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 (miroslav.velev@aries-da.com)

ARIES DESIGN AUTOMATION®

Keynote at NetWare'21 - CENICS'21

an IARIA conference

Speaker Bio

Miroslav N. Velev received B.S.&M.S. in Electrical Engineering, and B.S. in Economics from Yale University in 1994, and Ph.D. in Electrical and Computer Engineering from Carnegie Mellon University in 2004. In 2005 he started Aries Design Automation (Chicago, USA), where he is President and leads R&D on formal verification, AI, and other topics.

Distinctions:

- Fellow, American Association for the Advancement of Science (AAAS), 2017;
- Associate Fellow, American Institute of Aeronautics and Astronautics (AIAA), 2017;
- Distinguished Member (Scientist) of ACM, 2014;

Awards:

- IEEE Aerospace and Electronic Systems Society (AESS) Industrial Innovation Award, 2021;
- EDAA Outstanding Dissertation Award, 2005;
- Franz Tuteur Memorial Prize for the Most Outstanding Senior Project in Electrical Engineering, Yale University, 1994.

Motivation

Formal Verification (FV) is mathematically based proof of correctness of computer systems; if it scales, FV is exhaustive

FV is critical, e.g. Boeing 737 Max crisis: 346 people dead, >$18.6B loss for Boeing, >$6B for airlines

Cost of microprocessor bugs in weapon systems can be greater, including compromised national security

Outline

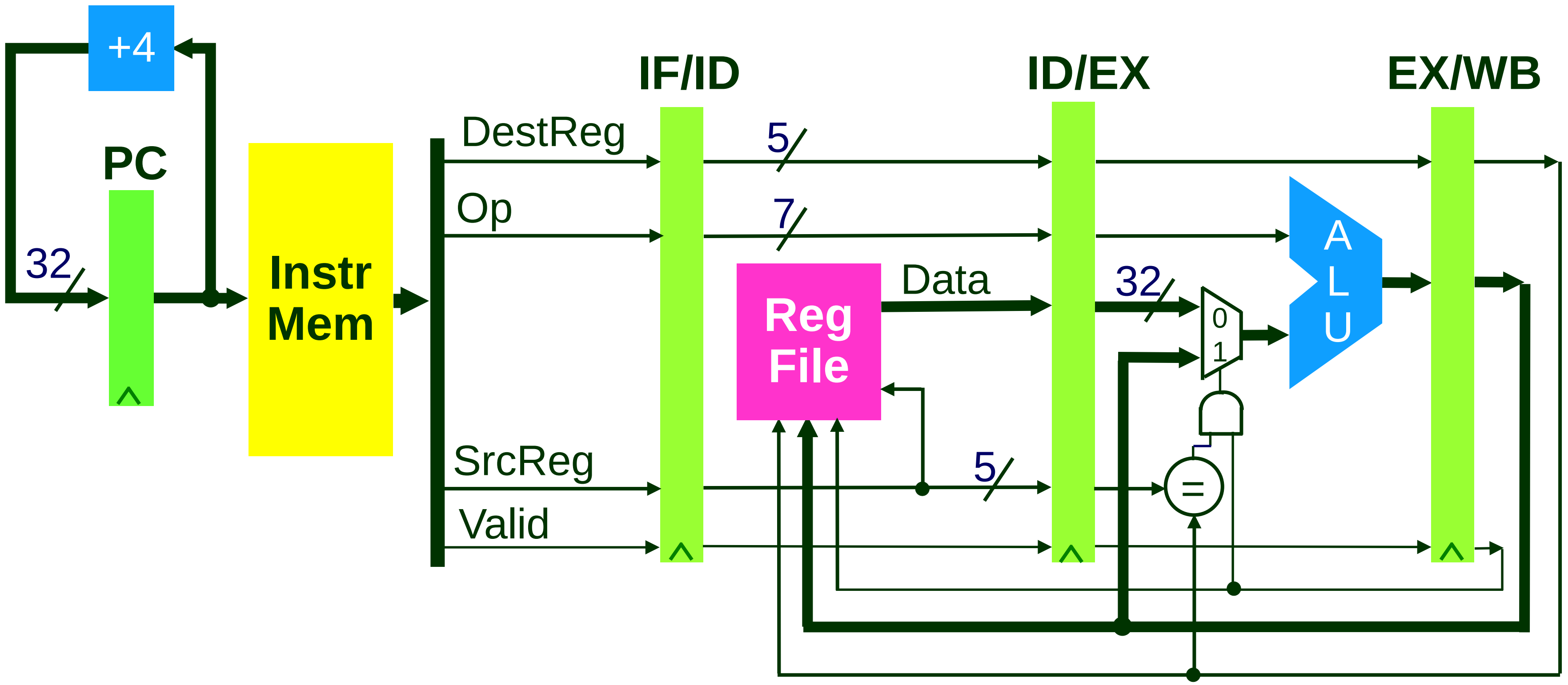
EUFM Background: Positive Equality

Block-Level Translation to SAT

Applications to Formally Verify Different Architectures

Conclusion

Gate-Level Microprocessor



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

Formal verification complexity is exponential

- Details in [Velev & Bryant, *FMCAD '98*]

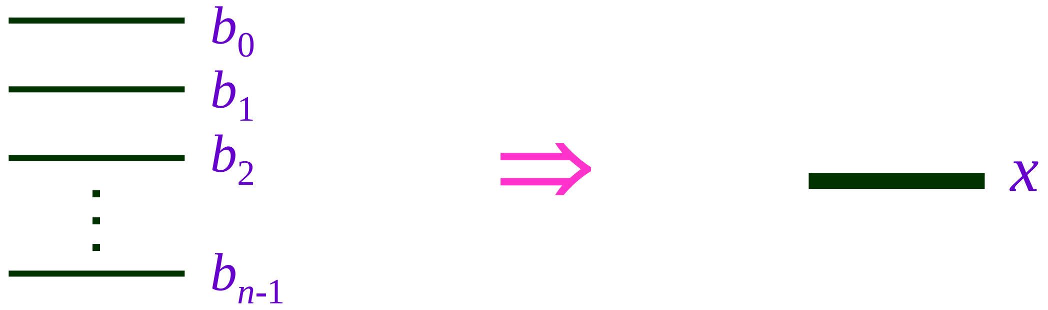
Two-Step Formal Methodology

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**

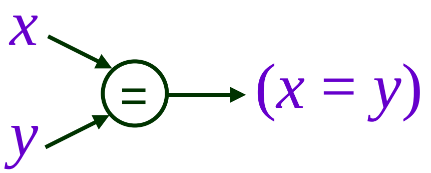
Abstracting Data

Terms abstract data values

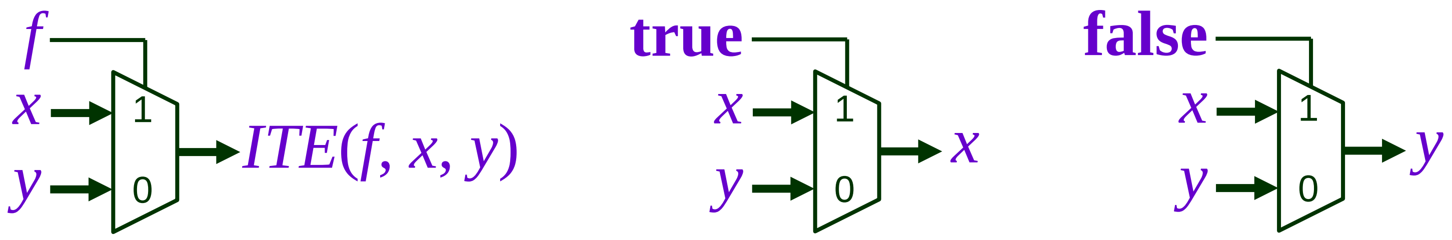


Properties:

- Equality comparison:



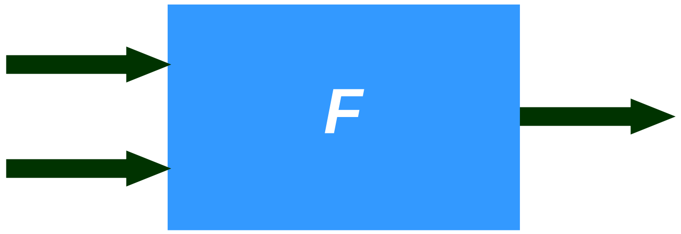
- Can be stored in memories
- Can be selected with *ITE* operators:



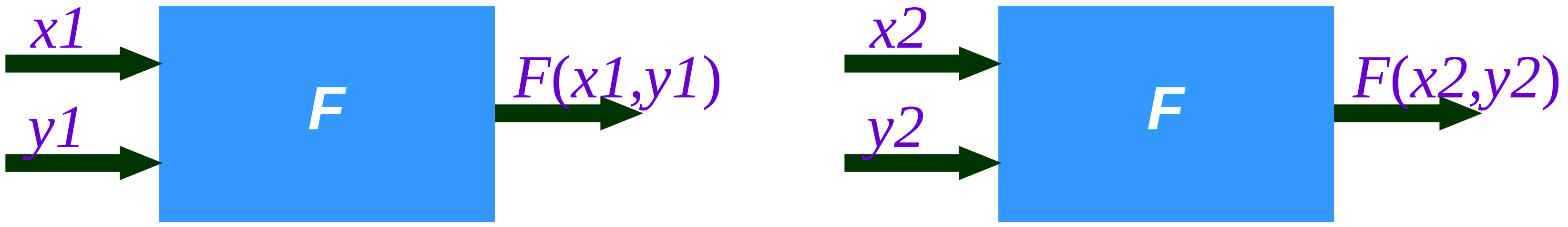
Abstracting ALUs

Uninterpreted Functions abstract computations

- internal implementation details removed



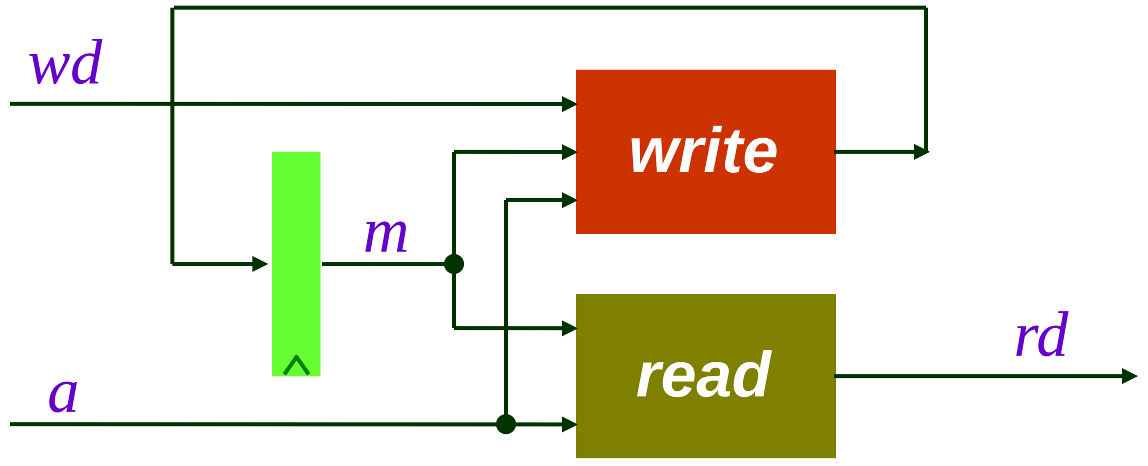
- functional consistency:



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

Abstracting Memories

FSM model:

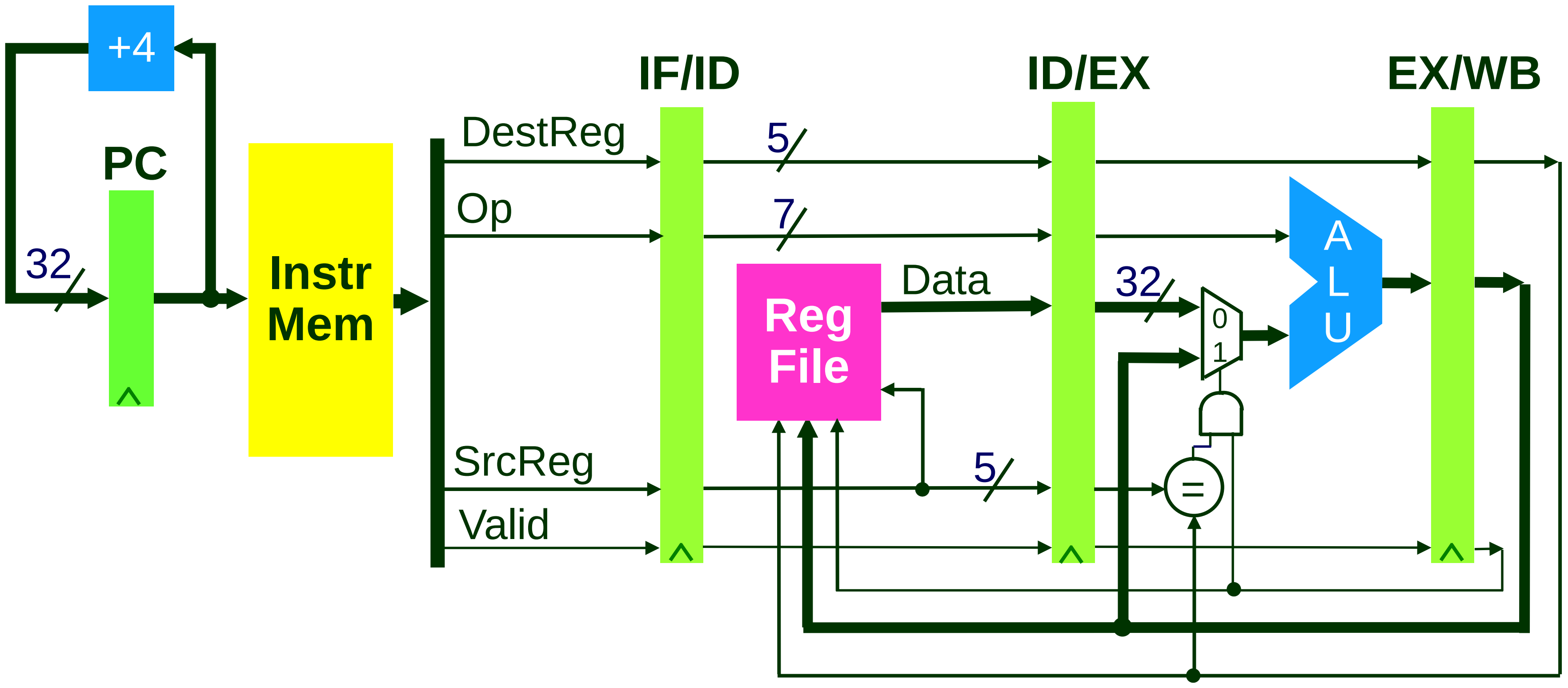


Functions *write* and *read* abstract memory operations

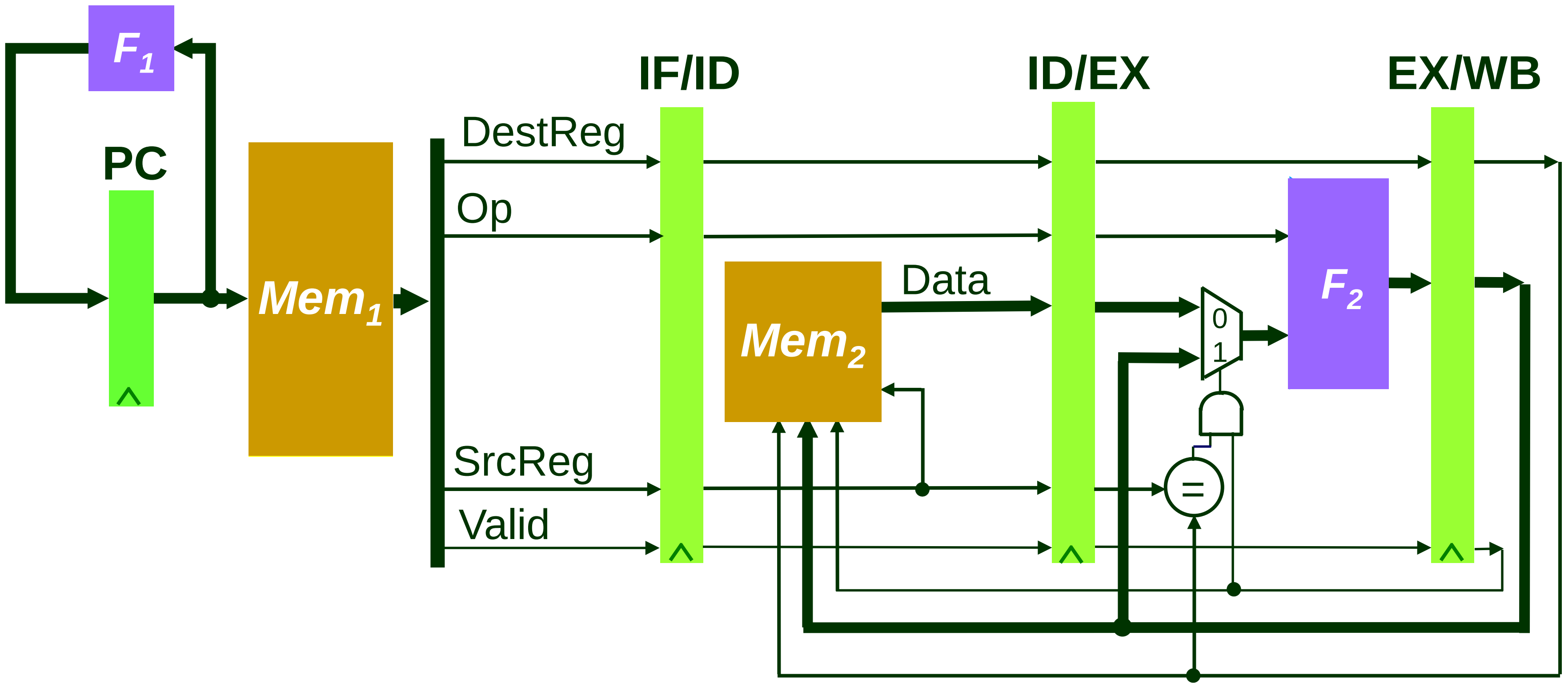
Forwarding property:

