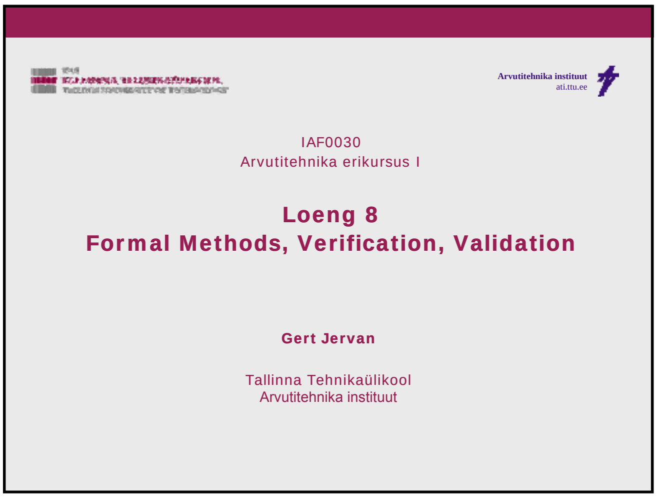


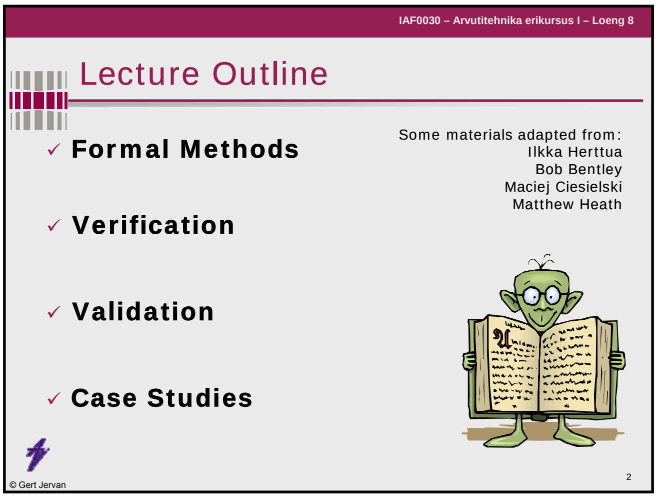
Arvutitehnika instituut
ati.ttu.ee

IAF0030
Arvutitehnika erikursus I

Loeng 8
Formal Methods, Verification, Validation

Gert Jervan

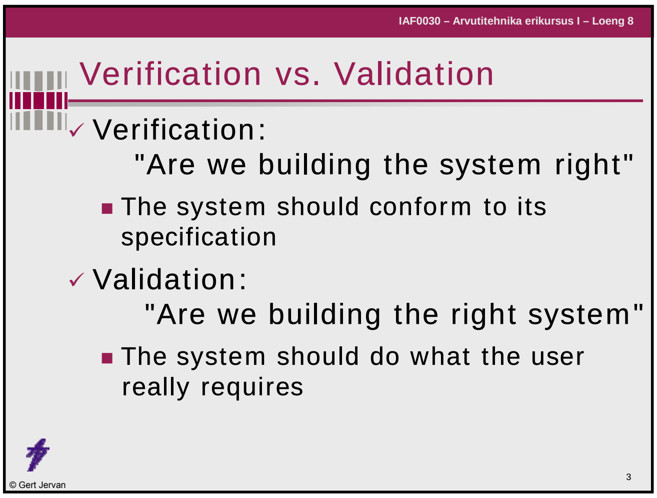
Tallinna Tehnikaülikool
Arvutitehnika instituut



Lecture Outline

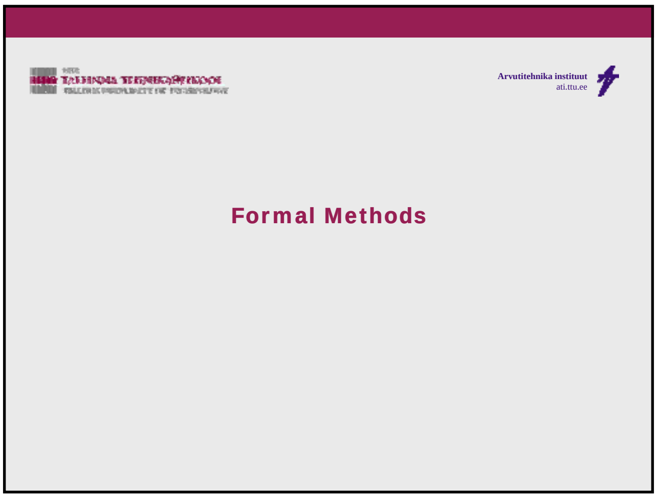
- ✓ **Formal Methods**
- ✓ **Verification**
- ✓ **Validation**
- ✓ **Case Studies**

Some materials adapted from:
Ilkka Herttua
Bob Bentley
Maciej Ciesielski
Matthew Heath



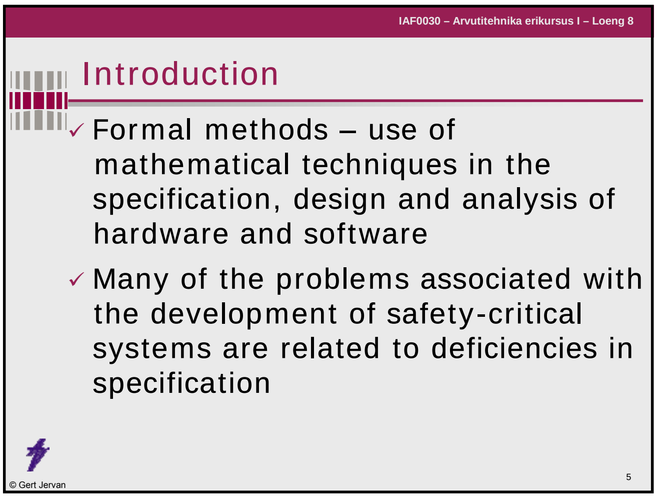
Verification vs. Validation

- ✓ Verification:
  "Are we building the system right"
  - The system should conform to its specification
- ✓ Validation:
  "Are we building the right system"
  - The system should do what the user really requires



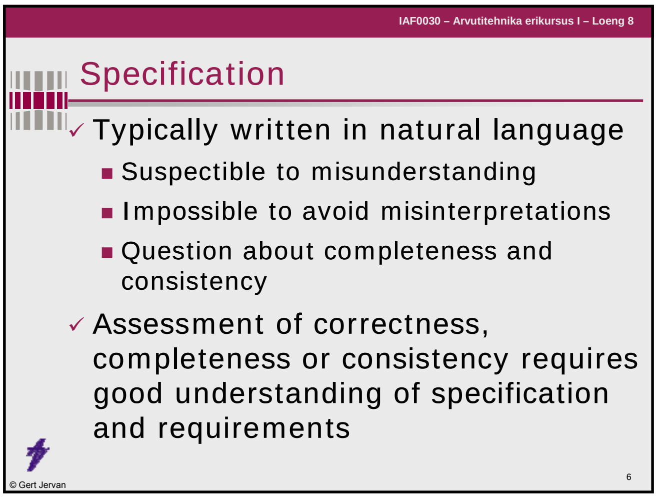
Arvutitehnika instituut
ati.ttu.ee

Formal Methods



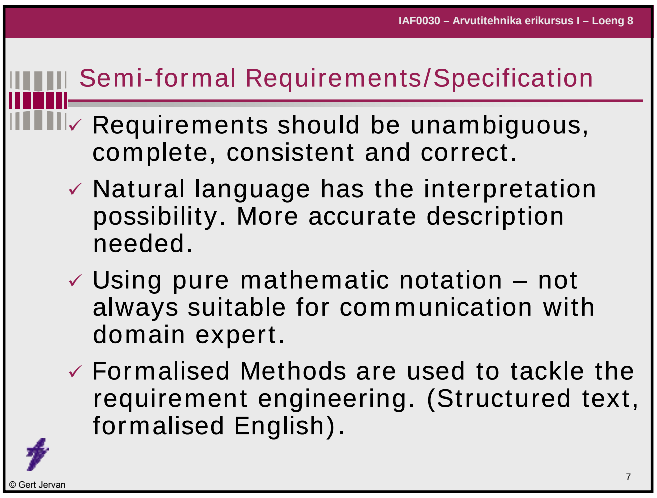
Introduction

- ✓ Formal methods – use of mathematical techniques in the specification, design and analysis of hardware and software
- ✓ Many of the problems associated with the development of safety-critical systems are related to deficiencies in specification



