Automation of Verification for Application Specific Instruction Set Processors

Sreenivas Machavaram, Chaithanya Kolikipudi, Milea Tal, Parakalan Venkataraghavan

(email: 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, like resource sharing, pipelining etc. Verifying functional correctness of the RTL implementation through an automated process is hence challenging task, which becomes even more difficult if the input space for each instruction is very large. In this paper, we describe an automated framework which we have 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 bugs in RTL implementation of instructions. 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, System Verilog**

I. INTRODUCTION

Intel Custom Processor Design Tools platform enables development of the ASIP with Custom Instructions. The RTL abstraction of the instructions are specified in Functional Unit modules (FU). The same information is passed to the tools in specific language formats to describe the core and system (TIM and HSD). Once the FU RTL, TIM (Timing Map) & HSD are available, the Processor Sub System RTL can be generated for the custom configuration. The TIM has the timing information and availability of operands/results on inputs/outputs and their widths. The RTL for FU have both the functional and the timing intent of the instruction. The custom Functional units with instruction implementation are instantiated in one or many instruction issue slots of the processor core. HSD format has the information of how the functional unit is connected in an issue slot of the processor, with the register files and memories. The compiler supports the scheduling of the custom instruction set with information available in the TIM and HSD Formats. As these instructions are user defined, there was no automated way of generating validation collaterals in the Custom Processor Design Tool Flow.

In our product line we use the Custom Processor Design tools to build processors for all the DSP applications.

For any processor verification it is important to bridge the gap between the HW and SW. Because of the complexity in the software applications, we can't wait until the hardware prototype is

ready to verify the software application. Because of the complexity in the application SW development, SW teams usually work on Processor Independent (PI) emulation C models. As shown in Figure **1** the same instruction set is defined in multiple formats in C, RTL and TIM. PI C models just have the functional abstraction of the Instruction. The Timing information is available in other design abstractions.

The challenges for a verification team for the closure of verification and developing the test bench are

- To check the coherency between all these design expressions (C, TIM, RTL, HSD) for same instruction set. As shown in Figure **1**, the RTL verification flow should bridge the gap between the SW verification and Tool Generation flows.
- Sanity of the Operand/Result connections with the input and output in TIM/HSD.
- Need to avoid manual test bench, test case changes for the FU RTL changes related to the instructions, instruction grouping, opcodes, pipeline stages, cycle availability of each operand and result. This is must during HW& SW Co-design phase.
- All the information required on the instruction grouping, instruction availability in a FU, Time shapes, opcode of the instruction, mapping of the operand and result are not available at one place until the processor core is generated by the Processor Tools until the core is generated with the custom Functional Units.
- Check sanity of the C and SV abstraction of the instruction in the System Verilog test bench.
- The data path is so huge and hence the C data types used are complex union of the structs. Which don't have an equivalent in System Verilog to enable DPI Checks. Extracting port widths from the generated cores and mapping them with C data types for the DPI checkers.
- Random verification environment generation for each custom Functional Unit for any custom processor.

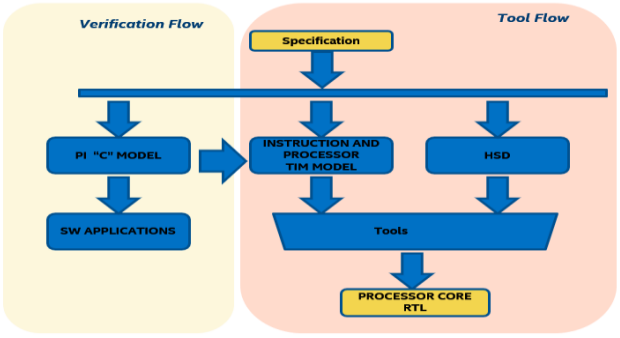

Figure 1 SW Verification and Processor Tool Flow

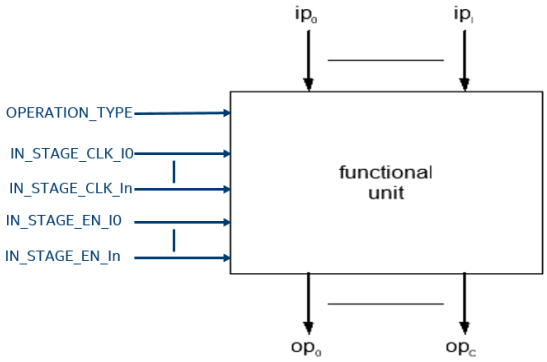

Figure 2 FU Block Diagram

There are some limitations for the industry standard formal tools on the dynamic memory allocation. The formal tools don't generate waveforms for passing case and generate only for the failing cases. They also have limitations using of very wide complex data types and string data types. There are also some limitations on the VHDL Language support for the formal Tools. So, any infrastructure built to support formal verification also have same limitations while writing the lemmas(properties) and constraints when the core is evolving as per application. So, the infrastructure built for automation and extraction of the information from the generated RTL core should be used to generate formal verification test benches. This environment should support initial control path data flow flushing for later use of Formal data path techniques.

