Formal Techniques for Hardware/Software Co-Verification

Daniel Kroening, Mandayam Srivas

26th International Conference on VLSI
January 2013
Pune, India

Acknowledgements: Madukar Kumar Singh, TRDC
Nassim Seghir, Oxford

cmi

UNIVERSITY OF OXFORD

Speaker Introduction: Daniel Kroening

Professor of Computer Science, University of Oxford

2007–2010	University of Oxford
2004–2007	Assistant professor, ETH Zürich
2001–2004	Post-doc at CMU
2001	PhD. Computer Engineering Saarland University

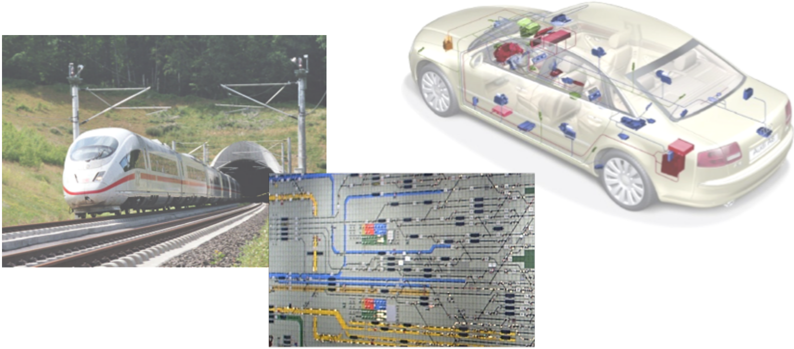
Speaker Introduction: Mandayam Srivas

Professor, Computer Science, Chennai Mathematical Institute, India
Research Scientist, Oxford University

2006–2012	General Manager, Texas Instruments
2003–2006	Director, Nulife Semiconductor
2001–2003	Verification Technologist, RealChip
1990–2001	Research Scientist, SRI International, Menlo Park, CA
1984–1990	CS Department, SUNY Stony Brook
1982	PhD, Computer Science, MIT, USA

Motivation

Software *most complex component* of critical systems



Motivation

- Manual inspection is error-prone and costly
  ⇒ Tool support needed

- Tools that rely on test-vectors
  - require human expertise;
  - may miss bugs.
  - are too labour-intensive for most projects

More detail here:

Vijay D'Silva, Daniel Kroening, Georg Weissenbacher,
"A Survey of Automated Techniques for Formal Software Verification",
IEEE Transactions on Computer Aided Design

IEEE TRANSACTIONS ON
COMPUTER-AIDED DESIGN
OF INTEGRATED CIRCUITS AND SYSTEMS

IEEE

`http://www.kroening.com/papers/tcad-sw-2008.pdf`

IEEE TCAD? Isn’t that a hardware journal?

- Tool-market in HW-design is well established ($\sim$ 4 Bn. US$/year)
- It’s normal to buy tools to improve design productivity
- Trend towards HW/SW co-verification
- EDA vendors are getting interested in this market!

Some History

Mathematical reasoning about programs is the oldest thing in computer science!

70s:

- Hoare, Dijkstra, ...: prove programs correct!
- Idea: prove your program correct wrt a specification
- Or: write a specification, and refine it into a program

- Typically considered too expensive for most programs!

Context: The Verifying Compiler

Tony Hoare

The Verifying Compiler:
a Grand Challenge for
Computing Research

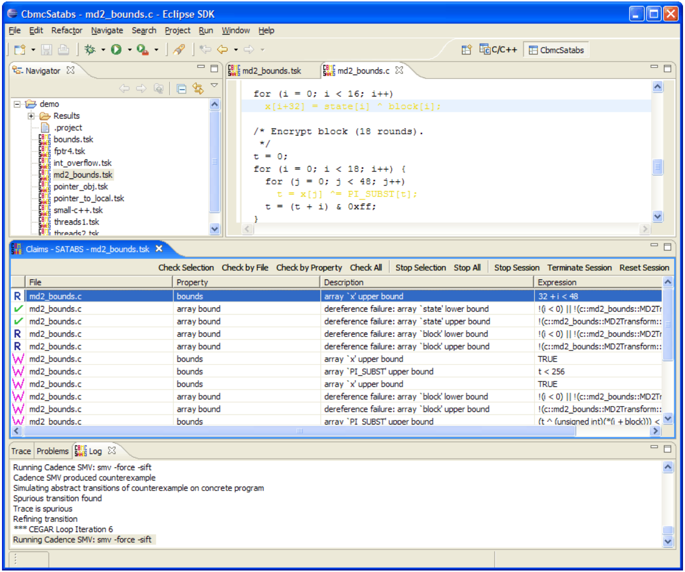
What Kind of Software?

```
State { int created = 0; }

IoCreateDevice.exit {
   if ($return==STATUS_SUCCESS)
     created = 1;
}

IoDeleteDevice.exit { created = 0; }

fun_AddDevice.exit {
   if (created && (pdevobj->Flags &
     DO_DEVICE_INITIALIZING) != 0) {
     abort "AddDevice routine failed to set "
             "~DO_DEVICE_INITIALIZING flag";
   }
}
```

Bit-wise AND

An Invariant of Microsoft Windows Device Drivers

Example of a Program Verifier



```
  for (i = 0; i < 16; i++)
    x[i+32] = state[i] ^ block[i];

  /* Encrypt block (18 rounds).
   */
  t = 0;
  for (i = 0; i < 18; i++) {
    for (j = 0; j < 48; j++)
      t = x[j] ^= PI_SUBST[t];
    t = (t + i) & 0xff;
  }
```

	File	Property	Description	Expression
R	md2_bounds.c	bounds	array `x' upper bound	32 + i < 48
✓	md2_bounds.c	array bound	dereference failure: array `state' lower bound	!(i < 0) || !(c::md2_bounds::MD2Tr
✓	md2_bounds.c	array bound	dereference failure: array `state' upper bound	!(c::md2_bounds::MD2Transform::
R	md2_bounds.c	array bound	dereference failure: array `block' lower bound	!(i < 0) || !(c::md2_bounds::MD2Tr
R	md2_bounds.c	array bound	dereference failure: array `block' upper bound	!(c::md2_bounds::MD2Transform::
W	md2_bounds.c	bounds	array `x' upper bound	TRUE
W	md2_bounds.c	bounds	array `PI_SUBST' upper bound	t < 256
W	md2_bounds.c	bounds	array `x' upper bound	TRUE
W	md2_bounds.c	array bound	dereference failure: array `block' lower bound	!(i < 0) || !(c::md2_bounds::MD2Tr
W	md2_bounds.c	array bound	dereference failure: array `block' upper bound	!(c::md2_bounds::MD2Transform::
W	md2_bounds.c	bounds	array `PI_SUBST' upper bound	(t ^ (unsigned int)(*(i + block))) <

```
Running Cadence SMV: smv -force -sift
Cadence SMV produced counterexample
Simulating abstract transitions of counterexample on concrete program
Spurious transition found
Trace is spurious
Refining transition
*** CEGAR Loop Iteration 6
Running Cadence SMV: smv -force -sift
```

What?

- Research on software quality is very broad

- We focus on techniques that:

  1. prove a *guarantee*, in theory and practice.
  2. are highly *automated* and scale reasonably well.

- We do not aim at a full specification

  → Do 'absence of specific bugs'

What?

Therefore won’t talk about:

- ✘ random testing and automatic test vector generation (usually not complete)
- ✘ Unit testing (not automatic)
- ✘ Refinement techniques
- ✘ Tools that require annotation (ESC/Java etc.)
- ✘ State enumeration (incomplete)

What?

> *"Things like even software verification, this has been the Holy Grail of computer science for many decades, but now in some very key areas, for example, driver verification we're building tools that can do* ***actual proof*** *about the software and how it works in order to guarantee the reliability."*

Bill Gates, April 18, 2002
Keynote address at WinHec 2002

What?

"*One of the least visible ways that Microsoft Research contributed to Vista, but something I like to talk about, is the work we did on what's called the Static Driver Verifier. People who develop device drivers for Vista can verify the properties of their drivers before they ever even attempt to test that. What's great about this technology is there is no testing involved. For the properties that it is proving, they are either true or false.*
You don't have to ask yourself
"Did I come up with a good test case or not?""

Rick Rashid, Microsoft Research chief
father of CMU's Mach Operating System (Mac OS X)
news.cnet.com interview, 2008

What?

> “*This change is going to have* **dramatic impact** *over the next five to 10 years, as we begin to bring these proof tools to bear on larger and larger problems in the software space. We are already doing research on saying, “How would you create an operating system environment from scratch, knowing that you have this kind of proof technology available?”*

Rick Rashid
news.cnet.com interview, 2008

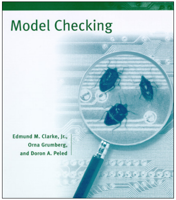
Static Analysis

Basic Idea

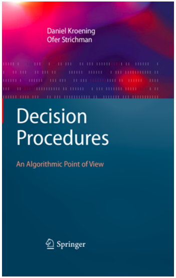
Efficiently compute approximate but *sound* answers.

- Found in compilers for decades
- “Approximate but sound”: e.g., compute superset of values
- Problem becomes decidable

Reading



Model Checking
Clarke/Peled/Grumberg



Decision Procedures
Kroening/Strichman

Part I: Basic principles

1. Propositional SAT
2. Bit-level circuit verification with SAT

3. Word-level modelling (Verilog, VHDL, SystemC, C/C++)
4. Word-level reasoning with SMT-AUFBV
5. Word-level verification: BMC, k-induction, interpolation

6. Outlook

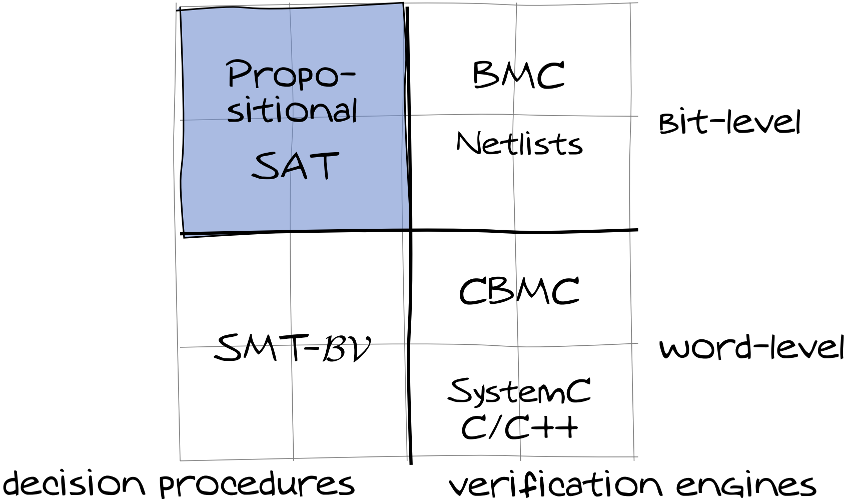
	decision procedures	verification engines
Bit-level	Propositional SAT	BMC Netlists
word-level	SMT-$\mathcal{BV}$	CBMC SystemC C/C++

Part II: Verification methodologies for application in practice

1. Requirements Analysis Case-Study
   - automotive
   - state charts

2. Verifying an RTL HW IP block against a C specification

3. Microprocessor case-study



	decision procedures	verification engines	
	Propo- sitional SAT	BMC Netlists	Bit-level
	SMT-$\mathcal{BV}$	CBMC SystemC C/C++	word-level

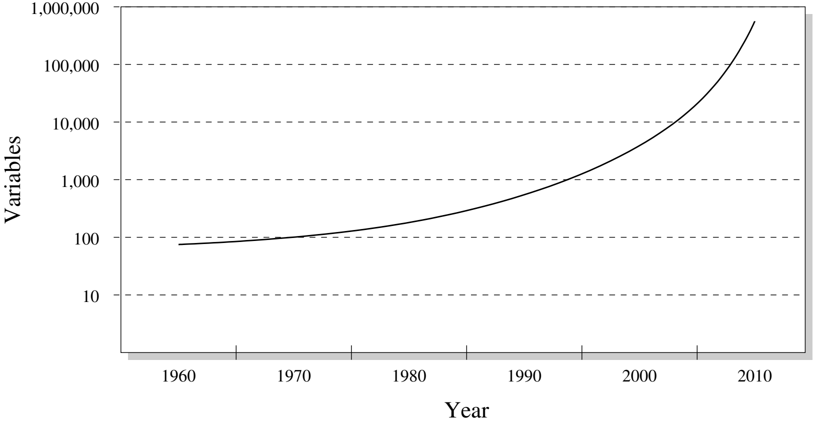
SAT

SAT (Satisfiability) the classical NP-complete problem:

Given a propositional formula f over n propositional variables $V = \{x, y, \ldots\}$.

Is there are an assignment $\sigma : V \to \{0, 1\}$ with $\sigma(f) = 1$?

Motivation


1,000,000
100,000
10,000
1,000
100
10
Variables
1960
1970
1980
1990
2000
2010
Year

Conjunctive Normal Form

Definition (Conjunctive Normal Form)

A formula in *Conjunctive Normal Form* (CNF) is a conjunction of clauses

$$C_1 \wedge C_2 \wedge \ldots \wedge C_n$$

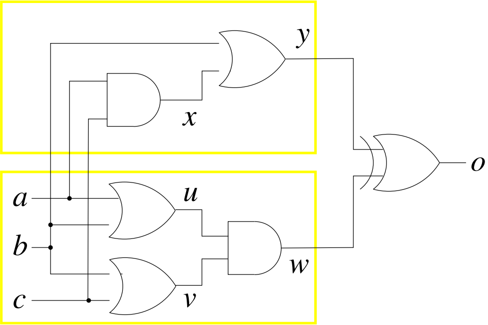
each clause C is a disjunction of literals

$$C = L_1 \vee \ldots \vee L_m$$

and each literal is either a plain variable x or a negated variable $\overline{x}$.

Example $(a \vee b \vee c) \wedge (\overline{a} \vee \overline{b}) \wedge (\overline{a} \vee \overline{c})$

Tseitin Transformation: Circuit to CNF



$$
\begin{aligned}
&{\color{red}o} \;\wedge\\
(x \;\leftrightarrow\;& a \wedge c) \wedge\\
(y \;\leftrightarrow\;& b \vee x) \wedge\\
(u \;\leftrightarrow\;& a \vee b) \wedge\\
(v \;\leftrightarrow\;& b \vee c) \wedge\\
(w \;\leftrightarrow\;& u \wedge v) \wedge\\
(o \;\leftrightarrow\;& y \oplus w)
\end{aligned}
$$

$${\color{red}o} \wedge (x \rightarrow a) \wedge (x \rightarrow c) \wedge (x \leftarrow a \wedge c) \wedge \;\ldots$$

$${\color{red}o} \wedge (\overline{x} \vee a) \wedge (\overline{x} \vee c) \wedge (x \vee \overline{a} \vee \overline{c}) \wedge \;\ldots$$

Algorithmic Description of Tseitin Transformation

Tseitin Transformation

1. For each non-input signal s: generate a new variable x_s
2. For each gate: produce input / output constraints as clauses
3. Collect all constraints in a big conjunction

Algorithmic Description of Tseitin Transformation

- The transformation is satisfiability-preserving:
  the result is satisfiable iff and only the original formula is satisfiable

- You an get a satisfying assignment for original formula by projecting the satisfying assignment onto the original variables

- Not equivalent in the classical sense to original formula:
  it has new variables

Tseitin Transformation: Input / Output Constraints

Negation:

$$\begin{aligned} x \leftrightarrow \overline{y} &\Leftrightarrow (x \rightarrow \overline{y}) \wedge (\overline{y} \rightarrow x) \\ &\Leftrightarrow (\overline{x} \vee \overline{y}) \wedge (y \vee x) \end{aligned}$$

Disjunction:

$$\begin{aligned} x \leftrightarrow (y \vee z) &\Leftrightarrow (y \rightarrow x) \wedge (z \rightarrow x) \wedge (x \rightarrow (y \vee z)) \\ &\Leftrightarrow (\overline{y} \vee x) \wedge (\overline{z} \vee x) \wedge (\overline{x} \vee y \vee z) \end{aligned}$$

Conjunction:

$$\begin{aligned} x \leftrightarrow (y \wedge z) &\Leftrightarrow (x \rightarrow y) \wedge (x \rightarrow z) \wedge ((y \wedge z) \rightarrow x) \\ &\Leftrightarrow (\overline{x} \vee y) \wedge (\overline{x} \vee z) \wedge (\overline{(y \wedge z)} \vee x) \\ &\Leftrightarrow (\overline{x} \vee y) \wedge (\overline{x} \vee z) \wedge (\overline{y} \vee \overline{z} \vee x) \end{aligned}$$

Equivalence:

$$\begin{aligned} x \leftrightarrow (y \leftrightarrow z) &\Leftrightarrow (x \rightarrow (y \leftrightarrow z)) \wedge ((y \leftrightarrow z) \rightarrow x) \\ &\Leftrightarrow (x \rightarrow ((y \rightarrow z) \wedge (z \rightarrow y)) \wedge ((y \leftrightarrow z) \rightarrow x) \\ &\Leftrightarrow (x \rightarrow (y \rightarrow z)) \wedge (x \rightarrow (z \rightarrow y)) \wedge ((y \leftrightarrow z) \rightarrow x) \\ &\Leftrightarrow (\overline{x} \vee \overline{y} \vee z) \wedge (\overline{x} \vee \overline{z} \vee y) \wedge ((y \leftrightarrow z) \rightarrow x) \\ &\Leftrightarrow (\overline{x} \vee \overline{y} \vee z) \wedge (\overline{x} \vee \overline{z} \vee y) \wedge (((y \wedge z) \vee (\overline{y} \wedge \overline{z})) \rightarrow x) \\ &\Leftrightarrow (\overline{x} \vee \overline{y} \vee z) \wedge (\overline{x} \vee \overline{z} \vee y) \wedge ((y \wedge z) \rightarrow x) \wedge ((\overline{y} \wedge \overline{z}) \rightarrow x) \\ &\Leftrightarrow (\overline{x} \vee \overline{y} \vee z) \wedge (\overline{x} \vee \overline{z} \vee y) \wedge (\overline{y} \vee \overline{z} \vee x) \wedge (y \vee z \vee x) \end{aligned}$$

Optimizations for the Tseitin Transformation

- Goal is smaller CNF (less variables, less clauses)
- Extract multi argument operands
(removes variables for intermediate nodes)
- NNF: half of AND, OR node constraints may be removed due to monotonicity
- use *sharing*

Example SAT: Circuit Equivalence

formula:

$$
\begin{array}{rcr}
 & & o \;\wedge \\
(x & \leftrightarrow & a \wedge c) \;\wedge \\
(y & \leftrightarrow & b \vee x) \;\wedge \\
(u & \leftrightarrow & a \vee b) \;\wedge \\
(v & \leftrightarrow & b \vee c) \;\wedge \\
(w & \leftrightarrow & u \wedge v) \;\wedge \\
(o & \leftrightarrow & y \oplus w)
\end{array}
$$

number assignment:

variable	number
o	1
a	2
c	3
x	4
b	5
y	6
u	7
v	8
w	9

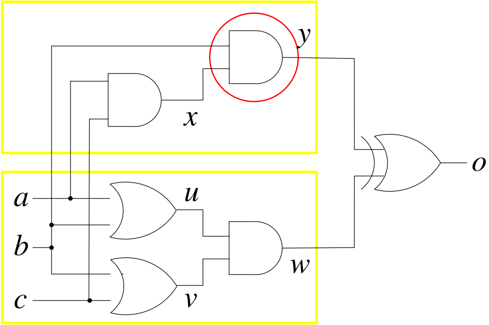
Simply in order of occurrence.

Example SAT: Circuit Equivalence

formula	clauses	DIMACS
o	o	`1 0`
$x \leftrightarrow a \wedge c$	$a \vee \overline{x}$	`2 -4 0`
	$c \vee \overline{x}$	`3 -4 0`
	$\overline{a} \vee \overline{c} \vee x$	`-2 -3 4 0`
$y \leftrightarrow b \vee x$	$\overline{x} \vee y$	`-4 6 0`
	$\overline{b} \vee y$	`-5 6 0`
	$x \vee b \vee \overline{y}$	`4 5 -6 0`
$u \leftrightarrow a \vee b$	$\overline{a} \vee u$	`-2 7 0`
	$\overline{b} \vee u$	`-5 7 0`
	$a \vee b \vee \overline{u}$	`2 5 -7 0`
$v \leftrightarrow b \vee c$	$\overline{b} \vee v$	`-5 8 0`
	$\overline{c} \vee v$	`-3 8 0`
	$b \vee c \vee \overline{v}$	`5 3 -8 0`
$w \leftrightarrow u \wedge v$	$u \vee \overline{w}$	`7 -9 0`
$\ldots$		

Example SAT: Circuit Equivalence

Let’s change the circuit!



$$\begin{aligned} & o \wedge \\ (x &\leftrightarrow a \wedge c) \wedge \\ (y &\leftrightarrow b \wedge x) \wedge \\ (u &\leftrightarrow a \vee b) \wedge \\ (v &\leftrightarrow b \vee c) \wedge \\ (w &\leftrightarrow u \wedge v) \wedge \\ (o &\leftrightarrow y \oplus w) \end{aligned}$$

Is the CNF satisfiable?

Example SAT: Circuit Equivalence

- Output of the SAT solver:

```
SATISFIABLE
1 2 3 4 -5 -6 7 8 9
```

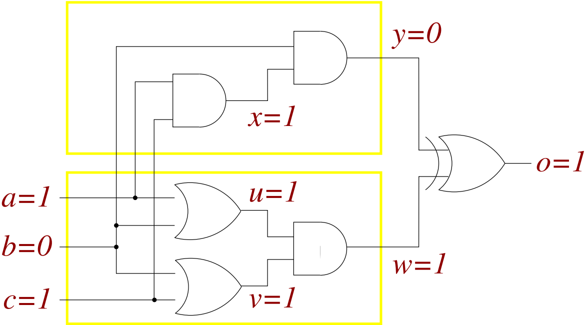
- Values of the variables:

variable	number	value
o	1	1
a	2	1
c	3	1
x	4	1
b	5	0
y	6	0
u	7	1
v	8	1
w	9	1

- Caveat: there is more than one solution

Example SAT: Circuit Equivalence

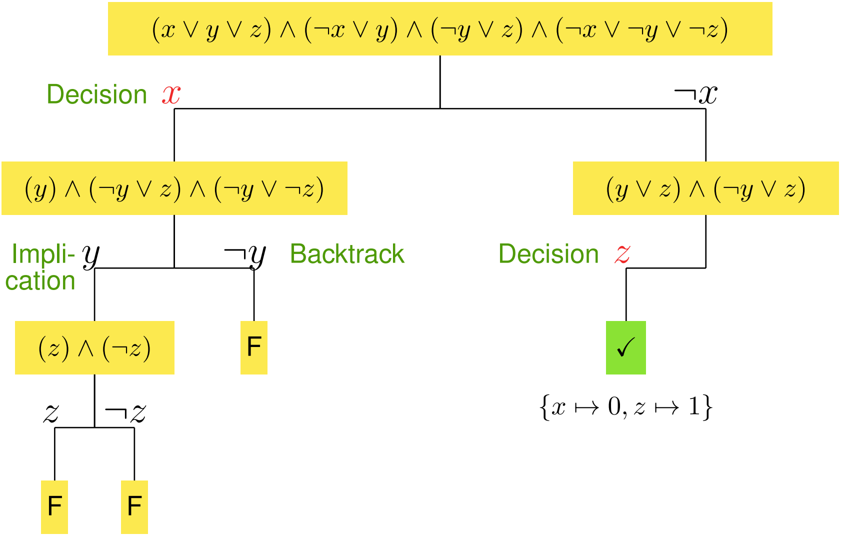
Satisfying assignment mapped to the circuit:



variable	value
o	1
a	1
c	1
x	1
b	0
y	0
u	1
v	1
w	1

Binary Search

Formula:



$(x \vee y \vee z) \wedge (\neg x \vee y) \wedge (\neg y \vee z) \wedge (\neg x \vee \neg y \vee \neg z)$

Decision x

$\neg x$

$(y) \wedge (\neg y \vee z) \wedge (\neg y \vee \neg z)$

$(y \vee z) \wedge (\neg y \vee z)$

Impli- cation y

$\neg y$

Backtrack

Decision z

$(z) \wedge (\neg z)$

F

✓

$\{x \mapsto 0, z \mapsto 1\}$

z

$\neg z$

F

F

Notation

Given the partial assignment

$$\{x_1 \mapsto 1,\ x_2 \mapsto 0,\ x_4 \mapsto 1\}$$

$(x_1 \lor x_3 \lor \neg x_4)$ is satisfied

$(\neg x_1 \lor x_2)$ is conflicting

$(\neg x_1 \lor \neg x_4 \lor x_3)$ is unit

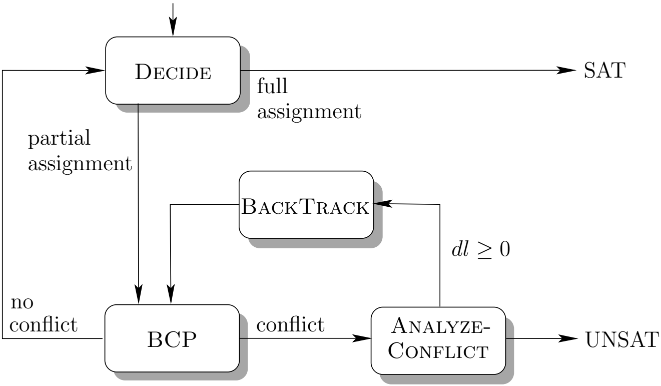
$(\neg x_1 \lor x_3 \lor x_5)$ is unresolved.

Basic DPLL

```
function DPLL
    if BCP() = ‘conflict’ then return ‘Unsatisfiable’;
    while (TRUE) do
        if ¬DECIDE() then  return ‘Satisfiable’;
        else
            while (BCP() = ‘conflict’) do
                $backtrack\text{-}level$ := ANALYZE-CONFLICT();
                if $backtrack\text{-}level < 0$ then
                    return ‘Unsatisfiable’;
                else
                    BACKTRACK($backtrack\text{-}level$);
```

- DECIDE: Choose next variable and value
- BCP: Propagate implications of unit clauses
- ANALYZE-CONFLICT: Determine backtracking level

Basic DPLL


DECIDE
SAT
full assignment
partial assignment
BACKTRACK
$dl \geq 0$
no conflict
BCP
conflict
ANALYZE-CONFLICT
UNSAT

Notation

- We organize the search in form of a *decision tree*
- Each node corresponds to a decision
  (no implied assignments in the tree)

- Def.: the depth of the node is the decision level
- $x@d$ means that x is set to 1 at level d
- $\neg x@d$ means that x is set to 0 at level d

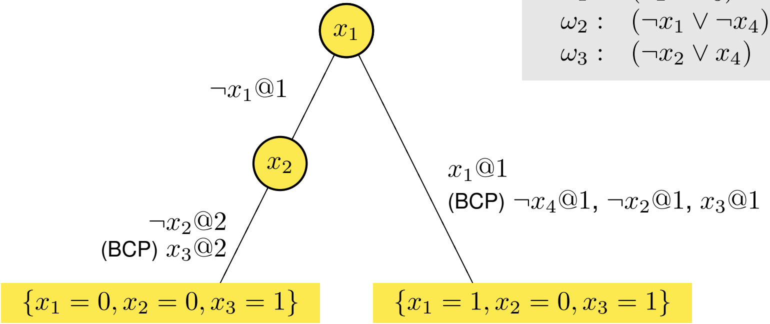
Example I

Formula

$\omega_1 : \quad (x_2 \vee x_3)$

$\omega_2 : \quad (\neg x_1 \vee \neg x_4)$

$\omega_3 : \quad (\neg x_2 \vee x_4)$



x_1
$\neg x_1 @ 1$
x_2
$\neg x_2 @ 2$
(BCP) $x_3 @ 2$
$x_1 @ 1$
(BCP) $\neg x_4 @ 1$, $\neg x_2 @ 1$, $x_3 @ 1$
$\{x_1 = 0, x_2 = 0, x_3 = 1\}$
$\{x_1 = 1, x_2 = 0, x_3 = 1\}$

No backtracking needed for this example,
regardless of the decision!

