Automation of Verification for Application Specific Instruction Set Processors

Sreenivas Machavaram, Chaithanya Kolikipudi, Tal Milea, Parakalan Venkataraghavan

(sreenivas.machavaram, chaithanya.kolikipudi, tal.milea, parakalan.venkataraghavan) @intel.com

***Abstract*— Application Specific Instruction-Set Processors (ASIPs) achieve high performance and flexibility by using specialized instructions implemented in C-programmable custom hardware functional units (FUs). Typically, the instruction set is formally represented in a high-level language like C and custom hardware is implemented in RTL with several optimizations necessary for practical silicon implementation (e.g. resource sharing, pipelining). Verifying functional correctness of the RTL implementation through an automated process is a challenging task. In this paper, we describe an automated framework developed for constrained-random functional verification of custom FUs for an application-specific SIMD processor with n-bit (n in order of 1000's) wide data-path which has been instrumental in finding RTL bugs. We are currently working on extending this process to use formal techniques for more comprehensive data-path verification.**

***Keywords*— Verification, Instruction Set, ASIPs, Automation, HW & SW Co-Design, UVM, SystemVerilog**

I. INTRODUCTION

Intel's custom processor design toolchain enables the development of ASIPs with custom instructions. A custom processor is generated from a processor-template [1, 2]. In the template, custom instructions are implemented in functional units (FUs), which are instantiated in one or more issue slots of a core. The processor specification language TIM is used to describe the configuration of the core and the properties of the FUs. The RTL implementation of the instructions is specified in FU modules. Once the FU RTL and TIM are available, the processor RTL can be generated for the custom configuration. The TIM specification includes the timeshape[1] of the operands, their mapping onto input and outputs ports and the port widths. The compiler supports the scheduling of the custom instruction set based on the timeshape specified in TIM.

In our product line we use the custom processor design tools to build processors for all DSP applications.

For the verification of any processor it is important to bridge the gap between the HW and SW. Because of the complexity in the software applications, we cannot wait until the hardware prototype is ready to verify the software application. Due to this complexity, the SW teams usually develop Processor Independent (PI) simulation C models of the instructions. As shown in Figure 1, the same instruction set is defined in multiple formats: C, RTL, and TIM. The PI C models include only the functional abstraction of the Instruction. The Timing information is available in the other abstractions. The challenges for the verification team for testbench development and verification closure are:

- Check the coherency between all the design abstractions (C, TIM, and RTL) for the same instruction set. As shown in Figure 1, the RTL verification flow should bridge the gap between the SW verification and tool generation flows.
- Check the sanity of the operand/result connections with the input and outputs of the FU.
- Avoid manual testbench implementation. Testbench parameters, instruction grouping, opcodes, pipeline stages, and timeshapes are prone to rapidly change during the HW & SW co-design phase.
- Consolidate the different resources into a single testbench. The details of an FU are available across multiple resources. A lot of effort is required to gather the information. The instruction grouping, pipeline stages, and timeshapes are available in TIM, instruction functionality is available in C models and opcodes are available after the processor generation.
- Bridge the gap between the programming language used for the implementation of the SW model (C), and the language used in the testbench (SystemVerilog).
- Support simultaneous development of multiple variants of processors with custom FUs.

[1] "The time shape indicates in which cycle, relatively to the start of the execution of an operation, the corresponding port of the corresponding function unit is used for sampling an input value or producing an output value" [3]

Presented at WinTechCon-2019, organized by IEEE CAS Bangalore Chapter, IEEE Bangalore Section, and IEEE WiE Council Bangalore

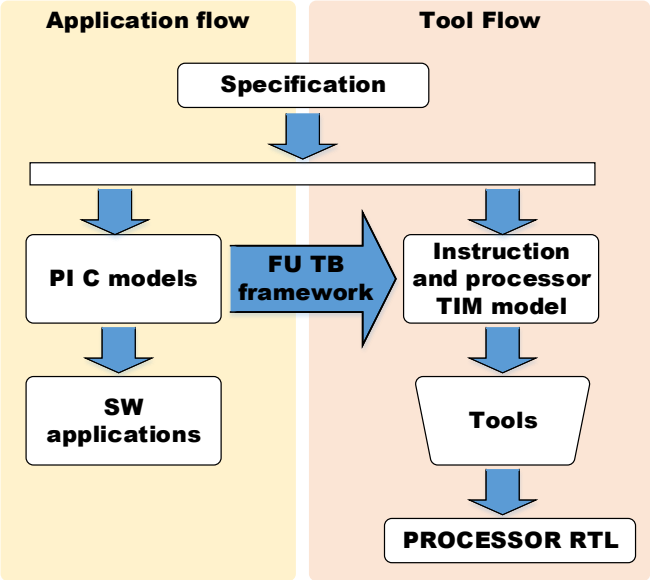

Figure 1 The FU TB framework bridges between the application and tools flows

In this paper we describe the solution we implemented while taking all of these challenges into consideration.

