pJ	$38935 \mu m^2$			
32	1	100	29.36 pJ	$54965 \mu m^2$	200	29.63 pJ	$59635 \mu m^2$
		300	30.43 pJ	$55468 \mu m^2$	400	29.68 pJ	$56643 \mu m^2$
		500	29.94 pJ	$58692 \mu m^2$	600	33.2 pJ	$68256 \mu m^2$
		700	55.86 pJ	$104335 \mu m^2$			

Table A.39: Benchmark results for Half-Width Dual-Cycle Multiplier With Accumulator (Section 7.3.4).

$w$	$n$	pipeline depth	pipeline delay	interconnect	FUs	delays
8	1	3	2	24	1	1
8	2	6	5	88	3	1
8	4	10	9	330	6	1
16	1	3	2	48	1	1
16	2	6	5	176	3	1
32	1	3	2	96	1	1

Table A.40: Benchmark results for Half-Width Dual-Cycle Multiplier With Accumulator (Section 7.3.4).

$w$	$n$	freq.	energy/op	area	freq.	energy/op	area
8	1	100	2.79 pJ	$2125 \mu m^2$	200	2.72 pJ	$2065 \mu m^2$
		300	2.73 pJ	$2070 \mu m^2$	400	2.71 pJ	$2076 \mu m^2$
		500	2.72 pJ	$2031 \mu m^2$	600	2.79 pJ	$2015 \mu m^2$
		700	2.77 pJ	$2060 \mu m^2$	800	2.82 pJ	$2019 \mu m^2$
		900	3.03 pJ	$2212 \mu m^2$	1000	3.25 pJ	$2370 \mu m^2$
		1100	3.47 pJ	$2599 \mu m^2$	1200	4.13 pJ	$3071 \mu m^2$
		1300	5 pJ	$3550 \mu m^2$			
8	2	100	8.71 pJ	$6196 \mu m^2$	200	8.59 pJ	$6084 \mu m^2$
		300	8.54 pJ	$6102 \mu m^2$	400	8.53 pJ	$6058 \mu m^2$
		500	8.53 pJ	$6184 \mu m^2$	600	8.58 pJ	$6093 \mu m^2$
		700	8.5 pJ	$6079 \mu m^2$	800	8.61 pJ	$6277 \mu m^2$
		900	8.81 pJ	$6812 \mu m^2$	1000	9.2 pJ	$7527 \mu m^2$
		1100	9.74 pJ	$8361 \mu m^2$	1200	11.04 pJ	$9783 \mu m^2$
8	4	100	28.18 pJ	$12658 \mu m^2$	200	27.66 pJ	$12723 \mu m^2$
		300	27.39 pJ	$12802 \mu m^2$	400	27.41 pJ	$12900 \mu m^2$
		500	27.78 pJ	$12890 \mu m^2$	600	28.06 pJ	$13428 \mu m^2$
		700	27.64 pJ	$13625 \mu m^2$	800	28.22 pJ	$14378 \mu m^2$
		900	29.15 pJ	$15469 \mu m^2$	1000	31.06 pJ	$16712 \mu m^2$
16	1	100	9.29 pJ	$7340 \mu m^2$	200	9.18 pJ	$7318 \mu m^2$
		300	9.6 pJ	$7400 \mu m^2$	400	9.36 pJ	$7523 \mu m^2$
		500	9.33 pJ	$7248 \mu m^2$	600	9.13 pJ	$7260 \mu m^2$
		700	9.19 pJ	$7351 \mu m^2$	800	10.13 pJ	$7918 \mu m^2$
		900	13.18 pJ	$10526 \mu m^2$	1000	20.02 pJ	$14241 \mu m^2$
16	2	100	24.44 pJ	$22389 \mu m^2$	200	23.82 pJ	$22347 \mu m^2$
		300	24.33 pJ	$22729 \mu m^2$	400	24 pJ	$22673 \mu m^2$
		500	24.01 pJ	$22298 \mu m^2$	600	23.29 pJ	$22070 \mu m^2$
		700	23.64 pJ	$22095 \mu m^2$	800	24.88 pJ	$24228 \mu m^2$
		900	32.71 pJ	$33022 \mu m^2$	1000	43.18 pJ	$40394 \mu m^2$
32	1	100	32.48 pJ	$28665 \mu m^2$	200	32.59 pJ	$28343 \mu m^2$
		300	33.43 pJ	$28634 \mu m^2$	400	32.72 pJ	$30447 \mu m^2$
		500	33.52 pJ	$31263 \mu m^2$	600	36.91 pJ	$34853 \mu m^2$

## A.4 Multiply-Accumulation

Table A.41: Benchmark results for Full-Width Multiply-Accumulation Baseline — 100 steps (Chapter 8).

