Stellingen

TR3439

behorende bij het proefschrift

Multimedia Hardware Accelerators

Edwin Antonius Hakkennes

Delft, 7 december 1999

1. Een overloop van een optelling onthouden of doorgeven, dat is de vraag in dit proefschrift.

2. Optellen is eenvoudiger als aftrekken.

3. Het aantal reclames voor nieuwe computers houdt gelijke tred met het aantal reclames waarin bedienend personeel wordt geworven.

4. Naarmate de tijd voortschrijdt, duurt alles korter.

5. Als redundantie en fout-corrigerende eigenschappen net zo makkelijk aan schakelingen toegevoegd kon worden als aan opgeslagen data, zou de opbengst van de verschillende chipfabrieken allang tot 100% zijn gestegen.

6. De lage prijs, de goede beschikbaarheid en het gemak van CD-recordables is zeker een factor waardoor Open Source Software in de lift zit.

7. Het algemene ontwikkelingsdoel van computer engineering is het maken van snellere computers. Hieruit kan worden afgeleid dat de huidige computers niet snel genoeg zijn.

8. De regels van de sociale zekerheid veranderen zodra je ervan afhankelijk wordt.

9. De levensduur van straatverlichtingsapparatuur wordt aanmerkelijk verlengd door het gebruik ervan.

10. Het woorduitvinden moet aan banden worden gelegd.

11. Het openbaar vervoer kan veel goedkoper als we het gratis maken.

12. *De betere stellingen worden pas bedacht,*
als het proefschrift naar de drukker is gebracht.

1. To carry or not to carry, that is the question in this thesis.
2. Addition is simpler than subtraction.
3. The number of commercials for new computers is at equal pace with the number of commercials recruiting people to operate them.
4. As time progresses, events last shorter.
5. If redundancy and error-correcting properties could be added to circuitry as easy as to stored data, the yield of the chip manufacturing industry would have increased to a 100% a long time ago.
6. The low price, the high availability and the ease of use of CD Recordables are reasons for Open Source Software to flourish.
7. The general research goal of computer engineering is to provide faster computers. This implies that the speed of today's computers is unsatisfactory.
8. Social security's rules change as soon as one becomes dependent on them.
9. Lamp-posts live longer by using them.
10. Wordinvention should be restrained.
11. Public transport could be much cheaper, if it is provided free of charge.
12. *The more impressive propositions (theses) spring to mind only at the moment the printer is printing the dissertation.*

Multimedia Hardware Accelerators

Edwin Hakkennes

Multimedia Hardware Accelerators

PROEFSCHRIFT

ter verkrijging van de graad van doctor
aan de Technische Universiteit Delft,
op gezag van de Rector Magnificus prof.ir. K.F. Wakker,
in het openbaar te verdedigen ten overstaan van een commissie,
door het College voor Promoties aangewezen,

op dinsdag 7 december 1999 te 16.00 uur

door

Edwin Antonius HAKKENNES

elektrotechnisch ingenieur
geboren te 's Gravenhage

Dit proefschrift is goedgekeurd door de promotor:
Prof.dr. S. Vassiliadis

De leden van de promotiecommissie zijn:

Rector Magnificus,	voorzitter
Prof.dr. S. Vassiliadis,	Technische Universiteit Delft, promotor
Prof.dr.ir. H.J. Sips,	Technische Universiteit Delft
Prof.dr. J.-M. Muller,	Ecole Normale Supérieure de Lyon
Prof.dr. J.G. Delgado-Frias,	State University of New York, Binghampton
Dr.ir. E.F.A Deprettere,	Technische Universiteit Delft
Dr. H. Corporaal,	Technische Universiteit Delft
Dr.ir. P. van der Wolf,	Philips Research Laboratories, Eindhoven

ISBN 90-9013314-3

Subject headings: computer arithmetic, computer design and engineering, multimedia enhancements, hardware accelerators, multimedia processors, multimedia architecture.

The text of this book was edited using various EMACS versions, running under the Linux operating system on a dual pentium workstation. It is typeset in 11 points Times-Roman by LaTeX2e. The figures were created using $xfig$ and the bibliography was compiled by BibTeX.

This dissertation is dedicated to Lyonne,

for her patience and understanding.

Multimedia Hardware Accelerators

Edwin Hakkennes

Abstract

In this dissertation we we investigate the possibility to implement in hardware complex expressions and operations shown to be frequently used in multimedia applications. The first group of expressions we consider can be formulated as a general add-multiply-add operation and thus they can be captured by a single compound instruction family. For this family of operations we show that hardware can be built that requires approximately the same execution pipeline stages as the multiply instruction. This implies that the Add-Multiply-Add family of instructions should require the same execution time as the multiply instruction, significantly speeding up the execution of multimedia applications. Furthermore we show that two operations frequently used in motion estimation, the Sum and the Mean of Absolute Differences, can be implemented in hardware requiring also approximately the same execution time as the multiplication. Our approach can be extended easily to provide the computation of the Sum and Mean of Absolute Differences of a 16x16 pixel block in no more than four machine cycles. Additionally we propose a codec hardwired mechanism for the Paeth predictor used in the Portable Network Graphics Standard (PNG). We show that this execution unit requires at most two general purpose ALU cycles. We extend the Paeth unit to include the median, maximum and minimum operations on three inputs with no additional cycle time and we also extend the Add-Multiply-Add unit to include the mean of three numbers. Finally we propose the designs of two units to accommodate all the proposed instructions. One unit comprises all multiply related operations, the other the ALU related operations. The first one can be viewed as an extension of the multiply execution stage hardware. The other can be viewed as a stand alone, ALU like unit. Both units can be combined to a single execution unit that operates on 32 instructions in total.

Preface and acknowledgements

The work presented in this dissertation has been carried out at the Computer Architecture and Digital Techniques (CARDIT) Laboratory, Electrical Engineering Department, of the Delft University of Technology. Numerous people have contributed in several ways to the results presented here. I will try to give a chronological list of the people who crossed (or should I say pushed my along) my research path. My work at CARDIT begun in September 1992, when ir. Reinoud Koolhaas hired me to supervise practical courses for both first and third year students. The experience gained from this teaching and the technical discussions regarding digital design with Reinoud, ir. Loek Thijssen and the other practical-supervisors (especially John van Leeuwen, Edwin Santing, Bryan Tjon-Kam, Gideon Zweijtzer and Steven Roos) were very valuable. Later on, I started with my masters project, guided by ir. Louis Muller and Jan Snelders. During this masters period, I gained insight in the problems of pattern recognition as well as design trajectory of a unit containing both custom hardware and software. To all the people who were part of this masters research, bedankt.

Towards the end of my masters project, I found out that I wanted to do more in this field. At this point I came in contact with my advisor, prof. dr. Stamatis Vassiliadis, who offered me a Ph.D. position. I want to thank him for giving me this opportunity and for the guidance and discussions during this research and the preparation of this dissertation. Stamatis, $\varepsilon\upsilon\chi\alpha\rho\iota\sigma\tau\acute{\omega}$. I also would like to thank dr. ir. Sorin Coţofană, with whom I shared the office for 2 years. Questions are always asked to the person nearest by, and Sorin mostly had an adequate answer. Îţi mulţumesc Sorin. Later, ir. Stephan Wong joined the group, and the scope of the research bended somewhat towards Multimedia. Stephan did a lot of literature research, which I used for both ideas and references. Stephan, hen^{v}gàn xìe.

The computer cluster used for the research at CARDIT is a heterogenous cluster of machines, which Bert Meijs administers, part of the time aided by ir. Tobias Nijweide and ir. Rogier Wolff. You provided not only me with an excellent working environment. Also, giving me the opportunity to experiment with some of the more exciting parts of system administration, such as installing Linux on not yet supported architectures and machines, provided me with an excellent distraction and perspective in computer engineering. Bert, Tobias and Rogier, Bedankt!

I want to thank Linus Torvalds, and all people working with him around the world, for creating and freely distributing Linux, which proved both a wonderful working environment and a welcome distraction of my research. Linus, kiitoksia oikein paljon.

I would like to thank my friends and family, for their interest in my work and finally I would like to thank my girlfriend, drs. Lyonne Zonneveld, for her patience, carefull proofreading and love. Lyonne, dank je wel.

Contents

Chapter 1

Introduction

Multimedia[1] processing is embedded processing especially tuned to efficiently execute multimedia applications. Accelerating multimedia processing is the subject of this dissertation, as it relates to the logic design of multimedia processors. More specifically we investigate the hardware implementation of execution units capable of improving the performance of multimedia processors. We note here that even though we discuss architectural[2] issues, the definition of a complete multimedia architecture is not the primary scope of this dissertation. While we discuss design issues for a number of operations and propose instruction formats, we do not perform performance analysis considering for example frequencies of instructions[3] and cycle count based estimations for an entire application. In essence we do not attempt to legitimize an architectural choice from performance benefit in terms of cycles saved in the entire execution of a program but rather we consider performance improvements due to the realization of a specific function in hardware rather than in software. Furthermore we do not consider issues related to the physical realization of the units we propose. We particularly focus on the execution stage of an instruction and consider a pipelined processor as described in e.g. [2] and [3].

Under the above assumptions we establish instructions that have "small" num-

[1] Multimedia is defined as the combination of diverse information including text, graphics, video, and audio. These forms of information comprised in multimedia are also referred to as *multimedia formats*.

[2] Architecture here and in the rest of this dissertation denotes the attribute of a system as seen by the programmer, i.e., the conceptual structure and functional behavior of the processor. It is distinct from the organization of the data flow and physical implementation of the processor [1].

[3] Instructions are the basic operations performed by a processor.

bers of cycle count requirements. More precisely the following is covered in general:

- consideration of the data flow of a proposed pipelined instruction and the logic design descriptions of the execution pipeline stage,
- estimation of the number of cycles required to perform the execution of the instructions we consider,
- providing estimations of the performance improvements for the proposed units when compared to other existing approaches where the same operations are executed in software.

We note that our work and proposals constitute the first basic steps for the definition of an architecture and that our considerations are necessary to further investigate the inclusion of new instructions to an architecture. Further discussion and substantiations of our choices will be found throughout the presentation of our proposals. Some general preliminary assumptions are discussed in the section 1.3.

The organization of the discussion of this chapter is as follows. First we provide some background information with some discussion on multimedia standards and some terminology definitions. Subsequently we discuss previous art and open questions followed by some general assumptions regarding architecture implementation and technologies. This chapter is concluded with a brief discussion of the organization of this dissertation.

1.1 Background

As indicated earlier, multimedia processors are processors especially tuned to perform "well" for multimedia applications. Multimedia is defined as the combination of the following forms of information: text, graphics, video, and audio – referred to as *multimedia formats* in the remainder of this dissertation. Traditionally, multimedia formats were being represented in an analog form, but currently we are observing a migration from the analog to the digital representation. The digital representation of the multimedia formats proved to have certain advantages. Examples of such advantages include easier editability and improved error resilience. While there are several advantages, the digital representation presented the scientific community and the industry with a sizeable

problem, which is the large volume of data to be processed, stored and communicated. The first solution to this problem is to increase the size of storage devices and the bandwidth of transmission lines.

The second solution, which can be combined with the first solution, is to *compress* the digital information. A substantial amount of research has been performed in this area, resulting in a multitude of compression techniques which are being used in current multimedia standards[4]. These techniques exploit the fact that redundancies exist within the digital information and that these redundancies can be removed from the digital information with the possibility to reconstruct the original digital information using the remaining (compressed) digital information. The compression techniques can be basically divided into two classes, namely *lossless compression* and *lossy compression*. In lossless compression, statistics of the digital information are used to reveal redundancies within the digital information. Decompression of lossless compressed data produces an exact identical copy of the original data. This makes lossless compression applicable to all classes of data, regardless of its contents. In lossy compression, other information besides the statistics of the digital information is used to remove redundant information. An example is using the Human Visual System (HVS) model which describes the inner working of the human eye. Using this model, information which cannot be detected by the human eye is removed from the original digital information in order to obtain higher compression ratios than lossless compression. An exact copy of the original digital information cannot be reconstructed if lossy compression is used, hence the term lossy is being used. This property makes lossy compression unusable for certain classes of data, such as text or programs. In general, most multimedia data can be compressed using lossy techniques. A large part of this dissertation focuses on multimedia architectural issues that relate to compression techniques which are used to compress digital pictures and digital video. Each of the compression techniques is highly tuned towards a certain multimedia format which is being used by multimedia applications

[4] A multimedia standard is generally defined as the collection of:

1. the definition of the structure of the bit-stream,
2. the rules on how to decode the bit-stream, and
3. a set of parameters which indicate certain constraints on the handled data.

Usually, a multimedia standard is general in the sense that multiple multimedia applications can be targeted by the standard. This is achieved by defining the parameters for each targeted application. One important fact to mention is that the encoding process is not specified in many multimedia standards.

to transfer information. In this sense, the multimedia applications[5] pose the requirements which must be met by the compression techniques. For example, the VideoCD application stores video and audio (with predetermined parameters, like resolution of the frames and sampling frequency) on a compact disc which is being read by a CD-player at the bit-rate of 150 kBytes/s. This fact already determines that certain compression techniques can not be used, because they can not deliver the compression ratio to obtain the desired bit-rate.

Multimedia standards usually target multiple multimedia applications by combining several compression techniques well-suited to meet the requirements posed by the targeted multimedia applications. Using the proposed compression techniques, the original digital information is compressed into a bit-stream which can be stored or transmitted. To obtain a copy of the original digital data, which can be an exact copy of the original or a copy closely resembling the original, the inverse of the compression technique is used. The process of obtaining the compressed bit-stream is called the *encoding* process and the inverse is called the *decoding* process. Most multimedia standards only define the decoding process and the structure of the bit-stream and leave the encoding process undefined on purpose. Because of this fact, future results of continued research into faster and more efficient algorithms can be used in the encoding process. Additionally, manufacturers are currently enabled to implement their own algorithms to distinguish their products from other manufacturers. However, extreme differences after decoding due to the different encoding algorithms must be avoided. To this end, a rough guideline is given containing several error measures which must not be exceeded.

As stated in the previous paragraph, the encoding process is often not defined in a multimedia standard and only rough guidelines are given how to implement the encoding process. Therefore, different implementations[6] are possible for the encoding process enabling manufacturers to differentiate their own products from other manufacturers. There are basically two approaches in how to implement multimedia standards. One approach is to implement the entire standard in software run on general-purpose processors. Using this approach, the implementation is fast in adapting to changes, but usually the performance is not good enough. The second approach is to implement the multimedia

[5] A multimedia application is defined as any digital system which uses one or more multimedia formats to transport information and which performs the associated functions to accomplish this goal. Usually, multimedia standards are used to handle the multimedia formats used by the application.

[6] A multimedia implementation is defined as a structure for realization and the realization itself in hardware or software for handling of multimedia formats specified by multimedia standards.

standards in hardware using Application Specific Integrated Circuits (ASICs). Using this approach, the highest achievable performance is possible, but usually adaptation to changes requires new designs which is a time-consuming and expensive process. This led to a hybrid solution between the two previous approaches, namely programmable Digital Signal Processors (DSPs) or Application Specific Instruction Set Processors (ASIPs). These processors are programmable as general purpose processors, and also have some specialized hardware execution units which can perform more complex, application-range specific, operations. This thesis studies such specialized execution units which may help to enrich the instruction set architectures of such processors to support multimedia applications. Additional information regarding these issues will be discussed found in the following section.

Before continuing on describing related work and open questions we briefly introduce some terminology that is found in multimedia and used throughout the dissertation. The following are some commonly used terms and standards:

- **Pixel or Pel:** The smallest element of any system that represents data in the form of 2-D arrays of visual information, e.g. on a video screen.
- **Resolution:** The fineness of the detail represented by any form of media: audio, images or video. In case of image and video, the resolution is defined by the number of pixels per picture.
- **Luminance:** The intensity, or brightness component, of an electronic image.
- **Chrominance:** The color portion of the information contained in an electronic color image.
- **Multimedia Standards:** Examples of multimedia standards are: JPEG, MPEG-1, MPEG-2, and H.261. Table 1.1 shows which multimedia applications are targeted by these standards. The JPEG standard is used to compress digital continuous-tone still-pictures using a wide variety of compression techniques while retaining good picture quality. The MPEG-1 and MPEG-2 standards are used to compress video and audio information. These standards provide video quality which is equal to or better than VHS tapes. The H.261 standard is very similar to the two MPEG standards, but targets applications like video-telephony (using regular telephone lines). This requires a very low bit-rate of the compressed bit-stream. Two newer standards are the Portable Network Graphics (PNG) and the Multiple Network Graphics (MNG) stan-

dards. These two provide lossless compression for images respectively sequences of images.

Multimedia standard	Targeted multimedia application(s)
JPEG	Lossy compression of photographic images
MPEG-1	VideoCD
MPEG-2	DVD digital broadcast of video HDTV
H.261	Video conferencing Video telephony
PNG	Lossless compression of images and graphics
MNG	Lossless compression of sequences of images or graphics

Table 1.1: Examples of multimedia applications targeted by different multimedia standards.

In the following sections we discuss the previous art and open questions, and we describe our assumptions that will be observed throughout the study.

1.2 Related work and open questions

In order to improve the processing of multimedia applications, four types of processors have been investigated, namely:

Specialized multimedia standard processors: In this class of processors a specific standard such as MPEG-2 is assumed, and for such a standard a processor is designed that uniquely performs this standards requirements. There are several processors available that assume this approach. Examples of such processors are for example MPEG-2 Decoders [4, 5].

Stream based video processors: This class of processors performs various operations on streaming video. These processors are not targeted to a specific standard, but rather to a specific application domain, such as video-recorders and television sets. The programmability of these processors is limited, in that they can only perform a predefined set of operation. Typical operations executed by these processors are image en-

hancement, picture-in-picture and on-screen-display. Examples of these processors can be found in [6, 7, 8, 9, 10, 11, 12, 13, 14] and [15].

Specialized processors for augmented multimedia processing: In this class of processors, programmability is assumed and no restrictions to standards are imposed. That is, the processing follows the usual general purpose paradigm of programmability and the instructions set definition. These C-programmable processors are general purpose processor with extensions that make them particularly suited for media processing. For the Philips Trimedia architecture and processors [16] these are fine-grain extensions to the VLIW processor as well as coarse-grain coprocessors for e.g. VLD decoding. Another example of this class of processors is the Texas Instruments Multimedia Video Processor (MVP) [17].

General purpose processors: The family of general purpose machine is extended with coprocessing capabilities to improve the performance of multimedia applications. This approach follows the traditional extension oriented processing. That is, a general purpose processor architecture is extended with new architectural features in order to improve a certain application domain, which allows the design of coprocessors specialized for the considered application. Examples of such extensions include the floating point and vector extension of general purpose computing. Examples of this type of extensions with respect to multimedia include:

- Intel MMX(Multi Media eXtensions) [18, 19],
- ALPHA MVI (Motion Video Instruction extensions) [20, 21, 22],
- Sun VIS (Visual Instruction Set) [23, 24, 25] and
- MIPS DME (Digital Media Extensions) [26].

All these classes of processors provide improvements in multimedia processing and there is discussion which of the approaches should be followed. In this dissertation we do not discuss nor decide which of these approaches is a better multimedia solution. Rather we propose mechanisms that provide some improvements to all possible types of processors by proposing new execution units.

In order to determine the scope of our discussion we consider some of the open questions left by the previous proposals. Processors specialized for a specific standard assume fixed functions for all processing including the execution

hardware. This approach is rather limited to a single application (imposed by the standard) and it can hardly follow the multimedia requirements in their entirety. For example, an MPEG-2 specialized processor has no capabilities to perform the functions required by other standards or applications. Furthermore, advancement in standards can not be followed, except by redesigning the processor. Consider the following scenario as an example. Assume a specific motion estimator has been designed using the Sum of Absolute Differences operation. Given that the motion estimation algorithm is implemented in hardware, no change can occur even when a new, possibly better, algorithm also containing the sum of absolute differences function is proposed and desirable to implement. The previous discussion leaves open the following question:

- Can frequently used functions be parametrically specified and provided with appropriate interfacing so that standard improvements can be followed with minimal design effort and no redefinition of functions?

The last three classes of multimedia processors partially resolve this open question by providing programmability and new instructions. There are several examples of this approach. For example, the sum of absolute differences discussed previously, has been implemented as the PERR instruction in Alpha MVI [22] and as the newly added PSADBW instruction in Intel MMX [27].The open question is resolved in part. While an appropriate interfacing is given (all programmable processor instructions have this property) the proposed instructions do not parameterize their implementation resulting in a potential bottleneck for their performance. To further elaborate, consider the sum of absolute differences function. If we assume a word parallel architecture based on 64-bit registers and if we assume a "word" is a byte then at most 8 "words" can be processed by a sum of absolute differences function instruction. While this is an improvement, it is also a bottleneck. It implies a maximum parallelism of 8 while the sum of absolute differences operates on blocks of 16 by 16 pixels. An additional open question imposed by this approach is that to the best of our knowledge[7], no specific direct implementations are reported that require minimum number of cycles necessary.

[7]Recently, a new instruction has been added in the Intel MMX architecture and possibly a new implementation has been reported in the direction we propose [27]. Following the discussion of the microprocessor report [27] we conclude that the described approach is not the best possible. To determine the sum of absolute differences it is stated that their implementation performs the following three micro-operations:

- determine for each pair of inputs the difference,
- take the absolute value of this difference,
- take the sum of these absolute differences.

An additional open issue is phrased by the following:

- In determining the instruction set it is of interest to include the highest number of possible functions that are related to each other, so that a unit may be implemented that allows a number of functions to be performed at approximately the same cycle time with the least hardware expense.

In general purpose computing this problem has been addressed extensively[8] (and plus or minus completely resolved). While multimedia instruction sets have added a number of instructions that follow the general purpose paradigm, the issue is far from a final settlement and continues to constitute an interesting open question phrased by the following:

- Which instructions can be added to the instruction set that are implementable with small hardware additions to the existing execution unit or that can be implemented as a new hardware unit capable of performing this new set of instructions?

We resolve in part[9] this open question by proposing:

- new instructions that can be added to existing units, e.g. a multiplier unit, with small additional cost and by proposing
- a separate "ALU like unit" capable of performing specialized multimedia functions, with cycle times comparable to general purpose ALU cycles.

The additions we introduce are meant to be the initiation of new unit design and the proposed research direction is to solidify this multimedia unit(s) by adding instructions in the future with similar hardware requirements.

Before we proceed with the further elaborations of our proposal we need to discuss a number of parameters. We dedicate the next section to the description of our assumptions.

As we have shown in [28] and describe in Chapter 3 of this dissertation, a better approach exists to compute this function.

[8] See for example ALU related operations for fixed point units and adder or multiply related operations in floating point instruction sets.

[9] In part here means that while we consider a number of multimedia formats we did not perform an exhaustive search. Basically we add-on on an existing art without pretending to examine all possible situations as such an attempt is rather impossible.

1.3 Assumptions

In order to determine new architectural features, a number of assumptions has to be put in order. The reason for such assumptions lies mostly in the difficulty to establish "universal" parameters for the evaluation and the acceptance of a proposal. Our assumptions for this dissertation are implied by the computer engineering general paradigm relating to the following factors:

- architecture,
- implementation,
- realization.

As suggested earlier we assume an add-on architectural approach, in that we assume a general purpose scenario rather than an application specific one. This assumption implies that our descriptions will strictly follow processing behavior that is programmable and possibly usable by multiple multimedia application requirements. Our assumptions regarding implementation rely on pipelined, possibly super-scalar machine organizations, which follow general logic design techniques, e.g. boolean logic instead of threshold logic [29]. Furthermore we assume "usual" pipeline structures. That is, we assume fixed point, non memory interfacing instructions, a pipeline structure of fetching, decoding, execute and write cycles. Additional cycles are assumed for load/store instructions[10] and other complex instructions such as multiply and divide.

Cycle related requirements such as critical paths, are imposed by well known algorithms for the design of general purpose units for addition related operations. That is, our cycle time assumptions mostly relate to the implementation of:

- ALU operations incorporating the design of parallel adders. See for example [30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40] and
- multipliers, dividers and other "multi-operand addition" functions, see for example [39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53].

We do not consider serial implementations as for example described in [54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67] as such implementations are not used commonly in general purpose computers.

[10]Note that given the fact that we are concerned mostly with the execution unit, very little is discussed about memory interfacing as such interfacing is not part of our concern in this dissertation.

Our technological assumptions are dictated by the implementation assumptions and the currently available technologies. In particular we assume that the technology is a gate array type of technology. For such technology we imply the availability of a library with gates called a bookset performing prespecified functions. The technology in which the gates are implemented is assumed to be CMOS. The assumed CMOS gates (bookset) consist of: a two-way XOR and XOR-INVERT; up to a three-way AND, OR, AND-INVERT, and OR-INVERT; and up to $\pm$ $3x4$ AO books where + indicates the AND-OR function and $-$ indicates the AND-OR-INVERT function. This is the same bookset assumed in [36] for the design of high speed, non-custom binary adders and it is also the bookset assumed in [68] for the design of high speed complex ALU operations. We further assume that every book in this library constitutes one stage of delay. While these assumptions are restricted to a particular technology, they do not limit the applicability of our discussion since such a bookset is common in currently available technologies and it is extendable to other technologies having similar characteristics or equivalent functional power within their booksets [36].

1.4 Framework of the dissertation

The contributions of this dissertation are discussed in five chapters. More specifically the following is discussed.

- In Chapter 2, entitled **Add Multiply Add**, we investigate complex (compound) instructions which could provide performance improvement for embedded systems and multimedia applications if implemented in hardware. In particular we show that a number of arithmetic expressions of certain forms occur frequently in embedded system applications. These arithmetic expressions have certain data-dependencies among the individual (fine grain) operations that constitute the arithmetic expression. A number of these expressions can be captured by an unique expression: $(A \pm B) * C \pm D$, or Add-Multiply-Add. Consequently, we propose two schemes for the implementation of such an expression, assuming two's complement number representation. The first scheme, the Inversion Selection technique, is a direct multiplication scheme for array and/or parallel implementations. The second scheme, the Half-Adder technique, performs a modified Booth Encoding and is also suitable for parallel design. The end result of the investigation in this chapter is that the Add-Multiply-Add expression can be computed in two machine

cycles (comparable to the machine cycles of a two cycle multiply), suggesting no more cycle time than the multiply instruction alone. This in turn suggests that the expressions we consider can experience a substantial benefit from the data dependency collapsing unit we propose. More specifically, the expression will require four machine cycles using standard hardware and three machine cycles using multiply-add hardware. This suggests that a speed-up between 1.5 and 2 will be achieved if the Add-Multiply-Add unit we propose is implemented in hardware.

- In Chapter 3, entitled **Sum and Mean of Absolute Differences**, we investigate the Sum Absolute Difference (SAD) operation, an operation frequently used by a number of algorithms for digital motion estimation. For such an operation, we propose a single vector instruction that can be performed (in hardware) on an entire block of data in parallel. We investigate possible implementations for such an instruction. Assuming a machine cycle that is comparable to the cycle of a two cycle multiply, we show that for a block of 16x1 or 16x16, the SAD operation can be performed in 3 or 4 machine cycles respectively. The proposed implementation operates as follows: first we determine in parallel which of the operands is the smallest in a pair of operands. Second we compute the absolute value of the difference of each pairs by subtracting the smallest value from the largest and finally we compute the accumulation. The operations associated with the second and the third step are performed in parallel resulting in a multiply (accumulate) type of operation. Our approach covers also the Mean Absolute Difference (MAD) operation at the exclusion of a shifting (division) operation.

- In Chapter 4, entitled **Paeth Prediction and Coding**, we describe an execution unit capable of computing the Paeth predictor, as used in the Portable Network Graphics (PNG) standard. PNG is a rather new, lossless compression method for real-world pictures. It features five prediction schemes, of which the modified Paeth predictor is the most computational intensive. This chapter focuses on a hardware implementation of the Paeth predictor. We propose a hardware Paeth codec, capable of computing three different quantities:
  - the Paeth predictor of three inputs,
  - the difference of the current pixel and the Paeth predictor of the other inputs (Coding), and
  - the sum of the coded input and the Paeth predictor of the other three inputs (Decoding).

The Paeth unit computes these values within two cycles, where a cycle is comparable to the cycle of a two cycle multiply. Depending on the mode of operation, the proposed mechanism produces the predictor or the (de/en)-coded pixel value.

- In Chapter 5, entitled **Median, Max, Min and Mean** we introduce an extension of the Paeth unit by which it can additionally compute the Median of three inputs. This median is used in video-deinterlacing, which is needed for displaying normal (interlaced) video on a non-interlaced computer screen or a modern, high-end television set. We will further extend the Paeth logic, so that is can also compute the maximum and minimum of the three inputs. Furthermore, we introduce an extension of the Add-Multiply-Add unit, described in Chapter 2 by which the Add-Multiply-Add unit can compute the Mean of three inputs. The overall direction of this chapter is to introduce new instructions and show that they can be of advantage when compared to their software equivalent.

- In Chapter 6, entitled **Putting it all together**, the data-formats of the input and output of our proposed unit design are described. We also describe how the units can be integrated in one single unit implementation. Finally we propose a data-flow capable of performing all 32 instructions.

In the conclusion chapter we briefly discuss our findings and summarize the contributions of our proposals. We conclude the discussion by indicating some possible directions for future research.

Chapter 2

Add-Multiply-Add

It is widely accepted that true data dependencies [2] constitute one of the major obstacles for the improvement of speed in the computer based computational paradigm. True data dependencies occur when the execution of an instruction has to wait, because the instruction requires operands that are the result of the execution of a previous instruction. In the recent past it has been shown that some important classes of true data dependencies for the general purpose computational paradigm can be resolved resulting in a substantial gain of performance [69, 70, 71, 72, 73].

A major open question regarding special purpose multimedia computations, and embedded systems in general, is phrased by the following: while special purpose engines execute code that can possibly substantially benefit from the resolution of data dependencies techniques, thus far little has been achieved in such an area for this class of computations. There is evidence suggesting that current research is headed to the direction of investigating such an open question. For example, the application analysis presented by F. Onion et al. [74] for applications such as 2D convolution, filters, FFT, DCT, histogram flattening, edge detection, etc., shows that more elaborate data dependency collapsing hardware is required to resolve most of the true data dependencies.

More in particular the investigation in [74] has revealed that the following expressions (assuming the compiler can expose them) appear frequently in the benchmarks they have considered.

- Add-Add
- Add-Multiply

- Multiply-Add
- Add-Multiply-Add
- Multiply-Add-Add

In this chapter we resolve in part the open question regarding the expressions which Onion et al. [74] have uncovered. We consider fixed point number representations and combine four of the above five exposed expressions in a single instruction, for which we propose a hardware implementation. We do not consider floating point notations which are left as a further research topic.

Our investigation strongly suggests that the add-multiply-add expression we consider, in addition to the covering most of the expressions or operations revealed in [74], can potentially be implemented using a parallel hardware organization and potentially be executed within two machine cycles. The previous conjecture is put in place by showing that the partial product matrix associated with the expression requires no more than $n + 2$ rows, n being the number of bits of the input-values, which will most likely require no more cycle time than a fixed point multiplier in most implementations. We propose two schemes for the computation of the add-multiply-add operation. The first scheme, the Inversion Selection technique, is a direct multiplication scheme for array and/or parallel implementations. The second scheme, the Half-Adder technique, performs a modified Booth Encoding and is also suitable for parallel design.

The organization of this chapter is as follows. In the section to follow we give some background information on the multiplication of two two's complement numbers. In Section 2.3 the first approach will be presented, the Inversion Selection Technique and in section 2.4 the second approach, the Half Adder Encoding is described. Section 2.5 concludes this chapter.

2.1 Preliminaries

The general form of the expression we investigate in this chapter is $(A \pm B) * C \pm D$. The four operands, A, B, C and D are assumed to be n-bit numbers in two's complement notation. The result is a $(2n + 1)$-bit number, also in two's complement notation. Figure 2.1 gives a graphical representation of the unit. Table 2.1 shows that a specific operation is carried out with the setting of the controls denoted as op_1, op_2. Additionally, by proper setting of the operands, a multiplicity of other operations, e.g., multiply, add, add-add can also be computed. In the next two sections we present two implementation schemes for the

expression we consider. In the final section we conclude the presentation with some remarks. We first present the two's complement notation and a modification of the Baugh-Wooley algorithm for multiplying two two's complement numbers.

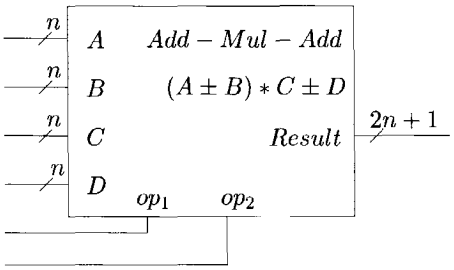


Figure 2.1: The outside view of the Add-Multiply-Add Unit.

$oper_1$	$oper_2$	$(A\ oper_1\ B) * C\ oper_2\ D$	op_1	op_2
+	+	$(A+B)*C+D$	0	0
+	−	$(A+B)*C-D$	0	1
−	+	$(A-B)*C+D$	1	0
−	−	$(A-B)*C-D$	1	1

Table 2.1: The operations that can be performed by the Add-Multiply-Add Unit.

