



TAMPERE UNIVERSITY OF TECHNOLOGY

*Degree Programme in  
Communications Electronics*

**ANTTI RASMUS**

**INTEGRATION OF HARDWARE ACCELERATORS  
INTO A SYSTEM-ON-CHIP VIDEO ENCODER**

Master of Science Thesis

Examiners: Professor Timo D. Hämäläinen  
Docent Marko Hännikäinen  
M. Sc. Ari Kulmala

Examiners and topic approved in the  
Information Technology Department Council  
meeting on 15 January 2007

## ABSTRACT

TAMPERE UNIVERSITY OF TECHNOLOGY

Master's Degree Programme in Communications Electronics

**RASMUS, ANTTI TAPIO:** Integration of Hardware Accelerators into a System-on-Chip Video Encoder

Master of Science Thesis, 74 pages

June 2007

Major: Digital and Computer Engineering

Examiners: Professor Timo D. Hämäläinen, Docent M. Hännikäinen, and M. Sc. Ari Kulmala

Keywords: hardware acceleration, video encoder, integration, system on chip, FPGA

This Thesis studies how hardware accelerators are integrated into a system on chip. It covers the approaches and phases of integration. In addition, the Thesis studies the impact on performance resulting from accelerator integration. Two hardware accelerators are integrated into multiprocessor system-on-chip video encoder as a case study. The one hardware accelerator performs motion estimation (ME) and the other combination of discrete cosine transform, quantization, inverse quantization, and inverse discrete cosine transform (DQ).

Integrating the hardware accelerators into the system is complex operation because the functionality and interfaces of the hardware accelerators differ from the rest of the system on both system and signal level. In addition, one must design how application software sees the accelerators through drivers. The work analyzes different integration strategies, one of which is implemented by manually creating wrapper components for the accelerators. In addition, the integration procedure and the wrapper block implementations are introduced. Moreover, this Thesis presents a solution for measuring the quality of integration and for managing the shared resources on multiprocessor system on chips.

The benefit of hardware acceleration to system performance is measured and analyzed in the video encoder. Furthermore, measurements cover the performance of hardware accelerators before and after the integration. The hardware-accelerated system encodes 18 frames per second, which is 40% more than in the system with only processors.

It was discovered that the accelerators perform worse as a part of the full video encoder than alone in simulation environment. In addition, the hardware accelerator that performed better in the simulation environment performs worse in the video encoder. The reason is in integration overheads, which reduce the frame rate from theoretical 21 frames per second. The integration overheads were analyzed and divided into factors. The proportions of integration overhead were 96% and 55% of the total execution time of the accelerated ME and DQ functions, respectively. Shared resource contention was the largest factor and it consumed 56% of ME and 33% of DQ execution time. The two other factors are low-level driver software delays and data delivery expenses.

The integration overhead factor proportions of execution time depend on the accelerator and the system. They cannot be fully generalized. However, identifying and analyzing them is the most important achievement in this Thesis. These integration overhead factors can be used to analyze and evaluate any hardware-accelerated system.

## TIIVISTELMÄ

TAMPEREEN TEKNILLINEN YLIOPISTO

Tietoliikenne-elektronikan koulutusohjelma

**RASMUS, ANTTI TAPIO:** Laitteistokiihdyttimien integrointi videonpakkausjärjestelmään järjestelmämikropiirillä

Diplomityö, 74 sivua

Kesäkuu 2007

Pääaine: Digitaali- ja tietokonetekniikka

Tarkastajat: professori Timo D. Hämäläinen, dosentti Marko Hännikäinen ja diplomi-insinööri Ari Kulmala

Avainsanat: laitteistokiihdytys, videonpakkaus, integrointi, järjestelmämikropiiri, FPGA

Työn tavoitteena on suorittaa laitteistokiihdyttimien integrointi järjestelmämikropiirille ja tutkia mitä lähestymistapoja ja vaiheita integrointiin liittyy. Lisäksi keskeisessä osassa on integroitujen laitteistokiihdyttimien vaikutus järjestelmän suorituskykyyn. Työssä suoritetaan tapaustutkimus integroimalla kaksi laitteistokiihdytintä monta rinnakkaista prosessoria hyödyntävään videonpakkausjärjestelmään, joka on toteutettu FPGA-piirillä. Laitteistokiihdyttimistä ensimmäinen suorittaa liikkeenestimointia (ME) ja toinen diskreettiä kosinimuunnosta, kvantisointia ja näiden käänteisfunktioita (DQ).

Laitteistokiihdyttimien lisääminen järjestelmään on monimutkaista, sillä laitteistokiihdyttimien toiminnallisuus ja rajapinnat tulee saada toimimaan tehokkaasti sekä järjestelmätasolla että signaalitasolla. Lisäksi tulee ratkaista, kuinka laitteistokiihdyttimet näkyvät sovellusohjelmalle ajuriohjelmiston kautta. Työssä tutkitaan mahdolliset integrointistrategiat ja todetaan tässä tapauksessa liitällohkojen suunnittelu sopivimmaksi lähestymistavaksi. Työ esittelee integroinnin toteutuksen ja liitällohkot. Lisäksi työssä on ratkaisu integroinnin toteutuksen mittaamiseen ja moniprosessorijärjestelmän jaettujen resurssien hallintaan.

Työssä mitataan ja analysoidaan integroitujen laitteistokiihdyttimien suoriutuminen videonpakkausjärjestelmässä sekä määritetään kiihdytyksestä saatu hyöty sovellukselle. Tuloksena syntynyt laitteistokiihdytetty järjestelmä pakkaa 18 kuvaa sekunnissa, mikä on 40% parempi kuin alkuperäinen yksinomaan prosessoreihin perustuva järjestelmä.

Tarkemmassa tarkastelussa havaitaan laitteistokiihdyttimien suoriutuvan järjestelmässä paljon huonommin kuin yksinään testiympäristössä. Lisäksi testiympäristössä paremmin pärjännyt kiihdytin suoriutuu huonommin integroidussa järjestelmässä. Syyksi havaitaan integroinnista aiheutuvat kustannukset, joita ilman saavutettaisiin 21 kuvan sekuntivauhti. Integroinnin kustannukset on analysoitu ja jaettu osatekijöihinsä. Kustannusten osuus operaatioiden suoritusajoista oli ME:llä 96% ja DQ:lla 55%. Suurin kustannustekijöistä molemmilla oli jaettujen resurssien ruuhkautuminen, joka söi suoritusajasta ME:llä 56% ja DQ:lla 33%. Kaksi muuta tekijää ovat matalan tason ohjelmistosta ja tiedonsiirrosta aiheutuvat viiveet.

Integrointikustannusten osuus suoritusajasta on kiihdytin- ja järjestelmäkohtaista, eikä tuloksia voida täysin yleistää. Kuitenkin integrointikustannustekijöiden tunnistaminen ja analysointi on työn keskeisin tulos. Niitä voidaankin käyttää minkä tahansa laitteistokiihdytyksen analysointiin ja arviointiin. Tällä tavoin löydetään suurimmat integroinnista aiheutuneet pullonkaulat.

## PREFACE

The work for this Thesis was carried out in Institute of Digital and Computer Systems at Tampere University of Technology in 2006-2007.

I would like to my sincere gratitude to my thesis supervisors Professor Timo D. Hämäläinen and Docent Marko Hännikäinen. I am particularly grateful to M. Sc. Ari Kulmala and M. Sc. Erno Salminen for their guidance and ideas concerning the work. In addition, special thanks to M. Sc. Olli Lehtoranta and M. Sc. Jarno Vanne for their expert help in video related issues.

I wish to thank my parents Reijo and Sinikka as well as my sister Anna-Mari for constant support during this work and my studies. I am thankful to all my friends for memorable undergraduate days, especially Sanna Hautala and Esko Laaksonen for spurring on to write this Thesis.

Tampere, 24<sup>th</sup> May 2007

Antti Rasmus

## TABLE OF CONTENTS

1.	INTRODUCTION .....	1
2.	INTEGRATION APPROACHES.....	3
2.1.	Integration strategies .....	5
2.2.	General integration flow .....	7
3.	HIBI-BASED SOC ARCHITECTURE.....	9
3.1.	Heterogeneous IP Block Interconnection.....	9
3.2.	SoC architecture implementation on Altera FPGA.....	12
3.3.	Environment and tools for verification and prototyping.....	14
4.	ARCHITECTURE FOR VIDEO ENCODER .....	16
4.1.	Basics of video compression.....	16
4.2.	Data parallel video encoding.....	17
4.3.	Partitioning for multiprocessor configuration.....	20
4.4.	FPGA implementation of the parallel video encoder.....	21
4.5.	The profile of the software video encoder .....	22
5.	HARDWARE-ACCELERATED VIDEO ENCODER.....	24
5.1.	New IP blocks .....	24
5.2.	Accelerated system .....	30
5.3.	Integration of HW accelerators .....	31
5.4.	Lessons learned .....	35
6.	IMPLEMENTATION OF WRAPPERS AND RESOURCE MANAGER.....	37
6.1.	DQ hardware accelerator.....	37
6.2.	Motion estimation hardware accelerator.....	38
6.3.	Wrapper component for DQ.....	41
6.4.	Wrapper component for Motion estimation.....	46
6.5.	Resource manager .....	55
7.	PERFORMANCE ANALYSIS .....	57
7.1.	Execution time and performance .....	57
7.2.	Clock frequency and chip area.....	60
7.3.	Video encoding speed .....	61
7.4.	DQ execution .....	62
7.5.	ME execution time .....	64
7.6.	Factors of the integration overhead.....	65
7.7.	Area usage on FPGA.....	67
7.8.	Performance per area.....	68
8.	CONCLUSIONS.....	70

## ABBREVIATIONS

ASIC	Application specific integrated circuit
CIF	Common intermediate format
CPU	Central processing unit
DCT	Discrete cosine transform
DMA	Direct memory access
DQ	Discrete cosine transform, quantization, inverse quantization, inverse DCT
EDA	Electronic design automation
FIFO	First in, first out
FLI	Foreign language interface
FPGA	Field programmable gate array
FPS	Frames per second
FSM	Finite state machine
HDL	Hardware description language
HW	Hardware
IDCT	Inverse discrete cosine transform
IO	Input/output
IP	Intellectual property
LE	Logic element
LED	Light emitting diode
ME	Motion estimation
MPEG	Moving picture experts group
NRE	Non-recurring engineering
OCP	Open core protocol
PLL	Phase locked loop
QCIF	Quarter common intermediate format
RGB	Red-green-blue
RISC	Reduced instruction set computer
RAM	Random access memory
RTL	Register transfer level
RM	Resource manager
ROM	Read-only memory
SAD	Sum of absolute difference
SDRAM	Synchronous dynamic random access memory
SoC	System on chip
SP	Simple profile
SPMD	Single program multiple data
SRAM	Synchronous random access memory
SW	Software
TX	Transmit
UART	Universal asynchronous receiver and transmitter
UML	Universal modeling language
VLC	Variable-length coding

# 1. INTRODUCTION

Most contemporary embedded systems include a *system on chip* (SoC), where memory, processor, and other logic are integrated and interconnected on a single chip, instead of using several discrete chips. SoC-based design reduces overall costs, power consumption, and size.

Using several processors (central processing unit, CPU) and hardware (HW) accelerators is a way to meet the performance requirements. Multiple processors increases performance, but complicate the system. Hardware accelerators are dedicated to executing a particular operation without programmability. However, not every operation is suitable for acceleration, and hardware design cycle is more time and resource demanding than software development.

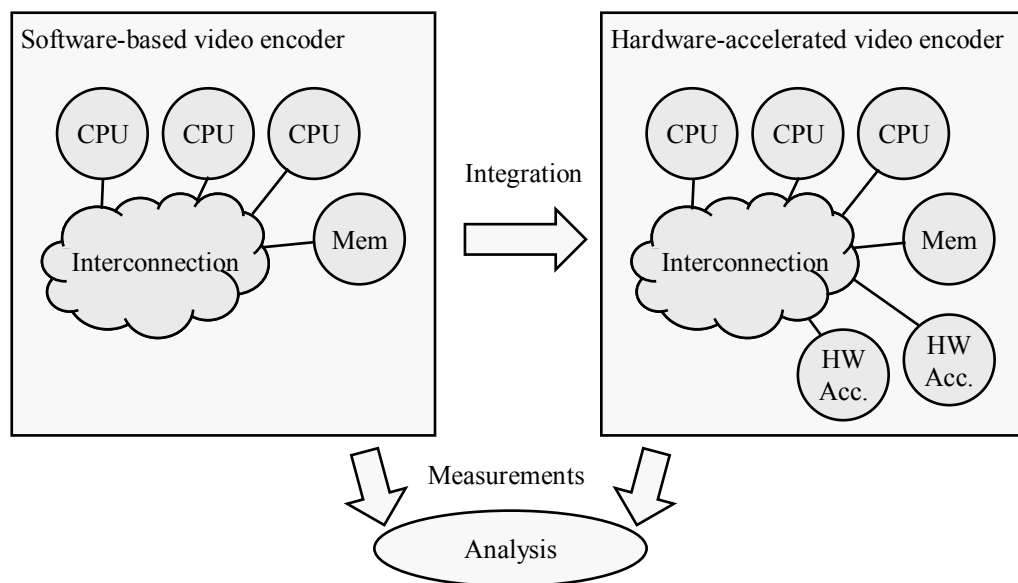
Although hardware accelerators have downsides, they can perform the same task tens or hundreds times faster than general-purpose processor [1]. In addition, processors are not energy or area efficient when compared to hard-wired processing elements [2]. This makes hardware accelerators a favorable option for battery-powered electronic devices.

In order to use new hardware accelerator components inside a SoC, one needs to integrate them. The signal-level interface of the new component must fit to the SoC interconnection network, and functionality must match with the rest of the system. Usually this is not the case, and integrator has to adapt the system, component, or both. Thus, there are several ways to perform the integration. Moreover, integration may decrease the performance of the introduced components due to *integration overheads*, and the original goal concerning the speed-up may not be reached.

In order to study the integration and its side effects, this work presents a case study of integrating two hardware accelerators into multiprocessor SoC based video encoder. Video encoder was chosen to the case study, since it is performance critical application. In mobile multimedia devices, the video must be encoded and decoded in real-time (25 frames per second) with cost-effective devices that also have to be energy-efficient and small. However, the user requirements for the video resolution and quality are increasing, which calls for higher performance. For instance, a non-optimized real-time MPEG-4 encoder for quarter common intermediate format (QCIF) sized video requires

performance of 1500 million instructions per second (MIPS) [3]. For instance, a Pentium II 300 MHz provides about 1000 MIPS, and Altera's embedded processor Nios II (185 MHz) 218 MIPS [4].

Video encoder execution is measured before and after the integration and analysis is carried out based on the results. Originally, the video encoder has been implemented purely on software and it utilizes several interconnected processors on field-programmable gate array (FPGA) chip. After the integration, the hardware accelerators perform five operations of the encoding. Moreover, a *resource manager* is implemented in the hardware-accelerated system to help sharing the accelerators for processors. Figure 1 summarizes the applied methodology of studying effects of integration in this work.



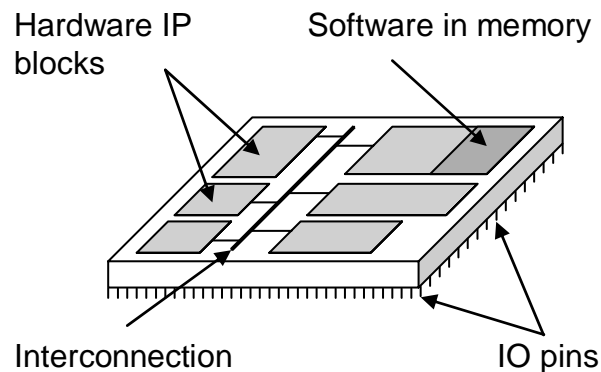
**Figure 1. Case study of this Thesis: Two hardware accelerators are integrated into software-based video encoder resulting in a hardware-accelerated video encoder.**

This Thesis covers the applicable integration strategies, implementation of the most suitable strategy, additionally required functionality besides the hardware accelerators, and arisen performance overheads. In addition, Thesis introduces the SoC architecture of the video encoder, and several new hardware components. Moreover, analysis of the integration is carried out based on the measurements done before and after the hardware acceleration.

Thesis first discusses integration strategies in Chapter 2. Moreover, it introduces the used system-on-chip architecture in Chapter 3. Chapter 4 presents the video encoder functionality and mapping to the architecture. The Chapter 5 describes the hardware accelerated video encoder, introduces the new components, and discusses how the integration was implemented. Chapter 6 covers technical aspects of the new blocks. The results of the integration and analyses are presented in Chapter 7, and Chapter 8 concludes the work.

## 2. INTEGRATION APPROACHES

“A *system on chip* is defined as an integrated circuit that implements most or all of the functions of a complete electronic system.” [5] Integrated circuit is a packed component that contains several electrical components, such as semiconductors, resistors, capacitors, and inductors, attached to each other. To be able to communicate with environment, the chip has also input and output (IO) pins. Different functions are typically implemented with software on general-purpose or digital signal processors, or with dedicated hardware IP blocks. IP blocks, which also include the processors, communicate with each other on the chip. For this purpose, there must be an interconnection logic between them. Figure 2 illustrates an all-digital system on chip that has several hardware IP blocks, program code and data in on-chip memory, a on-chip interconnection, and IO-pins to connect the chip to its environment, a printed circuit board.



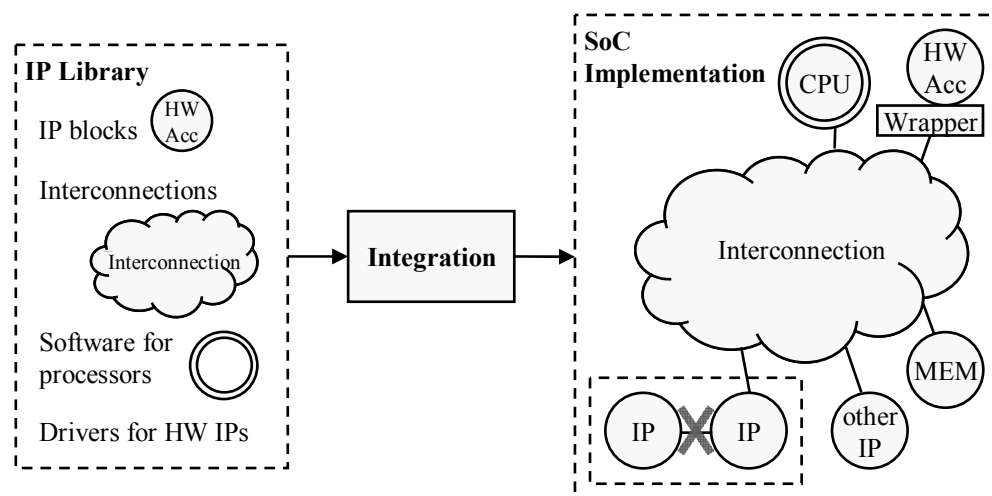
**Figure 2. Conceptual contents of an all-digital system on chip.**

*SoC architecture* means the relation between different IP blocks and the interconnection. It can be generic and application independent, but may contain features that are characteristic to some applications. SoC architecture guidelines include separation between computation and communication [6] and between function and architecture [7][8]. *Communication architecture* defines the topology and structure of the on-chip interconnection.

The chip size, complexity, and application requirements are ever increasing in semiconductor industry. This has driven the design engineers to develop methodologies and techniques to manage the design process and increase productivity. In *IP based design* [9], reusable IP blocks from IP library are used to build SoCs. Design time can be saved because pre-verified and already available components from IP library are reused, and therefore there is no need to redesign the blocks.

In IP based design approach, the IP blocks are chosen from the IP library and connected into an implementation of the system. The IP library contains hardware IP blocks, interconnections, and high and low-level software components. Because not all IP blocks and interconnections in the library are compatible, wrapper blocks have to be created to adapt interfaces and functionality. Wrapper blocks also have alternatives that are discussed in the next section.

The process of connecting and adapting blocks is called *integration*. It is depicted in Figure 3. This approach suits best for SoC architectures where all IP blocks are interconnected by one global communication network as in Figure 3. This way the resulting SoC architecture is independent of type or number of blocks, and thus flexible and modular. However, direct connection between two IP blocks is not supported without interconnection from IP library.



**Figure 3. The IP integration process.**

IP blocks are delivered in three different main forms: soft, firm, and hard [10]. Soft IPs are delivered using some hardware description language (HDL) that is synthesizable and usually parametrizable. This offers the integrator the flexibility to change the IP and reuse it. Moreover, it can be used on any silicon technology. The drawback of soft IPs is the unpredictability of performance, since the timing or power characteristics may not be fully guaranteed. The results depend on the possible modifications, synthesis tool, and target technology.

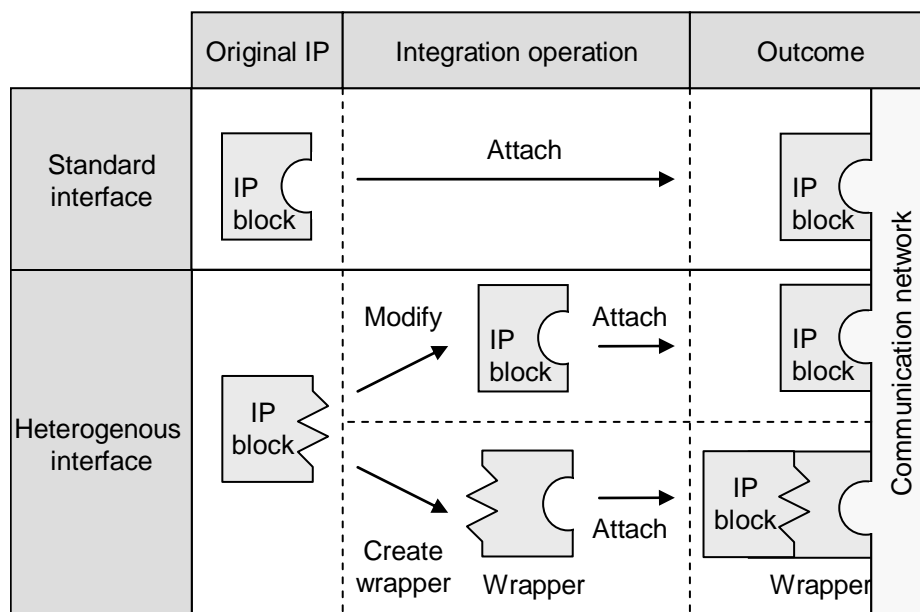
Hard IP blocks are optimized for a target technology with predefined transistor level layout. Therefore, they lack flexibility and are limited to certain chip manufacturers.

However, the timing and power consumption are strictly defined and performance is usually better.

Firm IP blocks are in-between the two previous ones. They may contain some parameters, but usually they are fixed. In addition, their layout is flexible. Firm IPs offer compromise between performance of hard and flexibility of soft IP blocks.

## 2.1. Integration strategies

Wagner *et al.* [9] have proposed three main strategies for integrating IP blocks, and they are based on a separation between communication and computation for each component. First, in the *standard-based strategy* the interfaces of the IP blocks are compatible with a standard. Second, in *IP derivation strategy* the IP source code is altered to fit the IP block to the communication network on chip. This is only possible when considering the soft IP blocks. Third, the *IP communication synthesis* is strategy where interface adapters, also called wrappers, are generated and inserted between the components and the network. The three strategies are depicted in Figure 4, where the edge shapes represent different interfaces.



**Figure 4. Different integration strategies.**

The standard-based strategy offers easy integration as the components fits directly to each other and everybody knows the standard. This improves reusability, since the components do not need modifications. This suits any of the three IP types. However, by adapting an application specific component to a standard interface, it might reduce performance since the standard has to offer universal compromise for every type of applications. Moreover, some of the standards come with flexible interface that can be reduced or extended according to the target application. This may lead to a situation where two blocks implement the same standard but do not fit to each other because

another uses extended or reduced interface. In addition, there exist multiple solutions for IP interface standardization. The most important ones are: AMBA [11], IBM's CoreConnect [12], OCP [13], Wishbone [14], and Silicon Backplane MicroNetwork [15]. This increases the possibility of incompatible interfaces of an IP block and the communication network.

The IP derivation is about modifying the original IP block to adapt to the interface of the communication network. This requires the modification of the possibly unfamiliar source codes of the IP. In addition, the documentation concerning the source code may not be complete. It is likely that the modifications cause problems in the functionality. At least the IP block has to be re-verified. As the interface is modified, the original testbenches of the IP may not work. Another weakness is the reusability issue. To fit three different interfaces, the IP block has to be modified three times. Maintaining three different versions of the IP block is awkward, if the IP provider publishes an updated version of the original IP block.

The automated communication synthesis generates a wrapper component between the IP block and the communication network based on the formal interface specifications. The communication network provider and the IP block vendor model their interfaces, for example in universal modeling language (UML) or in extensible markup language (XML). For instance, the latter is used in IP-XACT from the SPIRIT Consortium [16]. Such tools enable fast and easy integration. However, there are some practical limitations in this approach. Firstly, it is currently impossible to apply because such synthesis tools only exist in experimental academic environments and no commercial tools exist. Secondly, the IP block models might not be accurate enough or they might be modeled with a modeling language that the tool does not support. Lastly, this requires the learning of the modeling language.

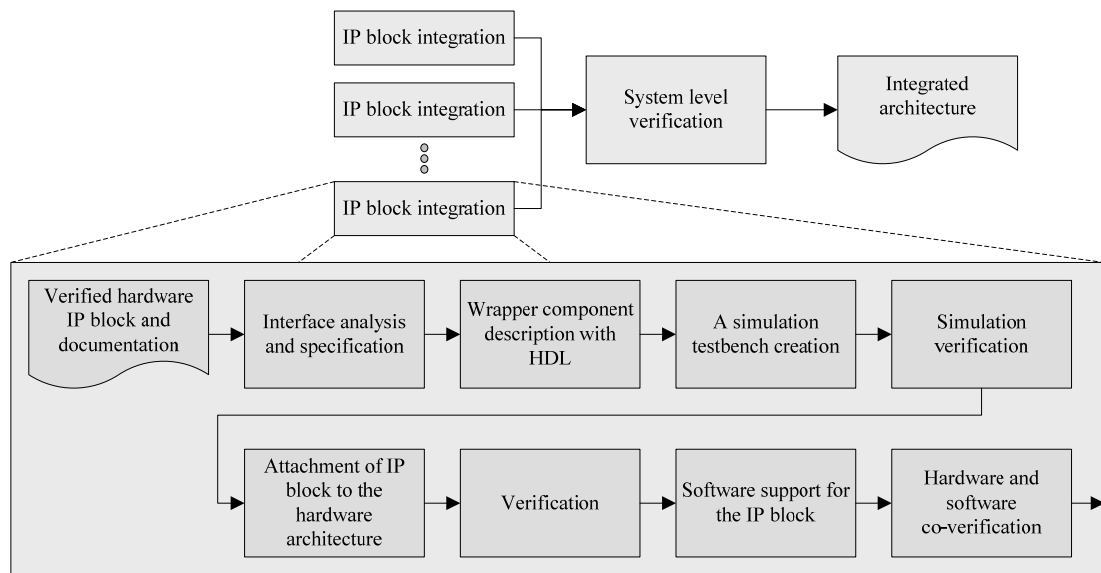
Koski [17] is a design flow for multi-processor SoCs with UML front-end. The flow optimizes a system by selecting IP blocks and on-chip network from a library. However, such design flow requires the components to be compliant with the on-chip network interface. For instance, OCP compliant interfaces are supported. Still, any third party computational component, such as hardware accelerator, cannot be directly added to the library. Currently, this requires manual design of a wrapper component.

