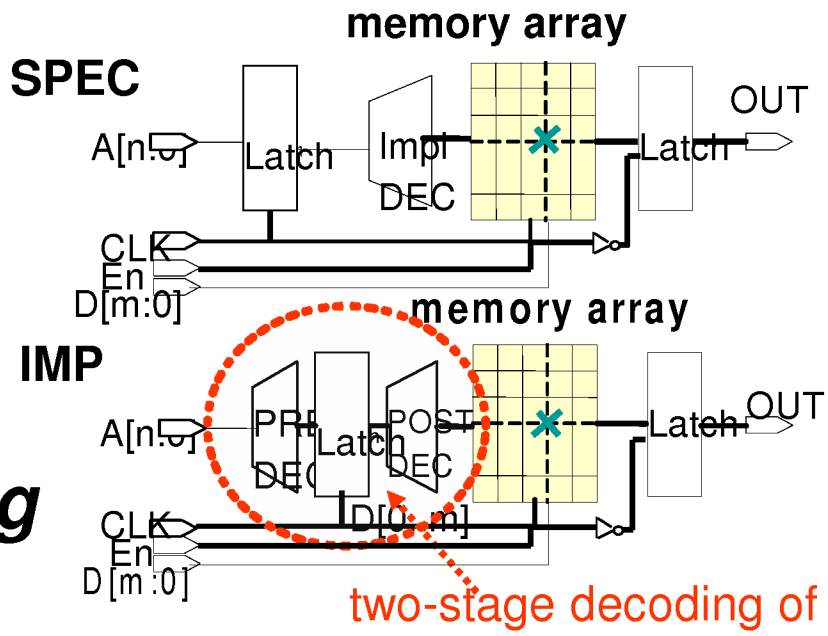
Verifying Equivalence of Memories using a FOL Theorem Prover

Zurab Khasidashvili (Intel)
Mahmoud Kinanah (Intel)
Andrei Voronkov (The University of Manchester)

Presented by:
Nikolaj Bjørner (Microsoft Research)

In a nutshell



New methods for *Equivalence Checking*

Old

Bit-level encoding:

Bit-vectors are *vectors of Boolean propositions*

SAT: BDD, DPLL

New

Algebraic:

Bit-vectors are *un-interpreted constants*

Really New and Super Cool and Fast

Relational:

Bit-vectors are *relations*

FO/EPR: Vampire iProver, Darwin, …

Overview

- Formal Equivalence Checking (FEC) of RTL and SCH memory designs today
- The challenges
- Identification of memories and decoders
- Decoder abstraction
- Encodings in FOL
- Experimental results
- Future work

Formal Equivalence Checking (FEC)

The Purpose of Equivalence Checking

- The same functional behavior can be implemented in many different ways
- The implementation must be optimized wrt:
  - Timing – to achieve better performance
  - Power – to reduce power consumption and ensure longer battery life
  - Area – to produce smaller computer chips
- We need to prove that hardware design optimization does not change the functional behavior

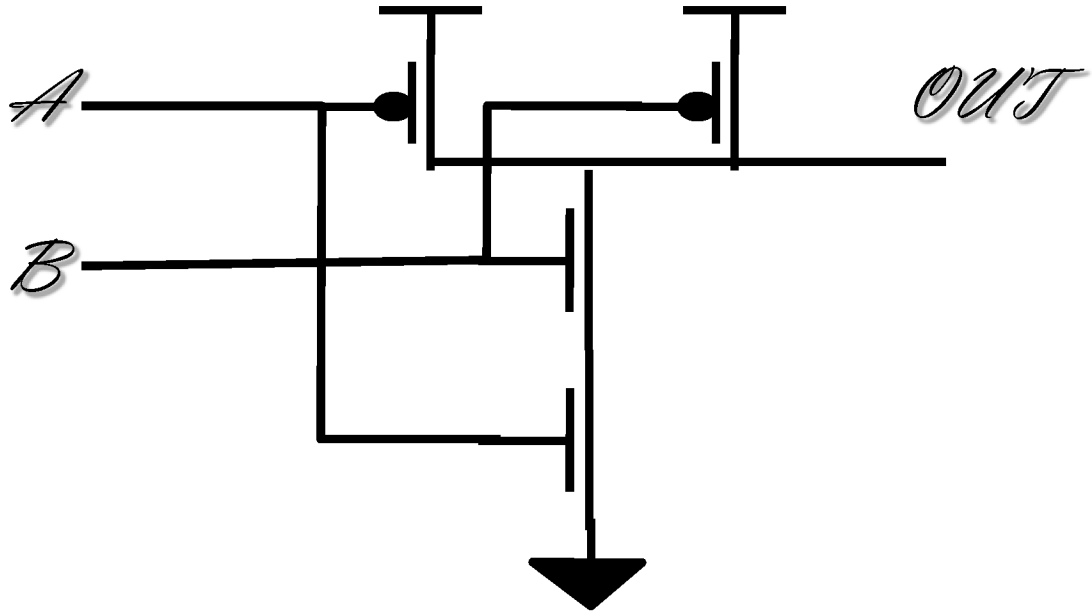
RTL Model

- Register-Transfer Level description written in a hardware description language (Verilog, System Verilog, etc.) looks like:

