Platform-based Design

Alberto Sangiovanni Vincentelli
The Edgar L. and Harold H. Buttner Chair of EECS
University of California at Berkeley

Abstract. Platform-based design is a powerful concept for coping with the increased pressure on time-to-market, design and manufacturing costs. The idea has been exploited for years in the design of Personal Computers. We present a generalization and formalization of this approach for the design of electronic systems, composed by software and hardware components, and integrated circuits. This approach has been vigorously pursued by the MARCO Giga-scale Silicon Research Center as a foundation of its efforts. In addition to a formal definition of platforms and platform stacks, we present examples of this concept at the most important articulation points of the design process: the hand-off between system level applications and implementation and the one between circuit design and manufacturing. We also introduce the concept of network platform to show how to extend this concept to higher levels of abstractions, demonstrating its general applicability.

1. Introduction

The complexity of electronic designs and the number of technologies that must be mastered to bring to market winning products have forced electronic companies to focus on their core competence. Product specification, IP creation, design assembly and manufacturing are, for the most part, no longer taking place in the same organization. Indeed, the electronic industry has been disaggregating from a vertically oriented model into a horizontally oriented one for a few years. Integration of the supply chain is today a serious problem. Time-to-market pressure, design complexity and cost of ownership for masks are driving towards more disciplined design styles that favor design re-use and correct-the-first-time implementations. The quest for flexibility in embedded system design coupled with the previous considerations is pushing the electronic industry towards programmable solutions for a larger class of designs than ever before. Design methodology has become THE focus: design infrastructure and tools must be developed in synchrony with design methodology.

Creation of an economically feasible electronic design flow requires a structured methodology that theoretically limits the space of exploration, yet in doing so achieves superior results in the fixed time constraints of the design. This approach has been very powerful in design for both integrated circuits and computer programs. For computer programs, the use of high-level programming languages has replaced for the most part assembly languages, for integrated circuits, regular structures such as gate arrays and standard cells have replaced transistors as a basic building block. The methodology promoted in this paper can be seen as the result of a natural progression in the quest for higher level of abstractions (see Figure 1).

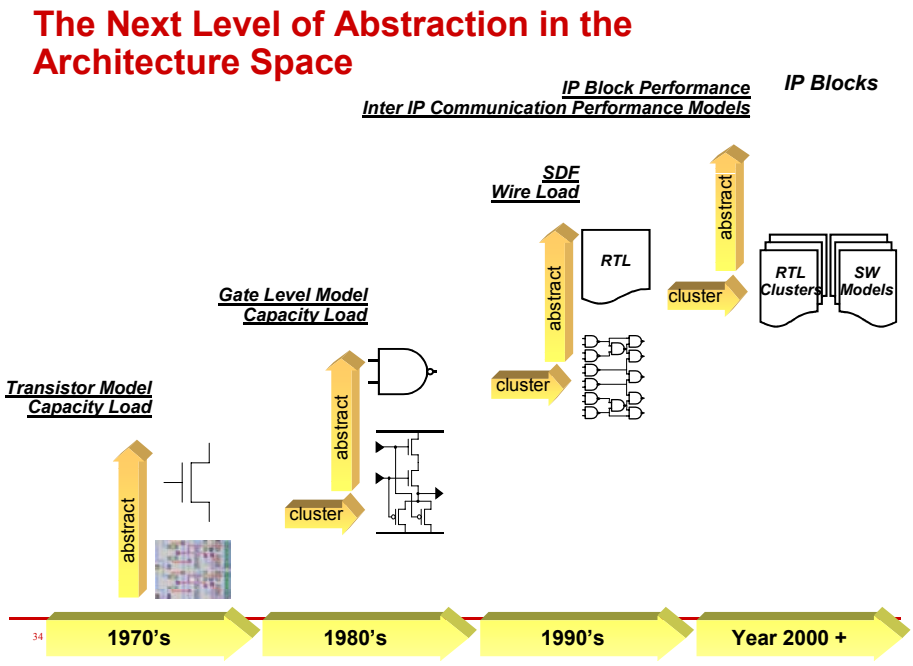

Figure 1 A brief history of abstraction in design (Source: F. Schirrmeister**).**

The methodology is based on defining ***Platforms*** at all of the key articulation points in the design flow. Each platform represents a layer in the design flow for which the underlying, subsequent design-flow steps are abstracted. By carefully defining the platform layers and developing new representations and associated transitions from one platform to the next, we believe that an economically feasible electronic system design flow can be realized.

The goal of this paper is to distill what has been done over the years so that a unified view of design could be based on platforms, i.e., abstraction layers that hide the unnecessary details of lower level of abstractions. We have been promoting platform-based design for a number of years and we have seen several instances of this principle come to market in the form of chips, chip-sets, architectures and middle-ware including Operating Systems, Compilers, Device Drivers and Network Communication Protocols.

The platform-based design methodology we have focused on is the outgrowth of the SoC debate where the economics of chip manufacturing and design has been carefully studied.

The overall goal of electronic system design is to balance *production costs with development time and cost in view of performance, functionality and product-volume constraints*.

1. *Manufacturing cost depends mainly on the hardware components* of the product. Minimizing production cost is the result of a balance between competing criteria. If we think of an integrated circuit implementation, then the size of the chip is an important factor in determining production cost. Minimizing the size of the chip implies tailoring the hardware architecture to the functionality of the product. However, the cost of a state-of-the-art fabrication facility continues to rise: it is estimated that a new 0.18µm high-volume manufacturing plant costs approximately $2-3B today.

2. *NRE (Non-Recurrent Engineering) costs* associated with the design and tooling of complex chips are growing rapidly. The ITRS predicts that while manufacturing complex System-on-Chip designs will be feasible, at least down to 50nm minimum feature sizes, *the production of practical*

masks and exposure systems will likely be a major bottleneck for the development of such chips. That is, the *cost of masks will grow even more rapidly for these fine geometries, adding even more to the up-front NRE for a new design.* A single mask set and probe card cost for a next-generation chip is over $1M for a complex part, up from less than $100K a decade ago (*note: this does not include the design cost*). Furthermore, the cost of developing and implementing a comprehensive test for such complex designs will continue to represent an increasing fraction of a total design cost unless new approaches are developed.

Increasing Mask and manufacturing setup costs are presently biasing the manufacturers towards *parts that have guaranteed high-volume production from a single mask set*. This translates to better response time and higher priorities at times when global manufacturing resources are in short supply.

3. *Design costs* are exponentially rising due to the increased complexity of the products, the challenges posed by physical effects for deep sub-micron and the limited human resources. Design productivity according to Sematech is falling behind exponentially with respect to the technology advances. Time-to-market constraints are also growing at such a fast pace that even if costs were not an issue, it is becoming plainly impossible to develop complex parts within the constraints. An additional problem is the lack of skilled work force that could implement future IC's considering the system aspects of the design and all second order physical effects that will be of primary importance in deep sub micron.