Specification

- ✓ Typically written in natural language
  - Suspectible to misunderstanding
  - Impossible to avoid misinterpretations
  - Question about completeness and consistency
- ✓ Assessment of correctness, completeness or consistency requires good understanding of specification and requirements

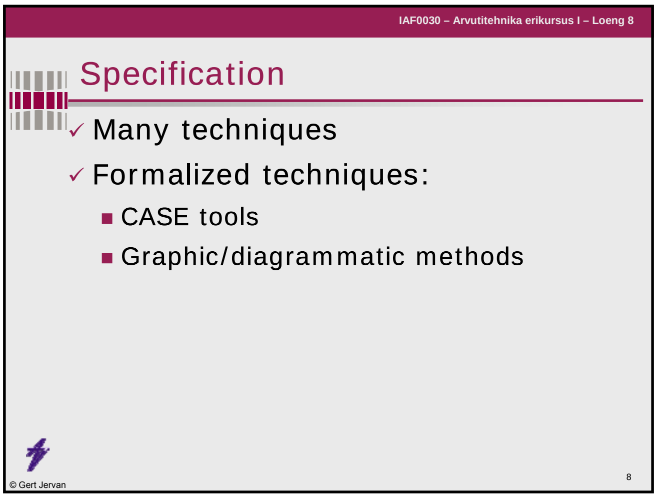


IAF0030 – Arvutitehnika erikursus I – Loeng 8

Semi-formal Requirements/Specification

- Requirements should be unambiguous, complete, consistent and correct.
- Natural language has the interpretation possibility. More accurate description needed.
- Using pure mathematic notation – not always suitable for communication with domain expert.
- Formalised Methods are used to tackle the requirement engineering. (Structured text, formalised English).

© Gert Jervan 7

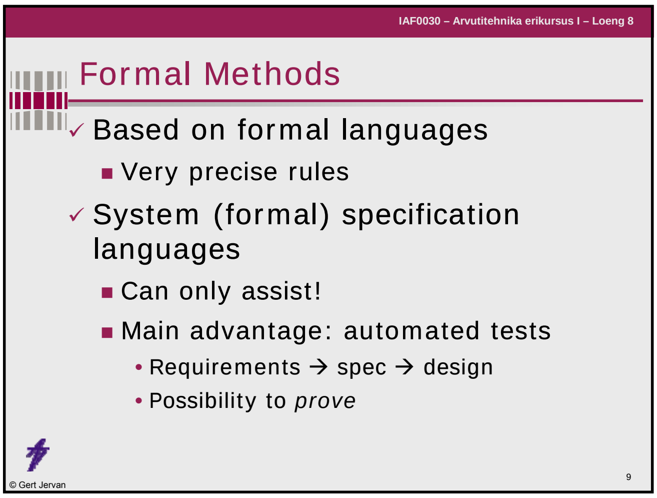


IAF0030 – Arvutitehnika erikursus I – Loeng 8

Specification

- Many techniques
- Formalized techniques:
  - CASE tools
  - Graphic/diagrammatic methods

© Gert Jervan 8

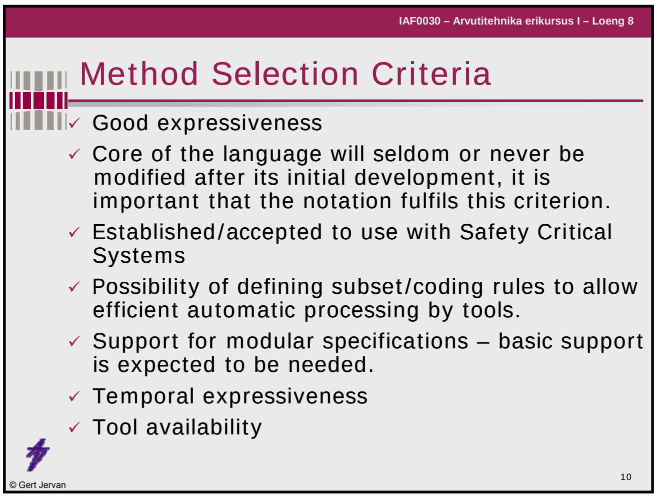


IAF0030 – Arvutitehnika erikursus I – Loeng 8

Formal Methods

- Based on formal languages
  - Very precise rules
- System (formal) specification languages
  - Can only assist!
  - Main advantage: automated tests
    - Requirements → spec → design
    - Possibility to *prove*

© Gert Jervan 9

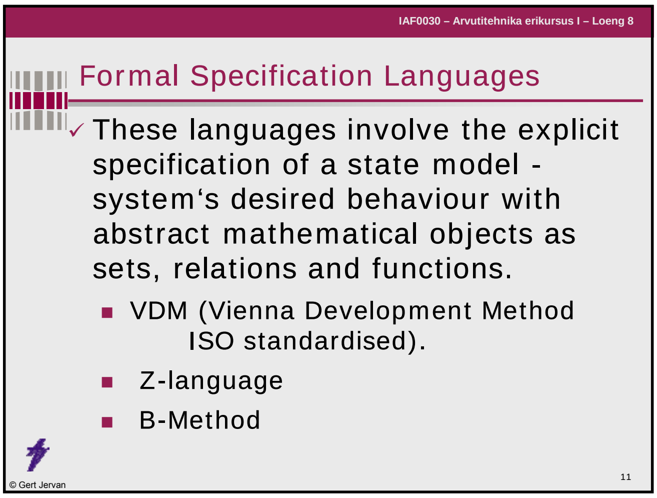


IAF0030 – Arvutitehnika erikursus I – Loeng 8

Method Selection Criteria

- Good expressiveness
- Core of the language will seldom or never be modified after its initial development, it is important that the notation fulfils this criterion.
- Established/accepted to use with Safety Critical Systems
- Possibility of defining subset/coding rules to allow efficient automatic processing by tools.
- Support for modular specifications – basic support is expected to be needed.
- Temporal expressiveness
- Tool availability

© Gert Jervan 10

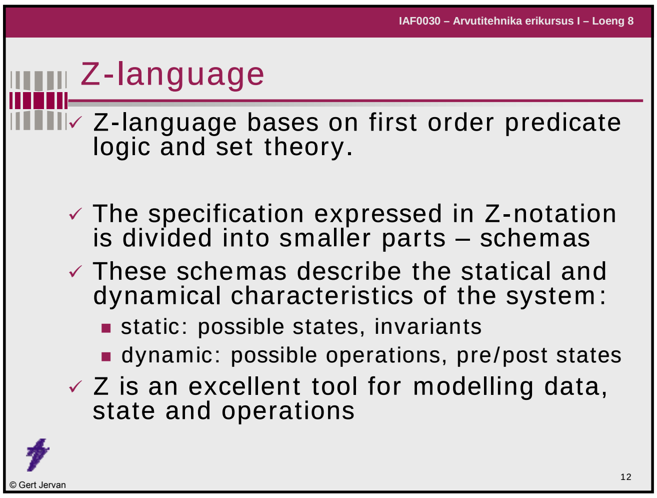


IAF0030 – Arvutitehnika erikursus I – Loeng 8

Formal Specification Languages

- These languages involve the explicit specification of a state model - system‘s desired behaviour with abstract mathematical objects as sets, relations and functions.
  - VDM (Vienna Development Method ISO standardised).
  - Z-language
  - B-Method

