UVM testbench design for ISA functional verification of a microprocessor

Gabriel Wang
Mediatek.inc, Beijing, China
gabriel.wang@mediatek.com

Hongtao Ma
Mediatek.inc, Beijing, China
hongtao.ma@mediatek.com

Maoduo Sun
Mediatek.inc, Beijing, China
maoduo.sun@mediatek.com

***Abstract*-This paper presents the design of UVM testbench for a microprocessor. Because of the large number of instructions and combinations of various instructions and operands, constrained random instruction is used fulfill verification driven by functional coverage. Design of the random instruction generator is emphasized. Several aspects of the random instruction generator are introduced, including instruction categorization and abstraction into UVM class, and layered instruction sequence. The execute-then-generate flow is also introduced to make instruction predictable. Some useful techniques, such as instruction constraint pool, register/memory mirror, virtual stack, and soft constraint are proposed and adopted to cope with dynamically updated inter-instruction constraints, random data dependency introduction, and random interrupt.**

I. Introduction

Processor verification is challenging as modern processors usually have large ISA (Instruction Set Architecture), different pipelines, variable operands, and complex constraints over instructions. Direct test pattern verification cannot fulfill verification requirements because it takes significant time and can hardly cover all possible instruction combinations. We turn to constrained random instruction to cope with this problem, and the completeness of verification can be measured by functional coverage. Another benefit of random instructions is that it may bring some “surprise”: it may produce certain scenarios or instruction combinations which would have never been produced by direct test pattern written by human.

We build our constraint random testbench based on UVM, which has been widely used in verification industry because of its solid and powerful infrastructure. In order to make random instruction generation effective and efficient, we adopted several schemes, making random instruction generation flow predictable, and implementing instruction constraint in different level. Another motivation of the schemes is to speed up the testbench development. By introducing instruction class and instruction sequence hierarchy, we can enable several engineers to develop UVM testbench in parallel.

The DUT described in this paper is a microprocessor with ISA of more than 300 32-bit instructions mainly including:

1) Common arithmetic instruction, such as addition, multiplication, division, logic operation, and shift.
2) Data Memory load/store instruction, such as load word/half-word, with different addressing scheme.
3) Peripheral register read/write instruction
4) Control instruction, including unconditional jump, conditional branch, and HW do loop, which usually causes PC(Programming Count) to redirect.
5) Customized algorithm instruction, such as rounding and clipping an integer value.

These instructions are executed in a typical 5-stage pipeline named IF, ID, E1, E2, and E3, with the support of HW stall, data forwarding and delay slot. Inside the DUT, there are 16 GPRs (General Purpose Register) and some special function registers such as SP (Stack Pointer), LR (Link Register), which can be used as instruction operands. There are also several peripherals such as VIC (Vector Interrupt Controller), which can handle with up to 32 external IRQ (interrupt requests).

Outside the DUT, there are DM (Data Memory) and PM (Program Memory), both with 32-bit data width. DM is the memory on which load/store instruction execute, while instruction binary is stored in PM from which DUT fetches instructions, decodes and executes them. Figure 1 show a typical partition of the PM, which is also followed by the testbench. Lowest 32 entries are filled with IVT (Interrupt Vector Table), which is occupied by an unconditional jump instructions jumping to one of 32 ISR (Interrupt Service Routine). On top of IVT and ISR, the “main code”, which is similar to the main function in C, is allocated in remaining higher space of PM. Normally after power-on reset, DUT starts from executing instructions at PC=0, which redirects to ISR0 entry(PC=128). ISR0 instructions is executed to perform certain HW initializations and ends up with another unconditional jump to main code.

It shall be noted that our verification scope is mainly on the DUT's ISA functionality, with less emphasis on peripherals. That is why we make a simplification on modeling external IRQ, which will be described in section B of chapter V.

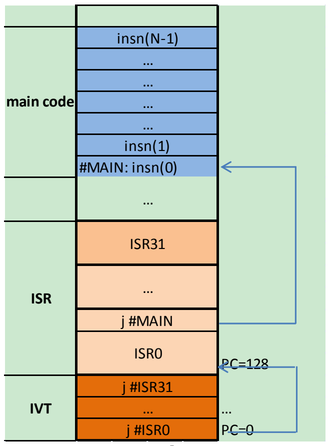


Figure 1. A typical PM partition

As far as the "ISA functional verification" concerned, a common addition instruction can be taken for example: "add rt, ra, rb". It is designed to perform addition operation "rt=ra+rb", where rt, ra, rb can be any one of the 16 GPRs, R0~R15. We think it should be verified as follows:

1) Correctness of the instruction operation: after the instruction executed, testbench should check that rt value equals the summation of ra and rb.

2) Completeness of the operand: testbench shall make sure all R0~R15 has been tested for rt, ra, and rb.

3) Combination with other instructions: this shall be verified for 2 main reasons.

Firstly, it is very common that an instruction may have data dependency with others and correspondingly pipeline may use HW stall or data forwarding logic to handle with this situation. An effective testbench shall cover this situation to make the corresponding logic exercised. Secondly, multiple instructions are usually used in a relatively fixed pattern by compiler or programmer, such as the "break" in a "do…while" loop body. Testbench shall make sure DUT executes these instruction combinations as expected.

