2022
DESIGN AND VERIFICATION™
DVCON
CONFERENCE AND EXHIBITION
EUROPE
MUNICH, GERMANY
DECEMBER 6 - 7, 2022

# Challenges and Solutions for Creating Virtual Platforms of FPGA and SASIC Designs

Kalen Brunham, Programmable Solutions Group, Intel, Toronto, Canada (kalen.brunham@intel.com)

Jakob Engblom, Software and Advanced Technologies Group, Intel, Stockholm, Sweden (jakob.engblom@intel.com)

***Abstract*—Virtual platforms (VPs) built on frameworks like Intel® Simics® software and QEMU are standard for application-specific integrated circuit (ASIC) designs, achieving concurrent hardware-software development, and software and integration shift-left. As field programmable gate array (FPGA) and structured ASIC (SASIC) designs are becoming increasingly complex and include processor cores, designers see the value VPs bring for software development before the hardware design is complete and for the ability to validate software without needing actual hardware boards. However, the creation of a VP for an FPGA/SASIC design presents unique challenges, given the inherent flexibility of the hardware and the existing design workflows. This paper reviews the challenges of creating VPs for these designs and presents tooling and workflows for FPGA/SASIC customers to create VPs for their designs.**

***Keywords*—*FPGA, Structured ASIC, Virtual Platform, Simics***

## Introduction

A virtual platform (VP) is a model of a hardware system that can run the same software stacks as the hardware platform it models [1]. A VP simulates the target processor cores and the rest of the software-visible hardware, where the underlying simulator software can run on a host computer that is different than the system being modeled. Many large application-specific integrated circuit (ASIC) engineering companies, including Intel, have adopted VPs as part of their standard engineering practice [1], since VPs enable software development and test before the hardware design is finalized, and regression testing of software changes without needing large numbers of physical hardware boards [1]. These companies also use VPs early in the hardware design phase to collect feedback from the software teams, allowing changes to be easily incorporated into the design and helping to avoid costly discoveries post-silicon.

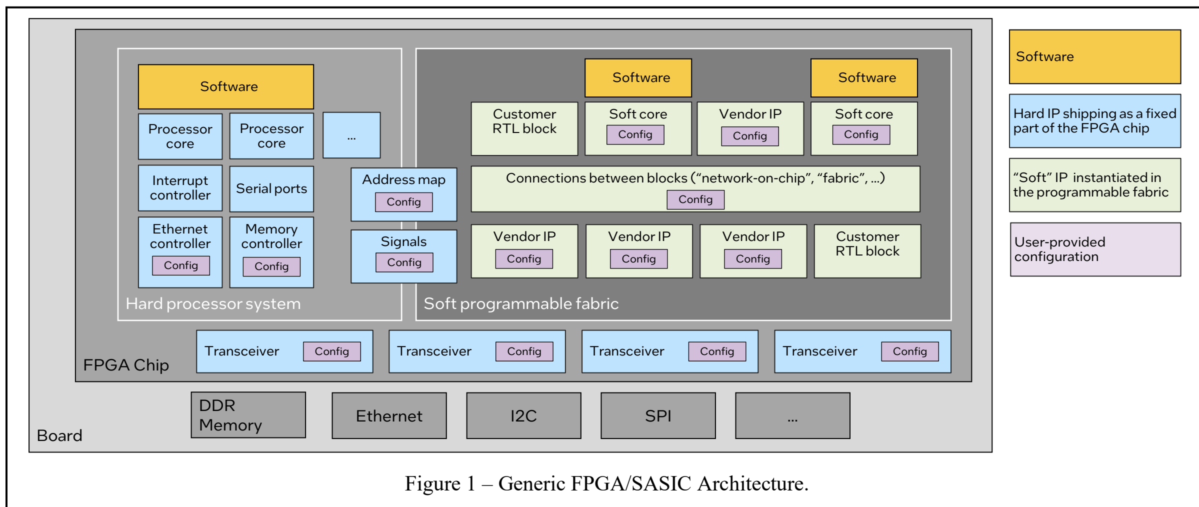


Figure 1 – Generic FPGA/SASIC Architecture.

