Deductive Verification of Pipelined Machines Using First-order Quantification

Sandip Ray and Warren A. Hunt, Jr.

Department of Computer Science
University of Texas at Austin

Email: {sandip,hunt}@cs.utexas.edu
web: http://www.cs.utexas.edu/users/{sandip,hunt}

Contributions

A theorem proving approach to verify pipelined machines.

- We show how to mechanically derive (stuttering) simulation proofs of pipelines from flush-point proofs.
  - We achieve this by defining a notion of witnessing state.
- We show how to use first-order quantification to specify witnessing states.

Outline

- Microprocessor Correctness
  - Simulation proofs
  - Flush point proofs
- Conversion Approach
- Handling Pipelines with advanced Features
  - Stalls
  - Interrupts
  - Out-of-order execution
- Future Work and Conclusion

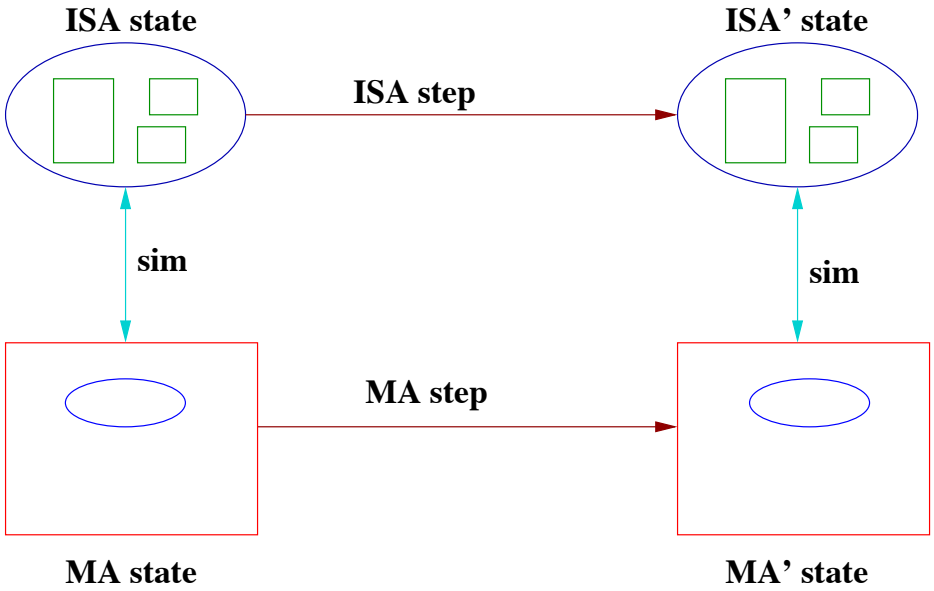
Microprocessor Correctness

Microprocessor correctness involves showing that a microarchitectural implementation (MA) correctly implements its instruction set architecture (ISA).

- ISA features:
  - Non-pipelined.
  - Instructions executed atomically at every step.
- MA features:
  - Pipelined.
  - Out-of-order execution.
  - Multiple instruction commit.

Simulation Correspondence

Simulation correspondence guarantees that for every execution of MA there is an execution of ISA with the same observable (programmer-visible) behavior.



Simulation Correspondence

Simulation correspondence guarantees that for every execution of MA there is an execution of ISA with the same observable (programmer-visible) behavior.

ISA is a simulation of MA if there is a relation *sim* over the states of MA and ISA:

- $sim(MA, ISA)$ implies *MA* and *ISA* have same observations.
- For every next state *MA'* of *MA* there is a state *ISA'* of *ISA* such that $sim(MA', ISA')$.
- For every initial state $init_m$ of MA, there is an initial state $init_i$ of ISA, such that $sim(init_m, init_i)$.

Simulation: Applicability

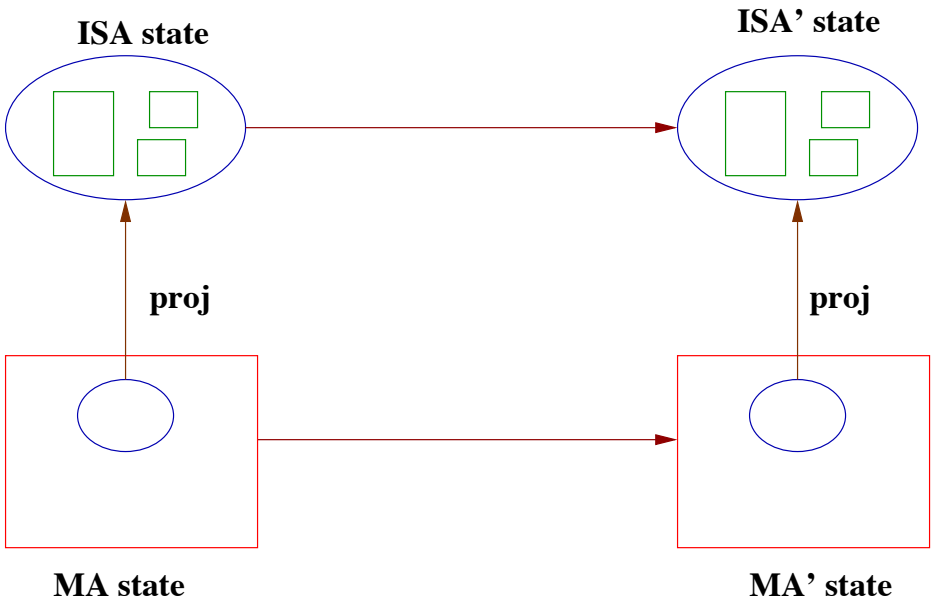
Qualities of *simulation* as a correspondence notion:

