Agenda

- Finite Automata
- Equivalence of Finite Automata
- Product of Finite Automata
- Acceptors for Finite Sequences
- Büchi Automata and acceptance of infinite sequences
- CNF Satisfiability
- Equivalence Checking
  - Combinational Equivalence Checking
    - Register Correspondence
    - Equivalence Checking of Retimed Circuits
  - Sequential Equivalence Checking
  - Equivalence and Minimization Algorithms
Finite Automaton

A finite deterministic automaton $M$ (transducer, Mealy machine, finite state machine FSM) is a 6-tuple:

$$M = (Q, \Sigma, \Delta, \delta, \lambda, q^0)$$

where:

- $Q$ is the finite set of states
- $\Sigma$ is the input alphabet
- $\Delta$ is the output alphabet
- $\delta: Q \times \Sigma \rightarrow Q$ is the transition function
- $\lambda: Q \times \Sigma \rightarrow \Delta$ is the output function
- $q^0$ is the start state (initial state)

If $\lambda$ is of the form $\lambda: Q \rightarrow \Delta$, then we have a Moore machine.
State and Output Sequences

**Path function:** \( \delta^*: Q \times (N \rightarrow \Sigma) \rightarrow Q \)

Given an input sequence \( \tilde{a} \), we have:
\[
\delta^*(q, \tilde{a}) := q' \text{ with } q^0 := q, q^{i+1} = \delta(q^i, a^i), \text{ and } q' := q^{\mid \tilde{a} \mid}
\]

**Path output sequence:** \( \lambda^*: Q \times (N \rightarrow \Sigma) \rightarrow (N \rightarrow \Delta) \)

Given an input sequence \( \tilde{a} \), we have:
\[
\lambda^*(q, \tilde{a}) := \tilde{u} \text{ with } q^0 := q, q^{i+1} = \delta(q^i, a^i), \text{ and } u^i = \lambda(q^i, a^i)
\]
Automata Equivalence

Two automata $M$ and $M'$ are called equivalent, if for an arbitrary input sequence applied at both automata, the same output sequence results:

$$\forall \bar{a} \cdot \lambda^*(q^0, \bar{a}) = \lambda'^*(q^0, \bar{a})$$
State Equivalence

Given two Mealy machines with the same input and output alphabet, \( M = (Q, \Sigma, \Delta, \delta, \lambda, q^0) \) and \( M' = (Q', \Sigma, \Delta, \delta', \lambda', q'^0) \).

The state equivalence relation \( \sim \subseteq Q \times Q' \) is the largest relation which satisfies the following:

\[
q \sim q' : \iff \forall a, a \in \Sigma . \; \lambda(q, a) = \lambda'(q', a) \text{ and } \delta(q, a) \sim \delta'(q', a)
\]

Two states \( q \) and \( q' \) are said to be equivalent, if \( q \sim q' \) holds.

Results:

- It holds that \( \forall \bar{a}, \bar{a} \in (N \rightarrow \Sigma) . \; q \sim q' \Rightarrow \delta^*(q, \bar{a}) \sim \delta'^*(q', \bar{a}) \)

- Two Mealy machines \( M \) and \( M' \) are equivalent, written as \( M \approx M' \), iff their initial states are equivalent: \( q^0 \sim q'^0 \).
State Minimization

- **Necessary and sufficient condition for two states to be equivalent:**

  \[ q_1 \sim q_2 \iff \forall a, a \in \Sigma \ . \ \lambda(q_1, a) = \lambda(q_2, a) \quad \text{and} \quad \delta(q_1, a) \sim \delta(q_2, a) \]

- **Equivalent states can be merged**
The product automaton of two automata \( M = (Q, \Sigma, \Delta, \delta, \lambda, q^0) \) and 
\( M' = (Q', \Sigma, \Delta, \delta', \lambda', q'^0) \) is defined as:

\[
M^p = (Q \times Q', \Sigma, \Delta, \delta^p, \lambda^p, (q_0, q'^0))
\]

with \( \delta^p: (Q \times Q') \times \Sigma \rightarrow (Q \times Q') \) and \( \lambda^p: (Q \times Q') \times \Sigma \rightarrow B \), defined by:

\[
\delta^p((q, q'), a) := (\delta(q, a), \delta'(q', a))
\]
\[
\lambda^p((q, q'), a) := (\lambda(q, a) = \lambda'(q', a))
\]

The product delivers only a value \( B \) which indicates whether for a given input the outputs of both automata are equal (T) or not (F).
Acceptors

