Power Efficient Arithmetic Circuits for Application Specific Processors

Georgios Plakaris

Kgs. Lyngby 2003
IMM-THESIS-2003-29

Power Efficient Arithmetic Circuits for Application Specific Processors

Georgios Plakaris

Kgs. Lyngby 2003

Technical University of Denmark
Informatics and Mathematical Modelling
Building 321, DK-2800 Lyngby, Denmark
Phone +45 45253351, Fax +45 45882673
reception@imm.dtu.dk
www.imm.dtu.dk

IMM-THESIS: ISSN 1601-233X

Preface

With this thesis I finalize my Master of Science studies in Computer Systems Engineering at the Technical University of Denmark.

The thesis has been carried out at the division of Computer Science and Engineering at the department of Informatics and Mathematical Modeling at DTU under the supervision of associate professor Jens Sparsø to whom I am grateful for many reasons. To begin with, he was the first Danish person I met at a conference in 1999, where I was working as a secretary. This is how I found out about DTU in the first place. I think, he still finds my coming here an odd decision. However, the whole experience has been very "educational" in every way.

As a supervisor he has always been available, responsive no matter how trivial my questions may have been, and supportive to my efforts. I would also like to thank him for sponsoring my going to the seminar "Design of low-power digital circuits: Techniques and tools" offered at the Technical University of Turin, Italy, as part of the INTRALED program that offers training in Low-Power Design. This seminar has been a dynamic kick-off to my project.

I would also like to thank Tone for her being with me and making my life better. Her support in making this project complete has been invaluable.

Last, I would like to express my gratitude to my parents for everything they have done for me. It has not been easy for them to finance my studies for 8 years now, but they really think they are making a good investment; I guess I should thank them in person.

Lyngby, 31 March, 2003

Georgios Plakaris

Abstract

This thesis presents a study on RT level power optimization techniques in terms of their applicability on data-flow intensive data path designs and their efficiency.

The dynamic power management techniques of clock gating and operand isolation are investigated and their efficiency evaluated by sample designs. Although, clock gating by itself offers significant power savings at low overhead in sequential blocks, it is not always the case that hold conditions can be extracted when input registers are shared among several resources. Latch based operand isolation, was also found quite effective, though savings are offset by the high overhead; evened out in case of the gate-based implementation for 32bit adder/subtractor units.

Fine clock gating is proposed as an approach that merges the merits of both methods and yields the highest power savings and the least performance degradation, for the same overhead.

The static RTL power optimization methods proposed are: power sensitive implementation selection and retiming.

The use of carry-save arithmetic to eliminate carry propagation in datapaths is deployed to improve timing slack and provide larger margins for the performance-power trade-off in other parts of the design.

The proposed methods are escorted by sample design examples to illustrate their efficiency. Further, by closely controlling unnecessary switching activity the overhead of sharing resources among operations of varying complexity is reduced.

The methods proposed are suitable for a synthesis-based design flow and achieve performance comparable to custom application specific processors.

KEYWORDS: Low-power, power efficient arithmetic, operand isolation, dynamic power management.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

For the last three decades, semiconductor industry has been facing a monotonic improvement in technology size, performance and cost, as predicted by G. Moore back in 1965. Ever since, circuits of increasing complexity and performance, though at affordable costs, have been produced. At the very early stages of this phenomenal progress, an increasing gap between technology and design productivity was noticed. This brought about the first Computer Aided Design (CAD) tools that would translate the circuit description, schematic at that time, into the lithographic masks necessary for the production phase. Today tools are taking over the designer as early as at the Register-Transfer (RT) level, while tools for behavioral synthesis have for long been a topic of research.

Along this evolution, the optimization goals have undergone changes. Performance will always be a metric that cannot be neglected. Power dissipation has attained significant importance as it can easily become the bottleneck of current designs, both because of cooling requirements and battery life of portable equipment. It is understood that power minimization will always come with some performance degradation, hence new metrics that capture this trade-off, such as the power-delay product are coming into play. This shift has also resulted in incorporating power awareness both in the CAD tools and in the systems' architectures.

This thesis comes to contribute in the area in between, namely at optimizing power at the block level in a synthesis based design flow, described by code at the RT level. More specifically, the design of power efficient datapath components for use in Application Specific Processors is going to be investigated.

In the remaining of this chapter the motivation for this work is described and the application domain is introduced. Then, power efficiency and optimization is put into perspective throughout the development phase of a product. The organization of the thesis concludes the introduction chapter.

1.1 Motivation and Aim of the Thesis

Design for low power has been the topic of many books. Most of the work refers to the transistor and gate level optimization techniques [15], some on computer-aided low power design [36] and few to system level power management [6]. Little has though been written about low power design at the RT level. In compliance with the general principal that the higher the level of abstraction, the higher the power savings, it is expected that considerable amount of power can be saved at the RT-level before the design is synthesized and gate level optimization algorithms are applied, as described later on.

Nowadays, a diversity of applications calls for high computing performance with very specific functionality. Example applications are image processing and communications (signal processing, compression). The demands of many such applications cannot be dealt with efficiently when using off-the-shelf hardware, making the development of application specific hardware necessary. This is particularly the case for real-time multimedia and signal processing embedded applications e.g. cellular phones, personal digital assistants, gaming applications etc. Application Specific Processors, which are entire processors designed specifically for an application (or application domain), provide for a complete and very efficient solution.

The major obstacles for using ASPs, however, are the required development effort and the related high development costs. Although, tools to streamline the design of ASPs have been announced [34] and are gradually making their way into the standard suites of synthesis tools, designers still have to design power efficient solutions under tight performance and time-to-market requirements. For this reason a synthesis based design flow with rich libraries of components is selected as the implementation platform. It is the purpose of this thesis to explore the design space and low power techniques that could be applied at this level of abstraction using the Design Compiler tool suite and Design-ware components library (both from SYNOPSYS). In this respect, it is believed that the results of this work in the form of simple guidelines for power efficient datapath design could be of great interest to hardware designers.

1.2 Application Specific Processors (ASPs)

The term Application or Domain Specific Processors refers to the midway solution between a custom ASIC and a general purpose computer. It can also be described as a degenerate or specialized Digital Signal Processor, which is a Domain Specific Processor with extended programmability. Figure 1.1 depicts the classification of the above mentioned choices in terms of figures of merit for integrated circuits.

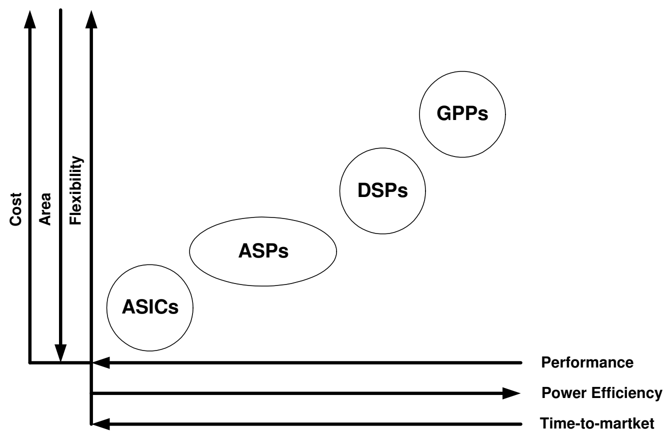


Figure 1.1: Classification of integrated processing solutions

To put it in words, ASPs are both a compromise and a necessity between the expensive, highly utilized, low power and hard to maintain ASICs and the inexpensive, flexible, but lowly utilized and power hungry general purpose computers.

They are meant to process computing intensive problems efficiently, performance- and power-wise. For this reason, they only have a limited, carefully selected instruction set inspired by the specific domain they are intended for and a specialized datapath. Depending on the control context of the domain, control instructions may be included or be mapped to a general purpose “co-processor”; this paves the way to reconfigurable computing where a

GPP is interconnected to a set of satellite co-processors by a network or bus system, as discussed in Paker's Ph.D. thesis [44].

According to Swartzlander [21], three are the main guidelines that should be followed in the design of application specific processors:

- "Use only as much arithmetic as necessary"
- "Use data interconnections that match the algorithm"
- "Use programmability sparingly"

In this work, which is focused on datapath design, the first and the last rules are the ones to be investigated. It is pointed out in the next section that power dissipation is closely related to switching activity. Hence, a fourth command would be to minimize switching activity and a major part of this work is to evaluate how this can be done efficiently in the selected scope.

1.3 Power Saving Techniques in a Top-Down Design Flow

1.3.1 Power Dissipation in Synchronous Digital Circuits

The sources of power dissipation in CMOS technology are summarized in formula 1.1 [20].

$$P = \frac{1}{2} \cdot C \cdot V_{DD}^2 \cdot f \cdot N + Q_S C \cdot V_{DD} \cdot f \cdot N + I_{leak} \cdot V_{DD} \tag{1.1}$$

The first term captures the switching activity power, the power required to charge and discharged the circuit nodes, where C is the node capacitance, V_{DD} is the supply voltage, f is the frequency of operation and N is a factor expressing the node's switching activity, the number of gate output transitions per clock cycle.

The second term represents the short-circuit power, the power dissipated during the gate transitions, when current flows directly from the power supply to the ground terminal through the network of p- and n-type CMOS transistors. The factor Q_{SC} accounts for the quantity of charge carried by the short circuit current per gate transition.

The third term expresses leakage current power, due to the leakage current I_{leak} formed by reverse bias currents at parasitic diodes and subthreshold transistor currents.

Traditionally the last two terms have been disregarded and the switching activity power in a well-designed technology accounted for over 90% of the total power consumption [20]. Recently, the significance of leakage power has been revised and its absolute value is expected to increase, as threshold voltages are lowered to maintain the performance for constantly shrinking supply voltages [13]. In [27], the lower limit of the contribution of leakage power to the total power dissipation for RTL optimized circuits is reported to be 23% for a 0.18μ technology. Throughout this thesis, only switching activity power is taken into account as it is seen as a design problem; leakage power is considered a technology problem and it is left to be faced by technology engineers.

1.3.2 Low Power Design Flow

To avoid costly redesign steps, it is mandatory that power dissipation is considered from the very beginning in the development phase of a design [33]. At each level of abstraction several alternatives need to be considered and their power consumption estimated. For this reason vivid activities in researching power estimation at various levels of abstraction is being done [25], [17], [29], [41]. Figure 1.2 is depicting a possible low-power design flow from the highest (system) down to the lowest (circuit) level with increasing accuracy and decreasing power savings[1]. In the paragraphs to follow, a brief discussion for the different levels of abstraction.

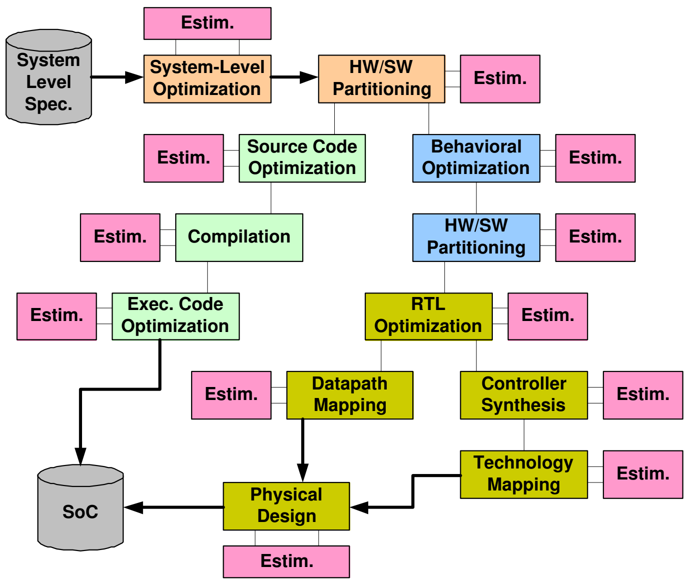


Figure 1.2: Low-Power System Design Flow

System Level

An extensive analysis on system-level power optimization is given by Benini and DeMichelli in [10]. To put things into perspective some general ideas are quoted here. An electronic system consists of a hardware platform and the application software. One level below, a hardware platform consists of three parts: a) computation units, b) communication units and c) storage units, and it is highly important that energy-efficient system level design should address power reduction in all three of them.

A generic hardware platform is illustrated at figure 1.3 comprising several computational units, an extensive memory hierarchy and an interconnection system based either on a bus system or a network. Focusing at the computational units, the implementation strategy may vary from ASICs to general purpose processors, as previously discussed, depending both on programmability, power and cost requirements.

Memory hierarchy can significantly affect power consumption as it is one, if not the biggest power consumer. A typical architecture envisions one or two levels of cashing with design parameters (associativity, word and block length, bandwidth) optimized for the expected workload, usually by use of simulations.

An other important issue is dynamic power management [6], which refers to the availability of different modes of operation of the individual components (sleep, doze, off) and the existence of a controller to control the components' transitions from one mode to the others. Dynamic power management can also be applied at lower levels (block and RT) through clock gating, as it will be discussed in section 2.2.

Another widespread system level power management technique is supply voltage scaling [15]. Although this method has yielded remarkable results, its applicability is limited in the deep-submicron region where supply voltages are as low as 1V.

[1]Colors identify different areas of research

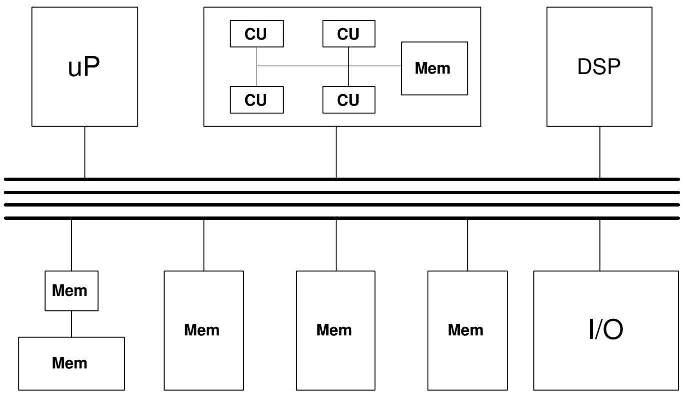


Figure 1.3: A general System-on-Chip Hardware Platform

Behavioral Level

The starting point for this level is a behavioral description of the algorithm captured in an hardware description language (HDL) together with a set of resource, scheduling, timing and, in few cases, power constraints. After this input has been transformed into a Directed Dependence Flow Graph (DDFG), optimization algorithms guided by cost functions are invoked to perform the tasks of scheduling and resource binding [14], [40], [47], [46]. Power awareness has been included in the form of power and effective capacitance models and modified cost functions to take into account total power consumption.

Behavioral synthesis, although promising to mitigate the designers task, is still at its infancy and it will be a while before it will be included in commercial CAD tool suites. In a common flow, the application is mapped on an existing hardware platform and behavioral synthesis is used for parts of the design, while the datapath is hand-crafted. The knowledge though of power and capacitance characteristics of functional units can lead the designer towards wise implementation selections. Part II of the report is inspired by this idea and it is its purpose to provide the designer with this insight. Some of the parameters appearing in the cost functions of behavioral synthesizers can also alert designers to avoid wrong decisions: power can either be reduced by deceasing the effective load capacitance or the switching activity.

Register-Transfer Level

Power optimization at the RT level has lately come into focus for design space exploration and early design validation. Compared to the behavioral and logic level it represents a trade-off between accuracy and computation effort. Power optimization at the RT level is discussed in chapter 2.

Logic Level

Although the amount of power to be saved at this level of abstraction is very small compared to the total power dissipation, because of the detailed and mathematically expressed formulation of the problem with boolean equations, satisfactory results have been achieved. Though, it is important to note that the power savings do not have an additive effect and usually do not exceed the amount of 10-15 percentage units [53]. The input to this optimization level is a nodal connectivity list and nodal switching probabilities. The idea is that nodes with high switching activity should be eliminated by one of the methods stated below.

- Technology Independent Techniques

  - Don't Care Minimization [50]
  - Common Sub-expression Extraction [48]
  - Synthesis of Timed Shannon Circuits [31]
  - State Assignment [7]
  - FSM decomposition [38]
  - Re-timing [37]
  - Guarded Evaluation [54]
- Technology Dependent Techniques
  - Technology Mapping [56]
  - Gate re-sizing [3]
  - Buffer insertion and Pin Swapping [33]
  - Use of Dual Voltage/ Threshold Voltage Gates [57], [27]

1.4 Organization of Chapters

The thesis is organized in three parts. Part one (chapters 2,3 and 4) deals with power saving techniques at the RT level. Part two (chapters 5 and 6) is a study on arithmetic components at the dispense of the RTL designer. The sense of efficiency is broadened to cover the other performance parameters, namely area and timing. Another important aspect is the exploration of what is available to the designer and how it can be used effectively. Finally, part three (chapters 7 and 8) contains the description and the results of a design that elaborates on the findings of the two previous parts. All parts are self containing and include a sample design to prove the points stated. The designs have been carefully selected to be both representative of the domain that is investigated and also manageable in complexity to allow for the extraction of solid conclusions.

Chapter 2 constitutes a literature study of the available power saving techniques at the RT level. The presented techniques are evaluated in terms of applicability, incurred overhead and relevancy to the field in question.

Chapter 3 elaborates on clock gating and operand isolation, the two most prominent techniques, through an examle design. A Complex Arithmetic Unit (CAU) that operates on complex fractional numbers is implemented and optimized for power through dynamic power management techniques.

Chapter 4 specializes on efficient operand isolation. Three alternative methods of operand isolation are suggested and evaluated after being applied on the CAU design from before.

Chapter 5 reviews arithmetic components. The topic is approached from the performance and implementation point of view both theoretically and practically in respect to the options available to the front-end designer and how they can be accessed.

Chapter 6 uses a Multiply-Accumulate unit (MAC) as a subject to evaluate the validity of the findings described in the previous chapter.

Chapter 7 describes the design of a more complicated design. It is characteristic of ASPs to include a separate Multiply Unit in parallel to the ALU. Such a unit that operates on multiple datatypes with varying length (16-, 32-bit) is implemented and optimized through the methods presented in the previous parts.

Chapter 8 concludes by summarizing the findings and extending them to relevant areas of research. Finally some guidelines in the form of “rules of the thumb” are extracted.

Part I

Low-Power Design at the RT Level

Chapter 2

Power Reduction in RT-Level

The RT level of abstraction has been created as an intermediate level between the logic and the architecture levels to facilitate manageability of large designs. It alleviates designers from the tedious and error prone tasks of capturing functionality at the gate level, resulting in considerable improvement in the productivity-design quality product.

In contrast to the system/architecture level of abstraction characterized by general specifications and inaccurate power consumption models for design components, the RT level contains enough implementation details to be used for constrained design space exploration and more precise power estimation. The local optimization techniques applicable at the logic level as mentioned in the previous chapter, not only limit the expected gain, but also incur very high computation requirements. This is due to the incremental nature of the algorithms used and the propagation of the effect of a local change and re-evaluation of the overall power consumption in the design [6]. In conclusion, the coarser RTL description of a design provides for efficient power optimization and estimation algorithms.

The purpose of this chapter is to provide an overview of the available RTL power optimization techniques and evaluate them in the context of this work. Before that, a decomposition of the RT level is performed and a theoretical background is built to enhance comprehensiveness of the proposed techniques.

2.1 The RT Abstraction Level

2.1.1 Decomposition of an RTL Design

At a first level, an RTL design can be decomposed into a control and a datapath unit. The control unit is usually captured as a finite state machine (FSM). A datapath comprises three distinct categories of components:

- Functional Units (e.g. adders, multipliers)
- Steering Logic (multiplexors, tristate buffers and registers)
- Interconnection buses

The controller and the datapath interface through the control signals, which are used to configure the steering logic, and the conditional signals (e.g. comparators' outputs), which express certain conditions and are used in the calculation of the controller's next state. As two independent sources of power dissipation, the controller and the datapath can be separately optimized. Additional power cuts can be achieved by carefully designing the interface between them as discussed in paragraph 2.5.1.

At this point it is important to distinguish between static and dynamic power optimization; a characterization that is independent of the abstraction level. Static refers to those techniques that do not change over time, in contrast to dynamic. Static techniques represent decisions throughout the design phase that affect average power consumption. For example, at the RT level, a low dissipative, but slower component, is selected against a faster, but more power consuming one. Dynamic refers to methods that in run time, under certain conditions, are activated to minimize power consumption (e.g. dual voltage operation under varying load requirements) by means of additional circuitry.

This circuitry imposes an area overhead, it may affect performance, if inserted on critical paths, and at corner cases, it may compromise the overall power reduction. Thus, an extremely important part of any power optimization algorithm is the identification of appropriate applicants to be power managed. RT level power estimation can be used for this purpose.

2.1.2 Power Consumption Guidelines in RTL Designs

Functional units are known to be the major power consumers in datapaths, however power distribution figures may vary among different applications. Ranghunathan in [45] differentiates between control- and data-flow intensive designs, stating the multiplexors and the registers by far as the major sources of power dissipation in control dominated designs. Table 2.1 summarizes the power distribution figures of the implementation of GCD[1] algorithm presented in the same article and the results of *PLAIN* design[2] (section 3.3) and supports the argument. As the power consumption sources may differ among designs, power optimization should be applied in a design specific manner.

Block	Power consumed GCD (% of total)	Power consumed PLAIN (% of total)
Functional Units	9.08	86.00
Random	4.67	1.00
Registers	39.55	12.00
Multiplexors	46.70	1.00

Table 2.1: Power distribution in the GCD implementation [45]

Musoll and Cortadella in [40] provide power models extracted by simulations for functional units in order to be used with high-level synthesis techniques. RTL design and high-level synthesis are closely related in the sense that in the latter, design experience is replaced by cost functions. In this respect, power models can be valuable tools for the RTL designer.

In [40], power dissipation is related to the switching activity of the input operands and more specifically to the hamming distance of two subsequent values. Simulations of an 8×8 Radix-4 Booth encoded multiplier showed 35% lower power consumption in the case when only one of the inputs changes, while the other remains constant. Similar guidelines are quoted in table 2.2.

In the table, factor β denotes the power relation between the adder and the multiplier, whereas factors α_{add} and α_{mull} denote the power ratio between operations with one and two operands changing, for an adder and a multiplier, respectively. It is important to note that those factors are weak functions of operand bit width and can be thus approximate factors for higher widths.

Another strong point implied is the affiliation of power dissipation to data correlation. Hence, it is advisable that data correlations are both searched for and preserved when available. In [59], for instance, two separate busses are preferred over a time-multiplexed one, when

[1]The Great-Common Devisor circuit (GCD) consists of functional units (three comparators and one subtractor), steering modules (10 multiplexors and 4 register banks) and random logic (FSM and decode logic

[2]*Plain design consists of 4 multipliers, 2 adders and 1 multiplexor*

Parameter	**Description**	**8-bit**	**12-bit**	**16-bit**
P_{add2} (nJ/op)	Avg. consumption of an adder when both operands change	0.35	0.53	0.90
P_{add1} (nJ/op)	Avg. consumption of an adder when one operand changes	0.26	0.4	0.70
P_{mul2} (nJ/op)	Avg. consumption of a multiplier when both operands change	5.7	13.68	28.9
P_{mul1} (nJ/op)	Avg. consumption of a multiplier when one operand changes	3.7	8.88	19.9
α_{add}	P_{add1}/P_{add2}	0.74	0.75	0.77
α_{mul}	P_{mul1}/P_{mul2}	0.65	0.65	0.068
β	P_{add2}/P_{mul2}	0.06	0.04	0.03

Table 2.2: Power models for functional units from [40]

the two streams are not correlated. Data correlations are very common in digital signal applications and should be utilized (e.g. the sigh- and higher order bits in a sequence of slowly changing two's complement numbers). In this respect, power estimation based on random, uniform input patterns may yield optimistic power savings. The transition probability of different order bits as a function of temporal correlation is given in [36]. There, the least significant bits have a uniform probability of switching independently of the correlation value. On the contrary the most significant bits are highly dependent on the correlation values.

Theoretical Concepts

The optimization techniques that follow aim at reducing switching activity in all parts of the design, the control and datapath units. Dynamic power management at the RT level is about extracting hold conditions of design components and by ingenious circuitry preventing useless switching power. In contrast to gate level description, extraction of hold conditions is very robust, yet they may be suboptimal [6]. Under this scope, RT level power management is a trade-off between computational efficiency and power saving.

Two terms are reported in [6] to formalize idleness of datapath components: *internal* and *external idleness.*

Internal idleness depends exclusively on the functionality of the unit in question. Giving the definition through an example, a multiplier which one input is set to zero is an internally idle unit, as any change in the other input is not observable at the output, even though the output port is fully observable.

External idleness is solely dependent on the environment of a component and is directly related to observability, the propagation of a change on the signal in question to primary outputs. A simple example is illustrated in figure 2.1: when the zero input of the multiplexer is selected, the output of the shifter is not observable and in this way useless. External idleness is very common in practical systems deploying several resources in parallel and will be extensively used as a means to minimize power dissipation.

Observability don't care condition (ODC) is the condition under which a signal is not observable at a primary output. It is computed by traversing the fan-out cone of a signal backwards from the primary output and concatenating the ODCs of the intermediate nodes met. ODCs can then be used to activate power management circuitry.

An important difference between internal and external idleness is that in case of the former, correct functionality has to be preserved (e.g. keep the output of the multiplier to zero, when one input is zero). In case of the later, the output of the externally idle unit is a "don't care" and could be set appropriately.

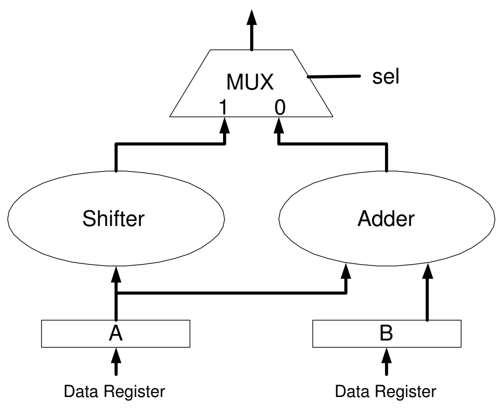


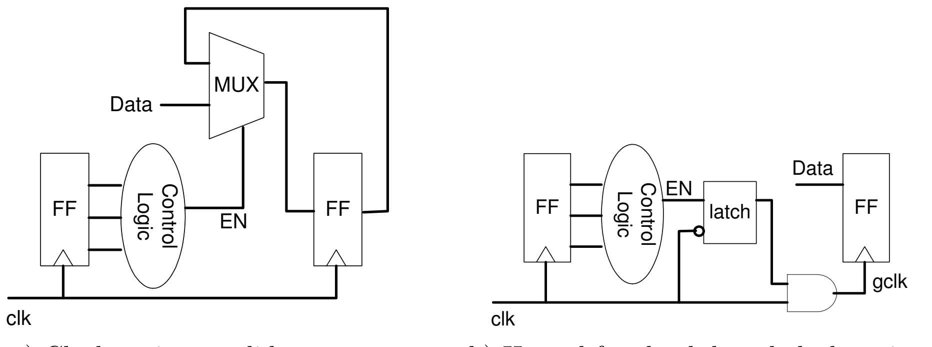
Figure 2.1: Example of external idleness

2.2 Clock Gating

Clock gating has in the last years changed status from forbidden “black magic” design craft to a well accepted power saving optimization method. The early unpopularity of clock gating is charged to the inability of the tools of that time to deal with the timing implications of the gated clock signals and by the reduced fault coverage achieved by logic testers. The purpose of this section is to clarify the principle of the clock-gating operation and to discuss its limitations and automation.

2.2.1 How it Works

Clock gating was originally conceived as a system level power optimization technique aiming to reduce the power dissipated on the clock network (which accounts up to 40% of the total) by deactivating parts of the system that are idle. Its applicability has been extended to the RT level as a power efficient implementation of registers on a hold condition. An enabled register is shown on the left of figure 2.2. During a hold condition, the register preserves its previous value at a high power cost. Unnecessary power is consumed on the clock line, the register itself and on the multiplexor on the feedback path. By controlling the clock driving the clock input of the register, reloading is only conditionally performed resulting in both reduced power consumption and area overhead.



a) Clock gating candidate b) Hazard-free latch-based clock gating

Figure 2.2: Implementing clock gating

2.2.2 Automation of Clock Gating

Although easy to apply, manual clock gating can be difficult to verify timing and testability wise. Due to the high potential savings at insignificant cost, clock-gating is fully automated in most commercial synthesis tools. This paragraph briefly introduces the automatic clock gating[3] feature in the Power Compiler tool from SYNOPSYS.

A robust set of options controlling all aspects of the implementation of clock gating are available to the designer in the form of variables, commonly included in a relevant script. Candidates are identified as registers that share the same clock and synchronous control signals, namely synchronous load-enable, set and reset signals, from sequential processes in HDL RTL code (see figure 2.3).

```
process(clk, rst) begin
    ...
    if clk'event and clk = '1' then
        if en = '1' then
            q <= data:
        end if;
    end if;
end process;
```

Figure 2.3: RTL identification of clock gating candidate

To be further considered, candidates should fulfill two requirements: their width should be higher or equal to the minimum set and the setup time of the enable signal should not be violated. The first condition has to do with the area and power overhead of the clock gating circuitry, which may be unjustifiable for small register banks. The setup condition qualifies correct operation and is dependent on the clock gating logic selected. The designer has two options at hand: a sequential and a combinational one. The former includes a latch to filter glitches from propagating to the clock signal during the first half of the clock cycle, while the latter is implemented with gates and is transparent to the glitches. For this reason, the sequential approach is strongly recommended, if this does not interfere with the setup condition. Isolation logic can be customized in many respects. One important choice is between *integrated* and *non-integrated* cells. Integrated cells refer to special library available clock gating components and should be preferred or resorted to in case the use of non-integrated ones results in setup time violation.

Regarding testability, additional observation points can be automatically inserted to improve controllability and observability problems caused by the clock gating logic.

2.3 Operand Isolation

Operand isolation is a technique to protect a functional block from being exposed to switching activity at its inputs by means of blocking logic.

It involves a candidate for isolation, the isolation circuitry and the activation condition that controls it. Figure 2.4, illustrates a common case that operand isolation assists clock gating, enabling better utilization of external idleness. Since, register A is shared by the two functional units, clock gating cannot block the switching activity in the shifter, when an addition is required. The activation condition for the blocking logic, active-low in this case, can be extracted by the *observability don't care* conditions as described in [54], an analytical method amenable to being automated.

[3]See chapter 9 in [53]

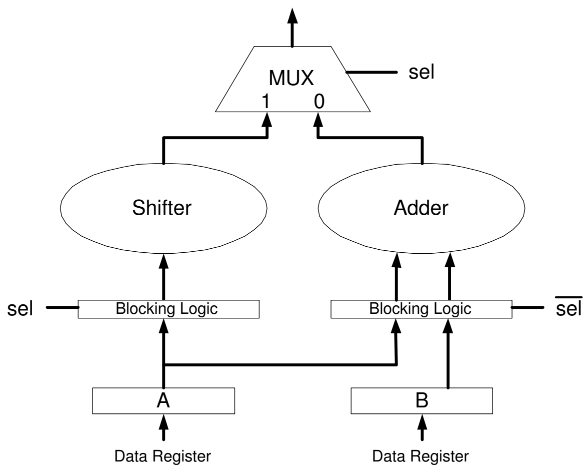


Figure 2.4: Operand isolated ALU

2.3.1 Implementation Details

Working at the RT level mitigates the tedious tasks of identifying operand isolation applicants and extracting activation signals, as the inputs to functional units and the multiplexor control signals can be readily used for this purpose.

The type of isolation logic used requires more consideration. Two approaches have been proposed [39]: a) transparent latches and b) combinational logic gates (AND/OR). In the former case, the latches are used to freeze the values of the inputs and in this way prevent the invocation of a new, redundant computation. In the latter case, inputs to functional units are isolated by setting appropriately the controlling inputs of combinational gates (a logical zero (one) for an "AND" ("OR") gate). "AND" or "OR" gates should be used for inputs with a high static probability to be assigned a logic one or zero, respectively.

Either implementations entail an area overhead; that is the sum of the area occupied by the isolation banks and the area occupied by the activation function. For most cases in RTL designs, the second term can be disregarded, as the control signals can usually be used as the activation functions.

Power saving can also be compromised by the power overhead in the isolation circuitry. Experiments performed on various testbench circuits under stimuli loads of different statistics in [39] showed that gate-based isolation yields at least equal power savings to latch-based isolation at a lower area overhead. Power reduction ranged from 12% to 30%, with 5% fluctuations under loads with different statistical properties. A known limitation of gate-based isolation is that isolation effectively takes place one clock cycle later, due to the isolation gates settling to their quiescent values, so it is not advisable for highly active activation signals. This drawback is eliminated in latch-based isolation at the expense of increased area and power overhead. Timing degradation can also be considerable and in cases unacceptable. Timing slack is decreased in two ways:

- As isolation banks are placed on the critical paths, their inherent delay is subtracted from the available slack.
- The timing path of the activation logic is also added on the critical path, further tightening timing constraints.

Despite those observations, in some of the experiments carried out in [39], the increase in the timing slack was annotated to additional boolean optimization opportunities emerging after the insertion of the isolation gates.

Testability is also affected by the isolation logic. Although, functionality is not put at stake, a stack-at-1 fault at the activation signal will render isolation logic inoperative and power dissipation increased by the isolations logic power overhead [53], as depicted in figure 2.5. For this case to be prevented, an additional observation point needs to be added.

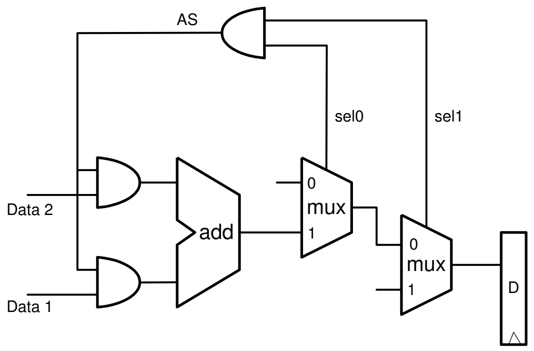


Figure 2.5: Unobservable stack-at-1 fault in operand isolation circuitry [53]

Taking into account the dependence of power dissipation to data correlations, two distinct power savings are expected, the *primary* and the *secondary*. *Primary* gains refer to minimization in the isolated unit, while *secondary* to savings in the fanout logic of the same unit. If the output of an isolated unit is an input to a unit in the same path, the reduced switching activity at the intermediate node will result in additional power reduction. For this reason, it is advisable that isolation logic is added as closer to the primary inputs of a design, as possible. Especially in gate-based isolation, it is extremely important that the activation signal is available at least at the same time as the operands to be isolated. A late arriving activation signal does not only impact timing, but also results in excessive switching activity and unexpected power dissipation. In such occasions the actual power dissipated is doubled due to the initiation of two useless computations, one with the new inputs and one with the isolation gates' quiescent value, plus the isolation's logic power overhead.

Based on the above discussion, operand isolation, if judiciously used, can considerably reduce unnecessary power dissipation at a small price. As power and area costs are somehow related to the width and the number of operands to be isolated, they can be amortized for highly complex arithmetic operators (multipliers).

2.3.2 Automation of Isolation Logic Insertion

As stated earlier, operand isolation is amenable to automation and, together with clock gating and resource sharing (not intended for power), they are the only RTL power optimization techniques that have found their way in commercial CAD tools, for example PowerCompiler[4] from Synopsys. This paragraph briefly introduces implementation of operand isolation in PowerCompiler. More information can be found in chapter 10 of [53].

Operand isolation is semi-automated, meaning that some interaction with the user is required. There are four tasks involved, in line with the algorithms presented in [39] and [54]:

a) Identification of operand candidates
b) Implementation selection
c) Extraction of activation conditions
d) Reporting and rollback

Identification is performed manually by the designer either in the RTL HDL code or in the GTECH level, the SYNOPSYS proprietary format of a design after the *analysis* and *elaboration* stages (see section 5.2.1). Figure 2.6 illustrates operand isolation in the RTL

[4]Yet, operand isolation is an infrequently used feature of PowerCompiler, in contrast to automatic clock gating

VHDL code. "*Pragmas*" are directives used to guide VHDL compiler and they only apply to singular arithmetic operators. If more complex expressions are used, they need to be partitioned into simpler ones containing a single operator or spanned over more lines.

```
...
...
p <= a + b; --pragma isolate_operands
...
...
```

Figure 2.6: Pragma based operand isolation in VHDL RTL code

Only "AND/OR" gate-based operand isolation is supported by SYNOPSYS and isolation logic is selected by setting a special variable (*set_operand_isolation_style*). Activation signals are automatically extracted[5]. Power compiler can be requested to generate timing, operand isolation and power reports as means to evaluate insertion of operand isolation logic. If the overhead is unacceptable, the designer can manually remove isolation logic by use of certain commands. Automatic rollback is also provided, if the maximum permissible negative slack is assigned a value prior to setting the design constraints and compilation.

As discussed in the previous section, operand isolation should be used with caution. Synopsys offers the simple guidelines below to assure successful operand isolation:

- Avoid isolating units when inputs are highly correlated to the activation signal (e.g. in case of a feedback loop from an output enabled register to the input of the unit)
- Choose sufficiently complex candidates (4-bit adder as minimum)
- Avoid isolating units that are highly utilized[6]

What is not suggested in the reference manual is simulation based power estimation. Library based power information may be highly unrealistic and in this way suboptimal or inferior power savings may be estimated. Power compiler supports back-annotation of switching activity information and this is highly recommended for more accurate results (see section 2.6).

2.3.3 Clock Gating and Operand Isolation Interaction

A limitation that may be resolved in later versions of the tool is the poor interoperability of the automatic clock gating and operand isolation features in Power Compiler. In the current setting, clock gating is introduced earlier in the design flow and may eliminate operand isolation opportunities by removing feedback multiplexors at the input of registers.

Figure 2.7 shows an example. By applying clock gating, the feedback multiplexor is eliminated and so is the activation condition. Depending on the complexity of the isolation candidate the overall power saving may be suboptimal.

The limitation is that when the output of a functional block is directly connected to a register, the activation signal extraction procedure does not consider whether the register itself is enabled or not. For this reason, it is advisable that the results of automatic operand isolation are carefully investigated and in some cases manually augmented. This may be necessary if latch based isolation is to be deployed. In chapter 3, the interaction of clock gating and operand isolation is elaborated and a combined approach is proposed to merge the merits of each. In chapter 5, alternative methods are proposed that overcome most of the above mentioned limitations.

[5]compilation effort should be set to high
[6]if utilization is more than 70% [53]

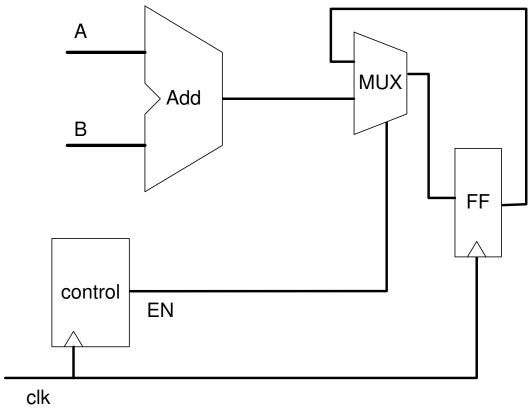


Figure 2.7: Operand isolation and clock gating interaction

2.4 Pre-computation

In [1], a powerful method for reducing useful switching activity, called pre-computation, is proposed. The basic architecture is shown in figure 2.8.