- Very generic and well-studied correspondence notion.
- Provides guarantees on (infinite) execution via single-step theorems.
- Hierarchically composible:
  - If A simulates B, and B simulates C, then A simulates C.

For microprocessor verification, *simulation* and its variants are typically used to verify non-pipelined machines.

Simulation for Microprocessors

We typically define a *simulation* relation by *projecting* the programmer-visible components.



$$sim(MA, ISA) \doteq (proj(MA) = ISA)\,.$$

Pipelines and Correspondence

We cannot specify such a simple *simulation* if MA is pipelined:

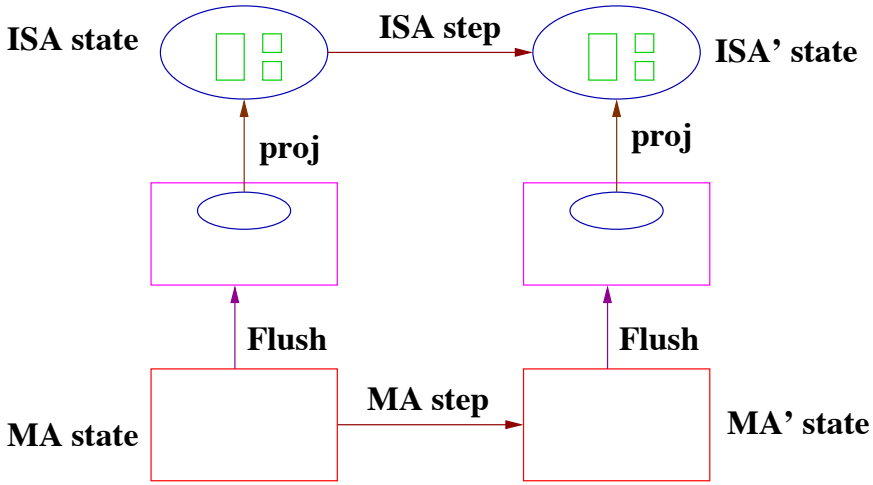
- When an instruction is completed in MA, some subsequent instruction has already been partially executed.
- ISA completes each instruction atomically in every step.

Different notions of correspondence are used for pipelined machines [ACDJ]:

- Comparisons at flush point.
- Comparisons based on stuttering simulation and bisimulation.

Burch and Dill Correspondence

- *Flush* the pipeline by completing all in-flight instructions but without fetching any new one.
- *Project* the programmer-visible components of the *flushed* state.



Burch and Dill: Applications

Burch and Dill (and its variants) have been used for verification of practical pipelines [SW, HSG].

- The flush signal of the pipeline is used to map MA state to ISA state.
- Verification can be typically achieved by symbolic simulation on MA.

Problems:

- Hierarchical composition is difficult.
- Further, Manolios [Man] shows variants of Burch and Dill notion are flawed.

Stuttering Bisimulation

A stuttering bisimulation (WEB) correspondence can be shown between pipelined MA and ISA [Man].

- Shows that MA and ISA have the same observable behavior up to finite stuttering.
- Preserves both safety and liveness properties.
- Applicable to deterministic and non-deterministic machines.
- Can be hierarchically composed.

Our Objective

- Flush point correspondences have been widely used to verify pipelined machines.
- Stuttering (bi)simulation is more satisfactory (and useful) as a correctness criterion.

Can we set up a correspondence framework so that when *some* flush-point correspondence is proven, we can derive a simulation-type theorem?

Outline

- Microprocessor Correctness
  - Simulation proofs
  - Flush point proofs
- Conversion Approach
- Handling Pipelines with advanced Features
  - Stalls
  - Interrupts
  - Out-of-order execution
- Future Work and Conclusion

Stuttering Simulation

We use stuttering simulation with one-sided stutter as our correctness criterion.

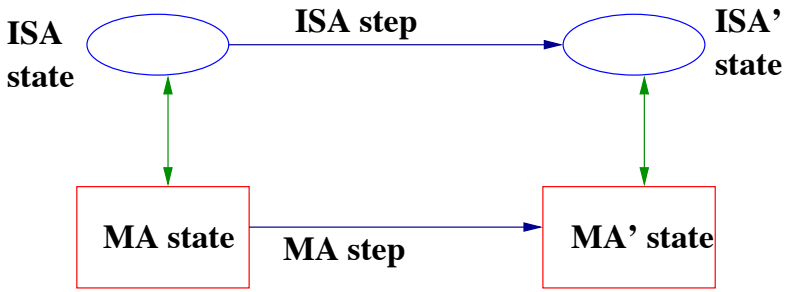
- The ISA is allowed finite stutter. MA never stutters.
  - If *some* instruction is completed at the current step by MA, then ISA matches the step, otherwise ISA stutters.
  - Finiteness of stuttering is guaranteed by arguments based on well-foundedness.
- Our notion is a (restricted) variant of WEBs [Man].

