Hardware/Software Codesign Across Many Cadence Technologies

33

Grant Martin, Frank Schirrmeister, and Yosinori Watanabe

Abstract

Cadence offers many technologies and methodologies for hardware/software codesign of advanced electronic and software systems. This chapter outlines many of these technologies and provides a brief overview of their key use models and methodologies. These include advanced verification, prototyping – both virtual and real, emulation, high-level synthesis, design of an Application-Specific Instruction-set Processor (ASIP), and software-driven verification approaches.

Acronyms

ADAS	Advanced Driver Assistance System
API	Application Programming Interface
ASIC	Application-Specific Integrated Circuit
ASIP	Application-Specific Instruction-set Processor
AXI	Advanced eXtensible Interface
CNN	Convolutional Neural Network
CPF	Common Power Format
DMA	Direct Memory Access
DSP	Digital Signal Processor
DUT	Design Under Test
ECO	Engineering Change Order
EDA	Electronic Design Automation
ESL	Electronic System Level
FFT	Fast Fourier Transform
FIFO	First-In First-Out
FPGA	Field-Programmable Gate Array

G. Martin (✉) • F. Schirrmeister • Y. Watanabe
Cadence Design Systems, San Jose, CA, USA
e-mail: gmartin@cadence.com; franks@cadence.com; watanabe@cadence.com

S. Ha, J. Teich (eds.), *Handbook of Hardware/Software Codesign*,
DOI 10.1007/978-94-017-7267-9_33

HLS	High-Level Synthesis
HSCD	Hardware/Software Codesign
HVL	Hardware Verification Language
HW	Hardware
IDE	Integrated Development Environment
IP	Intellectual Property
ISA	Instruction-Set Architecture
ISS	Instruction-Set Simulator
JTAG	Joint Test Action Group
LISA	Language for Instruction-Set Architectures
MAC	Multiply-Accumulator
NoC	Network-on-Chip
OFDM	Orthogonal Frequency Dependent Multiplexing
OS	Operating System
OVM	Open Verification Methodology
PCI	Peripheral Component Interconnect
PC	Personal Computer
RISC	Reduced Instruction-Set Processor
RTL	Register Transfer Level
SDK	Software Development Kit
SDS	System Development Suite
SIMD	Single Instruction, Multiple Data
SW	Software
TIE	Tensilica Instruction Extension
TLM	Transaction-Level Model
UML	Unified Modeling Language
UPF	Unified Power Format
USB	Universal Serial Bus
UVM	Universal Verification Methodology
VLIW	Very Long Instruction Word
VMM	Verification Methodology Manual
VP	Virtual Prototype
VSIA	Virtual Socket Interface Alliance
VSP	Virtual System Platform

Contents

33.1 Overview

Over the last couple of decades, the complexities of chip design have risen significantly. Where in 1995 reuse of Intellectual Property (IP) blocks was just starting and led to the foundation of the Virtual Socket Interface Alliance (VSIA) [5] in 1996, promoting IP integration and reuse, design teams are now facing the challenge of integrating hundreds of IP blocks. In 1996, most of the effort directly associated with chip design was focused on hardware itself, but since then the effort to develop software has become a budgetary item that can, depending on the application domain, dominate the cost of the actual chip development.

The Electronic Design Automation (EDA) industry responded quite early. Synopsys Behavioral Compiler, an early foray into high-level synthesis, was introduced in 1994 and Aart De Geus optimistically predicted a significant number of tape-outs before the year 2000. Gary Smith created the term Electronic System Level (ESL) in 1996, the same year that the VSIA was founded. In 1997 Cadence announced the Felix Initiative [17], which promised to make function-architecture codesign a reality. The SystemC [12] initiative was formed in 1999 to create a new level of abstraction above Register Transfer Level (RTL), but was initially plagued by remaining tied to the signal level until 2008, when the standardization of the TLM-2.0 Application Programming Interfaces (APIs) was completed. This helped interoperability for virtual platforms (also known as a Virtual Prototype (VP)) and made SystemC a proper backplane for IP integration at the transaction level. For another view on virtual prototypes, please consult ▸ Chap. 34, "Synopsys Virtual Prototyping for Software Development and Early Architecture Analysis".

When it comes to system and SoC design, at the time of this writing in 2016, the industry has certainly moved up in abstraction, but in a more fragmented way than some may have expected 20 years ago. The fundamental shortcoming of the assumptions of 1996 was the idea that there would be a single executable specification from which everything could be derived and automated. What happened instead is that almost all development aspects moved upward in abstraction, but in a fragmented way, not necessarily leading to one single description from which they can all be derived. As designers moved up in abstraction, three separate areas emerged – IP blocks, integration of IP blocks, and software.

For IP blocks, i.e., the new functions to be included into hardware and software, there is a split between IP reuse and IP development. With full-chip high-level synthesis never becoming a reality, IP reuse really saved the day, by allowing design teams to deal with complexities. It has developed into a significant market today. For IP development, there are six basic ways to implement a great new idea:

1. Manually implement in hardware.
2. Use high-level synthesis to create hardware.
3. Use an extensible or configurable processor core to create a hardware/software implementation.
4. Use tools to create a design of an Application-Specific Instruction-set Processor (ASIP).
5. Use software automation to create software from a system model.
6. Manually implement software and run it on a standard processor.

Interestingly enough, the nonmanual cases two to five all use higher-level descriptions as the entry point, but each one is different. High-level synthesis is driven by transaction-level descriptions in SystemC or C/C++, ASIPs as both IP and an associated tool flow are generated using specific language-like descriptions such as nML, Language for Instruction-Set Architectures (LISA), or the Tensilica Instruction Extension (TIE) description language [22]. Software can be auto-generated from Unified Modeling Language (UML) and MatLab/Simulink descriptions. The closest high-level unifying notations for a complete hardware/software system are SysML [10] or UML [16], as well as proprietary offerings such as MathWorks Simulink, from which both hardware blocks and software blocks can be generated automatically.

When it comes to connecting all the hardware blocks together, regardless of whether they were reused or built with one of the six options above, the user has five different options:

1. Connect blocks manually (good luck!).
2. Automatically assemble the blocks using interconnect auto-generated by ARM AMBA Designer, Sonics, Arteris, or another interconnect IP provider.
3. Synthesize protocols for interconnect from a higher-level protocol description.
4. Create a Network-on-Chip (NoC), such as a mesh NoC.
5. Use a fully programmable NoC that determines connections completely at run time.

Again, with the exception of the first (manual) and last (at run time) way to create the interconnect, the other items raise the level of abstraction. The ARM Socrates [23] and AMBA Designer environments feed information into Cadence tools such as Interconnect Workbench to set up a scenario for which performance analysis is needed, and there are specific tools to automatically create configurations of different interconnect topologies from higher-level descriptions as well.

Figure 33.1 illustrates the different methods of IP creation and integration, and the following sections of this chapter dive more deeply into two aspects – high-level synthesis using the Cadence Stratus high-level synthesis environment and the development of extensible processor cores using the Tensilica Xtensa technology. A third aspect is the software that can be found in these designs, much of it actually determining the functionality and the architecture of a chip. Efforts to achieve continuous integration of hardware and software have created what the

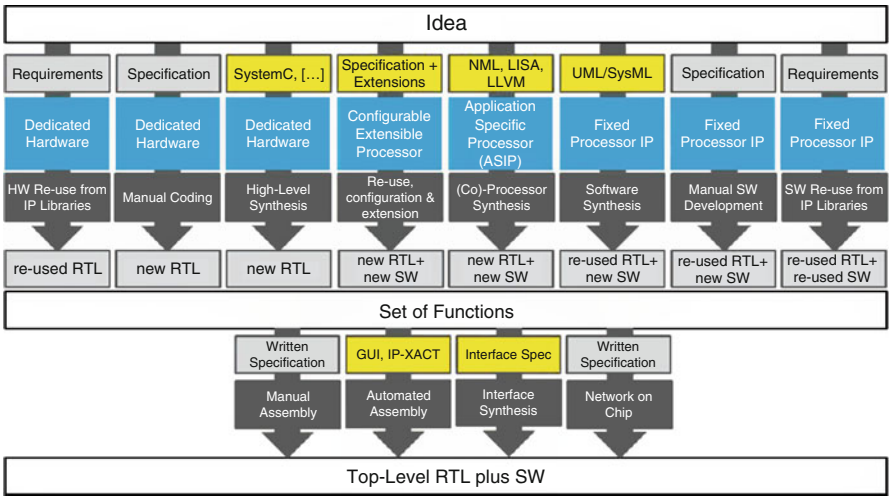

Fig. 33.1 IP creation and integration in modern chip design

industry refers to as a "shift left" – essentially, early representations or models of the hardware allowing some level of software execution to occur on the models. During a project flow today, shifting left has created various options for development vehicles on which to bring up and execute software:

1. Software Development Kits (SDKs), which do not model hardware in complete detail.
2. Virtual platforms that are register accurate and represent functionality of the hardware accurately, but without timing. Architectural virtual platforms may add cycle accuracy as a modeling style, slowing down execution, thus offering users a trade-off between speed and accuracy for architectural analysis.
3. RTL simulation is technically a representation of the hardware but is not often used for software development, unless for low-level drivers.
4. Emulation is the first platform that allows execution in the MHz range. Using emulation, users can run AnTuTu on mobile devices and bring up Linux on server chips. The intent is mainly to verify and optimize the hardware.
5. FPGA-based prototyping executes in the tens of MHz range, at times up to 100 MHz, and is a great vehicle for software development on accurate hardware.
6. The actual chip is often used in development boards to develop software.

All options except the last one use abstraction in one way or another to enable software development as early as possible. The trade-offs are time of availability during development, speed, and accuracy and the incremental effort needed for development of the development vehicle.

