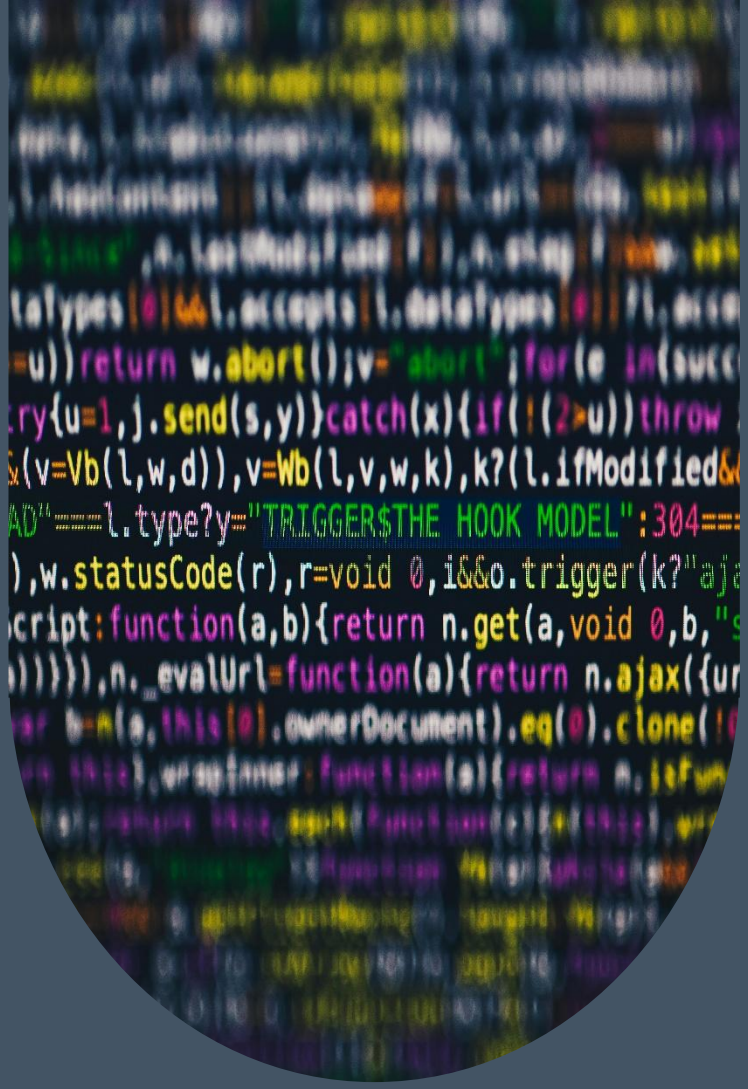
Games Engineering II

Finite State Machines

Dr. Omer Ali
omer.ali@setu.ie



SETU
Ollscoil Teicneolaíochta an Oirdheiscirt
South East Technological University

setu.ie
INSPIRING FUTURES

Lec 03 – Recap

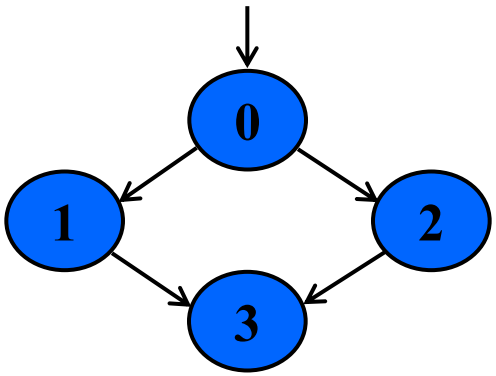
Loops in a Graph

Loops in Graphs

- If a graph contains a loop, it has an infinite number of paths
- Thus, CPC is not feasible
- SPC is not satisfactory because the results are subjective and vary with the tester

Simple Paths and Prime Paths

- Simple Path : *A path from node n_i to n_j is simple if no node appears more than once, except possibly the first and last nodes are the same*
- No internal loops
- A loop is a simple path
- Prime Path : *A simple path that **does not** appear as a proper subpath of any other simple path*



Simple Paths : [0, 1, 3, 0], [0, 2, 3, 0], [1, 3, 0, 1], [2, 3, 0, 2], [3, 0, 1, 3], [3, 0, 2, 3], [1, 3, 0, 2], [2, 3, 0, 1], [0, 1, 3], [0, 2, 3], [1, 3, 0], [2, 3, 0], [3, 0, 1], [3, 0, 2], [0, 1], [0, 2], [1, 3], [2, 3], [3, 0], [0], [1], [2], [3]

Prime Paths : [0, 1, 3, 0], [0, 2, 3, 0], [1, 3, 0, 1], [2, 3, 0, 2], [3, 0, 1, 3], [3, 0, 2, 3], [1, 3, 0, 2], [2, 3, 0, 1]

Prime Path Coverage

A simple, elegant and finite criterion that requires loops to be executed as well as skipped

Prime Path Coverage (PPC) : TR contains each prime path in G.