2.2 Two's Complement Multiplication

Let X and Y be two n-bit two's complement numbers, where X_0 is the least significant bit and X_{n-1} is the most significant bit, then the values of X and Y are defined as:

$$X = -x_{n-1} * 2^{n-1} + \sum_{j=0}^{n-2} x_j * 2^j \tag{2.1}$$

$$Y = -y_{n-1} * 2^{n-1} + \sum_{i=0}^{n-2} y_i * 2^i \tag{2.2}$$

and their product XY is equal to:

$$XY = (-x_{n-1} * 2^{n-1} + \sum_{j=0}^{n-2} x_j * 2^j) \quad (2.3)$$

$$* \quad (-y_{n-1} * 2^{n-1} + \sum_{i=0}^{n-2} y_i * 2^i) \quad (2.4)$$

This product can be written as a sum of four terms: $Term_1$, $Term_2$, $Term_3$ and $Term_4$, which are defined as follows:

$$\begin{aligned} Term_1 &= (-x_{n-1} * 2^{n-1}) * (-y_{n-1} * 2^{n-1}) \\ Term_2 &= (-x_{n-1} * 2^{n-1}) * \left(\sum_{i=0}^{n-2} y_i * 2^i\right) \\ Term_3 &= (-y_{n-1} * 2^{n-1}) * \left(\sum_{j=0}^{n-2} x_j * 2^j\right) \\ Term_4 &= \left(\sum_{j=0}^{n-2} x_j * 2^j\right) * \left(\sum_{i=0}^{n-2} y_i * 2^i\right) \end{aligned} \quad (2.5)$$

The four terms can be rewritten as:

$$\begin{aligned} Term_1 &= x_{n-1}y_{n-1} * 2^{2n-2} \\ Term_2 &= -2^{n-1} * \sum_{i=0}^{n-2} x_{n-1}y_i * 2^i \\ Term_3 &= -2^{n-1} * \sum_{j=0}^{n-2} y_{n-1}x_j * 2^j \\ Term_4 &= \sum_{j=0}^{n-2}\sum_{i=0}^{n-2} x_j y_i * 2^{j+i} \end{aligned} \quad (2.6)$$

The result of the multiplication of two n-bit two's complement numbers is a $2n$-bit two's complement number. It can be observed that $Term_2$ and $Term_3$ are negative. This is detrimental to a hardware implementation as it is required to perform element subtractions. While it is possible to be achieved, it requires multiple cell designs [40]. Baugh and Wooley [75] introduced an algorithm which requires no element subtraction. Given that in our investigation we will

also introduce an algorithm that contains no element subtraction we discuss for background purposes a modification of the Baugh-Wooley algorithm in this section.

Baugh and Wooley [75] indicate that instead of subtracting a partial product, the negation of this partial product can be added. The negative partial product rows can be converted to positive elements rows via the addition of 2^{2n} and truncation. As the result of the multiplication is a $2n$ bit result, the 2^{2n} constant is truncated in the final steps and therefore it doesn't influence the result.

More in particular, the negative elements elimination is achieved as follows. The negative $Term_2$

$$Term_2 = -2^{n-1} \sum_{i=0}^{n-2} x_{n-1} y_i * 2^i \tag{2.7}$$

can be rewritten to a positive term by adding 2^{2n}:

$$\begin{aligned}
2^{2n} + Term_2 &= 2^{2n} - 2^{n-1} \sum_{i=0}^{n-2} x_{n-1} y_i * 2^i \\
&= 2^{n-1} \left(2^{n+1} - \sum_{i=0}^{n-2} x_{n-1} y_i * 2^i \right) \\
&= 2^{n-1} \left(2^n + 2^{n-1} + 1 + \sum_{i=0}^{n-2} 2^i - \sum_{i=0}^{n-2} x_{n-1} y_i * 2^i \right) \\
&= 2^{n-1} \left(2^n + 2^{n-1} + 1 + \sum_{i=0}^{n-2} (2^i - x_{n-1} y_i * 2^i) \right) \\
&= 2^{n-1} \left(2^n + 2^{n-1} + 1 + \sum_{i=0}^{n-2} (1 - x_{n-1} y_i) * 2^i \right) \\
&= 2^{n-1} \left(2^n + 2^{n-1} + 1 + \sum_{i=0}^{n-2} \overline{x_{n-1} y_i} * 2^i \right)
\end{aligned} \tag{2.8}$$

This means that the negative $Term_2$ can be replaced with the following positive equivalent:

$$Term_2 = 2^{n-1} \left(2^n + 2^{n-1} + 1 + \sum_{i=0}^{n-2} \overline{x_{n-1} y_i} * 2^i \right) \tag{2.9}$$

$Term_3$ can be treated in the same way:

$$Term_3 = 2^{n-1}\left(2^n + 2^{n-1} + 1 + \sum_{j=0}^{n-2} \overline{y_{n-1}x_j} * 2^j\right) \tag{2.10}$$

From this point, our approach differs from Baugh and Wooley's. While their goal was to make a multiplication structure with the highest possible regularity, they had to eliminate the NAND function required for $Term_2$ and $Term_3$. Our goal is to minimize the number of terms to be added together. Therefore, we try to eliminate as much constants as possible by adding them together beforehand.

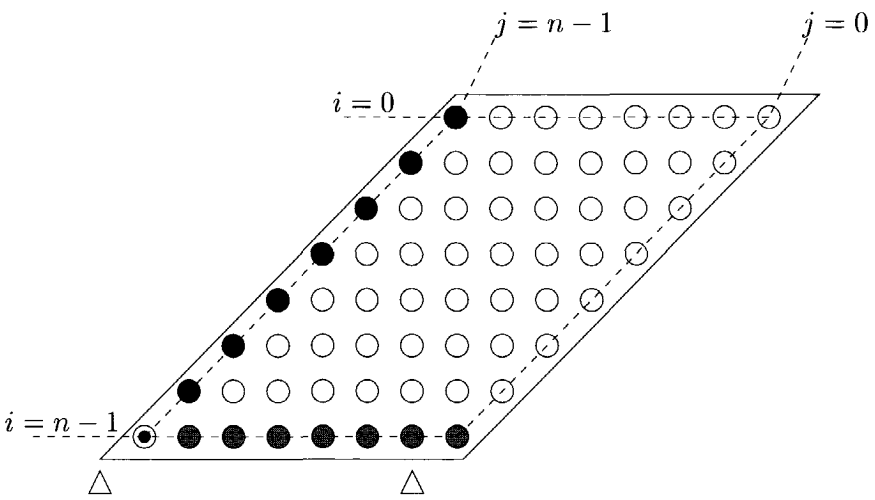
If we add the constant part of 2.9 and 2.10, we get:

$$\begin{aligned} Term_2\_Constant &= 2^{n-1}\left(2^n + 2^{n-1} + 1\right) \\ Term_3\_Constant &= 2^{n-1}\left(2^n + 2^{n-1} + 1\right) \quad + \\ \hline Term_{2,3}\_Constant &= 2^{2n} + 2^{2n-1} + 2^n \end{aligned} \tag{2.11}$$

where 2^{2n} is the carry-out of the addition, which can be ignored.

The end effect is that we have to invert two times $n-1$ bits, and add two single hot ones: on position n and on position $2n-1$. Figure 2.2 shows a graphical representation of this multiplication. The white 'dots' and the 'dot' in the lower left corner are the normal multiplication bits, $x_j y_i$. The black and gray 'dots' are the inverted partial products, $\overline{x_j y_i}$ and the two hot-ones are represented as two triangles. As we can see from the figure, the number of rows to be reduced equals n, the number of bits of the input numbers.

In this section we discussed the well known Baugh-Wooley algorithm [75]. This algorithm provides the capability of eliminating negative elements for the multiplication matrix, in order to avoid irregularities from these negative elements in the the partial product reduction hardware. In next section we describe two techniques, called inversion selection and half adder encoding for computing $(A \pm B) * C \pm D$. Both of them consist of two steps and use similar derivations of the Baugh-Wooley algorithm discussed previously. In the first step the partial products from $(A \pm B) * C$ are computed. Those partial products are added together with D to produce $(A \pm B) * C \pm D$ in the second step. The aim of both schemes is to provide a low-latency, hardwired unit with relatively modest hardware requirements, that computes $(A \pm B) * C \pm D$ in the same time required by a normal multiply.



- ◉ $Term_1$, which is inverted twice
- ● $Term_2$, which is inverted
- ● $Term_3$, which also is inverted
- ○ $Term_4$, the normal multiplication bits
- △ The combined constants

Figure 2.2: Graphical representation of a normal multiplication.

2.3 Inversion-selection

The inversion-selection-technique is based on the fact that in the two's complement notation $-X$ is almost equal to $\overline{X}$ (to be precise: $-X = \overline{X} + 1$).

We rewrite the $(A \pm B) * C$ so that we compute it with $A * C + B * \overline{C}$ (or $A * C + \overline{B} * \overline{C}$), in which the bits of C choose between passing the bits of A or $B(\overline{B})$.

As an example, we will compute $(A - B) * C$. We can do so by rewriting it to:

$$(A - B) * C = A * C - B * C \tag{2.12}$$
$$= A * C + B * (-C) \tag{2.13}$$

In this last equation, we use the two's complement rule in that we rewrite $-C$ as $\overline{C} + 1$. The hot one is not added directly in this case. This makes

$$(A - B) * C = A * C + B * (\overline{C} + 1) \tag{2.14}$$

$$(A - B) * C = A * C + B * \overline{C} + B \tag{2.15}$$

This means that for each bit of C, we have to add either A (if the bit of C is one) or B (if it is zero) to the partial product matrix, in stead of both or neither of them. C is used as a select signal to choose between them.

Normally, if a certain bit of C is one, we would have to feed A and $-B$ to the partial product matrix on that position. If that bit would have been zero, we would have to add nothing. The maximum number of partial product rows would be equal to two times the number of bits of C. In our scheme, each row is always filled with either A or B, and the number of rows equals the number of bits of C and one extra row for B.

We will now rewrite the four instances of $(A \pm B) * C \pm D$ so that they fit this scheme.

Depending on the operation, there are four instances of the expression $(A\ oper_1\ B)\ *\ C\ oper_2\ D$. These are shown below:

1) $oper_1 = +$ and $oper_2 = +$ $\qquad (op_1 = 0; op_2 = 0)$

$$\begin{aligned}(A + B) * C + D &= A * C + B * C + D \\ &= A * C + (-B) * (-C) + D \\ &= A * C + (\overline{B} + 1) * (\overline{C} + 1) + D \\ &= A * C + \overline{B} * \overline{C} + \overline{B} + \overline{C} + D + 1 \end{aligned} \tag{2.16}$$

2) $oper_1 = +$ and $oper_2 = -$ $\qquad (op_1 = 0; op_2 = 1)$

$$\begin{aligned}(A + B) * C - D &= A * C + B * C - D \\ &= A * C + (-B) * (-C) - D \\ &= A * C + (\overline{B} + 1) * (\overline{C} + 1) + (\overline{D} + 1) \\ &= A * C + \overline{B} * \overline{C} + \overline{B} + \overline{C} + \overline{D} + 2 \end{aligned} \tag{2.17}$$

3) $oper_1 = -$ and $oper_2 = +$ $\qquad (op_1 = 1; op_2 = 0)$

$$\begin{aligned}(A - B) * C + D &= A * C - B * C + D \\ &= A * C + B * (-C) + D \\ &= A * C + B * (\overline{C} + 1) + D \\ &= A * C + B * \overline{C} + B + D \end{aligned} \tag{2.18}$$

4) $oper_1 = -$ and $oper_2 = -$ $\qquad (op_1 = 1; op_2 = 1)$

$$\begin{aligned}(A-B)*C-D &= A*C-B*C-D \\ &= A*C+B*(-C)-D \\ &= A*C+B*(\overline{C}+1)+(\overline{D}+1) \\ &= A*C+B*\overline{C}+B+\overline{D}+1 \qquad (2.19)\end{aligned}$$

All the instances are summarized in Table 2.2. In the Table op_1 and op_2 (the operations) are the control signals to the hardware to indicate what needs to be computed. Equal to zero implies addition and equal to one implies subtraction.

The four instances can be combined into a single expression as follows:

$$\begin{aligned}&(A\ oper_1\ B)*C\ oper_2\ D = \\ &A*C+\widehat{B}*\overline{C}+\widehat{B}+\widehat{C}+\widehat{D}+\ \overline{op_1}\ +\ op_2 \qquad (2.20)\end{aligned}$$

in which $\widehat{B} = B \oplus \overline{op_1}$, $\widehat{C} = \overline{C}\ \&\ \overline{op_1}$ and $\widehat{D} = D \oplus op_2$. The computation

op_1	op_2	expression	expression used for the Inversion Selection Technique
0	0	$(A+B)*C+D$	$A*C+\overline{B}*\overline{C}+\overline{B}+\overline{C}+D+1$
0	1	$(A+B)*C-D$	$A*C+\overline{B}*\overline{C}+\overline{B}+\overline{C}+\overline{D}+2$
1	0	$(A-B)*C+D$	$A*C+B*\overline{C}+B+\quad D$
1	1	$(A-B)*C-D$	$A*C+B*\overline{C}+B+\quad \overline{D}+1$

Table 2.2: The possible operations used for the Inversion-Selection Technique.

of the expression 2.20 can be split in two parts. The first part is generating the partial products and correction-terms (correction-terms are needed to avoid negative elements). The second part is summing up the partial products and the correction-terms. We first concentrate on producing the partial products.

First we compute $A*C+\widehat{B}*\overline{C}$. This is achieved by two multiplications and an addition. It can immediately be noted that if a certain bit in C is 1 then the corresponding bit in $\overline{C}$ is 0, and vice versa. This implies that in generating the partial products certain eliminations of bits can be carried out. This indicates that a combined matrix can be generated directly and eliminates the hardware requirements for one of the two multiplications and the addition.

The same scheme that is used for the multiplication of X and Y, (formulas 2.5 through 2.11) can be used for computing $A*C+\widehat{B}*\overline{C}$. We simply compute both terms of the sum and add them together. The four terms, which were used in Section 2.2 can be computed as follows:

$$
\begin{aligned}
Term_1 &= (-c_{n-1} * 2^{n-1}) * (-a_{n-1} * 2^{n-1}) \\
&+ (-\overline{c_{n-1}} * 2^{n-1}) * (-\widehat{b_{n-1}} * 2^{n-1}) \\
&= c_{n-1} * a_{n-1} * 2^{2n-2} + \overline{c_{n-1}} * \widehat{b_{n-1}} * 2^{2n-2} \\
&= (c_{n-1} * a_{n-1} + \overline{c_{n-1}} * \widehat{b_{n-1}}) * 2^{2n-2} \\
&= (c_{n-1} \& a_{n-1} | \overline{c_{n-1}} \& \widehat{b_{n-1}}) * 2^{2n-2} \qquad (2.21)
\end{aligned}
$$

The $*$ between c and a and between $\overline{c}$ and $\widehat{b}$ (bitwise multiplication) is equivalent to the AND (&) operator, and the $+$ (arithmetic addition) is equivalent to the OR (|) operator, because either c_{n-1} or $\overline{c_{n-1}}$ is equal to zero.

$$
\begin{aligned}
Term_2 &= (-a_{n-1} * 2^{n-1}) * \left(\sum_{i=0}^{n-2} c_i * 2^i \right) + (-\widehat{b_{n-1}} * 2^{n-1}) * \left(\sum_{i=0}^{n-2} \overline{c_i} * 2^i \right) \\
&= -2^{n-1} * \sum_{i=0}^{n-2} a_{n-1} * c_i * 2^i - 2^{n-1} * \sum_{i=0}^{n-2} \widehat{b_{n-1}} * \overline{c_i} * 2^i \\
&= -2^{n-1} * \sum_{i=0}^{n-2} \left(a_{n-1} * c_i * 2^i + \widehat{b_{n-1}} * \overline{c_i} * 2^i \right) \\
&= -2^{n-1} * \sum_{i=0}^{n-2} (c_i \& a_{n-1} | \overline{c_i} \& \widehat{b_{n-1}}) * 2^i \qquad (2.22)
\end{aligned}
$$

$$
\begin{aligned}
Term_3 &= (-c_{n-1} * 2^{n-1}) * \left(\sum_{j=0}^{n-2} a_j * 2^j \right) + (-\overline{c_{n-1}} * 2^{n-1}) * \left(\sum_{j=0}^{n-2} \widehat{b_j} * 2^j \right) \\
&= -2^{n-1} * \sum_{j=0}^{n-2} c_{n-1} * a_j * 2^j - 2^{n-1} * \sum_{j=0}^{n-2} \overline{c_{n-1}} * \widehat{b_j} * 2^j \\
&= -2^{n-1} * \sum_{j=0}^{n-2} \left(c_{n-1} * a_j * 2^j + \overline{c_{n-1}} * \widehat{b_j} * 2^j \right) \\
&= -2^{n-1} * \sum_{j=0}^{n-2} (c_{n-1} \& a_j | \overline{c_{n-1}} \& \widehat{b_j}) * 2^j \qquad (2.23)
\end{aligned}
$$

$$
\begin{aligned}
Term_4 &= \left(\sum_{j=0}^{n-2} a_j * 2^j\right) * \left(\sum_{i=0}^{n-2} c_i * 2^i\right) \\
&+ \left(\sum_{j=0}^{n-2} \widehat{b_j} * 2^j\right) * \left(\sum_{i=0}^{n-2} \overline{c_i} * 2^i\right) \\
&= \sum_{j=0}^{n-2}\sum_{i=0}^{n-2} a_j * c_i * 2^{i+j} + \sum_{j=0}^{n-2}\sum_{i=0}^{n-2} \widehat{b_j} * \overline{c_i} * 2^{i+j} \\
&= \sum_{j=0}^{n-2}\sum_{i=0}^{n-2} (a_j * c_i + \widehat{b_j} * \overline{c_i}) * 2^{i+j} \\
&= \sum_{j=0}^{n-2}\sum_{i=0}^{n-2} (c_i \& a_j | \overline{c_i} \& \widehat{b_j}) * 2^{i+j} \qquad (2.24)
\end{aligned}
$$

It should be noted that the terms involved in the computation of $A * C + \widehat{B} * \overline{C}$ are similar to the terms involved in the computation of $X * Y$. The main difference is the use of a 2-input MUX instead of a 2-input AND (using the bit of C to select between A and $\widehat{B}$).

As the addition of negative terms introduces irregularities in the partial product reduction logic, we are are interested in eliminating the negative elements ($Term_2$ and $Term_3$). We modify these terms to positive using the fact that all computations are done in two's complement. Because the result is $2n+1$ bits wide, $N^- = 2^{2n+1} - N$.

$$
\begin{aligned}
Term_2 &= 2^{2n+1} - 2^{n-1} * \sum_{i=0}^{n-2} (c_i \& a_{n-1} | \overline{c_i} \& \widehat{b_{n-1}}) * 2^i \\
&= 2^{n-1} * \left(2^{n+2} - \sum_{i=0}^{n-2} (c_i \& a_{n-1} | \overline{c_i} \& \widehat{b_{n-1}}) * 2^i\right) \\
&= 2^{n-1} * \left(\sum_{i=n-1}^{n+1} 2^i + \sum_{i=0}^{n-2} 2^i + 1 - \sum_{i=0}^{n-2} (c_i \& a_{n-1} | \overline{c_i} \& \widehat{b_{n-1}}) * 2^i\right) \\
&= 2^{n-1} * \left(\sum_{i=n-1}^{n+1} 2^i + 1 + \sum_{i=0}^{n-2} (1 - c_i \& a_{n-1} | \overline{c_i} \& \widehat{b_{n-1}}) * 2^i\right) \\
&= 2^{n-1} * \left(\sum_{i=n-1}^{n+1} 2^i + 1 + \sum_{i=0}^{n-2} \overline{(c_i \& a_{n-1} | \overline{c_i} \& \widehat{b_{n-1}})} * 2^i\right) \qquad (2.25)
\end{aligned}
$$

And in the same way:

$$Term_3 = 2^{2n+1} - 2^{n-1} * \sum_{j=0}^{n-2} (c_{n-1}\&a_j|\overline{c_{n-1}}\&\widehat{b_j}) * 2^j$$
$$= 2^{n-1} * \left(\sum_{j=n-1}^{n+1} 2^j + 1 + \sum_{j=0}^{n-2} \overline{(c_{n-1}\&a_j|\overline{c_{n-1}}\&\widehat{b_j})} * 2^i \right) \quad (2.26)$$

Consequently the total multiplication $A * C + \widehat{B} * \overline{C}$ can be computed as:

$$A * C + \widehat{B} * \overline{C} = (c_{n-1}\&a_{n-1}|\overline{c_{n-1}}\&\widehat{b_{n-1}}) * 2^{2n-2}$$
$$+ 2^{n-1} * \left(\sum_{i=n-1}^{n+1} 2^i + 1 + \sum_{i=0}^{n-2} \overline{(c_i\&a_{n-1}|\overline{c_i}\&\widehat{b_{n-1}})} * 2^i \right)$$
$$+ 2^{n-1} * \left(\sum_{j=n-1}^{n+1} 2^j + 1 + \sum_{j=0}^{n-2} \overline{(c_{n-1}\&a_j|\overline{c_{n-1}}\&\widehat{b_j})} * 2^i \right)$$
$$+ \sum_{j=0}^{n-2} \sum_{i=0}^{n-2} (c_i\&a_j|\overline{c_i}\&\widehat{b_j}) * 2^{i+j} \quad (2.27)$$

If we group all constants from the above equation:

$$\begin{aligned} A * C + \widehat{B} * \overline{C} &= (c_{n-1}\&a_{n-1}|\overline{c_{n-1}}\&\widehat{b_{n-1}}) * 2^{2n-2} \\ &+ 2^{n-1} * \sum_{i=0}^{n-2} \overline{(c_i\&a_{n-1}|\overline{c_i}\&\widehat{b_{n-1}})} * 2^i \\ &+ 2^{n-1} * \sum_{j=0}^{n-2} \overline{(c_{n-1}\&a_j|\overline{c_{n-1}}\&\widehat{b_j})} * 2^j \\ &+ \sum_{j=0}^{n-2} \sum_{i=0}^{n-2} (c_i\&a_j|\overline{c_i}\&\widehat{b_j}) * 2^{i+j} \\ &+ 2^n * \left(1 + \sum_{i=n-1}^{n+1} 2^i \right) \end{aligned} \quad (2.28)$$

If we rewrite this last term using the fact that all computations are done modulo

2^{2n+1}, we get:

$$\begin{aligned} 2^n * \left(1 + \sum_{i=n-1}^{n+1} 2^i\right) &= 2^n + 2^n * \sum_{i=n-1}^{n+1} 2^i \\ &= 2^n + 2^{2n-1} + 2^{2n} + 2^{2n+1} \\ &= 2^n + 2^{2n-1} + 2^{2n} \end{aligned} \tag{2.29}$$

In order to compute $(A \pm B) * C \pm D$ we have to add $\widehat{B}$, $\widehat{C}$, $\widehat{D}$, $\overline{op_1}$ and op_2 to 2.28, as derived in 2.20.

The three constants, (2^{2n}, 2^{2n-1} and 2^n), and the negative sign bits of $\widehat{B}$, $\widehat{C}$ and $\widehat{D}$ (on position 2^{n-1}) can easily be combined to a single bit-string according to Table 2.3 and 2.4 .

Hot_ones $-\sum(\widehat{b_{n-1}}, \widehat{c_{n-1}}, \widehat{d_{n-1}})$	=	result
$2^{2n} + 2^{2n-1} + 2^n - 0 * 2^{n-1}$	=	$2^{2n} + 2^{2n-1} + 2 * 2^{n-1}$
$2^{2n} + 2^{2n-1} + 2^n - 1 * 2^{n-1}$	=	$2^{2n} + 2^{2n-1} + 1 * 2^{n-1}$
$2^{2n} + 2^{2n-1} + 2^n - 2 * 2^{n-1}$	=	$2^{2n} + 2^{2n-1} + 0 * 2^{n-1}$
$2^{2n} + 2^{2n-1} + 2^n - 3 * 2^{n-1}$	=	$2^{2n} + 2^{2n-1} - 1 * 2^{n-1}$

Table 2.3: Summing up the hot ones

Result	Bit_{2n}	Bit_{2n-1}	Bit_{2n-2}.. ..Bit_{n+1}	Bit_n	Bit_{n-1}
$2^{2n} + 2^{2n-1} + 2 * 2^{n-1}$	1	1	0	1	0
$2^{2n} + 2^{2n-1} + 1 * 2^{n-1}$	1	1	0	0	1
$2^{2n} + 2^{2n-1} + 0 * 2^{n-1}$	1	1	0	0	0
$2^{2n} + 2^{2n-1} - 1 * 2^{n-1}$	1	0	1	1	1

Table 2.4: The bit-string resulting from the hot-ones and the sign-bits.

From Table 2.4 we can derive the formulas for these bits.

$$\begin{aligned} Bit_{n-1} &= \widehat{b_{n-1}} \oplus \widehat{c_{n-1}} \oplus \widehat{d_{n-1}} \\ Bit_n &= \widehat{b_{n-1}} \& \widehat{c_{n-1}} \& \widehat{d_{n-1}} | \overline{\widehat{b_{n-1}}} \& \overline{\widehat{c_{n-1}}} \& \overline{\widehat{d_{n-1}}} \\ Bit_{n+1}..Bit_{2n-2} &= \widehat{b_{n-1}} \& \widehat{c_{n-1}} \& \widehat{d_{n-1}} \\ Bit_{2n-1} &= \overline{\widehat{b_{n-1}} \& \widehat{c_{n-1}} \& \widehat{d_{n-1}}} \\ Bit_{2n} &= 1 \end{aligned} \tag{2.30}$$

The block realizing these formula's (2.30) is as complex as a full-adder.

2.3.1 Putting the pieces together

The complete formula is as follows:

$$
\begin{aligned}
&(A\ oper_1\ B) * C\ oper_2\ D \\
&= A * C + \widehat{B} * \overline{C} + \widehat{B} + \widehat{C} + \widehat{D} + \overline{op_1} + op_2 \\
&= (c_{n-1}\&a_{n-1}|\overline{c_{n-1}}\&\widehat{b_{n-1}}) * 2^{2n-2} \\
&+ 2^{n-1} * \sum_{i=0}^{n-2} \overline{(c_i\&a_{n-1}|\overline{c_i}\&\widehat{b_{n-1}})} * 2^i \\
&+ 2^{n-1} * \sum_{j=0}^{n-2} \overline{(c_{n-1}\&a_j|\overline{c_{n-1}}\&\widehat{b_j})} * 2^j \\
&+ \sum_{j=0}^{n-2}\sum_{i=0}^{n-2}(c_i\&a_j|\overline{c_i}\&\widehat{b_j}) * 2^{i+j} \\
&+ \sum_{i=0}^{n-2}(\widehat{b_i} * 2^i) + \sum_{i=0}^{n-2}(\widehat{c_i} * 2^i) + \sum_{i=0}^{n-2}(\widehat{d_i} * 2^i) \\
&+ \overline{op_1} + op_2 + \sum_{i=n-1}^{2n} Bit_i * 2^i
\end{aligned}
\tag{2.31}
$$

Where the term $\sum Bit_i * 2^i$ is defined in 2.30. If we re-arrange the terms we get:

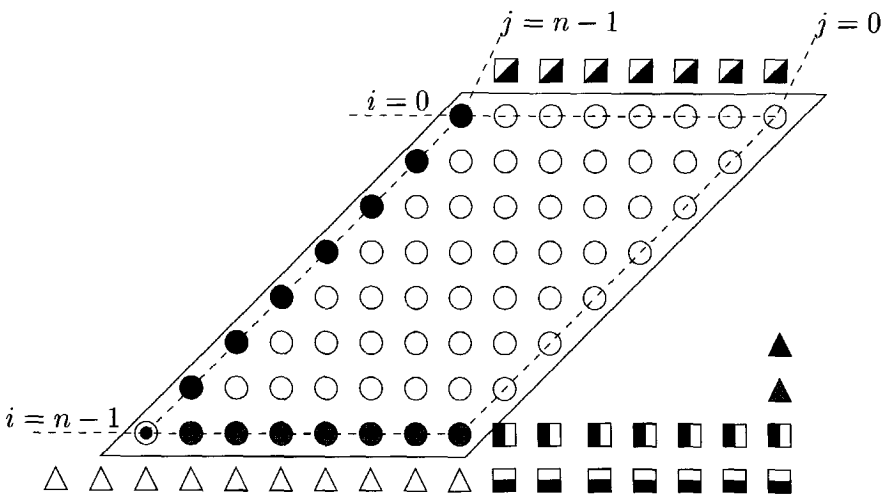
$$
\begin{aligned}
&(A\ oper_1\ B) * C\ oper_2\ D \\
&= A * C + \widehat{B} * \overline{C} + \widehat{B} + \widehat{C} + \widehat{D} + \overline{op_1} + op_2 \\
&= \sum_{j=0}^{n-2}\sum_{i=0}^{n-2}(c_i\&a_j|\overline{c_i}\&\widehat{b_j}) * 2^{i+j}
\end{aligned}
$$

$$
\begin{aligned}
&+ 2^{2n-2} * (c_{n-1}\&a_{n-1}|\widehat{\overline{c_{n-1}}\&b_{n-1}}) \\
&+ 2^{n-1} * \sum_{i=0}^{n-2} \overline{(c_i\&a_{n-1}|\widehat{\overline{c_i}\&b_{n-1}})} * 2^i \\
&+ 2^{n-1} * \sum_{j=0}^{n-2} \overline{(c_{n-1}\&a_j|\widehat{\overline{c_{n-1}}\&b_j})} * 2^j \\
&+ 2^{n-1} * \sum_{i=n-1}^{2n} Bit_i * 2^{i-n+1} \\
&+ \sum_{i=0}^{n-2}(\widehat{b_i} * 2^i) + \sum_{i=0}^{n-2}(\widehat{c_i} * 2^i) + \sum_{i=0}^{n-2}(\widehat{d_i} * 2^i) \\
&+ \overline{op_1} + op_2
\end{aligned}
\tag{2.32}
$$

Figure 2.3 shows a graphical representation of the Inversion Selection Technique. The dot in the lower right corner represents the first term in the above Equation (2.31). The second and third term, the inverted partial products, are indicated with black and gray dots, while the white dots represent the fourth term. The white triangles form the correction term Bit_x. The black and gray triangles are correction terms, resulting from negation of B ($\overline{op_1}$) and D (op_2). The squares represent the positive elements of $\widehat{D}$ and the two correction terms $\widehat{B}$ and $\widehat{C}$.

As can be seen in figure 2.3, all the components of the matrix form a compact structure, which can be reduced using counters. See for example [76]. The Figure shows that there are are only two extra rows required for the Add-Multiply-Add when compared to the normal multiply depicted in Figure 2.2. As setting up the partial products is not more complicated than setting up the partial products of a normal multiply, we conclude that this design will require the same execution time as a normal multiply.

Inversion selection technique remarks: In this section, we have introduced the Inversion Selection Technique. Such a technique uses the fact that the partial products generated by $c_i * A$ and those generated by $\overline{c_i} * B$ can be added by simply OR-ing them together. This technique results in $n + 2$ partial product lines, that can be summed up in an arbitrary counter structure. In the next section, the Half Adder Encoding Technique will be introduced.



- ◉ $Term_1$, which is inverted twice
- ● $Term_2$, which is inverted
- ● $Term_3$, which also is inverted
- ○ $Term_4$, the normal multiplication bits
- △ The correction element $Bit_{2n}..Bit_{n-1}$
- ▲ The first hot one depending on the operation, $\overline{op_1}$
- ▲ The second hot one depending on the operation, op_2
- ◪ The term $\widehat{D} = D \oplus op_2$ (except for the sign-bit)
- ◧ The term $\widehat{B} = B \oplus \overline{op_1}$ (except for the sign-bit)
- ◒ The term $\widehat{C} = \overline{C}$ & $\overline{op_1}$ (except for the sign-bit)

Figure 2.3: Graphical representation of the Add-Multiply-Add using the Inversion-Selection Technique.

2.4 Half-adder encoding

Half-Adder Encoding is derived from Booth-encoding [40]. The main difference is that in the Half-Adder Encoding the "set" of bits which are coded together come from the two operands A and B. This results in n partial products, n being the number of bits of operand A and B, and n hot-ones, resulting from the possible negation of these partial products. For this technique, we add or subtract A and B on a bitwise basis. For this operation, we use a half-

adder, hence the name Half-Adder Encoding. The key-advantage of the use of a half-adder as opposed to a full-adder is the absence of a carry-signal. Earlier we derived four expressions from the add-multiply-add operation, depending on the two controlling inputs (op_1 and op_2) which select the operation. That is the expression:

$$(A \pm B) * C \pm D \quad = \quad (A \; oper_1 \; B) * C \; oper_2 \; D$$

can assume one of the following: $(A + B) * C + D$; $(A + B) * C - D$; $(A - B) * C + D$ or $(A - B) * C - D$ representing one of the choices of $oper_1$ and $oper_2$. We treat the addition/subtraction of D later, for now we only treat the add-multiply part, which has two instances:

$$(A + B) * C$$
$$(A - B) * C$$

From Equation 2.5 through 2.11 we know that the product Z of X and Y, $(Z = X * Y)$ in two's complement is the sum of four terms. We substitute X by C and Y by $(A \; oper_1 \; B)$. This yields:

$$X = C = -c_{n-1} * 2^{n-1} + \sum_{j=0}^{n-2} c_j * 2^i \tag{2.33}$$

$$\begin{aligned} Y &= (A \; oper_1 \; B) = \\ &= -(a_{n-1} \; oper_1 \; b_{n-1}) * 2^{n-1} + \sum_{i=0}^{n-2} (a_i \; oper_1 \; b_i) * 2^i \end{aligned} \tag{2.34}$$

Since the result of $(a_i \; oper_1 \; b_i)$ can range from -1 to $+2$, we have to ensure that multiplying it with C doesn't pose any problems. If C is multiplied by 2, the result will be an $(n + 1)$-bit number. In order to keep all partial products of the same length, C has to be sign-extended by one bit. After this sign-extension, we can safely left-shift it one bit, and ignore the bit that will be shifted out. This yields $C = -c_{n-1} * 2^n + \sum_{j=0}^{n-1} c_j * 2^j$.

