Design Considerations for a Domain Specific Vector Microarchitecture

Bogdan Spinean, Cătălin Ciobanu, Georgi Kuzmanov, Georgi Gaydadjiev
{bogdan, catalin}@ce.et.tudelft.nl, {g.k.kuzmanov, g.n.gaydadjievg}@tudelft.nl
Computer Engineering Laboratory, Electrical Engineering Department,
Delft University of Technology,
Postbus 5031, 2600 GA Delft, The Netherlands
Tel (+31) 15 27 82 226, Fax (+31) 15 27 84 898

Abstract— **In this article, we analyze the speedup potentials of media and signal processing software on vector processors. We evaluate the impact on performance of several design decisions such as the vector register length, memory latency, memory bandwidth and the number of parallel lanes in the datapath. To quantify the influence of the aforementioned design parameters, we modify SimpleScalar 3.0 by adding new vector instructions, a vector register file, and vector functional units and simulate several media and signal processing applications. Simulation results indicate that through vectorization we can obtain kernel speedups ranging from 5.36x to 17.34x and application speedups of 1.82x and 1.37x for the MPEG2 encoder and decoder respectively.**

Keywords— **Vector processors, microarchitecture, simulation**

I. Introduction

The high amount of data level parallelism in current applications indicates vector architectures as strong candidates for increasing performance beyond the current limits of instruction level and thread level parallelism. In this article, we analyze the speedup potentials of media and signal processing software on vector processors. We evaluate the impact on performance of several architectural and microarchitectural design decisions. As presented in [3], the front end of a vector machine (decode and issue logic) is not a bottleneck for vector processors. Therefore, we focus on aspects concerning the datapath such as the vector register length, memory latency, memory bandwidth and the number of parallel lanes. We have chosen to focus on the MPEG2 encoder and decoder, we profile these applications, find the most computationally intensive kernels and vectorize their code. To quantify the influence of the aforementioned design parameters, we modify SimpleScalar 3.0 by adding new vector instructions, a vector register file, and vector functional units. After a preliminary design space exploration, we find a set of feasible ranges of configurations. We then thoroughly simulate the architecture in search of the best parameter values. Results indicate that through vectorization and properly set design parameters, we can obtain kernel speedups ranging from 5.36x to 17.34x and application speedups of 1.82x and 1.37x for the MPEG2 encoder and decoder respectively. Based on our experiments and analysis, we argue that the optimal vector processor organization for the considered application domains is a multi-lane vector unit tailored to short vectors and low memory latencies.

The remainder of this paper is organized as follows: Section II provides the necessary background and motivation for our work. Section III presents the framework we have developed and section IV describes the experiments we have performed as well as the results we have obtained. Section V presents our conclusions and future work.

II. Background

General purpose processors are designed to provide implementability for a wide range of applications. Whether these applications are computation intensive, memory intensive or control intensive, they can all be executed on a general purpose processor but in many cases with serious performance degradations. However, computationally intensive applications, such as multimedia and scientific applications can execute significantly faster on processors that have support for vector operations.

A vector processor is able to run logic or arithmetic operations on multiple data elements simultaneously. For a scalar processor, instructions are issued for processing every single piece of data (data element). In, contrast, vector processor's instructions operate on entire arrays of data elements. For instance, the addition of two vectors, element by element, can be done

by a vector processor using a single instruction.

Vector processors mainly exploit data level parallelism, multiple similar operations are packed in a single instruction greatly simplifying control by reducing instruction dependency checks. Thus, the focus of the design process shifts from the control part of the processor to the datapath.

The concept of vector processing through the years has undergone several transformations. It was the core principle behind the supercomputers of the 80's and mid 90's when these supercomputers and their applications were built specifically for vector processing. Starting from the 90's, vector processing was used in the form of multimedia instruction extensions of superscalar processors [5]. These provided good speedups for certain computation intensive applications at a reduced implementation cost.

Nowadays, we are facing a new paradigm shift, architects are looking towards heterogeneous systems containing various processors, each specialized in different operations and classes of operations. A good example is the Cell processor with its general purpose processor aided by 8 synergistic cores which are in fact a form of vector units [6]. This motivates the need to study further and better understand the inner workings of vector processors.

Figure 1 shows a block diagram of a vector processor that consists of a conventional scalar unit and a vector unit both working in parallel. The fetch unit distributes the program instructions to the corresponding scalar or vector unit. The segments of the program rich in branches and conditional statements are executed by the scalar processor. On the other hand, uniform, computation intensive, data parallel segments of code are executed by the vector unit.

