

The dusk of ASIC the dawn of ASIP

*Andreas Hoffmann, Achim Nohl
CoWare Inc.
Aachen, Germany
{andreas,achim}@coware.com*

Every year, Gartner Dataquest presents the day prior to the start of the Design Automation Conference (DAC) facts and trends for the EDA industry. Part of this event is the hand out of a list of the “*20 hottest companies and new products*” to see. These companies cover the complete bandwidth of EDA – from next generation layout to system level design.

This year, five out of these 20 companies were in the segment of “*algorithmic engines*” or application-specific instruction-set processors (ASIP). While the name ASIP is not very common, most people are familiar with realizations of ASIPs in different application domains: network processors, memory controllers, dedicated DSPs, etc.

What makes these processors different from standard RISCs or DSPs and why will they have a big impact on future SoC designs? This article tries to give some answers by motivating the usage of ASIPs from a technical and from a business perspective. Besides, one possible ASIP design solution will be introduced exemplarily.

Market Trends – from ASIC to ASIP

The continuing trend in consumer electronics is towards more complex products, while expecting longer stand-by times for mobile devices, offering more functionality at lower cost. These are conflicting requirements which translate into several challenges for the manufacturers on delivering products which feature higher performance, more functionality at lower power consumption. Since especially in the mobile market the requirements on the customer side is always on having the “*latest device*” featuring the latest features (e.g. 3D graphics, Bluetooth and wireless LAN), products need to be brought to the market in less time at lower cost.

Another big problem is that standards defining certain functionality like wireless LAN keep on changing – examples are the latest wireless LAN standards like UWB and H802. This calls for flexible and re-usable SoC architectures to adapt to late changes in the standard.

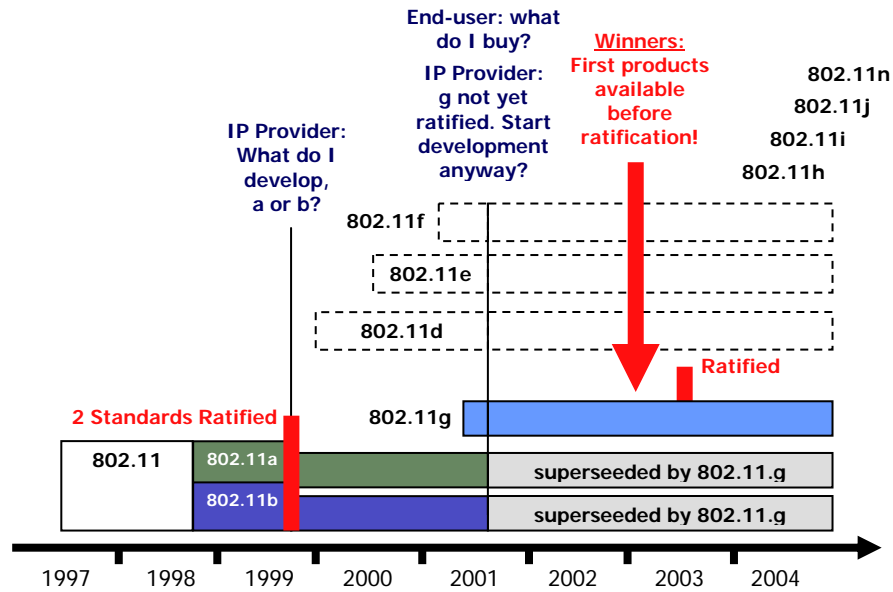


Figure 1: The evolution of the H802.11 standard over time

The dilemma when designing products with changing standards is shown in figure 1. Initially introduced as the H802.11 standard in 1997, two camps of companies could not find agreement on the standard and developed two different versions of it – 802.11a and b. Both of these standards were ratified, however, shortly after new flavours – 802.11d to g were brought into discussion. Customers were confused, so were the IP providers which did not know what to develop...

