DVCon 2012

Design & Verification Conference & Exhibition

February 28 – March 1, 2012

Supplementing Simulation of a Microcontroller Flash Memory Subsystem with Formal Verification

Othmane Bahlous

Senior Expert – Design, Verif. and RTL2GDS

infineon

Abdel Ayari

Design & Verification Solutions AE

Roger Sabbagh

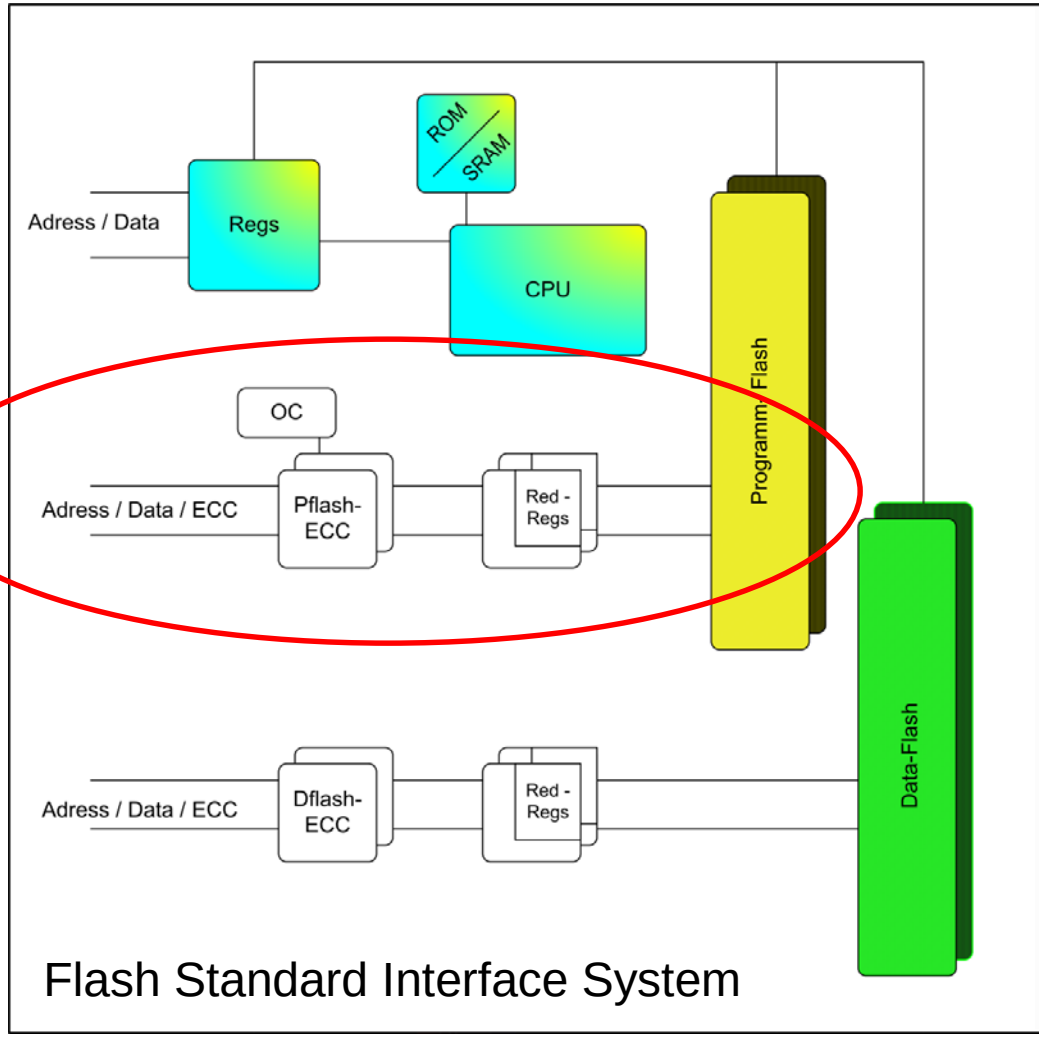
Questa Formal Product Manager

Mentor Graphics®

Outline

- Design Under Verification
- Error Correction Circuits
- Bitline Redundancy
- Integration of Formal and Simulation Results
- Conclusions and Outlook