$w$	$n$	pipeline depth	pipeline delay	interconnect	FUs	delays
8	1	101	101	1616	1	0
16	1	101	101	3232	1	0

*Continued on next page*

Table A.41: Benchmark results for Full-Width Multiply-Accumulation Baseline — 100 steps (Chapter 8).

*Continued from previous page*

$w$	$n$	pipeline depth	pipeline delay	interconnect	FUs	delays
32	1	101	101	6464	1	0

Table A.42: Benchmark results for Full-Width Multiply-Accumulation Baseline — 100 steps (Chapter 8).

$w$	$n$	freq.	energy/op	area	freq.	energy/op	area
8	1	100	2.11 pJ	1703 $\mu m^2$	200	1.99 pJ	1707 $\mu m^2$
		300	2.09 pJ	1791 $\mu m^2$	400	1.98 pJ	1663 $\mu m^2$
		500	2.08 pJ	1763 $\mu m^2$	600	2.59 pJ	2018 $\mu m^2$
		700	2.44 pJ	2059 $\mu m^2$	800	2.77 pJ	2296 $\mu m^2$
		900	3.16 pJ	2446 $\mu m^2$			
16	1	100	7.08 pJ	5955 $\mu m^2$	200	7.36 pJ	5955 $\mu m^2$
		300	7.39 pJ	6146 $\mu m^2$	400	9.65 pJ	7055 $\mu m^2$
		500	10.69 pJ	7921 $\mu m^2$	600	10.09 pJ	8108 $\mu m^2$
		700	13.96 pJ	10272 $\mu m^2$	800	26.24 pJ	16033 $\mu m^2$
32	1	100	31.34 pJ	20088 $\mu m^2$	200	33.82 pJ	25277 $\mu m^2$
		300	34.06 pJ	28395 $\mu m^2$	400	39.71 pJ	35376 $\mu m^2$
		500	40.05 pJ	40902 $\mu m^2$	600	67.34 pJ	52205 $\mu m^2$

Table A.43: Benchmark results for Half-Width Multiply-Accumulation Baseline — 100 steps (Chapter 8).

$w$	$n$	pipeline depth	pipeline delay	interconnect	FUs	delays
8	1	101	101	1616	1	0
16	1	101	101	3232	1	0
32	1	101	101	6464	1	0

Table A.44: Benchmark results for Half-Width Multiply-Accumulation Baseline — 100 steps (Chapter 8).

$w$	$n$	freq.	energy/op	area	freq.	energy/op	area
8	1	100	0.59 pJ	681 $\mu m^2$	200	0.59 pJ	695 $\mu m^2$
		300	0.59 pJ	712 $\mu m^2$	400	0.59 pJ	715 $\mu m^2$
		500	0.59 pJ	721 $\mu m^2$	600	0.59 pJ	722 $\mu m^2$

*Continued on next page*

Table A.44: Benchmark results for Half-Width Multiply-Accumulation Baseline — 100 steps (Chapter 8).

