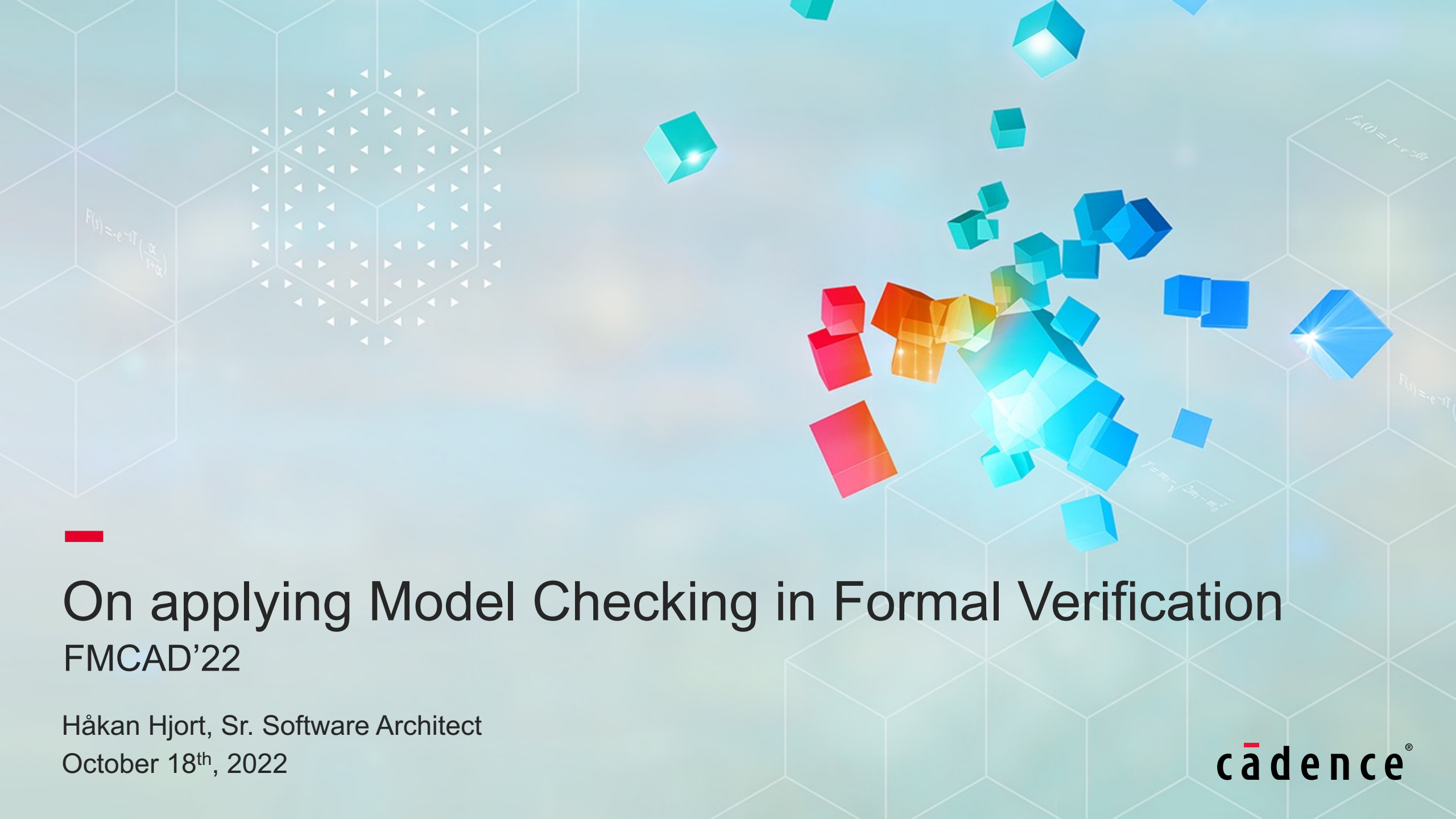


On applying Model Checking in Formal Verification

FMCAD'22

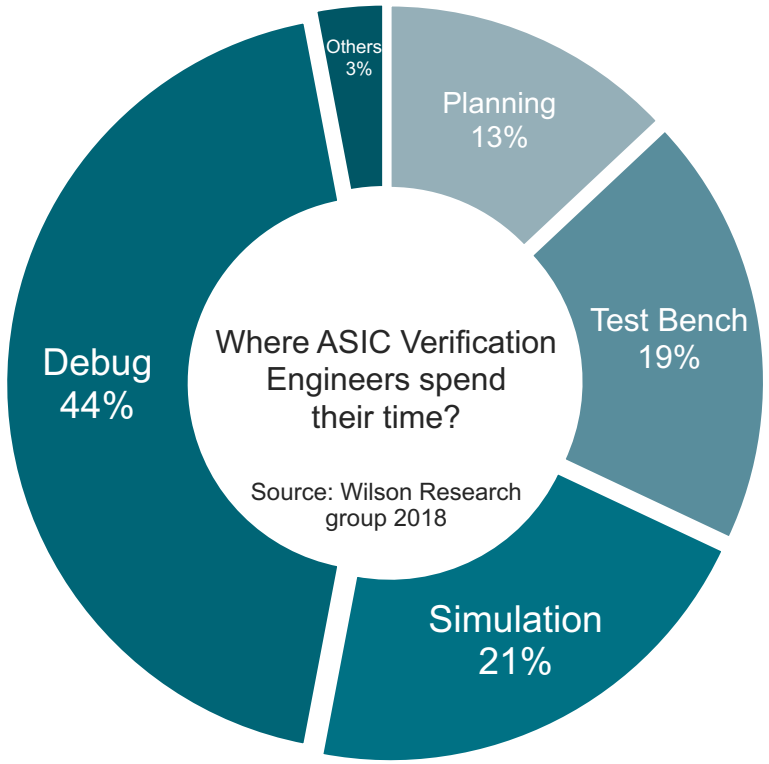
Håkan Hjort, Sr. Software Architect

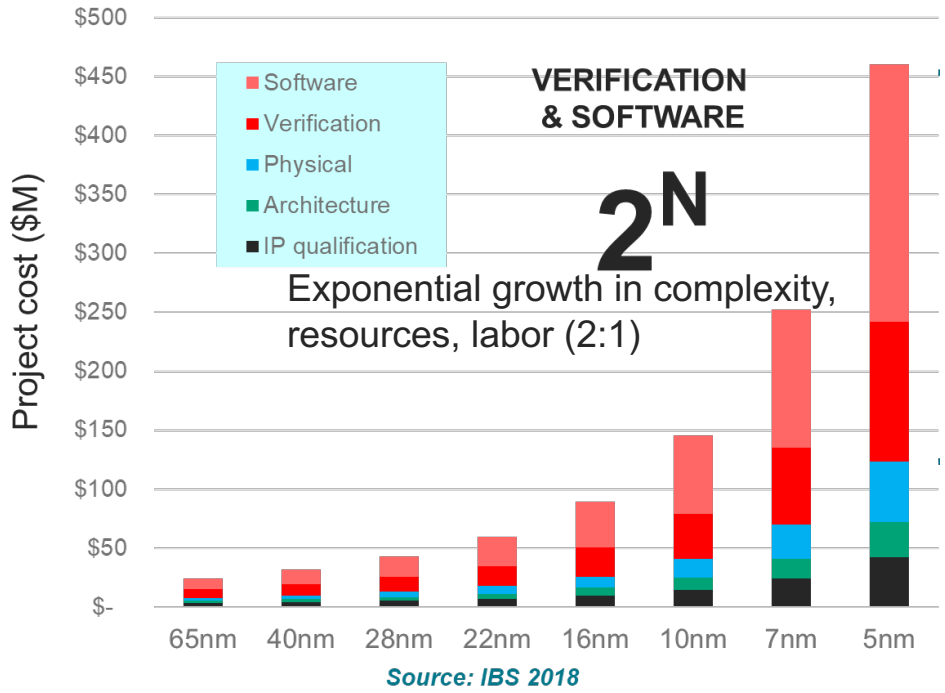
October 18th, 2022

cadence®

Intro and background

- Verification challenges, and so cost, keep increasing

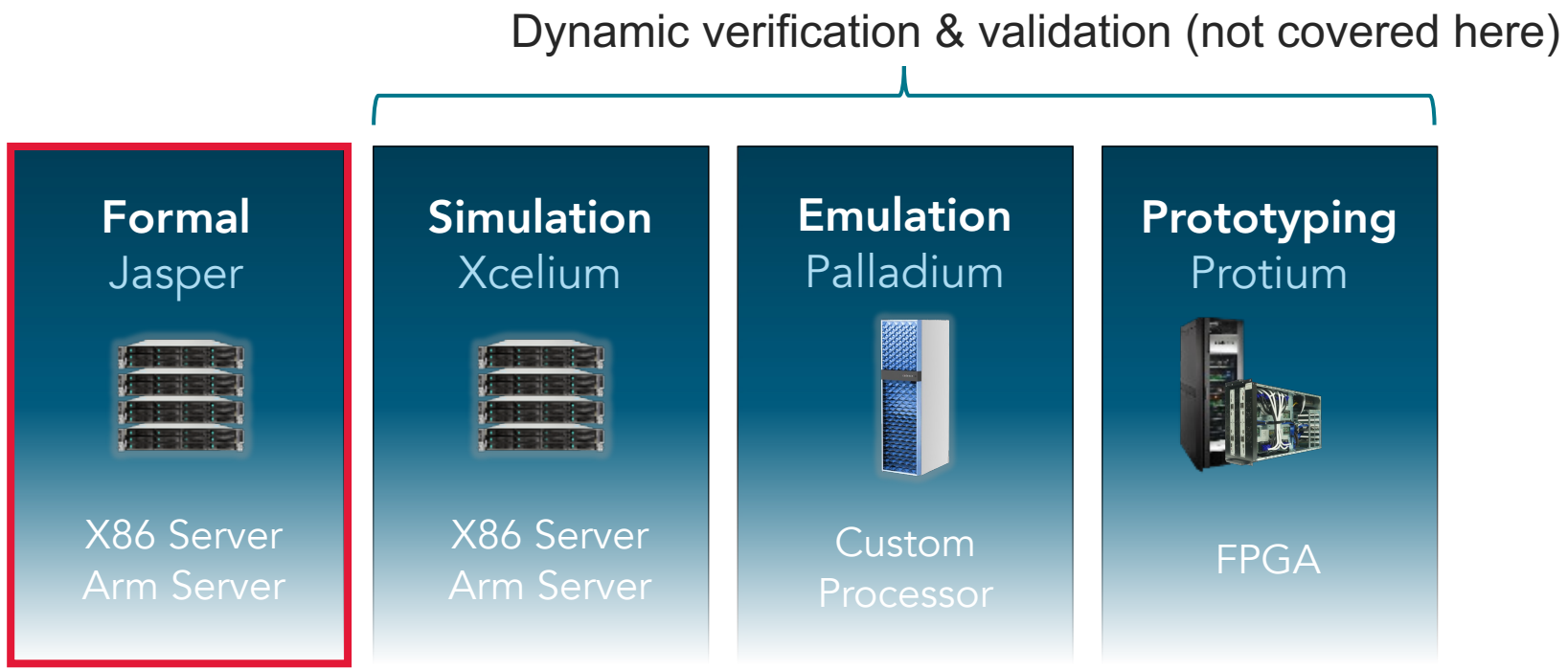




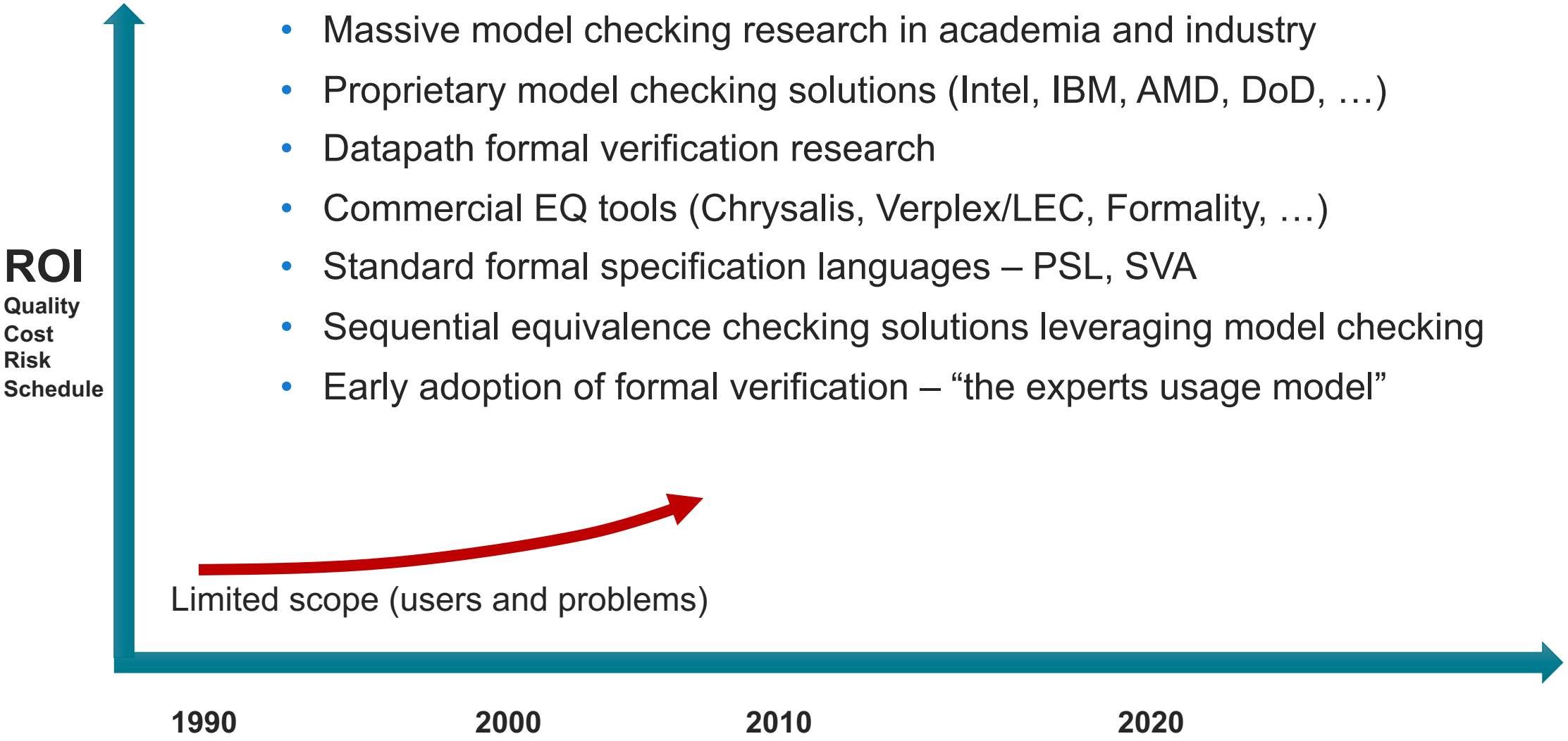
- To cope with system complexity, need to boost verification
  - No compromise in quality

Intro and background

- **Verification**: "Have we made what we were trying to make?"
  - i.e., does the product conform to the specifications?
- **Validation**: "Are we trying to make the right thing?"
  - i.e., is the product specified to the user's actual needs?

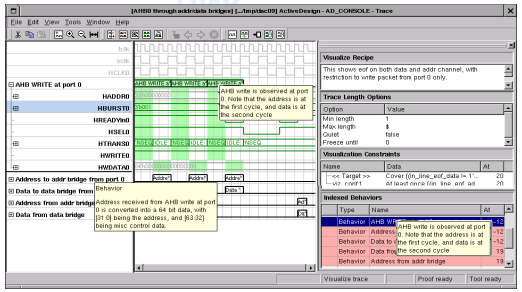


A Decade of Early Industrial Formal Verification (1995-2005)



- Massive model checking research in academia and industry
- Proprietary model checking solutions (Intel, IBM, AMD, DoD, …)
- Datapath formal verification research
- Commercial EQ tools (Chrysalis, Verplex/LEC, Formality, …)
- Standard formal specification languages – PSL, SVA
- Sequential equivalence checking solutions leveraging model checking
- Early adoption of formal verification – “the experts usage model”

Disruptive Formal Verification – Formal ‘Apps’