Design Under Verification



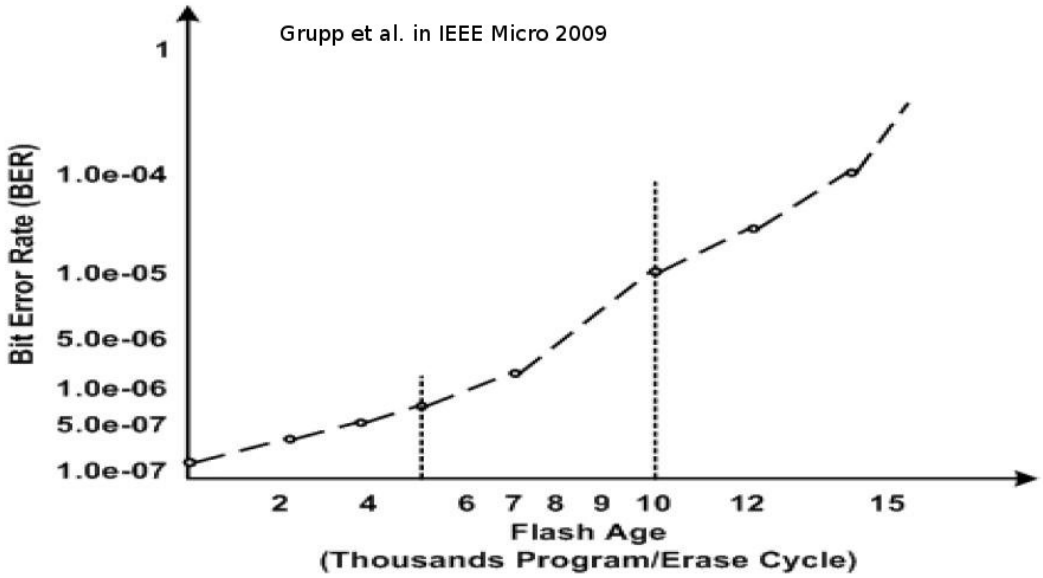
- Advanced Simulation Methodology
  - Verification plan
  - UVM testbench
  - Track coverage with UCDB
- Some functions not possible to cover completely
  - Error Correction Codes
  - Redundant Bit Line Scrambling

Outline

- Design Under Verification
- Error Correction Circuits
- Bitline Redundancy
- Integration of Formal and Simulation Results
- Conclusions and Outlook

Error Correction Circuits Description

- Detect and correct errors introduced during storage or transmission of data
  - Expect minor data corruption in Flash



- Two operation modes
  - Generation mode: computes the error detection code
    - ECC stored in Flash during write operation
  - Correction mode: use the ECC to detect and correct the errored bits
    - ECC checked during read operation

Error Correction Circuits Verification

- Simulation of ECC
  - Low coverage due to state space complexity
    - High data bit width: >200 bits leads to $1.6x10^{60}$ combinations
    - Variation in number of bit errors: 1-bit, 2-bit, etc.
    - Variation in location of bit errors
      - Single bit errors: >200 possible locations
      - 2-bit errors: k-permutations of $\mathrm{n} = \frac{n!}{2(n-k)!}$
        - » For n=200, k=2, #permutations ~ 20k
      - Etc.
  - Flash operations are very time consuming
- Goals of Formal Verification
  - Exhaustive analysis ECC circuits

