2014 DESIGN AND VERIFICATION DVCON CONFERENCE AND EXHIBITION INDIA
Sept 25-26, 2014
Hotel Park Plaza
Bangalore, India

A framework for verification of Program Control Unit of VLIW processors

Santhosh Billava, Saankhya Labs, Bangalore, India (santoshb@saankhyalabs.com)

Sharangdhar M Honwadkar, Saankhya Labs, Bangalore, India (sharangdhar@saankhyalabs.com)

***Abstract*— Program Control Unit of a VLIW (Very Long Instruction Word) processor performs several complex control intensive functions. Hence, it is critical to verify this comprehensively including all possible corner case scenarios. This paper presents a framework for functional verification of program control unit of VLIW processors. Though constrained random and directed tests are used for functional verification, it is difficult to cover all possible test combinations with these methods alone. The framework described in this paper improves the functional coverage by automatically generating targeted test cases for most of the test scenarios including the corner cases.**

Keywords— program control unit; VLIW; functional verification; directed tests.

I. INTRODUCTION

Very long instruction word (VLIW) processors are commonly used in software-defined radios (SDR) for baseband processing functions. These processors require very high performance digital signal processing capabilities to execute compute intensive algorithms like filtering, channel estimation, FFT, channel decoding etc. VLIW processors provide these capabilities through instruction level parallelism and pipelined execution.

VLIW processors have instruction sets similar to RISC processors with fixed or variable instruction widths. Each VLIW instruction encodes one or more operations for parallel execution. Instructions scheduling in VLIW processors are typically done at compile time using compilers. Compilers schedule at least one or more operations for the execution units. The number of execution units or execution slots in these processors depends on computational complexity requirements of target applications. These numbers of execution units directly translates to the length of instruction word.

Typical VLIW processors are pipelined and consists of a General Purpose Register file along with multiple execution units for computation, one or more Load-Store units to interface to a high bandwidth memory. A typical VLIW processor block diagram is shown in Figure 1.

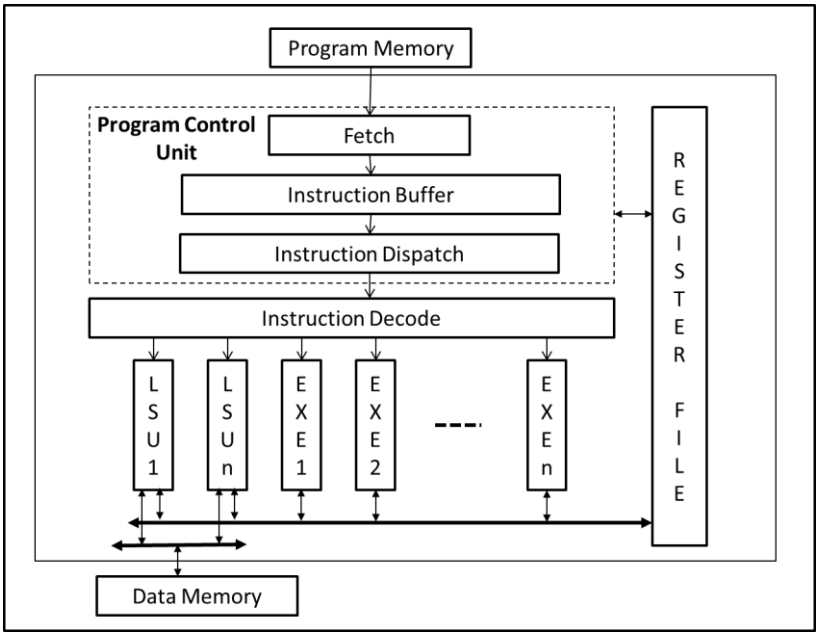

Figure-1: Typical VLIW Processor

II. PROGRAM CONTROL UNIT OF A VLIW PROCESSOR

Program control unit is one of the control intensive units in a VLIW processor. It fetches instructions from program memory and dispatches them to appropriate execution units. The program execution is typically not linear. There are program discontinuities like branches, jump, interrupts and exceptions. This unit monitors these activities and changes program execution flow accordingly.

In case of VLIW processors, number of instructions fetched from the memory and number of instructions dispatched to the execution units may not be same. In such cases, program control unit has to combine remaining instructions with next set of fetch instructions before successive dispatch. The program control unit uses an instruction buffer to decouple instruction execution and instruction fetch from program memory. This avoids unnecessary execution stalls due to delay in instruction fetch. At the same time these features increase the control complexity if a program discontinuity event happens simultaneously.

