Refinement and Theorem Proving

Panagiotis (Pete) Manolios
College of Computing
Georgia Institute of Technology

Supported by: NSF (grants 0429924, 0417413, 0438871),
Boeing & IBM

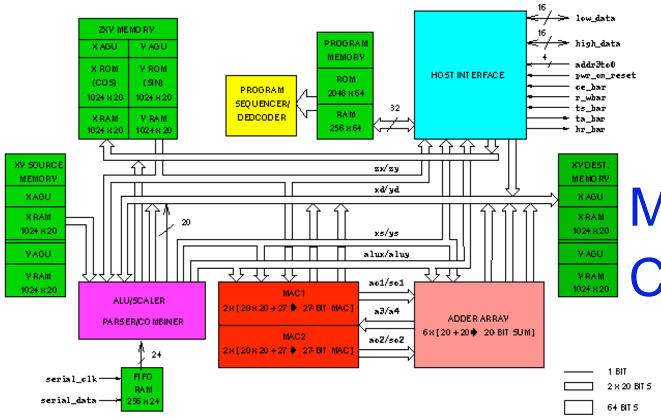
Joint work with: Peter Dillinger, Roma Kane, Sudarshan Srinivasan, Gayatri Subramanian, Daron Vroon, and Yimin Zhang

SFM Bertinoro, Italy May, 2006

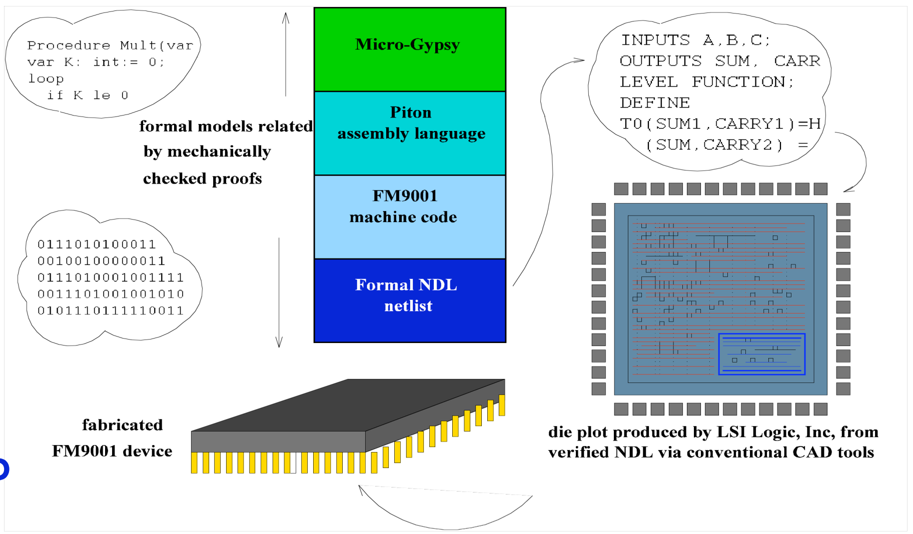
The ACL2 Theorem Prover

- ACL2 theorem prover.
  - ACM 2005 Software System Award.
  - Used to prove some of the most complex theorems ever proved about commercial systems.
- Rockwell Collins AAMP7.
  - MILS EAL-7 certification from NSA for their crypto processor.
  - Verified separation kernel.
- AMD Floating Point, … .

A COMPUTATIONAL LOGIC
ACL2
APPLICATIVE COMMON LISP



Motorola CAP DSP



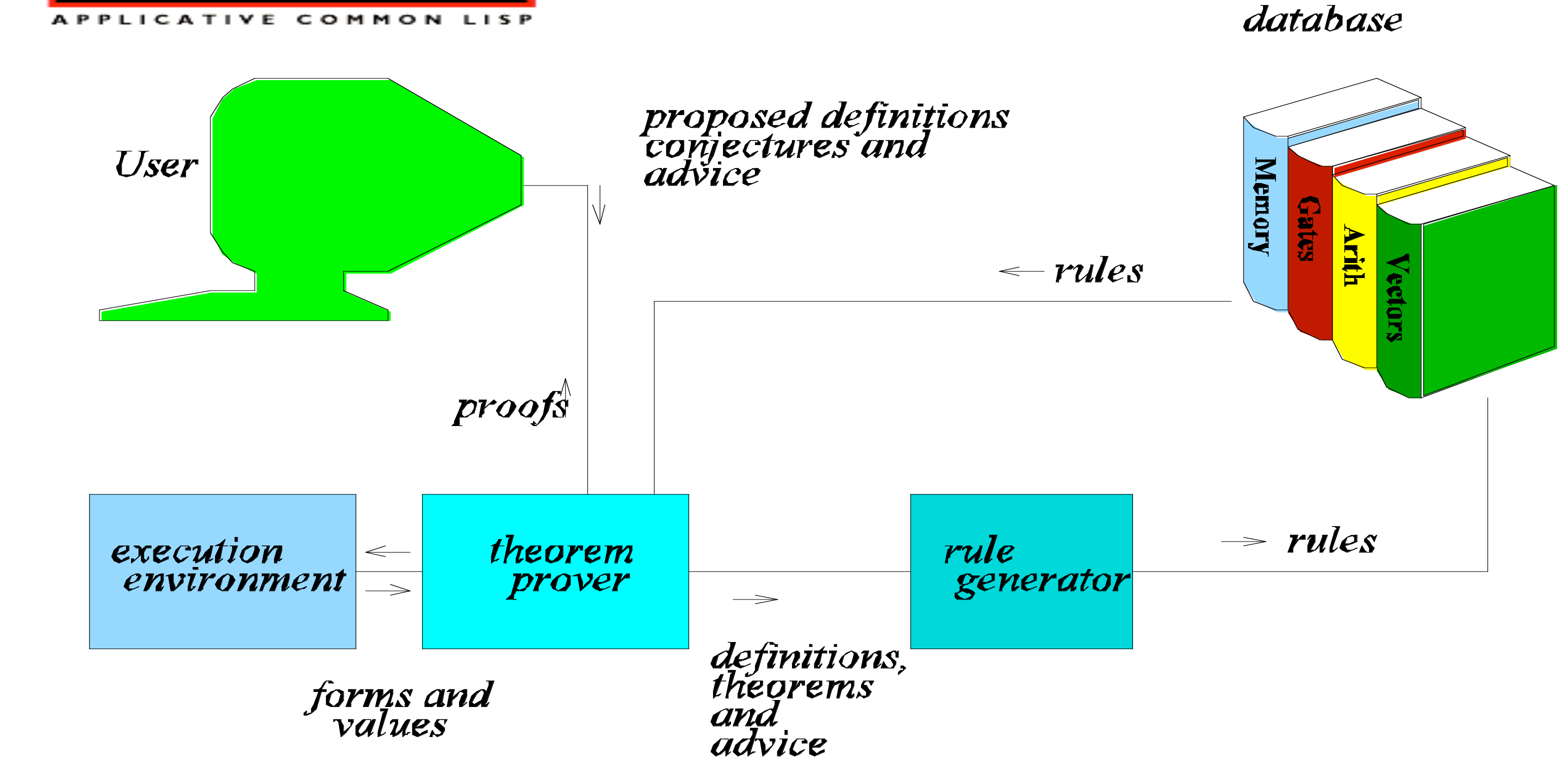
CLI Stack

The ACL2 Theorem Prover

- A Computational Logic for Applicative Common Lisp.
- Grounded in a programming language:
  - Functional (Pure, Applicative, Leibniz).
  - Lisp-like, all function total (untyped), first order.
  - Stobjs: 90%+ C speeds for hardware simulators.
- The logic:
  - First-order; most people use quantifier-free subset.
  - Definitions introduce axioms in a sound way (definitional principle).
  - Recursion and induction play a key role (software!).
  - Termination proofs are required (based on ordinals up to e_0).
  - Not just programming language semantics, but proof theory too.
- The theorem prover:
  - 30+ years of development.
  - Finely integrated proof techniques & decision procedures.
  - Will not try to cover internals.

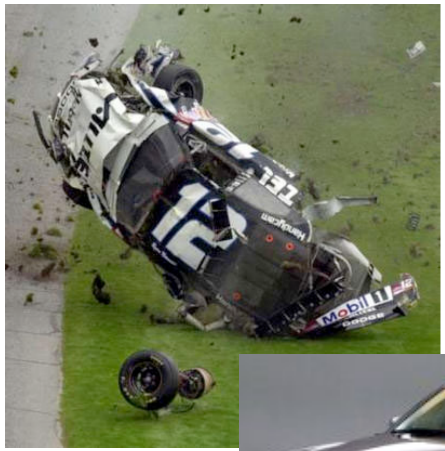
ACL2 System Architecture

Efficiently executable programming language, logic, theorem prover Kaufmann & Moore.



ACL2s

- ACL2 theorem prover.
  - Runs like a well-tuned race car in the hands of an expert.
  - Unfortunately, novices don't have the same experience.
  - Disseminate: wrote a book.
- ACL2s: The ACL2 Sedan
  - From race car to sedan.
  - Self-teaching.
  - Control a machine that is thinking about other machines.
  - Visualize what ACL2 is doing.
  - Levels & termination (CAV'06).
  - Use in processor design class.
  - Download it now.
  - Peter Dillinger, J Moore, Daron Vroon



A COMPUTATIONAL LOGIC
ACL2
APPLICATIVE COMMON LISP

ACL2s Demo Here
(1 Hour)

Hardware Verification: Motivation

International Technology Roadmap for Semiconductors, 2004 Edition.

Verification has become the dominant cost in the design process. On current projects, verification engineers outnumber designers, with this ratio reaching two or three to one for the most complex designs.

...

Without major breakthroughs, verification will be a non-scalable, show-stopping barrier to further progress in the semiconductor industry.

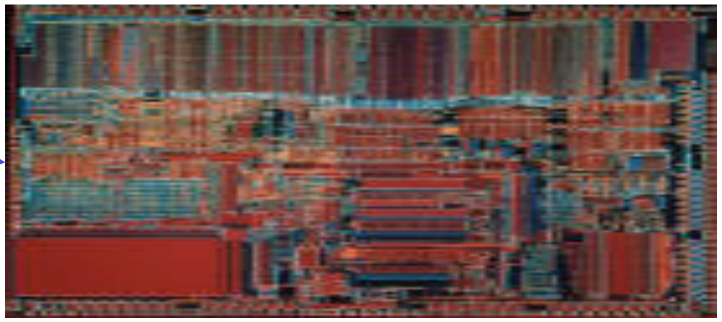
…

The overall trend from which these breakthroughs will emerge is the shift from ad hoc verification methods to more structured, formal processes.

Hardware Verification Challenge

- Verification costs range from 30%-70% of the entire design cost.
- R&D for typical CPU: 500+ team, costing \$0.5-1B.
- Pentium 4 (Bob Bently CAV 2005).
  - Full-chip simulation ~20Hz on Pentium 4.
  - Used ~6K CPUs 24/7: ~3 years later <1 minute of simulation cycles.
  - Exhaustive testing is impossible.
  - First large-scale formal verification at Intel: 60 person years.
  - Checked over 14K properties: Decode, Floating point, pipeline.
  - Found bugs, but no silicon bugs found to date in these areas.

Pentium FDIV
(Floating point DIVision) bug in
Pentium 386 led to a
\$475 million write-off by Intel
Bob Bently CAV **\$12B** 2005 terms



Outline

- Pleasantness Problem
- Refinement
- Local Reasoning
- Pipelined Machine Verification
- Automating Refinement
- Refinement Map Factor
- Compositional Reasoning
- Deductive Methods & Decision procedures
- Conclusions

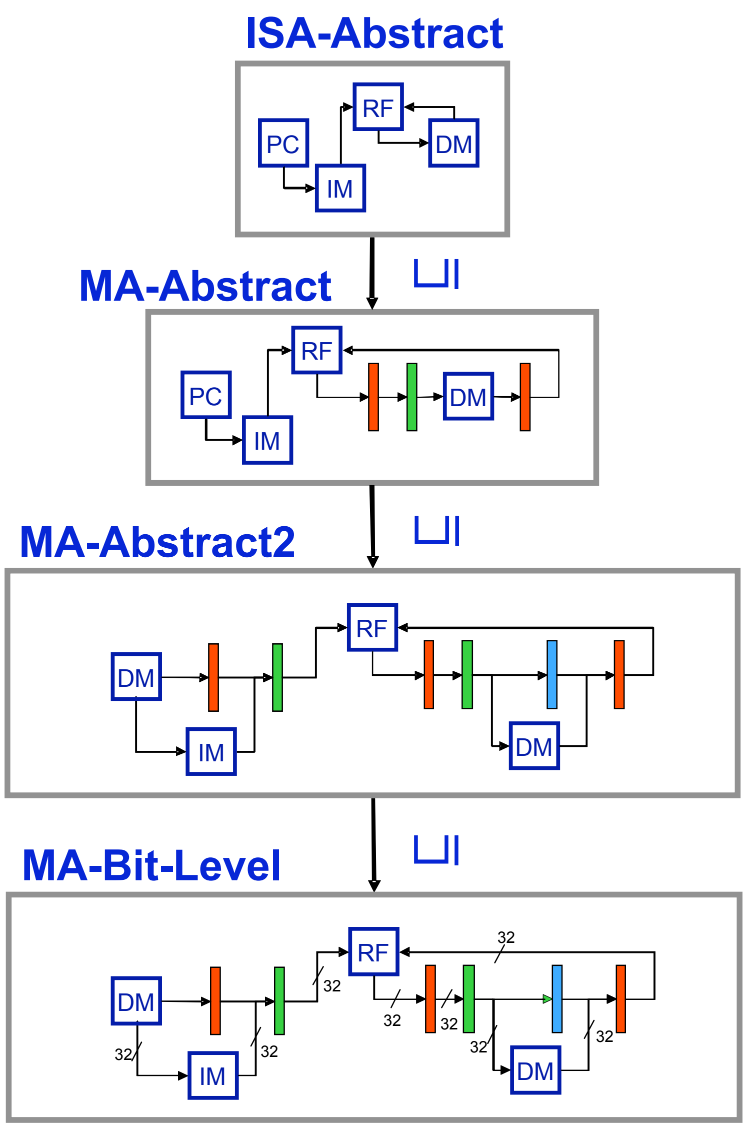
Approaches to Verification

- Property Based.
  - Relational, e.g., sorting.
  - Temporal logic, e.g., reactive systems.
- Refinement.
  - *I* refines *S* if every behavior of *I* is allowed by *S*.
  - The dual of abstraction.
  - Sorting.
  - Pipelined machines.
  - Communications Protocols.
  - Distributed Databases.

