THESIS FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

Efficient and Flexible Embedded Systems and Datapath Components

MAGNUS SJÄLANDER

Division of Computer Engineering
Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
Göteborg, Sweden 2008

Efficient and Flexible Embedded Systems and Datapath Components
Magnus Själander
ISBN 978-91-7385-137-4

Doktorsavhandlingar vid Chalmers tekniska högskola
Ny serie 2818
ISSN 0346-718X

Technical report 40D
Department of Computer Science and Engineering
VLSI Research Group

Department of Computer Science and Engineering
Chalmers University of Technology
SE-412 96 Göteborg, Sweden
Phone: +46 (0)31-772 10 00

Author e-mail: magnus@sjalander.com

Cover:
Picture by the author. Illustration of a multi-core architecture with several Flex-Cores, where each core has a twin-precision multiplier.

Printed by Chalmers Reproservice
Göteborg, Sweden 2008

Efficient and Flexible Embedded Systems and Datapath Components

Magnus Själander

Division of Computer Engineering, Chalmers University of Technology

ABSTRACT

The comfort of our daily lives has come to rely on a vast number of embedded systems, such as mobile phones, anti-spin systems for cars, and high-definition video. To improve the end-user experience at often stringent requirements, in terms of high performance, low power dissipation, and low cost, makes these systems complex and nontrivial to design.

This thesis addresses design challenges in three different areas of embedded systems. The presented FlexCore processor intends to improve the programmability of heterogeneous embedded systems while maintaining the performance of application-specific accelerators. This is achieved by integrating accelerators into the datapath of a general-purpose processor in combination with a wide control word consisting of all control signals in a FlexCore's datapath. Furthermore, a FlexCore processor utilizes a flexible interconnect, which together with the expressiveness of the wide control word improves its performance.

When designing new embedded systems it is important to have efficient components to build from. Arithmetic circuits are especially important, since they are extensively used in all applications. In particular, integer multipliers present big design challenges. The proposed twin-precision technique makes it possible to improve both throughput and power of conventional integer multipliers, when computing narrow-width multiplications. The thesis also shows that the Baugh-Wooley algorithm is more suitable for hardware implementations of signed integer multipliers than the commonly used modified-Booth algorithm.

A multi-core architecture is a common design choice when a single-core architecture cannot deliver sufficient performance. However, multi-core architectures introduce their own design challenges, such as scheduling applications onto several cores. This thesis presents a novel task management unit, which offloads task scheduling from the conventional cores of a multi-core system, thus improving both performance and power efficiency of the system.

This thesis proposes novel solutions to a number of relevant issues that need to be addressed when designing embedded systems.

Keywords: Embedded, Flexible, Multi-Core, Multiplier, Scheduling, Twin-Precision

List of Appended Papers

A Martin Thuresson, **Magnus Själander**, Magnus Björk, Lars Svensson, Per Larsson-Edefors, and Per Stenström, "FlexCore: Utilizing Exposed Datapath Control for Efficient Computing," Accepted March 4, 2008, for publication in *Springer Journal of Signal Processing Systems*.

B **Magnus Själander**, Per Larsson-Edefors and Magnus Björk, "A Flexible Datapath Interconnect for Embedded Applications," in *Proceedings of the 2007 IEEE Computer Society Annual Symposium on VLSI*, Porto Alegre, Brazil, May 9-11, 2007, pp. 15-20.

C **Magnus Själander** and Per Larsson-Edefors, "The Case for HPM-Based Baugh-Wooley Multipliers," in *Department of Computer Science and Engineering, Chalmers University of Technology, Technical Report 2008-8*, Sweden, March 4, 2008.

D **Magnus Själander**, Henrik Eriksson, and Per Larsson-Edefors, "An Efficient Twin-Precision Multiplier" in *Proceedings of the 2004 IEEE International Conference on Computer Design*, San Jose, California, USA, October 10-13, 2004, pp. 30-33.

E **Magnus Själander**, Mindaugas Draždžiulis, Per Larsson-Edefors, and Henrik Eriksson, "A Low-Leakage Twin-Precision Multiplier Using Reconfigurable Power Gating" in *Proceedings of the 2005 IEEE International Symposium on Circuits and Systems*, Kobe, Japan, May 23-26, 2005, pp. 1654-1657.

F **Magnus Själander** and Per Larsson-Edefors, "Multiplication Acceleration Through Twin Precision" Conditionally Accepted: Minor Revision, March 17, 2008, in *IEEE Transactions on VLSI Systems*.

G **Magnus Själander**, Andrei Terechko, and Marc Duranton, "A Look-Ahead Task Management Unit for Embedded Multi-Core Architectures," Accepted for publication in *Proceedings of the 2008 Euromicro Conference on Digital System Design: Architectures, Methods and Tools*, Parma, Italy, September 3-5, 2008.

Related publications that are not included in this thesis:

▷ **Magnus Själander** and Per Larsson-Edefors, "High-Speed and Low-Power Multipliers Using the Baugh-Wooley Algorithm and HPM Reduction Tree" Accepted for publication in *Proceedings of the 2008 IEEE International Conference on Electronics, Circuits, and Systems*, St. Julian's, Malta, August 31 - September 3, 2008.

▷ Mafijul Md. Islam, **Magnus Själander**, and Per Stenström, "Early Detection and Bypassing of Trivial Operations to Improve Energy Efficiency of Processors" in *Elsevier Journal on Microprocessors and Microsystems*, 2008, doi:10.1016/j.micpro.2007.10.001.

▷ Martin Thuresson, **Magnus Själander**, Magnus Björk, Lars Svensson, Per Larsson-Edefors, and Per Stenström, "FlexCore: Utilizing Exposed Datapath Control for Efficient Computing" in *International Symposium on Systems, Architectures, Modeling, and Simulation*, Samos, Greece, July 16-19 2007, pp.18-25.

▷ Magnus Björk, **Magnus Själander**, Lars Svensson, Martin Thuresson, John Hughes, Kjell Jeppson, Jonas Karlsson, Per Larsson-Edefors, Mary Sheeran, and Per Stenström, "Exposed Datapath for Efficient Computing" in *HiPEAC Workshop on Reconfigurable Computing*, Ghent, Belgium, January 28-30 2006.

- ▷ Henrik Eriksson, Per Larsson-Edefors, Mary Sheeran, **Magnus Själander**, Daniel Johansson, and Martin Schölin, "Multiplier Reduction Tree with Logarithmic Logic Depth and Regular Connectivity" in *Proceedings of the 2006 IEEE International Symposium on Circuits and Systems*, Kos, Greece, May 21-24, 2006, pp. 5-8.

Related Patent:

- ▷ On March 12, 2008, patent application EP08102525 was filed at the European Patent Office for mechanisms in **Paper G**.

Acknowledgments

I am grateful to the following people for what they have done for me, for my career, and for this thesis.

- ▷ Professor Per Larsson-Edefors† for believing in me and giving me the opportunity to pursue a research career.
- ▷ Martin Thuresson† for not only being a valued colleague and coauthor but also a good friend that have made the time at Chalmers both interesting and fun.
- ▷ Henrik Eriksson for introducing me to parallel multiplier implementations and Mindaugas Draždžiulis for our joint work on leakage reduction in twin-precision multipliers.
- ▷ Mafijul Islam for our fruitful collaboration on how to bypass trivial operations.
- ▷ Daniel Andersson, Daniel Eckerbert, Minh Quang Do, and Tung Hoang as part of the VLSI research group and creating an interesting and inspiring working environment.
- ▷ Jim Nilsson for our early brainstorm on reconfigurable multipliers.
- ▷ Lars 'J' Svensson† for our combined struggles in the FlexSoC project.
- ▷ Magnus Björk, Thomas Schilling, Jonas Ferry, Erik Ryman, Jan Mårts, and Tomas Carlqvist for their efforts in the FlexSoC project.
- ▷ John Hughes, Kjell Jeppsson†, and Mary Sheeran for their participation in the FlexSoC project.
- ▷ My colleagues at the department of Computer Science and Engineering, especially Magnus Almgren, Ewa Cederheim-Wäingelin, Peter He-

lander, Pierre Kleberger, Lars Kollberg, and Ulf Larsson for creating a good working environment.

- ▷ Marc Duranton for accepting me as an intern at NXP Semiconductors and Andrei Terechko for his guidance and many interesting discussions regarding multi-core architectures.
- ▷ Per Stenström and the HiPEAC European Network of Excellence for helping me fund my stay at NXP Semiconductors.
- ▷ The Swedish Foundation for Strategic Research (SSF) for their funding of my position at Chalmers University of Technology.
- ▷ Sten Gunnarsson† and Wolfgang John† for many great times while hiking in the woods, climbing in and around Göteborg, ice-skating on some lake that barely has ice, or kayaking in the archipelago.
- ▷ Tom Ashby, Loïc Baggetto, Ana Bosque Arbiol, Marius Grannæs, Robert Juenemann and Alberto, who I have had the great pleasure to get to know during my time as a Ph.D student and spent many great times with, in Göteborg and other exotic places.
- ▷ Jessica Leong for encouraging me to become a Ph.D student.
- ▷ Conny Olsson, Henrik Larsson, Marcus Östberg, Greger Sundqvist, and Lina Ohlsson for their long lasting friendship.
- ▷ My beloved grandmother Sonja Kallin for always supporting me and making it possible to start my academic career by letting me stay at her house when I had evening classes at the Mid Sweden University.
- ▷ My Uncle Harald Kallin for inspiring me to become an engineer.
- ▷ My parents Thomas and Ingeborg and my brother Lars for always supporting and encouraging me to pursue new challenges in life.
- ▷ Madelene Egardt for all her love and support. I love you.

A special thanks goes to all the people marked with the †-symbol who have read and given valuable comments on this thesis.

Magnus Själander
Göteborg, June 2008

Contents

1 Introduction

Our modern lifestyle depends on a wide range of functionalities that are provided by numerous embedded systems, such as the fuel injection system of combustion engines, home entertainment systems, and mobile devices. Moreover, there is a general trend towards increasing numbers of embedded systems in our daily lives and in order to meet customer demands, these systems tend to become increasingly more complex with each new generation.

The strive towards higher performance and more functionality in combination with stringent requirements makes general-purpose processors unsuitable for embedded systems. A high-end general-purpose processor tends to have excessively high power dissipation requirements, which would quickly drain the battery of a mobile device and would require costly parts for cooling. A common design practice used to achieve required performance at low power and small area is to design heterogeneous system-on-chips, where specialized hardware accelerators are controlled by one or more embedded microproces-

sors, such as an ARM core [1]. However, designing heterogeneous system-on-chips, which are flexible and can cater for new applications that are not within the original set of targeted applications, is difficult and require great engineering efforts. Application Specific Instruction-set Processors (ASIPs), such as Tensillica's Xtensa family [2], try to combine the performance benefits from heterogeneous system-on-chips with the flexibility of general-purpose processors. An ASIP is generally based on a small general-purpose processor with a limited instruction set, to which application-specific instructions are added. Instructions can be added by combining several conventional instructions into one new instruction, so called instruction-fusion, or by adding specialized hardware accelerators to the datapath of the processor. By introducing accelerators into the instruction set of the processor a homogeneous interface to the hardware is created. This allows for the use of a conventional software tool-flow, where specific hardware accelerators can be accessed through assembler instructions or by software macros in high-level languages. However, a disadvantage with an adaptable instruction set is that applications might not be binary compatible between different architectural implementations and need to be recompiled.

This thesis presents the FlexCore processor, which is based on the ideas of the FlexSoC [3] approach on how to build embedded systems. The FlexCore processor combines the flexibility of general-purpose processors with the performance of specialized hardware accelerator units. This is achieved by integrating hardware accelerators into the datapath of a general-purpose processor. In contrast to conventional processors and ASIPs, a FlexCore processor does not have a conventional instruction set, where an instruction controls a set of pipeline stages over several clock cycles. A FlexCore processor instead has a wide control word that controls the whole datapath for a single clock cycle. The wide control word consists of all control signals to all units and the interconnect in the FlexCore datapath. A datapath unit can in this case be anything from a register to some specialized hardware accelerator. Moving away from a conventional instruction set to an architecture with a wide control word, which is exposed to the compiler/programmer, gives an application detailed control of the hardware. The increased controllability has earlier been shown to improve a datapath's performance [4, 5].

The interconnect of a FlexCore processor is, in the early stage of the processor development, a fully connected crossbar switch that connects the different datapath units. The flexible interconnect together with an easily extendable control word make it extremely easy to add new datapath units to a FlexCore processor. The required input and output ports are added to the interconnect and the control word is appended with control signals for the new datapath unit and for addressing the new ports of the interconnect. However, a fully connected crossbar does not scale well. Therefore, once a set of datapath units has been chosen, the compiler together with the intended set of applications can be used to eliminate superfluous paths from the interconnect.

The wide control word would be an unsuitable format for storing applications, because of the large memory space it would require. The FlexSoC approach therefore envisions a dynamically reconfigurable instruction decoder that allows for a more compact representation of the control word. The reconfigurable instruction decoder makes it possible to tailor an instruction set for a specific application or set of applications. However, the reconfigurable instruction decoder is not within the scope of this thesis.

When designing digital systems, such as a FlexCore processor, it is important to have efficient datapath units to build the system from. One such datapath unit is the integer multiplier which is frequently used in typical embedded applications, such as signal processing and multimedia applications. Integer multiplier implementations tend to be both large in terms of area and have a high power dissipation, relative other arithmetic circuits. This thesis presents the twin-precision technique, which makes it possible for an integer multiplier to efficiently compute narrow-width multiplications (i.e. 16-bit multiplications on a 32-bit multiplier) and to increase the throughput by executing two narrow-width multiplications in parallel. Furthermore, an evaluation is presented where the modified-Booth algorithm [6] is compared to the Baugh-Wooley algorithm [7].

The increasing demand for performance leads to a point where single processors cannot, at least in a power efficient way, deliver required performance. At this point multi-core systems is the only promising solution. However, for a single application to take advantage of the available performance of a multi-core system it is necessary to schedule the application onto all cores of the system.

For this reason a task-management unit is proposed to accelerate and offload the scheduling overhead that is introduced when a multi-task application is distributed over a set of processor cores.

1.1 FlexCore

The FlexSoC [3] research project attempts to combine the performance and energy efficiency of hardware accelerators with the flexibility of programmable processors by introducing a uniform programming interface. This is achieved by moving away from conventional Instruction Set Architectures (ISAs). Instead, a two-tier ISA is defined, with a wide Native-ISA (N-ISA) capable of fine-grained control of all computational resources of a datapath and customizable Application Specific-ISAs (AS-ISAs) that can be made compact and thus suitable for storing applications in memory. A reconfigurable instruction decoder decodes AS-ISA instructions into N-ISA instructions. The instruction decoder is reconfigurable to allow for the creation of customized AS-ISAs for a particular application or set of applications. An illustration of the FlexSoC concept is shown in Fig. 1.1.

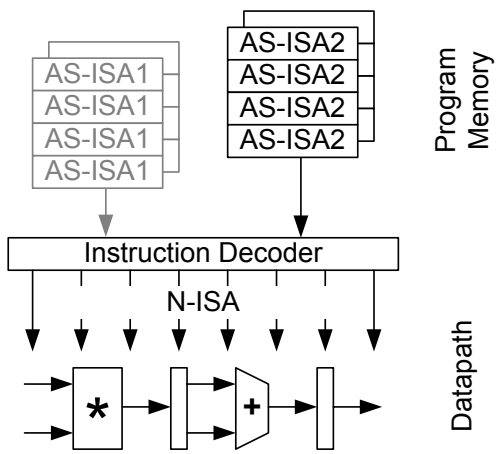


Figure 1.1: *Illustration of a FlexSoC system with AS-ISA to N-ISA decoding.*

A new processor architecture, called FlexCore, has been designed to evaluate the idea of a two-tier ISA. A FlexCore processor, in its simplest form, consists of all the datapath units of a conventional single-issue five-stage pipeline, similar to the Hennessy-Patterson 32-bit DLX and MIPS R2000 [8]. Application-specific hardware accelerators can then be added as new datapath units for improved performance and energy efficiency. All datapath units are connected to a flexible interconnect that allows data to be freely routed between the different datapath units. The datapath units together with the interconnect make it possible for a FlexCore processor to emulate a five-stage pipeline of a General-Purpose Processor (GPP) and can therefore offer the full programmability of a GPP. The N-ISA is defined as the collection of all control signals of each datapath unit and the address signals of the interconnect. Fig. 1.2 shows an illustration of the baseline FlexCore processor without any added hardware accelerators, where the N-ISA is seen as all the signals leaving the control.

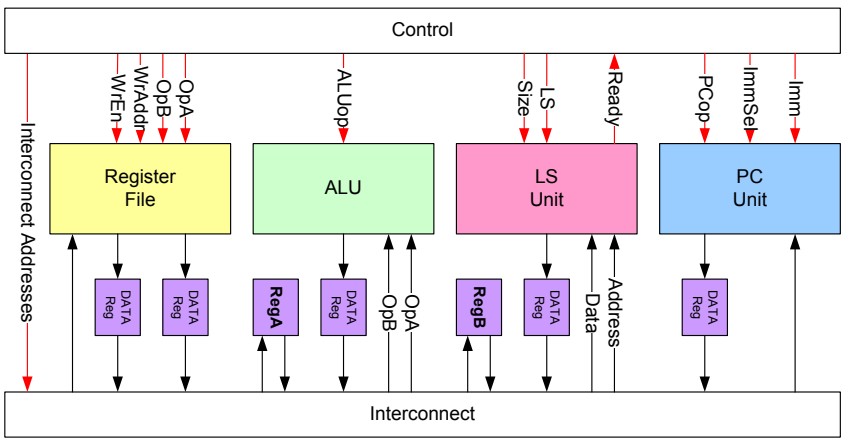


Figure 1.2: *Illustration of a baseline FlexCore. Note that RegA and RegB are used as pipeline registers when the FlexCore operates as a five-stage GPP pipeline.*

The N-ISA changes the abstraction level at which the programmer/compiler can control the datapath compared to a conventional ISA. The abstraction level of a conventional ISA is the register file, where the input-operand(s) of each instruction of a conventional ISA is assumed to be available. If an input-operand

is not yet available in the register file, the hardware either stalls the instruction or dynamically reroutes the result of a previous instruction, which has already been executed but not yet written its result to the register file. This abstraction simplifies scheduling of instructions, since it is not necessary to keep track of individual hardware resources in the datapath. However, it also limits the control of the datapath and creates unnecessary pressure on the register file, which can degrade performance. The extra pressure is created since all instructions write their result to the register file, even though in many cases the result of an instruction is temporal and consumed by the following executed instruction. In this case the result could be routed directly between the two consecutive instructions, without writing it to the register file.

The abstraction of the N-ISA is at the datapath level, which gives the compiler/programmer full control of the whole datapath in each clock cycle. This complicates scheduling, since the operation of each individual datapath unit and routing of data between the units have to be scheduled for each cycle by the compiler/programmer. However, the increased controllability makes it possible to more freely schedule instructions and route data between datapath units. This improves both performance and reduces register file access, which can have a positive effect on the power dissipation. An example of how the N-ISA makes it possible to improve instruction scheduling can be found in **Paper A**, together with a more detailed description of the FlexCore processor.

The FlexCore processor has a flexible interconnect, which is presented in **Paper B**. In the early stages of a FlexCore's development, the interconnect consists of a fully connected crossbar switch. The high connectivity between different datapath units allows for a more efficient scheduling, compared to more restricted interconnects, such as that of a GPP. The interconnect also makes it easy to add new datapath units to a FlexCore, since there never are any special cases, like forwarding paths, to cater for. A fully connected crossbar switch is costly to implement and does not scale well for reasonable large number of ports. However, a FlexCore processor is statically scheduled, which makes it possible to determine exactly the interconnect paths that will be used for a set of applications. The unused paths can then be removed without any impact on the performance in terms of executed cycles. By making sure that no paths associ-

ated with a conventional GPP interconnect is removed, the full programmability of a GPP can still be maintained.

The wide control word and the fully connected interconnect are in **Paper A** shown to reduce the executed number of cycles with 10-36% for a set of four different applications from the EEMBC Telecom and Consumer benchmark suites [9]. It is also shown that 48% of all the paths in the fully-connected interconnect are never used by the executed set of applications. Further, in comparison to a FlexCore processor with the interconnect of a GPP in a 65 nm process technology, a tailored interconnect is shown to have negligible impact on performance and power of a FlexCore processor, with a reasonable increase of 6% in area. The same tailored interconnect improves the total execution time and energy dissipation of the four applications with 9-19% and 10-21%, respectively. Interesting to note is that for one of the applications (Viterbi) the execution time is not improved by the wide control word, while the tailored interconnect contributes to an improved performance for all four applications.

The FlexCore architecture has been designed to be easily extendable with new datapath units. An example of its extendibility is presented as a case study in **Paper F**. In the case study, a twin-precision multiplier is added to a FlexCore processor and new operations are added to its Arithmetic Logic Unit (ALU). The extended FlexCore is then used to emulate the execution of a GPP, instead of utilizing its full scheduling potential. The software tool-flow for the FlexCore architecture was therefore updated with five new assembler instructions and a Fast Fourier Transform (FFT) application was manually modified to take advantage of the new instructions. The result is a FlexCore processor with light-weight Single Instruction Multiple Data (SIMD) support that can be programmed at the assembler level. For the FFT application, this translates to 15% faster execution, compared to executing the FFT on a FlexCore with a conventional multiplier and no SIMD support.

The FlexCore processor has made it possible to evaluate the performance benefits that can be achieved by the expressive N-ISA. Furthermore, N-ISA instruction traces from executions of applications provide vital information for experimenting with the implementation of reconfigurable instruction decoders and the creation of customized AS-ISAs.

1.2 Multiplier Implementations

Multiplication is a common and important arithmetic operation in typical embedded applications, such as multimedia and digital signal processing. However, multiplier implementations tend to be larger, slower, and more power dissipating, compared to other arithmetic circuits. It is not uncommon that a multiplier unit is larger than the whole Arithmetic Logic Unit (ALU) of a processor. An efficient multiplier implementation can therefore have significant impact on the size and power of an embedded processor.

1.2.1 Single-Precision Multiplier

There are several possible ways of implementing multiplications. From the simplest being an implementation in software, where the multiplication is iterated over an adder and shifter, to hardware-implemented parallel multipliers. Parallel multipliers can be divided into three distinct steps. In the first step, a set of partial products is generated that are then fed to the second step, which is some type of a reduction circuit. The reduction circuit reduces the set of generated partial products into two bit vectors. In the third and final step, the two remaining bit vectors are summed together using a final adder.

The first parallel multipliers, called array multipliers, were designed as an array of 3-input Full Adders (FAs) and utilized AND-gates to generate the partial products. Fig. 1.3 shows a block diagram of an unsigned 8-bit array multiplier, where the X multiplier-operand is multiplied with the Y multiplicand-operand in a $X \times Y = S$ multiplication. For each bit in the multiplier-operand (x_0 to x_7) a row of partial products is generated by connecting the bit of the multiplier-operand to the inputs of a row of 2-input AND-gates. The second input of each AND-gate is then connected to a bit of the multiplicand-operand (y_0 to y_7). The two first rows of partial products are then added together by a row of FAs. For each additional row of partial products a row of FAs is added. The row of FAs takes the row of partial products and adds it to the result from the previous row of FAs. By the end of the final row of FAs all the partial products have been added together and the result of the multiplication has been completed.

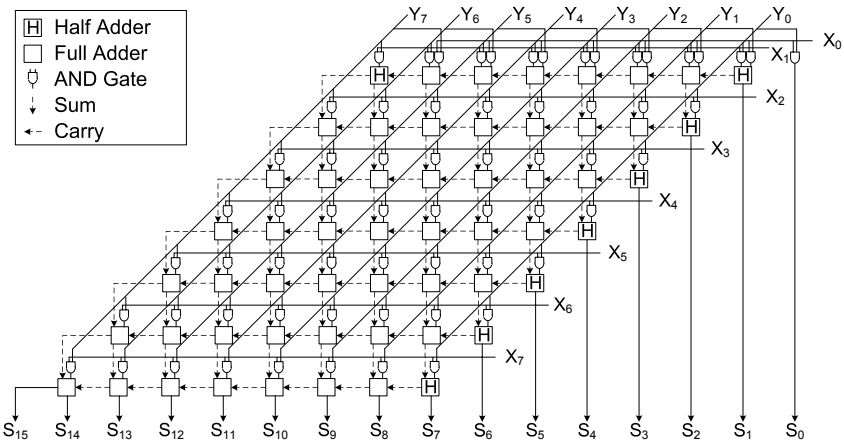


Figure 1.3: *Block diagram of an unsigned 8-bit array multiplier.*

One of the advantages with the array multiplier is that it has a regular layout and short wires. The downside is that the delay of the multiplier increases linearly with the bitwidth of the multiplier. For each added bit, a new row of full adders is needed, to sum the partial products with the previous result. Further, the gate structure of the array causes lots of glitches and therefore the array multiplier tends to have high power dissipation [10].

To reduce the delay, the number of rows of partial products that are generated can be reduced or a faster reduction circuit can be designed. Booth introduced a technique that reduces the number of generated partial products [11]. Instead of inspecting a single bit that can represent $\{0, 1\}$, the technique inspects two bits of the multiplier-operand (X) and encodes them into $\{-2, -1, 0, 1, 2\}$. The encoded bits are then multiplied with the multiplicand-operand (Y). The drawback of the Booth algorithm is that the number of rows with generated partial products is dependent on the input multiplier-operand. This makes the algorithm unsuitable for hardware implementation. MacSorley improved the technique by encoding three bits of the multiplier-operand into $\{-2, -1, 0, 1, 2\}$. The modified-Booth algorithm [6] ensures that exactly half the number of partial product rows will be generated, compared to the number of partial product rows generated using 2-input AND-gates as in the array multiplier in Fig. 1.3.

The other option to reducing delay is to improve the reduction circuit. The most common implementation choice for high performance parallel multipliers is a logarithmic-depth reduction tree. With a logarithmic-depth reduction tree the critical path is reduced from being linearly dependent on the number of partial-product rows, as for the array multiplier, to a logarithmic dependency. An added benefit with logarithmic-depth reduction trees is that they produce less glitches than array multipliers and, thus, has lower power dissipation [10]. The drawback of logarithmic-depth reduction trees, such as the popular Dadda [12], Wallace [13], and TDM [14] trees, is that they have an irregular interconnect structure. The irregularity increases the design effort, since it makes them difficult to place and route.

The first logarithmic-depth reduction tree with regular interconnect was introduced by Eriksson *et al.* with the High Performance Multiplier (HPM) [15]. The HPM reduction tree, which is a version of a Dadda tree [12], has the same logarithmic property as other logarithmic-depth reduction trees, with the advantage of a regular interconnect that makes it easier to place and route. An added benefit of the HPM reduction tree being a Dadda is that it performs better than the commonly used Wallace tree. Fig. 1.4 shows estimated power and delay values from placed and routed HPM and Wallace multipliers, in a 65 nm process technology. The figure clearly shows that the HPM reduction tree has both shorter delay and dissipates less power than a Wallace tree of equal bit-width.

A popular way of designing fast parallel multipliers is to combine the use of modified-Booth with a logarithmic-depth reduction tree, such as Wallace. The combination has the benefit of reducing the number of generated partial-product rows, which the logarithmic-depth reduction tree then quickly reduces to the inputs for the final adder. However, one disadvantage with the modified-Booth recoding scheme is that complex circuitry is needed to encode the multiplier-operand into $\{-2, -1, 0, 1, 2\}$ and then to decode the multiplicand-operand. This introduces extra logic in the critical path. Taking into account that the critical path through a logarithmic-depth reduction tree is only reduced by at most two full adder delays, when combined with modified-Booth recoding, it is not obvious that the total delay of a modified-Booth multiplier implementation will be significantly reduced.

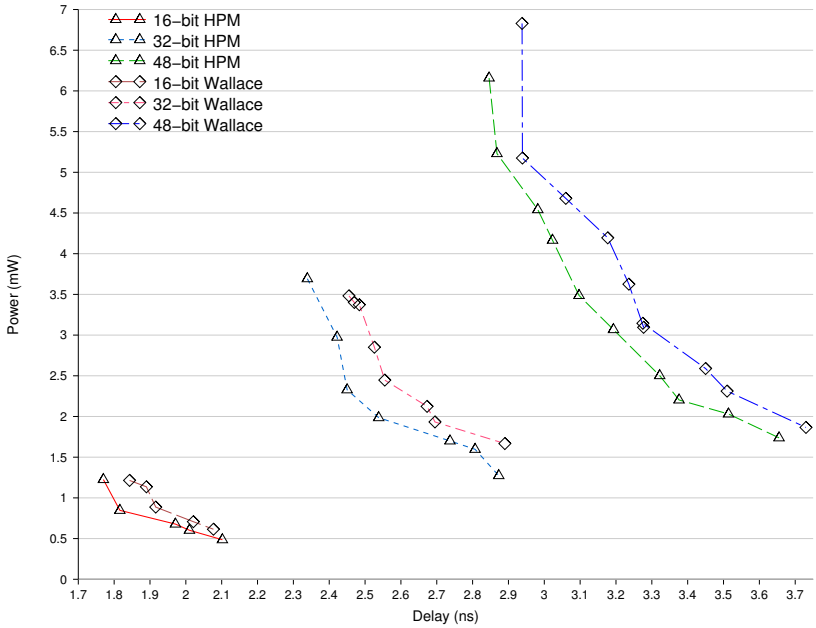


Figure 1.4: *Delay and power estimates for a 65 nm process technology of multipliers based on the HPM and Wallace reduction tree.*

In 1997 Callaway *et al.* presented a study [16] where they compare four different multiplier implementations. They come to the conclusion that the modified-Booth recoded Wallace tree is faster than a conventional Wallace tree. However, the modified-Booth recoded Wallace tree was found to dissipate more power and for the power-delay product the conventional Wallace tree was found to be the most efficient implementation.

The design practice used today is that of combining a logarithmic-depth reduction tree with modified-Booth recoding, even though Callaway *et al.* showed that a Wallace multiplier without modified-Booth recoding has a better power-delay product. An explanation can be that Callaway *et al.* made their study in a 2 μm process technology with two metal layers and extrapolated delay and power for other multiplier sizes than the used 16-bit implementation. Since then, the process technology has progressed to gate lengths of less than 100 nm,

for which wire lengths have a more dominant role for both power and delay. Further, several new Booth-recoding circuits have been proposed that claim they reduce the critical path [17, 18], which are obviously not part of Callaway's study.

Paper C presents a detailed evaluation of signed integer multipliers that are based on the HPM reduction tree and either the modified-Booth [6] or Baugh-Wooley [7] algorithm. The Baugh-Wooley algorithm was chosen because it does not require complex recoding circuits that introduce extra logic in the critical path. The Baugh-Wooley algorithm is based on simple 2-input AND or NAND-gates for its partial-product generation. However, the Baugh-Wooley algorithm does not reduce the number of partial-product rows, as the modified-Booth algorithm does. The evaluation is based on two recent Booth-recoding circuits that have been found in the literature and have shown promising results in terms of short delay. The multipliers have been implemented using VHDL and synthesized, placed, and routed in both 130 nm and 65 nm process technologies, for accurate estimations on delay, power, and area. The evaluation shows that Baugh-Wooley multipliers of sizes from 8 to 64 bits are about 20% smaller, dissipates 25-40% less power, and are at most 4% slower than modified-Booth multipliers of equal size for the 130 nm process technology. 32-bit implementations of the multipliers in a 65 nm process technology show even larger differences in area (22%) and power (60%) with only a 3% difference in delay. The study shows that the delay difference of a Baugh-Wooley and modified-Booth multiplier of equal size is so small that the benefits of smaller area and reduced power make the Baugh-Wooley multiplier the obvious choice of the two.

1.2.2 Twin-Precision Multiplier

Many of the typical embedded applications work on streaming data, where the same operation is performed on each data item of the stream. This type of applications can benefit from a technique called Single Instruction Multiple Data (SIMD), which makes it possible to execute the same operation on multiple data items in parallel, with a single instruction. For this type of applications it is also common that they work with data-widths that are less than 32 bits, henceforth called narrow-width operations.

There have been several proposals on how to implement multipliers that efficiently support applications that can benefit from narrow-width SIMD multiplications *and* those that need full-precision multiplications. Since multipliers require large area, the techniques either combine several narrow-width multipliers to perform one full-width multiplication [19–22] or they partition the partial product generation of a full-width multiplier, such that it can perform narrow-width operations [23–27].

Most of the proposed multi-precision multiplier designs [19, 20, 22–24, 26, 27] utilize modified-Booth recoding. The modified-Booth recoder generates two's complement numbers which makes it necessary to sign extend the partial product rows. Sign extending the partial product rows would require a larger reduction tree. This can be avoided by special techniques that add only a few correction bits to each partial product row. However, the sign extension prevention bits add complexity when a multi-precision multiplier is to be designed, since these bits need to be added for each possible precision to be computed by the multiplier.

Two of the proposed multi-precision multipliers [21, 25] base their design on the Baugh-Wooley algorithm, instead of the modified-Booth algorithm. The two designs have taken two different approaches: Lin's [21] is a recursive multiplier constructed from several small multipliers that are combined to form larger multipliers, while Krithivasan *et al.* [25] partition the partial product generation and utilize the reduction tree of a full precision multiplier. Earlier studies [22, 28] have shown that a recursive multiplier is slower than a multiplier based on a logarithmic-depth reduction tree.

At about the same time as Krithivasan *et al.* presented their multi-precision multiplier [25], **Paper D** presented the twin-precision technique, which also partitions the partial product array of a Baugh-Wooley multiplier into sub-arrays. The twin-precision technique allows one N-bit, one $N/2$-bit, or two $N/2$-bit multiplication(s) to be computed on an N-bit multiplier. The technique tries to minimize the impact on delay and power of the N-bit multiplication by making as few modifications as possible to a conventional N-bit HPM multiplier. The low complexity of the Baugh-Wooley multiplier allows for a straightforward solution. The only modifications needed for narrow-width operation are *i)* the

capability of forcing parts of the partial product array to zero, *ii)* the capability of negating some of the partial products, and *iii)* the addition of two extra sign bits connected to the reduction tree. The modifications introduce at most a 3-input instead of 2-input AND-gate to the critical path of the *N*-bit multiplier. The added logic is limited to *N*-2 XOR-gates, *N*/2 half-adder gates, and the replacement of $3/4$ of all $N \times N$ 2-input AND-gates to 3-input AND-gates. The twin-precision technique also allows a single *N*/2-bit multiplication to be computed. This allows for power savings when an application does not need the full precision of an *N*-bit multiplier, but cannot pack several multiplications together for SIMD operation.

When a twin-precision multiplier operates on *N*/2-bit operands, large parts of the multiplier are idle. **Paper E** presents a reconfigurable power gating technique that can be used to power off the inactive parts of the multiplier. The study shows that significant reduction in leakage power with limited impact on delay can be achieved by a customized power supply grid. In this particular study, the total power of a 16-bit twin-precision multiplier was reduced by 11% when executing 8-bit multiplications for a 130 nm process technology.

The modified-Booth algorithm is complex and gives rise to many special cases that need to be handled in a multi-precision multiplier. This complicates the design of multi-precision modified-Booth multipliers, which has a negative impact on their performance and power. In **Paper F** a twin-precision modified-Booth multiplier is compared to a twin-precision Baugh-Wooley multiplier. The comparison shows that for a 130 nm process technology a twin-precision modified-Booth multiplier is slower, dissipates more power, and is larger than a twin-precision Baugh-Wooley multiplier of equal bit-width. The study clearly shows that the modified-Booth algorithm is not suitable when implementing multi-precision multipliers.

The intention of the twin-precision technique is that it should be possible, during the design of a system, to replace a conventional multiplier with a twin-precision multiplier. The added flexibility should come at a minimal impact on the performance and power of the overall design. The twin-precision multiplier has therefore been designed to have a small or negligible impact on the performance of a conventional multiplier, when executing full-precision, *N*-bit, multi-

plications. **Paper F** presents a case study where the multiplier of an embedded 32-bit general-purpose processor is replaced by a twin-precision multiplier. The results show that the twin-precision technique has negligible impact on the performance of a general-purpose processor. Further, its SIMD capabilities allow, in this case, a Fast Fourier Transform (FFT) application to be executed 15% faster and with a 14% decrease in energy dissipation.

1.3 Task Scheduling for Multi-Core Architectures

The relentless demands for higher performance, e.g. to be able to provide high-definition (HD) video or increase the number of advanced features in mobile devices, have lead to that the performance of a single processor core is not sufficient. This is mainly due to five major challenges, so called walls, that limit the performance of a single processor core.

I) Instruction Level Parallelism Wall – A single core solution suffers from limited Instruction Level Parallelism (ILP) that can be found and exploited in sequential programs. It has been shown [29, 30] that it is difficult to reach an ILP of more than ten instructions per cycle without unlimited hardware resources.

II) Memory Wall – The increasing imbalance between the performance of a processor and its memory subsystem has lead to what has been termed the memory wall [31]. The large imbalance has lead to that the performance of the processor is unexploited, since it has to wait for data from memory when performing data accesses.

III) Power Wall – As the process technology is scaled down the leakage power has started to drastically increase. The increasing power dissipation has lead to that improvements in performance need to be matched with improvements in efficiency as well. Without improving the efficiency, the power dissipation of a single processor core will continue to increase to prohibitive levels [32].

IV) Clock Frequency Wall – Increasing the frequency of a processor typically requires deeper pipelines and more power. However, a longer pipeline increases the chance of resource conflicts, which will make the processor stall and the

penalty for miss speculations. Further, the optimal logic depth of a pipeline stage is about six to eight fan-out-of-four (FO4) delays [33], which puts a limit to the number of stages a design can be divided into. Therefore, increasing the frequency other than by process technology scaling is not easily achieved.

V) Complexity Wall – The design of a single core is becoming ever more complex and the complexity of the most advanced processors has reached such levels that they are extremely difficult to design and verify. Pollack observed that for a lead microprocessor the number of transistors was increased with a factor of about two for each new process generation, while the performance was only improved by 40% [32, 34].

With a multi-core architecture many of the challenges associated with a single-core architecture can be avoided. For a multi-core architecture it is possible to design a relatively simple processor that has a good tradeoff between power and performance. Desired system performance can then be achieved by instantiating several of these cores, which can also help to reduce the power dissipation. By adding a second core to a single processor system the theoretical performance is doubled. It is then possible to trade some of the performance gain against lower power dissipation. This can be achieved due to the quadratic dependency between power and the supply voltage, while the frequency has only a linear dependency to the supply voltage. Lowering the supply voltage reduces the power more than it reduces the performance. A parallel application consists of several threads, which makes it possible for a processor to switch to another thread instead of waiting on a memory accesses. With several threads it is also not necessary to increase the ILP of each thread to increase performance, one can instead increase the number of threads and the number of cores executing them.

Multi-core architectures require that the applications are parallelized for them to take advantage of the available performance. The parallelization consists of partitioning an application into a number of tasks (threads) that can then be distributed over several cores, henceforth called task scheduling, in order to accelerate application performance. Many of the current solutions [35–37] rely on software for partitioning the application into tasks and to distribute them. A software solution can be efficient as long as the overhead for creating and

distributing a new task is much smaller than the task itself. As the number of cores increases an application will have to be partitioned into more and, thus, smaller tasks. As the tasks become smaller, the overhead of a software solution can become prohibitively large. A natural step to reduce the task scheduling overhead is to add some type of hardware support. There exist several hardware techniques [38–41] that tries to address this problem. Kumar *et al.* proposed to add hardware queues for storing and distributing tasks [38] and showed that they are on average 70-110% faster than software task scheduling, for a set of fine-grained parallel applications.

Partitioning of applications from the multimedia domain will generally introduce dependencies between the tasks, which force the tasks to be executed in a certain order. For this kind of application workloads it is not enough to be able to efficiently create tasks and distribute them. It is also necessary to keep track of when a new task can be created. The algorithms needed for keeping track of the creation of tasks are not necessarily complex, but they can introduce large overheads. In **Paper G** it is shown that the code for calculating what task to be executed next accounts for 9% of the total execution of the parallelized sections of a Super HD H.264 decoder. The paper presents a task management unit that can offload and execute the task dependency code in parallel with other tasks. The task management unit is shown to reduce the overhead with 40-50%, compared to task scheduling through the use of a task queue.

1.4 Conclusion

Building embedded systems can be very complicated. For an efficient implementation it is necessary to consider all levels of an embedded system, from applications down to the actual implementation.

This thesis addresses issues of how to efficiently schedule tasks of an application onto different cores in a multi-core system, as well as scheduling of individual instructions onto a single core. Scheduling of instructions is limited by the hardware-software interface, which dictates in what ways a compiler/programmer can take advantage of available hardware resources. The envisioned two-tier instruction set in the FlexSoC project questions the limitation

of control given by conventional instructions sets, such as those for application-specific, general-purpose, and very-long-instruction-word processors. The envisioned two-tier instruction set gives the compiler/programmer expressive control of the hardware, which can be taken advantage of when scheduling instructions. The two-tier instruction set also opens up for new architectural features to be considered, such as the flexible interconnect and the ease of integrating hardware accelerators into the datapath of a FlexCore. However, the performance of a new architecture depends on the components it is built of. Arithmetic circuits are extensively used in all digital systems and are often timing critical. In particular, the integer multiplier is slow, power hungry, and large, relative to other arithmetic circuits. Implementation techniques that can improve the flexibility, performance, and efficiency of integer multipliers have therefore been presented. Further, as the process technologies continue to scale, leakage power has become a great concern when designing energy efficient embedded systems. To address the problem of increased power dissipation, a customized power gating technique for integer multipliers is presented.

This thesis shows the complexity involved and solutions to several relevant issues that need to be addressed when designing embedded systems.

Bibliography

[1] ARM Limited, 110 Fulbourn Road Cambridge, England CB1 9NJ, *ARM Architecture Reference Manual.*, i edition, July 2005.

[2] Steve Leibson, *Designing SOCs with Configured Cores : Unleashing the Tensilica Xtensa and Diamond Cores*, Morgan Kaufmann Publishers, 2006, ISBN-13: 978-0-12-372498-4.

[3] John Hughes, Kjell Jeppson, Per Larsson-Edefors, Mary Sheeran, Per Stenström, and Lars "J." Svensson, "FlexSoC: Combining Flexibility and Efficiency in SoC Designs," in *Proceedings of the IEEE NorChip Conference*, 2003.

[4] Bita Gorjiara and Daniel Gajski, "Custom Processor Design Using NISC: a Case-Study on DCT Algorithm," in *Workshop on Embedded Systems for Real-Time Multimedia*, September 2005, pp. 55–60.

[5] Bita Gorjiara, Mehrdad Reshadi, and Daniel Gajski, "Designing a Custom Architecture for DCT Using NISC Design Flow," in *Asia and South Pacific Conference on Design Automation*, 2006, pp. 116–117.

[6] O.L.MacSorley, "High Speed Arithmetic in Binary Computers," in *Proceedings of the IRE*, January 1961, vol. 49, pp. 67–97.

[7] Charles R. Baugh and Bruce A. Wooley, "A Two's Complement Parallel Array Multiplication Algorithm," *IEEE Transactions on Computers*, vol. 22, pp. 1045–1047, December 1973.

[8] David A. Patterson and John L. Hennessy, *Computer Organization & Design, The Hardware/Software Interface*, Morgan Kaufman Publishers Inc., 2nd edition, 1998.

[9] "Embedded Microprocessor Benchmark Consortium," `http://www.eembc.org`.

[10] Jeroen Leijten, Jef van Meerbergen, and Jochen Jess, "Analysis and Reduction of Glitches in Synchronous Networks," in *Proceedings of the European Design and Test Conference*, March 1995, pp. 398–403.

[11] Andrew D. Booth, "A Signed Binary Multiplication Technique," *Quarterly Journal of Mechanics and Applied Mathematics*, vol. 4, no. 2, pp. 236–240, 1951.

[12] Luigi Dadda, "Some Schemes for Parallel Adders," *Alta Frequenza*, vol. 34, no. 5, pp. 349–356, May 1965.

[13] Christopher S. Wallace, "A Suggestion for a Fast Multiplier," *IEEE Transactions on Electronic Computers*, vol. 13, pp. 14–17, February 1964.

