Sequential Logic Optimization

- State Minimization
  - Algorithms for State Minimization
- State, Input, and Output Encodings
  - Minimize the Next State and Output logic

Optimization in Context

- Understand the word specification
- Draw a picture
- Derive a state diagram and Symbolic State Table
- Determine an implementation approach (e.g., gate logic, ROM, FPGA, etc.)
- *Perform STATE MINIMIZATION*
- *Perform STATE ASSIGNMENT*
- Map Symbolic State Table to Encoded State Tables for implementation (*INPUT and OUTPUT encodings*)
- You can specify a specific state assignment in your Verilog code through `parameter` settings

Finite State Machine Optimization

- State Minimization
  - Fewer states require fewer state bits
  - Fewer bits require fewer logic equations
- Encodings: State, Inputs, Outputs
  - State encoding with fewer bits has fewer equations to implement
    - However, each may be more complex
  - State encoding with more bits (e.g., one-hot) has simpler equations
    - Complexity directly related to complexity of state diagram
  - Input/output encoding may or may not be under designer control

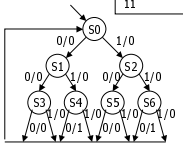
Algorithmic Approach to State Minimization

- Goal – identify and combine states that have equivalent behavior
- Equivalent States:
  - Same output
  - For all input combinations, states transition to same or equivalent states
- Algorithm Sketch
  - 1. Place all states in one set
  - 2. Initially partition set based on output behavior
  - 3. Successively partition resulting subsets based on next state transitions
  - 4. Repeat (3) until no further partitioning is required
    - states left in the same set are equivalent
  - Polynomial time procedure

State Minimization Example

- Sequence Detector for 010 or 110

Input Sequence	Present State	Next State X=0	Next State X=1	Output X=0	Output X=1
Reset	S0	S1	S2	0	0
0	S1	S3	S4	0	0
1	S2	S5	S6	0	0
00	S3	S0	S0	0	0
01	S4	S0	S0	1	0
10	S5	S0	S0	0	0
11	S6	S0	S0	1	0



Method of Successive Partitions

Input Sequence	Present State	Next State X=0	Next State X=1	Output X=0	Output X=1
Reset	S0	S1	S2	0	0
0	S1	S3	S4	0	0
1	S2	S5	S6	0	0
00	S3	S0	S0	0	0
01	S4	S0	S0	1	0
10	S5	S0	S0	0	0
11	S6	S0	S0	1	0

(S0 S1 S2 S3 S4 S5 S6)

(S0 S1 S2 S3 S5) (S4 S6)

(S0 S1 S2) (S3 S5) (S4 S6)

(S0) (S1 S2) (S3 S5) (S4 S6)

S1 is equivalent to S2

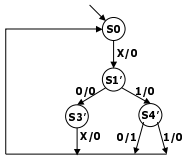
S3 is equivalent to S5

S4 is equivalent to S6

Minimized FSM

- State minimized sequence detector for 010 or 110

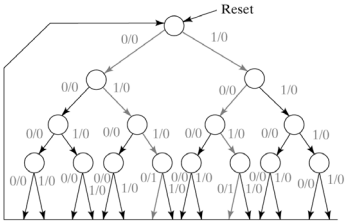
Input Sequence	Present State	Next State X=0	Next State X=1	Output X=0	Output X=1
Reset	S0	S1'	S1'	0	0
0 + 1	S1'	S3'	S4'	0	0
X0	S3'	S0	S0	0	0
X1	S4'	S0	S0	1	0



7 States reduced to 4 States
3 bit encoding replaced by 2 bit encoding

Another Example

- 4-Bit Sequence Detector: output 1 after each 4-bit input sequence consisting of the binary strings 0110 or 1010



State Transition Table

- Group states with same next state and same outputs

Input Sequence	Present State	Next State $X=0$	Next State $X=1$	Output $X=0$	Output $X=1$
Reset	S_0	S_1	S_2	0	0
0	S_1	S_3	S_4	0	0
1	S_2	S_5	S_6	0	0
00	S_3	S_7	S_8	0	0
01	S_4	S_9	S_{10}	0	0
10	S_5	S_{11}	S_{12}	0	0
11	S_6	S_{13}	S_{14}	0	0
000	S_7	S_0	S_0	0	0
001	S_8	S_0	S_0	0	0
010	S_9	S_0	S_0	0	0
011	S_{10}	S_0	S_0	1	0
100	S_{11}	S_0	S_0	0	0
101	S_{12}	S_0	S_0	1	0
110	S_{13}	S_0	S_0	0	0
111	S_{14}	S_0	S_0	0	0

