Electronic systems prototyping: Tools and methodologies for a better observability.

In an electronic system development flow, a prototyping phase is very diversely valued by the electronic system engineer community. Whether it is for system-on-chip, embedded system or system-on-board development[1]*, the key question posed by prototyping is:* ***"Is prototyping a productive technique to observe my system, test it and debug it?"***

Prototype is an old companion of the electronics engineer; and as for old couples sometimes, they have a history of love and hate. Before RTL simulation became one of the most successful methodologies for design, lots of systems were developed and debugged by connecting devices on a breadboard and running extensive 'try-and-observe' test programs. Of course, EDA tools have pushed the prototyping to the late stages of the development – in theory, with the purpose to check a design already pre-validated with these tools. Recently, system board prototyping gained a new wave of interest because the constantly growing complexities of electronic system led to impractical system simulation times – at least with traditional RTL / gate-level simulation approaches[2]. In brief, to speed up test and debug, *"there appeared a need to put a piece of real hardware back in the process"*. Let's have a look at the prototype place in the development flows.

System development and prototyping: not an obvious combination

Table 1 compares the RTL/gate-level simulation approach with prototyping as a system development and debug technique. We intentionally limited ourselves to technical factors only.

	RTL/gate-level simulation	Prototyping
Max design size	Limited by acceptable simulation time.	Limited by target technology and methodology issues.
System coverage	High for purely digital system parts. Can be very low for system peripherals and environment. Limited simulation cases.	Complete if the prototype environment is realistic.
Speed	Very low	(Near) actual system speed.
Setup length	Very short. High flexibility.	Very long.
HW debug	Very good. High observability.	Poor observability.
SW debug	Limited to very short execution sequences.	Unlimited (system execution speed), realistic and very good observability.

Table 1: Comparing simulation to prototyping

Let's examine each of these characteristics.

[1] In this paper, we distinguish 'embedded system' from 'system-on-board' as follows: an embedded system is essentially (embedded)-processor-centric: it is mainly composed of an embedded processor and quite standard peripherals, with the key functional differentiator in software (generally, an application software on top of an OS). A 'system-on-board' holds more custom data processing engines, e.g. implemented in FPGA. In this case, the µP/µC role is not as predominant as it is for embedded systems.

[2] 'Traditional simulation approach' covers the VHDL / Verilog hardware simulation methodologies. ESL language and recent other modelling approaches potentially bring a partial solution to the excessive simulation run-times. Prototyping is another (complementary?) potential solution.

Byte Paradigm – info@byteparadigm.com – www.byteparadigm.com
Chaussée de Namur, 119, bte 1 - B-1402 Nivelles (Thines) - Belgium

Design size

In theory, the maximum design size is unlimited for the 2 approaches.
For simulation, the practical limiting factor is the maximum acceptable simulation time, which is a trade-off between the overall system complexity (number of gates!) and the complexity of the stimuli (number of vectors!). Needless to say, it is also preferable to run the whole on a powerful workstation. In brief, with simulation, the design size is limited by the ability to simulate it in an 'affordable time'.

For a prototype, the practical limiting factor is a matter of technology. If your target is an embedded system (that is to say, 'embedded processor-centric'), all you need is the right board with the target processor and enough memory so the code can fit. If the board does not hold all the needed peripherals, they can be plugged onto the board extension connector (if your prototyping board does not have any extension connector, buy another one!). If your target is a system-on-chip (SoC), FPGAs are interesting for prototyping. The last generations of devices demonstrate rough performances equivalent or over these of the ASIC/SoC technologies and can be developed with similar EDA tools. Nevertheless, putting a 'network' of FPGAs on a board to map your next multi-million gates SoC won't be efficient unless you:

- Discipline your design so it is as portable and as modular as possible.
- Have the right methodology (and tools?) to automate the process when needed.

The first point is all about *really* gaining from a prototyping approach. Partitioning a system onto a prototype reduces to finding an efficient way to distribute the system functionalities on computational resources and defining powerful partition-to-partition interfaces. A badly partitioned prototype leads to loss of performance; this results in poor benefit from prototyping because it does not allow running at (near) target system speed. Aside, design portability is necessary to be able to actually emulate a given algorithmic functionality on a technology that is different from the final target technology.