- **Will tour all paths of length 0, 1, …**
- **That is, it *subsumes node* and *edge* coverage**
- **PPC does NOT subsume EPC**
  - **If a node *n* has an edge to itself, EPC will require [*n, n, m*]**
  - **[*n, n, m*] is not prime**

Round Trips

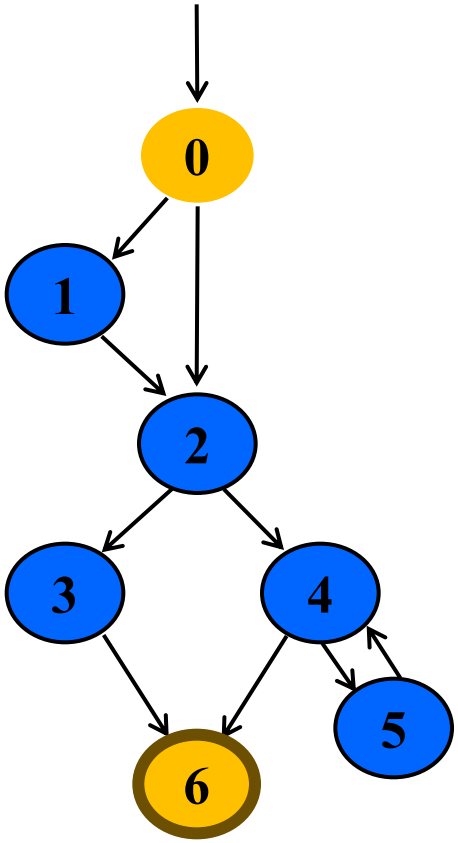
Round-Trip Path : *A prime path that starts and ends at the same node*

Simple Round Trip Coverage (SRTC) : TR contains at least one round-trip path for each reachable node in G that begins and ends a round-trip path.

Complete Round Trip Coverage (CRTC) : TR contains all round-trip paths for each reachable node in G.

Prime Path Example

- The previous example has ***38*** simple paths
- Only ***nine*** *prime paths*



Prime Paths

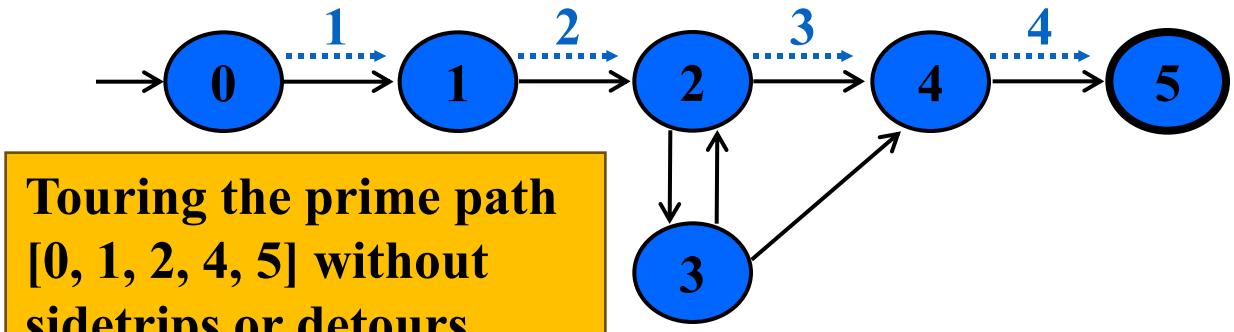
[0, 1, 2, 3, 6]
[0, 1, 2, 4, 5]
[0, 1, 2, 4, 6]
[0, 2, 3, 6]
[0, 2, 4, 5]
[0, 2, 4, 6]
[5, 4, 6]
[4, 5, 4]
[5, 4, 5]