The four terms for the computation of $(A \; oper_1 \; B) * C$ are:

$$Term_1 = (-c_{n-1} * 2^n) * (-(a_{n-1}\ oper_1\ b_{n-1}) * 2^{n-1}) \quad (2.35)$$

$$Term_2 = (-c_{n-1} * 2^n) * \left(\sum_{i=0}^{n-2}(a_i\ oper_1\ b_i) * 2^i\right) \quad (2.36)$$

$$Term_3 = (-(a_{n-1}\ oper_1\ b_{n-1}) * 2^{n-1}) * \left(\sum_{j=0}^{n-1} c_j * 2^j\right) \quad (2.37)$$

$$Term_4 = \left(\sum_{j=0}^{n-1} c_j * 2^j\right) * \left(\sum_{i=0}^{n-2}(a_i\ oper_1\ b_i) * 2^i\right) \quad (2.38)$$

with $(a_i\ oper_1\ b_i) \in \{-1, 0, 1, 2\},\ \forall\, i \in \{0, 1, 2, ...n-1\}$.

Because $(a_i\ oper_1\ b_i)$ can take only one of these four values, $((a_i\ oper_1\ b_i) * C)$ can always be written as an element of $n+1$ bits (and possibly a hot one). Moreover, it is trivial to derive this element from C. Table 2.5 defines how this can be done and Figure 2.4 shows an example of how partial products can be computed.

a	op	b	result	how to make this
0	+	0	0	set all to 0
0	+	1	1	just pass through
0	-	0	0	set all to 0
0	-	1	-1	invert all bits and add a hot-one
1	+	0	1	just pass through
1	+	1	2	shift one to the left
1	-	0	1	just pass through
1	-	1	0	set all to 0

Table 2.5: The result of the bit-wise addition(subtraction) of A and B.

Of these four "operations" performed on C, the multiplication with -1 is the most complex one. To accomplish this, all bits are inverted, and a hot one is added. This hot one comprises an extra "partial product", which is visualized by upward triangles in the graphical representation of the Half-Add technique in Figure 2.6 on page 36.

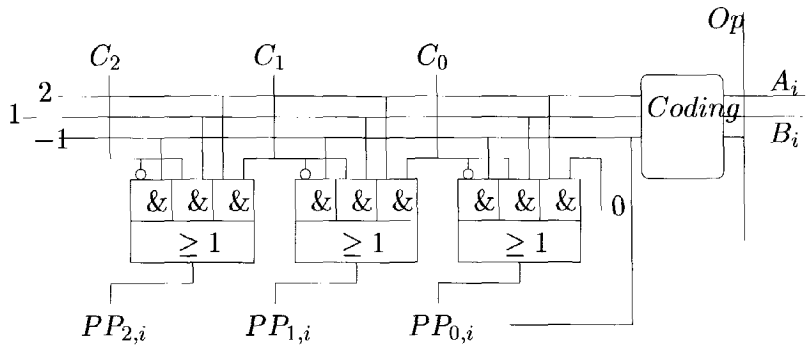


Figure 2.4: Part of a section producing the least significant bits of a partial product using the Half-Adder Encoding.

2.4.1 Adding the partial products together

For adding the partial products together, the same strategy is used as for the Inversion Selection Technique. To account for the negative weight of the sign-bits we invert them all and add a hot one on the position of the least significant inverted bit. Before we invert $Term_2$ and add the hot-one, $\widehat{D} = D \oplus op_2$ is added to the list of partial products. Its sign-bit is concatenated with the sign-bits on the upper diagonal ($Term_2$), yielding a negative partial product ranging from position $n-1$ to $2n-2$. The value of this element is:

$$-\sum_{i=0}^{n-2} c_{n-1}(a_i \; oper_1 \; b_i)2^{i+n} - \widehat{d_{n-1}}2^{n-1} \tag{2.39}$$

In order to account for the negative value, the element is sign-extended, inverted, and a hot one is added on position $n-1$.

So instead of adding the negative element

$$-\sum_{i=0}^{n-2} c_{n-1}(a_i \; oper_1 \; b_i)2^{i+n} - \widehat{d_{n-1}}2^{n-1} \tag{2.40}$$

we add

$$2^{2n} + 2^{2n-1} + \sum_{i=0}^{n-2} \overline{c_{n-1}(a_i \; oper_1 \; b_i)}2^{i+n} + \overline{\widehat{d_{n-1}}}2^{n-1} + 2^{n-1} \tag{2.41}$$

The two sign-extension-bits are later combined with the two sign-extension-bits of the other negative element and the hot-one on position $n-1$ is later combined with the hot-one of the last partial product.

The other negative element is the last row of the multiplication matrix ($Term_3$). Its value is

$$-\sum_{j=0}^{n-1} c_j(a_{n-1} oper_1 b_{n-1})2^{j+n-1} \tag{2.42}$$

This element is a bit more complicated, because it can have a positive value, namely when $(a_{n-1} oper_1 b_{n-1}) = -1$. In this case, these bits do not get inverted, and the most significant bit of the "extra element" (the upward triangles in Figure 2.6) is equal to zero. We combine $Term_3$ with $Term_1$. There are four cases, because $(a_{n-1} oper_1 b_{n-1})$ can have four different values.

$(a_{n-1} oper_1 b_{n-1})$	Value of $Term_1$	Value of $Term_3$	Ranges of the negative value
-1	negative	positive	$2n-1..2n-1$
0	positive	negative	$n-1..2n-2$
1	positive	negative	$n-1..2n-2$
2	positive	negative	$n-1..2n-2$

Table 2.6: The sign of $Term_1$ and $Term_3$ in each of the four possible cases.

$(a_{n-1} oper_1 b_{n-1})$	Operation on $Term_1$	Operation on $Term_3$	Position of hot ones	position of Sign-extension-bits
-1	Inversion	No inversion	$2n-1$	$2n$
0	No inversion	inversion	$n-1$	$2n-1$ and $2n$
1	No inversion	inversion	$n-1$	$2n-1$ and $2n$
2	No inversion	inversion	$n-1$	$2n-1$ and $2n$

Table 2.7: How to deal with the signs of $Term_1$ and $Term_3$.

As can be seen from Table 2.6 and 2.7, $Term_3$ is inverted where $Term_1$ is not inverted, and vice versa. This that means we can just invert the result from the same building blocks. $Term_1$ should only be inverted if $(a_{n-1}\ oper_1\ b_{n-1}) = -1$. In this case, the standard building block inverts this bit, so no extra inversion is needed here.

We can also observe that there are always two extra ones, on position $2n-1$ and $2n$. These two bits can be sign-extension bits, which is the case if $(a_{n-1} oper_1 b_{n-1}) \geq 0$, or a hot one and a sign-extension bit, if $(a_{n-1} oper_1 b_{n-1}) = -1$. $Term_2$ has two sign-extension bits, also on position $2n$ and $2n-1$. These sign-extension bits can be added together with the hot-ones form $Term_1$ and $Term_3$, also on position $2n$ and $2n-1$. This addition results in: $2^{2n} + 2^{2n-1} + 2^{2n} + 2^{2n-1} = 2^{2n+1} + 2^{2n}$ Of this sum, the 2^{2n+1} term can be discarded, as the result is truncated at $(2n+1)$ bits.

Figure 2.5 gives a 4-bit example of the Half-Adder Encoding in some detail. Note that all squares have the same, one-level functionality. The rounded squares are the half-adders, which generate a 2, a 1 and a -1 line. The real squares select one of the bits coming in on the top (or its inverse) depending on the value of the 2, 1 and -1 lines coming in on the left. Figure 2.6 gives an 8-bit example in the same representation used for the Inversion Selection Technique. As can be seen from the figure, the Half-Adder Encoding also requires only 2 rows more than a normal multiply. As setting up the partial products is hardly more complicated when compared to a normal multiply, this unit should operate as fast as a normal multiply.

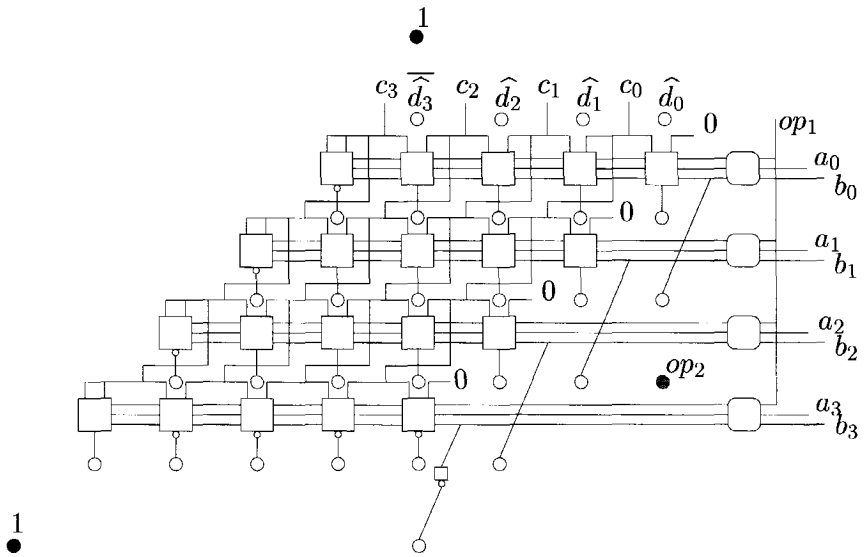
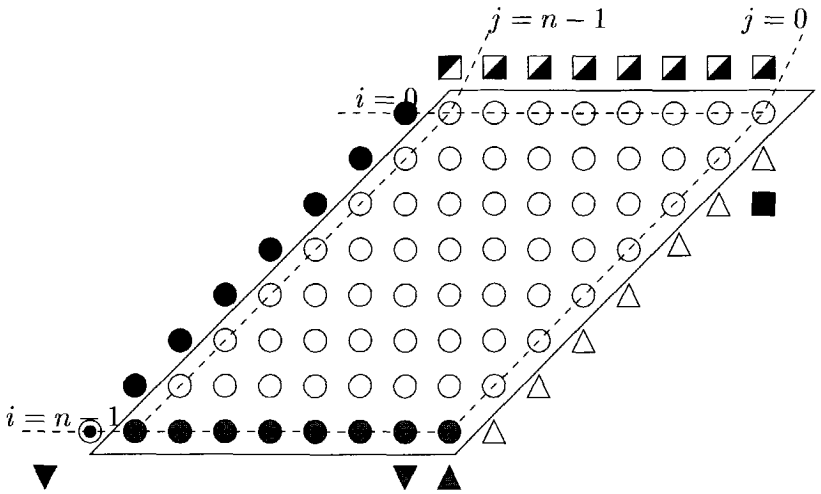


Figure 2.5: A 4 bit example of the Half-Adder Technique. (Note: The black dot on the top is only shown for clarity. It is easily combined with the white dot on the bottom of the same column.)

Half-adder encoding remarks: For the Half-Adder Encoding, we have shown that in two levels of hardware, it is possible to compute n partial products, an element of "hot-ones", an element for $\widehat{D}$, and two hot ones. The first level is a level of half-adders, which drive three selection lines each. These selection lines choose which derivate of C is inserted in the partial-product matrix.



- ◉ $Term_1$, which is inverted twice
- ● $Term_2$, which is inverted
- ● $Term_3$, which also is inverted
- ○ $Term_4$, the normal multiplication bits
- △ The hot ones that are added if $(a_i\ oper_1\ b_i) = -1$
- ▲ The hot one that is only added if $(a_i\ oper_1\ b_i) = -1$
- ▼ The hot one that is only added if $(a_i\ oper_1\ b_i) \neq -1$
- ▼ The result of the addition of the sign-extensions of $Term_2$ and $Term_3$
- ◩ The inverted sign-bit of D; $\widehat{\overline{d_{n-1}}}$
- ◪ The other bits of $\widehat{D}$; $\sum_{i=0}^{n-2} \widehat{d}_i 2^i$
- ■ The hot one that is added if $\widehat{D}$ is to be subtracted, op_2

Figure 2.6: Graphical representation of the Add-Multiply-Add using the Half-Adder Encoding.

2.5 Conclusions

In this chapter we investigated complex (compound) instructions that could provide a performance improvement for embedded systems and multimedia applications if implemented in hardware. Assuming two's complement representation, we propose two techniques for the implementation of such an expression. Both these two techniques provide a way of efficiently computing

compound expressions of the general form $(A \pm B) * C \pm D$, in other words, Add-Multiply-Add operations. We have shown that an unit capable of computing this expression can also be used to compute various simpler expressions. For both these techniques, a logic design of the Add-Multiply-Add unit is derived, where both use a derivate of the Baugh-Wooley scheme [75] to add up the negative and positive bits from the partial products. The Inversion-Selection Technique uses the fact that $\overline{X}$ and $-X$ are "almost equal". To be precise: $-X = \overline{X} + 1$. We exploit this by multiplying A by C and $-B$ by $\overline{C}$, in which case each pair of rows in the multiplication-array can be merged into one instead of being added together. The Half-Adder Encoding is similar to the inversion selection technique in that it also tries to reduce the number of partial products. The half-adder technique accomplishes this by "preprocessing" $(A\ oper_1\ B)$ in a way that no carries arise. The result of this operation is one of $\{-1, 0, 1, 2\}$, and has to be multiplied by C. The resulting partial products are added together with D (or $-D$). Given the way we compute the partial products, we should be able to compute $(A \pm B) * C \pm D$ as fast as a multiplication.

In the next chapter we will consider a unit which computes the Sum of Absolute Differences, which is a matching criterion used in video compression.

Chapter 3

Sum and Mean of Absolute Differences

Video compression is one of the main topics of Multimedia. Within video sequences, both spatial and temporal redundancy can be exploited for compression. The spatial redundancy is exploited by means of the DCT, and the temporal redundancy is exploited by transmitting only the differences with a previous frame and/or the next frame. Temporal compression could be optimal if the images didn't change over time. While this is not the case in practical video sequences, big parts of images generally move only small distances. In order to reach higher compression rates, compression algorithms try to compensate for the motion between two successive images. In order to do so, a motion estimation algorithm is needed that "determines" the motion. There are several algorithms and for a large number of them the Sum of Absolute Differences serves as maximum resemblance optimization criterion.

In block-based motion estimation [77, 78], that is motion estimation performed on a set of pixels, every frame is divided into blocks of equal size and for each block in the *current* frame a search is performed in the *reference* frame(s) to find the block resembling the current block the most. Because a search performed over the whole reference frame for each block in the current frame is computational intensive and movements in video sequences are usually small, the search is limited to a search area. After finding the best match for the current block, the motion vector (i.e. the displacement relative to the current block) is stored together with the differences between the two blocks. In determining which block in the searching area of the reference frame is the best match with the current frame, a best match method is employed. The best

match is usually established with the use of the *mean of absolute differences* (MAD) or the *sum of absolute differences* (SAD). The SAD is the sum created by adding the absolute difference of each corresponding pixel in the current and the reference frame. That is the SAD is defined as:

$$SAD(x,y,r,s) = \sum_{i=0}^{i=15} \sum_{j=0}^{j=15} |A_{(x+i,y+j)} - B_{((x+r)+i,(y+s)+j)}| \qquad (3.1)$$

where (x, y) is the position in the current frame and (r, s) is the motion vector for which the SAD is determined. The MAD is defined as the SAD divided by the number of pixels in the macro-block, in this case 256.

In this chapter, our primary concern is to propose a hardware solution to the SAD and the MAD operations. That is, our primary concern is to propose instructions that have "convenient" hardware implementations, where "convenient" in the context of this discussion mainly means parallel hardware vector related implementations. The design we propose is optimized for a low latency, which is needed in several data-dependent Motion Estimation Algorithms, such as the 2-D logarithmic search. We note here that if the division (shifting) operation is excluded from the MAD then both operations can be viewed as equivalent. In essence, discussing the SAD operation will also cover the MAD with an additional shift (divide) of the final result, thus MAD is no longer considered in the discussion to follow. Given that the SAD operation is usually considered for a block of 16 by 16 pixels (pels) [77] and because the search area could involve a high number of these blocks, performing the SAD operation could be time-consuming if traditional methods are used for its computation[1]. We propose a new instruction that is capable of producing the direct SAD operation. Furthermore we also show that the proposed instruction is scalable, depending on the constraints of the technology considered for the design. This is shown by considering a 16x1 sub-block element and an entire 16x16 element and showing that the implementation will require 3 machine cycles[2] for a 16x1 sub-block and 4 cycles for a 16x16 block. The 16x16 block performance is achieved by using hardware proportional in size to a 16x1 sub-block unit, that is, we achieve a 4 cycle 16x16 block SAD using approximately 16 times the area of the 16x1 SAD.

[1]Traditional here means that performing SAD requires a number of subtractions with proper complementation to produce the absolute value which are followed by an accumulation to perform the final operation.

[2]A cycle here is considered to be comparable to the cycle of a high-speed, 2-cycle, 32x32 bit multiplier [70, 79, 41]. Other implementations including systolic array implementations are also possible.

This chapter is organized as follows. Section 3.1 gives some background information about motion estimation and how it fits in the MPEG standard, followed by a discussion of the SAD operation. Section 3.2 describes the basic operation of our proposed Sum-Absolute-Difference unit, and Section 3.3 gives a sample implementation of the proposed unit. Section 3.4 concludes this chapter with some remarks and future research directions.

3.1 Background

In MPEG [77, 80], video-sequences are compressed by exploiting both spatial and temporal redundancies. Spatial redundancies can be seen as small differences between local pels. In many encoding schemes the spatial redundancies are exploited using DCT [81, 82, 83] or predictive coding [84]. Temporal redundancies can be seen as small differences between two successive video frames. These kind of redundancies can be exploited using predictive coding, but higher compression rates can be reached by using it together with motion compensation [85]. As an example, a diagram of the MPEG encoding process is given below. (Figure 3.1) Because the MPEG standard does not specify the encoding process, this diagram is only one possible implementation for the MPEG encoding process. In the diagram, FDCT denotes the Forward Dis-

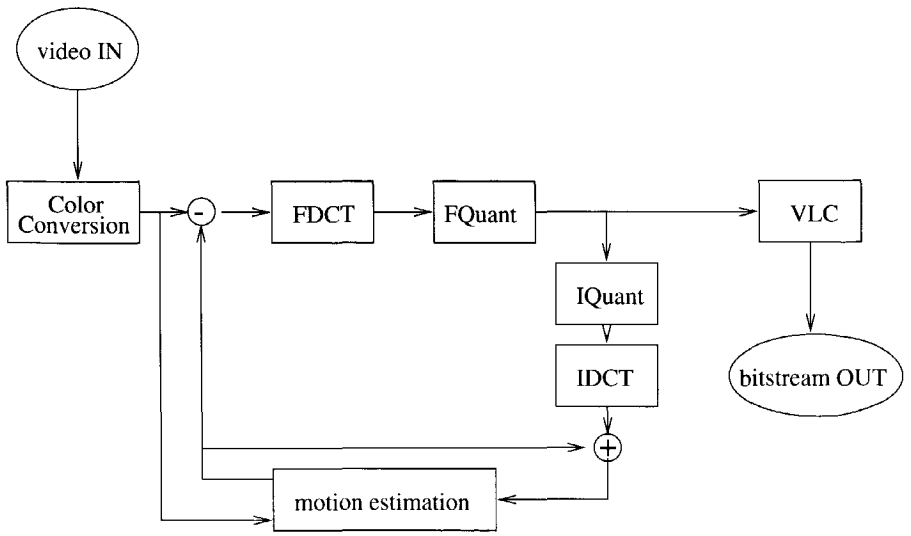


Figure 3.1: Diagram of a MPEG encoder implementation.

crete Cosine Transformation, FQuant denotes the Forward Quantization and VLC denotes Variable Length Coding. The IQuant and IDCT are the inverse operations needed to reproduce the picture as it is available at the decoder. The motion estimation block uses these decoded images as reference instead of original images, because the decoder only has access to decoded images. Adding "differences from original images" to decoded images which are already slightly different from the original images, would introduce unnecessary errors in the decoded images. This introduces an extra amount of computation in the encoder, which is in the order of the amount of computation in the decoder.

In MPEG coding, a video sequence is divided into frames. A small fraction of these frames, the I(ntra-coded) frames, are transmitted without the use of motion estimation. The P(redictive-coded) and B(idirectionally predictive-coded) frames use Motion Estimated prediction. Each frame is subdivided into macro blocks, which are 16 by 16 pels in size. For each macro block a Motion Vector is computed, which points to the block in a reference frame which it resembles most. The motion compensated and predicted macro-block is the result of the subtraction of this most resembling block from the current macro block. The assumption is that this block can be coded using fewer bits while retaining the same quality.

Figure 3.2 and 3.3 show the usefulness of Motion Estimation. Without Motion Estimation, the difference between the two pictures (the moved sun) has to be coded as difference values (left part of Figure 3.3). Using Motion Estimation, we can transmit a vector for each block that contains a part of the sun. As the sun has not changed color, the Motion Compensated difference frame contains only zeros. In a realistic setting, the Motion Compensated frame will still contain some values $\neq$ zero, however, those are small values which can easily be compressed. Each macro block is further subdivided into 4 basic blocks of 8 by 8 pels, on which the Discrete Cosine Transform is performed. The encoding of a video stream is done in several steps. Each of the steps depicted in Figure 3.1 is explained below. For simplicity, many details regarding the MPEG standard are left out.

Color conversion In this step the input color-space is transformed into the YCbCr color-space. Furthermore, the chrominances (color samples) are sub-sampled by a factor of two in both the horizontal and vertical direction. Thus, a macro block from the video signal results in four 8 by 8 luminance blocks, one 8x8 Cb block, and one 8 by 8 Cr block. These 8 by 8 blocks are used by the DCT. The 16 by 16 luminance block is used

by the motion estimation.

Motion Estimation In this step, for each block of 16 by 16 luminance pels in the current frame, a motion vector is computed. This motion vector contains the relative position of the block in the reference frame most closely resembling the current block (either in the past or future). To exploit spatial redundancies between these difference values, the difference values are also put through the DCT process.

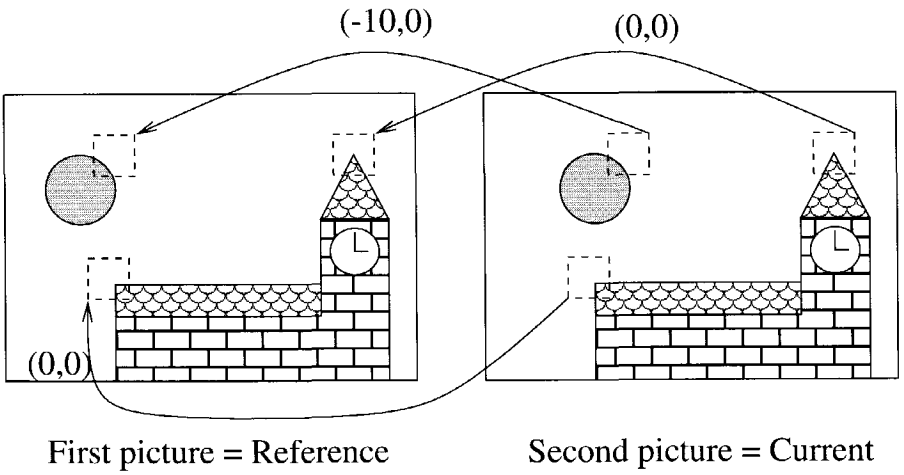


Figure 3.2: Example of Motion Vectors. The sun has moved to the right. The reference picture is scanned to find the corresponding piece of sun.

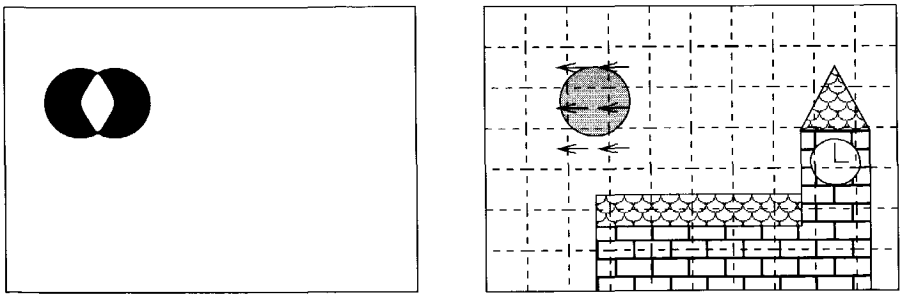


Figure 3.3: The left part shows what would be transmitted without motion vectors. The right part shows using motion vectors. For each block that has no vector, the vector is simply (0,0). The residual difference is zero in this example, as all matches are perfect.

Forward Discrete Cosine Transformation (FDCT) In this step a DCT is performed on each 8 by 8 blocks which can be either blocks from the frame or difference blocks. If this is a block of an I frame (reference frame), the 2D-DCT over this 8x8 block is computed. If the block is a motion compensated predicted block, e.g. the block minus the block it most resembles, the 2D-DCT is computed over this motion compensated predicted block.

Forward Quantization (FQuant) In this step the DCT coefficients computed in the DCT process are quantized. This step is the main contribution to the lossy nature of the MPEG coding standard. However, the information lost in this step is thought to be (almost) not perceivable, due to the use of the DCT. If there are constraints on the bit-rate, the quantization can be adjusted to meet these constraints. This can result in lower quality video if the quantization is too coarse.

Inverse Quantization (IQuant) In this step the quantized DCT coefficients are restored. The same rounding-errors occur here as those that will occur on the receiver side.

Inverse Discrete Cosine Transform (IDCT) In this step the inverse DCT is performed on the dequantized coefficients, which results in the original picture after color conversion. These two last steps are necessary in order to do the motion estimation on the restored picture, as the receiver also only has the received picture available.

Variable Length Coding (VLC) In this step the results of the quantization process are serialized into a bit-stream, using run-length coding and variable length coding (in this case Huffman coding).

The decoding process is depicted in Figure 3.4 which is basically the lower part of the encoding process.

First, the incoming bit-stream is decoded using variable length decoding. Second, the results are fed into an inverse quantization step and an inverse DCT step. If the blocks are not motion-compensated difference blocks, they can be directly fed to the output of the decoder. If motion vectors are decoded, then they are used to fetch data from reference frames, to which the decoded difference values are added. Worth noting is that the MPEG standard is not symmetric, meaning that the computational requirements for the encoder and the decoder are different. For example, the encoder has to put lots of effort

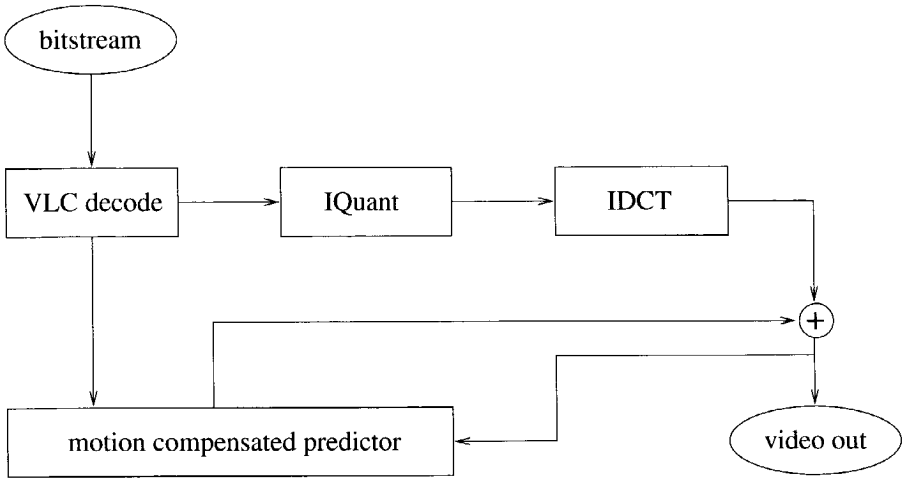


Figure 3.4: Diagram of a MPEG decoder implementation.

in calculating motion vectors, while the decoder just uses the motion vector to fetch the right blocks from memory.

In this chapter we focus on ways to speed up the motion estimation part of MPEG encoding, also denoted as *motion vector search*. There are several algorithms to compute which block in the reference frame most closely resembles the current block. At one end of the spectrum is the exhaustive search [84] which is time-consuming, but produces the best possible result, and at the other end there are several heuristic-based algorithms, which are much faster at the cost of a possibly less optimal result. Examples of these heuristic-based search algorithms are the Three Step Search [86] and the Two Dimensional Logarithmic search [87, 88], which is schematically displayed in Figure 3.5. The 2-D Logarithmic search uses two phases in the motion vector search. In the first phase, a fixed step-size equal to 2 is used. The SAD for the center and four points on a diagonal square around this center are computed. These points are denoted in the figure with a number 1. The point with minimal SAD will be the center for the next step. In this case the point (0,-2) is the center of the next step. The points checked in this step are denoted with a 2 in the figure. If the point with minimal SAD is the center-point, the algorithm proceeds with the second phase. This is the case after step 4, in which (-4,-2) is the point with minimal SAD. In the second phase, the SAD is computed for 8 points at distance 1 and $\sqrt{2}$. The center-point itself is also considered. These points are denoted with 5 in the figure. Of these nine points, the point with minimal SAD is chosen as Motion-Vector, which is (-5,-3) in this example.

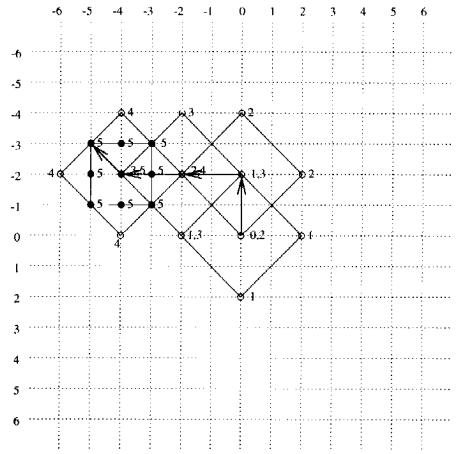


Figure 3.5: Graphical representation of the 2-D logarithmic-search.

Note that a point in this algorithm designates a vector, and for each vector a 16 by 16 block of pixels has to be compared. This means that the SAD for a point is really the SAD of the current macro-block and a block of the reference frame which is shifted over the vector.The assumption made by these algorithms is that the global minimum can be reached by following the steepest descent. If the results are less optimal, larger difference values will result from the subtraction. This in turn results in larger bandwidth requirements or quality degradation if bandwidth constraints apply, because coarser quantization must be applied.

As there are many search-algorithms (see Appendix A for some additional algorithms), which all have different characteristics for implementation in hardware, we will leave the search algorithm to be implemented in software. However, irrespective of the search algorithm to determine the best resemblance, a metric is used which indicates the "closeness" between compared blocks. The two most common metrics found in different search algorithms are the *mean square error* (MSE) and the *mean absolute difference* (MAD). The MSE is performed by the following (assuming blocks of 16x16 pixels):

$$MSE(x, y, r, s) = \frac{1}{256} \sum_{i=0}^{i=15} \sum_{j=0}^{j=15} \left(A_{(x+i,y+j)} - B_{((x+r)+i,(y+s)+j)} \right)^2 \tag{3.2}$$

The MAD is performed by

$$MAD(x,y,r,s) = \frac{1}{256}\sum_{i=0}^{i=15}\sum_{j=0}^{j=15}\left|A_{(x+i,y+j)} - B_{((x+r)+i,(y+s)+j)}\right| \tag{3.3}$$

MAD can also be rewritten as

$$MAD(x,y,r,s) = \frac{SAD(x,y,r,s)}{256} \tag{3.4}$$

where SAD is the summation of the absolute differences, that is:

$$SAD(x,y,r,s) = \sum_{i=0}^{i=15}\sum_{j=0}^{j=15}\left|A_{(x+i,y+j)} - B_{((x+r)+i,(y+s)+j)}\right| \tag{3.5}$$

In these equations (x,y) is the position of the current block and (r,s) denotes the motion vector, i.e. the displacement of the current block (A) relative to the block in the reference frame (B). The x and y in Equations 3.2 through 3.5 are multiples of 16[3] for MPEG and the values of r and s are determined by the algorithm. Given that the MAD metric, due to its computational simplicity, is used more often, we will not consider the MSE in our discussion. In the section to follow, we introduce a novel approach for the computation of the SAD which leads to the computation of the MAD with a trivial extension.

