Thesis for the degree of Licentiate of Technology
Sundsvall 2006

Memory Synthesis for FPGA Implementation of Real-Time Video Processing Systems

Najeem Lawal

Supervisors: Professor Mattias O'Nils
Docent Bengt Oelmann
Doctor Benny Thörnberg

Electronics Design Division, in the
Department of Information Technology and Media
Mid Sweden University, SE-851 70 Sundsvall, Sweden

ISSN 1652-8948
Mid Sweden University Licentiate Thesis 14

ISBN 91-85317-30-6

Akademisk avhandling som med tillstånd av Mittuniversitetet i Sundsvall framläggs till offentlig granskning för avläggande av teknologie licenciates examen i elektronik torsdagen den 22 Nov. 2006, klockan 10:30 i sal O111, Mittuniversitetet Sundsvall. Seminariet kommer att hållas på engelska.

Memory Synthesis for FPGA Implementation of Real-Time Video Processing Systems

Najeem Lawal

Electronics Design Division, in the
Department of Information Technology and Media
Mid Sweden University, SE-851 70 Sundsvall
Sweden

Telephone: +46 (0)60 148561

Printed by Kopieringen Mittuniversitetet, Sundsvall, Sweden, 2006

ABSTRACT

In this thesis, a both method and a tool to enable efficient memory synthesis for real-time video processing systems on field programmable logic array are presented. In real-time video processing system (RTVPS), a set of operations are repetitively performed on every image frame in a video stream. These operations are usually computationally intensive and, depending on the video resolution, can also be very data transfer dominated. These operations, which often require data from several consecutive frames and many rows of data within each frame, must be performed accurately and under real-time constraints as the results greatly affect the accuracy of application. Application domains of these systems include object recognition, object tracking and surveillance.

Developments in field programmable gate array (FPGA) have been the motivation for choosing them as the platform for implementing RTVPS. Essential logic resources required in RTVPS operation are currently available optimized and embedded in modern FPGAs. One such resource is the embedded memory used for data buffering during real-time video processing. Each data buffer corresponds to a row of pixels in a video frame, which is allocated using a synthesis tool that performs the mapping of buffers to embedded memories. This approach has been investigated and proven to be inefficient. An efficient alternative employing resource sharing and allocation width pipelining will be discussed in this thesis.

A method for the optimal use of these embedded memories and, additionally, a tool supporting automatic generation of hardware descriptions language (HDL) codes for the synthesis of the memories according to the developed method are the main focus of this thesis. This method consists of the memory architecture, allocation and addressing. The central objective of this method is the optimal use of embedded memories in the process of buffering data on-chip for an RVTPS operation. The developed software tool is an environment for generating HDL codes implementing the memory sub-components.

The tool integrates with the Interface and Memory Modelling (IMEM) tools in such a way that the IMEM's output - the memory requirements of a RTVPS - is imported and processed in order to generate the HDL codes. IMEM is based on the philosophy that the memory requirements of an RTVPS can be modelled and synthesized separately from the development of the core RTVPS algorithm thus freeing the designer to focus on the development of the algorithm while relying on IMEM for the implementation of memory sub-components.

SAMMANDRAG

I denna avhandling presenteras en metod och ett verktyg för möjliggörandet av effektiv minnessyntes för vidoebearbetande system i realtid på Field Programmable Gate Array (FPGA). I ett system som bearbetar video i realtid (RTVPS) upprepas en mängd processer i varje bildruta i en videosekvens. Dessa processer är ofta beräkningsintensiva och, beroende på videoupplösningen, kan de också vara mycket dataöverföringsstyrda. Processerna, som ofta kräver data från en mängd konsekutiva bildrutor och många dataserier inom varje ruta, måste genomföras exakt och under realtidsbegränsningar, då resultaten i hög grad påverkar tillämpningens exakthet. Tillämpningsområden för dessa system innefattar igenkänning av föremål, spårning av föremål samt övervakning.

Utvecklade produkter inom FPGA har motiverat användandet av dessa som plattform för tillämpning av RTVPS. De nödvändiga logikresurser som krävs för RTVPS-processer är för tillfället tillgängliga, optimerade och inbyggda i modern FPGA. En sådan resurs är det inbyggda minne som används för datalagring under videoprocessning i realtid. Varje datalager motsvarar en rad pixlar i en videoruta som automatiskt allokeras på FPGAs. Denna metod har undersökts och visat sig vara effektiv. Ett effektivt alternativ som utnyttjar resursdelning och anslag vid rörledning diskuteras i denna avhandling.

En metod för optimal användning av dessa inbäddade minnen och ett verktyg som stöder automatisk generering av HDL-koder för minnessyntes enligt den utvecklade metoden är fokus för denna avhandling. Denna metod består av minnesarkitektur, allokering och adressering. Metodens centrala mål är optimal användning av inbäddade minnen under lagring av data på chip för en RTVPS-operation. Den utvecklade mjukvaran är en miljö för att generera HDL-koder, där minneskomponenter tillämpas.

Verktyget integreras med IMEM-verktyg (Interface and Memory Modelling) på ett sådant sätt att IMEM:s utdata – minneskraven för ett RTVPS, importeras och behandlas för att generera HDL-koderna. IMEM baseras på filosofin att minneskraven för ett RTVPS kan modelleras och syntetiseras separat från utvecklandet av den ursprungliga huvudalgoritmen för RTVPS och därigenom ge designern frihet att fokusera på utvecklingen av algoritmen, medan IMEM används för tillämpning av minneskomponenter.

ACKNOWLEDGEMENTS

First of all I would like to greatly appreciate my supervisors Prof. Mattias O'Nils, Doc. Bengt Oelmann and Dr. Benny Thörnberg for their academic and scientific guidance and inspirations, and for giving me the opportunity to study for Ph.D. Prof. Hans-Erik Nilsson, Dr. Jerzy Kirrander and Håkan Norell are greatly acknowledged for their contributions and inspirations. I am grateful to Fanny Burman and Malin Bydén-Sjöbom for their kind supports.

I would like to thank all my colleagues at the Mid Sweden University namely, Suliman, Jon, Niklas, Cao, Göran, Peng, Kannan, Mats, Krister, Janne, Johan, Fredrik, Kent, Henrik, Claes, Börje, Lotta, Andreas and Magnus for their supports and companionship during this research and for making my life in Sweden more interesting.

I would also like to express my gratitude to the Mid Sweden Unviersity, the Swedish KK foundation and ARTES Graduate School for their financial supports.

I would like to appreciate Professor Petru Eles for accepting to be my opponent.

Finally, I am grateful to my family and friends for their supports.

Sundsvall, Nov 2006

Najeem Lawal

TABLE OF CONTENTS

ABBREVIATIONS AND ACRONYMS

ALU	....	Arithmetic Logic Unit
ASIC	....	Application Specific Integrated Circuit
ASIP	....	Application Specific Instruction set Processor
CAD	....	Computer Aided Design
CLB	....	Configurable Logic Block
CPLD	....	Complex PLD
CPU	....	Central Processing Unit
DRAM	....	Dynamic RAM
DSP	....	Digital Signal Processing
FIFO	....	First In First Out
FPGA	....	Field Programmble Gate Array
GMO	....	Global Memory Object
HDL	....	Hardware Description Language
HLL	....	High Level programming Language
IMEM	....	Interface and Memory Modeling
IP	....	Intellectual Property
LUT	....	Look Up Table
PLD	....	Programmble Logic Device
RAM	....	Random Access Memory
RISC	....	Reduced Instruction Set Computer
RTVPS	....	Real-Time Video Processing System
SLWC	....	Sliding Window Controller
SRAM	....	Static RAM
VHDL	....	VLSI HDL
VIP	....	Video/Image Processing
VLSI	....	Very Large Scale Integration

LIST OF FIGURES

LIST OF TABLES

LIST OF PAPERS

This thesis is mainly based on the following five papers, herein referred to by their Roman numerals:

Paper I	**RAM Allocation Algorithm for Video Processing Applications on FPGA**, Najeem Lawal, Benny Thörnberg, Mattias O'Nils and Håkan Norell, Accepted for publication in *Journal of Circuits, Systems and Computers*., Vol. 15, No. 5, October 2006.
Paper II	**Address Generation for FPGA RAMs for Efficient Implementation of Real-Time Video Processing Systems**, N. Lawal, B. Thörnberg, M. O'Nils, *Proceedings of the Conference on Field Programmable Logic and Applications*, Tampere, Finland, 2005, pp. 136 - 141. ISBN 0-7803-9362-7
Paper III	**Embedded FPGA Memory Requirements for Real-Time Video Processing Applications** Najeem Lawal and Mattias O'Nils, *Proceedings of the 23rd Norchip Conference, Oulu, Finland* November 2005, pp. 206 - 209. ISBN 1-4244-0064-3
Paper IV	**Comparison of FPGA and DSP performances in neighbourhood oriented real-time video processing** Najeem Lawal, Håkan Norell and Mattias O'Nils, *Submitted to Transactions of VLSI Systems Special Section on Configurable Computing Systems*,
Paper V	**Automatic Generation of Spatial and Temporal Memory Architectures for Embedded Video Processing Systems**, H. Norell, N. Lawal and M. O'Nils, Accepted for publication in *European Association for Signal and Image Processing (EURASIP) Journal on Embedded Systems*.

Related papers not included into this thesis:

Evaluation of embedded RAM characteristics for FPGA implementation of real-time image processing systems,
J. Rojas, N. Lawal and M. O'Nils,
In manuscript

C++ based System Synthesis of Real-Time Video Processing Systems targeting FPGA Implementation,
M. O'Nils, B. Thörnberg and N. Lawal,
In Proceeding of FPGAworld Conference, Nov 2007.

1 INTRODUCTION

This thesis concerns the memory synthesis for the real-time implementation of video processing systems on FPGA. Real-Time Video Processing System (RTVPS) will first be introduced, followed by a brief introduction to Field Programmable Gate Array (FPGA), which will be compared to other implementation platforms. A justification for the choice of FPGA as the target platform for the work in this thesis will be given. The motivation for embarking on the research work presented in this thesis is also provided. In the later sections, related works and the contributions of this thesis are presented.

1.1 REAL-TIME VIDEO PROCESSING SYSTEM (RTVPS)

In an RTVPS the video signal is processed sufficiently quickly so that the rate of generating output pixels matches the rate of receiving input pixels. Hence there is a throughput of one pixel per clock cycle. Thus after an initial delay, the system enters a state during which a pixel is being received at the input side and, at the same time, a pixel is being produced at the output side. This does not, however, imply that this output pixel is the result of the newly received input pixel since there would be delays due to data buffering and pipelines in the computation.

A common feature in RTVPS is that the majority of the operations are neighbourhood oriented and thus require buffers for the pixel data required in the neighbourhood [1]. A neighbourhood of pixels constitutes a set of pixel data from which an RTVPS operator in the processing algorithm calculates an output pixel corresponding to the neighbourhood's central pixel. The neighbourhood is built around each pixel in the input image to generate an output image. The consequence is that a large amount of data buffers (line- and frame-buffers) are required depending on the size of the video frame and the operation window to ensure that all the required pixel data for each operation are available. Line buffers are used to store rows of pixels in the spatial neighbourhood. A spatial neighbourhood normally has dimensions of *M*-by-*N*, where *M* and *N* are odd values such that the central pixel is symmetrical about any axis. *N* and *M* denote the height and width of the spatial neighbourhood and usually determine the number of line buffers and delay elements required by the spatial neighbourhood operator respectively. Frame buffers are used to store images in the temporal neighbourhood. A temporal neighbourhood normally has dimensions of *L*-by-*M*-by-*N*, where *M* and *N* are defined as above and *L*, also an odd value, denotes the temporal depth of the neighbourhood. *L* determines the number of frame buffers in the temporal neighbourhood. Line buffers are usually allocated to on-chip memories while external memories are required for frame buffers. The size of each element in these buffers depends on the dynamic range of the video signal. Hence a 5-by-5 spatial neighbourhood requires four line buffers while two line buffers are required by a 3-by-3 neighbourhood. In the temporal domain, a neighbourhood of seven frames will require six frame buffers. An efficient data management tool is required since memory accesses generally constitute major bottlenecks.

1.2 FIELD PROGRAMMABLE GATE ARRAY (FPGA)

Reconfigurable computing involves the use of reprogrammable hardware- / software-based devices such as a custom-built computing machine in order to implement the current functional demands of the systems [2]. This means that as the system requirements increase, more modules can be added to extend the computational capability of the system to meet the new requirements. The obvious advantages are that new chips would not have to be built each time the system requirements change and also the opportunity to customize the device through Intellectual Property (IP) reuse for different functions. Hence, with a library of IP cores, different components can be glued together to meet the system requirements. The sacrifice is that the platform is required to accommodate a very wide range of possible implementation updates and application areas, which means there is no guarantee that the total capacity of the devices will always be used and hence there is rarely optimal usage of resources. When compared to ASIC and ASIP, reconfigurable devices tend have low performance since there are based on generic hardware architecture which has not been optimized for any specific application domain.

Depending on the capacity and architecture of the constituent basic elements, reconfigurable hardware can be categorized as programmable logic devices (PLD), complex PLD (CPLD) and FPGA. An overview of technology, architecture and programming tools for programmable hardware devices is presented in [3]. FPGA programmability enables hardware designers to greatly reduce the overall product time-to-market as shown in Fig. 1.

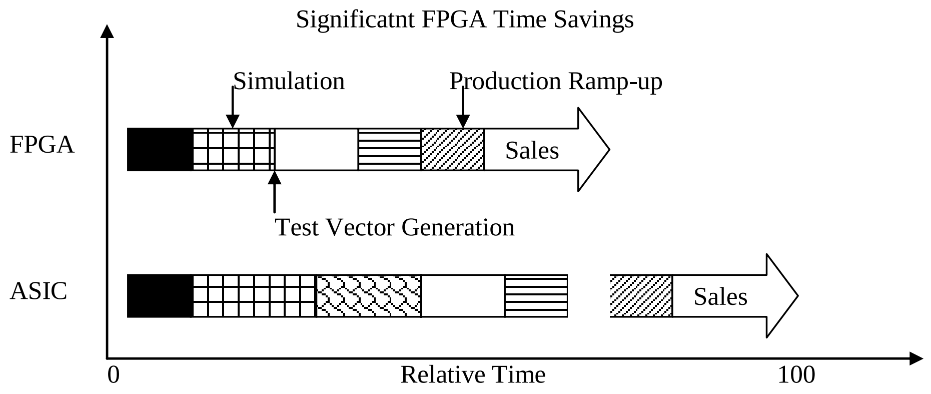


Fig. 1. Time-to-Market - FPGAs vs. ASIC [11]

Other advantages of a programmable solution include reduced development costs (little non-recurring engineering costs), the possibility of rapid prototyping and the ability to support field upgrades and remote downloads that will extend the longevity of the product in the market (time-in-market). These are depicted in Fig. 2. Hence according to Fig. 2, the sooner that hardware designers market their products, the greater their income. This is one of the major advantages of FPGAs and explains why many applications, which have historically been implemented in software and/or ASIC, are now being implemented on FPGAs [4], [5].

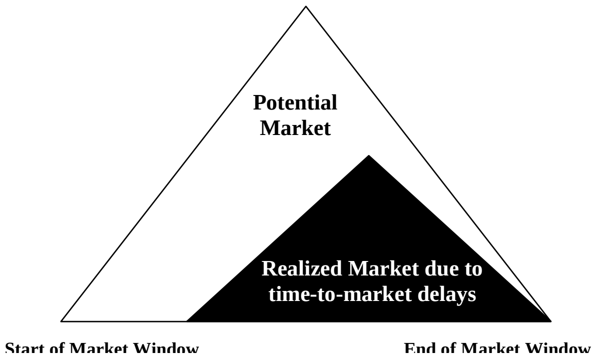


Fig. 2. Product time-in-market [11]

The implementation of computationally intensive and data transfer dominated applications, which are common in RTVPS, has previously been dominated by Digital Signal Processors (DSP) and dedicated applications specific integrated circuit (ASIC). However, developments in FPGA have made it possible to implement RTVPS applications using FPGA [4], [5]. Fig. 3 shows the implementation spectrum across computing devices. It should be noted that the different platforms in Fig. 3 are not isolated as depicted in the figure but are overlapping clouds.

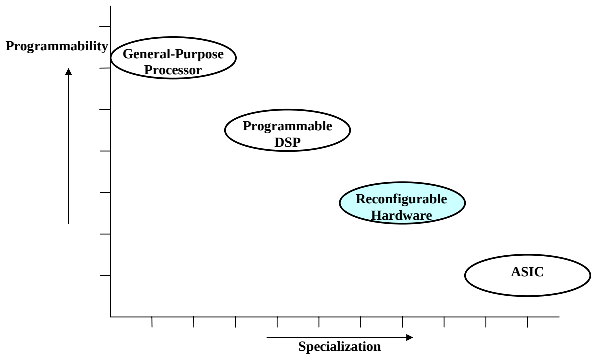


Fig. 3. Signal processing implementation spectrum [12].

FPGA has been compared to DSP and ASIC for many applications, domains [4] - [10]. [10] provides both quantitative and qualitative arguments for the performance advantages of FPGA. As shown in Fig. 3, FPGA offers specialization and performance which is close to ASIC, but it is not as easily programmable as DSP and processors. This limitation had led to the development of hardware description languages and tools to reduce the challenges of FPGA programming. The developed languages include the following, VHDL [13], Verilog [14], C/C++ [15], SystemC [16], MATLAB [17] and Java [18].