© Gert Jervan 11

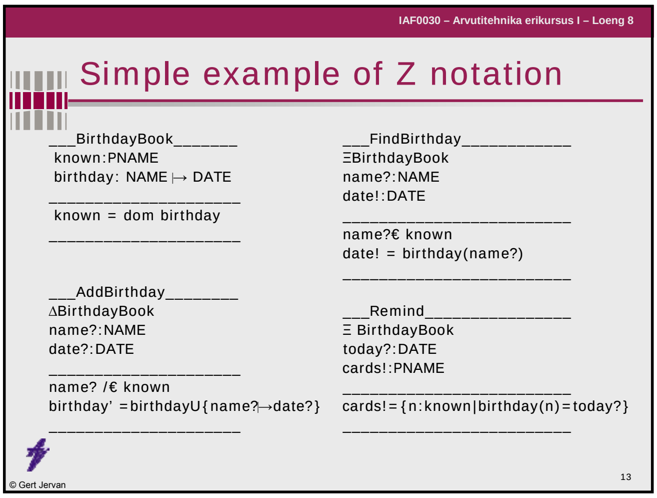


IAF0030 – Arvutitehnika erikursus I – Loeng 8

Z-language

- Z-language bases on first order predicate logic and set theory.
- The specification expressed in Z-notation is divided into smaller parts – schemas
- These schemas describe the statical and dynamical characteristics of the system:
  - static: possible states, invariants
  - dynamic: possible operations, pre/post states
- Z is an excellent tool for modelling data, state and operations

© Gert Jervan 12



IAF0030 – Arvutitehnika erikursus I – Loeng 8

Simple example of Z notation

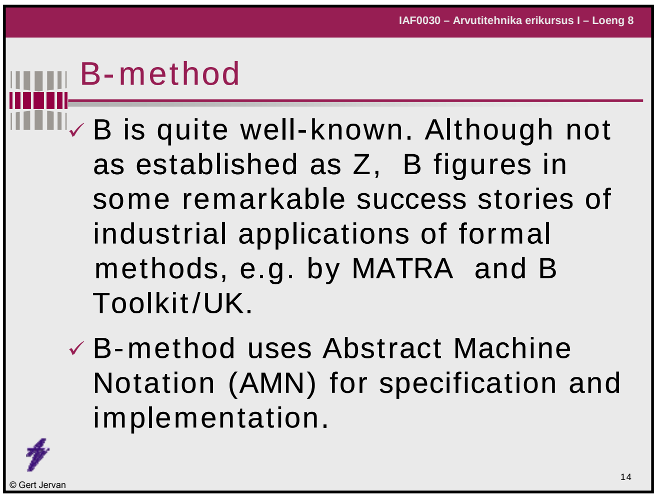
```
___BirthdayBook_______
 known:PNAME
 birthday: NAME |→ DATE
______________________
 known = dom birthday
______________________

___AddBirthday________
ΔBirthdayBook
name?:NAME
date?:DATE
______________________
name? /€ known
birthday' =birthdayU{name?|→date?}
______________________
```

```
___FindBirthday____________
ΞBirthdayBook
name?:NAME
date!:DATE
___________________________
name?€ known
date! = birthday(name?)
___________________________

___Remind__________________
Ξ BirthdayBook
today?:DATE
cards!:PNAME
___________________________
cards!={n:known|birthday(n)=today?}
___________________________
```

© Gert Jervan 13

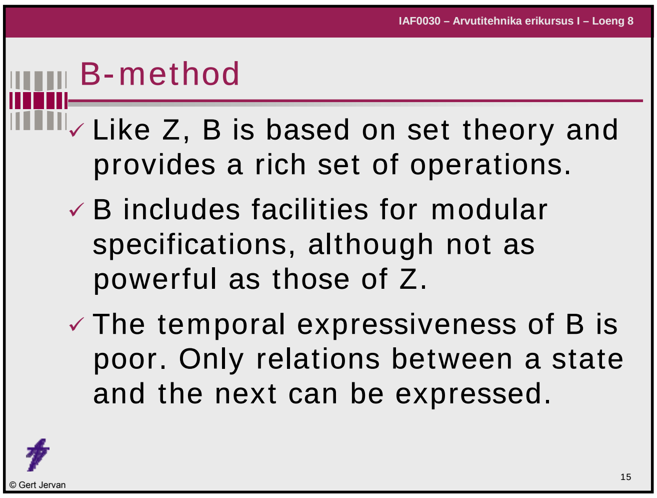


IAF0030 – Arvutitehnika erikursus I – Loeng 8

B-method

- ✓ B is quite well-known. Although not as established as Z, B figures in some remarkable success stories of industrial applications of formal methods, e.g. by MATRA and B Toolkit/UK.
- ✓ B-method uses Abstract Machine Notation (AMN) for specification and implementation.

© Gert Jervan 14

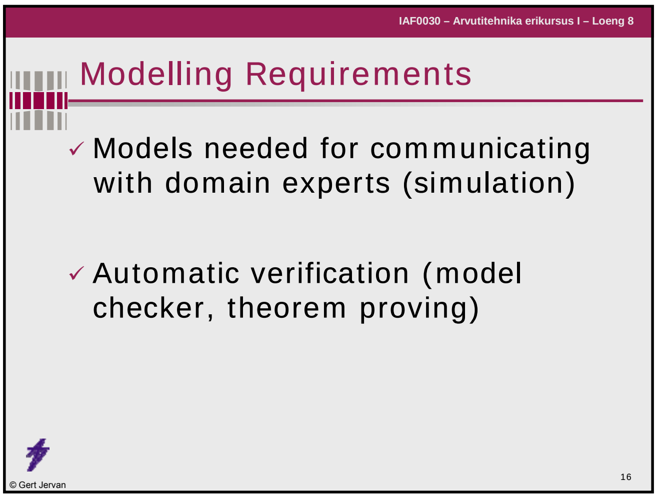


IAF0030 – Arvutitehnika erikursus I – Loeng 8

B-method

- ✓ Like Z, B is based on set theory and provides a rich set of operations.
- ✓ B includes facilities for modular specifications, although not as powerful as those of Z.
- ✓ The temporal expressiveness of B is poor. Only relations between a state and the next can be expressed.

© Gert Jervan 15

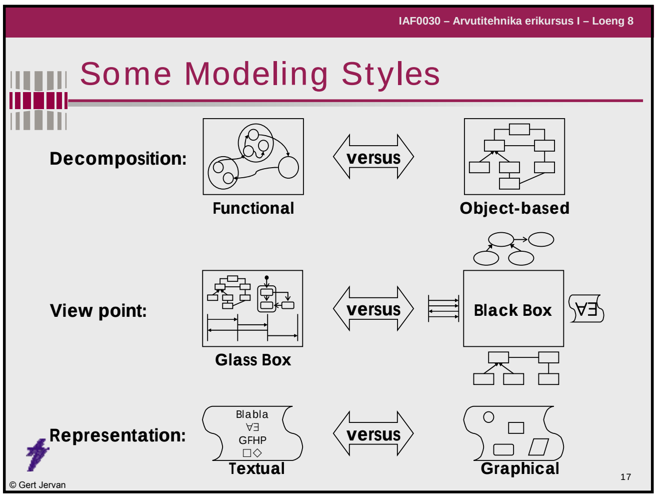


IAF0030 – Arvutitehnika erikursus I – Loeng 8

Modelling Requirements

- ✓ Models needed for communicating with domain experts (simulation)
- ✓ Automatic verification (model checker, theorem proving)

