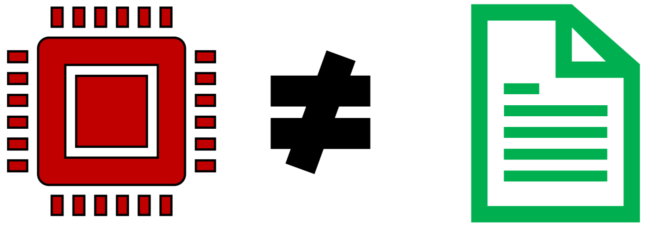
Formal Verification for Hardware Security

Mengjia Yan

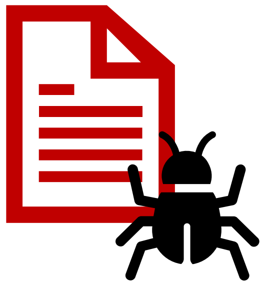
Spring 2025

Slides credit: Sharad Malik (Princeton)

Recall Hardware Bugs

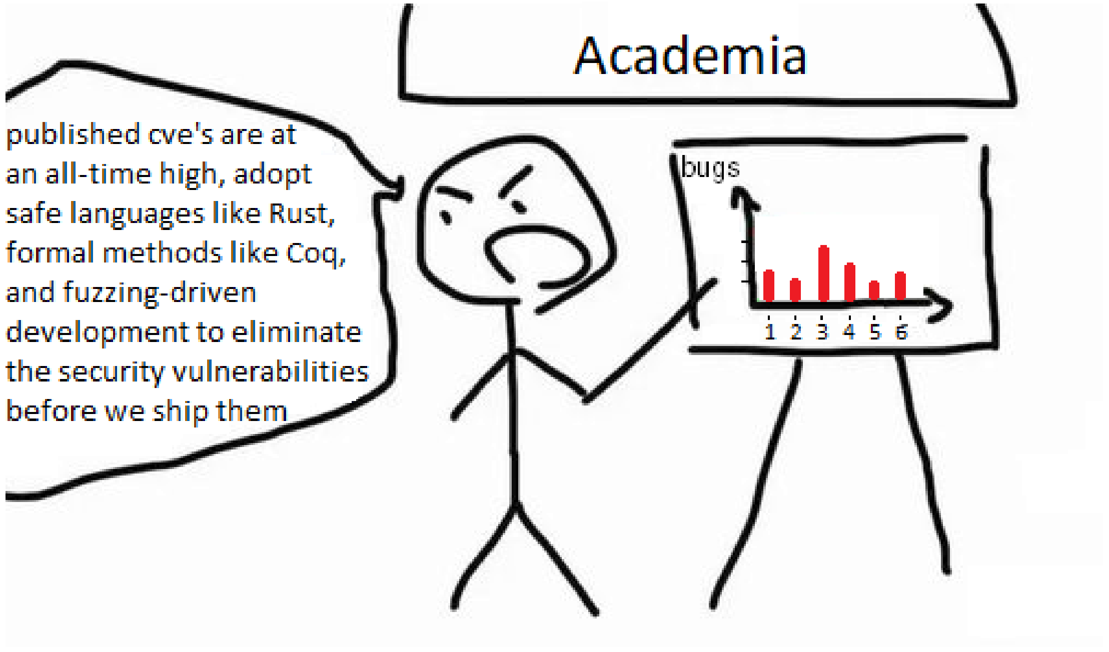


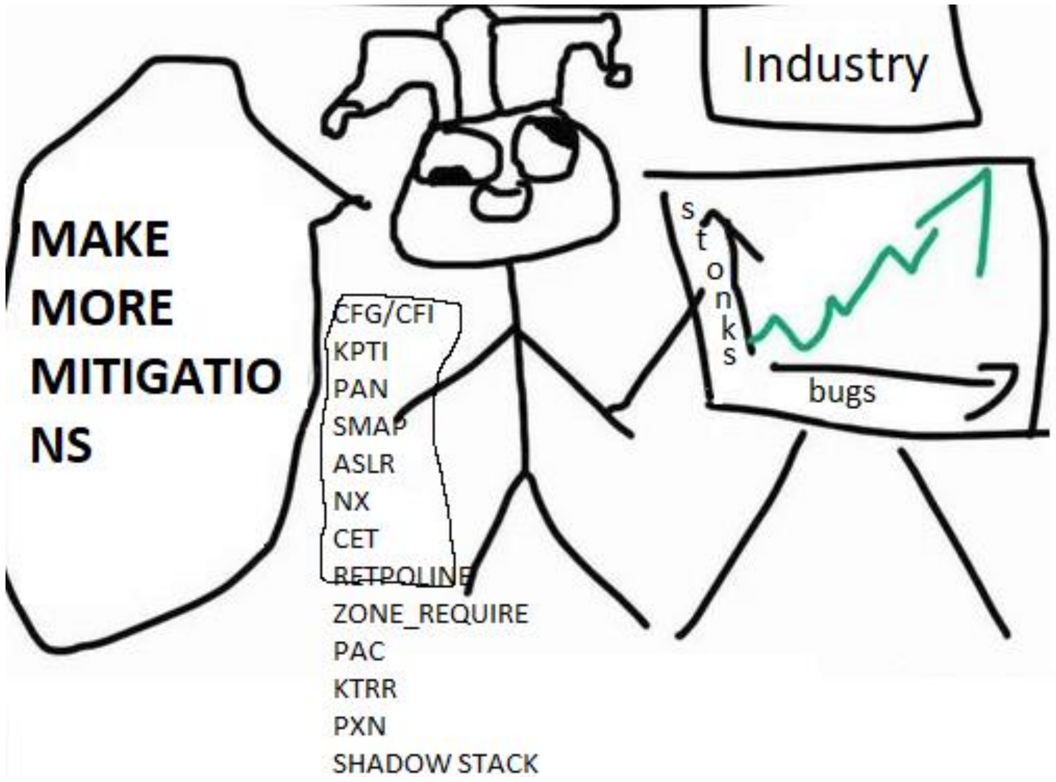
Implementation does not match specification (Errata)



Bugs in the specification

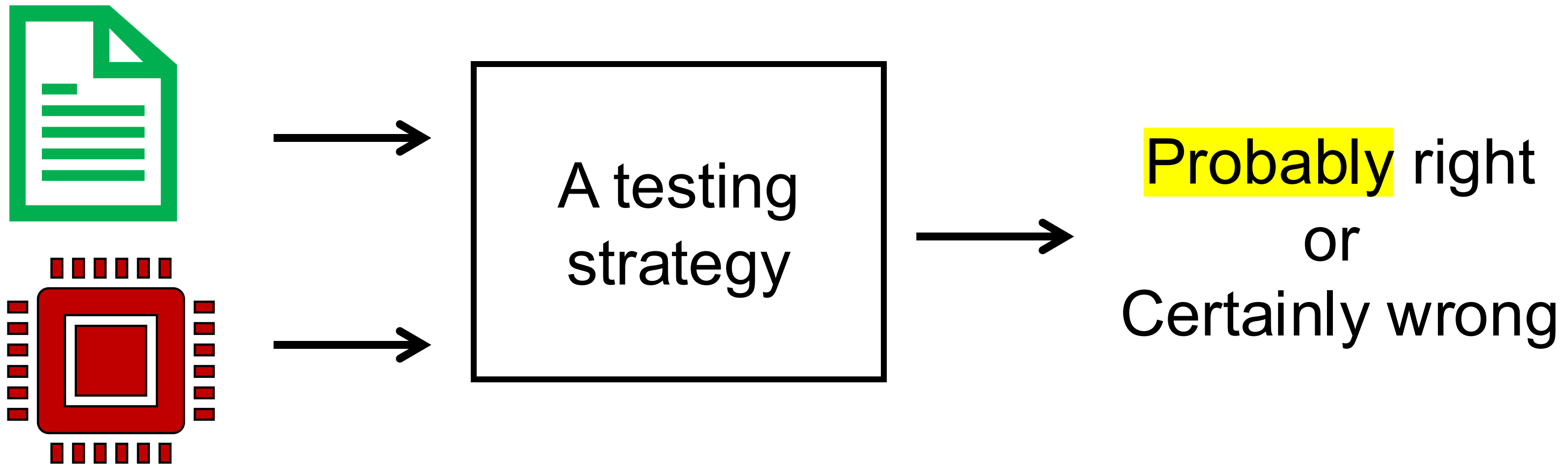
Vague specification


Academia
published cve's are at an all-time high, adopt safe languages like Rust, formal methods like Coq, and fuzzing-driven development to eliminate the security vulnerabilities before we ship them
bugs
1 2 3 4 5 6



https://twitter.com/gf_256/status/1321677851633029120/

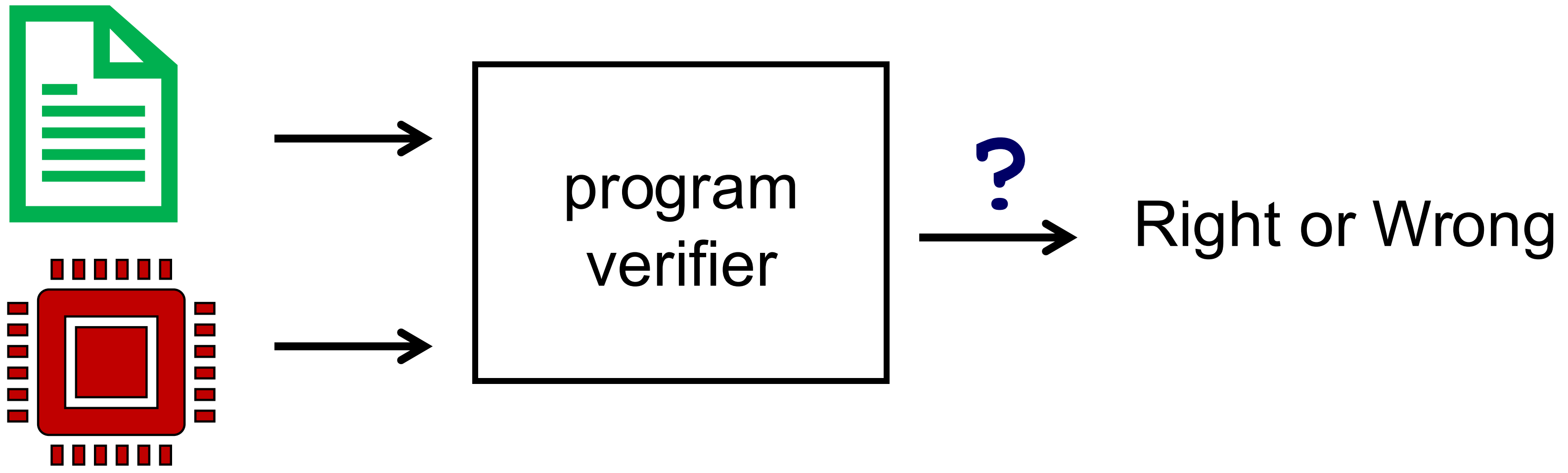
Program/Design Testing



Program testing can be quite effective for showing the presence of bugs, but is hopelessly inadequate for showing their absence.
– Edsger Dijkstra

- In principle: ***Exhaustive*** testing can prove correctness
- In practice: Test cases are generated to cover ***some* (not *all*)** inputs/statements/branches/paths etc.