Program discontinuity in pipelined processors introduces execution gaps between detection of program discontinuity and execution of destination instructions. These gaps also called as instruction delay slots need to be handled properly by the program control unit. Effective use of these delay slots can be done by performing valid operations in these cycles. However in case of interrupts and exceptions the delay slots are typically flushed and hence no operations are performed.

VLIW processors support zero overhead software pipelined loops to avoid unnecessary delay slots for repetitive loops. During execution of these loops, program controller tracks the program execution, fetches instructions in advance and dispatches them appropriately. Number of instructions inside these loops may fit within instruction buffer or may not. Both of these cases have to be handled by the program controller.

III. VERIFICATION CHALLENGES

The Program Control Unit supports various features which pose significant verification challenges as explained in this section.

A. Program discontinuity

- Instructions like branch, jump, subroutine calls, interrupts and exceptions cause program discontinuity.
- Program discontinuity instructions have predicated and non-predicated variants.
- Predicated program discontinuity can happen with and without delay slot execution.
- Interrupts and exceptions with variable priority.
- Nested interrupts and exceptions.

  These features require the following combinations to be covered.

  - Back to back branch, jump & subroutine call combinations.
  - Branch, jump and subroutine calls with positive and negative offsets.
  - Predicated branch, jump and subroutine calls with all supported flags for true and false conditions.
  - Nested branch, jump and subroutine call combinations.
  - Branch, jump and subroutine calls with and without delay slot execution.
  - Checks to monitor precise number of delay slot instruction execution.
  - Delay slot execution with variable instruction set combinations.
  - Interrupts and exceptions before, after, and along with branch, jump and subroutine calls.
  - Interrupts and exceptions during delay slot execution.
  - Nested interrupts and exception combinations.
  - Exclusive branch, jump & subroutine calls.

B. Zero overhead software pipelined loops(SFPLOOP)

Zero overhead software pipelined loop is a feature supported by the processor to execute a set of instructions in a loop without any loop overhead for checking the loop counts. The hardware maintains loop counts and fetches required instructions in advance to avoid program discontinuity delays. The following test case combinations need to be covered to verify this feature.

- SFPLOOP with allowed range of instructions with all valid instruction set combinations.
- SFPLOOP with instruction block sizes less than, equal, and more than the instruction buffer size.
- SFPLOOP starting (or ending) with start, middle, and end of the VLIW instruction packet fetched from program memory.
- SFPLOOP occurring inside branch or jump outer loops
- Nested SFPLOOP
- Interrupts and exceptions occurring before, after, concurrently or inside SFPLOOP
- Branch, jump, and subroutine calls to exit SFPLOOP.
- Exclusive branch/jumps inside SFPLOOP.

C. Instruction Buffer

This is used for decoupling instruction fetch and execution units and needs to be verified with the following scenarios.

- Occurrence of execution stalls, debug stalls, and instruction fetch delays.
- Occurrence of program discontinuity when instruction buffer is empty or full.
- Instruction fetch request when instruction buffer is almost full.

D. Software debug features

Software debug features like break point, debug stalls, trace buffer, watch points, and single steps have to be verified during program discontinuity and zero overhead Software Pipelined Loops.

One of the key challenges of verifying a complex program control unit supporting the aforementioned features is to have a synchronized constrained random instruction generator along with program discontinuity event generator. As the test combinations of above features are huge, it is difficult to cover all possible test scenarios through constrained random simulations in a time bound schedule with finite resources. Hence a framework is needed for improving functional coverage.

IV. AUTOMATED TEST CASE GENERATION

The framework described in this section not only covers the basic features of Program Control unit but also improves the functional coverage by generating targeted test cases for critical boundary conditions. These are self-checking directed tests with input stimulus and expected outputs.

A. Test case generator

Test case generator is a script written in high level languages like System Verilog or C for generating directed test cases based on user constraints. The generated test cases are assembly level tests for CPU program control and Data path verification. Inputs to the test case generator are,

a) Sequences

b) Sequence groups

c) Sequence group order information

d) Configuration file.