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

Application of Abstractions



Application of Abstractions

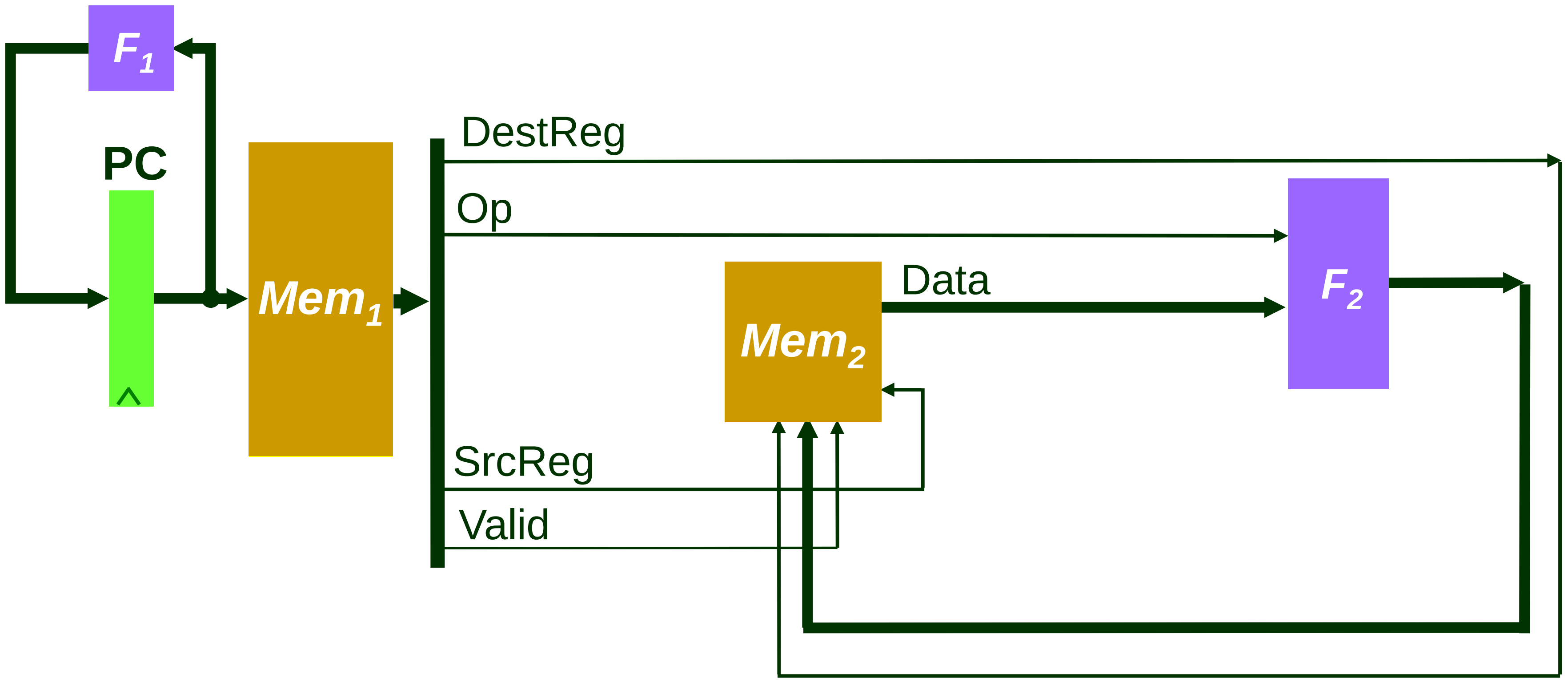


⇒ More general processor

- easier to prove correct

Functional units & memories formally verified separately

Specification Processor



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

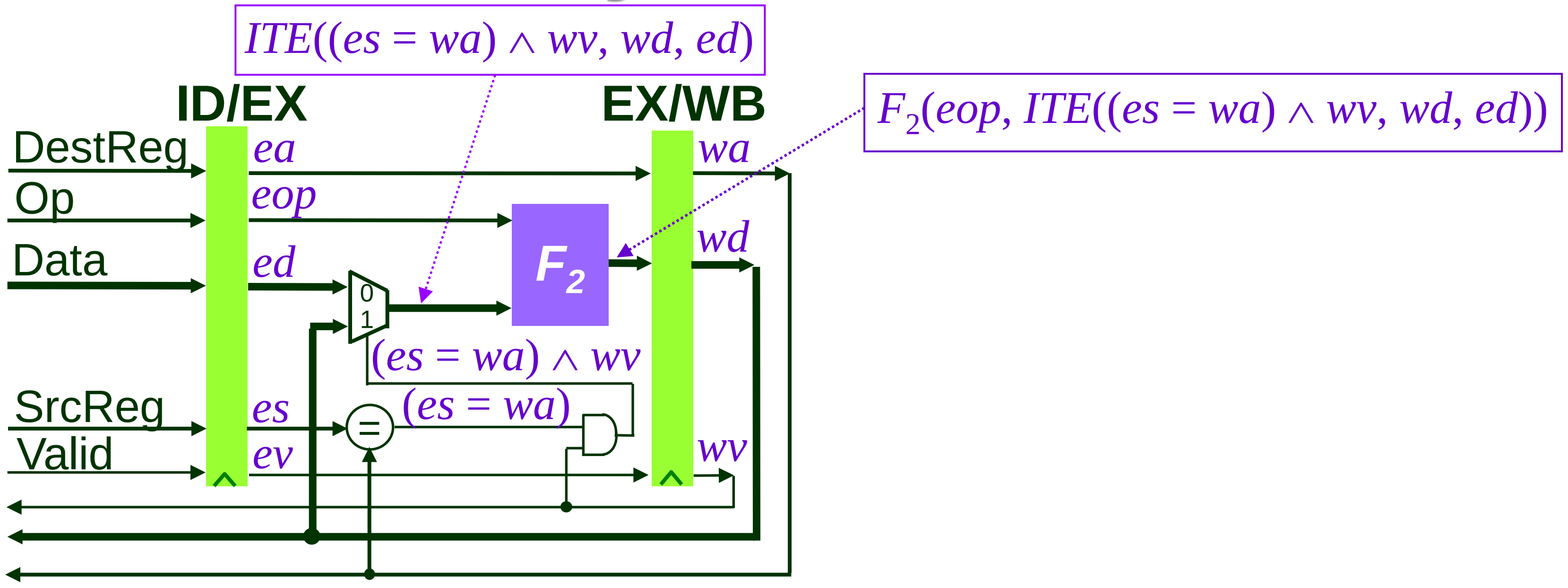
Safety Correctness Criterion

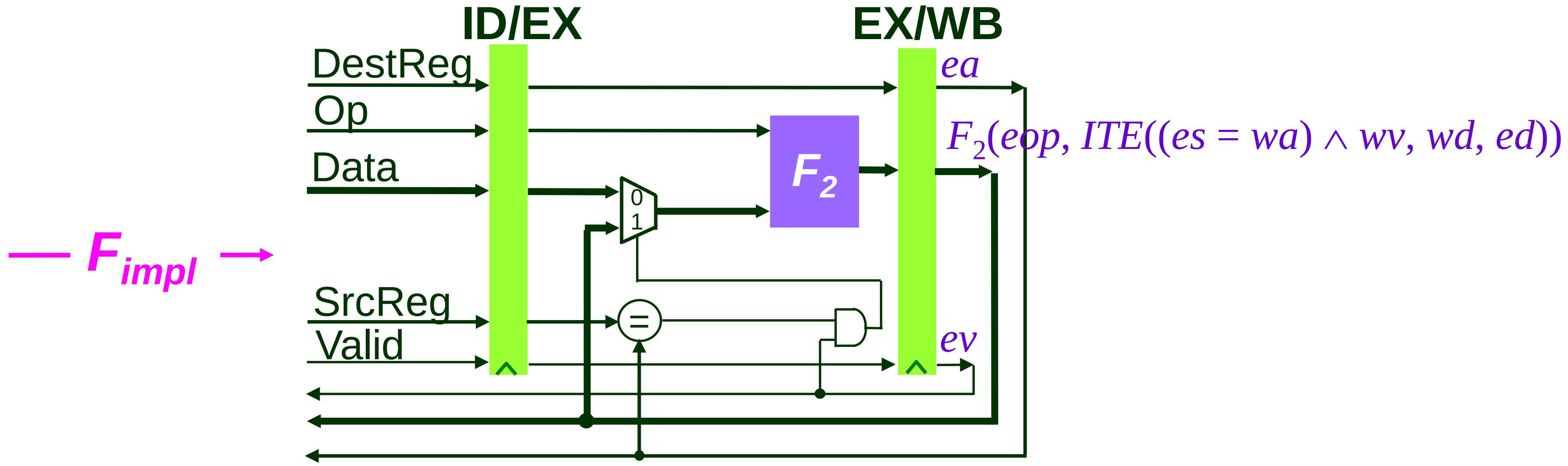
$$\begin{tikzcd}
Q^0_{spec} \arrow[rr, "F^k_{spec}"] & & Q^1_{spec} \\
Q^0_{impl} \arrow[u, Rightarrow, "Flush"] \arrow[rr, "F_{impl}"] & & Q^1_{impl} \arrow[u, Rightarrow, "Flush"]
\end{tikzcd}$$

Term-level symbolic simulation
of Implementation for 1 clock cycle

symbolic initial state
(represents
ANY initial state)

Term-Level Symbolic Simulation





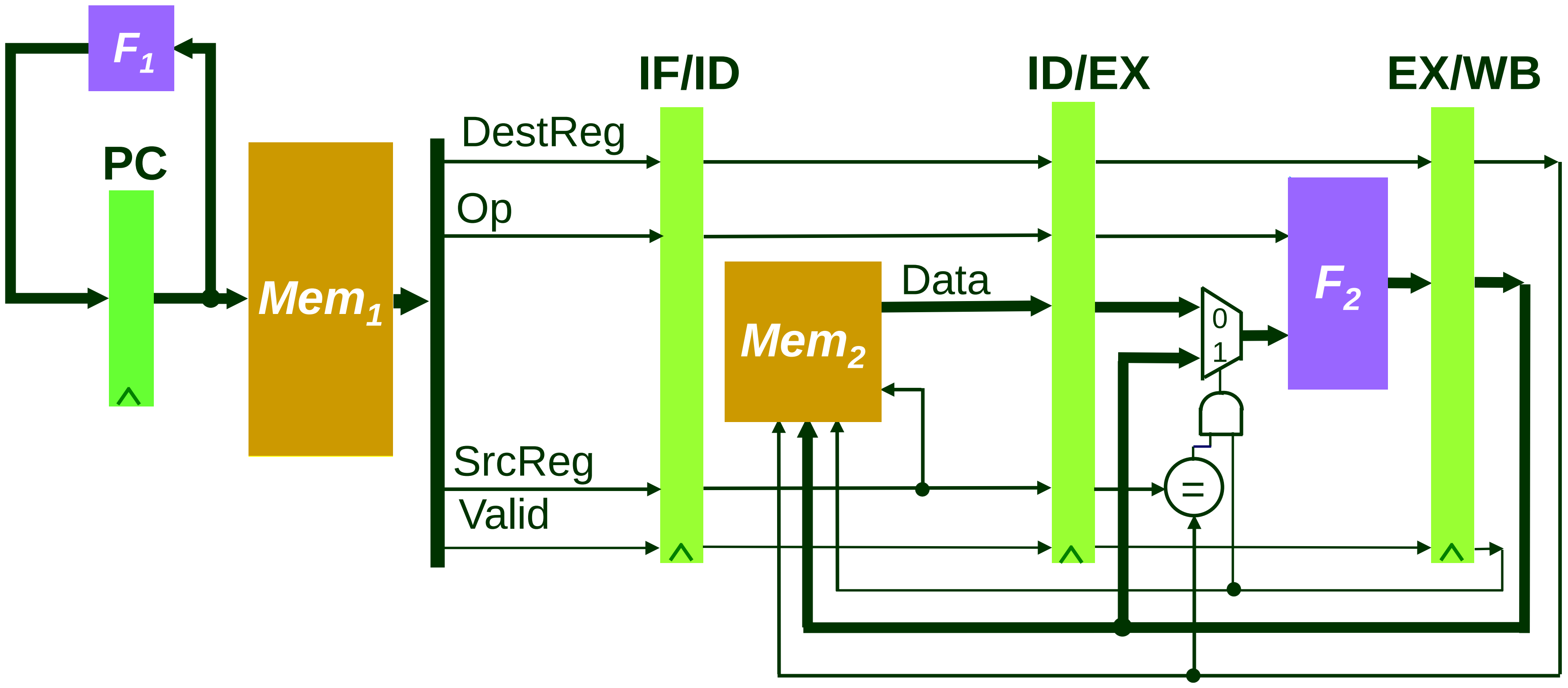
Safety Correctness Criterion

$$\begin{tikzcd}
Q^0_{spec} \arrow[rr, "F^k_{spec}"] & & Q^1_{spec} \\
Q^0_{impl} \arrow[u, Rightarrow, "Flush"] \arrow[rr, "F_{impl}"] & & Q^1_{impl} \arrow[u, Rightarrow, "Flush"']
\end{tikzcd}$$