As illustrated in Figure 1, modern field programmable gate array (FPGA) and structured ASIC (SASIC) [2] systems have become increasingly complex and commonly include a significant software component. The FPGA chip provides both fixed and programmable parts. The **fixed parts**, also known as hard IP, vary greatly between different chips, but

typically include transceivers, memory controllers, network interfaces, and even processor subsystems where the precise functionality and connections of the hard IP is configurable by the user using vendor supplied IP blocks. The **programmable fabric part** is what makes the FPGA into an FPGA, and lets the user instantiate both their own hardware designs, expressed in a register transfer language (RTL), and vendor-provided soft IP blocks. Software in these devices can be executing on processors provided in the hard IP of the FPGA, and on processor cores instantiated in the FPGA fabric.

The combination of hardware design and software design means that FPGA projects increasingly look like ASIC projects and makes the use of VPs compelling for FPGA/SASIC designs. However, creating VPs for programmable hardware has some unique challenges compared to building a VP for a standard ASIC, due to the configurable and flexible nature of FPGAs and SASICs. These challenges can be overcome by leveraging the information already available in the FPGA/SASIC design tool, as demonstrated in this paper.

## USING VIRTUAL PLATFORMS FOR FPGAS

In general, VPs provide a benefit to projects whenever hardware availability is an issue for software developers – either because the hardware is not yet ready, or because there are limits to the number of hardware units available to the development teams. In practice, very small FPGA teams, such as only one or two developers, may not use VPs or even RTL simulation. Such projects can get by using a few commercial hardware development boards, typically one for each developer. Design changes to either hardware or software are simply deployed and tested on the hardware, skipping simulation.

However, larger projects, from five developers and up, can typically benefit from decoupling hardware and software development using VPs. It becomes cost-prohibitive and impractical to provide all developers with development boards. Larger projects are also more likely to use custom system and board designs where the hardware is not immediately available, even if the FPGA chips are, and thus require the use of VPs to do pre-silicon (or shift-left) software development. Furthermore, applying modern software development techniques like agile, continuous integration, and DevOps assume that test and execution resources can be dynamically managed and used on-demand [3]– which is not compatible when relying on a fixed number of custom hardware setups.

To support software development and test, VPs have to be fast, which necessitates the use of loosely-timed (LT) transaction-level modeling (TLM) together with fast instruction-set simulators [1]. There are various alternatives, but none is as universally practical as the fast LT-TLM VP such as Simics simulator [4] or QEMU [5].

One alternative method to the cost of building an LT-TLM VP is to use *host-compiled simulation* to run software independently of hardware, such as the technique described in [6]. However, compared to using a VP, host-compiled simulation requires changing the source code and introduces an additional build target that needs to be maintained. Looking at the total cost, maintaining the host-compiled framework might be just as expensive as building a regular VP since the framework also requires building models of the hardware. Unlike an LT-TLM VP, the host-compiled simulation means the same source code and binary code cannot be used for testing both on a VP and on the real hardware.

Another alternative to using an LT-TLM VP is to use a VP-RTL co-simulation, such as [7], where the processor hard IP of the FPGA is provided as a VP and everything else is simulated in RTL. Such solutions are inherently slow as they are limited by the speed of the RTL simulation, which is typically a million times slower than TLM. Compiling RTL into SystemC/C++ code as done by tools like Verilator [8] provides some performance improvements, but still runs thousands of times slower than a TLM VP and provides limited debuggability given its one-way translation. Essentially, any solution that relies on the RTL or provides a model built to the same structure as the RTL, such as structural SystemC models, will be too slow to properly support software development at maximum velocity.

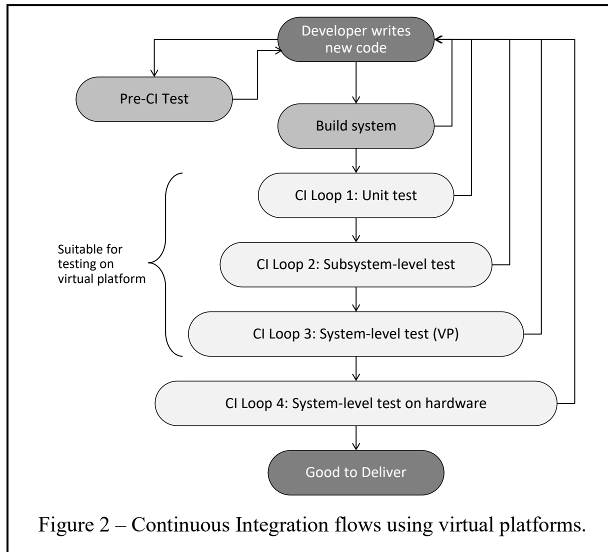


Figure 2 – Continuous Integration flows using virtual platforms.