II. FU TB Environment

The Block diagram of the FU is as shown in Figure 2. An FU can have any number of inputs and each having different widths. The max input operand width defines the max data path width. It can have any number of outputs of varied widths. Each output has an associated output valid strobe. The number of clocks pins and ip_stage_en* pins define the number of pipeline stages in the FU. Operation_type port defines the input opcode. The Functional unit 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. The Time shape 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 or same port. This applies to results ports on outputs of FU. Next instruction cannot be scheduled on the FU if the previous instructions have conflict on the ports for operands and results. An instruction time shape doesn't require to be same as another instruction time shape in the same Functional unit.

To build a test bench infrastructure for a custom Functional Unit of an ASIP, we need a UVM based environment which can generate constraint random stimulus and automated way of checking C models with the RTL implementation. The PI C models are integrated with the SV test bench with DPI headers and integrated into the checkers

So, initially we developed a frame work of templates for all the test bench, parameters, SV UVM Constraints, Monitors, Drivers, Sequence Libraries Test cases, file list, compile scripts, DPI Wrappers for each instruction, DPI Verilog Header files for imports and Instruction Checker Class. Then all the files mentioned are generated using the Python and Perl Scripts from the information extracted from the processor core generated xml files as shown in Figure **3**.

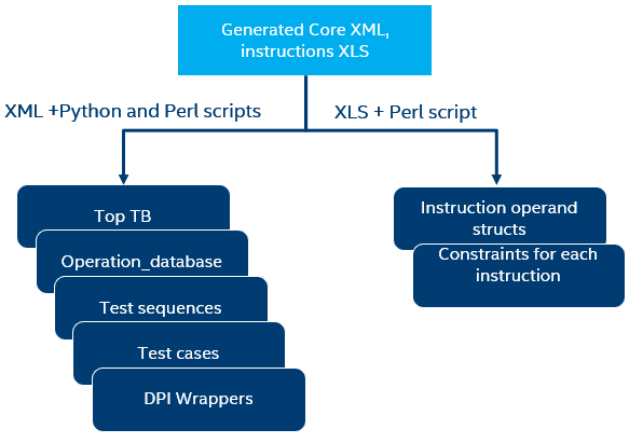

Figure 3 Verification Collaterals from Automated Flow

These generated files along with common test bench components driver, monitor and sequencer will create a generalized FU environment. The environment framework for a FU is as shown below in Figure **4**.

From the generated core from the Processor Tools, below information is extracted using the scripts and passed in different formats for the SV TB to parse.

- Number of Inputs and Outputs of the FU and the Widths of each port. Remember FU can have operands and results of any width.

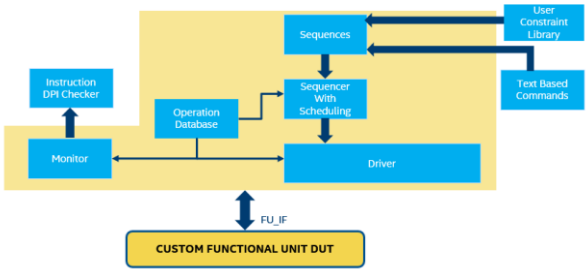

Figure 4 FU TB Environment

- Mapping of Operands and Results of an instruction in a FU with its subset of input and output ports
- Grouping of the instruction set to generate a 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.
- Time shape of the instruction (Cycle Availability of the instruction Operands and Results)
- Information of operands or results sharing same port
- Number of Pipe line stages in the FU.

As all this information is extracted from the generated core xml files and so it bridges the gap between multiple definitions of the instruction behavior. This is the same information available for the compilers to schedule the instructions.

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 test cases, 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 an operand. The same excel sheet would be used to generate a word documentation using the scripts.

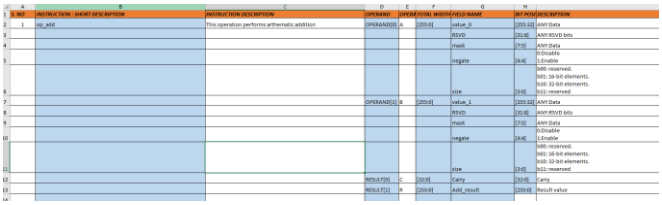

Figure 5 Instruction Documentation in Excel Sheet

In the UVM environment in Figure **4**, lies an Operation Database which is built using System Verilog Dynamic memory objects which has all the information about the Instructions (operands, results, cycles, port mapping etc.). This is database has information of opcodes generated, operands and time shapes all instructions in that FU.

```
constraint c_op_add{

    if (opcode == OP_ADD){

      operand0.op_add_ctrlA.reserved1  == 0;
      operand0.op_add_ctrlA.reserved0 == 0 ;
      operand0.op_add_ctrlA.size inside {0,1,2} ;

      operand1.op_add_ctrlB.reserved1  == 0;
      operand1.op_add_ctrlB.reserved0 == 0 ;
      operand1.op_add_ctrlB.size inside {0,1,2};



   }
}

typedef struct packed{
bit[MAX_OPERAND_WIDTH-1:256] reserved1;
bit [255:32] value_0;
bit [31:8] reserved0;
bit [7:5] mask;
bit negate;
bit [3:0] size;
}op_add_ctrlA_s;

typedef struct packed{
bit[MAX_OPERAND_WIDTH-1:256] reserved1;
bit [255:32] value_0;
bit [31:8] reserved0;
bit [7:5] mask;
bit negate;
bit [3:0] size;
}op_add_ctrlB_s;
```