ECC: Generation Mode

- Correct Generation of ECC Code

```
ref_ecc_code = Ref_Matrix  *  data
```

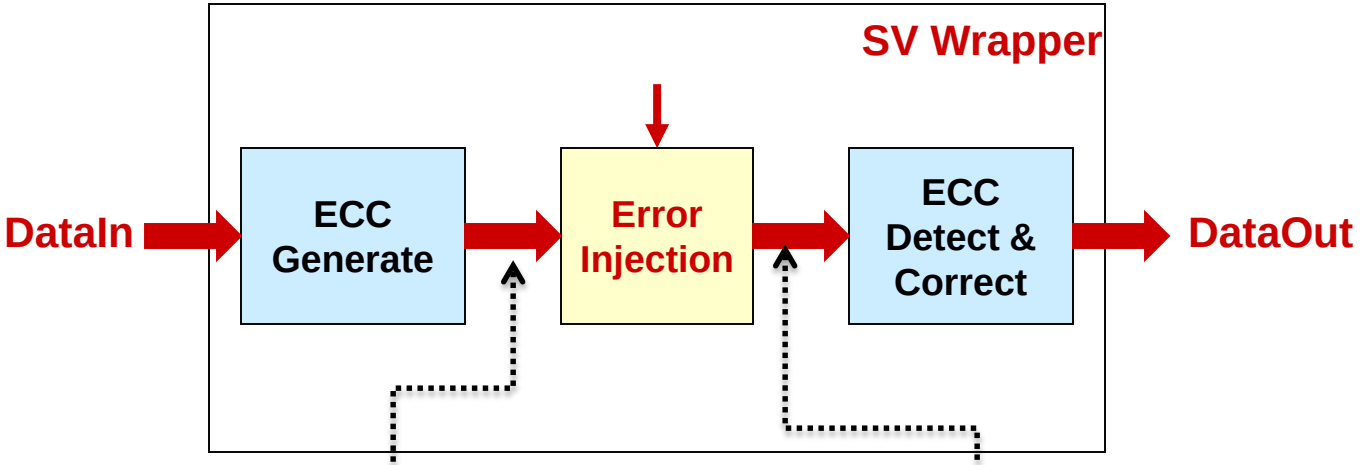
- Correct Bit Flags and Input Data is forwarded unchanged

```
//Check Error Flags
integer i;
  for (i=0; i <= c; i++) begin
    check_error_flag_is_low : assert property  (! error_flag[i]);
  end
//Check ECC Code and Data I/O
check_correct_ecc_gen :   assert property (
                                        ecc_code == ref_ecc_code);
```

- **All verification requirements are captured as SVA and proven valid within few seconds**

ECC: Correction Mode

- Modeling of Error Injection




```
assign data_ecc_code = {data, ecc_code};
```

```
assign buggy_data_ecc_code  = {buggy_data, buggy_ecc_code};
```

ECC: Correction Mode

- Requirements
  - Detect the errored bits and correct them
  - Generate the correct errored bit flags
- ECC with so many combinations is difficult for formal
  - Decomposition in two directions
    - Number of bit errors

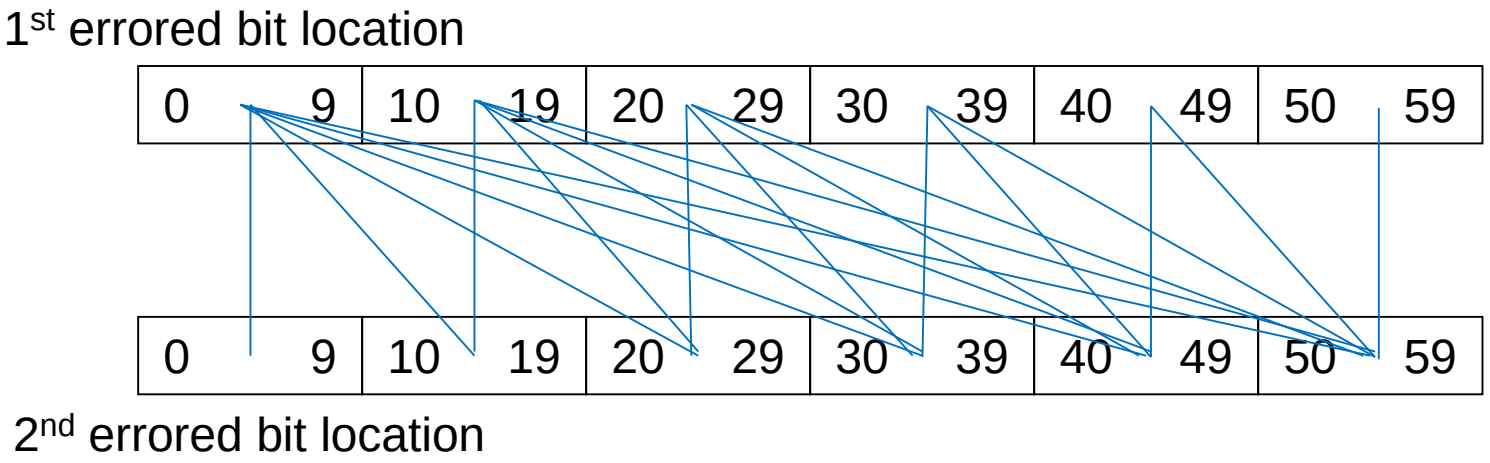
```
assume property (
          $countones(data_ecc_code xor buggy_data_ecc_code) <= n);
```