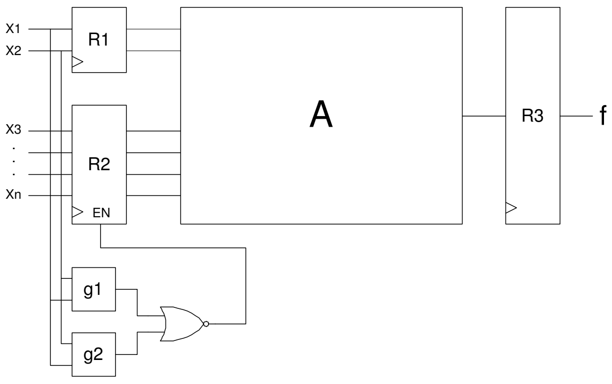


Figure 2.8: Subset input disabling pre-computation architecture

The method is based on selectively pre-computing the output of logic block A in figure 2.8 using the logic sub-blocks *g1* and *g2*, hereafter predictor functions, one clock cycle in advance, and using the pre-computed values in the preceding cycle to effectively reduce the switching activity and power. This is done by deactivating register *R2* and exposing block *A* only to a subset of the new inputs, those that have an effect on the output value.

The optimization task of extracting the predictor functions is of primary importance as power savings are offset by their power and area overhead. The objective is to maximize the probability of either of the predictors evaluating to a logic one, covering as many as possible of the input combinations that belong to the observability sets of the individual input variables as described in [36]. The basic architecture can be extended to apply to functions with multiple outputs and disabling of all inputs at the expense of higher complexity in the calculation of the predictor functions and increased power and area overhead. Timing performance should also be considered and if detrimental, critical paths should be excluded from the selection procedure.

The method was originally intended for strictly combinational circuits (gate level description or random control logic). It is fully automated and power reduction figures up to 75% for random logic are reported in [1] with insignificant area and timing overhead.

Power reduction up to 60% for functional units (comparators) were also reported. The applicability on functional units though is limited to control oriented blocks that contain comparators, carry select adders and *MIN/MAX* functionality [36]. This is due to the prohibitive overhead of the pre-computation logic imposed by the large number of inputs and outputs of the functional units; for example, all bits of both inputs to an adder are needed for the computation to be correct. In addition, pre-computation of functional units does not lend itself for automation and needs to be manually implemented. In that respect and because of the promising results achieved, it is recommended as an RTL power optimization technique, whenever applicable.

2.5 Minimizing Switching Activity

Similar to pre-computation, the methods described in the following are not dependent on the existence of idle conditions. They aim at reducing spurious transitions (glitches) which account for a considerable amount of the total power consumption, especially in designs with long paths.

2.5.1 Glitch Power Minimization

Glitching power in data-flow intensive designs is attributed to chaining of arithmetic functional units. Their outputs fluctuate before they stabilize to the final value and this switching activity is propagated down the fanout logic. In control-flow dominated designs, although the controller itself only accounts for a small fraction of the dissipated power, glitches on the control signals created by the decode logic may propagate to the datapath and in this way cause excessive switching activity. The generation of glitches and ways to eliminate them are presented in [45]. These methods have been automated in a tool and applied to testbench designs resulting in power savings up to 30%.

The suggested techniques are:

- Use of glitch blocking multiplexors
- Restructuring of multiplexor networks to enhance data correlations
- Restructuring of multiplexer networks to eliminate select signals with high glitch context
- Control Signal clocking
- Delay insertion

The first three techniques aim at reducing glitch power in multiplexer networks both in the select and input signals. The glitch context at the multiplexors' outputs is data dependent and the first technique proposes a modified architecture based on that observation. Restructuring, at the second technique, aims at creating opportunities for utilization of the modified multiplexor by creating data correlations. According to the third method (similar to technology mapping at the gate level), the multiplexor network is restructured to eliminate highly switching select signals either by using alternative ones or by pushing them closer to the end of the fanout path, to limit their effect. Clock gating of either select or input lines during the first half of the clock cycle is proposed as the final resort only to paths with positive time slacks. The effect is that a logic block is limited to perform minimum one and maximum two computations: one during the first half of the cycle on the gated values and, conditionally, one during the second half on the newly calculated and stabilized values computed by the fanin logic. Latch-based control signal gating, similar to latch based operand isolation eliminates the first computation. It can also be seen as a method to insert a pipeline stage operating on the falling edge of the clock in the middle of the path.

Glitches are the result of converging logic paths with varying delay and buffer insertion has been proposed as a countermeasure. Because of the power overhead in the delay elements and its vulnerability to process fluctuations, it is not recommended.

2.5.2 Retiming for Low Power

Retiming was originally proposed as a gate level method to minimize clock periods by inserting flip-flops (pipelining) or by changing the position of the existing ones. By increasing performance, voltage could be scaled down to match the throughput requirements with reduced power dissipation due to the quadratic dependence of power to the supply voltage (formula 1.1).

In [37], the authors propose a modified cost function that is power aware and tries to place flip-flops under timing constraints in a way that minimizes switching activity. The method is based on the fact that a flip-flop makes only one transition in a clock cycle and in this respect it is glitch-free.

Retiming in commercial CAD tools only takes performance into account. For example in Synopsys Design Compiler, retiming is used to create pipelined functional units by redistributing a cascade of registers placed at the output of the unit. For this reason, the designer should still be aware of the potential merits of carefully placing registers in the design. An example at the RT level is given in section 6.3, where power sensitive retiming is applied on a multiply accumulate unit.

2.5.3 Low Power Control Unit

In [22], the propagation of glitches from the control unit to the datapath is discussed. Looking at the controller in isolation, power can be spared both in the state register and in the next state logic by careful state assignment. Minimum hamming distance encoding (e.g. gray encoding, one-hot encoding) have been used to minimize switching activity at the state register. However, it was found that this resulted in larger next-state logic blocks due to the high I/O requirements, despite the reduced transition count [5]. Thus, state encodings of minimum state variables are recommended.

Clock gating has also been investigated as a power reduction technique for finite state machines. In [58], a priority encoding scheme is proposed, where multiple codes are assigned to states to enable more efficient clock gating. In [8], Moore state machines are praised for being clock gating friendly due to the ease of extracting idle conditions, in comparison to their Mealy alternatives. Further, transformations from Mealy to Moore type machines are used to reveal self-loops that lend themselves for clock gating. Power savings are reported to range from 10% to 30% using a fully automated synthesis process starting from a state-table specification.

In a way similar to pre-computation, [35] proposes the decomposition of the FSM into two sub-FSMs, a small and a bigger one. The former, due to its limited size dissipates little power. The states it includes are selected in a way that the sum of the transition probabilities between any two of them (other than the RESET state[7]) is the largest possible, while the sum of the transition probabilities involving the RESET stage is as small as possible. The above conditions guarantee that the small FSM will be active most of the time and that the transitions from one machine to the other will be kept to minimum. Under these specifications, the larger FSM can be shut off by clock gating for a large fraction of the time, resulting in significant power savings. Reduction in power consumption up to 80% in the control logic is reported.

2.5.4 Encoding for Low Power

Bus encoding was originally used to account for error correction in noisy channels. In power constraint applications though, techniques to reduce the per transfer power dissipation have been devised aiming to reduce switching activity on the bus. Several of these schemes

[7]the interface state between the two sub-FSMs

are described in [16]. The bottleneck of all coding schemes is the encoding and decoding procedures. For instance, the logarithmic number system could be used to greatly reduce the power dissipated in multipliers, but the prohibitive conversion cost, would render the approach power inefficient. The *Bus-invert* code does not suffer from this problem and is for this reason considered. According to this code, the source word or its "1's complement" is transmitted to yield a word of minimum hamming distance from the one previously transmitted. Its applicability on datapath design is mainly due to the following reasons:

- It has low encoding/decoding overhead.
- It is not based on an algebraic method[8]
- Arithmetic on one's complement signed numbers is well understood, yet more complicated than two's complement arithmetic [43]

For this reasons, further investigation is worthwhile to evaluate whether the power overhead offsets the power savings. The overhead is mainly due to the conversion layers, the additional bit lines to control the decoder and the relative performance of 1's and 2's complement arithmetic.

Gray code arithmetic is the topic investigated in [19], where it is reported to have an unacceptable area and power overhead, despite its intrinsic low switching. To overcome this, a hybrid arithmetic was devised that uses gray encoded sub-blocks with binary carry propagation techniques among them, resulting in increased area, but reduced timing and power performance compared to binary array multipliers. The above mentioned techniques, despite their potential power efficiency, are far from the established and well understood 2's complement arithmetic. In a synthesis based design, where functional units are selected from IP libraries, integrating units operating on unconventional arithmetic systems, will require conversion layers between the different systems that will most likely offset the obtained power savings.

2.6 Power Estimation

Power estimation is critical for power optimization, as it enables design space exploration and design validation, before the design is actually laid out on silicon. Due to the tighter power constraints, there is a very high demand on accurate estimation to the degree of absolute values, which has made power estimation a very active field of research. As was suggested in figure 1.2, power estimation should be performed all along the design flow, from the system down to the physical level. The higher the level, the lower the accuracy and estimation time.

The purpose of this section is only to introduce the concepts behind power estimation to the degree that allows the evaluation of the power optimization techniques described above and the interpretation of the experimental power figures obtained. The discussion is limited to power estimation at the gate level, which is used in the experiments described in this report.

2.6.1 Gate-Level Power Estimation Basics

Accuracy in power estimation is all about modelling the circuit, as close, as possible to the actual implementation, so that the simulation of the derived model will emulate the activity of the physical circuit.

Modelling issues

The parameters used to capture the power behavior of a circuit are:

[8]refers to encoding based on more than the current value e.g. the previous one

- Delay
- Capacitance

Logic gates are modelled by their delay and the capacitance of their I/O pins provided by the technology library. Interconnection nets are described by their capacitance, which is the capacitance (parasitic, gate and drain) of all pins that are connected to it. This information together with library cell information and the switching activity of the internal nodes of the circuit are used to calculate power dissipation by formula 1.1. Leakage power is statically calculated by the leakage characterization of the circuit's cells. Dynamic power, which consists of the internal and switching terms, is calculated as a function of the cells internal power, the capacitance of the nets and the nodal switching activity. From all variables, switching activity poses the highest computational challenge ant it is related to the way the delay of the cells is modelled.

Three approaches to modelling the gate delay are available in increasing order of accuracy:

a) Zero-Delay model
b) Unit-Delay model
c) Real-Delay model

In the zero delay model, all gates in the circuit have zero internal delay for all input combinations and all output pins. Thus, in every clock cycle, every gate can only make one transition. In this way, the propagation of the new input vector that stimulates the circuit occurs instantaneously, which is far from how actual behavior of the circuit. As a result, the intermediate states that a circuit is going through (glitch power) before it stabilizes to the quiescent state by the end of the clock cycle are not accounted in the calculation of the nodal switching activities, resulting in optimistic power figures.

Under the Unit-Delay model all gates for all input combinations have the same delay. During a clock cycle of a unit-delay simulation, a gate can make several transitions due to the finite propagation delay. This results in considerably more accurate power estimation, at the expense of higher computation times. However, the uniform delay means that all input signals are arriving simultaneously, which is not the case in reality.

The Real-Delay model, as implied by the name, bares the closest resemblance to the actual behavior of the physical gates. Phenomena that are taken into consideration are unequal pin-to-pin delays and different rise and fall delays. This method yields the most accurate results, however computational times may become impractical for large designs.

Estimation methods

One way to calculate the nodal switching activities is by simulation (dynamic method). Due to its data dependent nature, it is very important that the simulation patterns represent the properties of the actual work load. Another way is to describe the input data by their probabilities and use the logic of the circuit to propagate the probabilities to the internal nodes (static method).

2.6.2 Gate-Level Power Estimation with SYNOPSYS Power Compiler

The Power Compiler tool from SYNOPSYS is integrated with the Design Compiler synthesis engine and they can work together to estimate the power and to optimize a design under timing, area and power constraints. Both applications are available at the RT and gate level.

RTL power estimation

At the RT level, the two power optimization techniques offered are clock gating and operand isolation, as described earlier in this chapter. Power estimation at this level is done by annotating the primary inputs and the synthesis independent elements (hierarchical port map elements) with switching activities obtained by RTL level simulation and using the zero-delay model to propagate them in the gate level design after synthesis (static power estimation). Because of the zero-delay model and the probabilistic nature of the power analysis, accuracy is traded off for fast runtime.

The accuracy offered by RTL power estimation is enough to evaluate power behavior of different architectures, identify which module consumes the most power and where power is consumed within a block.

Gate-level power estimation

During the gate-level power estimation, power is calculated by using the switching activities extracted from the simulation of the gate-level design. The accuracy of power estimation depends both on the delay-model used during simulation and the number of nodes been annotated with simulated switching activity. Gate level estimation calculates all nodes' switching activities. If the simulator supports full-timing gate level simulation, the highest accuracy is achieved, as path dependencies and glitching phenomena are accounted for.

The simulator used in the experiments in this report (VSS) does not support full-timing gate-level delay model (SPICE does), so the results do not include glitching power.

Methodology flow

Figure 2.9 illustrates the methodology flow for gate-level power estimation and optimization.

Two steps are involved in gate-level power optimization:

a) Optimizing for area and timing
b) Optimizing for timing, area and power

During the first step the design is optimized for area and timing and during the second for power, as well. The priority of the constraints can be set by the user. By default, power optimization is performed without violating the timing constraints. The higher the positive slack on the critical path, the higher the optimization options. Gate-level power optimization is expected to improve power by 10-15%[53].

2.7 Summary

This chapter presented the power optimization techniques that are available at the RT level in a synthesis based design flow. The main focus is put in data-flow intensive designs, where functional units are identified as the major power consumers, in contrast to control-intensive designs dominated by the power consumed in register banks and multiplexor networks.

The power optimization techniques discussed target the reduction of either useless or useful switching activity and they are further divided into dynamic and static. Dynamic methods are special in the respect that their behavior is decided in runtime and depends on the qualification of hold conditions expressing either external (clock gating, operand isolation) or internal idleness and ”don't care” states (precomputation). Power reduction is then achieved by the means of “blocking” circuitry controlled by the hold conditions extracted.

Due to their hardware overhead and the data dependent nature of the expected power savings, efficiency of dynamic power management techniques should be evaluated on a per

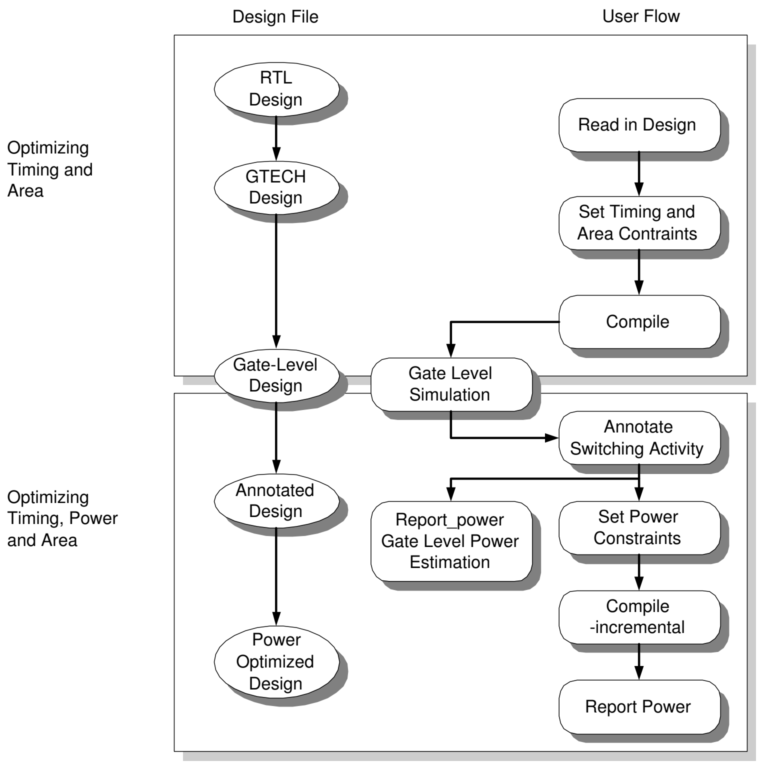


Figure 2.9: Gate-level power optimization methodology flow

application basis, as under certain conditions, power savings may be offset by the power overhead.

In the domain of interest, only clock gating and operand isolation appear as the most promising techniques, as the power overhead of precomputing the output of arithmetic functional units is prohibitive. Potential for power saving is also found in the control unit and at the interface with the datapath.

Static power optimization methods target the elimination of highly active nodes in the circuit. Restructuring of multiplexer networks, use of control signals with high static probability and power sensitive retiming of registers are the techniques that fit the purpose of this report.

In the following chapters, the methods discussed are eveluated through sample design experiments.

Chapter 3

Experiment 1: A Complex Arithmetic Unit

Complex arithmetic operations appear commonly in digital signal processing, and for this reason they are included in DSP processors usually as a separate unit, in parallel with the integer, logic and floating point units that form the execution stage of any modern general purpose processor. It is generally accepted that the major part of power consumption in processors is due to memory accesses and multiplication. Thus, for considerable power saving one should focus on these two operations. As this work focuses on power efficient arithmetic circuits the discussion will be limited on multiplication.

In this context, complex multiplication is recognized as a “hot point”, as it comprises four multiplications, an addition and a subtraction as dictated by the formula 3.1 and illustrated in figure 3.1. The real and imaginary parts of complex numbers are represented as 16-bit signed fractional numbers and a complex operand can be accommodated by a 32-bit word with the real and imaginary parts occupying its higher and lower part, respectively.

A permutation of 3.1 yields the alternative expression 3.2 with only three 16bit multiplications, but 3 more addition operations. This represents an optimization at the algorithm level that clearly illustrates the higher potential savings that can be obtained at higher levels of abstraction. Despite the higher potential saving, expression 3.1 is implemented and an evaluation of this choice will follow at the end of this chapter, after the experimental results have been presented.

$$(a + jb)(c + jd) = (ac - bd) + j(ad + bc) \tag{3.1}$$

$$(a + jb)(c + jd) = (ac - bd) + j((a + b)(c + d) - ac - bd) \tag{3.2}$$

In this chapter a simple complex arithmetic unit is designed to serve as the platform on which common dynamic power management techniques are applied to reduce its power consumption.

3.1 Design Considerations

Figure 3.2 shows a common architecture of the execution stage of a processor. The different computational units share the same input bus and their outputs are multiplexed based on the operation code control field indicating the current instruction. Although this is a very well defined architecture, it suffers from unnecessary power consumption. All units are active during every clock cycle doing possibly correct computations, while all results but one will be discarded. This case is identified as “external idleness” (see section 2.1.2) and provides

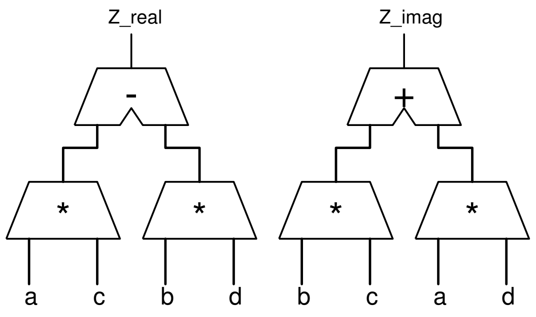


Figure 3.1: Complex multiplication block diagram

for dynamic power management, a technique to switch off parts of the design that are not active or not performing useful computation. In this case the operation code control field can be used as the disabling condition of inactive units.

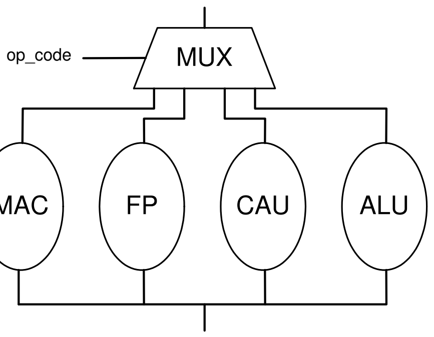


Figure 3.2: Execution stage of a DSP processor

In general, processors are not known for their high degree of utilization. Although they are capable of operating on maximum work load and delivering top performance, their average performance lies a little bit above idle. In this line, if care is taken to design different parts to operate on a mutual exclusive basis, power consumption can be significantly reduced down to the minimum possible. It is widely accepted that timing performance is not to be compromised and chip real estate comes at low cost. Based on these guidelines area and design effort remain to be traded-off for power. This statement is conflicting with the traditional belief that low area correlates to low power, unless dynamic power management techniques are brought in to play [6]. What defines the power efficiency of this strategy is the power overhead of the power management circuitry compared to the power savings and its utilization.

Going one hierarchical step down, and looking into the CAU, there are four 16-bit multipliers to allow for the completion of a complex multiplication in one clock cycle. Depending on the frequency of complex multiplication instructions, one could investigate whether it is worthwhile to share those resources with other instructions. Looking from the power perspective, the answer is not clear. Resource sharing is not longer blindly recognized as a way to save power. It has been proven, that resource sharing destroys data correlation which in turn results in higher switching activity that can be directly translated into higher power dissipation [59]. From this standpoint, power consumption becomes data dependent, so before abundant power reduction figures can be claimed, power estimation needs to be done based on realistic simulation patterns.

3.2 Design Specification

The design example used has intentionally been kept simple to illustrate the sources of power consumption, their cause and allow for simple to interpret and correlate power measurements before and after the application of power management techniques. This section deals with the functional specification of the design.

Instruction Set

The CAU implements the following instructions:

- Complex multiplication
- Multiplication on two pairs of 16-bit signed fractional numbers
- Multiplication of a single pair of 16-bit signed fractional numbers
- No operation

Additional instructions would be single/parallel addition/subtraction and multiply accumulate instructions on a sequence of complex numbers.

The block diagram of CAU is shown in figure 3.3. Similarly to the overall architecture of the execution stage (figure 3.1), the individual results are merged through a 3-to-1 multiplexer controlled by the operation code control field. The concatenation modules, annotated by "&", form the results of the individual instructions. They do not incur any more logic than manipulation of wires to account for truncation of the least significant bits to match the available precision and remove redundant wires that are used to evaluate overflow conditions. Although the overflow detection logic does not appear in the figure, the functionality is explained in a subsequent section.

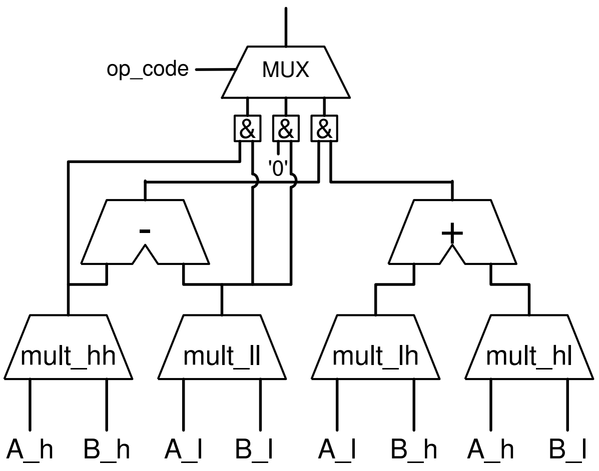


Figure 3.3: The complex arithmetic unit with enriched instruction set

The mnemonics assigned to the individual instructions in the order presented above are shown in table 3.1.

Instruction	Mux input	op_code
MCX	2	"1000"
MPF	0	"0100"
MSF	1	"0010"
NOP	-	"0001"

Table 3.1: CAU instruction set

The operation codes are one-hot encoded. Although this adds to the register count for the pipeline registers, it simplifies control logic and improves timing, as it will be explained later.

A "no operation" is present in all instruction sets and if not designed carefully it can also result in unnecessary power consumption, for example if the output is reset to logical zero. To avoid this situation it is sufficient to disable the control information carrying signals that can alter the state of the machine. Operands can then maintain their previous value, since they are not going to be written. This saves power both in the register banks and in the functional units, which would otherwise change state and hence dissipate power. The main idea that will be quoted in numerous places throughout this report is *to prevent any useless switching activity.*

Implementation

The top level design is split into two high level entities, a sequential (REG_BANK) and a combinatorial (CAU). The former implements the I/O registers, which can be perceived as the pipeline registers of the execution stage of a processor. The latter is the complex arithmetic unit and it includes the arithmetic and control units. The power management circuitry will also be considered as a part of the CAU to ease evaluation of power figures.

The size of the design does not justify the architectural partition in a sequential and a combinational block and the incurred code overhead. However, it serves the purpose of allowing the generation of hierarchical analysis reports (area and power), which allows unambiguous evaluation of the performance of the power management techniques applied.

Registers

The CAU is a combinatorial circuit and its inputs and outputs are registered as shown in table 3.2.

Register name	Width	Direction	Comment
in_A	32	I	Operand 1
in_B	32	I	Operand 2
opcode	4	I	Operation code
Z	32	O	Result
ovf	1	O	Overflow flag

Table 3.2: The input output registers

An optional property available is clock gating as described in section 2.2. It is available for all registers but the "opcode", which carries sensitive control information and should therefore always be enabled.

Arithmetic units

As illustrated in figure 3.3, the CAU comprises four multipliers, an adder and a subtractor. The implementation of those units are analyzed in table 3.3.

Furthermore, each multiplier consists of a wallace tree structure followed by a 25x25bits adder with Brent-Kung architecture (see section 5.1).

At this point, the implementation selection for the arithmetic units was left to be taken care of by the synthesis tool, based only on timing and area constraints. The idea was first to investigate the merits and overhead of the power management techniques alone and then to

Unit name	Implementation	Width
adder	Ripple carry adder	32x32
subtractor	Ripple carry subtractor	32x32
mult_hh	Non-booth encoded wallace tree multiplier	16x16
mult_hl	Non-booth encoded wallace tree multiplier	16x16
mult_lh	Non-booth encoded wallace tree multiplier	16x16
mult_ll	Non-booth encoded wallace tree multiplier	16x16

Table 3.3: Implementation of arithmetic units after synthesis

compare them against the results gained by the use of low power implementations of the same modules. Power management circuitry entails a certain power overhead that offsets power savings, thus there is a turn-over point; and it is one of the objectives of the thesis to discover this point.

Another possible degree of freedom is the accuracy in the calculations. At this point, no precision is sacrificed before the final, full-precision result is truncated. It could be worth investigating, whether precision in the computation could be traded-off for reduced hardware and hence lower power consumption [32]. This would however require design of customized arithmetic components.

Finally, timing and pipelining could be taken into account. Timing closure is not considered a problem, however if units were to be pipelined, the insertion point of pipe stages should be power sensitive, aiming at reduced switching activity as suggested in section 2.5.2 and applied in the design described in section 6.3.

Control logic

The control unit is responsible for the tasks of selecting the correct result, activating power management circuitry and detecting overflow.

For some applications (real-time signal processing, multimedia), performance is not to be compromised for power, so any power saving technique that degrades performance is bound not to find wide acceptance and applicability. However, the requirements may be relaxed, if there is some available slack. Both the clock gating and the operand isolation enable conditions are derived from the operation code. To minimize the delay, the depth of the logic inferred should be held as low as possible. In that respect, the absolute limit which can occasionally be achieved is zero. This only occurs when one-hot encoding, instead of binary, is used for the operation code, as illustrated in table 3.4.

	1-hot			Binary		
Instruction	code	cond.	depth	code	cond.	depth
MCX	"1000"	op(3)	0	"11"	$op(1) \cdot op(0)$	1
MPF	"0100"	op(2)	0	"10"	$op(1) \cdot \overline{op(0)}$	2
MSF	"0010"	op(1)	0	"01"	$\overline{op(1)} \cdot op(0)$	2
NOP	"0001"	op(0)	0	"00"	$\overline{op(1)} \cdot \overline{op(0)}$	2

Table 3.4: Power management delay overhead VS encoding style

For instance, for one-hot encoded operation code and isolation logic that is transparent when the control input is high, no other timing than the propagation delay through the isolation latch is added on the timing path. This point will be further clarified when the architecture based on operand isolation is introduced later on.

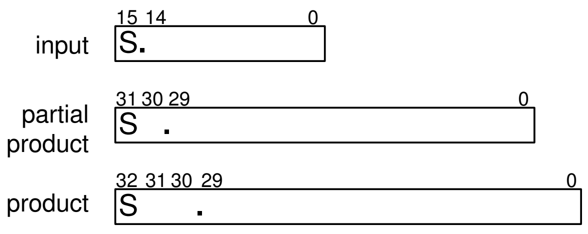


Figure 3.4: Representation of signed fractional intermediate results

The second role of the control logic is to set the overflow flag. In the design the length of the intermediate results is chosen so that no overflow can occur. However, since arithmetic is performed on signed fixed point fractional numbers, the available range is limited from -1 inclusive to +1 exclusive. Overflow can occur either at the final real and imaginary part of the complex multiplication which range from -2 to +2, both inclusive, or at the intermediate products which range from -1 to +1, both inclusive. Overflow on the partial products in the case of complex multiplication is not accounted for as the subsequent addition/subtraction may restore the result within the legal range. For the *MCX* instruction, overflow is identified when bits 32 down to 30 at the product holding register are not identical (see figure 3.4). For the *MPF* and *MSF* instructions overflow has occurred if the bits in positions 31 and 30 of the partial product are different.

3.3 Test Environment

To allow comparisons of power management techniques a testbench and a reference design are required.

The Testbench

The data dependent nature of power estimation calls for special attention on stimulating the synthesized design with realistic test patterns. The instruction mix in the testbench used comprises 50% “nop” instructions. The other 50% is shared equally to the remaining instructions. The “nop” instructions account both for the case that the execution stage is idle and the case that only CAU is idle and computation is performed on another execution unit. An unrealistic distribution of instructions does not invalidate the obtained results, as their data dependent nature is well understood and taken into account. For different workloads, results may appear degraded or upgraded depending on the relative frequency of the instruction that utilize the isolation logic. At the extreme where isolation logic remains transparent (100% MCX instruction mix), power figures will be actually worse due to the power dissipated in the isolation logic.

Instruction	Percentage
MCX	16%
MPF	16%
MSF	16%
NOP	52%

Table 3.5: Instruction mix

The two operand values are provided by two uncorrelated random number generators based on a uniform distribution. A logical one and zero are by default equiprobable, however a biased distribution towards “one” or “zero” is an option. The rationale is that switching activity is dependent both on the type of the isolation logic (gate- or latch-based) used for power management and the characteristics of the workload. So, different logic may perform better for certain input data.

As discussed in section 2.1.2, random data based on a uniform probability are not representative of applications where data are represented s in two's complement format. However, such a setting will yield the upper limit of power saving, which is still a valuable piece of information.

The reference point design ("PLAIN")

In this design the resister banks are implemented as free running registers and the CAU is not power managed. This configuration yields the highest switching activity and thus the highest power consumption which was found to be 4.47mW. The method used to estimate power dissipation is described in section 2.6. Throughout this chapter, power distribution in a design and power figures of individual components are expressed in percentage form relatively to the *PLAIN* design.

As illustrated in table 3.6, the register banks account only for 12% of the total power consumption, while the rest is mainly distributed evenly between the multipliers. The adder and the subtractor together are only responsible for 4% of the total power consumption. The important result from this experiment is that the multipliers are confirmed as the hot spots in a datapath, as stated in the previous chapter. It was also noticed that this relative power distribution holds for different instruction mixes.

Component	**Power share(%)**
REG_BANK	**12.00**
CAU TOTAL:	**88.00**
adder	2.00
subtractor	2.00
mult_hh	21.00
mult_hl	21.00
mult_lh	21.00
mult_ll	21.00

Table 3.6: Power distribution in the CAU (%)

In the subsequent sections, the clock gating and operand isolation power management techniques are applied to the *PLAIN* design and improvement in power consumption is estimated. In the immediate section, power savings from applying automatic clock gating are investigated. The second section introduces four designs to evaluate operand isolation alone and in cooperation with clock gating. The last section, focuses on the power overhead of resource sharing.

3.4 Clock Gating

Clock gating is a widely established power saving technique and can be found in most commercial designs. At a high level, it can be applied to idle blocks of a design resulting in considerable power saving both internally to individual blocks and on the clock tree. At the register transfer level it can be used as a replacement to enabled register banks, as explained in section 2.2. To evaluate the benefits, two additional designs are created: "REG_EN" and "CLK_GATED". They only differ from the plain design in the register bank entity.

The "REG_EN" architecture

This architecture uses the fact that not all registers should be updated at every clock cycle. In all cases, it is assumed that the necessary control signals (opcode) are propagated to the following pipeline stages, to prevent incorrect results from contaminating the state of the machine. The register enabling conditions and the expected results are summarized below:

- **MCX**
  No register is eligible to be hold inactive $\Longrightarrow$ No gain
- **MPF**
  No register is eligible to be hold inactive $\Longrightarrow$ No gain
- **MSF**
  a) The upper half of the input registers can be hold inactive, when a MSF instruction is about to enter the execution stage
     $\Longrightarrow$ *mult_hh* and half the input registers are held inactive
  b) The upper half of the output register can maintain its previous value
     $\Longrightarrow$ half the output register is hold inactive
- **NOP**
  a) Non of the registers holding the input operands should be updated when a NOP instruction arrives before the execution stage
     $\Longrightarrow$ the CAU and all input registers are hold inactive[1]
  b) Non of the output register needs to be updated when a NOP instruction is about to exit the execution stage
     $\Longrightarrow$ all output registers are hold inactive

The principle behind all these guarding conditions is, whenever possible, to prevent switching activity by not presenting functional units with new operands that would initiate a computation. The power merits over the plain design are gathered in table 3.7.

Component	Improvement(%)
REG_BANK	**4.27**
CAU TOTAL	**57.06**
adder	1.15
subtractor	1.49
mult_hh	16.12
mult_hl	12.99
mult_lh	12.79
mult_ll	12.19

Table 3.7: Relative power improvement (%) over PLAIN

It is evident that considerable power (57%) was saved. This is mainly due to preventing unnecessary computation while executing NOP instructions. The higher improvement in the upper part multiplier (mult_hh) is due to the MSF instruction, under which it remains inactive.

The "CLK_GATED" architecture

This design differs from the previous one only in the respect that enabled registers are replaced by clock gated ones by using the automatic clock gating facility provided by the Synopsys Design Compiler as described in section 2.2. After examining the clock gating report, 97 out of 101 registers were clock gated. The 4-bit register holding the opcode was correctly excluded, as it did not have an enable signal.

[1]setting input registers to zero in case of a nop initiates a totally useless computation. The effect is more immense when nops and other instructions alternate

As expected, only an improvement in the register bank entity was noticed, while the gain achieved in CAU remained intact; which confirms that the estimation method works correctly. An additional power improvement of 44% over REG_EN was noticed in REG_BANK. This is due to the power consumed by the feedback multiplexers in the enabled registers and the disabling of the registers.

The overall conclusion of this experiment is that clock gating is confirmed as a way to reduce power consumption in sequential logic. More importantly though, as it is effectively used as a means to isolate combinational fanout logic, high expectations from operand isolation based power management are built. How clock gating and operand isolation compare and combine with each other is the focus of the next experiment.

3.5 Operand Isolation

Operand isolation has not attracted considerable attention due to the incurred power, area and timing overhead. In respect to area, the isolation logic circuitry is significant, in contrast to clock gating, where area is actually benefited. As regards power, clock gating has a triple effect: it reduces power on the clock tree, at the register banks and in the functional units. Though, only the latter part will be considered, as operand isolation only relates to power in functional units. Under this condition, operand isolation by clock gating comes for free and the benefits are unconditional, as no additional logic is inferred. On the contrary, the net power gain from operand isolation is the one after the power overhead of the isolation circuitry has been deducted. Finally, timing wise, the main reason why operand isolation has remained just an idea and has not made its way to commercial designs is its impact on timing, as the isolation logic is added on the critical paths of the design. In this experiment it is assumed that timing is not an issue, however as described before, effort will be put on keeping the imposed delay to minimum.

In the following, three designs are compared against the PLAIN base design: *OP_ISOL*, *REG_EN_OP_ISOL*, *CLK_GATED_OP_ISOL*.

The "OP_ISOL" architecture

In this implementation, the register banks are free running and all inputs to all multipliers are operand isolated. This means that enable latches separate the shared input buses from the multipliers, as shown in figure 3.5. The adder and the subtractor are left unguarded, because of their insignificant power dissipation. Isolation logic could be added at the output of the multipliers in order to isolate the chained functional units both from unnecessary activity (MSF, MPF) and glitches from the multipliers in case of MCX instructions. Because of the insignificant contribution of the adder/subtractor units, the insertion of additional isolation logic is not justified. Yet, some secondary savings (see section 2.3.1) are still achieved by isolating the fanin logic.

By applying so fine operand isolation, it is made sure that only the necessary multipliers are active. The conditions under which latches become transparent and the incurred delays are:

Isolate	Condition	depth
mult_hh	opcode(3)or opcode(2)	$1 + t_L$
mult_hl	opcode(3)	t_L
mult_lh	opcode(3)	t_L
mult_ll	$\overline{opcode(0)}$	$1 + t_L$

In the table, t_L stands for the latch delay. As it can be seen logic depth is limited to 1 gate delay plus t_L. This can only happen if one-hot encoding for the opcode is used and

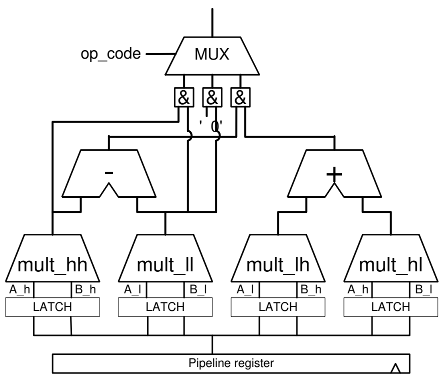


Figure 3.5: Latch-based operand isolation in the CAU design

resources are highly mutually exclusive. The area overhead of the isolation latches is 6.5% over the plain design; to visualize the overhead, it is approximately as much as an extra set of pipeline input registers. This point is used in the next chapter, where the duplication of input registers is evaluated as an alternative isolation method.

The decomposition of power improvements in the CAU is presented in table 3.8.

Component	**Improvement(%)**
REG_BANK	**21.52**
CAU TOTAL	**66.08**
adder	1.77
subtractor	1.36
mult_hh	15.31
mult_hl	19.14
mult_lh	18.89
mult_ll	11.38
Isolation logic	-2.07

Table 3.8: Relative power improvement in OP_ISOL(%) over PLAIN

Operand isolation compared to clock gating yields an additional net improvement of 9% in the CAU. The isolation logic is responsible for 2% of total power consumption. It is obvious that the power overhead is insignificant even for the extreme case that all inputs are isolated. There is also a 20% improvement in the register bank. This is due to the fact that after the insertion of latches, input registers see less capacitive load. Although it appears significant, one can not be sure before load capacitances have been back-annotated after physical layout.

The "CLK_GATED_OP_ISOL" architecture

The previous experiment proved that operand isolation offers great power savings and that it outperforms clock gating. After all, clock gating is meant as a power reduction technique for sequential logic. The purpose of the next experiments is to investigate how operand isolation performs together with clock gating.

This design deploys both clock gated input registers and operand isolation latches. Power dissipation in the CAU remains intact, while a total improvement of 55.29% is noticed in the register banks. This gives an additional 7% over the CLK_GATED design and is probably due to the reduced fanout load at the input registers.

