MASTER

The evaluation of the architecture synthesis tool Phideo

van de Braak, H.J.

Award date:
1996

Eindhoven University of Technology

Department of Electrical Engineering
Information- and Communication Systems

Master Thesis:

The evaluation of the Architecture Synthesis Tool Phideo

H.J. van de Braak

Supervision PCALE:	Ir. J.A.A.M. van den Hurk, Ir. P.H. Frencken
Supervision:	Prof. Dr. Ing. J.A.G. Jess
Period:	sept.1994 - may 1995
Place:	Digital Video Processing group, Department PCALE, Philips Semiconductors

The Department of Electrical Engineering of the Eindhoven University of Technology is not responsible for the contents of thesis reports

Master Thesis

The evaluation of the Architecture Synthesis Tool Phideo

H.J. van de Braak

Abstract

This report describes the evaluation of the architecture synthesis tool Phideo. Phideo is developed by Philips and is used to design high throughput Digital Signal Processing applications. Typical properties of these applications are repetitions, dedicated hardware which is often pipelined and large communication requirements. The synthesis and allocation of distributed memories play an important role in Phideo.

Phideo design method is based on the analysis of the manual design process and the different design decision that are taken. Phideo is not a fully automated push-button system, but important design decisions are left to the designer. The input of Phideo is written at high level in Phideo Input Language (PIF). Phideo generates a design and provides the necessary feedback to evaluate and improve the design iteratively by defining constraints which drive the scheduling process and the memory synthesis. The final output of Phideo is a synthesizable Register Transfer Level VHDL description.

As a test-case for Phideo a part of an MPEG2 decoder is used. The research goal and conclusions are based on design time, quality of the design and applicability in a product development environment.

Key Words

PHIDEO, Architecture synthesis, High Level synthesis, DSP,
MPEG-2, IDCT, IQ, IZZS

:

Summary

To cope with the increasing complexity and shorter design times in IC-design, it is necessary to use a well structured design strategy. In the digital video processing group at the Product Concept and Application Laboratory of Philips in Eindhoven a hierarchical design flow has been developed during the past years. This report describes the evaluation of the architecture synthesis tool Phideo (version 2.0), which automates part of the design flow. Phideo is a tool which is application driven and is used in the design of high throughput Digital Signal Processing applications. With Phideo, the design time of digital circuits can be decreased, while the achievable quality of designs is compared to manual design.

The Phideo design method is based on the analysis of the manual design process and the different design decisions that are taken. Phideo is not a fully automated push-button system. User interaction is of major importance. Phideo automates large part of the design actions, but important design decisions are left to the designer. This way Phideo combines the advantages of design automation with the qualities of the human designer.

The input of Phideo is a high level behavioural description in the Phideo Input Format together with constraints to steer the synthesis process. Phideo generates an implementation of the design together with the necessary feedback, such as schematic and estimated resource requirements, to evaluate the design and to improve the design iteratively. The final output of Phideo is a Register Transfer Level VHDL description which can be synthesised by existing RT-Level synthesis tools.

The architecture synthesis tool Phideo has been evaluated with the design of parts of an MPEG-2 video decoder. MPEG-2 is a video coding standard which is likely to replace the current analog standards in order to enhance the number of transmitted television programs and to enable new services. Some parts of the MPEG-2 decoder with the highest computation complexity were chosen to be implemented by Phideo, i.e. the Inverse Discrete Cosine Transform the Inverse Scan and the Inverse Quantisation.

In this report a number of different video decoder architectures are presented in order to reduce the required off-chip memory. These architectures imposes constraints on the throughput of the IDCT the IS and the IQ. After an extensive study of the Inverse Discrete Cosine Transform, implementations for the different video decoder architectures are made with Phideo. The resulting designs are compared to similar designs which are constructed manually.

It can be concluded that using Phideo, the quality of the designs in terms of chip-area is comparable to the quality of manual designs. In contrast, the design time can be decreased significantly. As a result the designer is able to evaluate different alternatives which results in a better exploration of the design space. Disadvantages are the input language of Phideo which cannot be executed and is not compatible with VHDL, the scarcely available manuals and documentations. Furthermore it can be very hard to define the right pragmas to get Phideo to do what the designer desires. This canSometimes because Phideo does not find a suitable solution, sometimes because the designer wants something which is not possible.

:

:

Foreword

This Master Thesis reflects the result of the final research of the course Information Technology at the Technical University Eindhoven (TUE). The research was performed from September 1994 till May 1995 at PCALE (Product Concept and Application Laboratory Eindhoven), a departement of Philips Semiconductors.

This project was carried out in cooperation with the Philips Research Laboratory Eindhoven (Nat.Lab.). Since the goal of the project was to evaluate the architecture synthesis tool Phideo which is being developed there, I spend most of the time at the Nat.Lab.

I hereby would like to thank my supervisors Joris van den Hurk and Peter Frenken of PCALE and Albert van der Werf of the Nat.Lab and especially my professor Prof.Dr.Ing. J.A.G. Jess of the Design Automation Section for all their support and advise. Furthermore I would like to thank Paul Lippens and Wim Verhaegh for their overall support. Last but not least I want to express my gratitude to my love and supporting girlfriend Mieke van der Heijden for her patience and understanding during the past year.

During my research I encountered a relevant anecdote. It is called "The banana becomes open to question". It goes as follows: One takes a cage with apes. At the top of the cage a banana is hung up, underneath a stair is placed. After a while one of the apes tries to climb the stair, but as soon as he puts his feet on it, all the apes are squired with water. Shortly thereafter another or the same ape tries to climb the stairs again, with the same result: all apes wet. From now on, if an ape tries to climb the stairs the others will prevent that. Now we remove one ape from the cage and replace it by a new ape. This one sees the banana and tries to climb the stairs. At his fright all apes threaten him. After another attempt he knows: if he tries to climb the stairs he will get a beating. Then we replace a second ape. The newcomer goes to the stairs and gets a beating. The last newcomer participates enthusiastically. A third goes out, a third comes in, goes to the stairs and gets a beating. Two of the apes that threaten him have no idea why it is not allowed to climb the stairs. Old ape out new ape in, until all apes who ever got wet are replaced. Nevertheless no ape tries to climb the stairs. "Why not, sir?" "We simply don't do that over here, young man."

Though experience is substantial and indispensable, it could yield a narrow-minded way and prejudiced way of thinking. I believe that self-learning and trail and error are the best ways to gain experience. Philips offered a pleasent environment conduct the research for my master thesis, providing the necessary liberty to do the research. When assistance or a second opinion was needed there was always someone willing to lent a helping hand.

:

Contents

Chapter 1

Introduction

In the past decades the art of integrated circuit (IC) design has evolved rapidly. In the early days ICs consisted of a small number of transistors which had to be designed by drawing the layout of the silicon layers. Nowadays the complexity and the number of transistors on a single IC has increased explosively. With millions of transistors on a single IC and growing complexity of the system to be designed, it has become impossible to design such an IC without automation tools. On top of that, the design time has decreased. Due to growing competition and shorter life times of products the time-to-market has become an important issue in the design of an IC. To meet all these design requirements, design automation is necessary in all stages of the design path.

In this report a new high-level design tool is studied and evaluated. As a test-case for this tool, the design of an MPEG-2 decoder is chosen which is used in a Digital Video Broadcasting system. In the next sections a brief introduction is given on Digital Video Broadcasting followed by a brief introduction of MPEG-2 in the succeeding section. Next we describe the design steps of the development of an IC and possibilities to automate this design. Since part of the study was to evaluated the new tool and determine whether or not it fits the approach of PCALE (a department of Philips Semiconductors where the research was done), the design flow at PCALE is described. The succeeding sections concern the architecture synthesis tool Phideo which is the subject of the research. After a brief introduction on Phideo, the research goals are given and explained. At the end of this introduction chapter an overview of the structure of this report is given.

1.1 Digital Video Broadcasting

The number of television programmes has increased significantly during the last decade. All these programmes have to be transmitted on the existing channels. This becomes a problem since the available bandwidth on the current cable, satelite and terrestrial networks is becoming too small to transmit all the programmes, containing video, audio and data (e.g. teletext). With the analog transmission standards every programme occupies one channel with a bandwidth of 6 MHz. With digital compression techniques it is possible to put between 4 and 10 programmes (video, audio and data) on the same channel. The number of programmes that can be transmitted on a single channel depends on the quality of the pictures and sound and the amount of data. With digital television standards it is therefore possible to increase the number of programmes to be broadcasted without expensive modifications and extensions of the transmission channels, like cable networks and satellite links.

Besides the increasing number of programmes and the reduced costs, digital television has more advantages. The picture quality of the programme can be varied between low resolution and high definition, depending on the requirement. The different programmes can be multiplexed easily, which makes a better bandwidth utilization possible. Sound quality can also be varied and multilingual sound and multilingual subtitling is possible. Digital data can easily be encrypted, which opens possibilities for video on demand and pay per view. Interactive television is also feasible and there are many other possibilities such as, home shopping, video games etc.

In figure 1.1 a model of a Digital Video Broadcasting system is depicted. The systems consists of two parts: the source coding part and the channel coding part.

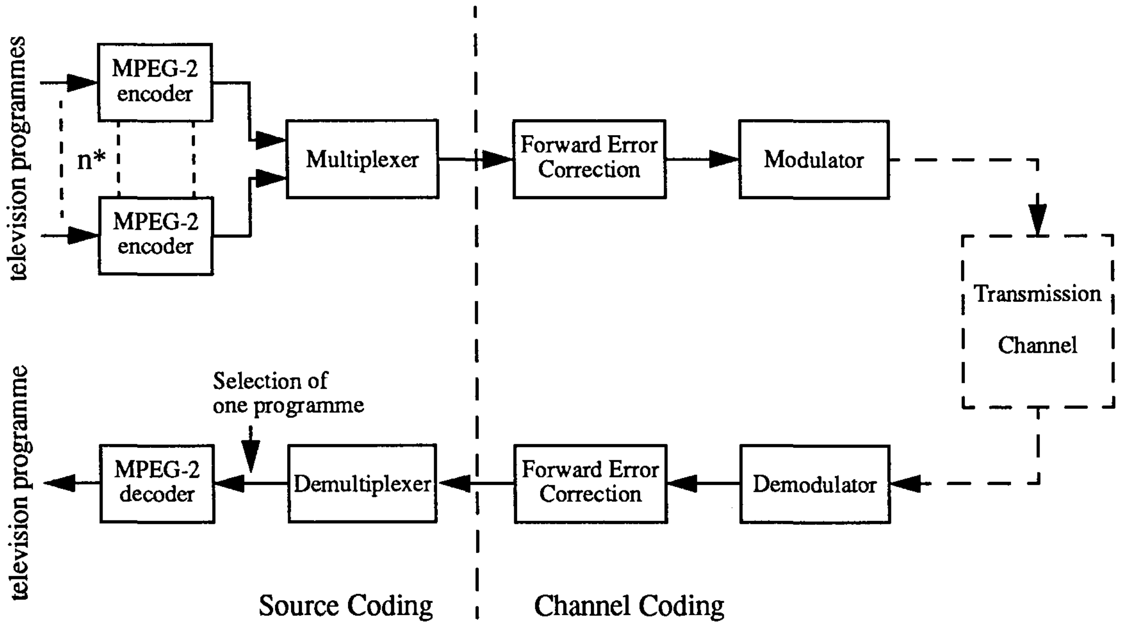


Figure 1.1 Model of a Digital Video Broadcasting system

The source image is successively coded by a source encoder followed by a channel encoder before it is transmitted. The task of the source encoder is to reduces the number of bits of the source message using the characteristics of the specific source. In case of source encoding television programmes (digital video information and associated audio) the MPEG-2 algorithm can be used which is discussed in the next section. After multiplexing different programmes in one bitstream, it is coded by the channel encoder.

The task of the channel encoder is to tune the message to the channel characteristics and to protect the message against transmission errors introduced by the channel. There are three common channels to transmit digital video information: cable, satellite and terrestrial. For satellite communication, a Forward Error Correction (FEC) can be used which consists of two stages: first a Reed-Solomon algorithm followed by a Viterbi algorithm. These two stages are necessary because of the low signal to noise ratio in satellite transmission. The resulting signal can be modulated using Quaternary Phase Shift Keying (QPSK). For terrestrial and cable transmission a FEC can be used which only consists of a Reed-Solomon algorithm. The modulation techniques which can be used are among others Orthogonal Frequency Division Multiplexing (OFDM) for terrestrial and 64-Quaternary Amplitude Modulation (64-QAM) for cable transmission.

At the receiver side the same process is carried out in reversed order. After demodulation and error correction the bitstream is demultiplexed and the selected programme is decoded.

1.2 A brief introduction to MPEG

MPEG is a standard of the International Standards Organization (ISO) for coded representation of moving pictures, associated audio, and their combination. MPEG is short for Moving Pictures Expert Group and is named after the expert group who developed it.

The MPEG standard is developed for the compression and decompression of moving pictures (video) and sound (audio) and the formation of a multiplexed common data stream that includes the compressed video and audio data plus any associated service data. Furthermore the MPEG standard provides means for synchronization of the video, audio, and service data during the playback of the decompressed signals. It is intended to serve a wide variety of applications and services such as digital storage media and television.

The first MPEG standard (MPEG-1) was capable of compressing SIF video (352x288 at a framerate of 30Hz) and compact disc audio at a combined coded bitrate of 1.5 Mbit/s, approximated the perceptual quality of consumer video tape (VCR). A second standard (MPEG-2) aimed at efficient coding of broadcast quality video. The major difference between MPEG-1 and MPEG-2 are the higher picture quality if desired and the representation of interlaced video signals.

The MPEG-2 standard is an extremely flexible one. It is intended to be generic in the sense that it serves a wide range of applications, bitrates, resolutions, qualities and services. The MPEG-2 standard provides a set of defined compression and systemization algorithms and techniques all combined in a single syntax. MPEG-2 defines a number of subsets of the syntax and a set of constraints on the parameters in the bitstream by means of profile and level. For example main level at main profile provides compression of Standard Definition TeleVision (SDTV) with coded bitrates up to 15 Mbit/s and High level at main profile is used for High Definition TeleVison (HDTV) with coded bitrates up to 80 Mbit/s.

More about the MPEG-2 standard and the applied compression techniques can be found in Chapter 3 on page 33.

1.3 Design automation

Due to the rapid technological development of the last decades, designers have come to rely on automatic or semiautomatic CAD systems for the design of complex ICs containing over a million transistors. In the early days, design automation was mainly concerned with design verification (circuit and logic simulation, layout verification), later on also design automation tools became feasible that allow for the specification or description of a system at various levels of abstraction and for the automated or even automatic implementation of a design. The transformation of a description or specification from one level of abstraction into another is called synthesis. This can done be either by hand or automated with the use of sythesis tools.

The first synthesis tools were designed for a full automatic placement and routing of a gate-level design description. The next step was the introduction of logic-level synthesis and optimization techniques allowing for automatic translation of truth tables into minimized networks of logic gates. Nowadays a lot of commercial software for these synthesis steps are available. However, the forces of time-to-market and growing system complexity demand the use of high-level, automated methods and tools. At this moment a lot of effort is put into synthesis methods, tools, and systems that transform a high-level design description, or even specification, into an adequate implementation. Moving up to higher levels of abstraction results in a further reduction of design cost and time-to-market. Another advantage is the reduction of design errors, since the design flow becomes highly automated which should yield less errors. Also working at higher levels of abstraction is less prone to error than dealing with the little details of lower levels. More about synthesis approach to digital system design can be found in [1].

The goal of design automation for electronic systems is to automate the transformation of a specification given at the highest level of abstraction into a description at the lowest level, e.g. the mask geometry which provides the interface to fabrication. A software system that can provide this transformation is called a silicon compiler. In figure 1.2 an overview of the subtasks in a silicon complil-er is shown. The nomenclature of the various level of description are conform “The synthesis approach to digital system designs” by P.Michel [1]..

At the highest level of abstraction, the *system level*, the system is specified by its functionality and a set of constraints to be met (e.g. speed, power consumption, fabrication cost). The description is mostly on paper.

The first synthesis step is *System-Level synthesis*. The result is a partitioning of the system into subsystems, and the synthesis of the behavioural description for each of these subsystems. The behavioural description (or algorithmic description) at the *Algorithmic Level* defines a precise procedure for the computational solution of a problem. It specifies behaviour in terms of operations and computation sequences on inputs to produce the required output. Basic elements of the description are similar to those of programming languages, like arithmetic and logic operations applied to variables, and control structures such as loops, and procedure calls. In this synthesis step the first optimization takes place. By altering the algorithm the number of operations and computation complexity can be reduced. Currently, both partitioning and specification of subsystems are performed manually.

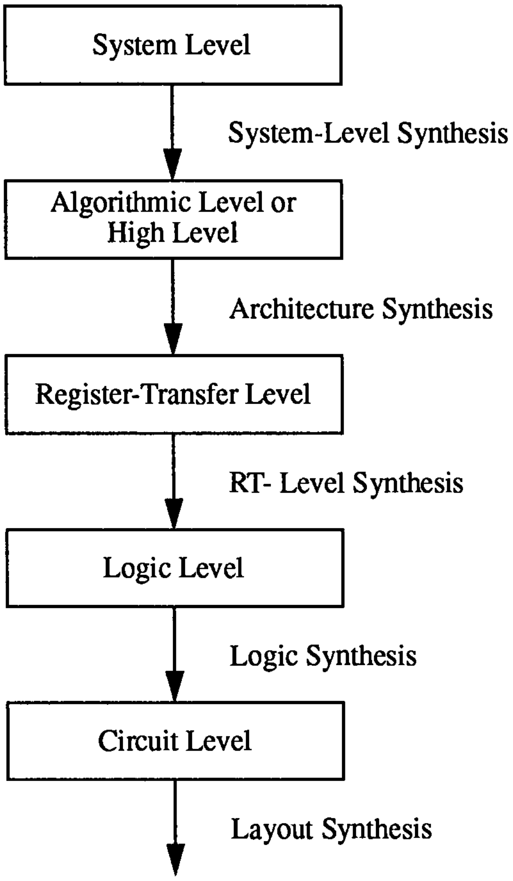


Figure 1.2 Design flow of a Silicon Compiler

A behavioural description (or algorithmic description) at the *Algorithmic Level* provides the starting point for architecture synthesis. In *architecture synthesis* (also called high-level or algorithmic-level synthesis), three different subtasks can be distinguished:

- resource allocation: functional units of appropriate type and number have to be selected
- scheduling: operations have to be assigned to time slots
- resource assignment: operations need to be assigned to specific functional units.

At this level a time/area trade-off (allocating more resources allows for more parallel execution of operations, giving higher performance at higher hardware cost) is be made.

The result of high-level synthesis is an initial description at the *Register Transfer Level* (RTL) of a data path and a controller. An RT-Level description is a system definition in terms of registers, multiplexers, and operations. In this description an initial assignment of operations to clock cycles has been made. The synthesis of the controller and the translation of the system into states and state transitions are typical tasks of the *Register-Transfer-Level synthesis*.

The result is a design at *Logic-Level* specified in terms of blocks of combinational logic and storage elements. The optimization and mapping of these block onto a gate-level hardware structure is the task of Logic Level synthesis (also called *Library or Technology Mapping*). Four subtasks can be distinguished in logic synthesis: state encoding, logic optimization, technology mapping and gate sizing. State encoding assigns binary codes to the states to minimize the final implementation of the state machine. In logic optimization the combinational circuit is optimized. During this optimization a trade-off between speed, area and power of the resulting circuit is made.Technology mapping maps the optimized circuit to matching physical library cells of a given target technology. In order to speed-up the circuit, gates on the critical path in the circuit can be sized. Finally in the *layout synthesis* these cells and their interconnect are placed and routed.

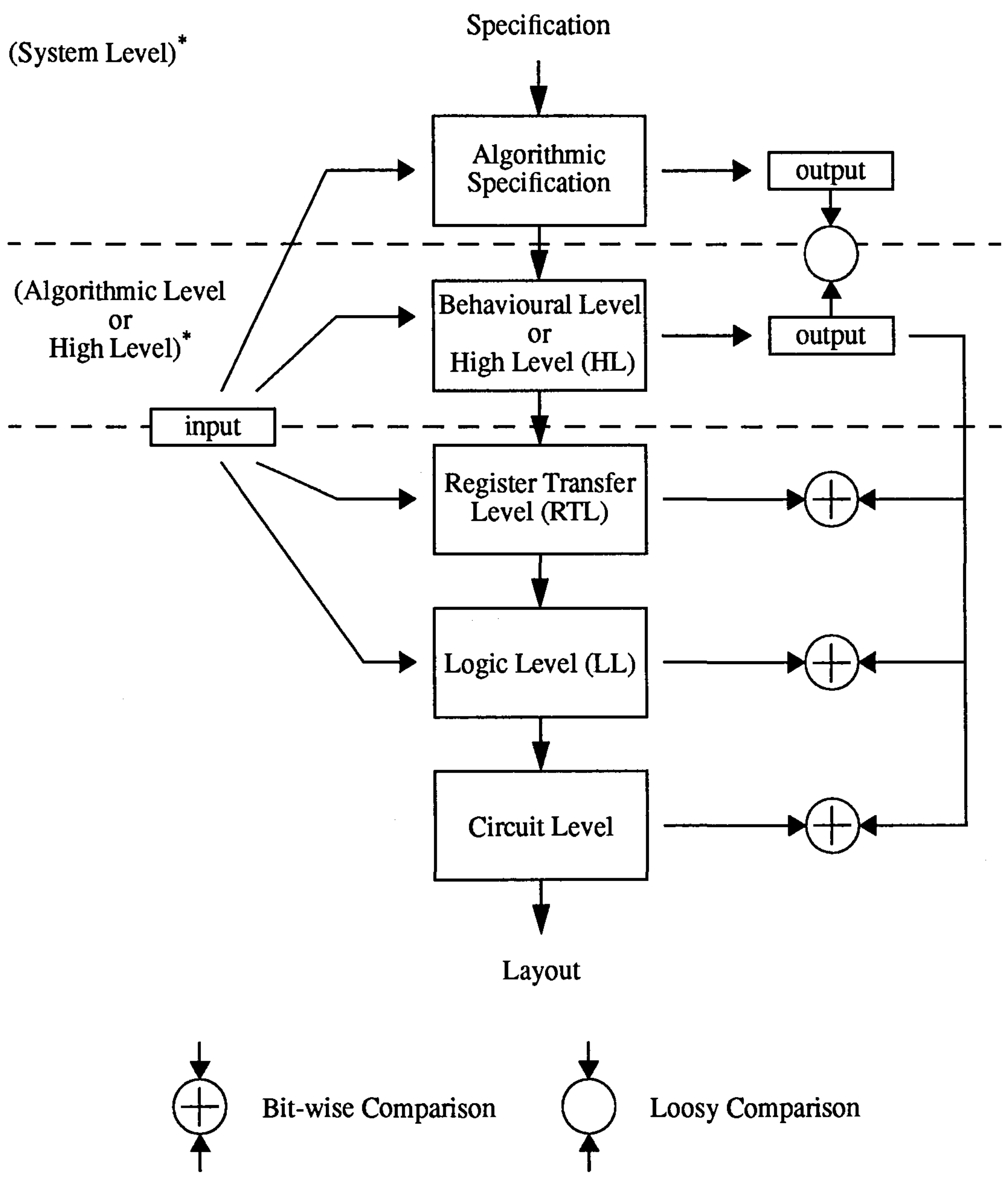
1.4 PCALE design flow

At Philips Concept and Application Laboratory Eindhoven (PCALE) a design flow is developed for the design of digital integrated circuits [2]. The PCALE design flow is top-down design flow, and is characterized by a number of description levels (see figure 1.3). The highest description levels are an abstract specification of the system to be developed. It only describes the functionality of the designs in a unambiguous manner and is independent of the implementation which is chosen. The highest level is used to verify each level in the system on it's functional correctness. Starting with the highest level of description, each level adds more details, implementation aspects and or timing aspects. Each time the correctness of the description is verified against the previous level. Until at the lowest level a layout is obtained which is transferred to the foundry where the first prototypes of the IC are manufactured.

All the levels are described in VHDL (Very high speed integrated circuit Hardware Description Language). It is a high level description language for system and circuit design. The language supports a wide range of description styles. These include structural descriptions, dataflow descriptions and behavioural descriptions. The structural and dataflow descriptions show a concurrent behaviour. That is, all statements are executed concurrently, and the order of the statements is not relevant. On the other hand behavioural descriptions are executed sequentially in processes, procedures and functions, which resemble high-level programming languages. All descriptions in VHDL can be simulated, which is indispensable for verification purposes.

When using the appropriate style (e.g. limit the number of different processes and limit the use of signals) the simulation speed of high-level VHDL is not significant slower than other high-level programming languages like C. In highest levels of description, all implementation details are omitted and therefore the execution and simulation speeds of these levels can be very high. This yields the greatest advantage of the PCALE design flow. With these high simulation speeds, extensive functional testing of the complete design is possible and functional errors or bottlenecks can be recognized in an early stage. At lower levels in the design the simulation speeds are much lower and the designs contain much more detail. At this level simulation of the complete design of complex systems is very slow and therefore restricted to a small number of tests. At this stage it takes a much more effort to track down functional errors.

The advantage of the use of one language for each level of design is that it is possible to test parts of the design from different levels of description at the same time. This way it easy to compare a subblock from a lower level of description with the same block at a higher level and verify it by com-


Specification
(System Level)*
Algorithmic Specification
output
(Algorithmic Level or High Level)*
Behavioural Level or High Level (HL)
output
input
Register Transfer Level (RTL)
Logic Level (LL)
Circuit Level
Layout
Bit-wise Comparison
Loosy Comparison

* The classifications of the levels as used by [1] and described in the previous section.

Figure 1.3 Verification at the various description levels.

paring their behaviour. Another advantage is that it is possible to connect a subblock of the design which is described at a low level of abstraction to the rest of the design which is described at a higher level. As a result only part of the system is at a low level, so the simulation remains high. This way a subblock from a lower level can be tested in the complete system instead of stand alone only.

In the next sections the description style of the various levels in the PCALE design flow are further explained, the characteristics of each level is summarized in table 1.1.

The design flow starts with a *specification* which is mostly on paper. Since a description in english language is ambiguous and not executable, the implications for the resulting design are not very clear. Therefore the design is first transformed into a formal description, an *Algorithmic Specification*. The advantage of an algorithm is that it is unambiguous, and executable. This way, only one interpretation is possible and thus no confusion. Since an algorithm is executable the results of certain requirement and constraints can be evaluated. If not satisfactory the algorithm can be adjusted or refined. In the Algorithmic Specification abstract data types like reals and integers are allowed, implementation details are omitted were possible.

At the *Behavioural Level* the design is divided into subsystems. The behaviour of each subsystems is described by means of input and output relation. At this level also a notion of time is introduced. The communication between the subsystems is *clock cycle true* regarding the inputs and outputs, i.e. all the inputs and outputs occur exactly at the same clock cycle as they will occur in the target design. This way a functional verification of the complete system of the design is possible. Since the intern behaviour of the subsystems is not yet related to the clock, very high simulation can be achieved. This way extensive tests and verifications can be performed which reduces the probability of functional errors.

Note the difference between this Behavioural Level and the High Level of the previous sectionsection (figure 1.2)! In this Behavioural Level a notion of time is introduced in contradiction to the High Level of the previous section. Another important difference is the internal of the subsystems. Whereas the High Level of the previous section specifies the precise procedure for the computation of the outputs, the Behavioural Level only specifies the outputs.

Since abstract data types cannot be represented in hardware without rounding errors, only bit-composite types (e.g. bit-vectors and boolean types) are used at this level. At this point the effects of finite precision calculations and finite word length calculations can be evaluated, and the required word length can be derived. The verification of the behavioural algorithm is done by means of a loosely comparison of the output results of the Behavioural Algorithm and the Algorithmic Specification on a a specific input sequence (see figure 1.1). The behavioural algorithm is correct if all differences can be ascribed only to the finite precision calculation and round off errors.

At the behavioural level all the inputs and outputs of the subsystems and the communication between the subsystems are exact the same as they are in the target design, this is called *bit-true* regarding its inputs and outputs. The behavioural level is used as a reference for the lower levels in the design flow. These levels can be verified by means of a bit-by-bit comparison to the behaviour of this level.

Note that the nomenclature of the levels of the PCALE design flow are confusing in relation to the these of figure 1.1 in the previous subsection. The Specification as well as the Algorithmic Specification fall under the System Level of the previous subsection which is illustrated in figure 1.3. In Hurk [2] this level was originally called the Algorithm Level, in this report it is referred to as Algorithmic Specification to prevent mistakes. The Behavioural Level resembles the Algorithmic Level or High Level of figure 1.2. The other levels are conform the nomenclature of figure 1.2.

At the *Register Transfer Level* the interior of the subsystems are designed. For each subblock a implementation is written in, using registers, functional units, and multiplexers. Functional units of appropriate type and number have to be selected. and the operations have to be assigned to time-slots and to specific units. The implementation of a subblock is bit-wise compared to the behavioural description. At this moment this step is done manually, which is very time consuming. Automation of this step results in a great reduction of design time and effort and therefore a significant faster time-to-market. Architecture synthesis tools like Phideo (described in the next section) and Mistral2 automate parts of this level. They translate a high-level (HL) description or behavioural description into a Register Transfer Level description while providing the necessary user interaction to control this process. This report describes the study of the applicability of the architecture synthesis tool Phideo in the design of complex ICs. A similar study of the architecture synthesis tool Mistral-2 is performed, which is described in [3].

The translation from a descriptions at the Register Transfer Level to Logic Level and further down to layout is fully automated. A lot of commercial synthesis tools like Synergy (Cadence), Synopsis or Compass are available. At PCALE Synergy is used for the translation of RT-Level VHDL to gate Level or Library Level. To verify the results of this step the resulting RT-Level VHDL is compared to the Algorithmic Specification. Since already extensive testing is done on the functional correctness of the system the tests can be focused on the subblocks. At this point the Design Flow of PCALE ends and the RT-Level description is transferred to a Production Centre where the translation into layout is done and finally the prototype of the IC is manufactured.

Description	Application	Characteristics	Translation to Higher Level of Description
Algorithmic Specification	Functional system development, evaluation, verification, and specification	• executable behavioural description (VHDL) • causal timing, unclocked • abstract data types (e.g. reals, integers) • Very high execution speeds	manual
Behavioural or High Level (HL)	Executable functional reference (formal specification) for system verification	• executable behavioural description (VHDL) • composite bit data types (e.g. ranged integers, bit vectors) • High simulation speeds • clock-cycle-true and bit-true regarding inputs and outputs	manual or High Level Synthesis Tools (e.g. Phideo, Mistral2)
RegisterTransfer Level (RTL)	Input to Logic Synthesis Tools, design verification	• executable behavioural description (VHDL) • composite bit data types • clock-cycle-true and bit-true	Logic Synthesis Tools (e.g. Synergy, Synopsis or Compass)
Gate or Logic Level (LL)	Input to Technology Mapping, performance and timing analysis and design verification	• executable behavioural description (VHDL) • propagation delay based timing • bit value data types	Technology Mapping Tools succeeded by Layout Synthesis Tools

Table 1.1 Characteristics and application fields of the various description levels

1.5 A brief introduction to Phideo

With increasing levels of complexity, the design time of one chip generation can be close to the lifetime of that generation. This is where *architecture synthesis* becomes important. Automating this part of the design flow results in a great reduction of design time and time-to-market. This is especially important when many possible alternative implementations have to be explored. Another main motivation for architecture synthesis is that it allows the designer to concentrate on a compact, well documented description of the design which makes abstraction from all the details of the implementation possible. Furthermore decisions at these levels have a much higher impact on the final results than decisions at the lower levels.

The *architecture synthesis* tools in the literature can be classified into two groups: general-purpose approaches and DSP domain specific approaches. General-purpose approaches often result in architectural bottlenecks or suboptimal solutions. Instead of trying to build one universal compiler, an approach which is driven by the application will generate more efficient architectures. DSP compilers exploit domain-specific knowledge. DSP algorithms are characterized by a repetition of the same algorithm over and over again on new input data. Each execution of the algorithm must be completed within a fixed time interval. The mapping of input signals onto output signals must be done at the rate of the input signal arrival, not slower and not faster. The goal of a DSP designs is to minimize area (=*goal*) within the above mentioned timing *constraint*. This makes it different from general-purpose numerical processing, where the timing is an optimization goal, not a constraint.

The Phideo (PHIlips viDEO) system is an architecture synthesis tool which is application driven and aims at high-speed video applications [7][8][9]. Video algorithms such as HDTV are characterised by the fact that sampling frequencies are close to the achievable clock freqencies on chip. As a consequence it is not possible to execute many operations on the same hardware. Therefore these algorithms are usually implemented on clusters of heavily pipelined datapaths with a low multiplex factor. The amount of resource sharing is limited. In Phideo these clusters are mapped upon so called processing units (PUs).

Traditionally most attention is paid to bottlenecks in the arithmetic units of the chip. This can lead to the introduction of processing units which are tuned to the application. In high-speed video applications the bottlenecks are situated at totally different resources. The area consumed by memories can be much larger than the area of processing units. Also communication problems (for example between memories and processing units) can be a bottleneck. The order and speed at which I/O communication takes place can have a drastic impact on the overall design. Therefore Phideo concentrates on memory allocation and communication, which is reflected in the target architecture model within Phideo.