Some Key Ideas

- Stuttering.
  - *I* may require several steps to match *S*.
- Refinement maps.
  - *I* may contain more state components and may use different data representations than *S*.
- Theoretical, semantic-based approach.
  - Use Kripke structures $M = \langle S, \dashrightarrow, L \rangle$.
  - Ignore language issues.
- Refinement is a well explored area.
  - This is how I like to think of it.
  - The gestalt is what’s interesting.

Refinement, the Picture

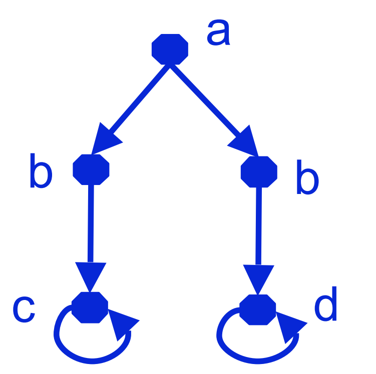


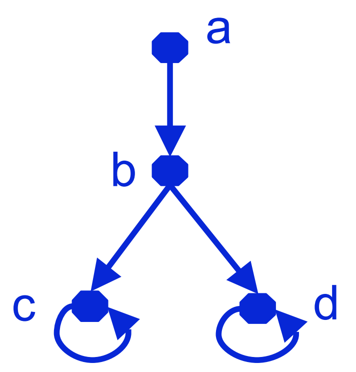
- Formal connection between different abstraction levels.
- Compositional.
- Preservation of safety and liveness.
- Avoid “leaky abstractions.”

Behaviors

I refines *S* if every behavior of *I* is allowed by *S*.
What are the behaviors of a system?

- Linear time.
  - Programs and properties are sets of infinite sequences.
  - Trace containment, equivalence (PSPACE-complete).
  - LTL.
- Branching time.
  - Programs and properties are sets of infinite trees.
  - Simulation, bisimulation (in P).
  - ACTL*, CTL*, μ-calculus.





Previous Work & Motivation

- Previous work on simulation & bisimulation.
  - Park81, Milner90: bisimulation & simulation.
  - Lynch & Vaandrager 94: forward & backward simulations.
  - BCG88: Stuttering bisimulation.
  - Namjoshi97: Proof rule for symmetric STBs.
  - MNS99: WEBs; combining MC & TP.
  - Manolios00: WEB Refinement for pipeline correctness.
  - Manolios01: Mechanical Verification of Reactive Systems.
- Motivation.
  - How does the branching view compare with the linear view?
    - Algorithmic advantages in the finite case.
    - Branching notions are structural, thus simpler; how much so?
    - Synergy?
  - Liveness.
  - Abstraction is powerful, but avoidance of “leaky abstractions”.
  - Use the theory to further automate pipelined machine verification.
  - Details are in CHARME03, DATE04/05, MEMOCODE05, ICCAD05, … .

Previous Work

Abadi and Lamport. *The Existence of Refinement Mappings,* Theoretical Computer Science, 1991.

- Spec: state machine & supplementary property (fairness).
- Proving that *I* refines *S* requires reasoning about infinite sequences: if *I* allows $\langle\langle e_0,z_0\rangle, \langle e_1,z_1\rangle, \langle e_2,z_2\rangle, \ldots\rangle$, then *S* allows the behavior $\langle\langle e_0,y_0\rangle, \langle e_1,y_1\rangle, \langle e_2,y_2\rangle, \ldots\rangle$.
- Reason locally (structurally)!
- Definition: If $f(e_n,z_n) = \langle e_n,y_n\rangle$, *f* can be used to prove, locally, that *I* preserves safety properties of *S*. If *f* preserves liveness, then it is a *refinement mapping*.
- Theorem: If the machine-closed specification *I* implements *S*, a specification that has finite invisible nondeterminism and is internally continuous, then there is a specification I^h obtained from *I* by adding a history variable and a specification I^{hp} obtained from I^h by adding a prophecy variable such that there exists a refinement mapping from I^{hp} to *S*.

Our Refinement Results

- A compositional theory of refinement that deals with liveness.
- Branching time.
  - Theorem: If *I* implements *S*, there exists a refinement mapping from *I* to *S*.
- Linear time.
  - Theorem: If *I* implements *S*, then there is a specification I^o, obtained from *I* by adding an oracle variable, such that there exists a refinement mapping from I^o to *S*.

Outline

Refinement

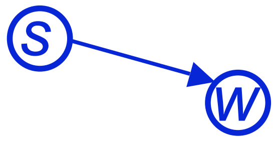
- Transition System (TS) $\mathcal{M} = \langle S, \dashrightarrow, L \rangle$
- Let
  - $\mathcal{M} = \langle S, \dashrightarrow, L \rangle$ (the implementation)
  - $\mathcal{M}' = \langle S', \dashrightarrow', L' \rangle$ (the specification)
  - $r : S \rightarrow S'$
- We say that $\mathcal{M}$ is a *simulation refinement* of $\mathcal{M}'$ with respect to refinement map r, written $\mathcal{M} \sqsubseteq_r \mathcal{M}'$, if there exists a relation, B, such that:
  - $\langle \forall s \in S :: sB(r.s) \rangle$
  - B is an STS on the TS $\langle S \uplus S', \dashrightarrow \uplus \dashrightarrow', \mathcal{L} \rangle$, where $\mathcal{L}.s = L'(s)$ for s an S' state, else $\mathcal{L}.s = L'(r.s)$.
- Compositional: $\mathcal{M} \sqsubseteq_r \mathcal{M}' \wedge \mathcal{M}' \sqsubseteq_q \mathcal{M}'' \Rightarrow \mathcal{M} \sqsubseteq_{r\,;q} \mathcal{M}''$

Stuttering Simulation

Transition System (TS) $\mathcal{M} = \langle S, \dashrightarrow, L \rangle$

B is a stuttering simulation (STS) on $\mathcal{M}$ iff
for all s, w such that sBw:

1. $L.s = L.w$



$\longrightarrow$: B

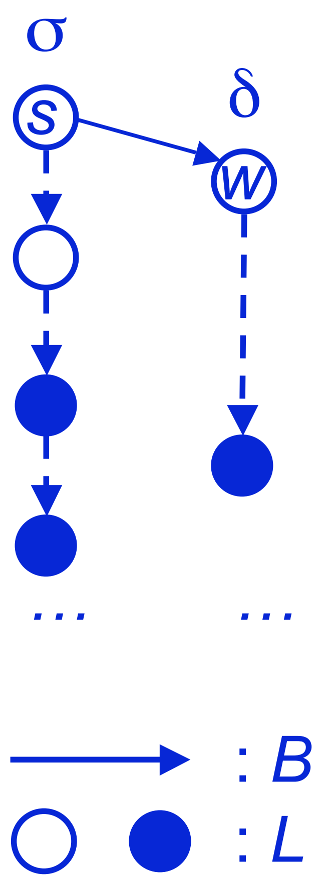
○ ● : L

Stuttering Simulation

Transition System (TS) $\mathcal{M} = \langle S, \dashrightarrow, L\rangle$

B is a stuttering simulation (STS) on $\mathcal{M}$ iff
for all *s*, *w* such that *sBw:*

1. $L.s = L.w$
2. $\langle \forall \sigma : fp.\sigma.s : \langle \exists \delta : fp.\delta.w : match(B,\sigma,\delta)\rangle\rangle$



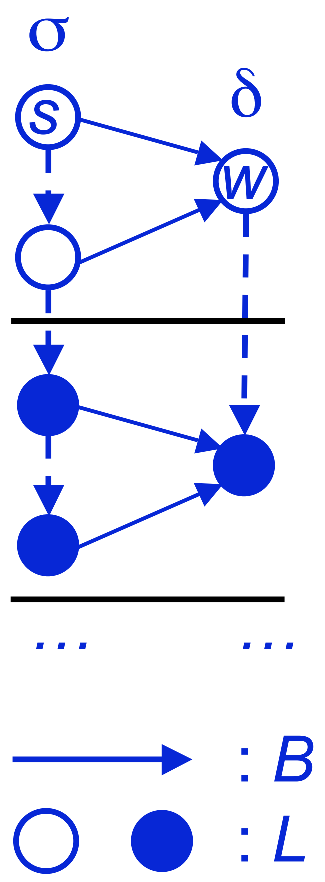
Stuttering Simulation

Transition System (TS) $\mathcal{M} = \langle S, \dashrightarrow, L\rangle$

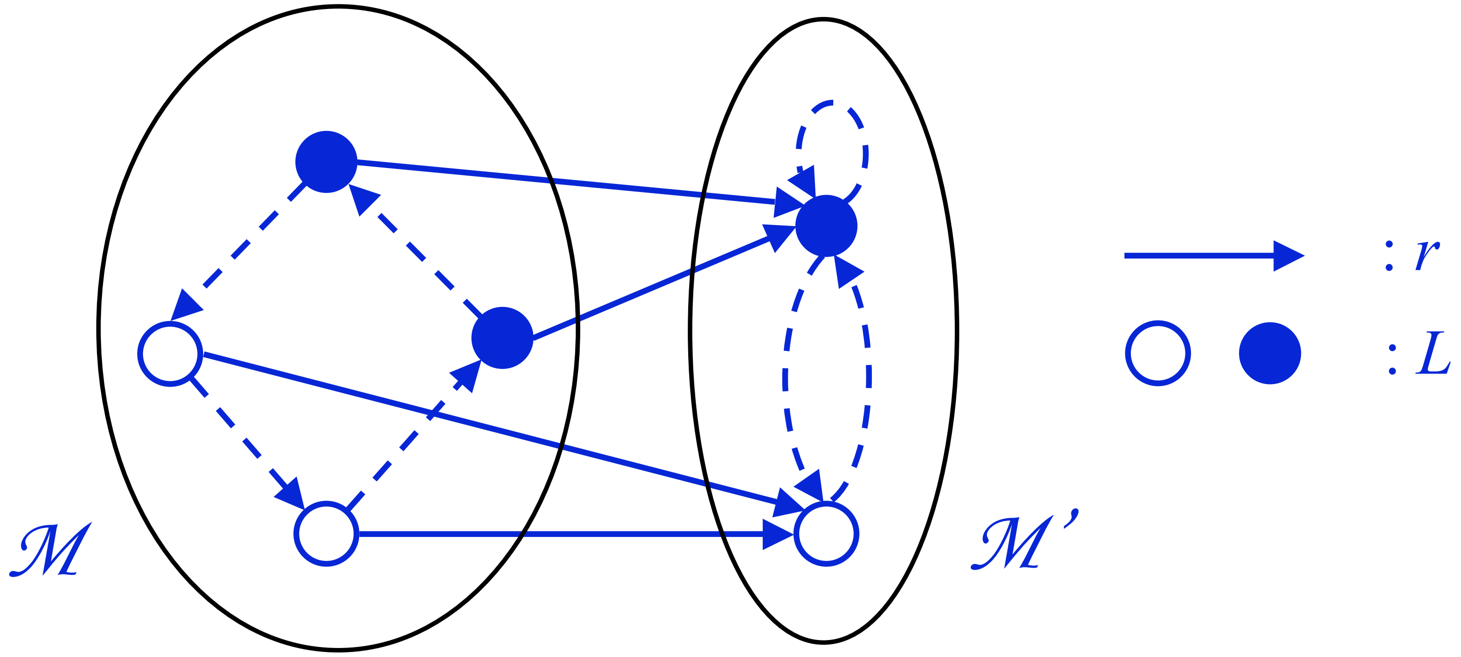
B is a stuttering simulation (STS) on $\mathcal{M}$ iff
for all *s*, *w* such that *sBw:*

1. $L.s = L.w$
2. $\langle \forall \sigma : fp.\sigma.s : \langle \exists \delta : fp.\delta.w : match(B,\sigma,\delta)\rangle\rangle$

match(*B*,σ,δ): σ,δ can be partitioned into non-empty, finite segments such that states in related segments are related by *B*.

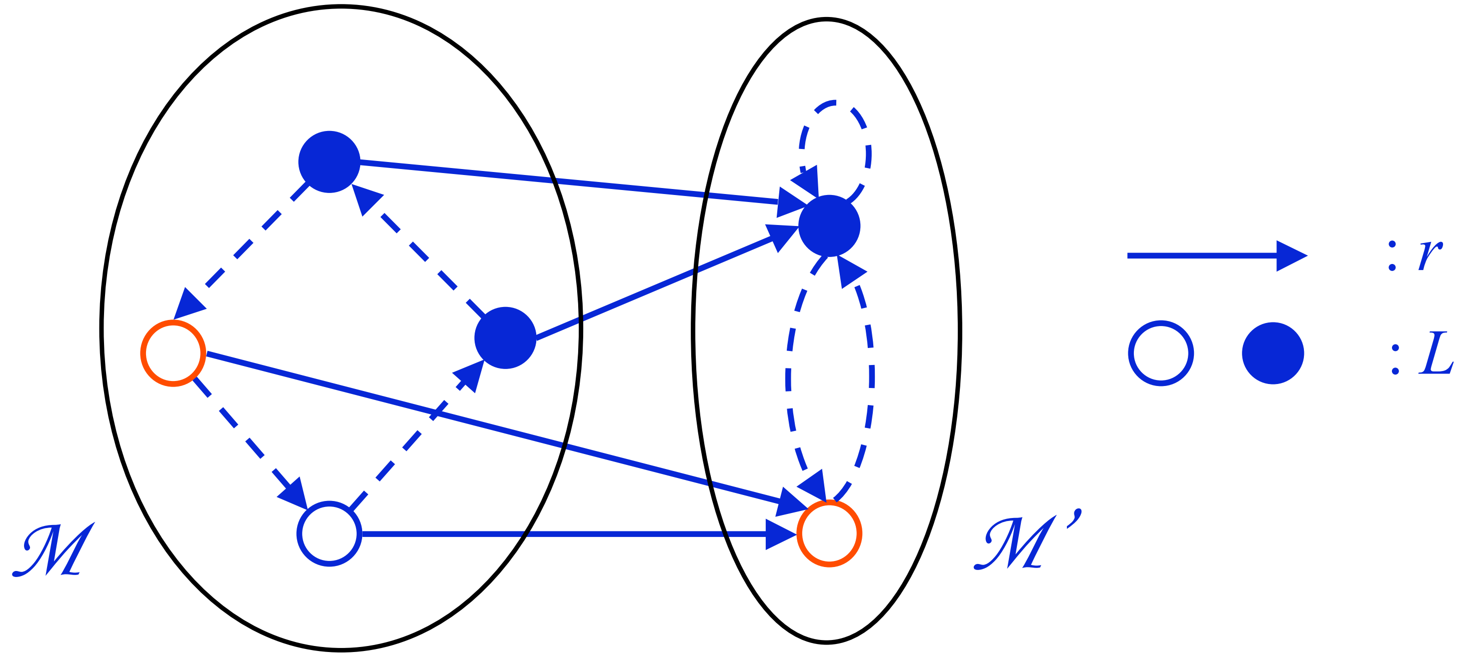


An Example



$\mathcal{M} \sqsubseteq_r \mathcal{M}'$ with witness *B*, the relation induced by *r*.