Basic Approach

We will show how to overlay a flush-point proof to determine stuttering simulation.

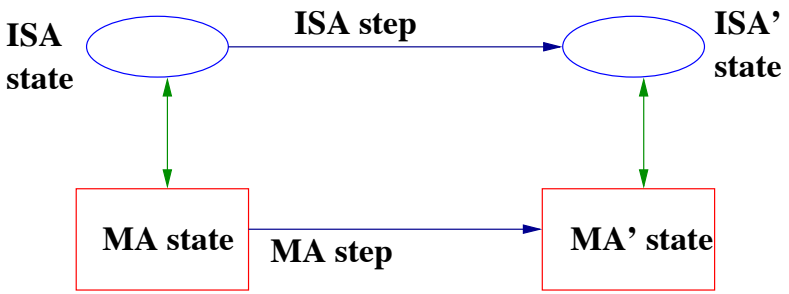
Basic Approach

We will show how to overlay a flush-point proof to determine stuttering simulation.



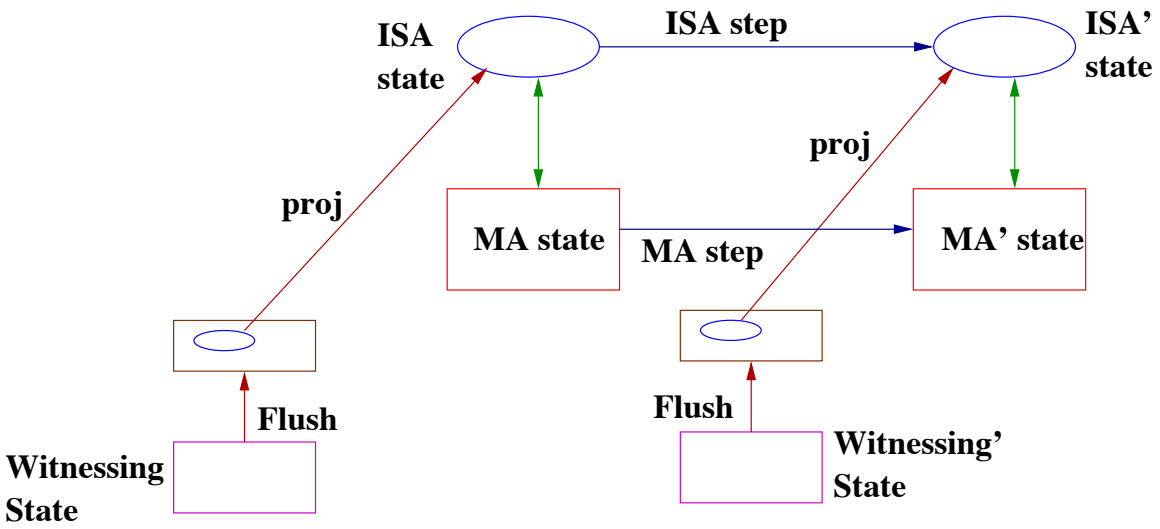
Basic Approach

We will show how to overlay a flush-point proof to determine stuttering simulation.



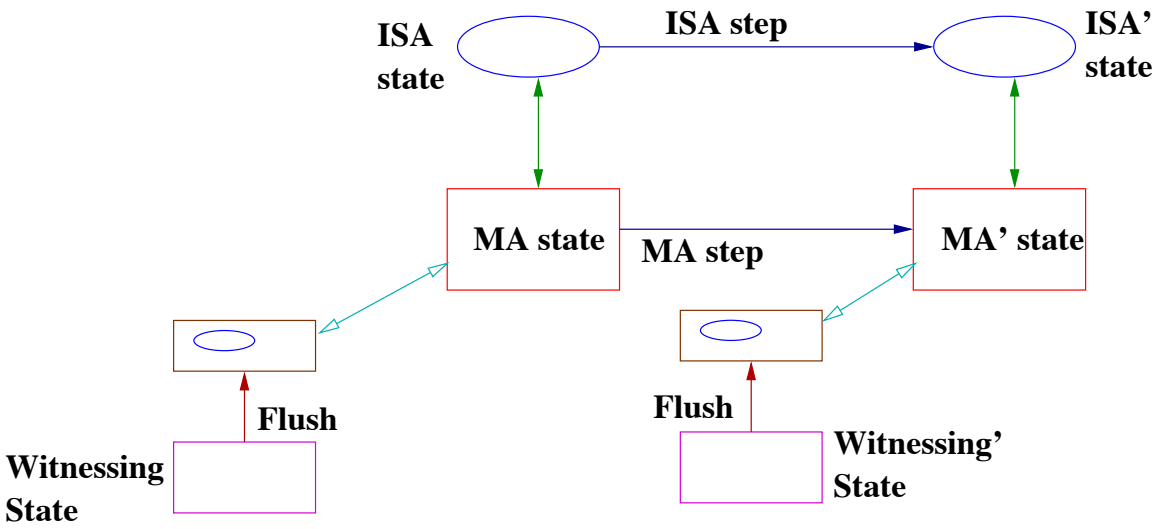
Basic Approach

We will show how to overlay a flush-point proof to determine stuttering simulation.