In many cases, hardware must take the role of executing software in the best possible way. This is why users deploy emulation and FPGA-based prototyping to

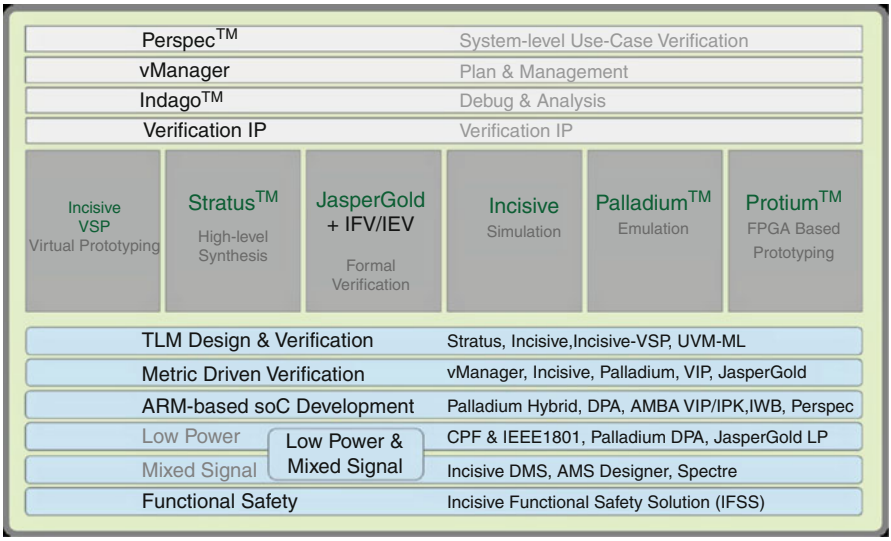

Fig. 33.2 System Development Suite

actually run mobile benchmarks like AnTuTu, as well as server benchmarks. The results help designers make changes to the design before finalizing it, to optimize performance, power, and thermal characteristics. So in a sense hardware/software codesign has become somewhat of a reality, but needs to be looked at across a family of platforms – some changes may not make it into the current design as it needs to be rolled out to meet time to market. They make it instead into the next derivative design.

Figure 33.2 shows the Cadence System Development Suite (SDS). This offers a continuous integration of development engines for verification and software development. The hardware-assisted aspects, to enable the industry demand for a shift left of software development, will be described in the following sections of this chapter.

The lessons of the last 20 years are twofold. First, no single human being is capable of comprehending all aspects of the hardware/software mix in order to generate a unified description. Complexity has simply grown too much and will continue to do so for high-end designs. The industry is just at the beginning of describing scenarios at higher levels of abstraction that can be used to allow team members with different expertise to efficiently interact. Work in Accellera on Portable Stimulus (see later details) looks promising, in defining scenarios for software-driven testing.

Second, for use cases such as performance analysis and power optimization, abstraction really has only provided a partial answer to the problem. When accuracy and predictability of the actual implementation is required, implementation really matters, to drive early design decisions, and execution at the RTL, or the level of cycle-accurate SystemC, with models abstracted from the implementation flow are predominant in order to determine power and performance. An example is a

combination of activity data gathered from RTL simulation and emulation with power characterizations abstracted from implementation representations such as .lib files, as in the combination of Cadence Palladium emulation and Cadence Joules power estimation. In contrast – for tasks such as software development of drivers in a low-power context – abstraction offers a solution using Transaction-Level Models (TLMs), sometimes combined in a hybrid fashion with RTL representations, that allows early functional verification of software, ignoring some of the detailed accuracy requirements. Examples are TLM virtual platforms annotated with low-power information and hybrid configuration of virtual platforms with emulation. As a result design teams are entering an era of both horizontal integration and vertical integration.

Horizontal integration enables verification on different engines in the flow using the same tests, sometimes referred to as "portable stimulus" as currently standardized in the Accellera working group of the same name [2]. Here are found UML-like descriptions, notations, and languages that describe scenarios. This is the next level above SystemVerilog for verification and definitely will be a hallmark of verification in the next decade, when verification shifts to the system level and designers have to rely on IP being largely bug-free. IP itself also will rise from the block level to subsystems, so the pieces to be integrated are getting bigger. The flow between the horizontal engines and hybrid engine combinations will also grow further in popularity.

Vertical integration keeps us grounded and may be the main obstacle in the way of a unified high-level design description. While in the days of the Felix Initiative, the team operated under the assumption that everything can be abstracted to enable early design decisions, it turns out that is not the case in reality. Performance analysis for chip interconnect has dropped down back to the RTL, or in the case of architectural virtual platforms, to the cycle-accurate SystemC level, simply because the pure transaction level does not offer enough accuracy to make the right performance decisions. Tools like Cadence Interconnect Workbench [13] are addressing this space today and vertically integrate higher-level traffic models with lower-level RTL and SystemC representations. The same is true for power. Abstracting power states to annotate power information to transaction-level models in virtual prototypes may give enough relative accuracy to allow development of the associated software drivers, but to get estimates accurate enough to make partitioning decisions, one really needs to connect to implementation flows and consider dynamic power. The integration of Palladium emulation with Joules power estimation from the RTL is a good example here.

Bottom line, today and for the years to come, design teams will deal with blocks to be integrated that will grow into subsystems; there will be even smarter interconnects to assemble systems on chip; and software development will have shifted left earlier. However, the separation of reuse (grown to subsystems), automatic creation (High-Level Synthesis (HLS)), and chip assembly (watch the space of integration and verification automation), plus the creation of early representations of the hardware to enable software development, will still be the predominant design techniques for very complex designs. The following sections will give more details on some of the areas touched above.

The rest of this chapter is organized as follows:

- Section 33.2 talks about the System Development Suite.
- Section 33.3 talks about virtual prototyping and hybrid execution
- Section 33.4 talks about hardware accelerated execution in emulation and FPGA-based prototyping.
- Section 33.5 talks about high-level synthesis technology.
- Section 33.6 talks about Application-Specific Instruction-set Processor technology.
- Section 33.7 talks about software-driven verification and portable stimulus.
- Section 33.8 concludes the chapter with an eye to future technology development.

33.2 System Development Suite

As indicated in the overview, a classic design flow for hardware/software projects is divided into creation, reuse, and integration of IP. Figure 33.3 shows some of the main development tasks during a project.

The horizontal axis shows the hardware-related development tasks starting with specification, IP qualification and integration, and implementation tasks prior to tape-out and chip fabrication. The vertical axis indicates development scope from hardware IP blocks through subsystems, System on Chips (SoCs), and the SoC in the actual end product (system) through software from bare-metal tasks to operating systems and drivers, middleware, and the user-facing applications.

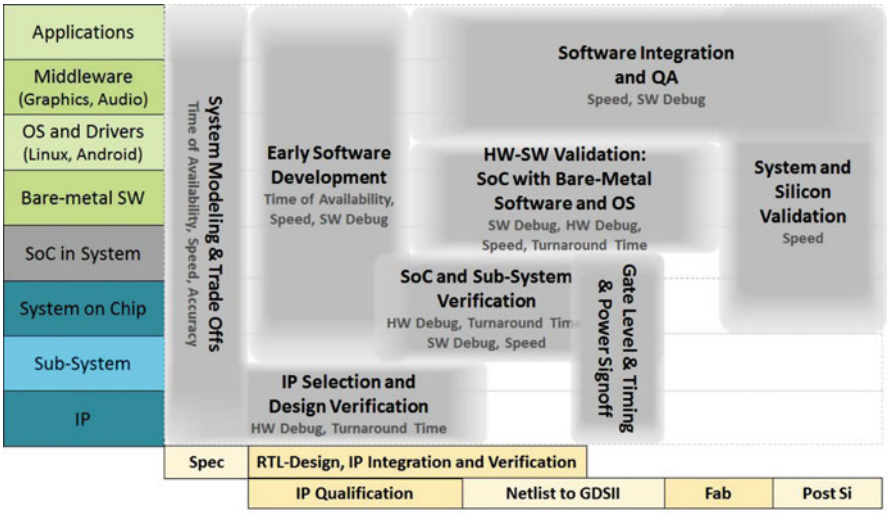

Fig. 33.3 Development tasks during a hardware/software development project

Development starts with system modeling and trade-off analysis executed by architects resulting in specifications. For system models, time of availability, speed, and accuracy are most important. Hardware development and verification is performed by hardware verification engineers for IP, subsystems, and the SoC. Initially, hardware debug and fast turnaround time are most important; once software enters the picture for subsystem verification, software debug and execution speed also become crucial. Software development happens in two main areas: hardware-aware software development for Operating System (OS) porting and utility development and application software development, requiring various levels of speed and model accuracy. The integration of hardware and software needs to be validated by HW/SW validation engineers prior to tape-out and again on silicon once actual chip samples are available. This flow can take 18–24 months; one of the major objectives is to allow agile, continuous integration of hardware and software, so developers use different execution engines and different combinations of these engines as soon as they become available.

As one can see, today's complex hardware/software designs involve many different types of developers, all with different requirements and concerns that cannot be satisfied by one engine alone. Here are the five main types of users:

1. **Application software developers** need a representation of the hardware as early as possible during a project. The representation needs to execute as fast as possible and needs to be functionally accurate. This type of software developer would like to be as independent from the hardware as possible and specifically does not need full timing detail. For example, detailed memory latency and bus delays are generally not of concern, except for specific application domains for which timing is critical.
2. **Hardware-aware software developers** would also like representations of the hardware to be available as early as possible. However, they need to see the details of the register interfaces, and they expect the prototype to look exactly like the target hardware. Depending on their task, timing information may be required. In exchange, this type of developer is likely to compromise on execution speed to gain the appropriate accuracy.
3. **System architects** care about early availability of the prototype, as they have to make decisions before all the characteristics of the hardware are defined. They need to be able to trade off hardware versus software and make decisions about resource usage. For them, the actual functionality counts less than some of the details. For example, functionality can be abstracted into representations of the traffic it creates, but for items like the interconnect fabric and the memory architecture, very accurate models are desirable. In exchange, this user is willing to compromise on speed and typically does not require complete functionality as the decisions are often made at a subsystem level.
4. **Hardware verification engineers** typically need precise timing accuracy of the hardware, at least on a clock cycle basis for the digital domain. Depending on the scope of their verification task, they need to be able to model the impact of software as it interacts with the hardware. In some cases they need to

assess mixed-signal effects at greater accuracy than standard cycle accurate RTL provides. Accuracy is considered as more important than speed, but the faster the prototype executes, the better the verification efficiency will be. This user also cares about being able to reuse test benches once they have been developed, across engines, to allow verification reuse.

5. **Hardware/software validation engineers** make sure the integration of hardware and software works as specified, and they need a balance of speed and accuracy to execute tests of significant length to pinpoint defects if they occur. This type of user especially needs to be able to connect to the environment of the chip and system to verify functionality in the system context.

Some characteristics are important to all users, but some of them are especially sensitive to some users. Cost is one of those characteristics. While all users are cost sensitive, software developers may find that a development engine may not be feasible in light of cheaper alternatives, even though the engine may have the desired accuracy or early availability in the project flow. In addition, the extra development effort that engines require beyond standard development flows needs to be considered carefully and weighed against benefits.