An Example



$\mathcal{M} \sqsubseteq_r \mathcal{M}'$ with witness *B*, the relation induced by *r*.

Stuttering Simulation

- For every TS $\mathcal{M}$, there is a greatest STS on $\mathcal{M}$:
  - Let C be a set of STS's, then $\cup_{B \in C}$ is an STS.

Stuttering Simulation

- For every TS $\mathcal{M}$, there is a greatest STS on $\mathcal{M}$:
  - Let C be a set of STS's, then $\cup_{B \in C}$ is an STS.
- If B is an STS, so is B*.
  - $B^* = \cup_{i \in \mathbb{N}} B^i$.
  - The identity relation is an STS.
  - If *R* and *S* are STSs, so is *R*;*S*, their composition.

Stuttering Simulation

- For every TS $\mathcal{M}$, there is a greatest STS on $\mathcal{M}$:
  - Let C be a set of STS's, then $\cup_{B \in C}$ is an STS.
- If B is an STS, so is B*.
  - $B^* = \cup_{i \in \mathbb{N}} B^i$.
  - The identity relation is an STS.
  - If *R* and *S* are STSs, so is *R*;*S*, their composition.
- The greatest STS, G, on $\mathcal{M}$ is a preorder.
  - G* is an STS.
  - $G \subseteq G^*$, but also $G^* \subseteq G$.

Stuttering Simulation

Theorem:

Let B be a STS on $\mathcal{M}$ and let sBw.
For every ACTL*\X formula f,
if $\mathcal{M},w \models f$, then $\mathcal{M},s \models f$.

Outline

- Pleasantness Problem
- Refinement
- Local Reasoning
- Pipelined Machine Verification
- Automating Refinement
- Refinement Map Factor
- Compositional Reasoning
- Deductive Methods & Decision procedures
- Conclusions

Proof Considerations

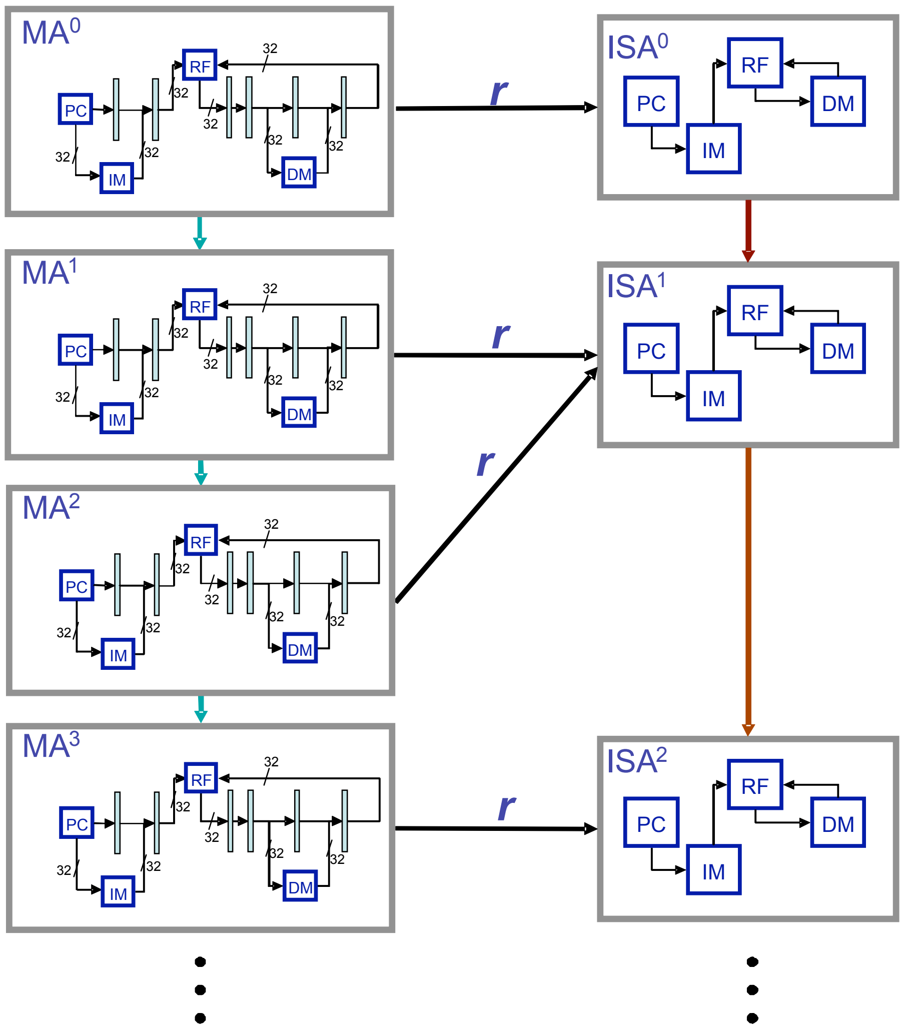
- Proving $\langle \forall \sigma{:}fp.\sigma.s : \langle \exists \delta{:}fp.\delta.w : match(B,\sigma,\delta)\rangle\rangle$ requires *global* reasoning; to do it mechanically is arduous.
- We define WFS and show it is equivalent to STS.
  - Soundness (easy).
  - Completeness (harder).
- Reasoning about WFS is *local.*
- The branching time result does not require machine closure, finite invisible nondeterminism, internal continuity, history variables, prophecy variables, etc.
- The linear time result only needs oracle variables.

Refinement via Local Reasoning

- Refinement requires matching infinite traces of the implementation and specification.
- Refinement maps allow a *local* check.

Technical overview:

- Define well-founded simulation & bisimulation.
- Local notions.
- Show soundness & completeness.



Well-founded Simulation (WFS)

1. $\langle \forall s,w \in S : sBw : L.s = L.w \rangle$
2. There are functions *rankt*: $S^2 \rightarrow W$, *rankl*: $S^3 \rightarrow \mathbb{N}$ such that $\langle W, \prec \rangle$ is well-founded and

$$\begin{tikzcd}
s \arrow[r] \arrow[d, dashed] & w \arrow[d, dashed] \\
u \arrow[r] & v
\end{tikzcd}$$

$$\langle \forall s,u,w \in S : (sBw \wedge s \dashrightarrow u) : \\ \langle \exists v : w \dashrightarrow v : uBv \rangle$$

Well-founded Simulation (WFS)

1. $\langle \forall s,w \in S : sBw : L.s = L.w \rangle$
2. There are functions *rankt*: $S^2 \rightarrow W$, *rankl*: $S^3 \rightarrow \mathbb{N}$ such that $\langle W, \prec \rangle$ is well-founded and

$$\begin{tikzcd} s \arrow[r] \arrow[d, dashed] & w \arrow[d, dashed] \\ u \arrow[r] & v \end{tikzcd}$$

or

$$\begin{tikzcd} s \arrow[r] \arrow[d, dashed] & w \\ u \arrow[ur] & \end{tikzcd}$$

$rankt(u,w) \prec rankt(s,w)$

$$\begin{aligned} &\langle \forall s,u,w \in S : (sBw \wedge s \dashrightarrow u) : \\ &\qquad \langle \exists v : w \dashrightarrow v : uBv \rangle \\ &\quad \vee\ (uBw \ \wedge\ rankt(u,w) \prec rankt(s,w)) \end{aligned}$$

Well-founded Simulation (WFS)

1. $\langle \forall s,w \in S : sBw : L.s = L.w \rangle$
2. There are functions *rankt*: $S^2 \rightarrow W$, *rankl*: $S^3 \rightarrow \mathbb{N}$ such that $\langle W, \prec \rangle$ is well-founded and

$$\begin{tikzcd} s \arrow[r] \arrow[d, dashed] & w \arrow[d, dashed] \\ u \arrow[r] & v \end{tikzcd} \quad \text{or} \quad \begin{tikzcd} s \arrow[r] \arrow[d, dashed] & w \\ u \arrow[ur] & \end{tikzcd} \quad \text{or} \quad \begin{tikzcd} s \arrow[r] \arrow[d, dashed] \arrow[dr] & w \arrow[d, dashed] \\ u & v \end{tikzcd}$$

$rankt(u,w) \prec rankt(s,w)$ $\quad$ $rankl(v,s,u) < rankl(w,s,u)$

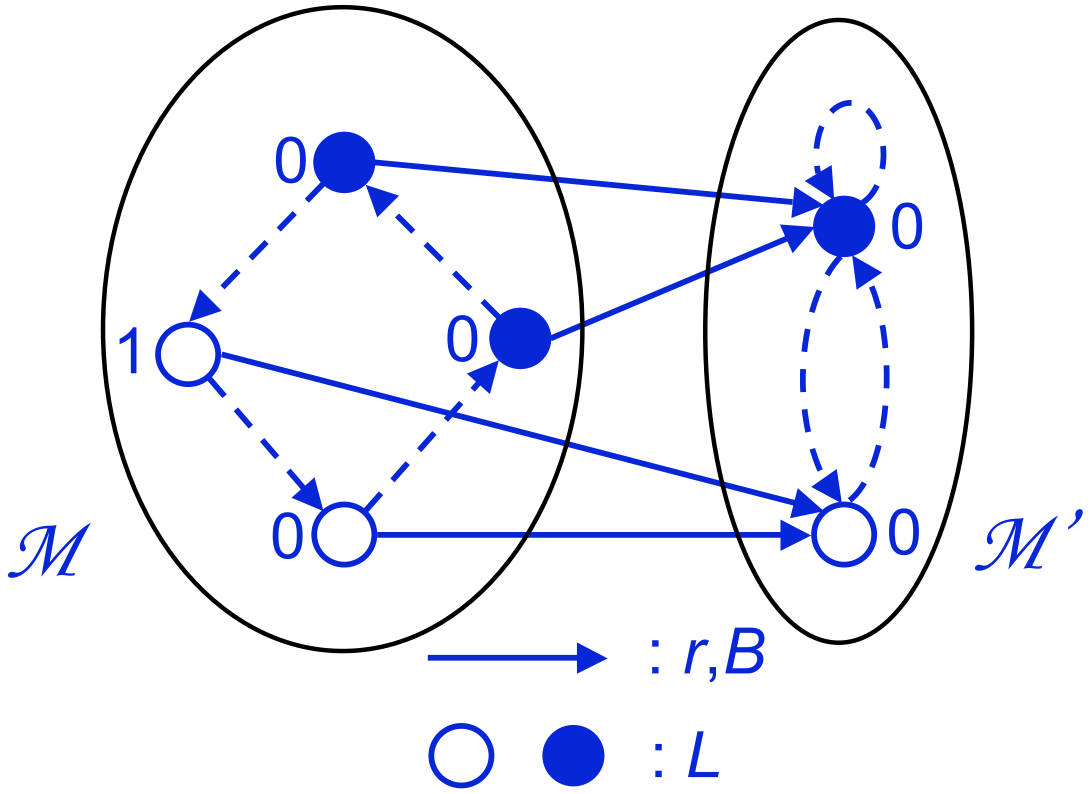
$$\langle \forall s,u,w \in S : (sBw \wedge s \dashrightarrow u) :$$

$$\langle \exists v : w \dashrightarrow v : uBv \rangle$$

$$\vee \ (uBw \wedge rankt(u,w) \prec rankt(s,w))$$

$$\vee \ \langle \exists v : w \dashrightarrow v : sBv \wedge rankl(v,s,u) < rankl(w,s,u) \rangle \rangle$$

An Example



rankt(*u*,*w*) = tag of *u*

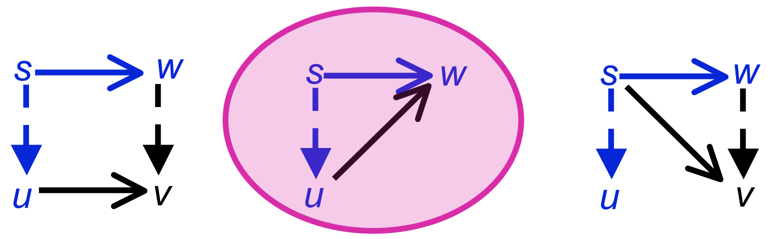
rankl(*v*,*s*,*u*) = tag of *v*

STS is WFS

Proof Outline.

