Hardware/Software Co-Design - from Academia to Industry

Objective

Hardware/Software Co-design has been mentioned quite a bit recently. However, the exact meaning of the co-design may not be obvious to many people, and the term co-design is sometimes used to loosely refer to hardware/software collaboration or co-verification. Having a Ph.D. on hardware/software co-design, I feel it may be useful to explain what hardware/software co-design is in the classical sense; what the goal of the co-design is in academia research; what the practical state is in the industry; and how to perform efficient co-design.

After reading this article, you can grasp a high level overview of the entire system design flow, with an emphasis on the hardware design part. The co-design assumes that the software can be easily adapted to the changing hardware.

The following sections contain some technical terminologies that may not be familiar to some software engineers. I have highlighted them with some short explanation. When discussing co-design with other people, those terms are often referenced. Throwing them out in the discussion makes you a pro :)

What is hardware/software co-design in one sentence?

Almost all current complex electronic systems contain both the software and hardware components. The hardware components implement the atomic computation on some specific functionalities, while the software components compose the atomic computation to realize more complicated functionalities.

When designing such systems for a set of applications, it is inevitable that parts of the applications are executed directly on hardware, and parts of the applications describe the order of execution in software. The process of deciding which part falls into hardware and which part falls into software while designing a system is called **hardware/software partitioning**.

Hardware/software co-design *is a design methodology that simultaneously design the hardware portion and the software portion of a complicated electronic system from a unified specification, with the hardware and software being partitioned and re-partitioned through most of the design cycle.*

Below, I describe what the hardware/software co-design is in the ideal scenario (in academia), and how it is actually used in industry.

In academia, the ultimate recipe for complicated electronic system designs

In simple electronic system designs (prior 1990), the design of the software system doesn't start until the hardware implementation is done. That creates several issues:

- The design cycle is very long since the hardware and software are designed in sequence.
- Design errors in parts of the system may not be uncovered until very late.
- Parts of the system may be over-designed or under-designed, since there is no early evaluation of the design options.

Starting early 90s, researchers are looking into designing the hardware and software in parallel and develop the co-design methodology over the years. The problem they try to solve is as follows:

Given an application or a set of applications, design a system that satisfy two of the following three constraints: performance, power, cost (area), while optimizing the other. Additional constraints such as time to market also need to be satisfied.