The verification scale can be roughly estimated by all possible combinations of instructions in each pipeline stage, i.e. 300^5 = 243 billion, not considering the operand variations in each instruction. The huge number of ISA instructions poses much challenge in instruction modeling, random instruction generation, coverage sampling, especially when a relatively tight schedule is going to be met. In addition, there are dozens of control instructions which may cause PC redirection, and the program may be executed as a mess if not handled properly, section D of chapter IV will describe this situation and how to avoid it.

II. Organization of the Papers

Following chapters are organized as follows: Chapter III explains the terms used in our testbench, and then briefly introduce the UVM testbench structure and how testbench works. Chapter IV puts emphasis on random instruction generator, describing instruction categorization and abstraction, execute-then-generate flow and control subsequence design. Apart from random instruction generation, check mechanism and coverage sampling is introduced in chapter V. Chapter VI gives a summary based on previous chapters.

III. Testbench Design and Structure

A. Terms and definition

This paragraph defines some terms that will be used in following chapters.

instruction pool: A global pool which stores the instruction class object. Testbench can put and get a instruction from instruction pool using PC as index.

instruction constraint pool: It is similar to instruction pool, but stores instruction constraint class object for instructions at particular PCs. Instruction constraint and instruction constraint pool are described in detail in section C of chapter IV.

main code sequence: Instructions that are designed to be executed by processor after power-on and initialization. They are modeled as uvm_sequence that generate random instructions in testbench.

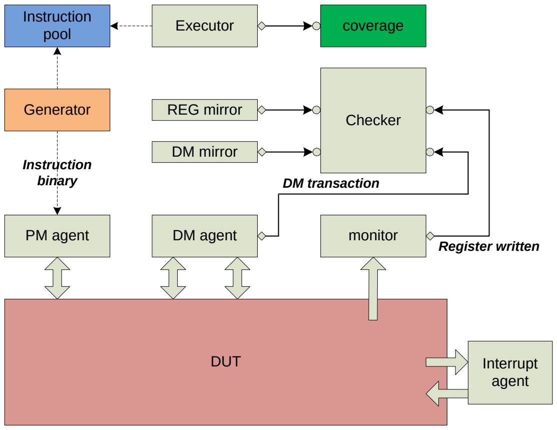
execute-then-generate: A method used when generating random instructions in main code sequence. It executes all previous existing instructions and update register and DM mirror, before generating a new one on next PC. By doing this, testbench can make use of latest register and DM value to avoid generating instructions may results in unintended scenarios such as infinite jump back. Execute-then-generate is described in detail in section B of chapter IV.

register mirror: An abstraction level model of the DUT's register. It provides read and write function for instruction class to complete instruction execution behavior. With the help of register mirror, testbench can execute instruction virtually and make execute-then-generate flow feasible. Register mirror also reflects the value change of each register in DUT, and provide the reference result when checking the register written transaction. Register mirror is described in detail in section B of chapter IV.

memory mirror: It is similar to register mirror, but stores the DM's read and written value.

virtual stack: A trick introduced in DM modeling to handle interrupt situations. When an interrupt happens and DUT enters into ISR, testbench replaces the stack region with another virtual stack, to let the push and pop instructions in ISR write and read values from the virtual stack, avoiding the stack region from being polluted by ISR. Virtual stack is described in detail in section B of chapter V.

B. Testbench overview



A testbench structure is proposed as shown in

Figure 2. It is a typical UVM testbench consisting of random stimulus generator (generator), reference model (executor), scoreboard (checker), and monitor which monitors internal DUT signals and helps executor on scheduling reference instruction execution and result check.

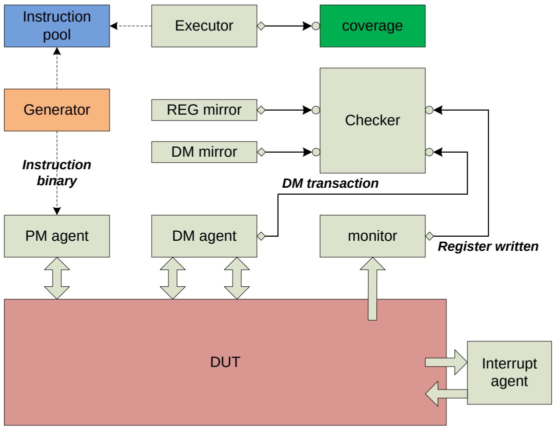


Figure 2. Testbench structure

Among the testbench components, the random instruction generator is the key of testbench, because the verification is mostly achieved by DUT executing different instructions and their combinations. It is straightforward to model a single instruction as a "sequence item" in UVM, and series of instructions as "instruction sequence" extended from uvm_sequence. The generation of random instructions is implemented in body() task of instruction sequence, it also handles the constraints imposed by the context of multiple instructions. The instruction generator is modeled as an uvm_component which starts one or multiple instruction sequences and fills the PM with binary values of the generated instruction. Besides being represented as binary code in PM, each generated instruction is also an instruction class object, which will be put in a container called **instruction pool**. Instruction pool acts like a high abstraction level counterpart of PM, which is a singleton class of the parameterized uvm_pool type. Instructions indexed by its 32-bit PC can be easily accessed via put() and get() functions predefined by UVM. Section A of chapter IV will describe how the instructions will be categorized and modeled in a high abstraction level.

```
// instruction pool class
typedef uvm_pool #(bit[31:0], instruction_base) instruction_pool;
```

After all the instructions which are generated by instruction sequence to be executed by DUT, the instruction binary will be put into PM. Random instruction generator is basically a component that starts instruction sequences and fills instruction binary into PM before DUT are started by a power-on reset.