ROI
Quality
Cost
Risk
Schedule

Arch | Design | Verify | Integrate | Post-Si

Arch | Design | Verify | Integrate | Post-Si

Savings

Limited scope

100s of active users

1990 **2000** **2010** **2020**

ARM

2.5X better ROI than simulation

Bugs found earlier: 82% code churn reduction

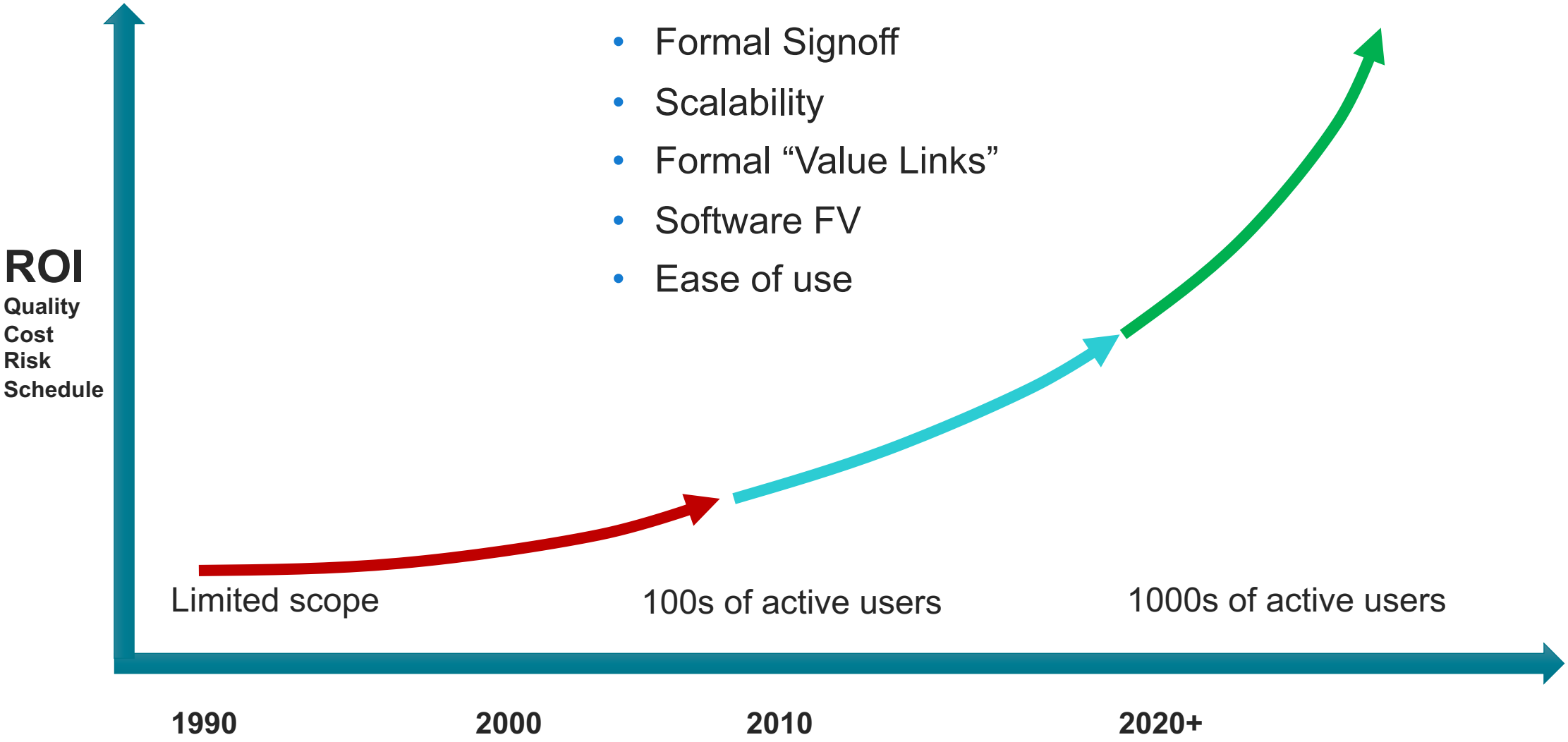
NVIDIA

84% of bugs found automatically

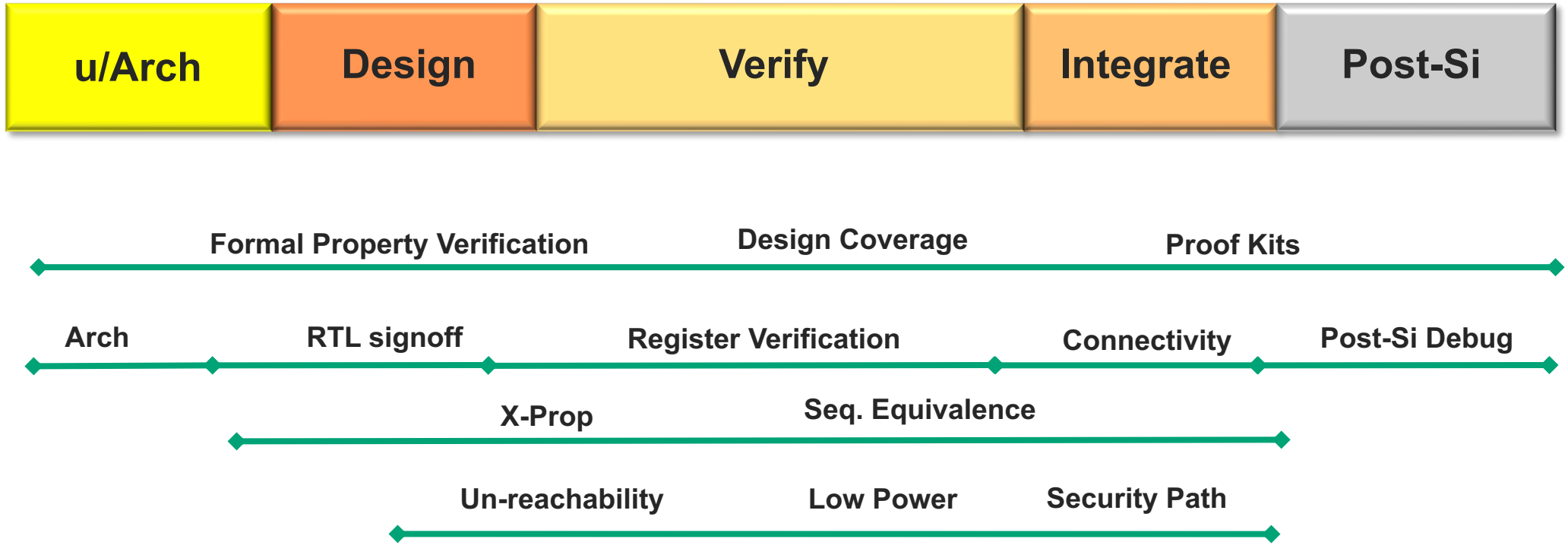
QUALCOMM

Late-change verification in a day vs. a week

Formal Verification becomes Mainstream



Formal is for Many People!



Intro and background

- Use of formal methods and in particular model checking in the EDA industry is wide spread and now considered an essential part of the verification flow.

- While there are many papers, and books, about SAT, SMT and Symbolic Model Checking, less is written about the specifics of how these methods are applied.
- Focus here is on RTL (register-transfer level) hardware designs.

- There is generally no formal semantics defined for the hardware design languages, nor for the intermediate representations in common use.
- As unsatisfactory as that may be, industry conventions and behaviour exhibited by real hardware have instead had to be used as guides.

Intro and background

- The increasing tool capacity, and user familiarity has lead to wider usage, both earlier and later in the design cycle, as well as on ever larger design parts.
  - Input might not be in Verilog/VHDL, could be in XLS, Murphy, SystemC/C++ etc.
  - Or could be on already synthesized (gate level) netlists with implementation artifacts present
- Applying formal on larger parts of the design flow comes with new challenges.
  - Scan and Test logic used by post-silicon debug and by testing machines to accept / reject individual chips.
    - A scan chain will make it appear as if almost all of the registers are in the COI of all others…
  - PAD rings, I/O pad drivers, ESD structures for bond pads etc.
    - Generally non-digital parts to interface the 'outside world'
  - Verilog UDPs (user defined primitives), Cell libraries, transistor primitives, …
  - UPF / CPF, power specifications that modify the RTL
    - To conserve power parts are turned off (and on), functionally should still be preserved
- There will be all kinds of 'bugs' (including input & expectations)

Topics

- Synthesis for verification, netlists, gates and hierachy flattening
- Value domains, Tri-states, weak-driver resolution
- Constraints and environment modeling
- Initialization / Reset
- Clocks and creating a discrete time model
- Feedback and loops

Synthesis for verification

Netlists, gates and hierachy flattening

Synthesis for verification - Input

- A common intermediate for digital circuit implementation is what's referred to as a register-transfer level (RTL) description.
- RTL is a design abstraction which models a synchronous digital circuit
  - Logical operations performed on signals and their flow between hardware registers.
  - It's low level enough to describe exactly behaviour per cycle while still allowing designers to understand and modify it.
- Hardware description languages (HDLs) are used to write RTL designs
  - Verilog, now SystemVerilog, IEEE Std 1800-2017 is the dominating language
  - There is also VHDL IEEE Std 1076-2019
- There are two industry standards for formal specifications
  - SystemVerilog Assertions (SVA) IEEE Std 1800-2017 and
  - Property Specification Language (PSL) IEEE Std 1850-2010, commonly used with VHDL.

