Control Flow Analysis for Bottom-up Portable Models Creation

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

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

***Abstract*—Portable Test and Stimulus Standard (PSS) is a game-changing standard in the field of simulation-based verification. This paper focuses on creating a top-level PSS model of the Design Under Verification (DUV) using PSS models of its components (submodels). This is one of the most challenging problems the PSS is currently facing, and it is called vertical reuse of portable models. The hardest part is to create proper constraints for the interconnection of submodels to represent behaviour intended to be verified. This paper aims to evaluate a hypothesis that with the analysis of the control flow inside the DUV, it is possible to significantly simplify the reusability of PSS models in the vertical direction. The control flow analysis can provide valuable information for creating constraints, as the control signals influence the behaviour of the DUV the most. As the DUV, the execution stage subsystem of the PULP platform processor was selected, which is an open-source representative of the RISC-V processor subsystem. Firstly, PSS models for all components inside this subsystem have been created. Then, the control signals of all these components were traced, and a map of dependencies from the subsystem point of view was assembled. Afterwards, the analysis was used to create constraints for the component-level PSS models while interconnecting them into the top-level PSS model.**

***Index Terms*—PSS, portable stimuli, portable models, simulation-based verification**

I. Introduction

The PSS [1] standard strives to simplify the definition of verification intent for stimuli generation. It aims to enhance readability, reduce redundancy, and promote portability among various platforms and design levels. Though advantageous, PSS is still new and faces challenges such as learning a new approach, immature tools, and achieving portability.

The primary goal of this work is to help verification engineers create portable models (PMs) and their transformations to different reuse scenarios. The main target is vertical reuse, which would allow the reuse of component-level PMs when building system-level PMs. The design selected for the experiments is the RISC-V processor from the PULP project [2], as it contains enough hierarchical layers for vertical reuse. PM for one of its components was manually implemented in the previous work, followed by theoretical ideas on how the control flow analysis can help in constraints definition at the subsystem level [3].

In the current paper, the progress in experimental work and the new findings are presented. Theoretical ideas from the previous paper were implemented. As a result, it is possible to practically demonstrate vertical reuse while showing how the control flow analysis actually helps in this process. The main objectives of this paper are:

- manual implementation of PMs for all identified components in the execution stage (EX-stage) of the RISC-V processor pipeline,
- control flow analysis of the EX-stage components,
- interconnection of PMs using the control flow analysis,
- creation of a PM for the whole EX-stage subsystem.

The paper consists of five sections. Section II briefly describes PSS and principles of PMs creation. It also explains the reuse options with a focus on vertical reuse and outlines the related work connected to vertical reuse. Section III shows a complete control flow analysis for all the EX-stage components of the RISC-V processor. Section IV outlines the implementation of their PMs, while also presenting the creation of the PM for the whole EX-stage subsystem using the control flow analysis. Section V concludes the paper and discusses the generalisation of the presented approach and options for future work.

II. Portable Models and Reuse

The verification intent is described as a set of rules forming a PM representing a set of scenarios for the DUV that will be checked. Nowadays, the definition of these rules is a manual job done by verification engineers, and it should be in conformance with PSS. The PSS compliant tools can visualise PM in the form of a graph to ease the debugging process. Moreover, tools automatically enumerate the minimum number of runs to cover the whole verification state space defined by PM.

The main abstraction mechanism in PSS is called *action*, representing a unit of behaviour. Depending on its purpose in the PM, it can use the DUV and verification environment functions via *exec_block* construct or combine other actions to create more complex behaviours.

The PSS provides various methods to limit the PM and reduce the size of its state space. Design specifications impose *resource constraints* that set practical limits on the PM, such as limiting the number of channels, states, and data flow items. Another type is *control restrictions* determined by coverage and constraints. The PSS tool automatically identifies the minimum set of runs required to cover the state space defined by the PM and its constraints.

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

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

From the related work connected to vertical reuse, it is clear that it is essential to involve some up-front planning. Otherwise, reuse can backfire and require more work without providing proportionate benefits [4]. In [5], it is stated that while moving from block to subsystem or system-level, several details may differ in the environment, such as memory addresses, device IDs, different constraints on certain operations, and sharing of resources. Paper [6] describes a complete cycle of interconnect bus verification - from IP to SoC-level, using PSS. One PM was reused on all levels, but *exec blocks* had to be rewritten to reflect specific requirements on every level.

