Using Control Logic Drivers for Automated Generation of System-level Portable Models

Petr Bardonek
Brno University of Technology
Brno, Czech Republic
ibardonek@fit.vutbr.cz

Marcela Zachariášová
Brno University of Technology
Brno, Czech Republic
zachariasova@fit.vutbr.cz

***Abstract*—Portable Test and Stimulus Standard is a new Accellera standard for an abstract definition of the verification intent that can be used for stimuli generation for different types of verification environments and at different levels of design hierarchy. Despite the idea behind the standard is clear and well received by the general public, there is still a lot of work to make the magic happen inside of the interpretation tools. In this paper, we focus on vertical reuse of portable models which is basically about adapting portable models for block-level designs to portable models defined at the subsystem or system-level. This adaptation is usually based on manually defined (sub)system-level control restrictions and resources sharing restrictions. Our goal is to define algorithms which do transformations of portable models so as the restrictions are automatically added or suggestions for the user are made. In our first experiments, we focus on building control restrictions based on the control logic drivers extracted from the subsystem-level design. We present our first results on the floating-point unit and RISC-V processor-based subsystem as they fit to represents the vertical reuse in a clear manner.**

***Index Terms*—PSS, portable stimuli, portable models, Universal Verification Methodology (UVM)**

I. Introduction

From the work of the first adopters of Portable Test and Stimulus Standard (PSS) [1], it can be seen that PSS is mostly used for defining an abstract verification intent that can be used for generating stimuli (tests) to different platforms such as logical simulators, FPGAs, or emulators (see *Platform reuse* definition in Section II). For example, authors in [3] and [4] explain how *Platform reuse* is executed via the test realization code encapsulated into platform-targeted *exec blocks*.

In general, the quality of the generated tests is better because the PSS interpretation tools (Questa InFact from Mentor, Perspec from Cadence, or Trek from Breker) allow to see every verification intent captured by a *portable model* (PM) as a graph and this helps engineers to gradually improve their models by adding constraints to define only legal transitions between states or coverage goals that should be reached by the generated stimuli. All of this helps to have the state space well defined and systematically explored by the tools, which significantly reduces redundancy in the generated data.

Redundancy is a frequent problem in verification environments which utilize a generator for creating stimuli (e.g. UVM). The reason is that the generator usually doesn't have a feedback about functionality that was already covered and therefore, produces the same stimuli all over again. That is why it is understandable that authors who are using PSS claim better verification performance, reaching coverage closure faster, or significant time savings in tests implementation [3], [7]. Utilization of PSS and Cadence's tool Perspec for verification of WIFI component is described in [6]. Authors highlight PSS advantages based on great results but admit that without help of a Perspec expert, the definition of PM would take much more time and would not have such a good quality. That is why there are several papers that aim at improving the engineer's experience with PSS or helping them to create PMs effectively. For example, authors in [8] defined the PSS framework in which PSS is automatically translated to *ruby_tests* and *ruby_uvm* which contain Ruby classes enveloping C-functions exported from UVM. The paper [9] exploits an option of incorporating PSS into IP-XACT, which is used for generating IP blocks, subsystems, or even SoCs. The idea is to use the same flow but in addition, extracting the verification intent from PSS for generating tests.

The goal of our work is to help engineers even more by defining algorithms which will do semi-automated transformations of PMs for different reuse scenarios. We focus mainly on *Vertical reuse* of PMs (see definition of *Vertical reuse* in Section II), because from our understanding, it is the most complex and challenging one. This paper covers:

1) Vertical reuse definition and selection of representative designs (floating point unit (FPU) and RISC-V processor subsystem) that clearly show vertical reuse in PMs.
2) Defining PMs for these designs.
3) Analyzing how the logic drivers of control inputs of FPU can be used for defining control restrictions in the PM at the processor subsystem-level.
4) Defining the outcomes of the analysis which can help to automate creation of such control restrictions.