*Continued from previous page*

$w$	$n$	freq.	energy/op	area	freq.	energy/op	area
16	1	700	0.59 pJ	$726 \mu m^2$	800	0.63 pJ	$723 \mu m^2$
		900	0.76 pJ	$767 \mu m^2$	1000	0.76 pJ	$833 \mu m^2$
		1100	0.83 pJ	$879 \mu m^2$	1200	0.88 pJ	$964 \mu m^2$
		1300	1.04 pJ	$1125 \mu m^2$	1400	1.29 pJ	$1378 \mu m^2$
		1500	1.8 pJ	$1665 \mu m^2$	1600	2.07 pJ	$1945 \mu m^2$
32	1	100	2.47 pJ	$2322 \mu m^2$	200	2.63 pJ	$2388 \mu m^2$
		300	2.55 pJ	$2282 \mu m^2$	400	2.5 pJ	$2210 \mu m^2$
		500	2.44 pJ	$2250 \mu m^2$	600	3.17 pJ	$2520 \mu m^2$
		700	3.59 pJ	$2872 \mu m^2$	800	4.25 pJ	$3314 \mu m^2$
		900	4.85 pJ	$3766 \mu m^2$	1000	6.94 pJ	$5119 \mu m^2$
		1100	11.7 pJ	$6590 \mu m^2$			
32	2	100	9.59 pJ	$9066 \mu m^2$	200	9.25 pJ	$9215 \mu m^2$
		300	9.8 pJ	$9308 \mu m^2$	400	12.82 pJ	$10283 \mu m^2$
		500	13.06 pJ	$10920 \mu m^2$			

#### A.4.1 Single-Cycle MAC With Accumulator

Table A.45: Benchmark results for Full-Width Single-Cycle Multiply-Accumulator With Accumulator — 100 steps (Section 8.3.1).

$w$	$n$	pipeline depth	pipeline delay	interconnect	FUs	delays
8	1	102	101	3317	3	0
8	2	302	301	13735	6	0
8	4	702	701	56171	11	0
16	1	102	101	6533	3	0
16	2	302	301	26567	6	0
32	1	102	101	12965	3	0

Table A.46: Benchmark results for Full-Width Single-Cycle Multiply-Accumulator With Accumulator — 100 steps (Section 8.3.1).

$w$	$n$	freq.	energy/op	area	freq.	energy/op	area
8	1	100	2.26 pJ	$5823 \mu m^2$	200	2.24 pJ	$5863 \mu m^2$
		300	2.25 pJ	$5834 \mu m^2$	400	2.21 pJ	$5715 \mu m^2$

*Continued on next page*

Table A.46: Benchmark results for Full-Width Single-Cycle Multiply-Accumulator With Accumulator — 100 steps (Section 8.3.1).

*Continued from previous page*

$w$	$n$	freq.	energy/op	area	freq.	energy/op	area
		500	2.28 pJ	6041 $\mu m^2$	600	1.99 pJ	6377 $\mu m^2$
		700	2.04 pJ	6987 $\mu m^2$	800	2.19 pJ	7904 $\mu m^2$
		900	2.34 pJ	8563 $\mu m^2$	1000	2.85 pJ	9707 $\mu m^2$
		1100	3.51 pJ	11279 $\mu m^2$			
8	2	100	11.35 pJ	11449 $\mu m^2$	200	11.19 pJ	11541 $\mu m^2$
		300	11.68 pJ	12456 $\mu m^2$	400	10.29 pJ	12845 $\mu m^2$
		500	10.52 pJ	13419 $\mu m^2$	600	10.72 pJ	13418 $\mu m^2$
		700	10.82 pJ	14659 $\mu m^2$	800	11.7 pJ	16491 $\mu m^2$
		900	12.02 pJ	18374 $\mu m^2$			
8	4	100	41.42 pJ	21845 $\mu m^2$	200	41.69 pJ	23007 $\mu m^2$
		300	38.39 pJ	25678 $\mu m^2$	400	38.42 pJ	25496 $\mu m^2$
		500	38.76 pJ	26720 $\mu m^2$	600	39.59 pJ	27631 $\mu m^2$
		700	40.09 pJ	30307 $\mu m^2$			
16	1	100	7.78 pJ	22116 $\mu m^2$	200	7.41 pJ	21677 $\mu m^2$
		300	7.52 pJ	21664 $\mu m^2$	400	7.09 pJ	21131 $\mu m^2$
		500	7.9 pJ	21399 $\mu m^2$	600	7.45 pJ	23474 $\mu m^2$
		700	8.99 pJ	28213 $\mu m^2$	800	12.94 pJ	35159 $\mu m^2$
16	2	100	35.62 pJ	44251 $\mu m^2$	200	33.71 pJ	43573 $\mu m^2$
		300	33.94 pJ	44389 $\mu m^2$	400	34 pJ	45904 $\mu m^2$
		500	33.88 pJ	45841 $\mu m^2$			
32	1	100	29.44 pJ	95268 $\mu m^2$	200	28.94 pJ	90276 $\mu m^2$
		300	27.03 pJ	88144 $\mu m^2$	400	25.29 pJ	95574 $\mu m^2$
		500	25.94 pJ	102471 $\mu m^2$	600	44.24 pJ	136854 $\mu m^2$

Table A.47: Benchmark results for Half-Width Single-Cycle Multiply-Accumulator With Accumulator — 100 steps (Section 8.3.1).