The design problems are presently pushing IC and system companies towards *designs that can be assembled quickly from pre-designed and pre-characterized components versus full custom design methods*. This translates to high-priority on design re-use, correct assembly of components, and fast, efficient compilation from specifications to implementations, correct-by-construction methodologies and fast/accurate verification[1].

Our objective is a design methodology (i.e., **Platform-based Design**) that can trade-off various components of manufacturing, NRE and design costs sacrificing as little as possible "potential" design performance. We define the methodology so that it applies *to all levels of abstraction of the design thereby providing an all-encompassing intellectual framework in which research and design practices can be embedded and justified.*

In this paper, we define the motivations and principles for ***Platform-based Design***[2]. We believe that the popularity of this concept has led to confusion as to what this term means and implies. The goal here is to define what these characteristics are so that a common understanding can be built and a precise reference can be given to the electronic system and circuit design community.

Platform-based design has been generically associated to design styles that favor software solutions over application specific hardware. The chip-in-a-day approach proposed by Bob Brodersen has been pitted against platform-based design as a more effective way of designing high-performance electronic systems. However, if the "formal" definition of platform-based design, as given here, is used, then one can see that design style extremes such as full custom and fully programmable styles can be embedded in the unifying framework offered here. In any case, we believe that the trend towards more software-based solutions is strong and is going to permeate electronic system design for many years to come.

[1] The design time issue can be addressed not only by re-use but also by tools that provide automatic mapping. For example, if we had the ultimate silicon compiler, an architecture platform could be the entire set of micro-architectures the silicon compiler can map into.

[2] The design methodology exposed here has been one of the major trusts of the MARCO Giga-scale Silicon Research Center, whose overarching goal is:
"Empowering designers to move from ad-hoc system-on-a-chip design to disciplined, platform-based design by enabling scalable, heterogeneous, component-based design with a single-pass route to efficient silicon implementation from a micro-architecture" (*Richard Newton and the GSRC team*)

2. Platforms

The concept of platform has been around for years. However, many are the definitions of platform that have been used and that depend on the domain of application.

In the IC domain, a platform is considered a "flexible" integrated circuit where customization for a particular application is achieved by "programming" one or more of the components of the chip. Programming may imply "metal customization" (Gate arrays), electrical modification (FPGA personalization) or software to run on a microprocessor or a DSP. For example, a platform has been considered a fixed micro-architecture to minimize mask-making costs, but flexible enough to warrant its use for a set of applications so that production volume will be high over an extended chip lifetime. Micro-controllers designed for automotive applications such as the Motorola Black Oak PowerPC are examples of this approach. The problem with this approach is the potential lack of optimization that may make performance too low and size too large. An extension of this concept is "a family" of similar chips that differ for one or more components but that are based on the same microprocessor. For the case of the Black Oak, Motorola indeed developed a family of micro-controllers, e.g., Silver Oak and Green Oak, that differ for flash memory size and peripherals. The TI OMAP platform for wireless communication, the Phillips Nexperia Platform, and the Xilinx Vertex Platform are a few examples of this approach.

"We define platform-based design as the creation of a stable microprocessor-based architecture that can be rapidly extended, customized for a range of applications, and delivered to customers for quick deployment." *Source: Jean-Marc Chateau (ST Micro)*

In the PC domain, PC makers have been able to develop their products quickly and efficiently around a standard "platform" that emerged over the years. The architecture standards can be summarized in the following list:

a. The x86 instruction set architecture (ISA) that makes it possible to re-use the operating system and the software application at the binary level[3];
b. A fully specified set of busses (ISA, USB, PCI) that make it possible to use the same expansion boards or IC's for different products[4];
c. Legacy support for the ISA interrupt controller that handles the basic interaction between software and hardware.
d. A full specification of a set of I/O devices, such as keyboard, mouse, audio and video devices.

All PCs should satisfy this set of *constraints*. If we examine carefully the structure of a PC platform, we note that *it is not the detailed hardware micro-architecture that is standardized, but rather an abstraction characterized by a set of constraints on the architecture (**a** through **d** above). The platform is an abstraction of a "family" of (micro)-architectures*. In this case, design time is certainly minimized since the essential components of the architecture are fixed and the degrees of freedom allow some optimization for performance and cost[5].

For system companies, the definition of platform is very loose. This quote from an Ericsson press release is a good example: "Ericsson's Internet Services Platform is a new tool for helping CDMA operators and service providers deploy Mobile Internet applications rapidly, efficiently and cost-effectively."

[3] In fact, the MS-DOS operating system can be run on any compatible x86 microprocessor.
[4] Note that expansion board re-usability is limited by the technology used.
[5] The concept of PC platform actually can be linked to the old Burroughs 'E-mode' mainframe architectural concept, where the ISA was common across many different implementations over many years. Burroughs E-mode was claimed to be so effective that programs compiled on the first machines in the early 60's would work (as object code) on the latest machines in the 80's and 90's.

3. Platform-Based Design

3.1. The Overarching Conceptual View

As we have seen, various forms of platform-based design have been used for many years. Our intention is to formally define the key principles of platform-based design that will serve as a framework for design technology research and practices.

The basic tenets of the Platform-based Design Methodology we propose are:

- Regarding design as a "meeting-in-the-middle process" where successive refinements of specifications meet with abstractions of potential implementations;
- The identification of precisely defined layers where the refinement and abstraction process take place. The layers then support designs built upon them isolating from lower-level details but letting enough information transpire about lower levels of abstraction to allow design space exploration with a fairly accurate prediction of the properties of the final implementation. The information should be incorporated in appropriate parameters that annotate design choices at the present layer of abstraction. These layers of abstraction are called ***Platforms*** to stress their role in the design process and their solidity.

From a historical perspective and the newly formed concepts stated above, we can state that the general definition of a platform is an *abstraction layer in the design flow that facilitates a number of possible refinements into a subsequent abstraction layer (platform) in the design flow.*

The "mille feuilles"[6] of Figure 2 is a rendition of the design process as a succession of abstraction layers. The analogy covers also the filling between consecutive layers. This filling corresponds to the set of methods and tools that allow to map the design from one abstraction layer to the next.

Often the combination of two consecutive layers and their "filling" can be interpreted as a unique abstraction layer with an "upper" view, the top abstraction layer and a "lower" view, the bottom layer. We will see some examples of how to use this concept.

Every pair of platforms, the tools and methods that are used to map the upper layer of abstraction into the lower level one is *a platform stack*.