The paper consists of six sections. Section II describes PSS and PM in more details, mainly how the process of defining the abstract verification content looks like and how stimuli can be generated by dedicated tools. It also explains the reuse options while most of the attention is paid to vertical reuse. This section also contains a related work connected to the vertical reuse. Section III outlines FPU and RISC-V processor subsystem which were selected as representative designs for vertical reuse. We have implemented PM for FPU. Section IV

978-1-7281-9938-2/20/$31.00 ©2020 IEEE

shows a complete analysis of the control flow assigned to control inputs of FPU, while Section V describes an idea of algorithmization of the findings. Section VI concludes the paper and outlines the ideas of our future work.

II. Portable Models and Reuse

The description of the verification intent (in the form of the so-called *rules*) results in a PM representing all possible scenarios for the verified system that will be explored during verification. Nowadays, the creation of such description is a manual job done by verification engineers.

Based on the rules, the PSS interpretation tool constructs a graph as a visual representation of the described model to give the verification engineer an easier way to debug. The main abstraction mechanism in PSS is called *action*, which represents a unit of behavior (leaf nodes of the graph). Actions can be directly connected to the operations/functions of the *design under test* (DUT) or to operations/functions of the verification environment. This is done inside of the structures called *exec blocks* (these actions are then called *atomic actions*). The content of exec block depends on both the platform and the design hierarchy level where verification takes place. Therefore, this content is usually modified when porting is executed. Moreover, actions can be compounded of more actions, while defining ordering and scheduling and thus specifying more complex scenarios. Atomic, as well as compound actions, are instantiated inside of *components*, which are the group elements intended for reuse and composition. Next to the actions, there are passive entities defined in PSS, such as resources, states, and data-flow items, which are collectively called objects. They constrain scenarios that can be actually generated (we call them *resources restrictions*).

PM represents a set of scenarios that can be generated and PSS tool automatically enumerates the minimum set of runs needed to cover the whole state space defined by PM. Because this state space still can be quite big, we can further narrow it by defining coverage and constraints (we call them *control restrictions*). The focus can be on specific values or preferred ranges. The aim of the tool will be reaching 100% of the defined coverage goals.

As for the portability, it is possible to categorize PSS applications according to what type of reuse is most central to the application. In [4], three reuse options were identified:

- *Platform reuse* – Reuse of verification intent on different platforms (UVM/SystemVerilog verification environment, FPGA, emulator).
- *Vertical reuse* – Reuse of verification intent from block- to subsystem- or to system-level verification.
- *Horizontal reuse* – Reuse of verification intent across derivatives of the same design or across designs with significant similarities.

In this paper, we focus on vertical reuse. Several different languages and techniques are used to create verification stimuli depending on whether a block, subsystem, or system is being verified. When verifying RTL blocks and small subsystems, SystemVerilog or 'e' sequence items, sequences and tests can be used, usually inside of UVM-based verification environments. Occasionally, SystemC, Verilog or VHDL test-benches are applied. However, at the SoC-level, the main source of stimuli for different parts of the system is embedded software running on one or more processors. What does it mean from the PM point of view?

1) We need PMs for blocks, as well as a PM for the whole system, while the first ones would be ideally a subset of the second one and to model system scenarios, only control and resources restrictions are added.
2) We need to achieve that stimulus representation generated from PMs at the block-level can be different than the one generated from PM at the SoC-level. This can be accomplished by defining various exec blocks inside of actions that reflect the target design hierarchy level.

From the related work connected to vertical reuse it is clear that it is important to involve some up-front planning [4]. In [5] it is stated that while moving from block to subsystem or system-level, there may be several details different from the block-level environment, such as memory addresses, device IDs, different constraints on certain operations, sharing of resources. Overall, there are additional layered constraints in PMs. Paper [11] describes a complete cycle of an interconnect bus verification - from IP to SoC-level, using PSS. One PM was reused on all levels but *exec blocks* must be rewritten to reflect specific requirements on every level.