The "CLK_GATED_OP_ISOL_OPT" architecture

The purpose of this experiment is to optimize the overlapping isolation effect of clock gating and isolation latches present in the previous design. It was shown that clock gating effectively isolates the mult_hh and mult_ll functional units. Thus the particular isolation latches can be removed. By doing so an additional 1% improvement in CAU is achieved due to the reduced isolation logic power overhead.

Overall this implementation yields the highest total power improvement of 69.7%.

The "DECOUPLED" architecture

Isolate	**Condition**	**Depth**
mult_hh	opcode(3)	t_L
mult_hl	opcode(3)	t_L
mult_lh	opcode(3)	t_L
mult_ll	opcode(3)	t_L
mult_16_sf_H	opcode(2)	t_L
mult_16_sf_L	opcode(1)	t_L

Table 3.9: Relative power improvement (%)

At the previous chapter it is mentioned that resource sharing can damage data correlations which directly translates to higher switching activity. Based on that, the next design focuses on evaluating the scenario where separate resources are allocated for every single instruction. The idea is similar to the one illustrated in figure 3.2, but at a lower level. In that respect, two more 16x16 bit multipliers are instantiated, each with its own isolation logic, to perform the *MPF* and *MSF* instructions. The extra resources are grouped to form the Signed Fractional Arithmetic Unit (SFAU). The new isolation conditions are summed in table 3.9.

Component	**Improvement(%)**
REG_BANK	**19.85**
CAU TOTAL	**79.70**
adder	79.70
subtractor	80.09
mult_hh	80.72
mult_hl	80.57
mult_lh	80.57
mult_ll	80.67
Isolation logic	-1.05
SFAU TOTAL	**-13.37**
mult_16_sf_H	-4.48
mult_16_sf_L	-8.03
Isolation logic	-0.83

Table 3.10: Relative power improvement in DECOUPLE(%)

It can be seen that the delay in the isolation logic is only that of the latch and hence minimum.

The power dissipation in every component is organized in table 3.10. The overall power distribution and absolute values are identical to those achieved with the OP_ISOL architecture. The 13% power improvement in CAU is in whole offset by the power dissipated in the SFAU unit implementing the remaining instructions. In other words, there is not a single reason to justify the significant area overhead of two extra multipliers.

In conclusion, at the end of this experiment pure operand isolation yields an identical power saving of 66% in CAU compared to the decoupled design. Power dissipation in the REG_BANK can be halved, if clock gating is applied. However, it may not always be possible to find clock gating conditions, as the input registers in an architecture like the one in figure 3.2, can only be inactive during NOP instructions; overall, they are not expected to exceed 13% of the total instruction count [24]. In this percentage load-save instructions are also accounted for. Although they do not utilize the functional units, fetched operands will have to travel though the execution stage and should not invoke new computations.

3.6 Results

Operand isolation alone has resulted in power saving of up to 60% and is thus recommended as a dynamic power management wherever applicable. When clock gating is an option, it should be preferred over operand isolation; this may however lead to a more complicated architecture and additional design effort. Operand isolation finds application, in places where clock gating does not.

Clock gating is fully automated in SYNOPSYS Design Compiler. It is well integrated in the design flow and poses no side effects such as testability issues, and it is hence safe to use. Operand isolation is partly automated and not designed to work in terms with clock gating. At this point, manual latch based operand isolation was used. The power overhead of the isolation logic did not exceed 2% (four 32-bit latches plus control) of the total power dissipation. However, the power overhead can be significant for small blocks.

Gate operand isolation was applied at the adder and subtractor blocks in the CLK_GATED_OP_ISOL architecture. Although power consumption in both blocks was reduced, it was offset at its whole by the power dissipated in the isolation gates. The degree of utilization of the isolation circuitry is dependent both on the data and the instruction mix and has a great impact on the relative overhead. Thus, in cases where the isolation conditions have high static logic zero probability, operand isolation may also be affordable for add/subtract modules, as isolation logic consumes power only when it switches.

To limit the area overhead of the isolation logic, the input register set could be split into a shared master set of latches followed by a fork of slave latches serving in the same time as isolation latches. In that case both area and timing overheads are minimized. Power consumption is still to be calculated.

A final question arises: if a set of isolation latches following a shared input register set is justified, what would the power consumption be, if a different set of clock gated input registers was provided for each functional unit. Such an architecture has an essence of the one proposed in [44], where dedicated, specialized cores operate on demand by a central control processor.

Before CAD tools become power aware and dynamic power management is fully automated, the price for low power will be a more structural design style and customized power management circuitry. Design effort can be held affordable by designing with power management in mind, keeping the architecture simple and the operations mutually exclusive. Such a design would seize the most out of the use of isolation circuitry by very closely preventing unnecessary switching activity.

Chapter 4

Efficient Operand Isolation

In the previous chapter, it was concluded that latch based operand isolation yields considerable power saving in datapath designs, despite the significant area overhead. It was also outlined that operand isolation offers greater flexibility than clock gating, as it can be applied selectively on parts of a design. However, due to the small area, timing and power overhead, clock gating should always be preferred over operand isolation.

The purpose of this chapter is, on the ground of the above arguments, to explore alternative architectures that overcome the above limitations or merge the benefits of the two techniques. The next section describes the simulation and testbench environment. The standard latch-based operand isolation scheme, described in the subsequent section, will serve as the reference point for comparisons. Two alternative architectures are proposed in sections 4.3 and 4.4: a) the master-slave latch-based and b) the fine clock gating architecture. In the final section, the findings and performance figures are analyzed.

4.1 Simulation Environment

To create a common evaluation frame, the three different isolation architectures are applied to the complex arithmetic unit described in the previous chapter. In this experiment, the architecture of the top level design is maintained. However, the isolation logic has been moved to the block containing the I/O registers to allow for comprehensive, consistent and easy to compare power estimates. Figure 4.1 illustrates the simulation environment.

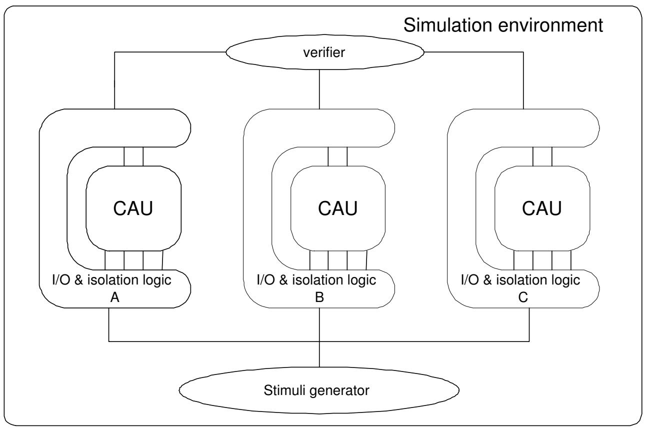


Figure 4.1: Simulation environment for the isolation architectures

Three instances of the complex arithmetic unit, each surrounded by a register block implementing the proposed isolation architectures, are instantiated. All three architectures have exactly the same functionality, and are exposed to the same input patterns. Hence they are expected to dissipate the same amount of power.

Within the register component four distinct parts are identified:

- Sequential context
  Accounts both for the isolation latches and the input registers and is expressed in number of equivalent latches (L), where a single bit register(R) counts for two latches (master-slave implementation).
- Control context
  Expresses the random control logic for the isolation latches in terms of logic primes (Pr) and the clock gating circuitry (Synopsys integrated cells). The latter is expressed in the number of disjoint register groups with different enabling conditions(CG).
- Interconnection context (see figures 4.2, 4.3 and 4.4)
  It quantifies the amount of wiring in number of separate 32-bit input (IB) and output buses (OB). Input buses are those before the input registers and output buses those after them and the isolation latches. As the load capacitance may vary, discrepancies in the wire-load model are noticed.
- Slack degradation
  This term expresses the depth of logic inserted on the timing paths both by clock gating (CGL) and isolation circuitry (OIL).

In the last section of this chapter all three architectures are compared based on the above mentioned factors.

4.2 Latch-Based Operand Isolation

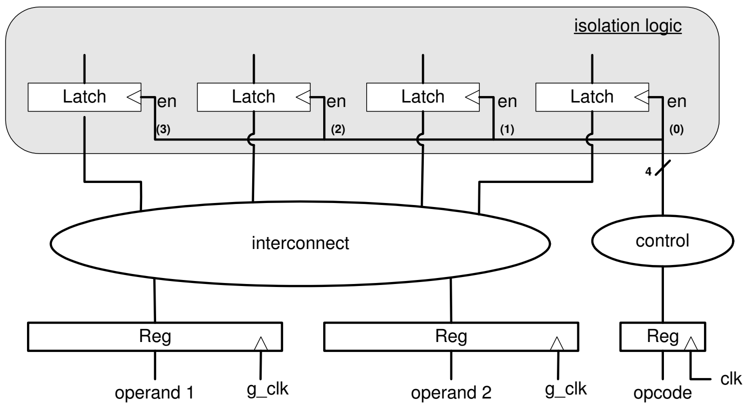


Figure 4.2: Latch-based operand isolation

As described in section 2.3, isolation circuitry, in that case level sensitive latches, is inserted at the inputs of all resources hooked on the same input buses, effectively blocking switching activity from being propagated down the different logical paths. In figure 4.2, isolation logic consists of the latches in the gray shaded area and the random logic annotated as *control.* The characterization of latch based operand isolation for the CAU is shown in table 4.1

4.3 Master-Slave Latch Operand Isolation

The high area overhead of latch-based operand isolation can be compensated by reducing the input registers into active low transparent latches. Combined together with the active

Category	Result
Sequential context	$64 \cdot R + 128 \cdot L \simeq 256 \cdot L$
Control context	$6 \cdot Pr + 2 \cdot CG$
Interconnection	$2 \cdot IB + 8 \cdot OB$
Timing	CGL=(AND + latch + register) OIL=(ANDOR + latch)

Table 4.1: Characterization of latch-based operand isolation

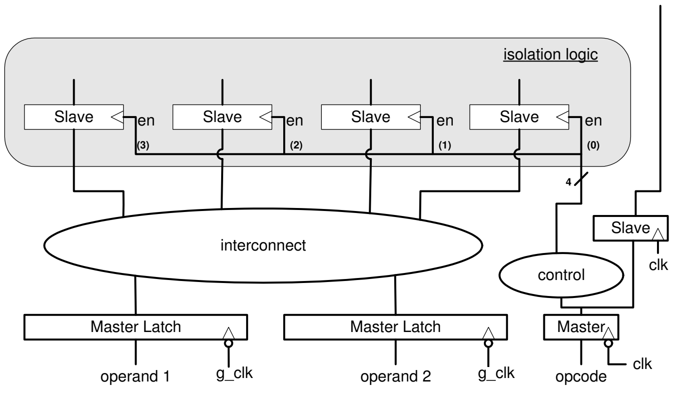


Figure 4.3: Master-slave latch-based operand isolation

high isolation latches, both the register and isolation functionality is provided. The concept is illustrated in figure 4.3 and the characterization in table 4.2.

Category	Result
Sequential context	$64 \cdot L + 128 \cdot L \simeq 196 \cdot L$
Control context	$7 \cdot Pr$
Interconnection	$2 \cdot IB + 8 \cdot OB$
Timing	CGL=(AND + latch) OIL=(ANDOR + latch)

Table 4.2: Characterization of master-slave operand isolation

It can be seen that the sequential and control context and timing are improved. From the CAD tool viewpoint, the VHDL compiler does not support constructs for master-slave latches and a special attribute[1] needs to be set to instruct the synthesizer of the existence of a master-slave latch. The technology library available does not include a master-slave latch and the custom made one has degraded performance compared to the edge-triggered flip-flop in all respects: area, timing and power. Timing analysis of circuits containing transparent latches may be problematic and manual inspection for timing violations is found necessary, as they may be invalid. Finally, the automatic clock gating facility is available only for edge-triggered flip-flops; thus manual clock gating has to be applied.

4.4 Fine Clock Gating Operand Isolation

To avoid the complications of using master-slave latches, the use of separate input registers for each functional unit is proposed as shown in figure 4.4, while the results are gathered in table 4.3. It can be seen that the equivalent sequential context is similar to that of the default latch-based operand isolation. The random control context is reduced at the expense

[1] *clock_on_also* see [51]

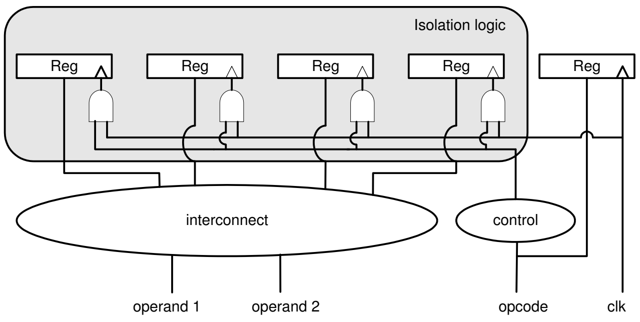


Figure 4.4: Proposal of minimum slack degradation operand isolation scheme

of two extra clock gating components imposed by the per functional unit application of clock gating. Overall, the control context is slightly increased. However, timing appears improved as the isolation functionality is pushed to the input registers and no logic is inserted on the critical paths. Interconnection as can be seen in figure 4.4, is slightly reduced compared to latch-based isolation and double as much compared to the non isolated design. Wire-loads are also reduced because of the introduction of dedicated (non-shared) buses. Last, but not least, aggressive clock gating can actually reduce the total switching activity in the register block with a direct positive effect on power dissipation. More on power consumption follows in the next section.

Category	Result
Sequential context	$128 \cdot R \simeq 256 \cdot L$
Control context	$2 \cdot Pr + 4 \cdot CG$
Interconnection	$8 \cdot IB$, OB = 0
Timing	CGL=(OR + AND + latch) OIL=0

Table 4.3: Characterization of fine clock gating operand isolation

4.5 Results

The performance metrics from the above experiment are summarized in table 4.4. It can be seen that both architectures proposed (master-slave and fine clock gating) outperform the default latch-based architecture in all respects. The area improvement in master-slave

Arch.	Area Comb	(LU^2) Seq	Tot. area (% impr)	Timing in ns	Timing (% impr)	Power (um)	Power (% impr)
L-B	270	31212	n/a	1.0	n/a	0.413	n/a
M-S	243	26505	15	0.74	26	0.328	20.5
F-CG	234	24183	22.5	0.36	64	0.326	21

Table 4.4: Comparison of isolation architectures

architecture is due to the replacement of edge-triggered registers by active low transparent latches implementing the master functionality. Timing is improved for the same reason. In the latch-based architecture, the timing arc from the inputs of the pipeline register and the functional units includes the clock to output delay of both the register and the isolation latch, plus the control logic. In the master slave architecture, the clock to output delay of the pipeline register is eliminated. Power is reduced due to the inherent lower power dissipation of the latch compared to a flip-flop.

The fine clock gating architecture, although it contains the same sequential context as the latch-based and higher than the master-slave, appears to have the lowest area of all. This is due to the fact that an edge-triggered flip-flop actually occupies less area than two latches in master-slave configuration. The impact of the isolation logic on timing is reduced to the absolute minimum, that of the propagation delay in the pipeline register. As it can be seen in figure 4.4, even the control logic has been moved before the pipeline register. Power consumption is almost identical to that of the master-slave implementation. Although latches dissipate less power than flip-flops, aggressive clock gating in the final architecture has allowed the reduction of switching activity to the absolutely necessary, and hence equivalent power. Table 4.5 illustrates the maximum switching activities in the register part for all available instructions and architectures. For all instructions, fine clock gating yields less switching activity. The same counts for the master-slave architecture compared to the latch-based, with one exception; the MCX instruction.

Instruction	L-B	M-S	F-CG
MCX	$64 \cdot R + 128 \cdot L$	$192 \cdot L$	$128 \cdot R$
MPF	$64 \cdot R + 64 \cdot L$	$128 \cdot L$	$64 \cdot R$
MSF	$64 \cdot R + 32 \cdot L$	$96 \cdot L$	$32 \cdot R$
NOP	0	0	0

Table 4.5: Switching activity in the register block

In conclusion, aggressive clock gating together with dedicated input registers is the most efficient method to apply operand isolation. For the specific experiment and only in the register part, a reduction of 22% in area, 64% in timing and 21% (5.5% overall) in power consumption are achieved. On top of that, the method is latch-free, hence it poses no timing analysis difficulties and testability problems, and it is easy to implement and integrate in the RT level design process.

Part II

Arithmetic in a Synthesis-Based Design Flow

Chapter 5

Arithmetic at the RT Level

Arithmetic components define the timing and power performance of datapath designs as they are usually forming the critical path and are responsible for a major part of the total power consumption. Several algorithms have been proposed and implemented covering a large range of requirements in terms of area, timing and power. It is important to note, that the most efficient is not always the preferred one, as it may result in a design overkill and increased cost.

In RTL HDL code, arithmetic components are usually inferred by arithmetic operators such $(\times, +, -)$ which are implementation independent. Operators are then interpreted to arithmetic primitives by the HDL compiler. The synthesis engine then, based on the constraints set, performs module selection for the arithmetic primitives from a vendor specific proprietary IP[1] component library[2]. Finally, the technology independent implementations are mapped to the specific technology library[3] used.

The purpose of this chapter is to shed light in the steps described above and the handles available to the RTL designer to interfere in this process.

5.1 A Review of Arithmetic Components

Addition and multiplication are the most commonly used arithmetic operations. Multiplication itself is formed as a sequence of shift and add operations of the partial products. Under this scope, all arithmetic operations can be decomposed to a network of 1-bit adders. Thus it is very important that high performance half- and full-adder cells are available.

5.1.1 Addition

2-Operand Addition

Arithmetic operators are binary, meaning that they perform on two operands. To add more than two operands more binary operators need to be used resulting in a chained or tree-like implementation that suffers from long critical paths. A more efficient implementation of multi-operand addition is discussed in the next paragraph.

What slows down addition is the propagation of carries. The adder architectures discussed below trade off area for performance and they all target minimization of the carry chain. In [11], the authors have estimated, simulated and physically measured the performance of different types of adders and the results are summarized in table 5.1.

[1]Intellectual Property
[2]the Design Ware Component Library from SYNOPSYS
[3]the 1.8V 0.25um CORELIB library from STM

Adder Type	Area (gates)	Delay (ns)	Power (W)
Ripple carry	144	54.27	1.17
Const. width carry skip	156	28.38	1.09
Var. width carry skip	170	21.84	1.26
Carry lookahead	200	17.13	1.71
Carry Select	284	19.56	2.16
Conditional Sum	368	20.05	3.04

Table 5.1: Area, timing and switching performance of 32-bit adders

The input length of 32 bits has been selected as a representative one for most applications. The authors, have concluded that absolute results from different methods to estimate performance may vary, yet they are qualitatively correct. The technology used was an obsolete for today 2um MOSIS[4]. All adders have been simulated for the same frequency of 10MHz which is 2 times smaller than the frequency of the slowest adder.

The carry-lookahead adder stands out as the fastest adder with an average area and power performance and it is hence a good compromise for high-speed applications. Though at its pure form, the high fanin and fanout requirements make it prohibitive for large adders. Alternative variations, such as the ripple carry lookahead and the super-block ripple carry lookahead adders, are suitable for average and large widths [43]. The ripple carry adder has the worst performance of all adders and the only reason to be preferred over a carry skip adder is its highly regular structure. For average speed constraints, the variable block width carry skip adder is a fairly good choice. Finally, the carry select and conditional sum adders have a good performance over all widths at high cost, though with high power consumption.

Multi-Operand Addition (> 2)

To avoid chaining of adders to calculate the sum of multiple operands, two methods are proposed:

- Adder arrays
- Adder trees

Adder arrays are constructed as a linear arrangement of either carry-propagate (CPA) or carry-save adders (CSA). In the latter case, a carry propagate adder is used to merge the carry and sum vectors. The fastest implementation is achieved by an array of CSAs followed by a fast CPA. The performance of an array of CSAs with an RCA or a fast final CPA is given by formulas 5.1 and 5.2, respectively, where m is the number of operands and n the width.

$$T = O(m + n) \tag{5.1}$$

$$T = O(m + log(n)) \tag{5.2}$$

Array adders have a more regular structure and lower interconnection, but lower performance when compared to adder trees. The performance of a tree adder is given by equation 5.3.

$$T = O(log(m) + log(n)) \tag{5.3}$$

Adder trees are constructed by a tree arrangement of compressors (see figure 5.1left) followed again by a carry propagate adder. In this way carry propagation is only performed once and postponed until after the tree. Effectively, arithmetic is performed in carry-save format

[4]Metal Oxide Semiconductor Implementation System

and the final carry-propagate can be perceived as a conversion layer between the carry-save and the standard 2's complement number representation.

An adder tree made of full-adders is commonly known as a wallace tree. A full-adder is effectively a (3,2) compressor that encodes three input bits to two. A (4,2) compressor and an optimized version of it are illustrated in figure 5.1. A (4,2) compressor achieves a higher compression rate and timing performance for the same area. For this reason it should be preferred for the construction of high fanin trees or the design of larger compressors (e.g. (8,2)).

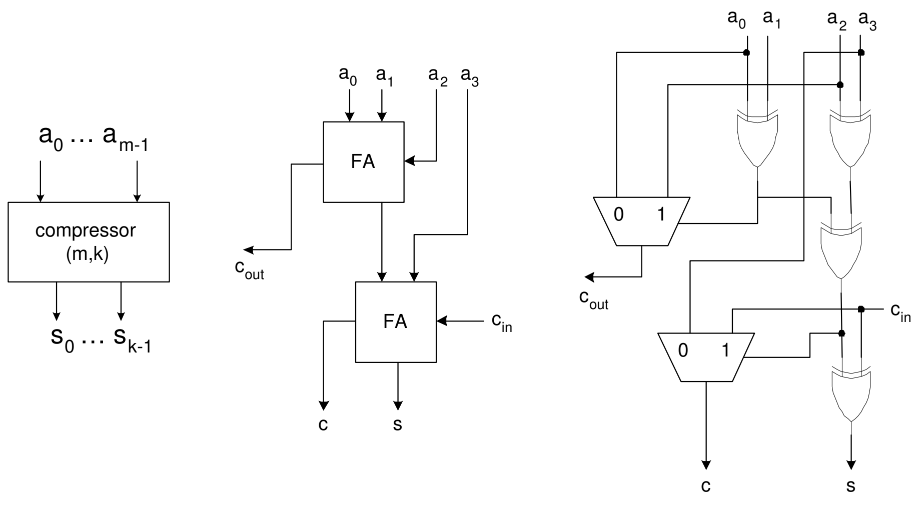


Figure 5.1: Implementation of a (4,2) compressor [61]

5.1.2 Multiplication

A multiplier is in practice a multi-operand adder with the partial products as input operands, which are created on the fly by the partial product generator, as depicted in figure 5.2. Thus, an array multiplier consists of a partial product generator and an adder array. Replacing the adder array with a wallace adder tree, a wallace multiplier is created. The Booth encoding can be used to reduce the number of partial products leading to smaller delay, reduced area in the wallace tree and potentially lower power dissipation due to the reduced adder tree.

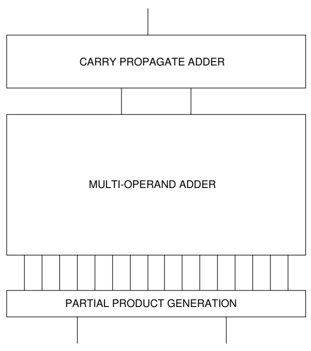


Figure 5.2: Architecture of a parallel multiplier

Similar to the array adders, array multipliers suffer from long delays and are only preferred for low to average performance applications. For average to high performance the wallace multiplier is commonly the best choice, while for ultra-high speed applications the booth recoded wallace multiplier is the default choice. Pipelining can then extend the operation speed further with increased latency.

Callaway and Swartzlander in [12] have estimated and simulated the delay and power dissipation of three commonly used multipliers:

- modified array multiplier
- Wallace tree multiplier
- Booth recoded wallace tree multiplier

Their results are given for a 2um CMOS technology. The delay is calculated as the number of full-adder cells on the critical path, hence it is technology independent and valid for smaller technologies. The power results are given in milliwatts, so they can only be used for qualitative comparisons. The delay and average power characterization for 8-, 16- and 32-bit multipliers are presented in tables 5.2 and 5.3, respectively.

Mult. Type	8bit	16bit	32bit
Modified array	7	15	31
Wallace	4	9	17
Radix-4 Booth	3	4	6

Table 5.2: Multiplier full-adder delays

Mult. Type	8bit	16bit	32bit
Modified array	14	47	275
Wallace	13	31	128
Radix-4 Booth	25	44	163

Table 5.3: Multiplier average power consumption (in mW)

Power figures are computed on the basis of the maximum frequency of the slowest multiplier.

An important observation is that for low to average performance requirements, an area optimized design is not necessarily a low power one. Although the wallace multiplier would be a design overkill when timing can be met by the modified array, it attains less power consumption. However, average power consumption is not a representative metric as it does not take into account the real operation conditions. The power-delay product should be used instead [12]; the results are quoted in table 5.4.

Mult. Type	8bit	16bit	32bit
Modified array	0.70	4.70	42.6
Wallace	0.65	3.12	19.2
Radix-4 Booth	1.26	4.37	24.5

Table 5.4: Multiplier power-delay product ($ns \times mW$))

The lower the power-delay product, the faster and more efficient the implementation. Based on that, the wallace multiplier represents the best choice when timing is not violated. Otherwise, the booth recoded version yields an acceptable performance.

As discussed in the multi-operand addition section, the wallace tree has an irregular layout and longer interconnects than the array adder. This may invalidate the above claims in future technologies, where wiring becomes the limiting factor. This argument is also supported by the foreseen increase importance of leakage power [27], which calls for minimum transistor implementations and may render area-performance trade-offs unacceptable. Pipelining may then be necessary, which is more straight forward in the modified array implementation.

5.2 Using Standard Cell Design-Ware Components

5.2.1 Synthesis-Based Design Flow

To fit the context of the previous section in a synthesis-based design flow, the synthesis process is briefly analyzed; a more detailed description can be found in [52]. Some definitions are given below:

- **Foundation library**
  is an IP library of parameterized, synthesizable, commonly used design components that comes with the SYNOPSYS synthesis suite. Each component may have several implementations.
- **Synthetic module**
  is a generic interface common to all implementations of a foundation component.
- **HDL operator** can be a built-in operator, a function or a language construct.
- **Synthetic operator**
  is an abstract link between an HDL operator and one or more synthetic modules capable to perform the required operation.

Figure 5.3 illustrates the definitions for the addition binary operator. The addition between two unsigned numbers, inferred by the overloaded "+" operator, is by definition mapped to the "ADD_UNS_OP" synthetic operator. This, in turn, is bound to three different Designware components: a combined adder/subtractor, an adder and an ALU, all of which are able to perform addition. This hierarchical level allows resource sharing optimizations. Each synthetic module can have several implementations, for instance an adder may be implemented with a ripple-carry, a carry-lookahead or a user-designed architecture.

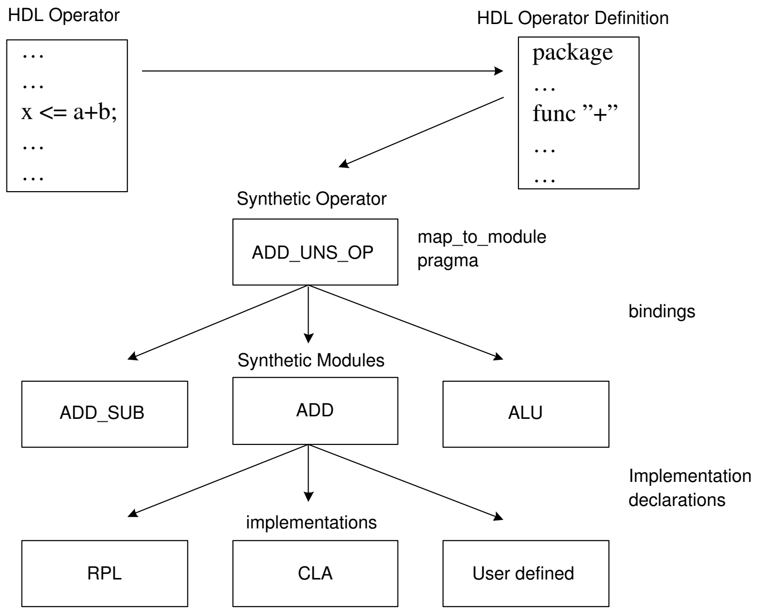


Figure 5.3: Design-Ware hierarchy

The synthesis process comprises three major steps: a) Analysis, b) Elaboration and c) Compilation. In the first step the code is checked for syntactical correctness. Subsequently, the RTL code is translated into a network[5] of synthetic operators.

Together with the design constraints, the outcome of the elaboration stage form the input to the compilation step. Based on the optimization objectives set (area and/or timing),

[5]SYNOPSYS proprietary GTECH format

synthetic operators are replaced by synthetic modules. Area and timing estimates for all implementations are created. Then, the one that maximizes the cost function driving the compilation process is selected and a first round of optimizations is performed. Finally, for timing-driven synthesis, critical paths are revisited, and Incremental Implementation Selection (IIS) takes place. That is, other implementations are tried out to improve overall cost and performance. The options for IIS are:

- *Area_only*
  The smallest implementation is selected.
- *Constraint_driven*
  Implementations that meet or do not worsen timing constraints are considered.
- *Use_fastest*
  The fastest implementation is selected. This is the default option.

So far, optimization has been power unaware and, as pointed out at the previous section, this may lead to suboptimal power performance. Logic inference, although it serves as the basis for automatic synthesis, it does not utilize the options offered by the foundation component library. Next, a study on the foundation library and ways to control the implementation selection is carried out under the scope of the discussion on arithmetic earlier in this chapter.

5.2.2 Handles to Design-Ware Components

VHDL and Design Compiler Directives

The VHDL Compiler directives are special VHDL comments that affect the actions of the VHDL and Design Compiler. They begin just as a regular comment, but they are identified by the *pragma* or *synopsys* keywords. The remaining text is treated as a directive. An example is given in figure 2.6 for pragma based operand isolation.

Similar to the VHDL attributes, the *SYNOPSYS.ATTRIBUTES* package[6] defines synthesis attributes for the VHDL compiler. Each *attribute* describes a specific property of a design object and works as a hint for the synthesis process. A comprehensive list of all available directives and relevant explanations can be found in chapter 10 of [53].

Inference

Inference is the task performed by the HDL compiler during the elaboration stage to map the HDL operators to synthetic operators, by the use of the *map_to_operator* “pragma”. It is facilitated by pre-compiled packages that contain the definitions of the operators and recognizable language constructs; hence the existence of recommended coding styles. The built-in VHDL operators that result in instantiation of Design-ware arithmetic components are given in table 5.5.

Type	Operators
Relational	= /= < <= > >=
Adding	+ - &
Unary	+ -
Multiplying	∗ / mod rem
Miscellaneous	∗∗ abs not

Table 5.5: Built-in VHDL operators

If the desired component and implementation for an HDL operator is known in advance, it can be annotated already in the RTL code as depicted in figure 5.4.

[6] PATH: $SYNOPSYS/packages/synopsys/src

```
library SYNOPSYS;
use SYNOPSYS.ATTRIBUTES.all

entity impl_sel(
    ...
end impl_sel;

architecture test of impl_sel is
begin
    process(A, B)
        variable result : ... ;
        constant  r0 : resource := 0;
        attribute map_to_module  of r0 : constant is "DW01_add";
        attribute implementation of r0 : constant is "cla";
        attribute ops            of r0 : constant is "a1";
    begin
        ...
        result := A + B; -- pragma label a1;
    end process;
end test;
```

Figure 5.4: Implementation selection in RTL code

The *label* directive is used to assign a label to the HDL operator. This label will also be used when the synthesized design is written to a file, hence it is advisable that a comprehensive label is annotated to every operator. A resource is created and connected to the operator through the "*ops*" attribute. The *map_to_module* attribute is used to force this resource to be implemented by the specified synthetic module. Finally, the *implementation* attribute selects the desired architecture.

Instantiation

VHDL supports instantiation of components and sub-blocks to facilitate hierarchical design [2]. In the same fashion, one can instantiate components from the Design-ware library, as from any other IP library. The "DW02" and "DW01" libraries include entity declarations for numerous arithmetic components [51] and pre-compiled packages with their configuration. The data sheet for the individual components can be find in the online documentation in the SYNOPSYS suite[7].

The Design-ware library contains a lot more components than those inferable from HDL operators. In that case or when a more structural and close to implementation description of a design is pursued, instantiation is required. Figure 5.5 provides an example to illustrate instantiation and implementation selection for a multiply-accumulate (MAC) unit. Important things to note are:

a) The Design-ware *library* and *use* clauses
b) The implementation selection
   Component *DW02_mac* is a synthetic module and it has a label (U1). Hence, unlike the example of figure 5.4, only the *implementation* attribute needs to be defined and set.
c) The use of the *translate_off/on* "pragma".
   The code in between is disregarded by Design Compiler.
d) The *sim_model* configuration connects the instance of a module with a behavioral model used for simulation purposes.

[7]path:$SYNOPSYS/doc/online/dw/dwf1/top.pdf

```
library DWARE, DW02;
use DWARE.DWpackages.all;
use DW02.DW02_components.all;

entity mac is port ( ...

end mac;

architecture mac_inst of mac is
    attribute implementation : STRING;
    attribute implementation of U1: label is "wall";
begin
    ...
    U1 : DW02_mac
        generic map (...)
        port map (
                    ...);
    ...
end mac_inst;

-- pragma translate_off
    library DW02;
    configuration sim_model of mac is
        for mac_inst
            for U1 : DW02_mac
                use configuration DW02.DW02_mac_cfg_sim;
            end for;
        end for;
    end sim_model;
-- pragma translate_on
```

Figure 5.5: Implementation selection for instantiated components

An alternative method for instantiation is through a function call. The instantiation and implementation selection of a specific component can be parameterized in a function call and be included in a package. In the packages where Design-ware components are configured, functions for the instantiation of some of the components are provided. For instance, the instantiation of the MAC block could be done by the *DWF_mac* function. Although it is possible to add the implementation selection code to the function definition, it can not be parameterized by a generic_port map. The reason is that "generics" of type "STRING" are not supported for synthesis. In that case the techniques described in the following paragraph can be applied.

Implementation selection in Design Compiler

Setting an implementation as early as at the RTL code, limits the optimization possibilities during synthesis. Handles to guide the implementation selection process are available both before and after the compilation stage.

Before compilation, it is possible to set the selected implementations of modules unavailable, if their performance, cost or power dissipation does not match the design requirements. This can be done by the "*dont_use*" directive after the elaboration stage. Figure 5.6 illustrates the commands to guide Design Compiler to exclude the slowest adder and multiplier implementations from being considered.

After any compilation run, the "*report_resources*" command can be used in the DC com-

```
dc_shell> analyze -f VHDL top_design.vhdl
dc_shell> elaborate top_design
dc_shell> dont_use standard/DW01_add/rpl
dc_shell> dont_use standard/DW01_add/cla
dc_shell> dont_use standard/DW02_mult/csa
dc_shell> compile
```

Figure 5.6: The use of the "*dont_use*" directive

```
dc_shell> analyze -f VHDL top_design.vhdl
dc_shell> elaborate top_design
dc_shell> compile
dc_shell> report_resources
dc_shell> set_implementation wall mult_9
dc_shell> compile
```

Figure 5.7: The use of the "*set_implemenation*" directive

mand window or shell to report the instantiated resources and their implementations during synthesis. If the exploration of timing, area or power reports indicate that suboptimal implementations have been selected, the "*set_implemantation*" command can be used. Suppose that the compilation step of the previous figure resulted in the selection of a booth encoded multiplier for the module labelled "mult_9". The commands in figure 5.7 shown how to manually select the wallace implementation. An extra compilation stage is needed for the changes to take effect.

5.3 Evaluating Synopsys Design-ware Library

Table 5.6 shows a selection of arithmetic components in the Design-ware library that can be used to built power efficient datapaths. The last two columns indicate whether the component is inferable from an HDL operator or needs to be instantiated. The non-inferable

Module	Description	Infer.	Inst.
DW01_add	Adder	✓	✓
DW01_addsub	Configurable Adder/Subtractor	✓	✓
DW01_sub	Subtractor	✓	✓
DW01_csa	Adder in carry-save format		✓
DW02_mult	Multiplier	✓	✓
DW02_multp	Partial product multiplier		✓
DW02_prod_sum1	Multiplier adder		✓
DW02_sum	Vector adder		✓
DW02_tree	Wallace tree compressor		✓

Table 5.6: Design-ware arithmetic modules

components are used in a number of experiments in the next chapter for the design of an efficient MAC unit and they will be discussed there.

5.3.1 Performance of Design-ware Arithmetic Components

The documentation of Design-ware components provides information on the usage and available implementations, as well as general design guidelines on the area and timing performance for different operand widths. The purpose of this paragraph is to provide a quantitative evaluation of the performance of addition and multiplication. The discussion is limited on 16bit multipliers and 32bit adders, as representative for datapath applications. The results are gathered in tables 5.7 and 5.8. Power figures have resulted from gate level simulation of 1000 random input vectors with a clock period of 16ns to accommodate the delay of the slowest carry-save multiplier.

Implementation	Abbr.	Area[8]	Timing (ns)	Power (mW)
Ripple carry	rpl	1	12.25	0.295
Ripple carry select	rpcs	1.45	8.70	0.544
Carry-lookahead	cla	1.86	6.48	0.444
fast cla	clf	2.24	3.03	0.589
Conditional sum	csm	4.03	2.97	0.410
Brent-Kung	bk	2.11	2.14	0.833

Table 5.7: Performance of 32bit DW adder implementation

Implementation	Abbr.	Area[9]	Timing (ns)	Power (mW)
Carry-save array	csa	1	15.53	2.918
Wallace tree	nbw	$\simeq 1$	6.67	2.905
Booth enc. nbw	wall	$\simeq 1$	7.37	2.578

Table 5.8: Performance of 16bit DW multiplier implementations

5.4 Summary

The previous part presented and evaluated the class of dynamic power optimization techniques. This part focuses on static techniques, which represent choices taken during the design stage. At the RT level, datapaths are built from arithmetic components provided from IP libraries and glue logic. To achieve an efficient design it is important that implementation selected matches the performance requirements of the application.

The chapter has presented and evaluated the available choices for adders and multipliers, as commonly used datapath components. Regarding adders, a wide range of implementations is available enabling close matching of design requirements and actual implementation. The ripple carry implementation due to its long delay is usually avoided. However, it makes a great candidate power-wise, when timing is not an issue. Design-ware library offers three different carry lookahead implementations, namely “cla”, “clf” and “bk”, with increasing performance and decreasing power-delay product, which sufficiently cover the application requirements. From the gate level simulation results of table 5.7, it can be seen, almost monotonically, that, the higher the performance, the higher the power consumption.

This in not the case though with multipliers, where the fastest implementation was actually found to be the less power dissipative. Some agreement is found between the power estimates of the Design-ware components and the actual measurements reported in the literature, which increases the confidence of the statement. Based on this assumption, the area and timing constrained nature of the optimization procedure of the synthesis engine, will fail to explore this part of the design space.

