Synthesizing Asynchronous Burst-Mode Machines without the Fundamental-Mode Timing Assumption

Gennette Gill

Montek Singh

Univ. of North Carolina
Chapel Hill, NC, USA

Contribution

- Synthesize robust asynchronous controllers
  - ... from “burst-mode” finite-state machine specifications
- No timing assumptions between controller and env.
  - Reduces need for timing verification
- Improves modularity and facilitates design reuse

Outline

- Motivation
- Background and Previous Work
- Fundamental Mode: Challenges
- Our Solution
  - FSM Architecture
  - Synthesis steps
  - Example
  - Informal proof of correctness
- Results
- Conclusion

Motivation: Why Asynchronous Design

- No central clock – may eliminate problems inherent to synchronous design
- Several potential benefits:
  - higher speed, lower power, more modularity
- Our focus is improving modularity

Motivation: Modularity

- Replace modules locally without global implications
- Design reuse
- No timing analysis between module and env.
  - Decreased design and verification effort
- Facilitates automation; improves design time

Research Focus

Focus:

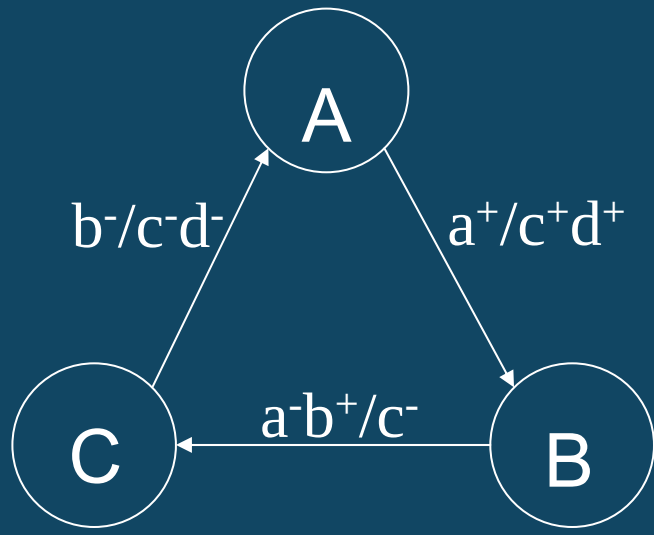
- Automated synthesis of asynchronous controllers

Objective:

- Minimize timing constraints between env and module
- Target is Quasi Delay-Insensitive (QDI) operation
  - Only timing assumption is “isochronic forks”
  - Equal wire delays along different fork branches

Burst Mode Finite State Machines

- A subset of Mealy finite state machines
- Consist of states connected by transitions
- Transitions consist of
  - An input burst (a set of input changes)
  - An output burst (a set of output changes)
- Values change in any order within bursts
  - Must be monotonic



Burst Mode Finite State Machines

Key properties:

- An output burst follows a *complete* input burst
- A new input burst follows a *complete* output burst
  - Restriction on the environment
- ***Maximal set property:*** No input burst can be a subset of another
  - Otherwise, specification is ambiguous
- ***Unique entry point:*** Each state can only be entered with a unique set of input values.
  - Facilitates hazard-free solution

Burst Mode: Previous Work

- Many existing methods
  - Minimalist [Fuhrer/Nowick et al. 1996-2001]
  - 3D [Yun et al. 1992]
  - ATACS [Meyers et al. 1999]
  - All operate using timing assumptions
- Minimalist = Basis for our work
  - Generates sum-of-products implementation
  - Optimal synthesis steps
  - Low Latency

Fundamental Mode

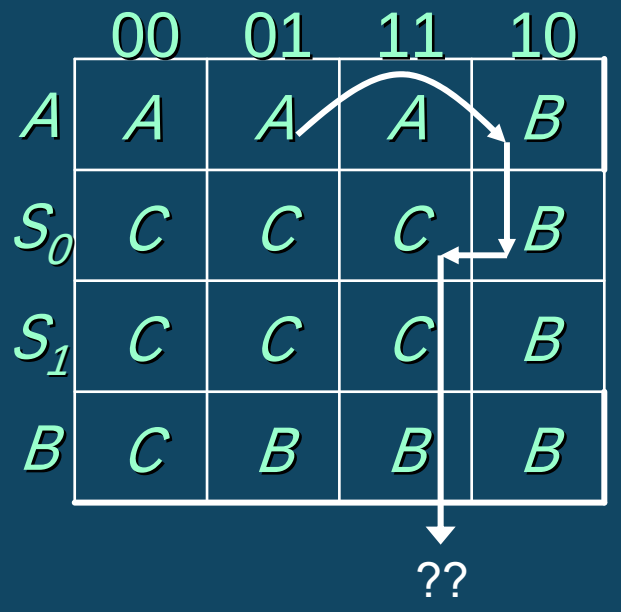
- The environment should not provide new inputs until after the machine has stabilized internally
- When placing a component in an environment
  - Perform timing analysis, or
  - Assume the environment is not too fast
- Timing analysis of every instance undermines modularity
- We break the timing assumption down into two separate challenges
  - State bits may not have changed when output changes
  - State bits may have changed, but not “stabilized”
  - In either case, environment may react too fast

Fundamental Mode: Challenge I

- Output produced before state changes
  - A new input arrives before the state changes
  - The machine is driven to an undesired state

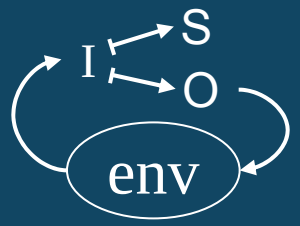
Example:

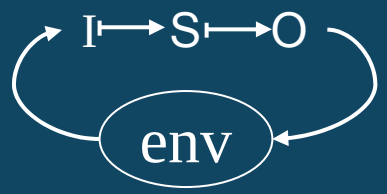
- A transition from state *A* to state *B* on input 10
- A new input arrives before all state bits have changed
- The machine is driven to state *C*

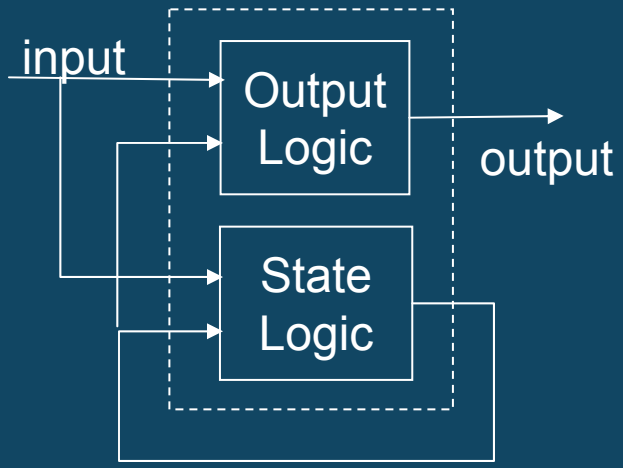


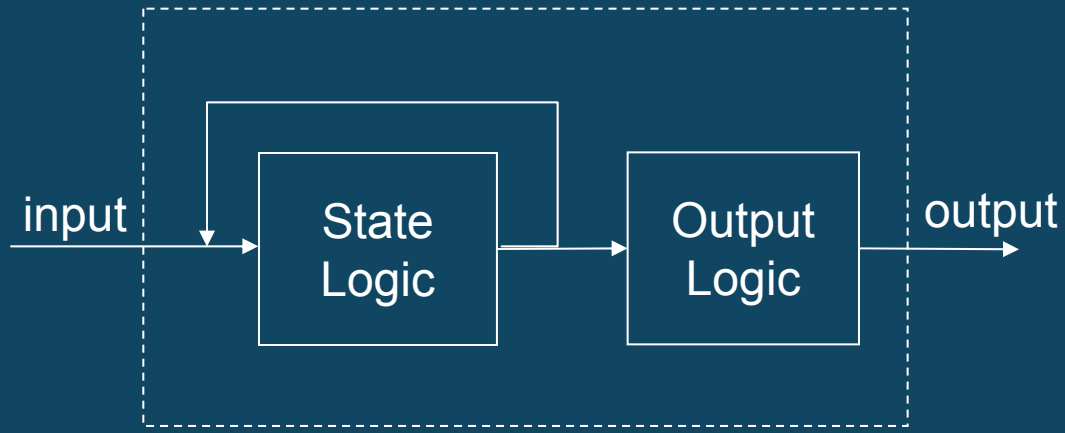
Challenge I: Solution

- Do not change output until state has changed







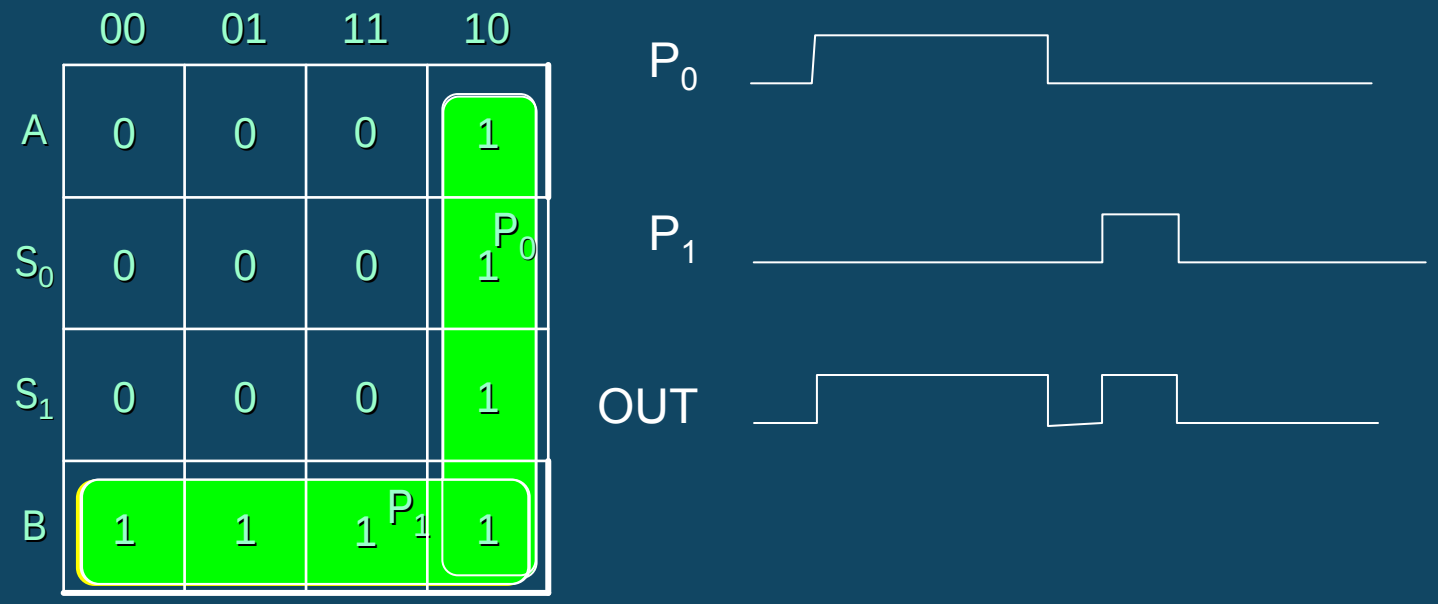


With fundamental mode
similar to a Mealy machine

Without fundamental mode
similar to a Moore machine

Fundamental mode: Challenge II

- A new input can arrive before all products that implement an output or state bit have "stabilized"
- "Stabilized" ≠ changed
  - Stabilized means there is no gate that is enabled but has yet to change
- Example: P_1 and P_2 are *unacknowledged internal paths*
  - internal changes that cause no external changes

