Non-Deterministic Specification for Sequential Synthesis

Forrest Brewer, Andrew Seawright, Andrew Crews
forrest@ece.ucsb.edu, andy@synopsys.com, crews@corona.ece.ucsb.edu

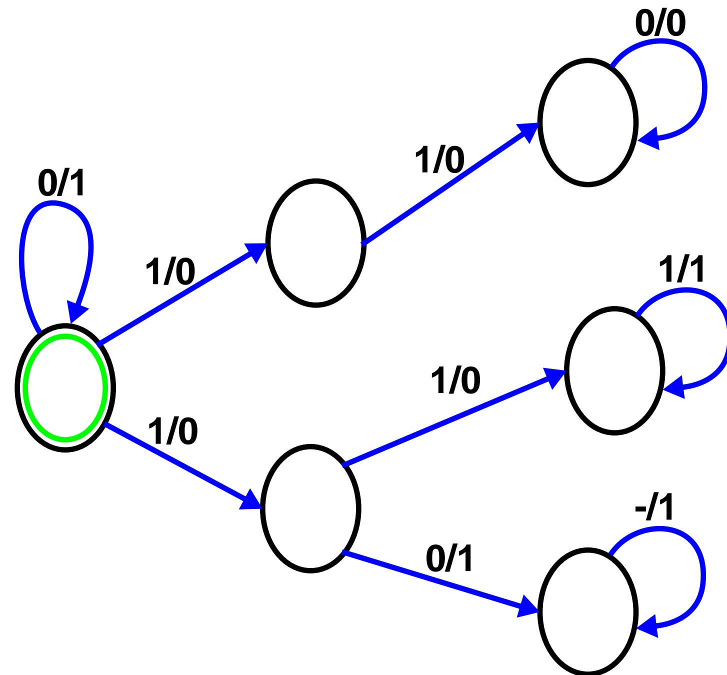
Department of Electrical and Computer Engineering
University of California, Santa Barbara

Why another sequential specification tool?

- **Deterministic Model encourages global state-based specification style**
  - Efficiency decreases rapidly with increasing numbers of states
  - Many properties of the machine make sense only for a subset of ‘bits’
  - Small changes in STG may lead to arbitrarily large changes in implementation
- **Designers are forced into partitioning the behavior into a set of managable DFSM’s**
  - No guarantee that chosen partition is globally efficient
  - Very easy to create specification problems such as safety, deadlock, etc.
  - Can be diffcult to obtain desired global behavior from local behaviors of submachines
  - Applicable tools force bottom-up design methodology
  - Engineering changes on global specification can be difficult to integrate in submachines
- **Current Tools support only very small DFSM’s**
  - State assignment optimized only for minimal (log), 1-hot or 2-hot, encodings
  - Tools limited by deterministic state complexity -- itself exponential in circuit size
  - Little correlation between STG and circuit implementation
  - Minimization of States/Latch # does not necessarily minimize design

Non-Deterministic Finite State Machines

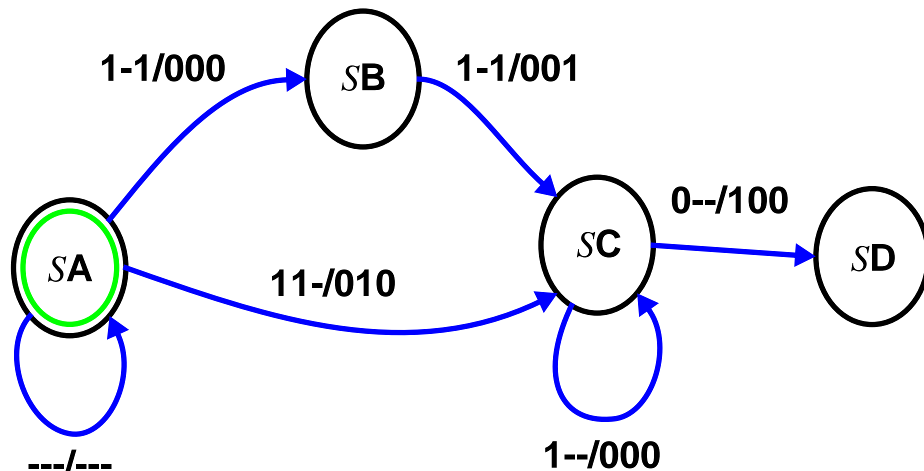
- Multiple tokens (control points)
- Can describe non-deterministic behavior
- Any DFA graph is also an NDFA graph
- Any NDFA graph can be rewritten to become a DFA, if it is deterministic
- Circuit model is effectively an NDFA-- local states only depend on logic fan-in tree
- Any synchronous circuit can be modeled by a comparably linear size NDFA



- **Use NDFA's to describe DFA behavior in a more powerful representation**
- **Can use NDFA to succinctly describe external (sequential) behavior constraints (ref. Brayton and Watanabe)**

An Example NDFA

- If a transition does not exist, the token is lost (destroyed)
- If multiple transitions for an input, tokens are created
- ‘1’ outputs dominate ‘0’ outputs