Figure 33.4 illustrates some of the dynamic and static development engines with their advantages and disadvantages.

The types of development engines can be categorized easily by when they become available during a project. Prior to RTL development, users can choose from the following engines:

- **SDKs** typically do not run the actual software binary but require recompilation of the software. The main target users are application software developers who do not need to look into hardware details. SDKs offer the best speed but lack accuracy. The software executing on the processors as in the examples given earlier runs natively on the host first or executes on abstraction layers like Java.

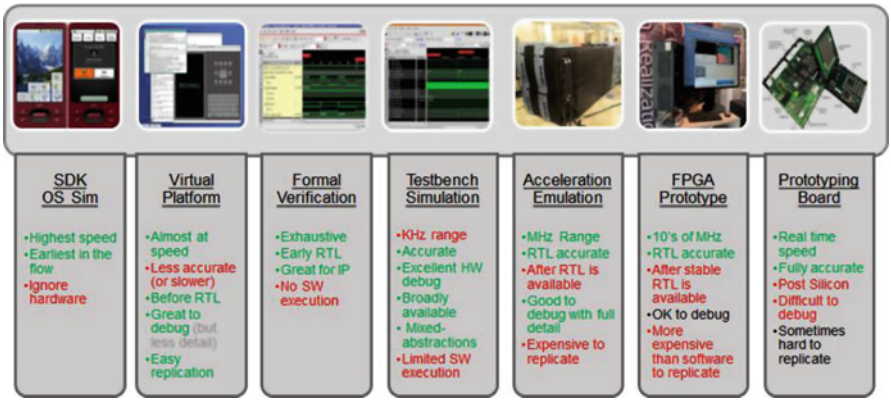

Fig. 33.4 Hardware/software development engines

Complex computation as used in graphics and video engines is abstracted using high-level APIs that map those functions to the capabilities of the development workstation.

Virtual platforms can be available prior to RTL when models are available and come in two flavors:

- **Architectural virtual platforms** are mixed accuracy models that enable architecture decision-making. The items in question – bus latency and contention, memory delays, etc. – are described in detail, maybe even as small portions of RTL. The rest of the system is abstracted as it may not exist yet. The main target users are system architects. Architectural virtual platforms are typically not functionally complete, and they abstract environment functionality into their traffic. Specifically, the interconnect fabric of the examples given earlier will be modeled in full detail, but the analysis will be done per subsystem. Execution speed may vary greatly depending on the amount of timing accuracy, but normally will be limited to tens to low hundreds of KHz. Given that cycle-accurate SystemC can be as accurate as RTL, minus sub-cycle timing annotations, automatic translation from RTL to SystemC is sometimes used: technologies like Verilator and ARM Cycle Model Studio are useful here.
- **Software virtual platforms** run the actual binary without recompilation at speeds close to real time – fifties to hundreds of MHz. Target users are software developers, both application developers and "hardware-aware software developers." Depending on the needs of the developer, some timing of the hardware may be more accurately represented. This prototype can be also used by hardware/software validation engineers who need to see both hardware and software details. Due to the nature of "just in time binary translation," the code stream of a given processor can be executed very fast, natively on the host. This makes virtual prototypes great for software development, but modeling other components of the example systems – such as 3D engines – at full accuracy would result in significant speed degradation.

Once RTL has been developed, RTL-based engines offer more accuracy:

- **RTL simulation** is the standard vehicle for hardware verification engineers. Given its execution in software, it executes slowly – in the range of hundreds of Hz – for all components in the system to be represented. It sometimes is used as an engine for lower-level software development for which great accuracy is required and appropriate length of execution can be achieved due to short simulation runs.
- **Simulation acceleration**: When RTL simulation becomes too slow, acceleration allows users to bring performance to the next orders of magnitude – 200 to 500 KHz. Acceleration is a mix of software-based and hardware-based execution. Interfaces to the real world are added, but selectively.

- **In-circuit emulation**: Now everything transitions into the emulator, test benches are synthesizable or the software executes as it will in the end product and users get even more speed – 1 to 2 MHz. Debug – especially for hardware – is great in emulation. More interfaces to the real world are added. For both *in-circuit emulation* and *acceleration*, the speed is much superior to basic RTL simulation and as such very balanced. However, when it comes to pure software execution on a processor, transaction-level models of a processor on a Personal Computer (PC) will execute faster.
- **FPGA-based prototyping**: When RTL has become mature, users can utilize Field-Programmable Gate Array (FPGA)-based platforms as even faster hardware-based execution environments. This works especially well for IP that already exists in RTL form. Real-world interfaces are now getting to even higher speeds of tens of MHz. Similarly to *acceleration* and *in-circuit emulation*, pure software execution on a processor, or transaction-level models of a processor on a PC, may still execute faster.

Finally, software development also happens on real silicon and can be split into two parts:

- Chips from the last project can be used especially for application development. This is like the SDK in the pre-RTL case. However, the latest features of the development for the new chip are not available until the appropriate drivers, OS ports, and middleware become available.
- Once the chip is back from fabrication, actual silicon prototypes can be used. Now users can run at real speed, with all connections, but debug becomes harder as execution control is not trivial. Starting, stopping, and pausing execution at specific breakpoints is not as easy as in software-based execution and prototypes in FPGA and acceleration and emulation.

To understand the benefits associated with each type of development engine, it is important to summarize the actual concerns derived from the different users and use models:

- **Time of availability during a project**: When can I get it after project start? Software virtual prototypes win here as the loosely timed transaction-level model (TLM) modeling effort can be much lower than RTL development and key IP providers often offer models as part of their IP packages. Hybrid execution with a hardware-based engine alleviates remodeling concerns for IP that does not yet exist as TLMs.
- **Speed**: How fast does the engine execute? Previous generation chips and actual samples execute at actual target speed. Software virtual prototypes without timing annotation are next in line, followed by FPGA-based prototypes and *in-circuit emulation* and *acceleration*. Software-based simulation with cycle accuracy is much slower.

- **Accuracy**: How detailed is the hardware that is represented compared to the actual implementation? Software virtual prototypes based on TLMs with their register accuracy are sufficient for a fair number of software development tasks including driver development. However, with significant timing annotation, speed slows down so much that RTL in hardware-based prototypes often is faster.
- **Capacity**: How big can the executed design be? Here the different hardware-based execution engines differ greatly. Emulation is available in standard configurations of up to several billion gates; standard products for FPGA-based prototyping are in the range of several hundreds of millions of gates, as multiple boards can be connected for higher capacity. Software-based techniques for RTL simulation and virtual prototypes are only limited by the capabilities of the executing host. Hybrid connections to software-based virtual platforms allow additional capacity extensions.
- **Prototyping development cost and bring-up time**: How much effort needs to be spent to build it on top of the traditional development flow? Here virtual prototypes are still expensive because they are not yet part of the standard flow. Emulation is well understood and bring-up is very predictable: in the order of weeks. FPGA-based prototyping from scratch is still a much bigger effort, often taking 3–6 months. Significant acceleration is possible when the software front end of emulation can be shared.
- **Replication cost**: How much does it cost to replicate the prototype? This is the actual cost of the execution vehicle, not counting the bring-up cost and time. Pricing for RTL simulation has been under competitive pressure and is well understood. TLM execution is in a similar price range; the hardware-based techniques of emulation and FPGA-based prototyping require more significant capital investment and can be measured in dollars per executed gate.
- **Software debug, hardware debug, and execution control**: How easily can software debuggers be attached for hardware/software analysis and how easily can the execution be controlled? Debugger attachment to software-based techniques is straightforward and execution control is excellent. The lack of speed in RTL simulation makes software debug feasible only for niche applications. For hardware debug the different hardware-based engines are differentiated – hardware debug in emulation is very powerful and comparable to RTL simulation, but in FPGA-based prototyping it is very limited. Hardware insight into software-based techniques are great, but the lack of accuracy in TLMs limits what can be observed. With respect to execution control, software-based execution allows one to efficiently start and stop the design, and users can selectively run only a subset of processors, enabling unique multi-core debug capabilities.
- **System connections**: How can the environment be included? In hardware, rate adapters enable speed conversion, and a large number of connections are available as standard add-ons. RTL simulation is typically too slow to connect to the actual environment. TLM-based virtual prototypes execute fast enough and virtual I/O to connect to real-world interfaces such as Universal Serial Bus (USB), Ethernet, and Peripheral Component Interconnect (PCI) have become a standard feature of commercial virtual prototyping environments.

- **Power analysis**: Can users run power analysis on the prototype? How accurate is the power analysis? With accurate switching information at the RTL level, power consumption can be analyzed fairly accurately, especially when vertically integrated with implementation flows. Emulation adds the appropriate speed to execute long enough sequences to understand the impact of software. At the TLM level, annotation of power information allows early power-aware software development, but the results are by far not as accurate as at the RTL level.
- **Environment complexity**: How complex are the connections between the different engines? The more hardware and software engines are connected (as in acceleration), the complexity can become significant and hard to handle, which needs to be weighed against the value.

Given the different types of users and their needs, the different engine capabilities, and the different concerns for the various development tasks, it is easy to see that there is no one "super"-engine that is equally suited for all aspects. Introduced in 2011, the System Development Suite is a set of connected development engines and has since then been enhanced to achieve closer integration between the engines as illustrated in Fig. 33.5.

The System Development Suite is the connection of dynamic and static verification platforms and starts with the Stratus HLS platform for IP development which is also used to raise the level of verification abstraction. The JasperGold formal verification platform is widely used throughout the flow with its different formal applications, ranging from block to SoC level. The Incisive platform for advanced verification extends from IP level to full SoCs and interacts with the Palladium acceleration and emulation platform quite seamlessly, with technologies such as hot swap between simulation and emulation.