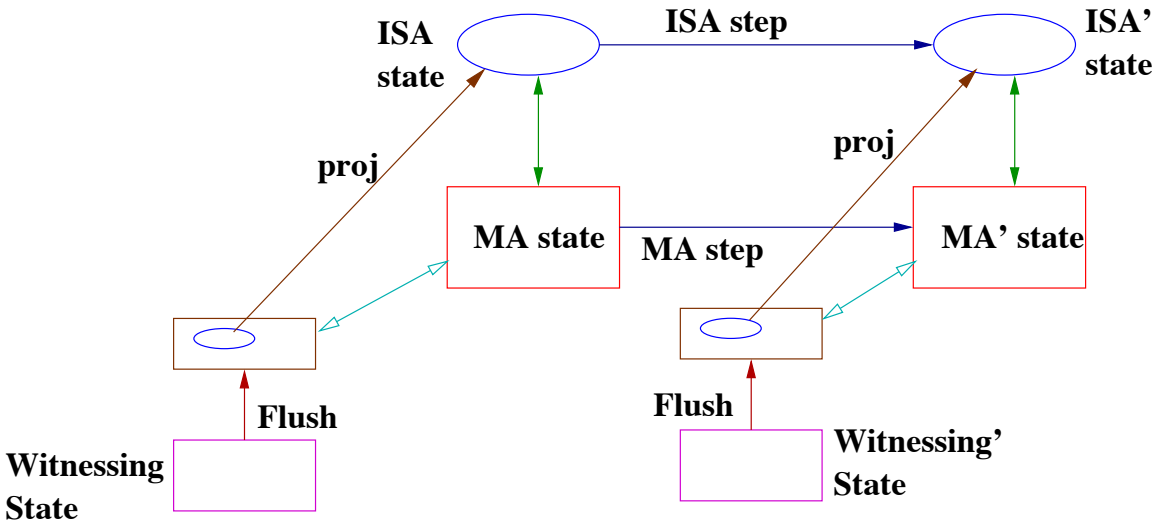
Basic Approach

We will show how to overlay a flush-point proof to determine stuttering simulation.



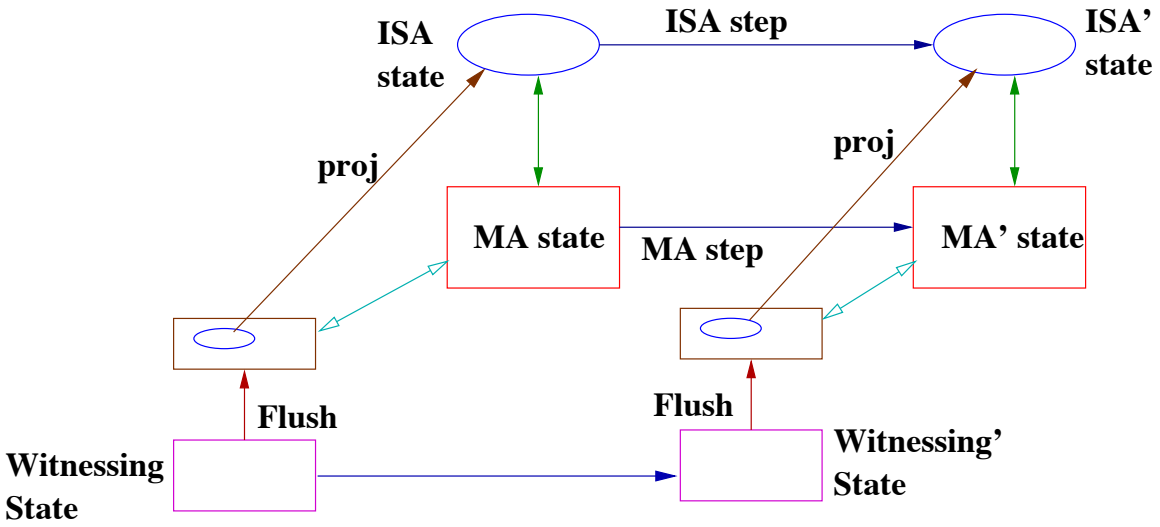
Basic Approach

We will show how to overlay a flush-point proof to determine stuttering simulation.



Basic Approach

We will show how to overlay a flush-point proof to determine stuttering simulation.