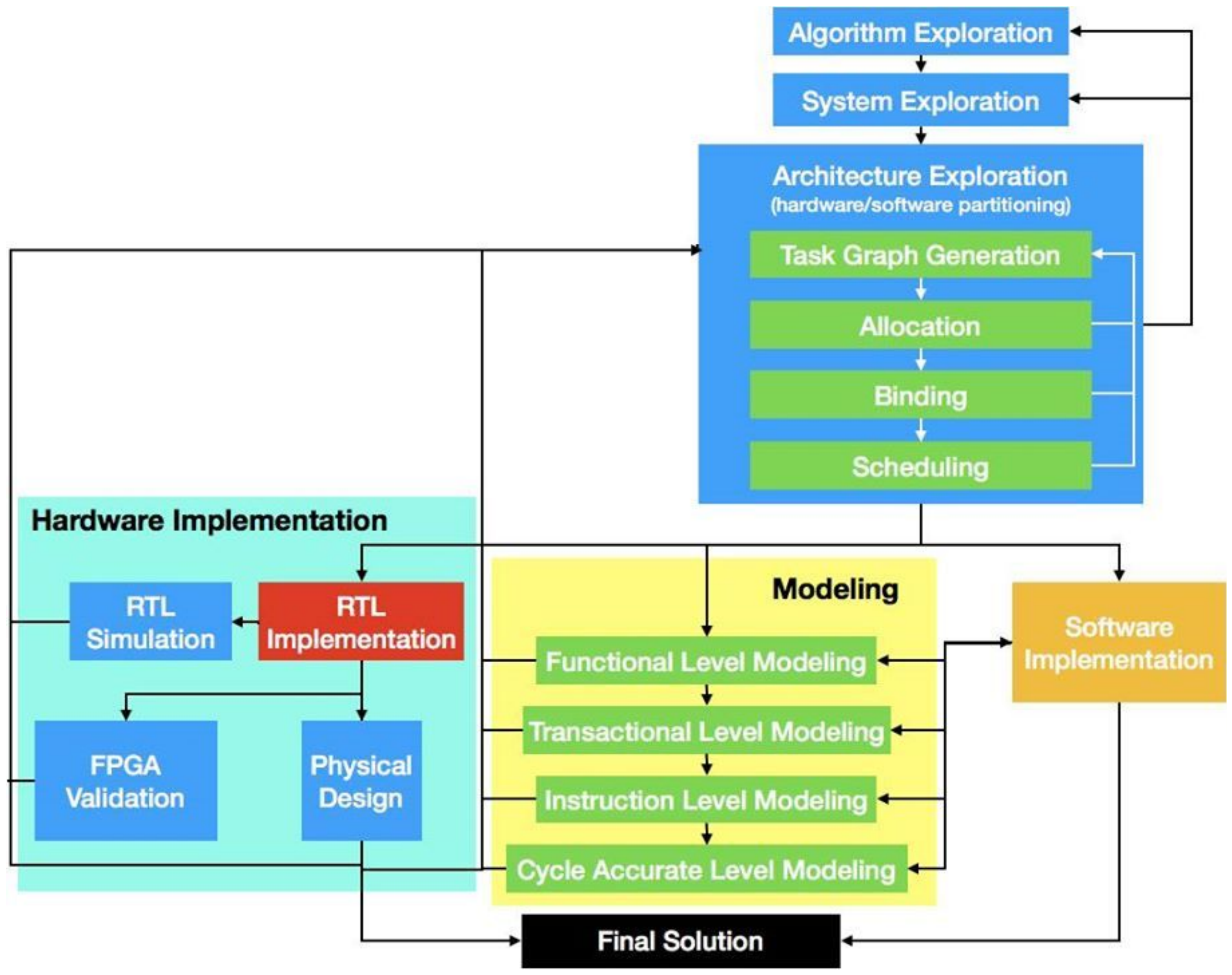
An extension of this problem is to design a number of systems that forms the pareto-optimal frontier of the performance, power, and area. **Performance**, **power**, and **area** are the three frequently referenced optimization objectives in system design, often abbreviated as **PPA**.

The main benefits of the co-design is the opposite of the earlier designs:

- The design cycle is reduced since the hardware and software are designed in parallel.
- The designs are verified early to avoid high-cost mistakes.
- The design options are evaluated early to avoid under- and over-designs.

The main drawback of the co-design is the effort. It may easily double or triple the effort of an optimal sequential design. However, since the design is unlikely to be optimal for the first time, the effort is well paid off.

In the most ideal scenario, the co-design flow is composed of the steps in the following diagram. I will explain the steps one by one.



Algorithm exploration

The first step is to explore the possible algorithms to implement the application. At this stage, there is no consideration of the physical hardware. In deep learning domain, deciding which ML model to use for the application belongs to this stage. Deciding whether the convolution is implemented using winograd also belongs to this stage.

System exploration

After the algorithms are roughly decided, the system exploration step decides the hardware/software features in order to realize the algorithms. This stage answers questions like:

- Do we need wifi, usb, serial or parallel ports etc. for this system?
- Do we need ECC or parity for memory?
- Do we need performance counters? How do we use the performance counters?

It determines the features or functionalities to support in the system, but leaves how to support them in the architecture exploration stage.

Architecture exploration

The next step is the architectural exploration stage. It is sometimes referred as **design space exploration** (**DSE**). This is the key stage of the co-design. In this stage, the hardware and software are partitioned. A lot of other stages (described later) provide refined inputs to this stage, so that the partition can be performed with more accurate data. If, during exploration, the algorithm is found inefficient, the design feeds back to the algorithm exploration stage and change the algorithm. The output of this stage already separates the software part and the hardware part. It also includes the schematics on how the software runs on the hardware.