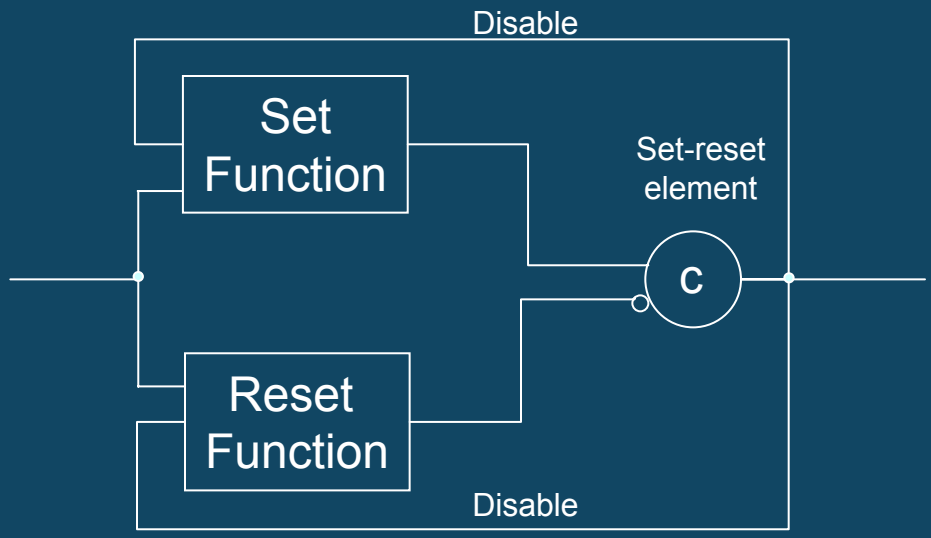


Challenge II: Solution

- Prevent more than one product from being enabled for a given function at any one time
  - Changed $\Rightarrow$ stabilized
- Two prevention methods:
  - If one product, P1 will assert before another product P2, use P1 to disable P2
  - If ordering cannot be determined, products should not overlap in Boolean space

Our system: target architecture

1. Set/reset implementation:
   - Each state and output bit has a set function and a reset function
   - Combine using a special C-element
2. Output is fed back to disable set/reset functions
   - Necessary to prevent unacknowledged internal paths

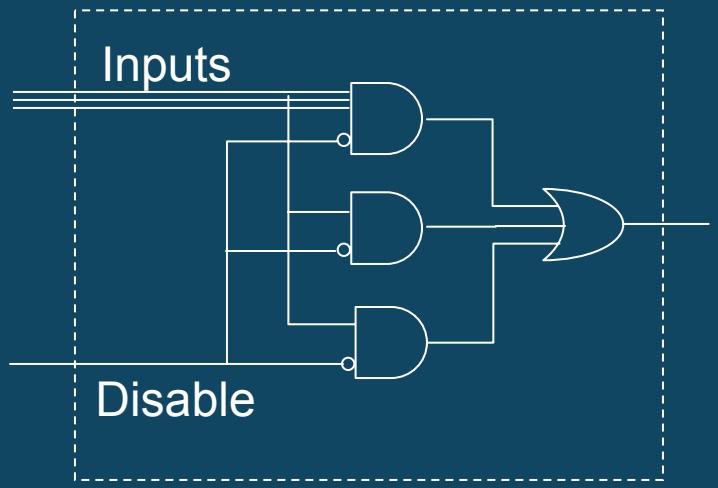


S	R	Out
0	0	Hold
0	1	0
1	0	1
1	1	Hold

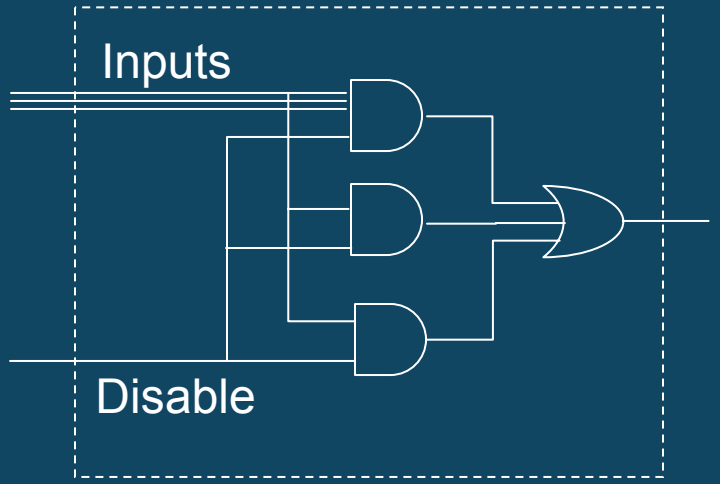
Target architecture: Set/reset functions

- Set and reset functions are both implemented as two-level sum-of-products
- Each function is disabled once it has caused output change
  - inverted disable bit for set function