If actually true, a three-fold power improvement is expected:

a) Power consumption in the multiplier is reduced
b) The improved timing can facilitate power-performance trade-offs in other parts of the design
c) More freedom is allowed to the gate-level power optimization procedure

Finally, the chapter presents the methods to increase the implementation awareness of RTL HDL coding through VHDL directives for both inference and instantiation of arithmetic components.

Chapter 6

Experiment 2: An Efficient MAC Unit

This chapter describes the design and optimization of a Multiply-Accumulate Unit (MAC) based on the findings in the previous chapter. Efficiency is defined relatively to the performance of a benchmark circuit, as described in the following section. The two methods adopted to increase efficiency are elimination of carry propagation by the use of carry-save addition and pipelining.

6.1 The Benchmark MAC Unit

The high level model of the MAC unit after synthesis is shown in figure 6.1. The multiplier consists of a partial product multiplier that generates the result in carry-save format and a final carry-propagate adder, as the converter between the two different number representations. The final adder in figure 6.1 accumulates the new product to the sum of the previous clock cycle.

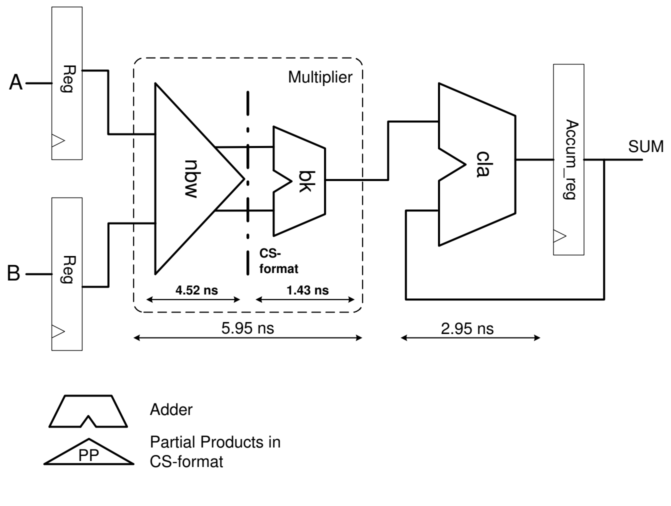


Figure 6.1: The benchmark MAC unit

The design has been synthesized for a 10ns clock period and has been simulated for 300 cycles with random input data to estimate power consumption. The results are given in table 6.1. Each block in figure 6.1 is annotated with its implementation (see section 5.3).

Based on the power analysis of Design-ware components in the previous chapter, little power could be saved by selecting the booth encoded wallace tree implementation for the multiplier and the conditional sum for the accumulator, without greatly affecting timing. Contrary to what was expected, a considerable increase in power consumption was noticed both in the multiplier and the accumulator. This can only mean that power estimates of stand alone components are not representative of the actual power consumption. Timing and area wise the results are close to the estimated values. For this reason, implementation selection for power reduction will not be further investigated.

Implementation	Area (units)	Timing (ns)	Power (mW)
nbw-cla	67977	9.32	7.266
wall-csm	90324	9.55	9.302

Table 6.1: Synthesis results for the MAC unit

With implementation selection out of the game, two other places susceptible to improvement are identified in the benchmark MAC unit:

a) The final carry-propagate adder in the multiplier
b) The uneven timing paths though the multiplier and the accumulator

As described in section 5.1.1, carry-save (CS) format can considerably improve timing by eliminating carry propagation in adder networks. Should the design be pipelined, the boundary between the multiplier and the accumulator would serve as an intuitive point for the insertion of the pipeline register. Still, due to the resulting unbalanced pipeline stages, the result would be far from optimal.

6.2 A Carry-save MAC Unit (CS-MAC)

A new MAC, named CS-MAC, is designed from non-inferrable Design-ware components that support carry-save arithmetic. The objective is to merge the final CPA adder in the multiplier with the accumulator.

The Multiplier-adder component "DW02_prod_sum1" accepts three input numbers A, B and C and performs the mathematical operation $SUM = A \times B + C$. The block diagram of the design after synthesis is presented in figure 6.2. The component was instantiated in the code, as explained in figure 5.5.

It can be seen that the chained addition has been merged into a three input wallace tree followed by a carry propagate adder (fast cla, in this case). The delay in the tree is that of a full adder. To reduce three n-bit operands into two, a single row of n disjoint full-adders is required. The results are given in table 6.2 and are expressed in % improvement over the benchmark design. Although the improvement seems to be marginal, it is consistent and it is expected to be more lucrative for bigger designs, in particular for addition with more than three operands. Such a case appears in the experiment carried out in the next chapter. There, wallace reduction trees are used for the addition of two and four operands, all in carry-save format.

Area (%)	Timing (%)	Power (%)
5.2	8.2	3.0

Table 6.2: Synthesis results for the CS-MAC unit relative to MAC

The wallace tree compressor that appears as part of "DW02_prod_sum1" is available as a parameterized component that accepts a user-defined number of inputs and reduces them in one carry-save output. A design that explicitly instantiates a partial product multiplier

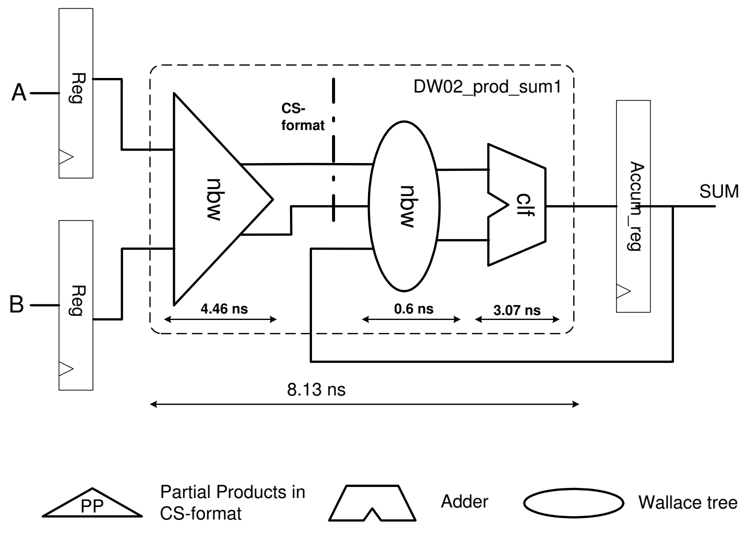


Figure 6.2: CS-MAC utilizing the DW02_prod_sum1 Design-ware component

("DW02_multp"), a wallace tree and a final CPA adder yielded similar results. Any discrepancies compared to the CS-MAC unit were due to the "clf" implementation of its final CPA selected by the tool.

6.3 Pipelining the MAC Unit

A pipeline register is inserted in the benchmark design between the multiplier and the accumulator (P-MAC). The clock period for the pipelined design is limited by the delay of the longest stage, in this case the multiply stage, at 5.42ns which amounts to 73% of the delay of the non-pipelined MAC unit. The two stages are clearly unbalanced.

By inserting the pipeline stage after the partial product multiplier, as shown in figure 6.3, the delay in the multiplication stage is reduced to 4.41ns or 59% of the delay of the non-pipelined design. The result is that the two stages are fairly balanced. Timing delays refer only to the combinational parts and clock-to-output and setup times for the registers are not taken into account. All results have been taken from timing reports generated by the synthesis tool.

Combinational area has again noticed a minor reduction due to the merging of the two addition operations. However, the carry-save representation, like any other redundant number system, has an increased demand in register count. This is because the carry-save format has twice the storage requirements of the 2's complement representation. For this reason, carry-save representation is preferably used within the limits of combinational blocks. In the MAC unit, however, if only the final result of the accumulation over long sequences of input data is needed, accumulation could be carried out in carry save format. An extra cycle would then be required to merge the two vectors. In that case, a performance close to that of the pipelined design would be achieved, by totally eliminating two 32bit carry propagations.

Power-wise, both the pipelined and the merged-pipelined designs consume more power than the original design. The power reduction by merging the two adders is offset by the power dissipated in the pipeline registers. The results for both designs are summarized in table 6.3.

The simulator used for power estimation does not account for glitch activity. This fact has two disadvantages: power consumption figures are optimistic and the merits of power

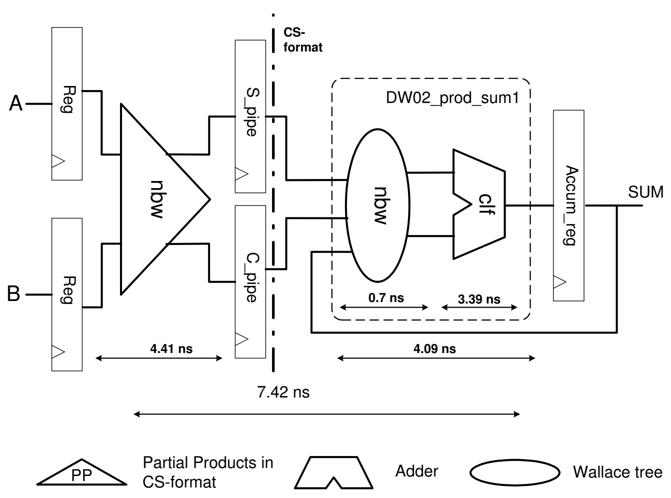


Figure 6.3: Balanced pipelined MAC unit (P-CS-MAC)

Design	Area (lib. units)		Timing	Avg. power (mW)	
	Comb.	Seq.	max (ns)	Comb.	Seq.
MAC	56727	11178	7.85	5,494	1,772
P-MAC	54477	15786	5.85	5,268	2,487
P-CS-MAC	51804	20610	4.85	5,023	3,162

Table 6.3: Synthesis results for the MAC unit

minimization techniques are pessimistic, especially for those techniques that aim to reduce spurious activity. On this ground and in accordance to the discussion in power-aware retiming (see section 2.5.2), the power overhead of the pipeline registers is expected to be offset by the reduced glitch power. In the non-pipelined design the output of the multiplier is expected to undergo several intermediate states before it reaches its final value, which will propagate to the accumulator. Finally, and specifically for the case of three operand addition, where the wallace tree consists of a single row of disjoint full-adders, due to the uniform propagation delay for all bit positions, the accumulator will see close to no spurious activity. However, for this to be true, full-adder cells with balanced carry and sum paths should be available. This may not be always the case, as the carry path is usually shorter.

6.3.1 Using a Public Available Library

Similar experiments were carried out with a publicly available library of synthesizable arithmetic components in VHDL [62]. The library offers numerous components, in line with the ones from the Design-ware library. The library is downloadable from the internet[1] and is presented to be compared against Design-ware components from Synopsys for a 0.35um technology and found to have similar performance and area. The experiments with the MAC unit only confirm the timing equivalence. Area was found to be as much as three times bigger than the results from designs based on Design-ware library. Power figures were also found to be higher by a factor ranging from 1.3 to 1.8, which is explainable considering the higher area. The results are given in table 6.4 and are normalized to those acquired from the corresponding designs presented in the previous sections.

Both the discrepancies in area and power were noticed in the multiplier. In the documentation of the library only a 10% of higher area compared to SYNOPSYS is quoted. Based on that the rest can only be attributed to the hierarchical nature of the design. The hierarchy

[1] *http://www.iis.ee.ethz.ch/ zimmi/arith_lib.html*

Design	Area	Timing	Power
MAC	2.74	$\simeq 1$	1.72
CS-MAC	2,38	1.1	1.34
P-MAC	2,28	1.25	1.32
P-CS-MAC	2,62	1.03	1.37

Table 6.4: Normalized performance of proprietary compared to Design-ware based designs

in the designs was preserved during synthesis to allow for hierarchical power estimation. The optimization process takes place locally within the boundaries of a hierarchical block. It is speculated, that flattening the design prior to optimization will bridge, but in no case vanish, the differences both in area and power. No further investigations have been made. However, after contacting the designer of the library, some explanation was given, but not verified. As the arithmetic components are optimally described, compilation should be performed with the "*-map_only*" flag to prevent logic optimizations. Otherwise, the logic optimization interferes with intended structure.

6.4 Results

The optimization techniques proposed in the previous chapter were used to optimize a multiply accumulate unit. It was found that carry-save arithmetic is an efficient alternative to the standard 2's complement arithmetic. By merging chained addition operations, area, timing and power in the benchamark MAC design were slightly improved and larger gains are expected from larger designs.

In the case of pipelining, carry-save format representation resulted in two balanced pipe stages and hence increased performance. However, carry-save representation suffers from increased storage requirements, which makes it a bad applicant for power sensitive retiming. The power dissipated in the pipeline registers of both the P-MAC and the P-CS-MAC offset the power reduction due to elimination of spurious switching activity in the accumulator. This experiment proves that power sensitive retiming should only be used to reduce spurious activity in large blocks (e.g. multipliers).

Thus, for maximum gains, carry-save arithmetic should not exceed sequential boundaries.

Part III

The Multi-Datatype Multiply-Accumulate Unit

Chapter 7

Experiment 3: A Multi-Datatype MAC Unit (MD-MAC)

It is common for both general purpose and application specific processors to perform different operations (logical, addition/subtraction, multiplication etc.), operate on multiple data types (e.g. fixed-point, floating point, integers and complex) and on different sizes (16-, 32bit). Functionality is customarily split in separate execution units based on the compatibility of the different operations, in a way that reflects the architecture of the processor's instruction set and their performance requirements. A common architecture is to have a separate floating-point unit, an arithmetic/logic unit and a multiply-accumulate.

In this chapter a Multi-Datatype Multiply-Accumulate unit (MD-MAC) that can operate on different data types is designed to serve as an evaluation platform for the power management techniques presented in the previous chapters. It is shown, that aggressive power management can keep the power overhead of sharing resources among different instructions to an acceptable level, with only a minor degradation in performance.

7.1 Design Specification

The MD-MAC design, as described below, merges the functionality of the CAU presented in chapter 3, of the MAC unit from chapter 6 as well as multiplication on 16- and 32bit integers. The specifications for the design are given in the following paragraphs

The Datatypes

The data types supported are:

- Complex signed fractional (CF)
- 16bit signed fractional (SF)
- A pair of 16bit signed fractional (PF)
- 16bit 2's complement integers (HI)
- 32bit 2's complement integers (FI)

The real and imaginary parts of the complex signed fractional numbers are 16bit wide and occupy the upper and lower part of a 32bit register, respectively. 16bit numbers, both fractional and integers, are placed in the lower part of the input register. The representation of the data types is illustrated in figure 7.1.

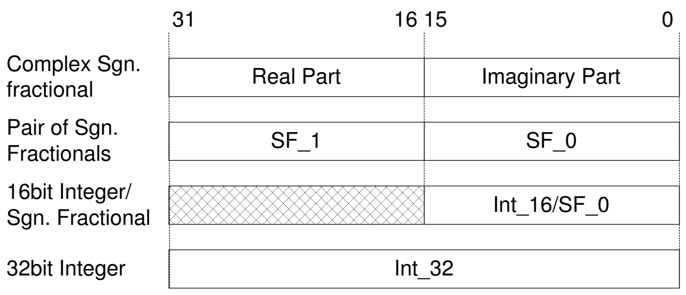


Figure 7.1: Supported data types in MD-MAC

With power optimization in mind already at the specification phase, an opportunity to reduce power arises. That is to clock gate the upper part of the input register, when it is used to store a 16bit value.

The Instruction Set Architecture

The implemented instructions together with their mnemonics and encoding are illustrated in table 7.1. The 1-hot encoded format for the operation code field, suggested in section 3.2, is also adopted here. The number next to the mnemonic refers to the bit position in the operation code field that indicates the instruction. The 1-hot encoding may become impractical for large instruction sets, in that case it is suggested that related instructions (e.g. those implemented in the same unit) are assigned codes with small hamming distance, to enable minimum delay in the decoding logic.

Instruction	**1-hot bit pos.**	**Comment**
NOP	0	No operation
Fractional class		
MSF	1	Multiplication of two SF numbers
MPF	2	Parallel multiplication of two PF numbers
MCX	3	Multiplication of 2 CF numbers
Integer class		
MHI	4	Multiplication of two HI numbers
MFI	5	Multiplication of two FI numbers
Multiply-accumulate class		
MAC	6	Multiply-Accumulate of two HI
ACC	7	MAC with propagation of the sum to the output
MCC	8	MAC with reset of the accumulator

Table 7.1: The MD-MAC instruction set

The MAC-instructions involve the use of an extended range accumulator register (34bit) that stores the intermediate sums. During a “MAC” instruction, the range of the accumulator is extended to 34bits and overflow occurs only if this is exceeded. The intermediate sum of products does not propagate to the output. This eliminates updating the output register, when only the final sum in a sequence of multiply-accumulate instructions is needed.

The “ACC” instruction is similar to “MAC” with the difference that the new sum is presented to the output of MD-MAC. An overflow occurs when the result exceeds the word size (32 bits).

The “MCC” instruction initiates a new multiply-accumulate sequence by resetting the accumulator.

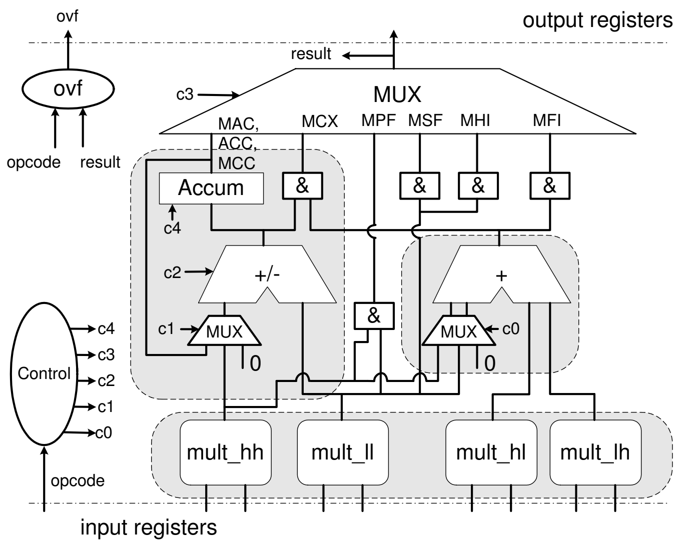


Figure 7.2: Block diagram of the MD-MAC unit

7.2 Block Level Design

The high level block diagram of the MD-MAC unit is presented in figure 7.2. As it can be seen, the basic architecture for the CAU and the allocation of the fractional instruction class (figure 3.3) are preserved. The diagram is partitioned in three gray-shaded conceptual blocks and the multiplexing functionality of the results at the output. The design will be analyzed at two levels: the functional and the circuit level. At the functional level, the allocation of instructions to the different resources and the incurred overhead are discussed in the following sections. The implementation level is the topic of section 7.3 .

7.2.1 Allocation of Instructions

This section describes the allocation of the integer and multiply accumulate instruction classes on the CAU platform.

The MHI, MAC, ACC, and MCC instructions

The multiplier annotated by "mult_ll" is feeded with the lower parts of the two input registers. As the "MHI" and the multiply-accumulate instructions use the same input operands, they are mapped on the same resource. By doing so, the utilization of the resource is increased and no additional interconnection is added at the input. At the output, two more inputs are added to the final multiplexor that selects the appropriate result. Even if these operations were mapped on separate units, the result selection at the output would not be avoided, but only shifted to another part of the design. This is because there is only a single result exiting the execution stage in a Single Instruction, Single Data (SISD) processor architecture, like the one considered.

Unlike the other data types considered, "FI" is different in size and the multiplication operation can not be provided by a single, already existing resource in the design. 32bit multiplication is considered to be a fairly demanding instruction, thus it appears only on application specific processors, as a separate unit. In general purpose processors, on the contrary, it is usually implemented as a software routine. The original CAU design already deploys four 16bit multipliers in order to provide single-cycle complex multiplication. Due to the high cost of multipliers, instantiation of a separate 32bit multiplier would be unacceptable, and resource sharing is considered.

The MFI instruction

In [22], the author proposes a new architecture for low power design of a multiplier. The full N-bit width multiplier is decomposed to smaller multipliers and their products serve as inputs to a weighed addition stage. The low power aspect of this architecture is neglected until later. At this point, only the suggested architecture is considered, as it fits nicely to the present circumstances. Suppose two binary numbers A and B that are partitioned to their upper and lower parts ($A = A_h \& A_l$ and $B = B_h \& B_l$), as for the case of complex multiplication. Formula 7.1 describes the behavior of the proposed architecture.

$$A \times B = (A_h \cdot B_h) \cdot 2^N + (A_h \cdot B_l + A_l \cdot B_h) \cdot 2^{N/2} + A_l \cdot B_l \tag{7.1}$$

The only obvious modification needed to map the multiplication of 32bit integers to the existing platform is the final addition of the sub-products. The implementation details of the 32bit multiplier are presented in section 7.3

7.2.2 Sharing Addition Functionality

In the previous section, the multiply-accumulate instructions were mapped to the "mult_ll" multiplier. Accumulation could either be performed on the subtractor following the multiplier to formulate the real part of the complex multiplication or on a separate resource. In the first case, a configurable adder/subtractor (add/sub) unit is required.

From experiment 1 in chapter 3, it was concluded, that operand isolation on small functional units, such as adders, is not justifiable. As the add/sub unit is placed on the fanout path of a highly active unit and operand isolation is not an option, by utilizing most of the switching activity it is exposed to, the useless switching power is minimized. For this reason accumulation is allocated on the add/sub unit.

Sharing this resource entails some control overhead. This can be multiplexing functionality on the inputs of the shared unit and control signals to configure both the multiplexors and the functional unit. The add/sub unit can be controlled by the MCX control line, as this is the only instruction that involves subtraction. For the multiply-accumulate operations, multiplexing functionality is required in front of the right input to selectively feed the value of the accumulator, a zero operand or the result of the "mult_hh" multiplier for a MAC/ACC, MCC and MCX instruction, respectively.

Based on the same principle, the adder used to formulate the imaginary part of the complex multiplication could be involved in the weighted addition of the sub-products for the "MFI" instruction, which in the worst case involves four 64bit operands. For this reason, the "MFI" instruction is expected to be the bottleneck in the design and the limiting factor on the performance achieved.

Result merging

Resource sharing incurs considerable interconnection overhead both at the input and the output of the shared resource. At the output side, in particular, in the case when the resource is shared over operations on data of different format, the output bus needs to be multiplicated. The small blocks in figure 7.2 annotated with "&" illustrate the point. Each is responsible for formulating the results for the individual instruction. This may include concatenation of signals or truncation in order to match the systems precision. If resources are not shared, the buses carrying the individual results still exist, but they do not stem from the same highly active point and switching can be controlled on a one-by-one basis with the techniques discussed in chapter 2.

In conclusion, the power dissipation on buses should not be neglected, when resource sharing is considered. A good metric would be the interconnect area, however, the wire load models in the technology library used do not include area information. Alternatively the switching activity of the output nets can be inspected to see the implications of resource sharing in the interconnection network.

7.3 Implementation Details

This section presents the implementation details of the design. Although the different parts of the design as shown in figure 7.2 are related to each other, the discussion will be organized based on that partitioning for manageability reasons.

7.3.1 Power and Delay Optimization

Power Management in the MD-MAC Unit

As was concluded from the results in part I, dynamic power management can significantly improve the power performance of a design. From the different schemes presented in chapter 5, it was shown that fine clock gating is preferred, as it performs equally or better than both the latched- and master-slave operand isolation schemes.

Implementing the "MFI" instruction on the CAU architecture is expected to yield lower performance compared to a non-shared architecture with a separate 32bit multiply unit. For this reason, any power optimization technique that further restrains performance should be rejected.

The input register part of the design and the clock gating conditions are presented in figure 7.3 and table 7.2, respectively. The output register is also clock gated, but it is only during the "NOP" instruction that it can retain its previous value.

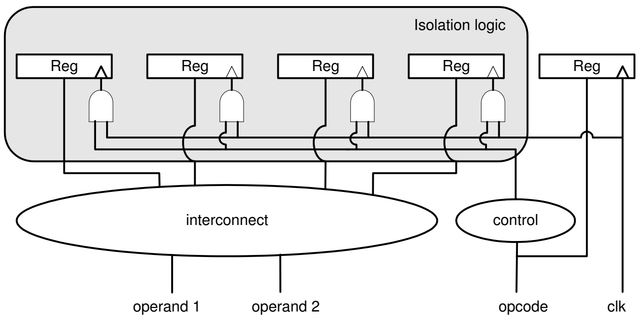


Figure 7.3: Operand isolation in the MD-MAC unit

It can be deducted from table 7.2, which is very sparsely populated, that many resources are not highly utilized. To obtain higher utilization more parallel instructions could be used, for instance dual multiply-accumulate instructions.

Carry-save arithmetic in the MD-MAC Unit

In chapter II, carry save arithmetic is presented as a means to improve performance in multi-operand addition. In the experiment of chapter 6, carry-save arithmetic was applied in the

Instruction	Reg_HH	Reg_HL	Reg_LH	Reg_LL
NOP				
MSF				✓
MPF	✓			✓
MCX	✓	✓	✓	✓
MHI				✓
MFI	✓	✓	✓	✓
MAC				✓
ACC				✓
MCC				✓

Table 7.2: Enabling conditions for the isolation logic in the MD-MAC unit

design of a MAC unit to reduce the delay and area in the multiply-accumulate operation by merging the final propagate adder of the multiplier with the accumulator. The delay was only slightly improved, due to the small number and size of the operands.

In the MD-MAC unit there are five instructions that involve multiply-accumulate functionality: "MCX", "MAC", "ACC", "MCC" and "MFI" and one ("MFI") that calls for the addition of four 64bit wide operands under tight timing constraints. In addition, as carry-save arithmetic actually doubles the number of operands, larger gains from multi-operand addition are awaited. For the above mentioned reasons, the application of carry-save arithmetic in the MD-MAC unit is expected to positively affect the efficiency of the design. The implementation details and design considerations are presented in the following section.

7.3.2 The Shared Add/Sub Functional Unit

Carry-save subtraction

2's complement subtraction is performed by adding to the minuend the 2's complement of the subtrahend. In practice, this is done by taking the 1's complement of the subtrahend and setting the carry-in input of the adder. Both the sum and the carry parts of a carry-save number are valid 2's complement numbers. Hence, in carry-save subtraction, both the sum and carry parts should be negated. This means that both parts should be inverted and the adder should be able to accept two carries.

A work-around applicable only in the case of 2 CS-operand subtraction is provided: the leftmost bit of the carry vector of a CS-number is by default zero; and so it is for minuend. This special condition, together with the associativity property of addition, can be used as a way to feed the second carry to the operation.

Reconfiguration logic

The required functionality for the individual instructions is shown in table 7.3.

Figure 7.4 shows the circuit diagram for the Add/Sub functional block.

By the use of carry-save arithmetic, the original two inputs of the Add/Sub unit, as it appears in the block diagram of figure 7.2, are doubled up. During the "MSF", "MPF", "MHI" and "MCC", which represent 50% of the instructions that utilize this resource, supposing they are equiprobable, the two leftmost inputs need to be set to zero. This is achieved by properly assigning the controlling inputs of the guarding "AND" gates. In this way, operand isolation of a part of the wallace tree is achieved for free. So, in this case, sharing of this resource has been successful, in the way that area, timing and power can be improved with minimum overhead, according to the findings in chapter 6. Adding operand isolation gates in front of

Instruction	Functionality
MSF	ll_s + ll_c
MPF	ll_s + ll_c
MCX	(hh_s + hh_c) -(ll_s + ll_c)
MHI	ll_s + ll_c
MAC	ACCUM + (ll_s + ll_c)
ACC	ACCUM + (ll_s + ll_c)
MCC	(ll_s + ll_c)

Table 7.3: Functionality of the shared Add/Sub unit

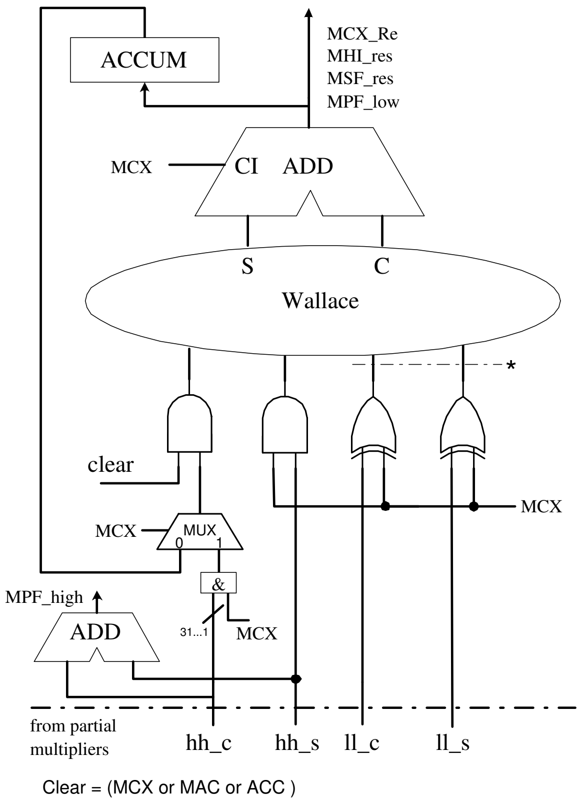


Figure 7.4: Circuit description for the Add/Sub functional block

the two other inputs, indicated in figure 7.4 by a dotted line and a star, would result in an overhead, as those inputs are isolated by the clock gated register in front of the ”mult_ll” unit.

As regards coding, the adder has been inferred, the wallace tree has been instantiated and the control and guarding logic has been described in a structural gate level.

Further optimizations to be applied

The brent-kung implementation for the final adder was selected automatically by the tool. This resource is not on the critical path and by inspection of the timing reports after synthesis, a 1.8ns positive slack was found. According to the performance of the Design-ware components from table 5.7, manual selection of the “fast-cla” implementation would result in 30% lower power dissipation in the adder, without violating the timing constraints.

A 4-input wallace tree is constructed by two rows of disjoint full-adder cells. By assigning the guarded inputs with higher static probability to the first row of adders and the more active inputs closer to the output, switching activity is minimized. As discussed in [45], data statistics can offer great help in guiding automatic power optimization algorithms. Although, it is difficult to estimate the savings of such low level optimizations and apply them at the RT level, information like that should not be neglected. An example of such an optimization provided as a hint to the RTL designer is found in [43] for the booth multiplier: *input "A" in the "wall" implementation is booth recoded, so in case of multiplication by a constant or multiplication of words of different size, assigning the constant or the smaller word to input "A", respectively, will result in a faster and smaller design.* Finally, if the (4,2) compressor of figure 5.1 was used, both area and timing would be improved.

It is obvious that gate level designs will result in more efficient implementations. One way to fight that from the RT level is the use of richer libraries providing more flexibility to the designer.

7.3.3 32bit Multiplication

Mapping 32bit multiplication on the CAU platform

The decomposed multiplier architecture in [22] is proposed for unsigned numbers, which are free from sign extension complications. In this case, both the upper and lower parts of the original input values can be considered as unsigned numbers and their multiplication is straight forward.

On the case of signed 2's complement representation, although the sign of the upper part is defined, the lower part does not have a sign, but it can always be considered as a positive number, as dictated by formula 7.2.

$$A = -2^{n-1} + \sum_{i=n/2}^{n-2} 2^i + \sum_{i=0}^{n/2-1} 2^i \tag{7.2}$$

Multiplication of a signed with an unsigned number requires their being sign extended by one bit prior to multiplication.

Based on these observations, several changes need to be made on the original setting. First of all, the "mult_ll" multiplier needs to be configured to operate on both signed and unsigned numbers. Two ways can be used to do that: inference (see figure 7.5) and instantiation.

```
if (sign = '1') then
    result := signed(A)*signed(A);
else
    result := unsigned(A)*unsigned(B);
end if;
```

Figure 7.5: Inferencing a signed/unsigned multiplier in VHDL

Both multiply operators in the code in figure 7.5, during elaboration, will be mapped to the same synthetic operator and during compilation they will be assigned to share the same module, resulting in the same effect as if a multiplier had been explicitly instantiated in the design. The "DW02_mult" module is parameterized both on the size of the operands and their representation. The "TC" input pin is used to indicate signed or unsigned operation. In this design multipliers have been instantiated.

The "mult_hl" and "mult_lh" from CAU have been replaced by two 17bit multipliers to be used on the sign extended inputs. Under all other instructions than "MFI", inputs are sign extended according to the rules for 2's complement representation. During an "MFI" instruction the upper parts are sign extended as 2's complement numbers, while the lower parts are extended by an extra zero. After multiplication the sign extension bits can be truncated. The "mult_hh" multiplier did not need to be modified. Figure 7.6 illustrates the implementation of the four multipliers.

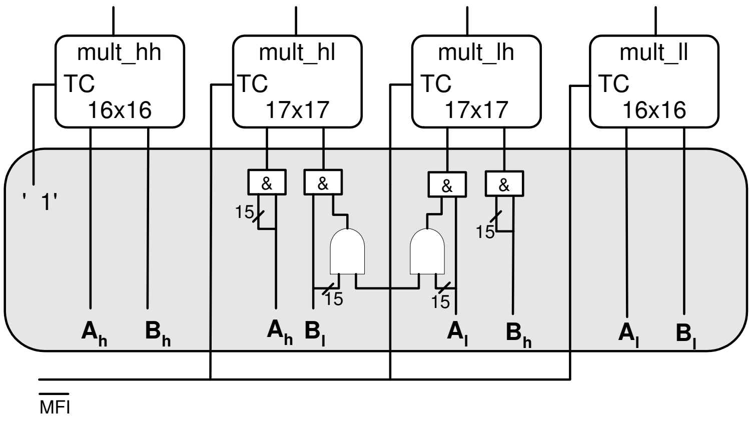


Figure 7.6: 32bit Multiplication on the CAU platform

Actual implementation

In order to accommodate carry-save arithmetic, multipliers are replaced by partial multipliers ("DW02_multp") presented in experiment 2 in chapter 6. They can only be instantiated in the code, but they share the same interface (port description) and implementations with the multiplier modules. Implementation selection was left to the tool. For average width sizes the non-booth encoded wallace architecture was chosen, as the one yielding the smallest and fastest circuits.

The sum and carry outputs of the partial multipliers are internally sign extended by two bits and can according to the Design-ware manual be truncated if not needed.

7.3.4 Weighted Addition of the Sub-products in MFI

Design considerations

Weighted addition means that the operands need to be properly aligned before they are added, as described by formula 7.1. Let "HH", "HL", "LH" and "LL" be the sub-products of the four 16bit multipliers; figure 7.7 illustrates their sign extension and aligning.

The lower 16bits of the final 64bit product are available and occupy the lowest part of the "LL" sub-product. The remaining 48 high order bits need to be calculated by adding the aligned sub-products from bit position 16 to 63. As the result of the multiplication of two 32bit operands is 64bits wide, sub-products need not be signed extended further and the carry-out at position 64 can be disregarded.

In order to reduce the delay, the carry-save multi-operand addition architecture is used. This choice has two side effects:

a) The number of operands to be added is doubled
b) The width of the addends is increased from 48 to 64

The second side effect is due to the carry that may be generated from the addition of the lowest 16bits of the sum and carry results of the "mult_ll" partial multiplier. As a result the full 64bit carry chain is added on the critical path. To cure this problem, a separate 16bit adder is used to generate the carry from those lower 16bits; it is illustrated in figure 7.8. As

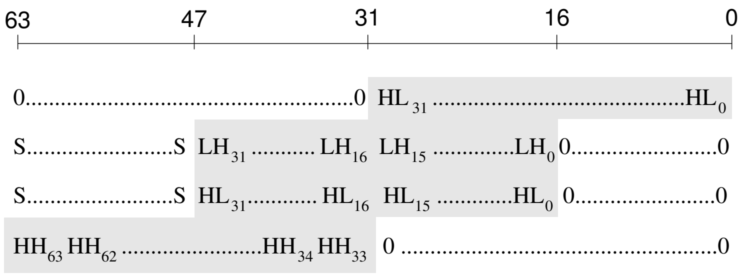

63
47
31
16
0
0......0 HL31 ......HL0
S......S LH31 ...... LH16 LH15 ......LH0 0......0
S......S HL31...... HL16 HL15 ......HL0 0......0
HH63 HH62 ......HH34 HH33 0 0

Figure 7.7: Weighted addition of the sub-products

this operation overlaps in time with the tree reduction operation, the carry will be available when needed.

Similar to the discussion on the "Add/Sub" unit, the question of whether to share the existing adder used for the calculation of the imaginary part of the "MCX" instruction or a separate unit reappears. For the same reasons, the existing unit is modified to accommodate the extra functionality. The width is extended from 32 to 48 and the number of operands doubled to 8.

Reconfiguration Logic

Figure 7.8 shows the modified circuit diagram.

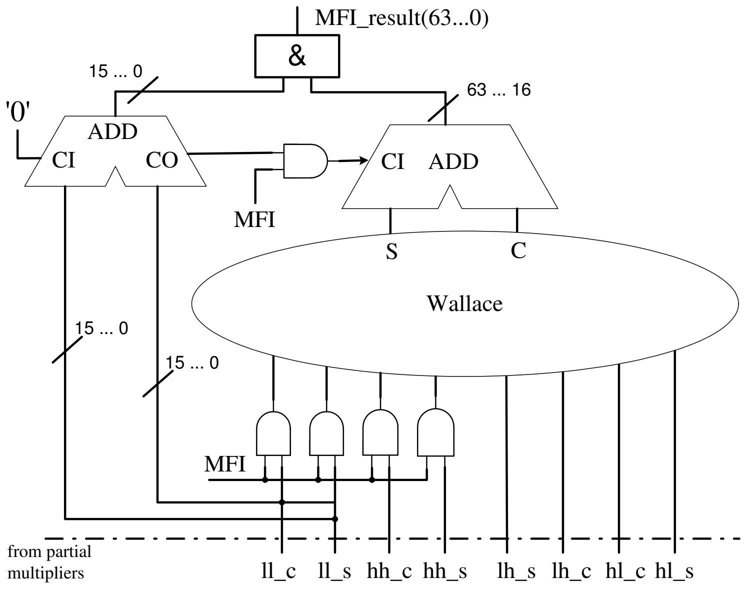


Figure 7.8: Circuit implementation of 64bit weighted addition

Logic "AND" gates control by the "MFI" signal line are used to set the not required operands to zero, effectively providing operand isolation services to the shared unit. The 16bit adder to the left is left unguarded, due to its insignificant power consumption. However, considering the high utilization of its fanin logic, gate-based operand isolation may actually pay off. This feature has not been investigated. Finally, the carry-out from the low 16 bits should only be propagated during an "MFI" instruction, hence the "AND" gate in front of the carry-in input of the weighted adder.

Actual implementation

The reduction tree is implemented as a wallace tree of depth four. The implementation chosen for the final carry-propagate adder is "brent-kung", which is the fastest, yet the most power dissipative. Any other implementation would result in lower performance.

The overhead of sign extension

It can be seen from figure 7.7 that the aligning of 2's complement operands results in extensive sign extension, which is redundant. One way to minimize sign extension, though not very common, is to use the signed magnitude number representation. If 2's complement representation is to be used, a technique to reduce the overhead of sign extension is described in [49]. By applying this technique the sign extension bits after the first are replaced by constants and full-adder cells in the reduction tree involved in the addition of sign bits can be simplified to half-adder cells or inverters. By doing so, area, timing and power dissipation are improved.

This optimization technique belongs at the gate level. However, it can be applied to the RT level in the following steps:

a) Apply sign extension as described in [49] at the inputs of the operator.
b) After the first compilation, the design hierarchy should be flattened to allow logic optimization to exceed the boundaries of modules during the second compilation.
c) An incremental compile may propagate the constants and in this way achieve the required effect.