Example – A ‘strange’ 2-bit counter and asserts in SystemVerilog

```
module counter (clk, rst, en,
cnt);

 input clk, rst, en;

 output reg [1:0] cnt;

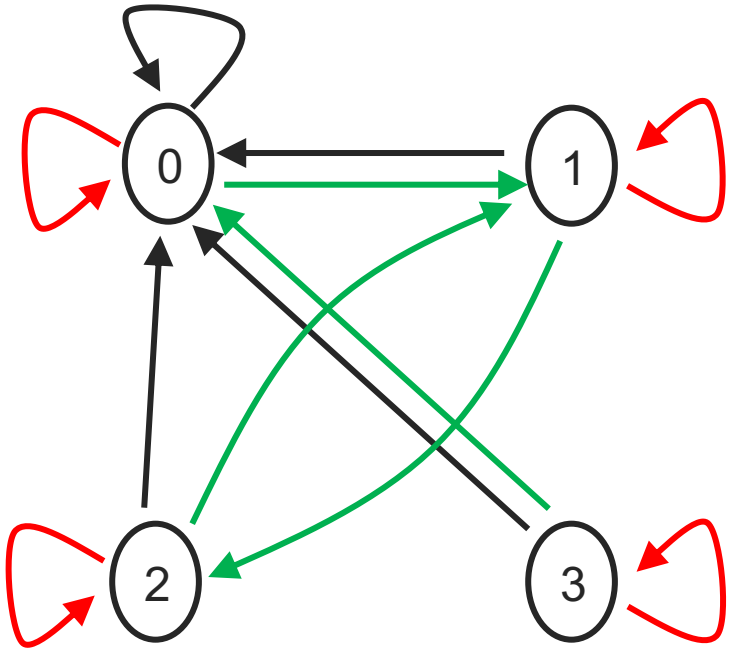
 always @(posedge clk)
  if (rst)
    cnt <= 2'd0;
  else if (en)
    if (cnt[1])
      cnt <= ~cnt;
    else
      cnt <= cnt + 2'd1;
```

```
  p1: assert property (
        @(posedge clk)
        disable iff (rst)
       en |=> cnt != 2'd0
  );

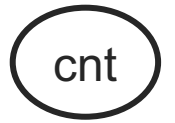
  p2: assert property (
        @(posedge clk)
       en |=> cnt <= 2'd3
  );

  p3: assert property (
        @(posedge clk)
        en |-> s_eventually
                 (cnt == 2'd0)
  );

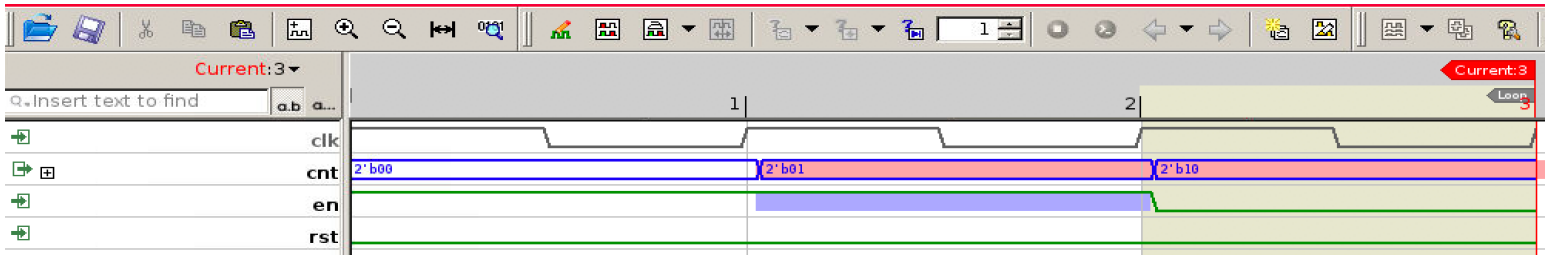
endmodule
```



states = {0,1,2,3}
init = ??
reachable = ??



rst
en&!rst
!en&!rst



Synthesis for verification

- Design compilation, also know as elaboration and *synthesis,* is used to create a gate netlist from the HDL.
  - When done for implementation this often leverages any semantic freedom in order to create a more efficient implementation.
  - In contrast for verification we prefer to preserve all possible behaviour of any valid implementation choice.
- For implementation, the target is gates/cells provided by the foundry.
  - Can change with technology node targeted (e.g TSMC 5nm)
- Any under-specified behaviour might be leveraged to reduce the logic
  - Verilog ‘x’ value is mostly considered to be a don’t-care value *for synthesis*.
  - F.ex. Indexing outside of an array bound is specified (for some types) as returning ‘x’
    - reg [3:0] a; a[5] should give 1’bx, but synthesis tool might decide to use only 2-bit to index a, resulting in a[1]
  - There can be explicit assignments of ‘x’, or overlapping conditions in switch statements
    - Or a promise that one case will match (but no guarantee).

Synthesis for verification

- When synthesising for verification we can chose the target gates
  - Reduce the complexity of synthesis
  - Be easy to consume for model checking
  - Ease the debug flow
- ‘Quick’ synthesis keeps gates close to source language constructs
  - Dataflow, width and symbol resolution is made explicit
- Common classes of gates
  - Logic: (N-ary) And, Or, Xor, etc and ‘reduction’ variants (N-bit input with 1-bit output)
  - Arithmetic: Adders, Shifts, Multipliers, Dividers, …
  - Control/Data flow: Mux (ITE), Selectors, Decoders, Array operations (concat/slice/index/...)
  - State elements: Flops, RAMs, Latches,
  - Other more specialized gates
    - Tri-state drivers, Transistor primitives, …

Hierachy flattening

- A gate Netlist is a hierarchical representation using Ports, Instances and Nets
  - Instances are components defined by another netlist or a ‘primitive’ gate
  - A Net represent a wire
  - Ports on instances are connected to other ports by Nets
- Removal of *hierarch* can largely be done replicating the logic (called flatten).
  - Easier to not have hierarchy but can lead to a large increase in size, also structure is lost.
- Most gate types represent combinatorial functions
  - These can be kept as is, or lowered to smaller subset of gate functions
    - For example AND-gates & NOT-gates if targeting And-Inverter graphs (AIGs).
  - Size might increase, and here too some of the original structure would be lost.
- The state holding gates, (Flip-)Flops (edge sensitive) and Latches (level sensitive) require some more care to model their (a)synchronous behaviour.
  - More on that later…

Specification - Properties

- Properties, in SVA and PSL, express assertions, covers and constraints*
  - These languages are mostly equivalent to LTL with a lot of syntactic ‘sugar’
    - Sequences looks a bit like regular expressions, makes it easy to express delays and repetitions.
  - PSL also specifies an optional branching extension that covers CTL properties.
  - Another extension is local variables, helpful when verifying data paths (transport) this takes things outside the LTL expressive power.
- The properties are compiled to an automat representation
  - Which is similarly translated to an circuit, and implemented as part of the netlist.
- All properties can be reduced down to a few canonical forms
  - Safety: the LTL formula ‘G !bad’ (a model for the negation ‘F bad’ is a CEX / Failure)
  - Liveness: the LTL formula ‘G F good’ (a model for ‘F G !good’ is a CEX / Failure)
  - (Deadlock: the CTL formula ‘AG EF good’, not expressible in LTL, sometimes Fair-CTL)

Specifications

- Properties can also be generated by the tool.
  - This is attractive for new / non-formal savvy users, and a compliment to user written properties.
- Generated from the structure of the code
  - Works like compiler Lint checks, or the now popular Clang/GCC sanitizers.
  - There can be a large number of these properties (for large code bases).
  - The advantage is that model checking provides a definite answer rather than relying on heuristic (with false positives) or requiring a test framework to trigger the ’bad’ scenario.
- Pre-packaged
  - In hardware design it’s common with standardized protocols for IP-blocks.
  - For these there often exist reusable collections of ‘checks’ and/or constraints
- There are many design rules and patterns that lend themself to automation
  - Reset & Clock networks, Connectivity (between components), Configuration and Status registers, Power intent. Often specified with IP-XACT, spread-sheets or the like.

Value domains, Tri-states, weak-driver resolution

Value domains

- Verilog wire uses a domain with 4-values {0,1,X,Z}
  - Z is high-impedance value / not-driving.
  - X is unknown value / ternary X
- The language specifies the behaviour of gates in the 4-value domain, f.ex.
  - assign a = b & 1'bX; // a is 0 iff b is 0, otherwise X
  - assign c = d & ~d; // c could be X even though it seem it should always be 0
  - if (a) d <= 1'b1; else d <= 1'b0; // d can't be X, it is 0 if a is 0 or X (or Z) otherwise 1
  - d <= a ? 1'b1 : 1'b0; // here d can be X, if a is X.
- Sometimes explicitly used in simulation to catch bad behaviour.
  - As they propagate far and are easy to check for on the outputs (when viewing a trace).
  - Problems like when x's vanish, as in the if-statement above, is referred to as X optimism.
- In the resulting implementation (silicon) result there are no ternary X's though!