As shown in Figure 1, IVT and ISR instructions are basically fixed or semi-fixed, and they are modeled as IVT and ISR sequence, which will be introduced in section B of chapter V. The "main code" consists of random instructions testbench shall generate and is modeled as **main code sequence**. Chapter IV will describe different aspects of the main code sequence design in detail.

IV. RANDOM INSTRUCTION GENERATION

A. Instruction categorization and abstraction

Although the ISA has 300 instructions, some instructions perform similar operations and have similar pipeline stages with slightly different operands. For example, ISA provides 4 instructions for addition operation as shown in TABLE 1, and they all read source register value in E1 stage, while perform addition operation in E2 stage, and write summation to the target register. Intuitively, it is a good choice to regard these instructions as the same type and encapsulate them into a class, naming alu_add. Based on the above categorization, the 300 instructions in ISA are partitioned into 40 types, correspondingly 40 classes.

TABLE 1
VARIATIONS OF ADDITION INSTRUCTION

Instruction	operation
add rt,ra,#imm16	rt=ra+#imm16
add rt, ra,rb	rt=ra+rb
addx rt,ra,rb	rt=ra+rb+mc
add rt, ra,rb,#1	rt=ra+(rb<<1)

Above all the 40 instruction classes, there should be a common parent class, since all instruction classes share similar properties and shall have consistent functions/task interface for use. So, class hierarchy in UVM testbench is as shown in Figure 3.

In the instruction_base class, several kinds of information are defined.

1) Basic information for an instruction, such as pc, instruction, pipeline stage information.

2) Global resources, such as register mirror, instruction constraint pool.

3) Common function tasks for external use: such as execute(), adjust_read_operand().

The concept of register mirror, instruction constraint pool, and the usage of adjust_read_operand() will be described in chapter IV in detail.

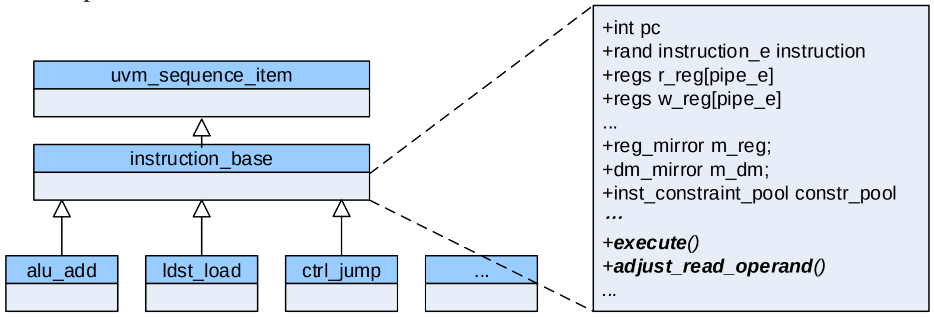


Figure 3. Instruction class hierarchy

It is noted that ISA functional verification in this paper do not include pipeline stage timing verification, which means the instruction class models instruction behavior in a high abstraction level. Take "add rt,ra,rb" for example, DUT will execute it in a 5-stage pipeline and each pipeline performs a separate operation. The pipeline propagation is more complicated when data dependency exists between multiple instructions and HW stall or data forwarding happens. Instruction class in behavior level, however, only need to implement the "rt=ra+rb" operation in its execute() function. Although r_reg[pipe_e] and w_reg[pipe_e] are defined in instruction_base class, they are for coverage sampling but not really for a pipeline operation.

By defining the above skeleton of instruction_base, 40 extended instruction classes are quite suitable for co-operative development by multiple verification engineers in parallel, each instruction class developer only need to fill in the function/tasks pre-defined in parent class. Besides that, their maintenance are much easier than those put all instructions into a single instruction class. Here is an example of implementation of alu_add.

```
// Code Example A1: alu_add class implementation
class alu_add extends instruction_base
  rand reg_e rt, ra, rb;
  rand bit signed [15:0] imm16;
  constraint constr_instruction {
     instruction inside {ADD_RRI,   // add rt,ra,#imm16
                         ADD_RRR,  // add rt, ra, rb
                         //ADDX_RRR, …
                         }
     if (inst_contr != null) { !(instruction  inside {inst_contr.instruction_exclude});}
  }
  constraint constr_operand {
     rt inside {[R0:R15]};  ra inside {[R0:R7]};  rb inside {[R0:R7]};
     if (inst_contr != null) { !(rt  inside {inst_contr.w_reg_exclude});
       !(ra inside {inst_contr.r_reg_exclude});
       !(rb inside {inst_contr.r_reg_exclude});
     }
   }
  virtual function bit [31:0]  execute();
    bit [31:0] ra_val,  rb_val;
    bit [31:0] rt_val;
    bit [31:0] next_pc;
    //1. read operand value from register mirror
    ra_val = m_reg.read(ra);  rb_val = m_reg.read(rb);
    //2. perform addition and write back to register mirror
     case (instruction)
       ADD_RRR: begin  rt_val = ra_val +rb_val; m_reg.write(rt,rt_val); end
       //ADDX_RRR and other instructions…
     endcase
    //3. move forward to next pc
    next_pc = this.pc +4;
    return next_pc;
  endfunction
endclass
```

B. Make instruction execution predictable

Imagine an instruction sequence that would be randomized over different instructions and different operands without considering any constraints by context, which is shown in TABLE 2.

