

**Results from More Than Two
Decades of Exploiting Efficient
Abstractions and Translation
to SAT to Formally Verify
Complex
Pipelined/Superscalar/VLIW
Microprocessors**

Miroslav N. Velev



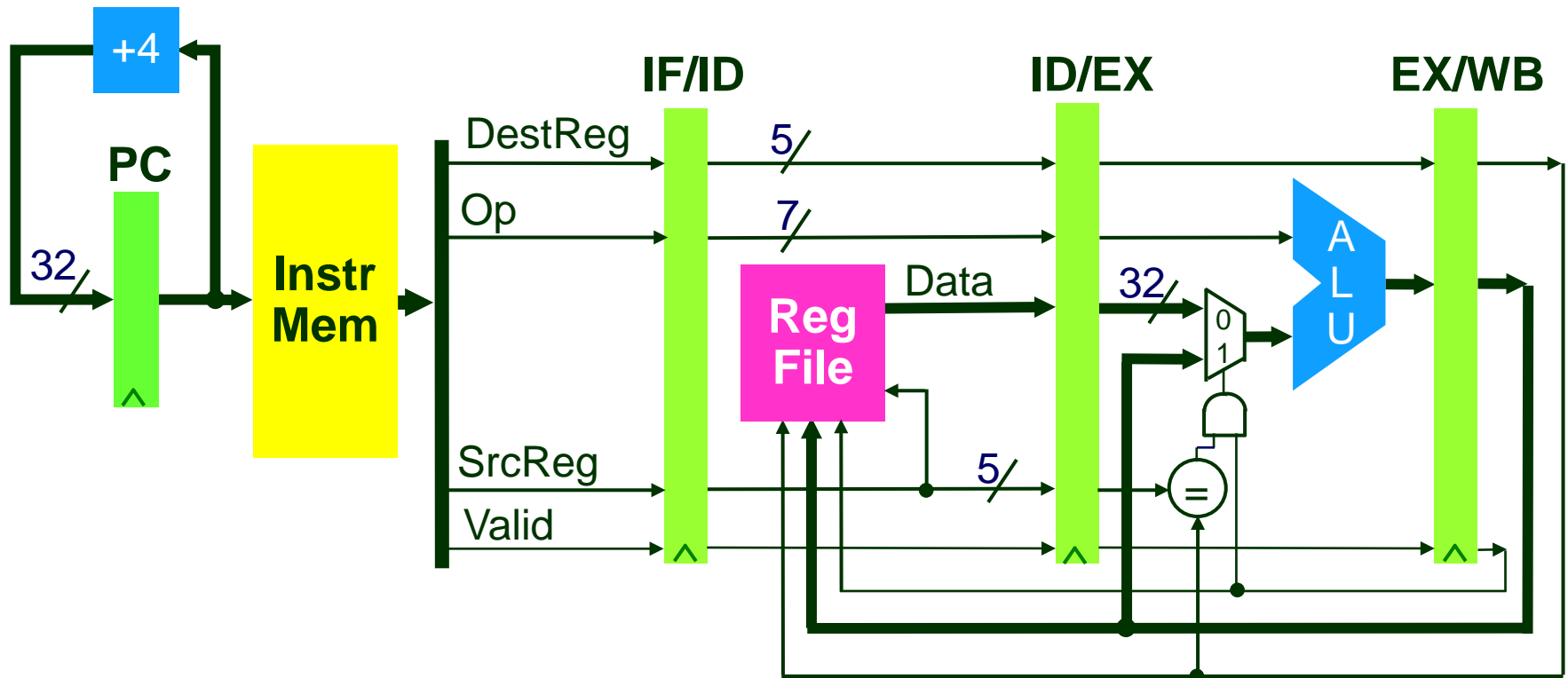
Outline

EUFM Background: Positive Equality

Applications to Formally Verify Different Architectures

Conclusion

Gate-Level Microprocessor



- Data: vectors of wires
- ALUs and memories: gates

Formal verification complexity is exponential

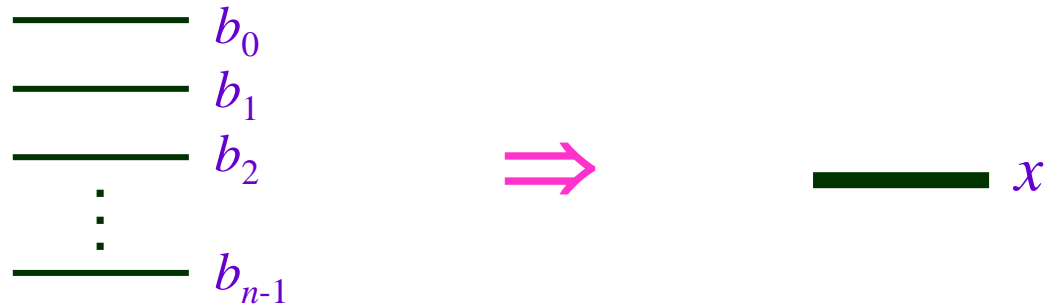
- Velev & Bryant [*FMCAD '98*]

Two-Step Formal Methodology

- 1) Formally verify the Functional Units (FUs) and Memories in isolation from the rest of the design
- 2) Formally verify the pipelined/superscalar/VLIW processor after abstracting the FUs and memories, but keeping the fully implemented control logic, data flow, placement of FUs and memories in pipeline stages
 - ❑ using our tool, HighCheck
 - ❑ applying suitable modeling techniques

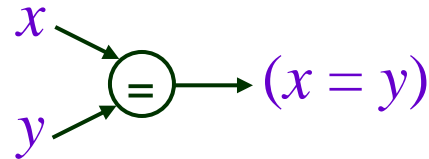
Abstracting Data

Terms abstract data values

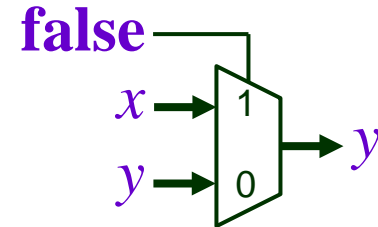
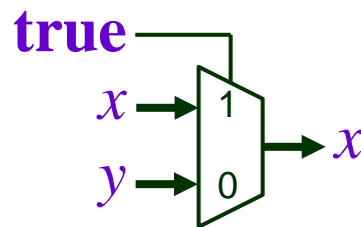
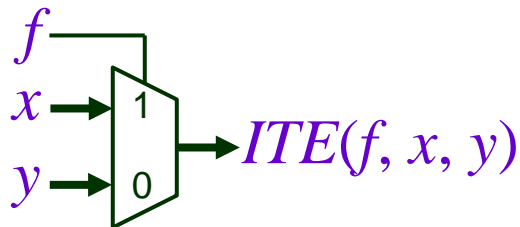


Properties:

- Equality comparison:



- Can be stored in memories
- Can be selected with *ITE* operators:



Abstracting ALUs

Uninterpreted Functions abstract computations

- internal implementation details removed



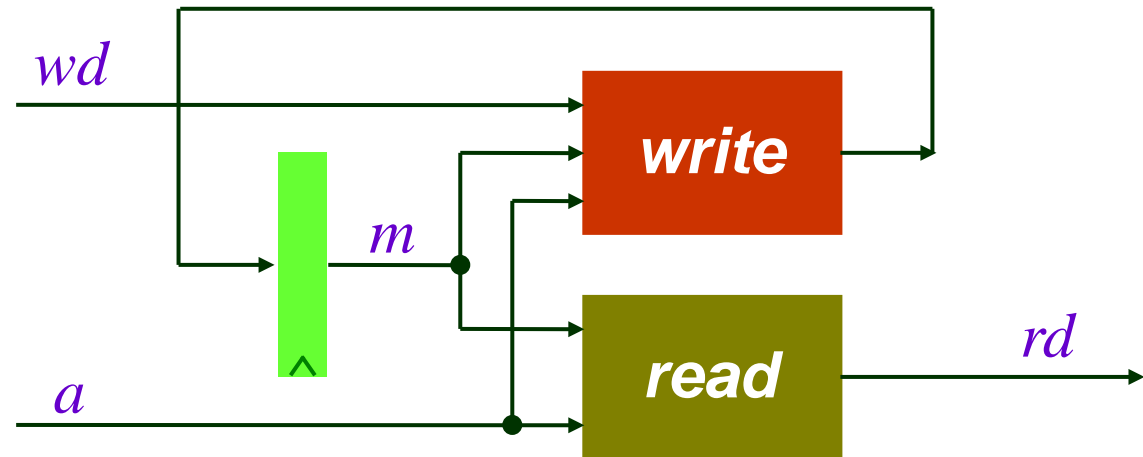
- functional consistency:



$$(x1 = x2) \wedge (y1 = y2) \Rightarrow F(x1, y1) = F(x2, y2)$$

Abstracting Memories

FSM model:

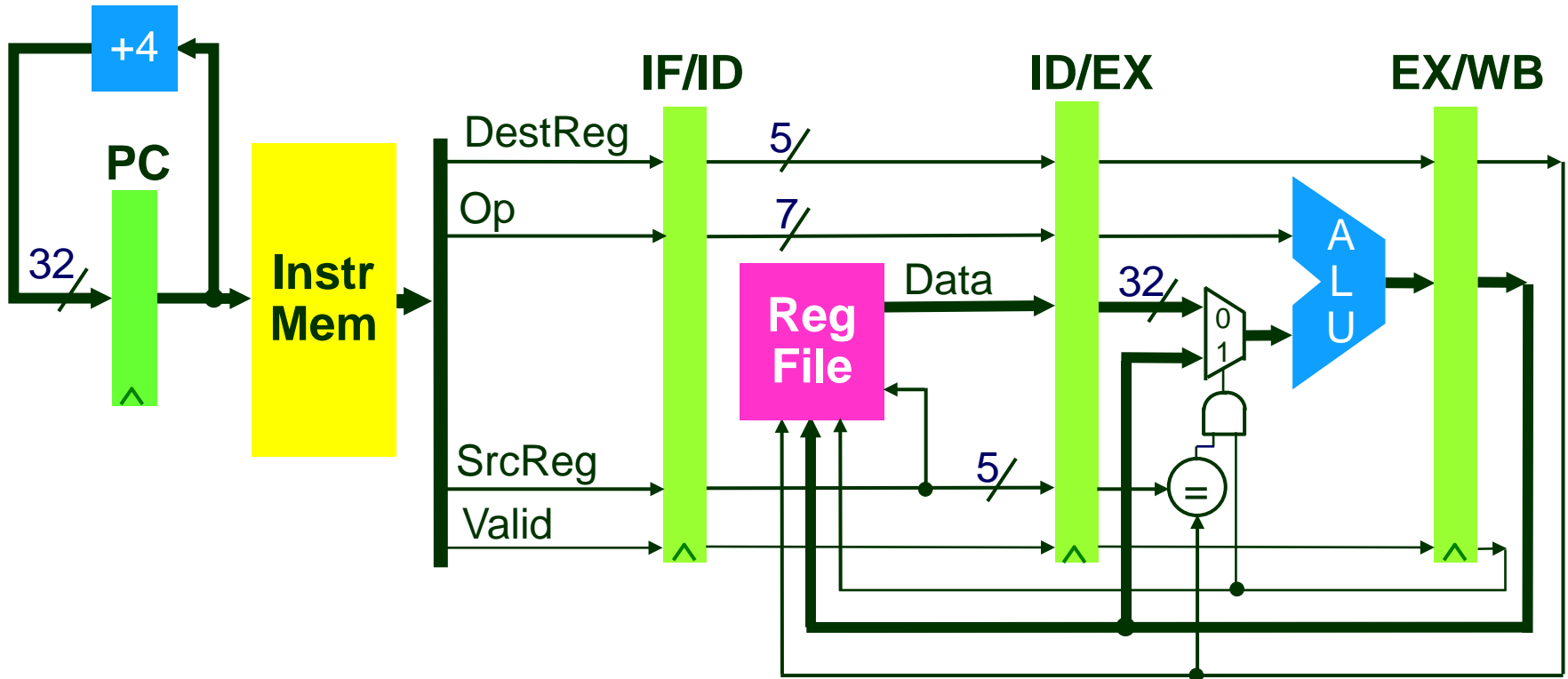


Functions **write** and **read** abstract memory operations

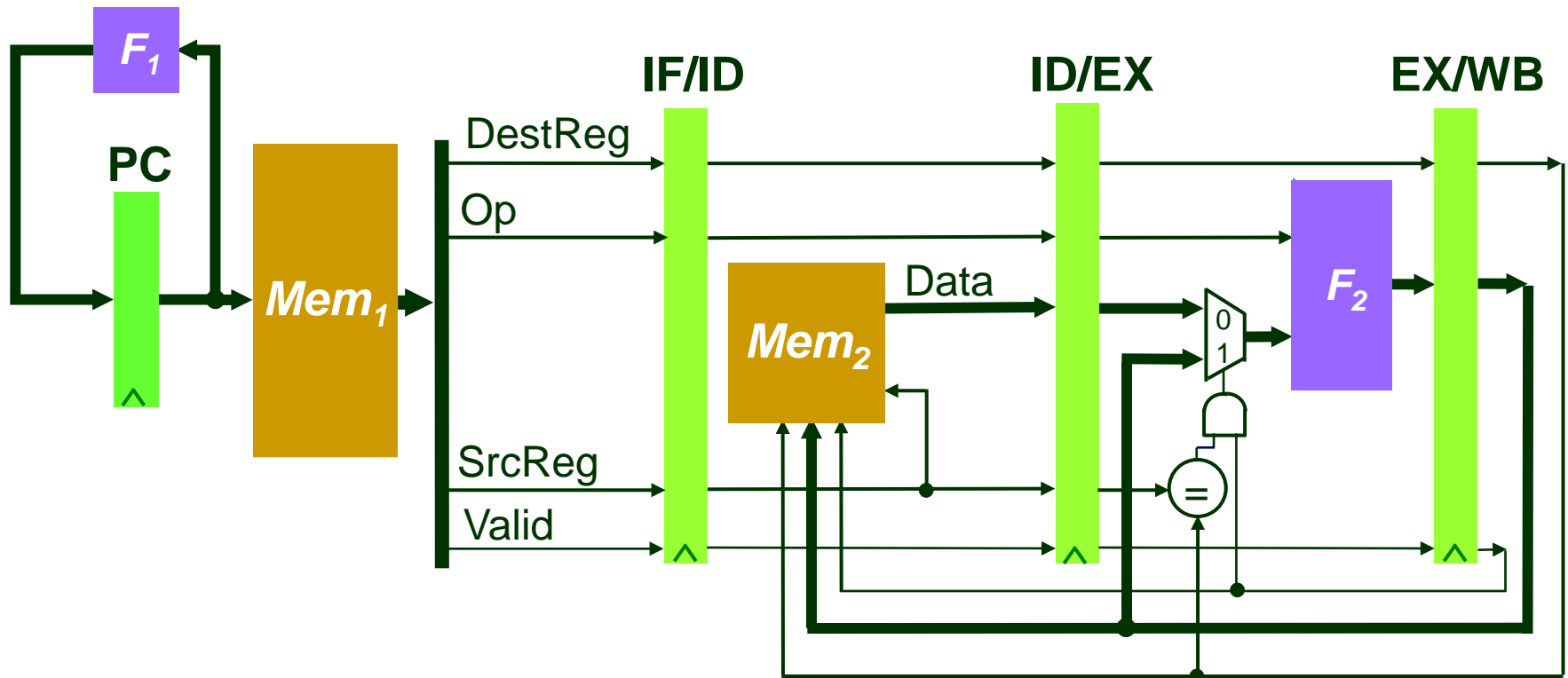
Forwarding property:

$$\mathit{read}(\mathit{write}(m_1, a_1, wd), a_2) = \mathit{ITE}(a_2 = a_1, wd, \mathit{read}(m_1, a_2))$$

Application of Abstractions



Application of Abstractions

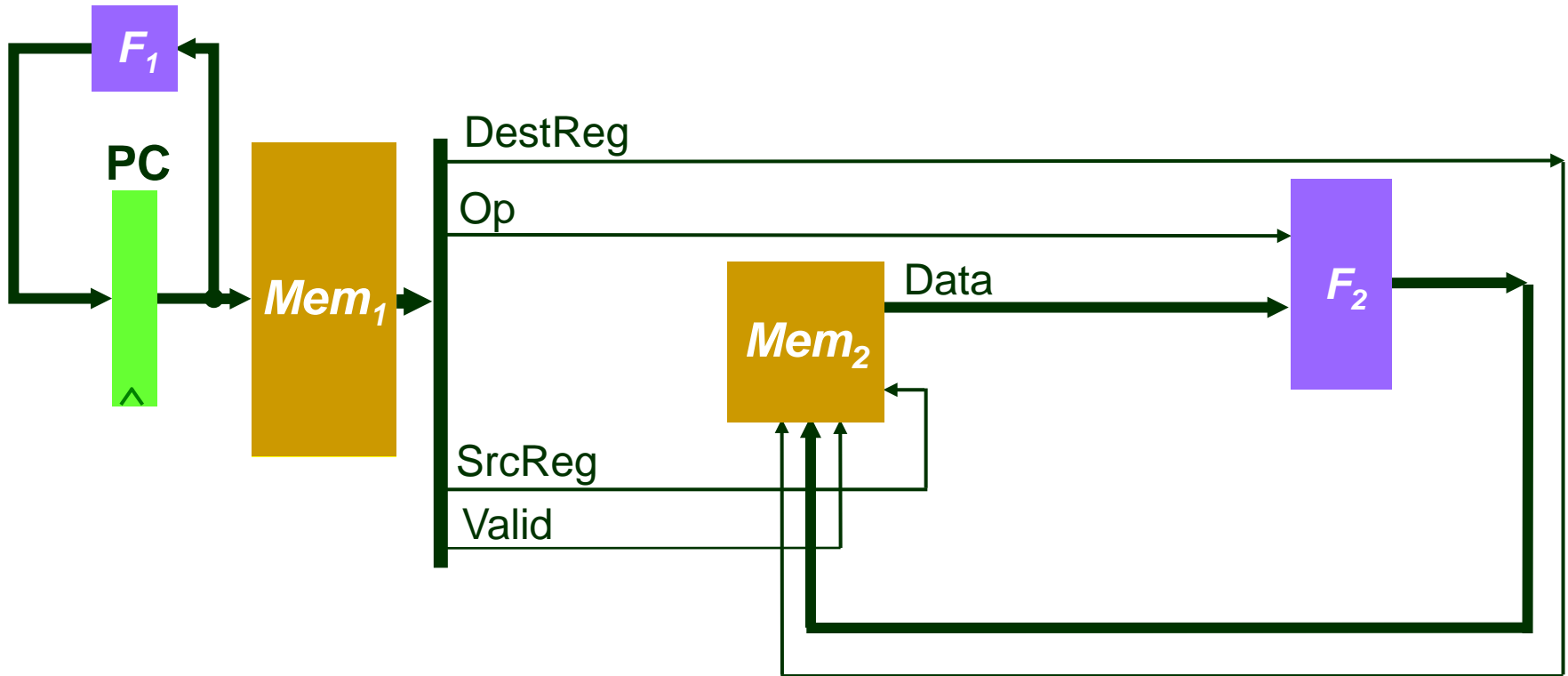


⇒ More general processor

- easier to prove correct

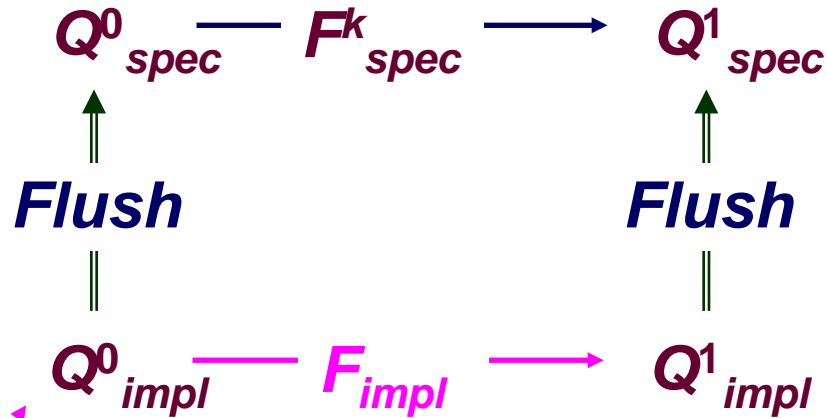
Functional units & memories formally verified separately