$w$	$n$	pipeline depth	pipeline delay	interconnect	FUs	delays
8	1	102	101	2408	2	0
8	2	302	301	8317	3	0
8	4	702	701	30935	6	0
16	1	102	101	4816	2	0
16	2	302	301	16333	3	0
32	1	102	101	9632	2	0

Table A.48: Benchmark results for Half-Width Single-Cycle Multiply-Accumulator With Accumulator — 100 steps (Section 8.3.1).

$w$	$n$	freq.	energy/op	area	freq.	energy/op	area
8	1	100	1.74 pJ	$3721 \mu m^2$	200	1.71 pJ	$3684 \mu m^2$
		300	1.73 pJ	$3649 \mu m^2$	400	1.7 pJ	$3648 \mu m^2$
		500	1.72 pJ	$3841 \mu m^2$	600	1.73 pJ	$3647 \mu m^2$
		700	1.64 pJ	$4298 \mu m^2$	800	1.83 pJ	$4654 \mu m^2$
		900	1.93 pJ	$4827 \mu m^2$	1000	2.35 pJ	$6032 \mu m^2$
		1100	3.03 pJ	$7055 \mu m^2$	1200	3.88 pJ	$8102 \mu m^2$
8	2	100	6.14 pJ	$5823 \mu m^2$	200	6.11 pJ	$5863 \mu m^2$
		300	6.1 pJ	$5834 \mu m^2$	400	6 pJ	$5715 \mu m^2$
		500	6.12 pJ	$6041 \mu m^2$	600	5.78 pJ	$6377 \mu m^2$
		700	5.8 pJ	$6987 \mu m^2$	800	6.19 pJ	$7904 \mu m^2$
		900	6.59 pJ	$8563 \mu m^2$	1000	8.21 pJ	$9707 \mu m^2$
		1100	9.73 pJ	$11279 \mu m^2$			
8	4	100	22.31 pJ	$11449 \mu m^2$	200	22.2 pJ	$11541 \mu m^2$
		300	22.97 pJ	$12456 \mu m^2$	400	20.33 pJ	$12845 \mu m^2$
		500	20.47 pJ	$13419 \mu m^2$	600	20.73 pJ	$13418 \mu m^2$
		700	20.77 pJ	$14659 \mu m^2$	800	22.26 pJ	$16491 \mu m^2$
		900	22.69 pJ	$18374 \mu m^2$			
16	1	100	6.91 pJ	$14031 \mu m^2$	200	6.51 pJ	$14109 \mu m^2$
		300	6.6 pJ	$14049 \mu m^2$	400	6.43 pJ	$14622 \mu m^2$
		500	6.17 pJ	$14043 \mu m^2$	600	6.54 pJ	$14350 \mu m^2$
		700	8.21 pJ	$16010 \mu m^2$	900	16.86 pJ	$27779 \mu m^2$
16	2	100	21.5 pJ	$22116 \mu m^2$	200	20.36 pJ	$21677 \mu m^2$
		300	20.62 pJ	$21664 \mu m^2$	400	19.4 pJ	$21131 \mu m^2$
		500	21.38 pJ	$21399 \mu m^2$	600	20.3 pJ	$23474 \mu m^2$
		700	24.18 pJ	$28213 \mu m^2$	800	35.68 pJ	$35159 \mu m^2$
32	1	100	27.88 pJ	$62020 \mu m^2$	200	27.31 pJ	$60564 \mu m^2$
		500	24.2 pJ	$69068 \mu m^2$			

#### A.4.2 Dual-Cycle MAC With Accumulator

Table A.49: Benchmark results for Full-Width Dual-Cycle Multiply-Accumulator With Accumulator — 100 steps (Section 8.3.2).