***Flush*, Burch & Dill [*CAV '94*]**

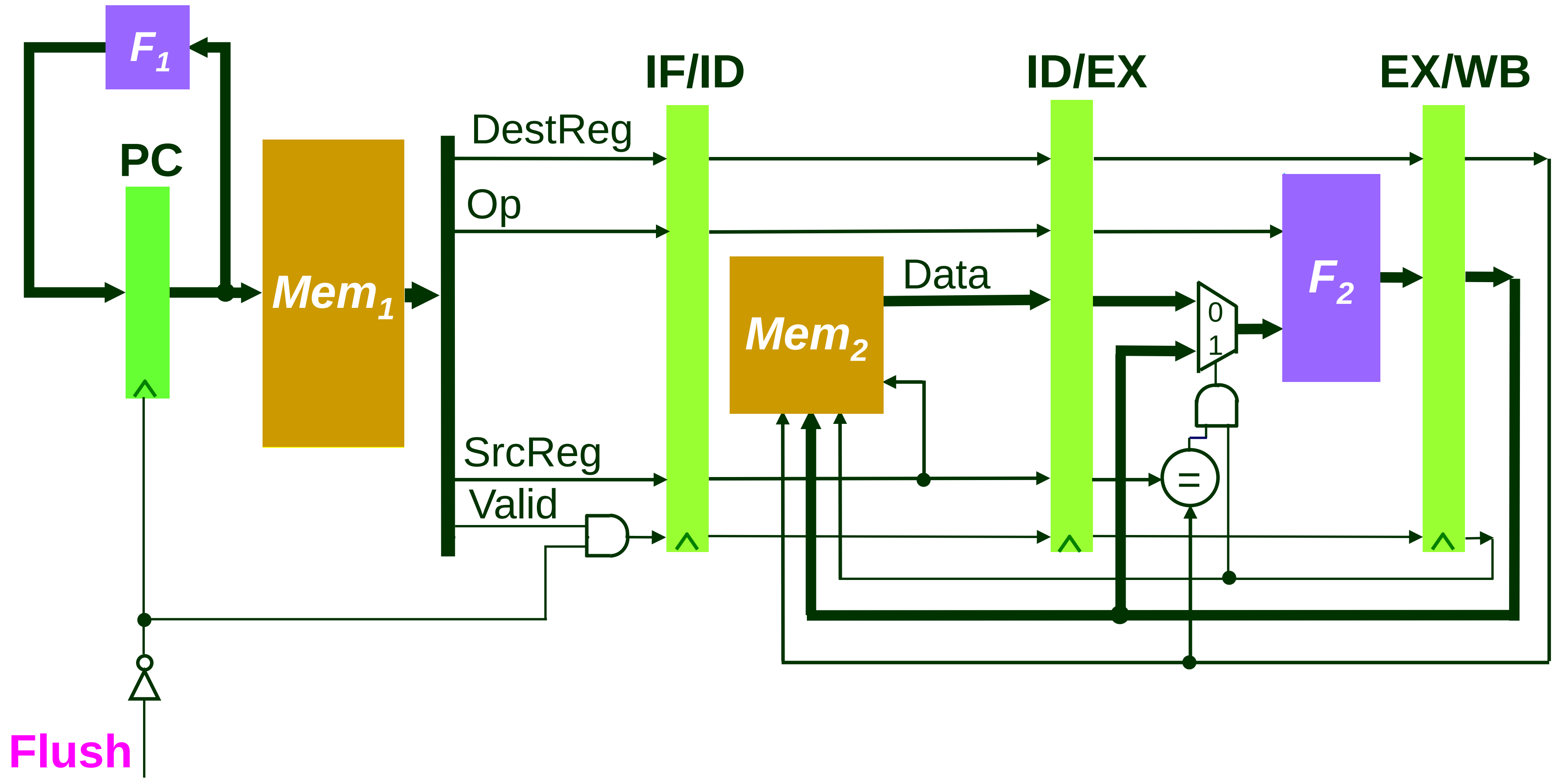
- **automatically maps state of pipeline to user-visible state**
- **completes partially-executed instructions**

Flushing



F_1
PC
Mem_1
IF/ID
ID/EX
EX/WB
DestReg
Op
Mem_2
Data
SrcReg
Valid
0
1
=
F_2

Flushing



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

Safety Correctness Criterion

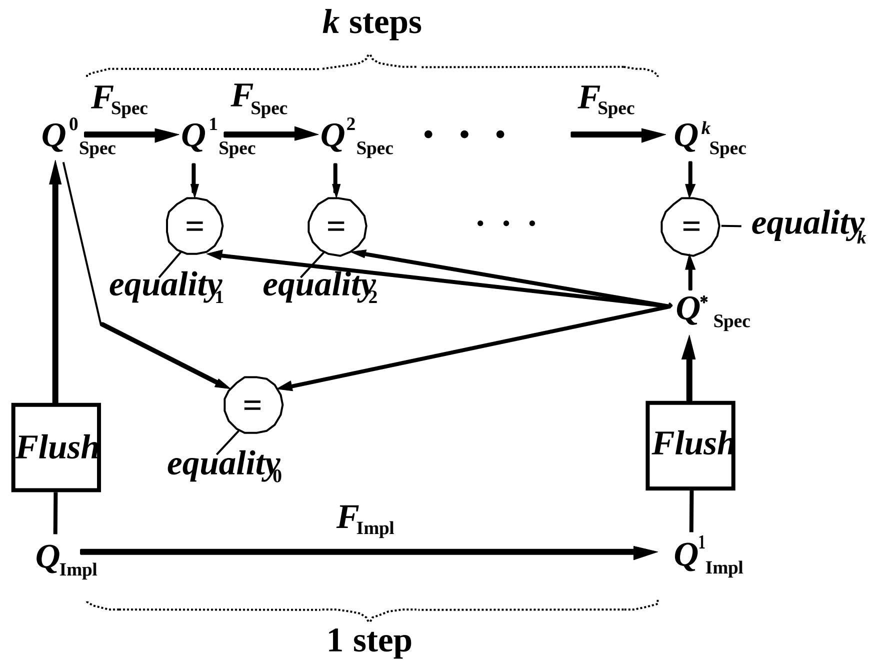
$$\begin{tikzcd}
Q^0_{spec} \arrow[r, "F^k_{spec}"] & Q^1_{spec} \\
Q^0_{impl} \arrow[u, Rightarrow, "Flush"] \arrow[r, "F_{impl}"] & Q^1_{impl} \arrow[u, Rightarrow, "Flush"']
\end{tikzcd}$$

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: $equality_0 \vee equality_1 \vee \ldots \vee equality_k = true$



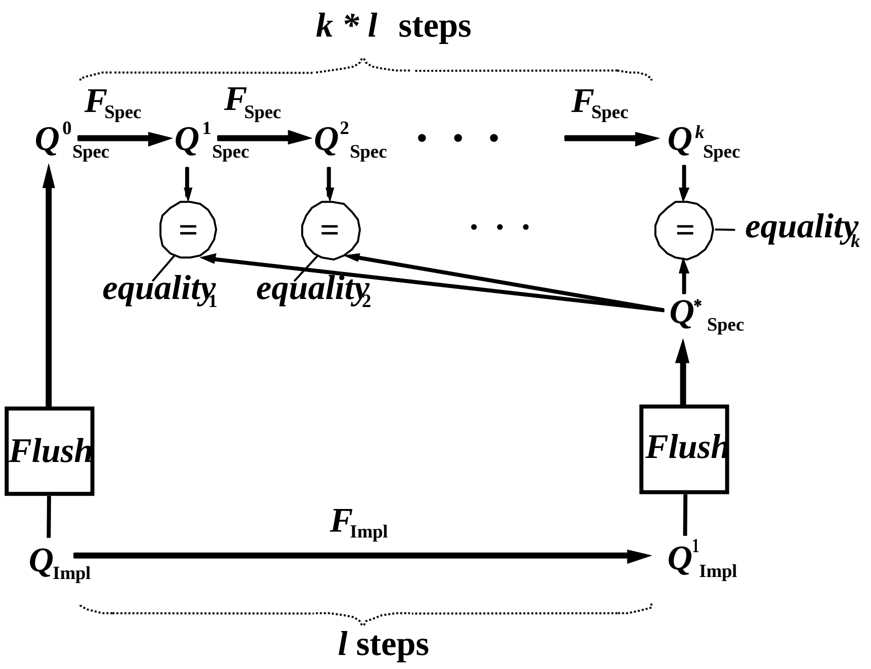
i.e., a proof that 1 step of the Implementation corresponds to between 0 and *k* steps of the Specification, where *k* is the issue width of the Implementation

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

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

Liveness Correctness Criterion

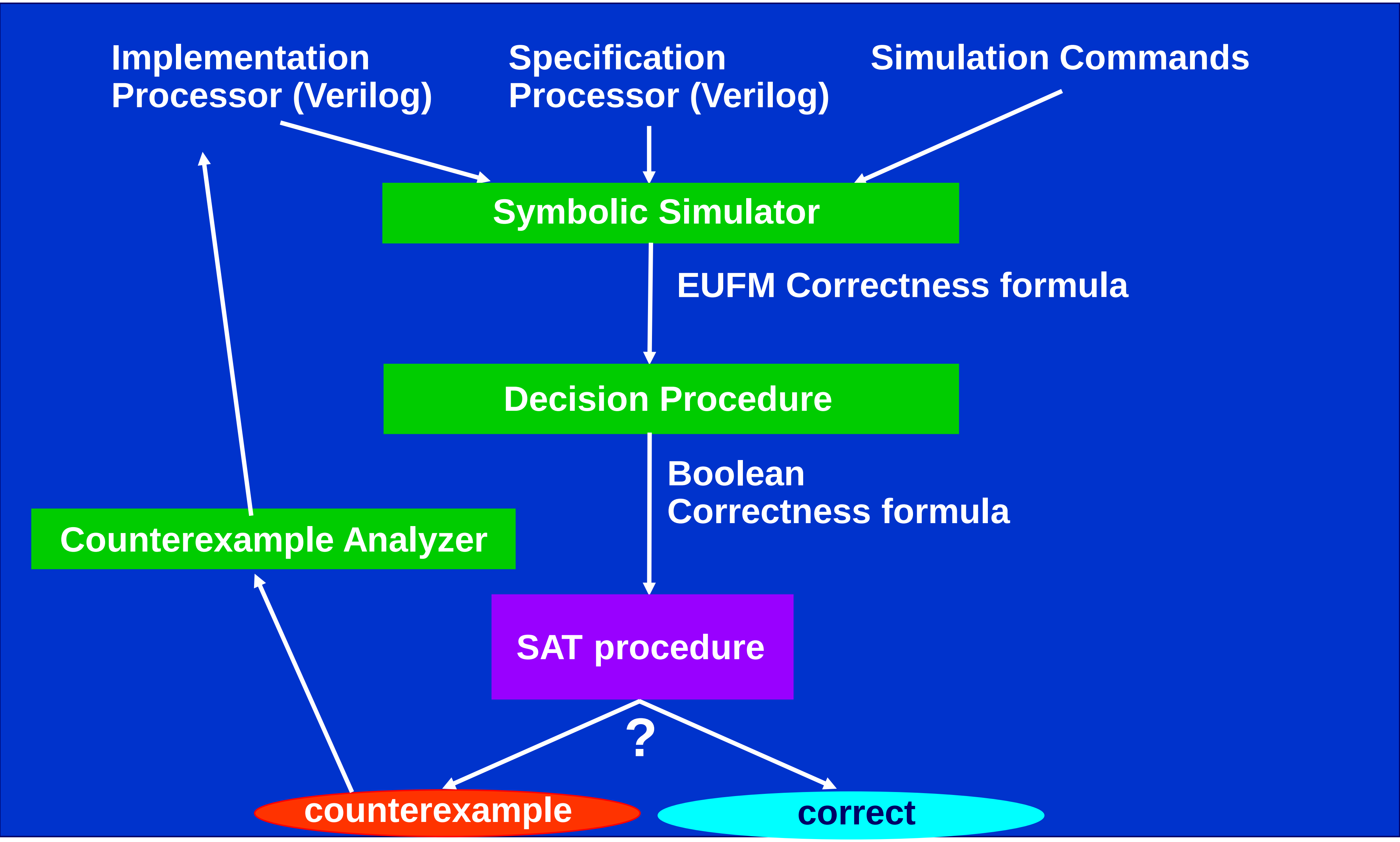
In the general case: $equality_1 \vee \ldots \vee equality_{k*l} = true$



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]

