

Challenges and Lessons for Software Verification

Jim Grundy
Strategic CAD Laboratories
Hillsboro, Oregon, USA

Lessons and Challenges for Software Verification

Economics made processor design an early adopter of formal verification

- How?
- How, and which, software has similar imperatives

Learned lessons on cost-effective FV

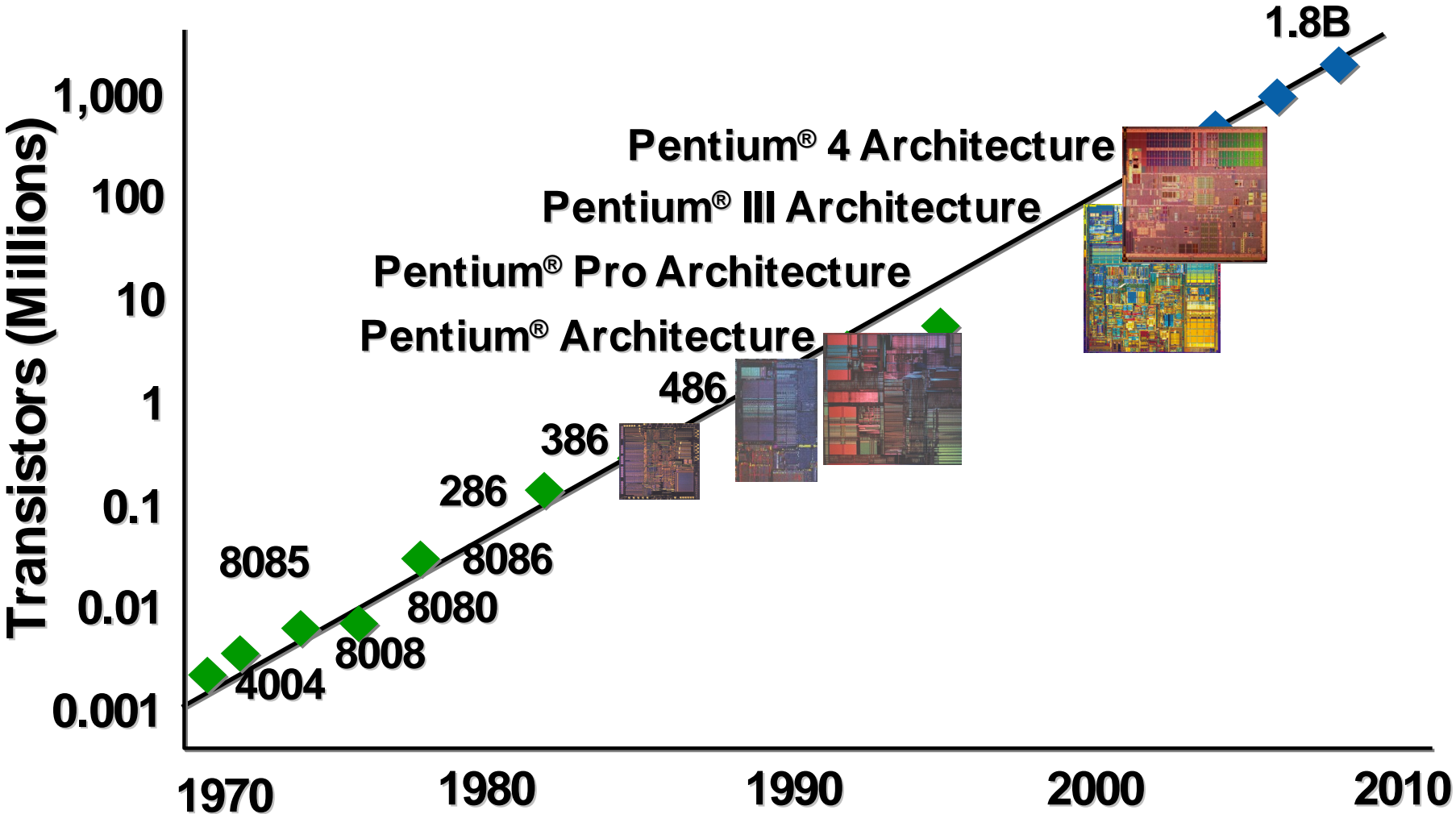
- Some already under investigation for software
- What about the others, can we use them too?