3.2 Computing the Sum of Absolute Differences

In this section we proceed by investigating the SAD operation and propose some possible parallel implementations leading to an instruction proposal for SAD. The general algorithm computing the Sum of Absolute Differences of two blocks is depicted in Equation 3.5. A direct approach in computing the SAD consists of the following steps:

- Compute $(A_i - B_i)$ for all 16x16 pixels in the two blocks A and B.

[3]We note that we assume in the remaining of the presentation 16x16 pixel MPEG macroblocks. This assumption is not restrictive as our proposal supports arbitrary block-sizes.

- Determine which $A_i - B_i$ are less than zero and produce in that case $B_i - A_i$ as the absolute value, else produce $A_i - B_i$.
- Perform the accumulate operation to all 16x16 absolute values.

In order to speed up the computation, we perform a multiplicity of operations in a single operation. In the case of the computation of the SAD we want to eliminate the absolute-difference operations. Generally, it is not possible to eliminate these operations, because of the inability to take an absolute operation out of a summation.

$$\sum |A_i - B_i| \neq |\sum (A_i - B_i)| \tag{3.6}$$

Our solution to this problem is as follows. By determining the smallest of both operands and subtracting it from a constant, which is $\geq$ the maximum value of a pixel, it becomes possible to eliminate the absolute operations. This subtraction is a trivial operation, if the constant is chosen correctly.

To achieve our goal, we first briefly describe an unit capable of computing the SAD of 16x1 pels in parallel, where each pel(pixel) is represented in 8 bits (in unsigned binary notation).

Determine the smallest of two operands: This is done by inverting one of the operands, and computing the carry-out which would arise from the addition of both operands.

Invert the smallest operand and pass both operands to an adder tree: The smallest operand is inverted, which means that its value changes to $2^8 - 1 - X = 255 - X$. Both the inverted smallest and the largest values are passed to the adder-tree, which corrects for this constant ($2^8 - 1 = 255$).

The above two steps can be carried out in parallel for 16 pels. This results in 32 8-bit values, on which the following steps are applied.

Addition of a correction term: The correction term is added to account for the $2^n - 1$'s introduced by the inverting of the smallest value.

Reduce the 33 rows to 2: The resulting 32 rows are passed to the adder tree together with the correction-term. These 33 rows are reduced to 2 rows by using a counter scheme, see for example [89, 90, 91].

Reduce the 2 **rows to** 1 **(accumulation):** In this final step, a full summation of the two remaining rows is performed. The carry_out of this addition is the total sum of all constants, which has to be discarded.

A more thorough explanation of each step follows below, which is using n for the number of bits to represent the luminance pels and m instead of 16 for the number of pels on which the unit operates. Note that if m is a power of 2, we have a special case which may simplify some computations.

Step 1, (Determining the smallest): Both operands A and B are positive numbers in binary representation in n bits and range from 0 to $(2^n - 1)$. The result of $|A - B|$ is also in unsigned binary representation, and also has the same range. To avoid the absolute operation, we can substitute $|A - B|$ with $A - B$ or $B - A$, depending whether A or B is the smallest. To determine which one is the smallest, we have to check whether the following inequality is true or false: $B > A$, that is $B - A > 0$. Generally it is not possible to subtract two positive numbers without the possibility of producing a negative result which can not be represented as an unsigned number. If we subtract A from its maximum value, $\sum_{i=0}^{n-1} 2^i = 2^n - 1$, the result is always positive or zero, and therefore representable as an unsigned number. The result of the subtraction $((2^n - 1) - A)$ is $\overline{A}$, the binary bit by bit inversion of A. This can be concluded from the following equation:

$$\begin{aligned} A + \overline{A} &= \sum_{i=0}^{n-1} a_i 2^i + \sum_{i=0}^{n-1} \overline{a_i} 2^i && (3.7) \\ &= \sum_{i=0}^{n-1} (a_i + \overline{a_i}) 2^i && (3.8) \\ &= \sum_{i=0}^{n-1} 2^i = 2^n - 1 && (3.9) \\ \text{Therefore} : \overline{A} &= 2^n - 1 - A && (3.10) \end{aligned}$$

We still have to check the following inequality:$B > A$. We rewrite this inequality as follows:

$$\begin{aligned} B &> A \\ 2^n - 1 - A + B &> 2^n - 1 \\ \overline{A} + B &\geq 2^n \end{aligned}$$

The last step is possible because we are dealing with natural, non-fractional, numbers.

The maximum value of $\overline{A} + B$ is $(2^n - 1) + (2^n - 1) = 2^{n+1} - 2$. This is representable as a $n + 1$ bit number. The most-significant bit, with weight 2^n, is computed as the carry-out of the n bit addition. Thus checking whether $\overline{A} + B \geq 2^n$, and hence whether $B > A$ means checking whether the addition of the bit inverted A and the operand B produces a carry out.

Step 2, (Inverting the smallest value): To compute $|A - B|$ in a single step (which will improve the computation of the SAD) we can compute separately $A - B$ and $B - A$ and determine which of the two has a negative result. Consequently, we could choose (multiplex) between the two results choosing the "positive" value. There are two drawbacks with this approach. One relates to the hardware, the other to delay. To perform the entire operation in parallel we must consider two adders per single operation and pay, in addition to the adder delay, the multiplexer delay. The two problems can be alleviated by doing the following. Instead of adding the two input values, we convert the smallest input value to $2^n - 1 - X = \overline{X}$, (that is the one's complement of X). In the remaining discussion, these two values form two rows and they are denoted as a couple.

There are two cases arising from the previous step:

No carry was generated: This implies $B \not> A$. In this case we should invert B to $\overline{B}$. As stated previously, the value of $\overline{B}$ is equal to the positive number $2^n - 1 - B$. This number is again in unsigned binary representation. The value A should be propagated unmodified. Their sum equals $2^n - 1 - B + A = 2^n - 1 + |A - B|$.

A carry was generated: This implies $B > A$. In this case we should invert A to $\overline{A}$ and propagate B unmodified. Their sum equals $2^n - 1 - A + B = 2^n - 1 + |A - B|$.

Thus in both cases, the $n+1$ bit sum of the two values is equal to $2^n - 1 + |A - B|$, which is the desired value $|A - B|$ plus a constant of $2^n - 1$. In the next step, this constant will be eliminated. It should be noted here, as also indicated in step1, that the inversion of the operands A or B is not known a priori. To determine which operand (A or B) to invert, it is enough to compute the carry out of the operation $\overline{A} + B$ (see step1 for further elaboration).

Step 2 takes one level of exors and can be performed in the same first cycle. This step produces two n-bit numbers, which are added in step 4. Figure 3.6 gives a graphical representation of the first two steps. We note here that steps 1 and 2 substitute two adders and multiplexing logic of the output of the adders with carry-out-detection logic and multiplexing of operands improving both the hardware and the delay requirements.

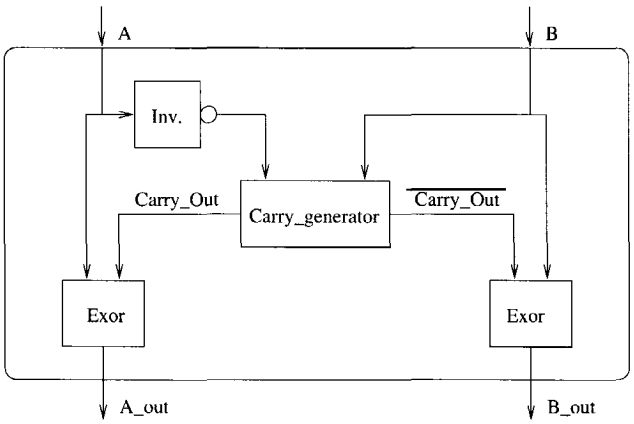


Figure 3.6: Graphical representation of the first two steps in computing the Sum of Absolute Differences (SAD).

Step 3, (Adding a correction term): In order to parallelize the computation of the SAD, the two previous steps are performed on m couples (that is A,B operands) in parallel. The $2m$ rows, the result of step 1 and 2, are positioned into a matrix which is then reduced (summed) using some well known counter-scheme and discussed in steps 4 and 5.

Each of the m couples has a sum equal to $2^n - 1 + |A_i - B_i|$, assuming A_i and B_i of length n. Thus the sum of all couples has the value

$$Sum = m * (2^n - 1) + \sum_{i=0}^{m-1} |A_i - B_i| \tag{3.11}$$

which can be rewritten as:

$$Sum = m * 2^n - m + \sum_{i=0}^{m-1} |A_i - B_i| \tag{3.12}$$

Because $|A_i - B_i|$ is always less than 2^n, the sum $\sum_{i=0}^{m-1} |A_i - B_i|$ will always be less than $m * 2^n$. The desired sum is therefore always representable in

$n + \lceil log_2(m) \rceil$ bits.

For this discussion, we define $q = \lceil log_2(m) \rceil$.

In order to eliminate the constant $m*2^n - m$ from the sum without subtraction, we have to be able to split the result in two parts. The first part consists of the lower $(q + n)$ bits, and the higher part consists of the most-significant bit, with value 2^{q+n}. We now have to make sure that the sum of the constants equals this most-significant bit, by adding an extra constant. The value of this $Extra\_Constant$ is computed at design-time with the following formula:

$$Extra\_Constant = 2^{q+n} - m * 2^n + m \tag{3.13}$$

Note that in the case that m is a power of 2, it simply takes the value m.

After adding this extra_constant, the total sum will be:

$$\begin{aligned} Total\_Sum &= Extra\_Constant + Sum \\ Total\_Sum &= 2^{q+n} - m * 2^n + m + \\ &\quad + m * 2^n - m + \sum_{i=0}^{m-1} |A_i - B_i| \\ Total\_Sum &= 2^{q+n} + \sum_{i=0}^{m-1} |A_i - B_i| \end{aligned} \tag{3.14}$$

Given that 2^{q+n} is not required to represent the result, it can be discarded producing the needed Final_Sum as:

$$\begin{aligned} Final\_Sum &= Total\_Sum - 2^{q+n} \\ Final\_Sum &= 2^{q+n} + \sum_{i=0}^{m-1} |A_i - B_i| - 2^{q+n} \\ Final\_Sum &= \sum_{i=0}^{m-1} |A_i - B_i| \end{aligned} \tag{3.15}$$

Step 4, (Matrix reduction): In step 4, we reduce the $2m + 1$ rows matrix resulting from the $2m$ A and B rows of each n bits and the single constant row to 2 rows. This matrix reduction can be done in several ways. We could use for example Lim counters [92], 6-2 counters [91, 93], or a tree of Carry-Save-Adders (CSA) [89, 90]. The Carry-Save-Adder-Tree approach is used in the

example in Section 3.3 and shows that for $m = 16$ and $n = 8$, 260 CSA's in 8 levels can reduce the 33 rows to 2.

Step 5, (Final reduction): The last step is the final reduction of the matrix. This is done using a fast carry-lookahead scheme.

Figure 3.7 shows a graphical representation of a 16x1 unit, that is a unit operating on 16 couples of elements producing a single output value. The top half shows 16 times steps 1 and 2 in parallel, and steps 4 and 5 are depicted in the bottom half. Step 3 is represented by the addition term at the left (16).

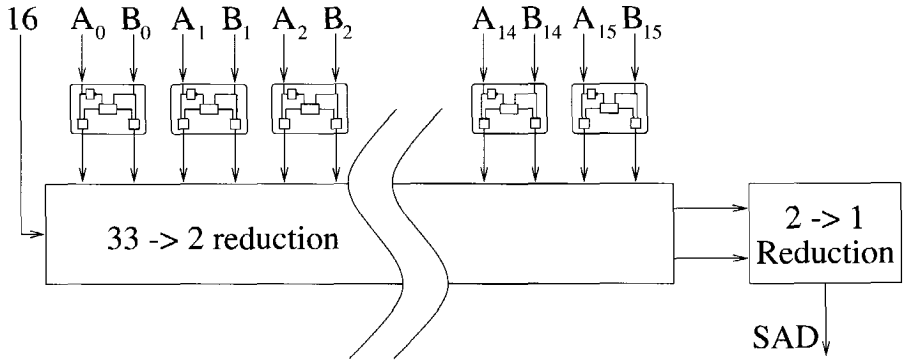


Figure 3.7: Graphical representation of the SAD computation of a 16x1 block by using 16 of the blocks from Figure 3.6 in parallel and a multiplier-like tree reduction.

The concept can be expanded to an array capable of computing the SAD of 16x16 pel blocks. In this case, the 2 rows going into the 2-to-1 reduction should go into another 32-to-2 reduction unit, together with the 30 rows of the 15 other units. The result of this 32-to-2 reduction is then reduced by a 2-to-1 final adder. This saves both the execution time and the area of 15 2-to-1 reduction units. For a block diagram of this extension see Figure 3.8

3.3 A Sample Hardware Implementation

As an example, we describe the implementation of a unit which computes the SAD of two 16x1 blocks, which can be either a row or a column of a 16x16 macro-block. We assume 8 bit values which is common in MPEG.

Step 1, (Determining the smallest): We need 16 parallel blocks to perform

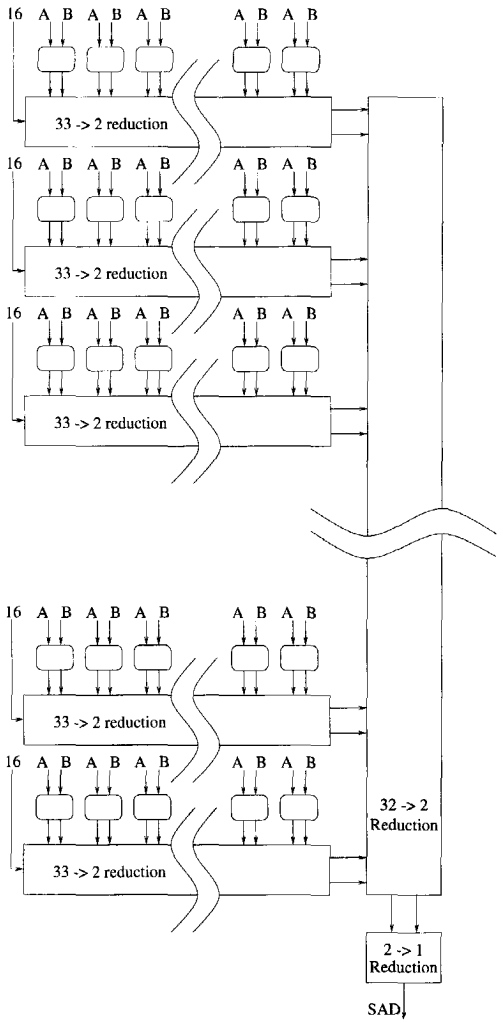


Figure 3.8: A 16x16 pel SAD computation unit. Note that each block is assumed to take one cycle, thereby making the total number of cycles for this unit equal to 4.

step 1. In each of these blocks, we first need to compute the C_{out} of the n bit addition $\overline{A} + B$. This is formed by:

$$C_{out} = G_0^7 + P_0^7 * C_{in} \tag{3.16}$$

In Equation 3.16, P stands for Propagate and G for Generate. Because C_{in} is always zero in this case, we can ignore P_0^7. The remaining term to compute

is G_0^7. This can be computed in 4 stages using 2x2 And-Or-Invert as the most complex gate as follows:

Stage 1

$$G_0^0 = \overline{a_0} * b_0 \quad (3.17)$$

....

$$G_7^7 = \overline{a_7} * b_7 \quad (3.18)$$

$$P_1^1 = \overline{a_1} + b_1 \quad (3.19)$$

....

$$P_7^7 = \overline{a_7} + b_7 \quad (3.20)$$

Stage 2

$$G_6^7 = G_7^7 + P_7^7 * G_6^6 \quad (3.21)$$

$$G_4^5 = G_5^5 + P_5^5 * G_4^4 \quad (3.22)$$

$$G_2^3 = G_3^3 + P_3^3 * G_2^2 \quad (3.23)$$

$$G_0^1 = G_1^1 + P_1^1 * G_0^0 \quad (3.24)$$

$$P_6^7 = P_7^7 * P_6^6 \quad (3.25)$$

$$P_4^5 = P_5^5 * P_4^4 \quad (3.26)$$

$$P_2^3 = P_3^3 * P_2^2 \quad (3.27)$$

Stage 3

$$G_4^7 = G_6^7 + P_6^7 * G_4^5 \quad (3.28)$$

$$G_0^3 = G_2^3 + P_2^3 * G_0^1 \quad (3.29)$$

$$P_4^7 = P_6^7 * P_4^5 \quad (3.30)$$

Stage 4

$$G_0^7 = G_4^7 + P_4^7 * G_0^3 \quad (3.31)$$

It might be convenient to compute $\overline{G_0^7}$ in the same stage.

$$\overline{G_0^7} = \overline{G_4^7 + P_4^7 * G_0^3} \quad (3.32)$$

$$\overline{G_0^7} = \overline{G_4^7} * \overline{P_4^7 * G_0^3} \quad (3.33)$$

Depending on the chosen technology, it might be possible to skip the first stage, and to merge it with the second stage.

Step 2, (Inverting the smallest): In the second step, we have to invert the smallest of A and B. Again, this needs to be done for 16 input-pairs. In terms of hardware, this operation is merged with the operation in step 1 and performed as follows.

Stage 5

$$\begin{aligned} \forall \quad i \quad & \in \{0..7\}, \\ a_{i,out} \quad &= \quad G_0^7 * \overline{a_i} + \overline{G_0^7} * a_i \qquad (3.34) \\ b_{i,out} \quad &= \quad G_0^7 * b_i + \overline{G_0^7} * \overline{b_i} \qquad (3.35) \end{aligned}$$

These 5 stages of step 1 and 2 can be executed in the first cycle.

Step 3, (Placing the correction term): In step 3 we place a correction-term to the terms to be added. Therefore, the number of rows to add up in step 4 becomes $16 * 2 + 1 = 33$. The correction term has a predetermined value, computed at design time with Equation 3.13. This step does not take any execution time.

Step 4, (Reducing the matrix): In step 4, we perform the matrix reduction. For a 33-to-2 reduction, a total of 260 Carry Save Adders in 8 levels is sufficient.

Step 5, (Final addition): Step 5 is the final 2-to-1 addition. This is done using a carry-lookahead scheme.

The last two steps each take one cycle, making the total number of cycles needed equal to 3.

3.4 Conclusions

In this chapter we addressed some issues associated with the acceleration of multimedia motion estimation algorithms. We considered for implementation the Mean Absolute Difference (MAD) and the Sum Absolute Difference (SAD), two frequently used operations in Motion estimation algorithms. We proposed instructions for such operations and considered hardware implementation for the SAD instruction. We excluded the MAD operation because conclusions reached for the SAD apply to the MAD as the MAD is trivially computed from the SAD. In particular, we considered two example implementa-

tions assuming 16x1 and a 16x16 pel blocks. The proposed sample implementation schemes compute the SAD in 3 or 4 machine cycles respectively. We were able to propose a hardware execution unit that can perform the SAD in a small number of cycles, because of the following two reasons:

- we have substituted complex operations (i.e subtract and absolute operation) with two simple operations (determining and inverting the smallest).
- we have substituted the subtractions and the accumulation operation by one multi-operand addition.

This speed advantage is especially beneficial for data-dependent algorithms, such as the three-step search algorithm. These algorithms need the SAD of the blocks in their first step to compute the addresses of the blocks in the second step.

In the chapter to follow we will introduce an hardwired Paeth codec, which is of use in coding and decoding images using the Portable Network Graphics standard.

Chapter 4

Paeth Prediction and Coding

This chapter describes an execution unit capable of computing the Paeth predictor [94, 95], as used in the Portable Network Graphics (PNG) standard. The PNG standard specifies a lossless compression method for real-world pictures. It features five prediction schemes, of which the modified Paeth predictor [95] is the most computational intensive. This chapter focuses on a hardware implementation of the Paeth predictor and a hardware Paeth codec, capable of computing three different quantities:

- the Paeth predictor of three inputs,
- the difference of the current pixel and the Paeth predictor of the other inputs (used for coding),
- the sum of the coded input and the Paeth predictor of the other three inputs (used for decoding).

It is interesting to note that the results of our investigation suggest that no more than two cycles are required to perform these operations. The cycle is assumed to be comparable to a general purpose ALU cycle. It is also noted that depending on the mode of operation, the proposed mechanism produces the predictor or the (de/en)-coded pixel value.

The chapter is organized as follows: first we provide some an introducing discussion by examining an assembly program for a general purpose architecture performing the Paeth operation. We continue with additional background information about the PNG standard and the Paeth predictor. In Section 4.3 we describe the software routines used in the Paeth predictor and the modifications needed to allow a hardware implementation. In Section 4.4 we describe

an extension of the predictor, the Paeth codec, which can code or decode a pixel with no additional cycle-time. In Section 4.5 we evaluate the unit and finally in Section 4.6 we conclude the discussion with some final remarks.

4.1 Introduction

A standard that is gaining popularity in image coding and compression is the Portable Network Graphic (PNG) [95] standard. The PNG standard has been created as an alternative solution to Graphics Interchange Format (GIF) [96]. PNG is a lossless compression scheme, based on predictive coding and deflate compression. The standard is written so that its speed would be high and that it would benefit from the multimedia extensions of general purpose processors [18]. One of the key-features of PNG is the ability to choose out of five predictors for the predictive coding, namely: none, up, left, average and Paeth. All five predictors can operate with the value of only three adjacent pixels. These pixels are positioned above, left and left-above the current pixel. After the prediction step, the coded pixel data is stored in a bit-stream which is subsequently deflated using the gzip algorithm [97].

The Paeth predictor is normally computed in software. The routine used for this is defined in the PNG standard and shown here as Figure 4.1.

```
int predict ( int a, b, c )
{
int p, pa, pb, pc
p = a + b − c                          /* this is the initial estimate */
pa = abs( p − a )                      /* distance of each member to the */
pb = abs( p − b )                      /* initial estimate */
pc = abs( p − c )
if ( pa ≤ pb ) and ( pa ≤ pc ) return ( a )        /* return */
else if ( pb ≤ pc ) return ( b )       /* element nearest to p, */
return ( c )                           /* in a,b,c tie-break order */
}
```

Figure 4.1: The Paeth encoding routine according to the PNG specification [95].

To compute the Paeth predictor on a SunSparc 10 processor, 21 instructions are needed, including 6 branches. This code is shown in Figure 4.2 for reference. It should be noted that in the figure all register names have been named to the

original variable names for readability. Note that the caller expects to see the result in the register where it stored a. If the code would be scheduled for a Very Large Instruction Word (VLIW) [3] machine, the computation of pa, pb and pc could be parallelized, but the number of cycles would still be around 15 (assuming pipelined operations).

```
_predict:
        add a,b,temp
        sub temp,c,p
        subcc p,a,pas
        bneg,a L2
        sub 0,pas,pa
L2:
        subcc p,b,pbs
        bneg,a L3
        sub 0,pbs,pb
L3:
        subcc p,c,pcs
        bneg,a L4
        sub 0,pcs,pc
L4:
        cmp pa,pb
        bg L9
        cmp pb,pc
        cmp pa,pc
        ble L8
        cmp pb,pc
L9:
        ble L8
        mov b,a
        mov c,a
L8:
        retl
        nop
```

Figure 4.2: Sparc-10 pseudo assembler code.

To improve the execution speed, we propose a hardwired Paeth prediction unit, which computes the Paeth predictor of a set of three input values in two ma-

chine cycles[1]. As the predictor is selected from the input values, and the critical path is the control of the output selectors, we can also precompute the difference or the sum of a fourth input (d) with each of the three inputs. This means that we are also able to compute the coded or decoded value within the same two machine cycles. We compute the predictor by directly computing the distances of the initial estimator (p) to each input, and selecting the input which has the smallest distance.

The proposed scheme operates as follows:

- Direct computation of the distance of each input to the initial estimate.
- Compare these distances using Carry-generators.
- Select the input with the lowest distance.

For the codec unit, we precalculate three temporal results, which are the sum (decoding) or difference (encoding) of the current pixel and each of the inputs, and select one of these precalculated values. Using this scheme, the critical path is not affected, and yet the number of executed operations is increased.

4.2 Background

In this section we will provide some information about compression of images, the type of compression schemes and in particular the compression method used by PNG.

We begin by indicating that pictures may contain information in a structured way, and this structure introduces redundancy. Redundancy means that all information may not be necessary to convey the message.

In order to diminish the size of the picture on the storage device (e.g. disk) or the transmission time over the Internet, we need a method to extract and describe the redundancy in pictures. There are two basic ways of compression, lossless and lossy. Lossy compression could be appropriate for photographic pictures. The decompression of a lossy compressed image results in a similar but not necessary 100% identical picture. The legitimacy for such a scheme relies on the fact that "small" differences are not visible or distinguishable to the human eye, thus losing information can be acceptable.

[1] A machine cycle assumed here is comparable to the cycle time of a general purpose ALU.

Lossless compression is mostly used for synthetic pictures, or computer generated graphics. When decompressing a lossless compressed image, the original image is restored, which is a 100% identical copy of the original.

PNG has been defined as an alternative solution to GIF and it has been developed with the following objectives [95]:

- Simple and portable: developers should be able to implement PNG easily.
- Legally unencumbered: to the best of knowledge of the PNG authors, no algorithms under legal challenge are used. (Some considerable effort has been spent to verify this.)
- Well compressed: both indexed color and true-color images are compressed as effectively as in many other widely used lossless format, and in most cases more effectively.
- Interchangeable: any standard-conforming PNG decoder must read all conforming PNG files.
- Flexible: the format allows for future extensions and private add-ons, without compromising interchangeability of basic PNG.
- Robust: the design must support full file integrity checking as well as simple, quick detection of common transmission errors.

These objectives have resulted in a rather quick adoption by both industry and the Open Source Software movement. PNG has become the native format for graphics in Microsoft Office 97 [98] and is also used more and more on the Internet. In addition, PNG is now in the early stages of international standardization, thanks largely to its inclusion in the VRML97 standard [99]. It is expected to become a joint ISO/IEC standard (ISO/IEC 15948) [99] by early 2000.

The "core business" of PNG is the compression of graphical data. This is achieved using predictive coding (filtering) and standard deflate compression [97]. In order to improve the compressibility of the data, filtering is used. The purpose of filtering is the extraction of spatial redundancy by recording only the differences between the current pixel and its prediction. This prediction is mostly based on the neighbors of the current pixel. PNG offers five filter types, namely: None, Up, Left, Average and Paeth. In this chapter, we develop

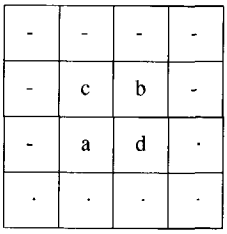


Figure 4.3: The definition of a, b, and c according to the PNG specification [95].

a codec scheme for the Paeth predictor. The other four predictors are trivial to implement in hardware.

Figure 4.3 gives the naming conventions of the Paeth predictor within the PNG standard. The pixels denoted with "-" are already transmitted and no longer of interest, the pixel denoted with "d" is the current pixel and the pixels denoted with "." will be transmitted in the future. Note that the naming, and as a result of the naming the tie-break-order, are not identical in the PNG standard and in the original Paeth predictor as defined in 1991 by Alan W. Paeth [94].

The Paeth predictor is used to achieve compression, as it is anticipated that the difference between the predictor and the actual pixel will be small. This is caused by the spatial redundancy in the picture. Pixels generally do not differ much from their neighbors. The difference between the predicted pixel value and the actual pixel value is transmitted, after compression using deflate techniques. The resulting differences are generally small numbers, which need fewer bits for transmission. In this way compression is achieved.

As indicated earlier the five filters of PNG are None, Up, Left, Average, and Paeth. The first is very simple, each pixel is predicted as zero. The Up and Left predictor depend on one of the neighbor pixels. The Average filter depends on both these neighbors. The Paeth predictor is the most complicated predictor. It depends on three neighboring pixels. The naming of the neighboring pixels is shown in Figure 4.3. Summarizing the five predictors and their function, the following holds true:

None The None filter transmits (d)

Up The Up filter transmits $(d - b)$

Left The Left filter transmits $(d - a)$

Average The Average filter transmits $(d - (a + b)/2)$

Paeth The Paeth filter transmits $(d - Paeth(a, b, c))$

The Paeth predictor makes an initial prediction of d with the following formula: $P_d = a + b - c$, where a, b and c are defined according to Figure 4.3. If the intensity of the picture is gradually increasing of decreasing in that area, this prediction is perfect. However, this predictor doesn't fulfill all four criteria which Paeth defined in his article. Those are:

Identity	$P(a, a, a)$	$=$	a	(1)
Transposition	$P(a, b, c)$	$=$	$P(b, a, c)$	(2)
Complementation	$P(a, b, c)'$	$=$	$P(a', b', c')$	(3)
Membership	$P(a, b, c)$	$\in$	$\{a, b, c\}$	(4)

Table 4.1: The criteria for the Paeth predictor.

The first criterion is trivial, if all pixels in some neighborhood have the same value, it is safe to predict this value.

The second criterion ensures that if rows and columns are interchanged, the result doesn't change. This seems contradictory to the tie-break order. However, in the original Paeth predictor one can't construct a case where this is of importance. In the PNG Paeth predictor, the second criterion doesn't hold anymore. However, as column and row interchanging is not used in PNG, it is not important.

The third criterion yields that the inversion of the raster (like a negative of a photo) results in an inversion of the predictor.

The fourth criterion ensures that the predictor is always within the bounds of the pixel-value range for each input combination. (No need for clipping/saturation.) This fourth criterion is implemented by selecting the element closest to (a+b-c) as the predictor. This may yield a less optimal predictor, but there are no worries about bounds. It means that all computations and comparisons within the Paeth predictor should be done using enough precision in order to produce a correct result.

Figure 4.1 on page 60 gives the software routine to compute the Paeth predictor. The PNG standard [95] deviates slightly from the original Paeth predictor in that it operates as follows:

1. It defines that all operations are done on 8-bit (byte) quantities. This simplifies the computations and encourages the use of MMX [18] instructions.

2. It defines a new naming scheme for the incoming pixels, which essentially results in a different tie-breaking order.

3. It defines the pixel left to the current pixel to be the last transmitted pixel. In case of interlacing, there can be quite a number of pixels "in between". The same holds for the vertical direction. The previous line is interpreted as the previous transmitted line. This technique reduces the amount of bufferspace needed. The different interlacing schemes ensure that the a, b, c, and d pixels are still the corners of a rectangle.

All these modifications are made in order to make a software implementation "straight forward" and possibly fast, maybe at the expense of a slightly less optimal compression. The first modification make the implementation of a hardware accelerator for the Paeth predictor or even a Paeth codec feasible.

In this chapter, we will assume the definition of the Paeth predictor as given in the PNG definition [95]. Under this definition, all operations are performed on 8-bit quantities (bytes). These bytes are interpreted as unsigned binary numbers. If the image is composed of 16-bit deep RGB values, (48 bits per pixel), the operations are performed 6 times on 6 bytes independently of each other. If the data is only black-and-white, 1 bit per pixel, this 1 bit is packed as a byte and consequently treated as a byte.

Due to the possible interlacing schemes defined in the PNG standard, the previous received pixel is not necessarily adjacent to the current pixel. This holds also for the vertical direction. The Paeth predictor selects the previous pixel as the last transmitted pixel, and the previous line as the last transmitted line. This might result in a slightly less optimal compression, but decreases the memory-requirements of the (de)coding process.

4.3 Computing the Paeth predictor

The routine displayed as Figure 4.1 is the Paeth predictor as defined in the PNG standard [95]. The inputs a, b and c are 8-bit, treated as unsigned binary numbers. However, the variables internal to the routine are not to be truncated to 8 bits.

In order to propose a hardware implementation of this routine, we rewrite it. In Figure 4.4 we can distinguish three steps. These steps will be exposed in the hardware implementation. In the first step, we compute pas, pbs and pcs. These are the signed variants of pa, pb and pc, denoted as 10 bit, two's

complement intermediate numbers. In the second step, we compute $Test\_1$, $Test\_2$ and $Test\_3$. The third step is the selection of the right input as the output.

---

int predict (int a, b, c)
{
int pas, pbs, pcs
bool $Test\_1$ $Test\_2$ $Test\_3$

pas = ($b - c$)
pbs = ($a - c$)
pcs = ($a + b - 2c$)

$Test\_1$ = ($|pas| \leq |pbs|$)
$Test\_2$ = ($|pas| \leq |pcs|$)
$Test\_3$ = ($|pbs| \leq |pcs|$)

If $Test\_1$ and $Test\_2$ return (a)
else if $Test\_3$ return (b)
return (c)
}

Figure 4.4: The Paeth Algorithm simplified, so that it can be mapped to hardware. Note that the result of this routine and the original (Figure 4.1) is the same.