Because of its software re-configurability, hardware performance, IP reuse, time-to-market and increased number of optimised embedded resources, FPGA has been chosen as the target platform for the method developed in this thesis.

1.3 MOTIVATION FOR EFFICIENT MEMORY SYNTHESIS

If a simple RTVPS application is considered involving only spatial domain, for example a Sobel Operator for detecting edges in a video frame, then a neighbourhood (3-by-3) would be built around each pixel in the frame. Building such a neighbourhood requires the necessary pixel data to be stored. Fig. 4 depicts such a neighbourhood where, p_{ij} represents the pixel data at the *i*-th column and *j*-th row in the neighbourhood, d_{pixel} is the pixel data entering the neighbourhood as the processing window traverses all the pixels in the image and *d* is a clock delay. Line buffers are required to store these pixel data in order to create the neighbourhood. In Fig. 4b, these buffers are represented as *line buffer.* A line buffer can be thought of as a First-In-First-Out shift register (FIFO) - with predetermined constant length - that can be implemented as a circular buffer allocated to a set of memory locations.

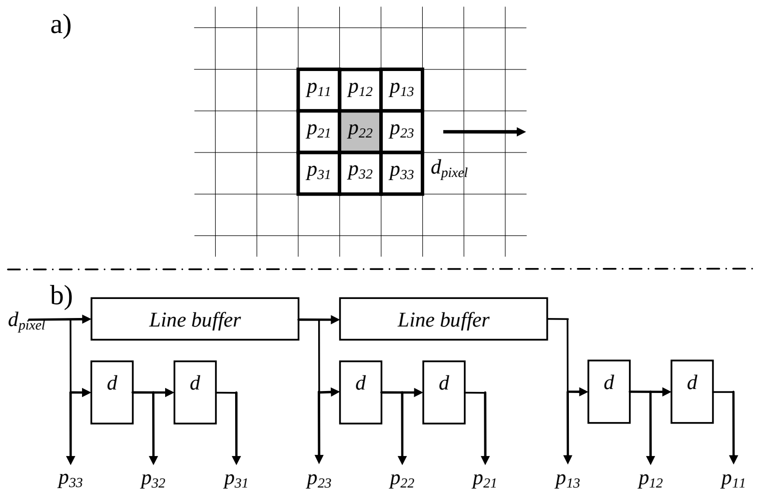


Fig. 4 Basic architecture for the implementation of line buffers in neighbourhood oriented image processing operations. Part a) shows an example of a 3-by-3 neighbourhood and b) its implementation.

Managing the line-buffers (memory objects) identified in Fig. 4 is the focus of this work. The main goal is to develop an automatic memory synthesis tool that makes the most efficient use of the addressable memory locations available in all of the instantiated FPGA on-chip memory before instantiating another.

1.4 PROBLEM DESCRIPTION

The method of allocating the line buffers identified in Fig. 4 to FPGA on-chip memory greatly affects use of the memory depending on the size of the on-chip memory. In addition, the length of the line buffer and the bit-width of each of the elements in the line buffer also affect the efficiency of the allocation. Increasing the neighbourhood dimension, in terms of the number of frames, *L*, the width of the video frame, *M* and number of line buffers, *N* as well as the number of operators in the RTVPS application leads to increasing complexities in data management. In general, managing the data required in such a neighbourhood leads to three major problems namely:

1. Data allocation problems due to pixel- and video-width (when the bit-width of each element in a line buffer and its length are not directly supported for allocation)
2. Data management problems caused by the increasing number of line buffers, *N*
3. Data management problems caused by the increasing number of RTVPS operators and number of frames, *L*

These problems will be discussed at a later stage in the thesis (Section 3.3).

1.5 MAIN CONTRIBUTIONS

The main contribution of this research is to provide solutions to the problems identified above. The following solutions are offered to the problems:

1. Memory architecture - organizing the data required by RTVPS operator.
2. Memory allocations and accessing.
3. Interfaces to frame data required by operators in temporal neighbourhood.

These solutions will be discussed at a later stage together with the results obtained by their use. Tests on the performance of the solutions and comparisons with other works are also discussed.

1.6 THESIS OUTLINE

The next section presents the developments and trends within FPGA with focus on embedded memory and DSP cores. Earlier research works relating to on-chip memory allocation and memory addressing are also presented. In Section 3, the contribution of this research work is presented. The connections between this work and others at Mid Sweden University are presented. Section 3 also presents the results of this work, the performance when increasing RTVPS complexity as well as FPGA technology and comparisons with other works. Section 4 summarises the work covered by all the papers included in this thesis. The papers, which represent original contributions of this research work, are added as appendices. Section 5 summarises and concludes the contribution of this thesis.

2 RELATED WORKS

In this section, FPGA features relevant to this research and earlier work in the field of memory optimization for FPGA are presented.

2.1 WHY FPGA?

FPGAs have been employed in implementing high-performance computations such as fuzzy logic controller, [19], complex Monte Carlos and percolation problem simulations [20]. In [4], an FPGA was used for face tracking in streaming video using an RBF neural network for real-time verification. The literature is exhaustive with regards to the use of FPGAs for network monitoring, audio/video signal processing and safety critical applications. These are the application areas previously earlier dominated by DSP. The attractions for implementing these applications on FPGAs can be traced to those features that distinguish them from other computing platforms. These features are listed as follows [21]

1. On-chip RAM blocks and distributed memories
2. Embedded processors
3. Dedicated computational units (multipliers and DSP block)
4. Programmable logic cells
5. Programmable interconnect
6. Programmable Input/Output cells
7. Logic cells can implement combinational and/or sequential logic.

Although specific implementation details vary among the vendors, the focus here is on Xilinx Spartan 3 [22] and additionally, the features common to the FPGA vendors are presented in detail. Fig. 5 shows the architectural overview of Xilinx Spartan 3. In the figure, DCM, IOB and CLB represent Digital Clock Manager, Input/Output Blocks and Configurable Logic Blocks respectively. The remaining part of this section will discuss the first four items in the above list.

2.1.1 Programmable Logic Cells

The programmable logic cell is the basic building block for implementing combinatorial and sequential logic. Logic cells are mostly categorized as either fine-grain or coarse-grain architectures, depending on the number of gates in them. Since the logic cell is the smallest unit available, it can be organized programmatically into complex units needed to perform functional requirement of the device. In an SRAM-based FPGA, a logic cell essentially consists of a lookup table (LUT) and a register to store the LUT value [23]. For example, LUTs provide the main resource for implementing logic functions. LUTs can be configured as Distributed RAM or a 16-bit shift register. The storage elements can be programmed as either a D-type flip-flop or a level-sensitive latch to provide a means to synchronize data to a clock signal. Wide-function multiplexers effectively combine LUTs in order to permit more complex logic operations. The carry chain, together with various dedicated arithmetic logic gates, supports rapid and efficient implementations of mathematical operations.

For Xilinx Spartan 3 FPGA, the logic cell is coarse-grain based and is referred to as the configurable logic block (CLB). Each CLB contains both combinatorial and sequential logics [24]. The function of a CLB is stored in a RAM-based look-up table (LUT) within the CLB. The programming on the LUT determines the use of a CLB for logical and data storage functions. Fig. 6 depicts the implementation of CLBs for Xilinx Spartan 3. Each CLB is organized into four interconnected slices. Each slice contains two logic function generators (LUTs), two storage elements, wide function multiplexers, carry logic and arithmetic gates in addition to other elements.

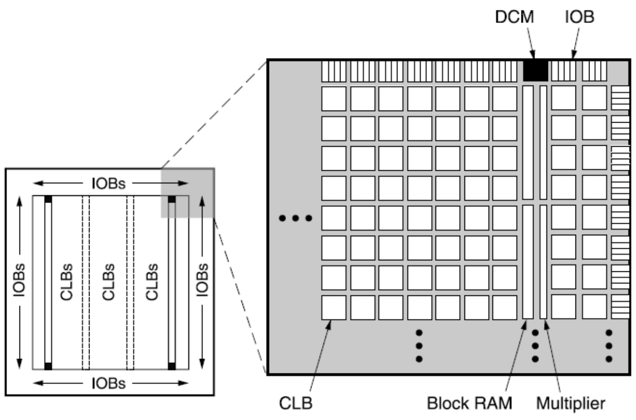


Fig. 5 Overview of Xilinx Spartan 3 [22]

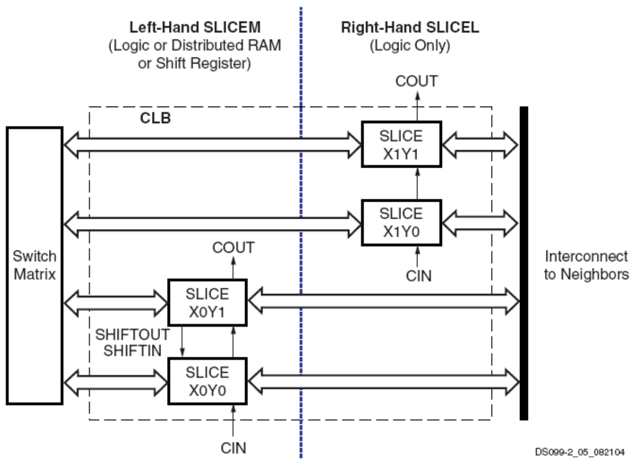


Fig. 6 Xilinx Spartan 3 CLB [22]

2.1.2 On-chip RAM

Access to data during signal processing greatly affects the performance of a system. Data fetches from external memory are subject to latency of the communicating devices and signal integrity due to cross-talk from neighbouring signals. The availability of on-chip RAM memory eliminates / reduces this latency. The random access memory (RAM) offers fast direct access to re-writeable memory locations making it appropriate for use with streaming data where buffering or caching of data is necessary. RAMs can be dynamic (DRAM) or static (SRAM). SRAMs are faster, larger (6 transistor core, Fig. 7) and require more power while DRAMs are slower, smaller (1 transistor core), requires less power, and requires that the data be periodically refreshed due to substrate leakage [23]. Although, SRAMs are bulkier than DRAMs they are used in fabricating FPGA basic logic cells and on-RAM due to speed and because they do not require data refresh. On-chip RAMs can be implemented as single-port, dual-port and multi-port [23]. Fig. 7 shows the SRAM core circuit. The memory value is stored in the loop connected pair of inverters. The two extreme transistors connect the bits line to the inverters. When the *select* (powered by the RAM enable) signal is low, the inverters reinforce their values. When the *select* is high, a read is performed by driving the bit lines value (pre-charged to *VDD*) to the value of the closer inverter. With a high value on *select*, a write is performed by loading the bit lines appropriately and using their values to drive the inverters. This is possible since the bit lines have higher capacitances than the inverters [23]. The core in Fig. 7 implements a single port RAM, but a dual-port RAM can be achieved by the additional select and bit lines connected to transistors at the opposite ends of the inverters (in parallel with those presently linking the bit lines to the inverters). In a similar manner, multi-port RAM can be built by increasing the number of bit lines and the linking transistor pairs. Obviously the additional costs for select line, two bit lines and two transistors make it expensive to implement multi port RAMs.

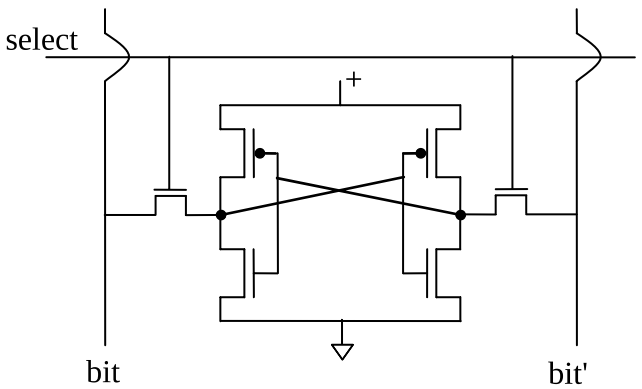


Fig. 7 SRAM core cell [23]

Typical on-chip dual- and single-port RAMs have the necessary control signals and, data and address busses for independent memory access (reading and writing) at a port [22]. In addition, a RAM block can be asynchronous or synchronous depending on whether the read and write cycles can be triggered by control and/or address transitions asynchronous to a clock or synchronous to the system clock [24]. Fig. 8 shows data path of a full implementation of true dual-port on the Xilinx Spartan 3 FPGA. In the figure, data path 1 implements write to and read from Port A, data path 2 write to and read from Port B, data path 3 implements data transfer from Port A to Port B, and data path 4 implements data transfer from Port B to Port A. Single port allocation can be achieved through data path 1 or 2 if implemented exclusively. Data paths 3 or 4 are used to implement dual port allocation. A true dual port allocation is achieved when data paths 1 and 2 are implemented together on a single Block RAM. The problem of address contention in dual- and multi-port can be solved by specifying the order of execution for example, read first or write first.

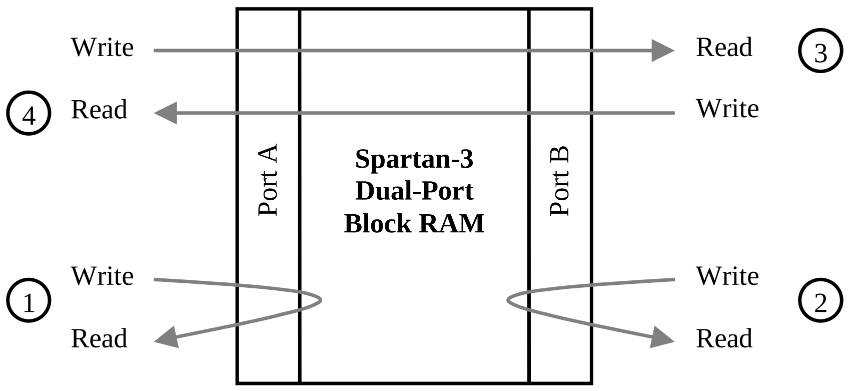


Fig. 8. RAM data path [24]

2.1.3 Embedded cores

Different FPGA vendors provide embedded core for implementing signal processing tasks that are not easily achievable in hardware or which have a reduced real-time performance. In the Stratix Architecture these are called Digital Signal Processing (DSP) Blocks [25], Embedded Multipliers in Spartan 3 [22] and Embedded Computational Units in Eclipse II [26]. Thus, DSP functions such as FIR filters, IIR filters, fast Fourier transforms, direct cosine transforms, correlators and functions such as multiply-add and multiply-accumulate can be readily implemented using these embedded cores. Multipliers are implemented as 9-by-9, 18-by-18 or 36-by-36 bits multipliers. However, they can be cascaded for higher multiplicands.

In addition to multipliers, FPGA often come with embedded processors for the implementation of control intense algorithms and divide functions that are better implemented via high level languages such as C/C++. It is also possible for a designer to implement micro-controller and processor core when the core is not embedded in the FPGA. Using the Xilinx Embedded Development Kit, a 32-bit RISC architecture-based soft processor that runs at 150 MHz to deliver up to 120 DMIPs [27] can be implemented on a Xilinx Spartan 3 FPGA. Fig. 9 shows the functional parts of the Spartan 3 MicroBlaze embedded processor [27].

IP cores optimized for different FPGAs are provided by the different FPGA vendors. In addition, glue logics for IP cores developed by third parties are provided. Hence FPGAs, which are primarily hardware platforms, provide a medium for implementing software algorithms which in turn, enable better implementation of complex function. When combined with on-chip RAM, soft cores reduce both latency, by means of their close proximity to the required data, and system costs through the elimination of external microcontrollers. Development suites for porting applications on this embedded processor or using the multipliers are usually provided by the FPGA vendors.

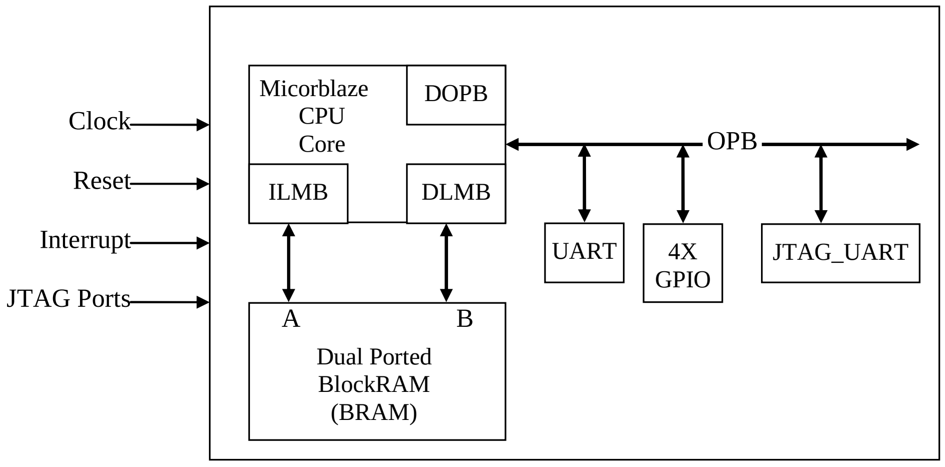


Fig. 9. Spartan 3 MicroBlaze embedded processor [27].

2.1.4 Challenges in system development on FPGA

Although FPGAs offer many opportunities, there are a number of challenges to system development particularly in the field of video processing. Some of these challenges include abstraction level, design verification, resource usage and power consumption which are discussed in the following sections:

Abstraction level

A major challenge to implementing applications on FPGAs is the programming model, which is at a very low level of logic abstraction through hardware description languages and thus requires a high level of expertise and time. Often designers familiar with software programming languages conceive algorithm executions in sequential order and thus try to program hardware in a similar manner. This leads to non-optimal implementations. They are many design tools whose aim is to translate software codes into hardware [15], [65] [66]-[67]. In this work, the abstraction level for implementing memory sub-component for an RTVPS by means of a memory allocation tool is raised.