III. Portable Model for FPU

We have selected RISC-V processor subsystem as a suitable candidate for vertical reuse demonstration. In particular, it is represented by an open-source processor implementation called RI5CY, which was developed as a part of the PULP platform [2].

In the verification of RISC-V processors, a hierarchical approach at the block-level, at the processor top-level and at the SoC-level is usually applied. At the block-level, we verify that the functionality of the independent blocks from which the processor consists of, is fine. The examples of such blocks are: pipeline control, FPU, memory management unit, register array, or interrupt controller. Subsequently, the processor as a complex unit is verified, and in this case, we focus on the cooperation of all blocks together. In the final phase, integration of the processor into SoC is checked, mainly its connection to the surrounding IP blocks, memory subsystems or other processors.

At this point we will shortly describe how our PM for FPU look like as it forms the base for our experimental work.

A. FPU Portable Model

FPU in the PULP platform has a lot of possible configurations but we have selected only one of them to target vertical reuse. Nevertheless, for the future experiments there is a nice variety of configurations representing horizontal reuse.

FPU has some standard inputs defining floating point operations and operands. From the PSS point of view, all operations have the same structure - one operation working with required

number of operands. Therefore, it can be represented by one base model which is further constrained or extended.

We started by defining two atomic actions, one for sending data to FPU (*fpu_send_data*) and one for checking the control signal which determines that FPU is ready to start a new computation (*fpu_wait_rdy*). The later one is enclosed in the cycle, because the model must wait for a component to be ready and just after that send data. Without waiting, we would throw away a lot of stimuli we wanted to use for verification as they would never be processed by the component. We encapsulated these two actions to a compound action called *fpu_wait_rdy*. The last step was to create a separate atomic action for generating data for these inputs. Furthermore, We created a compound action called *fpu_wait_done*. It is similar by structure to *fpu_wait_rdy*, but waits on the output signal of FPU determining the validity of the computed result. Both compound actions were enclosed into another one called *fpu_operation* to have the whole basic behaviour of FPU grouped together.

With the basic computational model of the verification intent ready, we had to consider the remaining inputs of FPU - reset and flush. We created a single atomic action for generating both of these inputs, because setting them to an active value results in same behaviour of FPU. We extended the already created atomic action *fpu_send_data* with these two inputs. We created a model for the situation when flush or reset is set while generating the duration of their active value. This behaviour is enclosed to the compound action called *flush_reset_rnd*.

As for the overall data flow of PM, first we generate all the inputs. If flush or reset is active we go to the branch containing the action *flush_reset_rnd*, otherwise we go to the branch containing the action *fpu_operation*. Coverage and constraints on generated data are defined only for the *fpu_operation* branch, because we do not need to specify values if flush/reset branch is selected. The base model is demonstrated in Figure 1.

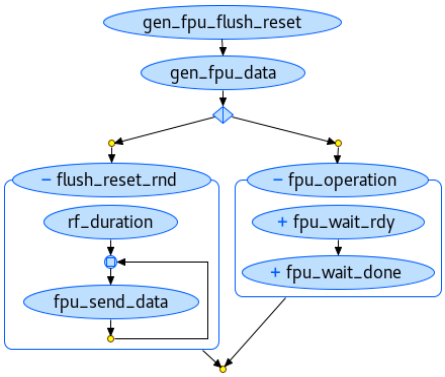

Fig. 1. The FPU base model.

IV. Analysis of Input Drivers for Control Restrictions Definition

In this section, we explore options how to semi-automatically create a PM for a subsystem (RI5CY) when we already have a PM of its sub-block available (FPU). First, we need to identify blocks in the subsystem and determine how they are connected together. We can do that by analyzing RTL and tracking sources for every input and tracking destinations for every output. This provides an insight, how the behaviour of the block (with its PM implemented like FPU), is potentially influenced by its environment (surrounding blocks). For this purpose, we decided to isolate the input control signals of FPU and analyze drivers of these control signals. Based on that it is possible to estimate control restrictions in the processor subsystem PM.