Program/Design Verification



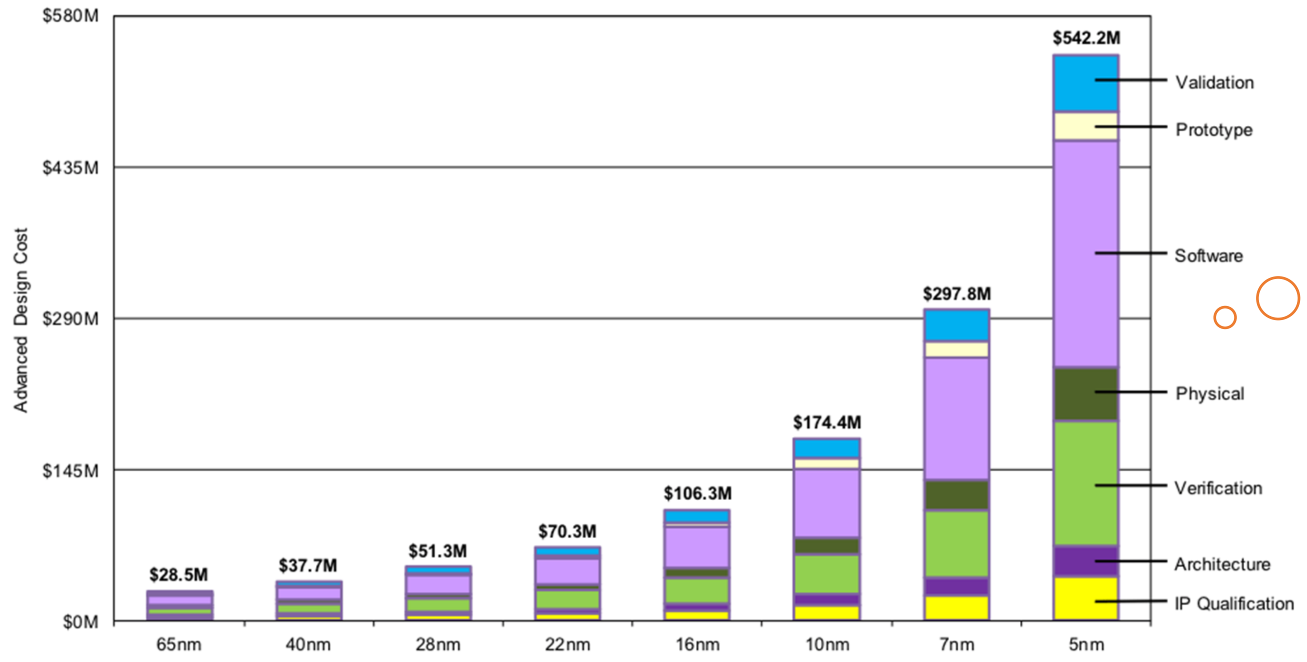
The goal: (under some conditions), program verifier

- can provide a proof (if program is right)
- or provide a counterexample (if program is wrong)

Formal Verification

"Verification": formally **prove** that the program/design is correct

- Rigor: uses well established mathematical foundations
- Exhaustiveness: considers all possible program behaviors
- Automation: uses computers to verify programs!

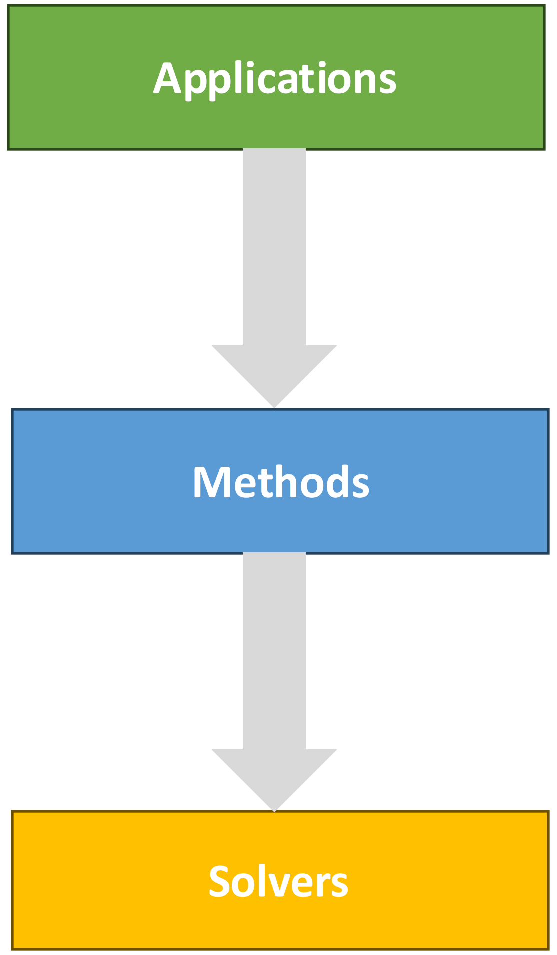


In many contexts, the term verification can be used in a loose way.

Design costs at recent nodes. Source: Handel Jones, IBS

Overall, it is a search problem…

How does formal verification work?



Program verification,
program synthesis,
test generation, etc.

Some SystemVerilog Code
+
Assertion check for
specification violation

Symbolic execution,
model checking,
invariant generation, etc.

(! (= a (* 2 (+ 10 b)))))

SAT, SMT, BDDs,
proof systems, etc.

Symbolic Execution: A Simple Example #1

C code:

```
int hash(int z){
   return (z+10)*2;
}

int obscure(int x, int y)
{
   if (x==hash(y))
      assert(false);
   return 1;
}
```

Rosette code:

```
(define (hash z)
  (* (+ z 10) 2)
  )

(define (obscure x y)
  (if (= x (hash y))
         (assert #t)
         (- x y))
  )
```

How will fuzzing behave to find this error?

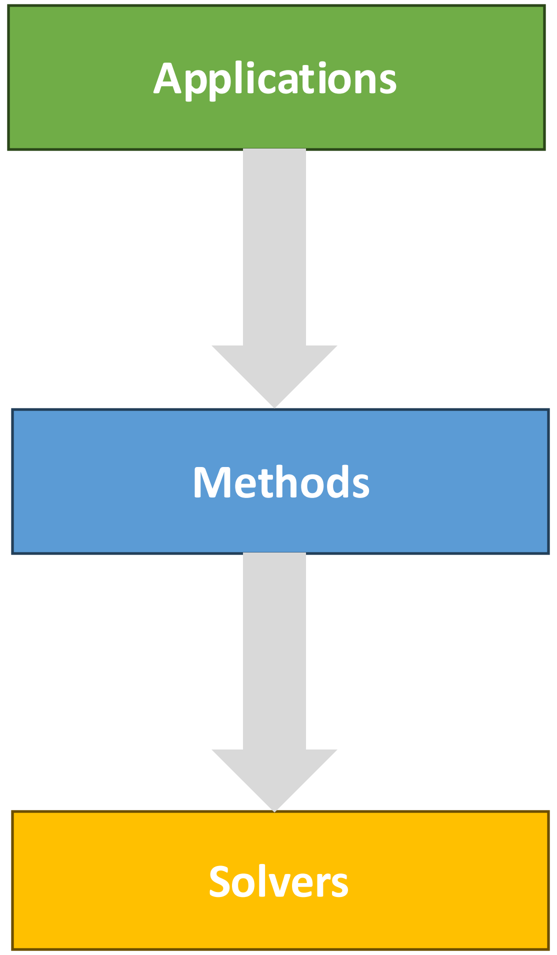
A Simple Example #2

```
int hash2(int z){
   if (z>10)
      z = z-10;
   return z;
}

int obscure(int x, int y)
{
   if (x==hash2(y))
      error();
   return x-y;
}
```

- Build execution tree with all the execution paths
- Each execution path has logical formula to describe path conditions
- The common pitfall: extremely large formula -> memory overhead and scalability issue

How does formal verification work?



Program verification, program synthesis, test generation, etc.

```
int hash2(int z){
   if (z>10)
      z = z-10;
   return z;
}

int obscure(int x, int y)
{
   if (x==hash2(y))
      error();
   return x-y;
}
```

=> Linux kernel, crypto libraries, processor Verilog code…

Symbolic execution, model checking, invariant generation, etc.

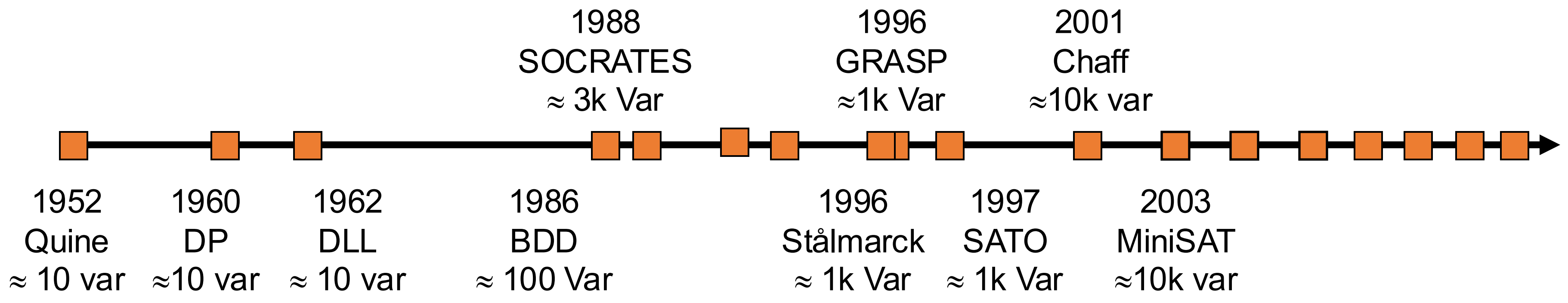
```
(! (= a (* 2 (+ 10 b))))
```

SAT, SMT, BDDs, proof systems, etc.

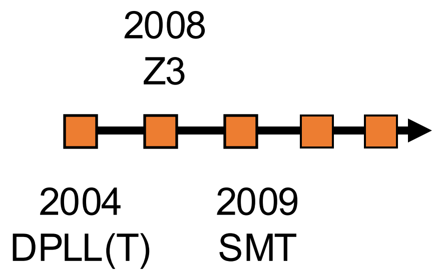
Success with SAT is at the heart of formal reasoning about systems.

Big Advancements in the Past Decade

(1) SAT: is a Boolean formula f satisfiable?



(2) SMT (Satisfiability Modulo Theory): is a first-order logic formula theory-satisfiable?

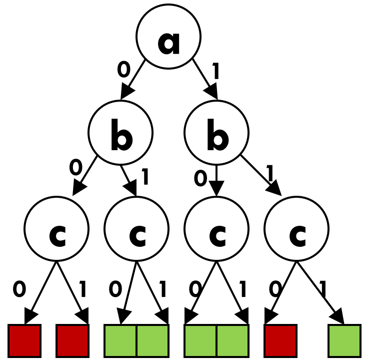


SAT in a Nutshell

- Given a propositional logic (Boolean) formula, find a variable assignment such that the formula evaluates to 1, or prove that no such assignment exists.

$$F = (a + b)(a' + b' + c)$$

- For *n* variables, there are 2^n possible truth assignments to be checked.



- First established NP-Complete problem.

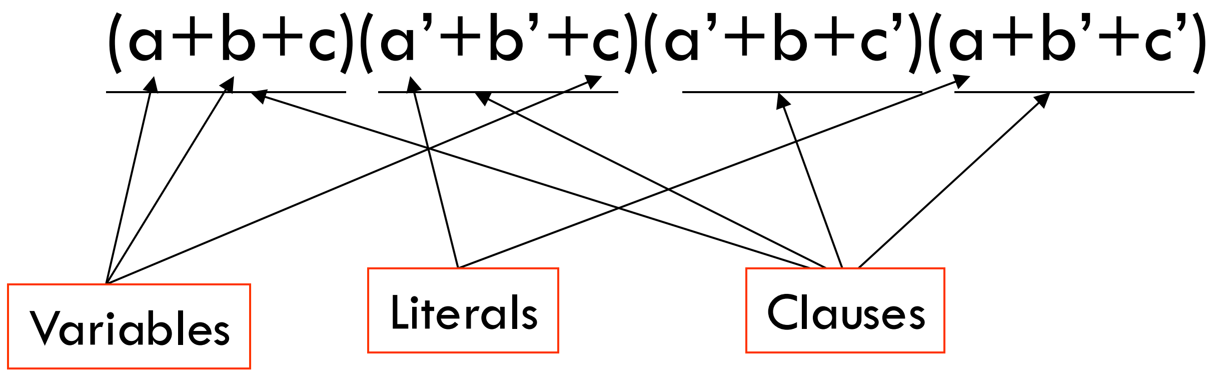
  S. A. Cook, The complexity of theorem proving procedures, *Proceedings, Third Annual ACM Symp. on the Theory of Computing*,1971, 151-158

Where are we today?

- Complexity of SAT: NP-complete
  - But often tractable in practice
- Intractability of the problem no longer daunting
  - Can regularly handle practical instances with millions of variables and constraints
- SAT has matured from theoretical interest to practical impact
  - Electronic Design Automation (EDA)
    - Widely used in many aspects of chip design
  - Increasing use in software verification
    - Commercial use at Microsoft, Amazon,…

Problem Representation

- Conjunctive Normal Form (CNF)
  - Representation of choice for modern SAT solvers
  - Every clause needs to be evaluated to TRUE



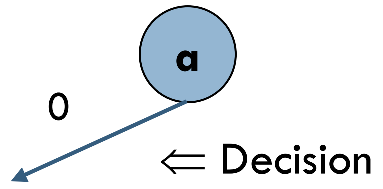
SAT Solvers: A Condensed History

- Deductive
  - Davis-Putnam 1960 [DP]
  - Iterative existential quantification by “resolution”
- Backtrack Search
  - Davis, Logemann and Loveland 1962 [DLL]
  - Exhaustive search for satisfying assignment
- Conflict Driven Clause Learning [CDCL]
  - GRASP: Integrate a constraint learning procedure, 1996
- Locality Based Search
  - Emphasis on exhausting local sub-spaces, e.g. Chaff, Berkmin, miniSAT and others, 2001 onwards
  - Added focus on efficient implementation
- “Pre-processing”
  - Peephole optimization, e.g. miniSAT, 2005

We cover these two algorithms to give you a taste of how the search works.