The above procedure is a proposal and has not been verified to work.

7.3.5 Output Multiplexing Functionality

As described in chapter 2, setting appropriately the "don't-care" states of control signals can spare unnecessary switching activity. A better solution, however would be to completely eliminate "don't care" conditions. This applies to the design of the output multiplexing functionality.

A common way to infer non-priority multiplexors in VHDL is by the use of the VHDL "case construct". Because of the resolved nature of the IEEE standard logic vector and the fact that the number buses to be multiplexed is not a power of two, the "*when others =>*" branch of the case-statement would result in switching activity during a NOP instruction. To avoid that, multiplexing functionality was coded with gates, where a result is selected by a two-dimensional network of "AND-OR" gates controlled by the operation code field.

7.4 Results

The MD-MAC design was synthesized for a clock frequency of 100MHz and simulated for 800 cycles with random data in order to extract the nodal switching activities. The instruction mix for the simulation consists of equal distribution for all instructions. To give ground for comparisons a second design (SPLIT-MD-MAC) was implemented that includes a separate 32bit multiplier. Figure 7.9 presents a simplistic block diagram depicting high level organization of the two designs. By providing a multiplication co-processor in the SPLIT-MD-MAC, the overhead imposed by the weighted addition of the sub-products is spared. The MULT block has its own pair of isolation registers, thus it only consumes power during an "MFI", during which the "CORE" block is idle.

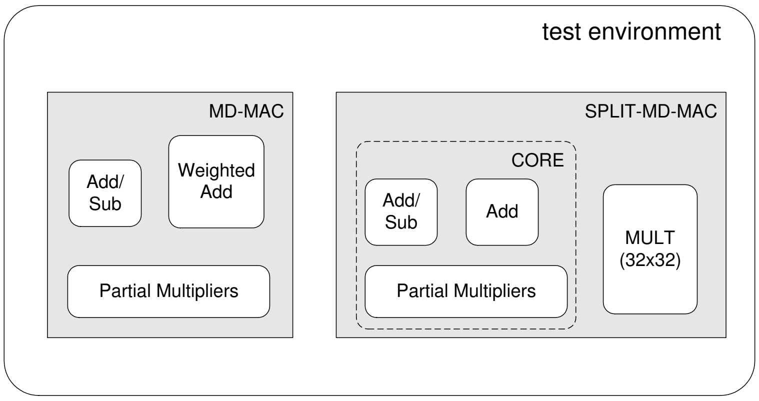


Figure 7.9: Block diagram of the benchmark (SPLIT-MD-MAC) and the MD-MAC designs

7.4.1 Area and Timing

Table 7.5 shows the combinational and sequential area of the two designs

Design	SPLIT-MD-MAC	MD-MAC	Diff. (%)
Comb. Area	406476	290512	28.5
Seq. Area	45297	38161	15.8
Total Area	451773	328698	27.2

Table 7.4: Area of the benchmark and test design

A total of 27% reduction in area was achieved in the resource shared implementation. The increase in the sequential area in the SPLIT-MD-MAC unit is due to the two 32bit isolation registers of the multiplication co-coprocessor.

Timing wise, the SPLIT-MD-MAC met marginally the timing constraint of 10ns. The critical path is defined by the 32bit multiplier. A booth encoded wallace tree followed by a "brent-kung" final adder were automatically selected by the tool for the multiplier's implementation. Implementation selection was based on the tight speed requirements.

The MD-MAC unit violated the timing constraints by 0.21ns, however its performance is very close to that of the benchmark design. As expected, the critical path is defined by the "MFI" instruction. A mutation of the MD-MAC design, named MD-MAC-NCS, which

Design	SPLIT-MD-MAC	MD-MAC	Diff %
Delay	9.94	10.21	-2.7

Table 7.5: Timing performance of the benchmark and test design

stands for non carry-save, was also implemented to evaluate the efficiency of the carry-save optimization. Partial multipliers are replaced by multipliers, though the wallace reduction tree in the weighted addition block is maintained to add the 4 aligned sub-products. The performance of this design was found to be 10.45ns. Area was increased by 2% and power by 3% compared to the MD-MAC design. Yet, once again the efficiency of the carry-save arithmetic optimization was confirmed.

7.4.2 Power Consumption

The power dissipation of the MD-MAC design was estimated to be only 8.93% higher than that of the benchmark design. Table 7.6 shows the power dissipation of the individual blocks in both designs and the normalized improvement, in order to identify the points that are responsible for the increase in the total power dissipation. It can be seen that the bottleneck is the weighted addition block, which has undergone the most drastic changes. The partial multiplier block in the SPLIT-MD-MAC column contains all multiplication functionality; that is, it accounts for both the partial multipliers block and the separate 32bit partial multiplier.

The close control on switching activity provided by the fine clock gating operand isolation method helped keep the power overhead from sharing resources to the minimum possible.

Block	**Power in mW**		Diff (%)	Norm. Diff. (%)
Add/Sub	**SPLIT-MD-MAC**	**MD-MAC**		
Input registers	2.912	2.264	22.3	3.0
Output registers	1.082	1.084	0.0	0.0
Add/Sub	1.60	1.779	-11.2	-1.3
Weighted Add	0.446	1.288	-184.3	-12.5
Partial multipliers	7.981	7.906	1.0	0.47
Glue Logic	0,927	0.693	33.3	1.4
Total:	15.248	16.664		-8.93

Table 7.6: Total power dissipation

Tables 7.7, 7.8 and 7.9 give the power consumption for each high level block, as shown in figure 7.2.

The MD-MAC-NCS design had a total of 12.5% increase in power dissipation compared to SPLIT-MD-MAC and 3% increase compared to the MD-MAC. The power saving from using carry save arithmetic lies in the same range as that from the MAC design in chapter 6, despite the higher expectations for larger designs.

Block	**Power in mW**	
Add/Sub	**SPLIT-MD-MAC**	**MD-MAC**
MPF_high adder	0.1	0.162
Wallace tree	0.686	0.743
CPA_adder	0.814	0.874
Total:	1.6	1.779

Table 7.7: Power dissipation in the Add/SUB block

Block	**Power in mW**	
Weighted addition	**SPLIT-MD-MAC**	**MD-MAC**
Wallace tree	0.106	0.658
CPA_adder	0.074	0.483
MFI_low_16 adder	0	0.147
Mult_32_CPA adder	0.266	0
Total:	0.446	1.288

Table 7.8: Power dissipation in the weighted addition block

Block	**Power in mW**	
Partial multipliers	**SPLIT-MD-MAC**	**MD-MAC**
mult_hh	1.126	1.62
mult_hl	0.541	1.241
mult_lh	0.528	1.219
mult_ll	3.516	3.826
mult_32	2.27	0
Total:	7.981	7.906

Table 7.9: Power dissipation in the partial multipliers block

Chapter 8

Conclusions

The aim of this thesis was to investigate the design of power efficient arithmetic circuits for application specific processors. The application domain of ASPs is specific in the sense that products have a limited time horizon. As large development costs can not be amortized over next generation products, a synthesis-based design flow is followed, in order to meet the tight performance and time-to-market constraints. Such a flow is characterized by RTL description of functionality and synthesis based on available IP libraries.

At this level, unlike the system and gate level, little flexibility is provided to the designer and the quality of the design is solely based on his/hers ingenuity. And this is the contribution of this work: to provide a study on the optimization techniques available at this level.

Another speciality of the application domain is that optimization is based on the power-delay performance metric, unlike the domains of ASICs and general purpose processors, where neither power nor performance are negotiable. Hence, in this case, power can be traded with performance and vice-versa. The approach followed in this work is to improve power with minimum impact on performance.

In this work, the Design Compiler automatic synthesis tool suite and the Design-ware IP library have been used as the synthesis platform. More specifically, the VSS simulator, the Design Compiler synthesis engine and the Power Compiler optimization tool have been used. Synthesized designs have been mapped on a 0.25um 1.8V CMOS technology from STMicroelectronics.

8.1 Optimization techniques

At this level, only dynamic power is of importance, which is a function of load capacitance and switching activity of the internal nodes of a design. At the RT level of description, only switching activity can be directly controlled, as it is dependent on the functionality. Capacitance first comes into play after the design has been synthesized and mapped onto technology dependent gates. Thus, discussion has been limited to the reduction of switching activity. Available techniques are classified into dynamic and static.

8.1.1 Switching Activity and Datapath Architecture

Two types of switching activity have been identified: useful and useless. Useful activity is the one that leads to the computation of a result, while useless is the activity that is not part of a computation. Useless switching activity can be either the result of bad design practice or inherent to the implementation. There is not much to be done about useful activity, as computation needs to take place. This could be better fought at the behavioral level, as for the case of the two alternative formulas that were presented in chapter 3 to calculate the result of a complex multiplication. Hence, the main effort is to limit the useless switching activity, which is very much dependent on the architecture of the design. Two kinds of architectures are identified: a convergent and a non-convergent. The former refers to architectures where most of the functionality is placed close to the input resulting in a triangle-shaped visualization of computations. The latter refers to the opposite case, where functionality stems from a point close to the input and branches out towards the primary outputs. It is easier to control switching activity close to the primary inputs by controlling the input registers. On the contrary, it is difficult to control switching activity of a high fanout point. For this reasons, a convergent architecture is expected to have more useful, than useless switching activity, contrary to a non-convergent, high-fanout one, and it should be preferred.

8.1.2 Dynamic Power Optimization Techniques

From the dynamic optimization techniques presented in chapter 2, only clock gating and operand isolation were found able to assist in reducing switching activity.

Clock gating

Clock gating has been confirmed as a powerful method to reduce area and power dissipated in sequential logic. 44% reduction in the power of the register bank in the "CLK_GATED" design from experiment 1 (chapter 3) was estimated when enabled registers were replaced with clock gated ones. What is important to note is that a clock gated register dissipates almost no power, contrary to a register that holds its previous value through a feedback multiplexor.

Operand isolation

Operand isolation was also proven to be an efficient power optimization technique as a means to block switching activity from functional units. Though, it comes with an overhead and should be used judiciously. Different to what was reported in the literature, the power saving from applying operand isolation to small blocks (32bit adder) is evened out by the power dissipated in the isolation logic itself. Gate-based operand isolation is found to be very sensitive to the workload and for this reason latch based isolation was used, despite the increased overhead. To give an estimate, the power dissipated by the isolation logic in the CAU unit was about the same as the power spared on the adder. For this reason, operand isolation is recommended only for large blocks, for instance multipliers. Furthermore, isolation logic imposes a timing overhead, as it is inserted on the critical path.

Fine clock gating

Merging the merits of both techniques, we propose the allocation of a separate input register for every functional unit, as the most efficient way to apply operand isolation. Compared to latch-based operand isolation, fine clock gating noticed superior performance, area and power. This scenario fits nicely with the convergent architecture of a datapath described before. The master-slave operand isolation scheme is proposed as an alternative that moderates the overhead of the pure latch-based operand isolation,

8.1.3 Static Power Optimization Techniques

Power sensitive retiming aims at reducing spurious switching activity at the outputs of functional units on the same path. Similar to operand isolation, noteworthy savings are expected only when applied to the inputs of high dissipative blocks or at the outputs of units with highly spurious outputs, such as comparators.

Implementation selection has been proposed as a way to enhance the power unaware selection of the implementation of modules, aiming at the utilization of the choices offered by the IP libraries. Experimental results did not rise up to verify the expected behavior.

Carry-save arithmetic for multi-operand addition was proven to improve both area, power and timing by merging structures of conventional binary adders.

Static optimization techniques, except carry-save arithmetic, failed to show promising results. The reason for that is that they are not really meant as RTL optimization techniques. Retiming is more amenable to the gate level, where it is expected to yield better results. Similarly, implementation selection is closer to the high level synthesis, where it is also expected to work better when supported by probabilistic models of components and tuned cost functions.

8.2 Limitations

Power estimation is an approximation of the actual behavior of a real circuit. According to SYNOPSYS, gate level simulations are fairly accurate, with fluctuations of 15-30% compared to transistor level simulations. Taking into account their data dependent nature, the results claimed above can only be confirmed for the specific data used. As discussed in the report, uniform probability data represent the best case for estimation of power savings. Real-life workloads have correlations that usually result in lower power dissipation. This means that the relative overhead of isolation logic is optimistic.

Regarding the spurious activity, it is not clear as to whether it is captured by the simulatior, resulting in pessimistic power savings.

In addition, the examples used have similar properties and by no means are they representative of all cases.

For the above mentioned reasons, the validity of the results obtained can not be trusted for all cases.

8.3 Future Work

First of all, experiments on larger and more realistic designs covering other classes should be carried out to evaluate the generality and the validity of the results claimed.

It is believed that the implementation selection power optimization method proposed has a good potential.

There is a low power aspect that should be followed up, inspired from the small overhead and comparable performance achieved by mapping the 32bit multiplication instruction on a 16bit platform. Together with the fine clock gating operand isolation method proposed, extensions to data segmented designs and dynamic size adaptation arithmetic can easily be imagined.

Prototyping could also be envisioned in order to verify the power merits.

It was discussed that there are only few RT level power optimization techniques. It is the writers opinion though, that those few that are around, they are not utilized fully. One way to extend their scope is by investigating architectures as discussed earlier. The fact that logic does not dissipate power (only a fraction) when idle could be used to limit useless switching activity by fine clock gating.

Bibliography

[1] Alidina, M., Monteneiro, J., Devadas, S., and Ghosh, A. Estimating dynamic power consumption of cmos circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 18(8), 1999.

[2] Ashenden, P. J. *The Designedr's Guide to VHDL.* Morgan Kaufmann Publishers, 1996.

[3] Bahar, I. R., Cho, H., Hatchel, D. G., Macci, E., and Somenzi, F. Symbolic timing analysis and re-synthesis for low power of combinational circuits containing false paths. *IEEE Transactions on Computer-Aided Design*, October 1997.

[4] Bartlett, A. V. and Dempster, G. A. Using carry-save adders in low power multiplier blocks. *IEEE*, 2001.

[5] Benini, L. and De Micheli, G. State assignment for low power dissipation. *Proceedings of IEEE Costum Integrated Circuit Conference*, pages 136–139, May 1994.

[6] Benini, L. and De Micheli, G. *Dynamic Power Mangement Design Techniques and CAD Tools.* Kluwer Academic Publishers, 1998.

[7] Benini, L. and De Michelli, G. State assignment for low power dissipation. *IEEE Journal of Solid-State Circuits*, 30:258–268, Mar 1995.

[8] Benini, L. and De Michelli, G. Automatic synthesis of low-power gated-clock finite-state machines. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 15(6), June 1996.

[9] Benini, L. and De Michelli, G. Automatic synthesis of low power gated-clock finite-state machines. *IEEE Transactions on Computer-Aided Design of Intgrated Circuits and Systens*, 15, June 1996.

[10] Benini, L. and De Michelli, G. System-level power optimization; techniques and tools. *ACM Transactions on Design Automation of Electronic Systems*, 5(2):115–192, April 2000.

[11] Callaway, K. T. and Swartzlander, E. E. J. Estimating the power consumption of cmos adders. In *Proceedings of the 11th Symposium on Computer Arithmetic*, pages 210–216, June 1993.

[12] Callaway, K. T. and Swartzlander, E. E. J. Power-delay characteristics of cmos multipliers. In *Proceedings of the 13th IEEE Symposium on Computer Arithmetic*, pages 26–32, July 1997.

[13] Chandrakasan, A., Yang, I., Vieri, C., and Antoniadis, D. Design considerations and tools for low-voltage disgital system design. *33rd ACM Design Automation Conference*, June 1996.

[14] Chandrakasan, P. A., Potkonjak, M., Rabaey, J., and Brodersen, W. R. HYPER-LP: A system for power minimization using architectural transformations. *International Conference on Computer-Aided Design ICCAD-92*, pages 300–303, Nov. 1992.

[15] Chandrakasan, P. A., Sheng, S., and Brodersen, W. R. Low-power cmos degital design. *IEEE Journal of Solid-State Circuits*, 27(4), April 1992.

[16] Cheng, W.-C. and Pedram, M. Memory bus encoding for low power: A tutorial. 2001.

[17] Cirit, A. M. Estimating dynamic power consumption of cmos circuits. *IEEE Int. Conference on Computer-Aided Design*, 1987.

[18] Correale, A. Overview of the power minimization techniques employed in the ibm powerpc 4xx embedded controllers. In *International Symposium on Low-Power Design.* ACM/IEEE, April 1995.

[19] Costa, E., Bampi, S., and Monteiro, J. Power efficient arithmetic operand encoding.

[20] Devadas, S. and Sharad, M. A survey of optimization techniques targeting low power vlsi circuits. *32nd ACM/IEEE Design Automation Conference.*

[21] Earl E. Swartzlander, J., (ed.). *Application Specific Processors.* Kluwer Academic Publishers, 1997.

[22] Fayed, A. and Bayoumi, M. A novel architecture for low-power design of parallel multipliers. *IEEE Computer Society Annual Workshop on VLSI (WVLSI 2001)*, April 2001.

[23] Hartley, I. R. and Parhi, K. K. *Digit-Serial Computation.* Kluwer Academic Publishers.

[24] Hennessy, J. L. and Patterson, D. A. *Computer Architecture : a Quantitative Approach.* Morgan Kaufmann Publishers, 2003.

[25] Kang, M. S. Accurate simulation of power dessipation in vlsi circuits. *IEEE Journal of Solid-State Electronics*, 21(5):889–891, October 1986.

[26] Kapadia, H., Benini, L., and De Micheli, G. Reducing switching activity on datapath buses with control signal gating. *IEEE Journal of Solid-State Circuits*, 34(3), March 1999.

[27] Khouri, S. K. and Jha, K. N. Leakage power analysis and reduction during behavioral synthesis. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 10:876–885, December 2002.

[28] Kim, T., Jao, W., and Tjisang, S. Arithmetic optimization using carry-save adders, 1998.

[29] Landman, P. High-level power estimation. *IEEE/ACM Int. Symposium on Low Power Electronics and Design*, pages 29–35, 1996.

[30] Landman, P. High-level power estimation. *ACM/IEEE Int. Symposium on Low Power Electronics and Design*, August 1996.

[31] Lavango, L., McGeer, C. P., Saldanha, A., and Sangiovanni, L. A. Timed-shannon circuits: a power-efficient design style and synthesis tool. *32nd ACM/IEEE Design Automation Conference*, 1995.

[32] Lee, K. H. and Rim, C. S. A hardware reduced multiplier for low power design. *Proceedings of the Second IEEE Asia Pacific Conference - AP-ASIC 2000*, pages 331 –334, 2000.

[33] Macii, E. Design of low-power digital circuits: Techniques and tools. Slides from IntraLed Seminar, October 2002.

[34] Meyr, H. and Noll, T. Designing complex socs for wireless communications: More than pushing mops and clock frequency. ISLPED 2002, available at http://www.iss.rwth-aachen.de/5_aktuell/files/ISLPED_public_version.pdf, 2002.

[35] Monteiro, C. J. and Oliveira, L. A. Finite state machine decomposition for low power. *Proceedings on Design Automation Conference*, pages 758–763, June 1998.

[36] Monteneiro, J. and Devadas, S. *Computer-Aided Design Techniques for Low Power Sequential Logic Circuits.* Kluwer Academic Publishers, 1997.

[37] Monteneiro, J., Devadas, S., and Ghosh, A. Retiming sequential circuits for low power. *IEEE/ACM Int. Conference on Computer-Aided Design*, 1993.

[38] Monteneiro, J. and Oliveira, A. Finite state machine decomposition for low power. *IEEE Int. Symposium on Circuits and Systems*, 1998.

[39] Munch, M., Wurth, B., Mehra, R., Sproch, J., and Wehn, N. Automating rt-level operand isolation to minimize power consumption in datapaths. *IEEE Design Automation and Test in Europe*, Marts 2000.

[40] Musoll, E. and Cortadella, J. High-level synthesis techniques for reducing the activity of functional units.

[41] Najm, F. and Xakellis, M. Statistical estimation of the switching activity in digital circuits. *ACM/IEEE Design Automation Conference*, pages 728–733, 1994.

[42] Noll, G. T. Carry-save arithmetic for high-speed signal processing. *IEEE*, 1990.

[43] Omondi, R. A. *Computer Arithmetic Systems: Algorithms, Architecture and Implementation.* Cambridge University Press, 1994.

[44] Paker, O. *Low Power Digital Signal Processing.* PhD thesis, Technical Univeristy of

Denmark, 2002.

[45] Raghunathan, A. Register transfer level power optimization with emphasis on glitch analysis and reduction. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 18(8), August 1999.

[46] Ranghunatathan, A., Dey, S., and Jha, K. N. Estimating dynamic power consumption of cmos circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 18(8), 1999.

[47] Ranghunathan, A. and Jha, N. Behavioral synthesis for low power. *Proc. of*, 1987.

[48] Roy, K. and Prasad, C. S. Syclop: Synthesis of cmos logic for low power applications. *IEEE Int. Conference on Computer Design*, October 1992.

[49] Salmon, O., G. J. and Klar, H. General algorithms for a simplified addition of 2's complement numbers. *IEEE Journal of Solid-State Circuits*, 30(7), July 1995.

[50] Savoj, H. and Brayton, K. R. The use of observability and external don't cares for the simplification of multi-level netwoks. *Proceedings of 27th IEEE/ACM Design Automation Conference*, pages 297–301, June 1990.

[51] Synopsys. Design compiler reference manual v2000.05, 2002.

[52] Synopsys. Design-ware reference manual v2000.05, 2002.

[53] Synopsys. Power compiler reference manual v2000.05, 2002.

[54] Tiwari, V. and Malik, S and, A. P. Guareded evaluation: Pushing power management to logic/synthesis design. *IEEE Transactions on Computer-Aided Design*, November 1998.

[55] Tiwari, V., Malik, S., and Pranav, A. Guarded evaluation: Pushing power management to logic synthesis/design. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 17(10), October 1998.

[56] Tsui, Y. C., Pedram, M., and Despain, M. A. Technology decomposition and mapping targetting low power dissipation. *IEEE/ACM Design Automation Conference*, June 1993.

[57] Usami, K. and Horowitz, M. Clustered voltage scaling technique for low power design. *IEEE/ACM Int. Symposium on Low Power Design*, April 1995.

[58] Wu, X. and Pedram, M. Low power sequential circuit design by using priority encoding and clock gating.

[59] Yeap, G. K. *Practical Low Power Digital VLSI Design.* Kluwer Academic Publishers, 1998.

[60] Yu, Z., Khoo, K.-Y., and Willson, N. A. J. The use of carry-save representation in joint module selection and retining.

[61] Zimmerman, R. Lectures on computer arithmetic: Principles, architectures and vlsi design. Integrated Systems Laboratory, ETH Zurich, available at http://www.iis.ee.ethz.ch/zimmi/publications /comp_arith_notes.ps.gz, 1999.

[62] Zimmermann, R. Vhdl library of arithmetic units. *IEEE*.

Appendix A

Source Code

A.1 Experiment 1: A Complex Arithmetic Unit

A.1.1 The testbench

```
-------------------------------------------------------------------------------
-- Title       : design testbench
-- Project     : High power arithmetic unit to be power managed
-------------------------------------------------------------------------------
-- File        : design_tb.vhd
-- Author      : Georgios Plakaris
-- Company     : Computer Systems Engineering, DTU
-- Date        : 12/11/2002
-------------------------------------------------------------------------------
-- Description :
-- A testbench for the design
-------------------------------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_signed.all;
use ieee.std_logic_arith.all;
use ieee.math_real.all;
library WORK;
use WORK.design_utils.all;
use WORK.sim_utils.all;

entity TOP_DESIGN_TB is

end TOP_DESIGN_TB;

architecture simple of TOP_DESIGN_TB is

  signal clk, rst, ovf : std_logic := '0';
  signal op1, op2, res : std_logic_vector(width-1 downto 0);
  signal opcode : std_logic_vector(ops-1 downto 0);

  constant Tpw_clk : time := 50 ns;

  component clock
    generic (
      period : TIME);
    port (
      clk : OUT std_logic := '0');
  end component;

  component bit_gen
    generic (
      bias : real);
    port (
      clk          : in  std_logic;
      word1, word2 : out std_logic_vector(width-1 downto 0));
  end component;

  component opcode_gen
    port (
      clk    : in  std_logic;
      rst    : in  std_logic;
      opcode : out std_logic_vector(ops-1 downto 0));
  end component;

  component top_design
    port (
      clk, rst         : in  std_logic;
      top1_in, top2_in : in  std_logic_vector(width-1 downto 0);
      top_opcode_in    : in  std_logic_vector(ops-1 downto 0);
      top_out          : out std_logic_vector(width-1 downto 0);
      top_ovf          : out std_logic);
  end component;

  signal word1_i, word2_i, result : std_logic_vector(width-1 downto 0);
  signal opcode_i : std_logic_vector(ops-1 downto 0);
  signal overflow : std_logic;

begin

  DUT: top_design
    port map (
        clk           => clk,
        rst           => rst,
        top1_in       => word1_i,
        top2_in       => word2_i,
        top_opcode_in => opcode_i,
        top_out       => result,
        top_ovf       => overflow);


  word_stimuli: bit_gen
    generic map (
        bias => 0.5)
    port map (
        clk   => clk,
        word1 => word1_i,
        word2 => word2_i);

  instruction_gen: opcode_gen
    port map (
        clk    => clk,
        rst    => rst,
        opcode => opcode_i);

  clock_gen: clock
    generic map (
      period => 50 ns)
    port map (
      clk => clk);

--   clock_gen : process is
--   begin
--     clk <= '1' after Tpw_clk, '0' after 2 * Tpw_clk;
--     wait for 2 * Tpw_clk;
--   end process clock_gen;

  rst <= '0', '1' after 120 ns;

  end simple;
```

A.1.2 The clock generator

```
--------------------------------------------------------------------------------
-- A simple clock generator. The period is specified in a generic and defaults
-- to 50 ns.
--------------------------------------------------------------------------------

LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;

ENTITY clock IS
    GENERIC (period :      TIME      := 50 ns);
    PORT    (clk    :  OUT std_logic := '0');
END clock;

ARCHITECTURE behaviour OF clock IS
BEGIN
    PROCESS
    BEGIN
        clk <= '1', '0' AFTER period/2;
        WAIT FOR period;
    END PROCESS;
END behaviour;
```

A.1.3 The opcode generator

```
-------------------------------------------------------------------------------
-- Title      : opcode generator
-- Project    : High power arithmetic unit to be power managed
-------------------------------------------------------------------------------
-- File       : opcode_gen.vhdl
-- Author     : Georgios Plakaris
-- Company    : Computer Systems Engineering, DTU
-- Date       : 12/11/2002
-------------------------------------------------------------------------------
-- Description :
-- Provides the sequence of instructions. LAter it should be modified to
-- represent tpical workloads of dsp processors
-------------------------------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_signed.all;
use ieee.std_logic_arith.all;
library WORK;
use WORK.design_utils.all;

entity opcode_gen is

  port (
    clk    : in  std_logic;
    rst    : in  std_logic;
    opcode : out std_logic_vector(ops-1 downto 0));

end opcode_gen;

architecture behavioral of opcode_gen is
  signal current_state, next_state : std_logic_vector(ops-1 downto 0);
  constant complex  : integer := 8;
  constant parallel : integer := 8;
  constant single   : integer := 8;
  constant noops    : integer := 1;
  signal counter    : integer;
begin  -- behavioral

  fsm: process (current_state, counter)
    variable temp_next : std_logic_vector(ops-1 downto 0) := nop;
  begin  -- process fsm
    if counter = 0 then
      case current_state is
        when nop        =>
            temp_next := mulc;
        when mulc       =>
            temp_next := mul16sfpar;
        when mul16sfpar =>
            temp_next := mul16sf;
        when mul16sf    =>
            temp_next := nop;
        when others     => null;
      end case;
    end if;
    next_state <= temp_next;
  end process fsm;

  state_reg: process (clk, rst)
    variable temp_count : integer;
  begin  -- process state_reg
    if rst = '0' then                   -- asynchronous reset (active low)
      current_state <= nop;
      counter <= noops;
    elsif clk'event and clk = '1' then  -- rising clock edge
      current_state <= next_state;
      if counter = 0 then
        case current_state is
            when nop =>
              counter <=  complex-1;
            when mulc =>
              counter <=  parallel-1;
            when mul16sfpar =>
              counter <=  single-1;
            when mul16sf =>
              counter <=  noops-1;
          when others => null;
        end case;
      else
        counter <= counter - 1;
      end if;
    end if;
  end process state_reg;
  opcode <= current_state;

end behavioral;
```

A.1.4 The design utilities package

```
-------------------------------------------------------------------------------
-- Title       : design_utils
-- Project     : Thesis
-------------------------------------------------------------------------------
-- File        : design_utils.vhd
-- Author      : Georgios Plakaris
-- Company     : Computer Systems Engineering, DTU
-- Date        : 12/11/2002
-------------------------------------------------------------------------------
-- Description :
-- A package containing declarations and subprograms useful to the poject
-------------------------------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;

package design_utils is

  constant width      : integer := 32;  -- width of the word
  constant half_width : integer := width/2;
  constant ops : integer := 4;
  -- define names for the opcodes.
  -- New opcodes are appended on the left side of the vector as exlusice 1-hot
  -- values
  constant nop : std_logic_vector(ops-1 downto 0) := "0001";
  constant mul16sf : std_logic_vector(ops-1 downto 0) := "0010";
  constant mul16sfpar : std_logic_vector(ops-1 downto 0) := "0100";
  constant mulc : std_logic_vector(ops-1 downto 0) := "1000";

  constant seed1 : integer := 111;
  constant seed2 : integer := 9763;
end design_utils;

package body design_utils is

end design_utils;
```

A.1.5 The simulation utilities package

```
-------------------------------------------------------------------------------
-- Title       : sim_utils
-- Project     : Thesis
-------------------------------------------------------------------------------
-- File        : sim_utils.vhd
-- Author      : Georgios Plakaris
-- Company     : Computer Systems Engineering, DTU
-- Date        : 12/11/2002
-------------------------------------------------------------------------------
-- Description :
-- A package containing declarations and subprograms useful to the poject
-------------------------------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
use ieee.math_real.all;

package sim_utils is
   procedure UNIFORM (variable Seed1,Seed2:inout integer; variable X:out real);
end sim_utils;

package body sim_utils is

      procedure UNIFORM(variable Seed1,Seed2:inout integer;variable X:out real) is
-- returns a pseudo-random number with uniform distribution in the
-- interval (0.0, 1.0).
-- Before the first call to UNIFORM, the seed values (Seed1, Seed2) must
-- be initialized to values in the range [1, 2147483562] and
-- [1, 2147483398] respectively.  The seed values are modified after
-- each call to UNIFORM.
-- This random number generator is portable for 32-bit computers, and
-- it has period ~2.30584*(10**18) for each set of seed values.
--
-- For VHDL-1992, the seeds will be global variables, functions to
-- initialize their values (INIT_SEED) will be provided, and the UNIFORM
-- procedure call will be modified accordingly.

variable z, k: integer;
begin
k := Seed1/53668;
Seed1 := 40014 * (Seed1 - k * 53668) - k * 12211;

if Seed1 < 0  then
Seed1 := Seed1 + 2147483563;
end if;


k := Seed2/52774;
Seed2 := 40692 * (Seed2 - k * 52774) - k * 3791;

if Seed2 < 0  then
Seed2 := Seed2 + 2147483399;
end if;

z := Seed1 - Seed2;
if z < 1 then
z := z + 2147483562;
end if;

X :=  REAL(Z)*4.656613e-10;
    end UNIFORM;



end sim_utils;
```

A.1.6 The top level design

```
-------------------------------------------------------------------------------
-- Title      : top_design
-- Project    : High power arithmetic unit to be power managed
-------------------------------------------------------------------------------
-- File       : top_design.vhdl
-- Author     : Georgios Plakaris
-- Company    : Computer Systems Engineering, DTU
-- Date       : 24/11/2002
-------------------------------------------------------------------------------
-- Description :
-- Connects the ALU in design with the environment registers.
-------------------------------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
library WORK;
use WORK.design_utils.all;

entity top_design is

  port (
    clk, rst         : in  std_logic;
    top1_in, top2_in : in  std_logic_vector(width-1 downto 0);
    top_opcode_in    : in  std_logic_vector(ops-1 downto 0);
    top_out          : out std_logic_vector(width-1 downto 0);
    top_ovf          : out std_logic);

end top_design;

architecture structural of top_design is
  component registers
    port (
      clk, rst              : in std_logic;
      in_1, in_2, in_res    : in  std_logic_vector(width-1 downto 0);
      in_opcode             : in  std_logic_vector(ops-1 downto 0);
      in_ovf                : in  std_logic;
      out_1, out_2, out_res : out std_logic_vector(width-1 downto 0);
      out_opcode            : inout std_logic_vector(ops-1 downto 0);
      out_ovf               : out std_logic);
  end component;

  component design
    port (
      A   : in  std_logic_vector(width-1 downto 0);
      B   : in  std_logic_vector(width-1 downto 0);
      op  : in  std_logic_vector(ops-1 downto 0);
      Z   : out std_logic_vector(width-1 downto 0);
      ovf : out std_logic);
  end component;

  signal A_i, B_i, Z_i : std_logic_vector(width-1 downto 0);
  signal op_i : std_logic_vector(ops-1 downto 0);
  signal ovf_i : std_logic;
begin  -- structural

  CALU: design
    port map (
        A   => A_i,
        B   => B_i,
        op  => op_i,
        Z   => Z_i,
        ovf => ovf_i);

  reg_banks: registers
    port map (
        clk        => clk,
        rst        => rst,
        in_1       => top1_in,
        in_2        => top2_in,
        in_res     => Z_i,
        in_opcode  => top_opcode_in,
        in_ovf     => ovf_i,
        out_1      => A_i,
        out_2      => B_i,
        out_res    => top_out,
        out_opcode => op_i,
        out_ovf    => top_ovf);

end structural;
```

A.1.7 The design testbench

```
-------------------------------------------------------------------------------
-- Title      : design testbench
-- Project    : High power arithmetic unit to be power managed
-------------------------------------------------------------------------------
-- File       : design_tb.vhd
-- Author     : Georgios Plakaris
-- Company    : Computer Systems Engineering, DTU
-- Date       : 12/11/2002
-------------------------------------------------------------------------------
-- Description :
-- A testbench for the design
-------------------------------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_signed.all;
use ieee.std_logic_arith.all;
library WORK;
use WORK.design_utils.all;

entity design_tb is

end design_tb;

architecture simple of design_tb is

  signal clk, rst, ovf : std_logic := '0';
  signal x_real, x_imag, y_real, y_imag, s_real, s_imag : std_logic_vector(half_width-1 downto 0);
  signal a, b, z : std_logic_vector(width-1 downto 0);
  signal opcode : std_logic_vector(ops-1 downto 0);

  type complex is record
      re, im : real;
    end record;

  signal x, y, s : complex := (0.0, 0.0);

  constant Tpw_clk : time := 50 ns;

  component design
    port (
      rst : in  std_logic;
      clk : in  std_logic;
      A   : in  std_logic_vector(width-1 downto 0);
      B   : in  std_logic_vector(width-1 downto 0);
      op  : in  std_logic_vector(ops-1 downto 0);
      Z   : out std_logic_vector(width-1 downto 0);
      ovf : out std_logic);
  end component;

  component to_vector
    port (
      r   : in  real;
      vec : out std_logic_vector(15 downto 0));
  end component;

  component to_fp
    port (
      vec : in  std_logic_vector(15 downto 0);
      r   : out real);
  end component;


begin

  x_real_converter : entity work.to_vector(behavioral) port map (x.re, x_real);
  x_imag_converter : entity work.to_vector(behavioral) port map (x.im, x_imag);
  y_real_converter : entity work.to_vector(behavioral) port map (y.re, y_real);
  y_imag_converter : entity work.to_vector(behavioral) port map (y.im, y_imag);


  a <= x_real&x_imag;
  b <= y_real&y_imag;
  s_real <= z(width-1 downto half_width);
  s_imag <= z(half_width-1 downto 0);

  dut: component design
    port map (
        rst => rst,
        clk => clk,
        A   => a,
        B   => b,
        op  => opcode,
        Z   => z,
        ovf => ovf);

  s_real_converter : entity work.to_fp(behavioral) port map (s_real, s.re);
  s_imag_converter : entity work.to_fp(behavioral) port map (s_imag, s.im);


  clock_gen : process is
  begin
    clk <= '1' after Tpw_clk, '0' after 2 * Tpw_clk;
    wait for 2 * Tpw_clk;
  end process clock_gen;

  rst <= '0', '1' after 120 ns;

  stimulus : process is
  begin
    -- single 16 fractional multiplication
    opcode <= mul16sf;
    x <= ( 0.0,  0.5); y <= ( 0.0,  0.5); wait until clk = '0';
    -- (x, 0.25)
    x <= ( 0.0,  0.5); y <= ( 0.0,  0.1); wait until clk = '0';
    -- (x, 0.05)
    x <= ( 0.0,  0.5); y <= ( 0.0, -0.5); wait until clk = '0';
    -- (x, -0.25)
    x <= ( 0.0, -0.5); y <= ( 0.0, -0.5); wait until clk = '0';
    -- (x, 0.25)
    -- test exceptions
    -- parallel 16 fractional multiplication
    opcode <= mul16sfpar;
    x <= ( 0.0,  0.5); y <= ( 0.0,  0.5); wait until clk = '0';
    -- (0, 0.25)
    x <= ( 0.5,  0.0); y <= ( 0.5,  0.0); wait until clk = '0';
    -- (0.25, 0)
    x <= ( 0.1,  0.5); y <= ( 0.1, -0.5); wait until clk = '0';
    -- (0.01, -0.25)
    x <= (-0.5, -0.5); y <= ( 0.5, -0.5); wait until clk = '0';
    -- (-0.25, 0.25)
    -- test exceptions
    -- complex multiplication
    opcode <= mulc;
    x <= ( 0.0,  0.5); y <= ( 0.0,  0.5); wait until clk = '0';
    -- (-0.25, 0)
    x <= ( 0.5,  0.0); y <= ( 0.5,  0.0); wait until clk = '0';
    -- (0.25, 0)
    x <= ( 0.1,  0.5); y <= ( 0.1, -0.5); wait until clk = '0';
    -- (0.26, 0)
    x <= (-0.5, -0.5); y <= ( 0.5, -0.5); wait until clk = '0';
    -- (-0.5, 0)
    -- test exceptions
    wait;
  end process stimulus;



end simple;

configuration rtl_design of design_tb is

  for simple
    for dut : design
      use entity WORK.design(SYN_high_power);
    end for;
--     for y_imag_converter : to_vector
--       use entity WORK.to_vector(behavioral);
--     end for;
--     for x_real_converter : to_vector
--       use entity WORK.to_vector(behavioral);
--     end for;
--     for y_real_converter : to_vector
--       use entity WORK.to_vector(behavioral);
--     end for;
--     for x_imag_converter : to_vector
--       use entity WORK.to_vector(behavioral);
--     end for;
--     for s_real_converter, s_imag_converter : to_fp
--       use entity WORK.to_fp(behavioral);
--     end for;
  end for;

end rtl_design;

-- configuration beh_design of design_tb is

--   for simple
--     for dut : design
--       use entity WORK.design(high_power);
--     end for;
--     for x_real_converter, y_real_vonverter, x_imag_converter, y_imag_converter : to_vector
--       use entity WORK.to_vector(behavioral);
--     end for;
--     for s_real_converter, s_imag_converter : to_fp
--       use entity WORK.to_fp(behavioral);
--     end for;
--   end for;

-- end beh_design;
```