[14] Vojin G. Oklobdzija, David Villeger, and Simon S. Liu, "A Method for Speed Optimized Partial Product Reduction and Generation of Fast Parallel Multipliers Using an Algorithmic Approach," *IEEE Transactions on Computers*, vol. 45, no. 3, pp. 294–306, March 1996.

[15] Henrik Eriksson, Per Larsson-Edefors, Mary Sheeran, Magnus Själander, Daniel Johansson, and Martin Schölin, "Multiplier Reduction Tree with Logarithmic Logic Depth and Regular Connectivity," in *IEEE International Symposium on Circuits and Systems*, May 2006, pp. 5–8.

[16] Thomas K. Callaway and Jr. Earl E. Swartzlander, "Power-Delay Characteristics of CMOS Multipliers," in *Proceedings of the 13th IEEE Symposium on Computer Arithmetic*, June 1997, pp. 26–32.

[17] Wen-Chang Yeh and Chein-Wei Jen, "High-Speed Booth Encoded Parallel Multiplier Design," *IEEE Transactions on Computers*, vol. 49, no. 7, pp. 692–701, July 2000.

[18] Steven K. Hsu, Sanu K. Mathew, Mark A. Anders, Bart R. Zeydel, Vojin G. Oklobdzija, Ram K. Krishnamurthy, and Shekhar Y. Borkar, "A 110 GOPS/W 16-bit Multiplier and Reconfigurable PLA Loop in 90-nm CMOS," *IEEE Journal of Solid-State Circuits*, vol. 41, no. 1, pp. 256–264, January 2006.

[19] K.C. Tang, Angus K. M. Wu, Anthony S. Fong, and Derek C. W. Pao, "Integrated Partition Integer Execution Unit for Multimedia and Conventional Applications," in *Proceedings of the IEEE Conference on Electronics, Circuits, and Systems*, September 1998, vol. 2, pp. 103–107.

[20] Martin S. Schmookler, Michael Putrino, Charles Roth, Mukesh Sharma, Anh Mather, Jon Tyler, Huy Van Nguyen, Mydung N. Pham, and Jeff Lent, "A Low-Power, High-Speed Implementation of a PowerPC^{TM}Microprocessor Vector Extension," in *Proceedings of the 14th IEEE Symposium on Computer Arithmetic*, April 1999, pp. 12–19.

[21] Rong Lin, "Reconfigurable parallel inner product processor architectures," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 9, no. 2, pp. 261–272, April 2001.

[22] Pedram Mokrian, Majid Ahmadi, Graham Jullien, and W.C. Miller, "A Reconfigurable Digital Multiplier Architecture," in *Proceedings of the IEEE Canadian Conference on Electrical and Computer Engineering*, 2003, pp. 125–128.

[23] Dimitri Tan, Albert Danysh, and Michael Liebelt, "Multiple-Precision Fixed-Point Vector Multiply-Accumulator Using Shared Segmentation," in *Proceedings of the 16th IEEE Symposium on Computer Arithmetic*, June 2003, pp. 12–19.

[24] Ya-Lan Tsao, Wei-Hao Chen, Ming Hsuan Tan, Maw-Ching Lin, and Shyh-Jye Jou, "Low-Power Embedded DSP Core for Communication Systems," *EURASIP Journal on Applied Signal Processing*, vol. 2003, no. 1, pp. 1355–1370, 2003.

[25] Shankar Krithivasan and Michael J. Schulte, "Multiplier Architectures for Media Processing," in *Proceedings of Signals the 37th IEEE Asilomar Conference on Systems and Computers*, November 2003, vol. 2, pp. 9–12.

[26] Alber Danysh and Dimitri Tan, "Architecture and Implementation of a Vector/SIMD Multiply-Accumulate Unit," *IEEE Transactions on Computers*, vol. 5, no. 4, pp. 284–293, May 2005.

[27] Yan Sun, Lanfei Dong, Daheng Yue, Shaoqing Li, and Minxuan Zhang, "Multiple-Precision Subword-Parallel Multiplier Using Correction-Value Merging Technique," in *Proceedings of the 7th IEEE International Conference on ASIC*, October 2007, pp. 48–51.

[28] Albert N. Danysh and Earl E. Swartzlander Jr., "A Recursive Fast Multiplier," in *Proceedings of the 32nd Asilomar Conference on Signals, Systems and Computers*, November 1998, vol. 1, pp. 197–201.

[29] David V. Wall, "Limits of Instruction-Level Parallelism," Tech. Rep. WRL-93-6, HP Labs Technical Reports, November 1993.

[30] Hsien-Hsin Lee, Youfeng Wu, and Gary Tyson, "Quantifying Instruction-Level Parallelism Limits on an EPIC Architecture," in *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*, April 2000, pp. 21–27.

[31] William A. Wulf and Sally A. McKee, "Hitting the Memory Wall: Implications of the Obvious," *ACM SIGARCH Computer Architecture News*, vol. 23, no. 1, pp. 20–24, 1995.

[32] Patrick P. Gelsinger, "Microprocessors for the new Millennium: Challenges, Opportunities, and new Frontiers," in *Technical Digest of the IEEE International Conference on Solid-State Circuits Conference*, February 2001, pp. 22–25.

[33] M.S Hrishikesh, Norman P. Jouppi, Keith I. Farkas, Doug Burger, Stephen W. Keckler, and Premkishore Shivakumar, "The Optimal Logic Depth per Pipeline Stage is 6 to 8 FO4 Inverter Delays," in *Proceedings of the 29th International Symposium on Computer Architecture*, May 2002, pp. 14–24.

[34] Fred J. Pollack, "New Microarchitecture Challenges in the Coming Generations of CMOS Process Technologies," Keynote. 32nd IEEE International Symposium on Microarchitecture, November 1999.

[35] Intel Corporation, *Intel Thread Building Blocks*, 2007, Revision 1.6.

[36] OpenMP Architecture Review Board, *OpenMP Application Program Interface*, November 2004, Version 2.5.

[37] Message Passing Interface Forum, *MPI-2: Extensions to the Message-Passing Interface*, July 1997.

[38] Sanjeev Kumar, Christopher J. Hughes, and Anthony Nguyen, "Carbon: Architectural Support for Fine-Grained Parallelism on Chip Multiprocessors," in *IEEE International Symposium on Computer Architecture*, June 2007, pp. 734–740.

[39] Kyriakos Stavrou, Costas Kyriacou, Paraskevas Evripidou, and Pedro Trancoso, "Chip Multiprocessor Based on Data-Driven Multithreading Model," *International Journal of High Performance Systems Architecture*, vol. 1, no. 1, pp. 24–43, 2007.

[40] Pelge Avieli, Oded Rubenov, and Nomrod Bayer, "Designing a Central Synchronization/Scheduling Unit For Multiprocessors," in *IEEE Convention of the Electrical and Electronic Engineers*, April 2000, pp. 495–498.

[41] Nimrod Bayer and Ran Ginosar, "High Flow-Rate Synchronizer/Scheduler Apparatus and Method for Multiprocessors," April 1993.

PAPER A

M. Thuresson, **M. Själander**, M. Björk,
L. Svensson, P. Larsson-Edefors, and P. Stenström

FlexCore: Utilizing Exposed Datapath Control for Efficient Computing

Springer Journal of Signal Processing Systems
Accepted for publication, March 4, 2008.

A

FlexCore: Utilizing Exposed Datapath Control for Efficient Computing

We introduce FlexCore, the first exemplar of an architecture based on the FlexSoC framework. Comprising the same datapath units found in a conventional five-stage pipeline, the FlexCore has an exposed datapath control and a flexible interconnect to allow the datapath to be dynamically reconfigured as a consequence of code generation. Additionally, the FlexCore allows specialized datapath units to be inserted and utilized within the same architecture and compilation framework.

This study shows that, in comparison to a conventional five-stage general-purpose processor, the FlexCore is up to 40% more efficient in terms of cycle

count on a set of benchmarks from the embedded application domain. We show that both the fine-grained control and the flexible interconnect contribute to the speedup. Furthermore, according to our VLSI implementation study, the FlexCore architecture offers both time and energy savings.

The exposed FlexCore datapath requires a wide control word. The conducted evaluation confirms that this increases the instruction bandwidth and memory footprint. This calls for efficient instruction decoding as proposed in the FlexSoC framework.

A.1 Introduction

Cost- and performance-sensitive application areas, such as cellular phones and other battery-powered multimedia devices, are not well served by present-day general-purpose computing platforms. To meet user expectations of features and battery capacity, designers instead resort to highly heterogeneous systems where a collection of specialized hardware accelerators (built for encryption, image and video coding, audio playback, etc) are controlled by an embedded microprocessor, such as an ARM core. For cost reasons, several accelerators will typically be collocated with the microprocessor on a single System-on-Chip (SoC).

The present practice has drawbacks. Tasks outside the set originally intended may not benefit from the computing capacity available: computing resources hidden inside an accelerator may be difficult or impossible to use in ways other than those considered by the accelerator designer. Even when possible, the software constructs necessary to access a "hidden" hardware block bear little resemblance to ordinary code.

For ease of software development and maintainability, a uniform hardware/-software interface, similar to those offered by general-purpose processors (GPPs), would be highly desirable; but present-day GPPs cannot compete with heterogeneous SoCs in terms of performance at a given power level. Merging the accelerator datapath elements into the GPP infrastructure would be possible in principle, but a very wide instruction word would be required to make fine-grained control possible. A wide instruction word potentially increases the

memory footprint of an application. For typical embedded systems, memory is expensive in terms of money and power. Because of the tight power and cost constraints together with the high volumes used, it is important to keep the memory usage and I/O activity down.

In our approach, called FlexSoC [1], we address these problems by moving away from the conventional GPP instruction set architecture (ISA) and use a wide control word. This allows us to control the datapath units in the datapath at a much more fine-grained level. Other important differences between our approach and the standard GPP-like ISA is that each control word, or Native-ISA (N-ISA), controls the datapath in the current cycle only and that forwarding needs to be done statically. Previous work on exposed control has shown significant performance improvements [2] when used within a hardware/software co-design system. In contrast, the FlexSoC approach includes a (extendable) pipeline together with a flexible instruction decoder.

To better understand the requirements of the instruction decoder, our study investigates how the instruction bandwidth as well as the static code size is affected by the exposed control. In particular, we note that even though the wide instructions allow for more efficient control, the number of instructions needed are still large, thus increasing the static code size considerably compared to a conventional GPP-version. In fact, all benchmarks are larger and require three times as much bandwidth as a GPP-version.

Consequently, in the FlexSoC framework we propose to use a programmable instruction decoder. This decoder allows the compiler to use a compressed Application-Specific ISA (AS-ISA) for each application, or set of applications, to be executed. Applications are stored as AS-ISA instructions that are expanded on-the-fly to N-ISA instructions when fetched from the program memory. Figure A.1 illustrates this scheme. As will be shown, the findings in this article clearly motivate such a scheme.

In this paper, we make the following main contributions. 1) We introduce the FlexCore datapath, a datapath with exposed control based on the FlexSoC framework. 2) We evaluate the microarchitecture in detail and show that it enables better performance in cycle-count and execution time as well as energy efficiency. A detailed study of the interconnect shows the usage patterns of our

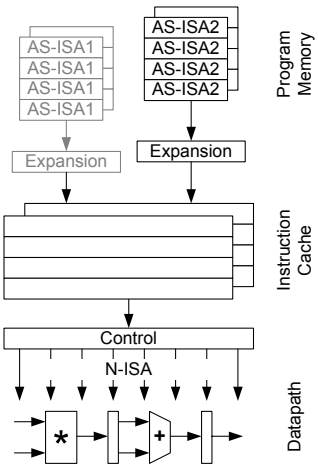


Figure A.1: *AS-ISA instruction decoding into N-ISA instructions*

applications and layout implementations of the schemes allow us to do qualitative comparisons. 3) In this framework, we also show that the exposed pipeline comes at a cost of both static code-size and instruction bandwidth.

This paper is organized as follows: The FlexCore architecture is introduced in Section A.2 and A.3, followed by compilation techniques used, Section A.4. Section A.5 presents how the FlexCore architecture was evaluated, and the results are presented in Section A.6. Related work is discussed in Section A.7; the paper is concluded in Section A.8.

A.2 The Baseline FlexCore Architecture

Application studies in our field of interest, in particular comparisons [3] of two audio compression standards (MP3 and Ogg Vorbis), have convinced us that full GPP functionality is necessary for flexibility. To offer the full programmability of a GPP, we have therefore decided to include all the datapath units necessary to emulate a full-featured processor in our baseline architecture. We have opted to use a conventional, single-issue, five-stage pipeline similar to the Hennessy-Patterson 32-bit DLX and MIPS R2000 as a template [4]. This is not a high-

performance processor design; in FlexSoC, however, our ambition is to provide application performance mainly through the use of specialized accelerators and fine-grained control rather than through conventional methods. Thus, our core is designed to be flexible and gradually extensible, with accelerators according to application requirements. Moreover, recent research have shown that simple cores can be very energy efficient and successfully used in high-performance systems [5, 6].

The datapath is fully exposed and is controlled through a 91-bit wide N-ISA control word. The baseline FlexCore consists of four datapath units (see Figure A.2): Register File, Arithmetic Logic Unit (ALU), Load/Store Unit (LS Unit), and a Program Counter Unit (PC Unit), with all units connected to a flexible, fully connected interconnect. To allow for GPP functionality, each datapath-unit output port is connected to a data register. The data registers act as pipeline registers in the various pipeline configurations that can be assembled using the flexible interconnect. Thus, it is easy to create different pipelines by routing a result from one datapath unit to the next.

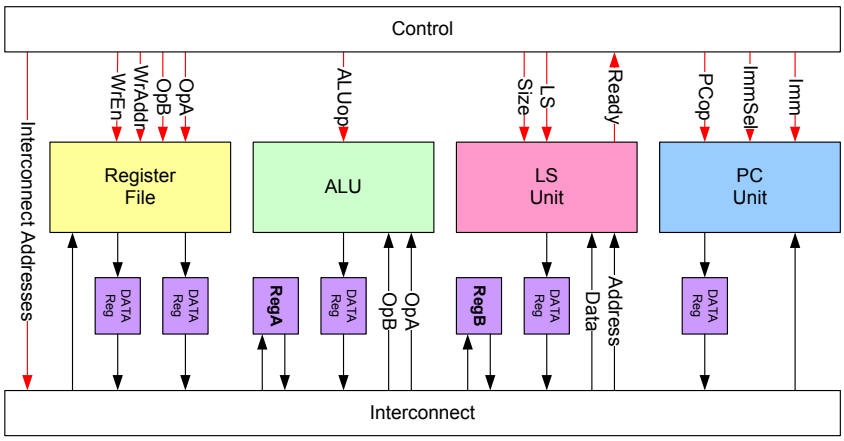


Figure A.2: *Illustration of a baseline FlexCore. Note that each DATA Reg also has a stall signal not shown in the figure.*

To allow instructions to be scheduled on the FlexCore in the same way as on a conventional five-stage pipeline, two data registers (RegA and RegB) have

been included. The two data registers are used in the execute and load/store stage of a conventional GPP and allow data to bypass the ALU and LS unit.

The baseline FlexCore can act as a GPP since it can emulate a conventional pipeline and run instructions in the same way. The FlexCore architecture also allows for high resource utilization due to the flexibility in scheduling and the fine-grained N-ISA control word.

A.2.1 N-ISA: Exposed Datapath

As described in Section A.1, the datapath of a FlexCore can be precisely controlled through the N-ISA control word. The N-ISA format depends heavily on the architecture and its datapath units. Figure A.3 shows the N-ISA control word for the baseline FlexCore architecture. Starting from the least significant bit, the control word consists of bits that control the interconnect, the PC Unit (which also includes the 32-bit immediate value), the two data buffers, the Load/Store Unit, the ALU, and finally the Register File.

Figure A.3: *FlexCore N-ISA control word. The different fields are: Interconnect (24 bits), PC (37 bits, of which 32 bits are immediate), D (Data buffers, 2 bits), LS (Load/Store, 5 bits), A (ALU, 5 bits), and Register (18 bits). The total length is 91 bits.*

The expected functionality of the datapath units allows us to identify the role of most of the bits in a straightforward manner. For example, the bits controlling the register file contain the fields denoting which two registers to read and which register to write. These bits also contain a write enable signal and two stall signals for each read port data register. The PC Unit handles the immediate value, and the ImmSel signal selects if the value emitted from the PC Unit should be the current Immediate value, or the address of the next instruction (which is used in jump-and-link-like instructions).

The N-ISA includes the bits controlling the interconnect. Since each output can be associated to any input in every cycle, the number of bits, n, needed to control an N-input, M-output interconnect is $n = M \cdot \lceil \log_2(N) \rceil$.

In each cycle, an N-ISA word controls all units in the datapath as well as the interconnect. The exposed-datapath approach differs from the conventional pipelined control word, which is found in general-purpose processors and digital-signal processors, and in which one control word (corresponding to one instruction) contains information about all pipeline stages, for this *and* consecutive cycles.

As can be seen in Figure A.3, the N-ISA word consists of 91 bits. Compared to a conventional 32-bit GPP, the FlexCore requires an instruction bandwidth that is almost three times as large, in order to keep the datapath busy. The N-ISA is clearly not an efficient representation. Therefore, FlexSoC assumes a reconfigurable instruction decoder/expander in the hardware/software interface. It comes as no surprise that our results confirm that both the static code size and the instruction bandwidth must be addressed.

The goal of giving the compiler complete control of the hardware has the drawback that binary compatibility between processors with different datapath architectures is lost. With a reconfigurable instruction decoder, as proposed in the FlexSoC framework [1], it may however be possible to reuse the same AS-ISA for different hardware configurations, but that is a topic for future research and, thus, not addressed in this work.

In Section A.6, we analyze performance gains from using an exposed datapath and a flexible interconnect with the same datapath units as a conventional five-stage pipeline.

A.3 Extensions to the Baseline FlexCore

An advantage of the FlexCore architecture is that it can be extended with application-specific accelerator units, simply by adding more ports to the flexible interconnect and extending the N-ISA control word to include control signals for the new units. Since different units are treated equally, we hope to avoid complex ad-hoc solutions usually found in irregular interconnects. For instance, for each unit added to a conventional pipeline, the forwarding network with control logic has to be modified. A conventional fixed pipeline depth also makes it cumbersome to add datapath units and utilize them efficiently: either the new

unit is put in the execute stage and can thereby only be used if the ALU is not used; or a new pipeline stage is added, which changes the architecture considerably; or the unit can be added as a co-processor, which causes communication overheads.

A fully connected crossbar guarantees that the interconnect will not restrict the scheduling of operations on the datapath units. This motivates its use in the explorative phase of the design. As seen in Section A.6, the full connectivity may not be needed for a given application domain; this provides an opportunity to reduce the area and power requirements, once a suitable collection of datapath units has been determined.

A.3.1 Multiplier Extension

For this study, the baseline FlexCore has been extended with a multiplier in order to be able to efficiently execute embedded application benchmarks, such as the Fast Fourier Transform (FFT). The 32-bit multiplier, which is pipelined into two stages to balance its critical path to the other datapath units, is connected to the interconnect that has been extended with two input and two output ports. The two output ports deliver operands to the multiplier, while the result is divided into two 32-bit values. Each value is connected to an input port of the interconnect. One of the ports carries the 32 least significant bits of the result, while the other port carries the 32 most significant bits of the result, as shown in Figure A.4.

An N-ISA instruction for the extended FlexCore consists of 108 bits. The multiplier has no control signals, since it is only capable of executing one operation. The extension to the N-ISA is due to extra bits for addressing the added ports in the interconnect, as well as two enable signals for the LSB and MSB register that is holding the result after a multiplication.

A.4 Compiling for FlexCore

The flexibility of architectures based on the FlexCore concept enables numerous compilation strategies. Given the ability of a FlexCore to emulate a con-

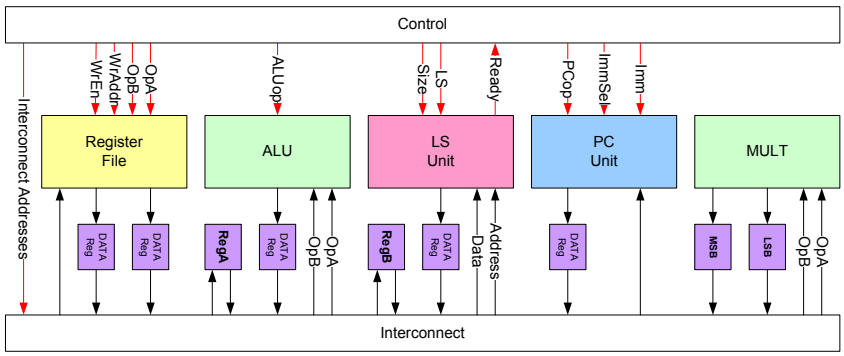


Figure A.4: *Illustration of a baseline FlexCore extended with a multiplier. Note that each DATA Reg has a stall signal, while RegA, RegB, LSB, and MSB have an enable signal which is not shown.*

ventional GPP, we chose as our initial approach to translate GPP-like assembly instructions into N-ISA code. As this work will show, the FlexCore can indeed emulate a conventional five-stage pipeline in real time.

The translation of single GPP instructions to N-ISA code is straightforward. We use the same datapath structure as a five-stage pipeline, but the instruction fetch stage is implicitly handled by the FlexCore control unit. In other words, each instruction spans four cycles. The first cycle uses the immediate port and the read ports of the Register File. The second cycle uses the ALU and RegA. The third cycle uses the Load/Store Unit and RegB. Finally, the fourth cycle uses the write port of the Register File. Sequences of such instructions are merged using static optimization techniques (see Section A.4.1 below) to achieve pipelining and forwarding.

Obviously, making the FlexCore operate as a GPP is not the best way to exploit this architecture. However, we chose to execute GPP programs on the FlexCore to establish a performance baseline. Even though the FlexCore interconnect allows for communication between any two units, GPP instructions use only the paths corresponding to those found in the GPP. Therefore, we aim to compile high-level code down to N-ISA along the lines of other compilation methods for general datapaths [7]. This enables the pipeline length and

structure to be changed as often as needed, and allows for programs to use the datapath units in any order. Currently, profiling allows the programmer to manually schedule performance-critical regions. The manually scheduled parts are written in an RTN[1]-like format that can be interleaved with GPP instructions.

A.4.1 Instruction-Level Static Code Optimization

Translating GPP assembly code into N-ISA code yields a number of N-ISA instruction sequences that should be scheduled as tightly as possible, overlapping each other as allowed by resource conflicts and data dependencies. Resource conflicts are not an issue in the case of GPP assembly instructions, due to their pipelined structure. Data dependencies are more important, since consecutive operations often use the same register.

When one operation uses the contents of a register that is updated by the previous operation, several cycles can often be saved by forwarding: taking the value directly from the datapath unit that produces it, rather than waiting for it to be written to the register file first.

Pipelined processors usually do these optimizations at run-time. On the FlexCore, however, they must be done statically due to the exposed control word. The basic operation is to compose two sequences of N-ISA instructions sequentially, with as much overlap as possible. This is done by annotating each instruction with information about what resources that are used, and the status of all registers. Each register can have status *available*, *unavailable*, or *rerouted(p)*, where p is the name of an output port of a datapath unit. Normally, registers are marked as *available*, which means that their value can be read from the register file. A register is *unavailable* when a new value for the register is currently being computed and is not yet available. When the value is available but not yet written to the register file, the *rerouted(p)*-annotation tells the compiler where the value can be found. In such a case, the register read is omitted, and the value is fetched from the port p instead of the register port.

Techniques such as these are not restricted to rescheduled conventional pipelined programs, but can be used for any N-ISA code. They help determine whether any two annotated N-ISA sequences are composable with a fixed

[1] Register Transfer Notation.

overlap. To find out the maximal possible overlap, we begin by composing them without overlap, then with one cycle overlap, thereafter with two cycles overlap, and so on until we fail. It may be possible to continue even further, but then we must perform a more careful analysis to make sure that no write order conflicts occur. However, we do not expect such aggressive optimizations to have a significant efficiency impact.

Even though the compiler at times needs to make pessimistic choices not to violate any data dependencies when scheduling, our results show that the generated code is only marginally slower on GPP-code compared to a fixed five-stage pipeline with dynamic forwarding and pipelined control.

A.4.2 Scheduling Opportunities

To exemplify what type of optimizations are possible with the fully connected interconnect and the wide control word, we list four general optimizations that are possible on the FlexCore architecture, but not on a GPP.

Dynamic pipeline depth

A conventional processor typically has a very rigid pipeline, generally for a five-stage pipeline they are called IF/ID/EX/MEM/WB. In the DLX case, each has a fixed one cycle latency, though more advanced pipelines might have different latency for various operations in the execution unit and have hardware support for out-of-order execution.

In FlexCore, the interconnect allows us to change the structure of the pipeline during execution of the program. Consider the case of a tight loop which performs advanced arithmetic calculations, but does not have any memory accesses. During this time, the pipeline can be modified to bypass the MEM-stage completely and thus reduce the latency for each instruction.

Static forwarding

Forwarding is a key principle in pipelines to increase performance. The time for a value to go from producer to consumer needs to be minimized, and in most architectures, a complex set of forwarding links and control logic are used

when forwarding opportunities are dynamically identified. With the fully connected interconnect available in the FlexCore network and the fine-grained control available, it is possible for the compiler to find these opportunities and make the forwarding of operands statically. Thus, eliminating the need for forwarding control logic.

Static optimization

Another important feature of the exposed control is that less redundant operations need to be performed in hardware. If two consecutive arithmetic GPP-instructions operate on the same register value, it is clear that the second instruction will overwrite the result of the first. If forwarding is used to pass the value from the first to the second instructions the value produced by the first instruction does not need to be written to the register file. With the granularity of GPP instructions, only dynamic hardware schemes can remove such redundant writes. With FlexSoC, each write is exposed and we can statically decided weather to write the result or not.

Instruction parallelism

Even though the instruction stream in RISC code is sequential, the execution of the operations does not have to be linear as long as the result is *as if* the instruction were executed sequentially. The amount of Instruction Level Parallelism (ILP) available in the program as well as the hardware resources available both limits the amount of parallelism that can be exploited. With the finer control possible with FlexSoC, we are able to find more opportunities to perform various operations in parallel. Also, if there is a bottleneck, the flexible interconnect will make it easy to add more resources which the compiler can utilize.

Example Schedule

To illustrate the different scheduling optimizations that can be done for the FlexCore datapaths we will consider the three GPP assembly instructions shown in Figure A.5.

```
ADD   $1, $3, 16
ADD   $1, $5, $1
LW    $3, 0($9)
```

Figure A.5: *Example of three consecutive GPP assembly instructions.*

In a conventional five-stage pipeline the instructions could be scheduled as shown in Figure A.6(a). The instructions are scheduled one after the other, each with a latency of four cycles. The load-word (LW) instruction is independent of the other two instructions and could have been scheduled earlier. However, the total number of cycles to execute the three instructions would not be affected.

	ID	EX	MEM	WB
1:	Read $3 & IM16			
2:	Read $1 & $5	ADD		
3:	Read $0 & $9	ADD	NOP	
4:		ADD	NOP	Write $1
5:			LW	Write $1
6:				Write $3

(a) GPP Schedule

	ID	EX	MEM	WB
1:	$3, IM16, & $9			
2:	Read $5	ADD	LW	
3:		ADD		Write $3
4:				Write $1

(b) FlexCore Schedule

Figure A.6: *Instruction scheduling on a GPP and FlexCore datapath.*

The schedule of the three instructions in Figure A.5 for the FlexCore datapath is shown in Figure A.6(b). One can directly notice that the schedule for the FlexCore datapath is only four cycles, instead of six as for the GPP schedule. This is achieved by applying the scheduling optimizations described above: *i)* Each instruction has a latency of three instead of the conventional four. *ii)* Forwarding is not explicitly shown in the schedules but are performed statically to transfer the result of the ADD in cycle two to the ADD in cycle three. *iii)* Register $1 is only written once, thus the write-port of the register-file can be used by another instruction. *iv)* Since the load-word instruction has an address-offset of zero, it is unnecessary to compute a new address from which to make the load from. This together with the available write-port of the register-file allows the load-word instruction to be scheduled in parallel with the first ADD instruction.

A.5 Experimental Framework

To evaluate the performance of FlexCore, four different benchmarks from the Embedded Microprocessor Benchmark Consortium (EEMBC) were selected. The four benchmarks are the Fast Fourier Transform (FFT), Autocorrelation (Autocor) and Viterbi Decoder (Viterbi) from the Telecom benchmark suite and the High pass grey-scale (RGBHPG) filter from the Consumer benchmark suite. All selected benchmarks were executed to completion and the result presented excludes code belonging to the test-harness.

To distinguish between the performance gains achieved by an exposed datapath and the flexible interconnect, a FlexCore with only the interconnects present in a conventional GPP pipeline has also been simulated; it is identified as “Exposed GPP” in the tables.

We chose MIPS as the ISA for the GPP pipeline, since it is supported by mature free-ware tools and cross-compilers. Each of the benchmarks was compiled using a cross-compiler to MIPS assembly with the default optimization flags for EEMBC (-O2). The assembly code was profiled to identify computational kernels, which subsequently were manually scheduled, using RTN notations, for the Exposed GPP and the FlexCore. All the benchmarks were man-

ually scheduled without doing any algorithmic changes or optimizations, such as loop unrolling. Each assembly instruction was translated into RTN notations and scheduled as early as possible onto the given datapath. An instruction can therefore be scheduled earlier than given by the original assembly code, if there are no data dependencies and no resource conflicts. Thus, three versions of each benchmark were generated:

- **GPP**- The output of the FlexCore compiler. As described in Section A.4, the compiler schedules the instructions in the same way as if they would have been executed on a conventional GPP.
- **Exposed GPP** - The most frequent instructions (inner loops) have been manually scheduled using only the communication paths of a conventional GPP. For example, RTN-operations make it possible to remove unnecessary register updates.
- **FlexCore** - Manual scheduling of the same instructions as for the Exposed GPP but with the full interconnect of the FlexCore. For example, data can be moved directly from the ALU to the register file, without updating the buffer registers.

The code was executed on a cycle-accurate simulator that models a FlexCore connected to an ideal memory architecture, with single cycle latency. The simulator has been verified against a VHDL implementation (see Section A.5.1 below). By using CRC-checks on the output of the benchmarks, we have verified that they execute correctly.

A.5.1 Hardware Implementation

To evaluate the performance in terms of delay, power/energy, and area, VHDL implementations were created for the different architectures. Note that no control logic has been implemented for the evaluated architectures in our comparison. The exposed GPP is a conventional GPP, where the control logic has been removed. Therefore, when disregarding control logic and instruction fetch, the exposed GPP and conventional GPP are equivalent. The influence of control

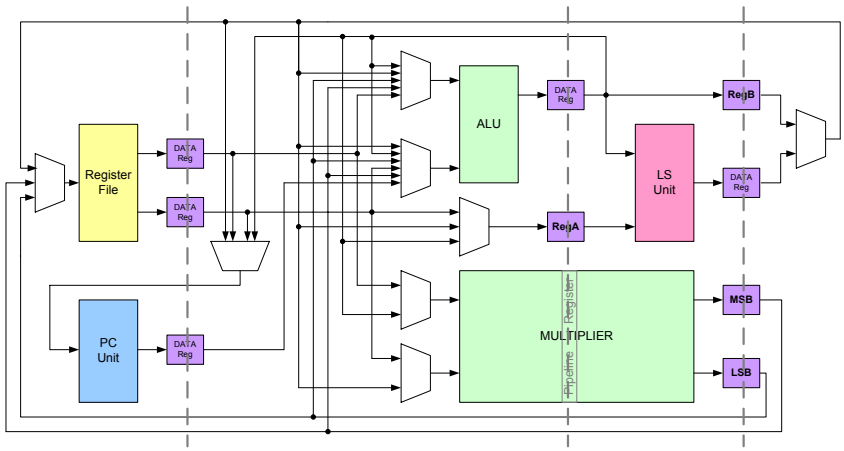


Figure A.7: *Schematic of the datapath for the GPP and exposed datapath. Only the communication paths for data are shown, without any control signals.*

logic and instruction fetch on the performance of the different architectures is an issue addressed within the FlexSoC project, but this is not a topic of this paper.

Figure A.7 shows a detailed schematic of the communication paths that are available to the GPP and the exposed datapath. The datapath is inspired by the DLX and MIPS R2000 datapaths [4]. The multiplier is an intrinsic part of the datapath and takes its inputs from the register file. To improve performance, it is possible to forward results from the ALU and LS Unit directly. The output registers of the multiplier do not work as conventional pipeline registers, where a new value is clocked in for each new clock cycle. Instead these registers hold the result from the previous multiplication until a new multiplication is performed.

The baseline FlexCore datapath using the flexible interconnect was designed such that an input port to a datapath unit can be connected to any output port of any datapath unit. This creates a fully connected crossbar switch as interconnect.

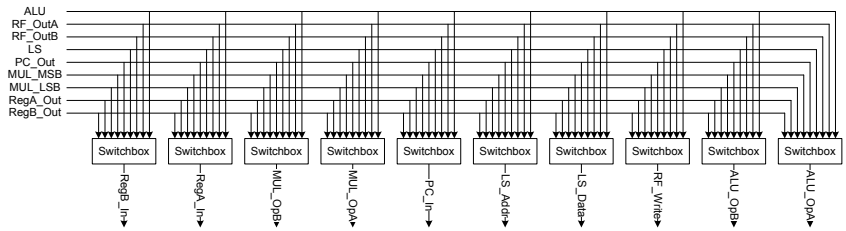


Figure A.8: *Illustration of the crossbar switch of the fully connected interconnect.*

Figure A.8 illustrates how the crossbar switch is constructed from a number of switchboxes. Each input to a datapath unit (in Figure A.7) as well as to the two registers RegA and RegB is connected to its own switchbox. Each switchbox is in turn connected to all the outputs of the datapath units. The switchbox functions as a multiplexer, according to Figure A.9.

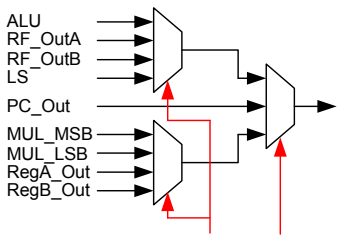


Figure A.9: *Illustration of a switchbox.*

The VHDL descriptions were synthesized, placed, and routed (Cadence First Encounter [8]) using a commercially available 65 nm low power technology, with standard threshold voltage. Delay and power estimations were done on the placed and routed netlists with extracted resistance and capacitance and for the worst case corner at 125 degrees Celsius. The power estimations are given for the maximum clock frequency for each of the datapaths.

Table A.1: *Application Cycle Count*

		GPP	Exposed GPP	FlexCore
Autocor	Full	1.5k (100%)	1.3k (88%)	1.0k (67%)
	Loop	1.3k (100%)	1.1k (86%)	0.8k (62%)
FFT	Full	58k (100%)	47k (80%)	37k (64%)
	Loop	43k (100%)	31k (71%)	22k (50%)
RGBHPG	Full	3.4M (100%)	3.2M (93%)	2.8M (82%)
	Loop	3.3M (100%)	3.2M (97%)	2.8M (86%)
Viterbi	Full	268k (100%)	268k (100%)	243k (90%)
	Loop	218k (100%)	222k (102%)	198k (90%)

A.6 Results

A.6.1 Cycle-Count Evaluation

In the first set of experiments we have executed the benchmarks and measured the dynamic cycle count. Table A.1 shows the dynamic cycle count for the benchmarks. The columns labeled Loop show the number of cycles spent in the manually scheduled inner loops. Here we see that at least 75% of the executed cycles are spent in the inner loops. The percentage in the table allows us to compare the cycle count between the three architectures.

The FlexCore architecture is between 10% and 50% better in cycle count compared to the GPP for all the applications. The Exposed GPP improves only some of the applications, with FFT being the best with 20% improvement, while Viterbi could not be improved at all.

In order to understand the difference in performance among the benchmarks, the utilization of the resources in the datapath was also monitored. As with many embedded media benchmarks we saw that the inner loops performed many arithmetic operations on register values.

Figure A.10 shows the relative number of cycles the ALU was used, for the various benchmarks. For all applications, the ALU utilization in the FlexCore is higher than that in the Exposed GPP, whose ALU utilization in turn is higher

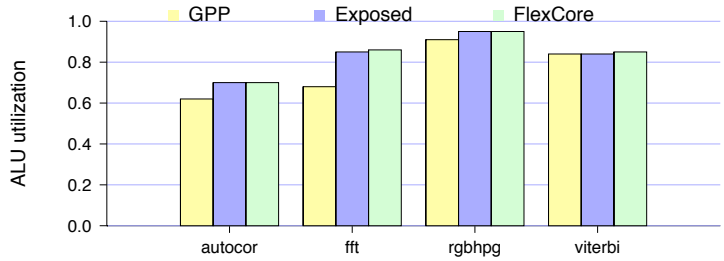


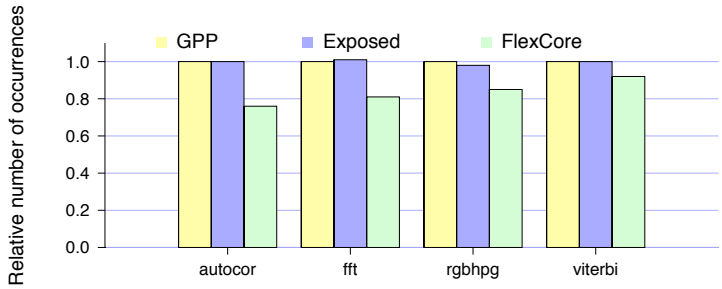
Figure A.10: *Percentage of cycles the ALU is used.*

than in the GPP. Comparing the results with the dynamic cycle count, we see that the benchmarks with the smallest ALU utilization gained the most by the FlexCore architecture. Autocor and FFT, which have only 60% and 70% ALU utilization, improved by 33% and 36% in cycle count with FlexCore, while RGBHPG and Viterbi with a 90% and 85% ALU utilization only gained 18% and 10%.

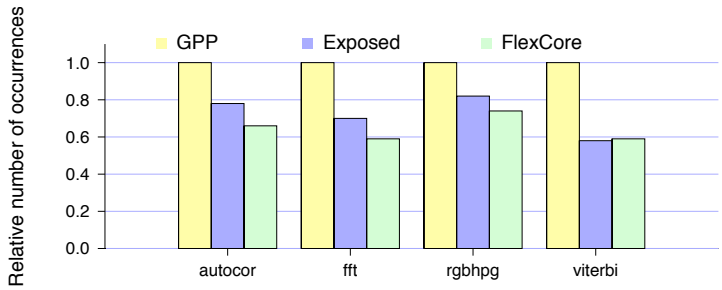
Furthermore, the number of ALU operations executed in total decreases in FlexCore. Figure A.11(a) shows that between 5% and 22% of all ALU operations executed by the GPP are redundant and have been removed for FlexCore. An example of redundant ALU operations is the address calculation with a zero offset for load and store operations.

In a GPP, all calculated values are written to the register file. Nevertheless, all written values that are used in the next instruction will be routed through the forwarding network. If the value is never again read from the register file, the write is unnecessary. In the FlexCore architecture, it is possible to skip the generation of such writes, as long as it is possible to statically find such superfluous register writes in the program. Figure A.11(b) shows the relative number of register writes for the different architectures compared to the GPP version. On average, the Exposed GPP allows us to remove 28% of all register writes and for FlexCore 36% are removed. This reduces the contention of the register file as well as saves power[2].

[2]It also complicates exception handling, which is however not the topic of this paper.



(a) ALU operations



(b) Register Writes

Figure A.11: *Number of operations for the benchmarks normalized to the GPP.*

A.6.2 Performance Evaluation

Cycle count alone is not sufficient to make a comparison of the different architectures. One also needs to consider implementation aspects. Table A.2 shows the result of timing, power, and area estimations for the FlexCore, the Exposed GPP and a conventional GPP. The implementations are done in a 65 nm technology, but values for a 130 nm technology are also included for reference. As previously explained, we have modeled the microarchitecture without control and thus the implementation result for the GPP and the Exposed GPP are the same.

The layout implementations were performed for minimum delay, and as expected, the more modern 65 nm version has better delay values and worse power. For the remainder of the paper, we will use the values for 65 nm as

our baseline. For the FlexCore architecture, the extra paths require 18% extra space. More importantly, the delay of the critical path is increased by 6% and the power dissipation is increased by 18%.

Table A.2: *Delay, Power, and Area Estimates*

(a) 65 nm - Baseline

Datapath	Timing (ns)	Power (mW)	Area (mm^2)
GPP	2.14 (100%)	7.36 (100%)	0.093 (100%)
Exposed GPP	2.14 (100%)	7.36 (100%)	0.093 (100%)
FlexCore	2.27 (106%)	8.69 (118%)	0.110 (118%)
Pruned	2.15 (100%)	7.33 (100%)	0.099 (106%)

(b) 130 nm

Datapath	Timing (ns)	Power (mW)	Area (mm^2)
GPP	2.89 (100%)	5.70 (100%)	0.260 (100%)
Exposed GPP	2.89 (100%)	5.70 (100%)	0.260 (100%)
FlexCore	3.21 (111%)	5.92 (104%)	0.293 (109%)
Pruned	3.02 (105%)	5.90 (104%)	0.267 (103%)

The high overhead in terms of delay and especially power makes a fully connected interconnect impractical. We have therefore investigated which paths of the interconnect that are actually used and if any paths can be pruned without losing any performance in terms of cycle count.

In the GPP architecture, there are 33 possible communication paths, while the fully connected interconnect offers 90 paths. Table A.3 lists all these paths and highlights the ones that are used by our four benchmarks. Paths marked GPP are used when executing GPP code; additionally, when the full interconnect is used, paths marked Flex are also used. Here we see that out of the 90 paths available in the fully connected interconnect, 43 are *never* used in the benchmarks of this study.

Figure A.12 illustrates the actual utilization of the paths in our fully connected interconnect. Each path listed in Table A.3 is represented on the x-axis; for each benchmark, the bars show how often the link is used during the full execution. For all the benchmarks, the path between register output A and the

Table A.3: *Interconnect Paths used by the benchmark on the GPP and on the FlexCore*

	PC In	RF Write	ALU OpA	ALU OpB	LS Address	LS Data	RegA In	RegB In	MULT OpA	MULT OpB
PC Out	1-	2-Flex	3-	4-GPP	5-	6-	7-	8-	9-	10-
RF ReadA	11-GPP	12-Flex	13-GPP	14-	15-	16-	17-	18-	19-GPP	20-
RF ReadB	21-GPP	22-	23-	24-GPP	25-Flex	26-Flex	27-GPP	28-Flex	29-	30-GPP
ALU Out	31-GPP	32-Flex	33-GPP	34-GPP	35-GPP	36-Flex	37-GPP	38-GPP	39-GPP	40-
LS Out	41-GPP	42-GPP	43-GPP	44-GPP	45-	46-	47-GPP	48-	49-Flex	50-GPP
RegA Out	51-	52-	53-Flex	54-Flex	55-Flex	56-GPP	57-	58-	59-	60-Flex
RegB Out	61-GPP	62-GPP	63-GPP	64-GPP	65-Flex	66-	67-GPP	68-	69-	70-
MULT LSB	71-	72-GPP	73-GPP	74-GPP	75-	76-	77-Flex	78-	79-	80-
MULT MSB	81-	82-GPP	83-GPP	84-GPP	85-	86-	87-	88-	89-	90-

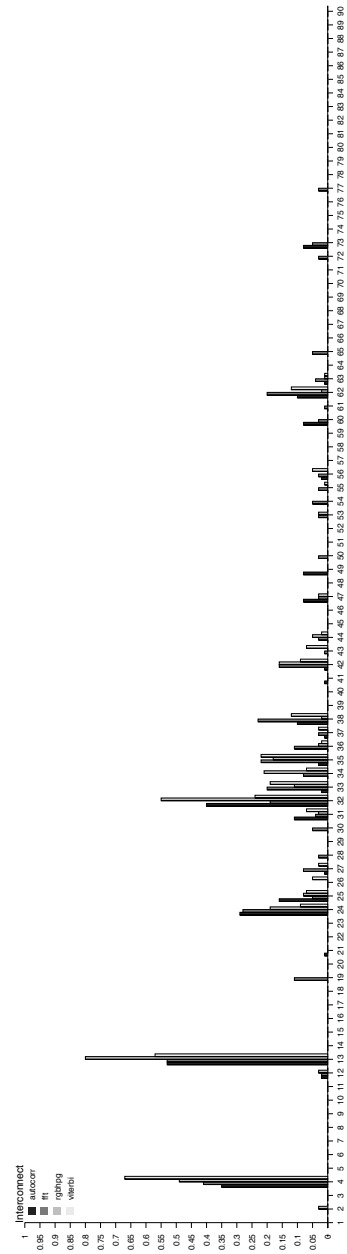


Figure A.12: *Details on utilization of interconnect links. Table A.3 shows the link-ID used on the x-axis. The y-axis shows how often the link is used during the total execution of the benchmark.*

ALU is very important (path number 13). Even though the trend is similar among all the four benchmarks, there are some variations. The total number of paths used in the benchmarks is very different between FFT (32) and the other three benchmarks (Autocor uses 18, FFT 32, RGBHPG 18, and Viterbi 19 paths). The distribution of used links also vary between FFT and the rest. The ten most used links in FFT cover 73% of all the transfers while the corresponding numbers are 87%, 93%, and 83% for Autocor, RGBHPG, and Viterbi. Note that the links do not necessarily need to be the same in the different benchmarks.