Basic DLL Search

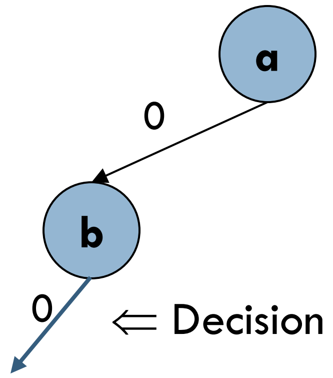
→ (a' + b + c)
(a + c + d)
(a + c + d')
(a + c' + d)
(a + c' + d')
(b' + c' + d)
→ (a' + b + c')
→ (a' + b' + c)



M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Communications of the ACM, 5:394–397, 1962*

Basic DLL Search

(a' + b + c)
(a + c + d)
(a + c + d')
(a + c' + d)
(a + c' + d')
→ (b' + c' + d)
(a' + b + c')
(a' + b' + c)



Basic DLL Search

(a’ + b + c)

(a + c + d)

(a + c + d’)

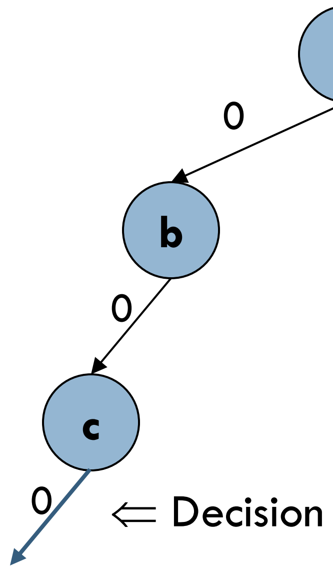
→ (a + c’ + d)

→ (a + c’ + d’)

(b’ + c’ + d)

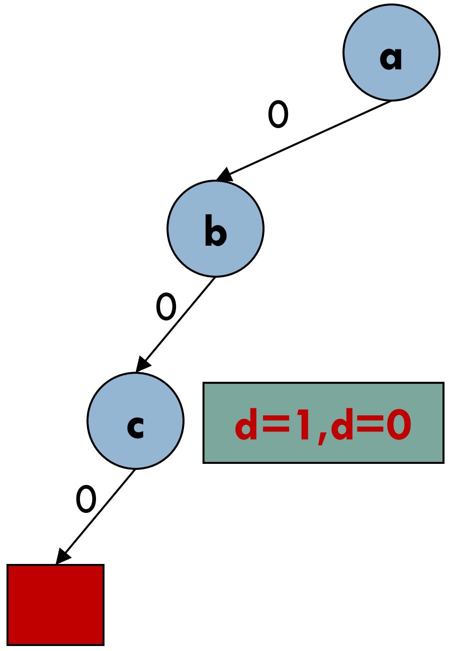
(a’ + b + c’)

(a’ + b’ + c)

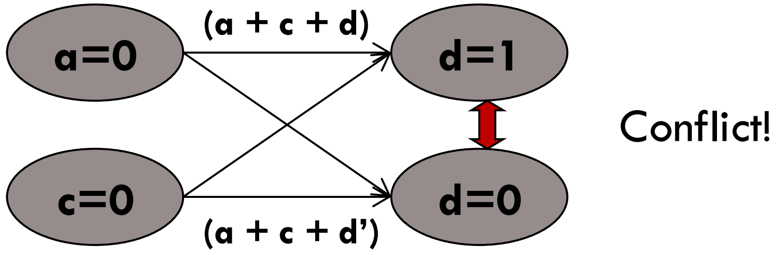


Basic DLL Search

	(a' + b + c)
→	(a + c + d)
→	(a + c + d')
	(a + c' + d)
	(a + c' + d')
	(b' + c' + d)
	(a' + b + c')
	(a' + b' + c)

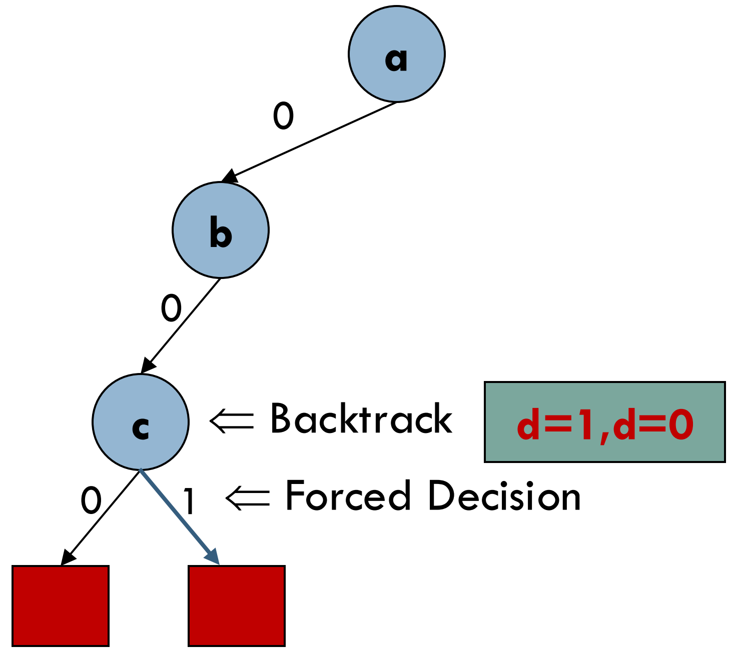


Implication Graph



Basic DLL Search

(a' + b + c)
(a + c + d)
(a + c + d')
→ (a + c' + d)
→ (a + c' + d')
(b' + c' + d)
(a' + b + c')
(a' + b' + c)



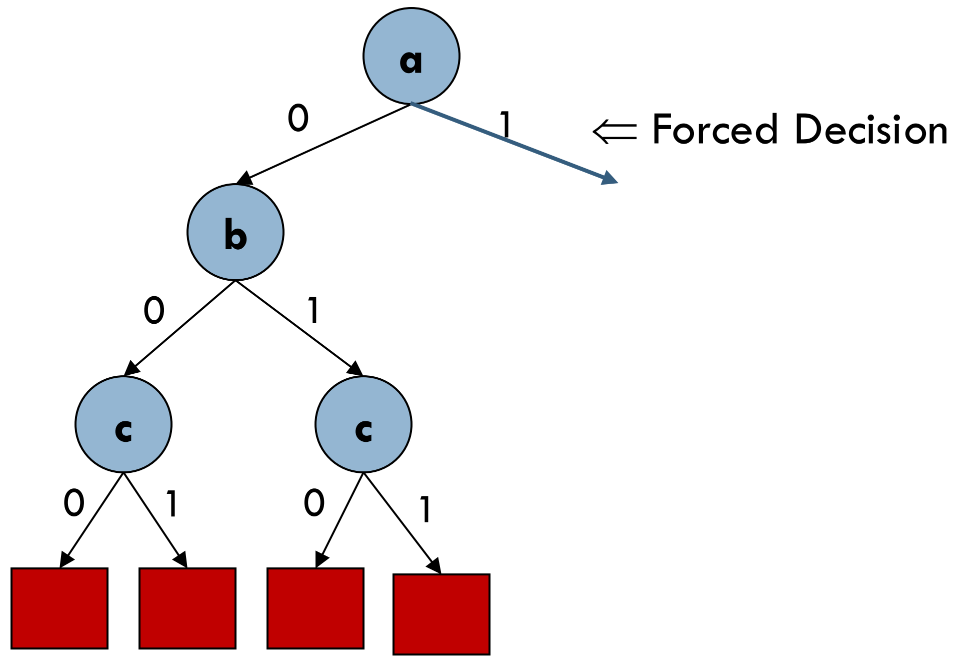
Think about the search performance:

- Do you always need to reach to the bottom to detect a conflict?

How fast a conflict is detected. Order matters.

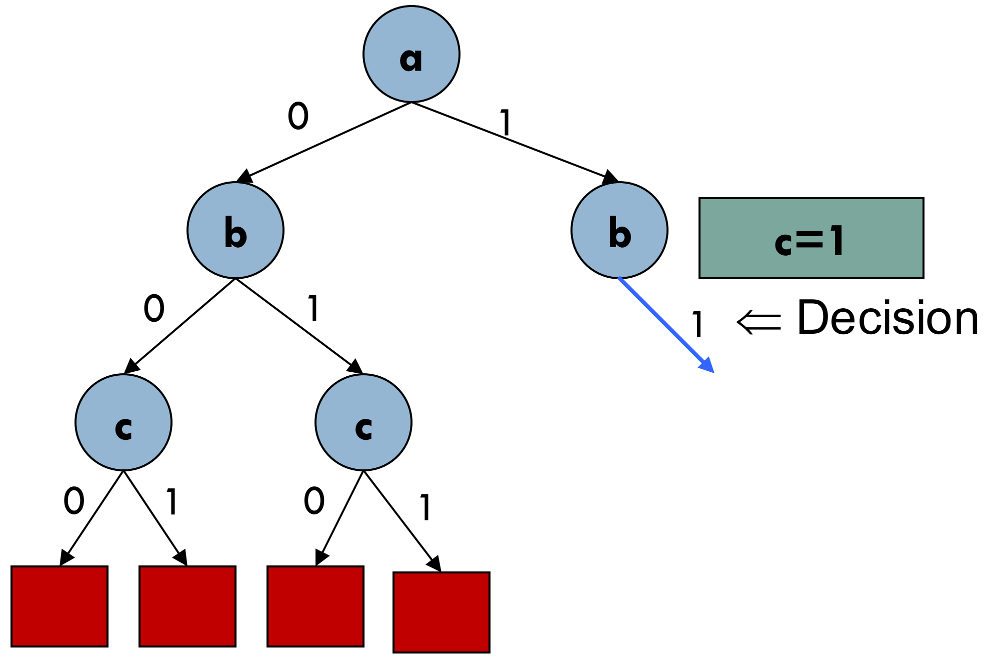
Basic DLL Search

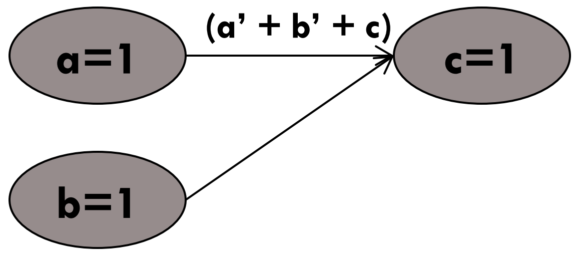
	(a' + b + c)
→	(a + c + d)
→	(a + c + d')
→	(a + c' + d)
→	(a + c' + d')
	(b' + c' + d)
	(a' + b + c')
	(a' + b' + c)



Basic DLL Search

→ (a' + b + c)
(a + c + d)
(a + c + d')
(a + c' + d)
(a + c' + d')
(b' + c' + d)
→ (a' + b + c')
→ (a' + b' + c)