Sample Input Sequence

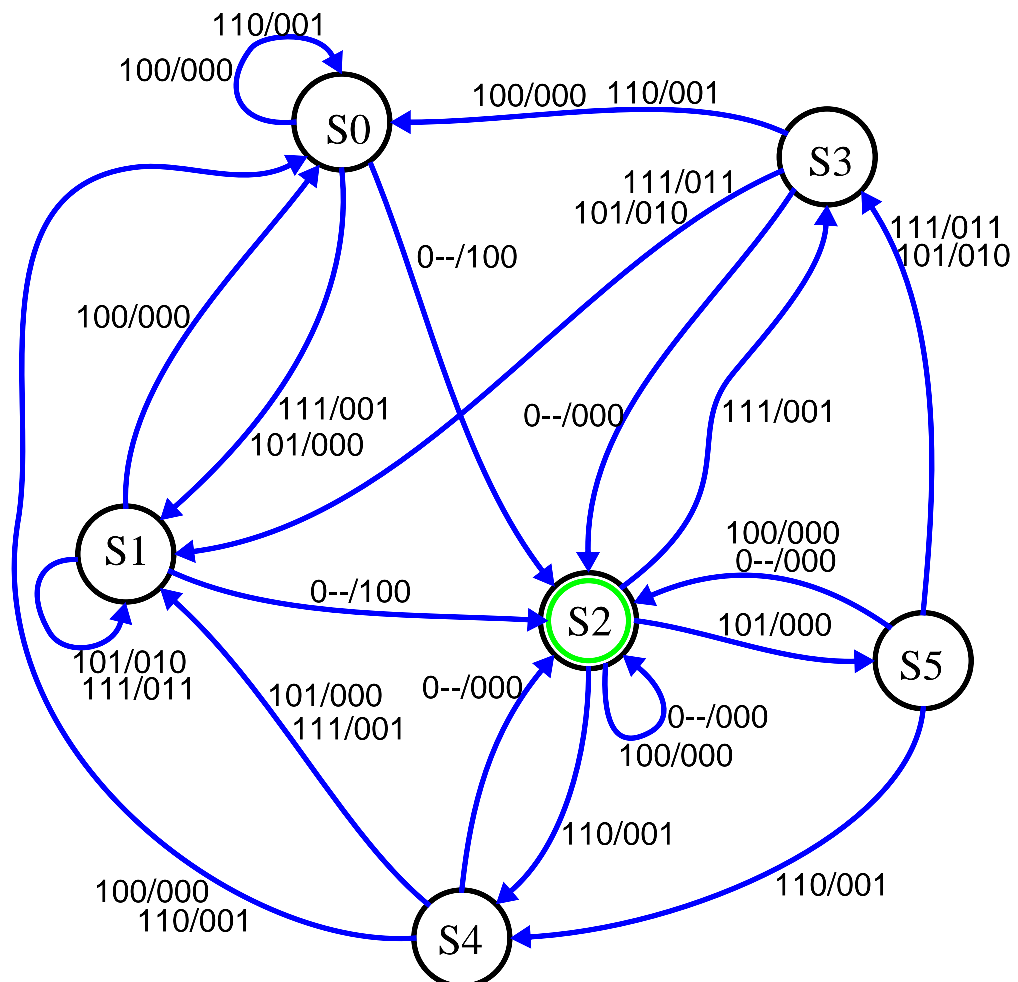
input	state(s)	output
start	A	
110	A,C	010
101	A,B,C	000
000	A,D	100
101	A,B,C	000
111	A,B,C	011
111	A,B,C	011

Eqivalent DFA Behavior

KISS Table

```
.i 3
.o 3
.p 32
.s 6
.r S2
0--  S0  S2  100
100  S0  S0  000
101  S0  S1  000
110  S0  S0  001
111  S0  S1  001
01-  S2  S2  000
111  S2  S3  001
110  S2  S4  001
001  S2  S2  000
101  S2  S5  000
-00  S2  S2  000
0--  S1  S2  100
100  S1  S0  000
101  S1  S1  010
110  S1  S0  001
111  S1  S1  011
0--  S3  S2  000
111  S3  S1  011
110  S3  S0  001
100  S3  S0  000
101  S3  S1  010
0--  S4  S2  000
111  S4  S1  001
110  S4  S0  001
101  S4  S1  000
100  S4  S0  000
01-  S5  S2  000
111  S5  S3  011
110  S5  S4  001
001  S5  S2  000
101  S5  S3  010
-00  S5  S2  000
```

State Transition Graph

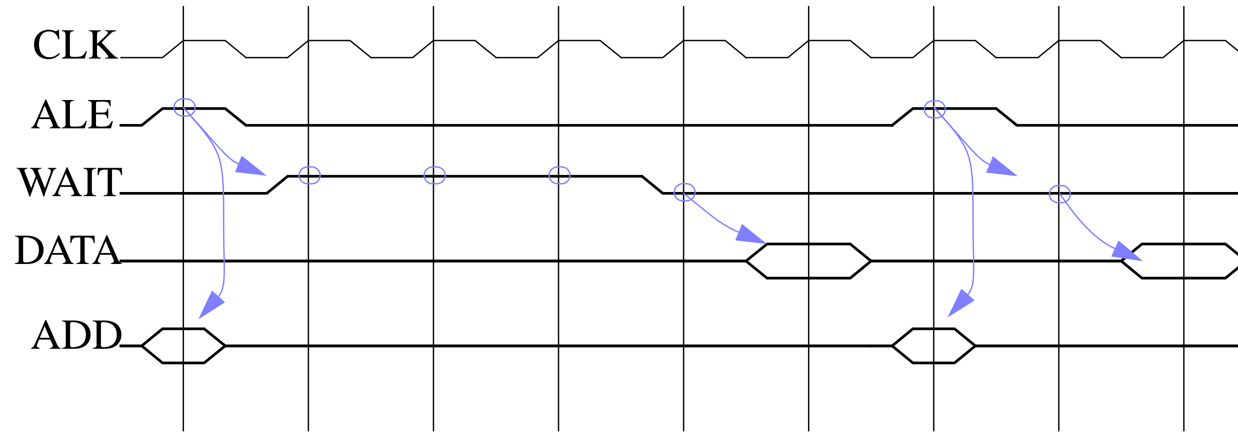


Machine Specification via Event Sequences

- **An alternative specification method is to describe those sequences of input stimula (events) which cause desired outputs (actions)**

  - a set of sequences defines DFA states implicitly

  - a timing diagram describes an event sequence, usually between 2 or more machines:



  - above, m1 outputs ALE and ADD and then looks at WAIT to see when DATA will be valid

  - m2 looks at ALE, latches the address, outputs WAIT until it is ready with the data

M1: [ALE][ADD], WAIT?, WAIT?, WAIT?, ~WAIT, Latch DATA

M2: ALE?[Latch ADD], [WAIT], [WAIT], [WAIT], [~WAIT], [DATA]