A deterministic finite acceptor (called DFA) \( M^a \) is a 5-tuple:

\[
M^a = (Q, \Sigma, \delta, q^0, F)
\]

where:

- \( Q \) is the finite set of states
- \( \Sigma \) is the input alphabet
- \( \delta: Q \times \Sigma \rightarrow Q \) is the transition function
- \( q^0 \) is the start state (initial state)
- \( F \subseteq Q \) is the set of final states (accepting states)

A finite sequence \( \bar{a} \) is said to be accepted by \( M^a = (Q, \Sigma, \delta, q^0, F) \), if \( \delta^*(q^0, \bar{a}) \in F \).
Acceptance of Infinite Sequences

- **Büchi automaton:**

  An accepting Büchi automaton $M^B$ is a 5-tuple,

  $$M^B = (Q, \Sigma, \delta, q^0, F)$$

  where $Q$ is the finite set of states, $\Sigma$ is the input alphabet, $\delta: Q \times \Sigma \rightarrow Q$ is the transition function, $q^0$ is the start state (initial state). $F \subseteq Q$ is the set of final states (accepting states).

- **Büchi acceptance:**

  An infinite sequence $\bar{a}$ is accepted by the Büchi automaton $M^B = (Q, \Sigma, \delta, q^0, F)$, if $\forall t \exists t', t' > t . \delta^*(q^t, \bar{a}^t \ldots t') \in F$.

  *In other words, an infinite sequence is accepted if the final set is visited infinitely often.*
Equivalence Checking Problem

- Two designs are defined to be functionally equivalent if they produce identical output sequences for all valid input sequences.
Equivalence Checking Paradigms

- **Sequential Equivalence Checking**
  - Compare state machines

- **Combinational Equivalence Checking**
  - Compare combinational Boolean functions

- If a one-to-one correspondence between the registers is given, then sequential equivalence checking can be solved using combinational equivalence checking
  - This is a popular approach – very useful in practice
Combinational Equivalence Checking

- Compare o/p function
- Compare transition function
Basic Approach

- **Step-1: Register Correspondence**
  - The register correspondence is either guessed using simple heuristics or computed exactly.

- **Step-2: Functional Comparison**
  - This step involves the actual functional comparison of the individual circuits.
  - This can be done using a variety of methods, including BDDs, SAT and ATPG.
In many practical design flows, a candidate register correspondence is derived from naming conventions.

Otherwise, register correspondence can be computed automatically as a greatest fixed point (to be explained)

- The algorithm starts with one equivalence class (bucket) containing all the registers
- During each iteration:
  - A unique variable is introduced for the outputs of all registers of each bucket
  - All next state functions are computed based on these variables
  - Next the buckets are partitioned into pieces that have identical next-state functions
REGISTER CORRESPONDENCE( ) {
    put all registers $r$ into bucket[0]
    do {
        forall buckets $i$ do {
            initialize output of all registers $r \in i$ with variable $v[i]$
        }
        forall registers $r$ do {
            compute next state function $\delta[r]$ based on inputs $v$
        }
        if $\forall$ buckets $i$: $r_1$, $r_2 \in i \Leftrightarrow \delta[r_1] = \delta[r_2]$ return
        split all buckets $i$ into multiple buckets $i_j$ s.t. $r_1$, $r_2 \in i_j \Leftrightarrow \delta[r_1] = \delta[r_2]$
    }
}
To check equivalence between \( f \) and \( g \), we add the following clauses:

\[
(f \lor g), (\neg f \lor \neg g)
\]

which is the EXOR between \( f \) and \( g \). If the set of clauses is satisfiable, then we have a valuation of \( a \) and \( b \) such that \( f \) and \( g \) receive conflicting values. Otherwise (as in this case), \( f \) and \( g \) are equivalent.
Retiming and Equivalence Checking

Before retiming

After retiming
Equivalence Checking of Retimed Logic

- In case of retiming, the next-state functions are not comparable
  - However, by preserving the retime logic from the synthesis flow and applying it to make both designs comparable, the equivalence checking problem can be reduced to a combinational problem
  - Both machines are patched with pieces of the retime logic to make the interfaces comparable
Sequential Equivalence Checking

- When register correspondence cannot be found easily or it does not exist, we may compare the state machines.

- Basic approach
  - Core problem: *Partition the state space into sets of equivalent states*
  - Equivalence can be defined in terms of input/output behavior
    - Bisimulation equivalence
    - Stuttering equivalence
A or C is redundant state
If an input sequence $X$ takes a machine from a state $S_i$ to $S_j$, then $S_j$ is said to be the $X$-successor of $S_i$.