It took four more years after ratification of a and b before it was clear who the winner would be – 802.11g. Interestingly enough, customers accepted this as the standard *before* it was actually officially ratified – this means that only those vendors won, who took the risk to develop their product based on a non-ratified standard. 802.11a and b were superseded by 802.11g. In the meantime, even more proposals came up (802.11.h - 802.11.n).

This example demonstrates that hardwired logic (ASIC) is not the right choice to implement functionality that is exposed to changing standards. The resulting re-designs caused high development cost and IP providers using this approach were consequently late in the market. The winners here were those companies with flexible solutions based on programmable architectures. By doing so, they could adapt to changing standards by changing the software only.

However, programmability also has a number of drawbacks – it typically comes at the price of worse energy efficiency. The trade-off between programmable and fixed architectures is depicted in figure 2.

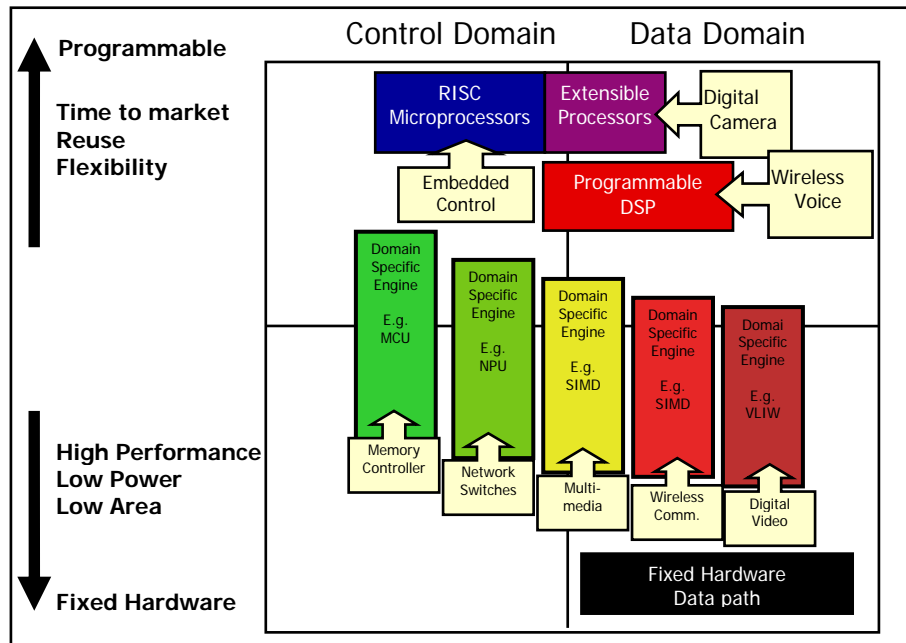


Figure 2: Processor specialization vs. ATE (Area, Time, Energy)

Hardwired logic on the one hand offers very high performance while having low power consumption and smaller area. The drawback is obviously that it does not allow any flexibility to adapt to changes after product roll-out. In contrast to ASICs, typical programmable cores like RISC or DSP microprocessors offer the required flexibility, however, can not cope with the requirements of modern portable devices with respect to power consumption and performance. SoCs containing a microcontroller (ARM/MIPS) and a DSP for the signal processing are widely used in the industry. However, spending more processing capacity by adding more programmable cores is hardly acceptable.

Besides the technical reasons for this with respect to power consumption and die size, there is also a business aspect attached to this - while paying royalties for one or two processors in one system is now widely accepted, it will hardly be acceptable to pay royalties for the 3rd/4th, etc. processor - margins especially in the wireless industry with mass products like mobile phones are very small (3-5% per device).

To conclude - the truth lies in the middle - next generation processors will be very application specific. They will be very close to ASIC with respect to energy efficiency and size while being programmable. To achieve this, these processors only provide a very limited programmability - just enough to adapt to changes e.g. in standards. Typically these processors feature complex and highly specialized instructions which can execute e.g. functions like FFT (Fast Fourier Transformation) calculations in a single cycle. This is the key to the energy efficiency.

Due to the high degree of specialization, there will be dedicated processors for different application domains like digital video, wireless communication, multimedia, etc.

