# X00TIC magazine

July 2008-Volume 13-Number 1

POST-MASTERS PROGRAMME SOFTWARE TECHNOLOGY

## Designing Deeply Embedded Systems

TASS software professionals

PTS

TOPIC EMBEDDED SYSTEMS EMBEDDED IN YOUR FUTURE

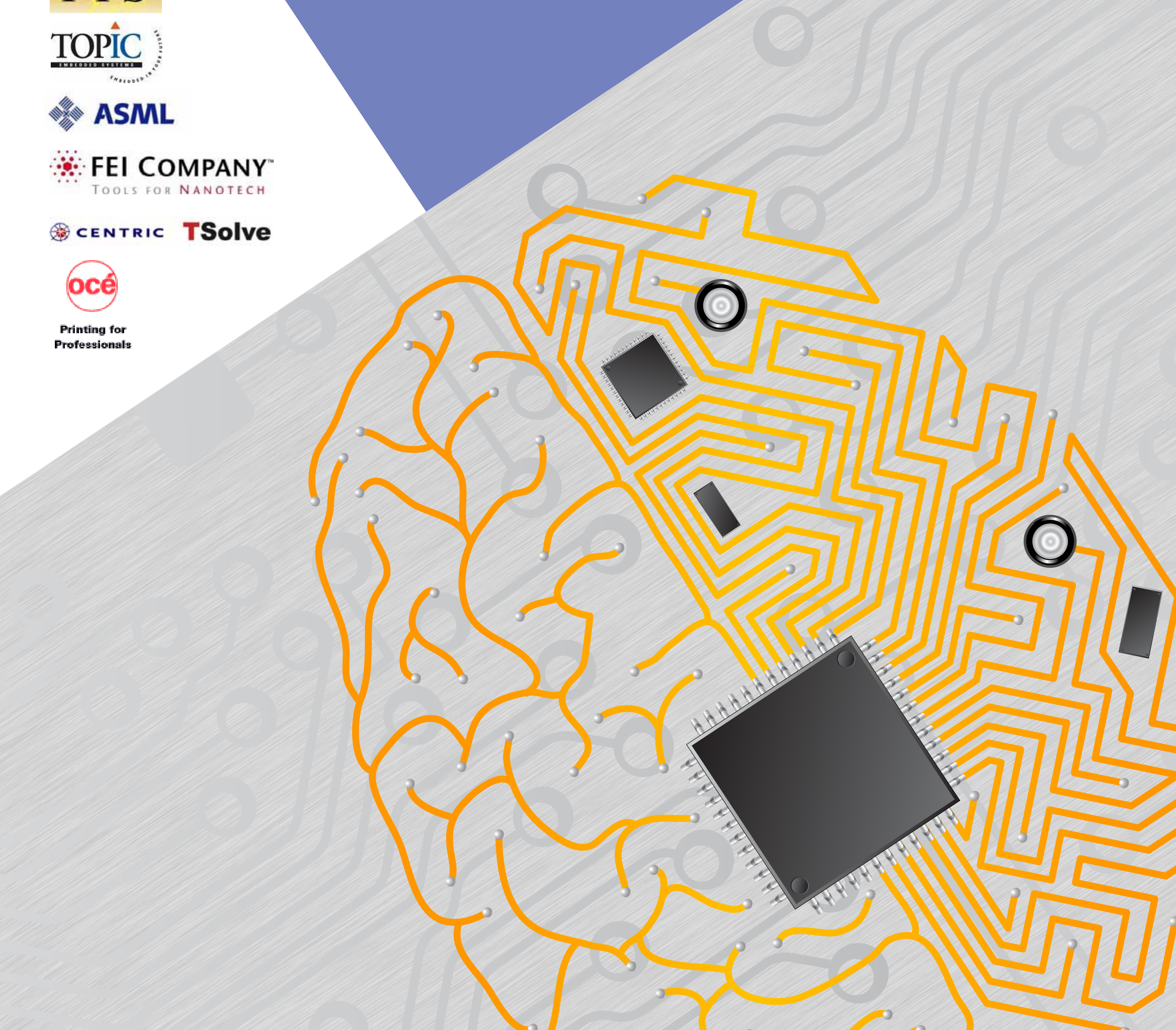
ASML

FEI COMPANY™ TOOLS FOR NANOTECH

CENTRIC

TSolve

océ Printing for Professionals



## Contents

XOOTIC
POST-MASTERS PROGRAMME SOFTWARE TECHNOLOGY

## Colofon

XOOTIC MAGAZINE
Volume 13, Number 1
July 2008

*Editors*
Yanja Dajsuren
Panagiotis Georgiadis
Jorn Bakker
Chris Delnooz
Menno Lindwer

*Address*
XOOTIC and XOOTIC MAGAZINE
P.O. Box 6122
5600 HC Eindhoven
The Netherlands
board@xootic.nl
`http://www.xootic.nl`

*Secretariat* OOTI
Technische Universiteit Eindhoven
Department of Mathematics and Computer Science
c.o. Ms. M. de Wert
HG 6.57, P.O. Box 513
5600 MB Eindhoven
The Netherlands
ooti@tue.nl
`http://wwwooti.win.tue.nl/`

*Print production*
WENS Creatie, Son en Breugel

# Designing Deeply Embedded Systems

editorial

In front of you, you find a new issue of XOOTIC Magazine. It has taken us one-and-a-half years to present you with a new issue. Many thought the Magazine was dead. However, nothing could be further from the truth. Your new Magazine Committee is still a bit rusty and experienced some booting issues on this release. But the first egg is out and we are sure that many nice ones will follow!

This issue is about a topic that is currently in the spotlight for several reasons. More than ever, tiny devices with lots of functions are becoming part of everybody's lives. By now, we all carry around mobile phones with integrated cameras. Many of those phones can act as navigation systems. However, a much more extreme example was announced June 13 this year: a medical eye implant of one qubic millimeter, containing a real system-on-chip, with a real functioning microcontroller [1]! These systems are becoming completely programmable. Contrary to the desktop computing community, the deeply embedded computing community has known for years that future power, area, and cost requirements can only be met if systems are designed to fit closely to the application domain and make sufficient use of parallelism and multi-core concepts. This issue of XOOTIC Magazine explores these concepts from a number of different angles.

The article by Jeroen Leijten and Lex Augusteijn gives an in-depth analysis of the bottlenecks in traditional system design. They also show solutions to designing such systems to be very efficient, yet managable targets for software development. Steve Leibson subsequently describes how such systems could be programmed, with a particular focus on super pipelining. Despite the efforts of Leijten, Augusteijn, and Leibson, in the 2007 Workshop, the OOTIs experienced that the design tools for these systems are far from ideal. Dorieke Schipper writes about how they harmonized a toolflow to take designs from top-level ANSI-C code all the way down to a ready-to-go FPGA implementation. Last but not least, Andreas Hansson and Benny Åkesson show how one can develop composable software for multi-core embedded systems.

When you are reading this, your Magazine Committee is already working on the next issue. By that time, we will be celebrating OOTI's 20th birthday. Obviously, that Magazine will look back on those 20 years.

Remains for us to wish you happy reading!

XOOTIC Magazine Committee

*Yanja Dajsuren*

*Georg Panagiotis*

*Jorn Bakker*

*Chris Delnooz*

*Menno Lindwer*

## References

[1] Center for Wireless Integrated MicroSystems *Annual Report 2007*
www.wimserc.org/annual_report/circuits_AR07.pdf

# Morgen kunnen we 10-nm-chips maken. Vandaag mag jij bedenken hoe.

Bij de chipproductie werd tot nog toe Deep UV-licht gebruikt (193 nm). Om kleinere chips mogelijk te maken, werkt ASML nu aan de toepassing van Extreem UV-licht (13.5 nm).

Een systeem van magnetisch gestuurde spiegels in een vacuüm 'kneedt' het EUV tot een constante bundel, sterk genoeg om het silicium te belichten, voor de chipproductie van 10-nm-structuren.

In het vacuüm zijn vluchtige koolwaterstofmoleculen aanwezig. EUV-fotonen slaan deze organische moleculen uiteen.

13.5 nm
EUV-fotonen
-1%

ASML zoekt naar oplossingen om het spiegelsysteem schoon te houden. Schoonvegen behoort niet tot de mogelijkheden.

Door het neerslaan van vrije koolstof-atomen (0.5 nm) op de spiegels in het vacuümsysteem wordt de reflectie verminderd.

## Voor engineers die vooruitdenken

Profiel: Wereldwijd marktleider in chip-lithografiesystemen | Marktaandeel: 65% | R&D-budget: 500 miljoen euro | Kansen voor: Fysici, Chemici, Software Engineers, Elektrotechnici, Mechatronici en Werktuigbouwkundigen | Ontdek: ASML.com/careers

ASML

# Designing and Programming Efficient Embedded Systems-on-Chip

Jeroen Leijten, Lex Augusteijn

Silicon Hive

*In this paper, we give an overview of the issues playing in embedded multi-processor designs. We can roughly divide this space into three categories: homogeneous multi-processors, heterogeneous multiprocessors,and fixed-function devices. The homogeneous multi-processors have been derived from desktop designs, containing multiple high-speed general-purpose processor cores. Heterogeneous multi-processors have their functions mapped onto sets of specially designed processor cores. Fixed-function devices are characterized in that most of the actual processing is performed by weakly programmable dedicated hardware blocks.*

## Introduction

The point of the whole exercise is to move away from fixed-function devices, in which all functionality is cast in hard-wired blocks. Until recently, these types of devices could not be rivaled in terms of efficiency. However, they are inflexible, very expensive to design, have a very limited application scope, and a short commercial lifetime. Thus, the world is looking for programmable solutions and turns to the software community for answers.

Because the software community tends to focus on solving performance issues using limited numbers of homogeneous high-speed general purpose processors, we will investigate some of the issues around this type of systems. We will discuss how these issues can be overcome with application-specific processor designs. But, in order to really compete with the fixed-function devices, the heterogeneous applicationspecific multi-processors have to break away from the classical DSP or enhanced-RISC approach.

Application-specific multi-processor architectures require some specific tooling and skills to program, which we discuss as well. In particular, there is a need for optimizing compilers, which can perform instructionselection for different instruction-sets with combinations of user-defined operations. Compilers have to extract parallelism from sequential code. In multicore systems, the compiler needs to find its way in compiling for the whole architecture. Design space exploration tooling must be able to reason about all aspects of the architecture, in order to assist in finding the right parameter settings. Last but not least, since the actual SoCs often are still being processed, most software development takes place on simulators. These simulators must simulate complete systems sufficiently fast, yet provide enough accuracy to allow both software and hardware designers to establish functional correctness and timing behaviour.

In the past, a chip had a single function. It could be a memory chip, a sound chip, or even a microprocessor. Together, these chips formed a system. Nowadays, most chips contain complete systems, performing large sets of functions. These chips are thus referred to as Systems-on-Chip (SoCs).