Automated communication synthesis is not yet an applicable solution, standard-based strategy is not yet fully adopted, and IP derivation is limited to soft IP blocks only. Hence, in this work wrapper blocks are created manually. This approach suits also to hard and firm IP blocks, and it improves reusability and maintainability by leaving the original IP untouched. Further, it is possible to design several wrapper blocks to each of the interface to be adapted. Moreover, a wrapper block can be created to adapt an IP block interface to a standard interface and apply standard-based strategy. Downside is the manual work required to design the wrapper block. However, since the automated tools are immature, manual work is inevitable at the moment.

## 2.2. General integration flow

Integration can be divided into two levels: higher system-level integration and lower block level integration. System-level integration concentrates on selecting IP blocks and communication architecture for the IP blocks, and verifying the system. Hardware IP blocks and high-level software components must meet the performance and functional requirements of application, whereas interconnection must provide reliable and fast enough communication between the IPs. After integration, the co-operation is verified in the system level. After verification, the integrated SoC architecture is ready.

Block level integration covers all other tasks like interface adoption, IP block attachment, testbench design, and verification. The block level integration generalizes easily. Based on the manual wrapper design for interface adapting, a block level integration flow is presented in Figure 5 as a part of the system-level integration flow.



**Figure 5. General system-level integration flow containing also the block level integration.**

The IP block integration flow starts when the pre-verified hardware IP block and documentation are ready. The documentation should contain at least the interface description, operation manual, and an example of instantiation.

At the first phase, interfaces to the IP block and to the communication network are studied and analyzed. This includes the specifying and designing of the wrapper component. Inaccurate specification might cause the IP block to malfunction and poor design increases debug time.

The designed wrapper component is written in a synthesizable HDL. At the same time or afterwards with wrapper component description, a simulation testbench will be created to verify the implementation functionally. The simulation testbench would ideally consist of two sub-testbenches: one that verifies the interface towards IP block and another that verifies the interface towards the communication system. This way,

half of the testbench can be reused, if either the same IP component will be integrated to different communication network, or another IP component will be integrated to the same communication network. To increase the testbench coverage, it is recommended that different persons create the testbench and the design under verification, which in this case is the wrapper component.

Verification is in the major role in digital designing. Evans *et al.* have discovered that well over 50 percent of design effort between the start of design and start of very-large-scale integration (VLSI) layout is spent on verification or related tasks [18]. To minimize effort spent on verification, verification is started as early as possible, since the errors and bugs are harder to find in bigger and complex designs. Functional verification is practically simulation where a set of stimuli is input to the block or system and the response is compared to a reference. Stimuli are chosen to cover as many test situations as possible. Timing verification can be static or done with simulation or prototype. However, the simulation is not as accurate as prototype. Electronic design automation (EDA) tools provide automated static timing analysis to check critical paths and to determine maximum clock frequency. Timing simulations need technology specific netlist, component library, and routing delays.

After the wrapper block has been verified, it will be attached to the communication network. In manual integration, this usually means adding the new components to the HDL description of the system. If the wrapper component interface is designed right, this phase is signal assigning, which is very straightforward operation.

The attached hardware component consisting of IP block and wrapper should be verified functionally again in the hardware platform before starting software integration. If the hardware is verified, it is easier to find errors and bugs in software integration, since the hardware is bug-free. The verification here can be made using simulation model of the SoC architecture, or even a field programmable gate array prototype with a processor or dedicated unit to verify the IP block functionality.

Since embedded software interacts with the integrated IP blocks, the modifications in the hardware dependent software can be done after the hardware verification. However, software development can be started as soon as the specification of wrapper component is ready. This design methodology is called hardware-software co-design and it enables rapid development of embedded systems. However, in order to utilize the methodology thoroughly, there must be a model of the hardware component to allow initial verification of software.

System level verification means the final verification after all components have been verified separately. It contains running of the target application or a testbench. Here, both the software and the hardware are verified together. The term co-verification is used to describe the process.

## 3. HIBI-BASED SOC ARCHITECTURE

This chapter introduces the SoC architecture used in the video encoder system. It is necessary to understand the architecture, in which integration occurs. Especially the interconnection, processors, and memories are important.

The SoC architecture comprises the guideline to separate communication from computation. For communication, the SoC architecture uses *Heterogeneous IP Block Interconnection* (HIBI) [19]. Both the SoC architecture and the interconnection have been developed in Institute of Digital and Computer Systems at Tampere University of Technology. The chapter introduces HIBI communication architecture and an implementation of the SoC architecture on Altera FPGA chip.

### 3.1. Heterogeneous IP Block Interconnection

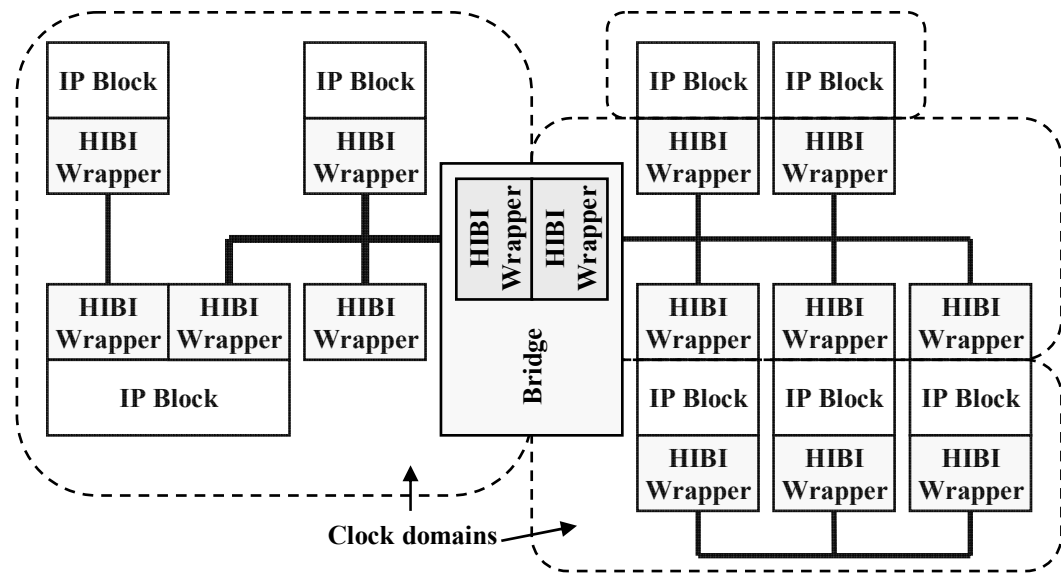
The HIBI is targeted for complex SoC designs. It is technology independent and aims at maximum efficiency and minimum energy in transfers. It supports hierarchical topologies and several clock domains and scales flexibly. In addition, it can be reconfigured at runtime. Moreover, it implements a first-in-first-out (FIFO) interface.

Hierarchical topologies and multiple clock domains save energy and area, since IP blocks with low requirements can use a segment with narrower bus width and lower clock frequency. In addition, IP blocks that exchange much data can be inserted into the same segment thus reducing system-wide interconnection load. Moreover, flexible communication architecture enables rapid upgrading and modifying of the application.

#### Topology

The HIBI architecture consists of HIBI wrappers that are attached to IP blocks. The HIBI wrapper can be used with OCP. This is important feature because it eases the use of third party IP blocks with HIBI. HIBI wrappers are connected to each other as shown in Figure 6. It depicts an example of irregular HIBI network with multiple segment and clock domains. There is a point-to-point link on left, hierarchical bus in the middle divided by the bridge, and a multibus topology around three rightmost IP blocks. In addition, the segment left to the bridge is wider than the others. Moreover, dotted lines

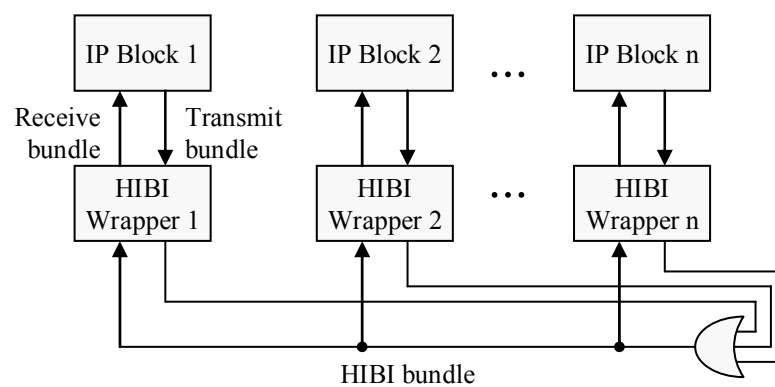
depict four different clock domains. The number of clock domains is not limited, but every wrapper in a segment must use the same clock.



**Figure 6. An example of a hierarchical HIBI network with multiple clock domains and bus segments [19].**

### Structure of HIBI segment

Figure 7 shows how IP blocks and wrappers are interconnected inside a HIBI segment. Every IP block communicates with a wrapper block. All HIBI wrappers are connected to each other with OR port. Thus, HIBI is OR-resolved bus. The interface between HIBI wrapper and IP block is simple bidirectional FIFO interface. There are separate FIFO buffers for receiving and transmitting data.



**Figure 7. Connecting IP blocks to a HIBI Wrapper.**

*Transmit bundle* in Figure 7 consists of signals that is used to send data from IP block to HIBI wrapper, whereas *Receive bundle* is used to read data from HIBI wrapper to IP block. An IP block works as a master when operating with HIBI wrapper. It initiates transmissions and receptions. This means that it reads the receiving FIFO buffer and writes to the transmission FIFO buffers, which ensure smooth data delivery and implement the efficient transmissions in bursts.

The most relevant parameters of HIBI wrapper are address, priority, and data width, that is, word size. As there are multiple HIBI wrappers along the bus, sender IP has to specify the target IP. This is done by address word that is always sent first, and data words, the payload, follow the address. Address consists of two parts. The most significant bits of the address define the target component on HIBI, whereas the other bits are for internal use of the target component. They may specify subcomponent or deliver information. The address decoding is distributed, so every HIBI wrapper listens for its own address on the bus.

Data width is an important factor of throughput of the interconnection, but it does not affect the latency of one-word-long transmission. Large word size increases buffers and thus the silicon area.

The meaning of priority parameter depends on the arbitration algorithm in use. The simplest algorithm is round-robin, where HIBI wrappers take turns to send data to each other. In more advanced algorithms, higher priority wrappers can send their data first. This, however, may cause starvation to the lower priority HIBI wrappers if one of the higher priority wrappers does not give up its turn [20].

Table 1 describes the signals presented in Figure 7 in more detailed. For every type of signal, an OR-port combines signals coming from the HIBI wrappers and delivers the output to every wrapper. The *HIBI bundle* has five signals. Signals ending with *\_out* come from HIBI wrapper and signals ending with *\_in* go to the HIBI wrapper. Data bus width  $W$  is parametrizable. In addition, there is clock signal *clk* and reset signal *rst\_n*.

**Table 1. The signals connecting IP blocks and HIBI wrappers.**

Signal name in HIBI bundle	Receive signal in IP block	Transmit signal in IP block	Width [bits]	Purpose
bus_comm_*	agent_comm_in	agent_comm_out	3	Command
bus_data_*	agent_data_in	agent_data_out	$W$	Data or address word
bus_av_*	agent_av_in	agent_av_out	1	Is the word in data-bus an address?
bus_lock_*			1	Informs other HIBI wrappers that the bus is in use
bus_full_*		agent_full_in	1	Are buffers full in HIBI wrapper?
		agent_we_out	1	Agent writes data to HIBI wrapper
	agent_re_out		1	Agent reads data from HIBI wrapper
	agent_empty_in		1	Is the receive buffer empty in HIBI wrapper?
* There are two signals: one ending with <i>_in</i> and the other with <i>_out</i> . Out-signals are the ones that go to OR-port and In-signals coming from the port.				

### 3.2. SoC architecture implementation on Altera FPGA

The HIBI communication architecture can be used in any silicon technology. Thus, it can be used in optimized application specific integrated circuit (ASIC) designs as well as on FPGA. This work uses HIBI-architecture and number of IP blocks on Altera Stratix FPGA chip to implement SoC.

The main IP blocks in the SoC implementation under consideration are multiple processors and a memory controller. The architecture also contains a phase locked loop (PLL) for generating suitable clock signals. The configuration of components is easy to change and the number of IP blocks is parameterizable. This enables scalable performance since adding more processors to the system increases computational resources.

#### Processors

Soft-core processor means that the processor is implemented using high-level synthesizable description rather than predefined transistor layout specific to dedicated block. It must be synthesized to the target platform. Therefore, it is not technology specific and can be used both in ASIC projects and in prototypes on FPGA. Soft-core processors can be configured to contain different modules and interfaces in addition to the processor core.

The SoC architecture utilizes Altera's Nios soft-core processor family consisting of Nios [21] and Nios II [4]. Nios processors are 32-bit, general-purpose, soft-core, reduced instruction set computer (RISC) processors. They utilize separate instruction and data buses. Thus, they classify as Harvard architecture. Nios processors require Altera FPGA chip.

Depending on the configuration, Nios processors might contain

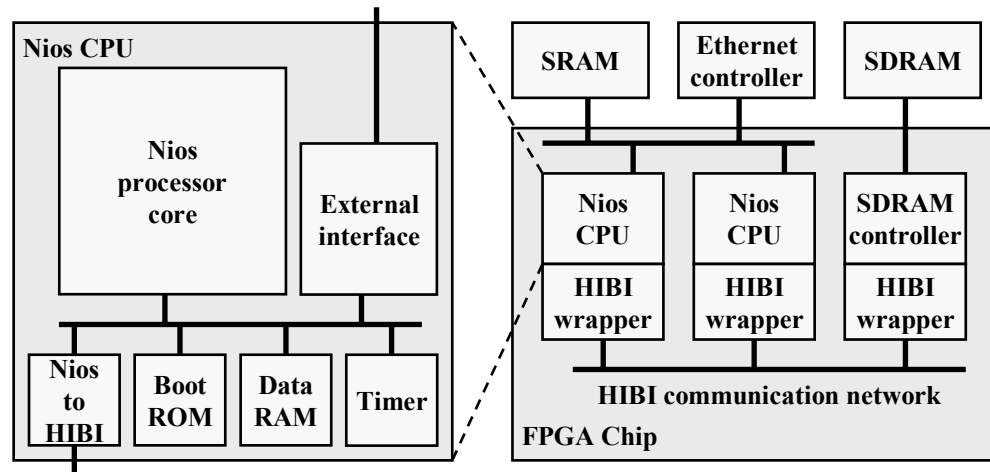
- data or instruction cache
- local data memory
- ready-only memory for boot
- timer and universal asynchronous receiver and transmitter (UART).

In addition, they might include interface for external peripherals such as

- synchronous random access memory (SRAM)
- synchronous dynamic random access memory (SDRAM)
- Ethernet controller
- light emitting diodes (LED) and push buttons.

Figure 8 depicts an example configuration of the SoC architecture on FPGA chip. There are two Nios processors and a SDRAM controller connected by HIBI communication network. The processors are directly connected to off-chip peripherals

SRAM and Ethernet controller, whereas the SDRAM is accessed through SDRAM controller. In this configuration, Nios CPU contains a core, timer, two internal memories, and an external interface. The cache memories reside inside Nios processor core.



**Figure 8. Example configuration of the architecture**

The Nios processors do not have native support for either the OCP or HIBI. Therefore, Nios processors in use have an extension called Nios-to-HIBI introduced in [22]. It provides HIBI compliant interface for the processors. In addition, it utilizes direct memory access (DMA) to allow parallel operation of the CPU core and the HIBI transmission. In other words, the CPU can execute an application while the Nios-to-HIBI transfers data on the background.

### The SDRAM memory controller

Since the chip does not provide enough memory for all applications, the SoC architecture utilizes an external off-chip memory. The SDRAM controller is attached to HIBI and used to access the off-chip memory through IO-pins of the chip. The controller has several read and write ports to provide memory access to multiple IPs simultaneously. However, only one off-chip memory operation can be performed at a time. Now, the SDRAM bus width is 32 bits and the capacity is 16 MB. Actually, every port of the controller is a buffer to ensure continuous utilization of relatively slow off-chip memory. This helps to cope with the memory bottleneck, as the SoC architecture does not use data cache.

The benefit of the SDRAM controller is that big data blocks can be read or written with a single request. The SDRAM controller increments the address automatically and allows even reading of two-dimensional blocks of data from picture memory. This way, the SDRAM controller reduces load on initiator IP and makes memory accesses more efficient.

To access the off-chip memory through the controller, the initiator IP has to request a read or write port from the controller. Next, the controller returns either a port number

or a value indicating that there are no free ports. After receiving the port, the initiator IP configures the read or write port by sending the starting address at the memory and dimensions of the read or write. After configuration, the IP either sends or receives the data. HIBI is responsible for relaying the communication of the initiator IP and the controller.

### **3.3. Environment and tools for verification and prototyping**

The SoC architecture was created using very-high-speed integrated circuit HDL known as VHDL [23], but there are also other alternatives as Verilog [24].

#### **Verification by simulating**

The purpose of simulations is to verify functionality and timing properties. Modelsim SE version 6.2a [25] from Mentor Graphics is used for this purpose. In addition, it supports C programming language-based verification through foreign language interface (FLI).

In order to verify dynamic timing conditions or functionality of gate level synthesis results using characteristics of target technology, the simulator needs synthesized netlist and technology library. In addition, routing information is required for verifying timing. In case of FPGA, netlist is a technology specific mapping of HDL assignments to logic elements in FPGA chip. Besides the netlist, verification needs technology libraries that contain information about the various logic elements in FPGA chip. These cell libraries include logical content and timing information of the logic elements. The cell libraries are shipped with Altera's Quartus II EDA tool.

#### **Verification on prototype**

Prototyping is the most efficient way of verifying design functionality in terms of speed. In this context, speed means the amount of different combinations of inputs verified in a time unit. With a prototype, for instance the same test sequence may take several times longer verification time compared to the actual hardware at ASIC. Further, emulators may take 10 to 100 [26] and simulations about 1 000 to 10 000 times longer time to verify the same sequence.

Whereas the ASIC prototyping is highly expensive due to the NRE costs, the FPGA prototyping is relatively cheap to perform. FPGA is a good trade-off between the ASIC prototype and the simulators. It is flexible, that is, reprogrammable, also cheap, and considerably faster than simulators or emulators.

In this work, Altera's Stratix Development Kit Pro Edition is used for prototyping. It includes Stratix EP1S40 [27] FPGA chip, the Nios II processor, and Quartus II design software. The software provides project management and complete toolset to manage flow from RTL to FPGA. This includes VHDL compilation, synthesis tools, as well as tools for creating chip layouts with placement and route operations.

---

The development board provides embedded logic analyzer through Quartus II interface, as well as physical Ethernet and UART ports. The embedded logic analyzer records the states of predefined signals for predefined amount of clock cycles. However, the embedded logic analyzer consumes logic elements (LE) and on-chip memory that limits the amount of signals and the time span to record.

## **4. ARCHITECTURE FOR VIDEO ENCODER**

The introduced general-purpose SoC architecture is adapted for parallel software video encoder application that acts as a starting point for the integration. Firstly, this chapter introduces basics of video compression and data parallel video encoding. Secondly, it describes how such video encoding is partitioned to multiple processors. Thirdly, it introduced a software video encoder implementation on FPGA. This software implementation acts as a starting point for the hardware accelerated video encoder that is covered in the next chapter.

### **4.1. Basics of video compression**

Raw video consists of adjacent pictures, frames, which are formed by multiple pixels in two dimensions. The raw video stream is displayed frame by frame, usually more than twenty frames per second in television and movie systems to obtain smooth perception of motion. Picture resolution defines how many pixels there are vertically and horizontally. In addition, each pixel consists of several bits that define the color the pixel represents, for example 8, 12, or 16 bits per pixel.

The raw video stream can be compressed either by lossy or lossless algorithms. Lossless video compression means that the original video can be fully recovered in decompression. It utilizes statistical information redundancy that always exists in the video. For instance, a fifteen-second-long QCIF (176x144 pixels) video sequence can be compressed losslessly into 9 800 KB with ZIP tool, when it takes 14 200 KB in raw format. This translates to 31% compression ratio. Instead, lossy video compression makes the encoded video smaller, but the original video cannot be fully reconstructed and distortion can be found.

Modern compression algorithms take advantage on imperfections of human eye. For example, color differences cannot be distinguished as accurately as differences in brightness and less bits can be used for storing the color information. Hence, although some data is lost during compression compression, most people do not notice much difference on video quality. Lossy algorithms are more common due to better compression ratio.

In general, there is a tradeoff between quality and compression ratio. The same example sequence was encoded to the size of 110 KB with lossy MPEG-4 algorithm, which equals 99% compression ratio. Thus, it requires the bandwidth of 8 KB/s to stream the sequence in real-time. For example, 3G mobile networks easily meet the requirement.

Pixel color can be represented in different ways. YUV color space is used in television systems, whereas red-green-blue (RGB) in cameras. In YUV, there is a luminance channel for brightness and chrominance for colors. Thus, it is possible to represent luminance using higher accuracy than chrominance and exploit the imperfections of human eye.

This work concentrates on lossy video encoding introduced in Moving Picture Experts Group's MPEG-4 Simple Profile (SP) standard, which is based on H.263 standard [28] that was developed for video conferences. The standard originally uses common intermediate format (CIF) and QCIF as input raw video format, but supports larger formats as well. The latter consists of 144 rows of 176 pixels and 12 bits per pixel. The color format in CIF and QCIF is YUV 4:2:0. The ratio defines the proportions of luminance and chrominance.

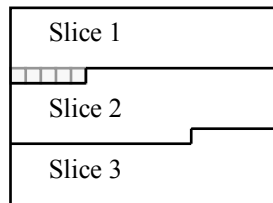
The lossy video compression is based on reducing redundancy in spatial and temporal domains. The H.263 standard uses *discrete cosine transform* (DCT) and *quantization* to remove spatial redundancy and *motion estimation* (ME) and compensation to remove temporal redundancy [29][30]. In H.263, every frame is divided into smaller sections. These sections are called macroblocks and they are square 16x16-pixel areas. This is the case in MPEG-4 SP as well. For instance, a QCIF-sized video sequence with resolution of 176x144 can be divided into 99 macroblocks. This results in 11 by 9 matrix of macroblocks. Encoder and decoder operate on macroblocks in order to make data handling more effective.

## 4.2. Data parallel video encoding

Nowadays, general-purpose processors use multiple processor cores to achieve more computation power and embedded multiprocessor SoCs are common. Although one powerful single core processor is enough to encode video without any parallelization, multiple processors or processor cores are needed for high-resolution video and new encoding algorithms. In order to utilize multiple processors or processor cores, the video encoding must be divided into computationally equal parts. The encoding process should also be scalable. Scalability enables the same encoding engine to encode computationally demanding high-quality video or save resources and energy by encoding low-quality video.

Video encoder can be parallelized using functional, instruction level, sub-word level, temporal, or spatial methods [31]. Only spatial and temporal methods provide high

scalability. However, temporal method induces latency to the encoding. Thus, this work concentrates to the spatial method. Granularity of one macroblock is used, which enables good scalability due to balanced load. The spatial method classifies as data-parallel meaning that the CPUs performs the same functions but operate on different data.



**Figure 9. Partitioning a frame into three slices horizontally.  
Few macroblocks are also drawn in the figure.**

The macroblocks of each frame are divided into slices. There are multiple ways to form the slices, but horizontal division is the most suitable for data parallel operation, since inter-slice motion prediction dependencies do not increase the complexity of the implementation, but only slight overhead in the bit stream [22]. The horizontal division is depicted in Figure 9.

The slices are assigned to processors, which encode the slices a macroblock at a time. The encoding of a macroblock consists of a few tasks that are discussed next. There are three lossless operations: motion estimation and compensation, DCT, and entropy coding. In addition, quantization is the lossy operation. Moreover, there are inverse operations for quantization and DCT.

### **Motion estimation and compensation**

In motion estimation, sections of two successive frames are compared in order to find similarities, because two consecutive frames usually contain almost uniform patterns but in slightly different places. The encoder goes through every macroblock of the current frame in row-wise order, and for every macroblock, it searches for the best matching macroblock from the reference frame that can be either preceding or subsequent. As picture might have moved slightly from the previous frame, the best matching macroblock is searched from the search area, which is also called reference area. In motion estimation, the encoder determines motion vectors that represent the relative distance to a best matching macroblock in the search area from the current macroblock.

*Motion compensation* follows the motion estimation. It constructs the best matching macroblock that is a macroblock of reference frame pointed by the motion vectors. Motion compensated macroblocks are used to calculate the prediction error of current and reference frame macroblocks by subtraction, since the best matching macroblock differs from the current macroblock. Prediction error of a macroblock is called residual.

The reference frame, a prediction error macroblock, and motion vectors have the same information than the current macroblock itself, but with less redundancy. Thus, fewer bits are needed to represent the current macroblock and the current frame. In other words, the next frame can be constructed from the previous one by copying the macroblocks to the places that motion vectors indicate and then add the prediction error macroblock. Both, encoder and decoder perform the motion compensation. The encoder uses the compensated macroblock to reconstruct frame, which is used as reference frame for the next encoded frame.

### **Discrete cosine transform**

*Discrete cosine transform* is a mathematical operation, which can be used to express the image in terms of amplitudes of different frequencies. These amplitudes are referred to as coefficients. Two-dimensional DCT operates on a block of  $N \times N$  samples and creates an  $N \times N$  block of coefficients [29]. In the case on transforming images, the samples are luminance and chrominance values of pixels. The operation has inverse function *inverse discrete cosine transform* (IDCT), which can be used to restore the original samples and hence the original image. The DCT or IDCT does not remove any redundancy. They just transform the image so that is more favorable to quantization.