[6] A "mille feuilles" is a French origin dessert. The version shown in Figure 2 is one of its Italian renditions where the custard layers are replaced by Chantilly cream layers enriched by wild berries and the dough layers are lighter and crispier.

Figure 2. Platforms, Mapping Tools and Platform Stacks

Note that we can allow a platform-stack to include several sub-stacks if we wish to span a large number of abstractions. This will largely depend on the capabilities of the tools and the outcome of research programs such as the GSRC's. We emphasize again that the upper view, the lower view and the tools that map the two platforms are the important constituents of the platform-stack. The larger the span, the more difficult it will be to map effectively the two, but the greater the *potential*[7] for design optimization and exploration.

Key to the application of the design principle is the careful definition of the platform layers. Platforms can be defined at several point of the design process. Some levels of abstractions are more important than others in the overall design trade-off space. In particular, the articulation point between system definition and implementation is a critical one for design quality and time. Indeed, the very notion of platform-based design originated at this point (see Ferrari and Sangiovanni-Vincentelli, 1999, Martin, Chang et al, 1999, Balarin et al., 1997, Keutzer et al, 2000). In studying this articulation point, we have discovered that at this level there are indeed two distinct platforms that need to be defined together with the methods and tools necessary to link the two. The one that has been studied the most is what we call the Architecture Platform.

3.2. (Micro-) Architecture Platforms

Integrated circuits used for embedded systems will most likely be developed as an instance of a particular (micro-) **architecture platform**. That is, rather than being assembled from a collection of independently developed blocks of silicon functionality, they will be derived from *a specific "family" of micro-architectures*, possibly oriented toward a particular class of problems, that can be modified (extended or reduced) by the system developer.

[7] Importantly, most of this potential is unrealized due to the complex and sometimes unrealistic nature of the design and verification processes for a single-pass design flow. Our research is toward restricting the space of exploration as little as possible, but while enabling a reliable single-pass process.

The elements of this family are a sort of "hardware" denominator that could be shared across multiple applications. Hence, *the (micro-) architecture platform concept as a family of micro-architectures that are closely related is mainly geared towards optimizing design time*: every element of the family can be obtained quickly by personalizing an appropriate set of parameters that control the micro-architecture. *For example*, the family may be characterized by the same programmable processor and the same interconnection scheme, but the peripherals and the memories of a particular *implementation* may be selected *from a pre-designed library of components* depending on the particular application. Depending on the implementation platform that is chosen – more on this to follow -- each element of the family may still need to go through the standard manufacturing process including mask making. *This approach then conjugates the need of saving design time with the optimization of the element of the family for the application at hand. Although it does not solve the mask cost issue directly,* it should be noted that the mask cost problem is primarily due to generating multiple mask sets for multiple design spins, which is addressed by the Architecture Platform methodology. The less constrained the platform, the more freedom a designer has in selecting an instance and the more potential there is for optimization —if time permits. However, more constraints mean stronger standards and easier addition of components to the library that defines the architecture platform (see the PC case). Note that the basic concept is similar to the cell based design layout style where regularity and the re-use of library elements allows faster design time at the expense of some optimality. The trade-off between design time and design "quality" as always to be kept in mind. The economics of the design problem have to dictate the choice of design style. The higher the granularity of the library, the more leverage we have in shortening the design time. Given that the elements of the library are re-used, there is a strong incentive to optimize them. In fact, we argue that the "macro-cells" should be designed with great care and attention to area and performance. It makes also sense to offer a variation of cells with the same functionality but with implementations that differ in performance, area and power dissipation.

Architecture platforms are, in general, characterized by (but not limited to) the presence of programmable components so that each of the platform instances that can be derived from the architecture platform maintains enough flexibility to support an application space that guarantees the production volumes required for economically viable manufacturing. The library that defines the architecture platform may also contain re-configurable components. [8]

3.2.1 Architecture Platform Instance

An **architecture platform instance** is derived from an architecture platform by choosing a set of components from the architecture platform library and/or by setting parameters of re-configurable components of the library.

The *flexibility, i.e., the capability of supporting different applications,* of a platform instance is guaranteed by programmable components. Programmability will ultimately be of various forms, including the two that exist today: software programmability to indicate the presence of a micro-processor, DSP or any other software programmable component, or hardware programmability to indicate the presence of reconfigurable logic blocks such as FPGAs, whereby logic function can be changed by software tools without requiring a custom set of masks. Some of the new architecture and/or implementation platforms being offered on the market mix the two into a single chip (for example, Triscend, Altera and Xilinx are offering FPGA fabrics with embedded hard processors). Software programmability yields a more flexible solution since modifying software is in general faster and cheaper than modifying FPGA personalities. On the other hand, logic functions mapped on FPGAs execute orders of magnitude faster and with much less power than the corresponding implementation as a software program. Thus, the trade-off here is between flexibility and performance.

[8] Reconfigurability comes in two flavors:

- Run-time reconfigurability, a la Triscend/Xilinx/Altera, where FPGA blocks can be customized by the user without the need of changing mask set, thus saving both design cost and fabrication cost;
- Design-time reconfigurability, a la Tensilica Xtensa, where the silicon is still application-specific; in this case, only design time is reduced.

3.2.2 Architecture Platform Design Issues

Today the choice of an **architecture platform** is more an art than a science. Seen from the application domain, the constraints that determine the architecture platform are often given in terms of performance and "size". For a particular application, we require that, to sustain a set of functions, a CPU should be able to run at least at a given speed and the memory system should be of at least a given number of bytes. Since each product is characterized by a different set of functions, the constraints identify different architecture platforms where applications that are more complex yield stronger architectural constraints. Coming from the IC manufacturer space, production and design costs imply adding platform constraints and consequently reducing the number of choices. The intersection of the two sets of constraints defines the architecture platforms that can be used for the final product. Note that, because of this process, we may have a platform instance that is over-designed for a given product; that is, the potential of the architecture is not fully utilized to implement the functionality of that product. Over-design is very common for the PC platform. In several applications, the over-designed architecture has been a perfect vehicle to deliver new software products and extend the application space. We believe that some degree of over-design will be soon accepted in the embedded system community to improve design costs and time-to-market. Hence, the "design" of an architecture platform is the result of a trade-off in a complex space that includes:

- The size of the application space that can be supported by the architectures belonging to the architecture platform. This represents the flexibility of the platform;
- The size of the architecture space that satisfies the constraints embodied in the architecture platform definition. This represents the degrees of freedom that architecture providers have in designing their hardware instances.

Once an architecture platform has been selected, then the design process consists of exploring the remaining design space with the constraints set by the platform. *These constraints cannot only be on the components themselves but also on their communication mechanism*. In fact, particular busses may be a fixed choice for the communication mechanism (for example, the AMBA bus for the ARM micro-processor family).