This block performs the actual co-design. All the blocks below in the diagram form **hardware/software co-verification**. That is a feed forward flow with the design verified at every stage. If the design constraints are violated, however, the design is changed locally without going back to the partition part.

Let's ignore the blocks inside it for the moment and move on.

Software implementation

Since the software part is already isolated, they can be implemented, by human manually or by tools automatically. If some part of the software is tied to a particular hardware, models of the software may also be generated to speed up the design verification.

Modeling

The hardware part is delivered to the modeling team to generate the hardware/architecture models with various accuracy. The higher the abstraction level, the simpler the model is, the faster the model runs, and a larger design space can be covered quickly (consider 10x speedup for every level up in hierarchy). The modeling serves dual purposes: the validation of the design, and the collection of the metrics for design space exploration.

- **Function level models** generate correct input/output of the hardware only, with nothing else.
- **Transaction level models** in addition monitors the actual transactions between the hardware blocks. At this level, the amount of data being transferred between the hardware blocks can be estimated. The actual runtime can also be roughly estimated with a large error.
- **Instruction level models** provide instruction level details for the processor and co-processor. The number of instructions is known and provides even more detailed

estimation on the runtime. Sometimes the cache behavior can be estimated at this level also.

- **Cycle accurate level models** are most accurate, but also are the slowest. They model the system cycle by cycle.

Running those models together with the software is called **hardware/software co-simulation**. At each level, a large amount of data is fed to those models to verify that 1) the architecture generates the correct result, and 2) the performance is inline with expectation. Thus, those models also serve as verification tools in the design.

The collected data at one level are fed back to the architecture exploration stage. The architectures are refined, new models are generated, and new data are collected etc. This loop is repeated until the designer is satisfied with design, and a lower level model is used to repeat this process.

This is unique in the co-design. Except some uncommon cases (e.g. the hardware is generated from the models), the sole purpose of the modeling is to collect more detailed runtime data. None of it will be used in the final solution.

Hardware implementation

Once the designer is satisfied with the design through modeling, the hardware implementation team starts to implement the actual hardware. In a lot of scenarios however, this process is in parallel with the modeling. They work on some parts less likely to be changed by the modeling first and then move to the riskier parts.

In some uncommon flow, the hardware can be implemented automatically using **behavior level synthesis** (or called **high level synthesis**), which takes a behavior description of the hardware, manually written or automatically generated during the architectural exploration, and the tools synthesize it to a lower level called **register transfer level** (**RTL**). However, this flow is not mature, and not widely adopted.

Usually the hardware team implements the RTL **HDL** (**hardware description language**) manually. Once it is implemented, the RTL code can be simulated and provides even further detailed data and thus feed back to the architectural exploration stage. The power may also be estimated at this stage with some error. Estimating power in absolute terms above this level may have limited success in restricted architectures, but in general is not very meaningful. Higher level power estimation is usually for the purpose of comparing relative quality of two architectures.

If only part of the RTL is implemented, it may still be simulated with the models for the rest of the system. That is called **mixed hardware/software simulation**.

The RTL code may be mapped to FPGA for validation. The result may affect the architecture of the design. It also serves as a verification tool.

The RTL code may go through the **physical design flow** (e.g. RTL synthesis to get the gate level netlist, placement and route, and signal integrity). This is quite standard in the hardware design flow so I won't explain the details, but at the end of each step, performance, power, and area can be more accurately retrieved to validate the design. If some aspects exceed the budget or the constraints get violated, first local design changes are performed to remedy the situation. If unsuccessful, the design needs to go back to architectural exploration stage to partition the hardware and software again. This is a very severe situation since the feedback loop is very long. In many cases, it means the design needs to be delayed, unless some very efficient tools are developed to cycle the entire flow quickly. Before co-design, this is the main feedback loop. With co-design, fortunately, the models at different levels can provide feedback much quicker and the likelihood of this feedback is reduced significantly.

Once all the criteria are satisfied, the software and the hardware work together to finish the complete system.