The computation of pas, pbs and pcs is done by an adder circuit [39, 48]. As the range of the unsigned bytes a, b and c is from 0 to $(2^8 - 1)$, the variable pcs can range from $0 + 0 - 2 * (2^8 - 1)$ to $2^8 - 1 + 2^8 - 1 - 2 * 0$, which is from $-2^9 + 2$ to $2^9 - 2$. This range is just covered by a 10-bit two's complement number, which ranges from -2^9 to $2^9 - 1$. Although pas and pbs are representable as 9-bit two's complement numbers, we also represent them as 10-bit two's complement numbers to facilitate the subsequent comparisons and to preserve the regularity of the unit. In binary notation, this leads to the following additions:

$$pas = \quad (b - c) \quad = (00b + 11\overline{c} + 1) \tag{4.1}$$

$$pbs = \quad (a - c) \quad = (00a + 11\overline{c} + 1) \tag{4.2}$$

$$pcs = \quad (a + b - 2c) \quad = (00a + 00b + 1\overline{c}1 + 1) \tag{4.3}$$

As can be concluded from the previous three formulas, a sign-extension takes

place to make all the input numbers 10 bits long. The negative value of $-c$ is computed using an inversion and the addition of a hot-one.

The computation of *pcs* involves a 3 to 1 addition, where one of the operands is shifted one position to the left (multiplied by 2). This is accommodated by using an extra level of Full-Adders, which performs a carry-save addition of the three operands, resulting in a sum and a carry word. These are then added in a 2-1 binary adder. A graphical representation is shown in Figure 4.5.

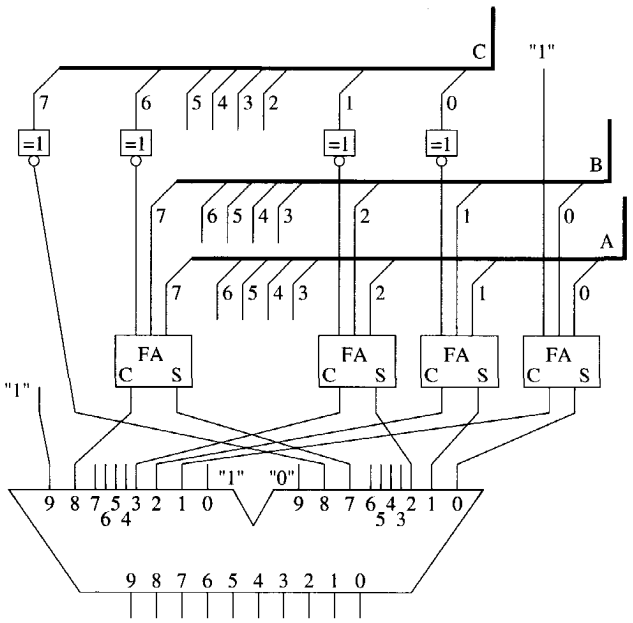


Figure 4.5: The adder used to compute *pcs* from a, b and c.

After the computation of *pas*, *pbs* and *pcs*, there are several ways to compute $Test\_1$, $Test\_2$ and $Test\_3$. To compute $Test\_1$ we have to find out whether $|pas| \leq |pbs|$. We can use a carry-based comparison of *pas* and *pbs*. This means that we add them in some form and that the resulting carry reflects whether the inequality was true or false.

We first have to adjust the signs of *pas* and *pbs*. In order to compare them, we check whether $|pbs| - |pas| \geq 0$. In order to facilitate this, we have to make sure that *pbs* has a positive sign and *pas* has a negative sign. If the sign is opposite, we invert the operand and add a hot-one to the result. This hot one is taken care off in the addition. If both *pas* and *pbs* are inverted, there are two hot ones. This means the carry-generator needs a special structure to

accommodate this. This is implemented as a layer of half-adders, as shown in Figure 4.6. It should be noted that in the figure bit number 9 is the Most Significant bit, the Sign-bit. The outputs 0 to 9 are not used, and need not be computed. They are only shown for clarity.

The test $|pbs| - |pas| \geq 0$ is now modified to $pb_{pos} + pa_{neg} \geq 0$. The test for carry_out is basically the test for result $\geq 2^{10}$. We have to keep in mind that the sign-bit of pas is interpreted as a positive number here, with value 2^9 in stead of -2^9. We are therefore essentially adding 2^{10}. The binary summation is therefore: $pa_{neg} + 2^{10} + pb_{pos} \geq 2^{10}$ So if $pb_{pos} + pa_{neg} \geq 0$ the binary addition $pb_{pos} + pa_{neg}$ generates a carry_out. Figure 4.6 gives a graphical representation of the unit which computes Test_1 from pas and pbs.

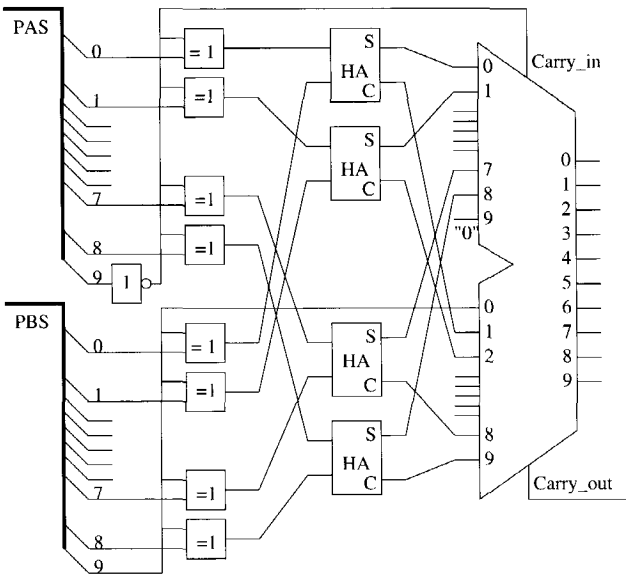


Figure 4.6: The computation of Test_1.

Test_2 and Test_3 are computed using similar logic. The control of two mux-boxes is trivial. Figure 4.7 gives a graphical representation of the entire unit.

4.4 Implementation of a Paeth codec

A possible extension of this unit is the extension to a Paeth codec, which has not only the a, b and c input, but also uses the to be encoded or decoded pixel, d.

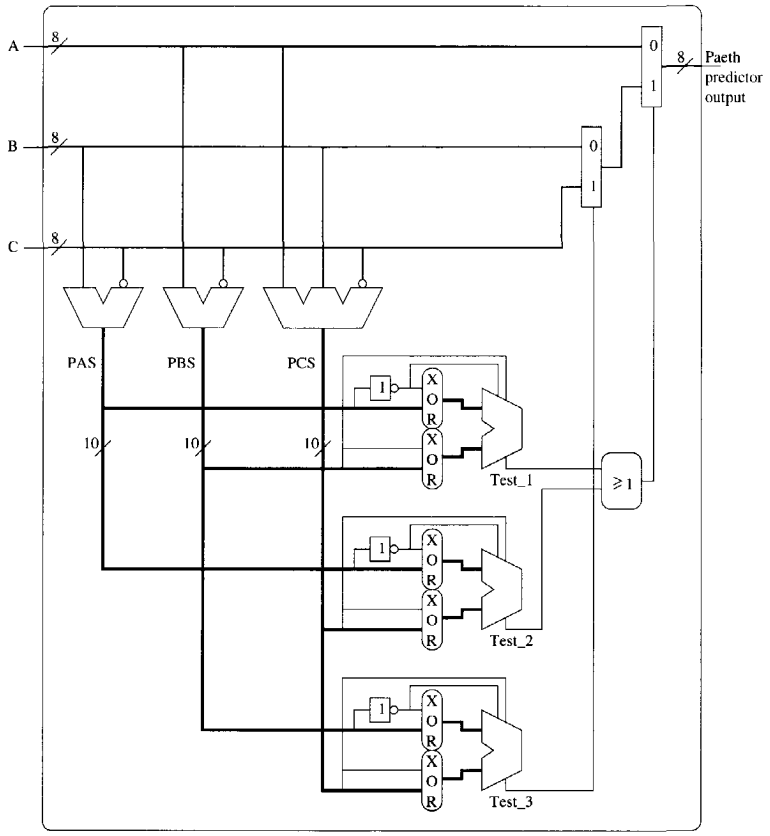


Figure 4.7: Proposed implementation of the PNG Paeth predictor.

When the prediction is known, the coding is simply subtracting the prediction from the actual data.

$$\begin{aligned} Coded\_Data &= Actual\_Data - Prediction \\ &= d - pred \end{aligned}$$

Decoding is done by adding the prediction and the received coded data.

$$\begin{aligned} Original\_Data &= Coded\_Data + Prediction \\ &= d + pred \end{aligned}$$

These operations are done modulo 256, as defined in the standard [95].

The most obvious accommodation of this addition/subtraction step is after the multiplexers, but this causes an increase of the latency of the unit. This is shown in Figure 4.8.

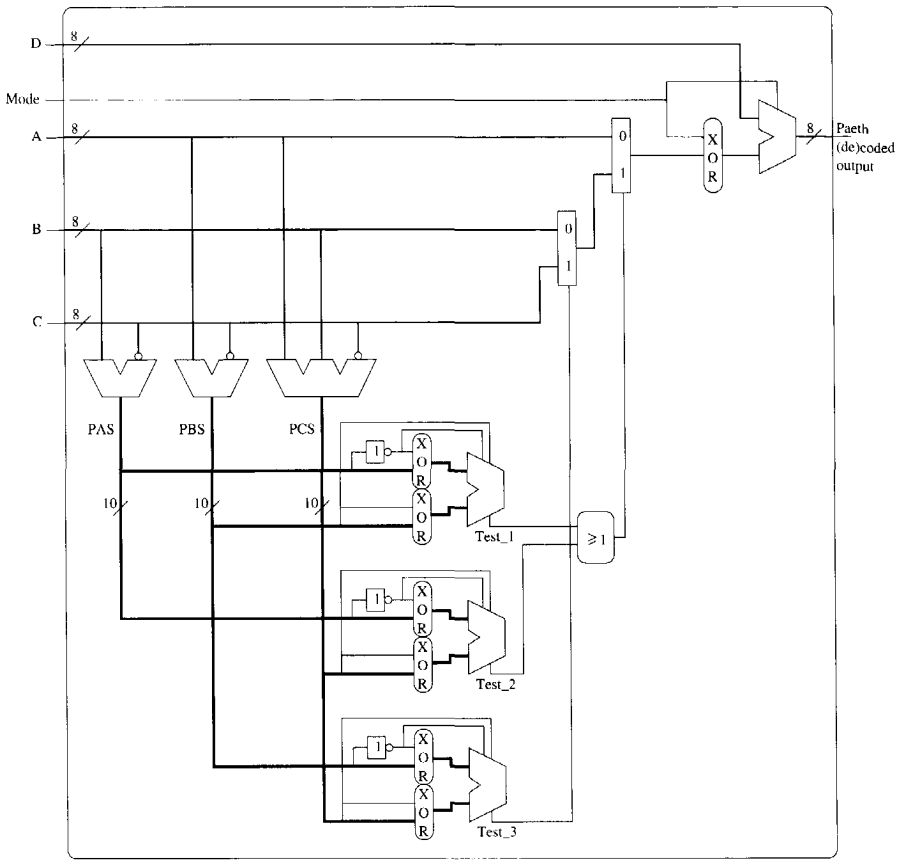


Figure 4.8: Direct implementation of the PNG Paeth codec.

As the predictor is always equal to one of the input values, and the path from the input to the data-input of the multiplexers is empty, (as opposed to the path from the input to the control-input of the multiplexers) the adding/subtraction step can be accommodated there. Thereby we effectively make a 2-cycle, 4-input Paeth codec, with three operation modes:

Code In this mode, the d input is set to the value of the to be coded pixel. The result is the coded pixel. (Mode=1)

Decode In this mode, the d input is set to the received coded input and the output is the reconstructed pixel. (Mode=0)

Predict In this mode, the d input is set to zero and the operation is set to decoding. This results in the output reflecting the normal Paeth predictor. (Mode=0)

The program-notation for this is given in Figure 4.9. A graphical representation of the implementation of this optimized execution unit is shown in Figure 4.10.

---

int codec (int a, b, c, d, $mode$)
{
int pas, pbs, pcs
unsigned int Res_a, Res_b, Res_c
bool $Test\_1$ $Test\_2$ $Test\_3$

pas = ($b - c$)
pbs = ($a - c$)
pcs = ($a + b - 2c$)
Res_a = $(d + (1 - 2 * mode) * a)$
Res_b = $(d + (1 - 2 * mode) * b)$
Res_c = $(d + (1 - 2 * mode) * c)$

$Test\_1$ = ($|pas| \leq |pbs|$)
$Test\_2$ = ($|pas| \leq |pcs|$)
$Test\_3$ = ($|pbs| \leq |pcs|$)

If $Test\_1$ and $Test\_2$ return (Res_a)
else if $Test\_3$ return (Res_b)
return (Res_c)
}

Figure 4.9: The Paeth codec algorithm, with the subtraction before the selection. This results in a codec which is as fast as a predictor in hardware.

4.5 Hardware and time estimations

We have presented a sample implementation of a 4-input Paeth codec, capable of coding and decoding images using the Paeth predictor as described in the

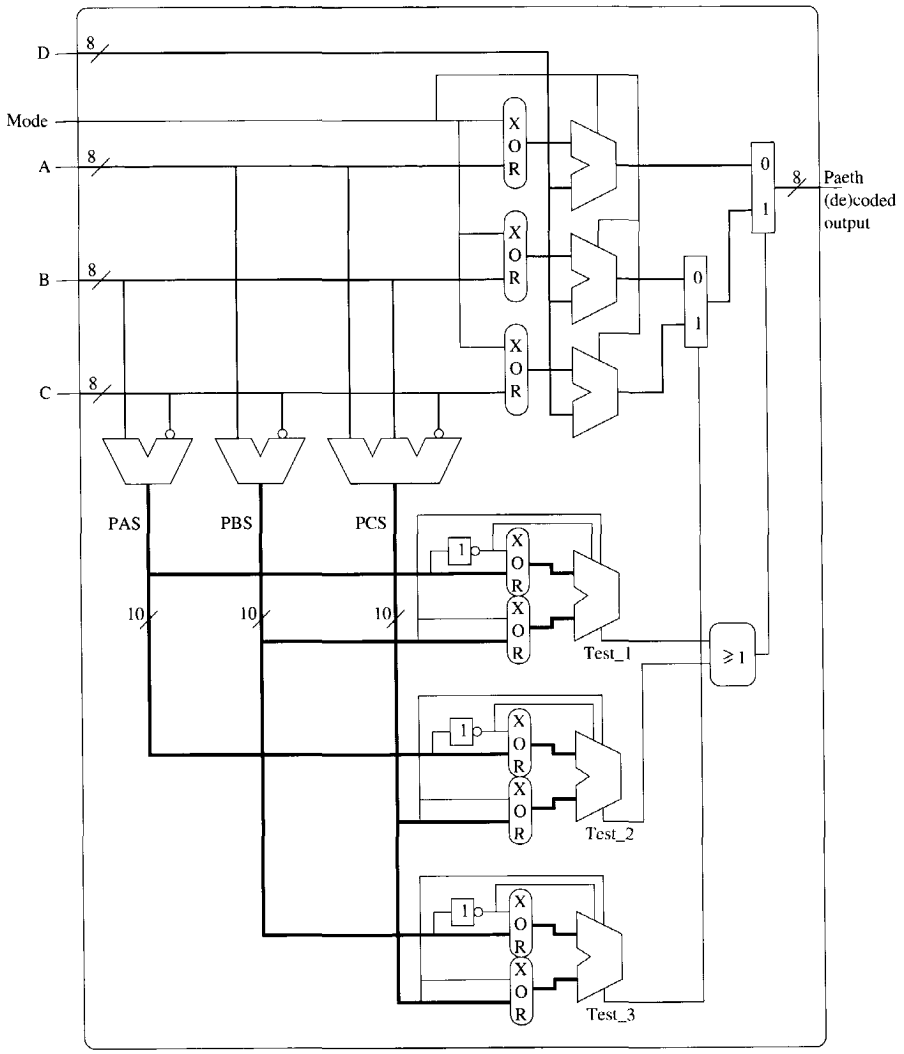


Figure 4.10: Optimized implementation of the PNG Paeth codec.

PNG standard. It computes which of the inputs to use as Paeth predictor and in parallel (de)codes the fourth input with all inputs of the predictor. After this step, the right result is chosen.

The critical path of the codec unit is the control of the output multiplexers. It is basically two levels of binary adders long. The path to the data-input of the multiplexers is only one level of adders long. We estimate that the speed of this unit equals that of a standard two-cycle multiply unit or requires two ALU

cycles. This estimation is based on the fact that the critical path is basically two times an adder delay, which is bounded by the ALU cycle time.

The hardware requirements for the proposed unit are relative modest for a 4-input unit. We need:

- 2 2-input 10-bit adders
- 1 3-input 10-bit adder
- 3 3-input 10-bit carry-generators
- 2 2-input 8-bit selectors
- 3 2-input 8-bit adders for the codec unit.

4.6 Conclusions

We presented a Paeth codec, which computes either the Paeth predictor, the Paeth-coded or the Paeth-decoded pixel with only a two cycle delay. Compared to a 21 machine instruction SPARC implementation, this is a ten-fold speedup. The unit is developed for PNG coding, but can also be useful in other graphic schemes. The hardware requirements for the unit are modest, in the order of 9 10-bit adders.

We compute the Paeth predictor using the following steps:

- Direct computation of the distance of each input to the initial estimate.
- Compare these distances using Carry-generators.
- Select the input with the lowest distance.

The codec variant computes the difference or sum of the value to be encoded/decoded in parallel to the first step, it does not alter the last two steps. The final step remains the selection of the right output. This means no additional cycles are needed to compute the encoded or decoded pixel value.

In the next chapter we consider hardwired accelerators for Median, Minimum, Maximum and Mean. We consider such operation for implementation as we will show a method to compute the Median of three input numbers. The Minimum and Maximum of three input numbers is a trivial extension of this Median Unit. For the computation of the Mean of the three inputs, we will reuse the Add-Multiply-Add unit we presented in Chapter 2.

Chapter 5

Median, Max, Min, and Mean

In the previous chapter, the Paeth unit was introduced. This unit is capable of computing the paeth predictor, which can be used to compress images using the Portable Network Graphics (PNG) standard [95]. In this chapter we will introduce an extension of the Paeth unit by which it can additionally compute the Median of three inputs. This median is used in video-deinterlacing, which is needed for displaying normal (interlaced) video on a non-interlaced computer screen or a modern, high-end television set. We will further extend the Paeth logic, so that is can also compute the maximum and minimum of the three inputs. Furthermore, we introduce an extension of the Add-Multiply-Add unit, described in Chapter 2 by which the Add-Multiply-Add unit can compute the Mean of three inputs. The overall direction of this chapter is to introduce new instructions and show that they can be of advantage when compared to their software equivalent.

The chapter is organized as follows. For background purposes we briefly discuss video-deinterlacing, a process where broadcasted interlaced video is converted to non-interlaced video. In this process, the median can be used to compute the missing lines of pixels. Consequently we discuss the potential performance improvement that could occur when using a hardwired median unit that performs the median. We furthermore show that such a unit can be easily build either as a stand-alone unit or as a simple extension of the paeth unit. We show that the median of three inputs can be performed in one machine cycle, providing substantial improvement over software solutions which perform the same functions. Additionally we show that trivial extensions to the unit can provide the maximum and the minimum of three inputs. Finally we describe an extension of the Add-Multiply-Add unit to perform the mean

Figure 5.1: Three successive fields of an interlaced video sequence.

operation of three inputs.

5.1 Background

Traditionally, television broadcast is performed using interlaced video. In interlaced video, a frame is divided into lines and only half of the lines of a frame are scanned and transmitted. This is performed as follows. The video sequence is divided into even and odd frames, in which the even fields only contain even lines and each odd fields only contain odd lines. See Figure 5.1 for an example. Note that the first and third frame in the figure only contain the even lines (starting from 0) and that the middle frame only contains the odd lines. Interlacing is a form of decimation, where the number of pixels to be transmitted is halved. Consequently, the after-glow time of the phosphorus on a television screen has to be long enough to "camouflage" this effect. In order to display such a video sequence on a non-interlaced screen, such as a computer-screen, video-deinterlacing must be performed.

Deinterlacing means that the missing pixel-values are "computed" from spatial or temporal neighbors. Three simple methods exist, namely:

- line repetition,
- line averaging,
- line insertion

Line repetition displays each line twice, which results in non-square pixels. Line averaging uses the average of the pixel above and the pixel below the current pixel as the value for the current pixel. Both methods are a form of repetition in space, and they perform well only on moving sequences, where the lower spatial resolution is not noticed. On still-images the lower resolution

will be noticeable. The third method, denoted as line insertion, inserts the line of the previous field. This is therefore a repetition in time, which only performs well in non-moving sequences, where the previous line indeed contains the right data. Line insertion comes closest to what we see on a standard television set, where the after-glow of the phosphorus performs this task. If line insertion is used on moving scenes, the distortion becomes visible. From the above, we can conclude that the best method depends on the fact whether there is movement in (part of) a frame. This in itself is non-trivial to determine.

Median filtering [100] provides an intermediate between the three methods. It selects the pixel-value which has the middle value of the pixel above, the pixel below and the pixel from the previous line, where previous means previous in time. The rationale behind this is that if there is motion, the median will likely be one of the spatial neighbors and if there is no motion, the median will be the previous pixel. The median filter thereby effectively chooses the appropriate method from the above three methods [100, 101].

Consider Figure 5.2 for an operative example of this scheme. This figure shows the result of three kinds of deinterlacing. On the top row, the original picture (Wilbur[1]) is shown as reference. The second row contains two even frames and one odd frame. The rightmost frame is shifted one pixel to the right, in order to simulate motion. The third row shows two frames reconstructed with line repetition, which results in relative low resolution. The fourth row shows the result of line insertion, which performs ideal in the absence of motion, but the right picture shows the undesirable result when there is motion between the frames. The median filter on the last row performs reasonable in both circumstances.

5.2 Computing the median: the software solution

The median is defined as the middle value of a sorted list of input values. For an odd number of inputs (three being the odd number in the case considered here), the median is defined as the value of the middle number of the input values. For the three input median a number of formulations have been used to compute the desired result. For example, the three input median can be computed by the following:

$$med = a + b + c - max(a, b, c) - min(a, b, c) \tag{5.1}$$

[1] Wilbur is the mascot of the Gnu Image Manipulation Program, GIMP.

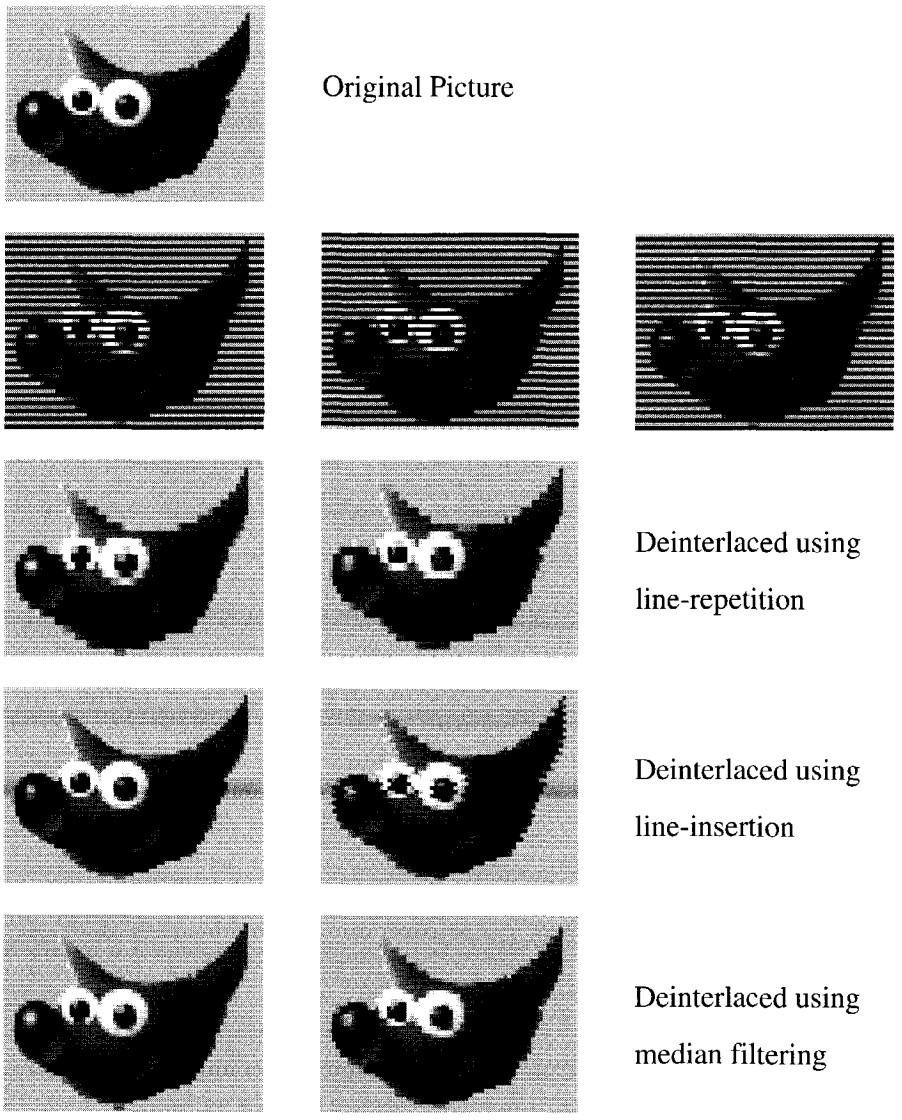


Figure 5.2: Deinterlacing examples.

This gives a mathematically correct result, even if a+b+c would result in an overflow, as long as no saturation[2] logic is used. The two subtractions will ensure that the result is within bounds, because two of the inputs are subtracted from the non-saturated result.

[2]Saturation means that the result is set to MAX if there is an overflow.

In a more common approach, the middle input is directly selected, instead of computing the Max and the Min of the 3-input values and subtracting them of the sum of the inputs. This approach is implemented in the program of Figure 5.3, which computes the Median of three inputs, a, b, and c, and places the result in the register denoted as a.

```
_median:
        cmp a,b
        bge L1
        cmp a,c
        bge L2
        cmp b,c
        bge L2
        mov c,a
        b L2
        mov b,a
L1:
        bl L2
        cmp b,c
        bl L2
        mov c,a
        mov b,a
L2:
        retl
        nop
```

Figure 5.3: SPARC-10 pseudo assembler code for computing the median.

To establish an approximate performance for the code fragment we note that this program would require between 6 instructions (cmp bge cmp bge cmp retl) and 9 instructions (cmp bge cmp bge cmp bge mov b retl), depending on the input values. Due to the branches, the number of cycles for this code fragment is variable. In the worst case scenario the code fragment requires 9 machine cycles if we assume that all instructions require at most one machine cycle to execute.

5.3 Efficient computation of the Median

To improve the performance of the computation of the median we propose a hardware unit which computes the median in two steps as follows:

Step 1: compute three inequations ($a \geq b, b \geq c, a \geq c$)

Step 2: select the median based on the outcome of the computations in the previous step.

This two-step approach is also reported in [100]. In the first step the inequations are determined and in the second step a decode mechanism is employed. Certain control logic is also added to make the final selection. In our description we explicitly determine the logic needed to compute the median plus we provide it incorporated in more complex logic and generalize it to an ALU type new unit. More specifically the mechanisms we describe perform the following:

Step 1

In the first step we compute the three inequations: $a \geq b$; $b \geq c$; $a \geq c$. As the Paeth computation uses the ten bits wide two's complement notation of both $b - c$ and $a - c$ as intermediate results, we will use the same hardware. As an example we show that the testing of $b \geq c$ is equivalent to the testing whether $b - c \geq 0$. This is a simple test on the sign-bit of the result of the subtraction. The inequation holds true if the sign-bit is zero. The carry-out of the addition is the inverse of this sign-bit[3]. That means that the carry-out is one if-and-only-if the sign-bit is zero. The computation of the first inequation is performed in a similar way, we compute the ten-bit, two's complement number $a - b$ and use the carry-out of that subtraction.

To summarize, we only have to add one carry-chain generator to the Paeth logic to perform the computation of the $a \geq b$ inequation. For the other two inequations, the logic is already there. Let $Test_1$, $Test_2$ and $Test_3$ represent the result of $a \geq b$, $b \geq c$ and $a \geq c$ respectively:

$$Test_1 \iff a \geq b \iff 00a + 11\overline{b} + 1 \geq 2^{n+2} \tag{5.2}$$

$$Test_2 \iff b \geq c \iff 00b + 11\overline{c} + 1 \geq 2^{n+2} \tag{5.3}$$

$$Test_3 \iff a \geq c \iff 00a + 11\overline{c} + 1 \geq 2^{n+2} \tag{5.4}$$

Note that the input-numbers are extended with two bits to ten bit inputs. This stems from the paeth unit, which needs the 10-bit result of $b - c$ and $a - c$.

[3]This is a common property of the two's complement representation. If a negative and a positive number are added, the result will always be in bounds and the carry-out will always be the inverse of the sign-bit.

We could also pick the 9^{th} bit of this subtraction, thereby reducing the critical path a little.

Step 2

Based on the three resulting carries, we select one of the three operands as result, using Table 5.1. The Table 5.1 has four columns. The first three columns describe the eight output combinations of the comparisons. The fourth column determines which of the operands needs to be chosen as the median. Note that two out of the eight combinations cannot occur, as there is no set of a, b, and c for which these conditions hold true. One would need $a \geq b$, $b \geq c$ and $a \ngeq c$. From the first two conditions, one can deduce that $a \geq c$, This means that if $Test_1$ and $Test_2$ are True, $Test_3$ also results in True.

$a \geq b$ $Test_1$	$b \geq c$ $Test_2$	$a \geq c$ $Test_3$	Median
True	True	True	b
True	True	False	–
True	False	True	c
True	False	False	a
False	True	True	a
False	True	False	c
False	False	True	–
False	False	False	b

Table 5.1: Median table.

In order to implement the median(a,b,c) instruction we note that the two steps require three carry generators and the implementation of the logic implied from Table 5.1. All median requirements can be performed from the logic depicted in Figure 5.4. This can be proven by the following: the first part of the logic (the carry-generators) determines which of the three conditions ($Test_1$, $Test_2$ and $Test_3$) holds true. That is carry generator 1 determines $Test_1$ ($a \geq b$), carry generator 2 determines $Test_2$ ($b \geq c$) and carry generator 3 determines $Test_3$ ($a \geq c$).

The control of the selectors in Figure 5.4 is deduced as follows: the information of Table 5.1 is rewritten as the following Karnaugh diagram:

In order to select a, we need a 0 on selector 7. In order to select b we need a 1 on selector 7 and a 0 on selector 6. Finally, to select c we need a 1 on selector 6 and 7. This is shown in the karnaugh diagrams in Figure 5.6.

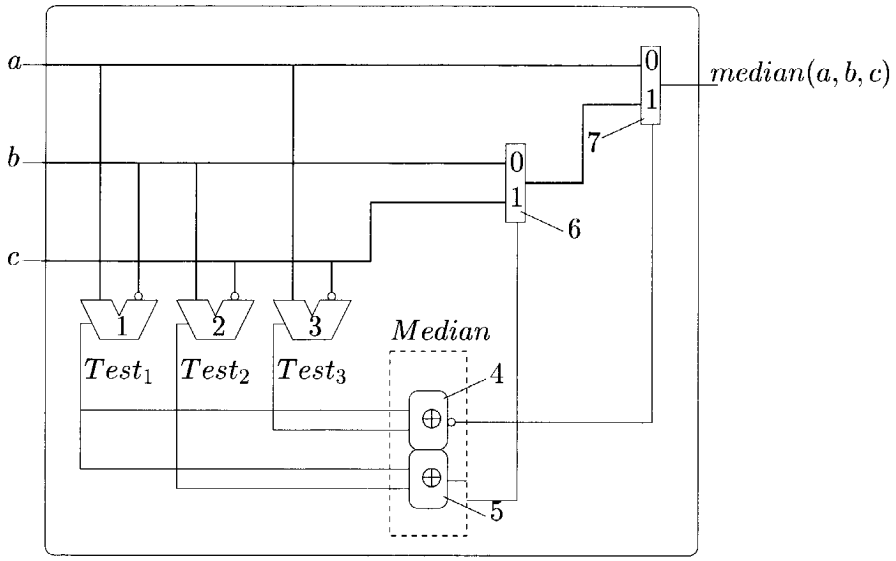


Figure 5.4: Implementation of the 3-input Median Filter.

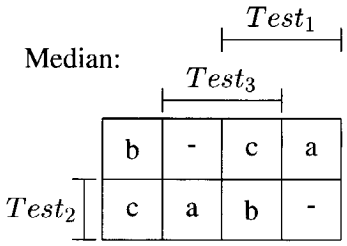
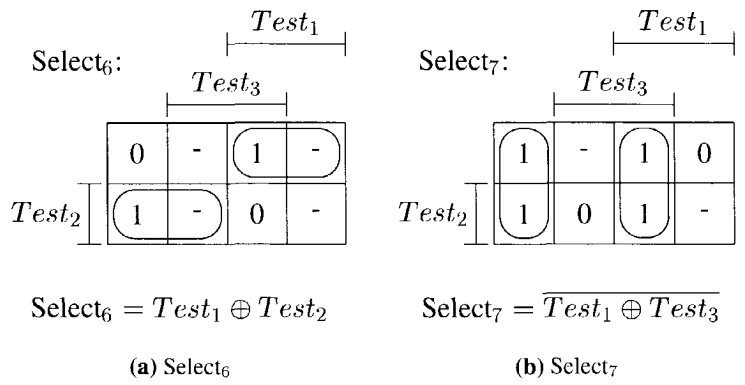


Figure 5.5: Karnaugh diagram of the select function for the median.