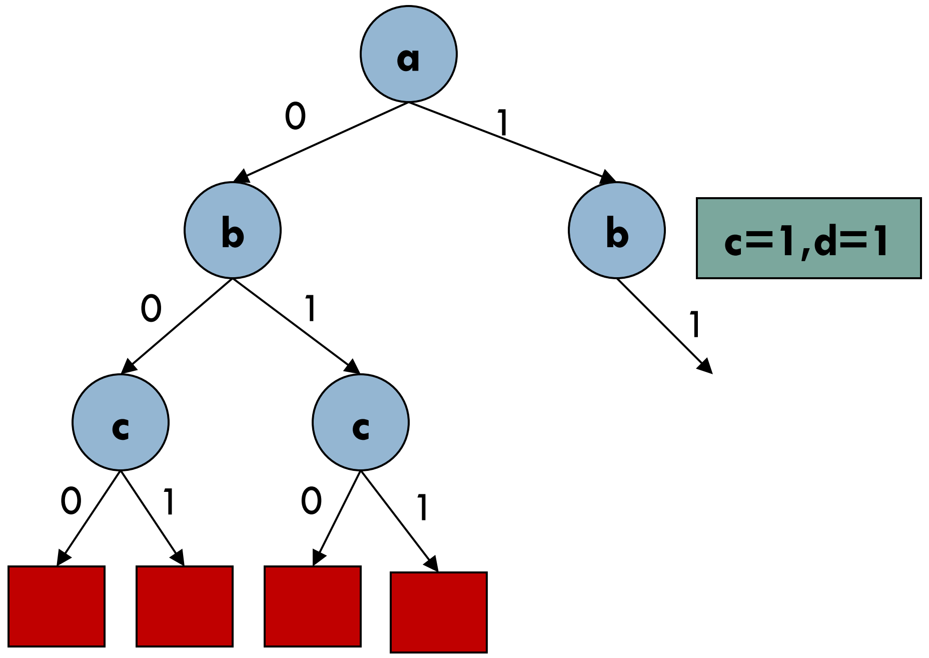


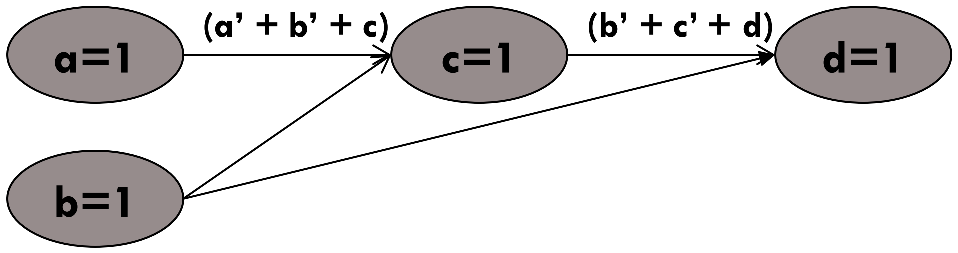


Implication Graph

Basic DLL Search

(a' + b + c)
(a + c + d)
(a + c + d')
(a + c' + d)
(a + c' + d')
→ (b' + c' + d)
(a' + b + c')
(a' + b' + c)



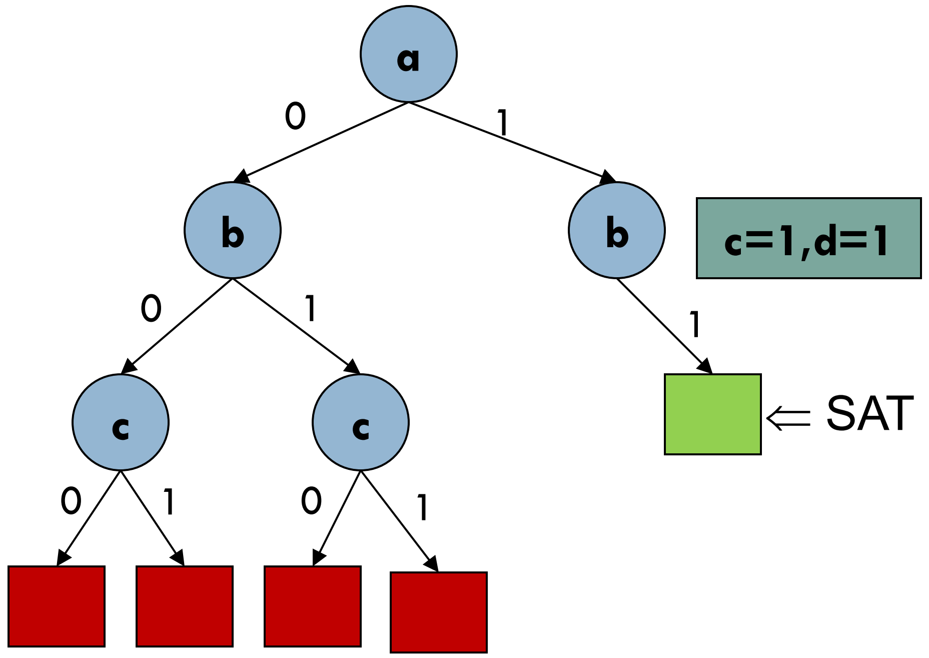


Implication Graph

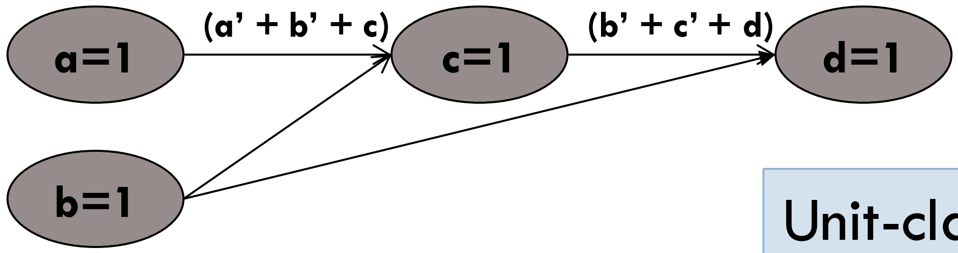
Basic DLL Search

→

(a' + b + c)
(a + c + d)
(a + c + d')
(a + c' + d)
(a + c' + d')
(b' + c' + d)
(a' + b + c')
(a' + b' + c)



Implication Graph



Unit-clause rule with backtrack search

Conflict Driven Learning and Non-chronological Backtracking

x1 + x4

x1 + x3’ + x8’

x1 + x8 + x12

x2 + x11

x7’ + x3’ + x9

x7’ + x8 + x9’

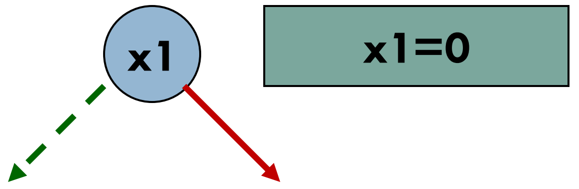
x7 + x8 + x10’

x7 + x10 + x12’

J. P. Marques-Silva and Karem A. Sakallah, “GRASP: A Search Algorithm for Propositional Satisfiability”, *IEEE Trans. Computers*, C-48, 5:506-521, 1999.

Conflict Driven Learning and Non-chronological Backtracking

x1 + x4
x1 + x3' + x8'
x1 + x8 + x12
x2 + x11
x7' + x3' + x9
x7' + x8 + x9'
x7 + x8 + x10'
x7 + x10 + x12'



Red text means evaluated to 0, and green means evaluated to 1

For the graph on the left:
Blue circles means free variable, and brown circles mean inferred variable.
Edge describes the inferred relationship.

Conflict Driven Learning and Non-chronological Backtracking

x1 + x4

x1 + x3' + x8'

x1 + x8 + x12

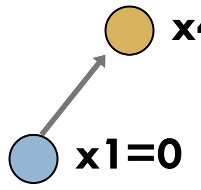
x2 + x11

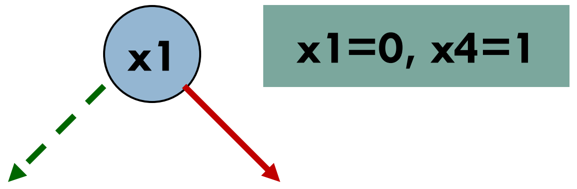
x7' + x3' + x9

x7' + x8 + x9'

x7 + x8 + x10'

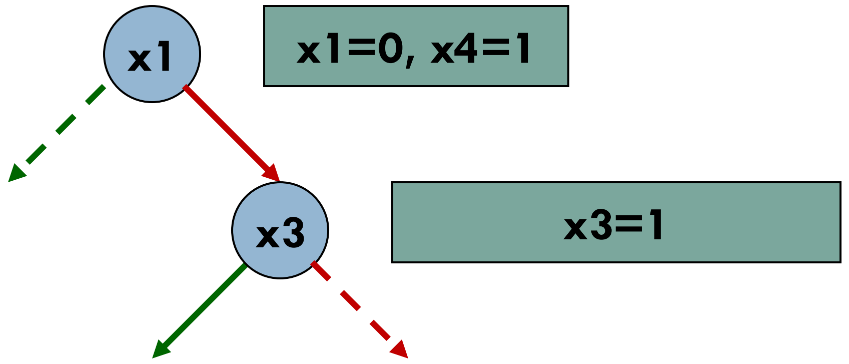
x7 + x10 + x12'

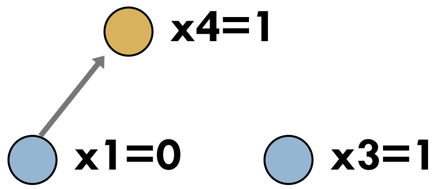




Conflict Driven Learning and Non-chronological Backtracking

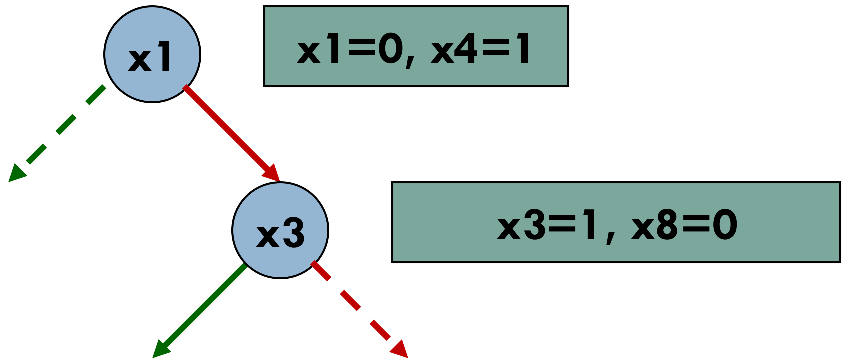
x1 + x4
x1 + x3’ + x8’
x1 + x8 + x12
x2 + x11
x7’ + x3’ + x9
x7’ + x8 + x9’
x7 + x8 + x10’
x7 + x10 + x12’

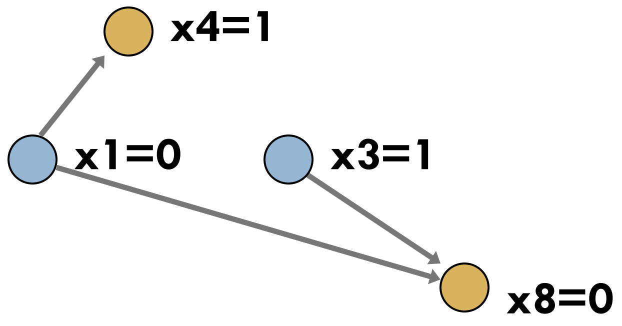




Conflict Driven Learning and Non-chronological Backtracking

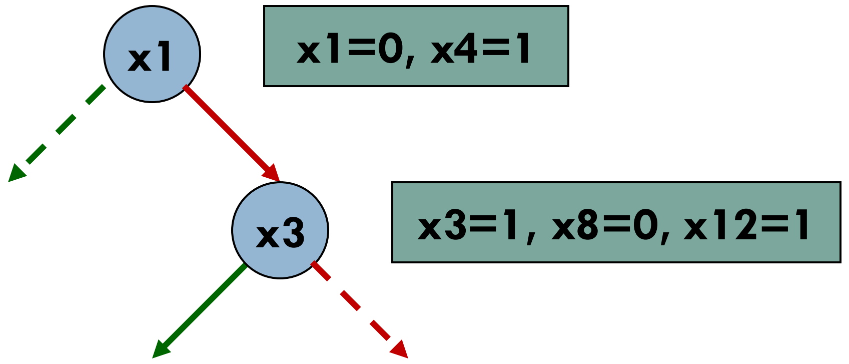
x1 + x4
x1 + x3’ + x8’
x1 + x8 + x12
x2 + x11
x7’ + x3’ + x9
x7’ + x8 + x9’
x7 + x8 + x10’
x7 + x10 + x12’