Since all data movements are statically scheduled by the compiler, it is possible to prune idle communication paths from the flexible interconnect, without losing performance. Table A.2 include the results of an architecture with the pruned interconnect as shown in Table A.3. The results show that pruning the interconnect improves power dissipation as well as delay and area.

It is also enlightening to calculate the minimum number of links necessary to account for 95% of the transfers. This metric is listed in Table A.4. Out of the used links, between 67% and 81% of the links use 95% of the transfers. This uneven distribution suggest that even more pruning is possible with small effect on performance. The combined statistics show that even though FFT uses a larger set of paths than the other benchmarks, it is not a true superset of the ones needed by Autocor, RGBHPG and Viterbi.

Table A.4: *Number of links needed for 95% of all transfers*

Benchmarks	#links
Autocor	13
FFT	26
RGBHPG	12
Viterbi	15
All	30

By combining the results from the placed and routed architectures with the cycle count, the true performance of the different approaches is shown. Table A.5 shows the total time as well as the amount of energy ($power \cdot clock\ period \cdot cycle\ count$) needed to execute each benchmark.

Table A.5: *Execution time and energy dissipation.*

	Autocor		FFT		RGBHPG		Viterbi	
	Time (μs)	Energy (nJ)	Time (μs)	Energy (nJ)	Time (ms)	Energy (μJ)	Time (μs)	Energy (μJ)
GPP	3.2 (100%)	24 (100%)	124 (100%)	912 (100%)	7.3 (100%)	54 (100%)	574 (100%)	4.2 (100%)
Exposed GPP	2.8 (88%)	21 (88%)	101 (81%)	743 (81%)	6.8 (93%)	50 (93%)	574 (100%)	4.2 (100%)
FlexCore	2.3 (72%)	20 (83%)	84 (68%)	730 (80%)	6.4 (88%)	56 (104%)	552 (96%)	4.8 (114%)
Pruned	2.2 (69%)	16 (67%)	80 (65%)	586 (64%)	6.0 (82%)	44 (81%)	522 (91%)	3.8 (90%)

These figures show that all four applications are executed faster on the FlexCore than on the GPP, when the cycle time is also considered. Compared to the Exposed GPP, we notice that the performance boost is attributed to both the exposed control and the flexible interconnect. With GPP as reference, the Exposed GPP gives up to 19% performance improvement, while the FlexCore delivers improvements up to 32% on the same benchmark (FFT).

The energy result on the other hand is not as promising. Comparing the FlexCore architecture to the GPP shows that only two of the benchmarks are executed with less energy (17% for Autocor and 20% for FFT) while two expend more energy (4% for RGBHPG and 14% for Viterbi). This shows that if energy is a first-level constraint, the improvement in cycle count is not large enough to outweigh the penalty associated with the fully connected interconnect.

Pruning the interconnect, as described earlier, reduces the delay and power significantly. This is reflected in the low execution time and energy dissipation when executing the four benchmarks. The FlexCore with the pruned interconnect is on average 19% faster and 14% less energy dissipating than the GPP.

A.6.3 Static Code Size

The total number of instructions for the benchmarks is shown in Table A.6. The results are presented for both the full benchmark (excluding library code) and for the manually scheduled inner loop. First focusing on the full benchmarks, we see that the optimized versions are about the same or slightly smaller than the GPP versions.

Note that any change in the number of instruction is the result from the manually scheduled inner loops. Therefore we also present the number of instructions for these. Here we see that the FlexCore architecture has made it possible to decrease the number of instructions significantly for some benchmarks. For example, the inner loop of the FFT needs only half the number of cycles as on the conventional GPP. However, since an N-ISA instruction is about three times as large as a conventional GPP instruction, the total static code size for FlexCore is still larger than for a conventional GPP.

Table A.6: *Code Size (number of instructions) for both application and the inner loop for each benchmark.*

		GPP	Exposed GPP	FlexCore
Autocor	Full	60 (100%)	61 (102%)	56 (93%)
	Loop	16 (100%)	17 (106%)	12 (75%)
FFT	Full	432 (100%)	424 (98%)	414 (96%)
	Loop	42 (100%)	30 (71%)	21 (50%)
RGBHPG	Full	197 (100%)	198 (101%)	191 (97%)
	Loop	43 (100%)	42 (98%)	37 (86%)
Viterbi	Full	366 (100%)	367 (100%)	360 (98%)
	Loop	66 (100%)	67 (102%)	61 (92%)

The increase of instructions seen for the Exposed GPP originates from some of the manual scheduling, which added prefix code to handle the first iteration of loops.

A.7 Related Work

Reconfigurable architectures is an active area of research. Dedicated hardware is becoming less attractive because of huge initial costs, long time to market, and inability to adapt to new and changing standards. Reconfigurable hardware is a promising approach to address these problems, without forsaking the performance of dedicated hardware. Hartenstein has compiled a thorough survey of reconfigurable architectures [9]. Many modes of reconfigurability have been proposed: reconfigurable accelerators may be connected to a standard pipeline [10]; or reconfigurable tiles may be orchestrated to solve given problems [11, 12]. In contrast, the FlexSoC approach as presented here employs reconfigurability only in the instruction decoding hardware, leaving the actual data processing to highly efficient dedicated hardware.

The exposed datapath concept has recently been used in the No Instruction Set Computer (NISC) [2, 7, 13] project, where the control pipeline is removed

and the controller emits a wide instruction word each cycle. Co-design refinement of hardware and software is used to reach the desired performance. The reported speedups are comparable to those we see for the FlexCore example. However, the static code size of a NISC program is claimed to be comparable to that of a GPP. While this might be true for a co-design approach where common complex operations can be implemented with few control bits, we have not seen the same results for the FlexCore architecture. In FlexSoC, we rely on compression and run-time expansion to solve the code size problem. Increased controllability of the datapath has been motivated by the reduction in hardware complexity, in a similar way as in the Transport Triggered Architecture [14].

Liang *et al* [15] propose an architecture based on a reconfigurable interconnect and show good performance for some domain-specific computations. It is, however, not clear how the results translate to a wider domain of applications.

A common way to accelerate multimedia applications is to add sub-word parallelism within the datapath units (SIMD). This technique is used both in modern general-purpose computers and specific media processors. For a five-stage DLX implementation, Nia and Fatemi report a speedup of more than a factor of three with only minor growth in chip area [16]. The approach is orthogonal to those proposed here and would seem to make a fruitful addition to a FlexSoC core.

Similarly to FlexSoC, the FITS project [17–19] also envisions the use of flexible instruction decoders. Application profiling allows the selection of a 16-bit application-specific ISA that gives the same performance as the 32-bit baseline case. FlexSoC combines a similar application-specific ISA approach with the performance gains offered by the exposed datapath and the flexible interconnect.

The translation envisioned in the FlexSoC project is somewhat similar to microcode processing, where a complex ISA is broken down into micro-operations that are executed on the pipeline. The main purpose of microcode is to separate the architecture from the implementation and the microcode is usually derived from the already given ISA. In FlexSoC, this constraint is relaxed and the AS-ISA can be created by the compiler to fit the needs of the applications.

A.8 Conclusion

The exposed datapath of FlexCore offers distinct performance benefits when compared to a GPP with corresponding datapath units. The flexible interconnect network further improves performance, and also allows special-purpose datapath units to be integrated while maintaining a uniform programming interface. With knowledge of the datapath structure, a compiler can realize these performance benefits. Additionally, it is always possible to execute programs as if they had been compiled for MIPS and at cycle counts comparable to a standard MIPS implementation.

We have analyzed four embedded applications and shown that for this set of applications, FlexCore is between 10% and 40% faster in terms of cycle count compared to a conventional five-stage GPP with the same datapath units. This speedup is attributed to both the exposed datapath and the flexible interconnect. Using cycle times obtained from placed and routed layouts, we show that this cycle count translates to a total execution-time improvement of up to 30%. We also show that it is feasible to prune the fully connected interconnect without losing any performance in cycle count. For our benchmarks, one third of the possible links account for about 95% of all the data transfers. This allows for a more energy-efficient implementation and also improves the cycle time. However, the boost in performance comes at a cost of both instruction bandwidth and static code size: the FlexCore instructions are about three times as wide as a 32-bit GPP instruction. The number of static instructions is reduced for the FlexCore, but not by a factor of three. Therefore, the static code size will be larger for an application compiled for FlexCore. A reconfigurable instruction decoder, as proposed in the FlexSoC framework, is clearly needed to reduce both the static code size and the instruction bandwidth.

Future work includes evaluating a reconfigurable instruction decoder together with FlexCore. Different compression schemes can be expected to be more or less suited to the ISA transformations needed, and to carry different implementation costs. Configuration of the instruction decoder could be a one-time event; but run-time, on-demand reconfiguration offers intriguing possibilities, where several tasks, each with a distinct AS-ISA, could share the same hardware.

Acknowledgment

We thank our FlexSoC colleagues (Hughes, Jeppson, Sheeran) for support and helpful discussions, Marius Grannæs for valuable comments, and Jonas Ferry for his help during the project. The FlexSoC project is sponsored by the Swedish Foundation for Strategic Research.

Bibliography

[1] John Hughes, Kjell Jeppson, Per Larsson-Edefors, Mary Sheeran, Per Stenström, and Lars "J." Svensson, "FlexSoC: Combining Flexibility and Efficiency in SoC Designs," in *Proceedings of the IEEE NorChip Conference*, 2003.

[2] M. Reshadi, B. Gorjiara, and D. Gajski, "Utilizing Horizontal and Vertical Parallelism with No-Instruction-Set Compiler for Custom Datapaths," in *International Conference on Computer Design (ICCD)*, October 2005.

[3] Jan Mårts and Tomas Carlqvist, "A Hardware Audio Decoder Using Flexible Datapaths," Msc thesis, Chalmers University of Technology, March 2006.

[4] David A. Patterson and John L. Hennessy, *Computer Organization & Design, The Hardware/Software Interface*, Morgan Kaufman Publishers Inc., 2nd edition, 1998.

[5] Poonacha Kongetira, Kathirgamar Aingaran, and Kunle Olukotun, "Niagara: A 32-Way Multithreaded Sparc Processor," *IEEE Micro*, vol. 25, no. 2, pp. 21–29, 2005.

[6] Saisanthosh Balakrishnan, Ravi Rajwar, Mike Upton, and Konrad Lai, "The Impact of Performance Asymmetry in Emerging Multicore Architectures," *SIGARCH Computer Architecture News*, vol. 33, no. 2, pp. 506–517, 2005.

[7] M. Reshadi and D. Gajski, "A Cycle-Accurate Compilation Algorithm for Custom Pipelined Datapaths," in *International Symposium on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, September 2005.

[8] *Cadence Encounter User Guide Version 6.2*.

[9] Reiner Hartenstein, "A Decade of Reconfigurable Computing: a Visionary Retrospective," in *Proceedings of Design, Automation and Test in Europe, 2001*, March 2001, pp. 642–649.

[10] Zhi Alex Ye, Andreas Moshovos, Scott Hauck, and Prithviraj Banerjee, "CHIMAERA: a high-performance architecture with a tightly-coupled reconfigurable functional unit," in *Proceedings of the 27th Annual International Symposium on Computer Architecture*, New York, NY, USA, 2000, pp. 225–235, ACM Press.

[11] Michael Bedford Taylor, Walter Lee, Jason Miller, David Wentzlaff, Ian Bratt, Ben Greenwald, Henry Hoffmann, Paul Johnson, Jason Kim, James Psota, Arvind Saraf, Nathan Shnidman, Volker Strumpen, Matt Frank, Saman Amarasinghe, and Anant Agarwal, "Evaluation of the RAW Microprocessor: An Exposed-Wire-Delay Architecture for ILP and Streams," in *Proceedings of the 31st annual international symposium on Computer architecture*, Washington, DC, USA, 2004, p. 2, IEEE Computer Society.

[12] Karthikeyan Sankaralingam, Ramadass Nagarajan, Haiming Liu, Changkyu Kim, Jaehyuk Huh, Doug Burger, Stephen W. Keckler, and Charles R. Moore, "TRIPS: A Polymorphous Architecture for Exploiting ILP, TLP, and DLP," *ACM Transactions on Architecture and Code Optimization*, vol. 1, no. 1, pp. 62–93, 2004.

[13] Bita Gorjiara, Mehrdad Reshadi, and Daniel Gajski, "Designing a Custom Architecture for DCT Using NISC Design Flow," in *Asia and South Pacific Conference on Design Automation*, 2006, pp. 116–117.

[14] Henk Corporaal, "TTAs: Missing the ILP Complexity Wall," *Journal of Systems Architecture*, vol. 45, no. 12-13, pp. 949–973, 1999.

[15] Xiaoyao Liang, Akshay Athalye, and Sangjin Hong, "Dynamic Coarse Grain Dataflow Reconfiguration Technique for Real-Time Systems Design," in *Proceedings of the 32th Annual International Symposium on Computer Architecture*, 2005, pp. 3511–3514.

[16] Elham Khorsandi Nia and Omid Fatemi, "Multimedia Extensions for DLX Processor," in *Proceedings of the 10th IEEE International Conference on Electronics, Circuits and Systems*, 2003, pp. 1010–1013.

[17] Allen Cheng, Gary Tyson, and Trevor Mudge, "FITS: Framework-Based Instruction-Set Tuning Synthesis for Embedded Application Specific Processors," in *Proceedings of the 41st Design Automation Conference*, 2004, pp. 920–923.

[18] Allen Cheng, Gary Tyson, and Trevor Mudge, "PowerFITS: Reduce Dynamic and Static I-Cache Power Using Application Specific Instruction Set Synthesis," in *IEEE International Symposium on Performance Analysis of Systems and Software*, 2005, pp. 32–41.

[19] A. C. Cheng and G. S. Tyson, "High-Quality ISA Synthesis for Low-Power Cache Designs in Embedded Microprocessors," *IBM Journal of Research and Development*, vol. 50, no. 2, pp. 299–309, 2006.

PAPER B

M. Själander, P. Larsson-Edefors and M. Björk

A Flexible Datapath Interconnect
for Embedded Applications

IEEE Computer Society Annual Symposium on VLSI
Porto Alegre, Brazil, May 9-11, 2007, pp. 15-20.

B

A Flexible Datapath Interconnect for Embedded Applications

We investigate the effects of introducing a flexible interconnect into an exposed datapath. We define an exposed datapath as a traditional GPP datapath that has its normal control removed, leading to the exposure of a wide control word. For an FFT benchmark, the introduction of a flexible interconnect reduces the total execution time by 16%. Compared to a traditional GPP, the execution time for an exposed datapath using a flexible interconnect is 32% shorter whereas the energy dissipation is 29% lower. Our investigation is based on a cycle-accurate architectural simulator and figures on delay, power, and area are obtained from placed-and-routed layouts in a commercial 0.13-μm technology. The results from our case studies indicate that by utilizing a flexible interconnect, significant performance gains can be achieved for generic applications.

B.1 Introduction

Previous studies [1, 2] have shown that by exposing the datapath, through the use of a wide control word, significant performance gains can be obtained. With an exposed datapath the compiler/programmer gets total control of the resources of the datapath for each executed cycle. This may allow for a more efficient utilization of datapath resources when scheduling an application's operations onto the datapath, since the compiler is not limited to the fixed control given by an Instruction Set Architecture (ISA). The previous studies focused on hardware/software co-design in that a datapath was crafted to a specific application, or possibly to an application domain. The performance for an application can indeed be drastically improved by adapting the datapath to the specific needs of resources and communication paths. However, adapting the datapath to a particular application's needs will most likely hamper the datapath's performance when used for other applications.

A study on a Discrete Cosine Transform (DCT) application showed a 20% reduction in execution time, when the datapath of a MIPS processor became exposed [1, 2]. The DCT study indicates that the performance of a General Purpose Processor (GPP), whose datapath and control traditionally are co-optimized for general-purpose processing, can benefit from exposing the datapath. However, when introducing the concept of an exposed datapath, it may be wise to reconsider what datapath components and interconnects that we use. After all, the datapath of a traditional GPP has been carefully optimized together with its ISA. Thus, it may be possible to further improve the general-purpose processing efficiency by adapting the datapath to the use of an exposed, wide control word.

In this study, the overall purpose is to ascertain if a flexible interconnect, in combination with the exposed datapath, can offer performance gains for *both general-purpose and dedicated processing*. The insertion of a flexible interconnect allows data to be efficiently routed between datapath units, thus, potentially improve the utilization of these units, beyond what was possible by the optimized ISA. We will later show that the execution time indeed can be reduced by introducing a flexible interconnect into an exposed GPP datapath.

The interconnect evaluation is conducted on an experimental platform called FlexCore, which has been developed within the FlexSoC project [3]. The FlexCore platform makes it possible to model different architectures having the wide control word of an exposed datapath. Each modeled architecture is used to evaluate application performance and to generate a Hardware-Description Language (HDL) representation of the architecture. To obtain values on delay, power, and area, each generated HDL representation is taken through a conventional ASIC backend tool flow for a 0.13-μm technology.

This paper is organized as follows: The architectural and experimental framework is presented in Section B.2 and B.3. Section B.4 presents the FFT case study that has been conducted. Section B.5 presents the results from the case study, followed by a discussion in Section B.6. The paper is finally concluded in Section B.7.

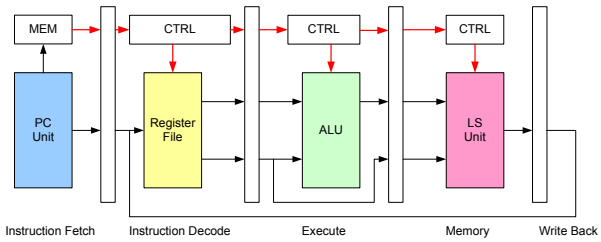
B.2 Architectural Framework

To evaluate what is the impact on performance when introducing a flexible interconnect into an exposed general-purpose datapath, three different architectures have been compared against each other. The compared architectures are illustrated in Figure B.1.

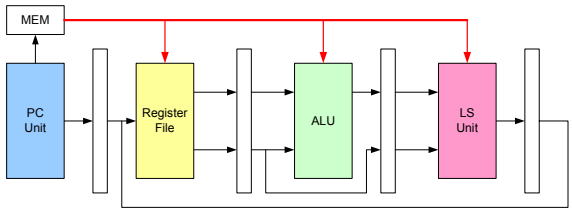
The first architecture is that of a traditional five-stage GPP pipeline, as shown in Figure B.1(a). In a traditional GPP, the movements of data are restricted to those defined by the Instruction Set Architecture (ISA) and the GPP's forwarding paths.

The second architecture is that of a GPP pipeline with an exposed datapath, Figure B.1(b). Here the programmer/compiler has total control of the whole datapath for each executed cycle. This makes the control more fine grained and gives the compiler greater freedom to move data between the datapath units, whose communication options are no longer dictated by an ISA.

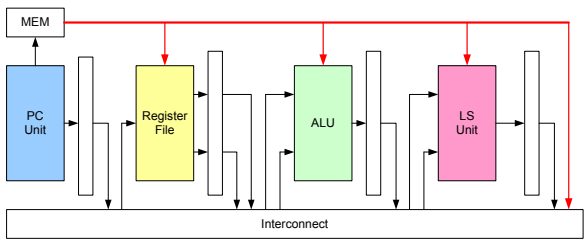
To take advantage of a flexible interconnect, the control word needs to be wider. A traditional GPP would not benefit from a flexible interconnect, since its rigid (ISA-defined) control limits the communication paths that could have been used in the datapath. Thus, the final architecture that we will evaluate



(a) GPP Datapath



(b) Exposed Datapath



(c) Exposed Datapath with Flexible Interconnect

Figure B.1: *Illustrations of the three architectures that have been compared against each other in this study.*

is that of an exposed datapath that has a fully connected crossbar switch as interconnect. Here all inputs and outputs of the datapath units are connected to the interconnect switch, as shown in Figure B.1(c).

A fully connected interconnect can appear to be an overkill in terms of flexibility. Still, we use this as a reference point for the maximum execution-cycle performance improvement that can be achieved, when there are no restrictions

on how data can move between the datapath units. A more realistic interconnect is when some of the communications paths are removed; those with no or negligible impact on application execution-cycle performance. This pruning of communication paths reduces delay, power, and area for the interconnect.

We will in the following sections compare the conventional GPP in Figure B.1(a) to *i)* the exposed datapath in Figure B.1(b) and to *ii)* the exposed datapath with a flexible interconnect in Figure B.1(c). This comparison will enable us to distinguish what performance gains are the result of exposing the datapath and what performance gains are the result of the flexible interconnect.

B.2.1 Modeling of the Architectures

We represent the three different architectures by using an experimental platform called FlexCore [4, 5], which has been developed within the FlexSoC project. The baseline FlexCore is inspired by a conventional single-issue, five-stage processor, but its datapath is fully exposed through a 91-bit wide control word (Figure B.2). The baseline FlexCore consists of four datapath units: a register file, an Arithmetic Logic Unit (ALU), a Load/Store Unit (LS Unit), and a Program Control Unit (PC Unit). All these datapath units are attached to a fully connected interconnect of crossbar type. To allow for GPP functionality each datapath unit's output port is connected to a data register, which acts as pipeline register in the various pipeline configurations that can be assembled using the interconnect. To allow instructions to be scheduled on the FlexCore in the same way as on a conventional five-stage pipeline, two data registers are included (RegA and RegB in Figure B.2). These registers are traditionally used in the execute and load/store stage of a five-stage GPP to allow data to bypass the ALU and LS Unit.

The three different architectures that are to be compared can be viewed as subsets of the FlexCore architecture. Actually, the third architecture, in which the datapath is exposed and has a flexible interconnect, is identical to FlexCore. We can represent the other two architectures simply by the way instructions are scheduled and by limiting the communication paths to those available for the architectures.

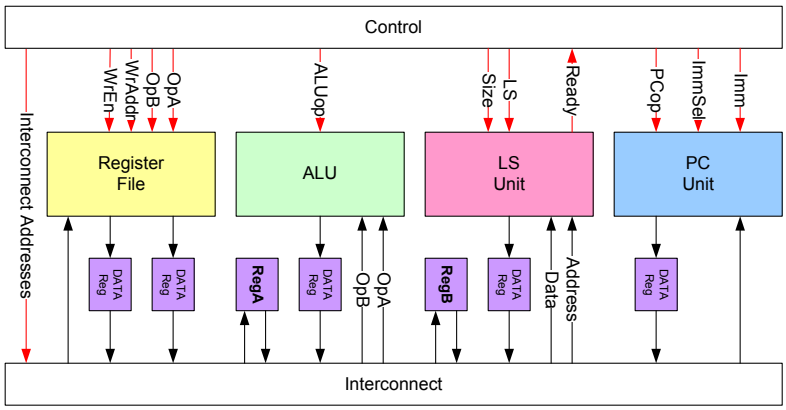


Figure B.2: *Illustration of a baseline FlexCore. Note that each DATA Reg, RegA, and RegB also has an enable signal (not shown in the figure).*

A compiler has been created for the FlexCore architecture. This takes GPP assembly code as input and schedules it as if the underlying architecture was a five-stage pipeline. The instruction scheduling of GPP assembly code onto the FlexCore architecture achieves an execution-cycle performance comparable to that of a traditional GPP.

By utilizing the datapath units more efficiently through the exposed control word, which can be viewed as a very expressive instruction, the instructions can be scheduled within fewer cycles. We model the exposed datapath by manually scheduling the instructions of an application for maximum datapath-unit utilization, while obeying the constraint of the rigid communication paths of the five-stage GPP datapath. Two examples of efficient scheduling are *i)* when we choose not to write temporary values to the register file and *ii)* when pipeline resources permit scheduling of instructions in parallel.

Similar to the exposed datapath, we model the exposed datapath with a flexible interconnect by using manual scheduling of the same part of the application. However, now we have no restrictions on the communication paths: Any input to a datapath unit can read its value from any output of all the datapath units. This corresponds to a fully connected crossbar switch as interconnect, with all inputs and outputs of the datapath units connected to it.

B.3 Experimental Framework

A set of tools, consisting of a compiler, a cycle-accurate simulator, and a Hardware-Description Language (HDL) generator, has been created for the FlexCore architecture. These three tools are used in our study on flexible interconnects and are consequently presented briefly in the following:

B.3.1 Compiler

The current version of the FlexCore compiler takes MIPS assembly code as input and generates a set of so-called Native ISA (N-ISA) instructions, which correspond to the instructions for the exposed datapath. The N-ISA instructions can be used for simulation, either by using the FlexCore simulator or by using standard HDL simulators with the HDL code obtained from the FlexCore HDL generator.

The compiler assumes a logical five-stage pipeline for scheduling the MIPS instructions onto the FlexCore architecture. For example, a fetched ADD instruction is translated into four partial N-ISA instructions. A partial N-ISA instruction consists only of the control signals needed for a particular instruction in a particular pipeline stage. The four partial N-ISA instructions for an ADD consist of: *i)* the control signals for reading the operands from the register file, *ii)* the control signals for the ALU to take the operands from the register file and perform an ADD operation, *iii)* the control signals to write the result from the ALU to one of the buffers, in order to bypass the load/store unit, and *iv)* the control signals for writing the result back to the register file.

The compiler performs static scheduling of the partial N-ISA instructions from several MIPS instructions, such that they overlap as much as possible, without causing conflicts between the control signals of different partial N-ISA instructions. When the partial N-ISAs are statically scheduled, the operands might not be available in the register file when they are to be read. The compiler therefore has to keep track of which values in the register file are valid. If a partial N-ISA instruction is requesting an invalid value from the register file, the compiler tries to either forward the value from the datapath or that particular instruction has to be delayed until the requested value is available.

The compiler does not have support for forwarding between N-ISA instructions from different basic blocks. This can cause a minor performance penalty compared to a conventional GPP, which uses dynamic forwarding.

It is currently only possible to take advantage of the fine-grained control of the exposed FlexCore datapath by doing manual scheduling of the operations to be performed. The FlexCore compiler therefore supports inline N-ISA instruction in the MIPS assembly code. The inline N-ISA instruction is written as Register Transfer Notations (RTNs) that are subsequently translated to N-ISA instructions by the compiler.

B.3.2 Cycle-Accurate Simulator

The simulator is a cycle-accurate model of a FlexCore implementation. The simulator takes the generated N-ISA code of the compiler as input and executes the application until it exits. The simulator has support for profiling of executed instructions as well as gathering statistics on resource utilization of the different FlexCore datapath units.

B.3.3 HDL Generator

The HDL generator allows a user to specify what datapath units a particular FlexCore implementation consists of and how they are connected to the interconnect. The generator has support for specifying which ports in the interconnect that should be connected with each other; specifically which input ports that are connected to a specific output port.

The generated HDL code can be taken through a conventional ASIC backend flow, in order to get accurate estimates on area, delay, and power. The generated HDL code can also be used in conventional HDL simulators, together with the instruction code from the compiler, to simulate the behavior of a FlexCore. Such simulations are routinely used to confirm the results from the cycle-accurate simulator.

B.4 Interconnect Evaluation

Performance comparisons of the different architectures have been conducted by running the Fast Fourier Transform (FFT) benchmark from the Embedded Microprocessor Benchmark Consortium (EEMBC) [6] on each architecture using the FlexCore cycle-accurate simulator. The FFT benchmark is part of EEMBC's TeleBench suite and implements a decimation-in-time 256-point 16-bit FFT using a butterfly technique. To get estimates on delay, power, and area, each architecture has been generated using the FlexCore HDL generator and subsequently taken through synthesis and place-and-route.

B.4.1 FFT Application Scheduling

The FFT application was chosen for this study because of its reasonable size. This makes it possible to manually schedule a significant portion of the algorithm onto the investigated architectures, thus, allowing us to study and compare the performance gains for exactly the same software application.

The manual scheduling was done by compiling the C-code for the FFT application to MIPS assembly. The MIPS assembly was subsequently fed to the FlexCore compiler, which generated the final N-ISA instructions. The generated code was profiled with the help of the simulator and the FFT application's computational kernels were identified. The FFT application starts with a setup phase, in which data and twiddle factors are generated, before the actual FFT computation is carried out. The setup is done once, after which it is possible to execute multiple FFT computations.

The FFT computation has one clearly identifiable computational kernel, consisting of three nested loops. More than 73% of the total execution time is spent in the inner-most loop, when 100 FFT computations are carried out. This particular code segment has therefore been the target for manual scheduling. Not surprisingly, the code of the inner-most loop corresponds to the computations of a single FFT butterfly.

The manual scheduling was performed without any algorithmic changes to the FFT application. The MIPS assembly of the inner-most loop was simply exchanged with inline RTN code. In the scheduling procedure we took each

MIPS instruction and scheduled it as early as possible onto the given datapath. If there exist no resource conflicts with already scheduled instructions, we might schedule the instruction that we currently consider *earlier* than what the MIPS assembly would suggest. The instruction order might therefore have changed. Unnecessary utilization of resources is avoided by, for example, not writing back temporary values to the register file.

To fully take advantage of the exposed datapath for general applications, there needs to be support by the compiler for efficient scheduling of computations onto the datapath. This is addressed not only in the FlexSoC project, but also in other research projects, such as that by Reshadi *et al* [7], even though their compiler targets co-design.

B.4.2 Hardware Implementation

The FlexCore HDL generator has been used to generate the architectures by specifying the communication paths that are available in the different datapaths. For this particular study, the baseline FlexCore has been extended with a 32-bit multiplier, which is pipelined in two stages. The two inputs to the multiplier are directly connected to the interconnect. Between the output of the multiplier and the interconnect we have inserted two 32-bit registers, where one holds the 32 most significant bits and the other holds the 32 least significant bits of the result.

In neither of the three different architectures, control logic and instruction fetch power were accounted for. By focusing on the datapath, the impact of the interconnect can be separated from the intrinsic gains of exposing the datapath, to avoid observing differences in delay and power that are due to different ways of handling the control or requirements on memory bandwidth. What implications various types of control have on the overall performance is an issue which is being addressed within the FlexSoC project.

When we discard the control logic and instruction fetch circuitry of the GPP datapath in Figure B.1(a), this datapath is exactly the same as that of the exposed datapath. Figure B.3 shows a detailed schematic of the communication paths that are available to the GPP and the exposed datapath. The datapath is inspired by the DLX and MIPS R2000 datapaths [8]. The multiplier is an intrinsic part

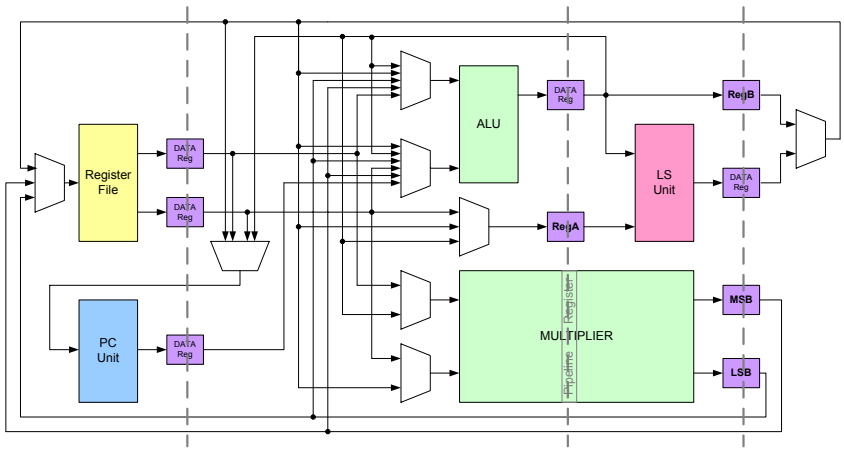


Figure B.3: *Schematic of the datapath for the GPP and exposed datapath. Only the communication paths for data are shown, without any control signals.*

of the datapath and takes its inputs from the register file. To improve performance, it is possible to forward results from the ALU and LS Unit directly. The output registers of the multiplier do not work as traditional pipeline registers, where a new value is clocked in for each new clock cycle. Instead these registers function such that they hold the result from the previous multiplication until a new multiplication is performed. This is to support the semantics of a GPP which uses move instructions to move data from the output registers to the register file.

The datapath which uses the flexible interconnect was generated by specifying the communication paths, such that an input port to a functional unit can be connected to any output port of any functional unit. This creates a fully connected crossbar switch as interconnect.

Figure B.4 illustrates how the crossbar switch is constructed out of a number of switchboxes. Each input to a datapath unit (in Figure B.3) as well as to the two registers RegA and RegB are connected to its own switchbox. Each switchbox is in turn connected to all the outputs of the datapath units. The switchbox functions as a multiplexer, according to Figure B.5.

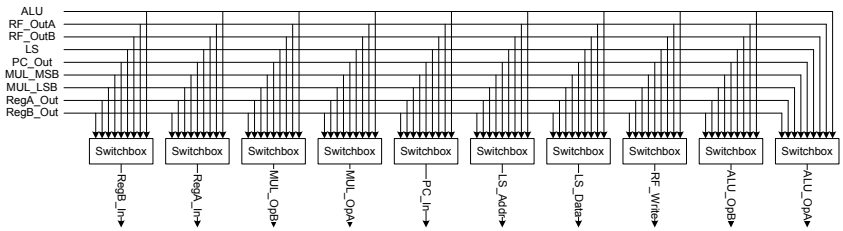


Figure B.4: *Illustration of the crossbar switch of the fully connected interconnect.*

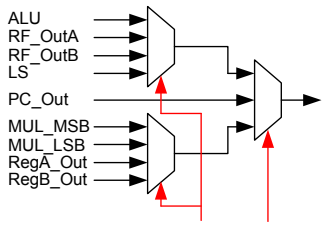


Figure B.5: *Illustration of a switchbox.*

The generated designs were taken through a synthesis and place-and-route flow [9, 10] for a commercial 0.13-μm technology. Delay and power estimates were obtained from resistance and capacitance extracted placed-and-routed designs.

B.5 Results

The three architectures were simulated with the FlexCore simulator running an application in the form of 100 consecutive FFT computations. Table B.1 summarizes the number of executed cycles for the whole application and the manually scheduled inner-most loop, respectively, for each architecture. Table B.1 shows that by deploying only the concept of exposed datapath, the inner-loop cycle count was reduced with as much as 30% compared to a traditional five-stage GPP. But by exploiting a fully connected interconnect, we show that the inner-loop cycle count can be improved with *a further 20%* compared to only utilizing the benefits of an exposed datapath.

We also observe that for the FFT application as a whole, the cycle count reductions are not as dramatic. This is because here almost 30% of the executed instructions have not been manually scheduled and therefore do not benefit from neither the exposed datapath nor the flexible interconnect.

Table B.1: *Simulated Application Cycle Count*

Datapath	Whole Application	Inner Loop
GPP	5 877 569 (100%)	4 300 800 (100%)
Exposed Datapath	4 750 769 (81%)	3 072 000 (71%)
Fully Connected	3 803 669 (65%)	2 150 400 (50%)

The delay, power, and area estimates of the different architectures are shown in Table B.2. Since the control has been excluded for all architectures, the estimates for the GPP and exposed datapath are the same. The fully connected interconnect increases the delay by 11% and the area by 15%. This is a reasonable overhead for the flexible interconnect, considering the application cycle count speedups. The power on the other hand is 23% higher than for the GPP and the exposed datapath, which indicates a high overhead for the interconnect.

Table B.2: *Delay, Power, and Area Estimates*

Datapath	Delay (ns)	Power (mW)	Area (mm^2)
GPP	2.25 (100%)	6.57 (100%)	0.35 (100%)
Exposed Datapath	2.25 (100%)	6.57 (100%)	0.35 (100%)
FlexCore	2.49 (111%)	8.07 (123%)	0.40 (115%)
Tailored	2.36 (105%)	6.83 (104%)	0.37 (106%)

Due to the high power dissipation for the fully connected interconnect, an interconnect was tailored for the FFT application. We designed this tailored interconnect by first simulating the architecture with a fully connected interconnect, using the FFT application and its manual scheduling. In a subsequent phase we extracted statistics of which communication paths that were utilized during the simulation. We used the statistics to restrict the communication paths

by pruning the interconnect in the FlexCore generator. In order to not lose any general-purpose processing efficiency, we made sure that none of the communication paths that exist in the GPP datapath were removed from the interconnect. Like for the other three architectures, the generated HDL code was taken through an ASIC backend flow.

With the tailored interconnect the power overhead is reduced from 23% to only 4% and the area overhead is reduced from 15% to 6% compared to the GPP. The delay for the tailored interconnect is also reduced from 11% to 5%.

Flexible interconnects appear to be at a slight disadvantage, in terms of delay and power dissipation. However, total application performance in terms of execution time (cycle count $\times$ clock period) and energy dissipation (execution time $\times$ power dissipation) should be considered next to delay and power. The FFT application runs 28% faster on the datapath that uses a fully connected interconnect, as compared to that of a traditional GPP. Thanks to its shorter clock period, the tailored interconnect performs even better, with a 32% reduction of the execution time (Table B.3).

Table B.3: *FFT Application Execution Time and Energy Dissipation*

Datapath	Execution Time (ms)	Energy Dissipation (μJ)
GPP	13.2 (100%)	86.7 (100%)
Exposed Datapath	10.7 (81%)	70.3 (81%)
Fully Connected	9.5 (72%)	76.7 (88%)
Tailored	9.0 (68%)	61.5 (71%)

When considering energy dissipation for executing the FFT application the architecture with the fully-connected, flexible interconnect performs 12% better than that of a traditional GPP, even though the flexible interconnect dissipates 23% more power. The tailored interconnect is not only the fastest solution but also the most energy efficient, with a 29% energy reduction compared to a conventional GPP.

B.6 Discussion

The results in the previous section clearly show that the rigid interconnect of a GPP datapath restricts the performance potential of an exposed datapath. On the other hand, a fully connected interconnect has high power dissipation. It is therefore interesting to further discuss if it is possible to create an interconnect that allows the full potential of exposed control to be exploited without leading to the somewhat high power dissipation of a fully connected interconnect.

The tailored interconnect for the FFT application shows that it is possible to reduce the number of communication paths without any degradation in execution-cycle performance. We will here give a short analysis of *i)* what paths helped to increase performance and *ii)* what paths in the fully connected interconnect that have not contributed to any performance improvement for the FFT application.

When we manually scheduled operations onto the exposed datapath with the rigid interconnect, some architectural shortcomings became evident. A major shortcoming is that when making a load or store to the memory the address has to come from the ALU, even though the address might already exist somewhere else in the pipeline. Similarly the data for the store instruction can only be read from RegA, Figure B.3. This leads to a waste of resources that could have been used for other operations.

In a more flexible interconnect, data and address to the load/store unit can be read from several datapath units. Further, with a flexible interconnect RegA and RegB can be used for temporary storage. Such a temporary storage is useful in order to distribute write requests to the register file over time, since writes to the register file can be delayed until the write port becomes available. It can also be used to eliminate writes to the register file, if the data exists for short time and is to be consumed by one of the datapath units. This is not possible in the GPP datapath, because RegA and RegB get contaminated by ALU and store operations.

The improvements to the interconnect that are listed above are generic, and their effect on the execution of the FFT application can be observed also for the tailored, but still flexible interconnect. Table B.4 shows the communication paths used by the FFT application, when it has been scheduled for a fully

connected interconnect. The table notation is as follows: A FFT in the matrix indicates that the communication path between the output port on the left hand side and the input port stated at the top is utilized by the benchmark. The paths marked by FFT, together with those marked with GPP, are those of the tailored interconnect. Table B.5 lists those communication paths that are not available in the traditional GPP datapath. Here we see that, in comparison to the GPP datapath, there exist more paths to the LS Unit, and to and from RegA and RegB. There are also more direct paths to and from the register file, which is a consequence of the fact that data for store instructions are not moved through RegA and that ALU results are not being moved through RegB.

By looking at what paths that can be pruned away from a fully connected interconnect, we directly realize that some of the communication paths contribute very little to increase the performance. For example, there is little need for a communication path from the LS Unit's output to its *data* input. The only time this path would be used is if data is to be moved from one memory location to another[1]. If this operation would be necessary it is still possible to use, for example, RegA to temporarily store the data, and then forward it to the LS data input port. There are other paths we can prune away; we can restrict the connections between datapath units with several output ports and datapath units with several input ports. An example of this would be to connect the register file with the ALU in the same way as for the GPP datapath in Figure B.3. Here only one output port of the register file is connected to each input of the ALU. All the listed path restrictions to a fully connected interconnect have no or negligible impact on the execution-cycle performance of an application.

Considering the FFT application we see that the output of the LS Unit is never used by the LS Unit itself. The possibility of only connecting one of the output ports of a datapath unit to one of the input ports of another datapath unit that has several input and output ports has also been exploited. Further, we see that the PC output port, which is used for immediate values and for storing the program counter on a jump-and-link instruction, is only connected to the ALU. In a more general application, a communication path to the register file would be suitable for easy storage of the program counter.

[1] Movement of larger amounts of data would be best served by a DMA unit, if available.

Table B.4: *Interconnect Paths Used by the FFT Application and the Tailored Interconnect*

	PC In	RF Write	ALU OpA	ALU OpB	LS Address	LS Data	RegA In	RegB In	MULT OpA	MULT OpB
PC Out				FFT						
RF ReadA	FFT	FFT	FFT						FFT	
RF ReadB	FFT			FFT	FFT	FFT	FFT	FFT		FFT
ALU Out	FFT	FFT	FFT	FFT	FFT	FFT	FFT	FFT	FFT	
LS Out	FFT	FFT	FFT	FFT			FFT		FFT	FFT
RegA Out			FFT	FFT	FFT	FFT				FFT
RegB Out	FFT	FFT	FFT	FFT	FFT		FFT			
MULT LSB		FFT	FFT	GPP			FFT			
MULT MSB		GPP	GPP	GPP						

Table B.5: *Non-GPP Interconnect Paths in the Tailored Interconnect*

From	To
Register File	Register File
Register File	LS Address
Register File	LS Data
Register File	RegB
ALU	Register File
ALU	LS Data
RegA	ALU
RegA	LS Address
RegA	MULT
RegB	LS Address
Multiplier	RegA

The tailored interconnect for the FFT application addresses the major communication-path drawbacks of the rigid GPP interconnect. It should be noted that it does not limit the communication paths in a way that would hamper the cycle-count performance for other applications. Minor improvements could be made to the tailored interconnect, such as adding a communication path between the PC's output port and the register file.

To evaluate the generality of the tailored interconnect, we manually scheduled parts of the Autocorrelation benchmark (here called AutoBench) from EEMBC's TeleBench suit. The execution cycle-time improvements for the exposed datapath and the datapath with a fully connected interconnect are similar to those for the FFT application, see Table B.6. The exposed datapath handles load and store instructions poorly, since the ALU is always utilized for these operations. This can be clearly observed by the low performance improvements for the exposed datapath. A flexible interconnect allows the ALU to be used for useful operations in parallel with load instructions in AutoBench. This can be noted by the significant performance improvements for the fully connected datapath.