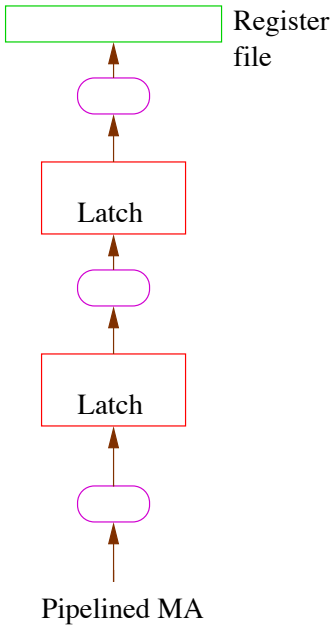
What is a witnessing State?

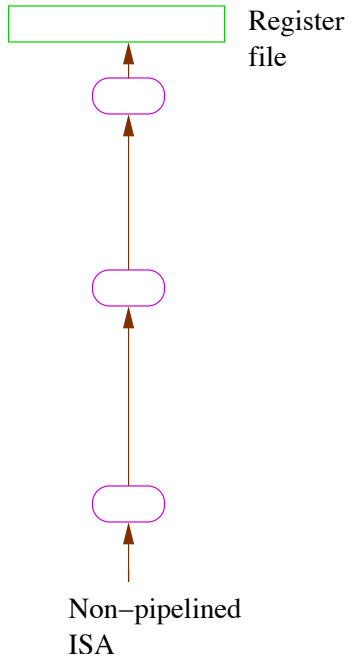
- For a current state *MA*, the witnessing state is one that has been encountered in the past.
- The witnessing state is one in which the instruction next to commit in *MA* was entering the pipeline.

How do we get such a state, and how do we use it to show stuttering simulation?

A Simple Example

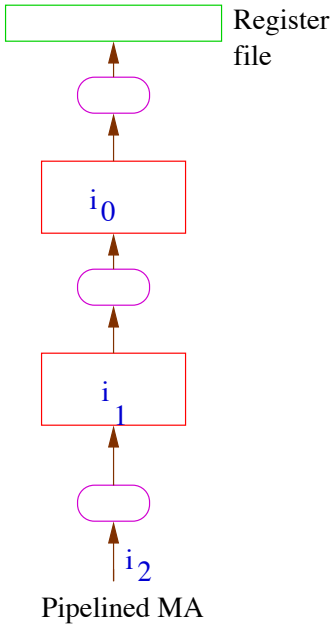
Consider a trivial pipelined machine and the corresponding ISA.



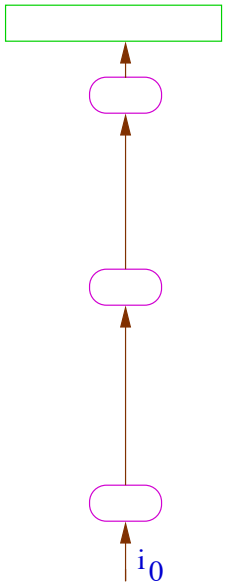


Approach

Consider the state in which instruction i_0 is about to be completed in the pipeline. If we need to match the updates of register file with the ISA then i_0 must be about to be executed in the ISA.



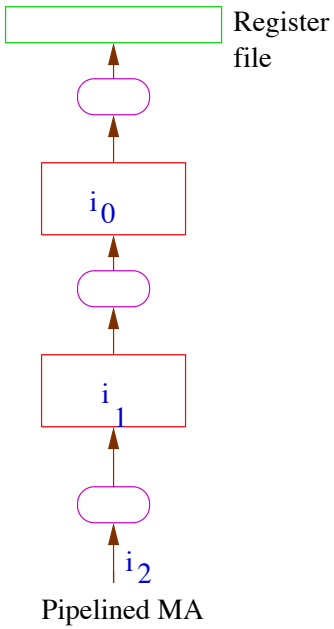
Pipelined MA



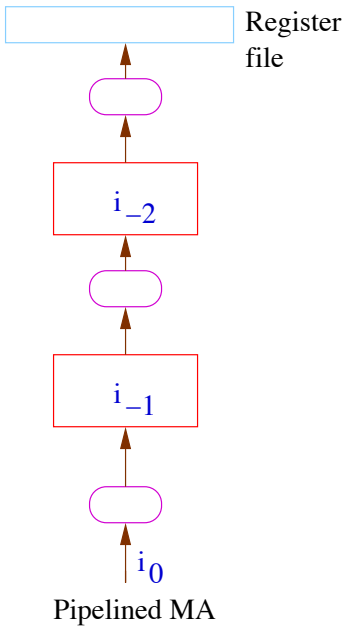
Non–pipelined
ISA

Witnessing State

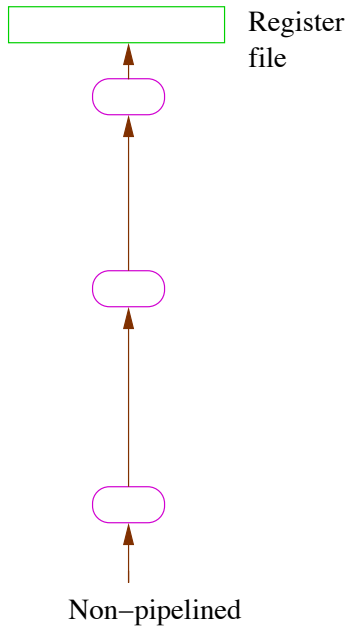
The machine must have encountered a state in the past when instruction i_0 was about to enter the pipeline. *That* state is the witnessing state.