Design verification

As FPGA capabilities and design complexities increase, verification and simulation become more complex. In order to satisfy the requirements of complex designs both Verilog and VHDL are often used to implement design sub-components, often through IP cores. Co-simulation and synthesis of the sub-components are both difficult and error prone. In addition access to simulation stimuli and responses are often complex and are provided by other tools written in other languages. This leads to coping with procedural language interfaces of the two languages within one design. Design considerations to overcome this problem are presented in [68] while [69] presented formal semantics for Verilog-VHDL co-simulation.

Resource usage

The essential logic resources on FPGAs are arithmetic and logic resources, embedded memory and logic cells. They are available optimised but in limited amount. It is necessary to have a balanced usage of these resources in an application in order to avoid shortage of one type of resource while having excess of others. In this work we have achieved efficient use of embedded memories. In the future we would find efficient use of the arithmetic resources and logic cells through resource reuse within each operator in an RTVPS. This operator-based resource reuse will minimise routing network thus increase speed performance at reduced the active power to the routing network.

Energy and power consumption

In FPGA two major sources of energy consumption include active power and leakage current. Energy consumption based on leakage current depends on the process technology [RR] and can only be address by the FPGA vendors. A study of the leakage current on Xilinx Spartan 2E, 3 and Virtex 2 shows an increasing trend. Energy consumption based active power depends on activities at the I/O blocks, switching activities on the routing network and logic cells, and memory accesses. By using embedded memory to implement line buffers, we reduce data transfer to external memories [70] and thus reduce I/O block switching activities. Power consumption can be further reduced through efficient embedded memory accesses, compact routing network and efficient logic design.

2.2 EARLIER WORKS ON ON-CHIP MEMORY SYNTHESIS

In this section, the focus is on memory allocation and addressing targeting FPGA on-chip memory. Works relating to memory estimations are not included since such works have been extensively studied and addressed while developing IMEM [58], [59]. In addition, allocation of external memories is not included in this thesis.

2.2.1 Allocation algorithms

There have been many algorithms for the optimal storage of a scalar variable. These approaches usually involve storing scalar variables with non-overlapping lifetime in the same register or grouping the scalars together to form an array which would be allocated to a Block RAM. A common feature of these approaches is the necessity for scheduling and determining memory access pattern. These efficient and well researched approaches cannot be use for allocating large array variables which result from the line buffers (identified in Fig. 4) because of the following:

1. it is assumed that the elements in the line buffers have regular cyclical read-and-write access patterns relating to the video frame width typical of FIFOs,
2. it is assumed that the size of the line buffers is large which often leads to allocating one line buffer to many Block RAMs hence grouping many line buffers into one Block RAM is not a feasible option
3. the identical access pattern of all the line buffers and the requirement of throughput of one pixel per clock cycle eliminates access scheduling

Because of the above concerns only related works which focus on allocation of array variable will be presented

Diniz et al. [28] presented a C-compiler that can extract storage requirements and considers data reuse as registers and allocates Block RAMs together with datapath- and control structures. The compiler employs data access patterns in a loop nest to minimize memory access and uses registers to exploit data queues after loop unrolling. However, exactly how the memory allocation is performed is not addressed by Diniz et al.

The MeSA algorithm [29] is based on the clustering of array variables to determine the memory configuration that will result in the minimum total memory area. The number of memory modules, the size of each module, the number of ports for each module and the cost of grouping a set of input array variables, are all computed. The number of ports is balanced for serialized memory accesses within a control and data flow graph. This algorithm cannot however be considered for implementing RTVPS on FPGA. This is because large array variables cannot be distributed among a set of memory modules.

A general approach to FPGA memory allocation and assignment was presented by Gokhale et al. [30]. This approach starts from C code, for which the presented method allocates both external and internal memories. Automatic partitioning of a single array among different memories is however not covered by this work.

Baradaran et al. [31] presented a close algorithm but the focus was on the analysis and identification of data reuse and allocation on an FPGA embedded Block RAM in the presence of a limited number of registers.

The work by Schmit and Thomas [32] performs array grouping (vertically and horizontally) and dimensional transformation (array widening and array narrowing). According to the authors, array widening is useful for read-only arrays and those accessed in loops with an unrolled number of iterations. Array narrowing slows the effective access time of the array. Vertical array grouping is similar to the global memory object architecture used in this thesis (details in Section 3) with the variation that the grouping is on memory objects required by one operation. Neither horizontal grouping nor the accompanying scheduling are considered in this work, however dual port mapping of two memory objects is implemented to achieve more efficient memory usage.

Jha and Dutt [33] presented two algorithms for memory mapping. The first, linear memory mapping, approximates target memory word-count to the largest power-of-two that is less than or equal to the source memory word-count. The second, exhaustive memory mapping, assumes that a target memory module may have larger bit-width and word counts. These approaches lead to unused memory space on the target memories particularly in on-chip memories. The work did not address multiple parallel accesses to a memory module via a different port.

2.2.2 Memory addressing

Memory accesses are a major contributor to the power consumption especially in data transfer intensive applications such as RTVPS. Activities in the memory address buffers, address decoding circuitry and off-chip drivers of the address bus, are reflected in the power dissipations. There have been many works aimed at lowering the impact of memory access on power consumption. The majority, however, are tailored towards their memory architecture for efficiency purposes. The general approach is to use a counter to evaluate the value of the address bus of on-chip memories. These approaches are reviewed in this section. In addition, latency in memory accesses affects the system performance. Hence effective optimization can be achieved through efficient memory architecture and addressing procedure.

In [46] it was noted that most behavioural synthesis tools do not support FPGA vendor specific external memory interfacing. The authors proposed an approach which includes target architecture oriented timing requirements for accessing memory and application specific memory access pattern information.

In [45] a technique exploiting regularity and spatial locality in memory access pattern in order to achieve low power mapping of arrays in behavioural specifications to physical memory was presented.

The work presented by Doggett et al. is optimal in the case of large numbers of memory banks being used, as is typical in volume rendering in medical applications [34]. The work presented a cubic addressing scheme and used FIFO buffers to minimize the pipeline stalling effect of cache misses

The address generation scheme by Grant el al. is an efficient option for accessing data with addresses within the power range of two [35]. The scheme uses a register and optionally an offset, to specify memory read/write addresses.

The memory exploration algorithm in [36] implements memory allocation and array-mapping to RAMs through tight links to the scheduling effect and non-uniform access speeds among the RAM ports to achieve near optimal memory area and efficient energy requirement. The algorithm is, however, complex and the execution time may slow down hardware design. Moreover the exploration targets SRAM and DRAM as opposed to the on-chip FPGA Block RAMs, which are the focus of this thesis.

The address generation technique in [37] is based on address bit inversion to yield effective access time to memory at the cost of up to an extra 17.4% of used memory.

In [38] and [47], various high-level optimizations were explored in order to reduce addressing overhead. Many efficient, often heuristics based, memory optimization algorithms have been developed similar to those in [39], [40], however, most of these are tailored to be efficient on DSP.

2.2.3 Response to related works

None of the allocation and addressing methods in Sections 2.2.1 and 2.2.2 were considered to be appropriate for managing memory requirements of RTVPS while using the limited embedded FPGA memories. This is because these algorithms do not fully utilize the configurable data port widths supported by the FPGA and the true dual port capabilities of the Block RAMs. In addition, we consider the data memory architecture in [60] is considered to be more efficient for RTVPS data management hence allocation and addressing methods based on this architecture would be efficient. This is the motivation behind the development and implementing a new allocation algorithm designed to maximize the memory usage while minimizing the read/write accesses. In addition, two approaches to access the allocated memories have been developed.

The work in this thesis achieves near optimal results in terms of the number of allocated memories, the amount of unused memories and the access speed by fully utilizing the combination of FPGA embedded memory capabilities and RTVPS regular data pattern.

3 MEMORY SYNTHESIS FOR REAL-TIME VIDEO PROCESSING SYSTEMS

This section presents the scientific work covered in Papers I to V. Memory synthesis generally refers to data storage and management. The process involves memory architecture, allocation and addressing. These terms are discussed in the following subsections. Managing the line-buffers (memory objects) identified in Fig. 4 is the focus of this work. The main goal is to develop an automatic memory synthesis tool that makes the most efficient use of all the addressable memory locations available in all the instantiated FPGA on-chip memory before instantiating another. The work in this thesis is part of the Mid Sweden University interface and memory modelling (IMEM) design tool [58], [59]. IMEM is based on the philosophy that the memory requirement of an RTVPS can be modelled and synthesised independently to the synthesis of the RTVPS filters. Thus this thesis presents the synthesis of the on-chip memory requirements specified by IMEM.

3.1 IMEM SYNTHESIS WORKFLOW

IMEM, an extension of SystemC, is a set of class libraries suitable for capturing, modelling and simulating RTVPS without implementation details. The IMEM synthesis workflow depicted in Fig. 10 demonstrates how research dealing with modelling and high level synthesis fits into an RTVPS implementation trajectory.

This workflow is defined at six different levels along the left-hand axis. The video-processing algorithm is developed and simulated using IMEM at level 1. This executable model can then be verified through functional simulation. Data dependency information, frame sizes, composition of the 3-dimensional neighbourhoods and colour space models are exported into an interface and memory model at level 2. This information is the input for the memory synthesis process at level 3. It is here that memory estimation, memory hierarchy optimization, memory allocation and address generation are performed. The work in this thesis is at this IMEM level.

Additionally, at level 2 the behavioural C++ description is separated from the memory model. At level 3, the SystemC functional description together with the interface template generated from the memory model is synthesized using a SystemC based commercial high-level synthesis tool (for example Agility from Celoxica). The VHDL codes from both the functional part and the optimized interface and memory model are integrated at level 4 and synthesized at level 5.

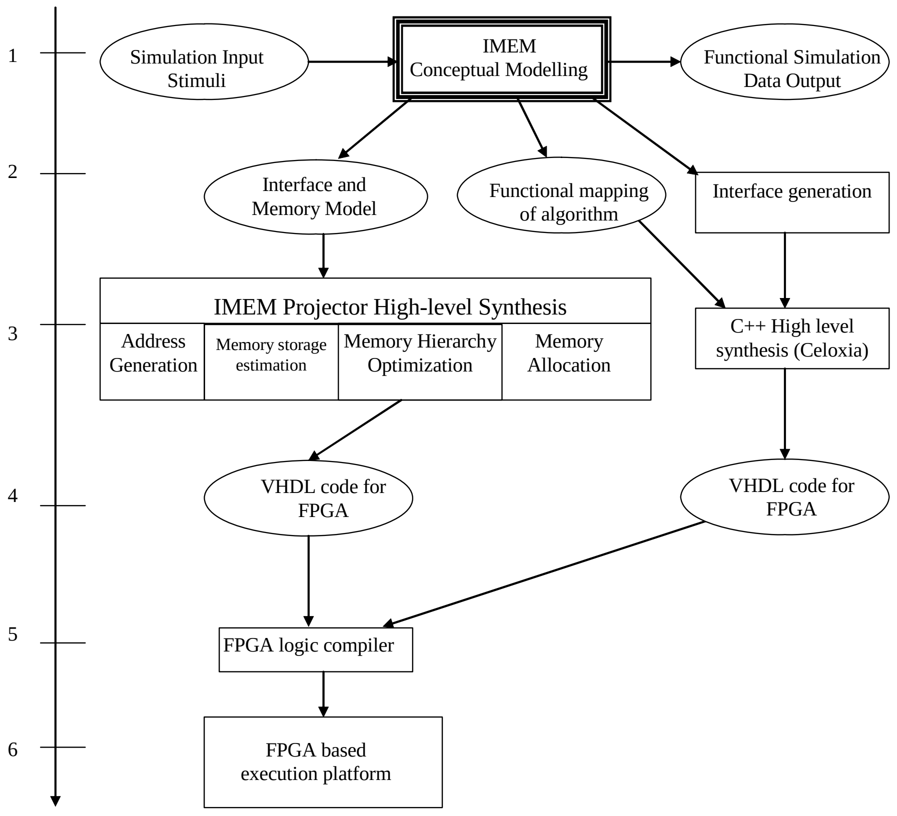

1
2
3
4
5
6
Simulation Input Stimuli
IMEM Conceptual Modelling
Functional Simulation Data Output
Interface and Memory Model
Functional mapping of algorithm
Interface generation
IMEM Projector High-level Synthesis
Address Generation
Memory storage estimation
Memory Hierarchy Optimization
Memory Allocation
C++ High level synthesis (Celoxia)
VHDL code for FPGA
VHDL code for FPGA
FPGA logic compiler
FPGA based execution platform

Fig. 10. System synthesis workflow

3.2 Memory Architecture

In a streamed hardware implementation only one operator can use the memory objects (Fig. 4) and all the memory objects are used simultaneously in the RTVPS. It is assumed that memory objects can be grouped together to form global memory objects at the operator level. This grouping can be achieved through:

$$W_{Ri} = n_{lines} \times w_p \tag{1}$$

where W_{Ri} is the width of the global memory object at the operator, n_{lines} is the number of required line buffers for an operator and w_p is the bit width representing a pixel. The length of the global memory object is equal to those of the memory objects that formed it, i.e. the image width [60]. This architecture is preferable to that of direct mapping of memory objects to memory location. This preference is because global memory objects require a minimal number of required memory entities in comparison to direct mapping architecture. Consequently, the number of memory accesses for an RTVPS operation is minimal for global memory object.

To illustrate the formation of the global memory objects, consider an RTVPS operator that requires a neighbourhood of a 5x5 window with 12-bit gray scale and a 640 by 480 frame size as the input video stream. This would result in the creation of four memory objects each of length *L* (=640) and width 12. The memory objects would be combined to create a GMO R_i of width 48. Fig. 4 depicts this illustration where *op_id* represents the operator requiring the GMO.

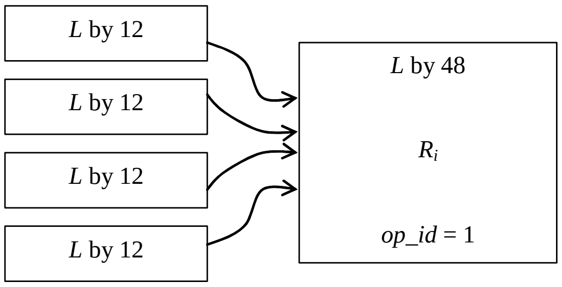


Fig. 11 Global Memory Object formation

3.3 Memory Allocation

Single port memory configuration or a dual-port in which one port is used for writing and another for read usually leads to unused memory areas. Fig. 12 depicts an example of such an allocation. As shown, after the allocation of memory objects 1 and 2 to memory areas A1 and A2 on Block RAMs 1 and 2 respectively, the remaining memory areas B1 and B2 remain unused and subsequent memory objects will be allocated to other Block RAMs. Hence these types of memory allocation approaches can be very inefficient unless the allocated data is exactly the size of the memory module which is, however, very rarely the case.

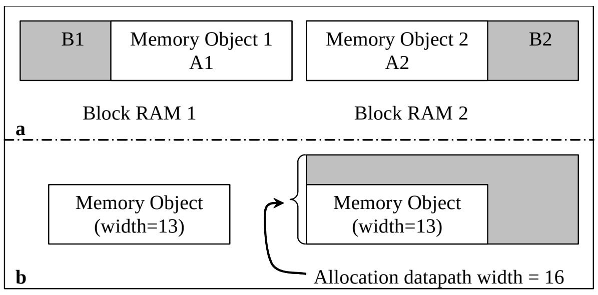


Fig. 12. Traditional memory allocation.

Although, FPGA allows the allocation of memory objects of any datapath widths, this allocation is, however, left to the designer. Traditionally, a higher datapath width is used when the width of the memory object is not a member of the datapath widths specified by the FPGA. Fig. 12b shows an attempt to allocate an object of width 13 on Xilinx Spartan 3. Since a datapath width of 13 is not specified by the FPGA and 16 is the next datapath width that is a member of Xilinx Spartan 3 datapath widths, allocation of the memory object is made using a datapath of 16. This will result in 3*L* bits being wasted, where *L* is the length of the memory object. An alternative is to partition the memory objects into using the supported widths.

These two stated sources of inefficient allocation are the reasons for researching both allocation architecture and an algorithm based on the architecture that makes optimal use of memories. To achieve efficient allocation, the advantage of parallel accesses to Block RAMs through two independent ports is exploited.

3.3.1 Allocation algorithm

As presented by O'Nils et al. dual-port configuration of FPGA Block RAMs and global memory object allocation for RTVPS provides an efficient use of Block RAMs [60]. An algorithm taking advantage of such efficient memory allocation techniques and the possibility of parallel accesses to Block RAMs through two independent ports will be presented. Fig. 13 shows attempts at finding an optimal use of the remaining memory resources identified in an FPGA Block RAM. If the remaining memory space is a single rectangular block as shown in Fig. 13A, allocation is made to it through the second data port. If the remaining memory space is not a single rectangular block, it is divided into two rectangular blocks B and C as shown in Fig. 13B. Allocation can be made to B or C through the second data port. Because Block RAMs currently support a maximum of two data ports, only one of block B or C can be allocated depending on its size and the sizes of the memory objects awaiting allocation while the other block will never be used. As a result, the developed algorithm seeks the allocation for which the unused memory space is minimal by ensuring that, after allocation through port A, the remaining memory space forms a rectangular block, and by finding the memory object that uses as much of this block as possible. Hence, one of the indicators used in measuring the efficiency of the algorithm is the size of the unused memory resources.