Example II

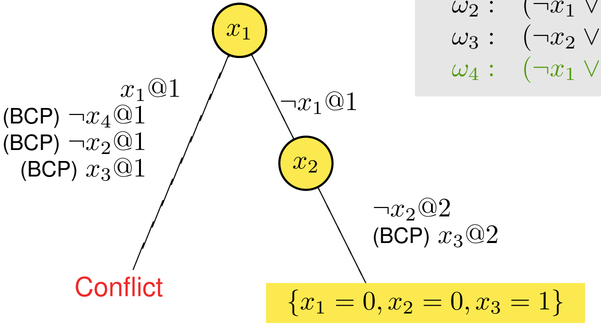
Formula

$\omega_1 : \quad (x_2 \lor x_3)$

$\omega_2 : \quad (\neg x_1 \lor \neg x_4)$

$\omega_3 : \quad (\neg x_2 \lor x_4)$

$\omega_4 : \quad (\neg x_1 \lor x_2 \lor \neg x_3)$



Decision Heuristics: DLIS

DLIS (Dynamic Largest Individual Sum)

choose the assignment that increases the number of satisfied clauses the most

- For every literal l, compute the number of unresolved clauses $C(l)$ that contain l
- This is the same as

$$C(l) = \sum_{l \in \omega, \omega \in \varphi} 1$$

- Make decision l that maximizes $C(l)$

Decision Heuristics: JW

Jeroslow-Wang method

For every literal l, compute:

$$J(l) = \sum_{l \in \omega, \omega \in \varphi} 2^{-|\omega|}$$

- $|\omega|$ is the length of the clause (count the literals)
- Make decision l that maximizes $J(l)$

- This gives exponentially higher weight to literals in shorter clauses
- Can be dynamic (only for unresolved clauses) or static ($J(l)$ computed upfront)

- We will see other (more advanced) decision heuristics soon.
- These heuristics are integrated with a mechanism called learning with conflict clauses, which we discuss next.

Implication Graphs

The implication graph tracks how assignments are implied.

Definition (Implication graph)

An *implication graph* is a labeled directed acyclic graph $G = (V, E)$ where

- V: literals of the current partial assignment.
  Labeled with the literal and the decision level.
- E: labeled with the clause that caused the implication.
- Can also contain a single conflict node labeled with κ and incoming edges labeled with some conflicting clause.

A Small Implication Graph Example

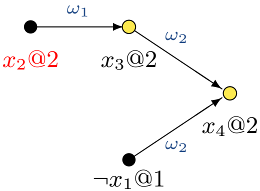
Current truth assignment: $\{\neg x_1 @1\}$

Decision: $x_2 @2$

Clauses

$\omega_1 = (\quad \neg x_2 \vee \; x_3 \quad)$

$\omega_2 = (x_1 \vee \quad \neg x_3 \vee x_4)$



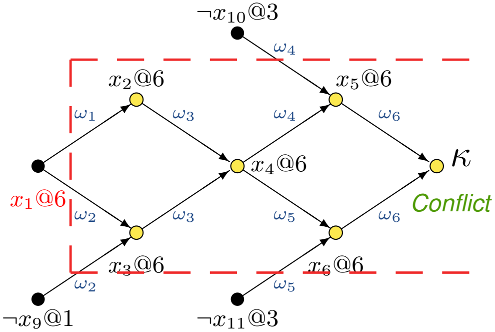
Implication Graphs and Learning

Current truth assignment: $\{\neg x_9@1,\ \neg x_{10}@3,\ \neg x_{11}@3,\ x_{12}@2,\ x_{13}@2\}$

Decision: $x_1@6$

Clauses

$$
\begin{array}{lcl}
\omega_1 & = & (\neg x_1 \vee x_2) \\
\omega_2 & = & (\neg x_1 \vee x_3 \vee x_9) \\
\omega_3 & = & (\neg x_2 \vee \neg x_3 \vee x_4) \\
\omega_4 & = & (\neg x_4 \vee x_5 \vee x_{10}) \\
\omega_5 & = & (\neg x_4 \vee x_6 \vee x_{11}) \\
\omega_6 & = & (\neg x_5 \vee \neg x_6) \\
\omega_7 & = & (x_1 \vee x_7 \vee \neg x_{12}) \\
\omega_8 & = & (x_1 \vee x_8) \\
\omega_9 & = & (\neg x_7 \vee \neg x_8 \vee \neg x_{13}) \\
\omega_{10} & = & (\neg x_1 \vee x_9 \vee x_{11} \vee x_{10})
\end{array}
$$



We learn the *conflict clause*

$$\omega_{10} = (\neg x_1 \vee x_9 \vee x_{11} \vee x_{10})$$

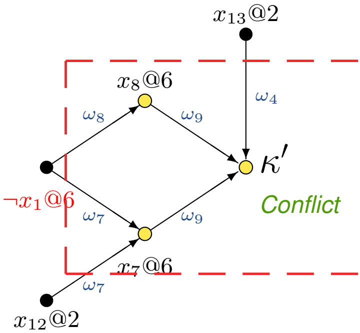
Backtracking

What now?

⇒ Flip the decision, i.e., $\neg x_1 @ 6$

Clauses

$$
\begin{aligned}
\omega_1 &= (\neg x_1 \lor x_2) \\
\omega_2 &= (\neg x_1 \lor x_3 \lor x_9) \\
\omega_3 &= (\neg x_2 \lor \neg x_3 \lor x_4) \\
\omega_4 &= (\neg x_4 \lor x_5 \lor x_{10}) \\
\omega_5 &= (\neg x_4 \lor x_6 \lor x_{11}) \\
\omega_6 &= (\neg x_5 \lor \neg x_6) \\
\omega_7 &= (x_1 \lor x_7 \lor \neg x_{12}) \\
\omega_8 &= (x_1 \lor x_8) \\
\omega_9 &= (\neg x_7 \lor \neg x_8 \lor \neg x_{13}) \\
\omega_{10} &= (\neg x_1 \lor x_9 \lor x_{11} \lor x_{10})
\end{aligned}
$$



Another conflict clause:

$$\omega_{11} = (\neg x_{13} \lor \neg x_{12} \lor x_1)$$

But where should we backtrack now? 5?

Non-Chronological Backtracking

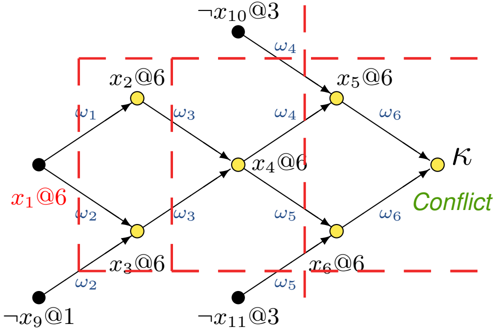
- So the rule is:

  backtrack to the *largest decision level* in the conflict clause.

- This works for both the initial conflict and any conflict after the flip.

More Conflict Clauses

- Def.: A *conflict clause* is any clause implied by the formula



$\neg x_1 \vee x_9 \vee x_{10} \vee x_{11}$

$\neg x_2 \vee \neg x_3 \vee x_{10} \vee x_{11}$

$\neg x_4 \vee x_{10} \vee x_{11}$

- Let L be a set of literals labeling nodes that form a cut in the implication graph, separating the conflict node from the roots

- Claim: $\bigvee_{l \in L} l$ is a conflict clause

More Conflict Clauses

- How many clauses should we add?
- If not all, then which ones?
  - The shorter ones?
  - Check their influence on the backtracking level ?
  - The “most influental”?
- Common answer:
  - *Asserting clauses*
  - *Unique implication points* (UIPs)

Conflict Clauses and Resolution

- *Binary Resolution* is a sound inference rule:

$$\frac{(a_1 \lor \ldots \lor a_n \lor \beta) \quad (b_1 \lor \ldots \lor b_m \lor \neg\beta)}{(a_1 \lor \ldots \lor a_n \lor b_1 \lor \ldots \lor b_m)}$$

  We say that we *resolve on* β

- Example:

$$\frac{(x_1 \lor x_2) \quad (\neg x_1 \lor x_3 \lor x_4)}{(x_2 \lor x_3 \lor x_4)}$$

- Also complete

Decision Heuristics: VSIDS

VSIDS (Variable State Independent Decaying Sum)

1. Each variable in each polarity has a counter initialized to 0.
2. When a clause is added, the counters are updated.
3. The unassigned variable with the highest counter is chosen.
4. Periodically, all the counters are divided by a constant.

$\Rightarrow$ variables appearing in recent conflicts
get higher priority

Decision Heuristics: VSIDS

- Keep a list of variables/polarities
- Updates only needed when adding a conflict clause
- Decisions are made in constant time (how?)

Decision Heuristics: VSIDS

VSIDS is a ‘quasi-static’ strategy:

- static as it does not depend on the current assignment
- dynamic as the weights change over time

VSIDS is called a *conflict-driven* decision strategy.

> *”...this strategy dramatically (i.e., an order of magnitude) improved performance...”*

Decision Heuristics: Berkmin

- Keep conflict clauses in a stack

- Choose the first unresolved clause in the stack
  If the stack is empty, use VSIDS

- Choose a variable + value from this clause according to some scoring (e.g., VSIDS)

- This gives *absolute priority* to conflicts.

	decision procedures	verification engines
Bit-level	Propositional SAT	BMC Netlists
word-level	SMT-$\mathcal{BV}$	CBMC SystemC C/C++

Bounded Model Checking

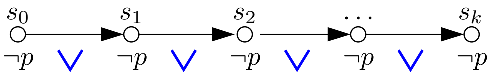
[BiereCimattiClarkeZhu99]

- Uses SAT for model checking
  - Historically not the first symbolic model checking approach
  - But scales better than original BDD-based techniques

- Mostly incomplete in practice
  - Focus on counterexample generation
  - Only counterexamples up to given length (the bound k) are searched

Bounded Model Checking for Safety

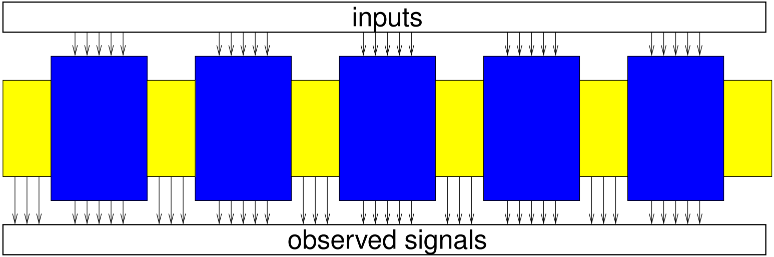
Checking safety property $\mathbf{G}p$ for a bound k as SAT problem:



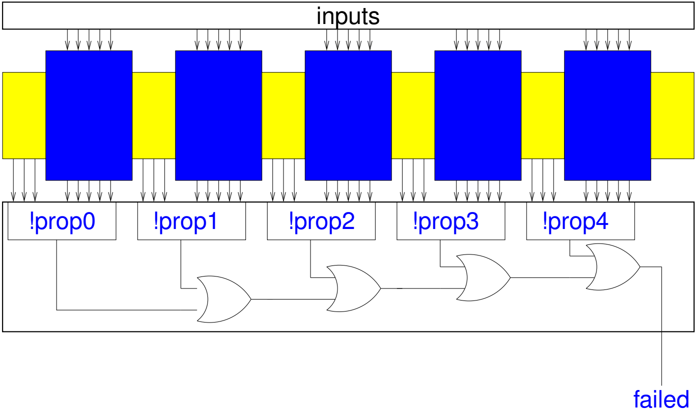
$$I(s_0) \;\wedge\; T(s_0, s_1) \;\wedge \ldots \wedge\; T(s_{k-1}, s_k) \;\wedge\; \bigvee_{i=0}^{k} \neg p(s_i)$$

Check occurrence of $\neg p$ in the first k states

Time Frame Expansion in HW

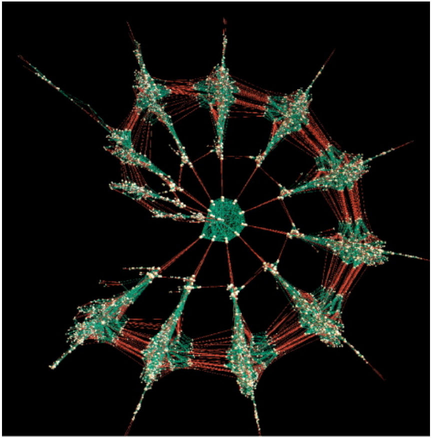


Bounded Model Checking Safety in HW



find inputs for which failed becomes true

Visualizing Bounded Model Checking

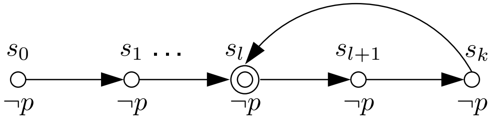


Nodes: variables,
edges: clauses
(binary clauses are red)

k = 12,
bounded cone-of-influence

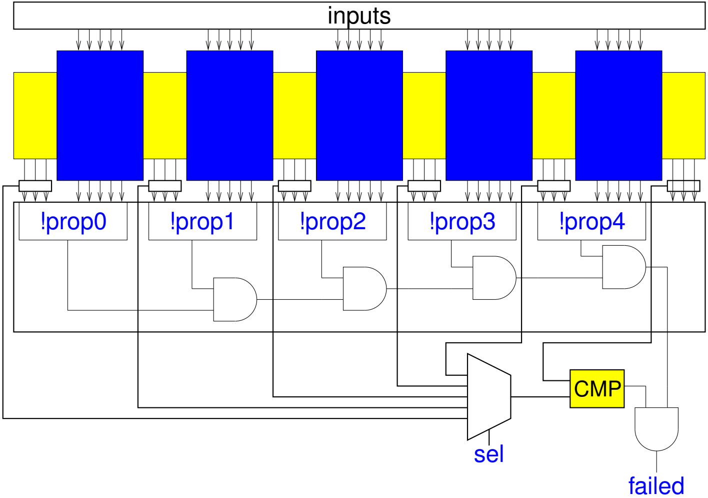
Bounded Model Checking for Liveness

Generic counterexample trace of length k for liveness $\mathbf{F}p$



$$I(s_0) \ \wedge \ T(s_0, s_1) \ \wedge \ldots \wedge \ T(s_k, s_{k+1}) \ \wedge \ \bigvee_{l=0}^{k} s_l = s_{k+1} \ \wedge \ \bigwedge_{i=0}^{k} \neg p(s_i)$$

Bounded Model Checking Liveness in HW



find inputs for which failed becomes true

Completeness in Bounded Model Checking

- Find bounds on the maximal length of counterexamples
  - also called **completeness threshold**
  - exact bounds are hard to find ⇒ approximations

- Induction
  - use inductive invariants

- Use SAT for quantifier elimination as with BDDs
  - then model checking becomes fixpoint calculation

Measuring Distances

Distance: length of shortest path between two states

$$\delta(s,t) \equiv \min\{n \mid \exists s_0,\ldots,s_n[s=s_0, t=s_n \text{ and } \bigwedge_{i=0}^{n-1} T(s_i,s_{i+1})]\}$$

(distance can be infinite if s and t are not connected)

Measuring Distances

Diameter: maximal distance between two connected states

$$d(T) \equiv \max\{\delta(s,t) \mid T^*(s,t)\}$$

(recall that T^* is the transitive reflexive closure of T).

Measuring Distances

Reachable Diameter: maximal distance between two reachable states (R)

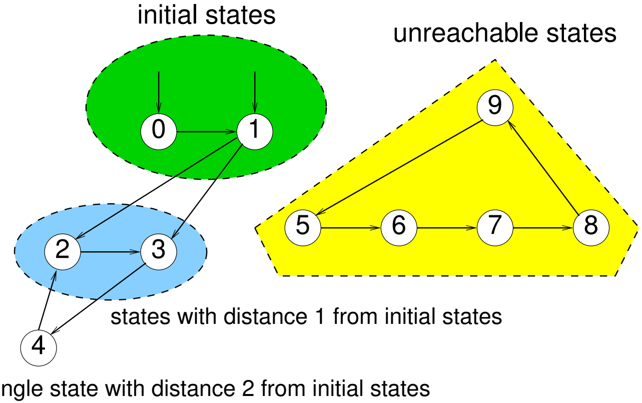
$$d(T) \equiv \max\{\delta(s,t) \mid T^*(s,t) \wedge R(s) \wedge R(t)\}$$

Initialized Diameter: the maximal distance from an initial state to a reachable state

$$\begin{array}{lll} r(T,I) \equiv \max\{\delta(s,t) & \mid & T^*(s,t) \text{ and } I(s) \text{ and} \\ & & \delta(s,t) \leq \delta(s',t) \text{ for all } s' \text{ with } I(s')\} \end{array}$$

(minimal number of steps to reach an arbitrary state in BFS; sometimes called *radius*)

Diameters Illustrated


initial states
unreachable states
0
1
2
3
4
5
6
7
8
9
states with distance 1 from initial states
single state with distance 2 from initial states

diameter 4, initialized diameter 2,
reachable diameter 3

Completeness Thresholds for Safety

- A bad state is reached in at most d_I steps from the initial states
- Thus, the (initialized/reachable) diameter is a completeness threshold for $\mathbf{G}p$
- Thus, for $\mathbf{G}p$, the max. k req. for BMC is the diameter
- If no counterexample of this length can be found the property holds

How to Determine the Diameter?

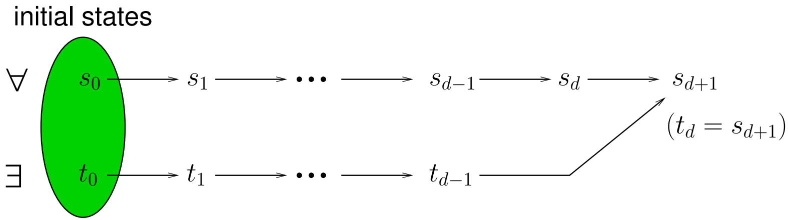
Reformulation:
The initialized diameter is the max. length d of a path leading from an initial state to a state t, such there is no other path from an initial state to t with length less than d.
Thus d is the minimal number that makes the following formula valid:

$$\forall s_0,\ldots,s_{d+1}[\,(I(s_0)\wedge\bigwedge_{i=0}^{d}T(s_i,s_{i+1}))\Rightarrow$$

$$\exists\, n\leq d\,[\,\exists t_0,\ldots,t_n[\,I(t_0)\wedge\bigwedge_{i=0}^{n-1}T(t_i,t_{i+1})\wedge t_n=s_{d+1}\,]\,]\,]$$

After replacing $\exists\, n\leq d\ldots$ by $\bigvee_{n=0}^{d}\ldots$ we get a **Quantified Boolean Formula** (QBF), which is hard to decide (PSPACE complete).

Visualization of Reformulation



(we allow t_{i+1} to be identical to t_i in the lower path)

Reoccurrence Radius/Diameter

- We cannot compute the diameter with SAT efficiently

- Overapproximation idea:
  - drop requirement that there is no shorter path
  - enforce *different* (no reoccurring) states on single path instead

Reoccurrence diameter:
length of the longest path without reoccurring states
(sometimes called *circumfence*)

Initialized reoccurrence diameter:
length of the longest initialized path without reoccurring states

Computing the Reoccurrence Diameter

Reformulation:
The reoccurrence diameter is the length of the longest path from initial states without reoccurring states (one may further assume that only the first state is an initial state)

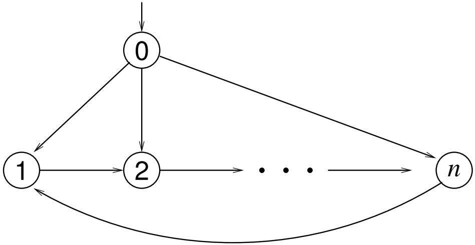
This is the minimal d that makes the following formula valid:

$$\forall s_0, \dots, s_{d+1} [\, (I(s_0) \wedge \bigwedge_{i=0}^{d} T(s_i, s_{i+1})) \;\Rightarrow \bigvee_{0 \leq i < j \leq d+1} s_i = s_j]$$

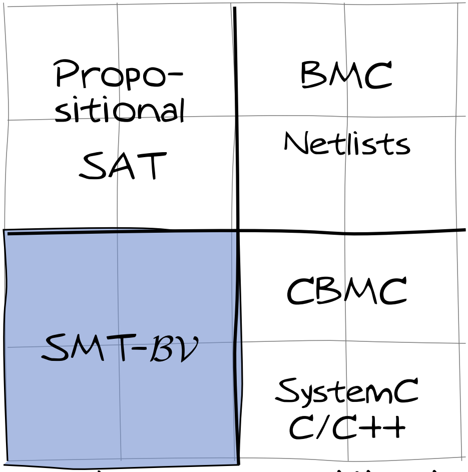
This is a propositional formula and can be checked by SAT!

(exercise: reoccurrence diameter is an upper bound for diameter)

Bad Example for Reoccurrence Radius



Initialized diameter 1,
initialized reoccurrence diameter n



	decision procedures	verification engines
Bit-level	Propositional SAT	BMC Netlists
word-level	SMT-$\mathcal{BV}$	CBMC SystemC C/C++

SMT-$\mathcal{BVAUF}$

What is SMT?
What is SMT-$\mathcal{BVAUF}$?

SMT = Satisfiability modulo theories

BVAUF = Bit-vectors and arrays and uninterpreted functions

Decision Procedures for System-Level Software

What kind of logic do we need for system-level software?

```
State { int created = 0; }

IoCreateDevice.exit {
  if ($return==STATUS_SUCCESS)
   created = 1;
}

IoDeleteDevice.exit { created = 0; }

fun_AddDevice.exit {
  if (created && (pdevobj->Flags & DO_DEVICE_INITIALIZING) != 0)
{
   abort "AddDevice routine failed to set "
         "~DO_DEVICE_INITIALIZING flag";
  }
}
```

Bit-wise AND

An Invariant of Microsoft Windows Device Drivers

Decision Procedures for System-Level Software

What kind of logic do we need for system-level software?

- We need bit-vector logic – with bit-wise operators, arithmetic overflow
- We want to scale to large programs – must verify large formulas
- Examples of program analysis tools that generate bit-vector formulas:
  - CBMC
  - SATABS
  - F-Soft (NEC)
  - SATURN (Stanford, Alex Aiken)
  - EXE (Stanford, Dawson Engler, David Dill)
  - Variants of those developed at IBM, Microsoft

Bit-Vector Logic: Syntax

$$
\begin{array}{rcl}
\mathit{formula} & : & \mathit{formula} \lor \mathit{formula} \mid \neg \mathit{formula} \mid \mathit{atom} \\
\mathit{atom} & : & \mathit{term}\ \mathit{rel}\ \mathit{term} \mid \mathit{Boolean\text{-}Identifier} \mid \mathit{term}[\,\mathit{constant}\,] \\
\mathit{rel} & : & = \ \mid \ < \\
\mathit{term} & : & \mathit{term}\ \mathit{op}\ \mathit{term} \mid \mathit{identifier} \mid \ \sim \mathit{term} \mid \mathit{constant} \mid \\
 & & \mathit{atom}\,?\,\mathit{term}\,{:}\,\mathit{term} \mid \\
 & & \mathit{term}[\,\mathit{constant} : \mathit{constant}\,] \mid \mathit{ext}(\,\mathit{term}\,) \\
\mathit{op} & : & + \mid - \mid \cdot \mid / \mid \ll \mid \gg \mid \& \mid \shortmid \mid \oplus \mid \circ
\end{array}
$$

- $\sim x$: bit-wise negation of x
- $\mathit{ext}(x)$: sign- or zero-extension of x
- $x \ll d$: left shift with distance d
- $x \circ y$: concatenation of x and y

Semantics

Danger!

$$(x - y > 0) \iff (x > y)$$

Valid over $\mathbb{R}/\mathbb{N}$, but not over the bit-vectors.

(Many compilers have this sort of bug)

Width and Encoding

- The meaning depends on the width and encoding of the variables.
- Typical encodings:
  - Binary encoding

$$\langle x \rangle_U := \sum_{i=0}^{l-1} a_i \cdot 2^i$$

  - Two's complement

$$\langle x \rangle_S := -2^{n-1} \cdot a_{n-1} + \sum_{i=0}^{l-2} a_i \cdot 2^i$$

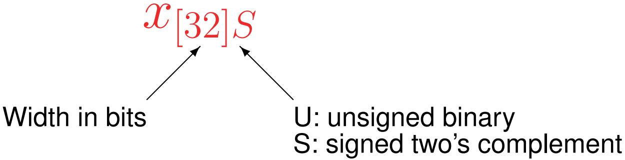
  - But maybe also fixed-point, floating-point, . . .

Examples

$$\begin{aligned}
\langle 11001000 \rangle_U &= 200 \\
\langle 11001000 \rangle_S &= -128 + 64 + 8 = -56 \\
\langle 01100100 \rangle_S &= 100
\end{aligned}$$

Width and Encoding

Notation to clarify width and encoding:



$x_{[32]S}$

Width in bits

U: unsigned binary
S: signed two's complement

Bit-vectors Made Formal

Definition (Bit-Vector)

A *bit-vector* is a vector of Boolean values with a given length l:

$$b : \{0, \ldots, l-1\} \longrightarrow \{0, 1\}$$

The value of bit number i of x is $x(i)$.

$$\underbrace{\boxed{b_{l-1}}\boxed{b_{l-2}}\cdots\cdots\boxed{b_2}\boxed{b_1}\boxed{b_0}}_{l \text{ bits}}$$

We also write x_i for $x(i)$.

Lambda-Notation for Bit-Vectors

λ expressions are functions without a name

Examples:

- The vector of length l that consists of zeros:

$$\lambda i \in \{0, \ldots, l-1\}.0$$

- A function that inverts (flips all bits in) a bit-vector:

$$bv\text{-}invert(x) := \lambda i \in \{0, \ldots, l-1\}.\neg x_i$$

- A bit-wise OR:

$$bv\text{-}or(x, y) := \lambda i \in \{0, \ldots, l-1\}.(x_i \vee y_i)$$

$\Longrightarrow$ we now have semantics for the bit-wise operators.

Example

$$(x_{[10]} \circ y_{[5]})[14] \iff x[9]$$

- This is translated as follows:

$$x[9] = x_9$$

$$(x \circ y) = \lambda i.(i < 5)?y_i : x_{i-5}$$

$$(x \circ y)[14] = (\lambda i.(i < 5)?y_i : x_{i-5})(14)$$

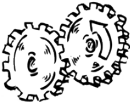
- Final result:

$$(\lambda i.(i < 5)?y_i : x_{i-5})(14) \iff x_9$$

Semantics for Arithmetic Expressions

What is the output of the following program?

```
unsigned char number = 200;
number = number + 100;
printf("Sum: %d\n", number);
```



On most architectures, this is 44!

```
     11001000   = 200
+    01100100   = 100
-----------------------
=    00101100   = 44
```

$\Longrightarrow$ Bit-vector arithmetic uses modular arithmetic!

Semantics for Arithmetic Expressions

Semantics for addition, subtraction:

$$\begin{aligned}
a_{[l]} +_U b_{[l]} = c_{[l]} &\iff \langle a\rangle_U + \langle b\rangle_U = \langle c\rangle_U \mod 2^l \\
a_{[l]} -_U b_{[l]} = c_{[l]} &\iff \langle a\rangle_U - \langle b\rangle_U = \langle c\rangle_U \mod 2^l \\
a_{[l]} +_S b_{[l]} = c_{[l]} &\iff \langle a\rangle_S + \langle b\rangle_S = \langle c\rangle_S \mod 2^l \\
a_{[l]} -_S b_{[l]} = c_{[l]} &\iff \langle a\rangle_S - \langle b\rangle_S = \langle c\rangle_S \mod 2^l
\end{aligned}$$

We can even mix the encodings:

$$a_{[l]U} +_U b_{[l]S} = c_{[l]U} \iff \langle a\rangle_U + \langle b\rangle_S = \langle c\rangle_U \mod 2^l$$

Semantics for Relational Operators

Semantics for $<$, $\leq$, $\geq$, and so on:

$$\begin{aligned} a_{[l]U} < b_{[l]U} &\iff \langle a\rangle_U < \langle b\rangle_U \\ a_{[l]S} < b_{[l]S} &\iff \langle a\rangle_S < \langle b\rangle_S \end{aligned}$$

Mixed encodings:

$$\begin{aligned} a_{[l]U} < b_{[l]S} &\iff \langle a\rangle_U < \langle b\rangle_S \\ a_{[l]S} < b_{[l]U} &\iff \langle a\rangle_S < \langle b\rangle_U \end{aligned}$$

Note that most compilers don't support comparisons with mixed encodings.

