Sequential Equivalence Checking - I

Virendra Singh
Associate Professor
Computer Architecture and Dependable Systems Lab.
Dept. of Electrical Engineering
Indian Institute of Technology
Bombay
viren@ee.iitb.ac.in

EE 709: Testing & Verification of VLSI Circuits

Lecture – 14 (Feb 02, 2012)

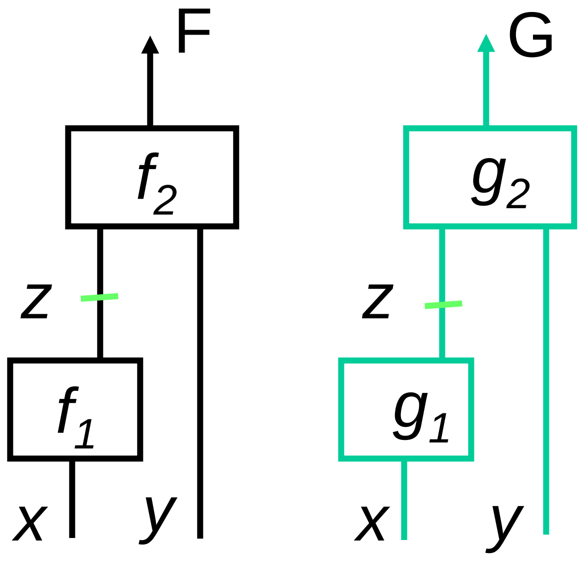
Functional Equivalence

- If BDD can be constructed for each circuit
  - represent each circuit as *shared* (multi-output) BDD
    - use the same variable ordering !
  - BDDs of both circuits must be *identical*

- If BDDs are too large
  - cannot construct BDD, memory problem
  - use partitioned BDD method
    - decompose circuit into smaller pieces, each as BDD
    - check equivalence of internal points

Functional Decomposition

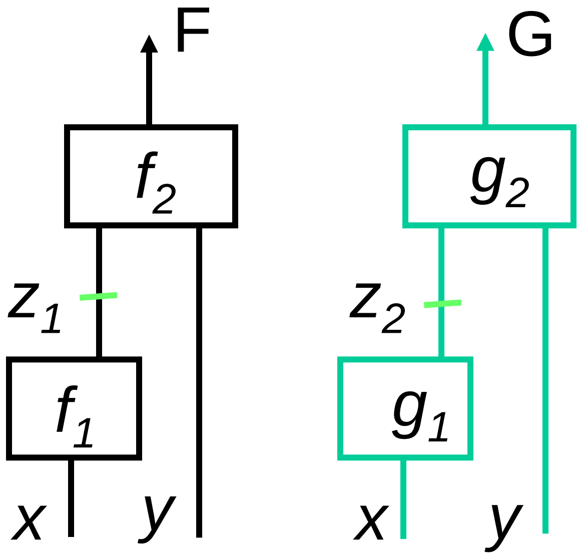
- Decompose each function into *functional* blocks
  - represent each block as a BDD (*partitioned BDD* method)
  - define *cut-points* (*z*)
  - verify equivalence of blocks at cut-points
  - starting at primary inputs



Cut-Points Resolution Problem

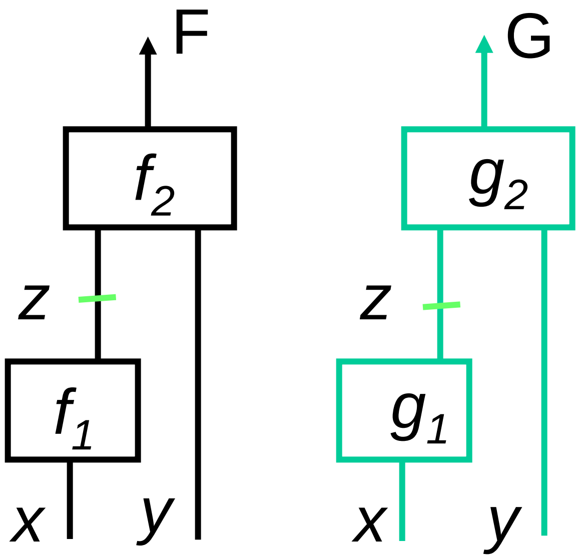
- If *all pairs* of cut-points (z_1,z_2) are equivalent
  - so are the two functions, F,G
- If *intermediate* functions (f_2,g_2) are not equivalent
  - the functions (F,G) may still be equivalent
  - this is called false negative