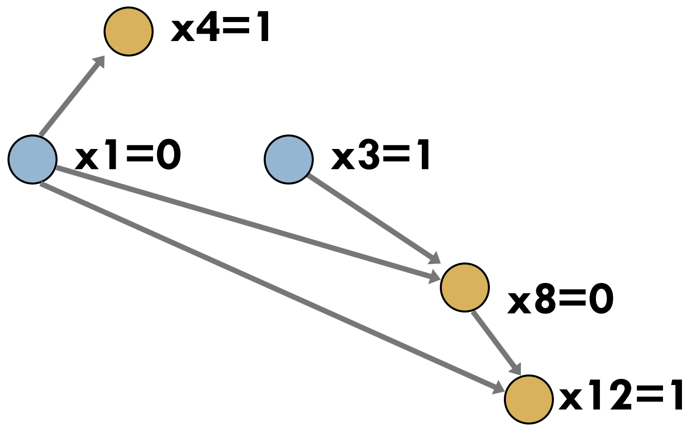




Conflict Driven Learning and Non-chronological Backtracking

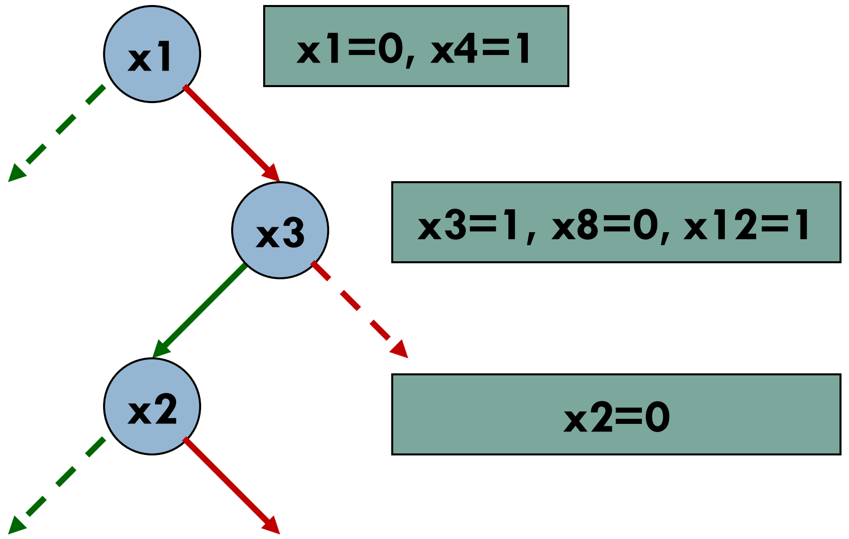
x1 + x4
x1 + x3' + x8'
x1 + x8 + x12
x2 + x11
x7' + x3' + x9
x7' + x8 + x9'
x7 + x8 + x10'
x7 + x10 + x12'

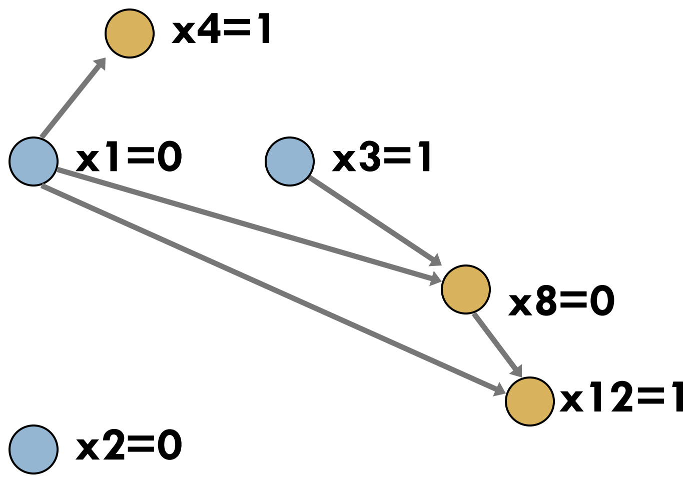




Conflict Driven Learning and Non-chronological Backtracking

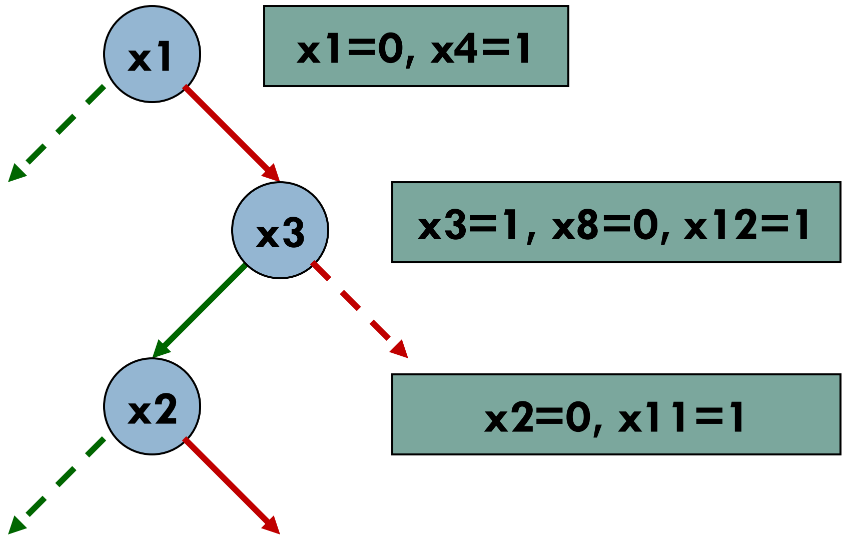
x1 + x4
x1 + x3’ + x8’
x1 + x8 + x12
x2 + x11
x7’ + x3’ + x9
x7’ + x8 + x9’
x7 + x8 + x10’
x7 + x10 + x12’

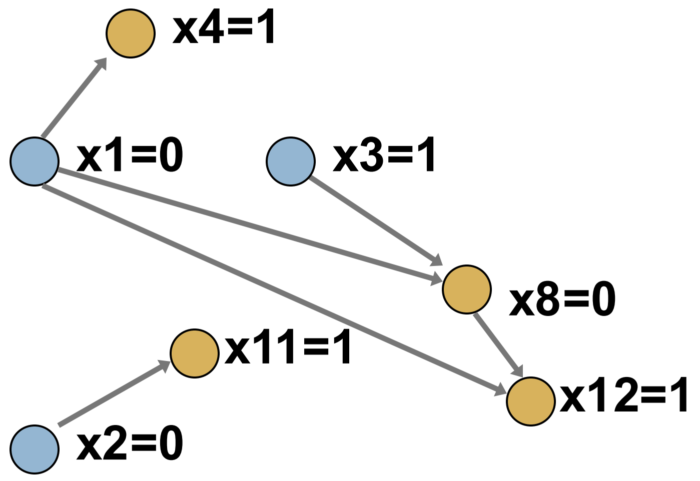




Conflict Driven Learning and Non-chronological Backtracking

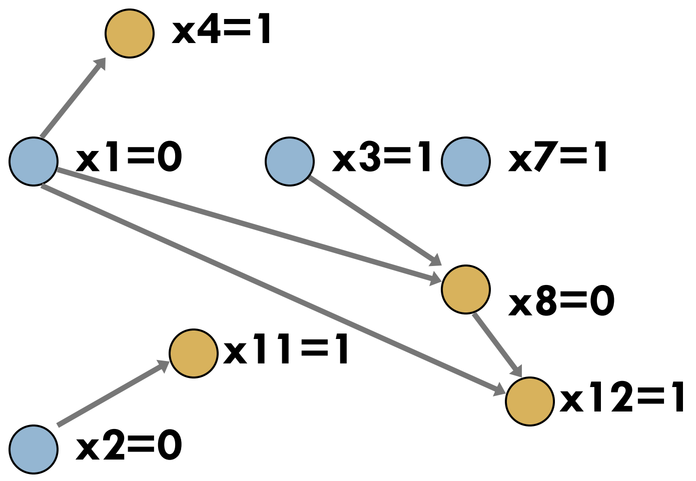
x1 + x4
x1 + x3' + x8'
x1 + x8 + x12
x2 + x11
x7' + x3' + x9
x7' + x8 + x9'
x7 + x8 + x10'
x7 + x10 + x12'

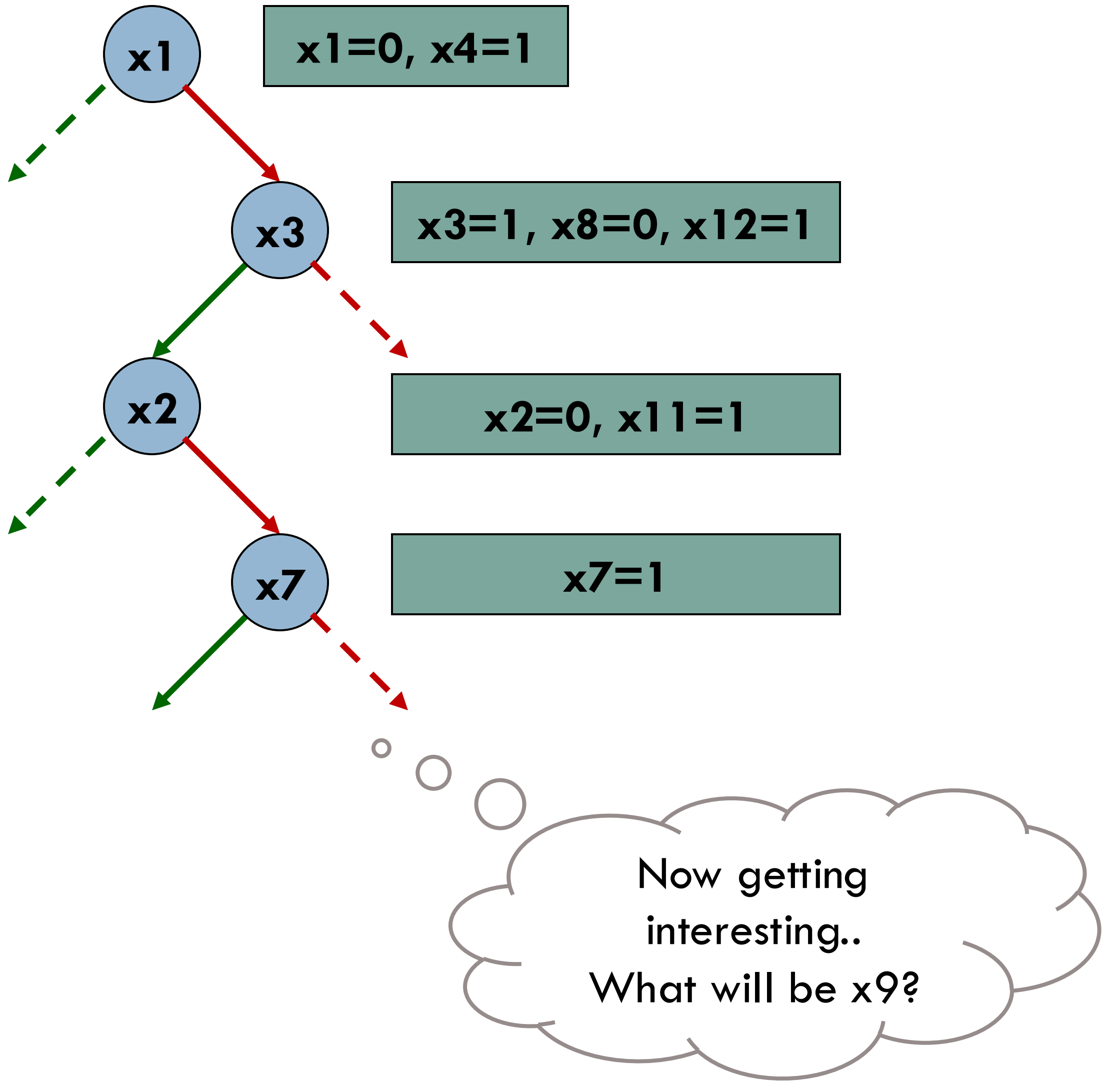




Conflict Driven Learning and Non-chronological Backtracking

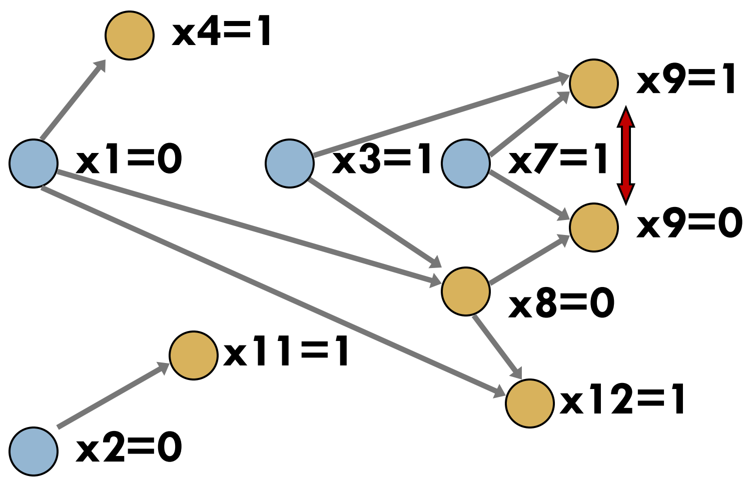
x1 + x4
x1 + x3' + x8'
x1 + x8 + x12
x2 + x11
x7' + x3' + x9
x7' + x8 + x9'
x7 + x8 + x10'
x7 + x10 + x12'