### **Quantization**

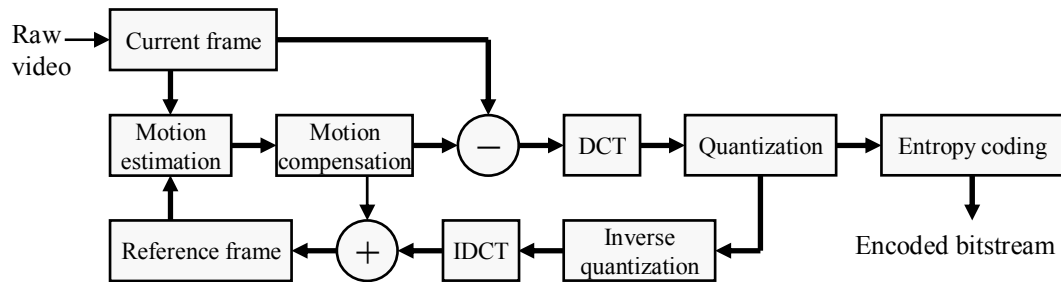
*Quantization* reduces the number of bits used per coefficient. Thus, bits can be saved and the image represented with fewer bits. Quantization is the only operation that causes loss of information. This is made by mapping every coefficient with one range of values to a quantized coefficient with another reduced range of values. For example, by mapping 8-bit value range to reduced 7-bit value range, one bit per coefficient would be saved. This implies 12.5% saving. In this case, the 7-bit range would utilize every second value of the 8-bit range. Quantization corresponds to integer division operation, where operands are divided by a coefficient and result is truncated. Inverse quantization is rescaling the reduced value range back to the original value range.

There are two reasons why the quantization is applied to the coefficients instead of pixel values. Firstly, the image quality remains better. Secondly, there are usually a considerable amount of the coefficients near to zero and thus can be quantized to zero. This improves the efficiency of further processing of the coefficients with entropy coder such as variable-length coding (VLC). For entropy coding, the two-dimensional coefficient matrix is converted into a one-dimensional array. The VLC can be efficiently applied if there are long runs of zeros. This is achieved with zigzag scanning where the coefficients are scanned diagonally starting from the upper left corner, where the low-frequency coefficients are and ending to the lower right corner [30].

### **Macroblock encoding**

A simplified frame encoding flow of MPEG-4 SP is represented in Figure 10. The uncompressed, raw, video in current frame is motion estimated and compensated

against reference frame. The residual frame is then put through the DCT, quantization, and entropy coder to create encoded bitstream. In addition, the motion vectors are included in the entropy coding and in the bitstream.



**Figure 10. Simplified encoding flow.**

The best matching macroblock and result of the quantization are reconstructed into the reference frame for the next current frame. The reconstruction contains also inverse quantization and IDCT that reverse the operations made to the residual. In addition, the best matching macroblock from the motion compensation is added to the reversed outcome. This common operating mode is called *inter coding*.

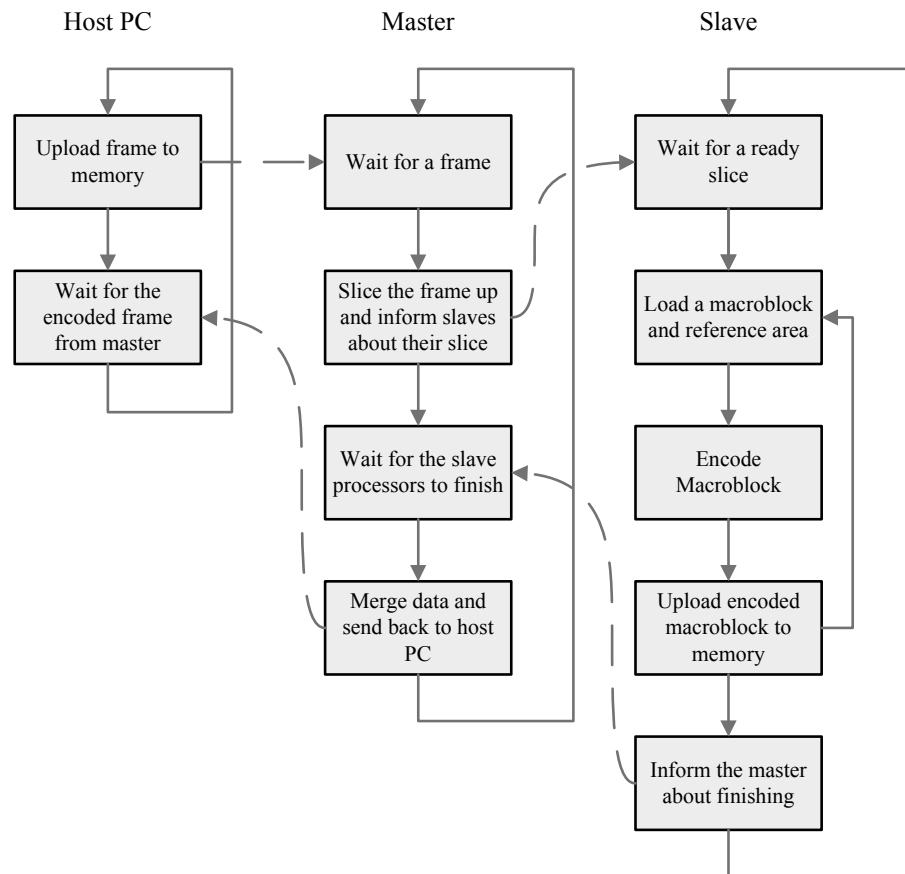
In case of poorly matched motion estimation, the best matching frame can be set to zero. In this case, the current frame is directly fed to the DCT, since subtrahend is zero. Again, the reference frame is directly the outcome of inverse operations, since addend is zero. This operation mode is called *intra coding*: no temporal redundancy is exploited. Motion estimation might match poorly, if the video stream contains big changes such as the change of scene. In the macroblock based encoding, a frame can consist of mixture of intra and inter coded macroblocks.

### 4.3. Partitioning for multiprocessor configuration

The previously introduced tasks of an encoder are suitable for parallel computation with multiple processors. In this spatial data-parallel approach, every processor performs all the tasks to the macroblocks of a slice. However, some tasks related to parallelization are easier for a single processor to perform. These unparallelizable tasks include, for instance, synchronization, centralized parameter configuring, frame slicing, and merging of encoded bitstream. These tasks are performed by one processor that is called master. All other processors are slaves and they do the actual encoding.

A master CPU and several slave CPUs implement the data parallel video encoding system. An external device, for instance a PC, provides the system with raw video and receives the encoded video. In this work, the system communicates with a PC via an Ethernet connection. The configuration provides easy way of delivering the data and decoding the result for comparison with the original. However, the raw video stream could also be originated from a camera or other source.

The relation between the PC, master processor, and slave processors is depicted in Figure 11. The master receives the raw video a frame at a time, stores it into the memory, partitions it into equal-size slices, informs slaves about memory address of their slice, waits for slaves to encode their slice, and finally merges bit streams encoded by the slaves and sends the merged stream back to the external device.



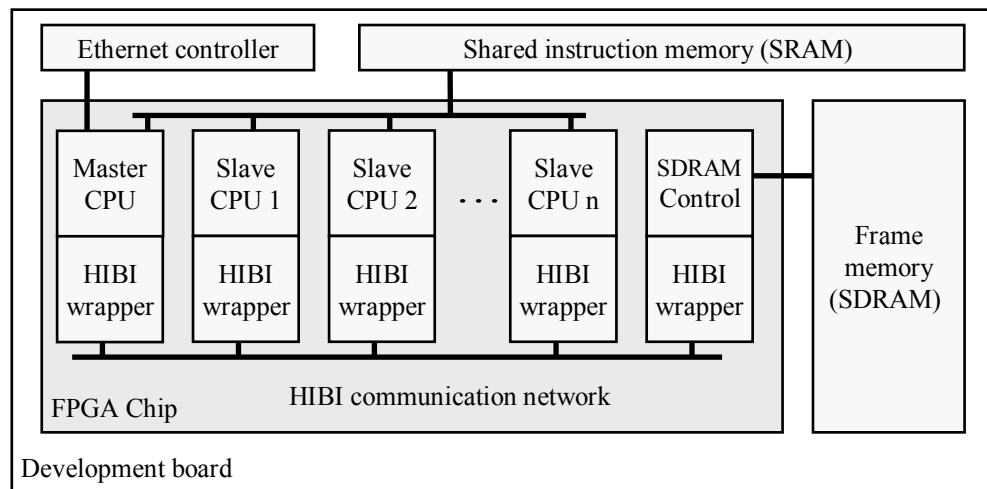
**Figure 11. The process of encoding video frame in the architecture.**

After a slave CPU has received a piece of information from master about its slice, it fetches one macroblock and corresponding reference area from the memory at a time and encodes the macroblock. The encoded macroblock, that is a bit stream, is then uploaded back to the memory. This is repeated until all macroblocks of the slice has been encoded. Then, the slave signals the master CPU about successful encoding.

#### 4.4. FPGA implementation of the parallel video encoder

The implementation of multiprocessor SoC encoder on FPGA uses Nios processors, HIBI communication architecture to interconnect them, and SDRAM controller to access external frame memory. There is the master processor and varying number of slaves. Figure 12 depicts the architecture topology for video encoder. All processors and the SDRAM controller are connected to each other through HIBI wrappers. For every generated slave CPU a HIBI wrapper is also created automatically. Multilevel hierarchy

is not exploited in the architecture. Instead, HIBI wrappers connect to each other directly. As the SDRAM has fixed width of 32 bits, the HIBI has been configured for the bit width of 32 as well.



**Figure 12. The SoC architecture for software video encoder.**

All the slave processors in the system are identical, so they use the same program code. To save memory, the processors share the instruction memory that is off-chip SRAM. This is called single program multiple data (SPMD) paradigm. They read the same program from the same memory but they have program counters of their own. However, a master processor has own program. The off-chip SRAM is split into two equal-size blocks for the programs for master and slaves.

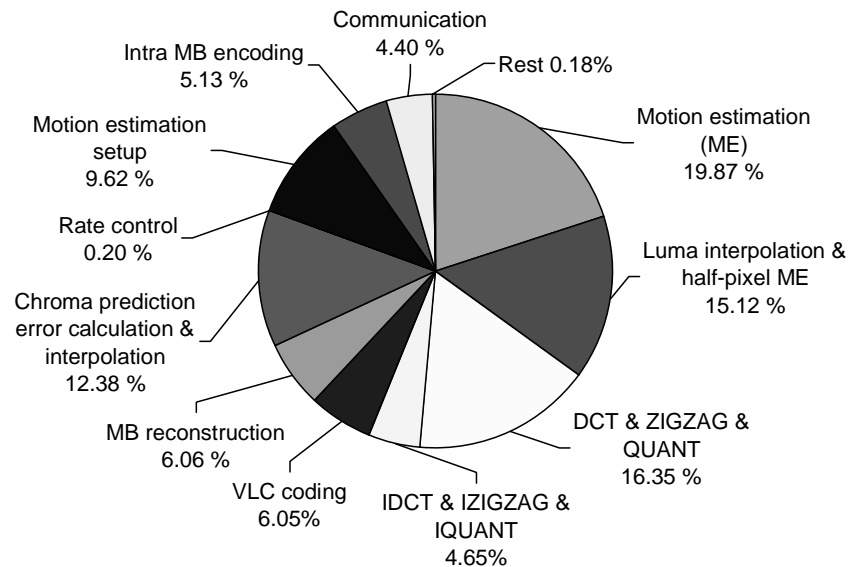
Whereas the slave processors are configured for efficient data processing and suitable for cloning, the master processor is provided with interface to the external Ethernet controller. Every processor has a fast local data memory, where they operate data. Local memory is of scratch-pad type, which means that transfer between it and the main memory are explicitly controlled from software.

The slave processors are configured to have 8 KB instruction cache, 64 KB local data memory, 2 KB of boot read-only memory, a timer, and UART. The timer is used for performance measuring purposes. In addition to slave CPUs, master processor contains another internal timer for timing Ethernet connection. All components of the encoder have a clock frequency of 50 MHz.

#### 4.5. The profile of the software video encoder

The processor-based software encoder on FPGA with three slave CPUs achieves the average frame rate of 13 QCIF frames per second. However, this is not enough for even real-time encoding. In order to increase frame rate, tasks running on slave processors need to be accelerated. However, one should only speed up the tasks that are the heaviest for processors to perform.

In order to find relative execution times of the tasks, the software execution was profiled [32]. A slave processor running the MPEG-4 SP encoder on the SoC architecture resulted in execution times presented in Figure 13. The motion estimation takes 19.9% and DCT, IDCT, quantization, inverse quantization, and zigzag scanning together take 21.0% of the total CPU time. In the latter, zigzag operation is relatively light to perform compared to the DCT and IDCT.



**Figure 13. Computation loads on different H.263 encoding tasks [32].**

The total execution time of DCT, IDCT, quantization, and inverse quantization was measured without the zigzagging in repeated study, and it was found out that they actually consume 15.8% of CPU time. In addition, the motion estimation consumed 21.2%. The differences between the two experiments are due to different test material and slightly different test configurations. The five operations, DCT, IDCT, quantization, inverse quantization, and motion estimation, take 37.0% of the overall execution time. Increasing the performance of these operations would have the biggest effect on the overall encoding performance. The frame rate of 21 could be achieved by reducing the execution time of the five operations to zero. This would not result in real-time encoder, but considerable speed-up to software encoder.

## 5. HARDWARE-ACCELERATED VIDEO ENCODER

This chapter discusses how the software implementation is accelerated. It briefly introduces the new hardware IP blocks and driver software that are integrated to the system. In addition, it presents the hardware accelerated architecture and its execution flow. Moreover, the implementation of integration is covered.

### 5.1. New IP blocks

In the software encoder, slave processors alone perform the encoding of macroblocks. In the hardware-accelerated encoder, the slave processors still do majority of the encoding process, although they use hardware accelerators to speed up the chosen tasks. In software profiling, it was found that DCT, quantization, inverse quantization, inverse DCT (DQ), and motion estimation consume most of the time and they are thus feasible to implement with hardware accelerators.

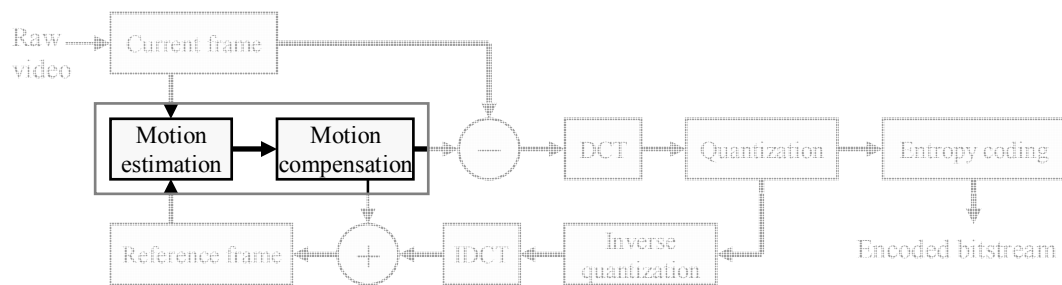
A number of new IP blocks have to be added to the encoder in order to accelerate these functions. Firstly, the hardware accelerators themselves are required. Secondly, the integration produces a wrapper block for each hardware accelerator. Thirdly, the hardware accelerators need a *resource manager* (RM) to let multiple processors access them safely. Fourthly, a hardware monitor is introduced for performance measuring purposes. Finally, all these components require own HIBI wrapper to interconnect.

The two hardware accelerators have been previously developed in Institute of Digital and Computer Systems at Tampere University of Technology. The wrappers, resource manager, driver software and hardware monitor have been implemented in this work. Next, the new IP blocks are introduced briefly. The accelerators and their wrappers are covered in Chapter 6 more thoroughly.

#### **Motion estimation hardware accelerator and wrapper**

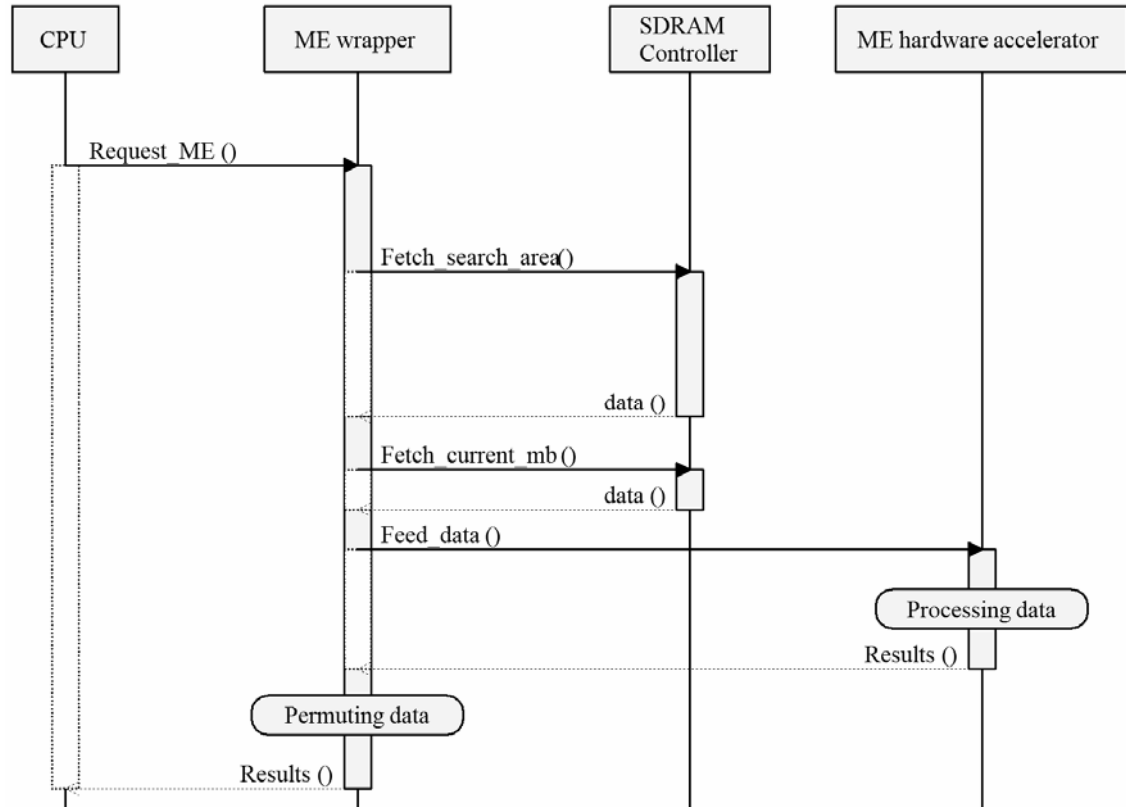
Motion estimation hardware accelerator is based on [33]. It finds best the matching macroblock from the search area. It outputs the sum of absolute difference (SAD) value

and motion vectors that define the upper right corner of the best matching macroblock in the search area. These are used in further stages of encoding. In addition, the hardware accelerator also includes functionality to do motion compensation. The operation results in the best matching macroblock. The functions that the hardware accelerator is responsible for is highlighted in Figure 14. Motion estimation is operated by feeding in a search area and current macroblock. The search area is from a reference frame and the current macroblock from current frame. Results come out when ready.



**Figure 14. The functions that motion estimation hardware accelerator implements.**

Instead of accessing the accelerator directly, processors use the ME wrapper to operate with the hardware accelerator. The wrapper has been designed to minimize processor interaction with the accelerated task. The processor only initiates the execution and receives the results, instead of constantly transferring data back and forth.

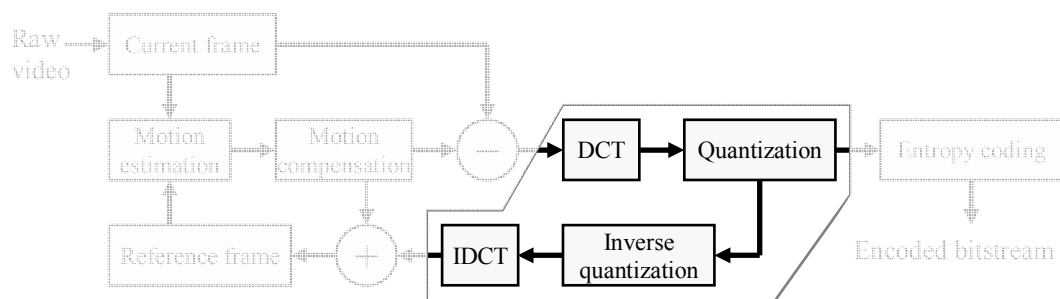


**Figure 15. The sequence diagram of interaction between processor, motion estimation hardware accelerator, its wrapper, and memory.**

The wrapper first fetches the data from SDRAM address that is indicated by the command from processor. The wrapper feeds the hardware accelerator with data. Thus, the wrapper implements protocol for accessing the SDRAM controller through HIBI. When the hardware accelerator finishes processing the data, it returns results to the wrapper. Since the results are in different order that the processor expects, the ME wrapper permutes the best matching macroblock before sending it back to the CPU. The process is depicted in Figure 15.

### DQ hardware accelerator and wrapper

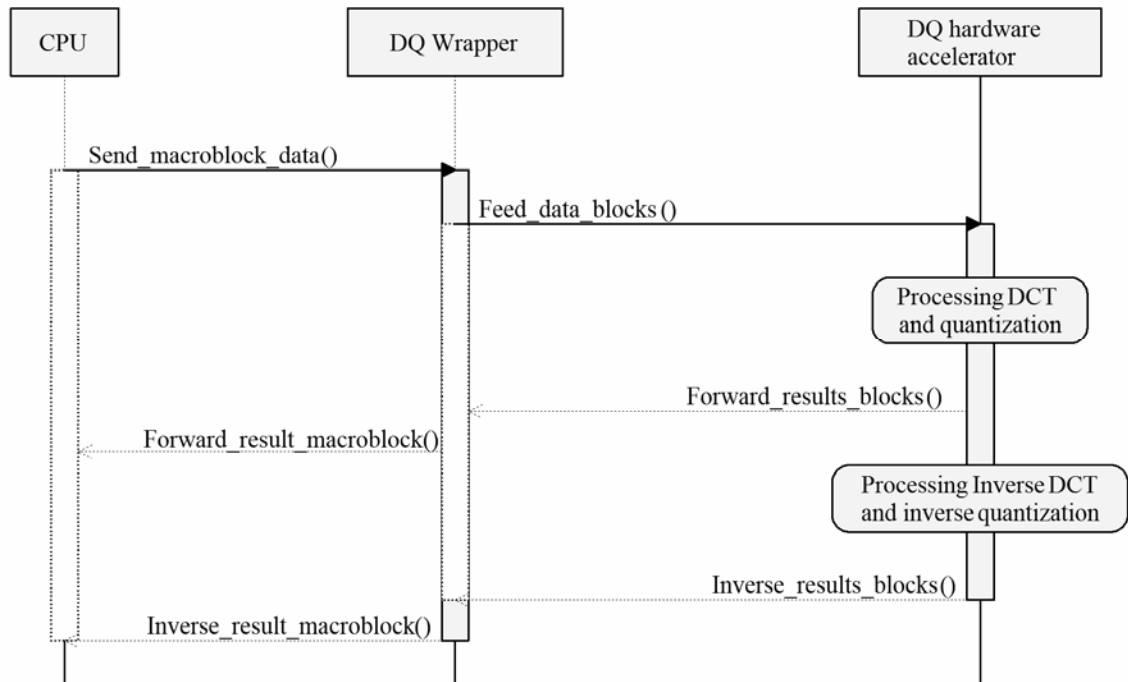
Since operations discrete cosine transfer, quantization, inverse quantization, and inverse discrete cosine transfer are all performed one after the other, it is efficient to partition them into a single hardware accelerator. Hence, the component is referred to as DQ. The quantization and inverse quantization implement the method described in H.263 standard [28]. The accelerator is responsible for tasks that are highlighted in Figure 16.



**Figure 16. The functions that DQ hardware accelerator implements.**

The hardware accelerator takes in one 8-by-8 block of pixels, transforms it to the frequency domain, and quantizes it. This result is both delivered out and forwarded to the inverse quantization and inverse DCT units that try to restore the original block. Therefore, it takes in one block and results in two blocks. The operations are performed differently depending on the *quantization parameters*, which are fed to the accelerator after every small block of input data.

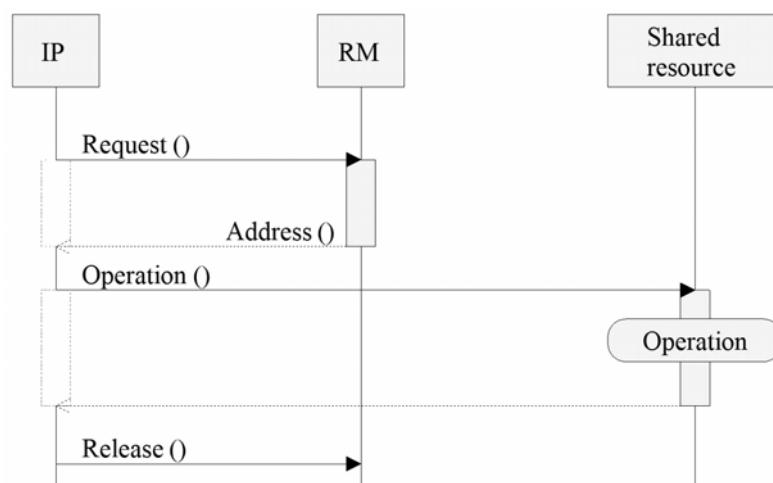
The wrapper makes DQ interface easier and more efficient for a CPU, since the CPU can send a macroblock at a time. Thus, the wrapper stores the macroblock (six 8x8 blocks) in a buffer and then feeds it to the accelerator one block at a time. For the results, it gathers blocks into a macroblock and sends it back to the CPU. In addition, the quantization parameter has to be sent only once to the wrapper. It will be sent first, before the macroblock data. The wrapper automatically forwards the parameter to the accelerator for every small block. The interaction between the wrapper and the hardware accelerator are shown in Figure 17.



**Figure 17. Sequence diagram of how CPU operates DQ hardware accelerator and its wrapper.**

### Resource manager

Resource manager is a component that schedules shared resource accesses. It allows multiple IP components, for instance CPUs, to share the same resource, such as shared hardware accelerator. RM can also be called as a mutual exclusion object, *mutex*. RM works as a link between IPs and shared resources. Every time an IP needs shared services, it has to request permission from RM. RM then either rejects or accepts the request. When the IP no longer needs the shared resource, the IP releases it. This operation sequence is depicted in Figure 18.



**Figure 18. Sequence diagram of RTM usage.**

RM contains a queue, where the requests are stored in case all shared resources are in use. However, the request queuing is optional, and requesting IP defines whether it is

used or not. Whether a request will be queued or not, it is called blocking or non-blocking. RM uses priority arbitration of units in first come first served fashion. In addition, it is easily expandable to support advanced algorithms.

