ENGG1100
Introduction to Engineering Design

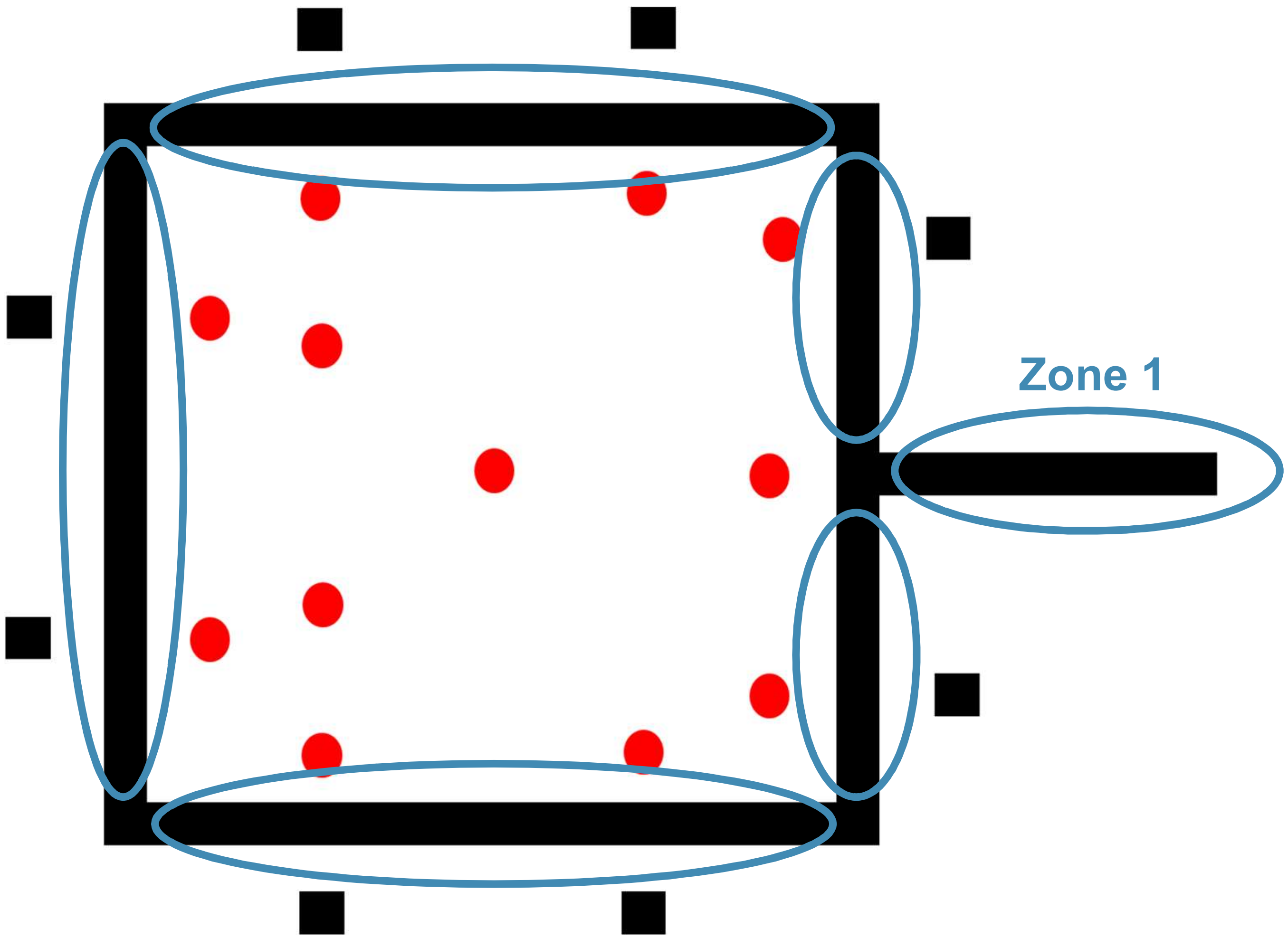
Finite-State Machine

Dr. Marco Ho
Department of Information Engineering

Why Finite-State Machine (FSM)?