The Intel Core Duo T2500 processors [10] are positioned for the embedded computing market. They are single chips, consisting of two processor cores, each with their own cache, an on-chip bus, and two levels of shared cache. The chips have various interfacing logic and power management units. As such, they are real SoCs.

The processor cores on the T2500 run at a maximum frequency of 2GHz. Having some parallel computing capability within each processor, the chip can calculate an absolute maximum of 24 GOPS (Giga-Operations Per Second), assuming that the software manages to keep all computational elements busy at all times. At this speed, the T2500 consumes 31 Watts.

In this article, we will compare architectures on the power they consume per actually executed (effective) byte-operation. This metric is expressed in Joules-per-operation. Given that, during regular benchmarks, a T2500 is utilized for about 70%, it consumes about 2 nano-Joules per operation.

So much for raw compute power. However, even if the application software could be scheduled such that all execution units would be kept busy, it is unlikely that they can fetch their data fast enough. The on-chip memory (L2 cache) is shared between the two cores and produces a maximum data stream of only 2.7 GB/s (Giga-Bytes per second).

Compared to that, the compute and data requirements for full HD (1080p) high-definition video decoding are roughly two times higher: 60GOP/s, 4 GB/s [1]. To fit this in a package that does not require a fan, the power consumption needs to be below 1 Watt. In the wireless world, the requirements are even stricter. A typical mobile phone battery pack delivers about 5Wh. Most of that is already spent on the display, signal reception, and control operations. In order to watch a whole movie, without the battery running out, the video processing may consume a few 100 mW. This boils down to energy consumption per operation between 2 and 20 pico-Joule.

Additionally, the embedded world needs to deal with strict requirements on form factor (size) and cost. At $322, a typical Intel T2500 would cost more than a whole mobile phone [2]. An image sensor SoC for highend mobile phones (e.g. [14]) with a 16 GOP/s image programmable processor [17] costs about $3 to $6. Similar figures hold in application domains such as mobile and wireless LAN communication standards. Security and automotive systems are moving in the same direction.

In order to rival hard-wired solutions, it follows that the Systems-on-Chip for these types of embedded functions need to be roughly a factor 100 to 1000 more efficient, both in terms of efficiency (GOP/s) and cost, than current-day practice in high-end general purpose embedded devices.

Now comes a little bit of philosophy: parallelism was pioneered in the ’70s by research institutes and companies such as Cray, Inc. Supercomputers. These systems were known for their huge size and even larger power bill. On the other hand, using ever-increasing clock speeds of a single processor, Intel is known for bringing ’80s mainframe compute power to the form factor (and power consumption) of a home sound system. Thus, parallel computing technology is not associated with efficiency. In fact, it seems counterintuitive to make calculations more efficient by making them happen in parallel. Why would it make a difference whether calculations happen in parallel or sequentially? The power for doing them will be consumed anyway. In fact, when you do them in parallel, you also need to spend power on communicating the results, right?

Wrong!

A fully programmable 30-core digital video receiver was produced in 2004 [7], [12]. This SoC had a maximum computational throughput of 60 GOP/s, consuming less than 1 Watt (i.e. about 16 pico-Joule per operation). As of April 2005, Intel is advocating the use of multi-core processors, because they are more power-efficient [11].

## So what's wrong with the traditional high-performance processor?

In general, traditional high-performance processors spend the majority of their chip real-estate (and thus also power consumption) on functions that have a secondary effect on the actual operation throughput.

There are roughly four reasons:

- Pushing technology to the limits means falling over the edge of diminishing return on chip real estate.
- Speculation is bad!
- In order to reach high clock speeds, hardware and power are increasingly spent on control, rather than on computation.
- Centralized resources are power and performance bottlenecks.

### Diminishing Return on Chip Real Estate

Chips are produced in so-called silicon processes. When pushing the clock speed of a processor significantly higher than the 'sweet spot' for the fabrication process, chip area and power consumption grow faster than linear. In order to make the T2500 operate at 2GHz, Intel had to take the following measures:

- Specify the operating voltage to be 1.3V [10], rather than 0.8V (customary for a 45 nm process). This increases power by a factor of 2.6.
- In order to support higher voltages and higher currents, the physical layout was pushed to produce fast-switching and strong wires, repeaters, and buffers.
- Push the construction methodology (RTL synthesis) to produce fast, but large, logical circuitry, with additional bypasses and gates.

#### Why is speculation bad?

Speculation boils down to using chip real estate for functions or operations of which you are not sure that you will need them. And if there is a chance that operations or stored data are eventually not used for the end result, a corresponding percentage of the power and area that is spent on them, is wasted.

Speculation is applied in several architectural forms in high-clocked processors. These are some examples:

- Caches speculate on once-fetched data being needed again. Storing data in a cache means storing the data twice, having to do expensive cache-lookups on every fetch, and having to keep several layers of caches synchronized. Thus, caches are far less efficient than having a simple local memory which is pre-loaded deterministically, to hold exactly that code that needs to be executed.
- Pipelined processors are processors in which the actual operations have been segmented into many small sub-steps, such as instruction fetch, decode, instruction reordering, etc (exceeding 10 stages on some CPUs). Deeper pipelines mean smaller and faster pipeline stages, thus increase in clock frequency. However, on conditional jumps, a pipelined processor will have prefetched many instructions, which are then wasted. Actually, on average real-life code, 20% of all instructions is a jump (see e.g. [6]). So, on deeply pipelined processor cores, pipeline flushes occur more often than not.
- In order to reduce performance loss because of jump mis-prediction, processors employ branch prediction tables. However, the tables act as caches in speculating that these branches will be visited again.
- Out-of-order execution is a process by which the processor itself analyzes incoming sequences of instructions to determine which ones can run in parallel. The processor's execution pipelines are replicated and parallel operations are distributed among them. But, often the calculations are actually executed speculatively, because it is not known in advance whether the results will actually be needed.

According to [8], 9% of the area of a typical X86 processor is spent on the integer and floating point processing units and 50% (!) is spent on the caches.

### What centralized resources?

Figure 1 (taken from [9]) illustrates the centralized resources in a processor that is similar to the cores within an Intel T2500. The Instruction Control Unit is connected to almost every other block in the design. The FPU Register File needs to feed three parallel floating point execution units. What's not depicted is that, similarly, the integer register file has to feed the three Integer Execution Units (IEUs). The Integer Scheduler has to analyze and feed instructions to three parallel IEUs. What is also very important is that all these units have to read and write their data through a single Load/Store Queue Unit.

When a single unit has to drive multiple other units spread over a large device, wiring becomes a problem and the design is not scalable. Also, a register file having to feed multiple parallel calculation units has to have at least as many read and write ports. Each additional read/write port adds to the size of the register file. This is roughly a logarithmic function.

It's difficult to quantify the impact of such centralized resources on the area and power efficiency of a design, without being able to analyze power simulation results from individual blocks within the design.

## So how to design an efficient multi-processor system?

In order to relieve the above issues, processor designers have made several improvements. For example, traditional embedded RISC processors are already much more efficient than desktop processors. Embedded RISC processors consume around 200 pJ/operation (90 nm silicon fabrication process). For a long time now, designers have turned to Digital Signal Processors (DSPs) which are typically more efficient than RISC processors on convolution-based algorithms. On these types of algorithms, DSPs consume 50 to 100 pJ/operation. Some processor designers went further, adding more application-specific operations to their processors (calling them "configurable processors"). This way, some managed to bring power consumption down to about 20 pJ/operation.

However, the fundamental flaws that we discussed before, were not dealt with. Even configurable processors contain relatively deep pipelines (5 to 7 stages), centralized resources, inflexible I/O sub-systems, and relatively high bandwidth requirements. In order to really compete with fixed-function hardware, power consumption has to drop by another factor of 10, as we mentioned in the introduction. This means that all resources have become fully scalable, speculation has to be fully removed, and all resources must be spent on calculation, rather than control.

Note that we can not wait for silicon process technology (Moore's law) to help out, because processing requirements increase even more rapidly. For example, we are already confronted with design requirements for Quad-HD video at 120 frames/second (i.e. requiring 16x performance over the 60 GOP/s calculated three years ago in [1]).

Basically, this boils down to four things: (i) inverting the steps that have been taken in the past to build very generic processors at very high clock speeds, (ii) remove all overhead by having the compiler schedule all resources, (iii) scale all architectural components, such that parallelism can be optimally exploited, and (iv) make system-level decisions visible to all tools, including the compiler, such that these tools can reason about all kinds off trade-offs.

### Increasing Efficiency of Processor Cores

- When more performance is needed, rather than increasing clock speed, keep the processor in the silicon process' sweet spot by adding more parallel compute resources, without increasing dependency on central resources.
- Rather than spending chip area and power on speculation, schedule all operations

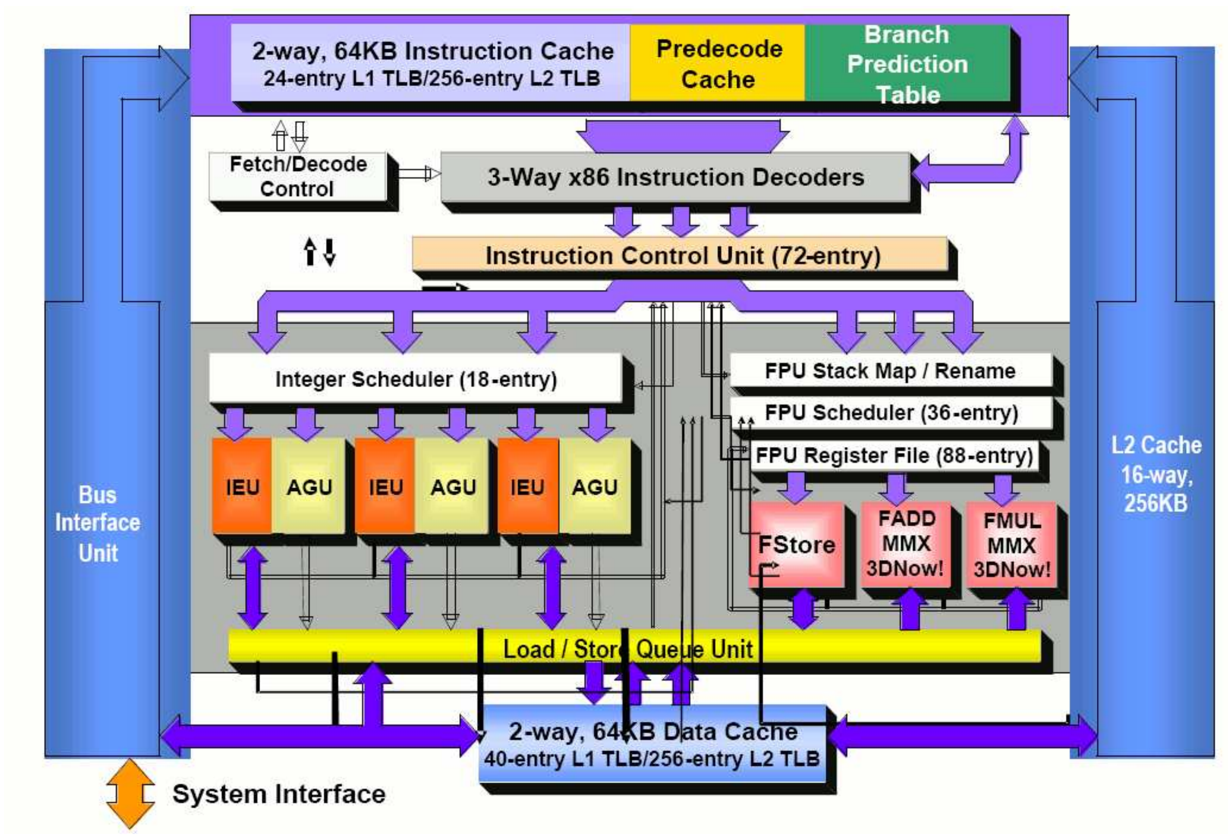


Figure 1: AMD Athlon Microarchitecture Block Diagram

and memory allocation statically, replace caches by scratch pad memories, reduce pipeline depth, and remove out-of-order execution.