Specification Processor



- single-cycle execution
- only user-visible state
- much simpler control logic

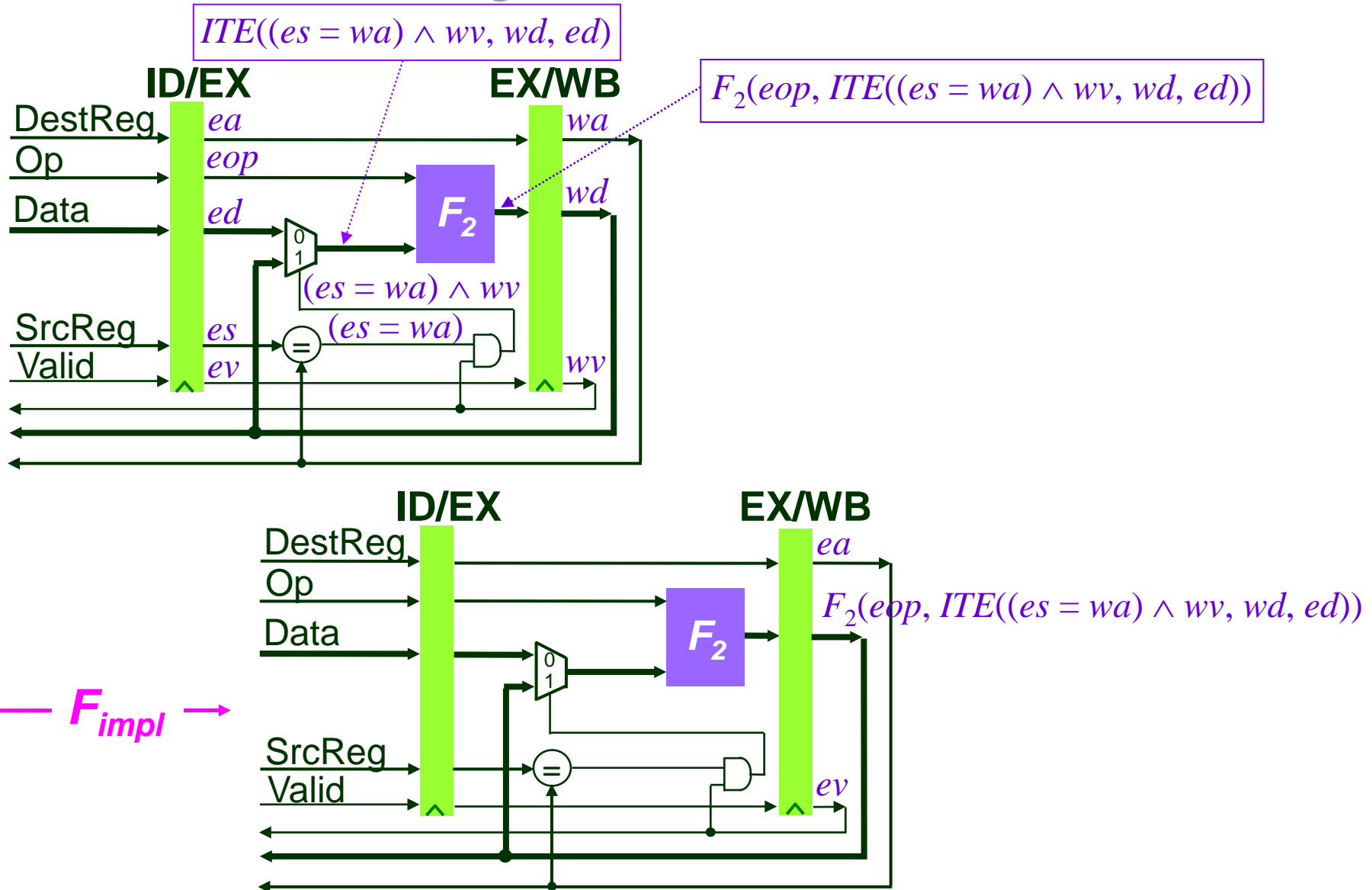
Safety Correctness Criterion



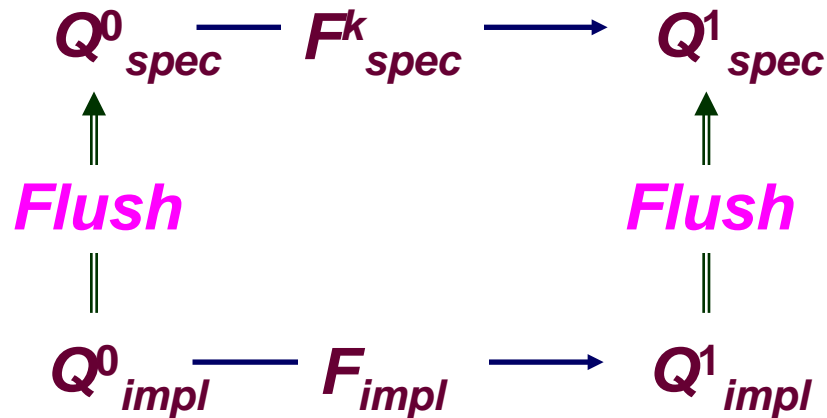
Term-level symbolic simulation
of Implementation for 1 clock cycle

symbolic initial state
(represents
ANY initial state)