Touring, Sidetrips and Detours

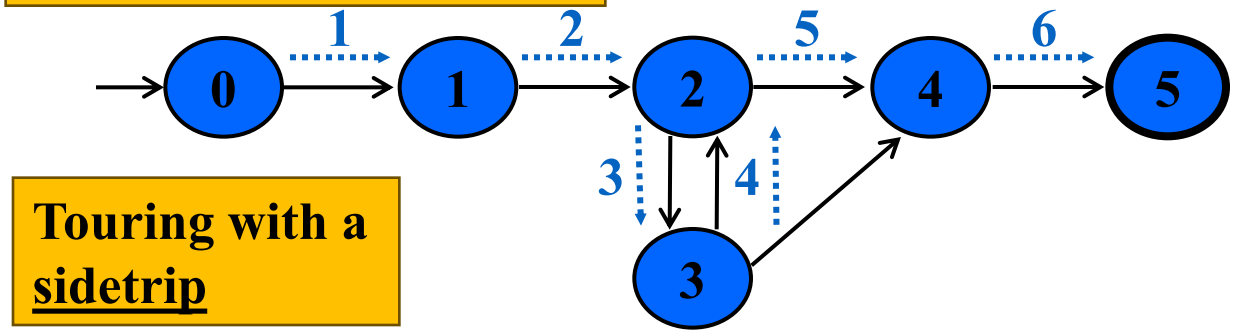
Prime paths do not have internal loops … test paths might

- **Tour** : ***A test path p tours subpath q if q is a subpath of p***
- **Tour With Sidetrips** : ***A test path p tours subpath q with sidetrips iff every edge in q is also in p in the same order***
  - **The tour can include a sidetrip, as long as it comes back to the same node**
- **Tour With Detours** : ***A test path p tours subpath q with detours iff every node in q is also in p in the same order***
  - **The tour can include a detour from node *ni*, as long as it comes back to the prime path at a successor of *ni***

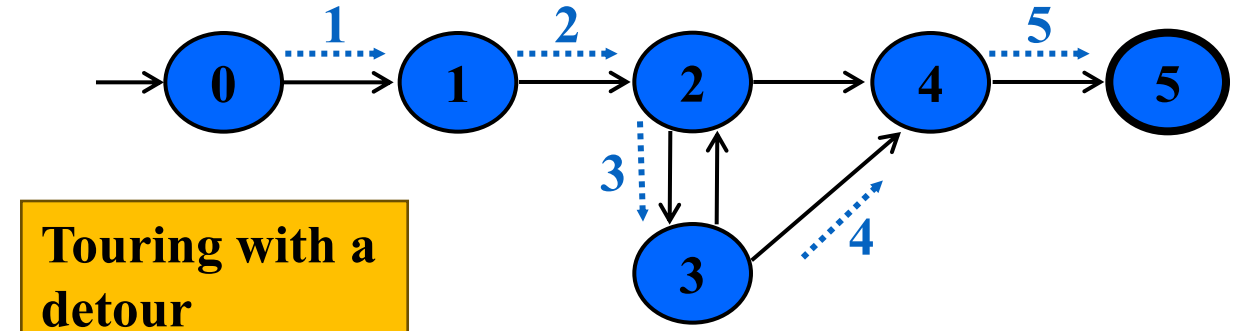
Sidetrips and Detours Example



Touring the prime path [0, 1, 2, 4, 5] without <u>sidetrips or detours</u>



Touring with a <u>sidetrip</u>



Touring with a <u>detour</u>

Infeasible Test Requirements

- An infeasible test requirement cannot be satisfied
- Unreachable statement (dead code)
- A subpath that can ***only be executed*** if a contradiction occurs ($X > 0$ and $X < 0$)

- **Most test criteria have some infeasible test requirements**
- **It is usually undecidable whether all test requirements are feasible**
- **When sidetrips are not allowed, many structural criteria have more infeasible test requirements**
- **However, always allowing sidetrips weakens the test criteria**

Practical recommendation – Best Effort Touring

– Satisfy as many test requirements as possible without sidetrips

– Allow sidetrips to try to satisfy unsatisfied test requirements

Finite State Machines

FSMs

- Logic or decision boundaries are called **States** (Nodes)
- Actions or triggers result in **transactions** (edges)
- FSMs can be deterministic (only one output per action)
- FSMs can also be non-deterministic (more that one output based on actions)

Definition of a State Machine

- All programmable logic designs can be specified in Boolean form. However some designs are easier to conceptualize and implement using non-Boolean models. The State Machine model is one such model.

Definition of a State Machine

- A state machine represents a system as a set of states, the transitions between them, along with the associated inputs and outputs.
- So, a state machine is a particular conceptualization of a particular sequential circuit. State machines can be used for many other things beyond logic design and computer architecture.