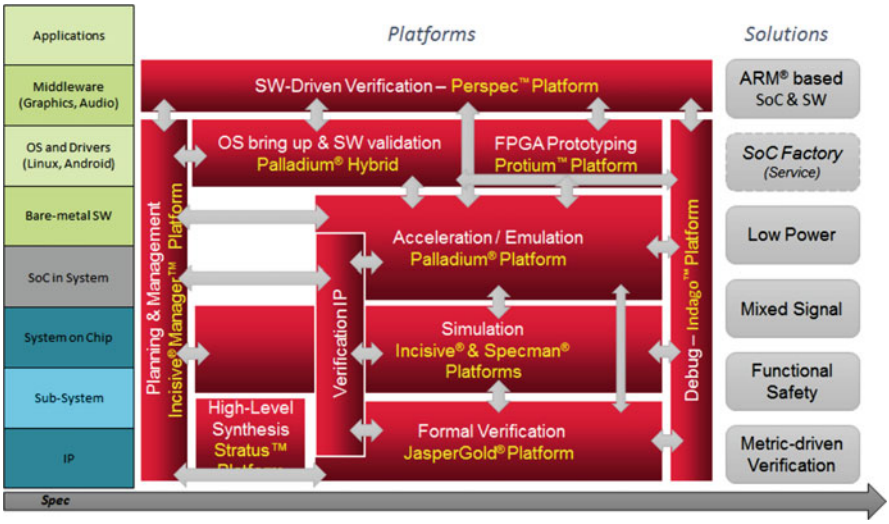

Fig. 33.5 System Development Suite engine integrations

The different engines tie together into the vManager verification center to collect and assess coverage, planning and monitoring how well verification proceeds throughout a project. Verification IP is usable across the different platforms, and with debug enabled by the Indago platform, the suite is being worked toward unified debug across the different verification engines.

Extending further into software development, the Palladium Hybrid technology connecting virtual platforms with emulation and the Protium FPGA-based prototyping technology enable software development at various levels of speed and hardware accuracy. The Perspec platform for use-case-driven verification allows the development of stimulus that is portable across the different dynamic verification engines.

Finally, there are specific solutions that combine the different engines to optimize development for ARM-based designs and low-power, mixed-signal, functional safety, and metric-driven verification. The SoC factory service enables the automation of integration and verification of IP-based designs with interfaces to IP-XACT and ARM's Socrates [23] tools.

Two system-level aspects – behavioral modeling and design space exploration – have attracted the attention of researchers for the better part of the last two decades, but so far have not become broadly supported in commercial tools. The adoption of behavioral modeling itself has been limited due to the absence of a universally accepted higher-level system language or representation. SystemC – while well adopted as an entry point for high-level synthesis and as glue for the assembly of virtual platforms, utilizing back-door interfaces as provided in SystemC TLM-2.0 APIs – has not been found suitable for higher-level descriptions. For these, proprietary techniques such as provided by National Instruments and the MathWorks and standardized entries like SysML or UML are more common. They cater to system architects and abstract both hardware and software.

In the context of the System Development Suite, SystemC is supported natively as part of multiengine simulation, while higher-level descriptions serve as references for verification with connections of MatLab/Simulink models into verification. In addition, UML style diagrams have become one option to describe system-level test scenarios to create portable stimulus that can be executed as software in multiple verification engines.

33.3 Virtual Prototyping and Hybrid Execution with RTL

Virtual prototyping was pioneered by start-ups like VasT, Virtutech, and Virtio, all of which were acquired in the last decade. It turns out that the modeling effort often is considered so high that these days "pure virtual prototypes" at the transaction level have become somewhat unusual, and mixed abstraction-level virtual prototypes, combining TLM and RTL have become predominant. Figure 33.6 shows the advantages of the different engines across the user concerns introduced in the previous section, showing clearly how the speed of virtual platforms, combined with the accuracy of RTL-based execution engines, can be advantageous.

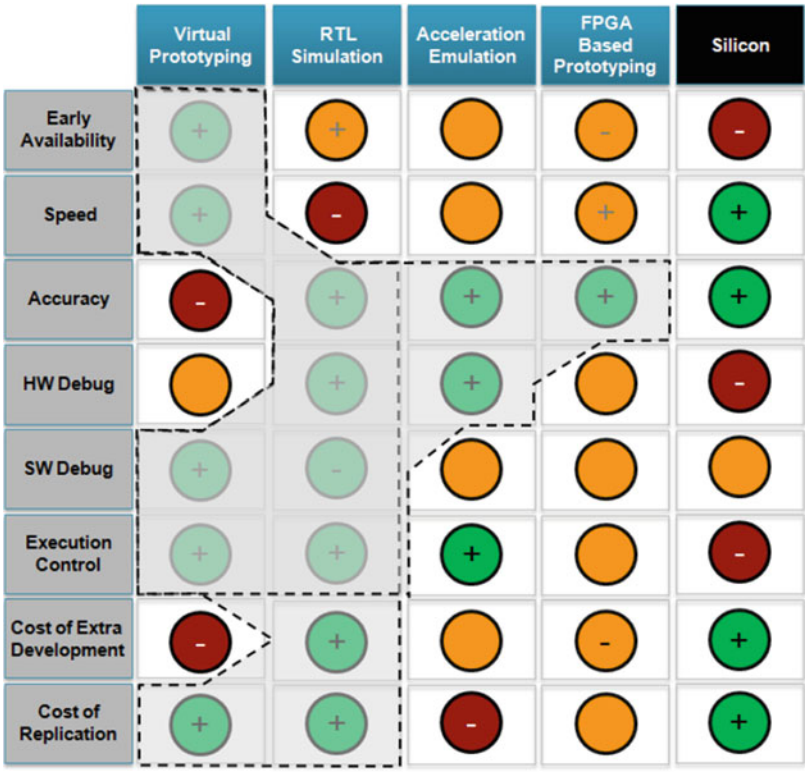

Fig. 33.6 Advantages of hybrid engine combinations

The combination of RTL simulation and virtual prototyping is especially attractive for verification engineers who care about speed and accuracy in combination. Software debug may be prohibitively slow on RTL simulation itself, but when key blocks including the processor can be moved into virtual prototype mode, the software development advantages can be utilized and the higher speed also improves verification efficiency.

The combination of emulation/acceleration and virtual prototyping is attractive for software developers and hardware/software validation engineers when processors, which would be limited to the execution speed of emulation or FPGA-based prototyping when mapped into hardware-based execution, can be executed on a virtual prototype. Equally, massive parallel hardware execution – as used in video and graphics engines – is executed faster in hardware-based execution than in a virtual prototype. For designs with memory-based communication, this combination can be very advantageous, calling graphics functions in the virtual prototype and having them execute in emulation or FPGA-based prototyping.

With hybrid techniques, users can achieve a greatly reduced time delay before arriving at the "point of interest" during execution, by using accelerated OS boot

(operating system boot-up). Billions of cycles of an operating system (OS) have to be executed before software-based diagnostics can start; therefore, OS boot itself becomes the bottleneck. The Palladium Hybrid solution combines Incisive-VSP virtual prototyping and ARM Fast Models with Palladium emulation to provide this capability.

Users such as NVIDIA [8], ARM, and CSR [20] have seen overall speedup of tests by up to ten times, when combining graphical processor unit (GPU) designs together with ARM Fast Models representing the processor subsystem. They demonstrated up to two hundred times acceleration of "OS boot," which brought them to the point of interest much faster than by using pure emulation. The actual speedup depends on the number of transactions between the TLM and RTL domains. The time to the point of interest can be accelerated significantly because during OS boot, the interaction between the TLM simulation and RTL execution in emulation (which limits the speed) is fairly limited. When the actual tests run after the OS is booted, the speedup depends again on how many interactions and synchronizations are necessary between the two domains. Some specific smart memory technology in the Palladium Hybrid solution with Virtual System Platform (VSP) and ARM Fast Models allows synchronization between both domains to be more effective (the concept can be likened to an advanced form of caching). Still, tests get accelerated the most when they execute a fair share of functionality in software.

33.4 Hardware Accelerated Execution in Emulation and FPGA-Based Prototyping

As pointed out earlier, software-based execution is limited by the number of events executed and hence has speed limitations. When considering hardware-based execution techniques, a key measure is the throughput for a queue of specific tasks, comprised of compile, allocation, execution, and debug.

Given thousands of verification and software development tasks, it is important to consider how fast the user can compile the design to create an executable of the job that then can be pushed into the execution queue. In emulation, these tasks are automated and for processor-based emulation, users compile for the latest Palladium Z1 emulation platforms at a rate of up to 140 million gates per hour, getting to results quite quickly. For simulation, the process is similar and fast. For FPGA-based prototyping, it may take much longer for manual optimization to achieve the highest speeds, often weeks if not months. Flow automation for the Protium platform, adjacent to Palladium, allows users to trade-off between bring-up and execution speed. The benefit of fast bring-up is offset by speeds between 3 and 10 MHz, not quite as fast as with manual optimization that often results in speeds of 50 MHz or more.

Allocation of tasks into the hardware platform determines how efficiently it can be used as a compute resource. For simulation farms, users are mostly limited by the number of workstations and the memory footprint. Emulation allows multiple

users, but the devil lies in the details. For large numbers of tasks of different sizes, the small granularity and larger number of parallel jobs really tips the balance here toward processor-based emulation such as the Palladium Z1 platform. In contrast the number of users per FPGA platform is typically limited to one.

The actual execution speed of the platform matters, but cannot be judged in isolation. Does the higher speed of FPGA-based prototyping make up for the slower bring-up time and the fact that only one job can be mapped into the system? It depends. As a result FPGA-based prototyping is mainly used in software development, where designs are stable and less in hardware verification. This usage is later in the cycle, but runs faster. For FPGA-based emulation, often considered faster than processor-based emulation, users have to look carefully how many jobs can be executed in parallel. And in simulation farms, the limit is really the availability of server capacity and memory footprint. The Palladium Z1 platform introduced in late 2015 is an enterprise emulation platform scalable to 9.2 billion gates for up to 2304 parallel tasks.

As the last steps of the throughput queue, debug is crucial. It is of the utmost importance to efficiently trigger and trace the debug data for analysis. FPGA-based prototyping and FPGA-based emulation slow down drastically when debug is switched on, often negating the speed advantages for debug-rich cases found when RTL is less mature. It all depends on how much debug is needed, i.e., when in the project phase the user is running the verification queue set up above. In addition, the way data is extracted from the system determines how much debug data is actually visible. Also, users need to assess carefully how the data generation slows down simulation. With processor-based emulation, debug works in a simulation-like manner. For FPGA-based systems, slowdown and accessibility of debug data need to be considered. Again, FPGA-based prototyping works great for the software development side, but for hardware debug it is much more limited compared to simulation and emulation.