The second point on methodology is a matter of *productivity*. According to your design, partitioning it may be a more or less straightforward process. If the whole system fits in one single FPGA, then it is fine, because you find a perfect chip-to-chip partition correspondence. If the whole system is to be partitioned on a meshed network of several tenths of chips, then you may run into problems if you have to manually partition your system each time its architecture is updated. In this case, you'll have to consider automating this task with an RTL multi-chip partitioning tool[3].

So, the point is: *how many gates will you be able to map onto one or several FPGAs and how efficiently will you be able to do it?*

System coverage

Behind **'system coverage'** lay the following questions:

1. *Which (pro)portion of my design will I be able to emulate/simulate before the final system is available?*
2. *How accurate can I be in doing it?*

How do simulation and prototyping compare for system validation coverage?

[3] Well-known products from leading and less leading EDA industry vendors are available.

Byte Paradigm – info@byteparadigm.com – www.byteparadigm.com
Chaussée de Namur, 119, bte 1 - B-1402 Nivelles (Thines) - Belgium

Simulation allows a very efficient bottom-up system development approach: the functionalities can be separately validated before being assembled as a system. If we limit the system to its pure digital parts (roughly, the system 'gates') and some easy-to-model analogue parts, the simulation potential for system coverage is extremely high for any system type. With some time spent to browse data sheets, some calculations and engineering sweat, about any device can be modelled. The accuracy thereof goes up to the gate-level, with a resolution beyond the system clock cycle. This is one of the reasons why simulation is an essential system development technique and why it is likely to remain so. However, the reality may show a very different picture:

- Some of the system simulation models are not developed in-house; this is mostly the case for IPs. Hence, the accuracy of the simulation relies on the 3rd-party vendor professionalism and quality standards.
- A model of a system - even the most complete and accurate - is just useless if this model is not simulated against the right set of input stimuli.

The real problem with simulation as a system validation approach is to define a 'stimulation environment' that accurately matches the future system. It is really a problem of imagination: is it possible to define a sufficient set of stimuli that will place the simulated system in the same conditions as in the reality? Unfortunately, with the current increase in the total system complexity with numerous interactions between software and hardware, the system coverage with 'human-made' simulation cases is going lower and lower. Partial solutions, like verification IPs, and tools, like code coverage software, can help improve it.

In contrast, because a prototype is of purpose to emulate up to all the system functionalities with a 'real hardware' it can offer a complete system coverage if it is placed in a realistic stimulation environment (that is, placing the prototype where the final system is supposed to work). With a sufficient service time in the field, the prototype is likely to undergo 'most of the actual use cases and reveal its bugs'[4]. Of course an efficient reporting must be foreseen to spot the bugs and track them.

As a summary, simulation and prototyping differently help increase the system validation coverage: simulation is essentially deterministic, whereas prototyping adds a statistical flavour to it. This is probably why the 2 approaches may be seen as complementary.

Speed

The execution speed is certainly the major drawback of simulation. Prototyping offers an obvious solution to it because it allows running at (near) actual system speed. Between pure simulation and prototyping, techniques like hardware emulation are also currently being explored to solve this major issue of simulation while keeping its qualities. The table hereafter gives an overview of the relative speeds that can be expected in these different cases.

Table 2: Compared speed of validation techniques

	Simulation	HW emulation	Prototyping
Speed	n x 1 Hz to n x 1 kHz	n x 10 kHz to n x 10 MHz[5]	n x 10 MHz to n x 100 MHz up to real system speed

[4] By the way, guaranteeing a 'total coverage' for any system appears to be impossible – unfortunately. Even if we imagine a system entirely matches the specification – which is actually possible with a very formal development approach – it might not behave as 'expected' because of ambiguities or incompleteness of the specification.
[5] Some vendors claim up to 200 MHz on new generation platforms. As for simulation, HW emulation speed may heavily depend of system complexity.