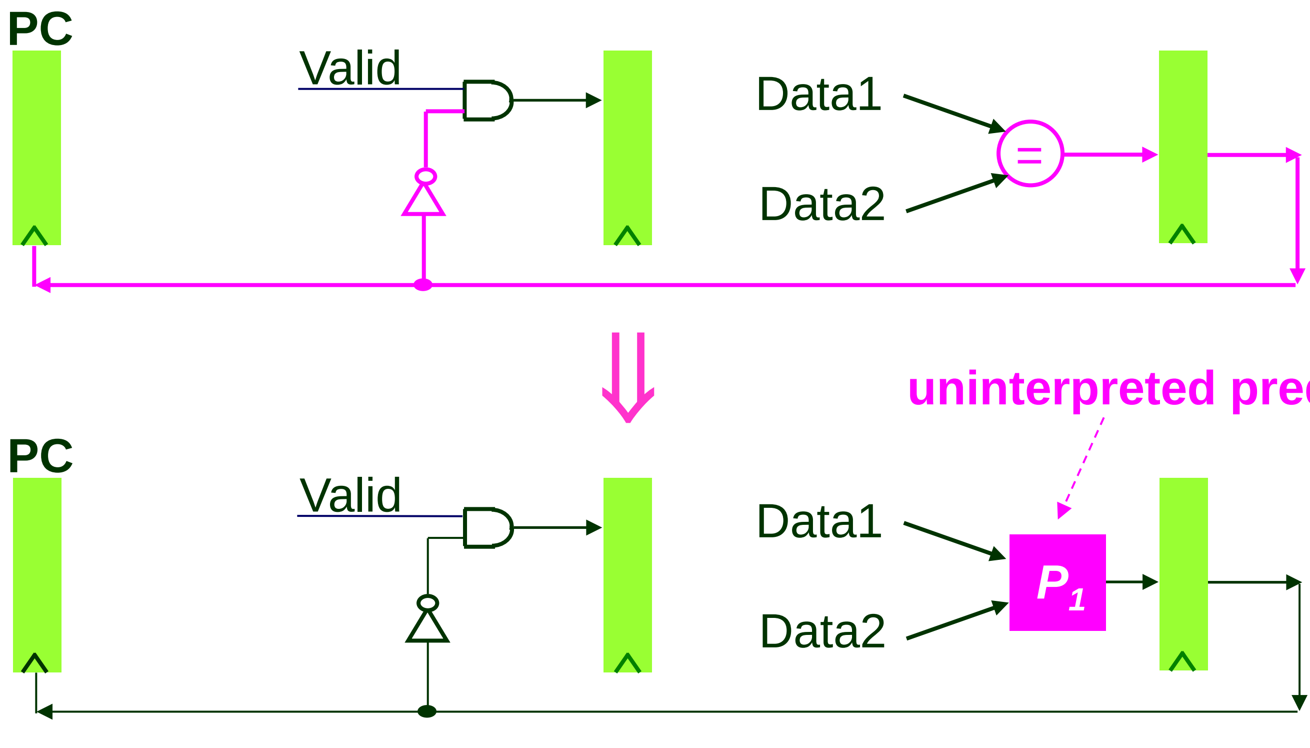
Our Tool: HighCheck



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

$m2 \leftarrow \boxed{F_1}(m1, a1, wd)$

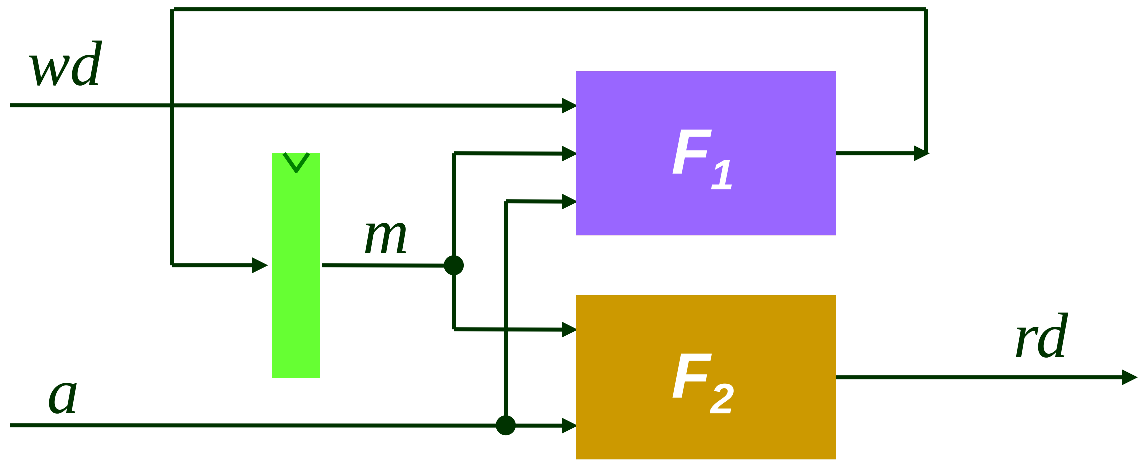
$rd \leftarrow \boxed{F_2}(m2, a2)$

Forwarding property NOT enforced

$rd = ITE(a2 = a1, wd, read(m1, a2))$

Conservative approximation of memory

FSM model:



Positive Equality

By imposing some simple restrictions on the processor modeling style, we obtain a special structure of the correctness formula, where:

- 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
- Block-Level Translation to SAT
- Applications to Formally Verify Different Architectures
- Conclusion

Motivation for Previous Efficient Translation to CNF

CNF-based SAT-solvers face 2 main hurdles:

- Boolean Constraint Propagation (BCP)
  - Up to 90% of the SAT time
- Many cache misses for big formulas

Conventional CNF translation [Tseitin 68]:

- Variable for every signal
- Set of clauses for every logic gate

We can speed up SAT solving by merging adjacent logic gates and representing them with a unified set of clauses without variables for intermediate signals

Example: AND→ITE group

$o \leftarrow \mathrm{ITE}(i, t, e)$

$t \leftarrow \mathrm{AND}(a1, \ldots, an)$ $\quad$ $fanout\_count(t) = 1$

equivalent constraints in the new translation:

$i \wedge \neg a1 \Rightarrow \neg o$

. . .

$i \wedge \neg an \Rightarrow \neg o$

$i \wedge a1 \wedge \ldots \wedge an \Rightarrow o$

$\neg i \wedge e \Rightarrow o$

$\neg i \wedge \neg e \Rightarrow \neg o$

This Translation Was Implemented for Following Gate Groups

ITE-Chains: the else-input is another ITE

ITE-trees

AND→ITE (AND is input to ITE)

OR→ITE

OR→AND (use FANIN heuristic to pick input)

ITE→AND

AND→OR

ITE→OR

Producing More ITE-Trees

We can preserve the ITE-tree structure of equation arguments when eliminating equations (T1 = T2),

Example: $ITE(c1, a1, a2) = ITE(c2, b1, b2)$

Before eliminated by pushing equation to its argument leaves until each argument is a variable:

$$c1 \wedge c2 \wedge (a1 = b1) \quad \vee \quad c1 \wedge \neg c2 \wedge (a1 = b2)$$
$$\vee \quad \neg c1 \wedge c2 \wedge \quad (a2 = b1) \quad \vee \quad \neg c1 \wedge \neg c2 \wedge (a2 = b2)$$

Now:

$$ITE(c1,\ ITE(c2, a1 = b1, a1 = b2),$$
$$ITE(c2, a2 = b1, a2 = b2))$$

Can further merge ITE-trees with 1 or more levels of AND or OR leaves

Results from This Translation

Up to 420× speedup on unsatisfiable CNF formulas with

- 100,000s of variables
- 1,000,000s of clauses
- 10,000,000s of literals

Best impact from preserving the ITE-tree structure of equation arguments and merging ITE-trees

Details in [Velev, *ASP-DAC'04*], [Velev, *DATE'04*]

Benefits from Merging ITE-trees

1) Reduced variables and clauses
2) Reduced solution space
3) Reduced BCP
4) Automatic use of signal unobservability
   - as soon as an ITE controlling signal selects a branch, then all clauses for other branches in the ITE-tree become satisfied
5) Reduced L2-cache misses
6) Guiding the SAT-solver branching
7) Higher ranking of variables controlling ITEs at the top of ITE-trees
8) Faster solving of case-splitting conditions

Outline

- ✓ EUFM Background: Positive Equality
- ✓ Block-Level Translation to SAT
- Applications to Formally Verify Different Architectures
- Conclusion

Results

Our tool flow scales for formally verifying correctness of:

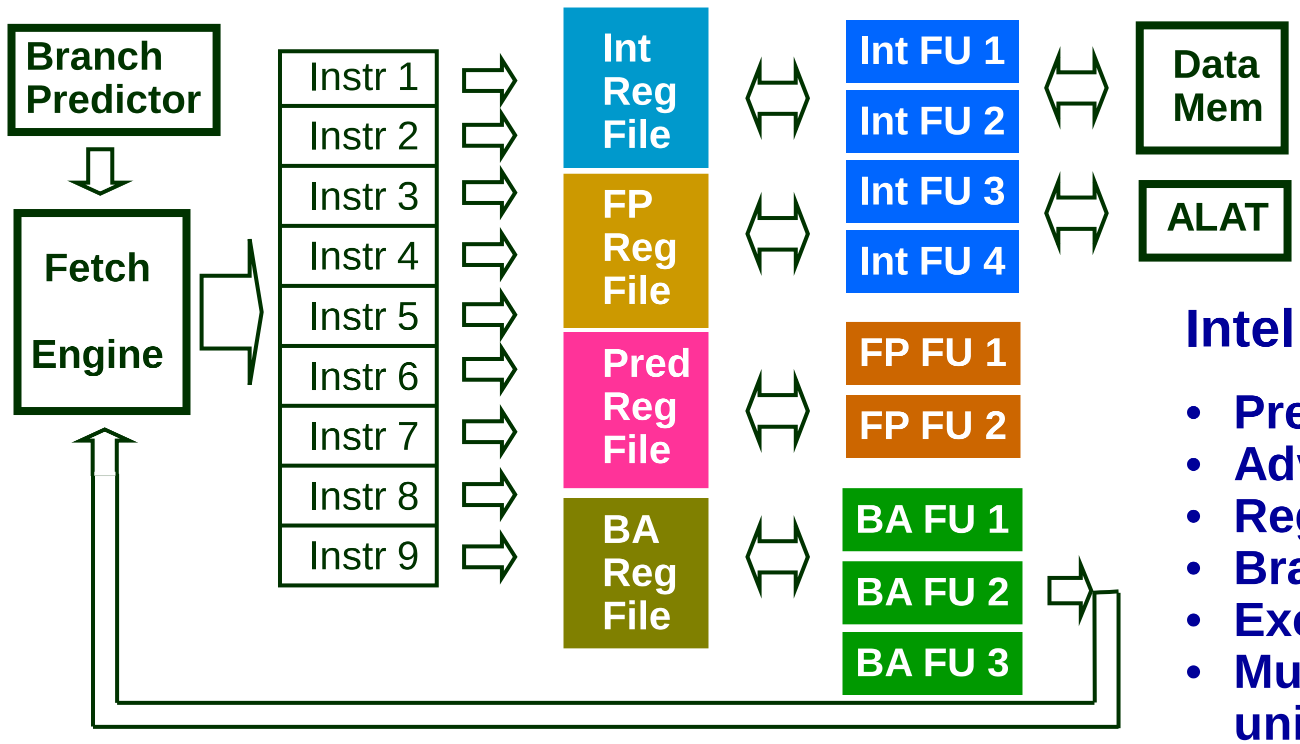
- **complex pipelined/superscalar/VLIW processors with many features:**
  - branch prediction
  - exceptions
  - multicycle functional units
  - advanced and speculative loads
  - predicated execution
  - register remapping
  - out-of-order execution based on a reorder buffer
  - delayed branches
  - 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.**

