Outline

- What's the difference between „verification in industry“ and „verification elsewhere“?
- The problem with abstractions
- Verification by simulation – why are we still doing it?
- Successful applications of (semi-) formal verification
- Verification templates – a cook book for verifying a class of logic
- Conclusion
Outline

- What's the difference between „verification in industry” and „verification elsewhere”?
- The problem with abstractions
- Verification by simulation – why are we still doing it?
- Successful applications of (semi-) formal verification
- Verification templates – a cook book for verifying a class of logic
- Conclusion

Formal Verification of Processors in the Literature

- Many papers about verification of in-order pipelines, out-of-order schedulers, cache-coherence protocols, etc.
- Cover a significant portion of the complexity of microprocessors
- Mostly on an abstract level
  - execution units black-boxed, number of instructions limited, infinite tag-space, ...
  - hard to bridge to the real design
- Verified on RTL level (and implemented): FM9001 & VAMP
**FM9001 / FM9801**

- Processor defined and verified in Nqthm theorem prover [Brock/Hunt 92]
- Very simple micro-architecture:
  - non-pipelined, 15 instructions, several addressing modes
- Specification is “instruction interpreter”
- Verification target is a gate-level netlist
- Has been implemented as ASIC
- Sawada/Hunt verified FM9801 in ACL2
- Much more elaborate micro-architecture:
  - pipelined, speculative execution, out-of-order, interrupts, load-bypass,...
  - up to 15 instructions in flight
- Likely the most complex design formally verified at that time [1999]

**VAMP**

- Verified at U Saarbrücken (Germany) [Beyer, Leinenbach, Jacobi, Kröning, Paul]
- Described at gate-level in PVS theorem prover
- Verified against high-level correctness statement (1 insn/cycle ref model)
- VAMP micro-architecture:
  - out-of-order Tomasulo scheduler
  - split, coherent I- and D-Caches
  - floating-point unit
  - precise interrupts
- Implemented on Xilinx FPGA
- Probably the most complex formally verified and implemented processor to date
- 2548 lemmas, 88622 proof steps
- 1.5M gates, 9100 register bits (excl caches)
The beasts we're talking about

**Power 5**
- Dual pSeries CPU, SMT core (2 virtual processors/core)
- 64 bit PowerPC
- 276 million transistors
- 8-way superscalar
- Split L1 Cache (64k I & 32k D) per core
- 1.92MB shared L2 Cache >2.0 GHz clock frequency
- Size: 389 sq mm
- 2313 signal I/Os

**POWER architecture:**
- many 100 instructions
- >17 instruction formats
- different address translation modes
- virtualization support
- week order multiprocessor memory coherency

Power5 microarchitecture

- Very complex micro-architecture
  - 2-way SMT
  - multiple branch-prediction mechanisms
  - 5 instruction/group dispatch
  - 2-thread shared register renaming
  - out-of-order issuing/execution
  - re-grouping after execution
  - in-order group commit
  - load/store reordering for architectural MP-coherency
  - associative segment and page table look-aside buffers
  - ....
- Dual-core per chip
  - shared L2-cache, on-chip L3 dir

→276M transistors incl ~2MB cache
But that's not all

Multiple cores and cache chips packaged into a multi-chip module, multiple MCMs packed into „books“, multiple „books“ form the whole SMP system

Comparison of VAMP and a recent IBM cache unit

- **VAMP processor:**
  - 1.5M gates, 9100 register bits
  - including register files, FPU, cache control (no SRAM), reorder buffer
  - 70k lines of PVS code incl specification
- **VAMP Cache**
  - split I/D cache, coherent
  - one fetch request at a time (per I/D)
  - blocking on miss
  - non-pipelined cache access (modeled as single cycle cache access)
  - no MP coherency handling
  - no store-queue/store-forwarding etc.

- **IBM Cache unit**
  - ~12000 latches
  - not counting dir+data SRAMs, register file buffers etc., adding several MBit state
  - ~80k lines of VHDL code

- **Complexities:**
  - multiple pipelined fetch requests
  - non-blocking on miss
  - coherency handling for I/D cache
  - MP coherency
  - store-queue, interacting with MP coherency

➤ A single unit in a real-live CPU has about the same complexity as the whole VAMP processor

(assuming that #latches, #lines of code etc says at least a bit about complexity...)
Outline

- What's the difference between „verification in industry“ and „verification elsewhere“?
- The problem with abstractions
- Verification by simulation – why are we still doing it?
- Successful applications of (semi-) formal verification
- Verification templates – a cook book for verifying a class of logic
- Conclusion

Verification Stack

- Need to verify „real design“

```vhdl
muxout <= gate_and(cond, d1(0 to 63)) or gate_and(cond, d2(0 to 63));
```
Verification Stack (cont.)