Value domains

- Literal ‘X’ will often be interpreted by synthesis tools as don’t-care condition
  - Allowing it to create smaller / better netlists.
  - Targeting formal verification this has to be prevented, to preserve all possible behaviour
- Example:
  - ```
    case(sig[1:0])
        2’b10: a <= 1’b1;
        2’b01: a <= 1’b0;
        default: a <= 1’bx;
    endcase
    ```
  - If the design ensures that ’sig’ here will only ever take the value b01 and b10
- Specialized formal models to check that X’s can’t propagate / interfere.
  - For this a differential encoding is used, and not a ternary (dual-rail) encoding!
  - This is to more closely match silicon, rather than simulation, behaviour.
    ```

Tri-states, weak-driver resolution

- Special care is also need to modelling is Tri-state elements (and weak drivers).
  - Tri-states conditionally drive a value, or hold it's output isolated.
  - The Z is high-impedance / not-driving value is involved here.
  - There are 'weak' constant 0 and 1 drivers, called pulldown and pullup.
- *Resolving* the drivers means replacing the gates, that drive a common wire, with a model for the resolved logic value
  - At most one non-weak driver? or maybe all strong drivers must have drive the same value..
  - If no strong driver, take value from a weak driver
    - must not have weak drivers driving different values
  - If any of the conditions are violated allow a 'free' (indeterminate) value
  - Can generate asserts (like lint/sanitizer checks) for the necessary conditions.
- It's also possible to specify 'strengths' on primitive gates (how much current can be driven). For logic correctness this mostly does not matter.

Inout / high-impedance example

```
module cregs(inout[15:0] mdata, input rst, clk, oe, we, input [1:0] addr, …)
  reg[15:0] data_out, reg_addr0, reg_addr1;
  wire[15:0] data_in;

  assign mdata = oe ? data_out : {16{1'bz}};
  assign data_in = mdata;