FV of Complex Dual-Issue Superscalar Processors

Exploiting Positive Equality to formally verify complex dual-issue superscalar processors

- Two 5-stage DLX pipelines
- Exceptions, multi-cycle functional units, and branch prediction were modeled in each pipeline, such that the instructions in the two pipelines interact
- 1 sec of CPU time to formally verify
- Speedup: at least 5 orders of magnitude relative to not using Positive Equality
- Details in [Velev & Bryant, IJES 2005]

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



Intel Itanium® features:

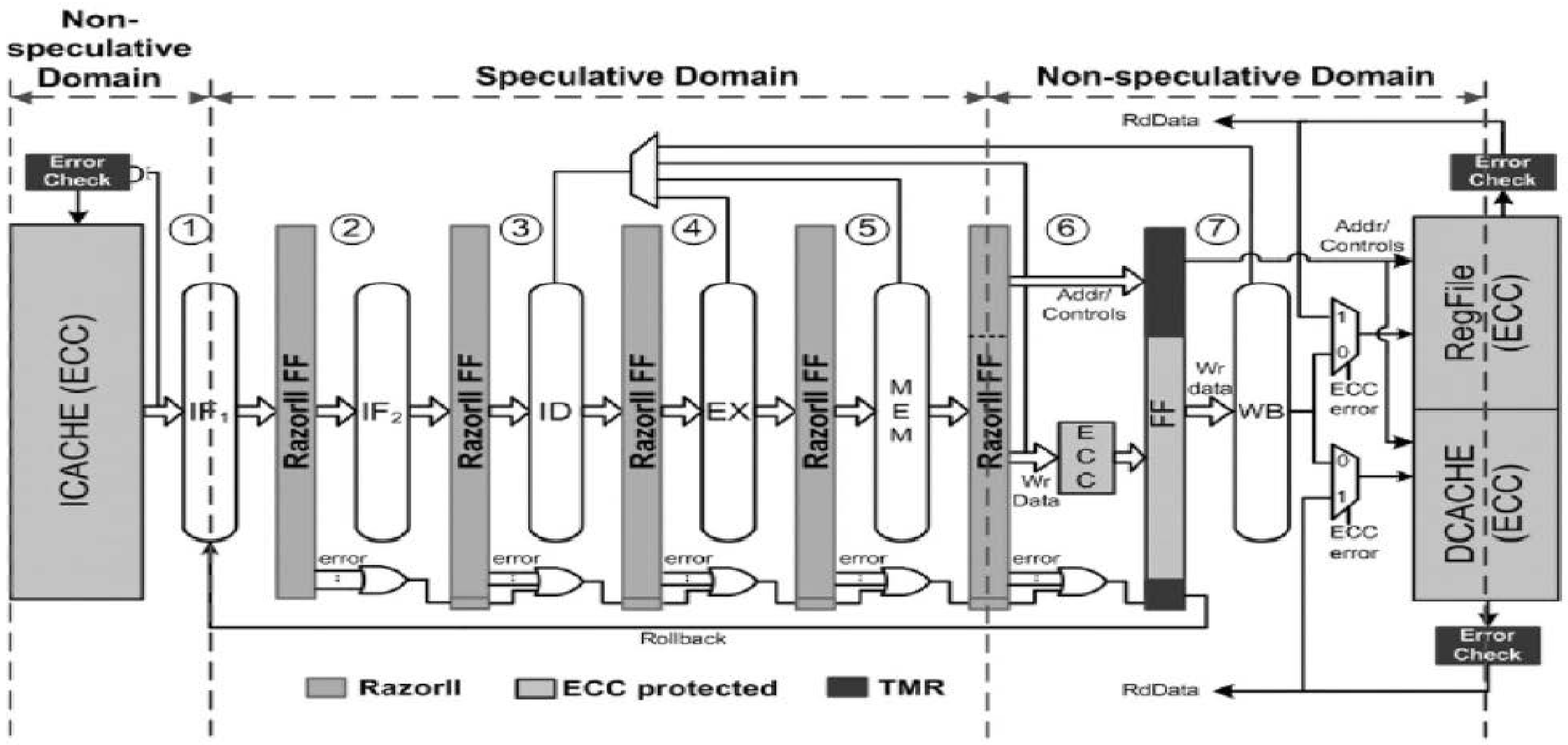
- **Predicated execution**
- **Advanced loads**
- **Register remapping**
- **Branch prediction**
- **Exceptions**
- **Multicycle functional units**

42 VLIW instructions
9 pipeline stages
4 VLIW-instruction queue

13 minutes to formally verify on 1 CPU core

Details in [Velev, *CAV'00*, *ASP-DAC'04*, *DATE'04*]

FV of Pipelined Processors That Detect & Correct Soft Errors



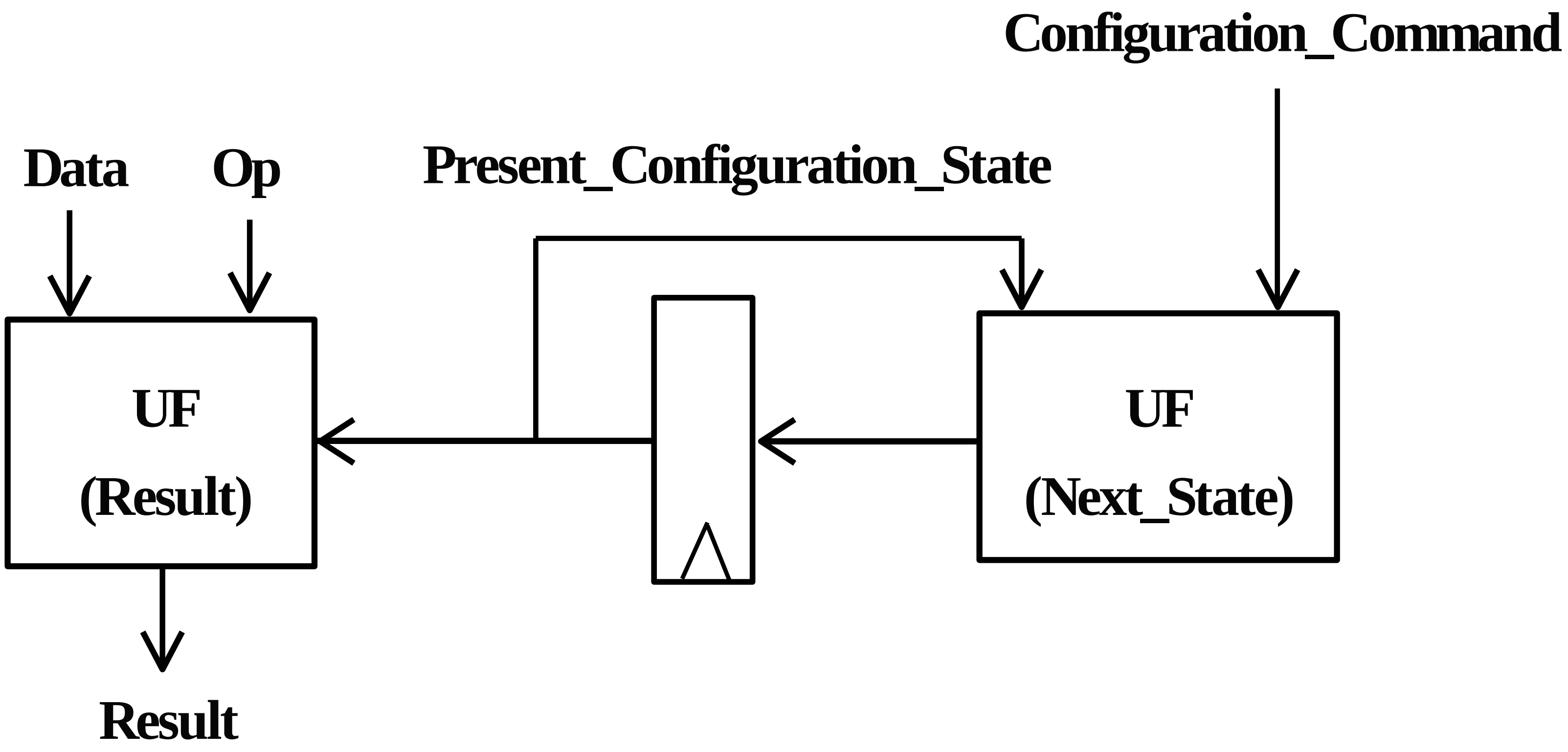
RazorII fault-detecting flip-flops [Das et al. 2009]

Instruction re-executed if soft error in any pipeline stage

Details in [Velev & Gao, *ICFEM'10*]

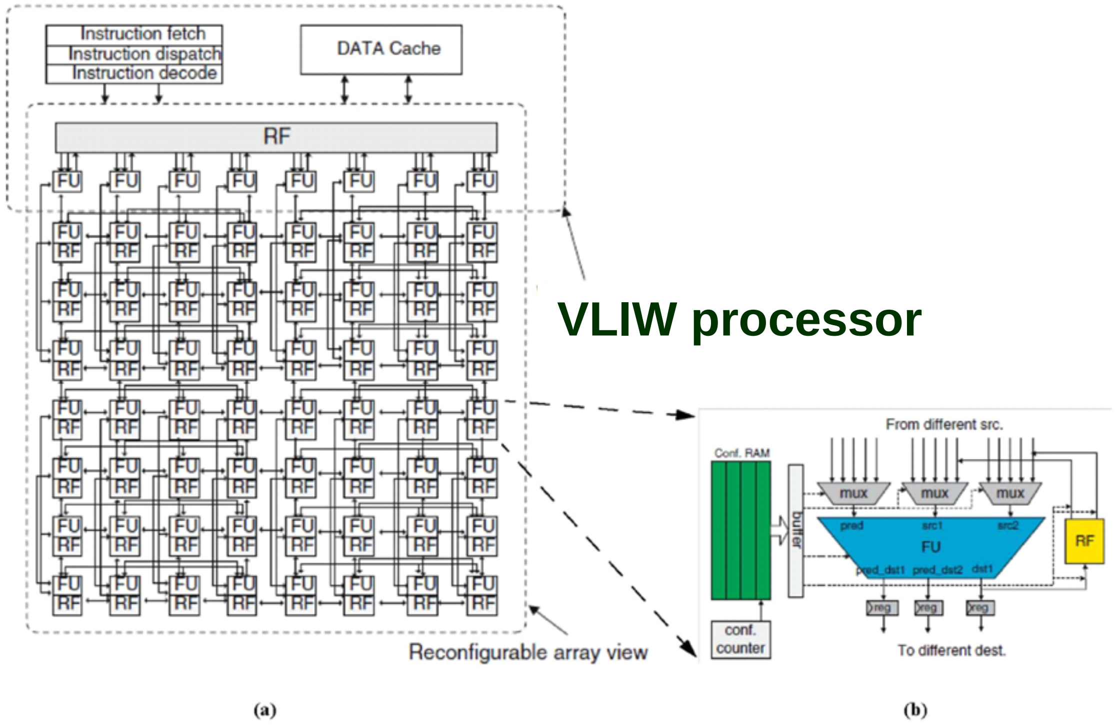
FV of Pipelined Processors with Reconfigurable Functional Units

A method to abstract reconfigurable functional units:



Details in [Velev & Gao, *ASP-DAC'11 Invited Talk*]

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).