Building a VP for just the hard processor IP of the FPGA helps to some extent. However, it still leaves open the question of how to model the portions of an FPGA design implemented in the programmable fabric and the rest of the programmable hard IP beyond the processor subsystem. Such limited models also tend to model the general behavior of the hard processor system, allowing more software to run than would run on the actual configuration of the hardware. As a result, negative testing is not possible, and the behavior might differ in the details from what would happen on actual hardware. This is antithetical to the foundational goal of a VP enabling simulation of the real hardware behavior. Configuring the VP to match the configuration of the hardware would have to be done manually by the user, as the vendor would not know the detailed setup of the user.

It should be noted that VPs do not obviate the need to test on hardware. Rather, they serve as an extension of the hardware farm. As shown in Figure 2, VPs are typically used for the first levels of continuous integration loops to provide fast feedback to developers. Once testing is successful on VP, tests move on to hardware. For this strategy to be efficient, the VP needs to be precise enough to enable both positive and negative tests – it is not sufficient for a VP to model a superset of the hardware functionality, as that will tend to hide errors. Critically, there should be no functional difference in the behavior of the software running on a VP versus the same software running on hardware.

## Challenges for Virtual Platforms of FPGA/SASIC Designs

*Fact: FPGA teams are smaller than ASIC teams.*

Compared to ASIC projects, FPGA-based projects have different economics. The design team for an ASIC project is typically much larger than for an FPGA project. Since we would expect the relative size of the VP modeling team to the design team to be about the same (in order to maintain the same level of return-on-investment, ROI), in absolute terms, the modeling team for an FPGA project should be smaller than for an ASIC. However, the modeling effort is of similar complexity. Something is needed to improve the productivity of the VP teams in FPGA projects, relative to the typical techniques used by ASIC VP teams.

*Problem: FPGA vendors cannot provide a VP for their chips. Customer needs to create it for their design.*

Providing a VP is different for an FPGA/SASIC vendor compared to an ASIC vendor. For an ASIC, a single model can be built by the vendor and used by multiple customers (since the ASIC design is known and fixed). On the other hand, for an FPGA/SASIC, each customer's design is by nature unique. The task of creating a VP for an FPGA/SASIC design is therefore an activity that must be done by the customer and not the FPGA vendor.

*Fact: Vendor-provided IP is highly parameterizable and can dramatically affect the software interface.*

Modern FPGA/SASIC designs almost exclusively use vendor-provided IP blocks for the hard IP of the FPGA and critical portions of their fabric design. The exact parameterization of these IPs is chosen by the customers and can change as the design evolves due to fitting the design in the target device, or as the requirements of the design change. These IPs, even the so-called hard IP, are configurable by the user. They are generally much more parameterizable than the IPs found in ASICs, and the parameterization can drastically change the software interface.

As an example, Figure 3 shows the Interval Timer FPGA IP [9] from the IP catalog of the Intel® Quartus® Platform Designer [10] software and its configuration options. The options affect both the functionality of the hardware and the register map visible to software. For example, if the counter size option is changed from 32 to 64, then register map changes as shown in Figure 4. In addition, options such as enabling start/stop control bits impact those bits in the register map. More complex IPs, like the F-Tile Ethernet Intel FPGA Hard IP [11], have an even greater parameterization space given the number of different Ethernet modes and IP core variations. A model of such an IP needs to cover all possible parameterizations and ensure that the model behaves as expected in both positive and negative tests.

Creating models for these vendor IPs can be extremely challenging based on their complexity and may require significant modeling expertise and internal implementation details to realize. Beyond the modeling skillset challenge, FPGA vendors do not always document their IPs with the sufficient level of detail needed for a customer or a 3rd party to create models of them. As a result, we assert that only vendors can reasonably create models for their IPs.

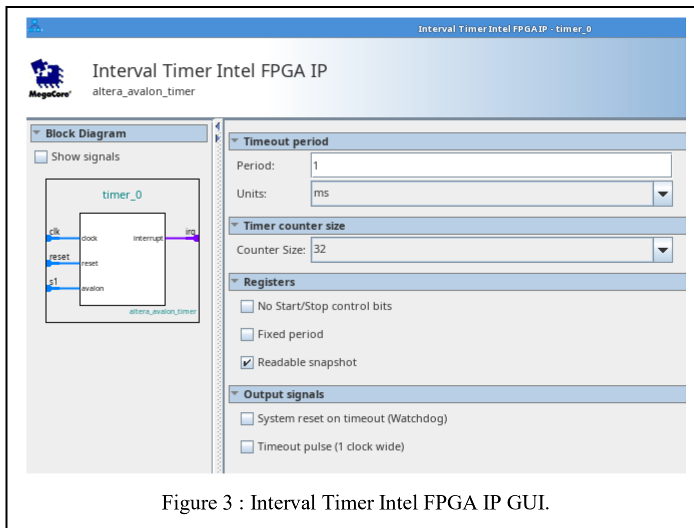


Figure 3 : Interval Timer Intel FPGA IP GUI.