Architecture exploration revisited

Understanding how the overall flow works, let's revisit the architecture exploration stage. In this stage, the design space is explored to find out the best implementation for the application.

The design space can be explored in many different ways. It can be explored manually, based on some experience and gut feel of the outcome. It can also be explored automatically via tools. Or the two can be combined together. Below, I will describe the design space exploration in a formal flow. The exploration based on experience usually intentionally or unintentionally falls into the flow anyway.

The first step is to divide the algorithmic description of the application to some densely connected parts and some loosely connected parts. A densely connected part is usually called a **task** and is treated as an entity. The loosely connected parts are the connections between the tasks. (Isn't this a clustering problem?) Usually this kind of division is done manually with some higher level understanding of the algorithmic characteristics of the application. This kind of abstraction forms a **task graph**.

Then comes the **allocation** step. In this step, the designers need to first figure out the available **processing elements** (**PEs**), which perform the actual computation on the tasks. Example of such PEs are: pre-existing (off-the-shelf) IP blocks such as RISC-V processor, DSP, GPU, CPU, FPGA, coprocessor, as well as custom hardware specifically designed for a few tasks such as hardware synthesized from the tasks. The designers also need to figure out the available connection mechanisms between the PEs, such as bus, point-to-pint, networks-on-chip, queues etc.

After that, the designers collect some metrics (performance power, area etc.) of running every task to every available PE. For each of the metrics, a table is generated with each row for one task and each column for one PE. If a PE cannot run a task, the corresponding cell value is infinite. Similar metrics are collected for the communication between each pair of PEs, and form more tables. Usually the communication cost is assumed to be only dependent on the amount of data and not on specific task. All the modeling and hardware design stages that feed back to this stage generate more detailed data in the tables.

After that is the **binding** step. In this step, the tasks are bound to the PEs, which means the PE each task runs on is determined. It is usually done by some heuristics or ML models analyzing the tables formed in the allocation step.

After that is the **scheduling** step. In this step, the sequence of running the tasks on those PEs is determined. It is also usually done by some heuristics or ML models analyzing the same tables.

Depending on the system's requirements, the binding and scheduling can be done online (the decision is made at runtime during the actual task execution), or offline (the decision is made statically regardless of the dynamic task execution situation).

The binding and scheduling can be done in sequence separately, which may have the phase ordering problem. Or, they can be addressed simultaneously in one pass iteratively, which is a much more difficult problem to solve. As an analogy, they are similar to the register allocation and instruction scheduling problems in the compiler domain.

Please note, the design space can be explored hierarchically. Within a task, the same process can be performed again to form finer grained task graphs and make finer grained trade-offs, but that result also affects the higher level coarser grained evaluations. The problem is extremely complicated.

In some setup, this architectural exploration stage is called **system level design**. If this stage is performed automatically by tools, it is called **system level synthesis**. If all (or most of) the blocks in the above diagram can be done automatically, it is called **hardware/software co-synthesis**.

I consider this stage different from **electronic system level**, which I view it as anything above RTL, verification included, which includes this stage and the modeling stages in the diagram.

So far, I've described the entire hardware/software co-design flow. Next, let's see how it is adopted in industry.

In industry, a reality check

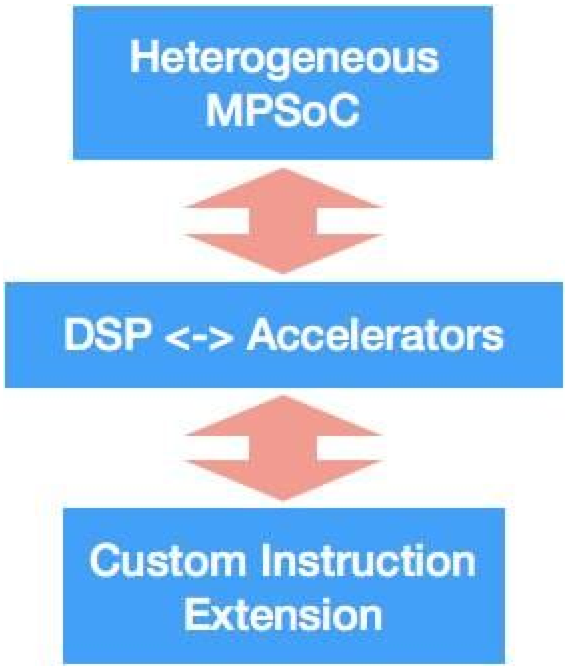
Even though the co-design flow has been introduced for over 20 years, the flow is still not completely adopted in the industry. On the one hand, parallel design of the hardware and the software of the system is the industry norm now a days. On the other hand, the design space exploration is still limited, mostly guided by the experience of the designers and the considerations of non-technical factors.