TABLE 2
INSTRUCTION SEQUENCE RANDOMIZED WITHOUT CONTEXT CONSTRAINT

PC	instruction	operation
0	mv r1,#0	**r1**=0
4	sub r1,r2,r8	**r1**=r2-r8
8	add r4,#64	r4=64
12	or r3,r4,r5	r3 = r4 | r8
16	lw r13,#0(r1)	r13 = DM[**r1**]
20	j **r1**	unconditional jump to pc: **r1**

Above instructions can be executed by DUT without any problems until PC=16 and 20, where a DM load and an unconditional jump instruction are located. The behavior of both instructions depends on the value of r1, which is determined by the execution results of instructions at PC=0, 4 and their precedents. If the value of r1 is too large, the DM load instruction shall access an illegal DM region. If it is too small, for example, 8, the unconditional jump may jump back to PC=8, resulting in an infinite jump. Obviously, both situations are not what we want.

Apparently, if the value of r1 is known when randomizing instructions at PC=16 and 20, the above troubles can be pre-detected and thus avoided by replacing with a different instruction or different operand. So, an “execute-then-generate” flow is adopted when generating instructions in main code: it will first execute existing instructions and record the execution results on a high abstraction level, and then step forward to generating the next instruction. In this way, all register values can be known before a new instruction generated randomly, and they are exactly the same as instructions will be executed in HW DUT afterwards.

To record the values of registers on a high level, a "register mirror" class is constructed as below. The principle of register mirror is similar to the "mirrored value" in UVM Register Abstraction Layer. It is a singleton class with common read(), write(), and reset() functions, providing the instruction and instruction sequence with a medium to operate on. Code example A1 in section A of chapter IV shows the usage of "m_reg" in the execute() function of a instruction class.

```
// Code Example B1: register mirror class
class register_mirror extends uvm_component;
    static protected reg_mirror m_reg;
    static bit [31:0] r_value[reg_e];
    …
    static function register_mirror get_global();
       if (m_reg==null) begin
          m_reg= new("m_reg",null);
       end
       return m_reg;
    endfunction
    function void reset();
    function bit[31:0] read(reg_e r);
    function void write(reg_e r, bit[31:0] value);
endclass
```

By introducing the register mirror, the "execute-then-generate" flow can make the randomization of instructions highly controllable and predictable, and thus making instructions be safely generated without causing unexpected PC redirect or illegal operation.

Similar to register mirror, a DM mirror is also constructed, which makes the load store instruction fully predictable. The only difference of DM mirror is that it needs to store the random return value when reading an unwritten address. These values will be used as the memory initial value for "DM agent" as shown in

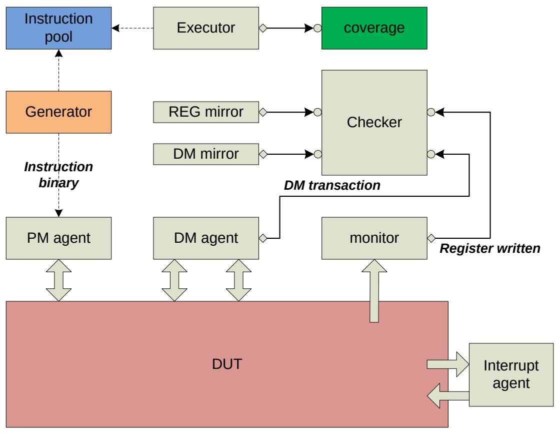


Figure 2. An associative array is used to store these values, since normally only a small portion of DM space (2^{32}=4GB) will be accessed in an instruction sequence. This makes the testbench get the same result when DM load/store instruction is executed in high level instruction sequence and HW DUT.

```
// Code Example B2: DM mirror class
class dm_mirror extends uvm_component;
   protected bit[31:0] mem[bit[31:0]];
   protected bit[31:0] mem_init[bit[31:0]];
   …
   static function reg_mirror get_global();
   function bit[31:0] read(bit[31:0] addr);
   function void write(bit[31:0] addr, bit[31:0] data);
endclass
```

C. Instruction constraint: impose different levels of constraint

There are basically two kinds of instruction constraints in testbench: intra-constraint and inter-constraint. Intra-constraint is mostly imposed for operand/value range limitation of a single instruction in each instruction class. Intra-constraint can be easily modeled like the "constr_operand" as shown in code example A1 in section A of chapter IV.

Inter-instruction constraint, on the other hand, is mostly imposed dynamically by instructions' context. For example, ISA rules that a division instruction shall not be on the delay slot position, and there are some illegal instructions which shall be excluded in the unconditional jump target position. During randomizing instruction, testbench shall impose these constraints to avoid illegal instruction combination. Similar to the instruction pool, an instruction constraint pool was constructed so that these dynamically updated "instructions to be excluded" could be easily put in and get from the pool by other instructions. According to the characteristics of different inter-instruction constraint, instruction constraint wass designed to indicate 3 layers of "to-be-excluded": instruction types, instructions, and register operands.

```
// Code Example C1: instruction constraint class
class instruction_constraint extends uvm_object;
  instruction_type_e           exclude_types[$];
  instruction_e                exclude_inst[$];
  reg_e                        exclude_r_regs[$];
  reg_e                        exclude_w_regs[$];

  function void add_exclude_type(instruction_e types[$]);
  function void add_exclude_inst(instruction_e insts[$]);
  function void add_exclude_r_reg(reg_e regs[$]);
  function void add_exclude_w_reg(reg_e regs[$]);
endclass
```

With the 3 layers of inter-constraints, the generation flow of an instruction at a certain PC could be as follows:

1) Testbench randomly chooses an instruction type, with excluded types removed from candidates.