*Problem: The parameters of the VP IPs need to be easily changeable and in sync with the hardware design.*

Given the relative ease of change and the lack of fabrication delay in an FPGA design, it is standard FPGA design practice to have much larger changes and relatively longer periods of design change compared to an ASIC design. The VP therefore needs to be quickly and easily changed to track the current design state. Matching the hardware design pretty much requires the VP to be generated from the FPGA design data – doing it manually is too slow and error-prone.

<table>
<tr><th></th><th colspan="2">Counter Size == 32</th><th colspan="2">Counter Size == 64</th></tr>
<tr><th>Offset</th><th>Reg Name</th><th>Description</th><th>Reg Name</th><th>Description</th></tr>
<tr><td>0</td><td>status</td><td>Run[1]|TO[0]</td><td>status</td><td>Run[1]|TO[0]</td></tr>
<tr><td>1</td><td>control</td><td>Stop[3]|Start[2]|Cont[1]|ITO[0]</td><td>control</td><td>Stop[3]|Start[2]|Cont[1]|ITO[0]</td></tr>
<tr><td>2</td><td>periodl</td><td>Timeout Period [15:0]</td><td>period_0</td><td>Timeout Period [15:0]</td></tr>
<tr><td>3</td><td>periodh</td><td>Timeout Period [31:16]</td><td>period_1</td><td>Timeout Period [31:16]</td></tr>
<tr><td>4</td><td>snapl</td><td>Counter Snapshot [15:0]</td><td>period_2</td><td>Timeout Period [47:32]</td></tr>
<tr><td>5</td><td>snaph</td><td>Counter Snapshot [31:16]</td><td>period_3</td><td>Timeout Period [64:48]</td></tr>
<tr><td></td><td></td><td></td><td>snap_0</td><td>Counter Snapshot [15:0]</td></tr>
<tr><td></td><td></td><td></td><td>snap_1</td><td>Counter Snapshot [31:16]</td></tr>
<tr><td></td><td></td><td></td><td>snap_2</td><td>Counter Snapshot [47:32]</td></tr>
<tr><td></td><td></td><td></td><td>snap_3</td><td>Counter Snapshot [64:48]</td></tr>
</table>

Figure 4 : Interval timer register map when used in 32-bit and 64-bit mode.

*Problem: Customer adds their own IP and wants to include them in the model.*

In addition to the vendor IP components in an FPGA/SASIC design, customers also provide their own hardware block designs, in order to implement custom functions or acceleration. If the customer-provided blocks are controlled by software, the customer blocks must be included in the VP to ensure that the execution of the software can be accurately simulated. Thus, the VP design flow must ensure that customer hardware is modeled in the VP and kept up-to-date with design changes.

### Leveraging the FPGA System Assembly Tool

FPGA system assembly tools, such as Intel Quartus® Platform Designer [10], provide the ability to quickly describe a complete system including both vendor IPs and customer blocks. The assembly tool captures all the information

needed to fully describe the hardware setup and software interface of the complete design, including the detailed configuration of all parts of the system including the memory map, and the connectivity. Today, the assembly tool is used to generate the RTL output used by the vendor compiler tool chain to create the hardware bitstream.

In our opinion, the answer to creating VPs for FPGA/SASIC designs is to generate the VP from the existing FPGA design creation and experimentation workflow using enhancements to existing assembly tools like Platform Designer. As illustrated in Figure 5, a designer using the existing assembly tool to select, parameterize, instantiate, and connect vendor-provided IPs and their own blocks, would get a VP configuration produced based on the design expressed in the tool as an additional output.