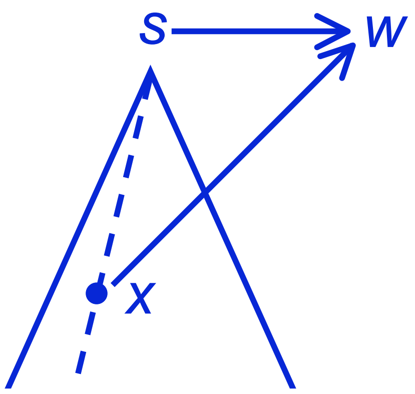
1. Define *rankt*(*s*,*w*):

tree(*s*,*w*) is defined to be the largest subtree of the computation tree rooted at *s* s.t. every non-root node $\langle s, \ldots, x\rangle$, we have *xBw* and $\langle \forall v : w \dashrightarrow v : \neg(xBv)\rangle$



$rankt(u,w) \prec rankt(s,w)$ $rankl(v,s,u) < rankl(w,s,u)$

$$\begin{aligned}
&\langle \forall s,u,w \in S : (sBw \wedge s \dashrightarrow u) : \\
&\qquad \langle \exists v : w \dashrightarrow v : uBv\rangle \\
&\quad \vee\ (uBw \wedge rankt(u,w) \prec rankt(s,w)) \\
&\quad \vee\ \langle \exists v : w \dashrightarrow v : sBv \wedge rankl(v,s,u) < rankl(w,s,u)\rangle\rangle
\end{aligned}$$



STS is WFS

Proof Outline.

1. Define *rankt*(*s*,*w*):

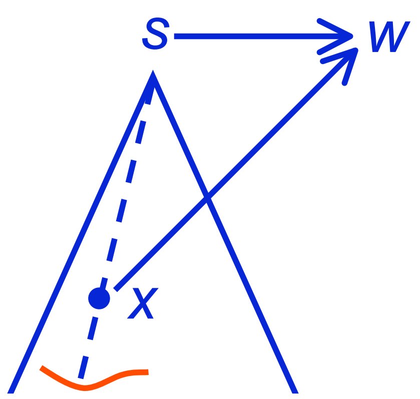
tree(*s*,*w*) is defined to be the largest subtree of the computation tree rooted at *s* s.t. every non-root node $\langle s, \ldots, x\rangle$, we have *xBw* and $\langle \forall v : w \dashrightarrow v : \neg(xBv)\rangle$

$$\begin{tikzcd} s \arrow[r] \arrow[d, dashed] & w \arrow[d, dashed] \\ u \arrow[r] & v \end{tikzcd}
\qquad
\begin{tikzcd} s \arrow[r] \arrow[d, dashed] & w \\ u \arrow[ru] & \end{tikzcd}
\qquad
\begin{tikzcd} s \arrow[r] \arrow[d, dashed] \arrow[rd] & w \arrow[d, dashed] \\ u & v \end{tikzcd}$$

$rankt(u,w) \prec rankt(s,w)$ $\quad rankl(v,s,u) < rankl(w,s,u)$

$$\begin{aligned} &\langle \forall s,u,w \in S : (sBw \wedge s \dashrightarrow u) : \\ &\qquad \langle \exists v : w \dashrightarrow v : \ uBv\rangle \\ &\quad \vee \ (uBw \ \wedge \ rankt(u,w) \prec rankt(s,w)) \\ &\quad \vee \ \langle \exists v : w \dashrightarrow v : \ sBv \ \wedge \ rankl(v,s,u) < rankl(w,s,u)\rangle\rangle \end{aligned}$$

Lemma: Every path of *tree* is finite.



STS is WFS

Proof Outline.

1. Define *rankt*(*s*,*w*):

tree(*s*,*w*) is defined to be the largest subtree of the computation tree rooted at *s* s.t. every non-root node ⟨*s*, …, *x*⟩, we have *xBw* and $\langle \forall v : w \dashrightarrow v : \neg(xBv)\rangle$

$$\begin{tikzcd} s \arrow[r] \arrow[d, dashed] & w \arrow[d, dashed] \\ u \arrow[r] & v \end{tikzcd}
\qquad
\begin{tikzcd} s \arrow[r] \arrow[d, dashed] & w \\ u \arrow[ur] & \end{tikzcd}
\qquad
\begin{tikzcd} s \arrow[r] \arrow[d, dashed] \arrow[dr] & w \arrow[d, dashed] \\ u & v \end{tikzcd}$$

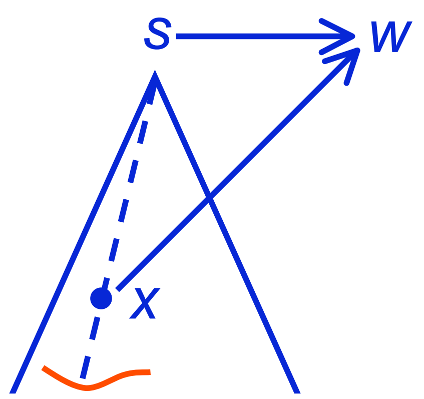
$rankt(u,w) \prec rankt(s,w)$ $\quad$ $rankl(v,s,u) < rankl(w,s,u)$

$$\begin{aligned}
&\langle \forall s,u,w \in S : (sBw \wedge s \dashrightarrow u) : \\
&\qquad \langle \exists v : w \dashrightarrow v : uBv\rangle \\
&\quad \vee\ (uBw \wedge rankt(u,w) \prec rankt(s,w)) \\
&\quad \vee\ \langle \exists v : w \dashrightarrow v : sBv \wedge rankl(v,s,u) < rankl(w,s,u)\rangle\rangle
\end{aligned}$$

Lemma: Every path of *tree* is finite.

Label nodes in *tree* using standard set-theory "rank" function.

Lemma: If $|S| \leq \kappa$ (where $\omega \leq \kappa$), then for all $s,w \in S$, *tree*(*s*,*w*) is labeled with an ordinal of cardinality $\leq \kappa$.

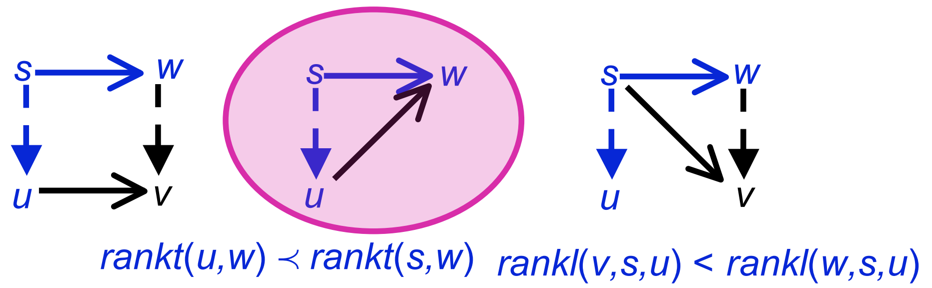


STS is WFS

Proof Outline.

1. Define *rankt*(s,w):

tree(s,w) is defined to be the largest subtree of the computation tree rooted at s s.t. every non-root node $\langle s, \ldots, x\rangle$, we have xBw and $\langle \forall v : w \dashrightarrow v : \neg(xBv)\rangle$

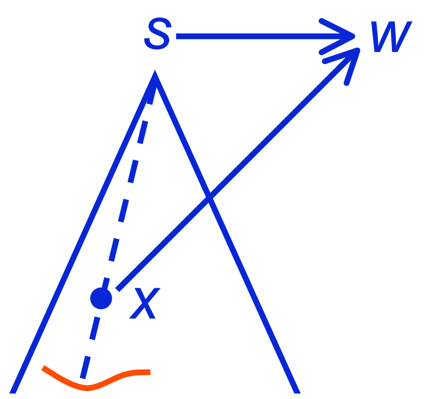


$rankt(u,w) \prec rankt(s,w)$ $\quad rankl(v,s,u) < rankl(w,s,u)$

$\langle \forall s,u,w \in S : (sBw \wedge s \dashrightarrow u) :$
$\quad \langle \exists v : w \dashrightarrow v : uBv\rangle$
$\quad \vee\ (uBw \wedge rankt(u,w) \prec rankt(s,w))$
$\quad \vee\ \langle \exists v : w \dashrightarrow v : sBv \wedge rankl(v,s,u) < rankl(w,s,u)\rangle\rangle$

Lemma: Every path of *tree* is finite.

Label nodes in *tree* using standard set-theory "rank" function.



Lemma: If $|S| \leq \kappa$ (where $\omega \leq \kappa$), then for all $s,w \in S$, *tree*(s,w) is labeled with an ordinal of cardinality $\leq \kappa$.

Def: $rankt = l.tree$, where $\langle W, \prec\rangle$ is $\langle (|S| + \omega)^+, \prec\rangle$.

Existence of Refinement Maps

- Branching time:

  Theorem: If $\mathcal{I}$ implements $\mathcal{S}$, then there is a refinement map r such that $\mathcal{I} \sqsubseteq_r \mathcal{S}$.

- Linear time:

  Theorem: If $\mathcal{I}$ implements $\mathcal{S}$ (the set of traces of $\mathcal{I}$ is a subset of the traces of $\mathcal{S}$), then there exists $\mathcal{I}'$, where $\mathcal{I}'$ is obtained from $\mathcal{I}$ by adding an oracle variable, and a refinement map r such that $\mathcal{I}' \sqsubseteq_r \mathcal{S}$.

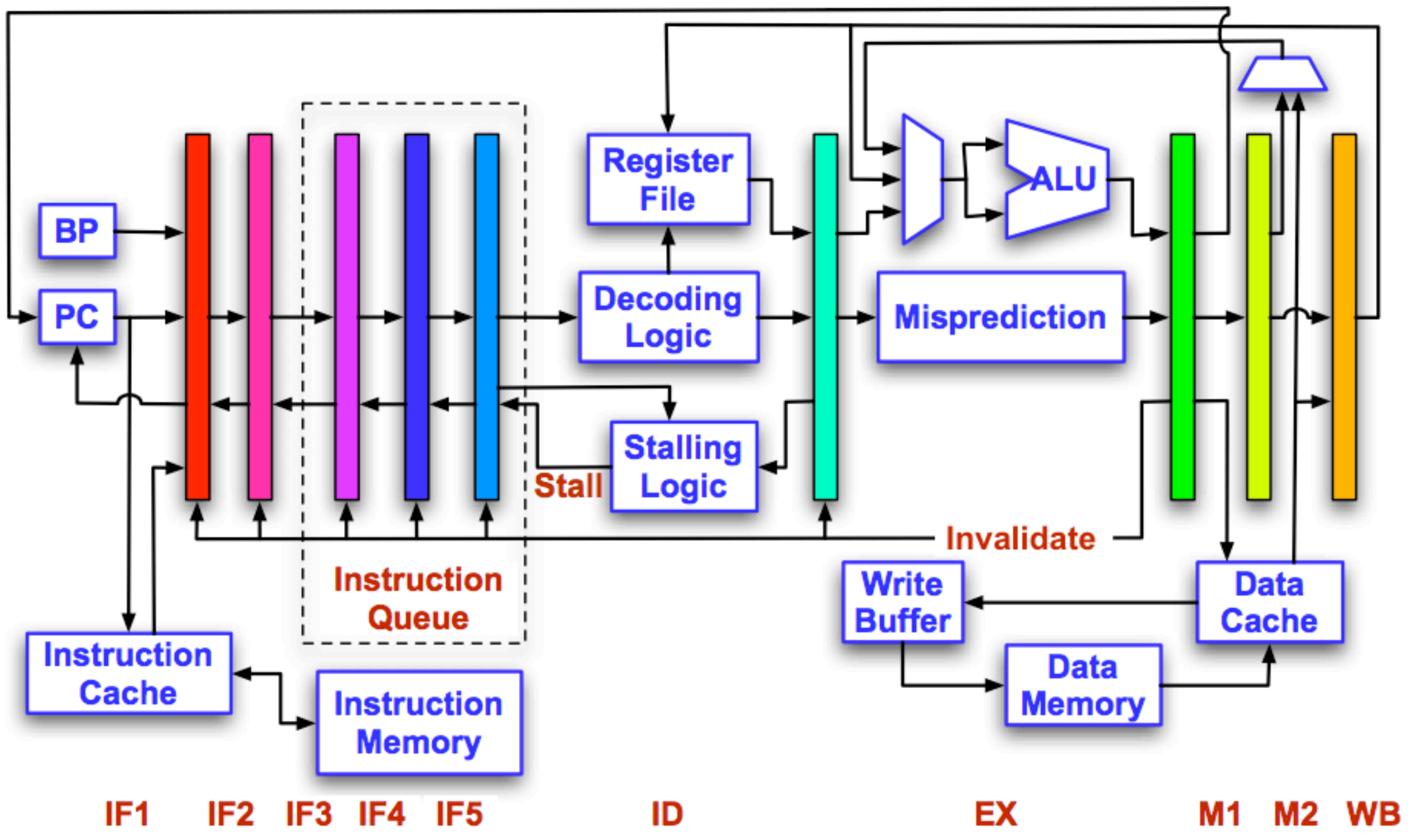
Outline

- Pleasantness Problem
- Refinement
- Local Reasoning
- Pipelined Machine Verification
- Automating Refinement
- Refinement Map Factor
- Compositional Reasoning
- Deductive Methods & Decision procedures
- Conclusions

Previous Work

- Theorem Proving.
  - Gopalakrishnan, Hosabettu, Srivas: PVS.
  - Arons, Pnueli: PVS.
  - Kroening, Paul, et al: PVS, Isabel/HOL.
  - Sawada & Hunt: ACL2.
- Decision procedures.
  - Burch & Dill.
  - Bryant, Lahiri, Seshia, CLU, UCLID.
  - Barrett, Dill, Stump CVC.
  - McMillan, Model checking.
- Motivation:
  - Automate refinement proofs.
  - Burch & Dill inspired commuting diagrams not enough.
  - Previously we showed that deadlock is possible.
  - B&D variants are interesting theorems, but provide a leaky abstraction, *e.g.*, liveness is ignored.
  - Can we use the theory as a handle to handle more complexity?