III. Control Flow Analysis

The control signals drive the behaviour of the DUV, and when connecting components to a bigger system, they play a crucial role. The hypothesis is that the same applies to portable models, so the idea is to connect PMs for components to a bigger PM using the control flow analysis of these signals. How does this analysis work? The standard means of logical simulators (Fig. 1) allow to track assignments into every control input throughout the component hierarchy to see which signals influence the behaviour.

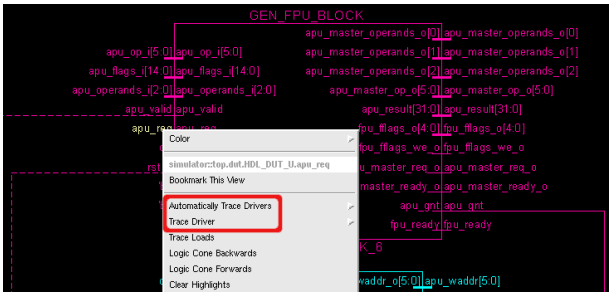

Fig. 1. Tracking signal drivers in an RTL simulator.

This analysis provides insight into how components influence each other, helping to form constraints for their PMs interconnection. Additionally, the control flow analysis will give information about smaller parts of the EX-stage, how they fit into the control flow, and influence its behaviour. The analysis itself is composed of:

- Isolation of control signals - an assumption is that the control connections and also restrictions between components are usually based on control signals.
- Analysis of control signals drivers - tracking assignments throughout the design, mapping the control flow which influences component's behaviour.

In the following sections, the control flow analysis of components inside the EX-stage is provided:

- MULT (Multiplication unit) - a computational unit for multiplication operations.
- FPU (Floating Point Unit) - a computational unit for operations with floating-point numbers.
- APU dispatcher (Auxiliary Processing Unit) - a control unit that is capable of offloading operations to the shared units and, at the same time, handling access contentions, checking data hazards, and write-back contentions with private execution units.
- ALU (Arithmetic Logic Unit) - a computational unit for arithmetic and bit-wise operations. Optionally, it can include division operations. For the purposes of this research, the division is included.

Please refer to Fig. 2 to better understand interconnections while reading the following sections. The main goal is to show the control flow analysis of the RTL code first and then describe how this analysis can be used for verification purposes and how it helps build the subsystem PM.

A. FPU Control Flow Analysis

- control inputs: *in_valid_i, out_ready_i, flush_i, rst_ni*
- control outputs: *in_ready_o, out_valid_o, busy_o*

Based on the analysis, the *flush_i* is constant-driven to zero, meaning FPU never interrupts its computation, throwing away the values. The *out_ready_i* is constant-driven to one, implying the FPU's environment is always ready to receive a result. The *in_valid_i* determines the validity of inputs, and if it is set along with the *in_ready_o*, signalling that FPU is ready, the unit will start the computation. It is worth mentioning that *apu_master_req_o* from APU drives *in_valid_i*.

B. MULT and ALU Control Flow Analysis

- control inputs: *rst_n, enable_i, ex_ready_i*
- control outputs: *ready_o*

These components are the same from the control flow point of view. Therefore, a joint analysis is provided.

Based on the analysis, the *enable_i* is driven directly from the Decoder (ID-stage) of the processor. The *ex_ready_i* determines that the component's environment is ready to process the result. The interesting thing about this signal is that part of its control flow includes the component itself, driven by the *ex_ready_o* logic. This logic comprises control signals from all over the subsystem, including the component's *ready_o*.

C. APU Dispatcher Control Flow Analysis

- control inputs: *apu_master_gnt_i, apu_master_valid_i, enable_i, apu_lat_i, rst_ni*
- control outputs: *apu_multicycle_o, apu_singlecycle_o, active_o, stall_o, read_dep_o, write_dep_o, perf_type_o, perf_cont_o, apu_master_req_o, apu_master_ready_o*

Decoder (ID-stage) sets the *enable_i*, one of the signals for request sending to the shared units, and the requests' latency on *apu_lat_i*. APU sends requests via *apu_master_req_o*.