When we march towards implementation by selecting components that satisfy the architectural constraints defining a platform, we perform a successive refinement process where details are added in a disciplined way to produce an architecture platform instance.

Architecture platform-based design is neither a top-down nor a bottom-up design methodology. Rather, it is a "meet-in-the-middle" approach. In a pure top-down design process, application specification is the starting point for the design process. The sequence of design decisions drives the designer toward a solution that minimizes the cost of the architecture. The design process selects the most attractive solution as defined by a cost function. In a bottom-up approach, a given architecture (instance of the architecture platform) is designed to support a set of different applications that are often vaguely defined and is, in general, much based on designer intuition and marketing inputs. In general, IC companies traditionally followed this approach trying to maximize the number of applications (hence, the production volume) of their platform instances. The trend is towards defining platforms and platform instances in close collaboration with system companies thus fully realizing the meet-in-the-middle approach.

Application developers work with an architecture platform by first choosing the architectural elements they believe are best for their purposes yielding a platform instance. Then, they must map the functionality of their application onto the platform instance. The mapping process includes hardware/software partitioning. While performing this step, the designers may decide to move a function from software implementation running on one of the programmable components to a hardware block. *This hardware could be implemented anywhere from programmable logic to full custom*, the representations of which would be abstracted from the implementation platform. Once the partitioning and the selection of the platform instance are finalized, the designer develops the final and optimized version of the application software.

Due to the market forces briefly outlined above, most of the implementation of a system functionality is done in software. There is in fact a very strong trend in system-level design towards software away from hardware implementations. Indeed, market data indicate that more than 80% of system development efforts are now in software versus hardware. This implies that an effective platform has to offer a powerful design environment for software to cope with development costs. In addition, one of the motivations toward standardization of the programmable components in platforms is software re-use. If the Instruction Set Architecture is kept constant, then software porting is much easier. However, this mechanism limits the degrees of freedom of system designers who may have to ignore very performing platforms in favor of older architectures to maintain software compatibility. Thus, there are two main concerns for an effective platform-based design:

1. Software development environment;
2. A set of tools that insulate the details of the architecture from application software.

This brings us to the definition of an **API platform**.

3.3. API Platform

The concept of architecture platform by itself is not enough to achieve the level of application software re-use we require. The architecture platform has to be abstracted at a level where the application software "sees" a high-level interface to the hardware that we call Application Program Interface (API) or Programmers Model. A software layer is used to perform this abstraction (see Figure 3). This layer wraps the essential parts of the architecture platform:

- The programmable cores and the memory subsystem via a Real Time Operating System (RTOS),
- The I/O subsystem via the Device Drivers, and
- The network connection via the network communication subsystem[9].

[9] In some cases, the entire software layer, including the Device Drivers and the network communication subsystem is called RTOS.

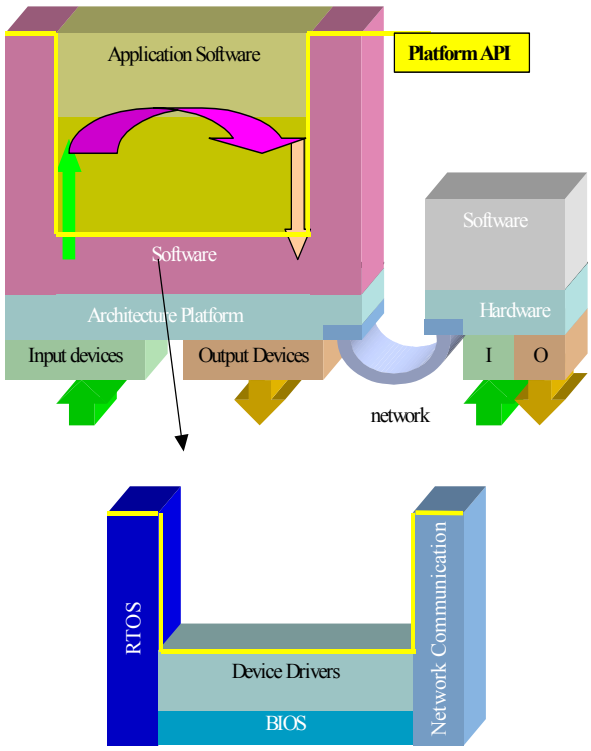

Figure 3. Layered software structure (Source: A. Ferrari)

In our conceptual framework, the programming language is the abstraction of the ISA, while the API is the abstraction of a multiplicity of computational resources (concurrency model provided by the RTOS) and available peripherals (Device Drivers)[10]. There are different efforts that try to standardize the API or Programmers Model.

In our framework, the API or Programmers Model is a unique abstract representation of the architecture platform via the software layer. With an API so defined, the application software can be re-used for every platform instance. Indeed the Programmers Model (API) is a platform itself that we can call the ***API platform****.*

Of course, the higher the abstraction level at which a platform is defined, the more instances it contains. For example, to share source code, we need to have the same operating system but not necessarily the same instruction set, while to share binary code, we need to add the architectural constraints that force to use the same ISA, thus greatly restricting the range of architectural choices.

In our framework, the RTOS is responsible for the scheduling of the available computing resources and of the communication between them and the memory subsystem. Note that in several embedded system applications, the available computing resources consist of a single microprocessor. In others, such as wireless handsets, the combination of a RISC microprocessor or controller and DSP has been used widely in 2G, now for 2.5G and 3G, and beyond. In set-top boxes, a RISC for control and a media processor have also been used. In general, we can imagine a multiple core architecture platform where the RTOS schedules software processes across different computing engines.

[10] Several languages abstract or embed directly the concurrency model avoiding the RTOS abstraction.

There is a battle that is taking place in this domain to establish a standard RTOS for embedded applications. For example, traditional embedded software vendors such as ISI and WindRiver are now competing with Microsoft that is trying to enter this domain by offering Windows CE, a stripped down version of the API of its Windows operating system. In our opinion, if the conceptual framework we offer here is accepted, the precise definition of the hardware platform and of the API should allow to synthesize automatically and in an optimal way most of the software layer, a radical departure from the standard models borrowed from the PC world.

3.4. System Platform-Stack

The basic idea of system platform-stack is captured in Figure 4. The vertex of the two cones represents the combination of the API or Programmers' Model and the architecture platform. A system designer maps its application into the abstract representation that "includes" a family of architectures that can be chosen to optimize cost, efficiency, energy consumption and flexibility. The mapping of the application into the actual architecture in the family specified by the Programmers' Model or API can be carried out, at least in part, automatically if a set of appropriate software tools (e.g., software synthesis, RTOS synthesis, device-driver synthesis) is available. It is clear that the synthesis tools have to be aware of the architecture features as well as of the API. This set of tools makes use of the software layer to go from the API platform to the architecture platform.