Table B.6: *Cycle Count, Execution Time and Energy Estimations for AutoBench*

Datapath	Cycle Count	Exec. Time (μs)	Energy (μJ)
GPP	168k (100%)	378 (100%)	2.5 (100%)
Exposed Datapath	150k (89%)	338 (89%)	2.2 (88%)
Fully Connected	118k (70%)	294 (78%)	2.4 (96%)
Tailored	118k (70%)	278 (74%)	1.9 (76%)

Table B.7 shows the interconnect paths used by AutoBench that are not part of a traditional GPP. The number of added paths are not as many as for the FFT benchmark, but what is important is that the added paths are *a subset of the interconnect that was tailored for the FFT application*. This allows AutoBench to be as efficiently scheduled on the tailored interconnect as on the fully connected interconnect.

Table B.7: *Non-GPP Interconnect Paths used by Autocorrelation*

From	To
Register File	Register File
Register File	LS Address
ALU	Register File
RegA	MULT

The tailored interconnect exhibits significant execution time and energy improvements (Table B.3 and B.6) and has shown that it can cater for the needs of the FFT and Autocorrelation benchmarks. We believe that these improvements are likely to apply also for other embedded applications, as argued for in this section.

B.7 Conclusion

The introduction of a tailored interconnect to an exposed GPP datapath has been shown to reduce the total execution time for an FFT and an autocorrelation ap-

plication with 26-32%. Further, the tailored interconnect also reduce the total energy dissipation for the two applications with 24-29%. The tailored interconnect has been evaluated with only two applications, but the construction of the interconnect has been done such that it does not hamper the performance of general-purpose processing. In fact, the tailored interconnect should give an improved performance not only for dedicated applications, but also for most general-purpose applications.

B.8 Acknowledgments

We thank our FlexSoC colleagues (Hughes, Jeppson, Sheeran, Svensson, Stenstrom, and Thuresson) whose work has made it possible to make this study. A special thanks goes to Martin Thuresson for his fruitful comments on the structure of this article and its illustrations. The FlexSoC project is sponsored by the Swedish Foundation for Strategic Research.

Bibliography

[1] Bita Gorjiara and Daniel Gajski, "Custom Processor Design Using NISC: a Case-Study on DCT Algorithm," in *Workshop on Embedded Systems for Real-Time Multimedia*, September 2005, pp. 55–60.

[2] Bita Gorjiara, Mehrdad Reshadi, and Daniel Gajski, "Designing a Custom Architecture for DCT Using NISC Design Flow," in *Asia and South Pacific Conference on Design Automation*, 2006, pp. 116–117.

[3] John Hughes, Kjell Jeppson, Per Larsson-Edefors, Mary Sheeran, Per Stenström, and Lars "J." Svensson, "FlexSoC: Combining Flexibility and Efficiency in SoC Designs," in *Proceedings of the IEEE NorChip Conference*, 2003.

[4] Magnus Björk, Magnus Själander, Lars Svensson, Martin Thuresson, John Hughes, Kjell Jeppson, Jonas Karlsson, Per Larsson-Edefors, Mary Sheeran, and Per Stenstrom, "Exposed Datapath for Efficient Computing," Tech. Rep. 2006–20, Department of Computer Science and Engineering, Chalmers University of Technology, December 2006.

[5] Magnus Björk, Magnus Själander, Lars Svensson, Martin Thuresson, John Hughes, Kjell Jeppson, Jonas Karlsson, Per Larsson-Edefors, Mary Sheeran, and Per Stenstrom, "Exposed Datapath for Efficient Computing," in *HiPEAC Workshop on Reconfigurable Computing*, January 2007.

[6] "Embedded Microprocessor Benchmark Consortium," `http://www.eembc.org`.

[7] M. Reshadi, B. Gorjiara, and D. Gajski, "Utilizing Horizontal and Vertical Parallelism with No-Instruction-Set Compiler for Custom Datapaths," in *International Conference on Computer Design (ICCD)*, October 2005.

[8] David A. Patterson and John L. Hennessy, *Computer Organization & Design, The Hardware/Software Interface*, Morgan Kaufman Publishers Inc., 2nd edition, 1998.

[9] *Synopsys Design Compiler User Guide Version W-2004.12*.

[10] *Cadence Encounter User Guide Version 4.1*.

PAPER C

M. Själander and P. Larsson-Edefors

The Case for HPM-Based Baugh-Wooley Multipliers[†]

Department of Computer Science and Engineering,
Chalmers University of Technology, Technical Report 2008-8
Göteborg, Sweden, March 4, 2008.

[†] A four page version of this article has been published as M. Själander and P. Larsson-Edefors "High-Speed and Low-Power Multipliers Using the Baugh-Wooley Algorithm and HPM Reduction Tree" in *IEEE International Conference on Electronics, Circuits, and Systems* St. Julian's, Malta, August 31st - September 3rd, 2008.

C

The Case for HPM-Based Baugh-Wooley Multipliers

The modified-Booth algorithm is extensively used for high-speed multiplier circuits. Once, when array multipliers were used, the reduced number of generated partial products significantly improved multiplier performance. In designs based on reduction trees with logarithmic logic depth, however, the reduced number of partial products has a limited impact on overall performance. The Baugh-Wooley algorithm is a different scheme for signed multiplication, but is not so widely adopted because it may be complicated to deploy on irregular reduction trees. We use the Baugh-Wooley algorithm in our High Performance Multiplier (HPM) tree, which combines a regular layout with a logarithmic logic depth. We show for a range of operator bit-widths that, when implemented in 130-nm and 65-nm process technologies, the Baugh-Wooley multipliers exhibit comparable delay, less power dissipation and smaller area foot-print than modified-Booth multipliers.

C.1 Introduction

Multiplication is an important arithmetic operation and multiplier implementations date several decades back in time. Multiplications were originally performed by iteratively utilizing the ALU's adder. As timing constraints became stricter with increasing clock rates, dedicated multiplier hardware implementations such as the array multiplier were introduced. Since then ever more sophisticated methods on how to implement multiplications have been proposed. One of the more popular implementations is that of the modified-Booth recoding scheme together with a logarithmic-depth reduction tree and a fast final adder. Modified-Booth recoding has the advantage of reducing the number of generated partial products by half, compared to partial-product generation based on 2-input AND-gates. This fact decreases the size of the reduction circuitry, which commonly is a logarithmic-depth reduction tree, e.g. Wallace, Dadda or TDM. Since such reduction trees are infamous for their irregular structures, which make them difficult to place and route during the physical layout of a multiplier, a decreased size of the reduction circuit eases the implementation and improves the performance of the multiplier.

Modified-Booth implementation strategies are commonly motivated by the need for fast multipliers. However, with the ongoing integration trend for which power dissipation is an ever pressing concern, modified Booth is no longer the obvious implementation choice. Already in 1997 Callaway *et al.* showed that, for a 2 μm process technology, a Wallace multiplier is more energy efficient than a modified-Booth multiplier [1]. Even though power has become a bigger concern since then, the modified-Booth multiplier is still prevailing as the main implementation choice for high-speed multipliers.

In this paper we compare a multiplier based on the modified-Booth algorithm against one based on the less used Baugh-Wooley algorithm. As will be shown in Sec. C.8, a Baugh-Wooley multiplier implemented in a 130-nm and 65-nm process technology is more power and energy efficient than a modified-Booth multiplier of equal bit-width. This efficiency comes with only an insignificant increase in delay. We will subsequently show that the high power dissipation makes modified Booth a poor implementation choice for high-speed multipliers.

C.2 Modified-Booth Multiplication

The original Booth algorithm [2] for the multiplication $s = x \times y$ recodes partial products by considering two bits at a time of one of the operands (x) and encoding them into $\{-2, -1, 0, 1, 2\}$. Each such encoded higher-radix number is subsequently multiplied with the second operand (y), yielding one row of recoded partial-product bits. The advantage of Booth recoding is that the number of recoded partial products is fewer than the number of un-recoded partial products, and this can be translated into higher performance in the circuitry, which reduces all partial-product bits into the final product.

The drawback of the original Booth algorithm is that the number of recoded partial products depends on input operand x, which makes this algorithm unsuitable for implementation in hardware. The modified-Booth (MB) algorithm [3] by MacSorley remedies this by looking at three bits at a time of operand x. Then we are guaranteed that only half the number of partial products will be generated, compared to a conventional partial-product generation using 2-input AND gates. Since it has a fixed number of partial products, the MB algorithm is suitable for hardware implementation.

An MB multiplier works internally with a two's complement representation of the partial products, in order to be able to multiply the encoded $\{-2, -1\}$ with the y operand. To avoid having to sign extend the partial products, we are using the scheme presented by Fadavi-Ardekani [4]. In the two's complement representation, a change of sign includes the insertion of a '1' at the least significant bit (LSB) position (henceforth called LSB insertion). To avoid an irregular implementation of the partial-product reduction circuitry, we draw on the idea called modified partial-product array [5]. Here, the impact of LSB insertion on the *two* least significant bits positions of the partial product is pre-computed. The pre-computation redefines the LSB of the partial product (Eq. C.1 in [5]) and moves the potential '1', which results from the LSB insertion, to the second least significant position (Eq. C.2). Note that our Eq. C.2 is different from the corresponding equation used in [5]. Fig. C.1 illustrates an 8-bit MB multiplication using sign extension prevention and the modified partial-product array scheme.

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

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

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

Figure C.1: *Illustration of an 8-bit modified-Booth multiplication.*

C.2.1 Modified-Booth Implementation

The implementation of the MB multiplier utilizes the regular reduction tree of the High Performance Multiplier (HPM) [6]. In Fig. C.2 the reduction tree of an 8-bit MB implementation is shown. The long arrows with '1"s and a_i's show the places where these signals, which are in the lower row of the partial-product rhomboid in Fig. C.1, have been inserted into the reduction tree. The figure does, however, not show the MB recoding circuits.

The MB recoding logic can be designed in many different ways. For the subsequent comparison we have chosen to implement two different, recent recoding schemes. The first recoding scheme is the one presented by Yeh *et al.* [5], since this is claimed to be faster than many previous schemes. The circuits for encoding and decoding using this scheme are shown in Fig. C.3.

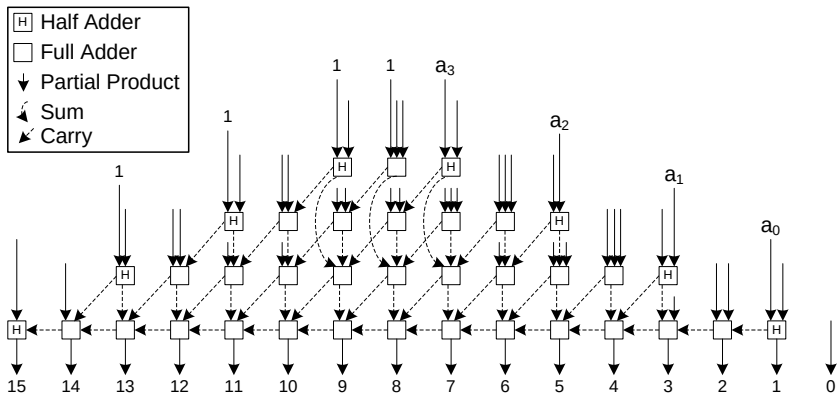
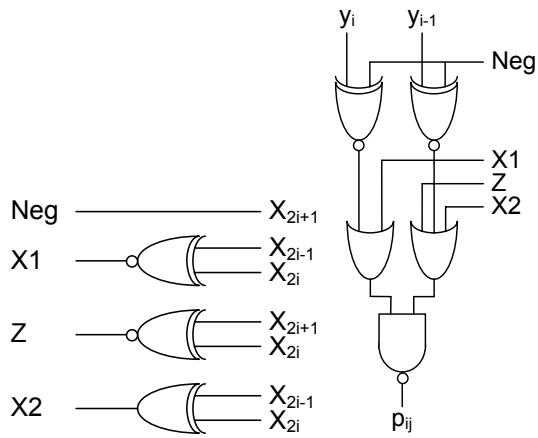


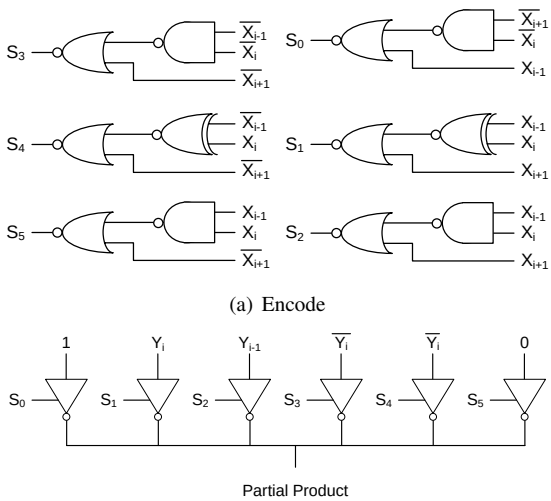
Figure C.2: *8-bit modified-Booth multiplier using an HPM reduction tree. For simplicity, the modified-Booth recoding logic is not shown. Furthermore, a simple ripple-carry adder is used as final adder.*



(a) Encode (b) Decode

Figure C.3: *Modified-Booth recoding circuits according to Yeh* et al..

We have also chosen to implement the recoding scheme of Hsu *et al.* [7]. We chose to implement this recoding scheme because it is a recent one, showing promising results. The encoding and decoding circuits are shown in Fig. C.4.



(a) Encode

(b) Decode

Figure C.4: *Modified-Booth recoding circuits according to Hsu* et al..

C.3 Baugh-Wooley Multiplication

The Baugh-Wooley (BW) algorithm [8] is a relatively straightforward way of doing signed multiplications; Fig. C.5 illustrates the algorithm for an 8-bit case, where the partial-product bits have been reorganized according to Hatamian's scheme [9]. The creation of the reorganized partial-product array comprises three steps: *i)* The most significant bit (MSB) of the first $N-1$ partial-product rows and all bits of the last partial-product row, except its MSB, are inverted. *ii)* A '1' is added to the Nth column. *iii)* The MSB of the final result is inverted.

C.3.1 Baugh-Wooley Implementation

To the best of our knowledge, performance investigations in the open literature concerning BW implementations have exclusively been based on reduction arrays, in the spirit of the original paper [8]. In this investigation, however, we have chosen to implement the BW algorithm on the HPM reduction tree [6] instead.

							y_7	y_6	y_5	y_4	y_3	y_2	y_1	y_0
							x_7	x_6	x_5	x_4	x_3	x_2	x_1	x_0
						1	$\overline{p_{70}}$	p_{60}	p_{50}	p_{40}	p_{30}	p_{20}	p_{10}	p_{00}
						$\overline{p_{71}}$	p_{61}	p_{51}	p_{41}	p_{31}	p_{21}	p_{11}	p_{01}	
					$\overline{p_{72}}$	p_{62}	p_{52}	p_{42}	p_{32}	p_{22}	p_{12}	p_{02}		
				$\overline{p_{73}}$	p_{63}	p_{53}	p_{43}	p_{33}	p_{23}	p_{13}	p_{03}			
			$\overline{p_{74}}$	p_{64}	p_{54}	p_{44}	p_{34}	p_{24}	p_{14}	p_{04}				
		$\overline{p_{75}}$	p_{65}	p_{55}	p_{45}	p_{35}	p_{25}	p_{15}	p_{05}					
	$\overline{p_{76}}$	p_{66}	p_{56}	p_{46}	p_{36}	p_{26}	p_{16}	p_{06}						
p_{77}	$\overline{p_{67}}$	$\overline{p_{57}}$	$\overline{p_{47}}$	$\overline{p_{37}}$	$\overline{p_{27}}$	$\overline{p_{17}}$	$\overline{p_{07}}$							
$\overline{s_{15}}$	s_{14}	s_{13}	s_{12}	s_{11}	s_{10}	s_9	s_8	s_7	s_6	s_5	s_4	s_3	s_2	s_1	s_0

Figure C.5: *Illustration of an 8-bit Baugh-Wooley multiplication.*

Implementing the BW multiplier based on the HPM tree is as straightforward as the basic algorithm itself. The partial-product bits can be generated by using a 2-input AND gate for each pair of operand bits. In the case a partial-product bit should be inverted, we employ a 2-input NAND gate instead. The insertion of '1' in column N is easily accommodated by changing the half adder at top of row N to a full adder with one of the input signals connected to '1'[1]. Finally, the inversion of the MSB of the result is done by adding an inverter. The final result of the implementation of the BW algorithm is depicted in Fig. C.6.

C.4 An Initial Gate-Level Study

A widespread belief is that the number of rows of partial-products bits has large impact on the total time to carry out a multiplication. For sequential multipliers, where for each cycle one of the operands is added to a partial result, the number of partial products to be added is indeed critical. Here, the reduction in rows resulting from Booth encoding has a direct and significant effect on the cycle count for a complete multiplication.

[1]The full-adder circuit can consequently be simplified, since one of its input signals is statically defined.

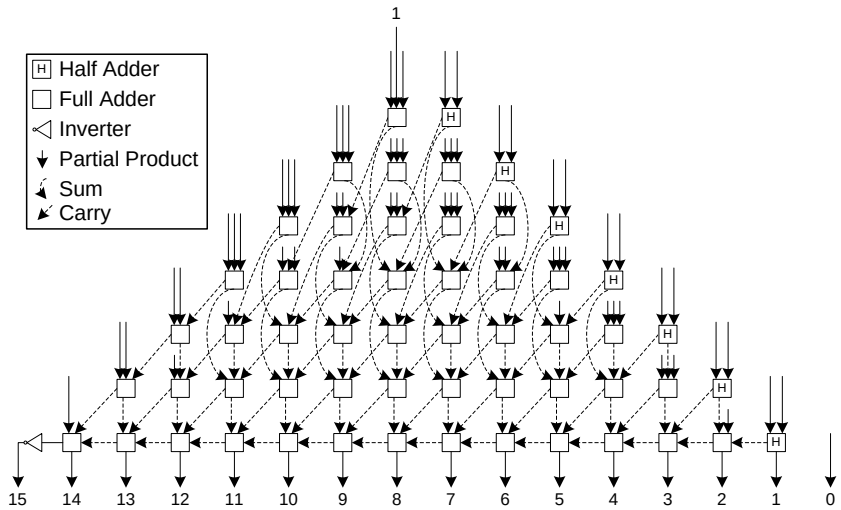


Figure C.6: *8-bit Baugh-Wooley multiplier using an HPM reduction tree. For simplicity, the* AND/NAND *gates for partial-product generation are not shown. Furthermore, a simple ripple-carry adder is used as final adder.*

At the time when the modified-Booth (MB) and Baugh-Wooley (BW) algorithms were introduced (in 1961 and 1973, respectively) hardware multipliers were at best built as an array of full adders. In an array multiplier, each row of partial-product bits adds one full adder to the logic depth of the multiplier[2]; in this context the reduction in partial-product rows resulting from MB has great effect on timing.

However, in dedicated high-performance multipliers that are built around a reduction tree, the number of rows becomes less critical, because of the logarithmic nature of the gate structure of the tree. With today's logarithmic-depth reduction trees, such as TDM, Wallace, Dadda, and HPM, the logic depth increases with the logarithm of the maximum number of adders in a column. The logic depth for the HPM is shown in Table C.1. The logic depth is here defined as the maximum number of adders that have to be traversed to reach the final

[2]With the exception of the first two rows of partial products, which only result in one row of full adders.

adder in an HPM tree built out of 3:2 adders (full adders). To be able to evaluate the logic depth of the reduction tree for BW and MB, the logic depth is given for the maximum number of adders in a column, instead of for the operand width of the multiplier.

Table C.1: *Logic depth through the HPM tree, with respect to the maximum number of adders in a column.*

Maximum number of adders (A)	Logic depth
$A = 1$	1
$A = 2$	2
$3 \leq A \leq 4$	3
$5 \leq A \leq 7$	4
$8 \leq A \leq 11$	5
$12 \leq A \leq 17$	6
$18 \leq A \leq 26$	7
$27 \leq A \leq 40$	8
$41 \leq A \leq 61$	9
$62 \leq A \leq 92$	10

The maximum number of adders in a column of the HPM reduction tree, for a certain operand width of BW and MB, can be obtained through the simple relation in Table C.2.

Table C.2: *Maximum number of adders in a column of the HPM tree based on the partial-product generation according to Baugh-Wooley and modified Booth.*

Operand width	Maximum number of adders	
	Baugh-Wooley	Modified Booth
N	$N - 2$	$N/2 - 1$

By combining the information on the logic depth through the HPM reduction tree, Table C.1, and the relation between the maximum number of adders and the operand width, Table C.2, we can obtain the logic depth for different operand widths of the BW and MB implementations. The results are given in Table C.3.

Table C.3: *Logic depth through the HPM reduction tree based on the partial-product generation according to Baugh-Wooley and modified Booth.*

Operand width	Logic depth		Difference in
	Baugh-Wooley	Modified Booth	logic depth
8	4	3	1
16	6	4	2
32	8	6	2
40	8	7	1
48	9	7	2
54	9	7	2
60	9	8	1
64	10	8	2

As Table C.3 shows, the logic depth of the reduction tree designed for modified Booth is at best two full-adder delays shorter than that designed for Baugh-Wooley, for operand widths up to at least 64 bits. The shorter logic depth through the MB reduction tree comes at the cost of a partial-product generation circuit that is more complex than that of BW, whose partial-product generation circuit utilizes only a simple 2-input AND-gate for each generated partial-product bit.

The conclusion of this initial gate-level study is that it is certainly not obvious that the deployment of the modified-Booth algorithm onto a fast reduction tree yields a faster implementation than a corresponding Baugh-Wooley implementation.

C.5 Multiplier Evaluation Setup

To evaluate the different multiplier designs a multiplier generator [10] was created. This generator is capable of generating gate-level VHDL netlist descriptions of modified-Booth (MB) and Baugh-Wooley (BW) multipliers of various operand sizes, according to the schemes presented in the previous sections. Both the MB recoding scheme of Yeh *et al.* and Hsu *et al.* have been implemented. All netlists are based on the regular reduction tree of a High Perfor-

mance Multiplier (HPM) [6], based on 3:2 adder (full adder) cells. A Kogge-Stone [11] adder is used as the final adder, placed on the reduction tree outputs.

By exhaustively simulating all input patterns and verifying the result using Cadence NC-VHDL [12], the VHDL generator was verified to generate functionally correct multiplier netlists of sizes up to at least 16 bits. For multipliers larger than 16 bits, the functionality of each multiplier size was verified for a random pattern of one million vectors.

The VHDL descriptions were synthesized using Synopsys Design Compiler [13] together with a commercially available 1.2 V 130-nm process technology. The synthesized netlist was taken through place and route using Cadence Encounter [14]. To create a common input and output interface to the multipliers and to enable higher place and route quality from the EDA tools, which do not handle purely combinatorial circuits well, registers were placed on the inputs and outputs of the multipliers. Delay estimates were obtained after RC extraction from the placed and routed netlists. Power estimates were derived from the same netlists, using value change dump (VCD) data from simulations of 10,000 random input vectors. All delay, power, and area figures that we present, include the input and output registers on the multipliers and are for the worst case corner at 1.08V.

Initially, the timing constraints were systematically set so as to push the limits of the timing that can be achieved for each generated VHDL description. This guarantees that a very fast implementation is obtained, however, the power dissipation becomes prohibitively high, due to excessive buffering and resizing of gates. Therefore, for each implementation, the timing was subsequently relaxed with 100, 300, and 600 ps, respectively, relative to the fastest timing obtained. Correspondingly, power estimates were obtained for each timing constraint.

C.6 Modified-Booth Multiplier Evaluation

Before we can compare a modified-Booth (MB) multiplier with a Baugh-Wooley (BW) multiplier, we first have to ascertain which of the previous two MB recoding schemes is best. The first scheme was that of Yeh *et al.* [5], as shown

in Fig. C.3, while the second scheme was that of Hsu *et al.* [7], as shown in Fig. C.4. The original circuit of Hsu uses transmission gates in the decoding circuit. However, with the cell library used for this evaluation we are limited to tristated inverters, which may hamper the efficiency of this scheme.

C.6.1 Yeh Recoding

If we consider the fanout for the recoding circuits of the MB multiplier using the Yeh recoding scheme, we notice that each decoder needs to drive N decoders, for a multiplier of size N. To find a trade off between the fanout of the x-inputs that drive the encoders and the fanout of the encoder output signals, we instantiate more than one encoder for the N decoders associated with one partial-product bit row. To reduce the load of the x-inputs, we use minimum-size inverters to buffer all three inputs of the encode circuit, as shown in Fig. C.7.

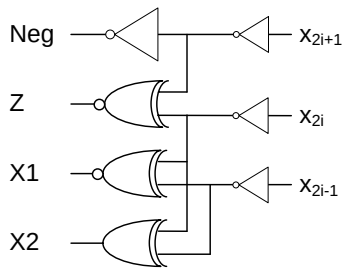


Figure C.7: *Buffered encode circuit for fanout reduction.*

To find a good trade off between the fanout of the x-inputs and the outputs of the encoding circuit, we varied the maximum number of decode circuits that are connected to a single encode circuit. The netlists of 32- and 64-bit multipliers with a maximum of 12, 16, or 20 decoders per encoder, were taken through synthesis, placement, and route. The delay and energy-per-operation can be found in Table C.4. As can be seen from the table, the minimum delay that can be achieved differs only with 30 ps (less than 1%) for the 64-bit case. However, the difference in energy-per-operation is more than 4 pJ (3%). Taking energy

into consideration, a maximum fanout of 16 decoders per encoder appears to offer a good trade off between delay and energy.

Table C.4: *Delay and energy dependency on fanout for the MB multiplier using a Yeh recoder.*

	Maximum Number of Decoders per Encoder		
Width	12	16	20
16	2.82 (ns) 9.24 (pJ)	2.79 (ns) 9.04 (pJ)	2.79 (ns) 9.04 (pJ)
32	3.69 (ns) 32.5 (pJ)	3.68 (ns) 32.6 (pJ)	3.65 (ns) 31.1 (pJ)
64	4.51 (ns) 135 (pJ)	4.52 (ns) 133 (pJ)	4.54 (ns) 137 (pJ)

C.6.2 Hsu Recoding

When we conduct the fanout investigation for the Hsu recoding scheme, we discover that the fanout for the x-inputs is reduced by adding minimum-sized inverters to all inputs of the encode circuits. The encode circuit is changed accordingly so that the logical function is preserved. To reduce the fanout of the encode circuits, the maximum number of decoders connected to a single encoder is limited, and to reduce the fanout of the y-inputs, we insert a number of inverters in parallel to buffer each input, so that each inverter drives a limited number of decoders.

The netlists of 32- and 64-bit multipliers with various fanout configurations, were taken through synthesis, placement, and route. The result for the 32-bit and the 64-bit multiplier implementations is given in Table C.5 and Table C.6, respectively. The choice of fanout configuration is not as straightforward as for the Yeh recoding scheme. For the 32-bit implementations we see that the configuration with 12 decoders per encoder and 10 decoders per inverter for the y-input yields the fastest implementation. However, this recoding configuration dissipates 9% more energy-per-operation than the implementation with the lowest energy-per-operation. Now, since the configuration with the lowest energy-per-operation also happens to be the second fastest, this is the most efficient configuration of all. When we consider also the 64-bit implementations, it is confirmed that 16 decoders per encoder and 10 decoders per inverter for the y-input is a good choice for configuration.

Table C.5: *Delay and energy dependency on fanout for 32-bit modified-Booth multipliers using a Hsu recoder. A row shows the maximum number of decoders per inverter for the y-inputs.*

	Maximum Number of Decoders per Encoder		
	12	16	20
6	3.80 (ns) 39.1 (pJ)	3.84 (ns) 38.9 (pJ)	3.83 (ns) 39.3 (pJ)
10	3.84 (ns) 39.8 (pJ)	3.79 (ns) 37.1 (pJ)	3.95 (ns) 38.9 (pJ)
14	3.70 (ns) 40.4 (pJ)	3.80 (ns) 38.5 (pJ)	3.83 (ns) 38.9 (pJ)

Table C.6: *Delay and energy dependency on fanout for 64-bit modified-Booth multipliers using a Hsu recoder. A row shows the maximum number of decoders per inverter for the y-inputs.*

	Maximum Number of Decoders per Encoder		
	12	16	20
6	4.84 (ns) 166 (pJ)	4.94 (ns) 156 (pJ)	4.83 (ns) 152 (pJ)
10	4.79 (ns) 166 (pJ)	4.80 (ns) 153 (pJ)	5.21 (ns) 158 (pJ)
14	4.70 (ns) 175 (pJ)	4.92 (ns) 163 (pJ)	4.85 (ns) 168 (pJ)

C.6.3 Recoding Scheme Comparison

Already from the results of Tables C.4, C.5, and C.6 it is clear that the Yeh recoding scheme outperforms the recoding scheme of Hsu. This is confirmed by Fig. C.8 that shows the delay and power for 16-, 32-, 48-, and 64-bit multipliers with the two different recoding schemes. Fig. C.8 shows the power dissipation for different delay constraints. The leftmost point for each graph in the figure depicts the shortest possible delay that is achieved using our evaluation setup. The second, third and fourth point is when the timing constraint is relaxed with 100 ps, 300 ps, and 600 ps, respectively, relative the shortest possible delay. From the figure it is clear that the Yeh recoding scheme is the best scheme both in terms of delay and power.

We believe that the Hsu recoding scheme somewhat suffers from the fact that our investigation is based on tristated inverters, rather than transmission gates. At the same time we doubt that the improvement gained from using

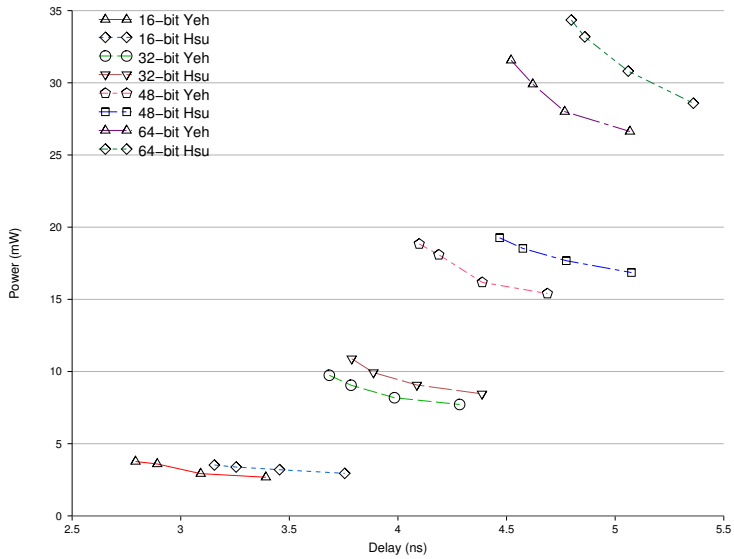


Figure C.8: *Delay and power for modified-Booth multipliers of various sizes, using two different recoding schemes.*

transmission-gate cells would be so significant that the Hsu recoding scheme would outperform the Yeh scheme.

C.7 Baugh-Wooley Multiplier Evaluation

The partial-product generation in the Baugh-Wooley (BW) implementation is much simpler than the one used in the modified-Booth multipliers. This difference in complexity becomes apparent also when we deal with fanout. In the BW multiplier each input drives N 2-input AND-gates for an N-bit multiplier. The effort to drive the AND-gates can simply be shared between a number of inverters that connect to the multiplier primary inputs. Table C.7 shows the result of varying the maximum number of AND-gates connected to an inverter from 6 to 14 gates. The table shows that connecting 10 AND-gates to each inverter yields both a fast and energy-efficient solution.

Table C.7: *Delay (ns) and energy (pJ) dependency on fanout for the Baugh-Wooley multiplier.*

	Maximum Number of Decoders per Inverter		
Width	6	10	14
16	2.91 (ns) 6.47 (pJ)	2.94 (ns) 6.27 (pJ)	2.94 (ns) 6.07 (pJ)
32	3.75 (ns) 24.5 (pJ)	3.63 (ns) 25.1 (pJ)	3.64 (ns) 23.7 (pJ)
64	4.65 (ns) 95.1 (pJ)	4.53 (ns) 95.4 (pJ)	4.68 (ns) 94.5 (pJ)

C.8 Comparison of Baugh-Wooley and Modified-Booth Multipliers

We begin the multiplier comparison of Baugh-Wooley (BW) and modified-Booth (MB), using the Yeh recoding scheme, by considering their efficiency in terms of delay and power[3]. Fig. C.9 shows the delay and power for 16-, 32-, 48-, and 64-bit multipliers, using *i)* a timing constraint that achieves the fastest implementation and *ii)* three different relaxed timing constraints: 100 ps, 300 ps, and 600 ps, respectively, slower than the fastest timing obtained. The figure shows that the MB implementation can be up to 150 ps faster than the BW implementation. However, for 32 bits the BW implementation outperforms that of the MB. In contrast to the timing, the power dissipation is consistently and significantly lower for the BW implementations. The power dissipation of the MB implementations is 25–40% higher than that of a BW multiplier of the same size, which operates at the same speed.

A useful metric when considering both delay and power is the energy that is required to complete a single multiplication operation. Since we are now dealing with single-cycle multipliers, the energy-per-operation is obtained as the product of power and delay for each point in the graph of Fig. C.9. Fig. C.10 shows the result of such a calculation for sizes from 8 to 64 bits: Among the four timing constraints, only the smallest energy-per-operation obtained for each size is plotted. The lowest energy-per-operation is commonly achieved at a timing that is 100 ps to 300 ps slower than the fastest possible. From the graph

[3]The power is given for a frequency that corresponds to the inverse of the delay.

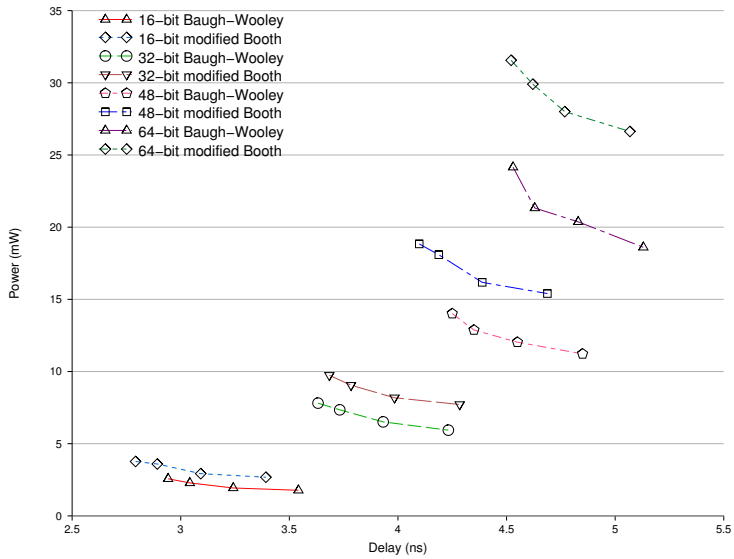


Figure C.9: *Delay and corresponding power dissipation for 16-, 32-, 48-, and 64-bit instances of modified-Booth and Baugh-Wooley multipliers.*

it is clear that the BW multiplier is a much more efficient implementation in terms of energy.

For some embedded systems, speed can be as crucial as power dissipation. To compare the shortest delay that can be achieved, Fig. C.11 shows the result for sizes of 8 to 64 bits. The graph shows that the BW multiplier can match the performance of the MB multiplier for sizes of up to 44 bits and even for larger multipliers the difference is no more than 150 ps, i.e. less than 4%.

If we consider area, which is an important factor for efficient multiplier implementations, the BW multiplier also in this respect outperforms the MB multiplier. Fig. C.12 shows the area of the different multiplier implementations and for all sizes the BW implementation consistently is the smallest. The BW implementation is about 20% smaller than that of a MB implementation of the same operand bit-width.

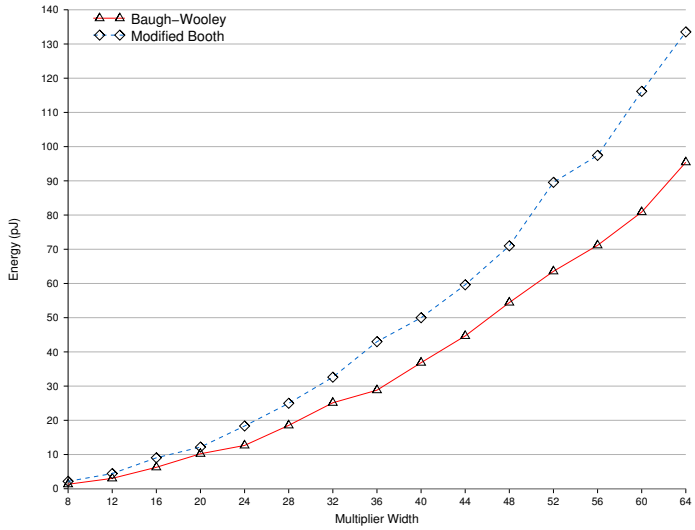


Figure C.10: *Energy-per-operation for modified-Booth and Baugh-Wooley multipliers.*

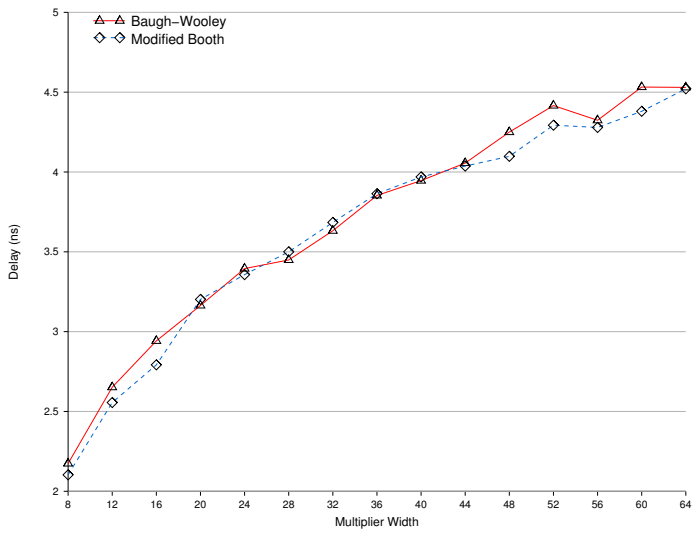


Figure C.11: *The shortest delay for modified-Booth and Baugh-Wooley multipliers.*

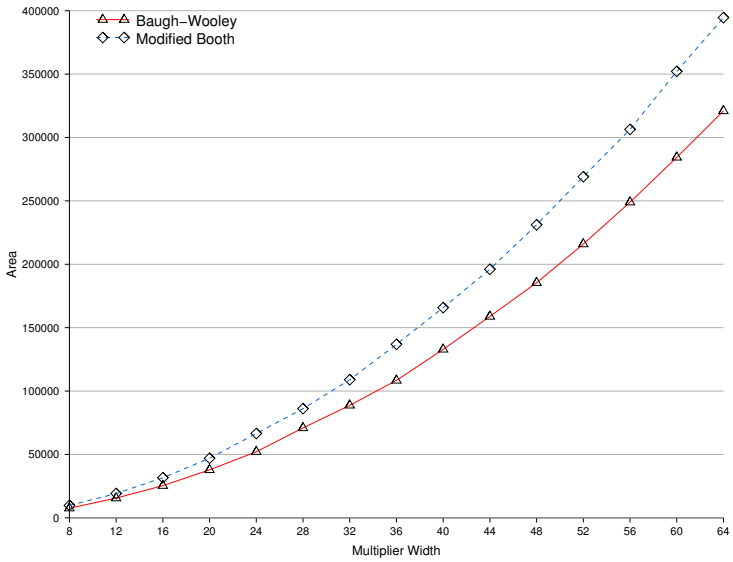


Figure C.12: *The area (μm^2) for modified-Booth and Baugh-Wooley multipliers.*

C.8.1 Dissecting the Timing

When we take a closer look at the timing contribution of *i)* the partial-product generation, *ii)* the reduction tree, and *iii)* the final adder to the total delay for MB and BW multipliers, we see that the delay that is gained by having a smaller reduction tree for the MB implementation is offset by the delay of the MB recoding circuitry. Tables C.8 and C.9 show that the partial-product generation of the MB multipliers are about two times as slow as that of generation in the BW multipliers. This comes as no surprise, since for BW there is only one 2-input AND-gate in the critical path, while for the MB multiplier the critical path goes through the encoder followed by the decoder. Taking fanout into consideration, the primary inputs of a BW multiplier need to drive only N minimum-sized 2-input AND-gates, while for the MB multiplier the x primary inputs first have to drive $N/2$ encoders, which in turn drive N decoders.

Table C.8: *The delay for the partial-product generation, the reduction tree, and the final adder for two 32-bit multipliers.*

	Baugh-Wooley		Modified Booth	
	Increment	Total	Increment	Total
Partial Product	0.459 (ns)	0.459 (ns)	0.953 (ns)	0.953 (ns)
Reduction Tree	2.092 (ns)	2.551 (ns)	1.679 (ns)	2.632 (ns)
Final Adder	1.081 (ns)	3.632 (ns)	1.052 (ns)	3.684 (ns)

Table C.9: *The delay for the partial-product generation, the reduction tree, and the final adder for two 48-bit multipliers.*

	Baugh-Wooley		Modified Booth	
	Increment	Total	Increment	Total
Partial Product	0.592 (ns)	0.592 (ns)	1.101 (ns)	1.101 (ns)
Reduction Tree	2.439 (ns)	3.031 (ns)	1.770 (ns)	2.871 (ns)
Final Adder	1.219 (ns)	4.250 (ns)	1.265 (ns)	4.136 (ns)

C.9 Implementation Aspects of the Reduction Tree

The synthesis of the reduction trees used for the results in the previous sections was limited to minimum-sized full-adder (3:2) cells. It would be possible to achieve higher speed by increasing the drive strength of the reduction tree's gates or by using larger (4:2, 7:3, or 9:2) counter cells. This would mainly favor the Baugh-Wooley (BW) implementation since this, in comparison to the modified-Booth (MB) implementation, has a larger critical-path portion inside the reduction circuit. A faster reduction tree circuit, thus, would have a proportionally larger impact on the total delay of the BW implementation. On the other hand, an increased drive strength of the reduction tree's gates would have a negative impact on the power dissipation. In this respect, a BW implementation would experience a relatively higher increase in power compared to an MB implementation. How the power would be affected by larger counters is more difficult to predict without knowing the power of the counter cell. If one counter cell would be less power hungry than the collection of full-adder cells that it replaces, the BW implementation would benefit from using counters.

By varying the full-adder cell drive strength in the reduction circuit, we found that even for the drive strength[4] that yields the highest speed, a BW implementation is still 20-30% more energy efficient than the MB implementation; the exact value of the gain depends on the operand bit-width of the multiplier. Fig.C.13 shows the delay and power for the full-adder drive strength that obtained the best timing. In a comparison with Fig. C.9, we see that the delay is improved at a cost in power.

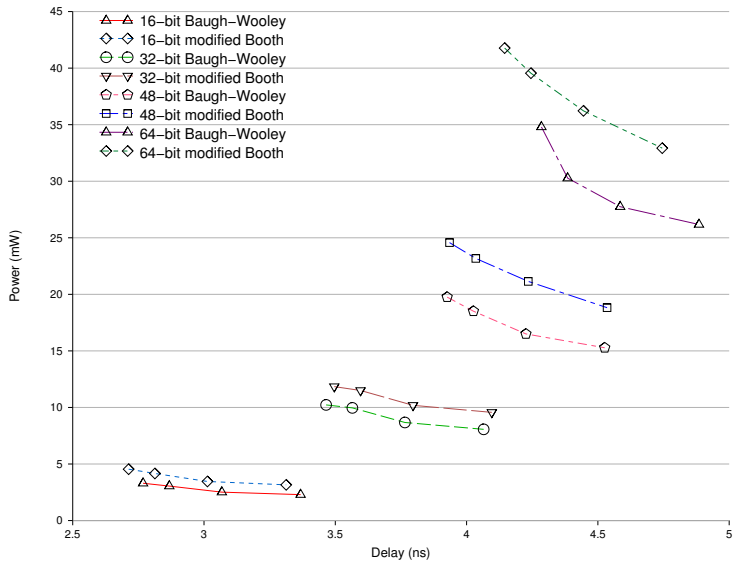


Figure C.13: *Delay and power for a 16-, 32-, 48- and 64-bit Baugh-Wooley and modified-Booth multipliers, for the drive strength of the full-adder cells in the reduction tree that obtains the lowest delay.*

[4]The full-adder cell comes in five different drive strengths in our cell library.