    - Position of bit errors

```
//Case 2 Bit Errors
for (p=0; p++; p < DATA_WIDTH)
 begin
  assign buggy_data_ecc_code[p] =  (p == pos_1 || p == pos_2)?
                                        not data_ecc_code[p]
                                   :   data_ecc_code[p]
 end
```

ECC: Problem Decomposition

- Limited location of bit errors to a sub-range out of the >200 possible locations in multiple formal runs
  - Small example: 60-bit data, 2-bit errors, 10 bit sub-range

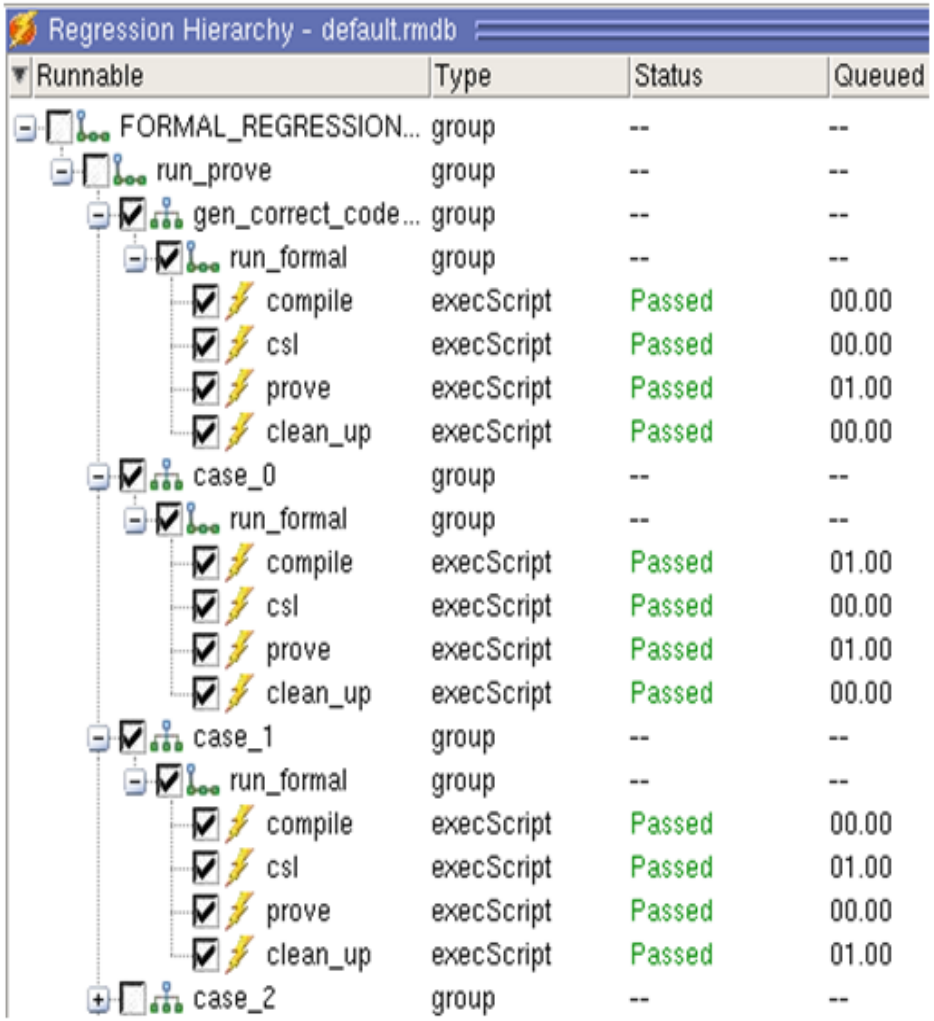


- Nothing lost in decomposition if all cases are covered

ECC: Formal Run Management

- Leads to a large number of formal jobs
- Use Questa VRM to launch multiple runs in parallel on a grid system