  always @(posedge clk) begin
    data_out <= addr == 2'd0 ? reg_addr0 :
                        addr == 2'd1 ? reg_addr1 : 16'd0;
    if (we && addr == 2'd0) reg_addr0 <= data_in;
    if (we && addr == 2'd1) reg_addr1 <= data_in;
end
```

Constraints and environment modeling

Constraints and environment modeling

- It is common to have configurations, modes of operation or parts that should not be validated.
- Forcing inputs to fixed values is referred to as pin, or environment, *constraints.*
  - Examples include disabling Test mode, Scan logic, tying power/ground ‘supply’ to 1/0, etc.
  - The verification plan might be divided such that each mode is verified independently.
  - Inputs not allowed to change during operation, use stability or pseudo-constant constraints.
- Input that should provide clocks, that have a fixed periodic behaviour.
  - Can often be given as a period & phase or an explicit repeating pattern.
- More complex constraints are normally considered part of the verification setup
  - Custom SV assumptions and support logic.
  - Standardized protocols might exist as pre-packaged IP modules.
  - Companies might also have re-usable collateral for internal interfaces/protocols.
  - In equivalence checking there is ‘reference’ implementation, not strictly a constraint.

Model reduction under environment constraints

- The simple constraints that give rise to fixed and periodic values can fairly easily be leveraged to reduce the representation.

- Rebuilding using structural hashing can handle constants, and trivially equivalent gates, but it's possible to do better.
- Ternary simulation* can also handle periodic signals
  - Equivalent ones can be merged, and normalized representation used for the remaining
  - *A cheap way to compute over-approximate 'reachability', using X for unknow inputs/state.

- Often leads to the set-of-support being reduced for many functions
- The now disconnected parts can be dropped (or disregarded in later stages)
  - Commonly referred to as Cone of influence (COI) reduction

Over-constraining & Dead-ends

- Almost all papers make an assumption that is often not true for our models.
- “The transition relation must be total, for every state s, there exists state s’, s.t. R(s,s’).”

- When the transition relation is the conjunction of (Boolean) next-state functions this hold because the functions are total.
- With constraints, meaning functions that must evaluate to True, it is easy to become non-total if they are part of the transition relation
  - A function can’t evaluate to True for some particular state (for any input values)
  - A combination of functions can’t all evaluate to True for some particular state …

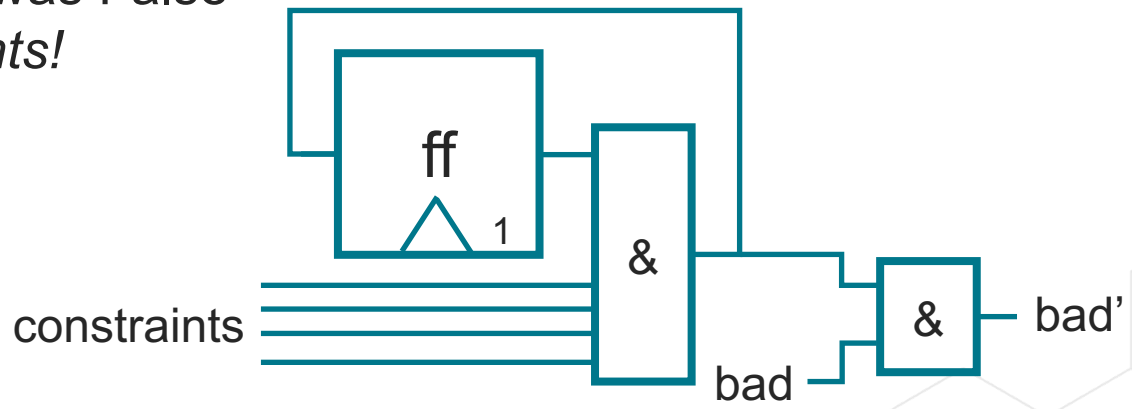
Over-constraining & Dead-ends

- A state that has no next-state (not even it self) is referred to as a dead-end.
- If all paths eventually dead-end, we say that the design is over-constrained.

- Does a safety property that has a finite contradicting evaluation on a over-constrained design fail?
  - The constraints are not ‘always’ satisfied…
- However that would mean that every proof would have to ensure that there exist an infinite future satisfying all the constraints.
- Users might also be surprised
  - ‘I can clearly see my property contradicted by this trace, why did the tool miss that?’
  - ‘Because your constraint will be violated in the future…’

Over-constraining & Dead-ends

- The established convention is that constraints do not need to hold after a failure.