- Maximize the use of locality of reference. De-centralize resources, by having multiple controllers (and thus multiple processor cores), multiple scratch pad memories per core, and multiple register files per core. This way, wires are kept short. Also thin wires and lean drivers can be instantiated, because they need to drive fewer modules over shorter distances.
- There is no one-size-fits-all solution. Processors will have to be designed specifically for each application domain. Processors will have application-specific operations.

So let's say we reduce the processing area to exactly what's needed for the actual calculations, by removing speculation. And suppose we can also convert caches into regular memories of the exact size needed for the application, then through the analyses of the previous chapter, it's easy to see that SoC real-estate can be reduced by several factors. Add to that the benefits of running at a low clock speed (at least a factor 3 less power consumption), we can see the path to obtaining efficient SoCs.

Actually, all these measures make processor design become much more easy. Processor designers no longer have to think about how to implement complicated cache eviction strategies, organize out-of-order execution, design bypass networks, and design arbitration for centralized resources. Since these features do not add to the overall functionality of the processor, they are very hard to verify. Thus, removing them makes processor verification much easier.

In fact, processors become *composable*. Their complexity no longer increases with size. In traditional processor design (and also in traditional DSP and application-specific accelerator design), centralized controllers and register files prevented replication of compute resources. When these bottlenecks are removed, parallel replication of compute units becomes trivial [13].

### Increasing Efficiency at the System Level

Already for a long time, it's acknowledged that bandwidth between cores in an SoC (be it processors, hardwired accelerators, or memories) is becoming a bottleneck. Additionally, with increasing numbers of transistors, the relative distance between SoC modules becomes larger. This means that on-chip signals need more than a single clock cycle to travel between SoC modules. Thus, they have to travel in multi-cycle hops, increasing the latency of data transfers.

Different kinds of solutions are being applied for these kinds of system-level issues. However, in systems with communicating heterogeneous processor cores, software tools and simulators need to be able to reason about such intra-core infrastructures. Therefore, the composability requirement holds for system-level interconnect as well.

Also, in contrast with the required heterogeneous structure of the processor cores, the interconnect infrastructure needs to be homogeneous. Else, tools and developers can not depend on deterministic behavior of the SoC.

Any system-level methodology needs to provide for deterministic interconnect architectures. On top of that the architecture needs to be composable and APIs and tools need to take latency, bandwidth and other parameters into account. Last but not least, the tooling needs to provide several levels of system simulation, such that designers and software developers can validate their assumptions.

## The Software Design Bottleneck

There was one very valid reason to achieving performance increase in the traditional way: the software design bottleneck. Functionality of today's ever more complex devices is expressed in software. Software development productivity had to increase. Software designers simply did not have the time to design their software for multitudes of processors, having to think about which type of processor core is best suited for a particular algorithm, having to split source data over multiple scratch pad memories, having to control communication and synchronization (message passing), having to determine in which register file data is to be stored, etc.

So, we can have the compilers for solving all these issues? Yes and no.

Some compilers can indeed schedule sequential code to take advantage of 10s of parallel computation units, and, at the same time determine which register files to use for which local data, which data lanes to use to exchange data between register files, etc. (e.g. [19]). Compilers have sophisticated data flow graph matching algorithms to automatically select operations with complicated functionalities. Additionally, software designers can make use of extended operator overloading mechanisms to keep their code portable and easy to understand.

The European ACOTES project [15] goes a step further and aims to automatically split sequential code in multi-processor code and to automatically extract vector parallelism from sequential code.

But, even though compilers get ever more powerful, the system designer still has a responsibility to design a balanced system that programmers can grasp. The system's features have to match the compute requirements of the application domain. Next to that, the software designer will have to have intimate knowledge of the system architecture. Using that knowledge, the software designer maps tasks onto processor cores, maps data onto scratch pad memories, and often also selects appropriate application-specific operations.

However, all of this only works if composability of architectures and applications is preserved at all levels of the system design. The whole system has to be built on a template that allows tools to reason about tradeoffs between different mappings. This means that SoC design can not be viewed only from the angle of instantiating multiple RISC cores and subsequently adding dedicated blocks for all sorts of functionality and on-chip communication.

As an example, we can compare HiveFlex VSP 2500 [16] with ARC 417V [4]. The former has a regular structure. For increasing compute requirements, the ANSI-C programmable video-specific processors can simply be replicated. The latter one has a number of dedicated hardware blocks for specific video functions (e.g. entropy encode, entropy decode, motion estimation), which can not be scaled. Also the latter has irregular interconnect, which does not allow replication. But, most importantly, the compiler tool chain only provides support for the controller CPU [5]. All other engines have to be manually programmed or controlled.

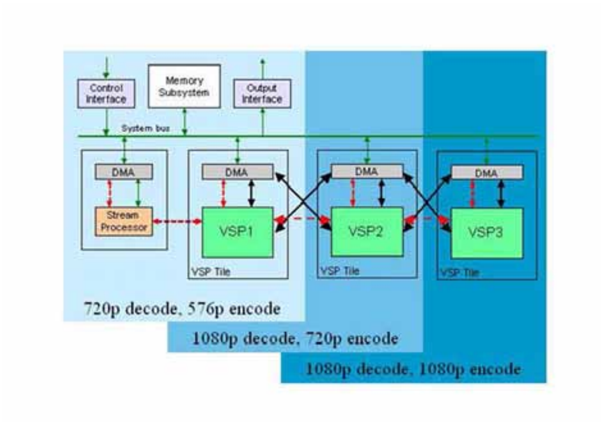


Figure 2: HiveFlex VSP, tiled video signal processor architecture

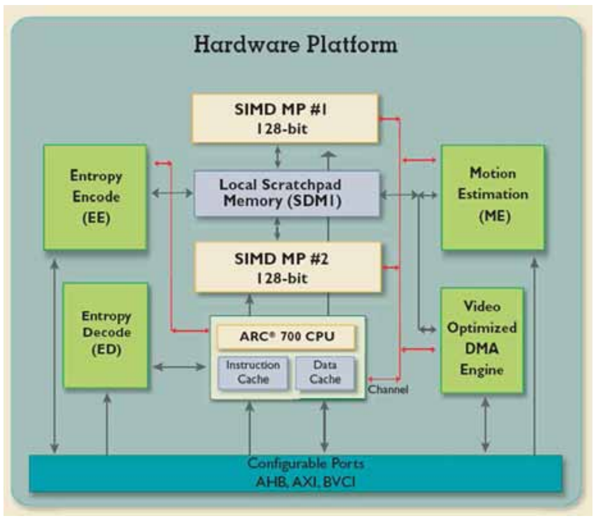


Figure 3: ARC AV417V video sub-system

## Conclusion

The world is turning to the software community to bring the solution to the ever-increasing cost of designing new SoCs for each application and for each new video, imaging, or communications standard.

However, in order to make programmable SoCs compete with their fixed-function counterparts, programmable solutions have to become a factor 100 to 1000 more efficient.

Some relief comes from configurable multi-core processors. Because of the application of multiple cores, parallelism and efficiency do increase. They provide some degree of application-specific functionality. However, generally this not enough to reach the required 100-fold efficiency increase and they are hard to program.

Composable multi-core processors have been shown to be good compiler targets and therefore relatively easy to program. Additionally, in order to really rival fixed-function devices, such composable architectures can be configured for high locality of reference, complete removal of centralized resources, shallow pipelines, scalable I/O, and distributed memories.

We have shown that the required composable architectures can be built today. Compilers and simulators for these architectures are also available. The ACOTES European project aims to improve these compilers further, in particular in regard to automatically scheduling over multi-core processors. However, the software architect has to understand the trade-offs and be trained to deal with multiple instruction sets, partitioning over sets of heterogeneous processors, and take bandwidth limitations and communication latencies into account.

## References

[1] Alvarez M, et al. *A Performance Characterization of High Definition Digital Video Decoding using H.264/AVC* Proc. IEEE Int'l Workload Characterization Symp., 6-8 Oct. 2005, pp. 24-33

[2] A-Power Online *Intel Core Duo T2500 2.0GHz 2M 667MHz FSB #1484*
www.apower.com/product-1484

[3] ARC International (www.arc.com)

[4] ARC International *Video 417V Subsystem*
www.arc.com/subsystems/video/AV417V_diagram.pdf

[5] ARC International *GNU Tools for ARC®Processors*
www.arc.com/software/development/gnutools.html

[6] Fritts, J.; Wolfe, W.; Liu, B. *Understanding multimedia application characteristics for designing programmable media processors* Dept. of Electrical Engineering, Princeton University

[7] Gruijters, P., et al. *Flexible Embedded Processors for Developing Multi-Standard OFDM Broadcast Receivers*
www.us.design-reuse.com/articles/8887/flexible-embedded-processors-fordeveloping-multi-standard-ofdm-broadcast-receivers.html

[8] Herring, C. *x86 Everywhere* MicroSoft WinHEC 2005
developer.amd.com/assets/WinHEC2005_x86_Everywhere.pdf

[9] Huynh, J. *The AMD Athlon$^{TM}$XP Processor with 512KB L2 Cache Technology and Performance Leadership for x86 Microprocessors*
Advanced Micro Devices, Inc. courses.ece.uiuc.edu/ece512/Papers/Athlon.pdf

[10] Intel *Intel Core Duo T2500 processors*
download.intel.com/design/intarch/prodbref/31127205.pdf

[11] Intel *Intel Multi-Core Technology* www.intel.com/multi-core/index.htm

[12] Leijten, J. *The Avispa Family of ULIW Parallel-Processing Cores for Multimedia and Communications* Spring Processor Forum, 25 April 2006

[13] Leijten, J.; Lindwer, M. *Multiprocessing Template for Media Applications* Proc. IEEE ISM, 2006

[14] MagnaChip *MC531B 1/3.2" 3.2MP eDoF SOC sensor*
www.magnachip.com/eng/download/MC531EB.pdf

[15] Munk, H.; et al. *Acotes; Advanced Compiler Technologies for Embedded Streaming*
www.hitech-projects.com/euprojects/ACOTES/