Pipelined MA



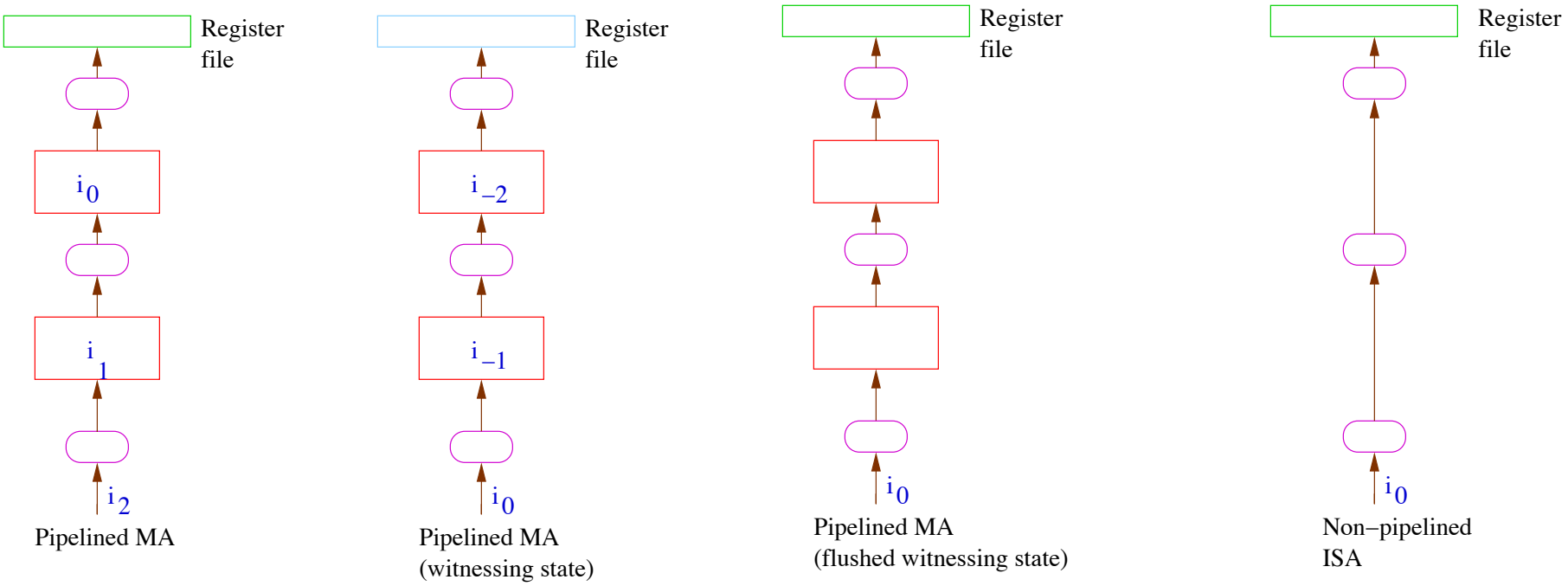
Pipelined MA
(witnessing state)