Figure 6 Generated Operand SV Struct Data types

This database is extracted automatically. As shown 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 on cycle 3 and inputs are on cycle 0 on different ports. This information is fed both 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 time shape 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; // Input A
    op_db[3].operands[0].cycle = 0;
    // input B
    op_db[3].operands[1].pin_id = IP1; // Input B
    op_db[3].operands[1].cycle = 0;
    // output C
    op_db[3].results[0].pin_id = OP0; // Output C
    op_db[3].results[0].cycle = 3;
    // output R
    op_db[3].results[1].pin_id = OP1; // Output R
    op_db[3].results[1].cycle = 3;
```

Figure 7 Generate Operation Database Entry

The environment also generates directed random tests, sequence library for each instruction. The environment provides hooks to run the vectors generated from a C only simulation on the RTL. It provides hooks to the designers to write tests in the High-Level String format. Internal text parser in the environment is capable of generating the random stimulus to any instruction as per the string commands. This will enable the designer to try out some tests without any knowledge on UVM or constraint-based testing. The designer can give a simple text file as shown in Figure **8**, with a command and pass what values he want on an operand of instruction. The text parser supports strings for random, random negative, random positive, negative/positive max/min, same data as earlier cycle, negative one etc.

```
Syntax: <op_type enum type> <num_of_repeats of instr> <IP0> <IP1> <IP2> <IP3> <IP4> <IP5> <IP6> <IP7> <IP8>

Example: OP_ADD 0x2 RANDOM 0xfffe0aaa

Example: OP_FRM 0x4 ZERO NEG_MAX SAME_AS_PREV RANDOM_POS RANDOM 0x11111111
```

Figure 8 High Level Random Tests for Designers

As already mentioned the data path for the instructions is wide and operands have a mix of the scalar, vector data (***n*** x scalar data width). In C, complex union of structs are defined in the function arguments. In RTL these correspond to a 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 as shown in Figure **9** SvArrayHandle in DPI wrappers. 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 SV data types are used. To achieve the automation of the DPI wrappers, we needed to enforce the C coding guidelines on using consistent data type for a particular data bit vector width in RTL. 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 automation of the DPI headers, DPI imports and checkers. DPI checkers ensure cycle accurate checking of any random instruction.

```
void op_add (const svOpenArrayHandle A_in, const svOpenArrayHandle B_in, svOpenArrayHandle C_out
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 same environment is also used to support the Load Store Unit verifications which interface the memories. This environment is particularly very useful for the verification of the memories along with the FU and doesn't need any formal data path verification. The challenges of the instruction evolving do exist in the Load Store Units.

The FU agent environment is then used in the Processor level environment in PASSIVE mode to check at Processor level with multiple agent 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.

RESULTS

The environment was used in verification of the SIMD instruction FUs of the 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 test bench. This also results in the saving of many man hours to maintain the TB and test cases when the design is changing. This provides early access to Random verification infrastructure for designers. And is friendly to designers who may not have system Verilog verification expertise. Easy access to high level testcases and directed tests creates chance to do directed stress validation of a particular instruction. Lot of directed testing helped in finding bugs in SIMD Functional Units in both C and RTL. Close cross-BU collaboration between the verification team and Processor tool teams has helped this automation infrastructure to be adopted into the Processor tool Flow.

Future work: This automation frame work is being used to develop templates for data path Formal verification Tools. We are currently evaluating Formal tools for data path convergence. Automation of the instruction set verification, along with formal techniques would help in coverage closure of the 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 test bench. Which will uncover issues early in the stage which may/may not be exposed by compiler/application or c limitation.

SUMMARY

Using the automated frame work to develop the Functional Unit test bench

- Minimizes the manual verification collateral changes when the instruction set is evolving during HW & SW Co-Design.
- Closes the gap between HW and SW validation of the instruction set.
- Is in line with the shift left and production worthy A0 silicon approaches.
- Creates frame work for the Formal Data path verification support.
- Enables the possibility to validate behavior of group of instructions to form a macro operations in the FU test bench. This particularly is not possible in data path formal tools as they expect C model for full macro operations.
- Single push button flow to generate multiple test benches for different FU and reduces multiple man hours required to setup a TB for custom FU and thereafter to maintain.
- Closure of verification of different abstractions of same design across and integration of verification with application generated vectors from C is achievable.
- Ease of access to environment for designers to validate complete control data of the instruction using the FU TB and can rely on data path formal tools for exhaustive data path coverage.
- The FU TB agents generated can be re-used at the top level and will be used to compare the schedule of the compiler with actual RTL schedule.

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

ACKNOWLEDGMENTS

We would like to thank our Manager Suresh Bandaru, entire Management in Intel and Custom Processor Group to enable a cross geo and BU collaboration work