[16] Silicon Hive BV *HiveFlex VSP2500 Series*
www.siliconhive.com/Flex/Site/Page.aspx?PageID=9263

[17] Silicon Hive *HiveFlex ISP2200 Series, Camera Image Signal Processor*
www.siliconhive.com/Flex/Site/Page.aspx?PageID=8879

[18] Tensilica Inc. (www.tensilica.com)

[19] Turley, J. *Avispa+ Buzzes with Innovation; High-End Core Combines Decades of Competing Architectural Ideas* MicroProcessor Report, 9 April 2004

# Multicore/MPSOC Design and Convenient Concurrency

Steve Leibson

Tensilica

*For the first 25 years of their existence, microprocessors were designed to be as general-purpose as possible to widen its use across as many design projects as possible to increase sales volumes and to amortize each processor's design across many, many ICs. When each processor chip was hand designed by teams consisting of dozens or hundreds of engineers, such amortization was almost mandatory. The cost of generating mask sets and fabricating production volumes of such processors was also very large. Thus, for the first quarter-century of its existence, microprocessor architectural design focused on creating relatively complex, fixed-ISA (instruction-set architecture) machines that had a lot of features intended to appeal to the broadest possible design audience.*

When systems design began to migrate from the board level to the chip level, it was a natural and logical step to continue using fixed-ISA processor IP in SOCs and to continue to treat the processor as an expensive resource to be fully exploited up to its performance limits. Consequently, many system designers became versed in the selection and use of fixed-ISA processors and the related tool sets for their system designs and became accustomed to a single-processor design mentality. Thus, when looking for processor IP to use in an SOC design, system designers naturally turned to fixed-ISA processor cores and generally limited themselves to one processor per design, augmenting the processor with hardware to achieve performance goals. However, when custom silicon serves as the system substrate, designers are not limited to one fixed-ISA microprocessor core as they are with board-level systems based on discrete, pre-packaged microprocessors. Configurable processor cores allow system designers to tailor one or more microprocessor cores to more closely fit the intended application (or set of applications) on the SOC. A closer fit means that each processor's register set is sized appropriately for the intended task and that the processor's instructions closely fit the intended tasks as well. For example, a processor tailored for digital audio applications may need a set of 24-bit registers for the audio data and a set of specialized instructions that operate on 24-bit audio data using a minimum number of clock cycles.

Processor tailoring offers the SOC design team several practical benefits:

- Tailored instructions perform assigned tasks in fewer clock cycles.
- For real-time applications such as audio processing, the reduction in clock cycles directly lowers operating clock rates, which in turn cuts power dissipation.
- Lower power dissipation extends battery life for portable systems and reduces the system costs associated with cooling in all systems.
- Lower clock rates also allow the SOC to be

fabricated in slower and therefore less expensive IC-fabrication technologies.

Even though the technological barriers to freer ISA selection were torn down by the migration of systems to chip-level design, system-design habits are hard things to break. Many system designers who are well versed in comparing and evaluating fixed-ISA processors from various vendors elect to stay with the familiar, which is perceived as a conservative design approach. When faced with designing next-generation systems, these designers immediately start looking for processors with higher clock rates that are just fast enough to meet the new systems performance requirements. Then they start to worry about finding batteries or power supplies with extra capacity to handle the higher power dissipation that accompanies operating these processors at higher frequencies. They also start to worry about finding ways to remove the extra waste heat from the system package. In short, this design approach is not nearly as conservative as it is perceived; it is merely old fashioned.

## Processor Parallelism and "Convenient Concurrency" for SOC Designs

Many articles, conference papers, and general discussions of multicore or multiprocessor SOCs (MPSOCs) and associated programming models narrowly focus on particular homogeneous, multiple-processor architectures. This narrow focus severely limits the possible ways in which multiple computing resources can be used to attack a problem. Such discussions tend to focus on "embarrassingly parallel" problems such as graphics. Article and paper authors frequently declare that other big problems cannot be solved until there are tools that can automatically partition problems into processor-sized chunks. However, the truth is that many design problems are conveniently concurrent and are easy to attack with multiple processor cores, though not necessarily using an SMP (ed: Symmetrical Multi-Processor) architecture.

Expanding our architectural thinking beyond SMP multicores uncovers at least two kinds of concurrency –heterogeneous, not homogeneous– that easily exploit multiple processors. Many embedded systems exhibit such "convenient concurrency." The first such system architecture exists in many consumer devices including mobile phones, portable multimedia players, and multifunction devices. You might call this sort of parallelism "compositional concurrency," where various subsystems –each containing one or more processors optimized for a particular set of tasks– are woven together into a product. Communications are structured so that subsystems communicate only when needed. For example, a user-interface subsystem running on a controller may need to turn audio processing on or off, control the digital camera imaging functions, or interrupt video processing to stop, pause, or change the video stream. In this kind of concurrent system, many subsystems operate simultaneously. Yet they have been designed to interact at only a high level and do not clash.

Figures 1 and 2 are, respectively, block diagrams of a Personal Video Recorder (PVR) and a Super 3G mobile phone that illustrate this idea. Figure 1 shows seven identified processing blocks (shown in gray), each with a clearly defined task. Figure 2 shows 18 such processing blocks. In Figure 1, it's easy to see how one might use as many as seven processors (or more for sub-task processing) to divide and conquer the PVR design problem. Similarly, Figure 2 shows how as many as 18 processors might be employed on a Super 3G mobile phone chip.

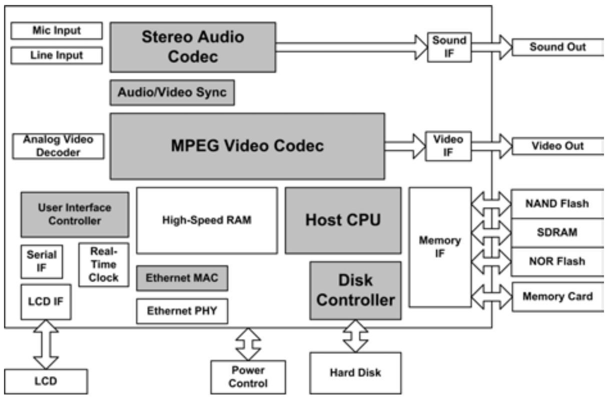


Figure 1: Personal Video Recorder Block Diagram

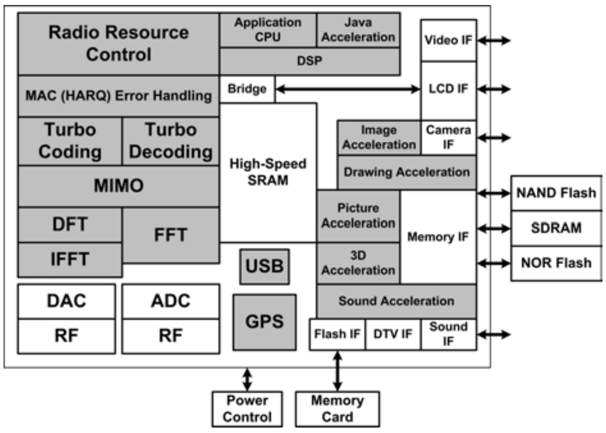


Figure 2: Super 3G Mobile Phone Handset Block Diagram

Some engineers might criticize this sort of architecture because of its theoretical inefficiency in terms of gate and processor count. Ten, twenty, or more processor cores could, at least in theory, be replaced with just a few general-purpose cores running at much higher clock rates. However, this criticism is misplaced. When processors were expensive, design styles that favored the use of few, big, fast processors held sway. With the end of Denard scaling (also called classical scaling) at 90nm (ed: IC fabrication feature size), transistors continue to get much smaller at each IC fabrication node but they no longer get that much faster and they no longer dissipate much less power. In fact, static leakage current has started to climb. As a result, the big processors' power dissipation and energy consumption have become unmanageable at high clock rates and system designers are now being forced to adopt design styles that reduce system clock rates before their chips burn to cinders under even normal operating conditions.

Compositionally concurrent system design offers tremendous system-level advantages:

- Distributing computing tasks over more on-chip processors trades transistors for clock rate, reducing overall system power and energy consumption. Given the continued progress of Moore's Law and the end of Denard scaling, this is a very good engineering trade off.
- Subsystems can be more easily powered down when not used –as opposed to keeping all the cores in a multicore SMP system running. Subsystems can be shut off completely and restarted quickly or they can be throttled back by using complex dynamic voltage and frequency scaling algorithms based on predicted task load.
- Because these subsystems are task-specific, they run more efficiently on application-specific instruction set processors (ASIPs), which are much more area and power efficient than general-purpose processors so the gate advantages of fewer general purpose cores may be much less than it seems on first consideration.
- Compositionally concurrent system designs avoid complex interactions and synchronizations between subsystems. Shutting down the camera subsystem on a compositional product is a trivial task to perform in software while making sure that such a task can safely be suspended in a cooperative, multitasking environment running on an SMP system can be significantly more complex. Proving that a 4-core SMP system running a mobile phone and its audio, video, and camera functions will not drop a 911 emergency call when other applications are running, or that low priority applications will be properly suspended when a high-priority task interrupts, is often a nightmare of analysis involving "death by simulation." Reasonably independent subsystems interacting at a high level are far easier to validate both individually and compositionally.

Pipelined dataflow, the second kind of concurrency, complements compositional concurrency. Computation often can be divided into a pipeline of individual task engines. Each task engine processes and then emits processed data blocks (frames, samples, etc.). Once a task completes, the processed data block passes to the next engine in the chain. Such asymmetric multiprocessing algorithms appear in many signal- and image-processing applications from cell-phone baseband processing to video and still-image processing. Pipelining permits substantial concurrent processing and also allows even sharper application of ASIP principles: each of the heterogeneous processors in the pipeline can be highly tuned to just one part of the task.

For example, Tensilica's Diamond Standard 388VDO Video Engine (Figure 3) mates two appropriately and differently configured 32-bit processor cores with a DMA controller to create a digital-video codec subsystem. One processor core in the subsystem is configured as a stream processor and the other as a pixel processor. The stream processor accelerates serial processing such as bitstream parsing, entropy decoding, and control functions. The pixel processor works on the video data plane and performs parallel computations on pixel data using a single instruction multiple data (SIMD) instruction architecture. Both processors have different local memory and data width configurations as required by their functional partition. This configuration decodes H.264 D1 main profile video while running at 200 MHz, which easily achieved with 130nm technology and is even easier to fabricate with more advanced IC fabrication processes.

A Pentium-class processor decodes H.264 D1 main profile video running at a clock rate of between 1 and 2 GHz while dissipating several tens of Watts. A paper presented at the recent ICCE (International Conference on Consumer Electronics) discussed decoding H.264 D1 main profile video using 125% of a 600MHz TI TMS320DM642 DSP, putting the required clock rate at 720 MHz. Unfortunately, you cannot synthesize SOC processors that run at 720 MHz –much less 1 to 2 GHz-using available ASIC foundry technologies. In this case, pipeline processing drops the required clock frequency considerably over the "one big, fast processor" design approach and allows the video decoder to be fabricated in a conventional ASIC manufacturing technology.