$w$	$n$	pipeline depth	pipeline delay	interconnect	FUs	delays
8	1	202	201	3417	3	0
8	2	402	401	14035	6	0
8	4	802	801	56871	12	0
16	1	202	201	6633	3	0

*Continued on next page*

Table A.49: Benchmark results for Full-Width Dual-Cycle Multiply-Accumulator With Accumulator — 100 steps (Section 8.3.2).

*Continued from previous page*

$w$	$n$	pipeline depth	pipeline delay	interconnect	FUs	delays
16	2	402	401	26867	6	0
32	1	202	201	13065	3	0

Table A.50: Benchmark results for Full-Width Dual-Cycle Multiply-Accumulator With Accumulator — 100 steps (Section 8.3.2).

$w$	$n$	freq.	energy/op	area	freq.	energy/op	area
8	1	100	4.18 pJ	6144 $\mu m^2$	200	4.13 pJ	6221 $\mu m^2$
		300	4.15 pJ	6200 $\mu m^2$	400	4.14 pJ	6131 $\mu m^2$
		500	4.16 pJ	6286 $\mu m^2$	600	4.25 pJ	6770 $\mu m^2$
		700	4.2 pJ	6632 $\mu m^2$	800	4.2 pJ	6974 $\mu m^2$
		900	4.28 pJ	7178 $\mu m^2$	1000	4.51 pJ	7735 $\mu m^2$
		1100	4.67 pJ	9063 $\mu m^2$			
8	2	100	15.37 pJ	12634 $\mu m^2$	200	15.24 pJ	12449 $\mu m^2$
		300	15.23 pJ	12616 $\mu m^2$	400	15.24 pJ	12905 $\mu m^2$
		500	15.48 pJ	13843 $\mu m^2$	600	15.4 pJ	13935 $\mu m^2$
		700	15.11 pJ	14174 $\mu m^2$	800	15.34 pJ	14866 $\mu m^2$
		900	16.21 pJ	16566 $\mu m^2$			
8	4	100	58.96 pJ	25759 $\mu m^2$	200	58.56 pJ	26288 $\mu m^2$
		300	57.83 pJ	28254 $\mu m^2$	400	58.14 pJ	28306 $\mu m^2$
		500	59.14 pJ	28868 $\mu m^2$	600	59.44 pJ	30253 $\mu m^2$
		700	59.38 pJ	32207 $\mu m^2$			
16	1	100	13.04 pJ	22349 $\mu m^2$	200	12.68 pJ	22711 $\mu m^2$
		300	12.97 pJ	22366 $\mu m^2$	400	12.93 pJ	22228 $\mu m^2$
		500	12.63 pJ	23145 $\mu m^2$	600	12.31 pJ	22973 $\mu m^2$
		700	12.47 pJ	23613 $\mu m^2$	800	13.77 pJ	28069 $\mu m^2$
16	2	100	47.58 pJ	47046 $\mu m^2$	200	45.89 pJ	46395 $\mu m^2$
		300	47.17 pJ	47003 $\mu m^2$	400	43.83 pJ	46776 $\mu m^2$
		500	45.65 pJ	46074 $\mu m^2$	600	46.29 pJ	46748 $\mu m^2$
		700	48.25 pJ	57839 $\mu m^2$			
32	1	100	42.01 pJ	86595 $\mu m^2$	200	43.27 pJ	88791 $\mu m^2$
		300	43.02 pJ	89010 $\mu m^2$	400	39.55 pJ	92789 $\mu m^2$
		500	37.69 pJ	99555 $\mu m^2$	600	42.12 pJ	121029 $\mu m^2$

Table A.51: Benchmark results for Half-Width Dual-Cycle Multiply-Accumulator With Accumulator — 100 steps (Section 8.3.2).

$w$	$n$	pipeline depth	pipeline delay	interconnect	FUs	delays
8	1	302	301	2408	2	0
8	2	402	401	8417	4	0
8	4	802	801	31235	7	0
16	1	302	301	4816	2	0
16	2	402	401	16433	4	0
32	1	302	301	9632	2	0

Table A.52: Benchmark results for Half-Width Dual-Cycle Multiply-Accumulator With Accumulator — 100 steps (Section 8.3.2).

