EE4218 L3 - State Machine Optimization

1 Modelling Synchronous Circuits

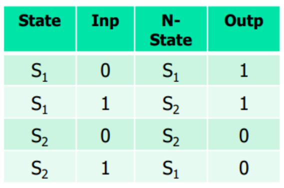
There are two main ways to model synchronous circuits; State-based models or Structural models.

- State-based model $\xrightarrow{\text{State Encoding}}$ Structural model
- Structural model $\xrightarrow{\text{State Extraction}}$ State-based model

1.1 State-based Model (ie. Behavioral Modelling)

Typically modelled using state-transition graphs.

- Model circuits as **FINITE-STATE MACHINES**
- Represent behavior via state-table/diagrams
- Explicit notion of state, implicit notion of area/timing
- State-based optimizations are...
  1. State Minimization (Merging similar states)
     - If two states are identical, there is no need for us to implement both of them
  2. State encoding (Assigning binary patterns to the symbolic states)
     - Represents the *symbolic* state function as a Boolean function, which can be implemented with *actual* hardware



State	Inp	N-State	Outp
S_1	0	S_1	1
S_1	1	S_2	1
S_2	0	S_2	0
S_2	1	S_1	0

Moore

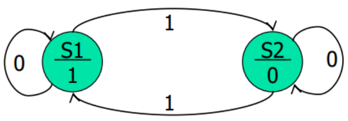


Figure 1: Example of State-based modelling of a Moore machine

1.2 Structural Modelling

Represent the circuit using *synchronous logic network*.

- Explicit notion of area/timing, implicit notion of state
- Structural-based optimizations are...
  1. Retiming
  2. Synchronous logic transformations (Not covered)

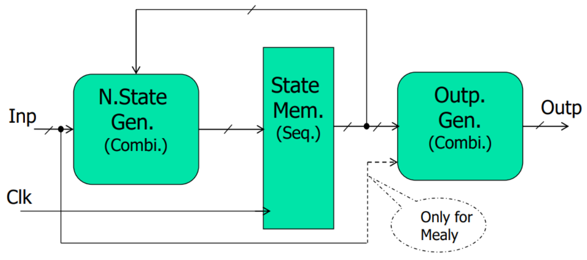


Figure 2: Generic structural model

Note that we will see how to transform the structural model of Fig2 into a synchronous logic network later.

2 State Machine Optimization

Assumptions

- Single-phase clocking
- No combinational feedback loops in the circuit (We operate at a higher abstraction level)
  - eg. In Fig3 below, a latch involves combinational feedback paths (to implement Memory)
  - However, we view the latch as a black-box unit

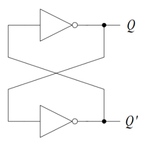


Figure 3: Bistable element used in latches

2.1 Optimization flow

1. FSM specification (ie. Algorithmic procedure to solve a problem)
2. State Minimization
3. State Encoding (Moving from State-based to Structural-based)
4. **COMBINATIONAL** logic optimization
   - Combinational <u>next-state</u> logic optimization
   - Combinational <u>output</u> logic optimization
5. State Extraction (Moving back from Structural-based to State-based)
   - Recovers state information from structural model, to verify the structural model
6. Retiming and synchronous transformations
   - For the synchronous portion of the State Machine

2.2 State Encoding

Lets cover State Encoding first, i.e Converting from State-based model to Structural-based model.

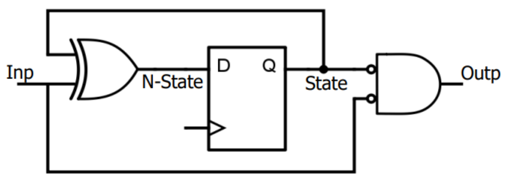
As seen in Fig4, structural modelling is not unique. Which circuit is better depends on the PPA requirements.

State	Inp	N-State	Outp
S_1	0	S_1	1
S_1	1	S_2	0
S_2	0	S_2	0
S_2	1	S_1	0

State assignment
S1 -> 0
S2 -> 1

State	Inp	N-State	Outp
0	0	0	1
0	1	1	0
1	0	1	0
1	1	0	0

N-State = State $\oplus$ Inp
Outp = State' . Inp'



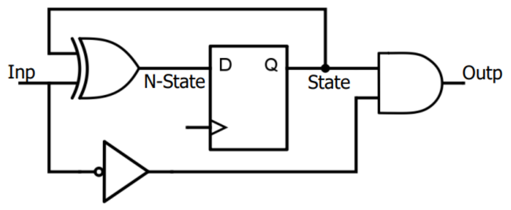
(a) State Encoding 1

State	Inp	N-State	Outp
S_1	0	S_1	1
S_1	1	S_2	0
S_2	0	S_2	0
S_2	1	S_1	0

State assignment
S1 -> 1
S2 -> 0

State	Inp	N-State	Outp
1	0	1	1
1	1	0	0
0	0	0	0
0	1	1	0

N-State = State $\oplus$ Inp
Outp = State . Inp'



(b) State Encoding 2

Figure 4: Non-uniqueness of Structural models

2.3 State Minimization

Objective is to reduce the number of states (and hence the hardware area)