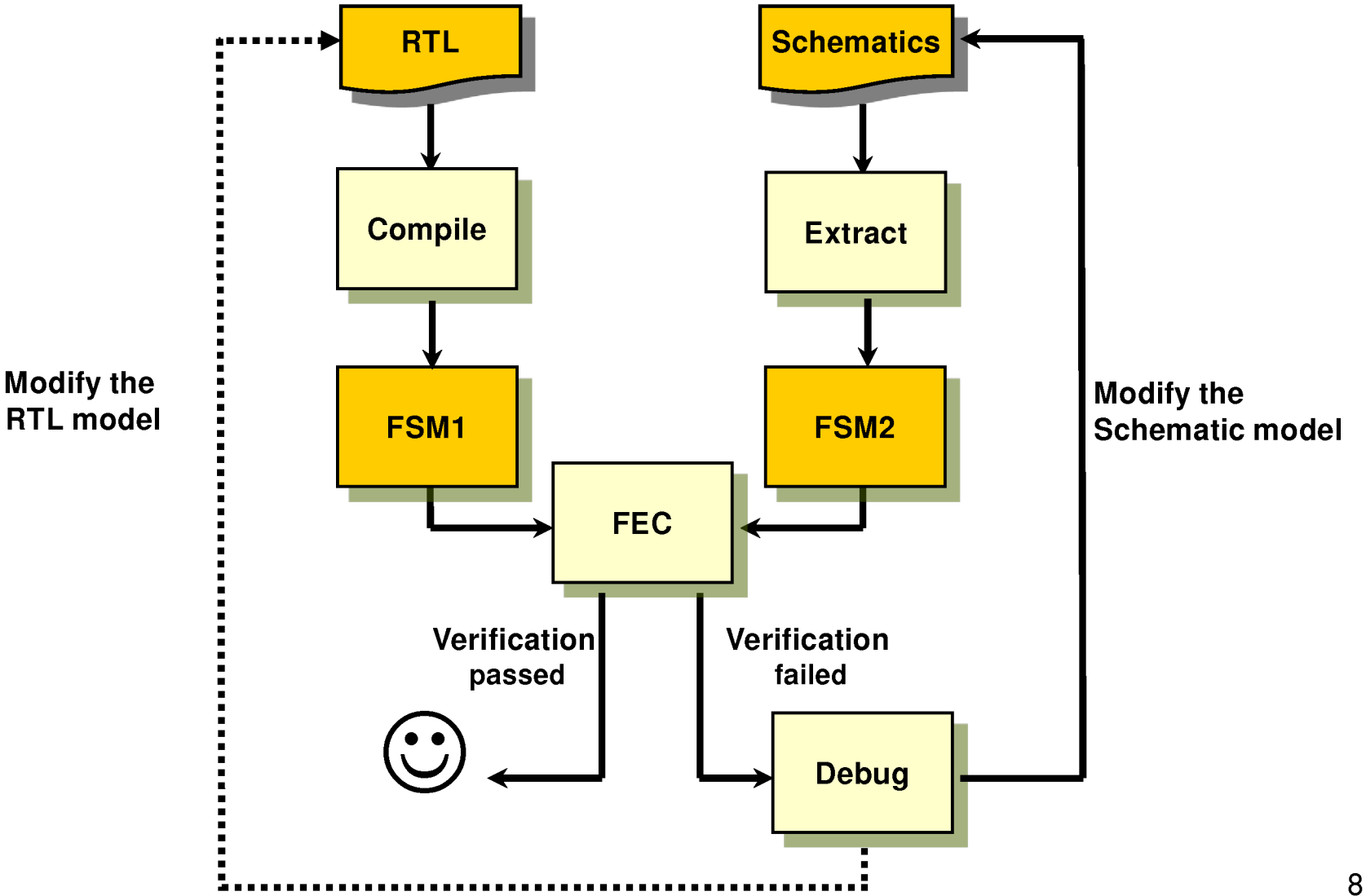
```
always_latch begin
    for(int portnum = 0; portnum <= (WR_PORTS-1); portnum++)
        if(!ckwrcbout[portnum])
          for(int i = WR_LATENCY-1; i > 0; i = i-2)
               LAT_Wr[portnum][i] <= LAT_Wr[portnum][i-1];
end
```