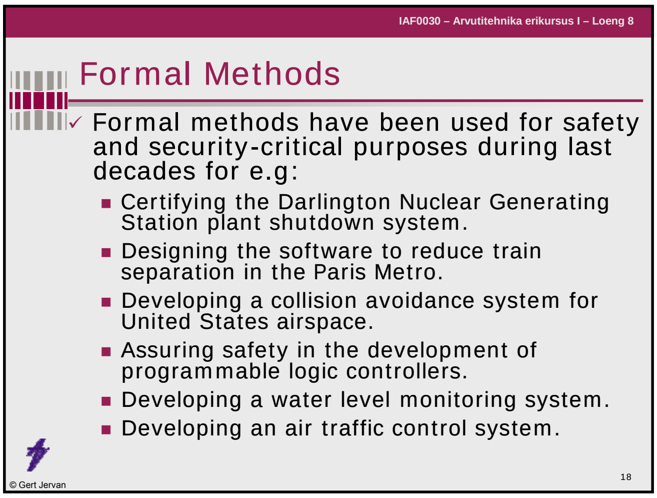
© Gert Jervan 16



IAF0030 – Arvutitehnika erikursus I – Loeng 8

Some Modeling Styles

© Gert Jervan 17

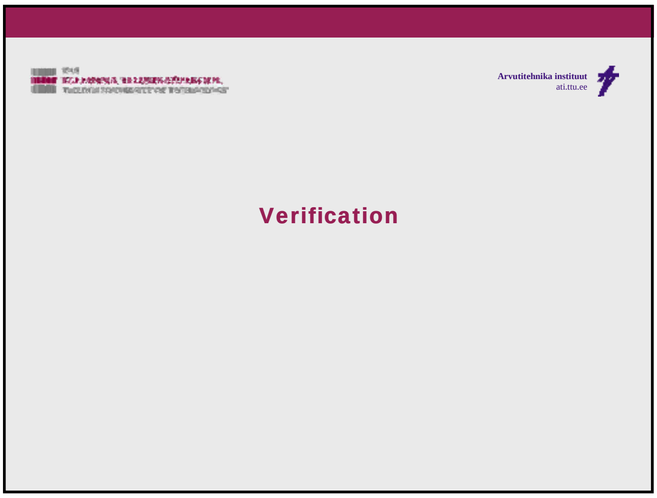


IAF0030 – Arvutitehnika erikursus I – Loeng 8

Formal Methods

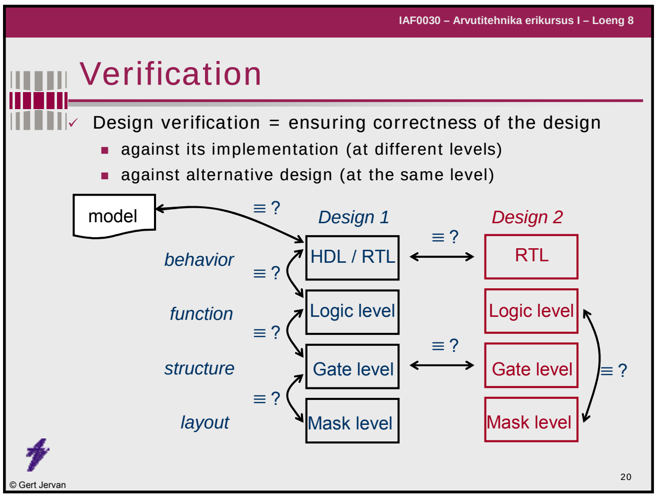
- ✓ Formal methods have been used for safety and security-critical purposes during last decades for e.g:
  - Certifying the Darlington Nuclear Generating Station plant shutdown system.
  - Designing the software to reduce train separation in the Paris Metro.
  - Developing a collision avoidance system for United States airspace.
  - Assuring safety in the development of programmable logic controllers.
  - Developing a water level monitoring system.
  - Developing an air traffic control system.

© Gert Jervan 18



Arvutitehnika instituut
ati.ttu.ee

Verification



Verification

- ✓ Design verification = ensuring correctness of the design
  - against its implementation (at different levels)
  - against alternative design (at the same level)

model ≡ ?

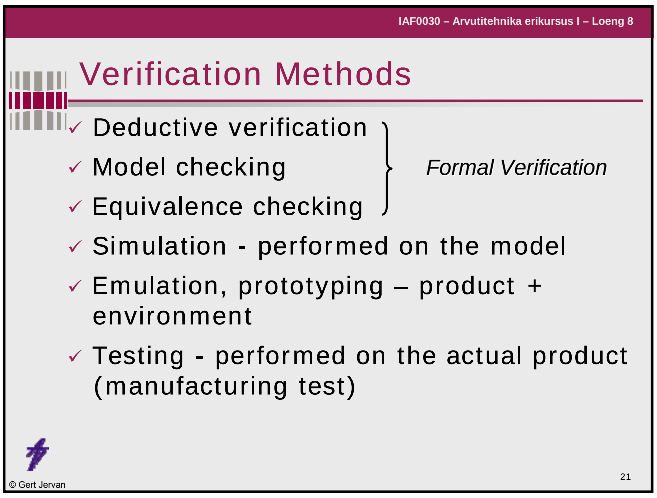
Design 1 — Design 2

behavior ≡ ? HDL / RTL ←→ ≡ ? RTL

function ≡ ? Logic level — Logic level

structure ≡ ? Gate level ←→ ≡ ? Gate level ≡ ?

layout Mask level — Mask level

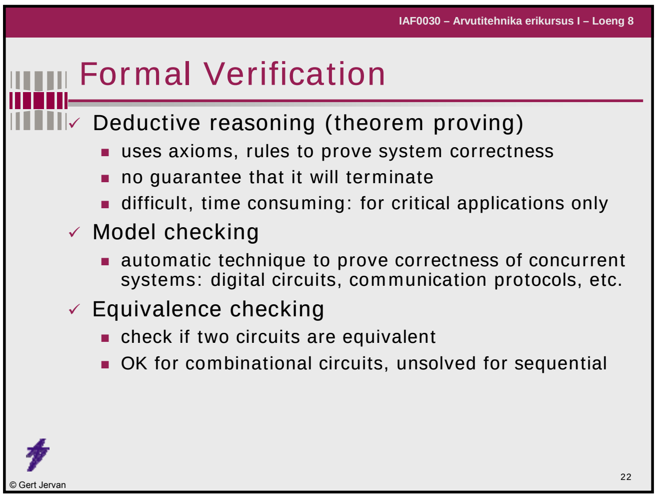


Verification Methods

- ✓ Deductive verification
- ✓ Model checking
- ✓ Equivalence checking

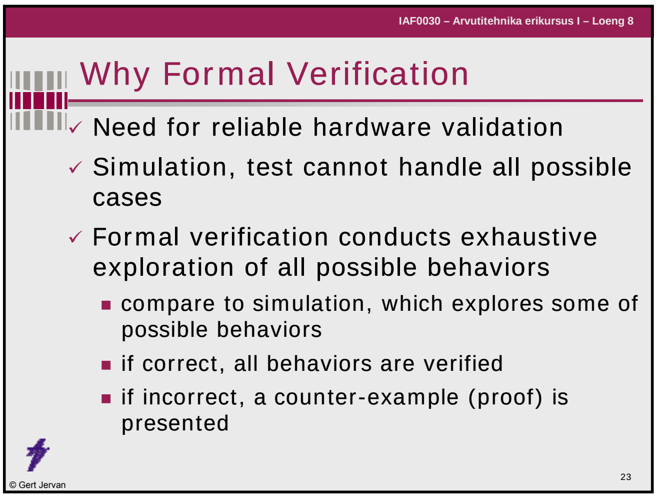
} *Formal Verification*

- ✓ Simulation - performed on the model
- ✓ Emulation, prototyping – product + environment
- ✓ Testing - performed on the actual product (manufacturing test)