Challenges remain for us

- Refine challenges we set ourselves with these lessons



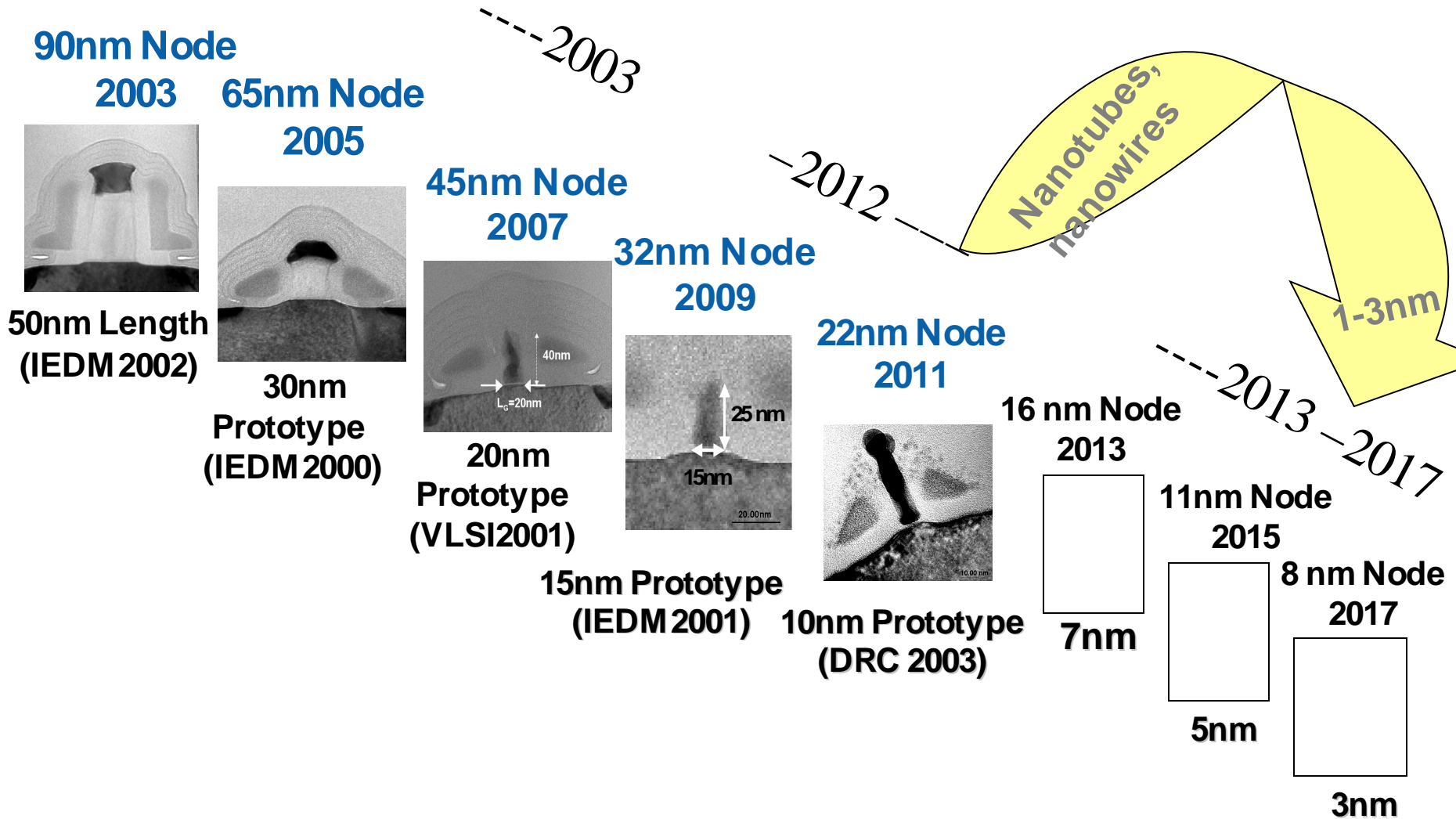
Moore's Law Continues To Hold



Source: Intel Corporation



Moore's Law Continues To Hold



The Validation Challenge

CPU validation driven by economics of Moore's Law

- Each new process doubles the number of transistors available to processor architects
- Some of this is consumed by larger structures (caches, etc.), which have no significant impact to validation
- The rest goes to increased complexity:
 - Out-of-order, speculative execution machines
 - Deeper pipelines
 - New features (Hyper-Threading, 64-bit, virtualization, ...)
 - Multi-core designs

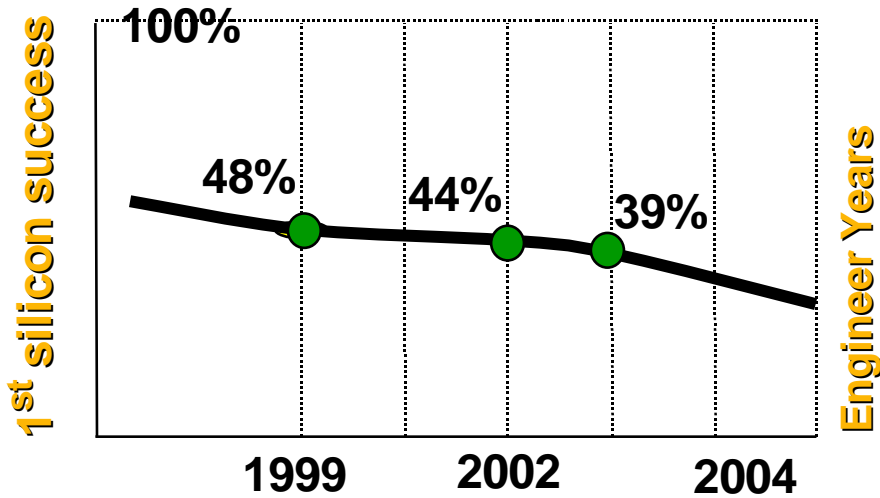
Complexity leads to validation effort and *risk*



Design Challenge: Verification

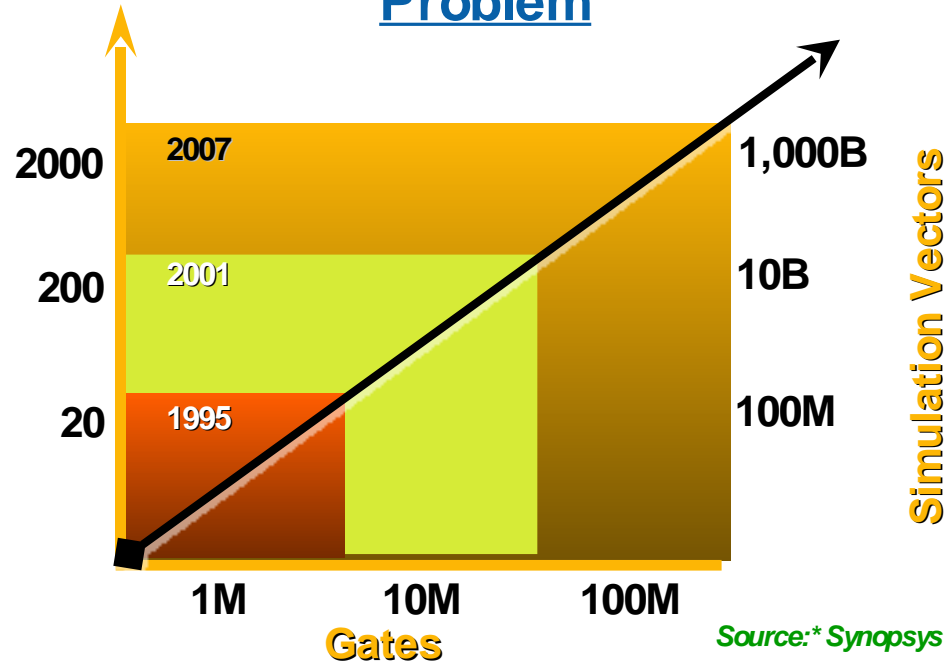
Verification Gap Is Killing Schedules

N. America SoC Statistics



Source: * 2002 Collett International Research and Synopsys

The Designer's Escalating Problem



Source: * Synopsys

What Can We Do?