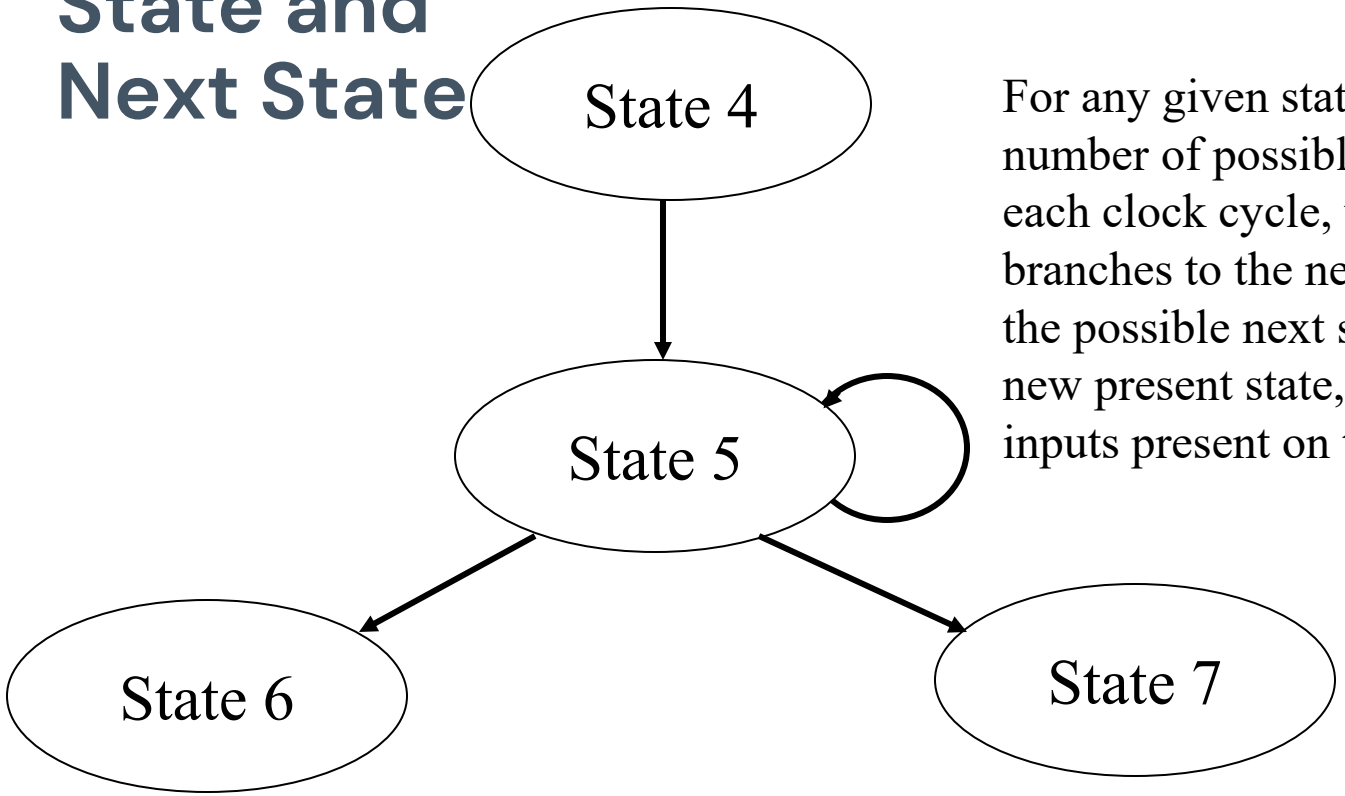
Finite State Machines

- Any Circuit with Memory Is a Finite State Machine
- Even computers can be viewed as huge FSMs
- Design of FSMs Involves
- Defining states
- Defining transitions between states
- Optimization / minimization
- Above Approach Is Practical for Small FSMs Only

State Machines: Definition of Terms

- State Diagram
  - Illustrates the form and function of a state machine. Usually drawn as a bubble-and-arrow diagram.
- State
  - A uniquely identifiable set of values measured at various points in a digital system.
- Next State
  - The state to which the state machine makes the next transition, determined by the inputs present when the device is clocked.
- Branch
  - A change from present state to next state.
- Mealy Machine
  - A state machine that determines its outputs from the present state and from the inputs.
- Moore Machine
  - A state machine that determines its outputs from the present state only.