- It can perform **predetermined** sequence of **actions**, depending on a sequence of **events** that takes place.
- Suitable to be used in **event-driven** systems.
- An ideal tool for you to learn how to **divide-and-conquer** a complex problem.
  - A **solution** can first be developed for a **part of problem**. When the **same part of problem** occurs, the **same solution** can be reused.
  - Then the problem can be divided into multiple small problems, and small solutions are developed to conquer them.
  - Eventually, when the system includes **all the states** (which is a **finite** number) that the problem has, the **full solution** is developed.

Example: Your Project



What is Finite-State Machine?

- A mathematical **model** of computation
- A set of **states** and how to get from one state to another
- An ideal representation of a computer/machine
  - It can be in **exactly one of the states** at a time
  - A state describes the computer at any given point
  - A large but **finite** number of states
- It represents legal steps of a process
  - Valid **inputs**
  - Valid **outputs**
  - Some **computation**

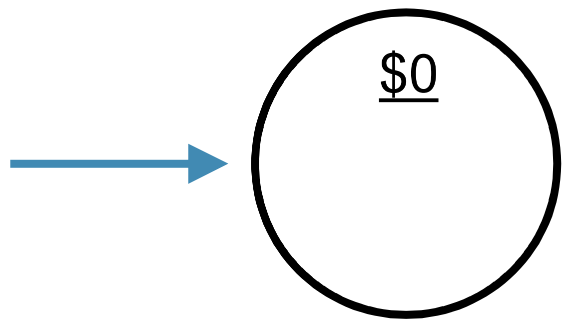
Vending Machine

- A can of coke costs $6.
- The machine only accepts $1, $2 and $5 coins.

- Consider the state of the vending machine.
  - Needs to include **all possible cases**
  - What is the **initial state**?

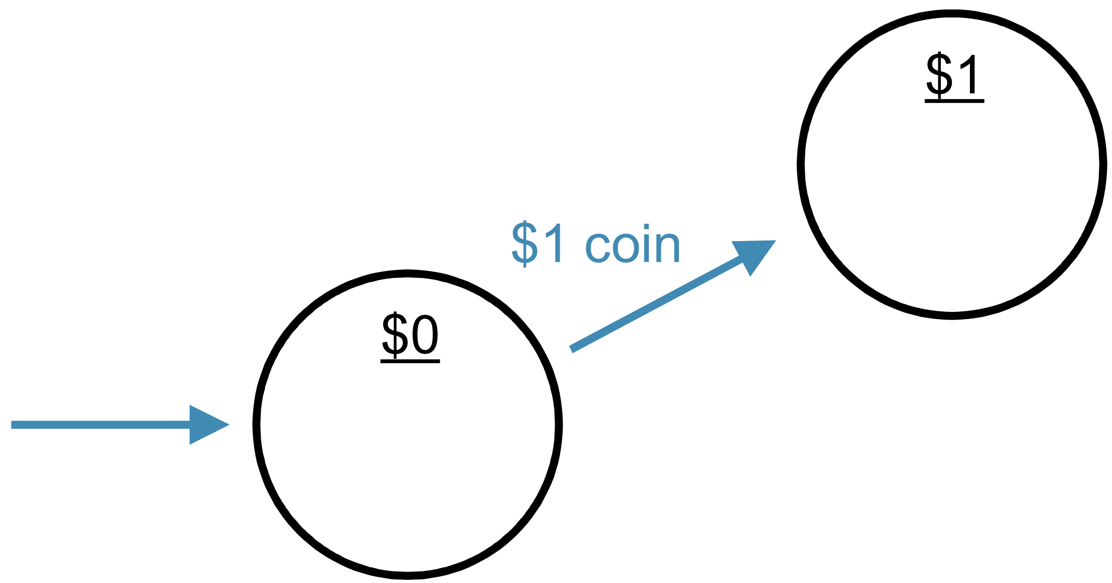
Vending Machine Initial State

- Start with $0 deposited
  - Called **initial state** of the machine
  - A **circle** represents the state
  - **State name** is **underlined** and inside the circle
  - An **arrow** indicates it is an initial state