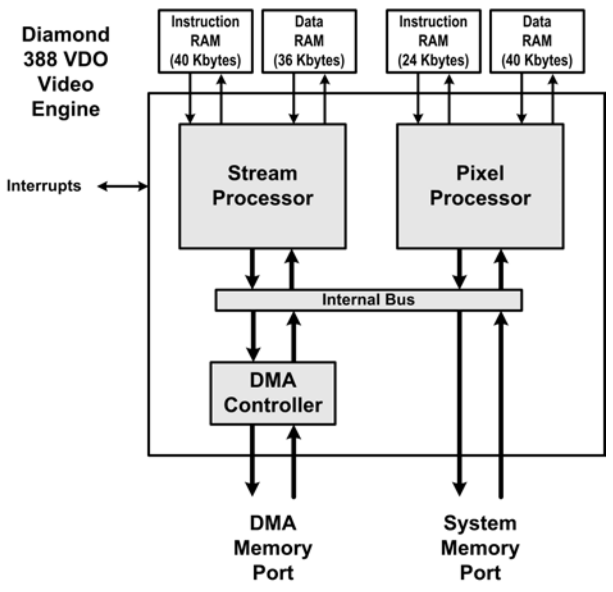


Figure 3: Tensilicas Diamond 388VDO Video Engine IP core

Combining the compositional-subsystem style of design with asymmetric multiprocessing (AMP) in each subsystem makes it apparent that products in the consumer, portable, and media spaces may need 10 to 100 processors-each one optimized to a specific task in the product's function set. Programming each AMP application is easier than programming each multithreaded SMP application because there are far fewer intertask dependencies to worry about. Experience shows that this design approach is eminently practical. By using this approach, you will avoid many of the optimization headaches associated with multiple application threads running on a limited set of identical processors in an SMP system.

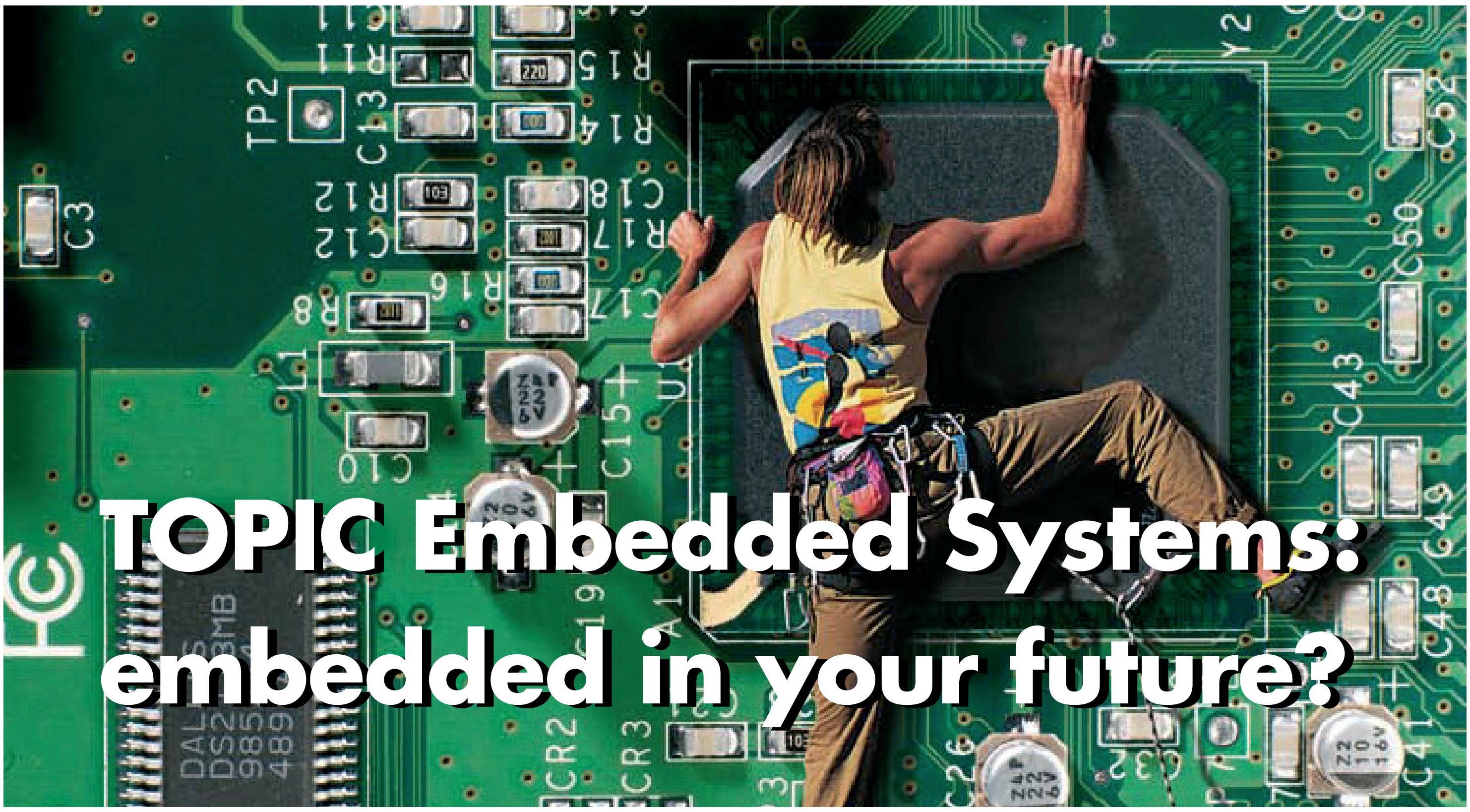
Computer Science…
Beyond the Order

océ
Printing for Professionals

*'Tijdens mijn studie kunstmatige intelligentie heb ik mij behalve in informatica verdiept in psychologie, filosofie en taalkunde. Die achtergrond komt mij goed van pas bij het bedenken van nieuwe software voor Océ. Eindgebruikers weten doorgaans niet welke toepassingen ze missen. Ik probeer hun werkwijze te begrijpen om die vervolgens door middel van technologie verder te kunnen verbeteren. Dat open denken vanuit de klant vind ik erg boeiend.'*

Joost Meijer

Ben jij ook op zoek naar een Beyond the Ordinary Job? Kijk dan op **www.oce.nl/jobs**



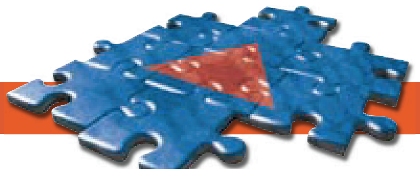
## Gedreven software- of hardware engineer?

**Bij ons krijg je uitdaging als arbeidsvoorwaarde!**

TOPIC Embedded Systems in Best is gespecialiseerd in embedded software ontwikkeling, test-, integratie- & configuratiemanagement en hardware design. De bijna 150 gedreven software- en hardware specialisten die behoren tot de besten in hun vakgebied, werken bij TOPIC aan state-of-the-art en gevarieerde projecten in de consumentenelectronica, medische- en professionele systemen. TOPIC onderscheidt zich door de persoonlijke benadering van medewerkers en opdrachtgevers, de kwaliteit en professionaliteit van onze TOPIC'ers en daardoor het niveau van onze projecten.

Vanwege de succesvolle groei van TOPIC Embedded Systems hebben wij uitdagende functies voor ervaren **embedded software engineers, hardware designers, software testers en configuration managers.** Beschik jij over minimaal 2 jaar werkervaring in bijvoorbeeld C, C++, C#, Windows, UNIX, UML, RUP, O.O., ISEB, TMap, CMMi, TPI, CMSynergy, DSP's, VHDL, ASIC's, FPGA's, real-time en embedded software? En wil jij je blijven ontwikkelen in projecten en via ons Personal Improvement Program? En wil je graag arbeidsvoorwaarden op maat? Mail dan snel je motivatie met CV naar recruitment@topic.nl of bel eerst met Frank de Roo, manager recruitment, (0499) 336 979.

TOPIC EMBEDDED SYSTEMS
EMBEDDED IN YOUR FUTURE



Voor meer informatie: WWW.TOPIC.NL

# OOTIs contribute to the Daedalus system-level design framework

Dorieke Schipper, Andy Pimentel

Eindhoven University of Technology
University of Amsterdam

*Advances in silicon processing technology enable integration of ever more complex systems, containing multiple processing elements on a single chip, as a result of which the design complexity increases. The successful deployment of these Heterogeneous Multi-Processor Systems-on-Chip (MP-SoC) requires a very high design productivity to deal with the growing complexity of the system with limited design resources. This design productivity problem has led to the emergence of system-level design.*

## Introduction

System-level design offers a higher level of abstraction, which allows designers to model and simulate the behavior of complex embedded systems with minimal effort. Furthermore, the use of architectural platforms facilitates reuse of IP components. Hiding complexity and implementation details from the users makes system-level design very suitable for exploring architectures in the early design stages, contributing to make better design decisions in an early stage of the design process. Simulation and prototyping of architectures plays a crucial role in matching and selecting candidate architectures to the requirements, such as execution speed or cost, of the final product.

System-level design for MP-SoC-based embedded systems however still involves a substantial number of challenging design tasks. For example, applications need to be decomposed into parallel specifications so that they can be mapped onto an MP-SoC architecture. Subsequently, applications need to be partitioned into HW- and SW-parts since MP-SoC architectures often are heterogeneous in nature. To this end, MP-SoC platform architectures need to be modeled and simulated to study system behavior and to evaluate a variety of different design options. Once a good candidate architecture has been found, it needs to be synthesized, which involves the synthesis of its architectural components as well as the mapping of applications onto the architecture.

To accomplish all of these tasks, a range of different tools and tool-flows is often needed, potentially leaving designers with all kinds of interoperability problems. Moreover, there typically remains a large gap between the deployed system-level specifications (or models) and actual implementations of the system under study, known as the *implementation gap*.

## OOTI workshop 2007: Daedalus

The OOTI workshop 2007 was the last project assignment before the OOTIs went to the companies for their final projects. The goal of this three-month workshop was to improve usability and integration of the Daedalus system-level design framework, developed by the Leiden Embedded Research Center (LERC) in cooperation with the University of Amsterdam (UvA). This framework addresses the challenges described above and provides a single environment for rapid system-level architectural exploration, high-level synthesis, programming, and prototyping of MP-SoC architectures.

In Figure 1, the conceptual design flow of the Daedalus framework is depicted. Here, a key assumption is that the MP-SoCs are constructed from a library of pre-defined and preverified IP components. These include a variety of programmable and dedicated processors, memories, and interconnects, thereby allowing the implementation of a wide range of MP-SoC platforms. This means that Daedalus aims at *composable MP-SoC design*, in which MP-SoCs are strictly composed of IP library components. Starting from a sequential multimedia application specification in C, the KPNgen tool allows for automatically converting the sequential application into a parallel Kahn Process Network (KPN) specification. Here, the sequential input specifications are restricted to so-called static affine nested loop programs, which is an important class of programs in, e.g., the scientific and multimedia application domains.