You Can Do Theorem Proving



Theorem Proving is Practical

Complete correctness of key features is formally verified:

- Floating point units
- Instruction length decoders
- ***We don't just check some properties of them***

We do this with interactive theorem proving

- Using HOL-style theorem provers
- Properties we prove and the designs we prove them of remain beyond automated proof

Cost Effective Theorem Proving Through Volume

Theorem proving seems too expensive

- Proof of the instruction decoder took 6 months

But, it is a fixed cost per product

Cost of recall is not fixed, rises with volume

- Pentium® recall cost \$475M
- volume *much* higher now



300mm Semiconductor Economics

Fab	\$3 billion
Pilot line	\$1–2 billion
R&D process team	\$0.5–1 billion

High volume can bring a \$5 billion investment down to reasonable unit cost



Testing is Also Expensive

Testing the silicon is **very** expensive

- Only arrives late in development process

Simulation is **much** slower than real silicon

- Full-chip simulation with checkers runs ~20Hz
- Our compute farm has ~6K CPUs running 24/7 to get tens of billions of simulation cycles per week
- The sum total of Pentium® 4 simulation cycles prior to first silicon < **1 minute on a single 2 GHz system**



For Software?

Theorem proving is an inexpensive way of reducing recall risk for high volume product

Look for software verification opportunities

- in high-volume products
- where defects might trigger a recall

Examples:

- Microcode
- Consumer Electronics
- Cell Phones



It's Verifying Change That Counts



Good Products Don't Die

Products come in related families and generations

- Families: IA-32 server, desktop and mobile parts
- Generations: Pentium® II, Pentium® III, Pentium® 4

Good theorem proving application is where cost is amortized across product families and generations

The [instruction decoder] verification was ported to a new ... design that used a completely different algorithm. ... the initial ... specification and verification took almost 6 months ..., the port ... took less than 2 weeks

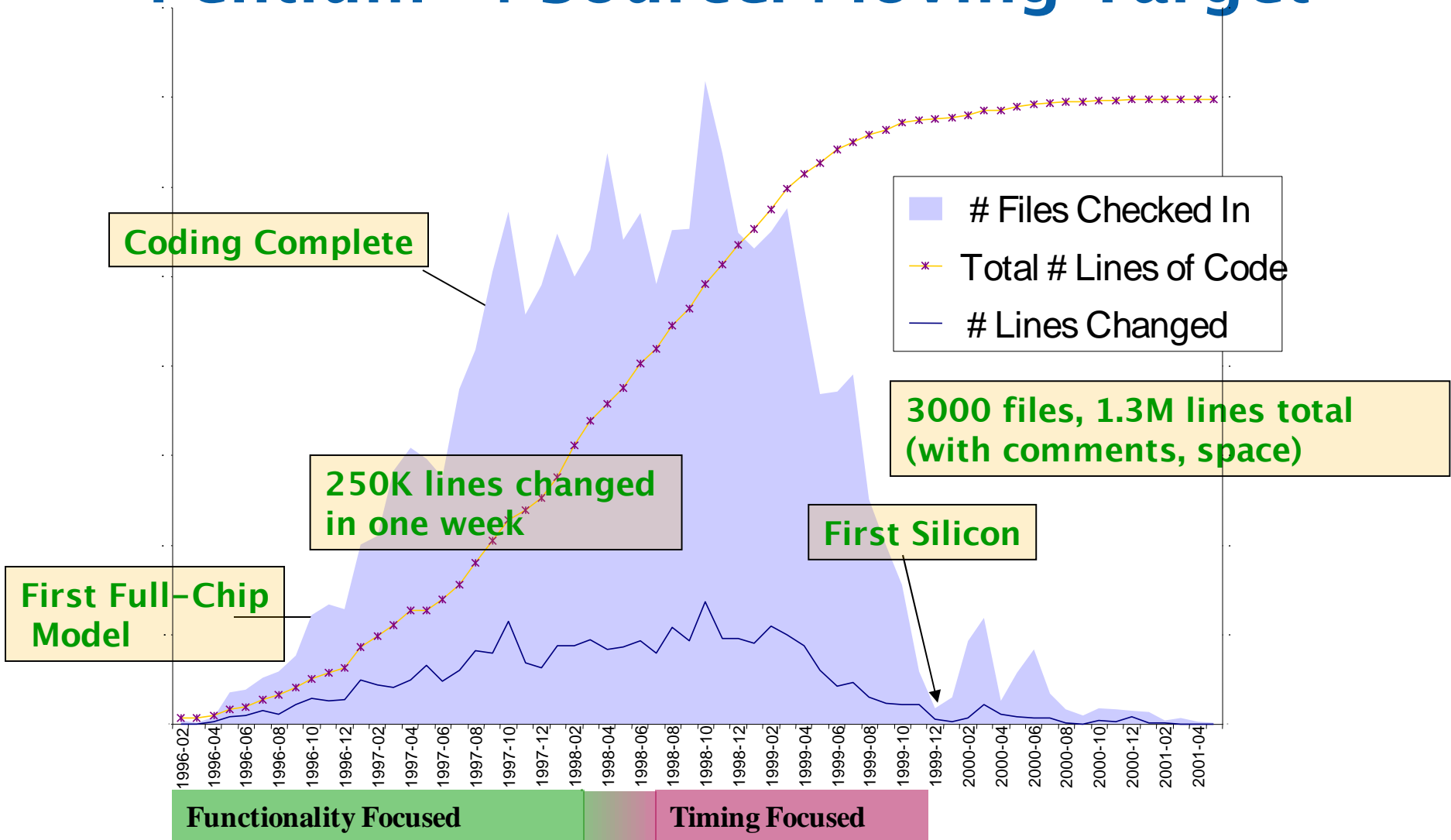
Aagaard, Jones, Kaivola, Kohatsu & Seger

Formal Verification of Iterative Algorithms in Microprocessors

Robust reusable proofs are extremely valuable



Pentium® 4 Source: Moving Target



Coping with Change: Circuit APIs

Interface change is particularly common

- Signals are renamed and re-encoded
- Timing changes (IO set-up, number of pipe-stages change)

We abstract away from these changes

- Develop a stable abstract interface for reasoning
- Develop executable HOL function, the *Circuit API*, connecting the interface and circuit
- Model Check circuit properties with respect to the stable interface
- Prove correctness by composing model checked properties