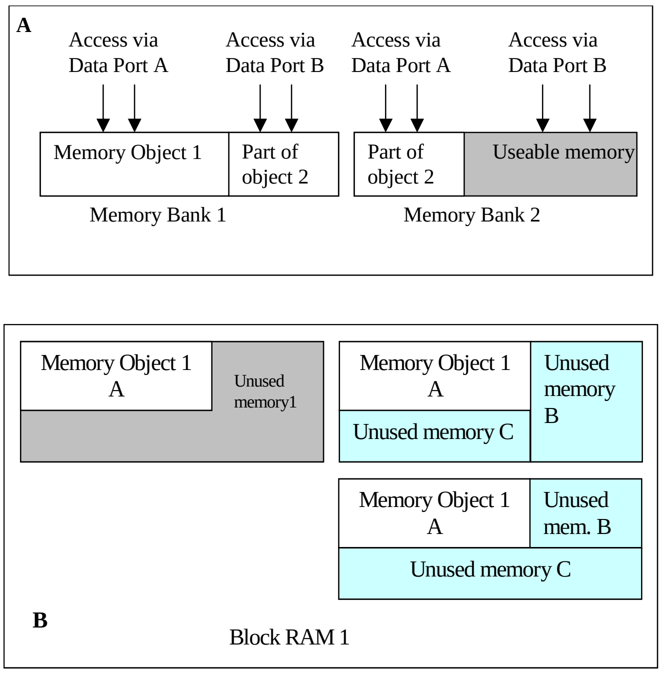


Fig. 13. Proposed memory allocation.

3.3.2 Definitions

To find the optimal use of the Block RAM, the algorithm must observe some definitions and constraints. These are listed as follows:

(i) M is the set of all available Block RAM M_k and K is the number of Block RAMs.

$$M = \{M_k | k = 1,2,..., K\} \qquad (2)$$

(ii) S_{Mk} is the size of the Block RAM M_k and is specified by the FPGA. For example, in Xilinx Spartan 2E FPGA S_{Mk} is 4096 bits [61]. The memory objects allocated to the Block RAM determine the length L_{Mk} and width W_{Mk} of M_k.

(iii) W_c is the set of all possible datapath widths W_n for Block RAMs on the FPGA. For example, 1, 2, 4, 8, and 16 are allowed on Xilinx Spartan 2E FPGA [61].

$$W_c = \{W_n | n = 1,2,..., N\} \qquad (3)$$

(iv) R is the set of all memory objects R_i to be allocated and I is the number of memory objects.

$$R = \{R_i | i = 1,2,..., I\} \qquad (4)$$

(v) The size S_{Ri} of memory object R_i is defined as product of the length L_{Ri} and data width W_{Ri} of the memory object R_i.

$$S_{Ri} = L_{Ri} \times W_{Ri} \qquad (5)$$

(vi) Each global memory object is characterised by a quadruple of *op_id*$_{Ri}$, L_{Ri}, W_{Ri} and x_{Ri}.

$$R_i(\text{op\_id}_{Ri}, L_{Ri}, W_{Ri}, x_{Ri}) \qquad (6)$$

where *op_id*$_{Ri}$ is an identifier for the operator where the memory objects making up the global memory object R_i are defined and x_{Ri} is the segment in which a memory object is located on the global memory object after partitioning into units of allowable data widths in W_c.

(vii) If W_{Ri} is not a member of W_c, R_i is partitioned into r_j partitions such that the width, w_R, of each partition is a member of W_c where j = 1, 2, … J and J is the number of partitions in object R_i.

$$R_i = \{ r_j(op\_id_{Ri}, L_{Ri}, w_{Ri}, x_{Ri}) | w_{R_i} \in W_c, j = 1,2,\ldots,J \} \qquad (7)$$

(viii) Memory object R_i may be allocated to as many Block RAMs as required.

$$\sum_{k=1}^{K} L_{i,k} \times W_{Ri} \leq S_{Ri} \qquad (8)$$

where $L_{i,k}$ is the part of length L_{Ri} allocated at M_k.

(ix) Block RAM only supports a maximum of two data ports.

(x) $D_{i,k}$ is the decision to allocate some or all of the memory objects R_i at M_k.

$$\sum_{i=1}^{I} D_{i,k} \leq 2 \qquad (9)$$

(xi) For all R_i in R and M_k in M that form part of the $D_{i,k}$, the sum of the allocations may not be more than the size of the Block RAM.

$$\sum_{i=i}^{I} L_{i,k} \times W_{Ri} \leq S_{Mk} \qquad (10)$$

(xii) For all $D_{i,k}$, in the set of allocation decisions, AD, the unused memory space in M_k is defined as UM_k.

$$\forall D_{i,k} \in AD, UM_k = S_{Mk} - \sum_{i=i}^{I} L_{i,k} \times W_{Ri} \qquad (11)$$

(xiii) The objective function of the algorithm is to minimize the sum of all UM_k.

$$\forall D_{i,k} \in AD, \min\left(\sum_{i,k} UM_k \right) \qquad (12)$$

To illustrate definitions (vi) and (vii), if the global memory object R_i of width 48 in Fig. 11 were to be allocated on Xilinx Spartan 2E, R_i would then be partitioned into three r_j each with a width of 16 since it is not possible to have a datapath width of 48 on a Spartan 2E. x_{Ri} will be 1, 2 and 3 for the first, second and third partitions indicating least, middle and most significant partitions on R_i. Fig. 14 depicts this illustration.

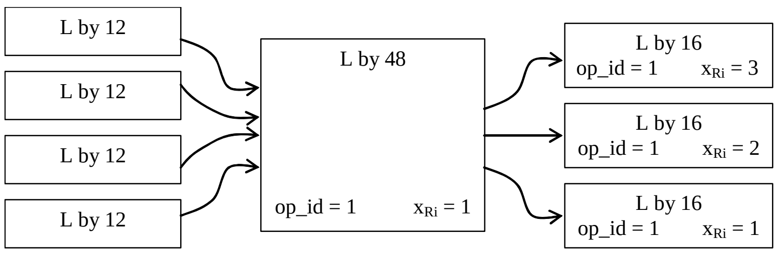


Fig. 14. Partitioning global memory object.

For every Block RAM available on the FPGA, attempts are made to allocate a global memory object to it. The amount of unused memory space UM_k is estimated. If UM_k is zero, the allocation decision is stored and the iteration continues to the next memory object or Block RAM. Other possibilities are then considered such that UM_k is minimal. The final decision is based on the allocations offering the least amount of the sum of unused memory space on all Block RAM. Fig. 15 shows the allocation algorithm in relation to the definitions and constraints listed above before making the final decision. In the figure, any R_i is an allocation candidate to any M_k. Since M_k supports only two ports and in line with definition (x), only two R_is that minimize UM_k are selected such that our objective function, definition (xiii) is achieved after all R_is are allocated. According to definitions (vii) and (viii), a global memory object may be partitioned into many smaller units exploiting FPGA parallel access to Block RAMs which enables the reconstruction of the object in order to achieve a throughput of one pixel per clock cycle.

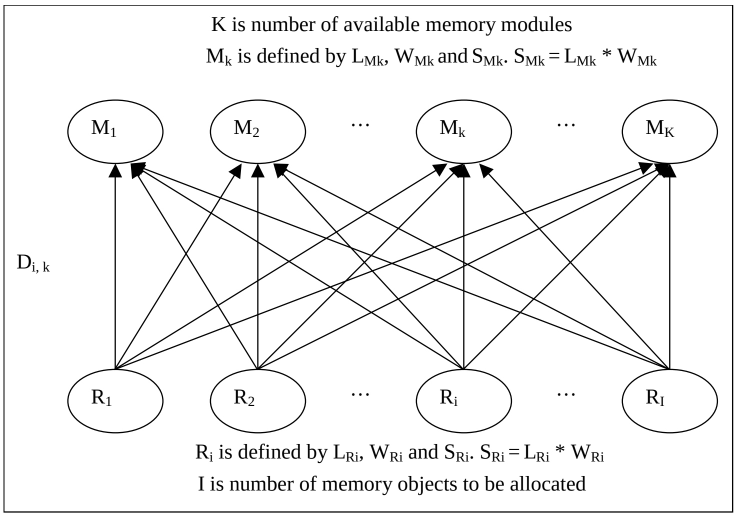

K is number of available memory modules
M_k is defined by L_{Mk}, W_{Mk} and S_{Mk}. $S_{Mk} = L_{Mk} * W_{Mk}$
M_1
M_2
…
M_k
…
M_K
$D_{i, k}$
R_1
R_2
…
R_i
…
R_I
R_i is defined by L_{Ri}, W_{Ri} and S_{Ri}. $S_{Ri} = L_{Ri} * W_{Ri}$
I is number of memory objects to be allocated

Fig. 15. Allocation model.

3.3.3 Proposed algorithm

The proposed allocation algorithm is presented in Fig. 16. A more detailed form of the algorithm in the form of pseudo-code is presented in Figure A.1 in Appendix A. In step 1, the algorithm creates global memory objects according to Eq. (1). In step 2, the algorithm ensures that the global memory objects conform to the allowable port width configuration according to definition vii. This step is captured in a procedure, *configure_global_memory_objects(R)*, presented at the lower part of Figure A. Steps 3, 6, 7 and 8 ensure that the algorithm iterates through all the memory objects starting with the first. In step 4 the global memory objects are allocated to the Block RAMs according to definitions (viii) to (xi) while optimal use of unallocated memory space in the Block RAM through the second port is implemented in step 5, which is also in accordance with definitions (viii) to (xi). Optimal allocation is that for which the unused memory space is a minimum, preferably zero using either one or two ports in the Block RAM.

The Proposed Allocation Algorithm

```
Algorithm:  Memory Allocation(R, M)
Parameters: R[$R_1$ … $R_I$] set of I memory objects;
            M[$M_1$ … $M_K$] set of K Block RAMs;
Return:     $M_A$[$M_{A1}$ … $M_{AK}$] set of K Allocated Block RAMs;

1. Create global memory objects (GMO)
2. Configure GMOs
3. Starting with the first GMO and the first Block RAM
4. Allocate GMO to Block RAM via port A.
5. If Block RAM is not fully used find maximum use of
   remaining memory via port B using another GMO.
6. Select the next GMO when the current has been fully
   allocated.
7. Select the next Block RAM when all the memory space
   has been optimally used.
8. Return the set of allocated Block RAMs after
   allocating all GMOs.
```

Fig. 16. The proposed allocation algorithm.

3.3.4 Complexity analysis

In estimating the complexity of the algorithm, the number of available Block RAMs, *K*, and the number of memory objects, *I*, after partitioning with respect to their width, play major roles. Since the algorithm makes one iteration through the sets of Block RAMs and two iterations through the set of memory objects as shown in steps 3, 4 and 13 in Fig. 16 (see also Figure A in the Appendix), the allocation algorithm *AA* is a function of *K* and *I* and its complexity can be expressed as

$$AA(K, I) = \mathrm{O}(K \cdot I^2) \tag{13}$$

The algorithm is thus, at worst, of the third order of the larger of *K* and *I*. Implementation costs depend on the representations of the properties of the Block RAMs, memory objects and allocation objects, and the arithmetic and logic operations defined for them.

3.4 MEMORY ACCESSING

The allocation software ensures that each entry of a Block RAM data object stores information concerning the width and length of the GMO segment allocated to it, the port used for allocation and the hierarchy of its segment in the GMO. In addition, each partition stores information about the Block RAM to which it is allocated, the port of allocation and its start address on the Block RAM, the GMO and segment to which it belongs.

The advantage of sequential accesses to memory for RTVPS applications can lead to improved memory performance by using pointers whose values increase whenever there are valid pixel values. Using the GMO architecture further reduces the number of such pointers to one for each RTVPS operator. The pointers may be implemented by using a single register for each GMO, further referred to as the base pointer, or by using a register for each partition in a GMO, further referred to as the distributed pointers.

To this end, the results from the memory allocation stage are imported into the address generation module. From these allocation results GMOs are reconstructed, and address spans for each partition in a Block RAM are generated. The start and end addresses for each partition are calculated. Offsets are considered where dual ports are used for the allocation on Block RAMs for different partitions in order to avoid memory overlap. The generated addresses are used to determine the location of each GMO element. The descriptions for accessing the GMO elements using two approaches, namely the base pointer approach and the distributed pointer approach are presented as follows. These two approaches are depicted in Fig. 17 while details of their implementations are presented in the following subsections.

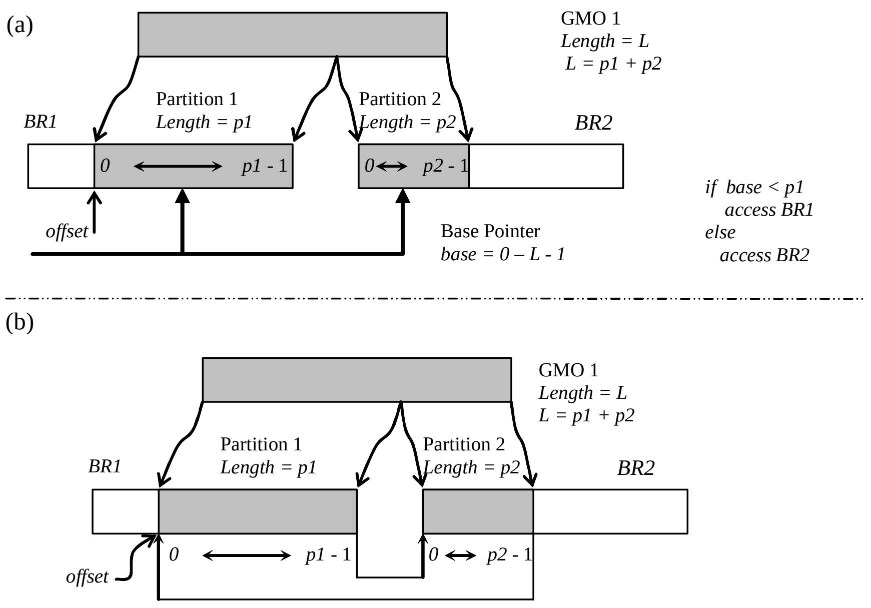


Fig. 17 Two memory accessing approaches

3.4.1 Base Pointer Approach

In this approach, a single pointer is used to track the location of the element to be accessed in the GMO. The pointer starts at zero and increases to one less than the length of the GMO and then resets to zero. Since the memory accesses are clocked, the value of the pointer increases with clocked access to the Block RAM when there are valid data. Address spans for each partition of the GMO are used to determine the relevant Block RAM relating to the element accessed, depending on the value of the pointer. Hence, only the relevant Block RAMs are enabled while the other related Block RAMs are disabled. Fig. 17a depicts this approach for a simplified case in which a GMO consists of a single segment with two partitions.

In the figure, partitions *p1* and *p2* are allocated to Block RAMs *BR1* and *BR2*. From Fig. 17a, when the value of the counter *base* is within the span of *p1*, the appropriate port on *BR1* is enabled and accessed while the relevant port on *BR2* is disabled. The reverse is the case when *base* is no longer within the span of *p1*, i.e. within the span of *p2*. This simple example could be extended to cases in which more than one segment makes up a GMO and each segment has more than 2 partitions. A formal description of this approach is shown in Fig. 18a.

Fig. 18b depicts the base pointer implementation of the GMO shown in Fig. 14. In the figure, *BR1_EN_A*, *BR2_EN_A* and *BR2_EN_B* represent the enable signals on port A of *BR1,* port A of *BR2*, and port B of *BR2* respectively. Likewise, *BR1_A_Adr*, *BR2_A_Adr* and *BR2_B_Adr* are the address signals on port A of *BR1,* port A of *BR2*, and port B of *BR2* respectively. A Block RAM is enabled or disabled by assigning ‘1’ or ‘0’ to its enable signal.

(a)

For each GMO:

- create Address Table from segments and partitions that make up the GMO to determine when to enable Block RAMs among related partitions
- create an incrementable pointer of length $\lceil \log_2(L) \rceil$ which increases when there are valid pixel values
- using Address Table and pointer value enable appropriate Block RAMs and set the values of address signals.

(b)

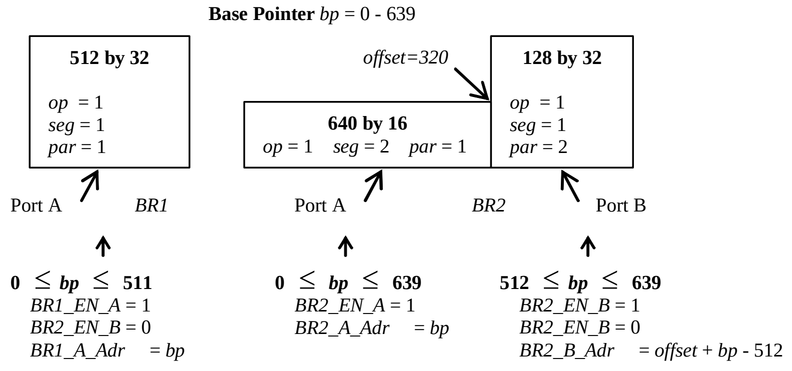


Fig. 18. Base Pointer Approach.