Term-Level Symbolic Simulation



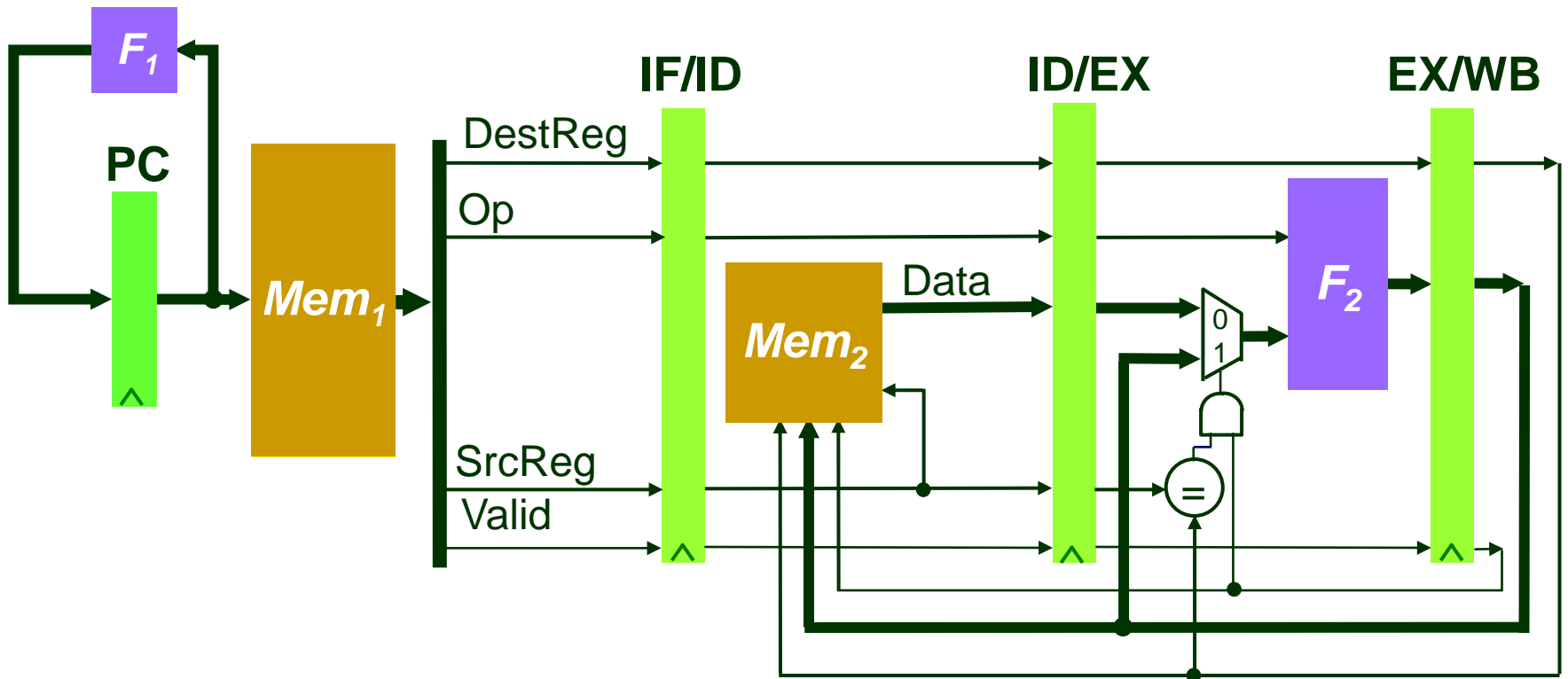
Safety Correctness Criterion



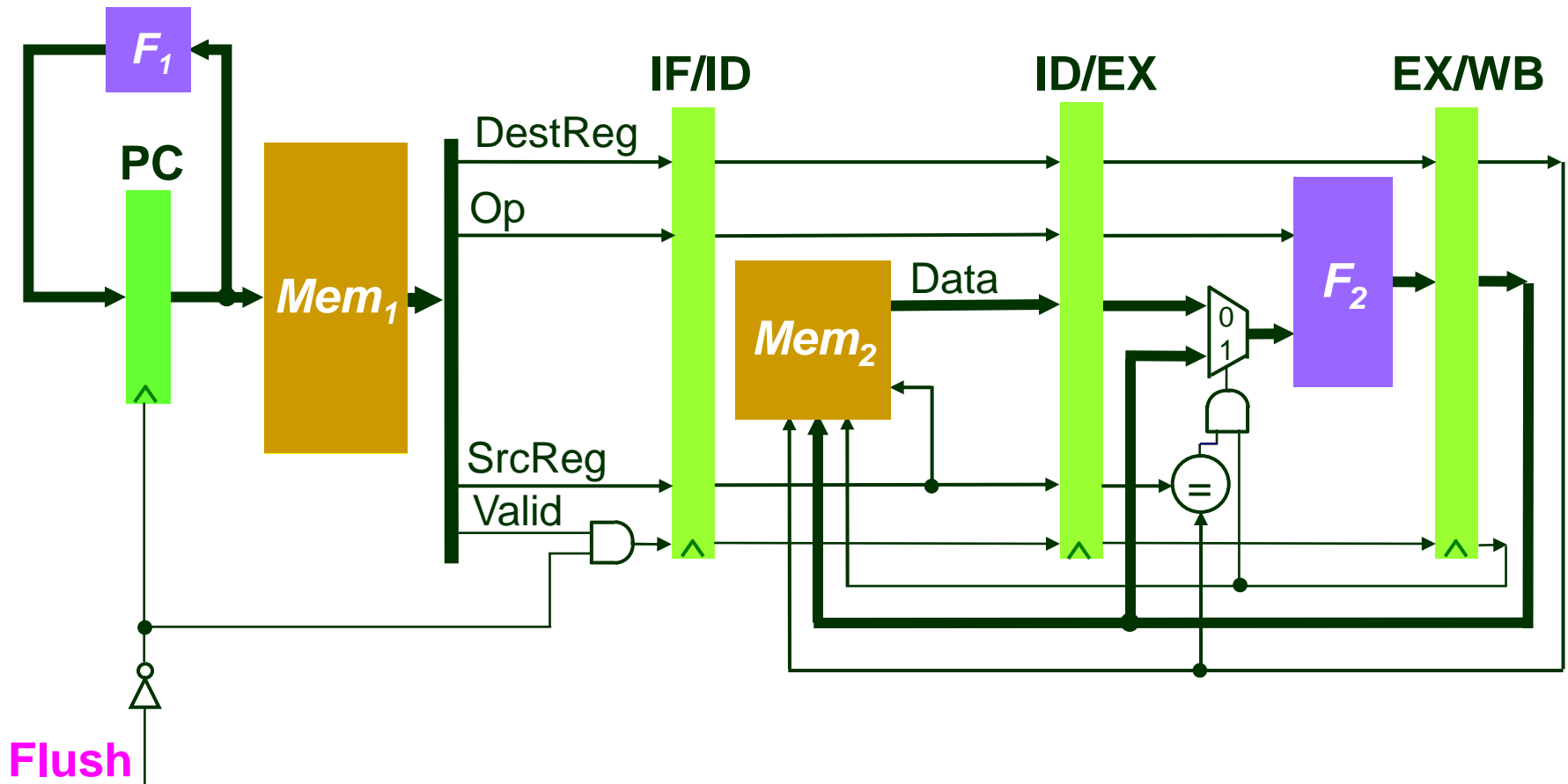
Flush, Burch & Dill [CAV '94]

- automatically maps state of pipeline to user-visible state
- completes partially-executed instructions

Flushing

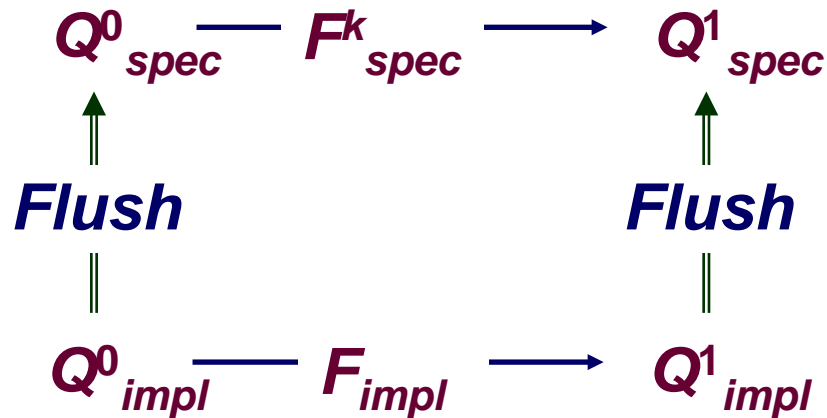


Flushing



- Flush = *false* during regular operation
- Flush = *true* during flushing

Safety Correctness Criterion

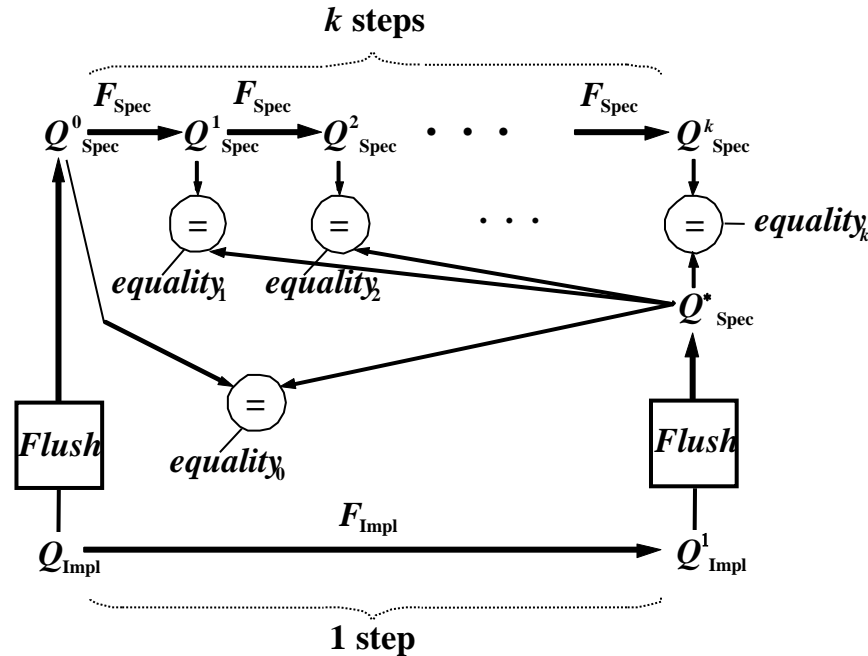


Requirement

- One pipelined Impl step F_{impl} matches up to k Spec steps F_{spec}
 - k is issue-width of processor
 - stalled or canceled instruction: $k = 0$

Safety Correctness Criterion

In the general case: $equality_0 \vee equality_1 \vee \dots \vee equality_k = true$



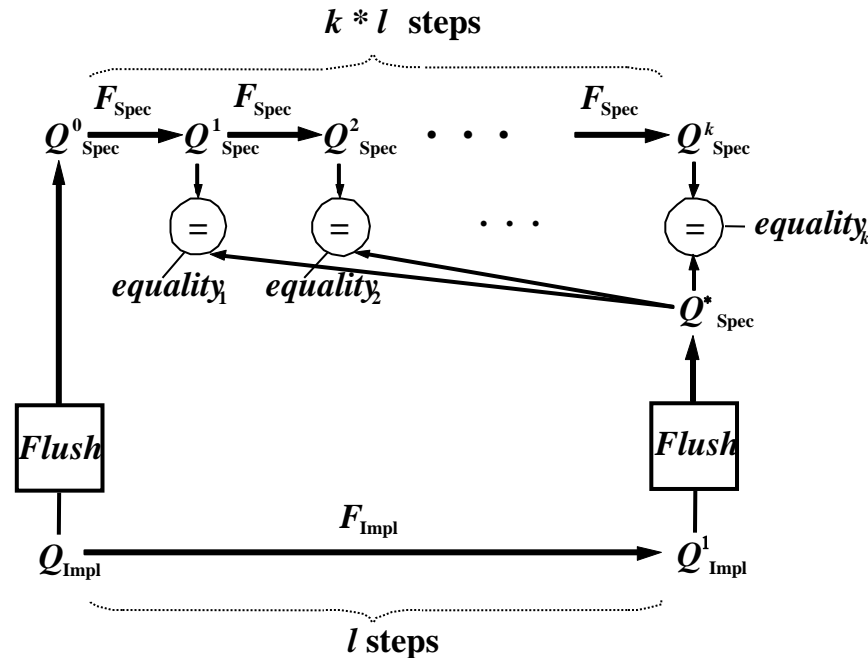
i.e., a proof that 1 step of the Implementation corresponds to between 0 and k steps of the Specification, where k is the issue width of the Implementation

This is the inductive step of proof by induction: initial Impl state Q_{Impl} is arbitrary => **critera will hold from ANY state**, including next Impl state Q^1_{Impl}

$$Q^0_{impl} \xrightarrow{F_{impl}} Q^1_{impl} \xrightarrow{F_{impl}} Q^2_{impl} \xrightarrow{F_{impl}} \dots$$