When the circuit does change

- Build new API connecting the changed circuit to the old interface
- Re-do the model checking runs
- **Interactive proof is reused**



Coping with Change: Partitioning

LTL capacity limits automatic verification to circuits of a few hundred state elements

Decomposition is necessary, but challenging

many difficult ... properties could be verified through clever decompositions. However, the decompositions could be ... complicated ... were often fragile and required complex changes as the [source] evolved

Schubert

High Level Formal Verification of Next-Generation Microprocessors



Partition the Specification

Specifications more stable than implementations

- Proof decompositions based on the IO space of the problem are more robust than those following the implementation

Floating point addition can be decomposed into:

- Exponent of operand B \gg exponent of A
- Exponents of operands =
- Exponent of operand B $>$ exponent of A, by 1, ... n
- Similarly for A and B swapped

Decomposition applies to a wide variety of adders



Change and Software

Software comes in families and generations of related products

Seek validation that copes with change

Locked in progress for when code is reused in another version of Windows

Das

Formal Specifications on Industrial Strength Code – From Myth to Reality

- Contrast with code inspection
 - cost scales with change

Challenge:

- Can we develop circuit APIs for software



There May Be No Specification



There is No *Formal* Specification

My nightmare was that we would build a very fast computer that could not ... run all x86 code properly

Colwell

The Pentium® Chronicles

The Pentium® Pro team had a specification

- Very low level specification, the previous source

Exceptions are rare

- The IEEE floating–point specification



Why There Is No Specification

An abstraction helps us know what to build

But once built, when the change requests start rolling in, it is not maintained

Why?

- It is the implementation that *must* be changed
- It takes effort update the specification
- The effort doesn't help make the change

In the future, when verification might be possible

- There will be no specification to verify against
- Because in the past there was no value in maintaining it



Some Specifications Get Maintained

Specifications that have use during the evolution of a system are maintained

- Example: A software model of a component
 - Can be compared with implementation by testing
 - Enough value to keep the model up to date

User level documentation is maintained

- User level documentation is part of the product
- It will be kept up to date



Specification Research Challenges

Testing can maintain connection between abstract software model and an implementation

- A more formal connection can be established later

Can we promote specification languages that:

- Offer more abstraction than C programs
- Are easier to reason about than C programs
- Offer value during development/maintenance

Readable documentation with a formal semantics?

- A processor manual is full of tables and pseudo-code
- Formal table and pseudo-code languages could help
- Consistency can be approximated through testing



No Spec? No Problem!

Two common sources of change:

- Change resources (area/speed) preserving functionality
- Add new functionality while preserving old

There is a formal spec, the old version

Formal Equivalence Verification (FEV) *extremely* useful

- FEV tells you more than regression testing
- Probably faster than regression testing

FEV for software would be useful

People are scared of introducing a bug.

Das
Formal Specifications on Industrial Strength Code – From Myth to Reality



Design Transformation is Feasible



Design Transformation is Feasible

I once believed in program derivation

a tool which supports ... program development. The tool uses a ... mechanism for transformational reasoning.... A graphical ... interface provides ... menus of transformations and the ability to select and focus on subcomponents

Butler, Grundy, Lånbacka, Rukšėnas, and von Wright

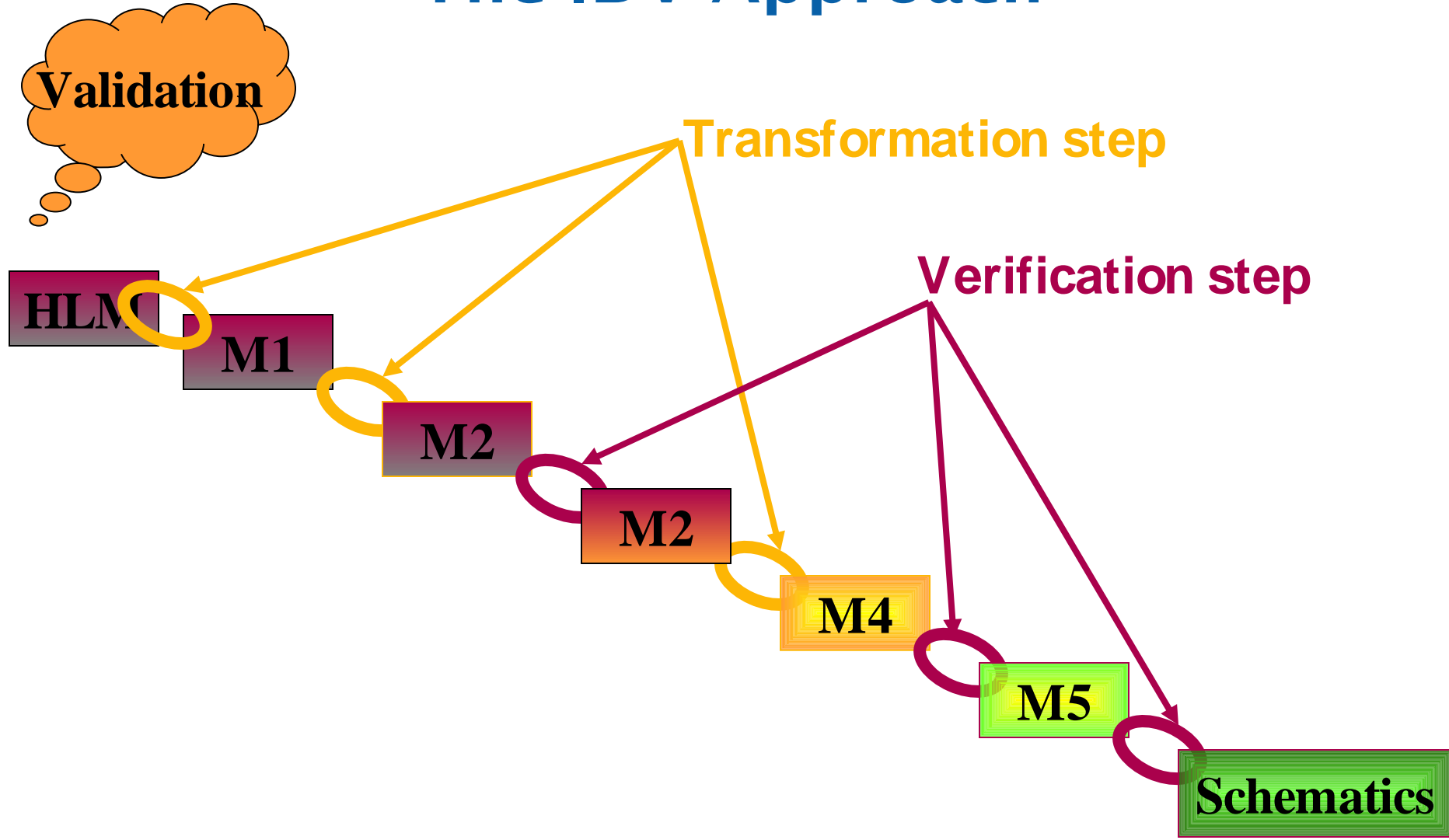
The Refinement Calculator: Proof Support for Program Refinement