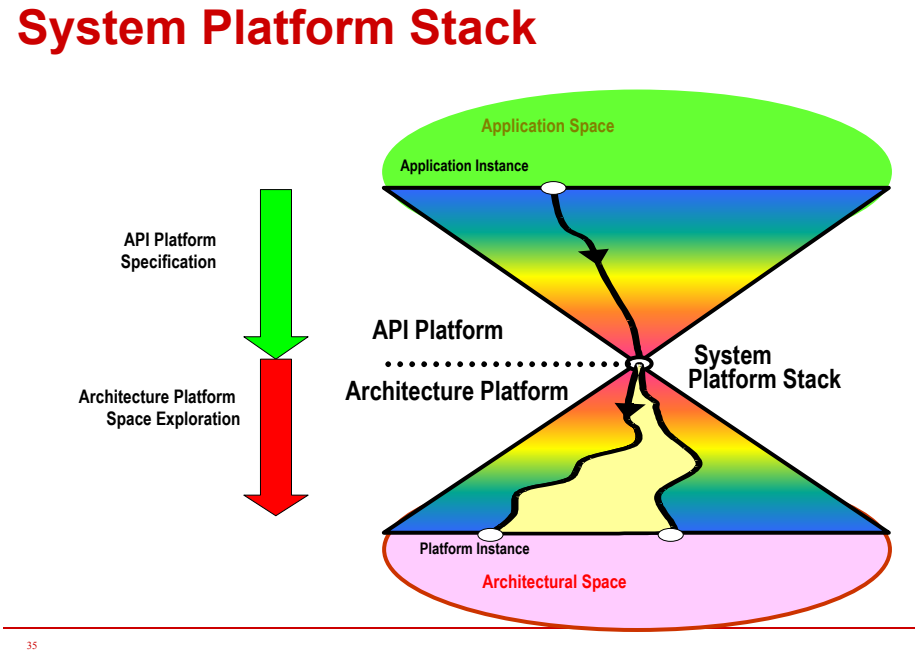

Figure 4 System platform stack

The System Platform-Stack is the combination of two platforms and the tools that map one abstraction into the other.

We recall that the **platform–stack** can be seen as a "single" layer obtained by gluing together the top platform and the bottom platform whereby the upper view is the API platform and the lower view is the collection of components that comprise the architecture platform.

In the design space, there is an obvious trade-off between the level of abstraction of the Programmers' Model and the number and diversity of the platform instances covered. The more abstract the Programmers' Model the richer is the set of platform instances, but the more difficult it is to choose the

"optimal" architecture platform instance and map automatically into it. Hence, we envision a number of system platform-stacks that will be handled with somewhat different abstractions and tools. For example, traditional platforms that include a small number of standard components such as microprocessors and DSPs have an API that is simpler to handle than that for reconfigurable architectures[11].

Generalizing our thinking process, we view *design as primarily a process of providing abstraction views*. That is, an "API" platform is a pre-defined layer of abstraction above some more complex device or system that can be used to design at a higher level. Suppose our low level machine is a micro-controller with some peripheral devices and some programmable logic. A high-level language compiler, a small RTOS and a logic synthesis tool for the programmable logic, along with models of the fixed peripherals, might provide the link between the API platform and this machine architecture.

Following this model, we see that a structural view of the design is abstracted into the "API" model that provides the basis for the design process that rests upon this layer of abstraction. To choose the right architecture platform we need to export at the API level an "execution" model of the architecture platform that estimates the performance of the lower level architecture platform. This model may include size, power consumption and timing; variables that are associated to the lower level abstraction (from the implementation platform) and that cannot be computed at the "API" level. On the other hand, we can pass constraints from higher levels of abstraction down to lower levels to continue the refinement process satisfying the original design constraints. Together with constraints and estimates, we may also use cost functions to select among feasible solutions.

In summary, the system platform–stack is a comprehensive model that includes the view of platforms from both the application and the implementation point of views. It is the vertex of the two cones in Figure 4. Note that the system platform effectively decouples the application development process (the upper triangle) from the architecture implementation process (the lower triangle). Note also that, once we use the abstract definition of "API" as described above, we may obtain extreme cases such as traditional PC platforms on one side and full hardware implementation on the other. Of course, the programmer model for a full custom hardware solution is trivial since there is a one-to-one map between functions to be implemented and physical blocks that implement them. In this latter case, platform-based design amount to adding to traditional design methodologies some higher level of abstractions. Re-usability is of course almost not existent. Figure 5 shows how different design styles can be framed in a disciplined platform-based design approach. In this figure, we also show the mapping of an application onto the appropriate platform and the export of performance parameters from the lower level platforms to the API.

[11] There are also mixed-level platforms (e.g., C and assembler, or RTOS and direct register read/write). They may be non-ideal from the viewpoint of abstraction, but they are still used and should be mentioned. In the absence of good mapping tools, they may be the only option. Another way of viewing them is the ability of mixing blocks designed using various platform layers in the same design.

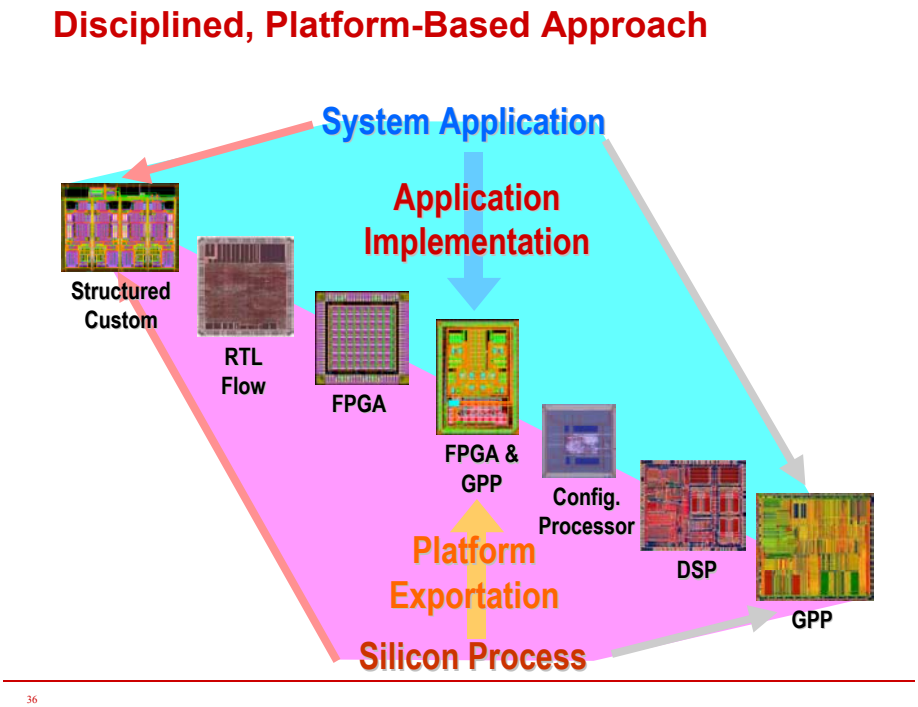

Figure 5 Design styles for architecture implementation in platform-based design (source: R. Newton)**.**

3.5. Silicon Implementation Platform Stack

The Silicon Implementation Platform (SIP) Stack of the design flow takes all of or components of the architecture platform and maps them to a physical implementation. In this case, we can identify a set of abstractions that are related by mapping methods in the same vein that we followed above. In addition, we can also export to the physical implementation problem the view of platform stack to include the combination of two platforms and of the intermediate levels of abstraction with the transformations and tools needed to go from one abstraction to the next.