S'10

Iterate the Row Matching Algorithm

Input Sequence	Present State	Next State $X=0$	Next State $X=1$	Output $X=0$	Output $X=1$
Reset	S_0	S_1	S_2	0	0
0	S_1	S_3	S_4	0	0
1	S_2	S_5	S_6	0	0
00	S_3	S_7	S_8	0	0
01	S_4	S_9	S'_{10}	0	0
10	S_5	S'_{11}	S'_{10}	0	0
11	S_6	S_{13}	S_{14}	0	0
000	S_7	S_0	S_0	0	0
001	S_8	S_0	S_0	0	0
010	S_9	S_0	S_0	0	0
011 or 101	S'_{10}	S_0	S_0	1	0
100	S_{11}	S_0	S_0	0	0
110	S_{13}	S_0	S_0	0	0
111	S_{14}	S_0	S_0	0	0

S'7

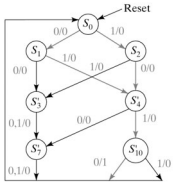
Iterate One Last Time

Input Sequence	Present State	Next State $X=0$	Next State $X=1$	Output $X=0$	Output $X=1$
Reset	S_0	S_1	S_2	0	0
0	S_1	S_3	S_4	0	0
1	S_2	S_5	S_6	0	0
00	S_3	S'_7	S'_7	0	0
01	S_4	S'_7	S'_{10}	0	0
10	S_5	S'_7	S'_{10}	0	0
11	S_6	S'_7	S'_7	0	0
not (011 or 101)	S'_7	S_0	S_0	0	0
011 or 101	S'_{10}	S_0	S_0	1	0

S'3 S'4

Final Reduced State Machine

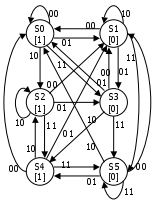
Input Sequence	Present State	Next State $X=0$	Next State $X=1$	Output $X=0$	Output $X=1$
Reset	S_0	S_1	S_2	0	0
0	S_1	S'_3	S'_4	0	0
1	S_2	S'_4	S'_3	0	0
00 or 11	S'_3	S'_7	S'_7	0	0
01 or 10	S'_4	S'_7	S'_{10}	0	0
not (011 or 101)	S'_7	S_0	S_0	0	0
011 or 101	S'_{10}	S_0	S_0	1	0



15 states (min 4 FFs) reduced to 7 states (min 3 FFs)

More Complex State Minimization

- Multiple input example



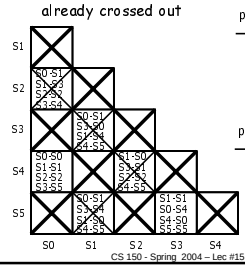
inputs here

present state	next state 00	01	10	11	output
S0	S0	S1	S2	S3	1
S1	S0	S3	S1	S4	0
S2	S1	S3	S2	S4	1
S3	S1	S0	S4	S5	0
S4	S0	S1	S2	S5	1
S5	S1	S4	S0	S5	0

symbolic state transition table

Minimized FSM

- Implication Chart Method
  - Cross out incompatible states based on outputs
  - Then cross out more cells if indexed chart entries are already crossed out



present state	next state 00	01	10	11	output
S0	S0	S1	S2	S3	1
S1	S0	S3	S1	S4	0
S2	S1	S3	S2	S4	1
S3	S1	S0	S4	S5	0
S4	S0	S1	S2	S5	1
S5	S1	S4	S0	S5	0

present state	next state 00	01	10	11	output
S0'	S0'	S1	S2	S3'	1
S1	S0'	S3'	S1	S3'	0
S2	S1	S3'	S2	S0'	1
S3'	S1	S0'	S0'	S3'	0

minimized state table (S0==S4) (S3==S5)

Minimizing Incompletely Specified FSMs

- Equivalence of states is transitive when machine is fully specified
- But its not transitive when don't cares are present

  e.g., state output

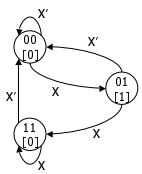
  S0 - 0 S1 is compatible with both S0 and S2

  S1 1 - but S0 and S2 are incompatible

  S2 - 1
- No polynomial time algorithm exists for determining best grouping of states into equivalent sets that will yield the smallest number of final states

