Application Specific Processor Design
Architectures, Design Methods and Tools

Achim Nohl
Solution Architect - Synopsys, Inc.
System Level Solutions
52134 Herzogenrath, Germany
achim@synopsys.com

Frank Schirrmeister
Marketing Director - Synopsys, Inc.
System Level Solutions
Mountain View, CA 94043
fschirr@synopsys.com

Drew Taussig
Processor Specialist – Synopsys Inc.
System Level Solutions
Chandler, AZ 85226
dtaussig@synopsys.com

Abstract—In this tutorial paper, we will outline a solution for prototyping, programming and implementing Application Specific Instruction-set Processors (ASIPs). A general introduction into this class of processor architectures and their characteristics is provided. The Synopsys Processor Designer tool suite and the LISA language for ASIP design are jointly introduced in the context of a H.264 design example. Finally, implementation results are presented.

I. INTRODUCTION

Electronic product evolution always follows the same path: perform more functions, do them faster, and be at a lower cost. This is coupled with pressure on design teams to reduce time-to-market and lower development costs. As a result, products need to be designed faster, have to be re-usable for multiple derivatives, and have to scale with growing performance demands. Increased performance is driven by two factors, computational complexity and data rates. More computational operations have to be performed on more data samples in a constant time window. Evidence for this trend can be found in application domains like video processing, wireless communications, and industrial automation. For video, high definition (HD) is the de-facto consumer standard. In order to not exceed the channel bandwidth for broadcasting, HD video is encoded with more advanced and computation intensive video codecs such as H.264. In the wireless domain, the symbol rates are increasing dramatically with the new LTE standard. In order to achieve these rates using the same frequency spectrum, more complex algorithms are used for encoding and decoding. For industrial automation, a robot microcontroller has to perform more operations per second to manufacture more products per day. In all examples frequency scaling is no longer an option. Either the design is already operating at the frequency limit or a higher frequency would violate power constraints. As a result, parallelization and specialization of hardware platforms must be employed to achieve the necessary energy efficient performance (MIPS/mW). This specialization implies that designs are fixed, which contradicts with another emerging requirement: flexibility. Video and wireless subsystems need to handle multiple standards and modes. Even after tape-out they need to be flexible enough to accommodate unforeseen updates and changes. A robot controller in an industrial assembly line cannot be limited to manufacture a specific product. Specialization also implies risks for the design. A single requirement that has not been captured exactly right during the specification phase may result in an unusable design. So called Application specific instruction-set processors (ASIPs) have emerged as a solid solution where specialization does not result in loss in flexibility, and this flexibility does not compromise performance and power goals.

II. APPLICATION SPECIFIC PROCESSORS

The term ASIP is widely used in industry and academia to classify programmable architectures that are tailored to a specific class of applications and algorithms. However, the attribute “application specific” does not necessarily refer to just a software application. Instead, it refers to the application of a processor in a specific system context, performing specific functions with distinct design goals. An ASIP has to be designed and implemented in a way that ensures the entire system meets its design goals. Often the instruction-set of the processor is considered to be the main attribute that makes a processor application specific. However, with the broader view of an application, two other characteristics become equally important: interfaces and micro-architecture. The interfaces define how the architecture interacts with the external control and communication interfaces. For data intensive algorithms, communication interfaces can quickly become a bottleneck if they do not map to the way data is consumed and produced by the algorithm and the surrounding system. Just tailoring the instruction-set of a RISC type architecture with highly parallel and specialized instructions is not enough if, for example, the load/store unit of the processor cannot handle data fast enough. The architecture of an interface can be just as important as the bandwidth. A load/store type of architecture already implies a scheme for how data is consumed and produced which may not reflect the way it is demanded by the algorithm or and surrounding system. Often, data has to be passed through a whole pipeline of functional blocks which are inter-connected via specialized FIFOs. The same holds true for the way data is processed internally in the ASIP. The pipeline of an ASIP needs to be designed in a way that matches the way data has to be processed to achieve the frequency goals. A traditional RISC fetch, decode, memory, execute, write-back scheme can become counterproductive for the design. ASIPs, e.g. for Turbo Decoding in LTE [1] can easily have 15 stages, each one of them containing dedicated processing units using internal and external interfaces.

Moreover, instead of organizing functional units in a typical processor hardware pipeline where each unit is assigned to a fixed stage, pipelining can be achieved through software. Here, parallel executed instructions are composing a "software pipeline". In a hardware pipeline each unit can only be scheduled in combination with the proceeding or following units. Scheduling can only be done with the delay of the proceeding stages. With software pipelining, all
functional units that need to operate in parallel reside in the same hardware pipeline stage. The instruction word contains a “slot”, where each slot corresponds to a sub-instruction that controls a dedicated set of units. Attributes of each slot define the input and output data routing between the functional units. This routing can happen through registers as done in Very Large Instruction Word (VLIW) architectures. In the case of a so called Transport Triggered Architecture (TTA) [2], the result of on unit is directly used as input of another in the next processing cycle. As every design decision comes with a trade-off, the drawback of parallel instructions is the size of the instruction word, impact program memory, interfaces as well as fetch and decode logic. In case of a VLIW architecture, complex bypass logic is often required between the slots. Furthermore, not all slots can be always used in context of a program. This can result in program code overhead.