The platform stack from architecture to actual physical implementation is called the **Silicon Implementation Platform (SIP) Stack**.

The top-level view of the SIP stack is the platform architecture view, as shown in

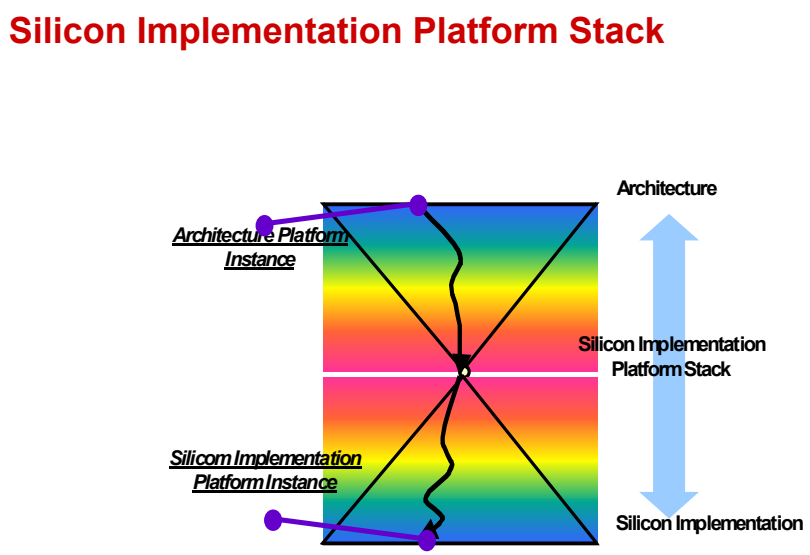

Figure 6.

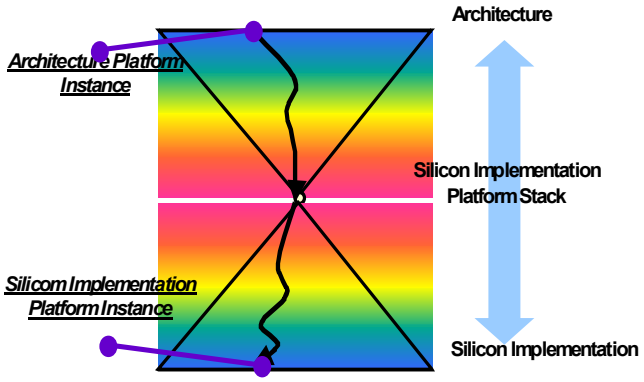

Figure 6 Silicon implementation stack

The view from beneath the architecture platform is a net-list of abstract views of computational blocks and interconnects structures. The next platform, or subsequent level of abstraction towards implementation, can vary somewhat based on the underlying technologies that are abstracted. For example, the layout of the blocks and of the interconnections among blocks affects all the performance parameters such as size, timing and power consumption. The constraints from the architecture platform, that were generated based on implementation layer abstractions, are used to determine whether a particular layout of the architecture platform instance can be used. In some cases a floor-plan view would be used to export upward the parameters related to various components, including interconnect. In this case, the floor plan is itself a platform (the **floor-plan platform**) in our conceptual framework. In other cases, including those which result from creation of new implementation platforms and silicon implementation methodologies as part of our GSRC research, the silicon implementation platform stack can be a single multi-faceted layer, as shown in Figure 4. Similarly, the floor-plan platform can be directly incorporated as part of the architecture platform. More research remains to be done on this topic.

The concept of implementation platform is strongly related to the concept of **regular design** and design re-use. Based on extensive reuse, floor-plan platforms represent blocks and associated interconnection patterns that are selected from a library of pre-designed components. These components may be described at the detailed physical level or at a higher level of abstraction where new abstractions are used to represent the details of the manufacturing process and isolation from the complex manufacturing interface. More generally, the architecture components can be mapped to regular blocks of logic that are synthesized into geometrically and physically regular, often programmable and/or reconfigurable, fabrics. Importantly, these fabrics that lie in the SIP stack incorporate accurate abstractions from the manufacturing interface. In addition, the myriad of components and instances which lie in the SIP stack are based on regular structures that facilitate: correct-by-assembly design, ease of verification, construction of reliable components from widely fluctuating parameters, and manufacture of high-yielding reliable silicon ICs.

It should be noted that this design process from system conception to implementation could also stop at other platform abstractions where the actual implementation is fully instantiated. Any design style (or at least the most meaningful design styles) can be *restructured and reorganized* to fit within a platform-based concept. It can span all the way from a customized solution to a fully programmable standard part. Of course, in the case we do not leverage re-use and flexibility via novel regular structures, there is no difference from traditional design flows. The important point here is that this approach does capture re-use and flexibility but it is itself flexible enough to mix and match design styles to accommodate a wide spectrum of applications and implementation platforms.

The final stack on the path to implementation and manufacture is the mask set and the recipes for manufacturing. The technology files are the characterization of the **manufacturing interface platform** used to generate the most accurate performance and cost parameters, as shown in Figure 7.

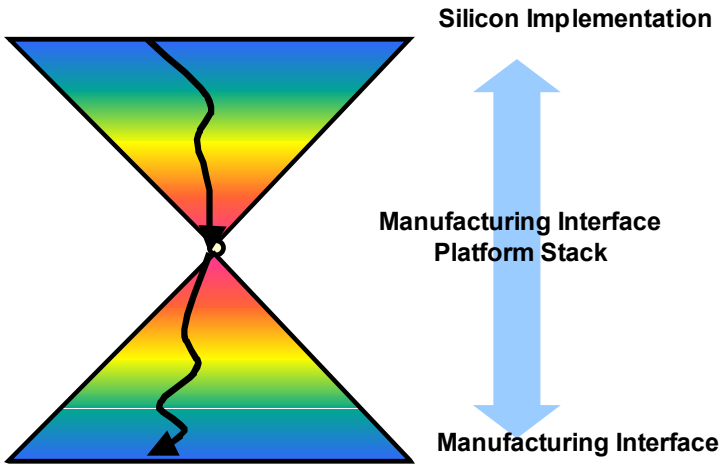

Figure 7 Manufacturing interface platform stack (source: L. Pileggi)**.**

In the SIP stack, the abstract views exported above and below are fairly well understood since they have been part of the ASIC design flow in use for years. For deep sub-micron technologies though, the accuracy of these approximations is in discussion since the actual layout of the wire-transistor pattern affects the performance parameters in such a substantial way that the decomposition into logic synthesis and detailed layout creates timing closure problems. In addition, what were considered physical second-order effects (e.g., cross talk, power distribution) a few years ago are now very important for the performance of the design and must handled via regular design structures that facilitate correct-by-construction assembly and accurate abstraction to the higher platform layers. Clearly, estimation, characterization, cost functions and constraint passing are all essential components in the design flow when we consider the final implementation of the architecture platform instance.