An IP that wants to access a shared resource first sends a request to the RM, which returns the address of the reserved resource. The request contains three pieces of information: the type of shared resource, type of request, and a return address, where the response will be sent by the RM. There are two types of requests: blocking and non-blocking ones. If there are free resources, both blocking and non-blocking requests work the same way. The behavior differs when all the resources of requested type are reserved; blocking requests will be stored to queue and processed when the requested resource is released, whereas a zero word is returned to the sender of the non-blocking request. When the IP no longer uses the shared resource, it releases the resource in order to let other IPs use it. The release is performed by sending a release command to the RM

RM increases configurability, as it alone has to be aware of how many shared resources per type there is present in the system, and their addresses. For example, if the amount of hardware accelerators is increased, only the RM needs to be updated by changing a number in the configuration file. CPUs need only to hold the address of the RM in the memory. RM also enhances scalability, since it is easy to choose how many of the existing shared resources are used. The RM is therefore useful in a multiprocessor architecture with multiple shared hardware accelerators.

The shared resources are easier to implement if they are able to handle only one request at a time. Since the RM handles simultaneous access requests, IPs do not have to worry about new requests while processing the previous one or to implement a queue to store request. Thus, IPs can be simpler and therefore they utilize less logic and have more performance. In this way, the RM also reduces amount of logic on chip and eases the design.

Without RM, the IPs or CPUs would need to exchange messages in order to decide which can access to shared resource. This would complicate and slow down the accesses. Furthermore, CPUs could not process data when handling the exclusion messages.

## **Hardware monitor**

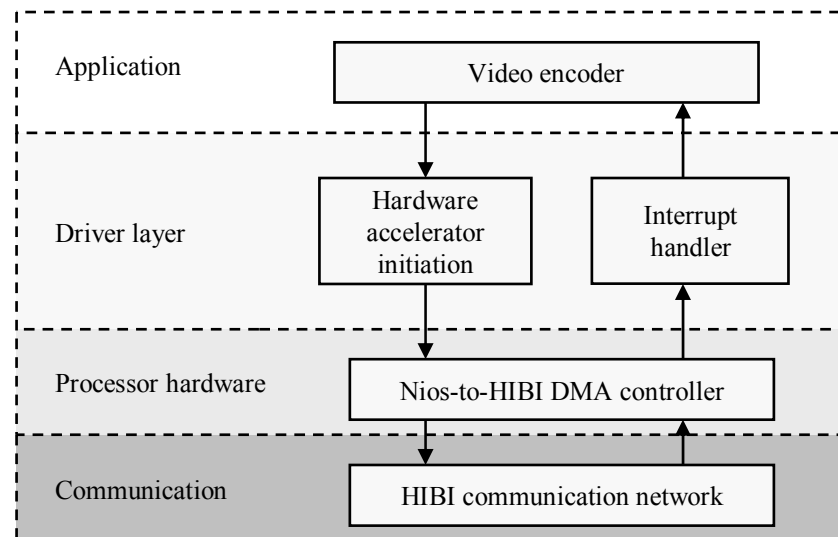
*Hardware monitor* is a new IP block that is meant for debugging and benchmarking of HIBI-based SoCs. In this work, it is used for acquiring exact execution times for hardware-accelerated tasks.

It monitors predefined signals and counts for how many clock cycles each signal is active. The results are got by sending a report command to the monitor. The monitor answers by returning the values of the counters. There are also commands for clearing the counters and for starting and stopping the counting. All commands are received and

the results sent through HIBI. The signals that will be monitored must be connected to the hardware monitor before synthesis.

### Driver software

The idea behind the driver software is that one can separate application from SoC architecture. The application is working on higher abstraction level and only has to know the name of the driver function, whereas the driver knows how to communicate with the hardware.



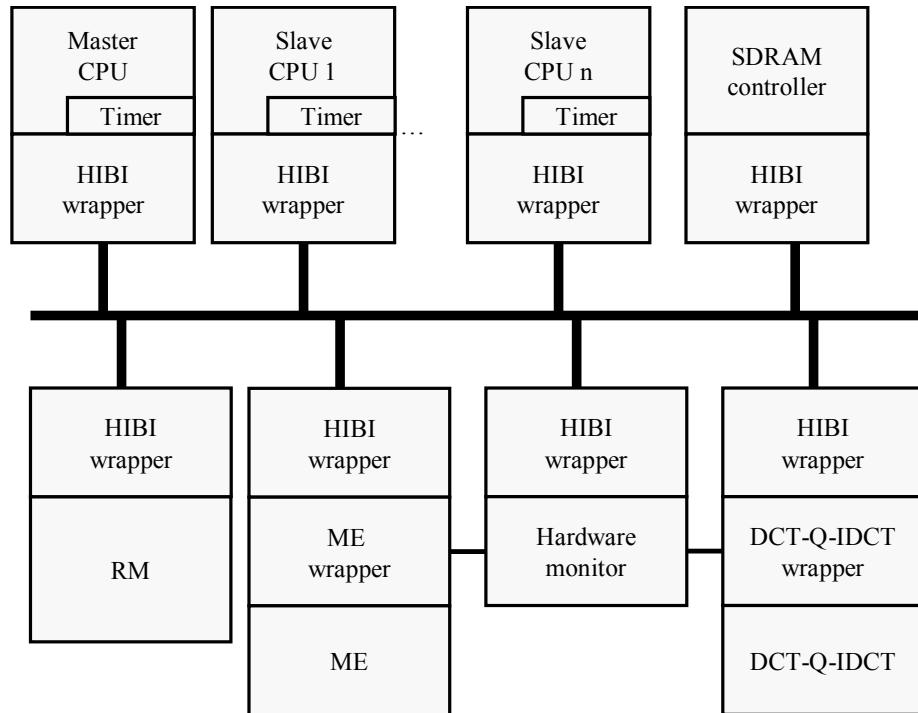
**Figure 19. Different layers of software and hardware at Nios processor.**

Figure 19 depicts the layers from software application to hardware communication network. Video encoder application is on top. It communicates with the software on driver layer, which further communicates with Nios-to-HIBI DMA controller. In addition, the Nios-to-HIBI DMA controller accesses the HIBI on-chip communication network. It is implemented in C.

In order to perform hardware accelerated operation, the video encoder software calls hardware accelerator initiation function with a pointer and reception data amount as parameters. The function first requests the HIBI address of a hardware accelerator's wrapper from the RM. It then commands the DMA controller to send request or transmit input data to the wrapper, depending on whether ME or DQ is initiated. Thereafter, the initiation function returns. When the results arrive from HIBI communication network, Nios-to-HIBI DMA controller forwards them to a address in local memory defined by the pointer parameter. When Nios-to-HIBI has received data amount indicated by the second parameter, it throws an interrupt to the interrupt handler, which activates a variable that indicates the arrival of results. The application software checks the value of the variable before it uses the results. If the results are not ready, the application will have to wait for the variable to activate.

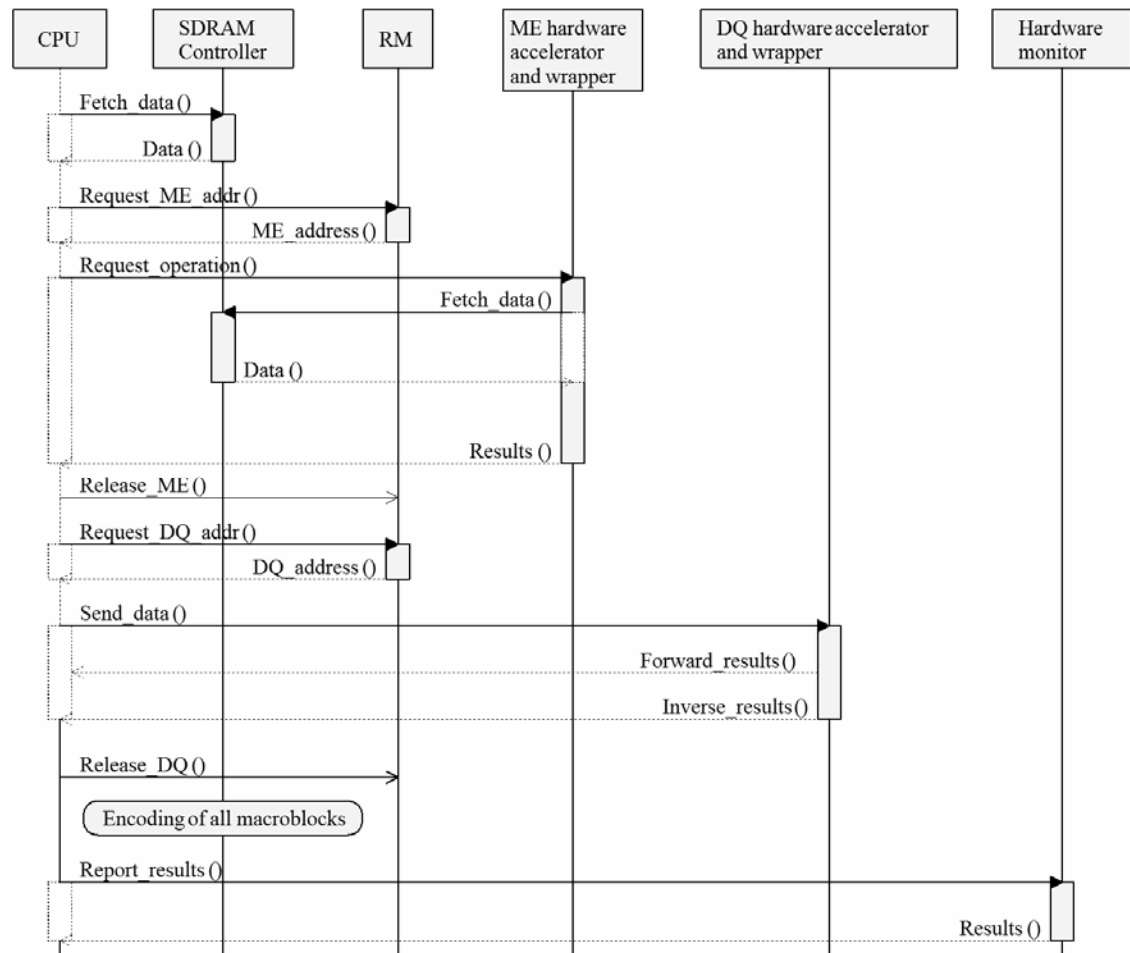
## 5.2. Accelerated system

The integration of hardware accelerators and the RM results in the integrated architecture presented in Figure 20. In addition to the already introduced IP blocks, the CPU timers are drawn in the figure. Moreover, the hardware monitor is connected to both hardware accelerators. These connections represent the signals that the monitor observes. The CPU timers and hardware monitor are used to measure execution time of the hardware accelerators and wrappers in order to show the benefit of hardware acceleration, and analyze integration overhead.



**Figure 20. Block diagram of integrated video encoder**

Figure 21 shows the order of transactions in the system. A slave CPU is on the left. It first fetches current macroblock and search area from SDRAM. Right after the memory fetch, the CPU requests address of motion estimation hardware accelerator from the RM, which returns the address to the CPU. CPUs query the address every time when accelerated task is performed. The slave CPU initiates motion estimation by sending the request to the ME wrapper. It handles the request, fetches data from the SDRAM at the addresses indicated by the request, and finally sends results back to the CPU. The CPU then sends releases message to ME and processes the results. It calculates residual and decides whether to use results from ME or not, based on the SAD value. Otherwise, it will utilize the macroblock fetched at the beginning.



**Figure 21. Sequence diagram of hardware accelerated encoding of a macroblock.**

In the case of successfully estimated motion, CPU sends the residual to the DQ unit. The operation is started again with querying access from the RM. After the RM grants request, the data is sent to the DQ through its wrapper, which forward it to the accelerator and send results back to CPU. The CPU releases the accelerator. The CPU runs quantized results through zigzag scan and entropy coding, whereas reverse operated (inverse quantization, IDCT) data is summed with the best matching macroblock to form reconstructed reference frame for the encoding of next frame. After successfully encoding a frame of video, the CPU requests result report from the hardware monitor.

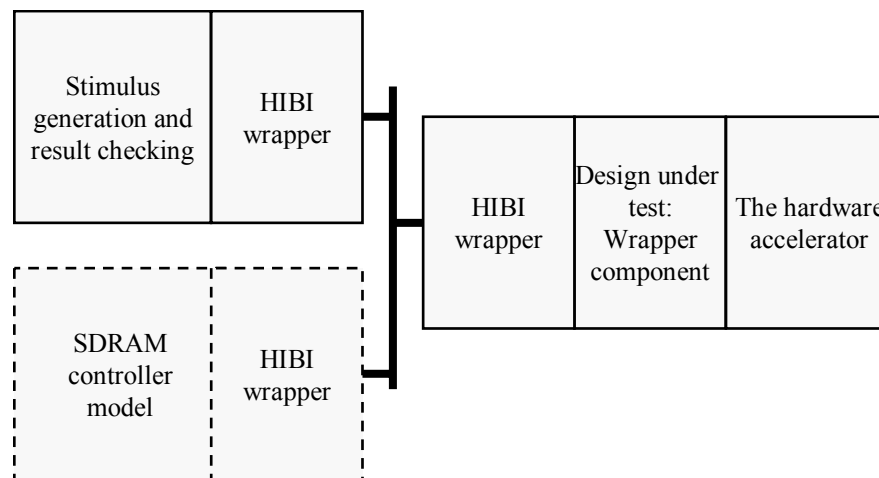
### 5.3. Integration of HW accelerators

The integration was implemented by creating wrapper components manually to adopt the interface and protocol of the hardware accelerators to the communication architecture.

The integration flow starts at the analyses of the interface of the acquired IP blocks. The low-level interface includes signals, signal widths, signal levels, and signaling protocols. On higher level, one must consider the functionality and visibility to the

processor. In order to make accelerated function execution as efficient as possible, the goal was to minimize the interaction with processor. Thus, the processor is able to perform other tasks uninterrupted in parallel. Moreover, the video encoder dataflow was important, because the accelerators require data from the specific phase of encoding. For instance, ME has to be performed before DQ and software determines whether to use the results of ME or not. The analyses were carried out before describing any logic and resulted in block level specifications of the wrappers. These are presented in sections 6.3 and 6.4

Thereafter, the specified wrapper components were described using VHDL. In order to verify the written descriptions, testbenches had to be created as well. Both of the testbenches worked alike and were created using VHDL. They consisted of the wrapper component, the provided simulation model of the hardware accelerator, a HIBI wrapper, and test stimulus generation unit that also checked the results. Since the wrappers and hardware accelerators require totally different input values and produce different results, own stimulus generation and result checking had to be created for each of the configurations. Since ME required SDRAM access, also a simulation model of SDRAM controller with memory had to be created. The testbench structure is depicted in Figure 22. The SDRAM controller model is drawn with dashed line, because it was only used for ME wrapper testing.



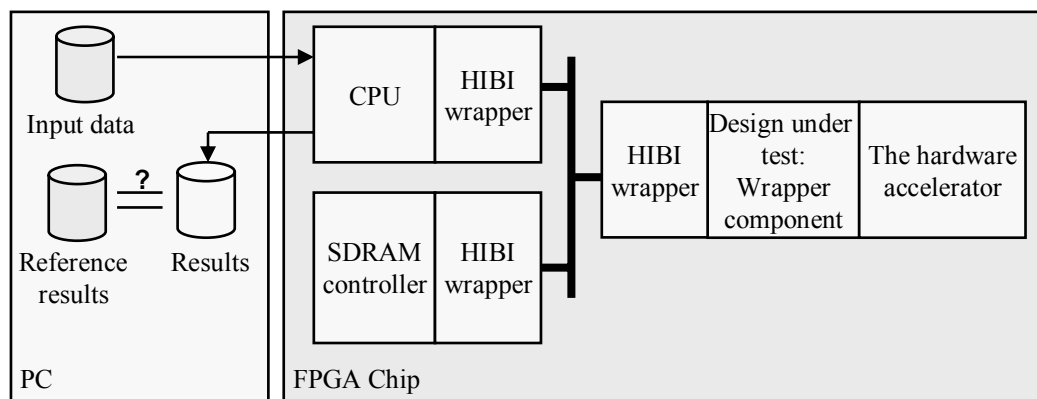
**Figure 22. The testbench structure for wrapper components**

After the successful functional verification in the testbench, the verified wrapper and the IP block were next attached to the target architecture. This was a straightforward operation, where output signals of the wrapper were assigned to the incoming signals of communication network and vice versa. In addition, a new clock signal had to be created to provide hardware accelerators with own parametrizable operating frequency.

In order to ensure a successful attachment and the operation of the hardware before taking application software into use, the system was verified on system level on FPGA. This was implemented using one Nios II processor and the SDRAM controller. The verification configuration is illustrated in Figure 23. In the case of ME hardware

accelerator, stimulus was generated and results checked against reference results on PC that was connected to the FPGA via Ethernet. In the case of DQ, the PC was not used. Instead, the stimulus and result checking was performed on Nios II. For both cases, small test programs were designed. The source of error is easier to extract in simple test program than with the whole application. These test programs were also the starting point for the drivers that were implemented for the video encoder software.

For running the test software, the design had to be synthesized and fitted to the FPGA chip. The wrapper component was described using synchronous design style, so static timing analysis was used to verify that the required timing was reached. Quartus II tool performed the synthesis, fitting, and static timing analysis.



**Figure 23. The testbench structure of simple FPGA test.**

After the hardware was verified, the application software was updated to support the accelerators. This was done by replacing the software implementation of accelerated functions with driver function calls that initiate the acceleration. The modifications were tested on the FPGA in system level verification with one encoding slave CPU. Here, the initial agreement for the application correctness was that the accelerated video encoder could run continuously without errors for nine hours, which was accomplished.

The reason why only one processor was used at this point is that hardware accelerators could be used by only one processor at time. Introducing several processors without resource management would have caused simultaneous accelerator accesses. Thus, the accelerator would have mixed up the input data and request words causing the corrupted outcome, or the results would have been sent to wrong addresses, hence halting the system. The resource manager was implemented to avoid this kind of situations.

The VHDL simulation and simple FPGA testbenches verified different features and on varying levels. Different levels and test cases are listed in Table 2. Letter *X* in a column denotes that the testbench was exclusively designed to test the feature. When the testbench was not designed for a feature, but still implicitly verifies it, *I* is marked to the corresponding column.

The VHDL simulation mainly verified signal, message, and system level test cases, whereas simple FPGA was used to verify test cases on software level. The final application was used mainly for software, but at the same time, it verified nearly all features implicitly.

**Table 2. Test case coverage in different testbenches. Letter X denotes that the testbench was explicitly designed to verify a feature, whereas letter I means that the testbench was not designed for the feature but still implicitly verifies it.**

Level	Test case or feature	VHDL Simulation	Simple FPGA test	Final application
Signaling	Different word width	X		
	HIBI FIFO buffer reading and writing	I	I	I
	Input data writing to HW accelerator	X	I	I
	Result reading from HW accelerator	X	I	I
Message	Request of operation	X	I	I
	SDRAM controller protocol (ME)	X	X	I
	SDRAM controller single data mode (ME)	X		
	Random length transfers on HIBI	X	I	I
	Intra/inter mode operation (DQ)	X		I
System	Endianness of data (ME)	X		
	Different address spaces		I	I
	Self releasing	X		I
Software	Driver communicates with accelerator		X	I
	Executing an operation		X	I
	Encoding of a macroblock			X
	Encoding of a frame			X

Different word widths were verified only in VHDL simulation, since they were too complicated to implement on chip. HIBI FIFO buffer reading and writing is performed in every testbench, but neither of them exclusively verified the operation.

In message level, the simplest test case is to send a request of operation and some dummy data to the wrapper and see, if it returns anything. SDRAM controller protocol was tested on this level with a model of SDRAM controller and in simple FPGA test

with real SDRAM controller. In SDRAM single data mode, SDRAM controller model was modified to send one data word at a time irregularly. It is closely related to the random length transfers on HIBI, which was implemented in VHDL simulation to interrupt HIBI transfers randomly. This simulates the real situation in encoder, where the length of HIBI transmissions varies. Intra/inter mode operation in DQ is a test case of using different parameters for quantization unit.

In system level, it was impossible to modify the CPU in FPGA implementations to support different endianness. Thus, VHDL simulation tested how ME handled different endianness of input data. In addition, different address spaces were not explicitly tested, but simple FPGA test and the final application used different addressing. Furthermore, the simple FPGA test did not include RM and could not verify the self-releasing feature, which means that the accelerator itself sends release command to the RM.

In software level, the simplest test is to check whether a driver on processor can communicate with an accelerator. The next step is to execute an operation. These tests were exclusively performed on simple FPGA test. Two encoding test cases were tested only in final application.

#### **5.4. Lessons learned**

There were two versions of ME wrapper before the final one. The first was not entirely described using synchronous design conventions. Instead, there were asynchronous signal assignments that could have caused indeterministic behavior and long critical paths. The whole wrapper code was rewritten. The second version had unnecessary complexity, and thus it was simplified. In addition, support for bidirectional data transfer was added. This way the wrapper could forward input data and results simultaneously resulting improved performance. A new feature to change image width was added for the final version.

The DQ wrapper implementation was much easier because the development of ME wrapper design taught a lot. In addition, the DQ wrapper is much simpler. In fact, there is only one version of DQ. However, numerous bugs in the design were observed in simulation testbench and fixed. These included syntactical errors, carelessness in resetting signals in startup, incorrect state transitions, and badly timed signal activations.

The introduced wrapper testbench was successfully used to discover several bugs in designs. However, the described testbench was not perfect, since it did not test wrapper's interface signals explicitly. In addition to this high-level testbench, a more specific testbench for testing wrapper's interface signals would have been useful. The testbench only used a FIFO implementation against the wrapper's interface, instead of setting for instance empty and full signals high and low randomly. This interface test would have helped to find a bug, which was perceived on FPGA when a FIFO buffer

---

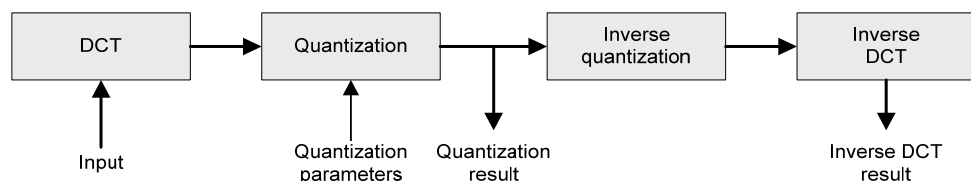
implementation in HIBI wrapper was changed. The bug prevented the system from running. The reason for malfunction was that the wrapper logic assumed that the data bus holds its value after *agent\_empty\_in* signal activates. As a conclusion, testing in simulation environment proved to be much easier especially in locating the design errors. However, FPGA testing was also important due to its speed.

## 6. IMPLEMENTATION OF WRAPPERS AND RESOURCE MANAGER

This chapter describes the interfaces of hardware accelerators as well as the implementation of wrappers and the resource manager in detail. All IP blocks presented in this chapter are implemented with synthesizable RTL VHDL. The DQ hardware accelerator was delivered in VHDL description files and with interface specification document. Hence, it is soft IP block, although the description files are not commented. However, the ME hardware accelerator was provided as gate level netlist. Thus, it is considered firm IP. The content of the component is unknown, only the interface is documented. Both hardware accelerators are designed independent of the video encoder architecture presented in this work. Thus, different interfaces and functionality between hardware accelerators and the rest of the system results in complex wrappers.

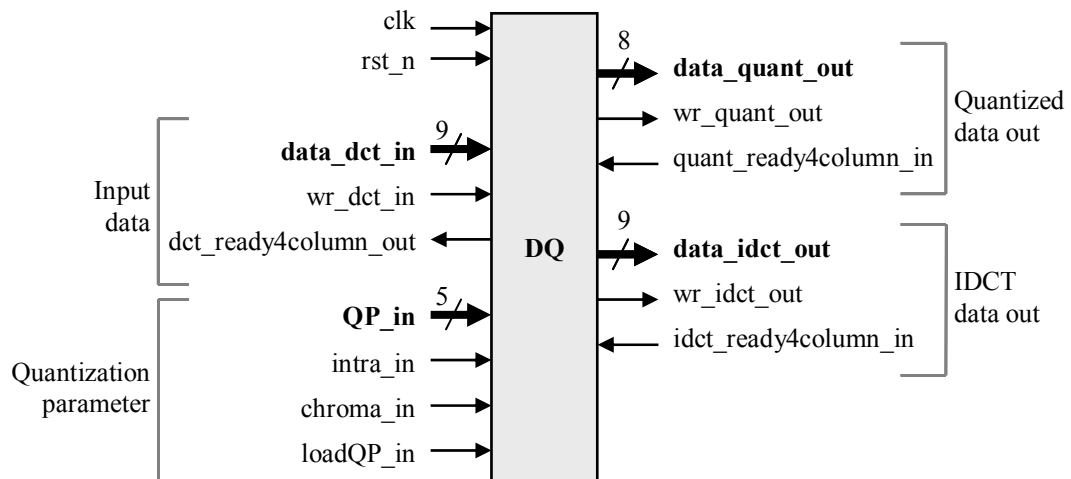
### 6.1. DQ hardware accelerator

The DQ hardware accelerator consists of four units in a row. Units are DCT, quantization, inverse quantization, and inverse DCT. They are depicted in Figure 24. The component takes in 8-by-8 blocks of 9-bit data and quantization parameters. It produces two results. For the first result, only two first operations are performed. Data are transformed into frequency domain and quantized. The second result goes through all four units. Therefore, the units of the inverse quantization and inverse DCT process the first results further. The first results are in frequency domain, but the second ones are transformed back to the pixel presentation. The operations are pipelined in such way that new input block can be loaded while the accelerator operates the previous block.



**Figure 24. DQ hardware accelerator data flow.**

The interface of the hardware accelerator is depicted in Figure 25. It contains four groups of signals, one clock, and one reset signal. The first group is *input data*, which is for inputting data. The second, *quantization parameter*, is for sending in the quantization parameters related to the input data. The third group is for outputting the data gone through DCT and quantization, *quantized data out*. Finally, the fourth group is for sending out the data that has gone through all four operations. It is marked on the figure with *IDCT data out*.



**Figure 25. The interface of the DQ hardware accelerator.**

In the group *input data*, the signal *dct\_ready4column\_out* is active when the block is ready to receive eight values, which equal one row or column of an 8-by-8 block. The values are set on the 9-bit bus *data\_dct\_in* and signal *wr\_dct\_in* determines the validity of the data.