Schematic model

A	B	¬ (A&B)
0	0	1
0	1	1
1	0	1
1	1	0

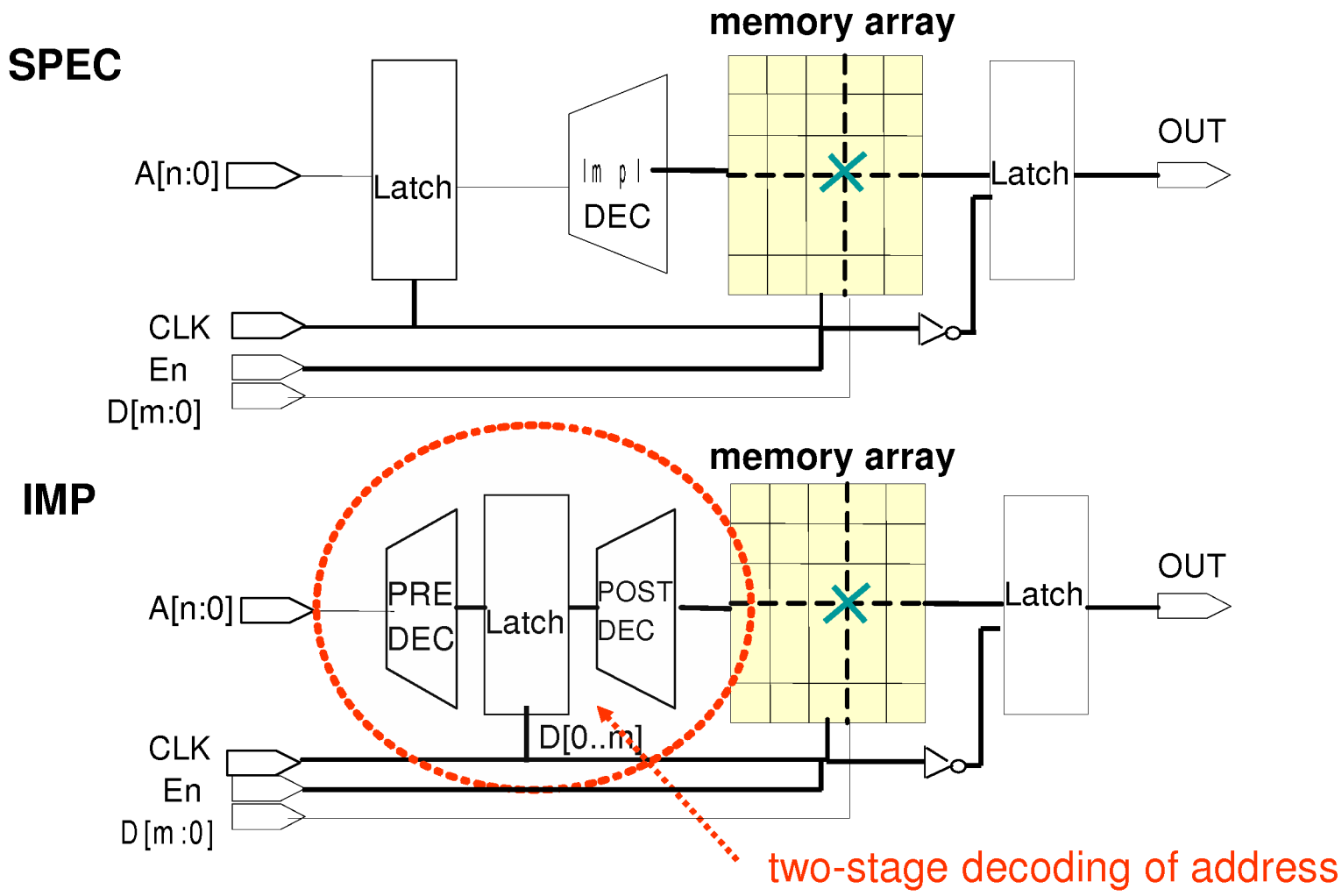


Basic FEC flow



FEC of RTL and SCH memory designs today

Non-State-Matching coding for Arrays



The challenges (1/2)

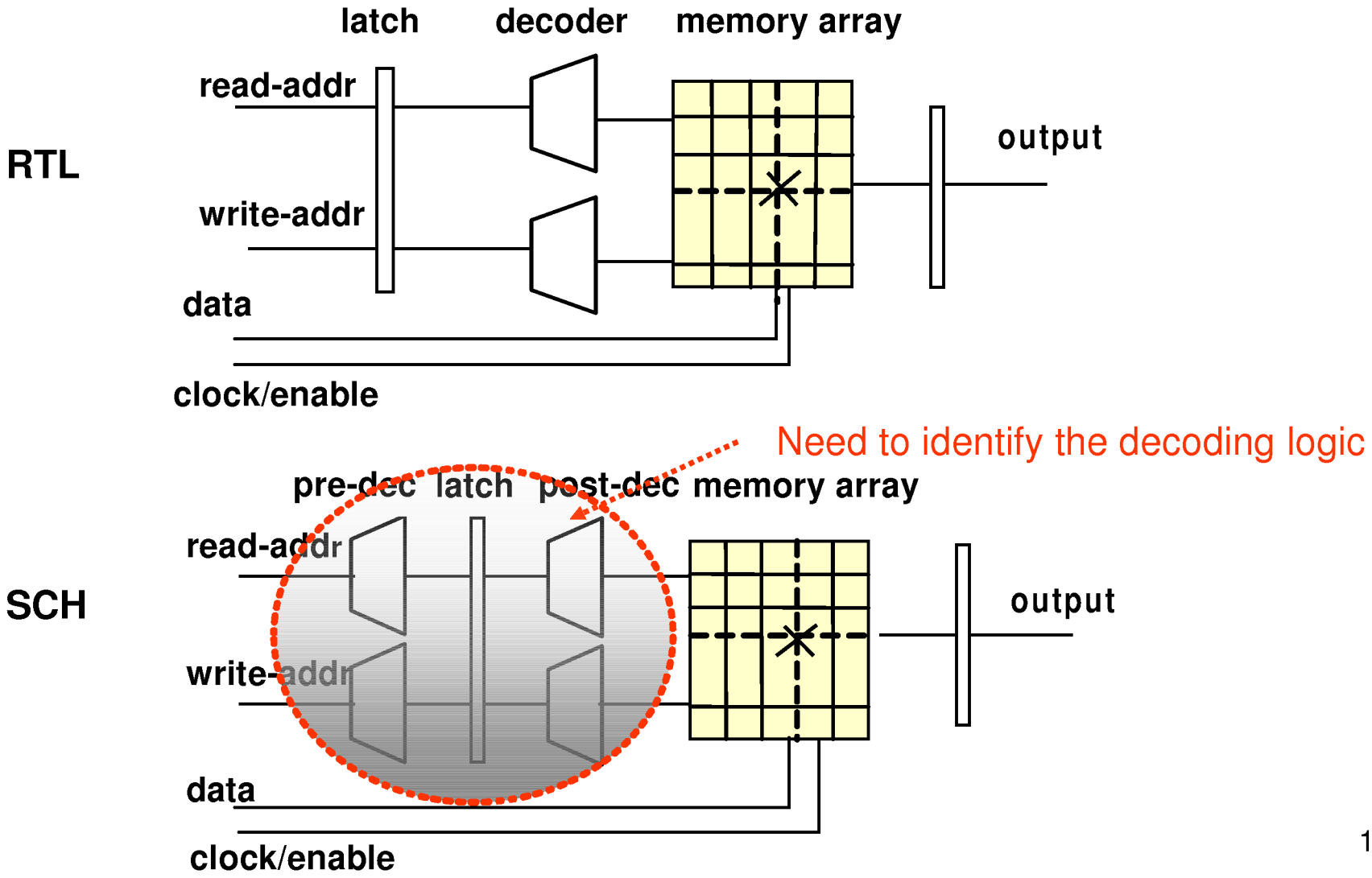
- It takes 1-2 days of user effort to get the mapping of memory cells right
  - Each pair of “mapped” memory cells in the two memory designs must have the same values (in post-reboot states)
- FEC of two memories (with given mapping) may take hours and often runs into complexity
  - Each mapped pair of memory cells must be proved equivalent; the read data in the two models must also be proved equivalent
  - Dedicated model-checking strategies are needed, tuning the strategies is a non-trivial task
- The new method has a major impact on
  - memory FEC productivity -- aiming at less mapping and debugging effort, and
  - overall FEC effort -- arrays constitute ~50% of the chip area and 20% of RTL (number of Functional Unit Blocks -- FUBs)