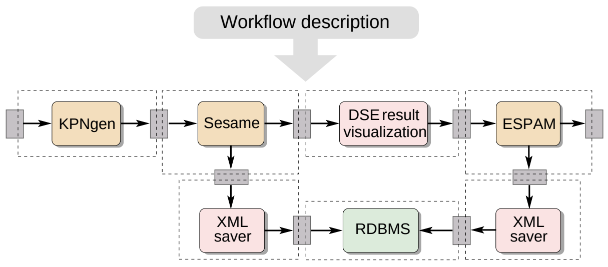


Figure 1: The Daedalus framework

The generated or handcrafted KPNs (the latter in the case that, e.g., the input specification did not entirely meet the requirements of the KPNgen tool) are subsequently used by the Sesame modeling and simulation environment to perform system-level architectural design space exploration. To this end, Sesame uses (high-level) architecture model components from the IP component library (see the left part of Figure 1). Sesame allows for quickly evaluating the performance of different application to architecture mappings, HW/SW partitionings, and target platform architectures. Such exploration should result in a number of promising candidate system designs, of which their specifications (system-level platform description, application-architecture mapping description, and application description) act as input to the ESPAM tool. This tool uses these system-level input specifications, together with RTL (ed: Register Transfer Level, i.e. common hardware design language abstraction level) versions of the components from the IP library, to automatically generate synthesizable VHDL (ed: standardized hardware description language) that implements the candidate MP-SoC platform architecture. In addition, it also generates the C code for those application processes that are mapped onto programmable cores. Using commercial synthesis tools and compilers, this implementation can be readily mapped onto an FPGA for prototyping. Such prototyping also allows for calibrating and validating Sesames system-level models, and as a consequence, improving the trustworthiness of these models.

The Daedalus framework is targeted to support MP-SoC architects in industry, as well as small and medium-sized enterprises. However, the Daedalus framework itself is still in development and at the start of the OOTI-project the three main tools (KPNgen for generating KPNs from ‘C’ code, Sesame for design space exploration and simulation, and ESPAM for generating synthesizable VHDL) were only provisionally connected. Installing the framework or setting up an experiment could only be done by a software expert with extensive knowledge about the tools used within Daedalus.

The main goal of the OOTI workshop was to build a framework to make the tools accessible for the user, who is not a software expert, to do system-level design of MP-SoCs. A second goal was to allow different tool flows to be composed, such that newly developed tools can easily be integrated into the Daedalus framework. A non-software expert should be able to set up and configure a complete experiment, running from a C-application to a working FPGA prototype.

## Results

During the OOTI workshop a couple of improvements have been made to the Daedalus framework and several supporting tools have been added to improve the user-friendliness, flexibility and deployability of the framework. Key improvement is the new notion of a configurable tool flow, solving important interoperability issues and improving flexibility; thereby making the Daedalus environment ready for integration of new tools as well as for customization of the design flow.

The design flow (or tool flow) in Daedalus is composable and constructed from design flow blocks. These design flow blocks, which are illustrated as the dashed boxes in Figure 2, are the tools that take part in the design flow together with their input- and output descriptions. The latter descriptions, illustrated by the grey boxes in Figure 2, provide information about what input/output data a tool consumes/produces and from/to where it reads/writes this data. A design flow is described as a composition of the design blocks. For example, Figure 2 shows a design flow which includes a visualization block to graphically show Sesame's design space exploration results and which stores both these results and ESPAM's prototyping results in a database using the so-called XML saver tool. Evidently, this composability of the design flow allows for easily adding new design steps to a design process, or to customize design flows for specific application domains.

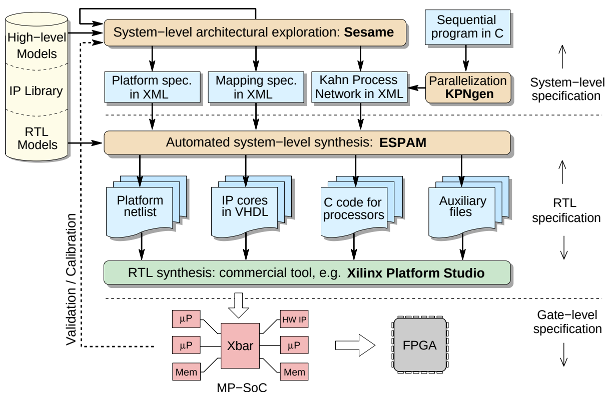


Figure 2: A Daedalus design flow

Second, a database is added to the Daedalus framework, storing the settings and results of every experiment. The Oracle Berkeley DB XML relational database management system has been chosen to be integrated into Daedalus, because most design information, as well as experimental results, is described using XML-based descriptions, making a native XML-database the most suitable choice. Enough information is stored to completely reproduce past experiments. Furthermore, a user interface is developed to allow the user to easily search and browse experimental results.

The third major addition to the Daedalus framework, developed during the OOTI workshop, is a separate FPGA control and monitoring tool, including a configuration manager, an execution control panel, and an on-line monitoring console. The VHDL, generated by ESPAM, is synthesized using commercial synthesis tools and compilers. The control and monitoring tool can be used to evaluate these FPGA-based prototypes of MP-SoCs. The tool is designed in such a way that users unfamiliar with FPGA prototyping boards can perform experiments to improve the MP-SoC design as a final step in the design process.

## Daedalus in business

Recently, a project has been initiated together with the Dutch SME Chess B.V. to put Daedalus to work in a real-life situation. Goal of the project is to employ Daedalus to develop a still image compression system for very high resolution images. Chess B.V. is a company that provides image processing solutions for customers that build industrial process monitoring and medical appliances. With respect to this, the still image compression systems for different customers have to meet different performance and cost requirements.

Daedalus makes it possible to evaluate a large range of candidate designs in a matter of days, yielding valuable information about the cost, design time, space, performance, etc. of the different candidate designs. Promising candidates can be evaluated in more detail using the automatically generated FPGA prototype and Daedalus control and monitoring tool. Daedalus thereby contributes to finding the best solution for every single customer.

## Conclusion

During the OOTI workshop 2007, OOTIs have gained experience with system-level design of MP-SoCs and the design challenges faced when building large, complex tool chains to support system-level design. The main objective of the workshop was to improve the user-friendliness, flexibility and deployability of the Daedalus system-level design framework. Thanks to the OOTI contribution, Daedalus can now be employed in a real-life situation, where it provides MP-SoC designers the opportunity to make important design decisions based on robust simulation results, in an early stage of the design process.

# Multi-Processor Programming for Embedded Systems

Andreas Hansson, Benny Åkesson, Andrew Nelson and Jef van Meerbergen

Eindhoven University of Technology
Philips Research Laboratories, Eindhoven

*This article discusses the structure of an advanced practical course on how to program multi-processor embedded systems. Programming of multi-processor embedded systems brings many challenges compared to software development for desktop computers, mainly due to differences in the processor architecture and tooling, the functionality and interaction with the memory system, and how the programmer is exposed to the parallelism in the system.*

## Introduction

Compilers and tooling for embedded processors are typically less forgiving than those of common desktop processors, and the processors have tight area and power budgets, which often lead to functionality such as floating-point support being left out. Additional complications arise as the embedded programmer is exposed to the memory architecture. The processors often lack caches and require explicit memory management. Moreover, the memory is often distributed, with non-uniform sizes and access latencies. It is also common that the processors do not run an operating system, and hence do not have memory management functions or a file system. The low-level view of the memory system enables the designer to efficiently map algorithms to the platform, but requires detailed knowledge about the architecture.

Despite the difficulties in porting an application to an embedded processor, the biggest challenge that faces the embedded programmer is typically to exploit the parallelism of the system. Most programmers are used to sequential languages, like C, and are unfamiliar with parallel programming. The starting point is therefore often an algorithm, implemented as a sequential program, that has to be partitioned into tasks and the tasks mapped to processors. Since the algorithm is distributed, the programmer also has to solve the task of communication and synchronisation between tasks.

The course Embedded Systems Laboratory, given by the Electronic Systems Group at Eindhoven University of Technology, aims to familiarise students with the aforementioned issues. The goal of the course is to teach how programming for embedded multi-processor systems and ordinary desktop computers differ. Students furthermore learn how algorithm implementation, processor synchronisation and communication, and memory mapping affect the quality of an application executing on multiple processors. The course is project based and the assignment concerns mapping a JPEG decoder on a multi-processor platform, about which more presently.

## Assignment starting point

The application used in the course is a fully functional JPEG decoder written in ANSI C.

The decoder is to be mapped to the hardware platform depicted in Figure 2. The architecture consists of three uniform Silicon Hive [1] Very Large Instruction Word (VLIW) processor cores, a large off-chip SRAM and a frame buffer for video output. The different components are interconnected by an instance of the Æthereal Network on Chip (NoC) [2]. In addition, a general-purpose host CPU is attached to the system. Next, we give a brief introduction to the concepts of JPEG decoding, and to the different building blocks of the architecture.

## Application

Decoding a JPEG image is a non-trivial task involving similar steps as many other media codecs, such as MP3, AAC, and H264. The core of all the aforementioned standards is a discrete cosine transform, that transforms data into the frequency domain. This allows pruning of frequencies to which humans are less sensitive. The formats are hence lossy, but enable a trade-off between file size and signal quality. To achieve a good compression ratio, however, the transformation into the frequency domain is combined with other techniques, like run-length encoding. The JPEG decoder can be generalised into three main decoding stages, shown in Figure 1: Variable Length Decoding (VLD), Inverse Discrete Cosine Transform (IDCT) and Colour Conversion (CC).

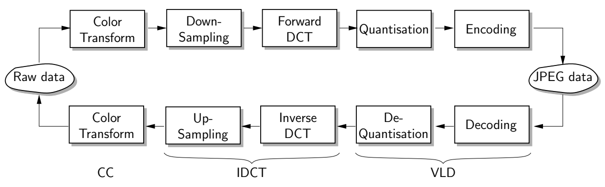


Figure 1: JPEG encoding and decoding.

The VLD decodes the variable-length encoded JPEG data, dequantises it and arranges it into blocks of 8x8 values, referred to as minimum coded units (MCU). The MCUs initially contain frequency data, but are transformed from the frequency domain to the pixel domain, and upscaled if necessary, by the IDCT step. After this transformation, the blocks contain image data in the YCbCr colour format that is converted to RGB in the CC step.

The JPEG decoder is chosen for the course as it offers a manageable amount of code to familiarise with. At the same time, the decoder retains the technical complexities of its audio and video counterparts, thus giving the application good educational value. The decoder additionally has the benefit of being familiar to the students and fun to work with, as the results can be presented on a screen attached to the actual hardware platform.