According to the specification of the block, the three parameters for quantization should be loaded within 30 clock cycles after the 8-by-8 block of input data has been loaded. Right values are read in from signals *QP\_in*, *intra\_in* and *chroma\_in*, when the signal *loadQP\_in* is active. Signals *intra\_in* and *chroma\_in* define whether the input block is of type intra or inter and chrominance or luminance, respectively.

The results are being uploaded similarly to each other. When the target is ready to read eight result values, it informs DQ by activating *quant\_ready4column\_in* or *idct\_ready4column\_in* signals depending on the requested type of data. The block then sets value to the corresponded output bus and activates the *wr\_quant\_out* or *wr\_idct\_out* depending on the requested type of data.

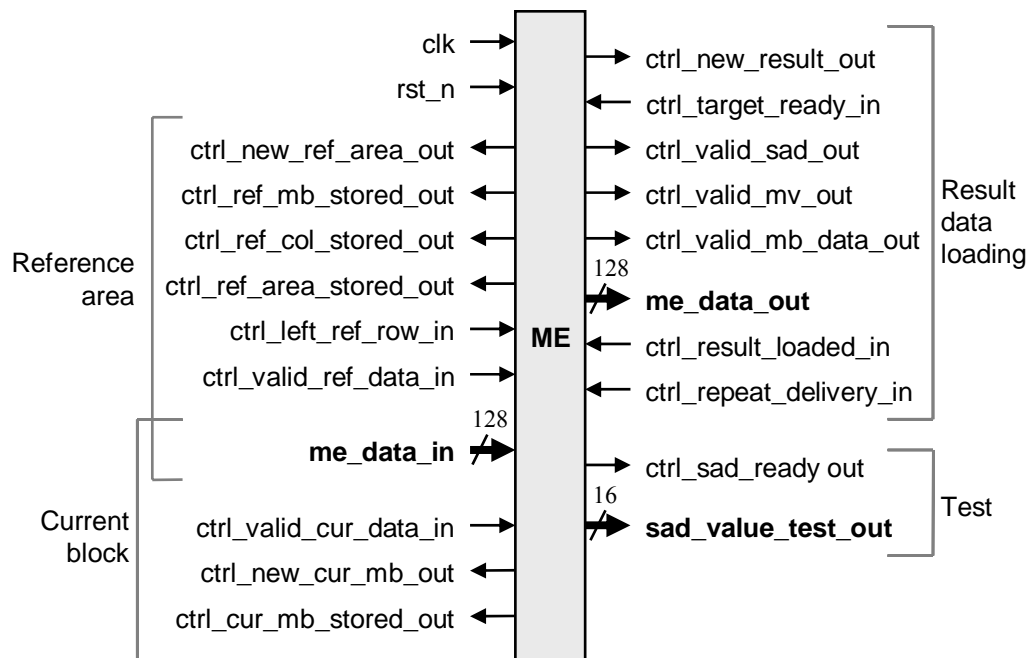
## 6.2. Motion estimation hardware accelerator

The hardware accelerator for motion estimation performs the operations in full-pixel resolution. The motion estimation hardware accelerator takes in altogether ten macroblocks of two adjacent frames. From the prior frame, the accelerator needs the whole search area, which is three-by-three macroblock matrix. From the latter frame,

that is the current frame, the accelerator needs only one macroblock, the current macroblock. Reference area and current macroblock total ten macroblocks. Each macroblock is a 16-by-16 square of 8-bit pixels, since only luminance values are used in estimation. This means 20480 bits of input data.

There is an option in the ME hardware accelerator that can be used to save duplicate bits in input data. As the macroblocks of the frame are operated in row-wise order, two neighboring macroblocks have overlapped search area. When the option is turned on, only the rightmost column of macroblocks needs to be uploaded from the new reference frame, whereas the other two columns are two rightmost columns of the previous reference frame. This way the amount of input data could be reduced down to four macroblocks, which equals 8192 bits of input data with 60 percent decrease. Unfortunately, due to architecture's parallel processing, back-to-back encoding of macroblocks is not guaranteed. Thus, it is impossible to use the option because different CPUs may request the ME consecutively.

As a result, the motion estimation gives out the best matching macroblock, a two-dimensional motion vector indicating the relative position of the best matching macroblock to the current macroblock, and a SAD value that indicates the similarity of the match and the current macroblock. The smaller the SAD is, the less difference there is. The best matching macroblock is 16-by-16 square of 8-bit pixels, the SAD value is 16 bits wide, and two components of a motion vector have width of 8 bits, there is constant amount of 2080 bits of results coming out. The data buses in and out of the motion estimation accelerator are both 128 bits wide (i.e. 16 pixels). Thus, it can read the input values in 160 clock cycles and send out results in 18 clock cycles.



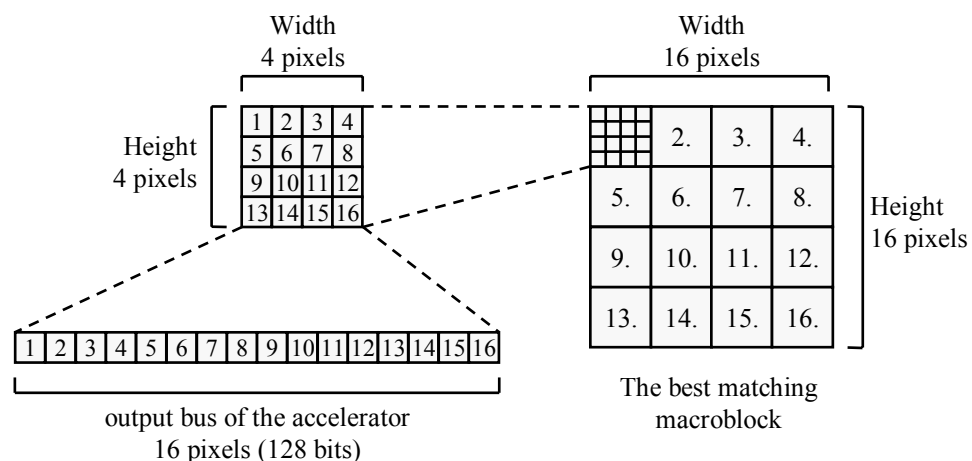
**Figure 26. The interface of the motion estimation hardware accelerator.**

The interface of motion estimation hardware accelerator interface consists of several signals. The signals can be divided into four categories as shown in Figure 26. One is for result data loading with 128-bit output bus *me\_data\_out*. Two of them are for inputting the reference area and current macroblock data, which both utilize the same 128-bit bus *me\_data\_in*. In addition, the component contains clock and reset signals: *clk* and *rst\_n*.

The hardware accelerator indicates a need of input data by activating either *ctrl\_new\_ref\_area\_out* or *ctrl\_new\_cur\_mb\_out* depending on whether it needs reference area or current macroblock. When the data arrives, it is put to the *me\_data\_in* bus and sent by activating *ctrl\_valid\_ref\_data\_in* or *ctrl\_valid\_cur\_data\_in*. Signals ending with *\_stored\_out* indicate successful receiving of input data.

The accelerator informs the finish of the calculation by activating the *ctrl\_new\_result\_out*. The results will not come out before the accelerator is told that target is ready and activates *ctrl\_target\_ready\_in*. The data comes in *me\_data\_out* bus. Whether the data is SAD value, motion vectors, or the best matching macroblock a corresponding *ctrl\_valid\*\_out* signal is activated. If the target fails to receive the data, it can request repeated delivery by activating the *ctrl\_repeat\_delivery\_in* signal. After the target has received all the data, *ctrl\_result\_loaded\_in* has to be activated in order to allow the accelerator to calculate new values and replace the old ones in a buffer.

The best matching macroblock is sent out in small squares of 4 by 4 pixels. These sixteen 8-bit pixels equal 128 bits in a small square, and thus they are delivered on *me\_data\_out* in one clock cycle. The best matching macroblock can be constructed from these 4-by-4 squares in the order Figure 27 presents.



**Figure 27. The relation between 4-by-4 blocks in the best matching macroblock and in output bus of the motion estimation hardware accelerator.**

### 6.3. Wrapper component for DQ

The DQ wrapper is designed in such a way that any device capable of sending data over HIBI can also control the accelerator. It is not by any means limited to be used with Nios CPU, or with a CPU at all.

The wrapper is operated by sending three words of configuration data followed by the input data itself. The first configuration word is the HIBI address, where the quantization results are sent. The second word is the HIBI address of the target that will receive IDCT results. The last configuration word is a control word that includes the quantization parameter. The results of the quantization and IDCT are sent back to the configured addresses.

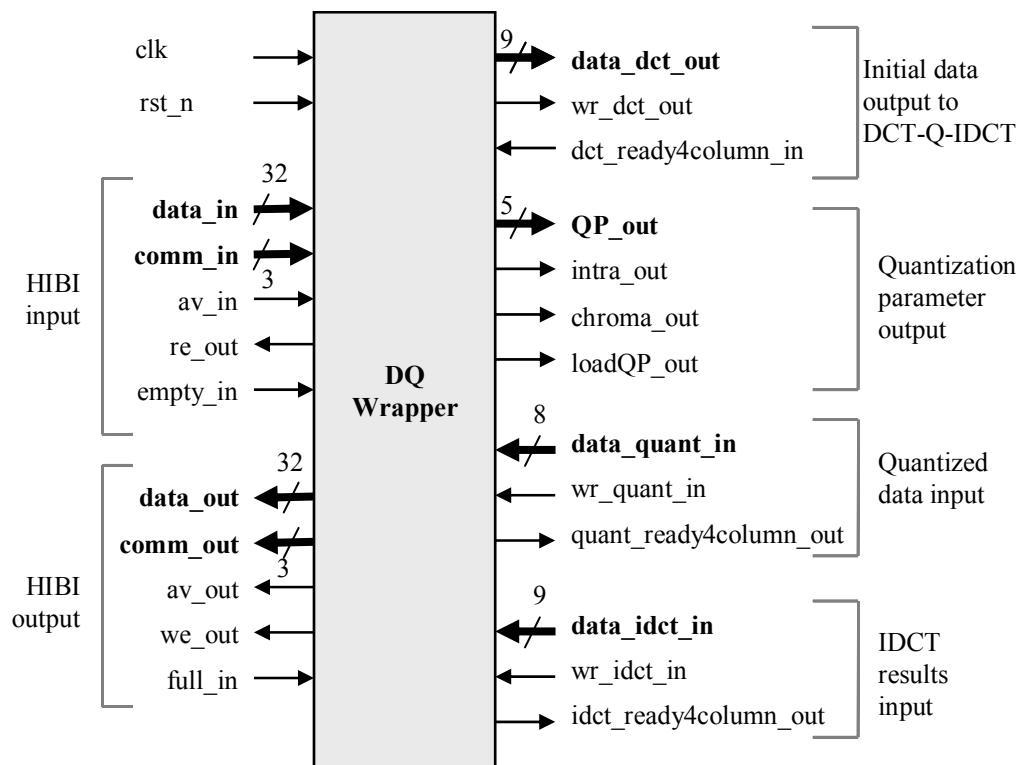
When considering the software implementation of the video encoder. The DQ hardware accelerator needs either the best match of motion estimation or the current macroblock depending on the encoding mode. In this case, the CPU decides the mode and already possesses the data. Therefore, instead of reading the data from the shared memory, the CPU sends the data directly to the wrapper component.

For efficiency, the CPU sends a macroblock at a time to the wrapper. The accelerator takes in 8-by-8 pixel blocks. One macroblock with both luminance and chrominance information can be extracted into six 8-by-8 blocks: four luminance blocks and two chrominance blocks. Thus, the pipeline in the hardware accelerator can be exploited because there are multiple 8-by-8 blocks available in the wrapper. Therefore, the wrapper is designed to receive six 8-by-8 blocks, feed them to the accelerator, and thus take advantage of the pipeline.

In addition to the adapting, the wrapper implements also a zero checking functionality, that checks the quantized result data for blocks that contains only zero coefficients and sends the result of zero-checking to the CPU. This simplifies the implementation in software. Moreover, DQ has a feature to send release command to the RM after receiving all the input data. Otherwise, the CPU has to send the release after receiving the results. This way the hardware accelerator can start the next operation earlier.

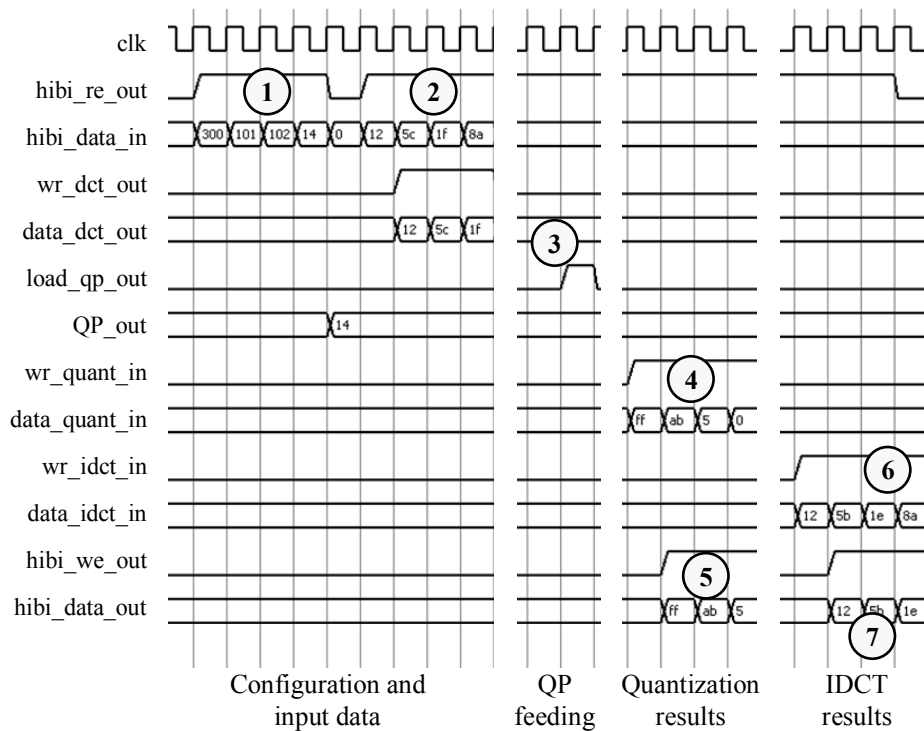
## Interface signals and generic parameters

The signal level interface of the wrapper component is shown in Figure 28. The left side of the component is attached to a HIBI wrapper and the right side to the hardware accelerator. The interface signals of HIBI have been described in Section 3.1 and the signals of the hardware accelerator in Section 6.1. For simplicity, the interface signals are grouped into signal bundles that are used in the following block diagrams.



**Figure 28. The interface of the DQ wrapper component.**

Figure 29 represents the timing behavior of wrapper and the accelerator. The three words of configuration data (101, 102, and 14) are received from HIBI (1). The preceding 300 is the address of the wrapper on HIBI. 101 and 102 are the return HIBI addresses in this example and 14 is the quantization parameter. Thereafter, the actual data arrive (2). Only the four first words are drawn for simplicity. They are forwarded to the DQ unit that it is accessed through *data\_dct\_out* bus. When the feeding of 8-by-8 input data block is finished, the wrapper provides the accelerator with quantization parameter (3). When the accelerator finishes processing of the first results, *wr\_quant\_in* informs the wrapper of valid quantized result data on *data\_quant\_in* bus (4). It is forwarded to HIBI output (5). Finally, the data from the IDCT unit come out (6) and it is again forwarded to HIBI output (7).



**Figure 29. Timing behavior of DQ wrapper.**

The wrapper has several synthesis-time parameters, called *generics* in VHDL. These affect the functionality of the wrapper block as well as the interface. All the generics, their default values and descriptions are listed in the Table 3. Width parameters define the data and communication bus widths of the HIBI wrapper and the accelerator. With generic *use\_self\_rel\_g* the self-releasing feature can be turned on or off. The two last generics configure the HIBI addresses used in the self-release feature.

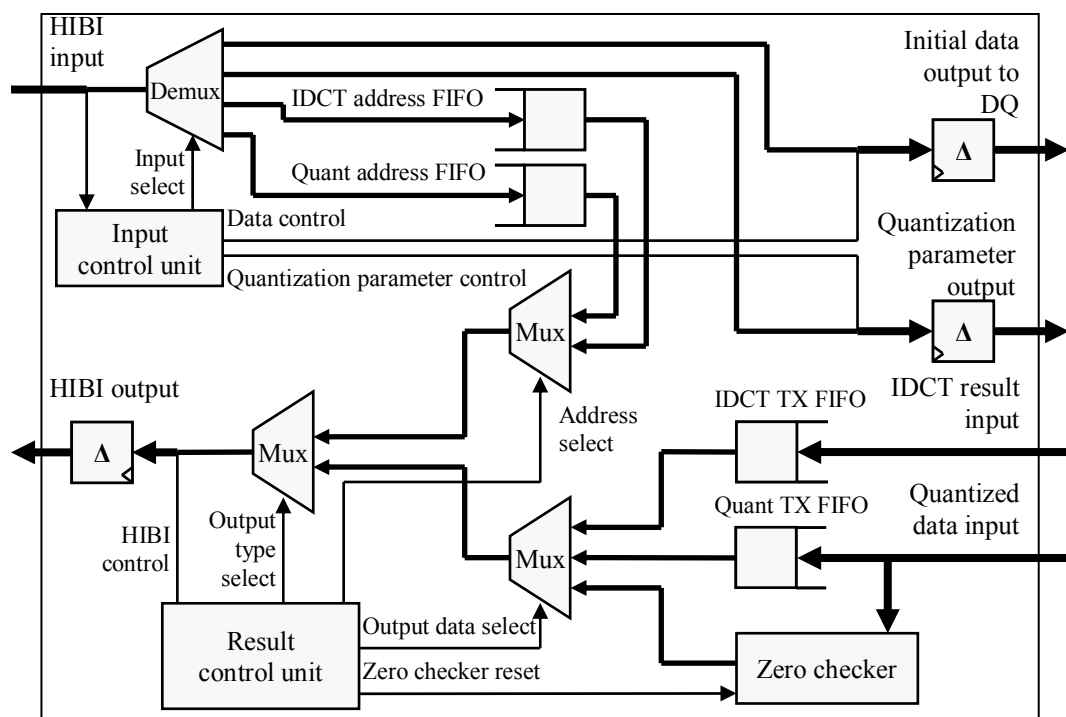
**Table 3. Synthesis-time parameters of DQ wrapper block.**

Generic name	Default value	Description
data_width_g	32	Width of the HIBI data bus
comm_width_g	3	Width of the HIBI communication bus
dct_width_g	9	Data width of DQ input
quant_width_g	8	Quantized result data width
idct_width_g	9	Result data width of IDCT
use_self_rel_g	0	Should the wrapper release itself
own_address_g	N/A	The own HIBI address for self release
rm_address_g	N/A	The HIBI address of the RM for self release

## Structure and operation

The wrapper is logically divided into two parts, which are controlled by separate control units. The first is responsible for relaying the data from HIBI to corresponding FIFO buffer or register. The second part controls the HIBI output and sends out either return address, result data of either kind, or the value of a zero checker block.

The basic structure in the form of block diagram is shown in Figure 30. The left side of the block is attached to a HIBI wrapper with two interface bundles *HIBI input* and *HIBI output*. The right side of the block is attached to the DQ hardware accelerator with signal bundles *initial data output to DQ*, *quantization parameter output*, *IDCT result input*, and *quantized result input*.

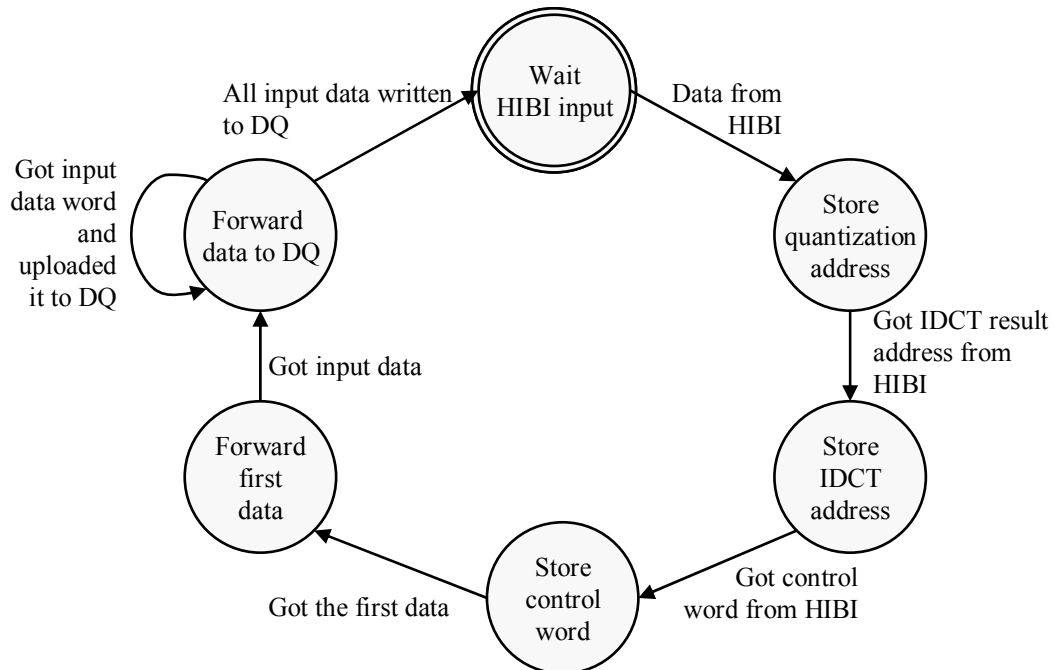


**Figure 30. Block diagram of the DQ wrapper component.**

All the data sent by a CPU comes from HIBI input. The configuration data, the two return addresses and the control word, are directed into corresponding FIFO buffers and the *quantization parameter output* register shown in the figure. The actual payload data that follows the configuration goes to the register remarked as *initial data output to DQ* and input control unit validates it. After every 8-by-8 pixel block, the input control unit triggers the sending of the quantization parameters, which determine quantization amount and type of the block. For this purpose, input control unit also contains counters that monitor the amount of written data and the corresponding type of data, as the pixel block can represent either luminance or chrominance.

Besides the counters and some random logic, the input control unit contains finite state machine (FSM) that controls the demux at the input. The state diagram is shown in Figure 31, where circles are states and arrows transitions to the next state. The text next

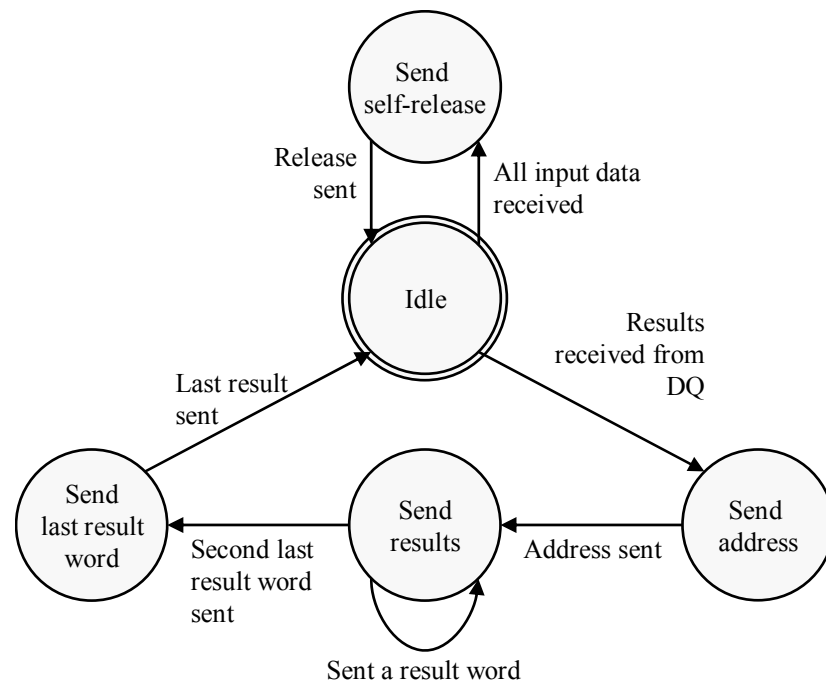
to the arrow defines the condition when the transition is triggered. For simplicity, the operations and signal assignments performed in state transitions are not mentioned. The circle that has double outline is the starting state. The controller waits for input from HIBI. It forwards the first data to quantization result address FIFO buffer. After this, the IDCT result address is stored to corresponding FIFO buffer, and control word to quantization parameter buffer. The actual data is the written to DQ in the state *forward to DQ*.



**Figure 31. State diagram of input control unit in DQ wrapper.**

The result control unit is responsible for transmitting result data from either of the transmit (TX) FIFO buffers or from the zero checker block. The zero checker block observes the quantized results and checks if 8-by-8 block contains only zero values. This is actually an operation that the processor would have to do to the received data. In wrapper, it can be performed parallel with data sending, but CPU would have to do it sequentially after receiving the data. Thus, hundreds of clock cycles are saved from the execution time.

Before sending out the result data, wrapper also sends the corresponding address from result address FIFO buffer using *address select* signal to choose address and *output type select* signal to choose between address and data. The result control unit contains also a FSM. It is depicted in Figure 32. Here, from the state *idle* the control can branch to either releasing or result sending, if the self-release option is in use. The result sending is straightforward. A counter counts the number of sent words. When all the result words are sent, it sends the value from the zero checker block in the state *send last result word*.