Byte Paradigm – info@byteparadigm.com – www.byteparadigm.com
Chaussée de Namur, 119, bte 1 - B-1402 Nivelles (Thines) - Belgium

Setup length

At RTL level, whether you use VHDL or Verilog, when you set-up a simulation, you never really quit your familiar programming environment. Even if this environment is extended to other algorithmic languages such as C/C++ or system-level language like SystemC, System Verilog or extended VHDL, it is all part of the same 'algorithmic', 'formal language' world. Changing the design at this level and creating new validation cases is all about writing the right code lines, mixing the whole in a compiler and see how it runs. Gate-level simulations involve another algorithmic process – that is, generating the netlist with some synthesis tool, that maps the design onto the right technology library models. Once again, this is all done without quitting your favourite computing station and the whole can be automated with the right tools and the right scripts in between.

Setting up a prototype to validate some precise part of the design can be really uneasy. Fundamentally, there is a *rupture* between the development environment and the prototype environment. And there come most of the issues from. Prototyping is the world of labs, with intensive use of multimeters, oscilloscopes, logic analysers, protocol analysers, pattern and waveform generators. Installing a validation and debug environment with a prototype even involves considering *developing new tools* to generate stimuli or to control stimuli generation. The question that often arises is: *'How am I going to extract my validation results from my prototype?'*

Whereas an algorithmic language offers objects, commands and tools for any validation on any design, going to a prototype often involves redefining a new environment for each new system development. As a major consequence, going on prototype always brings the risk that your engineering team ends up designing *for the prototype environment* and *not for the target system itself*.

HW debug / SW debug

In a simpler world, simulation would be dedicated to hardware debug and prototyping would be reserved for software debug. We must admit that each approach finds its most successful applications when hardware and software are not too much mixed all together.

For pure hardware debug, under the condition that a 'reasonable gate count' is involved, simulation brings full visibility over the hardware and the maximum accuracy over the system 'time space', up to the clock cycle (and even beyond).

For (embedded) software debug, under the condition that a prototype board with the adequate processor and set of peripherals exists (if it not the case, you'd better select another processor), a particular form of prototyping (software emulation) also brings a good visibility over the system and also the maximum accuracy over the 'time space' – that is, in this case – up to a single instruction execution level.

In both cases, the system engineer has got a sufficient grasp of his design and he is able to debug it with the adequate execution time resolution. The tools are there, the methods well known, and the process is really successful.

Things get worse when you start designing larger systems, with less standard hardware and that also feature a great deal of custom software. Things go even more critical when your system development requires from you to buy 3rd party functionalities in the form of intellectual properties (IP) or stand-alone modules (mezzanine boards ...).

Byte Paradigm – info@byteparadigm.com – www.byteparadigm.com
Chaussée de Namur, 119, bte 1 - B-1402 Nivelles (Thines) - Belgium

In these cases, there is a need to verify your architecture choices, select architecture configuration options, possibly evaluate technologies unknown to you and validate if your software and your hardware harmoniously work together. In other words: *check your system as a whole!*

In theory, you can use RTL and gate-level simulation to run your software on a model of your processor. You'll end up watching your simulation run during months… just to start executing a few instructions after system boot! So, for normal execution sequences, you can just forget about RTL simulation[6].

Actually, prototyping a system is not such a straightforward solution for HW/SW validation. When your system is processor-centric, then your embedded processor implements most of your system functionality and you can use it efficiently to validate and analyse the whole system (together with standard I/O peripherals like a terminal connected to an UART, a few GPIOs, and some peripheral like a network connection or a video output). But when your system is built on multiple heavy-processing engines, multiple system bus masters and complex random data flows between them, you may end up having to track a bug about anywhere… and you don't have any easy way to get to it. While software emulation offers about the same observability on software execution as the one you get from your common debug environment, prototyping is far being the perfect counterpart of a RTL or gate-level simulation.

Complementary techniques?

As a summary, here are the key points we have learned from the above comparison:

- About all the simulation limitations come from an execution speed problem: it limits the acceptable design size that can be simulated and makes simulation impractical for validating embedded software[7].
- Speed of execution is certainly the key advantage of prototyping for software validation.
- Porting a design onto a prototype board requires the proper **methodology and tools**, especially at the definition of the functionalities interfaces. This issue is especially critical to fully benefit from the prototype potential performances and may limit the productivity of a prototype approach for big and complex designs.
- The major drawbacks of prototyping are the **prototype setup length** and the **hardware observability**, 2 aspects in which simulation performs very well. Hardware observability is especially critical for non-processor centric systems with intensive integration of custom hardware and software together with 3rd-party functionalities.
- Prototyping is an interesting approach for a 'statistical validation' of the system - that is: placing the system being developed in a realistic environment that reproduces the complexity of the stimuli it has to handle. This is probably one of the major reasons why you *always* have to consider a prototyping approach during system development.
- **'Environment continuity'** is one of the essential characteristics of a successful validation experience. This is why simulation is successful for setting up validation cases on hardware and observing results from them. This is also one of the main reasons of the success of software emulation as embedded system development and validation approach.