Phideo is based on an analysis of the manual design process and the different design decisions that are taken. It was found that some design actions are of a bookkeeping nature and are repeated over and over again. For other design actions an accurate cost function could be formulated and the problem could be solved formally. These two types of design actions can be automated. But other design decisions are left to the responsibility of the designer. In the Phideo system tools are used for subproblems in the design which have a well defined optimization goal and which deal with an amount of data which is too large for a human designer. The Phideo is an iterative system where the user interaction is very important. The designer remains responsible for guiding the exploration of the design space. Using this Phideo a significant reduction of the design time and design effort is possible. More about Phideo can be found in Chapter 2 on page 13.

1.6 Research Goal and Method

The goal of this master thesis is to evaluate the applicability of the architecture synthesis tool Phideo in a product development environment. As a test case, a part of an MPEG2 video decoder is taken. Since Phideo is developed to design high-throughput video applications, some parts of the decoder are chosen which have the highest throughput and the highest data rate. In the MPEG2 video decoder these are amongst others the Inverse Discrete Cosine Transform (IDCT), the Inverse Zig Zag Scan (IZZS) and the Inverse Quantizer (IQ). In these components the sample frequencies are approximate 13.5 MHz. With a clock frequency of 27 MHz this is one sample every second clock cycle.

Conventional MPEG2 decoder implementations need a lot of memory to decode standard format video sequences. In this paper an architecture proposal is described which reduces these memory requirements (see Chapter 4: "Video Decoder Architecture Study"). A consequence of this architecture proposal is the need for a higher speed decompressor. This implies an IDCT, an IQ and an IZZS which can operate at about twice the conventional speed. In this case the data rate approximates the clock rate, so approximately every clock cycle one sample must be processed.

The IDCT, IQ and IZZS of the MPEG2 decoder are designed with Phideo. The resulting design and the design method are evaluated and compared to conventional design methods, using the following criteria:

1. design time: - how much time does it take to make a design?
2. quality of the design: - how good is the design in terms of chip area?
3. applicability - what kind of designs can be implemented successfully?
4. design method: - does it fit in the PCALE design flow?

At the end of the research, it should be clear what category of designs can be implemented efficiently using Phideo, and whether the output of the Phideo, an RT-Level VHDL description is ready to be synthesized using existing RTL synthesis tools. And finally is it useful to use Phideo in the design of high-throughput video applications?

1.7 Report Overview

In Chapter 2: "Phideo" the architecture level synthesis tools Phideo is described. In Chapter 3: "MPEG-2" an introduction is given to the MPEG2 video compression standard, which is the source of the test-case. In Chapter 4: "Video Decoder Architecture Study" a video decoder architecture is presented which reduces the memory requirements. This architecture imposes a number of constraints and performance requirements on the computational units that must be designed. In Chapter 5: "Discrete Cosine Transform" discusses the transform and different algorithms to compute the Discrete Cosine Transform. Chapter 6: "Implementation of the IDCT with PHIDEO" discusses the implementation aspects of the IDCT with PHIDEO. In Chapter 7: "Conclusions on the use of Phideo" Phideo is evaluated and the advantages and disadvantages as well as the applicability are discussed.

Chapter 2

Phideo

The Phideo design method is based on an analysis of the manual design process and the different design decisions that are taken. It was found that some design actions are of a bookkeeping nature and are repeated over and over again. For some other design actions an accurate cost function could be formulated and the problem could be solved formally. These two types of design actions can be automated. But other design decisions are left to the responsibility of the designer, to exploit the experience and the creativity of the human designer. This has led to a new design method which is iterative and where the user interaction is very important. Using this new method a significant reduction of the design time and design effort is possible.

Architecture synthesis is very time consuming in the case of a manual design. This is especially important when many possible alternative implementations have to be explored. Indeed, the influence of design choices on the result is often not known at the time that the decisions are made. This could lead to backtracking on some decisions which is time consuming, or to a non-exploration of a part of the design space which could lead to suboptimal results. For this reason the development of tools is relevant to help the designer in this decision making process.

As a result the designer can concentrate on a compact, well documented description of the design at a high level of abstraction, which makes abstraction from all the details of the implementation. These decisions at this high level have a much higher impact on the final results than decisions at the lower levels.

Phideo is not a fully automated push-button system. User interaction is of major importance. Phideo automates large part of the design actions. But important design decisions are left to the designer. Phideo constructs a solution and provide the necessary information with this solution to adjust and improve it iteratively.

. This chapter discusses the global concepts and motivations of the Phideo design method. First a general introduction to architecture synthesis and the basic concepts and definitions are given. Next the characteristics of high throughput DSP (Digital Signal Processing) applications, the Phideo model of periodic operations and the target architecture of Phideo are discussed. This is followed by an overview of Phideo and its design method. More about Phideo can be found in [7][8][9][10].

2.1 Architecture synthesis

The input of architecture synthesis (also called high-level synthesis) is a behavioural specification (see figure 2.1). The behaviour is precisely described in terms of operations and computation sequences on the inputs to produce the required output. Architecture synthesis translates the behavioural specification into a description at Register Transfer Level. The position of architecture synthesis in the design path is described in "Design automation" on page 4.

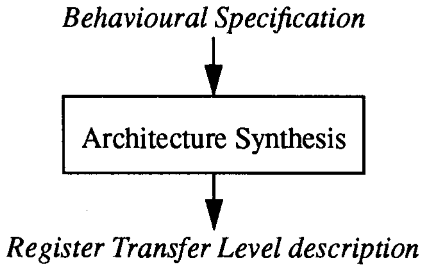

Behavioural Specification
Architecture Synthesis
Register Transfer Level description

Figure 2.1 Architecture synthesis

As shown before, , three different subtasks can be distinguished in architecture synthesis :

- resource allocation
- scheduling
- resource assignment

Resource allocation is the selection of the number and type of functional units which will be used, this includes the allocation of memories and registers. Scheduling is the assignment of each operation to a cycle step in which it will be executed. Furthermore each operation must be mapped on one of the allocated functional units, to define on which unit each operation will be executed (resource assignment). In case of memories, variables must be assigned to exact memory locations to define where in the memory these variables are stored (location assignment), which is also a part of resource assignment.

These three subtasks are not independent. A scheduler must know in advance which functional units or modules are available. The allocation process must know how many operation of a certain type must be executed at the same time to prevent conflicts. The resource assignment must know at what cycle step a unit is occupied to prevent conflicts in module usage. Hence scheduling and allocation are closely related to each other, and their results heavily depend on each other.

The following example illustrates the different tasks within the architecture synthesis. Consider the following behavioural specification:

e = a + b + c + d

In figure 2.2 two different schedules are given for this example. The number of operations with the same type in the same cycle determine the number of modules that are necessary to implement this schedule. We can describe these resource needs by means of a *distribution function.* This function gives the number of operations which are executed simultaneous. Only the maximum number of operations is relevant to determine the resource requirement, since it reflects the maximal number of simultaneous operations which determines the minimal resource requirements. The allocator must select enough modules to perform these operations. The schedule of figure 2.2a needs at least two adders and can be executed in 2 cycle steps. The schedule of figure 2.2b needs only one adder but takes 3 cycles to execute.

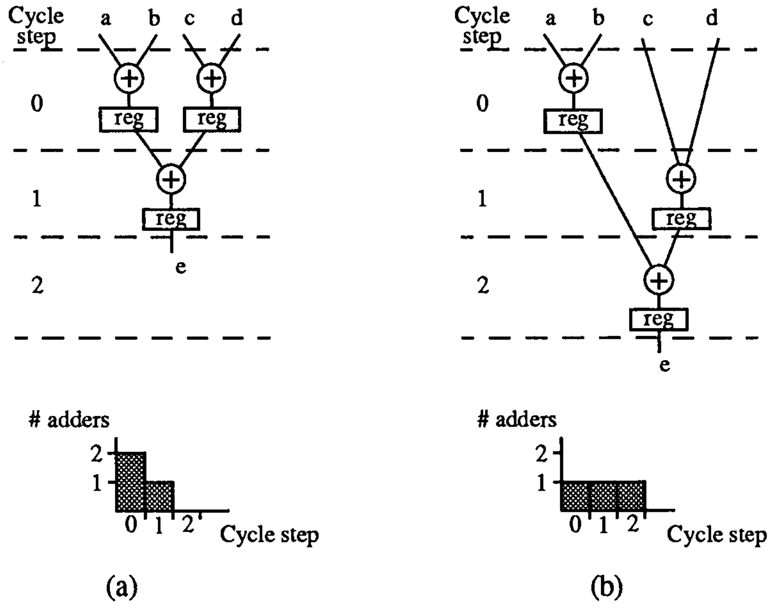


Figure 2.2 Two different schedules and distribution functions

The number of clock cycles that a function needs to be executed is called the *latency.* The latency of an addition in the example is 1 cycle. The complete function has a latency of 2 and 3 cycles, respectively. It is possible to *pipeline* the function. Pipelined modules can start a new execution before the previous execution has finished. This means that the module can contain more than one data value at the same time. In the example new data could be inserted in the function after the execution of the operations in cycle step 0. The *frame* is the time between two successive executions, which is also called the *data introduction interval* (DII), in this case the data introduction interval is 1 clock cycle. The *throughput* of a function is the number of executions of this function per cycle. Although the latency of the function of figure 2.2a is 2 cycles the throughput is 1 execution per cycle.

2.2 Application driven architecture synthesis

Traditionally most attention is paid to bottlenecks in the arithmetic units of the chip. This can lead to the introduction of processing units which are tuned to the application. But there also exist designs where the bottlenecks are situated at totally different resources. For example, for realtime video applications the area consumed by memories can be much larger than the area of processing units. Also communication problems (for example between memories and processing units) can be a bottleneck. The order and speed at which I/O communication takes place can have a large impact on the overall design. The identification of bottlenecks is important because it is very useful to design this part first. The less critical parts are designed afterwards. For example in the case of arithmetic bottlenecks, the designer may decide to design the processing kernel of the IC first. This can be done in much detail (possibly at the layout level) before the rest of the chip is designed.

Each specific application domain has its typical properties and its own bottlenecks. Instead of trying to build one universal compiler, the Phideo approach is driven by the application, which has great advantages. Typical properties of the application domain that are exploited by the designers are reflected in the design method and in the design flow.

The design of an application driven architecure synthesis tool starts from an analysis of the application field. Important decisions are taken at the architectural level. So the characteristics of the application domain are reflected in a *target architecture.* The target architecture must allow enough flexibility so that it can span the complete target application domain. Several target architectures exists for DSP. They can be classified dependent on the application domain and the corresponding sampling frequency.

1. For linear filters in the audio, speech and telecommunication applications with relative low sampling frequencies a hard-wired bit serial approach is often used. Signals are processed bit by bit, least significant bit first. The data rate approximates the clock rate divided by the word length, e.g. 10 MHz clock and 16 bit words imply a data rate of 660 kHz.

2. For complex decision-making applications with low to medium sampling frequencies (10 kHz up to 1 MHz) microcoded approaches are used. Typical architectures use a limited number of general-purpose arithmetic building blocks (such as an ALU with 32 different instructions) and a limited number of on-chip memories.

3. For high-throughput DSP applications, like real-time video applications, with high sampling frequencies (up to 100 MHz) more parallelism is needed. This is reflected in the architecture because distributed processing and storage elements will be used. Typical applications are real-time video and image processing.

2.3 High throughput DSP applications

High throughput Digital Signal Processing algorithms are characterized by a repetition of the same algorithm over and over again on new input data. Each execution of the algorithm must be completed within a fixed time interval. In case of pipelined units, not the latency of the algorithm is of interest but the throughput. The unit must be able to execute the algorithm once per fixed time interval. The mapping of input signals onto output signals must be done at the rate of the input signal arrival, not slower and not faster, to prevent additional buffering requirements. The goal of a DSP design is to minimize area (=*goal*) within the above mentioned timing *constraint*. This makes it different from general-purpose numerical processing, where the timing is an optimization goal, not a constraint.

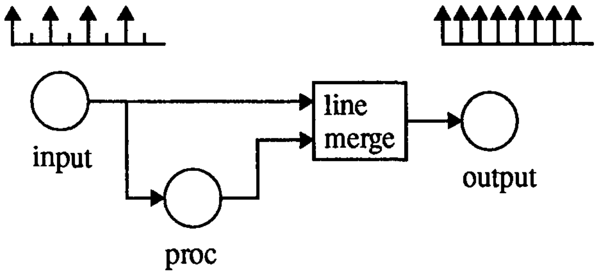
Video processing algorithms are characterised by the fact that sampling frequencies are close to the achievable clock freqencies on chip. As a consequence it is not possible to execute many operations on the same hardware. For video applications we need much more parallelism in the architecture. Furthermore the arithmetic units become much more dedicated, and will be used to perform one function over and over again. Complete clusters of operations are mapped on one processing unit (PU) which can be heavily pipelined. The amount of resource sharing is limited. As a consequence we also need distributed memories because otherwise we would have a communication bottleneck in the architecture. This is different from the centralized memory approach for low and medium throughput applications

Since communication is an important bottleneck in high throughput DSP applications, Phideo concentrates on the communication bottlenecks. The synthesis and allocation of distributed memories play an important role in Phideo. This is reflected in the *target architecture* of Phideo. Also the need for heavily pipelined dedicated units to provide the required throughput is recognized and reflected in the target architecture.

In the next section the Phideo model of periodic operations is discussed which plays an important role in the description of repetitions in high throughput applications. Followed by the target architecture which is used as the skeleton for Phideo designs.

2.4 Phideo model of periodic operations

In video applications many different rates occur. Input and output rates are defined in the specification. For internal signals an optimum rate can be chosen. In figure 2.3 a small example is given. Half of the output samples are computed by an operator *proc*. The output is the original signal merged with the computed signal which results in a double output rate.

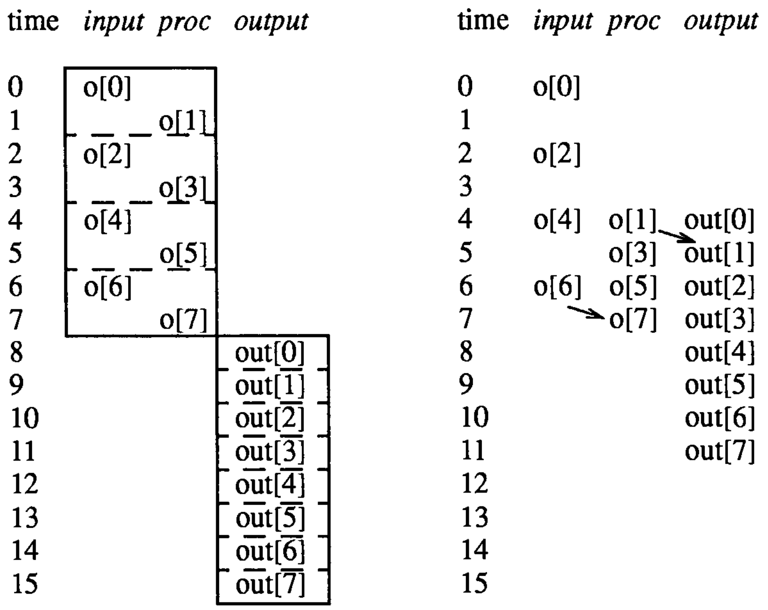


```
for i = 0 to 3
begin
    o[2i]= input();
    o[2i+1] = proc ( o[2i] );
end
for j = 0 to 7
    output( o[j] );
```

Figure 2.3 Small multi-rate example

In figure 2.4 two possible schedules are given. Figure 2.4a interprets the loop in a procedural way although the functional specification allows more parallelism. This way the first loop has to be finished before the second loop can start. The solid lines in the figure denote the borders of the loops, and the dashed lines denote the borders of the iterations within the loops. However, these borders are artificial constraints, which come on top of the true data dependency constraints. Such a solution may be acceptable for low throughput applications, but is far from optimal for high throughput applications.

Figure 2.4b shows an implementation with overlapping loop executions. Also the repetition times are different for operations within one loop, e.g. for operation *proc* and *input*. The second implementation shows less latency. This leads to smaller memory requirements since the life-times of the variables are shorter. For real-life examples with a large number of samples instead of only 4 samples, the difference becomes quite large. This optimization across loop boundaries is essential for an efficient implementation.



time	*input*	*proc*	*output*
0	o[0]		
1		o[1]	
2	o[2]		
3		o[3]	
4	o[4]		
5		o[5]	
6	o[6]		
7		o[7]	
8			out[0]
9			out[1]
10			out[2]
11			out[3]
12			out[4]
13			out[5]
14			out[6]
15			out[7]

(a)

time	*input*	*proc*	*output*
0	o[0]		
1			
2	o[2]		
3			
4	o[4]	o[1]	out[0]
5		o[3]	out[1]
6	o[6]	o[5]	out[2]
7		o[7]	out[3]
8			out[4]
9			out[5]
10			out[6]
11			out[7]
12			
13			
14			
15			

(b)

Figure 2.4 Two schedules of the example of figure 2.3

Within Phideo this problem is solved as follows. First each operation in the specification is considered as a separate design entity. Second the assumption is made that executions of the same operation are periodical in time, which is inspired by the nature of video signals. This leads to the *model of periodic operations* which is a basic concept of Phideo.

Periodic operations can be multidimensional. The translation from loops to periodic operations is done as follows. All individual operations in a loop body are translated to separate periodic operations in order to decouple those operations. The number of dimensions of a periodic operation is equal to the number of enclosing loops. The number of executions in a periodic operation is given by the iteration bounds of the enclosing loops. Each periodic operation can be defined by the start time and the periods. The periods are defined by the user, the start time is determined by the scheduler. All operations are handled as separate entities and thus may be scheduled concurrently.

In figure 2.5 an example of a periodic operation with two dimensions is given. For the example in figure 2.4. we have three periodic operations with one dimension each indicated as *input*, *proc* and *output*. For the *proc* operation we have 4 executions, a start time of 4 and period of 1.

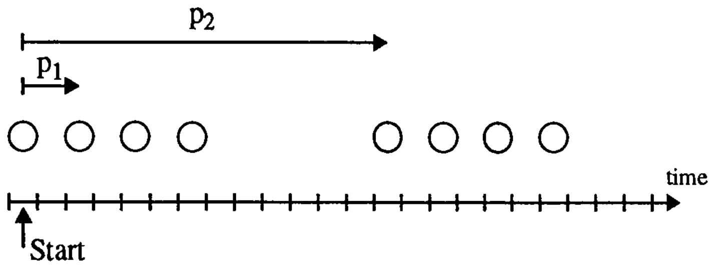


Figure 2.5 Example of a periodic operation with two dimensions, p1 and p2 denote the two periods of the operations.

With this model the original loop hierarchy is replaced by a new hierarchy, in which all operations are independent concerning their periodicity and iteration bounds. It would have been hard to obtain the same results by unfolding and splitting the loops.

2.5 Target Architecture of Phideo

Processing units (PUs) play an important role in Phideo. They can become complex and irregular because data-conditional operations are included. For performance and area reasons these operations are moved from the controller into the PUs in Phideo. The user must identify clusters of tightly coupled operations in the specification that can be mapped on the same processing unit. These clusters are handled as single operations and an abstraction of the PU made for the use in the rest of Phideo.

Figure 2.6 shows an overview of the target architecture model of Phideo. A number of PUs (PU1, PU2, .. PUk), a number of memories (M1, M2, .. Mn) and a number of address generators (AG1, AG2, .. AGm) and a controller can be distinguished. Input and output terminals can be considered as simple PUs. All types of memories can be used in the architecture, including static RAMs, dynamic RAMS, register files and single registers. The role of the memories is to transport data from producing PUs to consuming PUs. The number of allocated PUs and memories can be different. In general, the PUs are active simultaneously.

Generally, all PUs can have access to the same memory locations. The interconnections take care of the data transport between PUs and memories. Similarly the role of the address generators is to generate the correct addresses for the memories at the correct points in time. Again interconnections take care of the transport of addresses to the memories.

The controller generates the signals which perform the selection of the function of a PU, read/write signals for the memories, next address signals for the address generators and the control of the multiplexers in both communication networks. These control signals are periodic as well which is reflected in the use of counters in the multidimensional controller.

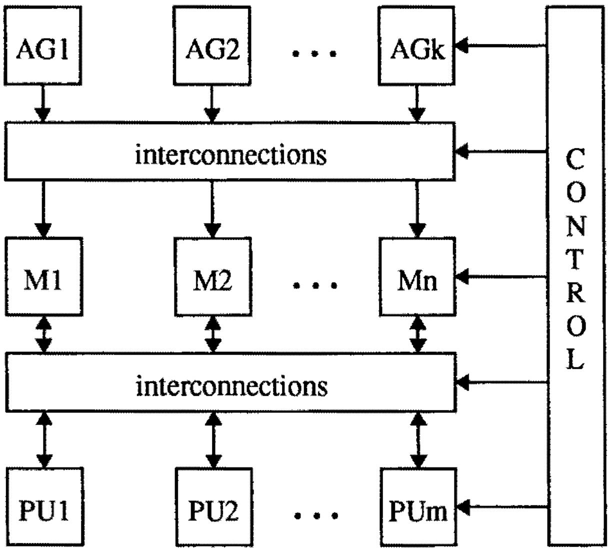


Figure 2.6 The Phideo architectural model, which allows freedom with respect to the number and types of processing unit (PU1, PU2, ... PUk), memories (M1, M2, ... Mn) and address generators (AG1, AG2, ... AGm)

2.6 Overview of Phideo

Phideo consists of a set of tools which can be executed in succession. The development of the tools within Phideo is driven by the architectural model, which include a scheduler to define the start times of the periodic operations and a set of tools for memory synthesis, including address generation and controller generation tools.

The first part of the design process is the clustering of operations (figure 2.7). The selection of the operations that are mapped onto so called Processing Units (PUs) is performed manually and is an important task for the user. Groups of operations which are tightly coupled and which often contain the main arithmetic complexity of an algorithm are selected to be implemented in a processing unit. The scheduler handles such a cluster as single operations.

The next step is the definition of the algorithm in PIF (Phideo Input Format). Functions and Processing Units are declared and the algorithm is defined. With pragmas, user defined constraints can be defined to drive the scheduling process and the memory synthesis. Phideo makes a schedule and gives additional information like distribution functions, number of variables alive at the same time and the number of simultaneous memory accesses. The user can use this information to adjust the clustering process or to steer the scheduler by defining additional constraints.

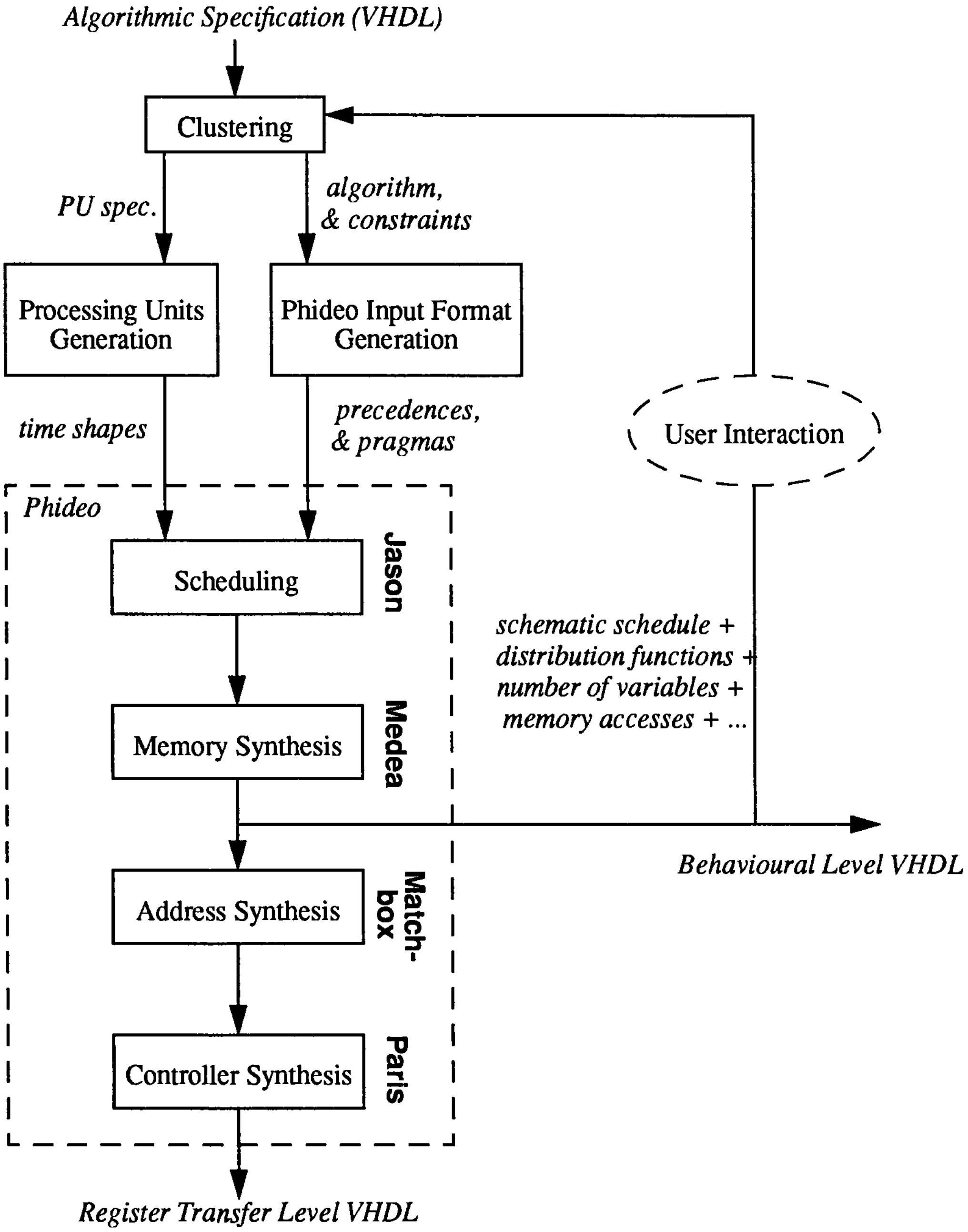


Figure 2.7 Overview of Phideo

If the schedule and the memory and PU-allocation are performed satisfactory, the assignement of variables to exact memory locations can be made together with address generators to provide the memories with the proper addresses. Finally the controller can be constructed which is partly generated by all the tools in Phideo. In the next section an overview of the synthesis tasks can be found.

2.6.1 Processing Units design

The design of the processing units is done using RT level synthesis or by manual design. Because of the high sampling frequencies that occur in the application, these units are often pipelined to meet the high throughput requirements. It is also possible to design the processing units with Phideo. In that case the processing unit contains a complete Phideo designs can be used hierachically in a higher level design.

Phideo makes an abstract model of a processing unit which is called a *time shape.* The time shape defines the timing of inputs and outputs, relative to the first input. An example of a time shape of a PU with 4 inputs and 1 output is shown in figure 2.8. The time shape is used by the scheduler in the next step.

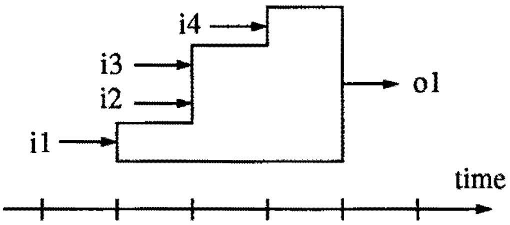


Figure 2.8 Time shape of a PU with 4 inputs and 1 output

Instead of designing a PU at RT level, it is possible to define the time shape without designing the complete PU. The timing of the inputs and outputs can be based upon an estimation. At a later stage when a preliminary feasibility study with Phideo has provided enough information, a real implementation can be made. This way unnecessary effort can be saved if the design proved to be infeasible. In order to simulate the designs the internal behaviour of the processing unit can be described at a behavioural level containing integer and real value operations.

Figure 2.9 and figure 2.10 illustrate the construction of a time shape for two different implementations of the function e = a + b + c + d. The two different time shapes F and G are used in the next section as an example.

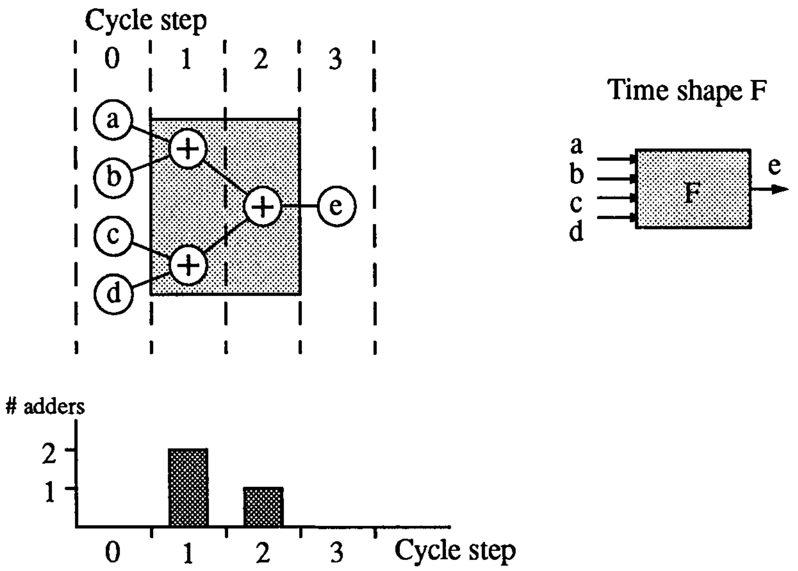


Figure 2.9 Implementation of a PU with the function e = a + b + c + d, with the distribution function of the adders and the corresponding time shape F of this PU.

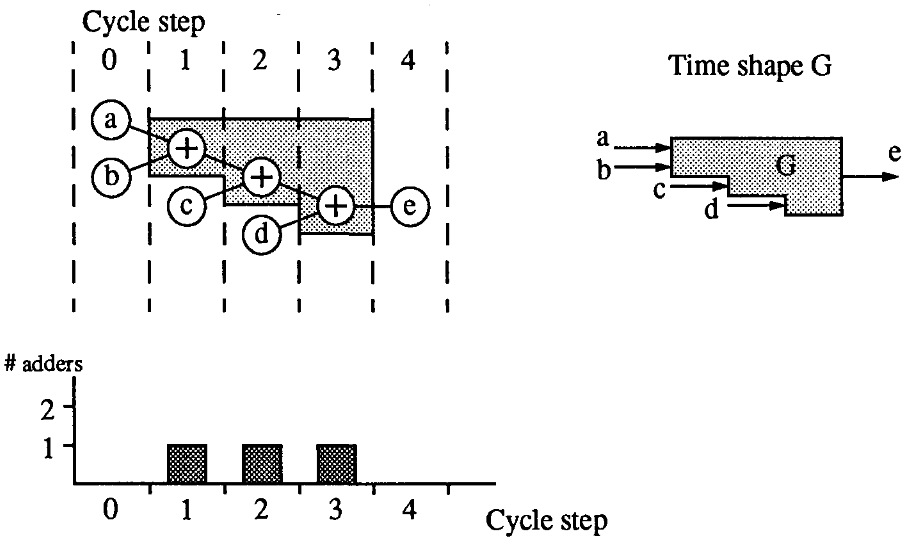


Figure 2.10 Another implementation of a PU with the function e = a + b + c + d, with the distribution function and the corresponding time shape G of this PU.

2.6.2 Scheduling and PU-allocation

The scheduling and PU-allocation are performed by the tool called *Jason.* Jason selects the necessary units (*PU allocation*) and generates a schedule. Furthermore feedback is given on this schedule to evaluate the results and improve the schedule and PU-allocation by defining additional constraints in terms of pragmas. The feedback consists of the distribution functions, memory requirement functions and memory access functions which are explained later on. .

The schedule of Jason defines which data is written to memories at which times. In the following PIF example the function F is used with the time shape defined in figure 2.9. An input and an output terminals is defined. The global period is 4 which defines that the algorithm must be repeated every 4 clock cycles. The inputs are read with a period of 1, so every cycle an input is read. Function F adds these 4 inputs and the result is passed to the output.

```
infunc input = in_term;
outfunc output = out_term;

func F( a, b, c, d ) e = F_pu;

{4} /* global period */

        (i: 0 .. 3) {1} ::
{in}           in[i] = input (); /* a,b,c,d, resp. */

{func}  e = F(in[0], in[1], in[2], in[3]);

{out}   = output(e)
```

Every time an operation is executed a new instance of the time shape is generated. Because of the model of periodic operations, consecutive executions of the same operation are equidistant in time. Different operations can have independent start times and different periods, the global period is equal for all operations.

The task of the scheduler (Jason) in Phideo is to select the start times of the operations. The definition of the periods is still a manual step. Figure 2.11 shows the schedule with the matching functions. The *distribution function* is used to represent the resource requirements as a function of time.

The *memory requirement function* represents the number of variables which are alive simultaneously. This function is used as a measure for the required memory size, since these variables need to be stored in a memory. The *memory access function* reflects the communication bandwidth and is an indication for the number of memories needed for implementation.