2) After a certain type is chosen, an instance of that type is randomized with exclusive constraints in instructions and operands.

3) Put randomized instruction into instruction-pool.

4) Update constraint-pool, as current instruction may add new constraints to the context.

Code example C2 shows an example of how instruction constraints will be dynamically got from and put into the constraint pool. It is noted that, once after the instruction constraint handle is assigned to an instruction object, the instruction and operand level constraints implicitly take effect when the randomize() function is called.

```
// Code Example C2: dynamically update constraint pool in main_code_sequence
  task body();
    …
    pc = next_pc;
    …
     if (constr_pool.exists(pc)) begin
         inst_constr = constr_pool.get(pc);
         exclude_types = {exclude_types, inst_constr.get_exclude_type()};
     end
     …
     std::randomize(inst_type) with {!(inst_type inside {exclude_types})};
     cur_inst = inst_type::type_id::create($sformatf("inst_at_%d", pc));
     cur_inst.inst_constr = this.inst_constr;
     //instruction and operand level constraint will take effect when calling randomize()
     //as shown in code example A1 of section A of chapter IV.
     cur_inst.randomize();
     inst_pool.add(pc, cur_inst); //put instruction into instruction pool

     //put newly imposed instruction constraints to pool
     inst_contr = constr_pool.get(sequetial_pc);
     if (cur_inst.get_delay_slot() ==1) //delay slot constratint
        inst_constr(sequetial_pc).add_exclude_type(ALU_DIV);
     //if certain register need to be protected from being written, below code can be used.
     //  inst_constr(sequetial_pc).add_exclude_w_reg(R1);

     next_pc = cur_inst.execute(); //execute this instruction to make register/memory predictable
    …
  endtask
```

D. Layered sequence: main code and control subsequence

Basically, main code sequence can be regarded as random mixed instructions of normal, non-control instructions and control instructions. As mentioned in section C of chapter IV, control instruction usually involves context instruction combination and needs special handling like specific instruction generation and constraint pool updating. So control instructions such as unconditional jump, conditional branch, HW do loop and their context instruction were naturally modeled as a **control-subsequence**.

As shown in Figure 4, a control-subsequence usually contains a control instruction, its target instruction, related delay slots, body instructions, etc.

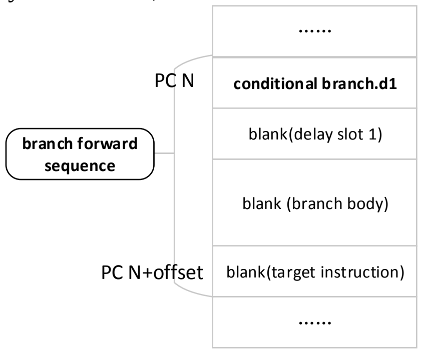


Figure 4. Branch control-subsequence

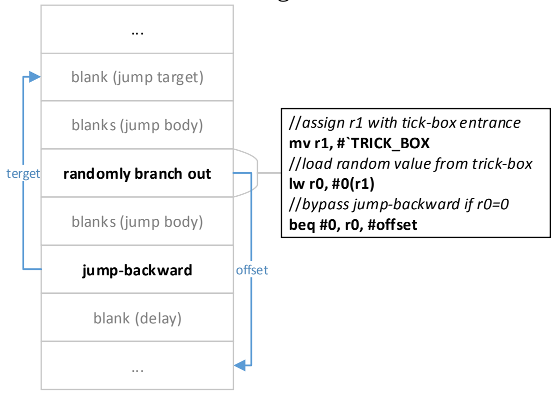
The instructions were generated in two phases.

Phase1: Generate the instruction framework of a sub-sequence, only randomize and generate key nodes (e.g., branch in Figure 4 which determines the redirect offset), leave other instructions (e.g., delay slot, branch body and target) undetermined as a "blank" instruction, but with determined PC. Update instruction constraint pool that imposed by the determined instruction.

Phase2: after returning from the control subsequence, main code sequence will execute-then-generate from the start of the subsequence, execute the determined instruction, and generate random instructions to replace those "blank" under proper context constraint.

With the above partitioning method, control subsequence concentrate on generating control instructions and redirect path, leaving all context-constrained randomization to main code sequence, thus making it very suitable to develop main code sequence and control-subsequence in parallel by multiple verification engineers. It is proved to be efficient both in our testbench development and testbench code maintenance.

Among different types of control subsequences, jump (or branch) backward subsequence is challenging. As shown in Figure 5. As jump backward is unconditional, the most obvious problem is that the program might be stuck in infinite backward if there is no path to bypass jump-backward. To avoid infinite backward jump, a piece of random branch-out instructions is introduced. It is consisted of instructions which load data from a reserved DM region called trick-box, and a conditional branch instruction to bypass jump-backward with the loaded trick-box value as branch condition. Little tricks are made on the trick-box by testbench: instead of return the original stored value, each time the trick-box is read, testbench manipulate the DM agent and DM mirror to return a new random value between 0 and 1. Thus making a "randomly branch out" and eliminating infinite backward.



```
//assign r1 with tick-box entrance
mv r1, #`TRICK_BOX
//load random value from trick-box
lw r0, #0(r1)
//bypass jump-backward if r0=0
beq #0, r0, #offset
```

Figure 5. Avoid infinite jump-backward

E. Layered sequence: control subsequence and do loop sequence

Similar to the jump or branch subsequence, the structure of a do loop sub-sequence is as shown in TABLE 3. It consists of three parts including head, do instruction and loop body. The head part is a bunch of instructions to determine the repeat time. One of the do instructions looks like: do ra, #uimm12.where ra represents the repeat time of this loop and #uimm12 represents the PC offset to the loop end instruction of the loop body.