It is important to note that the testbench we implemented is meant to be complementary to formal verification of the FUs. The formal verification tools presented several issues. First, the tools encountered dynamic-memory allocation issues for designs with a large state-space. In addition, some tools do not generate waveform traces for exploring the RTL design (waveforms can only be generated as counter-examples for failing properties). Furthermore, the tools encounter issues when using VHDL or when using strings or very wide data types, which are used in our RTL designs or C models. These limitations made a constraint-random simulation-based testbench necessary. The testbench presented in this paper is used for comprehensive exercising of the FU control-path and to allow the designers to explore their design and achieve confidence. Formal verification techniques will be used to verify the data-path. In addition, collaterals for formal verification are planned to be generated in a similar automated approach.

The rest of this paper will be organized as follows: In section II we provide the details of an FU and the automated testbench generation. In section III we discuss the impact of the work on the quality of our products. We then conclude with a summary of the work in section IV.

II. FU TB Environment

The block diagram of a generic FU is shown in Figure 2. An FU can have any number of inputs with various widths. The maximal input operand width defines the maximal testbench width. Similarly, an FU can have any number of outputs with various widths. The number of clock pins and ip_stage_en pins define the number of pipeline stages in the FU. The operation_type port defines the input opcode. The FU can contain any number of instructions and the grouping is done as per the application. Each instruction can have one or many operands and results. Recall that the timeshape of the instruction defines the number of cycles taken by the instruction to finish and availability of operands and results on the input /output ports. It is not required for all the operands to be available in same cycle. The operands can be available on any port, including the use of the same port for different operands across multiple cycles. The same applies to the results on the outputs of an FU. An instruction cannot be scheduled on the FU if a previous instruction is occupying a port.

To build a testbench infrastructure for a custom FU, we need a UVM-based environment [4] which can generate constraint-random stimuli and an automated way of checking the RTL using the C implementation. The PI C models are integrated with the UVM testbench with DPI headers and integrated into the checkers.

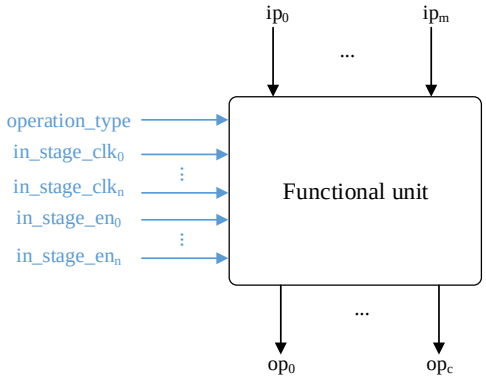

Figure 2 FU block diagram

After the processor is generated, the processor toolchain provides an XML file containing the processor properties. The toolchain also provides a Python API to query for properties and a framework for the user to implement file templates using Mako [5]. We implemented Mako and Perl file templates for all the FU-specific files: testbench, parameters, constraints, monitors, drivers, sequence library, test-cases, file list for simulation tools, compile scripts, DPI wrappers for each instruction, SystemVerilog DPI header files for imports, and instruction checker classes. Then all the above mentioned files are generated using Python and Perl scripts based on the information available in the core-specific generated XML file as shown in Figure 3.

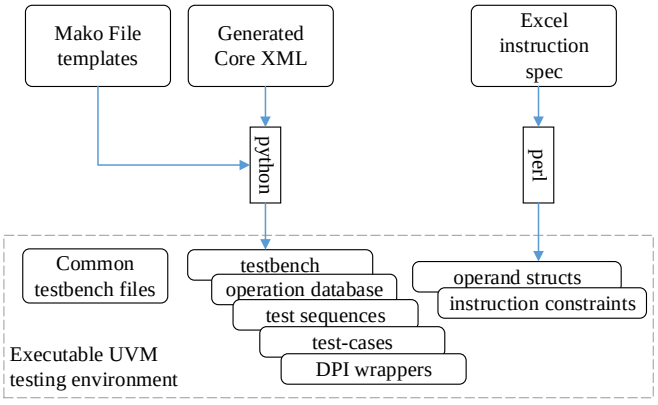

Figure 3 Verification collaterals generation from the core-specific XML and Excel instruction database

These generated files along with common testbench components (driver, monitor, and sequencer) will compose a generalized FU environment. The environment for an FU is shown in Figure 4.