A.1.8 The multiplier

```
-------------------------------------------------------------------------------
-- Title       : multiplier
-- Project     : High power arithmetic unit to be power managed
-------------------------------------------------------------------------------
-- File        : mult.vhd
-- Author      : Georgios Plakaris
-- Company     : Computer Systems Engineering, DTU
-- Date        : 12/11/2002
-------------------------------------------------------------------------------
-- Description :
-- A component insantiating a multiplier. Different architectures in the future
-- will accommodate different kind of mulipliers.
-------------------------------------------------------------------------------
library ieee;--, SYNOPSYS;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
--use SYNOPSYS.attributes.all;
library WORK;
use WORK.design_utils.all;

entity mult is
  generic (
    width : integer := half_width);
  port (
    op1 : in  std_logic_vector(width-1 downto 0);
    op2 : in  std_logic_vector(width-1 downto 0);
    res : out std_logic_vector(2*width-1 downto 0));

end mult;

architecture behavioral of mult is

begin  -- behavioral

    multiplication: process (op1, op2)
      variable a, b : signed(width-1 downto 0);
      variable c : signed(2*width-1 downto 0);
    begin  -- process addition
      a := signed(op1);
      b := signed(op2);
      c := a*b;
      res <= std_logic_vector(c);
    end process multiplication;

end behavioral;
```

A.1.9 The subtractor

```
-------------------------------------------------------------------------------
-- Title       : subtractor
-- Project     : High power arithmetic unit to be power managed
-------------------------------------------------------------------------------
-- File        : subtruct.vhd
-- Author      : Georgios Plakaris
-- Company     : Computer Systems Engineering, DTU
-- Date        : 12/11/2002
-------------------------------------------------------------------------------
-- Description :
-- A component insantiating a subtractor. Different architectures in the future
-- will accommodate different kind of subtractors.
-------------------------------------------------------------------------------
library ieee; --,SYNOPSYS;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
--use SYNOPSYS.attributes.all;
library WORK;
use WORK.design_utils.all;

entity subtruct is

  port (
    op1 : in  std_logic_vector(width-1 downto 0);
    op2 : in  std_logic_vector(width-1 downto 0);
    res : out std_logic_vector(width downto 0));

end subtruct;

architecture behavioral of subtruct is

begin  -- behavioral

    subtraction: process (op1, op2)
      variable a, b : signed(width downto 0);
      variable c : signed(width downto 0);
    begin  -- process addition
      a := signed(op1(width-1)&op1);
      b := signed(op2(width-1)&op2);
      c := a - b;
      res <= std_logic_vector(c);
    end process subtraction;

end behavioral;
```

A.1.10 The adder

```
-------------------------------------------------------------------------------
-- Title       : adder
-- Project     : High power arithmetic unit to be power managed
-------------------------------------------------------------------------------
-- File        : adder.vhd
-- Author      : Georgios Plakaris
-- Company     : Computer Systems Engineering, DTU
-- Date        : 12/11/2002
-------------------------------------------------------------------------------
-- Description :
-- A component insantiating an adder. Different architectures in the future
-- will accommodate different kind of adders.
-------------------------------------------------------------------------------
library ieee;--, SYNOPSYS;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
--use SYNOPSYS.attributes.all;
library WORK;
use WORK.design_utils.all;

entity add is

  port (
    op1 : in  std_logic_vector(width-1 downto 0);
    op2 : in  std_logic_vector(width-1 downto 0);
    res : out std_logic_vector(width downto 0));

end add;

architecture behavioral of add is

begin  -- behavioral

    addition: process (op1, op2)
      variable a, b : signed(width downto 0);
      variable c : signed(width downto 0);
    begin  -- process addition
      a := signed(op1(width-1)&op1);
      b := signed(op2(width-1)&op2);
      c := a + b;
      res <= std_logic_vector(c);
    end process addition;

end behavioral;
```

A.1.11 The design used in “PLAIN”, “REG_EN” and ‘CLK_GATED”

```
-------------------------------------------------------------------------------
-- Title       : design
-- Project     : High power arithmetic unit to be power managed
-------------------------------------------------------------------------------
-- File        : design.vhd
-- Author      : Georgios Plakaris
-- Company     : Computer Systems Engineering, DTU
-- Date        : 12/11/2002
-------------------------------------------------------------------------------
-- Description :
-- A kick off arithmetic core to be investigated for low power
-------------------------------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
library WORK;
use WORK.design_utils.all;

entity design is

  port (
    A   : in    std_logic_vector(width-1 downto 0);
    B   : in    std_logic_vector(width-1 downto 0);
    op  : in    std_logic_vector(ops-1 downto 0);
    Z   : out   std_logic_vector(width-1 downto 0);
    ovf : out   std_logic               --active high
  );

end design;

architecture basic of design is
  -- interconnect signals
  signal Z_i                      : std_logic_vector(width-1 downto 0)      := (others => '0');
  signal A_h, A_l, B_h, B_l       : std_logic_vector(half_width-1 downto 0) := (others => '0');
  signal mult_hh_out, mult_ll_out  : std_logic_vector(width-1 downto 0)      := (others => '0');
  signal mult_hl_out, mult_lh_out  : std_logic_vector(width-1 downto 0)      := (others => '0');
  signal mulc_res, mul_16sfpar_res : std_logic_vector(width-1 downto 0)      := (others => '0');
  signal mul_16sf_res             : std_logic_vector(width-1 downto 0)      := (others => '0');
  signal add_out, sub_out         : std_logic_vector(width downto 0)      := (others => '0');
  signal ovf_mulc                 : boolean                                 := false;
  signal ovf_mul16                : boolean                                 := false;
  signal ovf_mul16par             : boolean                                 := false;
  -- component declaration
  component add
    port (
      op1 : in  std_logic_vector(width-1 downto 0);
      op2 : in  std_logic_vector(width-1 downto 0);
      res : out std_logic_vector(width downto 0));
  end component;
  component subtruct
    port (
      op1 : in  std_logic_vector(width-1 downto 0);
      op2 : in  std_logic_vector(width-1 downto 0);
      res : out std_logic_vector(width downto 0));
  end component;
  component mult
    generic (
      width : integer);
    port (
      op1 : in  std_logic_vector(width-1 downto 0);
      op2 : in  std_logic_vector(width-1 downto 0);
      res : out std_logic_vector(2*width-1 downto 0));
  end component;

begin  -- basic
  -- component instantiation
  mult_hh: component mult
    generic map (width => half_width)
    port map (
        op1 => A_h,
        op2 => B_h,
        res => mult_hh_out);

  mult_ll: component mult
    generic map (width => half_width)
    port map (
        op1 => A_l,
        op2 => B_l,
        res => mult_ll_out);

  mult_lh: component mult
    generic map (width => half_width)
    port map (
        op1 => A_l,
        op2 => B_h,
        res => mult_lh_out);

  mult_hl:component mult
    generic map (width => half_width)
    port map (
        op1 => A_h,
        op2 => B_l,
        res => mult_hl_out);

  subtractor: component subtruct
    port map (
        op1 => mult_hh_out,
        op2 => mult_ll_out,
        res => sub_out);

  adder:component add
    port map (
        op1 => mult_lh_out,
        op2 => mult_hl_out,
        res => add_out);

  -- functionality
  mulc_res <= sub_out(width)&sub_out(width-3 downto half_width-1)
            &add_out(width)&add_out(width-3 downto half_width-1);
  mul_16sfpar_res <= mult_hh_out(31)&mult_hh_out(width-3 downto half_width-1)&
                     mult_ll_out(31)&mult_ll_out(width-3 downto half_width-1);
  mul_16sf_res        <= conv_std_logic_vector(0,16)&
                         mult_ll_out(31)&mult_ll_out(width-3 downto half_width-1);

  -- overflow logic
  ovf_mul16par <= (op = mul16sfpar) and (((mult_ll_out(31) xor mult_ll_out(30))
                                        or (mult_hh_out(31) xor mult_hh_out(30)))='1');
  ovf_mul16 <= (op = mul16sf) and ((mult_ll_out(31) xor mult_ll_out(30))='1');
  ovf_mulc  <= (op = mulc)and ((((sub_out(32)xor sub_out(31))
                                  or (sub_out(32)xor sub_out(30)))
                                or ((add_out(32)xor add_out(31))
                                  or (add_out(32)xor add_out(30))))='1');
  ovf <= '1' when ((ovf_mul16par or ovf_mul16) or ovf_mulc)=true
           else '0';
  -- steering logic
    output_mux: process (mulc_res, mul_16sfpar_res, mul_16sf_res, op)
  begin  -- process output_mux
    Z_i <= mulc_res;
    case op is
      when mulc =>
        Z_i <= mulc_res;
      when mul16sfpar =>
        Z_i <= mul_16sfpar_res;
      when mul16sf =>
        Z_i <= mul_16sf_res;
      when others =>
        null;
    end case;
  end process output_mux;
  Z <= Z_i;
  -- input connections
  A_h <= A(width-1 downto half_width);
  A_l <= A(half_width-1 downto 0);
  B_h <= B(width-1 downto half_width);
  B_l <= B(half_width-1 downto 0);

end basic;

configuration simple of design is

  for basic
    for adder : add
      use entity WORK.add(behavioral);
    end for;
    for subtractor : subtruct
      use entity WORK.subtruct(behavioral);
    end for;
    for  mult_hh, mult_ll, mult_hl, mult_lh  : mult
      use entity WORK.mult(behavioral);
    end for;
  end for;

end simple;
```

A.1.12 The design used in “OP_ISOL” and “CLK_GATED_OP_ISOL”

```
-------------------------------------------------------------------------------
-- Title       : design
-- Project     : High power arithmetic unit to be power managed
-------------------------------------------------------------------------------
-- File        : design.vhd
-- Author      : Georgios Plakaris
-- Company     : Computer Systems Engineering, DTU
-- Date        : 12/11/2002
-------------------------------------------------------------------------------
-- Description :
-- A kick off arithmetic core to be investigated for low power
-------------------------------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
library WORK;
use WORK.design_utils.all;

entity design is

  port (
    A   : in    std_logic_vector(width-1 downto 0);
    B   : in    std_logic_vector(width-1 downto 0);
    op  : in    std_logic_vector(ops-1 downto 0);
    Z   : out   std_logic_vector(width-1 downto 0);
    ovf : out   std_logic                --active high
  );

end design;

architecture basic of design is
  -- interconnect signals
  signal Z_i                      : std_logic_vector(width-1 downto 0)      := (others => '0');
  signal A_h, A_l, B_h, B_l       : std_logic_vector(half_width-1 downto 0) := (others => '0');
  signal hh_1, hh_2, ll_1, ll_2   : std_logic_vector(half_width-1 downto 0) := (others => '0');
  signal hl_1, hl_2, lh_1, lh_2   : std_logic_vector(half_width-1 downto 0) := (others => '0');
  signal hh, ll, lh, hl           : std_logic_vector(width-1 downto 0)      := (others => '0');
  signal isol_hh, isol_ll         : std_logic_vector(width-1 downto 0)      := (others => '0');
  signal isol_lh, isol_hl          : std_logic_vector(width-1 downto 0)      := (others => '0');
  signal mult_hh_out, mult_ll_out  : std_logic_vector(width-1 downto 0)      := (others => '0');
  signal mult_hl_out, mult_lh_out  : std_logic_vector(width-1 downto 0)      := (others => '0');
  signal mulc_res, mul_16sfpar_res : std_logic_vector(width-1 downto 0)      := (others => '0');
  signal mul_16sf_res              : std_logic_vector(width-1 downto 0)      := (others => '0');
  signal add_out, sub_out          : std_logic_vector(width downto 0)      := (others => '0');
  signal ovf_mulc                  : boolean                                 := false;
  signal ovf_mul16                 : boolean                                 := false;
  signal ovf_mul16par              : boolean                                 := false;
  signal ctrl_hh, ctrl_ll, ctrl_hl, ctrl_lh : std_logic;
  -- component declaration
  component add
    port (
      op1 : in  std_logic_vector(width-1 downto 0);
      op2 : in  std_logic_vector(width-1 downto 0);
      res : out std_logic_vector(width downto 0));
  end component;
  component subtruct
    port (
      op1 : in  std_logic_vector(width-1 downto 0);
      op2 : in  std_logic_vector(width-1 downto 0);
      res : out std_logic_vector(width downto 0));
  end component;
  component mult
    generic (
      width : integer);
    port (
      op1 : in  std_logic_vector(width-1 downto 0);
      op2 : in  std_logic_vector(width-1 downto 0);
      res : out std_logic_vector(2*width-1 downto 0));
  end component;

  component isolation_logic
    generic (
      width : integer;
      style : integer);
    port (
      as          : in  std_logic;
      op          : in  std_logic_vector(width-1 downto 0);
      op_isolated : out std_logic_vector(width-1 downto 0));
  end component;
  constant isolation_style : integer := 2;
begin  -- basic
  -- component instantiation
  mult_hh: component mult
    generic map (width => half_width)
    port map (
        op1 => hh_1,
        op2 => hh_2,
        res => mult_hh_out);

  mult_ll: component mult
    generic map (width => half_width)
    port map (
        op1 => ll_1,
        op2 => ll_2,
        res => mult_ll_out);

  mult_lh: component mult
    generic map (width => half_width)
    port map (
        op1 => lh_1,
        op2 => lh_2,
        res => mult_lh_out);

  mult_hl:component mult
    generic map (width => half_width)
    port map (
        op1 => hl_1,
        op2 => hl_2,
        res => mult_hl_out);

  subtractor: component subtruct
    port map (
        op1 => mult_hh_out,
        op2 => mult_ll_out,
        res => sub_out);

  adder:component add
    port map (
        op1 => mult_lh_out,
        op2 => mult_hl_out,
        res => add_out);
```

```
-- functionality
mulc_res <= sub_out(width)&sub_out(width-3 downto half_width-1)
            &add_out(width)&add_out(width-3 downto half_width-1);
mul_16sfpar_res <= mult_hh_out(31)&mult_hh_out(width-3 downto half_width-1)&
                   mult_ll_out(31)&mult_ll_out(width-3 downto half_width-1);
mul_16sf_res        <= conv_std_logic_vector(0,16)&
                       mult_ll_out(31)&mult_ll_out(width-3 downto half_width-1);

-- overflow logic
ovf_mul16par <= (op = mul16sfpar) and (((mult_ll_out(31) xor mult_ll_out(30))
                                       or (mult_hh_out(31) xor mult_hh_out(30)))='1');
ovf_mul16 <= (op = mul16sf) and ((mult_ll_out(31) xor mult_ll_out(30))='1');
ovf_mulc  <= (op = mulc)and ((((sub_out(32)xor sub_out(31))
                                 or (sub_out(32)xor sub_out(30)))
                               or ((add_out(32)xor add_out(31))
                                 or (add_out(32)xor add_out(30))))='1');
ovf <= '1' when ((ovf_mul16par or ovf_mul16) or ovf_mulc)=true
         else '0';

-- steering logic
  output_mux: process (mulc_res, mul_16sfpar_res, mul_16sf_res, op)
begin  -- process output_mux
  Z_i <= mulc_res;
  case op is
    when mulc =>
      Z_i <= mulc_res;
    when mul16sfpar =>
      Z_i <= mul_16sfpar_res;
    when mul16sf =>
      Z_i <= mul_16sf_res;
    when others =>
      null;
  end case;
end process output_mux;
Z <= Z_i;

--operand isolation logic
ctrl_hh <= op(3) or op(2);
ctrl_ll <= not op(0);
ctrl_lh <= op(3);
ctrl_hl <= op(3);
isolate_hh: isolation_logic
  generic map (
    width => width,
    style => isolation_style)
  port map (
    as          => ctrl_hh,           -- activate when mulc or par
    op          => hh,
    op_isolated => isol_hh);
isolate_hl: isolation_logic
  generic map (
    width => width,
    style => isolation_style)
  port map (
    as          => ctrl_hl,             -- activate only when mulc
    op          => hl,
    op_isolated => isol_hl);
isolate_lh: isolation_logic
  generic map (
    width => width,
    style => isolation_style)
  port map (
    as          => ctrl_lh,             -- activate only when mulc
    op          => lh,
    op_isolated => isol_lh);
isolate_ll: isolation_logic
  generic map (
    width => width,
    style => isolation_style)
  port map (
    as          => ctrl_ll,             -- isolate only when nop
    op          => ll,
    op_isolated => isol_ll);
-- input connections
A_h <= A(width-1 downto half_width);
A_l <= A(half_width-1 downto 0);
B_h <= B(width-1 downto half_width);
B_l <= B(half_width-1 downto 0);
hh <= A_h&B_h;
lh <= A_l&B_h;
hl <= A_h&B_l;
ll <= A_l&B_l;
```

```
  -- isolation output connections
  hh_1 <= isol_hh(width-1 downto half_width);
  hh_2 <= isol_hh(half_width-1 downto 0);
  lh_1 <= isol_lh(width-1 downto half_width);
  lh_2 <= isol_lh(half_width-1 downto 0);
  hl_1 <= isol_hl(width-1 downto half_width);
  hl_2 <= isol_hl(half_width-1 downto 0);
  ll_1 <= isol_ll(width-1 downto half_width);
  ll_2 <= isol_ll(half_width-1 downto 0);

end basic;

configuration simple of design is

  for basic
    for adder : add
      use entity WORK.add(behavioral);
    end for;
    for subtractor : subtruct
      use entity WORK.subtruct(behavioral);
    end for;
    for  mult_hh, mult_ll, mult_hl, mult_lh  : mult
      use entity WORK.mult(behavioral);
    end for;
    for others : isolation_logic
      use entity WORK.isolation_logic(structural);
    end for;
  end for;

end simple;
```

A.1.13 The design used in "'CLK_GATED_OP_ISOL_OPT"

```
-------------------------------------------------------------------------------
-- Title      : design
-- Project    : High power arithmetic unit to be power managed
-------------------------------------------------------------------------------
-- File       : design.vhd
-- Author     : Georgios Plakaris
-- Company    : Computer Systems Engineering, DTU
-- Date       : 12/11/2002
-------------------------------------------------------------------------------
-- Description :
-- A kick off arithmetic core to be investigated for low power
-------------------------------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
library WORK;
use WORK.design_utils.all;

entity design is

  port (
    A   : in    std_logic_vector(width-1 downto 0);
    B   : in    std_logic_vector(width-1 downto 0);
    op  : in    std_logic_vector(ops-1 downto 0);
    Z   : out   std_logic_vector(width-1 downto 0);
    ovf : out   std_logic               --active high
  );

end design;

architecture basic of design is
  -- interconnect signals
  signal Z_i                      : std_logic_vector(width-1 downto 0)      := (others => '0');
  signal A_h, A_l, B_h, B_l       : std_logic_vector(half_width-1 downto 0) := (others => '0');
  signal hl_1, hl_2, lh_1, lh_2   : std_logic_vector(half_width-1 downto 0) := (others => '0');
  signal lh, hl                   : std_logic_vector(width-1 downto 0)      := (others => '0');
  signal isol_lh, isol_hl         : std_logic_vector(width-1 downto 0)      := (others => '0');
  signal mult_hh_out, mult_ll_out  : std_logic_vector(width-1 downto 0)      := (others => '0');
  signal mult_hl_out, mult_lh_out  : std_logic_vector(width-1 downto 0)      := (others => '0');
  signal mulc_res, mul_16sfpar_res : std_logic_vector(width-1 downto 0)      := (others => '0');
  signal mul_16sf_res             : std_logic_vector(width-1 downto 0)      := (others => '0');
  signal add_out, sub_out         : std_logic_vector(width downto 0)      := (others => '0');
  signal ovf_mulc                 : boolean                                 := false;
  signal ovf_mul16                : boolean                                 := false;
  signal ovf_mul16par             : boolean                                 := false;
  signal ctrl_hl, ctrl_lh : std_logic;
  -- component declaration
  component add
```

```
    port (
      op1 : in  std_logic_vector(width-1 downto 0);
      op2 : in  std_logic_vector(width-1 downto 0);
      res : out std_logic_vector(width downto 0));
  end component;
  component subtruct
    port (
      op1 : in  std_logic_vector(width-1 downto 0);
      op2 : in  std_logic_vector(width-1 downto 0);
      res : out std_logic_vector(width downto 0));
  end component;
  component mult
    generic (
      width : integer);
    port (
      op1 : in  std_logic_vector(width-1 downto 0);
      op2 : in  std_logic_vector(width-1 downto 0);
      res : out std_logic_vector(2*width-1 downto 0));
  end component;

  component isolation_logic
    generic (
      width : integer;
      style : integer);
    port (
      as          : in  std_logic;
      op          : in  std_logic_vector(width-1 downto 0);
      op_isolated : out std_logic_vector(width-1 downto 0));
  end component;
  constant isolation_style : integer := 2;
begin  -- basic
  -- component instantiation
  mult_hh: component mult
    generic map (width => half_width)
    port map (
        op1 => A_h,
        op2 => B_h,
        res => mult_hh_out);

  mult_ll: component mult
    generic map (width => half_width)
    port map (
        op1 => A_l,
        op2 => B_l,
        res => mult_ll_out);

  mult_lh: component mult
    generic map (width => half_width)
    port map (
        op1 => lh_1,
        op2 => lh_2,
        res => mult_lh_out);

  mult_hl:component mult
    generic map (width => half_width)
    port map (
        op1 => hl_1,
        op2 => hl_2,
        res => mult_hl_out);

  subtractor: component subtruct
    port map (
        op1 => mult_hh_out,
        op2 => mult_ll_out,
        res => sub_out);

  adder:component add
    port map (
        op1 => mult_lh_out,
        op2 => mult_hl_out,
        res => add_out);

  -- functionality
  mulc_res <= sub_out(width)&sub_out(width-3 downto half_width-1)
              &add_out(width)&add_out(width-3 downto half_width-1);
  mul_16sfpar_res <= mult_hh_out(31)&mult_hh_out(width-3 downto half_width-1)&
                     mult_ll_out(31)&mult_ll_out(width-3 downto half_width-1);
  mul_16sf_res        <= conv_std_logic_vector(0,16)&
                           mult_ll_out(31)&mult_ll_out(width-3 downto half_width-1);

  -- overflow logic
  ovf_mul16par <= (op = mul16sfpar) and (((mult_ll_out(31) xor mult_ll_out(30))
                                        or (mult_hh_out(31) xor mult_hh_out(30)))='1');
  ovf_mul16 <= (op = mul16sf) and ((mult_ll_out(31) xor mult_ll_out(30))='1');
  ovf_mulc  <= (op = mulc)and ((((sub_out(32)xor sub_out(31))
                                  or (sub_out(32)xor sub_out(30)))
                                or ((add_out(32)xor add_out(31))
                                  or (add_out(32)xor add_out(30))))='1');
  ovf <= '1' when ((ovf_mul16par or ovf_mul16) or ovf_mulc)=true
         else '0';

  -- steering logic
    output_mux: process (mulc_res, mul_16sfpar_res, mul_16sf_res, op)
  begin  -- process output_mux
    Z_i <= mulc_res;
    case op is
      when mulc =>
        Z_i <= mulc_res;
      when mul16sfpar =>
        Z_i <= mul_16sfpar_res;
      when mul16sf =>
        Z_i <= mul_16sf_res;
      when others =>
        null;
    end case;
  end process output_mux;
  Z <= Z_i;

  --operand isolation logic
  ctrl_lh <= op(3);
  ctrl_hl <= op(3);
  -- mult hh is operand isolated by clock gating the upper
  -- parts of the registers in case of nop or single
  isolate_hl: isolation_logic
    generic map (
      width => width,
      style => isolation_style)
    port map (
      as          => ctrl_hl,             -- activate only when mulc
      op          => hl,
      op_isolated => isol_hl);
  isolate_lh: isolation_logic
    generic map (
      width => width,
      style => isolation_style)
    port map (
      as          => ctrl_lh,             -- activate only when mulc
      op          => lh,
      op_isolated => isol_lh);
  -- mult_ll is operand isolated by clock gating in case of nop. Otherwise it
  -- is active.

  -- input connections
  A_h <= A(width-1 downto half_width);
  A_l <= A(half_width-1 downto 0);
  B_h <= B(width-1 downto half_width);
  B_l <= B(half_width-1 downto 0);
  lh <= A_l&B_h;
  hl <= A_h&B_l;
  -- isolation output connections
  lh_1 <= isol_lh(width-1 downto half_width);
  lh_2 <= isol_lh(half_width-1 downto 0);
  hl_1 <= isol_hl(width-1 downto half_width);
  hl_2 <= isol_hl(half_width-1 downto 0);

end basic;

configuration simple of design is

  for basic
    for adder : add
      use entity WORK.add(behavioral);
    end for;
    for subtractor : subtruct
      use entity WORK.subtruct(behavioral);
    end for;
    for  mult_hh, mult_ll, mult_hl, mult_lh  : mult
      use entity WORK.mult(behavioral);
    end for;
    for others : isolation_logic
      use entity WORK.isolation_logic(structural);
    end for;
  end for;

end simple;
```

A.1.14 The design used in “DECOUPLED”

```
-------------------------------------------------------------------------------
-- Title      : design
-- Project    : High power arithmetic unit to be power managed
-------------------------------------------------------------------------------
-- File       : design.vhd
-- Author     : Georgios Plakaris
-- Company    : Computer Systems Engineering, DTU
-- Date       : 12/11/2002
-------------------------------------------------------------------------------
-- Description :
-- A kick off arithmetic core to be investigated for low power
-------------------------------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
library WORK;
use WORK.design_utils.all;

entity design is

  port (
    A   : in    std_logic_vector(width-1 downto 0);
    B   : in    std_logic_vector(width-1 downto 0);
    op  : in    std_logic_vector(ops-1 downto 0);
    Z   : out   std_logic_vector(width-1 downto 0);
    ovf : out   std_logic               --active high
  );

end design;

architecture basic of design is
  -- interconnect signals
  signal Z_i                        : std_logic_vector(width-1 downto 0)      := (others => '0');
  signal A_h, A_l, B_h, B_l         : std_logic_vector(half_width-1 downto 0) := (others => '0');
  signal hh_1, hh_2, ll_1, ll_2     : std_logic_vector(half_width-1 downto 0) := (others => '0');
  signal hl_1, hl_2, lh_1, lh_2     : std_logic_vector(half_width-1 downto 0) := (others => '0');
  signal hh, ll, lh, hl, isol_A, isol_B : std_logic_vector(width-1 downto 0)      := (others => '0');
  signal isol_hh, isol_ll           : std_logic_vector(width-1 downto 0)      := (others => '0');
  signal isol_lh, isol_hl           : std_logic_vector(width-1 downto 0)      := (others => '0');
  signal mult_hh_out, mult_ll_out   : std_logic_vector(width-1 downto 0)      := (others => '0');
  signal mult_hl_out, mult_lh_out   : std_logic_vector(width-1 downto 0)      := (others => '0');
  signal mulc_res, mul_16sfpar_res  : std_logic_vector(width-1 downto 0)      := (others => '0');
  signal mul_16sf_res               : std_logic_vector(width-1 downto 0)      := (others => '0');
  signal add_out, sub_out           : std_logic_vector(width downto 0)      := (others => '0');
  signal ovf_mulc                   : boolean                                 := false;
  signal ovf_mul16                  : boolean                                 := false;
  signal ovf_mul16par               : boolean                                 := false;
  signal ctrl_hh, ctrl_ll, ctrl_hl, ctrl_lh, ctrl : std_logic;
  -- component declaration
  component add
    port (
      op1 : in  std_logic_vector(width-1 downto 0);
      op2 : in  std_logic_vector(width-1 downto 0);
      res : out std_logic_vector(width downto 0));
  end component;
  component subtruct
    port (
      op1 : in  std_logic_vector(width-1 downto 0);
      op2 : in  std_logic_vector(width-1 downto 0);
      res : out std_logic_vector(width downto 0));
  end component;
  component mult
    generic (
      width : integer);
    port (
      op1 : in  std_logic_vector(width-1 downto 0);
      op2 : in  std_logic_vector(width-1 downto 0);
      res : out std_logic_vector(2*width-1 downto 0));
  end component;

  component isolation_logic
    generic (
      width : integer;
      style : integer);
    port (
      as          : in  std_logic;
      op          : in  std_logic_vector(width-1 downto 0);
      op_isolated : out std_logic_vector(width-1 downto 0));
  end component;
  constant isolation_style : integer := 2;
begin  -- basic
```

```
  -- component instantiation
  mult_hh: component mult
    generic map (width => half_width)
    port map (
        op1 => A_h,
        op2 => B_h,
        res => mult_hh_out);

  mult_ll: component mult
    generic map (width => half_width)
    port map (
        op1 => A_l,
        op2 => B_l,
        res => mult_ll_out);

  mult_lh: component mult
    generic map (width => half_width)
    port map (
        op1 => A_l,
        op2 => B_h,
        res => mult_lh_out);

  mult_hl:component mult
    generic map (width => half_width)
    port map (
        op1 => A_h,
        op2 => B_l,
        res => mult_hl_out);

  subtractor: component subtruct
    port map (
        op1 => mult_hh_out,
        op2 => mult_ll_out,
        res => sub_out);

  adder:component add
    port map (
        op1 => mult_lh_out,
        op2 => mult_hl_out,
        res => add_out);

  -- functionality
  mulc_res <= sub_out(width)&sub_out(width-3 downto half_width-1)
              &add_out(width)&add_out(width-3 downto half_width-1);

  -- overflow logic
  ovf_mulc  <= (op = mulc)and ((((sub_out(32)xor sub_out(31))
                                  or (sub_out(32)xor sub_out(30)))
                                or ((add_out(32)xor add_out(31))
                                  or (add_out(32)xor add_out(30))))='1');
  ovf <= '1' when ovf_mulc=true else '0';
  -- steering logic
  Z <= mulc_res;

  --operand isolation logic
  ctrl <= op(3);

  isolate_1: isolation_logic
    generic map (
      width => width,
      style => isolation_style)
    port map (
      as          => ctrl,             -- activate only when mulc
      op          => A,
      op_isolated => isol_A);
  isolate_2: isolation_logic
    generic map (
      width => width,
      style => isolation_style)
    port map (
      as          => ctrl,             -- activate only when mulc
      op          => B,
      op_isolated => isol_B);

--   -- input connections
  A_h <= isol_A(width-1 downto half_width);
  A_l <= isol_A(half_width-1 downto 0);
  B_h <= isol_B(width-1 downto half_width);
  B_l <= isol_B(half_width-1 downto 0);

end basic;

configuration simple of design is
```

```
  for basic
    for adder : add
      use entity WORK.add(behavioral);
    end for;
    for subtractor : subtruct
      use entity WORK.subtruct(behavioral);
    end for;
    for  mult_hh, mult_ll, mult_hl, mult_lh  : mult
      use entity WORK.mult(behavioral);
    end for;
    for others : isolation_logic
      use entity WORK.isolation_logic(structural);
    end for;
  end for;

end simple;
```

A.1.15 The register entities

The register entities are common for all architectures

```
entity registers is

  port (
    clk, rst             : in  std_logic;
    in_1, in_2, in_res    : in  std_logic_vector(width-1 downto 0);
    in_opcode            : in  std_logic_vector(ops-1 downto 0);
    in_ovf               : in  std_logic;
    out_1, out_2, out_res : out std_logic_vector(width-1 downto 0);
    out_opcode           : inout std_logic_vector(ops-1 downto 0);
    out_ovf              : out std_logic);

end registers;
```

A.1.16 The register architecture for “PLAIN”

```
architecture structural of registers is

begin  -- structural
    regs: process (clk, rst)
    begin  -- process regs
      if rst = '0' then                 -- asynchronous reset (active low)
        out_1   <= (others => '0');
        out_2   <= (others => '0');
        out_res <= (others => '0');
        out_opcode <= (others => '0');
        out_ovf <= '0';
      elsif clk'event and clk = '1' then  -- rising clock edge
        out_res <= in_res;
        out_ovf <= in_ovf;
        out_1 <= in_1;
        out_2 <= in_2;
        out_opcode <= in_opcode;
      end if;
    end process regs;
  end structural;
```

A.1.17 The register architecture for “REG_EN”

```
architecture structural of registers is

begin  -- structural
    regs: process (clk, rst)
    begin  -- process regs
      if rst = '0' then                 -- asynchronous reset (active low)
        out_1   <= (others => '0');
        out_2   <= (others => '0');
        out_res <= (others => '0');
        out_opcode <= (others => '0');
        out_ovf <= '0';
      elsif clk'event and clk = '1' then  -- rising clock edge
        if out_opcode(0) = '0' then
          out_res <= in_res;
          out_ovf <= in_ovf;            -- overflow is invilidated in case of nop
        end if;
        if in_opcode(0) = '0' then
```

```
          out_1(half_width-1 downto 0) <= in_1(half_width-1 downto 0);
          out_2(half_width-1 downto 0) <= in_2(half_width-1 downto 0);
        end if;
        if ((in_opcode(0) = '0') and (in_opcode(1) = '0')) then
          out_1(width-1 downto half_width) <= in_1(width-1 downto half_width);
          out_2(width-1 downto half_width) <= in_2(width-1 downto half_width);
        end if;
        out_opcode <= in_opcode;
      end if;
    end process regs;
  end structural;
```

A.1.18 The register architecture for "OP_ISOL"

```
architecture structural of registers is

begin  -- structural
    regs: process (clk, rst)
    begin  -- process regs
      if rst = '0' then                 -- asynchronous reset (active low)
        out_1   <= (others => '0');
        out_2   <= (others => '0');
        out_res <= (others => '0');
        out_opcode <= (others => '0');
        out_ovf <= '0';
      elsif clk'event and clk = '1' then  -- rising clock edge
        if out_opcode(0) = '0' then
          out_res <= in_res;
          out_ovf <= in_ovf;            -- overflow is invilidated in case of nop
        end if;
        if in_opcode(0) = '0' then
          out_1(half_width-1 downto 0) <= in_1(half_width-1 downto 0);
          out_2(half_width-1 downto 0) <= in_2(half_width-1 downto 0);
        end if;
        if ((in_opcode(0) = '0') and (in_opcode(1) = '0')) then
          out_1(width-1 downto half_width) <= in_1(width-1 downto half_width);
          out_2(width-1 downto half_width) <= in_2(width-1 downto half_width);
        end if;
        out_opcode <= in_opcode;
      end if;
    end process regs;
  end structural;
```

A.1.19 The register architecture for "CLK_GATED_OP_ISOL"

```
architecture structural of registers is

begin  -- structural
    regs: process (clk, rst)
    begin  -- process regs
      if rst = '0' then                 -- asynchronous reset (active low)
        out_1   <= (others => '0');
        out_2   <= (others => '0');
        out_res <= (others => '0');
        out_opcode <= (others => '0');
        out_ovf <= '0';
      elsif clk'event and clk = '1' then  -- rising clock edge
        if out_opcode(0) = '0' then
          out_res <= in_res;
          out_ovf <= in_ovf;            -- overflow is invilidated in case of nop
        end if;
        if in_opcode(0) = '0' then
          out_1(half_width-1 downto 0) <= in_1(half_width-1 downto 0);
          out_2(half_width-1 downto 0) <= in_2(half_width-1 downto 0);
        end if;
        if ((in_opcode(0) = '0') and (in_opcode(1) = '0')) then
          out_1(width-1 downto half_width) <= in_1(width-1 downto half_width);
          out_2(width-1 downto half_width) <= in_2(width-1 downto half_width);
        end if;
        out_opcode <= in_opcode;
      end if;
    end process regs;
  end structural;
```

A.1.20 The register architecture for "CLK_GATED_OP_ISOL_OPT"

```
architecture structural of registers is
```

```
begin  -- structural
    regs: process (clk, rst)
    begin  -- process regs
      if rst = '0' then                 -- asynchronous reset (active low)
        out_1   <= (others => '0');
        out_2   <= (others => '0');
        out_res <= (others => '0');
        out_opcode <= (others => '0');
        out_ovf <= '0';
      elsif clk'event and clk = '1' then  -- rising clock edge
        if out_opcode(0) = '0' then
          out_res <= in_res;
          out_ovf <= in_ovf;            -- overflow is invilidated in case of nop
        end if;
        if in_opcode(0) = '0' then
          out_1(half_width-1 downto 0) <= in_1(half_width-1 downto 0);
          out_2(half_width-1 downto 0) <= in_2(half_width-1 downto 0);
        end if;
        if ((in_opcode(0) = '0') and (in_opcode(1) = '0')) then
          out_1(width-1 downto half_width) <= in_1(width-1 downto half_width);
          out_2(width-1 downto half_width) <= in_2(width-1 downto half_width);
        end if;
        out_opcode <= in_opcode;
      end if;
    end process regs;
  end structural;
```

A.1.21 The register architecture for "DECOUPLED"

```
architecture structural of registers is

begin  -- structural
    regs: process (clk, rst)
    begin  -- process regs
      if rst = '0' then                 -- asynchronous reset (active low)
        out_1   <= (others => '0');
        out_2   <= (others => '0');
        out_res <= (others => '0');
        out_opcode <= (others => '0');
        out_ovf <= '0';
      elsif clk'event and clk = '1' then  -- rising clock edge
        out_res <= in_res;
        out_ovf <= in_ovf;
        out_1 <= in_1;
        out_2 <= in_2;
        out_opcode <= in_opcode;
      end if;
    end process regs;
  end structural;
```

A.1.22 The isolation logic for "OP_ISOL", "CLK_GATED_OP_ISOL" and "CLK_GATED_OP_ISOL_OPT"

```
-------------------------------------------------------------------------------
-- Title      : isolation_logic
-- Project    : High power arithmetic unit to be power managed
-------------------------------------------------------------------------------
-- File       : isolation_logic.vhdl
-- Author     : Georgios Plakaris
-- Company    : Computer Systems Engineering, DTU
-- Date       : 27/11/2002
-------------------------------------------------------------------------------
-- Description :
-- An operand isolation latch
-------------------------------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
library WORK;
use WORK.design_utils.all;

entity isolation_logic is

  generic (
    width : integer := half_width;
    style : integer := 1);

  port (
    as          : in  std_logic;
    op          : in  std_logic_vector(width-1 downto 0);
    op_isolated : out std_logic_vector(width-1 downto 0));

end isolation_logic;

architecture structural of isolation_logic is

begin  -- structural

  and_based: if style = 0 generate
    gen_logic: for i in op'range generate
      op_isolated(i) <= op(i) and as;
    end generate gen_logic;
  end generate and_based;

  or_based: if style = 1 generate
    gen_logic: for i in op'range generate
      op_isolated(i) <= (not as) or op(i);
    end generate gen_logic;
  end generate or_based;

  latch_based: if style = 2 generate
    gen_logic: process (as, op)
    begin  -- process gen_logic
      if as = '1' then
        op_isolated <= op;
      end if;
    end process gen_logic;
  end generate latch_based;
end structural;
```