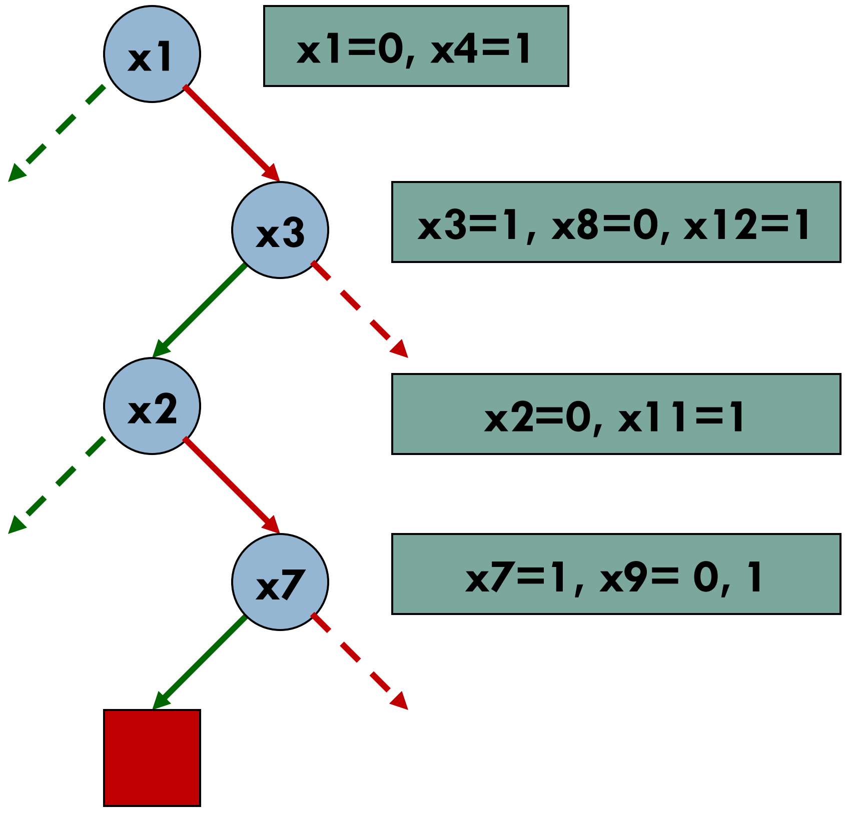




Conflict Driven Learning and Non-chronological Backtracking

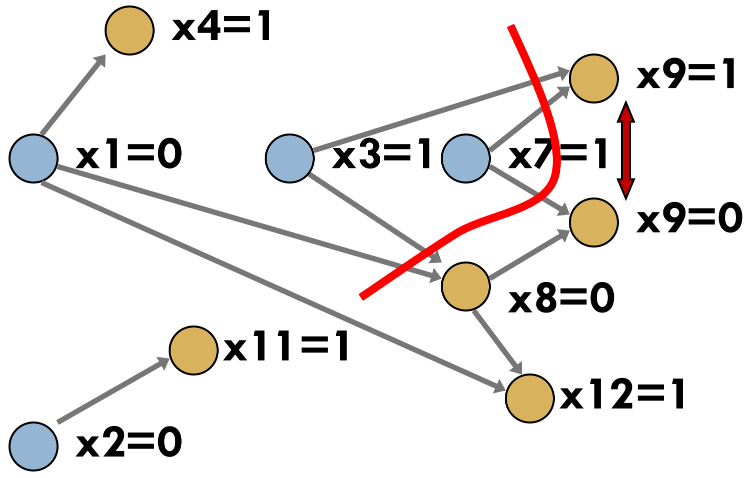
x1 + x4
x1 + x3’ + x8’
x1 + x8 + x12
x2 + x11
x7’ + x3’ + x9
x7’ + x8 + x9’
x7 + x8 + x10’
x7 + x10 + x12’

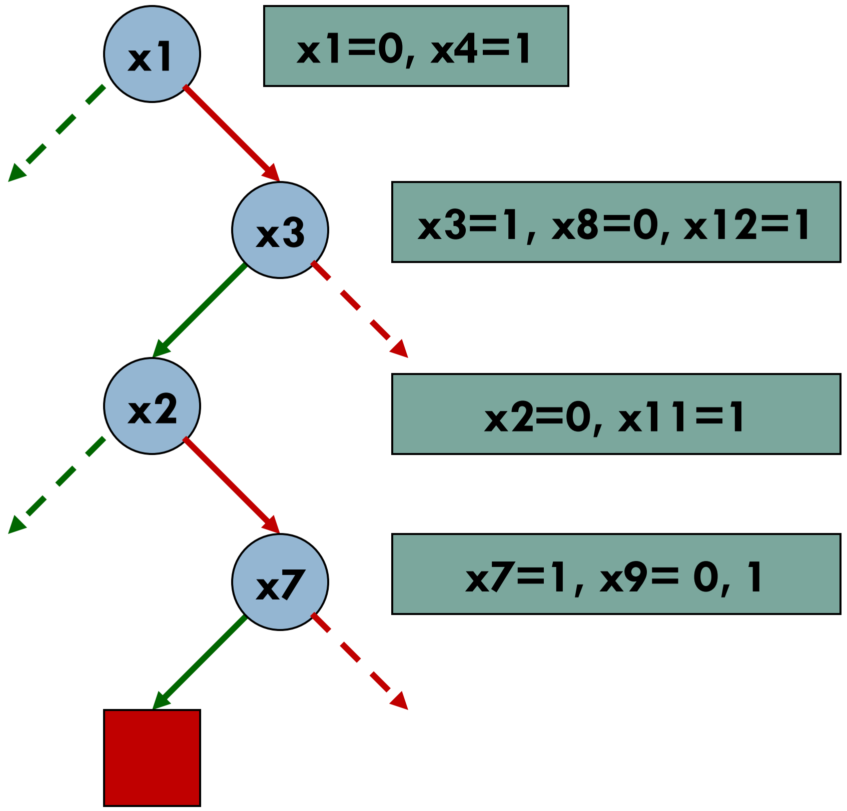




Conflict Driven Learning and Non-chronological Backtracking

x1 + x4
x1 + x3' + x8'
x1 + x8 + x12
x2 + x11
x7' + x3' + x9
x7' + x8 + x9'
x7 + x8 + x10'
x7 + x10 + x12'



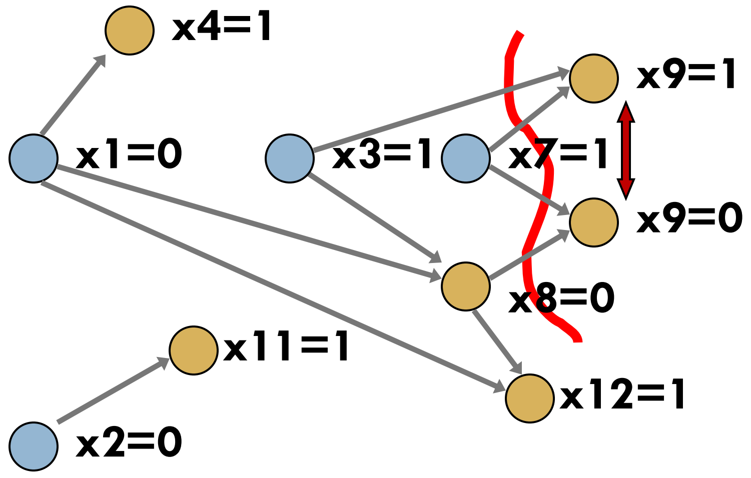


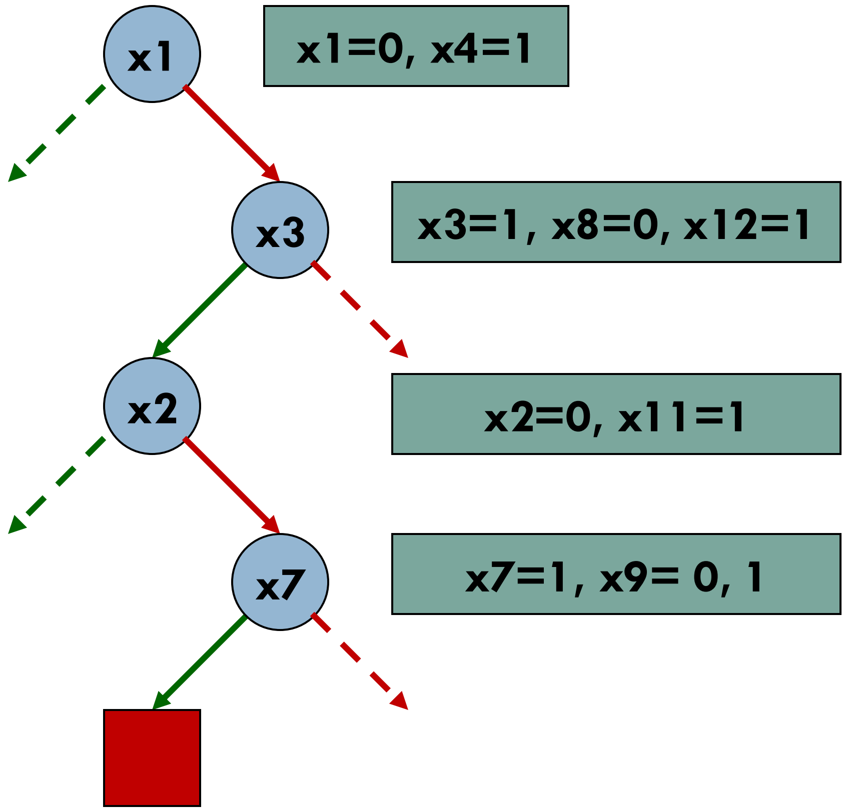
x3=1∧x7=1∧x1=0 → conflict

Add conflict clause: x3'+x7'+x1

Conflict Driven Learning and Non-chronological Backtracking

x1 + x4
x1 + x3’ + x8’
x1 + x8 + x12
x2 + x11
x7’ + x3’ + x9
x7’ + x8 + x9’
x7 + x8 + x10’
x7 + x10 + x12’





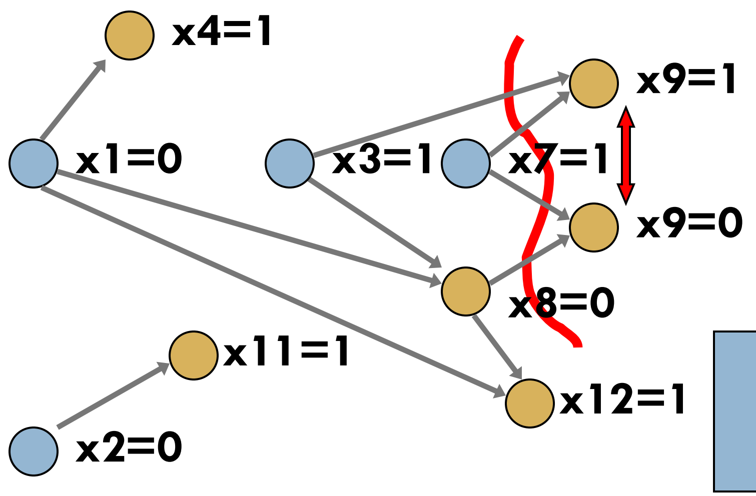
$x3=1 \wedge x7=1 \wedge x8=0 \rightarrow$ conflict