- Why do we have false negative ?
  - functions are represented in terms of *intermediate* variables
  - to prove/disprove equivalence must represent the functions in terms of *primary inputs* (BDD composition)



Cut-Point Resolution – Theory

- Let $f_1(x)=g_1(x)\ \forall x$
  - if $f_2(z,y) \equiv g_2(z,y),\ \forall z,y$ then $f_2(f_1(x),y) \equiv g_2(f_1(x),y) \Rightarrow F \equiv G$
  - if $f_2(z,y) \neq g_2(z,y),\ \forall z,y \not\Longrightarrow f_2(f_1(x),y) \neq g_2(f_1(x),y) \not\Rightarrow F \neq G$



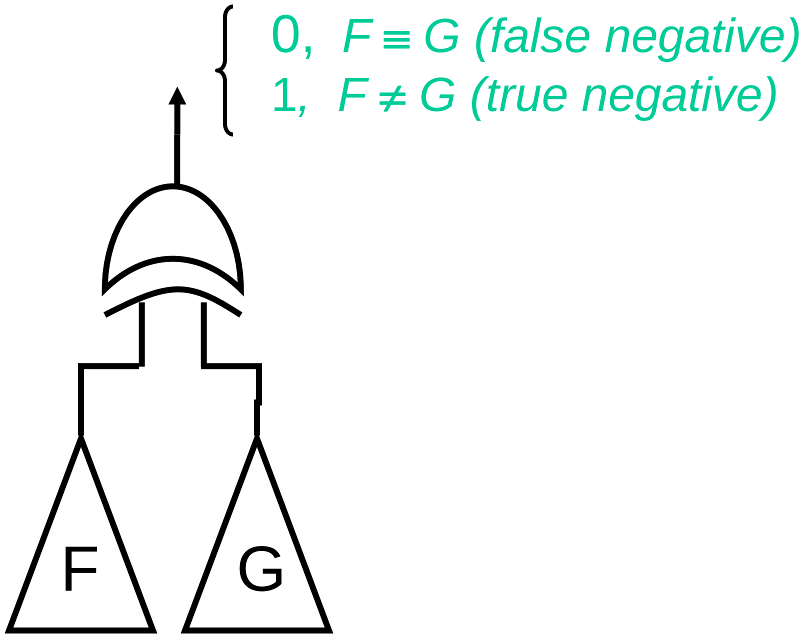
We *cannot* say if $F \equiv G$ or not

- *False negative*
  - two functions are equivalent, but the verification algorithm declares them as different.

Cut-Point Resolution

- How to verify if negative is *false* or *true* ?

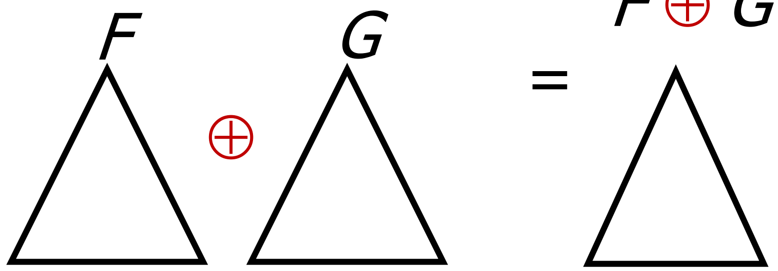
- Procedure 1: create a miter (XOR) between two potentially equivalent nodes/functions
  - perform ATPG test for *stuck-at 0*
  - find test pattern to prove $F \neq G$
  - efiicient for true negative
  - (gives *test vector*, a proof)
  - inefficient when there is no test



$\begin{cases} 0, & F \equiv G \text{ (false negative)} \\ 1, & F \neq G \text{ (true negative)} \end{cases}$
F
G