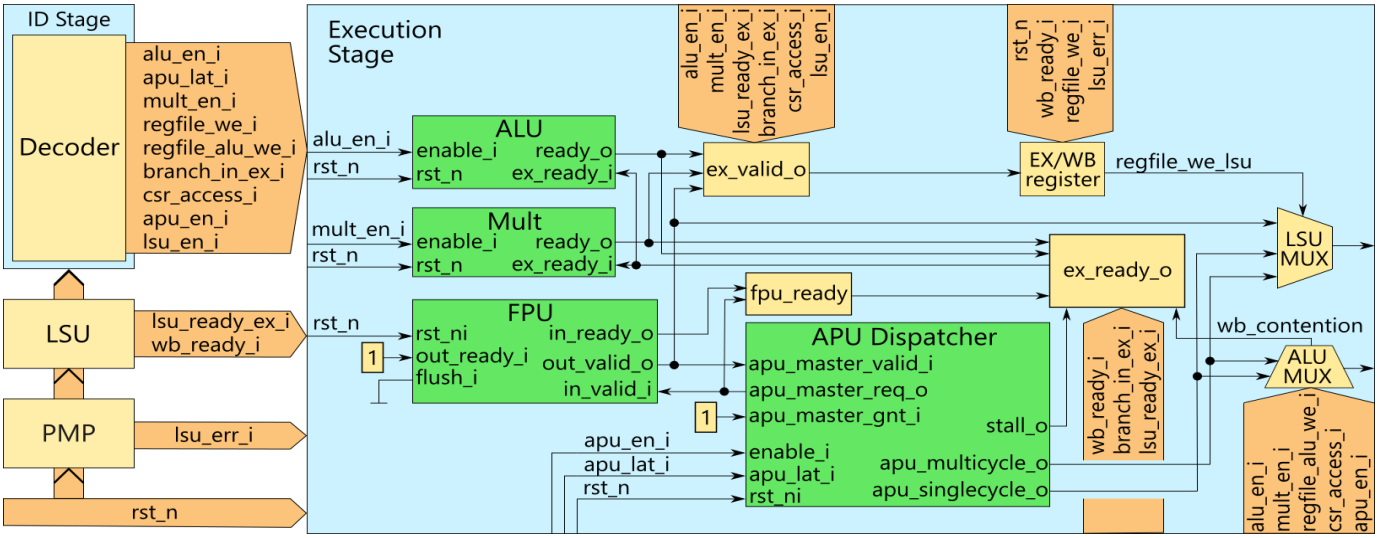

Fig. 2. The subsystem control flow analysis.

apu_master_gnt_i informs that the requested component is granted. It is constant-driven to one. The *apu_master_valid_i* confirms the validity of results from the granted component. It is controlled directly by the FPU's *out_valid_o* control signal.

D. EX-stage Subsystem Control Flow Analysis

The reset inputs within the EX-stage are globally driven. Through control flow analysis, five smaller components were identified that influence the subsystem's behaviour.

The analysis starts with two bigger combinatorial logics resulting in *ex_ready_o* and *ex_valid_o* signals. The former signals the subsystem's environment that it is prepared to receive new data, while the latter complements it by indicating the completion of processing.

To set *ex_ready_o*, all components in the subsystem have to be ready. It is worth mentioning that the APU and FPU combine their signals to report the readiness of the FPU. Another condition is an absence of stall, which APU can issue through *stall_o*, based on the received requests. The logic forming *ex_ready_o* also includes LSU (Load Store Unit) signals from outside. It informs that EX and WB stages are ready for new data. The last condition is no contention for storing FPU's result caused by different operation latencies.

Similarly to *ex_ready_o*, *ex_valid_o* requires ALU and MULT to be ready. Additionally, one of these conditions must occur: FPU result is valid, MULT or ALU is enabled, access to control status register or load from memory is issued. The signal conditions the data storage into the EX/WB register.

The last part consists of two multiplexers, one for the LSU write port and the second one for the ALU write port. The first one uses the values saved in the EX/WB register to determine if it writes two-cycle operations of FPU or loads data from the memory to the register file. The second one is used to write the results of the components to the register file and forward them to the ID-stage to use them for the subsequent computation.