Sequences are basic elements of the test case generator. A set of required sequences are identified and developed based on the targeted test scenarios. These sequences have two parts; first part consists of an assembly code and the other part consists of an equivalent behavioral code. The assembly code is used for generating assembly level program and the behavioral code written in high level language is used for calculating the expected outputs. A sample sequence is shown in Figure 2.

```
// Sample Sequence Used for Conditional Branch Exection/Selection
task conditional_branch_seq1;
begin
   bit cb_cbnop;
   cb_cbnop  = $urandom_range(0, 1);

   rmac2_exp(R16, R17, ACC1, ACC0, ACC1, ACC0);   // Complex MAC operation
   if(!eq_flag_set || !cb_cbnop)                  // Call functions based on flag
   begin                                          // and other conditions
      rnd_q31_to_q15_exp(R16, R18[15:0]);         // Fixed point conversion
      exp_calc(R16,R19);                          // Exponent calculation
      cmul2_exp(R17, R16, R23, R22, 1, R25, R24); // Complex multiplication
      cnmul2_exp(R17, R16, R19, R18, 1, R27, R26);// Complex* multiplication
      accx_truncate(ACC0, 16'd15, R20);           // Data movement
      accx_truncate(ACC0, 16'd0, R21);            // Data movement
   end                                                          } Behavioral Code and instruction selection

   $fwrite(ASM,"//conditional_branch_seq1\n");
   $fwrite(ASM,"{\n");
   $fwrite(ASM,"S3.RMAC2 R16, R17\n");
   if(cb_cbnop)
      $fwrite(ASM,"[S1.EQ] S1.CBNOP #L%-0d\n", label_count); // Branch without Delay
   else                                                      // slot instructions
      $fwrite(ASM,"[S1.EQ] S1.CB #L%-0d\n", label_count);    // Branch with Delay
   $fwrite(ASM,"}\n");                                       // slot instructions
   $fwrite(ASM,"S3.RND R16, R18\n");
   $fwrite(ASM,"S1.EXP R16, R19\n");
   $fwrite(ASM,"S3.CMUL2 SV8, SV11, SV12, #1 \n");
   $fwrite(ASM,"S3.CNMUL2 SV8, SV9, SV13, #2 \n");
   $fwrite(ASM,"S2.MOV ACC0, #0, R20\n");
   $fwrite(ASM,"S2.MOV ACC1, #0, R21\n");                    } Assembly Code
end
endtask
```

Figure-2: Sample Sequence

The sequences which are similar in feature and the sequences targeted to cover variants of a feature are hierarchically grouped together to form sequence sub-groups and sequence groups. The sequence groups may have sequences, sub-groups and other sequence groups. A sample sequence group is shown in Figure-3.

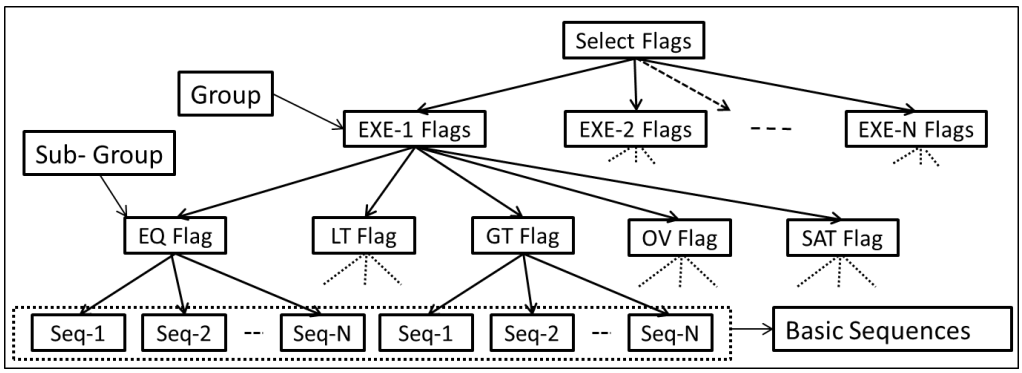

Figure-3: Sample Sequence Group

Sequence group order is information to the tool to select sequence group and sub-groups for a selected test scenario. The test cases are generated by randomizing and stitching sequences within sequence group/sub-groups as directed by the sequence group order. System Verilog *randsequence* or similar features are used for randomizing and stitching the sequences. A sample Sequence group order is shown in Figure-4.

A free running test case generator selects sequence group order randomly and generates random test cases. The test case generation is controlled with the help of configuration file. The configuration file constrains the tool to generate test cases for specific scenarios.