Cut-Point Resolution

- Procedure 2: create a BDD for $F \oplus G$
  - ➢ perform satisfiability analysis (SAT) of the BDD
    - if BDD for $F \oplus G = \varnothing$, problem is *not* satisfiable, *false* negative
    - BDD for $F \oplus G \neq \varnothing$, problem is satisfiable, *true* negative



$$F \oplus G = \begin{cases} \varnothing, & F \equiv G \text{ (false negative)} \\ \text{Non-empty}, & F \neq G \end{cases}$$

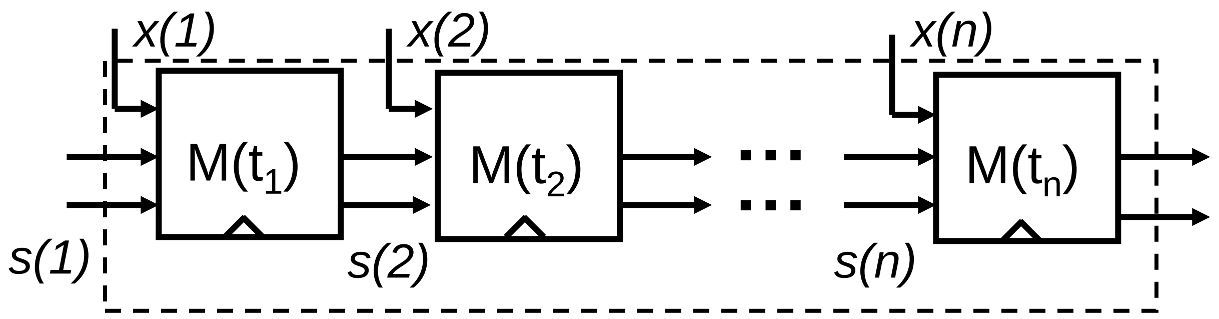
Note: must compose BDDs until they are equivalent, or expressed in terms of primary inputs

– the SAT solution, if exists, provides a *test vector* (proof of non-equivalence) – as in ATPG

– unlike the ATPG technique, it is effective for false negative (the BDD is empty!)

Sequential Equivalence Checking

- Represent each sequential circuit as an FSM
  - verify if two FSMs are equivalent
- Approach 1: Reduction to *combinational* circuit
  - unroll FSM over *n* time frames (flatten the design)