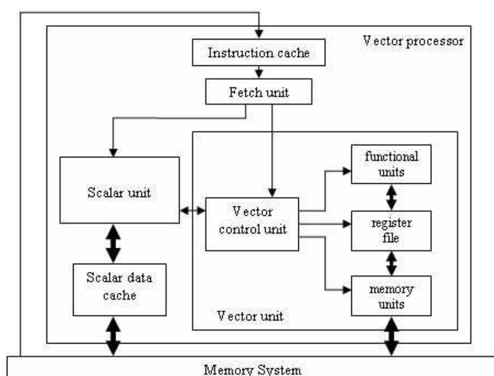


Fig. 1. Block diagram of a vector processor

From Figure 1, we can identify the key components of the vector unit. Data is loaded into the vector register file and each vector register contains multiple elements, typically 64 or 128, each element being 32 or 64 bits wide. Vector functional units usually are fully pipelined and can start a new operation on every clock cycle. Since an instruction generates tens of operations back to back, throughput is more important than latency. The parallel semantics of a vector instruction allows an implementation using either a deeply pipelined functional unit or an array of parallel functional units, or a combination of parallel and pipelined functional units. Each parallel functional unit is called a lane. Note that while the vector register length is an architectural parameter and fixed by the ISA, the number of lanes is a microarchitectural decision allowing a great flexibility when tackling the tradeoff between cost and performance.

Another very important particularity of the vector processors is the memory system. In order for the datapath to work at full speed, it requires support from an equally fast memory system to transfer data between main memory and registers. The memory must provide a very fast flowing stream of data in and out of the processor. The bandwidth is the most important aspect, large latencies can be tolerated because the startup time is mitigated over a large number of elements transferred.

The following section describes the simulation framework we have developed in order to evaluate the influence of the major design parameters involved when designing a vector processor.

III. Simulation framework

A. Simulation Model

We have modified the out of order processor model of the SimpleScalar 3.0 simulator suite. Sim-outorder is a cycle accurate simulator that models an out of order superscalar processor. To the original superscalar processor simulated by sim-outorder, we have added a vector register file with the corresponding bit vectors, vector functional units, memory units and vector instructions.

The vector instruction set we have implemented is very similar to the instruction of the Cray-1 [7] and the VMIPS [4]. The vector instructions take as their input either a pair of vector registers (v.add) or a vector register and a scalar register (v.sadd). For the latter instruction, the value in the scalar register is used as the input for all operations. Most vector operations have a vector destination register, although few, such as *population count*, produce a scalar value

which is stored in a scalar register. Load instructions have as destination a vector register while Store instructions have the vector register as source. Support for conditional execution is provided through the use of mask registers which are Boolean vectors that control the execution of a vector instruction just as scalar predicated (conditionally executed) instructions use a Boolean condition to determine whether an instruction writes back its result. When the vector mask mode is enabled, vector instructions operate only on the vector elements whose corresponding entries in the mask register are logic '1' . If the vector mask register is set by the result of a condition, only elements satisfying the condition will be affected.

We have added two types of computational units and two types of memory access units. The vector ALU can perform additions and subtractions between two vector registers or between a vector register and a scalar register. Also, the vector ALU assigns bit vector values as results of comparisons. As in the case of computations, the comparisons can be between two vector registers or between the elements of a vector register and a scalar register. The vector multiplication unit can perform multiplications and divisions, both between two vector registers or between a vector register and a scalar register. These functional units have the same execution latency for all of the operations they can execute and is of the following form:

$$Execution\ latency = Startup\ latency + \frac{SectionSize}{Number\ of\ lanes} \quad (1)$$

All of these values are simulator parameters:

- Startup latency depends on the operation and in our experiments we have decided to use 2 cycles for addition and 4 cycles for multiplication.
- Section size is equal to the number of elements each vector register can store
- The number of lanes is the number of physical units inside a functional unit that work in parallel to perform the operations on the elements of a vector register. For instance in a two lane implementation there are two physical functional units, one performing operations on the odd elements of the vector register, the other one on the even elements of the vector register.