Pipelined Machine Model



Outline

Automating Refinement

$$
\begin{aligned}
&\langle \forall s,u,w \in S : (sBw \wedge s \dashrightarrow u) : \\
&\qquad\qquad \langle \exists v : w \dashrightarrow v : \; uBv \rangle \\
&\qquad \vee \; (uBw \;\wedge\; rankt(u,w) \prec rankt(s,w)) \\
&\qquad \vee \; \langle \exists v : w \dashrightarrow v : \; sBv \;\wedge\; rankl(v,s,u) < rankl(w,s,u) \rangle \rangle
\end{aligned}
$$

Automating Refinement

$$\langle \forall s,u,w \in S : (sBw \wedge s \dashrightarrow u) :$$
$$\quad \langle \exists v : w \dashrightarrow v : \; uBv \rangle$$
$$\quad \vee \; (uBw \;\wedge\; rankt(u,w) \prec rankt(s,w))$$
$$\quad \vee \; \langle \exists v : w \dashrightarrow v : \; sBv \;\wedge\; rankl(v,s,u) < rankl(w,s,u) \rangle \rangle$$

Gets reduced to:

$$\langle \forall w \in \text{MA} ::$$
$$\quad s = r.w \;\wedge\; u = \text{ISA-step}(s) \;\wedge$$
$$\quad v = \text{MA-step}(w) \;\wedge\; u \neq r.v$$
$$\Rightarrow$$
$$\quad s = r.v \wedge \; rank.v < rank.w \rangle$$

$$\begin{tikzcd}
s \arrow[d, dashed] & w \arrow[l, "r"'] \arrow[d, dashed] \\
u & v \arrow[l, "r", "/" marking] \arrow[ul, "r"']
\end{tikzcd}$$

rank.v < *rank.w*

Joint work with Sudarshan Srinivasan

Automating Refinement

Core Theorem:

$$
\begin{aligned}
&\langle \forall w \in \text{MA} :: \\
&\qquad s = r.w \ \wedge\ u = \text{ISA-step}(s) \ \wedge \\
&\qquad v = \text{MA-step}(w) \ \wedge\ u \neq r.v \\
&\Rightarrow \\
&\qquad s = r.v \wedge\ rank.v < rank.w \rangle
\end{aligned}
$$

$$\begin{tikzcd}
s \arrow[d, dashed] & w \arrow[l, "r"'] \arrow[d, dashed] \\
u & v \arrow[l, "r", "/" marking] \arrow[ul, "r"']
\end{tikzcd}$$

$rank.v < rank.w$

- This is expressed in CLU, a fragment of first order logic.
- UCLID is a decision procedure for CLU, which generates a SAT instance.
- We use the Siege SAT solver to check the SAT instance.

Refinement Maps & Ranks

- Flushing refinement map:
  - Finish all partially executed instructions (Drain).
  - MA-step is used to define flushing.
  - For our example, 14 symbolic simulations required.
  - Consistency invariants required for write-through caches.
- Rank function:
  - Number of steps to fetch an instruction that eventually completes.
- Both can be defined automatically.

Safety and Liveness Results

Processor	Safety				Safety and Liveness						
	CNF Vars	CNF Clauses	Verification Time [sec]		CNF Vars	CNF Clauses	CNF Size [KB]	Verification Time [sec]			
			Siege	Total				Siege	Chaff	Stdev	Total
CXS	12,930	38,215	35	38	12,495	36,925	664	29	6,552	3.4	32
CXS-BP	24,640	72,859	284	289	23,913	70,693	1,336	300	7,861	48.7	305
CXS-BP-EX	24,651	72,841	244	249	24,149	71,350	1,344	233	4,099	50.2	238
CXS-BP-EX-INP	24,699	72,880	255	261	24,478	72,322	1,368	263	3,483	34.1	269
FXS	28,505	36,929	140	154	53,441	159,010	3,096	160	796	24.4	175
FXS-BP	33,964	100,624	170	185	71,184	211,723	4,136	187	586	50.4	203
FXS-BP-EX	35,827	106,114	179	195	74,591	221,812	4,344	163	759	17.6	180
FXS-BP-EX-INP	38,711	114,742	128	147	81,121	241,345	4,736	170	1,427	32.3	189

The base machine is a six stage XScale-inspired term-level model. (DATE04)

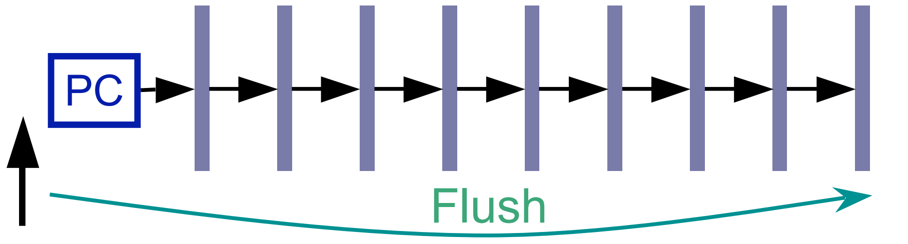
Outline

- Pleasantness Problem
- Refinement
- Local Reasoning
- Pipelined Machine Verification
- Automating Refinement
- Refinement Map Factor
- Compositional Reasoning
- Deductive Methods & Decision procedures
- Conclusions

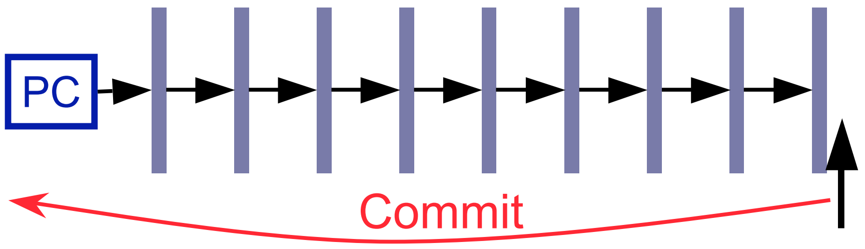
The Refinement Map Factor

- The refinement maps used can have a drastic impact on verification times.
  - It is possible to attain orders of magnitude improvements in verification times.
  - Can enable the verification of machines that are too complex to otherwise automatically verify.
- Beyond flushing.
  - Commitment (FMCAD 00, DATE 04).
  - GFP (Memocode 05).
  - Intermediate maps (DATE 05).
  - Collapsed flushing (DATE 06).

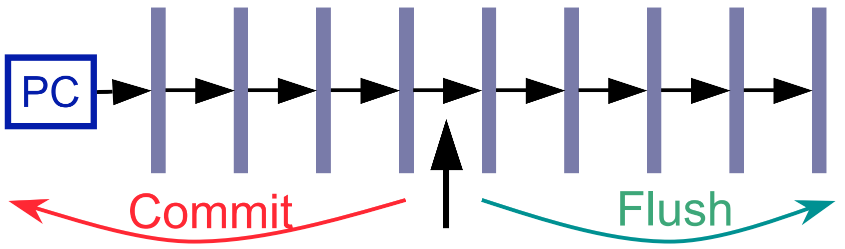
Intermediate Refinement Maps



IR0: Flush 9 latches,
(Flushing)



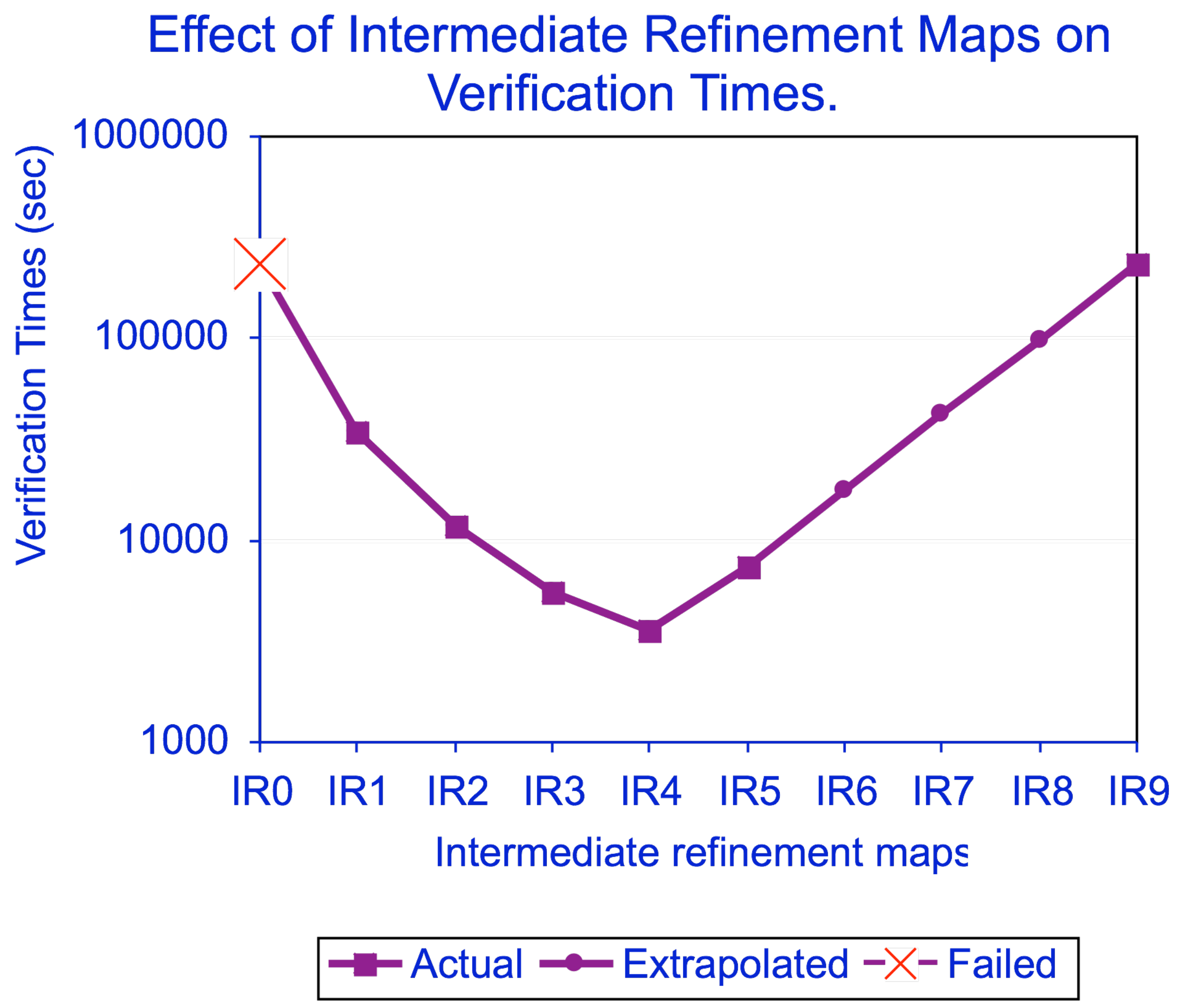
IR9: Commit 9 latches,
(Commitment)



IR4: Commit first 4 latches,
Flush last 5 latches

- Verification time is exponential in the pipeline "complexity": $O(2^c)$.
- For intermediate refinement maps:
  - Complexity of flushing and commitment parts is c/2.
  - Resulting verification time is $O(2^{c/2})$.