Minimizing States May Not Yield Best Circuit

- Example: edge detector - outputs 1 when last two input changes from 0 to 1



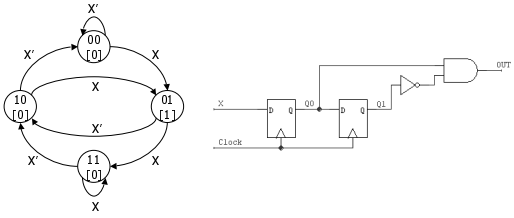
X	Q_1	Q_0	Q_1^+	Q_0^+
0	0	0	0	0
0	0	1	0	0
0	1	1	0	0
1	0	0	0	1
1	0	1	1	1
1	1	1	1	1
–	1	0	0	0

$Q_1^+ = X\ (Q_1 \text{ xor } Q_0)$

$Q_0^+ = X\ Q_1'Q_0'$

Another Implementation of Edge Detector

- "Ad hoc" solution - not minimal but cheap and fast



State Assignment

- Choose bit vectors to assign to each "symbolic" state
  - With n state bits for m states there are $2^n! / (2^n - m)!$ [log n <= m <= 2^n]
  - 2^n codes possible for 1st state, 2^n-1 for 2nd, 2^n-2 for 3rd, ...
  - Huge number even for small values of n and m
    - Intractable for state machines of any size
    - Heuristics are necessary for practical solutions
  - Optimize some metric for the combinational logic
    - Size (amount of logic and number of FFs)
    - Speed (depth of logic and fanout)
    - Dependencies (decomposition)

State Assignment Strategies

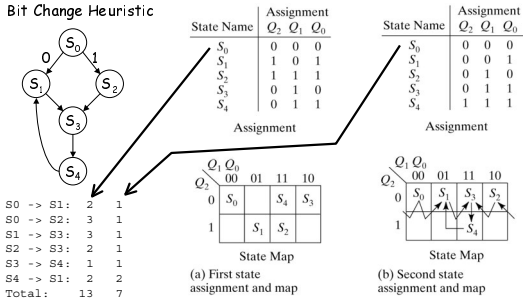
- Possible Strategies
  - Sequential - just number states as they appear in the state table
  - Random - pick random codes
  - One-hot - use as many state bits as there are states (bit=1 -> state)
  - Output - use outputs to help encode states
  - Heuristic - rules of thumb that seem to work in most cases
- No guarantee of optimality – another intractable problem

One-hot State Assignment

- Simple
  - Easy to encode, debug
- Small Logic Functions
  - Each state function requires only predecessor state bits as input
- Good for Programmable Devices
  - Lots of flip-flops readily available
  - Simple functions with small support (signals its dependent upon)
- Impractical for Large Machines
  - Too many states require too many flip-flops
  - Decompose FSMs into smaller pieces that can be one-hot encoded
- Many Slight Variations to One-hot
  - One-hot + all-0

State Maps and Counting Bit Changes

Bit Change Heuristic

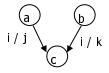


Adjacency Heuristics for State Assignment

- Adjacent codes to states that share a common next state
  - Group 1's in next state map

I	Q	Q+	O
i	a	c	j
i	b	c	k

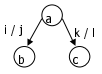
c = i * a + i * b



- Adjacent codes to states that share a common ancestor state
  - Group 1's in next state map

I	Q	Q+	O
i	a	b	j
k	a	c	l

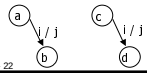
b = i * a
c = k * a



- Adjacent codes to states that have a common output behavior
  - Group 1's in output map

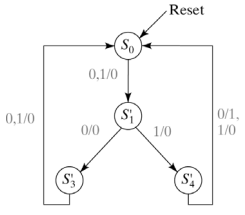
I	Q	Q+	O
i	a	b	j
i	c	d	j

j = i * a + i * c
b = i * a
d = i * c



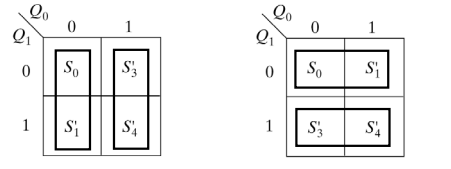
Heuristics for State Assignment

Successor/Predecessor Heuristics

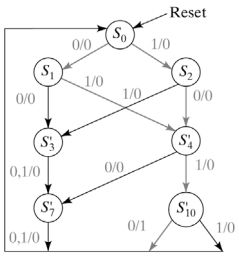


- High Priority: S'_3 and S'_4 share common successor state (S_0)
- Medium Priority: S'_3 and S'_4 share common predecessor state (S'_1)
- Low Priority:
  - 0/0: S_0, S'_1, S'_3
  - 1/0: S_0, S'_1, S'_3, S'_4