Machine Specification via Event Sequences II

- **Typically, a timing diagram represents an infinite set of such sequences**
  - no bound on the number of wait’s above...
- **a synchronous event sequence samples inputs and sets outputs on clock edges**
  - particular clocking scheme can be abstracted to “sampling” and “signaling” events
  - sampling and signalling events must be well ordered
  - logically dependent signals disambiguate timing via clock synchronization
- **a set of such sequences can be ‘open’ or ‘closed’**
  - a set is closed if any input sequence matches at least one event sequece
  - an implementation is necessarily closed, but not necessarily specified
- **a set of sequences can be “forbidden”**
  - a specification may contain sequences that are required, some that are don’t care, and some that must not occur
  - typically, there is design freedom implicit in how the don’t care sequences are implemented

Regular Expressions describe sets of event sequences

- RE's can represent *any* finite automata, and many possible event sequences

- **3 Classical Regular Expression Operators**
  - concatenation: "A,B" means "input A followed by input B"
  - closure: "A*" means "zero or more occurrences of A"
  - union: "A || B" means "either input A or input B"
- **A 4th "operator": the action.**
  - Creates a "complete" language allowing arbitrary symbolic machine description
  - However, the description is not *guaranteed* to be concise or complete

M1: (.[ALE][ADD], WAIT*, ~WAIT, .[Latch DATA])*

M2: (ALE[Latch ADD], ~Dready*[WAIT], Dready[DATA])*

- **Above pair of Regular Expressions describe *all* possible sequences implied by the previous timing diagram**

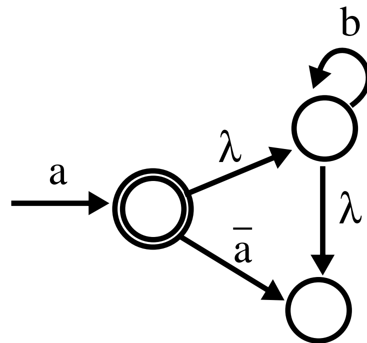
Why Regular Expressions?

- **Regular expressions directly map to NDFA's**
  - NDFA has one state per identifier in the RE
- **A Circuit can always be constructed directly from RE**
  - One memory element per NDFA state is always sufficient
- **So-- A simple Regular Expression => a Simple Circuit**

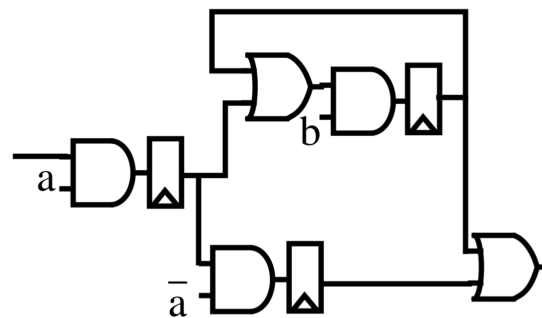
Regular Expression

a,(b* || ~a)

NDFA state diagram



Circuit



- **A valid specification may be 'open' -- allowing great freedom to optimize the closure or DFA**

Extended Regular Expressions

- **We can extend RE's by adding several operators which keep the language finite, but greatly improve the conciseness:**

- **Sequential AND: A&&B both A and B must accept at the same time**

- **Sequential NOT: !A -- accepts whenever A is not accepting**

- **Exception: A !E B -- B is enabled if A 'fails'**

  - a failure occurs if the input sequence matches no part of Regular Expression A

  - exceptions provide a succinct means of describing how to close a machine

  - although the complement of a DFA is not a state machine, the complement of a RE defined as all sequences which do not match the given set -- *is* a DFA!

  - consider a protocol decoder machine on a network: if a series of noise impulses arrives at the input, the machine may not recognize specified protocol. Exceptions provide a controlled means of transfering control in such cases.

  - Exceptions allow specification by all sequences *not* matching the given subset

- **Actions: A, B[start foo] -- provide a symbol to trigger on acceptance**

  - Actions provide a means to describe the required output on recognition

“Production Based Specification” (PBS)

- **An extended regular expression language**
  - Additional operators do not violate “1 memory element per symbol” construction
  - Further enhance the conciseness of machine specifications
- **Hierarchical description**
  - Allows reuse of named ‘productions’
  - Provides natural partitions which can be used for circuit optimization
  - productions from concise DAG of sequential operators
- **Boolean function terminals**
  - Use of BDD’s allows arbitrary Boolean functions to be atomic recognition points
  - Boolean terminals form the leaves of the production DAG’s
  - Sequential nodes from the branches
- **Actions allowed on any production acceptance**
  - actions can symbollically describe output behavior or any controlled activity
  - actions are implicitly ordered from most refined to least refined

PBS Implementation

- **Currently, PBS supports 2 design styles:**

  1. Controller Generator w. symbolic outputs (BLIF, ATT or other logic form)

  - Output is the controller only
  - Machine outputs correspond to action activation signals

  2. VHDL Code Generator to produce Synopsys® synthesizable VHDL

  - General system specification since actions are arbitrary (1-cycle) behaviors
  - Controller is PBS synthesized, VHDL combinatorial structure
  - Compiler supports VHDL code drop-ins for declarations, initialization
  - Compiler-Compiler format similar to YACC

- **Internal support for action conflict analysis**

  - Since actions are tied to separate event sequences, determining what actions are simultaneously possible is important

- **Intent is to provide an alternative specification/implementation strategy which is compatible with conventional design flows**

Action Details

- **Actions annotate productions to describe which contexts are externally observable**
- **Multiple actions may fire at the same time- if the same action appears in several places, acceptance of any context will fire the action**
- **Two actions, belonging to different levels of hierarchy of the *same* tree carry an implicit ordering-- lower levels execute first**

```
#inputs a b
#productions top p1 p2 p3 p4
::
top -> p1, p2*, p3; [a1]
p1 -> a+,~a; [a2]
p2 -> b [a3];
p3 -> (~b)[a2], p4;
p4 -> p1 && p2;
::
```