Complexity

- Satisfiability is undecidable for an unbounded width, even without arithmetic.

- It is NP-complete otherwise.

A Simple Decision Procedure

- Transform Bit-Vector Logic to Propositional Logic
- Most commonly used decision procedure
- Also called '*bit-blasting*'

Bit-Vector Flattening

1. Convert propositional part as before
2. Add a *Boolean variable for each bit* of each sub-expression (term)
3. Add *constraint* for each sub-expression

We denote the new Boolean variable for bit i of term t by $\mu(t)_i$.

Bit-vector Flattening

What constraints do we generate for a given term?

- This is easy for the bit-wise operators.

- Example for $a|_{[l]}b$:

$$\bigwedge_{i=0}^{l-1} (\mu(t)_i = (a_i \vee b_i))$$

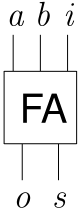
(read $x = y$ over bits as $x \iff y$)

- We can transform this into CNF using Tseitin's method.

Flattening Bit-Vector Arithmetic

How to flatten $a + b$?

$\longrightarrow$ we can build a *circuit* that adds them!



Full Adder

$$\begin{array}{lclcl} s & \equiv & (a + b + i\ ) \bmod 2 & \equiv & a \oplus b \oplus i \\ o & \equiv & (a + b + i\ ) \operatorname{div} 2 & \equiv & a \cdot b + a \cdot i + b \cdot i \end{array}$$

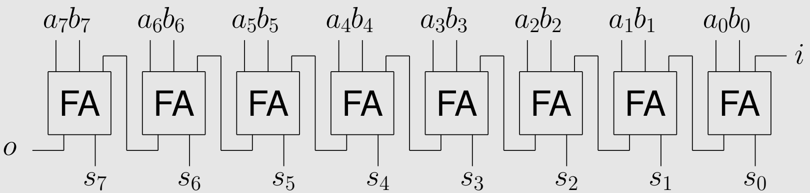
The full adder in CNF:

$$\begin{array}{l} (a \vee b \vee \neg o) \wedge (a \vee \neg b \vee i \vee \neg o) \wedge (a \vee \neg b \vee \neg i \vee o) \wedge \\ (\neg a \vee b \vee i \vee \neg o) \wedge (\neg a \vee b \vee \neg i \vee o) \wedge (\neg a \vee \neg b \vee o) \end{array}$$

Flattening Bit-Vector Arithmetic

Ok, this is good for one bit! How about more?

8-Bit ripple carry adder (RCA)



- Also called *carry chain adder*
- Adds l variables
- Adds $6 \cdot l$ clauses

Multipliers

- Multipliers result in very hard formulas

- Example:

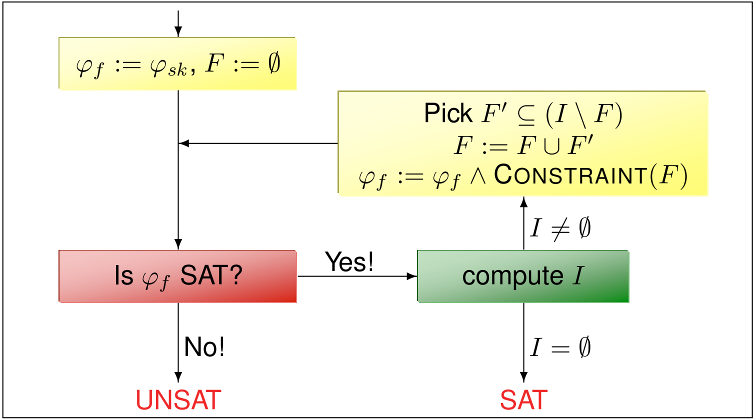
$$a \cdot b = c \wedge b \cdot a \neq c \wedge x < y \wedge x > y$$

  CNF: About 11000 variables, unsolvable for current SAT solvers

- Similar problems with division, modulo

- Q: Why is this hard?
- Q: How do we fix this?

Incremental Flattening



$\varphi_f := \varphi_{sk}, F := \emptyset$
Pick $F' \subseteq (I \setminus F)$
$F := F \cup F'$
$\varphi_f := \varphi_f \wedge$ CONSTRAINT(F)
$I \neq \emptyset$
Is φ_f SAT?
Yes!
compute I
No!
$I = \emptyset$
UNSAT
SAT

φ_{sk}: Boolean part of φ
F: set of terms that are in the encoding
I: set of terms that are inconsistent with the current assignment

Incremental Flattening

- Idea: add ’easy’ parts of the formula first
- Only add hard parts when needed
- φ_f only gets stronger – use an incremental SAT solver

Motivation

Arrays are an important data structure:

- “Native” implementation in most processor architectures
- Offered by most programming languages
- $O(1)$ index operation
  E.g., all data structures in Minisat are based on arrays
- Hardware: memories

Formalization

- Mapping from an *index type* to an *element type*

- T_I: index type
- T_E: element type
- $T_A = (T_I \longrightarrow T_E)$: array type

- Assumption: there are relations

  $$=_I \subseteq (T_I \times T_I) \quad \text{and} \quad =_E \subseteq (T_E \times T_E)$$

  The subscript is omitted if the type of the operands is clear.

- The theories used to reason about the indices and the elements are called *index theory* and *element theory*, respectively.

Basic Operations

Let $a \in T_A$ denote an array.

There are two basic operations on arrays:

1. *Reading*: $a[i]$ is the value of the element that has index i
2. *Writing*: the array a where element i has been replaced by e is denoted by $a\{i \longleftarrow e\}$

More About the Index Theory

What theory is suitable for the indices?

- Index logic should permit existential and universal quantification:
  - “$\textit{there exists an array element that is zero}$”
  - “$\textit{all elements of the array are greater than zero}$”
- Example: *Presburger arithmetic*, i.e., linear arithmetic over integers with quantification

n-dimensional arrays:
For $n \geq 2$, add $T_A(n-1)$ to the element type of $T_A(n)$.

A Very General Definition of Array Logic

Syntax defined by extending the syntactic rules for the index logic and the element logic

- $atom_I$: atom in the index logic
- $atom_E$: atom in the element logic
- $term_I$: term in the index logic
- $term_E$: term in the element logic

Syntax

$$\begin{array}{rcl} atom & : & atom_I \mid atom_E \mid \neg atom \mid atom \wedge atom \mid \\ & & \forall\, \textit{array-identifier}\,.\ atom \\ term_A & : & \textit{array-identifier} \mid term_A\{term_I \longleftarrow term_E\} \\ term_E & : & term_A\,[\,term_I\,] \end{array}$$

Equality between arrays a_1 and a_2: write as $\forall i.\ a_1[i] = a_2[i]$

Semantics

Main axiom:

Axiom (Read-over-write Axiom)

$$\forall a \in T_A.\ \forall e \in T_E.\ \forall i, j \in T_I.$$

$$a\{i \longleftarrow e\}[j] = \begin{cases} e & : \quad i = j \\ a[j] & : \quad \textit{otherwise}\ . \end{cases}$$

Program Verification Example I

```
a:   array 0..99 of integer;
i:   integer;

for i:=0 to 99 do
        /* $\forall x \in \mathbb{N}_0.\ x < i \Rightarrow \mathtt{a}[x] = 0$ */
        a[i]:=0;
        /* $\forall x \in \mathbb{N}_0.\ x \le i \Rightarrow \mathtt{a}[x] = 0$ */
done;
/* $\forall x \in \mathbb{N}_0.\ x \le 99 \Rightarrow \mathtt{a}[x] = 0$ */
```

Program Verification Example II

Main step of the correctness argument:
invariant in line 7 is maintained by the assignment in line 6

Verification condition:

$$\begin{array}{ll} & (\forall x \in \mathbb{N}_0.\ x < i \Rightarrow \mathtt{a}[x] = 0) \\ \wedge & \mathtt{a}' = \mathtt{a}\{i \longleftarrow 0\} \\ \Rightarrow & (\forall x \in \mathbb{N}_0.\ x \leq i \Rightarrow \mathtt{a}'[x] = 0) \end{array}$$

Decidability

Q: Is this logic decidable?

A: No, even if the combination of the index logic and the element logic is decidable

Arrays as Uninterpreted Functions

Fragment: no quantification over arrays

Arrays are functions! (From indices to elements)

Idea: use procedures for uninterpreted functions!

Example

$$(i = j \land a[j] = \texttt{'z'}) \Rightarrow a[i] = \texttt{'z'}$$

$\texttt{'z'}$: read as an integer number

F_a: uninterpreted function introduced for the array a:

$$(i = j \land F_a(j) = \texttt{'z'}) \Rightarrow F_a(i) = \texttt{'z'}$$

Example

$$(i = j \wedge F_a(j) = \texttt{'z'}) \Rightarrow F_a(i) = \texttt{'z'}$$

Apply Bryant's reduction:

$$(i = j \wedge F_1^* = \texttt{'z'}) \Rightarrow F_2^* = \texttt{'z'}$$

where

$$F_1^* = f_1 \quad \text{and} \quad F_2^* = \begin{cases} f_1 & : \quad i = j \\ f_2 & : \quad \text{otherwise} \end{cases}$$

Prove this using a decision procedure for equality logic.

Array Updates

What about $a\{i \longleftarrow e\}$?

1. Replace $a\{i \longleftarrow e\}$ by a fresh variable a' of array type

2. Add two constraints:
   a) $a'[i] = e$ for the value that is written,
   b) $\forall j \neq i.\ a'[j] = a[j]$ for the values that are unchanged.

   Compare to the read-over-write axiom!

This is called the *write rule*.

Array Updates: Example I

Transform

$$a\{i \longleftarrow e\}[i] \geq e$$

into:

$$a'[i] = e \Rightarrow a'[i] \geq e$$

Array Updates: Example II

Transform

$$a[0] = 10 \Rightarrow a\{1 \longleftarrow 20\}[0] = 10$$

into:

$$(a[0] = 10 \wedge a'[1] = 20 \wedge (\forall j \neq 1.\ a'[j] = a[j])) \Rightarrow a'[0] = 10$$

and then replace a, a':

$$(F_a(0) = 10 \wedge F_{a'}(1) = 20 \wedge (\forall j \neq 1.\ F_{a'}(j) = F_a(j))) \Rightarrow F_{a'}(0) = 10$$

Q: Is this decidable in general?
Say Presburger plus uninterpreted functions?

Array Properties

Now: restricted class of array logic formulas in order to obtain decidability.
We consider formulas that are Boolean combinations of **array properties**.

Definition (array property)

A formula is an *array property* iff if it is of the form

$$\forall i_1, \ldots, i_k \in T_I .\ \phi_I(i_1, \ldots, i_k) \Rightarrow \phi_V(i_1, \ldots, i_k)\ ,$$

and satisfies the following conditions:

1. The predicate ϕ_I must be an *index guard*.
2. The index variables $i_1, \ldots, i_k$ can only be used in array read expressions of the form $a[i_j]$.

The predicate ϕ_V is called the *value constraint*.

Index Guards

Definition (Index Guard)

A formula is an *index guard* iff if follows the grammar

$$\begin{array}{rcl} iguard & : & iguard \wedge iguard \,|\, iguard \vee iguard \,| \\ & & iterm \leq iterm \,|\, iterm = iterm \\ iterm & : & i_1 \,|\, \ldots \,|\, i_k \,|\, term \\ term & : & \textit{integer-constant} \,| \\ & & \textit{integer-constant} \cdot \textit{index-identifier} \,| \\ & & term + term \end{array}$$

The “*index-identifier*” used in “$term$” must not be one of $i_1, \ldots, i_k$.

Array Properties: Example

The extensionality rule defines the equality of two arrays a_1 and a_2 as element-wise equality. Extensionality is an array property:

$$\forall i.\ a_1[i] = a_2[i]$$

How about the array update?

$$\mathtt{a}' = \mathtt{a}\{i \longleftarrow 0\}$$

Is this an array property as well?

Array Properties: Array Update

An array update expression can be replaced by adding two constraints:

$$a'[i] = 0 \quad \wedge \quad \forall j \neq i.\, a'[j] = a[j]$$

The first conjunct is obviously an array property.

The second conjunct can be rewritten as

$$\forall j.\, (j \leq i - 1 \vee i + 1 \leq j) \Rightarrow a'[j] = a[j]$$

Algorithm

Input: Array property formula ϕ_A in NNF
Output: Formula ϕ_{UF}

1. Apply the write rule to remove all array updates from ϕ_A.
2. Replace all existential quantifications of the form $\exists i \in T_I.\ P(i)$ by $P(j)$, where j is a fresh variable.
3. Replace all universal quantifications of the form $\forall i \in T_I.\ P(i)$ by

$$\bigwedge_{i \in \mathcal{I}(\phi)} P(i)\ .$$

4. Replace the array read operators by uninterpreted functions and obtain ϕ_{UF};
5. **return** ϕ_{UF};

The Set I

$\mathcal{I}(\phi)$ denotes the index expressions that i might possibly be equal to.

Theorem: This set contains the following elements:

1. All expressions used as an array index in ϕ that are not quantified variables.
2. All expressions used inside index guards in ϕ that are not quantified variables.
3. If ϕ contains none of the above, $\mathcal{I}(\phi)$ is $\{0\}$ in order to obtain a nonempty set of index expressions.

Example

We prove validity of

$$\begin{array}{rl} & (\forall x \in \mathbb{N}_0.\ x < i \Rightarrow \mathtt{a}[x] = 0) \\ \wedge & \mathtt{a}' = \mathtt{a}\{i \longleftarrow 0\} \\ \Rightarrow & (\forall x \in \mathbb{N}_0.\ x \leq i \Rightarrow \mathtt{a}'[x] = 0)\ . \end{array}$$

That is, we check satisfiability of

$$\begin{array}{rl} & (\forall x \in \mathbb{N}_0.\ x < i \Rightarrow \mathtt{a}[x] = 0) \\ \wedge & \mathtt{a}' = \mathtt{a}\{i \longleftarrow 0\} \\ \wedge & (\exists x \in \mathbb{N}_0.\ x \leq i \wedge \mathtt{a}'[x] \neq 0)\ . \end{array}$$

Example

Apply write rule:

$$\begin{array}{ll} & (\forall x \in \mathbb{N}_0.\ x < i \Rightarrow \mathtt{a}[x] = 0) \\ \wedge & \mathtt{a}'[i] = 0 \wedge \forall j \neq i.\ \mathtt{a}'[j] = \mathtt{a}[j] \\ \wedge & (\exists x \in \mathbb{N}_0.\ x \leq i \wedge \mathtt{a}'[x] \neq 0)\ . \end{array}$$

Instantiate existential quantifier with a new variable $z \in \mathbb{N}_0$:

$$\begin{array}{ll} & (\forall x \in \mathbb{N}_0.\ x < i \Rightarrow \mathtt{a}[x] = 0) \\ \wedge & \mathtt{a}'[i] = 0 \wedge \forall j \neq i.\ \mathtt{a}'[j] = \mathtt{a}[j] \\ \wedge & z \leq i \wedge \mathtt{a}'[z] \neq 0\ . \end{array}$$

Example

The set $\mathcal{I}$ for our example is $\{i, z\}$.
Replace the two universal quantifications as follows:

$$\begin{array}{ll} & (i < i \Rightarrow \mathsf{a}[i] = 0) \wedge (z < i \Rightarrow \mathsf{a}[z] = 0) \\ \wedge & \mathsf{a}'[i] = 0 \wedge (i \neq i \Rightarrow \mathsf{a}'[i] = \mathsf{a}[i]) \wedge (z \neq i \Rightarrow \mathsf{a}'[z] = \mathsf{a}[z]) \\ \wedge & z \leq i \wedge \mathsf{a}'[z] \neq 0 \; . \end{array}$$

Remove the trivially satisfied conjuncts to obtain

$$\begin{array}{ll} & (z < i \Rightarrow \mathsf{a}[z] = 0) \\ \wedge & \mathsf{a}'[i] = 0 \wedge (z \neq i \Rightarrow \mathsf{a}'[z] = \mathsf{a}[z]) \\ \wedge & z \leq i \wedge \mathsf{a}'[z] \neq 0 \; . \end{array}$$

Example

Replace the arrays by uninterpreted functions:

$$
\begin{array}{ll}
 & (z < i \Rightarrow F_a(z) = 0) \\
\wedge & F_{a'}(i) = 0 \wedge (z \neq i \Rightarrow F_{a'}(z) = F_a(z)) \\
\wedge & z \leq i \wedge F_{a'}(z) \neq 0 \; .
\end{array}
$$

By distinguishing the three cases $z < i$, $z = i$, and $z > i$, it is easy to see that this formula is unsatisfiable.

Outlook SMT-$\mathcal{BVAUF}$

- The instantiations of the array axioms and the function conconsistency rule are typically done incrementally
$\rightarrow$ this is over-approximation

- Usually combined with constraints on hard bit-vector operators
$\rightarrow$ this is under-approximation

- Yes, both in the same instance!

- The rule instantiation extends to (incomplete) treatment for quantifiers (Z3 is good at this)

	decision procedures	verification engines
Bit-level	Propositional SAT	BMC Netlists
word-level	SMT-$\mathcal{BV}$	CBMC SystemC C/C++

Bounded Program Analysis

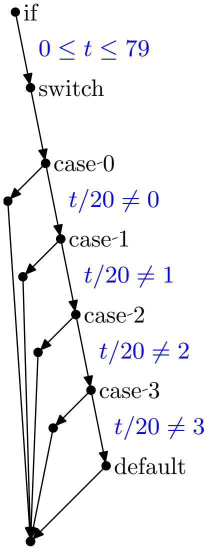
Goal: check properties of the form $\mathbf{G}p$,
say assertions.

Idea: follow paths through the CFG to an assertion

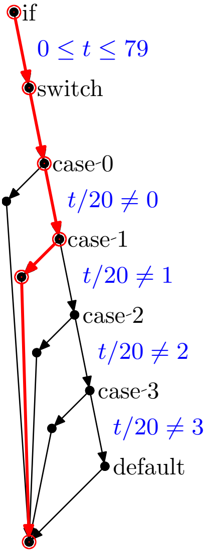
Example

```
if ( (0 <= t) && (t <= 79) )
  switch ( t / 20 )
  {
  case 0:
      TEMP2 = ( (B AND C) OR (~B AND D) );
      TEMP3 = ( K_1 );
      break;

  case 1:
      TEMP2 = ( (B XOR C XOR D) );
      TEMP3 = ( K_2 );
      break;

  case 2:
      TEMP2 = ( (B AND C) OR (B AND D) OR (C AND D) );
      TEMP3 = ( K_3 );
      break;

  case 3:
      TEMP2 = ( B XOR C XOR D );
      TEMP3 = ( K_4 );
      break;

  default:
      assert(0);
  }
```

(from an implementation of SHS)



Example



$$
\begin{array}{ll}
 & 0 \leq t \leq 79 \\
\wedge & t/20 \neq 0 \\
\wedge & t/20 = 1 \\
\wedge & \mathit{TEMP2} = B \oplus C \oplus D \\
\wedge & \mathit{TEMP3} = K\_2
\end{array}
$$

Example

We pass

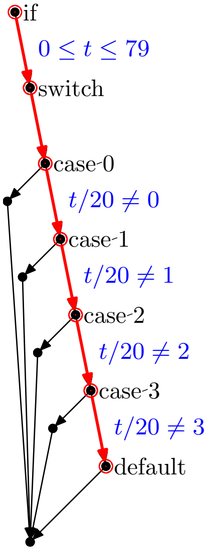
$$\begin{array}{ll} & 0 \leq t \leq 79 \\ \wedge & t/20 \neq 0 \\ \wedge & t/20 = 1 \\ \wedge & \mathit{TEMP2} = B \oplus C \oplus D \\ \wedge & \mathit{TEMP3} = K\_2 \end{array}$$

to a decision procedure, and obtain a satisfying assignment, say:

$$t \mapsto 21,\ B \mapsto 0,\ C \mapsto 0,\ D \mapsto 0,\ K\_2 \mapsto 10,$$
$$\mathit{TEMP2} \mapsto 0,\ \mathit{TEMP3} \mapsto 10$$

✔ It provides the values of any inputs on the path.

Another Example



$$
\begin{array}{ll}
 & 0 \leq t \leq 79 \\
\wedge & t/20 \neq 0 \\
\wedge & t/20 \neq 1 \\
\wedge & t/20 \neq 2 \\
\wedge & t/20 \neq 3
\end{array}
$$

That is UNSAT, so the assertion is unreachable.

What If a Variable is Assigned Twice?

```
x=0;

if (y>=0)
  x++;
```

Rename appropriately:

$$
\begin{array}{cl}
 & x_1 = 0 \\
\wedge & y_0 \geq 0 \\
\wedge & x_1 = x_0 + 1
\end{array}
$$

This is a special case of SSA (static single assignment)

Pointers

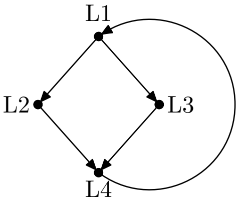
How do we handle dereferencing in the program?

```
int *p;
p=malloc(sizeof(int)*5);
 ...

p[1]=100;
```

$$\begin{array}{ll} & p_1 = \&\mathit{DO1} \\ \wedge & \mathit{DO1}_1 = (\lambda i. \\ & i = 1 ? 100 : \mathit{DO1}_0[i]) \end{array}$$

Track a ‘may-point-to’ abstract state while unwinding!

Scalability of Path Search



This is a loop with an `if` inside.

Q: how many paths for n iterations?

Bounded Model Checking

- Bounded Model Checking (BMC) is the most successful formal validation technique in the *hardware* industry

- Advantages:
  - ✔ Fully automatic
  - ✔ Robust
  - ✔ Lots of subtle bugs found

- Idea: only look for bugs up to specific depth

- Good for many applications, e.g., embedded systems

Transition Systems

Reminder: A transition system has a

- set of states S,
- a set of initial states $S_0 \subset S$, and
- a transition relation $T \subset (S \times S)$.

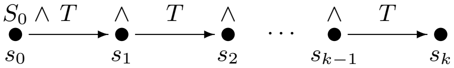
The set S_0 and the relation T can be written as their characteristic functions.

The graph with nodes S and edges T is called the Kripke structure.

Unwinding a Transition System

Q: How do we avoid the exponential path explosion?

We just ”concatenate” the transition relation T:



Unwinding a Transition System

As formula:

$$S_0(s_0) \wedge \bigwedge_{i=0}^{k-1} T(s_i, s_{i+1})$$

Satisfying assignments for this formula are traces through the Kripke structure

Example

$$T \subseteq \mathbb{N}_0 \times \mathbb{N}_0$$

$$T(s, s') \iff s'.x = s.x + 1$$

…and let $S_0(s) \iff s.x = 0 \vee s.x = 1$

An unwinding for depth 4:

$$\begin{array}{ll} & (s_0.x = 0 \vee s_0.x = 1) \\ \wedge & s_1.x = s_0.x + 1 \\ \wedge & s_2.x = s_1.x + 1 \\ \wedge & s_3.x = s_2.x + 1 \\ \wedge & s_4.x = s_3.x + 1 \end{array}$$

Unwinding a Transition System

Suppose we want to check a property of the form $\mathbf{G}p$.

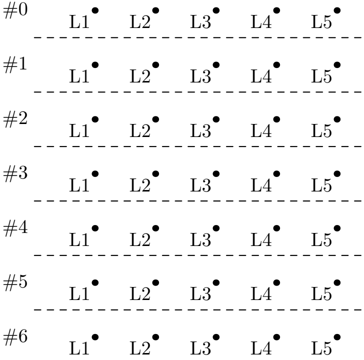
We then want at least one state s_i to satisfy $\neg p$:

$$S_0(s_0) \wedge \bigwedge_{i=0}^{k-1} T(s_i, s_{i+1}) \quad \wedge \quad \bigvee_{i=0}^{k} \neg p(s_i)$$

Unwinding Software

We can do exactly that for our transition relation for software.

E.g., for a program with 5 locations, 6 unwindings:



#0 L1 L2 L3 L4 L5

#1 L1 L2 L3 L4 L5

#2 L1 L2 L3 L4 L5

#3 L1 L2 L3 L4 L5

#4 L1 L2 L3 L4 L5

#5 L1 L2 L3 L4 L5

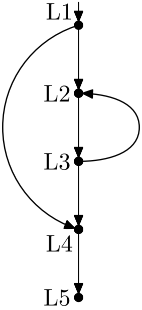
#6 L1 L2 L3 L4 L5

Unwinding Software

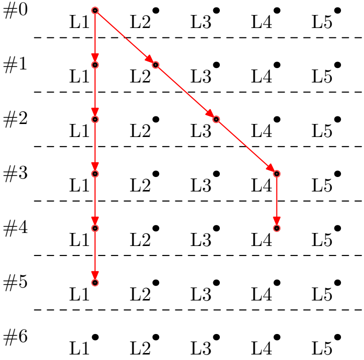
Problem: obviously, most of the formula is never ’used’, as only few sequences of PCs correspond to a path.

Unwinding Software

Example:



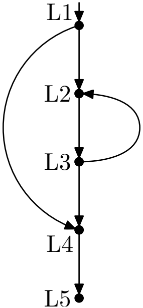
CFG



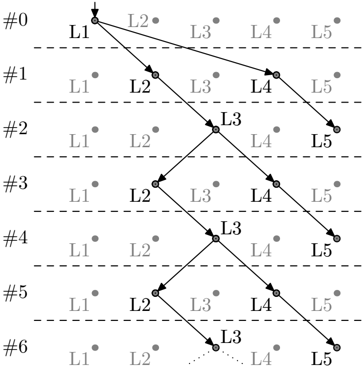
unrolling

Unwinding Software

Optimization:
don’t generate the parts of the formula that are not ’reachable’



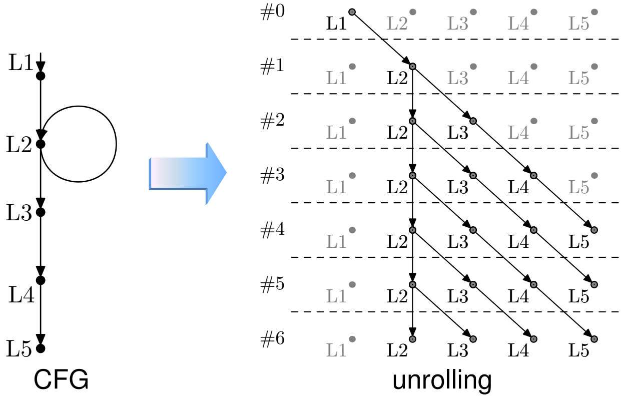
CFG



unrolling

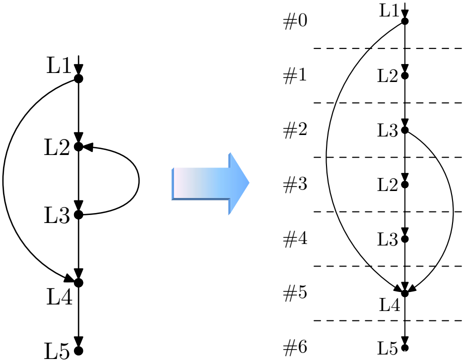
Unwinding Software

Another problem:



Unrolling Loops

Idea: do exactly one location in each timeframe:



CFG

unrolling

Unrolling Loops

This essentially amounts to unwinding loops:

```
if(cond) {
  Body;
  if(cond) {
    Body;
    if(cond) {
      Body;
      while(cond)
        Body;
    }
  }
}
```

Unrolling Loops

Problem: bad performance on some shallow bugs.

Solution: build multiple instances, in a BFS fashion

Solving the Decision Problem

Suppose we have used some unwinding, and have built the formula.

For bit-vector arithmetic, the standard way of deciding satisfiability of the formula is *flattening*,
followed by a call to a propositional SAT solver.

In the SMT context: SMT-$\mathcal{BV}$

Completeness

BMC, as discussed so far, is incomplete.
It only refutes, and does not prove.

How can we fix this?

