Gap-Free verification of weakly programmable IPs against their operational ISA model

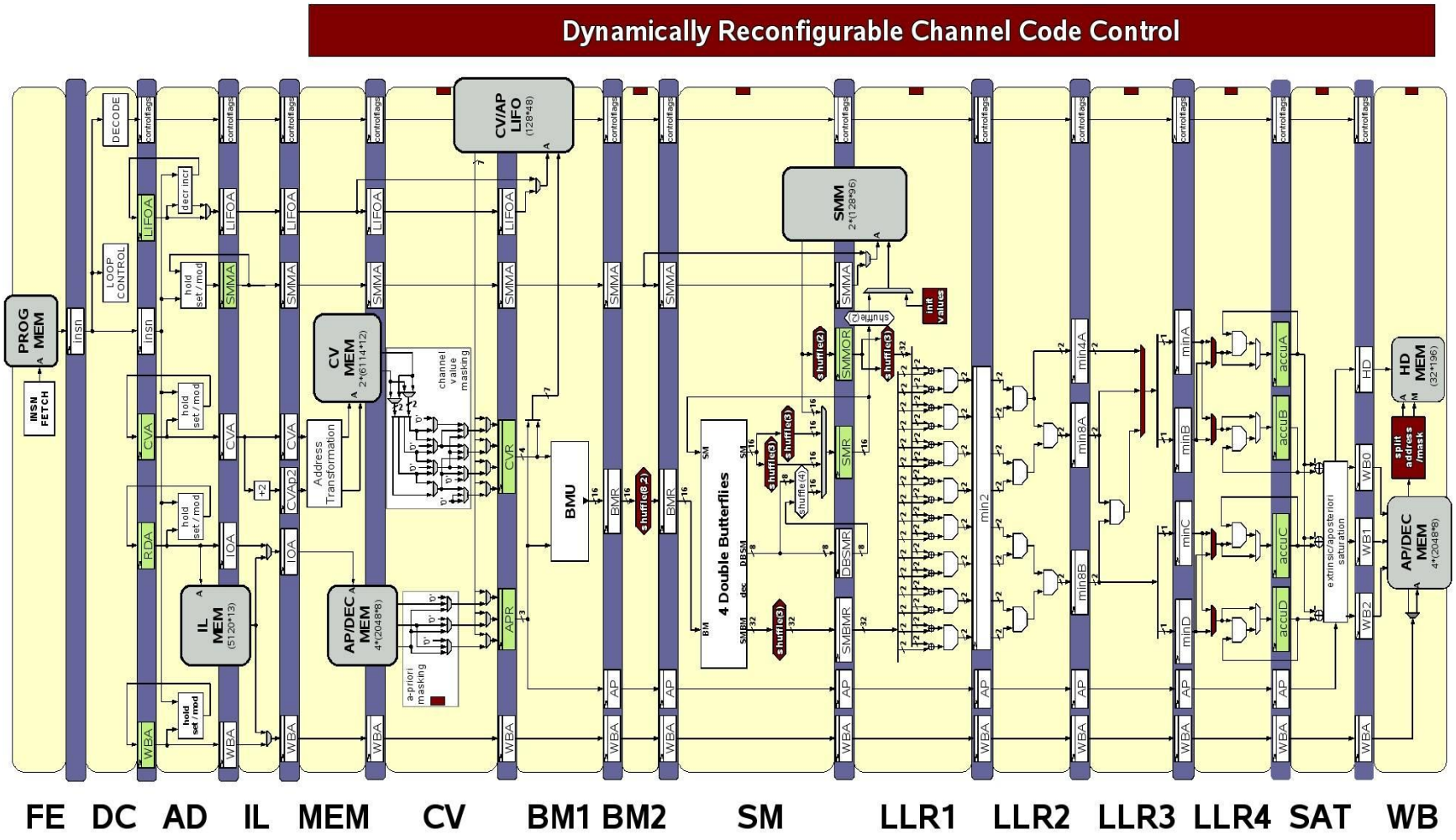
Markus Wedler, Sacha Loitz, Wolfgang Kunz
Department of Electrical and Computer Engineering

University of Kaiserslautern/Germany

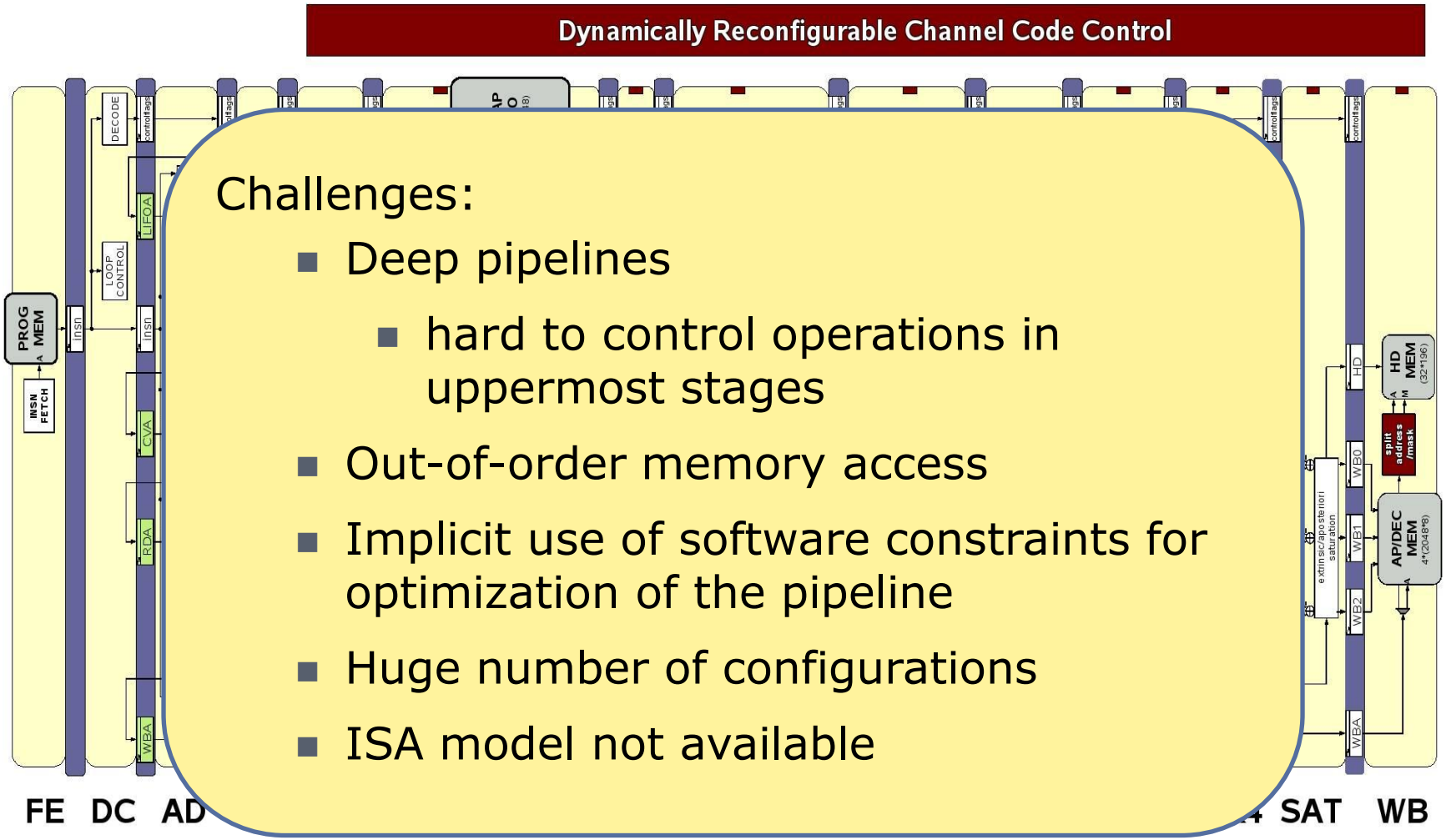
Outline

- Challenges for Formal Verification imposed by Weakly-Programmable IPs (WPIP)
- Interval Property Checking
  - Specification Methodology
  - Gap-Free Specifications
- Operational ISA model
  - Operation-oriented specification
  - Software Constraints
  - Completeness considerations
- Applications