Over the years, the **EDA** (**electronic design automation**) industry has tried really hard to build tools to automate every block in the diagram above. They have achieved great success in the physical design, have made significant progress in the higher level modeling and verification. But they have made limited progress in the higher level design space exploration.

Below I describe what I think the challenges of the co-designs are. It is mostly based on my experience at Tensilica and reading public publications. It may not cover the entire industry landscape.

The design space is huge

As I have explained earlier, tasks graphs are hierarchical. When we make the design choices concrete, it may be nailed down to three levels: heterogeneous MPSoC, DSP plus accelerators, and custom instruction extension, as shown in the figure below:



At the top level, the designers need to choose from a basket of heterogeneous IPs and assemble them to a system, mapping the top level task graph to it. Whether to choose a particular IP depends on the internal architecture of the IP. The IP can be further divided into a more general purpose processor (GP, such as DSP) and one or multiple hardware accelerators serving as co-processors. Each of the tasks mapped to each of the IPs is further broken down to a smaller task graph. Decisions need to be made to decide which tasks remain in the GP and which tasks are off-loaded to the accelerators. The GP can be further divided into the basic instruction set and custom instruction extensions. Again each task mapped to the GP is divided into a smaller task graph, and some tasks are mapped to the basic instructions while others are mapped to custom instruction extensions.

In the most ideal scenario, consider the boundary of hardware and software very fluid (i.e. the hardware accelerators and custom instructions can be generated effortlessly). The boundary may flow back and force in all three levels at the same time. There is a huge design space to explore at each level alone, not to say cascading them together. The higher level decisions depend on the lower level decisions. But once the higher level decisions are changed, the lower level decisions need to be revisited. This is a big feedback loop! (This is my Ph.D. btw)

Practically, life is much simpler. Since virtually no tool exists to facilitate exploring the combined design space, companies jump to one of the levels and dig deeper in that level. For example, CEVA goes with the DSP plus accelerators approach; Tensilica goes with the custom instruction extensions approach. The system integrators have to choose one or the other, but not both. When they make those decisions, they have little data. But once they make the decisions, there is no turning back, because of the huge capital invested in the tool chains and the human cost learning the tool chains. The choice becomes a belief, a religion. This choice leaves huge design improvement opportunities on the table, but with limited time, budget and poor tool support, this is the best they can do.

The effort of modeling is non-trivial

Quickly designing/generating the models at each level is critical to the fast iteration of the co-designs. However, designing the models and maintaining multiple levels of the models is very time consuming, tedious and error prone.

Usually designs start from a higher abstraction level and iterate. Once the designers are satisfied with the result, a lower level model is designed to provide more detailed data. This is where the complication starts:

- Should the designers make the effort to make sure that all changes on the design are reflected at both levels?
    - The effort is huge, when there are 4-5 different levels.

    - The effort is not well rewarded, as the higher level model no longer provides much value for the current design if no major changes to the design are needed.
        - What if there are major changes to the design?
        - What about the design of the next generation?
- Should the designers just deprecate the higher level models and focus on the lower level models alone?
    - How do we know the lower level model is designed correctly?
        - Verifying the higher level model and the lower level model with the same data is a good verification method.
        - Maybe the higher level model and the lower level model can be formally verified.
    - What about the design of the next generation?
    - What if there are major changes to the design?