Heuristics for State Assignment



Another Example

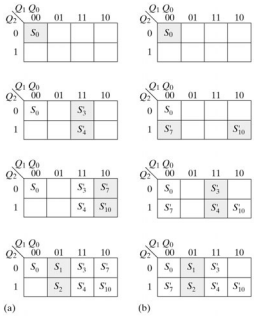


High Priority:
S'_3, S'_4
S'_7, S'_{10}

Medium Priority:
S_1, S_2
$2 \times S'_3, S'_4$
S'_7, S'_{10}

Low Priority:
0/0: $S_0, S_1, S_2, S'_3, S'_4, S'_7$
1/0: $S_0, S_1, S_2, S'_3, S'_4, S'_7, S_{10}$

Example Continued



(a) (b)

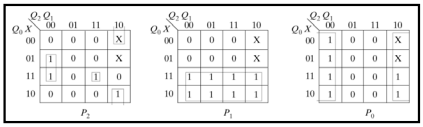
- Choose assignment for S0 = 000
- Place the high priority adjacency state pairs into the State Map
- Repeat for the medium adjacency pairs
- Repeat for any left over states, using the low priority scheme

Two alternative assignments at the left

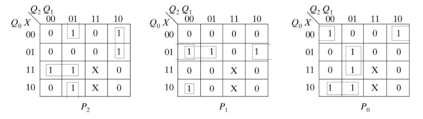
Why Do These Heuristics Work?

- Attempt to maximize adjacent groupings of 1's in the next state and output functions

Current State	Next State $X=0$	Next State $X=1$
(S_0) 000	001	101
(S_1) 001	011	111
(S_2) 101	111	011
(S'_3) 011	010	010
(S'_4) 111	010	110
(S'_7) 010	000	000
(S'_{10}) 110	000	000



Current State	Next State $X=0$	Next State $X=1$
(S_0) 000	001	010
(S_1) 001	011	100
(S_2) 010	100	011
(S'_3) 011	101	101
(S'_4) 100	101	110
(S'_7) 101	000	000
(S'_{10}) 110	000	000



General Approach to Heuristic State Assignment

- All current methods are variants of this
  - 1) Determine which states "attract" each other (weighted pairs)
  - 2) Generate constraints on codes (which should be in same cube)
  - 3) Place codes on Boolean cube so as to maximize constraints satisfied (weighted sum)
- Different weights make sense depending on whether we are optimizing for two-level or multi-level forms
- Can't consider all possible embeddings of state clusters in Boolean cube
  - Heuristics for ordering embedding
  - To prune search for best embedding
  - Expand cube (more state bits) to satisfy more constraints

Output-Based Encoding

- Reuse outputs as state bits - use outputs to help distinguish states
  - Why create new functions for state bits when output can serve as well
  - Fits in nicely with synchronous Mealy implementations

Inputs C	TL	TS	Present State	Next State	Outputs ST	H	F
0	–	–	HG	HG	0	00	10
–	0	–	HG	HG	0	00	10
1	1	–	HG	HY	1	00	10
–	–	0	HY	HY	0	01	10
–	–	1	HY	FG	1	01	10
1	0	–	FG	FG	0	10	00
0	–	–	FG	FY	1	10	00
–	1	–	FG	FY	1	10	00
–	–	0	FY	FY	0	10	01
–	–	1	FY	HG	1	10	01

HG = ST'H1'H0'F1 F0' + ST H1 H0'F1'F0
HY = ST H1' H0' F1 F0' + ST'H1' H0 F1 F0'
FG = ST H1' H0 F1 F0' + ST'H1 H0'F1'F0'
HY = ST H1 H0'F1'F0' + ST'H1 H0'F1'F0

Output patterns are unique to states, we do not need ANY state bits – implement 5 functions (one for each output) instead of 7 (outputs plus 2 state bits)

Current State Assignment Approaches

- For tight encodings using close to the minimum number of state bits
  - Best of 10 random seems to be adequate (averages as well as heuristics)
  - Heuristic approaches are not even close to optimality
  - Used in custom chip design
- One-hot encoding
  - Easy for small state machines
  - Generates small equations with easy to estimate complexity
  - Common in FPGAs and other programmable logic
- Output-based encoding
  - Ad hoc - no tools
  - Most common approach taken by human designers
  - Yields very small circuits for most FSMs

Sequential Logic Implementation Summary

- Implementation of sequential logic
  - State minimization
  - State assignment
- Implications for programmable logic devices
  - When logic is expensive and FFs are scarce, optimization is highly desirable (e.g., gate logic, PLAs, etc.)
  - In Xilinx devices, logic is bountiful (4 and 5 variable TTs) and FFs are many (2 per CLB), so optimization is not so crucial an issue as in other forms of programmable logic
  - This makes sparse encodings like One-Hot worth considering