Details in [Velev & Gao, *ICFEM'11*]

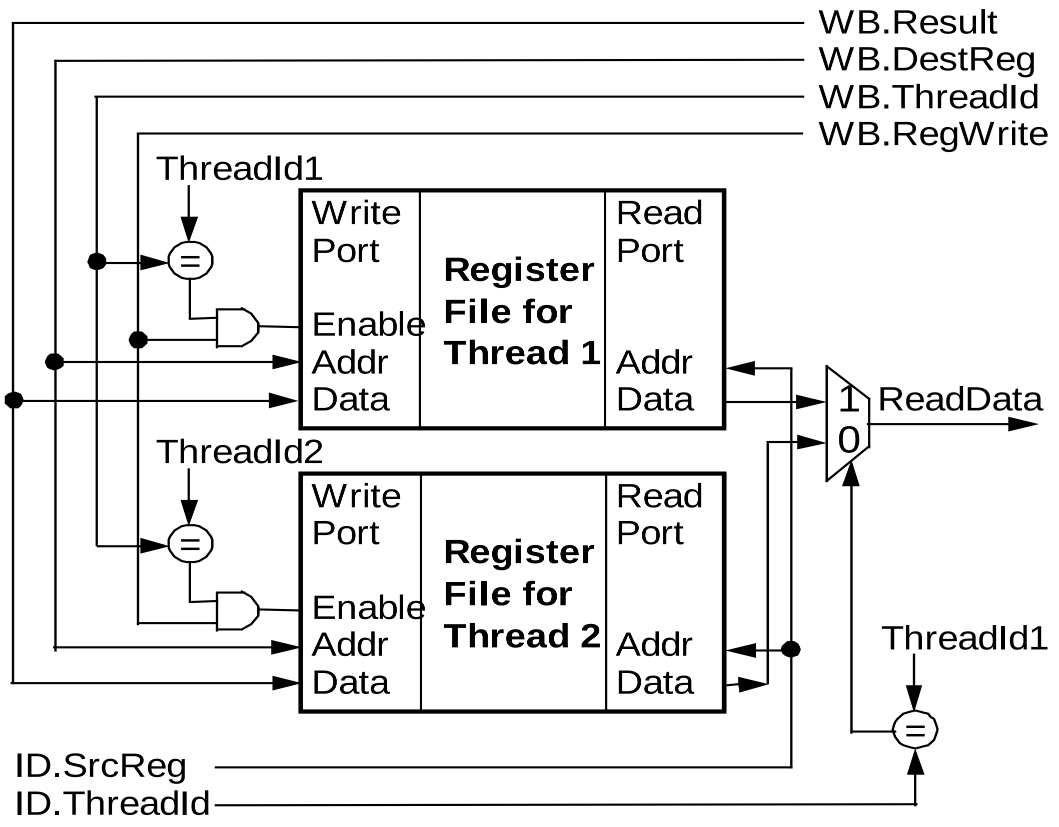
FV of Pipelined Processors with Hardware Support for Multithreading

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

- **Can scale for GPUs**

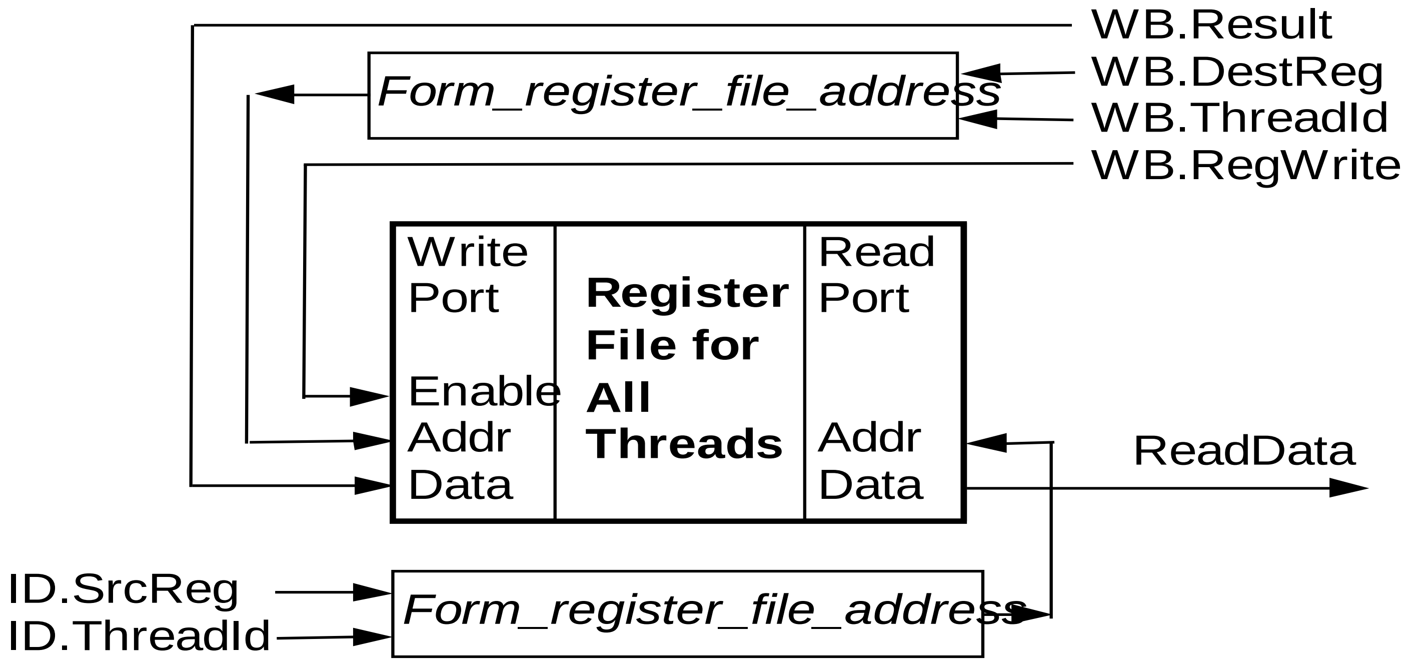
Details in [Velev & Gao, *ICCAD'11*]

Direct Model of Register File in Multithreaded Pipelined Processors



Located in the Instruction Decode (ID) stage, with results for writing from the Write Back (WB) stage

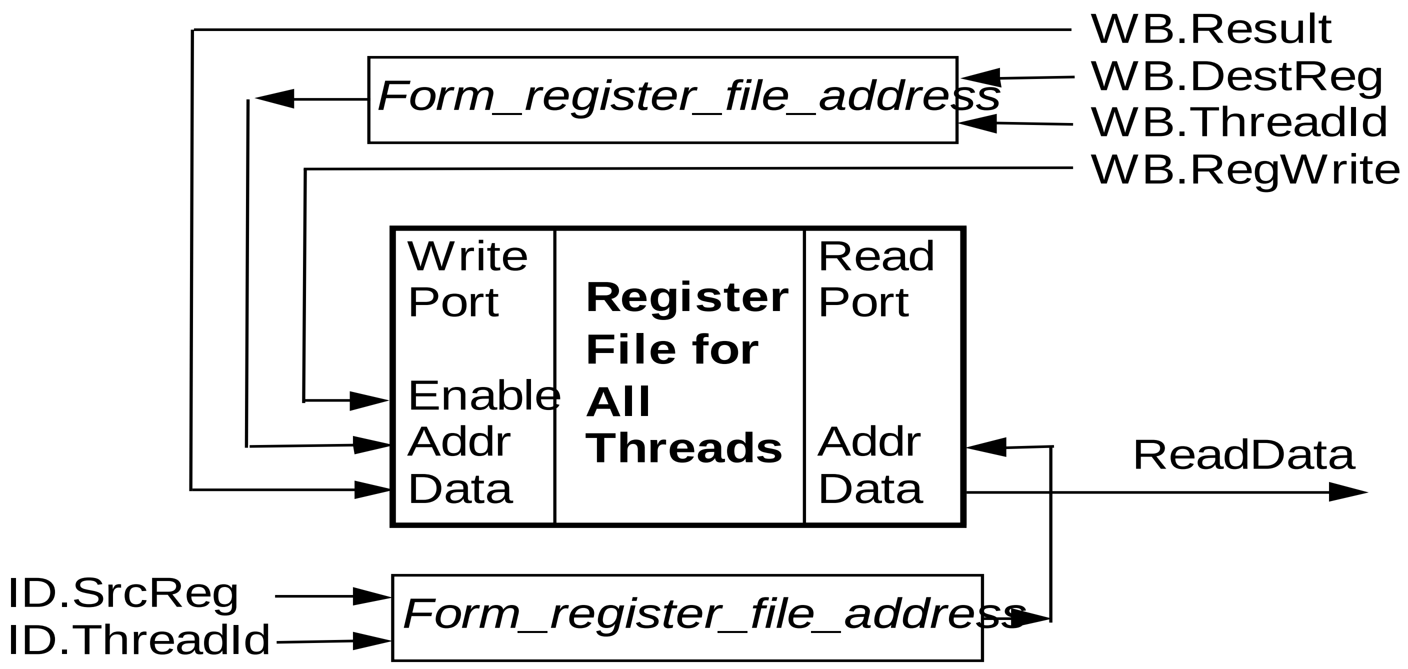
Abstracted Register File (1 of 2)



Unified model of all register files in pipelined processor with hardware support for any number of threads

UF *Form_register_file_address* abstracts concatenation of register id and its corresponding thread id

Abstracted Register File (2 of 2)

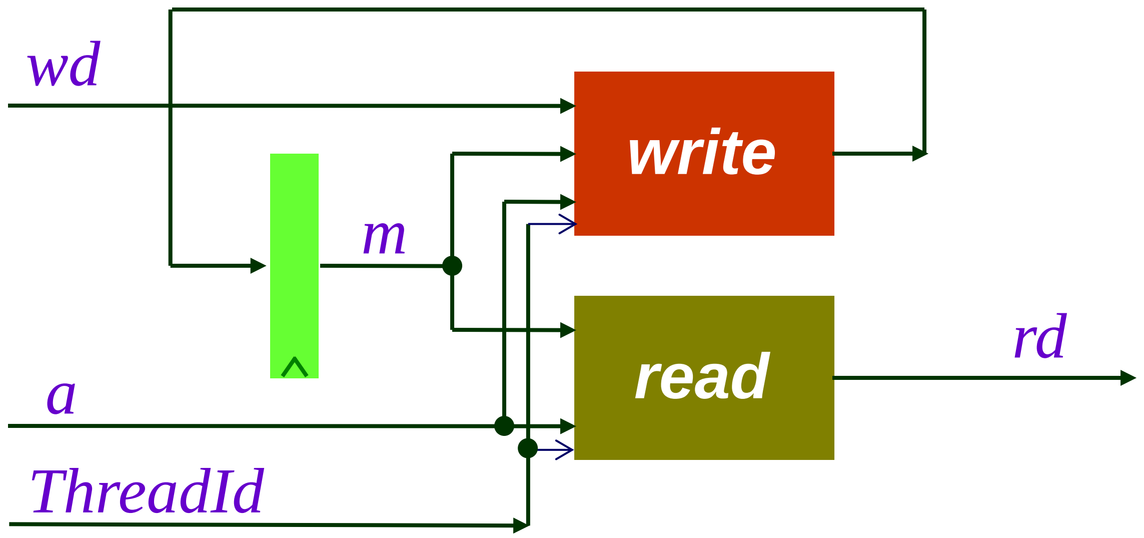


UF *Form_register_file_address* is also used in a modified version of forwarding and load-interlock stalling logic