- This causes some problem too
  - When does a property fail?
    - is ‘X False’, contradicted now, or only after one time step?
    - The compilation of the properties to automat / circuits need to carefully consider this.
  - Effects of constraints must not be applied too early, or too late…
    - Must take extra care when extracting invariants or making temporal transforms
- Common implementation strategy is to ‘fold’ constraints into the property check
  - Property becomes non-bad if any constraint was False
  - *Reachable states are not limited by constraints!*



Initialization / Reset

Initialization

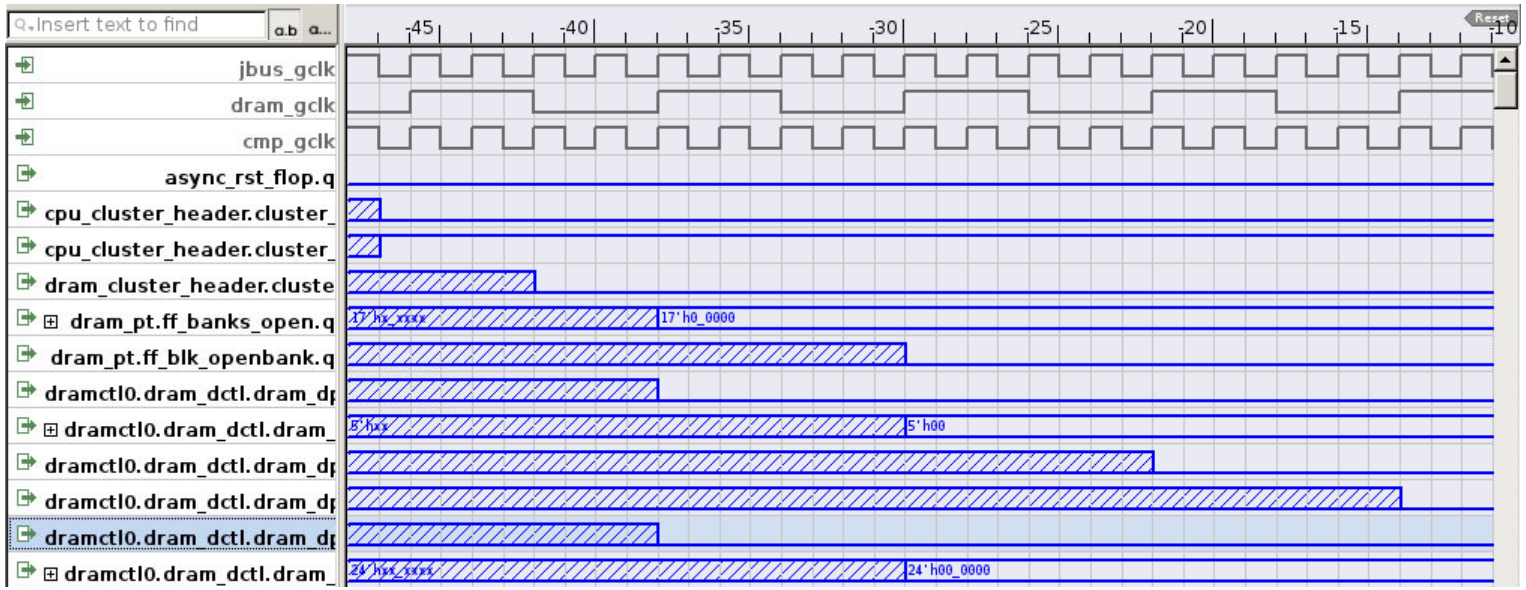
- In simulation it is possible to simply force a particular initial value
  - A code block that will only execute initially before main simulator event loop.
    - Verilog has ‘initial’ blocks for this that, unlike ‘always’ blocks, only execute once.
  - Using ‘force’ statements from the test bench code (test driver).

- On the hardware side, ‘starting’ means applying power to the design.
  - The state elements is best modelled as having arbitrary (unknown) values when coming from an un-powered state.
- To initialize the hardware a cold/hard reset is done
  - That is a sequence of inputs are used to transition the system from any state to one of set of states from which it will exhibit predictable (and hopefully correct) behaviour.

Flop reset

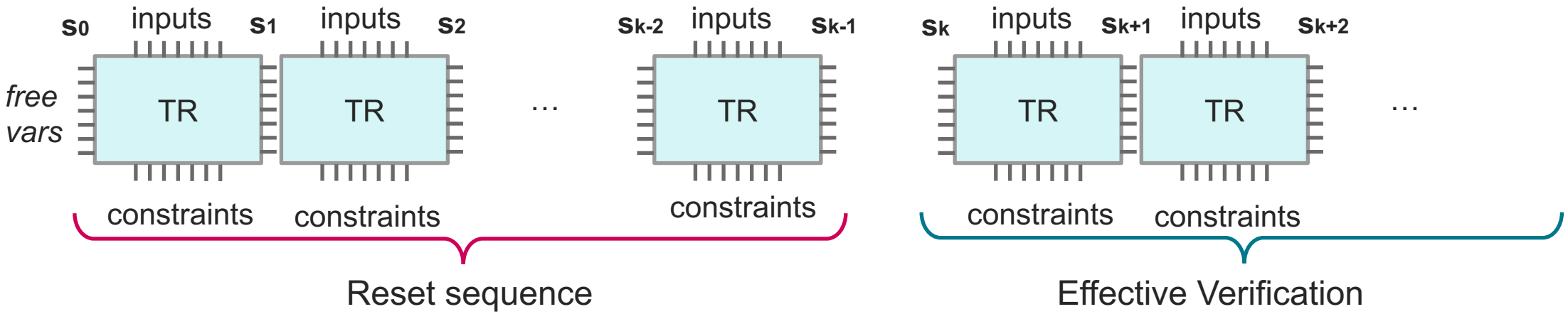
- Design are written to enable being initialized with reset.
- Flops can have asynchronous reset (not depending on the clock)
  - always @(posedge clk or posedge rst)
    if (rst)
      areg <= 1'b0;
    else
      reg <= areg_nextstate;
- Synchronous reset, or receive values from other flops and/or inputs
  - always @(posedge clk) sreg <= rst ? 1'b0 : sreg_nextstate; // Reset if clk is know to 'tick'
  - always @(posedge clk) reg <= ns; // Reset correctly if 'ns' gets a reset value
- Yet other might be left uninitialized if their value does not affect the behaviour.
  - i.e., updated before they are used in a way to will influence something else

- Simple reset is a few signals held active a fixed number of cycles
  - Convention is often for reset signals to be active low (e.g False / 0)
    - Commonly denoted with a ‘n’ or ‘l’ prefix/suffix.
- Examples from OpenSPARC T1 / Niagra
  - reset -expression ~dram_arst_l ~dram_adbginit_l ~jbus_grst_l ~dram_grst_l ~cmp_grst_l
  - `INFO (IRS018): Reset analysis simulation executed for 48 iterations. Assigned values for 6517 of 15784 design flops, 0 of 4 design latches, 486 of 486 internal elements.`



Initialization / Reset

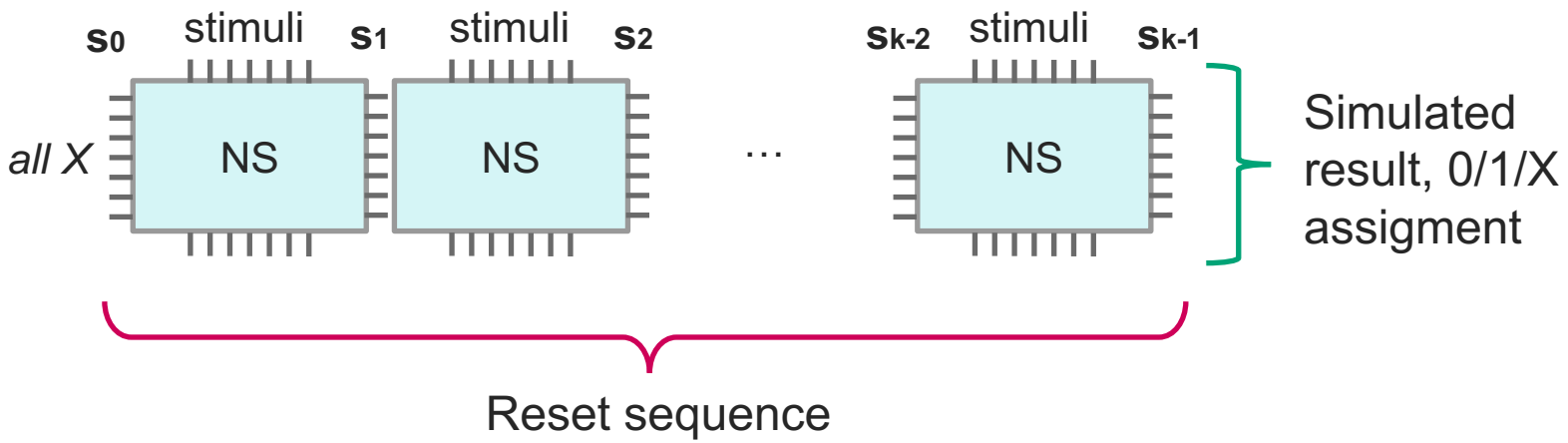
- The formal model would start from every design state (all state variable free) and apply the reset sequence as temporal constraints.
- This is exact but makes the problem sequentially deeper (by the length of the reset sequence).



- Temporal decomposition can be used to partition the problem, avoiding or significantly reducing the overhead.

Initialization / Reset

- Due to the reset sequence commonly being specified only on inputs, ternary simulation can be used to derive an over approximate set of initial states.
  - Initial state values and inputs not specified in the sequence are taken as ternary X



- Any X in the simulated end state is treated as free value (input) for the formal Model Checking problem.

Clocks and Discrete Time Models

Clocks and creating a discrete time model

