The FM9001 Microprocessor Verification and Proving Very Large Theorems

Warren A. Hunt, Jr.

Department of Computer Sciences
1 University Station, M/S C0500
The University of Texas
Austin, TX 78712-0233

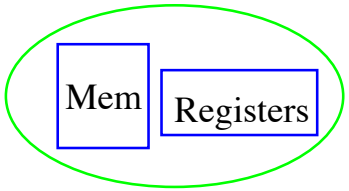
E-mail: hunt@cs.utexas.edu
TEL: +1 512 471 9748
FAX: +1 512 471 8885

Overview of the Talk

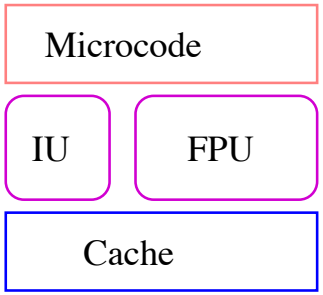
- Levels of abstraction
- The use of logic for validating hardware
- FM9801
- Power Considerations
- Big Theorems
- Conclusion

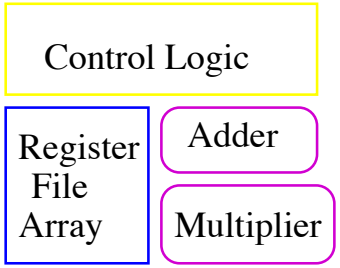
Levels of Abstraction

ISA

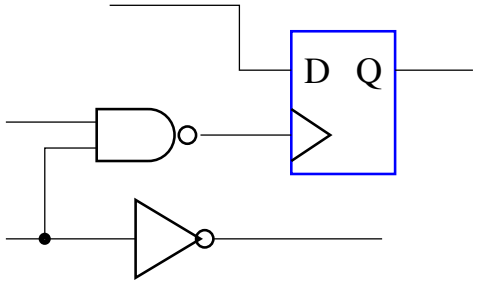


Microarchitecture

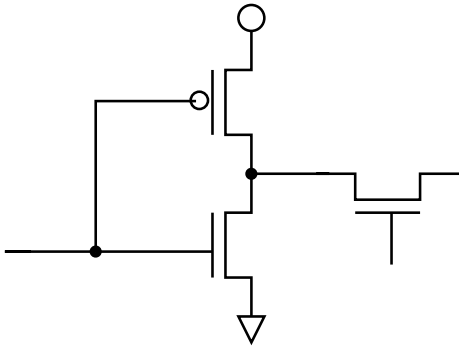




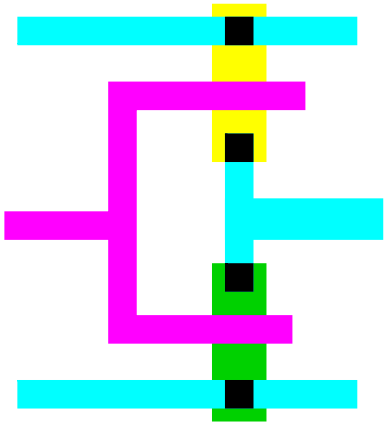
Netlist



Schematic



Layout



- Different specification languages are used at different levels.

The Verification Problem

- Different specification languages are used at each level.
  - ISA: C, C++ models
  - Architecture: Drawings, Charts, Graphs, Natural Language
  - Microarchitectures: More diagrams, charts, etc.
  - Register-transfer: VHDL, Verilog
  - Netlist: VHDL, Verilog
  - Transistor Schematic: "Stick diagrams"
  - Layout: Colored Polygons
- The Size
  - ISA models: hundreds of pages
  - RTL models: thousands of pages
  - Netlist models: millions of pages

Modeling & Microprocessor Calculus

- At the transistor level, modeling with differential equations is appropriate.

$$\int_0^\infty f(t) + \delta(t - t_1)dt + ag(t^2) + \ldots$$

  - Using differential equations is far too detailed except for tiny circuits.
  - Verification is done by simulation: "rectangle approximation."

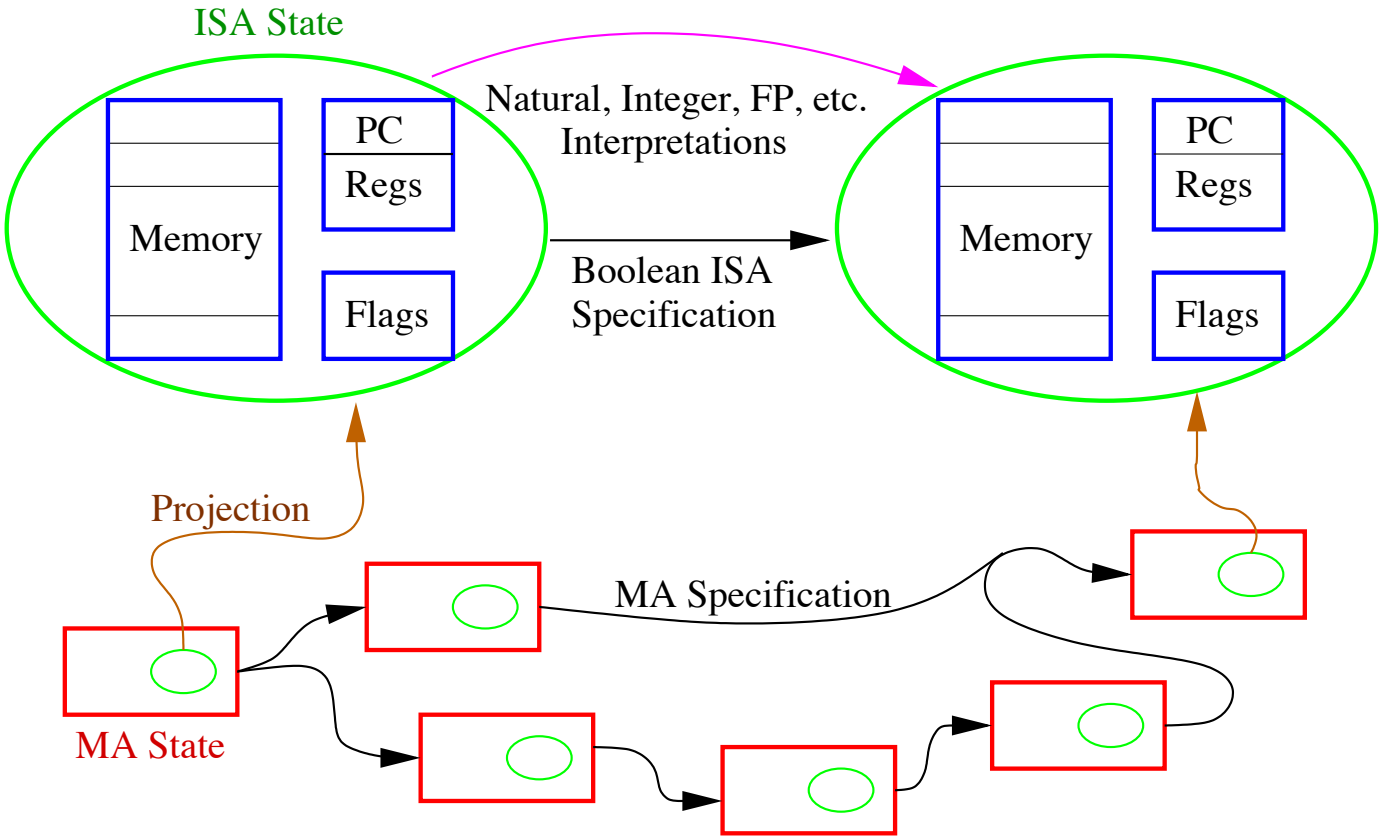
- Using general-purpose logic is *microprocessor calculus*.
  - Example calculi: ACL2, HOL, PVS.
  - Using microprocessor calculus requires direct interaction.