The challenges (2/2)

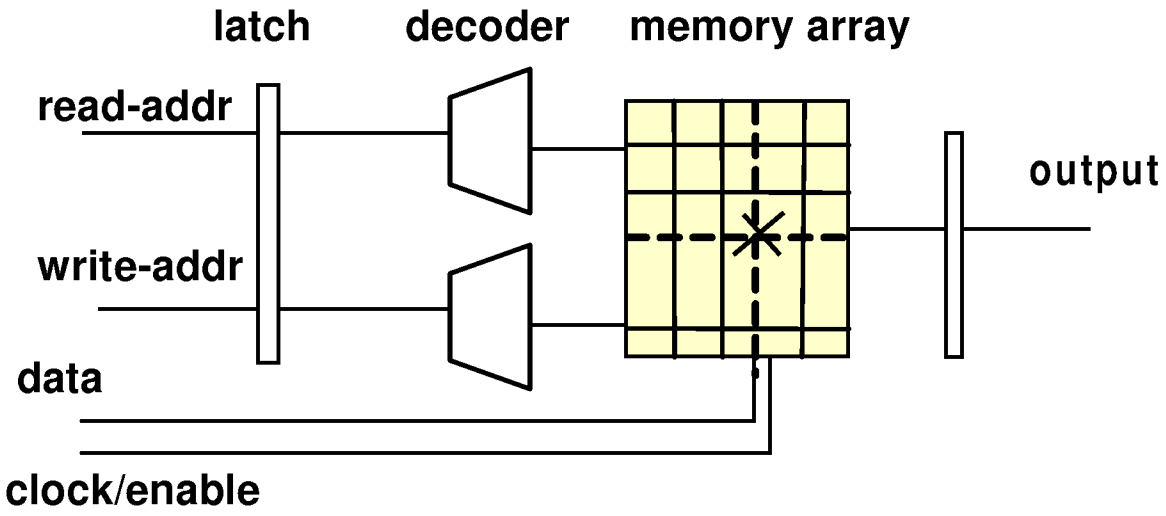
- We need to develop a more efficient memory FEC method with
  - powerful abstraction
  - modular verification
  - negligible mapping effort
  - minimized need for assumptions
  - modular debugging

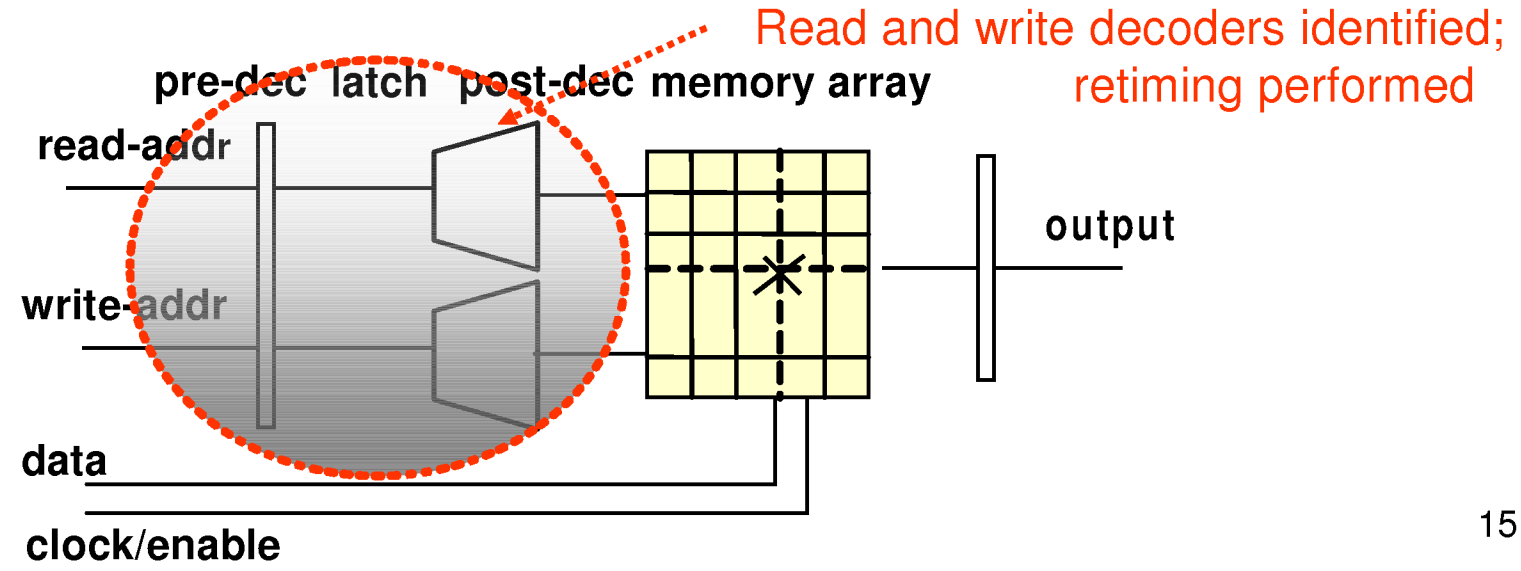
Extraction of memories and decoders from SCH model