- For power and performance reasons it is common that designs are multi-clocked, or that clocks are gated (can be turned off and on).

- To have global *synchronous model* for verification we need to reduce these multi-clock systems to a single global system (or tool) clock.
- It's assumed that there is one common domain (all changes are synconous)
  - Checking the interaction, when this is not assumed, is referred to as Clock Domain Crossing

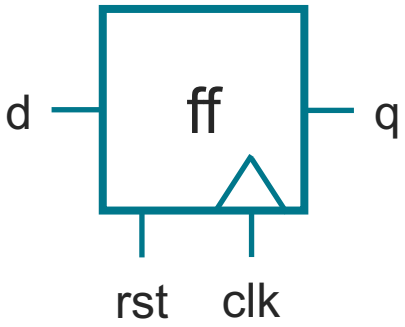
- The environment constraints specify the rate of change of the inputs.
  - If clock are not specified with a fixed pattern all possible rates and interleavings will be allowed. This can create prohibitively difficult model checking problems though.

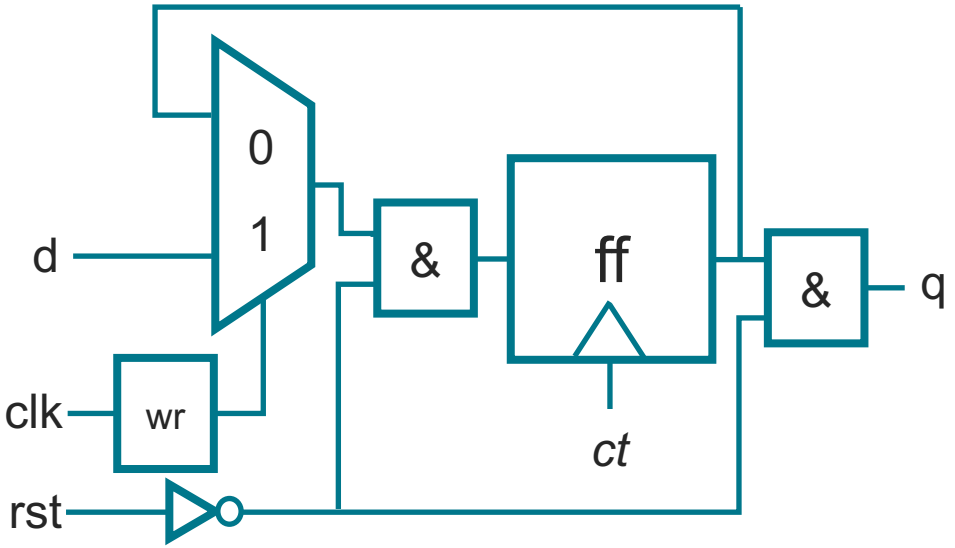
Clocks and creating a discrete time model

- Flops update on the rising edge of their clock.
  - always @(posedge clk) // ‘execute’ the block when there is an posedge event
    flop_name <= driver;
- Latches are transparent when enable is high and hold their value when its low.
  - always @(enable or driver)
    if (enable)
    latch_name <= driver;
- Inputs are free to take any value, but environment specifies the ‘rate’
  - Clock generators, as mentioned, give their inputs a deterministic, periodic, pattern.
- Flops can also have additional asynchronous inputs
  - A reset/set (or both) that will update the flop value regardless of if the clock has a rising edge

Clocks and creating a discrete time model

- The basis of the model is handled by mux-feedback added for the flops/latches
  - A state/delay element that update on the global clock.
  - Need to create the condition under which the value will be updated.
- Flop with reset and clock

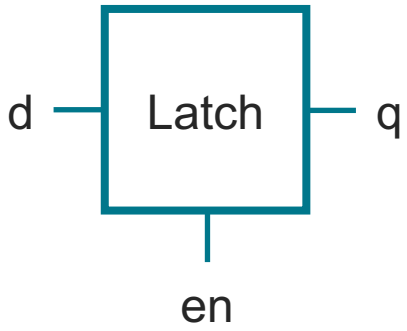


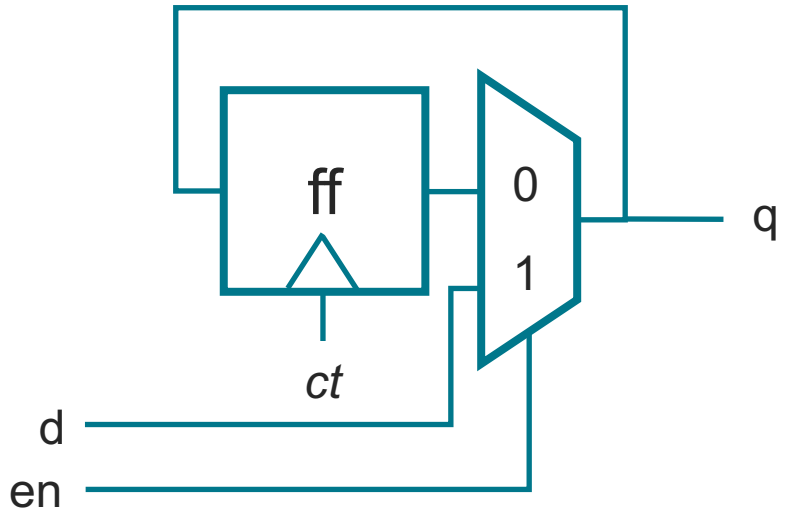


  - next(state) := !rst & (wr(clk) ? d : state)
  - wr(clk) := !clk & next(clk)
  - q := !rst & state

Clocks and creating a discrete time model

- The basis of the model is handled by mux-feedback added for the flops/latches
  - A state/delay element that update on the global clock.
  - Need to create the condition under which the value will be updated.
- Latch





  - next(state) := en ? d : state
  - q := en ? d : state

Feedback and loops

Feedback and loops

- In real hardware there will be some delay between an input changing and when the effect can be seen on the gate output.
  - There can also be limits to how closely together (in time) two inputs may change (e.g. the data input of a flop must be stable for some time before the clock edge).
  - This is what limits the rate of the clock (the MHz/GHz one hears about).
- In formal verification we would prefer to not deal with this.
  - There are other tools that validate these timing constraints.

- Informally a zero-delay model means 'infinite speed of updates'
  - i.e. gate inputs & outputs always have consistent values (without delay)
- No net can have more than one value within / per time point
  - Matches normal Boolean functions/predicates

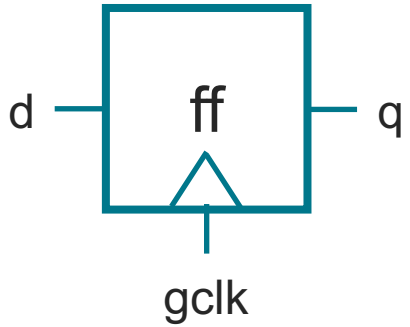
Feedback and loops

- Feedback loops in the netlist could cause contradictions
  - When a net would have two (or more) values, had there been a non-zero delay propagating values through the gates.

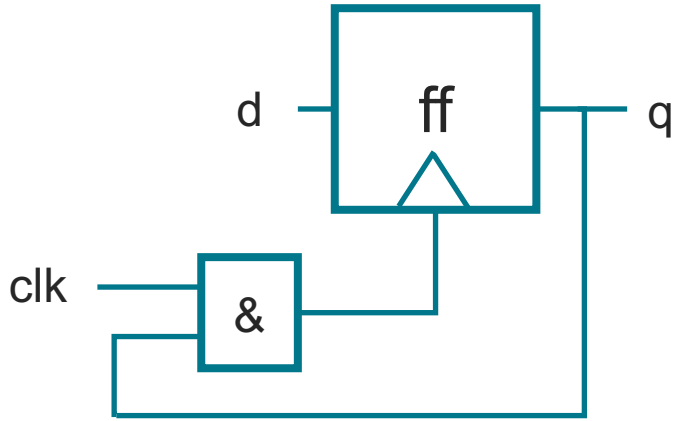
Type	Potential Issues
Flop - data	None - Formal friendly (there is time step delay)
Flop - clock	Glitches
Latch - data	Stable / Unstable loops
Latch - enable	Multiple open/close events
Combinatorial gates	Ruled out by construction? (Stable / Unstable loops)