Completeness: Summary

1. Unwinding assertions

2. Completeness thresholds

3. k-induction

Unwinding Assertions

Let’s revisit the loop unwinding idea:

```
if(cond) {
   Body;
   if(cond) {
      Body;
      if(cond) {
        Body;
        while(cond)
            Body;
      }
   }
}
```

Unwinding Assertions

- This allows us to prove that we have done enough unwinding.

- This is a proof of a high-level worst-case execution time (WCET).

- Appropriate for embedded software.

Completeness Thresholds

- Let's write

$$M \models_k \phi$$

  for "ϕ holds on paths of M up to length k".

- Idea: for finite state models, there is obviously some d with

$$M \models_d \phi \quad\iff\quad M \models \phi$$

- Such a d is called completeness threshold or cutoff.

- Getting smallest such d is as hard as deciding $M \models \phi$.

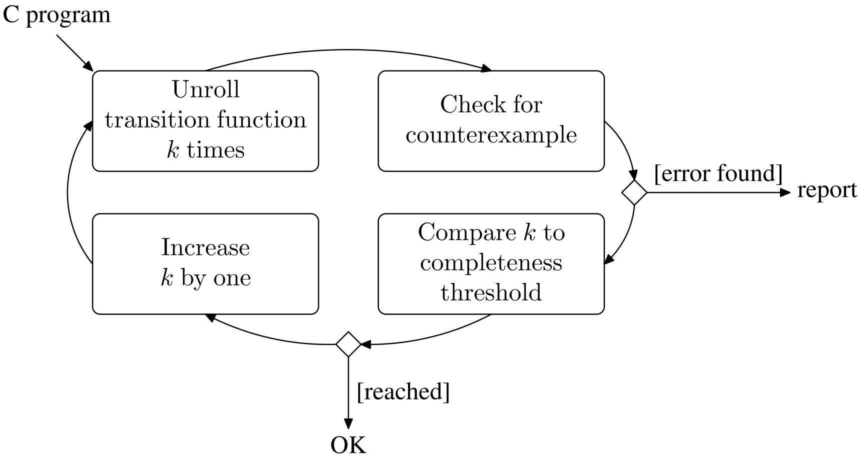
Completeness Thresholds

- Completeness thresholds are therefore overapproximated.

- Can be done in a property-specific way.

- Often yields a bound that is small enough.

Using the Completeness Threshold



Induction

In many cases, we can use inductive reasoning to show that assertions hold for an unbounded number of loop iterations:

```
int array[n];
 ...
for(unsigned i=0;
      i!=n;
      i++)
{
   assert(i<n);
    ...
}
```

$$\begin{aligned} & i' = i + 1 \land i < n \land i \neq n \\ \Rightarrow \quad & i' < n \end{aligned}$$

Induction

```
for(unsigned i=0;
      i!=10;
      i++)
{
   assert(i!=100);
    ...
}
```

$$\begin{array}{rl} & i' = i + 1 \wedge i \neq 100 \wedge i \neq 10 \\ \Rightarrow & i' \neq 100 \end{array}$$

k-induction

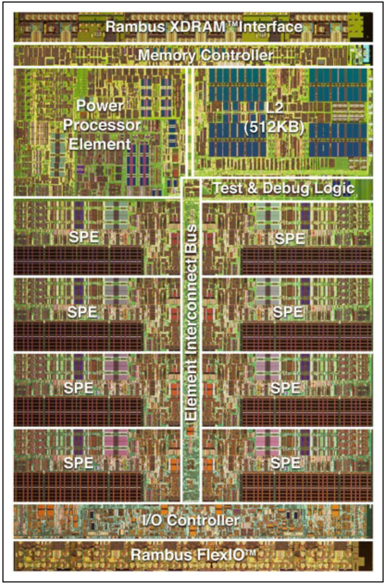
Idea:

- Induction step assumes that k iterations are successful
- Often elininates the need for invariant strengthening
- Useful loops that have “bounded memory”

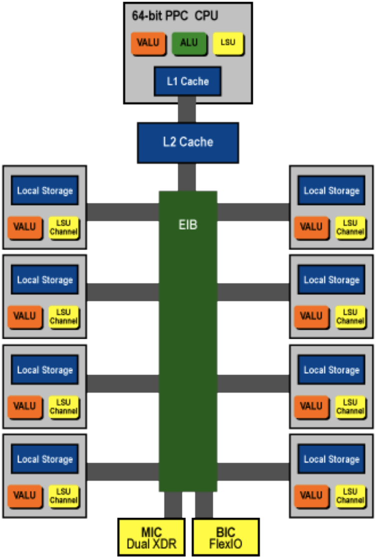
For formalization, see TACAS 2010 paper.

The Cell Broadband Engine Processor

- Used in Sony's PlayStation 3
- Also in the top supercomputer



The Cell Broadband Engine Processor



- EIB: element interface bus
- Four sixteen-byte data rings with 64-bit tags
- Transfers 96 bytes/cycle
- Handles over 100 outstanding requests

DMA on the Cell BE

- put(l, h, s, t)

  issues a transfer of s bytes from local store address l to host address h, identified by tag t

- get(l, h, s, t)

  issues a transfer of s bytes from host address h to local store address l, identified by tag t

- wait(t)

  blocks until completion of all pending DMA operations identified by tag t

Example

```
float buffers[3][CHUNK/sizeof(float)]; // Triple-buffering requires 3 buffers

void triple_buffer(char* in, char* out, int num_chunks) {
    unsigned int tags[3] = { 0, 1, 2 }, tmp, put_buf, get_buf, process_buf;

(1) get(buffers[0], in, CHUNK, tags[0]); // Get triple-buffer scheme rolling
    in += CHUNK;
(2) get(buffers[1], in, CHUNK, tags[1]);
    in += CHUNK;
(3) wait(tags[0]); process_data(buffers[0]); // Wait for and process first buffer
    put_buf = 0; process_buf = 1; get_buf = 2;
    for(int i = 2; i < num_chunks; i++) {
(4)   put(buffers[put_buf], out, CHUNK, tags[put_buf]); // Put data processed
      out += CHUNK;                                     //   last iteration
(5)   get(buffers[get_buf], in, CHUNK, tags[get_buf]); // Get data to process
      in += CHUNK;                                     //   next iteration
(6)   wait(tags[process_buf]);                  // Wait for and process data
      process_data(buffers[process_buf]);       //   requested last iteration

      tmp = put_buf; put_buf = process_buf; // Cycle the buffers
      process_buf = get_buf; get_buf = tmp;
    }
    ... // Handle data processed/fetched on final loop iteration
}
```

DMA Races

Definition

Let $\mathsf{op}_1(l_1, h_1, s_1, t_1)$ and $\mathsf{op}_2(l_2, h_2, s_2, t_2)$ be a pair of simultaneously pending DMA operations, where $\mathsf{op}_1, \mathsf{op}_2 \in \{\mathsf{put}, \mathsf{get}\}$.

The pair is said to be *race free* if the following holds:

$$((\mathsf{op}_1 = \mathsf{put} \wedge \mathsf{op}_2 = \mathsf{put}) \vee (l_1 + s_1 \leq l_2) \vee (l_2 + s_2 \leq l_1)) \wedge$$
$$((\mathsf{op}_1 = \mathsf{get} \wedge \mathsf{op}_2 = \mathsf{get}) \vee (h_1 + s_1 \leq h_2) \vee (h_2 + s_2 \leq h_1)).$$

Experiments

- Implementation on top of CBMC
- 22 benchmarks from IBM Cell SDK
- Runtime for most $< 1\,s$
- Found previously unknown bug in SDK example

Experiments

	Correct		
Benchmark	iterations	time	speedup
1-buf	15	9.49	23.14 ×
2-buf	>100	>1352.43	>417.78 ×
3-buf	>100	>4344.98	>120.9 ×

	Buggy		
Benchmark	iterations	time	speedup
1-buf	3	1.25	2.91 ×
2-buf	20	33.62	59.97 ×
3-buf	69	4969.03	6641.47 ×

Speedup in comparison to SATABS

References

- Clarke, E., Kroening, D., Lerda, F.:
  A tool for checking ANSI-C programs.
  In: TACAS. Volume 2988 of LNCS, Springer (2004) 168–176

- Cordeiro, L., Fischer, B., Marques-Silva, J.:
  SMT-based bounded model checking for embedded ANSI-C software.
  In: ASE. (2009)

- Kroening, D., Strichman, O.:
  Efficient computation of recurrence diameters.
  In: VMCAI. Volume 2575 of LNCS, Springer (2003) 298–309

- Sheeran, M., Singh, S., Stålmarck, G.:
  Checking safety properties using induction and a SAT-solver.
  In: FMCAD. Volume 1954 of LNCS, Springer (2000) 108–125

Outlook

There is more. Ask me about

- Concurrency (including weak consistency)
- Automated abstraction refinment (SLAM and the like)
- Floating-point arithmetic
- Automated test-suite generation
- Combinations of SAT and abstract interpretation

Formal Techniques for Hardware/Software Co-Verification

Daniel Kroening, Mandayam Srivas

26th International Conference on VLSI

January 2013

Pune,India

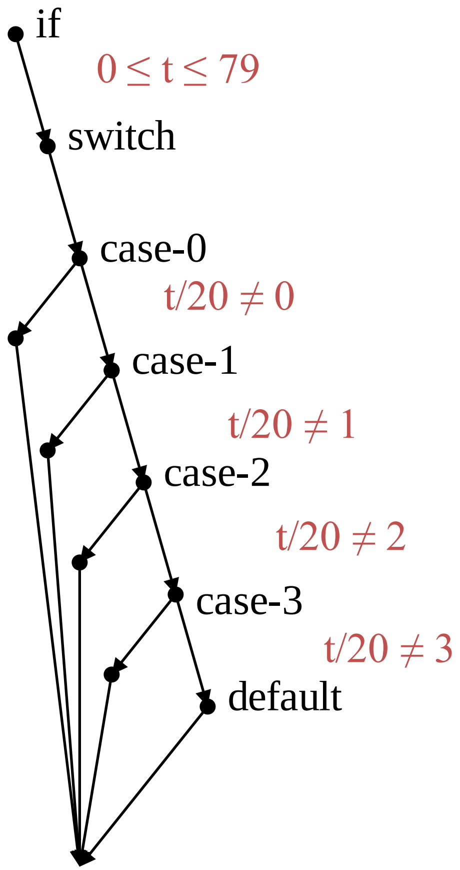
- Acknowledgements: Madukar Kumar Singh, TRDC
Nassim Seghir, Oxford

Outline

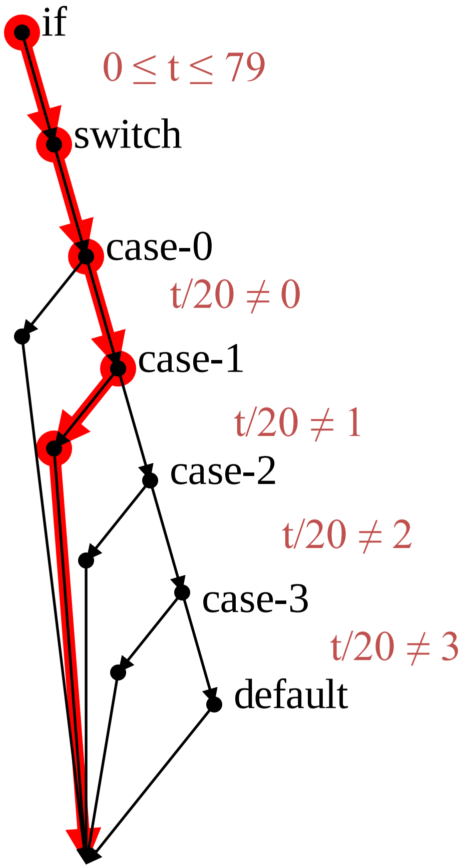
1. BMC-based FV methodology overview
2. Introduction to CBMC on Binsearch
3. RTL (Verilog) verification
4. Multi-media IP verification
   - K-induction
5. Model-based verification of automotive SW
6. Microprocessor verification
   - Sequential circuit equivalence
   - C Vs RTL

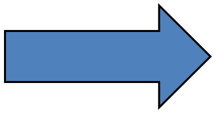
Step-1: Convert C-code into Control-Flow-Graph

```
if ( (0 <= t) && (t <= 79) )
 switch (t/20)
  { case  0:
      TEMP2 = ( (B AND C) OR (!B AND D) );
      TEMP3 = ( K_1 );
       break;
    case  1:
      TEMP2 = ( (B XOR C XOR D) );
      TEMP3 = ( K_2 );
       break;
    case  2:
      TEMP2 = ( (B AND C) OR (B AND D) OR (C AND D) );
      TEMP3 = ( K_3 );
       break;
    case  3:
      TEMP2 = ( B XOR C XOR D );
      TEMP3 = ( K_4 );
       break;
    default: assert(0);
   }
```



Step-2: Generate Formula for Path





$$
\begin{aligned}
& 0 \leq t \leq 79 \\
\wedge \quad & t/20 \neq 0 \\
\wedge \quad & t/20 \neq 1 \\
\wedge \quad & \mathrm{TEMP2} = \mathrm{B} \; xor \; \mathrm{C} \; xor \; \mathrm{D} \\
\wedge \quad & \mathrm{TEMP3} = \mathrm{K\_2}
\end{aligned}
$$

Step-3: Pass formulas to SAT Solver

Pass

$$
\begin{aligned}
& 0 \leq t \leq 79 \\
\wedge\ & t/20 \neq 0 \\
\wedge\ & t/20 \neq 1 \\
\wedge\ & \mathrm{TEMP2} = \mathrm{B}\ \mathit{xor}\ \mathrm{C}\ \mathit{xor}\ \mathrm{D} \\
\wedge\ & \mathrm{TEMP3} = \mathrm{K\_2}
\end{aligned}
$$

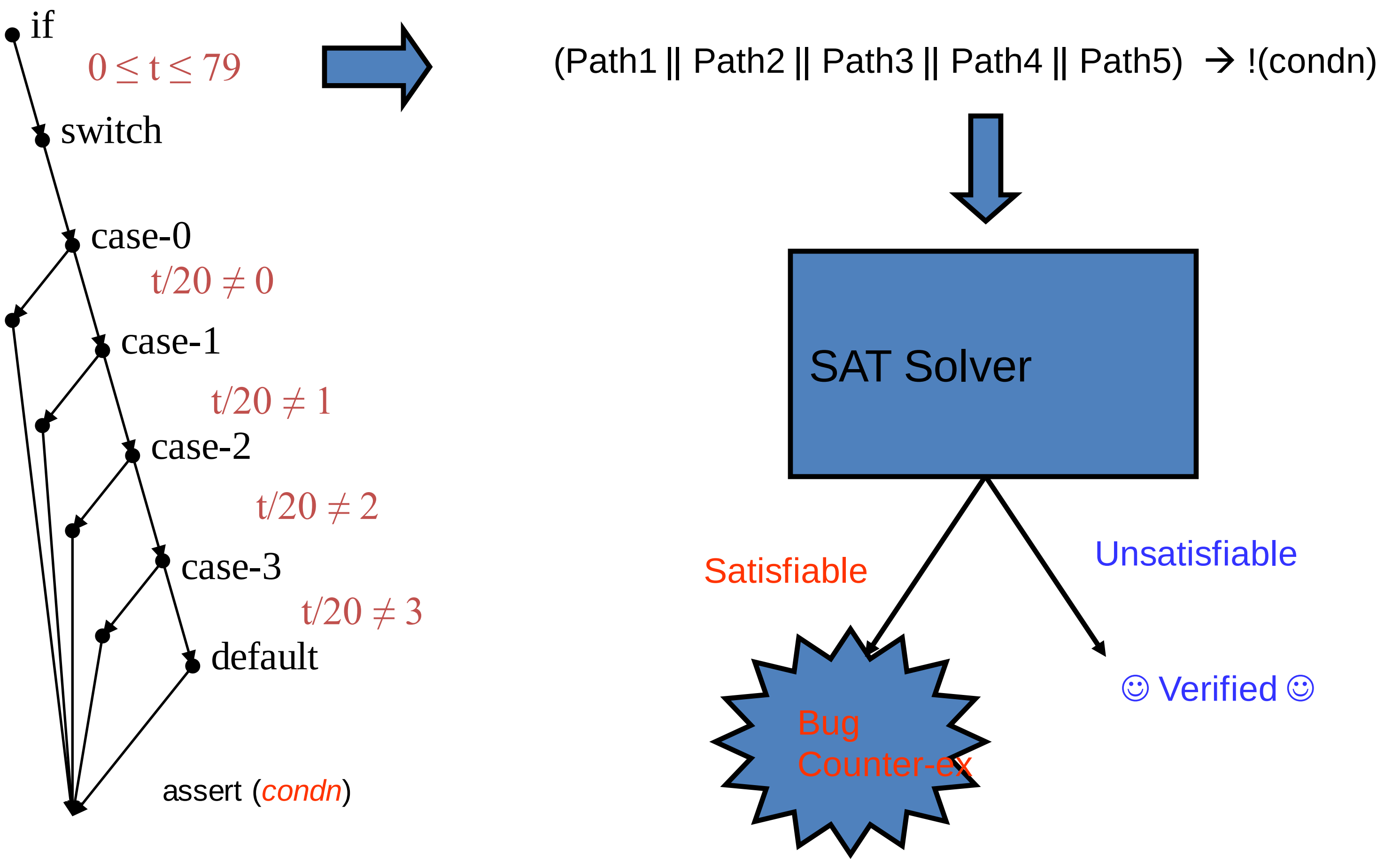
to a SAT solver to obtain a satisfying assignment of values

$$
t \rightarrow 21,\ B \rightarrow 0,\ C \rightarrow 0,\ D \rightarrow 0,\ K\_2 \rightarrow 10, \\
\mathrm{TEMP2} \rightarrow 0,\ \mathrm{TEMP3} \rightarrow 10
$$

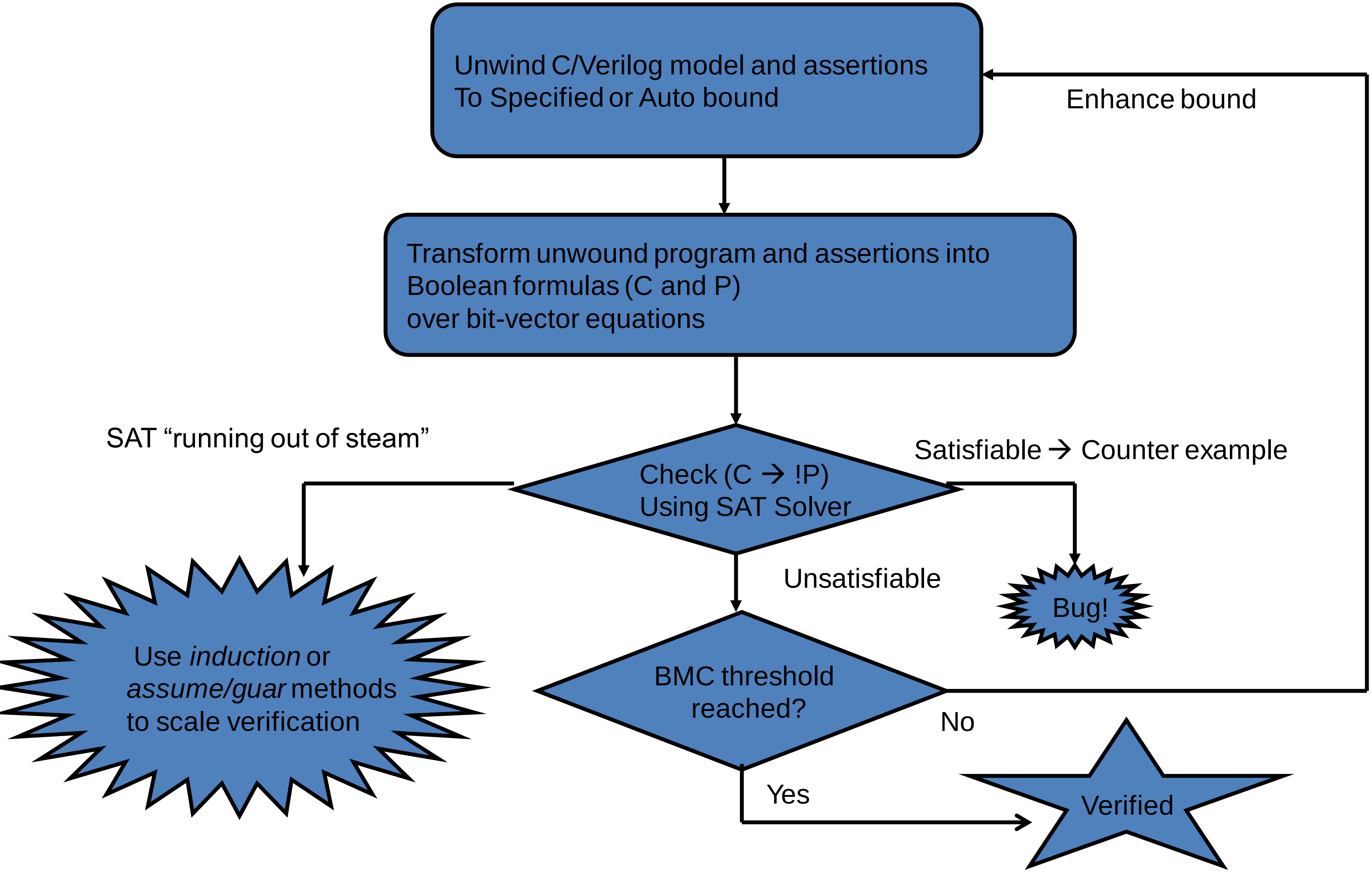
Denotes a set of possible values that any inputs can take on the path

Can be used to check if user-defined assertions hold at program locations

Putting It All Together: Step-1+Ste-2+Step3



Bounded Model Chekcing (BMC): Verification Flow-chart



Unwind C/Verilog model and assertions
To Specified or Auto bound

Transform unwound program and assertions into
Boolean formulas (C and P)
over bit-vector equations

Check (C → !P)
Using SAT Solver

SAT "running out of steam"

Use *induction* or
assume/guar methods
to scale verification

Satisfiable → Counter example

Bug!

Unsatisfiable

BMC threshold
reached?

No

Enhance bound

Yes

Verified

Outline

1. BMC-based FV methodology overview
2. Introduction to CBMC (Binsearch)
3. RTL (Verilog) verification
4. Multi-media IP verification
   - K-induction
5. Model-based verification of automotive SW
6. Microprocessor verification
   - Sequential circuit equivalence
   - C Vs RTL

CBMC Features to Hi-light

- Assumptions: constrained non-determinism
- Assertions:
  - Implicitly generated
  - Getting effect of Quantification
- Basic CBMC commands
  - show-claims, show-loops
  - Checking assertions
  - Controlling unwinding depth
- Analyzing error traces

Outline

1. BMC-based FV methodology overview
2. Introduction to CBMC (Binsearch)
3. RTL (Verilog) verification
4. Multi-media IP verification
   - K-induction
5. Model-based verification of automotive SW
6. Microprocessor verification
   - Sequential circuit equivalence
   - C Vs RTL

Objectives of the Exercise

- Verilog module verification
  - 3 designs implement same function with different timing and resource usage
  - Verify them against same C-specification
- How do you handle timing/clock issues?
- How do you monitor and drive verilog signals?
- How do you analyze verilog error traces?

```
always @(posedge clk) begin : update
   if (rst) begin
     xavg_out   <= 0;
     xdif_out   <= 0;
   end else begin : normal
     reg [9:0] xsum_next;
     reg [7:0] xavg_next;
     xsum_next  = x0_in + x1_in + x2_in + x3_in;
     xavg_next  = xsum_next >> 2;
     xavg_out   <= xavg_next;
     xdif_out   <= (xavg_next >= x3_in)?(xavg_next - x3_in):(x3_in - xavg_next);
   end
 end
```