Set function

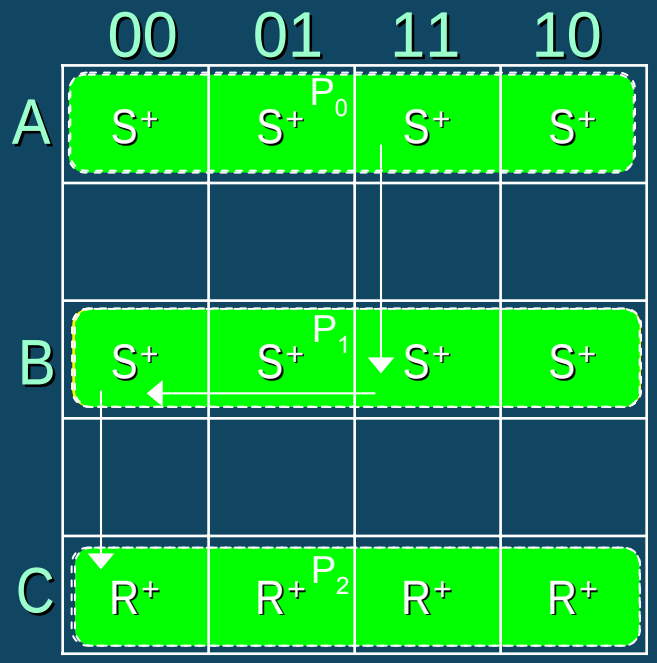


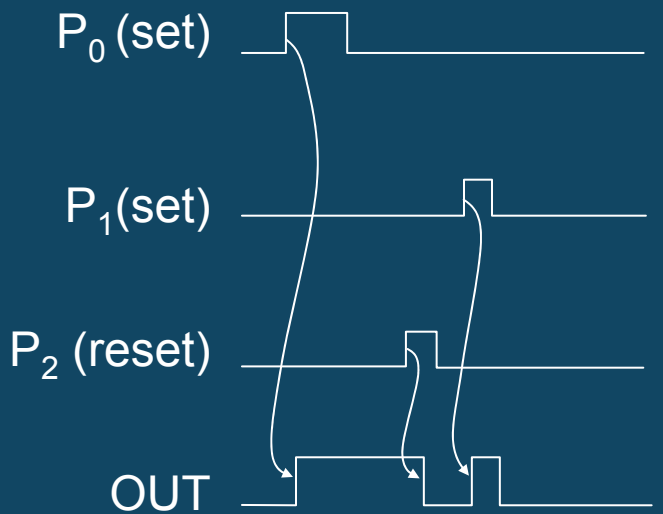
Reset function



Target architecture: Fed back outputs

- In this scenario, the fed-back disable prevents possible glitches



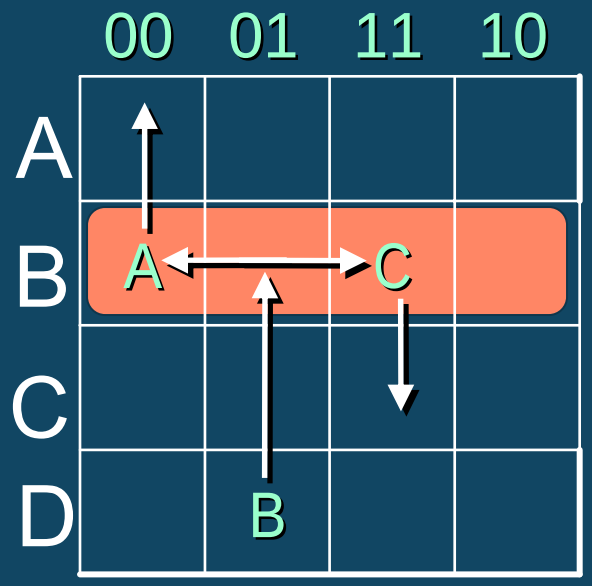


Synthesis Approach

- Two steps:
  - Constrained State Encoding
    - Critical-race-free constraints
    - Non-overlapping cube constraints
    - Guarantees existence of correct logic covering
  - Constrained Logic Covering
    - Hazard-free cover
    - Non-overlapping products
    - Imposes constraints on state encoding

Logic Covering

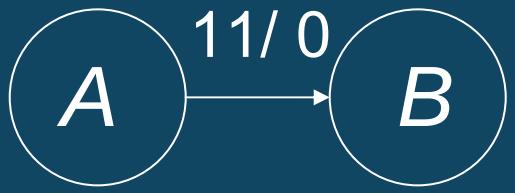
- We use a simple (non-optimized) logic covering step
  - Does not introduce product overlaps or hazards
  - Depends on the valid input region of a state