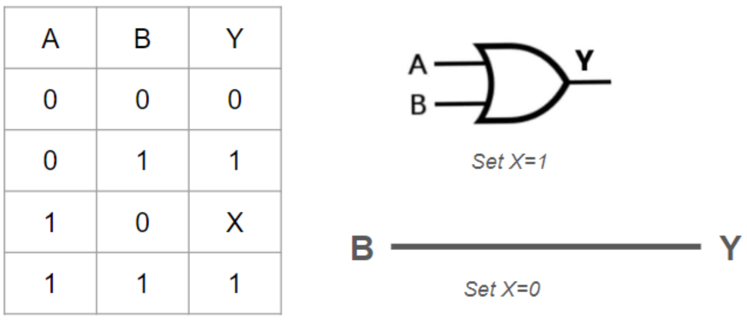
Note that the complexity of State Minimization depends on the **degree** of FSM specification

- Completely-specified FSM's (CS-FSM)
  - No don't-care condition(s)
  - There are polynomial time solutions to do state minimization
- Incompletely-specified FSM (IS-FSM)
  - $\exists$ don't-care condition(s); Therefore there are unspecified transitions and/or outputs
  - Problem is intractable (NP). We have to brute-force iterate through all possible combinations of the X's, to see which scenario is best

2.3.1 What's the deal with don't-cares?

They complicate analysis of the system.

Fig5 is an example of how don't cares increase complexity (in the context of logic simplification, not state minimization).



A	B	Y
0	0	0
0	1	1
1	0	X
1	1	1

Figure 5: Hardware implementation is simplified, if we set X=0

- Don't-cares in the output means that the output was not specified (ie. Output can randomly be 0 or 1, it does not affect system functionality). It does not mean that the output doesn't change.

  Why? Because recall that if output doesn't change when input changes, it implies that the system has memory. Since truth tables are implementing combinational logic, we cannot have any form of memory.
- More complicated scenarios can be tackled with K-maps.
- Recognize that the inclusion of X's means that we have to try out every possible combination of X's to find out which scenario gives us the 'best' circuit. There is no polynomial time method.

Lets move back to the context of state minimization now.

For an IS-FSM, each X denotes two (X=0, X=1) possible CS-FSMs. We must then state-minimize each CS-FSM using the polynomial time methods, before comparing across all permutations of the CS-FSMs to find the solution for state-minimizing the original IS-FSM.

2.3.2 State Minimization - Completely specified FSMs

Note: The resulting "minimized" FSM is **unique**.

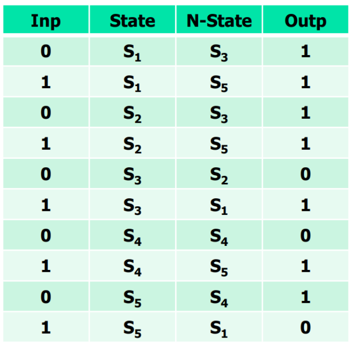
2.3.2.1 Concept of *equivalent* states

- Assume we have two states S_1 and S_2. S_1 and S_2 defined as *equivalent* iff for some input sequence...
  - Their (combinational) outputs are identical, and
  - Their (combinational) next-states are *equivalent* (note the recursiveness) as well
- Note that equivalence is *transitive*
  - S_1 is equivalent to $S_2 \wedge S_2$ is equivalent to $S_3 \longrightarrow S_1$ is equivalent to S_3

2.3.2.2 Algorithm for State Minimization of CS-FSMs

1. Π_0 : Initial partition.
   - Assume that all states $\{S_1, S_2, \ldots, S_k\}$ are equivalent
   - This means that our FSM is a *combinational-only* machine (combinational machines have no concept of states; ie. All next-states are equivalent) that produces the same output for all inputs
   - This is too optimistic an assumption, so we will refine it accordingly below...
2. Π_1 : Partition based on Output.
   - We consider $\{S_i, S_j\}$ to be equivalent iff for some input, their outputs are identical
   - Partition $\{S_1, S_2, \ldots, S_k\}$ accordingly
3. $\Pi_k, k \geq 2$: Iterative partitioning.
   - $\{S_i, S_j\}$ were equivalent in $\Pi_{k-1} \wedge \{S_m, S_n\}$ (next states of $\{S_i, S_j\}$) were equivalent in Π_{k-1} for some input $\longrightarrow$ $\{S_i, S_j\}$ are considered "truly" equivalent now.
   - Keep repeating until we reach convergence (Suppose S_i and S_j which were previously in the same block for Π_{k-2} were changed to be in different blocks for Π_{k-1}. Then during Π_k, we just need to re-check equivalence for any states S_k that had S_i or S_j as their next-states)
4. $\Pi_{k+1} = \Pi_k$: Convergence.
   - No more simplification can be done, we have accomplished state minimization.

Let's examine an example.