Liveness Correctness Criterion

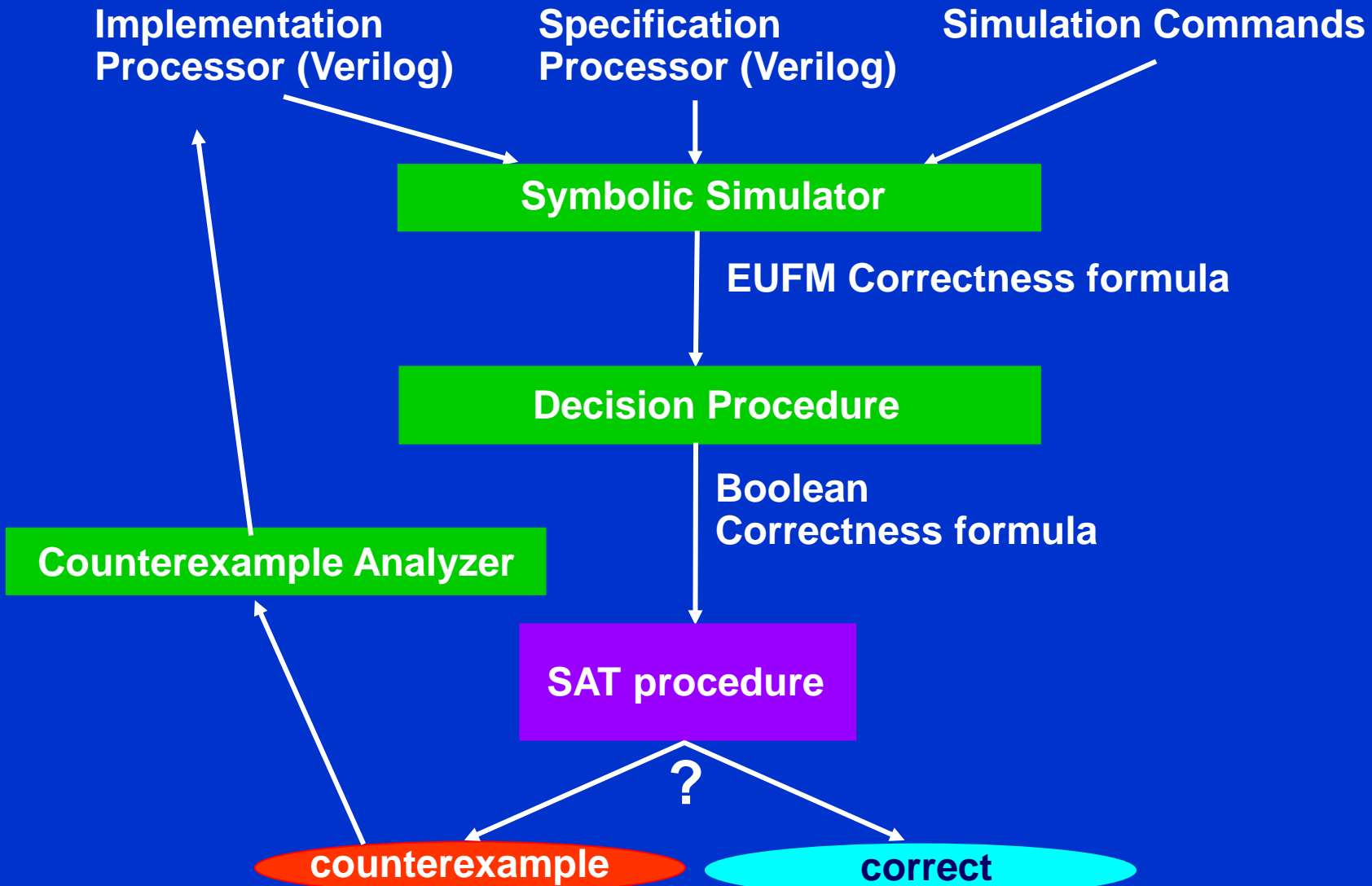
In the general case: $equality_1 \vee \dots \vee equality_{k * l} = true$



i.e., a proof that l steps of the Implementation correspond to between 1 and $k * l$ steps of the Specification, where k is the issue width of the Implementation

Indirect method to prove this property: [Velev, ASP-DAC'04]

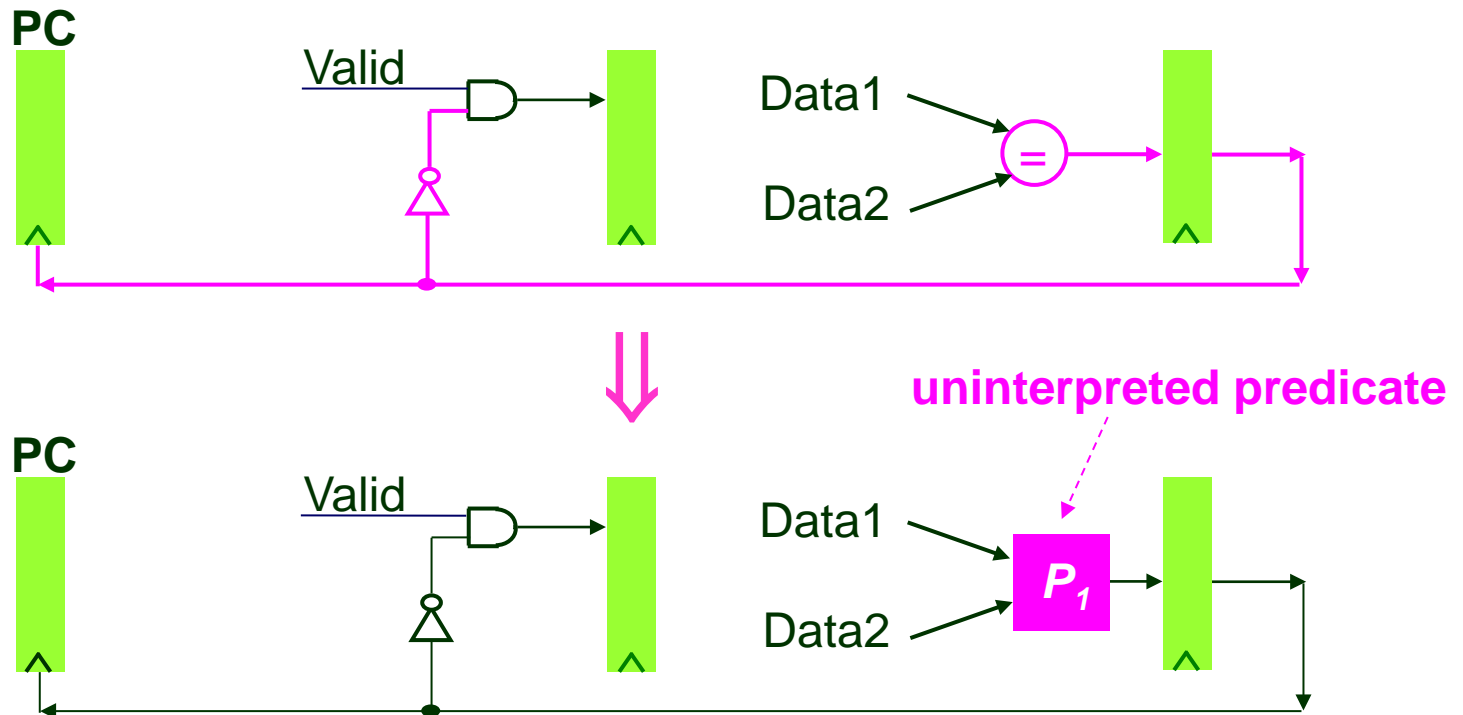
Our Tool: HighCheck



Restriction 1

Abstract data equalities that are both positive & negated

Example 1: Branch-on-equal decisions



Note: Can still model the same features

Restriction 2: Data Memory Model

read and **write**: abstract memory operations

$$m2 \leftarrow F_1(m1, a1, wd)$$

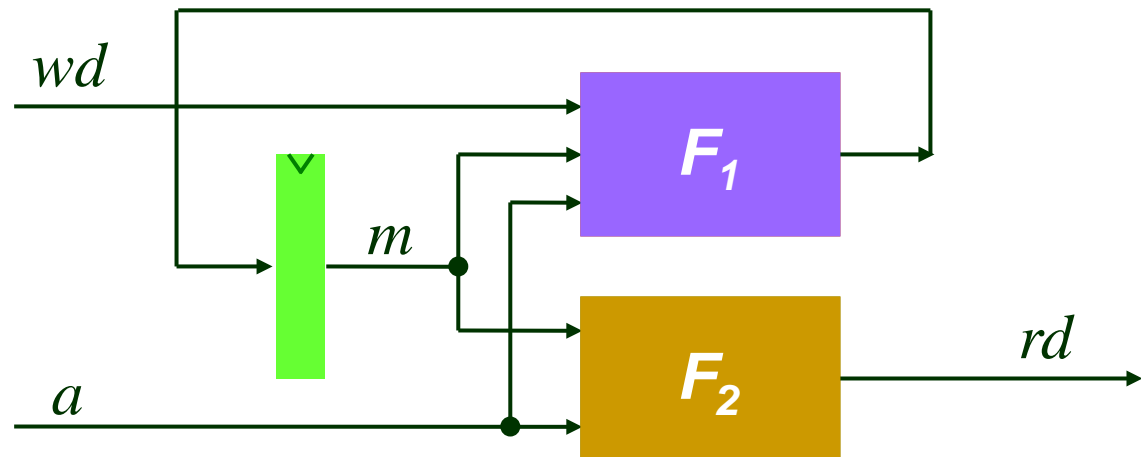
$$rd \leftarrow F_2(m2, a2)$$

Forwarding property **NOT enforced**

$$rd \approx \text{ITE}(a2 = a1, wd, \text{read}(m1, a2))$$

Conservative approximation of memory

FSM model:



Positive Equality

P-terms are compared only in positive equations

- **Connected only with monotonically positive operators AND, OR**

G-terms are compared in both positive and negated equations

As a result of the restrictions, most of the terms become p-terms and can be treated as **DISTINCT CONSTANTS**

G-terms are assigned small domains of values that have to be indexed with fresh Boolean variables

Outline

✓ EUFM Background: Positive Equality

Applications to Formally Verify Different Architectures

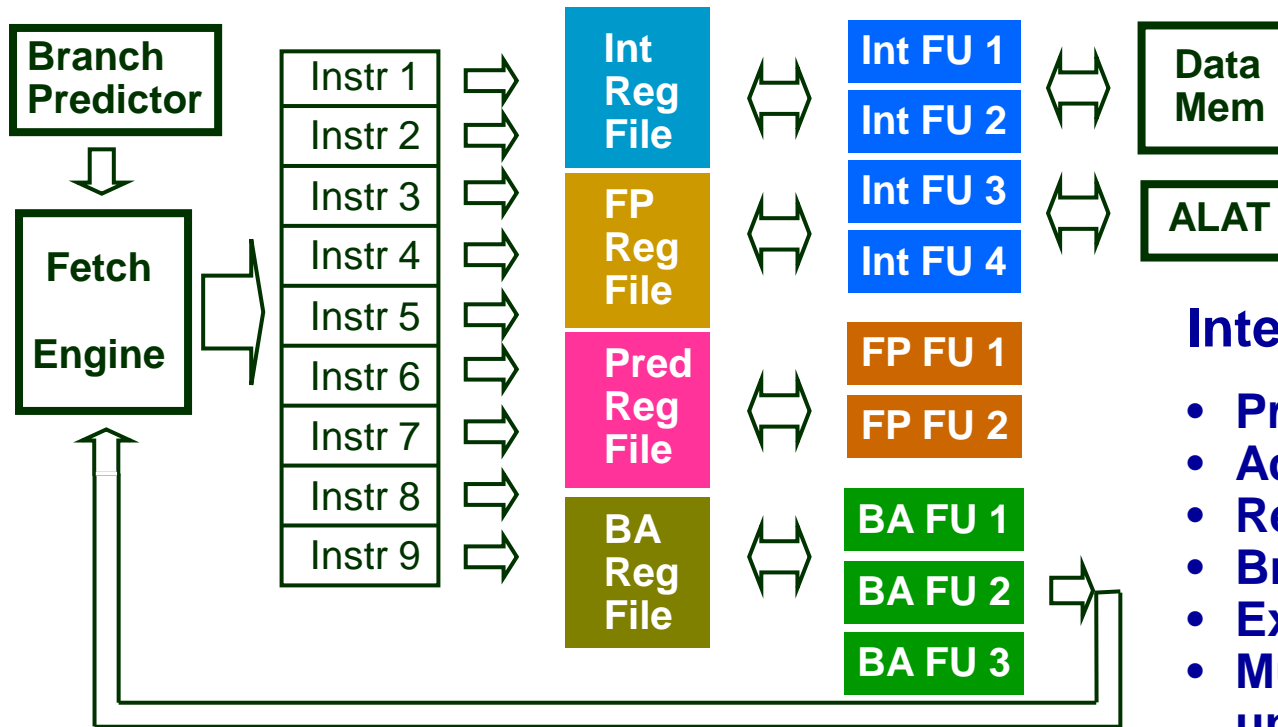
Conclusion

Results

Our tool flow scales for formally verifying correctness of:

- **complex pipelined/superscalar/VLIW processors with many features:**
 - branch prediction
 - exceptions
 - multicycle functional units
 - advanced and speculative loads
 - predicated execution
 - register remapping
 - out-of-order execution based on a reorder buffer
 - delayed branches
 - data-value prediction
 - mechanisms to correct soft errors by re-executing affected instructions
 - reconfigurable functional units
 - arrays of reconfigurable processing elements
 - multi-threaded execution
 - reconfigurable polymorphic heterogeneous multi-core architectures
- **executable code for a given Instruction Set Architecture, including cybersecurity properties.**

We Formally Verified VLIW Processor (DSP) Based on Intel Itanium



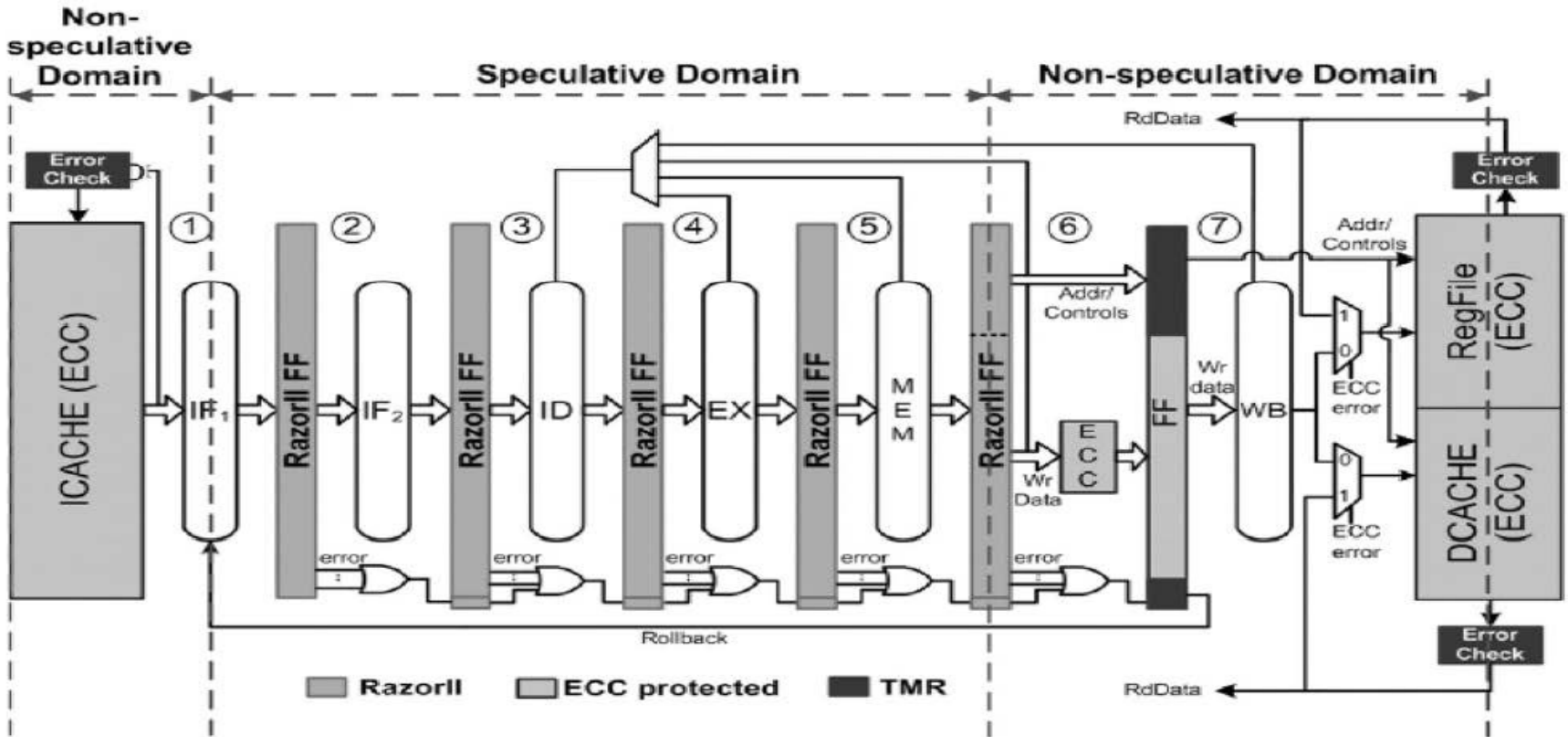
Intel Itanium® features:

- Predicated execution
- Advanced loads
- Register remapping
- Branch prediction
- Exceptions
- Multicycle functional units

