UVM testbench design for ISA functional verification of a microprocessor

Gabriel Wang, MediaTek (Beijing) Inc.
Hongtao Ma, MediaTek (Beijing) Inc.
Maoduo Sun, MediaTek (Beijing) Inc.
Verify “add rt,ra,rb”

- **Functional correctness**
  - Check: rt = ra + rb

- **Operand completeness**
  - Coverage
    - rt: R0~R15
    - ra/rb: R0~R15

- **Instruction combination**
  - Instruction sequence
Abstraction and categorization

- **Abstraction**
  - Instruction $\rightarrow$ UVM sequence item
  - Multiple instructions $\rightarrow$ UVM sequence

- **Categorization**
  - Mapping 300+ instructions into 40 UVM classes

### Table: Instruction and Operation

<table>
<thead>
<tr>
<th>Instruction</th>
<th>operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>add rt, ra, #imm16</td>
<td>rt = ra + #imm16</td>
</tr>
<tr>
<td>add rt, ra, rb</td>
<td>rt = ra + rb</td>
</tr>
<tr>
<td>addx rt, ra, rb</td>
<td>rt = ra + rb + mc</td>
</tr>
<tr>
<td>add rt, ra, rb, #1</td>
<td>rt = ra + (rb &lt;&lt; 1)</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

**Addition operation**

- read src reg@E1, write dst reg@E2
Make random instructions predictable

• **Execute-then-generate**
  
  – **“blind randomization”**: register value is not referenced

<table>
<thead>
<tr>
<th>PC</th>
<th>instruction</th>
<th>operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>8</td>
<td>sub r1,r2,r8</td>
<td>r1=r2-r8</td>
</tr>
<tr>
<td>12</td>
<td>or r3,r4,r5</td>
<td>r3 = r4</td>
</tr>
<tr>
<td>16</td>
<td>lw r13,#0(r1)</td>
<td>r13 = DM[r1]</td>
</tr>
<tr>
<td>20</td>
<td>j r1</td>
<td>jump to pc=r1</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

if r1 value too large, e.g. 0xFFFF
May cause to access illegal memory region

if r1 value too small, e.g.12
May cause infinite backward jump

– **“execute-then-generate”**

  • All instructions are first “executed” during random instruction generation

<table>
<thead>
<tr>
<th>PC</th>
<th>instruction</th>
<th>operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>8</td>
<td>sub r1,r2,r8</td>
<td>r1=r2-r8</td>
</tr>
<tr>
<td>12</td>
<td>or r3,r4,r5</td>
<td>r3 = r4</td>
</tr>
<tr>
<td>16</td>
<td>lw r13,#0(r2)</td>
<td>r13 = DM[r2]</td>
</tr>
<tr>
<td>20</td>
<td>sw r1, #4(r3)</td>
<td>DM[r3+4]=r1</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Knowing all register/DM value, testbench can avoid randomizing unintended instructions
Layered sequence

• “main code” sequence and control sub-sequence

- **“main code” sequence, Random generate**
  - Plain instruction: no PC redirection
  - Control subsequence: may cause PC redirection
  - “Execute-then-generate” flow

- **Random subsequence(conditional branch)**
  - Generate control instruction Skelton
  - Leave non-control part as blank
  - let main code sequence random fill it

- **Random subsequence(unconditional jump)**
  - Take care to avoid infinite backward jump
Avoid infinite backward jump

• Control sub-sequence example
  – Backdoor manipulation of certain reserved DM address
    • → no infinite jump: jump instruction will be randomly skipped

```plaintext
//load random value from trick-box
mv r1, #TRICK_BOX
lw r0, #0(r1)
//bypass jump-backward if r0=0
beq #0, r0, #offset
```
Bench structure

- Functional correctness
Instruction Constraint

- **Intra-instruction constraint**
  - **Operand/value constraint**
  - implemented in instruction class

- **Inter-instruction constraint**
  - **Dynamically introduced by context**

```plaintext
// 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
        } 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});
        }
    }
endclass
```