Facing those problems, companies try to make the model design easier, or generate the models automatically.

systemC is developed to make the model design easier. It uses a top-down approach, and is more focused on **transaction level modeling** (**TLM**). Cycle accurate models can be designed and some tools can synthesize the models to the actual RTL implementation.

SystemVerilog combines synthesis and verification in one language. The verification can be used to model the system components and extract useful metrics. It uses a bottom-up approach and the design environment is very similar to the hardware implementation environment.

At Tensilica, the reference models of the custom instruction extensions can be written by the designers, and the cycle-accurate level, instruction level, pin level, transaction level, and function level models can be automatically generated from the reference models. The reference models can also be formally verified with the actual semantic implementation. But that only works on a small subset of the design space the Tensilica processor enables.

Even with the tools' support, modeling is non-trivial amount of work.

Some long feedback cycles still exist

The goal of co-design is to enable fast iteration on the design space exploration. However, due to the deficiency of the tools, some metrics can only be extracted at lower levels. For example, it is difficult to model power above RTL; and it is difficult to measure timing (frequency) and area above gate level. They all require the actual hardware implementation to be in place to get the data. But those are the metrics we want to optimize! What do the higher levels do? They still provide useful data to make design decisions:

- Function level: whether the design actually works.
- Transaction level: whether the amount of transferred data is reasonable.
- Instruction and cycle-accurate levels: whether the application can finish in a reasonable amount of time, assuming a target frequency.

However, the feedback path for the physical properties are still long!

There is no good solution here. Some hacky methods are used, such as implementing non-functional blocks with the ballpark of the datapath and some dummy control logic, feeding random data, just to get the frequency/area/power estimation. Some other solutions like using high-level synthesis tools to speed up the design process and retrieve the metrics are not yet mainstream.

Tools still need to be improved to reduce the long feedback path.

Trimmed down co-design

Ideally, the designs should be performed proactively. Multiple designs are explored with different strengths and weaknesses. Sub pareto-optimal ones are pruned out. This, however, is not feasible without significant tool support to automate the entire design process. It is just a far fetched dream for now.

The other extreme is that the designs only contain a forward pass. The design decisions are made with limited data, mostly based on the designers prior experience. Then the designs are implemented based on the initial spec. In this case, it is not useful to perform modeling so only the hardware designs are implemented. It is still a co-design since the hardware/software are partitioned once. However, it lacks the essential feedback path and still exhibits issues in the sequential designs.

In industry, it is a common practice to perform reactive design. The designers have a vision of the system, and partition the system to software and hardware accordingly. If later the design turns out to be insufficient in some aspects, the design is first modified locally in the hardware or software side alone. If the local modifications are insufficient to compensate, the software and hardware need to be re-partitioned and the design re-implemented... and the loop continues.

Due to the implementation cost, the design space is not thoroughly explored. Thus, the experience of the designers are especially important to ensure only the most promising areas are explored. Many experienced engineers, even if they don't know the co-design methodology, can still perform reactive designs very well.

System integration

The companies doing system integration are usually different from the companies providing the IPs. Even in very big companies, the teams doing system integration may be far away from the teams providing IPs. Thus, the integration teams don't have a lot of freedom in making the trade-offs, if they just assemble the off-the-shelf IPs together. Their input to the co-design is to decide which IP to put in the system (but that decision may not be technical). This is usually a feed forward process, deciding which IP performing which task and make sure the IP satisfies the constraints. Thus, detailed co-design usually happens inside the IP, but not in the system integration. If, one day, the hardware and software need to be repartitioned at this level, something really bad has happened.

However, the decisions the system integration team make have long lasting effect. The decision may not only influence the co-design at the IP and below levels, but also the software partners using the IPs and even the end customers of the software partners. Once the ecosystem is built in both the software and the hardware, it is very difficult to change.

The decisions of the system integrators may last multiple hardware generations. This makes the first generation product very critical to design right.

How to perform efficient hardware/software co-designs?