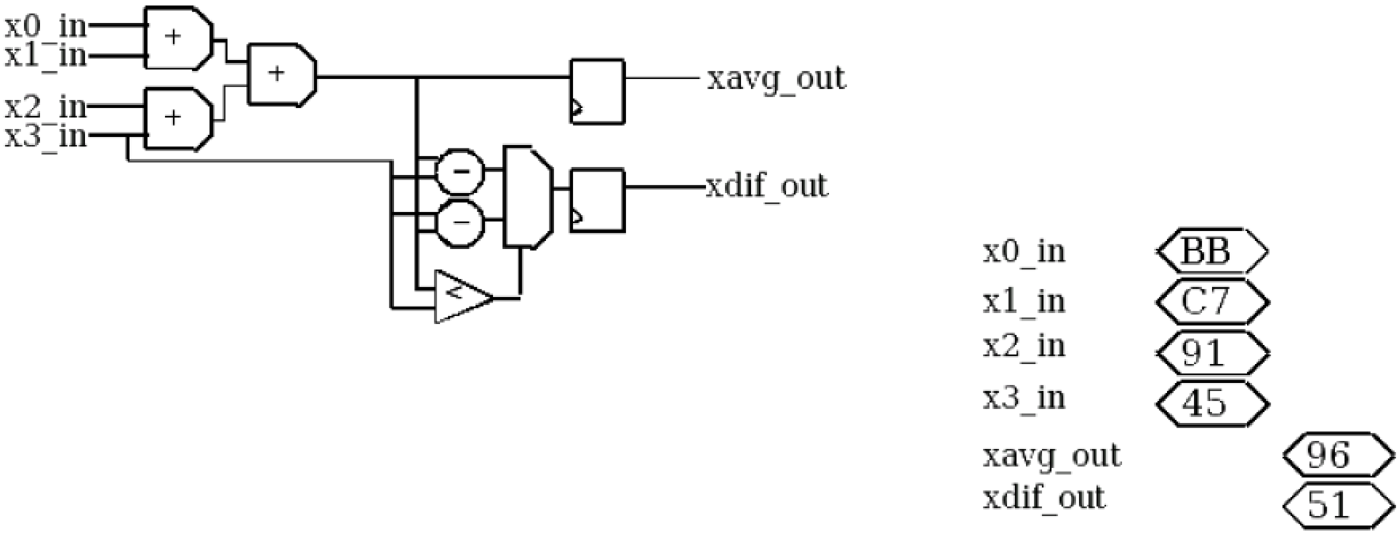


Figure 1: Average4 design

Courtesy: Calypto Design

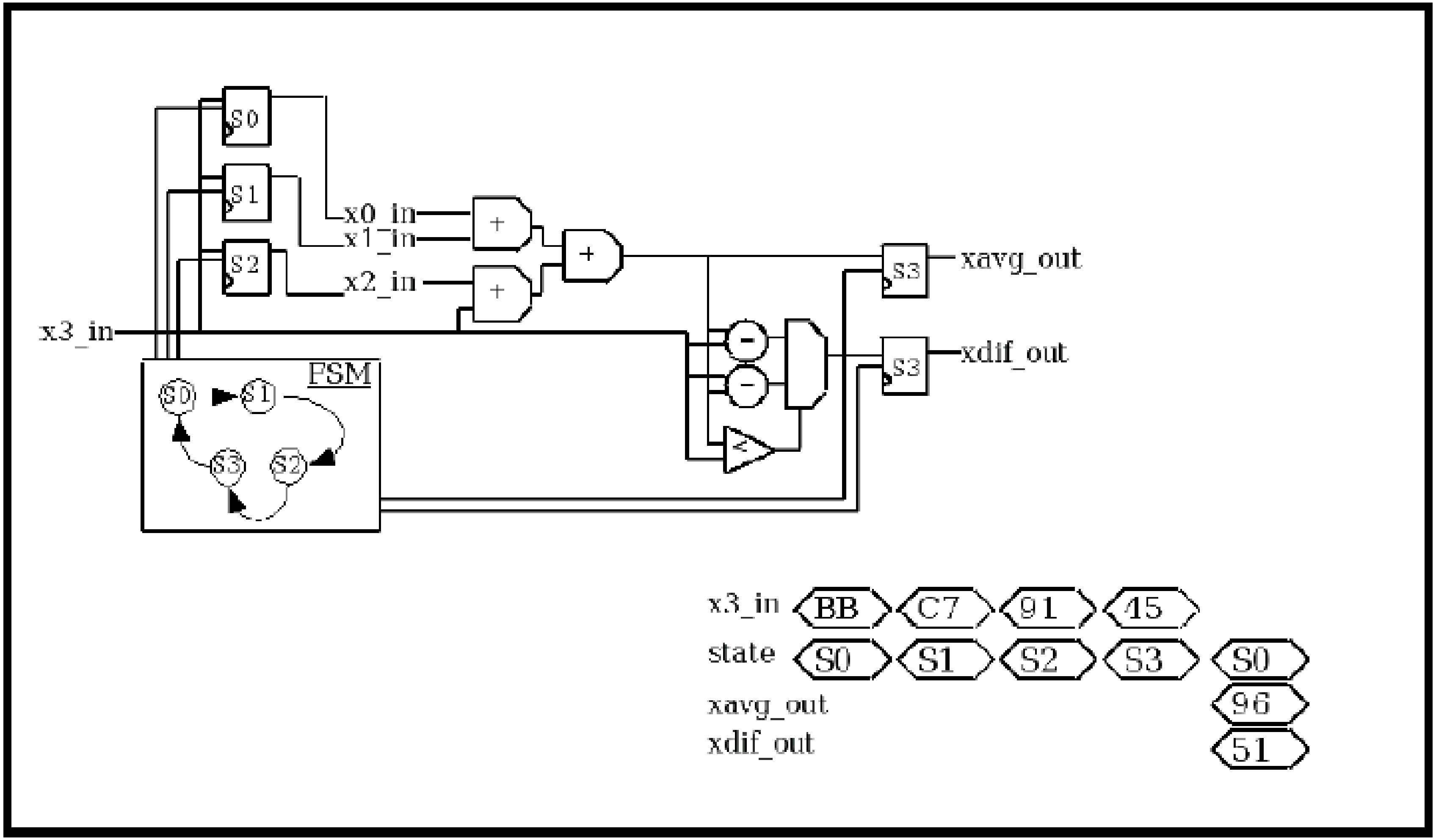
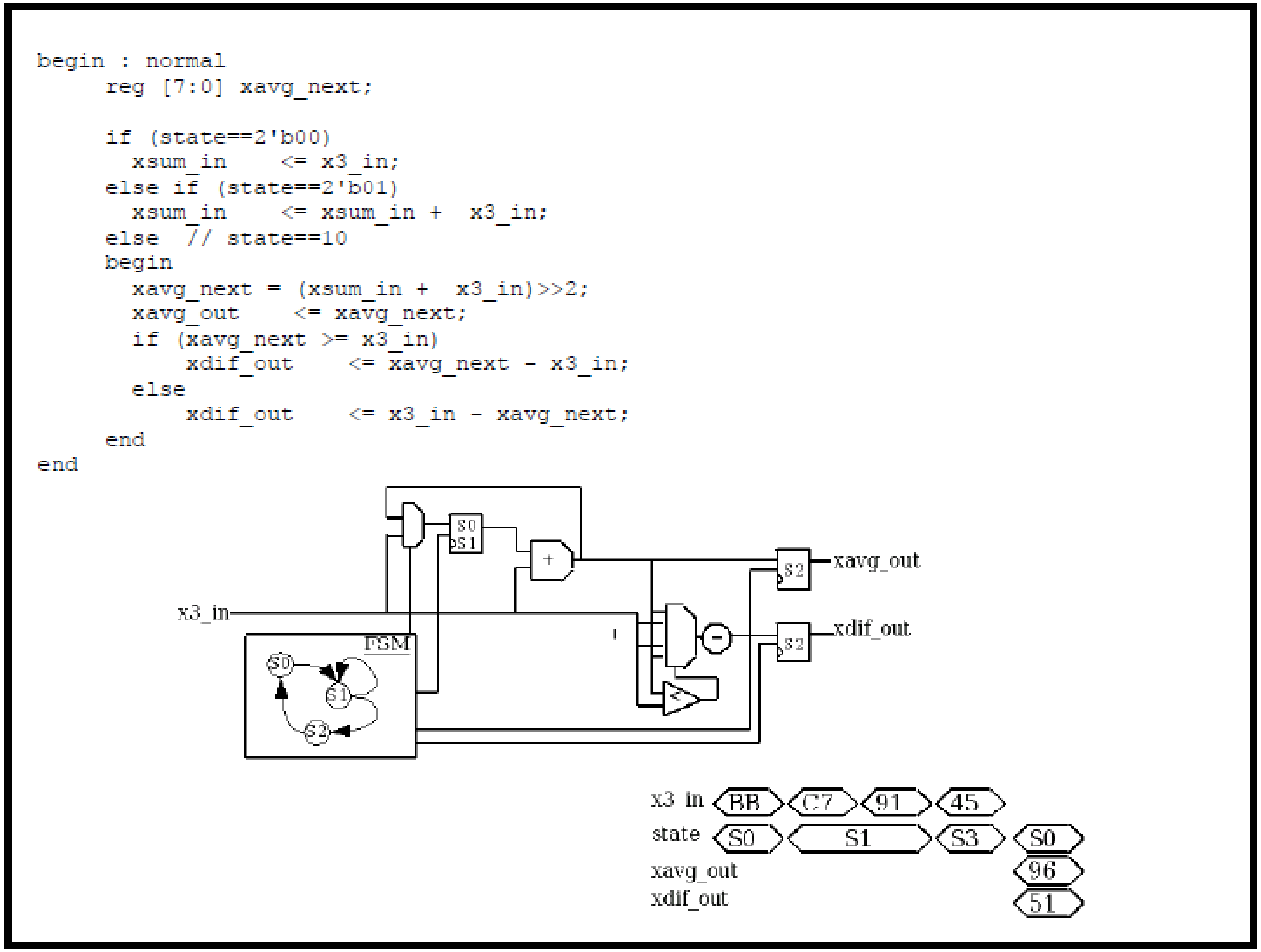


Figure 2: Serial implementation of Average4 circuit

Courtesy: Calypto Design

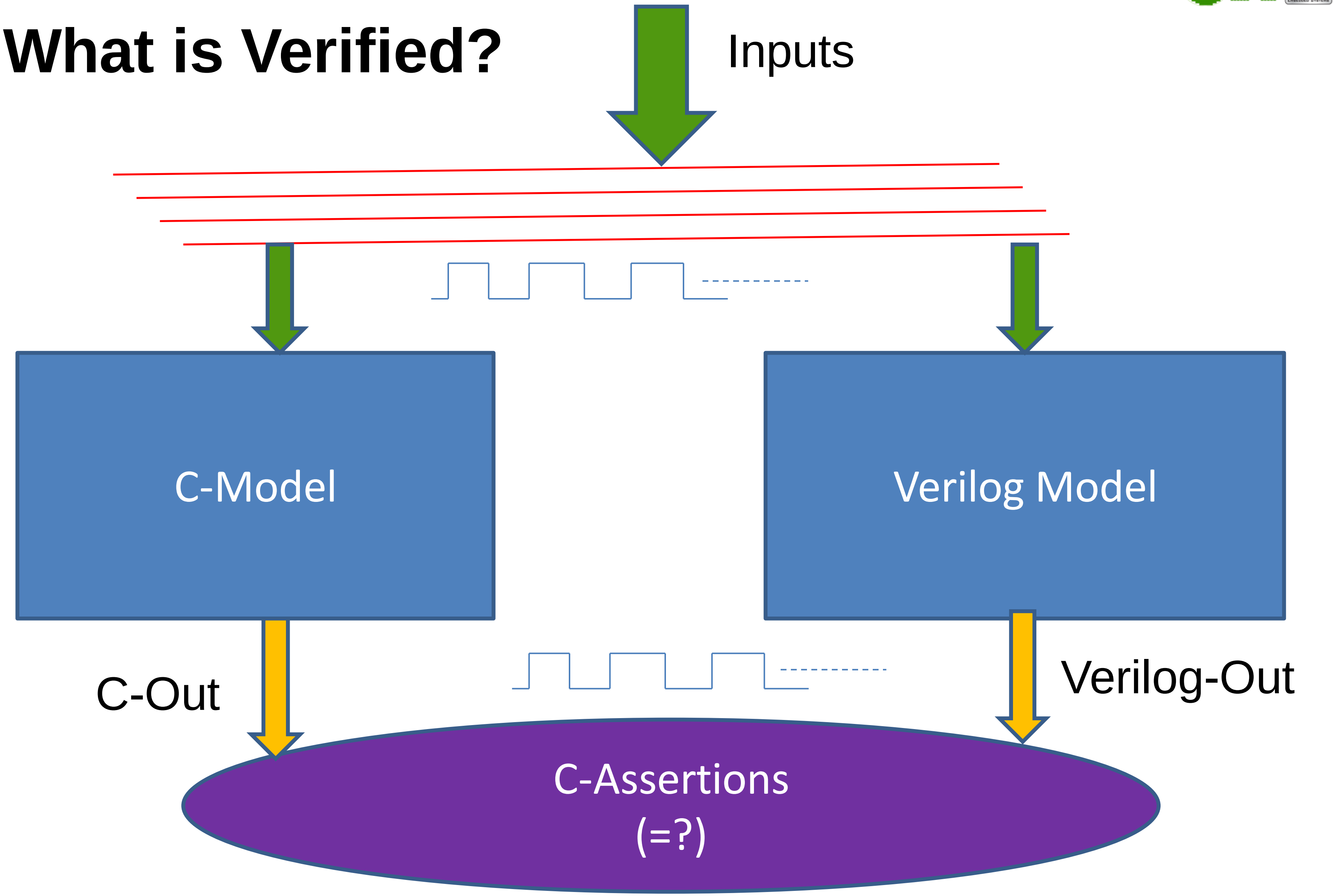


```
begin : normal
     reg [7:0] xavg_next;

     if (state==2'b00)
       xsum_in    <= x3_in;
     else if (state==2'b01)
       xsum_in    <= xsum_in +  x3_in;
     else  // state==10
     begin
       xavg_next = (xsum_in +  x3_in)>>2;
       xavg_out    <= xavg_next;
       if (xavg_next >= x3_in)
           xdif_out    <= xavg_next - x3_in;
       else
           xdif_out    <= x3_in - xavg_next;
     end
end
```

Figure 3: Power optimized Average4 design

Courtesy: Calypto Design

What is Verified?



CBMC Infrastructure for RTL verification

- “Import” a verilog module into a C-Wrapper:
- Synthesize verilog into a transition system
  - pin-accurate and clock-cycle-accurate
  - next_timeframe() effect of once clock-cycle transition
- All verilog signals can be accessed
  - Verilog input signals can be forced/constrained from the C-wrapper via __CPROVER_assume/set_inputs()
  - Output and any intermediate signals can be monitored and check
- C Vs. RTL consistency check can be done by importing C-model and verified assertions on C-model into the C-wrapper

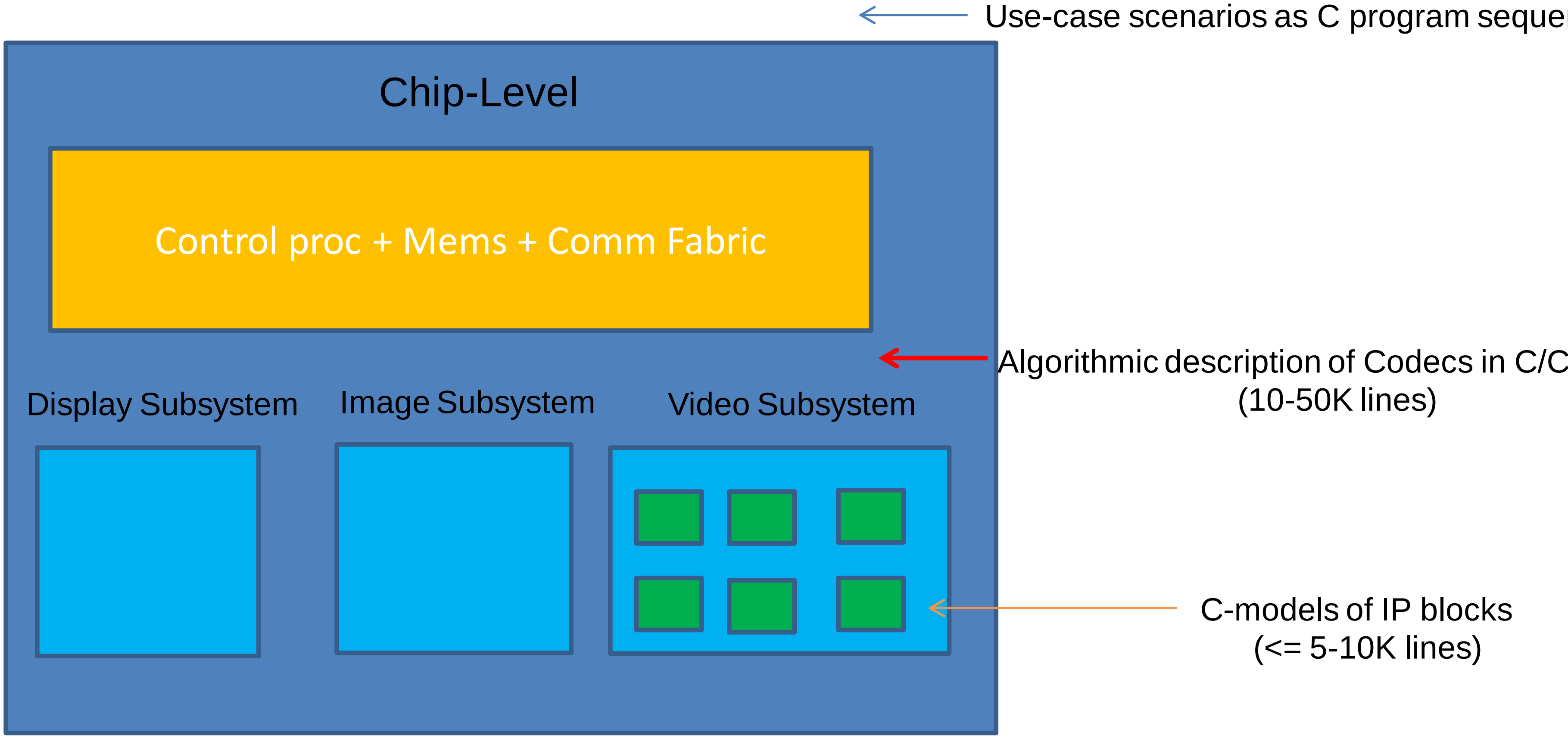
Steps for performing C Vs. RTL FV

1. Construct C-wrapper
2. Specify required reset/input signal protocols
3. Define and check C- behavioral assertions
    a. Map C vars used in C-assertions to corresponding Verilog signals
    b. Specify trigger events each C-assertions need to be triggered
    c. This depends on the latency and timing or RTL behavior
    d. Steps 1&2 will yield mapped trigger-event qualified assertions for RTL
4. Specify unwind depth for verilog module
5. Fire FV run

Outline

1. BMC-based FV methodology overview
2. Introduction to CBMC (Binsearch)
3. RTL (Verilog) verification
4. Multi-media IP verification
    - K-induction
5. Model-based verification of automotive SW
6. Microprocessor verification
    - Sequential circuit equivalence
    - C Vs RTL

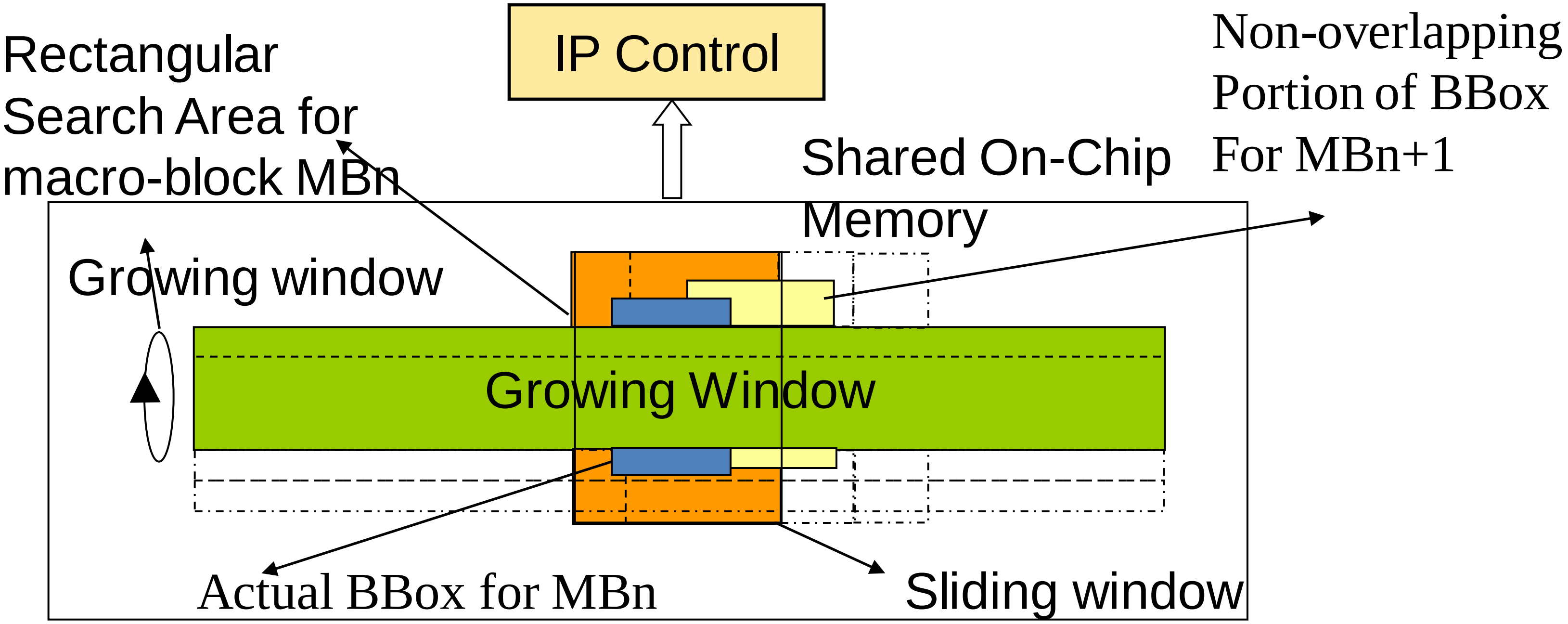
C/C++ Models in SoC Design Flow


Use-case scenarios as C program seque
Chip-Level
Control proc + Mems + Comm Fabric
Algorithmic description of Codecs in C/C
(10-50K lines)
Display Subsystem
Image Subsystem
Video Subsystem
C-models of IP blocks
(<= 5-10K lines)

Typical Multi-media IP blocks

- Large (8K X 8K) array processing: 2-3 deep nested loops
- Arrays consist of "macro blocks" of pixel clusters
  - Mostly similar computation for each block with some history
  - Sometimes, computation is cumulative for the whole array
- Data-oriented with lot of fixed-point arithmetic

ME5 – Bbox HWOD: Hybrid Window With Overlap Detection

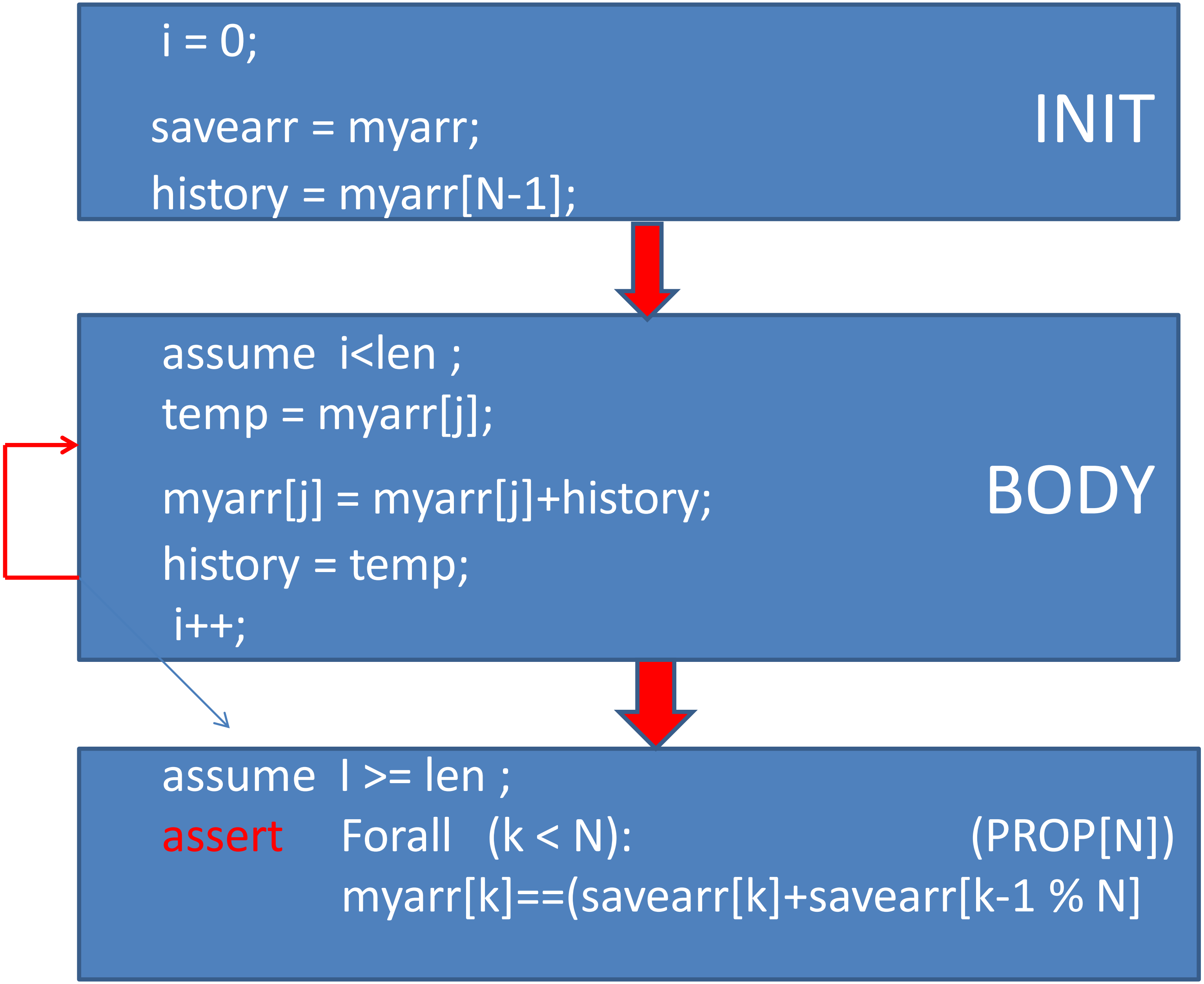


- **Hybrid Window: Uses** part sliding and part growing window to efficiently utilize the available DDR bandwidth and on chip memory
- **Overlap detection (OD): Only** non overlapping regions of the ***bounding box*** w.r.t. the previous macro-block are identified and fetched (DMA commands)from DDR
  - ~1500 lines of C and ~3000 lines of verilog
- **FV Goals**:
  - Define <u>complete</u> behavior of algorithm as formal assertions on logical coordinates
    - OD: “Every part of non-overlapping region and nothing else” is fetched
  - Formally verify the properties for <u>both</u> C-model and RTL
  - Verify “physical address” generation by showing direct C Vs RTL code equivalence

Technical challenges

- Unwinding for whole array is not practical
  - Solution: Use induction
- Challenges in employing induction:
  - May need to formulate “loop invariants”
    - To capture history recorded b/w iterations
  - K-induction may help [Donaldson,et.al., SAS2011]
- Coding of HW-timing protocol
  - Solution: Automatic synthesis from timing diagrams

LOOP PROBLEM



```
i = 0;
savearr = myarr;
history = myarr[N-1];
```

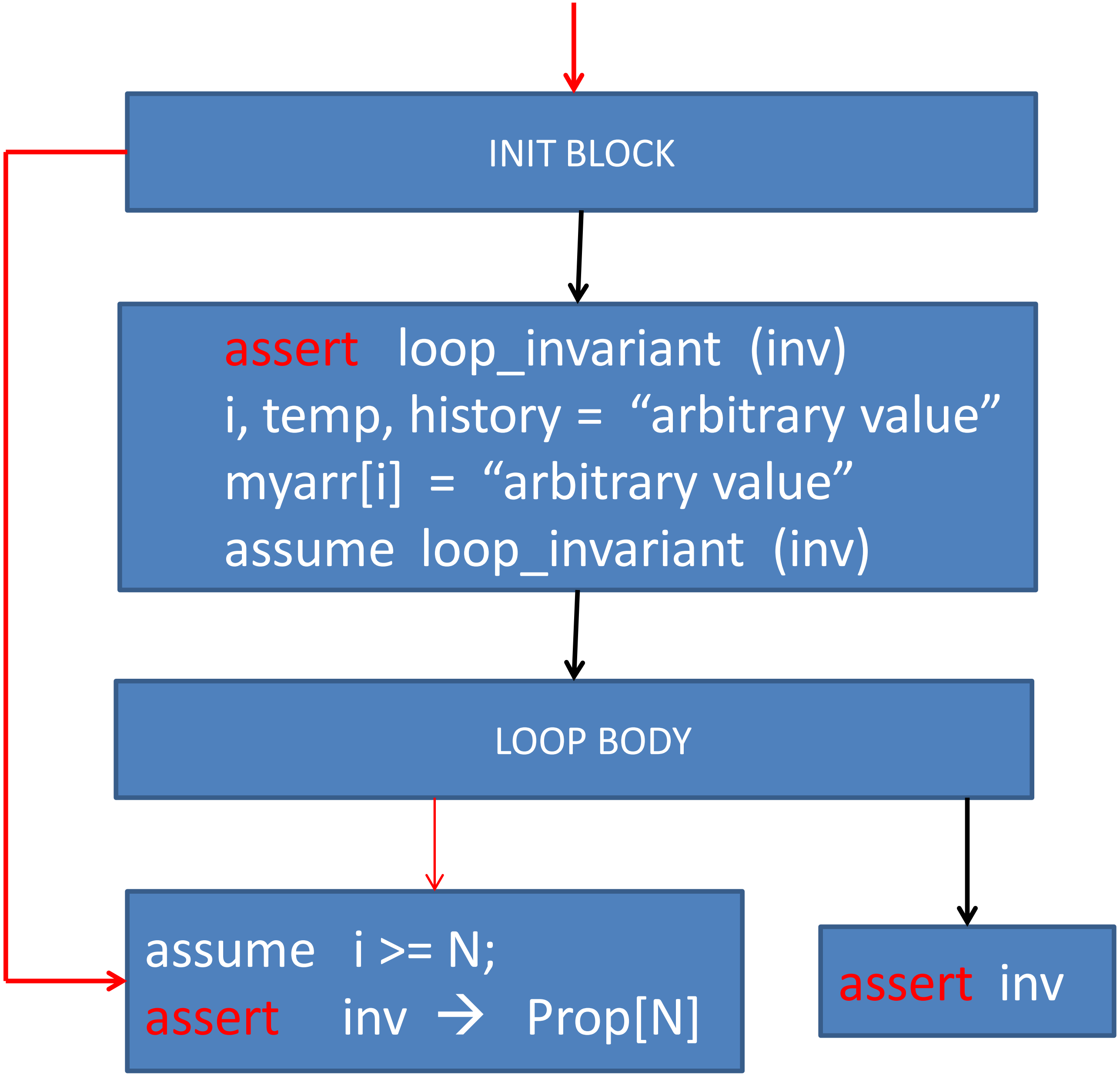
INIT

```
assume  i<len ;
temp = myarr[j];
myarr[j] = myarr[j]+history;
history = temp;
 i++;
```

BODY

```
assume  I >= len ;
assert     Forall   (k < N):                    (PROP[N])
           myarr[k]==(savearr[k]+savearr[k-1 % N]
```