As part of the System Development Suite, the Palladium platform for emulation and Protium platform for FPGA-based prototyping offer a continuum of use models as indicated in Fig. 33.7. These use models range from hardware-centric development with simulation acceleration through detailed hardware/software debug with the Palladium emulation series and faster throughput regressions as well as software-centric development with the Protium platform.

33.5 High-Level Synthesis

The history of HLS is long [18]. It was already an active research topic in the EDA community in the 1970s, and by the early 1990s it was often introduced as the "next big thing", following the significant and very successful adoption of logic synthesis. However, only recently have commercial design projects started using this technology as the primary vehicle in the hardware design flow. Even then, its commercial use was limited to design applications that were historically considered as its sweet spot, dominated by data-processing or datapath functions with little

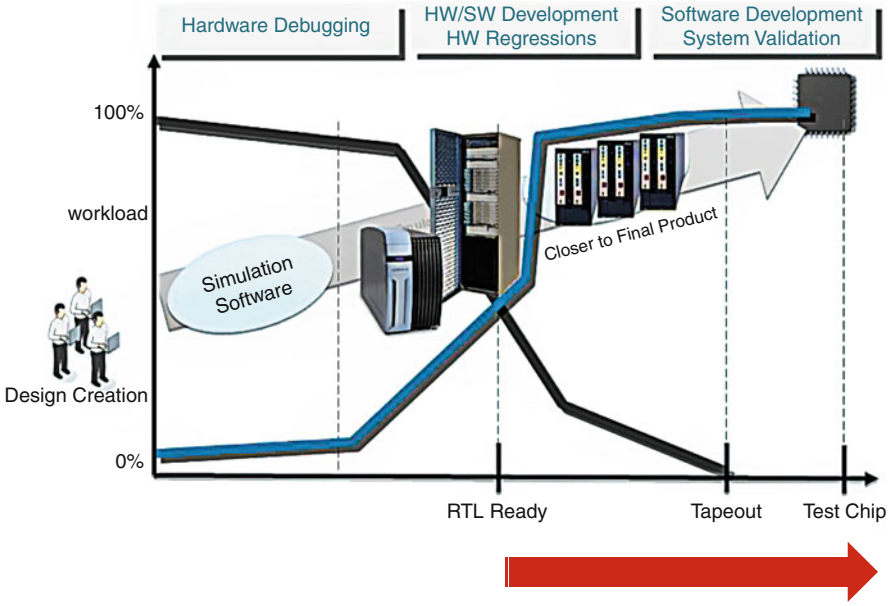

Fig. 33.7 Continuum of hardware-assisted development engines

control logic. This might suggest that HLS has had limited commercial success. On the other hand, industry users who have adopted this technology in their commercial design projects unanimously state that they would never go back to the RTL-based design flow. For them, HLS is an indispensable technology that enables them to achieve a quality of designs in tight project schedules that are not possible with RTL.

The IP blocks in today's complex designs are no longer just single datapath components, but are subsystems that include local memories for efficient data access, components that manage data transfers, and controllers for managing operations in the IPs with the rest of the system, in addition to core engines that implement algorithms to provide services defined by the IPs. These subsystems are integrated into a broad range of SoCs, which impose very different requirements in terms of implementation such as clock frequencies or performance constraints, as well as functionality on specific features or I/O interface configurations. Further, these requirements often change during the design projects. This is inevitable because nobody can foresee precisely what would be required in such complex systems before starting the projects, and details are often found when the designs are implemented or integrated into larger systems. It is therefore necessary that the design teams for those IP subsystems be able to support a broad range of design requirements imposed by different SoCs that integrate their designs, while at the same time responding to changes in requirements that arise throughout the design phases for each of them.

Stratus™ HLS, as illustrated in Fig. 33.8, addresses this need by providing three relevant characteristics that are essential for using HLS as the primary design

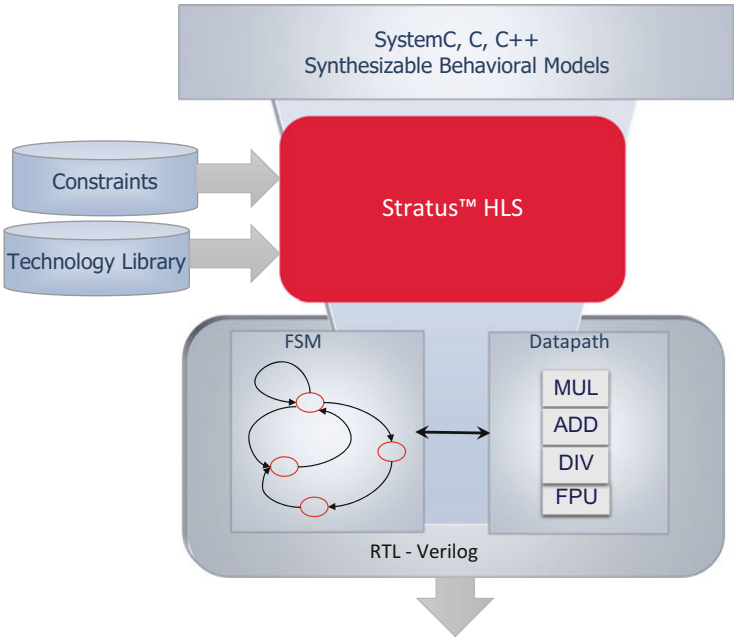

Fig. 33.8 Stratus™ HLS

technology in practice. First, it produces high-quality implementations for all components of IP subsystems that design teams need to deliver. It is no longer a tool for just datapath components. It takes as input behavioral descriptions of the target functionality in a highly configurable manner. The descriptions are specified using the SystemC language, where standard C++ techniques are used to define features and microarchitectures that can be included in the design through simple reconfiguration of the same descriptions. It also takes design requirements of the target implementations and technology library and produces synthesizable RTL for downstream implementation processes.

The breadth of configurations one can achieve with these descriptions is far beyond what is possible with RTL or parameterized RTL models, because the behavioral descriptions for Stratus HLS can result in totally different RTL structures just by changing the design parameters. The level of abstraction of these behavioral descriptions allows the designers to specify their design intent by focusing only on a few key specifics of the architectures while leaving the tool to figure out all the other details automatically. With this, they can easily evaluate various architectural choices of not only individual components of the IP but the whole subsystem.

For example, in achieving high-performance hardware implementations of algorithms, it is often important to take into account not only the cost of implementing the arithmetic computation of the algorithms but also the impact of accessing the data required for the algorithms. To address this concern, designers evaluate

the architecture of the memory hierarchy. In RTL design, they typically consider allocation of data to the memory hierarchy in such a way that data required by the individual arithmetic operations can be located close to the resources for executing the operations. This kind of exploration is easy in HLS, where one can change the memory hierarchy using design parameter configurations and data allocation to specific type of memories can be decided automatically.

The second aspect with which Stratus HLS provides strong value to IP design teams is the integration of this technology with the rest of the design and verification flow. Since HLS produces implementation from abstracted behavioral descriptions, it inevitably lacks detailed information that becomes available only in subsequent phases of the implementation flow. This causes a risk in general that design decisions made by HLS could cause issues that are difficult to close later in the design process. To mitigate this risk, one could either incorporate downstream tools within HLS or establish a closed loop from those tools back to an HLS tool. Stratus HLS does both. It uses the logic synthesis engine during the optimization process, so that it makes design decisions by accurately taking into account the information of actual resources implemented by logic synthesis. To cope with the wire congestion issue, the tool provides a back annotation mechanism to correlate the resources that cause high congestion during the layout phase to objects in the input behavioral descriptions, so that the designer can evaluate the root causes of wire congestion quickly.

The HLS design flow is also required to work with existing RTL designs, so that if the components designed with HLS are adjacent to components already written in RTL, the connections between them must be done seamlessly, despite the fact that they are written in different languages and using different abstraction levels for the interfaces. Stratus HLS provides features that automatically produce interlanguage interface adapters between the behavioral and RTL components. The user can decide on simulation configurations of a design that have mixtures of HLS components and RTL components, and the tool automatically inserts the adapters to establish the necessary connections. Such a mixture of behavioral and RTL descriptions also arises within a component that is fully designed with HLS.

Typically, a behavioral description for the component is written in a hierarchical manner, so that the design can be implemented gradually. When designers analyze the quality of implementation, they often focus on a particular subcomponent, leaving the rest of the design either at the behavioral level or at RTL depending upon the progress of the design phase. Stratus HLS provides a capability where the user can define multiple architectural choices in the individual subcomponents and then specify for each of them whether they want to use the behavioral description in simulating the subcomponent or the RTL description made for a particular architectural choice defined for it. The tool then automatically synthesizes the subcomponents as specified and combines the resulting RTL with behavioral descriptions of the remaining subcomponents to produce a simulation image. With this, the user can seamlessly verify the functionality of the component while focusing on particular subcomponents to explore various architectural choices to produce a high-quality implementation.

The third aspect that is extremely important for the adoption of HLS in practice is the support for Engineering Change Orders (ECOs). In the context of HLS, the main concern is support for functional ECOs, at a late stage, when design components have already been implemented to the logic or layout level and verification has been done, and the need arises to introduce small changes in the design functionality. In the RTL-based design flow, the designers carefully examine the RTL code and find a way to introduce the changes with minimal and localized modification of the code. If the designer tries to do the same with HLS, by introducing small changes in the behavioral description, when HLS is applied to the new description, the generated RTL often becomes very different from the original one. The logic implemented in RTL may change very significantly even if the functionality is very similar to the original one.

Stratus HLS provides an incremental synthesis feature to address this issue. In this flow, the tool saves information about the synthesis of the original design, and when an ECO happens, it takes as input this information together with the newly revised behavioral description. It then uses design similarity as the main cost metric during synthesis and produces RTL code with minimal differences from the original RTL code while meeting the specified functionality change.

High-quality implementations obtained from highly configurable behavioral descriptions for the whole IP subsystem, the integration with the existing design and verification flow, and the support for ECOs are the primary concerns that one needs to address when adopting high-level synthesis technology for designing new components of IPs. The fact that major semiconductor companies have successfully adopted Stratus HLS as an indispensable technology in their critical design projects is attributed to its strong capabilities in these aspects.

More information on HLS capabilities can be found in [6].

33.6 Application-Specific Instruction-Set Processors

This section discusses the concept of an ASIP and relates them specifically to hardware/software codesign. This concept is used to develop a particular codesign methodology: "processor-centric design." For another view on ASIPs, see ▸ Chap. 12, "Application-Specific Processors".