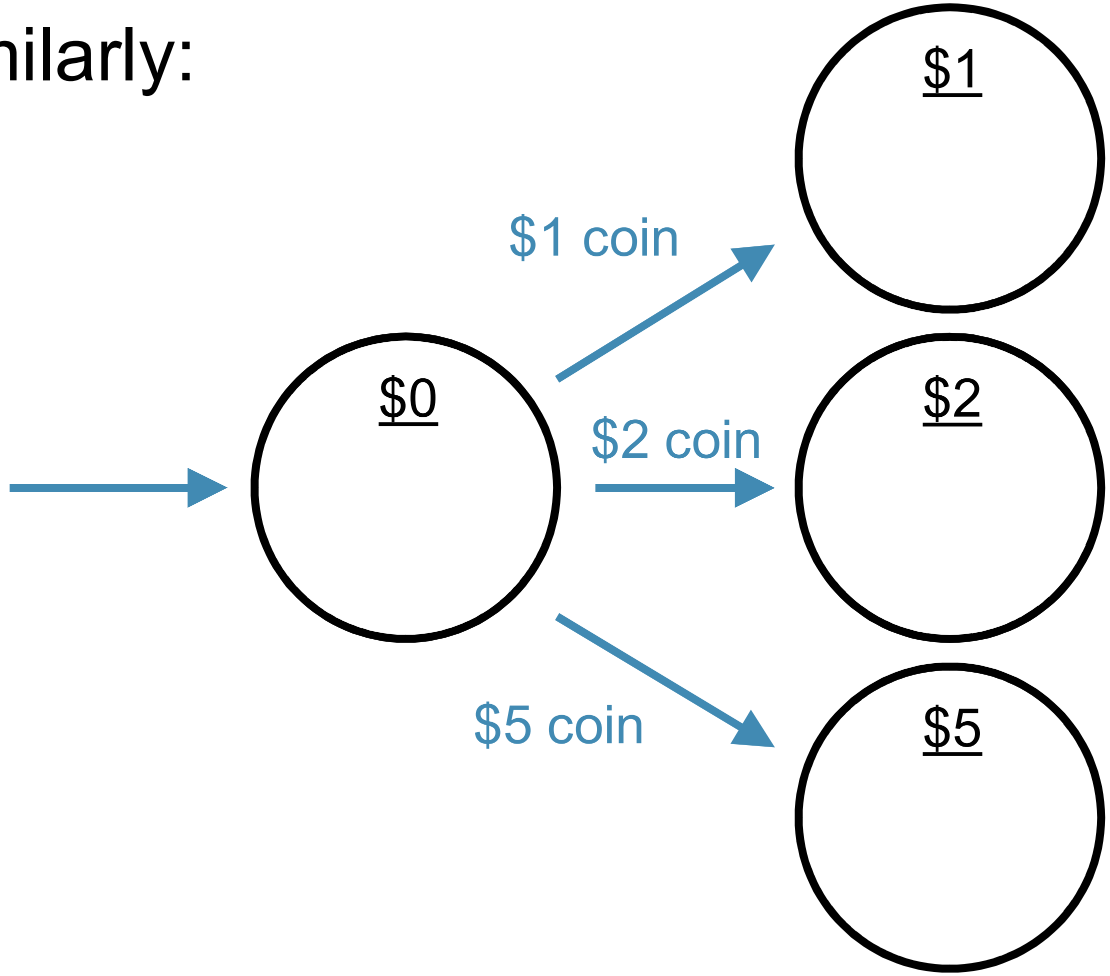
Vending Machine Next State

- If $1 is deposited
  - Add an **arrow** for the **transition**
  - Label it with the **input condition**