Inp	State	N-State	Outp
0	S_1	S_3	1
1	S_1	S_5	1
0	S_2	S_3	1
1	S_2	S_5	1
0	S_3	S_2	0
1	S_3	S_1	1
0	S_4	S_4	0
1	S_4	S_5	1
0	S_5	S_4	1
1	S_5	S_1	0

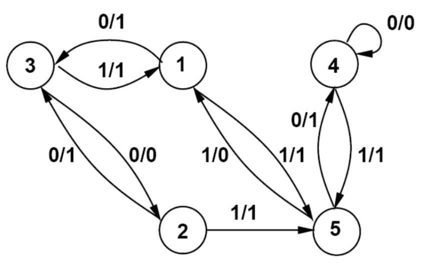


Figure 6: Perform state minimization on this completely-specified state model

1. $\Pi_0 = \{S_1, S_2, S_3, S_4, S_5\}$
2. $\Pi_1 = \{S_1, S_2\}, \{S_3, S_4\}, \{S_5\}$; Partition based on Output
3. $\Pi_2 = \{S_1, S_2\}, \{S_3, S_4\}, \{S_5\}$; Iterative partition 0, consider $S_i = S_1, S_j = S_2$
   - Consider Input = 0 for $\{S_1, S_2\}$.
     - S_1's next-state is S_3, S_2's next-state is S_3
     - S_3 must be equivalent to S_3, since they are the same state
   - Consider Input = 1 for $\{S_1, S_2\}$.
     - S_1's next-state is S_5, S_2's next-state is S_5
     - S_5 must be equivalent to S_5, since they are the same state
   - Therefore, S_1 equivalent to S_2 (and grouped in the same 'block')
4. $\Pi_3 = \{S_1, S_2\}, \{S_3\}, \{S_4\}, \{S_5\}$; Iterative partition 1, consider $S_i = S_3, S_j = S_4$
   - Consider Input = 0 for $\{S_3, S_4\}$.
     - S_3's next-state is S_2, S_4's next-state is S_4
     - Is S_2 equivalent to S_4? No, since they were not in the same 'block' for Π_2
   - Therefore, S_3 cannot be equivalent to S_4 (and they must be grouped in seperate 'blocks')
5. No furthur refinement can be made to Π_3, therefore we have reached convergence and achieved state minimization.

2.3.3 State Minimization - Incompletely specified FSMs

Note: The resulting "minimized" FSM is **not unique**.

2.3.3.1 Concept of *compatible* states

- Assume we have two states S_1 and S_2. S_1 and S_2 defined as *compatible* iff for some **applicable** input sequence ...
  - Their (combinational) outputs are identical **if specified**, and
  - Their (combinational) next-states are *compatible* (note the recursiveness) as well
- The key difference in this definition, is that the input sequence must be **applicable**, before we can even consider whether the outputs are identical (if they are even **specified** at all)
  - An inapplicable input is one where it is don't-care (X). By definition, we then wouldn't care what it's output is.
  - Only if the input is applicable, do we have to consider whether the output is identical/next-state is compatible
  - An example is shown in Fig7 below. FSM_B receiving an 'X', means the particular input (outputted from FSM_A) is guaranteed not to happen (that's why it's 'X' in the first place). So we need not bother with what FSM_B outputs in this case.

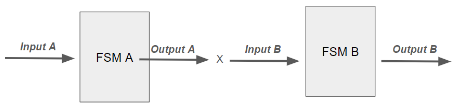


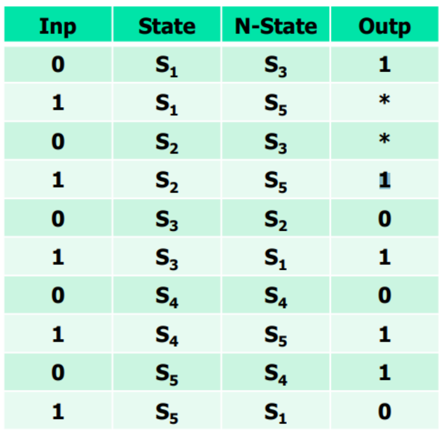
Figure 7: Propagation of 'X' from FSM_A to FSM_B

- Note that compatibility is **not** transitive. We will prove this via example.
  - Suppose for a particular input; S_1 has output 10, S_2 has output 1*, S_3 as output 11
    - S_1 compatible with S_2, assuming their next-states are compatible
    - S_2 compatible with S_3, assuming their next-states are compatible
    - S_1 not compatible with S_3, irregardless of their next-states. Thereby proving transitivity doesn't hold.

2.3.3.2 Algorithm for State Minimization of IS-FSMs

1. n don't cares implies 2^n CS-FSMs
2. Hence, we have to iterate through and minimize 2^n possible CS-FSMs, to find the minimized IS-FSM

Let's examine an example.



Inp	State	N-State	Outp
0	S_1	S_3	1
1	S_1	S_5	*
0	S_2	S_3	*
1	S_2	S_5	1
0	S_3	S_2	0
1	S_3	S_1	1
0	S_4	S_4	0
1	S_4	S_5	1
0	S_5	S_4	1
1	S_5	S_1	0

Figure 8: Perform state minimization on this incompletely-specified state-based model

- One possible CS-FSM: Replace all '*' with '1'; (ie. The previous example on CS-FSM)
  - $\Pi_{convergence} = \{S_1, S_2\}, \{S_3\}, \{S_4\}, \{S_5\};$
- Another possible CS-FSM: Replace all '*' with '0';
  - $\Pi_{convergence} = \{S_1, S_5\}, \{S_2, S_3, S_4\}$

  - Intuitively, we can see that replacing all '*' with '0' results in a better minimization. This is because there are less possible output combinations, therefore more terms are grouped 'in the same block' in Π_1.

- Another possible CS-FSM: Replace 1^{st} '*' with '0', 2^{nd} '*' with '1';
  - Intuitively, you can see that this will result in a worse minimization. Using the same logic as before, this is because now there are more possible output combinations.