In terms of the instruction format, ASIP instructions often do not match the classical format which is composed of a mnemonic and register/memory operands. Instruction operands define the internal communication between functional units, and the external communication using the ASIP interfaces. The number, format and type of instructions of an ASIP follow the flexibility requirements of the design. In general, instructions activate certain functional units in the design along with the configuration of register/memory address lines. A classical processor uses general purpose registers for inter-unit data flow. In case of a hardware design, the dataflow is mostly hardwired or controlled via configuration registers. An ASIP can implement both equally efficiently. For mostly constant operands, implicitly used configuration register might a better choice than general purpose registers as addresses do not need to be encoded in the instruction word. Registers are useful for frequently changing operand values. Constant operands, which are directly encoded in the instruction-words, save pre-loading them into a register. Finally, the best choice is depending on the characteristics of the application, specifically the dataflow within the time critical algorithm kernels.

To summarize, the term ASIP does not describe a certain type of processor architecture such as RISC, CISC, Superscalar or VLIW. ASIP refers to a design paradigm where the architecture is a result of the application and design goals. Here, the application does not only relate to the function, but also to the system context. ASIPs may more resemble a hardwired block than a processor from an architectural and interface perspective.

III. PROCESSOR DESIGNER

Synopsys Processor Designer (PD) [3] is a tool based solution for the design and implementation of ASIPs. PD takes a C based description (LISA [4], Language for Instruction-Set Architectures) of the processor as an input. As shown in Figure 1, PD automatically generates a simulation model, software programming and debugging-tools, as well as synthesizable RTL. The PD generated models and RTL feed seamlessly into RTL synthesis, RTL simulation, FPGA and virtual prototyping flows. A LISA model captures all aspects of the processor core architecture such as storage resources, interfaces, functional units, instruction-set encoding and assembly syntax. Functional units refer to all elements in the processor that contribute to functionality which can be data-processing, program control or interfacing. Functional units are described on a register transfer level using C as an input language. However, in contrast to C based algorithmic synthesis tools, the pipelining of functional units is explicitly described using dedicated language constructs. This way, the designer has full control over the resulting micro-architecture and area vs. performance design trade-offs.

From a LISA description, a full software programming tool chain can be generated. This tool chain comprises a compiler, assembler and linker. At each stage in the design, the hardware and software can be co-analyzed using a debugger that provides architectural as well as software analysis and debug information. The debugger supports various backends such as the automatically generated cycle accurate simulator as well as RTL simulation, FPGA or Virtual Prototypes. The ASIP’s RTL is automatically generated from LISA, including configurable debug logic and interfaces and can be mapped to an RTL simulator or FPGA. As virtual prototyping is becoming increasingly important to mitigate the risk of HW/SW integration errors in complex multi-core system, PD automatically generates SystemC TLM 2.0 compliant virtual prototyping models.

IV. LISA AND ASIP DESIGN

In this section, LISA will be briefly introduced along with a few small examples taken from an H.264 de-blocking filter design. A de-blocking filter is a post-processing filter within and video encoder and decoder that removes encoding artifacts which result from block based quantization. With full HD resolution, the de-blocking filter has become a major bottleneck in the critical path of a video encoder and decoder. A well optimized software filter implementation requires still a few hundreds of cycles on a RISC type architecture. However, real time constraints to filter full HD (1920x1088x30fps, 192 filter iterations per macro block) at ~300 Mhz do not allow for more than 5 cycles per filter iteration. Therefore, most de-blocking filter designs are implemented hardwired. However, each video standard requires slightly different variations and heuristics for the filtering process which result in complex design, expensive verification and risky designs.

This section will briefly introduce how an ASIP instruction-set, micro-architecture and memory architecture can be derived from a reference implementation.

A. Data load and store

In the reference implementation’s filter core, four “Q” pixels, are loaded from the pixel storage for one filter iteration (here, vertical edge). Afterwards, four “P” pixels are stored to the main memory after a horizontal edge filtering process. The pixels required for vertical edge filtering are organized in a row. The filtered pixels that are available after the horizontal edge filtering are organized in a column. The following example depicts the reference implementation for loading and storing pixels.

```plaintext
q[0] = pixel_mainmemory;  
pixel_mainmemory = pixel_mainmemory + next_column_increment;
q[1] = pixel_mainmemory;  
pixel_mainmemory = pixel_mainmemory + next_column_increment;  
```
The programmability of the load/store instruction is tailored to the requirements resulting from the de-blocking filter algorithm. There is no requirement to add more flexibility, since this would not result in any benefit. For example, it is not required to store Q pixels or to provide further addressing modes. The ability to initialize an arbitrary address increment is fully sufficient for this algorithm and allows adapting to arbitrary organizations of the pixels in the memory.