The following information is extracted using the scripts and passed in different formats for the FU TB to parse:

- The number of inputs and outputs of the FU and the width of each port. Remember that an FU can have operands and results of any width.
- Mapping of operands and results of an instruction in an FU with its subset of input and output ports
- The grouping of the instruction set to FUs. This is used to generate an FU DPI checker with multiple DPI calls as per the instruction opcode. Instruction opcodes are machine generated and are assigned to instructions in the FU by tools.
- The timeshape of the instructions.
- Number of pipeline stages in the FU.

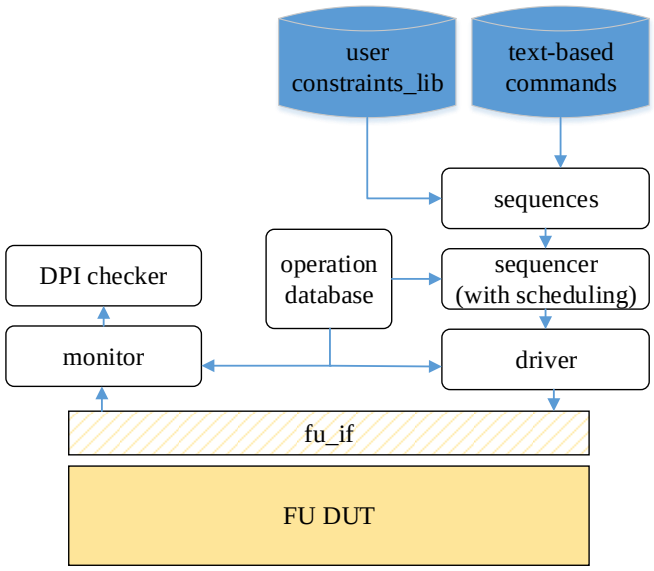

Figure 4 FU TB environment

As all this information is extracted from the generated core XML files, it bridges the gap between the distributed definitions of the instruction behavior. This is the same information available for the compiler to schedule the instructions.

Additional instruction description is specified in an Excel sheet by designers in a pre-defined format. This Excel sheet is used to generate operand constraint files for each instruction. Excel sheet per FU as shown in Figure 5. This document is used to generate the user constraint library and operand structs as shown in Figure 6. There are macros available in the sheet which the scripts parse to generate constraints on an operand. The same Excel sheet would be used to generate a word documentation using the scripts.

S. NO	INSTRUCTION	OPERAND	OPERAND NAME	TOTAL WIDTH	FIELD NAME	BIT POSITION	DESCRIPTION
1	add	OPERAND[0]	A	[31:0]	A	[31:0]	ANY:Vector Data
		OPERAND[1]	B	[31:0]	B	[31:0]	ANY:Vector Data
		RESULT[0]	R	[31:0]	R	[31:0]	ANY:Vector Data
2	shift_left	OPERAND[0]	A	[31:0]	A	[7:0]	ANY:Vector Data
		OPERAND[0]	B	[31:0]	SHIFT_AMOUNT	[11:0]	ANY:Vector Data
					SIGN_EXTEND	[13:12]	0,1,2
					RSVD	[31:14]	0
		RESULT[0]	R	[31:0]	R	[31:0]	ANY:Vector Data
	FINISH						

Figure 5 Instruction documentation in Excel

```
constraint c_op_add{
 if(opcode == OP_ADD){
   operand0.op_add_ctrlA.rsvd1 == 0;
   operand0.op_add_ctrlA.rsvd0 == 0;
   operand0.op_add_ctrlA.size inside {0,1};

   operand1.op_add_ctrlB.rsvd1 == 0;
   operand1.op_add_ctrlB.rsvd0 == 0;
   operand1.op_add_ctrlB.size inside{1};
 }
}

typedef struct packed{
  bit[MAX_OPERAND_WIDTH-1:256] rsvd1;
  bit[255:32] value0;
  bit[31:8] rsvd0;
  bit[7:5] mask;
  bit negate;
  bit[3:0] size;
} op_add_ctrlA_s;

typedef struct packed{
  bit[MAX_OPERAND_WIDTH-1:256] rsvd1;
  bit[255:32] value0;
  bit[31:8] rsvd0;
  bit[7:5] mask;
  bit negate;
  bit[3:0] size;
} op_add_ctrlB_s;
```

Figure 6 Generated constraints and structs for the OP_ADD operation

The UVM environment in Figure 4 contains an operation database which is built using SystemVerilog dynamic memory objects that include the instruction information (e.g. operands, results, cycles, port mapping). This database holds the information of the opcodes generated, operands and timeshapes for all instructions in that FU.