2.3.3.3 Important difference between Equivalence and Compatible

We already know that equivalence is transitive, compatible is non-transitive. However, we also note that the following important difference

- If S_1 and S_2 are 'truly' equivalent (ie. belonging to the same 'group' from Π_2 onwards), then they **will** be grouped together in the final optimized minimized CS-FSM (ie. belonging to the same 'group' in $\Pi_{convergence}$)
- Even if S_1 and S_2 are 'truly' compatible (ie. belonging to the same 'group' from Π_2 onwards), they **might not** be grouped together in the final optimized minimized IS-FSM (ie. might belong to different 'groups' in $\Pi_{convergence}$)
  - You can observe this from the Fig8 example on IS-FSM.

    S_1 and S_2 are 'truly' compatible, yet they do not belong to the 'best' minimized IS-FSM (corresponding to when we replace all '*' with '0')

2.3.3.4 Compatibility and Implications

Note: Implications also exist in the case of Equivalence, we are just explaining it wrt Compatibility context now.

There is a notion of 'implicit compatibility', where claiming S_i and S_j are compatible *implies* that S_m and S_n are compatible. Referring back to the Fig8 example ...

- $\{S_3, S_4\}$ are compatible $\xleftrightarrow{implies}$ $\{S_2, S_4\}$ are compatible $\wedge$ $\{S_1, S_5\}$ are compatible

2.3.4 Formal definition of the State Minimization problem

We want to find the minimal number of partition groups st ...

1. All states are covered (ie. Every state S_i belongs to some group ϕ_j in our final partition $\Pi_{convergence}$)
2. All implications are satisfied (we term this the *closure* property)

Exact solutions are computationally expensive, but good approximation methods do exist (not covered).

2.4 State Encoding (Revisited)

As seen in Section 2.2, State Encoding is the process of assigning bit-vectors to each symbolic state.

2.4.1 How many possible ways are there to do State Encoding?

Rewriting the question; Given m states and n bits, how many possible state-assignments are there?

- We know that the *minimal* number of bits required to encode m states is $n = ceil\Big(log_2(m)\Big)$ bits
- Therefore with the lower limit in-place, we have $n \geq ceil\Big(log_2(m)\Big)$
- Now let's work out how many ways there are to encode some state S_i, i $\leq$ m
  1. We have 2^n possible ways to encode the *first* state S_1
  2. We have $2^n - 1$ possible ways to encode the *second* state S_2; One permutation is already taken up by S_1
  3. We have $2^n - 2$ possible ways to encode the *third* state S_3; Two permutations are already taken up by S_1, S_2
  4. I think you get the pattern ...
- Therefore, we have $\Big((2^n) * (2^n - 1) * (2^n - 2) * \cdots * (2^n - (m+1))\Big) = \frac{(2^n)!}{(2^n - m)!}$ possible ways to encode m states using n bits

2.4.2 Optimal state encoding

With such a large solution space, which is the *optimal* one? Well, it depends on what you are trying to optimize for.

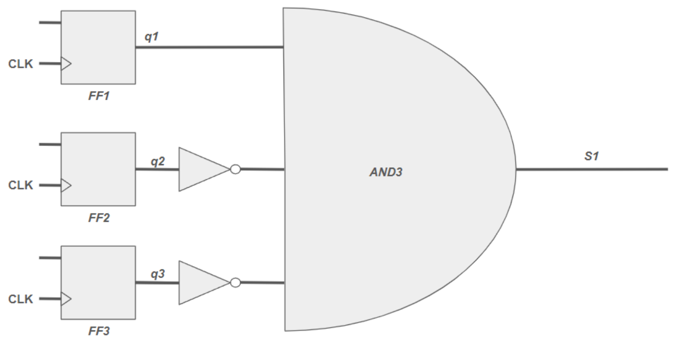
- For example, if we are trying to optimize for size (amount of combinational logic and number of FFs), we might choose $n = ceil\Big(log_2(m)\Big)$; i.e Lowest number of FFs required to encode all states.
  - Subsequently we could perform *sequential* encoding, where we encode states in an incremental binary fashion. eg. $S_1 \rightarrow 000$, $S_2 \rightarrow 001$, $S_3 \rightarrow 010$, $S_4 \rightarrow 011$
- However, we might also choose to optimize for speed (Depth of combinational logic and fanout), in which-case the lowest number of FFs might not be ideal. One popular method to optimize for speed is utilizing *One-Hot Encoding*.

2.4.2.1 One-Hot Encoding

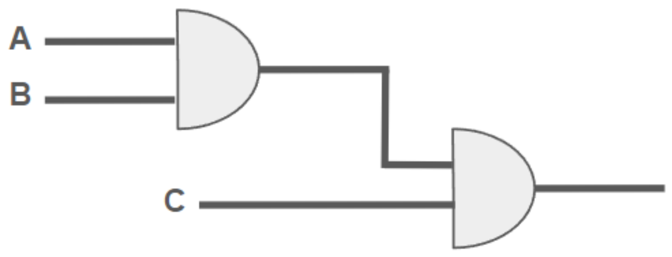
One-Hot Encoding utilizes as many FFs as there are states, and assigns in a one-hot manner.
eg. $S_1 \rightarrow 0001$, $S_2 \rightarrow 0010$, $S_3 \rightarrow 0100$, $S_4 \rightarrow 1000$

The main advantage of One-Hot Encoding, is that it reduces (might not optimally reduce, but still reduces) the complexity/area of the resulting combinational logic. Let's why.