33.6.1 ASIP Concept and Tensilica Xtensa Technology

The foundation for processor-centric subsystem design is configurable, extensible processor technology, which has been developed by a number of academic and commercial groups since the 1990s [14, 22]. Tensilica technology [15, 27] dates from the late 1990s and has been applied to a wide variety of ASIP designs.

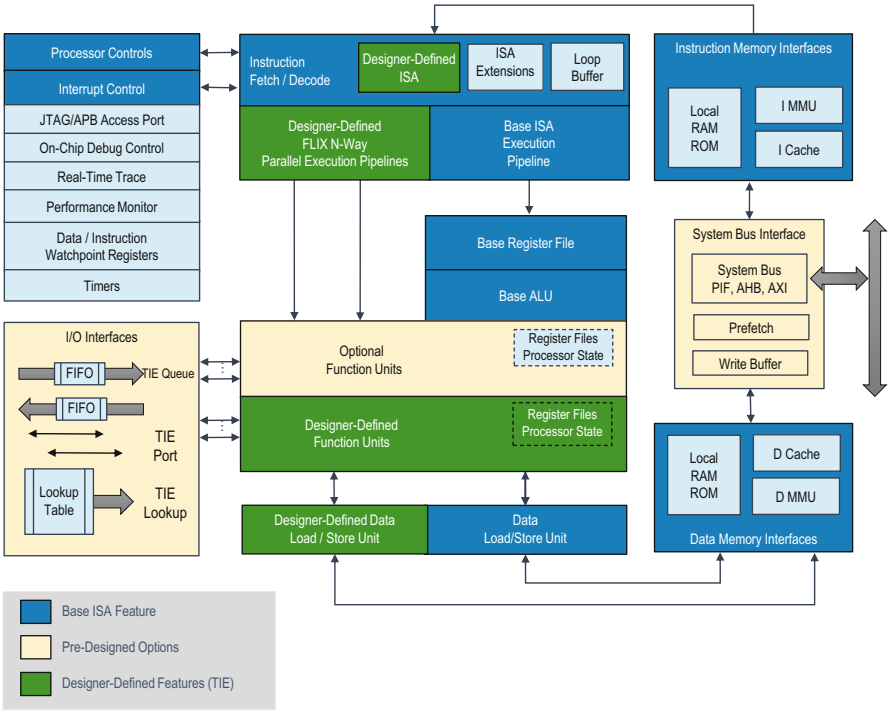

Fig. 33.9 Xtensa ASIP concept

Configurable, extensible processors allow designers to configure structural parameters and resources in a base Reduced Instruction-Set Processor (RISC) architecture as shown in Fig. 33.9. Extensibility allows design teams to add specialized instructions for applications. Automated tool flows create the hardware and software tools required, using specifications for structural configuration, and instruction extensions, defined by an architectural description language [21].

Configurable structural architecture parameters include:

- Size of register files
- Endianness
- Adding functional units, e.g., Multiply-Accumulators (MACs) and floating point
- Local data and instruction memory interfaces including configurable load-store units and Direct Memory Access (DMA) access and memory subsystem configuration
- Instruction and data cache attributes
- System memory and bus interfaces including standard buses such as Advanced eXtensible Interface (AXI)

- Debug, tracing, Joint Test Action Group (JTAG)
- Timers, interrupts and exceptions
- Multi-operation Very Long Instruction Word (VLIW) operation bundling
- Pipeline depth and microarchitecture choice
- Port, queue, and lookup interfaces into the processor's datapath

Instruction extensions, defined in Tensilica's TIE language [30], define specialized register bank width and depth, special processor state, operations of almost arbitrary complexity, their specification and optimized hardware implementation, SIMD-width, encoding, scheduling (single or multi-cycle), usage of operands and register ports, and bundling into multi-operation VLIW instructions. In addition, a number of documentation descriptions and software properties that influence operation scheduling in the compiler can be defined in TIE. Other aspects of the user programming model using instruction extensions, such as support for new C-types and operator overloading, and mapping of instruction sequences into a single atomic operation or group of operations can also be defined in TIE. Aggressive use of parallelism and other techniques in user-defined TIE extensions can often deliver 10X, 100X, or even greater performance increases compared to conventional fixed instruction-set processors or Digital Signal Processors (DSPs).

The automated tool flow generates the tooling for compilers, assemblers, instruction-set simulators, debuggers, profilers, and other software tools, along with scripts for optimized hardware implementation flows targeting current Application-Specific Integrated Circuit (ASIC) technologies [3].

Xtensa technology has been developed for over 17 years, from the founding of Tensilica as a separate company and its acquisition in 2013 [27]. This technology has been extensively verified [4, 24]. Designers perform their optimization and create their ideal Xtensa processor by using the Xtensa processor generator. In addition to producing the processor hardware RTL [11, 31], the Xtensa processor generator automatically generates a complete, optimized software-development environment. Two additional deliverables with Xtensa are:

1. Xtensa Xplorer Integrated Development Environment (IDE), based on Eclipse, which serves as a cockpit for single- and multiple-processor SoC hardware and software design. Xtensa Xplorer integrates software development, processor optimization, and multiple-processor SoC architecture tools into one common design environment. It also integrates SoC simulation and analysis tools.
2. A multiple processor (MP)-capable Instruction-Set Simulator (ISS) and C/C++ callable simulation libraries, along with a SystemC development environment XTSC.

ASIPs support Hardware/Software Codesign (HSCD) methodologies, albeit not quite in the classical sense of "all Hardware (HW)" vs. "all Software (SW)". ASIPs allow the computation and communications required by particular algorithms and applications to be mapped into flexible combinations of classical SW and application-oriented operations which are tuned to the application requirements.

Algorithms which are control-dominated can be mapped into an ASIP which is like a classical RISC machine, with configurability limited to aspects such as the memory subsystem and debug attributes. Algorithms heavy on computation with many application-specific operations can be mapped into ASIPs with extensive instruction extensions that greatly reduce the number of cycles required to execute and as a corollary, reduce the overall energy consumption of the algorithm by a large fraction. Algorithms heavy on communications methods or needing ancillary hardware execution units can utilize the port, queue and lookup interfaces to both simplify and improve the performance possible in passing data and control information from one core to another or to adjunct hardware blocks.

In this sense, ASIPs explode the design space exploration possibilities available to designers. They no longer need to live with just hardware or just selecting one from a list of predefined processor cores. They can tune one processor or a group of homogeneous or heterogeneous processors specifically to the particular application domain and algorithms their design is focused on. A good overview of design space exploration using Xtensa processors can be found in Chap. 6 of [4]. Design space exploration is discussed using this concept of processor-centric design.

33.6.2 DSP Design Using Xtensa

Xtensa ASIP technology has been applied by customers to create their own application-specific processors. It has also been applied internally within the research and development teams to create DSPs tuned to particular application domains. The key domains addressed through the years have been audio processing, communications, and video, imaging, and vision processing applications.

Audio [19] has been for many years a major focus of ASIP technology and audio DSPs. Several variations of audio DSPs exist, with distinct tradeoffs of power, speed performance, area, and cycle-time performance. As a result, the family of audio DSPs allow distinct hardware/software tradeoffs to be made by choosing the optimal audio DSP for a particular requirement. Software audio codecs and audio post-processing applications are also an important part of the offering.

A video codec subsystem called 388VDO [7,9] was developed several years ago. This consisted of two DSPs: a stream processor and a pixel processor, with adjunct DMA block, and an optional front-end Xtensa control processor. Several video encoders and decoders were offered as software IP with this subsystem, supporting major standards (such as MPEG2, MPEG4, JPEG, H264) and resolutions up to D2. The design of the Instruction-Set Architecture (ISA) for the two DSPs was done in close collaboration with the software team developing the video codecs and drew heavily on the concepts of hardware/software codesign, profiling, and performance analysis.

More recently, advanced vision and image processing processors [26, 29], are applicable to a wide variety of applications, have been developed. Computer vision is one of the fastest-growing application areas as of 2016, with particular attention being paid to Advanced Driver Assistance System (ADAS) in automotive

and security applications, gesture recognition, face detection, and many more. For another view on embedded computer vision and its relationship to ASIPs, see ▸ Chap. 40, "Embedded Computer Vision".

In the communications domain, a focus on wireless baseband processing was the impetus for development of specialized configurable DSPs [25]. In fact a family of DSPs was developed using a common ISA, variable Single Instruction, Multiple Data (SIMD) widths (16, 32, and 64 MACs) and a scalable programming model, based on evolving an earlier 16 MAC baseband DSP [28]. The basis is the combination of a real and complex vector processor, with specialized instructions for FFT for Orthogonal Frequency Dependent Multiplexing (OFDM).

33.6.3 Processor-Centric Design and Hardware/Software Design Space Exploration

This section describes the processor-centric design approach enabled by configurable, extensible ASIP methodologies, drawing on details to be found in Chap. 6 of [4].

Processor-centric design is a family of design approaches that includes several alternative methodologies. What is common to all of them is a bias toward implementing product functionality as software running on embedded processor(s), as opposed to dedicated hardware blocks. This does not mean that there are no dedicated hardware blocks in a processor-centric design; rather, these blocks are present as a matter of necessity rather than choice. In other words, dedicated hardware blocks will be present in the design where they *must* be, rather than where they *could* be. This could be to achieve the required level of performance, to achieve the desired product cost target, or to minimize energy consumption.

Traditional fixed ISA processors offer very stark tradeoffs for embedded product designers. They are generic for a class of processing and have few configurability options to allow them to be tailored more closely to the end application. The rise of ASIPs meant that designers could no longer consider the use of fixed embedded processors for an increasing number of the end-product application. ASIPs can now offer enough performance and sufficiently low energy consumption, at a reasonable cost, to take over much of the processing load that would have heretofore relied on dedicated hardware blocks. Thus ASIPs have been a key development enabling a much more processor-centric design style.

Traditional fixed ISA processors can be simply divided into control- and data-plane processors. Control processors, such as ARM and MIPS cores, are often used for non-data intensive applications or parts of an application, such as user interfaces, general task processing, high-level user applications, protocol stack processing, and the like. Data-plane processors are often fixed ISA DSPs that have special instructions and computational and communications resources that make them more suitable for data-intensive computation, especially for real-time signal and image processing.