Additionally, two registers are required for the storage of the pixel address (PXADDR) and the pixel address increment (PIXINC). As a next step, the assembly format needs to be defined in order to give the programmer access to this instruction. Since the load instruction and store instruction is implicitly incrementing the pixel address, it is recommended to expose this to the programmer. Example 2 shows a possible instruction format and instruction sequence.

Example 2 - Instruction-set and sequence for pixels loading

Here, a pixel is loaded into register Q0 from the pixel memory at the address PXADDR. Then, the register PXADDR is incremented by PXINC. Thus, the next pixel is loaded from the next address.

B. LISA Implementation

After having introduced some few aspects of the instruction-set design, this section will focus on modeling using LISA. The given example is based on a de-blocking filter SIMD instruction that packs four 8-bit Q pixels into a 32-bit register. In LISA an instruction is modeled using one or multiple operations. Each operation captures some aspects of the instruction such as coding, syntax and behavior. In order to achieve pipelining, each operation can be assigned to a different pipeline stage. The PACK instruction is modeled using two operations, pack_q_db in stage DC, and pack_into_col (see Example 3, line 1 & 21) which is assigned to stage MEM. The ordering of stages has been declared in a called resource definition in the LISA model (not shown here) along with registers, memories and pins.

Example 3 - PACK instruction implementation in LISA

The lines 7 and 8 show the definition of the instruction encoding and assembly syntax. Line 9-15 capture the instruction behavior for the stage DC. Here four 8-bit pixels from the Q register file are packed and stored in the pipeline. Line 17 activates a group (choice) of operations modeling the row or column destination register file. The logic of those operations will be generated into the stage MEM and store the packed value from the pipeline in either the column or row register. The selection is made upon the instruction encoding as described in line 23 for the column register file. The pipeline stage assignment of the individual operations can easily be changed. This allows a late optimization of the pipeline towards an optimal fit (no data dependency delays) for the final instruction sequence as shown in Figure 2.
C. Programming, Debug and Simulation

At any stage in the design, programming tools such as compiler, assembler, linker can be generated to map the firmware into binary code for the ASIP. The compiler provides various ways to support complex instructions such as pattern matching, intrinsic or inline assembly. The generated cycle-accurate simulator allows the designer to validate the architecture and firmware from both a functional and performance standpoint. For debugging purposes, a graphical debugger exposes architectural details such as pipeline activity, resource accesses, and unit utilization, along with software performance information. Joint HW/SW analysis enables seamless root cause analysis from software down into the internals of the micro-architecture such as data hazards shown in Figure 2.

Example 4 – Modeling instruction level parallelism

Instead of having all instructions contained in a single exclusive group of instructions, all parallel instructions are assigned to two individual groups of instructions (slot_0 and slot_1) that are both contained in the VLIW instruction. The software pipelined kernel with parallel loading and filtering is shown in Example 5.

Example 5 – Software pipelined filter kernel

Due to parallelism in filtering and loading, a new register file pre-loading filter data QL has been also introduced.

V. DESIGN RESULTS

The H.264 de-blocking filter has been designed and verified with a cost of 12 man weeks. The final design consumed 57Kgates and was able to operate at 300Mhz. Next to the data centric de-blocking filter, another ASIP targeting the control dominated motion vector decoding has been developed. As a reference, an existing hardwired motion vector decoder implementation using 45Kgates, required 100 cycles per macro-block and 1mW peak power. In contrast, the 50KGate ASIP required 216 cycles per macro-block, 3mW peak power, but was able to serve multi-codes and multi-format including J-PEG, MPEG 1-4, H.264, VC-1 and VP6. The motion vector decoder ASIP was able to serve multi-codes and multi-format including J-PEG, MPEG 1-4, H.264, VC-1 and VP6. The motion vector decoder ASIP supports full-HD (1900x1088@30fps) at 200Mhz. The ASIP has been design within 15 man weeks, including specification and verification time.

VI. SUMMARY

Today’s design automation tools turn ASIPs into a design solution that can be implemented at a competitive cost compared to hardwired logic. Cost refers to the cost of designing as well as the cost of the design. In contrast to hardwired logic, ASIPs provide flexibility which allows longer and broader use of the design in the market. Moreover, ASIPs mitigate design risks. Complexity is shifted from silicon into software which can be changed even if silicon is at hand. ASIP design automation tool suites, such as the Synopsys Processor Designer, eliminate the software tool creation overhead that is introduced by changing from hardwired to programmable solutions.