- Logics with algorithmic decision procedures: *microprocessor algebra*.
  - Examples: Equivalence checking, model checking, symbolic simulation.
  - Systems are generally "programmed" by a user, e.g., variable ordering.

- Microprocessor calculus examples: FM8501, FM8502, FM9001, FM9801, and Motorola CAP DSP.
- Array verification is an application of microprocessor algebra.

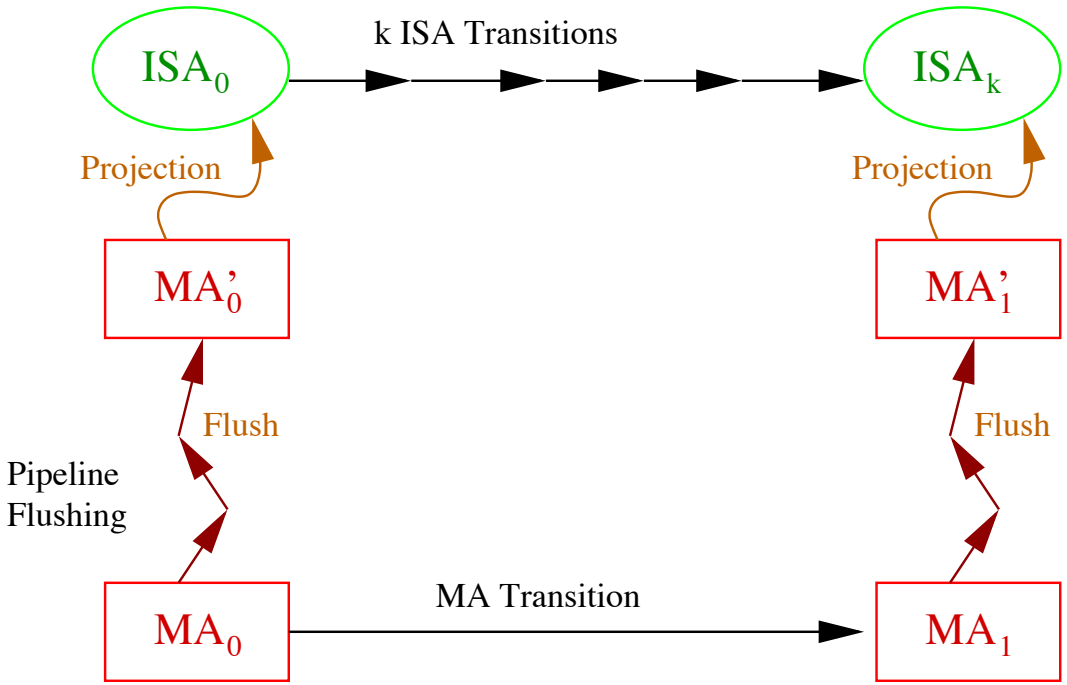
Microprocessor Correctness

- Microprocessor correctness is demonstrated by showing that some microarchitectural design (MA) implements its instruction-set architecture (ISA).
- This kind of verification is an application of microprocessor calculus.



Correctness of Pipelined Microprocessors

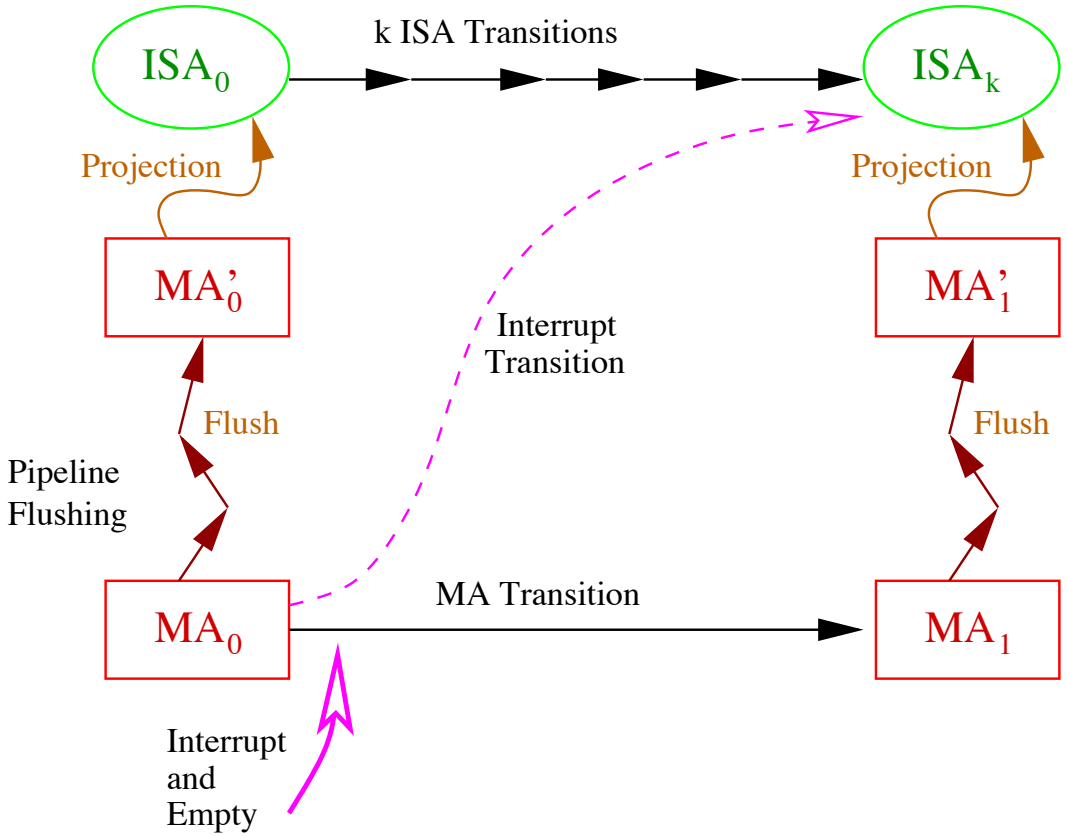
- The verification of pipelined microprocessors requires a more sophisticated abstraction function because of: out-of-order execution, speculative execution, exceptions and interrupts, and self-modifying programs.
- Burch and Dill proposed using the processor's own flushing mechanism as the abstraction function.