The schedule shows that the variables on the input which arive in sequence, are not used immediately. These have to be stored temporarily which is reflected in the memory requirement function and the memory access function. These variables are retrieved from the memory and passed to function F at cycle 4. After a latency of 2 cycles the result is sent to the output.

If another implementation of the function F is used, i.e. the implementation with time shape G, the schedule of figure 2.12 is generated. This leads to different access profiles and different memory requirements. Note that although the throughput of the implementation with time shape F is higher (i.e. once execution per two cycles) the use of the implementation with time shape G (throughput of one per three cycles) results in a better schedule since the number of variables and number of accesses are smaller. Note that the function G is scheduled before the loop which takes in the inputs is finished.

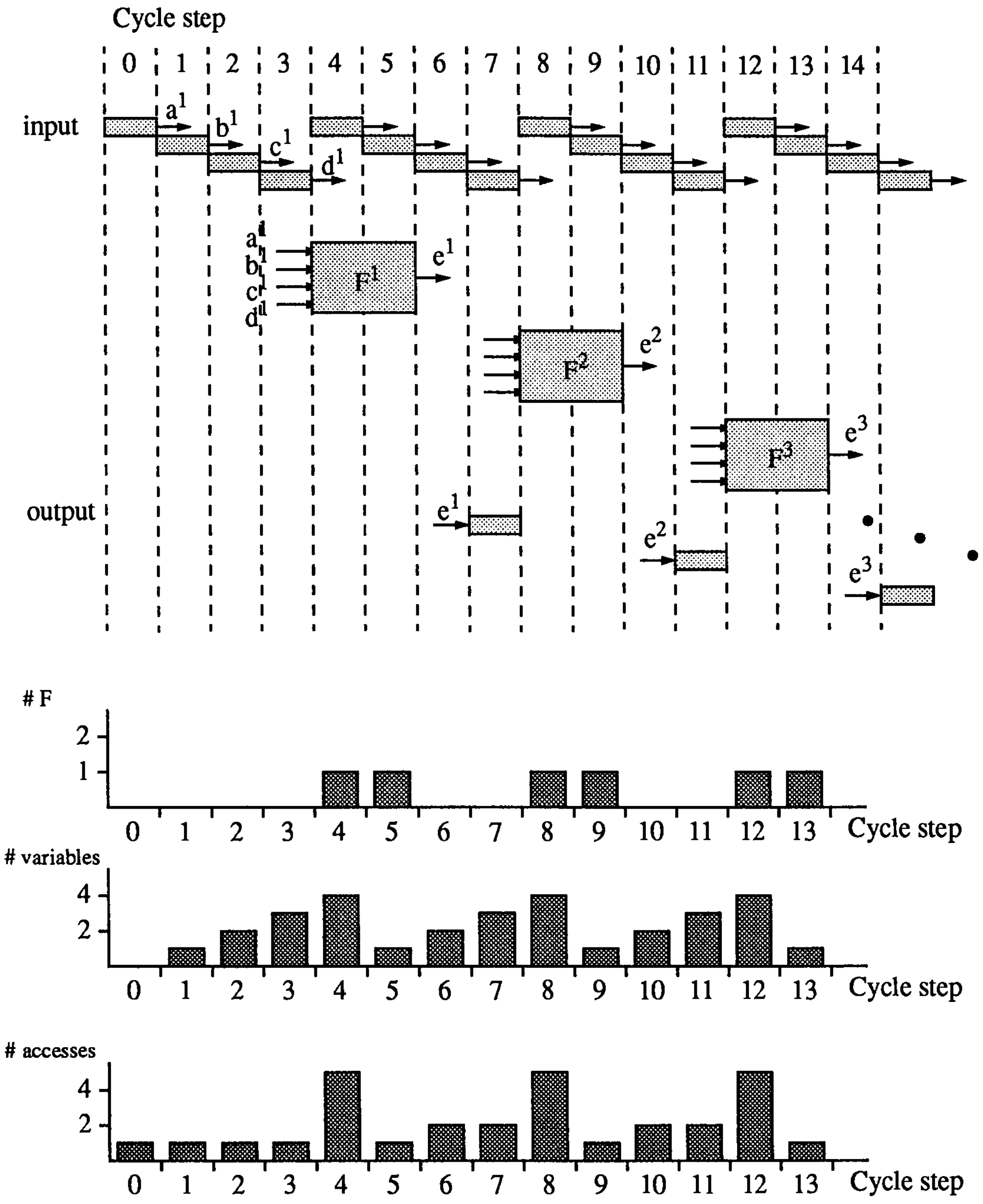


Figure 2.11 Schedule generated by Phideo using time shape F, together with a distribution function, the memory requirement function and the memory access function.

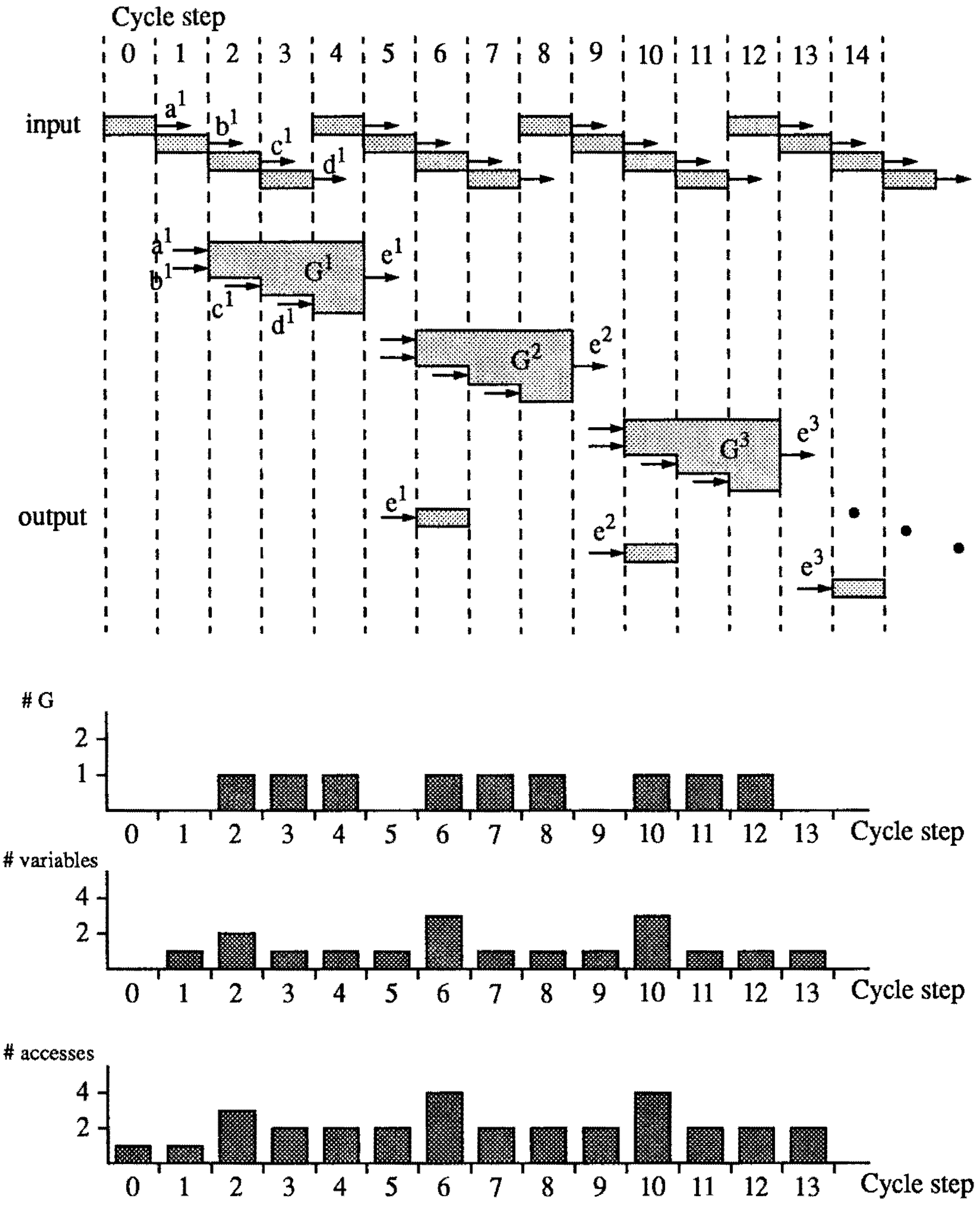


Figure 2.12 Schedule generated by Phideo using time shape G, together with a distribution function, the memory requirement function and the memory access function.

Pipelined schedule

The main difference between an conventional scheduler and the scheduler in Phideo is that the schedule which is generated by Phideo can be a pipelined schedule. The scheduler in Phideo generates a schedule which is to be repeated with a defined period, the so called frame period. If the latency of the algorithm is larger than the frame period, the schedule is designed to be pipelined with a data introduction interval equal to the frame period. The schedule is folded to determine the resource needs and memory requirements in each cycle step.

In figure 2.13 an example of a pipelined schedule is given. The function $F : e = (a + b) * c + d$ is scheduled with a frame period of 2 cycles. Since the function is repeated every two cycles several instances of the function F (i.e. F^{-1}, F^{0}, F^{1}, F^{2}, ... F^{5}) are depicted. It can be seen (distribution function) that the implementation of this schedule requires 2 adders. When a constraint on the number of adders is given, Phideo will optimise this schedule by moving the last addition of the function and the output to the next cycle step (illustrated by the arrows). As a result each cycle step only contains one addition and the schedule can be implemented using only one adder. The cost of this adaptation is two register since both inputs of the addition must be stored for one clock cycle. Note that it is not relevant that the latency has increased by one cycle, since the throughput remains the same (i.e. once per two cycles)!

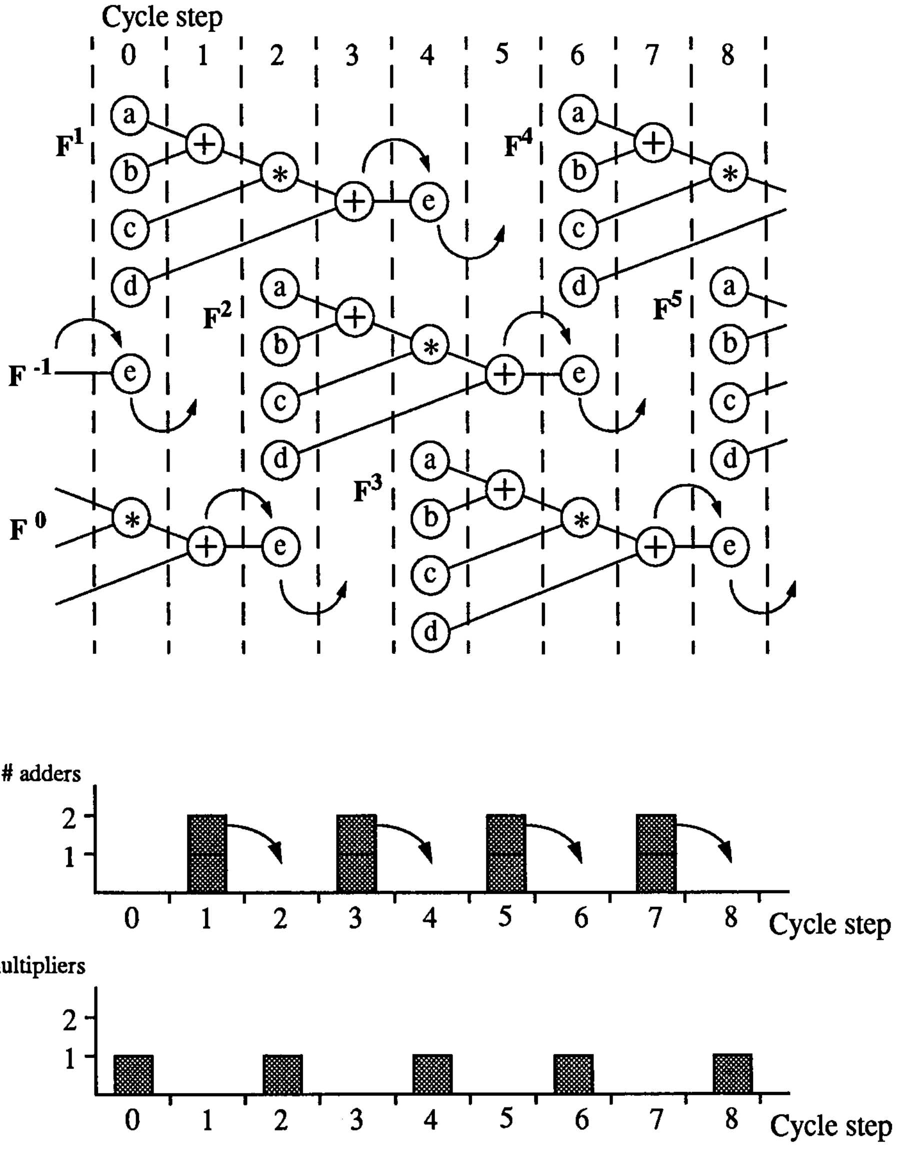


Figure 2.13 Pipelined schedule, global period is 2 clock cycles.

2.6.3 Memory synthesis

Given the schedule of Jason, *Medea* determines the number of memories needed to implement the communication requirements (*Memory Allocation*) and decides which data is to be written in which memory (*Memory Assignment*) to solve communication bottlenecks. The result is a complete data path, consisting of processing units and memory units.

The memory allocation and memory assignment is subjected to the constraints which are defined by the memory types. The available memories can be all types of memories including random accessible memories, registers and register files. The memories can be characterised by the number of read ports (R), the number of write ports (W) and the number of bidirectional ports (RW). For example, a single port RAM is characterised by R=W=0 and RW=1 and a two-port register file by R=W=1 and RW=0. For every type the maximum and minimum size and the access times (in clock cycles) are specified.

Medea designs a memory architecture and optimizes the estimated size of the memories and the number of memories. At this point the exact sizes of the different memories are not yet known. Also the precise locations of the variables in these memories are not known. These are defined by the address synthesis which is described in the next section. During the address synthesis also the address generators are constructed. The controller is partly specified in each tool and is synthesized afterwards by the controller synthesis tool.

2.6.4 Address synthesis

The address synthesis process is performed by the tool *Matchbox.* Matchbox takes as input the schedule and the memory allocation information generated by Jason and Medea. At this stage the number and the type of memories which are required is exactly known. Furthermore it is known which data has to be stored in or retrieved from which memories as well as the exact timing. Matchbox must assign the data samples to exact location in each memory (*location assignment*) and generate the address generators which must produce the required address sequences (*address generation*).

As a result of the Phideo model of periodic operations the data that is produced by the PUs (and the data that is consumed by the PUs have a very regular and periodic nature as well. Therefore the address generators are based upon counters. During the location assignment a trade-off between the required memory size and the complexity of the address generator is applied. At this moment three different types of address generators are available which offers a good trade-off.

2.6.5 Controller synthesis

After the address synthesis, all parts of the netlist generated by the different tools are connected and a controller is generated by the controller synthesis tool *Paris.* From this netlist automatically a synthesizable RT-level VHDL description is generated. Also a schematic drawing of the complete design can be shown, which can be very useful as feedback to the designer.

2.7 Phideo Input Format

PIF is a temporary interface to a number of Phideo tools (scheduler, memory synthesis, address generation). In the near future PIF should be replaced by another language (possibly VHDL extended with additional commands). In this chapter a brief list of the most commonly used commands is given.

An example of a PIF description is shown below:

```
/* delarations */

infunc input = interm;
outfunc output = outterm;
func +(input1, input2) output = adder;

signal a,b,x = 8;

/* algorithm */

{16} : [0,16] /* global period */

         (i: 0 .. 7) {1} ::
         begin
{out}          = output(x[i]);
               x[i] = a[i] + b[7-i];
{ina}          a[i] = input();
{inb}          b[i] = input();
         end;
```

A PIF description consists of two main parts: a declarative part and an algorithmic part. Furthermore additional constraints or pragmas can be defined to steer the synthesis process. With the declarations processing units are defined by means of functions and the input and output terminals can be defined as well as signals and their widths. The algorithmic part describes the algorithm in terms of loops, assignments and function calls. Finally the *pragmas* are used to impose constraint on the schedule and the allocation process. More about PIF can be found in appendix A.

2.8 Phideo hierarchy

It is possible to use a Phideo design hierarchically in a higher level design. At this moment a lower level Phideo design must be included by means of a macro. The Phideo design is simply substituted in the new design. This way Phideo's hierarchy is not very clear and certainly not transparent. In future releases hierarchy is planned to be expanded.

After running Jason a bounding box is generated which contains a description of the time shape of the PIF design. The input and outputs terminals are modelled by functions and the input and output order and period of the signals is copied. This bounding box can be used in the higher level design.

Usualy the names and the index of variables are transparant and not of any influence. This is not true in this case! When the macro is used at a higher level, the Phideo preprocessor substitutes the names of the variables in the macro. The order and period of the signal remain the same. As a result the internal index order of the input and output variables are used for the substituted variables as well.

When familiar with this phenomenon, the hierarchy by means of macros can be used safely. An example of the use of a macro in Phideo given in appendix B.

2.9 Testing and simulating Phideo designs

Within the Phideo design path three levels of abstraction can be distinguished as shown in figure 2.7:

1. Algorithmic specification (PIF)
2. Behavioural Level description (VHDL)
3. Register Transfer Level description (VHDL)

At the lower levels of abstraction the design contains much more detail which leads to slower simulation times and it takes much more effort to track down functional errors. It is desired that the design can be tested at the higher levels of abstractions as well.

At this point it is not possible to simulate the initial PIF description directly. Though the initial PIF description is very transparent and easy to be read, typing errors and other small errors can corrupt the design. First a initial pass through Phideo is needed, to generate a Behavioural Level VHDL which can be verified. This can be done quite quickly since no optimization is needed. It would be useful if the PIF algorithm would be made executable so that a simple functional test can be performed. At this point it is to recommended to test the design after the first passage before starting the iteration process, in order to verify the PIF algorithm at an early stage. Otherwise a lot of unnecessary time can be spent in optimising a design iteratively.

The Behavioural Level VHDL description which is generated by Phideo is of major importance in the design path. This description contains all the functionality of the target design and is clock-cycle true regarding its inputs and outputs. The memories which are allocated during the scheduling process are simulated by a behavioural model. At this point it is known which variable are to be stored in which memories. The exact location within the memories and the address generation for storage and retrieval of the data in these memories not is not implemented at this stage. This way the simulation speed of the behavioural VHDL can be kept much higher, so that extensive testing is possible.

Another way of increasing the simulation speed at this level is the definition of the Processing Units. The internal behaviour of these PUs can be described at a behavioural level as well. Arithmetic operations like multiplications or additions can be described using integers and reals. The only constraint is that the inputs and outputs of the PUs must be clock-cycle true compared to the final design. At a later stage the internal of the PUs can be translated into Register Transfer Level. This part can be verified by means of a one-on-one comparison of the outputs as described in "PCALE design flow" on page 6.

After memory synthesis the Register Transfer Level VHDL can be verified. Since extensive testing are performed at the higher levels most functional errors will be removed already, so the number of test at these lower levels can be limited.

2.10 General conclusions

With increasing complexity and design time of Integrated Circuit architecture synthesis becomes important. Architecture synthesis can reduce the design time and the time-to-market. To avoid architectural bottlenecks the architecture synthesis tools Phideo is driven by the application. It aims at high throughput DSP applications. Phideo concentrates on the memory allocation and communication, imposed by the traditional bottlenecks of these high throughput applications.

Phideo is not a push-button design system, user interaction is very important. Several design steps are automated. These include bookkeeping tasks and tasks which could be optimized formally. With Phideo it is possible to explore the design space by comparing alternative implementations. Not only the solution to a problem is of importance, the necessary information to judge and improve the design iteratively are at least as important. Phideo provides this information and offers the possibility to interact and steer the synthesis process. Phideo is a powerful design method that gives the designer the opportunity and tools to explore the design space and to implement a design in a short time.

Chapter 3

MPEG-2

3.1 Standardization effort

Standardization of video compression techniques has become a high priority because only a standard can reduce the high cost of video compression encoders/decoders and resolve the critical problem of interoperability of equipment from different manufacturers. The existence of a standard is often the trigger to the volume production of integrated circuits (VLSI) necessary for significant cost reductions.

International standardization committees have been working on the specification of several compression algorithms. The Joint Photographic Experts Group (JPEG) of the International Standards Organization (ISO) has specified an algorithm for compression of still images [17]. The International Telecommunication Union (ITU, former CCITT) proposed the H.261 Standard for video telephony and video conference [15]. The Moving Pictures Expert Group (MPEG) was established in 1988 in the framework of the Joint ISO/IEC Technical Committee. MPEG was established to develop standards for coded representation of moving pictures, associated audio, and their combination when used for storage and retrieval on digital storage media (i.e. CD-ROM, DAT, tape, VCR) as well as transmission on telecommunication channels (i.e. cable networks and satellite links).

The motion Pictures experts Group (MPEG) of ISO has completed its first standard MPEG-1 [18], which can be used for interactive video and provides a picture quality comparable to VCR quality. Currently MPEG is working on the second (MPEG-2 [11]-[14]), which will provide audiovisual quality of both broadcast TV and HDTV. Originally the need for a third standard was foreseen (MPEG-3), which was intended for HDTV, but this standard was dropped when it became apparent that the functionality supported by the MPEG-2 requirements made this standard redundant. A fourth MPEG standard (MPEG-4) targets at coding audio and video signals at very low bitrates.

The major characteristics of image formats and coding standards are listed in table 3.1 and table 3.2. These tables are not complete. They should give just some indications of image formats, and application fields.

Video Format	**Frame Size Luminance**	**Frame Rate**	**Frame Store (luminance only)**
HDTV (16:9)	1920x1152	50 Hz	17 Mb
HDTV (4:3)	1440x1152	50 Hz	13 Mb
ITU-R-601 (former CCIR-601)	720x576 (PAL) or 720x480 (NTSC)	25 Hz 30 Hz	3.2 Mb 2.6 Mb
CIF/SIF	352x288	30 Hz	0.8 Mb
QCIF	172x144	30 Hz	0.2 Mb

Table 3.1 Digital Video Formats

Name	**Typical Application**	**Typical Image Format**	**Coded Bit Rate**
JPEG (ISO)	Photo-CD, Photovideotext	Any size 8 b/pel	0.25 .. 2.25 b/pel
H.261 (ITU)	Video telephony, Video conference	QCIF, CIF 10 Hz .. 30 Hz	p x 64 kb/s $1 \leq p \leq 30$
MPEG-1 (ISO)	CD-ROM, CD-I, Computer applications	SIF 25 Hz, 30 Hz	≤ 1.5 Mb/s
MPEG-2 (ISO)	Broadcast TV	ITU-R-601 HDTV	≤ 100 Mb/s

Table 3.2 Video Coding Standards

The MPEG activity was not started without consideration to other standards committees. The activities of JPEG (Joint Photographic Expert Group) played an important role in the beginning of MPEG. Although JPEG focused exclusively on still-image compression, the distinction between still and moving image is thin; a video sequence can be thought of as a sequence of still images to be coded individually, but displayed sequentially at video rate. However the "sequence of still images" approach has the disadvantage that it fails to take into consideration the extensive frame-to-frame redundancy present in all video sequences. MPEG aims at this potential additional factor of three in compression exploiting the temporal redundancy. More about the MPEG standardization approach can be found in [19].

3.2 Levels and Profiles

The MPEG-2 standard is intended to be generic in the sense that it serves a wide range of applications, bit rates, resolutions, qualities and services. Applications should cover, among other things, digital storage media, television broadcasting and communications. Considering the various requirements prompted by the different applications, necessary algorithmic elements have been developed, and they have been integrated into a single syntax. Hence this standard facilitates the bitstream interchange among different applications.

However the implementation of the full syntax of the specification is not very practical. Not all applications need all the features and since the inclusion of a feature introduces additional costs (more memory, faster processing, etc.), this is not desired. The MPEG-2 standard therefore defines a number of subsets of the syntax indicated by means of profile and level.

A profile is a defined subset of the entire MPEG-2 syntax. A level is defined as a set of constraints imposed on parameters in the bit stream. The constraints may include limits on the values of the parameters, such as limits on the picture resolution and frame rate.

The MPEG-2 standard foresees five profiles:

- Main profile: with no scalability and maximum quality
- Simple profile: same as Main, but without interpolated pictures (B-pictures, see "Temporal Redundancy Reduction" on page 38) in order to save memory
- SNR scalable profile: an improvement over Main giving scalability in signal-to-noise ratio
- Spatially scalable profile: also scalability in spatial picture resolution is supported and
- High profile: supporting 4:2:2 (see "Source coding format" on page 36) and full scalability.

Levels are associated to each Profile. MPEG-2 identifies four levels:

- Low Level: similar to CIF of ITU-T Rec. H.261 or SIF of MPEG-1
- Main Level: corresponding to conventional television
- High1440 Level: roughly corresponding to HDTV with 1440 samples per line
- High Level: roughly corresponding to HDTV with 1920 samples per line

In order to maximise interoperability, only a subset of all permutations are permitted. Those combinations which are allowed are shown in table 3.3.

	Simple Profile	Main Prolfile	SNR Scalable Profile	Spatially Scalable Profile	High Profile	Maximum Frame-Size, Frame-rate
High Level		X			X	1920x1152 60 Hz
High-1440 Level		X		X	X	1440x1152 60 Hz
Main Level	X	X	X		X	720x576 30 Hz
Low Level		X	X			352x288 30 Hz

Table 3.3 Allowed profile/level combinations and upper bounds for frame-size and frame-rate

3.3 The MPEG-2 compression algorithm

The MPEG compression algorithm consists of three stages:

- bandwidth reduction: by matching the source resolution to the MPEG source format
- the compression algorithm itself,
  - removal of spatial and temporal redundancy by means of waveform analysis and subjectively adapted quantization
  - followed by lossless compression using entropy coding (variable length codes)
- mapping of the resulting information losslessly into a bitstream by way of a syntax.

The MPEG video compression algorithm relies on two basic techniques: block based motion compensation for the reduction of the temporal redundancy and transform domain (Discrete Cosine Transform) based compression for the reduction of spatial redundancy. Temporal prediction techniques with motion compensation are used to exploit the strong temporal correlation of video signals. Temporal prediction is applied with both causal (pure predictive coding) and non-causal (interpolative coding). The remaining signal (prediction error) is further compressed with spatial redundancy reduction (8x8 DCT). The information relative to motion is based on 16x16 blocks and is compressed using variable length codes to achieve maximum efficiency. Most of this chapter is based upon the MPEG standard [12] and publications [19]-[24].

3.3.1 Source coding format

A television image consists of pixels. Each pixel can be characterised by its colour (*chrominance*) and the intensity (*luminance*). The pixels on one row are called a line. All the lines in an image form a picture or frame. By displaying a large number of pictures in succession at a high rate, the impression of moving images is obtained.

In television images also the notion of interlacement is introduced. An interlaced picture/frame is composed of two fields. One field contains the odd lines and the other field the even lines (figure 3.1). The two fields are displayed alternately and they are separated in time by a field period. Between displaying two fields a small blank period called field blanking is inserted. The time interval between to pixels in a line is constant and equals the sampling period. The lines are separated by a so called line blanking.

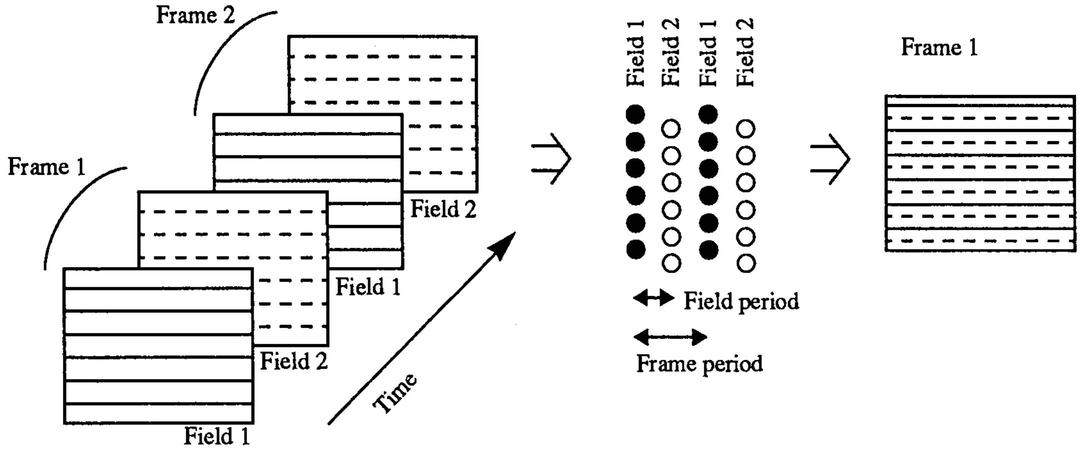


Figure 3.1 Example of an interlaced video sequence

The MPEG-2 specification deals with coding of both progressive and interlaced sequences. The output of the decoding process, for interlaced sequences, consists of a series of fields that are separated in time by a field period. The two fields of a frame may be coded separately (field-pictures) or together as a frame (frame-pictures).

In general, a digitized picture is characterized by the following elements:

- the resolution (the number of lines and the number of pixels per line)
- the format, which is determined by the sample frequency of respectively the luminance and the chrominance, and their spatial position.
- the aspect ratio, which depends on characteristics of the sampling process (the distance between two pixel samples and the distance between two lines) and the resolution.

A frame can be represented by three rectangular matrices of integers; a *luminance* matrix (Y), and two *chrominance* matrices (U, V). The relation between these Y, U and V components and the primary (analogue) Red, Green and Blue signals can be specified in the MPEG stream.

MPEG-2 uses the same colour space (Y, U, V) as the ITU recommendation 601. Three different chrominance formats can be used: 4:2:0, 4:2:2 and 4:4:4 (figure 3.2). In format 4:4:4 all pixels are encoded by three samples. Format 4:2:2 uses only half the sample frequency in horizontal direction for chrominance samples. Format 4:2:0 uses half the sample frequency in vertical direction as well. The respective spatial locations of the luminance and chrominance samples is specified by MPEG and appropriate care has to be taken for the design of the filters.

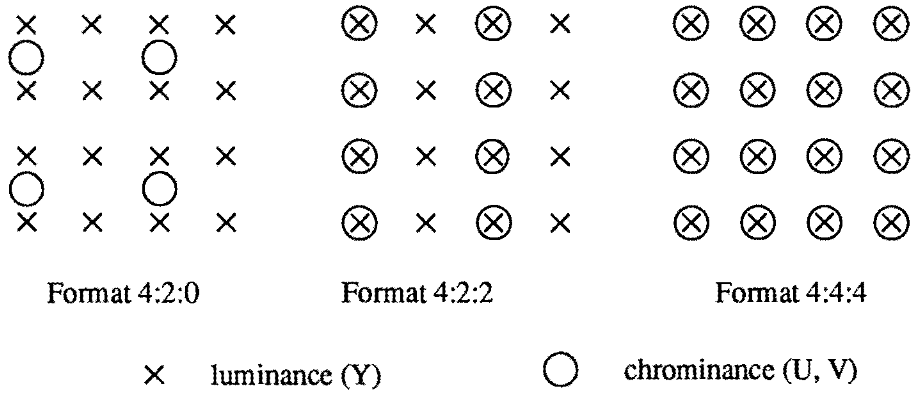


Figure 3.2 Position of luminance and chrominance samples; formats 4:2:0, 4:2:2 and 4:4:4

At Main Level at Main Profile MPEG-2 the source coding format has a frame rate of 25 Hz and the resolution of the pictures is 720x576 which corresponds to PAL (respectively 30 Hz, 720x480, NTSC). The chrominance format is 4:2:0.

3.3.2 Temporal Redundancy Reduction

The MPEG standard takes advantage of temporal redundancy (the fact that much of the information in a picture within a video sequence may be similar to the information in adjacent pictures) to represent some pictures in terms of their differences from a reference picture. This way part of the picture can be predicted using data from reference pictures and only the differential information has to be transmitted (figure 3.3).

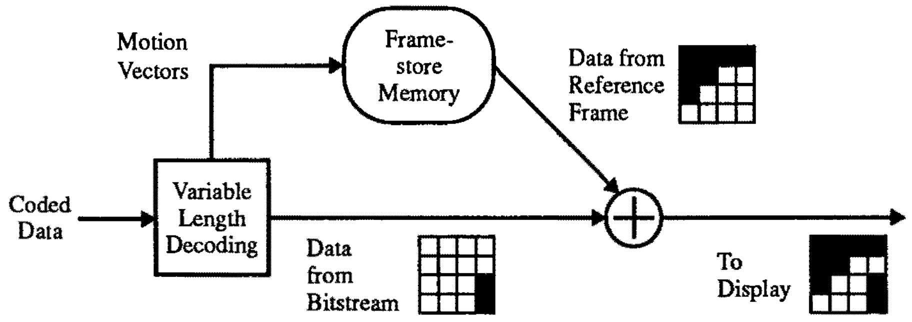


Figure 3.3 Differential coding of picture data

In MPEG-2 three types of pictures are considered: Intra-pictures (I), Predicted pictures (P) and Interpolated pictures (B - for bidirectional prediction). Intra pictures provide access points for random access but only with moderate compression, the complete picture is encoded without reference information. Predictive pictures are coded with reference to a past picture (Intra or Predicted). Bidirectional Pictures provide the highest amount of compression but require both a past and a future reference for prediction, in addition Bidirectional pictures are never used as reference. In all cases when a picture is coded with respect to a reference, motion compensation is used to improve the coding efficiency. The relationship between the three pictures types is illustrated in figure 3.6.