Example WPIP FlexiTreP



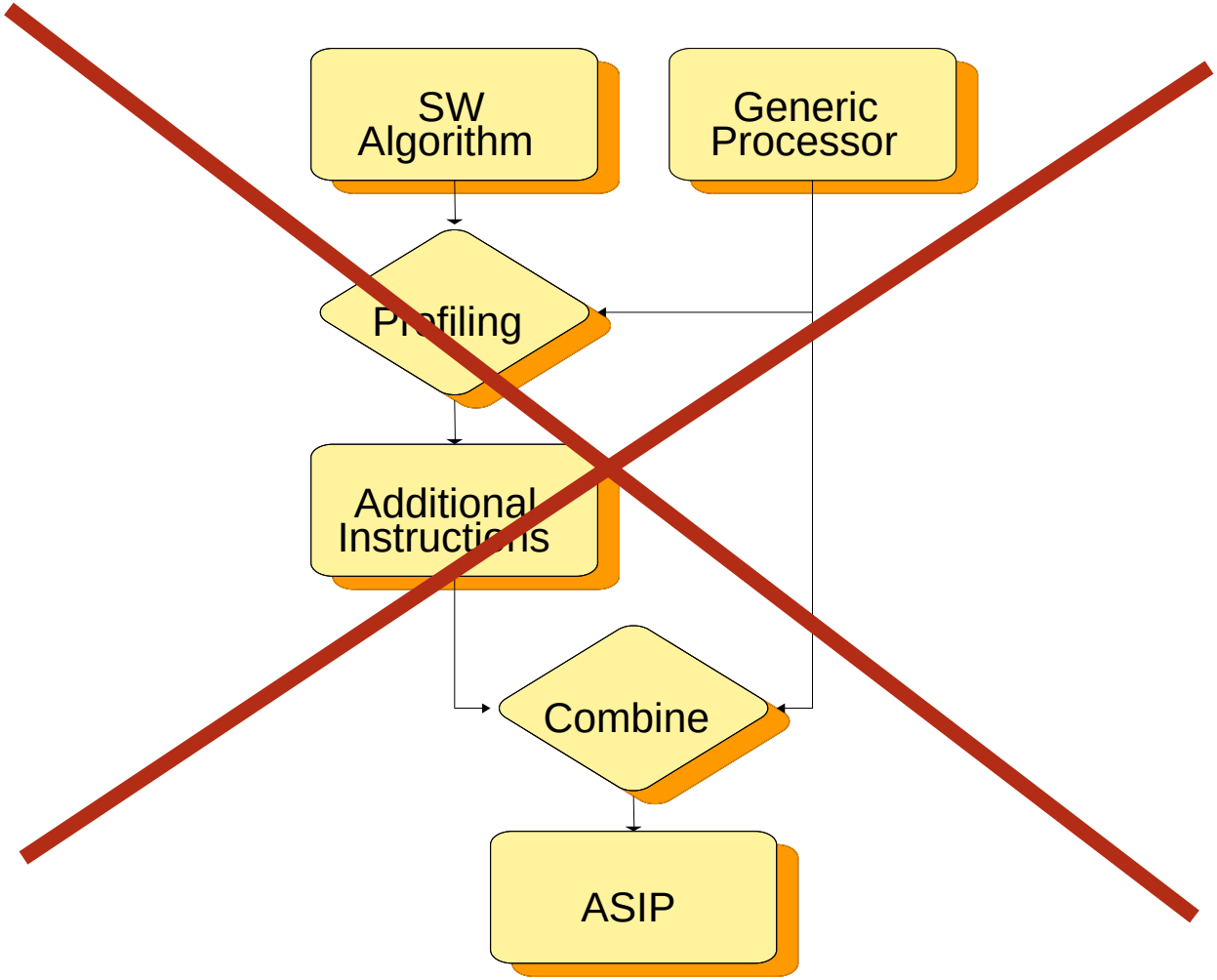
Example WPIP FlexiTreP



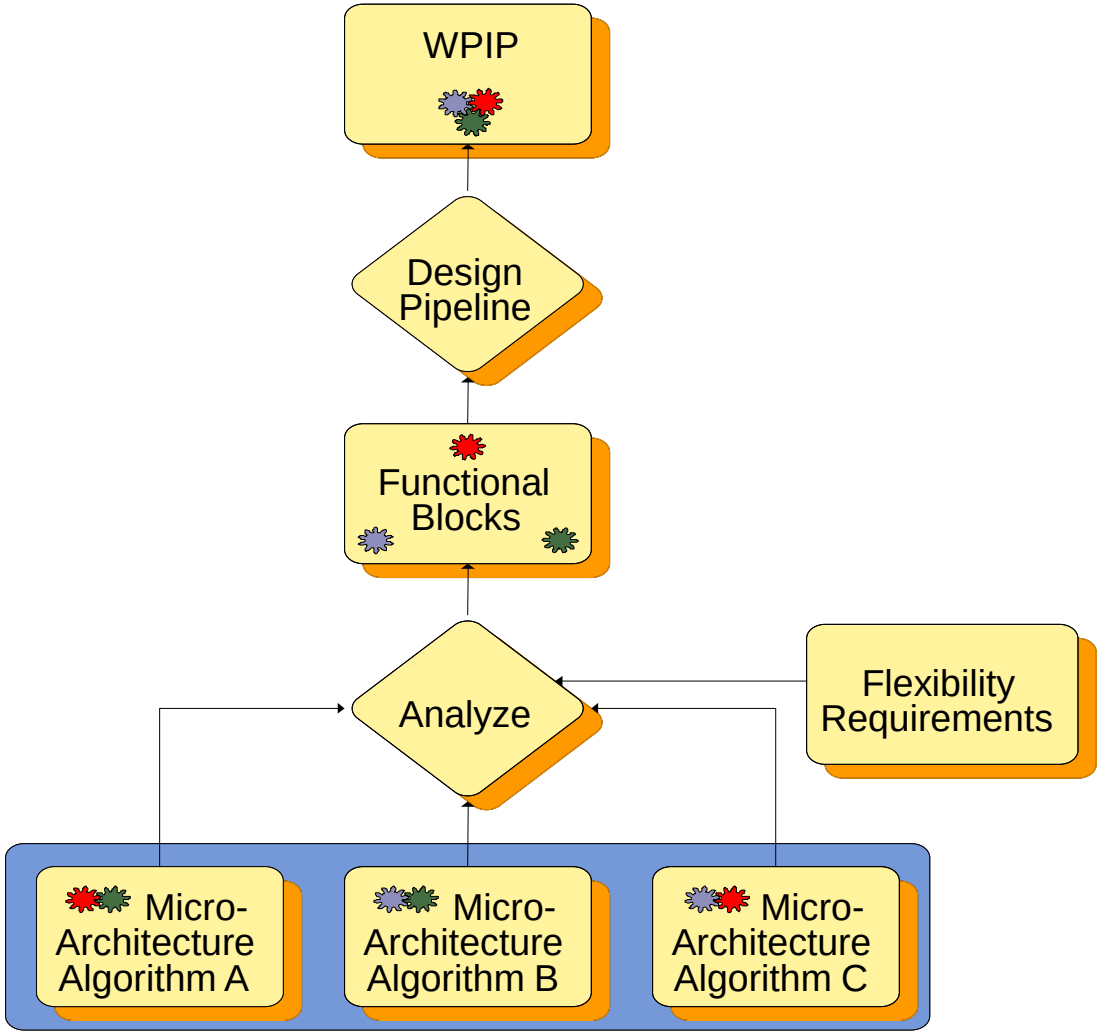
Challenges:

- Deep pipelines
  - hard to control operations in uppermost stages
- Out-of-order memory access
- Implicit use of software constraints for optimization of the pipeline
- Huge number of configurations
- ISA model not available

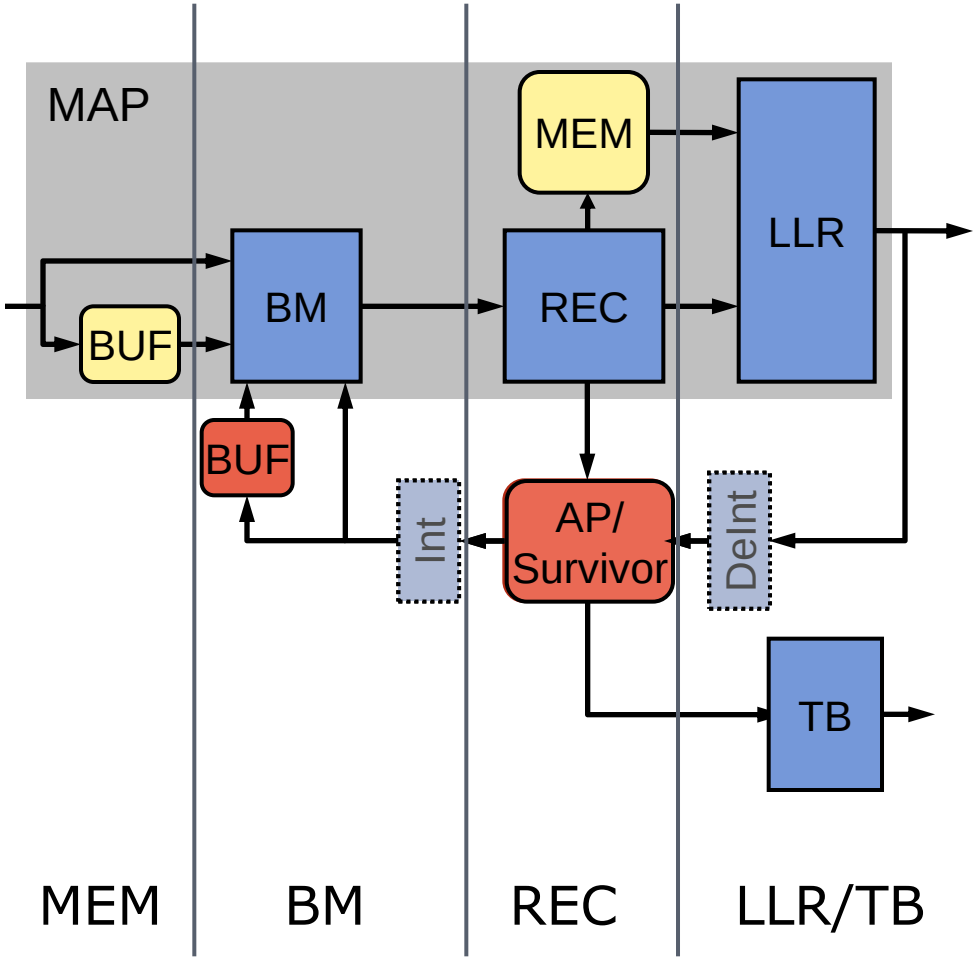
Standard design flow for ASIPs



Bottom-up Design Flow for WPIPs



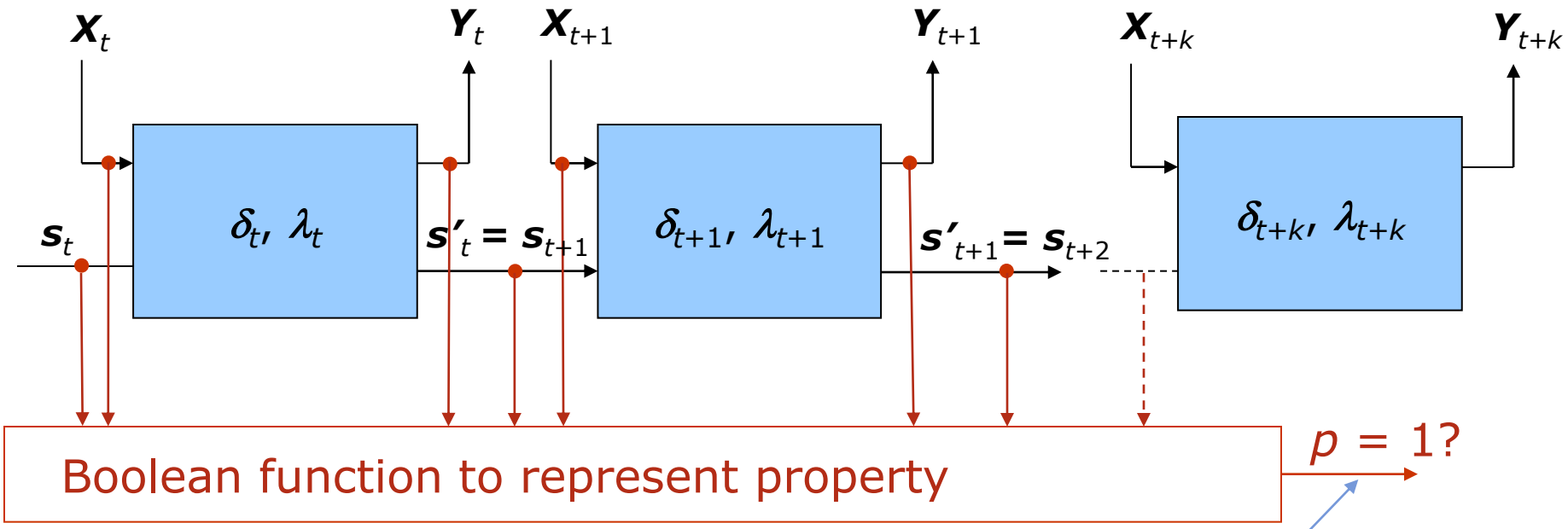
WPIP FlexiTreP



- MAP micro-architecture
- Turbo micro-architecture
- Viterbi micro-architecture

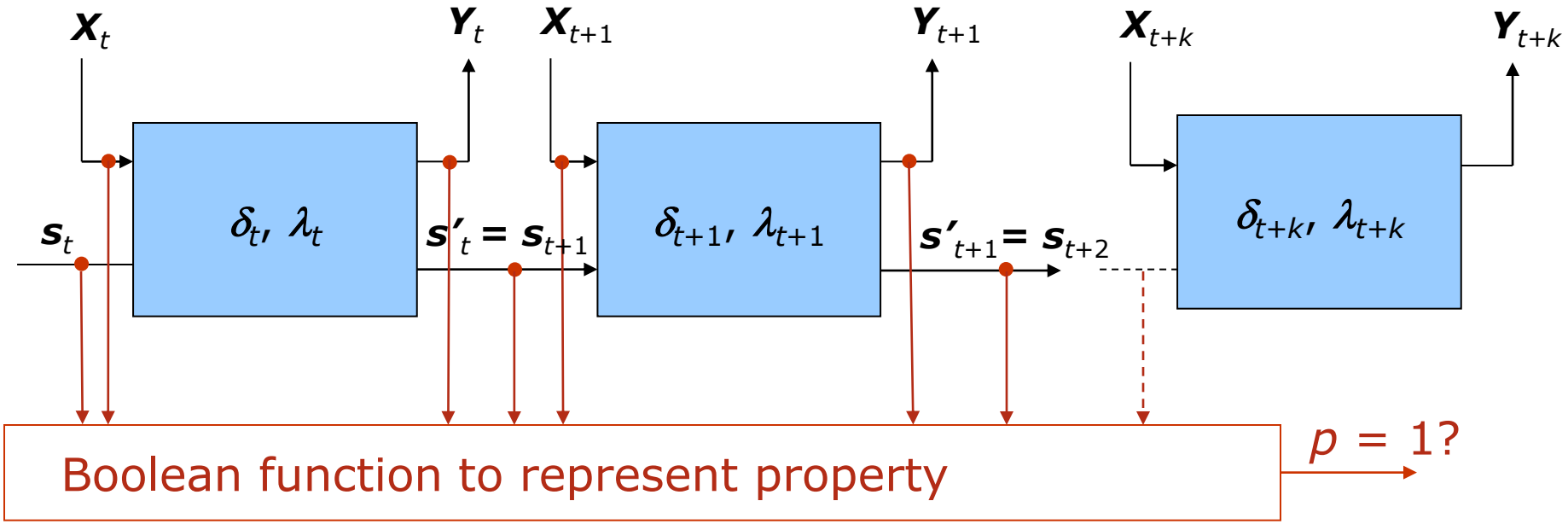
SAT-based Property Checking

Iterative Circuit Model: from $i = t$ to $i = t + k$



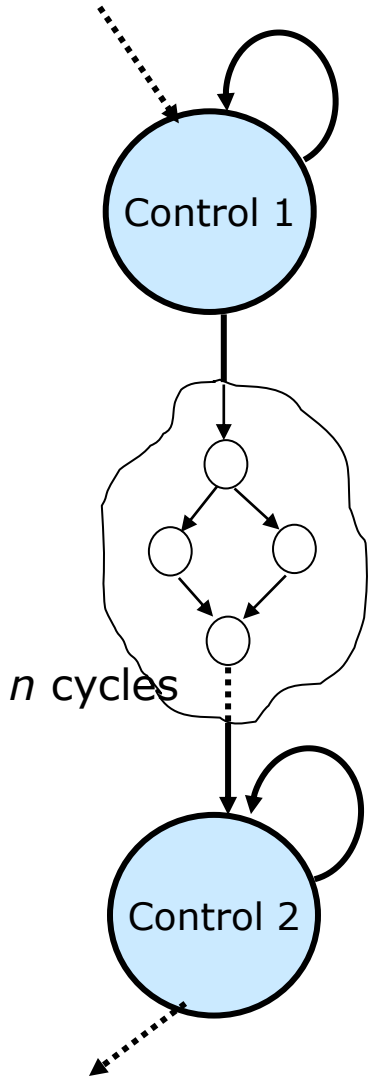
Boolean satisfiability problem (SAT)
SAT modulo Theory (SMT) problem

SAT-based Property Checking



- Unsatisfiability guarantees unbounded validity of **G**(*p*)
- *p* is specified by a *timed Boolean predicate* (*TBP*) in terms of design signals consisting of:
  - Boolean connectives (∧,∨,...)