Present State and Next State



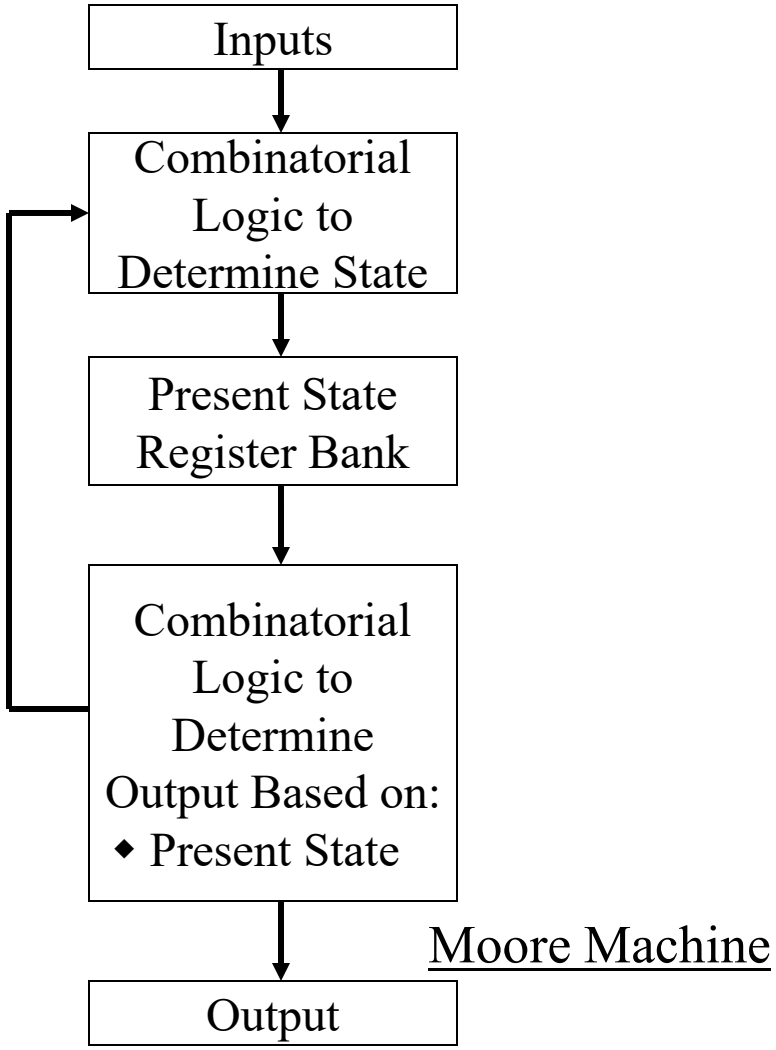
For any given state, there is a finite number of possible next states. On each clock cycle, the state machine branches to the next state. One of the possible next states becomes the new present state, depending on the inputs present on the clock cycle.

- On a well-drawn state diagram, all possible transitions will be visible, including loops back to the same state. From this diagram it can be deduced that if the present state is State 5, then the previous state was either State 4 or 5 and the next state must be either 5, 6, or 7.