**Figure 32. State diagram of result control unit in DQ wrapper.**

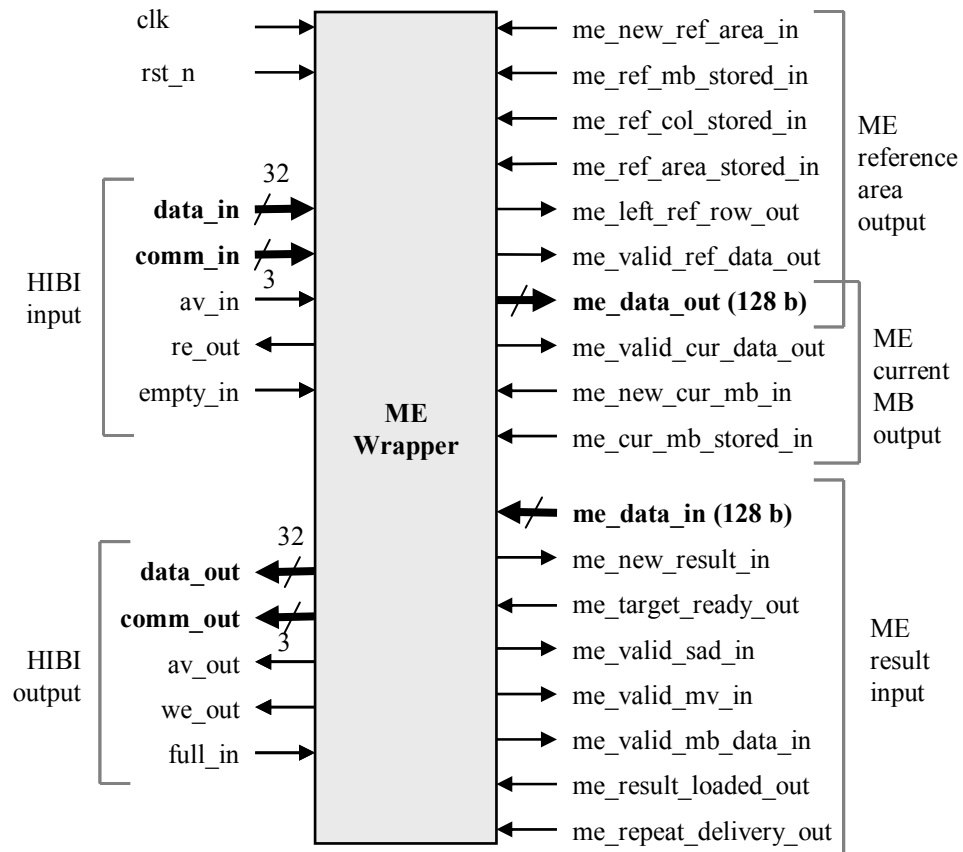
Self-release happens after the all input data has been uploaded to the DQ. This way, the RM can assign accelerator to the next CPU, which starts sending new input data while the results of the previous calculation are still being processed. Hence, the pipelined execution of hardware accelerator can be better exploited. Without the self-release, the CPU would release its accelerator after receiving the last word of data. Self-release can potentially save hundreds of clock cycles of execution time.

#### 6.4. Wrapper component for Motion estimation

The ME wrapper is more complex than the DQ wrapper, since it implements more functionality. Thus, it is presented hierarchically divided into top-level block and three sub-blocks. As the DQ wrapper, the ME wrapper can be also controlled by any HIBI compliant device. However, in contrast to DQ, the data is not sent to the wrapper directly. Instead, only a three-word long request is sent to the wrapper, and wrapper fetches data itself from the SDRAM using the on-chip SDRAM controller. So, the ME wrapper implements also a simple DMA controller. However, the results are directly sent to a HIBI device whose return address is specified in the request, a slave CPU in the video encoder. The request contains the SDRAM memory start address of the current macroblock, the memory start address of the reference area, and the result address where the results are sent. The ME wrapper implements the same self-release feature than the DQ wrapper allowing the operation to be performed more frequently.

## Interface signals and generic parameters

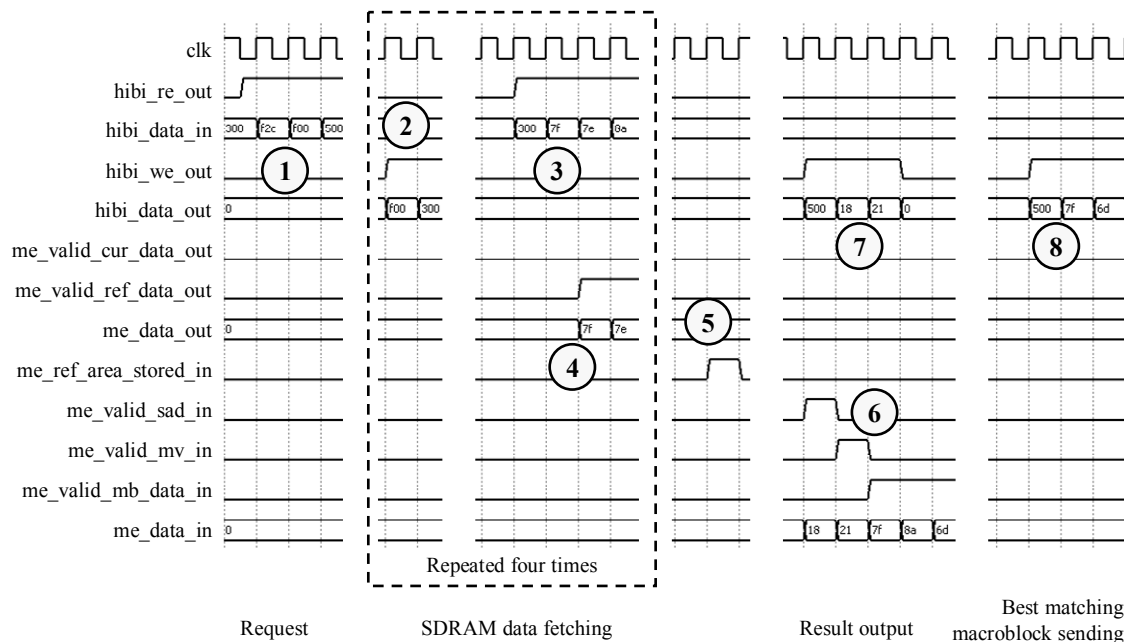
The signal level interface of the wrapper component is shown in Figure 33. The left side of the component is attached to a HIBI wrapper and the right side to the hardware accelerator. The interface signals of HIBI have been described in Chapter 3.1 and the hardware accelerator side in Chapter 6.2. Again, the interface signals are grouped into signal bundles that are used in following block diagrams.



**Figure 33. The interface signals of ME wrapper component.**

Timing behavior of ME wrapper component is depicted in Figure 34. First, the wrapper receives the request from HIBI (1). The *hibi\_re\_out* indicates that the component is reading from HIBI wrapper's FIFO. Word 300 is ME wrapper's HIBI address in this example. The next three words are all HIBI addresses. The first two ones point to picture memory and the last to the CPU that sent the request. Next, the wrapper configures the SDRAM controller for reading data (2) by using the SDRAM addresses at the request. This is actually takes tens of clock cycles, but is simplified. In this case, *f00* is the HIBI address of the SDRAM. Signal *hibi\_we\_out* indicates that the ME wrapper is writing to HIBI wrapper's FIFO - that is sending data. After successfully configuring the SDRAM controller, read data from SDRAM arrives through HIBI (3). The data are then forwarded to the ME accelerator (4) by activating corresponding signal that indicates whether the data are from current macroblock or reference area. In

the example, the data are type of reference area, since signal named *me\_valid\_ref\_data\_out* is activated. The figure is simplified, since the actual width of *me\_data\_out* is 128 bits but only 8-bit bus is drawn. The dash-lined rectangle is performed four times. The search area is fetched in three vertical slices from the SDRAM. In addition, the current macroblock is fetch from the SDRAM. After the whole search area is fetched and fed to the ME hardware accelerator, it confirms successful reading with signal *me\_ref\_area\_stored\_in* (5). For simplicity, other these kinds of confirmation signals are left out from the figure. The accelerator informs ready results with three different valid signals. There is one for SAD, motion vectors, and best matching macroblock (6). Corresponding data are on *me\_data\_in* bus while the signals are active. The motion vectors and SAD values are immediately sent to target over HIBI (7). In this example, SAD value is 18, and motion vector components are two and one. However, the best matching macroblock is sent after permutation operation, which takes tens of clock cycles (8).



**Figure 34. Timing behavior of ME wrapper.**

The wrapper has several synthesis-time parameters that affect the functionality of the wrapper block as well to the interface signal widths. All the generics, their default values and descriptions are listed in the Table 4. The generic *ip\_addr\_width\_g* is utilized in extracting the base HIBI address of an IP block from any HIBI address with offset. Generic *rm\_address\_g* needs to be set only if generic *use\_self\_rel\_g* has value one. Moreover, generic *big\_endian\_g* is for processors that store data to memory in different byte order.

**Table 4. Synthesis-time parameters of ME wrapper block.**

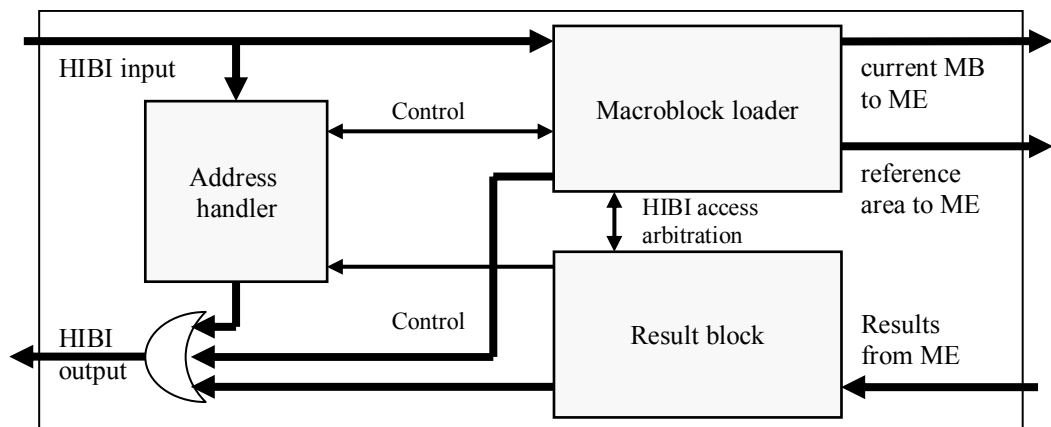
Generic name	Default value	Description
data_width_g	32	Width of the HIBI data bus
comm_width_g	3	Width of the HIBI command bus
ip_addr_width_g	N/A	Number of bits belonging to IP block in HIBI address
use_self_rel_g	0	Should the wrapper release itself
own_address_g	N/A	Own HIBI address for self release and SDRAM operations
rm_address_g	N/A	The HIBI address of the RM for self release
big_endian_g	0	Determines if the byte order in a word is big (=1) or little (=0) endian. Nios II is little endian.

The top-level block diagram of the ME wrapper is shown in Figure 35. There are three sub-blocks in the wrapper, address handler, macroblock loader, and result block. All the components require access to HIBI output, but only address handler and macroblock loader are connected to HIBI input. Depending on the incoming HIBI address offset, either address handler or macroblock loader receives the data.

The *address handler* is relatively simple and only serves the two other blocks by storing the addresses from the request. In addition, it sends addresses out to the HIBI according to the control signals from the other blocks.

The *macroblock loader* implements the DMA controller, as it controls the data fetching operation from SDRAM. It is also responsible for feeding the received data to the hardware accelerator.

*Result block* is responsible for permutation and forwarding the ME results to HIBI. In addition, it is responsible for arbitrating the *HIBI output*, since both *macroblock loader* and *result block* can start HIBI transmissions.

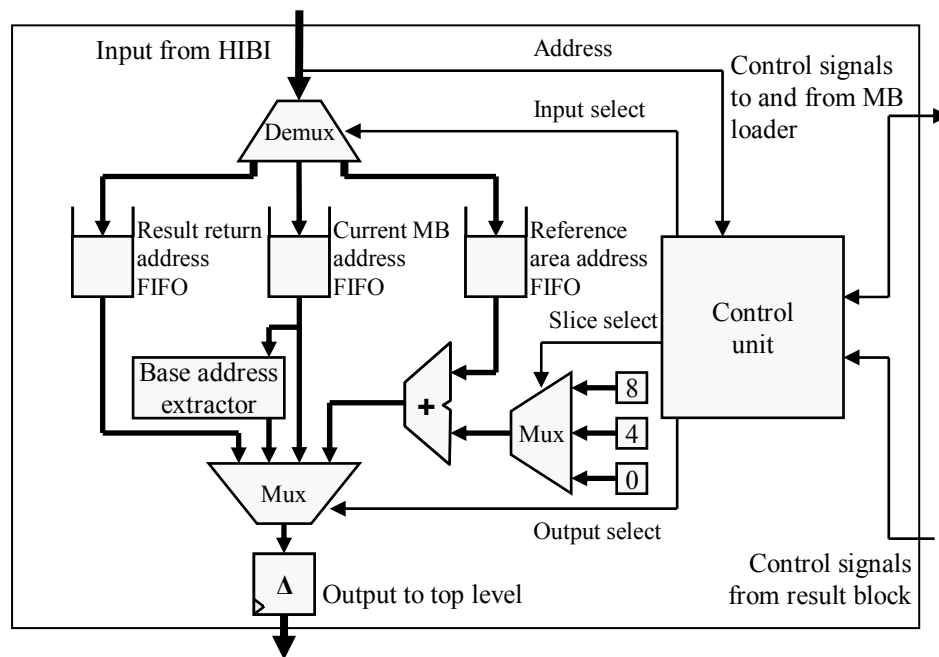
**Figure 35. The top-level block diagram of the ME wrapper.**

Next, implementations of all three sub blocks in ME wrapper are discussed separately and in detail.

### Address handler sub-block

*Address handler* stores three addresses for other blocks and sends them to its HIBI output when requested. One is address for returning results and the two are SDRAM memory addresses, where the current macroblock and search area are. In addition, *address handler* contains logic to alter memory address to point to the HIBI base address by masking off the least significant bits.

The hardware accelerator requests the search area in three vertical slices that are one macroblock wide (16 pixels) and three macroblocks high (48 pixels). Instead of storing one address per slice, the address handler is responsible for storing one for the leftmost slice and calculating the other two addresses from it. SDRAM memory is with word addresses. Each 32-bit word stores four pixel values (8 bits each). Hence, one 16-pixel row in macroblock takes four words. Thus, to calculate the two addresses *address handler* adds number zero, four, or eight to point to any of the three slices of the search area.



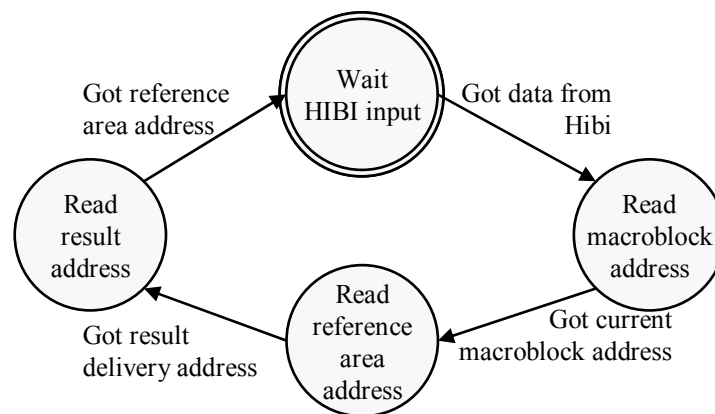
**Figure 36. The block diagram of the ME wrapper sub-block Address handler.**

The block diagram of address handler is shown in Figure 36. It consists of a *control unit*, data paths through one of the three FIFO buffers, and an adder unit to modify the value of a FIFO buffer with one of the static numeric register values. When the *input from HIBI* is address and matches the offset defined in *control unit*, the *control unit* starts forwarding data from HIBI to corresponding FIFO buffer. The address, where the results are sent, is stored in the *result return address FIFO* buffer. The SDRAM

memory addresses of current macroblock and reference area are stored to FIFO buffers *current MB address* and *reference area address*.

The result return address is directly sent to the output when requested by other blocks via control signals. Current macroblock address has two options and reference area address can be added with any of the three numeric values zero, four, or eight. There is also option to set zero value to the output, but for simplicity, it is not shown in the figure. The summed term with reference area address is determined by signal *slice select* and it depends on the control signals from the block *macroblock loader*.

The base address extracting is implemented with small unit *base address extractor* that modifies the current macroblock address. The extractor masks out the least significant bits of the address thus creating the HIBI base address of the target IP. This is useful, since a read port is requested from the SDRAM controller using its base address, not the exact memory address. The address can be forwarded to the output either directly or through the unit.



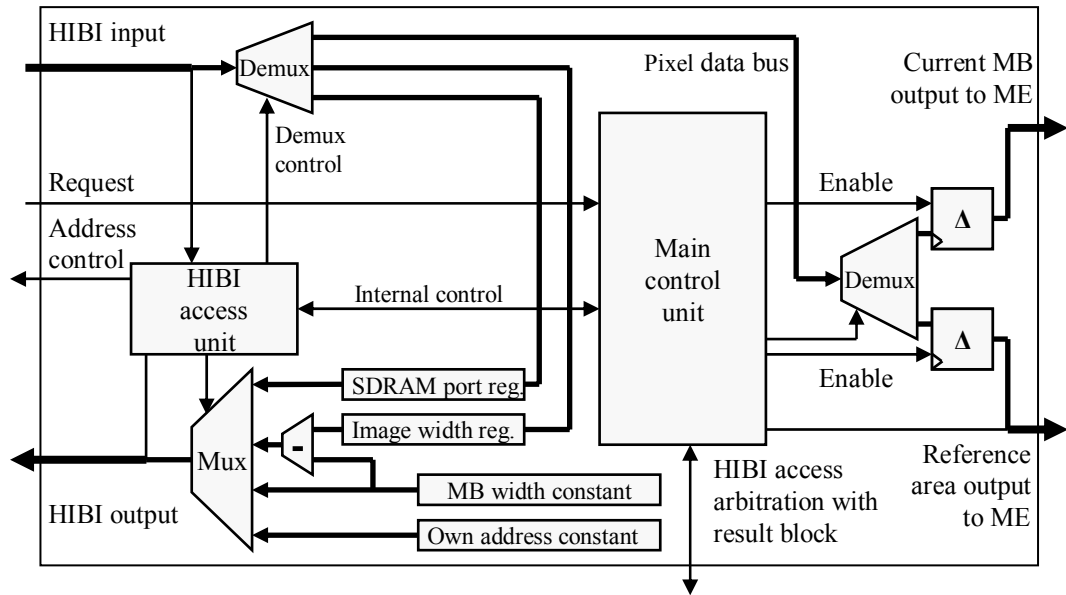
**Figure 37. The state diagram of ME wrapper's sub-block Address handler.**

A FSM responsible for forwarding data is depicted in Figure 37. It simply waits for three separate words of data and stores them to the corresponding FIFO buffer.

### Macroblock loader sub-block

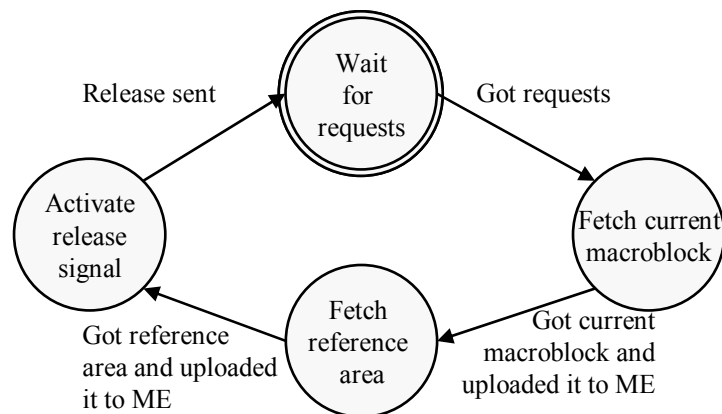
The *macroblock loader* component consists of two control units, a divided data path, and several data value registers. The input data from HIBI is directed to a data value registers or to either of the inputs of ME. The structure is shown in Figure 38.

*HIBI access unit* fetches a macroblock or reference area from SDRAM, when it detects simultaneous active request signals from address handler and ME hardware accelerator. When the data arrives from SDRAM, *HIBI access unit* switches the first demux to direct data to *pixel data bus* that *main control unit* further routes to either *current macroblock output* or *reference area output*. *Main control unit* lets the *HIBI access unit* to access *HIBI output* only if *HIBI access arbitration* signals indicate that the result block is not using the output.



**Figure 38. The block diagram of the ME wrapper's Macroblock loader sub-block.**

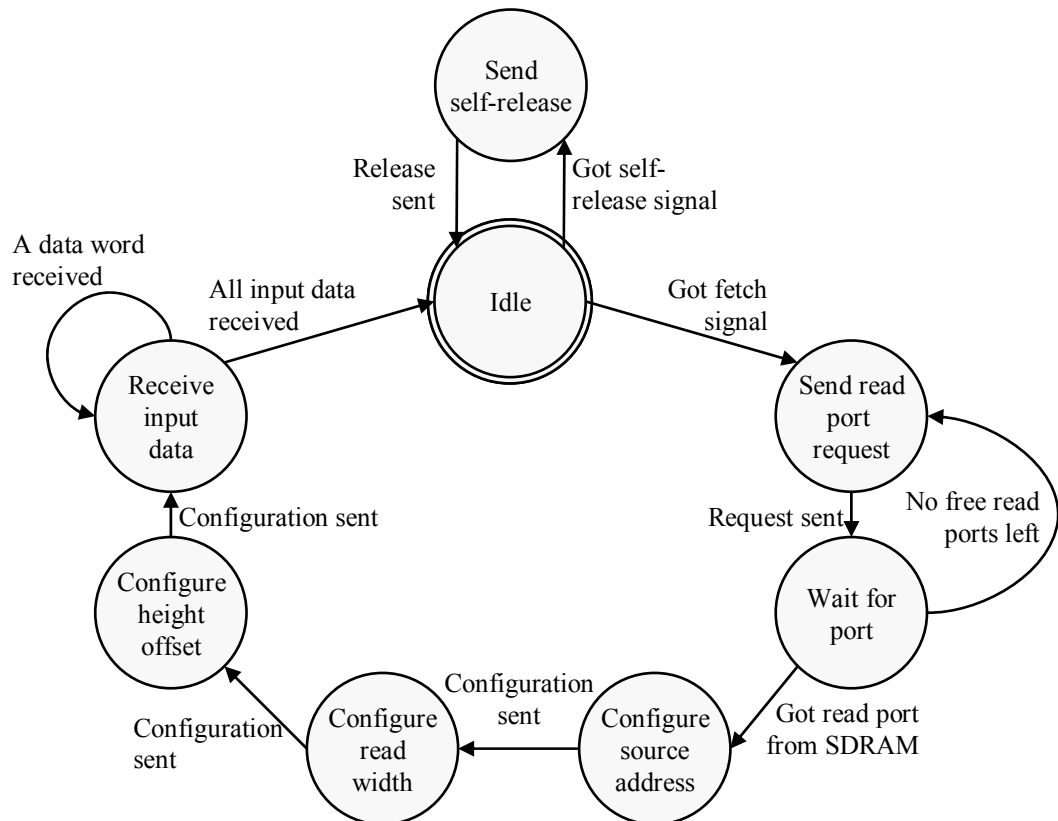
The FSM inside the main control unit is depicted in Figure 39. It first waits for the requests. In states *fetch current macroblock* and *fetch reference area*, the FSM activates fetch signal, which initiates *HIBI access unit*. Since the fetch operation takes several clock cycles, also the state machine spends the same amount of cycles in these two states. After receiving all the data, the *main control unit* activates release signal in state *activate release signal*, which causes the *HIBI access unit* to send release to the RM. In the previous figure, these signals are in the *internal control* bundle.



**Figure 39. State diagram of ME wrapper's sub-block Macroblock loader: main control unit.**

The *HIBI access unit's* state machine is depicted in Figure 40. This state machine implements the SDRAM protocol. First, FSM waits in idle state for the fetch signal that is asserted by *main control unit*. The unit sends read port request to the SDRAM. The corresponding address comes from FIFO buffers of *address handler* sub-block. If there are no free read ports left, the request is repeated. Otherwise, the read port address is saved to the *SDRAM port register* and used for configuring the SDRAM control. The

source address points to the top-left corner of a macroblock or reference area. Again, it is stored in the *address handler*. After sending the source address configuration, the next parameter is the width of the read operation. This is constant value and set in the *macroblock width register* in the block diagram. The last configuration parameter is the height offset. This is obtained by subtracting the width of the read operation from the width of the image. By default, the width of the image is set to QCIF width, but it can be configured at run-time. The HBI access unit FSM waits for all input data to be received and then idles. The release operation works in the same manner than in the DQ wrapper in Chapter 6.3.



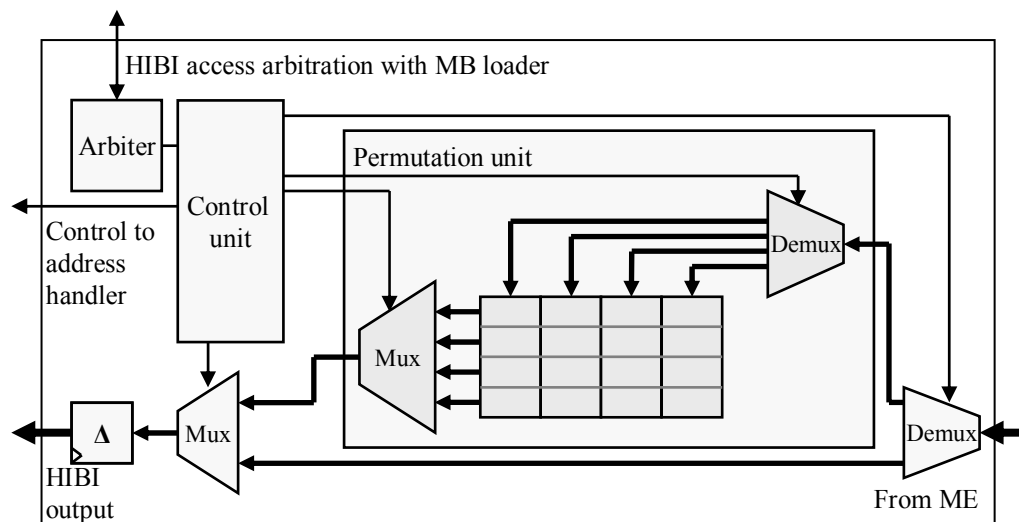
**Figure 40. State diagram of ME wrapper's sub-block Macroblock loader: HIBI access unit.**

### Result block

The *result block* consists of *arbiter unit*, *control unit*, *permutation unit*, and data path from motion estimation to HIBI output directly or through the *permutation unit*. In addition, there are interface signals to *macroblock loader* and *address handler*. The block diagram is shown in Figure 41.

The *control unit* is responsible for directing the data to HIBI output via demuxes and muxes. When the data is ready to be sent, the *control unit* uses *address handler* in the same way than *macroblock loader*: the *address handler* first sends out the target address that is followed by the actual data from *result block*. The motion vectors and minimum SAD values are directly forwarded to the register in *HIBI output*. However, the best

matching macroblock data has to be permuted to deliver it to a CPU in the rows of pixels, whereas it comes out from the ME in 4-by-4 pixel blocks as depicted earlier in Figure 27.



**Figure 41. The block diagram of ME wrapper's sub-block Result block.**

Each of the 16 boxes in the permutation unit represents a 32-bit register that equals four 8-bit pixel values. When considering a column of registers, a 4-by-4 pixel block of data can be saved there. Thus, the whole register matrix can contain four 4-by-4 pixel blocks of data. This is minimum amount of registers that is required to rearrange the data. Input data is stored column-wise in the registers. Each 128-bit result word fills up one of the columns. The first quarter of a 128-bit result word is stored in the top-left register, the second quarter to the register below that, etc. When all the registers are filled up, the mux is controlled to direct one register content to the output at a time. The registers are read in the row-wise order. For simplicity, the mux in the permutation unit of Figure 41 is drawn with four inputs. Actually, every 32-bit register of the matrix are directed to the output at a time. This totals 16 inputs. However, the size of the mux could be optimized by merging four adjacent registers as a shift register.

