FORMAL VERIFICATION OF PROCESSORS
AND LOW LEVEL SOFTWARE

W. J. Paul

wjp@cs.uni-sb.de

May 25, 2002

Joint work with
Berg, Beyer, Even, Hillebrand, Jacobi, Knuth,
Kröning, Leister, Müller, Preiß, Seidel
MOTIVATION FOR FORMAL VERIFICATION

- lives (or worse money) depend on correctness
- classical debugging of systems too slow ($2^{128}$ different inputs...)
- classical debugging of proofs too slow (decades)
- formal verification is not terribly slow
A WORD ON TOOLS FOR VERIFICATION

- verification systems capable of verifying real computer systems exist
  - not easy to use
  - high salary ...
- verification sometimes slow because high school mathematics etc. not yet formalized (temporary)
- model checking (checking \(2^{32}\) but not \(2^{64}\) cases) fast special case of theorem proving
THE MAIN TECHNICAL PROBLEMS IN VERIFICATION OF REAL SYSTEMS

- reengineer computer systems in a structured way (without it you still can model check ...)
- formulate correctness of modules
- proofs of lemmas usually easy (as usual)
- automate as much as possible for real (not toy) systems
OVERVIEW

structuring the construction of a parallel processor

1. arithmetic circuits
2. sequential control
3. pipelining
4. forwarding and stalling
5. speculation
6. interrupts
7. out of order execution
8. caches
9. memory management units
10. virtual shared memory
1. ARITHMETIC CIRCUITS (INFORMATIK 2)

- numbers and their representation
- semantics of circuits (every node has a depth ...)

\[
\text{adder: } \langle a \rangle + \langle b \rangle + c_{in} = \langle s \rangle
\]
1. **ARITHMETIC CIRCUITS (Computer Architecture)**

- Def: finite set $R$ of IEEE-representable numbers
- Def: $r : \mathbb{R} \to R \cup \{-\infty, \infty\}$ (Rounding)
- Spec: $\circ_{\text{IEEE}}(x, y) = r(x \circ y), \circ \in \{+, -, *, \ldots\}$
- exceptions ...
- 220 pages of NICE mathematics
2. Sequential Control (Informatik 2)

- construction: finite precomputed control (tens of simple boolean circuits)
- correctness:
  - implementation machine: the hardware
  - specification machine: from specification of machine language

Example: add

\[ RD := RS_1 + RS_2; \quad PC = PC + 4 \]
\[ GPR(RD) := GPR(RS_1) + GPR(RS_2) \]
\[ [GPR(RD)] := [GPR(RS_1)] + [GPR(RS_2)] \pmod{32} \]
\[ [c^{i+1} \cdot GPR(RD)] = [c^i \cdot GPR(RS_1)] + \ldots \pmod{32}, \]
\[ RS_1 := (M(c^i \cdot PC)[25:21]) \]

- Simulation theorem: computer algebra
3. Pipelining

- DP/Con: $k$-stage data path and precomputed control
- $L$: length of longest backward edge
- $u_{e_k}$: update enable signal for stage $k$
3. Pipelining

Correctness:

**Hypothesis:** \( I_i \) generates data \( \Rightarrow I_{i+1}, \ldots I_{i+L} \) do not use it.

**Statement:** \( M_{pipe} \) simulates \( M_{seq} \)

**Proof:**

- \( \forall T \forall k: \) pipelined stage \( k \) has in cycle \( T \) same input as seq stage \( k \) in corresponding (scheduling) cycle.
- Induction on \( T \).
- corresponding stages have in corresponding cycles (scheduling) identical inputs
4. Forwarding and Stalling

R4 = R1 + R5

R1 = R2 + R3

Forwarding

W. J. Paul: Formal Verification of Processors and Low Level Software
4. FORWARDING AND STALLING

R4 = R1 + R5

R1 = M[R2]
4. FORWARDING AND STALLING

\[ R4 = R1 + R5 \]

\[ R1 = M[R2] \]

stalling

empty
4. FORWARDING AND STALLING

- $hazard_i$: stage $i$ should not be updated
- $full_i$: stage $i$ has data

\[
\begin{align*}
stall_i & := full_i \land (hazard_i \lor stall_{i+1}) \\
ue_i & := full_i \land \overline{stall_i} \\
full_i & := ue_i \lor (full_i \land \overline{ue_{i+1}})
\end{align*}
\]

(2000)
4. FORWARDING AND STALLING

Correctness:

Statement: \( M_{pipe} \) simulates \( M_{seq} \)

Proof: \( \forall T \forall k: \) pipelined stage \( k \) has in cycle \( T \) same input as sequential stage \( k \) in corresponding (scheduling) cycle.
\( T \rightarrow T + 1: \) in order \( K, K - 1, K - 2, \ldots \)

Remark: superpipelining:

- pipeline RAMs
- reduce circuit depth of stall engines to \( O(1) \) indep. of \( K \)
- pipeline forwarding
5. Speculation

guess future data, say in stage 1
- predict branch not taken
- predict there will be no interrupt

\[
\text{flush}_i := \text{misspec}_i \lor \text{flush}_{i+1}
\]

\[
\text{full}_i := \overline{\text{old}} \land \overline{\text{flush}_i}
\]

\[
\text{restart} := \text{flush}_1
\]

**Correctness:** 2 cases in simulation theorem (flush/no flush)
6. (Nested maskable precise) Interrupts

very complex
restart:

- save (old or new) interrupt mask bits
- save interrupt cause
- save exception data
- mask all interrupts
- save old PC (or short history of PC’s)
- PC := SISR (start interrupt service routine)

similar rfe (return from exception) instruction

- restore interrupt mask bits.
6. (Nested maskable precise) Interrupts

Complication:

- special purpose register file (for save, restore)
- move instruction between general purpose and special registers

Correctness of hardware: no further complication
6. (Nested maskable precise) Interrupts

Correctness of specification:

Hypothesis: 2 pages specification for interrupt service routines

Statement: jump to interrupt service routine is ’like’ procedure call

Formalization: semantics of programming languages

Proof: show that calls and returns are atomic (except reset)
- 9 pages mathematics

formal proof? today no
would you bet your head on it??
7. OUT OF ORDER EXECUTION

Tomasulo scheduler

- classical benchmark problem for verification
- usually simplified
- usually not synthesizable hardware
- usually not integrated into entire processor
- liveness nontrivial
8. CACHES

M: main memory, big and slow
C: cache, small and fast
B: bus!!, has protocol

![Diagram of cache system with CPU, cache (C), and main memory (M) connected by bus (B)]

- MAd: 1st address
- burst
- w/r
- req
- reqp
- Brdy
- MDat
  - D
  - D1
  - D2
  - D3
  - D4

A, D, w

Formal Verification of Processors and Low Level Software
8. CACHES

- Automata for cache, main memory implement protocol

**Lemma (complex!):** automata implement protocol

**Proof:** presently theorem proving

better model checking?
Correctness statement (extremely simple!):

Cache + main memory behave like uniform memory

\[ m(x) := \begin{cases} 
D & \text{if } x = \langle A \rangle \text{ and } w = 1 \\
\text{otherwise} & \text{otherwise}
\end{cases} \]
8. CACHES

- instruction cache (stage 0)
- data cache (stage 3)

<table>
<thead>
<tr>
<th>Stage</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>I-Cache</td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>D-Cache</td>
</tr>
</tbody>
</table>

Cache coherence requires speculation:
no self modifying code within 3 instructions

$I_i, I_{i+1}, I_{i+2}$ do not write $m(c^{i+2} . PC')$
9. MEMORY MANAGEMENT UNITS

Tasks:

- isolation of user tasks
- memory relocation (zero copy)
- virtual memory

M: main memory, small, fast
DM: disk memory, large, slow
9. MEMORY MANAGEMENT UNITS

M: is cache for DM
PT: page table, 'cache directory' for M
TLB: table look aside buffer, cache for PT

’cache miss’ in M $\rightarrow$ page fault interrupt
$\rightarrow$ interrupt service routine (ISR) $\rightarrow$ rfe

Correctness statements:

- TLB, PT: CPU sees uniform page table
- M, VM, ISR: user task sees uniform virtual memory

Complicated correctness of interrupt mechanism is lemma, hidden (!!!) in correctness statement of M
10. VIRTUAL SHARED MEMORY

SMM: shared main memory, sequentially consistent
(cache based or SB-PRAM)

SYNC: tracks $CP = \{(p, i) |$ processor $p$ currently accesses page $i\}$

ISR for page fault plans to evicts page $i$

$\leadsto$ SYNC stalls ISR until all $(p, i) \in CP$ have completed access

$\leadsto$ ISR invalidates PT(i) $\leadsto$ …

Correctness: processors see sequentially consistent virtual shared memory
This opens the way of proving the correctness of a mainframe!