For the manufacturing interface, what was once a process taken for granted, it is now a critical step in controlling and predicting cost, performance and reliability. Models of the manufacturing realities must be accurately abstracted to the implementation platform, which requires new instantiations of geometrical regularity. This lowest layer of regularity provides a foundation for subsequent layers of implementation regularity.

3.6. Network Platform

The Platform-based design paradigm can be extended to levels of abstraction higher than the API Platform. In particular, it can be applied to the design of large-scale distributed systems such as communication networks.

To implement communication networks, the functionality is mapped onto a *Network Platform (NP)* that consists of a set of processing and storage elements (nodes) and physical media (channels) carrying the synchronization and data messages exchanged by nodes. Nodes and channels are the architecture resources in the NP library and are identified by parameters like processing power and storage size (nodes) and bandwidth, delay, error rate (channels).

The task of choosing an NP requires selecting from the NP library an appropriate set of resources and a network topology that defines how channels connect nodes. A broad range of options of physical channels (e.g. cable, wireless link, fiber) and network topologies (mesh, star, ring…) are usually available. Therefore, the design space to explore is quite large.

Moreover, choosing an NP is especially challenging for distributed networks due to the inherently lossy nature of the physical channels that connect nodes at distant locations. In these cases, when reliable communication is required, it is necessary to introduce additional resources (such as request/acknowledgment protocols) to overcome the effects of noise and interference. Introducing protocols requires adding processing power and memory units at the communicating nodes; as a result, the protocol components often dominate the implementation cost function and the design effort. Therefore, in selecting an NP, it is essential to balance the cost of all the different components and tradeoff between the use of complex protocols and that of more reliable (and more expensive) channels.

Channels can be defined at different levels of abstraction. Physical channels, such as coaxial cables or optical fibers, merely transport bits, while more abstract “logical” channels (e.g. ATM virtual circuit), that consist of a physical channel and a set of protocol functions, may provide more sophisticated communication services, such as in-order and reliable packet delivery.

Communication is usually described in terms of a stack of layers, where each layer defines an abstraction level and, hence, a Network Platform. The description of an NP is usually given in terms of a set of

interface function primitives that the applications running on it can use. This set of primitives defines the *Network API Platform* and allows hiding the application designer many lower layer details. Primitives typically present in an NAPI Platform include confirmed/unconfirmed data push, data request, reliable/unreliable send/receive, and broadcast/multicast send.

For example, consider the multicast-send primitive. It is a useful abstraction and is invoked when the application requires one node to address a group of nodes having a common attribute. The underlying software layer translates the symbolic address identifying the destination nodes and automatically sends a copy of the packet to all the destinations relieving the application programmer from the task to call multiple one-destination send.

4. Conclusions

The main motivation that has prompted us to develop these concepts was **design re-use and regularity to the fullest extent** due to the economics of building future electronics systems. We defined a platform stack as a stack of layers of abstraction:

1. The upper layer is the abstraction of the design below so that an application could be developed on the abstraction without referring to the underlying layers;
2. The lower layer is the set of rules that allow one to classify a set of components as part of the platform.

In this framework, the components of a platform are in general partially or completely pre-designed and the upper layer is used to decouple the “application” from the implementation of the platform. While this concept has been used for years in the PC domain, its generalization is novel. Its use in industry is now accepted. It relates mostly to architecture platforms and the Application Programmer Interface (API) abstraction as the upper layer of the system platform stack. However, because of the definition of platforms and platform stacks we use, it is possible and desirable to extend the platform concept to cover all steps of design from conception to implementation (see Figure 8). The tools and methods used to map the upper and lower layers of a platform-stack are the glue that keeps the platforms together. Hence, we can imagine a platform stack that spans several levels of abstraction of a design. To be able to choose efficiently the instances of the platform we need to deal with the constraint propagation process in the top down aspect of the design, while we need an annotation mechanism (estimation) that can be associated to the top levels of abstraction to capture lower level details. These mechanisms are an essential part of platform-based design.