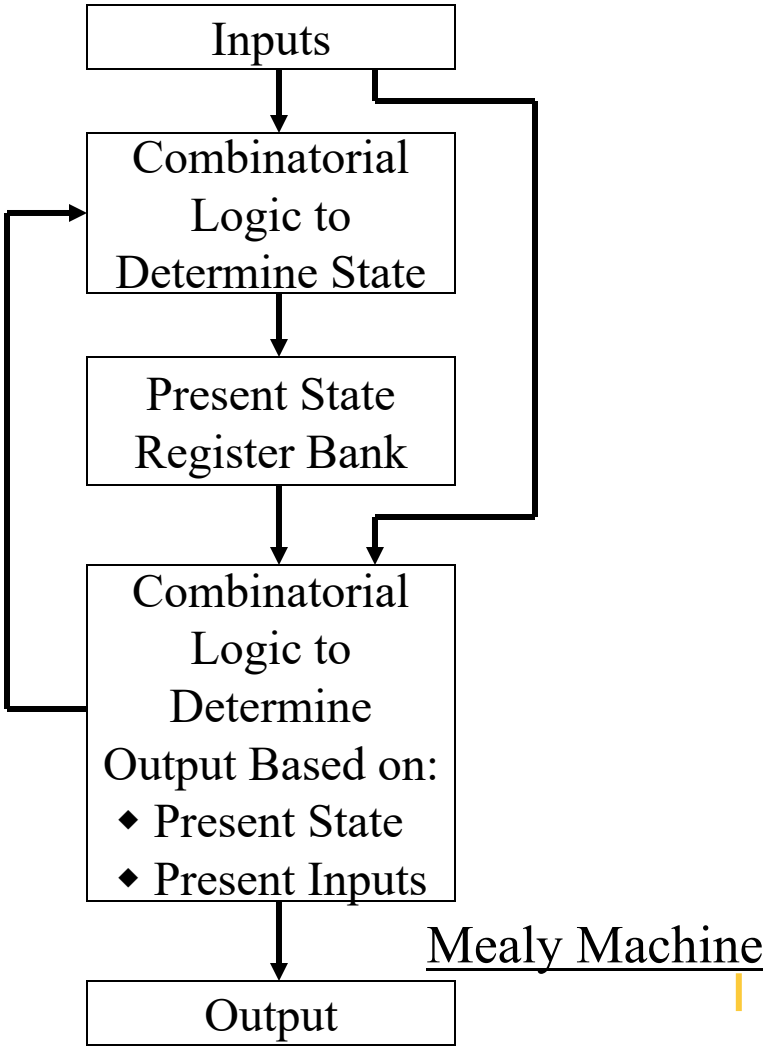
Moore and Mealy Machines

- Both these machine types follow the basic characteristics of state machines, but differ in the way that outputs are produced.
- **Moore Machine**:
- Outputs are independent of the inputs, ie outputs are effectively produced from within the state of the state machine.
- **Mealy Machine**:
- Outputs can be determined by the present state alone, or by the present state and the present inputs, ie outputs are produced as the machine makes a transition from one state to another.