A.1.23 The isolation logic for "DECOUPLED"

```
-------------------------------------------------------------------------------
-- Title       : isolation_logic
-- Project     : High power arithmetic unit to be power managed
-------------------------------------------------------------------------------
-- File        : isolation_logic.vhdl
-- Author      : Georgios Plakaris
-- Company     : Computer Systems Engineering, DTU
-- Date        : 27/11/2002
-------------------------------------------------------------------------------
-- Description :
-- An operand isolation latch
-------------------------------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
library WORK;
use WORK.design_utils.all;

entity isolation_logic is

  generic (
    width : integer := half_width;
    style : integer := 1);

  port (
    as          : in  std_logic;
    op          : in  std_logic_vector(width-1 downto 0);
    op_isolated : out std_logic_vector(width-1 downto 0));

end isolation_logic;

architecture structural of isolation_logic is
  signal temp : std_logic_vector(width-1 downto 0) := (others => '0');
begin  -- structural

  and_based: if style = 0 generate
    gen_logic: for i in op'range generate
      temp(i) <= op(i) and as;
    end generate gen_logic;
  end generate and_based;

  or_based: if style = 1 generate
    gen_logic: for i in op'range generate
      temp(i) <= (not as) or op(i);
    end generate gen_logic;
  end generate or_based;

  latch_based: if style = 2 generate
    gen_logic: process (as, op)
    begin  -- process gen_logic
      if as = '1' then
        temp <= op;
      end if;
    end process gen_logic;
  end generate latch_based;

  op_isolated <= temp;
end structural;
```

A.2 Experiment 2: An Efficient MAC Unit

A.2.1 The testbench for the MD-MAC design

```
-------------------------------------------------------------------------------
-- Title       : core design testbench
-- Project     : Multidata type MAU to be tested
-------------------------------------------------------------------------------
-- File        : test.vhd
-- Author      : Georgios Plakaris
-- Company     : Computer Systems Engineering, DTU
-- Date        : 14/02/2003
-------------------------------------------------------------------------------
-- Description :
-- A testbench for the core design
-------------------------------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
--use ieee.std_logic_signed.all;
--use ieee.std_logic_arith.all;
use ieee.math_real.all;
use ieee.numeric_std.all;
library WORK;
use WORK.design_utils.all;
use WORK.sim_utils.all;

entity TEST is

end TEST;

architecture simple of TEST is

  constant Tpw_clk : time := 10 ns;

  signal clk, rst : std_logic;
  signal A_i     : std_logic_vector(width-1 downto 0);
  signal B_i     : std_logic_vector(width-1 downto 0);
  signal opin_i  : std_logic_vector(inst_count-1 downto 0);
  signal Z_i, Z_model     : std_logic_vector(width-1 downto 0);
  signal opout_i, opout_model : std_logic_vector(inst_count-1 downto 0);
  signal ovf_i, ovf_model   : std_logic;
  constant imp_style : integer := 0;
begin
  verifier: process (ovf_model,ovf_i, Z_model, Z_i, opout_i)
  begin  -- process verifier
    if (ovf_i or ovf_model) = '0' then
      assert Z_model = Z_i report "error in result at operation" severity note;
    end if;
  end process verifier;

  BENCH: core_model
    port map (
      rst   => rst,
      clk   => clk,
      A     => A_i,
      B     => B_i,
      opin  => opin_i,
      Z     => Z_model,
      opout => opout_model,
      ovf   => ovf_model);

  DUT: core
    port map (
      rst   => rst,
      clk   => clk,
      A     => A_i,
      B     => B_i,
      opin  => opin_i,
      Z     => Z_i,
      opout => opout_i,
      ovf   => ovf_i);

  word_stimuli: bit_gen
    generic map (
        bias => 0.5)
    port map (
        clk   => clk,
        word1 => A_i,
        word2 => B_i);

  instruction_gen: opcode_gen
    port map (
        clk    => clk,
        rst    => rst,
        opcode => opin_i);

  clock_gen: clock
    generic map (
      period => 10 ns)
    port map (
      clk => clk);

  rst <= '0', '1' after 22 ns;

  result_tests: process (clk, rst)
  begin  -- process result_tests
    if clk'event and clk = '0' then  -- rising clock edge

    end if;
  end process result_tests;

end simple;
```

A.2.2 The opcode generator

```
-------------------------------------------------------------------------------
-- Title       : opcode generator
-- Project     : High power arithmetic unit to be power managed
-------------------------------------------------------------------------------
-- File        : opcode_gen.vhdl
-- Author      : Georgios Plakaris
-- Company     : Computer Systems Engineering, DTU
-- Date        : 12/11/2002
-------------------------------------------------------------------------------
-- Description :
-- Provides the sequence of instructions. LAter it should be modified to
-- represent tpical workloads of dsp processors
-------------------------------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_signed.all;
use ieee.std_logic_arith.all;
library WORK;
use WORK.design_utils.all;

entity opcode_gen is

  port (
    clk    : in  std_logic;
    rst    : in  std_logic;
    opcode : out std_logic_vector(inst_count-1 downto 0));

end opcode_gen;

architecture behavioral of opcode_gen is
  signal current_state, next_state : std_logic_vector(inst_count-1 downto 0);
  constant NMCX    : integer := 5;
  constant NMSF    : integer := 5;
  constant NMPF    : integer := 5;
  constant NNOP    : integer := 3;
  constant NMHI    : integer := 5;
  constant NMFI    : integer := 5;
  constant NMAC    : integer := 5;
  constant NMCC    : integer := 1;
  constant NACC    : integer := 5;
  signal   counter : integer;
begin  -- behavioral

  fsm: process (current_state, counter)
    variable temp_next : std_logic_vector(inst_count-1 downto 0);
  begin  -- process fsm
    temp_next := current_state;
    if counter = 0 then
      case current_state is
        when NOP_v =>
            temp_next := opid(MCX);
        when MCX_v =>
            temp_next := opid(MPF);
        when MPF_v =>
            temp_next := opid(MSF);
        when MSF_v =>
            temp_next := opid(MHI);
        when MHI_v =>
            temp_next := opid(MFI);
        when MFI_v =>
            temp_next := opid(MAC);
        when MAC_v =>
            temp_next := opid(MCC);
        when MCC_v    =>
            temp_next := opid(ACC);
        when ACC_v    =>
            temp_next := opid(NOP);
        when others     => null;
      end case;
    end if;
    next_state <= temp_next;
  end process fsm;

  state_reg: process (clk, rst)
    variable temp_count : integer;
  begin  -- process state_reg
    if rst = '0' then                   -- asynchronous reset (active low)
      current_state <= NOP_v;
      counter <= NNOP;
    elsif clk'event and clk = '0' then  -- rising clock edge
      current_state <= next_state;
      if counter = 0 then
        case current_state is
            when NOP_v =>
              counter <=  NMCX-1;
            when MCX_v =>
              counter <=  NMPF-1;
            when MPF_v =>
              counter <=  NMSF-1;
            when MSF_v =>
              counter <=  NMHI-1;
            when MHI_v =>
              counter <=  NMFI-1;
            when MFI_v =>
              counter <=  NMAC-1;
            when MAC_v =>
              counter <=  NMCC-1;
            when MCC_v =>
              counter <=  NACC-1;
            when ACC_v =>
              counter <=  NNOP-1;
          when others => null;
        end case;
      else
        counter <= counter - 1;
      end if;
    end if;
  end process state_reg;
  opcode <= current_state;

end behavioral;
```

A.2.3 The benchmark and carry-save MAC units

```
-------------------------------------------------------------------------------
-- Title       : mac_0.vhdl
-- Project     : A simple mac unit
-------------------------------------------------------------------------------
-- File        : mac_0.vhdl
-- Author      : Georgios Plakaris
-- Company     : Computer Systems Engineering, DTU
-- Date        : 27/01/2003
-------------------------------------------------------------------------------
-- Description :
-- A simple multiply acummulate unit. It infinately accumulates the product of
-- two numbers. Overflow is not an issue at this point as we are only
-- interested in the power consumed during operation. Correctness can then be
-- added by providing for overflow flags.
-------------------------------------------------------------------------------
library ieee, DWARE, DW02;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use DWARE.DWpackages.all;
use DW02.DW02_components.all;
library WORK;
use WORK.design_utils.all;

entity mac_0 is

  generic (
     w  : integer := 16;
     mw : integer := 32;
     aw : integer := 34;
     arch: integer := 0);

  port (
    rst : in  std_logic;
    clk : in  std_logic;
    A   : in  std_logic_vector(w-1 downto 0);
    B   : in  std_logic_vector(w-1 downto 0);
    clr : in  std_logic;
    sum : out std_logic_vector(w-1 downto 0);
    ovf : out std_logic);

end mac_0;

architecture basic of mac_0 is
    signal A_r, B_r    : std_logic_vector(w-1 downto 0);
    signal mult        : std_logic_vector(mw-1 downto 0);
    signal acc         : std_logic_vector(aw-1 downto 0);
    signal int_ovf, TC : std_logic;
    signal tmp_acc, C  : std_logic_vector(aw downto 0);
begin  -- basic

  non_pipelined: if arch = 1 generate

  multiplication: process (clk, rst)
    variable mult : std_logic_vector(2*w-1 downto 0);
    variable tmp_acc  : std_logic_vector(aw downto 0);
  begin  -- process func
    if rst = '0' then                   -- asynchronous reset (active low)
      A_r <= (others => '0');
      B_r <= (others => '0');
      mult := (others => '0');
      int_ovf <= '0';
      acc <= (others => '0');
    elsif clk'event and clk = '1' then  -- rising clock edge
      mult     := signed(A_r)*signed(B_r);
      tmp_acc  := signed(mult(mw-1)&mult(mw-1)&mult(mw-1)&mult)
+signed(acc(aw-1)&acc);
      if clr = '1' then
        acc <= (others => '0');
        int_ovf <= '0';
      else
        acc <= tmp_acc(aw-1 downto 0);
        int_ovf <= tmp_acc(aw)xor tmp_acc(aw-1);
      end if;
      A_r <= A;
      B_r <= B;
    end if;
  end process multiplication;

  sum <= acc(aw-1)&acc(2*mw-aw-1 downto 2*mw-aw-1-w+2);
  ovf_logic: process (int_ovf, acc)
    variable tmp_ovf : std_logic;
  begin  -- process ovf_logic
     tmp_ovf := '0';
     for i in aw-2 downto 2*mw-aw loop
       tmp_ovf := tmp_ovf or (acc(aw-1)xor acc(i));
     end loop;  -- i
     ovf <= int_ovf or tmp_ovf;
   end process ovf_logic;

  end generate non_pipelined;
  np_merged: if arch = 2 generate

  multiplication: process (clk, rst)
  begin  -- process func
    if rst = '0' then                   -- asynchronous reset (active low)
      A_r <= (others => '0');
      B_r <= (others => '0');
      int_ovf <= '0';
      acc <= (others => '0');
    elsif clk'event and clk = '1' then  -- rising clock edge
      if clr = '1' then
        acc <= (others => '0');
        int_ovf <= '0';
      else
        acc <= tmp_acc(aw-1 downto 0);
        int_ovf <= tmp_acc(aw)xor tmp_acc(aw-1);
      end if;
      A_r <= A;
      B_r <= B;
    end if;
  end process multiplication;

  sum <= acc(aw-1)&acc(2*mw-aw-1 downto 2*mw-aw-1-w+2);
  ovf_logic: process (int_ovf, acc)
    variable tmp_ovf : std_logic;
  begin  -- process ovf_logic
     tmp_ovf := '0';
     for i in aw-2 downto 2*mw-aw loop
       tmp_ovf := tmp_ovf or (acc(aw-1)xor acc(i));
     end loop;  -- i
     ovf <= int_ovf or tmp_ovf;
   end process ovf_logic;

   TC <= '1';                          -- numbers are signed
   C <= acc(aw-1)&acc;                 -- signed extended result
-- Instance of DW02_prod_sum1
   U1 : DW02_prod_sum1
     generic map ( A_width => w, B_width => w, SUM_width => aw+1)
     port map ( A => A_r, B => B_r, C => C, TC => TC, SUM => tmp_acc );

  end generate np_merged;

end basic;

-- pragma translate_off
    library DW02;
    configuration MERGED of mac_0 is
      for basic
        for np_merged
          for U1 : DW02_prod_sum1
            use configuration DW02.DW02_prod_sum1_cfg_sim;
          end for;
        end for;
      end for;
    end MERGED;
-- pragma translate_on
```

A.2.4 The pipelined MAC unit

```
-------------------------------------------------------------------------------
-- Title       : mac_lp.vhdl
-- Project     : A simple mac unit
-------------------------------------------------------------------------------
-- File        : mac_lp.vhdl
-- Author      : Georgios Plakaris
-- Company     : Computer Systems Engineering, DTU
-- Date        : 27/01/2003
-------------------------------------------------------------------------------
-- Description :
-- A simple multiply acummulate unit. It differs from mac_base in respect that
-- the ripple carry adder of the multiplier is moved to the second pipeline
-- stage.
-------------------------------------------------------------------------------
library ieee, SYNOPSYS, DWARE, DW02;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use SYNOPSYS.attributes.all;
use DWARE.DWpackages.all;
use DW02.DW02_components.all;
```

```
library WORK;
use WORK.design_utils.all;
use WORK.arith_utils.all;


entity mac_lp is

  generic (
     w    : integer := 16;
     mw   : integer := 32;
     aw   : integer := 34;
     arch : integer := 0);

  port (
    rst : in  std_logic;
    clk : in  std_logic;
    A   : in  std_logic_vector(w-1 downto 0);
    B   : in  std_logic_vector(w-1 downto 0);
    clr : in  std_logic;
    sum : out std_logic_vector(w-1 downto 0);
    ovf : out std_logic);

end mac_lp;

architecture basic of mac_lp is

  signal A_r, B_r : std_logic_vector(w-1 downto 0);
  signal mult_ST_r, mult_CT_r, mult : std_logic_vector(mw-1 downto 0);
  signal acc : std_logic_vector(aw-1 downto 0);
  signal acc_i : std_logic_vector(aw downto 0);
  signal int_ovf : std_logic;
  signal add_1, add_2 : std_logic_vector(aw downto 0);
  -- partial products
  signal PP : std_logic_vector((18)*(32)-1 downto 0);
  -- intermediate sum/carry bits
  signal ST, CT : std_logic_vector(31 downto 0);
  signal pp0, pp1 : std_logic_vector(33 downto 0);
  signal TC : std_logic;
  constant N   : integer := 3;
  constant Wv  : integer := 32;
  constant AWv : integer := Wv+3;
  signal vec_in : std_logic_vector(N*AWv-1 downto 0);
  signal Ase, Bse, Cse : std_logic_vector(AWv-1 downto 0);
begin  -- basic

  -----------------------------------------------------------------------------
  -- pipelined using synopsys components
  -----------------------------------------------------------------------------
  pipe_synopsys : if arch = 1 generate
  TC <= '1';
  Ase <= "000"&mult_ST_r;
  Bse <= "000"&mult_CT_r;
  Cse <= acc(aw-1)&acc;
  vec_in <= Ase&Bse&Cse;
  acc_i <= std_logic_vector(DWF_sum(SIGNED (vec_in), N));

  U1 : DW02_multp
    generic map (
      a_width   => 16,
      b_width   => 16,
      out_width => 34)                  -- a_width+b_width+2
    port map (
      a    => A_r,
      b    => B_r,
      tc   => TC,
      out0 => pp0,
      out1 => pp1);

    multiplication     : process (clk, rst)
    begin  -- process func
      if rst = '0' then                   -- asynchronous reset (active low)
        A_r       <= (others => '0');
        B_r       <= (others => '0');
        mult_ST_r <= (others => '0');
        mult_CT_r <= (others => '0');
        int_ovf   <= '0';
        acc       <= (others => '0');
      elsif clk'event and clk = '1' then  -- rising clock edge
        if clr = '1' then
          acc     <= (others => '0');
          int_ovf <= '0';
        else
          acc     <= acc_i(aw-1 downto 0);
          int_ovf <= acc_i(aw)xor acc_i(aw-1);
        end if;
        mult_CT_r <= pp0(mw-1 downto 0);
        mult_ST_r <= pp1(mw-1 downto 0);

        A_r       <= A;
        B_r       <= B;
      end if;
    end process multiplication;

    sum <= acc(aw-1)&acc(2*mw-aw-1 downto 2*mw-aw-1-w+2);
    ovf_logic          : process (int_ovf, acc)
      variable tmp_ovf : std_logic;
    begin  -- process ovf_logic
      tmp_ovf   := '0';
      for i in aw-2 downto 2*mw-aw loop
        tmp_ovf := tmp_ovf or (acc(aw-1)xor acc(i));
      end loop;  -- i
      ovf <= int_ovf or tmp_ovf;
    end process ovf_logic;
  end generate pipe_synopsys;
end basic;


-- pragma translate_off
    library DW02;
    configuration PIPED of mac_lp is
      for basic
        for pipe_synopsys
          for U1 : DW02_multp
            use configuration DW02.DW02_multp_cfg_sim;
          end for;
        end for;
      end for;
    end PIPED;
-- pragma translate_on
```

A.3 Experiment 3: Multi-datatype MAC unit (MD-MAC)

A.3.1 The top level SPLIT-MD-MAC and MD-MAC architectures

```
-------------------------------------------------------------------------------
-- Title      : core.vhdl
-- Project    : A multi-datatype MAU unit
-------------------------------------------------------------------------------
-- File       : core.vhdl
-- Author     : Georgios Plakaris
-- Company    : Computer Systems Engineering, DTU
-- Date       : 12/02/2003
-------------------------------------------------------------------------------
-- Description :
-- The core performs several multiply based operations as indicated by the
-- instruction set.
-------------------------------------------------------------------------------

library ieee, SYNOPSYS, DW01, DW02, DWARE, WORK;
use WORK.design_utils.all;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use SYNOPSYS.attributes.all;
use DWARE.DWpackages.all;
use DW01.DW01_components.all;
use DW02.DW02_components.all;

entity core is
  port (
    rst   : in  std_logic;
    clk   : in  std_logic;
    A     : in  std_logic_vector(width-1 downto 0);
    B     : in  std_logic_vector(width-1 downto 0);
    opin  : in  std_logic_vector(inst_count-1 downto 0);
    Z     : out std_logic_vector(width-1 downto 0);
    opout : out std_logic_vector(inst_count-1 downto 0);
    ovf   : out std_logic);

end core;
```

```
architecture structural of core is
 signal HH, HL, LH, LL, result, Z_tmp : std_logic_vector(width-1 downto 0);
 signal accum : std_logic_vector(width-1 downto 0);
 signal opcode : std_logic_vector(inst_count-1 downto 0);
 signal overflow : std_logic;
 constant style : integer := 1;
begin  -- structural

  input_registers: iregs
--     generic map (
--       style => style)
    port map (
      rst   => rst,
      clk   => clk,
      A     => A,
      B     => B,
      opin  => opin,
      opout => opcode,
      HH    => HH,
      LL    => LL,
      HL    => HL,
      LH    => LH);

  output_registers: oregs
    port map (
      rst    => rst,
      clk    => clk,
      opin   => opcode,
      ovfin  => overflow,
      Zin    => result,
      accum  => accum,
      opout  => opout,
      Zout   => Z,
      ovfout => ovf);

  MAU : design
    port map (
      rst   => rst,
      clk   => clk,
      op    => opcode,
      HH    => HH,
      HL    => HL,
      LH    => LH,
      LL    => LL,
      Z     => result,
      accum => accum,
      ovf   => overflow);
end structural;

configuration status of core is

  for structural
    for input_registers : iregs
      use entity work.iregs(rtl)
        generic map (1);
    end for;
    for output_registers : oregs
      use entity work.oregs(rtl);
    end for;
    for MAU : design
      use entity work.design(structural);
    end for;
  end for;

end status;


architecture split of core is
 signal HH, HL, LH, LL, result, Z_tmp : std_logic_vector(width-1 downto 0);
 signal accum, res_MAU, res_MFI : std_logic_vector(width-1 downto 0);
 signal opcode : std_logic_vector(inst_count-1 downto 0);
 signal overflow, ovf_MAU, ovf_MFI : std_logic;
 signal en_mult32, en_mult32_pipe : std_logic;
 constant style : integer := 1;
begin  -- structural

  input_registers: iregs
    port map (
      rst   => rst,
      clk   => clk,
      A     => A,
      B     => B,
      opin  => opin,
      opout => opcode,
      HH    => HH,
      LL    => LL,
      HL    => HL,
      LH    => LH);

  merge_logic: process(en_mult32_pipe, res_MFI, res_MAU, ovf_MAU, ovf_MFI)
  begin  -- process merge_logic
    case en_mult32_pipe is
      when '1' =>
        result <= res_MFI;
      when others =>
        result <= res_MAU;
    end case;
    overflow <= (ovf_MAU and (not en_mult32_pipe))or (ovf_MFI and en_mult32_pipe);
  end process merge_logic;

  output_registers: oregs
    port map (
      rst    => rst,
      clk    => clk,
      opin   => opcode,
      ovfin  => overflow,
      Zin    => result,
      accum  => accum,
      opout  => opout,
      Zout   => Z,
      ovfout => ovf);

  en_mult32 <= opin(MFI);
  en_mult32_pipe <= opcode(MFI);
  multiplier32: mult
    port map (
      rst => rst,
      clk => clk,
      en  => en_mult32,
      op1 => A,
      op2 => B,
      res => res_MFI,
      ovf => ovf_MFI);

  MAU : design
    port map (
      rst   => rst,
      clk   => clk,
      op    => opcode,
      HH    => HH,
      HL    => HL,
      LH    => LH,
      LL    => LL,
      Z     => res_MAU,
      accum => accum,
      ovf   => ovf_MAU);
end split;

configuration status_split of core is
  for split
    for input_registers : iregs
      use entity work.iregs(rtl)
        generic map (2);
    end for;
    for output_registers : oregs
      use entity work.oregs(rtl);
    end for;
    for MAU : design
      use entity work.design(split);
    end for;
  end for;

end status_split;
```

A.3.2 The top level MD-MAC NCS architecture

```
-------------------------------------------------------------------------------
-- Title      : core_ncs_shared.vhdl
-- Project    : A multi-datatype MAU unit
-------------------------------------------------------------------------------
-- File       : core_ncs_shared.vhdl
-- Author     : Georgios Plakaris
-- Company    : Computer Systems Engineering, DTU
-- Date       : 12/02/2003
```

```
-------------------------------------------------------------------------------
-- Description :
-- The core performs several multiply based operations as indicated by the
-- instruction set.
-------------------------------------------------------------------------------

library ieee, SYNOPSYS, DW01, DW02, DWARE, WORK;
use WORK.design_utils.all;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use SYNOPSYS.attributes.all;
use DWARE.DWpackages.all;
use DW01.DW01_components.all;
use DW02.DW02_components.all;

entity core_ncs_shared is
  port (
    rst   : in  std_logic;
    clk   : in  std_logic;
    A     : in  std_logic_vector(width-1 downto 0);
    B     : in  std_logic_vector(width-1 downto 0);
    opin  : in  std_logic_vector(inst_count-1 downto 0);
    Z     : out std_logic_vector(width-1 downto 0);
    opout : out std_logic_vector(inst_count-1 downto 0);
    ovf   : out std_logic);

end core_ncs_shared;

architecture structural of core_ncs_shared is
 signal HH, HL, LH, LL, result, Z_tmp : std_logic_vector(width-1 downto 0);
 signal accum : std_logic_vector(width-1 downto 0);
 signal opcode : std_logic_vector(inst_count-1 downto 0);
 signal overflow : std_logic;
 constant style : integer := 1;
begin  -- structural

  input_registers: iregs
--     generic map (
--       style => style)
    port map (
      rst   => rst,
      clk   => clk,
      A     => A,
      B     => B,
      opin  => opin,
      opout => opcode,
      HH    => HH,
      LL    => LL,
      HL    => HL,
      LH    => LH);

  output_registers: oregs
    port map (
      rst    => rst,
      clk    => clk,
      opin   => opcode,
      ovfin  => overflow,
      Zin    => result,
      accum  => accum,
      opout  => opout,
      Zout   => Z,
      ovfout => ovf);

  MAU : design_ncs_shared
    port map (
      rst   => rst,
      clk   => clk,
      op    => opcode,
      HH    => HH,
      HL    => HL,
      LH    => LH,
      LL    => LL,
      Z     => result,
      accum => accum,
      ovf   => overflow);
end structural;

configuration status of core_ncs_shared is

  for structural
    for input_registers : iregs
      use entity work.iregs(rtl)
        generic map (1);
    end for;
    for output_registers : oregs
      use entity work.oregs(rtl);
    end for;
    for MAU : design_ncs_shared
      use entity work.design_ncs_shared(structural);
    end for;
  end for;

end status;
```

A.3.3 The intput registers

```
-------------------------------------------------------------------------------
-- Title      : iregs.vhdl
-- Project    : A multi-datatype MAU unit
-------------------------------------------------------------------------------
-- File       : iregs.vhdl
-- Author     : Georgios Plakaris
-- Company    : Computer Systems Engineering, DTU
-- Date       : 12/02/2003
-------------------------------------------------------------------------------
-- Description :
-- The input registers of the core design
-------------------------------------------------------------------------------

library ieee, SYNOPSYS, DW01, DW02, DWARE, WORK;
use WORK.design_utils.all;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use SYNOPSYS.attributes.all;
use DWARE.DWpackages.all;
use DW01.DW01_components.all;
use DW02.DW02_components.all;

entity iregs is

  generic (
    style : integer:= 2);

  port (
    rst   : in  std_logic;
    clk   : in  std_logic;
    A     : in  std_logic_vector(width-1 downto 0);
    B     : in  std_logic_vector(width-1 downto 0);
    opin  : in  std_logic_vector(inst_count-1 downto 0);
    opout : out std_logic_vector(inst_count-1 downto 0);
    HH    : out std_logic_vector(width-1 downto 0);
    LL    : out std_logic_vector(width-1 downto 0);
    HL    : out std_logic_vector(width-1 downto 0);
    LH    : out std_logic_vector(width-1 downto 0));

end iregs;

architecture rtl of iregs is
 signal hh_i, ll_i, lh_i, hl_i, A_r, B_r : std_logic_vector(width-1 downto 0);
 signal hh_en, hl_lh_en,  ll_en : std_logic;
begin  -- rtl
-------------------------------------------------------------------------------
-- Operand isolated input registers by splited pipes
-------------------------------------------------------------------------------
  isol_split: if style = 2 generate

    -- output connections
    isol_regs: process (clk, rst)
    begin  -- process isol_regs
      if rst = '0' then                 -- asynchronous reset (active low)
        HH    <= (others => '0');
        HL    <= (others => '0');
        LH    <= (others => '0');
        LL    <= (others => '0');
        opout <= (others => '0');
      elsif clk'event and clk = '1' then  -- rising clock edge
        opout <= opin;
        if hh_en = '1' then
          HH <= hh_i;
        end if;
        if hl_lh_en = '1' then
          HL <= hl_i;
        end if;
        if hl_lh_en = '1' then
```

```
          LH <= lh_i;
        end if;
        if ll_en = '1' then
          LL <= ll_i;
        end if;
      end if;
    end process isol_regs;

    -- input connection
    hh_i <= upper(A)&upper(B);
    hl_i <= upper(A)&lower(B);
    lh_i <= lower(A)&upper(B);
    ll_i <= lower(A)&lower(B);

    -- control logic
    ll_en <= not (opin(NOP) or opin(MFI));
    hh_en <= opin(MPF)or opin(MCX);
    hl_lh_en <= opin(MCX);
  end generate isol_split;

end rtl;
```

A.3.4 The output registers

```
-------------------------------------------------------------------------------
-- Title       : oregs.vhdl
-- Project     : A multi-datatype MAU unit
-------------------------------------------------------------------------------
-- File        : oregs.vhdl
-- Author      : Georgios Plakaris
-- Company     : Computer Systems Engineering, DTU
-- Date        : 12/02/2003
-------------------------------------------------------------------------------
-- Description :
-- The output registers of the core design
-------------------------------------------------------------------------------

library ieee, SYNOPSYS, DW01, DW02, DWARE, WORK;
use WORK.design_utils.all;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use SYNOPSYS.attributes.all;
use DWARE.DWpackages.all;
use DW01.DW01_components.all;
use DW02.DW02_components.all;

entity oregs is

  port (
    rst    : in  std_logic;
    clk    : in  std_logic;
    opin   : in  std_logic_vector(inst_count-1 downto 0);
    ovfin  : in  std_logic;
    Zin    : in  std_logic_vector(width-1 downto 0);
    accum  : in  std_logic_vector(width-1 downto 0);
    opout  : out std_logic_vector(inst_count-1 downto 0);
    Zout   : out std_logic_vector(width-1 downto 0);
    ovfout : out std_logic);

end oregs;

architecture rtl of oregs is
  signal to_output : std_logic_vector(width-1 downto 0);
begin  -- structural

  regs_out : process (clk, rst)
  begin  -- process regs_out
    if rst = '0' then                   -- asynchronous reset (active low)
      Zout   <= (others => '0');
      opout  <= (others => '0');
      ovfout <= '0';
    elsif clk'event and clk = '1' then  -- rising clock edge
      ovfout <= ovfin;
      opout  <= opin;
      if opin(NOP) = '0' then
        Zout <= to_output;
      end if;
    end if;
  end process regs_out;
  to_output <= Zin when opin(ACC) = '0' else accum;

end rtl;
```