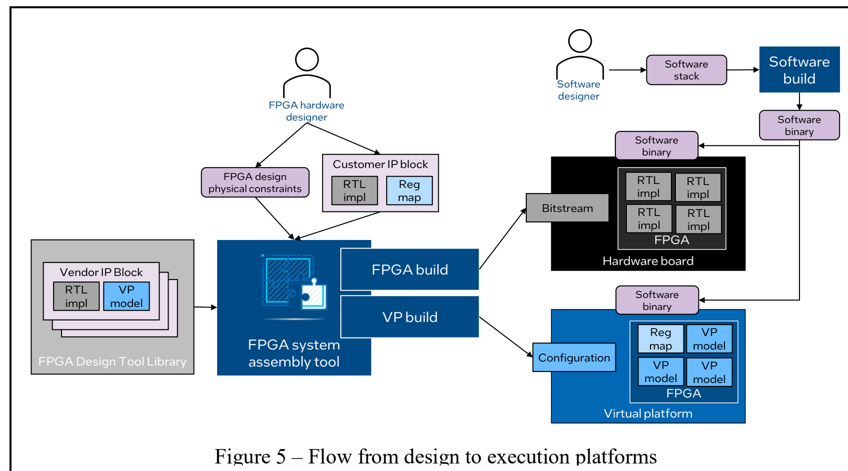


Figure 5 – Flow from design to execution platforms

Automatic VP generation is practical based on two facts; 1) most designs rely extensively on vendor IP blocks for the shell of their design, and 2) designers use this dedicated system assembly tool to build the system. Inside the existing assembly tool is a library of IP models provided by the vendor that can cover most of the shell for a design. The library can be augmented to provide both RTL and VP models for the vendor IP blocks. The VP models themselves expose the same parameters as the RTL and these parameters modify the behavior of the model. When the assembly tool generates the output for an IP instance, it writes a wrapper around the VP IP that specifies all the parameters and effectively results in a fixed function VP IP model. This generation is equivalent to the existing process that occurs for generating the RTL.

For customer IP blocks, the assembly tool knows the IP's memory-mapped addresses and other connections to the rest of the system. This is sufficient to automatically generate stub models for each of the customer's IP which can also be manually extended into fully functional VP models. If the customer supplies a register map for their blocks, it can also be included in the generated stub model, reducing the work needed to build a functional model.

Since the assembly tool knows the complete system memory map and connectivity of the system, plus the parameterized vendor IP models and stub models for customer IP, a complete VP can be generated that is faithful to the FPGA design captured in the assembly tool without any custom modeling on the part of the designer. The generated system setup follows the structure of the generated RTL, with different types of information described at different levels of the hierarchy. The top level declares the set of sub-components and how they are connected, while each sub-component describes a particular instance of an IP block with all its parameters.

Going beyond the FPGA chip, the design captured in the assembly tool also includes information such as external memory sizes and IO pin assignments. In the hardware, they are needed to correctly interface the FPGA to the rest of the board. This information can be leveraged to generate a board-level VP configuration that includes external components and the VP connections required to connect to models of external processor chips and other ASICs.

## EXAMPLE: CREATING A VIRTUAL PLATFORM FOR AN FPGA DESIGN

We have implemented a prototype of this concept, using the Intel Quartus® Prime Pro FPGA design tool generating a VP for the Intel Simics simulator [4]. The prototype generates Python code using the Simics simulator *component*

*framework* to create a simulation configuration. The components can be nested hierarchically as well as connected laterally, to provide a simulation setup that matches the hardware design (at the TLM level of abstraction). The components are not active during a simulation, and simply create and configure the set of simulation objects to be used at runtime. The simulation objects are instances of VP models such as devices, memory maps, processor cores, etc. The simulation objects are configured at run time or at build time, depending on the nature of the configuration. The assembly tool maps IP configuration options to Simics simulator configuration attribute values or device model configurations using during the VP build step.

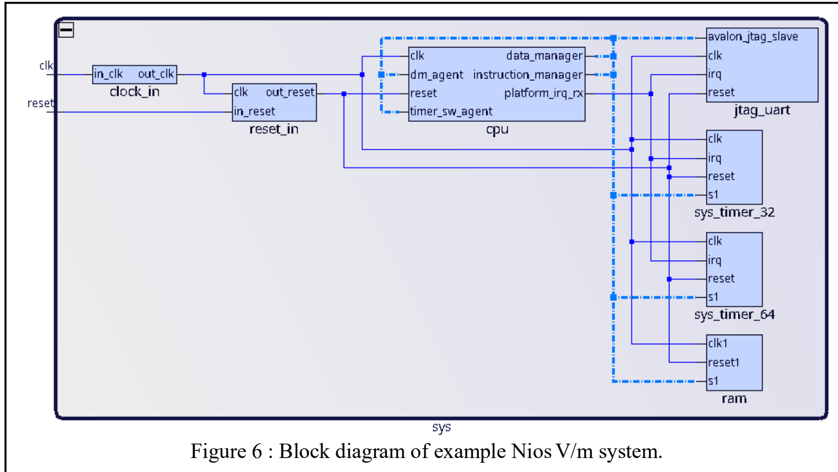


Figure 6 : Block diagram of example Nios V/m system.