TABLE 3
AN EXAMPLE OF DO LOOP SUBSEQUENCE

Assembly	**comment**
and ra,ra,0xff	limit repeat times
…	blank
do ra, #uimm12	
…	loop start: blank
…	blank
…	loop end: blank

In the most radical case, the max repetition number is 2^{42}. In order to make a balance between flexibility of randomization and efficiency, the number of the multiplication would be better to be limited. Thus an "and ra, ra, 0xff" instruction was put in TABLE 3 to limit up to 255 repeat times for do loop. As for the inter-instruction constraint, they are imposed as described in section C of chapter IV.

Surely, there is a verification requirement of "big loop" scenario, especially to cover each bit of ra and #uimm12 value. It requires large repeat times and PC offset. To cover these scenarios, 2 types of special testcases were created. One was with large repeat times, and the other with large offset. However, in order to balance between the verification target and simulation time, testcases with large repeat times were designed to limit imm12 to a relatively small range, vice versa for testcases with large offset.

F. Introducing data dependency

Hardware stall or data forwarding may happen between two pipeline stages if there is data dependency between the two instructions on them. Since the pipeline stages are usually occupied by instructions with consecutive PC(except for those control instructions), and all instructions are randomized for each PC, the

HW stall and forwarding functionality can be easily covered if data dependency exists between these consecutive instructions.

However, the probability of data dependency between to instructions is quite low: take two "add rt,ra,rb" for example, each with the probability of 1/16 to write R1, and thus the probability of WAW(Write after Write) over R1 will be 1/256. It is necessary to introduce additional methods to increase the probability of data dependency between instructions.

TABLE 4 shows a piece of instructions that need to introduce data dependency at PC=16. Instructions from PC=0 to 12 have been generated, whose read and write registers are recorded by each instruction object as shown in Figure 3. When it turns to PC=16, a "lw rt, #0(ra)" instruction is randomized, where rt can be any registers inside [R0:R15] while ra inside [R0:R7]. Take RAW(Read After Write) for example, to introduce data dependency between PC=16 and previous 4 instructions, ra shall be randomized inside {R2, R4, R10} as much as possible.

TABLE 4
INSTRUCTION COMBINATION WITH DATA DEPENDENCY

PC	Instruction	Read register	Write register
0	insn.(0)	R1	R2
4	insn.(1)	R3	R4
8	insn.(2)	R9	R10
16	"lw rt, #0(ra)"	ra	rt

To realize this, a queue array r_reg_candidate[$] that holds the register candidate was introduced in the instruction class, along with a function adjust_read_operand() which adjust the instruction's read register to be inside the candidate as much as possible.

```
// Code Example F1: register candidate and adjust operand
reg_e r_reg_candidate[$];
…
virtual function void adjust_read_operand();
  if (instruction == LW_RR) begin
    if  (!(ra inside {r_reg_candidate})) begin
      std::randomize(ra) with {
                                          ra inside {[R0:R7]};                //intrinsic hard constraint
                                          soft ra inside {r_reg_candidate}; //additional soft constraint
                                         };
    end
  end
endfunction
```

It is noted that soft constraint was used in adjust_read_operand() function, because not all the r_reg_candidate might comply with the intrinsic operand constraints of "ra inside {[R0:R7]}", which is imposed by ISA and regared as "hard constraint" . Take the example in TABLE 4 for example, if the write registers of previous instruction were {[R8, R9, R10]} instead of {[R2, R4, R10]}, the instruction generation would rather give up data dependency than violate the intrinsic constraints. In other words, the intrinsic constraints take precedence over those introducing data dependency, and the "soft constraint" in SystemVerilog is born to cope with this situation.

V. Handling with Interrupt, Result Check and Coverage

A. Instruction execution in correspondence with DUT

As described in section A of chapter IV, testbench mainly focuses on the "final" execution result of each instruction instead of its behavior on each pipeline stage. Two facts made it unavoidable to check the instruction execution order is as expected.

The first is there are dozens of control instructions in ISA such as unconditional jump and conditional branch which cause PC to redirect. In fact, most of these instructions are designed to redirect instruction without writing registers or DM. So, after the instruction executed in our executor, it is necessary to check if instruction execution in DUT has been changed as expected.

The second is that, after the generated instruction put into PM and DUT works, there is a situation which testbench shall handle with: PC may redirect to a corresponding ISR entry every time after external IRQ introduced at a random time, but testbench cannot exactly know after which PC DUT will redirect to ISR.

Because of the above two facts, the executor was designed that it follows the PC change of DUT, and checks every PC change is as expected, while the PC change is known by monitoring DUT internal signals.

As for the unpredictable IRQ timing, testbench make a trade-off: whenever the instruction executor find unexpected PC redirection, it checks if the target PC is an ISR entry before it reports error, as described in Figure 6.