- However, this verification approach does not work with interrupts.

Problem with External Interrupts

- When receiving an external interrupt, modern microprocessors flush in-flight work and take the interrupt.

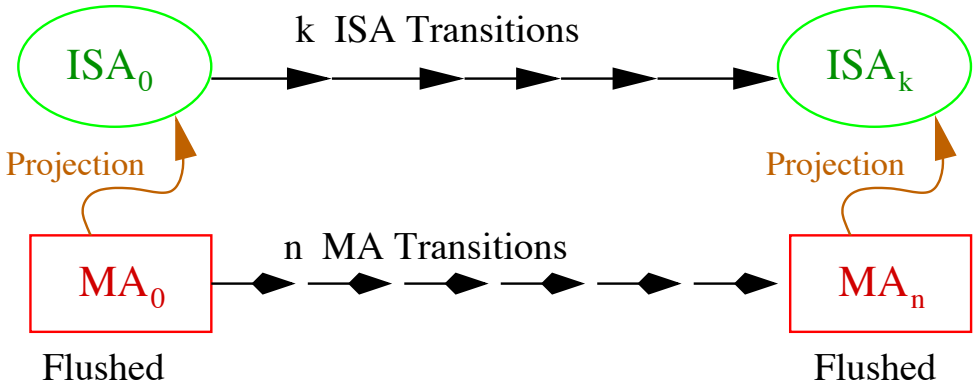


- The Burch and Dill approach does not permit an "empty" the machine flush.

Correctness of Superscaler Microprocessors

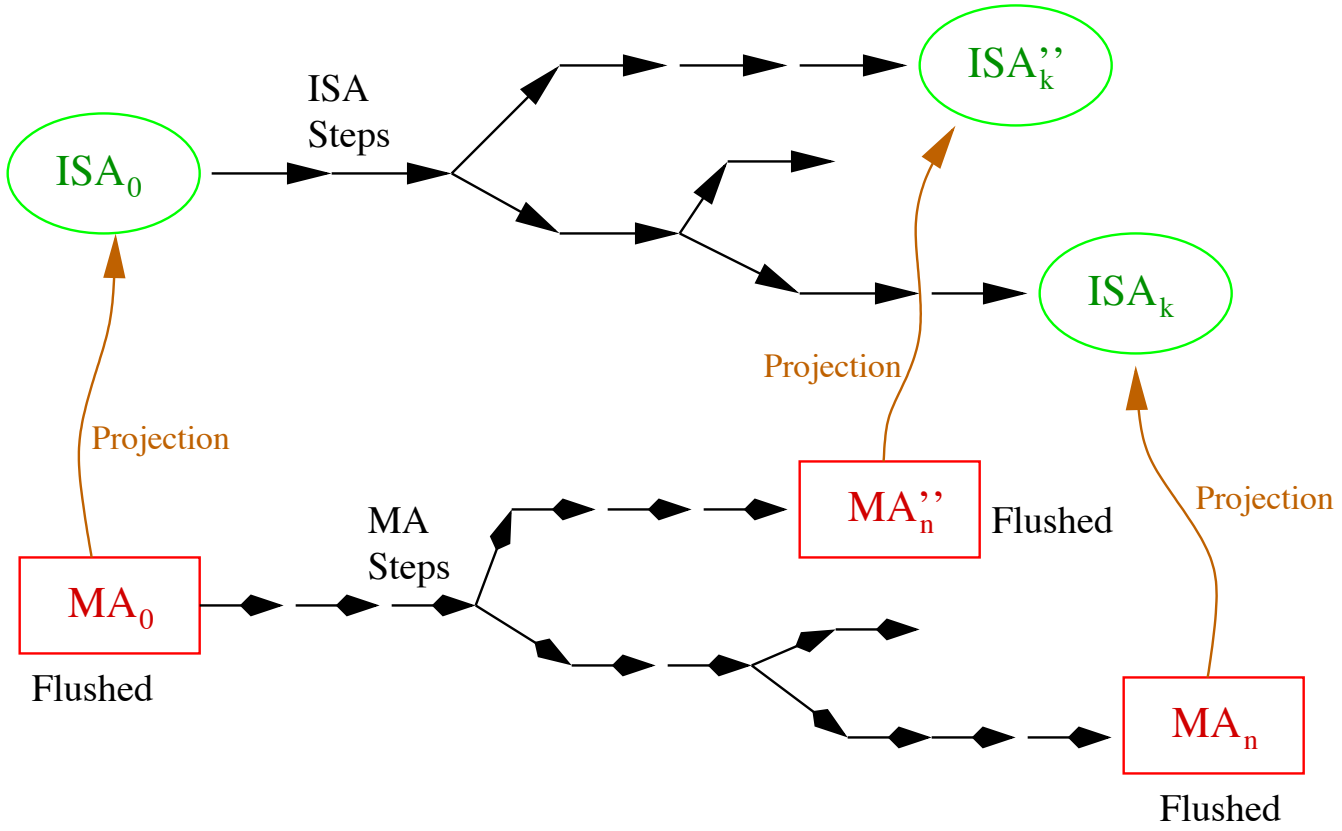
Joint work with Jun Sawada

- The commutative diagram is the basis of our correctness criterion.
  - For n-step MA state transitions, the initial and the final states are flushed.
  - Let m be the number of instructions executed during the MA execution.
  - We compare the n-step transition of the pipelined machine (MA) to the m-step transition of the specification machine (ISA).
  - Additionally, we assume the program does not modify itself.



- This correctness criterion is applicable to out-of-order execution, speculative execution, and internal exceptions, but not to interrupts.

