Automated Test Generation to Verify IP Modified for System Level Power Management

Christophe Lamard, STMicroelectronics
Frederic Dupuis, Cadence
GPU Power optimization

- We integrate internal and third party GPU IP
  - Replace generic Macro Block according to the technology
  - Tune power capability
  - Split hierarchy design according to layout team request
  - Adjust DFT structure
GPU power optimization

Optimized GPU Dynamic Power Profile
Dynamic Power verification

• New bug types comes with power management
  – Missing Isolation bugs
  – Control Sequencing bugs
  – Retention scheme errors
  – Memory corruption
  – Power sequence scheduling errors
  – Software/Hardware dead lock
  – Power On Reset bugs
  – ....

• Verification team need to manage dynamic power simulation:
  – Create Power test sequence
  – Run dynamic power simulation
  – Add power checker
Sub-system Integration

- We deliver our test to SOC, Validation and SW team
  - need for portable tests (not UVM)
  - Tests are written in ‘C’
- We have created:
  - A test bench API to control:
    - IP Powers state
    - IP top signal like clamp, reset, clock
  - An IP API to control IP power sequence
    - Retention sequence
    - Clock, reset and clamp control
  - Some Power monitor/checker
Closer look at tests requirements

Let's consider 2 power switch A and B.

- 3 different possible values: nom, overdrive1, overdrive2
- They are linked by a rule:

  value power switch B \leq value power switch A

**Blue triangle** = initial state
**Red Square** = transition state
**Green points** = functional state

**Gray line** = rule
Transitions below the line are allowed.
Defining one test

- A test is always:
  1. Put the power logic in a functional state
  2. Run a functional tests on the IP

**Functional state:** Power switch A = OV2 and Power Switch B = OV1
User need to define transition from initial state to this functional state
Defining one test (cont)

- **Additional power elements** must be configured to be in a functional state:
  - **Clock** for power domain A and B
  - **Reset and Clamp** for A and B

- **Different rules** also exist for these elements:
  - Can be configured when power switch is not off
  - Some clock frequency cannot be used with some power switch value

- **Some more complex tests scenario are needed:**
  - Go in a functional state,
  - Switch off the power
  - Go in a different functional state
Manually developed C code

- Here is one C code example to check this functional state:

```c
turn_on_power_switchA();
turn_on_power_switchB();
setup_clamp_reset_powerB();
setup_clock_powerB(freq0)
setup_clamp_reset_powerA();
setup_clock_powerA(freq1) ;
change_power_switchA(ov1);
change_power_switchB(ov1);
change_power_switchA(ov2);
run_functional_test();
```

- Obviously there are much more possible tests:
  - Change the path (different transitions)
  - Change clock frequency, change setup time of clamp/reset
Need for automation

• On this simple example we would need at least 6 tests (for each functional state)
• In our GPU design we have:
  – More power switch (up to 6)
  – More complex rules
  – Specification may change during project
• Developing a test requires **deep knowledge** of power spec
• Not possible to **create and maintain** all needed tests.
• We have developed **20** tests (targeting 20 states)
  – Most of the time same path is used (extension of previous test)
  – Other power elements often configured same time
• **Need for automation** to create tests for all possible state
Defining new methodology using automation tool

- Main contribution of the automation tool are
  - Model based approach
    simple and abstract way to define and constrain the power elements
  - Goal directed test creation
    Thanks to the use case based solver (describe what, not how). It means describing expected power state, not the transitions to reach it.
  - Automated test generation
    Simple way to achieve 100% coverage of specified goals. Goals here would be the complete list of functional state
Model based approach

• Do not describe path from initial state to functional state

• But:
  – Describe all power elements and their possible values
  – Define all possible transitions and their relations with other power elements
  – Map to each transition the associated API call

• Force to have a systematic description and completely understand the power specification
Describe all power elements

- If we use power switch B of our previous example
  - Possible values: off, nominal, overdrive1, overdrive2
  - List of transition, associated rules and API