The vector memory units have very similar parameters. For the load we have:

$$Load\ latency = Memory\ latency + \frac{SectionSize}{Words\ per\ cycle} \quad (2)$$

These parameters are:

- Memory latency is the number of cycles it takes for the first word to come from memory after a read has been requested
- Section Size is the number of elements contained in a vector register
- Words per cycle is the number of elements that can be transferred each cycle, it is the memory bandwidth expressed in words.

The vector memory accesses are bypassing the cache, accessing main memory directly. The timing of these accesses (the vector memory latency) is modeled into the functional unit latency. This has reduced the complexity of our simulator eliminating the need to model a vector memory system.

For the store unit, the memory latency is not visible to the processor. Thus, a store instruction completes in at most section size cycles after it starts.

$$Store\ latency = \frac{SectionSize}{Words\ per\ cycle} \quad (3)$$

B. *Simulation Statistics*

In addition to the functionality of our simulator, we have also added simulation statistics to better evaluate performance and to aid us in the process of decision making and in refining our tests. There are two major types of statistics that will be described in the following subsections:

- Instruction related statistics
- Functional unit related statistics

The instructions statistics count the number of vector instructions that are issued. Statistics have been implemented for:

- The total number of vector instructions issued
- The percentage of the total amount of instructions occupied by the vector instructions
- The total number of vector computation instruction
  - The number of instructions that use the vector ALU
  - The number of instructions that use the vector multiply unit
- The total number of vector memory instructions
  - The number of vector stores
  - The number of vector loads
- The number of vector control instructions (like those used for turning mask operation on or off, auto-sectioning instructions)

The functional unit related statistics track the usage of the vector functional units on a cycle basis. We count the number of cycles each of the functional unit is busy and we gather the following statistics for each:

- A total number of cycles that each functional unit was busy
- An occupancy rate meaning the percent of all execution cycles that each functional unit was busy

Another class of statistics is the history of occupancy rate for each of the functional units. For instance, we can calculate the occupancy rate of a functional unit over each thousand cycles.

Using the history of the occupancy rates for any of the functional units or memory units, we can visually follow the evolution of the simulation. Also, these statistics can be a very valuable tool for debugging both the simulator and the process of vectorizing the applications.

C. *Experimental Procedure*

SimpleScalar supports multiple instruction set architectures. We have used the default one: PISA (Portable ISA), a MIPS based instruction set. The simulator also comes with a compiler, a linker and a loader for this particular architecture. The compiler is gcc-2.7.2.3 with a modified back-end to generate code for PISA.

For every selected application, we performed the following experimental procedure:

1. Profile the application in search of computation intensive kernels
2. Inspect the kernels for their vectorization potential
3. Compile the application for the PISA architecture, the architecture used by our simulator
4. Run the application on the simulator to assess the initial performance, before vectorization
5. Manually Vectorize the computation intensive kernels previously identified
6. Simulate the vectorized applications by varying parameters
7. Analyze results

For profiling the selected applications, we have used the Linux application *callgrind*. The output can be graphically visualized using *kcachegrind*.

An important tool for our experiments was SSIAT [2] which is an application that automatically modifies the source code of SimpleScalar by adding new instructions, functional units and register files to the base architecture. SSIAT has a configuration file that contains all of the parameters of the new instructions, functional units and register files. This file contains most of the architectural configuration data needed by our simulator.

Most of the micro-architectural details of the simulation parameters are passed to the SimpleScalar by directly editing the simulator's source code through a bash script. Each simulation iteration consists of the following phases:

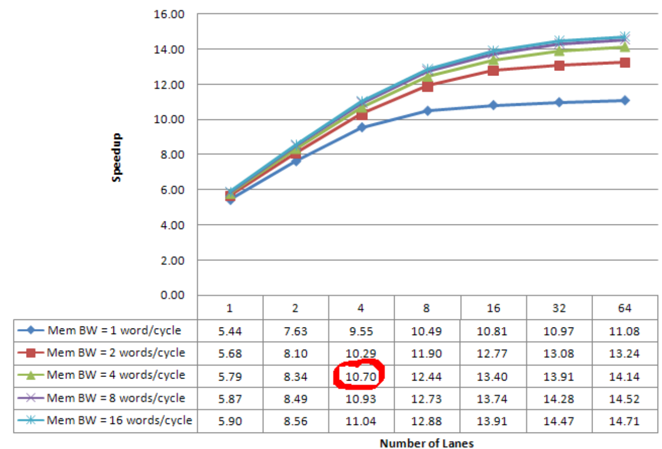
- Run SSIAT to modify architectural parameters of SimpleScalar
- Update the micro-architectural parameters
- Recompile the simulator
- Run the application through the simulator
- Log results

IV. Experimental results

To evaluate the impact of the memory bandwidth, the memory latency, the length of the vector registers and the number of lanes in the datapath, we have analyzed a free version of the MPEG2 encoder and decoder [1]. Through profiling, we have identified the computation intensive kernels and after inspecting the code for vectorization potentials, we have decided which functions to vectorize. After the design parameters have been determined and evaluated, we explore their impact on the actual kernel performance, and the overall application speedup.