The following figure shows quantitative numbers by comparing energy efficiency measured in mega-operations/instructions per mW (MOPS/mW) for different architectures

running the same benchmark. The application is taken from the area of digital video broadcast terrestrial (DVB-T) and was run on processor architectures from the RISC, DSP and ASIP space – a StrongARM, a Texas Instruments TMS320C5401 and an Infineon ICORE, respectively. Dedicated hardware executing the same application was not measured.

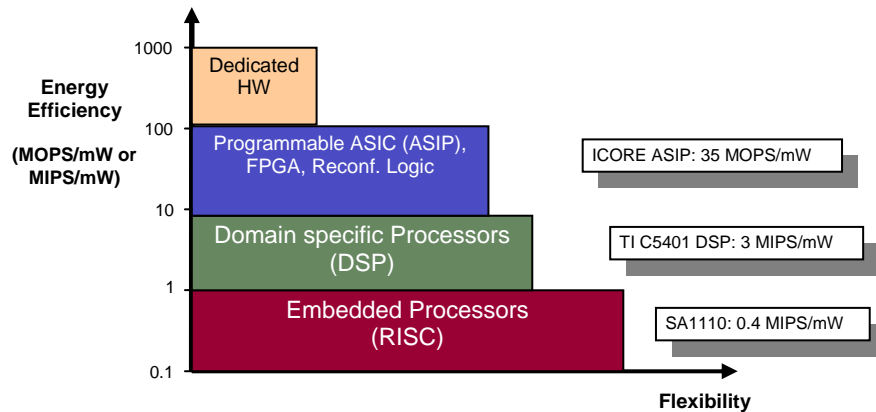


Figure 3: Flexibility vs. energy efficiency

It shows that there is roughly one order in magnitude of energy efficiency between a RISC embedded processor, a domain specific DSP, and an ASIP optimized for this particular application (number for ICORE is a courtesy of Infineon Technologies).

The remainder of this article goes exemplarily through the design process of an ASIP using the CoWare’s LISATek Processor Design tool-suite.

Application-specific Processor Design

In opposite to the design of traditional general purpose processors, the ASIP design flow is driven by a set of target applications, frequently specified in C/C++, CoWare SPW or Matlab. The design generally starts with a successive *architecture exploration* process that involves stepwise refinement of the architecture and timing abstraction levels based on the simulation and profiling results during architecture development. The abstraction levels range from un-timed high-level language instruction-set accurate (ISA) models down to cycle-accurate RTL synthesis models.

Due to the model refinement capabilities of an instruction-set language (ISL) the designer can abstract from architecture details and concentrate on the essentials in early design phases which is a big advantage in opposite to the traditional purely HDL based processor design flow.

Processor modelling with LISA

The Language for Instruction-set Architectures (LISA) is a powerful representative of ISLs. The current version, LISA 2.0, was initially developed at the Aachen University of Technology (www.iss.rwth-aachen.de) and serves as the basis for CoWare’s LISATek Processor Designer.

LISA captures all aspects of a processor architecture which required are to describe the hardware functionality as well as the instruction-set. While the functionality description is completely ANSI C based, the language has capabilities to describe instructions-sets with their binary encoding and assembly syntax. Moreover, LISA allows expressing timing relations of instructions in processors – an example is a pipelined architecture where instruction execution is spread over multiple cycles.