C.10 Implementation in a 65-nm Process Technology

We had limited access to a commercial 65-nm low-power process technology and used it to implement a 32-bit Baugh-Wooley (BW) and modified-Booth (MB) multiplier with standard threshold voltage cells. For the 65-nm design flow, Cadence Encounter [15] is used for synthesis, placement, and routing. For this process technology we were not able to do the same fanout investigations and buffer insertion, as described for the 130-nm process. The synthesis was restricted to use only full-adder cells. However, the placement tool is capable of doing re-synthesis and has in some cases implemented the full-adder functionality as a set of logic cells. Timing and power estimates are for the worst-case 125-degrees corner at 1.1 V. The timing is for lowest delay possible and the power dissipation was estimated using value change dump (VCD) data from simulations with 10,000 random input vectors, as for the 130-nm process.

Table C.10: *Delay, power, energy, and area for 32-bit Baugh-Wooley and modified-Booth multipliers in a 65-nm process.*

	Delay (ns)	Power (mW)	Energy (pJ)	Area (μm^2)
Baugh	2.59 (100%)	23.4 (100%)	60.6 (100%)	48.1k (100%)
Booth	2.50 (97%)	37.5 (160%)	93.8 (155%)	52.1k (108%)

Table C.11: *Delay, power, energy, and area for 32-bit Baugh-Wooley and modified-Booth multipliers in a 130-nm process.*

	Delay (ns)	Power (mW)	Energy (pJ)	Area (μm^2)
Baugh	3.63 (100%)	7.81 (100%)	28.4 (100%)	88.8k (100%)
Booth	3.68 (101%)	9.74 (125%)	35.8 (126%)	108.9k (123%)

The results for 32-bit BW and MB multipliers in the 65-nm process are shown in Table C.10. The high power and energy dissipation of the MB multiplier does not justify the small performance advantage; 90 ps (3%) faster than the BW multiplier. Furthermore, the MB multiplier is also 8% larger in terms of area.

For reference, the corresponding data (those for shortest delay) for the 32-bit 130-nm implementations is shown in Table C.11. The relationship between the BW and MB multiplier does not change much in terms of shortest delay. However, in relation to BW, the power and energy for the MB multiplier increase significantly with process scaling. Regarding area, the area penalty for the MB implementation is reduced from 23% in 130-nm to 8% in 65-nm.

Considering the results for the 65-nm and 130-nm it is clear that at least for a 32-bit multiplier, a MB implementation has higher power dissipation, larger area and only a negligible delay improvement compared to a BW implementation.

C.11 Partial-Product Generation and Final Adders

The usage of the modified-Booth (MB) algorithm makes the reduction tree flatter. This has the effect that the timing profile of the remaining pair of partial products, which are to be summed up by the final adder, are also flatter than compared to a Baugh-Wooley (BW) multiplier. There have been extensive research [16, 17] in adapting the final adder to the timing profile of the partial-product pair from the reduction tree in order to reduce power as well as area. As the timing profile becomes flatter, the room for exploiting timing slacks becomes more limited: A faster final adder is required for the less significant portion of the final result.

For the investigations conducted in this paper, a fast Kogge-Stone adder has been used as final adder. However, the final adder could have been optimized for each specific implementation. In this context, the BW implementations would have gained, in terms of power and area, more from an optimization of the final adder than the MB implementations.

C.12 Conclusion

The modified-Booth algorithm, which is commonly used today, makes multiplier design complex and a significant design effort is needed to obtain an efficient implementation. The design decision of using the modified-Booth al-

gorithm is most likely based on the notion that a smaller reduction circuitry achieves a better implementation. This was once true, but the introduction of logarithmic-depth reduction trees, and particularly the regular structure of the HPM reduction tree, has made the size of the reduction circuit less of a concern when designing a multiplier. The logic depth through the HPM reduction tree differs by only one or two full adders for a modified-Booth and Baugh-Wooley implementation of the same operand bit-width. Considering that the critical path of a modified-Booth multiplier is located in its encoder and decoder, it is difficult to envision a modified-Booth implementation that can be much faster than a Baugh-Wooley implementation, regardless of the recoding scheme used. Taking power, energy per operation, and area into consideration, it is clear that the gain by reducing the reduction circuitry is lost in the recoding circuitry, making a modified-Booth implementation perform worse than a Baugh-Wooley implementation.

Bibliography

[1] Thomas K. Callaway and Jr. Earl E. Swartzlander, "Power-Delay Characteristics of CMOS Multipliers," in *Proceedings of the 13th IEEE Symposium on Computer Arithmetic*, June 1997, pp. 26–32.

[2] Andrew D. Booth, "A Signed Binary Multiplication Technique," *Quarterly Journal of Mechanics and Applied Mathematics*, vol. 4, no. 2, pp. 236–240, 1951.

[3] O.L.MacSorley, "High Speed Arithmetic in Binary Computers," in *Proceedings of the IRE*, January 1961, vol. 49, pp. 67–97.

[4] Jalil Fadavi-Ardekani, "MxN Booth Encoded Multiplier Generator Using Optimized Wallace trees," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 1, no. 2, pp. 120–125, 1993.

[5] Wen-Chang Yeh and Chein-Wei Jen, "High-Speed Booth Encoded Parallel Multiplier Design," *IEEE Transactions on Computers*, vol. 49, no. 7, pp. 692–701, July 2000.

[6] Henrik Eriksson, Per Larsson-Edefors, Mary Sheeran, Magnus Själander, Daniel Johansson, and Martin Schölin, "Multiplier Reduction Tree with Logarithmic Logic Depth and Regular Connectivity," in *IEEE International Symposium on Circuits and Systems*, May 2006.

[7] Steven K. Hsu, Sanu K. Mathew, Mark A. Anders, Bart R. Zeydel, Vojin G. Oklobdzija, Ram K. Krishnamurthy, and Shekhar Y. Borkar, "A 110 GOPS/W 16-bit Multiplier and Reconfigurable PLA Loop in 90-nm CMOS," *IEEE Journal of Solid-State Circuits*, vol. 41, no. 1, pp. 256–264, January 2006.

[8] Charles R. Baugh and Bruce A. Wooley, "A Two's Complement Parallel Array Multiplication Algorithm," *IEEE Transactions on Computers*, vol. 22, pp. 1045–1047, December 1973.

[9] Mehid Hatamian, "A 70-MHz 8-bit x 8-bit Parallel Pipelined Multiplier in 2.5-μm CMOS," *IEEE Journal on Solid-State Circuits*, vol. 21, no. 4, pp. 505–513, August 1986.

[10] Magnus Själander, "HMS Multiplier Generator," `http://www.sjalander.com/research/multiplier`, February 2008.

[11] Peter M. Kogge and Harold S. Stone, "A Parallel Algorithm for the Efficient Solution of a General Class of Recurrence Equations," *IEEE Transactions on Computers*, vol. 22, no. 8, pp. 786–793, August 1973.

[12] *Cadence NC-VHDL Simulator Help Version 5.1*.

[13] *Synopsys Design Compiler User Guide Version W-2004.12*.

[14] *Cadence Encounter User Guide Version 4.1*.

[15] *Cadence Encounter User Guide Version 6.2*.

[16] Vojin G. Oklobdzija, David Villeger, and Simon S. Liu, "A Method for Speed Optimized Partial Product Reduction and Generation of Fast Parallel Multipliers Using an Algorithmic Approach," *IEEE Transactions on Computers*, vol. 45, no. 3, pp. 294–306, March 1996.

[17] Jianhua Liu, Shuo Zhou, Haikun Zhu, and Chung-Kuan Cheng, "An Algorithmic Approach for Generic Parallel Adders," in *IEEE International Conference on Computer Aided Design*, December 2003, pp. 734–740.

PAPER D

M. Själander, H. Eriksson, and P. Larsson-Edefors
An Efficient Twin-Precision Multiplier
IEEE International Conference on Computer Design
San Jose, California, USA, October 10-13, 2004, pp. 30-33.

D

An Efficient Twin-Precision Multiplier

We present a twin-precision multiplier that in normal operation mode efficiently performs N-b multiplications. For applications where the demand on precision is relaxed, the multiplier can perform $N/2$-b multiplications while expending only a fraction of the energy of a conventional N-b multiplier. For applications with high demands on throughput, the multiplier is capable of performing two independent $N/2$-b multiplications in parallel. A comparison between two signed 16-b multipliers, where both perform single 8-b multiplications, shows that the twin-precision multiplier has 72% lower power dissipation and 15% higher speed than the conventional one, while only requiring 8% more transistors.

D.1 Introduction

Recent development at the micro architecture level shows that there is an increasing interest in datapath components that are capable of performing computations with variable operand size, e.g. adders capable of doing both N and $N/2$-b additions [1]. By using only a part of the datapath component for computation, it has been demonstrated [2] that reductions in the total power dissipation can be effected. Datapath components that can perform both one N, one single $N/2$, or two $N/2$-b operations give the designer the opportunity to design a system which can adapt to changing modes, such as low-power, high-throughput, or high-precision operation. Such a datapath component could be used for dynamic power reduction in the same way as described by Abddollahi *et al*. [2]; by using the same kind of logic for detecting if the effective bit rate is within $N/2$-b precision, it is possible to control at what precision the datapath component should be operating. This versatile type of datapath component is also suitable for systems in which several applications, having quite different requirements on precision and/or throughput, are executed [3]. Furthermore, such a datapath component could prove useful in processors that can support several instruction sets. In a processor that combines x86-32 and x86-64, a flexible datapath could be used for 64-b operations as well as for Single Instruction Multiple Data (SIMD) instructions, where two 32-b operations are performed in parallel.

It has been shown [4] that it is relatively straightforward to partition an array multiplier, so as to obtain a multiplier that can perform multiplications with varying operand size[1]. In comparison to tree multipliers, however, an array multiplier is slow and power hungry which makes it a poor design choice when a fast and efficient multiplier is needed [5]. It was claimed, but not substantiated, that the power-reduction techniques used for array multipliers [2] can be applied also to tree multipliers. It is certainly not straightforward to transfer the proposed technique to tree multipliers. Mokrian *et al.* presented a reconfigurable multiplier, which is constituted by several smaller tree multipliers [6]. However, the recursive nature of this multiplier is, due to an addition of reduction

[1] This was done by gating parts of the array of carry-save adders and by using multiplexers to read out the data from a low-precision multiplication.

stage(s), likely to have a large impact on the delay for the N-b multiplication, compared to the multiplier proposed in this paper.

In the following we explore the possibility of combining N and $N/2$-b multiplications in the same N-b tree multiplier: we call this a twin-precision multiplier. The key challenges in designing a twin-precision multiplier are to limit the impact of flexibility on power dissipation, delay, and area. The proposed twin-precision multiplier efficiently performs either one N-b multiplication, one single $N/2$-b multiplication, or two $N/2$-b multiplications in parallel.

D.2 Design Exploration

Based on a simple representation of an array multiplier, Figure D.1, it is obvious that if the partial product bits not being used in a low-precision multiplication are set to zero, the array multiplier will produce the correct result without the need of any additional logic. The 2-input AND gates corresponding to the partial product bits that are not being used in the low-precision multiplication can be replaced by 3-input AND gates to force those bits to zero [2].

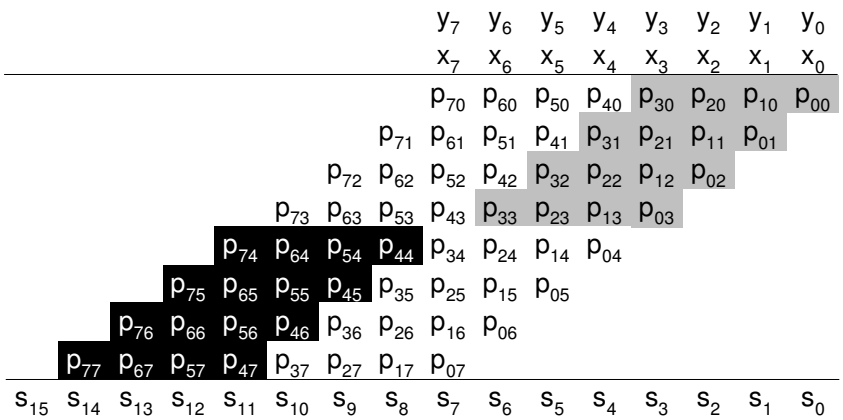


Figure D.1: *Partial product representation of a 4-b multiplication in an 8-b multiplier.*

[2] When performing only one $N/2$-b multiplication it is possible to set the most significant bits of the operands to zero instead.

When doing an $N/2$-b multiplication within an N-b multiplier only one quarter of the logic is being used, as seen in grey in Figure D.1. This makes it possible to use the multiplier for two parallel and independent $N/2$-b multiplications. We can partition the partial product bits of the N-b multiplier, such that an $N/2$-b multiplication can be performed in the Least Significant Part (LSP) of the multiplier in parallel with another $N/2$-b multiplication in the Most Significant Part (MSP), without using any additional logic in the partial product reduction tree, as seen in grey and black, respectively, in Figure D.1. To be able to switch between N, $N/2$, or two $N/2$-b multiplications, the 2-input AND gates used to create the partial products need to be replaced with 3-input AND gates and two control signals for selecting the operating mode of the multiplier need to be introduced.

D.2.1 Tree Multiplier

Until now we have implicitly used the array multiplier to demonstrate the twin-precision feature. The array multiplier is, however, slow and power dissipating in comparison to a logarithmic tree multiplier. The implementation of the twin-precision feature in an N-b tree multiplier is similar to that of the array multiplier; all that is needed is to set the partial products bits not being used to zero and to partition the partial products bits of the two multiplications into the respective LSP and MSP of the tree. To reduce the critical path for the $N/2$-b multiplications the partial products bits used during the computation are moved as far down the tree as possible, Figure D.2. In this paper we use a tree multiplier with regular connectivity [7].

To further reduce the critical path of the $N/2$-b multiplications it is possible to move the partial products even further down the tree by adding multiplexers on lower levels[3]. This makes it possible to select either the carry and sum from higher levels, when doing the N-b multiplication, *or* the partial products bits, when doing the $N/2$-b multiplication. This introduces multiplexers in the critical path of the N-b multiplier, which significantly increases the delay of the N-b multiplication. Thus, this alternative has not been considered here, since our

[3] Moving further down in the tree implies approaching the final adder.

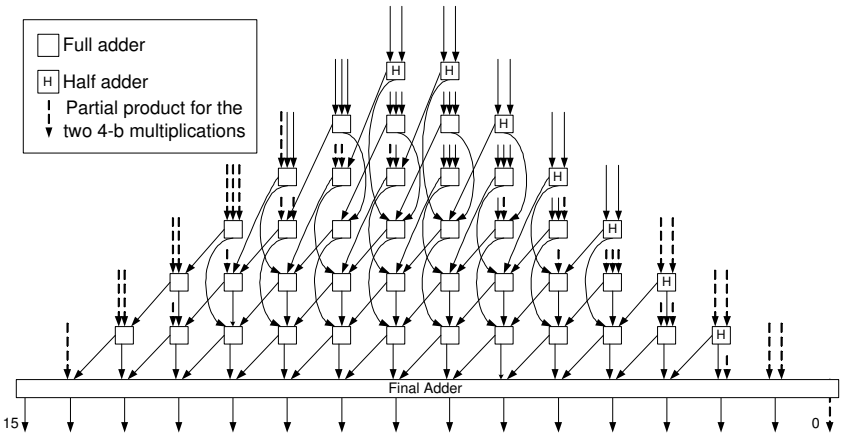


Figure D.2: *Partitioned tree of an 8-b multiplier.*

goal is to find a good design tradeoff between the delay of N-b multiplications and $N/2$-b multiplications, respectively.

D.2.2 Signed Multiplication According to Baugh-Wooley

We used the Baugh-Wooley algorithm [8] to investigate the impact of the twin-precision feature on delay and power of a signed tree multiplier [4]. Here, signed multiplication is performed by first inverting all partial product bits that are results of the most significant bit (MSB) of exactly one of the operands, Figure D.3. Second, for each executed multiplication, a logical one (framed) is added to column N (column 0 is to the far right in Figure D.3) and, third, the MSB of the product is inverted. This is directly mapped onto the tree multiplier as shown in Figure D.4.

To be able to generate the inverted partial product bits, we chose to replace the AND gates corresponding to the inverted bits with NAND gates followed by XOR gates. The option to either invert or not invert the signal from the NAND gates makes it possible to switch between signed and unsigned multiplication

[4] Modified Booth does not impose any fundamental problems to the twin-precision concept. It has been evaluated, but is not included in this paper because of space constraints.

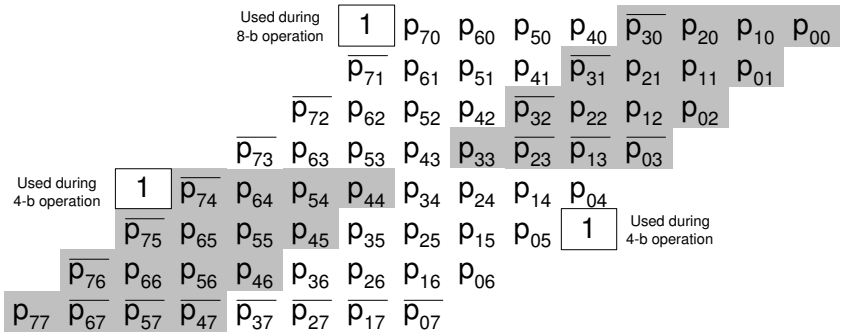


Figure D.3: *Example showing the inverted partial product bits of two signed 4-b multiplications within a signed 8-b multiplication.*

The inversion of the MSB of the product is also done with an XOR gate. The insertion of the logical one to column N of the multiplication is straightforward for the N-b and the $N/2$-b multiplication in the LSP by changing the half adder of that column to a full adder and adding the logical one to the new adder. For the $N/2$-b multiplication in the MSP there is no half adder that can be replaced, but an extra level of half adders has to be added, seen at the far left in Figure D.4. This added level of half adders does not increase the delay for the N-b multiplication, since none of the half adders are in the critical path.

D.3 Final Adder

The choice of final adder is very important in order to get short delay for both N and $N/2$-b multiplications. The recommendations given by Oklobdzija *et al.* [9] are not directly applicable in our twin-precision multiplier, since the delay profile of the multiplier varies with the multiplication precision. It would be possible to use the adder scheme presented by Oklobdzija *et al.* to reduce the delay for the N-b multiplication, but this could introduce long delays for the two independent $N/2$-b multiplications. In order to not increase the delay too much for the $N/2$-b multiplications, it is therefore important to have a final adder that is fast for both N and $N/2$-b multiplications. Mathew *et al.* [1] presented

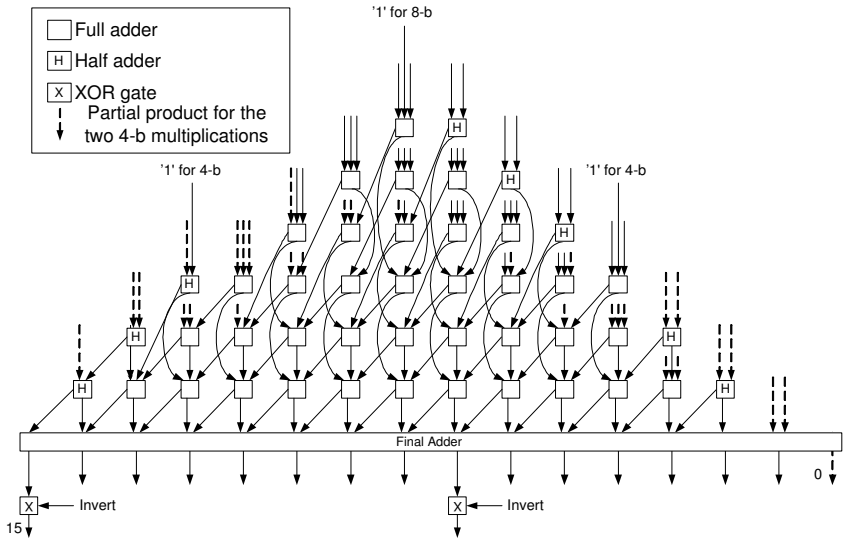


Figure D.4: *Signed 8-b multiplier capable of doing two signed 4-b multiplications using Baugh-Wooley.*

a sparse-tree carry-lookahead adder that is capable of doing both fast 64-b and fast 32-b additions. This adder scheme has been adapted to the appropriate word length in order to obtain short delays for both N and $N/2$-b multiplications, Figure D.5.

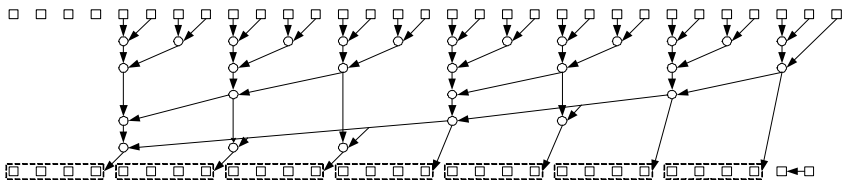


Figure D.5: *Example of 31-b final adder for a 16-b twin-precision multiplier.*

D.4 Simulation Setup and Results

To evaluate delay and power dissipation, simulations have been performed in a commercially available 0.13-μm technology. The simulated circuit is a 16-b twin-precision tree multiplier, which is capable of performing two 8-b multiplications in parallel, and which uses the fast final adder of Section 3. As reference we use a conventional 8-b and 16-b multiplier, respectively, which both use a Kogge-Stone as final adder. For signed multiplication the Baugh-Wooley algorithm [8] has been implemented for both the conventional and the twin-precision multiplier. All simulations have been done using Spice transistor netlists including estimated wire capacitances. All logic has been implemented as static logic and designed to resemble what could be expected to be found in a standard-cell library. The implemented version of the multiplier cancels the inactive partial products by forcing the AND gates to zero. The impact of using sleep-mode techniques on power and delay has not been investigated[5]. For power simulation 50 random input vectors were applied to HSpice at 500 MHz, a supply voltage of 1.2 V, and an operating temperature of 25 °C. Delay was obtained using PathMill.

Table D.1: *Reference values normalized to the conventional 8-b multiplier.*

Reference	Delay	Power
16 bit	1.40	4.06
8 bit	1.0	1.0

Table D.1 lists the values for the conventional 8-b and 16-b multipliers used as comparison references. Table D.2 lists delay and power for the twin-precision multiplier. All values have been normalized to the 8-b reference multiplier.

With a conventional 16-b multiplier as reference, the delay of the 16-b twin-precision multiplier operating in 16-b mode was 9.0% larger whereas the power dissipation was less than 1.0% larger. When using the 16-b twin-precision multiplier in single 8-b mode, the power dissipation is only 28% of the reference 16-b multiplier. The reason for the power reduction is that in single 8-b mode

[5]We expect no fundamental problems in introducing sleep-mode techniques in the twin-precision multiplier.

Table D.2: *Simulation results for a twin-precision 16-b multiplier, where columns 2 and 3 are normalized to the conventional reference 8-b multiplier. Columns 4 to 7 are comparisons against the conventional reference multipliers given in Table 1.*

Mode	Delay	Power	Compared to 8-b		Compared to 16-b	
			Delay	Power	Delay	Power
16-b	1.52	4.10			9.0%	0.9%
2x8-b	1.29	2.34	29.0%	16.9%	-7.5%	-42.4%
8-b	1.18	1.13	18.2%	13.3%	-15.3%	-72.1%

about two thirds of the multiplier tree is kept at constant zero, eliminating the dynamic power in these parts. The additional decrease in power comes from the reduction of glitches in the multiplier.

With a conventional 8-b multiplier as reference, the delay of the 16-b twin-precision multiplier operating in single 8-b mode was 18.2% larger whereas the power dissipation was 13.3% larger.

When doing two 8-b multiplications in parallel the power dissipation increases by about 4% and the delay is increased with 10% compared to a single 8-b multiplication. The increase in the delay is due to an increased logic depth—the 8-b multiplication in the MSP of the multiplier tree has a longer critical path than the 8-b multiplication in the LSP has, Figure D.4. Additionally the logical depth of the final adder is one gate deeper for the MSP which contributes to a longer critical path for the 8-b multiplication computed in the MSP of the tree.

The power dissipation for driving the control signals to set the mode of the multiplier was not included in the power simulation. The control signal used to set the partial product bits to zero, when doing two $N/2$-b multiplications, is connected to the input of N AND gates. In order to cancel out the second $N/2$-b multiplication the control signal is connected to the input of $N/2$ AND gates. It has been shown that it is realistic to expect the multiplier to operate in the same mode for longer durations [2]. Since the control signals only toggle when the mode of the multiplier is changed, the power dissipation for these signals is negligible when the multiplier stays in one mode for longer durations.

D.5 Conclusion

The twin-precision multiplier presented in this paper offers a good tradeoff between precision flexibility, area, delay and power dissipation by using the same multiplier for doing N, $N/2$ or two $N/2$-b multiplications. In comparison to a conventional 16-b multiplier, a 16-b twin-precision multiplier has 8% higher transistor count and 9% longer delay. The relative transistor count overhead decreases for larger multipliers, since the number of AND gates needed to set the partial products to zero does not grow as fast as the number of adders in the tree.

Bibliography

[1] Sanu Mathew, Mark Anders, Brad Bloechel, Tran Nguyen, Ram Krishnamurthy, and Shekhar Borkar, "A 4GHz 300mW 64b Integer Execution ALU with Dual Supply Voltages in 90nm CMOS," in *Proceedings of the International Solid State Circuits Conference*, 2004, pp. 162–163.

[2] Afshin Abddollahi, Massoud Pedram, Farzan Fallah, and Indradeep Ghosh, "Precomputation-Based Guarding for Dynamic and Leakage Power Reduction," in *IEEE Proceedings of the 21st International Conference on Computer Design*, 2003, pp. 90–97.

[3] John Hughes, Kjell Jeppson, Per Larsson-Edefors, Mary Sheeran, Per Stenström, and Lars "J." Svensson, "FlexSoC: Combining Flexibility and Efficiency in SoC Designs," in *Proceedings of the IEEE NorChip Conference*, 2003.

[4] Zhijun Huang and Miloš D. Ercegovac, "Two-Dimensional Signal Gating for Low-Power Array Multiplier Design," in *Proceedings of the IEEE International Symposium on Circuits and Systems*, 2002, pp. I–489–I–492 vol.1.

[5] Thomas K. Callaway and Earl E. Swartzlander, Jr., "Optimizing Multipliers for WSI," in *Proceedings of the Fifth Annual IEEE International Conference on Wafer Scale Integration*, 1993, pp. 85–94.

[6] Pedram Mokrian, Majid Ahmadi, Graham Jullien, and W.C. Miller, "A Reconfigurable Digital Multiplier Architecture," in *Proceedings of the IEEE Canadian Conference on Electrical and Computer Engineering*, 2003, pp. 125–128.

[7] Henrik Eriksson, *Efficient Implementation and Analysis of CMOS Arithmetic Circuits*, Ph.D. thesis, Chalmers University of Technology, 2003.

[8] Charles R. Baugh and Bruce A. Wooley, "A Two's Complement Parallel Array Multiplication Algorithm," *IEEE Transactions on Computers*, vol. 22, pp. 1045–1047, December 1973.

[9] Vojin G. Oklobdzija, David Villeger, and Simon S. Liu, "A Method for Speed Optimized Partial Product Reduction and Generation of Fast Parallel Multipliers Using an Algorithmic Approach," *IEEE Transactions on Computers*, vol. 45, no. 3, pp. 294–306, March 1996.

PAPER E

M. Själander, M. Draždžiulis, P. Larsson-Edefors, and H. Eriksson
A Low-Leakage Twin-Precision Multiplier Using Reconfigurable Power Gating
IEEE International Symposium on Circuits and Systems
Kobe, Japan, May 23-26, 2005, pp. 1654-1657.

E

A Low-Leakage Twin-Precision Multiplier Using Reconfigurable Power Gating

A twin-precision multiplier that uses reconfigurable power gating is presented. Employing power cut-off techniques in independently controlled power-gating regions, yields significant static leakage reductions when half-precision multiplications are carried out. In comparison to a conventional 8-bit tree multiplier, the power overhead of a 16-bit twin-precision multiplier operating at 8-bit precision has been reduced by 53% when reconfigurable power gating based on the SCCMOS power cut-off technique was applied.

E.1 Introduction

Recent development of embedded systems indicates an increased interest in reconfigurable functional units that dynamically can adapt the datapath to varying computational needs. A system may need to switch between, for example, one application that needs speech-encoding functional units operating at 8-bit precision and another application that needs 16-bit functional units to perform audio decoding. Since most embedded systems are associated with a strictly limited power budget, both static and dynamic power are of critical importance. In this context, it is hardly acceptable to have idle functional units which dissipate useless static power while the application is running at low precision.

The twin-precision (TP) multiplier [1] can switch between N-bit and $N/2$-bit precision multiplications without significant performance and area overhead. However, in half-precision ($N/2$-bit) mode the TP multiplier is dissipating considerably more power than a conventional fixed-precision $N/2$-bit multiplier, since idle, higher-precision logic gates within the TP multiplier are leaking. In order to efficiently utilize reconfigurable datapath units, leakage reduction techniques need to be incorporated.

In this paper we describe a twin-precision multiplier that uses a power-gating strategy that dynamically adapt to the precision needed and shuts down idle portions of the circuit to save static leakage power.

E.2 Preliminaries

E.2.1 The Twin-Precision Multiplier

The twin-precision multiplier [1] is an N-bit tree multiplier that is capable of performing either *i)* single N-bit, *ii)* single $N/2$-bit or *iii)* two concurrent and independent $N/2$-bit multiplications. When compared to a conventional fixed-precision N-bit tree multiplier, the general gate count overhead of the twin-precision multiplier was very small, and the overhead in delay and active power dissipation for the full-precision mode was shown to be small [1]. The most important features of the twin-precision technique are illustrated in Fig. E.1.

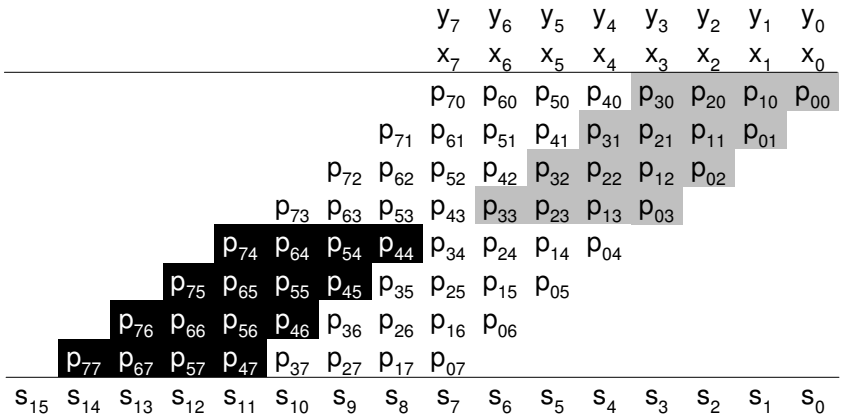


Figure E.1: *Partial product representation of an 8-bit multiplication. When performing a 4-bit multiplication in a conventional multiplier only one quarter of the logic gates are performing any useful operations (the grey regions). The twin-precision technique rearranges logic to greatly reduce power dissipation and delay for 4-bit multiplications. Also, this technique allows the logic gates, which are not used in the 4-bit multiplication taking place in the grey region, to perform a second, independent 4-bit multiplication (the black regions).*

In order to switch between the three different precision modes it is necessary that the partial products in white and black regions (Fig. E.1) can be forced to zero independently of each other. The 2-input AND gates, which in conventional multipliers generate the partial product bits, are replaced with 3-input AND gates, whose extra input is used to mask the output to zero.

In a twin-precision multiplier it is crucial that the partial products used for *N*/2-bit multiplications are moved as close to the final adder as possible. In half-precision mode, this gives the shortest critical path and the largest number of unused logic gates, which can be translated into dramatic dynamic power reductions. The final adder is also important, since its delay profile will determine the delay for the *N*-bit and the two *N*/2-bit multiplications. In conventional adder design, the delay is optimized for full-precision operation. This generally makes such adders relatively slow when used for lower precision. A good

tradeoff between the delay for N-bit and $N/2$-bit precision is given by the adder presented by Mathew *et al.* [2].

The application of the twin-precision technique to signed N-bit and $N/2$-bit multiplications is quite straightforward. An example of an 8-bit signed multiplier using the Baugh-Wooley algorithm [3], capable of performing two concurrent 4-bit multiplications, is shown in Fig. E.2. The twin-precision technique can also be applied to modified Booth multipliers, but this requires a slightly extended implementation effort.

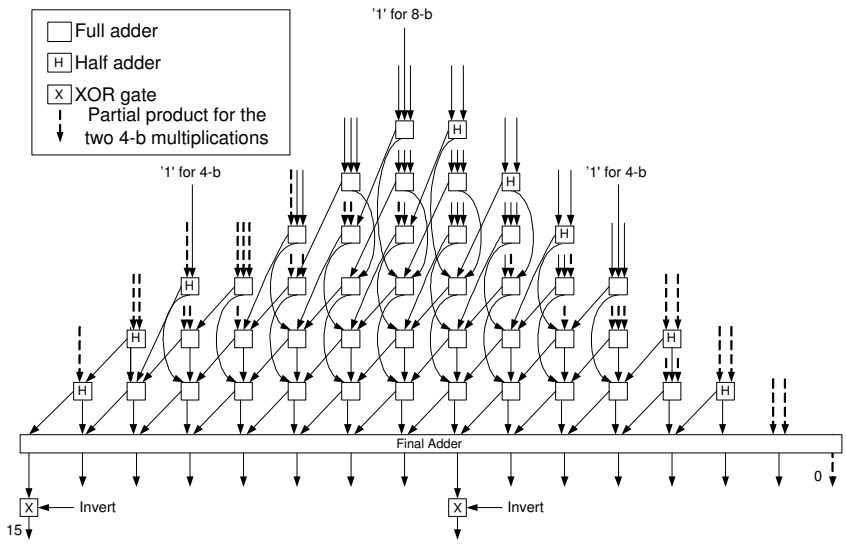


Figure E.2: *Signed 8-bit twin-precision Baugh-Wooley multiplier.*

E.2.2 Circuits for Leakage Reduction

As soon as unused circuit parts of a twin-precision (TP) multiplier stop switching or when the entire multiplier goes into sleep mode, static leakage currents become visible. Transistor stacking, body biasing, multi-V_t technologies and power cut-off techniques are common methods used to suppress static leakage. We have chosen to study only power cut-off techniques in this paper, be-

cause these are compatible with conventional CMOS design and require no special process technologies. Power cut-off techniques are using power switches (PMOS or NMOS transistors) that are turned *on* when logic circuits are active and *off* when circuits are in sleep mode. Generally, a power cut-off technique is efficient for circuits that stay in sleep mode for relatively long periods of time.

In this paper we have considered a number of power cut-off techniques [4] that can be applied to a TP multiplier: Super Cut-Off CMOS (SCCMOS), Zigzag Super Cut-Off CMOS (ZSCMOS) and Gate leakage Suppression CMOS (GSCMOS). We found that SCCMOS has long sleep-to-active (wake-up) time, while ZSCCMOS has short wake-up time at the expense of a need for sleep-mode state control (input forcing). The GSCMOS technique, on the other hand, is particularly efficient when gate leakage is significant, but it has complex supply rail routing. All considered power cut-off techniques need on-chip voltage generators to control the power switches. Although power cut-off techniques corrupt logic data during sleep, this is not a serious issue for the TP multiplier, since a fresh multiplication is performed every time the precision mode has been changed.

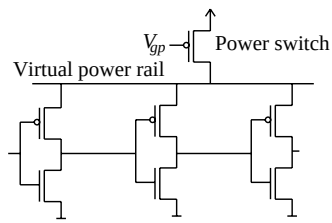


Figure E.3: *The SCCMOS technique applied to an inverter chain.*

We chose the SCCMOS technique (Fig. E.3), since this fits our needs and constraints (e.g. low supply routing complexity, a wake-up time within a few clock cycles, and absence of gate leakage) in the 0.13-μm technology used. The power switch used here is a low-V_t PMOS transistor, which in sleep mode is overdriven (i.e. above V_{dd}) with ΔV, which roughly is the threshold voltage difference between a typical high-V_t and a typical low-V_t device. The overdrive voltage completely turns the power switch *off*, thus the voltage on the virtual power rail drops and leakage currents are reduced.

E.3 Power Supply Grid and Tree Organization

To suppress static leakage in idle gates of the twin-precision (TP) multiplier, we deploy the SCCMOS technique so that three separate power-gating regions (Fig. E.4) are obtained. Independent of other regions, the power switches of one region can be turned *on* or *off*, depending on multiplier precision mode.

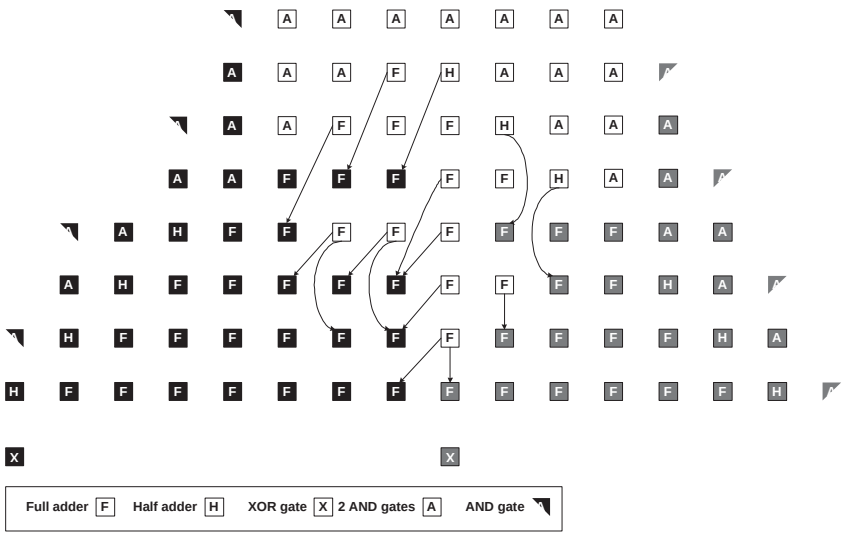


Figure E.4: *Power-gating regions of an 8-bit TP multiplier. When one* N*/2-bit multiplication is performed, the power switches in black and white regions are turned* off. *When two concurrent* N*/2-bit multiplications are performed, the power switches in the white region are turned* off. *Finally, when an* N*-bit multiplication is performed, the power switches in all regions are turned* on.

When the TP multiplier performs an $N/2$-bit multiplication or two concurrent $N/2$-bit multiplications, outputs of some sleeping logic gates are connected to the inputs of active logic gates (Fig. E.4). Since those outputs now are floating, the inputs of the active logic gates can not be asserted via the 3-input AND gates that were used in the original TP multiplier. Instead, the inputs of these active logic gates are pulled down by NMOS transistors, which are connected to the respective outputs of the sleeping gates. When the TP multiplier per-

forms an *N*/2-bit multiplication or two concurrent *N*/2-bit multiplications, the pull-down NMOS transistors are turned *on*, thus forcing the outputs of sleeping logic gates (i.e. the inputs of the active logic gates) to ground. The *off*-state power switches eliminate short-circuit currents in sleeping logic gates.

The virtual power rails that are introduced with the SCCMOS technique need special consideration. Within each power-gating region, the virtual power rails can be partitioned into networks of finer granularity than a single network spanning the entire region. In fact, every logic gate could have its own virtual power rail and power switch, leading to all logic gates having individual virtual supply voltages (which are functions of the gate input pattern) and guaranteeing overall optimal leakage suppression in sleep mode. However, this calls for an extra power-switch control wire that needs to be routed through *every* cell in a layout, in turn causing cell footprints and wire lengths to grow. Longer signal wires lead to larger switched capacitance and, thus, both the dynamic power dissipation and the delay of the active TP multiplier regions will increase. As for the other extreme, i.e. leaving the virtual power rails within each power-gating region non-partitioned, this is generally a recipe for larger than minimal leakage currents. This is because all logic gates of a large power-gating region will have one common voltage level on the virtual power rail. In sleep mode this common level rarely leads to minimal leakage currents in individual logic gates.

For the SCCMOS implementation of this paper (Fig. E.5) we employ one virtual power rail and one power switch to groups of four full-adder (FA) cells (or for any group of logic cells that have the same total footprint). This partitioning yields a regular power supply grid that is easy to implement in layout. In our implementation, we assume constant cell pitch (cell height) and, as shown in Fig. E.5, two AND gates are concatenated into one cell to make cell footprints uniform (two AND gates are now as wide as one FA cell).

In active mode, the power switch resistance will cause supply voltages on virtual power rails to drop below external supply voltage levels and, hence, the circuit delay will increase. The worst (largest) supply voltage drop takes place when the gates on a particular virtual power rail receive the input signals that make the maximum number of PMOS transistors switch simultaneously.

We choose to size the power switches so that during switching, for the worst-case conditions described above, the voltage drop on all virtual power rails is equal. Therefore, all virtual power rails have identical width ratios (K), where K is defined for each virtual power supply network as the power switch width divided by the total width of the PMOS transistors that simultaneously switch for the worst-case condition.

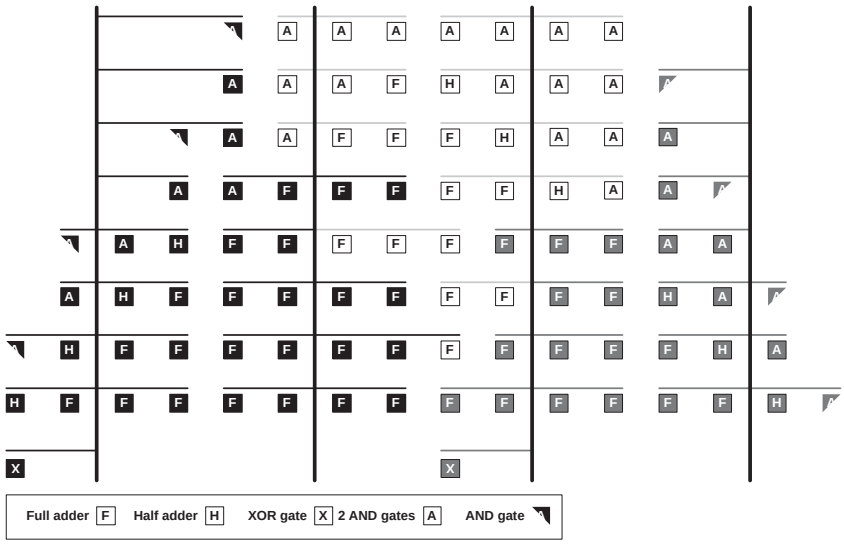


Figure E.5: *The power supply grid of the 8-bit TP multiplier. Here, virtual power rails are routed horizontally, whereas external power and ground rails are routed vertically.*

Regarding the layout, power switches are inserted where an external power rail crosses a virtual power rail (Fig. E.5). If an external power rail is routed between two different power-gating regions then *two* power switches (for two different virtual power rails) are inserted at each crossing. Since the N-wells of the PMOS transistors inside the logic gates are connected to the virtual power rails, the external power rails do not have to be routed horizontally. Hence, in the proposed power supply grid and tree organization the area penalty is limited to the vertically routed external supply rails. Since these rails increase signal wire lengths between cells, we also have a delay and power penalty when circuits are active.

Logic gates in the white region, which are "trapped" between the grey and black regions and which do not have external supply rail routed through them, are connected to the black-region virtual power rails; the "white" FA-cell in the third row from the bottom in Fig. E.5 is an example of a "trapped" logic gate. This connection simplifies the routing but incurs a small power penalty when two concurrent $N/2$-bit multiplications are performed, since "trapped" gates will be connected to the external power rails via the *on*-state black-region power switches.

Fig. E.5 has a final adder (in the second row from the bottom) which for the sake of simplicity in this example is a ripple-carry adder. This adder can be replaced by any kind of tree adder, which subsequently can be partitioned in the same way as a multiplier reduction tree.