Using the Platform Designer assembly tool (part of the Intel Quartus Prime Pro software), we created a simple RISC-V processor system using the existing RTL design flow. The source code for this example can be found at [12]. The resulting system as seen in Platform Designer is shown in Figure 6 and includes a Nios® V/m RISC-V processor core, an on-chip memory, a UART, and two interval timers. The IP blocks are instantiated and parameterized in the tool along with the definition of the connections between the IPs and the system memory map. The generated RTL hierarchy as seen during the Intel Quartus FPGA compilation flow is shown in Figure 7. The RTL hierarchy is derived from, and matches, the design as expressed in Platform Designer. The RTL for each of the variations of the IPs in the system are parameterized as defined in the design captured in Platform Designer.

Using this complete system, we are able to automatically generate parameterized virtual platform models and then a complete virtual platform system definition. Figure 8 shows the system running in the Simics simulator. The hierarchy

| Instance | Entity |
|---|---|
| Arria 10: 10AS066N3F40E2SG | |
| top | |
| auto_fab_0 | alt_sld_fab_0 |
| issp_inst | altsource_probe |
| u_sys_inst | sys |
| clock_in | sys_clock_in |
| cpu | cpu |
| intel_niosv_m_inst | cpu_intel_niosv_m_2230_23rxb3q |
| irq_mapper | sys_altera_irq_mapper_2000_m3l6mky |
| jtag_uart | jtag_uart |
| altera_avalon_jtag_uart_inst | jtag_uart_altera_avalon_jtag_uart_1920_vygddhy |
| mm_interconnect_0 | sys_altera_mm_interconnect_1920_qtui2vi |
| ram | ram |
| altera_avalon_onchip_memor... | ram_altera_avalon_onchip_memory2_1936_clyavqi |
| the_altsyncram | altsyncram |
| auto_generated | altsyncram_nrl1 |
| reset_in | sys_reset_in |
| rst_controller | altera_reset_controller |
| alt_rst_req_sync_uq1 | altera_reset_synchronizer |
| alt_rst_sync_uq1 | altera_reset_synchronizer |
| sys_timer_32 | sys_timer_32 |
| sys_timer_32 | sys_timer_32_altera_avalon_timer_1930_gyskr5a |
| sys_timer_64 | sys_timer_64 |
| sys_timer_64 | sys_timer_64_altera_avalon_timer_1930_ozxmi3q |

Figure 7 : RTL hierarchy of the example design as seen in the Intel Quartus software

```
simics> list-objects -tree -hide-port-objects
└ vp0 ┐
      ├ cell ┐
      │      └ ps
      ├ cell_context
      ├ cell_rec0
      └ u_sys_inst ┐
                   ├ cpu ┐
                   │     ├ hart ┐
                   │     │      └ vtime ┐
                   │     │              ├ cycles
                   │     │              └ ps
                   │     ├ phys_mem
                   │     └ timer_module
                   ├ jtag_uart ┐
                   │           ├ jtag_uart
                   │           ├ stty
                   │           └ stty_console ┐
                   │                          ├ con ┐
                   │                          │     ├ frontend
                   │                          │     ├ tcp
                   │                          │     └ unix_socket
                   │                          └ serial
                   ├ ram ┐
                   │     └ ram ┐
                   │           └ image
                   ├ sys_timer_32 ┐
                   │              └ sys_timer_32
                   └ sys_timer_64 ┐
                                  └ sys_timer_64
```

Figure 8 : Hierarchy of the example design as seen in Simics simulator – matching the RTL hierarchical structure

of the simulated system is generated to match the RTL hierarchy. The hierarchy is likely deeper than what you find in a manually created VP, but the runtime cost of a few extra objects is negligible. The benefit of following the RTL hierarchy is that it is possible to maintain a one-to-one mapping between simulation components and the IP blocks of the Platform Designer system, including the structure of the generated VP source code.