  - Generated next state operator $\mathbf{X}^t$
- A TBP *p* refers to bounded inspection interval of time $[t_f,t_l]$

RT-level module verification: operation by operation



Typical methodology for Property Checking of SoC modules:

- Adopt an operational view of the design
- Each operation can be associated with certain *important control states* in which the operation starts and ends
- Specify a set of properties for every operation, i.e., for every important control state
- Verify the module *operation by operation* by moving along the important control states of the design
- The module is verified when every operation has been covered by a set of properties

Property Checking of processor pipeline

Goal: Prove that instructions are performed correctly

Spec: Safety properties of type: **G**(*a*→*c*) with bounded inspection interval

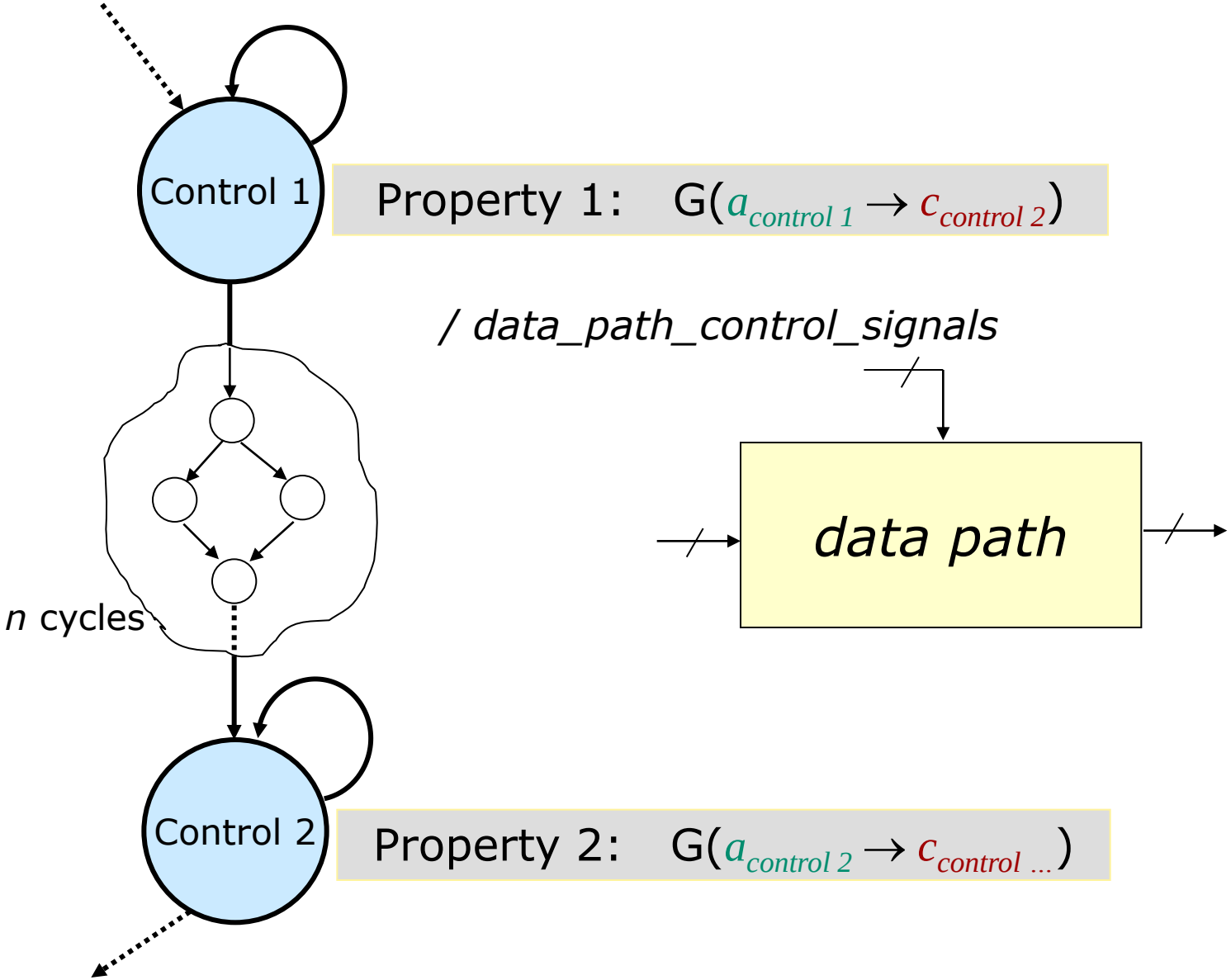
Example: Property in ITL (Interval Language)