Superscaler Correctness Criterion with External Interrupts

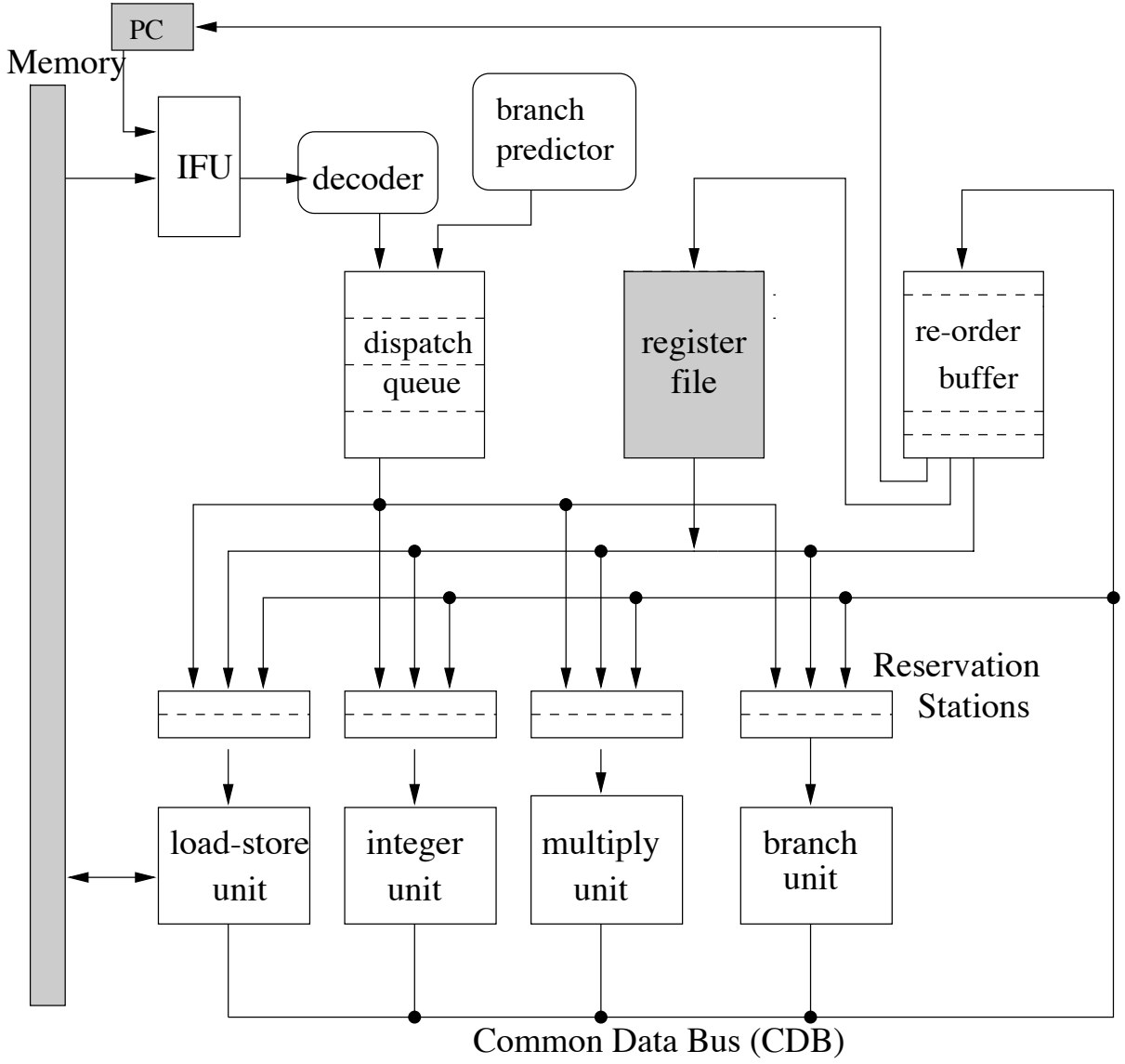


- Branching Behavior implies Multiple MA paths.
- For each MA path, there exists an ISA path that executes and interrupts the same instructions as the MA does.
- This commutative diagram holds for corresponding ISA and MA paths.

The FM9801 Microprocessor

- Our superscaler correctness criterion was used to verify the FM9801 microprocessor defined in Sawada's dissertation.
- The FM9801 microprocessor features:
  - Out-of-order instruction issue & completion using Tomasulo's algorithm.
  - Out-of-order memory accesses.
  - Speculative execution with branch prediction, where up to 11 instructions may be in flight.
  - Internal exceptions and an external interrupt.
- Formally specified in the ACL2 logic.
  - The ISA (specification) and the microarchitecture(implementation).
  - Early debugging by simulation using the ACL2 execution capability.
  - Too complicated for a fully-automated verification.

Block Diagram of FM9801 Implementation



Microarchitecture and Instruction-Set Architecture

- The FM9801 is formally specified at two levels:
  - Instruction-Set Architecture (ISA) is specified with about 900 lines and about 30 functions.
    - Non-pipelined.
    - Executes exactly one instruction every step.
    - Includes only the programmer visible states.
    - Has 11 different classes of instructions.
  - Microarchitecture (MA) is specified with 3300 lines and 170 functions.
    - Pipelined.
    - Clock cycle accurate model.
    - All components are included, including a memory model and branch prediction.
- The goal of verification is to show that the MA (implementation) and the ISA (specification) always compute the same results.

The FM9801 Verification
Mechanical proof done by Jun Sawada

- The entire microprocessor model has been verified with the ACL2 prover.
- Verification Steps
  - **Defined a suitable Intermediate Abstraction – The FM9801 MAETT**
  - Defined and Verified the Invariant Conditions
  - Verified the Correctness Criterion