Identifying the decoders

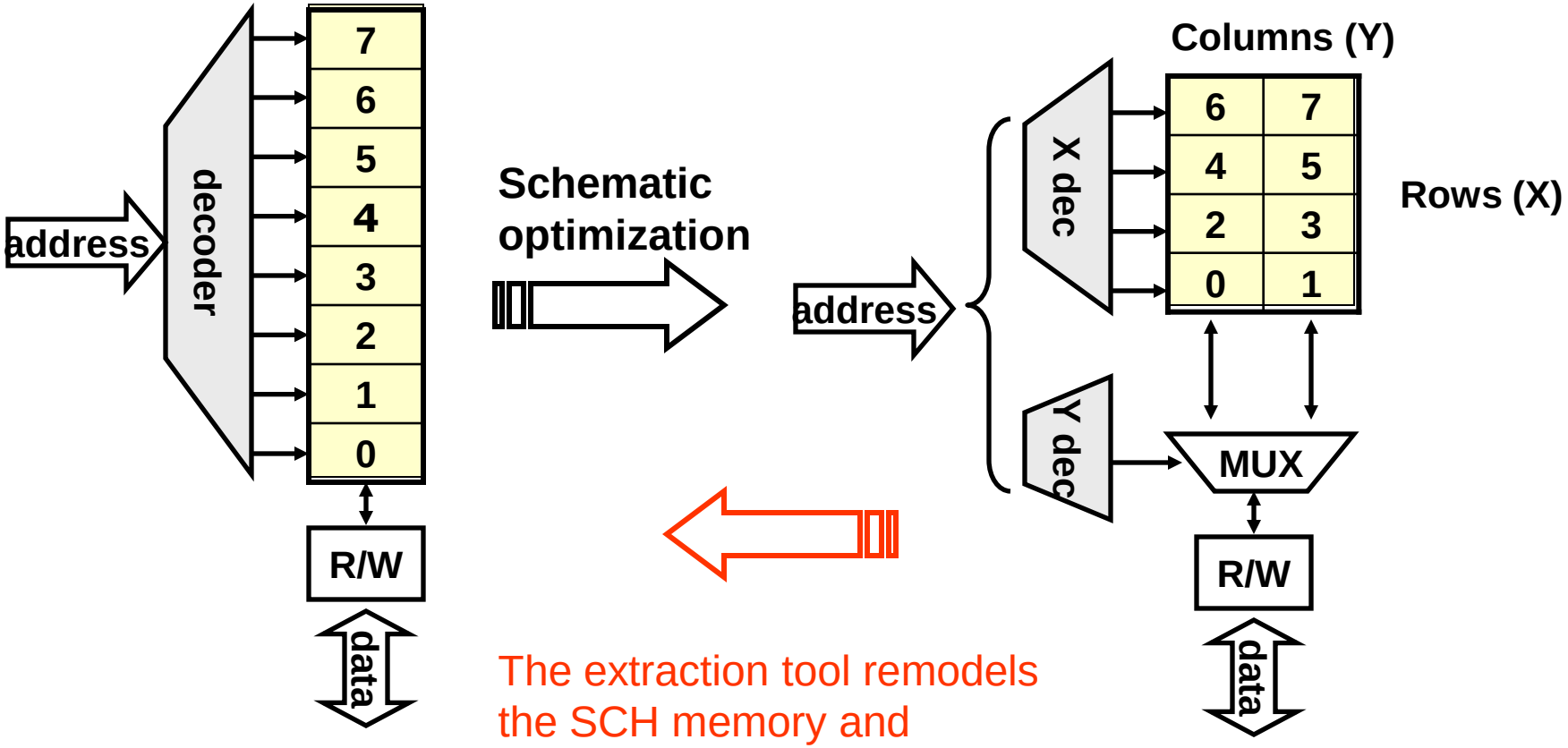


Identifying the decoders

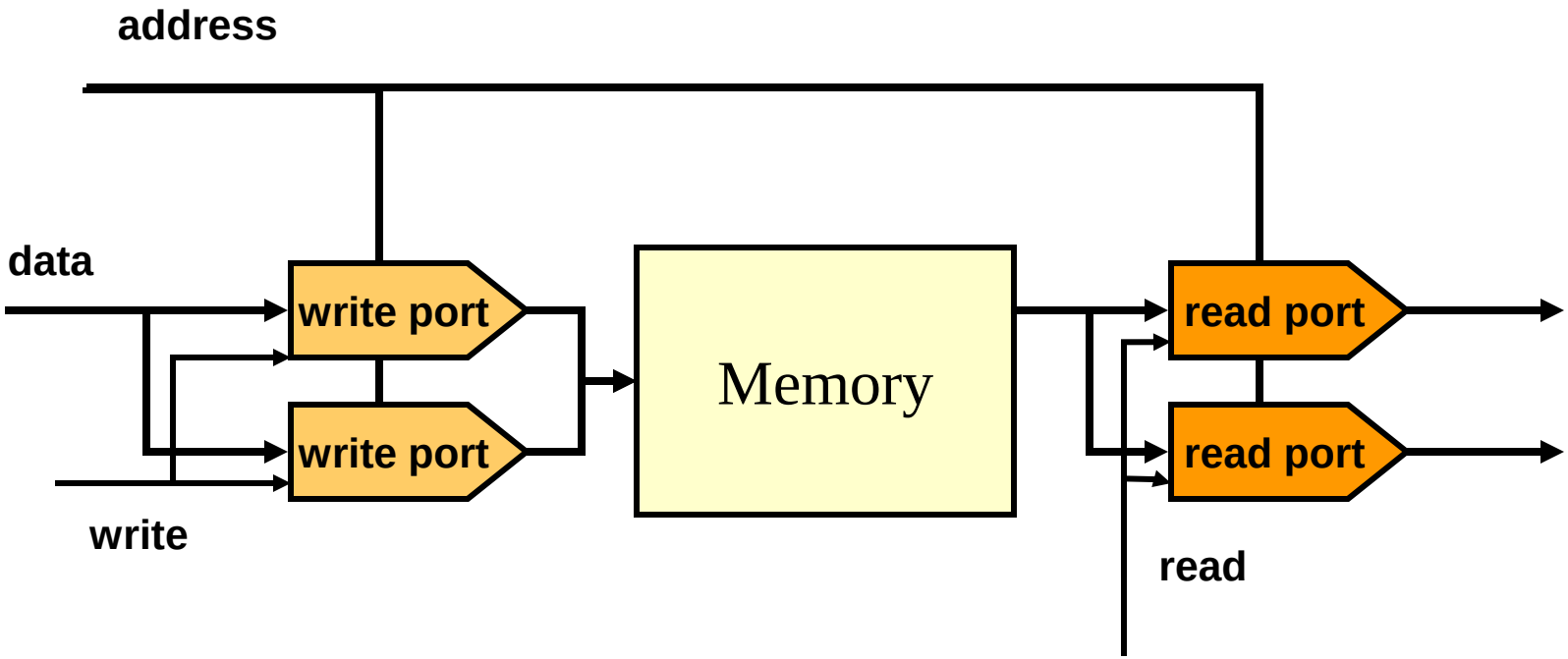




Different layout in spec and imp memories



Memory with two read and write ports



The relevant modeling is done in verification front-end and during building of the invariant formula

Decoder Abstraction

Axiom for decoding in RTL: decoding correctness

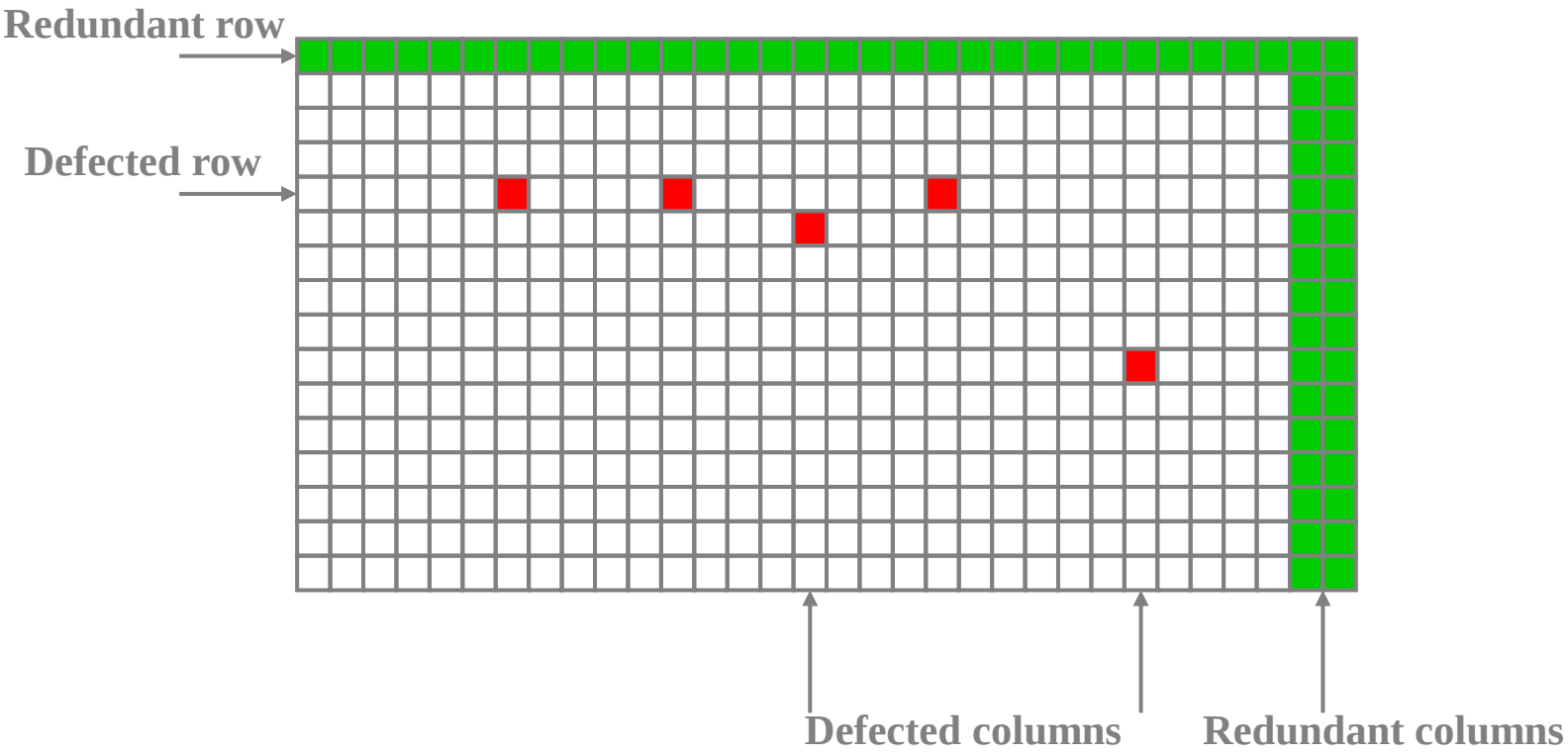
- In the RTL (=spec) model, there is an implicit decoder, **decs,** which is (implicitly) used both during write/update and read/value operations
- We need to assume that decs is 1-1: encoded addresses X and Y are equal iff the corresponding decoded addresses are
- We call this axiom ***decoding correctness***
  - $\forall X, Y : (decs(X) = decs(Y)) \Rightarrow (X = Y)$

Axioms for decoding in SCH: read-write consistency

- In the SCH (=imp) model, we have a write decoder **wdeci** and read decoder **rdeci**
- We need to assume that wdeci and rdeci are 1-1 (i.e., wdeci and rdeci are correct), and that they are the same functions – we call the later ***read-write decoding consistency*** axiom:
  - $\forall X\ \forall Y :\ (\text{rdeci}(X) = \text{rdeci}(Y)) \Rightarrow (X = Y)$
  - $\forall X\ \forall Y :\ (\text{wdeci}(X) = \text{wdeci}(Y)) \Rightarrow (X = Y)$
  - $\forall X :\ \ \text{rdeci}(X) = \text{wdeci}(X)$

Decoding correctness

- The axiom is valid also in the situation where there are redundant (i.e., inaccessible) rows in the memory array