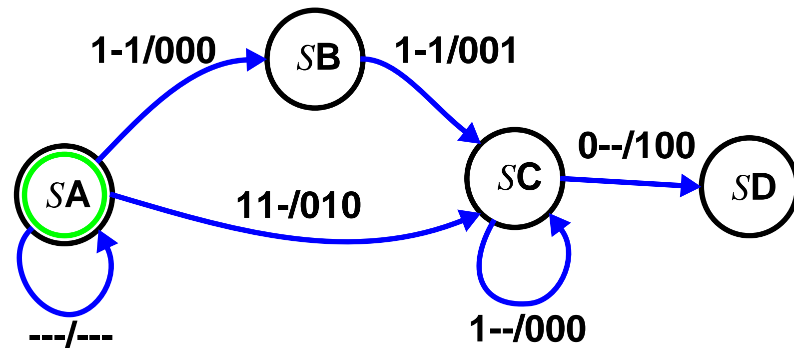
- Action a2 fires when either p1 or (~b) accepts
- Actions a1, a2 and a3 fire simultaneously when p3 accepts-- a2, a3 logically execute before a1, even though both are in same clock cycle
- although a2 and a3 can fire in the same cycle, they are logically unordered

“Production Based Specification” (PBS)

PBS Code Fragment

```
::
#input z0 z1 z2
#production p1 p2 p3 p4
::
p1 -> p2 || p3; {action0}
p2 -> .*, (z0 & z1) {action1}, p4;
p3 -> .*, (z0 & z2), (z0 & z2) {action2}, p4;
p4 -> z0+,~z0;
::
```

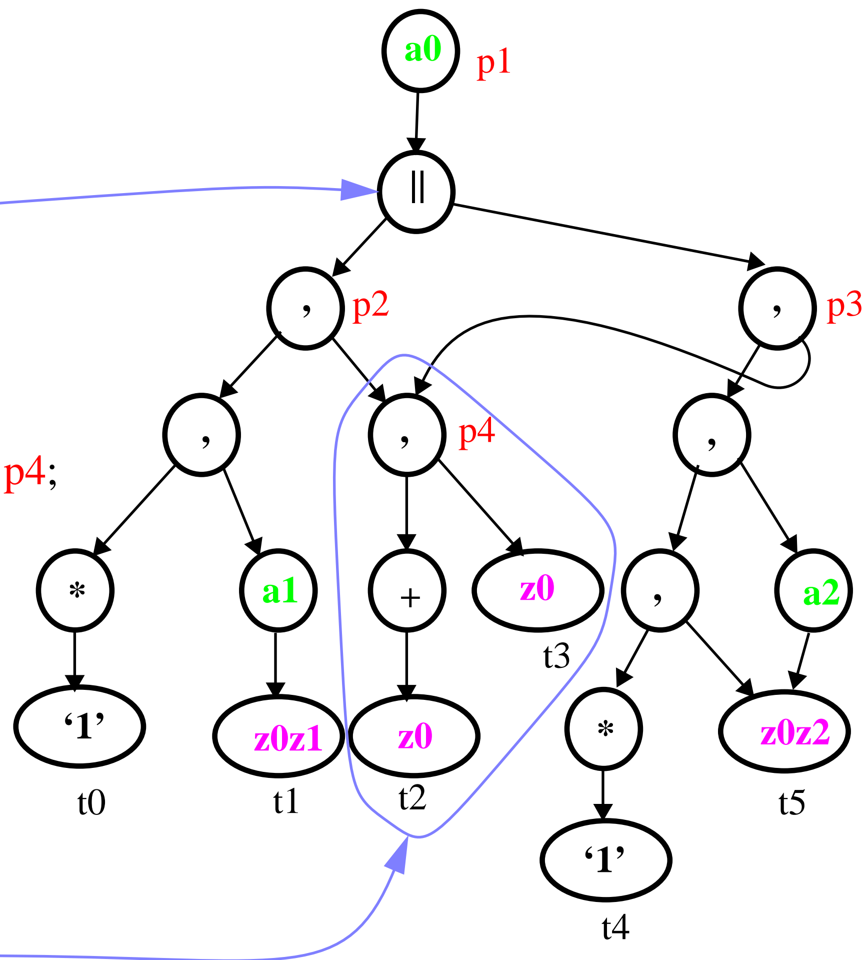
Corresponding NDFA



- **Code fragment represents a Regular Expression Tree that encodes all event sequences of the NDFA**

The *Production DAG*

```
::
#input z0 z1 z2
#production p1 p2 p3 p4
::
p1 -> p2 || p3; {action0}
p2 -> .*, (z0 & z1) {action1}, p4;
p3 -> .*, (z0 & z2), (z0 & z2) {action2}, p4;
p4 -> z0+,~z0;
::
```