Intermediate Abstraction MAETT

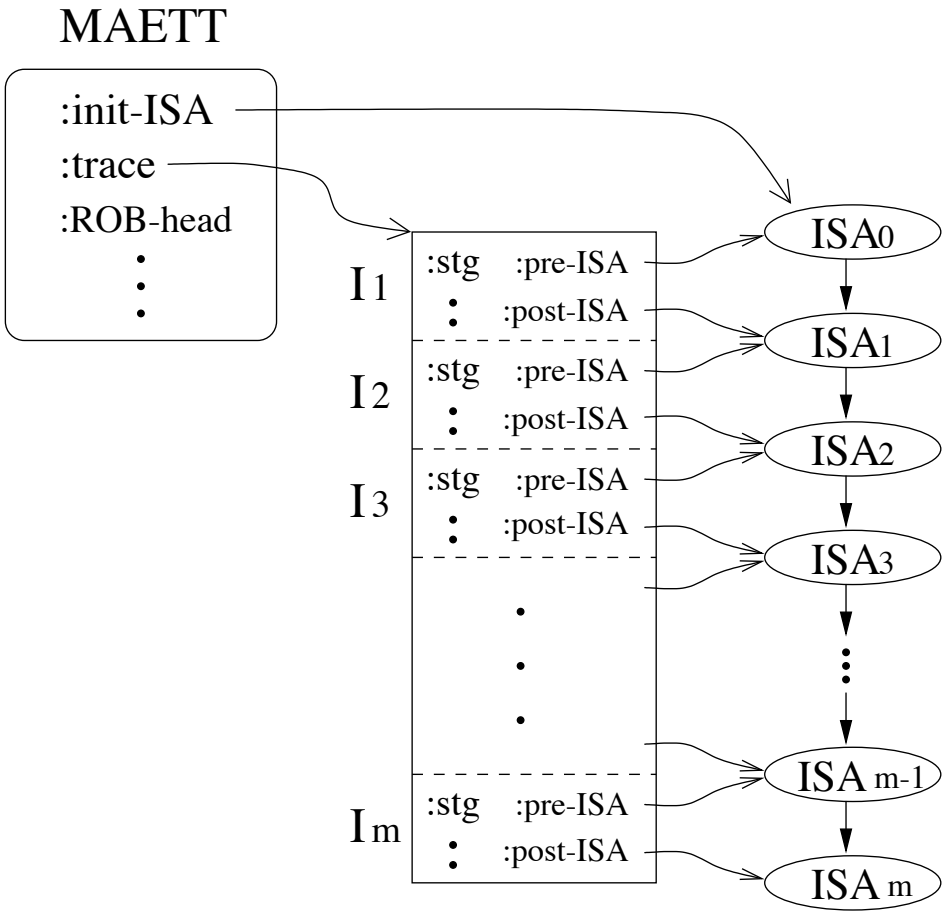
- The correctness criterion was not enough, we needed a mechanism to keep track of instructions as they were processed.
- We introduced the MAETT (Micro-architectural Execution Trace Table) to track the progress of instructions.
  - Each row records the progress of each issued ISA instruction.
  - Evolving columns of the MAETT resembles reservation table entries.

	MA_0	MA_1	MA_2	MA_3	MA_4
i_0		(IFU)	(DQ 0)	(IU RS0)	(complete)
i_1			(IFU)	(DQ 0)	(IU RS1)
i_2				(IFU)	(DQ 0)
i_3					(IFU)

- MAETT records a list of completed and in-flight instructions in program order.

Structure of MAETT

- The MAETT is list where instructions appear in program order, which makes it possible to definite properties as recursive predicates.
- The pre-ISA and post-ISA fields record the ideal ISA execution steps.



Representation of Instructions

- The status of an instruction is represented with a structure:

i_k.stg = '(IFU) ← The current stage of instruction i_k.
i_k.specultv? = 1 ← Instruction i_k is executed speculatively.
i_k.tag ← Tag used in Tomasulo's algorithm
i_k.br-predict = 1 ← Branch prediction result.
i_k.pre-ISA ← The ideal ISA state before executing i_k.
i_k.post-ISA ← The ideal ISA state after executing i_k.

```
Defstructure INST {
    bitp          modified? ;       // Modified by Self-Modifying Code?
    bitp          first-modified? ; // First Modified Instruction
    bitp          speculative? ;    // Speculatively Executed?
    bitp          br-predict? ;     // Branch Prediction Result
    bitp          exintr? ;         // Externally Interrupted
    word-p        word ;            // Instruction Word
    stage-p       stg ;             // Current Stage
    ROB-index-p   tag ;             // Tag used in Tomasulo's Algorithm
    ISA-state-p   pre-ISA ;         // Pre-ISA state
    ISA-state-p   post-ISA ;}       // Post-ISA state
```

Functions and Predicates on Instructions

- Various values of instructions are defined as functions and predicates.
  - The program counter value before executing i_k.
    $$\text{INST-pc}(i_k) = i_k.\text{pre-ISA.pc}$$
  - The memory state before executing i_k.
    $$\text{INST-mem}(i_k) = i_k.\text{pre-ISA.mem}$$
  - The instruction word of i_k.
    $$\text{INST-word}(i_k) = \text{read-mem}(\text{INST-pc}(i_k), \text{INST-mem}(i_k))$$
  - The opcode of i_k
    $$\text{INST-op}(i_k) = \text{INST-word}(i_k).\text{opcode}$$
  - And more..
- We defined 58 such functions and predicates for FM9801.
- Each of these embody a concept of an instruction.

Functions and Predicates on the MAETT

- Functions that takes a MAETT as an argument.
  - For instance, specifying that instruction i preceeds j can be written as a recursive function with MT as an argument.

    $$i \textbf{ precedes } j \textbf{ in } MT$$

- Basic theorems can the be proven about instruction flow in the MA.
  - Transitivity and Antisymmetry of program order.

    THEOREM: INST-in-order-transitivity

    $$((i \textbf{ precedes } j \textbf{ in } MT) \wedge (j \textbf{ precedes } k \textbf{ in } MT)) \wedge \cdots) \rightarrow (i \textbf{ precedes } k \textbf{ in } MT)$$

    THEOREM: INST-in-order-p-total

    $$((\neg\ (j \textbf{ precedes } i \textbf{ in } MT)) \wedge (i \neq j) \wedge \cdots) \rightarrow (i \textbf{ precedes } j \textbf{ in } MT)$$

Verification Steps

- Verification Steps
  - Defining Intermediate Abstraction
  - **Define and Verify Invariant Conditions**
  - Verify the Correctness Criterion

Defining Properties with the MAETT

- To prove our correctness property, we need to know a number of things about our design. We specify such properties using the MAETT.
- For instance, instructions are dispatched and committed in order in the FM9801.
- To estabilish such a fact, we can define $\text{in-order-dispatch-commit-p}(MT)$ using recursion on the list of instructions, $MT.\text{trace} = (i_0\ i_1\ \cdots\ i_m)$.
- Using this predicate, we can establish instruction ordering properties.