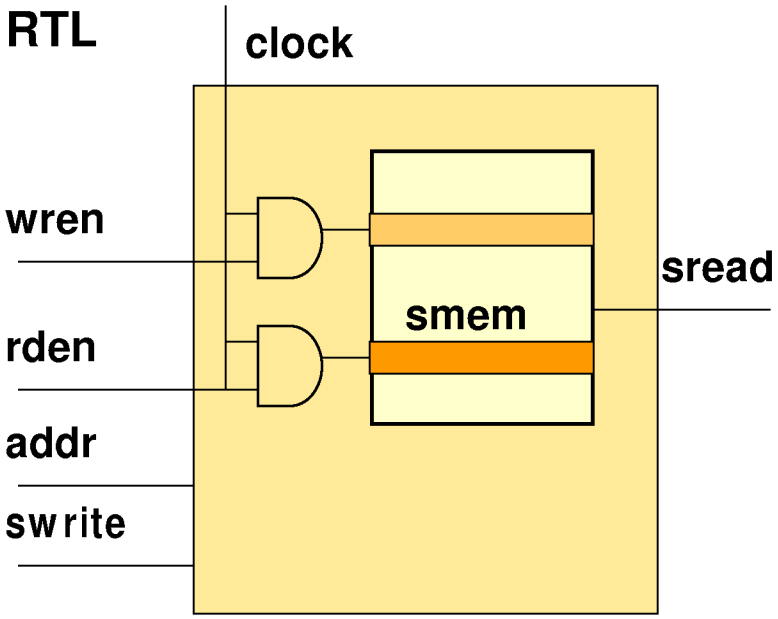
Checking decoding correctness and consistency axioms

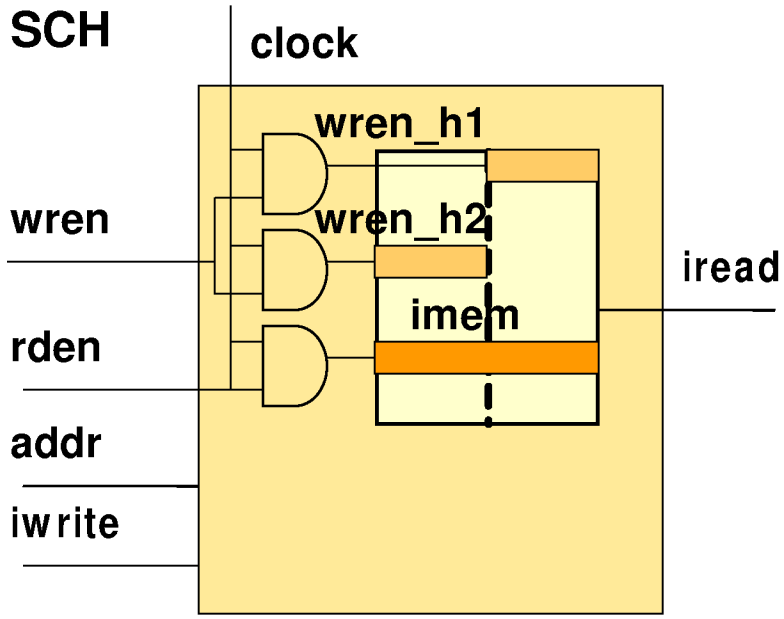
- The decoding correctness axiom in RTL model is **correct by construction**, based on how the RTL compiler defines the address decoders for memories
- The decoding correctness axioms in SCH model and the read-write consistency axiom in SCH model **are checked** based on the analysis that the circuit extraction tool performs in order to identify encoded memories
- **In the encoding to FOL, the decoders are abstracted away and are treated implicitly**

Encodings to FOL Solvers

By example

Toy RTL and SCH memories





Write data in SCH memory is written in 2 halves;
wren_h1 and wren_h2 are the enables

Encoding to FOL solvers – Relational Approach

- Bit-vectors are unary relations:
  - bv(*B*) - the *B*-th bit of bv
- Memories are binary relations:
  - *mem*(*A*,*B*) - the value of *B*-th bit at address *A*
- Bit-selection/concatenation: use sub-range:
  - $B \in [35:0]$ - constrain index to be in range
  - $B \leq n$ - constrain index to be below/above *n*

Relational: Bit-vectors are *relations*

Relational Approach = Effectively Propositional Fragment = EPR

EPR: $\exists y \forall x \varphi(y,x)$ - decidable

Skolemize: $\forall x \varphi(c,x)$ - finite Herbrand universe

Clauses:

$$\forall x_1, x_2, x_3 (R(x_1, x_2, c_1) \vee \neg Q(x_3, x_2, c_2) \vee x_1 = c_2)$$

$$\forall x_1, x_2 (R(x_2, x_1, x_1) \vee x_2 = c_1 \vee x_2 = c_2)$$

Propositional:

$$(R(c_1, c_1, c_1) \vee \neg Q(c_1, c_1, c_2) \vee c_1 = c_2) \wedge$$

$$(R(c_1, c_2, c_1) \vee \neg Q(c_1, c_2, c_2) \vee c_1 = c_2)$$

....