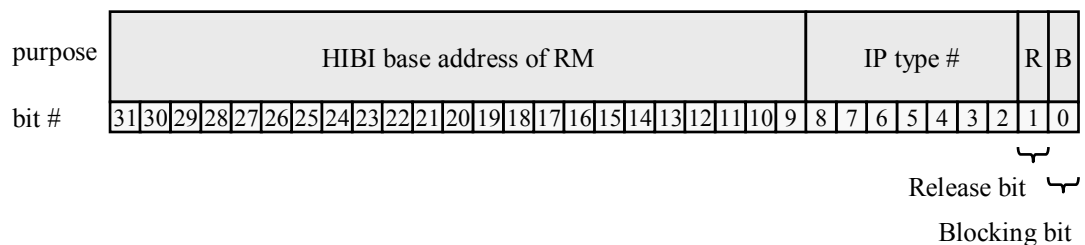
The *arbiter* is responsible for managing the shared *HIBI output*. It assures, that only one block can access the output at a time. Since the *address handler* is controlled by both the *macroblock loader* and *result block*, the output has to be arbitrated only between these two blocks. The block that first initiates a transmission is allowed to use it until freed in the end of the transfer. However, the *result block* has higher priority. If the *HIBI output* is free and both of the sub-blocks are trying to access *HIBI output* at the same clock cycle, the *result block* wins and gains the access.

## 6.5. Resource manager

The resource manager is used as a general purpose mutual exclusion and scheduling unit of shared resources in HIBI based systems, not just in the video encoder discussed here.

### Request message

The RM is used through HIBI with request messages consisting of two words: a HIBI address and data. The structure of the HIBI address is always the same. First, in the most significant end of the HIBI address word there is HIBI address of the RM. It is followed by the 7-bit type field that determines the type of the shared resource, which is formed by consecutive numbering starting from zero. The two least significant bits indicates if it is about release or request, and whether the request is blocking or non-blocking. This is illustrated in Figure 42 for 32-bit address.

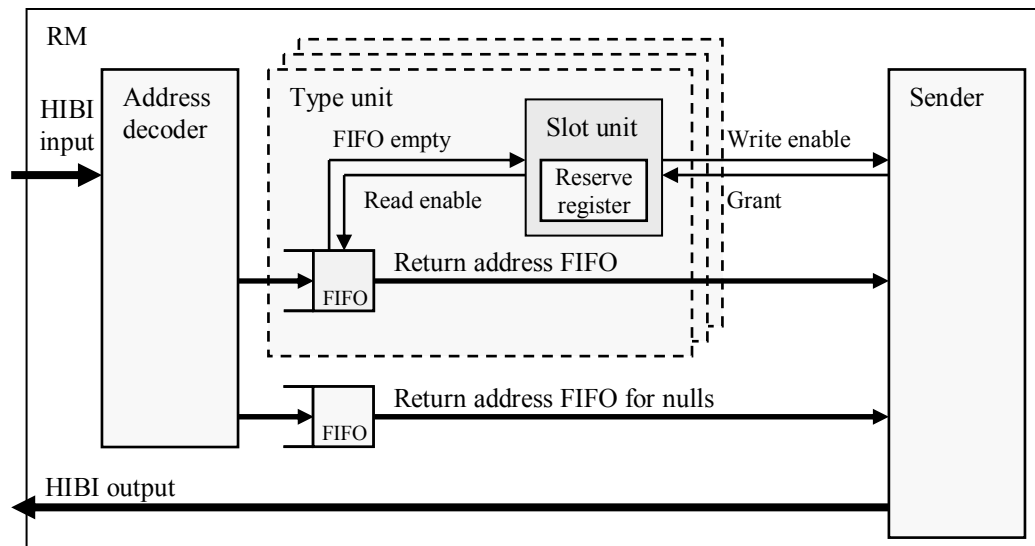


**Figure 42. The structure of address in 32-bit-wide HIBI address.**

Interpretation of the data following the address depends whether the message is a request or a release. The data represents a response return HIBI address or the HIBI address of the shared resource to be released, respectively.

### Structure

The structure of the RM is depicted in Figure 43. The RM consists of an *address decoder*, *type units*, a *sender*, and a FIFO buffer for addresses. A *type unit* is created for every supported type of a shared resource. Every *type unit* contains a FIFO buffer and *slot unit*. The FIFO buffer stores blocking requests, whereas the *slot unit* contains a reserve register for each accelerator of a type and maps requests to resource addresses. For example, two *type units* are needed when three DQ and one ME accelerators are used. However, there are as many reserve registers as instances in *slot units*: three and one. The addresses of hardware accelerators are configured at design time.



**Figure 43. A block diagram of the RM.**

In normal behavior, the address decoder distinguishes incoming HIBI addresses between operations (request or release) and directs them to corresponding *type unit*. If FIFO buffer in *type unit* is full or the non-blocking request cannot be handled instantly, the request is set to the *return address FIFO for nulls*. However, the release is always handled immediately. The *return address FIFO for nulls* is used for returning zero value fast to the initiator.

When a *type unit* has handled a request, it informs the *sender* by activating *write enable* signal. The *sender* goes through *write enable* signals of every *type unit* in turns. In the case of active signal, the *sender* activates corresponding *grant* signal and sends the answer to the address at *return address FIFO*. In the case where none of the *type units* have *write enable* active, *sender* also checks if there are pending zero word replies in *return address FIFO for nulls*.

## 7. PERFORMANCE ANALYSIS

The effect of integration on performance and silicon area indicates the usefulness of integration. This chapter introduces the measured quantities, the results of integration in form of video encoding speed, accelerated function execution time, and area usage on FPGA. Moreover, factors of integration overhead are extracted from the bases of function execution times.

### 7.1. Execution time and performance

Performance is inversely proportional to execution time. It is reasonable to evaluate performance on system level, but execution time on task level. As the execution time depends on the used clock frequency, measuring clock cycles gives more general and repeatable results.

In order to evaluate the benefit of hardware accelerators to the system, it is required to measure system performance before and after the integration. An encoding frame rate was obtained from the inverse of frame execution time average. The frame execution time was measured with the CPU timers. As the architecture is scalable, the encoding frame rate was measured using one, two, and three encoding slave CPUs using either none, one, or both of the hardware accelerators. In addition, since execution time and encoding speed depend on the video sequence, three different 90-frame-long QCIF-sized video sequences were used: *carphone*, *akiyo*, and *foreman*. The results are averages from the three sequences. Grayscaled screenshots of the sequences are presented in Figure 44.

The CPU timers were also used to measure clock cycles spent on DQ and ME functions from the processor's point of view. The CPU timers were set to measure the clock cycles between the driver function call that initiates the accelerated task and the interrupt request that is thrown at the processor when the results arrive back to the CPU. Hence, it includes all integration overheads. In addition, the execution time of the original software function was measured to provide a point of comparison. The CPU timers are accessed through driver function calls in the software program. Thus, they cause small delay to the execution. However, the delay was measured and compensated.



**Figure 44. The three video sequences used in video encoding benchmarks: carphone, akiyo, and foreman.**

The hardware monitor is implemented to obtain more detailed results on the hardware accelerators and the wrapper components. It is designed so that it can be used with CPU timers to measure the actual usage of the hardware accelerator in separation of the overheads, caused by the software, shared resource contention, and data delivery.

Hardware acceleration execution time was measured in three different configurations: simulation, simple FPGA test, and hardware accelerated encoder. CPU timers were used in simple FPGA test and hardware accelerated encoder, whereas HW monitor was used in all three configurations. HW monitor and CPU timers were present in the SoC architecture block diagram at Figure 20. These three configurations and the software encoder configuration are discussed next.

### **Simulation**

The first configuration is a hardware accelerator attached to a simulation testbench written in VHDL and C using a foreign language interface. It measures the pure accelerator execution time without any overheads. Wrapper components are not used at this point, since the testbench supports the native interface of the component. In the testbench, the data are always available to the IP block. The configuration is referred to as “simulation” in the results. Because of the simulated environment, the CPU timers cannot be used, but HW monitor is available.

### **Simple FPGA test system**

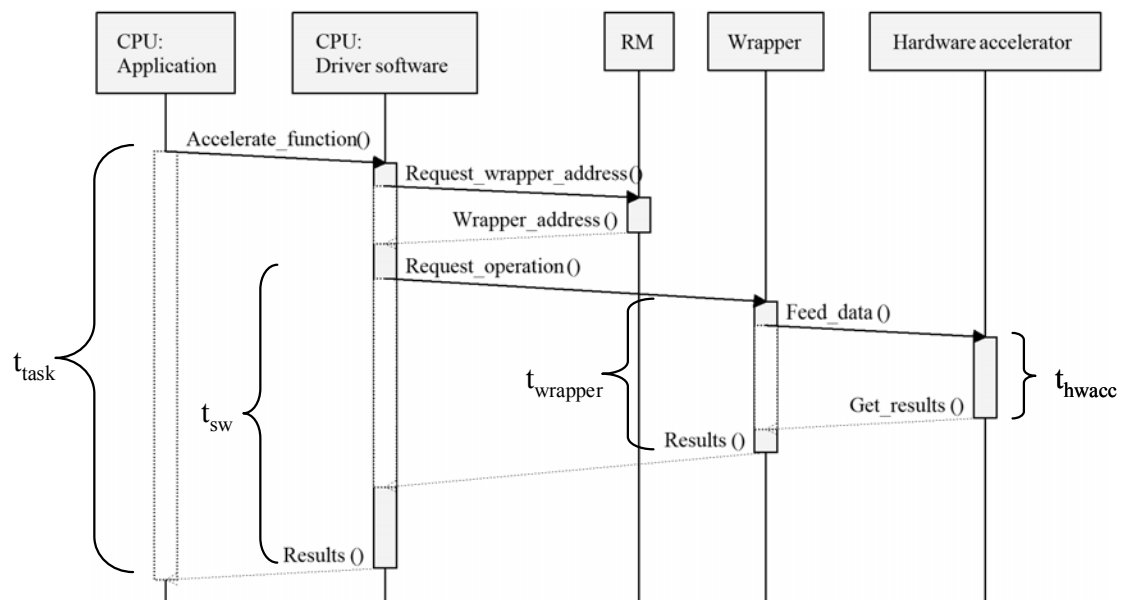
In the second configuration, the wrapper component is introduced and attached between the hardware accelerator and HIBI wrapper. The hardware also contains Nios II CPU and the SDRAM controller. In this limited system, the CPU runs a test program, which is dedicated to run the accelerators consecutively. The configuration is as minimal as possible to run the accelerators on FPGA with a processor. Since there are no other parallel operations running on the chip, all resources are available when required. Even the shared bus is always available, since neither the DQ nor ME sends and receives data simultaneously. This configuration is used to measure data delivery expenses on chip. In addition, this is used to measure the overhead of software and configuration of accelerators. This is referred to as “simple test” in the results. The block diagram of the simple FPGA test is shown in Figure 23.

## Hardware accelerated video encoder

The third configuration for measuring function execution time is the video encoder application; with two encoding slave CPUs, both of the hardware accelerators, the RM, and the SDRAM controller. The real application running on the architecture is the most important configuration. The block diagram of the real application is shown in Figure 20.

With this configuration, the contention of shared resources such as bus, SDRAM, and accelerators can be extracted. In addition, the delay of using the RM is counted in the contention section. This is justified because the only purpose of the RM is to provide system with flexible shared resource management. This configuration is referred to as “HW accelerated encoder” in the results. In addition, same kind of configuration with slave processor count from one to three is used to measure the video encoding speed as well.

Figure 45 illustrates the times different configurations measure. Simulation was used to measure plain hardware accelerator execution time ( $t_{hwacc}$ ). The wrapper introduces some overheads. Hence, the time the wrapper is active is denoted as ( $t_{wrapper}$ ). The wrapper configuration is the same in hardware-accelerated encoder and in small FPGA test system. Moreover, CPU timers measure time spent in hardware accelerated operation from software ( $t_{sw}$ ) in simple FPGA test system. In addition, time ( $t_{task}$ ) is the hardware-accelerated operation in encoder including the waiting time of wrappers.



**Figure 45. Time spans that HW monitor and CPU timers measure in different configurations.**

In addition to the HW monitor and CPU timers, the embedded logic analyzer was utilized in the accelerated video encoder in order to get timing information about the execution of the hardware-accelerated functions in clock cycle accuracy. Every sent address and data over HIBI is shown in the logic analyzer’s screen.

## Software video encoder

The fourth configuration is for measuring the software implementation execution times. The configuration utilizes software video encoder with two encoding slave CPUs, a master CPU, and SDRAM controller. Obviously, there are no hardware accelerators on the software video encoder. Hence, the HW monitor cannot be utilized. Instead, only software (SW) operated CPU timers measure the execution time. This is referred to as “SW encoder.”

## 7.2. Clock frequency and chip area

The execution speed is directly proportional to the used clock frequency. Gaining performance by increasing clock frequency is not always good idea, since it increases power consumption and might prevent system from running, when pushing the frequency to the limits of the chip. In ASIC implementation, the maximum clock frequency is always greater than in FPGA prototypes. This is because FPGAs use fixed logic and routing elements causing longer critical path, whereas in ASIC routes between logic are optimized.

Area of digital system in general means the amount of transistors the design uses when fitted to a silicon wafer, but it is measured in multiple of equivalent gates. The actual area can also be reported in square millimeters. The square millimeter area depends, for example, on the transistors' channel width in target technology. The area defines the size of the package required for the chip. Further, the complexity and size of the printed circuit board depends on the size and IO pin count of the package. Consequently, the size of the product itself depends on the size of the PCB.

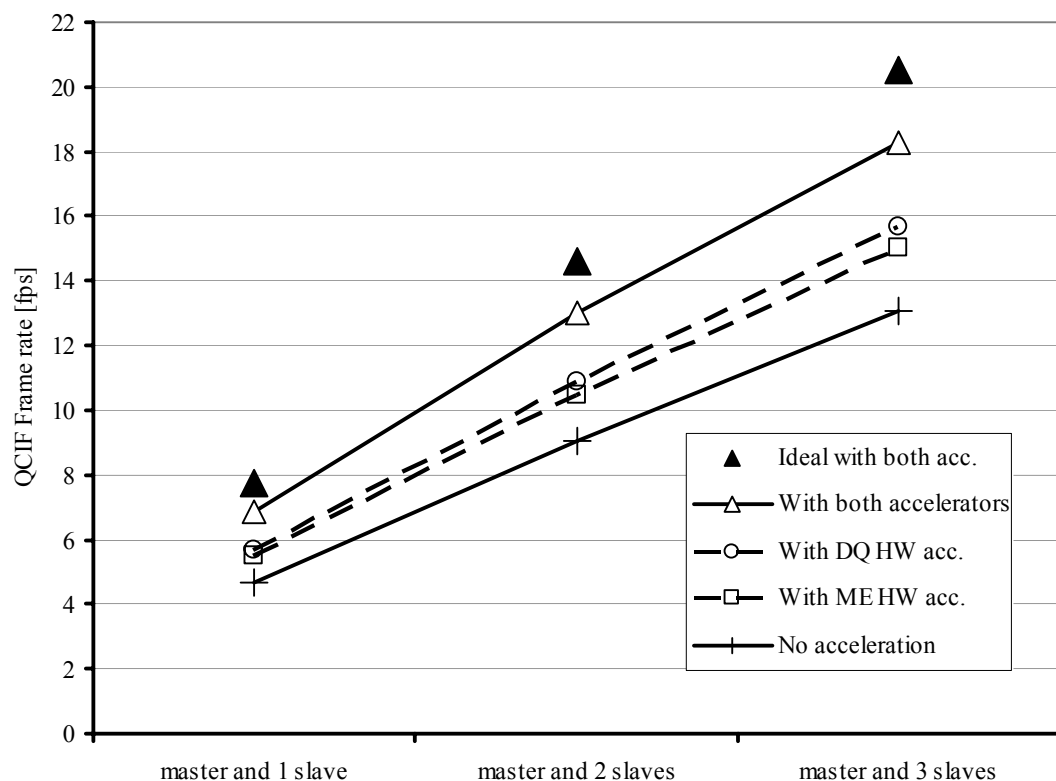
On FPGA, the amount of transistors is fixed and only the configurable logic blocks and routing channels are configured to perform the functionality of the design. Hence, the area determines if the design fits to the FPGA or not. Configurable logic blocks are called logic elements in Altera FPGA chips. In addition, the amount of memory used by the design on FPGA is counted in memory bits.

Determining the area and maximum clock frequency is a straightforward operation with proper EDA toolset. Synthesis tool gives out the approximations of the amount of logic gates and logic elements needed and the length of the critical path in time. The inverse of the critical path is the maximum clock frequency. The results depend on the target technology, and it must be set up before running synthesis. Performing placement and route operations in EDA tool gives more specified information, since the tool maps each gate to a logic element in the case of FPGA. This way the routing delays are also taken into account and the both results are accurate.

### 7.3. Video encoding speed

The encoding speed is shown in Figure 46 as a function of processor count. There is always a master and from one to three slave processors. The clock frequency of the processors and other hardware is 50 MHz. The curves define the hardware accelerator configuration; none, one, or both accelerators are utilized. The video encoding speed scales up flexibly with all four combinations of hardware accelerators. The DQ hardware accelerator alone provides a slightly better performance gain than the ME hardware accelerator.

The trend of scalability would not continue with accelerated systems if the number of processors were further increased. This is because the constant number of shared accelerators would cause the growth of the contention and saturate the curves. Starting from the minimum configuration with one slave CPU and no acceleration, one can see that it is more beneficial to add a single CPU than any acceleration. However, the case with both accelerators and two slave CPUs is faster than three slave CPUs without acceleration. DQ hardware acceleration provides 20-21% speed-up to the frame rate, ME hardware acceleration 15-17% and both together 40-47%. Thus, the system benefits more when using both accelerators and hence reaches maximum frame rate of 18.



**Figure 46. Video encoding speed with various hardware accelerator and processor configurations.**

The ideal speed-up obtained with zero integration overhead was calculated for the case of two slaves and both accelerators. The speed-up is 61% when compared to non-

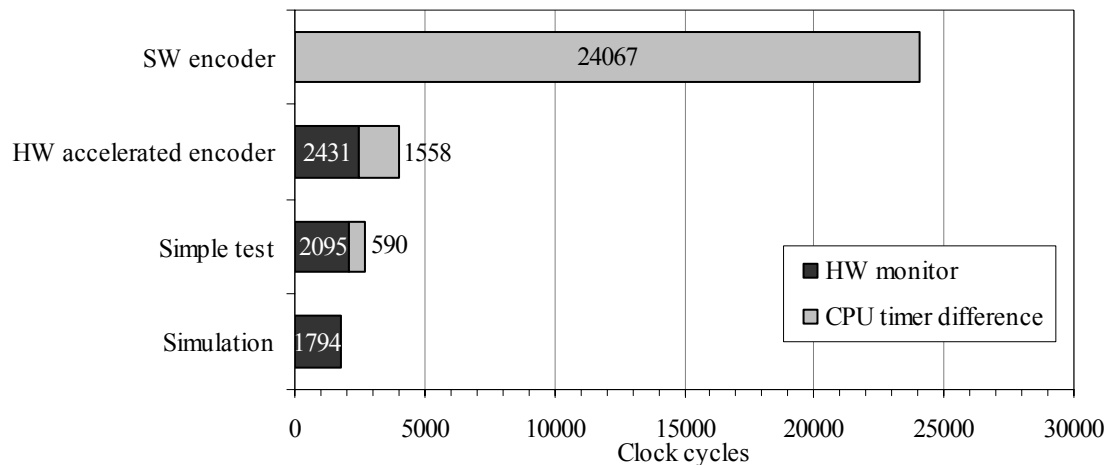
accelerated execution. The ideal encoding speed is 12% better than in real system. The ideal speed-ups in configurations of one and three slaves were obtained by scaling up the real results by the same 12%. Hence, the maximum performance without integration overheads would have been 21 frames per second (FPS).

The ideal encoding frame rate with both accelerators in case of two encoding slave processors was calculated from the execution times of the DQ and ME. The ideal frame rate was obtained by replacing the actual video encoder execution time with pure hardware acceleration time measured in simulation. The result was further compensated with estimated amount of parallel execution of software during hardware acceleration. The execution times of the accelerated functions in different test configurations are discussed next.

#### 7.4. DQ execution

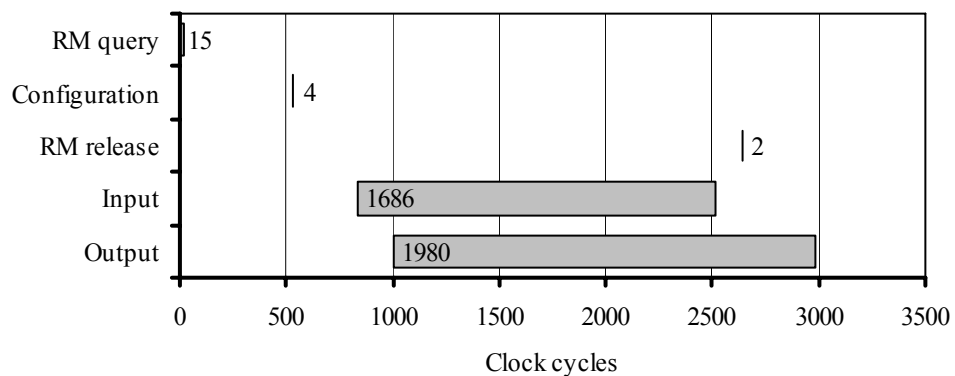
Figure 47 illustrates the results of the DQ function execution times from the software's point of view measured with the CPU timers and hardware monitor. HW monitor bar is the execution time from the wrapper's and hardware accelerator's point of view, whereas the *CPU timer difference* means the difference of CPU timer and HW monitor results. This is essential because CPU timer result contains also HW monitor results. In addition, it contains software overheads and RM query, though the RM queries only exist in the encoder. The simple FPGA test does not utilize RM.

Firstly, the total execution time in *simple test* is 50% longer than in *simulation*. The increase is due to the wrapper component, introduced HIBI communication, and software. The proportion of software can be seen in *CPU timer difference* of *simple test*. Secondly, the *HW accelerated encoder* takes 49% longer time in total than *simple test*. Here the increase is due to the contention of communication and the RM access, which also includes the waiting time for getting access to the accelerator if it is being used at the time of query. Hence, the average bus access time increases, which consequently adds delay to the DQ transactions. Overall, the overall HW encoding time is 122% longer than in simulation. Although the actual execution time is over doubled when compared to the simulation, it still is 17% of the software implementation execution time. Hence, the speed-up is 5.8x.



**Figure 47. DQ function execution time in different configurations.**

The embedded logic analyzer reveals more information concerning the hardware-accelerated execution of motion estimation. However, the software functionality is unknown at this point, since the logic analyzer only listens to HIBI signals. Figure 48 depicts the execution. The first transaction on HIBI is the RM query. The 15-cycle long bar includes the query sending, processing, and answer sending. As soon as the slave CPU has processed the RM answer, it sends configuration to the DQ followed by the data. It takes three to five hundred cycles to initiate the transmission due to the slowness of general-purpose processor. The input and output bars indicate the time that the transaction is active. This includes empty cycles on HIBI and the contention of waiting each other, since only one transmission is on at a time. In practice, the about 1500-cycle long period consists of 5-to-10-word-long transmissions to the wrapper.



**Figure 48. Execution profile of hardware-accelerated DQ function.**

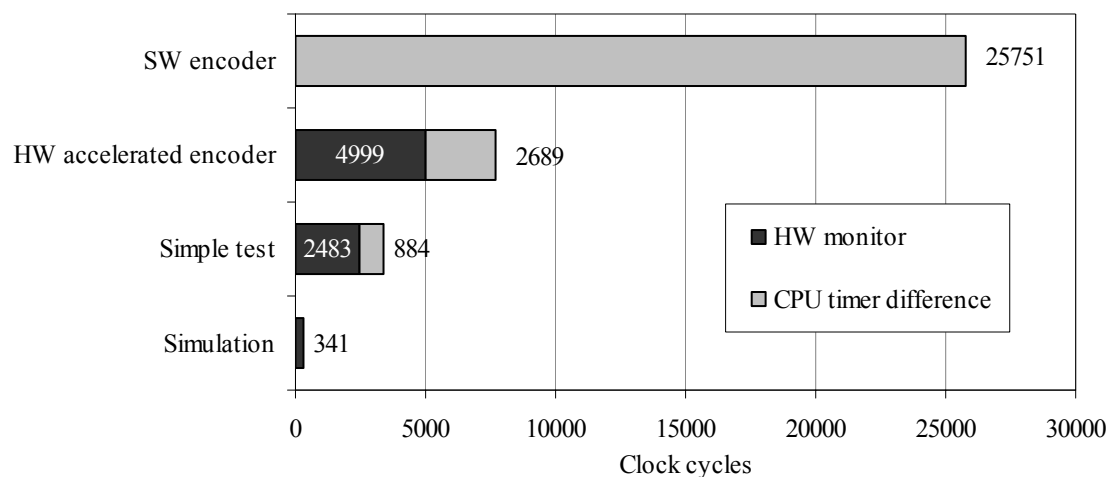
After the RM query, there is about 500 clock cycles before the slave CPU sends the configuration. This time could be shortened by modifying the software driver. It is now implemented to wait for interrupt. However, calling the interrupt handler causes overhead to the execution. This driver function could have been implemented with

polling the register that indicates the new data from HIBI. This could decrease the latency.

## 7.5. ME execution time

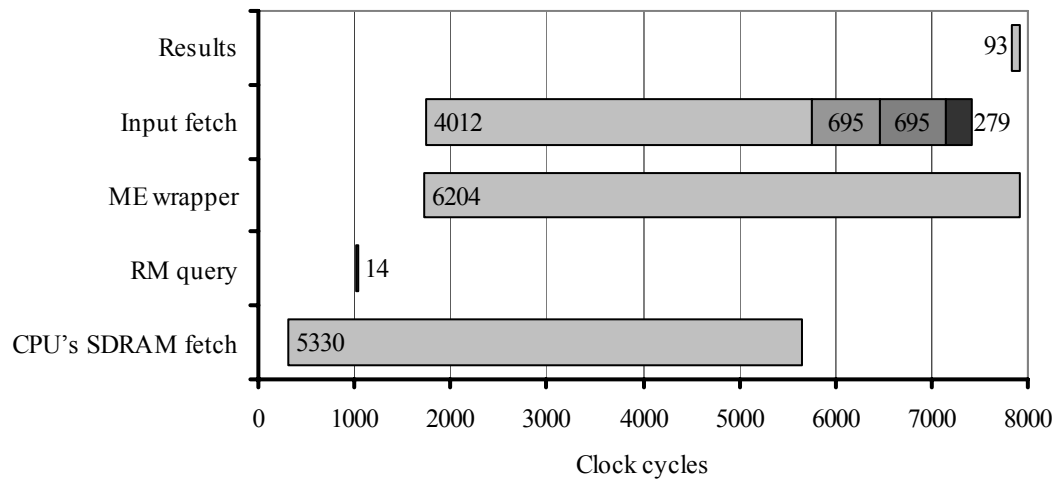
In measurements of ME hardware accelerator, the results between the three test configurations vary considerably as shown in Figure 49. When compared to the simulation, the total execution time in *simple test* shows 9.9x increment and in *HW accelerated encoder* 22.5x increment. In *simple test* and *HW accelerated encoder*, the wrapper has to do the permutation operation to the result macroblock that causes extra latency. In addition, ME hardware accelerator utilizes 128-bit wide input and output bus, whereas the SDRAM can only provide it with 32 bits. Moreover, the data fetching from the SDRAM delays the beginning of the ME operation itself.