Machine Models



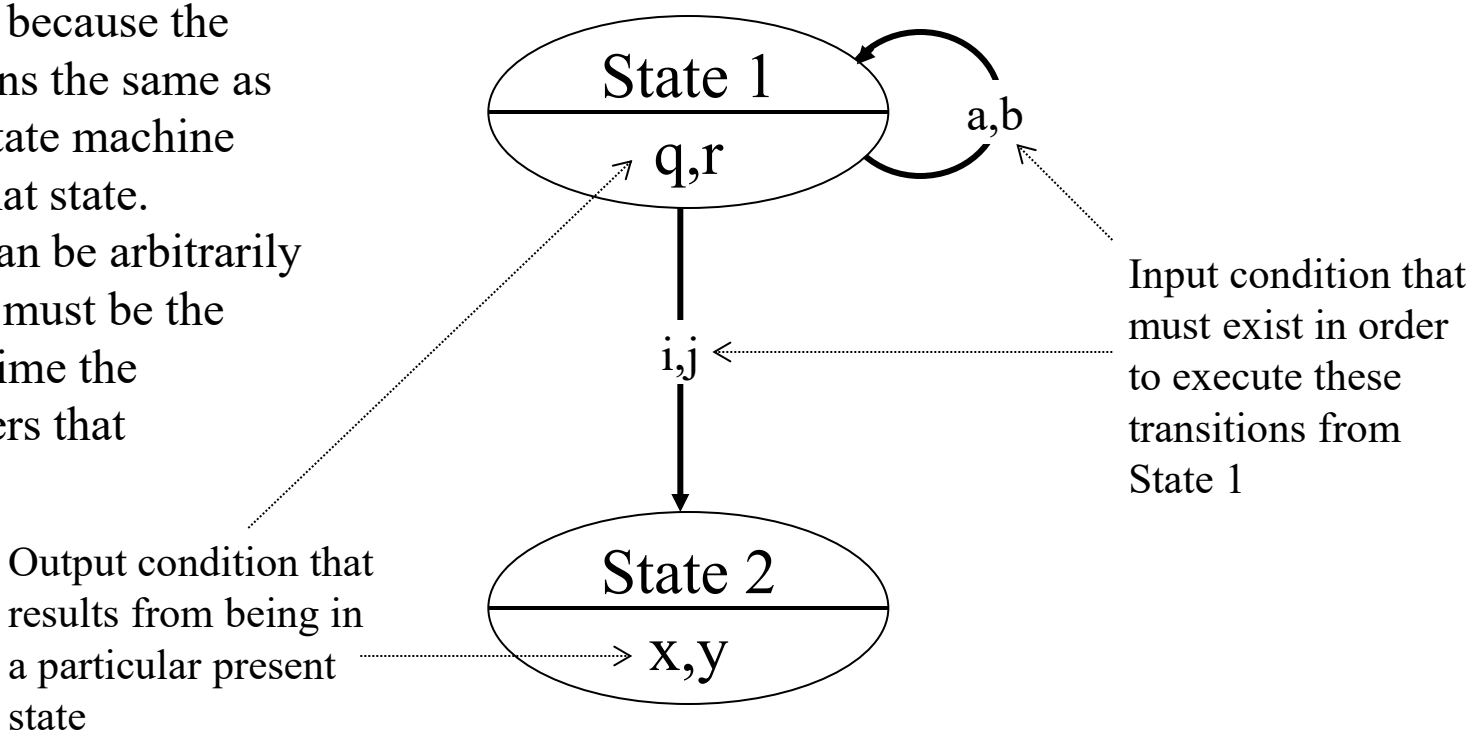
Moore Machine



Mealy Machine

Moore Machine Diagrams

The Moore State Machine output is shown inside the state bubble, because the output remains the same as long as the state machine remains in that state.
The output can be arbitrarily complex but must be the same every time the machine enters that state.