"assumptions"

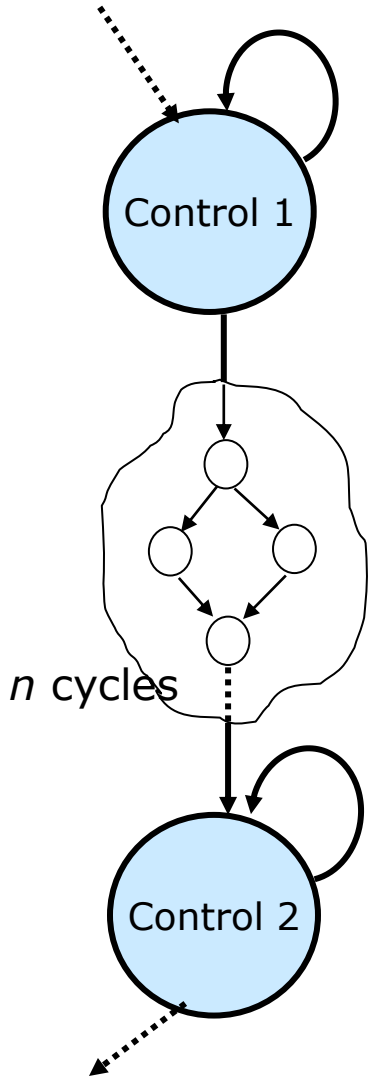
"commitments"

```
property instr_XYZ
assume:
  at t: next_instr_can_be_issued();
  at t: command_dec(XYZ,res,op1,op2);
  during[t,t+3]: no_reset;
  during[t,t+3]: no_interrupt;
  …
prove:
  at t+3: res == compute_res(XYZ,op1,op2);
  at t+3: stable_other_regs(res);
  at t+1: next_instr_can_be_issued();
end
```

CPU verification: instruction by instruction



RT-level module verification: operation by operation



Typical methodology for property checking of SoC modules:

- Adopt an operational view of the design
- Each operation is associated with certain *states* in which it ends
- Every important
- *Verification by* operation along the important control states of the design
- The module is verified when every operation has been covered by a set of properties

How to guarantee that every scenario is covered?

Mutation coverage

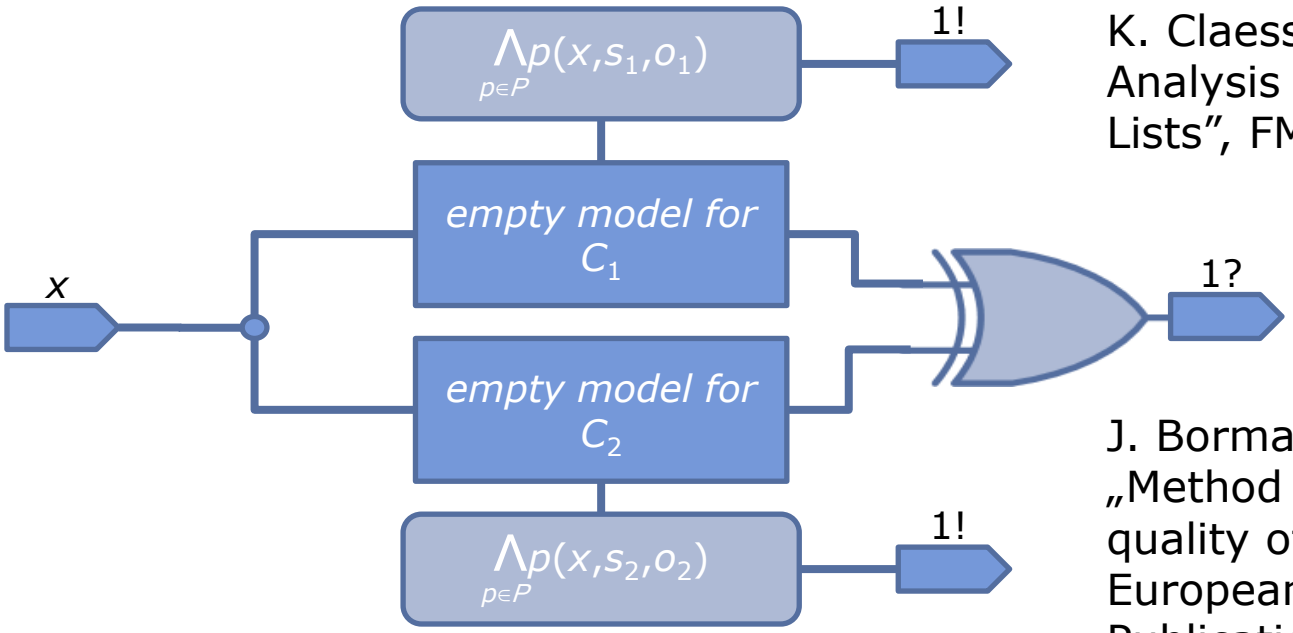
A set of (operational) properties *P* is complete for a design *C with respect to a set of mutations* $M=\{C_1,\ldots,C_n\}$, if *C* satisfies the properties in *P* and for every mutation C_i at least one property fails.

Problems:

- Criterion design-dependent
- Do the mutations reflect designer mistakes?

Completeness

A set of (operational) properties P is complete if every two designs C_1, C_2 satisfying the properties in P are sequentially equivalent.

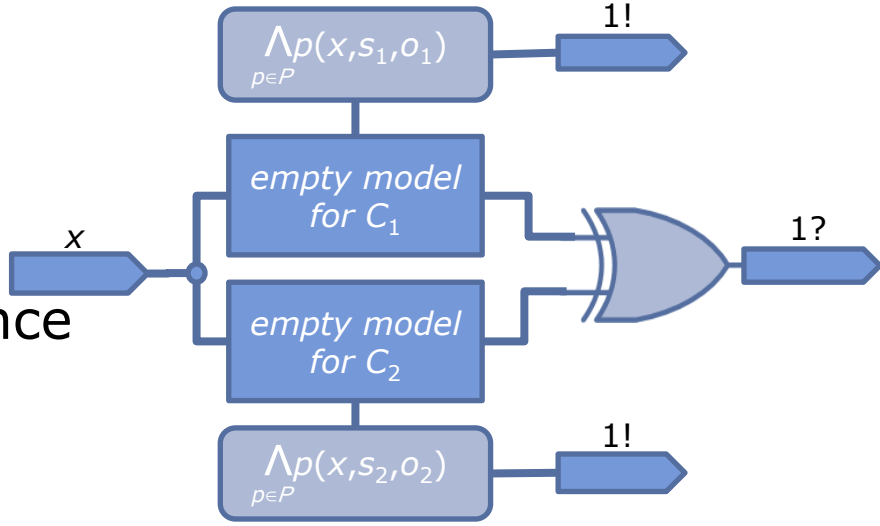


K. Claessen: "A Coverage Analysis for Safety Property Lists", FMCAD 2007

J. Bormann and H. Busch: „Method for determining the quality of a set of properties" European Patent Application, Publication Number EP1764715, 2005.