LOOP INV APPROACH



INIT BLOCK

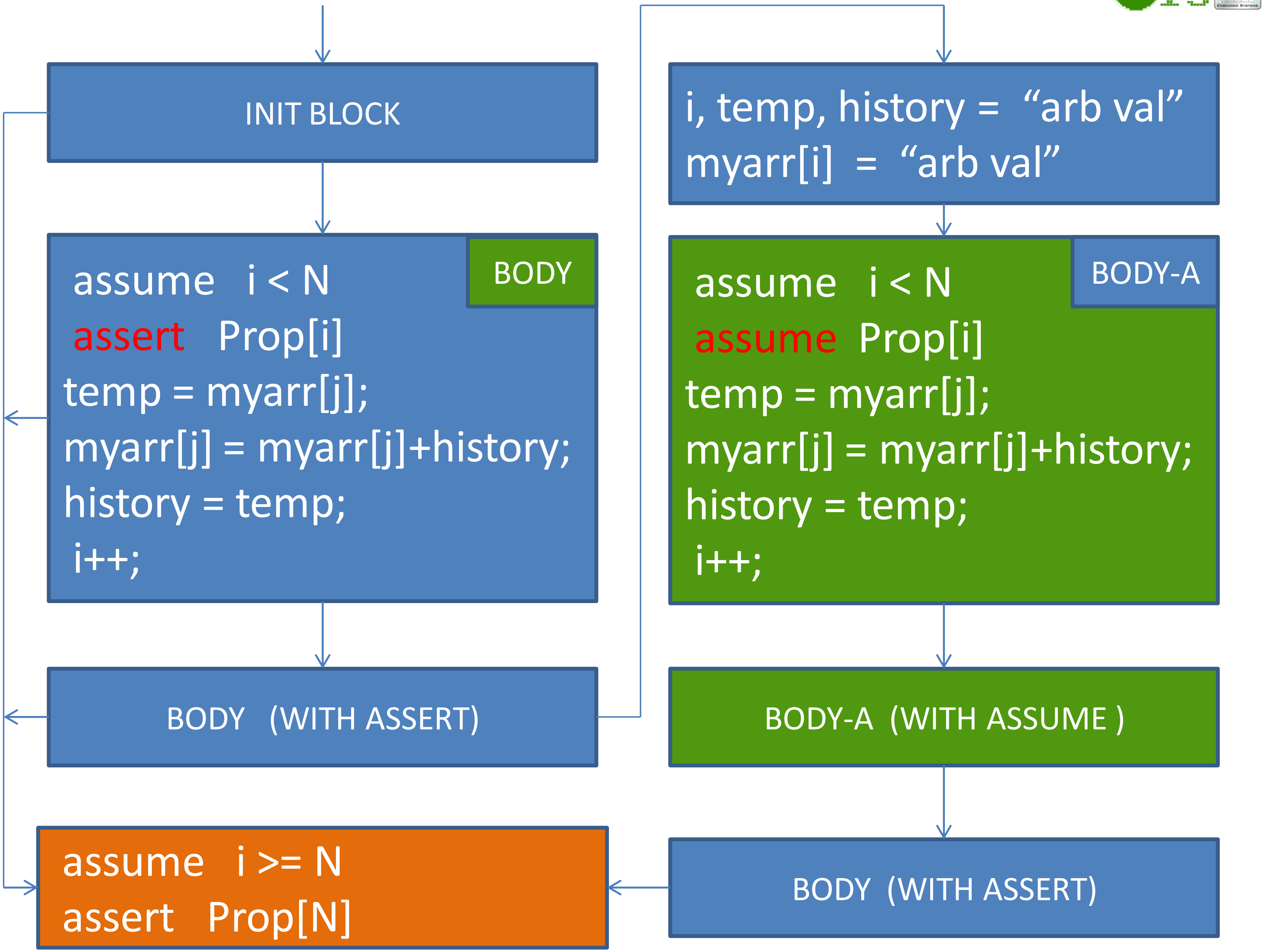
assert loop_invariant (inv)
i, temp, history = “arbitrary value”
myarr[i] = “arbitrary value”
assume loop_invariant (inv)

LOOP BODY

assume i >= N;
assert inv → Prop[N]

assert inv

LOOP K (2) INDUCTION



INIT BLOCK

BODY

```
assume   i < N
assert   Prop[i]
temp = myarr[j];
myarr[j] = myarr[j]+history;
history = temp;
 i++;
```

BODY   (WITH ASSERT)

```
assume   i >= N
assert   Prop[N]
```

```
i, temp, history =  "arb val"
myarr[i]  =  "arb val"
```

BODY-A

```
assume   i < N
assume  Prop[i]
temp = myarr[j];
myarr[j] = myarr[j]+history;
history = temp;
 i++;
```

BODY-A (WITH ASSUME)

BODY (WITH ASSERT)

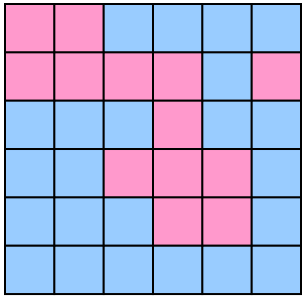
Memory Access Merging – Problem and Properties

Design problem: Optimize DMA access by increasing size of "block" fetches

Implementation: Scans a 2-dim array of "tiles" looking to merge adjacent "fetches" into a single rectangular larger multi-fetch (~300 lines of C and RTL (~400 lines verilog)

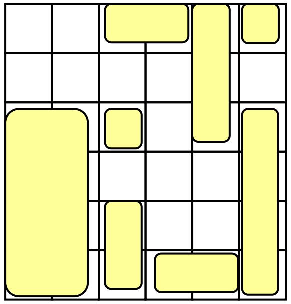
- Bigger block: 800 lines of C and 2500 lines of verilog

Input example



🟥 Don't Fetch
🟦 Fetch

Output example



🟨 Merged for Fetch

Verification problem

- 2 power 36 input combinations
- 4 times the input combinations considering arrival of 8x8 blocks (aligned or non-aligned)
- **Functional Bug:** A bug where a required data is not fetched - will cause MC failure
- **Scheme Bug** : Implementation is not following expected scheme
- **Performance Bug: Non** optimality - can cause potential functional failure due to increase in bw

Some example properties:

- **Functional Property**: "Every blue box in input should be part of *at-least* one yellow box in output"
- **Performance property:** "Every blue box in input should be part of *exactly* one yellow box in output"

FV Goal:

- **Verify all assertions for C-model, first and verify the same on RTL**

Technical challenges: Memory_access_merge

- Doesn’t scale beyond 8X8 arrays at C-level
  - RTL verification scales much better 8X20
  - Parallelism inherent in RTL seems to help
- Direct application of K-induction doesn’t work
  - Worst-case history-depth can be as large as array dimensions
- Internal buffer needs to be exposed to define loop invariant

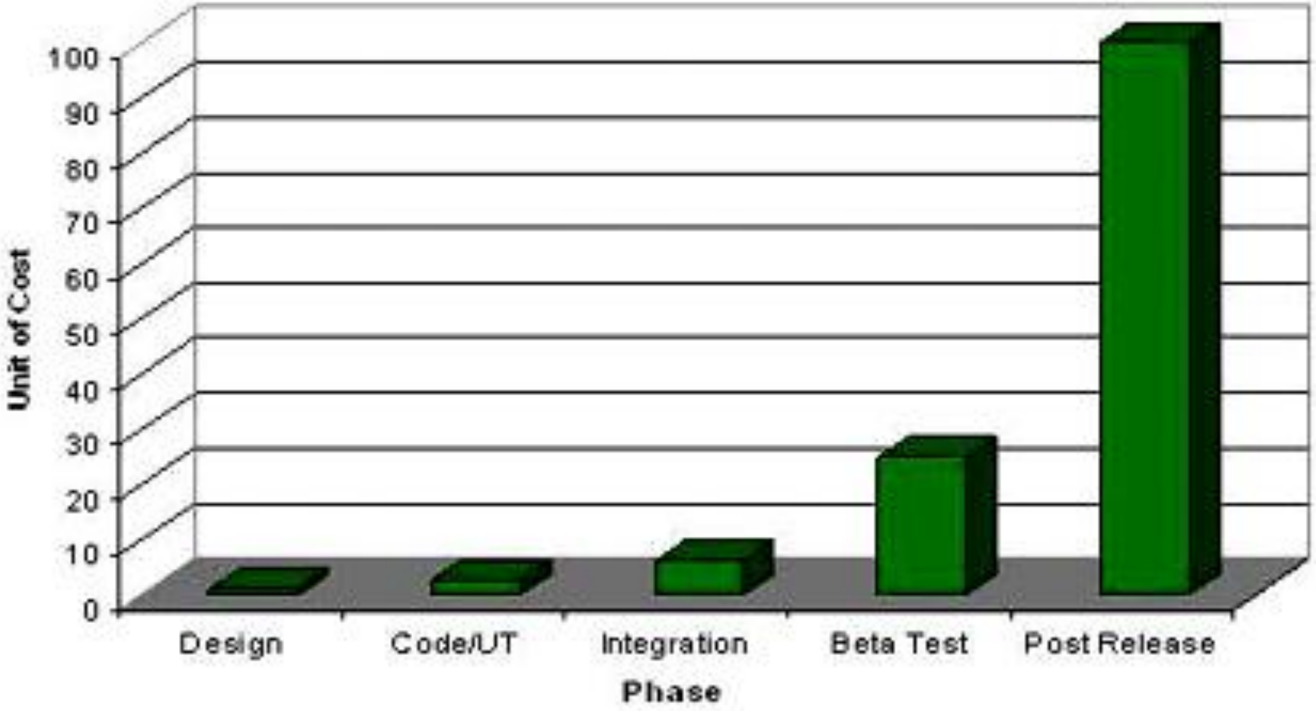
Some Useful General Techniques

- Redefine Property using info in history buffers
  - Need to combine information in internal and output buffers
- Instrument assertions to Force-Flush history information before checking assertions
  - Can reverse-engineer existing code
- Both require design knowledge
- Use loop-invariant generation techniques
  - An active research area

Outline

1. BMC-based FV methodology overview
2. Introduction to CBMC (Binsearch)
3. RTL (Verilog) verification
4. Multi-media IP verification
   - K-induction
5. Model-based verification of automotive SW
6. Microprocessor verification
   - Sequential circuit equivalence
   - C Vs RTL

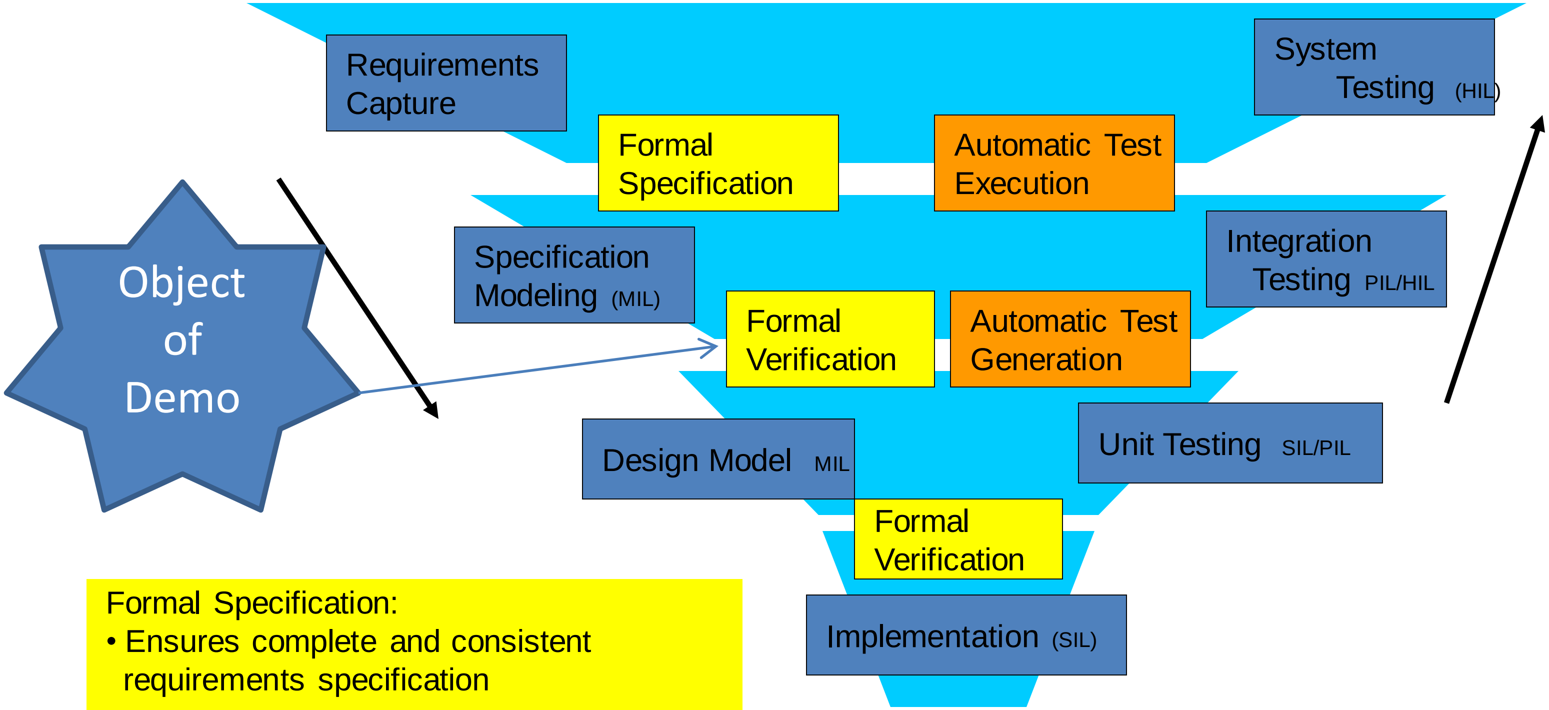
Earlier Verification is Better



Courtesy: David N. Kleidermache, EE Times

Model-based System Development:

Formal Verification Value-Addition

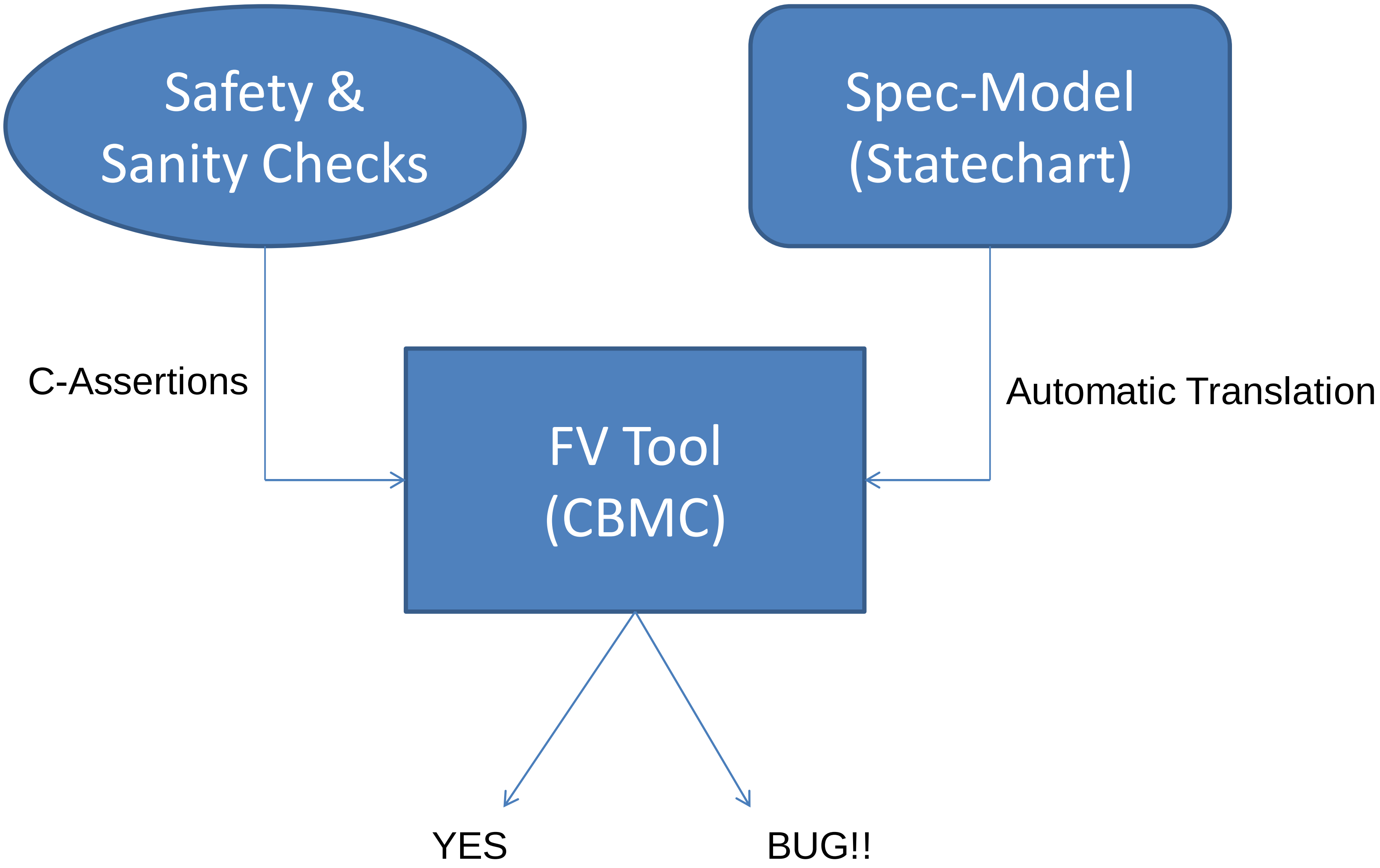


Formal Specification:
- Ensures complete and consistent requirements specification

Formal Verification:
- Ensures *expected* behavior
- Excludes *unintended* behavior
- Ensures full verification *coverage*
- Leads to reliable models and code

Model-Based SW Development

- Some popular Spec-modeling languages
  - Statechart/Statemate [Harel&Politi98]
  - Matlab/Simulink
- Verification on executable spec-models
  - Sanity, safety, and consistency checks
    - A good candidate for FV
  - Equivalence b/w spec-model and generated code



Safety &
Sanity Checks

Spec-Model
(Statechart)

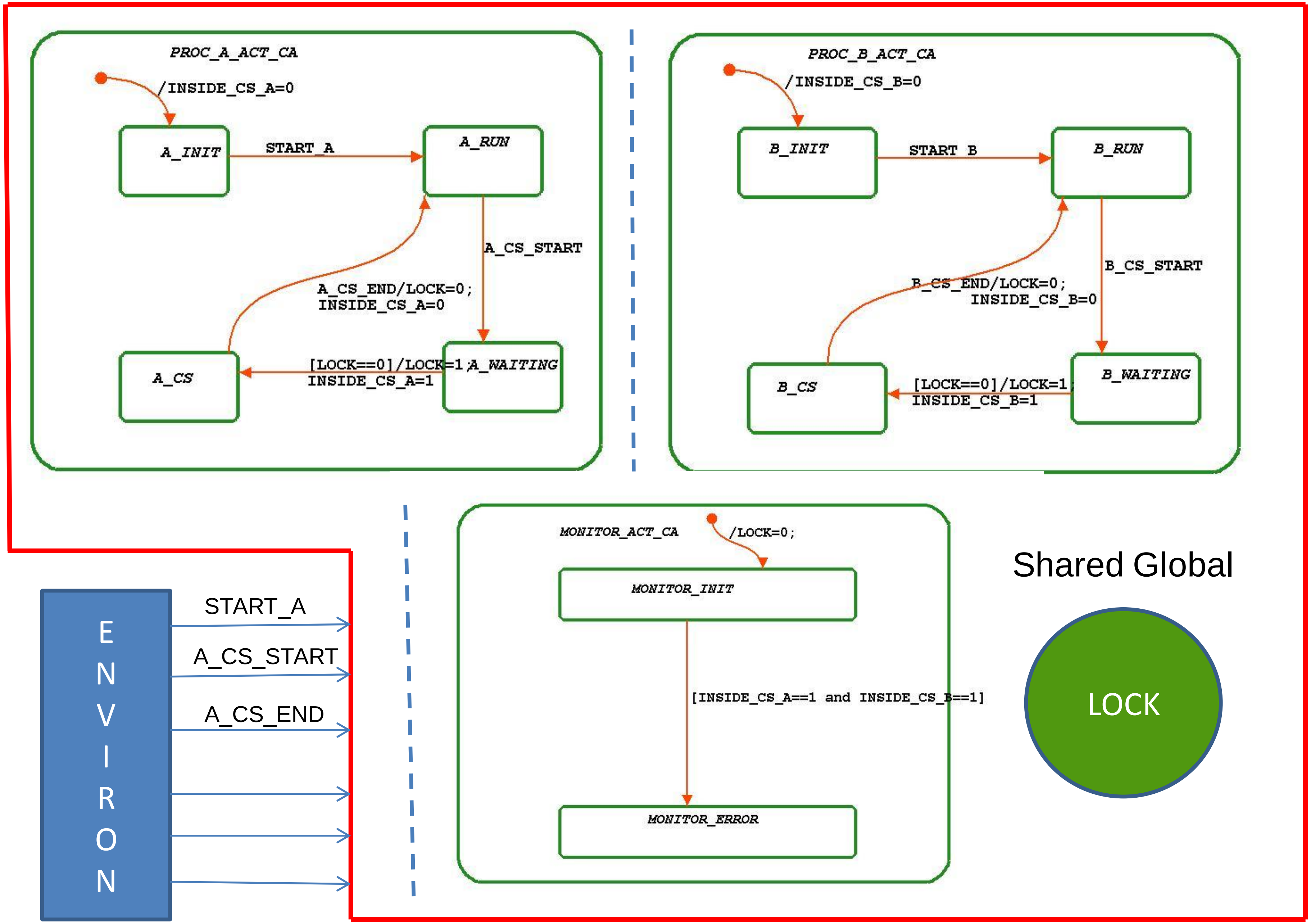
C-Assertions

Automatic Translation

FV Tool
(CBMC)

YES

BUG!!



PROC_A_ACT_CA

/INSIDE_CS_A=0

A_INIT

START_A

A_RUN

A_CS_START

A_CS_END/LOCK=0;
INSIDE_CS_A=0

A_CS

[LOCK==0]/LOCK=1;
INSIDE_CS_A=1

A_WAITING

PROC_B_ACT_CA

/INSIDE_CS_B=0

B_INIT

START B

B_RUN

B_CS_START

B_CS_END/LOCK=0;
INSIDE_CS_B=0

B_CS

[LOCK==0]/LOCK=1;
INSIDE_CS_B=1

B_WAITING

MONITOR_ACT_CA

/LOCK=0;

MONITOR_INIT

[INSIDE_CS_A==1 and INSIDE_CS_B==1]

MONITOR_ERROR

Shared Global

LOCK

ENVIRON

START_A

A_CS_START

A_CS_END

Main Features of Statecharts

- A hierarchy of finite-state machines (FSM)
  - OR-state: Regular FSM where a state can be refined into another statechart
  - AND-state: “Synchronous” composition of a set of OR-states
- Transition: “Guard/Action”
  - Guards: boolean events/conditions
  - Actions: Modify events and variables
  - Enabled, if guard evaluates to true
- Statecharts share global variables

Reactive System Semantics

```
While (true) { //BigStep
    1. Environment: update external events
    2. While (Exists: Enabled Transitions) { //Step
        a. Evaluate guards of transitions
        b. Pick a "maximal set" of enabled transitions
        c. Compute results of  actions in enabled transitions
        d. Update results  in arbitrary total order of transitions
        e. }  //End of Step
    3. } //End of Bigstep
```

Reactive Semantics Special Notes

- Pick one enabled transition from each OR-state in an AND-state
  - if >1, pick one non-deterministically
- All transitions are evaluated on old value
- Update events/variables in some total order
- External events change only once every BigStep
- Repeat Step until there are no enabled transitions

Some Properties of Interest

- Safety: Bad states are never reachable
  - NOT(in(A_CS) && in(B_CS))
- Progress: “BigStep Convergence”
  - BigStep always terminates
  - i.e., must eventually reach an idle state
- Determinism:
  - Behavior is invariant w.r.t. choice and order of transitions

Challenges in Statechart Analysis

- Scaling to system with a few hundreds of charts and a dozen deep hierarchy
- A few techniques that can help
  - Exploit determinism in || composition
  - Exploit Inherent structure in OR-state
    - Assume-Guarantee reasoning
    - Abstraction

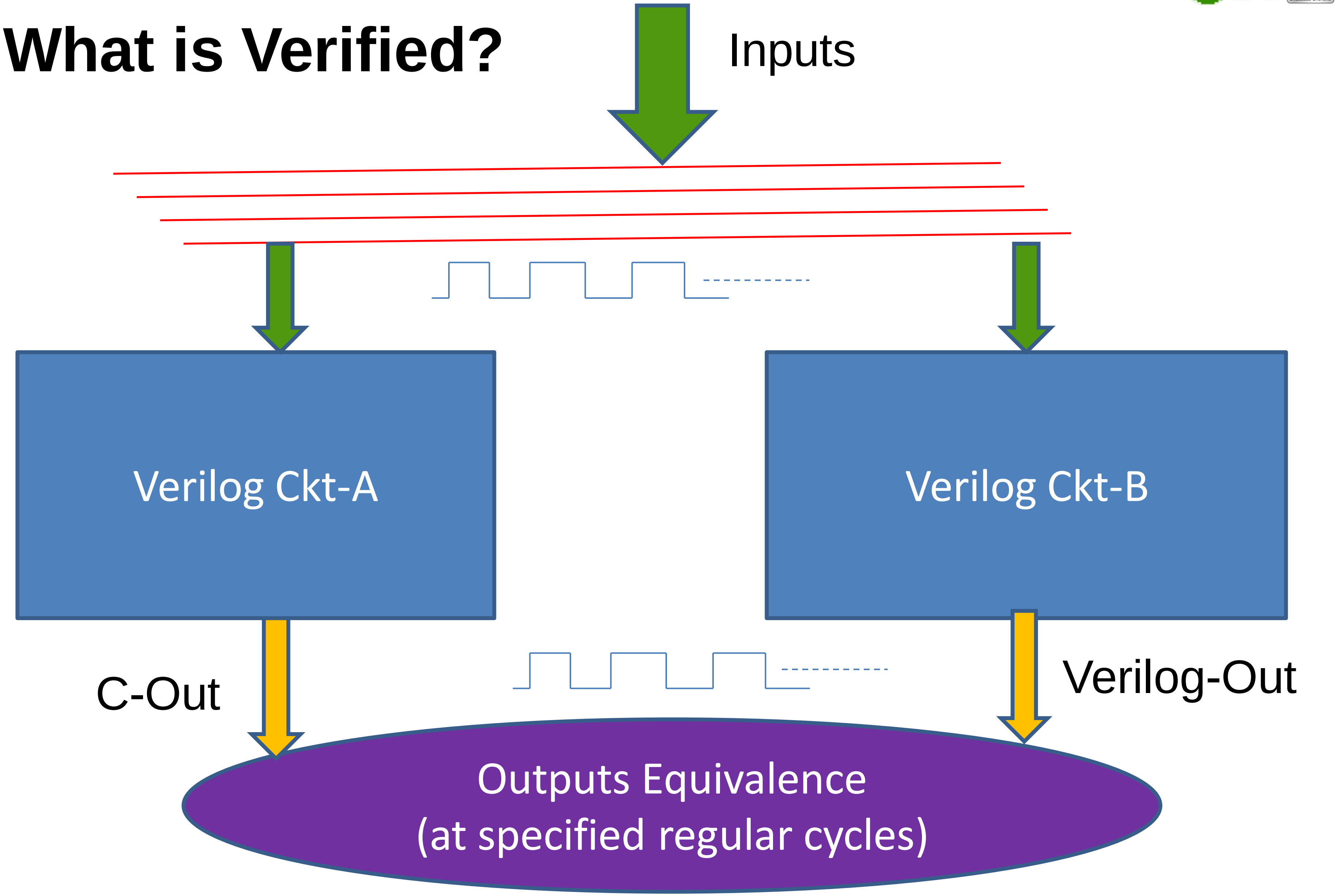
Statechart Demo Objectives

- Modeling statecharts in C
  - Transition systems in C
  - Statemate semantics of || composition
- Efficiency gain of partial-order reduction
- Safety property checking
- BigStep convergence checking
- Lasso-like-loop checking for more effective reachability property checking [BiereCyrilleSchuppan]

Outline

1. BMC-based FV methodology overview
2. Introduction to CBMC (Binsearch)
3. RTL (Verilog) verification
4. Multi-media IP verification
   - K-induction
5. Model-based verification of automotive SW
6. Microprocessor verification
   - Sequential circuit equivalence
   - C Vs RTL

What is Verified?