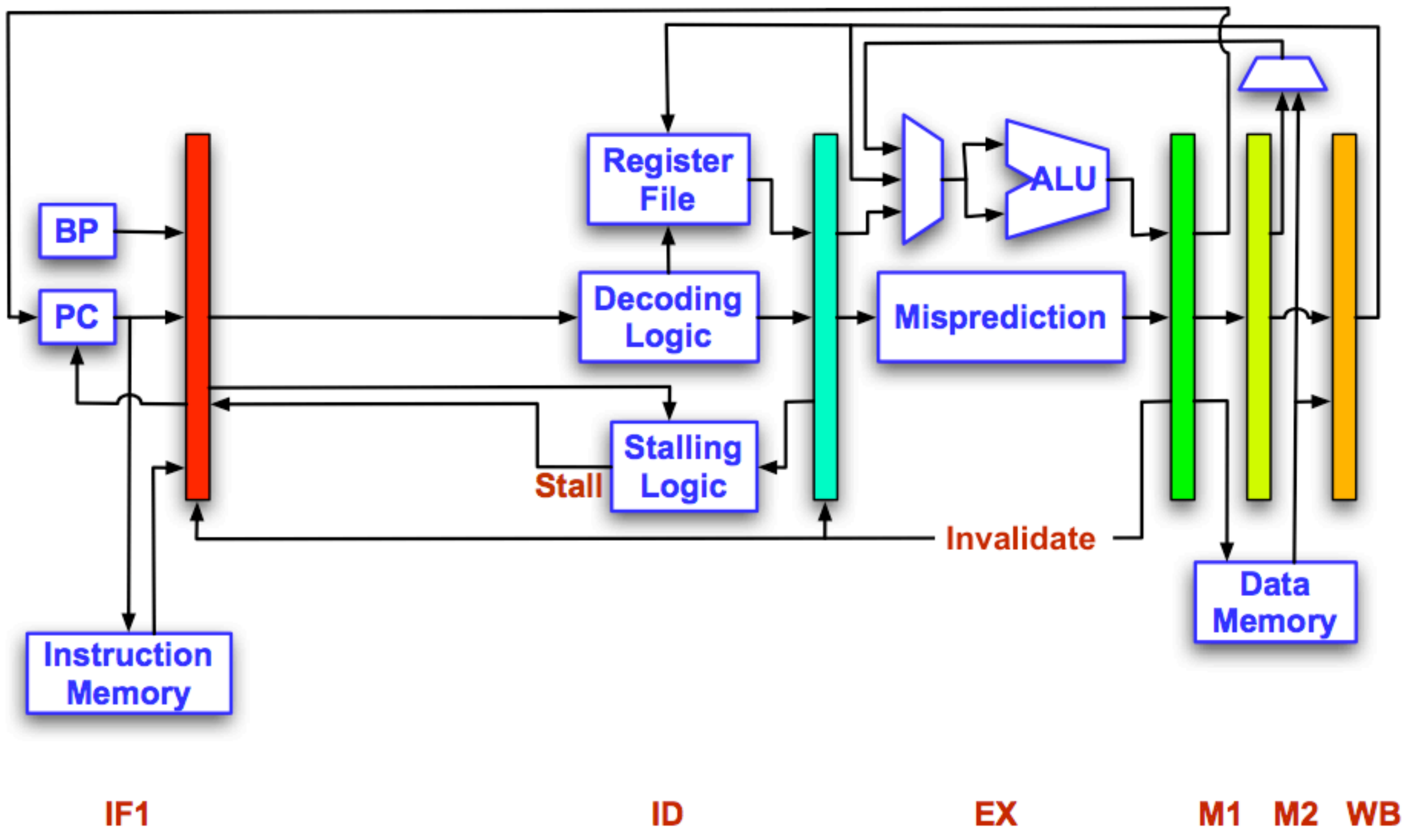
IR Results



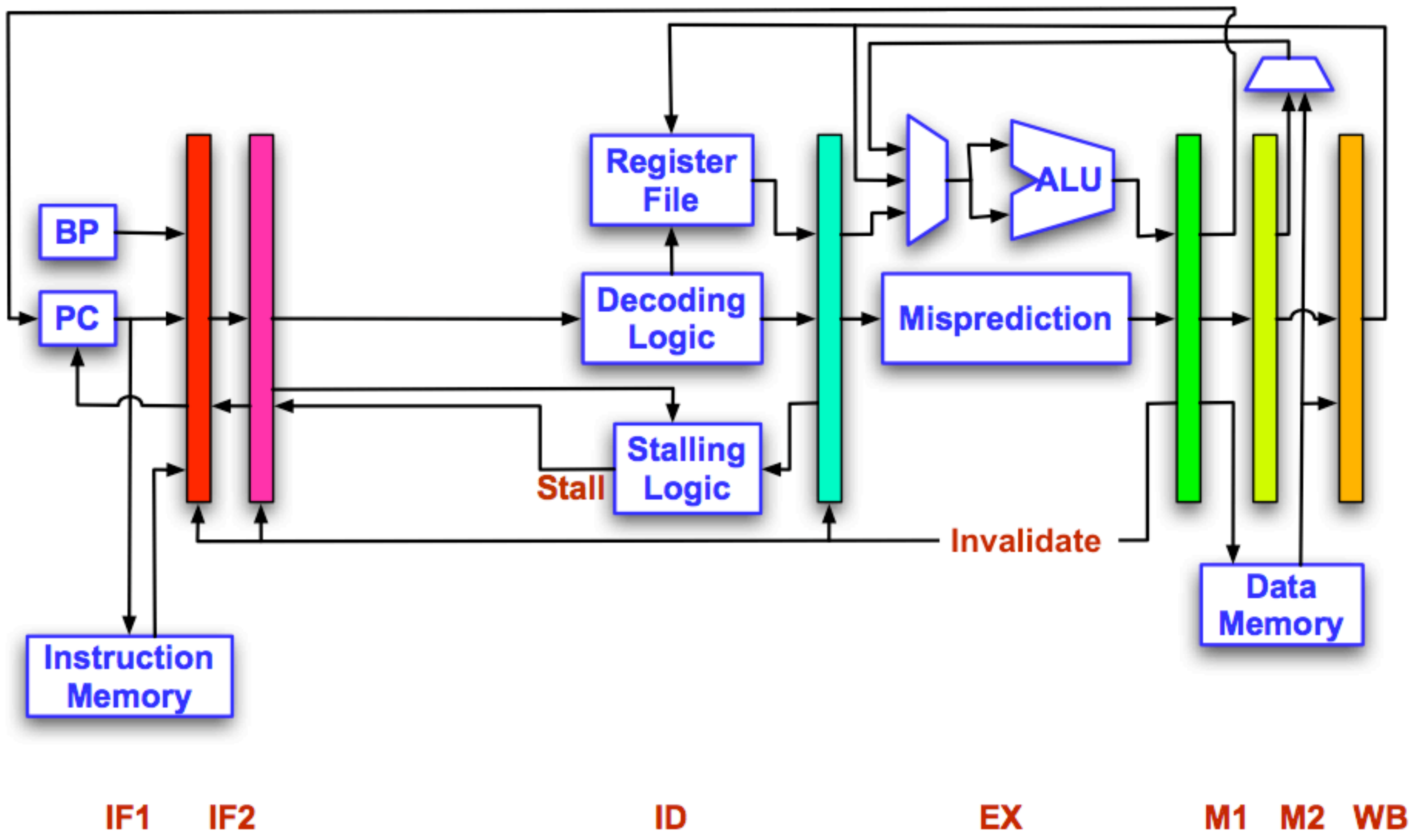
Effect of Intermediate Refinement Maps on Verification Times.
Verification Times (sec)
1000000
100000
10000
1000
IR0
IR1
IR2
IR3
IR4
IR5
IR6
IR7
IR8
IR9
Intermediate refinement maps
Actual
Extrapolated
Failed