Two states $S_i$ and $S_j$ are distinguishable iff 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.

A and B are distinguishable. Consider input sequence 0.
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**.

States that are not $k$-distinguishable are called **$k$-equivalent**.

A, B are 1-distinguishable
A, C are not 2-distinguishable and hence are 2-equivalent

States $S_i$ and $S_j$ are said to be **equivalent** iff for every possible input sequence, the same output sequence is produced regardless of whether $S_i$ or $S_j$ is the initial state.

A, C are equivalent
The State Minimization Problem

Input: state machine $M$

Output: minimize $(M)$, the state machine with the fewest states that is equivalent to $M$

Two machines $M_i$ and $M_j$ are equivalent iff, for every state in $M_i$, there is a corresponding equivalent state in $M_j$ and vice versa.
The Minimization Procedure

1. Partitions states of $M$ into subsets such that all states in the same subset are 1-equivalent: $P_1$

2. Partitions states of $M$ into subsets such that all states in the same subset are 2-equivalent: $P_2$

... 

Until for some $k$, $P_{k+1} = P_k$
The Minimization Procedure

States
The Minimization Algorithm
The Minimization Algorithm
The Minimization Algorithm
The Minimization Algorithm
The Minimization Algorithm
The Minimization Algorithm
The Minimization Algorithm
The Minimization Algorithm

0/1

0/1

0/1
The Minimization Algorithm
The Minimization Algorithm

1. Let $Q$ be set of all reachable states of $M$.

2. Maintain a set $P$ of state sets:

   Initially let $P = \{ Q \}$.

   2a. Repeat until no longer possible: output split $P$.

   2b. Repeat until no longer possible: next-state split $P$.

3. When done, every state set in $P$ represents a single state of the smallest state machine equivalent to $M$. 
If there exist

- a state set $R \in P$
- two states $r_1 \in R$ and $r_2 \in R$
- an input $x \in \text{Inputs}$

such that

$$\text{output} \ (r_1, x) \neq \text{output} \ (r_2, x)$$

then

$$\text{let } R_1 = \{ \ r \in R \mid \text{output} \ (r, x) = \text{output} \ (r_1, x) \ \}\ ;$$

$$\text{let } R_2 = R \setminus R_1 ;$$

$$\text{let } P = ( P \setminus \{ R \} ) \cup \{ R_1, R_2 \} .$$
Output split
Next-state split $P$

If there exist

two state sets $R \in P$ and $R' \in P$
two states $r_1 \in R$ and $r_2 \in R$
an input $x \in \text{Inputs}$
such that

$$\text{nextState} (r_1, x) \in R' \text{ and } \text{nextState} (r_2, x) \notin R'$$

then

let $R_1 = \{ r \in R \mid \text{nextState} (r,x) \in R' \}$
let $R_2 = R \setminus R_1$
let $P = (P \setminus \{ R \}) \cup \{ R_1, R_2 \}$. 
Next-state split

\[ R \]
\[ \text{r1} \]
\[ \text{r2} \]
\[ \text{x/y} \]

\[ R' \]
\[ \text{x/y} \]
Next-state split

R1

\[ r1 \]

\[ x / y \]

R1

\[ r2 \]

\[ R2 \]

\[ x / y \]

R’
Example

\[ Q = \{ a, b, c, d, e, f, g \} \]
\[ P = \{ \{ a, b, c, d, e, f, g \} \} \]
P = \{ \{ a, b, c \}, \{ d, e, f, g \} \}

Output split
$P = \{ \{ a \}, \{ b, c \}, \{ d, e, f, g \} \}$

Next-state split
Next-state split

\[ P = \{ \{ a \}, \{ b, c \}, \{ d, e \}, \{ f, g \} \} \]
Minimal bisimilar state machine

\{ a \} \rightarrow \{ b, c \} \rightarrow \{ d, e \} \rightarrow \{ f, g \}

\rightarrow 0/1 \rightarrow 1/0 \rightarrow 0/1 \rightarrow 0/0 \rightarrow 1/1

\rightarrow 1/1 \rightarrow 0/0 \rightarrow 0/0 \rightarrow
4 instead of 7 states
How to check if $M_1$ and $M_2$ are equivalent

1. Minimize $M_1$ and call the result $N_1$
2. Minimize $M_2$ and call the result $N_2$
3. Check if the states of $N_1$ can be renamed so that $N_1$ and $N_2$ are identical