A. Control Signals Isolation

Our assumption is that control connections and also restrictions between blocks are usually based on control input port signals of those blocks. Therefore, we divided the port signals of FPU into four categories:

- *control inputs*: in_valid_i, out_ready_i, flush_i, rst_ni
- *control outputs*: in_ready_o, out_valid_o, busy_o
- *data inputs*: operands_i, rnd_mode_i, op_i, op_mod_i, src_fmt_i, dst_fmt_i, int_fmt_i, vectorial_op_i, tag_i
- *data outputs*: result_o, status_o, tag_o

B. Control Flow

At this point, it is important to define the control flow assigned to these control inputs. For this purpose, most of the simulation tools can help because they build a comprehensive model of logic drivers. In other words, for every single control input, it is possible to track assignments throughout the hierarchy of blocks and see, which signals can influence the behavior of a particular block.

When considering FPU, we have analyzed all logic drivers for all control inputs: the reset signal, the input valid signal and the output ready signal. Based on the analysis, the flush input is constant-driven to the 0 value, which means that FPU never executes flush (interrupt and throw away) of already started operations. Almost the same applies to the output ready signal, because it is constant-driven to the 1 value, which means that the receiver of FPU outputs must be always ready. The reset signal is globally driven for the whole processor subsystem thus there is not much to analyze either, it is a fairly straightforward implication. The input valid signal, which determines the validity of inputs, is, on the other hand, much more interesting. When is the RI5CY core configured in a way it includes FPU, also a component called APU dispatcher is connected which has control over the FPU instead of ALU. If we look inside of the APU dispatcher, we will find out that the setup of the input valid signal of FPU depends on a combination of six signals. They are part of the logic inside of APU which is responsible for solving dependencies between requests. The solution comprises storing a request when the previous one is getting processed, stalling the processing if requests cannot be stored anymore or if there is a type conflict caused by different latency of requests. The most important thing to note here is that the APU logic directly depends on two signals coming from another component, the decoder. The

decoder needs to set an enable signal to activate the APU dispatcher and the instruction latency. See Figure 2.

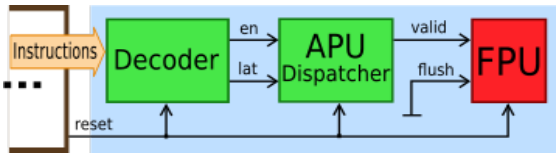

Fig. 2. Logic drivers of the FPU input valid control signal.

Based on the overall analysis, it is possible to recognize the following control restrictions. It is clear that control inputs of FPU are directly influenced by the control outputs of the APU dispatcher block (a block responsible, in general, for control over special units, shared or integrated, instead of ALU) and the decoder block (responsible for the decoding stage of the pipeline). Furthermore, scheduling of these control points, modelled by actions, can be deduced from the ordering of drivers themselves, where every step is constrained by the previous one:

- Setting up the right instruction.
- Decoder setting up the enable signal and instruction latency.
- APU dispatcher setting up the FPU input validity signal.

V. Algorithmization

Based on the analysis in Section IV, we have defined the following ideas for automation:

Automated analysis of RTL implementation of the subsystem/system. This analysis can be built on existing tools (simulation tools) completely, or based on post-processing of their logs by a series of scripts. This will depend on the outputs of the tools while taking into account the needs of the next processing. The output from this step should provide identification of the blocks and connections between them (dependencies). The dependency analysis will result into the dependency model between identified components. The dependency model can be divided into two separate models, one focusing on the control logic drivers, the second one on the data paths. This step can help a user significantly as it gives a picture of the subsystem/system that is being verified.