$w$	$n$	freq.	energy/op	area	freq.	energy/op	area
8	1	100	3.8 pJ	$3974 \mu m^2$	200	3.73 pJ	$3961 \mu m^2$
		300	3.73 pJ	$3923 \mu m^2$	400	3.73 pJ	$3892 \mu m^2$
		500	3.73 pJ	$3892 \mu m^2$	600	3.77 pJ	$3925 \mu m^2$
		700	3.73 pJ	$3899 \mu m^2$	800	3.77 pJ	$3931 \mu m^2$
		900	3.9 pJ	$4491 \mu m^2$	1000	3.96 pJ	$4607 \mu m^2$
		1100	4.31 pJ	$5746 \mu m^2$	1200	5.03 pJ	$6122 \mu m^2$
		1300	6.1 pJ	$7201 \mu m^2$			
8	2	100	10.05 pJ	$8351 \mu m^2$	200	9.96 pJ	$8296 \mu m^2$
		300	9.87 pJ	$8334 \mu m^2$	400	9.9 pJ	$8255 \mu m^2$
		500	9.91 pJ	$8645 \mu m^2$	600	9.98 pJ	$8722 \mu m^2$
		700	9.76 pJ	$8810 \mu m^2$	800	9.93 pJ	$9182 \mu m^2$
		900	10.15 pJ	$9491 \mu m^2$	1000	10.95 pJ	$10454 \mu m^2$
		1100	11.43 pJ	$11500 \mu m^2$			
8	4	100	33.34 pJ	$14853 \mu m^2$	200	32.98 pJ	$14565 \mu m^2$
		300	32.83 pJ	$14753 \mu m^2$	400	32.88 pJ	$15027 \mu m^2$
		500	33.33 pJ	$15531 \mu m^2$	600	33.06 pJ	$16407 \mu m^2$
		700	32.4 pJ	$16305 \mu m^2$	800	32.93 pJ	$17221 \mu m^2$
		900	34.59 pJ	$18791 \mu m^2$			
16	1	100	12 pJ	$14189 \mu m^2$	200	11.65 pJ	$14115 \mu m^2$
		300	11.91 pJ	$14031 \mu m^2$	400	11.69 pJ	$14214 \mu m^2$
		500	11.75 pJ	$15459 \mu m^2$	600	11.43 pJ	$15279 \mu m^2$
		700	11.27 pJ	$14077 \mu m^2$	800	12.3 pJ	$16799 \mu m^2$
		900	16.21 pJ	$21679 \mu m^2$			
16	2	100	32.23 pJ	$30573 \mu m^2$	200	30.95 pJ	$30264 \mu m^2$
		300	31.74 pJ	$30449 \mu m^2$	400	31.53 pJ	$30420 \mu m^2$
		500	30.47 pJ	$30232 \mu m^2$	600	30.03 pJ	$29115 \mu m^2$
32	1	100	39.07 pJ	$54392 \mu m^2$	200	39.57 pJ	$60264 \mu m^2$
		300	40.7 pJ	$61150 \mu m^2$	400	39.54 pJ	$56315 \mu m^2$
		500	39.33 pJ	$59307 \mu m^2$	600	43.33 pJ	$67518 \mu m^2$

*Continued on next page*

Table A.52: Benchmark results for Half-Width Dual-Cycle Multiply-Accumulator With Accumulator — 100 steps (Section 8.3.2).

*Continued from previous page*

$w$	$n$	freq.	energy/op	area	freq.	energy/op	area
		700	78.06 pJ	98950 $\mu m^2$			

#### A.4.3 Distributed Multiply-Accumulator

Table A.53: Benchmark results for Full-Width Distributed Multiply-Accumulator — 100 steps (Section 8.3.3).

$w$	$n$	pipeline depth	pipeline delay	interconnect	FUs	delays
8	1	103	103	1618	2	1
8	2	108	108	6491	4	0
8	4	116	116	25983	16	0
16	1	103	103	3234	2	1
16	2	108	108	12979	4	0
32	1	103	103	6466	2	1

Table A.54: Benchmark results for Full-Width Distributed Multiply-Accumulator — 100 steps (Section 8.3.3).