THEOREM: INST-in-order-dispatched-undispatched

$$(\text{dispatched-p}\,(i) \wedge (\neg\ \text{dispatched-p}\,(j)) \wedge \cdots) \rightarrow (i\ \textbf{precedes}\ j\ \textbf{in}\ MT)$$

THEOREM: INST-in-order-commit-uncommit

$$(\text{committed-p}\,(i) \wedge (\neg\ \text{committed-p}\,(j)) \wedge \cdots) \rightarrow (i\ \textbf{precedes}\ j\ \textbf{in}\ MT)$$

Examples of Correct Intermediate Values

- The instruction fetch unit(IFU) fetches and stores instructions.
- The field, word, of the IFU stores the instruction word.
- The function $\mathrm{INST\text{-}word}(i)$ represents the correct instruction word for i.
- The correctness of the intermediate value is represented as:

$$(i.\mathrm{specultv?} = 1 \wedge \neg\mathrm{INST\text{-}fetch\text{-}error\text{-}detected\text{-}p}(i) \wedge \cdots)$$
$$\Rightarrow MA.\mathrm{IFU.word} = \mathrm{INST\text{-}word}(i).$$

- The predicate $\mathrm{MT\text{-}INST\text{-}inv}(MT, MA)$ checks all instructions in MT have correct intermediate values in MA.
  - It is defined to be a collection of equalities similar to the one above.

List of Invariant Properties

- We defined invariant properties in 20 predicates.
- Invariants are local properties that can be verified independently of each other.

#	Property Name	Brief Description
0	**weak-invariants:**	A well-formedness predicate for a MAETT.
1	**pc-match-p:**	Correct state of the program counter.
2	**SRF-match-p:**	Correct state of the special register file.
3	**RF-match-p:**	Correct state of the general register file.
4	**mem-match-p:**	Correct state of the memory.
5	**no-speculative-commit-p:**	No speculatively executed instruction commits.
6	**MT-inst-invariants:**	Valid intermediate data values in the pipeline.
7	**correct-speculation-p:**	Instructions following a mis-predicted branch are speculatively executed.
8	**correct-exintr-p:**	Externally interrupted instructions retire immediately.
9	**in-order-dispatch-commit-p:**	Instructions dispatch and commit in program order.
10	**in-order-DQ-p:**	The dispatch queue is a FIFO queue.
11	**in-order-ROB-p:**	The re-order buffer is a FIFO queue.
12	**no-stage-conflict:**	No structural conflict at pipeline stages.
13	**no-robe-conflict:**	No structural conflict in the re-order buffer.
14	**in-order-LSU-inst-p:**	Certain orders are preserved for instructions in the load-store unit.
15	**consistent-RS-p:**	Reservation stations keep track of instruction dependencies.
16	**consistent-reg-tbl-p:**	The register reference table keeps track of the newest instruction that updates each general register.
17	**consistent-sreg-tbl-p:**	The register reference table keeps track of the newest instruction that updates each special register.
18	**consistent-MA-p:**	The conjunction of miscellaneous conditions.
19	**misc-invariants:**	The conjunction of miscellaneous conditions.

Invariant Verification

- We prove the validity of all the invariants listed on the previous slide by induction.
- **Base Case:** Initial pipeline flushed states satisfy inv.

$$\mathrm{flushed?}(MA_0) \Rightarrow \mathrm{inv}(MT_0, MA_0)$$

- **Induction Step:** If inv is true for the current state, it is true for the next state, given that no self-modifying code is executed,

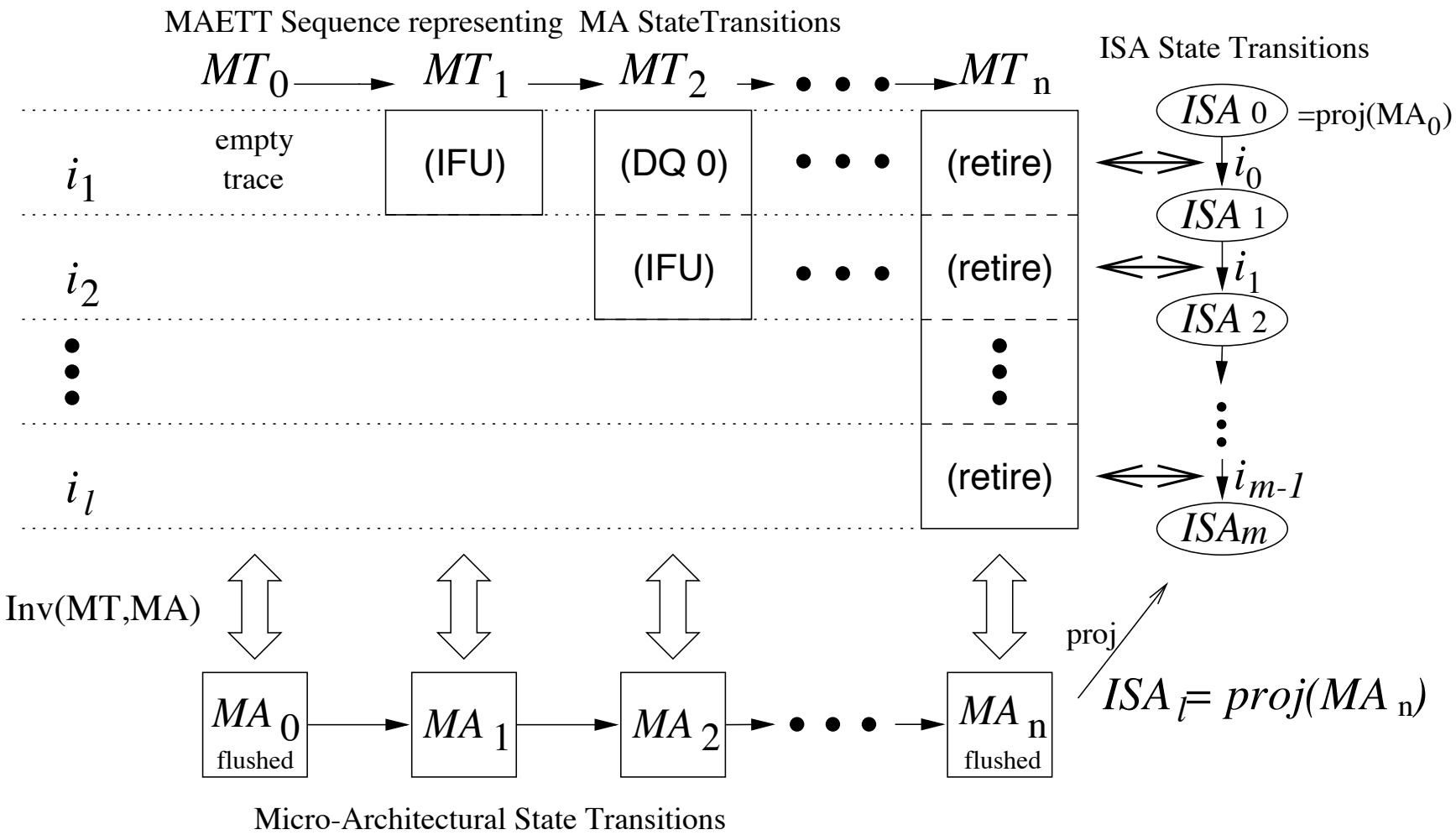
$$\mathrm{inv}(MT_n, MA_n) \Rightarrow \mathrm{inv}(MT_{n+1}, MA_{n+1}) \vee \mathrm{MT\text{-}CMI\text{-}p}(MT_{n+1})$$

  – where predicate $\mathrm{MT\text{-}CMI\text{-}p}(MT)$ is true if self-modifying code is executed and committed.