As demonstrated earlier, ASIPs have grown in variety, number, and importance in recent years. Because an ASIP can be configured and extended to optimize its performance for a specific application, ASIPs offer much greater performance (say, 10–100X) and much lower energy consumption (perhaps half to one-quarter) than the same algorithm compiled for a fixed-ISA standard embedded processor – even a DSP. There are a few simple reasons to account for this advantage:

1. ASIPs allow coarse-grained configuration of their basic structure to better match the particular applications. If an application is mainly control processing, an ASIP may offer a fairly basic instruction set, but if an application is mainly intensive data processing (e.g., from the "data plane") – for example, audio, video, or other image processing – it may offer special additional instructions (zero-overhead loops, MACs) tuned to media or DSP kinds of applications.
2. The size and widths of registers can be tuned to be appropriate for the particular application domain.
3. Interfaces, such as memory interfaces, and caches can be configured or left out of the design dependent on data and instruction locality and the nature of the underlying algorithmic data access patterns. Sometimes caches may be more effective than local instruction and data (scratchpad) memories; sometimes the opposite may be the case.
4. Memory or bus interfaces may also be configured as to width and protocol – e.g., AMBA AHB or AXI.
5. Diagnosis and debug features such as trace ports, JTAG interfaces, and the like may be added or left out.
6. Interrupts and exception handling may be configured according to design need. Often the elaborate exception recovery mechanisms used in general purpose processors may be unnecessary in an ASIP tuned to run a very specific algorithm deeply embedded in a system.
7. VLIW style multi-operation instructions may be added to processors to support applications with a large amount of irregular instruction-level parallelism that can take advantage of such features.
8. SIMD type instructions – e.g., 2-, 4-, 8-, 16-way, or larger – may be added to processors to support vector-style simultaneous instructions acting on large chunks of data at a time.
9. Instructions may be tuned to specific algorithmic requirements. For example, if two 13-bit quantities need to be multiplied in an inner loop that dominates an algorithm, use of a 32-bit multiplier is both wasteful of area and energy and possibly performance.
10. Fine-grained instruction extensions including instruction fusions drawn from very specific algorithmic code can lead to significant increases in performance and savings in power. For example, a sequence of arithmetic operations in a tight loop nest that might account for 90% of the cycles in executing the algorithm on a data sample may be replaced with a single fused instruction that carries out the sequence in one or a few clock cycles.

Use of ASIPs instead of general purpose processors can lead, for known algorithms, to a radical improvement in performance and power consumption. This is true whether an ASIP is totally designed to support one very specific algorithm or if it is designed to support a class of applications drawn from a single domain. A specialized audio processing ASIP could be designed just to support MP3 decoding or could be slightly generalized so that it will support many different audio codecs – possibly optimizing one codec such as MP3 that is very widely used, but with general audio instructions added so that new codecs can still take advantage of the specific instructions and hardware incorporated in the ASIP.

Sometimes the complexity of a specific application domain may lead to a heterogeneous multi-processor, multi-ASIP design as being optimal for a certain target range of process technologies. Video codecs, baseband, vision, and imaging are examples.

A processor-centric design methodology needs to support design space exploration when deciding whether particular functional requirements for a design can be mapped to a single fixed ISA processor running at a suitable rate, a multi-processor implementation (such as a cache-coherent symmetric multi-processing "multi-core" cluster), a special fixed ISA processor such as a DSP, a single ASIP, a set of ASIPs configured to work together as a heterogeneous multi-processor subsystem, a combination of fixed ISA processor(s) and ASIP(s), and finally, mapping any part of the function into dedicated hardware blocks, almost certainly working in conjunction with the processors. A wide range of communications architectures, from shared memory accessed via buses through dedicated local memories, DMA blocks to permit concurrent data and instruction movement, direct communications such as First-In First-Out (FIFO) queues between processors and from processors to hardware, and NoCs may be used. In general, the processor-centric design flow has the following steps:

1. Start with an algorithm description. This is often reference C/C++ code obtained from a standards organization. Alternatively, it may be a reference code generated from an algorithmic description captured in a modeling notation such as the MathWorks' MatLab or Simulink, or in UML or one of its profiles, and using code generation to obtain executable C or C++.
2. Characterize the algorithm by running it on a very generic target processor. This will give designers some idea of the general computational and communications requirements of the algorithm (communications being defined as both data access and control access communicating into and out of the algorithm).
3. Identify "hot spots" in the target application. These will very often be loop nests in which multiple instructions are executed over large data samples. Techniques such as instruction fusion (combining multiple instructions into one); vectorization (SIMD) methods, where the same instruction is applied to many data items; and multi-operation instructions – where several operations without dependencies could be executed simultaneously on a VLIW-style architecture – are commonly identified.

4. Configure the processor and add instruction extensions to accelerate the execution of the algorithm. Re-characterize the code running on the new modified target. It may be necessary to restructure the code or insert pragmas into it in order that the compiler can take full advantage of vectorization (SIMD) or fused instructions.
5. If the performance targets for the algorithm are met and the estimates of power consumption and cost (area in terms of gates) are satisfactory, stop: this processor is now a reasonable choice for the function. Otherwise, further code restructuring and further configuration exploration and additional instruction extensions may be important. In this case, repeat the last few steps until either a satisfactory result is achieved, or it is necessary to add specialized hardware blocks as coprocessors in order to achieve the desired results.
6. If hardware blocks are necessary, they may be created using high-level synthesis tools, based on the algorithmic description for that part of the algorithm which must migrate to hardware. The design team may explore a variety of mechanisms for tying such accelerating blocks to the main processor – hardware FIFOs, coprocessor interfaces, or loosely coupled with systems buses, or DMA.

33.7 Software-Driven Verification and Portable Stimulus

The industry is rapidly approaching a new era in dynamic verification as indicated in Fig. 33.10.

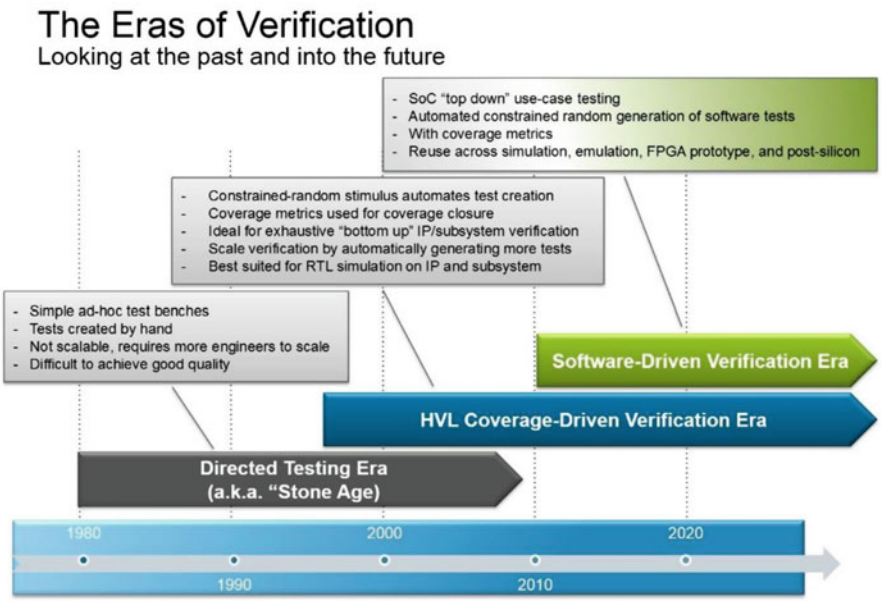

Fig. 33.10 The eras of verification

In the early days of verification, the "Stone Age" directed testing dominated verification. Design and verification engineers, at the time still emerging, were developing simple ad hoc test benches and creating tests by hand. This approach was not very scalable, as it required more engineers when more verification was required. As a result, it was very difficult to achieve good quality, and the confidence in how to get there and whether everything was verified was very hard to achieve.

In synchronization with the era of heavy IP reuse – sometime in the late 1990s to the early 2000s – the era of Hardware Verification Languages (HVLs) began. This is where specific verification languages such as VERA, ***e***, Superlog, and eventually SystemVerilog fundamentally changed the verification landscape. Methodologies were developed, including the Verification Methodology Manual (VMM), Open Verification Methodology (OVM), and later Universal Verification Methodology (UVM). In this era of verification, constrained-random stimulus automated test creation and coverage metrics were introduced to measure coverage closure. The level of automation involved in this era allowed users to scale verification by automatically generating more tests and made the HVL-based approaches ideal for exhaustive "bottom-up" IP and subsystem verification.

By 2016 the objects to be verified – modern SoCs – have evolved. They now contain many IP functions, from standard I/Os to system infrastructure and differentiating IP. They include many processor cores, both symmetric and asymmetric, both homogeneous and heterogeneous. Software executes on these processors, from core functionality such as communication stacks and infrastructure components such as Linux and Android operating systems all the way to user applications. Experts seem to agree that the UVM, while great for verification of IP blocks, falls short for SoC verification. The two main reasons are software and verification reuse between execution engines. It is important to note that UVM will not likely go away – it is fine for the "bottom-up" IP and some subsystem verification – and will continue to be used for these applications. However, UVM does not extend to new approaches for "top-down" SoC-level verification.

When switching from bottom-up verification to top-down verification, the context changes. In bottom-up verification, the question to verify is how the block or subsystem behaves in its SoC environment. In top-down verification, the correctness of the integrated IP blocks itself is assumed, and verification changes to scenarios describing how the SoC behaves in its system environment. An example scenario may be "view a video while uploading it." On top of the sequence of how the hardware blocks in the system interact, this scenario clearly involves a lot of software.

This is where traditional HVL-based techniques run up against their limits. They do not extend well to the software that is key to defining scenarios. Scenarios need to be represented in a way that they can be understood by a variety of users, from SoC architects, hardware developers, and software developers to verification engineers, software test engineers, and post-silicon validation engineers. They need to be comprehended by a variety of different users to allow efficient sharing. Also, the resulting test/verification stimulus needs to be portable across different

verification engines and even the actual silicon once available, enabling horizontal reuse. Software executing on the processors in the system – called software-driven verification – is the most likely candidate. Third and finally, the next wave of verification needs to allow both IP integration as well as IP operation within its system context to be tested, i.e., vertical reuse.

The Perspec System Verifier platform is one means to achieve portable stimulus by means of software-driven verification. Consider a use case from above: "view a video while uploading it." This six-word statement translates into bare-metal actions at the SoC level that need to be executed in a form such as "take a video buffer and convert it to MPEG4 format with medium resolution using any available graphics processor. Then transmit the result through the modem via any available communications processor and, in parallel, decode it using any available graphics processor and display the video stream on any of the SoC displays supporting the resulting resolution."