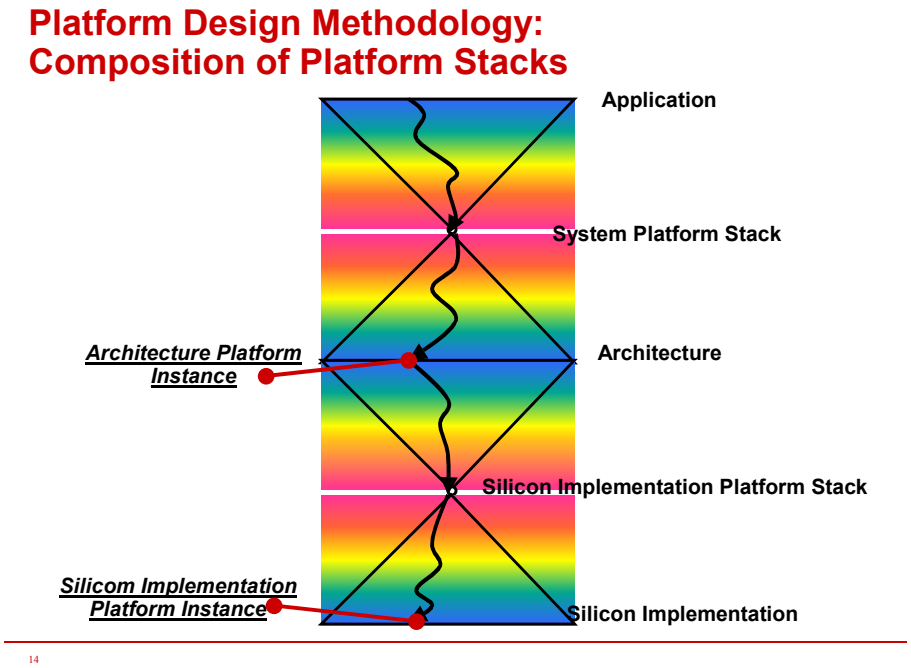

Figure 8 Composition of platform stacks

At the architecture platform level, flexibility is an essential property. We expect that programmable components will take the lion's share of the computing power of a platform, so that electronic design will be increasingly dependent on software. The leading platforms today such as Philips Nexperia for multi-media and TI OMAP for cellular phones are all based on programmable components: a micro-processor (MIPS) with a VLIW processor for Nexperia and a micro-processor (ARM) with a DSP for OMAP. However, there is a strong interest in developing an approach that can take into considerations re-configurable processors and logic, an intermediate platform between a fixed processor one and an ASIC. In addition, approaches such as the chip-in-a-day process are of great interest when performance and power efficiency are sought. In this case, the use of abstraction layers is limited to lower level libraries and the tools that generate lower level layers automatically from higher ones are of great interest. In this case, the concern of mask making and NRE costs is offset by gains in design "quality". The investigation and choice of the most appropriate level of abstractions and of the tools that support this design method, represents a core part of the GSRC research.

5. Acknowledgements

This paper is the result of many interactions with colleagues: Alberto Ferrari contributed to the first conceptualization of the system-platform stack, Henry Chang and his coauthors were among the first to conceptualize architecture platforms. Felice Balarin and his coauthors were the first to articulate the design process as a sequence of function, architecture and mapping steps. Grant Martin was instrumental in developing the concepts and mapping them in the world of embedded software. Ted Vucurevich supported the ideas and found terrific ways of exposing the basic principles. Richard Newton contributed to the overarching principle, as did the entire GSRC team. Among the GSRC investigators, Larry Pileggi contributed to the articulation of the Silicon Implementation Platform and helped correcting inconsistencies. Wojcieh Maly insisted in pointing out the importance of hardware solutions and of estimation for physical manufacturing processes. Marco Sgroi contributed the notion of Network Platforms. Randy Bryant and Kurt Keutzer provided feedback and useful editorial comments. Luciano Lavagno exposed some early inconsistencies. Jan Rabaey was an important sounding board and provided most helpful comments especially with respect to the PicoRadio design work together with strong support for the effort.

The overall effort can be seen as a summary and a manifesto for the GSRC Research program directed by Richard Newton and sponsored by DARPA and the SIA. The entire effort of the program can be summarized in Figure 9.

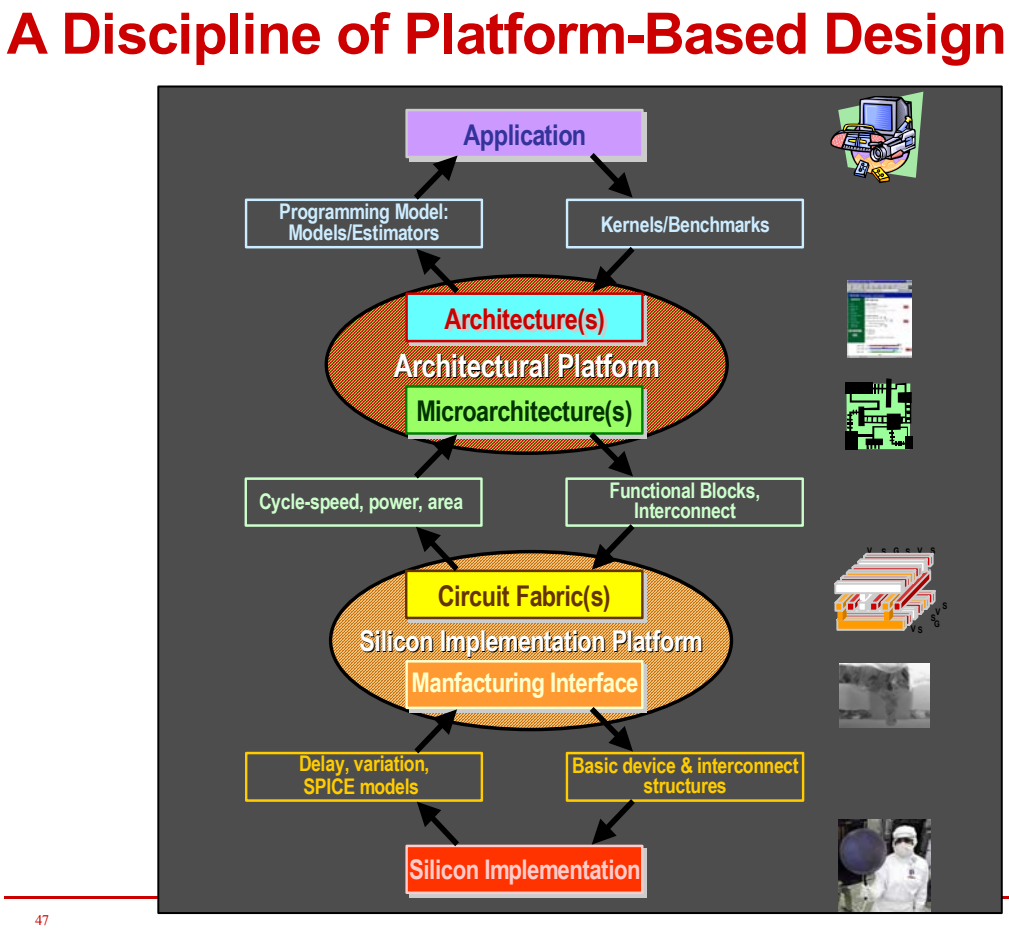

Figure 9 The overall methodology (Source: R. Newton)

6. References

Felice Balarin, Massimiliano Chiodo, Paolo Giusto, Harry Hsieh, Attila Jurecska, Luciano Lavagno, Claudio Passerone, Alberto Sangiovanni-Vincentelli, Ellen Sentovich, Kei Suzuki, and Bassam Tabbara, *Hardware-Software Co-Design of Embedded Systems: The POLIS Approach,* Kluwer Academic Publishers, Boston/Dordrecht/London, 1997.

A. Ferrari, A. Sangiovanni-Vincentelli, *System Design: Traditional Concepts and New Paradigms*, The Proceedings of the International Conference on Computer Design, ICCD '99, Austin, TX, USA, pp 1-12, Oct. 1999. (Key-note Address)

G. Martin, H. Chang, et al, *Surviving the SOC Revolution: A Guide to Platform Based Design*, Kluwer Academic Publishers, Sept. 1999.

K. Keutzer, S. Malik, A. R. Newton, J. M. Rabaey, and A. Sangiovanni-Vincentelli, *System Level Design: Orthogonalization of Concerns and Platform-Based Design*, invited paper, IEEE Transactions on Computer-Aided Design, Vol. 19, No. 12, December 2000.