I stopped

It is time to believe again



The IDV Approach

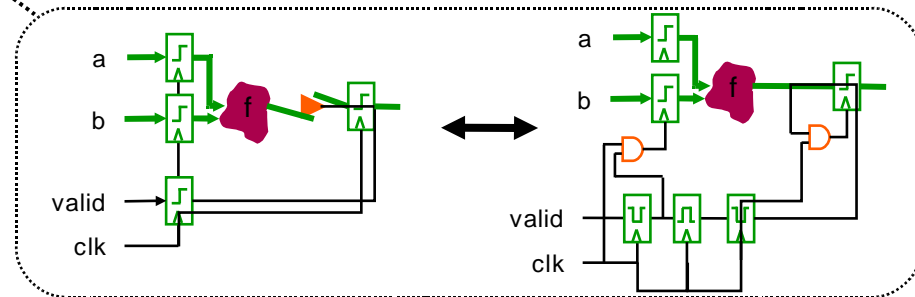
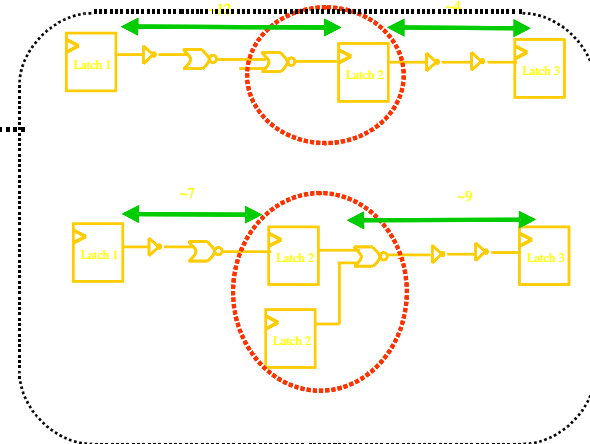
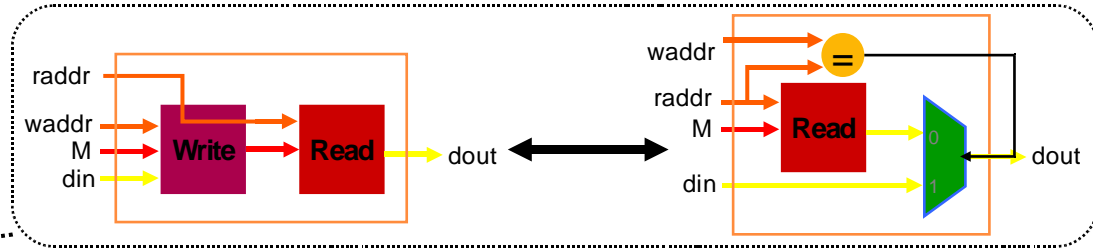


Logical Design Transformations

Add correct-by-construction implementation

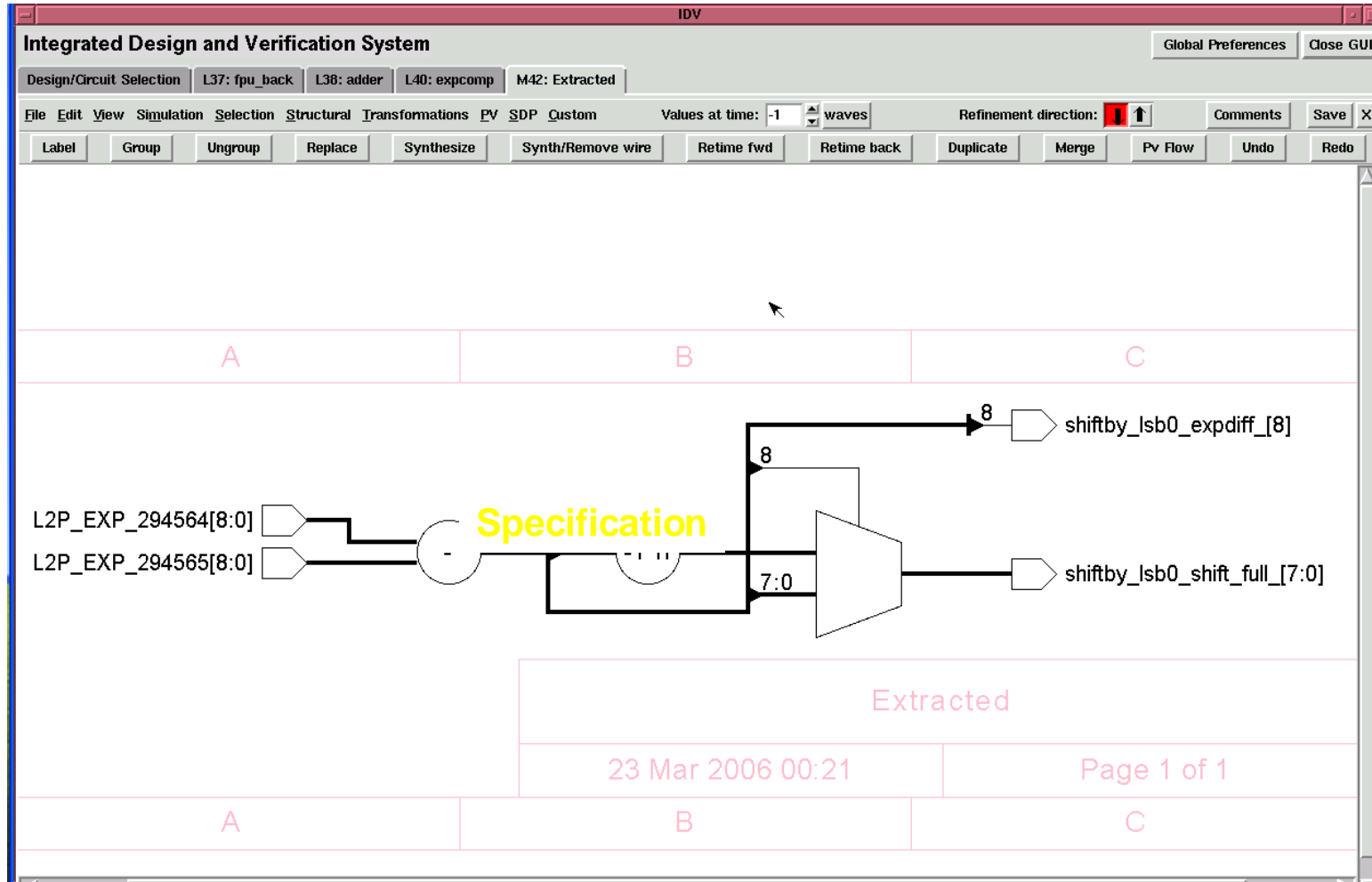
- Bypass
- Re-timing
- Duplication/merging of logic
- Changing state encoding
- Introducing clock gating