IV. Portable Models

Once the analysis is complete, everything is ready to proceed with the verification process. This section outlines the PMs manually implemented for the EX-stage subsystem and its components. Firstly, PMs for the components were implemented as outlined in Sections IV-A and IV-B. Subsequently, Section IV-C demonstrates how is the information from the control flow analysis used to develop a subsystem-level PM. To enhance comprehension, this paper utilises graphical representations of the PMs. Moreover, all source codes of the PMs and DUV (to provide insight into DUV complexity) will be available on the authors' page [7] along with the analysis of all control signals of the EX-stage (only important ones are presented in the paper).

A. FPU, MULT and ALU Portable Model

From the PSS modelling point of view, most of the computational units follow the same pattern: one operation working with a required number of operands. Based on this fact, it is possible to generalise the description of PMs for FPU, MULT, and ALU components, creating a single base model that can be constrained and extended as needed. PM can be divided into three parts, an external verification IP (VIP), representing the input environment of DUV, a model of the verification intent, and an internal VIP, which takes the outputs of DUV.

The main focus is on the verification intent, as it defines a set of scenarios to explore during the verification of the DUV. The proposed general approach to its modelling is to divide it into smaller submodels, which can then be used to compose a model for more complex scenarios. In the case of FPU, MULT, and ALU components, two submodels were defined based on the specification: one for **reset**, second for computational **operations** (Fig. 3).

The **operation** submodel consists of three actions: *wait_rdy*, *operation*, and *wait_done*. Both wait actions are used for stimuli transmission control. The first waits for a component

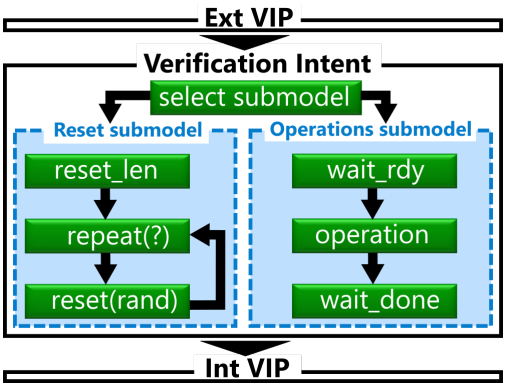

Fig. 3. A component-level model.

to be ready for a new operation. The latter is used to pair stimuli with their results. The *operation* action generates stimuli considering the defined coverage and constraints. The submodel for **reset** randomises the number of repeats for random stimuli generation with an active reset.

B. APU Dispatcher Portable Model

Compared to other components, the APU Dispatcher is used for control. The PM for it is divided into **reset**, **basic**, **latency**, and **hazard** submodels.

The **basic** submodel (Fig. 4) is for verification of APU behaviour without any hazards or contentions. The submodel is composed of seven actions, which cover the whole APU processing of the request. The submodel starts by requesting an operation with the *op_request* action. For verification purposes, the unit for the requested operation is granted with a randomly generated delay with the *grant_delay* action. During the delay, random values are sent with the *rand_vals* action, with the only constraint being that the unit is not granted. After the delay, the unit is granted with the *grant_unit* action.

The grant of the unit is followed by a random delay generated with the *valid_delay* action to simulate the processing time of the requested operation. The valid signal for result values is then sent with the *valid_vals* action. Similarly to the previous case, random values are sent with the *rand_vals* action during the delay, with the only constraint now being not to send the valid signal.

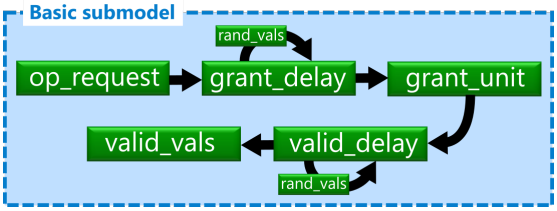

Fig. 4. Basic submodel for APU.

The **latency** submodel (Fig. 5) is intended for the verification of access contentions. The submodel consists of four actions. The first action, *grant_vals*, sends a request for an operation while also granting a unit for its execution. The request is followed by a random delay generated by the *valid_delay* action. The delay is within the range of the processing time of the possible operations. Random values are sent with the *rand_vals* action during the delay. The action has constraints that prevent the unit from being granted and from validating results with valid signal. The last action, *valid_vals*, sets the valid signal for the results of the processing unit. It also sets the grant signal, potentially starting another operation without delay. Access contentions are invoked by granting the unit by the *grant_vals* and *valid_vals* actions, while only the latter validates the results.