This is *Design for Formal Verification*

Abstract in Same Way Other Architectural State Elements

E.g., Data Memory:



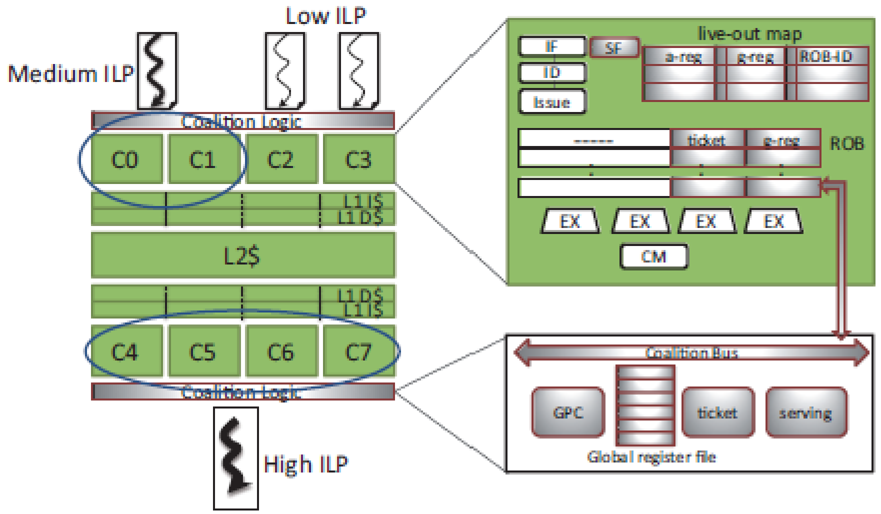
Results

Processor	CNF Vars	CNF Clauses	Time [s]
DSP_base	14,540	214,842	4.4
DSP_threads_4	63,271	1,448,725	24
DSP_threads_16	291,748	7,382,962	151
DSP_threads_64	1,519,228	33,891,549	885
DSP_threads_256	6,912,327	151,367,229	5,908
DSP_abstraction	18,936	316,120	5.1

Details in [Velev & Gao, *ICCAD'11*]

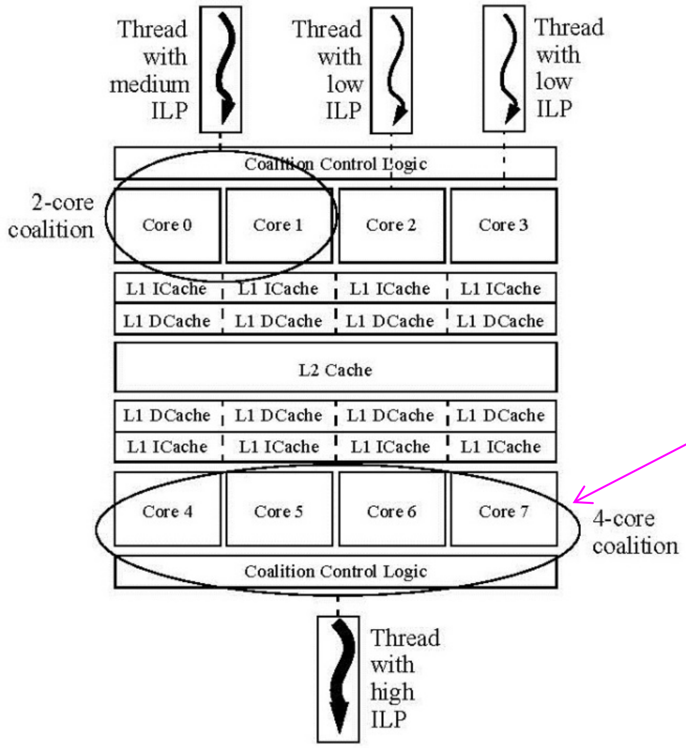
Polymorphic Heterogeneous Multi-Core Architectures

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

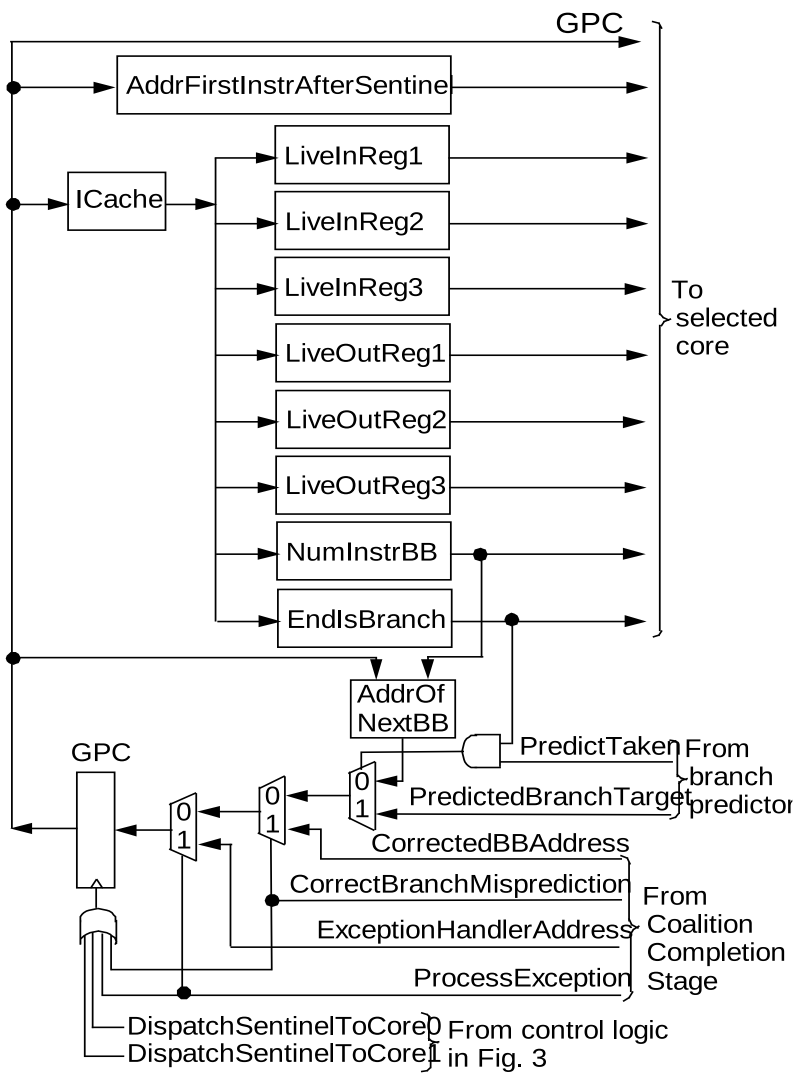


Our method was showcased in *NASA Tech Briefs* (LEW-19207-1, 2014), which publishes only the best NASA-funded inventions

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

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

Abstraction of Coalition Dispatch

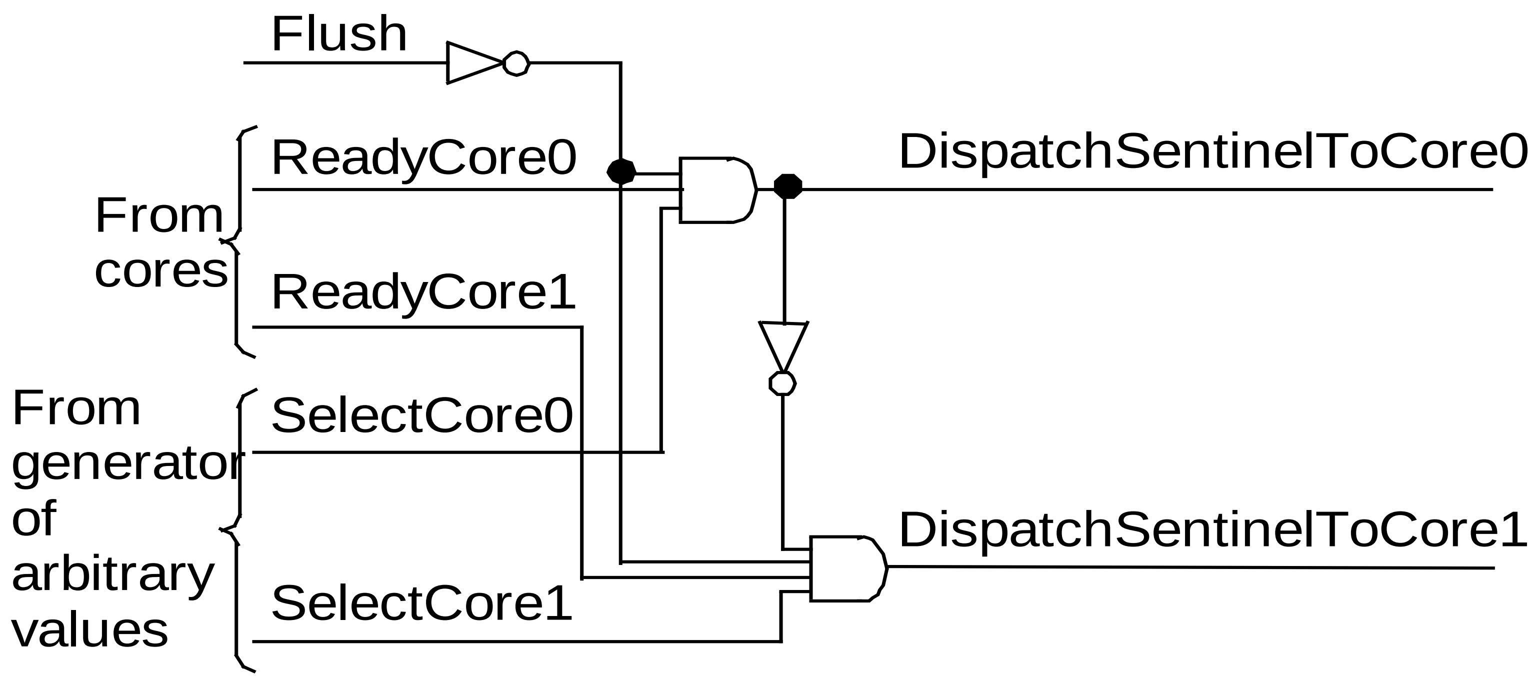


GPC = General PC

- 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

Results

Experiments on 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)

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

Details in [Velev & Gao, *ISQED'14*]

Outline

✓ EUFM Background: Positive Equality

✓ Block-Level Translation to SAT

✓ 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

Positive Equality resulted in at least 5 orders of magnitude speedup when formally verifying complex dual-issue superscalar processors, which take 1 sec to formally verify, and the speedup is increasing with the processor complexity

Block-level translation to SAT produced at least 2 additional orders of magnitude (420×) speedup

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.

Questions?

Keynote Speaker:

Miroslav N. Velev (miroslav.velev@aries-da.com)

ARIES DESIGN AUTOMATION®