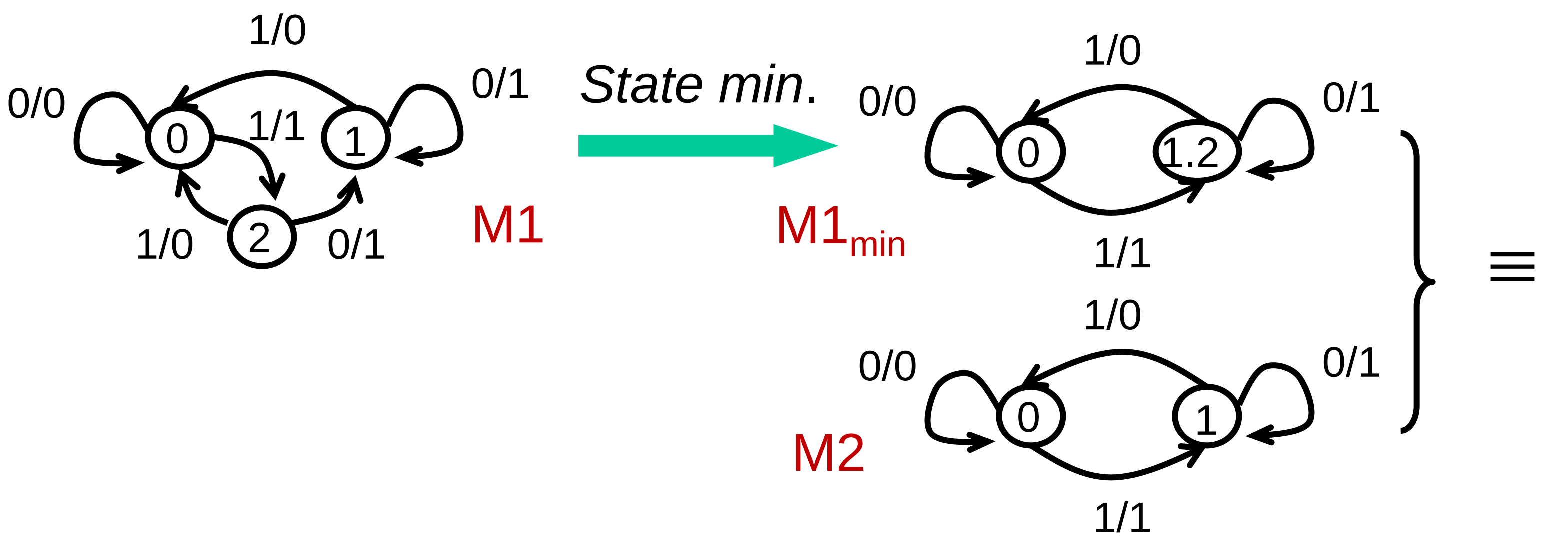


Combinational logic: *F(x(1,2, …n), s(1,2, … n))*

- check equivalence of the resulting combinational circuits
- problem: the resulting circuit can be too large too handle

Sequential Verification

- Approach 2: Based on isomorphism of state transition graphs
  - two machines M1, M2 are *equivalent* if their state transition graphs (STGs) are *isomorphic*
  - perform state minimization of each machine
  - check if STG(M1) and STG(M2) are isomorphic



State Minimization

X-Successor – If an input sequence X takes a machine from state S_i to state S_j, then S_j is said to be the X-successor of S_j

Strongly connected:- If for every pair of states ($S_{i,}$ S_j) of a machine M there exists an input sequence which takes M from state S_i to S_j, then M is said to be strongly connected

State Equivalence

- Two states S_i and S_j of machine M are distinguishable if and only if there exists at least one finite input sequence which, when applied to M, causes different output sequences, depending on whether S_i or S_j is the initial state
- The sequence which distinguishes these states is called a distinguishing sequence of the pair $(S_{i,}\ S_j)$
- If there exists for pair $(S_{i,}\ S_j)$ a distinguishing sequence of length k, the states in $(S_{i,}\ S_j)$ are said to be k-distinguishable

State Equivalence

Machine M1

PS	NS, z	
	X = 0	X = 1
A	E, 0	D, 1
B	F, 0	D, 0
C	E, 0	B, 1
D	F, 0	B, 0
E	C, 0	F, 1
F	B, 0	C, 0

(A, B) – 1 Distinguishable

(A, E) – 3 Distinguishable

Seq - 111

k-equivalent – The states that are not *k*-distinguishable are said to be k-equivalent

Also *r-equivalent* r<k

State Equivalence

- States S_i and S_j of machine **M** are said to be equivalent if and only if, for every possible input sequence, the same output sequence will be produced regardless of whether S_i or S_j is the initial state
- States that are k-equivalent for all $k < n-1$, are equivalent
- $S_i = S_j$, and $S_j=S_k$, then $S_i=S_k$

State Equivalence

- The set of states of a machine M can be partitioned into disjoint subsets, known as equivalence classes
- Two states are in the same equivalence class if and only if they are equivalent, and are in different classes if and only if they are distinguishable

Property: If S_i and S_j are equivalent states, their corresponding X-successors, for all X, are also equivalent

State Minimization Procedure

1. Partition the states of M into subsets s.t. all states in same subset are *1-equivalent*
2. Two states are 2-equivalent iff they are 1-equivalent and their I_i successors, for all possible I_i, are also 1-equivalent

PS	NS, z	
	X = 0	X = 1
A	E, 0	D, 1
B	F, 0	D, 0
C	E, 0	B, 1
D	F, 0	B, 0
E	C, 0	F, 1
F	B, 0	C, 0

P0 = (ABCDEF)

P1 = (ACE), (BDF)

P2 = (ACE), (BD), (F)

P3 = (AC), (E), (BD), (F)

P4 = (AC), (E), (BD), (F)