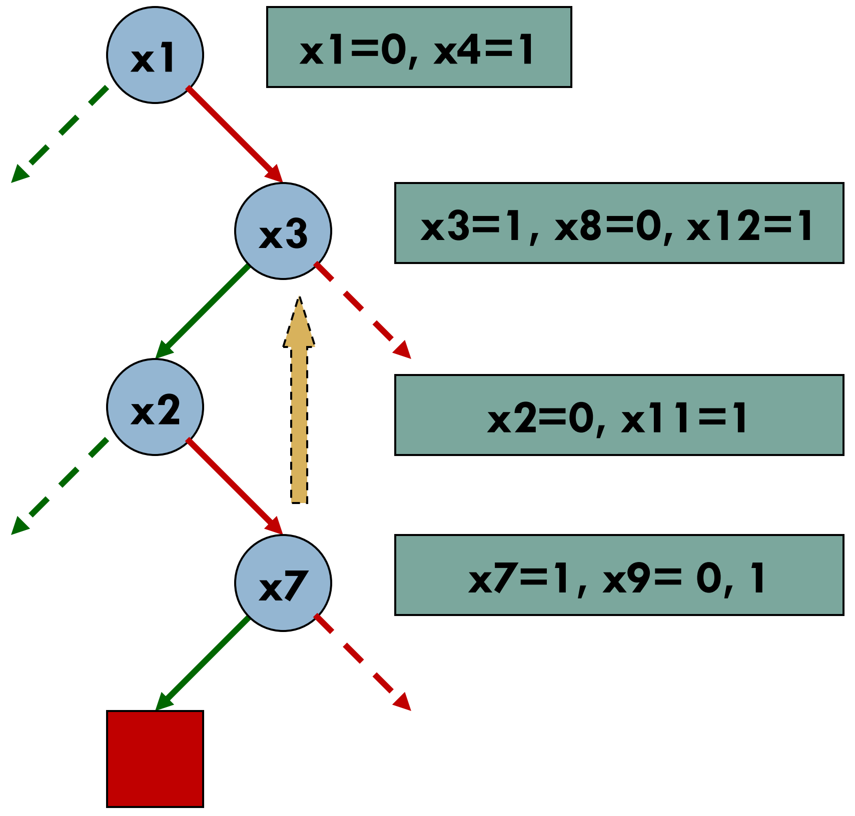
Add conflict clause: x3’+x7’+x8

Conflict Driven Learning and Non-chronological Backtracking

x1 + x4
x1 + x3' + x8'
x1 + x8 + x12
x2 + x11
x7' + x3' + x9
x7' + x8 + x9'
x7 + x8 + x10'
x7 + x10 + x12'

x3'+x7'+x8

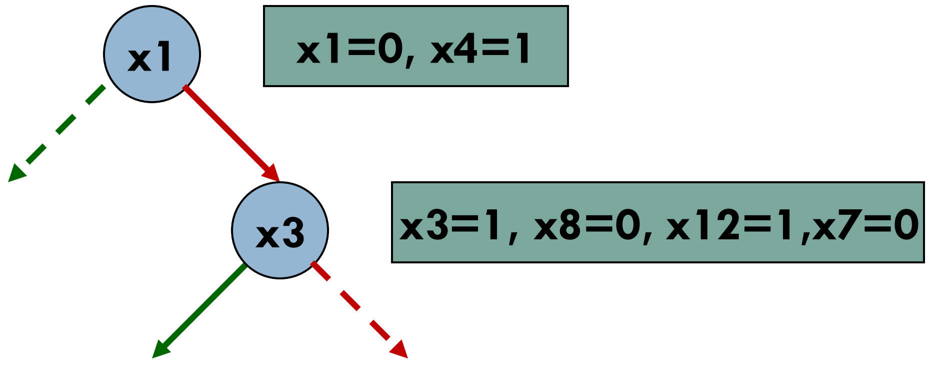


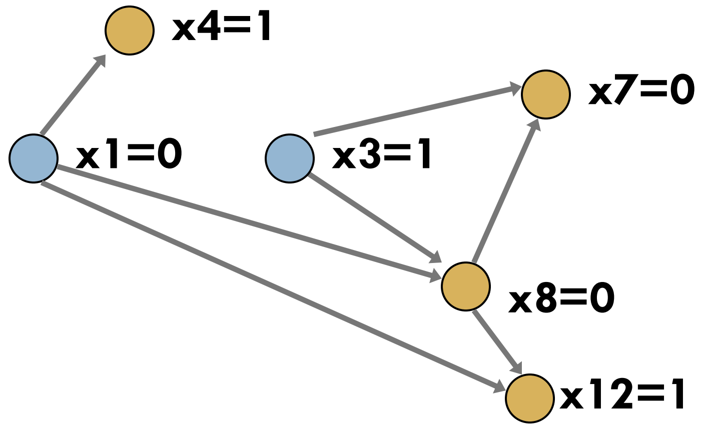


Backtrack to the decision level of x3=1

Conflict Driven Learning and Non-chronological Backtracking

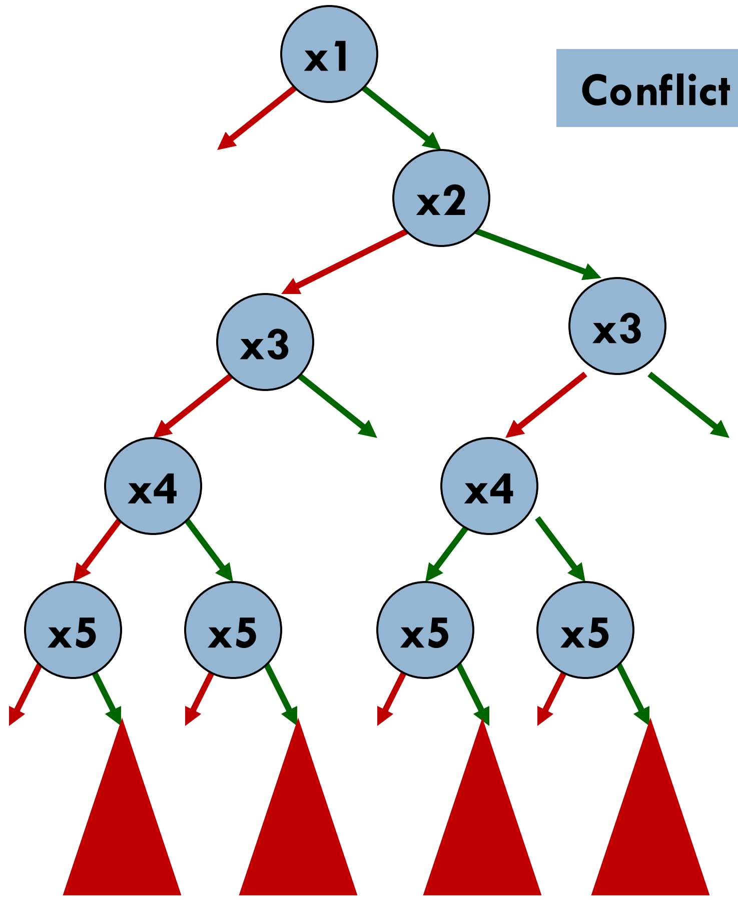
x1 + x4
x1 + x3’ + x8’
x1 + x8 + x12
x2 + x11
x7’ + x3’ + x9
x7’ + x8 + x9’
x7 + x8 + x10’
x7 + x10 + x12’
x3’ + x7’ + x8 ←new clause





Backtrack to the decision level of x3=1
Assign x7 = 0

What's the big deal?



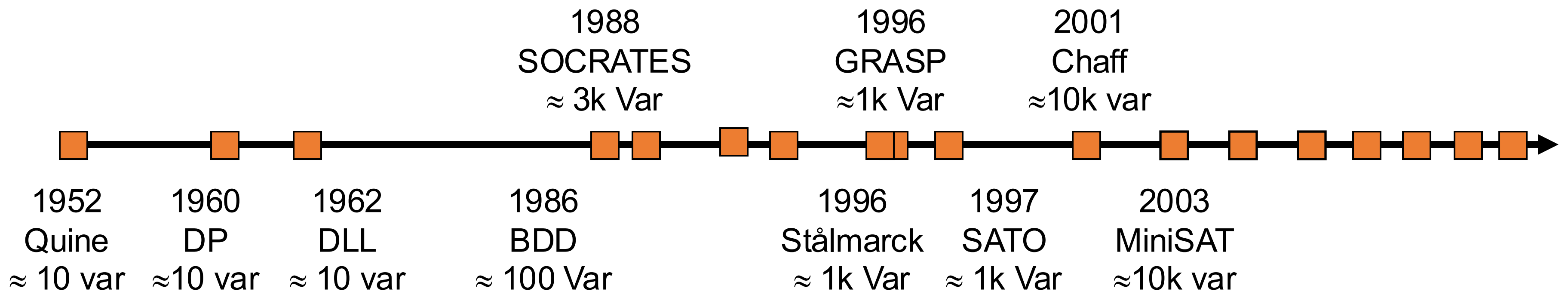
Conflict clause: x1'+x3+x5'

Significantly prune the search space – learned clause is useful forever!

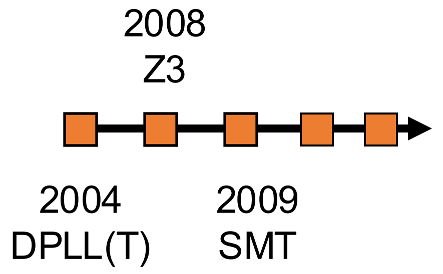
Useful in generating future conflict clauses.

Big Advancements in the Past Decade

(1) SAT: is a Boolean formula f satisfiable?

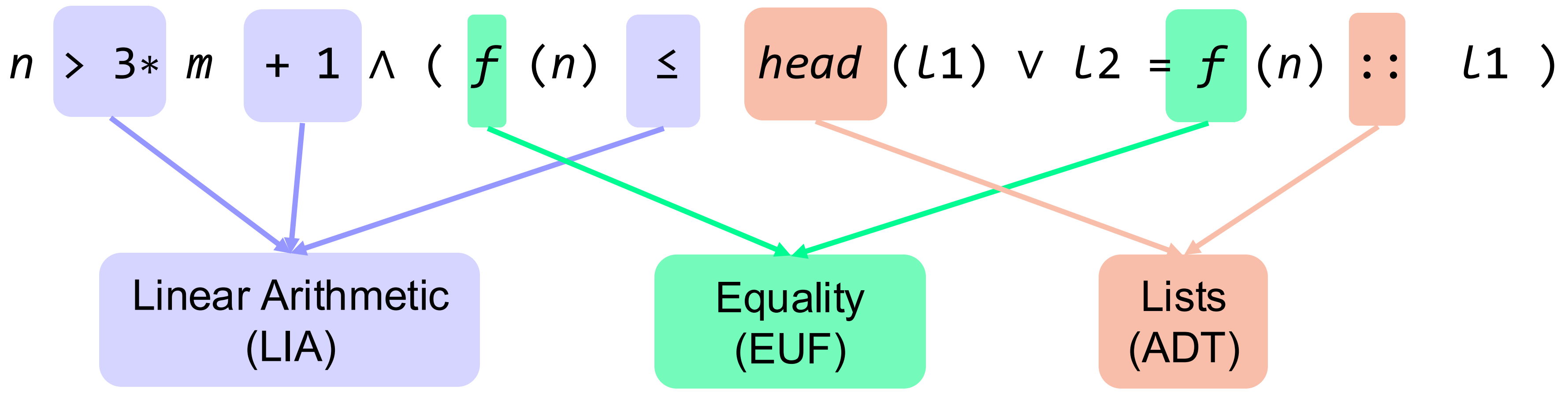


(2) SMT (Satisfiability Modulo Theory): is a first-order logic formula theory-satisfiable?



The Basic SMT Problem

- Determining the satisfiability of a logical formula with regards to some combination of background theories



$n > 3 * m + 1 \wedge (f(n) \leq head(l1) \vee l2 = f(n) :: l1)$
Linear Arithmetic (LIA)
Equality (EUF)
Lists (ADT)

Background Theories

Uninterpreted Funs	$x = y \Rightarrow f(x) = f(y)$
Integer/Real Arithmetic	$2x+y = 0 \wedge 2x-y = 4 \Rightarrow x = 1$
Floating Point Arithmetic	$x+1 \neq NaN \wedge x < \infty \Rightarrow x+1 > x$
Bit-vectors	$4 \cdot (x \gg 2) = x \mathbin{\&} {\sim}3$
Strings and RegExs	$x = y \cdot z \wedge z \in ab* \Rightarrow \lvert x\rvert > \lvert y\rvert$
Arrays	$i = j \Rightarrow \mathrm{store}(a, i, x)\,[j] = x$
Algebraic Data Types	$x \neq Leaf \Rightarrow \exists l, r : Tree(\alpha).\ \exists a : \alpha.\ x = Node\,(l, a, r)$
Finite Sets	$e1 \in x \ \wedge\ e2 \in x \setminus e1 \Rightarrow$ $\exists y, z : Set(\alpha).\ \lvert y\rvert = \lvert z\rvert \wedge x = y \cup z \wedge y \neq \emptyset$
Finite Relations	$(x, y) \in r \wedge (y, z) \in r \Rightarrow (x,z) \in r$
...	