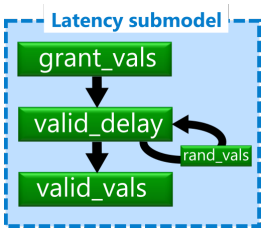

Fig. 5. Latency submodel for APU.

The **hazard** submodel (Fig. 6) is intended for verification of data hazards. It contains two actions, *send_vals*, and *send_hazard_vals*. Both actions request an operation and grant the unit for its execution without delay.

For achieving the occurrence of data hazards, several constraints are defined. The *send_vals* action has the latency of an operation set to a value greater than one to enable the formation of data hazards within the *send_hazard_vals* action. The other important constraints are on the registers being used. A new operation must use the same register as its preceding operation (the registers of operands for the read operation, the register for a result for the write operation).

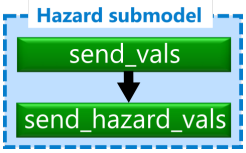

Fig. 6. Hazard submodel for APU.

C. EX-stage Subsystem Portable Model

The goal was to reuse the existing PMs as building blocks to create the PM of the subsystem. Moreover, some interconnection logic outside of the components had to be considered, which was identified by the control flow analysis (registers, multiplexers, control combinatorial logic).

When looking at the subsystem, it has a different outside environment than its components, meaning it has a different external VIP. The first step for vertical reuse was connecting the verification intent of PMs to the new external VIP through constraints based on the control flow analysis.

The next step was the definition of the verification intent. The specification guided its structure. For example, only one operation can be issued at a time, which resulted in a direct reuse of intents from the component-level PMs. FPU and APU models were merged for reuse convenience as they depend

on each other from the functional perspective (Fig. 2). The wait actions were moved from individual PMs' intents to the subsystem PM along with the submodel for the reset (Fig. 7). Moreover, the constraints of wait actions had to be modified as waiting depends on signals from all over the subsystem now.

Additionally to subsystem PM for reuse of computational units verification, two separate subsystem PMs were created based on the APU scenarios for verification of data hazards and access contentions.

To conclude, three subsystem PMs were created with significant reuse, one for verification of computational units (Fig. 7) and the other for data hazards and access contentions verification based on the APU scenarios.

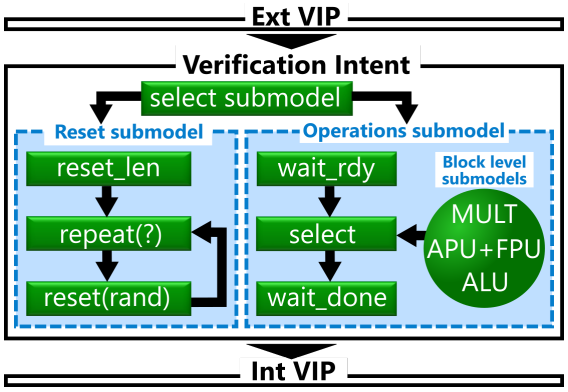

Fig. 7. The EX-stage's verification intent.

Alongside with the above-mentioned summary, the following three examples demonstrate how the control flow analysis can be used for constraint setup in the subsystem-level model.

Example 1:

- **Control flow analysis output:** signals *ex_ready_o* and *ex_valid_o* are combined to get the result of the current operation and commence processing of new input data for the next one. Both signals need information from every component in the EX-stage, as well as input signals from external sources, for example, LSU (Fig. 8).

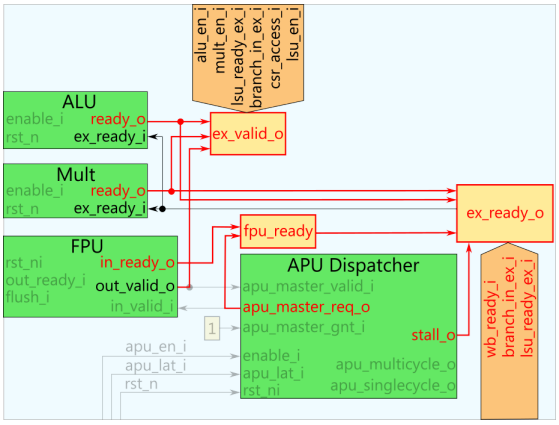

Fig. 8. Control flow analysis of *ex_ready_o* and *ex_valid_o* signals.