Machine Equivalence

- Two machines M1, M2 are said to be equivalent if and only if, for every state in M1, there is corresponding equivalent state in M2
- If one machine can be obtained from the other by relabeling its states they are said to be isomorphic to each other

PS	NS, z	
	X = 0	X = 1
AC - α	β, 0	γ, 1
E - β	α, 0	δ, 1
BD - γ	δ, 0	γ, 0
F - δ	γ, 0	α, 0

State Equivalence - Example

Machine M2

PS	NS, z	
	X = 0	X = 1
A	E, 0	C, 0
B	C, 0	A, 0
C	B, 0	G, 0
D	G, 0	A, 0
E	F, 1	B, 0
F	E, 0	D, 0
G	D, 0	G, 0

P0 = (ABCDEFG)

P1 = (ABCDFG) (E)

P2 = (AF) (BCDG) (E)

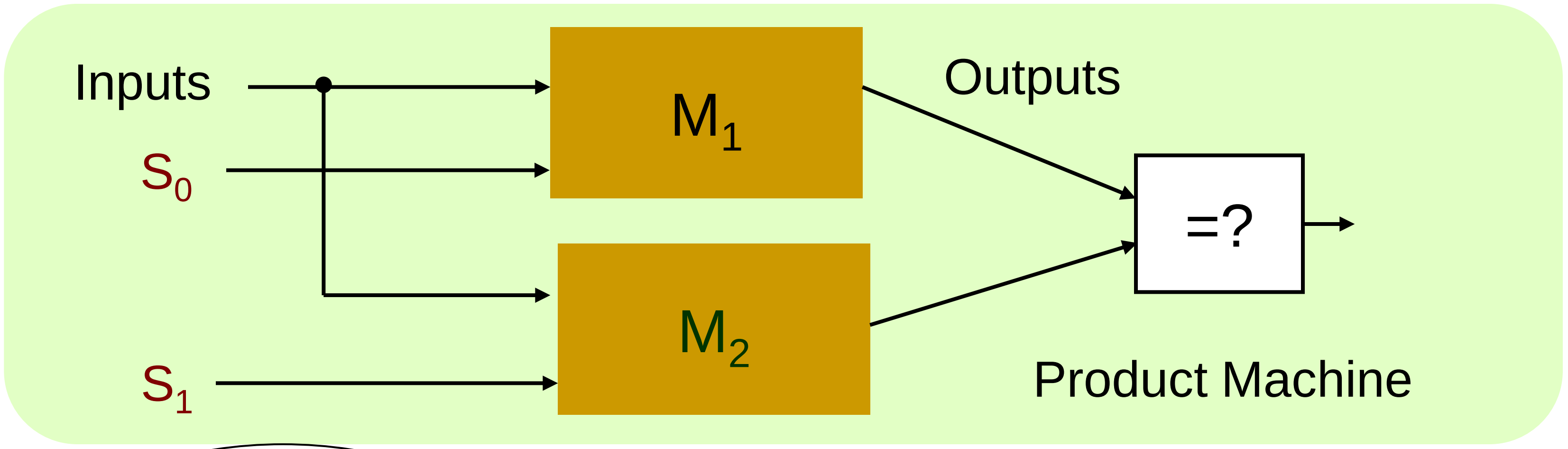
P3 = (AF) (BD) (CG) (E)

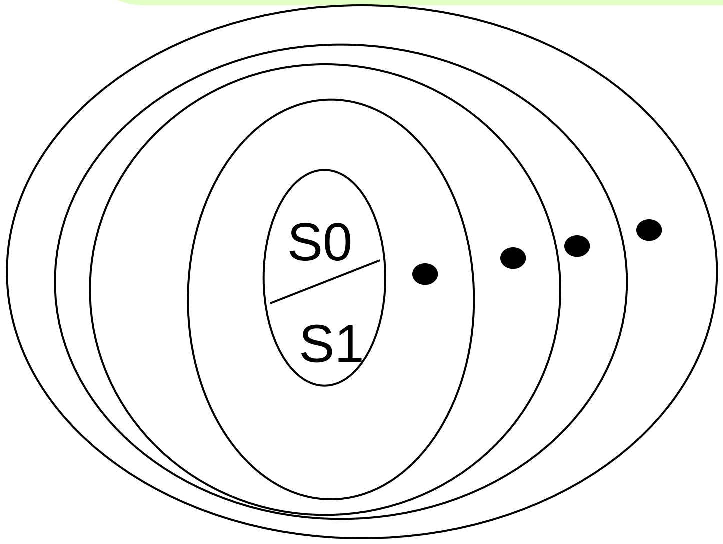
P4 = (A) (F) (BD) (CG) (E)