Inputs

Verilog Ckt-A

Verilog Ckt-B

C-Out

Verilog-Out

Outputs Equivalence
(at specified regular cycles)

Seq. Circuit Equiv. Problem

- Given: 2 RTL ckts (A,B) with identical I/O
- Check:
  - For identical seq. of inputs (after reset),
  - A and B have equivalent outputs
  - In every cycle or (at specified regular points)
- Infinite trace equivalence property
  - Q: How to convert into a finite-distance property for BMC?
- Consider Avg4 circuits considered earlier
  - Are they output equivalent at every t+3, t>=0?

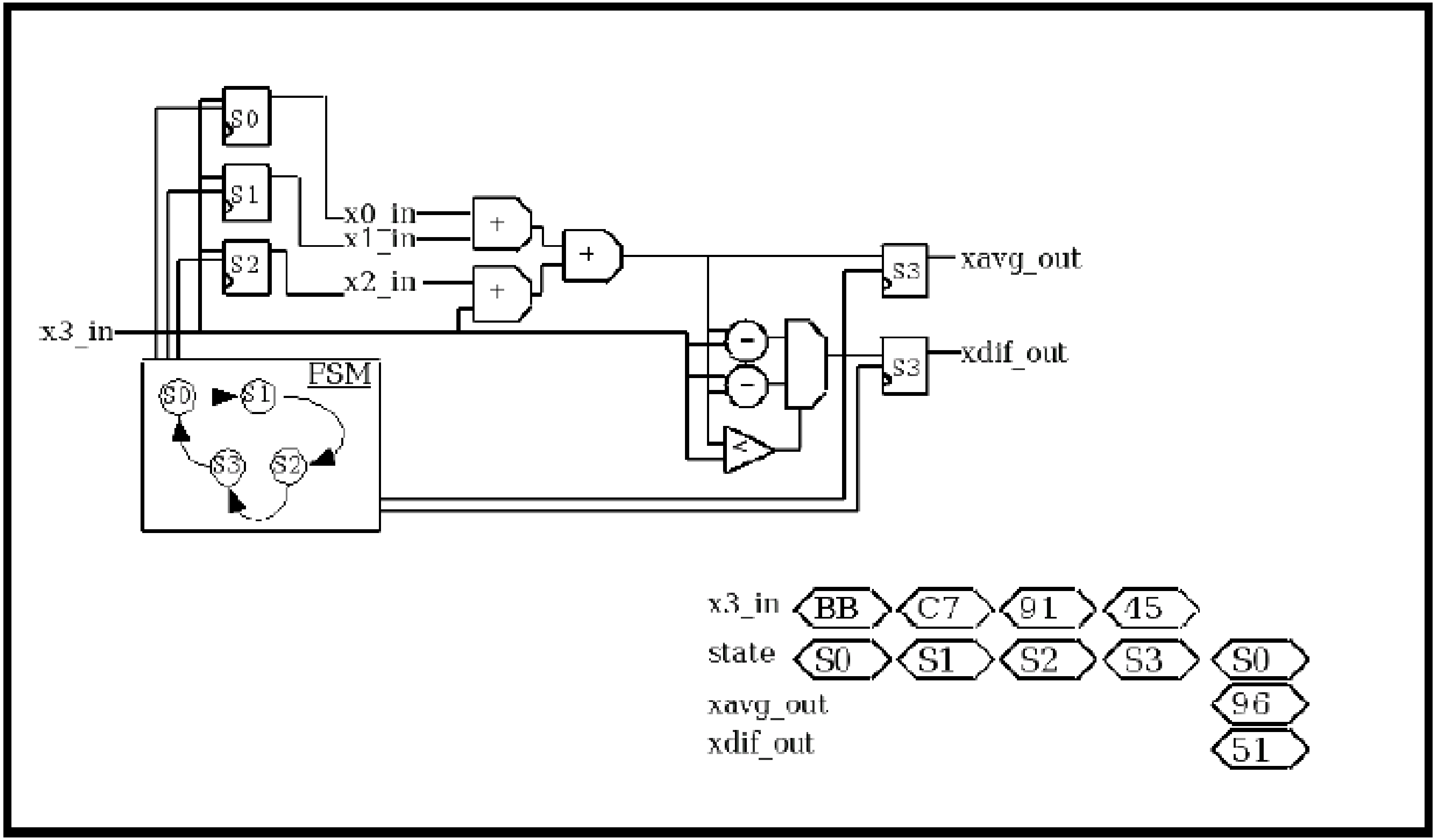
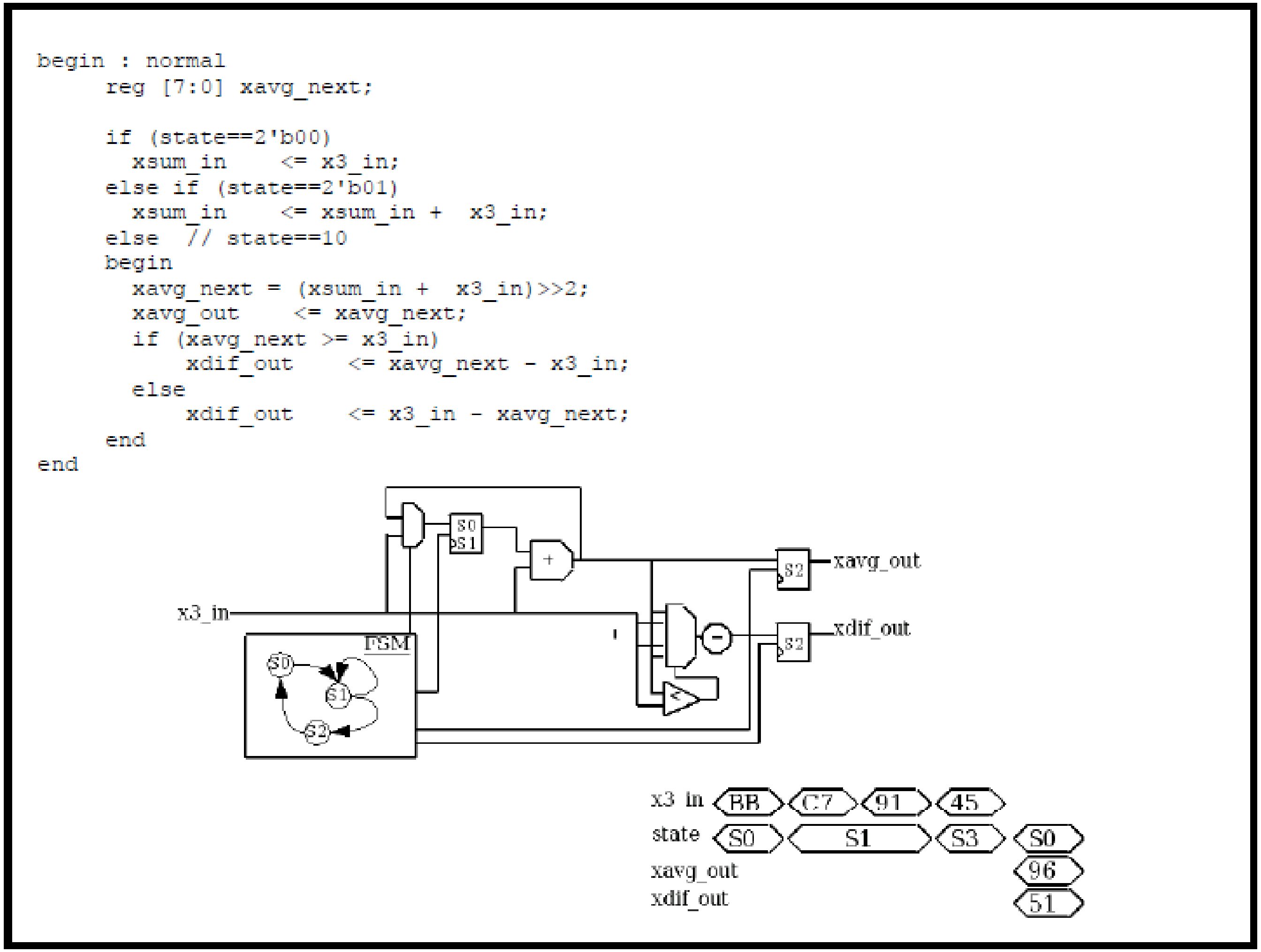


Figure 2: Serial implementation of Average4 circuit

Courtesy: Calypto Design



```
begin : normal
     reg [7:0] xavg_next;

     if (state==2'b00)
       xsum_in    <= x3_in;
     else if (state==2'b01)
       xsum_in    <= xsum_in +  x3_in;
     else  // state==10
     begin
       xavg_next = (xsum_in +  x3_in)>>2;
       xavg_out    <= xavg_next;
       if (xavg_next >= x3_in)
           xdif_out    <= xavg_next - x3_in;
       else
           xdif_out    <= x3_in - xavg_next;
     end
end
```

Figure 3: Power optimized Average4 design

Courtesy: Calypto Design

How to apply it to Avg4 equiv.?

- Identify an invariant (Inv) condition expected to hold at equivalent checking points
  - Inv(Ckt) = (Ckt.fsm_cntrl_state == S0)
- Properties to check:
  - Base: Inv and Output Equiv holds after reset
  - Induct:
    - Assume Inv holds for both ckts at time t
    - Check Inv and output equiv holds at (t+3)
- Exercise: Can K-induction be used to generate Inv?

Can a similar approach be used to verify uProcessor?

A Simple DSP

- Low-cost embedded DSP meant for processing
- Dual memory multiplier-accumulator architecture
  - 32-bit data path, 2K instrn mem, two 1K data mems, 32 registers
- 4-stage pipeline
- 135 instructions:
  - MUL, MAC, SHIFT
  - Direct and indirect memory addressing
  - Special-purpose address generation logic
  - CALL, RET, REP (loop body) instructions
  - Interrupt instructions
- RTL of DSP core: 2600 lines verilog (w/o mem & I/O)
  - ~30K gates (w/o memories)
- C-Model: ISA and ISS

Global Constraints:

Forall addr:
Assume imem[addr]
isVALID &&
(!(isOUT) | | isINT)

No constraints on
Xmem and ymem

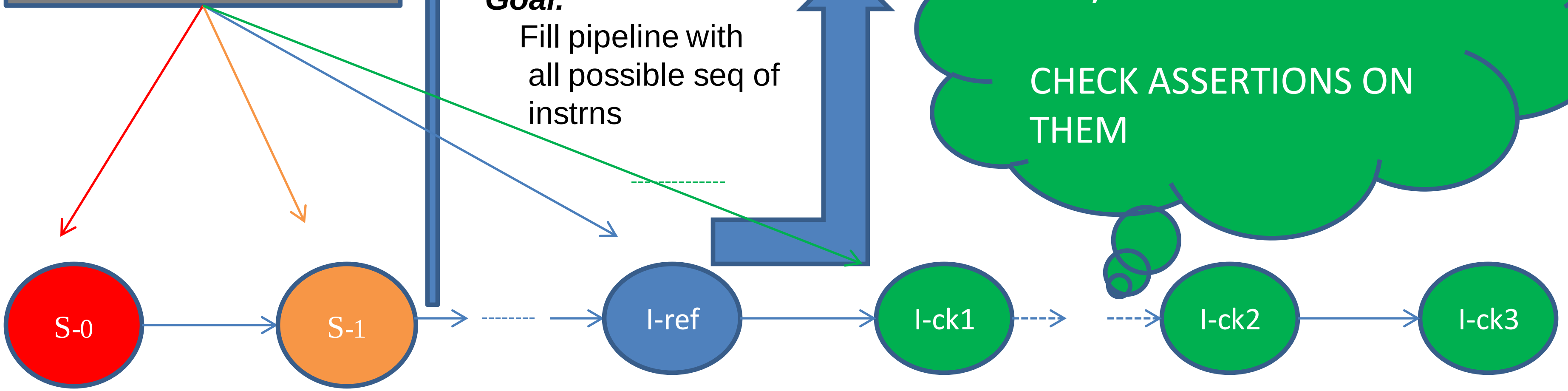
- I-ref denotes a SET of RTL states
- Captures result of 6 valid instructions after a reset
- Includes ALL possible ways in which pipeline can be filled with 4 possible instructions
- Covers ALL possible variations of every instruction
- Covers ALL possible combinations of UNCONSTRAINED input signals

Symbolic sim RT L
for n cycles
n >=4 (say 6)

Goal:
Fill pipeline with
all possible seq of
instrns

"How can we use result of symbolic simulation?"

CHECK ASSERTIONS ON THEM



Resetz==0 &&
core_en==1

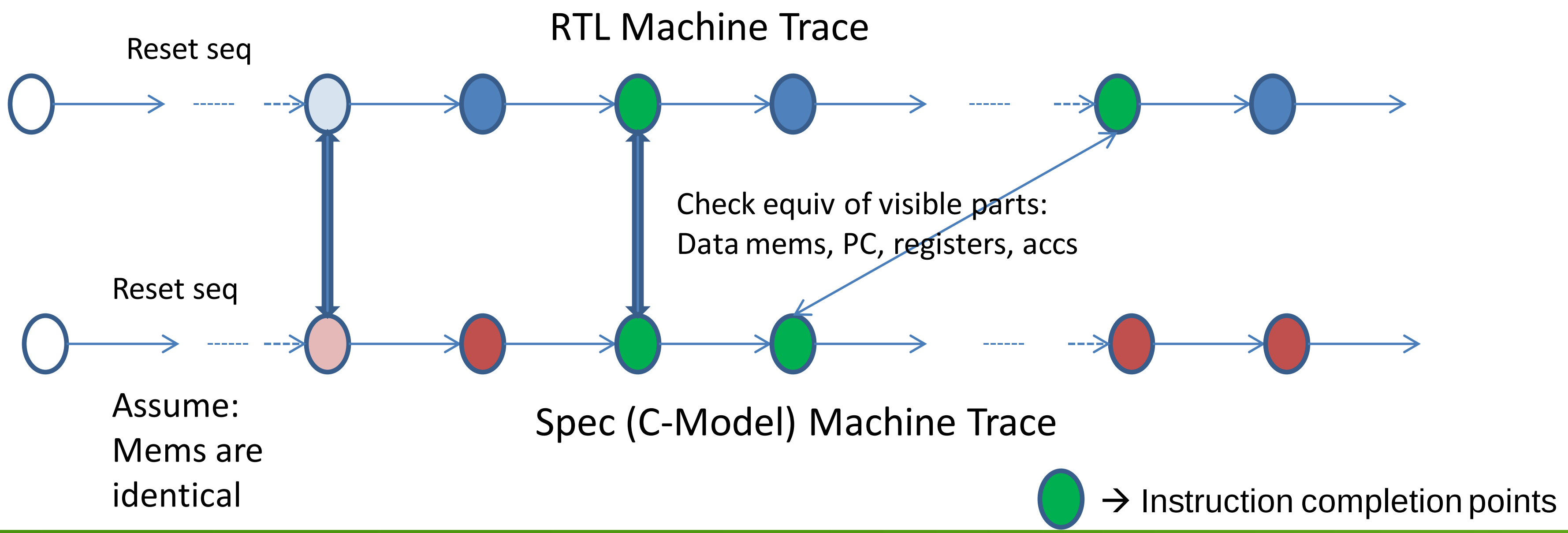
Resetz==1 &&
core_en==1 &&
Idata==
imem[prev_iaddr]

Resetz==1 &&
core_en==1 &&
Idata==
imem[prev_iaddr]

Resetz==1 &&
core_en==1 &&
Idata==
imem[prev_iaddr]

Microprocessor Verification: Problem Statement

- ***Sequential Equivalence b/w RTL and ISA machines: [SrivasMiller]***
  - For ***ALL identical*** sequences of instructions
- ***Challenges:***
  - ISA and RTL machines may complete instructions at ***different rates***
    - RTL is pipelined but ISA may not be
    - ISA is Pipelined, but may not be cycle accurate
  - ISA state is an ***abstraction*** of RTL state



RTL Machine Trace

Reset seq

Check equiv of visible parts:
Data mems, PC, registers, accs

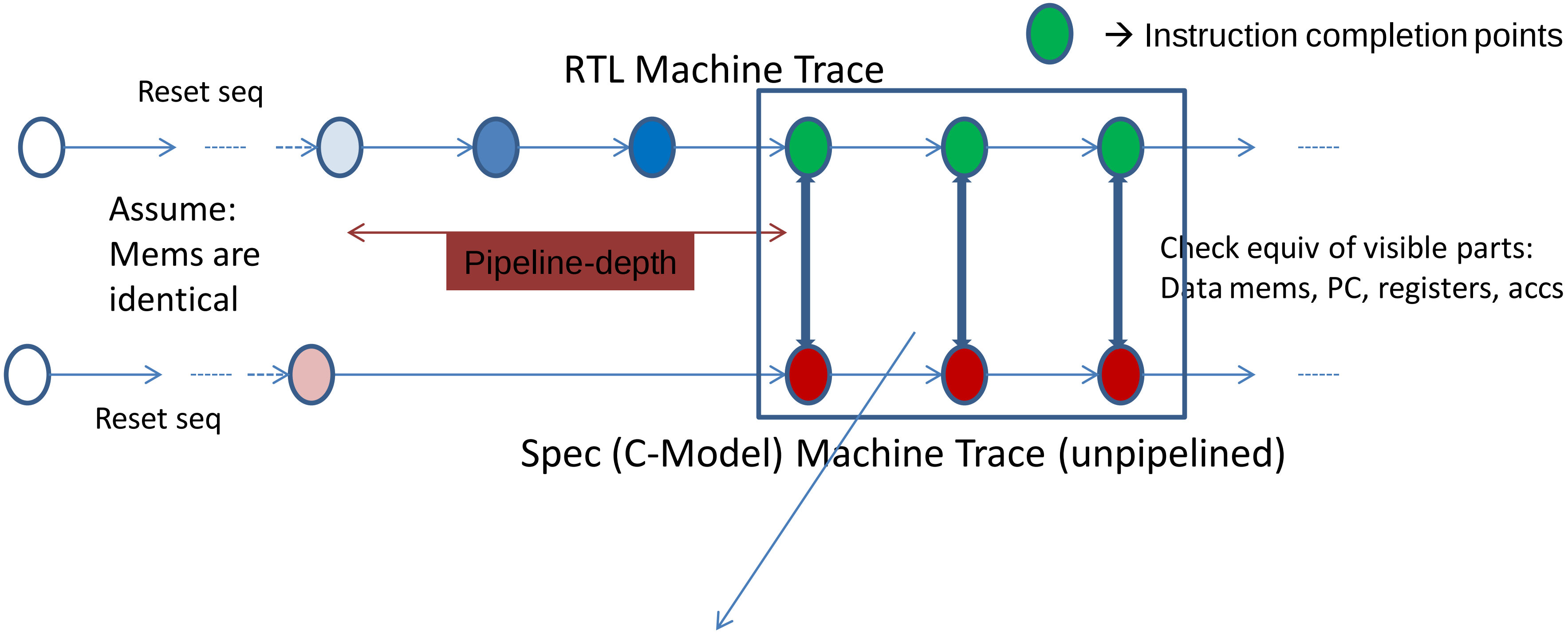
Reset seq

Assume:
Mems are
identical

Spec (C-Model) Machine Trace

● → Instruction completion points

Verifying Sequential Equivalence: Using "finite-distance" assertions

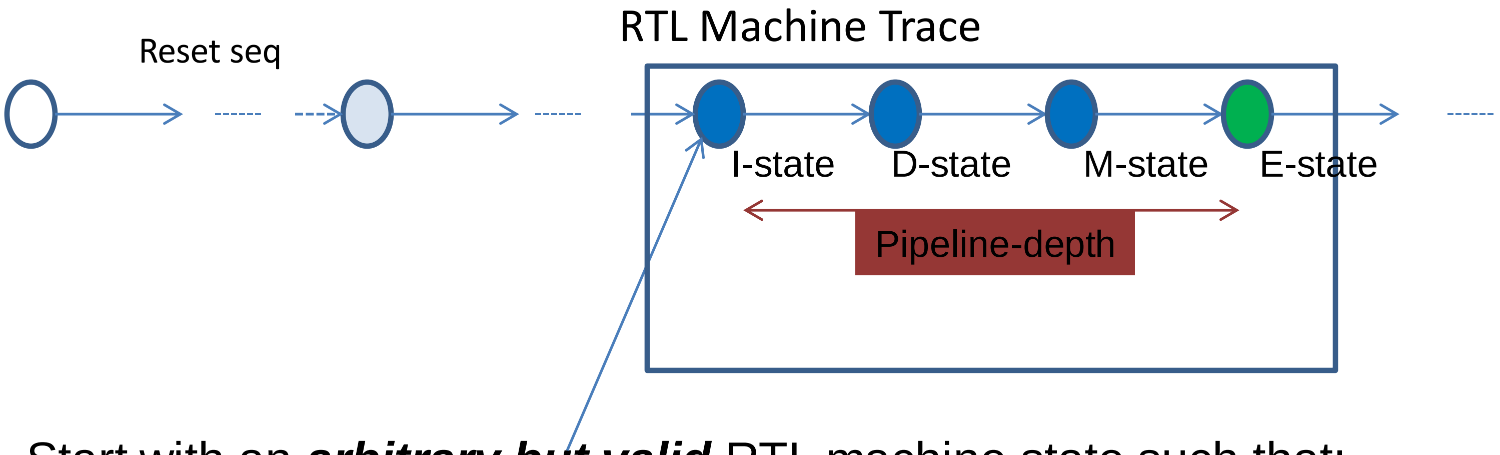

→ Instruction completion points
RTL Machine Trace
Reset seq

Assume:
Mems are
identical
Pipeline-depth
Check equiv of visible parts:
Data mems, PC, registers, accs
Reset seq
Spec (C-Model) Machine Trace (unpipelined)

In general, enough to check "pipeline-deep" assertions

- involving states that are separated at most pipeline depth apart
- on traces originating from an arbitrary pipeline state
  - with arbitrary sequence of instructions in flight

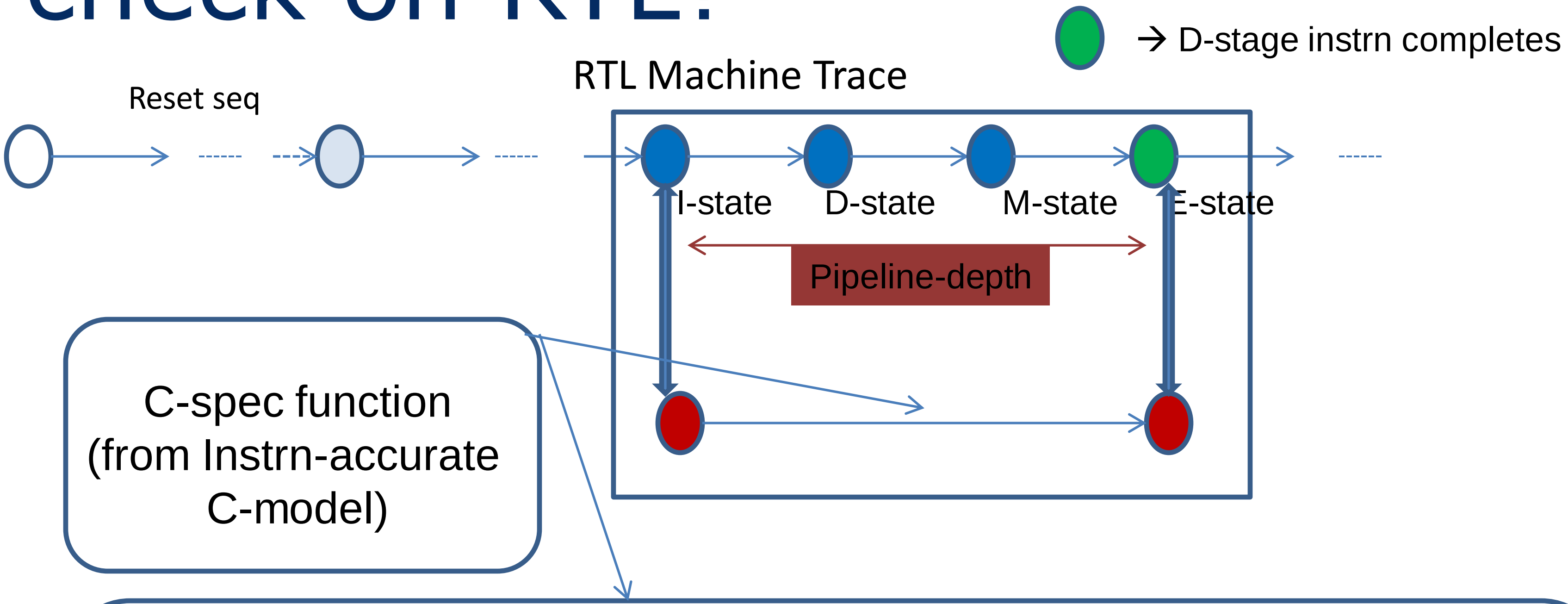
Finite-distance Inductive Assertions: What do they Assume?



Start with an ***arbitrary but valid*** RTL machine state such that:

- pipeline is filled with arbitrary sequence of legal instructions
  - F-stage, D-stage, M-stage, E-stage
- the instrns in the piepline satisfy all required pipeline restrictions

Finite-distance Inductive Assertions: What do they check on RTL?