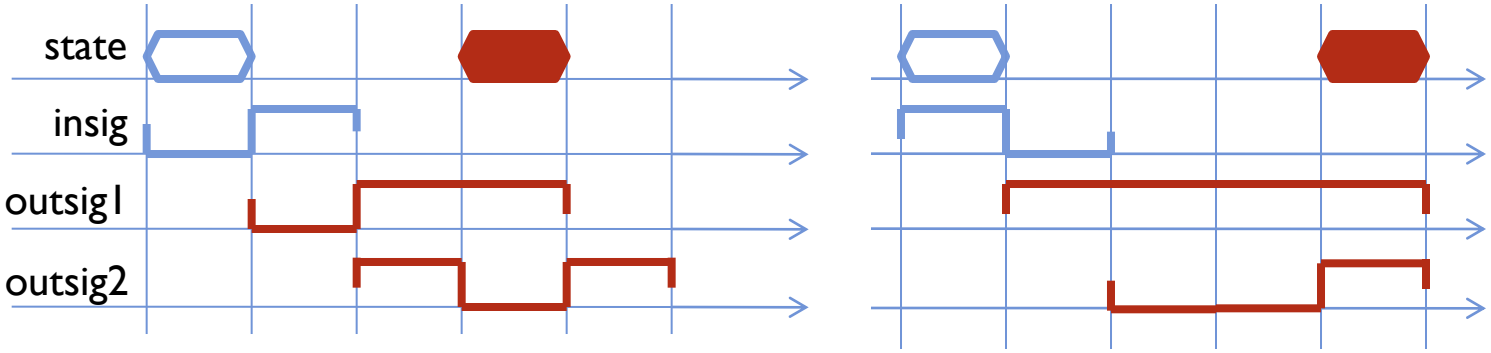
Completeness

- Practical extensions:
  - Allow explicit constraints on inputs of designs
  - Weaken sequential equivalence condition by introduction of determination requirements



- Decompose proof with respect to the given properties $p \in P$.
  - Sucessor /Case-Split Test:
    Every input trace can be covered with a uniquely determined sequence of properties $(p_i \mid i \in \mathbb{N})$ such that the determination intervals match without gaps.
  - Determination Test:
    Every property uniquely determines the outputs within its determination interval.

Completeness



- Decompose proof with respect to the given properties $p \in P$.
  - Sucessor /Case-Split Test:
    Every input trace can be covered with a uniquely determined sequence of properties $(p_i \mid i \in \mathbb{N})$ such that the determination intervals match without gaps.
  - Determination Test:
    Every property uniquely determines the outputs within its determination interval.

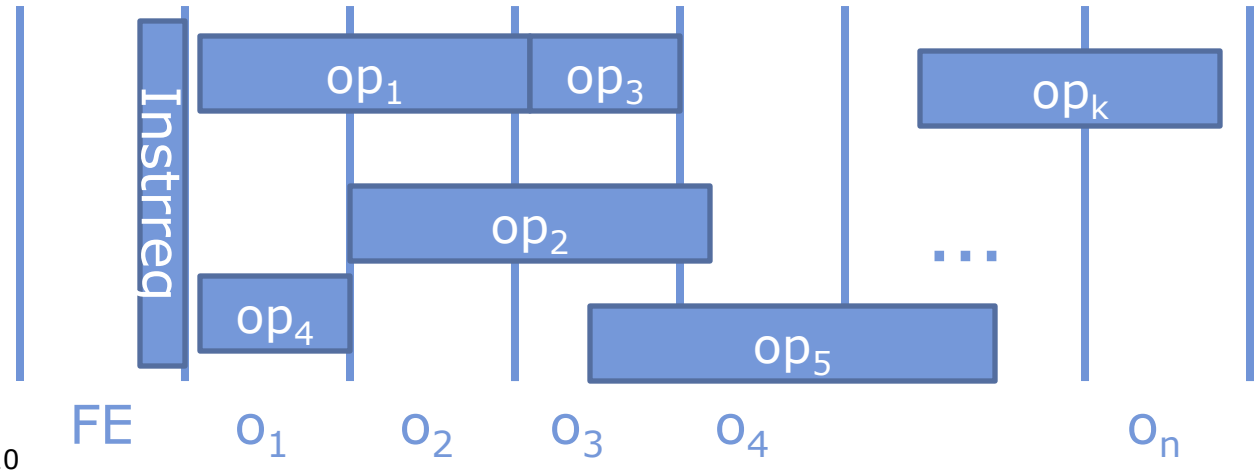
Operational ISA model

- Due to specific programming models WPIPs often lack a classical ISA model
  - Instructions correspond to hundreds of classical RISC instructions (referred to a nuclei)
  - Semantics often implicitly given by functional blocks (operations) involved in the execution

How to specify functional behavior of a WPIP?

Operational ISA model

- The operational ISA model for a WPIP consists of:
  - A relation $OISA \subseteq I \times O$ between the set of instructions I and the set of (pipeline) operations O
  - Timed Boolean predicates:
    - $instr_i Fetched()$: determines whether the instruction $i \in I$ is issued into the pipeline at a time-point t
    - $op_o()$: specifies functionality of the operation $o \in O$



Operational ISA model

Manual specifications given by the verification engineer

- $OISA \subseteq I \times O$
- $instr_i Fetched()$: determines whether the instruction $i \in I$ is issued into the pipeline at a time-point t
- $op_o()$: specifies functionality of the operation $o \in O$

Everything else will be generated automatically!