- Suppose we have 6 states $\{S_1, S_2, \ldots, S_6\}$ in an FSM.
- Implementing in sequential encoding method, we use 3 FFs $\{FF_1, FF_2, FF_3\}$, resulting in the circuit below.



(a) Sequential Encoding to determine state S_1



(b) 3AND is formed from 2 2AND

Figure 9: Sequential Encoding saves on FFs, but has more combinational delay

- Implementing in one-hot encoding method, we use 6 FFs $\{FF_1, FF_2, \ldots, FF_6\}$, resulting in the circuit below.

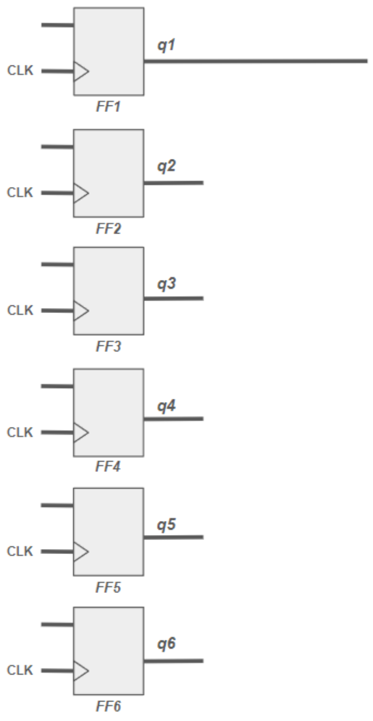


Figure 10: One-Hot Encoding to determine state S_2; Splurges on FFs, but less combinational delay

2.4.3 Heuristics for State Minimization

The problem of *optimal* state minimization is intractable as mentioned previously. Therefore, we need to rely on heuristic methods for practical applications.

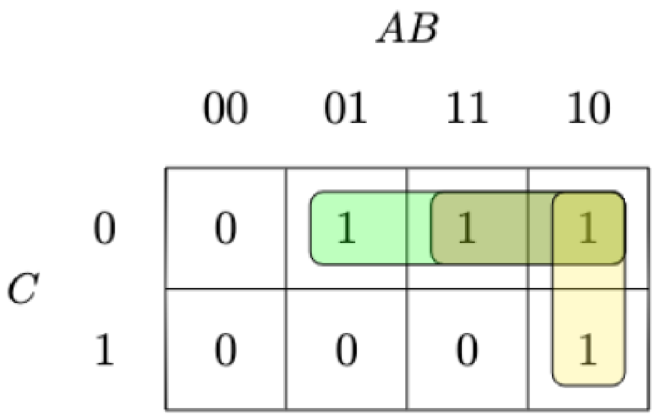
Most of these heuristic methods rely on assiging **adjacent codes** to states that are very 'similar', with adjacent codes having the property of requiring only one bit change for each successive state in a counting sequence.

The purpose of using adjacent codes is because simplification using K-Maps will be better.

- Adjacent codes (1bit difference in the binary assignment) causes 1s to be placed beside each other in the K-Map
- A bigger 'cluster' of 1s in the K-Map means that more variables can be grouped together, and more simplification can be done

A	B	C	E
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	0

(a) Truth table for some FSM



(b) K-Map, notice that the adjacent codes in the FSM are located side-by-side and allowing for greater 'clustering'

Figure 11: Showing why adjacent codes result in better K-Map simplification

Some examples would be ...

- Clustering of 1s in next-state K-Map
  - Assigning adjacent codes to states that share a *common next-state*
  - Assigning adjacent codes to states that share a *common predecessor-state*
- Clustering of 1s in output K-Map
  - Assigning adjacent codes to states that have a *common output behavior*

Popular choices for adjacent codes are *Gray Codes* and *Johnson Codes.*

2.5 State Extraction

Not covered in detail, but the basic idea is straight-forward.

- A circuit with n FFs has 2^n states, however not all states may be reachable
  - eg. Suppose we have 3 FFs implementing 3 states, in a one-hot encoding scheme
    - Legal states are only 001,010,100
- Therefore, there is a need to verify whether our structural model erroneously enters an illegal state

2.5.1 State Extraction Algorithm

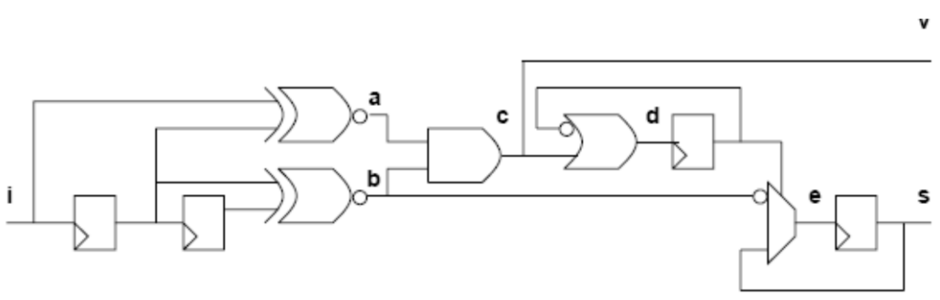
1. Find all states reachable from the initial state, and add these states to the set of reachable states
   - Note that we are able to find next-state from current-state, since we have the combinational expression for next-state generation from our structural model
2. From this set, determine all it's reachable states, and add these states to the set of reachable states
3. Repeat above step until convergence is reached (no new reachable states can be added to set)