P5 = (A) (F) (BD) (CG) (E)

Reachability-Based Equivalence Checking

Approach 3: Symbolic Traversal Based Reachability Analysis



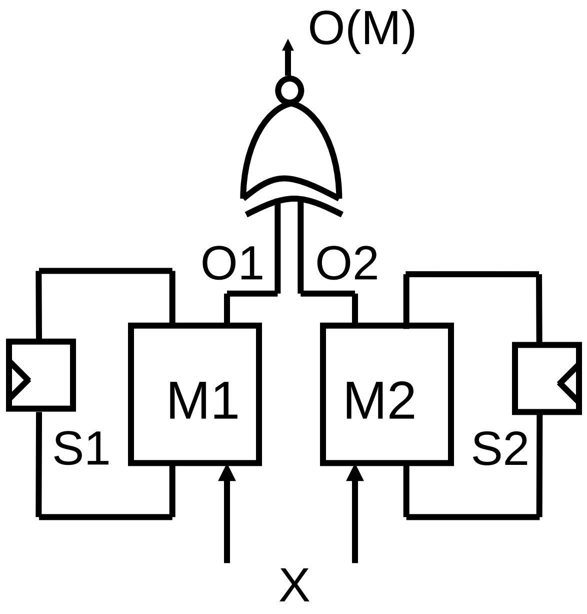


- Build product machine of M_1 and M_2
- Traverse state-space of product machine starting from reset states S_0, S_1
- Test equivalence of outputs in each state
- Can use any state-space traversal technique

Sequential Verification

- Symbolic FSM traversal of the product machine

- Given two FSMs: $M_1(X,S_1, \delta_1, \lambda_1,O_1)$, $M_2(X,S_2, \delta_2 , \lambda_2,O_2)$
- Create a product FSM: $M = M_1 \mathbf{x}\, M_2$
  - ➢ traverse the states of M and check its output for each transition
  - ➢ the output O(M) =1, if outputs $O_1= O_2$
  - ➢ if all outputs of M are 1, M_1 and M_2 are *equivalent*
  - ➢ otherwise, an *error state* is reached
  - ➢ *error trace* is produced to show: $M_1 \neq M_2$

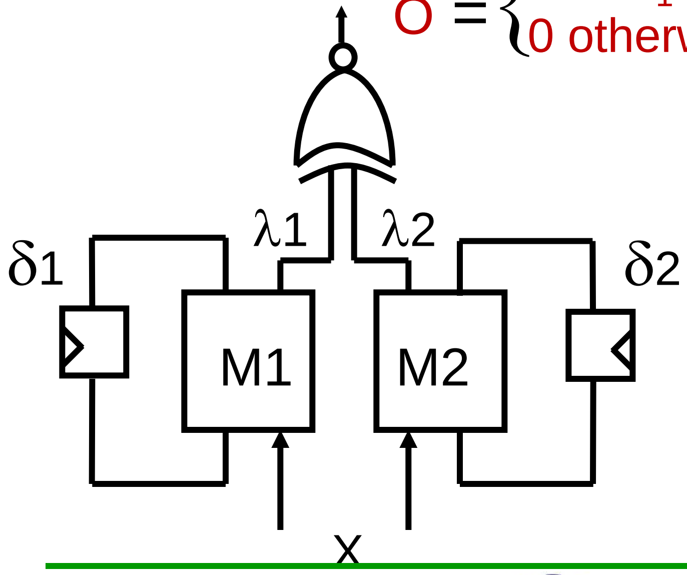


Product Machine - Construction

- Define the product machine $M(X, S, S^0, \delta, \lambda, O)$
  - states, $S = S_1 \times S_2$
  - next state function, $\delta(s, x) : (S_1 \times S_2) \times X \rightarrow (S_1 \times S_2)$
  - output function, $\lambda(s, x) : (S_1 \times S_2) \times X \rightarrow \{0,1\}$