Some snippets of the generated Simics simulator system component code are shown in Figure 9. The generator creates constants for the properties of the system, then writes out code for each instance of an IP model instantiated in the system. Calls to **add_pre_obj()** add runtime simulation objects to the setup, while **add_component()** adds subsystems. Some subsystem parameters are expressed as configuration attributes on the components, while others are encoded as constants in the component code. This aligns with the configurability of the hardware system.

When adding objects, some configuration aspects are provided as attribute values, while others are compiled into the code of the models themselves. In particular, the connections between objects are expressed as attributes. The address configuration of the FPGA fabric is coded as mappings in memory space objects (see the assignments to **phys_mem** in Figure 9). Some object parameters do not make sense to hardcode into the compiled models, such as the frequency, core ID, and reset vector of a processor (see the assignments to **cpu_core** attributes in Figure 9).

```
class fpgadesign_sys_comp(StandardConnectorComponent):
    _class_desc = "sys component for sys.qsys"
    # Constants from ‘sys.qsys’
    CLOCK_IN_FREQ_MHZ = 100.0

    RAM_BASE_ADDRESS = 0x00000000
    RAM_ADDRESS_SPAN = 0x00049400

    # [...]

    def add_objects(self):
        # Instantiate 'cpu' block from its own component
        self.add_component(
            "cpu", "fpgadesign_cpu_comp",
            # Pass down the processor core frequency to the sub-component
            [["freq_mhz", self.CLOCK_IN_FREQ_MHZ]]
        )
        cpu_inst_core = self.get_slot("cpu.hart")
        # Retrieve a reference to the processor memory map
        cpu_inst_phys_mem = self.get_slot("cpu.phys_mem")

        # Instantiate 'ram' block from its component
        self.add_component("ram", "fpgadesign_ram_comp", [])
        ram_inst = self.get_slot("ram.ram")
        # Map the ram into the processor’s memory map
        # By appending new mappings to the map created in the right-hand code
        cpu_inst_phys_mem.map.append(
            [
                self.RAM_BASE_ADDRESS, ram_inst,
                0, 0, self.RAM_ADDRESS_SPAN,
            ]
        )
        # [...]
```

```
class fpgadesign_cpu_comp(StandardConnectorComponent):
    _class_desc = "Nios Vm CPU from cpu.ip (intel_niosv_m)"
    # Constants from ‘cpu.ip’
    TIMER_SW_AGENT_BASE_ADDRESS = 0x00090000
    TIMER_SW_AGENT_ADDRESS_SPAN = 0x00000040
    CPU_RESET_VECTOR = 0x00000000
    CPU_PC = 0x00000000

    # [...]

    def add_objects(self):
        timer = self.add_pre_obj(
            "timer_module", "nios_v_timer",
            freq_mhz=self.timebase_freq_mhz.val
        )
        phys_mem = self.add_pre_obj("phys_mem", "memory-space")
        cpu_core = self.add_pre_obj(
            "hart",
            "riscv-nios-v-m",
            # Configuring the processor frequency from dynamic parameter
            freq_mhz=self.freq_mhz.val,
            # Connections to other objects
            physical_memory = phys_mem,
            clint           = timer,
        )
        timer.hart = cpu_core
        # CPU core parameters fixed by the IP component parameters
        cpu_core.mhartid = 0
        cpu_core.reset_vector = self.CPU_RESET_VECTOR
        cpu_core.pc = self.CPU_RESET_VECTOR
        # Initial memory map contents
        phys_mem.map = [
            [
                self.TIMER_SW_AGENT_BASE_ADDRESS,[timer, "regs"],
                0, 0, self.TIMER_SW_AGENT_ADDRESS_SPAN,
            ]
        ]
        # [...]
```

Figure 9 : Excerpts from the Simics simulator component code for the example

On the other hand, the choice between 32-bit and 64-bit modes for the timer in Figure 3 makes perfect sense to handle at model compile time since it has a major impact on the nature of the model – just like it has for the RTL. Figure 10 shows an example of compile-time configuration for the **timer_32** VP model (written in the Device Modeling Language, DML [13]). The parameters shown on the left are generated from the design captured in the assembly tool and reflect the parameterization of the specific IP variant. The code on the right shows generic configurable code where behavior and the set of registers vary based on the parameters. The result is an automatically generated VP model that matches the specific parametrization of the example system and the FPGA design.