A. *Design Parameters Evaluation*

We start by exploring the possible configurations of our customizable vector unit by focusing on four parameters: the number of lanes in the datapath, the vector memory bandwidth, the section size (vector register length) and the vector memory latency.



	1	2	4	8	16	32	64
Mem BW = 1 word/cycle	5.44	7.63	9.55	10.49	10.81	10.97	11.08
Mem BW = 2 words/cycle	5.68	8.10	10.29	11.90	12.77	13.08	13.24
Mem BW = 4 words/cycle	5.79	8.34	10.70	12.44	13.40	13.91	14.14
Mem BW = 8 words/cycle	5.87	8.49	10.93	12.73	13.74	14.28	14.52
Mem BW = 16 words/cycle	5.90	8.56	11.04	12.88	13.91	14.47	14.71

Fig. 2. Speedup vs. the memory bandwidth and the number of lanes

Number of lanes: Figure 2 plots the speedup of kernel *qna* for fixed values of the section size (16 elements) and memory latency (20 cycles) and various values of the memory bandwidth and the number of lanes in the datapath.

Discussion: The number of lanes influences the number of cycles required to complete computation instructions. Multiple lanes improve performance

over the single lane organization in a similar way as processing vectors over processing scalars. By adding more lanes the computation time is reduced linearly with the number of lanes, but the instruction and functional unit startup latencies remain constant. However, for two, four or eight lanes the computation time is significantly larger than the startup latency and we can consider that performance increases by a factor of 2, 4 and 8 respectively.

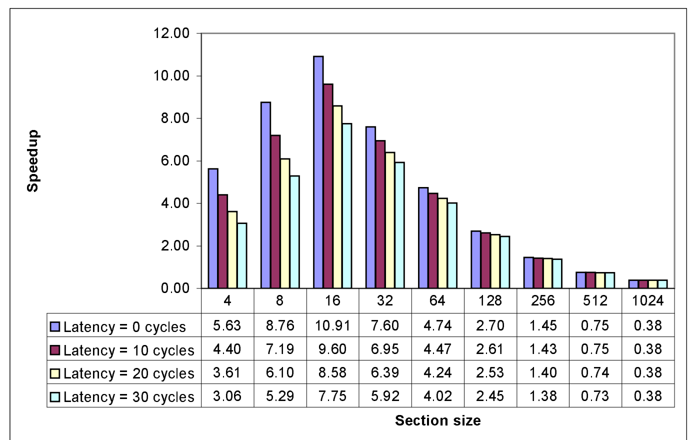
Memory bandwidth: Dictates the number of cycles required to complete memory transfer instructions. A multi-lane datapath needs a fast memory system to transfer data in and out of the processor. Our results suggest that the memory bandwidth must be coherent with the processing speed of the datapath. There has to be a balance between these the number of lanes and the memory bandwidth since in the average case there is no use in having a memory system that can transfer data much faster than it is processed or having a datapath processing data much faster than the memory system can transfer.

Discussion: For a fixed memory bandwidth, once there are enough lanes (enough processing speed) to take advantage of the full potential of the memory bandwidth, further increasing the number of lanes does not bring any noticeable improvements in execution time. By looking at the problem the other way around, by fixing the number of lanes and varying the memory bandwidth, we obtain a similar behavior. Memory bandwidth and the number of lanes in the datapath have similar impact on vector performance: the former influencing memory transfer instructions, see (2) and (3), the latter affects the computation instructions, see (1).

Having more lanes and higher memory bandwidth increases performance. However, there is a break-even point when performance gains are outweighed by the increase in area and design complexity. From Figure 2 we can observe that the best cost per performance is achieved using a configuration with 4 lanes and a memory bandwidth of 4 words/cycle/memory unit. From this point on, all our simulations have these parameters fixed to the previously mentioned values.