2.6 Retiming

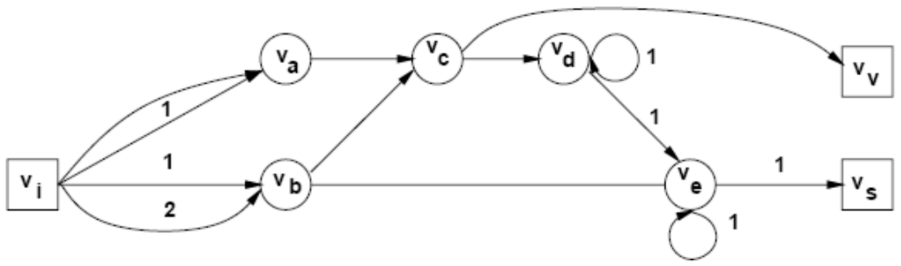
EE4415 covers this in much greater detail.

2.6.1 Synchronous Network Graph (SNG)

- Vertices: SOP/POS combinational logic; *Logic gates*
- Edge weight: Synchronous delay; *Registers/FFs*
- Edge direction: Dependencies; *Nets*



$$
\begin{aligned}
a^{(n)} &= i^{(n)} \overline{\oplus}\, i^{(n-1)} \\
b^{(n)} &= i^{(n-1)} \overline{\oplus}\, i^{(n-2)} \\
c^{(n)} &= a^{(n)} b^{(n)} \\
d^{(n)} &= c^{(n)} + d'^{(n-1)} \\
e^{(n)} &= d^{(n)} e^{(n-1)} + d'^{(n)} b'^{(n)} \\
v^{(n)} &= c^{(n)} \\
s^{(n)} &= e^{(n-1)}
\end{aligned}
$$

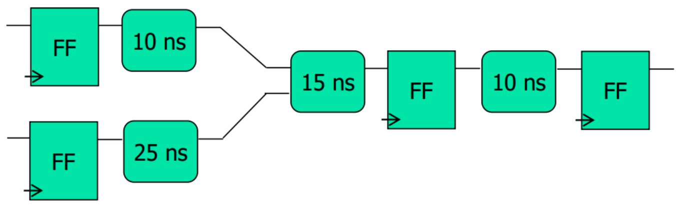


$$
\begin{aligned}
a &= i \,\overline{\oplus}\, i@1 \\
b &= i@1 \,\overline{\oplus}\, i@2 \\
c &= a\ b \\
d &= c + d@1' \\
e &= d\ e@1 + d'\ b' \\
v &= c \\
s &= e@1
\end{aligned}
$$

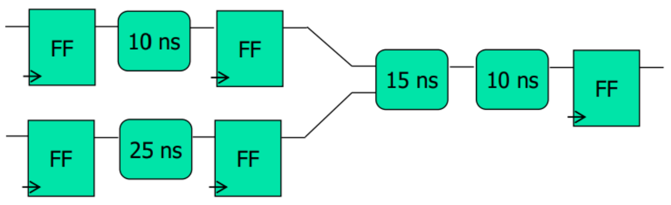
Figure 12: SNG derived from a structural model

2.6.2 Basic concept of Retiming

Rearranging of register positions to either reduce *critical path delay* **OR** *register count* is termed Retiming.



(a) Reduced register count



(b) Reduced critical path delay

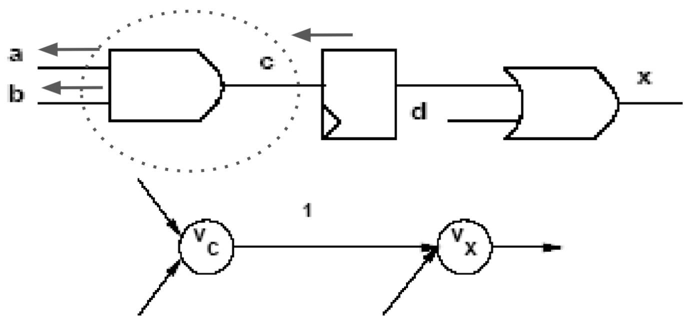
Figure 13: Register Retiming for different optimization targets

Note that retiming still maintains the original **system** functionality (ie. Delay between I/O does not change).

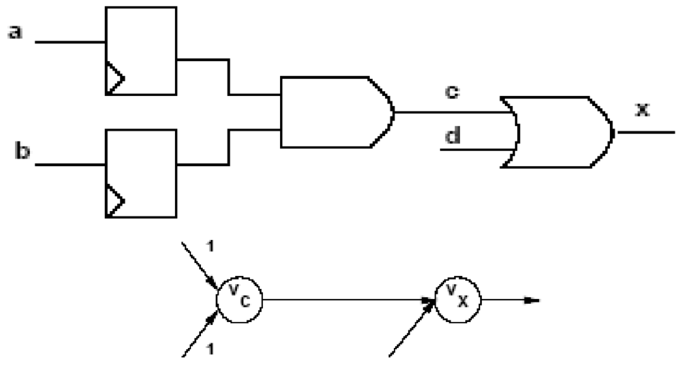
Observe the example in Fig14, where we do *backward* retiming.