Flop clock feedback

- Recall the flop model (here without reset)
  - next(state) := wr(gclk) ? d : state
  - wr(gclk) := !gclk & next(gclk)
  - q := state



- What if ‘clk’ has ‘q’ in it’s support (fan-in)?
  - next(state) := wr(gclk) ? d : state
  - wr(gclk) := !gclk & next(gclk)
  - gclk := clk & state
  - q := state



- Humm…
  - wr(gclk) := !(clk & state) & next(clk) & next(state)
  - next(state) := !(clk & state) & next(clk) & next(state) ? d : state
  - NOT GOOD: if state is True, then either d must be True, or !clk &next(clk) must be False!

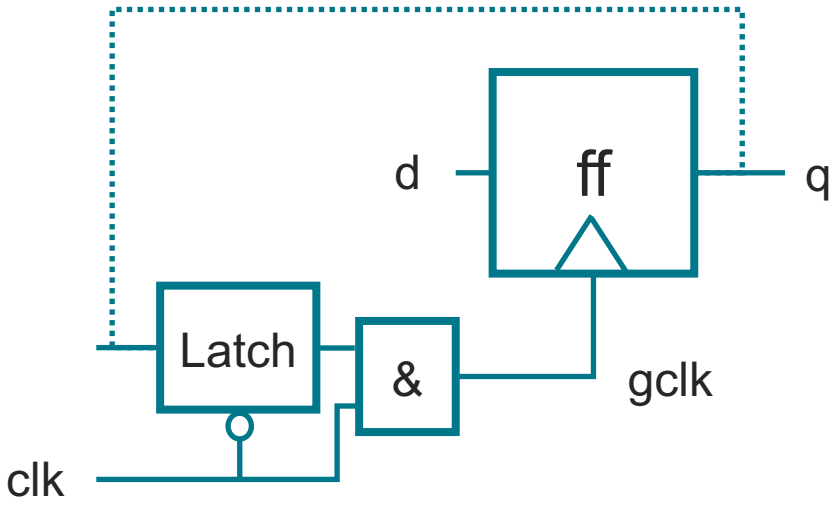
Flop clock feedback

- Mostly, in correct designs, this does not happen.
  - To ensure that the clock does not change twice a latch is used to stabilize the path.

  - next(Fstate) := wr(gclk) ? d : Fstate
  - wr(gclk) := !gclk & next(gclk)
  - q := Fstate

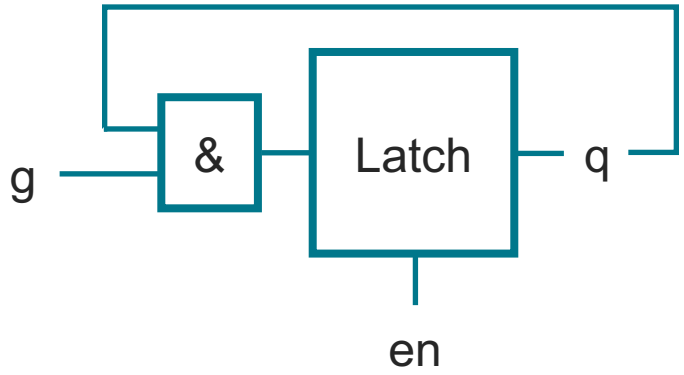
  - next(Lstate) := !clk ? q : Lstate
  - lq := !clk ? q : Lstate

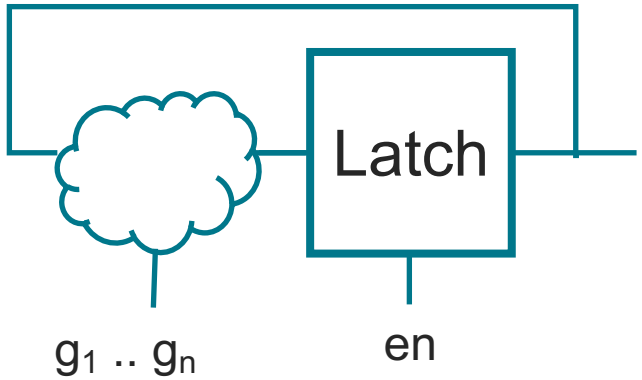
  - gclk := lq & clk
  - … gclk := (!clk ? q : Lstate) & clk
  - … gclk := Lstate & clk
  - … wr(gclk) := !(Lstate & clk) & next(Lstate) & next(clk)
  - … wr(gclk) := !(Lstate & clk) & (!clk ? q : Lstate) & next(clk)
  - … wr(gclk) := q & !clk & next(clk) // e.g. q & wr(clk)



Latch data feedback

- Latches have a direct (combinatorial) path when enabled
  - feedback from the output back to its data input can create a loop





- Functional loop condition
  - en & g
  - en & *function*(g_1 .. g_n)
- If the condition is False the model is unproblematic.
  - In benign cases the '*function*(g_1 .. g_n)' will be False whenever 'en' is True
  - Often there are pairs of latches with exclusive enable conditions

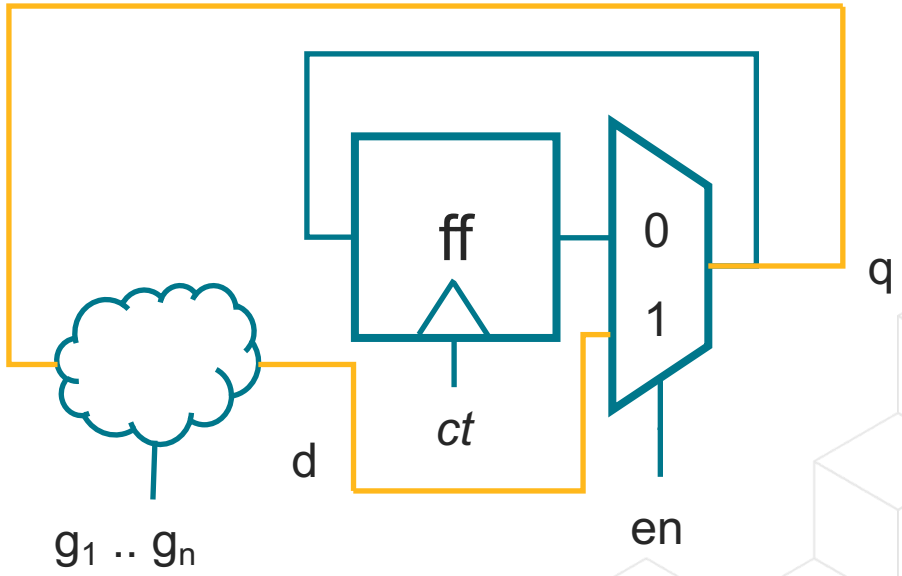
Latch data feedback

- Latches have a direct (combinatorial) path when enabled
  - feedback from the output back to its data input can create a loop

- When ‘en & *function*(g_1 .. g_n)’ is True the path is open
  - For a polarity inverting path, there will be a contradiction (d == !q && q == d).
  - Essentially ruling out models where ‘en & *function*(g_1 .. g_n)’ would be True.

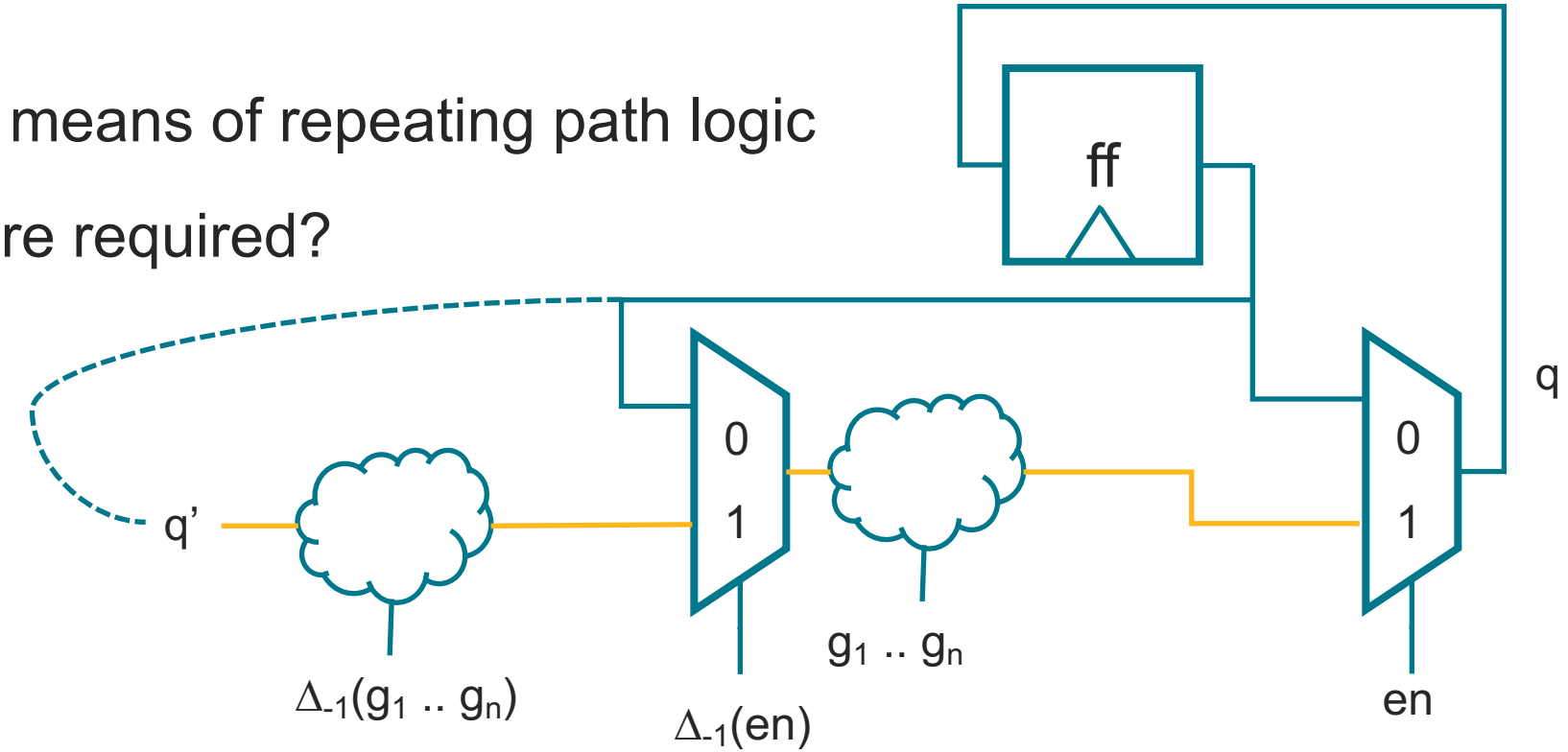
- For a polarity preserving path ’q’ will be free
  - It will be independent even from the history (state in ‘ff’)

- Changing value without any cause is a divergence from simulation as well as silicon behaviour.



Feedback and loops

- Adding ‘delta’ delay by means of repeating path logic
- How many such step are required?



- If the path is false, q’ is irrelevant (not observable at q).
- If the path is true, and the extra steps did not resolve the functional dependency
  - Polarity preserving: q and there for q’ should maintain its value from the previous ‘full’ cycle.
  - Polarity inverting: The value of q will be inverted or not depend how many step where used…

Feedback and loops

- As seen the zero-delay model has some undesirable properties when there are loops in the netlist.

- Adding ‘delta’ delay on the path?
  - Can lead to a large increase of the representation if there are many loops
  - Will work if (a conservative limit on) the number of ‘delta’ steps needed can be determined.
  - However for oscillating loops there is no upper limit.
    - A final ‘break’ or use of prior full will still be needed.

- Can we solve it by adding extra check to flag netlists that have these issues?
  - Deriving the loop condition (’*function*(g_1 .. g_n)’) is a challenge in the general case.