$$O = \begin{cases} 1 & \text{if } O_1 = O_2 \\ 0 & \text{otherwise} \end{cases}$$

$$\boxed{\lambda(s,x) = \lambda_1(s_1,x) \overline{\oplus} \lambda_2(s_2,x)}$$



- Error trace (*distinguishing sequence*) that leads to an error state
  - sequence of inputs which produces 1 at the output of M
  - produces a state in M for which M1 and M2 give different outputs

FSM Traversal - Algorithm

- Traverse the product machine $M(X,S,\delta, \lambda,O)$
  - start at an initial state S_0
  - iteratively compute symbolic image $Img(S_0,R)$ (set of *next states*):

$$Img(S_0,R) = \exists_x \exists_s S_0(s) \bullet R(x,s,t)$$

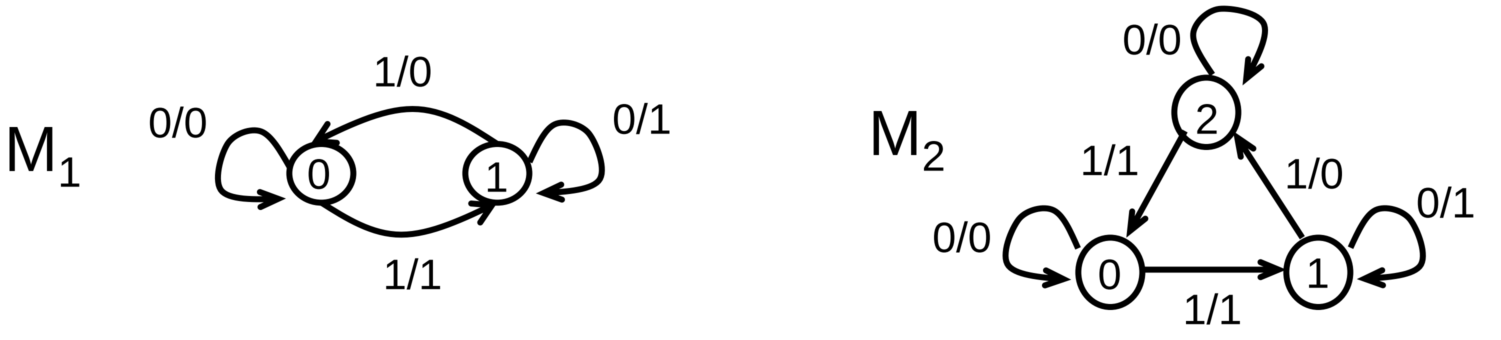
$$R = \prod_i R_i = \prod_i (t_i \equiv \delta_i(s,x))$$

    until an *error state* is reached

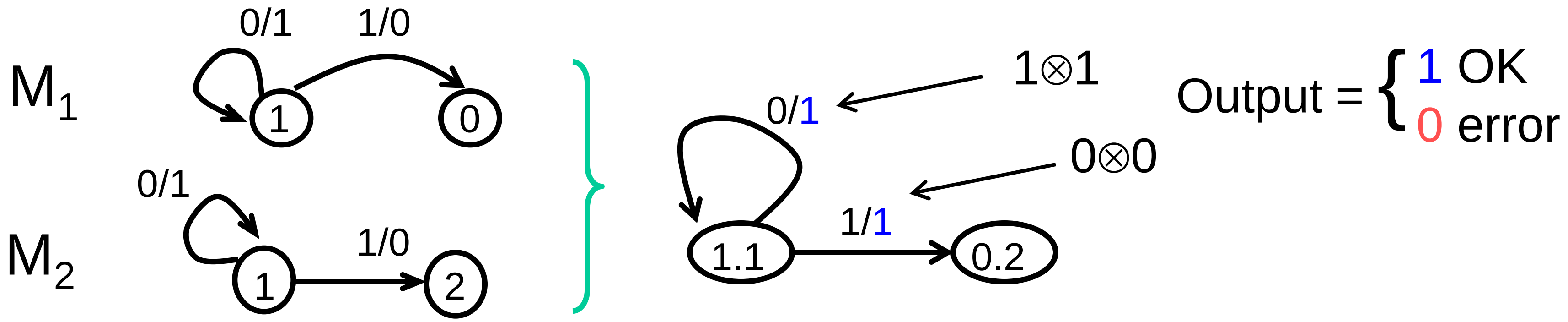
  - transition relation R_i for each next state variable t_i can be computed as $t_i = (t \otimes \delta(s,x))$