$w$	$n$	freq.	energy/op	area	freq.	energy/op	area
8	1	100	2.23 pJ	3267 $\mu m^2$	200	2.24 pJ	3211 $\mu m^2$
		300	2.13 pJ	3075 $\mu m^2$	400	2.1 pJ	3068 $\mu m^2$
		500	2.1 pJ	3160 $\mu m^2$	600	2.12 pJ	3172 $\mu m^2$
		700	2.05 pJ	3281 $\mu m^2$	800	2.01 pJ	3420 $\mu m^2$
		900	2.07 pJ	3674 $\mu m^2$	1000	2.3 pJ	3947 $\mu m^2$
		1100	2.87 pJ	4608 $\mu m^2$	1200	3.12 pJ	5327 $\mu m^2$
		1300	3.8 pJ	5851 $\mu m^2$	1400	4.32 pJ	6270 $\mu m^2$
8	2	100	8.65 pJ	12498 $\mu m^2$	200	8.66 pJ	12596 $\mu m^2$
		300	8.22 pJ	12619 $\mu m^2$	400	8.12 pJ	12581 $\mu m^2$
		500	8.1 pJ	12640 $\mu m^2$	600	8.11 pJ	13062 $\mu m^2$
		700	7.89 pJ	12946 $\mu m^2$	800	7.75 pJ	14088 $\mu m^2$
		900	7.93 pJ	14580 $\mu m^2$	1000	8.76 pJ	15939 $\mu m^2$
		1100	10.96 pJ	19158 $\mu m^2$	1200	12.13 pJ	21076 $\mu m^2$
		1300	15.03 pJ	23315 $\mu m^2$			
8	4	100	34.46 pJ	52052 $\mu m^2$	200	34.52 pJ	53050 $\mu m^2$
		300	32.74 pJ	52033 $\mu m^2$	400	32.33 pJ	52016 $\mu m^2$

*Continued on next page*

Table A.54: Benchmark results for Full-Width Distributed Multiply-Accumulator — 100 steps (Section 8.3.3).

*Continued from previous page*

$w$	$n$	freq.	energy/op	area	freq.	energy/op	area
16	1	500	32.28 pJ	52970 $\mu m^2$	600	32.17 pJ	53361 $\mu m^2$
		700	31.53 pJ	54161 $\mu m^2$	900	31.45 pJ	60612 $\mu m^2$
		1300	60.72 pJ	95776 $\mu m^2$			
16	2	100	7.67 pJ	10104 $\mu m^2$	200	7.69 pJ	10072 $\mu m^2$
		300	7.23 pJ	10056 $\mu m^2$	400	7.37 pJ	10114 $\mu m^2$
		500	7.34 pJ	10174 $\mu m^2$	600	7.03 pJ	10196 $\mu m^2$
		700	6.71 pJ	10463 $\mu m^2$	800	7.86 pJ	11586 $\mu m^2$
		900	10.82 pJ	14727 $\mu m^2$	1000	15.78 pJ	18470 $\mu m^2$
		1100	22 pJ	21620 $\mu m^2$			
32	1	100	30.11 pJ	40962 $\mu m^2$	200	30.12 pJ	40929 $\mu m^2$
		300	28.49 pJ	40288 $\mu m^2$	400	28.95 pJ	39771 $\mu m^2$
		500	28.41 pJ	40946 $\mu m^2$	600	27.26 pJ	40755 $\mu m^2$
		700	26.19 pJ	42457 $\mu m^2$	800	30.8 pJ	46331 $\mu m^2$
		900	42.63 pJ	60224 $\mu m^2$	1000	59.37 pJ	74502 $\mu m^2$

Table A.55: Benchmark results for Half-Width Distributed Multiply-Accumulator — 100 steps (Section 8.3.3).

$w$	$n$	pipeline depth	pipeline delay	interconnect	FUs	delays
8	1	102	102	1608	1	0
8	2	104	104	4833	3	0
8	4	108	108	16155	10	0
16	1	102	102	3216	1	0
16	2	104	104	9665	3	0
32	1	102	102	6432	1	0

Table A.56: Benchmark results for Half-Width Distributed Multiply-Accumulator — 100 steps (Section 8.3.3).