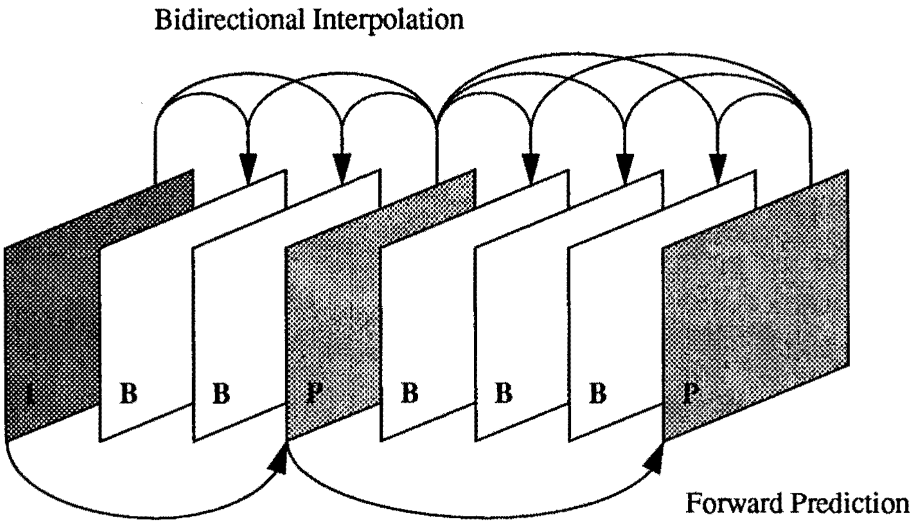


Figure 3.4 Example of temporal picture structure

Before decoding and displaying the Bidirectional pictures the reference pictures, I- or P- pictures must be decoded. Therefore the order of the coded frames in the bitstream is different from the display order. The P-pictures must be sent prior to the associated B-pictures. The MPEG decoder must reorder these frames before displaying them.

Video stream order:

$I_1\ P_4\ B_2\ B_3\ P_8\ B_5\ B_6\ B_7$

Displaying order

$I_1\ B_2\ B_3\ P_4\ B_5\ B_6\ B_7\ P_8$

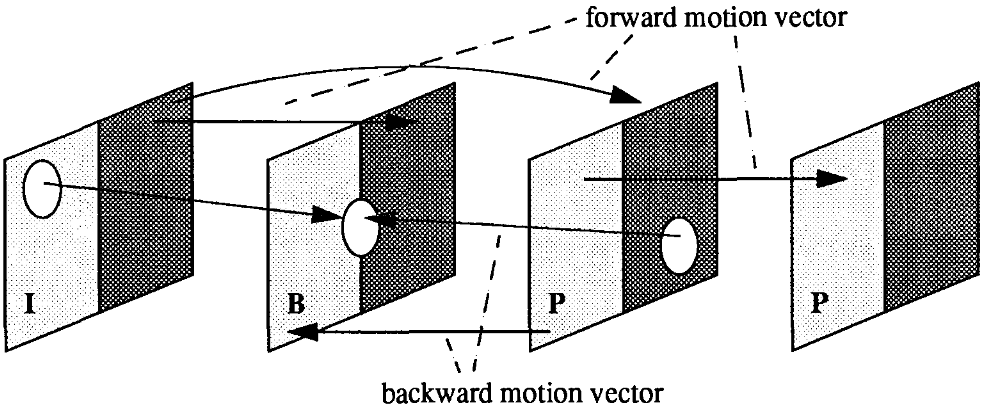
Figure 3.5 Reordering of pictures

Motion Compensation

Motion compensated prediction assumes that "locally" the current picture can be modelled as a translation of the pictures at some previous time. Locally means that the amplitude and the direction of the displacement need not be the same everywhere in the picture. Motion compensation is block based and is performed on blocks of 16x16 pixels, a so called Macroblocks. Each macroblock is coded using one or two motion vectors depending on the prediction mode. The motion vectors refer to picture data of previous or future pictures and result in a predicted macroblock. Together with the differential data from the bitstream the macroblock can be reconstructed. MPEG doesn't specify how theses vectors must be computed however block-matching techniques are likely to be used. In block-matching techniques; the motion vector can be obtained by minimizing a cost function measuring the mismatch between a block and each predicted candidate.

In MPEG, four possible prediction modes are possible for motion compensation:

- Intra-coded, no motion compensation data is used
- Forward prediction, in which the closest prior I- or P-picture serves as the reference
- Backward prediction, in which the closest future I- or P-picture serves as the reference
- Bidirectional prediction, in which two pictures serve as the reference, one being the closest prior I- or P-picture one the closest future I- or P-picture.



I = Intra coded

P = Forward predictive coded

B = Backward predictive coded, Bidirectionally coded

Figure 3.6 The prediction modes in the 3 picture types.

In the predicted pictures the macroblock can be either Intra or predicted, Intra is reserved for use when the temporal prediction process fails for example when new elements appears in the picture. Bidirectional and backward predictive coded macroblocks may only be used in B-pictures. In table 3.3 the permitted prediction modes and related macroblock types for each picture type are listed.

MacroBlock type	I-picture	P-picture	B-picture
Intra coded	X	X	X
Forward predictive coded		X	X
Backward predictive coded			X
Bidirectional coded			X

Table 3.1 Permitted macroblock types in I-, P- and B-pictures

3.3.3 Spatial Redundancy Reduction

Both still image and prediction error signals have a very high spatial redundancy. Because of the block based nature of the motion compensation process, a block based redundancy reduction techniques is used. In MPEG transform coding techniques with a combination of visually weighted scalar quantization and run length coding is applied. The compression consists of three stages (see figure 3.7):

1. Computation of the transform coefficients
2. Quantization of the transform coefficients
3. Conversion of the transform coefficients into {run - amplitude} pairs after reorganisation of the data in a zig zag scanning order

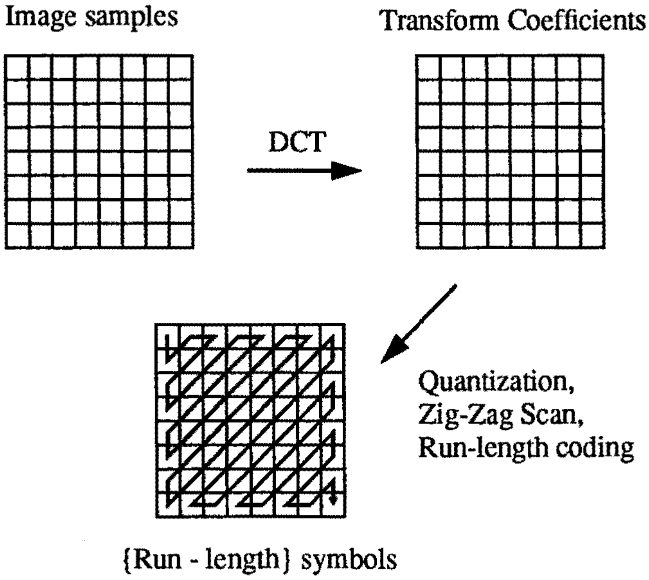


Figure 3.7 Transform Coding, Quantization and Run-length Coding

Discrete Cosine Transform

The discrete cosine transform is selected because it has a certain number of advantages over other transforms. The discrete cosine transform is an orthogonal transform. Orthogonal transforms are filter-bank-oriented (i.e., they have a frequency domain interpretation). The samples in a 8 x 8 block are sufficient to compute 64 transform coefficients and no overlap of blocks is needed. The DCT is the best of the orthogonal transforms with a fast algorithm and very close approximation to the optimal for a large class of images. More about the discrete cosine transform and its fast algorithms can be found in Chapter 5 on page 55.

In MPEG-2 the discrete cosine transform has inputs in the range [-255, 255] because prediction error signals are used and output signals in the range [-2048, 2047], to provide enough accuracy. The accuracy of the inverse transform must meet the ITU-T Recommendation H.261 [17].

Quantization

After DCT the transformed coefficients are quantized. Quantization in MPEG-2 is a key operation since it is the combination of quantization and run-length coding which is responsible for most of the compression. Furthermore through quantization the encoder is able to match its output to a given bitrate.

Subjective perception of quantization error greatly varies with the frequency. It possible to use coarser quantizers for the higher frequencies, so the related coefficients can be encoded using a smaller number of bits at the cost of a smaller accuracy. The lower frequencies and the DC-component can use a finer quantizer and encoded with a larger number of bits. The exact quantization is defined in the so called quantization matrix which defines the quantization for each coefficient. It is possible to define a custom matrix for certain sequences, these can be sent together with the compressed video. If no custom matrix is defined, a default quantization matrix is used.

3.3.4 Entropy coding

In order to further increase the compression, variable length coding is used. A Huffman like table for the DCT coefficients is used to code events corresponding to a pair {Run, Length}.

The Run Length Codes are used to encode the large number of zeroes occurring in the DCT coefficients. This large number of zeroes can be expected as a direct result of the DCT combined with the quantization. In order to group the occurring zeroes of the quantized coefficient matrix the coefficients are ordered using the zig zag scan. The zig zag scan orders the coefficients in descending order of frequency, since most of the zeroes occur in the larger frequencies components. In figure 3.8. an example is given of {Run, Length} encoding.

Only those {Run, Length} codes with a high probability of occurrence are coded with a Variable Length Code. The events with the highest probability are encoded with the shortest codewords. The less likely events are coded with an escape symbol followed by a fixed length code, so as to avoid extremely long codewords.

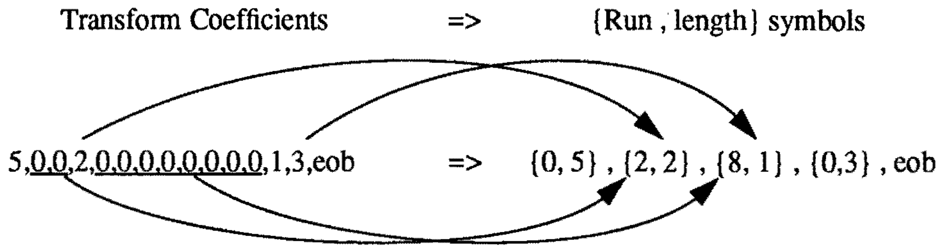

Transform Coefficients
=>
{Run , length} symbols
5,0,0,2,0,0,0,0,0,0,0,0,0,1,3,eob
=>
{0, 5} , {2, 2} , {8, 1} , {0,3} , eob

Figure 3.8 Example of Run Length encoding

3.3.5 Layered structures; Syntax and bitstream

The MPEG-2 syntax specifies a hierarchical structure, starting with a Transport Stream (TS) consisting of packets of 188 bytes each. These packets contain a 4 byte header, and 184 byte payload. The payload contains sections with data, or parts of a Packetized Elementary Stream (PES). Each type of data (audio, video, teletext, etc.) is represented with its own Packet IDentification (PID). This way several MPEG bitstreams from different television programmes and their associated audio can be multiplexed into one Transport Stream. In case of an MPEG-2 video bitstream the data in the Packetized Elementary Stream contains a Video Sequence.

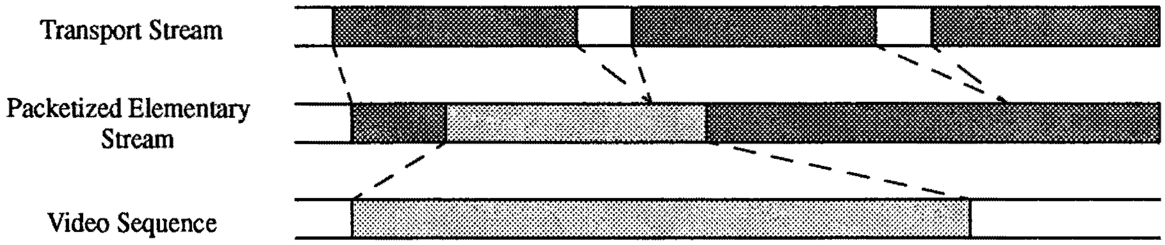


Figure 3.9 MPEG-2 hierarchical structure

The syntax of an MPEG-2 video bitstream contains six layers. Each layer supports a definite function: either a signal processing function (DCT, Motion Compensation) or a logical function (Resynchronization, random access point). The following six layers can be distinguished:

- Sequence layer (Random access unit: Context)
- Group of pictures layer (Random access unit: Video coding)
- Picture layer (Primary coding unit)
- Slice layer (Resynchronization unit)
- Macroblock layer (Motion compensation unit)
- Block layer (DCT unit)

The contents of the various layers is visualised and depicted in figure 3.10. The highest syntactical structure of the coded video bitstream is the video sequence. A video sequence starts with a sequence header followed by a group of pictures header and by one or more coded frames. The order of the coded frames in the coded bitstream is the order in which the decoder processes them but not necessarily the order in which they are displayed. The video sequence is terminated by a sequence_end_code.

Each layer starts with a header which contains the necessary additional information for the decoding process at that level. Followed by the data of the lower levels. The sequence header contains for example the aspect_ratio_information and can contain custom quantization matrices. The Group of Picture header contains a time_code to provide random access. A picture or frame consists of slices which are basically a series of macroblocks. The motion vectors are needed for the reconstruction of the single macroblocks and therefore are contained in the macroblock header.

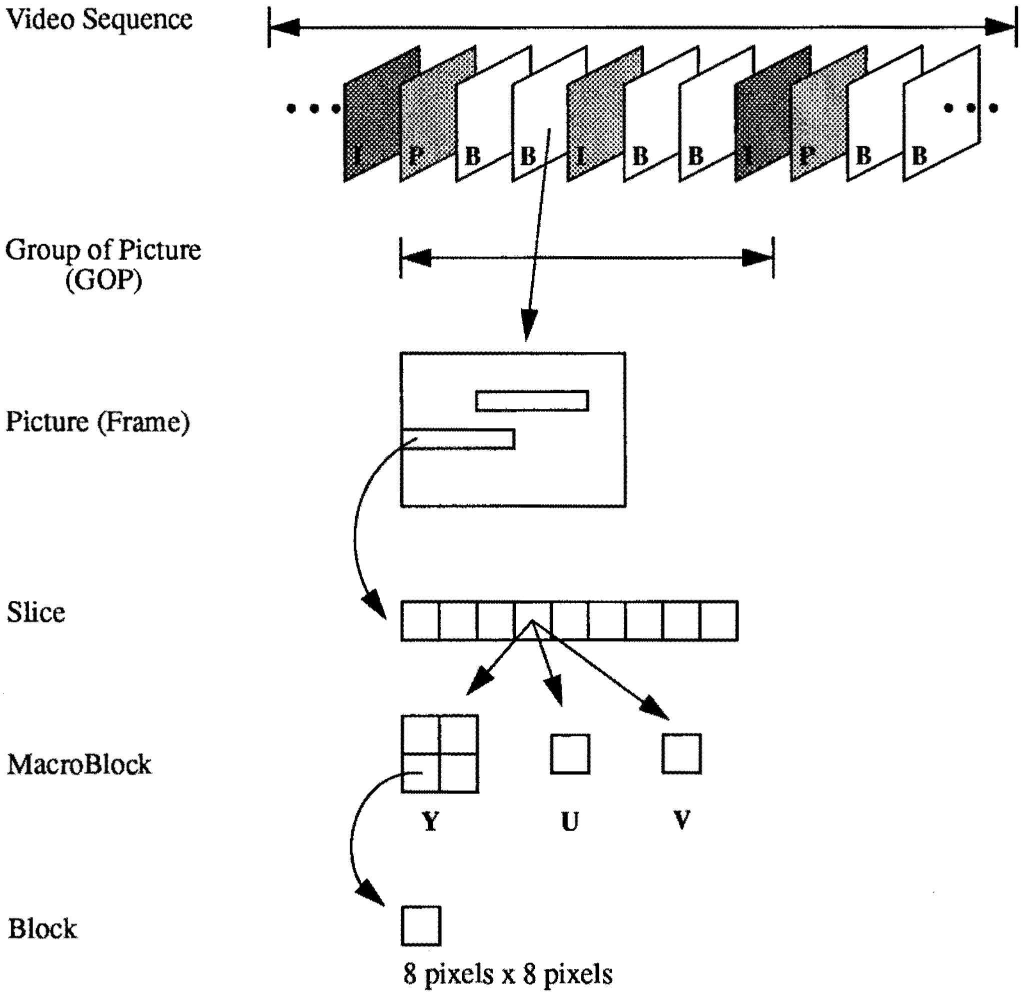
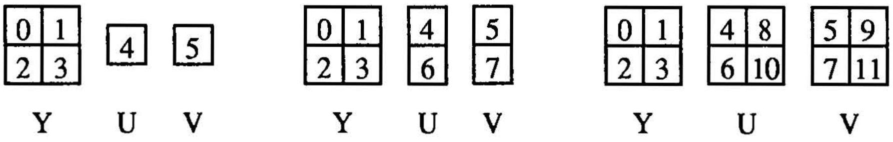


Figure 3.10 MPEG-2 video data hierarchy

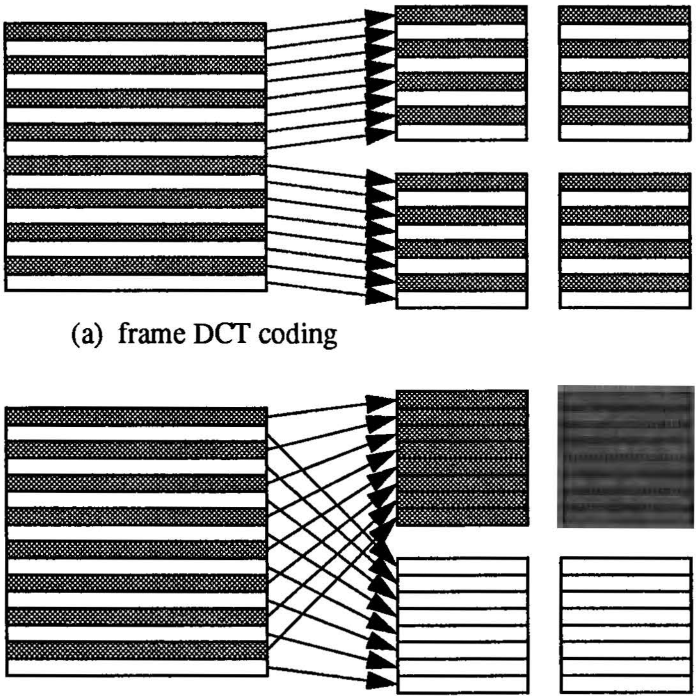
Each macroblock contains 4 luminance blocks. Depending on the macroblock structure a macroblock can contain 2, 4 or 8 chrominance blocks (see figure 3.11). At Main Profile at Main Level macroblock format 4:2:0 is used.



Format 4:2:0 Format 4:2:2 Format 4:4:4

Figure 3.11 Macroblock structures; formats 4:2:0, 4:2:2 and 4:4:4

MPEG-2 can deal with coding of both progressive and interlaced sequences. For interlaced sequences the output of the decoder consists of two fields, which are displayed separated by a field period. The two fields of a frame may be coded separately (field pictures) or alternatively the two fields may be coded together as a frame (frame picture). In frame pictures, macroblocks can be either frame or field DCT coded. In case of frame DCT coding the macroblock is composed of lines from the two field are alternately (figure 3.12a). In case of field DCT coding each block shall be composed from lines from only one of the two fields (figure 3.12b). In field pictures the macroblock only contains lines form one field. In this case the macroblock always are field DCT coded.



(a) frame DCT coding

(b) field DCT coding

Figure 3.12 Luminance macroblock structure

3.4 Decoding process

The MPEG-2 standard defines the decoding process, not a decoder. There are many ways to implement a decoder. The decoder structure of figure 3.13 is a typical decoder structure with a buffer at the input of the decoder. The minimum buffer size necessary to decode the bitstream of a certain level and profile is specified in the MPEG-2 specification.

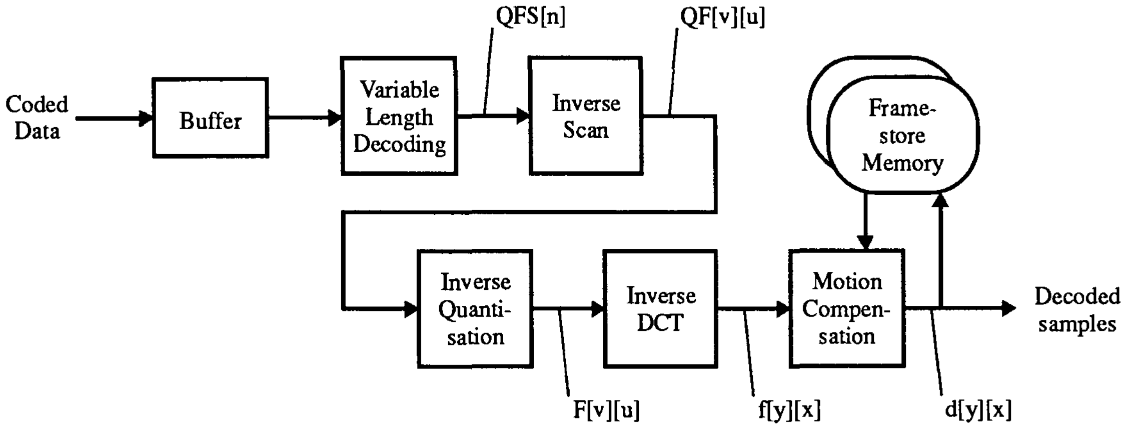


Figure 3.13 Simplified video decoding process

First the bitstream is demultiplexed into overhead information such as motion information, quantizer step size, macroblock type and coded DCT coefficients. The coded DCT coefficients are decoded by the Variable Length Decoding process which includes the Run-Length Decoding. After reordering by the Inverse Scan the quantized DCT coefficients are dequantized and input to the Inverse Cosine Transform (IDCT). The reconstructed blocks from the IDCT is added to the result of the prediction from the reference pictures.

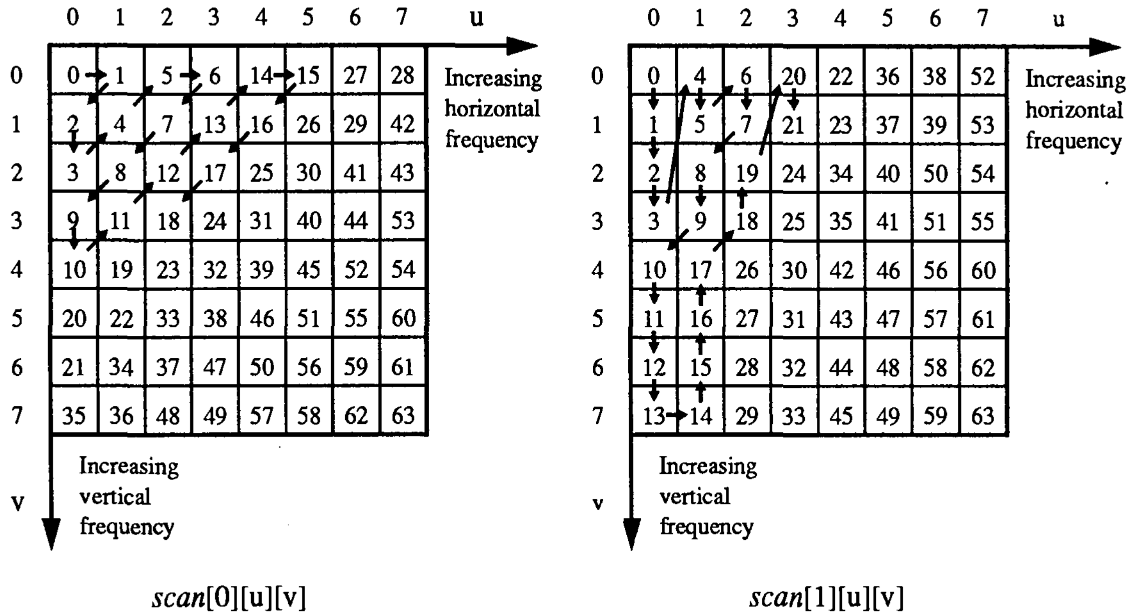
Since the Inverse Scan, the Inverse Quantization and the Inverse Discrete Cosine Transform are used as the design objective in this report, the algorithms defining these processes are discussed below.

Inverse Scan

The Inverse Scan converts the output of the Variable Length decoder, i.e. one-dimensional data QFS[n] with n in the range 0 to 63, into a two dimensional array of coefficients denoted by QF[v][u] (3.1).

```
for (v=0; v<8; v++)
    for (u=0; u<8; u++)
        QF[v][u] = QFS[scan[alternate_scan][v][u]]          (3.1)
```

Two scan patterns are defined. The scan that shall be used is determined by alternate_scan which is encoded in the picture header. Figure 3.14 defines *scan*[alternate_scan][v][u].


u
Increasing horizontal frequency
v
Increasing vertical frequency

v \ u	0	1	2	3	4	5	6	7
0	0	1	5	6	14	15	27	28
1	2	4	7	13	16	26	29	42
2	3	8	12	17	25	30	41	43
3	9	11	18	24	31	40	44	53
4	10	19	23	32	39	45	52	54
5	20	22	33	38	46	51	55	60
6	21	34	37	47	50	56	59	61
7	35	36	48	49	57	58	62	63

scan[0][u][v]

v \ u	0	1	2	3	4	5	6	7
0	0	4	6	20	22	36	38	52
1	1	5	7	21	23	37	39	53
2	2	8	19	24	34	40	50	54
3	3	9	18	25	35	41	51	55
4	10	17	26	30	42	46	56	60
5	11	16	27	31	43	47	57	61
6	12	15	28	32	44	48	58	62
7	13	14	29	33	45	49	59	63

scan[1][u][v]

Figure 3.14 Definition of *scan*[alternate_scan][u][v]

Inverse Quantization

The two-dimensional array of coefficients, QF[v][u], is inverse quantized to produce the reconstructed DCT coefficients. This process is essentially a multiplication by the Quantizer step size. The Quantizer step size is defined in two ways: a weighting matrix is used to modify the step size within a block and a scale factor is used to scale the complete block.

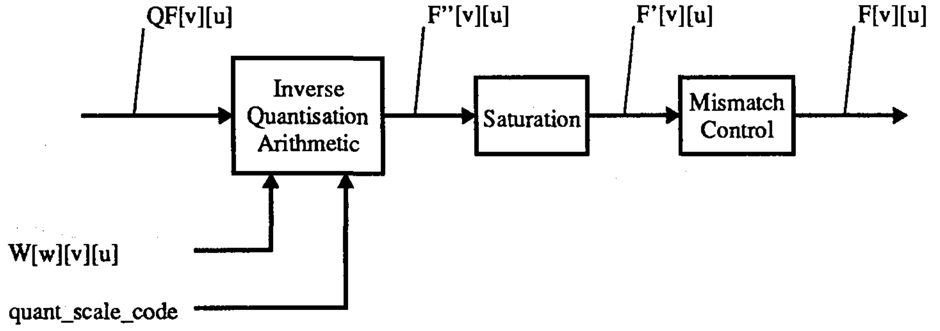


Figure 3.15 Inverse Quantization process

The DC coefficients of intra coded blocks shall be inverse quantized in a different manner to all other coefficients. In intra blocks F"[0][0] shall be obtained by multiplying QF[0][0] by a constant multiplier *intra_dc_mult* which is specified in the bitstream (3.2).

F"[0][0] = intra_dc_mult x QF[0][0] **(3.2)**

All coefficients other than the DC coefficient of an intra block shall be inverse quantized using the following reconstruction formula.

$$F''[v][u] = ((2 \times QF[v][u] + k) \times W[w][v][u] \times \mathit{quantiser\_scale})/32 \qquad \textbf{(3.3)}$$

where:

$$k = \begin{cases} 0 & \text{intra blocks} \\ \mathit{Sign}(QF[v][u]) & \text{non-intra blocks} \end{cases}$$

When 4:2:0 data is used two weighting matrices are used. One is used for intra blocks, W[0][v][u], the other is used for non-intra blocks, W[1][v][u]. When macroblock format 4:2:2 and 4:4:4 is used, two additional weighting matrices are used for chrominance blocks.

After inverse quantization of the resulting coefficients F"[u][v] are saturated to lie in the range [-2048, +2047]. Furthermore mismatch control checks if the sum of all coefficients F'[u][v] is even and if not, a correction shall be made to coefficient F[7][7] which comes down to increasing or decreasing F[7][7] by one.

Inverse Discrete Cosine Transform

Once the DCT coefficients, F[v][u], are reconstructed, the inverse DCT transform defined in the formula below is applied to obtain the inverse transformed values, f[y][x].

$$f[x][y] = \frac{2}{N}\sum_{u=0}^{N-1}\sum_{v=0}^{N-1} C(u)C(v)F[u][v]\cos\frac{(2x+1)u\pi}{2N}\cos\frac{(2y+1)v\pi}{2N} \qquad \textbf{(3.4)}$$

with u,v,x,y = 0,1,2,...N-1

where x,y are coordinates in the sample domain
u,v are coordinates in the transform domain

$$C(u), C(v) = \begin{cases} \frac{1}{\sqrt{2}} & \text{for u,v} = 0 \\ 1 & \text{otherwise} \end{cases}$$

The resulting values must be saturated so that: $-256 \le f[y][x] \le 255$, for all x, y. The N by N inverse discrete transform shall conform to ITU-T Recommendation H.261 [15].

Chapter 4

Video Decoder Architecture Study

This chapter focuses on the architecture of the MPEG-2 video decoder. The design objectives are given. Three architecture proposals are given and discussed to reduce the memory requirements resulting in a cheaper implementation of the video decoder. Furthermore the implications for the Inverse Qantisation, the Inverse Scan and the Inverse Discrete Cosine Transform are discussed.

4.1 Requirements

The design objective is an MPEG-2 video decoder conform the specifications of the MPEG-2 standard. Apart from correct functionality, the design must meet the following requirements:

- clock rate = 27 MHz
- Full macroblock decoding has to be supported for Main Profile at Main Level MPEG-2

The clock rate of 27 MHz is derived from 13.5 MHz which is a common clock rate in video applications. The clock rate of 13.5 MHz originates from the pixel rate in PAL and NTSC video (4.1) and is used in MPEG as well. Due to the high throughput of parts of the MPEG-2 video decoder a clock rate of 27 MHz is chosen which is twice as high.

Video pixel rate:

full PAL resolution x frames / sec = (864 x 625) x 50 = 13.5 MHz

full NTCS resolution x frames / sec = (858 x 525) x 60 = 13.5 MHz **(4.1)**

At Main Level and Main profile MPEG-2 a frame consists of 1620 Macroblocks and the macroblock chrominance format 4:2:0 is used, thus each Macroblock consists of 6 blocks. With a clock rate of

27 MHz the number of clock cycle per macroblock is 666 and the number of clock cycles per block is 111 (4.2).

clock rate = 27 MHz, 25 frames per second

clock cycles per frame = clock rate / frames per second

= 27.000.000 / 25 = 1.080.000

Using line blanking and field blanking:

clock cycles per Macroblock = clock cycles per frame / Macroblocks per frame

1.080.000 / 1620 = 666

clock cycles per block = clock cycles per Macroblock / blocks per Macroblock

= 666 / 6 = 111 **(4.2)**

4.2 Conventional video decoder architecture

The conventional approach for MPEG-2 decoding is depicted in figure 4.1. According to the MPEG-2 specification, which assumes instantanteous decoding, an input buffer of 1.8 Mbit is needed. In practical decoders at least 2.7 Mbit. Furthermore two frame buffers are needed for motion compensation which require 5 Mbit each (4.3). At the end of the decoder a display buffer is needed which requires also 5 Mbit to buffer an entire frame.

Frame buffer size:

Macroblocks per frame x blocks per Macroblock x pixels per block x bits per pixel =

1620 x 6 x 64 x 8 = 5 Mbit **(4.3)**

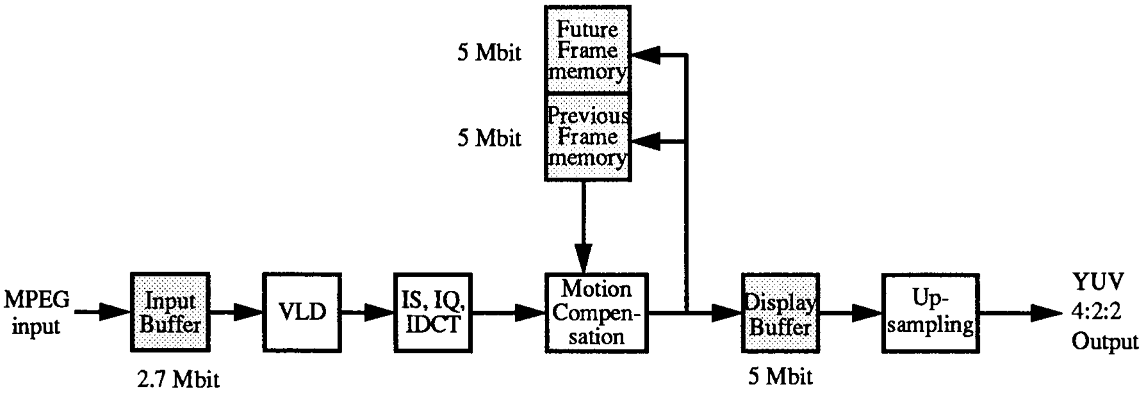


Figure 4.1 Memory requirements in a conventional MPEG-2 video decoder.