E.4 Simulation and Results

We will now evaluate power dissipation and delay of a 16-bit twin-precision multiplier *with power-gating regions employing the SCCMOS technique*. As the first reference design we use a conventional twin-precision multiplier, which *does not* use power gating. In the rest of this section, *TP* refers to the conventional twin-precision multiplier in [1], whereas *TP-cutoff* refers to the proposed power-gated twin-precision multiplier. In order to strike a good balance between the delay of 8-bit and 16-bit multiplications, both the TP and TP-cutoff multipliers use the adder presented by Mathew *et al.* [2] as final adder.

As a second set of references we use conventional 8-bit and 16-bit tree multipliers. These references represent fixed-precision multipliers, which have final adders that are optimized for either 8-bit or 16-bit multiplications. Here, we used the Kogge-Stone [5] structures for final adders. Both conventional tree multipliers employ the SCCMOS technique with a power supply network partitioning which is similar to that of the TP-cutoff multiplier.

Simulation data were obtained by running HSpice on a commercially available 0.13-μm technology, at a supply voltage of 1.2 V and an operating temperature of 80°C. Circuit netlists were constructed using static CMOS gates and estimated wire capacitances (pre-layout). Power switches were overdriven with

0.2 V. Power figures were obtained by running simulations for 50 random input vectors applied at a 500-MHz frequency. Power dissipation from control signal transitions and overdrive voltage generators was not included.

We could not use PathMill [6] directly to obtain reliable delay figures for the power-gated multipliers. Instead, delay figures for all multipliers had to be obtained by the following steps: First, using PathMill, we found the critical path for a multiplier that did not use the SCCMOS technique. Then we constructed netlists of *only the critical path* for *i)* a multiplier without the SCCMOS technique and *ii)* a multiplier using the SCCMOS technique. These netlists combine, first, the structure of the critical path originally obtained by PathMill and, second, the logic gates used in the HSpice netlists of the respective complete multipliers. Finally, the critical paths were simulated using HSpice. Inputs to the critical paths were asserted such that signal rippling was guaranteed. To define accurate fan-outs, the critical-path netlists also contained all logic gates and respective wire capacitances that the rippling signals were driving. To realistically model capacitance on the virtual power rails, all logic gates connected to the power switches were also included.

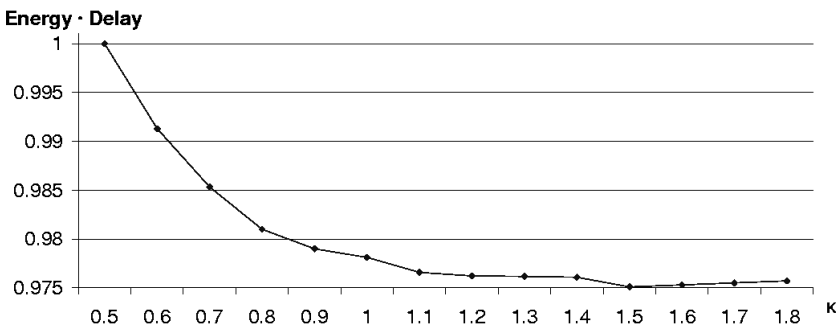


Figure E.6: *Normalized energy-delay product for different power switch sizes.*

Fig. E.6 shows the energy-delay product in a 16-bit TP-cutoff multiplier (operating at 8-bit precision) as a function of power-switch size. When K=1.0, for each virtual supply network the power switch width is equal to the total width of the PMOS transistors that simultaneously switch for the worst-case condition. From Fig. E.6 it can be observed that K=1.1 gives a good compro-

mise between expended energy and delay, so K=1.1 is used henceforth. Also, with K=1.1 the wake-up times for the power-gated multipliers were within one clock cycle.

Table E.1: *Total power dissipation [*mW*] for 8-bit multiplications.*

(a) Conventional

Mult.	Mode	Cutoff
8-bit	8-bit	0.9667
16-bit	8-bit	3.473

(b) Twin-precision

Mult.	Mode	TP	TP-cutoff
16-bit	8-bit	1.216	1.083

The power dissipation of the 16-bit TP-cutoff multiplier operating at 8-bit precision can be compared to the power of an active conventional 8-bit multiplier, which incorporates power cut-off circuit techniques. In fact, the latter multiplier represents the lower power dissipation bound for any 16-bit multiplier that operates in 8-bit precision mode. As Table E.1 shows, the 16-bit TP-cutoff multiplier only dissipates 12% more power than a conventional 8-bit multiplier. In comparison to a conventional 16-bit multiplier, which operates on 8-bit precision operands that are sign extended to 16 bits, the TP-cutoff multiplier has 3.2 times lower power dissipation. We finally observe that by using precision-dependent power-gating based on the SCCMOS technique, the power overhead (with reference to an active conventional 8-bit multiplier) of the 16-bit TP multiplier operating at 8-bit precision is reduced by as much as 53%.

Table E.2: *Delays [*ns*] for 8-bit and 16-bit multiplications.*

(a) Conventional

Mult.	Mode	Cutoff
8-bit	8-bit	1.190
16-bit	16-bit	1.687

(b) Twin-precision

Mult.	Mode	TP	TP-cutoff
16-bit	8-bit	1.400	1.402
16-bit	16-bit	1.759	1.805

As shown in Table E.2, in 16-bit precision mode the TP multiplier is less than 3% faster than the TP-cutoff multiplier. The conventional 16-bit tree multiplier that uses the SCCMOS technique is 7% faster than the TP-cutoff multiplier operating at 16-bit precision, a ratio that is largely due to the use of different

final adders—Kogge-Stone in the former and the adder by Mathew *et al.* in the latter.

The difference in delay between the TP multiplier and the TP-cutoff multiplier operating in 8-bit precision mode is hardly noticeable. However, not surprisingly, the conventional 8-bit tree multiplier using the SCCMOS technique outperforms both twin-precision multipliers with 18%. This is due to *i)* use of a fixed 8-bit final adder and *i)* lower logic depth in the 8-bit reduction tree.

E.5 Conclusions

We have shown that power cut-off techniques can be deployed in different regions of a twin-precision functional unit, so that static leakage reduction can be effected not only when the entire unit is idle, but also when only parts of the unit are active, i.e. when the unit operates in half-precision mode.

Bibliography

[1] Magnus Själander, Henrik Eriksson, and Per Larsson-Edefors, "An Efficient Twin-Precision Multiplier," in *IEEE Proceedings of the 22nd International Conference on Computer Design*, October 2004, pp. 30–33.

[2] Sanu Mathew, Mark Anders, Brad Bloechel, Tran Nguyen, Ram Krishnamurthy, and Shekhar Borkar, "A 4GHz 300mW 64b Integer Execution ALU with Dual Supply Voltages in 90nm CMOS," in *Proceedings of the International Solid State Circuits Conference*, 2004, pp. 162–163.

[3] Charles R. Baugh and Bruce A. Wooley, "A Two's Complement Parallel Array Multiplication Algorithm," *IEEE Transactions on Computers*, vol. 22, pp. 1045–1047, December 1973.

[4] Mindaugas Dražd̄iulis, Per Larsson-Edefors, Daniel Eckerbert, and Henrik Eriksson, "A Power Cut-Off Technique for Gate Leakage Suppression," in *European Solid-State Circuits Conference*, September 2004, pp. 171–174.

[5] Peter M. Kogge and Harold S. Stone, "A Parallel Algorithm for the Efficient Solution of a General Class of Recurrence Equations," *IEEE Transactions on Computers*, vol. 22, no. 8, pp. 786–793, August 1973.

[6] *PathMill user guide, Version U-2003.03-SP1*.

PAPER F

M. Själander and P. Larsson-Edefors
Multiplication Acceleration Through Twin Precision
IEEE Transactions on VLSI Systems
Conditionally Accepted: Minor Revision, March 17, 2008.

F

Multiplication Acceleration Through Twin Precision

We present the twin-precision technique for integer multipliers. The twin-precision technique can reduce the power dissipation by adapting a multiplier to the bitwidth of the operands being computed. The technique also enables an increased computational throughput, by allowing several narrow-width operations to be computed in parallel. We describe how to apply the twin-precision technique also to signed multiplier schemes, such as Baugh-Wooley and modified-Booth multipliers. It is shown that the twin-precision delay penalty is small (5-10%) and that a significant reduction in power dissipation (40-70%) can be achieved, when operating on narrow-width operands. In an application case study, we show that by extending the multiplier of a general-purpose processor with the twin-precision scheme, the execution time of a Fast-Fourier Transform is reduced with 15% at a 14% reduction in datapath energy dissipation. All our evaluations are based on layout-extracted data from multipliers implemented in 130-nm and 65-nm commercial process technologies.

F.1 Introduction

Multiplication is a complex arithmetic operation, which is reflected in its relatively high signal propagation delay, high power dissipation, and large area requirement. When choosing a multiplier for a digital system, the bitwidth of the multiplier is required to be at least as wide as the largest operand of the applications that are to be run on that digital system. The bitwidth of the multiplier is, therefore, often much larger than the data represented inside the operands, which leads to unnecessarily high power dissipation and unnecessary long delay. This resource waste could partially be remedied by having several multipliers, each with a specific bitwidth, and use the particular multiplier with the smallest bitwidth that is large enough to accommodate the current multiplication. Such a scheme would assure that a multiplication would be computed on a multiplier that has been optimized in terms of power and delay for that specific bitwidth. However, using several multipliers with different bitwidths would not be an efficient solution, mainly because of the huge area overhead[1].

There have been several studies on operand bitwidths of integer applications and it has been shown that for the SPECint95 benchmarks more than 50% of the instructions are instructions where both operands are less than or equal to 16 bits [1] (henceforth called narrow-width operations). This property has been explored to save power, through operand guarding [1–3]. In operand guarding the most significant bits of the operands are not switched, thus power is saved in the arithmetic unit when multiple narrow-width operations are computed consecutively. Brooks *et al.* [1] showed that power dissipation can be reduced by gating of the upper part of narrow-width operands. For the SPECint95 and MediaBench benchmarks, the power reduction of an operand guarded integer unit was 54% and 58%, respectively, which accounts for a total power reduction of 5-6% for an entire datapath.

Narrow-width operands have also been used to increase instruction throughput, by computing several narrow-width operations in parallel on a full-width datapath. Loh [4] showed a 7% speedup for the SPECint2000 benchmarks by using a simple 64-bit ALU, which excluded the multiplier, in parallel with four

[1] There is also a potential static power overhead of inactive multipliers, but this could be alleviated by power gating.

simple 16-bit ALUs that share a 64-bit routing. Brooks *et al.* did a similar investigation, where they envisioned a 64-bit adder that could be separated into four 16-bit adders by severing the carry chain.

There have been several studies on operand guarding for multipliers. Huang *et al.* [2] introduced a two-dimensional operand guarding for array multipliers, resulting in a power dissipation that was only 33% of a conventional array multiplier. Han *et al.* [3] did a similar investigation on a 32-bit Wallace multiplier and were able to reduce the switching activity by 72% with the use of 16-bit operand guarding.

While there have been a lot of work on simple schemes for operand guarding, work that simultaneously considers multiplication throughput is more scarce. Achieving double throughput for a multiplier is not as straightforward as, for example, in an adder, where the carry chain can be cut at the appropriate place to achieve narrow-width additions. It is of course possible to use several multipliers, where at least two have narrow bitwidth, and let them share the same routing, as in the work of Loh, but such a scheme has several drawbacks: *i)* The total area of the multipliers would increase, since several multiplier units are used. *ii)* The use of several multipliers increases the fanout of the signals that drive the inputs of the multipliers. Higher fanout means longer delays and/or higher power dissipation. *iii)* There would be a need for multiplexers that connect the active multiplier(s) to the result route. These multiplexers would be in the critical path, increasing total delay as well as power dissipation. Work has been done to use 4:2-reduction stages to combine small tree multipliers into larger multipliers [5, 6]. This can be done in several stages, creating a larger multiplier out of smaller for each extra 4:2 reduction stage. The desired bitwidth of the multiplication is then obtained by using multiplexers. This technique introduces multiplexers in the critical path, which has a negative impact on the delay.

We present the twin-precision technique [7] that offers the same power reduction as operand guarding *and* the possibility of double-throughput multiplications. The twin-precision technique is an efficient way of achieving double throughput in a multiplier with low area overhead and with only a small delay penalty. We show how to apply the twin-precision technique on signed multi-

pliers based on the regular High Performance Multiplier (HPM) reduction tree. The two algorithms for signed multiplications that have been used are Baugh-Wooley and the popular modified-Booth algorithm.

To exemplify the practical value of the twin-precision technique, we have added a light-weight SIMD extension that utilizes a twin-precision multiplier to a simple five-stage processor (similar to a MIPS R2000 [8]). In comparison to the baseline processor, the modified processor has negligible delay and power overhead, but can still achieve significant application speedups and energy savings by utilizing its intrinsic SIMD capabilities.

This paper is organized as follows: First, we introduce the twin-precision technique in Sec. F.2. This is followed by a description of how the twin-precision technique can be applied to a Baugh-Wooley, Sec. F.3, and a modified-Booth multiplier, Sec. F.4. The evaluation setup and netlist generation is reviewed in Sec. F.5. Sec. F.6 and Sec. F.7 present the evaluation results for a 130-nm and a 65-nm process, respectively. Sec. F.8 presents the proportion of narrow-width (16-bit) multiplications in some typical embedded applications, while Sec. F.9 presents a case study. Finally, the paper is concluded in Sec. F.10.

F.2 Twin-Precision Fundamentals

For a first analysis of the twin-precision technique, the discussion will be based on an illustration of unsigned binary multiplication. In an unsigned binary multiplication each bit of one of the operands, called the multiplier, is multiplied with the second operand, called multiplicand (Eq. F.1). That way one row of partial products is generated. Each row of partial products is shifted according to the position of the bit of the multiplier, forming what is commonly called the partial-product array. Finally, partial products that are in the same column are summed together, forming the final result. An illustration of an 8-bit multiplication is shown in Fig. F.1.

$$p_{ij} = y_i x_j \tag{F.1}$$

Let us look at what happens when the precision of the operands is smaller than the multiplier we intend to use. In this case, the most significant bits of the

								y_7	y_6	y_5	y_4	y_3	y_2	y_1	y_0
								x_7	x_6	x_5	x_4	x_3	x_2	x_1	x_0
								p_{70}	p_{60}	p_{50}	p_{40}	p_{30}	p_{20}	p_{10}	p_{00}
							p_{71}	p_{61}	p_{51}	p_{41}	p_{31}	p_{21}	p_{11}	p_{01}	
						p_{72}	p_{62}	p_{52}	p_{42}	p_{32}	p_{22}	p_{12}	p_{02}		
					p_{73}	p_{63}	p_{53}	p_{43}	p_{33}	p_{23}	p_{13}	p_{03}			
				p_{74}	p_{64}	p_{54}	p_{44}	p_{34}	p_{24}	p_{14}	p_{04}				
			p_{75}	p_{65}	p_{55}	p_{45}	p_{35}	p_{25}	p_{15}	p_{05}					
		p_{76}	p_{66}	p_{56}	p_{46}	p_{36}	p_{26}	p_{16}	p_{06}						
	p_{77}	p_{67}	p_{57}	p_{47}	p_{37}	p_{27}	p_{17}	p_{07}							
s_{15}	s_{14}	s_{13}	s_{12}	s_{11}	s_{10}	s_9	s_8	s_7	s_6	s_5	s_4	s_3	s_2	s_1	s_0

Figure F.1: *Illustration of an unsigned 8-bit multiplication.*

operands will only contain zeros, thus large parts of the partial-product array will consist of zeros. Further, the summation of the most significant part of the partial-product array and the most significant bits of the final result will only consist of zeros. An illustration of an 8-bit multiplication, where the precision of the operands is four bits, is shown in Fig. F.2.

Fig. F.2 shows that large parts of the partial-product array only consist of zeros and are, thus, not contributing any useful information for the final result. What if these partial products could be utilized for a second, concurrent multiplication?

Since partial products of the same column are summed together, it would not be wise to use any of the partial products that are in the same column as the multiplication that is already computed. Looking closer at the 4-bit multiplication marked in white in Fig. F.2, one can also observe that the column at position S_7 should not be used either. This is because that column might have a carry from the active part of the partial-product array that will constitute the final S_7. Altogether this makes only the partial products in the most significant part of the partial-product array available for a second multiplication.

In order to be able to use the partial products in the most significant part, there has to be a way of setting their values. For this we can use the most signif-

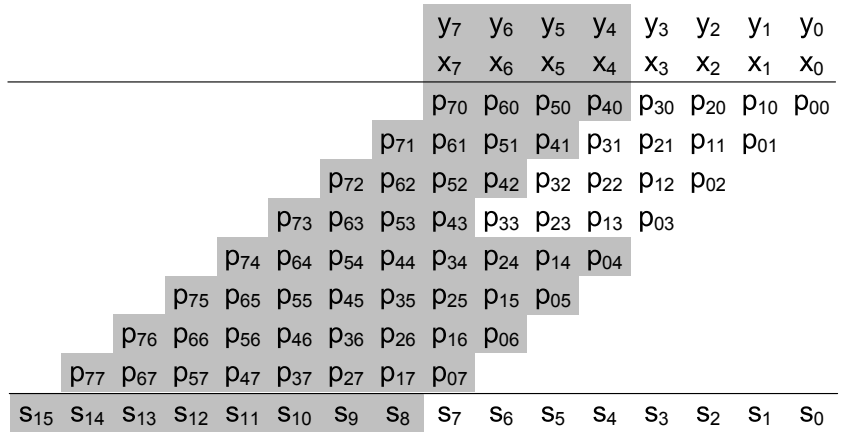


Figure F.2: *Illustration of an unsigned 8-bit multiplication, where the precision of the operands is smaller than the precision of the multiplication. Unused bits of operands and product, as well as unused partial products, are shown in gray.*

icant bits of the operands, since these are not carrying any useful information. If we are only looking at the upper half of the operands, the partial products generated from these bits are the ones shown in black in Fig. F.3. By setting the other partial products to zero, it is then possible to perform two multiplications within the same partial-product array, without changing the way the summation of the partial-product array is done. How the partial products, shown in gray, can be set to zero will be investigated in the implementation section later on.

Assume, for now, that there is a way of setting unwanted partial products to zero: Then it suddenly becomes possible to partition the multiplier into two smaller multipliers that can compute multiplications in parallel. In the above illustrations the two smaller multiplications have been chosen such that they are of equal size. This is not necessary for the technique to work. Any size of the two smaller multiplications can be chosen, as long as the precision of the two smaller multiplications together are equal or smaller than the full precision (N_{FULL}) of the multiplication, Eq. F.2. To be able to distinguish between the two smaller multiplications, they are referred to as the multiplication in the Least Significant Part (LSP) of the partial-product array with size N_{LSP}, shown

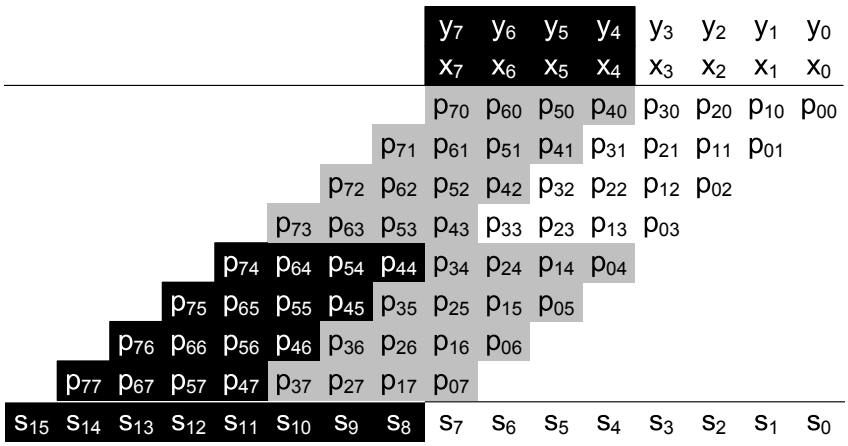


Figure F.3: *Illustration of an unsigned 8-bit multiplication, where a 4-bit multiplication, shown in white, is computed in parallel with a second 4-bit multiplication, shown in black.*

in white, and the multiplication in the Most Significant Part (MSP) with size N_{MSP}, shown in black.

$$N_{FULL} \geq N_{LSP} + N_{MSP} \tag{F.2}$$

It is functionally possible to partition the multiplier into even more multiplications. For example, it would be possible to partition a 64-bit multiplier into four 16-bit multiplications. Given a number K of low-precision multiplications, their total size needs to be smaller or equal to the full-precision multiplication.

$$N_{FULL} \geq \sum_{i=1}^{K} N_i \tag{F.3}$$

For the rest of this investigation, the precision of the two smaller multiplications will be equal and half the precision ($N/2$) of the full precision (N) of the multiplier.

F.2.1 A First Implementation

The basic operation of generating a partial product is that of a 1-bit multiplication using a 2-input AND gate, where one of the input signals is one bit of the multiplier and the second input signal is one bit of the multiplicand. The summation of the partial products can be done in many different ways, but for this investigation we are only interested in parallel multipliers that are based on 3:2 full adders[2]. For this first implementation an array of adders will be used because of its close resemblance to the previously used illustration of a multiplication, see Fig. F.4.

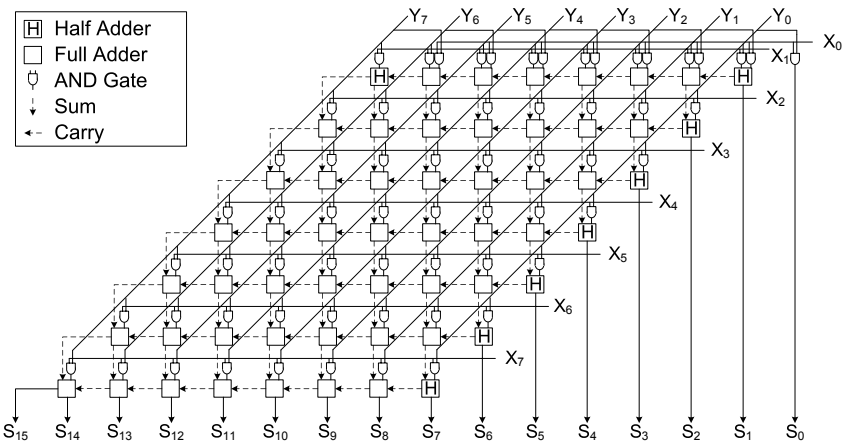


Figure F.4: *Block diagram of an unsigned 8-bit array multiplier.*

In the previous section we assumed that there is a way of setting unwanted partial products to zero. This is easily accomplished by changing the 2-input AND gate to a 3-input AND gate, where the extra input can be used for a control signal. Of course, only the AND gates of the partial products that have to be set to zero need to be changed to a 3-input version. During normal operation when a full-precision multiplication is executed the control signal is set to high, thus all partial products are generated as normal and the array of adders will sum them together and create the final result. When the control signal is set to

[2]Higher-radix compression *is* compatible with our strategy.

low the unwanted partial products will become zero. Since the summation of the partial products are not overlapping, there is no need to modify the array of adders. The array of adders will produce the result of the two multiplications in the upper and lower part of the final output. The block diagram of an 8-bit twin-precision array multiplier capable of computing two 4-bit multiplications is shown in Fig. F.5. The two multiplications have been colored in white and black to visualize what part of the adder array is used for what multiplication.

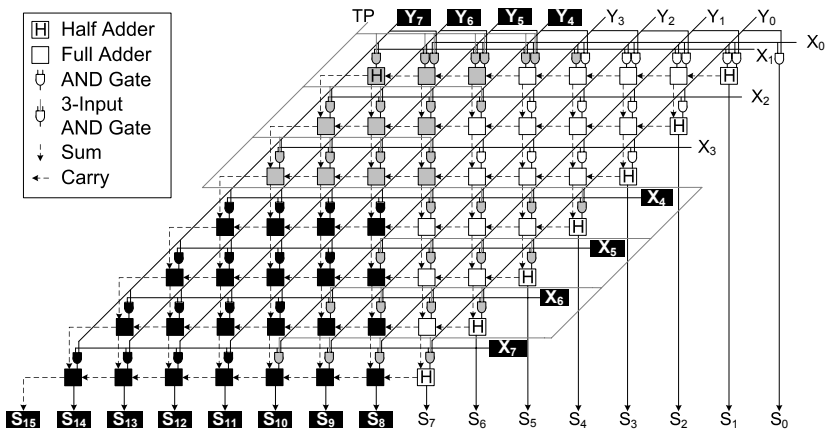


Figure F.5: *Block diagram of an unsigned 8-bit twin-precision array multiplier. The TP signal is used for controlling if one full-precision multiplication should be computed or two 4-bit multiplications should be computed in parallel.*

More flexibility might be wanted, like the possibility to compute a single narrow-width multiplication or two parallel narrow-width multiplications, within the same multiplier. This can be done by changing the 2-input AND gates for the partial product generation of the narrow-width multiplications as well. In the array multiplier in Fig. F.5, the AND gates for the 4-bit MSP multiplication, shown in black, can be changed to 3-input AND gates to which a second control signal can be added. Assuming the multiplier is divided into two equal parts, this modification makes it possible to either compute an N-bit, a single $N/2$-bit or two concurrent $N/2$-bit multiplications.

F.2.2 An HPM Implementation

The array multiplier in the previous section was only used to show the principle of the twin-precision technique. For high-speed and/or low-power implementations, a logarithmic reduction tree such as the TDM [9], Dadda [10], Wallace [11] or HPM [12] is preferred for summation of the partial products. A logarithmic reduction tree has the benefit of shorter logic depth. Further, a logarithmic tree has fewer glitches making it less power dissipating. A twin-precision implementation based on the regular HPM reduction tree is shown in Fig. F.6.

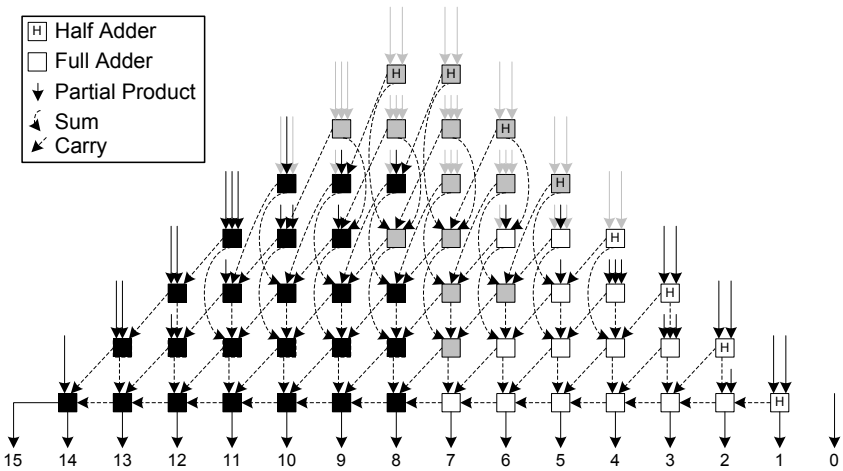


Figure F.6: *Block diagram of an unsigned 8-bit twin-precision multiplier that is based on the regular HPM reduction tree. A 4-bit multiplication, shown in white, can be computed in parallel with a second 4-bit multiplication, shown in black. For simplicity of the figure the* AND *gates for partial product generation is not shown and a ripple carry is used as final adder.*

F.2.3 The Final Adder

The ripple-carry adder as a final adder has been used to simplify the figures and the understanding of the twin-precision technique. However, the ripple-carry

adder can be exchanged for any other more suitable adder, without modifying the adder itself. The reason for this is that when operating on narrow-width operations, there will be no carry passing between the two multiplications and hence, no modification of the adder is needed.

F.3 A Baugh-Wooley Implementation

In the previous section, the concept of twin-precision was introduced by looking at an unsigned multiplication. However, for many applications signed multiplications are needed and consequently an unsigned multiplier is of limited use. In this section a twin-precision multiplier based on the Baugh-Wooley (BW) algorithm will be presented.

F.3.1 Algorithms for Baugh-Wooley

The BW algorithm [13] is a relative straightforward way of doing signed multiplications. Fig. F.7 illustrates the algorithm for an 8-bit case, where the partial-product array has been reorganized according to the scheme of Hatamian [14]. The creation of the reorganized partial-product array comprises three steps: *i)* the most significant partial product of the first $N-1$ rows and the last row of partial products except the most significant has to be negated, *ii)* a constant one is added to the Nth column, *iii)* the most significant bit (MSB) of the final result is negated.

F.3.2 Twin-Precision Using the Baugh-Wooley Algorithm

It is not as easy to deploy the twin-precision technique onto a BW multiplication as it is for the unsigned multiplication, where only parts of the partial products need to be set to zero. To be able to compute two signed $N/2$ multiplications, it is necessary to make a more sophisticated modification of the partial-product array. Fig. F.8 illustrates an 8-bit BW multiplication, in which two 4-bit multiplications have been depicted in white and black.

When comparing the illustration of Fig. F.7 with that of Fig. F.8 one can see that the only modification needed to compute the 4-bit multiplication in the

								y_7	y_6	y_5	y_4	y_3	y_2	y_1	y_0
								x_7	x_6	x_5	x_4	x_3	x_2	x_1	x_0
							1	$\overline{p_{70}}$	p_{60}	p_{50}	p_{40}	p_{30}	p_{20}	p_{10}	p_{00}
							$\overline{p_{71}}$	p_{61}	p_{51}	p_{41}	p_{31}	p_{21}	p_{11}	p_{01}	
						$\overline{p_{72}}$	p_{62}	p_{52}	p_{42}	p_{32}	p_{22}	p_{12}	p_{02}		
					$\overline{p_{73}}$	p_{63}	p_{53}	p_{43}	p_{33}	p_{23}	p_{13}	p_{03}			
				$\overline{p_{74}}$	p_{64}	p_{54}	p_{44}	p_{34}	p_{24}	p_{14}	p_{04}				
			$\overline{p_{75}}$	p_{65}	p_{55}	p_{45}	p_{35}	p_{25}	p_{15}	p_{05}					
		$\overline{p_{76}}$	p_{66}	p_{56}	p_{46}	p_{36}	p_{26}	p_{16}	p_{06}						
	p_{77}	$\overline{p_{67}}$	$\overline{p_{57}}$	$\overline{p_{47}}$	$\overline{p_{37}}$	$\overline{p_{27}}$	$\overline{p_{17}}$	$\overline{p_{07}}$							
$\overline{s_{15}}$	s_{14}	s_{13}	s_{12}	s_{11}	s_{10}	s_9	s_8	s_7	s_6	s_5	s_4	s_3	s_2	s_1	s_0

Figure F.7: *Illustration of a signed 8-bit multiplication, using the Baugh-Wooley algorithm.*

MSP of the array is an extra sign bit '1' in column S_{12}. For the 4-bit multiplication in the LSP of the array, there is a need for some more modifications. In the active partial-product array of the 4-bit LSP multiplication (shown in white), the most significant partial product of all rows, except the last, needs to be negated. For the last row it is the opposite, here all partial products, except the most significant, are negated. Also for this multiplication a sign bit '1' is needed, but this time in column S_4. Finally the MSB of the results needs to be negated to get the correct result of the two 4-bit multiplications.

To allow for the full-precision multiplication of size N to coexist with two multiplications of size $N/2$ in the same multiplier, it is necessary to modify the partial-product generation and the reduction tree. For the $N/2$-bit multiplication in the MSP of the array all that is needed is to add a control signal that can be set to high, when the $N/2$-bit multiplication is to be computed and to low, when the full precision N multiplication is to be computed. To compute the $N/2$-bit multiplication in the LSP of the array, certain partial products need to be negated. This can easily be accomplished by changing the 2-input AND gate that generates the partial product to a 2-input NAND gate followed by an XOR gate. The second input of the XOR gate can then be used to invert the output of

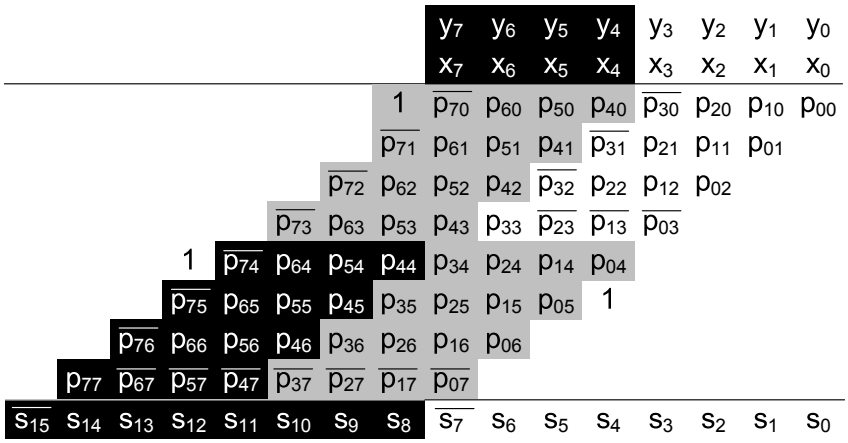


Figure F.8: *Illustration of a signed 8-bit multiplication, using the Baugh-Wooley algorithm, where one 4-bit multiplication, shown in white, is computed in parallel with a second 4-bit multiplication, shown in black.*

the NAND gate. When computing the $N/2$-bit LSP multiplication, the control input to the XOR gate is set to low making it work as a buffer. When computing a full-precision N multiplication the same signal is set to high making the XOR work as an inverter. Finally the MSB of the result needs to be negated and this can again be achieved by using an XOR gate together with an inverted version of the control signal for the XOR gates used in the partial-product generation. Setting unwanted partial products to zero can be done by 3-input AND gates as for the unsigned case.

Fig. F.9 shows an implementation of a twin-precision 8-bit BW multiplier. The modifications of the reduction tree compared to the unsigned 8-bit multiplier in Fig. F.6 consist of three things; *i)* the half adders in column 4 and 8 have been changed to full adders in order to fit the extra sign bits that are needed, *ii)* for the sign bit of the 4-bit MSP multiplication there is no half adder that can be changed in column 12, so here an extra half adder has been added, which makes it necessary to also add half adders for the following columns of higher precision, and *iii)* finally XOR gates have been added at the output of column 7 and 15 so that they can be inverted.

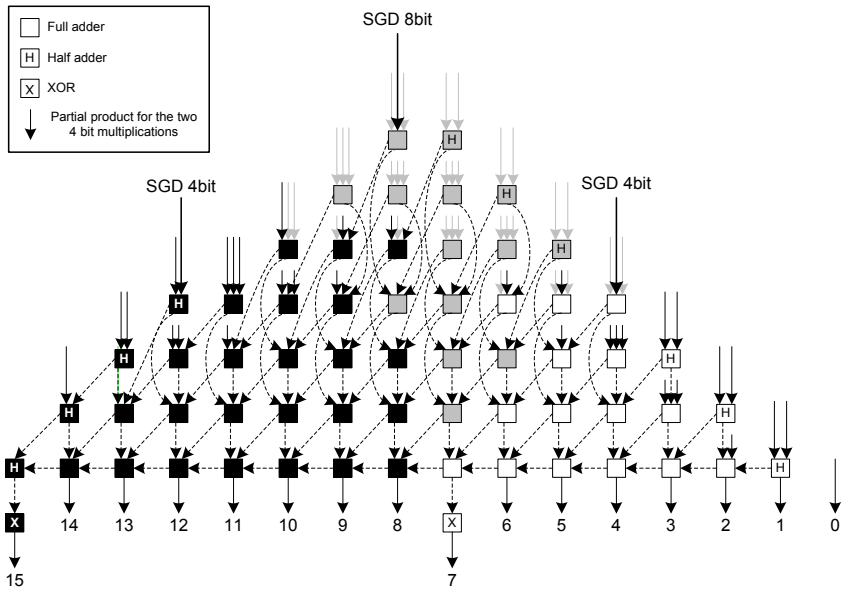


Figure F.9: *Block diagram of a signed 8-bit multiplication, using the Baugh-Wooley algorithm, where one 4-bit multiplication, shown in white, is computed in parallel with a second 4-bit multiplication, shown in black.*

The simplicity of the twin-precision BW implementation makes it easy to also compute unsigned multiplications. All that is needed is to set the control signals accordingly, such that none of the partial products are negated, the XOR gates are set to not negate the final result and all the sign bits are set to zero.

F.4 A Modified-Booth Implementation

Modified Booth (MB) is a popular and common algorithm for implementation of signed multipliers. MB is a more complicated algorithm than Baugh-Wooley (BW), but it has the advantage of only producing half the number of partial products. In this section a twin-precision multiplier based on the MB algorithm will be presented.

F.4.1 Algorithms for Modified Booth

The original Booth algorithm [15] is a way of coding the partial products generated during a $s = x \times y$ multiplication. This is done by considering two bits at a time of the multiplier, x, and coding them into $\{-2, -1, 0, 1, 2\}$. The encoded number is then multiplied with the multiplicand, y, into a row of recoded partial products. The number of recoded partial products is fewer than for a scheme with unrecoded partial products, which during implementation may translate into higher performance.

The drawback of the original Booth algorithm is that the number of generated partial products depends on the x-multiplier, which makes the Booth algorithm unsuitable for implementation in hardware. The modified-Booth (MB) algorithm [16] by MacSorley remedies this by looking at three bits at a time of the multiplier. Then we are guaranteed that only half the number of partial products will be generated, compared to a conventional partial product generation using 2-input AND gates. With a fixed number of partial products the MB algorithm is suitable for hardware implementation. Fig. F.10 shows which parts of the multiplier that are encoded and used to recode the multiplicand into a row of partial products.

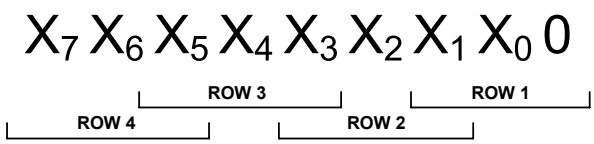


Figure F.10: *8-bit modified-Booth encoding.*

A MB multiplier works internally with two's complement representation of the partial products, in order to be able to multiply the encoded $\{-2, -1\}$ with the multiplicand. To avoid having to sign extend the rows of recoded partial products, the sign-extension prevention scheme presented by Fadavi-Ardekani [17] has been used. In two's complement representation, a change of sign includes the insertion of a '1' at the Least Significant Bit (LSB) position. To avoid getting an irregular partial-product array we draw on the idea of Yeh *et al.* [18], called modified partial-product array. The idea is to pre-compute

the impact on the two least significant positions of a row of recoded partial products by the insertion of a '1' during sign change. The pre-computation calculates the addition of the LSB with the potential '1', from which the sum is used as the new LSB for the row of recoded partial products. A potential carry from the pre-computation is inserted at the second least significant position. The pre-computation of the new LSB can be done according to Eq. F.4. The pre-computation of a potential carry is as given by Eq. F.5. Here Eq. F.5 is different from that presented by Yeh *et al.* [18].

$$p_{LSBi} = y_0(x_{2i-1} \oplus x_{2i}) \tag{F.4}$$

$$a_i = x_{2i+1}(\overline{x_{2i-1} + x_{2i}} + \overline{y_{LSB} + x_{2i}} + \overline{y_{LSB} + x_{2i-1}}) \tag{F.5}$$

An illustration of an 8-bit MB multiplication with the sign-extension prevention and modified partial-product array scheme can be seen in Fig. F.11.

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

Figure F.11: *Illustration of a signed 8-bit multiplication using the modified-Booth algorithm.*

F.4.2 Twin-Precision Using the modified-Booth Algorithm

Implementing the twin-precision scheme on the MB algorithm is not as straightforward as for the BW implementation (Sec. F.3). It is not possible to take the partial products from the full-precision MB multiplication and use only the partial products that are of interest for the narrow-width MB multiplications. The

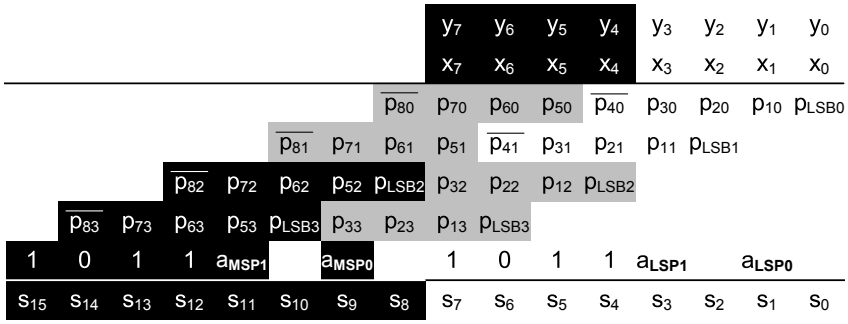


Figure F.12: *Illustration of a signed 8-bit multiplication, using the modified-Booth algorithm, where one 4-bit multiplication, shown in white, is computed in parallel with a second 4-bit multiplication, shown in black.*

reason for this is that all partial products are not computed the same way and there exist several special cases that need to be handled.

By comparing the two multiplications in Fig. F.11 and Fig. F.12 (for which we used $N = 8$), we can see what needs to be handled in order to switch between the different modes of operation:

- The partial products that are denoted p_{40} and p_{41} during normal 8-bit multiplication (Fig. F.11) need to define partial products that are used to prevent sign extension in the narrow-width 4-bit multiplication in the LSP (Fig. F.12).

- The partial-products that are denoted p_{42} and p_{43} during normal 8-bit multiplication need to define p_{LSB2} and p_{LSB3} for the narrow-width 4-bit multiplication in the MSP.

- The a_{MSP0} and a_{MSP1} that are needed for the multiplication in the MSP have to be added.

- The pattern of 1's and 0's for the normal 8-bit multiplication cannot be used in narrow-width mode. For the two 4-bit multiplications, we need two shorter patterns of 1's and 0's.

The implementation of the MB twin-precision multiplication does not call for any significant changes to the reduction tree of a conventional MB multiplier. When comparing the multiplications in Fig. F.11 and Fig. F.12, we can see that the position of the signals in the lowest row is the only difference that has an impact on the reduction tree. This means that there is a need for an extra input in two of the columns ($N/2$ and $3N/2$) compared to the conventional MB multiplier; this requires two extra half adders in the reduction tree.

The biggest difference between a conventional MB multiplier and a twin-precision MB multiplier is the generation of inputs to the reduction tree. To switch between modes of operation, logic is added to the recoder to allow for generation of the partial products needed, for sign-extension prevention as well as p_{LSBi}, which are needed for $N/2$-bit multiplications in the LSP and the MSP, respectively. There is also a need for multiplexers that, depending on the mode of operation, select the appropriate signal as input to the reduction tree. Further, partial products that are not being used during the computation of $N/2$-bit multiplications have to be set to zero in order to not corrupt the computation. An example of an 8-bit MB twin-precision multiplier is shown in Fig. F.13.

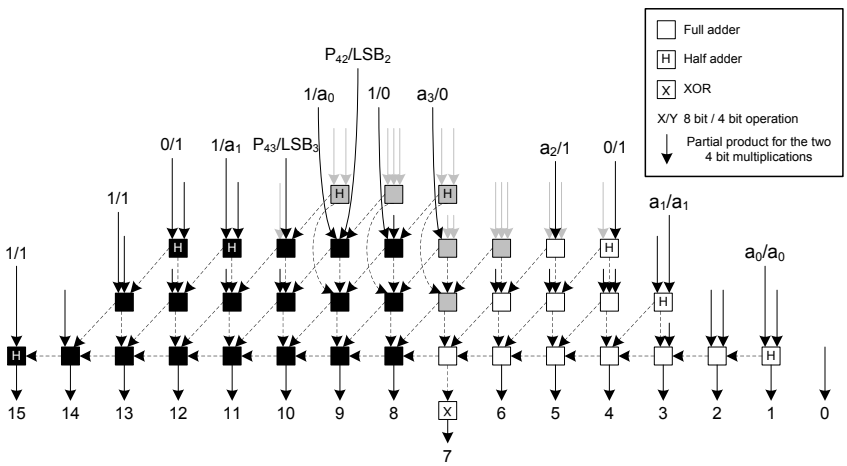
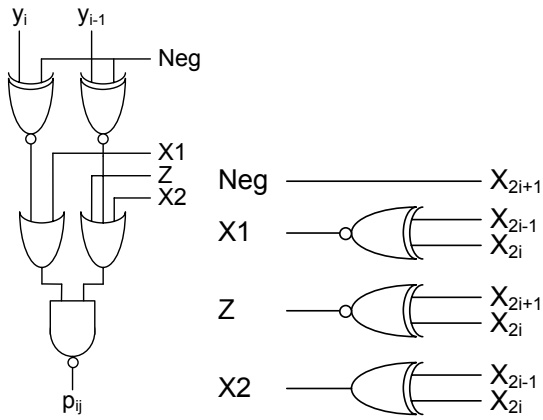


Figure F.13: *Block diagram of a signed 8-bit multiplication using the modified-Booth algorithm, where a 4-bit multiplication, shown in white, is computed in parallel with a second 4-bit multiplication, shown in black.*

The recode logic can be implemented in many different ways. For this implementation we have chosen the recoding scheme presented by Yeh *et al.* [18]. The circuits for encoding and decoding are shown in Fig. F.14.