- **Update of PM constraints:** constraints for setup of operations and wait for results must be updated to reuse PMs of all components. First, the action *operation* inside the verification intent of the included block-level model (Fig. 3) must now include constraints for *lsu_ready_ex_i*, *wb_ready_i* and *branch_in_ex_i* signals to start the computation and get the result for single-cycle operations. These signals are direct inputs to the *ex_ready_o* from the outside environment of the EX-stage. Secondly, the action *wait_done* (Fig. 7) must include these constraints to get multi-cycle operations results. The updated constraint for signals on the EX-stage interface can be seen in Fig. 9.

```
// ex_ready_i == 1 / out_ready_i == 1
lsu_ready_ex_i == READY; // LSU ready
wb_ready_i == READY; // WB stage ready
branch_in_ex_i != READY; // not branching
// rest of ctrl signals
regfile_alu_we_i == 1; // forward back to ID-stage
regfile_we_i == 1; // write to register through WB
lsu_err_i == 0; // no errors for now on LSU
```

Fig. 9. Signals on the EX stage interface to set the ex_ready_o signal.

Example 2:

- **Control flow analysis output:** APU and FPU are tightly connected. All inputs and outputs of the FPU are processed by the APU. The latency of the FPU operation is set on the input signal *apu_lat_i*. This input is part of the logic in APU for the correct handling of access contentions, checking data hazards, and write-back contentions. The APU uses *stall_o*, *apu_multicycle_o* and *apu_singlecycle_o* signals for the control. For verification of the FPU operations without hazards, the latency on the APU input interface has to be set based on the generated operation (Fig. 10).

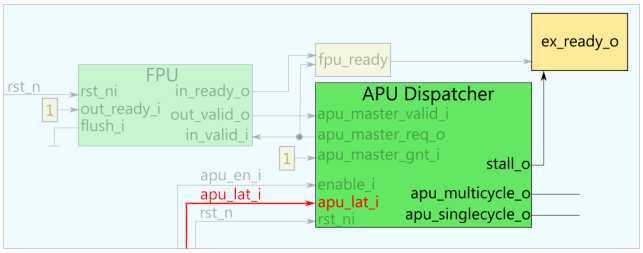

Fig. 10. Control flow analysis of FPU latency.

- **Update of PM constraints:** constraints for FPU operations setup must include latency constraints for the reuse of FPU and APU PMs (Fig. 11). In particular, latency for division and square root operations has to be set to three clock cycles and for the other operations to one.

```
if ( op_i in [DIV, SQRT] )
    apu_lat_i == 3;
else
    apu_lat_i == 1;
```

Fig. 11. FPU operation setup on the subsystem level.

Example 3:

- **Control flow analysis output:** the EX-stage can execute operations and also access the memory. These operations require the setting of their own enable signal to start, and based on the specification, only one operation can be issued at a time. Therefore, only one sub-block should be enabled by its *enable_i* signal (Fig. 12).

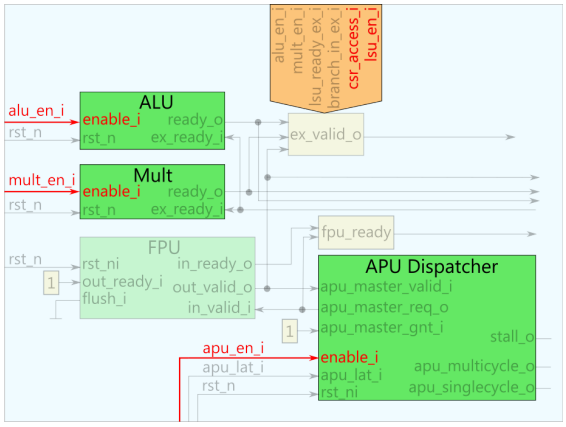

Fig. 12. Control flow analysis of operation execution.

- **Update of PM constraints:** constraints for the operation setup and waiting for the result must be updated to reuse PMs of all components. First, the action *operation* must be extended to include all enable signals. An example for the ALU can be seen in Fig. 13. Additionally, the action *wait_done* must disable all enable signals to not start another computation while waiting for multi-cycle operations to finish.

```
alu_en_i == ENABLED &&
mult_en_i != ENABLED &&
apu_en_i != ENABLED &&
lsu_en_i != ENABLED &&
csr_access_i != ENABLED;
```

Fig. 13. Example of a signal setup to enable only one operation.