In total the required memory is about 17.7 Mbit (3 x 5 + 2.7), which just exceeds the limit of 16 Mbit imposed by the size of available memory elements.

Using a Philips patent the required memory can be reduced to about 16.2 Mbit. This is realised by re-using the memory locations used by a line, immediately after this line has been sent to the display. This method requires only a more complex control of the display buffer.

4.3 Reduced memory video decoder architecture

The display buffer is used to store a frame and bridge the time between decoding and displaying the fields of a frame. It is possible to reduce this display buffer by applying a more complex control of the display buffer. In this section a new approach is presented which makes the display buffer redundant. The only fields that need to be stored in the display buffer are B-fields since the I-frames and P-frames are already stored in the reference buffers. If both B-fields could be decoded separately the display buffer could be omitted. This can be realised in the following way:

1. decode the data from input buffer
2. use the odd field for display and discard the even field
3. decode the data from the input buffer again
4. use the even field for display and discard the odd field

This approach is often refered to as *B-on-the-fly.* It implies however that the data needs to be kept in the input buffer one field period longer, which requires a larger input buffer. With a bitrate of 15Mbit/sec at the input of the input buffer of the video decoder, the buffer must be increased by about 300kbit (15Mbit/sec * 20 msec).

When decoded blocks are displayed on-the-fly it is not desirable to start decoding in the field blanking, because this introduces large additional buffering requirements. When the field blanking is not used and only the line blanking is used the number of clock cycles available per macroblock is 614 and the number of clock cycles per block is 102 (4.4).

Using only line blanking, no field blanking:

clock cycles per Macroblock = clock cycles per frame / Macroblocks per frame

$$= 1.080.000 \times (576 / 625) / 1620 = 614$$

clock cycles per block = clock cycles per Macroblock / blocks per Macroblock

$$= 614 / 6 = 102 \qquad (4.4)$$

The decoder hardware of this new architecture must have the double throughput, since the data from the input buffer must be processed twice. With a clock rate of 27 MHz the number of clock cycles to process a block of 64 coefficients is only 55 (4.5). When the field blanking is not used to prevent additional buffering the number of clock cycles is only 51 (4.6).

Using line blanking and field blanking, clock rate = 27 MHz:

clock cycles per block

$$= 111 \text{ clock cycles} / 2 = 55 \qquad (4.5)$$

Using only line blanking, no field blanking, clock rate = 27 MHz:

clock cycles per (half) block

= 102 clock cycles / 2 = 51 **(4.6)**

One way to realise this throughput is by increasing the clock rate. A second and higher clock rate in the system in not desirable. An alternative is pipelining the decoder or double part of the hardware to process data in parallel. Though this could be very expensive in terms of chip area, it is possible for the IDCT, the IS and the IQ. Unfortunately this solution cannot be applied to the VLD. The VLD doesn't know in advance how many bits are to be taken from the input stream each clock cycle, since the encoded data has a variable length. Only after decoding it knows how many bits may be flushed. If the VLD would know in advance where the next data starts or the next macroblock, two VLDs could be used in parallel. This information can be obtained by using a single speed VLD to preparse the data and store the Macroblock header and the length of the encoded macroblock. In the second stage two VLDs in parallel could be used to increase the throughput. The architecture for this reduced memory video decoder is depicted in figure 4.2.

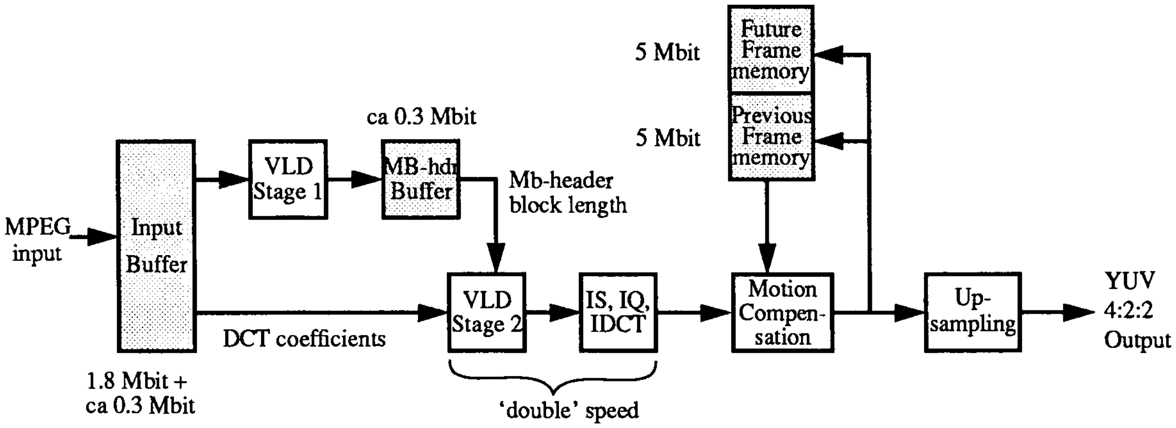


Figure 4.2 Architecture for the reduced memory MPEG-2 video decoder.

The second stage VLD and the IS, IQ and IDCT must process the data at real time to prevent additional buffering. The implication for the second stage VLD is that it must be of a fixed output rate type. Since the header information takes only about 200 bits per macroblock this information can be stored in a small buffer of about 0.3 Mbit (1620 x 200 bits). This way the first stage VLD can be of a fixed input rate type which can be implemented by a small and efficient finite state machine.

After the first stage VLD in this two stage VLD architecture it is know which parts of the data contains header information and which parts contain coefficient data. Now it is possible to send that part of the data containing the coefficients to the second stage VLD. Note that the data from the input buffer must be read twice since this each block is processed twice, which implied an increase of the input buffer by 300 kbit.

4.4 Low cost video decoder architecture

The previous section described a two stage decoding architecture that requires extra decoding hardware, but less memory than conventional architectures. The second stage VLD needs to process 31.104.000 coefficients per second (4.7). If one fixed output rate VLD would be used it would a clock rate of 32 MHz.

coefficients per frame =

2 x 1620 Macroblocks x 6 blocks x 64 coefficients = 1.244.160

coefficients per second =

25 frames per second x 1.244.160 coefficients = 31.104.000 **(4.7)**

Introducing a second clock in the system is not desired, and the alternative of using two parallel VLDs is very expensive. Another solution is to look at the average case. The double speed decoding was only required for B-frames since I- and P-frames were stored in the reference frame buffers anyway. In general B-frames contain much less data than I- and P-frames. The relation between the number of bits in B-frames and the number of bits in I- and P-frames is expected to lie in the range:

B:P:I = 1:2:4 ... 1:3:9

The blocks are encoded in the frequency domain and ordered according to the so called zig zag scan which implies that the highest frequency components are positioned at the end of the blocks. The coefficients containing the highest frequency components are less significant than the lower frequency components and the DC component since they represent the detail information.

At a clock rate of 27 MHz, the average number of bits available to process a block of 64 coefficients was 51 clock cycles when field blanking is not used (4.5). What could be done is the following:

1. use one second stage VLD running at 27 MHz
2. decode 51 coefficients of each block
3. skip the last 13 coefficients, which if present contain the detail information

It is very unlikely that all coefficients of blocks from B-frames are encoded en present in the block. If present the loss of quality by discarting the last coefficients is small since these coefficients contain only detail information. It is possible to skip the last few coefficients because it is known where the next block starts, thanks to the preparser. Even in worst case situation, the B-frames can be processed and displayed on-the-fly as well, albeit with some loss of quality. In general this loss of quality is unlikely and will be small. In case of I- and P-frames all coefficients can be decoded like in the conventional architecture sincethe decoding result is not directly required for display or as reference.

4.5 Overview

Three different architectures have been described in this chapter. Each architectures puts constraints on the VLD and the IDCT, IQ and IS.

1. Conventional video decoder architecture:
    - The display buffer must store two fields per frame which requires 5 Mbit.
    - 1620 Macroblocks per frame must be processed.
    - At a clock rate of 27 MHz, the VLD and the IQ, IS and IDCT require a throughput of one block every 102 clock cycles.
    - VLD must be of fixed output rate type, to prevent buffering between the VLD and the IQ, IS and IDCT
2. Reduced memory video decoder architecture
    - I- and P-pictures are treated conventially. Both fields of B-pictures are processed at real time and displayed on-the-fly. The display buffer is not required, the input buffer must be enlarged to retain the input data one field-period longer.
    - Each macroblock from B-pictures is processed twice, once the lines from the even field is used the next time the lines from the odd field.
    - At a clock rate of 27 MHz, the VLD and the IQ, IS and IDCT require a throughput of one block every 51 clock cycles.
    - Either a VLD running at 32 MHz is required or when using a clock rate of 27 MHz a two stage VLD is required.

      The first stage VLD is used to preparse the input and decode the header information and the block lengths. This VLD can be of the fixed input rate type which can be implemented very efficiently.

      The second stage VLD requires a clock rate of 32 MHz or could consist of two parallel VLDs. The second stage VLD requires a fixed output rate to prevent buffering between the VLD and the IQ, IS and IDCT.
    - A small buffer is required to store the header information and the block lengths.
3. Low cost video decoder architecture:
    - Also requires the two stage VLD.

      Now the second stage consists of one VLD running at 27 MHz. By skipping the least significant coefficients in case of B-frames, the throughput is met at the cost of a small loss in quality.
    - At a clock rate of 27 MHz, the VLD and the IQ, IS and IDCT require a throughput of one block per 51 clock cycles.

The following chapters of this report focus on the implementation of the IQ, IS and the IDCT. In the next chapter the DCT is discussed. Followed by an architecture study and implementation of the IQ, IS and the IDCT. Both for the conventional video decoder architecture and the low cost video decoder architecture. The implementation of the VLD for the conventional architecture and the two stage VLD for the low cost architecture will not be discussed further in this report. A more detailed study of this part of the video decoder can be found in “MPEG-2 as a test case for the high-level synthesis tool Mistral2” by E.J. van Dalen [3].

Chapter 5

Discrete Cosine Transform

In this chapter a short introduction on transform coding is given, in the next section the definition of the one-dimensional and the two-dimensional Discrete Cosine Transform (DCT) is given and discussed. In the following sections the computation of the DCT is discussed and fast algorithms are given to decrease the computation complexity. Finally these algorithms are evaluated and compared with respect to implementation on silicon.

5.1 Transform coding

In the last years, transform coding has become one of the most attractive systems for data compression in storage applications and transmission. In spatial transform coding, the input signal is segmented into small blocks of, mostly 8x8 or 16x16 samples. The transform attempts to remove spatial correlation by converting the block to components which are more or less uncorrelated. By applying the inverse transform the original block can be retrieved, apart form changes due to finite precision calculation. The utility of transformations in image data compression is based on the particular ability of the transform to reduce the image energy of the samples by removing the statistical dependency of the data and concentrate most of the energy into fewer samples. Many algorithms have been designed using transform techniques to compress information. However a major step in data reduction is generally obtained by also applying block quantization, by means of the suppression of irrelevant signal components.

Numerous transforms have been proposed for transform coding, such as the Walsh-Hadamar transform (WHT), the discrete Fourier transform (DFT), the Haar transform (HT), the Slant transform (ST) and the Discrete Cosine Transform (DCT). The performance of these transforms is generally compared with that of the Karhunen-Loève transform (KLT) which is known to be theoretically optimal for first-order Markov stationary random data with respect to the following performance measures: variance distribution, estimation using the mean-square error criterion, and the rate-distortion function [26]. Although the KLT is optimal, there is no general algorithm that enables its fast computation. Ahmed [26] showed that the performance of the DCT compares more closely to the KLT relative to the performances of the DFT, WHT and the HT.

5.2 Discrete Cosine Transform

The Discrete Cosine Transform (DCT) was first introduced in 1974 by Ahmed et al [25]. Primarily applied to real data values, this transform has found wide applications in speech and image processing, data compression, filtering, and other fields.

The one-dimensional DCT (1-D DCT) is carried out on a vector of N samples and is defined as:

$$F(u) = K_{DCT} \cdot C(u) \sum_{x=0}^{N-1} f(x) \cos\frac{(2x+1)u\pi}{2N} \tag{5.1}$$

with $u,x = 0,1,2,...N-1$

$$C(u), C(v) = \begin{cases} \frac{1}{\sqrt{2}} & \text{for } u,v = 0 \\ 1 & \text{otherwise} \end{cases}$$

The one-dimensional Inverse DCT (1-D IDCT) is defined as:

$$f(x) = K_{IDCT} \sum_{u=0}^{N-1} C(u) F(u) \cos\frac{(2x+1)u\pi}{2N} \tag{5.2}$$

In order to prohibit amplitude scaling, the constants K_{DCT} and K_{IDCT} must satisfy equation (5.3). In the MPEG-2 standard [12] these constants are chosen to be $K_{DCT} = K_{IDCT} = \sqrt{2/N}$.

$$K_{DCT} \cdot K_{IDCT} = \frac{2}{N} \tag{5.3}$$

The 2-D DCT is carried out on an array of NxN input samples. The result of the transformation is an array of NxN coefficients representing the frequency contents of the given block. The DCT coefficient value in the upper left corner of the 2-D array represents the energy of the zero-frequency or direct current (dc) term. (For example, if the original image has a constant value, then only the dc term in the transformed domain is nonzero.) The other coefficients are nonzero-frequency coefficients. They correspond to the signal terms with increasing horizontal frequency from left to right and for terms with increasing vertical frequency from top to bottom.

The NxN two-dimensional DCT (2-D DCT) conform the MPEG-2 standard is defined as:

$$F(u,v) = \frac{2}{N} C(u) C(v) \sum_{x=0}^{N-1} \sum_{y=0}^{N-1} f(x,y) \cos\frac{(2x+1)u\pi}{2N} \cos\frac{(2x+1)v\pi}{2N} \tag{5.4}$$

with $u,v,x,y = 0,1,2,...N\text{-}1$

where x,y are coordinates in the sample domain
u,v are coordinates in the transform domain

$$C(u), C(v) = \begin{cases} \frac{1}{\sqrt{2}} & \text{for } u,v = 0 \\ 1 & \text{otherwise} \end{cases}$$

The two dimensional inverse DCT (2-D IDCT) is defined as:

$$f(x,y) = \frac{2}{N} \sum_{u=0}^{N-1} \sum_{v=0}^{N-1} C(u)C(v)F(u,v)\cos\frac{(2x+1)u\pi}{2N}\cos\frac{(2y+1)v\pi}{2N} \qquad (5.5)$$

5.3 DCT computation

Let us now have a close look at the definition of the IDCT (5.2). This equation can be represented by a matrix multiplication of an NxN matrix containing constant coefficients times a vector F of length N. With N=8 this results in the following equation:

$$\begin{bmatrix} f(0) \\ f(1) \\ f(2) \\ f(3) \\ f(4) \\ f(5) \\ f(6) \\ f(7) \end{bmatrix} = \begin{bmatrix} k_{00} & k_{01} & k_{02} & k_{03} & k_{04} & k_{05} & k_{06} & k_{07} \\ k_{10} & k_{11} & k_{12} & k_{13} & k_{14} & k_{15} & k_{16} & k_{17} \\ k_{20} & k_{21} & k_{22} & k_{23} & k_{24} & k_{25} & k_{26} & k_{27} \\ k_{30} & k_{31} & k_{32} & k_{33} & k_{34} & k_{35} & k_{36} & k_{37} \\ k_{40} & k_{41} & k_{42} & k_{43} & k_{44} & k_{45} & k_{46} & k_{47} \\ k_{50} & k_{51} & k_{52} & k_{53} & k_{54} & k_{55} & k_{56} & k_{57} \\ k_{60} & k_{61} & k_{62} & k_{63} & k_{64} & k_{65} & k_{66} & k_{67} \\ k_{70} & k_{71} & k_{72} & k_{73} & k_{74} & k_{75} & k_{76} & k_{77} \end{bmatrix} \cdot \begin{bmatrix} F(0) \\ F(1) \\ F(2) \\ F(3) \\ F(4) \\ F(5) \\ F(6) \\ F(7) \end{bmatrix} \qquad (5.6)$$

With: $k_{ux} = \frac{C(u)}{2}\cos\frac{(2x+1)u\pi}{16}$

After rewriting the coefficients of this matrix multiplication we obtain the matrix of equation (5.7). To compute the 8 point 1-D IDCT defined in (5.2), the input DCT vector has to be multiplicated by the matrix shown below.

$$\begin{bmatrix} f(0) \\ f(1) \\ f(2) \\ f(3) \\ f(4) \\ f(5) \\ f(6) \\ f(7) \end{bmatrix} = \frac{1}{2} \cdot \begin{bmatrix} c_4 & c_1 & c_2 & c_3 & c_4 & c_5 & c_6 & c_7 \\ c_4 & c_3 & c_6 & -c_7 & -c_4 & -c_1 & -c_2 & -c_5 \\ c_4 & c_5 & -c_6 & -c_1 & -c_4 & c_7 & c_2 & c_3 \\ c_4 & c_7 & -c_2 & -c_5 & c_4 & c_3 & -c_6 & -c_1 \\ c_4 & -c_7 & -c_2 & c_5 & c_4 & -c_3 & -c_6 & c_1 \\ c_4 & -c_5 & -c_6 & c_1 & -c_4 & -c_7 & c_2 & -c_3 \\ c_4 & -c_3 & c_6 & c_7 & -c_4 & c_1 & -c_2 & c_5 \\ c_4 & -c_1 & c_2 & -c_3 & c_4 & -c_5 & c_6 & -c_7 \end{bmatrix} \cdot \begin{bmatrix} F(0) \\ F(1) \\ F(2) \\ F(3) \\ F(4) \\ F(5) \\ F(6) \\ F(7) \end{bmatrix} \qquad (5.7)$$

With: $c_n = \cos\frac{n\pi}{16}$

This result can also be written as follows represented by $f = 1/2 \cdot [C] \cdot F$. In other terms a vector f can be transformed by applying the matrix $1/2 \cdot [C]$. The inverse transform can be performed by applying the inverse matrix $1/2 \cdot [C]^{-1}$. Since the discrete cosine transform has a real nonsingular transform matrix, the transform matrix is orthogonal, its inverse is easily obtained as its transpose! The DCT is a so called *orthogonal* transform. The inverse transform can therefore be performed by applying the transposed matrix, $F = 1/2 \cdot [C]^{T} \cdot f$.

In terms of computation complexity, this comes down to a matrix multiplication of an NxN matrix. Since the computation complexity of a multiplication is much larger than the computation complexity of an addition, the complexity of DCT algorithms is often measured in number of multiplications needed. The number of multiplications needed to compute an N-points one-dimensional DCT is N^2 multiplications.

5.4 Two-dimensional DCT by reduction to one-dimensional DCT

According to the definition of the two-dimensional DCT (5.4), the number of multiplications needed to compute an NxN two-dimensional DCT straightforwardly is N^4 multiplications (N^2 picture elements times N^2 multiplications per element).

Instead of a direct computation, the 2-D DCT can also be obtained by decomposing the transform into a series of one dimensional DCTs, in horizontal direction and in vertical direction. This property is valid for any separable transform such as DCT, WHT, DFT, ST, HT, etc. Using this property an (NxN) point 2-D DCT can be implemented by applying first a 1-D DCT on each row followed by a 1-D DCT on each column of the input data matrix or visa versa, as illustrated in figure 5.1. Hence an NxN 2-D DCT is equivalent to 2xN 1-D DCTs. This way the number of multiplications necessary to compute a 2D-DCT can be reduced by a factor N/2, which leads to $2N^3$ multiplications!

Another advantage of this property is the use of the one-dimensional DCT itself. Since the one-dimensional DCT is studied intensity, many fast algorithms are know to compute a one-dimensional DCT with reduced computation complexity.

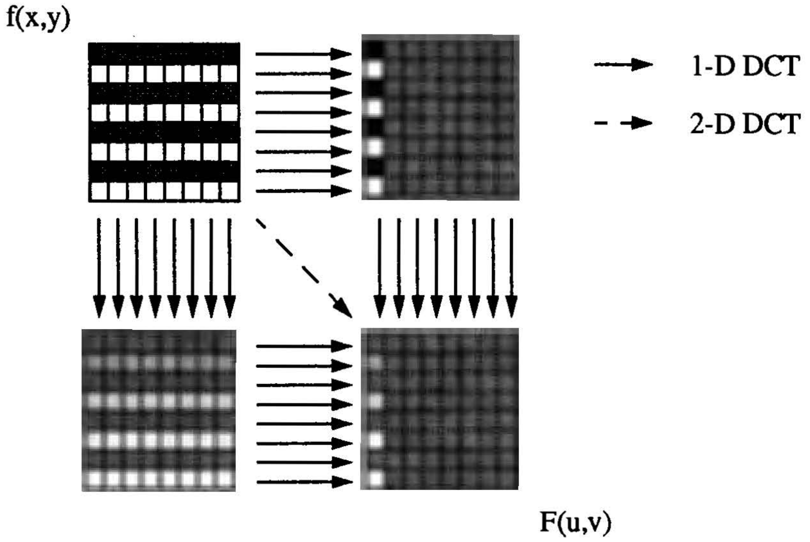


Figure 5.1 Computation of a two dimensional DCT using 2xN one-dimensional DCTs

5.5 Fast DCT algorithms

Due to the computation intensity of the DCT many fast computational algorithms are developed which reduce the complexity of the DCT algorithm by reducing the number of multiplications required. The development of efficient algorithms for the computation of DCT began soon after Ahmed [25]. Initial attempts focused on the computation of the DCT by using the fast Fourier transform (FFT) algorithms, due to its relation to the discrete Fourier transform (DFT) [27]. Fast algorithms can also be obtained by considering the factorization of the DCT matrix. When the components of this factorization are sparse, the decomposition represents a fast algorithm. Since matrix factorization is not unique there exist a lot of fast algorithms [28][29][30]. Other algorithms can be obtained through the computation of other discrete transforms, or through recursive computation [31].

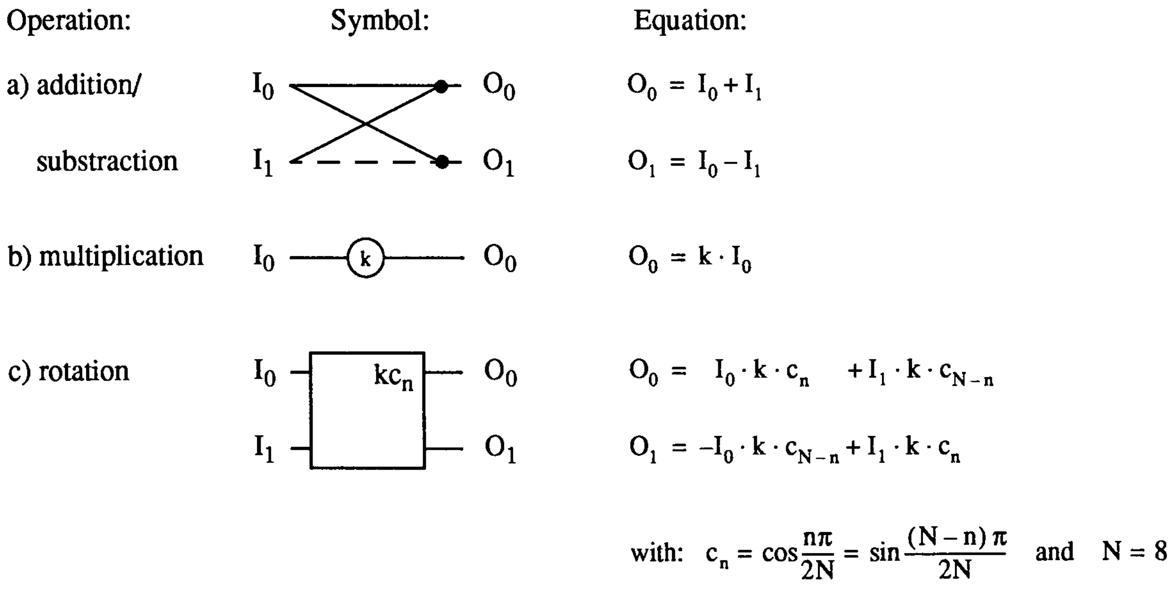
As mentioned before the DCT is a orthogonal transform, the inverse transform is performed by applying the transposed matrix. Therefore if a fast algorithm for the DCT is known, also a fast algorithm for the IDCT with the same computation complexity is known. Since an MPEG decoder contains an 8 points IDCT, the following sections concentrate on fast algorithms for a 8 points IDCT.

A DCT algorithm can be represented by matrices and matrix-multiplication. Another way common way to represent a DCT algorithm is by means of a so called data flow graph. In the next section the notation and symbols used in these data flow graphs is illustrated. In the following sections five algorithms are discussed. First a simple well known algorithm (Butterfly) is discussed to illustrate possible simplification in the algorithm. Next an algorithm is discussed with minimal number of multiplications, and a variation of this algorithm with parallel multiplications and unfortunately an additional multiplication. In the last section these algorithms are compared regarding computation complexity and suitability for implementation with Phideo.

5.5.1 Graphical representation of an algorithm

Another way of representing an algorithm instead of a matrix-multiplication is by means of a data flow graph. The purpose of this section is not to define a formal way of representing an algorithm but is meant to define some symbols and notations to explain and simplify the graphical representation used in the following sections.

Figure 5.2 depicts the symbols together with their arithmetical meaning. The edges represent data signals, the nodes represent operations. There are three different operations, an addition/substraction which is represented by a dot (a), a fixed-coefficient-multiplication which is represented by a circle containing the coefficient (b), and a rotation which itself contains a number of operation (c). Whenever the edge which is connected to an operation is dotted, the negated data value must be used (i.e. an addition becomes a substraction).



$$O_0 = I_0 + I_1$$

$$O_1 = I_0 - I_1$$

$$O_0 = k \cdot I_0$$

$$O_0 = I_0 \cdot k \cdot c_n + I_1 \cdot k \cdot c_{N-n}$$

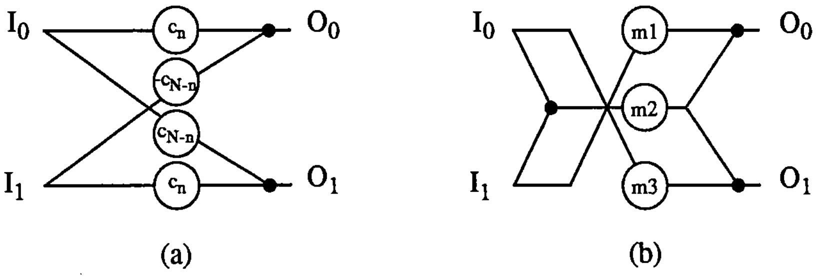
$$O_1 = -I_0 \cdot k \cdot c_{N-n} + I_1 \cdot k \cdot c_n$$

with: $c_n = \cos\frac{n\pi}{2N} = \sin\frac{(N-n)\pi}{2N}$ and $N = 8$

Figure 5.2 Symbols used to represent the algorithm structures

The rotation of figure 5.2 can be computed using 4 multiplications and 2 additions (see figure 5.3a). Another way to compute the rotation is shown in figure 5.3b, this way only 3 multiplications and 3 additions are needed to compute the rotation.

$$\begin{bmatrix} O_0 \\ O_1 \end{bmatrix} = \begin{bmatrix} c_n & -c_{N-n} \\ c_{N-n} & c_n \end{bmatrix} \cdot \begin{bmatrix} I_0 \\ I_1 \end{bmatrix} \qquad \begin{bmatrix} O_0 \\ O_1 \end{bmatrix} = \begin{bmatrix} -(c_{N-n}+c_n) & c_n & 0 \\ 0 & c_n & c_{N-n}-c_n \end{bmatrix} \cdot \begin{bmatrix} I_0 \\ I_0+I_1 \\ I_1 \end{bmatrix}$$



$$m1 = -(c_{N-n}+c_n)$$
$$m2 = c_n$$
$$m3 = c_{N-n}-c_n$$

Figure 5.3 Rotation with 4 multiplications (2 different coefficients) and 2 additions (a), resp. 3 multiplications (3 different coefficients) and 3 additions (b).

A convenient characteristic resulting from the orthogonality property of the discrete cosine transform is the following. The inverse transform can be graphically determined using the data flow graph representation of the transform. 'Reading' the data flow graph in opposite direction (from right-to-left instead of left-to-right) leads to the inverse transform. At each point where a data signal is split in two, a addition/substraction is added. Each existing addition/substraction has to removed and must be seen as a split of the data signal. The fixed-coefficient-multiplications and the rotation remain the same with equal coefficients and parameters.

5.5.2 Butterfly

The straightforward implementation of a 8-point IDCT needs N^2=64 multiplications. Looking more closely to equation (5.7) a lot of symmetry can be found, which can be used to reduce the number of multiplications to compute the algorithm. In figure 5.4 the so called butterfly algorithm is shown which exploits these symmetries. In the graphical representation these symmetry result in so called butterfly structures. In equation (5.8) the sparse matrix factorization is given, which matches the butterfly algorithm. The reduction Note the additional factor 0.5, which can be implemented by a simple shift operation or by scaling the coefficients.

The resulting algorithm contains only 22 multiplications and 28 additions. The multiplications of the odd part of the algorithm consists of four rotations. Using the implementation of figure 5.3b on 61 for the rotations, only 17 multiplications and 33 additions are needed.

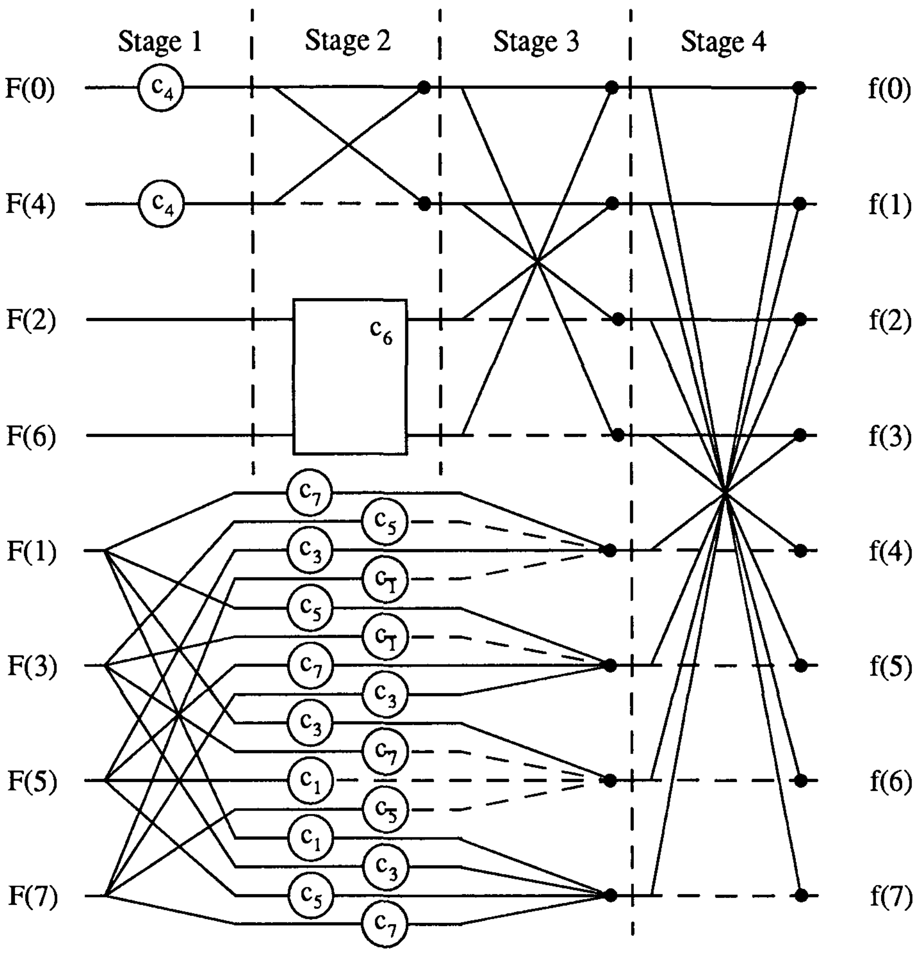


Figure 5.4 8-point fast IDCT butterfly algorithm

$$\begin{bmatrix} F(0) \\ F(4) \\ F(2) \\ F(6) \\ F(1) \\ F(3) \\ F(5) \\ F(7) \end{bmatrix} = \frac{1}{2} \begin{bmatrix} c_4 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & c_4 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & c_6 & c_2 & 0 & 0 & 0 & 0 \\ 0 & 0 & -c_2 & c_6 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & c_7 & c_5 & c_3 & c_1 \\ 0 & 0 & 0 & 0 & -c_5 & -c_1 & -c_7 & c_3 \\ 0 & 0 & 0 & 0 & c_3 & c_7 & -c_1 & c_5 \\ 0 & 0 & 0 & 0 & -c_1 & c_3 & -c_5 & c_7 \end{bmatrix} \cdot \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & -1 & -1 & 1 & 1 & -1 & -1 & 1 \\ 0 & 1 & -1 & 0 & 0 & -1 & 1 & 0 \\ 1 & 0 & 0 & -1 & -1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & -1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & -1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & -1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & -1 \end{bmatrix} \cdot \begin{bmatrix} f(0) \\ f(1) \\ f(2) \\ f(3) \\ f(4) \\ f(5) \\ f(6) \\ f(7) \end{bmatrix} \qquad \textbf{(5.8)}$$

5.5.3 Loeffler

Loeffler[32] presented an algorithm to compute the discrete cosine transform with only 11 multiplications and 29 additions/substractions. Loeffler referred to publications which showed that this number of multiplications is the theoretical lower bound for a 8-point DCT. The algorithm exploits among others the freedom to choose the constants K_{DCT} and K_{IDCT} in the definition of the DCT (5.1)(5.2). In Loeffler's algorithm both constants are defined as follows: $K_{DCT} = K_{IDCT} = 1/2\sqrt{2}$. After forward and inverse transformation the original signal times 8 is obtained. The remaining factor 8 can be easily implemented by right-shifting the transformed signal. Since MPEG-2 defines the constants (5.3) different, this algorithm seems unusable. However using this 1-dimensional inverse transform in both horizontal and vertical to implement the 2-dimensional transform the original signal times 8 (= $(1/2\sqrt{2})^2$) is obtained. After right-shifting the result the original signal is retrieved!