- Therefore, invariant $\mathrm{inv}(MT, MA)$ is true for all reachable states, as long as no self-modifying code is executed.

$$\mathrm{flushed?}(MA_0) \Rightarrow \mathrm{inv}(MT_n, MA_n) \vee \mathrm{MT\text{-}CMI\text{-}p}(MT_n)$$

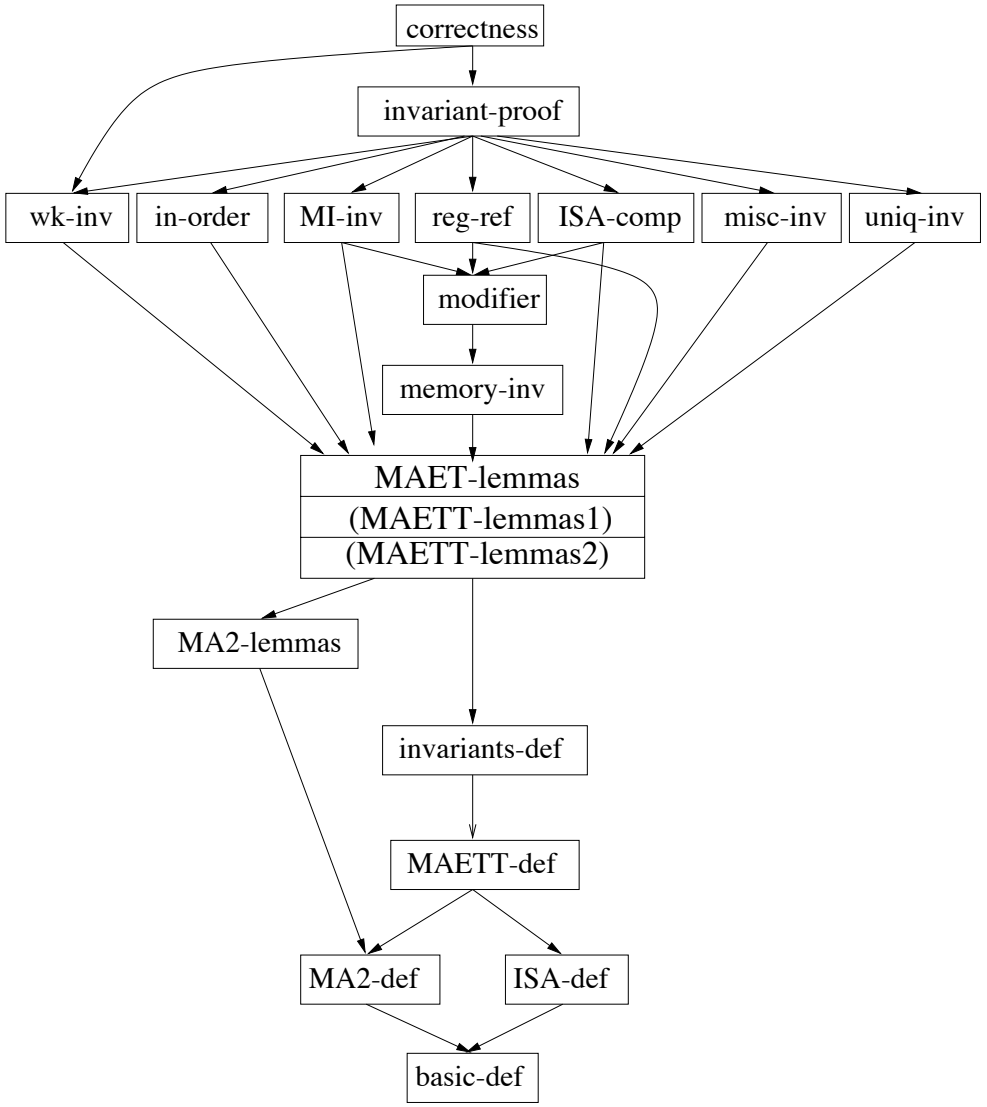
Pictorial Proof of the Correctness Criterion

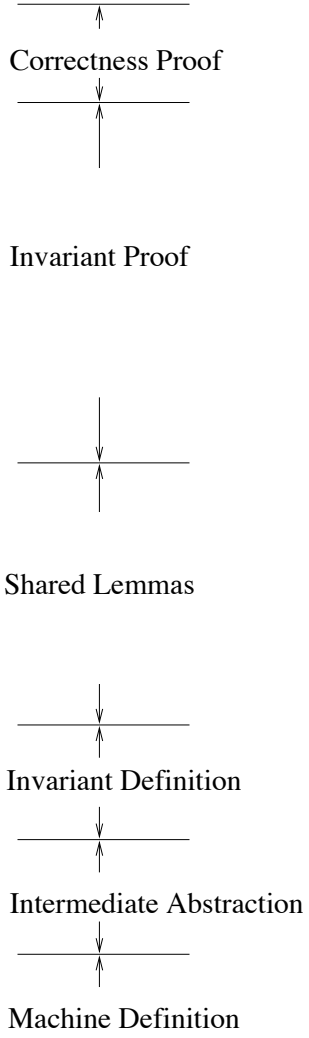


Proof Decomposition

- Temporal Decomposition
  - The correctness criterion involves n-step MA state transitions.
  - The verification of an invariant involves a single step analysis.
  - Avoiding the direct verification of the criterion reduces the cost.
- Spatial Decomposition
  - Invariant proof is divided into the proof of many properties.
  - Each property is related to a few components in the entire architecture.
  - Verifying properties individually reduces the cost.
- Because of the one-step invariants, we could use DUAL-EVAL to implement the FM9801.