- Need to verify „real design”

Pervasive Verification
- Scan testing
- LBIST verification
- Test-pattern generation
- Signature computation

Logic Design (e.g., VHDL)

muxout <=
gate_and(cond, d1(0 to 63)) or
gate_and(cond, d2(0 to 63));

Transistor-level schematic

How does this look for formal verification?
Formal Verification on abstract model

- Need to verify „real design“

Verification of an abstract model is nice to have, but doesn't solve our problem! Ultimately, we must verify the real design!

Consequences for abstractions

- Complete flow relies on low-level VHDL, ensures end-to-end verification
- Even silver-bullet for high-level verification solves only fraction of the problem
- For any new tool/methodology/abstraction trick/language/... to be applied:
  - it needs to understand the low-level language, and needs to bridge to it
  - abstract models/abstract languages currently not consistently used and maintained, won’t change unless the gap is closed

- A revolution is unlikely to happen: we have to live with the low-level stuff (at least in the high-frequency design field.)
  - and there’s many technical and non-technical reasons for that
Outline

- What’s the difference between „verification in industry” and „verification elsewhere”?
- The problem with abstractions
- Verification by simulation – why are we still doing it?
- Successful applications of (semi-) formal verification
- Verification templates – a cook book for verifying a class of logic
- Conclusion

Verification Steps

- **Unit-sim** is the work-horse of verification
  - unit is LSU, ISU, FXU, FPU, ...
  - most bugs are found on unit-level
- Once stable, units integrated to **core-level simulation**
  - can execute real programs here
  - special „test program generators“ tailored to test interesting scenarios [AVPGEN; GPro]
  - verification on architecture level (reusable!)
- Multiple cores/chips/IO/memory etc. integrated into **system sim**
  - can verify MP effects with real cores/memory
  - can verify „power-on & boot“ on this model
    - including config scan, bootstrap init, PLL ...
    - (back to „all the details there“)
  - only few bugs slip into system sim → mostly „power-on stuff“
Macro- vs. Unit-level Verification

- Macros are blocks with 100-1000 registers
  - cover a certain functionality, and tie them together as one PD-entity
  - unit comprises dozens of macros
- Many macros heavily interact to achieve a certain functionality
  - FPU: typical macros are multiplier, shifter, adder, exponent macros, etc.
  - large interaction between macros for datapath control (shift-amount, carry's, etc.)
  - cache: fetch controller, address queue, directory compare, data access, ECC, ...
- Macro I/Os change late due to timing & bugs

- Unit is the lowest „transactional level“
  - perform multiply-add, fetch, store, ...
- Relatively stable & well-documented interfaces, which eases verification
  - usually a unit has ~200 I/O-signals and busses
  - a macro also has ~200 I/Os, and a unit has dozens of macros

\( \rightarrow \) attempts made, but macro level too much overhead as main verif target

Benefits and Drawbacks of Simulation

- It scales: from unit level to system level, always working on the real VHDL
  - nearly linear time / model-size
- Find most bugs: the simple ones immediately, the complex ones after some „cooking time“.
- Proven methodology \( \rightarrow \) first hardware usually works amazingly well

**We know how it works**
- huge investment in training: re-use concepts, lessons-learned, sometimes code from previous project
- want to verify a new unit design: „there's always somebody around who's done something similar before“.
- project manageability: predictable technology

Drawbacks:
- some bugs found very late, never sure you got all
- some bugs not found at all before tape-out
Outline

- What's the difference between „verification in industry“ and „verification elsewhere“?
- The problem with abstractions
- Verification by simulation – why are we still doing it?
- Successful applications of (semi-) formal verification
- Verification templates – a cook book for verifying a class of logic
- Conclusion

Successful Applications of Formal Methods

- Cache Coherency Protocol Verification [German]
  - on a very abstract protocol level
  - finds problems in the coherency protocol before its RTL implementation
  - sometimes serves as guide for RTL implementation

- Sub-unit verification [Gott et.al.]
  - used complementary with unit simulation on „special focus areas“
  - sort of „multi-macro-level verification“
  - verify certain aspects of that area, not the complete functionality thereof
  - typical areas: arbitration logic, queue control (overflow/underflow), ...

- „Pervasive Logic“ Verification [Gloekler et.al.]
Successful Applications of Formal Methods (cont.)

- Equivalence Checking on late design changes (e.g. for timing) [Paruthi]

- Testfloor bug re-creation [Gott et al.]
  - very hard to find by sim (obviously, went through sim regression before tape-out)
  - stable VHDL, known bug area, known symptoms \(\rightarrow\) can reduce model to make suited for SFV
  - write driver and checker to see how problematic symptom could be reached and verify fix

Successful Applications of Formal Methods (cont.)