## Hardware platform

The platform used in the course builds on the concept of using multiple distributed computational and storage resources, interconnected by a scalable NoC. Silicon Hive provides their low-cost, low-power domain-specific VLIW processing cores, and NXP provides the interconnect fabric. The students hence work with industrially relevant intellectual property components. Using this type of embedded platform in the course is representative for signal-processing architectures where low power, and support for many features and standards is imperative. A complete architecture instance is mapped to a Xilinx Virtex4 LX-160 FPGA. The instance runs in 48 MHz, occupies roughly 7 Mgates, and contains: three VLIW cores, a host CPU, a central memory, a frame buffer, and a NoC that ties it all together. The FPGA is available in the classroom during the laboratory sessions, and it is possible for students to work remotely between sessions.

The Silicon Hive VLIW cores are customisable, making it possible to adapt the costs and the performances of the various computation nodes to a given application, as advocated in [3,4]. For the course, we use a simple, three-issue-slot architecture, *without a floating-point unit*, with a single multiplier and Load Store Unit (LSU). The cores use a memory-mapped architecture and have a master interface to enable reads and writes to memories external to the processor. As shown in Figure 2, every processor also has its own private instruction and data memory. Having memory distributed

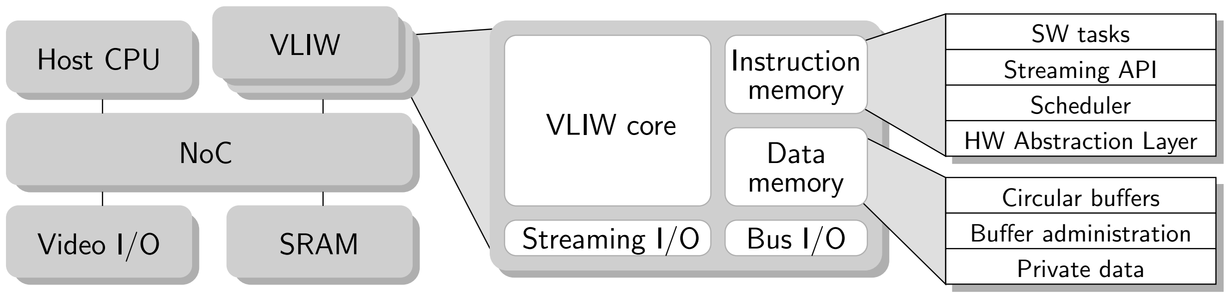


Figure 2: Architecture template.

over the different processing devices provides higher bandwidth with lower latency, which results in a higher performance at a lower power consumption [5]. The memories are also accessible through a slave port on the processor's bus interfaces, forming a distributed memory together with the dedicated memory tiles. The challenges that arise due to the selected processor architecture are: 1) the application must use fixed-point arithmetic, 2) multiplication and load/store operations compete for the same instruction slot, 3) the application must fit in a local memory of 32 kbyte (which is chosen to just barely fit the complete JPEG decoder), 4) the processor core *has no caches* and requires explicit memory management.

As seen in Figure 2, the system contains not only VLIW processing cores, but also a host CPU. The host is a general-purpose processor that is responsible for the initialisation and orchestration of the hardware resources. Although the host processor is typically part of the SoC, e.g. like in the IBM Cell [6], we choose to let the host interface with the system via USB, thus allowing interaction with an external computer. Together with an extensive set of system libraries, this enables the students to use their own PCs as host CPUs during the labs.

In addition to the 32 kbyte of data memory in each core, a central memory tile, here after referred to as *external memory*, provides 8 Mbyte of SRAM. While being significantly larger, the external memory, however, has an access time an order of magnitude larger than the local memories. This is due to the traversal of the NoC, and the sharing of the memory read and write port. The platform instance has only one external memory, as is commonly the case, either for cost reasons or due to a limited number of pins [7]. In addition to the background memory, the cores can also write to a frame buffer, where a designated display controller presents the contents on a DVI output port. This functionality is used during the laboratory sessions to get immediate visual feedback of the results.

All the aforementioned hardware blocks are physically interconnected by the Æthereal NoC [2]. In Æthereal, the different master and slave interfaces are logically interconnected by *connections*. A connection can be seen as virtual wires, offering a certain bandwidth and latency. Together, a set of connections forms a *use-case*, which acts as a virtual on-chip infrastructure. Network resources are allocated for a number of use-cases, using the UMARS tool [8], and it is left as an exercise for the students to choose an appropriate use-case for their specific JPEG decoder implementations. This involves determining which cores need to communicate and characterising their bandwidth and latency requirements. There are also use-cases where only certain cores can access the external memory.

## Tools and support libraries

The Silicon Hive cores are supplied together with a retargetable compiler, assembler and linker, as well as a complete system development environment. This enables the developer to quickly implement and debug applications, evaluate e.g. memory footprint and instruction schedule, and thus decide on mapping to processors and memories.

In the development environment, a number of refinement steps are possible, going from fast checking of the functional correctness, to cycle-accurate simulation. First, all code is compiled with `gcc`, to verify that the algorithm is working for the supplied set of reference images. Second, the code that is to be run on the cores is compiled with `hivecc`, but not scheduled. This enables the programmer to generate code with instruction semantics of the specified core. Third, the compiled code is scheduled to maximise Instruction Level Parallelism (ILP), and the programmer thus gets a complete view of the utilisation of the core's resources, i.e. the register files and interconnect. In this step, the tools also provide feedback about memory usage, instruction slot scheduling, and detailed profiling information. The fourth and last step uses the FPGA, with the student's computer acting as a host. The host then loads the microcode to the embedded cores on the FPGA and starts the execution.

The availability of development tools and support libraries remove much tedious and error-prone work, and thus enable the students to focus on solving the actual assignment.

## Assignment overview

The assignment is to map the JPEG application on the presented hardware platform. The students work in design teams with four members. Effort is made to ensure that all groups are multi disciplinary and multi cultural and hence contain students with different educational and cultural backgrounds. After an initial week of introductory exercises of tutorial nature, the teams assign roles with different responsibilities to their members. The four roles are: 1) application expert, 2) architecture expert, 3) embedded programming expert, and 4) group leader. The task of the application expert involves learning the details of the JPEG decoding algorithm, and to identify the important functions in the code and their interfaces. The architecture expert focuses on the details of the processing cores, network-on-chip and memories. The embedded programming expert learns how to port and upload code to the embedded VLIW core, and how to use the system support libraries. Lastly, the group leader is responsible for dividing the work among the members, reporting the team progress, and helping the team wherever needed.

To pass the course, each team has to present a functionally correct implementation of the application executing on a single core and at least two parallel implementations. Each team furthermore has to deliver a report explaining their solutions and benchmarking results, showing what they have learnt during the course. Students are graded individually and must hence also present their personal contributions to the group during an oral exam.

The project is carried out in three phases: 1) porting the application to execute on a single core on the target platform, 2) parallelising the application to use multiple cores, and 3) optimising the solutions to reduce the decoding time. These steps are discussed further in the subsequent sections.

## Porting the application

The JPEG application is distributed as sequential C code that executes on a normal desktop computer. The first challenge of the design teams is to port the code to execute on a single VLIW core. The major issues to solve involve memory management, and handling of console and file I/O.

No standard library function is provided for dynamic memory allocation, since the memory architecture is non-uniform, creating multiple placement options. Memory allocations are hence done statically, and the programmer determines if a particular variable should be mapped to the limited amount of faster local memory of the core, or to the larger but slower external memory. A challenge in this step is that statically allocating arrays requires algorithmic knowledge from the programmer, since they must be dimensioned for the worst case.

The application makes rich use of the console to print debug information in case there is something wrong in the implementation or the encoded image. The architecture has no means of outputting this information, since it does not have a console.

The original JPEG decoder uses file system I/O to read the encoded bit stream and to write the decoded image. However, the provided architecture does not have a file system. Instead, the core must read the encoded image from the external memory, which is the only memory large enough to store it, and write the decoded image to the frame buffer. The host is used to transfer the encoded image from the file to the external memory, which requires familiarity with the system support libraries for communication between the host application and the SoC. The decoded image is also written back to external memory during development, allowing the host application to read the output and compare to a reference image that was decoded before porting the code. Automating this procedure allows bugs introduced during porting to be discovered quickly.

## Parallelising the application

After successfully porting the application to the target platform and performing initial benchmarks, the design teams proceed by parallelising the application to make use of multiple cores. As previously mentioned, the assessment criteria require each group to implement and benchmark at least two different parallelisations. The two most common solutions involve exploiting *data parallelism* by allowing multiple cores to work on different parts of the image, and *functional parallelism* where the decoding functions are mapped to the different cores. Many variations of these solutions have been explored during the course, including hybrid versions that aim to combine the best of both. In this article, we limit the discussion to the two basic solutions, which are presented next.

### Data parallelism

The idea of a data parallel implementation of the application is that multiple cores are assigned to decode different parts of the image. A benefit of this approach is that very few changes are required to the ported code executing on a single core. All cores execute the same program, but use a unique identifier to determine which part of the image to decode.

The image can be divided among the cores according to different strategies that distribute the complexity of the image differently among the cores. This is illustrated in Figure 3 where the image is partitioned among three cores according to the different shades of grey. Dividing the image in three horizontal slices, as done in the left part of the figure, would create an unbalanced load in a scenic picture with a blue sky in the top, since significantly less computation is required by the IDCT for this part of the image. Another strategy that is better in this respect is tiling, shown in the right part of the figure, where a core decodes every third MCU block.

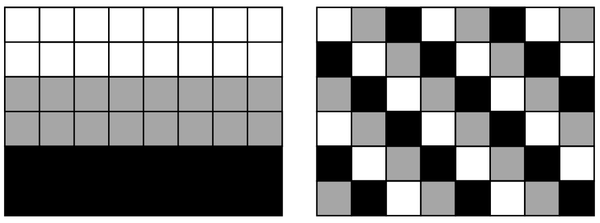


Figure 3: Strategies for data partitioning

The main drawback with the data parallel JPEG decoder is that the VLD is inherently sequential, and it is not possible to know exactly where a block begins without decoding the previous ones. This implies that all cores must read the encoded image from external memory and perform the VLD, although the IDCT and colour conversion is skipped if the current MCU block does not correspond to its assigned part of the image. Increasing the number of cores hence creates additional contention for the external memory, limiting the scalability of this approach. Note that the partitioning strategy to the left in Figure 3 only requires the first core to read 1/3 of the image from external memory, and the second core 2/3, while the last core must read all of it. This can be compared to the tiling strategy on the right in the figure, which requires all cores to read the entire memory, increasing memory contention.

## Functional parallelism

In this solution, the decoding functions are mapped to the different cores, creating a pipeline where an MCU block is processed by all the cores in sequence before decoding is complete and it is written to the frame buffer. An important challenge is to determine how to partition the functions among the different cores to get a balanced load and to minimise inter-core communication. The cycle-accurate simulation model of the VLIW core provides profiling information containing execution times and number of invocations of the different functions. Although this information is very helpful when deciding how to split the functions between the cores, it comes with the assumption that communication is instantaneous and that all memory accesses take one cycle. This information should hence be used with extreme care and must be validated on the actual FPGA implementation. A common way to split the decoder is according to the three stages, VLD, IDCT, and CC that were previously explained. This partitioning has the benefit of providing clear interfaces between the functions where only frequency blocks and pixel blocks are communicated between the cores.