[6] In fact, this drawback is just a consequence of the speed issue in simulation.
[7] Isn't it, by the way, another 'size problem', not as silicon gates, but software lines?

Byte Paradigm – info@byteparadigm.com – www.byteparadigm.com
Chaussée de Namur, 119, bte 1 - B-1402 Nivelles (Thines) - Belgium

If we consider simulation and prototyping only[8], this list of relative strengths and weaknesses brings an obvious conclusion: simulation and prototyping are essentially complementary development approaches. Well, this is not really a revolutionary finding.

However, what we see happening today is that the simulation drawbacks get worse with increased system complexity. Therefore, there is a need to constantly improve all complementary techniques - such as prototyping - in order to fill the so-called 'productivity gap' in microelectronic system development.

Bottom line - a 'bird view' on a prototype

On figure 1, we tried to show a representative prototype, with possible custom and standard components.

As described in the previous sections, since **hardware observability** is a major concern for a prototype, this figure features the possible **prototype access points** through which the system can be observed for validation. They are listed in table 2 with more details.

Table 3: Prototype possible access points

Access point	Example	Type
Custom / Standard high-speed serial interface	SATA, PCI-Express, Infiniband, … connection	Board-level, functional high throughput data transfer port
Networking / Datapath interface (non high-speed serial)	Ethernet connection	Board-level functional communication port.
Backplane interface	PCI (PCI-X, PCIe) connection	Board-level, functional control & communication
System peripherals	16C550 UART, GPIO, …	Board-level / chip level (when embedded) functional resource
JTAG interface	(obvious)	Board-level debug & programming port; chip-level debug port that use JTAG protocols
Board debug / LA connector(s)	Standard pin connector, specific LA connector, linked to a relevant set of signals	Board-level dedicated debug resource
Device functional IOs	(obvious)	Chip-level, functional and debug resource (the IO must be accessible with a connector)
Device standard debug port	Microprocessor JTAG port	Chip-level standard debug & programming port.
Device custom debug port	Reserved FPGA pins connected to internal functional signals	Chip-level, custom debug port
On-chip Instrumentation (OCI)	FPGA resources dedicated to gather data from the FPGA internal nodes and control a custom debug port to make them available	Chip-level, custom debug resource
Embedded software	Application running on embedded microprocessors and holding debug and diagnostic routines	System-level, software

[8] For instance, an approach like 'hardware emulation' – often purely dedicated to SoC development – can also be considered as an interesting development technique for system development 'with a piece of hardware in it'.

Byte Paradigm – info@byteparadigm.com – www.byteparadigm.com
Chaussée de Namur, 119, bte 1 - B-1402 Nivelles (Thines) - Belgium