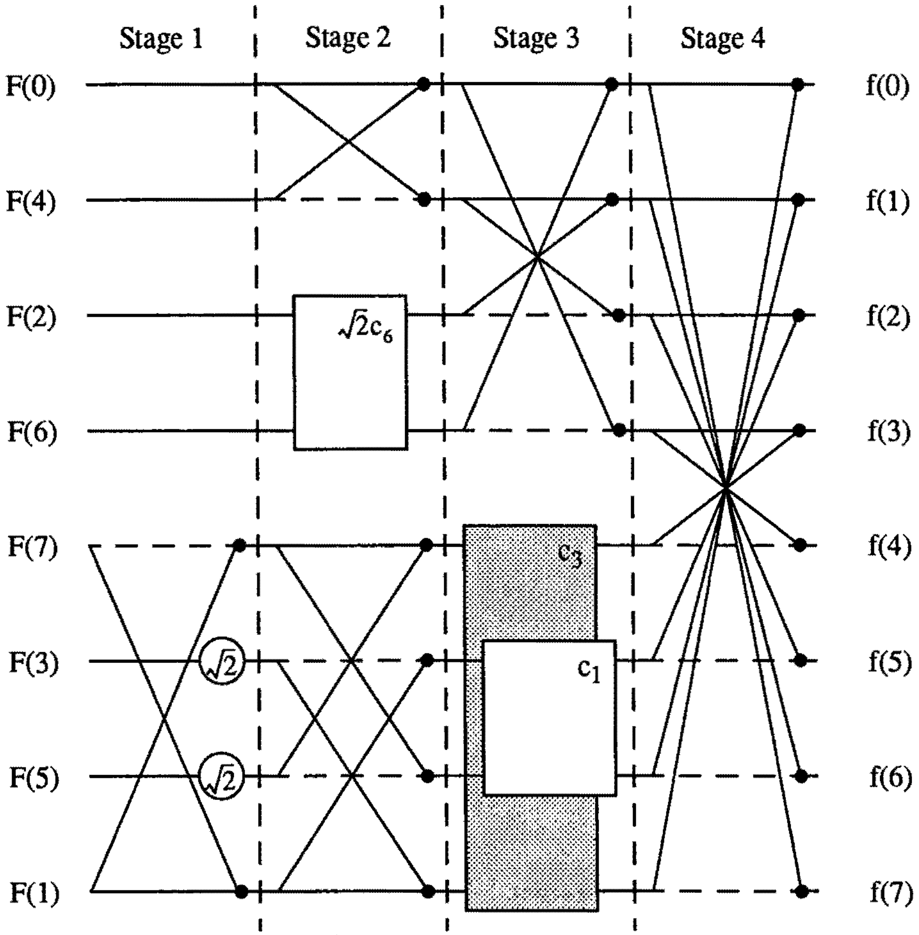


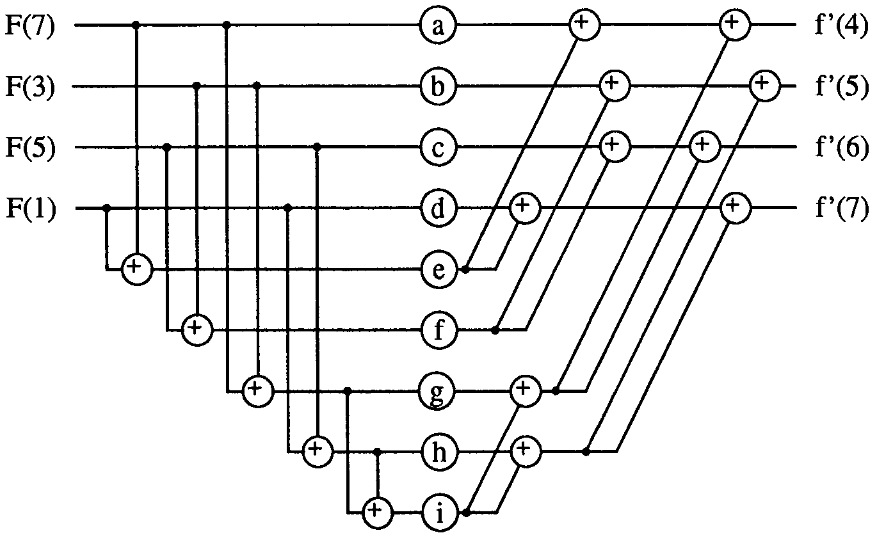
Figure 5.5 8-point 1D-IDCT with 11 multiplications and 29 additions, for symbols see figure 5.2

A withdraw of this algorithm is the length of the critical path in this algorithm which contains two multiplications instead of one. Therefore a variation on this algorithm is given in the next section with parallel multiplications.

$$\begin{bmatrix} F(0)\\ F(4)\\ F(2)\\ F(6)\\ F(7)\\ F(3)\\ F(5)\\ F(1) \end{bmatrix} = \frac{1}{2\sqrt{2}} \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0\\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0\\ 0 & 0 & \sqrt{2}c_6 & \sqrt{2}c_2 & 0 & 0 & 0 & 0\\ 0 & 0 & -\sqrt{2}c_2 & \sqrt{2}c_6 & 0 & 0 & 0 & 0\\ 0 & 0 & 0 & 0 & -c_3-c_5 & c_1+c_7 & c_1-c_7 & c_3-c_5\\ 0 & 0 & 0 & 0 & -\sqrt{2}c_5 & -\sqrt{2}c_1 & -\sqrt{2}c_7 & \sqrt{2}c_3\\ 0 & 0 & 0 & 0 & \sqrt{2}c_3 & \sqrt{2}c_7 & -\sqrt{2}c_1 & \sqrt{2}c_5\\ 0 & 0 & 0 & 0 & c_3-c_5 & c_7-c_1 & c_1+c_7 & c_3+c_5 \end{bmatrix} \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1\\ 1 & -1 & -1 & 1 & 1 & -1 & -1 & 1\\ 0 & 1 & -1 & 0 & 0 & -1 & 1 & 0\\ 1 & 0 & 0 & -1 & -1 & 0 & 0 & 1\\ 0 & 0 & 0 & 1 & -1 & 0 & 0 & 0\\ 0 & 0 & 1 & 0 & 0 & -1 & 0 & 0\\ 0 & 1 & 0 & 0 & 0 & 0 & -1 & 0\\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & -1 \end{bmatrix} \begin{bmatrix} f(0)\\ f(1)\\ f(2)\\ f(3)\\ f(4)\\ f(5)\\ f(6)\\ f(7) \end{bmatrix} \tag{5.9}$$

5.5.4 Loeffler with parallel multiplications

The critical path in Loefler's algorithm contained two multiplications. In this section an algorithm is presented which critical path contains only one multiplication. The algorithm uses a different implementation for the first three stages in the odd part of the former algorithm, which is depicted in figure 5.6. The price for having at most one multiplication per path is, one additional multiplication and 3 additions.



with:

$a = (-c_1 + c_3 + c_5 - c_7)\sqrt{2}$

$b = (\ c_1 + c_3 - c_5 + c_7)\sqrt{2}$

$c = (\ c_1 + c_3 + c_5 - c_7)\sqrt{2}$

$d = (\ c_1 + c_3 - c_5 - c_7)\sqrt{2}$

$e = (-c_3 + c_7)\sqrt{2}$

$f = (-c_1 - c_3)\sqrt{2}$

$g = (-c_3 - c_5)\sqrt{2}$

$h = (-c_3 + c_5)\sqrt{2}$

$i = c_3\sqrt{2}$

and: $c_n = \cos\frac{n\pi}{16}$

Figure 5.6 variation of stage 1 to 3 of the odd part with parallel multiplications

5.6 Implementation

In this chapter a few algorithms to compute the DCT/IDCT have been discussed. It can be seen that the number of operations needed to compute the transform can be reduced by using a fast algorithm. This reduction is an important optimization step in the implementation of an algorithm. Usually High-Level synthesis tools start with a fixed algorithm and don't optimize the algorithm. The number of operations the length of the critical path and the signal width are therefore predefined and must be optimized before High-Level synthesis. An additional profit of the reduction in number of operations is the reduction in power dissipation.

In table 5.1 the algorithms and their characteristics are summarized. Loeffler's algorithm needs the smallest number of operation and seems to be the best candidate for implementation with the High-Level Synthesis tool Phideo. If the length of the critical path would appear to be a bottleneck, the variant with parallel multiplications can be used. In the table also the number of different coefficients of the multiplications is listed. If fixed-coefficient multipliers could be used, the area could be reduced since these are much smaller than normal multiplications. In this case the number of different coefficients is of importance since only multiplications with the same coefficient can be mapped onto the same multiplier. When a implementation with fixed-coefficient multipliers is chosen, the Butterfly algorithm with seven different coefficients could possibly be a better option.

1D-IDCT algorithm	**#Multiplications**	**#Additions**	**#different coefficients**	**Critical Path**
straight-forward	$N^2 = 64$	N(N-1)= 56	7	1 mult, 3 additions
Butterfly (a)	22	28	7	1 mult, 3 additions
Butterfly (b)	17	33	8	1 mult, 4 additions
Loeffler	**11**	**29**	10	2 mult, 4 additions
Loeffler with parallel multiplications	12	32	12	1 mult, 5 additions

Table 5.1 Computation complexity IDCT algorithms

Chapter 6

Implementation of the IDCT with PHIDEO

In the Phideo design path several subtasks are defined (see figure 2.7 on page 21). The following tasks must be performed:

1. PU design;
2. algorithm design
3. pragmas

The design of the PUs include the *clustering* of operations and the definition of the *time shape* of the PU. At this point also the subdivision of the design into hierarchical blocks is made. If a cluster of operations is very large, the user can decide to design this unit in Phideo as well. The algorithm is described in PIF in terms of loops and PUs. After running Phideo, the output of Phideo can be interpreted and the user can define pragmas and restart Phideo until the result is satisfactory.

At first sight is seems that the order of the subtasks is very tight. The opposite is true. It is possible and often desired to apply top-down design in Phideo. Many times it is desired to design the bottlenecks first, without concerning about the implementation of detailed PUs. The user can define a function called unit with an estimated latency and use it as if it were available. If the higher level design seems feasible, a real implementation can be made, small changes in latency or pipelines can be taken care of in some additional iterations of Phideo. It is even possible to define and use complete hierarchical levels without implementing them. The only thing that must be specified is the time shape, defining when signals arrive at the input and when the outputs are generated. This time shape can be based upon an estimation. With the assumption that such kind of PU or hierarchical design can be realised, the higher level design can be evaluated and bottlenecks can be examined.

The implementations in this chapter are made with Phideo version 2.0. At the time the research after Phideo was conducted, the work on this version was stopped, since a new release of Phideo was being developed which contained substantial modifications concerning parametric designs generation. The main functionality of the previous release was intended to be preserved in this newer version. At this point still no proper manual for either of the releases is available and error messages contain a lot of uninteresting and unclear messages which are not documented as well.

Part of the functionality of release 2.0 was added in last stages of the design and is not completely functional. This primarily concerns the final synthesis stage of the design. As a result designs could be made which could not be synthesised completely. For example conditional constructs could be designed up to the memory synthesis. However the controller which had to control these conditional construct was not adapted yet, so that the final VHDL design contained unconnected signals and the design could not be verified completely. These incompletions did not stand in the way of a proper evaluation, and hopefully these problems will be solved in the near future release.

In this chapter first an architecture study is made for the implementation of the Inverse Discrete Cosine Transform (IDCT) the Inverse Quantisation (IQ) and the Inverse Scan (IS). This is done for three different video decoder architectures which are described in Chapter 4. All these architectures impose their own requirements and constraints on the design. In the second part of this chapter the implementation of the IDCT with Phideo is discussed which encloses the three subtasks as described before. Finally the design flow is discussed.

6.1 Architecture study IS, IQ and IDCT

The video decoding process (as described in paragraph 3.4 on page 46) contains an Inverse Scan followed by an Inverse Quantisation and a two-dimensional Inverse DCT. In figure 6.1 a diagram of this part of the video decoding process is depicted. The decoding process is block based.

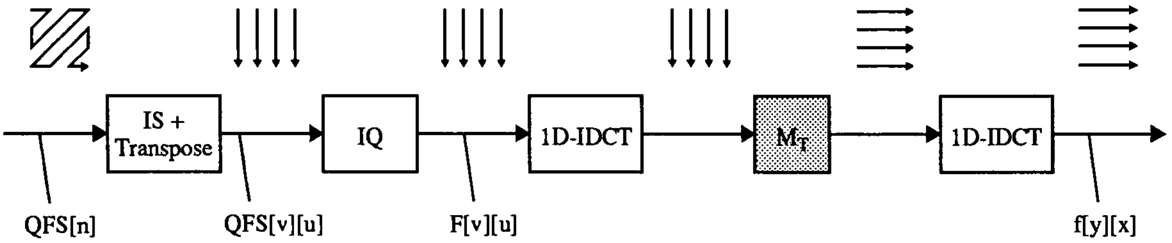


Figure 6.1 Inverse Quantisation, Inverse Scan and Inverse DCT

To reduce the number of operation the decomposition of the two-dimensional IDCT (as described in paragraph 5.4 on page 58) is used, which uses two series of one-dimensional IDCT. First the 1D-

IDCT has to be performed on each column followed by the 1D-IDCT on each row or vice versa. Between the two 1D-IDCT a transposition memory (M_T) is placed to transpose the coefficient matrix.

The data of a block at the output of the variable length decoder is denoted by QFS[n], n is in the range 0 to 63. Two scan patterns are defined to convert the one-dimensional data QFS[n] into a two-dimensional array of coefficients denoted by QF[v][u], u and v both in the range 0 to 7. Since the 1D-IDCT is performed on the columns first, the scan can be combined with the transposition of the two-dimensional array to save an additional transposition memory. In reality the two-dimensional data is transmitted in series column by column or row by row.

Each block which has to be decoded contains 64 coefficients. Each block must be decoded within 102 clock cycles (4.4). With a cycle budget of 102 clock cycles each coefficient must be decoded in 1 clock cycle. The minimum Data Introduction Interval (DII) for the blocks is 64 clock cycles, i.e. every 64 clock cycles the data of a new block can be inserted. Since it is highly likely that the functional units will be pipelined, data of different blocks will be present at the same time.

When two one-dimensional IDCTs are used each 1D-IDCT must perform eight 1D-IDCT, one per row respectively one per column. In that case the data introduction interval of the 1D-IDCT must be DII= 8.

6.1.1 Conventional video decoder architecture

In the decoder described in the previous section two one-dimensional IDCT with DII = 8 were needed. If one 1D-IDCT could be used for both the horizontal as well as the vertical IDCT the hardware requirements could be reduced. Figure 6.2 shows an architecture with one 1D-IDCT. With a data introduction interval of 8 for the 1D-IDCT, the data introduction interval of the 2D-IDCT would be 16 x 8 = 128 which is to large to fit the cycle budget of 102 clock cycles.

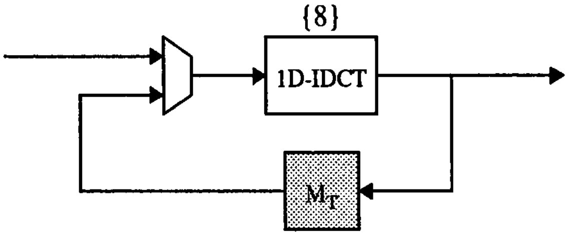


Figure 6.2 2D-IDCT with DII = 128, using one 1D-IDCT with DII = 8

If one 1D-IDCT is to be used it must process two coefficients per clock cycle and the data introduction interval must be DII = 4. This means that two coefficients in parallel must be transferred at the input as well as at the output. For the transposition memory this means that two coefficients must be read at the same time and another two written at the same time. If only 2-port memories are available, the memory must be split to increase the memory bandwidth. In figure 6.3 the diagram of such a two-dimensional IDCT using only one one-dimensional IDCT is depicted. The minimum data introduction interval is 64. Since 102 clock cycles are available wait cycles are introduced.

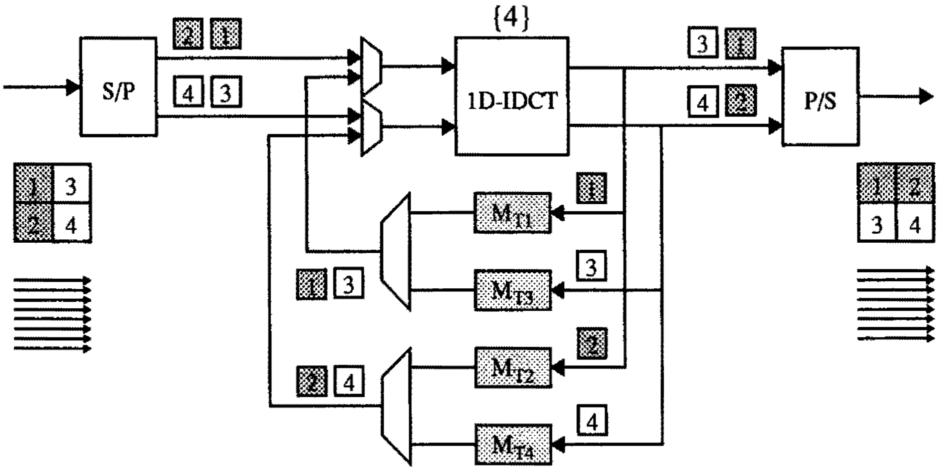


Figure 6.3 2D-IDCT with DII = 64, using one 1D-IDCT with DII = 4

The block is divided into four parts. The top four rows of the block are represented by two coloured parts containing number 1 and 2 (each numbered part contains 64/4=16 coefficients). Note that the two-dimensional array at the input is already transposed, the arrows indicate the transmission order of the data in the blocks. First the columns are processed by the 1D-IDCT. The first four coefficients of each column (part 1 and 2) are transferred to the top-input of the 1D-IDCT and the other four coefficients of each column (part 3 and 4) are transferred to the top-input of the 1D-IDCT. The output of the 1D-IDCT is written in the transposition memories M_{T1} - M_{T4}. After transposition the rows are processed by the 1D-IDCT, the first coefficients (part 1 and 3) and the last four coefficients of each row (part 2 and 4) are transferred to the 1D-IDCT.

At the output of the 1D-IDCT before transposition, part 1 and 3 are produced at the same time and therefore must be written in different memories. The same holds for part 2 and 4. At the input of the 1D-IDCT after transposition, part 1 and 2 (and part 2 and 4) are consumed at the same time and therefore must be read from different memories. This results in four different memories, each containing the coefficients of one of the parts 1 to 4 and named M_{T1} - M_{T4}.

The Serial/Parallel converter and the Parallel/Serial converter are needed to reorder and buffer the input and output data and can be implemented by a small memory. The next section describes the consequenses for the Inverse Quantisation and the Inverse Scan. To save memory the Serial/Parallel converter at the input of the IDCT can be combined with the Inverse Scan. This can be realised by reversing the order of the Inverse Quantisation and the Inverse Scan (figure 6.4).

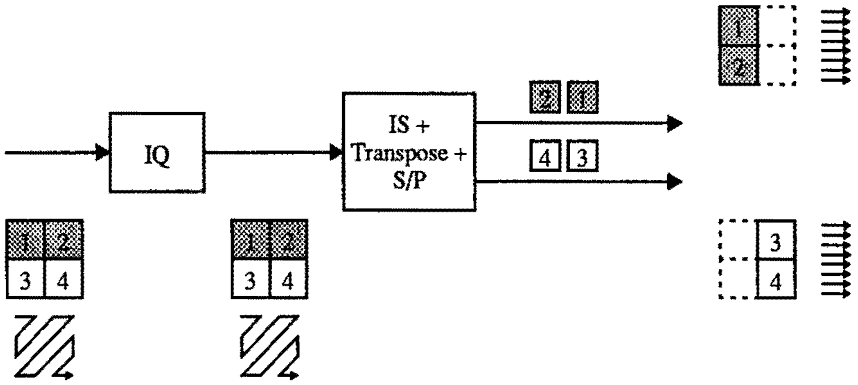


Figure 6.4 Inverse Quantisation and Inverse Scan in reverse order, DII = 64

Inverse Quantisation, Inverse Scan

This section describes the consequences for the Inverse Scan and the Inverse Quantisation when these are applied in reversed in order and combined with the Serial/Parallel buffer as described in the previous section.

Since the Inverse Scan must produce two coefficients each clock cycle, the Inverse Scan requires two memories (see figure 6.5). One memory (M1) is used to store the top four rows and one (M2) to store the bottom four rows. By writing the coefficients into the memory in a different order then reading them, the Inverse Scan is performed. At the same time the block transposition is performed.

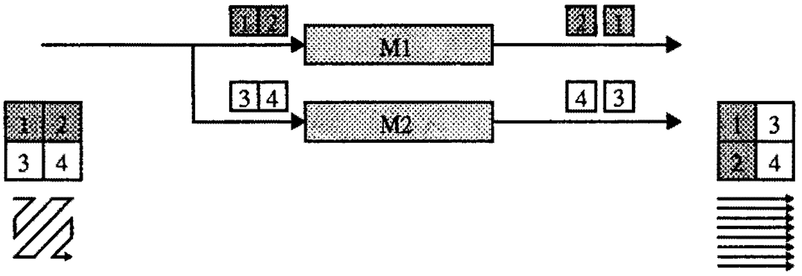


Figure 6.5 Inverse Scan, Transpose and Serial-Parallel converter, DII = 64

In the Inverse Quantisation (see figure 6.6), each coefficient has to be multiplicated by an element of the quantisation matrix. When 4:2:0 data is used, two quantisation matrices are used. One for intra blocks and one for non-intra blocks. Each matrix has a default set of values which may be overwritten by downloading a user defined matrix. When the matrices are downloaded they are encoded in the bitstream in a scan order that is identical to one that is used for coefficients. For matrix downloading the scan defined in figure 3.14 (e.g. scan[0][v][u]) is always used.

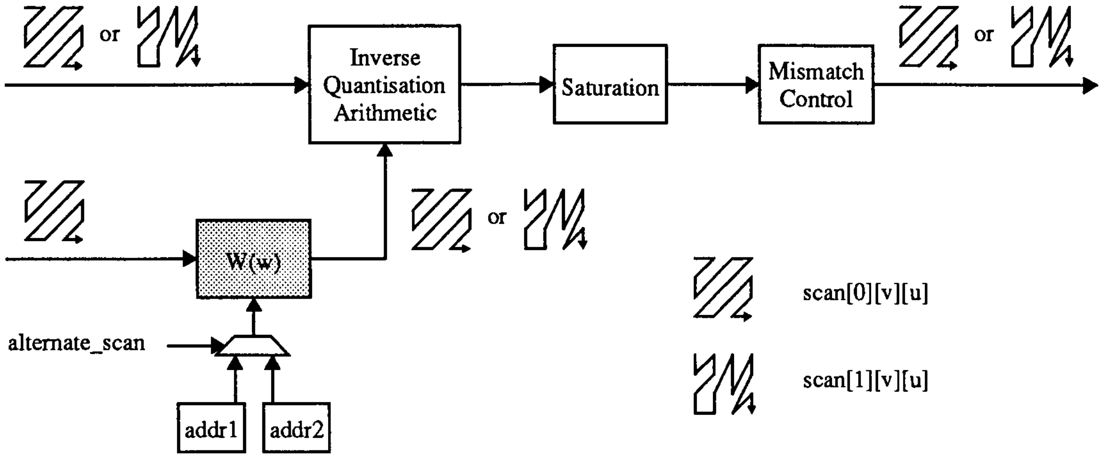


Figure 6.6 Inverse Quantisation when placed before Inverse Scan, DII=64

When the coefficients are encoded by the same scan order the weighting matrix can be written and read in the same order they arrive. In this case the address generator could be a simple counter (i.e. addr1). When the other scan order is used for the coefficients (i.e. scan[1][v][u], the coefficients have to be read in a different order. In this case a second address generator is needed. Since the incoming and outgoing order are completely different a look-up table has to be used.

6.1.2 Low cost video decoder architecture

In the low cost video decoder architecture the target was to reduce the output buffer. This could be realised by decoding each block from B-pictures twice.The first time only the lines of the odd field are used and displayed directly and the second time (one field later) the lines of the even field are used. In this new architecture the number of clock cycles available to decode a block is 51 (6.1).

Using only line blanking, no field blanking:

clock cycles per (half) block

= 102 clock cycles / 2 = 51 **(6.1)**

Since the two-dimensional IDCT is decomposed into two series of one-dimensional IDCTs, it is not necessary to compute two complete IDCTs. Conventionally eight 1D-IDCTs has to be performed on the columns and eight on the rows. In the new architecture only half the rows have to be decoded. This means that only 8+4=12 1D-IDCT have to performed in the available clock cycles. Therefore the number of clock cycles per 1D-IDCT is 4 (6.2).

clock cycles per 1D-IDCT = clock cycles per (half) block / 1D-IDCTs per half block

= 51 / 12 = 4 **(6.2)**

Since only have the rows have to be decoded, each column has to be stored half in the transposition memory. Therefore only two transposition memories are needed ($M_{T1/3}$ and $M_{T2/4}$). In figure 6.7 the new architecture is depicted. The minimal data introduction interval of this 2D-IDCT is 48. Since 51 clock cycles per block are available, three wait cycles are introduced.

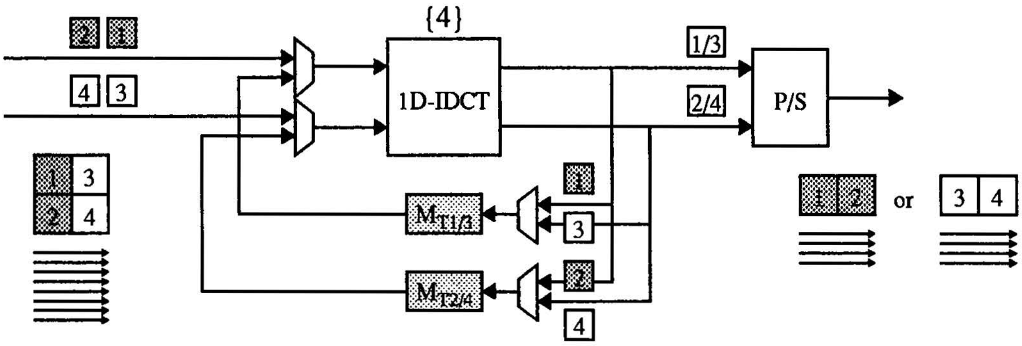
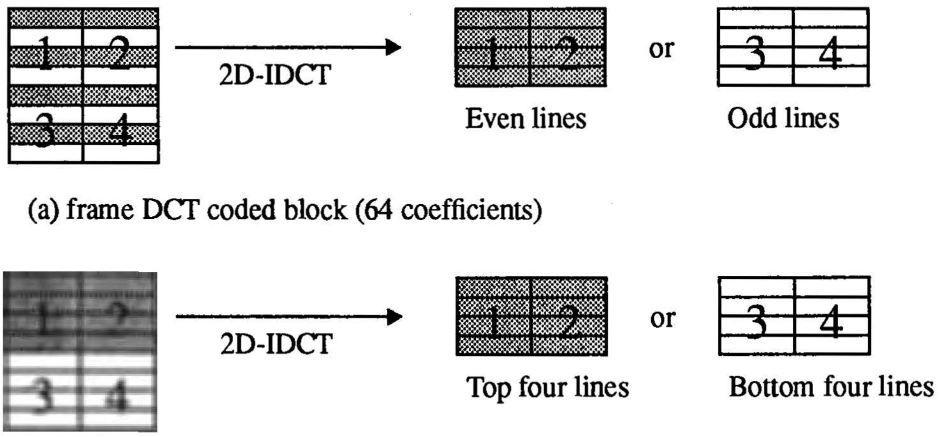


Figure 6.7 New Architecture 2D-IDCT with DII = 51, using one 1D-IDCT with DII = 4

To decode a block from a I- or P-picture the data can be read from the memory in the Inverse Scan twice, and the block can be decoded in two parts. A withdraw of this method is that the one-dimensional IDCT over the columns are computed twice. Since the hardware is available this only results in some additional power dissipation. If power dissipation has to optimized, this can be prevented by introducing two additional transposition memories like in the conventional decoder architecture (figure 6.3).

A small problem remains to be solved. After the series of 1D-IDCTs over the columns the proper lines have to be selected to be transposed. There are two different luminance macroblock structures (paragraph 3.3.5 on page 43), frame DCT coding and field DCT coding. When frame DCT coding is used (figure 6.8a) a block contains lines of even fields alternated with lines of odd fields.When field DCT coding is used a block contains only data form one field (figure 6.8b).



(a) frame DCT coded block (64 coefficients)

(b) field DCT coded block

Figure 6.8 2D-IDCT decoding for the new video decoder architecture

When the block is field DCT coded parts 1 and 2 (respectively parts 3 and 4) must be stored. This selection can easily be made with a multiplexer (figure 6.7).

When frame DCT coding is used this selection is more complex. Unfortunately the coefficients of the even lines arrive two-by-two simultaneous, because both part 1 and part 2 contain even lines (the same for the odd lines coefficients). In this case the even lines coefficients must be delayed and written during the arrival of the odd lines coefficients and vice versa.

Inverse Quantisation, Inverse Scan

The Inverse Quantisation and the Inverse Scan in the low cost architecture can be implemented similar to the conventional architecture except for the smaller data introduction interval (figure 6.9).

In case of blocks from B-pictures, the data introduction interval in the low cost architecture is 51. Since the number of coefficients is 64 either more than one coefficient in parallel has to be processed or a number of coefficients have to be omitted.

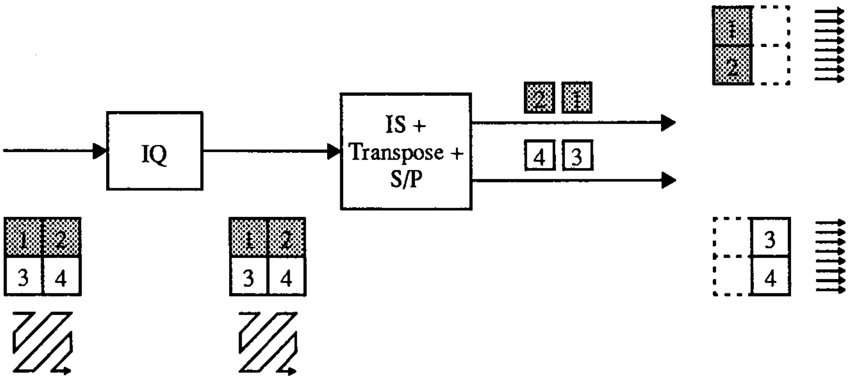


Figure 6.9 Inverse Quantisation and Inverse Scan in reverse order, DII = 51/102

In case of blocks from I- and P-pictures both halves of one block can be computed in sequence. Therefore the Inverse Quantisation and Inverse Scan have to be computed only once for this block and they may use 64 clock cycles, with a data introduction interval of 102 clock cycles instead of 51. Since the IDCT decodes only half a block each time, the data must be read from the memory of the Inverse Scan twice.

In figure 6.10 an implementation of the Inverse Scan is presented which meets these requirements. In case of blocks from I- and P-pictures the coefficients are written into the memories in 64 clock cycles. This data is read from these memories twice and send to the IDCT, once to compute the lines from the odd field and once for the lines of the even field. This is repeated every 102 clock cycles, DII = 102.

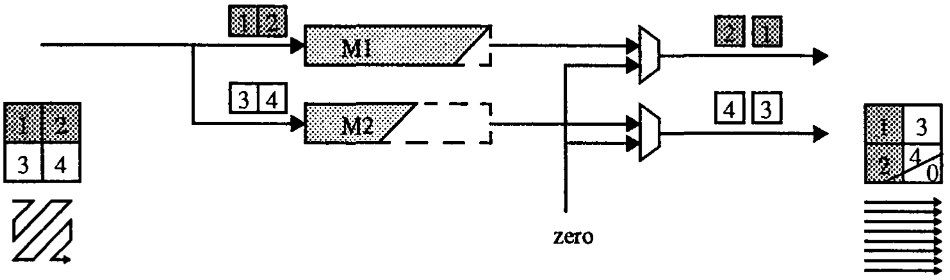


Figure 6.10 Inverse Scan, Transpose and Serial-Parallel converter, DII = 51/102

In case of B-pictures only 51 coefficients are written into the memories. The missing coefficients are stuffed with zeroes. This process is repeated every 51 clock cycles, DII = 51.