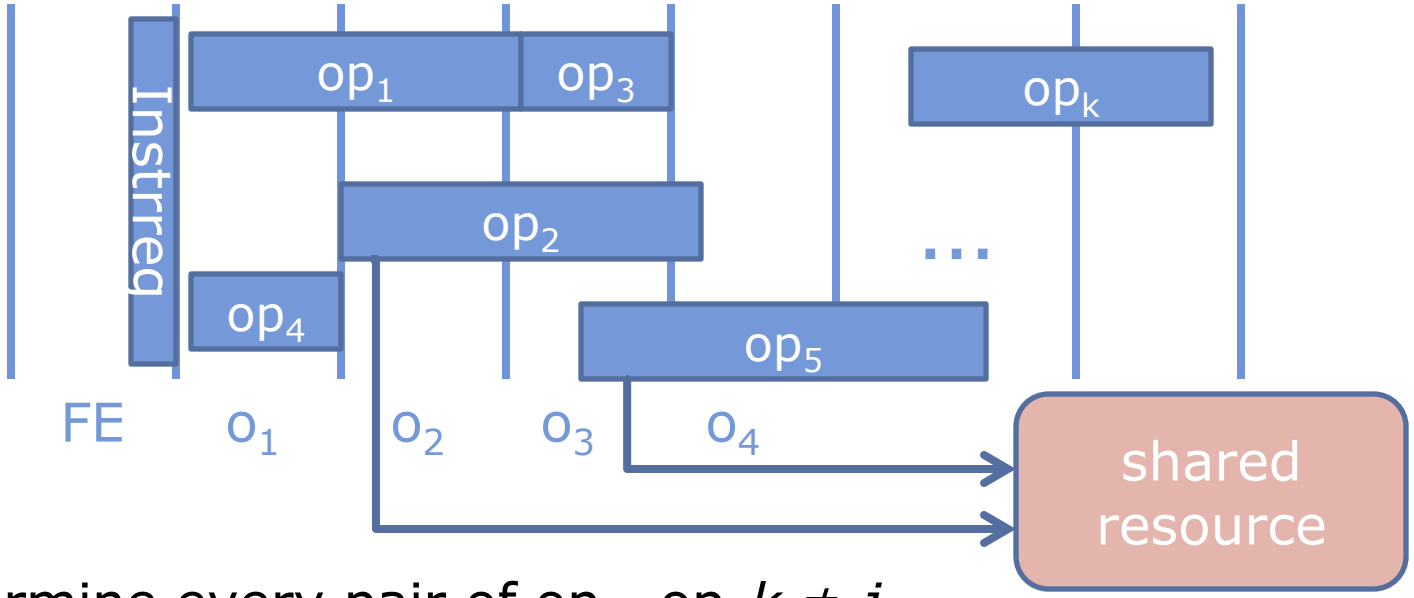
Operational ISA model

- Timed Boolean predicates that are automatically generated from operational ISA model:
  - $instr_i Performed() = \bigwedge_{(i,o) \in OISA} op_o()$
  - $op_o Triggered() = \bigvee_{(i,o) \in OISA} instr_i Fetched()$
  - Per-Instruction properties:
    - $instr_i Exec() = nextInstrState() \wedge instr_i Fetched() \rightarrow instr_i Performed() \wedge X^{t(i)}\ nextInstrState()$

Just another operation

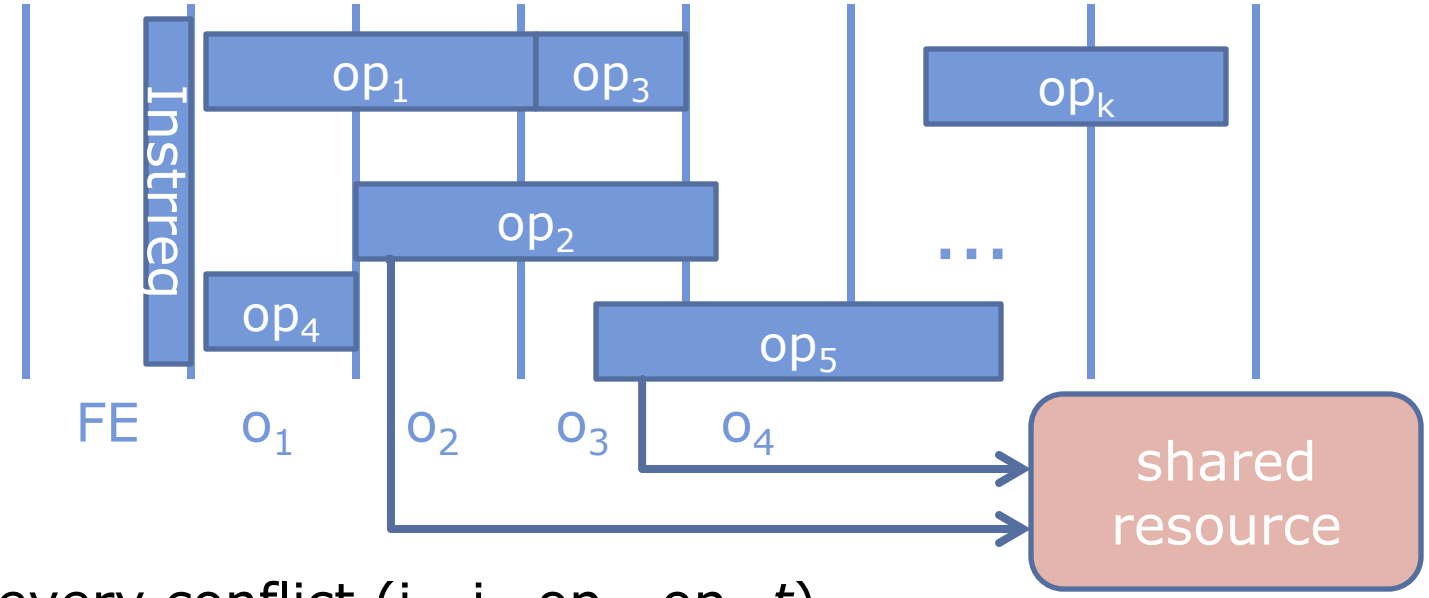
  - Per-Operation properties:
    - $op_o Exec() = nextInstrState() \wedge op_o Triggered() \rightarrow op_o()$

Hazards imply software constraints



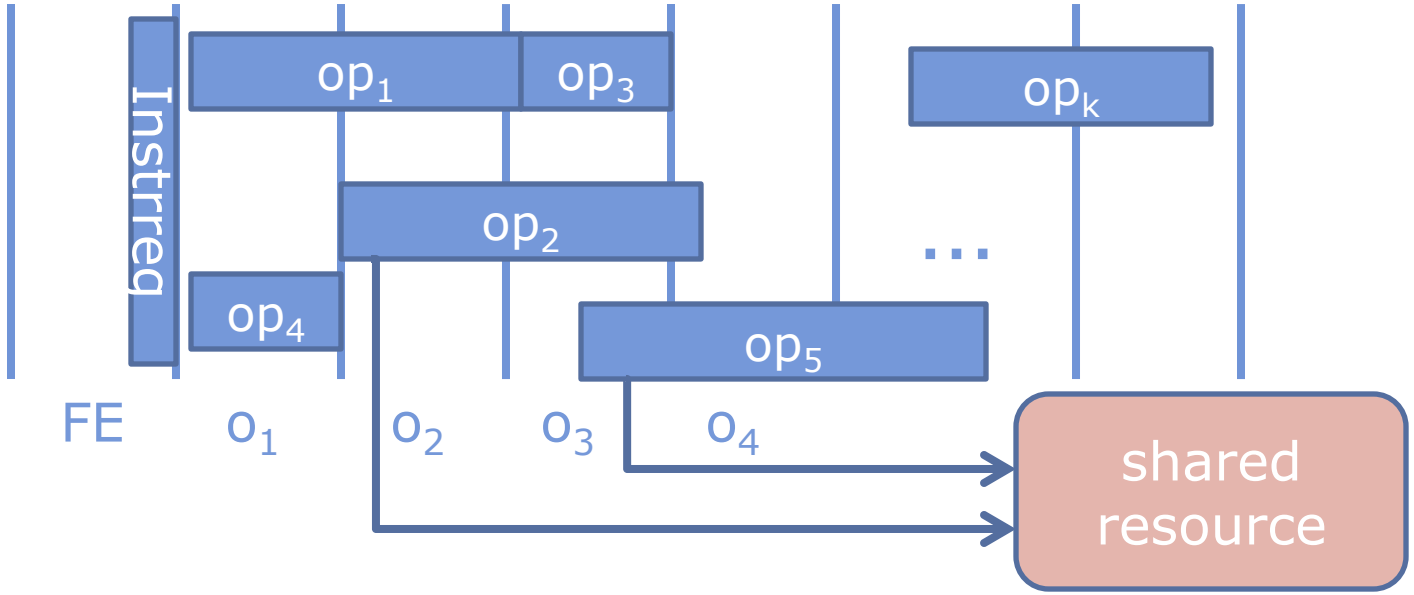
- Determine every pair of op_k, op_j $k \neq j$ that refer to the same resource with time slack t
- For all related instructions i_k, i_j store (i_k, i_j, op_k, op_j ,t) in conflict list