3.4.2 Distributed Pointer Approach

In this approach, each partition is handled separately, starting with the first partition in a segment. Local pointers equal in length to that of each partition are created. As long as the enable signal of Block RAM for a partition is high, memory access is initiated at its first position using its pointer and continues incrementally, if valid data are available until its full length is achieved. During this period, the partition ensures its enable signal is re-asserted while the enable signals of the neighbouring partitions of the same segment are kept low. Controls are transferred to the next partition of a similar segment when the upper limit of the partition is reached. If however, the partition is the last in the segment, controls are transferred to the first partition. Since the address buses of partitions on Block RAMs provide appropriate bit vectors to cover their entire lengths, they are used as the local pointer. In this approach, the enable signals of all the first partitions are set to high at start-up to ensure that memory accesses start with the first partitions. Fig. 17b depicts this approach. A simplified case of a GMO consisting of a single segment with two partitions *p1* and *p2* allocated on Block RAMs *BR1* and *BR2* respectively is considered in Fig. 17b. Fig. 19a and Fig. 19b show formal descriptions and implementations of the GMO depicted in Fig. 14 using this approach. Signals in Fig. 19b have similar meanings to those in Fig. 18b. Since the 640-by-16 partition is the only one in its segment, it is always enabled and the address is reset to 0 when it reaches its upper limit.

(a)

For each segment in each GMO:

- create Address Table for each partition in the segment
- create an incrementable pointer of length $\lceil \log_2(p) \rceil$ which increases when there are valid pixel data for partitions in the segment
- start memory access with the first partition with start address of 0
- enable Block RAM of currently active partition and disable Block RAMs of related partitions while pointer is less than partition's length
- if pointer of active equals partition's length less one, reset it to 0, disable it and enable next (or first partition if this is the last partition).

(b)

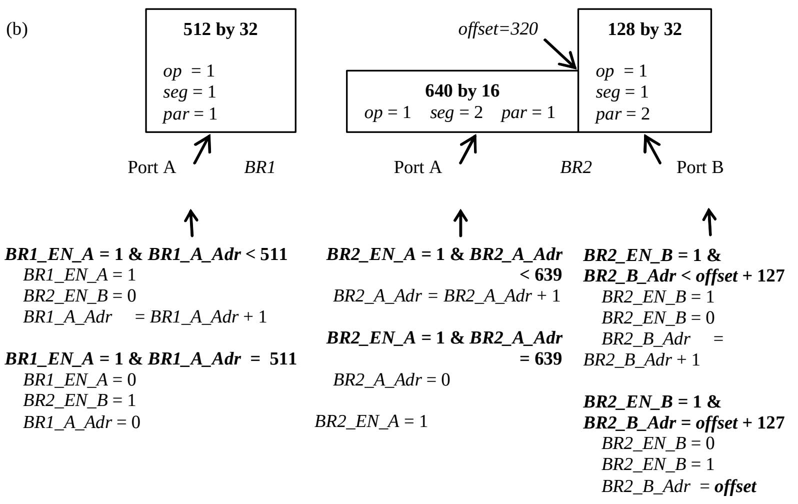


Fig. 19. Distributed Approach.

3.5 RESULTS

In this section the results obtained after the implementing the algorithm and the analysis of its performance are presented as follows. Section 3.5.1 presents the performance of the algorithm under real-time video processing design. Section 3.5.3 presents its performance, under two test scenarios modelled upon one of the real-time design cases. Performance of the memory synthesis with varying memory requirements is presented in Section 3.5.4. Section 3.5.5 presents the performance analysis for eleven video processing systems published by other researchers. Section 3.5.6 compares the performance of the two memory addressing schemes presented in Section 3.4.

3.5.1 Real-time video processing design cases

The algorithm has been implemented in C++ using the object-oriented approach. The implementation was simulated using the memory requirements of real-time video processing design cases [60]. The first design case was a spatio-temporal median filter with a neighbourhood of seven frames and two line buffers. Two instances of this design case were considered. The first, (1-1), being a VGA frame with 24-bit RGB pixels and a 640 frame length while the second, (1-2), was a PAL frame with an 8-bit gray scale pixel and a 708 frame length. The second design case was a machine vision system with a median filter, segmentation and three 1-bit morphological operations. For this design case two instances were also considered. The first, (2-1), being an 8-bit gray scale with VGA resolution as the input video stream while the second, (2-2), had a 12-bit gray scale with 1.3 MPixel resolution as its input video stream. Table 1 shows the summary of the memory requirements for the design cases considered. In the table column 2 shows the number of video processing filters in the design case while column 3 shows the number of line buffers required by each filter. For design cases 1-1 and 1-2 seven 3x3 filters were used, each requiring two line buffers while for design cases 2-1 and 2-2, one 5x5 median filter, one segmentation operation and three 17-by-17 morphological filters requiring four, one and sixteen line buffers respectively were used. Columns four, five and six represent pixel resolution, length of the line buffer and memory requirement for each filter respectively.

Table 1. Memory requirement of considered design cases.

Design Case	#	Rows	Width	Length	Size (Kbit)
Case 1-1	7	2	24	640	210
Case 1-2	7	2	8	708	77.4
Case 2-1	1	4	8	640	20.0
	1	1	19	256	4.75
	3	16	1	640	30.0
Case 2-2	1	4	12	1300	60.94
	1	1	21	4096	84.0
	3	16	1	1300	60.94

3.5.2 Allocation Results

Table 2 and Table 3 show the results obtained using the implementation of the algorithm for allocating the design cases considered on Xilinx Spartan 2E and Spartan 3 FPGA respectively.

Table 2. Allocation result of the algorithm on Spartan 2E.

Design Case	minima	Block RAM	% minima
Case 1-1	53	53	100
Case 1-2	20	20	100
Case 2-1	14	14	100
Case 2-2	52	52	100

Table 3. Allocation result of the algorithm on Spartan 3.

Design Cases	Minima	Block RAM	% minima
Case 1-1	14	14	100
Case 1-2	5	6	120
Case 2-1	4	5	125
Case 2-2	13	13	100

In the tables the theoretical minima Block RAM required for allocation were estimated from equation (14) [60].

$$minimal = \left\lceil \frac{Size}{\mathrm{size}(BRAM)} \right\rceil \tag{14}$$

where *Size* is the number of bits required by the design case, given in column 6 of Table 1, and the size of BRAM is the numbers of bits in one block RAM, 4 Kbit for a Xilinx Spartan 2E and 16kbit (without parity) for Spartan 3 [61], [24]. Table 2 shows that the algorithm requires no more than the minimum value for the allocation of each of the design cases on Spartan 2E. Hence, it is in total agreement with the minimum requirements for Spartan 2E. On Spartan 3, allocation requirements were equal to the minimum values except for two of the design cases. The minimum value is calculated for allocation on a Block RAM with an infinite number of ports. The minimum value, however, only indicates the effectiveness of the allocation but not its feasibility, since it is not possible to have Block RAMs with an infinite number of ports. The implementation for Spartan 3 did not consider parity. The parity feature on Xilinx Spartan 3 FPGA increases the available Block RAM size by providing an additional bit for every 8 bits [24]. When the parity bit is taken into consideration it makes it possible to have width configurations that are multiples of 9-bit on the Block RAM. In this manner, 18Kbits of Block RAM size can be achieved instead of 16Kbits. This parity feature was not considered since it is only specific to only some of the Xilinx FPGA families and not all FPGAs have this feature. From Table 3, the non-minimum result of the algorithm in design cases 1-2 and 2-1 is because, if a design case has many operators in relation to the total storage requirement and/or the size of each Block RAM, the number of ports on each Block RAM will limit the allocation.

Fig. 20 shows the mapping of the memory objects to the Block RAMs for the design Case 2-1 on Xilinx Spartan 2E. The identifiers of the global memory objects and the Blocks RAMs are shown. In addition, the figure shows that memory objects were allocated to as many Block RAMs as required. This is a case of dynamic partitioning with respect to the length. In the figure, each block is annotated by "*W*x*L*" and "op_id: *y*" where *W*, *L* and *y* represent the width, memory depth and operator id of the allocated partition respectively. BRAMs 7 and 8 in the figure exploit the independence of the data path width and memory depth for the two ports on a dual-ported RAM. In BRAM 7, Port A is allocated with a partition which has a data path width of 2 and a depth of 256 while Port B is allocated with a partition with a data path width of 16 and a depth of 224.

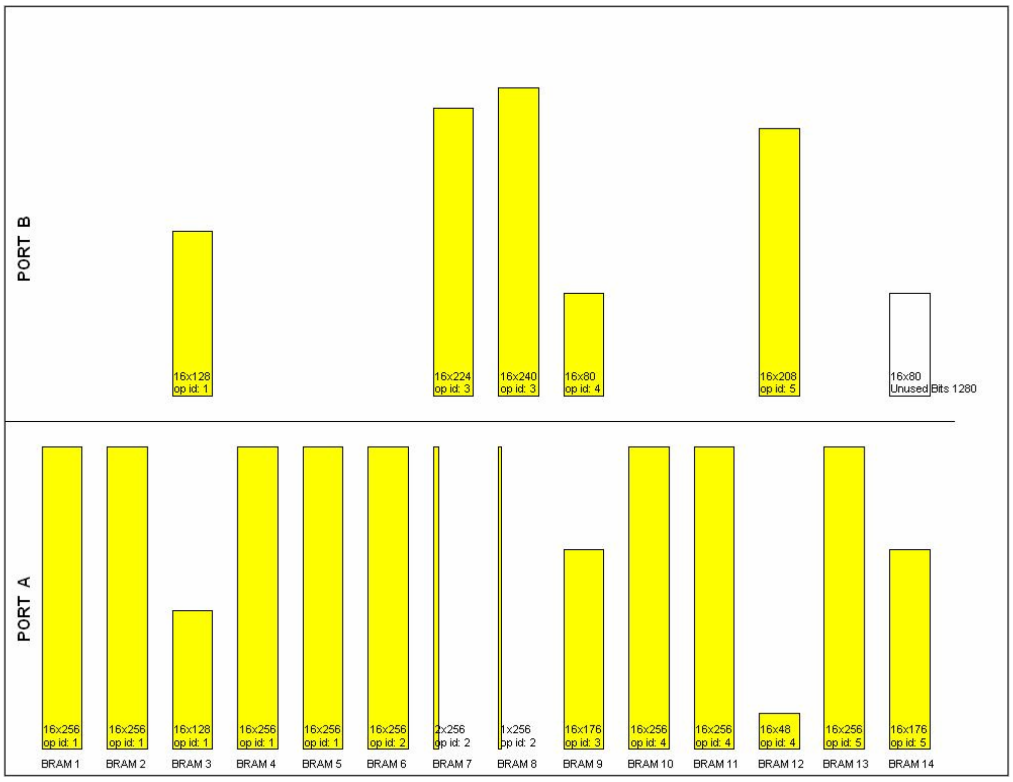


Fig. 20. Memory allocation of Case 2-1 on Xilinx Spartan 2E FPGA.

In Fig. 20 memory object 1 of width 32 bits and length 640 was firstly partitioned width-wise into two partitions each of width 16 bits and length 640. Then the first partition was allocated to Block RAMs 1, 2 and 3, by partitioning it length-wise and allocating partitions of lengths 256, 256 and 128 respectively, completely filling the Block RAMs 1 and 2 using only one port. The second partition of memory object 1 was also partitioned length-wise and allocated to Block RAMs 3, 4 and 5. This width-wise and length-wise partitioning of the memory object makes it possible to allocate a memory object to many Block RAMs and to configure the memory object with widths feasible in the FPGA. In the figure, the lower and upper allocations were through ports A and B respectively. The figure also indicates the width and length of the memory objects allocated at each Block RAM. In addition, unused memory space is specified on Block RAM 14 where it occurred. This memory space can be used through the second port.

3.5.3 Performance analysis with varying length and width

To test the performance of the algorithm, the memory requirements for allocation were varied under two scenarios such that they are similar to design case 1-1. The two test scenarios are presented as follows.

3.5.3.1 First test scenario

In this test scenario, four frame lengths *L* (320, 640, 1280 and 2560) were used while the widths *W* were determined by the memory requirement, which was allowed to vary from 100kbit to 2000kbit. This test scenario was simulated for XILINX Spartan 2E and 3 FPGA. The minimum Block RAM allocation was plotted along with the estimated Block RAMs for the four values *L*. On Spartan 2E, the minima were equal to those estimated for all values of *L* while on Spartan 3, the minima differed from the values obtained for *L* = 320 and the estimated values obtained for other values of *L* equalled the minima for most of the memory requirements. Fig. 21 shows the performance of the algorithm for this test scenario.

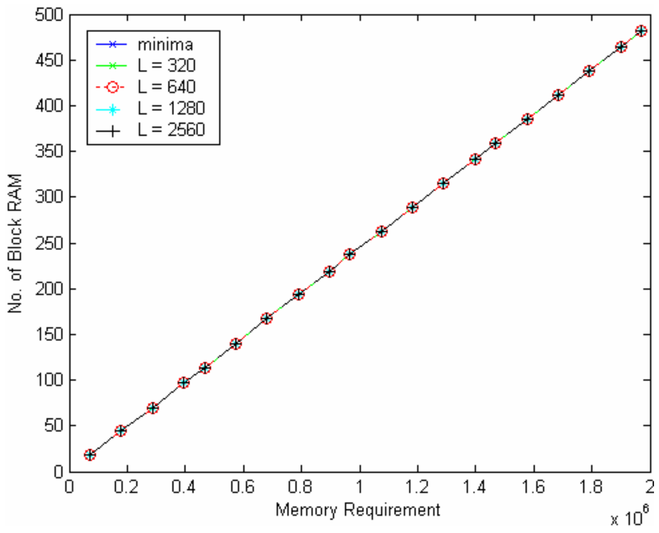


Fig. 21a. First test scenario on Spartan 2E.

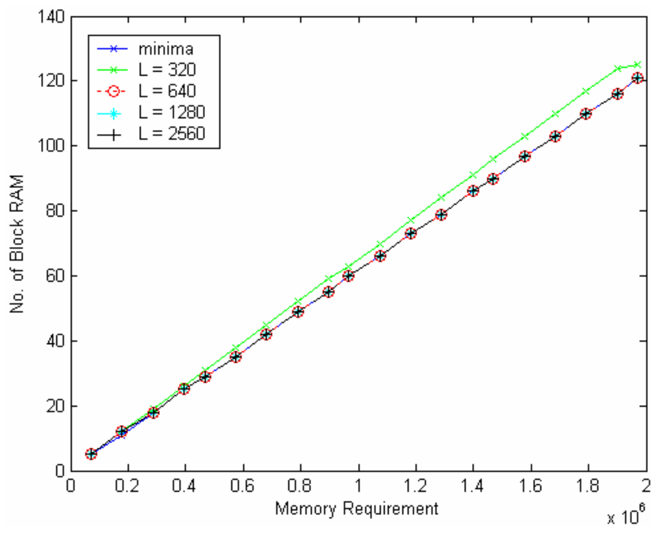


Fig. 21b. First test scenario on Spartan 3.

3.5.3.2 *Second test scenario*

In this test scenario, four values of width *W* (3, 6, 12 and 24) were used while the length *L* was determined by the memory requirement, which also ranged from 100 to 2000 Kbits. The test scenario was simulated for Spartan 2E and 3. The results obtained for the theoretical minima and the estimated Block RAMs for the different values of *W* were equal when Spartan 2E was used but differed when memory requirements less than 200kbit on Spartan 3 was used. Fig. 22 shows the performance of the algorithm for this test scenario.

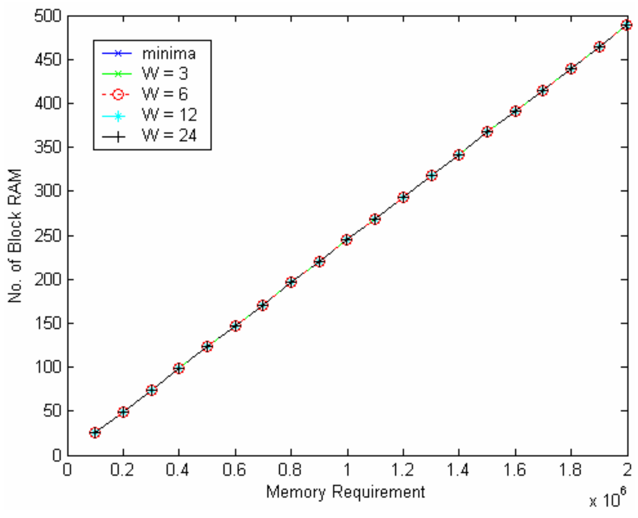


Fig. 22a. Second test scenario on Spartan 2E.

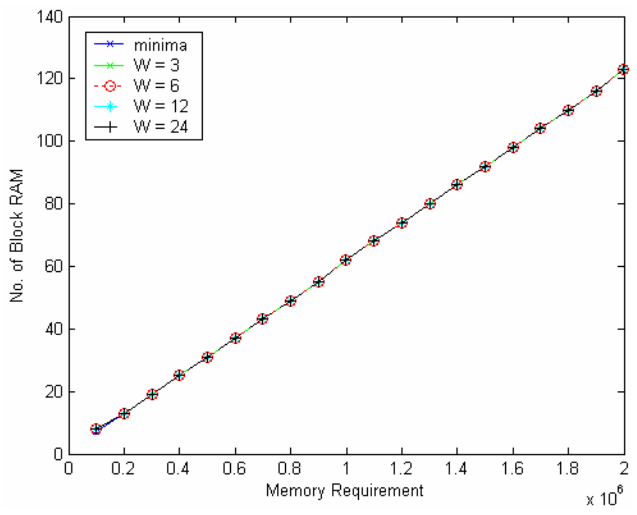


Fig. 22b. Second test scenario on Spartan 3.