2.6.3 Retiming on the SNG

Retiming is best understood from the SNG's POV.



(a) Original



(b) Backward-retimed

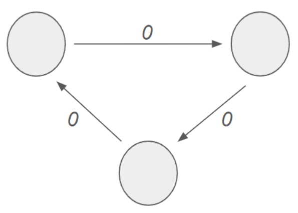
Figure 14: Backward Retiming (remove register from output, insert registers at input)

For all cases of retiming (*forward* or *backward*) retiming, we observe that ...

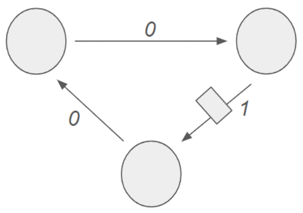
- Network topology is preserved
- Area is affected (register count is changed)
- Clock frequency is affected (path delays between register pairs are changed)
  - If *critical* path delay is reduced, then clock frequency is increased as a result
  - We note that path delays between I/O register pairs **must not** be changed, to preserve system functionality

2.6.3.1 Retiming assumptions

1. **Cycles** must have positive weights (ie. No combinational feedback loop can exist)
   - This is because register retiming on a recursive DFG (ie. one with a loop) **alters** the system functionality, which is unacceptable
   - Therefore, any combinational feedback loop must be broken by at least one register.



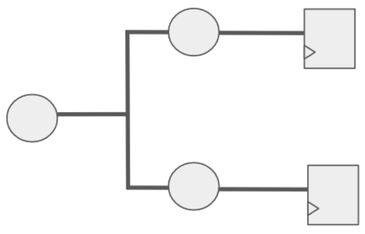
(a) Combinational feedback loop



(b) Broken combinational feedback loop

Figure 15: Breaking of combinational feedback loop using registers

2. Edges must have non-negative weights
3. Combinational logic (ie. vertices) have no fan-out delay dependency (ie. Delay is independent of loading)


(a) Input vertex has fanout of 2

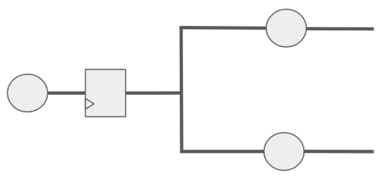
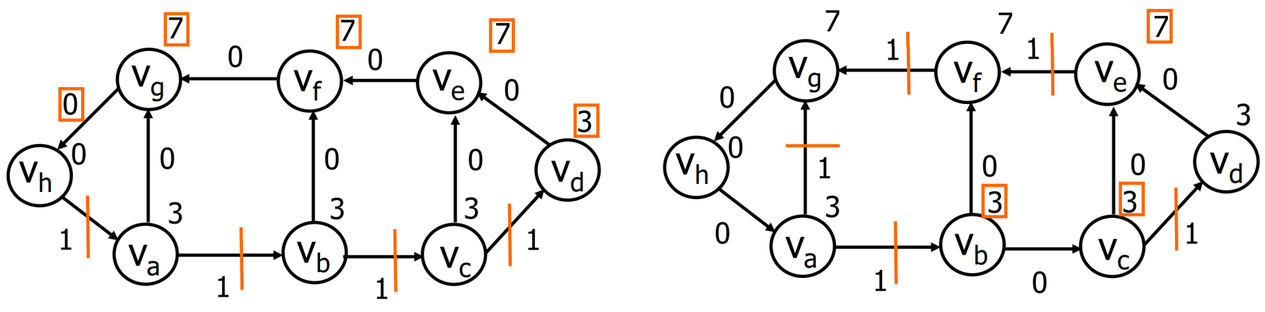

(b) Input vertex has fanout of 1

Figure 16: Vertex fan-out

4. No false path analysis

2.6.3.2 Retiming on SNG

Note: Critical path is always calculated from some *launching* register to a *capturing* register



(a) Original; Critical delay=24 (b) Retimed; Critical delay=13

Figure 17: Retiming on SNG

Furthur note that for the same critical delay, the corresponding retimed graph is **not** unique.

2.6.3.3 Relaxation-based retiming

Idea is to look for paths with excessive delays, and reduce the delay via retiming (ie. Insert registers along the path)

1. Start with a desired cycle-time φ that we wish to achieve
2. Find vertices $\{v_1, v_2, \ldots, v_j\}$ with *data ready* $> \varphi$
   - *data ready* of a vertex v_i is defined as $\Big((\text{delay from a } \textbf{register} \text{ to } v_i) + (\text{weight of vertex } v_i)\Big)$
3. Iteratively retime these vertices $\{v_1, v_2, \ldots, v_j\}$
   - Since register retiming must 'take' registers from somewhere, other paths may become longer
   - Also ensure that no edge has negative weight after 'taking' of registers
4. If at some iteration Φ_k there is no vertex v_i with *data ready* $> \varphi$, then a legal retiming has been found
   - We might further optimize by choosing a smaller φ, and repeating the process
5. If a legal retiming is not found by iteration Φ_m where m is our chosen *iteration bound*, then no legal retiming exists for that φ
   - Try again with a larger φ

2.6.3.4 Relaxation retiming on SNG

Assume that our goal is to have $\varphi = 13$

1. Remove all edges with weight ≥ 1, to identify *data ready* of vertices better
2. We identify that v_f has a *data ready* of 17, let's retime that first