## CONCLUSIONS

This paper has presented the current challenges in creating VPs for FPGA/SASIC customer designs; the end design is created by the customer, includes a large amount of vendor-provided configurable IP, and the design itself is expected to change. While methods currently exist to perform hardware-software co-simulation, they do not deliver on the promise of a fast TLM-LT VP. To allow the quick creation of a fast TLM-LT VP, we extend an existing vendor tool

chain to automatically generate VP configurations that match the hardware configuration already described in the tool. The solution enables FPGA/SASIC projects to create and use VPs that match their hardware designs using the existing FPGA design tool flow, without massive costs for manual VP coding.

```
dml 1.4;

device fpgadesign_timer_32;

param desc = "Interval Timer Intel FPGA IP ‘timer_32’";

// [...]

// Constants for model parameters from 'timer_32.ip'
// Controls the behavior of the generic code on the right
param counter_reload_val = 128;
param counter_size = 32;
param no_start_stop_bits = false;
param fixed_period = false;
param readable_snapshot = true;
param system_reset_on_timeout = false;
param interrupts_enabled = true;

// [...]
```

```
// Generic model code that has different behavior based on parameters
bank regs {
  param register_size = 2;

  register status @ 0x00 "Status regtest" {
    field reseved @ [15:2] is (unimpl);
    field run @ [1] is (read_only);
    field to @ [0] is (read, write);
  }
  register control @ 0x02 "Control register" {
    field reseved @ [15:4] is (unimpl);
    #if (no_start_stop_bits) {
      field stop @ [3] is (unimpl);
      #if (system_reset_on_timeout) {
        field start @ [2] is (read, write);
      } #else {
        field start @ [2] is (unimpl);
      }
    } #else {
      field stop @ [3] is (read, write);
      field start @ [2] is (read, write);
    }
    field cont @ [1] is (read, write);
    field ito @ [0] is (read, write);

  }

  #if (counter_size == 32) {
    register period[i < counter_size/8] @ ((0x04 << 2) + i*2) is (read, write) "Timeout Period";

    #if (readable_snapshot) {
      register snap[i < counter_size/8] @ ((0x08 << 2) + i*2) is (read, write) "Counter Snapshot";
    } #else {
      register snap[i < counter_size/8] @ ((0x08 << 2) + i*2) is (unimpl) "Counter Snapshot";
    }
  } #else {
    register period[i < counter_size/8] @ ((0x04 << 2) + i*2) is (read, write) "Timeout Period";

    #if (readable_snapshot) {
      register snap[i < counter_size/8] @ ((0x0C << 2) + i*2) is (read, write) "Counter Snapshot";
    } #else {
      register snap[i < counter_size/8] @ ((0x0C << 2) + i*2) is (unimpl) "Counter Snapshot";
    }
  }
}
```

Figure 10 : Excerpts from the configurable timer model source code for the timer_32 variant.

## REFERENCES

[1] D. Aarno and J. Engblom, Software and system development using virtual platforms: Full-system simulation with Wind River Simics, Waltham, MA : Elsevier, Morgan-Kaufmann, 2015.

[2] H. K. Phoon, M. Yap and C. K. Chai, "A Highly Compatible Architecture Design for Optimum FPGA to Structured-ASIC Migration," 2006 IEEE International Conference on Semiconductor Electronics, 2006, pp. 506-510.

[3] J. Engblom. “Continuous Integration for Embedded Systems using Simulation”, *Embedded World 2015 Congress*, Germany, 2015.

[4] The Intel® Simics® Simulator Public Release, https://developer.intel.com/simics-simulator

[5] F. Bellard, "QEMU a Fast and Portable Dynamic Translator", Proceedings of USENIX Annual Technical Conference, pp. 41-46, June 2005.

[6] Intel Corp., “FPGA Soft Processor Unit Test Example Design.” https://github.com/intel/fpga-soft-processor-unit-test, August 2022.

[7] Xilinx Inc., *Versal ACAP Design Guide*, UG1273, v2022.1, 2022.

[8] Verilator homepage, https://veripool.org/verilator/

[9] Intel Corp., Embedded Peripherals IP User Guide, UG 01085, v2022.06.21, June 2022.

[10] Intel Corp., Intel Quartus Prime Pro Edition User Guide: Platform Designer, UG 683609, v2022.06.20, June 2022.

[11] Intel Corp., F-Tile Ethernet Intel FPGA Hard IP User Guide, Section 8.1, UG 20313, v2022.06.20, June 2022.

[12] Intel Corp., “Simics Simulator for Intel FPGA Example Designs.” https://github.com/intel/simics-fpga-examples, unpublished.

[13] Device Modeling Language, https://github.com/intel/device-modeling-language