CDCL(T): Key Idea

- SAT solver handles Boolean structure of the formula
  - Treat each *atomic* formula as a propositional variable
  - Resulting formula is called a *Boolean abstraction (B)*
- Example

$$F: \underbrace{(x=z)}_{b1} \wedge (\underbrace{(y=z}_{b2} \wedge \underbrace{x = z+1)}_{b3} \vee \neg \underbrace{(x=z)}_{b1})$$

$B(F)$: $b1 \wedge ((b2 \wedge b3) \vee \neg b1)$

Boolean abstraction (B) is defined inductively over formulas

B is a bijective function, B^{-1} also exists

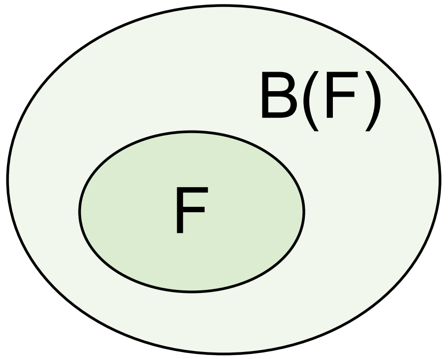
$B^{-1}(b1 \wedge b2 \wedge b3)$: $(x=z) \wedge (y=z) \wedge (x=z+1)$

$B^{-1}(b1 \vee b2')$: $(x=z) \vee \neg(y=z)$

CDCL(T): Key Idea

$$F: \underbrace{(x=z)}_{b1} \land ((\underbrace{y=z}_{b2} \land \underbrace{x = z+1}_{b3}) \lor \neg \underbrace{(x=z)}_{b1})$$

$B(F): b1 \land ((b2 \land b3) \lor \neg b1)$



B(F) is an *over-approximation* of F

- Use SAT solver to decide satisfiability of B(F)
  - If B(F) is Unsat, then F is Unsat
  - If B(F) has a satisfying assignment A, *F may still be Unsat*
- Example: b1, b2, b3 are not independent propositions!

  SAT solver finds a satisfying assignment A: $b1 \wedge b2 \wedge b3$

  But, $B^{-1}(A)$ is unsatisfiable modulo theory

  $(x=z) \wedge (y=z) \wedge (x=z+1)$ is not satisfiable

CDCL(T): Simple Version

1. Generate a Boolean abstraction B(F)
2. Use SAT solver to decide satisfiability of B(F)
   - If B(F) is Unsat, then F is Unsat
   - Otherwise, find a satisfying assignment A
3. Use theory solver to check if $B^{-1}(A)$ is satisfiable modulo T
   - If $B^{-1}(A)$ is satisfiable modulo theory T, then F is satisfiable
   - Otherwise, $B^{-1}(A)$ is unsatisfiable modulo T
     Add $\neg A$ to B(F), and backtrack in SAT

Repeat (2, 3) until there are no more satisfying assignments

Interacting with SAT/SMT Solvers

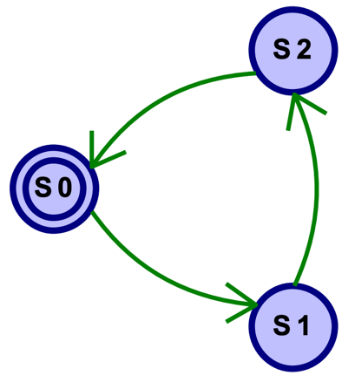
Interact with a solver

- A counterexample is generated.
  You can use it to fix your program. 😉
- A proof is generated. Your program is bug-free! 😎
- (most of the time) …
  Clueless. Basically the solver does not generate a result since the search cannot complete. 😕

Need to consult other approaches, which require formal-method expertise:
Induction proof, find invariants, theorem proving, etc.
If interested, check out 6.512 https://frap.csail.mit.edu/main

Verifying Hardware Designs

- Hardware RTL code works as if a big loop



A divide-3 FSM

```
module divideby3FSM (input clk, input reset, output q);
    reg [1:0] state, nextstate;

    always @ (posedge clk) // state register
        if (reset) state <= 2'b00;
        else state <= nextstate;

    always @ (*) // next state logic
        case (state)
        2'b00: nextstate = 2’b01;
        2’b01: nextstate = 2’b10;
        2’b10: nextstate = 2’b00;
        default: nextstate = 2’b00;
    endcase

    assign q = (state == 2'b00); // output logic
endmodule
```

Toolchains to Verify Hardware

```
module divideby3FSM (input clk, input reset, output q);
    reg [1:0] state, nextstate;

    always @ (posedge clk) // state register
        if (reset) state <= 2'b00;
        else state <= nextstate;

    always @ (*) // next state logic
        case (state)
        2'b00: nextstate = 2’b01;
        2’b01: nextstate = 2’b10;
        2’b10: nextstate = 2’b00;
        default: nextstate = 2’b00;
    endcase

    assign q = (state == 2'b00); // output logic
endmodule
```

Verilog code

Compilation Toolchain (in Recitation)

A representation that supports symbolic execution (e.g., Rosette)

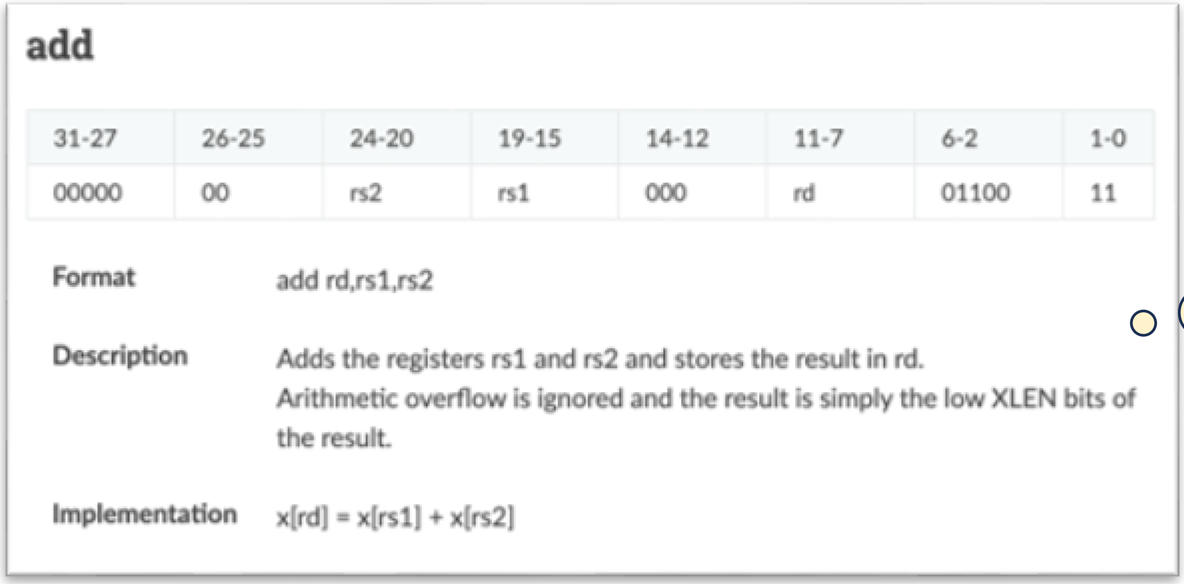
Verify hardware as if verifying software

Directly use hardware verification tools

cādence®

SYNOPSYS®

An Example: Verify ISA Correctness


add

31-27	26-25	24-20	19-15	14-12	11-7	6-2	1-0
00000	00	rs2	rs1	000	rd	01100	11

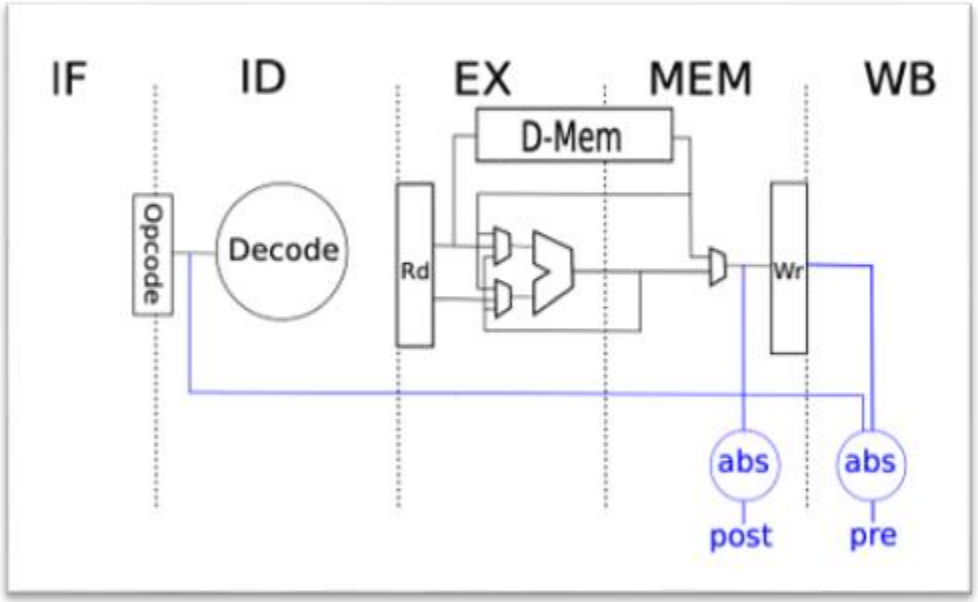
Format	add rd,rs1,rs2
Description	Adds the registers rs1 and rs2 and stores the result in rd. Arithmetic overflow is ignored and the result is simply the low XLEN bits of the result.
Implementation	x[rd] = x[rs1] + x[rs2]

RISC-V Instruction Set Specification

If interested, check x86 ISA specification for "add"

- Question 1: What assertion should we put into our RTL code?
- Question 2: If I have a 5-stage pipelined processor, when do I place the assertion?
- Question 3: If I want to catch some bypass bugs, how should I initialize the state of the processor?

A Tentative Plan



The instruction encoding below follows ARM ISA, different from RISCV from the last slide.

```
assign ADD_retiring = (pre.opcode & 16'b1111_1110_0000_0000) == 16'b0001_1000_0000_0000;
assign ADD_result = pre.R[pre.opcode[8:6]] + pre.R[pre.opcode[5:3]];
assign ADD_Rd = pre.opcode[2:0];

assert property (@(posedge clk) disable iff (reset_n)
ADD_retiring |-> (ADD_result == post.R[ADD_Rd]));
```

End-to-End Verification of ARM® Processors with ISA-Formal; Reid et al.; CAV'16

A Problem: Register Renaming

- A performance optimization to resolve WAW (write-after-write) data dependency

```
addi r1, r1, 4
ld   r2, 0(r1)
addi r1, r1, 4
ld   r3, 0(r1)
addi r1, r1, 4
ld   r4, 0(r1)
```

$\rightarrow$

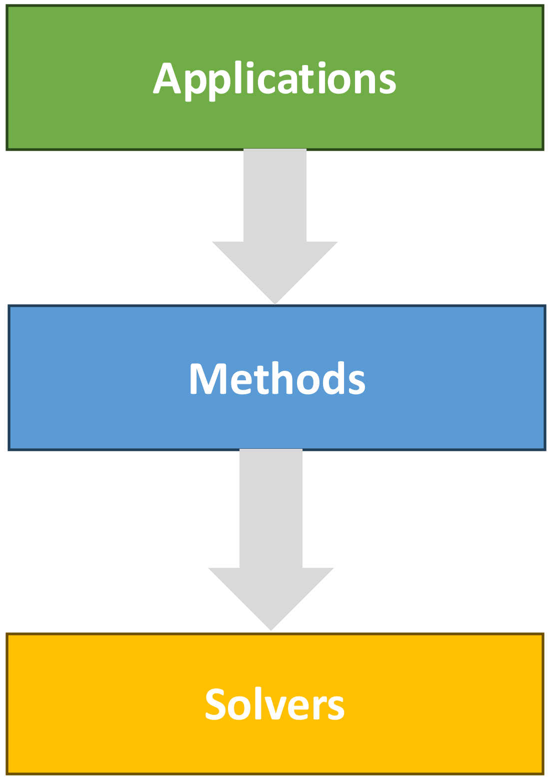
```
addi r1, r1, 4
ld   r2, 0(r1)
addi r11, r1, 4
ld   r3, 0(r11)
addi r12, r11, 4
ld   r4, 0(r12)
```

- Modern out-of-order processors do register renaming on-the-fly
  - Many different implementations, check out 6.823/6.5900
- Problem: How do we verify such processors?

Shadow logic to implement correct renaming logic

Summary

- Formal Verification: rigor, exhaustiveness, automation



For hardware verification: often needs domain expertise to translate specification to assertions

See symbolic execution as an example
There exist many other approaches: model checking, theorem proving, etc.

See some algorithms for SAT and SMT
Understand how complex and unpredictable the solver’s performance can be

Next: Recitations

Hardware Formal Verification Toolchains