Hazards imply software constraints



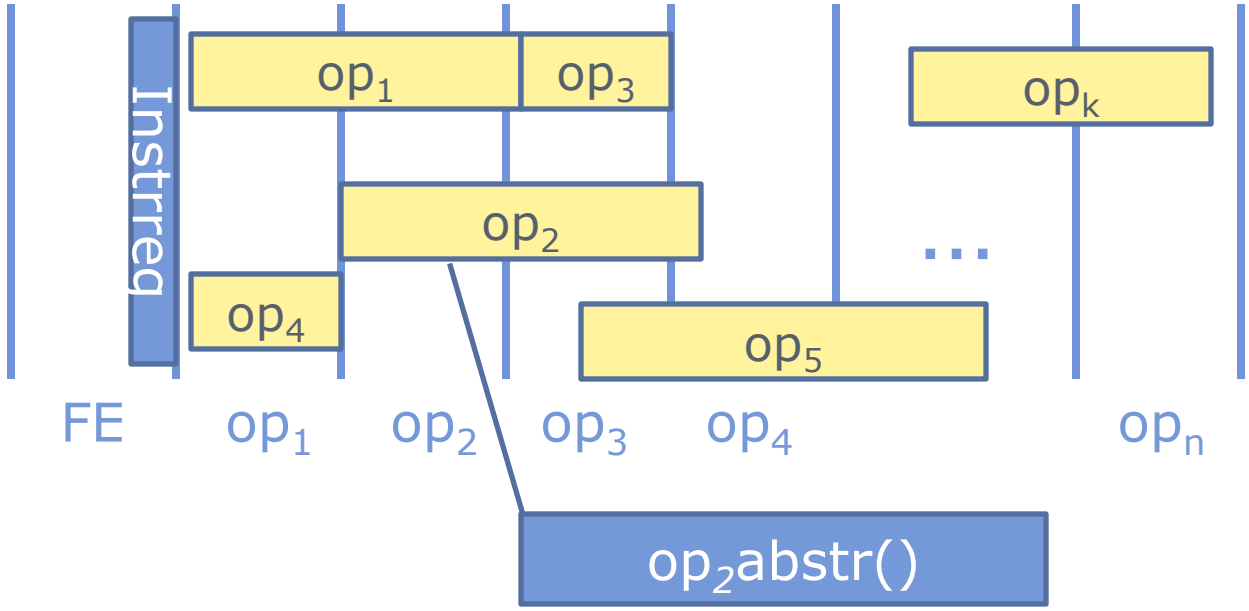
- For every conflict (i_k, i_j, op_k, op_j ,t) in conflict list decide:
  - Store automatically generated constraint that forbids sequences where i_k follows i_j after t clock cycles
  - Manually find weaker constraint
    - $swConstraint_{j,k}() = instr_{i_k}Fetched() \rightarrow flags_{i_k}()$

Software compliance with constraints



- Strong abstraction feasible for checking compliance of software with detected and now explicitly specified constraints

Software compliance with constraints



- Strong abstraction feasible for checking compliance of software with detected and now explicitly specified constraints
  - Empty models for operations (only signal names)
  - TBPs op_kabstr() describe abstracted behavior
    - Consider behavior of $flags_{i_k}$() only

Completeness by construction

- Case split and successor tests obviously hold and this can easily be verified by a completeness checker

Problem:

- TBPs for operations $op_o()$ only describe modified values for involved state holding elements

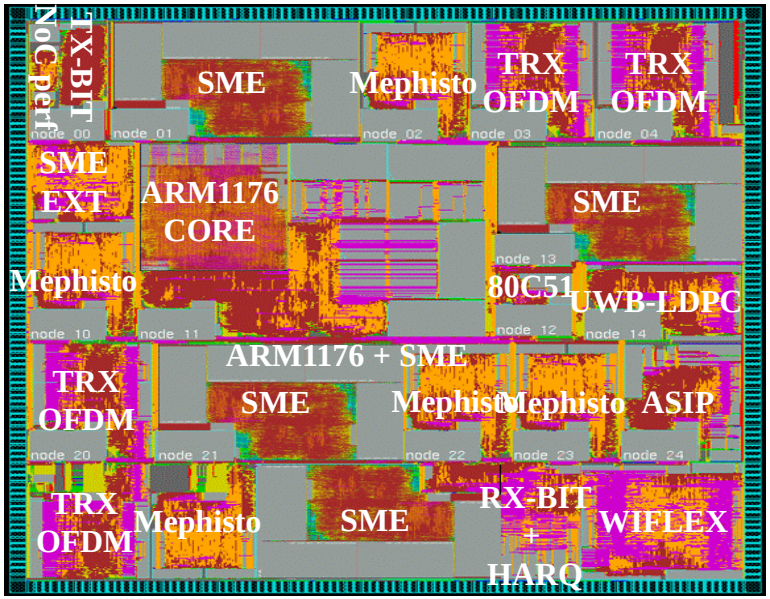
  ⇒ other registers/memory cells remain undetermined

- Description of default behavior is required
  - keep value
  - take default value
- Tedious identification of situations where default behavior needs to be applied is completely automated

Experimental Results

- HW verification:
  - MAP and FlexiTreP, two WPIPs for channel decoding were successfully verified.
  - During the verification subtle HW bugs were discovered which had escaped sign-off simulation before
  - FlexiTreP has been taped out successfully

- 65nm low power technology
- 41741 standard cells, 15 macros
- Die size without interface 0.74 mm^2
- 360Mhz, core power ~100mW@1.1V
- Logic utilization 77%
- Silicon available since March 2009



LETI MAGALI MPSoC Chip

4G mobile baseband IP demonstrator

Design characteristics

	MAP	FlexiTreP
# Instructions	16	104
Lines of RTL Code	22689	114040
Lines of ADL Code	1521	8634
# Operations (properties)	28	83
# Generated properties	14	52
CPU Time regression	37,67 s	18h
Memory Usage	593 MB	14,3 GB

Intel(R) Xeon(R) CPU E5440 @ 2.83GHz / SUSE 11.1

Bugs discovered by FV

- Wrong sign extensions: `res = op1 + op2`
- Wrong saturation condition in stage 13 out of 14
- Confirmed bug in RTL code generation for nested if-then-else statement of commercial ASIP design tool identified
- Scenario for a race condition of parallel value assignments to the same variable identified
- Software constraints have been ignored by some programs

Results for automatic completion

- FlexiTreP (for industrial application)
  - Automatic completion of the OISA model revealed several inconsistencies/gaps within the property suite
  - All inconsistencies have been successfully resolved
  - All gaps have been closed
- MAP
  - SW-constraints and TBPs for default behavior have originally been set up manually.
  - Automatic analysis revealed that the manual process missed important software constraints
  - Completeness of the generated property set successfully proven with OneSpin 360 MV
    - Additional manual effort one week