The state space this scenario creates is vast. Various resolutions, different video algorithms, different resources, different types of memory buffers, etc. need to be considered. Writing such a test manually, if even feasible, is hard to do and requires valuable system knowledge. And then the resolution, memory, or resources change – which makes it harder. This is where UML-like use-case definitions and constrained-random solving techniques to instantiate data and control flow with valid combinations of parameters come in as shown in Fig. 33.11.

The abstract use case reads from a memory buffer, converts data into a second memory buffer, and then in parallel transmits and decodes for display. The UML-based description is intuitive and can be understood by the various stakeholders. The automation involved transforms this description into an actual UML activity diagram with randomized video buffers, specific choices of video conversion

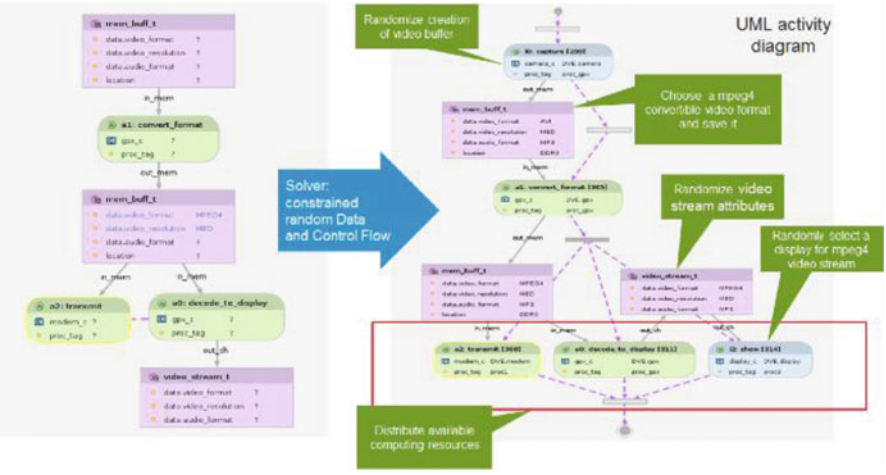

Fig. 33.11 UML use-case definition in Perspec System Verifier

formats such as MPEG4 to save into specific buffers, randomized video stream attributes, random selection of a display for the stream playback, and distribution across available compute resources.

Perspec System Verifier automatically generates the associated tests that execute on the processors in the design and run them on the various validation engines of the System Development Suite – from virtual platforms through RTL simulation, emulation, FPGA-based prototyping, and the actual silicon.

33.8 Conclusion

This chapter has surveyed a number of technologies for hardware/software codesign and coverification. They are undergoing constant evolution, and new applications are being found for these various technologies as technology and design practices evolve and change. This snapshot from 2016 represents state of the art in these areas as of that time. There are several new technology directions being explored.

Xtensa technology will evolve in two ways. First, it will support a wider range of microarchitectural choices and features, giving users even more options for creating ASIPs that meet their application needs. Secondly, it will be used in new and emerging application domains to offer new types of DSPs to users. Vision processing is a hot area in 2016 and likely to remain so, especially for emerging automotive applications. An even hotter subset of vision processing is the use of "AI" or deep-learning techniques such as Combinational Neural Networks and variations to support automotive ADAS applications. Both general vision DSPs with special Convolutional Neural Network (CNN) capabilities and highly application-specific CNN or other neural network ASIPs are possibilities.

System Development Suite (SDS) continues to move toward closer integration of different verification engines. The concept of "Continuum of Verification Engines" (COVE) [1] has been publicly discussed. Further connection of virtual prototyping, high-level synthesis, formal verification, RTL simulation, emulation, and FPGA-based prototyping have been recent and ongoing trends:

- **Verification acceleration**: Connection of the verification computing platform and RTL simulation to achieve accelerated execution is second only to *in-circuit emulation* applications. The Design Under Test (DUT) resides on the emulator and the test bench on the host; the host execution of the test bench controls the overall speed and users report 200–300X speedup over pure simulation.
- **Simulation/emulation hot swap**: This is a unique capability with SDS; users can run in one environment for a certain time, stop, and switch to the other.
- **Virtual platform/emulation hybrid**: This allows teams to reduce the time to the point of interest using, for example, fast models from ARM in virtual platforms connected to emulation.
- **Multi-fabric compilation for hardware engines**: In SDS, users have a multi-fabric compiler that can target both emulation and FPGA-based prototyping for in-circuit emulation, avoiding lengthy reengineering.

- **Unified Power Format (UPF)/Common Power Format (CPF) Low-Power Verification**: Power verification using either standard can be run in emulation and RTL simulation, for example, to verify the switching on and off of various power domains.
- **Portable stimulus**: Enables reuse of verification across various engines, including the chip itself. Vertical reuse from IP to subsystems to full SoCs is possible. Finally reuse is possible across various engineering disciplines.
- **Interconnect performance analysis**: Integrates verification IP (VIP) and RTL simulation to enable performance optimization and verification for interconnect.

A key capability going forward will be the automation of integration in a general way – for example, Interconnect Workbench (IWB) automatically generates test benches targeting different platforms.

References

1. http://www.deepchip.com/items/0549-04.html
2. Andrews M, Hristov B (2015) Portable stimulus models for c/SystemC, UVM and emulation. In: Design and verification conference and exhibition (DVCON). Accellera Systems Initiative
3. Augustine S, Gauthier M, Leibson S, Macliesh P, Martin G, Maydan D, Nedeljkovic N, Wilson B (2009) Generation and use of an ASIP software tool chain. In: Ecker W, Müller W, Dömer R (eds) Hardware-dependent software: principles and practice. Springer, Berlin, pp 173–202
4. Bailey B, Martin G (2010) ESL models and their application: electronic system level design and verification in practice. Springer, Boston
5. Bailey B, Martin G, Anderson T (eds) (2005) Taxonomies for the development and verification of digital systems. Springer, New York. The Virtual Socket Interface Alliance lasted from 1996 to 2008 but its archival web site is no longer functional as of 2015. This book may be all that is left of its work
6. Balarin F, Kondratyev A, Watanabe Y (2016) High level synthesis. In: Scheffer L, Lavagno L, Markov I, Martin G (eds) Electronic design automation for integrated circuits handbook, vol 1, 2nd edn. CRC Press/Taylor and Francis, Boca Raton
7. Bellas N, Katsavounidis I, Koziri M, Zacharis D (2009) Mapping the AVS video decoder on a heterogeneous dual-core SIMD processor. In: Design automation conference user track. IEEE/ACM. http://www.dac.com/46th/proceedings/slides/07U_2.pdf
8. Bianchi M, Snyder T, Grabowski D (2015) Denver IP acceleration leveraging enhanced debug. In: CDNLive silicon valley. Cadence Design Systems
9. Ezer G, Moolenaar D (2006) Mpsoc flow for multiformat video decoder based on configurable and extensible processors. In: GSPx Conference
10. Friedenthal S, Moore A, Steiner R (2014) A practical guide to SysML, 3rd edn. Morgan-Kaufmann, Boston
11. Gonzales R (2000) Xtensa: a configurable and extensible processor. IEEE Micro 20(2):60–70
12. Grötker T, Liao S, Martin G, Swan S (2002) System design with SystemC. Kluwer Academic Publishers, Dordrecht
13. Heaton N, Behar A (2014) Functional and performance verification of SoC interconnects. Embed Comput Des. http://embedded-computing.com/articles/functional-performance-verification-soc-interconnects/
14. Ienne P, Leupers R (2006) Customizable embedded processors: design technologies and applications. Morgan Kaufmann/Elsevier, San Francisco
15. Leibson S (2006) Designing SOCs with configured cores: unleashing the Tensilica Xtensa and diamond cores. Morgan Kaufmann/Elsevier, San Francisco

16. Martin G, Müller W (eds) (2005) UML for SoC design. Springer, Heidelberg
17. Martin G, Salefski B (2001) System level design for SoC's: a progress report – two years on. In: Ashenden P, Mermet J, Seepold R (eds) System-on-chip methodologies and design languages. Springer, Heidelberg, pp 297–306
18. Martin G, Smith G (2009) High-level synthesis: past, present, and future. IEEE Des Test 26(4):18–25
19. Maydan D (2011) Evolving voice and audio requirements for smartphones. In: Linley mobile conference. The Linley Group
20. Melling L, Kaye R (2015) Reducing time to point of interest with accelerated os boot. In: CDNLive silicon valley. Cadence Design Systems
21. Mishra P, Dutt N (2006) Processor modeling and design tools. In: Scheffer L, Lavagno L, Martin G (eds) Electronic design automation for integrated circuits handbook, vol 1, 1st edn. CRC Press/Taylor and Francis, Boca Raton
22. Mishra P, Dutt N (eds) (2008) Processor description languages. Elsevier-Morgan Kaufmann, Amsterdam/Boston
23. Murray D, Boylan S (2013) Lessons from the field: IP/SoC integration techniques that work. In: Design and verification conference and exhibition (DVCON). Accellera Systems Initiative
24. Puig-Medina M, Ezer G, Konas P (2000) Verification of configurable processor cores. In: Proceedings of design automation conference (DAC). IEEE/ACM, pp 184–188
25. Rowen C (2012) Power/performance breakthrough for LTE advanced handsets. In: Linley mobile conference. The Linley Group
26. Rowen C (2015) Instruction set innovation in fourth generation vision DSPs. In: Linley processor conference. The Linley Group
27. Rowen C, Leibson S (2004) Engineering the complex SoC: fast, flexible design with configurable processors. Prentice-Hall PTR, Upper Saddle River
28. Rowen C, Nuth P, Fiske S, Binning M, Khouri S (2012) A DSP architecture optimised for wireless baseband. In: International symposium on system-on-chip (ISSOC)
29. Sanghavi H (2015) Baby you can drive my car: vision-based SoC architectures. In: Linley processor conference. The Linley Group
30. Sanghavi H, Andrews N (2008) TIE: an ADL for designing application-specific instruction set extensions. In: Mishra P, Dutt N (eds) Processor description languages. Elsevier-Morgan Kaufmann, San Francisco
31. Wang A, Killian E, Maydan D, Rowen C (2001) Hardware/software instruction set configurability for system-on-chip processors. In: Proceedings of design automation conference (DAC). IEEE/ACM, pp 184–188