A.3.5 The SPLIT-MD-MAC and MD-MAC designs

```
-------------------------------------------------------------------------------
-- Title      : design.vhdl
-- Project    : A multi-datatype MAU unit
-------------------------------------------------------------------------------
-- File       : design.vhdl
-- Author     : Georgios Plakaris
-- Company    : Computer Systems Engineering, DTU
-- Date       : 12/02/2003
-------------------------------------------------------------------------------
-- Description :
-- The processing unit of the core design. A clearly combinatorial circuit,
-- apart from some latches to gate control signals and the accumulator
-------------------------------------------------------------------------------

library ieee, SYNOPSYS, DW01, DW02, DWARE, WORK;
use WORK.design_utils.all;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use SYNOPSYS.attributes.all;
use DWARE.DWpackages.all;
use DW01.DW01_components.all;
use DW02.DW02_components.all;

entity design is

  port (
    rst    : in  std_logic;
    clk    : in  std_logic;
    op     : in  std_logic_vector(inst_count-1 downto 0);
    HH, HL : in std_logic_vector(width-1 downto 0);
    LH, LL : in std_logic_vector(width-1 downto 0);
    Z      : out std_logic_vector(width-1 downto 0);
    accum  : out std_logic_vector(width-1 downto 0);  -- 34bit accumulator
    ovf    : out std_logic);

end design;

architecture structural of design is
-- parameters for the multp trees
constant a_width, b_width : integer := half_width;
constant multp_width      : integer := a_width+b_width+2;
signal hh_sum, hh_carry : std_logic_vector(multp_width-1 downto 0);
signal hl_sum, hl_carry : std_logic_vector(multp_width+1 downto 0);
signal lh_sum, lh_carry : std_logic_vector(multp_width+1 downto 0);
signal ll_sum, ll_carry, ll_carry_inv, ll_sum_inv : std_logic_vector(multp_width-1 downto 0);
signal TC_hh, TC_hl, TC_lh, TC_ll : std_logic;
signal hh_A, hh_B, ll_A, ll_B, ll_B_inv : std_logic_vector(half_width-1 downto 0);
signal hl_A, hl_B, lh_A, lh_B  : std_logic_vector(half_width downto 0);
--signal hl_A, hl_B, lh_A, lh_B  : std_logic_vector(half_width-1 downto 0);
attribute implementation : string;
attribute implementation of hh_pp, hl_pp, lh_pp, ll_pp : label is "nbw";
-- parameters for the vector trees
constant vec4 : integer := 4;
constant vec6 : integer := 8;
signal vec4_in : std_logic_vector(vec4*(width+3)-1 downto 0);
signal vec4_in0, vec4_in1   : std_logic_vector(width+2 downto 0);
signal vec4_out0, vec4_out1 : std_logic_vector(width+2 downto 0);
signal vec6_in : std_logic_vector(vec6*(51)-1 downto 0);
signal vec6_out0, vec6_out1 : std_logic_vector(50 downto 0);
signal hh_sum_ext, hh_carry_ext : std_logic_vector(width+2 downto 0);
signal ll_sum_ext, ll_carry_ext : std_logic_vector(width+2 downto 0);
signal ext_accum                : std_logic_vector(width+2 downto 0);
-- parameters for the cp adders for the vec trees
signal vec4_res : std_logic_vector(width+3 downto 0);
signal vec6_res : std_logic_vector(50 downto 0);
signal vec6_CI, vec6_CO, vec4_CI : std_logic;
signal tmp_vec6_in0, tmp_vec6_in1 : std_logic_vector(half_width+1 downto 0);
signal tmp_vec6_in6, tmp_vec6_in7 : std_logic_vector(49 downto 0);
signal vec6_in0, vec6_in1       : std_logic_vector(50 downto 0);
signal vec6_in2, vec6_in3       : std_logic_vector(50 downto 0);
signal vec6_in4, vec6_in5       : std_logic_vector(50 downto 0);
signal vec6_in6, vec6_in7       : std_logic_vector(50 downto 0);
-- parameters for the adders for the MFI instruction
signal MFI_low_16 : std_logic_vector(half_width-1 downto 0);
signal MFI_high_res : std_logic_vector(half_width-1 downto 0);
signal MFI_low_16_CO : std_logic;
signal MFI_low_A, MFI_low_B   : std_logic_vector(half_width-1 downto 0);
signal MPF_high : std_logic_vector(width+1 downto 0);
-- parameters for the steering logic
signal MSF_res, MPF_res, MCX_res : std_logic_vector(width-1 downto 0);
signal MHI_res, MFI_res : std_logic_vector(width-1 downto 0);
-- parameters for the accumulator
attribute sync_set_reset_local of accumulator : label is  "accum_en" ;
signal accum_en : std_logic;
signal acc_res : std_logic_vector(width+3 downto 0);
-- parameters for the overflow logic
signal MHI_ovf_flag, MFI_ovf_flag : std_logic;
signal overflow_flags : std_logic_vector(inst_count-1 downto 0);
begin  -- structural
  -- signed number selector
  TC_hh <= '1';
  TC_hl <= '1';
  TC_lh <= '1';
  TC_ll <= not op(MFI);

  -- overflow control
  MHI_ovf_flag <= ((not vec4_res(31)) and det_one(vec4_res(30 downto 15)))or
                  ((vec4_res(31)) and det_zero(vec4_res(30 downto 15)));

  MFI_ovf_flag <= ((not vec6_res(47)) and det_one(vec6_res(46 downto half_width-1)))or
                  ((vec6_res(47)) and det_zero(vec6_res(46 downto half_width-1)));

  overflow_control: process(op, vec4_res, vec6_res, MPF_high, MHI_ovf_flag, MFI_ovf_flag, ll_A, ll_B, acc_res)
    variable ovf_vec : std_logic_vector(inst_count-1 downto 0);
    variable MPF_ovf_h, MPF_ovf_l, MCX_ovf_re, MCX_ovf_im : std_logic;
    variable MAC_ovf_flag, ACC_ovf_flag, SA, SB : std_logic;
  begin  -- process overflow_control
    ovf_vec := (others => '0');
    MPF_ovf_l := (vec4_res(31) xor vec4_res(30))or
                 (vec4_res(31) xor vec4_res(32))or
                 (vec4_res(31) xor vec4_res(33))or
                 (vec4_res(31) xor vec4_res(34))or
                 (vec4_res(31) xor vec4_res(35));
    MPF_ovf_h := (MPF_high(31)xor MPF_high(30))or
                 (MPF_high(31)xor MPF_high(32))or
                 (MPF_high(31)xor MPF_high(33));
    MCX_ovf_re := (vec4_res(32)xor vec4_res(31))or
                  (vec4_res(32)xor vec4_res(30))or
                  (vec4_res(32)xor vec4_res(33))or
                  (vec4_res(32)xor vec4_res(34))or
                  (vec4_res(32)xor vec4_res(35));
    MCX_ovf_im := (vec6_res(32)xor vec6_res(31))or
                  (vec6_res(32)xor vec6_res(30))or
                  (vec6_res(32)xor vec6_res(33))or
                  (vec6_res(32)xor vec6_res(34))or
                  (vec6_res(32)xor vec6_res(35));
    ovf_vec(MSF) := op(MSF) and MPF_ovf_l;
    ovf_vec(MPF) := op(MPF) and (MPF_ovf_h or MPF_ovf_l);
    ovf_vec(MCX) := op(MCX) and (MCX_ovf_re or MCX_ovf_im);
    ovf_vec(MHI) := op(MHI) and MHI_ovf_flag;
    ovf_vec(MFI) := op(MFI) and MFI_ovf_flag;
    SA := ll_A(half_width-1)xor ll_B(half_width-1);
    SB := acc_res(width+1);
    MAC_ovf_flag := (SA xnor SB) and (SA xor vec4_res(34));
    -- to be fixed
    ACC_ovf_flag := (vec4_res(31)xor vec4_res(32))or
                    (vec4_res(31)xor vec4_res(33))or
                    (vec4_res(31)xor vec4_res(34));
    ovf_vec(MAC) := op(MAC) and MAC_ovf_flag;
    ovf_vec(MCC) := '0';
    ovf_vec(ACC) := op(ACC) and ACC_ovf_flag;
    overflow_flags <= ovf_vec;
  end process overflow_control;
  -- connect overflow flag to output;
  ovf <= det_one(overflow_flags);

  -- accumulator
  accum_en <= op(MAC)or op(ACC) or op(MCC);
  accumulator: process (clk, rst)
  begin  -- process accumulator
    if rst = '0' then                   -- asynchronous reset (active low)
      acc_res <= (others => '0');
    elsif clk'event and clk = '1' then  -- rising clock edge
      if accum_en = '1' then
        acc_res <= vec4_res;
      end if;
    end if;
  end process accumulator;
```

```
-- create results
MHI_res <= conv_std_logic_vector(0,16)&vec4_res(31)&vec4_res(half_width-2 downto 0);
MFI_res <= vec6_res(47)&vec6_res(half_width-2 downto 0)&MFI_low_16;
MSF_res <= conv_std_logic_vector(0,16)&
                       vec4_res(31)&vec4_res(width-3 downto half_width-1);
MPF_res <= MPF_high(31)&MPF_high(width-3 downto half_width-1)
           &vec4_res(31)&vec4_res(width-3 downto half_width-1);
MCX_res <= vec4_res(width)&vec4_res(width-3 downto half_width-1)
             &vec6_res(width)&vec6_res(width-3 downto half_width-1);
accum <= acc_res(width+1)&acc_res(width-2 downto 0);
mum <= vec6_res(47 downto 0)&MFI_low_16;
-- steering output multiplier
output_mux: process(op, MSF_res, MPF_res, MCX_res, MHI_res, MFI_res)
  variable MSF_mux, MPF_mux, MCX_mux : std_logic_vector(width-1 downto 0);
  variable MHI_mux, MFI_mux : std_logic_vector(width-1 downto 0);
begin  -- process output_mux
  for i in width-1 downto 0 loop
    MSF_mux(i) := MSF_res(i)and op(MSF);
    MPF_mux(i) := MPF_res(i)and op(MPF);
    MCX_mux(i) := MCX_res(i)and op(MCX);
    MHI_mux(i) := MHI_res(i)and op(MHI);
    MFI_mux(i) := MFI_res(i)and op(MFI);
  end loop;  -- i
  Z <= MCX_mux or ((MSF_mux or MPF_mux)or (MFI_mux or MHI_mux));
end process output_mux;

-- adder for the lower 16 bits of the MFI instruction
MFI_low_A <= ll_sum(half_width-1 downto 0);
MFI_low_B <= ll_carry(half_width-1 downto 0);
MFI_low_16_cpa: process (MFI_low_A, MFI_low_B)
  variable tmp_sum : unsigned(half_width downto 0);
  variable A_ext, B_ext : std_logic_vector(half_width downto 0);
begin  -- process MFI_low_16_cpa
  A_ext := '0'&MFI_low_A;
  B_ext := '0'&MFI_low_B;
  tmp_sum := unsigned(A_ext) + unsigned(B_ext);  -- pragma label MFI_low_cpa
  MFI_low_16    <= std_logic_vector(tmp_sum(half_width-1 downto 0));
  MFI_low_16_CO <= tmp_sum(half_width);
end process MFI_low_16_cpa;
vec6_CI <= MFI_low_16_CO and op(MFI);  --propagate carry only in MFI instruction

-- adder for the high part of MPF
MPF_high_part_adder: process (hh_sum, hh_carry)
  variable sum : signed(width+1 downto 0);
begin  -- process MFI_high_cond_sum
  sum := signed(hh_sum) + signed(hh_carry);  -- pragma label MPF_high_add
  MPF_high <= std_logic_vector(sum);
end process MPF_high_part_adder;
vec4_CI <= op(MCX);
-- propagate adder for vec4_tree
vec4_cpa: process (vec4_out0, vec4_out1, vec4_CI)
constant r0 : resource := 0;
attribute map_to_module of r0  : constant is "DW01_add";
attribute implementation of r0 : constant is "bk";
attribute ops of r0 : constant is "cpa4";
  variable vec4_CI_v  : signed(width+3 downto 0);
  variable vec4_res_i : signed(width+3 downto 0);
  variable op1, op2 : std_logic_vector(width+3 downto 0);
begin  -- process vec4_cpa
  vec4_CI_v := (others => '0');
  vec4_CI_v(0) := vec4_CI;
  op1 := (not vec4_out0(width+2))&vec4_out0;
  op2 := '1'&vec4_out1;
  vec4_res_i := vec4_CI_v + signed(op1) + signed(op2);  -- pragma label cpa4
  vec4_res   <= std_logic_vector(vec4_res_i);
end process vec4_cpa;

vec6_cpa: process(vec6_out0, vec6_out1, vec6_CI)
constant r1 : resource := 0;
attribute map_to_module of r1  : constant is "DW01_add";
attribute implementation of r1 : constant is "bk";
attribute ops of r1 : constant is "cpa6";
  variable vec6_CI_v : unsigned(49 downto 0);
  variable vec6_res_v : unsigned(50 downto 0);
begin  -- process vec6_cpa
  vec6_CI_v := (others => '0');
  vec6_CI_v(0) := vec6_CI;
  vec6_res_v := vec6_CI+unsigned(vec6_out0)+unsigned(vec6_out1);  -- pragma label cpa6
  vec6_res <= std_logic_vector(vec6_res_v);
end process vec6_cpa;

-- instantiation of trees for the vector adders
```

```
vec4_tree: DW02_tree
  generic map (
    num_inputs  => vec4,
    input_width => width+3) -- extended intermediate range of accumulator
  port map (
    INPUT => vec4_in,
    OUT0  => vec4_out0,
    OUT1  => vec4_out1);

vec6_tree: DW02_tree
  generic map (
    num_inputs  => vec6,
    input_width => 51)
  port map (
    INPUT => vec6_in,
    OUT0  => vec6_out0,
    OUT1  => vec6_out1);

-- input connections for the vec4_tree
hh_sum_ext   <= sgn_ext(hh_sum, 1);
hh_carry_ext <= sgn_ext(hh_carry, 1);
invert_sum_carry: for i in ll_sum'range generate
  ll_sum_inv(i)   <= ll_sum(i)xor op(MCX);
  ll_carry_inv(i) <= ll_carry(i)xor op(MCX);
end generate invert_sum_carry;
ll_sum_ext   <= sgn_ext(ll_sum_inv, 1);
ll_carry_ext <= sgn_ext(ll_carry_inv, 1);
ext_accum    <= acc_res(width+1)&acc_res(width+1 downto 0);
vec4_in <= vec4_in0&vec4_in1&ll_sum_ext&ll_carry_ext;

vec4_tree_inputs: process (hh_sum_ext, hh_carry_ext, ext_accum, op)
  variable ctrl_in0, ctrl_in1, reset_in1 : std_logic;
  variable vec4_in1_v : std_logic_vector(width+2 downto 0);
begin  -- process vec4_tree_inputs
  ctrl_in0 := op(MCX);
  ctrl_in1 := op(MCX);
  reset_in1 := not(op(MSF)or op(MPF)or op(MHI)or op(MFI)or op(MCC));
  for i in hh_sum_ext'range loop
    vec4_in0(i) <= hh_sum_ext(i)and (ctrl_in0);
  end loop;  -- i
  case ctrl_in0 is
    when '0' =>
      vec4_in1_v := ext_accum;
    when others =>
      vec4_in1_v := hh_carry_ext(width+2 downto 1)&op(MCX);
  end case;
  for j in vec4_in1_v'range loop
    vec4_in1(j) <= vec4_in1_v(j)and reset_in1;
  end loop;  -- j
end process vec4_tree_inputs;

-- input connections for the vec6_tree
tmp_vec6_in0 <=  ll_sum(width+1 downto half_width);
tmp_vec6_in1 <=  ll_carry(width+1 downto half_width);
tmp_vec6_in6 <=  hh_sum(width+1 downto 0)&conv_std_logic_vector(0,16);
tmp_vec6_in7 <=  hh_carry(width+1 downto 0)&conv_std_logic_vector(0,16);
vec6_tree_inputs: process (tmp_vec6_in0, tmp_vec6_in1, tmp_vec6_in6, tmp_vec6_in7,op)
  variable ctrl_vec6              : std_logic;
  variable vec6_in0_i, vec6_in1_i : std_logic_vector(50 downto 0);
  variable vec6_in6_i, vec6_in7_i : std_logic_vector(50 downto 0);
begin  -- process vec6_tree_inputs
  ctrl_vec6  := op(MFI);
  vec6_in0_i := sgn_ext(tmp_vec6_in0,33);
  vec6_in1_i := sgn_ext(tmp_vec6_in1,33);
  vec6_in6_i := sgn_ext(tmp_vec6_in6,1);
  vec6_in7_i := sgn_ext(tmp_vec6_in7,1);
  for j in vec6_in0_i'range loop
    vec6_in0(j) <= vec6_in0_i(j)and ctrl_vec6;
    vec6_in1(j) <= vec6_in1_i(j)and ctrl_vec6;
    vec6_in6(j) <= vec6_in6_i(j)and ctrl_vec6;
    vec6_in7(j) <= vec6_in7_i(j)and ctrl_vec6;
  end loop;  -- j
end process vec6_tree_inputs;
vec6_in2 <= sgn_ext(hl_sum,15);
vec6_in3 <= sgn_ext(hl_carry,15);
vec6_in4 <= sgn_ext(lh_sum,15);
vec6_in5 <= sgn_ext(lh_carry,15);
vec6_in  <= vec6_in0&vec6_in1&vec6_in2&vec6_in3&vec6_in4&vec6_in5&vec6_in6&vec6_in7;

-- partial product generators instantiation
hh_A <= upper(HH);
hh_B <= lower(HH);
hh_pp: DW02_multp
```

```
    generic map (
      a_width   => a_width,
      b_width   => b_width,
      out_width => multp_width)
    port map (
      a    => hh_A,
      b    => hh_B,
      tc   => TC_hh,
      out0 => hh_sum,
      out1 => hh_carry);

  fix_inputs_hl: process (HL, op)
    variable signB : std_logic;
  begin  -- process fix_inputs_hl
   signB := HL(half_width-1);
   if op(MFI) = '1' then
     signB := '0';
   end if;
   hl_A <= HL(width-1)&upper(HL);
   hl_B <= signB&lower(HL);
  end process fix_inputs_hl;

  hl_pp: DW02_multp
    generic map (
      a_width   => a_width+1,
      b_width   => b_width+1,
      out_width => multp_width+2)
    port map (
      a    => hl_A,
      b    => hl_B,
      tc   => TC_hl,
      out0 => hl_sum,
      out1 => hl_carry);

  fix_inputs_lh: process (LH, op)
    variable signA : std_logic;
  begin  -- process fix_inputs_hl
   signA := LH(width-1);
   if op(MFI) = '1' then
     signA := '0';
   end if;
    lh_A <= signA&upper(LH);
    lh_B <= LH(half_width-1)&lower(LH);
  end process fix_inputs_lh;

  lh_pp: DW02_multp
    generic map (
      a_width   => a_width+1,
      b_width   => b_width+1,
      out_width => multp_width+2)
    port map (
      a    => lh_A,
      b    => lh_B,
      tc   => TC_lh,
      out0 => lh_sum,
      out1 => lh_carry);

  ll_B <= lower(LL);
  ll_A <= upper(LL);
  ll_pp: DW02_multp
    generic map (
      a_width   => a_width,
      b_width   => b_width,
      out_width => multp_width)
    port map (
      a    => ll_A,
      b    => ll_B,
      tc   => TC_ll,
      out0 => ll_sum,
      out1 => ll_carry);

end structural;

-- configure simulation models for the dware components
-- pragma translate_off
library DW02;
configuration sim_models of design is
  for structural
    for hh_pp, hl_pp, lh_pp, ll_pp : DW02_multp
      use configuration DW02.DW02_multp_cfg_sim;
    end for;
  end for;
end sim_models;
```

```
-- pragma translate_on


architecture split of design is
-- parameters for the multp trees
constant a_width, b_width : integer := half_width;
constant multp_width      : integer := a_width+b_width+2;
signal hh_sum, hh_carry : std_logic_vector(multp_width-1 downto 0);
signal hl_sum, hl_carry : std_logic_vector(multp_width-1 downto 0);
signal lh_sum, lh_carry : std_logic_vector(multp_width-1 downto 0);
signal ll_sum, ll_carry, ll_carry_inv, ll_sum_inv : std_logic_vector(multp_width-1 downto 0);
signal TC_hh, TC_hl, TC_lh, TC_ll : std_logic;
signal hh_A, hh_B, ll_A, ll_B : std_logic_vector(half_width-1 downto 0);
signal hl_A, hl_B, lh_A, lh_B  : std_logic_vector(half_width-1 downto 0);
attribute implementation : string;
attribute implementation of hh_pp, hl_pp, lh_pp, ll_pp : label is "nbw";
-- parameters for the vector trees
constant vec4 : integer := 4;
constant vec6 : integer := 8;
signal vec4_in : std_logic_vector(vec4*(width+3)-1 downto 0);
signal vec4_in0, vec4_in1   : std_logic_vector(width+2 downto 0);
signal vec4_out0, vec4_out1 : std_logic_vector(width+2 downto 0);
signal vec6_in : std_logic_vector(vec4*(34)-1 downto 0);
signal vec6_out0, vec6_out1 : std_logic_vector(33 downto 0);
signal hh_sum_ext, hh_carry_ext : std_logic_vector(width+2 downto 0);
signal ll_sum_ext, ll_carry_ext : std_logic_vector(width+2 downto 0);
signal ext_accum                : std_logic_vector(width+2 downto 0);
-- parameters for the cp adders for the vec trees
signal vec4_res : std_logic_vector(width+3 downto 0);
signal vec6_res : std_logic_vector(33 downto 0);
signal vec4_CI : std_logic;
signal MPF_high : std_logic_vector(width+1 downto 0);
-- parameters for the steering logic
signal MSF_res, MPF_res, MCX_res : std_logic_vector(width-1 downto 0);
signal MHI_res : std_logic_vector(width-1 downto 0);
-- parameters for the accumulator
attribute sync_set_reset_local of accumulator : label is  "accum_en" ;
signal accum_en : std_logic;
signal acc_res : std_logic_vector(width+3 downto 0);
-- parameters for the overflow logic
signal MHI_ovf_flag : std_logic;
signal overflow_flags : std_logic_vector(inst_count-1 downto 0);
begin  -- structural
  -- signed number selector
  TC_hh <= '1';
  TC_hl <= '1';
  TC_lh <= '1';
  TC_ll <= not op(MFI);

  -- overflow control
  MHI_ovf_flag <= ((not vec4_res(31)) and det_one(vec4_res(30 downto 15)))or
                   ((vec4_res(31)) and det_zero(vec4_res(30 downto 15)));

  overflow_control: process(op, vec4_res, vec6_res, MPF_high, MHI_ovf_flag,  ll_A, ll_B, acc_res)
    variable ovf_vec : std_logic_vector(inst_count-1 downto 0);
    variable MPF_ovf_h, MPF_ovf_l, MCX_ovf_re, MCX_ovf_im : std_logic;
    variable MAC_ovf_flag, ACC_ovf_flag, SA, SB : std_logic;
  begin  -- process overflow_control
    ovf_vec := (others => '0');
    MPF_ovf_l := (vec4_res(31) xor vec4_res(30))or
                 (vec4_res(31) xor vec4_res(32))or
                 (vec4_res(31) xor vec4_res(33))or
                 (vec4_res(31) xor vec4_res(34))or
                 (vec4_res(31) xor vec4_res(35));
    MPF_ovf_h := (MPF_high(31)xor MPF_high(30))or
                 (MPF_high(31)xor MPF_high(32))or
                 (MPF_high(31)xor MPF_high(33));
    MCX_ovf_re := (vec4_res(32)xor vec4_res(31))or
                  (vec4_res(32)xor vec4_res(30))or
                  (vec4_res(32)xor vec4_res(33))or
                  (vec4_res(32)xor vec4_res(34))or
                  (vec4_res(32)xor vec4_res(35));
    MCX_ovf_im := (vec6_res(32)xor vec6_res(31))or
                  (vec6_res(32)xor vec6_res(30))or
                  (vec6_res(32)xor vec6_res(33));
    ovf_vec(MSF) := op(MSF) and MPF_ovf_l;
    ovf_vec(MPF) := op(MPF) and (MPF_ovf_h or MPF_ovf_l);
    ovf_vec(MCX) := op(MCX) and (MCX_ovf_re or MCX_ovf_im);
    ovf_vec(MHI) := op(MHI) and MHI_ovf_flag;
    ovf_vec(MFI) := '0';
    SA := ll_A(half_width-1)xor ll_B(half_width-1);
    SB := acc_res(width+1);
    MAC_ovf_flag := (SA xnor SB) and (SA xor vec4_res(34));
    -- to be fixed
    ACC_ovf_flag := (vec4_res(31)xor vec4_res(32))or
                    (vec4_res(31)xor vec4_res(33))or
                    (vec4_res(31)xor vec4_res(34));  -- is it correct???
    ovf_vec(MAC) := op(MAC) and MAC_ovf_flag;
    ovf_vec(MCC) := '0';
    ovf_vec(ACC) := op(ACC) and ACC_ovf_flag;
    overflow_flags <= ovf_vec;
  end process overflow_control;
  -- connect overflow flag to output;
  ovf <= det_one(overflow_flags);

  -- accumulator
  accum_en <= op(MAC)or op(ACC) or op(MCC);
  accumulator: process (clk, rst)
  begin  -- process accumulator
    if rst = '0' then                   -- asynchronous reset (active low)
      acc_res <= (others => '0');
    elsif clk'event and clk = '1' then  -- rising clock edge
      if accum_en = '1' then
        acc_res <= vec4_res;
      end if;
    end if;
  end process accumulator;

  -- create results
  MHI_res <= conv_std_logic_vector(0,16)&vec4_res(31)&vec4_res(half_width-2 downto 0);
  MSF_res <= conv_std_logic_vector(0,16)&
                          vec4_res(31)&vec4_res(width-3 downto half_width-1);
  MPF_res <= MPF_high(31)&MPF_high(width-3 downto half_width-1)
             &vec4_res(31)&vec4_res(width-3 downto half_width-1);
  MCX_res <= vec4_res(width)&vec4_res(width-3 downto half_width-1)
               &vec6_res(width)&vec6_res(width-3 downto half_width-1);
  accum <= acc_res(width+1)&acc_res(width-2 downto 0);

  -- steering output multiplier
  output_mux: process(op, MSF_res, MPF_res, MCX_res, MHI_res)
    variable MSF_mux, MPF_mux, MCX_mux : std_logic_vector(width-1 downto 0);
    variable MHI_mux : std_logic_vector(width-1 downto 0);
  begin  -- process output_mux
    for i in width-1 downto 0 loop
      MSF_mux(i) := MSF_res(i)and op(MSF);
      MPF_mux(i) := MPF_res(i)and op(MPF);
      MCX_mux(i) := MCX_res(i)and op(MCX);
      MHI_mux(i) := MHI_res(i)and op(MHI);
    end loop;  -- i
    Z <= ((MSF_mux or MPF_mux)or (MCX_mux or MHI_mux));
  end process output_mux;

  -- adder for the high part of MPF
  MPF_high_part_adder: process (hh_sum, hh_carry)
    variable sum : signed(width+1 downto 0);
  begin  -- process MFI_high_cond_sum
    sum := signed(hh_sum) + signed(hh_carry);  -- pragma label MPF_high_add
    MPF_high <= std_logic_vector(sum);
  end process MPF_high_part_adder;
  vec4_CI <= op(MCX);

  -- propagate adder for vec4_tree
  vec4_cpa: process (vec4_out0, vec4_out1, vec4_CI)
  constant r0 : resource := 0;
  attribute map_to_module of r0  : constant is "DW01_add";
  attribute implementation of r0 : constant is "bk";
  attribute ops of r0 : constant is "cpa4";
    variable vec4_CI_v  : signed(width+3 downto 0);
    variable vec4_res_i : signed(width+3 downto 0);
    variable op1, op2 : std_logic_vector(width+3 downto 0);
  begin  -- process vec4_cpa
    vec4_CI_v := (others => '0');
    vec4_CI_v(0) := vec4_CI;
    op1 := (not vec4_out0(width+2))&vec4_out0;
    op2 := '1'&vec4_out1;
    vec4_res_i := vec4_CI_v + signed(op1) + signed(op2);  -- pragma label cpa4
    vec4_res   <= std_logic_vector(vec4_res_i);
  end process vec4_cpa;

  vec6_cpa: process(vec6_out0, vec6_out1)
  constant r1 : resource := 0;
  attribute map_to_module of r1  : constant is "DW01_add";
  attribute implementation of r1 : constant is "bk";
  attribute ops of r1 : constant is "cpa6";
    variable vec6_res_v : unsigned(33 downto 0);
  begin  -- process vec6_cpa
```

```
    vec6_res_v := unsigned(vec6_out0)+unsigned(vec6_out1);  -- pragma label cpa6
    vec6_res <= std_logic_vector(vec6_res_v);
  end process vec6_cpa;

  -- instantiation of trees for the vector adders
  vec4_tree: DW02_tree
    generic map (
      num_inputs  => vec4,
      input_width => width+3) -- extended intermediate range of accumulator
    port map (
      INPUT => vec4_in,
      OUT0  => vec4_out0,
      OUT1  => vec4_out1);

  vec6_tree: DW02_tree
    generic map (
      num_inputs  => vec4,
      input_width => width+2)
    port map (
      INPUT => vec6_in,
      OUT0  => vec6_out0,
      OUT1  => vec6_out1);

  -- input connections for the vec4_tree
  hh_sum_ext   <= sgn_ext(hh_sum, 1);
  hh_carry_ext <= sgn_ext(hh_carry, 1);
  invert_sum_carry: for i in ll_sum'range generate
    ll_sum_inv(i)   <= ll_sum(i)xor op(MCX);
    ll_carry_inv(i) <= ll_carry(i)xor op(MCX);
  end generate invert_sum_carry;
  ll_sum_ext   <= sgn_ext(ll_sum_inv, 1);
  ll_carry_ext <= sgn_ext(ll_carry_inv, 1);
  ext_accum    <= acc_res(width+1)&acc_res(width+1 downto 0);
  vec4_in <= vec4_in0&vec4_in1&ll_sum_ext&ll_carry_ext;

  vec4_tree_inputs: process (hh_sum_ext, hh_carry_ext, ext_accum, op)
    variable ctrl_in0, ctrl_in1, reset_in1 : std_logic;
    variable vec4_in1_v : std_logic_vector(width+2 downto 0);
  begin  -- process vec4_tree_inputs
    ctrl_in0 := op(MCX);
    ctrl_in1 := op(MCX);
    reset_in1 := not(op(MSF)or op(MPF)or op(MHI)or op(MFI)or op(MCC));
    for i in hh_sum_ext'range loop
      vec4_in0(i) <= hh_sum_ext(i)and (ctrl_in0);
    end loop;  -- i
    case ctrl_in0 is
      when '0' =>
        vec4_in1_v := ext_accum;
      when others =>
        vec4_in1_v := hh_carry_ext(width+2 downto 1)&op(MCX);
    end case;
    for j in vec4_in1_v'range loop
      vec4_in1(j) <= vec4_in1_v(j)and reset_in1;
    end loop;  -- j
  end process vec4_tree_inputs;

  -- input connections for the vec6_tree
  vec6_in <= hl_sum&hl_carry&lh_sum&lh_carry;

  -- partial product generators instantiation
  hh_A <= upper(HH);
  hh_B <= lower(HH);
  hh_pp: DW02_multp
    generic map (
      a_width   => a_width,
      b_width   => b_width,
      out_width => multp_width)
    port map (
      a    => hh_A,
      b    => hh_B,
      tc   => TC_hh,
      out0 => hh_sum,
      out1 => hh_carry);

   hl_A <= upper(HL);
   hl_B <= lower(HL);
  hl_pp: DW02_multp
    generic map (
      a_width   => a_width,
      b_width   => b_width,
      out_width => multp_width)
    port map (
      a    => hl_A,
      b    => hl_B,
      tc   => TC_hl,
      out0 => hl_sum,
      out1 => hl_carry);

   lh_A <= upper(LH);
   lh_B <= lower(LH);
  lh_pp: DW02_multp
    generic map (
      a_width   => a_width,
      b_width   => b_width,
      out_width => multp_width)
    port map (
      a    => lh_A,
      b    => lh_B,
      tc   => TC_lh,
      out0 => lh_sum,
      out1 => lh_carry);

  ll_B <= lower(LL);
  ll_A <= upper(LL);
  ll_pp: DW02_multp
    generic map (
      a_width   => a_width,
      b_width   => b_width,
      out_width => multp_width)
    port map (
      a    => ll_A,
      b    => ll_B,
      tc   => TC_ll,
      out0 => ll_sum,
      out1 => ll_carry);

end split;

-- configure simulation models for the dware components
-- pragma translate_off
library DW02;
configuration sim_models of design is
  for split
    for hh_pp, hl_pp, lh_pp, ll_pp : DW02_multp
      use configuration DW02.DW02_multp_cfg_sim;
    end for;
  end for;
end sim_models;
-- pragma translate_on
```

A.3.6 The multiplier for the SPLIT-MD-MAC design

```
-------------------------------------------------------------------------------
-- Title      : multiplier
-- Project    : High power arithmetic unit to be power managed
-------------------------------------------------------------------------------
-- File       : mult.vhd
-- Author     : Georgios Plakaris
-- Company    : Computer Systems Engineering, DTU
-- Date       : 12/11/2002
-------------------------------------------------------------------------------
-- Description :
-- A component insantiating a multiplier. Different architectures in the future
-- will accommodate different kind of mulipliers.
-------------------------------------------------------------------------------
library ieee, SYNOPSYS;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use SYNOPSYS.attributes.all;
library WORK;
use WORK.design_utils.all;

entity mult is

  port (
    rst : in  std_logic;
    clk : in  std_logic;
    en  : in  std_logic;
    op1 : in  std_logic_vector(width-1 downto 0);
    op2 : in  std_logic_vector(width-1 downto 0);
    res : out std_logic_vector(width-1 downto 0);
    ovf : out std_logic);

end mult;
```

```
architecture structural of mult is
signal reg_A, reg_B : std_logic_vector(width-1 downto 0);
signal result64 : std_logic_vector(2*width-1 downto 0);
begin  -- behavioral

  mult32_iregs: process (clk, rst)
  begin
    if rst = '0' then                   -- asynchronous reset (active low)
      reg_A <= (others => '0');
      reg_B <= (others => '0');
    elsif clk'event and clk = '1' then  -- rising clock edge
      if en = '1' then
        reg_A <= op1;
        reg_B <= op2;
      end if;
    end if;
  end process mult32_iregs;

  multiplication: process (reg_A, reg_B)
      variable a, b : signed(width-1 downto 0);
      variable c : signed(2*width-1 downto 0);
    begin  -- process addition
      a := signed(reg_A);
      b := signed(reg_B);
      c := a*b;
      result64 <= std_logic_vector(c);
    end process multiplication;

  res <= result64(63)&result64(width-2 downto 0);
  ovf <= ((not result64(63)) and det_one(result64(62 downto width-1)))or
         ((result64(63)) and det_zero(result64(62 downto width-1)));

end structural;
```

A.3.7 The MD-MAC NCS design

```
-------------------------------------------------------------------------------
-- Title      : design_ncs_shared.vhdl
-- Project    : A multi-datatype MAU unit
-------------------------------------------------------------------------------
-- File       : design_ncs_shared.vhdl
-- Author     : Georgios Plakaris
-- Company    : Computer Systems Engineering, DTU
-- Date       : 12/02/2003
-------------------------------------------------------------------------------
-- Description :
-- The processing unit of the core design. A clearly combinatorial circuit,
-- apart from some latches to gate control signals and the accumulator
-------------------------------------------------------------------------------

library ieee, SYNOPSYS, DW01, DW02, DWARE, WORK;
use WORK.design_utils.all;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use SYNOPSYS.attributes.all;
use DWARE.DWpackages.all;
use DW01.DW01_components.all;
use DW02.DW02_components.all;

entity design_ncs_shared is

  port (
    rst    : in  std_logic;
    clk    : in  std_logic;
    op     : in  std_logic_vector(inst_count-1 downto 0);
    HH, HL : in std_logic_vector(width-1 downto 0);
    LH, LL : in std_logic_vector(width-1 downto 0);
    Z      : out std_logic_vector(width-1 downto 0);
    accum  : out std_logic_vector(width-1 downto 0);  -- 34bit accumulator
    ovf    : out std_logic);

end design_ncs_shared;

architecture structural of design_ncs_shared is
-- parameters for the multp trees
constant a_width, b_width : integer := half_width;
signal hh_prod, ll_prod : std_logic_vector(width-1 downto 0);
signal hl_prod, lh_prod : std_logic_vector(width+1 downto 0);
signal ll_prod_inv      : std_logic_vector(width-1 downto 0);
signal TC_hh, TC_hl, TC_lh, TC_ll : std_logic;
signal hh_A, hh_B, ll_A, ll_B : std_logic_vector(half_width-1 downto 0);
signal hl_A, hl_B, lh_A, lh_B : std_logic_vector(half_width downto 0);
attribute implementation : string;
attribute implementation of hh_mp, hl_mp, lh_mp, ll_mp : label is "nbw";
constant vec4 : integer := 4;
signal add4_in0, add4_in1   : std_logic_vector(width+2 downto 0);
signal add6_in : std_logic_vector(vec4*(50)-1 downto 0);
signal add6_out0, add6_out1 : std_logic_vector(49 downto 0);
signal add6_in0, add6_in1       : std_logic_vector(49 downto 0);
signal tmp_add6_in0 : std_logic_vector(half_width-1 downto 0);
signal tmp_add6_in3 : std_logic_vector(47 downto 0);
signal add6_in2, add6_in3       : std_logic_vector(49 downto 0);
signal ext_accum                : std_logic_vector(width+2 downto 0);
-- parameters for the cp adders for the vec trees
signal add4_res : std_logic_vector(width+2 downto 0);
signal add6_res : std_logic_vector(50 downto 0);
signal add4_CI : std_logic;
-- parameters for the steering logic
signal MSF_res, MPF_res, MCX_res : std_logic_vector(width-1 downto 0);
signal MHI_res, MFI_res : std_logic_vector(width-1 downto 0);
-- parameters for the accumulator
attribute sync_set_reset_local of accumulator : label is  "accum_en" ;
signal accum_en : std_logic;
signal acc_res : std_logic_vector(width+1 downto 0);
-- parameters for the overflow logic
signal MHI_ovf_flag, MFI_ovf_flag : std_logic;
signal overflow_flags : std_logic_vector(inst_count-1 downto 0);
begin  -- structural
  -- signed number selector
  TC_hh <= '1';
  TC_hl <= '1';
  TC_lh <= '1';
  TC_ll <= not op(MFI);

  -- overflow control
  MHI_ovf_flag <= ((not ll_prod(31)) and det_one(ll_prod(30 downto 15)))or
                  ((ll_prod(31)) and det_zero(ll_prod(30 downto 15)));

  MFI_ovf_flag <= ((not add6_res(47)) and det_one(add6_res(46 downto half_width-2)))or
                  ((add6_res(47)) and det_zero(add6_res(46 downto half_width-2)));

  overflow_control: process(op, add4_res, add6_res, hh_prod, ll_prod,  MHI_ovf_flag, MFI_ovf_flag, ll_A, ll_B, acc_res)
    variable ovf_vec : std_logic_vector(inst_count-1 downto 0);
    variable MPF_ovf_h, MPF_ovf_l, MCX_ovf_re, MCX_ovf_im : std_logic;
    variable MAC_ovf_flag, ACC_ovf_flag, SA, SB : std_logic;
  begin  -- process overflow_control
    ovf_vec := (others => '0');
    MPF_ovf_l := (ll_prod(31) xor ll_prod(30));
    MPF_ovf_h := (hh_prod(31) xor hh_prod(30));
    MCX_ovf_re := (add4_res(32)xor add4_res(31))or
                  (add4_res(32)xor add4_res(30))or
                  (add4_res(32)xor add4_res(33))or
                  (add4_res(32)xor add4_res(34));
    MCX_ovf_im := (add6_res(32)xor add6_res(31))or
                  (add6_res(32)xor add6_res(30))or
                  (add6_res(32)xor add6_res(33))or
                  (add6_res(32)xor add6_res(34));
    ovf_vec(MSF) := op(MSF) and MPF_ovf_l;
    ovf_vec(MPF) := op(MPF) and (MPF_ovf_h or MPF_ovf_l);
    ovf_vec(MCX) := op(MCX) and (MCX_ovf_re or MCX_ovf_im);
    ovf_vec(MHI) := op(MHI) and MHI_ovf_flag;
    ovf_vec(MFI) := op(MFI) and MFI_ovf_flag;
    SA := ll_A(half_width-1)xor ll_B(half_width-1);
    SB := acc_res(width+1);
    MAC_ovf_flag := (SA xnor SB) and (SA xor add4_res(34));
    -- to be fixed
    ACC_ovf_flag := (add4_res(31)xor add4_res(32))or
                    (add4_res(31)xor add4_res(33))or
                    (add4_res(31)xor add4_res(34));
    ovf_vec(MAC) := op(MAC) and MAC_ovf_flag;
    ovf_vec(MCC) := '0';
    ovf_vec(ACC) := op(ACC) and ACC_ovf_flag;
    overflow_flags <= ovf_vec;
  end process overflow_control;
  -- connect overflow flag to output;
  ovf <= det_one(overflow_flags);

  -- accumulator
  accum_en <= op(MAC)or op(ACC) or op(MCC);
  accumulator: process (clk, rst)
  begin  -- process accumulator
    if rst = '0' then                   -- asynchronous reset (active low)
      acc_res <= (others => '0');
    elsif clk'event and clk = '1' then  -- rising clock edge
      if accum_en = '1' then
        acc_res <= add4_res(33 downto 0);
      end if;
    end if;
  end process accumulator;

  -- create results
  MHI_res <= conv_std_logic_vector(0,16)&ll_prod(31)&ll_prod(half_width-2 downto 0);
  MFI_res <= add6_res(47)&add6_res(half_width-2 downto 0)&ll_prod(half_width-1 downto 0);
  MSF_res <= conv_std_logic_vector(0,16)&
                          ll_prod(31)&ll_prod(width-3 downto half_width-1);
  MPF_res <= hh_prod(31)&hh_prod(width-3 downto half_width-1)
             &ll_prod(31)&ll_prod(width-3 downto half_width-1);
  MCX_res <= add4_res(width)&add4_res(width-3 downto half_width-1)
               &add6_res(width)&add6_res(width-3 downto half_width-1);
  accum <= acc_res(width+1)&acc_res(width-2 downto 0);
  -- steering output multiplier
  output_mux: process(op, MSF_res, MPF_res, MCX_res, MHI_res, MFI_res)
    variable MSF_mux, MPF_mux, MCX_mux : std_logic_vector(width-1 downto 0);
    variable MHI_mux, MFI_mux : std_logic_vector(width-1 downto 0);
  begin  -- process output_mux
    for i in width-1 downto 0 loop
      MSF_mux(i) := MSF_res(i)and op(MSF);
      MPF_mux(i) := MPF_res(i)and op(MPF);
      MCX_mux(i) := MCX_res(i)and op(MCX);
      MHI_mux(i) := MHI_res(i)and op(MHI);
      MFI_mux(i) := MFI_res(i)and op(MFI);
    end loop;  -- i
    Z <= MCX_mux or ((MSF_mux or MPF_mux)or (MFI_mux or MHI_mux));
  end process output_mux;

  add4_CI <= op(MCX);
  -- propagate adder for vec4_tree
  add4_cpa: process (add4_in1, ll_prod_inv, add4_CI)
  constant r0 : resource := 0;
  attribute map_to_module of r0  : constant is "DW01_add";
  attribute implementation of r0 : constant is "bk";
  attribute ops of r0 : constant is "cpa4";
    variable add4_CI_v  : signed(width+2 downto 0);
    variable add4_res_i : signed(width+2 downto 0);
  begin  -- process vec4_cpa
    add4_CI_v := (others => '0');
    add4_CI_v(0) := add4_CI;
    add4_res_i := add4_CI_v + signed(add4_in1) + signed(ll_prod_inv);  -- pragma label cpa4
    add4_res   <= std_logic_vector(add4_res_i);
  end process add4_cpa;

  vec6_cpa: process(add6_out0, add6_out1)
  constant r1 : resource := 0;
  attribute map_to_module of r1  : constant is "DW01_add";
  attribute implementation of r1 : constant is "bk";
  attribute ops of r1 : constant is "cpa6";
    variable add6_res_v : unsigned(50 downto 0);
    variable op1, op2   : std_logic_vector(50 downto 0);
  begin  -- process vec6_cpa
    op1 := (not add6_out0(49))&add6_out0;
    op2 := '1'&add6_out1;
    add6_res_v := unsigned(op1)+unsigned(op2);  -- pragma label cpa6
    add6_res <= std_logic_vector(add6_res_v);
  end process vec6_cpa;

  vec6_tree: DW02_tree
    generic map (
      num_inputs  => vec4,
      input_width => 50)
    port map (
      INPUT => add6_in,
      OUT0  => add6_out0,
      OUT1  => add6_out1);

  -- input connections for the add4_cpa
  invert_ll_prod: for i in ll_prod'range generate
    ll_prod_inv(i)   <= ll_prod(i)xor op(MCX);
  end generate invert_ll_prod;
  ext_accum    <= acc_res(width+1)&acc_res(width+1 downto 0);

  add4_inputs: process (hh_prod, ext_accum, op)
    variable ctrl_in1, reset_in1 : std_logic;
    variable add4_in1_v : std_logic_vector(width+2 downto 0);
  begin  -- process vec4_tree_inputs
    ctrl_in1 := op(MCX);
    reset_in1 := not(op(MSF)or op(MPF)or op(MHI)or op(MFI)or op(MCC));
```

```
    case ctrl_in1 is
      when '0' =>
        add4_in1_v := ext_accum;
      when others =>
        add4_in1_v := sgn_ext(hh_prod, 3);
    end case;
    for j in add4_in1_v'range loop
      add4_in1(j) <= add4_in1_v(j)and reset_in1;
    end loop;  -- j
  end process add4_inputs;

  -- input connections for the add6_tree
  tmp_add6_in0 <=  ll_prod(width-1 downto half_width);
  tmp_add6_in3 <=  hh_prod(width-1 downto 0)&conv_std_logic_vector(0,16);
  vec6_tree_inputs: process (tmp_add6_in0, tmp_add6_in3)
    variable ctrl_vec6             : std_logic;
    variable add6_in0_i : std_logic_vector(49 downto 0);
    variable add6_in3_i : std_logic_vector(49 downto 0);
  begin  -- process vec6_tree_inputs
    ctrl_vec6  := op(MFI);
    add6_in0_i := sgn_ext(tmp_add6_in0,34);
    add6_in3_i := sgn_ext(tmp_add6_in3,2);
    for j in add6_in0_i'range loop
      add6_in0(j) <= add6_in0_i(j)and ctrl_vec6;
      add6_in3(j) <= add6_in3_i(j)and ctrl_vec6;
    end loop;  -- j
  end process vec6_tree_inputs;
  add6_in1 <= sgn_ext(hl_prod,16);
  add6_in2 <= sgn_ext(lh_prod,16);
  add6_in  <= add6_in0&add6_in1&add6_in2&add6_in3;

  --product generators instantiation
  hh_A <= upper(HH);
  hh_B <= lower(HH);
  hh_mp : DW02_mult
    generic map (
      a_width => a_width,
      b_width => b_width)
    port map (
      a       => hh_A,
      b       => hh_B,
      tc      => TC_hh,
      product => hh_prod);

  fix_inputs_hl: process (HL, op)
    variable signB : std_logic;
  begin  -- process fix_inputs_hl
   signB := HL(half_width-1);
   if op(MFI) = '1' then
     signB := '0';
   end if;
   hl_A <= HL(width-1)&upper(HL);
   hl_B <= signB&lower(HL);
  end process fix_inputs_hl;

  hl_mp : DW02_mult
    generic map (
      a_width => a_width+1,
      b_width => b_width+1)
    port map (
      a       => hl_A,
      b       => hl_B,
      tc      => TC_hl,
      product => hl_prod);

  fix_inputs_lh: process (LH, op)
    variable signA : std_logic;
  begin  -- process fix_inputs_hl
   signA := LH(width-1);
   if op(MFI) = '1' then
     signA := '0';
   end if;
    lh_A <= signA&upper(LH);
    lh_B <= LH(half_width-1)&lower(LH);
  end process fix_inputs_lh;

  lh_mp : DW02_mult
    generic map (
      a_width => a_width+1,
      b_width => b_width+1)
    port map (
      a       => lh_A,
      b       => lh_B,
      tc      => TC_lh,
      product => lh_prod);

  ll_B <= lower(LL);
  ll_A <= upper(LL);
  ll_mp: DW02_mult
    generic map (
      a_width   => a_width,
      b_width   => b_width)
    port map (
      a       => ll_A,
      b       => ll_B,
      tc      => TC_ll,
      product => ll_prod);

end structural;

-- pragma translate_off
library DW02;
configuration sim_models of design_ncs_shared is
  for structural
    for hh_mp, hl_mp, lh_mp, ll_mp : DW02_mult
      use configuration DW02.DW02_mult_cfg_sim;
    end for;
  end for;
end sim_models;
-- pragma translate_on
```