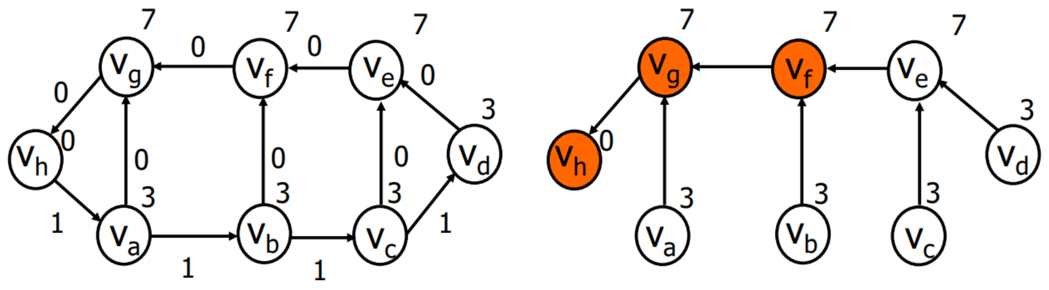


Figure 18: Iteration Φ_1

3. We have retimed v_f accordingly. Now simply repeat the previous steps to identify that v_g needs to be retimed now.

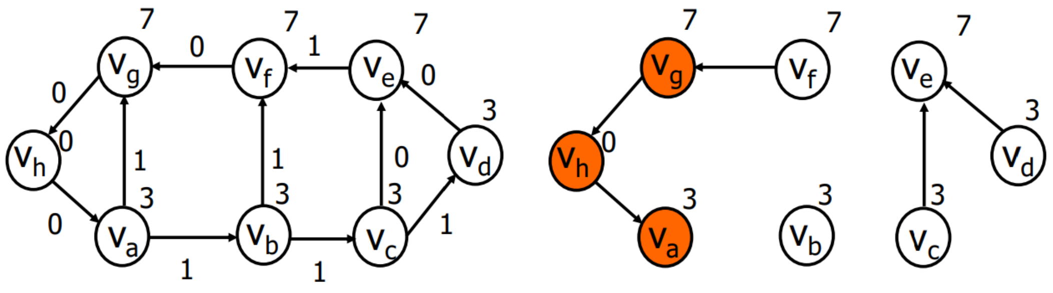


Figure 19: Iteration Φ_2

4. We have retimed v_g accordingly, and reached our desired φ

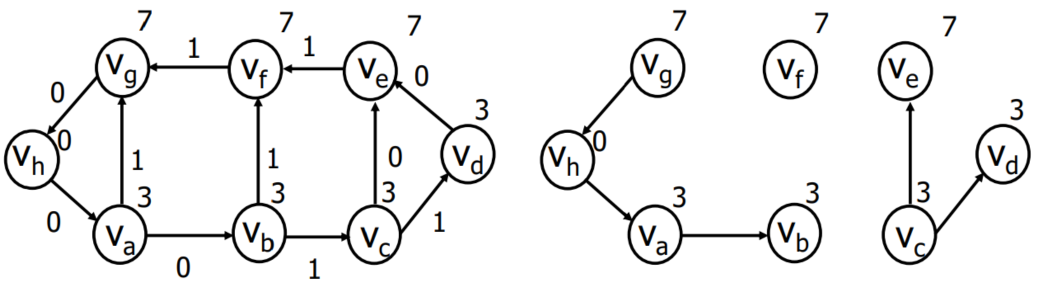


Figure 20: Iteration Φ_3; Convergence

2.6.3.5 Retiming for minimal area

Previously, all our discussion was on retiming for minimal *delay*. We can also retime for minimal *area*.

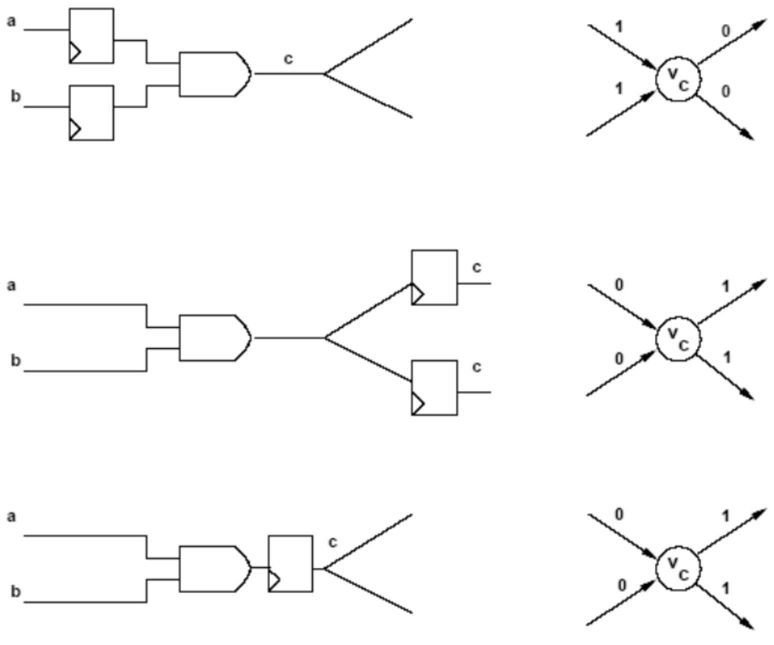


Figure 21: Retiming for minimal area