Recently several important problems (model checking, planning) were encoded into EPR, and FOL solvers are optimized for this class

- There is a FOL TP competition category of EPR formulas

Relational: Bit-vectors are *relations*

Encoding next state functions

Next State Functions for RTL/spec memory array

$$\forall A\left(\begin{array}{c}(wren \wedge clock \wedge A = addr)\\ \Rightarrow smem'(A) = swrite)\end{array}\right)$$

$$\forall A\left(\begin{array}{c}\neg(wren \wedge clock \wedge A = addr)\\ \Rightarrow smem'(A) = smem(A))\end{array}\right)$$

$$\forall A\left(\begin{array}{c}(wren \wedge clock \wedge A = addr)\\ \Rightarrow \forall B(smem'(A,B) \Leftrightarrow swrite(B))\end{array}\right)$$

$$\forall A\left(\begin{array}{c}\neg(wren \wedge clock \wedge A = addr)\\ \Rightarrow \forall B(smem'(A,B) \Leftrightarrow smem(A,B))\end{array}\right)$$

Encoding next state functions

Next State Functions for RTL/implementation memory array

$$wren_{h1} \Leftrightarrow wren \wedge clock$$
$$wren_{h2} \Leftrightarrow wren \wedge clock$$
$$\forall A \left(\begin{array}{c} (wren_{h1} \wedge A = addr) \\ \Rightarrow imem'(A)[35:0] = iwrite[35:0] \end{array} \right)$$
$$\forall A \left(\begin{array}{c} \neg(wren_{h1} \wedge A = addr) \\ \Rightarrow imem'(A)[35:0] = imem(A)[35:0] \end{array} \right)$$
$$\forall A \left(\begin{array}{c} (wren_{h2} \wedge A = addr) \\ \Rightarrow imem'(A)[71:36] = iwrite[71:36] \end{array} \right)$$
$$\forall A \left(\begin{array}{c} \neg(wren_{h2} \wedge A = addr) \\ \Rightarrow imem'(A)[71:36] = imem(A)[71:36] \end{array} \right)$$

Algebraic

$$wren_{h1} \Leftrightarrow wren \wedge clock$$
$$wren_{h2} \Leftrightarrow wren \wedge clock$$
$$\forall A \left(\begin{array}{c} (wren_{h1} \wedge A = addr) \\ \Rightarrow \forall B(B \in [35:0] \Rightarrow imem'(A,B) \Leftrightarrow iwrite(B)) \end{array} \right)$$
$$\forall A \left(\begin{array}{c} \neg(wren_{h1} \wedge A = addr) \\ \Rightarrow \forall B(B \in [35:0] \Rightarrow imem'(A,B) \Leftrightarrow imem(A,B)) \end{array} \right)$$
$$\forall A \left(\begin{array}{c} (wren_{h2} \wedge A = addr) \\ \Rightarrow \forall B(B \in [71:36] \Rightarrow imem'(A,B) \Leftrightarrow iwrite(B)) \end{array} \right)$$
$$\forall A \left(\begin{array}{c} \neg(wren_{h2} \wedge A = addr) \\ \Rightarrow \forall B(B \in [71:36] \Rightarrow imem'(A,B) \Leftrightarrow imem(A,B)) \end{array} \right)$$

Relational

Encoding output functions

- sread, sread':

$$\forall B(sread'(B) \Leftrightarrow ite(clock \wedge ren, smem(addr, B), sread(B)))$$

$$sread' = ite(clock \wedge ren, smem(addr), sread))$$

- iread' of iread is defined similarly

Bit-vectors:

Algebraic

$$concat(extract(71,36,X),extract(35,0,X)) = X$$

$$extract(71,36,neg(X)) = neg(extract(71,36,X))$$

$$extract(35,0,neg(X)) = neg(extract(35,0,X))$$

Extraction: a function

Bit-vector not using function *neg*

vs. Relational

$$B \in [35:0] \Leftrightarrow B = 0 \vee .. \vee B = 35$$

$$or:$$

$$B \in [35:0] \Leftrightarrow B \leq 35$$

Extraction: implicit

Bit-vector not built-in

The Equivalence Conjecture

$$corr(Ms, Mi, os, oi) \Rightarrow corr(Ms', Mi', os', oi')$$

Where

$$corr(Ms, Mi, os, oi) \Leftrightarrow \forall A : Ms(A) = Mi(A) \wedge os = oi$$

Algebraic

$$corr(Ms, Mi, os, oi) \Leftrightarrow \forall A, B : \begin{pmatrix} Ms(A, B) \Leftrightarrow Mi(A, B) \\ \wedge (os(B) \Leftrightarrow oi(B)) \end{pmatrix}$$

Relational

Modular verification and debugging

The mismatch types; modularity of debugging

- Mismatch caused by an incorrect decoder in imp – this should be detected as part of decoder recognition in the extraction tool (pre FEC activity)
- Mismatch caused by inconsistent read and write decoding in imp – this should be detected as part of decoder recognition in the extraction tool (pre-FEC)
- Mismatch caused by differences in the read or write delays – this must be detected when checking the verification invariant in FOL TP
- There is no need for mapping the memory cells

Experiments

- Toy and real-life memory equivalence instances can be solved in seconds in Vampire FOL solver
- For this class of problems, the relational axiomatization is much more efficient than the algebraic one
- We have also run the Darwin solver on some real-life memory equivalence problems
- Currently some manual optimizations are involved in generating TPTP instances from equivalence checking problems
- The flow will be productized and deployed in Intel's sequential equivalence checking tool, Seqver.

Vampire Results: Algebraic vs Relational Encodings

Test number	Translation type	Run time (seconds)	Memory (Mbytes)	Generating inferences	Simplifying inferences	Result status
1	algebraic	5.7	52.2	189,388	194,474	unsatisfiable
	relational	0.1	19.4	7.896	10,507	unsatisfiable
2	algebraic	1025.5	286.9	55,663,356	49,702,181	unknown
	relational	0.1	18.5	5,824	7,093	satisfiable
3	algebraic	4.4	35.4	167,702	173,345	unsatisfiable
	relational	0.1	19.4	11,767	15,107	unsatisfiable
4	algebraic	938	286.9	20,861,220	22,070,968	unknown
	relational	0	18.5	3,052	4,114	satisfiable

Future Work

- The counter-examples returned by Darwin that we have inspected were meaningful and useful for debugging
-
- However, because our abstraction of bit-vectors is forgetful of their widths, meaningless counter-examples and false positives are also possible

- It is a challenging problem to refine the abstraction so that false negatives will be impossible

- We are also exploring usage SMT solvers for FEC of memories based on decoder abstraction, and we are comparing the performance of SMT and FOL solvers

Thank You!