42 VLIW instructions
9 pipeline stages
4 VLIW-instruction queue

13 minutes to formally verify on 1 CPU core

FV of Pipelined Processors That Detect & Correct Soft Errors



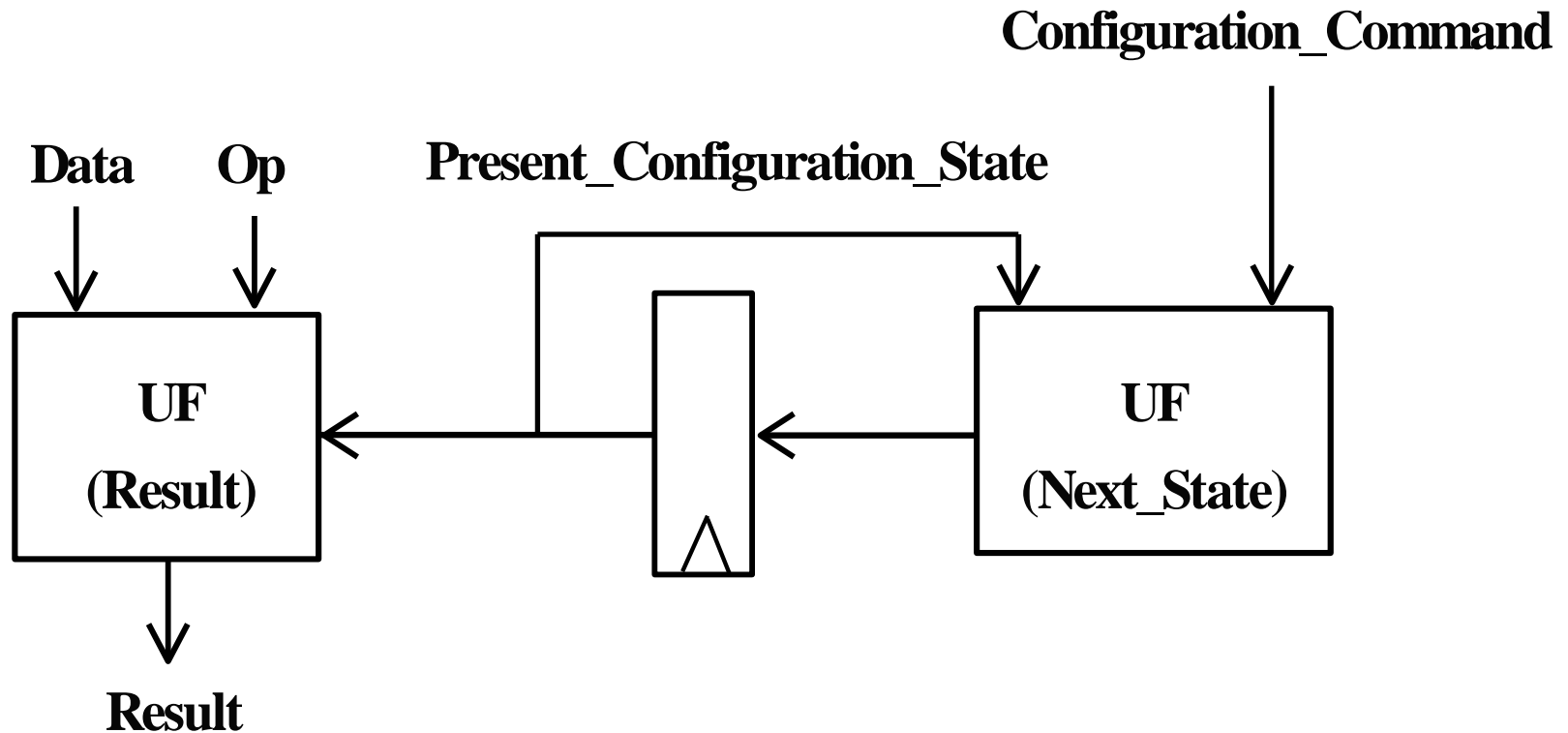
RazorII fault-detecting flip-flops [Das et al. 2009]

Instructions affected by soft errors are re-executed

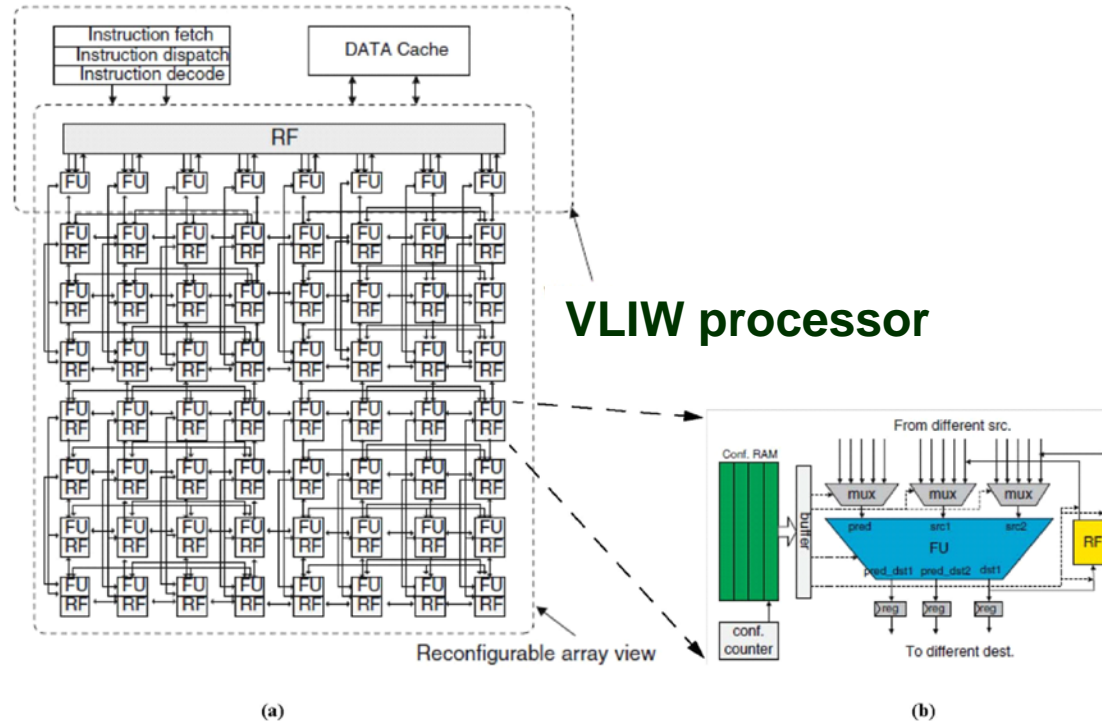
Instruction re-executed if soft error in any pipeline stage

FV of Pipelined Processors with Reconfigurable Functional Units

A method to abstract reconfigurable functional units:



We Formally Verified ADRES Processor with Reconfigurable Array



A Very Long Instruction Word (VLIW) processor, shown at the top, is combined with a coarse-grained reconfigurable array (a), where each reconfigurable functional unit (FU) has its dedicated register file (RF), and configuration memory (Conf. RAM), as shown in (b).

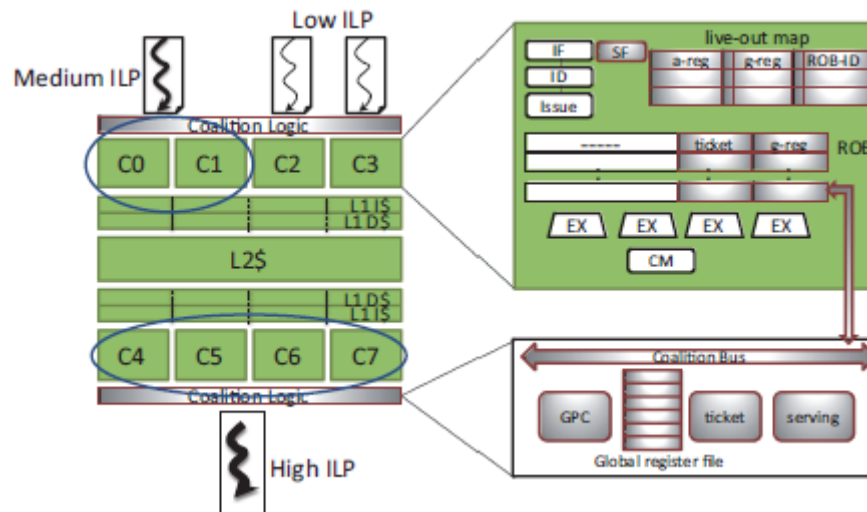
FV of Pipelined Processors with Hardware Support for Multithreading

We developed abstraction techniques that allow us to formally verify pipelined processors with hardware support for ANY number of threads

- **Can scale for GPUs**

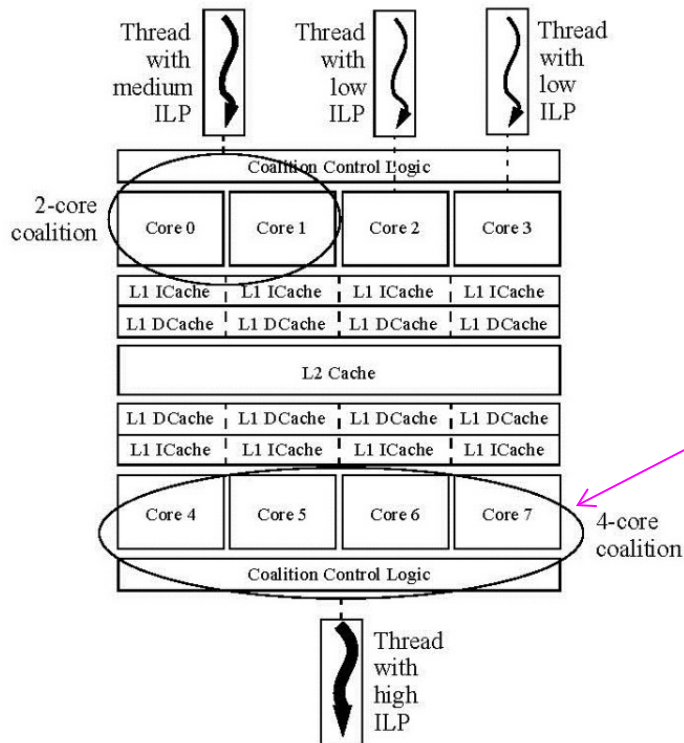
Polymorphic Heterogeneous Multi-Core Architectures

Bahurupi architecture—several simple cores combined with coalition dispatch and completion logic to accelerate execution of 1 thread [Pricopi & Mitra 2012]



Performance comparable or greater than that of wide superscalar design with issue width = sum of issue widths of the cores in a coalition, but lower power consumption, and higher reliability

We Formally Verified Polymorphic Heterogeneous Multi-Core Processor for Space Applications



Our method was showcased in *NASA Tech Briefs* (LEW-19207-1, 2014), which publishes only the best NASA-funded inventions

Large coalitions can accelerate mission-critical threads, e.g., to analyze trajectory of approaching missile and determine how to maneuver a jet fighter to avoid the missile