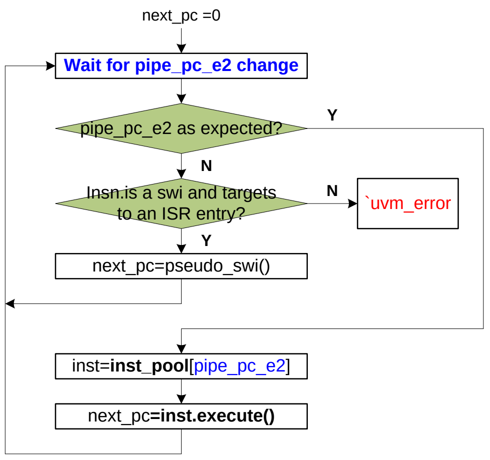

next_pc =0
Wait for pipe_pc_e2 change
pipe_pc_e2 as expected?
Y
N
Insn.is a swi and targets to an ISR entry?
N
`uvm_error
Y
next_pc=pseudo_swi()
inst=inst_pool[pipe_pc_e2]
next_pc=inst.execute()

Figure 6. Reference instruction execution

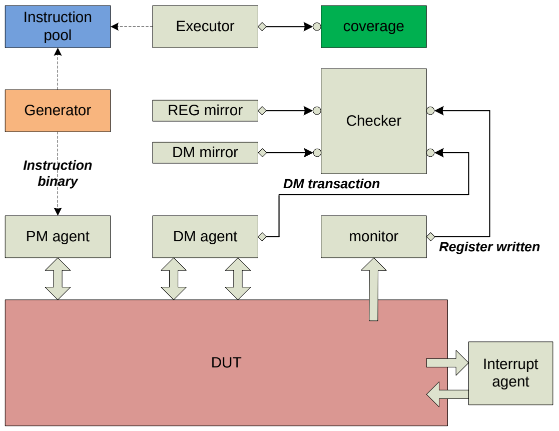
Another problem arises with the random IRQ is that instructions in ISR may modify register values, and thus the register values may be different from that when the main code were generated. To solve this problem, we design the ISR so that it is transparent to the main code without modifying register values and affecting main code execution. The design of ISR will be described in next section.

B. Handling with random interrupts

Random IRQ is necessary to be introduced for the following reasons:

1) DUT is designed to support that, program might be externally interrupted and continue after ISR;

2) ISA includes special instruction, halt, which stops program running and can only be waked up by external interrupt;



An “interrupt agent” as shown in Figure 2, was introduced to generate IRQ at random timing when DUT executes the main code instruction.

Except for ISR0, in which instructions are basically fixed to initialize HW after power-on reset, the other ISRs are filled with random instructions. The biggest difference of ISR sequence from main code sequence is that ISR is not randomized in an execute-then-generate way, because the interrupt and execution of ISR cannot be predicted when generating the main code. To avoid GPR values being changed by ISR, which may make the main code execution different from when it was generated, ISR sequence were designed as typical ISR, starting with push instructions to preserve all the register would be written by ISR and ending up with corresponding pop instructions to restore them, as shown in Figure 7.

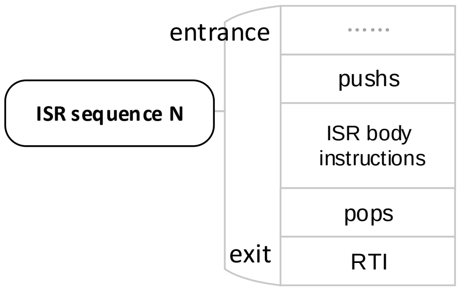


Figure 7. ISR sequence

Although push and pop instructions protect GPR from being polluted, the push instruction, which writes the register value to stack region of DM, still pollutes the stack. Take Figure 8 as an example. When the main code was first generated, the iow instruction executed normally because the lw instruction loaded a value to r1 from stack which made r1+100 a legal IO write target address. However, after the main code was put into PM and was executed by DUT, a random IRQ interrupted the main code to ISR, where the original stack was overridden by push instructions. After returning from ISR, main code continued to load data from specific address in stack, got the value changed by push. That changed value might cause the iow instruction to write at an illegal IO address, which might lead to unexpected error.

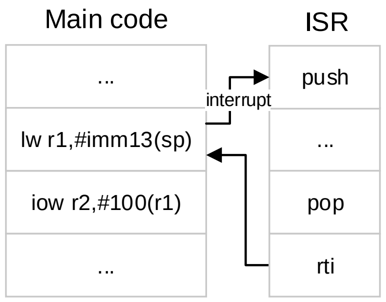


Figure 8. Entry and exit of ISR

Since all this unexpected value changes happen in stack, a “virtual stack” in DM agent was introduced to solve this issue. It was designed as a copy of real stack. Whenever testbench detects the main code turning to an ISR, as described in section A of chapter V, the virtual stack is turned on to let all push/pop instructions operate on it. After the main code returns to main code from ISR, the virtual stack is turned off. By doing this, any unexpected changes by IRQ became transparent to instructions in main code.

C. Checking result properly

As described in section A of chapter V, the DUT’s PC value change is compared to the reference model to make sure DUT’s execution order is correct. Apart from PC value check, each instruction’s execution result, for example, writing a specific GPR, or reading a specific DM address, need to be checked for correctness verification. Except for writing register, most of instructions read register values, but unlike memory, reading is implicit without a signal transaction can be traced. So, the following 3 types of result was checked by the testbench.