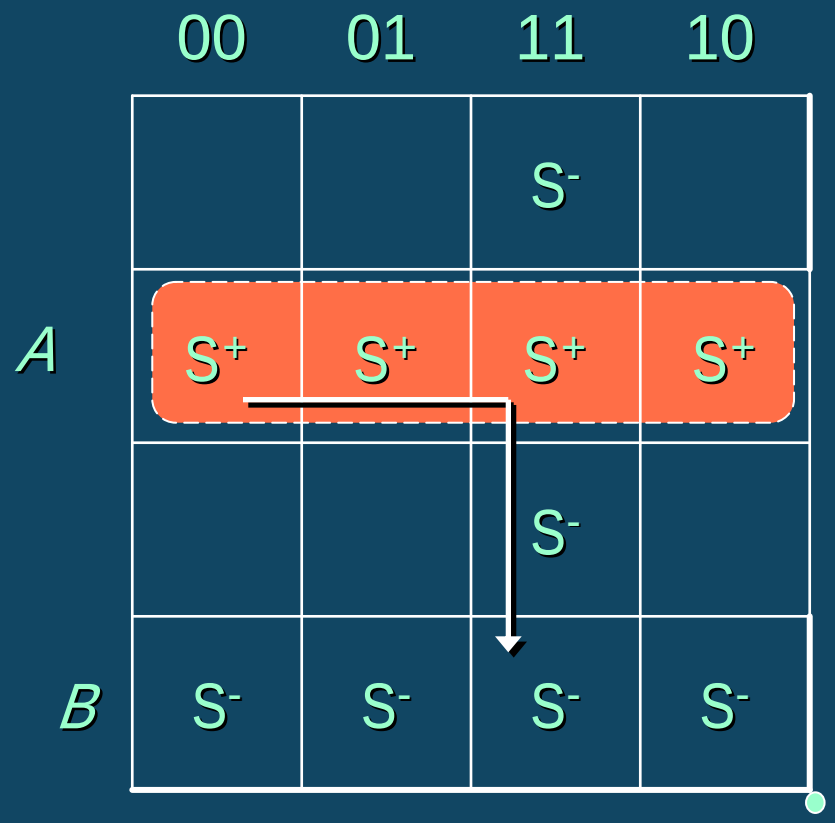


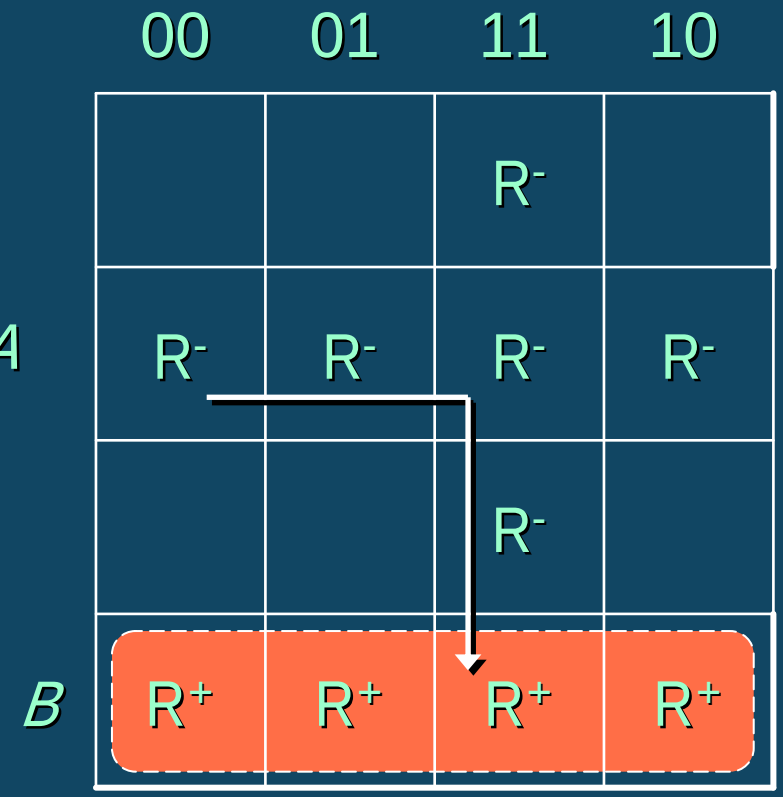
Definition: *valid input region*
Smallest cube that contains the entry point and all exit points of a state

Logic Covering: Output Logic

- Entry point of A is 00
- A transitions to B on input 11
- Output changes from 1 to 0

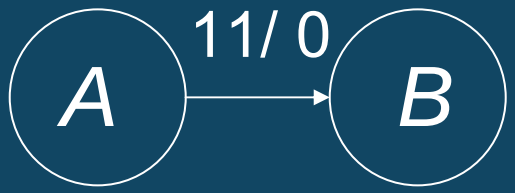






Logic Covering: State logic

- Entry point of *A* is 00
- *A* transitions to *B* on input 11
- output changes from 1 to 0



bit 1

	00	01	11	10
00			S+ R-	
A (01)			S+ R-	
11			S- R-	
B (10)	Sdc R-	Sdc R-	S- R-	Sdc R-

bit 2

	00	01	11	10
00			S- R-	
A (01)			S- R+	
11			S- R+	
B (10)	S- Rdc	S- Rdc	S- Rdc	S- R-

State Encoding

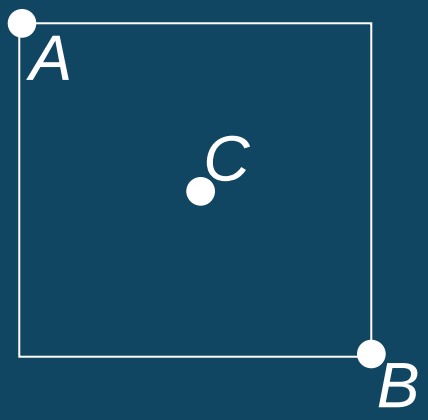
Constraints:

- Critical-race-free constraints
  - Protect a transition from: another transition or stable state
  - i.e., to ensure transition from A to B does not get diverted to some other state X
- Non-overlapping cube constraints
  - Protect a state's valid input region from other transitions
  - i.e., to ensure that a single product cover exists
  - Subsume critical-race-free constraints

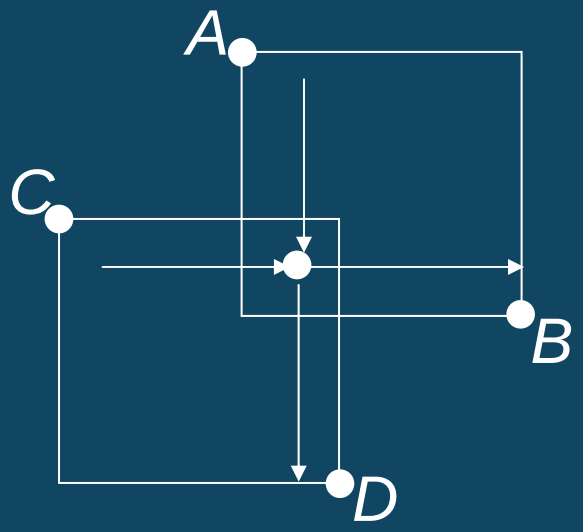
Types of Dichotomies

Constraints expressed as dichotomies:

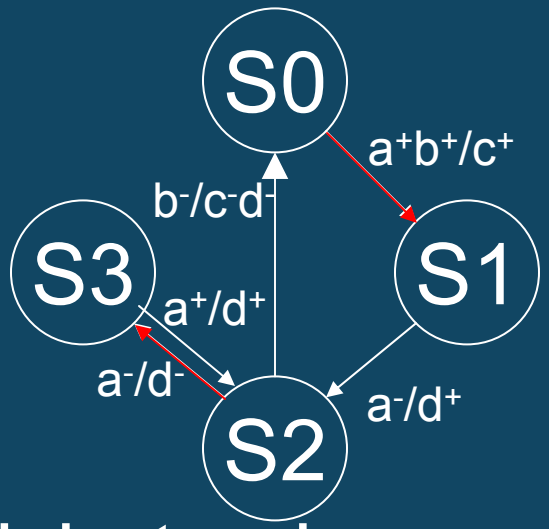
- A dichotomy (Sa; Sb) is a constraint between two sets of symbolic states that prohibits their respective smallest containing binary cubes from intersecting

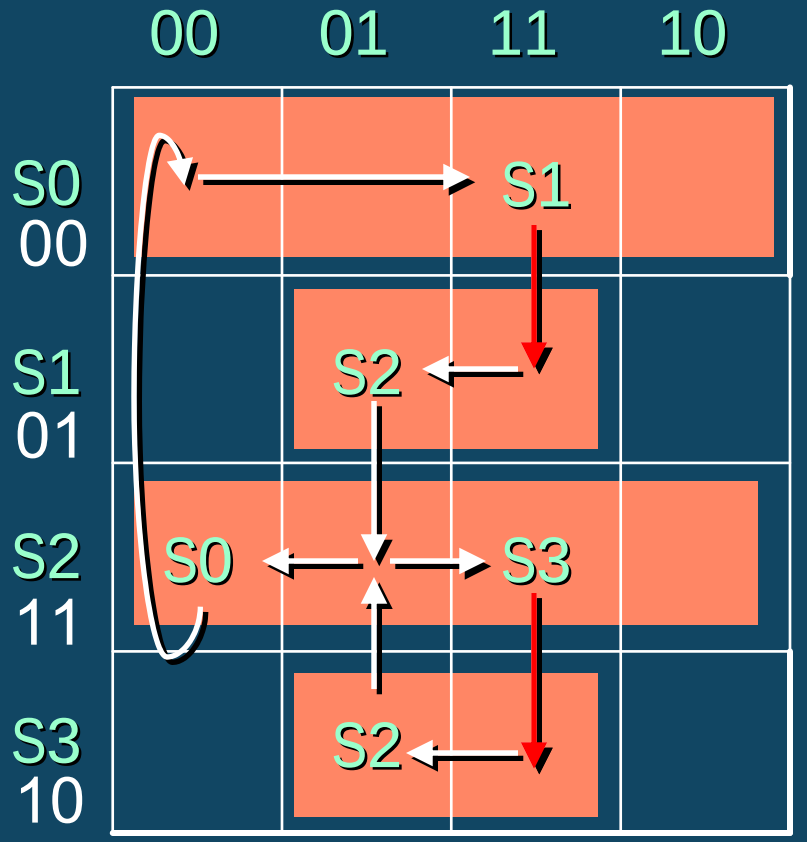


* Constraints of type (AB; C)
  - Prevent state C from being embedded in transition AB. Not necessary if C is don't care in the input column.
* Constraints of type (AB; CD)
  - Prevent transition AB from intersecting transition CD.



Example





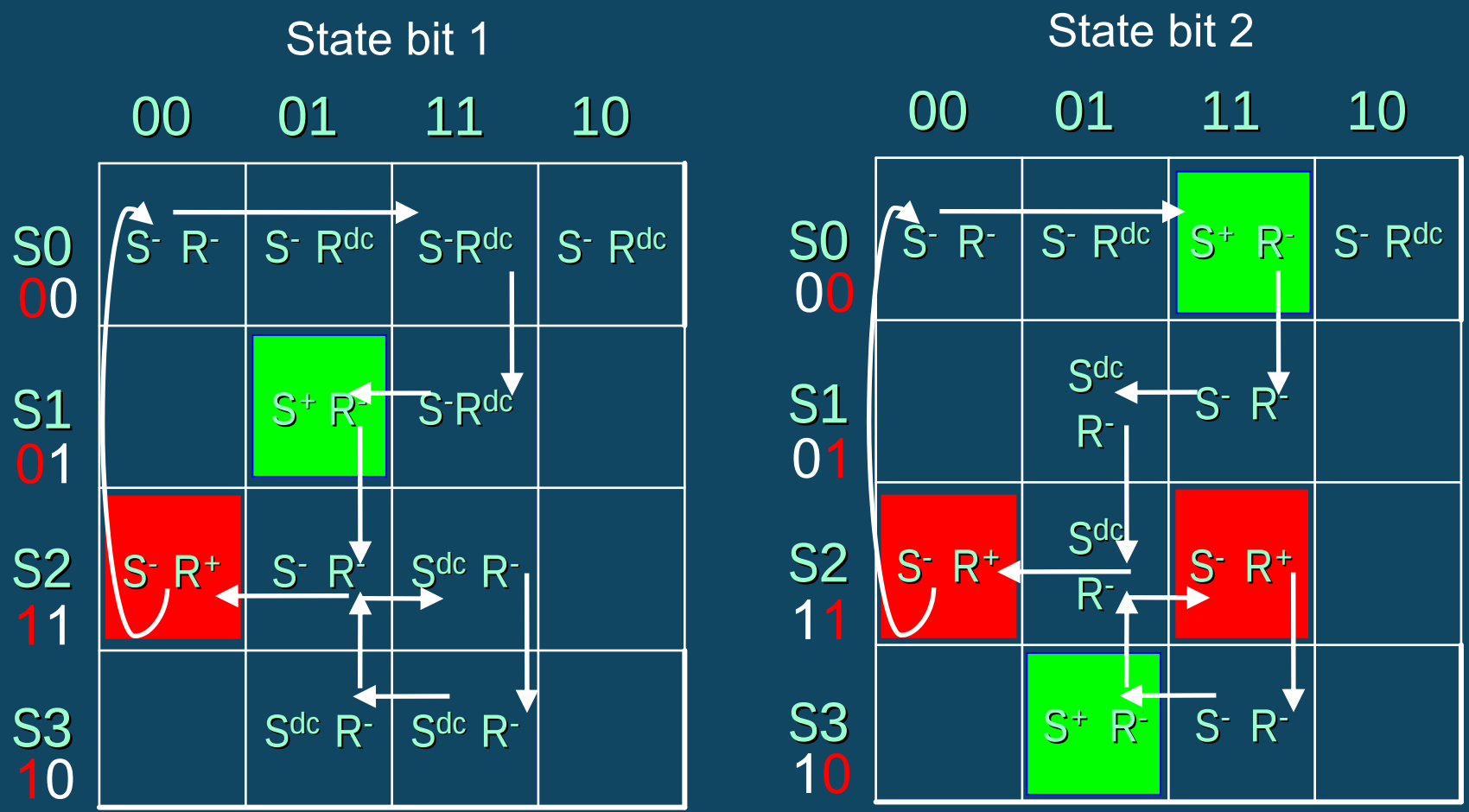
Dichotomies:

{0 1 2 ; 3} {0 1 3 ; 2} {0 2 3 ; 1} {1 2 3 ; 0}

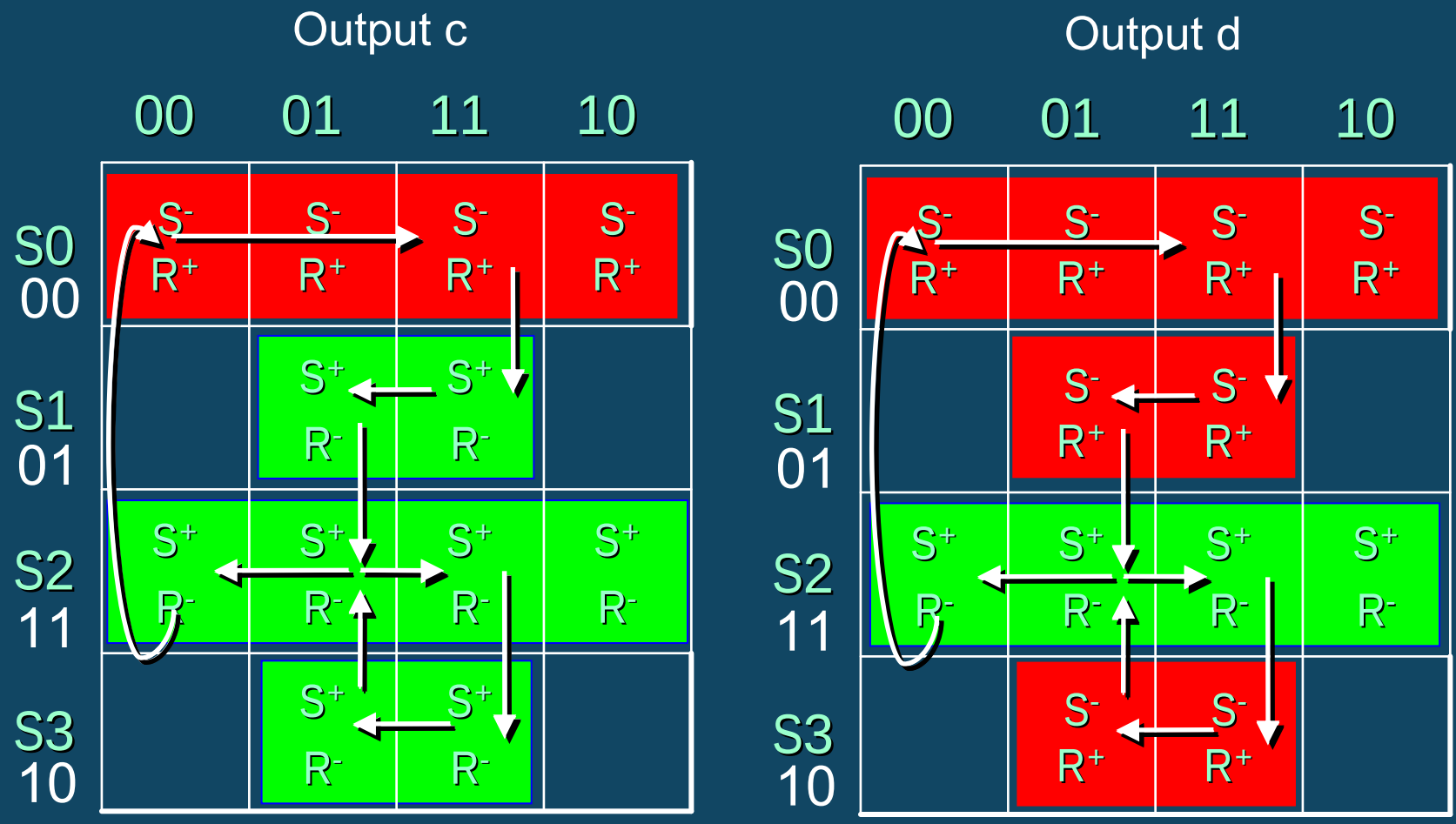
Intersections

{2 3 ; 0} {2 3 ; 1}

Example: State Cover



Example: Output Cover



Solution to Challenges

- Challenge I: Output change does not imply state change is complete
  - Solved using the Moore model
- Challenge II: Change does not imply stabilization
  - Allow only single product activation: change → stability
  - Logic covering approach = “single product cover”
    - Dichotomies to protect valid input regions
    - Fed back outputs to disable further product assertion
  - Net result: only one product is activated at any given time for a given state or output bit