Simplification of the Regular Expression

- **The PBS is parsed to form a Production DAG and then to a RE tree**

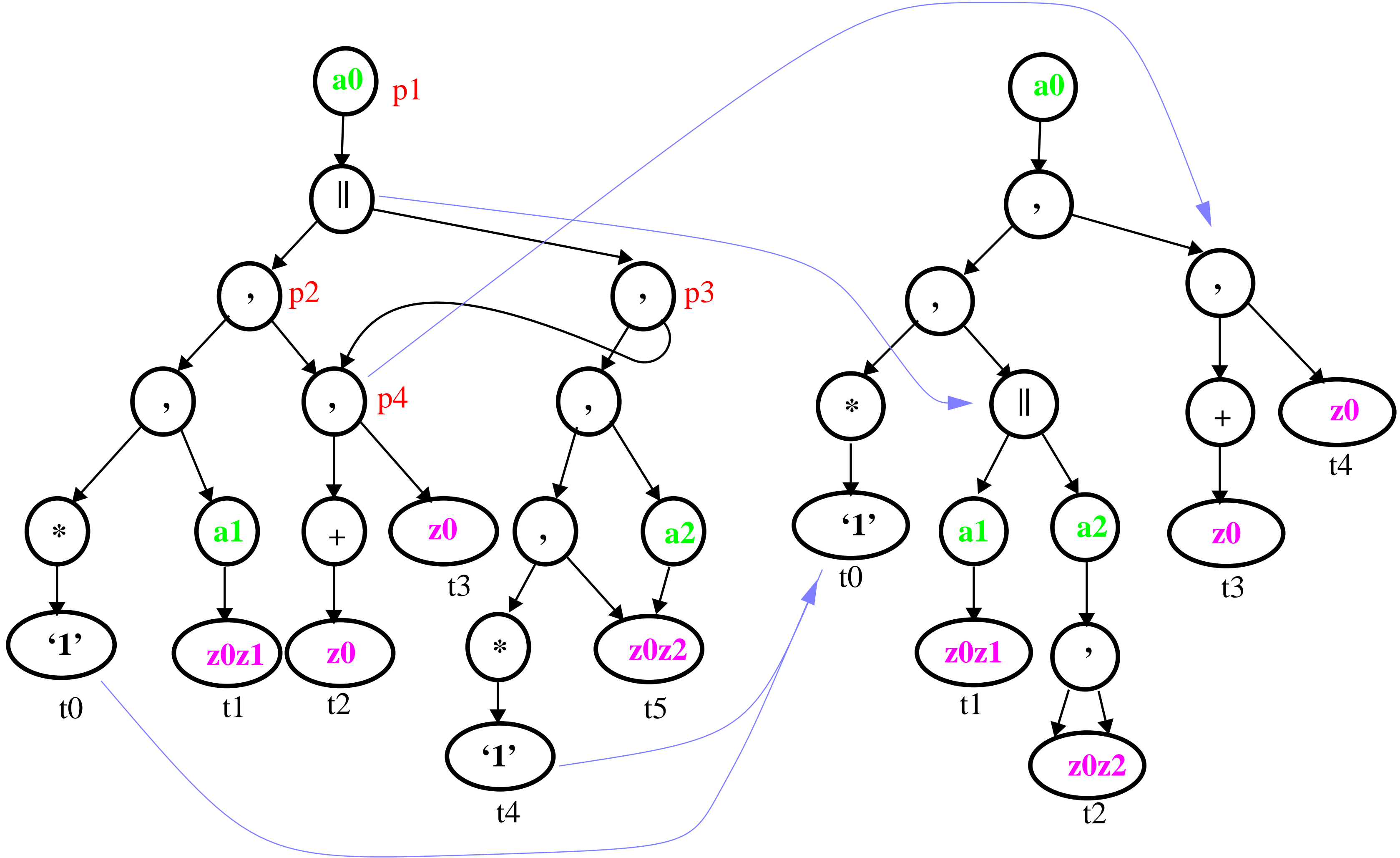
- **Several minimization rules reduce the number of terminals:**

  Eg:
  (A,B) || (A,C) => A, (B || C)
  (A,B) || (C,B) => (A || C), B
  A || A => A
  (A*)* => A*
  A,A* => A*,A => A+

- **Build "Unique Table" for production node sub-trees**

  - simplifies efficient recognition of equivalent sub-trees
  - allows rule set application by ordered pattern matching hashed node list
  - terminal reduction is directly reflected in latch elimination

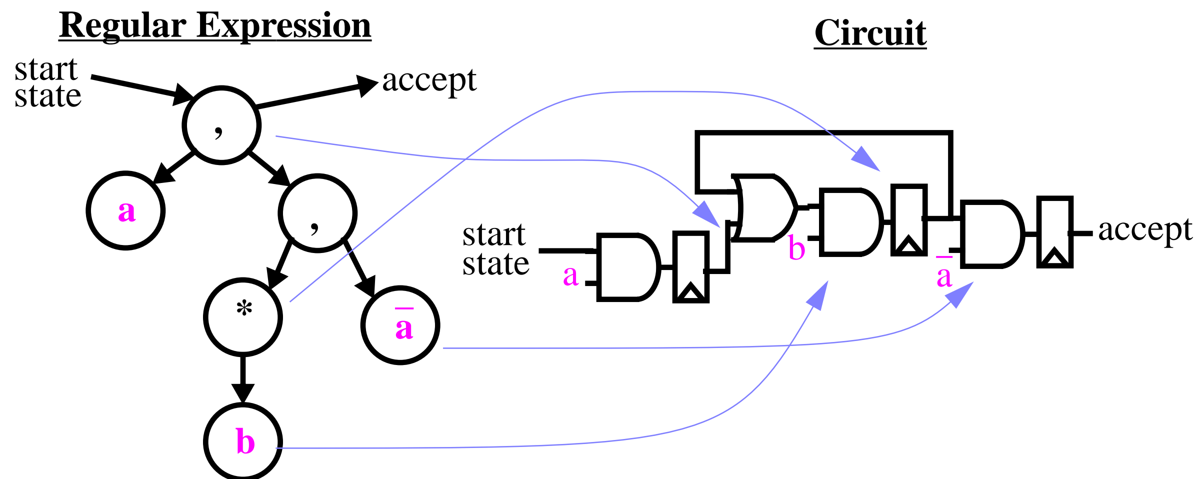
Reduced Production Tree



- Reduce the number of terminal nodes

Machine Construction

- **Construction is *linear time* traversal of simplified RE tree**
- **Each RE node has a substitution template which is passed the current context and which returns the resulting context**
- **The Start or Reset context is passed to the initial node of the traversal**



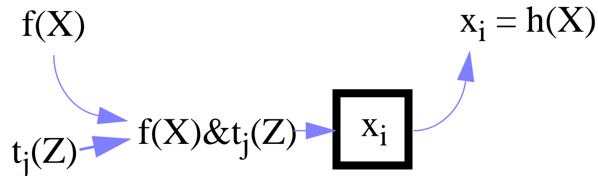
Build Function

- **Build is the recursive traversal construction algorithm**
- **Its time complexity is pseudo-linear, including the logic manipulations**

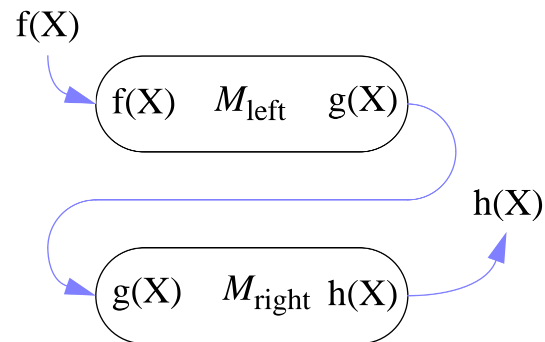
```
Build (node: *n, Boolean function: f(X)) {
    if (n is a terminal function t_j(Z)) {
        create new control point x_i;
        set h(X) = f_i(X, Z) = y_i = f(X)∧t_j(Z);
    } else if (n is concatenation node) {
        g(X) = Build(node->left, f(X));
        h(X) = Build(node->right, g(X));
    } else if (n is sequential and node) {
        g(X) = Build(n->left, f(X));
        h(X) = Build(n->right, f(X));
        h(X) = g(X)∧h(X);
    } else if (...) {
        ... other cases ...
    }
    if (action a_k attached to node) {
        set c_k(X) = h(X);
    }
    return h(X);
}
```