Allow arbitrary changes coupled with machine-checked justification

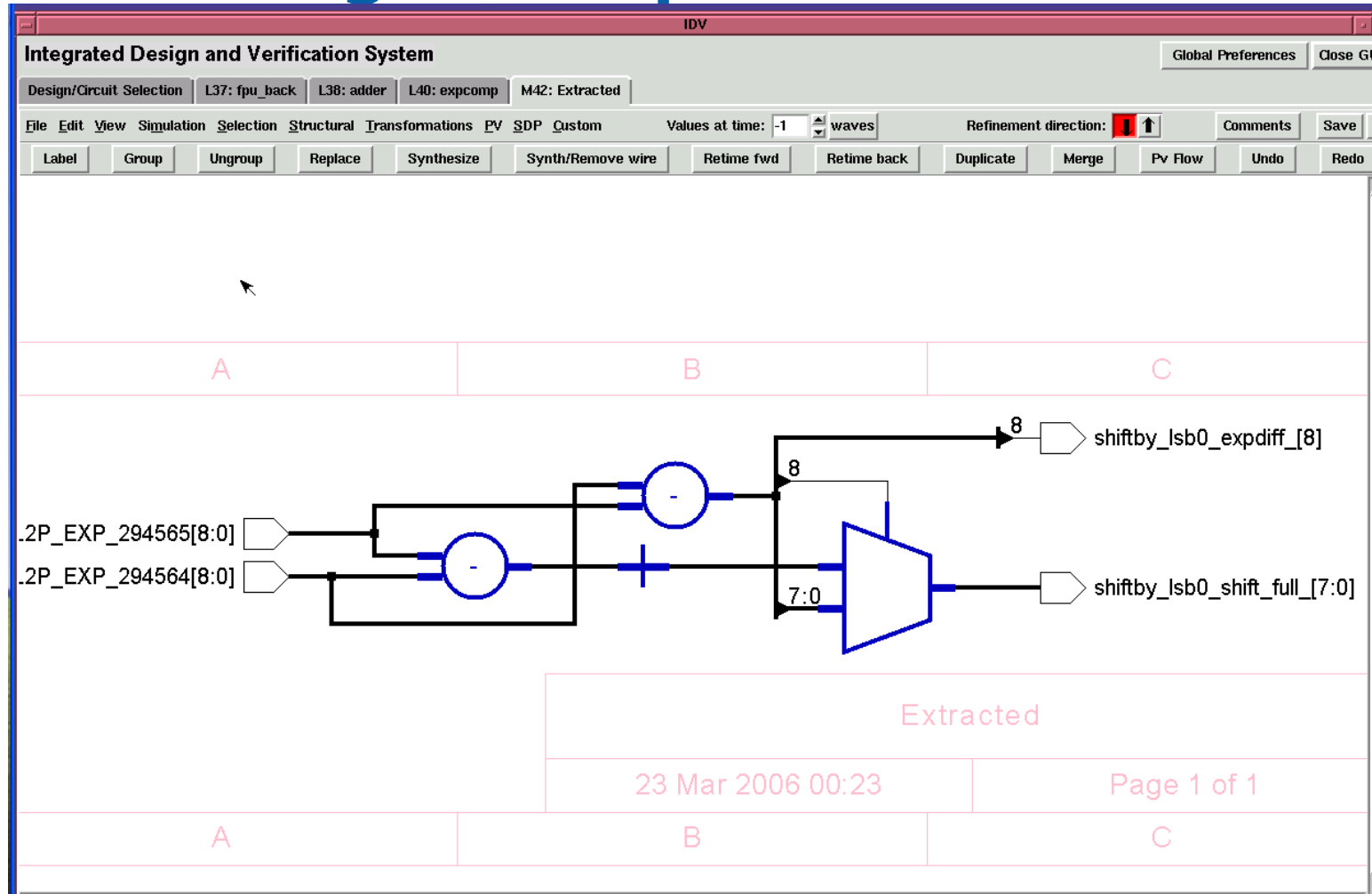


Mid-Level Design in IDV

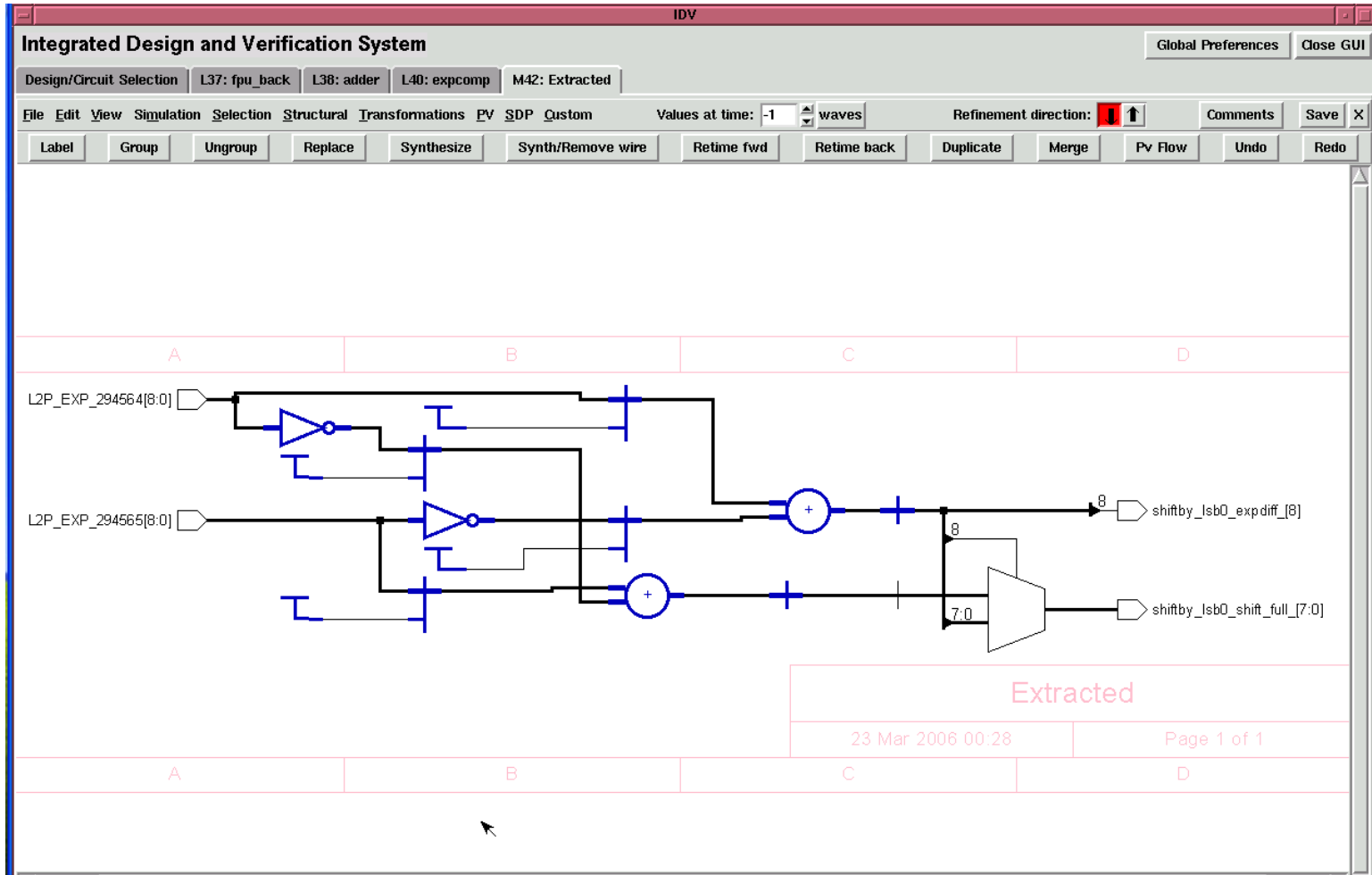
Problem: Subtraction and negation in series!



Step 1: Make new design and FEV against specification



Step 2: Design a subtractor from adder and replace occurrences



Step 3: Get suitable adder from library (speed/power/area) and find/replace

The screenshot shows the Integrated Design and Verification System (IDV) interface. The main window displays a circuit diagram with a red circle highlighting an adder component. A 'Replace' dialog box is open, titled 'Choose replacement design'. The dialog box contains a list of adder options:

- Kogge-Stone adder 308 ps -1 mW 931 mu^2 #5072
- Kogge-Stone adder 326 ps -1 mW 849 mu^2 #5075
- Kogge-Stone adder 419 ps -1 mW 501 mu^2 #5078
- Kogge-Stone adder 510 ps -1 mW 501 mu^2 #5081
- Kogge-Stone adder 582 ps -1 mW 491 mu^2 #5084
- (/home/cseger/Pilot/multiplier/DB_mult_props) add_assoc_10 #91
- (/home/cseger/Pilot/multiplier/DB_addition_properties) a+b == b+a

The dialog box also includes a 'View' button, an 'Order' section with a 'Sort by Date' checkbox, a 'Matching' section with checkboxes for 'Exact matching', 'Remove dangling outputs', 'Replace N', and 'Stloppy', and a 'Keep folded after replacement' checkbox. At the bottom, there are buttons for 'Replace Once', 'Find and Replace', 'Repeatedly Find and Replace', and 'Cancel'. The 'Find and Replace' button is highlighted.