Non–pipelined
ISA

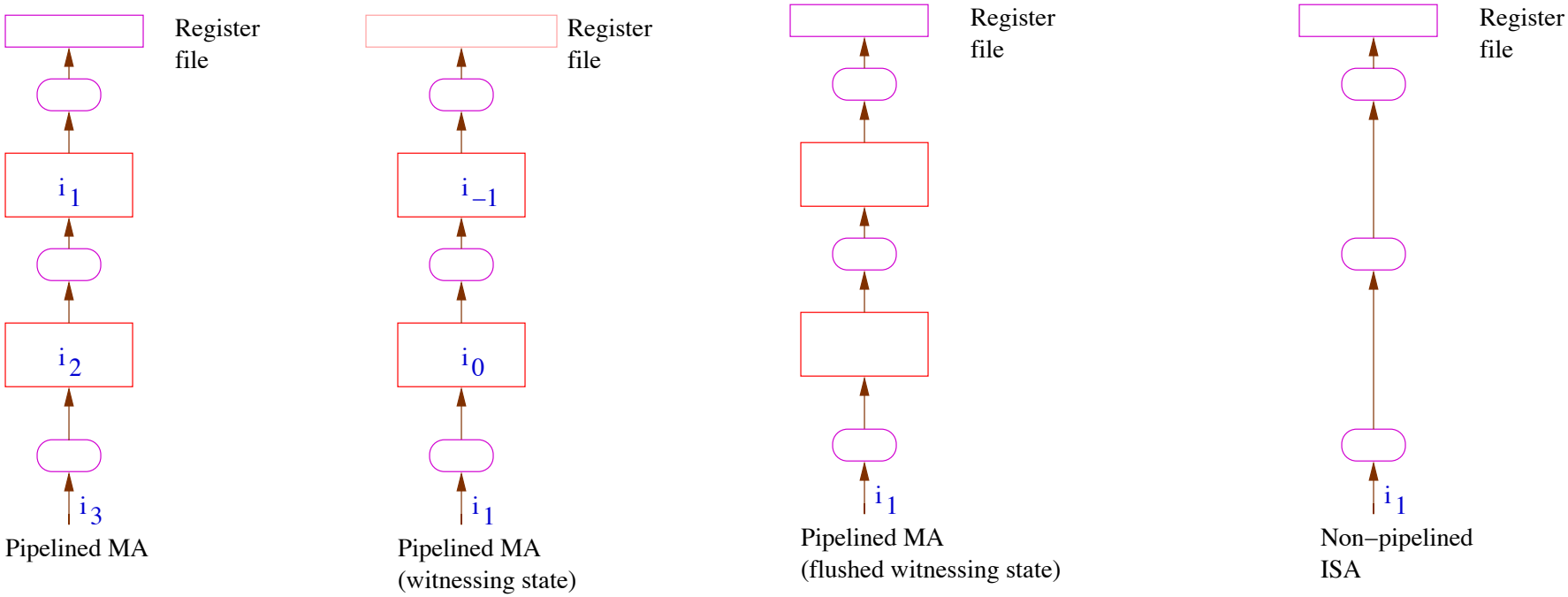
An Observation

Now *flush* from the witnessing state. You must have a state with the same updates as the original state.



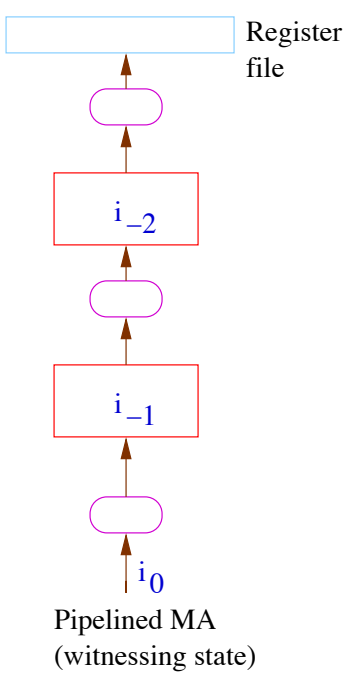
One Step Next

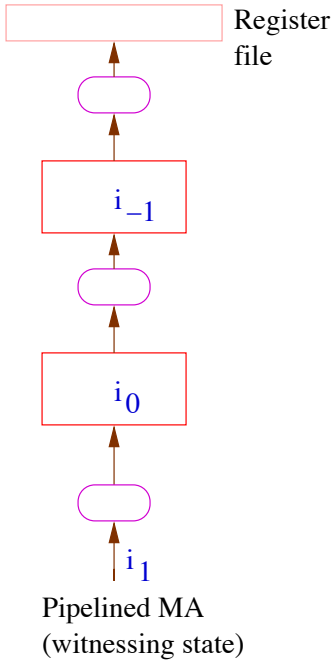
Consider the scenario one step forward:



Another Observation

Let us compare the two witnessing states:

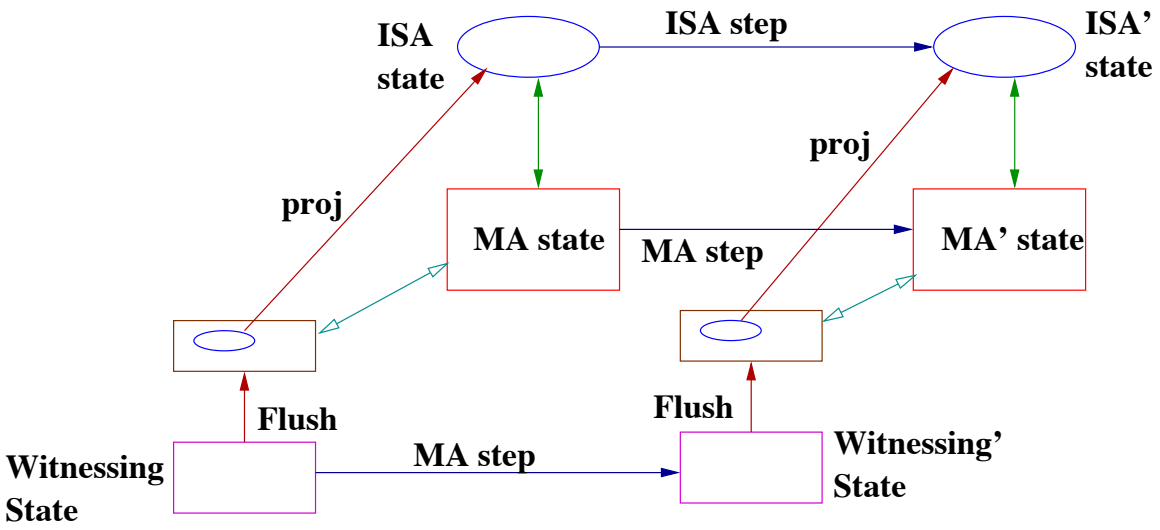




But the second one is merely one step away from the first one!