1. Monitor progress
2. Automatically collect the results



Runnable	Type	Status	Queued
FORMAL_REGRESSION...	group	--	--
run_prove	group	--	--
gen_correct_code...	group	--	--
run_formal	group	--	--
compile	execScript	Passed	00.00
csl	execScript	Passed	00.00
prove	execScript	Passed	01.00
clean_up	execScript	Passed	00.00
case_0	group	--	--
run_formal	group	--	--
compile	execScript	Passed	01.00
csl	execScript	Passed	00.00
prove	execScript	Passed	01.00
clean_up	execScript	Passed	00.00
case_1	group	--	--
run_formal	group	--	--
compile	execScript	Passed	00.00
csl	execScript	Passed	01.00
prove	execScript	Passed	00.00
clean_up	execScript	Passed	01.00
case_2	group	--	--

Outline

- Design Under Verification
- Error Correction Circuits
- Bitline Redundancy
- Integration of Formal and Simulation Results
- Conclusions and Outlook

Bitline Redundancy Description

- The bitline redundancy is used to improve flash yield
  - Defects detected during manufacturing
  - Redundant bitlines replace defective bitlines

- Each redundant bitline has a configuration register containing information about:
  - Whether the bitline itself is defective
  - Whether it is in use
  - The address of the defective bitline that it's replacing

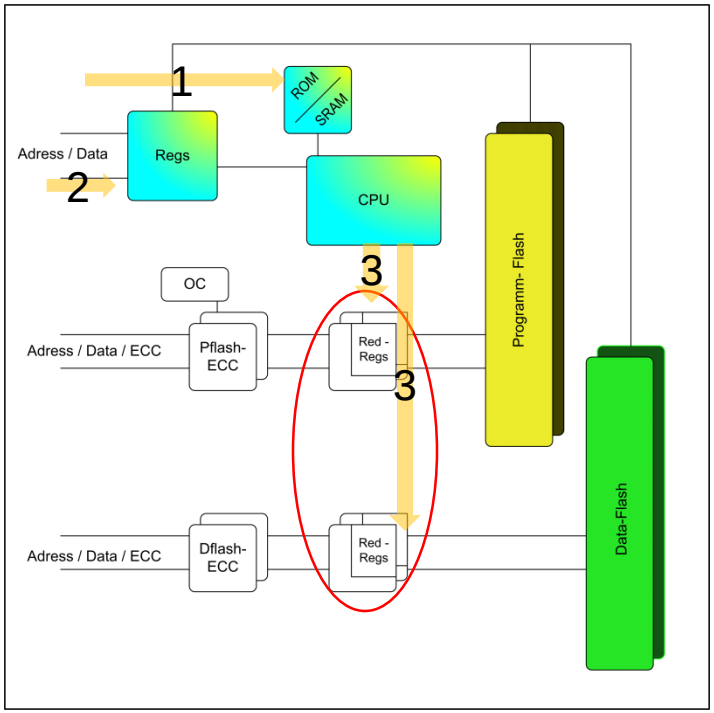
Bitline Redundancy Verification Requirements

Must verify:

- Control of redundant bit lines:
  - The redundancy mapping can be centrally disabled
  - Configuration registers are accessible for read and write
- The scrambling is happening according to the specification
  - Redundant bitlines correctly swapped for defective ones

Bitline Redundancy Simulation Verification Challenges

- Registers only written from CPU
- Initialized once at startup
- Needs special µcode
- Is time consuming
  - in particular writing to the flash
- Leads to low coverage
  - <1% of possibilities covered