Outline

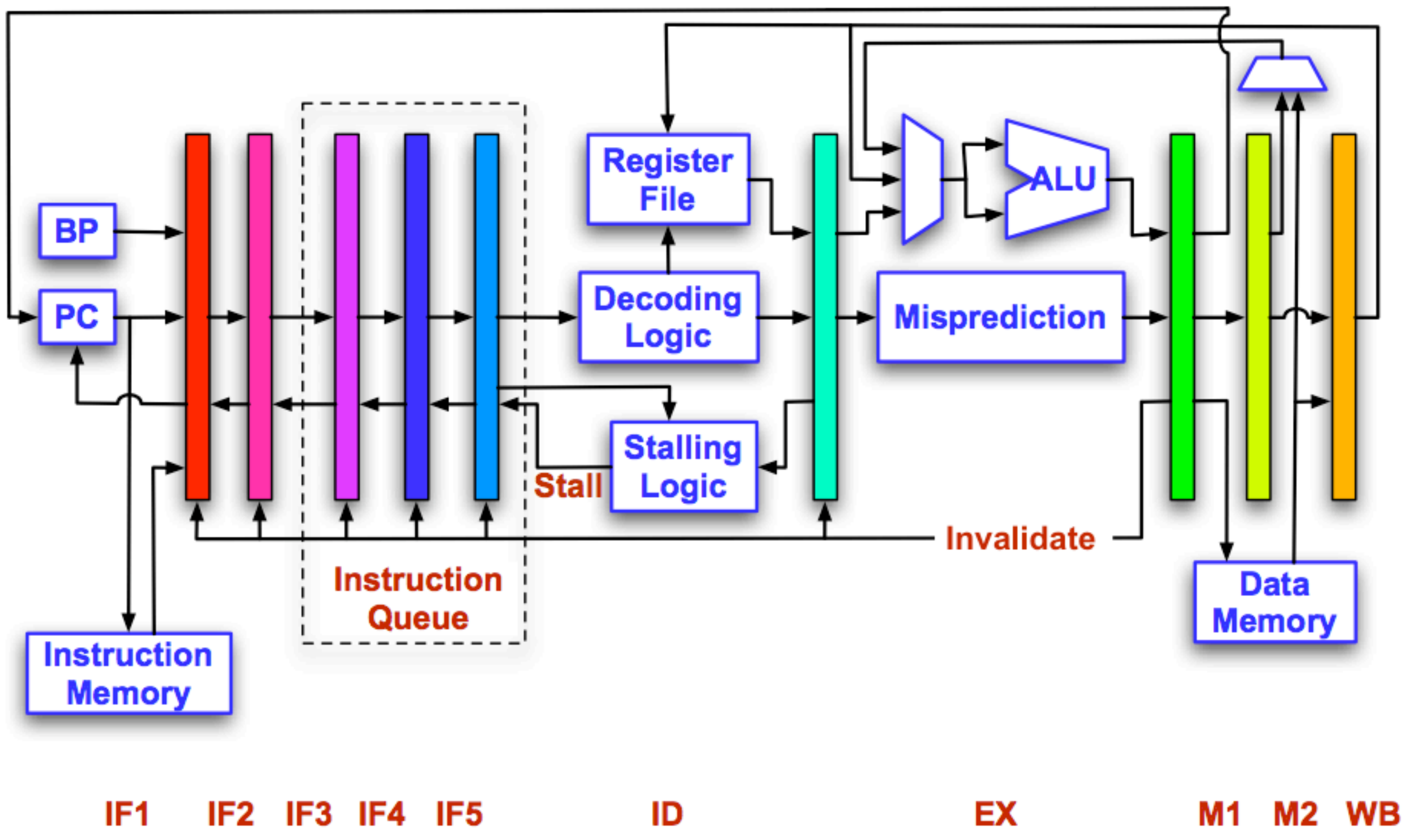
Pipelined Machine M6



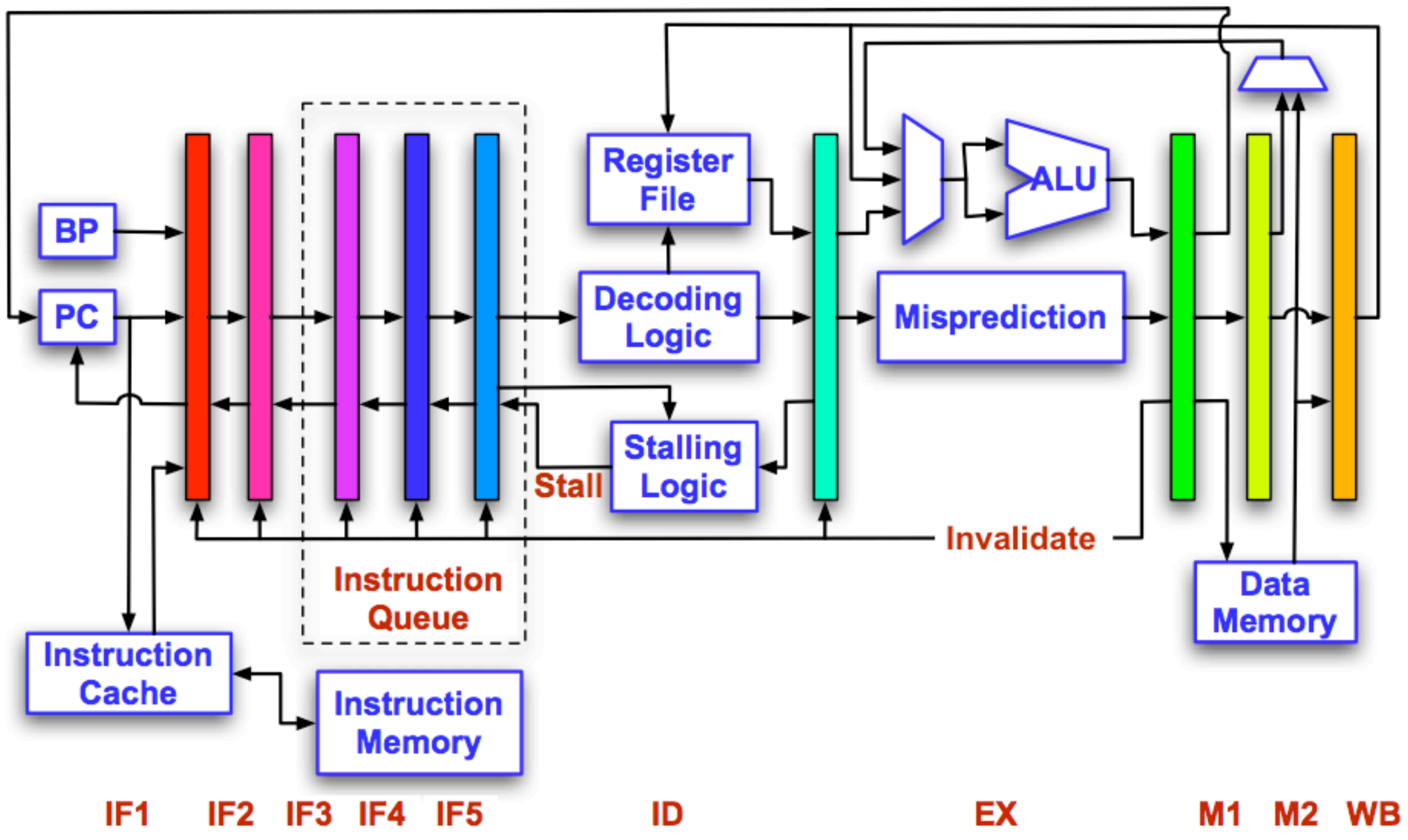
Pipelined Machine M7



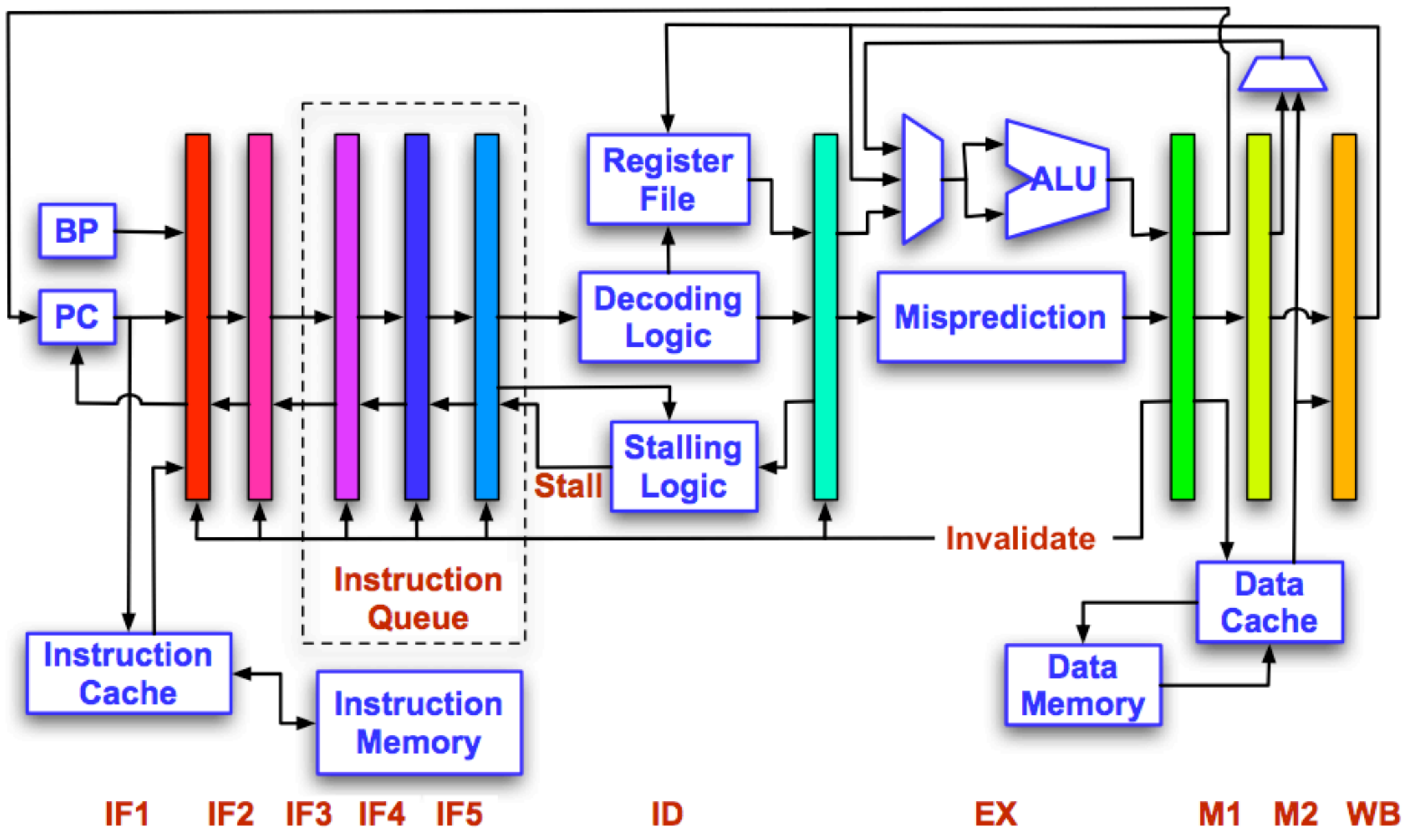
Pipelined Machine M10



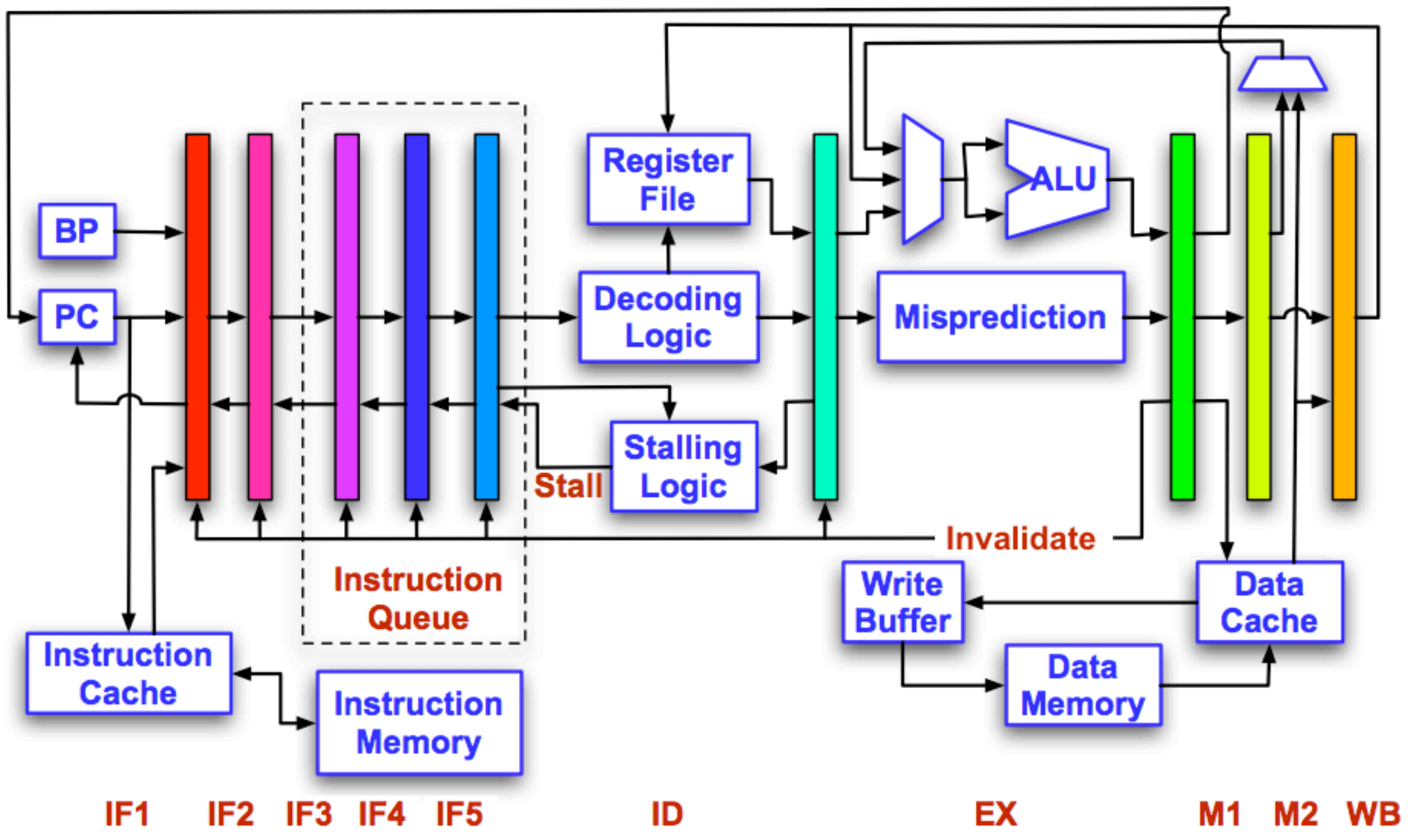
Pipelined Machine M10I



Pipelined Machine M10ID



Pipelined Machine M10IDW



Monolithic Verification

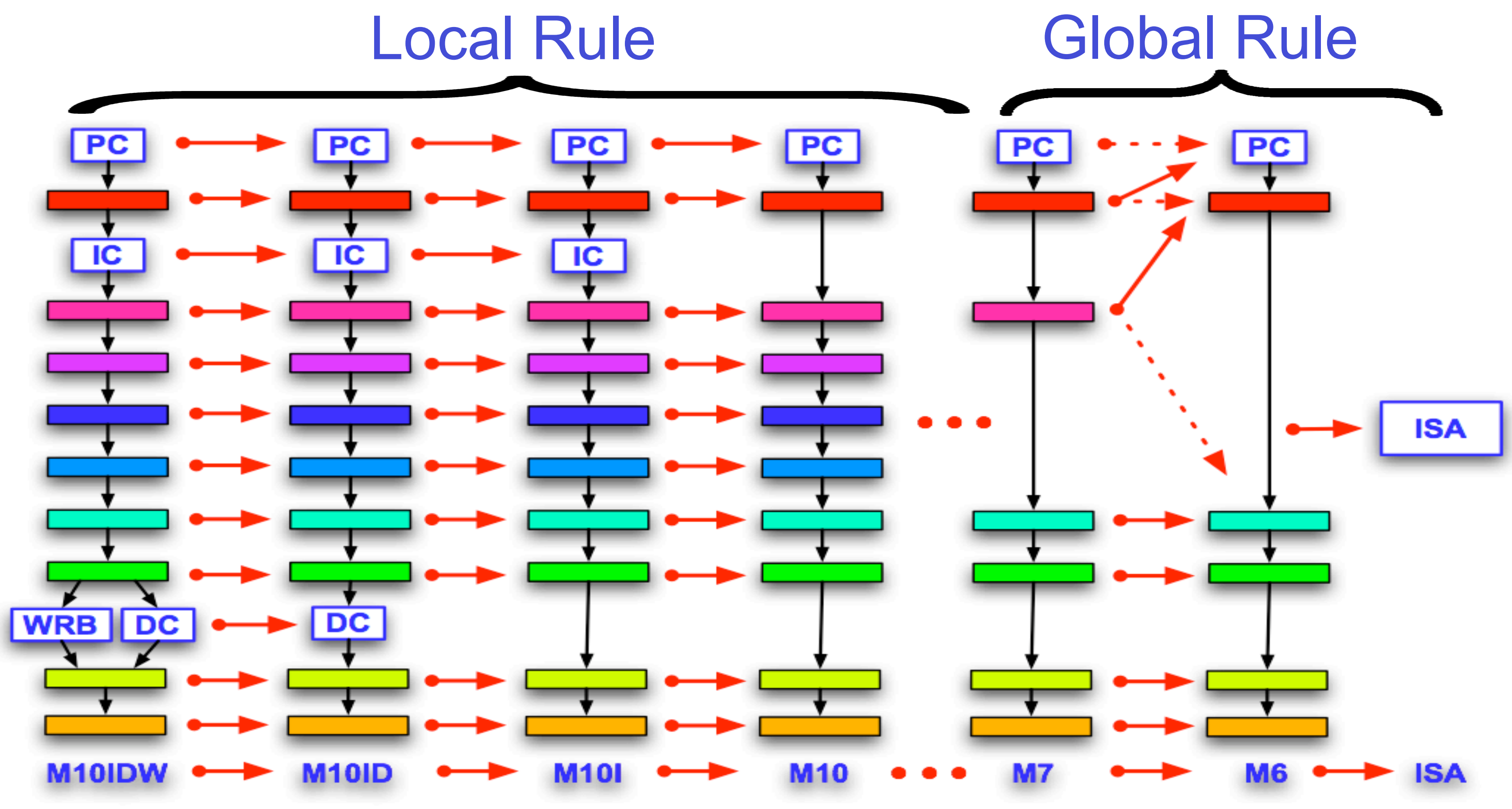
- We modeled the machines using UCLID.
- UCLID compiles to SAT; we used Siege.
- Results: exponential increase in verification time.

Pipelined Machine	CNF		Verification Times (sec)		
	Vars	Clauses	UCLID	Siege	Total
M6	28,256	83,725	8	10	18
M7	53,165	158,182	15	150	165
M8	95,092	283,465	25	766	791
M9	144,045	429,973	41	2,436	2,477
M10	198,375	592,660	55	6,762	6,817
M10I	293,862	876,820	92	8,641	8,733
M10ID	580,355	1,730,704	244	Fail	NA
M10IDW	690,598	2,060,557	297	Fail	NA

Compositional Verification

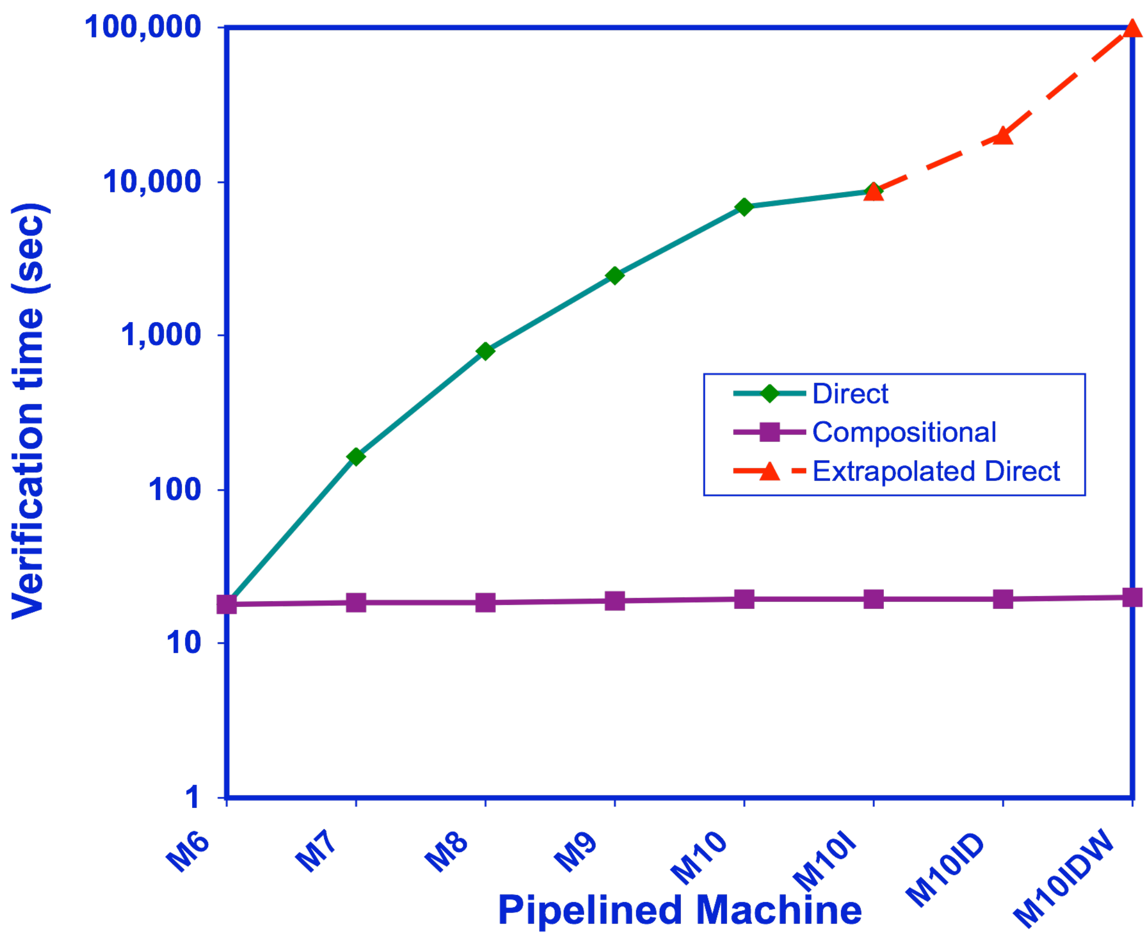
- Idea: verify the machines the way we defined them, one step at a time.
- Developed a complete, compositional framework for pipelined machine verification.
- Preserve safety & liveness.
- M10IDW now takes ~20 seconds to verify!
- Counterexamples tend to be much simpler: they can be isolated to a refinement step.
- Appeared in ICCAD 05.

Compositional Proof



Refinement maps and ranks are easier to define piecewise.

Compositional vs. Direct



Why such good results?

Complexity of proof depends on semantic gap between machines.

Our compositional framework makes this gap manageable.

