A UVM Based Methodology for Processor Verification

Abhineet Bhojak, Tejbal Prasad
Freescale Semiconductor
Processor Verification Challenges

• Presence of multicycle instruction along with single cycle instruction
  – Various hazards (e.g. WAR, WAW)
• Dedicated Hardware Accelerator
• Excessive number of GPR (~200)
• Debug hooks for easier debugging

Stimuli Generator
- Hazard scenario
- Accelerator
- Jump & Loop cmd

Test generation
- Meaningful program
- Need of multiple test

Debug
- Hooks for localization of failure
Existing Technologies

• Random test pattern generation (RTPG) and Test plan automation
  • Combine architecture specific knowledge and testing knowledge and uses a CSP solver to generate efficient test programs
  • there is a significant learning curve involved to leverage these RTPG’s in an industrial environment

• Formal verification
  • Useful and efficient in some cases
  • it requires significant mathematics skill and computational resources to relate to the scenarios and analyze them

• Pure directed testing
  • Gives confidence on different functionalities
  • Achieving desired coverage may take large amount of time
PROPOSED METHODOLOGY

Layered architecture

- Scenario
- Program Generator
- Atomic Transaction
Proposed Stimulus Generation Flow

• A fine blend of Top Down control and Bottom layer intelligence

Scenario
• Skeleton of the program is generated
• Size of the program is controlled
• Data for the program is controlled
• Provides the Top Down Control

Program Generator
• It does the decision making based on Scenario level information
• It Randomizes atomic transaction based on fixed/random type or to a fixed/random instruction

Atomic Transaction
• It randomizes all the fields and pack them into one instruction.
• Bottom layer intelligence
  • Takes care of infinite loop
  • Does instruction operand interlinking
• Extension for instruction grouping for better reuse
Bottom Layer Intelligence

load address1, R3
load address2, R4
add R3, R4, R5
store R5, address3

Potential Hazard Candidate

Interesting scenario

Conventional Way
Randomize the whole program
Use foreach constraint to make relation between instruction operand

Innovative Way
Randomize one instruction at a time.
Keep copy of last few instruction.
While randomizing current instruction 1st decide to what depth (rel_depth) you want to link it
Use last instruction copy & rel_depth to decide the current instruction operand

How to put in this intelligence w/o complicating constraint solver?
Bottom Layer Intelligence

```
class processor_transaction_atomic extends uvm_sequence_item;

  // represent instruction groups
  rand processorMasterCommandT master_cmd;
  // represents individual instructions
  rand processorCommandT processor_command_type;

  // Variables for construction the final instruction
  bit [31:0] processor_instruction;
  rand bit [7:0] processor_input_operand_1;
  rand bit [7:0] processor_input_operand_2;
  rand bit [7:0] processor_destination_address;

constraint processor_master_command_decoding {
  (master_cmd) inside
    { /*define the valid instruction groups here */};
  (master_cmd == MASTER_ADD) ->
    processor_command_type == PROCESSOR_CMD_S_ADD;
  (master_cmd == SCALAR_COMMANDS) ->
    {processor_command_type inside
      {PROCESSOR_CMD_S_MOV,PROCESSOR_CMD_S_ADD}};
}

constraint processor_operand_interlinking {
  (dest_op_relation) inside { INDEP, DEPO, DEPI }
  (dest_op_relation == INDEP) -> {};
  (dest_op_relation == DEPO) ->
    {processor_destination_address == last_operand_type};
  (dest_op_relation == DEPI) ->
    {processor_destination_address == 2nd_last_operand_type};
}

function generate_instruction_data();
  if(processor_command_type == PROCESSOR_CMD_S_ADD)
    begin
      processor_instruction[23:16] = processor_destination_address;
      processor_instruction[15:8] = processor_input_operand_1;
      processor_instruction[7:0] = processor_input_operand_2;
    end
  endfunction : generate_instruction_data
endclass : processor_transaction_atomic
```
Bottom layer Intelligence

INSTRUCTION GROUPS
- BRANCH
  - BEQ
  - BLT
  - BGT
  - BN
  - BC

<table>
<thead>
<tr>
<th>ALU</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD</td>
</tr>
<tr>
<td>SUB</td>
</tr>
<tr>
<td>XOR</td>
</tr>
<tr>
<td>OR</td>
</tr>
<tr>
<td>CMP</td>
</tr>
</tbody>
</table>

ALU
- ADD
- SUB
- XOR
- OR
- CMP

BRANCH
- BEQ
- BLT
- BGT
- BN
- BC

FILL

Jump Length constraints
- LEN1
- LEN2
- W-MARK
- LENGT H

W- MARK

Infinite Loop Avoidance

Loop:
- MOV R1, VAL
- JMP LEN1
- SUB R1, 1
- BZ End
End:
- JMP LEN2
- SUB R1, 1
- BNZ Loop
- HALT
Debugging Hooks

- Zero time Reference model Vs pipelined processor
- Out-of-order execution of pipeline Vs In-order execution of model
- Debug of Debug
- File & queue based Checker

Debug cannot be an afterthought.

- Checking only at interfaces is not enough for complex scenarios
- Need checker based on register content change – Data trace checker

Localization of Failure

- At some point in time we need to run programs developed by other team
- Switch based flow for directed stimulus

Scenario Replication
Debugging Hooks

Diagram showing the components and flow of a debugging system, including:
- Scoreboard
- Debug Logic Checker
- Interface level checker
- Data trace checker
- Interface monitor
- Register data sequence monitor
- Stimuli Generator
- Register trace file
- Golden output file
- Memory Model
- DUV
- Reference Model
- Input Program
- Register Programming sequence
- Input data
- Backdoor loading
Evaluation of the Proposed Flow

Design Complexity
Scalar, Vector & Matrix operation, 9 ALUs, 4 Multiplier, ~256 GPRs & Hardware Accelerator like SORT, HISTOGRAM etc

Verification
30 man weeks of effort, Verification Environment created from Scratch, ~200 test /15 K runs, ~10 k functional cover points, 200 odd defects were found

First Pass Success
No additional bugs found after IP signoff. Silicon has been evaluated - considered to be a first pass success.
Thank You
Q & A