Below I describe what I think forms the foundation of efficient hardware/software co-designs.

Good Understanding of the co-design methodology

I have explained above the entire co-design flow with detailed description on each block. Understanding what the co-design entails is critical to the success of the co-design.

However, it doesn't mean the flow needs to be followed strictly. In industry, no one follows the complete flow. It is important to leverage the right tools on the feasible blocks and use human expertise, skill, intuition to connect the missing blocks.

It is perfectly fine to perform trimmed down reactive co-designs, as long as the risks are aware by the designers.

Consideration of the difference between the hardware designs and the software designs

Hardware designs and software designs are completely different beasts. Teams at either side need to be considerate to the other side. Engineers familiar with both sides can serve as lubricants to reduce the friction and tension between the two sides.

Here I use two examples to illustrate the differences.

Example one

In software, the code written is mostly sequential. Even in multi-threading code, the way of thinking is mostly sequential. Only need to consider fork, join, and semaphore in some critical parts of the code.

In hardware, however, circuit goes everywhere at the same time. The design is inherently parallel. It is against the human brain's design, which tends to think sequentially. Thus, even though people can write complicated software with ease, people have difficulty implementing a slightly complex **FSM** (**finite state machine**). The way people attack this problem is to use divide-and-conquer, to separate complex FSM to simpler FSMs. Then for simple FSM, a lot of smart engineers get together to draw the **state transition diagrams**, waveforms, and review them again and again. They build a big verification team to simulate the module with all kinds of inputs, especially the boundary inputs. In some critical parts, they even have a separate group of people designing the same FSM independently and formally compare the two.

As a side note, a less obvious benefit (and less appreciated benefit by the customers) of the Tensilica's approach to hardware design is that it eliminates the FSM design part, which is the most error prone part of the design. The designers only need to think sequentially on the datapath. That make it possible for a software engineer to write reference hardware code.

Example two

In software, engineers are very savvy, and want to use the most updated tools and languages. It is not uncommon for people to use C++11 or C++14 features. The cost of error in software is very low and people use a try and fix approach.

In hardware, engineers are very conservative. The most recent tools means the tools having most bugs. It is not uncommon for hardware companies to stay one, two, or even three generations behind.

Three or four years ago at Tensilica, I once used a simple frontend Verilog2001 construct, thirteen years after the language standard was released, and that was the officially supported language by the company. However, it triggered a heated discussion within the company. Then it was found out that some part of the vendor tools did not support that construct. The RTL simulation speed was ten times slower. When that piece of code appeared in the critical path after synthesis, everyone blamed the usage of that “new” construct. I had to reluctantly revert back to use the Verilog95 standard.

The lesson I learned from the experience was that the EDA tools might not be polished even decades after the language was adopted. So, stick with the oldest, most used features of the language.

This problem is usually not severe in the established hardware companies, but it might be a challenge for a software centric company entering hardware design for the first time.

Fast feedback between the hardware and the software teams

The goal of the hardware/software co-design is to facilitate fast metric data collection and fast design space exploration at the cost of the extra modeling work. The communication channels between the hardware and software teams need to be smooth. It is better to have at least one person understanding both sides to attend meetings on both sides. Distill the requirements and the collected data, and pass them to the other side appropriately and efficiently.

Reasonable expectations on the co-design methodology

Co-design is not a silver bullet that can handle everything. It is more of a guidance that needs to be refined and re-tuned in every concrete situation. With the limited tool support, the co-design methodology is ad-hoc carried out in the industry. It cannot replace the experience of the engineers. On the contrary, it exemplifies the need of experienced engineers in the whole process.

Final words

In ISSCC 2018, David Patterson projected the future of processors is to go domain specific. That kind of architectures have been practiced in the embedded area for many years. But with the whole industry shifting towards that direction, such architectures will be in the spotlight once again.

The resurgence of the domain specific architectures may also mean the resurgence of the hardware/software co-design flow, which is a perfect flow to search the design space for a few carefully selected algorithms.

Let's embrace a new era of hardware/software co-design.