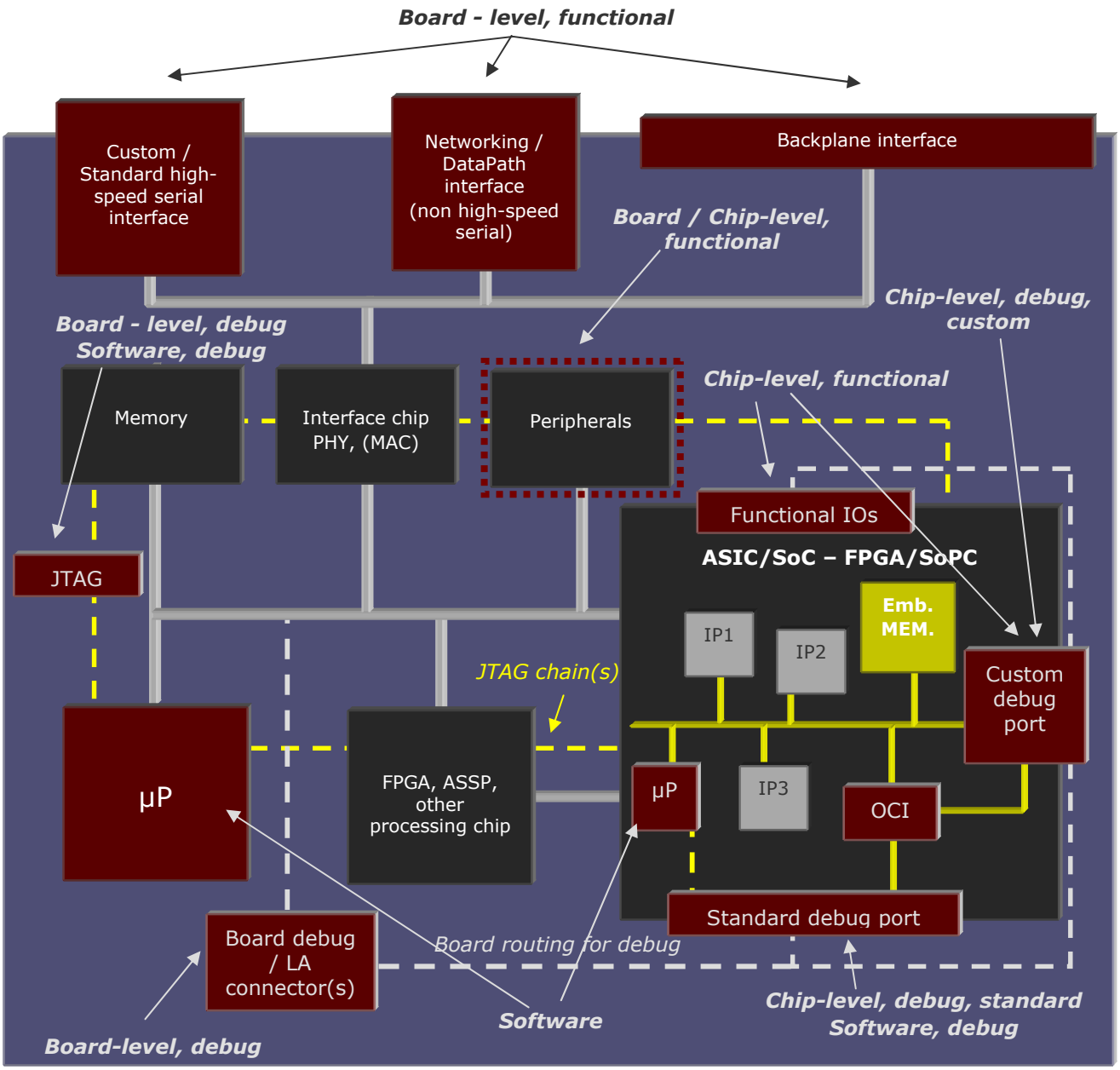

Figure 1: Bird view on a prototype

Combining approaches & choosing the right tools

At Byte Paradigm, we believe that successful prototyping approach lies in the combination of many techniques and tools. Stand-alone oscilloscopes, logic analysers, protocol analysers, JTAG probes, and software emulator pods, OCI techniques, ... when properly combined, enhance your validation phase.

To really benefit from this combination, prototyping requires thinking *from the start* about the access points you'll use to *observe* your system. A prototype with too few access points leads

Byte Paradigm – info@byteparadigm.com – www.byteparadigm.com
Chaussée de Namur, 119, bte 1 - B-1402 Nivelles (Thines) - Belgium

to poor validation productivity because *when you'll track a bug, you'll want to observe its effects from different angles.*

Moreover, as stated before, tools that reduce your prototype setup length and reduce the rupture between the 'development environment' and the 'prototype environment' increase the efficiency of the prototyping approach.

Byte Paradigm is committed to deliver PC-based instruments to test and debug electronic and embedded systems. Applied to design, test on post-manufacturing prototypes and on-the-field maintenance, our PC-controlled products offer fast setup, multiple interfaces and a rich set of functionalities.

http://www.byteparadigm.com.

About the author
Frédéric Leens is Sales and Marketing Manager at Byte Paradigm.
He can be reached at: frederic.leens@byteparadigm.com