Automated generation of empty PMs for blocks. Based on the previous step, empty PMs for all identified blocks will be generated. Moreover, not only blocks as separate units but also common data flow objects can be generated based on the dependency model. This step allows an easy interconnection between generated PMs and provides an option of pairing existing PMs with the generated ones. It would require assistance from the user who can also replace/adjust the data flow objects. The later one could be semi-automated resulting in the creation of a connection layer between user-specified and generated data flow objects.

Automated creation of subsystem/system-level PM from existing block-level PMs. Using the dependency model, components with compound actions connecting existing PMs into one bigger model of subsystem/system is created. This step will use the dependency model focused on control logic drivers for generation of control restrictions for scheduling compound actions and connections inside of the created components.

VI. Conclusion

In this paper, we presented our first ideas of utilizing control logic drivers for semi-automated creation of subsystem/system-level PMs from existing block-level PMs. The purpose of this work is to help verification engineers with vertical reuse of PMs, as it is recognized as a challenging task when applying PSS and PSS-dedicated tools into the process of functional verification. In our future work, we plan to implement an example of the algorithm proposed in Section V and refine it based on further experiments with different blocks. Of course, there are still open questions about how other control aspects can be automatically added to PMs. For example, we plan to explore coverage control or power management control based on UPF models.

Acknowledgment

This work was supported by Brno University of Technology under grant number FIT-S-20-6309.

References

[1] Accellera Systems Initiative, PSS 1.0a Language Reference Manual, https://www.accellera.org/downloads/standards/portable-stimulus, 2020.
[2] Integrated Systems Laboratory of ETH Zürich and Energy-efficient Embedded Systems group of the University of Bologna, "PULP platform", https://pulp-platform.org/index.html, 2020.
[3] S. Rosenberg, "The Powerful Synergy Between UVM and PSS", Design and Verification Conference Europe 2019 (DVCon Europe 2019), 2019.
[4] M. Ballance, "Designing a PSS Reuse Strategy", Design and Verification Conference Europe 2019 (DVCon Europe 2019), 2019.
[5] T. Fitzpatrick, M. Ballance, "Results Checking Strategies with the Accellera Portable Test Stimulus Standard", Design and Verification Conference Europe 2019 (DVCon Europe 2019), 2019.
[6] E. Shneydor, S. Salnikov, S.L. Kosovizer, S. Greenberg, "Portable Stimuli Over UVM, Using Portable Stimuli in HW Verification Flow", Design and Verification Conference Europe 2019 (DVCon Europe 2019), 2019.
[7] G. Bhatnagar, D. Brownell, "Portable Stimulus vs Formal vs UVM: A Comparative Analysis of Verification Methodologies Throughout the Life of an IP Block", Design and Verification Conference US 2018 (DVCon US 2018), http://events.dvcon.org/2018/proceedings/papers/02_1.pdf, 2018.
[8] T. Yang, E. Qin, "Bridge the Portable Test and Stimulus to UVM Simulation Environment", Design and Verification Conference US 2018 (DVCon US 2018), http://events.dvcon.org/2018/proceedings/papers/02_3.pdf, 2018.
[9] P. Karppa, L. Matilainen, M. Ballance, "Building Portable Stimulus Into Your IP-XACT Flow", Design and Verification Conference US 2018 (DVCon US 2018), http://events.dvcon.org/2018/proceedings/papers/02_2.pdf, 2018.
[10] A. Vintila, I. Tolea, "Portable Stimulus Driven SystemVerilog/UVM Verification Environment for the Verification of a High-capacity Ethernet Communication Endpoint", Design and Verification Conference Europe 2018 (DVCon Europe 2018), http://events.dvcon.org/Europe/2018/proceedings/papers/06$_1$.*pdf*, 2018.
[11] G. Bhatnagar, C. Fricano, "Product Life Cycle of Interconnect Bus: A Portable Stimulus Methodology for Performance Modeling, Design Verification, and Post-Silicon Validation", Design and Verification Conference US 2019 (DVCon US 2019), http://events.dvcon.org/2019/proceedings/papers/10$_3$.*pdf*, 2019.