From Figure 5.6 we deduce the following formula's for the two select signals: $Select_6 = Test_1 \oplus Test_2$ and $Select_7 = \overline{Test_1 \oplus Test_3}$.

Finally we observe the following. The proposed implementation has as critical path: one carry generator, one XOR and multiplexing logic. This implies the unit will not require more than one machine cycle to perform the median, given the availability of the three inputs. In essence the scheme described in this section provides substantial advantage over the software solution and the approach proposed in the previous section. It must be noted that the implementation reported in Figure 5.4 will most likely require no more than one machine cycle as it requires less logic stages than are required by a general purpose ALU design. If the median operation is included in the Paeth unit,



(a) Select$_6$ **(b)** Select$_7$

Figure 5.6: Karnaugh diagrams and formula's of the two select signals.

the operation may take two cycles. This additional cycle time is viewed as a performance/hardware tradeoff.

5.4 Extending the Median Unit with Min and Max

Minimum (min) and Maximum (max) of a series of numbers can also be useful for general purpose architectures. Given that min and max can be obtained with a trivial extension to the median unit we described in the previous section, we will describe such an extension in this section. Clearly, once we have computed the three inequations, it is trivial to select the maximum and the minimum of the three inputs. In order to select them, we extend the unit with an extra set of multiplexers, which controls the control-signals to the output multiplexers.

To comprehend the impact of the one cycle min and max instructions, we consider the implementation of the min and max of three numbers in software using a common instruction set. The routine implementing this is depicted in Figure 5.7. This program will take 7 machine cycles to compute the minimum or maximum of the three inputs. Clearly the performance improvement is rather obvious and substantiates the addition of such instructions.

To compute the min and max of three inputs, we need to determine which input to select on the basis of the three computed test signals. These selection requirements are deduced from Table 5.2 which determines the operand to select as min and max output. The implementation of the table can be found in Fig-

```
_maximum:
        cmp a,b
        bge,a L2
        mov b,a
L2:
        cmp a,c
        bge,a L3
        mov c,a
L3:
        retl
        nop
```

```
_minimum:
        cmp a,b
        ble,a L2
        mov b,a
L2:
        cmp a,c
        ble,a L3
        mov c,a
L3:
        retl
        nop
```

Figure 5.7: SPARC-10 pseudo assembler code for the maximum and minimum of three numbers.

ure 5.8. The proof of the implementation can be done following the discussion of the median with some appropriate additions. It can also be observed that the additional 3-1 multiplexer is in the critical path. This should not create a critical path problem. This means our implementation of min, max and median should still require no more than one machine cycle.

$a \geq b$ $Test_1$	$b \geq c$ $Test_2$	$a \geq c$ $Test_3$	Min	Median	Max
True	True	True	c	b	a
True	True	False	–	–	–
True	False	True	b	c	a
True	False	False	b	a	c
False	True	True	c	a	b
False	True	False	a	c	b
False	False	True	–	–	–
False	False	False	a	b	c

Table 5.2: Requirements for Min, Max and Median.

5.5 The mean of three numbers

The mean of three inputs is used in a number of multimedia applications for filtering purposes. Given that division by three can not be achieved with shifting,

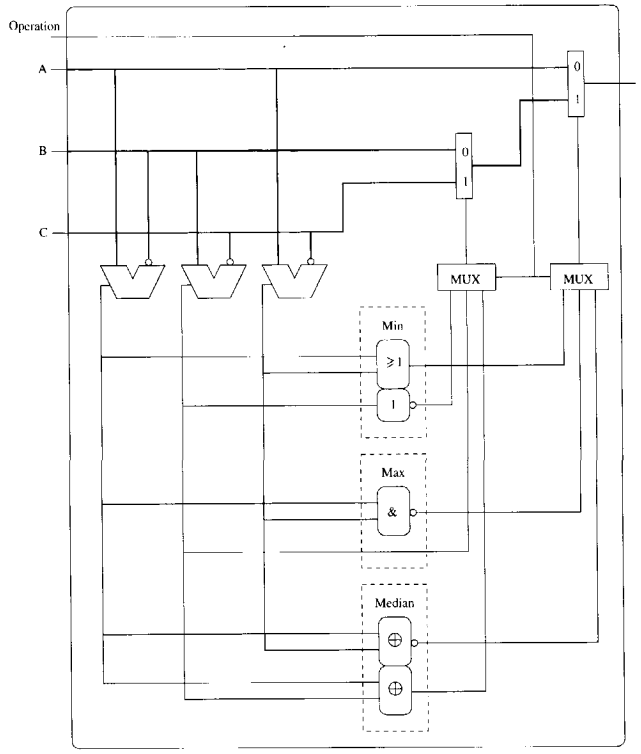


Figure 5.8: Extension of the 3-input Median Filter with a three input max and min function.

a full division operation has to be performed. This implies that the operation needs to employ division hardware, which in general is expensive in terms of performance and/or hardware. In this section we propose a modification of the add-multiply-add unit of Chapter 2 that allows the entire mean of three numbers to be performed by the add-multiply-add unit, providing substantial improvement when compared to the direct implementation of the mean function by software on an existing instruction set.

In order to compute the mean of three numbers, we perform the following. Assume that the input numbers, a, b and c are in two's complement notation and that each number is representable in 8 bits. Perform the following steps:

Step 1: Compute X and Y as the carry-save sum of $A + B + C$. In this step, three eight-bit numbers are converted into two nine-bit numbers, using a carry-save addition. Both are sign extended to ten-bit, two's complement numbers.

Step 2: Compute ((X+Y)*341+511) using a ten-bit input Add-Multiply-Add unit described in Chapter 2. The result is a 21-bit number in two's complement notation. Note that $\frac{341}{1024} \approx \frac{1}{3}$, and $\frac{511}{1024}$ is as close as we can come to $\frac{1}{2}$ using 10-bit 2's complement numbers[4].

Step 3: Shift the result 10 positions to the right (discarding the lower 10 and the three upper bits).

Proof of correctness: Assume the result of the mean-operation is P. This implies that the sum of A, B and C equals either $3P$, $3P + 1$ or $3P - 1$ (because the mean is equal to one third of this sum, rounded to the nearest whole number).

If we undo step 3 (the rounding) we observe that the result of step 2 is $P * 1024 + rest$, where $0 \leq rest \leq 1023$. This means we have to proof that rest remains within its range for all P which can be expected as output. The range of the mean of any number of inputs is equal to the range of the input numbers. This means we have to prove that for all values P can take as a two's complement, n-bit number, the rest is within bounds. Consequently we have to prove whether rest remains within bounds for all P in the output range, taking the rounding into account. That is $P * 1024 + rest$ equals $(SUM) * 341 + 511$ for all input combinations.

There are three cases to consider:

Case 1: Sum=3P

$$P * 1024 + rest = 3P * 341 + 511$$
$$P + rest = 511$$
$$rest \in \{0...1023\}$$
$$P \in \{-511...512\}$$

Case 2: Sum=3P+1

$$P * 1024 + rest = (3P + 1) * 341 + 511$$
$$P + rest = 511 + 341 = 852$$
$$rest \in \{0...1023\}$$
$$P \in \{-171...852\}$$

[4]There is no power of 2 which is dividable by 3, so we have to approximate.

Case 3: Sum=3P-1

$$P * 1024 + rest = (3P - 1) * 341 + 511$$
$$P + rest = 511 - 341 = 170$$
$$rest \in \{0...1023\}$$
$$P \in \{-853...170\}$$

The intersection of these three sets is $P \in \{-171...170\}$. This covers the range of the input values, $\{-128...127\}$, so the result will be correct for all input combinations.

Figure 5.9 gives a graphical representation of the mean-of-three unit, implemented using a Carry-Save adder and the Add-Multiply-Add unit introduced in Chapter 2. It is noted that hardware is added to perform a carry-save opera-

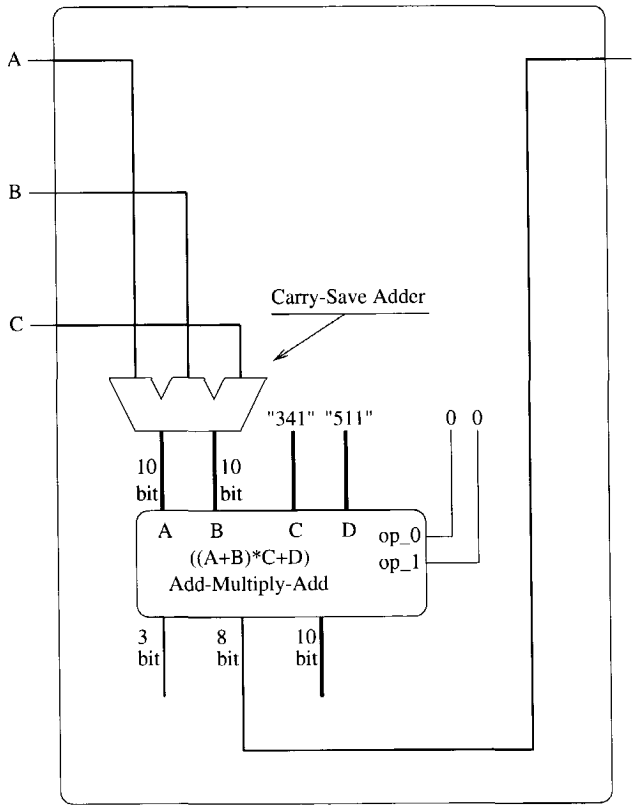


Figure 5.9: Implementation of the 3-input Mean unit using an Add-Multiply-Add unit.

tion which may produce a critical path delay problem (the extra logic required is the XOR of 3 inputs). If this poses a problem, then an additional cycle could be added in order to perform the mean operation. This would still be an improvement when compared to an implementation which does not use this hardware extension. (An addition of three requires already two machine cycles and then there is a divide instruction to performed).

5.6 Conclusions

This chapter has been dedicated to the introduction of some instructions which can be implemented with small modifications of the hardware required for the instructions proposed in the previous chapters. More specifically we have proposed new instructions for the median, the maximum, the minimum and the mean. All instructions operate on three input numbers and produce a single output value. For the proposed instructions we have shown hardware implementations, which indicate the following.

- No complex modifications are required to implement the instructions using available hardware.
- No cycle time penalties would be introduced, (except perhaps for the mean) if the additional instructions proposed in this chapter are implemented.

In particular it has been shown that one cycle is required to perform the median, min and max. Depending on the technology, 2 or 3 cycles are required to perform the mean (the third cycle could be added to avoid critical path delay problems). Furthermore it has been shown that there are substantial advantages in using our proposal when compared to software solutions. More specifically we have shown that performing the median in hardware requires 1 cycle, while performing the same operation in software using an usual instruction set (e.g. Sun SPARC) would require 6 to 9 cycles. For min and max, which are also 1 cycle operations, the software solution will require 7 cycles. Finally, the mean of three number as proposed will require 2 to 3 cycles, which is substantial less than using a commonly available instruction set which requires two addition operations and a division.

While this chapter has aimed to include additions to the requirements of previous instructions to perform new instructions, not all considerations have been

given to accommodate all instructions to the execution unit(s) of our proposed set of instructions. Clearly hardware is designed to incorporate the execution of multiple instructions in a "single" unit. This requirement is rather common as it is not acceptable to have individual units for individual instructions for most of the instructions in the instruction set. An example of this in general purpose machines is the ALU which normally operates on circa 10 to 15 instructions, depending on the instruction set. The next chapter is dedicated to this issue, the design of an execution unit that accommodates multiple instruction requirements.

Chapter 6

Putting it all together

In the previous chapters, four different units have been introduced. In Chapter 2 and in [102], we introduced the Add-Multiply-Add unit, capable of computing $(A \pm B) * C \pm D$. In Chapter 3 and in [28] we introduced the SAD unit, capable of computing the Sum of Absolute Differences between two blocks of pixels data. The Paeth unit we introduced in Chapter 4 and in [103] is an aid in image compression, Finally, in Chapter 5 we introduced the Median/Min/Max/ unit which can substantially improve the performance of video deinterlacing and the three input mean unit, which is an extension of the Add-Multiply-Add unit. The goal of this chapter is to explore which configurations or combinations of these units can be put together so that hardware is saved via hardware re-use. In building a multimedia extension to a general purpose processor, it might be more convenient to follow the general processor paradigm which has few units with each unit used by multiple instructions. That is, it is of interest to combine the units into one single execution unit.

This chapter is organized as follows. We first describe the instructions and associated opcodes. This is done in approximately the same style as the SPARC architecture reference manual [104]. After this specification we conclude with a sample implementation.

6.1 Instruction Set

We begin by noting that one common aspect of all described units is that they operate on more than the usual two operands (with the exception of some of the simpler instructions of the Add-Multiply-Add unit and possibly the SAD

instruction, which can be computed over any even number of operands). This suggests combining the units in one general unit, which has at least four inputs. If two instruction(slot)s are needed to specify the operands for the unit, we could think of delivering two results as well. This would only need some extra logic on the output-stage. Note that some combinations would be impossible. Another solution to this problem is the specification of register pairs. This register addressing mode implies that only two source registers are specified, and that the other two source registers are implicit. The machine instruction ama **r2**, **r6**, **r5** would specify that **r5** gets the result of (**r2+r3**)*__r6+r7__. This means if a register is specified as a source register, its "upper neighbor" is also specified implicitly. The "upper neighbor" of a register is the subsequent register.

In order to determine the various instructions we first discuss briefly the data-types used in the instructions. The basic data-types we support are:

unsigned byte (8 bit) This is frequently used for pixel-data, where it contains one color-component, or the luminance of a pixel.

signed byte (8 bit) This is used for intermediate results.

unsigned half-word (16 bit) This is also used for pixel-data in very high color-depths. The PNG standard supports this, but states that the 16 bits should be treated as two independent bytes.

signed half-word (16 bit) This is used for audio samples.

unsigned word (32 bit) This is used for intermediate results.

signed word (32 bit) This is used for intermediate results.

unsigned double-word (64 bit) This is used for intermediate results.

signed double-word (64 bit) This is used for intermediate results.

The base of our representation is 2. The bit enumeration is from high to low, which means that the most significant bit has the highest number and the least significant bit has the number 0. We note that all signed numbers are represented as two's complement numbers. The supported instructions can be found in Table 6.1, from which it can be observed that the instructions have been divided into 4 categories.Furthermore the instructions are divided according to the required number of inputs. The specifics of all categories are reported in the following subsections.

1 input	2 input	3 input	4 input
Add-Multiply-Add unit			
Negate	Add	Add-Multiply	Add-Multiply-Add
	Subtract	Subtract-Multiply	Add-Multiply-Subtract
	Multiply	Multiply-Add	Subtract-Multiply-Add
		Multiply-Subtract	Subtract-Multiply-Subtract
		Add-Add	
		Add-Sub	
		Sub-Add	
		Sub-Sub	
		Mean	
Sum-Absolute-Difference unit			
SAD_Stream	SAD_2	SAD_Accumulate	SAD_4
Paeth unit			
		Paeth_PNG	Paeth_PNG_Encode
			Paeth_PNG_Decode
Minimum Maximum Median unit			
		Min	Min encode
			Min decode
		Max	Max encode
			Max decode
		Median	Median encode
			Median decode

Table 6.1: Instruction set.

The SPARC architecture [104] explicitly specifies the possibility of a co-processor and also reserves some opcode space for such a co-processor. The general form of instructions executed by the co-processor is specified in Figure 6.1. In the following text we will assume a SPARC [104] Coprocessor-like

Not affecting Condition Codes:

1 0	rd	1 1 0 1 1 0	rs1	opc	rs2	
31	29	24	18	13	4	0

Affecting Condition Codes:

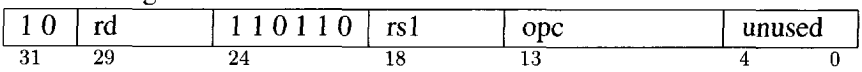
1 0	rd	1 1 0 1 1 1	rs1	opc	rs2	
31	29	24	18	13	4	0

Figure 6.1: The operation format specifying two source register pairs.

implementation, with two of the four source registers are implicitly following from the register number of the other two registers. In other words, if register **r2** is specified as **rs1** (source register 1) then the value of **rs1a** is the value of register **r2** and the value of **rs1b** is the value of register **r3**. This method is also used to specify operands which cannot be held in one register to the floating-point unit. We assume an extension to version 8 of the SPARC architecture manual, as we assume all registers to be 64 bit wide.

For the Neg, Add, Sub, Sad_Stream and SAD_2 instructions, which use only one source register(pair) the format is specified in Figure 6.2.

Not affecting Codition Codes:



Affecting Condition Codes:

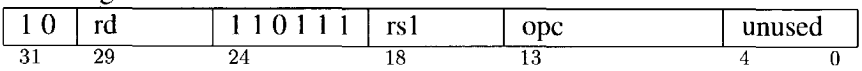


Figure 6.2: The operation format specifying one source register pair.

Bit-numbers 31 and 30 are 1 and 0 respectively, which specifies a type 3 coded instruction [104, p. 149]. Furthermore, bits 24 to 19 are fixed at 110110 for an instruction that doesn't change the condition codes or 110111 for an instruction that does affect the condition codes. Nine bits, denoted as opc and numbered 13 through 5 in the above figures, can be used to specify one of 512 possible coprocessor instructions. While this seems a huge number, we will show that we will use almost all of these nine bits. Bits 29 through 25 are used to specify one of the 32 registers as destination register. Bits 18 through 14 and bits 4 through 0 specify the two source registers, **rs1** and **rs2** respectively. This approach needs 10 bits of the 32 bits of the instruction to specify the source operands. We should note that all operations are register-to-register operations. For this version of the instruction set we do not support direct memory operations or operations using immediates. As most of our instructions use 4 source operands, and we do not want to spend 20 of the 32 bits of the instruction for the specification of the source registers, we use a partial implicit register addressing scheme. We feel that although this might harm flexibility, it is still flexible enough to be useful. In our partial implicit register addressing scheme only two source registers are specified, the other two are the "upper neighbors" of the two specified registers. The "upper neighbor" of a register is the register which has its number incremented by one. That is, if we specify **r4** and **r9** as source registers in an instruction, the hardware will use the contents of **r4**, **r5**,

r9 and **r10** as the operands of the instruction.

The floating point processor of the SPARC processor uses a similar scheme in order to work with floating point numbers which use more bits than a register is wide. A floating point number is then loaded into **r4** and **r5** for instance. While the floating point unit generates a trap whenever a source register is not aligned properly, our unit explicitly allows the specification of "odd" source registers to specify a register pair. While in floating point notation the two parts are used to specify one value, the register pairs of our coprocessor unit contain two independent values. As a side note it should be noted that the "upper neighbor" of **r31** is assumed **r0**. In the discussion of the instructions, **rs1** is used to denote source register 1 and **rs2** is used to denote source register 2. Furthermore, **rs1a** is the same register as **rs1**, and **rs1b** is the "upper neighbor" of **rs1**. Equivalently, **rs2a** specifies the same register as **rs2**, and **rs2b** is the "upper neighbor" of **rs2**.

In the following subsections we describe all the instructions which our proposed unit can execute.

6.1.1 Add-Multiply-Add Instructions

This subsection describes the Add-Multiply-Add family of instructions. The mnemonics of each instruction are in the column denoted as opcode. The column opc defines the bit settings of bits 13 through 5 of the opcode and the column denoted as operation gives a general description of the operation that is performed.

Supported data types:

The Add-Multiply-Add instructions operate only on 32-bit, signed or unsigned inputs.

opcode	opc	operation
AMA	00x111100	Add-Multiply-Add
AMS	00x111101	Add-Multiply-Subtract
SMA	00x111110	Subtract-Multiply-Add
SMS	00x111111	Subtract-Multiply-Subtract
MA	00x110100	Multiply-Add
MS	00x110101	Multiply-Subtract
AM	00x011110	Add-Multiply
SM	00x011111	Subtract-Multiply

opcode	opc	operation
AA	00x101110	Add-Add
AS	00x101111	Add-Subtract
SA	00x101110	Subtract-Add
SS	00x101111	Subtract-Subtract
ADD	00x001110	Add
SUB	00x001111	Subtract
MUL	00x010100	Multiply
NEG	00x000010	Negate

opc_6, which is denoted as x in the above table is used to select between signed and unsigned operation.

(un)signed	Operator length	opc_6
unsigned	word	0
signed	word	1

Suggested Assembly Language Syntax	
ama	rs1,rs2,rd
ams	rs1,rs2,rd
sma	rs1,rs2,rd
sms	rs1,rs2,rd
ma	rs1,rs2,rd
ms	rs1,rs2,rd
am	rs1,rs2,rd
sm	rs1,rs2,rd
aa	rs1,rs2,rd
as	rs1,rs2,rd
sa	rs1,rs2,rd
ss	rs1,rs2,rd
add	rs1,rd
sub	rs1,rd
mul	rs1,rs2,rd
neg	rs1,rd

The semantics of all instructions is as follows.

- In the Add-Multiply-Add (AMA) instruction the contents of **rs1a** is added to the contents of register **rs1b** and the result is multiplied by the contents of register **rs2a**. The contents of register **rs2b** is added to the result, and the final result is written to register **rd**.
- In the Add-Multiply-Subtract (AMS) instruction the contents of **rs1a** is

added to the contents of register **rs1b** and the result is multiplied by the contents of register **rs2a**. The contents of register **rs2b** is subtracted from this, and the final result is written to register **rd**.

- In the Subtract-Multiply-Add (SMA) instruction the contents of **rs1b** is subtracted from the contents of register **rs1a** and the result is multiplied by the contents of register **rs2a**. The result is added to the contents of register **rs2b**, and the final result is written to register **rd**.
- In the Subtract-Multiply-Subtract (SMS) instruction the contents of **rs1b** is subtracted from the contents of register **rs1a** and the result is multiplied by the contents of register **rs2a**. The contents of register **rs2b** is subtracted from the result, and the final result is written to register **rd**.
- The Multiply-Add instruction (MA) ignores the contents of register **rs1b**, and multiplies the contents of register **rs1a** with the contents of register **rs2a**. To this result the contents of register **rs2b** is added and the final result is written to register **rd**.
- The Multiply-Subtract instruction (MS) multiplies the contents of register **rs1a** with the contents of register **rs2a**. From this result the contents of register **rs2b** is subtracted and the final result is written to **rd**.
- The Add-Multiply instruction (AM) adds the contents of register **rs1a** to the contents of register **rs1b**. This result is multiplied with the contents of register **rs2a** and written to **rd**.
- The Subtract-Multiply instruction (SM) subtracts the contents of register **rs1b** from the contents of register **rs1a**. This result is multiplied with the contents of register **rs2a** and written to **rd**.
- The Add-Add instruction (AA) adds the contents of registers **rs1a, rs1b** and **rs2b** and writes the result to register **rd**.
- The Add-Subtract instruction (AS) adds the contents of registers **rs1a** and **rs1b** and subtracts the contents of register **rs2b**. The result is written to register **rd**.
- The Subtract-Add instruction (SA) subtracts the contents of register **rs1b** from the contents and **rs1b** and adds the contents of register **rs2b**. The result is written to register **rd**.
- The Subtract-Subtract instruction (SS) subtracts the contents of registers **rs1b** and **rs2b** from **rs1a** and writes the result to register **rd**.

- The Add operation (ADD) adds the contents of registers **rs1a** and **rs1b** and writes the result to register **rd**.
- The Subtract operation (SUB) subtracts the contents of register **rs1b** from the contents of **rs1a** and writes the result in register **rd**.
- The Multiply operation (MUL) multiplies the contents of register **rs1a** and the contents of register **rs2a** and writes the result in register **rd**.
- The Negate operation (NEG) negates the contents of register **rs1b** and writes the result to register **rd**.

This can be summarized by the following table, which specifies the mathematical operation for each of the above instructions.

opcode	formula
AMA	rd ← (([rs1a]+[rs1b])*[rs2a])+[rs2b]
AMS	rd ← (([rs1a]+[rs1b])*[rs2a])-[rs2b]
SMA	rd ← (([rs1a]-[rs1b])*[rs2a])+[rs2b]
SMS	rd ← (([rs1a]-[rs1b])*[rs2a])-[rs2b]
MA	rd ← (([rs1a])*[rs2a])+[rs2b]
MS	rd ← (([rs1a])*[rs2a])-[rs2b]
AM	rd ← (([rs1a]+[rs1b])*[rs2a])
SM	rd ← (([rs1a]-[rs1b])*[rs2a])
AA	rd ← (([rs1a]+[rs1b]))+[rs2b]
AS	rd ← (([rs1a]+[rs1b]))-[rs2b]
SA	rd ← (([rs1a]-[rs1b]))+[rs2b]
SS	rd ← (([rs1a]-[rs1b]))-[rs2b]
ADD	rd ← (([rs1a]+[rs1b]))
SUB	rd ← (([rs1a]-[rs1b]))
MUL	rd ← (([rs1a])*[rs2a])
NEG	rd ← (-[rs1b]))

As we will show in the implementation section of this chapter, the Add-Multiply-Add group of instructions only operates on 32 bits wide words. These inputs can be either signed or unsigned. As all registers are 64 bits wide, and the result of the Add-Multiply-Add unit is in general $2n + 1 = 65$ bits wide, we store the most significant bit in the condition codes.

General description:

The Add-Multiply-Add family of instructions takes four operands, of which the first two, specified by **rs1a** and **rs1b**, are summed or sub-

tracted and multiplied by the third operand, specified by **rs2a**. Finally, the fourth operand, specified by **rs2b**, is added or subtracted.

Overflows:

The Add-Multiply-Add family of instructions can easily generate an overflow, as the result is in general 2n+1 bits wide for n-bit input numbers, see Section 2.1. As our unit only operates on 32-bit inputs and the result is still a 64-bit register, we only have to store one extra bit. This bit is stored in the condition codes.

Traps:

None.

Notes to the implementor:

The connection of the opcode-bits to the implementation of the execution unit are given in the following table:

opc_0	op_0 of AMA
opc_1	op_1 of AMA
opc_2	Block A to zero
opc_3	Block B to zero
opc_4	Block C to 1
opc_5	Block D to zero
opc_6	Unsigned (0) or Signed (1) operation
opc_7	0 to specify AMA operation
opc_8	0 to specify AMA operation

6.1.2 Mean3 Instruction

The Mean3 instruction computes the mean of three input numbers.

Supported data types:

The Add-Multiply-Add instructions operates only on 32-bit, signed or unsigned inputs.

opcode	opc	operation
MEAN3	11xxxxxx0	Mean of the three inputs

- The Mean3 instruction computes the mean of **rs1a**, **rs1b** and **rs2a** and writes the result in **rd**.

Suggested Assembly Language Syntax
mean3 rs1,rs2,rd

opcode	formula
MEAN3	rd ← ([rs1a]+[rs1b]+[rs2a])/3

General description:

The Mean of Three operation computes the mean of three input numbers, specified as the contents of **rs1a**, **rs1b** and **rs2a**. This is done by using the Add-Multiply-Add hardware.

Traps:

None.

Notes to the implementor:

Note that for n-bit inputs, the Add-Multiply-Add hardware needs to be $n+2$ bits wide and the shifter on the bottom of the tree-reducer needs to arithmetically shift the result over $n+2$ bits to the right, neglecting any bits shifted out. As we also want to support unsigned operation, another bit is added to the width of the needed Add-Multiply-Add unit.

6.1.3 Sum of Absolute Differences (SAD) Instructions

We describe two architectural implementations of the SAD instruction. The first implementation fits in the scheme of four source registers and one destination register, where two of the four source registers follow implicit from the other two source registers. The second architectural implementation is a streaming mode implementation, which operates on two implicit registers, which are loaded using a special stream-load instruction. We do not supply a MAD instruction, as the rounding should not take place before the final addition.

Supported data types:

The Sad instructions operate on bytes or half-words, unsigned inputs.

SAD-Register

opcode	opc	operation
SAD	01xxxxx00	Sum of Absolute Differences
SAD2	01xxxxx11	Sum of Absolute Differences
SADa	01xxxxx01	Sum of Absolute Differences Accumulated

The semantics of the Sum of Absolute Differences instructions are as follows:

- The Sum of Absolute Differences (SAD) instruction computes the Sum Absolute Difference of the sub-words of **rs1a** and **rs1b** and the sub-words of **rs2a** and **rs2b** and stores the result in register **rd**.
- The Sum of Absolute Differences (SAD2) instruction computes the Sum Absolute Difference of the sub-words of **rs1a** and **rs1b** and stores the result in register **rd**.
- The Sum of Absolute Differences Accumulated (SADa) instruction computes the Sum Absolute Difference of the sub-words of **rs1a** and **rs1b**, adds this to the contents of register **rs2a** and stores the result in register **rd**.

(un)signed	Operator length	opc_6
unsigned	byte	0
unsigned	half-word	1

Suggested Assembly Language Syntax	
sad	rs1,rs2,rd
sad2	rs1,rd
sada	rs1,rs2,rd

opcode	formula
SAD	rd ← SAD([rs1a],[rs1b],[rs2a],[rs2b])
SAD2	rd ← SAD([rs1a],[rs1b])
SADa	rd ← [rs2a]+SAD([rs1a],[rs1b])

General description:

The four input registers, specified as two register pairs, are subdivided into bytes or half-words. For each pair of bytes or half-words, the smallest is subtracted from the largest. The results of these subtractions are accumulated, together with **rd** in case of accumulation, and written to **rd**. Note that the actual implementation operates slightly different to allow faster operation. See Chapter 3.

Overflow:

None. Using a 16 bit result half-word is sufficient for a 16x16 bytes SAD and a 24 bit result word is sufficient for a 16x16 half-word SAD.

Traps:

None.

Notes to the implementor:

Note that the actual subtraction of the smallest does not take place. Instead, the smallest of each pair of inputs is inverted (subtracted from $2^n - 1$)and a constant is added to accommodate for this. See Section 3.2. This constant is equal to $2^q - n * (2^p - 1)$, where q is the number of bits in the destination register, n is the number of pixels the unit operates on and p is the width of the source operands.

SAD-Streaming

This section describes the opcodes for the streaming mode of SAD. We assume there are two implicit source registers, which are loaded by a special autoincrement load instruction. The length of this register is not defined within the architecture. However, there is a special, implementation dependent register from which the length of the registers can be deduced during runtime. Note that the length of the source registers is anticipated to be equal to the memory-bandwidth of the processor. In other words, we assume that the processor will be able to process at least 8 pixels in parallel. Note also that the specified source register is a general purpose register containing the SAD over the previous pixels.

opcode	opc	operation
SADs	01xxxxx10	SAD-Streaming-mode

Suggested Assembly Language Syntax
sads rs1,rd

opcode	formula
SADs	rd ← rs1a + SAD([Implicit Reg_1], [Implicit Reg_2])

General description:

This instruction operates on pairs of unsigned bytes or pairs of unsigned half-words. The input registers are split in the appropriate number of

items and for each pair the smallest is subtracted from the largest. After this step all bytes or half-words are summed up together with a constant and source register **rla** to form the result, which is written to **rd**.

Note that this instruction is capable of accumulation by specifying the same register as source and destination. Therefore a separate SADsa instruction (SAD-Streaming-Accumulating) is not necessary.

Overflow:

None. Using a 16 bit result half-word is sufficient for a 16x16 bytes SAD, using a 24 bit result word is sufficient for a 16x16 half-word SAD.

Traps:

Note that the width of the implicit source registers is not specified. Therefore, the programmer does not know how many cycles are needed to do a full block. The SADs instruction will generate a trap if an attempt is made to read further then a block.

Notes to the implementor:

Note that the actual subtraction of the smallest doesn't take place. Instead, the smallest is inverted and a constant is added to accommodate for this. See Section 3.2. This constant is equal to $2^q - n * (2^p - 1)$, where q is the number of bits in the destination register, n is the number of pixels the unit operates on and p is the width of the source operands.

6.1.4 PAETH Instructions

The Paeth family of instructions is used in PNG [95] encoding and decoding.

opcode	opc	operation
PAE	100000000	Compute Paeth predictor
PAEE	100000011	Encode pixel using the Paeth predictor
PAED	100000010	Decode pixel using the Paeth predictor

The semantics of the Paeth instructions are as follows:

- The Paeth (PAE) instruction computes the Paeth predictor from the contents of registers **rs1a**, **rs1b**, and **rs2a** and writes the result to register **rd**.
- The Paeth-Encode (PAEE) instruction computes the Paeth predictor from the contents of registers **rs1a**, **rs1b**, and **rs2a** and subtracts this

value from the contents of register **rs2b**. The result is written to register **rd**.

- The Paeth-Decode (PAED) instruction computes the Paeth predictor from the contents of registers **rs1a**, **rs1b**, and **rs2a** and adds this value to the contents of register **rs2b**. The result is written to register **rd**.