Hierarchy of FM9801 Verification Scripts





The Cost of the Verification

- The FM9801 is verified exclusively using the ACL2 theorem prover.
- The proof script can be re-certified in few hours.
- It seems to scale well with respect to the machine size.

Type of ACL2 Script	ACL2 Script Size	CPU Time to Certify
Definitions of ISA and MA	140 KBytes	14 minutes
MAETT modeling	55 KBytes	6 minutes
Definitions of Our Invariant	89 KBytes	7 minutes
Proof of Shared Lemmas	481 KBytes	58 minutes
Proof of Our Invariant	1034 KBytes	211 minutes
Proof of Criterion	37 KBytes	11 minutes

Verified Machine	Machine Spec	Total Verification
Small Example Machine	13 KBytes	169 KBytes
Pipelined Design presented in CAV '97	78 KBytes	757 KBytes
FM9801	140 KBytes	1909 KBytes

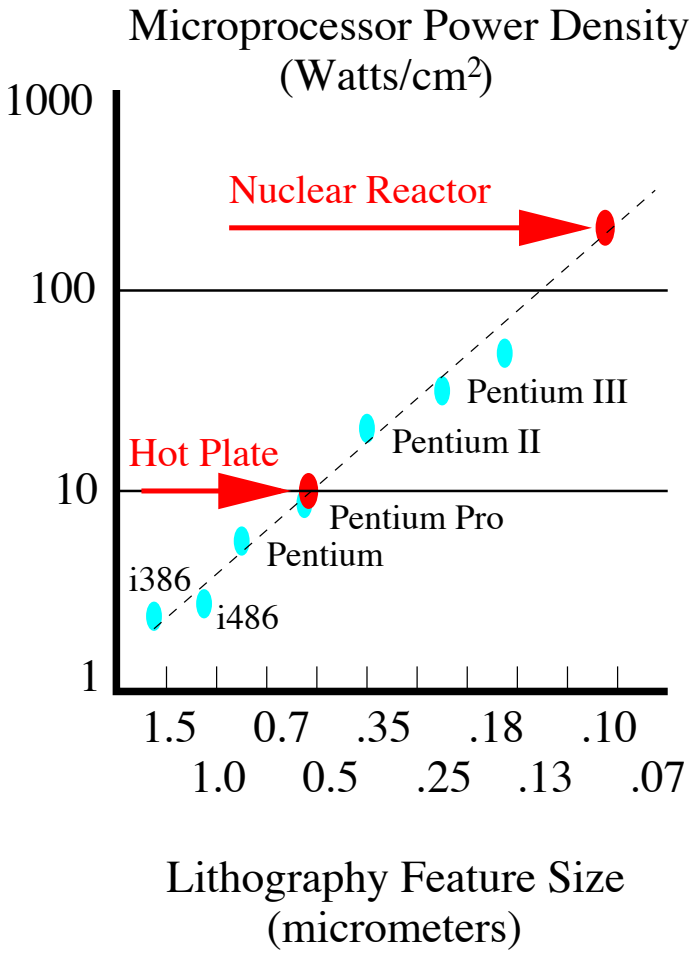
- It would be very interesting to see how much of the invariant proof effort could be automated with algorithmic proof techniques.
  - The invariants properties are of the form: AXp.

Detected Design Faults

- We simulated the machine before starting the formal verification and eliminated most of the bugs.
- Found 12 bugs and 2 glitches during the formal verification process.
  - Bugs are design faults that cause visible incorrect behaviors.
  - Glitches may not cause visible wrong behaviors.
    - One of the glitches may have caused performance degradation.
  - All bugs were found during the verification of the invariants.
- Bugs were found in
  - Branch predictor (Leads to incorrect speculative execution.)
  - Decoder
  - Reservation station
  - Load-Store Unit
  - Multiply Unit

Combined Power & Functional Specifications

- The power density of microprocessor is now first-order problem.
- We are finding ways to trade power for performance on small circuit elements.
  - Greater use of asynchronous and self-timed circuits.
  - Circuits with different number of clock cycles.
- We have initiated a research program to combine functional and power specifications into a single language.
- Functional circuit verification will now require knowing the voltage as well as the netlist.



Source: F. Pollack, Intel, New Microprocessor Challenges in Coming Generations of CMOS Technologies, Micro32

Hardware Verification Theorems Are Large

- Hardware design theorems may be the largest theorems ever proven.
  - The microprocessor correctness statements require more than 100 pages to state.
  - The correctness statement for some of the arrays we have verified require more than 1000 pages to state.
- When it is possible to use proof, the payback is great.
  - It is clear what is known.
  - It is much faster than simulation.

Hardware Verification Requires Yet Larger Theorems

- Theorems involving computer hardware (and software) are enormous.
  - Recent IBM Power 4 (Regatta) design:
    - 170,000,000 transistors,
    - 30,000 pages of RTL, and
    - ISA simulator is 100s of pages.
- Goal: to prove the correctness of designs the size of Power 4.
  - Will require support of many branches of computing science:
    - New theories, new algorithms, and new data representations;
    - Visualization of proofs, automated counter examples generation;
    - Networks of fault-tolerant computing (proof) systems;
    - Architectural, operating system, and database support; and
    - Development of hardware and software theory libraries.
- This goal will necessarily involve group cooperation.

Conclusion

- Hardware verification is technically challenging – designers create research problems for us much faster than we can solve them.
- We need to be able to prove theorems that are four to five orders of magnitude larger than those we now prove just to match what is currently being built.
- There are many PhD dissertations waiting for interested students.
- Hardware verification is important – it is often the least costly method to establish correctness.
  - Comparing equations is cheaper than comparing simulations of equations.
  - Functional verification is more than 30% of an industrial design effort.
  - Hardware verification provides a means to reduce cost while increasing coverage.
- The beauty of mathematics is that it can scale to meet these needs.

Further Reading

– *Microprocessor Verification*, (editor) special issue of "Formal Methods in Systems Design," Kluwer Academic Publishers, March 2002.

– "Verifying the FM9801 Microarchitecture," with Jun Sawada, in *IEEE Micro*, IEEE Press, pp. 47–55, May-June, 1999.

– "Formal Analysis of the Motorola CAP DSP," with Bishop C. Brock, in *Industrial-Strength Formal Methods*, edited by Mike Hinchey and Jonathan Bowen, Springer-Verlag, 1999.

– "The `DUAL-EVAL` Hardware Description Language and Its Use in the Formal Specification and Verification of the FM9001 Microprocessor," with Bishop C. Brock, in *Formal Methods in Systems Design*, Volume 11, pp. 71–105, Kluwer Academic Publishers, 1997.