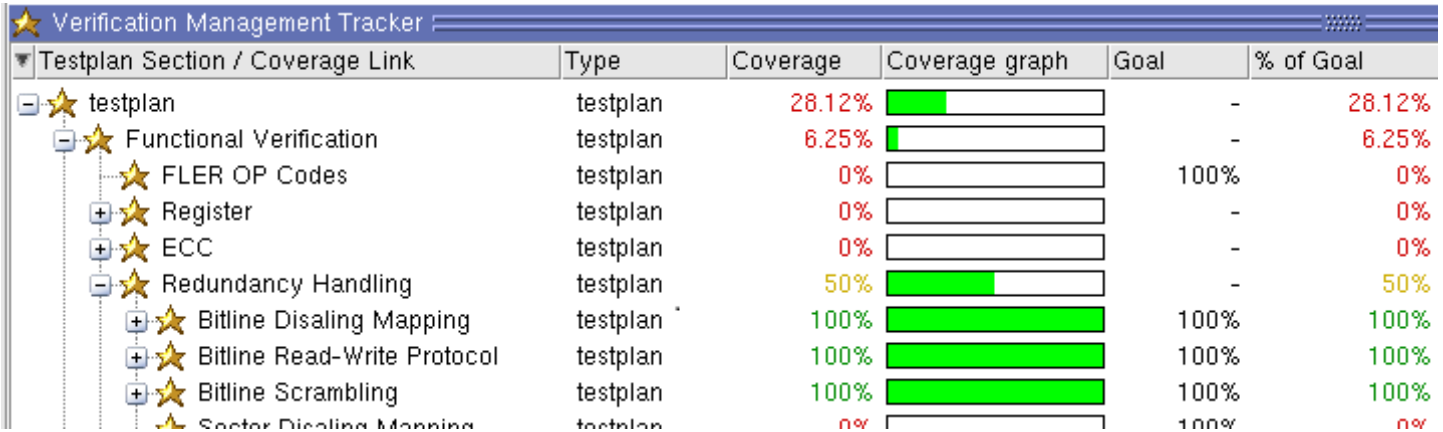
Bitline Redundancy Formal Verification

- Modeling:
  - White box approach (access internal signals using shared VHDL packages)
  - Use of SVA and SV bind
  - A nested for-loop generates the assertions addressing the correctness for each page
    - Total number of assertions 340 (see paper)
  - Tool setup and run with minimal effort
- Results:
  - Complete and full proofs within 5 hours
    - No decomposition required - all combinations of scrambling verified

Outline

- Design Under Verification
- Error Correction Circuits
- Bitline Redundancy
- Integration of Formal and Simulation Results
- Conclusions and Outlook

Integration of Formal and Simulation Results



Testplan Section / Coverage Link	Type	Coverage	Coverage graph	Goal	% of Goal
testplan	testplan	28.12%		-	28.12%
Functional Verification	testplan	6.25%		-	6.25%
FLER OP Codes	testplan	0%		100%	0%
Register	testplan	0%		-	0%
ECC	testplan	0%		-	0%
Redundancy Handling	testplan	50%		-	50%
Bitline Disaling Mapping	testplan	100%		100%	100%
Bitline Read-Write Protocol	testplan	100%		100%	100%
Bitline Scrambling	testplan	100%		100%	100%

- The verification plan is imported into Questa Sim's Unified Coverage Database (UCDB)
- The verification plan UCDB can be merged with the simulation coverage data
- UCDBs from different tools and methodologies can be merged together

Outline

- Design Under Verification
- Error Correction Circuits
- Bitline Redundancy
- Integration of Formal and Simulation Results
- Conclusions and Outlook

Conclusion and Outlook

Lessons learned:

- Formal techniques easier than ever to use
- Pushing the limits of formal is possible w/o losing quality through decomposition
- Formal complements simulation-based verification to reach higher coverage
  - Leverages Verification Management and UCDB

Future work:

- Further use of formal methods are planned
  - SoC interconnectivity validation
  - Interrupt Handler verification

If you liked this presentation, please send feedback to:

Othmane.Bahlous@infineon.com

Questions?