We can formally verify such multi-core processors completely with our technology

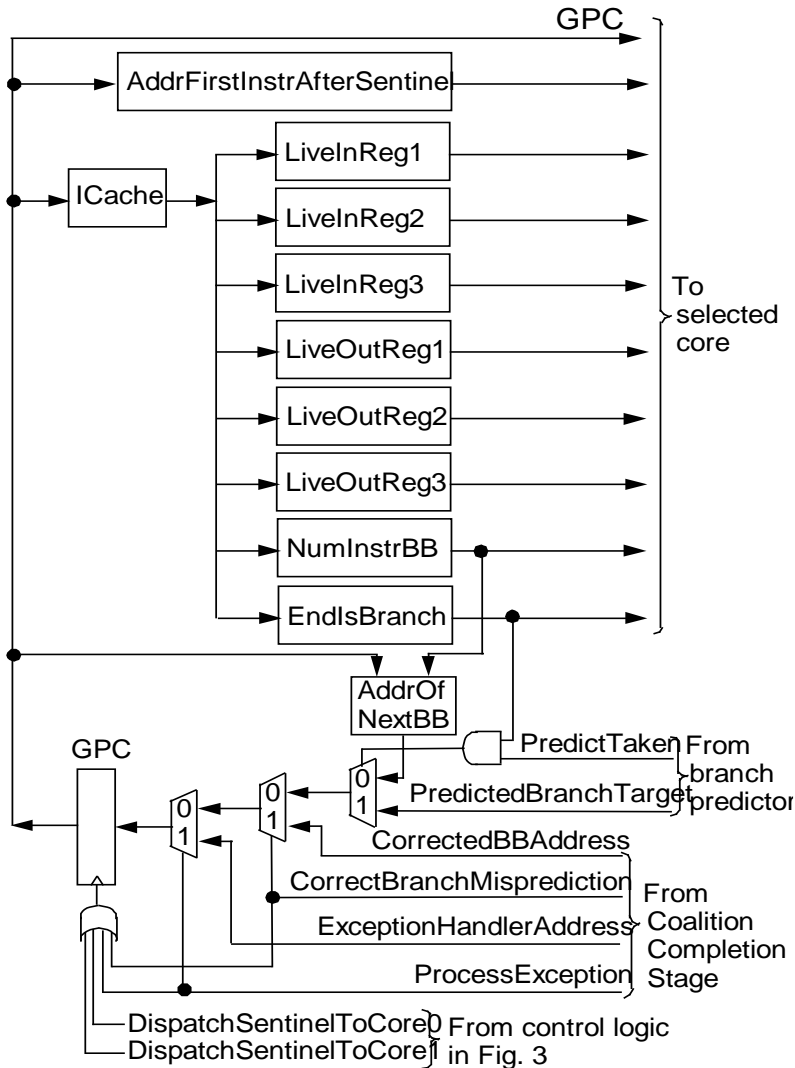
Abstraction of Coalition Dispatch

GPC = General PC

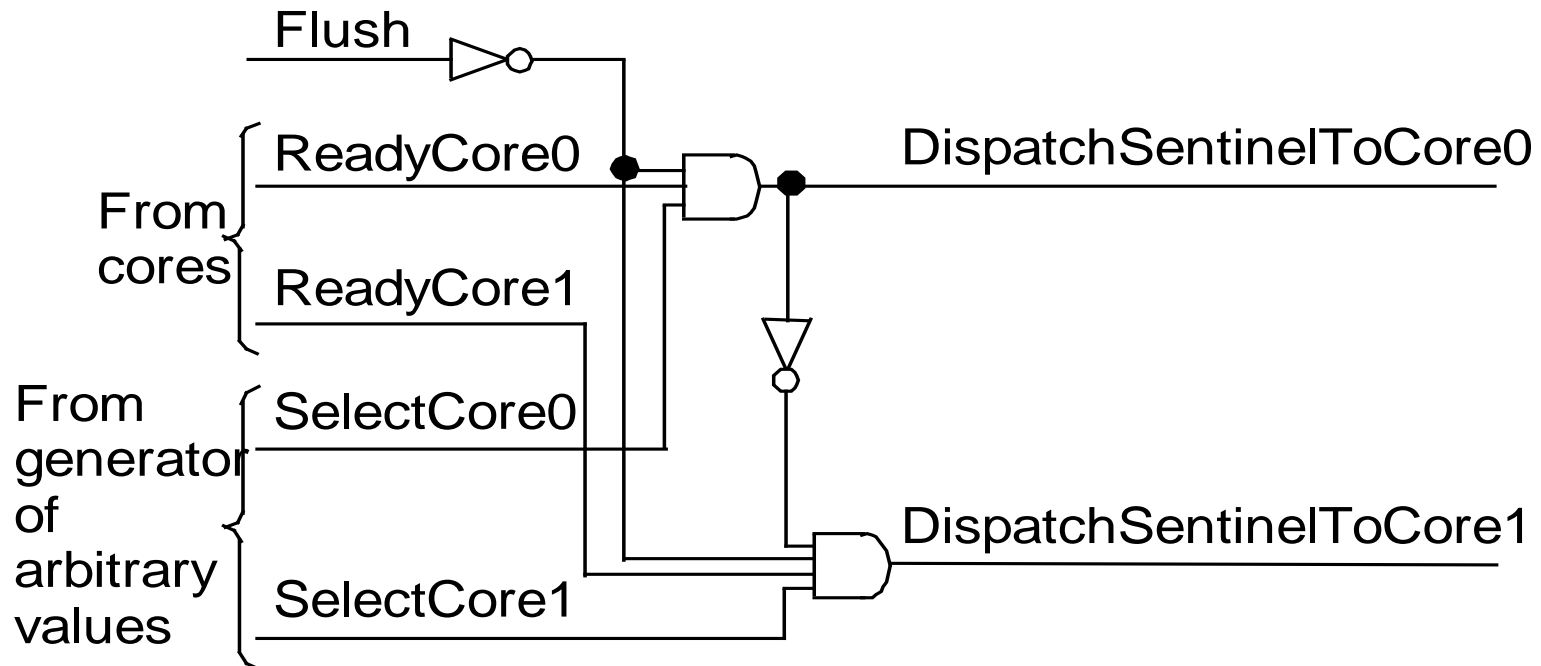
points to next BB to be fetched

initialized with address of first BB

Each BB begins with a sentinel instruction



Abstraction of Control Logic That Selects Which Core to Dispatch to



Flush signal is used to determine controlled flushing

Ticket Register in Dispatch

Ticket Register in Dispatch Stage

- gives unique id to each dispatched BB
- incremented in each cycle when a BB is dispatched to a core
- incrementing it is abstracted with UF NextTicket

Serving Register in Completion Stage

- contains id of next BB to be completed
- Incrementing it also abstracted with UF NextTicket

Abstraction of Each Core

3 FSMs, each abstracting the execution of a BB

At most two FSMs have valid BBs initially

**The FSM with no BB can accept a new BB
non-deterministically**

An FSM with valid BB:

**can compute its results non-deterministically in a
cycle of regular symbolic simulation, as long as all
input operands are available**

**computes the results of such a basic block in every
clock cycle during flushing**

Computations abstracted with UFs and UPs

Modeling of Coalition Completion Stage

Completes an entire BB per clock cycle

if the BB's results are computed

and BB's ticket equals the current value of the Serving register

If the completed BB ends on a branch then

the condition for a branch misprediction is formed based on the branch prediction made for that BB in the Coalition Dispatch Stage

**Serving Register updated with term produced by UF
NextTicket applied to current term for value of
Serving Register**

Required Invariants (1 of 2)

- 1) if there are k valid BBs in the cores, then the term abstracting the current value of the Ticket Register equals k applications of UF NextTicket to the term abstracting the current value of the Serving Register;
- 2) if a BB in a core is valid, then the term abstracting its ticket equals either the term for the current value of the Serving register, or up to $k - 1$ applications of UF NextTicket to the term for the current value of the Serving register, where k is the number of valid BBs in the cores;
- 3) if a BB in a core is valid, then the term abstracting its ticket does not equal the term abstracting the ticket of another valid BB in a core, or the current value of the Ticket Register;

Required Invariants (2 of 2)

- 4) if a BB in a core is valid, then each of its live-in registers either has its data value available, or the renaming tag of that live-in register equals the renaming tag of a live-out register whose data value is not computed yet, and that belongs to a valid BB that is in a core and has a ticket that is ahead of the ticket of the given BB, i.e., the term abstracting the ticket of the given BB is equal to one or more applications of UF NextTicket to the term abstracting the ticket of the BB that will compute the data value;
- 5) if a valid BB in a core is ready for completion, then the data values of its live-out registers, exception condition, as well as branch direction and target if the BB ends on a branch, have been computed.

Non-Pipelined Specification

Defined to fetch, execute, and complete one BB per clock cycle

Uses the same UFs and UPs to compute the results of instructions as the abstractions of the cores

No branch prediction & no register renaming

Setup & Tool Flow

Workstation with two 3.47-GHz six-core Intel Xeon x5690 processors, and 64 GB of memory, running Red Hat Enterprise Linux v6.4.

(Only a single core was used.)

Tool flow:

- **HighCheck**

Results

Proving safety of model with 2 cores: < 1 sec

Proving safety of model with 4 cores: < 3 sec

Note: these times include the checking of the invariants

Outline

- ✓ **EUFM Background: Positive Equality**
- ✓ **Applications to Formally Verify Different Architectures**

Conclusion

Conclusion (1 of 2)

We presented abstraction techniques that allow us to exploit the property of Positive Equality to formally verify a wide range of processor architectures very efficiently

These techniques:

- **outperform other approaches for formal verification of microprocessors by orders of magnitude**
- **require minimal manual intervention**

Conclusion (2 of 2)

Our tool flow scales for formally verifying correctness of safety and liveness of complex pipelined/superscalar/VLIW processors with:

- branch prediction
- exceptions
- multicycle functional units
- advanced and speculative loads
- predicated execution
- register remapping
- out-of-order execution based on a reorder buffer
- delayed branches
- data-value prediction
- mechanisms to correct soft errors by re-executing affected instructions
- reconfigurable functional units
- arrays of reconfigurable processing elements
- multi-threaded execution
- reconfigurable polymorphic heterogeneous multi-core architectures

CNF formulas generated in this work 20 years ago have been used in the development of all academic and industrial SAT solvers since then.