Counterexamples

- UCLID generates counterexamples when it fails.
  - Understanding counterexamples is hard!
  - Students spend a lot of time on this; prefer code inspection.
- Our framework leads to simpler counterexamples.
  - Occur at the refinement stage where the error appears.
  - Tend to be much smaller in size.
  - Tend to involve less simulation steps.
  - Can be generated more quickly.
- Aids debugging and design understanding.
- Example of a cache error.
  - Invariant: 1/2 second for both approaches.
  - Direct: 69 simulation steps, 6076 lines, 1026 sec.
  - Compositional: 2 simulation steps, 445 lines, <20 sec.

Outline

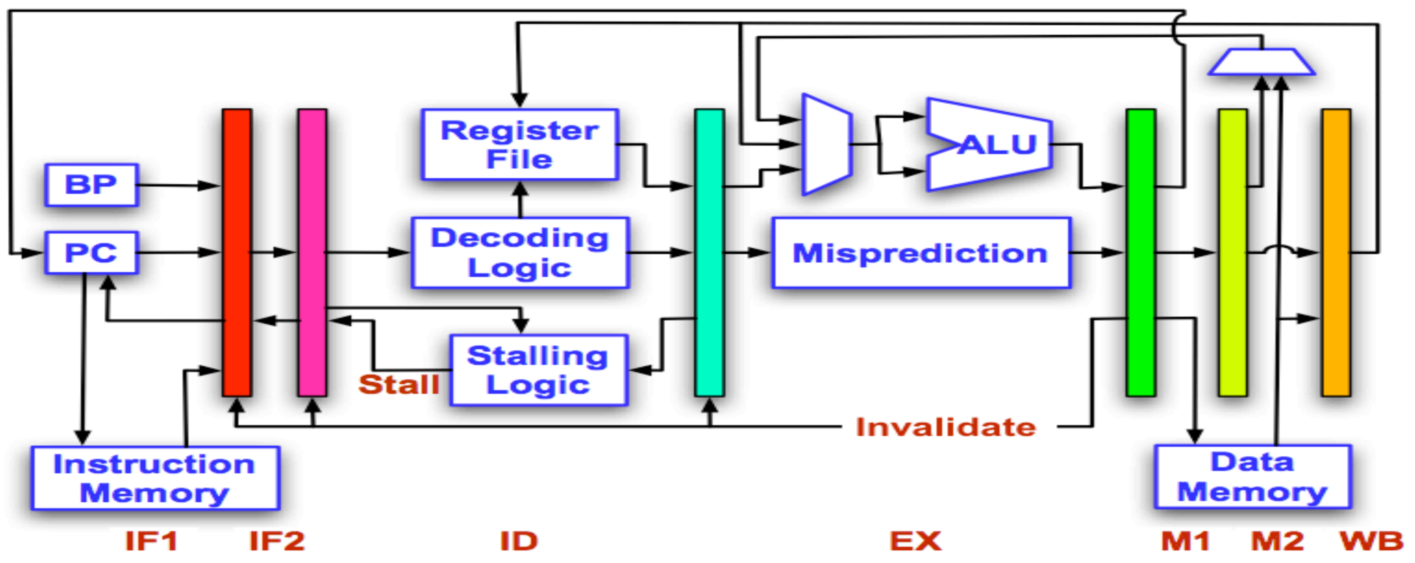
Limitations of UCLID/Decision Procedures

- Correctness statement not expressible in CLU.
    - “Core theorem” is expressible.
- Term-level Modeling.
    - Datapath, decoder, etc. abstracted away.
    - Only small subset of instruction set implemented.
    - Restricted modeling language: no modules.
    - Restricted logic: forces us to add extra state & control logic.
    - Far from bit-level or executable.
    - Not clear how to relate to RTL, bit-level designs.
    - No way to reason about programs: have no semantics.

Combining ACL2 & UCLID

- Idea: Use ACL2 to reduce correctness of bit-level, executable machines to term-level problems which UCLID can handle.
- ACL2 is used to manage the proof process.
- We can state the full refinement theorem.
- We can separate concerns.
  - Models and refinement maps are dealt with separately.
- ACL2 is used to reason at the bit level.
- UCLID is used to reason about the pipeline.
- Result: We can verify executable machines with bit-level interfaces without heroic effort.
- Enables us to reason about machine code running on the pipelined machine.
- See ICCAD 2005.

Processor Model



- Xscale inspired machine defined in ACL2 programming language.
- Executable, *e.g.*, dynamic programming solution for knapsack.
- Bit-level, except for ALU, memory & register file (bit-level interface).
- 32-bit instructions, 16 registers, parameterized by word size .
- Register-register & register-immediate addressing modes.
- Instructions: 16 ALU, 15 branch & various jump, load, store, etc.
- Predicated instructions based on 16 conditions: 593 instructions.

Proof Outline

$$\begin{tikzcd}
MB \arrow[r, "\text{Memory}"] & MM \arrow[r, "\text{Bit-level}"] & ME \arrow[r, "\text{Pollute}"] & MEP \arrow[r, "\text{Pipeline}"] & IEP \arrow[r, "\text{Purify}"] & IM \arrow[r, "\text{Memory}"] & IE
\end{tikzcd}$$

MB: Pipeline bit-level, executable
MM: MB, with evaluator memories
ME: Pipeline, integer, executable
MEP: Polluted ME
IEP: ISA version MEP
IM: Purified IEP
IE: IM with alist memories

$A \longrightarrow B$ A refines B (proof by ACL2)

Proof Outline

$$\begin{tikzcd}
MB \arrow[r, "\text{Memory}"] & MM \arrow[r, "\text{Bit-level}"] & ME \arrow[r, "\text{Pollute}"] & MEP \arrow[r, "\text{Pipeline}"] & IEP \arrow[r, "\text{Purify}"] & IM \arrow[r, "\text{Memory}"] & IE \\
 & & & MA \arrow[r] \arrow[u, Rightarrow] & IA \arrow[u, Rightarrow] & &
\end{tikzcd}$$

MA, IA abstract MEP, IEP

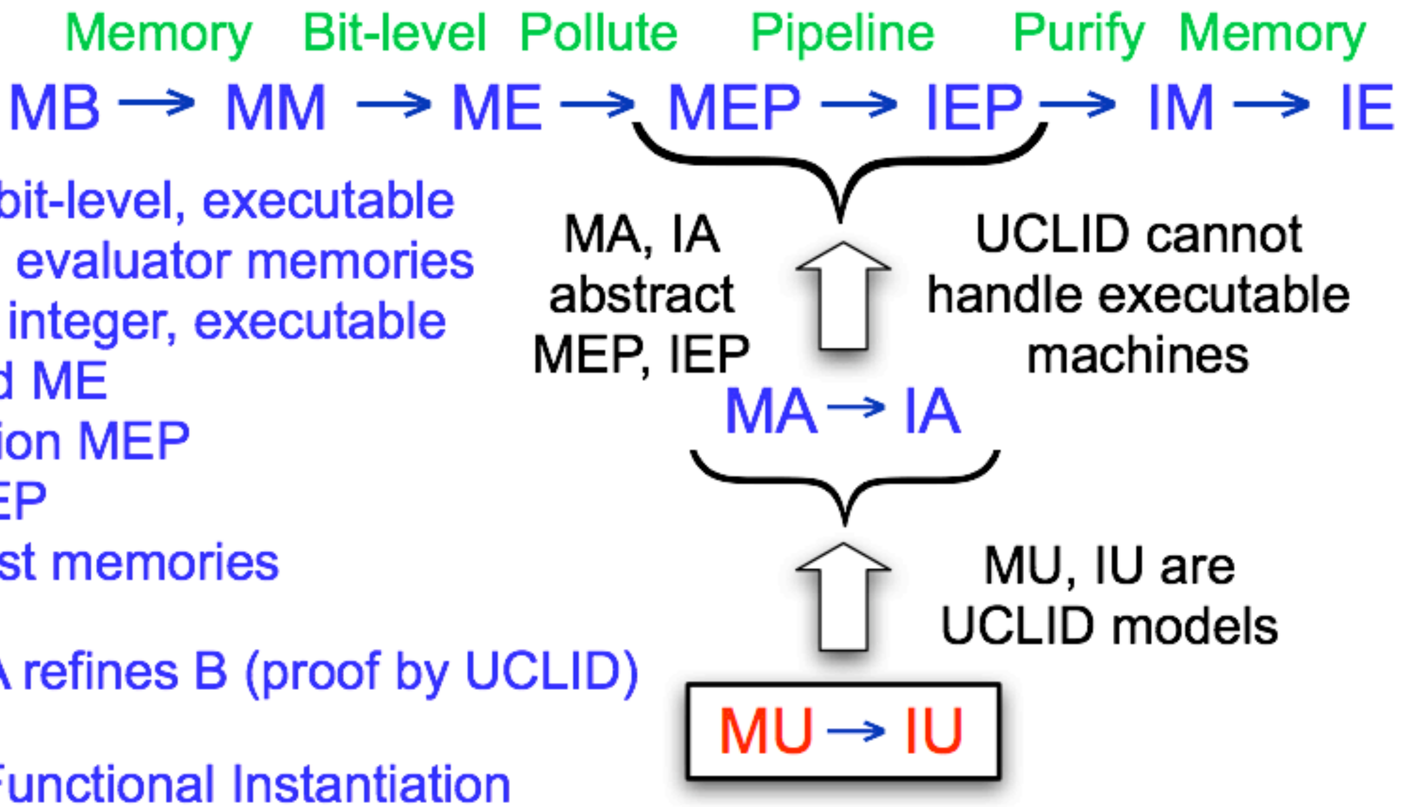
UCLID cannot handle executable machines

MB: Pipeline bit-level, executable
MM: MB, with evaluator memories
ME: Pipeline, integer, executable
MEP: Polluted ME
IEP: ISA version MEP
IM: Purified IEP
IE: IM with alist memories

$\Rightarrow$ Functional Instantiation

$A \rightarrow B$ A refines B (proof by ACL2)

Proof Outline



MB: Pipeline bit-level, executable
MM: MB, with evaluator memories
ME: Pipeline, integer, executable
MEP: Polluted ME
IEP: ISA version MEP
IM: Purified IEP
IE: IM with alist memories

$\boxed{A \rightarrow B}$	A refines B (proof by UCLID)
⇨	Functional Instantiation
$A \rightarrow B$	A refines B (proof by ACL2)

Verification Statistics

Proof Step	Proof Time (sec)	User Effort (man-weeks)
MU $\rightarrow$ IU	157	3
MA $\rightarrow$ IA	91	2
MEP $\rightarrow$ IEP	36	2
IEP $\rightarrow$ IM	4	1
IM $\rightarrow$ IE	601	1
ME $\rightarrow$ MEP	21	2
MM $\rightarrow$ ME	182	3
MB $\rightarrow$ MM	625	1

Times estimate the effort that would be required for an ACL2 & UCLID expert and do not include the integration effort.

Program Verification

- Executable models describe semantics of instructions.
- They can be used to simulate the machines (6.6 KHz).
- Simulation revealed errors!
  - Decoder functions read bits in reverse order.
  - Processor status flag arguments swapped.
  - Condition for updating the register file wrong.
- Our refinement proof allows us to reduce reasoning about programs on MB to reasoning about IE programs.

$$\mathrm{MA} \approx_r \cdots \approx_q \mathrm{ISA}$$

$$\frac{\mathrm{ISA} \;\|\; \mathrm{P} \;\vdash \varphi}{\mathrm{MA} \;\|\; \mathrm{P} \;\vdash \varphi}$$

- Preservation of liveness is crucial for this, otherwise we have to look inside of MB.

Dynamic Programming Solution for Knapsack

Given a knapsack of capacity T and a set of n items, each of which has a cost C(·), and a value V(·), what is the maximum value the knapsack can attain without exceeding it’s capacity?

```
storei r1 0         add r11 r11 r13
movi r6 0           movi r0 20
addi r6 r6 1        sub r9 r11 r7
movi r10 0          bn r0
movi r7 0           mov r7 r11
add r14 r3 r10      movi r0 5
add r15 r4 r10      sub r11 r5 r10
addi r10 r10 1      bnz r0
load r12 r14        add r11 r1 r6
load r13 r15        store r11 r7
movi r0 20          movi r0 2
sub r11 r6 r12      sub r11 r2 r6
bn r0               bnz r0
add r11 r11 r1      add r11 r1 r2
load r11 r11        load r9 r11
```

```
3886092288    3768299533
3815792640    3815768084
3800457217    3780874247
3815809024    1249902592
3815796736    3787157515
3767787530    3815768069
3767857162    3780489226
3800735745    444596224
3854483470    3767644166
3854553103    3853234183
3815768084    3815768066
3780554764    3780292614
1249902592    444596224
3768299521    3767644162
3854282763    3854274571
```

Outline

- Pleasantness Problem
- Refinement
- Local Reasoning
- Pipelined Machine Verification
- Automating Refinement
- Refinement Map Factor
- Compositional Reasoning
- Deductive Methods & Decision procedures
- Conclusions

Conclusions

- Hardware verification is a major challenge.
  - What do we prove and how do we automate verification?
- We presented a theory of refinement with advantages over classical work of Abadi & Lamport.
  - We do not require machine closure, finite invisible nondeterminism, or internal continuity.
  - No history variables (blow up the state space of the implementation).
  - No prophecy variables (destroy the branching structure of the implementation and to slow down the implementation).
  - We address their question about about proving liveness properties for systems that are not internally continuous.
- We showed how to apply it to pipelined machine verification.
  - Reduced proof obligations to decidable fragments of logic.
  - Automated safety and liveness.
  - Showed that verification times depend on refinement maps.
  - Introduced commitment & intermediate refinement maps.
  - Developed compositional reasoning framework.
  - Showed how to verify executable, bit-level machines by combining deductive methods & decision procedures.

Future Work

- Automation is a major challenge.
  - Even “automatic” methods require human effort.
  - The languages and tools used are important.
  - Understanding counterexamples is important.
- RTL verification.
  - By combining ACL2 & UCLID we avoided heroic effort.
  - We verified the numerous abstractions used in term-level modeling.
  - We were able to relate term-level models with RTL-level designs.
  - Challenge is to reduce ~4x increase in effort to $1+\varepsilon$.
- Tools that operate directly on HDLs.
  - Current tools support very simple subsets.
  - Develop decision procedures that exploit structure in HDL designs.
  - BAT: Bit-vector analysis tool.