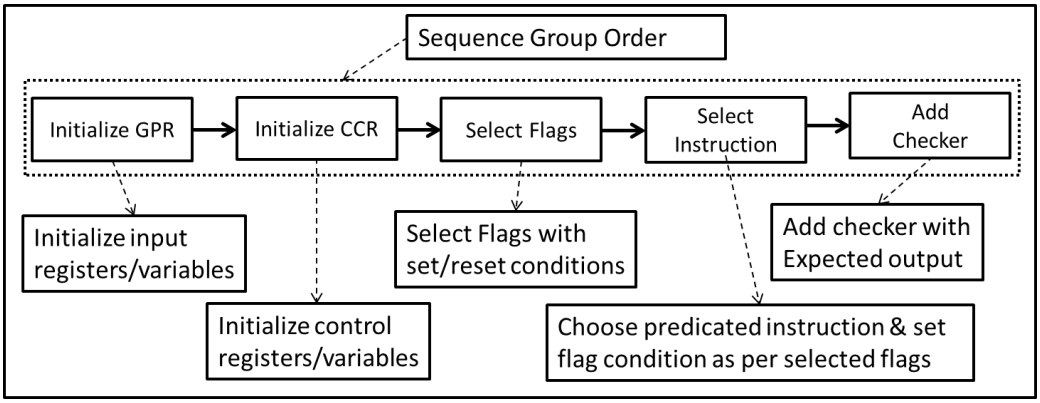

Figure-4: Sample Sequence Group Order

B. Test case generation

The flow diagram for test case generation is shown in Figure-5. The test case to be generated for a test scenario is randomly selected or taken from a user configuration file. Input stimulus to the test is provided through a pre-defined set of general purpose registers. Similarly expected outputs are re-directed to a pre-defined set of registers for self-checking.

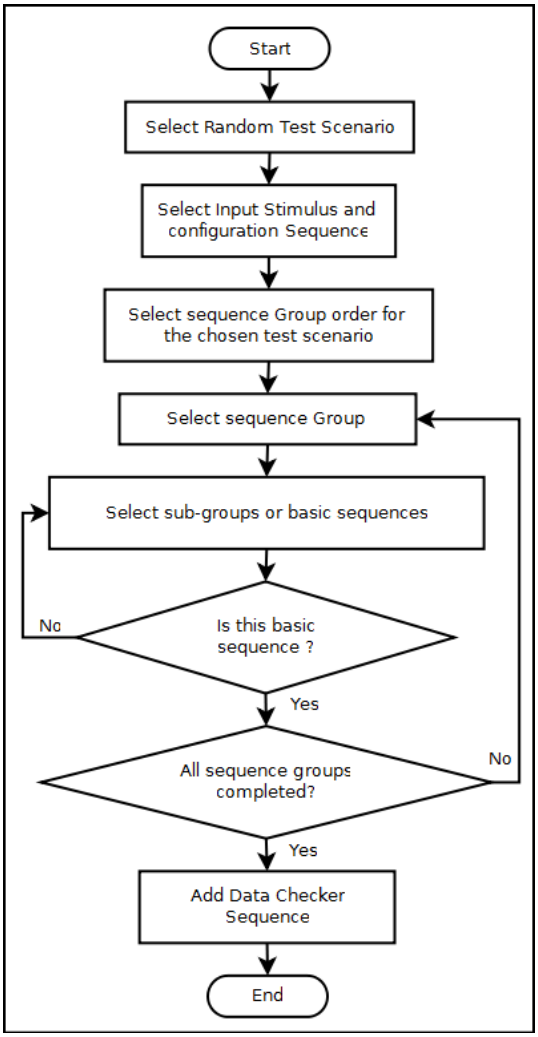

Figure-5: Test case generation flow diagram

Input stimulus to the test case is provided through initialization sequences. There could be more than one initialization sequences for a selected test. Number of initialization sequences is decided by the feature list of a selected test case. These initialization sequences are used for configuring control registers and general purpose registers.

After input initialization, test specific sequence group order is selected. This sequence order is pre-defined based on the event orders within a test case. These sequence groups may have sub-groups. These sub-groups are hierarchically selected till it reaches a basic sequence. Once the basic sequence is selected, it goes to next sequence group till it completes traversing all the required sequence groups. Since each of these sequences have behavioral codes written in high level language, the tool computes expected results for every test case and adds a checker at the end of a test case to make it self-checking.

An example for random sequence selection and stitching is shown in Figure-6. This example shows a test case generation sequence for "Interrupt occurring inside branch delay slots". The test scenario triggers an interrupt event during branch delay cycle execution of the processor. The test case is generated by calling sequence groups A, B, C, D, E, F and G in a pre-defined order as shown in Figure6. Each of these groups has hierarchical sub-groups.