(a) Decode (b) Encode

Figure F.14: *Encode and decode circuit for modified Booth.*

For the partial products that need to be set to zero, the output of the decode circuit can be connected to the input of a 2-input AND gate. The second input of the AND gate can then be used as a control signal, as in the case of for the unsigned and BW implementations. This is a straightforward method, and it is possible to construct more efficient solutions for setting the partial product to zero, if the option of custom designed cells is available. This is not necessary for correct operation, but it could increase the speed of the decode circuit.

For correct operation the input to the encoder for the first row in the $N/2$-bit MSP multiplication has to be set to zero, instead of using $x_{N/2-1}$ as its input. An example of the encoding scheme for two 4-bit multiplications can be seen in Fig. F.15.

In order to separate the two different $N/2$-bit multiplications, such that the multiplication in the LSP does not interfere with the multiplication in the MSP, we need to consider another other issue. By looking at the pattern of 1's and 0's that is used for sign-extension prevention, we see that the most significant '1'

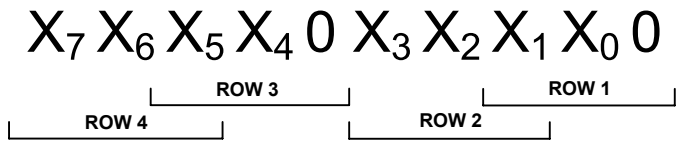


Figure F.15: *Encoding scheme for two 4-bit multiplications.*

is only used to invert the final s_{2N-1}-bit. However, the carry that potentially could be generated by this extra '1' is not of interest for the final result. If the most significant '1' for the multiplication in the LSP would be inserted into the reduction tree it would mean that a carry could be generated. This potential carry would propagate into the multiplication in the MSP and corrupt the result. To avoid inserting the most significant '1', an XOR gate is added after the final adder allowing the MSB of the $N/2$-bit LSP multiplication to be negated, which is the sole purpose of the most significant '1'.

F.5 Netlist Generation and Evaluation Setup

To evaluate the efficiency of the twin-precision technique a multiplier generator was written [19]. The generator is capable of generating VHDL descriptions of conventional Baugh-Wooley (BW) and modified-Booth (MB) multipliers as well as twin-precision versions of both of these, according to the schemes presented in the previous sections (F.3 and F.4)[3]. The VHDL generator was verified to generate correct multipliers of sizes up to 16 bits by simulating all possible input patterns and verifying the result using Cadence NC-VHDL [21]. For multipliers larger than 16 bits, the functionality was verified by feeding the multipliers with one million random input vectors and verifying the result.

The VHDL descriptions were synthesized using Synopsys Design Compiler [22] together with a commercially available 1.2 V 130-nm process. The synthesized netlist was taken through place and route using Cadence Encounter [23]. To create a common interface to the multipliers and ease the usage of the EDA tools that do not interact efficiently with purely combinatorial circuits,

[3] A Kogge-Stone [20] adder was chosen as final adder for all types of multipliers.

registers were inserted at the primary multiplier inputs and outputs. Delay estimations are based on RC extracted data from the placed and routed netlists, while power estimates are obtained after applying value change dump (VCD) statistics of simulations of 10,000 random input vectors for each of the possible operational modes of the multiplier. Delay, power, and area figures include the input and output registers on the multipliers and are given for the worst-case corner at 1.08 V.

The timing constraints for place-and-route were set systematically through extensive use of scripting; the goal was to push the limits of the timing for each generated VHDL description[4]. This strategy indeed achieves a fast implementation, however, the power dissipation becomes high, due to excessive buffering and resizing of gates. Therefore, the timing was relaxed with 100, 300, and 600 ps, relative to the fastest achieved timing for each implementation. Power estimates were subsequently obtained for each timing constraint.

F.5.1 Synthesis and Layout of Baugh-Wooley Netlists

Our layout and timing analysis showed that the simplicity of the BW implementation comes with an added benefit in that it does not create high fanout signals. The signals with highest fanout in the BW case are the input signals, which are connected to the input of N 2-input AND gates for a multiplier of size N. This creates a reasonable fanout of the input signals for multipliers up to at least 64 bits, without degrading the performance significantly. The mapping of the BW multiplier code to gate-level netlist during synthesis was only constrained in the way that half and full adders of the reduction tree were mapped to their respective minimum-size cells and the map effort in Design Compiler was set to high.

[4]For the twin-precision implementations, the control signals for switching mode between N, 1xN/2, and 2xN/2 were not included when estimating timing. However, these control signals do not have any significant effect on the timing in a practical case, as will be shown in Sec. F.9.

F.5.2 Synthesis and Layout of Modified-Booth Netlists

The first attempt at creating layouts for the VHDL generated MB multipliers exposed a big problem with high fanout signals. This can easily be realized by investigating the encode and decode circuits from Fig. F.14. As can be seen, the x_{2i+1} input signal goes straight through the encoder as the NEG signal and drives two XNOR gates for each partial product of a single row. This means that the fanout of half of the primary x-inputs is at least $2N$ XNOR gates for a multiplier of size N. Further, the encoder outputs X1, Z, and X2 drive N decoders creating a need for large XOR and XNOR gates in the encode stage, which increases the fanout of the x-inputs even more.

To deal with the fanout problem, the fanout of X1, Z and X2 was reduced by instantiating multiple encoders for each row of partial products. To limit the fanout of the x_{2i+1} signal, an inverter buffer was inserted to drive the NEG signal. Finally, minimum-size inverters were added to x_{i-1}, x_i, and x_{i+1}, inputs of the encoder, thus, the fanout of primary x-inputs is reduced. The new encode circuit can be viewed in Fig. F.16.

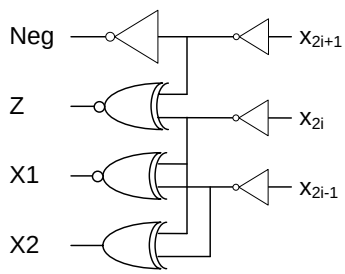


Figure F.16: *Buffered encode circuit for fanout reduction.*

The mapping of the MB multiplier code to gate-level netlist was constrained in such way that Design Compiler could not remove the minimum-size inverters of the new encoder, Fig. F.16. In all other respects the synthesized netlist of the MB multiplier was constrained in the same way as the BW netlist. In other words, half and full adders of the reduction tree were mapped to their respective cells of minimum size from the cell library and the map effort was set to high.

F.6 Results and Discussion

One of the goals of the twin-precision technique is to keep the performance degradation of the multiplier's full-precision operation at a minimum. To compare twin-precision implementations against conventional Baugh-Wooley (BW) and modified Booth multipliers (MB), each multiplier was taken through the EDA flow outlined previously.

Fig. F.17 shows the delay and power dissipation for conventional and twin-precision enabled BW and MB multipliers of size 16, 32, and 48 bits. The figure shows how the power changes as the timing requirements are relaxed with 100, 300, and 600 ps. A clear trend is that a BW implementation is more power efficient than a MB implementation. However, a MB implementation can, in some cases, exhibit higher maximum speed.

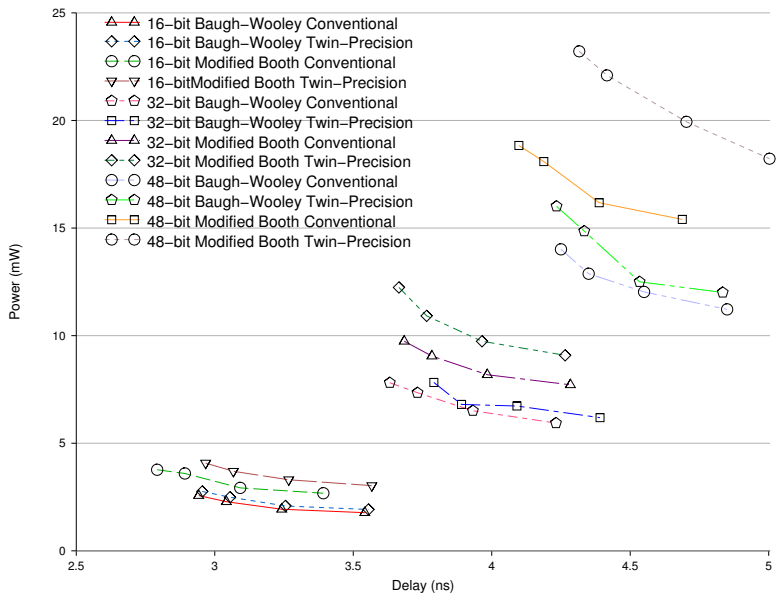


Figure F.17: *The power dissipation for different timing constraints applied to conventional and twin-precision Baugh-Wooley and modified-Booth multipliers of size 16, 32, and 48 bits.*

The result of the comparison of the twin-precision implementations with their conventional counterparts is that a twin-precision implementation of Baugh-Wooley performs equal in terms of delay for the 16 and 48-bit case and is only 160 ps slower for the 32-bit case. When we consider power, the twin-precision implementation dissipates 8%, 5%, and 6% more power than a conventional 16, 32, and 48-bit BW implementation.

From our results it is clear that the complexity of the MB recoding circuit makes it difficult to efficiently update an MB implementation into a twin-precision multiplier. The MB twin-precision implementation has the poorest performance, both in terms of delay and power, of the four compared implementation choices.

F.6.1 Delay

It is clear that the delay is not greatly degraded by the introduction of the twin-precision technique. Fig. F.18 shows the minimum delay for conventional and twin-precision enabled BW and MB multipliers. As can be seen the difference in timing is not large, ranging from 0 to 150 ps in difference when comparing the two BW multipliers[5]. The figure shows that a twin-precision enabled BW implementation is the better choice compared to a twin-precision enabled MB implementation.

F.6.2 Energy per Operation

One of the reasons to choose a twin-precision implementation instead of a conventional multiplier implementation is that energy can be saved[6] by reducing the precision of the multiplier, when operating on narrow-width operands. To compute the energy-per-operation, we extracted delay and power values (Fig. F.17) for various sizes of each multiplier design. We then computed the energy, as $energy = delay \cdot power$, for each delay and power pair and from these we chose the smallest energy for each multiplier design and size. The result from this investigation is shown in Fig. F.19.

[5] Not counting the 64-bit implementation.

[6] The other reason is to improve computational throughput.

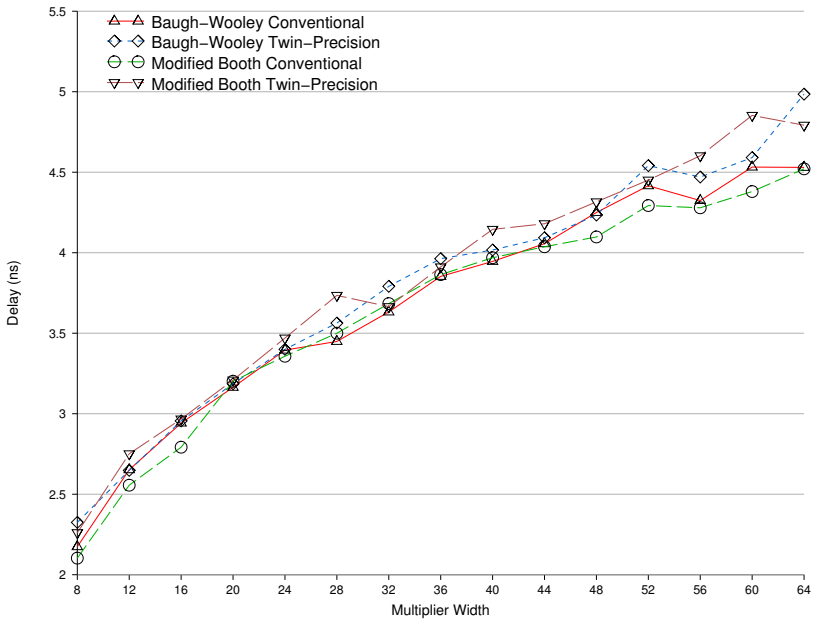


Figure F.18: *The delay for conventional and twin-precision Baugh-Wooley and modified-Booth multipliers of sizes from 8 to 64 bits.*

The twin-precision multiplier on average has 10% higher energy-per-operation, when operating on full-precision N-bit data, than a conventional BW implementation. However, for half of the implementations the energy is on average only 5% higher. The main benefit, in terms of energy, of the twin-precision technique is when operating the multiplier in narrow-width mode. Fig. F.19 also shows the energy-per-operation for a single multiplication of size $N/2$-bit when computing two $N/2$ multiplications concurrently. When working with narrow-width data the twin-precision multiplier has on average 59% lower energy dissipation than the conventional BW multiplier.

From Fig. F.19 we can establish that a MB implementation is not energy efficient, whether it is a conventional or twin-precision implementation.

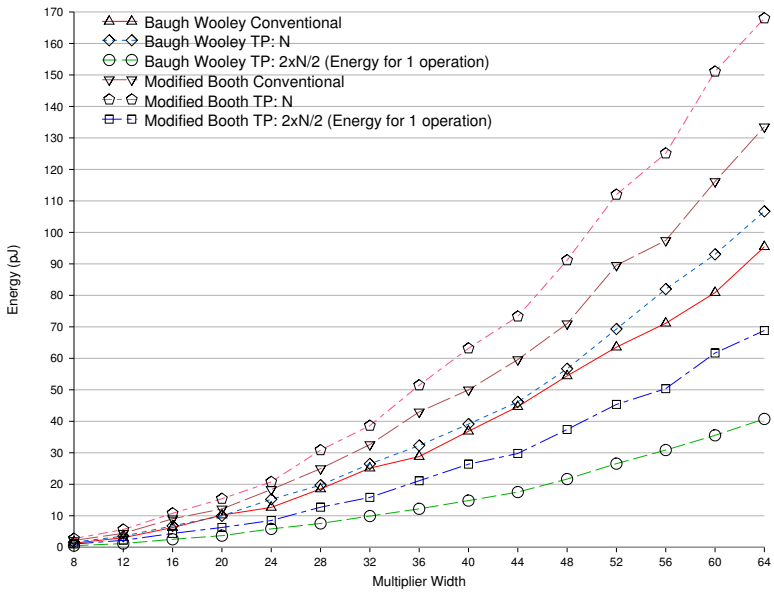


Figure F.19: *The energy-per-operation for conventional and twin-precision Baugh-Wooley and modified-Booth multipliers of sizes from 8 to 64 bits.*

F.6.3 Area

Fig. F.20 shows that a twin-precision multiplier requires slightly more area than its conventional counterpart. However, the twin-precision implementation based on the BW algorithm is smaller than the commonly used conventional MB implementation.

F.6.4 Power Reduction by the Use of Power Gating

Even though the reduction in power by the use of the twin-precision technique is substantial, the power overhead of an $N/2$-bit multiplication compared to a conventional multiplier of size $N/2$ is high. For the 32-bit twin-precision BW implementation operating on a single 16-bit data, the overhead is 55% in our 130-nm technology evaluation. This overhead is reduced to 15% per operation

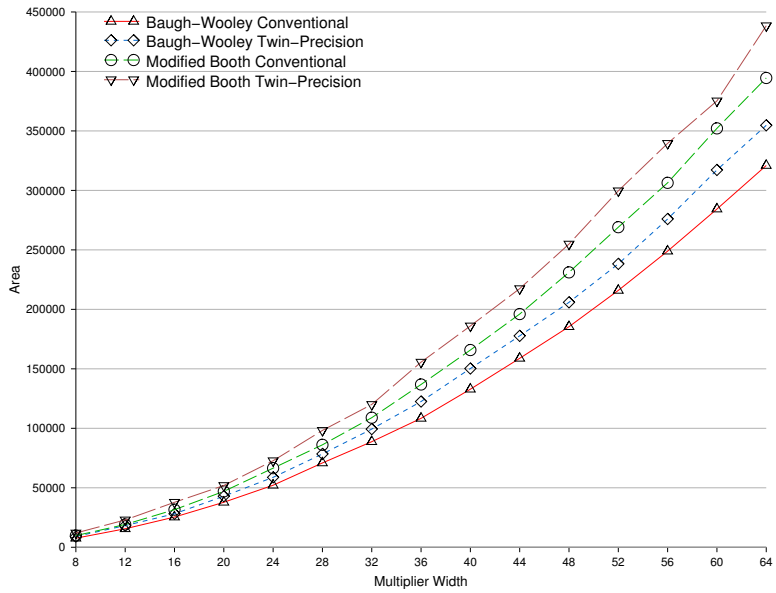


Figure F.20: *The area for conventional and twin-precision Baugh-Wooley and modified-Booth multipliers of sizes from 8 to 64 bits.*

if the twin-precisions multiplier is operating on two 16-bit data concurrently. The reduction in power overhead is due to less logic of the multiplier being idle and leaking, and that the static power dissipation of the idle logic is amortized over two 16-bit operations.

To reduce the power dissipation overhead of the narrow-width mode it is possible to apply power gating, instead of only setting the partial products to zero by the use of 3-input AND gates. Power gating involves adding power cut-off transistors to the logic cells that should be gated off when idle, effectively isolating the cell from the power supply when the cell is idle. The output of the power-gated cells will, thus, be undefined and to avoid these signals to interfere with the computation in the active part of the multiplier, they need to be forced to zero. This can be done by adding a transistor, on the output of the cells, that is connected to ground. Thus, when the power supply is gated off, the transistor

is activated making the output go low. This calls for a custom layout though, since such cells, where the output is forced to zero when power gated, can be hard to find in standard-cell libraries.

To be able to apply power gating to different areas of the partial-product generation, reduction tree, and the final adder, a suitable power grid is needed. Fig. F.21 shows an example of a power grid for an 8-bit twin-precision BW multiplier.

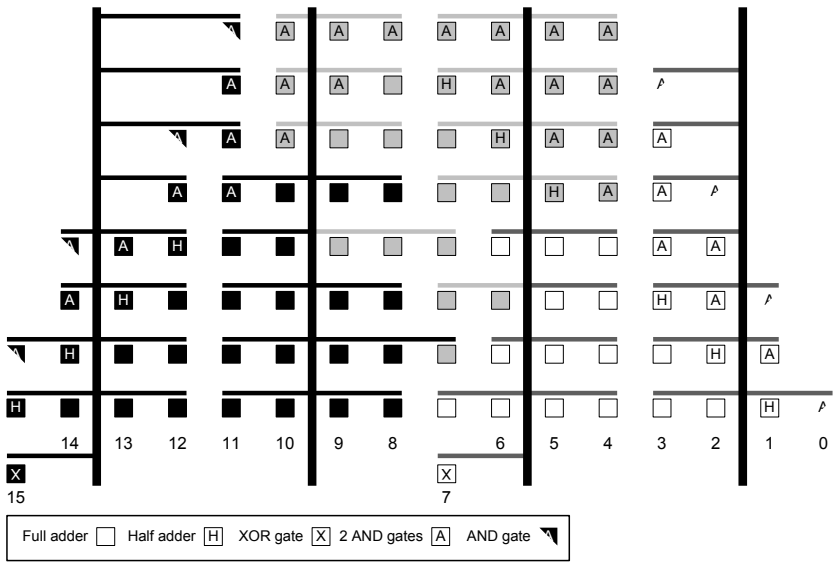


Figure F.21: *Power grid for an 8-bit twin-precision Baugh-Wooley implementation.*

Previous work of ours [24] showed that by using precision-dependent power-gating based on the SCCMOS technique, the power overhead for a 16-bit BW multiplier can be reduced with 53% at a delay penalty of less than 3%. The results from that work cannot be directly applied to these investigations, because both process technology and implementation abstraction level are different[7]. However, that work indicates the potential in using precision-dependent power-gating for reducing the power dissipation even further for twin-precision multipliers.

[7]The process technologies are both 130-nm, however the vendors are different.

F.7 Implementation in a 65-nm Process

We also used a commercial 65-nm process (low power cell library with standard V_T) to extend our evaluation. We implemented four different 32-bit multipliers; a conventional Baugh-Wooley (BW), a conventional modified-Booth (MB), a twin-precision BW and a twin-precision MB multiplier.

For this 65-nm EDA flow, Cadence First Encounter [25] was used for synthesis, placement, and routing. For the 65-nm process, we were not able to do the same fanout investigations and buffer insertions for the MB multipliers, as described for the 130-nm process. All VHDL descriptions were taken through the EDA flow using the same script and setup. The synthesis was restricted to use only full and half adder cells. However, the technology mapping algorithm is able to improve the designated gate structure, and thus the adder functionality in some cases is implemented as a set of elementary standard cells. Timing and power estimates are obtained for the worst-case 125 °C corner at 1.1 V. The power dissipation was estimated using value change dump (VCD) data from simulations with 10,000 random input vectors; that is, just as in the case of the 130-nm evaluation. The timing we present includes switching of control signals for changing operational mode of the twin-precision multipliers.

Table F.1: *Delay, power, and area estimates for 32-bit conventional and twin-precision enabled Baugh-Wooley and modified-Booth multipliers in a 65-nm process.*

	Delay (ns)	Power (mW)	Area (kμm^2)
Baugh Conv.	2.59 (100%)	23.4 (100%)	48 (100%)
Baugh 1x32-bit	2.80 (108%)	24.2 (103%)	53 (111%)
Baugh 1x16-bit	-	8.8 (38%)	-
Baugh 2x16-bit	-	14.5 (62%)	-
Booth Conv.	2.50 (97%)	37.5 (160%)	52 (108%)
Booth 1x32-bit	2.68 (103%)	38.0 (162%)	53 (111%)
Booth 1x16-bit	-	20.2 (86%)	-
Booth 2x16-bit	-	27.0 (115%)	-

The results from the placed and routed 65-nm 32-bit conventional and twin-precision enabled BW and MB multipliers can be found in Table F.1. The table shows the minimum delay and area for the four different multiplier implementations, as well as the power dissipation for the conventional multipliers and for the three different operational modes of the twin-precision multipliers.

The twin-precision implementation is about 200 ps slower than its conventional counterpart. Further, the MB implementation is slightly faster than the corresponding BW implementation. The twin-precision BW implementation occupies 11% more area than a conventional BW implementation, while for the twin-precision MB implementation the increase in area is only 3%, compared to the conventional MB.

More interestingly, there is a large power reduction when a 32-bit twin-precision multiplier operates on 16-bit data. The power dissipation is reduced with more than 60%, when computing one 16-bit multiplication with the twin-precision enabled BW multiplier, compared to a conventional BW multiplier. If an application is adopted to compute two 16-bit multiplications in parallel, the reduction in power dissipation per operation is improved further and would be close to 70%.

We can observe similar improvements for the twin-precision enabled MB multiplier, but the generally high power dissipation makes MB a poor choice for low-power implementations.

Table F.2: *Delay, power, and area estimates for 32-bit conventional and twin-precision enabled Baugh-Wooley and modified-Booth multipliers in a 130-nm process.*

	Delay (ns)	Power (mW)	Area (kμm^2)
Baugh Conv.	3.63 (100%)	7.8 (100%)	89 (100%)
Baugh 1x32-bit	3.99 (110%)	7.3 (94%)	99 (111%)
Baugh 1x16-bit	-	4.0 (51%)	-
Baugh 2x16-bit	-	5.5 (71%)	-
Booth Conv.	3.68 (101%)	9.7 (124%)	109 (122%)
Booth 1x32-bit	3.75 (103%)	11.1 (142%)	120 (135%)
Booth 1x16-bit	-	7.2 (92%)	-
Booth 2x16-bit	-	9.3 (119%)	-

For reference, Table F.2 shows the same data for the 130-nm process technology as Table F.1 shows for the 65-nm process technology. We notice that the timing and area for the 32-bit conventional and twin-precision enabled BW implementation show similar results for the two different technologies. Considering power we can see that the overhead for the twin-precision multiplier, when operating on 32-bit data, is higher for the 65-nm technology. However, the reduction in power when operating on 16-bit data is significantly larger for the 65-nm process than for the older 130-nm process.

When comparing the MB implementations for the two different process technologies, the 65-nm process offers large improvements for the timing and area values. The BW implementations do not gain as much from scaling. However, the power dissipation of the MB implementations drastically increases relative a BW implementation when moving to the newer process.

F.8 Workload Characterization

The SimpleScalar [26] (Version 3.0d) instruction-set simulator has been used to simulate ten different benchmarks from the EEMBC [27] TeleBench and ConsumerBench benchmark suites. This was done in order to get realistic data of the amount of narrow-width (16-bit) multiplications in typical embedded workloads. The sim-fast version of the SimpleScalar simulator was modified such that the result for each executed multiplication is monitored: If a results is small enough to be represented using only 16 bits, a counter variable is incremented. Fig. F.22 shows the result from the simulations, with the number of narrow-width multiplications shown as the percentage of the total number of executed multiplications. The figure clearly shows that many of the multiplications being performed could be represented using only 16-bits. On average, for all the benchmarks and their different data sets, the fraction of narrow-width operations is 52% of all executed multiplications.

The way the multiplications of the benchmarks were monitored assures that the result from the multiplier can indeed be fed, without any truncation, directly to another unit operating on narrow-width operands, since the result is only 16-bit wide. However, this gives a pessimistic value of the number of narrow-

width multiplications that really exists in the benchmarks, since only the input operands need to be explicitly expressed using 16 bits. Take for example the FFT benchmark, which works with 16-bit data: In the FFT benchmark simulation 25-40% of the multiplications were found to be of narrow-width type, depending on the input data. However, the FFT benchmark works only with 16-bit data, which has the consequence that the result of the multiplication is right-shifted 15 bits. This inherent truncation makes it possible to compute all multiplications in narrow-width mode, as will be shown in Sec. F.9.

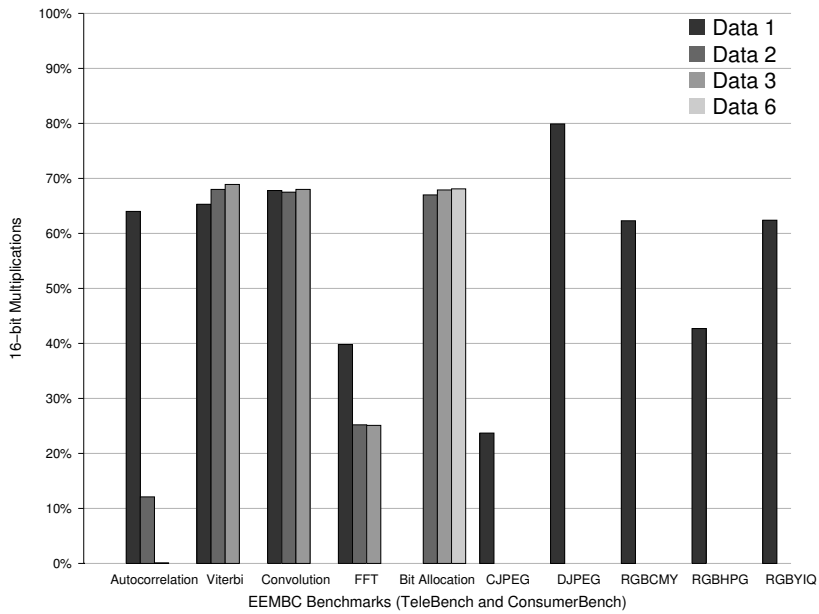


Figure F.22: *Fraction of narrow-width (16-bit) multiplications in the EEMBC TeleBench and ConsumerBench suites.*

From the data given in Tables F.1 and F.2, it is possible to calculate the energy dissipation when performing multiplications with a certain fraction of 16-bit operands. Fig. F.23 shows the energy dissipation for twin-precision Baugh-Wooley (BW) multipliers, as the fraction of signed 16-bit operands is varied.

The energy is shown relative that of a conventional (signed) BW multiplier that dissipates roughly the same energy regardless of what input operand bitwidth is used.

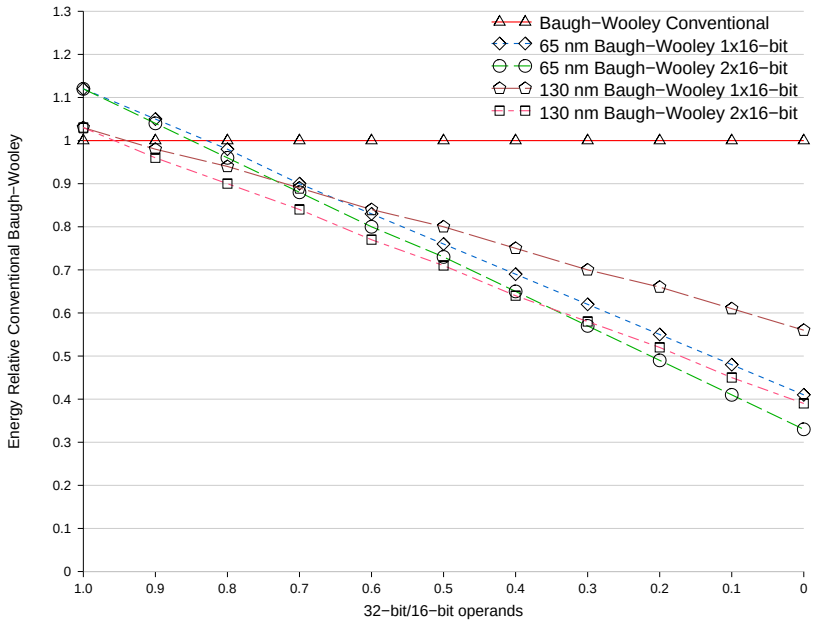


Figure F.23: *Energy dissipation when varying the relation between 32-bit and 16-bit operands for twin-precision Baugh-Wooley multipliers in the 65-nm and 130-nm processes, relative their respective conventional multiplier.*

In the 65-nm process, for a twin-precision BW multiplier to have equal to or less energy dissipation than a conventional BW multiplier, at least 15-18% of the operations have to be executed in 16-bit mode. For the 130-nm process, the same figure is only 5-7%. However, as the fraction of 16-bit operations increases, further to the right in the figure, the benefit of utilizing a twin-precision multiplier increases more rapidly for the 65-nm process.

The energy dissipation when performing the multiplications for the ten chosen benchmarks can on average be reduced with 25-28% for the 65-nm im-

plementation and 21-30% for the 130-nm implementation, if the 1x16-bit and 2x16-bit modes would be utilized by the benchmarks. The energy reduction from using the twin-precision technique is, thus, similar across the entire set of workloads. Specifically an energy reduction of about 25% can be expected for the multiplier unit.

F.9 SIMD Multiplier Extension, a Case Study

Single Instruction Multiple Data (SIMD) is a way of exploiting data parallelism in an application and is commonly used in high-end and embedded processors. SIMD instructions are commonly supported by a specialized datapath unit or coprocessor [28–30] that requires additional hardware units or as an intrinsic part of the standard datapath [31–34] that commonly lack support for SIMD integer multiplications.

The twin-precision technique makes it possible to support SIMD multiplications without the need to add another costly multiplier unit. A designer can apply the twin-precision technique, at a very small cost, on the conventional multiplier of a processor. This allows for SIMD multiplier functionality in processors, where it is normally not considered.

To evaluate this possibility, we have chosen to modify a processor that is inspired by the MIPS R2000 [8]. The Arithmetic Logic Unit (ALU) and the multiplier are modified such that the processor not only can perform 32-bit operations, but such that it also has support for performing operations on packed data. Here, a packed data item represents the concatenation of two 16-bit data items; we henceforth refer to this as *a packed 16-bit data*. A schematic of the implemented datapath is shown in Fig. F.24.

F.9.1 Instruction Set Extension

New instructions need to be augmented to the basic instruction set, in order to allow the processor to support 16-bit SIMD operations. But to reduce the impact on the processor cycle time, we add as few instructions as possible. Logical operations are performed bit-wise, thus, their functionality is the same

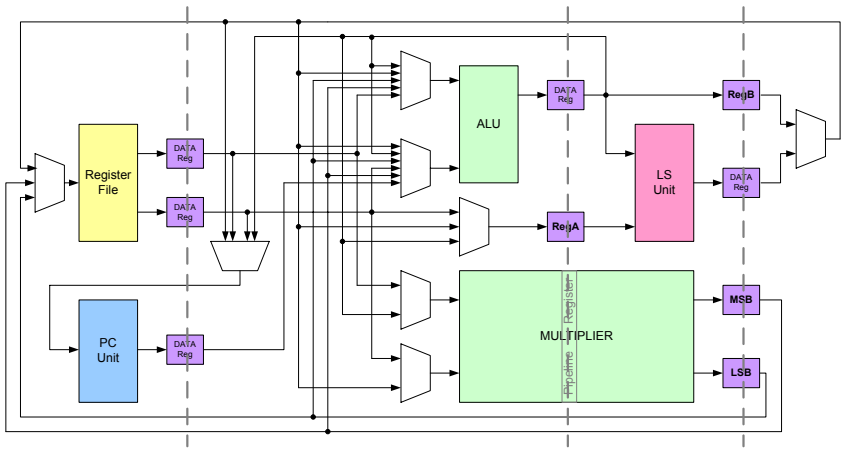


Figure F.24: *Schematic view of the implemented processor datapath.*

regardless if the data is 32-bit wide or if it is packed 16-bit data. Therefore, there is no need to modify or add any instructions for logical operations. To support arithmetic operations on packed 16-bit data, three new instructions are added: Addition (PADD), subtraction (PSUB), and multiplication (PMULT). In order to limit the instruction growth's impact on performance, specialized instructions for shift, compare and branch are not added.

To ease conversion from 32-bit to packed 16-bit data, an instruction (PACK) is added: As illustrated in Fig. F.25, PACK takes two 32-bit data items, and generates a packed 16-bit data item out of each respective lower 16 bits.

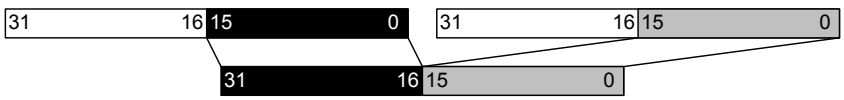


Figure F.25: *Two 32-bit data items are converted to one packed 16-bit item.*

The conversion of a packed 16-bit data item into two 32-bit data items is partly supported by existing instructions: *i)* The 16 higher bits of the packed 16-bit data item are extracted through a shift of 16 positions to the right. This can either be done with the conventional Shift Right Logic (SRL) or Shift Right

Arithmetic (SRA) instruction to get an unsigned or signed 32-bit data item. *ii)* For the 16 lower bits, the upper half of the packed 16-bit data is truncated (for unsigned representation) or it needs to represent a sign extension (for signed representation). A truncation can be done by a normal AND instruction, and a bit mask with '0' for the 16 higher bits and '1' for the 16 lower bits. However, the sign extension of the lower 16 bits can not be performed with a single instruction from the conventional instruction set of the processor. An instruction (SIGN16) that sign extends the lower 16 bits is therefore included.

Table F.3 lists the five instructions that we add to the basic instruction set and the functional units that subsequently need to be modified, to provide the hardware support.

Table F.3: *Added Instructions to the Processor Instruction Set*

Instruction	Functional Unit
PADD	ALU
PSUB	ALU
PACK	ALU
SIGN16	ALU
PMULT	Multiplier

F.9.2 Arithmetic Logic Unit

The adder in the Arithmetic Logic Unit (ALU) needs to be explicitly designed to support the PADD and PSUB instructions. Our implementation is based on a Sklansky prefix adder [35], an adder type which is known to represent a good trade-off between performance and area. More importantly, for this work, it has a SIMD-friendly structure that inherently offers the twin-precision capability for addition: By only cutting the carry chain between the lower and the higher 16 bits of the adder, a 32-bit Sklansky adder can also support the computation of two 16-bit additions in parallel. The ALU part of our light-weight SIMD extension relies on the insertion of a single AND-gate (A) in the carry-propagation path and the connection of one of its inputs to a control

signal (T), see Fig. F.26. If control signal T is high, the adder will operate as a conventional 32-bit adder. On the other hand, if T is low, the carry propagation path is broken, and the adder will compute two 16-bit additions instead.

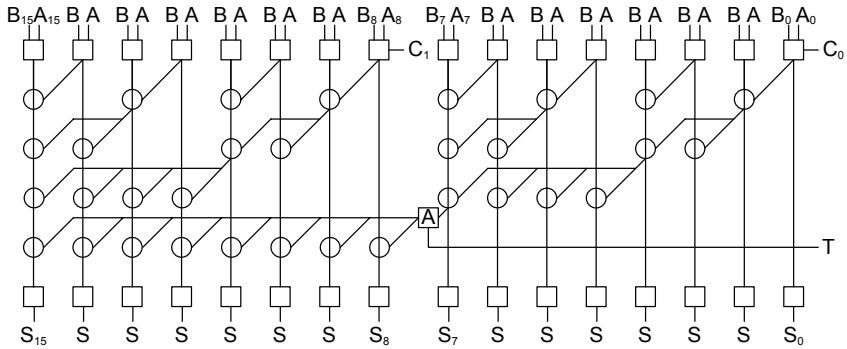


Figure F.26: *A 16-bit Sklansky adder can be modified to optionally support also two 8-bit additions.*

To support the PSUB instruction it is necessary to insert a carry at position 16 of the adder: For this purpose we use, at position 16, a circuit that generates the propagate and generate signals from bits of the A and B operands and an input carry signal (C).

The ALU is also extended with the PACK and SIGN16 instructions. These are low-complexity instructions, which only cause input signals to be rerouted.

F.9.3 Evaluation and Results

The processor using the light-weight SIMD extension was evaluated using the compiler, simulation and implementation framework of the FlexSoC project [36, 37]. The experimental framework consists of a compiler that reads MIPS assembly, and generates data traces both used for cycle-accurate architectural simulations and for logic simulation of our VHDL implementations.

The experimental framework was updated so that the compiler supports the new PMULT, PADD, PSUB, PACK and SIGN16 instructions. The VHDL implementation of the processor datapath was updated with the modified ALU and a twin-precision Baugh-Wooley multiplier, as described in the previous

sections. In this implementation, in order to reduce the critical path of the processor, the multiplier is pipelined into two stages. Both the reference and the twin-precision enabled processor used multipliers that had a level of registers situated between reduction tree and final adder[8]. The VHDL implementation does not include the logic for the control circuitry, but timing and power estimates do include buffers for the datapath's control signals. The addition of only five instructions is expected to have a negligible impact on processor performance, in terms of control circuitry.

To evaluate the added SIMD capability, we modified the Fast Fourier Transform (FFT) application from the EEMBC [27] TeleBench suite. The application implements a decimation-in-time 256-point 16-bit FFT. The application source code was compiled to assembly using GCC mips-cross-compiler with EEMBC's default optimization flags set. The computational kernel was identified and MULT, ADD, and SUB instructions were manually changed to PMULT, PADD, and PSUB instructions when applicable. The PACK instruction was used for packing the two 32-bit results from a PMULT instruction into one packed 16-bit data item. The SIGN16 instruction was never used by the FFT application, but it is still included in the implementation as a support for other applications.

VHDL implementations of the processor with and without the light-weight SIMD extension were taken through a synthesis [22] and place&route [23] flow for the 130-nm process used in previous sections. Delay and power estimations are based on RC extracted data from the placed and routed netlists. For power estimation, value change dump (VCD) data from logical simulations of the FFT applications was used to capture accurate switching activities. The presented power estimates are defined at the maximum clock frequency for each of the implementations.

Table F.4 shows the results for the two processor implementations. The delay and power values are almost identical; a 60 ps and a 0.09 mW increase can be observed for the processor implementation with SIMD support. The processor with SIMD extension has a limited area penalty of 6%.

[8] Pipelining a twin-precision multiplier is as straightforward as for conventional multipliers.

Table F.4: *Delay, Power, and Area Estimates*

	Delay (ns)	Power (mW)	Area (kμm^2)
Conv.	2.29	6.41	267
SIMD	2.35	6.50	283

The number of executed cycles for the FFT application is reduced by 18%, when comparing the result from the original FFT application executed on the conventional processor with that of the execution of the SIMD-enabled FFT application executed on the processor with SIMD extension, Table F.5. This translates into a 15% speedup of total execution time and a 14% energy reduction for the SIMD-enabled FFT application.

Table F.5: *Cycle Count, Execution Time, and Energy Dissipation*

	Cycle Count	Time (ms)	Energy (μJ)
Conv.	57368 (100%)	131 (100%)	842 (100%)
SIMD	47292 (82%)	111 (85%)	722 (86%)

The main memory was not modeled here and, thus, power dissipation for memory accesses is not included. However, since retrieving and storing the data of the FFT computation is done on packed 16-bit data, where two 16-bit data items are read in parallel, the SIMD implementation has 39-43% less read and write accesses to main memory, Table F.6. This is close to the optimal reduction of memory accesses that can be achieved by utilizing SIMD execution. If all performed accesses are for retrieving and storing data of the FFT computation, the number of memory access could ideally be reduced by 50%, since two data items are read in parallel. The reason that the reduction of memory access deviates from the ideal is due to storing and retrieving variables, such as address pointers, on the stack located in main memory.

In embedded multimedia systems, access to main memory and traffic on shared buses are one of the main concerns during system design. With a non-ideal memory subsystem, the access to memory could take longer time than

Table F.6: *Memory Read and Write Access*

	Read	Write
Conv.	9013 (100%)	5568 (100%)
SIMD	5170 (57%)	3371 (61%)

the single clock cycle we have assumed. This could have a larger negative impact on the execution time for the conventional implementation. The fewer number of accesses should also reduce the energy dissipation for the SIMD implementation, even though more accesses are 32-bit wide instead of 16-bit as for the conventional implementation.

F.10 Conclusions

The presented twin-precision technique allows for flexible architectural solutions, where the variation in operand bitwidth that is common in most applications can be harnessed to decrease power dissipation and to increase throughput of multiplications.

It turns out that the Baugh-Wooley algorithm implemented on a HPM reduction tree is particularly suitable for a twin-precision implementation. Due to the simplicity of the implementation, only minor modifications are needed to comply with the twin-precision technique. This makes for an efficient twin-precision implementation, capable of both signed and unsigned multiplications.

Currently a lot of research is done on reconfigurable architectures, where the architecture can be adapted to the applications that are being executed. Some of these proposed architectures can adapt their arithmetic logic units to operate on different bitwidths, depending on the application. One such reconfigurable architecture is that of the FlexSoC project [38]. In these types of architectures it is necessary to have a multiplier that can efficiently operate over a wide range of bitwidths. The twin-precision technique, which offers flexibility at a low implementation overhead, makes it possible to efficiently deploy these flexible architectures.

F.11 Acknowledgment

We thank the Swedish Foundation for Strategic Research (SSF) for funding through the FlexSoC project. We also wish to thank our coworkers and students that have contributed to this project: Dr. Henrik Eriksson, Dr. Mindaugas Draždžiulis, Lic Eng. Mafijul Islam, Martin Brinck, Kristian Eklund, Camilla Berglund, Lars Johansson, Mikael Samuelsson, and Johan Moe.

Bibliography

[1] David Brooks and Margaret Martonosi, "Dynamically Exploiting Narrow Width Operands to Improve Processor Power and Performance," in *5th International Symposium on High Performance Computer Architecture*, January 1999, pp. 13–22.

[2] Zhijun Huang and Miloš D. Ercegovac, "Two-Dimensional Signal Gating for Low-Power Array Multiplier Design," in *IEEE International Symposium on Circuits and Systems*, May 2002, pp. 489–492.

[3] Kyungtae Han, Brian L. Evans, and Earl E. Swartzlander Jr., "Data Wordlength Reduction for Low-Power Signal Processing Software," in *IEEE Workshop on Signal Processing Systems*, 2004, pp. 343–348.

[4] Gabrial H. Loh, "Exploiting Data-Width Locality to Increase Superscalar Execution Bandwidth," in *35th International Symposium on Microarchitecture*, 2002.