Section size (vector register lengt): Consider Figure 3 that shows the speedup obtained for kernel *component_prediction* for various section sizes and memory latencies (kernel *component_prediction* performs computations on vectors 16 elements long).

Discussion: By increasing the section size, the speedup also increases but only up to a certain point.



	4	8	16	32	64	128	256	512	1024
Latency = 0 cycles	5.63	8.76	10.91	7.60	4.74	2.70	1.45	0.75	0.38
Latency = 10 cycles	4.40	7.19	9.60	6.95	4.47	2.61	1.43	0.75	0.38
Latency = 20 cycles	3.61	6.10	8.58	6.39	4.24	2.53	1.40	0.74	0.38
Latency = 30 cycles	3.06	5.29	7.75	5.92	4.02	2.45	1.38	0.73	0.38

Fig. 3. Speedup vs. the memory latency and the section size

If the vector register length is larger than the application vector length, the performance actually degrades because the execution time of a vector instruction linearly depends on the section size, and by increasing the section size, we actually increase the execution time by processing unneeded elements. This degradation in performance can be alleviated through microarchitectural support to end instructions after all the useful elements have been processed. It is important to note that increasing the section size beyond a certain limit does not bring any benefits. This is because only the vectorized part of the application runs faster, there is a break even point after which the scalar instructions dominate the execution time. Thus, even for infinite vector lengths, there is this point when doubling the section size increases chip area much more then it increases performance.

Memory latency: Figure 3 also illustrates the relation between the memory latency and speedup.

Discussion: It can be observed that the larger the section size, the fewer vector instructions we execute. By reducing the number of vector instructions, we reduce the amount of times that instruction startup latency is encountered and so, we reduce the total execution time. Looking again at Figure 3, we can see that memory latency can significantly degrade vector performance especially for short section sizes. As the section size grows, the impact of memory latency is becoming less of an issue. This explains why vector processors typically have best performance for long vectors.

B. Speedups

After profiling and inspecting the code for vectorization potentials, for the MPEG2 encoder we have

selected functions $dist1$, $fdct$ and $quant_non_intra$ that occupy more than 85% of the execution time. For the decoder, the functions $component_prediction$ and $Clear_Block$ account for almost 30% of the computation. These kernels contain loops that are regular enough to have potential for high vectorization rates.

Kernel speedup: Table I shows the aggregated information for the vectorized kernels of both the MPEG2 encoder and decoder. The second column contains the kernel name, the third column contains the percentage of the total computation time spent inside the function. Columns four to seven show the kernel speedups for the vector register length of 8, 16, 32 and 64 elements respectively.

TABLE I
MPEG2: Comparative kernel speedups for section size of 8, 16, 32 and 64 elements

	kernel	time	Speedup			
Elements			8	16	32	64
Encoder	dist1	73.0%	2.13	3.06	2.27	1.50
	fdct	7.84%	2.42	2.22	1.90	1.48
	qna	4.8%	7.26	10.20	12.56	14.84
Decoder	comp_pred	20.75%	6.10	8.58	6.39	4.24
	clr_blk	8.01%	6.03	9.14	11.29	10.58

From this Table we can estimate the influence of each kernel over the entire application. Each kernel has different vector lengths and for a fixed section size, some perform better than others. For instance, the MPEG2 Encoder contains kernels $dist1$, $fdct$ and qna that have vector lengths of 16, 8 and 64 elements respectively. Thus, $dist1$ will perform best for vector registers of 16 elements while for vector registers of 64 elements will perform poorly, $fdct$ performs best for vector registers of 8 elements. On the other hand, kernels have varying degrees of parallelism and amount of computation which result in wide spectrum of speedups. For instance, kernel $dist1$ performs computation on vectors that have a small degree of dependencies and through vectorization we could obtain speedups of only 3x. In contrast, for kernel qna we have obtained speedups of almost 15x. This is because kernel qna performers heavy computations and has a high degree of parallelism. The overall application speedup is dictated by the individual kernel speedups weighted by the percent of time spent inside each kernel.

Overall application speedup: Table II presents the overall application speedup for various vector register lengths (section size). Vectorization level does not mean the percent of vector instructions from all executed instructions but instead, is the cumulative execution time of the considered scalar kernels of the scalar application execution time. Note that the number of vector instruction is dependent on the section size. For instance, for the scalar version of the MPEG2 Encoder, functions $dist1$, $fdct$ and qna account for 85.64% of the execution time. Only these three functions could be efficiently vectorized. The maximum theoretical speedup is considered for infinite kernel speedup. Of course this value can never be reached but is presented here as the upper bound of the achievable speedup. We can estimate the efficiency of our work by comparing our results to this upper bound.