Node Templates

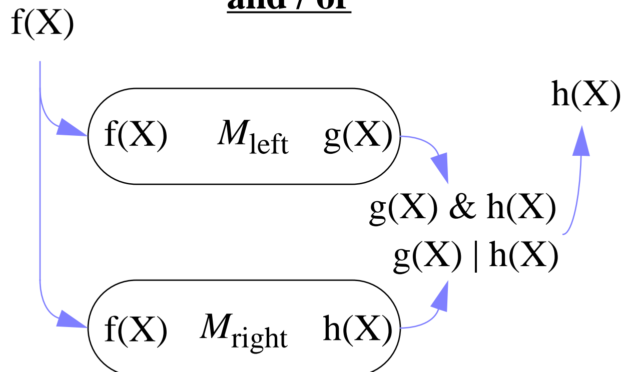
terminal function



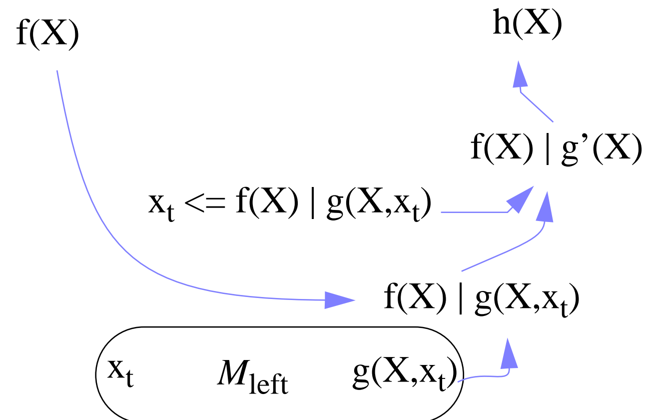
concatination



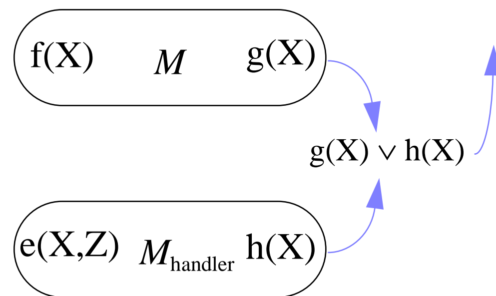
and / or



Kleene Closure



Exception Construction

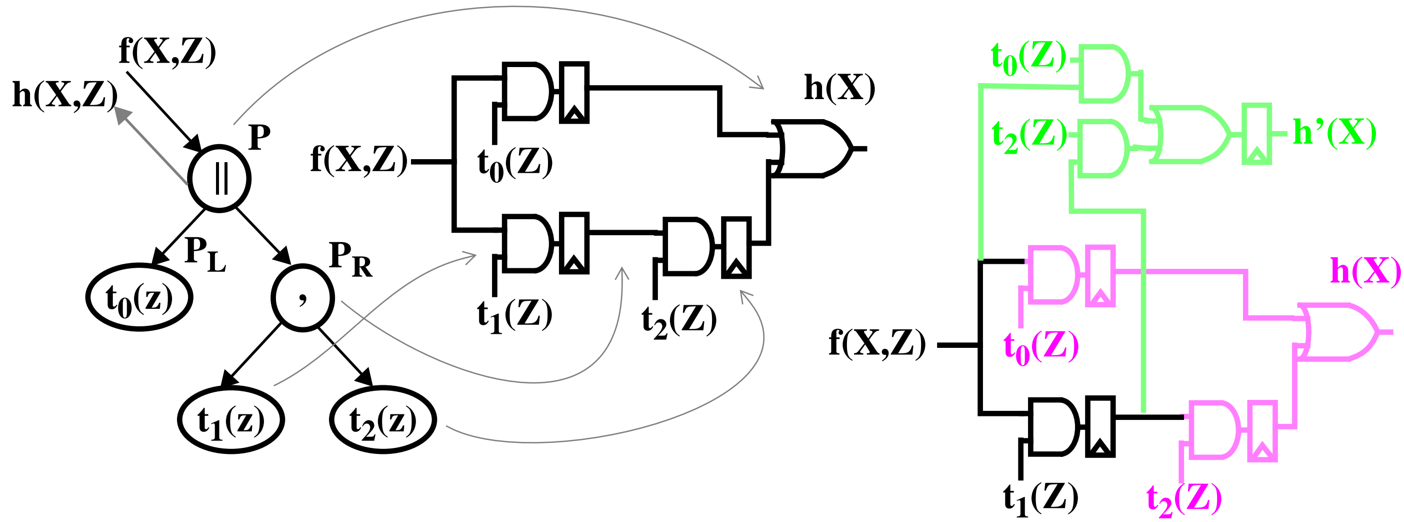


- **Exception machines are constructed simultaneously with the traversal construction of Build. Each template also contains an exception sub-machine and a "working" context construction.**

As the traversal progresses, both the valid and complement machines are built, if an exception is needed, the complement machine context is used.

Merging Equivalent States

- **During Construction, identify and merge some equivalent NDFA states**
  - since only a subset of control points have actions, it is possible that many states are output indistinguishable
  - could maintain a list of current DFA states-- re-introducing state-space explosion
- **However, we can often eliminate redundant states without global analysis!**
  - in the figure below, at context h(X), we can not tell if the context was activated by t0 acceptance, t2 acceptance or both-- since there are no actions on these contexts, we don’t care which. A redundant context h’(X) is created and other logic may become redundant and be removed.