The following example taken from one of CoWare’s ASIP sample models shows an excerpt of a LISA description.

```

RESOURCE
{
  /* Harvard memory map for program and data memory*/
  MEMORY_MAP {
    PAGE(0),RANGE(0x0000, 0x0fff) -> prog_mem[(31..0)];
    PAGE(1),RANGE(0x0000, 0x0fff) -> data_mem[(31..0)];
  }
  /* FLAGS are set to R|X - prog_mem is read- and executable */
  MEMORY uint32 prog_mem {
    SIZE(0x1000); BLOCKSIZE(32); FLAGS(R|X); };
  MEMORY uint32 data_mem {
    SIZE(0x1000); BLOCKSIZE(32); FLAGS(R|W); };

  /* Register file with 16 registers */
  REGISTER uint32 R[0..15];
  /* 3 stage pipeline */
  PIPELINE pipe = { FE ; DC ; EX };
  /* A pipeline register to pass data through the pipeline */
  PIPELINE_REGISTER IN pipe {
    uint32 pc; /* the address of the instruction */
    uint32 insn; /* the instruction word */;
  }
  OPERATION fetch IN pipe.FE {
    BEHAVIOR
    {
      /* read instruction word and increment PC*/
      OUT.insn=prog_mem[PC];
      PC=PC+1;
    }
    ACTIVATION { decode }
  }
}

```

```

OPERATION decode IN pipe.DC {
  DECLARE
  {
    /* A group of operations for 32bit instructions */
    GROUP instruction = { add || jmp || nop };
  }

  CODING { IN.insn == instruction }
  SYNTAX { instruction }
  ACTIVATION { instruction }
}

OPERATION add IN pipe.EX {
  DECLARE
  {
    /* alu modes */
    GROUP mode = { add || sub };
    /* operand register addresses */
    GROUP src1,src2,dest = { reg };
  }
  CODING { 0b1001 0bx[16] src1 src2 dest }
  SYNTAX { "ADD" dest "," src1 "," src2 }
  BEHAVIOR
  {
    R[dest] = R[src1] + R[src2];
  }
}

```

The RESOURCE section declares all memories, buses, registers and pipelines. In the example, the processor uses a data and program memory using the same address space – one particular memory is selected via the address plus the specification of the memory page. This information is contained in the description of the MEMORY_MAP. Here, also bus connections would be modelled (not part of the example). Each memory is defined using the MEMORY keyword – a size and block size are assigned and some attributes are set (read, write, execute). The example model also contains 16 registers which are 32 bits wide. Moreover, a 3-stage pipeline is defined using the PIPELINE keyword and pipeline registers are assigned to it.

The instruction-set is modelled in LISA in so-called OPERATIONS. These operations contain the instruction-set information including binary encoding (CODING), assembly syntax (SYNTAX), state transfer function (BEHAVIOR) and timing (ACTIVATION). Operations in LISA are hierarchically ordered – several LISA operation form one instruction. GROUPs in LISA indicate alternatives – here, only add, jmp and the nop

instructions are valid. Exemplarily, the example shows the implementation of an add instruction. The binary encoding is split into terminal and non-terminal elements. Similar to the encoding, the assembly syntax shows that the mnemonic is followed by the encoding of the destination, source1 and source2 registers separated by commas. The BEHAVIOR specifies in pure C-code the state transition in case the respective operation is executed.

Timing relations between operations are modelled in LISA by assigning operations to pipeline stages (using the IN keyword) and activating each other. In the example, operation decode is assigned to stage DC and add is assigned to stage EX. When decode “activates” instruction (see ACTIVATION keyword), this implicitly means that add (if decoded) will be executed on cycle later than decode due to the special ordering of pipeline stages.

There are more than 40 LISA models in industry and academia from different architectural categories – RISC, DSP and ASIP. This includes different ARM and MIPS models, DSPs from Texas Instruments, StarCore and Ceva, as well as application specific processors from Infineon, STMicroelectronics, etc. As LISA allows modelling processors on different levels of abstraction, so are the previously mentioned models.

Exploring the design space

As stated earlier, the key to achieving an optimal architecture for a given set of application is an extensive architecture exploration and refinement process. However, the design space is huge as it is composed by four different dimensions (see figure 3), which need to be explored by the designer:

- *The instructions-set and its impact on the efficiency of the C-Compiler.*

There is a variety of possibilities regarding the optimal instruction-set of a processor architecture for a particular class of applications. Design decisions include the degree of parallelism in the application code that can be explored by the instruction-set using VLIW (Very Long Instruction-Word) instructions as well as the definition of special purpose instructions to accelerate specific portions of the application code while reducing power consumption. Image processing algorithms can be perfectly supported by data parallel SIMD (single instruction, multiple data) instructions. As the degree of instruction-set specialization increases, so does the re-usability of the instruction-set for other applications decrease.