This database is extracted automatically. As illustrated in Figure 7, each opcode of an instruction has an entry. In this case the instruction OP_ADD has two operands and two results. The results are available on cycle 3 and the arguments on cycle 0 on different ports. This information is provided to the driver, monitor, and instruction scheduling sequencer. This will enable instruction scheduling without any resource (port/cycle) conflicts from the sequencer. The parameterized driver gets information of all the port widths and the cycle in which an instruction operand/result should be driven or sampled. A UVM-based cycle-accurate model samples the inputs and outputs as per the instruction timeshape and passes information to the instruction DPI checker. The instruction checker is a generated class that subscribes to the data coming from

the monitor and calls instruction PI C models using the DPI. The RTL results are then compared with the C model results.

```
//optype 3 (op_add)
  op_db[3] = new();
  //input A
  op_db[3].operands[0].pin_id = IP0;
  op_db[3].operands[0].cycle = 0;
  //input B
  op_db[3].operands[1].pin_id = IP1;
  op_db[3].operands[1].cycle = 0;
  //output C
  op_db[3].results[0].pin_id = OP0;
  op_db[3].results[0].cycle = 3;
  //output R
  op_db[3].results[1].pin_id = OP1;
  op_db[3].results[1].cycle = 3;
```

Figure 7 An entry in the generated operation database

Our framework also generates a sequence library as well as directed and random tests for each instruction. The environment provides hooks to run vectors that are flushed from C simulations. The algorithms team can run simulations and provide input test-vectors and their expected results. The traces are played-back on the RTL and the RTL results are checked against the C results. In addition, it provides hooks to the designers to write tests in a high-level string format. A built-in text parser in the environment is capable of generating the random stimuli for any instruction as per the string commands. This will enable the designer to develop simple tests without any knowledge on UVM or constraint-based testing. The designer can give a simple text file with a command and pass the values of operands of instruction (as shown in Figure 8). The text parser supports strings for different modes (random, random negative, random positive, negative/positive max/min, same data as previous cycle, and -1).

```
//syntax:
//<optype> <#instructions> <in0>, …, <inN>
//Examples:
OP_ADD 0x2 RANDOM 0xfffe0aaa
OP_ADD 0x2 NEG_MAX SAME_AS_PREV
OP_FRM 0x4 ZERO RANDOM_POS RANDOM 0x11111111
```

Figure 8 High-level test specification used by designers

As already mentioned, the FUs can have instructions with wide operands and may have a mix of the scalar and vector data (***n*** x scalar data width). In C, complex unions of structs are defined in the function arguments. In RTL, these correspond to bit arrays. To pass this data into DPI functions we used lowest byte level abstraction of arrays in the monitor to keep the monitor generic for any operand or result port widths. To pass these byte arrays into the DPI C functions, we used svOpenArrayHandle data types and the dpi_copy* functions as shown in Figure 9.

```
void op_add(
 const svOpenArrayHandle A_in,
 const svOpenArrayHandle B_in,
 const svOpenArrayHandle C_out,
 const svOpenArrayHandle R_out
){
   _t__int256 A;
   _t__int256 B;
   _t__int32 C;
   _t__int256 R;

   dpi_copy_256_sv2c(A_in, &A, SV2C);
   dpi_copy_256_sv2c(B_in, &B, SV2C);

   R = op_add_R(A, B);
   C = op_add_C(A, B);

   dpi_copy_256_sv2c(R_out, &R, C2SV);
   dpi_copy_256_sv2c(C_out, &C, C2SV);
}
```

Figure 9 SvArrayHandle in DPI wrappers. The functions dpi_copy* map SystemVerilog data types to C data types

For every instruction, wrapper functions are generated. These convert the svOpenArrayHandle data types into the struct data types used in the function. For all the scalar data types standard equivalent SystemVerilog data types are used. To achieve the automation of the DPI wrappers, we enforced the C coding guidelines on using consistent data type for a particular data bit vector width in RTL. In the C implementation, the arguments and results of an operation are available on different variables The RTL implementation differs in that several operands or results may be mapped to the same port (in different cycles). The function port argument ordering in the C functions and the arguments in the TIM instruction specification needed to be matched. The function names in the C models needed to be matched with instruction semantic names in the compiler. This enabled the automated generation of the DPI headers, DPI imports and checkers. DPI checkers ensure cycle -accurate checking of any random instruction.