The Inverse Quantisation unit is equal to the one from the conventional architecture (figure 6.6). In case of I- and P-picture 64 coefficients are computed with a data introduction interval of 102. In case of B-pictures only 51 coefficients are computed with a data introduction interval of 51.

6.1.3 Reduced memory video decoder architecture

The reduced memory video decoder architecture imposes a number of performance requirements on the Variable Length Decoding and Inverse Scan, Inverse Quantisation and the IDCT. The low cost video decoder architecture is a low cost solution for these requirements at the cost of a small quality loss. If this quality loss is not acceptable, the reduced memory video decoder architecture can be used. This section discusses the implications for the IDCT, the Inverse Scan and the Inverse Quantisation.

The reduced memory video decoder architecture must be able to decode half a block in 51 clockcycles similar to the requirements for the low cost video decoder (6.1). The IDCT of the low cost video decoder as described in the previous section already meets these requirements.

The back-end of the Inverse Scan of the low cost video decoder is able to produce 64 coefficients in 51 clockcycles as described in the previous section. The bottleneck for the Inverse Scan is the input. If the Inverse Scan must be able to decode a complete block of 64 coefficients the input must arrive in parallel. If the Inverse Quantisation would produce two coefficients in parallel this Inverse Scan satisfies the requirements.

The only real extension has to be made in the Inverse Quantisation. This module would require additional hardware to meet the higher throughput requirements.

6.2 The implementation of the IDCT with Phideo

6.2.1 Arithmetic precision

MPEG-2 specifies most of the signal ranges and the number of bits to represent these signals. However the internal signals of the IDCT are not specified. The accuracy of the IDCT is defined in terms of mean, mean square and peak error rates. The IEEE standard specifications for the implementations of 8 x 8 Inverse Discrete Cosine Transforms[16] describes an exact procedure to generate test sequences and to measure these criteria. The high accuracy requirements are needed to control the accumulation of mismatch errors, since every macroblock can be coded 132 times as predictive macroblocks. Macroblocks in B-pictures are excluded from the summmation because they do not lead to the accumulation of errors.

After superficial testing of to the implementation of the Inverse Discrete Cosine Transform using the transform decomposition (section 5.4) and Loeffler's algorithm (section 5.5.3), an estimation of the signal widths is made. The DCT coefficients in the algorithm are to be represented by 17 bits, from which 3 to represent fractions. The coefficients in the algorithm are to be represented by 14 bits. The inputs of the IDCT must therefore be scaled by a factor 8 before inverse transformation. In addition the multiplier and adders must apply proper rounding on temporary results. Saturation of results should not be necessary since signal widths should be large enough to prohibit overflow of signals. In the VHDL code an error message is generated in case of overflow for testing purposes.

Instead of rounding all temporary results, it might be cheaper to postpone the rounding of the results from additions and substractions and increase the number of bits of these signals by two bits (at most 3 additions or substractions in a row). Now a separate rounding unit can be used to round the final results and the temporary results, which are used as an input to the multiplications.

6.2.2 PUs design

The implementation of a PU has to be done manually. The PU must be described at Register Transfer Level in VHDL. Since this part of the design process remains the same it is only discussed briefly. In appendix C some examples of PUs which were used to implement the IDCT, are presented.

For the implementation of the one-dimensional IDCT two main PUs are needed, i.e. a multiplier and an adder. Both PUs must contain a proper rounding, which comes down to an extra addition before truncating. For the implementation of the multiplier a generic parametric multiplier with Booth encoding and carry save Wallace tree construction is used. For the interested reader, more about Booth encoding and Wallace trees can be found in [36] and [37].

6.2.3 One-dimensional IDCT

For the implementation of the two-dimensional IDCT in the conventional and the low cost video decoder, two types of one-dimensional IDCT were needed. The conventional video decoder needed one one-dimensional IDCT with a data introduction interval of DII = 8. The low cost video decoder architecture needed a one-dimensional IDCT with DII = 4, which requires parallel inputs.

1D-IDCT Loeffler, DII=8

Before implementing an algorithm, the algorithm must be optimized. Phideo doesn't optimise the algorithm but only implements the given algorithm. In section 5.5 Loeffler's algorithm is described which only requires 11 multiplications. Since resource requirements are an important criterion, this algorithm was chosen to be implemented. First the algorithm was described in terms of multiplications and additions/substractions. The data introduction interval was specified by choosing the global frame time. Furthermore the time shape of the design had to be specified. To reduce the bandwidth requirements for the memories in the global design, the inputs must arrive in serial. Since 8 cycles are available for 8 inputs this was possible. The implementation required 4 adders/substractors, 2 multipliers and 59 registers. An example of the PIF description of a one-dimensional IDCT is given in appendix D.

An alternative implementation was the use of fixed-coefficient multipliers. Since the multiplicand is known in advance, this multiplier can be implemented very efficiently. The use of fixed-coefficient multipliers could be defined in Phideo by enforcing each different multiplication to be mapped upon a unique multiplier. This could be done by means of an assign pragma. Afterwards these allocated multipliers can be replaced by fixed-coefficient multipliers. After another iteration of Phideo, a new schedule was generated which required 10 fixed-coefficient multipliers. A fixed-coefficient multiplier is about 4 times smaller than a general multiplier depending on the fixed-coefficient. Since two general purpose multipliers will probably smaller than 10 fixed-coefficient multipliers, the implementation with general multipliers will probably result in a smaller design.

1D-IDCT Loeffler, DII=4

For the implementation of the one-dimensional IDCT with a data introduction interval of 4, the same PIF algorithm could be used. The only difference was the implementation of the inputs and the outputs. Since only 4 clock cycles were available to insert 8 inputs the inputs are imported in serial over two parallel input ports as described in section 6.1.1. The same applies to the outputs. After one iteration of Phideo, a new schedule was generated. The implementation of that schedule required 3 multipliers, 8 adders/substractors and 47 registers. When using fixed-coefficient multiplications, 10 fixed-coefficient multipliers, 8 adders/substractors and 39 registers were needed. Using the fixed-coefficient multipliers probably results in a smaller implementation.

1D-IDCT butterfly, DII=4

When writing this report, another solution was found. The butterfly algorithm requires 22 multiplications and 28 additions (see section 5.6). Furthermore, the butterfly algorithm uses only 7 coefficients for the multiplications. All multiplications with the same coefficient are used maximally 4 times. If this algorithm is implemented with fixed-coefficient multipliers, still only 7 multipliers are needed. The required number of adders/substractors is 7 or 8 (28 additions/substractions per 4 cycles is at least 7 additions/substractions in parallel). The number of registers in not expected to differ a lot from the previous designs. Though the number of operations and in special multiplications is not minimal, the overall solution will be smaller since resource utilization will be better. Note that the optimization of the number of operation is *not the only* important criterion!

Gate count

The size of a design measured in micron is not a very good criterion to compare design sizes, since it highly depends on the applied fabrication technology. A better measure is a gate count, this can be found by dividing the total chip area in a certain technology by the area of a NAND-port in that technology. This way a gate count in NAND equivalents is obtained which is technology independent.

In table 6.1 an estimation of the size of different implementation for the 1D-IDCT is given as well as the estimation of the size of an addition/substraction and multiplier and a fixed-coefficient multipier. At this point and with release 2.0, Phideo's designs were not fully synthesizable yet, for example the memories and registers were not linked to a library in the RT-level VHDL. Only the size of the memories in words and a model describing the behaviour in terms of access times and number of read/write ports, were available. This made it very hard to obtain an accurate gate count. The sizes of the memories are estimated by measuring the equivalent amount of memory in flip-flops which should give an upper bound estimation.

Design	DII	resources & gate-count	
adder/substractor	{1}	17x17 = 300 gates	300
multiplier	{1}	17x14 = 2700 gates	2700
fixed-coeff-mult	{1}	17x14 = 2700/4 ?	700 ?
1D-IDCT11_8 Loeffler	{8}	Ctrl, AG, Mux = 2650 59 Registers = 3850 2 mult = 5400 4 add/sub = 1200	13100
1D-IDCT11_4 Loeffler	{4}	Ctrl , AG, Mux, Reg = 1700 39 Registers = 2700 10 fixed-coeff-mult = 7000 8 add/sub = 2400	13800
1D-IDCT11_4 Butterfly	{4}	Ctrl , AG, Mux, Reg = ? 1700 39 Registers = ? 2700 7 fixed-coeff-mult = 4900 8 add/sub = ? 2400	11700 ?

Table 6.1 Estimated of the size of different implementations for the 1D-IDCT.

6.2.4 Two-dimensional IDCT

In this section the implementation of four different designs is discussed. The first is a conventional two-dimensional IDCT with a data introduction interval of DII=128. The throughput of this implementation, when using a clock rate of 27Mhz, is not sufficient to be used in the video decoder architectures which are described in Chapter 4. In this section two more implementations for the two-dimensional IDCT are discussed, these are the following and correspond to the video decoder architectures with the same name:

- Conventional 2D-IDCT with DII = 64
- New architecture 2D-IDCT with DII=48

Conventional 2D-IDCT, DII=128

A two-dimensional IDCT can be described in PIF very easily if a one-dimensional IDCT is available. In this case one 1D-IDCT is to be used, which is a hierarchical design. If this was not the case a simple function or macro could have been designed which describes the time shape of the 1D-IDCT. In this case a unit which imports eight samples in serial at the input and after a certain latency exports the results in serial at the output. The exact size of the latency does not substantially alter the design but does influence the memory requirements. The 1D-IDCT is now included in the higher level design. The 1D-IDCT must be invoked eight times for the rows and eight times for the columns (see figure 6.2). The transposition of the data is imposed by the different order of consumption of the second series of 1D-IDCTs. To address these signals an alias for the identifier had to be defined which contained the two-dimensional data in transposed organization. Phideo allocated memory to perform the transposition and took care of the address generation.

Instead of a memory size equal to the size of 64 samples, Phideo allocated a larger transposition memory which could contain 104 samples. After interpreting the results it became clear that this is a result of the pipeline in the design. This is clarified in figure 6.11. The inputs and the outputs of the first series of 1D-IDCTs are indicated by *h_idct_in* and *h_idct_out*, the inputs and outputs of the transposition memory are indicated by *transp_in* and *transp_out*. The interconnecting lines indicate the precedence relations. The latency of the first series of IDCTs followed by the transposition prohibited the second series of IDCTs (*v_idct*) to start immediately after the completion of the first series of IDCTs. As a result a second 1D-IDCT is needed, which is shown by the distribution function. If the second series of 1D-IDCT is postponed, as illustrated by the arrow, only one 1D-IDCT is needed. This results in a larger transposition memory since the samples of the second invocation arrive before the processing of the previous samples by the IDCT is completed. In appendix D an example of the PIF description is given.

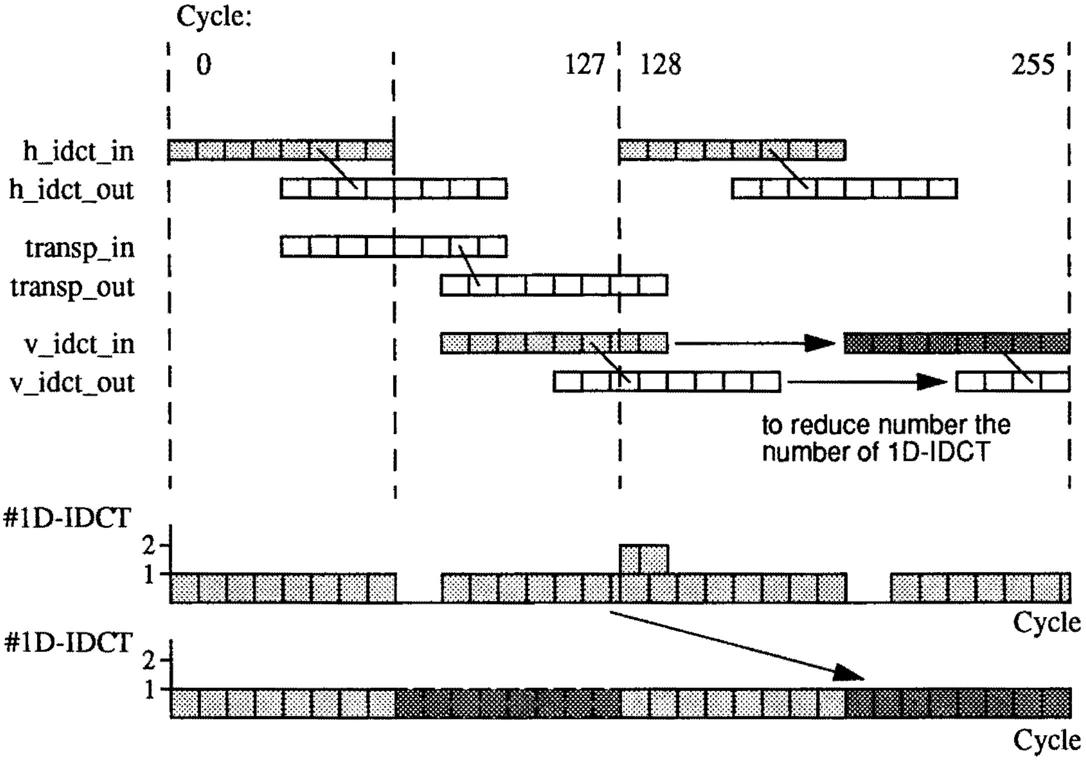


Figure 6.11 Implications for the schedule of a 2D-IDCT when using only one 1D-IDCT

Conventional 2D-IDCT, DII=64

To increase the throughput at a given clock rate the 2D-IDCT architecture of figure 6.3 can be used, which has a data introduction interval of DII=64. In this case a 1D-IDCT with DII=4 is required. At the input a serial/parallel converter is needed and at the output a parallel/serial converter is needed. These converters can be implemented by a small memory, because the only thing that has to be done is the buffering of some signal values. By describing the input order and rate and describing the output order and rate (see example below), Phideo introduces two memories together with the address generators.

```
(x : 0 .. 7) {8} ::
  (y : 0 .. 7) {1} ::
        serial_in[x][y] = input();

(x : 0 .. 7) {4} ::
  (y : 0 .. 3) {1} ::
  begin
        = output1(serial_in[x][y  ]);
        = output2(serial_in[x][y+4]);
  end;
```

The PIF input of Phideo consists of a description of the algorithm defining exactly which data is processed by the 1D-IDCT and in what order. Subsequently, Phideo generates a schedule and introduces distributed memories and registers to apply the required signals to the specified units at the correct time. By specifying constraints by means of pragmas, the signal can be mapped upon specified memories. This way the memory architecture of figure 6.3 can be obtained. Phideo's initial solution required about the same amount of memory but used more and smaller memories.

As was found in the implementation of the conventional 2D-IDCT in the previous section as well, again the overall size of the memories was larger than expected. Instead of a memory size equal to the size of a block which contains 64 samples, a memory with the size of 112 samples was allocated. Again the pipeline in the design was the cause. The second series of 1D-IDCTs was scheduled one frame period later and so that data from the next block arrived before the data of the previous block was processed.

If this design was made manually, this design problem probably was detected at a late stage in the design process. This could lead to backtracking and redesigning part of the implementation. The use of Phideo resulted in a fast insight in the consequences of certain design decisions, which can save a lot of time.

New architecture 2D-IDCT, DII=48

The new architecture as described in figure 6.7 is basically the same as the architecture of the previous section. The major difference are related to the second series of 1D-IDCT. Instead of 8 rows, only the rows from one of the fields are processed, e.g. only 4 rows. As a result the transposition memory can be reduced since only half the block is processed by the 1D-IDCT for the second time. This results in a small change in the PIF input. To start with, the frame period had to be changed to 48.

The main effort in designing this architecture is the selection of the samples which need to be transposed and processed for the second time. This depends on two conditions. The field which must be decoded and the type of the macroblock structure which is used. In case of frame DCT coding either the even lines or the odd lines must be decoded (see figure 6.8). In case of field DCT coding either the top four lines or the bottom four lines must be decoded.

This selection of these video lines can be described by means of a conditional construct in PIF as shown below.

```
control alternate_scan <0,1>; /* 0=field, 1=frame */
control even_odd <0,1>; /* 0=even, 1=odd */
```

```
case alternate_scan of
begin
         0: /*field*/
               case even_odd of
               begin
               0: /*even*/
               1: /*odd*/
               end;
         1: /*frame*/
               case even_odd of
               begin
               0: /*even*/
               1: /*odd*/
               end;
end;
```

Under each of the conditions another group of video lines can be selected to send to the transposition memories. In this case Phideo did not detect the resemblance between the different cases automaticaly. As a result Phideo allocates memory and generates address generators for each combination of conditions. The size of the memory allocated by Phideo is determined by the maximal number of memory locations required under one of the conditions. When using a pragma to assign the signals from both cases to one memory, additional memories were allocated to facilitate the imposed constraint.

It was very difficult to map the lines which were selected by the case statement under various conditions onto the same memories. The solution is an additional dummy instruction, which defines an alias for a given signal stream. This is illustrated by the following example:

```
case ctrl of
begin
         0: (i: 0 .. 3) {2}
               begin
{c0_ev}             c[2*1  ]= ... ;
{c0_od}             c[2*i+1]= ... ;
               end;
         1: (i: 0 .. 7) {1}
{c1}                c[i] = ... ;
end;

         (i : 0 .. 7) {1}
{rev}          d[i] = c[7-i];
```

In the case statement the signal stream *c* is depending on the *ctrl* signal filled by either one stream with period 1 or by two streams with period 2 and a smaller number of iterations. This is followed by an assignment which reverses the signal, which imposes a small memory. Unfortunately Phideo doesn't detect the similarity of the two cases, since the period enclosing the streams in both cases differ. As a result Phideo allocates two small memories, while one would be sufficient. This can be prevented by introducing a dummy signal assignment, an *alias*, which combines the result of the two cases into one signal stream (see below).

```
         (i : 0 .. 7) {1}
{alias}        tmp_c[i] = c[i];
```

This statement can be inserted between the case statement and the reverse operation *{rev}*, using *tmp_c* instead of *c* in the succeeding assignments. Since the output order and period equals the input of the alias, no memory is needed at the input of the alias. Now Phideo has to deal with a single signal and introduces only one memory instead of two.

As could be expected, the resulting design generated by Phideo required only half the amount of memory compared to the conventional implementation. In this case a memory which could contain 51 samples was required.

Gate count

In table 6.1 an estimation of the size of different implementation for the 2D-IDCT is given. The two-dimensional IDCTs use the 1D-IDCT with DII=4, make use of the estimation of the 1D-IDCT implementation containing Loefller's algorithm. It is likely that the implementation of the 1D-IDCT with the butterfly will result in a cheaper solution. If the estimations are accurate the sizes of the implementations must be decreased by 2100 gates!

Design	DII	Hardware & gate-count	
2D-IDCT_8 (8 x row, Transpose, 8 x column)	{128}	Ctrl , AG, Mux, Reg = 560 1D-IDCT11_8 = 13100 104 Registers = 6000	19700
2D-IDCT_4_conv (8 x row, Transpose, 8 x column)	{64}	Ctrl , AG, Mux, Reg = 700 1D-IDCT11_4 = 13800 112 Registers = 6300	20800
2D-IDCT_4_new	{48}	8 x row, Transpose, 4 x column Ctrl , AG, Mux, Reg = 1800 1D-IDCT11_4 = 13800 51 RegistersFile = 2900	18500

Table 6.1 Estimated of the size of different implementations for the 2D-IDCT.

6.3 Comments on the design flow

Phideo is an architecture level synthesis tool which requires the interaction of the designer. It is not a push-button design tool. Therefore it is essential to design a global architecture and specify the requirements, inputs and outputs before starting with Phideo. When applied correctly Phideo can be very helpful and takes care of lots of time consuming tasks. Phideo combines the advantages of design automation with the qualities of the human designer.

Phideo can be used to implement algorithms but cannot be used to optimise an algorithm. Before using Phideo it is needed to optimise an algorithm. For example the definition of an 8x8 one-dimensional IDCT uses 512 multiplications. When this algorithm is implemented with Phideo, enough multiplications are allocated to execute all multiplications in the specified time. If on the other hand an optimized algorithm for the one-dimensional IDCT is used, only 11 multiplications have to be executed. This will surely result in a cheaper solution since a smaller number of multiplications are needed for the implementation.

The reduction of the number of operations in an algorithm is *not the only* important criterion, as can be seen in the implementation of the butterfly algorithm (77). The *utilization of the resources* is also of major importance. Though the butterfly algorithm needed more operations, less resources were needed to implement the algorithm. If only a small number of operations are needed but all operations differ, then these cannot be mapped upon the same resource. This way the total number of resources could be larger while the utilization of the resources is very low. When another algorithm requires more operations and these operations can be shared, the total number of resources can be smaller because the resource utilization is better.

What algorithm is best suited to be implemented by Phideo can not be defined precisely. In general the number of operations must be reduced as far as possible. It is possible to evaluated different options. Some times bottlenecks in the implementation can be solved by adapting the input algorithm. These design decisions can not be automated and require the creativity and experience of the designers.

Phideo is very suitable to evaluate the feasibility of certain designs. It is possible to apply a top-down design method. The Processing Units which are needed and even complete hierarchical sub-designs can be described superficially in terms of time shape defining the order and timing of the inputs and outputs. This way the higher level design can be evaluated and bottlenecks in the architecture can be detected and solved in an early stage of the design process.

The use of pragmas makes it possible to evaluate different variants of an algorithm and to evaluate consequences of certain design decisions, without rewritting large parts of the implementation. The algorithmic description is defined at a high level and is defined only once. The rest of the work primarily consists of defining pragmas to steer the synthesis process. It is not necessary to rewrite the source time after time. With the definition of a single pragma, the effects of reduction of available resources or the effects of pipelining the design can be studied. Phideo takes care of lots of time consuming tasks and the designer can concentrate on the important design decisions and their consequences.

Though the amount of work to define additional constraints and to evaluate different implementations is limited to the definition of a number of pragmas, it can be very hard to steer Phideo. It is often hard to tell the consequences of the inclusion of a constraint on the output of Phideo. This has two reasons. First it is hard to tell the reason of certain choices or decisions by Phideo, where the exact bottlenecks were and why Phideo allocated the certain amount of resources. Secondly, it is hard to tell beforehand what the consequences of user defined constraints will be. Sometimes it costs a lot to satisfy the constraints, resulting in a worse implementation.

Overall, it can take a lot of time to get Phideo to do what the designer wishes. Sometimes the designer wants something that is not possible, sometimes the pragmas are insufficient or Phideo simply does not find the desired solution. The user interaction is of major importance, the experience and the creativity of the designer are indispensable. Therefore the feedback of Phideo to evaluate the temporary results are very important. Most of the times the resulting designs when using Phideo are comparable to the results the designer could achieve with manual design. Using Phideo does not make him a better designer. Nevertheless, the shorter design time and the possibility to evaluate the consequences of design decisions easier results in a better exploration of the design space. This can lead to better implementation since good alternatives could have been overlooked.

Chapter 7

Conclusions on the use of Phideo

The implementations and research of this report concern Phideo release 2.0. At the time of the research the work on this release had been stopped, since simultaneous a new release of Phideo was being developed which demanded all the effort. Part of the functionality concerning the controller generation was not implemented completely and still no proper manual or documented error messages were available. Fortunately this was no obstacle to evaluate Phideo. Hopefully these incompletions and imperfections will be revised in the near future.

Though immature and not completed fully, the main functionality of Phideo was already implemented and could be tested. As was described in "Research Goal and Method" on page 11, the goal of this report was to evaluate Phideo and judge the results according the following criteria:

1.	design time:	- how much time does it take to make a design?
2.	quality of the design:	- how good is the design in terms of chip area?
3.	applicability	- what kind of designs can be implemented successfully?
4.	design method:	- does it fit in the PCALE design flow?

In the next sections these criteria are discussed one by one. Followed by a section which discusses suggestions for improvements and finally the overall conclusions.

7.1 Design time

When starting from scratch, with no experience with Phideo, it takes a several weeks to become familiar with the Phideo design methods and to gain experience. It is an advantage to be familiar with architecture synthesis, and the main objectives and problems involved. Phideo's design method differs substantially from common general purpose synthesis approaches. Phideo is based upon periodic operations, algorithms are repeated at a regular rate and the generated schedules are pipelined.

The input of Phideo is at a high level of description which can be compared to behavioural VHDL. The design time of an algorithm in PIF is similar to the design time of a behavioural description. The main effort is not the design of the algorithm but the inclusion of contraints described in pragmas. The pragmas are designed in a iterative process and are used to steer the synthesis process. The task of the user is to interpret the results of Phideo and fine tune the tools by defining additional constraints. This involves only small modifications of the source code, which remains orderly and simple. This is a great advantage over a manual design flow which includes multi-levels of handwritten VHDL. This is in contrast to the design of RT-level descriptions, which would require a complete redesigning of all functional blocks in VHDL at a lower and more complex level.

Pragmas can be a very powerful instrument. The exploration of the design space can be done by changing or enclosing pragmas. This way it is very easy to compare different alternatives. With the modification of the frame period, the design can be pipelined and the data introduction interval can be changed. Pragmas can be used to assign matching signals to one memory or at a certain processing unit. At this point the experience of the user is very important. The strength of human insight can be exploited and main architectural decisions can be imposed by the user.

The implementation of an algorithm at Register Transfer Level starting with a Algorithmic Specification can be done in several hours to a few days depending on the complexity of the design. An estimation is of the total design time needed for an experienced user to create a design is given in table 7.1. Since the designs are not implemented manually, the design times for manual designs are only rough estimations based upon experience gained by manual design. The design times include the design of the PUs and the evaluation of several alternatives. The 8 one-dimensional IDCT, containing 4 variations of two different algorithms. Both algorithms are designed with 2 different data introduction intervals (DII=4 and DII=8) and using fixed-coefficient multipliers or normal multipliers. The two-dimensional IDCT is designed with a data introduction interval of DII=64 and once with DII=51, additionally a design was made which combined the two previous designs.

Design name	**Manual design time**	**Design time using Phideo**
8 one-dimensional IDCTs	8 weeks	2 weeks
3 types of two-dimensional IDCTs	10 weeks	3 weeks
Inverse Scan	2 weeks	3 days
Inverse Quantisation	3 weeks	1 week

Table 7.1 Estimated design time of various manual designs and Phideo designs.

7.2 Quality of the design

The quality of a design can be measured in silicon area, since the optimization goal is the size of the design. Since IC fabrication technology is subject to rapid changes, the size of a design measured in microns is not fixed and depends on the technology. A better measure is a gate count, this can be found by dividing the total chip area in a certain technology by the area of a NAND-port in that technology. This way a gate count in NAND equivalents is obtained which is more technology independent.

At this point and with release 2.0, Phideo's designs were not fully synthesizable yet, for example the memories and flip-flops were not linked in the RT-level VHDL and conditional constructs allocated separate memories for each condition. This made it very hard to obtain an accurate gate count. The area size of the memories had to be estimated by expressing the number of memory locations in flip-flop equivalents. The estimations of the manually designed equivalents are based upon implementations studies made at another group within Philips. Since the precise implementation was not available these estimations are based upon numbers from internal papers. The estimations in table 7.1 must be seen in this perspective.

Design name	DII	Manually generated	Phideo
one-dimensional IDCT	{8}	12000 gates	13000 gates
two-dimensional IDCT	{128}	20000 gates	20000 gates

Table 7.2 Estimated design size in gates

Another measure for the quality of the design is the following. Since the user plays an important part in the design process and the design decision are transparent to the user, it is possible to value the resulting design. The designer can compare each decision with the one he or she would have made in case of manual design. Even more, the designer can impose the same decision on the Phideo design. The main task of Phideo is the automatic generation of the schedules and the results. This way a design can be made with comparable quality but in less time.

7.3 Applicability

Phideo was designed to be used in the design of high throughput DSP applications. It should not be a surprise that Phideo works well for the implementation of this kind of algorithms. Especially when pipelined designs are concerned, Phideo can be of great help since manual design can be complex.

Phideo is not designed to optimise algorithms. Before using Phideo the algorithm needs to be optimized. All operations which are specified in the algorithm are scheduled by Phideo and enough resources are allocated to execute all operations under the given timing constraints. The algorithm optimisation is still a task of the human designer.

Nothing Phideo does cannot be done manually. Moreover this is not the primairy goal of Phideo. The main advantage of Phideo is the automation of lots of bookkeeping-like tasks. A simple schedule can be generated by every designer, though with Phideo it can be done much quicker and it is much easier to evaluate different alternatives. The main effort is changing pragmas and steering the tool.

In this perspective Phideo can be used for other applications. The implementation of a simple algorithm with sample frequencies smaller than the clock frequency can also be implemented with Phideo. Phideo takes care of the generation of the controller and the synthesis of the data path with its interconnect. Furthermore, Phideo takes care of the introduction of registers when a signal is delayed and the allocation of memories complete with address generation and control signals when more signals are concerned. Possibly Phideo has to be told to map a number of signals onto one memory instead of using distributed memories. If the memory construction and the location assignment of the signals is a task the user wants to do manually, a Processing Unit can be defined by the user which contains this functionality.

A large backdraw of designing algorithms with small sample rates, i.e. smaller than one sample per clock cycle, is that Phideo is not able to optimise accross hierarchical boundaries. If resources are used in a PU, this resource cannot be shared and used in another PU or at another hierarchical level. This way resources could be used very excessively. A processor like design with all purpose units very likely will result in better resource utilization. For this kind of applications the architecture level synthesis tool Mistral-2 [3] would be much more adequate.

Another application can be the generation of a memory unit which stores and retrieves signals which arrive in regular order and at regular times. In this case it can be sufficient to describe the time shape of the incoming and outgoing signals. When Phideo knows when signals arrive and at what time they have to be retrieved, it can generate the hardware including the memory and the addressing logic to perform this task. Such a simple design can be made within an hour!

A class of application which is not suited for implementation with Phideo are algorithms which have an irregular data flow, like a Variable Length Decoder, or algorithms were the data flow depends highly on the processed data. In Phideo, conditional flows must be encapsulated in processing units as much as possible. The resulting data flows of those irregular algorithms very likely will produce data which must be stored independently in a memory. As a result Phideo will not able to generate efficient address generators and is not equiped to solve this kind of design problems otherwise.

Algorithms implemented with Phideo are designed to be repeated regularily in time. Of course it is possible to execute the algorithm only once or once in a while. If the algorithm is pipelined, the outputs of a previous execution will be available after the introduction of the inputs of the next execution. An new execution can be restarted each frame period. It is also possible to stop or delay the complete design by masking the clock, and proceed when desired.

7.4 Design method

At PCALE a design method is used which has proven to be very fruitful. Phideo is suited to be applied in this design method. The algorithmic specification can be translated to an initial PIF description easily since both description are at the same level of description but in a different formalism.

Especially the intermediate output of Phideo which is at the Behavioural Level can be very useful for simulation and verification purposes. The intermediate output of Phideo is, as well as the final output, VHDL which makes it easier to run mixed simulations. Since memories are modelled by variables and address generation and storage and retrieval of data from memories is not needed in this model, the simulation speeds can be very high. Furthermore the inputs and outputs are bit-true and clock-cycle true which makes it suitable to be used as Behavioural Level VHDL in the PCALE design flow (section 1.4).

7.5 Suggestions for improvement

There are two points which can be improved in future releases of Phideo.

1. Phideo should take VHDL as an input language

2. Hierarchical Design should be extended

It would be an improvement over PIF, if VHDL could be used as an input language. This way the input could be executed and simulated to track down and remove functional errors at an early stage. At this moment a quick run has to be made without the need for optimizations, whereafter the first results can be verified. If correct, the input of Phideo was also correct. Another advantage of VHDL over PIF is that the translation of the Algorithmic Specification into the Phideo's input would become much easier.

The other improvement concerns Phideo's hierarchy. At this point hierarchy is limitted to the use of macros which are substituted in the higher level design. This way Phideo's hierarchy is not very clear and certainly not transparent. When true hierarchy was possible, this would be a real improvent.

Furthermore a few small improvements can be made. The first one is embraces the graphical representation of precedence relations in the schedule generated by Phideo which is already implemented. This way it becomes clear to the designer why certain units are scheduled the way they are and what operations are submitted to critical precedence relations. The user can determine how many precedence relations must be displayed, most time only the critical precedence relations are of interest to the designer.

Another improvement also involves the information given by Phideo on the schedule. Right now it is not displayed which operation in the schedule is assigned to which PU. The distribution function of a single PU, when several are allocated, cannot be obtained either. This way it is difficult to determine the resource utilization, which can be a measure for the efficiency of the implementation.

7.6 Overall conclusions

In this section the main advantages and disadvantages of Phideo are summarized, which were described in the comments on the design flow in the previous chapter and the previous paragraphs in this chapter.