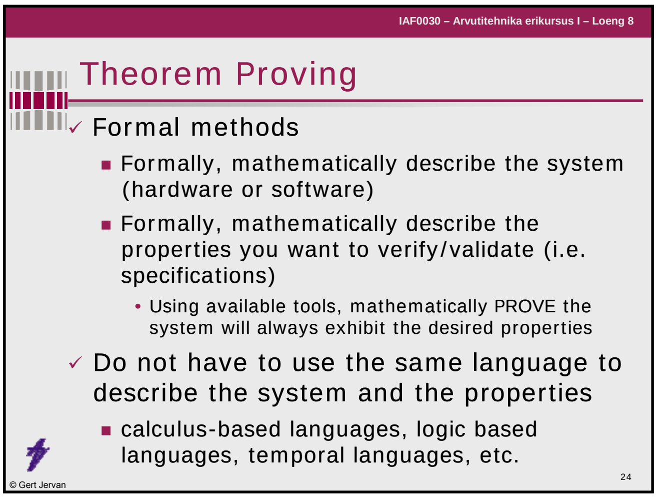
Formal Verification

- ✓ Deductive reasoning (theorem proving)
  - uses axioms, rules to prove system correctness
  - no guarantee that it will terminate
  - difficult, time consuming: for critical applications only
- ✓ Model checking
  - automatic technique to prove correctness of concurrent systems: digital circuits, communication protocols, etc.
- ✓ Equivalence checking
  - check if two circuits are equivalent
  - OK for combinational circuits, unsolved for sequential



Why Formal Verification

- ✓ Need for reliable hardware validation
- ✓ Simulation, test cannot handle all possible cases
- ✓ Formal verification conducts exhaustive exploration of all possible behaviors
  - compare to simulation, which explores some of possible behaviors
  - if correct, all behaviors are verified
  - if incorrect, a counter-example (proof) is presented



Theorem Proving

- ✓ Formal methods
  - Formally, mathematically describe the system (hardware or software)
  - Formally, mathematically describe the properties you want to verify/validate (i.e. specifications)
    - Using available tools, mathematically PROVE the system will always exhibit the desired properties
- ✓ Do not have to use the same language to describe the system and the properties
  - calculus-based languages, logic based languages, temporal languages, etc.

Model Checking

- ✓ Algorithmic method of verifying correctness of (finite state) concurrent systems against temporal logic specifications
  - A practical approach to formal verification
- ✓ Basic idea
  - System is described in a formal model
    - derived from high level design (HDL, C), circuit structure, etc.
  - The desired behavior is expressed as a set of properties
    - expressed as temporal logic specification
  - The specification is checked against the model

Model Checking

- ✓ How does it work
  - System is modeled as a state transition structure (Kripke structure)
  - Specification is expressed in propositional temporal logic (CTL formula)
    - asserts how system behavior evolves over time
  - Efficient search procedure checks the transition system to see if it satisfies the specification

Model Checking

- ✓ Characteristics
  - searches the entire solution space
  - always terminates with YES or NO
  - relatively easy, can be done by experienced designers
  - widely used in industry
  - can be automated
- ✓ Challenges
  - state space explosion – use symbolic methods, BDDs
- ✓ History
  - Clark, Emerson [1981] USA
  - Quielle, Sifakis [1980's] France

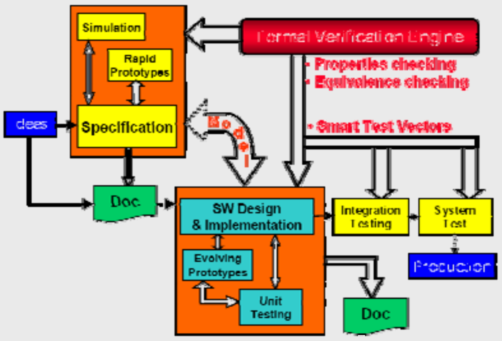
Model Checking - Tasks

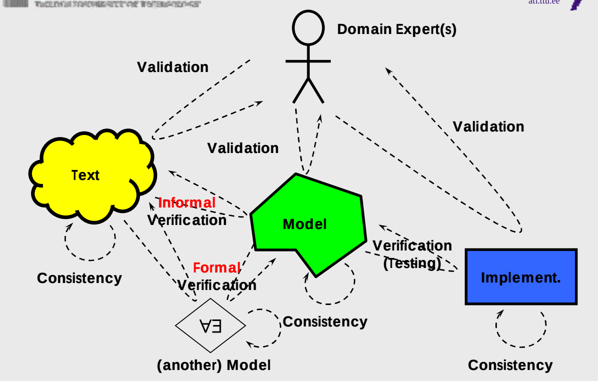
- ✓ Modeling
  - converts a design into a formalism: state transition system
- ✓ Specification
  - state the properties that the design must satisfy
  - use logical formalism: temporal logic
    - asserts how system behavior evolves over time
- ✓ Verification
  - automated procedure (algorithm)

Model Checking - Issues

- ✓ Completeness
  - model checking is effective for a given property
  - impossible to guarantee that the specification covers all properties the system should satisfy
  - writing the specification - responsibility of the user
- ✓ Negative results
  - incorrect model
  - incorrect specification (false negative)
  - failure to complete the check (too large)

Verified software process





Domain Expert(s)

Validation

Validation

Validation

Text

Informal Verification

Model

Verification (Testing)

Implement.

Formal Verification

Consistency

∀∃

Consistency

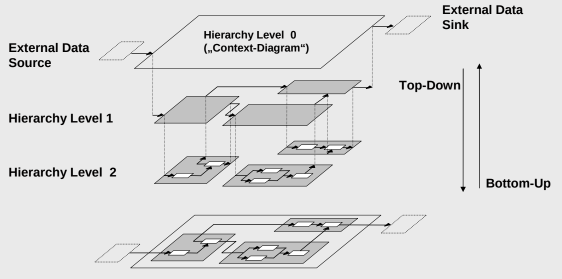
(another) Model

Consistency

Functional Decomposition

- ✓ Functional decomposition breaks down complex systems into a hierarchical structure of simpler parts.
- ✓ Breaking a system into smaller parts enables users to understand, describe, and design complex systems.
- ✓ Functional decomposition consists of the following steps:
  - Define the system context.
    - This will help define the system boundaries.
  - Describe the system in terms of high-level functions and their interfaces.
  - Refine the high-level functions and partition them into smaller, more specific functions.

Functional Decomposition



External Data Source

Hierarchy Level 0 („Context-Diagram“)