Suggested Assembly Language Syntax	
pae	rs1,rs2,rd
paee	rs1,rs2,rd
paed	rs1,rs2,rd

opcode	formula
PAE	rd $\leftarrow$ Paeth_predict([rs1a],[rs1b],[rs2a])
PAEE	rd $\leftarrow$ [rs2b]-Paeth_predict([rs1a],[rs1b],[rs2a])
PAED	rd $\leftarrow$ [rs2b]+Paeth_predict([rs1a],[rs1b],[rs2a])

General description:

The Paeth instructions compute the Paeth predictor according to the PNG [95] standard. The Paeth instructions only operate on unsigned bytes. Each input register is split in the appropriate number of bytes, and for each byte the operation is performed independently and in parallel. The Paeth predictor is defined as the value of that input a,b or c which is closest to $a + b - c$ in the tie-break-order a, b, c. Where a is the pixel left from the current pixel, b is the pixel above the current pixel, and c is the pixel left-above the current pixel, see Figure 4.3. Note that the PNG standard [95] specifies the Paeth predictor slightly different then Paeth [94] defined it originally.

Traps:

None.

Notes to the implementor:

The connection of the opcode-bits to the implementation of the execution unit are given in the following table:

opc_0	op_0 of PMMM
opc_1	op_1 of PMMM
opc_2	$Mode$ of PMMM
opc_3	Block D to zero
opc_4	0, as the Paeth predictor is only defined for unsigned byte
opc_5	0, as the Paeth predictor is only defined for unsigned byte
opc_6	0, as the Paeth predictor is only defined for unsigned byte
opc_7	0 to specify PMMM operation
opc_8	1 to specify PMMM operation

6.1.5 MMM Instructions

The MMM family of instructions includes the Median, Minimum and Maximum of three inputs. These operations are available for all defined data-types.

opcode	opc	operation
MED	10xxx1101	Median
MEDE	10xxx1111	Encode using the Median
MEDD	10xxx1110	Decode using the Median
MIN	10xxx0101	Minimum
MINE	10xxx0111	Encode using the Minimum
MIND	10xxx0110	Decode using the Minimum

opcode	opc	operation
MAX	10xxx1001	Maximum
MAXE	10xxx1011	Encode using the Maximum
MAXD	10xxx1010	Decode using the Maximum

The xxx in the table above defines the dataformat of the operands of the instruction. This is shown in the following table.

(un)signed	Operator length	opc_6	opc_5	opc_4
unsigned	byte	0	0	0
signed	byte	1	0	0
unsigned	half-word	0	1	0
signed	half-word	1	1	0
unsigned	word	0	0	1
signed	word	1	0	1
unsigned	double-word	0	1	1
signed	double-word	1	1	1

The semantics of the MMM family of instructions is as follows:

- The Median instruction (MED) computes the Median of the contents of registers **rs1a**, **rs1b** and **rs2a** and writes the result to register **rd**.
- The Median Encode instruction (MEDE) computes the Median of the contents of registers **rs1a**, **rs1b** and **rs2a**. This value is subtracted from the contents of register **rs2b** and the result is written to register **rd**.
- The Median Decode instruction (MEDD) computes the Median of the contents of registers **rs1a**, **rs1b** and **rs2a**. This value is added to the contents of register **rs2b** and the result is written to register **rd**.
- The Minimum instruction (MIN) computes the Minimum of the contents of registers **rs1a**, **rs1b** and **rs2a** and writes the result to register **rd**.
- The Minimum Encode instruction (MINE) computes the Minimum of the contents of registers **rs1a**, **rs1b** and **rs2a**. This value is subtracted from the contents of register **rs2b** and the result is written to register **rd**.
- The Minimum Decode instruction (MIND) computes the Minimum of the contents of registers **rs1a**, **rs1b** and **rs2a**. This value is added to the contents of register **rs2b** and the result is written to register **rd**.
- The Maximum instruction (MAX) computes the Maximum of the contents of registers **rs1a**, **rs1b** and **rs2a** and writes the result to register **rd**.
- The Maximum Encode instruction (MAXE) computes the Maximum of the contents of registers **rs1a**, **rs1b** and **rs2a**. This value is subtracted from the contents of register **rs2b** and the result is written to register **rd**.
- The Maximum Decode instruction (MAXD) computes the Maximum of the contents of registers **rs1a**, **rs1b** and **rs2a**. This value is added to the contents of register **rs2b** and the result is written to register **rd**.

Suggested Assembly Language Syntax	
med	rs1,rs2,rd
mede	rs1,rs2,rd
medd	rs1,rs2,rd
min	rs1,rs2,rd
mine	rs1,rs2,rd
mind	rs1,rs2,rd
max	rs1,rs2,rd
maxe	rs1,rs2,rd
maxd	rs1,rs2,rd

opcode	formula
MED	rd ← Median([rs1a],[rs1b],[rs2a])
MEDE	rd ← [rs2b]-Median([rs1a],[rs1b],[rs2a])
MEDD	rd ← [rs2b]+Median([rs1a],[rs1b],[rs2a])
MIN	rd ← Min([rs1a],[rs1b],[rs2a])
MINE	rd ← [rs2b]-Min([rs1a],[rs1b],[rs2a])
MIND	rd ← [rs2b]+Min([rs1a],[rs1b],[rs2a])
MAX	rd ← Max([rs1a],[rs1b],[rs2a])
MAXE	rd ← [rs2b]-Max([rs1a],[rs1b],[rs2a])
MAXD	rd ← [rs2b]+Max([rs1a],[rs1b],[rs2a])

General description:

The Median, Minimum and Maximum operations can operate on bytes, half-words, words and double-words, both signed and unsigned. This has to be encoded into the op-code, which takes 3 bits.

Overflows:

For the MED, MIN and Max operations, overflow is not possible, as one of the inputs is simply selected, the result will always be within bounds.

For the Encoding and Decoding instructions, all adding and subtracting is done modulo 2^n, which means a wrap-around takes place in case of overflow.

Traps:

None.

Notes to the implementor:

The operations are defined for unsigned inputs. If the operations are needed for signed numbers in two's complement notation, an inversion of the sign-bit on the inputs of the comparing full-adders will ensure proper operation. With this inversion, the range of the input numbers is shifted from $\{-2^{n-1}...2^{n-1}-1\}$ to $\{0...2^n-1\}$ effectively. Because all operations are comparisons and selection (that is no arithmetic operations are performed) the right result will be selected. As the selectors operate on the unmodified inputs, the right result will appear on the output.

The connection of the opcode-bits to the implementation of the execution unit are given in the following table:

opc_0	op_0 of PMMM
opc_1	op_1 of PMMM
opc_2	$Mode$ of PMMM
opc_3	Block D to zero
opc_4	Datatype, see above
opc_5	Datatype, see above
opc_6	Datatype, see above
opc_7	0 to specify PMMM operation
opc_8	1 to specify PMMM operation

6.2 Sample implementation

In this section we describe a possible implementation of the unit by describing a general data-flow. Control logic is not discussed here, as it is dependent on various other factors, such as decoding, accessing registers, structural hazards etc. Moreover no specific unit logic implementation details, such as adders, multipliers and carry-logic etc, are presented and they are left to individual designers.

The general data-flow of the execution unit we propose is described in Figure 6.3. The unit, denoted as SPIL[1], is designed to operate on four inputs, denoted by the source registers **rs1a**, **rs1b**, **rs2a**, and **rs2b**. Control logic signals are added to determine the correct flow of data. The unit comprises a number of blocks, denoted as subunits, which are described in the following text.

The subunit PaethMMM, computing the Paeth, Median, Minimum and Maximum related operations is described in Figure 6.4. In this figure the data inputs are called A, B, C and D. These are connected to **rs1a**, **rs1b**, **rs2a** and **rs2b** respectively. The three control signals, op_1, op_2 and mode are used to control the operation of the unit. These three inputs are decoded from the opcode. All data inputs A, B, C, and D are 64 bits wide. These inputs can be split into several smaller parts, namely eight times 8 bits, four times 16 bits or two times 32 bits. For clarity, this is not shown in the figure. Furthermore, the inputs can be signed or unsigned. The unit was originally designed for unsigned inputs. A trivial adaptation makes it possible to operate on signed inputs in two's complement as well. We should note that this is not defined for the Paeth operations, because the PNG standard [95] explicitly states that all operations are carried out on bytes, which have to be interpreted as unsigned.

[1]The spil is the most important, vertical axis of a mill. We have chosen this name, because it is an important execution unit in our project, called Molen (Mill)

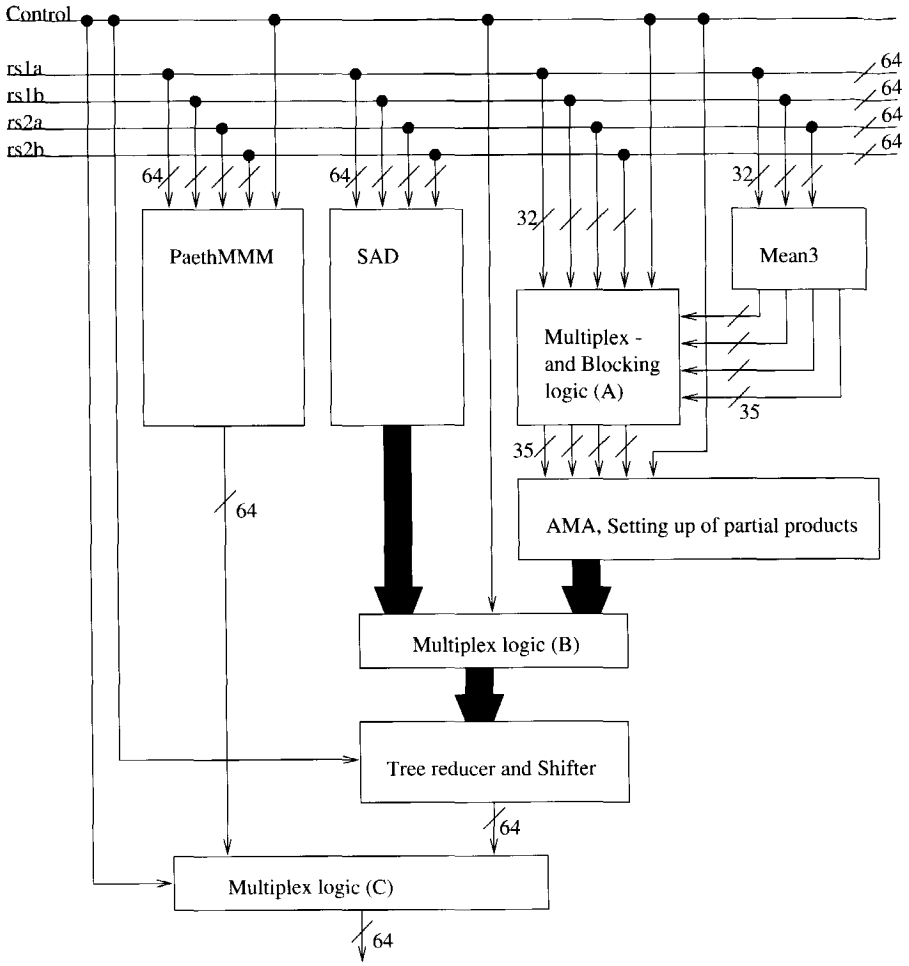


Figure 6.3: The SPIL unit.

We describe the operation for one eight bit slice of the PaethMMM unit, operating on unsigned data. In the SPIL unit, eight of these units are present and operate in parallel. In order to operate on larger data types, several signals have to be chained together.

The operation of the PaethMMM unit is as follows: Carry-generator 1 and Adders 2 and 3 are used for the comparison of inputs A, B, and C. Based on these results, $MTest\_1$, $MTest\_2$ and $MTest\_3$, the dashed block denoted as Min, Max and Median determine which of the inputs has to be selected for these operations. At the same time, Adders 2, 3 and 4 compute the intermediate

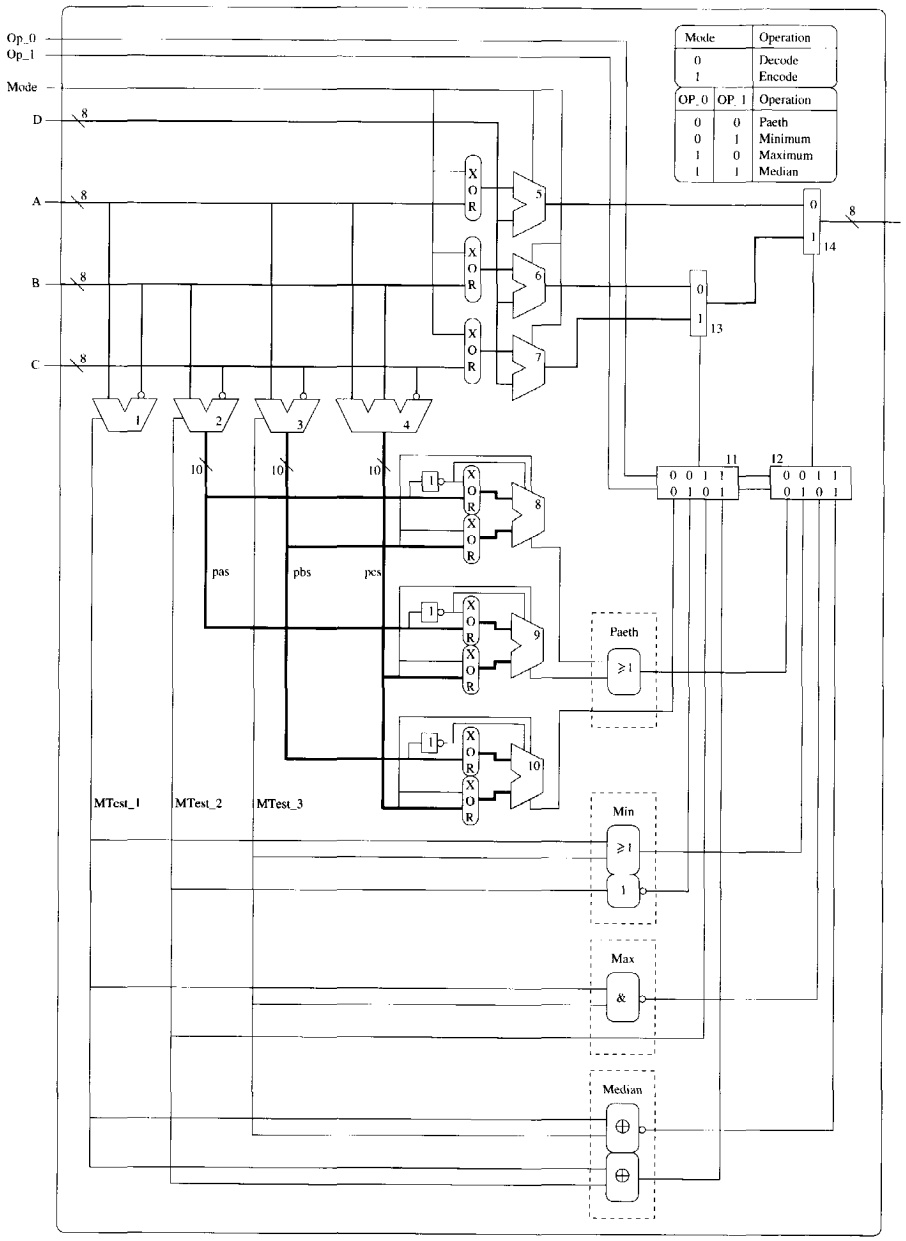


Figure 6.4: The total PaethMMM unit

results pas, pbs and pcs. These are compared in blocks 8, 9 and 10, which determine which of them has the lowest absolute value, as described in Section

4.3. This results in three similar test-signals, $Test\_1$, $Test\_2$ and $Test\_3$. The dashed block denoted as Paeth determines which of A, B, and C is to be selected as the Paeth predictor. Selectors 11 and 12 are used to determine which of the four functions is shown on the outputs. Selectors 13 and 14 select the actual value. This can be the value A, B or C if input D is blocked, or (D-A), (D-B), (D-C) in case of encoding or (D+A), (D+B), (D+C) in case of decoding. Using these precalculated values it is possible to compute the Paeth-encoded value in the same number of cycles as needed to compute the Paeth predictor. Note that these additions and subtractions are done modulo 2^8, in accordance with the PNG specification [95]. Finally we state that in case of an instruction is executed on the PaethMMM unit, the bottom multiplexer (C) of the SPIL unit (Figure 6.3) needs to select its left input.

We assume the Mean3 instructions work only on 32-bit values. In order to accommodate unsigned numbers, we will internally use 33 bits. That is we sign-extend with a zero if we operate on unsigned numbers and with the proper sign if we operate on signed numbers. As we need two extra bits in order to get the required precision, we need an Add-Multiply-Add unit which is 35 bits wide.The Mean3 part of Figure 6.3, is represented in more detail in Figure 6.5. It supplies the internal Add-Multiply-Add unit with the intermediate results of the carry-save addition of **rs1a**, **rs1b** and **rs2a**, (input A and B) and the constants $2^{35}/3$ (input C) and $2^{34} - 1$ (input D). These four inputs are used in an add-multiply-add instruction, and the result is shifted to the right over 35 bits.

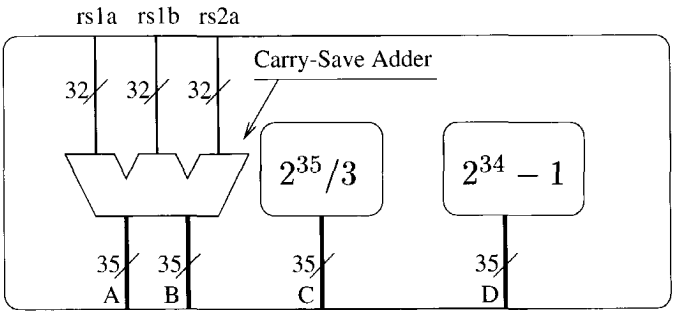


Figure 6.5: Graphical representation of the Mean.

The Add-Multiply-Add logic needed for the MEAN3 instruction is also available for general use. The 35 bit wide unit can support both signed and unsigned 32-bit words. For unsigned operation, we sign-extend the 32-bit inputs with zero's and for signed operation we sign-extend the 32-bit inputs with their own

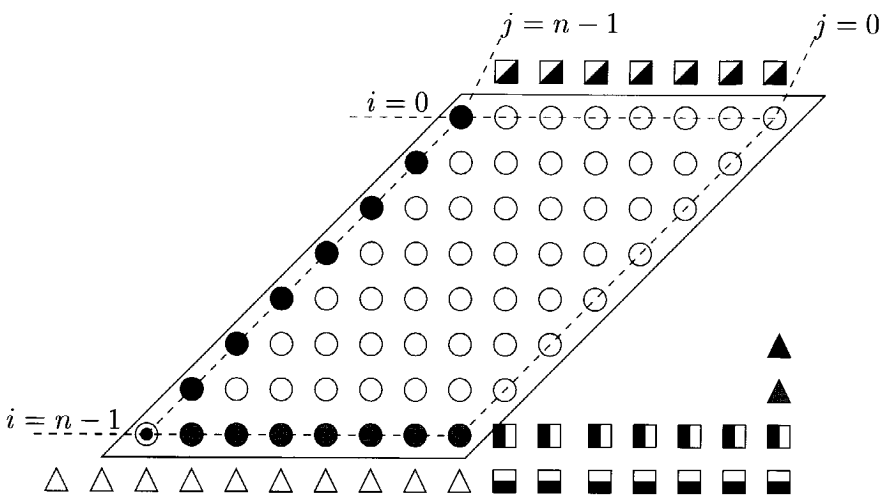


Figure 6.6: Eight-bit example of setting up the partial products for the Add-Multiply-Add unit using the Inversion Selection Technique.

sign-bit. After this sign-extension we can treat the inputs as signed, 35 bit values.

The Add-Multiply-Add subunit is represented in Figure 6.6. The inputs of the Add-Multiply-Add unit come from multiplexer (A), which chooses between the output of the Mean3 unit, depicted in Figure 6.5, or the direct inputs of the total unit. This multiplexer can also block one or more of the inputs, by which it enables the computation of several simpler expressions, such as the Add-Add instruction and Multiply-Add instruction.

The Add-Multiply-Add unit can operate in two different ways, explained in Chapter 2. The Inversion Selection Technique uses the fact that $-X$ is almost equal to $\overline{X}$. To be precise, $-X = \overline{X} + 1$. Therefore we can rewrite for instance $(A - B) * C + D$ as $(A * C) + (B * -C) + D$, which can be rewritten as $(A * C) + (B * (\overline{C} + 1)) + D == (A * C) + (B * \overline{C}) + B + D$. As a given bit of C can only be 1 or 0, and the corresponding bit of $\overline{C}$ is 0 respectively 1, we are essentially multiplexing A and B instead of adding them together. The Half-Adder Technique uses half-adders to pre-add A and B. Depending on the "per-bit" sum, which can be either -1, 0, 1, or 2, C is inserted in the product matrix as $-C$,0, C or $2C$ respectively.

The SAD unit of Figure 6.3 is depicted in more detail in Figure 6.7. This figure shows the configuration of a SAD operation on unsigned bytes as pixels. The operation takes place on 16 pixels in parallel, using 32 bytes as input. For each

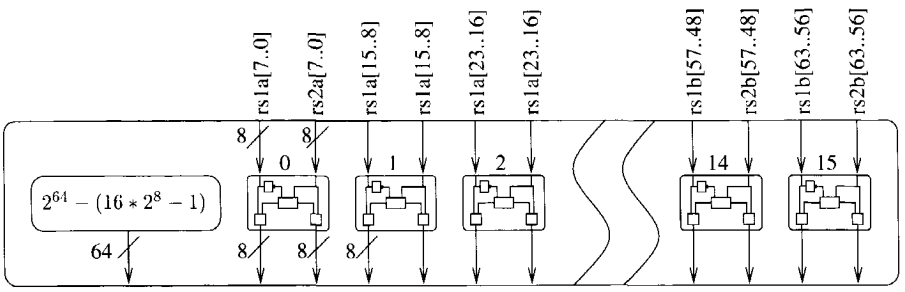


Figure 6.7: Graphical representation of the SAD computation of a 16x1 block of bytes using 16 of the blocks from Figure 3.6 in parallel. The summation of the constant and the operands is performed in the tree of Figure 6.8.

pair of input bytes, it is determined which of them is the smallest. This is done by inverting the first operand and adding the result to the second operand. The carry out of this addition determines which of them is the smallest. Then the smallest of each pair of inputs is inverted. Note that the figure shows the configuration for unsigned bytes as input. By coupling unit 0 and 1 it is possible to operate on 8 couples of half-words as well. In that case the carry-out of unit 0 is used as carry-in for unit 1. The carry-out of unit 1 is then used to determine which of the operands has to be inverted. The constant 16 should be changed to 8 in that case.

As a common ground for the requirements of both the SAD subunit and the Add-Multiply-Add subunit we note that a reduction array is needed. The Add-Multiply-Add subunit needs $n+2$ rows, which means 37 in this case. The Sum of Absolute Differences subunit needs 33 rows. The width at the top needed by the Add-Multiply-Add subunit is 34. These requirements are combined to the tree-reduction logic matrix of Figure 6.8. The implementation of such a reduction system is not pursued in detail. We only note that a number of different approaches are available, such as a Wallace tree [89], Lim counters [92], 4-2 or 6-2 counters [91] and other schemes such as those described in [93, 105, 106, 107]. In case of a Mean3 instruction a shift over 35 bits should occur after the tree-reduction. The input of the tree reduction logic is either originating from the SAD or the AMA unit. Multiplexer (B) chooses between these two inputs.

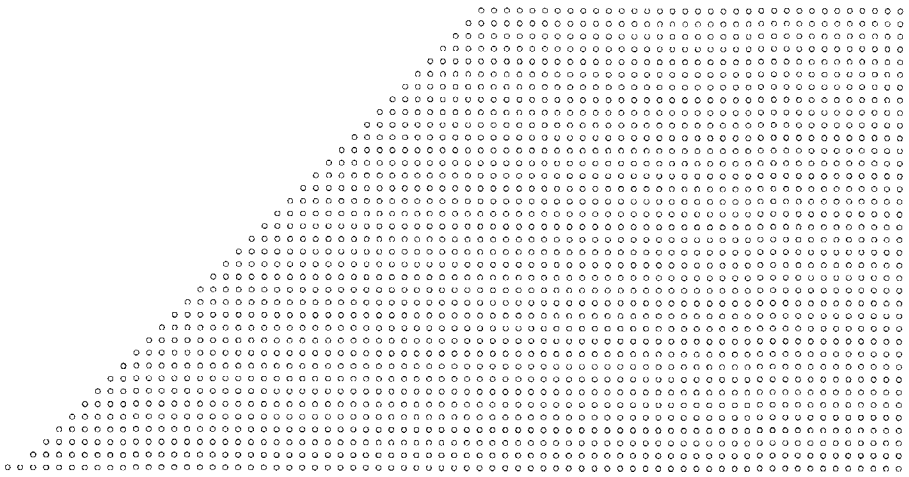


Figure 6.8: Matrix reduction requirements of the Add-Multiply-Add and the Sum-Absolute Differences unit.

6.3 Conclusions

In this chapter, we introduced the instructions and opcodes which can be executed using the hardware execution units we proposed in this dissertation. These opcodes are specified as if the hardware would be implemented as a SPARC Coprocessor. SPARC is only used as a common reference frame here, as we feel that this unit can be implemented as an add-on to all modern processors. The second part of this chapter dealt with some implementation issues. We will now continue with the final remarks and conclusions of this dissertation.

Chapter 7

Final Remarks and Conclusions

This chapter is dedicated to some final remarks regarding our investigation. It is organized in three parts. In the first part we briefly discuss the overall conclusions we have derived throughout the investigation. The second part discusses briefly the original contributions of the dissertation. The final part proposes some new research directions.

The overall conclusions and general findings of this dissertation are as follows:

- In Chapter 2 we have shown two ways of computing compound expressions of the general form $(A \pm B) * C \pm D$. We have shown that an unit capable of computing this expression can also be used to compute various simpler expressions. We have proposed two general schemes implementing this expression.

  The Inversion-Selection Technique uses the fact that $\overline{X}$ and $-X$ are "almost equal". To be precise: $-X = \overline{X} + 1$. We exploit this by multiplying A by C and $-B$ by $\overline{C}$, in which case each pair of rows in the multiplication-array can be merged into one instead of being added together.

  The Half-Adder Encoding is similar to the inversion selection technique in that it also tries to reduce the number of partial products. The half-adder technique accomplishes this by "preprocessing" $(A \; open \; B)$ in a bitwise manner, using a carry-save adder. The result of this operation is one of $\{-1, 0, 1, 2\}$, and has to be multiplied by C. This operation is trivial. The resulting partial products are added together with D (or $-D$).

  Given the way we compute the partial products, we should be able to

compute $(A \pm B) * C \pm D$ as fast as a multiplication.

- In Chapter 3 we have investigated the Sum of Absolute Differences (SAD) operation, an operation frequently used by a number of algorithms for digital motion estimation. For such an operation, we proposed a single vector instruction, which can be performed (in hardware) on an entire block of data in parallel. We investigated possible implementations for such an instruction. Assuming a machine cycle comparable to the cycle of a two cycle multiply, we showed that for a block of 16x1 or 16x16, the SAD operation can be performed in 3 or 4 machine cycles respectively. The proposed implementation operates as follows: first we determine in parallel which of the operands is the smallest in a pair of operands. Second we compute the absolute value of the difference of each pair by subtracting the smallest value from the largest one and finally we compute the accumulation. The operations associated with the second and the third step are performed in parallel resulting in a multiply (accumulate) type of operation. Our approach also covers the Mean Absolute Difference (MAD) operation, except for the shifting (division) operation.

  We are able to perform the SAD in a small amount of cycles because of the following two reasons:

  - we have substituted complex operations (i.e the subtract and absolute operation) with two simple operations (determining and inverting the smallest).
  - we have substituted the subtractions and the accumulation operation by one multi-operand addition.

  This speed advantage is especially beneficial for data-dependent algorithms, such as the three-step search algorithm. These algorithms need the SAD of the blocks in their first step to compute the addresses of the blocks in the second step.

- In Chapter 4, we presented a Paeth codec, which computes either the Paeth predictor, the Paeth-coded or the Paeth-decoded pixel with only a two cycle delay. Compared to a 21 machine instruction SPARC implementation, this is a ten-fold speedup. The unit is developed for PNG coding, but can also be useful in other graphic schemes. The hardware requirements for the unit are modest, in the order of nine 10-bit adders.

  We compute the Paeth predictor using the following steps:

- Direct computation of the distance of each input to the initial estimate.
- Compare these distances using Carry-generators.
- Select the input with the lowest distance.

The codec variant computes the difference or the sum of the current pixel and the values to be encoded/decoded in parallel to the first step, it does not alter the last two steps. The final step remains the selection of the right output.

- In Chapter 5 we introduced some instructions, which can be implemented with small modifications of the hardware required for the instructions proposed in the previous chapters. More specifically we proposed four new instructions for the median, the maximum, the minimum and the mean. All instructions operate on three input numbers and produce a single output value. For the proposed instructions we showed hardware implementations, which indicate the following:

  - No complex modifications are required to implement the instructions using available hardware.
  - No cycle time penalties should be introduced, (except perhaps for mean) if the additional instructions proposed in this chapter are implemented.

  In particular it has been shown that one cycle is required to perform the median, min and max. The mean may require 2 or 3 cycles (the third cycle could be added to avoid critical path delay problems) depending on the technology used to implement the units. Furthermore it has been shown that there are substantial advantages in using our proposal when compared to software solutions. More specifically we have shown that performing the median in hardware requires 1 cycle, while performing the same operation in software using an usual instruction set (e.g. Sun SPARC) would require 6 to 9 cycles. For min and max, which are also 1 cycle operations, the software solution will require 7 cycles. Finally, the mean of three numbers as proposed will require 2 to 3 cycles which will be substantial less than using a commonly available instruction set which requires two addition operations and a division.

- In Chapter 6, we introduced the instructions and opcodes which can be executed using the hardware execution units we proposed in the previous

chapters. These opcodes are specified as if the hardware would be implemented as a SPARC Coprocessor. SPARC is only used as a common reference frame here, because we feel that this unit can be implemented as an add-on to all modern processors. The second part of this chapter dealt with some implementation issues.

7.1 Contributions

The main original contributions of this dissertation are as follows:

- We have shown that a number of true data dependencies, which have been identified to be present in large percentages in embedded system applications as shown by [74], can be captured by an unique expression: $(A \pm B) * C \pm D$. Consequently, assuming two's complement representation, we propose two schemes for the implementation of such an expression. Both schemes require no more machine cycles than the number of machine cycles needed for the multiplication of two numbers.

- For a very frequent motion estimation operation, the Sum of Absolute Differences (SAD), we proposed a vector instruction and we investigated possible implementations for such an instruction. Assuming a machine cycle which is comparable to the cycle of a two cycle multiply, we have shown that for a block of 16x1 or 16x16, the SAD operation can be performed in 3 or 4 machine cycles respectively.

- We proposed a hardwired solution for the paeth predictor used in the Portable Networks Graphics standard. Moreover we proposed a hardware Paeth codec capable of computing three different quantities: the Paeth predictor of three inputs, the difference between the current pixel and the Paeth predictor of the other inputs (Coding), and the sum of the coded input and the Paeth predictor of the other three inputs (Decoding). These computations are performed within two cycles, where a cycle is comparable to a general purpose ALU cycle. Depending on the mode of operation, the proposed mechanism produces the predictor or the (de/en)-coded pixel value.

- We have shown that without additional cycle time penalties an extension of the Paeth unit is possible by which it can additionally compute the Median of three inputs [100]. This median is used in video-deinterlacing, which is needed when displaying normal video on a non-

interlaced computer screen or a modern, high-end television set. We have also shown that the median operation can be computed by itself in one machine cycle.

- We have introduced a number of new instructions and shown that these instructions can be implemented with trivial additions to the multiply (or multiply-add) hardware together with other well known instructions, for example multiply, multiply-add, multiply-subtract. We have also shown that the remaining instructions can be executed by a new execution unit, which is no more complex than a traditional ALU design.
- As a minor contribution we have shown that using the Paeth extended logic, it is trivial to compute also the maximum and minimum of the three inputs. Furthermore, we introduce an extension of the Add-Multiply-Add unit, described in Chapter 2 whereby the Add-Multiply-Add unit can compute the Mean of three inputs.

7.2 Future research directions

There are a number of open questions left out from our investigation. In the final section of this dissertation we mention some possible ways to continue our research. In our opinion, research could be continued in the following areas.

- Although we have proposed several hardwired execution units, we did not perform an "exhaustive" search to find all possible operations which could improve the performance if implemented in hardware. While obviously such "exhaustive" search is not quite possible, finding new functional requirements is a desired research direction because it may expand the units we propose with some trivial additions.
- Our proposal can be considered only the beginning of an architectural proposal. We did not perform an overall estimation of performance gains in multimedia benchmarks. While our approach is the first step to propose an architecture, performance evaluations should additionally be carried out to further justify the instruction set. We believe this is another important research direction to be followed as a continuation of our research.
- Our investigation has been especially tuned to the execution stages of the pipeline. No special considerations have been paid to the memory

interfacing. Due to the nature of applications and the substantial amount of data movement, special attention should be paid to such a data movement instruction set. Clearly this is another important direction which future investigations should follow.

- On the design aspects we did provide strong evidence of meeting the cycle time requirements. We believe that certain units such as the Paeth unit could have even better cycle times then general purpose ALU like execution stages. We have not addressed such possibility, as the "silicon" design was not the objective of this dissertation. We believe that producing an implementation in silicon of this unit is also another direction to be followed in the near future.