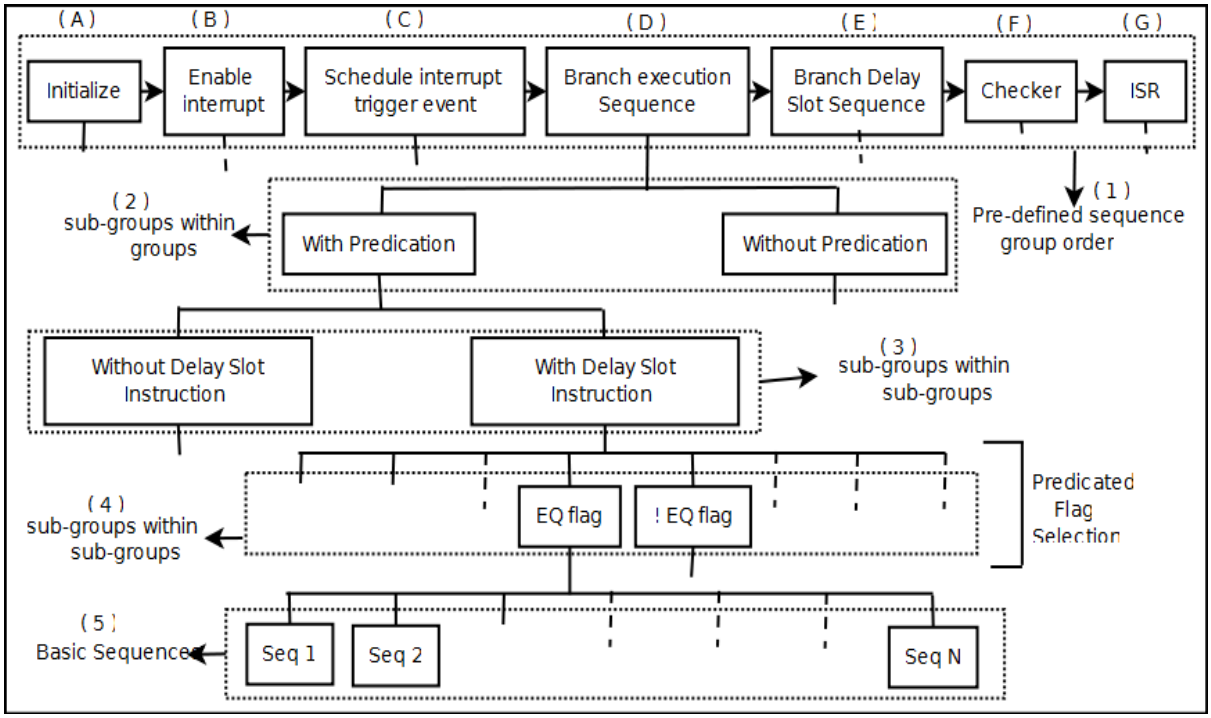

Figure-6: Sequence Group Selection Order

In the above example, the sequence group "Branch execution Sequence" has sub-groups "with predication" and "without predication". The sub-group "with predication" has further sub-groups "with delay slot instruction" and "without delay slot instruction". The sub-group "with delay slot instruction" has sub-groups for flag selection and each flags have basic sequences. In this example, the chosen order is "with predication" □ "with delay slot execution" □ predication flag "Zero" □ basic sequences "Seq1" to "SeqN". After selecting a basic sequence, it goes to the main sequence group "Branch Delay Slot Sequence". After completing all sequence groups, the stitching engine adds a checker with expected data.

V. CONCLUSION

This framework has been exhaustively used for verification of Program Control Unit for multiple VLIW cores in our group. The main advantage of this methodology is that the generated test cases can be re-used for FPGA as well as post silicon validation. Hence reduces the silicon validation time in addition to serving the purpose of providing good functional coverage. Since these test cases improved the functional coverage and covered all corner cases exhaustively, we were able to achieve first pass silicon success.

REFERENCES

[1] "IEEE Standard for System Verilog — Unified Hardware Design, Specification, and Verification Language". New York: IEEE 2005 (a.k.a. System Verilog Language Reference Manual, or LRM.)

[2] Philips Semiconductors, "An Introduction To Very-Long Instruction Word (VLIW) Computer Architecture"