● → D-stage instrn completes

RTL Machine Trace

Reset seq

I-state D-state M-state E-state

Pipeline-depth

C-spec function (from Instrn-accurate C-model)

Assert-1: The I-stage instrn (in I-state) will move to D-stage in (D-state)

Assert-2: Expected NEW instrn will be the I-stage instrn (in D-state)

Assert-3: Rest of visible state (mem, internal regs, ACC, etc.) in E-state will correspond as per the C-spec function

Instruction behavior correctness:
NOP cc s2

NOP-assert1: "The F-instrn in I-ref state will ALWAYS EVENTUALLY move to D-stage "

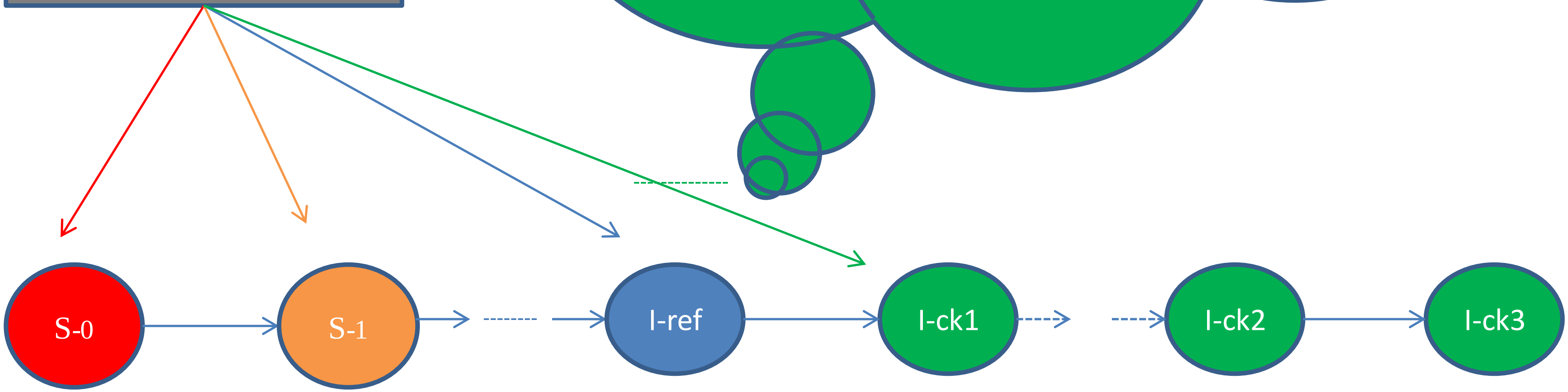
- ***in I-ck1 state, if !isNOP(D-instrn)***
- ***#wait-cycles later, if isNOP(D-instrn)***

•NOP-assert2: "Visible state MUST not change from Iref+2 to Iref+#wait-cycles

Global Constraints:

Forall addr:
Assume imem[addr]
isVALID &&
(!(isOUT)| | isINT)

No constraints on
Xmem and ymem



Resetz==0 &&
core_en==1

Resetz==0   &&
core_en==1   &&
Idata==
imem[prev_iaddr]

Resetz==0   &&
core_en==1   &&
Idata==
imem[prev_iaddr]

Resetz==0   &&
core_en==1   &&
Idata==
imem[prev_iaddr]

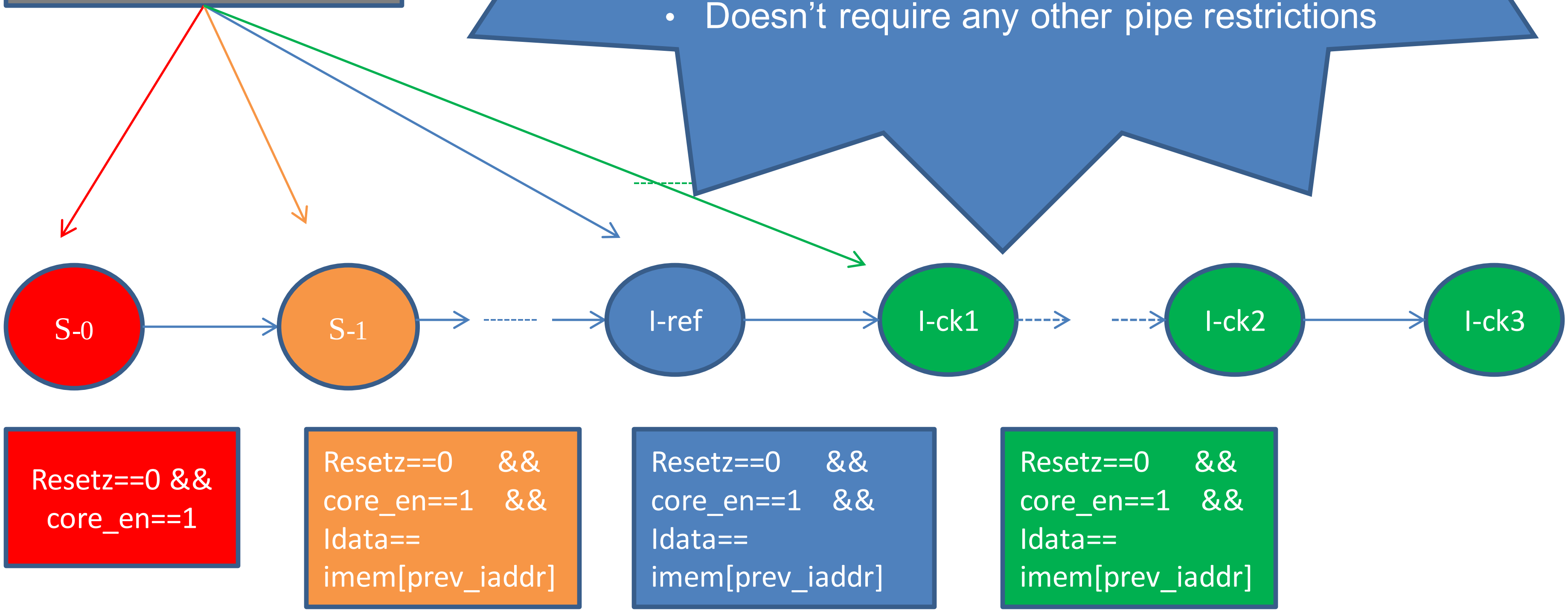
Global Constraints:

Forall addr:
Assume imem[addr]
isVALID &&
(!(isOUT) | | isINT)

No constraints on
Xmem and ymem

NOP-assert1 FV Revelations

- ***Requires Pipe-7 restrn as precondition***
  - "A condtnl ST instrn cannot have a non ST condtnl instrn in 2nd place after ST"
- ACC sign/zero condn change during NOP countdown, can change NOP cycles
  - #nop-cycles is also impacted by n-1 and n-2 instrns!!
  - is this expected or a BUG?
- Doesn't require any other pipe restrictions

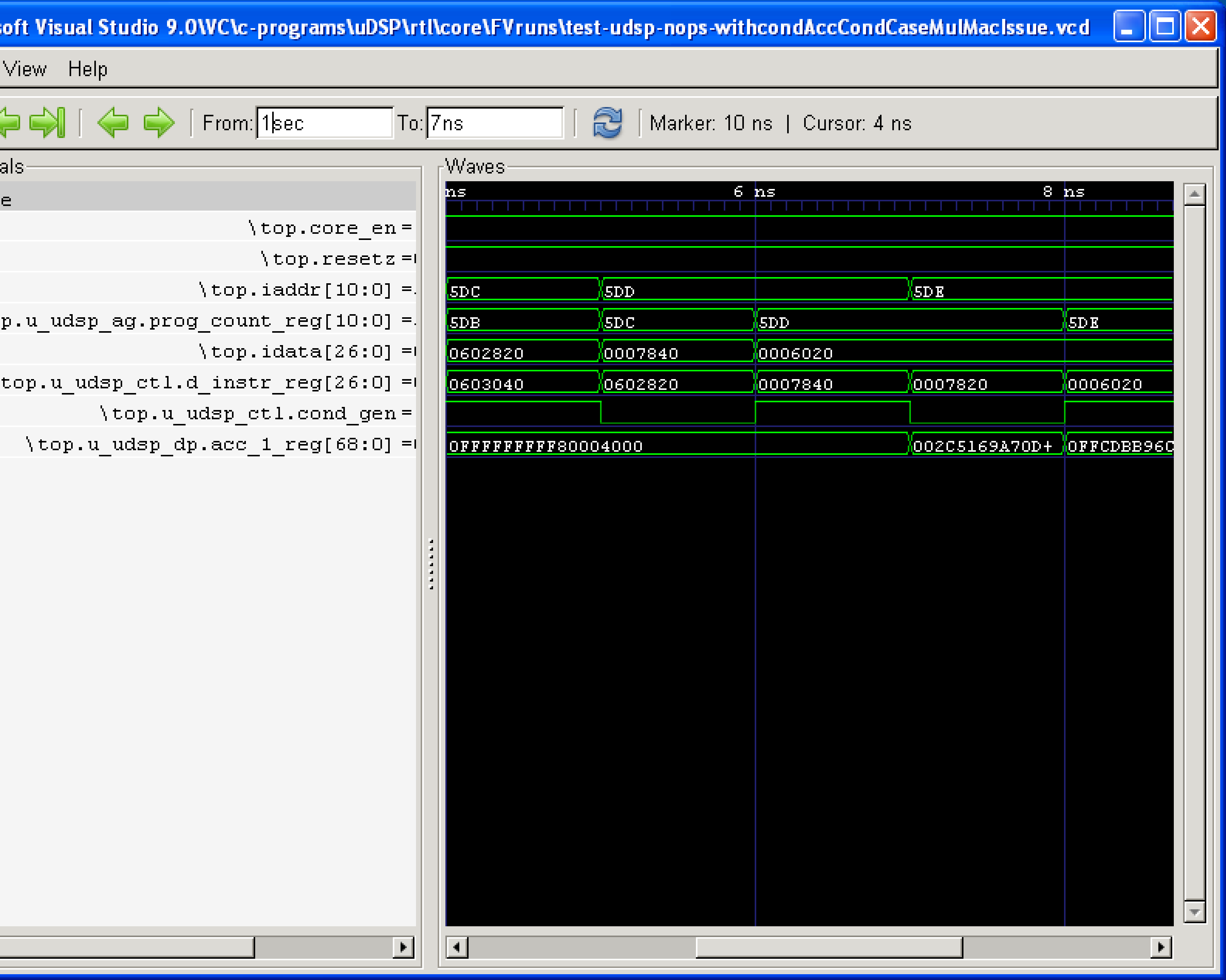


Resetz==0 &&
core_en==1

Resetz==0 &&
core_en==1 &&
Idata==
imem[prev_iaddr]

Resetz==0 &&
core_en==1 &&
Idata==
imem[prev_iaddr]

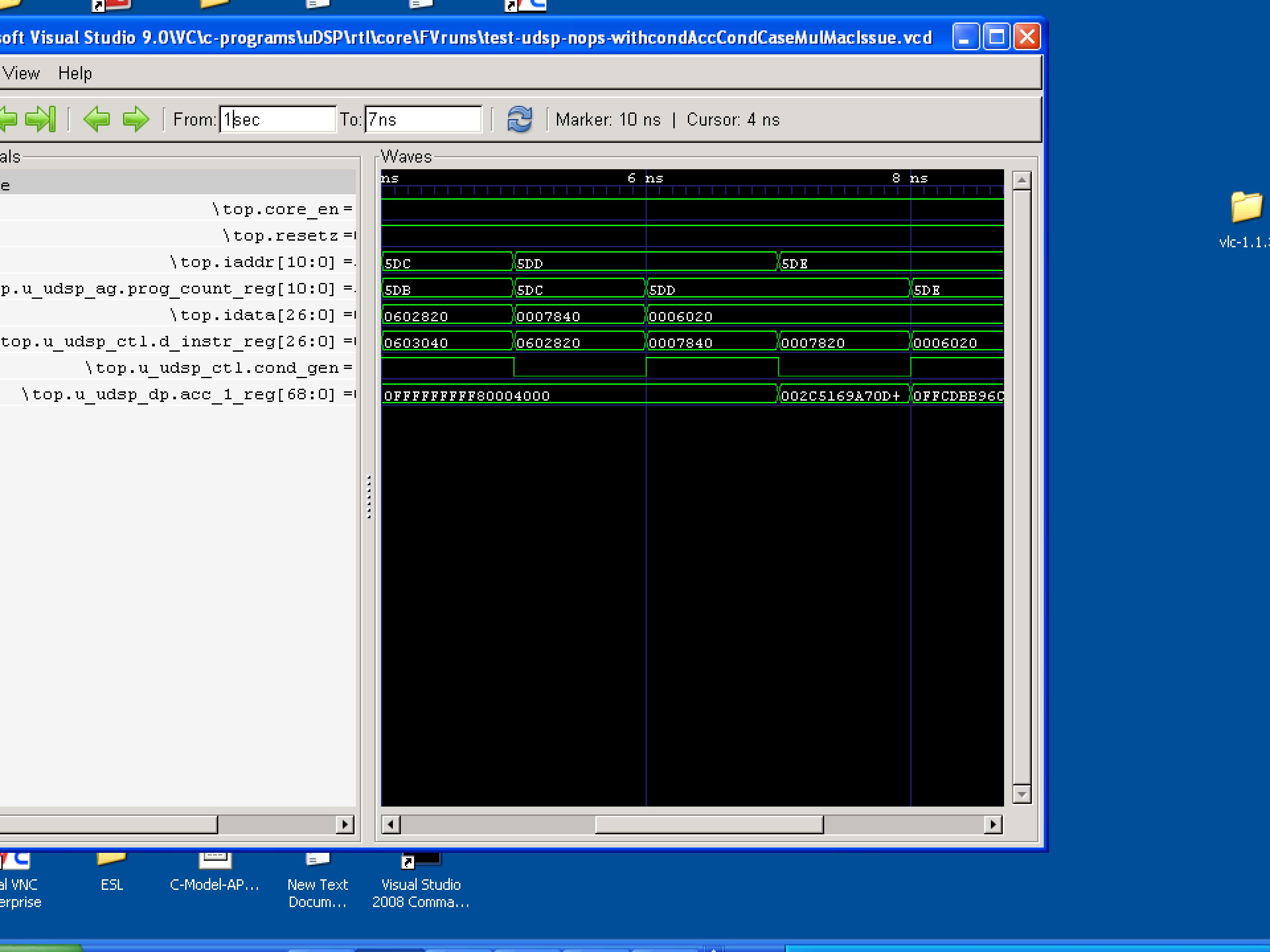
Resetz==0 &&
core_en==1 &&
Idata==
imem[prev_iaddr]



soft Visual Studio 9.0\VC\c-programs\uDSP\rtl\core\FVruns\test-udsp-nops-withcondAccCondCaseMulMacIssue.vcd

From: 1 sec To: 7ns | Marker: 10 ns | Cursor: 4 ns

```
\top.core_en =
\top.resetz =
\top.iaddr[10:0] =         5DC              5DD                        5DE
p.u_udsp_ag.prog_count_reg[10:0] =   5DB   5DC      5DD               5DE
\top.idata[26:0] =         0602820          0007840    0006020
top.u_udsp_ctl.d_instr_reg[26:0] =   0603040  0602820  0007840  0007820  0006020
\top.u_udsp_ctl.cond_gen =
\top.u_udsp_dp.acc_1_reg[68:0] =     0FFFFFFFF80004000           002C5169A70D+  0FFCDBB96C
```

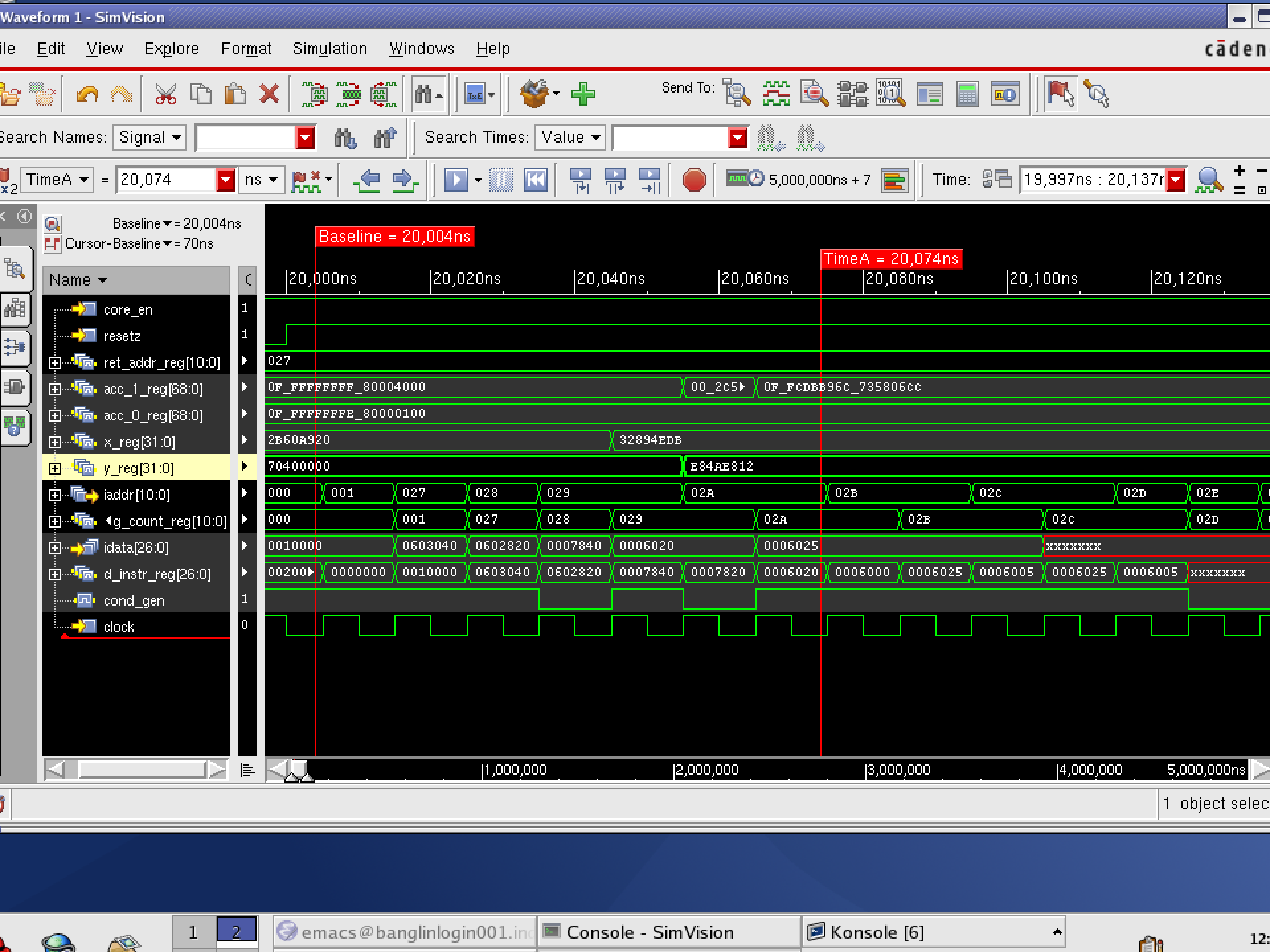


soft Visual Studio 9.0\VC\c-programs\uDSP\rtl\core\FVruns\test-udsp-nops-withcondAccCondCaseMulMacIssue.vcd

From: 1sec To: 7ns | Marker: 10 ns | Cursor: 4 ns

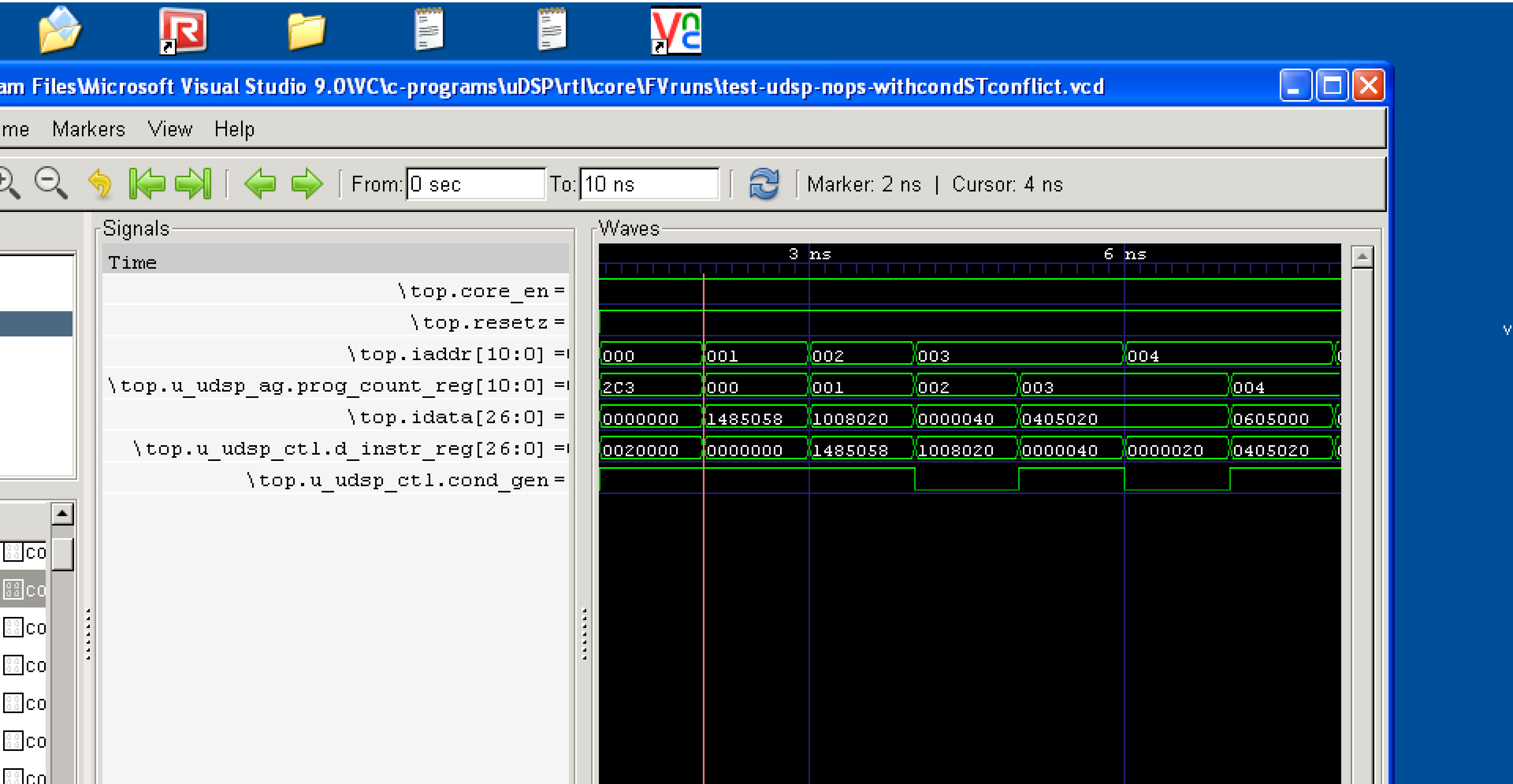
```
\top.core_en =
\top.resetz =
\top.iaddr[10:0] =
p.u_udsp_ag.prog_count_reg[10:0] =
\top.idata[26:0] =
top.u_udsp_ctl.d_instr_reg[26:0] =
\top.u_udsp_ctl.cond_gen =
\top.u_udsp_dp.acc_1_reg[68:0] =
```

```
iaddr:          5DC      5DD                        5DE
prog_count_reg: 5DB      5DC       5DD                        5DE
idata:          0602820  0007840   0006020
d_instr_reg:    0603040  0602820   0007840   0007820          0006020
acc_1_reg:      0FFFFFFFF80004000            002C5169A70D+    0FFCDBB96C
```



```
core_en            1
resetz             1
ret_addr_reg[10:0]     027
acc_1_reg[68:0]        0F_FFFFFFFF_80004000 | 00_2C5 | 0F_FCDBB96C_735806CC
acc_0_reg[68:0]        0F_FFFFFFFE_80000100
x_reg[31:0]            2B60A920 | 32894EDB
y_reg[31:0]            70400000 | E84AE812
iaddr[10:0]            000 | 001 | 027 | 028 | 029 | 02A | 02B | 02C | 02D | 02E
g_count_reg[10:0]      000 | 001 | 027 | 028 | 029 | 02A | 02B | 02C | 02D
idata[26:0]            0010000 | 0603040 | 0602820 | 0007840 | 0006020 | 0006025 | xxxxxxx
d_instr_reg[26:0]      00200 | 0000000 | 0010000 | 0603040 | 0602820 | 0007840 | 0007820 | 0006020 | 0006000 | 0006025 | 0006005 | 0006025 | 0006005 | xxxxxxx
cond_gen           1
clock              0
```

FV Error Trace: Conditional ST Issue



am Files\Microsoft Visual Studio 9.0\VC\c-programs\uDSP\rtl\core\FVruns\test-udsp-nops-withcondSTconflict.vcd

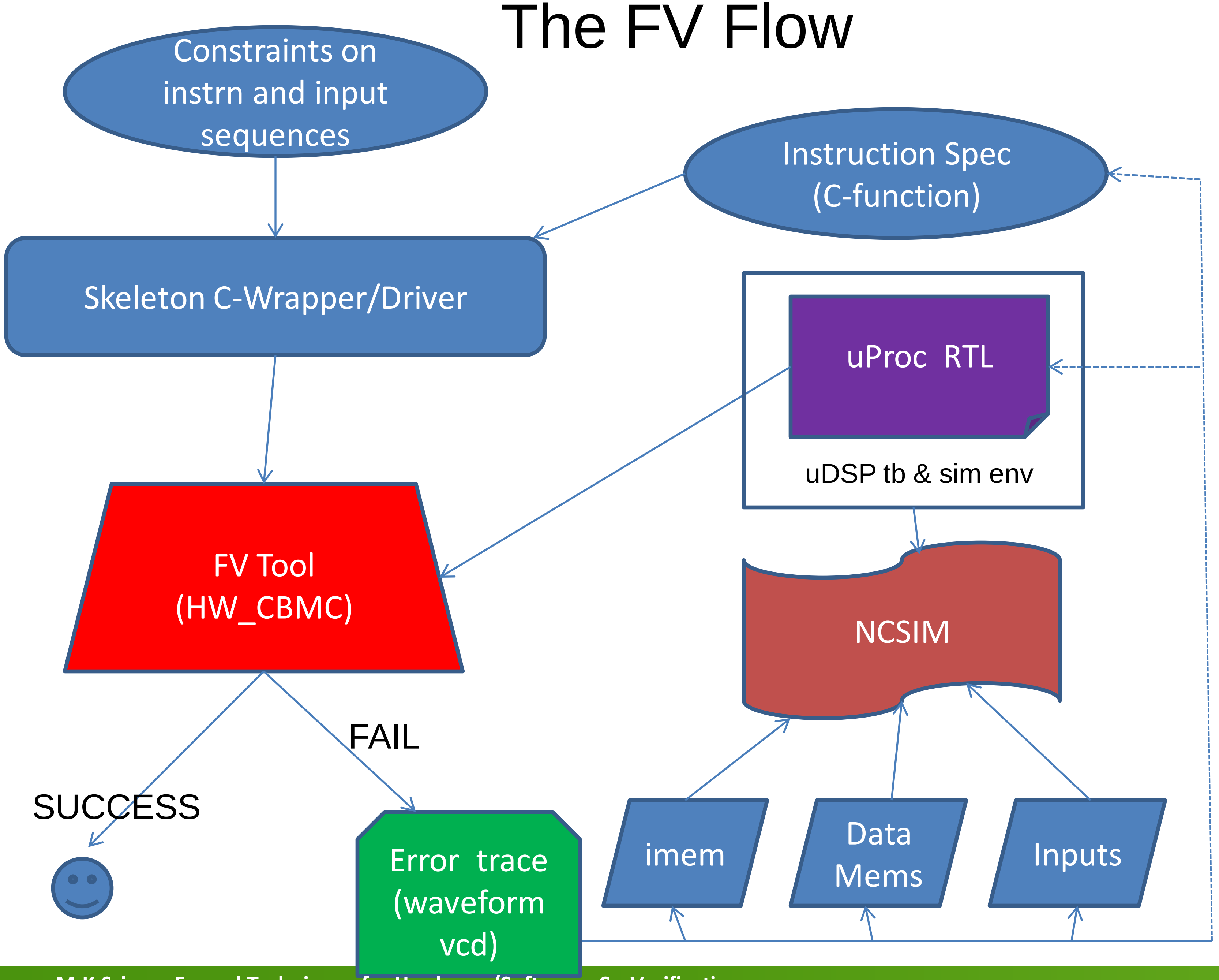
From: 0 sec To: 10 ns | Marker: 2 ns | Cursor: 4 ns

```
Time
\top.core_en =
\top.resetz =
\top.iaddr[10:0] =      000      001      002      003               004
\top.u_udsp_ag.prog_count_reg[10:0] =   2C3      000      001      002      003               004
\top.idata[26:0] =      0000000  1485058  1008020  0000040  0405020           0605000
\top.u_udsp_ctl.d_instr_reg[26:0] =     0020000  0000000  1485058  1008020  0000040  0000020  0405020
\top.u_udsp_ctl.cond_gen =
```

Advantages of FV for uProcessors

- Coverage: A single symbolic FV run covers
  - All possible variations of instruction under test and external conditions
    - Eg., for NOP all variations of CC and #op cycles (up to a limit)
    - Eg., for CTL, all possible CC conditions, target values
  - All possible combinations of sequences of 4 instrns in flight in the pipeline
    - With and without pipeline restriction
  - All possible combinations of two instructions following instrn under test
- Counter-example (→ reduced debug time)
  - Generates an offending error trace (waveform), if assertion fails
  - Error trace has all information needed to reproduce in simulation
- Large reduction in number of test-cases
  - Roughly one assertion per major class of instructions (15 – 20 classes)
    - Eg., all CTL instructions and their variations handled by a single assertion
    - Eg., all LD instrns should be possible to do with a single assertion
  - Verif plan: 162 test cases with >= 14788 variations

The FV Flow



Constraints on instrn and input sequences
Instruction Spec (C-function)
Skeleton C-Wrapper/Driver
uProc RTL
uDSP tb & sim env
FV Tool (HW_CBMC)
NCSIM
FAIL
SUCCESS
Error trace (waveform vcd)
imem
Data Mems
Inputs

REFERENCES

A. Biere, A. Cimatti, E. Clarke, and Y. Yhu, Symbolic model checking without BDDs, In “Tools for Algorithms for Construction and Analysis of Systems, pp 193-207, 1999.

Mandayam K. Srivas and Steve P. Miller, Applying Formal Verification to the AAMP5 Microprocessor: A Case-study in the Industrial Use of Formal Methods, Formal Methods n System Design, vol.8, No.2, pp 153-188, 1996.

R. Hosabettu, M.K. Srivas, and G. Gopalakrishnan, Proof of Correctness of a Processor with Reorder Buffer using the Completion Functions Approach, CAV 1999.

A. Donaldson, L. Haller, D. Kroening, Software Verification Using K-Induction, Static Analysis Symposium 2011, LNCS, pp. 351-368, Springer

Armin Biere, Cyrille Artho, Viktor Schuppan, Liveness Checking as Safety Checking, Electronic Notes in Theoretical Computer Science 66 No. 2 (2002), URL: http://www.elsevier.nl/locate/entcs/volume66.html 18 pages