The background circuit diagram shows a multiplier implementation. It includes two 8-bit inputs labeled 'L2P_EXP_294564[8:0]' and 'L2P_EXP_294565[8:0]'. The multiplier is implemented using a shift-and-add algorithm. A red circle highlights an adder component in the multiplier's internal structure. The multiplier's output is labeled 'shiftby_Isb0_shift_full_[7:0]'. The circuit is divided into four regions labeled A, B, C, and D. Region C is labeled 'Extracted' and contains the text '23 Mar 2006 00:31' and 'Page 1 of 1'.

Source: Seger, Memocode 2006



Step 4: Synthesize remaining logic

The screenshot displays the 'Refinement Transform' dialog box in a design tool. The dialog is titled 'Refinement Transform' and has a 'Cancel' button. It is divided into several sections:

- Before:** Name: [empty] View Export
- After:** Name: tmp_1068 View Import Visual Edit Templates Textual Edit
- Synthesis:** LDS LDS Options PC Don't Care Rewrites tabs. Export Filter: Standard. Import Filter: Standard. Mappers: lds_bdm, lds_bdm_w_choice, nomap. Scripts: alg-mfs-delay, ecs, mfs, algebraic, rugged, alg-abc-delay, alg-abc-area, abc-delay, abc-area. Synthesize Interactive Do Batch buttons.
- Verification:** Engine: Scalar Simulation, Symbolic Simulation, SAT, BDD, SEQVER. Verification options: Binary states. Status: ?. State Relation Case Split Verify Counter Ex. buttons.
- Transformation:** Name: transf_5087. Keep folded after replacement. Replace Once Find and Replace Repeatedly Find and Replace Cancel buttons.
- Error messages:**

The background shows a circuit diagram with components like L2P_EXP_294564[8:0] and L2P_EXP_294565[8:0].