What All This Really Gives Us

We have thus overlayed a *flush-point* proof to determine stuttering simulation, *if we can find the witnessing state*.



The "trick" was to flush from the witnessing state rather than from *MA*.

Verification Approach

How do we find the witnessing state given some state MA?

- *Constructing* the witnessing state requires keeping track of history of execution.
  - But, the following predicate is an invariant of MA:
    - $witness\text{-}exists(MA) \doteq \exists MA_1 : witness(MA_1, MA)$.
- We can specify the simulation relation using the *Skolemization* of this predicate as our witnessing state.

Outline

- Microprocessor Correctness
  - Simulation proofs
  - Flush point proofs
- Conversion Approach
- Handling Pipelines with advanced Features
  - Stalls
  - Interrupts
  - Out-of-order execution
- Future Work and Conclusion

Elaborate Pipelines

The central step is to find the witnessing state. We can do this for:

- Arbitrary but finite stalls
- Interrupts
- Out-of-order execution with in-order commit

We **cannot** handle:

- Multiple-instruction commit
- Out-of-order commit

Stalls

If a pipeline is stalled, then no instruction enters the pipeline at that state.

- If MA_1 is a witnessing state of MA then MA_1 is not stalled.

If MA_1 is the witnessing state of MA then what will be the witnessing state of the next state MA' of MA?

- The "next non-stalled state" after MA_1.

For deterministic machines, the state is given by $clock(step(MA_1))$:

$$clock(s) \doteq \begin{cases} 0 & \text{if } \neg stalled(s) \\ 1 + clock(step(s)) & \text{otherwise} \end{cases}$$

Interrupts

Both ISA and MA are interruptible.

- The witnessing state is either:
  1. A state in which MA entered the interrupt service (if MA is currently processing an interrupt), or
  2. When the instruction next to be committed came in (if MA is not currently processing interrupts).

- In this case, the machines are non-deterministic.

We have not analyzed machines with nested interrupts.

Out-of-order Execution

We can only deal with out-of-order execution when:

- instructions enter the pipeline and are committed in program order, and
- at most one instruction is committed in one step.

To find the witnessing state, we need to find the next instruction to be committed.

- This can be done by symbolic simulation of the pipeline *forward*.

Multiple Instruction Commit

If MA commits multiple instructions in one step, then we **cannot** handle such a pipeline.

- ISA executes only one instruction in each step.
  - There is *no* simulation relation such that ISA simulates MA.

We will need to change the ISA so that it can execute a sequence of instructions in each step.

Theorem proving

All the proofs have been done with the ACL2 theorem prover.

- We use ACL2's *encapsulation* feature to verify generic pipelines.
  - We can encapsulate the functionality of the different components that are visible to both MA and ISA, and only reason about the control flow.
  - We can *instantiate* such generic proofs with concrete pipeline implementations.

The applicability of the method depends on the efficiency of flush-point proofs.

Outline

- Microprocessor Correctness
  - Simulation proofs
  - Flush point proofs
- Conversion Approach
- Handling Pipelines with advanced Features
  - Stalls
  - Interrupts
  - Out-of-order execution
- Future Work and Conclusion

Drawbacks and Future Work

- We have not applied the approach to verify large practical pipelines.
  - All our proofs are on *toy* examples.
  - We are planning to verify a substantial pipeline module of a microprocessor.
  - The main complexity in such a verification is anticipated to be on:
    - Determination of witnessing states.
    - Doing the flush-point proof.
- We are working on handling more advanced out-of-order features.
  - A plausible (but not tested) approach is described in the paper.

Conclusion

- We have shown how to derive stuttering simulation proofs of pipelines.
  - We verify a flush-point proof
  - Use quantification and Skolemization to derive a simulation relation.

  An expressive logic is imperative for doing this.
- Our approach works for *generic* pipelines.
  - The generic proof can be instantiated by concrete pipelines.
- In this (and other) verification efforts, we have found quantification extremely helpful.
  - The witnessing state cannot be defined constructively without sufficient history variables and definition of complicated invariants.

References

[ACDJ] M. D. Aagaard, B. Cook, N. Day, and R. B. Jones. A Framework for Microprocessor Correctness Statements. In *CHARME 2001*, pages 443–448. Springer-Verlag.

[HSG] R. Hosabettu, M. Srivas, and G. Gopalakrishnan. Proof of Correctness of a Processor with Reorder Buffer using the Completion Functions Approach. In *CAV 1999*, pages 47–59. Springer-Verlag.

[Man] P. Manolios. Correctness of pipelined machines. In *FMCAD 2000*, pages 161–178. Springer-Verlag.

[SW] J. Sawada and W. A. Hunt, Jr. Trace Table Based Approach for Pipelined Microprocessor Verification. In *CAV 1997*, pages 364–375. Springer-Verlag.