A difficulty with a functional partitioning is that different pictures place very different computational requirements on the functions in the decoding algorithm. Figure 4 shows the decoding time required for the VLD, IDCT, and CC, respectively, on a single core. The two images are both XGA resolution (1024 x 768), but *Noise* contains a lot of high frequency information and has a size of 748 KB, whereas *Quiet* contains mostly uni-coloured MCUs and occupies only 53 KB. Note that for the largely uni-coloured picture, the colour conversion takes more than one third of the time. Conversely, for an image with a lot of detail, the VLD requires almost two thirds of the time. This shows that it is extremely difficult to partition the decoder in a way that creates a good balance between the cores for all pictures.

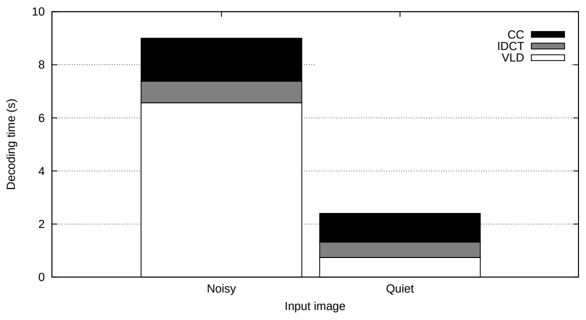


Figure 4: Single-core profiling.

Communication between the cores is done using the C-HEAP protocol [9], provided with the hardware platform. C-HEAP uses logical circular FIFOs of programmable size that are allocated in the local memories of the cores. Communication is blocking and reading from an empty FIFO or writing to a full FIFO temporarily causes a core to stall. The sizes of the FIFO buffers hence affect the throughput of the application, resulting in a trade-off between memory usage and performance. Communication is structured such that a producing core writes into the memory of a consuming core. This is because writes are posted, while a reading core blocks until a response has arrived. Local reads and remote writes hence reduce the load on the interconnect and provides increased performance. Using C-HEAP for all communication has the benefit that it is possible to measure how much time a core spends waiting for another by instrumenting it with a few counters. This allows the load balance of the partitioning to be verified on the actual system with real communication and memory access latencies.

Compared to a data parallel implementation, the functional partitioning is scalable in terms of memory requirements. The cores only have a subset of the decoding functions and hence require less instruction memory. There is furthermore only a single core reading from external memory, reducing contention. A drawback of this approach is that it is difficult and time consuming to create a good load balance. Additionally, the partitioning has to be reconsidered if the platform architecture or software is modified, as this may cause the current bottleneck to shift.

## Optimising the decoder

Once a solution is functionally correct, an iterative optimisation and benchmarking phase begins to improve its quality. In this section, we elaborate on the benchmarking procedure and optimisations for a decoder executing on both a single core and on multiple cores.

### Benchmarks

The teams are encouraged to continuously evaluate their designs through quantitative benchmarks throughout the development process. This allows them to directly see the impact of design decisions on quality, and learn about the trade-offs involved. The benchmarking procedure is standardised by a committee, comprised of representatives from all design teams. This ensures that all teams are familiar with the procedure, and that their results are comparable. The standardised benchmarks consider two aspects of embedded systems being performance, in this case decoding time, and memory requirements.

Decoding time is measured by starting a timer on the host after uploading the encoded JPEG image to the external memory. After starting the timer, the host starts the three cores and waits until all of them have completed. A benefit of this benchmarking method is that it is easy to implement, although a drawback of the approach is that the time required for the host to start the cores and to detect that they finished execution is captured by the measurement. Since the host processor is connected via USB, this overhead may add up to a second to the decoding time. Benchmarking the memory requirements of a solution is simple, as the required amount of instruction and data memory is output by the tooling for every core.

### Optimisations for a single-core decoder

The optimisation process is guided by profiling the code using the previously mentioned cycle accurate simulator. Profiling helps identifying functions that are called often or requires a lot of time to execute, which indicates that they may be good candidates for optimisation.

Optimisations targeting the single core decoder can be categorised as: 1) algorithmic short cuts, 2) adaptations to fit with the computational cores, and 3) adaptations to fit better with the communication infrastructure. The first category involves using knowledge about the JPEG decoding algorithm to speed up decoding, such as throwing away higher frequency components, or exploiting common cases in the image format. The second category concerns making the computation more efficient by adapting it to the processor core architecture to get a more efficient instruction schedule. The third category considers rewriting the code to reduce the number of memory accesses. For the first category, the programmer must have deep insight into the JPEG algorithm. The latter two categories require the programmer to be intimately familiar with the target architecture and tooling.

The most influential algorithmic short cut in JPEG decoding is that of IDCT-bypassing. That is, when an MCU is uni-coloured and does not contain any frequency components, the IDCT can be skipped. The short cut does not compromise the result, and in the case of the *Quiet* benchmark image, more than half of the MCUs are skipped. Another important optimisation is that of detecting common colour encodings in the CC. Most JPEGs use only two types of encoding (4:2:2 and 1:1:1), and by implementing special CC functions for these common cases, the indexing in the CC is greatly simplified.

There are many opportunities to improve the JPEG decoding time by exploiting knowledge of the processor core architecture. One of the major adaptations is to replace the given Loeffler IDCT with Chen-Wang IDCT. The latter uses fewer multiplications and is better matched to the VLIW in question. By further adapting the code to use variables rather than arrays, the load on the register banks increase, but extra transfers to memory are avoided, resulting in a net gain. Another technique, useful in the CC, is to use look-up tables with precomputed values, and thus avoid a multiplication and a shift operation.

The last category of optimisations targets the memory architecture, aiming to reducing the number of accesses to remote memories, and to use the accesses more efficiently. A significant speed-up is achieved by using local memory rather than the shared external memory (or the memory of another core). The size is, however, very limited, and not all data will fit in the local memories. To use the remote memory accesses more efficiently, the code must be adapted to read/write whole words rather than sub-words, such as characters. The latter optimisation, for example, reduces the time required for the VLD by almost two times.

### Optimisations for parallel decoders

The optimisations used for the single-core decoder are also applicable for the parallel decoders, but it is quickly noted by the students that the speed-ups observed for the single-core solution are not reflected when they are applied to code that runs on multiple cores. This demonstrates the influence that communication (and not only computation) has on the decoding time.

There are refinements of data parallel implementations, addressing the memory contention. One such refinement involves ensuring that only one core performs the VLD on a particular line and shares the important results with the other cores through a structure in memory, allowing them to skip the line. This optimisation reduces the decoding time for both noisy and quiet with approximately 15%. A drawback of this refinement is that additional memory (approximately 2 kbyte) is required to store the information shared by the cores. The optimisations for the functionally pipelined version mostly considers moving smaller blocks of code between the cores to improve the load balance, or reducing the amount of data that is communicated between the cores.

### Example results

The single core implementation requires roughly 16 seconds to decode the noisy picture after initial porting. After the aforementioned optimisations, the decoding time is almost cut in half to 9 seconds. The data parallel and functionally pipelined version requires 7.2 and 7.8 seconds, respectively, to decode the noisy picture after all optimisations have been applied. This shows that parallelising the JPEG decoder improves performance, although the benefits are far away from the theoretical maximum. The single core and the parallel solutions all decode the simpler quiet image in about the same time, around two seconds, indicating that the benefits of parallelisation are diminishing as the amount of computation is reduced and communication becomes a dominant factor. One of the likely remaining bottlenecks is that burst behaviour could not be fully exploited.

The memory requirements for the different implementations are shown in Table 1. Note that both multi-core solutions use far more memory than the single core. Especially data parallel, which is essentially a triple single core, although with some specific optimisations. The functionally pipelined implementation has in comparison been greatly reduced in size, since each core does not have to contain the full JPEG decoder.

Table 1: Memory footprint for the different implementations.

| | Data memory (bytes) | Instruction memory (bytes) |
|---|---|---|
| Single core | 2910 | 19908 |
| Data parallel | 3 x 3446 | 3 x 26894 |
| Pipelined | 5670 + 1724 + 2504 | 23380 + 7204 + 4438 |

## Conclusions

In the Embedded Systems Laboratory, the students get to familiarise with many of the difficulties involved in programming multi-processor embedded systems. By the end of the course, they have successfully ported a JPEG decoder to the target multi-processor platform and evaluated a range of parallelisations on an actual FPGA instance. The assignment presents many challenges, ranging from working in a group to choosing the right compiler directives for a critical piece of an algorithm.

Embedded Systems Laboratory ran for the second time in 2008 with 31 participating master students, both from the Electrical Engineering and International Masters programme on Embedded Systems. The course concepts have furthermore been exported to the Technical University of Delft, where a similar course was given by the Computer Engineering department for the first time this year. Next year, we aim to introduce even more challenges in the assignment, and prepare another class of students for the problems we are facing with the wide-spread adoption of multi-processor systems.

## References

[1] Silicon hive, Available from: http://www.siliconhive.com, Silicon Hive, 2007.

[2] K. Goossens, J. Dielissen, and A. Rădulescu, The Æthereal network on chip: Concepts, architectures, and implementations, IEEE Des. and Test of Comp., 2005.

[3] A. Jerraya, A. Bouchhima, and F. Petrot, Programming models and HW-SW interfaces abstraction for multi-processor SoC, Pr. DAC, 2006.

[4] C. Rowen and S. Leibson, Engineering the Complex SOC: Fast, Flexible Design with Configurable Processors. Prentice Hall PTR, 2004.

[5] D. Soudris, N. D. Zervas, A. Argyriou, M. Dasygenis, K. Tatas, C. Goutis, and A. Thanailakis, Data-reuse and parallel embedded architectures for low-power, real-time multimedia applications, IEEE International Workshop on Power and Timing Modeling, Optimization and Simulation (PATMOS), pp. 243254, 2000.

[6] J. Kahle, M. Day, H. Hofstee, C. Johns, T. Maeurer, and D. Shippy, Introduction to the Cell multiprocessor, IBM Journal of Research and Development, vol. 49, no. 4/5, 2005.

[7] J. Leijten, J. van Meerbergen, A. Timmer, and J. Jess, Prophid: a platform-based design method, Journal of Design Automation for Embedded Systems, vol. 6, no. 1, pp. 537, 2000.

[8] A. Hansson, M. Coenen, and K. Goossens, Undisrupted quality-of-service during reconfiguration of multiple applications in networks on chip, in Proc. DATE, 2007.

[9] A. Nieuwland et al., C-HEAP: A Heterogeneous Multi-Processor Architecture Template and Scalable and Flexible Protocol for the Design of Embedded Signal Processing Systems, Des. Autom. for Emb. Syst., vol. 7, no. 3, 2002.

Meer dan 50 vacatures bij dé professionals in embedded software!

TASS

software professionals

TASS HAALT HET BESTE UIT DE BESTEN

jobs.tass.be
jobs.tass.nl