In *simple test* configuration, the software and data delivery increase the execution time like in the case of DQ. In encoder, large amount of data is transferred to and from the SDRAM controller over the on-chip network, and the HIBI bus utilization grows when parallel operations are running in the system. Hence, the SDRAM data fetching operation slows down dramatically. This explains the more than doubled execution time in *HW accelerated encoder* than in *simple test* situation in total. Using wider or more efficient SDRAM would be beneficial but was not possible in the current prototype, as the development board only includes 32-bit-wide external memory chip.



**Figure 49. ME function execution times in different configurations.**

Based on the logic analyzer's execution profile, the SDRAM is the bottleneck of ME execution. Using the logic analyzer is manual work. The most typical traffic scenario of the several measures is presented in Figure 50. The only difference between scenarios was that CPU's SDRAM fetch was only initiated slightly earlier or later.



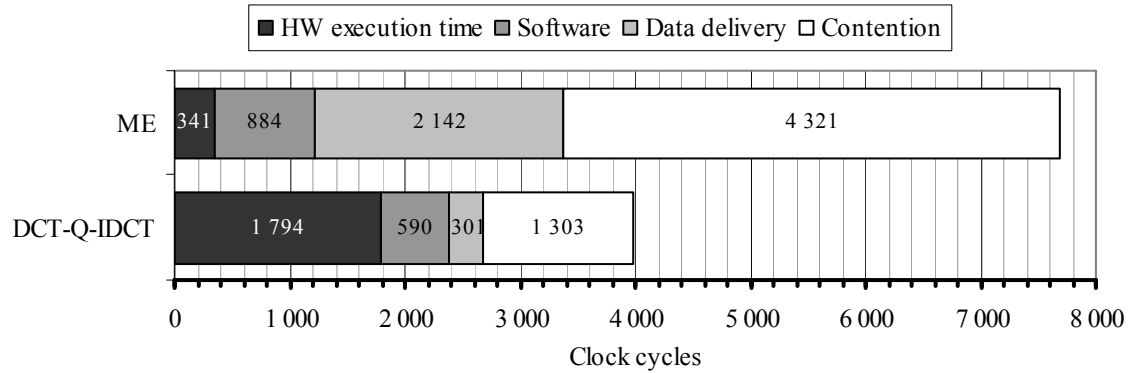
**Figure 50. Execution profile of hardware accelerated motion estimation measured on HIBI.**

Operation *input fetch* consists of four fetches that are depicted in different colors in Figure 50. The last bar indicates the fetching time of current macroblock and the three previous bars are the three slices of a search area. Slices are equal in size, so they should take the same time to transfer. However, the first takes nearly six times longer. This is due to the parallel data operation, seen in the figure as *CPU's SDRAM fetch* that begins before the acceleration itself and ends just before the fetch of first slice ends.

The ME wrapper bar starts when the wrapper receives the request from CPU and ends when the last result has been sent to the CPU. As seen in the figure, the results take only little time to transfer. The gap between the end of *input fetch* bar and the beginning of *results* bar is the pure hardware accelerator execution time including the permutation in ME. However, the permutation operation contributes clock cycles in the both bars *input fetch* and *results*.

## 7.6. Factors of the integration overhead

The execution times of the accelerators in the video encoder are further analyzed based on the data of HW monitor and CPU timers. The factors of the integration overhead are extracted in Figure 51. Firstly, *HW execution time* means the pure time that the accelerator takes to perform its duty. Secondly, *software* means the overhead caused by the driver function call and the interrupt handling that occurs when the results arrive back to the CPU. There is no operation system, which would slow down the execution even more. Thirdly, *data delivery* means the waiting time and latency of data transactions in the system without any extra traffic on the HIBI. Finally, *contention* means all kind of contention that appears in the HW accelerated encoder, when there are other parallel operations running as well. This contention includes waiting for HIBI and SDRAM access as well as RM queries.



**Figure 51. Factors of integration overhead in ME and DQ functions.**

The pure ME hardware accelerator execution time is only 4% from the overall function execution time. It plays minor role when compared to the factors of integration overhead. The software expense is tolerable, but data delivery delay and contention of shared resources are immense. These are due to the SDRAM and bus width issues pointed out earlier.

In contrast to the ME, the hardware execution time of the DQ is in the major role. Because of the small bus widths and well-parallelized computation and communication, the DQ does not consume time waiting for data in *simple test*. However, in HW accelerated encoder it is forced do wait the interconnection that is shared with other components. In addition, the RM query delay contributes clock cycles in the contention bar.

The contention is relatively larger with the ME because the ME has longer data transfers, and instead of only competing for HIBI like the DQ, the ME has to wait for the access to SDRAM as well. Moreover, the total execution time excluding the contention is slightly larger than that of the DQ. Due to the longer execution time, it is more probable that the ME is in use when another processor tries to access it. In addition, this increases the average processor waiting time for getting access to the accelerator if it is being used at the time of the query.

DCT and ME have similar SW runtime, but ME has clearly shorter hardware execution time. However, the overall time needed for ME during the encoding is larger due to bigger overheads. Hence, comparing the accelerators based on their “ideal” times in testbench proves to be misleading. Similarly, running accelerator with higher frequency does not offer notable speed-up since the overheads remain the same.

The proportion of actual hardware accelerator execution time from function execution time is 4% and 45% for ME and DQ respectively. Hence, speeding up the ME accelerator execution time would have only slight effect on the whole function execution. This renders lower latency and execution time optimizations meaningless.

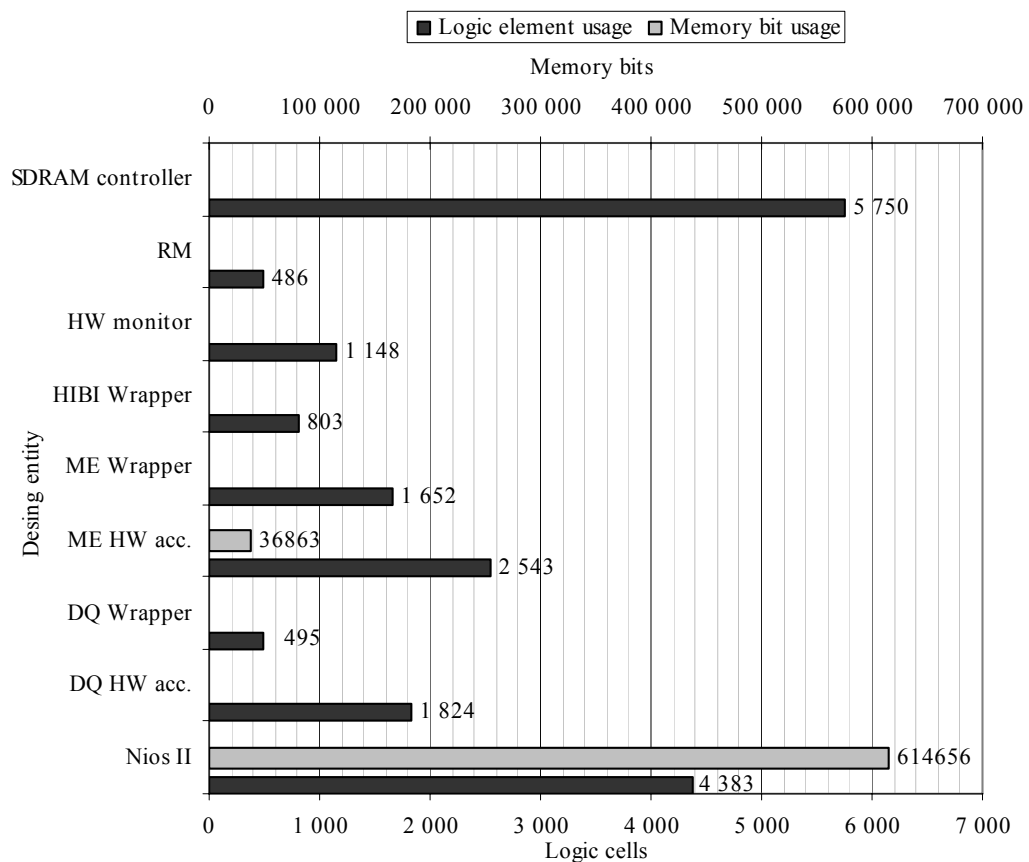
For speculation, [34] introduces a new architecture for the SAD operation, that is faster and smaller than the previously proposed. Using such architecture in the ME

hardware accelerator would probably optimize a couple of clock cycles from its execution time. However, this would not result in any notable speed-up to the encoder because the proportion of integration overheads is as large as 96%. Since the proportion of the integration overhead of DQ is much smaller (55%), optimizing DQ operation would benefit the encoder more. For instance, [35] propose fast and flexible optimizations to IDCT that could be used to gain encoding speed-up.

In [36], two cases are presented: error-correction coder/decoder and multiply/accumulate. It is found that hardware accelerated execution on FPGA brings speed-up of 1.70x and 1.27x, accounting the overheads, when the rest of the software lies in general-purpose processor. Without overheads, the pure hardware acceleration execution induces speed-up of 3.28x and 2.19x. Thus, the overheads are 48% and 42%, respectively. However, they use higher clock frequency for the processor than for the accelerators, which probably increases overheads. In addition, FPGA and processor are discrete components, not integrated in a chip.

## 7.7. Area usage on FPGA

The logic element and memory bit consumption of the system is depicted in Figure 52. The areas of the hardware accelerators, wrappers, HIBI wrapper, resource manager, hardware monitor, and Nios II slave processor are included.



**Figure 52. FPGA area usage per design entity.**

The SDRAM controller is the largest component in the system when considering only the logic element consumption. It has big FIFO buffers to store data and configuration messages. The resource manager is the smallest component in the system. The size depends on the amount of supported IP types. In this case, it supports two types.

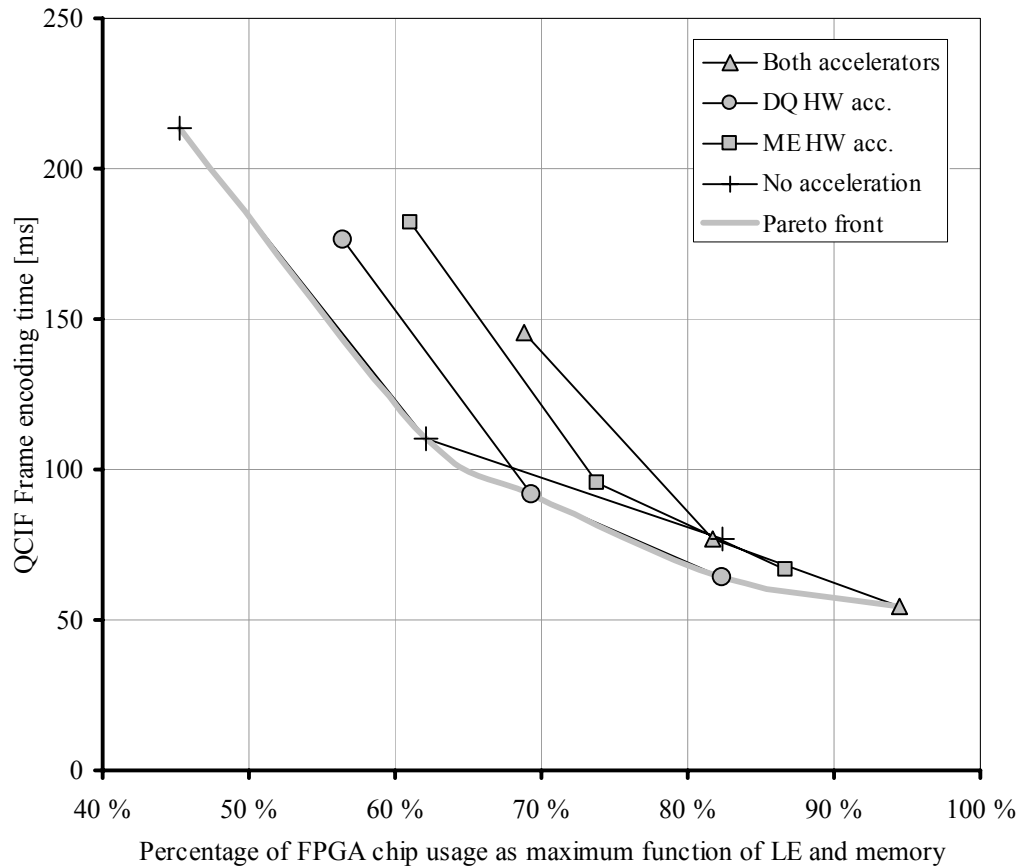
The wrapper components are large in area, when compared to the accelerators. The DQ wrapper size is 27% of the accelerator area, whereas it is 65% in the case of ME. The permutation logic and buffers greatly increase the logic element usage of ME wrapper. However, as the ME hardware accelerator utilizes on-chip memory, the areas of the wrapper and the accelerator are not fully comparable. The logic that handles the protocol of SDRAM controller further increases the area of ME wrapper. Nios II processor area includes the Nios-to-HIBI DMA controller.

The wrapper components for ME and DQ components could have been reduced by implementing the FIFO buffers and permutation matrix as memory instead of logic registers, since there is relatively more memory bits free in the system than logic elements. However, implementing them as logic register was easier and thus the integration was performed in shorter time.

## 7.8. Performance per area

Performance per area is presented in Figure 53. The vertical axis represents encoding time of a frame, whereas the horizontal axis is percentage of FPGA chip usage. The two variables of FPGA chip utilization complicate the usage calculation slightly, since usages of logic elements and memory bits have to be weighted properly. The figure presents the highest relative usage as an area. For instance, if a design consumes 20% of LEs and 40% of memory bits, it would use 40% of the total chip area. This is reasonable in case of FPGA chip, because the design is limited by both the maximum LE and memory bit counts. In other words, the FPGA chip usage percentage is a max-function of LE and memory usage.

Each presented configuration has three points. From left to right, the points have one, two, or three slave processors in addition to a master processor. Pareto front connects a series of *pareto* points. They are candidates for the point of the best performance per area ratio. *No acceleration* configuration provides best performance per area ratio when chip usage is less than 65%. Since the ME is bigger and less effective, it will always lose to DQ. Using only DQ is actually the best solution in range 65% to 85%. However, the configuration of both accelerators with three slaves provides the most efficient encoding. In the case of FPGA chip, it is reasonable to maximize utilization. Thus, this configuration can be interpreted as optimal.



**Figure 53. A frame encoding time as function of FPGA chip usage.**

It seems that further increasing number of slave processors benefits the accelerated performance per area ratio. However, measuring this is impossible within the limits of the current FPGA chip. Moreover, there is a saturation point for the number of slaves. Beyond the point, one hardware accelerator of a kind is not anymore enough to serve the CPUs. Thus, the number of hardware accelerators would have to be increased as well. However, this would not cause further complications due to flexible resource manager unit.

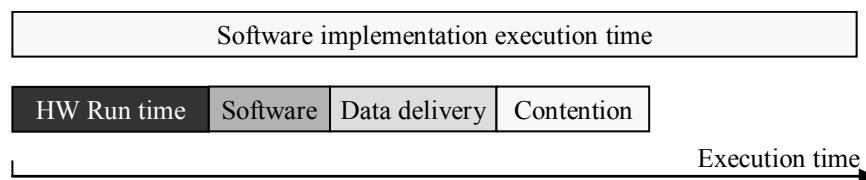
If the wrapper components were implemented as memory instead of logic elements, the relative areas would have been slightly smaller. However, this would have only minor effect to the area per performance figure since there is only one hardware accelerator of a kind in the system. The meaning of such optimization would grow if the number of hardware accelerators were increased in the video encoder.

## 8. CONCLUSIONS

This Thesis presented the integration of hardware accelerators and measured how the integration affected the execution of accelerated operations and the system performance in the case study of SoC video encoder. Different integration approaches were evaluated for the SoC architecture and the most suitable was applied. The process included designing the wrapper components for the accelerators and implementation of the resource manager as well as hardware monitor for measuring execution time. Finally, measurements and analysis of the integrated video encoder revealed the benefit of hardware acceleration and overhead factors that affected the performance.

With the prototype on Stratix 1S40 FPGA chip, the hardware accelerated video encoder in general was found better than software encoder in terms of encoding speed. Whereas the software encoder performed 13 FPS, the accelerated system reaches 18 FPS. Ideal encoding speed would have been 21 FPS without performance overheads of integration. The performance was increased by 40% in total compared to the original implementation.

The integration overheads decreased the speed-up obtained from hardware accelerators. Three factors were extracted from the overhead: low-level software overhead, data delivery delays, and shared resource contention. These are depicted in Figure 54, where proportions are only conceptual.



**Figure 54. The factors of integration overhead.**

Based on execution time analyses, hardware-accelerated ME took 7700 clock cycles whereas the software implementation took 26000. In the case of DQ, the execution times were 24000 for software and 4000 cycles for hardware accelerated implementation. Moreover, execution times of accelerated operations DQ and ME were 45% and 4%, respectively, of the total accelerated operation time. The major integration

overhead factor for ME was contention with 56%. Hardware accelerator runtime dominated the accelerated DQ execution. Its largest overhead was also contention with 33%. The factors depend on the compatibility of the hardware accelerator and the architecture. In addition, the actual performance gain of the accelerators differs from that the background data indicate: initially, the ME accelerator seemed to be more efficient, but performed worse in the encoder. In addition, the work taught that optimizing a few clock cycles in the computation is only marginal, if the integration overhead contributes huge latency and drops throughput due poor interface or IP component compatibility.

Firstly, the work was successful, because the case study showed that hardware acceleration increased system performance significantly and three integration overhead factors were found. Secondly, the work was important, since the integration overheads are significant for system integrator, and the overhead analysis is needed in automatic design space exploration tools in order to choose the best performing IP block to a system. Lastly, the results are reliable since a FPGA prototype was used. However, they are very case dependent, since every integration scenario is different. The results prove this, since even within the same system, two highly different proportions of overhead were measured.

It seems that the SDRAM is a bottleneck in the current implementation. By boosting the SDRAM performance in the future, it is possible to reduce integration overhead in the system – especially the contention factor. In addition, by using larger FPGA chip or several FPGA chips together will make it possible to instantiate even more slave CPUs and multiple hardware accelerators. Moreover, in order to choose the best performing IP block to a system in general, it is necessary to predict the integration overheads beforehand. This requires new methodology and tools for IP block evaluation that will be developed in the future.

## REFERENCES

- [1] F. Vermeulen, L. Nachtergaele, F. Catthoor, D. Verkest, and H. De Man, "Flexible hardware acceleration for multimedia oriented microprocessors," Proceedings of International Symposium on Microarchitecture (MICRO), ACM Press, New York, USA, December 2000, pp. 171-177
- [2] O. Silven and K. Jyrkkä, "Observations on Power-Efficiency Trends in Mobile Communication Devices," Proceedings of Embedded Computer Systems: Architectures, Modeling, and Simulation 5th International Workshop (SAMOS), Springer, Samos, Greece, July 2005, pp. 142-151
- [3] P. Kuhn, "Algorithms, Complexity Analysis and VLSI Architectures for MPEG-4 Motion Estimation", Kluwer Academic Publishers, Boston, USA, 1999
- [4] Altera Corporation, "Nios II Processor Reference Handbook", version NII5V1-6.1, November 2006.
- [5] A. Jerraya and W. Wolf, eds., "Multiprocessor Systems-on-Chips," Morgan Kaufmann/Elsevier, St Louis, USA, 2004
- [6] J. Rowson and A. Sangiovanni-Vincentelli, "Interface-based design," Proceedings of Design Automation Conference (DAC), ACM, Anaheim, USA, June 1997, pp. 178-183
- [7] G. Martin, "Design methodologies for system level IP," Proceedings of Design, Automation and Test in Europe (DATE), IEEE Computer Society Press, Paris, France, March 1998
- [8] K. Keutzer, S. Malik, A. Richard Newton, J. Rabaey, and A. Sangiovanni-Vincentelli, "System-level design: orthogonalization of concerns and platform-based design," IEEE Transactions Computer-Aided Design of Integrated Circuits, December 2000, Volume 19, Issue 12, pp. 1523–1543
- [9] F. R. Wagner, W. O. Cesário, L. Carro, and A. A. Jerraya, "Strategies for the integration of hardware and software IP components in embedded systems-on-chip," Integration, the VLSI Journal, September 2004, Volume 37, Issue 4, pp. 223-252
- [10] M. Keating and P. Bricaud, "Reuse Methodology Manual: For System-on-a-Chip Designs," 3<sup>rd</sup> edition, Kluwer Academic Publishers, Norwell, 2002
- [11] ARM website, "Fabric IP solutions," online: [http://www.arm.com/products/solutions/fabric\\_overview.html](http://www.arm.com/products/solutions/fabric_overview.html), read 14<sup>th</sup> December 2006
- [12] IBM CoreConnect Bus Architecture, online: <http://www-3.ibm.com/chips/products/coreconnect/index.html>, read 15<sup>th</sup> February 2007
- [13] OCP-IP Alliance, "Open Core Protocol Specification, Release 1.0," Portland, USA, 2001
- [14] Wishbone SoC Interconnect Architecture, online: <http://www.opencores.org/projects.cgi/web/wishbone/wishbone>, read 14<sup>th</sup> December 2006
- [15] SMART Interconnects solutions, online: <http://www.sonicsinc.com>, read 15<sup>th</sup> February 2007

- [16] SPIRIT Consortium, IP-XACT specification, online: <http://www.spiritconsortium.org/>, read 1<sup>st</sup> March 2007
- [17] Tero Kangas, Petri Kukkala, Heikki Orsila, Erno Salminen, Marko Hännikäinen, Timo D. Hämäläinen, Jouni Riihimäki, Kimmo Kuusilinna, "UML-based Multi-Processor SoC Design Framework," *Transactions on Embedded Computing Systems*, ACM, Hingham, USA, May 2006, Volume 5, Issue 2, pp. 281-320
- [18] A. Evans et al., "Functional Verification of Large ASICs," *Proceedings of Design Automation Conference (DAC)*, ACM Press, San Francisco, USA, 1998, pp. 650-655
- [19] E. Salminen, T. Kangas, J. Riihimäki, V. Lahtinen, K. Kuusilinna, and T. Hämäläinen, "HIBI Communication Network for System-on-Chip," *Journal of VLSI Signal Processing-Systems for Signal, Image, and Video Technology*, vol. 43, pp. 185-205.
- [20] A. Kulmala, E. Salminen, and T. D. Hämäläinen, "Distributed Bus Arbitration Algorithm Comparison on FPGA Based MPEG-4 Multiprocessor SoC," *Norchip 2006*, IEEE, Linköping, Sweden, November 2006, pp. 167-170
- [21] Altera Corporation, "Nios 3.0 CPU datasheet," October 2004
- [22] A. Kulmala, O. Lehtoranta, T. D. Hämäläinen, and M. Hännikäinen, "Scalable MPEG-4 Encoder on FPGA Multiprocessor SOC," *EURASIP Journal on Embedded Systems*, Hindawi Publishing Corporation, June 2006, Volume 2006, Issue Field-Programmable Gate Arrays in Embedded Systems, pp. 1-15
- [23] ANSI/IEEE Std 1076-1993, "IEEE Standard VHDL Language Reference Manual," IEEE, 1994
- [24] IEEE Std 1364-1995, "IEEE Standard Hardware Description Language Based on the Verilog Hardware Description Language," IEEE, 1995
- [25] Mentor Graphics Corporation, "ModelSim SE User's Manual," Software version 6.2a, June 2006
- [26] H. Krupnova, "Mapping Multi-Million Gate SoCs on FPGAs: Industrial Methodology and Experience," *Proceedings of Design, Automation, and Test in Europe (DATE)*, IEEE Computer Society, Washington, USA, February 2004, Volume 2, pp. 1236-1241
- [27] Altera Corporation, "Stratix Device Handbook," July 2005
- [28] ITU-T, "Draft H.263: Video coding for Low Bitrate Communication", May 1996
- [29] I. E. G. Richardson, "H.264 and MPEG-4 Video Compression," Wiley, England, 2003
- [30] M-T. Sun and A. Reibman, "Compressed Video over Networks," Marcel Dekker, New York, USA, 2001
- [31] I. Ahmad, Y. He, and M. L. Liou, "Video Compression with Parallel Processing," *Parallel Computing in Image and Video Processing*, Elsevier Science Publishers, Amsterdam, Netherlands, August 2002, Volume 28, Issue 7-8, pp. 1039-1078
- [32] O. Lehtoranta, T. D. Hämäläinen, and V. Lappalainen, "Parallel implementation of video encoder on quad DSP system," *Microprocessors and Microsystems*, Elsevier, February 2002, Volume 26, Issue 1, pp. 1-15

- [33] J. Vanne, E. Aho, T. D. Hämäläinen, and K. Kuusilinna, "A High-Performance Sum of Absolute Difference Implementation for Motion Estimation," *IEEE Transactions on Circuits and Systems for Video Technology*, July 2006, Volume 16, Issue 7, pp. 876-883
- [34] D. Guevorkian, A. Launiainen, P. Liuha, and V. Lappalainen, "Architectures for the sum of absolute differences operation," *Proceedings of IEEE Workshop on Signal Processing Systems Design and Implementation (SIPS)*, San Diego, USA, October 2002, pp. 57-62
- [35] J. Lee, N. Vijaykrishnan, and M. J. Irwin, "Inverse Discrete Cosine Transform Architecture Exploiting Sparseness and Symmetry Properties," *IEEE Transactions on Circuits and Systems for Video Technology*, May 2006, Volume 16, Issue 5, pp. 655-662
- [36] M. D. Edwards and J. Forrest, "Software acceleration using programmable hardware devices," *IEE Proceedings of Computers and Digital techniques*, January 1996, Volume 143, Issue 1, pp. 55-63