<table>
<thead>
<tr>
<th>PC</th>
<th>instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>add r2,r4,r6</td>
</tr>
<tr>
<td>4</td>
<td>...</td>
</tr>
<tr>
<td>8</td>
<td>...</td>
</tr>
<tr>
<td>12</td>
<td>mv r8, #0x8000 //set target address</td>
</tr>
<tr>
<td>16</td>
<td>...</td>
</tr>
<tr>
<td>20</td>
<td>j.d2 r8</td>
</tr>
<tr>
<td>24</td>
<td>...</td>
</tr>
<tr>
<td>28</td>
<td>...</td>
</tr>
</tbody>
</table>

To protect target address: PC=16 shall not write r8

Delay slot limitation: division instruction should not be present in PC=24,28
Inter-instruction constraint

• Modeled as an “instruction constraint” for each PC
  – Dynamically updated by instruction sequence
  – Constraint modeled as
    • exclude types, instructions, w_regs, ...
  – Put into a instruction constraint pool

// Code Example C1: instruction constraint class
class instruction_constraint extends uvm_object;
  bit [31:0] pc;
  instruction_type_e exclude_type[$];
  instruction_e exclude_inst[$];
  reg_e exclude_r_regs[$];
  reg_e exclude_w_regs[$];
endclass

typedef uvm_pool #(bit[31:0], instruction_base) instruction_pool;
typedef uvm_pool #(bit[31:0], instruction_constraint) instruction_constraint_pool;
Introducing data dependency

- **soft** constraint in `adjust_read_operand()`
  - “TRY-TO” but not a MUST

<table>
<thead>
<tr>
<th>PC</th>
<th>Instruction</th>
<th>Read register</th>
<th>Write register</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>insn_a</td>
<td>R1</td>
<td>R2</td>
</tr>
<tr>
<td>4</td>
<td>insn_b</td>
<td>R3</td>
<td>R4</td>
</tr>
<tr>
<td>8</td>
<td>insn_c</td>
<td>R9</td>
<td>R10</td>
</tr>
<tr>
<td>16</td>
<td>“lw rt, #0(ra)”</td>
<td>ra</td>
<td>rt</td>
</tr>
</tbody>
</table>

To randomly induce RAW data dependency:
Let ra inside \{R2,R4,R10\}

```cpp
class ldst_ld extends instruction_base;
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]\};
      soft ra inside \{r_reg_candidate\}; //additional soft constraint
    }
  end
end
endfunction
...
endclass
```

Intrinsic constraint may conflict with data dependency intension!

→ soft constraint
Handling with random interrupt

- ISR instructions modeled as isr sub sequence
  - The only sequence not be executed during instruction generation
- push+pop provides basic protection of GPR
- How to protect DM?
  - To makes “execute-then-generate” flow effective

What if #imm13 happens to be 0?

First push instruction override 0x8000

Let ISR’s push/pop operates on a “virtual stack”
Result check

• **Timing gap between reference and DUT**

Reference model

<table>
<thead>
<tr>
<th>PC</th>
<th>insn_a</th>
<th>PC=0: write R1=1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>insn_a</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>insn_b</td>
<td>PC=4: write R2=2</td>
</tr>
<tr>
<td>8</td>
<td>insn_c</td>
<td>PC=8: write R3=3</td>
</tr>
<tr>
<td>12</td>
<td>insn_d</td>
<td>PC=12: write R1=4</td>
</tr>
</tbody>
</table>

DUT

<table>
<thead>
<tr>
<th>insn_a</th>
<th>ID</th>
<th>E1</th>
<th>E2</th>
<th>E3</th>
</tr>
</thead>
<tbody>
<tr>
<td>insn_a</td>
<td>ID</td>
<td>E1</td>
<td>E2</td>
<td>E3</td>
</tr>
<tr>
<td>insn_b</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>insn_c</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>insn_d</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

• **Compare in a multi-stream way**

Reference result

Monitored from DUT