<table>
<thead>
<tr>
<th>Transition</th>
<th>Rules</th>
<th>Associated API</th>
</tr>
</thead>
<tbody>
<tr>
<td>off_to_nom</td>
<td>pswitchA != off</td>
<td>turn_on_power_switchB()</td>
</tr>
<tr>
<td>nom_to_off</td>
<td>pswitchA != off (*)</td>
<td>turn_off_power_switchB()</td>
</tr>
<tr>
<td>nom_to_ov1</td>
<td>pswitchA &gt; nom</td>
<td>change_power_switchB(ov1);</td>
</tr>
<tr>
<td>ov1_to_nom</td>
<td>pswitchA &gt; nom (*)</td>
<td>change_power_switchB(nom);</td>
</tr>
<tr>
<td>ov1_to_ov2</td>
<td>pswitchA == ov2</td>
<td>change_power_switchB(ov2);</td>
</tr>
<tr>
<td>ov2_to_ov1</td>
<td>pswitchA == ov2 (*)</td>
<td>change_power_switchB(ov1);</td>
</tr>
</tbody>
</table>

(*) implicit rules (already put for power switch A) but added for clarity and debug purposes
Goal directed test creation

• User can define a test in a goal directed way.
• Define what (the expected functional state) and not how (the path to reach it)

  Power switch B == OV1 and Power switch A == OV2

• Tool will automatically find a path from initial state to this state.

• Each generation may create a different path.
• It is not possible to create a test that contradicts the rules

  Power switch B == OV2 and Power switch A == OV1 (illegal state)
  – Tool will report an error.
  – No time spent on trying to run/debug a wrong test
Automated test generation

• User can also request tests in all possible functional state:

  Power switch B == “all value” and Power switch A == “all value”

• Only legal tests will be created (following the rules)
  – No need to know all of them, the tool will find them
  – Tool will also report the non valid case, useful for debug

• In our case, tool has been able to create 192 tests, all reaching a different functional state.
  – Different path have been used and the different power elements have been configured at different point of time
  – Coverage of all possible path might be possible too but this was not our main requirement
Pros/Cons of the methods

• Pros :
  – Enable to divide power specification and focus piece by piece
  – Create a kind of embedded power specification
    • Usable and readable by anyone
  – Changes to power specification could easily be reported
    • Add/remove a transition, add/remove/change rules
  – Goal directed test creation is very efficient (develop a new test in seconds)

• Cons :
  – Extra work for first tests.
  – Need to develop a new model and learn a new language
  – Model has to be exhaustive
Choice of automation tool: Perspec System Verifier

- We developed and used this methodology using **Perspec System Verifier**:
  - **Model based** approach of the tool
  - **Graphical representation** of the generated test
    - UML activity diagram showing all transition from initial to final state
  - Generated C test is linear and readable
    - Simplify debug
- First model was developed with local AE support
- No need for deep knowledge of the language
  - Learning curve is in days
  - Not the complexity of UVM for example
Goal directed test creation

- User can ask the tool to create a test in a specific functional state:
Goal directed test creation

Scenario is partially represented

Tool identified a path, and created a scenario with all the transitions from initial state up to final state

The graphical representation also enables:
- Quick analysis of the solution, and identification of model bugs
- Exchange /discuss with other stakeholders like design and architects.
- Simplify debug analysis of failing tests
Automatic test creation

• User can ask the tool to create a test in all specific functional state: 192 tests in our case
Results

- **Higher** coverage in **less time** than manual tests development
  - All 192 generated tests are different and cover all states
  - Covering transition we did not think off
  - Estimated manual effort to reach same coverage: 192 days

<table>
<thead>
<tr>
<th></th>
<th>Nb tests</th>
<th>Lines of code</th>
<th>Development</th>
<th>Maintenance (each change)</th>
<th>Nb tests /Day</th>
<th>Nb test/day in case of 5 changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Manual</td>
<td>20</td>
<td>2k (100x20)</td>
<td>20 days</td>
<td>3 to 4 days</td>
<td>1</td>
<td>0.57</td>
</tr>
<tr>
<td>Perspec</td>
<td>192</td>
<td>800</td>
<td>10 days</td>
<td>1 day</td>
<td>19.2</td>
<td>12.8</td>
</tr>
<tr>
<td>Ratio</td>
<td><strong>0.4</strong></td>
<td></td>
<td></td>
<td></td>
<td><strong>19.2</strong></td>
<td><strong>22.4</strong></td>
</tr>
</tbody>
</table>
Conclusion and future works

- Coverage confirmed by PST coverage during simulation
- Identified bugs in:
  - Embedded design checkers
  - UPF file (missing/unreachable states)
- Increase confidence in power sequence supports
- Methodology could be applied to any LP verification

Future works:
- Reuse model at SOC level to create system level LP tests
- Deploy on even more complex IPs
- Model based enable to combine LP and functional tests