    (this is an alternative way to compute transition relation, when design is specified at gate level)

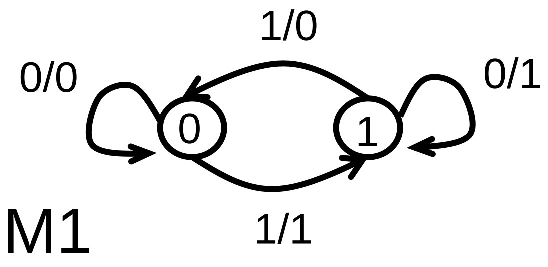
Construction of the Product FSM

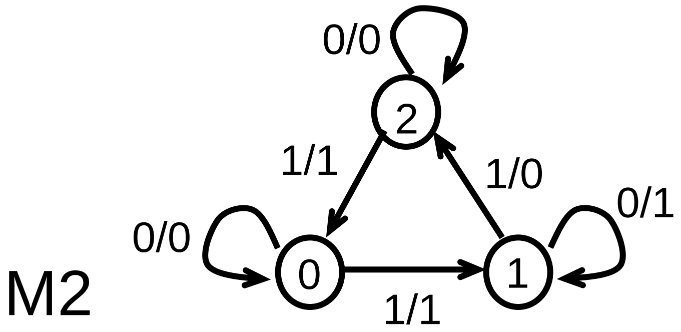


- For each pair of states, $s_1 \in M_1$, $s_2 \in M_2$
  - ➢ create a combined state $s = (s_1.\ s_2)$ of M
  - ➢ create transitions out of this state to other states of M
  - ➢ label the transitions (*input/output*) accordingly



FSM Traversal in Action

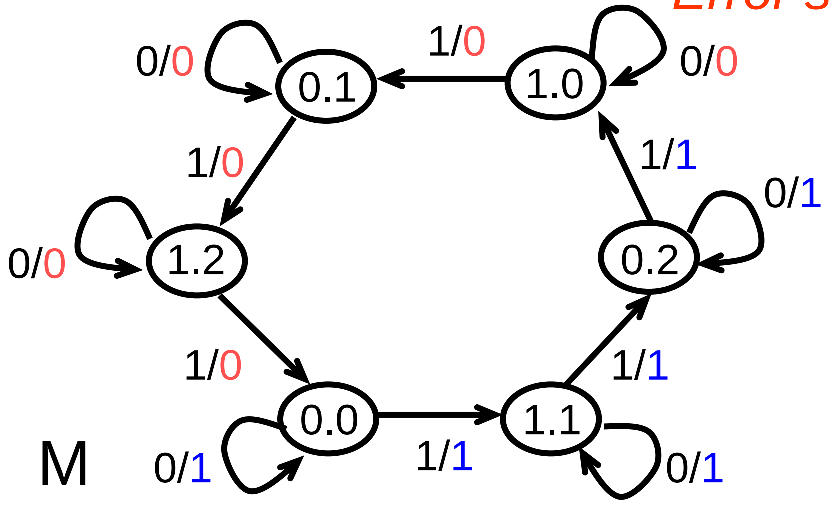




Initiall states: s_1=0, s_2=0, s=(0.0)

State reached	*Out(M)* *x=0*	*x=1*
• *New* ^{0} = (0.0)	1	1
• *New* ^{1} = (1.1)	1	1
• *New* ^{2} = (0.2)	1	1
• *New* ^{3} = (1.0)	0	0

Error state



- STOP - backtrack to initial state to get *error trace: x*={1,1,1,0}

Thank you