External Data Sink

Top-Down

Hierarchy Level 1

Hierarchy Level 2

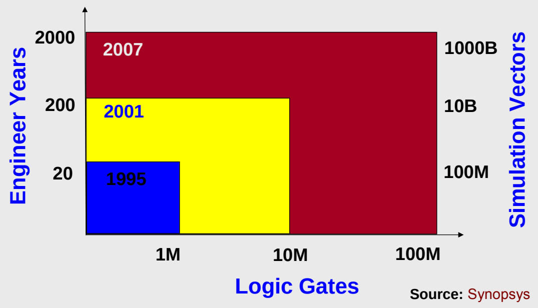
Bottom-Up

Hierarchical Structured Activity Chart

Arvutitehnika instituut
ati.ttu.ee

Validation

Functional Validation of SoC Designs



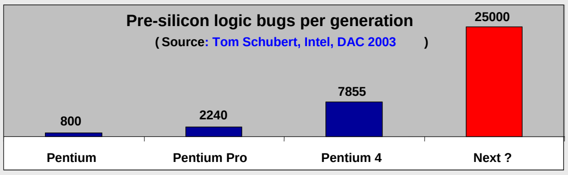
Source: Synopsys

71% of SOC re-spins are due to logic bugs

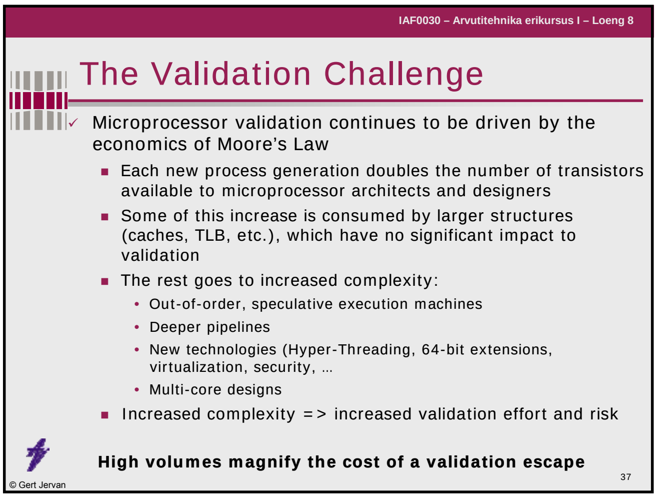
Source: G. Spirakis, keynote address at DATE 2004

Functional Validation of Microprcessors

- ✓ Functional validation is a major bottleneck
  - Deeply pipelined complex microarchitectures



- ✓ Logic bugs increase at 3-4 times/generation
  - Bugs increase (exponential) is linear with design complexity growth.



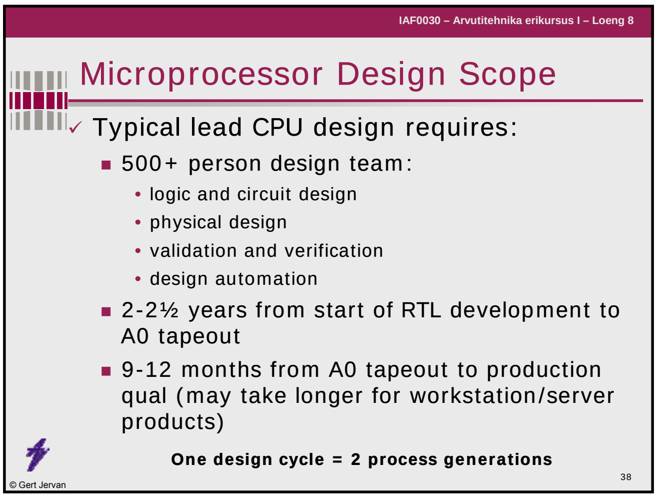
IAF0030 – Arvutitehnika erikursus I – Loeng 8

The Validation Challenge

- ✓ Microprocessor validation continues to be driven by the economics of Moore's Law
  - Each new process generation doubles the number of transistors available to microprocessor architects and designers
  - Some of this increase is consumed by larger structures (caches, TLB, etc.), which have no significant impact to validation
  - The rest goes to increased complexity:
    - Out-of-order, speculative execution machines
    - Deeper pipelines
    - New technologies (Hyper-Threading, 64-bit extensions, virtualization, security, …
    - Multi-core designs
  - Increased complexity => increased validation effort and risk

High volumes magnify the cost of a validation escape

© Gert Jervan 37



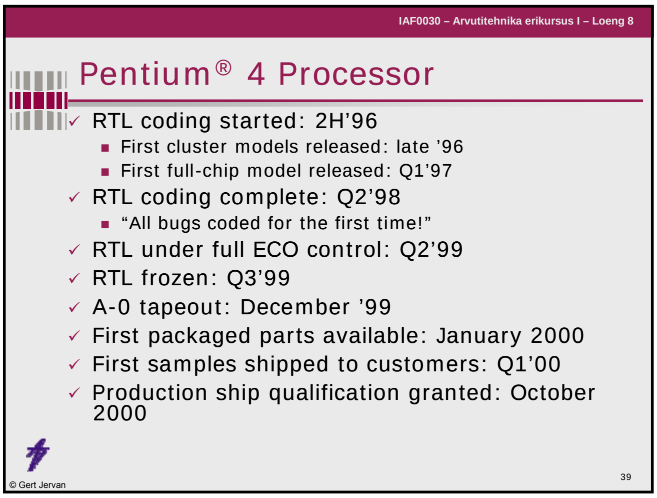
IAF0030 – Arvutitehnika erikursus I – Loeng 8

Microprocessor Design Scope

- ✓ Typical lead CPU design requires:
  - 500+ person design team:
    - logic and circuit design
    - physical design
    - validation and verification
    - design automation
  - 2-2½ years from start of RTL development to A0 tapeout
  - 9-12 months from A0 tapeout to production qual (may take longer for workstation/server products)

One design cycle = 2 process generations

© Gert Jervan 38

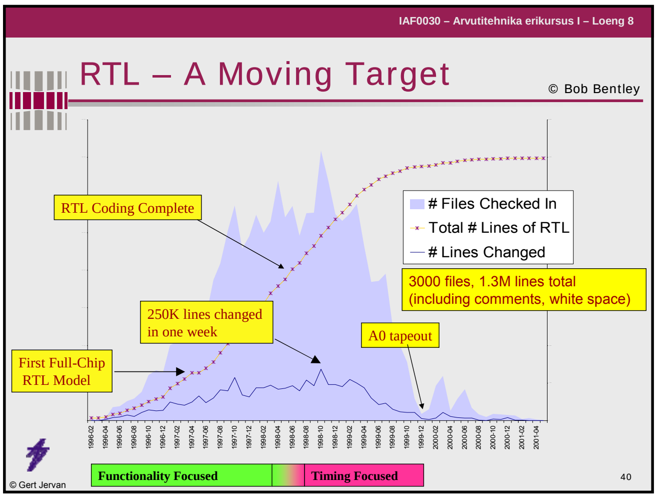


IAF0030 – Arvutitehnika erikursus I – Loeng 8

Pentium® 4 Processor

- ✓ RTL coding started: 2H'96
  - First cluster models released: late '96
  - First full-chip model released: Q1'97
- ✓ RTL coding complete: Q2'98
  - "All bugs coded for the first time!"
- ✓ RTL under full ECO control: Q2'99
- ✓ RTL frozen: Q3'99
- ✓ A-0 tapeout: December '99
- ✓ First packaged parts available: January 2000
- ✓ First samples shipped to customers: Q1'00
- ✓ Production ship qualification granted: October 2000

© Gert Jervan 39



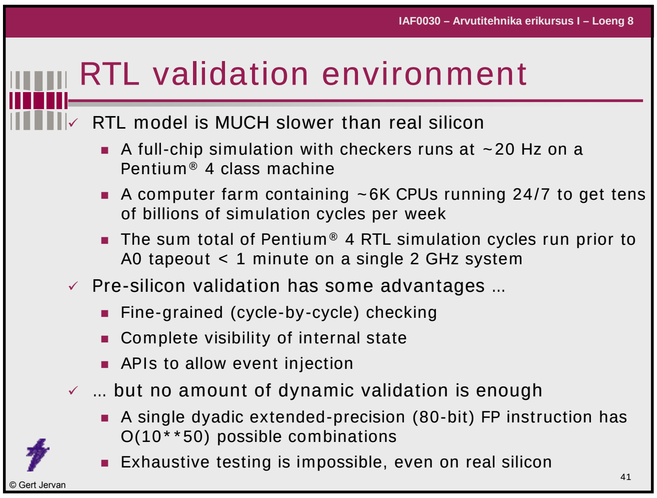
IAF0030 – Arvutitehnika erikursus I – Loeng 8

RTL – A Moving Target

© Bob Bentley

3000 files, 1.3M lines total (including comments, white space)