Another important aspect of the instruction-set design is what is frequently referred to as “*processor-compiler co-design*”. Since most of the processor architectures require a C-Compiler for application programming, it is important to design a compiler which is aligned to the design goal of the processor architecture. For the C-Compiler, this can be translated into the requirements “high performance with high code density”. Unfortunately, C-Compiler design is strongly dependent on the instruction-set and micro-architecture, therefore it is essential to keep the C-Compiler design in the exploration loop of the processor architecture. There are certain architectural characteristics which help the C-Compiler to generate optimal

code, e.g. a sufficient number of allocatable registers or certain addressing modes that need to be supported.

- *The processor micro-architecture*

There are countless different possibilities to implement an instruction-set in a micro-architecture. These design decisions affect the overall architecture performance as well as die size and power consumption. Basic parameters to be fixed are the definitions of instruction and data pipelines, bypassing logic as well as the memory subsystem to reduce data and instruction access latencies. In contrast to high performance general purpose or even domain specific processor architectures where the micro-architectures are very complex and thus less power efficient, typically ASIPs feature micro-architectures that are simple with a three to five stage single pipeline, none or at most simple MMU (memory management unit) with a single level of caches.

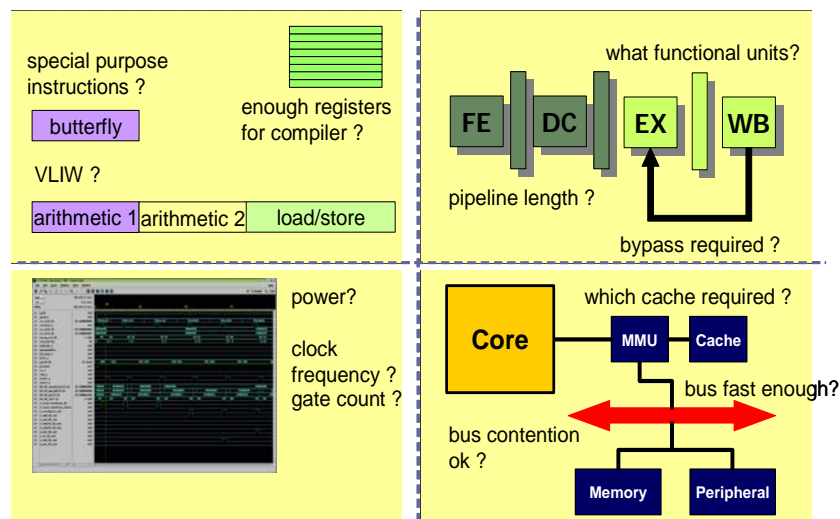


Figure 4: Four dimensions of processor design

- *RTL2GDSII implementation of the processor*

Meaningful numbers on power consumption, clock frequency and gate count can only be gathered after a synthesis run with the target technology. Based on these results, certain design decisions regarding the instruction-set, processor register count as well as the micro-architecture need to be re-iterated as there are several trade-offs to be made. In particular, the trade-off on the flexibility of the processor vs. the product of area, time, and energy needs constant monitoring.

- *Impact of the system on processor performance*

A common mistake that is made when designing ASIPs is that the integration into the target system context happens at the very end of the design process. However, the system behaviour and interaction with the processor has an immediate impact on the optimal processor micro-architecture, especially with respect to processor performance.

A typical example is a processor accessing memory via a shared system bus. In case of bus contention, the processor will stall until the data is delivered by the bus. The impact on the overall processor performance increases with deeper instruction pipelines. To mitigate this, a simple data and/or instruction cache can be inserted into the processor model to reduce the memory access latencies.