Merging Equivalent States

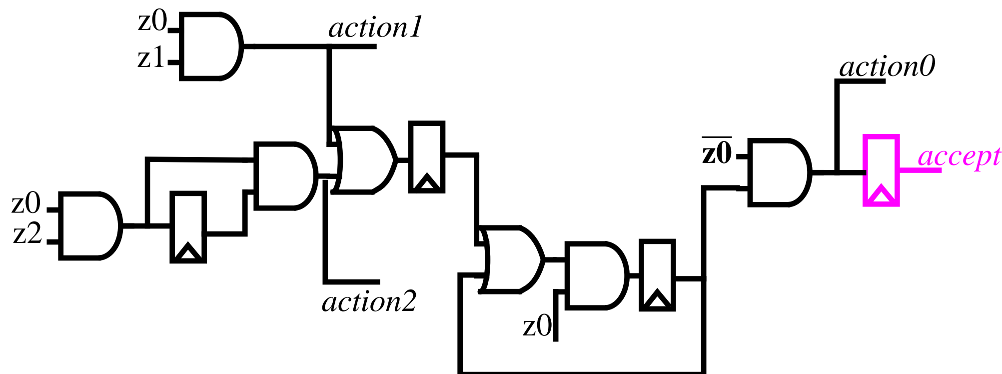
- **Create a redundant state bit:**
  - For any current context function, $h(x_1,x_2,...)$, we create a new state bit, x_i as follows:

    $\delta_i(x_1,x_2,...) = h(\delta_1,\delta_2,...)$ where each δ_j corresponds to state bit x_i, by retiming each x_i.

  - This state bit is redundant, but may allow simplification of the acceptance function:

    $h'(X) = x_i$
  - More importantly, all such contexts are added to the context hash table, where any later bit construction can make use of them
- **Context to be removed must not carry an action (observability point)**
- **Logic commonly eliminated at:**
  - Closure and '+' nodes
  - Sequential function nodes (and, or, not)
  - Replicated sub-machines if context allows

Constructed Machine



	Literals	Depth	Registers
PBS	***13***	***3***	***3***
NOVA	**13(16)**	**3(4)**	**3**
JEDI	**22**	**9**	**3**
one-hot	**38**	**9**	**6**

Encoding Results

design example	Current			STG states	Clairvoyant			min STG states	NOVA			JEDI			one hot		
	L	D	R		L	D	R		L	D	R	L	D	R	L	D	R
ex1 (Fig. 1)	**13**	**5**	**3**	8	22	5	8	6	23	11	3	22	9	3	38	9	6
mouse	**10**	**3**	**2**	4	18	5	4	3	10	3	2	10	3	2	10	3	3
xymouse	**20**	**3**	**4**	10	36	5	8	9	42	11	4	35	6	4	59	4	9
mouse2	**24**	**4**	**6**	8	36	5	10	7	36	14	3*	28	12	3*	41	8	7
xymouse2	**48**	**4**	**12**	50	72	5	20	49	557	67	6*	592	75	6*	618	18	49
count0	**8**	**5**	**3**	5	16	5	4	4	8*	4*	2*	11	6	2*	12	6	4
qr42	**42**	**6**	**13**	33	77	6	21	16	64	17	4*	82	24	4*	121	12	16
i8251ar	**67**	**8**	**13**	15	84	16	14	15	84	10	4*	84	10	4*	81	9	15
i2c	**115**	**8**	**24**	70	156	10	37	51	323	36	6*	273	46	6*	427	18	51
midi	**223**	**16***	**107**	107	604	22	166	104	447	56	7*	420	72	7*	262	14*	104
match1	**10**	**3**	**3**	5	12	4	5	3	7*	4	2*	8*	3*	2*	9*	5	2*
match2	**21**	**3**	**7**	15	26	4	12	11	38	10	4*	43	12	4*	61	12	11
match4	**43**	**4**	**16**	111	56	4	28	79	333	58	7*	370	73	7*	468	18	79
match8	**87**	**4**	**36**	6151	116	6	63	unable									
match16	**175**	**5**	**80**	~10^7	236	6	138	unable									
match32	**351**	**5**	**176**	~10^{14}	476	6	297	unable									

Example: Interlocked Pipeline

- **Idea: a pipeline in which each stage checks the next stage to see if its context is not stalled**

- **Want description to only refer to next stage, if possible**

- **Want ability to interlock (stall some stage) on arbitrary event**

- **Requires new operator: *marker***

- **A *marker* is a special action which provides a Boolean output that can be tested to determine the current or previous state of other productions**

    - allows a production to test the acceptance of any set of simultaneously active productions
    - provides concise non-local access to context of other productions
    - can make description far simpler and more intuitive, but can be overused since:

- **Markers allow arbitrary state transition description as well as regular expression format specification and mixed modes.**