© Gert Jervan 40

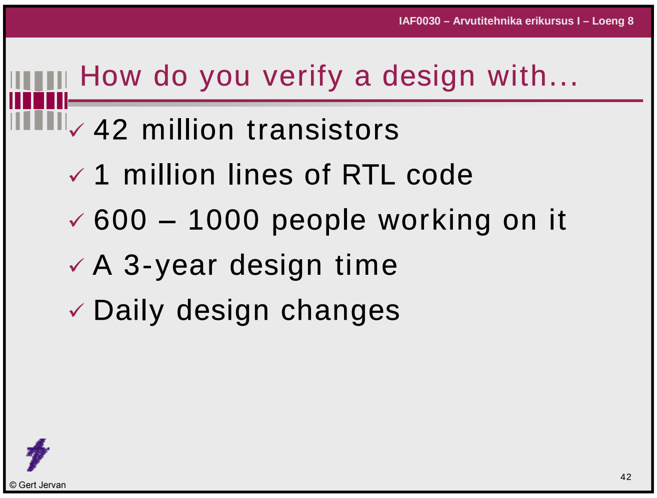


IAF0030 – Arvutitehnika erikursus I – Loeng 8

RTL validation environment

- ✓ RTL model is MUCH slower than real silicon
  - A full-chip simulation with checkers runs at ~20 Hz on a Pentium® 4 class machine
  - A computer farm containing ~6K CPUs running 24/7 to get tens of billions of simulation cycles per week
  - The sum total of Pentium® 4 RTL simulation cycles run prior to A0 tapeout < 1 minute on a single 2 GHz system
- ✓ Pre-silicon validation has some advantages …
  - Fine-grained (cycle-by-cycle) checking
  - Complete visibility of internal state
  - APIs to allow event injection
- ✓ … but no amount of dynamic validation is enough
  - A single dyadic extended-precision (80-bit) FP instruction has O(10**50) possible combinations
  - Exhaustive testing is impossible, even on real silicon

© Gert Jervan 41

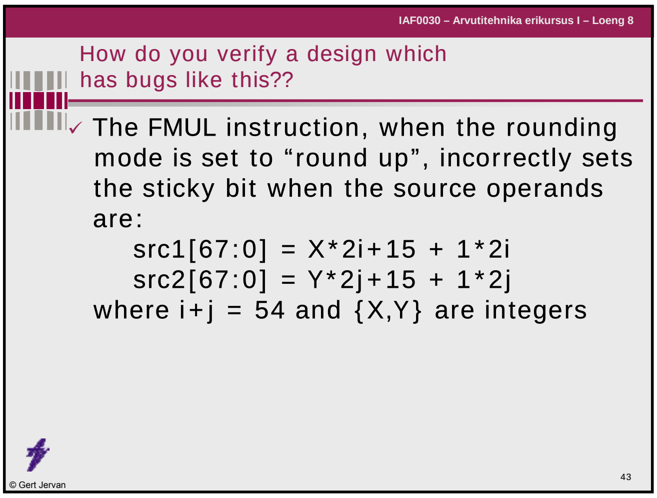


IAF0030 – Arvutitehnika erikursus I – Loeng 8

How do you verify a design with…

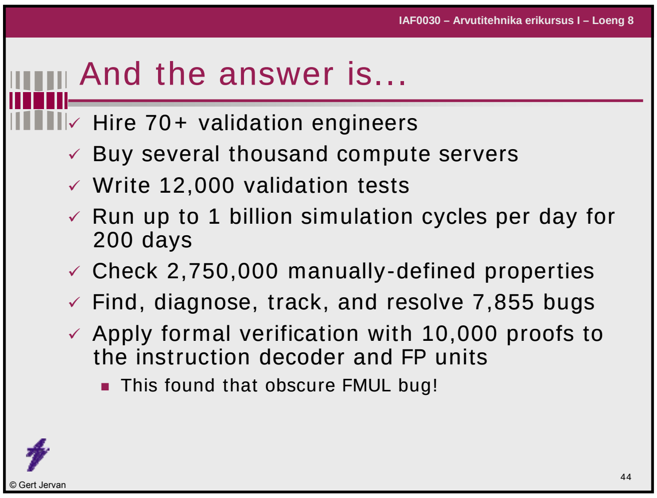
- ✓ 42 million transistors
- ✓ 1 million lines of RTL code
- ✓ 600 – 1000 people working on it
- ✓ A 3-year design time
- ✓ Daily design changes

© Gert Jervan 42



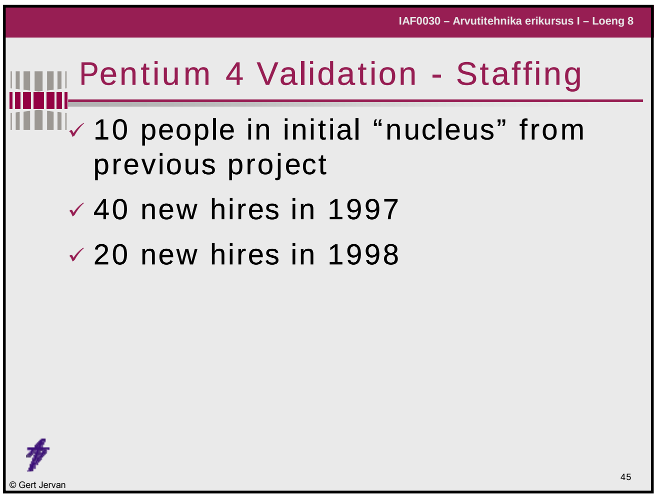
How do you verify a design which has bugs like this??

- The FMUL instruction, when the rounding mode is set to “round up”, incorrectly sets the sticky bit when the source operands are:
  src1[67:0] = X*2i+15 + 1*2i
  src2[67:0] = Y*2j+15 + 1*2j
  where i+j = 54 and {X,Y} are integers



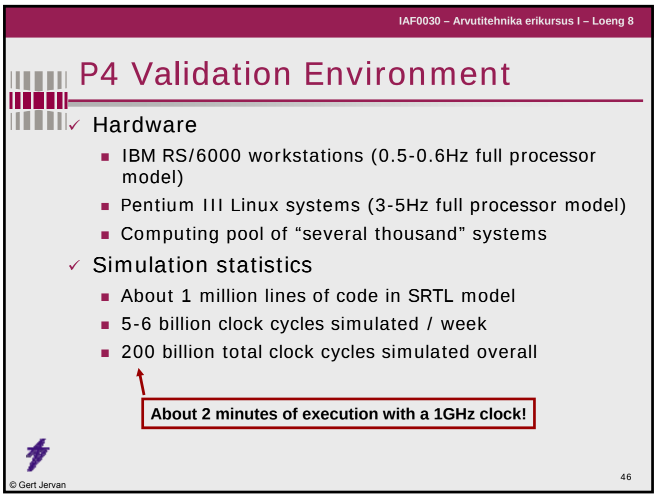
And the answer is...

- Hire 70+ validation engineers
- Buy several thousand compute servers
- Write 12,000 validation tests
- Run up to 1 billion simulation cycles per day for 200 days
- Check 2,750,000 manually-defined properties
- Find, diagnose, track, and resolve 7,855 bugs
- Apply formal verification with 10,000 proofs to the instruction decoder and FP units
  - This found that obscure FMUL bug!