[5] Alber Danysh and Dimitri Tan, "Architecture and Implementation of a Vector/SIMD Multiply-Accumulate Unit," *IEEE Transactions on Computers*, vol. 5, no. 4, pp. 284–293, May 2005.

[6] Pedram Mokrian, Majid Ahmadi, Graham Jullien, and W.C. Miller, "A Reconfigurable Digital Multiplier Architecture," in *Proceedings of the IEEE Canadian Conference on Electrical and Computer Engineering*, 2003, pp. 125–128.

[7] Magnus Själander, Henrik Eriksson, and Per Larsson-Edefors, "An Efficient Twin-Precision Multiplier," in *IEEE Proceedings of the 22nd International Conference on Computer Design*, October 2004, pp. 30–33.

[8] David A. Patterson and John L. Hennessy, *Computer Organization & Design, The Hardware/Software Interface*, Morgan Kaufman Publishers Inc., 2nd edition, 1998.

[9] Vojin G. Oklobdzija, David Villeger, and Simon S. Liu, "A Method for Speed Optimized Partial Product Reduction and Generation of Fast Parallel Multipliers Using an Algorithmic Approach," *IEEE Transactions on Computers*, vol. 45, no. 3, pp. 294–306, March 1996.

[10] Luigi Dadda, "Some Schemes for Parallel Adders," *Alta Frequenza*, vol. 34, no. 5, pp. 349–356, May 1965.

[11] Christopher S. Wallace, "A Suggestion for a Fast Multiplier," *IEEE Transactions on Electronic Computers*, vol. 13, pp. 14–17, February 1964.

[12] Henrik Eriksson, Per Larsson-Edefors, Mary Sheeran, Magnus Själander, Daniel Johansson, and Martin Schölin, "Multiplier Reduction Tree with Logarithmic Logic Depth and Regular Connectivity," in *IEEE International Symposium on Circuits and Systems*, May 2006.

[13] Charles R. Baugh and Bruce A. Wooley, "A Two's Complement Parallel Array Multiplication Algorithm," *IEEE Transactions on Computers*, vol. 22, pp. 1045–1047, December 1973.

[14] Mehid Hatamian, "A 70-MHz 8-bit x 8-bit Parallel Pipelined Multiplier in 2.5-μm CMOS," *IEEE Journal on Solid-State Circuits*, vol. 21, no. 4, pp. 505–513, August 1986.

[15] Andrew D. Booth, "A Signed Binary Multiplication Technique," *Quarterly Journal of Mechanics and Applied Mathematics*, vol. 4, no. 2, pp. 236–240, 1951.

[16] O.L.MacSorley, "High Speed Arithmetic in Binary Computers," in *Proceedings of the IRE*, January 1961, vol. 49, pp. 67–97.

[17] Jalil Fadavi-Ardekani, "MxN Booth Encoded Multiplier Generator Using Optimized Wallace trees," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 1, no. 2, pp. 120–125, 1993.

[18] Wen-Chang Yeh and Chein-Wei Jen, "High-Speed Booth Encoded Parallel Multiplier Design," *IEEE Transactions on Computers*, vol. 49, no. 7, pp. 692–701, July 2000.

[19] Magnus Själander, "HMS Multiplier Generator," `http://www.sjalander.com/research/multiplier`, February 2008.

[20] Peter M. Kogge and Harold S. Stone, "A Parallel Algorithm for the Efficient Solution of a General Class of Recurrence Equations," *IEEE Transactions on Computers*, vol. 22, no. 8, pp. 786–793, August 1973.

[21] *Cadence NC-VHDL Simulator Help Version 5.1*.

[22] *Synopsys Design Compiler User Guide Version W-2004.12*.

[23] *Cadence Encounter User Guide Version 4.1*.

[24] Magnus Själander, Mindaugas Draždžiulis, Per Larsson-Edefors, and Henrik Eriksson, "A Low-Leakage Twin-Precision Multiplier Using Reconfigurable Power Gating," in *International Symposium on Circuits and Systems*, May 2005, pp. 1654–1657.

[25] *Cadence Encounter User Guide Version 6.2*.

[26] Todd Austin, Eric Larson, and Dan Ernst, "SimpleScalar: An Infrastructure for Computer System Modeling," *Computer*, vol. 35, no. 2, pp. 59–67, February 2002.

[27] "Embedded Microprocessor Benchmark Consortium," `http://www.eembc.org`.

[28] Srinivas K. Raman, Vladimir Pentkovski, and Jagannath Keshava, "Implementing Streaming SIMD Extensions on the Pentium III Processor," *IEEE Micro*, vol. 20, no. 4, pp. 47–57, July/August 2000.

[29] Keith Diefendorff, Pradeep K. Dubey, Ron Hochsprung, and Hunter Scale, "AltiVec Extension to PowerPC Accelerates Media Processing," *IEEE Micro*, vol. 20, no. 2, pp. 85–95, Marsh/April 2000.

[30] Marc Tremblay, Michael O'Connor, Venkatesh Narayanan, and Liang He, "VIS Speeds New Media Processing," *IEEE Micro*, vol. 16, no. 4, pp. 10–20, August 1996.

[31] Ruby B. Lee, "Accelerating Multimedia with Enhanced Microprocessors," *IEEE Micro*, vol. 15, no. 2, pp. 22–32, April 1995.

[32] Ruby B. Lee, "Subword Parallelism with MAX-2," *IEEE Micro*, vol. 16, no. 4, pp. 51–59, August 1996.

[33] John Goodacre and Andrew N. Sloss, " Parallelism and the ARM Instruction Set Architecture," *IEEE Computer*, vol. 38, no. 7, pp. 42–50, July 2005.

[34] ARM Limited, 110 Fulbourn Road Cambridge, England CB1 9NJ, *ARM Architecture Reference Manual.*, i edition, July 2005.

[35] J. Sklansky, "Conditional-Sum Addition Logic," *IRE Transaction on Electronic Computers*, pp. 226–231, June 1960.

[36] Magnus Själander, Per Larsson-Edefors, and Magnus Björk, "A Flexible Datapath Interconnect for Embedded Applications," in *IEEE Computer Society Annual Symposium on VLSI*, May 2007.

[37] Martin Thuresson, Magnus Själander, Magnus Björk, Lars Svensson, Per Larsson-Edefors, and Per Stenstöm, "FlexCore: Utilizing Exposed Datapath Control for Efficient Computing," in *IEEE International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation*, July 2007, pp. 18–25.

[38] John Hughes, Kjell Jeppson, Per Larsson-Edefors, Mary Sheeran, Per Stenström, and Lars "J." Svensson, "FlexSoC: Combining Flexibility and Efficiency in SoC Designs," in *Proceedings of the IEEE NorChip Conference*, 2003.

PAPER G

M. Själander, A. Terechko, and M. Duranton

A Look-Ahead Task Management Unit for Embedded Multi-Core Architectures

Euromicro Conference on Digital System Design
Architectures, Methods and Tools
Parma, Italy, September 3-5, 2008.

G

A Look-Ahead Task Management Unit for Embedded Multi-Core Architectures

Efficient utilization of multi-core architectures relies on the partitioning of applications into tasks and mapping the tasks to cores. In some applications (e.g. H.264 video decoding parallelized at macro-block level) these tasks have dependencies among each other. Task scheduling, consisting of selecting a task with satisfied dependencies and mapping it to a core, is typically a functionality delegated to the Operating System. In this paper we present a hardware Task Management Unit (TMU) that looks ahead in time to find tasks to be executed by a multi-core architecture. The look-ahead functionality is shown to reduce the task management overhead by 40-50% when executing a parallelized version of an H.264 video decoder on an architecture with up to 16 cores. In overall, the TMU-based multi-core architecture reaches a speedup of more than 14x

on 16 cores running H.264 video decoding, assuming CABAC is implemented in a dedicated coprocessor.

G.1 Introduction

The current trend in computer architecture is to add more cores to increase performance. This is also true in the embedded domain, where multi-core solutions are common and for which the general trend is even towards many-core architectures. In order to improve performance on a multi-core architecture, it is necessary that applications are partitioned into tasks, which can be executed in parallel on separate cores.

As the number of cores increases, it becomes necessary to partition applications into more and smaller tasks, to keep all the cores busy and accelerate overall application performance. The creation and distribution of tasks (henceforth called *task scheduling*) has commonly been handled in software. However, as tasks become smaller and increase in number, a software solution will introduce overheads and will not be efficient. Kumar *et al.* [1] showed that by adding hardware support for scheduling of fine grained parallel applications, speedups of 1.7-2.1 can be achieved compared to full software task scheduling.

In the multimedia domain, partitioning of an application will commonly introduce dependencies between tasks, forcing tasks to be executed in a certain order. Examples of such dependencies will be shown for a H.264 decoder in section G.2 of this article. For this kind of application workloads, task scheduling becomes a *task management* problem. Together with task creation and distribution, it is also necessary to introduce algorithms that keep track of dependencies and that determine when tasks are ready to be executed. The algorithms do not have to be complex, but they still can introduce large overheads. For a parallelized decoder for Super HD H.264 resolution (3840 pixels x 2160 lines), the code for keeping track of when a task can be executed accounts for 9% of the execution time of parallelized sections.

Code for managing tasks is generally simple, consisting of arithmetic operations (such as integer addition, subtraction, and comparison), branching and atomic loads and stores. These operations can be efficiently implemented in a *task management unit* that will offload the dependency computation from con-

ventional cores. This will allow for a more efficient use of resources, since a tailored unit can execute dependency checks more efficiently, in terms of power and area, than executed on a conventional core.

A task management unit can wait until a task executed by a core has finished its execution before updating its dependencies. In this way, the task management unit will have the current state of dependencies and will know exactly which task(s) that can be executed next. However, if there is no previous task that is ready for execution when the core finishes its current execution, it has to wait for the task management unit to update the dependencies and see if there are new tasks ready for execution. This introduces extra delays, since the check is performed in between two executed tasks.

It is in general not possible to update dependencies before a task has finished its execution. However, while a task is being executed, it may be possible to find what dependencies will be solved by the currently executed task. If there are tasks that only depend on the currently executed task, then these tasks will be ready for execution, once the currently executed task is finished. These ready tasks can be prepared for execution by a task management unit, such that once the core has finished the current execution it can immediately start the execution of the next task. Updates of dependencies can then be executed by the task management unit in parallel with the execution of the task. An added benefit is that tasks dependent on each other are executed by the same core, which can improve data locality.

The proposed architecture consists of look-ahead Task Management Units (TMUs), capable of executing task-dependency checks in parallel with the execution of tasks. Each TMU offloads dependency checks/updates from a number of conventional cores and tries to schedule dependent tasks onto the same core, thus preserving data locality. Distribution of tasks between TMUs is done through a task queue. By executing the task-dependency checks on the TMUs in parallel with the conventional cores, an execution-time speedup of 4.5% for a multi-core architecture running a H.264 decoder for Super HD resolution is achieved, compared to a multi-core architecture solely based on a task queue for task distribution. This constitutes a 50% reduction of the overhead for managing the tasks. Furthermore, offloading work from conventional cores to the

TMUs, improves the energy efficiency of a multi-core architecture.

This paper is organized as follows. In section G.2 parallel applications with inter-task dependencies are presented. Section G.3 describes the general problem of managing tasks for parallel applications with inter-task dependencies. We then present the task management unit and the look-ahead functionality used for accelerating task management in section G.4. The evaluation setup and results are presented in section G.5 and G.6. The paper is concluded in section G.9.

G.2 Parallel Applications with Task Dependencies

For this work, we target application workloads where the partitioning of an application into tasks introduces dependencies between the tasks (e.g. H.264 parallel video decoding with macro-block level parallelism [2] and spatio-temporal motion prediction 3DRS [3]). This kind of applications differs from other parallel workloads, such as server workloads with multiple incoming requests, desktop workloads consisting of multiple programs, and scientific workloads, where the tasks are commonly independent of each other and can be executed in random order. For applications with inter-task dependencies, the execution order is crucial for correct application behavior. The execution order can not always be totally statically determined at compile time, because of variations in task execution time. Hence, our approach manages task dependencies dynamically at run time.

G.2.1 H.264 Video Decoder

The driving application for the proposed architecture is Super HD (3840x 2160 pixels) H.264 video decoding [4] and will be used throughout the article as a representative application with task dependencies. The H.264 advanced video compression standard dominates the embedded markets in both high-resolution video (e.g. BluRay players) and low-bandwidth (e.g. Sony PSP) segments. H.264 video decoding in Super HD requires a multi-core architecture, to reach the performance necessary for decoding in real-tin (30 frames per second). With the most powerful single core solutions currently on the market, it is possible

to decode HD (720p) H.264 in real time [5]. However, Super HD requires about nine times more performance, which with the current trend of more cores instead of one faster core, makes a multi-core architecture for decoding Super HD H.264 a necessity.

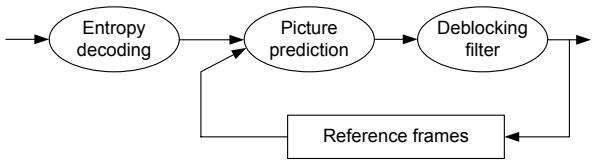


Figure G.1: *The main components of a H.264 decoder.*

Fig. G.1 shows the main components of a H.264 video decoder. Processing of a frame starts with entropy decoding, consisting of either Context-Adaptive Binary Arithmetic Coding (CABAC) or a Context-Adaptive Variable Length Coding (CAVLC) and both are sequential by nature. The frame is then passed on to the picture prediction stage where it is divided into macro-blocks of maximum 16 times 16 pixels. At this stage, also inter-picture prediction and motion-vector estimation are calculated for each macro-block. The frame is then filtered through a deblocking filter to reduce artifacts at block boundaries introduced by the picture-prediction stage. The resulting frame has then been decoded and can be transferred to the display. The decoded frame is also buffered for inter-picture prediction.

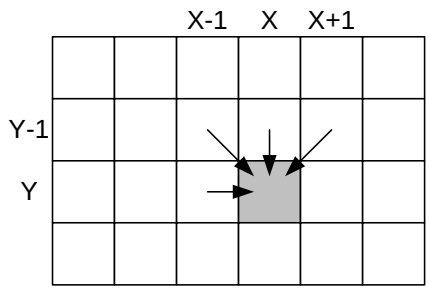


Figure G.2: *Dependencies between macro-blocks in the H.264 video compression standard.*

The picture prediction and deblocking filter is suitable for parallelization, where the execution of a macro-block can be treated as a task. However, a macro-block cannot be executed before certain neighboring macro-blocks have been executed [2]. A generalized illustration of the macro-block dependencies is shown in Fig. G.2. From the figure, it can be seen that before the macro-block marked in gray can be executed, the macro-block to the left and the three at the top have to be executed.

A task dependency graph can be constructed from the macro-block dependencies, as shown in Fig. G.3. The graph starts with the macro-block in the upper left corner, since all its dependencies are solved. When the first task (0/0) has been executed, the second task (1/0) can start. Each of the new tasks can potentially start the execution of one or two other tasks, for example, after task (1/0) both (2/0) and (0/1) can start.

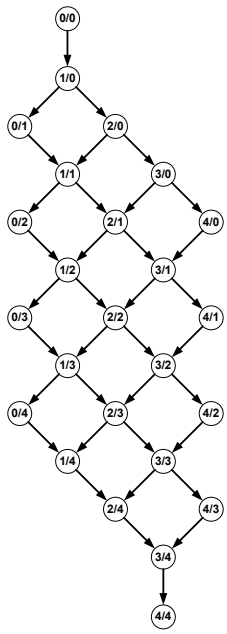


Figure G.3: *A task dependency graph for a H.264 decoder, with 25 (5x5) macro-blocks in a frame.*

Task dependencies can be tracked by storing the number of tasks that each task depends on. For H.264 this value is zero, one, or two. For each finished task, the array with dependencies is updated, decrementing the values of the tasks that depend on the finished task. A task can execute once its value in the array becomes zero [6]. The update of the dependencies for the H.264 decoder can be described by the pseudo code shown in Fig. G.4. Here X and Y are the indices of the macro-block that just finished its execution. The first if statement checks that the dependent macro-block is within the frame. If the macro-block is within the frame, its value of number of dependencies is decremented with one. If the value becomes zero the macro-block has no more dependencies and is ready to be executed. Note that the decrement operations must be executed atomically.

```
if X<FrameWidth-1 then
  decrement(dependencies[X+1][Y])
  if dependencies[X+1][Y] = 0 then
    ready_to_execute(macro_block[X+1][Y])
  end if
end if

if X>0 and Y<FrameHeight-1 then
  decrement(dependencies[X-1][Y+1])
  if dependencies[X-1][Y+1] = 0 then
    ready_to_execute(macro_block[X-1][Y+1])
  end if
end if
```

Figure G.4: *Pseudo code for inter-macroblock dependency checks.*

G.3 Task Management

In an architecture based on task queues for task scheduling, the execution of a task is followed by a piece of code (Fig. G.5) that updates dependencies and checks for tasks ready to be executed. Fig. G.5 illustrates the execution of twelve different tasks that are followed by the task dependency execution. Each task is, for simplicity of the drawing, assumed to be identical in execution time,

which is not the case in practice. The code for the task dependency check is often simple, but still it can be a significant part of the total execution time. For a Super HD H.264 decoder, the dependency check has been found to increase a tasks' execution time with 9%, on average. Or in other words, the compute power of a complete core is required for managing tasks for every 11 cores in the architecture.

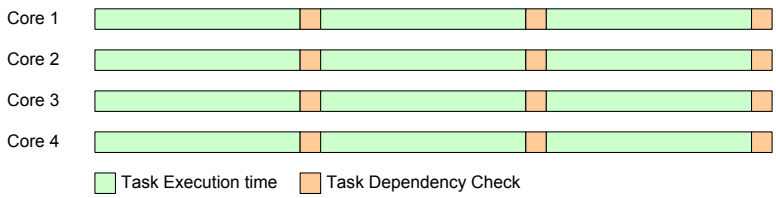


Figure G.5: *Illustration of the execution for a conventional architecture.*

To increase the execution speed of an application with task dependencies and increase silicon efficiency, the dependency checks could be accelerated by a Task Management Unit (TMU) that keeps track of task dependencies and determines when tasks are ready to be executed. A straight-forward implementation of such a Task Management unit (TMU) is simply to offload the computation of the task dependencies from the core executing the task. Once a task is done, it signals to the TMU that the task is finished and the TMU then starts to look for a new task to be executed by the core. The TMU can be designed to execute task dependency code more efficiently than a conventional core and therefore will execute the code faster. However, if the computation of dependencies is started right after the core finishes its current execution, it will have to wait until the TMU finishes updating dependencies and finds a new task to be executed. Even worse, there will be communication overheads in terms of the core signaling the TMU and TMU instructing the core which task to execute. Therefore, even if the TMU executes the dependency check code more efficiently than a conventional core, the added communication overhead could even degrade the overall performance. Furthermore, if several cores finish at the same time, they will have to wait for each others' dependency code to be executed or the TMU would have to be multi-threaded. Fig. G.6 illustrates the execution of twelve tasks executed on an architecture with a naive TMU implementation.

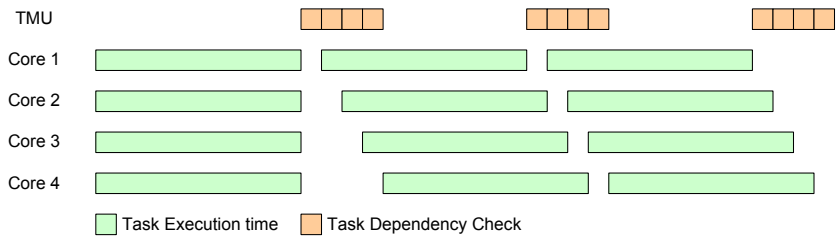


Figure G.6: *Illustration of the execution for a naive implementation of a task management unit (TMU).*

For an efficient solution, it is necessary to execute as much as possible of the task dependency code in *parallel* with the execution of tasks. In the case where there are more tasks than cores, it is possible to fetch tasks from a task queue and have a core to execute it as soon as the core is done with its current task. The TMU will have to keep track of which task finishes and, when it gets spare time, execute the task dependency code for each task and populate the task queue with new tasks. This implementation choice has two major drawbacks: *i)* If there are fewer tasks than cores, the TMU will still have to execute the task dependency code before it can find new tasks for a core to execute, thus introducing delays similar to the straight-forward solution. *ii)* By always storing the task in the task queue, data locality might be lost, since tasks depending on each other are more likely to be executed on different cores. This would cause extra pressure on the memory subsystem and have a negative impact on execution time.

It is possible to find all tasks that depend on the currently executed task during its execution. All tasks found are then candidates for execution by the core. Of all the found candidate tasks, there can be tasks that only depend on the task that is currently executed by the core. One of these tasks can immediately be executed by the core, once the core is finished with its current task. For example, in Fig. G.3 while processing (1/0) the TMU can find out that two tasks, (2/0) and (0/1), can start immediately after (1/0) has been executed. Updating dependencies of the tasks not chosen for execution can then be done in parallel by the TMU. This allows a minimum amount of time between the execution of tasks (henceforth called *turn-around time*) and dependent tasks are

executed on the same core, thus improving data locality. For the case where the TMU is not able to find a task ready for execution, it still can improve the turn-around time between tasks. By identifying all dependent tasks and have them readily available, the update of dependencies can be made as fast as possible. Fig. G.7 shows an example where twelve tasks are being executed on four different cores, with their look-ahead code being executed in parallel on a TMU. For the first executed task on the first core and the second executed task on the second core, a task is found with its dependencies fulfilled and can be started immediately, once the core has finished the current execution. For the other tasks, the dependencies need to be updated before a task is found for execution.

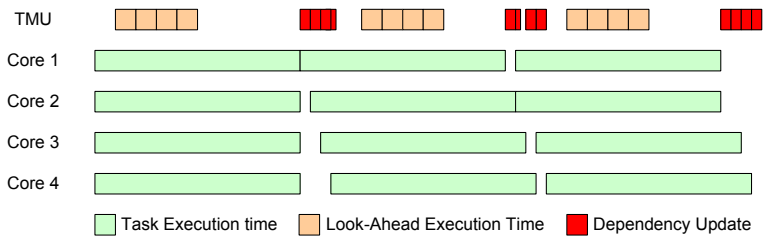


Figure G.7: *Illustration of an execution of tasks, where the task management unit (TMU) in parallel with the tasks executed on the cores, looks ahead to find tasks with solved dependencies.*

G.4 Task Management Unit

The purpose of the Task Management Unit (TMU) is to offload the management of tasks from conventional cores in a multi-core system. The TMU is therefore closely connected to a number of cores that signals to the TMU each time they have finished a task. While tasks are being executed on the cores, the TMU tries to find tasks that are ready to be executed and have them prepared, so that a core can directly start executing a new task when it finishes its current execution.

For each task being executed, the TMU executes a small function that looks ahead in time, in order to try to find tasks that will be ready for execution. The more cores there are in a system the more look-ahead functions need to be executed at a single point of time. In order to be scalable to more than a few

cores, several TMUs are instantiated, with each TMU connected to a limited number of cores. The ratio between TMUs and cores depend on the ratio of task load versus control load. The TMU offloads the control load from several cores and execute them sequentially. A TMU should therefore be faster than the combined control-load for the cores. With one TMU offloading four cores the ratio between the task load and control load can be as high as four to one (25% of the application load). Task queues are then used to distribute tasks between the TMUs. The task queues stores ready tasks that are currently not being executed by a core. Fig. G.8 shows an architecture with four TMUs, with each TMU connected to four cores. In this case the TMUs share one task queue. However, more advanced queuing systems can be used to improve performance.

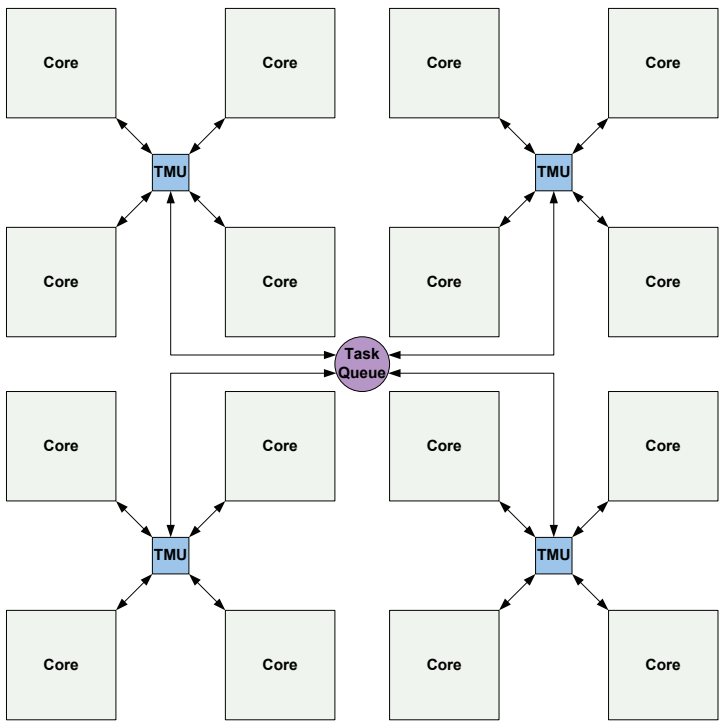


Figure G.8: *Architecture with 16 cores and 4 task management units (TMU), which share one task queue.*

The TMU should be programmable to allow for complex task dependencies, be capable of executing simple arithmetic operations, and have fast branching. Furthermore, a programmable TMU can support irregular task dependencies, even not known statically at compile time. For greater flexibility, each TMU also has an associated hardware mailbox that can be used for message passing. This allows, for example, a programmer to send messages from within the execution of a task to a TMU.

G.4.1 Look-Ahead Task Management

To keep track of tasks and speed up the turn-around time between executed tasks, each core has a dedicated hardware list. The hardware list holds necessary information about each candidate task. A candidate task is a task that could become ready for execution, once the core has finished with its current task. Fig. G.9 shows the list for storing candidate tasks and the information stored for each task.

Task Pointer: The task pointer holds the instruction address of the first instruction of the task.

Argument(s) Pointer: Holds the address to where arguments for the task are stored.

Look-Ahead Pointer: The look-ahead pointer tells the TMU what look-ahead function to execute while the task is executed by the core.

Dependency Pointer: The dependency pointer holds the address to the memory location that stores the number of dependencies that still have to be solved before the task can be executed.

Flags: A set of flags are used for synchronization of the core and TMU and to tell which state the task is in.

The main purpose of the candidate list is to speed up the turn-around time between tasks being executed by a core. This is achieved by allowing direct access to the list from the core. The list is controlled by a simple state machine that inspects the state of the flags of each task in the list. If there is a task ready for execution, then the core reads the task and argument pointer from the candidate list and starts to execute the new task. Then, in parallel with the execution of the task, the TMU will decrement the value given by the dependency pointers

Flags	Look-Ahead Pointer	Dependency Pointer	Task Pointer	Argument(s) Pointer

Figure G.9: *The candidate list that stores candidate tasks for a core.*

for the tasks not being executed. In case there is no ready task the core will have to wait until the TMU has updated the dependencies to see if a task becomes ready for execution.

The TMU is governed by a simple state machine that checks the state of the task queue, the flags of the candidate list for each core, and for incoming messages. If there is an idle core and a task is in the task queue, a task will be fetched from the queue and the candidate list will be updated. This instructs the core that a task is ready for execution. A small routine is called if a core has finished its execution of a task. The routine first checks for tasks that are ready for execution, but are not executed by the core. These task(s) are stored in the task queue for execution at a later time. Then dependency values for tasks not ready to be executed are decremented. Therefore, only the dependencies of candidate tasks not found ready for execution need to be updated. This reduces the total number of atomic accesses and the pressure on the memory subsystem. Finally, the look-ahead and argument pointer are read for the task currently being executed and the look-ahead function is executed by the TMU, which updates the candidate list with new tasks. Each TMU has also a mailbox for message passing to improve programmability. The state machine checks for incoming messages and will execute a routine for each new message.

The candidate list is a hardware structure with a fixed number of entries. This limits the number of dependent tasks (children) that a particular task (parent) can have. For the H.264 decoder there are no tasks with more than two children, but this might not be the case for other applications. To allow tasks with unlimited number of children, it is possible to add tasks with the only purpose of updating other task's dependencies. This will introduce overheads,

so the candidate list should have enough entries for the most common number of children of the applications to be run on the system. However, Stavrou *et al.* [7] performed code analysis that showed that the majority of data-driven multithreading tasks have at most two dependencies, so a candidate list with four entries should be sufficient in most cases.

G.5 Evaluation Setup

The evaluation of the architecture has been conducted through emulating the architecture in the TTISim simulator [8]. TTISim is a multi-core simulator for the TriMedia VLIW processor family [9]. Each core and Task Management Unit (TMU) is modeled by a TriMedia 3270 processor [10], with the behavior of the TMU implemented in software. The software implementation of the TMU is of course not as efficient as a hardware implementation would be. To emulate the performance of a hardware implemented TMU a stalling technique has been used to remove software overheads.

The *stalling* technique works as follows. For each functionality implemented in software, an estimate has been made of how many cycles it would take to execute the same functionality in hardware. The software implementation is then allowed to run for as many cycles as estimated before all other cores in the system are being stalled. This allows the execution of the remaining part of the software implementation to be executed without influencing the behavior of the rest of the parallel system. The execution time for an application can then be calculated by counting the number of cycles that are being stalled and deducting them from the total execution time. An example of when the TriMedia processors are stalled is when a TMU is updating one of its candidate lists with new tasks. The candidate lists will be implemented in hardware, but for the simulation, they are software structures stored in shared memory. To emulate fast access to the candidate lists parts of the update is therefore stalled.

The TMU is assumed to execute the look-ahead functions at least as fast as the TriMedia processor. This is a reasonable assumption, since the inherent nature of scheduling/control code is that it lacks instruction level parallelism and contains many branches. A specialized datapath with simple arithmetic

operations and fast branching should have similar performance or be even faster than the TriMedia processor for this type of code. Furthermore, a specialized datapath should dissipate less power and have a smaller area footprint.

G.5.1 TriMedia H.264 Video Decoder

A H.264 video decoder [11] that decodes a Super HD (3840x2160 pixels per frame) video stream, was selected as application workload. The original H.264 decoder was manually optimized for a single TriMedia TM3270 processor. The optimizations included adding SIMD operations, restrict pointers, loop unrolling, code restructuring, etc.

G.5.2 H.264 on the Reference Architecture

The H.264 decoder has been manually parallelized to be executed on an architecture based on a shared software task queue [6]. The parallelization of the H.264 decoder consisted of identifying the nested loops and functions for computing the picture prediction and deblocking filter of a frame. The functions for computing a single macro-block were augmented with code for checking what macro-blocks are ready to be executed. The augmented code is similar to the code described in Fig. G.4. If only one macro-block is found to be ready, the indices X and Y are updated and a simple jump back to the beginning of the function is made. In the case where two macro-blocks are found to be active, one of them gets stored in the task queue before updating the indices and jumping back to the beginning of the function. In the case where no macro-block is found to be ready, the function is exited and a new task is fetched from the task queue. The jump to the beginning of the function means that dependent tasks are executed on the same core. This preserves data locality between macro-blocks and removes task creation overhead. Before the first task of each frame is called, an array storing the number of dependencies each macro-block has is initialized. To accurately capture the state of the dependencies, all subsequent updates of the dependency array have to be made atomically.

The reason why an architecture based on a software implementation of a shared task queue has been chosen is because only 0.8-1.7% of the tasks where

found to be stored in the task queue when decoding a Super HD H.264 stream, on systems with four to 16 cores. For 98.5-99.3% of the tasks executed, a new macro-block was found ready to be executed with a simple jump to the beginning of the function as a result. The architecture is simulated using the same simulator and 3270 TriMedia processors, which allows for accurate performance comparisons with the proposed TMU architecture. For a different application workload, with more accesses to the task queue, both the architecture based on the task queue and the TMU would benefit from a hardware implementation of the queue(s).

The execution time of each task was measured to improve the understanding of the behavior of the H.264 decoder. The distribution of tasks is shown in Fig. G.10. The picture prediction stage of the H.264 decoder has been implemented as two separate functions, PicturePrediction and VectorPrediction. Further, frames can be of P or B type and the distribution for each frame type is shown.

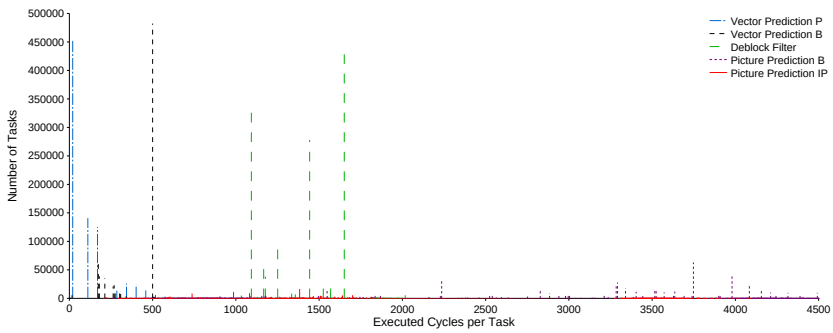


Figure G.10: *Task execution cycle time for different computations in the H.264 decoder. Tasks larger than 4500 is not shown because they are so few that they would not be visible in the graph.*

Table G.1 shows a summary of the task distribution and the percentage of the total number of executed cycles that is contributed by each function. The average length of a task was found to be 1270 cycles.

Table G.1: *Task execution cycle time range and percentage of total execution cycle time for Super HD H.264 decoding.*

	Cycle Times	% of Total Executed Cycles
Vector Prediction	20-1153	6.65%
Picture Prediction	208-5326	57.31%
Deblock Filter	978-2017	36.04%

The execution time of the scheduling for the task queue implementation was also measured and was found to be on average 114 cycles. This means that, on average, the overhead for scheduling the next task to be executed is 9%. For tasks with the distribution of the VectorPrediction of frame type P, where many of the tasks are as short as 20 cycles, the average scheduling overhead is 109%. A cycle execution time of 20 cycles is far less than what is commonly considered as the smallest task length for scheduling. Kumar *et al.* [1] considers tasks with at least a few hundred cycles as the smallest task for their carbon architecture.

G.5.3 H.264 on the TMU Architecture

For the TMU architecture, the parallelized H.264 video decoder was manually modified. The modifications are done such that the code, which checks for ready macro-blocks is removed from the tasks and is executed by the TMU instead. The code executed by the TMU only inspects the value stored in the dependency array and if the value is found to be one, then the task is marked as ready in the candidate list.

A single look-ahead function was used to parallelize the H.264 decoder, since all dependencies between macro-blocks can be described by the code given in Fig. G.4. However, before the execution of vector prediction, picture prediction, and deblocking filter, initialization functions are needed for setting up the dependency array. The TMU specific code of the H.264 decoder consists of less than 300 lines of C++.

The TMU and task queue architectures where simulated with 4, 8, and 16 cores. For the TMU based architecture, a configuration with four cores per TMU was chosen. This configuration is not based on an empirical evaluation and will require further investigation. However, a configuration with four cores per TMU allows for a regular floorplan, similar to the illustration shown in Fig. G.8. All simulations are for a perfect memory subsystem. However, the execution order of macro-blocks is similar for both architectures and we should see the same performance degradation for a more realistic memory subsystem.

G.6 Results

The result from the simulations is shown in Fig. G.11. The speedup is given for the parallelized sections of the H.264 decoder. The CABAC encoding for the Super HD video stream is inherently sequential and is assumed to be executed by dedicated hardware. Therefore, its contribution to the total execution time is not considered in the presented speedups. As a reference, the ideal linear speedup is shown in the graph.

As shown by Fig. G.11, the Task Management Unit (TMU) architecture outperforms the task-queue architecture. The simulations showed that for 82% of the tasks executed, a task was already waiting to be executed when the current task finished its execution. This is translated into a performance improvement of 4.5% for the eight core configurations and a lowest performance improvement of 3.4% for the 16 core configurations. This constitutes a 40-50% reduction of the task management overhead seen for the task queue simulations.

When the number of cores increases, there are fewer tasks to execute per core and hence, in some cases, there will be idle cores. This explains why there is a drop in the speedup as the number of cores increase.

Our simulations show that the execution of VectorPrediction tasks is on average 5% slower when executed on the TMU than if solely computed by the conventional cores. This suggests that the TMU becomes a bottleneck when tasks are extremely small. The TMU sequentializes the control and when the control load offloaded from each processor, is higher than 25% of the cores application load, the TMU becomes overloaded. The overloaded TMU introduces delays, since the cores have to wait for the TMU to execute the look-ahead func-

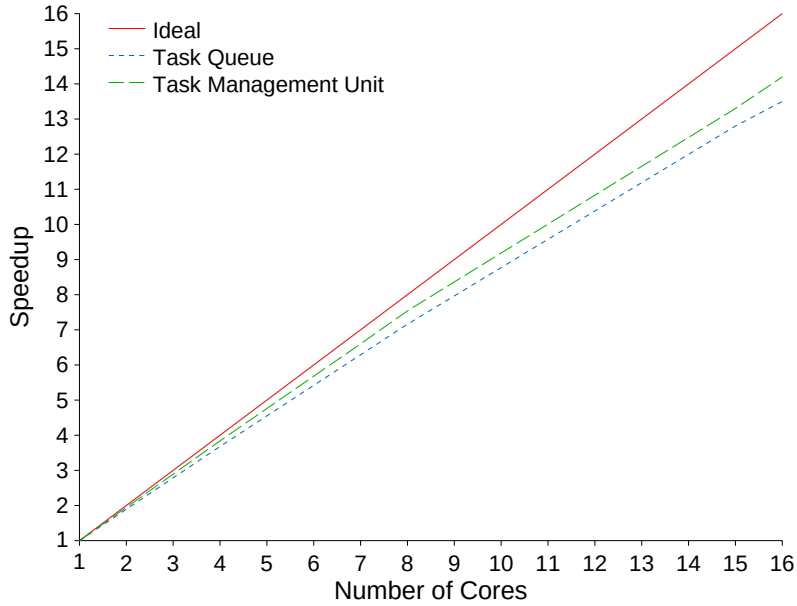


Figure G.11: *The graph shows the speedup as a relation to the number of cores, for executing a Super HD H.264 decoder on two different architectures based on task management unit(s) or a task queue.*

tions. To avoid this, a different configuration with fewer cores per TMU can be used or tasks could be combined to form slightly larger tasks, thus reducing the number of look-ahead functions executed by the TMU. As mentioned in the previous section, VectorPrediction has many small tasks (20 cycles) and tasks of such small size are commonly not considered for task management.

The deblocking filter stage of the H.264 decoder also has a small section that can be parallelized without introducing dependencies between the tasks. For this kind of tasks the TMU does not bring an added value, since there is no need to execute any look-ahead functions. The simulations have shown that the TMU architecture executes these as fast as the task pool architecture. These tasks are not shown in Fig. G.10, but is included when calculating the speedups for Fig. G.11.

G.7 Related Work

Carbon [1] by Kumar *et al.* accelerates creation and distribution of tasks by the introduction of hardware task queues. However, updates and checks of task dependencies are executed by the cores themselves. The Task Management Unit (TMU) offloads the update and check code from the cores, which allows for a more efficient execution. Carbon could be used in conjunction with the TMU to speed up distribution of tasks between different TMUs.

Stavrou *et al.* describes a thread synchronization unit for data-driven multithreading [7] that resembles the presented TMU. Their method is efficient as long as there are more tasks ready for execution than the number of cores. However, the look-ahead technique presented in this paper allows new tasks to be found even in the case where all currently ready tasks are being executed. The look-ahead functionality also allows prefetching to be started earlier for tasks found ready for execution by the TMU.

The synchronization and scheduling unit proposed by Bayer *et al.* [12, 13] is based on a centralized unit and an associated network for task distribution. Centralization of the unit will incur delays as the network scales with more cores. The scheme relies on duplicable tasks that can be packaged into a single package, which is then split into individual tasks and distributed by the network. The TMUs on the other hand, are tightly coupled to each of their cores, thus reducing communication overhead and instead centralize the storage of the task dependency values. The update and check of dependencies are not speed critical for the case when the TMU finds a task to be executed, since the update and check is decoupled from the execution of the task by the look-ahead technique. Furthermore, the TMU does not rely on duplicable tasks, which are not always present in an application (e.g. the H.264 decoder).

G.8 Future Work

Future work includes detailing the instruction set architecture and micro architecture of the Task Management Unit (TMU) and to evaluate power and area benefits of offloading control load from conventional cores to the TMUs. Fur-

ther evaluation is also needed to find the right configuration of the number of cores per TMU for various workloads. An interesting extension to the TMU would be the possibility to perform data prefetching for tasks found ready for execution by the TMU.

G.9 Conclusions

The Task Management Unit (TMU) outperforms the task pool based multi-core architectures due to its look-ahead capability. 82% of the executed tasks benefited from the look-ahead functionality, thereby reducing the time to compute dependencies for next tasks. Overall, the TMU-based multi-core reaches a speedup of more than 14x on 16 cores running Super HD H.264 video decoding. The overall performance is improved by 4.5% for the eight core configuration. This equals to a 50% reduction of the task management overhead of 9% seen for the task queue simulations. Further, the TMU-based multi-core architecture can execute non-dependent tasks as fast as the task queue architecture.

G.10 Acknowledgments

We thank the HiPEAC Network of Excellence for financing the internship during which this research was done. We also thank our colleague Jan Hoogerbrugge at NXP for his help with TTISim and the parallelization of the H.264 decoder.

Bibliography

[1] Sanjeev Kumar, Christopher J. Hughes, and Anthony Nguyen, "Carbon: Architectural Support for Fine-Grained Parallelism on Chip Multiprocessors," in *IEEE International Symposium on Computer Architecture*, June 2007, pp. 734–740.

[2] Erik B. van der Tol, Egbert G.T. Jaspers, and Rob H. Gelderblom, "Mapping of H.264 decoding on a multiprocessor architecture," in *Image and Video Communications and Processing*, 2003, pp. 707–718.

[3] G. de Haan, P. W. A. C. Biezen, H. Huijgen, and O. A. Ojo, "True-motion estimation with 3-D recursive search block matching," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 3, pp. 368–379, October 1993.

[4] Thomas Wiegand, Gary J. Sullivan, Gisle Bjø:ntegaard, and Ajay Luthra, "Overview of the H.264/AVC Video Coding Standard," *IEEE Transaction on Circuits and Systems for Video Technology*, vol. 13, no. 7, pp. 560–576, July 2003.

[5] Apple Inc., "QuickTime HD Gallery System Requirements," `http://www.apple.com/quicktime/guide/hd/recommendations.html`.

[6] Jan Hoogerbrugge and Andrei Terechko, "A Multithreaded Multicore System for Embedded Media Processing," *To Appear in Transactions on High-Performance Embedded Architectures and Compilers*, 2008.

[7] Kyriakos Stavrou, Costas Kyriacou, Paraskevas Evripidou, and Pedro Trancoso, "Chip Multiprocessor Based on Data-Driven Multithreading Model," *International Journal of High Performance Systems Architecture*, vol. 1, no. 1, pp. 24–43, 2007.

[8] NXP Semiconductors, "Nexperia Development Kit for TriMedia processors," `http://www.nxp.com`.

[9] S. Rathnam and G. Slavenburg, "An architectural overview of the programmable multimedia processor, TM-1," in *In Proceedings of Compcon*, 1996, pp. 319–326.

[10] Jan-Willem van de Waerdt, Stamatis Vassiliadis, Sanjeev Das, Sebastian Mirolo, Chris Yen, Bill Zhong, Carlos Basto, Jean-Paul van Itegem, Dinesh Amirtharaj, Kulbhushan Kalra, Pedro Rodriguez, and Hans van Antwerpen, "The TM3270 Media-Processor," in *IEEE/ACM International Symposium on Microarchitecture*, November 2005, p. 12pp.

[11] Jan-Willem van de Waerdt, *The TM3270 Media-processor*, Ph.D. thesis, Delft University of Technology, 2006.

[12] Nimrod Bayer and Ran Ginosar, "High Flow-Rate Synchronizer/Scheduler Apparatus and Method for Multiprocessors," April 1993.

[13] Pelge Avieli, Oded Rubenov, and Nomrod Bayer, "Designing a Central Synchronization/Scheduling Unit For Multiprocessors," in *IEEE Convention of the Electrical and Electronic Engineers*, April 2000, pp. 495–498.