1) Register written transaction
2) DM read transaction
3) DM written transaction

When checking register written transaction, it was very straightforward to let reference model executed instruction one by one and send each register written transaction to one side of checker, as shown in

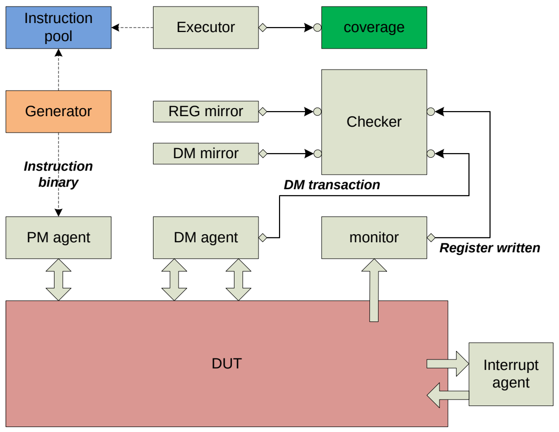


Figure 2. The transaction arrived to checker will look like following order:

PC=0: write R1=1;
PC=4: write R2=2;
PC=8: write R3=3;
PC=12: write R1=4;

On the other side of checker, a monitor was needed to monitor each register's written transaction. Due to the pipeline operation of instruction and other HW design, monitored transaction may not be exactly the same, and 2 major problems arose. The first was that R3 may be written earlier R2 in time. Secondly, DUT's register write had no information of PC indicating that which PC issued the writing, which was the same story for DM read and write. As a result, the transaction provided by monitor looked like:

Write R1=1;
Write R3=3;
Write R2=2;
Write R1=4;

To cope with the above situation, the checker was designed in a multi-stream way as shown in Figure 9: the register written transaction received was dispatched to dozens of "streams" according to their register index, with each stream independently compared register transaction from both sides in order. This makes the check insensitive to the DUT's pipeline timing while keep the logical order of the reference model.

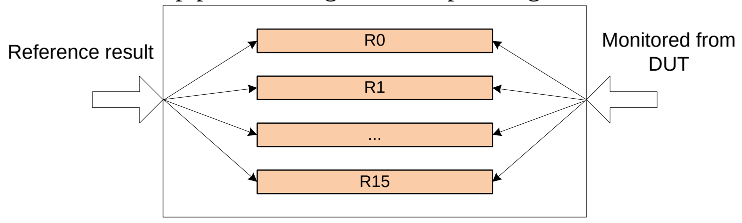


Figure 9. Multi-stream check for GPR result

As for the absence of PC value in the DUT monitored register transaction, the check was designed to keep the PC value but skip PC value comparison, which was very simple to be realized using a "UVM_NOCOMPARE" in UVM field macro when specifying the transaction class.

The kept PC value was useful when printed in log file, because when a mismatch happened, it indicated on which PC the mismatch happened, which helped us a lot during debug.

D. Coverage granularity and priority

Because of the large number of instructions and its combinations, coverage metric became very important in this testbench. Ideally, it would be good if all combinations of operands, instructions, pipeline stages were covered, but that would lead to huge number of coverage bins. Take data forwarding for example, roughly 150000 combinations of instructions are able to cause data forwarding in pipeline, and if operand variation were included, for example, a GPR operand rt could be any one of R0~R15, the resulting coverage bins would be 16 times more. It would be inefficient if all types of functionality were covered within the same granularity, especially when a tight task schedule was there. We proposed several solutions to cope with situation.

Firstly, different granularities were used in different coverage. For example, operand level coverage is applied for single instruction like “add rt,ra,rb”. When it comes to covering all possible instructions in a delay slot, a coarser granularity of instruction level is used. As for the above example of data forwarding, its coverage is defined and sampled in instruction type level whose number of combinations reduced to 9000, since the data forwarding path is less coupled with operand, and instructions of the same type usually has the same pipeline operation. In particular coverage items, a mixed granularity was adopted: some instructions are covered in instruction type level while some in instruction level, because design engineers suggests that different instructions of the same type behaves slightly different in affecting the functionality to be covered.

Secondly, coverage priority was decided in our task execution stage, to catch up with the project schedule. As for this task, the control-path related coverage such as HW stall and data forwarding was covered first, while data-path related coverage such as operand coverage was sampled and tuned at the last phase of our regression.

It shall be noted that all the solutions including trade-offs and priority decision was initiated by verification engineer, discussed with and feedback by design engineers, who clearly has more insightful view on certain HW design logic particularly in control-path like pipeline stage, HW stall, and forwarding. Cooperative participation of design engineers in verification plan and coverage convergence improved quality and schedule of this verification task.

VI. Conclusion and summary

Based on the testbench introduced, we had run about 80000 random testcases to achieve 100% functional coverage, for which more than 700 covergroups were defined. An average of 1500 main code instructions was randomized for each testcase, with less than 2 minutes CPU run time.

In general, we think the testbench provides good reference for verification of similar microprocessors with a large number of instructions, variety of operands, especially with much intra-instruction and inter-instruction constraints. The concept of instruction constraint pool and soft constraint in our testbench is a good option to randomly introducing data dependency. With the help of register mirror, the idea of execute-then-generate can be used avoid unintended scenarios. In addition, the categorization and abstraction of instruction classes, the coverage priority offers some inspiration for speeding up testbench development and coverage convergence in real-world projects.

References

[1] David A. Patterson, John L. Hennessy, “Computer Organization and Design”, Elsevier, 2014.

[2] John L. Hennessy, David A. Patterson, “Computer Architecture: A Quantitative Approach”, Elsevier, 2012.

[3] Accellera, UVM User Guide, v1.1, www.uvmworld.org

[4] IEEE Std 1800-2012 for SystemVerilog, IEEE, 2013