The difficulty in finding the optimal architecture from a huge design space as described above has hindered designers in the past to move non-programmable RTL into ASIPs. With the availability of new design tools supporting this exploration cycle, more and more designers will select a programmable solution in favour of hardwired logic to benefit from the advantages of programmability while keeping power consumption down.

CoWare's LISATek Processor Design tool-suite

When starting to design a new processor, the designer typically takes an educated guess to select one of the sample processor models as a starting point. The sample model library that comes with the tools contains on the one hand LISA models for different architectural categories (RISC, DSP) with different architectural features (VLIW, SIMD, MIMD) as well as sample models already optimized for the execution of algorithmic kernels like FFT, Viterbi, Motion estimation, etc.

Starting with one of these models (or if required from scratch with an empty model), the *Processor Designer* tool generates a complete set of SW development tools including optimizing C-Compiler and fast instruction-set simulator. CoWare partners with the market leader ACE (www.ace.nl) on the C-Compiler to deliver compilers with highest code density and execution speed.

The application code is compiled and run on a virtual prototype of the processor in software. During the simulation, profiling data is collected that points to hot spots in both the architecture as well as the application. Besides, RTL code and system integration models of the processor architecture for simulation in SystemC based system environments are generated.

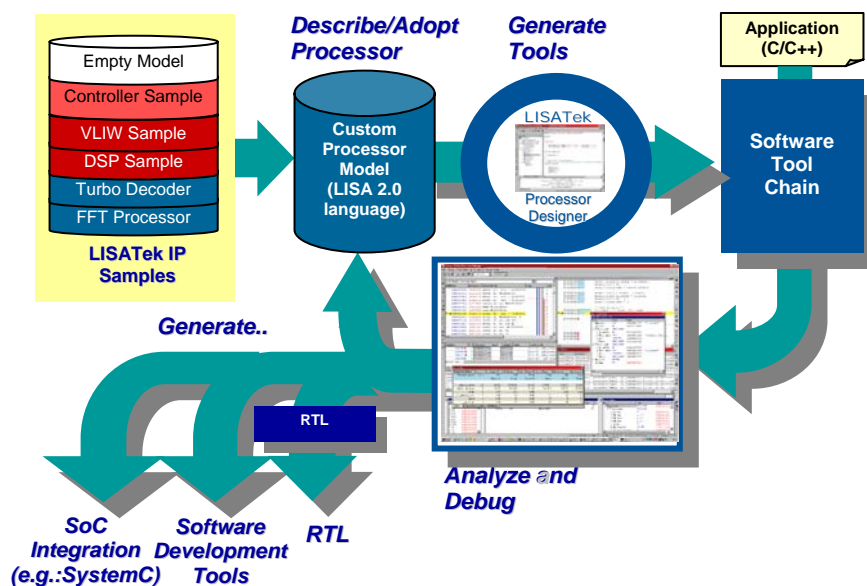


Figure 5: ASIP development process with CoWare’s LISATek Processor Designer

Based on the profiling results, the LISA model of the architecture under investigation is modified and the tools are re-generated. This is an iterative process which is repeated, until the best fit between application and architecture is obtained.

In a final step, the tool generates *end-user packages* consisting of all software tools as well as debuggers and the system integration models that can be shipped to the application developer or system integrator for immediate usage.

Summary

In a recent keynote, Kurt Keutzer from UC Berkley gave the following outlook on future SoC design: “ASIPs are the NAND gates of the future”. Obviously, we are not fully there, but there is a wide agreement among developers of such complex systems that there is a need for more programmability while power consumption needs to be kept down. The limiting factor in the past on the design of ASIPs was clearly a lack of design tools to automate the exploration and implementation of these processors, along with the respective tools to program and debug the software.

Recently, a number of companies have started to provide tooling to design these processors – some of them offer domain specific, configurable solutions (MIPS CorExtend, ARM Optimode, Synfona Pico for VLIW), others language based approaches (CoWare LISATek and Target Compiler).

The future direction will be towards the linkage of algorithm and processor design tools, extending already existing hardware implementation flows like CoWare SPW HDS to a programmable solution.