As shown in Fig. 21 and Fig. 22, allocations on Spartan 2E equalled the theoretical minima while those on Spartan 3 differed slightly. This is because the Block RAM sizes are smaller in Spartan 2E and were more easily managed. In Fig. 21b, allocations with *L*=320 required I excess of the theoretical minima due to the small sizes of the memory objects with respect to the sizes of the Block RAMs. The average variation of the number of Block RAMs from the theoretical minima is 6%. In Fig. 22b, the first allocation with using *W*=3 had a variation of 14% from theoretical minima also due to the small sizes of the memory objects. Configuring the global memory objects width-wise to only data-path widths allowed by the FPGA technology leads to efficient utilization of the Block RAMs. This enables the allocation results to be close to theoretical minima.

By definition, the theoretical minimum assumes a Block RAM with infinite number of ports making it possible to allocate to the Block RAM until it is fully used. It is not a practical value but rather a metric used to measure the optimality of the algorithm. Consequently, the higher the number of ports on Block RAMs the closer the algorithm result is to theoretical minimum.

3.5.4 Performance analysis with varying length and Block RAM sizes

The performance of the memory synthesis has been investigated in this thesis using varying memory requirements with respect to the frame resolutions of RTVPS design cases in Table 1. The analysis is performed such that the design cases are allocated onto different existing and extrapolated FPGA memory architectures. Fig. 23 shows the results obtained for high (twice), medium (normal) and low (half) frame resolutions of the design cases in Table 1. In the figure the columns represent the frame resolutions. The upper and the lower rows represent the number of Block RAMs used for allocating the memory objects and the percentage of unused memories respectively. In the upper row Block RAM sizes were presented in increasing order from left to right but in decreasing order in the lower row.

The results reveal that for a given resolution, the amount of unused memory increases with Block RAM size. Also for high frame resolutions the amount of unused memory in the allocated Block RAMs is small when compared to the medium and low frame resolutions. This result is to be expected since the allocation of large memory objects onto small Block RAMs is more efficient than the allocation of small memories onto large Block RAMs. Hence, the use of un-multiplexed memory architecture will lead to more costly implementations. To avoid this, the FPGAs should support multiple RAMs sizes and wider data-paths. Alternatively, efficient use of current large RAMs can be achieved through the time-multiplexed architecture. However, this will degrade the performance and possibly increase the power consumption, which will make the FPGA architecture less attractive for video processing systems. These results can guide both RTVPS designers and the development of new FPGA architectures.

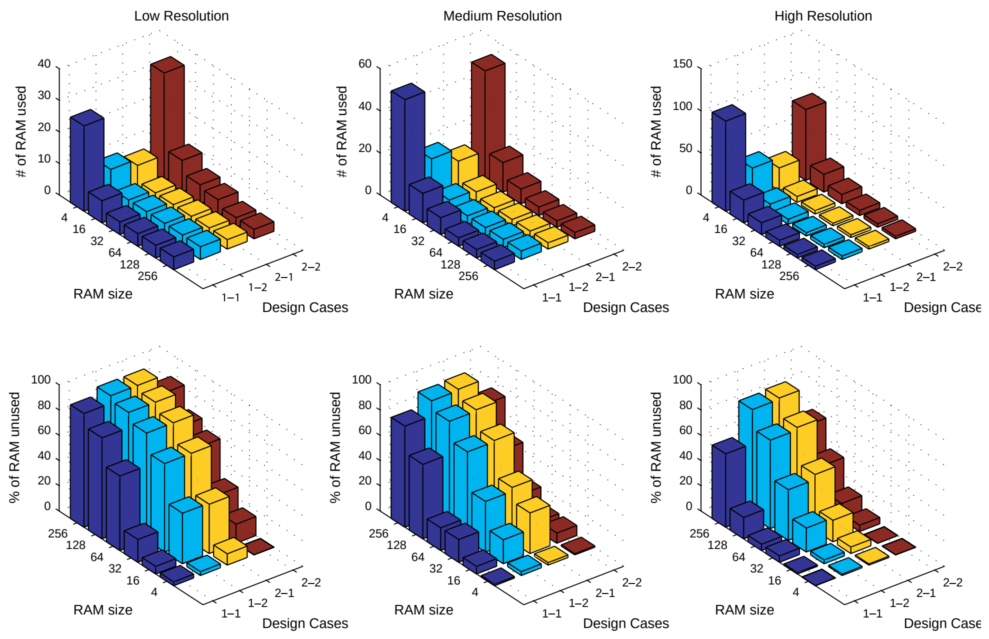


Fig. 23. Block RAM usage with varying memory requirements

3.5.5 Performance Analysis for video processing systems

In this section, the performance test of the allocation algorithm on video processing systems published in the literature [48]-[57] is presented. The tests are still in manuscript form and are to be sent for publication after further extensive testing. The algorithm was implemented using Block RAM sizes of 2, 4, 8, 16 and 32Kbits, each with data path width configurations of 2, 4, 8, 16, 32 bits.

The average results for the allocation of the test designs [48]-[57] are shown in Table 4. The results for all design cases were combined together in order to observe the memory sets producing the best allocation results. The most satisfactory allocation results were acquired using a RAM size of 4Kbits and a data path width of 16 or 32 bits, and this achieved an average allocation efficiency of 91.8%. However, larger memory sets, up to 16Kbit, also generated satisfactory results when combined with wide data path widths.

Table 4. Average allocation results for all cases

Average allocation efficiency					
	2 bit	**4 bit**	**8 bit**	**16 bit**	**32 bit**
2Kb	74,1%	88,2%	89,3%	91,5%	91,5%
4Kb	55,5%	82,4%	90,5%	91,8%	91,8%
8Kb	45,0%	62,4%	86,4%	92,5%	92,5%
16Kb	35,8%	52,9%	68,4%	88,1%	90,9%
32Kb	31,2%	44,0%	60,1%	73,2%	82,5%

The use of large memory sets, as predicted, proved to be inferior to that for small sets in the majority of cases which is in agreement with the allocation results for the architecture initially produced by O'Nils in [60]. The allocation efficiencies of the algorithm on RAMs with a size corresponding to the configuration of a Xilinx Spartan-2 and Spartan-3 are presented in Table 5 and Table 6. On both Spartan-2 and Spartan-3, the algorithm achieves a 100% allocation efficiency in 9 out of 11 cases.

Table 5. Allocation on Spartan II

Allocation result of the algorithm on Spartan 2			
Design case	Min. req. BRAM	Block RAM	Allocation efficiency
Case A [48]	3	5	60%
Case B [49]	5	5	100%
Case C [50]	1	1	100%
Case D [51]	2	2	100%
Case E [51]	3	3	100%
Case F [52]	1	1	100%
Case G [53]	5	10	50%
Case H [54]	7	7	100%
Case I [55]	1	1	100%
Case J [56]	21	21	100%
Case K [57]	51	51	100%

Table 6. Allocation on Spartan III

Allocation result of the algorithm on Spartan 3			
Design case	Min. req. BRAM	Block RAM	Allocation efficiency
Case A [48]	1	2	50%
Case B [49]	2	2	100%
Case C [50]	1	1	100%
Case D [51]	1	1	100%
Case E [51]	1	1	100%
Case F [52]	1	1	100%
Case G [53]	2	4	50%
Case H [54]	2	2	100%
Case I [55]	1	1	100%
Case J [56]	6	6	100%
Case K [57]	13	13	100%

3.5.6 Results of the addressing

Table 7 shows the resources required to access the allocated memory objects for the design cases in Table 1, the number of Block RAMs required for the allocations and the hardware operating frequency for the two approaches. Xilinx Spartan 3 FPGA was the target platform for implementing both approaches.

Table 7. Comparison of the two approaches.

	Case 1-1		**Case 1-2**		**Case 2-1**		**Case 2-2**	
	BP	**Dist**	**BP**	**Dist**	**BP**	**Dist**	**BP**	**Dist**
No. of 4 input LUTs:	653	994	334	356	155	191	560	804
No. of BRAMs:	14	14	6	6	5	5	13	13
Max. Frequency (MHz):	116	186	106	183	140	214	91	173
Frequency Comparison (%):	100	160	100	173	100	153	100	190

Depending on the number of partitions relating to a GMO, address look-up tables are required to set the enable signals and the values of the address signals to the appropriate Block RAMs on which the element of the GMO currently being pointed at is allocated, while also disabling related Block RAMs. In the Base Pointer Approach, these accesses to the Block RAMs are centrally controlled at the GMO level using a pointer. Hence, only one set of address look-up tables is required for each GMO. By contrast, in the Distributed Approach, each partition has its separate address look-up table, unrelated to those of related partitions. The use of a partition's address look-up table depends on the value of its enable signal. Hence the total number of address look-up tables for one GMO depends on the number of partitions making up the GMO. This is evident by comparing Fig. 18c and Fig. 19c. The first row of Table 7 confirms this. Thus the Base Pointer Approach yields more efficient use of hardware resources than does the Distributed Approach. The differences in resource requirements are however marginal, amounting to less than 3% of the available resources, for example, Xilinx Spartan 3 XC3S400 series [22].

Delays associated with large counter values in single based pointers and the distribution of the pointer values are eliminated in the Distributed Pointer Approach since each Block RAM partition has one local pointer. The use of small counters to evaluate addresses for each partition in the Distributed Pointer Approach increases the speed of memory accesses and consequently, increases operating frequency. This is because all signals required for memory accesses are calculated simultaneously at the clock edge. As the third and fourth rows in Table 7 show, the Distributed Approach yields more rapid access to data than does the Base Pointer Approach.

3.6 COMPARISON WITH DSP

The work in this thesis has been compared with digital signal processor. The objective of the comparison is to find the relationship between power consumption, performance and resource usage on FPGA and DSP and size of neighbourhood window required in RTVPS. The comparisons were conducted by means of three scenarios, namely, 1-bit morphology erosion, 8-bit average filter and 8-bit convolution filter. These filters are representative of different operations in neighbourhood oriented RTVPS - logical operation, addition and multiplication. For the convolution filters, 8-bit mask values were assumed. For these comparisons three neighbourhood sizes (3x3, 5x5 and 7x7) were used. For simplicity, neighbourhoods with square dimensions were chosen. Input video streams with 640-by-480 frame resolution were used.

3.6.1 DSP Implementation

The Texas Instrument, TMS320C64x DSP [64] was selected to implement the functions on a DSP. The C64x central processing unit (CPU) capable of operating at 500, 600 and 700 MHz consists of eight functional units (two of which are multipliers), two register files, and two data paths. The C64x multiplier has been enhanced so that it is capable of performing two 16-bit x 16-bit (or four 8-bit x 8-bit) multiplies every clock cycle.

To achieve the best performance, the simple approach to handling boundary conditions in neighbourhood oriented video processing system was adopted. The image size was simply increased in order to ensure that the boundary pixels are accurately filtered. This approach was motivated by the Texas Instruments implementation of a 3x3 convolution filter included in the IMGLIB [62], [63]. With this approach, the filter complexity is minimal with regards to the cost of speed due to extra rows and columns required for the boundary pixels. This approach is close to bench mark performance figures in [62].

The experiment set-up for DSP is as follows, the TMS320C6418 DSP is assumed to runs at 600MHz and that the input data stream is assumed at 10 MPixels/s thus lowering the CPU utilization and power consumption. Since this implementation avoids boundary conditions by increasing the image size, perfect cache hits are assumed as are local memory allocations for all the line-buffers. Additionally, one data read for the newest neighbourhood pixel and one memory write for the newly computed data corresponding to the centre pixel in the output image are also assumed. Using the Texas Instrument Code Composer Studio software version 2.10, it was possible to profile and achieve performances closer to the benchmarks values [63].

3.6.2 FPGA Implementation

The experiment set-up for FPGA took advantage of the memory architecture in this thesis. Fig. 24 depicts the implementation RTVPS filters. It was assumed that the input video stream is limited by the FPGA performance rather than the camera. The implementation was synthesised using Xilinx Integrated Software Environment software version 8.1i to obtain the post-place and route resource usage and performance. The Xilinx XPower software was used to calculate the power consumption per clock cycle.

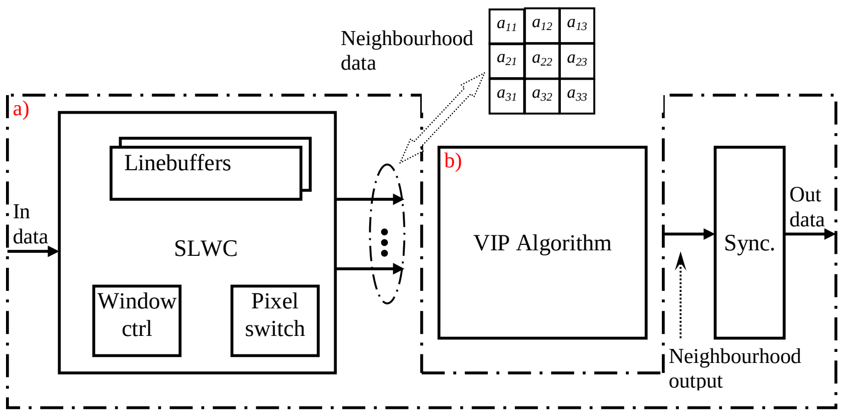


Fig. 24 Boundary conditions implementation architecture

The architecture in Fig. 24 handles data storage and boundary conditions for the spatial pixel neighbourhood in Fig. 4. In this figure the video/image processing (VIP) algorithm is the neighbourhood oriented RTVPS filter. It is glued to the architecture through the port interfaces for all the pixels data required in the neighbourhood. The sliding window controller (SLWC) monitors the central pixel in a spatial neighbourhood and using the position information provides valid data for all the pixels in the spatial neighbourhood through the *Linebuffers, Window ctrl* and *Pixel Switch*. The *Linebuffers* implement the line buffers in Fig. 4.

Window control (*Window ctrl*) provides the control signal used by the Pixel switch to build a spatial neighbourhood around the current pixel. The Pixel switch replaces all pixels in a spatial neighbourhood affected by the boundary condition, using predefined default values if the central pixel is at the image boundary. The output sync is required to realign the pixels with other streaming video signals since the generated output pixel corresponding to the central pixel of a neighbourhood oriented video processing system is usually skewed with respect to other image signals by an amount dependent on the neighbourhood size.

3.6.3 Comparison Results

Fig. 25 - Fig. 28 show the results obtained. It should be noted for the performance figures, that as long as there are available recourses on the FPGA, the performance for the system will be the same regardless of the number of active operators. For the DSP the performance (samples per second) will decrease when additional functionality is added to the system. Thus, the performance numbers are somewhat biased towards the DSP. The energy figures are also fairer in a comparison between the two architectures.

The results show that for this class of operations, with optimized memory allocation and the accessing method presented in this thesis, and full parallel and pipeline operations, FPGA achieves a better performances in between 2.0 to 8.7 in terms of throughput and an average reduced energy consumption of 80 times per sample.

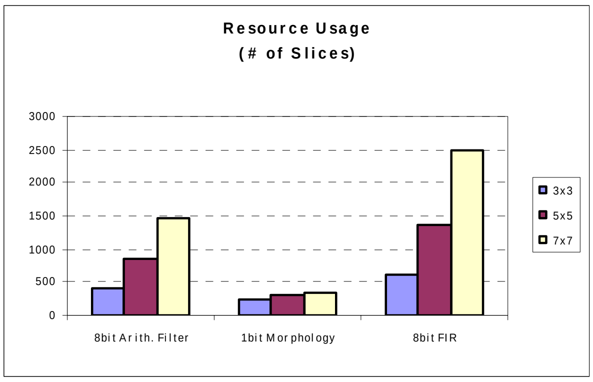


Fig. 25 Resource usage on FPGA

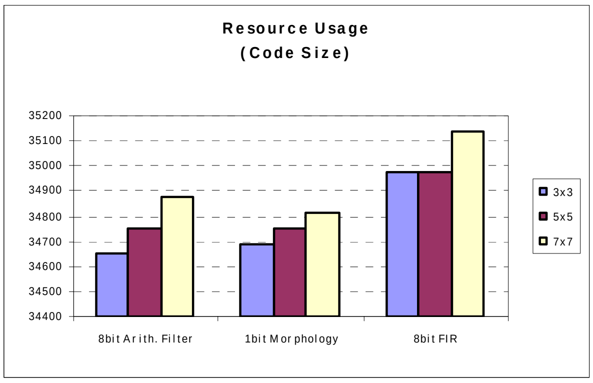


Fig. 26 Resource usage on DSP

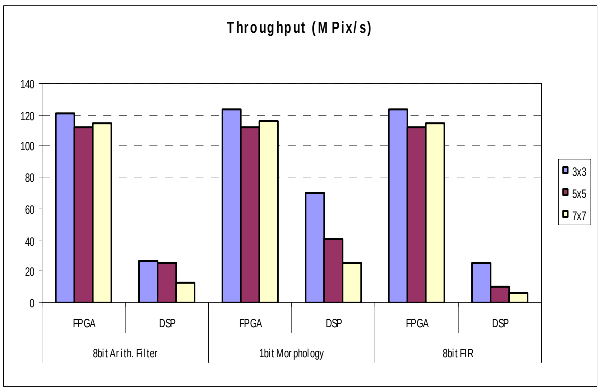


Fig. 27 Performance

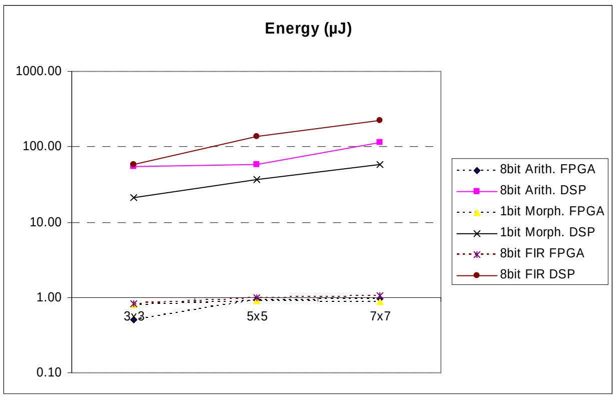


Fig. 28. Power consumption

3.7 INTEGRATION WITH IMEM

Fig. 29 depicts the integration of the tools used in IMEM and the steps required in the system synthesis and verification. The results from the memory requirement description from IMEM are accepted by into memory synthesis in order to generate a memory management module in VHDL and a SystemC wrapper module that contains a black-box reference to a memory management module implemented in VHDL. The SystemC wrapper also implements the C/C++ RTVPS filter function as a standalone clock sensitive thread.