$w$	$n$	freq.	energy/op	area	freq.	energy/op	area
8	1	100	2.17 pJ	3081 $\mu m^2$	200	2.17 pJ	3168 $\mu m^2$
		300	2.07 pJ	2980 $\mu m^2$	400	2.04 pJ	3192 $\mu m^2$
		500	2.04 pJ	3216 $\mu m^2$	600	2.06 pJ	3119 $\mu m^2$

*Continued on next page*

Table A.56: Benchmark results for Half-Width Distributed Multiply-Accumulator — 100 steps (Section 8.3.3).

*Continued from previous page*

$w$	$n$	freq.	energy/op	area	freq.	energy/op	area
8	2	700	1.99 pJ	$3146 \mu m^2$	800	1.95 pJ	$3353 \mu m^2$
		900	1.99 pJ	$3511 \mu m^2$	1000	2.17 pJ	$3848 \mu m^2$
		1100	2.69 pJ	$4570 \mu m^2$	1200	2.96 pJ	$5198 \mu m^2$
		1300	3.77 pJ	$5806 \mu m^2$	1400	4.37 pJ	$6272 \mu m^2$
	4	100	6.39 pJ	$9457 \mu m^2$	200	6.4 pJ	$9519 \mu m^2$
		300	6.07 pJ	$9415 \mu m^2$	400	5.99 pJ	$9462 \mu m^2$
		500	5.99 pJ	$9430 \mu m^2$	600	5.97 pJ	$9645 \mu m^2$
16	1	700	5.85 pJ	$9710 \mu m^2$	800	5.7 pJ	$10708 \mu m^2$
		900	5.78 pJ	$10914 \mu m^2$	1000	6.36 pJ	$12055 \mu m^2$
		1100	8.01 pJ	$14122 \mu m^2$	1200	8.99 pJ	$15619 \mu m^2$
		1300	11 pJ	$17373 \mu m^2$			
		100	21.04 pJ	$32448 \mu m^2$	200	21.07 pJ	$32077 \mu m^2$
		300	19.93 pJ	$32115 \mu m^2$	400	19.69 pJ	$32211 \mu m^2$
		500	19.67 pJ	$32974 \mu m^2$	600	19.54 pJ	$33280 \mu m^2$
32	2	700	19.15 pJ	$32723 \mu m^2$	800	18.69 pJ	$36557 \mu m^2$
		900	19.15 pJ	$37000 \mu m^2$	1000	20.89 pJ	$40377 \mu m^2$
		1100	24.77 pJ	$47607 \mu m^2$	1200	29.78 pJ	$53723 \mu m^2$
		1300	38.41 pJ	$58890 \mu m^2$			
		100	7.57 pJ	$9928 \mu m^2$	200	7.58 pJ	$9991 \mu m^2$
64	1	300	7.12 pJ	$9675 \mu m^2$	400	7.26 pJ	$9963 \mu m^2$
		500	7.22 pJ	$9861 \mu m^2$	600	6.91 pJ	$10051 \mu m^2$
		700	6.6 pJ	$10384 \mu m^2$	800	7.69 pJ	$11166 \mu m^2$
		900	10.72 pJ	$14578 \mu m^2$	1000	15.4 pJ	$18385 \mu m^2$
		1100	22.44 pJ	$21793 \mu m^2$			
		100	22.34 pJ	$30813 \mu m^2$	200	22.35 pJ	$30470 \mu m^2$
128	2	300	21.17 pJ	$29706 \mu m^2$	400	21.5 pJ	$30829 \mu m^2$
		500	21.01 pJ	$30459 \mu m^2$	600	20.12 pJ	$29908 \mu m^2$
		700	19.4 pJ	$31350 \mu m^2$	800	22.59 pJ	$34119 \mu m^2$
		900	31.69 pJ	$44774 \mu m^2$	1000	46.11 pJ	$56175 \mu m^2$
		100	29.04 pJ	$37906 \mu m^2$	200	29.89 pJ	$38443 \mu m^2$
256	1	300	29.01 pJ	$38491 \mu m^2$	400	28.66 pJ	$38747 \mu m^2$
		500	26.05 pJ	$39365 \mu m^2$	600	26.15 pJ	$39799 \mu m^2$
		700	52.27 pJ	$58372 \mu m^2$	800	96.25 pJ	$74589 \mu m^2$