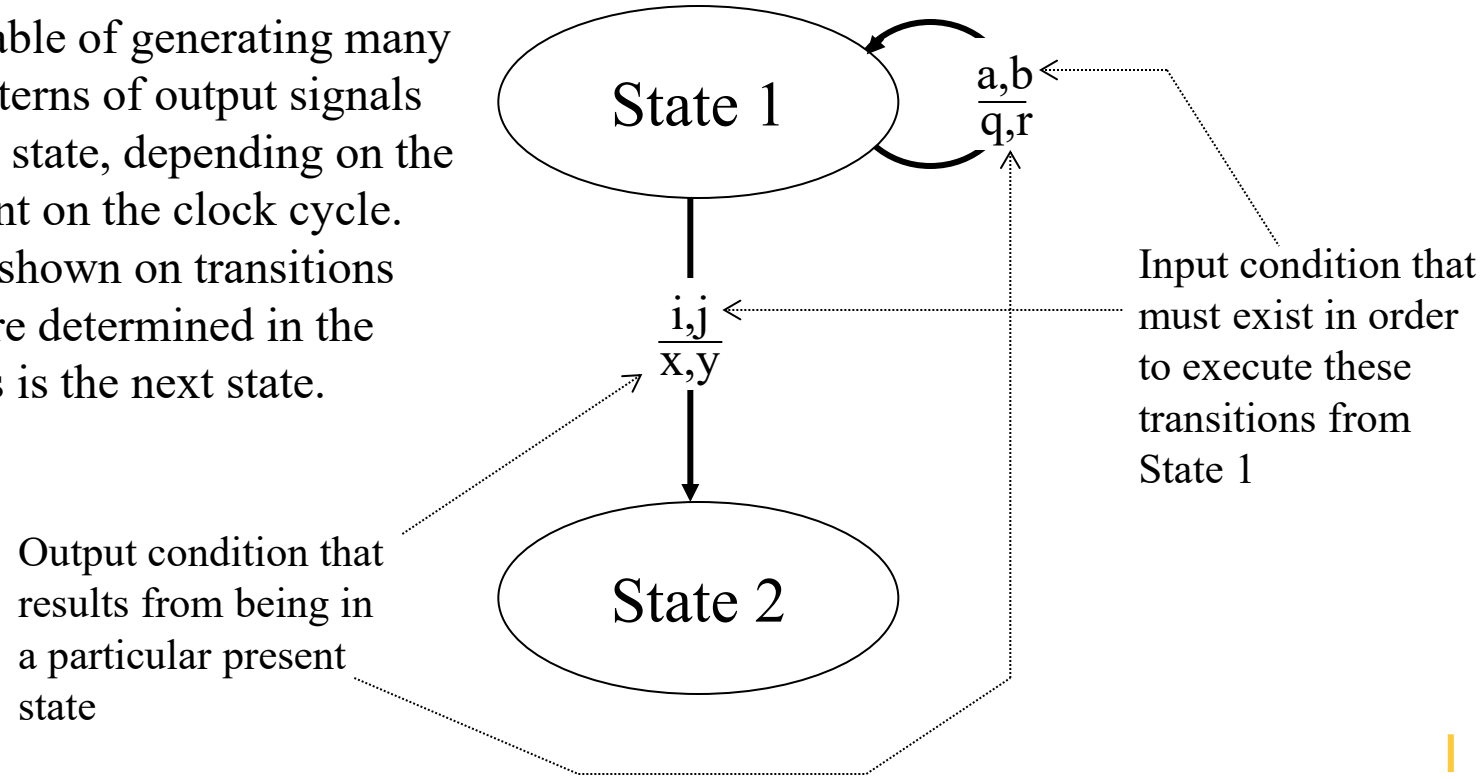


Mealy Machine Diagrams

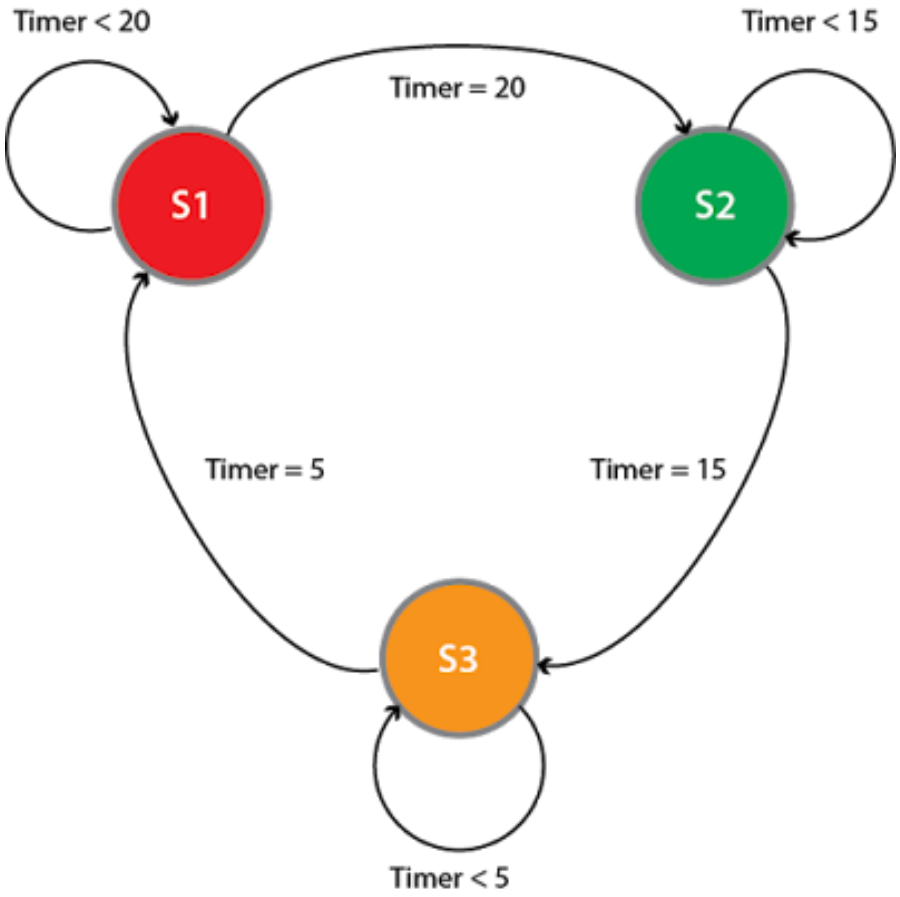
The Mealy State Machine generates outputs based on:

- The Present State, and
- The Inputs to the M/c.

So, it is capable of generating many different patterns of output signals for the same state, depending on the inputs present on the clock cycle. Outputs are shown on transitions since they are determined in the same way as is the next state.



Traffic Light Example



Traffic Lights Example

Encoding Traffic Lights Sequence

Current State	Transition/Action	Next State	Encoding (2-bit)	Encoding (3-bit)
0:0 R	00	R	00	000
0:1 R	01	G	01	001
0:0 G	00	G	00	010
0:1G	01	Y	10	011
0:0Y	00	Y	00	100
0:1Y	01	R	11	101

Traffic Lights Example

```cpp
#include <cassert>
#include <iostream>
enum LightState { Red, Green, Yellow };

class TrafficLight {
public:
TrafficLight() : state(Red) {} // Initial state is Red
void timerTick() {
// Advance to next state in cycle
switch(state) {
case Red: state = Green; break;
case Green: state = Yellow; break;
case Yellow: state = Red; break;
}
}
LightState getState() const { return state; }
private:
LightState state;
};

int main() {
TrafficLight tl;
assert(tl.getState() == Red); // Initially Red
tl.timerTick();
assert(tl.getState() == Green); // Red -> Green on timer
tl.timerTick();
assert(tl.getState() == Yellow); // Green -> Yellow on
    timer
tl.timerTick();
assert(tl.getState() == Red); // Yellow -> Red on timer (
    cycle complete)
std::cout << "TrafficLight FSM test passed\n";
}
```

Questions & Feedback?

Please feel free to email for your queries.

Course materials will be uploaded to Blackboard after every lecture.

setu.ie

INSPIRING FUTURES

Ollscoil Teicneolaíochta an Oirdheiscirt

South East Technological University