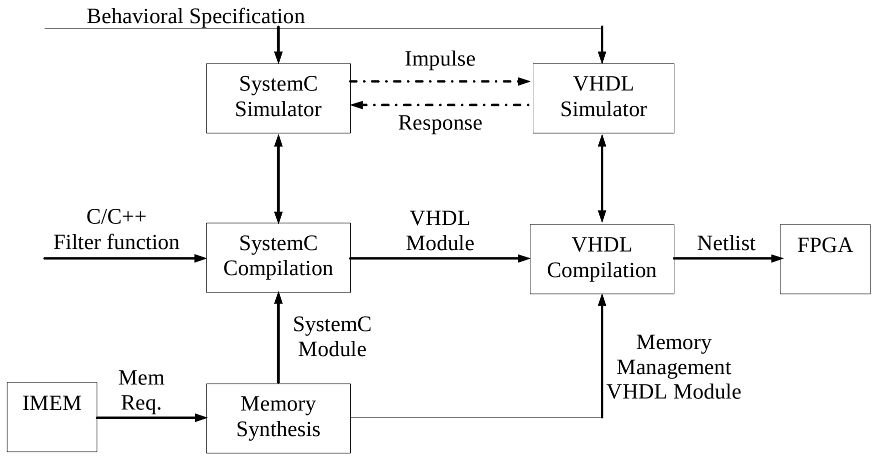


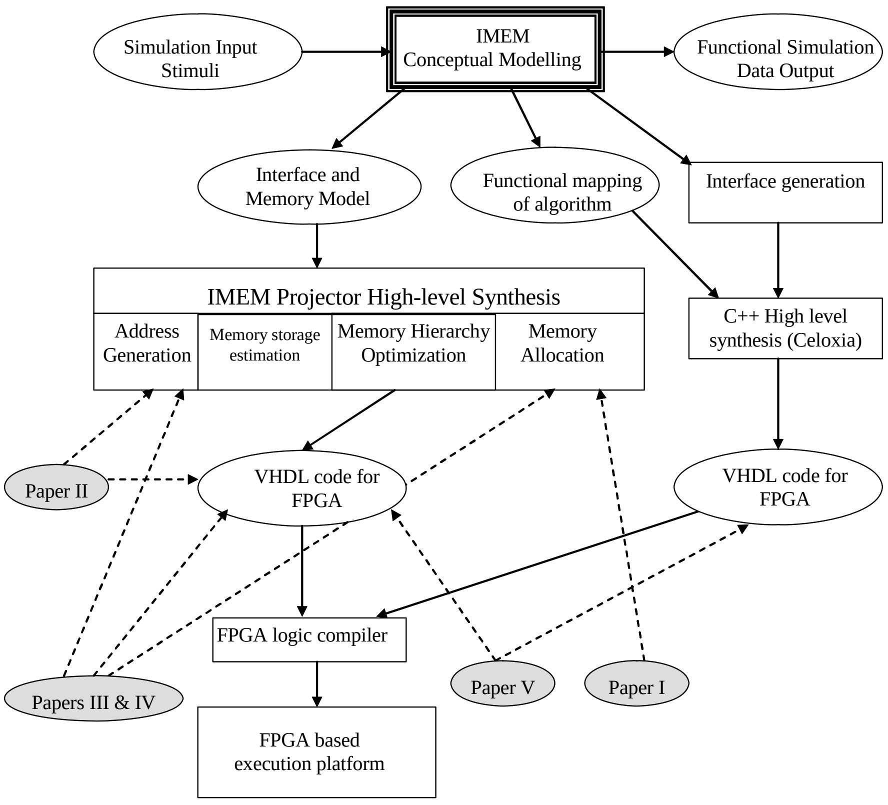
Fig. 29 System integration and verification

SystemC compilation iteratively refines the filter function embedded through simulation until a synthesizable module which satisfies the expected behavioural specifications of the RTVPS is achieved. This module is then synthesized into the VHDL module using the Agility Compiler. The SystemC simulator is also used to provide video signal impulse data to the VHDL simulator test-bench and to write its video response, which is used to verify that the VHDL module produces the expected result.

VHDL compilation instantiates the memory management module and the synthesizable filter function, implements the timing relation of the system data-flow and verifies the behaviour of the system through simulation. The final VHDL module is synthesized and downloaded into the FPGA.

4 PAPERS SUMMARY

Using the IMEM workflow as a guideline, the relationship between the four main papers in this thesis is shown in Fig. 30. The papers can be grouped as memory synthesis (allocation and addressing) and performance analysis. The papers are summarised as follows.



Simulation Input Stimuli
IMEM Conceptual Modelling
Functional Simulation Data Output
Interface and Memory Model
Functional mapping of algorithm
Interface generation
IMEM Projector High-level Synthesis
Address Generation
Memory storage estimation
Memory Hierarchy Optimization
Memory Allocation
C++ High level synthesis (Celoxia)
Paper II
VHDL code for FPGA
VHDL code for FPGA
FPGA logic compiler
Papers III & IV
Paper V
Paper I
FPGA based execution platform

Fig. 30 Relationship between thesis papers.

4.1 MEMORY SYNTHESIS

4.1.1 Paper I

This paper proposed and developed the allocation algorithm for allocating the estimated on-chip memory requirements. The algorithm is based on heuristics and near optimally allocates memories based on previously proposed memory architecture which was concluded to be efficient for real-time video processing systems. Optimal allocations is one in which the amount of unused memory location on instantiated memories is minimal, preferably zero.

4.1.2 Paper II

This paper proposed and developed two memory accessing approaches for allocated memories. The two approaches were compared and it was shown that one approach was more area efficient while the other was more speed efficient. Automatic generation of VHDL modules for managing (allocating and addressing) memories was implemented in order to access the efficiency of the two accessing approaches. The works this paper in conjunction with Paper I, provide a more efficient means of allocating on-chip memories than current practices in automatic synthesis tools.

4.2 PERFORMANCE ANALYSIS

4.2.1 Paper III

This paper presented an analysis of a variety of memory requirements of video processing systems allocated using these embedded memory resources. The analysis was performed using the memory architecture, allocation and addressing approaches in this thesis over a wide range of possible on-chip memory capacities and video resolutions. The analysis shows that should FPGAs support multiple memory sizes, greater use of on-chip memories would be achieved because according to the results obtained the amount of unused memory increases with Block RAM size for a given resolution. The paper also shows that the amount of unused memory reduces as video frame resolutions increases.

4.2.2 Paper IV

This paper presented a comparison of memory usage (based on the memory architecture, allocation and addressing developed in Papers I - III) in FPGA and a digital signal processor in neighbourhood oriented real-time video processing systems. The paper showed that this approach to memory management achieves better performances than DSP for different classes of operation.

4.2.3 Paper V

This paper presented a platform that automatically and optimally implements memory requirements for spatial and temporal real-time video processing systems targeting FPGAs. The platform is built on the works in this thesis in order to provide data interfaces to a filter core. The work manages boundary conditions to provide accurate data at image boundaries. The work in this paper relieves the video processing designer the burden of managing on memory requirements. It provides and instantiates a wrapper module for the filter such that the designer is only required to implement the filter algorithm in the wrapper.

4.3 AUTHORS CONTRIBUTIONS

The exact contributions of the authors of the four central papers in thesis are summarized in Table 8. In the table M and C represent the main author and co-author respectively.

Table 8. Authors' Contributions

Paper #	**NL**	**MO**	**HN**	**BT**	**Contributions**
I	M	C	C	C	NL: Developed and implemented the allocation algorithm MO: Supervisor HN: Analysis and discussion on algorithm feasibility BT: Analysed the algorithm results from formal modelling viewpoint.
II	M	C		C	NL: Developed and implemented the addressing approaches MO: Supervisor BT: Writing of introduction
III	M	C			NL: Implemented the experimental analyses MO: Supervisor
IV	M	C	C		NL: Implemented the comparison filters and test vehicles analyses MO: Supervisor HN: Provided the Module for managing the boundary conditions
V	C	C	M		NL: Implemented spatial memories and, spatial and spatio-temporal filters to test the implemented architecture MO: Supervisor HN: Provided the Module for managing the boundary conditions and provision of interface to background memory for temporal neighbourhood
1. Najeem Lawal (NL) 2. Mattias O'Nils (MO) 3. Håkan Norrel (HN) 4. Benny Thörnberg (BT)					

5 THESIS SUMMARY

Algorithms for allocating and accessing the memory requirements of neighbourhood oriented RTVPS operations have been presented in this thesis. The work in this thesis has been inspired by the efforts involved in finding best practices in memory allocation to FPGA embedded memory and IMEM's philosophy of memory modelling and synthesis independence of the synthesis of core RTVPS filters. This has led to demands for accurate memory estimations and efficient synthesis other than those currently available.

An introduction to the research area addressed in this thesis has been presented in Section 1. Section 2 summarised the FPGA resources relevant to this research and reviews of the previous works on memory allocation and addressing. In Section 3, the work achieved with referenced to finding automatic and efficient memory synthesis is presented. Section 4 provides brief summaries of the original papers covered by this thesis and the contributions of the authors to the papers.

This section presents the conclusion of the research work in this thesis and possible future works.

5.1 DISCUSSIONS

5.1.1 Memory architecture

For each neighbourhood oriented operation in an RTVPS, the developed memory architecture groups all the required memory objects (line buffers) to form a global memory object. This approach offers the advantage of reducing the number of memory object to be managed by the design. The architecture is based on the fact that all the memory objects required by an operator will be accessed simultaneously. This architecture leads to approximate savings of 50% with regards to the number of allocated memories for an operator. This is verified by observing that four memories would have been required to allocate the four line buffers identified in Fig. 11 if the conventional allocation approach has been followed as against the two allocated memories in Fig. 18 and Fig. 19.

5.1.2 Memory allocation

An allocation algorithm has been developed and implemented to the optimal use of allocated memories. This is based on the fact that inefficient allocations are performed by the current synthesis tools in which memory objects are allocated using high datapath widths whenever the memory object width is not supported. The approach in this algorithm is to partition such unsupported widths. The advantage of true dual-port memory allocations with the capability of writing and reading at both ports in one clock cycle was adopted in order to achieve optimal results. By this means, up to four memory-accessing operations could be performed in one clock cycle on one memory. The performance of the algorithm has been investigated using various on-chip memory sizes and video frame resolutions. It has been shown that efficient memory utilization increases are possible with smaller memories and larger memory requirements (as depicted in Fig. 23).

5.1.3 Memory addressing

Two addressing approaches for accessing memory have been proposed. The approaches are based on the regular pattern of data availability and production typical in video processing. One of the approaches tends to be implementation cost efficient giving about 3% savings in resources usage while the other produces higher access speed with about 50% higher speed performance. These two approaches give the designer possibility of choosing between resource and speed optimisation.

5.1.4 Boundary conditions management

The memory allocation and addressing algorithms have been implemented in order to provide all the pixel data in the first column in a pixel neighbourhood. Local register are required to delay pixel data for other locations in the neighbourhood (See Fig. 4). However, in order to ensure valid data are used at the image boundaries, an architecture has been developed and implemented, which replaces those neighbourhood pixels not within the image with a predetermined default value depending on the operation performed.

5.1.5 IMEM interfaces

The work in this research is part of the IMEM tool. It interfaces with IMEM to accept the description of the on-chip memory to be implemented as input and produces VHDL modules to manage the memory requirements. At the top level, data and control interfaces are provided for the core video processing algorithm (Fig. 24). This work allows the video processing designer to focus on the development of the processing algorithm while relying on IMEM to manage the memory requirements.

5.2 CONCLUSIONS

This thesis presents memory architecture and synthesis optimized for neighbourhood oriented real-time video processing systems in which memory write and read accesses exhibit a regular pattern.

The architecture considers the memory requirements of each operator in the video processing system in order to create one memory object. This memory object is synthesised using embedded memories in order to minimise external memory accesses. The synthesis and addressing of the memory requirements has been automated into a tool that accepts the description of the spatial memory requirements for all the operators in the video processing system to generate hardware description language (HDL) modules implementing the memories.

The work in this thesis has been integrated with other modelling and synthesis tools in order to create an environment for modelling, estimating, optimising and implementing both on-chip and off-chip memory requirements of neighbourhood-oriented video processing systems in addition to the boundary conditions of the algorithm. Within this environment, video processing engineers are only required to describe the memory requirements of the operators in terms of the number of frames, frame resolution, pixel resolutions and neighbourhood dimensions. The tools are able to implement all the memory requirements and thus enable the engineer to focus on the core algorithm for the system.

This work has been tested using many video processing systems with a variety of frame and pixel resolutions, neighbourhood dimensions and different sizes of embedded memories. The results were found to be very close to theoretical minima and still with high memory access speed performances.

FPGAs have been chosen as the target platform for the video processing systems studied in this thesis. This choice was made despite the challenges of programmability due to possibilities of reduced time-to-market, low non-recurring engineering cost and increasing embedded resources in comparison to ASICs, and efficiency of hardware implementation and high performance of embedded systems in comparison to DSPs. The contributions of this work reduce the challenges of system implementation on FPGA by reducing design time through efficient automated memory synthesis.

5.3 FUTURE WORKS

In the future, research works would focus on increasing the efficiency of allocating temporal data required in RTVPS, integrating algorithm compilers (as depicted in Fig. 24) and other IMEM tools (as depicted in Fig. 10). The goal is to provide a complete modelling, simulation and synthesis CAD-tool that follows the IMEM workflow to optimally implement both on- and off-chip memory for RTVPS. The tasks which are required to be carried out are described below.

5.3.1 Video data interface

The neighbourhood oriented operations in RTVPS require data from both even and odd rows. This sequence of data is not available in the current interlace video data streams. At Mid Sweden University, a video format has been developed to address this problem. In future works, interfaces will be provided for this video format both at the video input and at the output to video graphics adapter.

5.3.2 Tools interfacing

There are currently three tools in the IMEM workflow namely, IMEM modelling tool, IMAPPER tool and the memory allocation tool presented in this thesis. The modelling tool optimizes the temporal and spatial memory requirement of an RTVPS and provides the description of the requirement. The IMAPPER tool implements the temporal memory and boundary conditions of the RTVPS while the work in this thesis deals with the implementation of the spatial memory. In the future, these tools will be interfaced into a single CAD tool.

5.3.3 Prototyping environment

In accordance with the architecture in Fig. 24, an environment that accepts video processing algorithms in the form of VHDL modules (written manually or generated form C/C++/SystemC through for example, the Agility compiler) and manages all the memory requirements for the algorithms in addition to implementing video data input and VGA output will be developed. To use this environment the RTVPS engineer only requires to specify the video input format, the video and pixel resolutions and the memory requirements of the RTVPS algorithm. The environment will implement all the memory related issues required, data capture and VGA controller while the algorithm will be implemented by the engineer. The complete RTVPS will be synthesised and downloaded to a prototyping board. The Digilent Virtex II Pro Development System board equipped with a Digilent VDEC video decoder board and a 256Mbytes of fast DDR DRAM is currently being used for this work

5.3.4 Neighbourhood oriented operations

The memory objects managed in this thesis considered only data along rows in a video frame, although it is possible to extend this approach to columns of data. In future works, investigations into how this work can be extended to pixel blocks or tiles organised within a frame will take place. In this case, comparison will be made with regards to the effect of allocating smaller memories for image tiles in terms of the number of allocated memories, speed and area cost of addressing the allocated memories with those already developed in this thesis.

5.3.5 Central Controller State Machine

In the future, the number of pipelining stages involved for each operator in the video processing system and the number of frame and column/row buffers will be modelled and implemented into a central state-machine to control the sequence of operations in the system. This is necessary for power management and data synchronization among operators. An obvious advantage of this state machine is the elimination of data synchronisation buffers required in between video processing operators.

5.3.6 Power models

In the future, the difference in power consumptions between this work and traditional memory synthesis will be investigated. A means of achieving lower power consumption through efficient memory and power models will also be sought.

6 REFERENCE

[1] Gonzalez, R., and R. Woods, *Digital Image Processing*, 2nd edition, Addison-Wesley Pub., 2002.

[2] Bhatia, D., “Reconfigurable Computing”, *In Proc. of IEEE 10th Intl. Conf. on VLSI Design,* pp. 356-359, I997

[3] Brown, S. J., “An overview of technology, architecture and CAD tools for programmable logic devices”, *In Proc. of IEEE on Custom Integrated Circuits Conf.*, pp. 69-76, 1994.