V. Conclusion and Future Work

This paper presented an experimental work with the PSS application on vertical reuse utilising the control flow analysis.

The starting point was a control flow analysis of the EX-stage subsystem of the RISC-V processor, executed by tracking assignments throughout the design. The objective was to identify control signals and their potential influence on the behaviour of the whole subsystem.

The next step was the manual creation of PMs for all the components within the subsystem. These PMs were then utilised as building blocks for creating the PM for the whole subsystem. Information obtained from the control flow analysis was used to determine PMs interconnection and verification scenarios constraints, effectively constructing the subsystem PM. This is the primary outcome of the paper. Without the analysis, much more time would have been spent troubleshooting errors in the subsystem model, such as omitting specific signal settings or being trapped in an infinite loop.

This paper outlines an important step towards automation planned for the future. Specifically, the objective is to replace manual control flow analysis with an automated process, utilising a customised version of the open-source logical analyser Pyverilog [8]. The system/subsystem-level PM may be pre-generated, while actions can be automatically extracted from the block-level PMs. The most critical component, constraints, will be added based on pattern recognition from the control flow analysis output. These patterns will be defined in a dedicated library. Once this is achieved, conducting more practical experiments with diverse and complex DUVs and producing quantifiable outcomes will become more feasible.

Regarding the generalisation of this method, it should scale fairly to more complex systems. Discussions may arise about handling interfaces involving more control signals or standard bus protocols. The extent of control flow analysis is essentially limited by the capabilities of the RTL simulators, which determine the level of the DUV complexity that can be managed. The analysis can delve deeply, involving numerous signals in a chain, presenting too many details. A potential solution is to experiment with a restricted depth of the analysis, for instance, by stopping at the closest neighbour component within the subsystem and checking if the analysis still yields sufficient information for defining the subsystem PM constraints. Bus protocols are often handled by separate portable models, which can be excluded from the control flow analysis once identified, linking their PM to the subsystem PM instead.

To conclude, the experiments presented in this paper were time-consuming to do manually. Therefore, any automation achieved in this process would be valuable for all the verification engineers using PSS.

Acknowledgment

This work was supported by Brno University of Technology under the project number FIT-S-23-8141.

References

[1] Accellera Portable Stimulus Working Group, “Portable Test and Stimulus Standard,” online, 2019. [Online]. Available: https://www.accellera.org/downloads/standards/portable-stimulus

[2] Integrated Systems Laboratory of ETH Zürich and Energy-Efficient Embedded Systems group of the University of Bologna. (2021) PULP platform. [Online]. Available: https://pulp-platform.org/index.html

[3] P. Bardonek and M. Zachariášová, “Using Control Logic Drivers for Automated Generation of System-level Portable Models,” in *2020 23rd International Symposium on Design and Diagnostics of Electronic Circuits Systems (DDECS)*, 2020, pp. 1–4.

[4] M. Ballance, “Designing a PSS Reuse Strategy,” in *Design and Verification Conference Europe 2019 (DVCon Europe 2019)*, 2019.

[5] T. Fitzpatrick and M. Ballance, “Results Checking Strategies with the Accellera Portable Test & Stimulus Standard’,” in *Design and Verification Conference Europe 2019 (DVCon Europe 2019)*, 2019.

[6] G. Bhatnagar and C. Fricano, “Product Life Cycle of Interconnect Bus: A Portable Stimulus Methodology for Performance Modeling’,” in *Design and Verification Conference US 2019 (DVCon US 2019)*, 2019.

[7] P. Bardonek. (2021) PhD Code Snippets. [Online]. Available: https://www.fit.vutbr.cz/~ibardonek/web/phd/phdcodesnippets.html

[8] S. Takamaeda-Yamazaki, “Pyverilog: A Python-Based Hardware Design Processing Toolkit for Verilog HDL,” in *Applied Reconfigurable Computing*, ser. Lecture Notes in Computer Science, vol. 9040. Springer International Publishing, Apr 2015, pp. 451–460. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-16214-0_42