Step 5: FEV the result

Refinement Transform

Cancel

Before
Name: View Export

After
Name: tmp_1068 View Import Visual Edit Templates Textual Edit

Synthesis

LDS LDS Options PC Don't Care Rewrites

Export Filter: Standard
Import Filter: Standard

Mappers:
lds_bdm
lds_bdm_w_choice
nomap

Scripts:
alg-mfs-delay
ecs
mfs
algebraic
rugged
alg-abc-delay
alg-abc-area
abc-delay
abc-area

Synthesize
Interactive
Do Batch

Verification

Engine

Scalar Simulation
Symbolic Simulation
SAT Saturation limit: 2 Backtrack limit: 2
BDD Variable Order Dynamic ordering
SEQVER Clockname: clk Time-out: 120

Verification options

Binary states

Status: Y

State Relation
Case Split
Verify
Counter Ex.

Transformation

Name: transf_5087

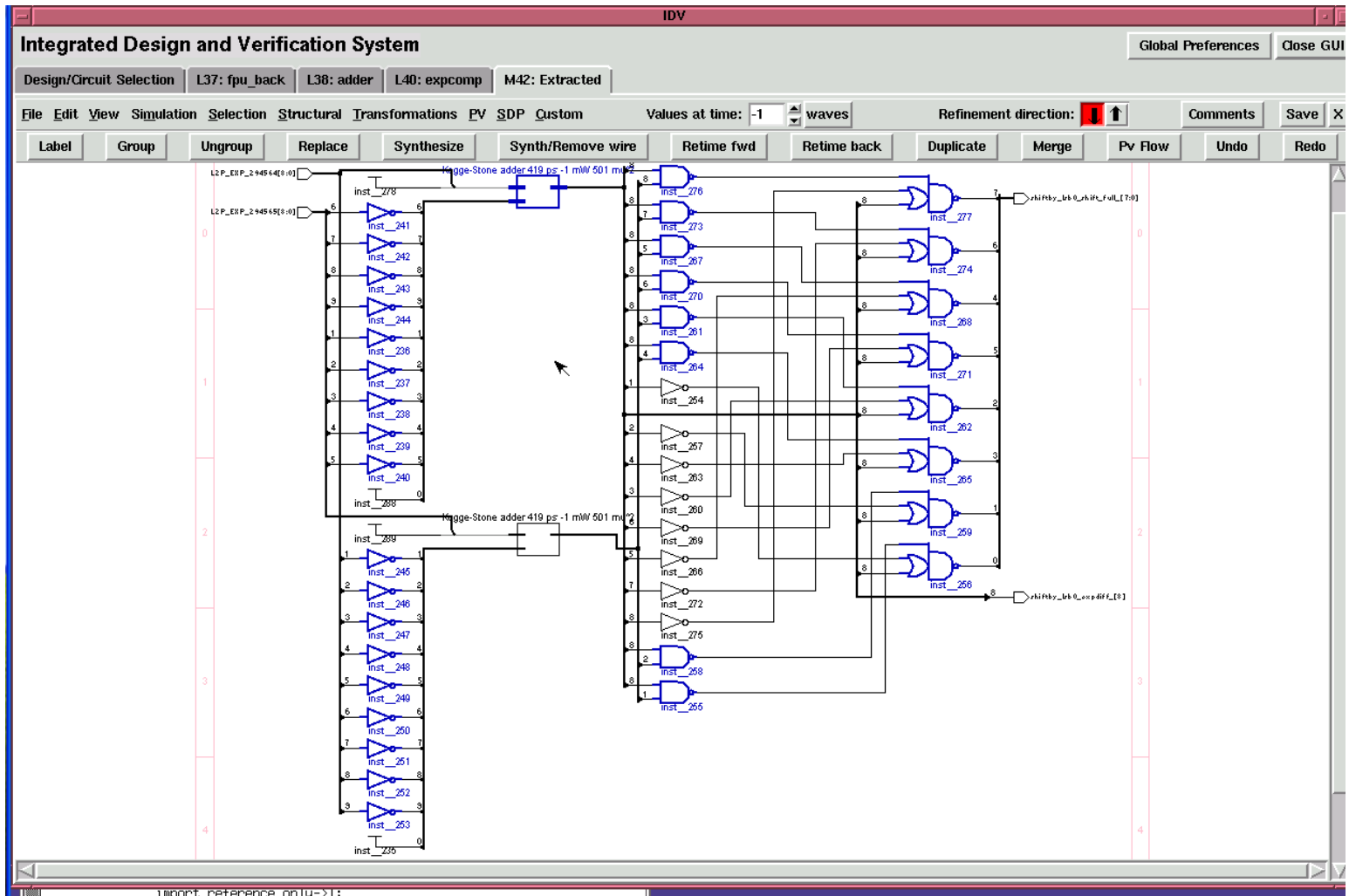
Keep folded after replacement

Replace On Find and Replace Repeatedly Find and Replace Cancel

Error messages

Save this transformation and replace before with after circuit.

This Yields



Source: Seger, Memocode 2006



A Challenge for Software

IDV can apply semantics preserving transformations to existing designs

Can we get a tool like IDV for software?

Perhaps only when FEV for software is capable enough



Summary

Focus on high volume embedded software

- Economics justifies commercial formal verification

It is really about verifying change, not verifying

Develop specification languages with clean semantics that can be used as checkers

- Give developers a reason to maintain specifications

Consider the problem of FEV for software

- It is a key hardware tool, seems under used in software

Interactive proof can be justified

- If you think about how it copes with change

Even transformational design might work



A Grand Challenge Competition

Perhaps ... stimulating competitions would be based on the ... programming competitions already set ... on the web, by ... requiring that the ... program be verified. ... the ... goal would be to develop correct programs just as fast or faster than incorrect ones are today

Hoare & Misra

*Verified Software: Theories, Tools, Experiments
Vision of a Grand Challenge Project*

Economic case for software FV can be made compelling sooner than such competitions would suggest

Here is an idea for an alternative competition

1. Challenge problem set
 - Contestants get a year to develop a verified implementation
2. Problem is changed!
 - 1st to develop a verified implementation of new problem wins



A Really Compelling Software Verification Case Study

Pick a key component of Linux version 1.0

Develop a specification for it

(Debug and) verify it

Then do it again for the next version

And again for the next version

Then repeat on equivalent BSD component