[4] F. Yang and M. Paindavoine, “Implementation of an RBF Neural Network on Embedded Systems: Real-Time Face Tracking and Identity Verification”, *IEEE Trans. on Neural Networks*, Sept. 2003, pp. 1162 - 1175.

[5] B. A. Draper, J. R. Beveridge, A. P. W. Bohm, C. Ross and M. Chawathe, “Accelerated image processing on FPGAs”, *IEEE Trans. on Image Processing*, Dec. 2003, pp. 1543 - 1551.

[6] R. B. Lazarus and F. M. Meyer, “Realization of a dynamically reconfigurable preprocessor”, *IEEE Nat. Aerospace Electron Conf., 1993, pp 74-80.*

[7] J. Jiang, W. Luk. and D. Rueckert, “FPGA-based computation of free-form deformations in medical image registration”, *In Proc IEEE Intl. Conf. on Field-Programmable Technology (FPT)*, 2003, pp. 234 - 241

[8] A. S. Dawood, S. J. Visser and J. A. Williams, “Reconfigurable FPGAS for real time image processing in space”, *In Proc. IEEE Intl Conf. on DSP*, July 2002, pp. 845 - 848.

[9] P. McCurry, F. Morgan and L. Kilmartin, “Xilinx FPGA implementation of an image classifier for object detection applications”, *In Proc. of the IEEE Intl Conf. on Image Processing,* Oct. 2001, pp. 346 - 349.

[10] Z. Guo, W. Najjar, F. Vahid and K. Vissers, “A quantitative analysis of the speedup factors of FPGAs over processors”, *In Proc. of ACM/SIGDA 12th Intl Symp. On FPGA, 2004*, pp. 162 - 170*.*

[11] Xilinx, *Spartan FPGAs - Gate Array solutions*, www.xilinx.com

[12] Tessier, R. and Burleson, W., “Reconfigurable Computing for Digital Signal Processing: A Survey”, *Journal of VLSI Signal Processing, Kluwer Academic Publishers, 2001*.

[13] Standard VHDL Language Reference Manual http://www.eda.org/vhdl-200x/

[14] Verilog, http://www.eda.org/sv/

[15] De Micheli, G., “Hardware synthesis from C/C++ models”, *Proc. of IEEE Design, Auto & Test in Europe Conf & Exhibition*, pp. 382-383, Mar 1999.

[16] Open SystemC Initiative, “SystemC User’s Guide”, version 2.0.1, www.systemc.org

[17] Haldar, M., Nayak, A., Choudhary, A., and Banerjee, P., “A system for synthesizing optimized FPGA hardware from MATLAB”, *IEEE/ACM Inter. Conf. on CAD*, pp. 314-319, Nov 2001.

[18] Kuhn, T. and Rosenstiel, W. “Java based object oriented hardware specification and synthesis” *In Proc. of ASP-DAC* pp. 579-581, Jan 2000.

[19] Kim, D., “An Implementation of Fuzzy Logic Controller on the Reconfigurable FPGA System”, *IEEE Trans. on Industrial Electronics*, Jun 2000, pp 703-715

[20] Cowen, C.P. and Monaghan, S., “A reconfigurable Monte-Carlo clustering processor (MCCP)”, *In Proc. of IEEE Wksp on FPGAs for Custom Computing Machines*, pp. 59 – 65, Apr. 1994.

[21] Smith, M. J. S., *Application Specific Integrated Circuits*, Addison Wesley
[22] Xilinx, “Spartan-3 FPGA Family: Complete Data Sheet”, www.xilinx.com
[23] Wolf, W., FPGA-Based System Design, Prentice Hall, 2004
[24] Xilinx. “Using Block RAM in Spartan-3 FPGAs”, www.xilinx.com
[25] Altera, “Stratix II Architecture”, www.altera.com
[26] QuickLogic, “Eclipse II Family Data Sheet”, www.quicklogic.com
[27] Xilinx, “MicroBlaze Microcontroller Reference Design User Guide”, www.xilinx.com
[28] P. Diniz, and J. Park, “Automatic synthesis of data storage and control structure for FPGA-based computing engines.” *In Proc FCCM’00, 2000, IEEE Computer Society Press*, pp. 91 - 100.
[29] L. Ramachandran, D. D. Gajski, and V. Chaiyakul, “An Algorithm for Array Variable Clustering”, *In Proc. Europ. Des. Test. Conf.*, Feb.1994, pp. 262 - 266.
[30] M. Gokhale and J. Stone “Automatic Allocation of Arrays to Memories in FPGA Processors with Multiple Memory Banks”, *in Proc. of the IEEE Symp. on Field-Programmable Custom Machines*, 1999, pp. 63-69.
[31] N. Baradaran, J. Park, and P.C. Diniz, “Compiler reuse analysis for the mapping of data in FPGAs with RAM blocks”, *In Proc. IEEE Intl. Conf. on Field-Programmable Technology*, pp. 145-152, 2004.
[32] H. Schmit and D.E. Thomas, “Synthesis of application-specific memory designs”, *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, pp. 101 - 111, March 1997.
[33] P.K. Jha, and N.D. Dutt, “High-level library mapping for memories”, *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, pp. 566 - 603, July 2000.
[34] M. Doggett, and M. Meissner, “A Memory Addressing And Access Design for Real Time Volume Rendering”, *In Proc of IEEE Int. Symp. on Circuits and Systems*, pp. 344 - 347, June 1999.
[35] D. Grant, P.B. Denyer, and I. Finlay, “Synthesis of Address Generators”, *Digest of Tech. Papers of IEEE Int. Conf on Computer-Aided Design*, pp 116-119, Nov 1989.
[36] J. Seo, T. Kim, and P.R. Panda, “Memory Allocation and Mapping in High-Level Synthesis - An Integrated Approach”, *IEEE Trans. on VLSI Syst.*, pp. 928 - 938, Oct. 2003.
[37] H. Schmit and D. E. Thomas, “Address generation for memories containing multiple arrays”, *IEEE Trans. on CAD of Integ. Cct. and Sys.*, pp. 377 – 385, May 1998.
[38] M. Miranda, F. Catthoor, M. Janssen and H. De Man, “High-level address optimization and synthesis techniques for data-transfer-intensive applications”, *IEEE Trans. on VLSI Systems,* pp. 677-686, Dec 1998.
[39] R. Leupers, and P. Marwedel, “Algorithms for Address Assignment in DSP Code Generation”, *Digest of Tech. Papers of IEEE/ACM Int. Conf. on Computer-Aided Design*, pp. 109 - 112, Nov. 1996.
[40] N. Sugino, H. Miyazaki, S. Iimuro, and A. Nishihara, "Improved Code Optimization Method Utilizing Memory Addressing Operation and its Application to DSP Compiler", *IEEE International Symposium on Circuits and Systems*, pp. 249 - 252, May 1996.

[41] Vitabile, S., Gentile, A., Siniscalchi, S.M. and Sorbello, F., “Efficient Rapid Prototyping of Image and Video Processing Algorithms”, *In Proc. of EUROMICRO Systems on Digital System Design*, 2004

[42] Drayer, H.T., and Araman, P.A., “A development System for Creating Real-Time Machine Vision Hardware Using Field Programmable Gate Arrays”, *Proc. of 32n Hawaii Inter. Conf. on System Sciences*, 1999.

[43] Bariamis, D. G., Iakovidis, D. K., Maroulis, D. E. and Karkanis, S.A., “An FPGA-Based Architecture for Real Time Image Feature Extraction”, *Int’l Conf. on Pattern Recognition (ICPR'04)*, pp. 801-804

[44] Longfei, R. and Songyu, S. “Real-time duplex digital video surveillance system and its implementation with FPGA”, *In Proc. of IEEE Int’l Conf. on ASIC,* 2001, pp 471-473.

[45] Panda, P.R. and Dutt, N.D., “Reducing Address Bus Transitions for Low Power Memory Mapping”, *IEEE* 1996, pp. 63 - 67.

[46] Park, J. and Diniz, P.C., “Synthesis of Pipelined Memory Access Controllers for Streamed Data Applications on FPGA-based Computing Engines”, *ISSS*, Oct. 2001, pp 221 - 226.

[47] Herz, M., Hartenstein, R., Miranda, M., Brockmeyer, E. and Catthoor, F., “Memory Addressing Organization for Stream-Based Reconfigurable Computing”, *IEEE* 2002, pp 813 – 817.

[48] J. Pan, S. Li, and Y. Zhang, “Automatic extraction of moving objects using multiple features and multiple frames”, *In Proc. Of IEEE International Symposium on Circuits and Systems*, *Emerging Technologies for the 21st Century*, May 2000.

[49] A. Smith, and M. Teal, “Identification and tracking of maritime objects in near-infrared image sequences or collision avoidance”, *7th International Conference on Image Processing and Its Applications* (Conf. Publ. No.465), pp.250-4 vol.1, July 1999.

[50] J. Jang, D. Yu, and Z. Sun, “Real-time image processing system based on FPGA for electronic endoscope”, *In Proc. of IEEE Asia-Pacific Conference on Circuits and Systems. Electronic Communication Systems*. (Cat. No.00EX394), pp. 682-5, Dec. 2000.

[51] I. Andreadis, and G. Louverdis, “Real-time adaptive image impulse noise suppression”, *IEEE Trans. Instrum. Measurements*, pp. 798-806, June 2004.

[52] T. Zhang, and C. Suen, “A fast thinning algorithm for thinning digital patterns”, *Commun. ACM*, vol.27, no.3, pp. 236-239, Mar. 1984.

[53] R. Rad, and M. Jamzad, “Real-time classification and tracking of multiple vehicles in highways”, *Pattern Recognition Letters*, Vol. 26, July 2005.

[54] A. Bosco, M Mancuso, S. Battiato, and G. Spampinato, “Temporal noise reduction of bayer matrixed video data”, *In Proc. of IEEE International Conference on Multimedia and Expo* (Cat. No.02TH8604), pp.681-4 vol.1, Aug. 2002.

[55] J. Zheng, D. Feng, Y. Zhang, W. Siu, and R. Zhao, “An algorithm for video monitoring under a slow moving background”, *In Proc. of the First Intl Conf. on Machine Learning and Cybernetics*, Beijing, 4-5 Nov 2002.

[56] D. Zheng, Y. Zhao, and J. Wang, “An efficient method of licence plate location”, *Pattern Recognition Letters*, pp. 2431-2438, Vol. 26, Issue 15, June 2005.

[57] A. Abouelela, H. Abbas, H. Eldeeb, A. Wahdan, and S. Nassar, “Automated vision system for localizing structural defects in textile fabrics”, *Pattern Recognition Letters*, 26 pp. 1435-1443, 15 July 2005.

[58] Thörnberg, B., Norell, N. and O'Nils, M., ”IMEM: An object-oriented memory- and interface modelling approach for real-time video systems”, *In Proc. of the Forum on specification & Design Languages, Marseille*, September 2002

[59] Thörnberg, B., Norell, N. and O'Nils, M., ”Conceptual Interface and Memory-Modeling for Real-Time Image Processing Systems. IMEM: A tool for Modeling, Simulation and Design Parameter Extraction”, *In Proc. of IEEE Workshop on Multimedia Signal Processing*, Dec. 2002.

[60] M. O’Nils, B. Thörnberg and H. Norell, “A Comparison between Local and Global Memory Allocation for FPGA Implementation of Real-Time Video Processing Systems”, *In Proc of IEEE Int.Conf. on Signals and Electronics Systems*, Sept 2004.

[61] XILINX, Using Block SelectRAM+ Memory in Spartan-II FPGAs, XAPP173 (v1.1), Dec 2000.

[62] Texas Instruments, TMS320C64x Image/Video Processing Library, http://www.ti.com

[63] Texas Instruments, *TMS320C6000 Programmer’s Guide*, http://www.ti.com

[64] Texas Instruments, *TMS320C64x Technical Overview*, http://www.ti.com

[65] Sanguinetti, J. and Pursley, D., “High-Level Modeling and Hardware Implementation with General-Purpose Languages and High-level Synthesis”, April 2002.

[66] Synopsys, C2HDL Compiler, www.synopsys.com

[67] Agility Compiler, www.celoxica.com/agility

[68] Martinolle, F. and Parvathy, U., "Mixed language design data access: procedural interface design considerations", *In Proc. of VHDL Intl Users Forum Fall Workshop,* 2000, pp. 95 - 99.

[69] Sasaki, H., "A formal semantics for Verilog-VHDL simulation interoperability by abstract state machine", *In Proc. of DATE Conference and Exhibition* 1999, pp. 353 – 357

[70] Catthoor, F. et al., Custom Memory Management Methodology. Kluwer Academic Publishers, 1998, ISBN 0-7923-8288-9.

APPENDIX A

The proposed allocation algorithm is presented in Figure A in pseudo-code. In step 1, the algorithm creates global memory objects according to Eq. (1). In step 2, the algorithm ensures that they conform to the allowable port width configuration according to definition vii. This step is captured in a procedure, *configure_global_memory_objects(R)*, presented below the algorithm in Figure A.1. In steps 3 through to 10, the global memory objects are allocated to the Block RAMs according to definitions viii to xi. In steps 11 through to 20, the algorithm finds the optimal use of unallocated memory space in the Block RAM through the second port. This allocation is also in accordance with definitions viii to xi. Steps 5 and 14 handle the partitioning of the global memory objects with respect to length by allocating part of the length of the memory object to the Block RAM until the memory object has been completely allocated. In steps 7 to 9 and 15 to 17, the algorithm estimates the amount of the memory object possible for allocation to the available space on a Block RAM. This amount is used to update the memory object and the Block RAM if the allocation decision is made. In steps 18 to 20, the algorithm finds the memory object which, when allocated to the remaining space on the current Block RAM through port B, yields the optimal use of the Block RAM. The optimal allocation is that for which the unused memory space is minimum, preferably zero.

The procedure for configuring the width of the global memory objects, *configure_global_memory_objects(R)*, is based on definitions (iii) and (vii). In step 1 of the procedure, a container for the set of global memory objects is created. In this procedure, as the global memory objects are configured they are placed in this container. The container is returned in step 17 as the output of the procedure. As the procedure loops through the set of global memory objects in step 2, the width of each global memory object, $W_{Ri,}$ is obtained in step 3 and compared in step 4 with W_n. If W_{Ri} is not supported by the FPGA, the segment identifier is created in step 5. In steps 6 to 14, W_c is looped through and its members, W_n, are compared with the W_{Ri}. This comparison starts from the largest W_n down to the smallest. An appropriate number of times by which W_{Ri} is greater than W_n is used in creating segments according to definition vii. W_{Ri} is updated and reused until it is reduced to zero. If the FPGA supports W_{Ri}, in steps 15 and 16, the object is left un-partitioned and placed in the returned container.

The Proposed Allocation Algorithm

```
Algorithm:  Memory Allocation(R, M)
Parameters: R[$R_1$ … $R_I$] set of I memory objects;
            M[$M_1$ … $M_K$] set of K Block RAMs;
Return:     M[$M_1$ … $M_K$] set of K Allocated Block RAMs;

{
1. create global memory object;
2. R := configure_global_memory_objects(R);
3. for $M_k$ := $M_1$ upto $M_K$
4. { for $R_i$ := $R_1$ upto $R_I$
5.   { determine length of $R_i$ to be allocated;
6.     determine port on $M_k$ for allocation;
7.     Allocate $R_i$ to $M_k$;
8.     update $M_k$;
9.     update $R_i$;
10.    if $M_k$ has been completely used
       { take next $M_k$;
       }
11.    else
12.    { if no_of_ports on $M_k$ = 1
13.      { pair($R_i$,$M_k$.unused) best_alloc;
14.        flag := TRUE;
15.        for $R_j$ := $R_1$ upto $R_I$
16.        { determine length of $R_j$ to be allocated
17.          temporarily Allocate $R_j$ to $M_k$;
18.          temporarily update $M_k$;
17.          temporarily update $R_j$;
19.          if $M_k$ is completely used
             { Allocate $R_j$to $M_k$;
               flag = FALSE;
               take next $M_k$;
             }
20.          pair($R_j$,$M_k$.unused) temp_alloc;
21.          if temp_alloc.second < best_alloc.second
             { best_alloc := temp_alloc;
             }
           }
22.        if flag = TRUE
           { $R_i$ := best_alloc.first;
             Allocate $R_i$ to $M_k$;
             update $M_k$;
             update $R_i$;
           }
         }
       }
     }
   }
}

Procedure:  configure_global_memory_objects(R)
Parameters: R[$R_1$ … $R_I$] set of I memory objects;
Return:     R[$R_1$ … $R_I$] set of I memory objects;

{
1.  create new set of memory objects New_R;
2.  for $R_i$ := $R_1$ upto $R_I$
3.  { width := $R_i$.width;
4.    if width $\notin W_c$
5.    { segment_id := 1;
6.      foreach $W_i$ in $W_c$
7.      { if width $\geq W_i$
8.        { count_max := width / $W_i$; // integer division
9.          width := width – ($W_i \times$ count_max);
10.         for count := 1 upto count_max
11.         { Mem_Obj temp($W_i$, $R_i$.length, $R_i$.operator_id);
12.           temp.set_segment(segment_id);
13.           add temp to new_R;
14.           segment_id := segment_id + 1;
            }
          }
        }
      }
15.    else
16.    { add $R_i$ to new_R;
       }
    }
17. return new_R;
}
```

Figure A.1. The proposed allocation algorithm.