Bibliography

[1] A. Pagegs, B. Moore, R. Smith, and W. Buchholz. The IBM System/370 vector architecture: Design considerations. *IEEE Transactions on Computers*, 37(5):509–520, May 1988.

[2] P. M. Kogge. *The Architecture of Pipelined Computers*. Advanced computer science series. McGraw-Hill Book Company, 1981.

[3] J. L. Hennesy and D. A. Patterson. *Computer Architecture, a Quantitative Approach*. Morgan Kaufmann Publishers Inc., 1990.

[4] SGS Thomson Microelectronics. MPEG audio/MPEG-2 Video intergrated decoder STi3520. http://www.st.com, October 1996.

[5] SGS Thomson Microelectronics. MPEG 2.5 Layer III audio decoder STA013. http://www.st.com, September 1999.

[6] A. van Roermund, P. Snijder, H. Dijkstra, C. Hemeryck, C. Huizer, J. Schmitz, and R. Sluijter. A General-Purpose Programmable Video Signal Processor. *IEEE Transaction on Consumer Electronics*, 1989.

[7] J. Goto, K. Ando, T. Inoue, M. Yamashina, H. Yamada, and T. Enomoto. 250-Mhz BiCMOS Super-High-Speed Video Signal Processor (S-VSP) ULSI. *IEEE Journal of Solid-State Circuits*, 1991.

[8] T. Minami, R. Kasai, H. Yamauchi, Y. Tashiro, J. ichi Takahashi, and S. Date. A 300-MOPS Video Signal Processor with a Parallel Architecture. *IEEE Journal of Solid-State Circuits*, 1991.

[9] H. Yamauchi, Y. Tashiro, T. Minami, and Y. Suzuki. Architecture and Implementation of a Highly Parallel Single-Chip Video DSP. *IEEE Transactions on Circuits and Systems for Video Technology*, 1992.

[10] K. Aono, M. Toyokura, T. Araki, A. Ohtani, H. Kodama, and K. Okamoto. A Video Digital Signal Processor with a Vector-Pipeline Architecture. *IEEE Journal of Solid-State Circuits*, 1992.

[11] P. A. Ruetz, P. Tong, D. Bailey, D. A. Luthi, and P. H. Ang. A High-Performance Full-Motion Video Compression Chip Set. *IEEE Transactions on Circuits and Systems for Video Technology*, 1992.

[12] D. Bailey, M. Cressa, J. Fandrianto, D. Neubauer, H. K. Rainnie, and C.-S. Wang. Programmable Vision Processor/Controller. *IEEE Micro*, 12(5):33–39, October 1992.

[13] H. Fujiwara, M. L. Liou, M.-T. Sun, K.-M. Yang, M. Matuyama, K. Shomura, and K. Ohyama. An All-ASIC Implementation of a Low Bit-Rate Video Codec. *IEEE Transactions on Circuits and Systems for Video Technology*, 1992.

[14] K. Gaedke, H. Jeschke, and P. Pirsch. A VLSI Based MIMD Architecture of a Multiprocessor System for Real-Time Video Processing Applications. *Journal of VLSI Signal Processing*, 5:159–169, 1993.

[15] K. Herrmann, M. Seifert, K. Gaedke, H. Jeschke, and P. Pirsch. *Architecture and VLSI implementation of a RISC Core for a Monolithic Video signal Processor.*, pages 368–377. VLSI Signal Processing. IEEE, New York, 1994.

[16] S. Rathnam and G. Slavenburg. An Architectural Overview of the Programmable Multimedia Processor, TM-1. In *Proceedings of COMPCON '96*, pages 319–326. IEEE, 1996.

[17] K. Guttag, R. J. Gove, and J. R. van Aken. A single-chip multiprocessor for multimedia: The MVP. *IEEE Computer Graphics and Applications*, November 1992.

[18] A. Peleg and U. Weiser. MMX Technology Extension to the Intel Architecture. *IEEE Micro*, 16(4):42–50, August 1996.

[19] A. Peleg, S. Wilkie, and U. Weiser. Intel MMX for multimedia PCs. *Communications of the ACM*, 40(1):24–38, Jan 1997.

[20] *Alpha Architecture Handbook.* Digital Equipment Corporation, 1996.

[21] R. L. Sites and R. Witek. *Alpha AXP Architecture: Reference manual 2ed.* Digital Press, 1995.

[22] P. Rubinfeld, B. Rose, and M. McCallig. Motion Video Instruction Extensions for Alpha. http://www.digital.com/semiconductor/papers/pmvi-abstract.htm, October 1996.

[23] M. Tremblay, J. M. O'Conner, V. Narayanan, and L. He. Vis speeds new media processing. *IEEE Micro*, 16(4):10–20, August 1996.

[24] *Visual Instruction Set (VIS) User's Guide.* Sun Microsystems, 1997.

[25] The Visual Instruction Set (VIStm): On-chip Support for New-Media Processing. http://www.sun.com/sparc/whitepapers/wp95-028, 1995.

[26] Mips digital media extension. http://www.sgi.com/MIPS/products/.

[27] K. Diefendorff. Pentium III = Pentium II + SSE, Internet SSE Architecture boosts multimedia performance. *Microprocessor Report*, 13(3):5–11, March 1999.

[28] S. Vassiliadis, E. Hakkennes, J. Wong, and G. Pechanek. The Sum-Absolute-Difference Motion Estimation Accelerator. In *Proceedings of the 24th EUROMICRO Conference*, volume II, pages 559–566. IEEE Computer Society, IEEE, August 1998.

[29] S. D. Cotofana. *Addition Related Arithmetic Operations with Threshold Logic.* PhD thesis, Delft University of Technology, 1997.

[30] S. Vassiliadis, D. S. Lemon, and M. Putrino. S/370 Sign-magnitude Floating-point Adder. *IEEE J. Solid-State Circuits*, 24(4):1062–1070, Aug. 1989.

[31] J. Sklansky. Conditional-sum Addition Logic. *IEEE Trans. Electron. Comput.*, EC-9:226–231, 1960.

[32] H. Ling. High-Speed Binary Adder. *IBM J. Res. Develop.*, 25(3):156–166, May 1981.

[33] G. Bewick, P. J. Song, G. D. Micheli, and M. J. Flynn. Approaching a Nanosecond: a 32 Bit Adder. In *IEEE ICCD 1988*, pages 221–226, Oct. 1988.

[34] H. Srinivas and K. Parhi. A Fast VLSI Adder Architecture. *IEEE Journal of Solid State Circuits*, 27(5):761–767, May 1992.

[35] S. Vassiliadis. A Comparison Between Adders with New Defined Carries and Traditional Schemes for Addition. *Int. J. of Electronics*, 64(4):617–626, Apr. 1988.

[36] S. Vassiliadis. Recursive Equations for Hardwired Binary Adders. *Int. J. of Electronics*, 67(2):201–213, Aug 1989.

[37] C.J.Chang, S. Vassiliadis, and J. Delgado-Frias. An Investigation of Binary CLA and Ripple CMOS Adder Designs. *Microprocessing and Microprogramming*, 40(1):1–21, January 1994.

[38] N. T. Quach and M. J. Flynn. High Speed Addition in CMOS. Technical Report CSL-TR-90-415, Stanford Univ., 1990.

[39] S. Waser and M. J. Flynn. *Introduction to arithmetic for digital systems*. CBS College Publishing, 1982.

[40] K. Hwang. *Computer Arithmetic: Principles, Architecture and Design*. John Wiley & Sons, New York, 1979.

[41] S. Vassiliadis, E. M. Schwarz, and B. M. Sung. Hard-Wired Multipliers with Encoded Partial Products. *IEEE Transactions on Computers*, 40(11):1181–1197, November 1991.

[42] V. Oklobdzija and D. Villeger. Improving Multiplier Design by Using Improved Column Compression Tree and Optimized Final Adder in CMOS Technology. *IEEE Transactions on VLSI Systems*, 3(2):292–301, June 1995.

[43] V. Oklobdzija, D. Villeger, and S. Liu. A Method for Speed Optimized Partial Product Reduction and Generation of Fast Parallel Multipliers Using an Algorithmic Approach. *IEEE Transactions on Computers*, March 1996.

[44] O. MacSorley. High-Speed Arithmetic in Binary Computers. *Proceedings IRE,* Vol. 49, pp.67-91, January 1961.

[45] L. P. Rubinfield. A Proof of the Modified Booth's Algorithm for Multiplication. *IEEE Transactions on Computers,* Vol. C-24, pp.1014-1015, October 1975.

[46] L. Dadda. Squarers for Binary Numbers in Serial Form. In *Proceedings of the 7^{th} IEEE Symposium on Computer Arithmetic*, pages 173–180, 1985.

[47] J. E. Robertson. A New Class of Digital Division Methods. *IEEE Transactions on Computers*, C-7:218–222, Sept. 1958.

[48] N. R. Scott. *Computer number systems and arithmetic*. Prentice Hall, Englewood Cliffs, New Jersey, 1985.

[49] I. Koren. *Computer Arithmetic Algorithms*. Prentice Hall, 1993.

[50] S. D. Pezaris. A 40-ns 17-bit by 17-bit Array Multiplier. *IEEE Transactions on Computers*, C-20:442–447, Apr. 1971.

[51] D. M. Mandelbaum. A Systematic Method for Division with High Average Bit Skipping. *IEEE Transactions on Computers*, 39(1):127–130, Jan. 1990.

[52] R. Stefanelli. A Suggestion for a High-Speed Parallel Binary Divider. *IEEE Transactions on Computers*, C-21(1):42–55, Jan. 1972.

[53] H. Srinivas and K. Parhi. A Fast Radix-4 Division Algorithm. *IEEE Transactions on Computers*, 44(6):826–831, June 1995.

[54] L. Dadda. On Serial-Input Multipliers for Two's-Complement Numbers. *IEEE Transactions on Computers*, C-38(9):1341–1345, September 1989.

[55] M. Ercegovac and T. Lang. *On-Line Arithmetic: A Design Methodology and Applications*, volume VLSI Signal Processing, III, chapter 24. IEEE Press, Los Angeles, 1988.

[56] J. Muller. On-line Computing: A Survey and Some New Results. In P. Quinton and Y. Robert, editors, *Algorithms and Parallel VLSI Architectures II*, pages 261–272. Elsevier, 1992.

[57] P. Ienne and M. Viredaz. Bit-Serial Multipliers and Squarers. *IEEE Transactions on Computers*, C-43(12):1445–1450, December 1994.

[58] L. Dadda. Some Schemes for Serial Input Multipliers. In *Proceedings of the* 6^{th} *IEEE Symposium on Computer Arithmetic*, pages 52–59, June 1983.

[59] M. J. Irwin and R. M. Owens. Digit-Pipelined Arithmetic as Illustrated by the Paste-Up System: A Tutorial. *IEEE Computer*, (1):61–73, 1987.

[60] H. J. Sips. Bit-Sequential Arithmetic for Parallel Processors. *IEEE Transactions on Computers*, C-33:7–20, January 1984. (also reprinted in Tutorial on Advanced Computer Architecture, ed. D.P. Agrawal, IEEE press, 1986, pp. 93-106).

[61] H. J. Sips. Comments on $O(n)$ Parallel Multiplier with Bit-Sequential Input and Output. *IEEE Transactions on Computers*, C-31:325–327, April 1982.

[62] I. Chen and R. Willoner. An O(n) Parallel Multiplier with Bit Sequential Input and Output. *IEEE Transactions on Computers*, C-28:721–727, October 1979.

[63] L. Dadda. Fast Multipliers for Two's-Complement Numbers in Serial Form. In *Proceedings of the* 7^{th} *IEEE Symposium on Computer Arithmetic*, pages 57–63, 1985.

[64] R. Gnanasekaran. On a Bit-Serial Input and Bit-Serial Output Multiplier. *IEEE Transactions on Computers*, C-32(9):878–880, September 1983.

[65] T. Rhyne and N. Strader. A Signed Bit-Sequential Multiplier. *IEEE Transactions on Computers*, C-35(10):896–901, October 1986.

[66] M. J. Irwin and R. M. Owens. A Case for Digit Serial VLSI Signal Processing. *Journal of VLSI Signal Processing*, (1):321–334, 1990.

[67] V. Oklobdzija and M. Ercegovac. An On-Line Square Root Algorithm. *IEEE Transactions on Computers*, C-31(1):70–75, January 1982.

[68] J. E. Phillips. *High Performance Execution Engines for Instruction Level Parallel Processors*. PhD thesis, Delft University of Technology, 1996.

[69] E. Hokenek and R. Montoye. Leading-zero anticipator (LZA) in the IBM RISC System/6000 floating point execution unit. *IBM Journal of Research and Development*, 34(1):71–77, January 1990.

[70] R. Montoye, E. Hokenek, and S. Runyon. Design of the IBM RISC System/6000 floating-point execution unit. *IBM Journal of Research and Development*, 34(1):59–70, January 1990.

[71] J. Phillips and S. Vassiliadis. Result equal zero predictor for 3-1 interlock collapsing ALUs. *International Journal of Electronics*, 75(3):379–392, April 1993.

[72] J. Phillips and S. Vassiliadis. High-Performance 3-1 Interlock Collapsing ALU's. *IEEE Transactions on Computers*, pages 257–268, March 1994.

[73] S. Vassiliadis, J. Phillips, and B. Blaner. Interlock Collapsing ALU's. *IEEE Transactions on Computers*, 42(11):825–839, July 1993.

[74] F. Onion, A. Nicolau, and N. Dutt. Compiler feedback in ASIP design. Technical Report 94-2, Department of Information and Computer Science, University of California, September 1994.

[75] C. Baugh and B. Wooley. A Two's Complement Parallel Array Multiplication Algorithm. *IEEE Transactions on Computers*, C-22:1045–1027, december 1973.

[76] L. Dadda. Composite Parallel Counters. *IEEE Transactions on Computers*, C-29(10):942–946, October 1980.

[77] J. L. Mitchell, W. B. Pennebaker, C. E. Fogg, and D. J. LeGall. *MPEG Video Compression Standard.* Digital Multimedia Standard Series. Chapman and Hall, 1996.

[78] B. Furht, J. Greenberg, and R. Westwater. *Motion Estimation Algorithms for Video Compression.* Kluwer Academic Publishers, 1997.

[79] S. Vassiliadis, E. M. Schwarz, and D. J. Hanrahan. A general proof of overlapped multiple-bit scanning multiplications. *IEEE Transactions on Computers*, 38(2):172–183, February 1989.

[80] D. L. Gall. Mpeg: A video compression standard for multimedia applications. *Communications of the ACM*, 34(4):46–58, April 1991.

[81] N. Ahmed, T. Natarajan, and K. Rao. Discrete Cosine Transform. *IEEE Transactions on Computers*, pages 90–93, jan 1974.

[82] H. Kitjima. A symmetrric Cosine Transform. *IEEE Transactions on Computers*, c-29(4):317–323, april 1980.

[83] B. T. an W.C. Miller. On Computing the Discrete Cosine Transform. *IEEE Transactions on Computers*, c-27(10):966–968, Oct 1978.

[84] A. N. Netravali and B. G. Haskell. *Digital Pictures; Representation, Compression, and Standards*. Plenum Press, 1994.

[85] S. Kappagantula and K. Rao. Motion compensated predictive coding. In *Proc. Int. Tech. Symp. SPIE*, San Diego, CA, August 1983.

[86] T. Koga, K. Linuma, A. Hirano, Y.Iijima, and T. Ishiguro. Motion-compensated interframe coding for video conferencing. In *NTC 81 Proc.*, pages G5.3.1–5, New Orleans, LA, December 1981.

[87] J. R. Jain and A. K. Jain. Displacement measurement and its applications in interframe image coding. *IEEE Transactions on Communications*, COM-29(12):1799–1808, December 1981.

[88] R. Srinivasan and K. Rao. Predictive coding based on motion estimation. *IEEE Transactions on Communicatios*, COM-33:1011–1014, September 1985.

[89] C. Wallace. A Suggestion for Parallel Multipliers. *IEEE Trans. Electron. Comput.*, EC-13:14–17, 1964.

[90] L. Dadda. Some schemes for parallel multipliers. *Alta Frequenza*, 34:349–356, May 1965.

[91] S. Vassiliadis, J. Hoekstra, and H.-T. Chiu. Array Multiplication scheme using (p,2) counters and pre-addition. *Electronics Letters*, 31(8):619–620, April 1995.

[92] R. Lim. High-speed multiplication and multiple summand addition. In *Proc. IEEE 4thSymp. Com. Arithmetic*, pages 149–153, October 1978.

[93] P. Song and G. De Michelli. Circuits and Architecture trade-offs for high-speed multiplication. *IEEE Journal of Solid-State Circuits*, SC-26(9):1184–1198, 1991.

[94] A. W. Paeth. Image file compression made easy. In J. Arvo, editor, *Graphic GEMS II*. Academic Press Inc., 1991.

[95] T. Boutell and T. Lane. PNG (Portable Network Graphics) Specification, version 1.0. ftp://ftp.uu.net/graphics/png/documents/png-1.0-w3c.ps.gz.

[96] C. W. Brown and B. J. Shepherd. *Graphic File Formats; reference and guide*. Manning Publications Co., 1995.

[97] P. Deutch, J.-L. Gailly, and M. Adler. *GZip*. http://www.gzip.org.

[98] G. Roelofs. http://www.cdrom.com/pub/png/pnghist.html. Here is stated that Office 97 is using PNG as the native image format.

[99] G. Roelofs. News and History of the PNG Development Group. http://www.cdrom.com/pub/png/pngnews.html.

[100] T. Doyle and P. Frenken. Median filtering of television images. In *Proceedings of the International Conference on Consumer Electronics, Digest of Technical Papers*, pages 186–187, June 1986.

[101] A. Riemens, R. Schutten, and K. Vissers. High speed video deinterlacing with a programmable TriMedia VLIW core. In *Proceedings of the International Conference on Signal Processing applications and Technology, ICSPAT'97*, pages 1375–1380, September 1997.

[102] E. Hakkennes, S. Vassiliadis, and S. Cotofana. Fast computation of compound expressions in two's complement notation. In A. Sydow, editor, *15th Imacs World Congress on Scientific Computation, Modelling and Applied Mathematics*, volume 4, Artificial Intelligence and Computer Science, pages 689–694, Berlin, August 1997. Wissenschaft & Technik Verlag.

[103] E. Hakkennes and S. Vassiliadis. Hardwired Paeth Codec for Portable Network Graphics (PNG). In *Proceedings of the 25th EUROMICRO Conference*, volume II, pages 318–325. IEEE Computer Society, IEEE, September 1999.

[104] SPARC International, Menlo Park, California. *The SPARC Architecture Manual, Version 8*, 1992.

[105] M. Santoro and M. Horowitz. A pipelined 64x64 b iterative array multiplier. *ISSCC Dig. Tech. Papers*, 1988.

[106] D. Shen and A. Weinberger. 4-2 carry-save adder implemtation using send circuits. *IBM Tech. Disc. Bull.*, 20:3594–3597, Feb. 1987.

[107] A. Weinberger. 4-2 carry-save adder module. *IBM Tech. Disc. Bull.*, 23, 1981.

[108] G. de Haan, P. W. Biezen, H. Huijgen, and O. A. Ojo. True Motion Estimatiuon with 3-D Recursive Block Matching. *IEEE Transactions*

on Circuits and Systems for Video Technology, 3(5):368–379, october 1993.

[109] B. Liu and A. Zaccarin. New Fast Algorithms of the Estimation of Block Motion Vectors. *IEEE Transactions on Circuits and Systems for Video Technology*, 3(2):148–157, apr 1993.

Samenvatting

In dit proefschrift onderzoeken we de mogelijkheden om een hardware implementatie te maken van units die van samengestelde expressies uitrekenen. De focus ligt hierbij op samengestelde bewerkingen die veel voorkomen in multimedia applicaties. De eerste groep van samengestelde expressies die we bekijken kan worden beschreven als Optel-Vermenigvuldig-Optel (Add-Multiply-Add) expressie. Voor deze familie van expressies laten we zien dat een hardware unit gebouwd kan worden die ongeveer dezelfde logische structuur heeft als de unit voor de normale vermenigvuldig instructie. Dit impliceert dat de Optel-Vermenigvuldig-Optel instructie net zo snel is als een gewone vermenigvuldiging, wat een significante versnelling kan opleveren in multimedia applicaties. Verder laten we zien dat twee veelgebruikte operaties video compressie, de Som van Absolute Verschillen en het Gemiddelde van Absolute Verschillen, in hardware geïmplementeerd kunnen worden, waarbij het resultaat in ongeveer dezelfde tijd kan worden uitgerekend als een vermenigvuldiging. Onze aanpak kan eenvoudig worden uitgebreid tot de berekening van de Som van Absolute Verschillen van een 16 bij 16 pixels groot blok in niet meer dan twee maal de tijd benodigd voor een vermenigvuldiging. Verder stellen we een Coder-Decoder structuur voor in hardware, die de Paeth predictor van de Portable Network Graphics (PNG) standaard berekend. We laten zien dat ook deze unit niet meer tijd nodig heeft als een vermenigvuldiging, en we breiden deze unit tevens uit met de operaties mediaan, minimum en maximum. Verder breiden we de Optel-Vermenigvuldig-Optel unit nog iets uit zodat deze in staat is om het gemiddelde van drie inputs te berekenen. Tot slot stellen we twee ontwerpen voor, die samengesteld zijn uit bovenbeschreven onderdelen. De eerste unit voorziet in alle operaties die aan vermenigvuldigen gerelateerd zijn, de tweede is voor alle operaties van de Paeth, mediaan, minimum en maximum groep. Beide units worden gecombineerd tot één unit, die 32 verschillende instructies kan uitvoeren.

Curriculum Vitae

Edwin Hakkennes was born in The Hague on December 18^{th}, 1970. After his secondary education at the "Christelijk Lyceum" in Gouda, he studied at the Electrical Engineering Department of Delft University of Technology, where he graduated "Cum Laude" in September 1995. His masters thesis was in the area of pattern recognition on images which are scanned on a hexagonal raster. The prototype resulting from this research was implemented in an FPGA and partly in a semi-custom chip design. During this study, he was awarded the Siemens Nederland Stipendium, a six month stay in Berlin, where he studied 3 months at the Technical University of Berlin, and also did three months of practical work at the Surface Mounting Laboratory of Siemens.

From September 1992 until September 1995, he was appointed supervisor on the laboratory courses on digital techniques for first and third years students. He continued to supervise these courses when he was given a Ph.D. position in the group of professor Stamatis Vassiliadis. This position started on November 15, 1995, and ends with the presentation of this dissertation. In 1997 and 1998, Edwin was also appointed as "specialist" for the computer logic design course, where sixteen second year students design a complete chip in 10 weeks.

Appendix A

Motion Estimation

In this appendix we give some additional background information on motion estimation, and we describe some common algorithms used for motion estimation. There are two basic kinds of motion estimation, namely true motion estimation, which is used for scan-rate conversion, and block-based motion estimation for motion compensation which is used in video compression algorithms. In the second kind of motion estimation, the algorithms are optimized to find the best match of a given block with respect to a given reference frame. Motion estimation for scan rate conversion is used to calculate intermediate frames, which were never received. For each pixel in this intermediate frame, a value is computed using the (per pixel) motion vector of the previous frame to the next frame. Basically, the motion between two received frames is divided in two equal steps and the intermediate result is inserted as a new frame.

In the following two sections, we will show some of the algorithms used for the two kinds of motion estimation. The emphasis is on block-based motion estimation, which is in the scope of this dissertation.

A.1 True Motion Estimation

True motion estimation is used in scan-rate conversion, video-deinterlacing, and surveillance video systems. One of the algorithms used is 3-D Recursive Search Block Matching [108]. This is used in scan-rate-conversion for 100Hz televisions, which display frequency is 100Hz, where the transmission is only 50Hz interlaced. Two estimators are used for the current vector, which both produce 4 candidate-vectors. The candidate-vector with a minimal Sum of Ab-

solute Differences (SAD) associated is selected. The estimators that are used are the blocks top-left and top-right from the current block. These estimation-directions are perpendicular, at 45 and -45 degrees, thereby trying/ensuring that convergence is reached in at least one direction.

Substantial attention is paid to the computation of "nice guesses" for the motion vector. With these guesses, the motion vector is computed. It should be noted that knowledge from other blocks in the current frame, and from other blocks in the past frames is used to determine the motion. Smoothness in the motion-plane is one of their main goals, there are "penalties" for discontinuities in the velocity-plane.

The "criteria" used for this algorithm are: Modified Mean Square Error (M2SE), (spatial) Smoothness of the vector field and Operation Count. In all three criteria, their 3D-Recursive Search algorithm performs well to very well. In other words, it is very suitable for block-matching in applications such as consumer field-rate conversion.

A.2 Block-Based Motion Estimation

In this section we discuss block-based motion estimation. As an example we show the simplest, but most compute intensive motion estimation algorithm in some detail. We show that it can be divided into layers, which we also encounter in the other algorithms. The simplest motion estimation algorithm is exhaustive search. This is the only algorithm which guarantees to find the global minimum within the bounds of the search range. It is therefore often used as a standard against which other algorithms are compared. For this discussion, we use macro-blocks of 16 by 16 pixels and a search range of $\pm$ 7 pixels. The exhaustive search operates in one single step. The control is therefore data-independent, which makes it a suitable algorithm for implementation in hardware. As candidate motion vectors, all $15^2 = 225$ possible motion vectors are used. For each candidate motion vector, the SAD over the current block and a block of the reference frame is computed. The block of the reference frame has coordinates of the block in the current added with the motion vector.

The SAD for motion vector (a, b) for a current block which has top-left coordinates at (x, y) is defined as:

$$SAD(x, y, a, b) = \sum_{i=0}^{15} \sum_{j=0}^{15} |A_{(x+i,y+j)} - B_{(x+a+i,y+b+j)}| \qquad \text{(A.1)}$$

where A is the current frame and B the reference frame. For each candidate motion vector, the SAD is computed. This takes 256 subtract absolute operations and 255 additions. If we neglect the absolute operation, this is 511 operations. However, the test for negative is probably the most expensive operation, as it introduces data-dependency into the control of the program. For each block, 225 candidate motion vectors have to be evaluated, making a total of 225*511=114975 operations per block. For each frame, consisting of 720*480 pixels, 1350 blocks have to be evaluated. This makes the number of operations 1.552e+08. For one second of video at 25 Hz, it would require 3.880e+09 operations.

Most of the algorithms described below try to reduce this number by reducing the number of candidate vectors. We introduce the different layers in motion estimation in some more detail below.

One second of video: One second of video consists of 25 frames. From the top down, we see one second of video. This is our (arbitrary) starting point. The frames can be coded using the scheme of Figure A.1. Note that this is not standardized and therefore each encoder can choose its own strategy.

This scheme repeats after 12 frames. Based upon the position of the frame (in time), each frame is put into one of three categories:

I These are the Intra-coded frames. No motion estimation is used in the compression of these frames. These frames are needed to accommodate scene changes and they are used as anchor points for the motion estimation.

P The Predictive-coded frames are predicted with respect to the closest I frame.

B The Bidirectionally Predictive-coded frames are predicted with respect to either the preceding or following P or I frame. These frames benefit the most from Motion Estimation and Compensation.

Each frame is then processed according to the category in which it is put.

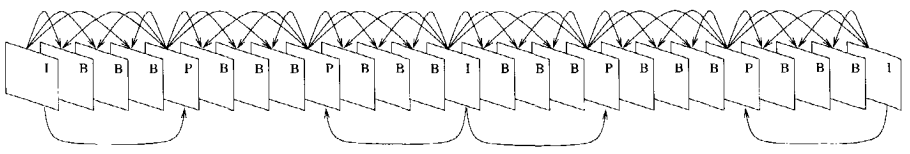


Figure A.1: 25 of frames of video, subdivided into I, B and P frames.

Note that this is a layer of the encoding scheme, and not a layer of motion estimation. We have included it to put things in perspective.

The frame layer: We have one or two reference frames and one current frame. Now that we have determined for each frame how it should be coded, we can subdivide the frame in basic blocks of 16 by 16 pixels. In this layer decimation can take place, in that different algorithms are used for different blocks. In general, all blocks are treated the same. However, the sub-sampled motion field estimation algorithm [109] uses a checkerboard approach, where for all black blocks the motion vector is computed using a standard algorithm, and for all white blocks only the motion vectors of the four neighbors are evaluated. Figure A.2 gives a graphical representation of this decimation. In the figure the motion vectors of the black blocks (depicted gray for clarity) are computed using an arbitrary scheme, e.g. full-search. The resulting motion vectors are indicated with arrows in each block. For the white blocks, only the motion vectors of the neighboring black blocks are candidate motion vectors. This means only four motion vectors need to be evaluated.

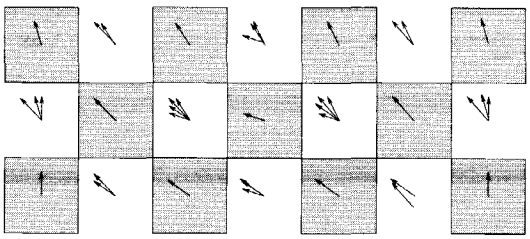


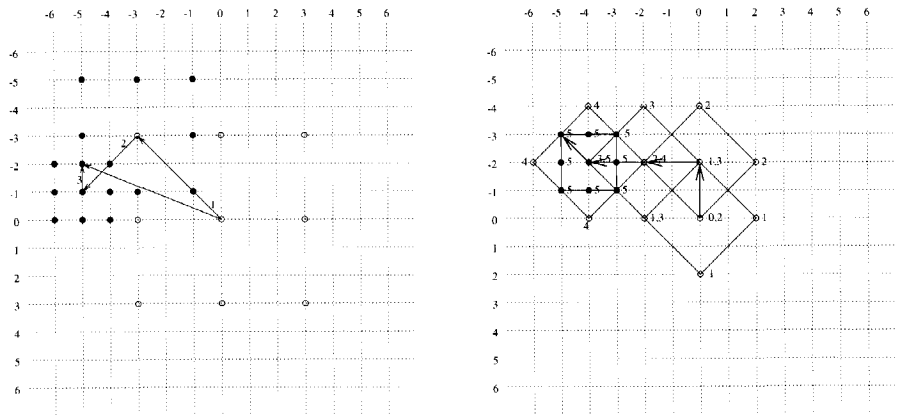
Figure A.2: Sub-sampled Motion-Field Estimation.

Other approaches to reduce the cost in this layer include the telescopic search algorithm, which uses the motion vectors obtained for the previous frame as guideline for the search, see [78].

The candidate motion vector layer: We have one block in the current frame and one or two reference frames. This is the layer where most motion estimation algorithms try to eliminate operations. The exhaustive search evaluates all possible 15^2 motion vectors and chooses the vector resulting in the lowest SAD. Other algorithms try to do some sort of binary search to minimize the criterion. In the first step, the center and four to eight other motion vectors on a search step distance from the center are evaluated (where the search step starts at half the search range). The motion vector resulting in the lowest SAD is chosen as the center for the next step, where the search step is halved. Once the search step equals one, the algorithm terminates and the motion vector

resulting in the lowest SAD is chosen.

Two common examples are the Three Step Search (Figure A.3(a)) and the 2-D Logarithmic Search (Figure A.3(b)), which is described in more detail in Chapter 3.



(a) Three Step Search

(b) 2 Dimensional Logarithmic Search

Figure A.3: Graphical representation of two common algorithms.

The decimation layer: In this layer we have a current block and a candidate motion vector, which refers to a block in a reference frame. The decimation layer is not always present. It tries to minimize computations by selecting only a fraction of the pixels of each frame for inclusion in the SAD criterion. The fraction is typically $\frac{1}{4}$, that is a decimation takes half of the pixels in both horizontal and vertical direction. Special care has to be taken to use suitable decimation schemes that do not introduce artifacts.

The criterion layer: In this layer we have a number of pixels from both the current frame and the reference frame. In the criterion layer, a metric for resemblance is computed over the pixels. Most algorithms specify that a given metric should be used, but it is really a separate issue. A commonly used metric is the Sum Absolute Difference (SAD), which is defined above (Formula A.1). A derived metric is the Mean Absolute Difference, which is equal to the SAD divided by the number of pixels in a block. This number is almost always a power of 2, so the division is a simple shift operation.

One of the more complex metrics is the Mean Square Error, which is defined

as:

$$MSE(x,y,a,b) = \frac{1}{256} * \sum_{i=0}^{i=15} \sum_{j=0}^{j=15} (A_{(x+i,y+j)} - B_{(x+a+i,y+b+j)})^2 \quad \text{(A.2)}$$

for a block on position (x, y) and a motion vector (a, b).

As the objective of the criterion is to estimate the achievable compression rate using a certain motion vector, a theoretical criterion would be the number of bits needed to be transmitted for a given motion vector. In normal circumstances this is computational prohibitive, as the 2D-Dicrete Cosine Transform (DCT) would have to be computed for each candidate motion vector.

After the Motion Estimation: After a motion vector has been found for a certain macro block, a motion compensated block is computed by simply subtracting the block where the motion vector points to from the current block. This is also done for the two chrominance blocks, where the sub-sampling has to be taken into account. This motion compensated block is then subdivided into 4 basic blocks. From these blocks, the 2D-DCT is computed, which is (after Variable Length coding) transmitted to the receiver together with the motion vector.