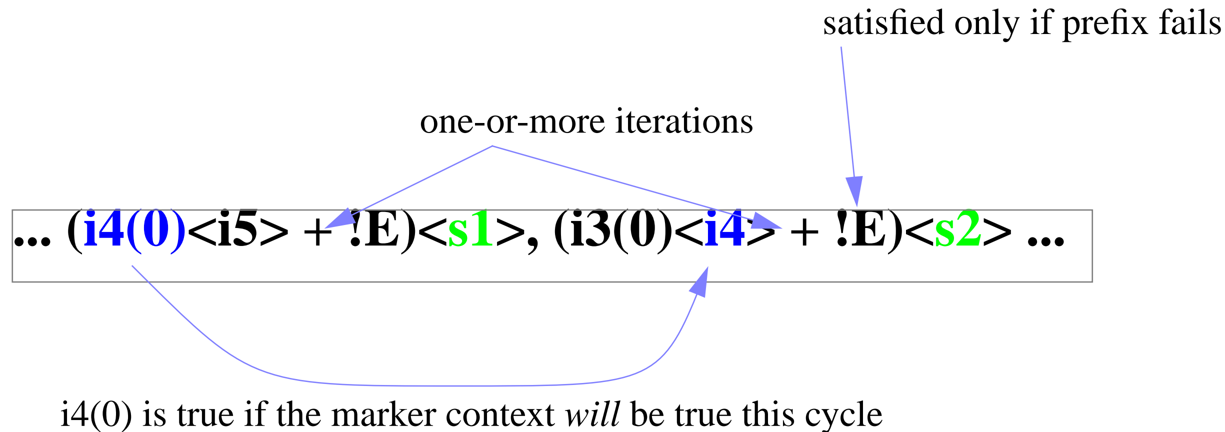
Interlocked Pipeline Example

- **Code format is local -- only the dependence on the next stage is identified**

```
moore {}              -- mealy machine format => actions available in same cycle as input
::
#input n              -- declare only input
#production machine trans it1 it2 it3 it4 it5
#marker i1 i2 i3 i4 i5 i6            -- create observation points for context
#action s0 [s0 <= '1';]              -- VHDL actions
#action s1 [s1 <= '1';]
#action s2 [s2 <= '1';]
#action s3 [s3 <= '1';]
#action s4 [s4 <= '1';]
::
machine -> (.*, trans);              -- Start transaction each clock
trans  -> (it5<i6> + !E)<s0>, (it4<i5> + !E)<s1>, (it3<i4> + !E)<s2>, (it2<i3> + !E)<s3>,
(it1<i2> + !E)<s4><i1>;
it1    -> i1(1);       -- 1-delay, this causes <s4> acceptance only on alternate cycles
it2    -> i2(0);       -- any kind of end throttling will work
it3    -> i3(0);
it4    -> i4(0);
it5    -> i5(0)|~n;    -- start new transaction only when n is samples true
::
```

Detailed Interlock

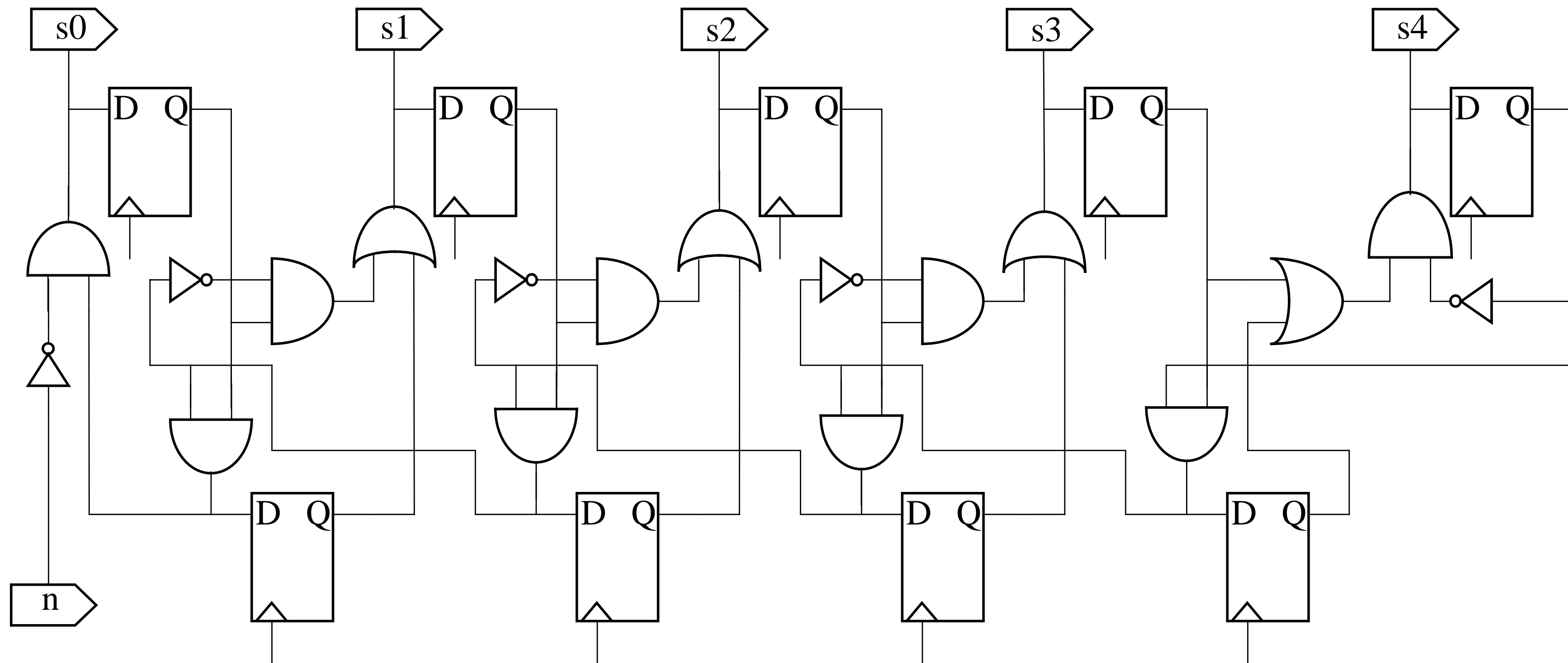
- **Each stage tries to detect a stall in the next stage, next cycle**



- **Since each context is activated on arrival of a control point, the construction detects if the next stage is halted (i.e. contains an active control point) if so -- it stalls the current stage.**
- **Note that *every* Zero delay marker is part of a combinatorial fan-in network**
- **Can easily create inconsistent logic -- PBS conservatively warns the user**

Interlocked Pipeline-- PBS Encoding/SIS Min Area

- Depth = 5*, States = 48 (24 unique)
- Literals = 24



Interlock Pipeline Comparison

- PBS Machine has 48 states -- Minimal machine has 24 states
- PBS version has 23 literals, depth of 5 (mapped in two-input gates)
- NOVA (PLA optimal) machine has 5 latches, 110 literals, depth of 14
- JEDI machine also has 5 latches, 60 literals, depth of 7
- Retiming Jedi machine increases latches to 12, saves 15% cycle time cost of 90 literals. (Depth is still 7).
- Retimed PBS machine still has 9 latches, but 4 levels -- nearly 2x as fast as best JEDI design, even given unrestricted literals
- PBS design mapped into Willamette technology (Yuji's mapper): required 3 levels of logic, 45 literals (10 gates, 11 inv) mapped.
- Retimed PBS machine also mapped into 3-levels, 51 literals (12 gates, 10 inv).