Pentium 4 Validation - Staffing

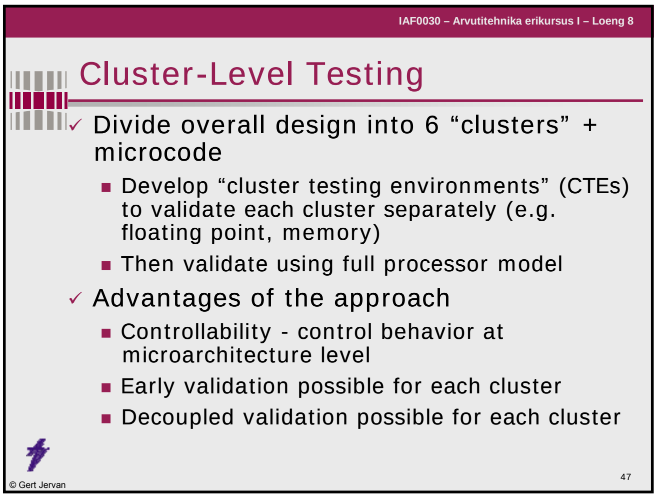
- 10 people in initial “nucleus” from previous project
- 40 new hires in 1997
- 20 new hires in 1998



P4 Validation Environment

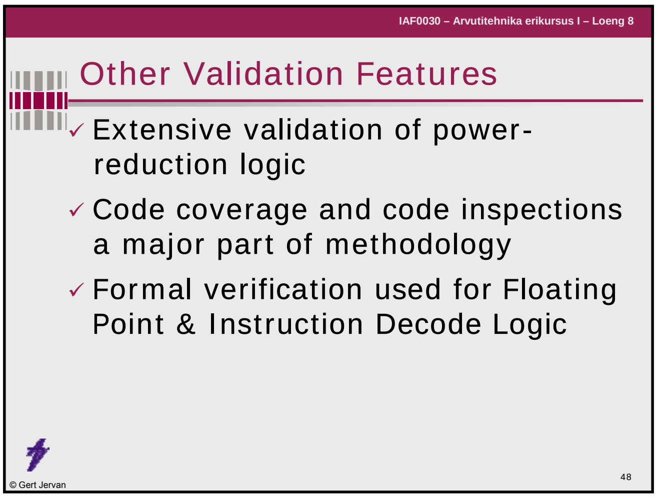
- Hardware
  - IBM RS/6000 workstations (0.5-0.6Hz full processor model)
  - Pentium III Linux systems (3-5Hz full processor model)
  - Computing pool of “several thousand” systems
- Simulation statistics
  - About 1 million lines of code in SRTL model
  - 5-6 billion clock cycles simulated / week
  - 200 billion total clock cycles simulated overall

About 2 minutes of execution with a 1GHz clock!



Cluster-Level Testing

- Divide overall design into 6 “clusters” + microcode
  - Develop “cluster testing environments” (CTEs) to validate each cluster separately (e.g. floating point, memory)
  - Then validate using full processor model
- Advantages of the approach
  - Controllability - control behavior at microarchitecture level
  - Early validation possible for each cluster
  - Decoupled validation possible for each cluster



Other Validation Features

- Extensive validation of power-reduction logic
- Code coverage and code inspections a major part of methodology
- Formal verification used for Floating Point & Instruction Decode Logic

Power Reduction Validation

- ✓ Power consumption was a big concern for Pentium 4
  - Need to stay within the cost-effective thermal envelope for desktop systems at 1.5+ GHz
- ✓ Extensive clock gating in every part of the design
- ✓ Mounted a focused effort to validate that:
  - Committed features were implemented as per plan
  - Functional correctness was maintained in the face of clock gating
  - Changes to the design did not impact power savings
- ✓ ~12 person years of effort, 5 heads at peak
- ✓ Fully functional on A-step silicon, measured savings of ~20W achieved for typical workloads

Formal Verification in P4 Validation

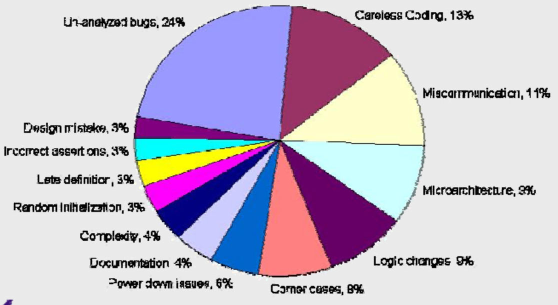
- ✓ Based on model checking
  - Given a finite-state concurrent system
  - Express specifications as temporal logic formulas
  - Use symbolic algorithms to check whether model holds
- ✓ Constructed database 10,000 “proofs”
- ✓ Over 100 bugs found
- ✓ 20 were “high quality” bugs not likely to be found by simulation
- ✓ Example errors: FADD, FMUL

Validation Results

- ✓ 5809 bugs identified by simulation
  - 3411 bugs found by cluster-level testing
  - 2398 found using full-chip model
- ✓ 1554 bugs found by code inspection
- ✓ 492 bugs found by formal verification
- ✓ Largest sources of bugs: memory cluster (25%)

Pentium® 4 Bugs Breakdown

Source: Bob Bentley, HLDVT 2002



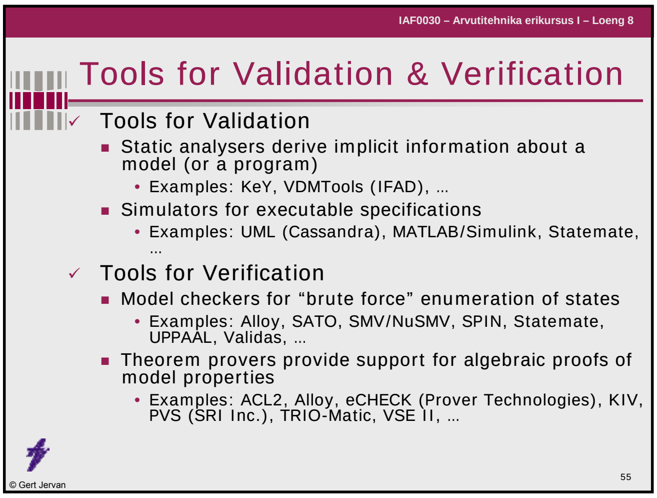
Micro-architectural complexity is a major contributor

Methodology drivers

- ✓ Regression
  - RTL is “live”, and changes frequently until the very last stages of the project
  - Model checking automation at lower levels allows regression to be automated and provides robustness in the face of ECOs
- ✓ Debugging
  - Need to be able to demonstrate FV counter-examples to designers and architects
  - Designers want a dynamic test that they can simulate
  - Waveform viewers, schematic browsers, etc. can help to bridge the gap
- ✓ Verification in the large
  - Proof design: how do we approach the problem in a systematic fashion?
  - Proof engineering: how do we write maintainable and modifiable proofs?

Other Challenges

- ✓ Dealing with constantly-changing specifications
  - Specification changes are a reality in design
  - Properties and proofs should be readily adapted
  - How to engineer agile and robust regressions?
- ✓ Protocol Verification
  - This problem has always been hard
  - Getting harder (more MP) and more important (intra-die protocols make it more expensive to fix bugs)
- ✓ Verification of embedded software
  - S/W for large SoCs has impact beyond functional correctness (power, performance, …)
  - Not all S/W verification techniques apply because H/W abstraction is less feasible
  - One example is microcode verification



IAF0030 – Arvutitehnika erikursus I – Loeng 8

Tools for Validation & Verification

- ✓ Tools for Validation
  - Static analysers derive implicit information about a model (or a program)
    - Examples: KeY, VDMTools (IFAD), …
  - Simulators for executable specifications
    - Examples: UML (Cassandra), MATLAB/Simulink, Statemate, …
- ✓ Tools for Verification
  - Model checkers for “brute force” enumeration of states
    - Examples: Alloy, SATO, SMV/NuSMV, SPIN, Statemate, UPPAAL, Validas, …
  - Theorem provers provide support for algebraic proofs of model properties
    - Examples: ACL2, Alloy, eCHECK (Prover Technologies), KIV, PVS (SRI Inc.), TRIO-Matic, VSE II, …

© Gert Jervan

55

Arvutitehnika instituut
ati.ttu.ee

Questions?

Gert Jervan

Tallinna Tehnikaülikool
Arvutitehnika instituut