The main advantages of Phideo are the following:

- **Short design times.** It is possible to implement a design in a relative short time and to obtain an estimation of the resource requirements, like number of operations and memory requirements (registers, RAM).
- **Better exploration of the design space.** Since the design time is short, it is possible to compare different architectures and designs within a given period of time.
- **High Level of description.** The input of Phideo is at a very high level. This way the designer can concentrate on a compact and orderly design. The designer can dissociate from lots of implementation details and focus on the bottlenecks of the design first.
- **Quality comparable to manual designs.** This is a direct result of the extensive user interaction which can intervene with all processes and decisions in the design process. The designer can make the design the same as a manual design, in shorter time. Possibly even better results could achieved since a better exploration of the design space can be made which would be impractical with manual design because that would take to much time.
- **Pipelined schedules can be generated.** A important strength of Phideo is the scheduler Jason, which generates a schedule with a pipeline, so the design can be restarted before it is completely executed. Since the data introduction interval is smaller than the overall latency, a better occupation of the resources can be realized. This way a high throughput can be achieved with less resources.
- **Automated allocation of memory and construction of address generators.** Another strength of Phideo is the address generator Matchbox. The only thing which has to be known is the input and output order and period of the variables, this is extracted from the design automatically. Phideo minimizes the overall memory by choosing a smart location assignment and generates address generators.
- **Phideo generates also a Behavioural VHDL.** The Behavioural Level VHDL can be used as a behavioural model for verification purposes. Since memories are modelled by variables and address generation and storage and retrieval of data from memories is not needed, the simulation speeds can be very high. Furthermore the inputs and outputs are bit-true and clock-cycle true which makes it suitable to be used as Behavioural Level VHDL in the PCALE design flow.

A few disadvantages of Phideo are at this moment:

- **Often hard to steer Phideo.** Phideo can be steered by means of the definition of constraints in terms of pragmas. It is often difficult to oversee all the consequences of the inclusion of a certain constraint. This can result in additional hardware since Phideo will try to meet the additional constraints. Sometimes this is the blaim of the designer who wants something what is not possible, sometimes Phideo simply does not find the solution. This emphasizes the importance of the feedback. This feedback must provide the user with enough information to improve the design if possible. The better the feedback the better the designer can evaluate and improved the designs.

- **Little documentation and unclear error messages.** Simultaneous with the work of this report a new release of Phideo was being developed. As a result the work on the previous release had been stopped, since at that time all effort was put into the designs of the new release. At this point still no proper manual for either of the releases exists and error messages contain a lot of uninteresting and unclear messages which are not documented as well. Hopefully this will be improved in the future.
- **VHDL cannot be used as input language.** The Phideo Input Language can not be executed directly. At this moment with Phideo release-2.0 the input of Phideo can not be executed. To circumvent this problem it is possible to carry through Phideo without optimization and verify the first result on its functional correctness. If so, the algorithmic input of Phideo was correct. It would be a great advantage if PIF would be replaced by VHDL as soon as possible.
- **Hierarchy in Phideo is not transparent.** Phideo does not support true hierarchy. Hierarchy is limited to the use of macros which are substituted in the higher level design. This way Phideo's hierarchy is not very clear and certainly not transparent. In future releases hierarchy is planned to be expanded.

High throughput DSP applications are best suited to be designed by Phideo, as could be expected since Phideo is designed for these applications. Unfortunately not all types of applications can be designed. When lower freqencies and smaller throughput are required other tools will probably be much more efficient. Phideo has a large number of advantages but still has to mature. At this moment manuals are not available and error messages are not documented. The main reason for this is that the tool is still in the development stage, this will surely improve in time. In time I expect Phideo to develop into a very valuable and efficient tool, which it largely already is.

Appendix A Phideo Input Language (PIF)

A PIF description consists of two main parts: a declarative part and an algorithmic part. Furthermore additional constraints or pragmas can be defined to steer the synthesis process.

Declarations

There are three kinds of declarations:

1. Input and output terminals:

```
infunc input = in_term;
outfunc output = out_term;
```

The I/O-terminals with the names *input* of type in_term and *output* of type out_term are created.

2. Processing units and their functions

```
func +(in1, in2) out = add_pu;
func *(in1, in2) out = mult_pu;
func some_func(a,b,c) x,y = some_pu;
func multi_cycle(a,b) x = {2} a_pu;

func add(in1, in2) out <sel=0:*> = addsub_pu;
func sub(in1, in2) out <sel=1:*> = addsub_pu;
```

The identifiers between parenthesis specify the inputs of the function the succeeding identifiers specify the outputs. The identifier succeeding the equality sign defines the processing units. These PUs should be defined in VHDL. An example of a Processing Unit in VHDL can be found in appendix C.

A number between brackets preceding the Processing Units denotes the *data introduction interval* or restart time. The functions *add* and *sub* are mapped upon the same PU, addsub_pu. An additional input *sel* is created to control the function of this unit.

3. Signals and their widths

```
signal a,b = 8;
signal c = 14;
```

With this declarations signals and their widths in bits can be defined.

Algorithm

The algorithmic part of the design starts with the definition of the global period and execution interval:

```
{4} : [0,100]
```

The algorithm should be restarted every 4 clock cycles, the *data introduction interval* is 4. All operations should start between cycle 0 and cycle 100. By adjusting global period the data introduction interval of the algorithm can be changed. If the data introduction interval is made smaller than the latency, a pipelined unit is designed.

The algorithmic part consist of a description of the algorithm in terms of the declared PU-functions. Each assignment is threaded as a single assignment. Ordering of statements has no meaning, the ordering of operations is determined by the data dependencies. Only one function call per statement is allowed, so:

```
a = b + c + d;
y = f(g(x));
```

are **not** allowed!

Three types of function calls are possible:

1. Input functions:

```
result = funcname();

a = input();
```

2. Output functions:

```
= funcname(arg);

= output(a);
```

3. PU functions:

```
results = func(args);
result = arg1 operator arg2;

u,v = some_func(x,y,z);
a = b + c;
```

All functions calls may be preceded by a label

```
{lbl}    a = b + c;
```

Iterations or loops in Phideo describe a 'time' loop! All iterations of a single assignment or function within the loop will be mapped upon the same Processing Unit. The operations are scheduled with a fixed period conform the Phideo model of periodic operations.

```
(iterator : minval .. maxval) {period} ::
```

```
(y : 0 .. 312) {864} ::
        (x : 0 .. 719) {1} ::
        begin
                a[i] = a_func(b[i]);
                x[i+10] = other_func(a[i],y[2*i]);
        end;
```

The iterator is incremented every *period* clock cycles.

Conditional parts of the algorithm can be found at two levels. In the Processing Units and in the algorithm. It is preferred to implement the conditional parts of the algorithm in Processing Units as much as possible since this reduces the complexity of the scheduling process. The conditional constructs in the algorithm which are not mapped upon a Processing Unit are submitted to a constraint. The control value of this conditional construct must be know at the start of the algorithm. It is not possible to use a variable which is computed within the algorithm as the control value. This control value must be declared with as follows:

```
control cname = <values>;

control select = <0,1,2,3,4,5,6>
```

An extra terminal called *select* is created wide enough to represent the values (in this case 3 bits). The conditional assignment has the following construct:

```
case sig of
v1: ...
<v2,v3>: ...
else ...
esac:

case select of
1:        a = f(b);
<2,3>:    a = g(b,c);
else      a= h(b,c,d);
esac:
```

Pragmas

In addition to the assignments to describe the algorithm, a number of commands are defined to give additional constraints and hints to the Phideo synthesis tools. These commands or *pragmas* can be use in the iteration loop of the Phideo design method to steer the scheduler and the allocation process.

The *sequence pragma* puts constraints on the scheduling time points of operations. With sequence pragmas either the exact schedule time of a operation or the relative schedule time of two operations can be specified. The arguments of the sequence pragmas denote the labels of operations.

```
%ina = 5;
%inb - ina = 2;
```

The *assign pragma* poses assignment constraints to operation. In its first form (one '@'), several operations of the same resource type can be mapped on the same unit by assigning the same value. The second form (two '@'), all operations with the same assignment value are mapped to a uniquely for these operations reserved unit.

```
%m1, m2 @1
%a @@2
```

*MemAlloc pragm*a and *signalMem pragma* are used to pre-allocate a memory of a certain type and to assign a signal (array) to an pre-allocated memory. The available types should be in the Phideo memory library file, e.g. Regfile or SRAM. The name of the memory is defined between double quotes. The number of read and write ports and access times are defined in the library file.

```
%memory “one“ = SRAM;
%memory “two“ = REGFILE;
%temp = “one“ ;
```

A pre-allocation for the number of Processing Units of a certain type can be defined in the declaration of a function. With square brackets enclosing a number, the number of Processing Units can be specified.

```
func *(in1, in2) out = mult_pu[3];
```

Appendix B Hierarchical design in PIF

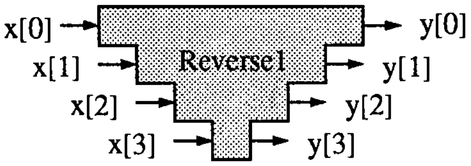
To illustrate the use of hierarchy in PIF, two PIF descriptions are given below. Both algorithms are designed to reverse the order of 4 successive samples. The goal is to design a component which reverses the order of eight successive samples. Below two PIF routines are showed, Reverse1 and Reverse2, which realize this function.

```
Reverse1:

           (i: 0 .. 3) {1} ::
{in}              x[i] = input();

           (i: 0 .. 3) {1} ::
                  y[i] = x[i];

           (i: 0 .. 3) {1} ::
{out              = output(y[3-i]);
```

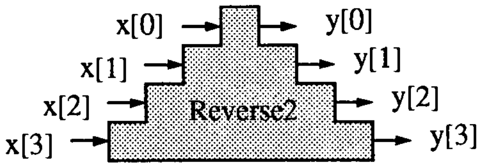


```
Reverse2:

           (i: 0 .. 3) {1} ::
{in}              x[3-i] = input();

           (i: 0 .. 3) {1} ::
                  y[i] = x[i];

           (i: 0 .. 3) {1} ::
{out              = output(y[i]);
```



The only difference between the two descriptions is the index order of the inputs and outputs. Usually names and the index of variable are transparent and not of any influence. This is not true in this case! When the macro is used at a higher level, the Phideo preprocessor substitutes the names of the variables in the macro. The order and period of the signal remain the same. As a result the internal index order of the input and output variables are used for the substituted variables as well.

The macro of description Reverse2 is shown below. Note that the variables x and y originate from the PIF description and therefore must differ!

```
MACRO reverse2_bbx(label, instance ,y ,x)

           (i: 0 .. 3) {1} ::
{func_in}         func_in(x[7-i]);

           (i: 0 .. 3) {1} ::
{func_out}        func_out(y[i]);
```

At a higher level of hierarchy this file can be used in another PIF description. The *.bbx* file can be included as shown bellow (not followed by a semicolon!), now the macro can be used like a function as shown in the following example. The input and output parameters *a* and *b* are substituted. The label of the function is *Reverse* and the *instance* number is 1.

```
#include "file_name.bbx"
         (i: 0 .. 3) {1} ::
{in}           a[i] = input();

         file_name_bbx (reverse, 1, b, a);

         (i: 3.. 0) {1} ::
{out}          = output(b[i]);
```

When the macro Reverse2 is used the order and period of the variable *a* and *b* are determined by the macro. The order of *a* will be *a[3-i] with (i: 0 .. 3) and period {1}*. The order of *b* will be *b[i] with (i: 0 .. 3) and period {1}*. Since the production order of the input does not match the consumption order of the macro 'reverse', Phideo inserts an additional to match the production and consumption of the data (figure 7.1). The same can be seen between the output of the macro 'reverse' and the output terminal.

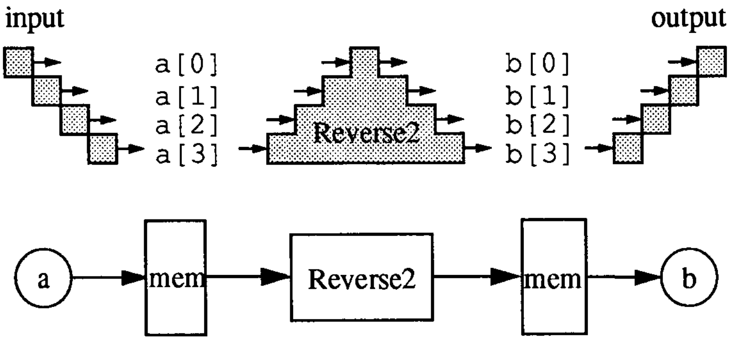


Figure 7.1 Time shapes of the input, the output and the function Reverse, and the resulting schematic when using Reverse2.

The use of the macro Reverse1 in the previous example will result in the schematic of figure 7.2. Note that the order and period of the signal produced by the input and the signal consumed by Reverse1 match, therefore no additional memory is necessary.

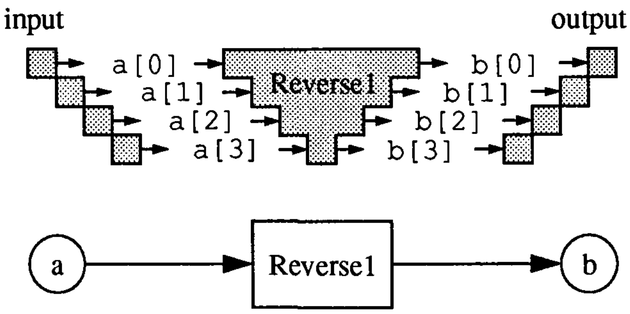


Figure 7.2 Time shapes of the input, the output and the function reverse, and the resulting schematic when using Reverse1.

Appendix C Example of user-defined PUs

PUs used for one-dimensional IDCT

```
-- "idct11_8_pus.vhd"

-- PUs for one-dimensional IDCT
-- used for:
-- idct11_8.pif, DII=8 and
-- idct11_4.pif, DII =4

library ieee;
use ieee.std_logic_1164.all;

package idct11_4_pus is
     component mult_pu
     port(
          ck : in std_ulogic;
          input1, input2: in std_ulogic_vector;
          output : out std_ulogic_vector
     );
     end component;

     component addsub_pu
     port(
          ck : in std_ulogic;
          input1, input2: in std_ulogic_vector;
          output : out std_ulogic_vector;
          c : in std_ulogic
     );
     end component;

     component lut_pu
     port(
          ck : in  std_ulogic;
          output : out std_ulogic_vector(13 downto 0);
          addr : in  std_ulogic_vector(3 downto 0)
     );
     end component;
end idct11_4_pus;

library ieee;
use ieee.std_logic_1164.all;

library synergy;
use synergy.constraints.all;

entity mult_pu is
     port(
          ck : in std_ulogic;
          input1, input2: in std_ulogic_vector;
          output : out std_ulogic_vector
     );
     attribute preserve of mult_pu: entity is true;
end mult_pu;
```

```
architecture behaviour of mult_pu is
BEGIN
END behaviour;

library ieee;
use ieee.std_logic_1164.all;

library synergy;
use synergy.constraints.all;

entity addsub_pu is
    port(
        ck : in std_ulogic;
        input1: in std_ulogic_vector(16 DOWNTO 0);
        input2: in std_ulogic_vector(16 DOWNTO 0);
        output : out std_ulogic_vector(16 DOWNTO 0);
        c : in std_ulogic
    );
    attribute preserve of addsub_pu: entity is true;
end addsub_pu;

use ieee.std_logic_arith.all;

architecture behaviour of addsub_pu is
BEGIN
    add: PROCESS (input1, input2, c)
        VARIABLE op1, op2, t: std_ulogic_vector(17 DOWNTO 0);
    BEGIN
        IF is_x(input1) OR is_x(input2) OR is_x(c) THEN
            output <= (OTHERS=>'X');
        ELSE
            op1 := input1(16) & input1(16 DOWNTO 0);
            op2 := input2(16) & input2(16 DOWNTO 0);
            IF (c='1') THEN t := op1+op2; ELSE t := op1-op2;
            END IF;

            -- saturation
            IF NOT(t(17)=t(16)) THEN
                ASSERT (t(17)='1') REPORT "Positive overflow
in addsub_pu" SEVERITY warning;
                ASSERT (t(17)='0') REPORT "Negative overflow
in addsub_pu" SEVERITY warning;
                t(15 DOWNTO 0) := (OTHERS=> NOT(t(17)));
            END IF;
            output <= t(17) & t(15 DOWNTO 0);
        END IF;
    END PROCESS add;
END behaviour;
```

PUs used for two-dimensional IDCT

```
-- "idct_8_pus.vhd"

-- PUs for two-dimensional IDCT
-- used for:
-- idct_8.pif, DII=128 and
-- idct_4_x_conv.pif, DII = 64 and
-- idct_4_x_nieuw.pif, DII = 51 and
-- idct_4_x_both.pif, DII =64 resp. DII= 51

library ieee;
use ieee.std_logic_1164.all;

package idct_4_x_pus is
    component idct11_4 port(
        ck : in std_ulogic;
        in1_term : in std_ulogic_vector (16 downto 0);
        in2_term : in std_ulogic_vector (16 downto 0);
        out1_term : out std_ulogic_vector (16 downto 0);
        out2_term : out std_ulogic_vector (16 downto 0);
        global_reset : in std_ulogic);
    end component;

    component times8_pu
    port(
        ck : in std_ulogic;
        input : in std_ulogic_vector;
        output : out std_ulogic_vector
    );
    end component;

    component divide64_pu
    port(
        ck : in std_ulogic;
        input : in std_ulogic_vector;
        output : out std_ulogic_vector
    );
    end component;
end idct_4_x_pus;

library ieee;
use ieee.std_logic_1164.all;

library synergy;
use synergy.constraints.all;

entity idct11_4 is
    port(
        ck : in std_ulogic;
        in1_term : in std_ulogic_vector (16 downto 0);
        in2_term : in std_ulogic_vector (16 downto 0);
        out1_term : out std_ulogic_vector (16 downto 0);
        out2_term : out std_ulogic_vector (16 downto 0);
        global_reset : in std_ulogic
        );
    attribute preserve of idct11_4: entity is true;
end idct11_4;
```

```
ARCHITECTURE behaviour OF idct11_4 IS
BEGIN
END behaviour;

library ieee;
use ieee.std_logic_1164.all;

entity times8_pu is
     port(
          ck : in std_ulogic;
          input : in std_ulogic_vector;
          output : out std_ulogic_vector
     );
end times8_pu;

ARCHITECTURE behaviour OF times8_pu IS
BEGIN
     times8: PROCESS (input)
          CONSTANT m: natural := input'LENGTH-1;
          VARIABLE tmp : std_ulogic_vector(m DOWNTO 0);
     BEGIN
          tmp := input(m) & input(m-4 DOWNTO 0) & "000";
          IF NOT( (input(m)=input(m-1)) AND
                  (input(m)=input(m-2)) AND
                  (input(m)=input(m-3)) ) THEN
               ASSERT (input(m)='1') REPORT "Positive overflow
in times8_pu" SEVERITY error;
               ASSERT (input(m)='0') REPORT "Negative overflow
in times8_pu" SEVERITY error;
               tmp(m-1 DOWNTO 0) := (OTHERS=> NOT(input(m)));
          END IF;
          output <= tmp;
     END PROCESS times8;
END behaviour;

library ieee;
use ieee.std_logic_1164.all;

entity divide64_pu is
     port(
          ck : in std_ulogic;
          input : in std_ulogic_vector;
          output : out std_ulogic_vector
     );
end divide64_pu;

USE ieee.std_logic_arith.ALL;

ARCHITECTURE behaviour OF divide64_pu IS
BEGIN
     divide64: PROCESS (input)
          CONSTANT m: natural := 16; --  input'length-1;
          VARIABLE rnd : std_ulogic_vector(m+1 DOWNTO 0);
          VARIABLE sat : std_ulogic_vector(m DOWNTO 0);
          VARIABLE pos_overflow, no_neg_overflow : std_ulogic;
     BEGIN
          IF is_x(input) THEN
               output <= (OTHERS=>'X');
          ELSE
```

```
            --  divide by 64, keep 1 extra bit for rounding
            rnd := input(m) & input(m) & input(m) &
                   input(m) & input(m) & input(m) &
                   input(m DOWNTO 5);
            rnd := rnd + 1;
            sat := rnd(m+1 DOWNTO 1);

            --  saturation
            pos_overflow := '0';
            no_neg_overflow := '1';
            FOR i IN 15 DOWNTO 8 LOOP
                pos_overflow := pos_overflow OR sat(i);
                no_neg_overflow:= no_neg_overflow AND sat(i);
            END LOOP;

            IF (((sat(16)='0') AND (pos_overflow='1')) OR
                ((sat(16)='1') AND (no_neg_overflow='0')) )
            THEN
                --  ASSERT (sat(16)='1') REPORT "Positive
overflow in div64_pu" SEVERITY warning;
                --  ASSERT (sat(16)='0') REPORT "Negative
overflow in div64_pu" SEVERITY warning;
                sat(15 DOWNTO 8)  := (OTHERS=> sat(16));
                sat( 7 DOWNTO 0)  := (OTHERS=> NOT(sat(16)));
            END IF;

            output <= sat(16 DOWNTO 0);
        END IF;
    END PROCESS divide64;
END behaviour;
```

Appendix D Example of PIF design

```
// "idct11_8.pif"

// uses PUs:"idct11_8_pus.vhd"

// One dimensional IDCT
// algorithm: loeffler with 11 multiplications
// Data Introduction Interval = 8

// declarations

infunc input1 = in_term [1];
outfunc output1 = out1_term [1];

func + (input1, input2) output {1} <c=1:*> = addsub_pu [4];
func - (input1, input2) output {1} <c=0:*> = addsub_pu [4];
func * (input1, input2) output {1} = mult_pu [10];

signal idct_in, idct_out = 17;
signal hulp, m, d, a, b, c = 17;
signal coeff = 14;

// algorithm

{8} : [0, 100]

         (i : 0 .. 7) {1} ::
{in1}           idct_in[i] = input1();

{a4}     a[4] = idct_in[1] - idct_in[7];
{a5}     a[5] = idct_in[3] * coeff;
{a6}     a[6] = idct_in[5] * coeff;
{a7}     a[7] = idct_in[1] + idct_in[7];

         .
         .
         .

{o0}     idct_out[0] = c[7] + c[0];
{o1}     idct_out[1] = c[6] + c[1];
         .
         .
{o7}     idct_out[7] = c[0] - c[7];

         (i : 0 .. 7) {1} ::
{out1}          = output1(idct_out[i]);

// pragmas

% out2 - out1 = 0;

%m1 @@1; %m2 @@2; %m3 @@3; %m4 @@4; %m5 @@5;
%m6 @@6; %m7 @@7; %m8 @@8; %m9 @@9;
%a5, a6 @@10;
```

Appendix D:Example of PIF design

```
// "idct11_4.pif"

// uses PUs:"idct11_4_pus.vhd"

// One dimensional IDCT
// algorithm: loeffler with 11 multiplications
// Data Introduction Interval = 4
// => two input and two output terminals

// declarations

infunc input1 = in1_term [1];
infunc input2 = in2_term [1];
outfunc output1 = out1_term [1];
outfunc output2 = out2_term [1];

func + (input1, input2) output {1} <c=1:*> = addsub_pu [8];
func - (input1, input2) output {1} <c=0:*> = addsub_pu [8];
func * (input1, input2) output {1} = mult_pu [10];

signal idct_in_top, idct_in_bot = 17;
signal idct_out_top, idct_out_bot = 17;
signal m, d, a, b, c = 17;
signal coeff = 14;

// algorithm

{4} : [0, 100]

        (i : 0 .. 3) {1} ::
        begin
{in1}         idct_in_top[i] = input1();
{in2}         idct_in_bot[i] = input2();
        end;

{a4}    a[4] = idct_in_top[1  ] - idct_in_bot[7-4];
{a5}    a[5] = idct_in_top[3  ] * coeff;
{a6}    a[6] = idct_in_bot[5-4] * coeff;
{a7}    a[7] = idct_in_top[1  ] + idct_in_bot[7-4];

        .
        .

{o0}    idct_out_top[0  ] = c[7] + c[0];
{o1}    idct_out_top[1  ] = c[6] + c[1];
        .
        .
{o7}    idct_out_bot[7-4] = c[0] - c[7];

        (i : 0 .. 3) {1} ::
        begin
{out1}        = output1(idct_out_top[i]);
{out2}        = output2(idct_out_bot[i]);
        end;

// pragmas
% out2 - out1 = 0;
%m1 @@1; %m2 @@2; %m3 @@3; %m4 @@4; %m5 @@5;
%m6 @@6; %m7 @@7; %m8 @@8; %m9 @@9;
%a5, a6 @@10;
```

```
// idct_8.pif

// Two-dimensional IDCT
// using idct_11_8.pif, one-dimensional IDCT with DII=8
// Data Introduction Interval = 128
// (8 * idct_row + 8 * idct_row = 128)

// declarations

infunc input = in_term [1];
outfunc output = out_term [1];

func times8 (input) output = times8_pu [1];
func divide64 (input) output = divide64_pu [1];

#include "idct11_8.bbx"

signal in1, in8, out1, out64 = 17;
signal v_idct, v_idct_t, h_idct= 17;

// algorithm

{128} : [0, 2*128]

         (x : 0 .. 7) {8} ::
           (y : 0 .. 7) {1} ::
           begin
{in1}          in1[x][y] = input();
{in8}          in8[x][y] = times8(in1[x][y]);
           end;
%in1 = 0;

         (x : 0 .. 7) {8} ::
               idct11_8_bbx(v_idct,1, in8[x], v_idct[x]);

         (x : 0 .. 7) {8} ::
           (y : 0 .. 7) {1} ::
{transp}       v_idct_t[x][y] = v_idct[y][x];
%transp : alap;

         (x : 0 .. 7) {8} ::
               idct11_8_bbx(h_idct,1, v_idct_t[x], h_idct[x]);

         (x : 0 .. 7) {8} ::
           (y : 0 .. 7) {1} ::
           begin
{out64}        out64[x][y] = divide64(h_idct[x][y]);
{out1}         = output(out64[x][y]);
           end;
```

Appendix E Bibliography

[1] P. Michel, U. Lauther, P. Duzy,
"The synthesis approach to digital system designs,"
Boston/Dordrecht/London, Kluwer Academic Publishers

[2] J.A.A.M. van den Hurk,
"Evaluation of the PCALE VLSI design flow for HDTV ICs,"
Philips Semiconductors Internal Laboratory Report, ETV92004, 20-07-1992.

[3] E.J. van Dalen,
"MPEG as a test-case for the high-level synthesis Tool Mistral2,"
Master thesis, Delft University of Technology, The Netherlands.

[4] R.K. Brayton, R. Camposano, G. Demicheli, et al.
"The Yorktown Silicon Compiler,"
Silicon Compilation, 1988, pp. 204-311

[5] H. De Man, F. Catthoor, G. Goossens, et al.
"Architecture-driven synthesis techniques for VLSI implementation of DSP algorithms,"
Proceedings of the IEEE, Feb 1990, pp. 319-335

[6] R. Woudsma, F. Beenker, J. Van Meerbergen, et al.
"PIRAMID : an architecture-driven silicon compiler for complex DSP applications,"
Proceedings IEEE Int. Symp. on Circuits and Systems, 1990, pp. 2696-2700

[7] P. Lippens, J.. van Meerbergen, A. van der Werf, et al.
"Phideo : a silicon compiler for high speed algorithms,"
Proceedings of the EDAC, Amsterdam (Netherlands), pp. 436-441, Feb. 1991.

[8] J. van Meerbergen, P. Lippens, B. McSweeney, et al.
"A Design Strategy for High-Througput Applications,"
VLSI Signal Processing V, pp. 150-165, Oct. 1992,

[9] J. van Meerbergen, P. Lippens, B. McSweeney, et al.
"Architectural Strategies For High-Throughput Applications,"
Journal of VLSI Signal Processing 5, pp. 201-220, 1993.

[10] J. van Meerbergen, P. Lippens, W. Verhaegh, et al.
"Phideo: High-Level Synthesis for High Throughput Consumer Applications"
submitted for the Journal of VLSI Signal Processing, special issue on DSP Synthesis.

[11] ISO/IEC 13818-1 (MPEG-2 Systems),
"Information technology - Generic coding of moving pictures and associated audio"
International Organization for Standardization, International Standard
ISO/IEC JTC1/SC29/WG11, Nov. 1994.

[12] ISO/IEC 13818-2, ITU-T Recommendation H.262, (MPEG-2 Video)
"Information technology - Generic coding of moving pictures and associated audio,"
International Organization for Standardization, Final Draft of International Standard
ISO/IEC JTC1/SC29/WG11, 20 Jan. 1995.

[13] ISO/IEC 13818-3 (MPEG-2 Audio),
"Information technology - Generic coding of moving pictures and associated audio,"
International Organization for Standardization, International Standard
ISO/IEC JTC1/SC29/WG11, 11 Nov. 1994.

[14] ISO/IEC 13818-4 (MPEG-2 Compliance),
"Information technology - Generic coding of moving pictures and associated audio,"
International Organization for Standardization, Committee Draft
ISO/IEC JTC1/SC29/WG11, 11 Nov. 1994.

[15] ITU-T Recommendation H.261 (Formerly CCITT Recommendation H.261)
"Codec for audiovisual services at px64 kbit/s,"
Geneva, 1990

[16] IEEE Std 1180-1990
"IEEE standard specification for the implementations of 8 x 8 Inverse Discrete"
Dec. 1990.

[17] ISO/IEC 10918-1, ITU-T Recommencations T.81, (JPEG)
"Digital compression and coding of continuous-tone still images,"

[18] ISO/IEC 11172 (MPEG-1)
"Information technology - Coding of moving picture and associated audio for digital storage media at up to about 1,5 Mbit/s,"
International Organization for Standardization, International Standard, 1993

[19] L. Chiariglione,
"The Development of an Integrated Audiovisual Coding Standard,"
Proceedings of the IEEE, Vol. 83, No. 2, Feb. 1995.

[20] S. Baron, W. R. Wilson,
"MPEG overview"
SMPTE Journal, June 1994.

[21] D. J. Le Gall,
"The MPEG video compression algorithm"
Signal Processing: Image Communication, Vol. 4, No. 2, pp. 129-149, April 1992.

[22] D.J. Le Gall,
"The MPEG Video Compression Algorithm: A Review"
SPIE, Vol. 1452 Image Processing Algorithms and Techniques II, pp. 444-457, 1991.

[23] P. Pirsch, N. Demassieux, W. Gehrke,
"VLSI Architectures for Video Compression - A survey"
Proceedings of the IEEE, Vol. 83, No. 2, Feb. 1995.

[24] A. Puri,
“Video coding using MPEG-2 compression standard”
SPIE Visual Communications and Image Processing, Vol. 2094, pp. 1701-1713, 1993

[25] N. Ahmed, T. Natarajan, K.R. Rao,
“Discrete Cosine Transform,”
IEEE Transactions on Computers, Vol. C-23, Jan. 1974, pp. 90-93.

[26] K.R. Rao, P.Yip,
“Discrete cosine transform, algorithms, advantages, and applications,”
London, Academic Press, 1990.

[27] M. Vetterli and H. Nussbaumer,
“Simple FFT and DCT algorithms with reduced number of operations,”
Signal Processing, Vol. 6, pp. 267-278, Aug 1984.

[28] W.H. Chen, C.H. Smith, and S.C. Fralick,
“A fast computational algorithm for the discrete cosine transform,”
IEEE Trans. on Communications, Vol. COM-25, No. 9, pp. 1004-1011, Sept. 1977.

[29] Z. Wang,
"Reconsideration of a fast computational algorithm for the discrete cosine transform,"
IEEE Trans. on Communications, Vol. COM-31, pp. 121-123, Jan. 1983.

[30] B. Lee,
"A new algorithm to compute the discrete cosine transform,"
IEEE Trans. Acoust., Speech, and Signal Processing, Vol. ASSP-32, No. 6, pp. 1243-1245, Dec. 1984

[31] H.S. Hou,
"A fast recursive algorithms for computing discrete cosine transform,"
IEEE Trans. Acoust., Speech, and Signal Processing, Vol. ASSP-35, No. 10, pp. 1455-1461, Oct. 1987.

[32] C. Loeffler, A. Ligtenberg, G.S. Moschytz,
“Practical fast 1-D DCT algorithms with 11 multiplications,”
Proceedings IEEE Int. Conf. on Acoust., Speech, Signal Processing,
ICASSP-89, Vol. 2, pp. 988-991, May 1989.

[33] **“Video codec for audiovisual service at p x 64 kbits/s,”**
ITU-T Recommendation H.261,
CCITT Study Group XV: Jul. 1990.

[34] **“IEEE standard specifications for the implementations of 8x8 inverse discrete cosine transform,”**
IEEE Std 1180-1990, Mar. 1991.

[35] A.M. Rensink, A. van der Werf,
“MPEG-2 as a test case for Phideo,”
Philips Research Laboratories, Nat.Lab. Technical Note Nr. 223/93.

[36] H.L. Garner,
“Number systems and arithmetic,”
Advances in Computers, Vol. 6, New York: Academic, 1965, pp. 163-164.

[37] K. Hwang,
“Computer arithmetic, principles, architecture, and design,”
New York, John Wiley & Sons Inc.

Appendix F Index & glossary