TABLE II
MPEG2: Overall application speedup

	Vectorization level (relative to execution time)	Maximum theoretical speedup (for kernel speedup of infinity)	Obtained speedup for section size of		
			16	32	64
Encoder	85.64%	6.9	1.82	1.63	1.33
Decoder	28.76%	1.40	1.25	1.37	1.23

Aside from simulating each kernel independently, we have also simulated the whole vectorized applications consisting of both the vectorized kernels and the scalar part that could not be vectorized. The speedup for the MPEG2 Encoder is determined mostly by the behavior of the $dist1$ kernel since it takes 73.0% of all the scalar execution time. This kernel has vector lengths of 16 elements therefore, the whole application performs best for section size of 16 elements. Unfortunately, $dist1$ can offer only a speedup of 3x which explain the overall application low speedup compared to the maximum theoretical achievable speedup. Even though kernel qna reaches speedups of up to almost 15x, it only accounts for less than 5% of the computation time thus, it hardly influences the overall application speedup.

For the MPEG2 Decoder, the kernels that could be vectorized take only 28.76% of the execution time thus the highest possible speedup is 1.40x. The considered kernels, $comp_prediction$ and $clear_block$ offer good speedups, on average 8x and 10x respectively and have vector lengths of 64 and 16 respectively. Also in this case when one of the kernels has best performance with respect to the section size, the other kernel offers poor speedups. However, because the two kernels take less than 30% of all the application execution time the speedup obtained through vectorization is not spectacular but still is quite close to the theoretical upper bound.

V. Conclusions

We evaluated the impact on performance of the vector register length, memory latency, memory bandwidth and the number of parallel lanes in the datapath. For this purpose we modified SimpleScalar 3.0d by adding new vector instructions, a vector register file and simulated the MPEG2 encoder and decoder. Our experiments suggest that vector memory latency can seriously degrade performance especially for short vector registers. Long vector registers amortize the latency over many elements. However, the section size should be smaller than the average application vector lengths. We derived theoretically and verified experimentally the relations among the section size, the memory latency, memory bandwidth and the number of parallel lanes in the datapath. Based on our experiments and analysis, we conclude that the optimal vector processor organization is a multi-lane vector unit tailored for short vectors and low memory latencies.

In **future research**, we plan to develop a more accurate vector processor simulator that can model more precisely the architectural and microarchitectural parameters. More specifically, we wish to test a new architecture of the vector register file that will allow a significant increase in flexibility through reconfigurability. Another important future goal is to extensively investigate the vector memory system and ways to adapt caches to vector memory access patterns.

VI. Acknowledgements

This work was partially supported by the Dutch Ministry of Education, Culture and Science through the HSP Huygens Programme; the Dutch Technology Foundation STW, applied science division of NWO, the Technology Program of the Dutch Ministry of Economic Affairs (project DCS.7533); and the European Commission in the context of the Scalable computer ARChitectures (SARC) integrated project #27648 (FP6).

References

[1] Mpeg org webpage http://www.mpeg.org/mpeg/video/mssg-free-mpeg-software.html.

[2] Roel J. Meeuws Gerard Th. Aalbers B.H.H. (Ben) Juurlink, Demid Borodin and Hugo Leisink. The simples-calar instruction tool (ssit) and the simplescalar architecture tool (ssat).

[3] Roger Espasa and Mateo Valero. Multithreaded vector architectures. *Proceedings of the 3rd International Conference on High Performance Computing*, 1997.

[4] David A. Patterson John L. Hennessy. *Computer Architecture - A Quantitative Approach*. Morgan Kaufman Publishers, 2002.

[5] R.B. Lee. Multimedia extensions for general-purpose processors. *Signal Processing Systems, 1997. SIPS 97 - Design and Implementation., 1997 IEEE Workshop on*, 1997.

[6] S. Bolliger M. Day M.N. Pham, D. Asano. The design and implementation of a first-generation cell processor. *Solid-State Circuits Conference, 2005. Digest of Technical Papers. ISSCC. 2005 IEEE International*, 2005.

[7] Richard M. Russell. *Readings in computer architecture*. Morgan Kaufman Publishers, 2000.