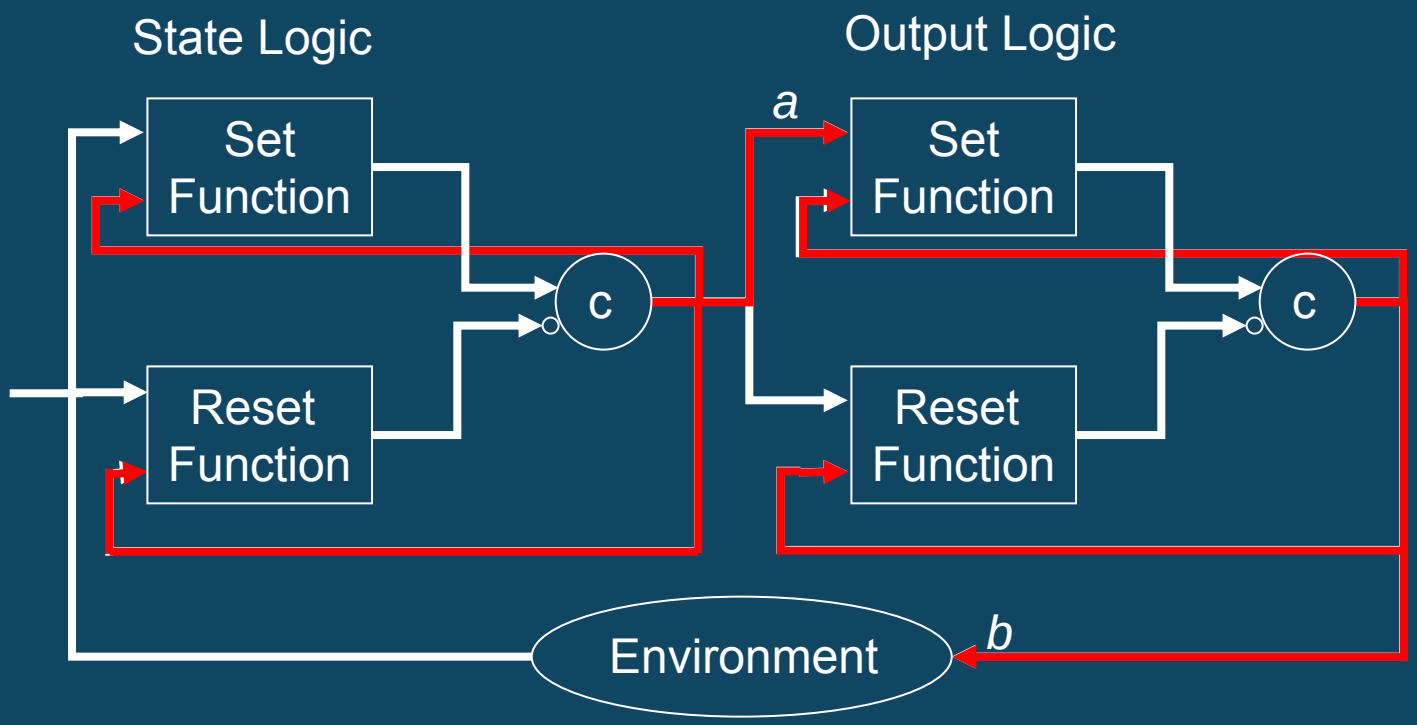
Informal Proof of Correctness

- Define: *Stable internal state*
  - At most one product is active for each output and state bit
  - All fed back disables have reached the products in the set and reset functions
- Assume: machine begins in a stable internal state
- Claim: machine will return to a new stable internal state before a new input arrives

Informal Proof of Correctness

- Proof:
  - State change visible at point *a* $\Rightarrow$ state logic is stable
  - Output change visible at point *b* $\Rightarrow$ output logic is stable

  $\Rightarrow$ FSM is stable when environment produces new input



Results

- Automated synthesis tool
  - Built off of Minimalist
  - Modified state encoding and logic covering
- Several burst-mode FSM benchmarks
  - Successfully able to handle all benchmarks
  - Some overhead w.r.t. Minimalist (as expected)
  - Not considered: state merging or fed-back output

Results

- Comparison: Our method vs. Minimalist
  - I/S/O = number of inputs, states, and outputs
  - #b = number of state bits
  - #c = product count
    - Product count does not include set/reset elements

Key Results:

- Our method has the same number of state bits in each case
- Product count:
  - Max overhead = 58%
  - In some cases, up to 16% improvement

Design	I/S/O	Us		Min-crf	
		#b	#c	#b	#c
Sbuf-read-ctl	3/7/3	3	19	3	12
Sbuf-send-ctl	3/8/3	3	19	3	19
Rf-control	6/12/4	4	35	4	24
It-control	5/10/7	4	30	4	28
Pe-send-ifc	5/11/3	4	33	4	28
Dram-ctrl	7/12/6	4	35	4	27
Pscsi-ircv	4/6/3	3	16	3	19

Conclusion

New method for synthesizing asynchronous finite-state machines

- Eliminate fundamental-mode timing assumptions
- Allow modularity and design reuse
- Modest product count overhead

Further work:

- Optimized logic covering: cube merging
- State merging: reduce total number of states
- Fed back outputs as state bits