- FPU Datapaths – the #1 success-story in FV
  - received a lot of attention after the ’94 Pentium FPU flaw
  - research community developed feasible solutions to FPU datapath verification problem
  - FV now standard on FPU datapaths:
    - AMD uses ACL2 [Russinoff]
    - Intel uses STE [Kaivola et al. / O'Leary et al.]
    - IBM uses BDD based symbolic simulation [Jacobi et al.]
FPU datapath verification

- Reference Model approach
- One single instruction issued and arithmetic result is compared

- Complexity is alleviated by case-splitting
  - split shift-amount on alignment shifter and normalization shifter
  - multiplier black-boxed and verified separately
- After case-splitting FPU is amenable to BDD-based symbolic simulation

- Case-splits are done on reference FPU, which is reusable
- Case-splits are „based on architecture“, not micro-architecture
  → makes case-splits applicable to other FPUs with different micro-architecture !!!

FPU datapath verification (cont.)

- Formal Verification is the main method for FPU datapath verification
- FPU is the only mainline unit where FV prevails for a big chunk of logic
  - complete specification of arithmetic correctness
  - exhaustive verification of that specification
  - fundamentally different than other FV applications

- Why is FV being used here? **Because we know how it works!**
- FV on FPU datapath no longer art or science, it’s engineering
  - there was previous work [Chen/Bryant] that led the way
  - the principal methodology is portable between different FPUs
  - it has been applied to multiple FPUs
  - if we hit a problem there’s knowledge on how to debug & alleviate
More units for FV to come?

- FPU is the only unit where FV in the front-seat, for all other units FV is in the back-seat

- Why not use FV on other units in the same way as on FPUs?
  - no experience on FV for other units
  - risky to develop a new methodology on a running project (predictability)
  - we have a working verification strategy: simulation
    → formal verification remains in the back-seat

- Need help from the academic community!
  - create „templates“ for certain classes of logic

Outline

- What's the difference between „verification in industry“ and „verification elsewhere“?
- The problem with abstractions
- Verification by simulation – why are we still doing it?
- Successful applications of (semi-) formal verification
- Verification templates – a cook book for verifying a class of logic
- Conclusion
Verification Templates

- "Template" serves as a recipe for how to verify a certain type of logic
  - should be precise enough to allow for easy implementation
  - should be portable to different implementations of that type of logic
  - should allow RTL-level verification of target logic

- Templates don’t solve the verification problem directly,
  instead turn a science and black art into an engineering problem
  - predictable, plannable, repeatable
  - and very important: teachable
- such that FV becomes applicable in a project environment.
  → compensate for missing experience on FV usage

What’s in a template?

- What type of logic is covered: cache, dataflow control, dispatch/issue, arithmetic, pervasive, recovery, ...
- What language for the testbench: SystemC, e, VHDL, Sugar/PSL, ...
  - what can be reused (shared) from (with) other projects/units: memory manager, instruction queue, instruction reference model, data-structure library, ...
  - language has a great impact on tools and methodology!
- How to specify: assertions/properties, reference model, ...
  - assertions/bounded properties suited for switching network, but not for cache (?)
- Strategy: semi-formal, formal, structural analysis, ...
- What tricks exist:
  - every other instruction, preferred cache lines, immediate invalidation, ...
  - case-splitting on shift amounts, model reductions, ...
What’s in a template?

- What abstractions exist? how do they bridge to the RTL?
  - uninterpreted functions, symmetry, ...
  - need to justify each abstraction against real design!

- How to transfer to a similar type of logic, i.e. different implementation

- How to debug if something goes wrong
  - essential for an engineering task

- Plea for help: need templates
  - write papers about how you verify an implementation of a certain type of logic
  - write more papers about how you verify a different implementation of the same type
  - what’s the difference between the 1st and 2nd time?

Industrial Strength Templates

- Templates need to be „industrial strength“, cope with the complexity of real designs
  - they do not need to deal with all the gory details, but the essential ones

- Example: cache controller
  - deep request pipeline, multi-cycle SRAM access (non-pipelined)
  - segmented cache lines
  - multiple ownership states (shared/exclusive)
  - non-blocking accesses (multiple requests from core, multiple requests to mem)
  - interleaved, buffered data transfers
  - store queue, fetch/XI compare against queued stores
  - broadcast-invalidations compare against STQ and directory and „ops in flight“
Conclusion

- Microprocessors are very complex projects
- No single verification methodology solves all problems
- Simulation is doing surprisingly well
- Formal methods an important complimentary strategy already
- For FV taking a front seat need more experience
- Need academia help for developing experience for FV
- Verification target is RTL, not some abstract level

Hardware-Verification is not a solved problem!