The FU TB environment is used also to support load-store units (LSUs) verification. An LSU is a type of FU that is connected to a memory. This environment is particularly very useful for the verification of memories along with the LSU FU and does not require any additional formal verification.

The FU agent environment can be used in the processor level environment in PASSIVE mode. The processor simulation environment runs with multiple agents instantiated. All the agents in the processor level can be grouped to check instruction scheduling in full core. The random constraints generated per custom instruction can also be used for random assembly or C code generation at processor level.

III. Results

The environment was used to verify the SIMD instruction FUs of a vector processor. Once the RTL for a particular FU was developed, basic automated verification of all instructions was completed within one hour. This is at least one-week reduction in man hours required to set up an initial testbench. This also results

in the saving of many man-hours to maintain the TB and test cases when the design is changing.

The FU TB provides early access to random verification infrastructure for designers without requiring previous knowledge in UVM verification. Easy access to high level test cases and directed tests creates the opportunity to run directed stress validation of a particular instruction. Directed testing helped in finding bugs in SIMD FUs in both C and RTL.

Automatically generated random tests have exposed a bug in a custom instruction, causing saturation in an arithmetic-logic FU. Automation of the instruction set verification, along with formal techniques would help in coverage closure of 60% of the SoC, as there are 20+ such processors in the chip. There are plans to use the instruction level constraints for grand random C program generation at processor level and FU instruction level testcases and environment agents to be re-used in processor level testbench. This will uncover issues early in the stage which may/may not be exposed by compiler/application or C limitation.

This automation framework is being used to develop templates to be used by formal verification Tools. We are currently evaluating formal tools for testbench convergence.

IV. Summary

A myriad of generated FUs and their rapidly changing specification called for an automated way to create their testbench. In this paper we presented an automated framework to generate FU testbenches. The framework we developed:

- Minimizes the manual changes to verification collaterals as the instruction set evolves during HW & SW co-design.
- Closes the gap between HW and SW validation of the instruction set.
- Helps in enabling production-worthy first-silicon.
- Provides a simple interface for designers to drive data to the FU in order to achieve initial confidence in their design.
- Enables the possibility to validate a sequence of dependent instructions.
- Provides a push-button flow to generate an FU testbench. This reduces the many man-hours required to set up and maintain testbenches for multiple custom FUs.
- Consolidates the different abstractions of the design into a single testbench thus assisting in verification closure.
- Allows playback in RTL of traces generated by C simulations run on real algorithms. This provides high confidence in the correctness of the RTL for real applications.
- Generates FU agents that can be reused at the top level and will be used to compare the schedule of the compiler with the actual RTL schedule.

This approach can be used for any chip which also uses a 3rd party ASIP IP that supports a custom instruction set.

Acknowledgment

We would like to thank our manager Suresh Bandaru and the management at Intel and Custom Processor Group that enabled a cross-geographical and cross-business-unit collaboration work. We also want to thank Erik Rijshouwer for his tool support and guidance in implementing file templates.

References

[1] J. Leijten, G. Burns, J. Huisken, E. Waterlander, and A. van Wel, "A massively parallel reconfigurable accelerator," in Proc. International Symposium on System-on-Chip, pp. 165-168, November 2003

[2] J. Leijten and M. Lindwer, "Multiprocessing template for media applications," in Proc. Eighth IEEE International Symposium on Multimedia, pp. 475-480, December 2006

[3] A. S. Nery, N. Nedjah, F. M. Franca, L. Jozwiak, and H. Corporaal, "Automatic complex instruction identification for efficient application mapping onto application-specific instruction set processors," IEEE 5th Latin American Symposium on Circuits and Systems, pp. 1-4, 2014

[4] Universal Verification Methodology (UVM): www.accellera.org/downloads/standards/uvm

[5] Mako Templates for Python: www.makotemplates.org