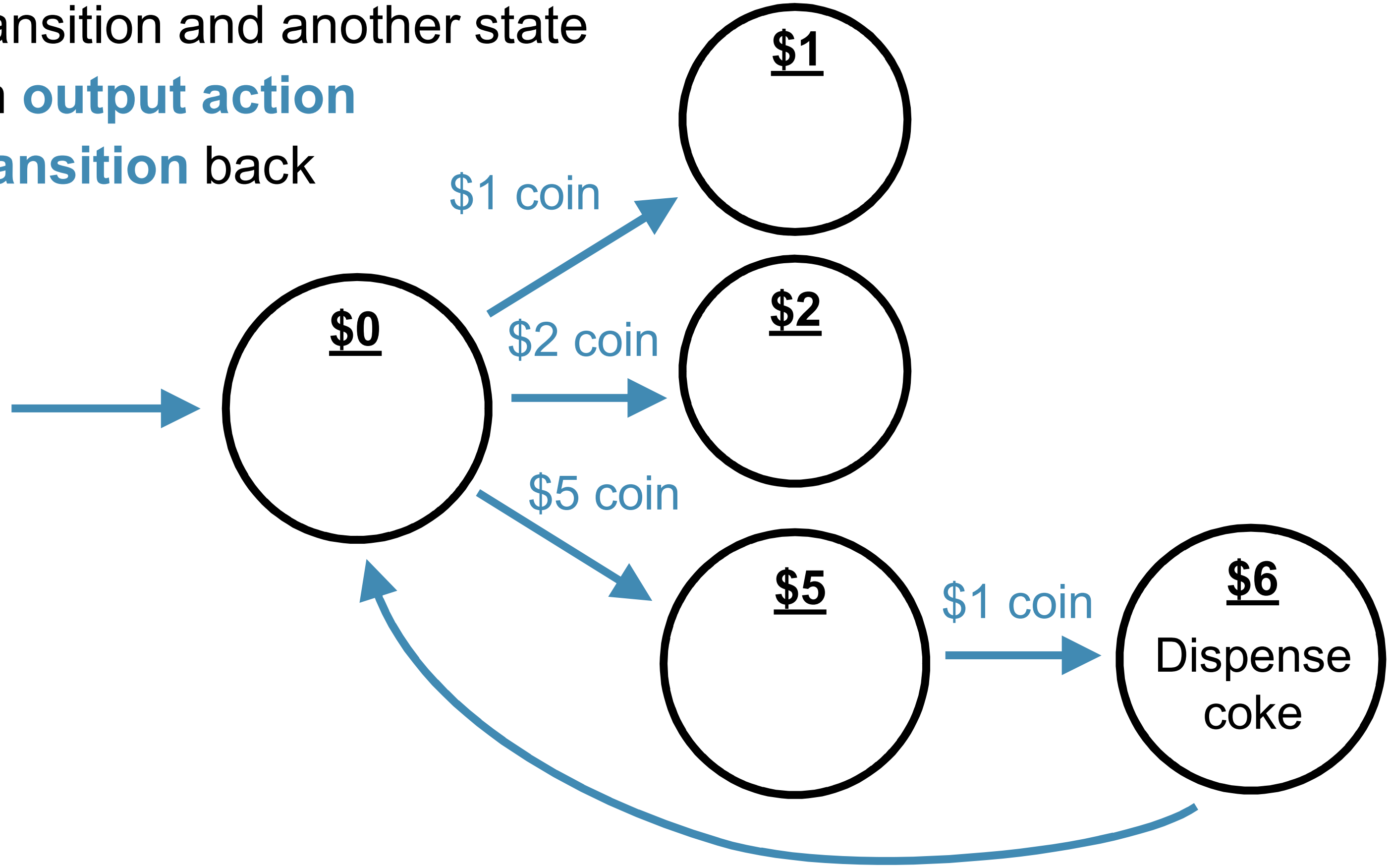
Vending Machine States

- Similarly:

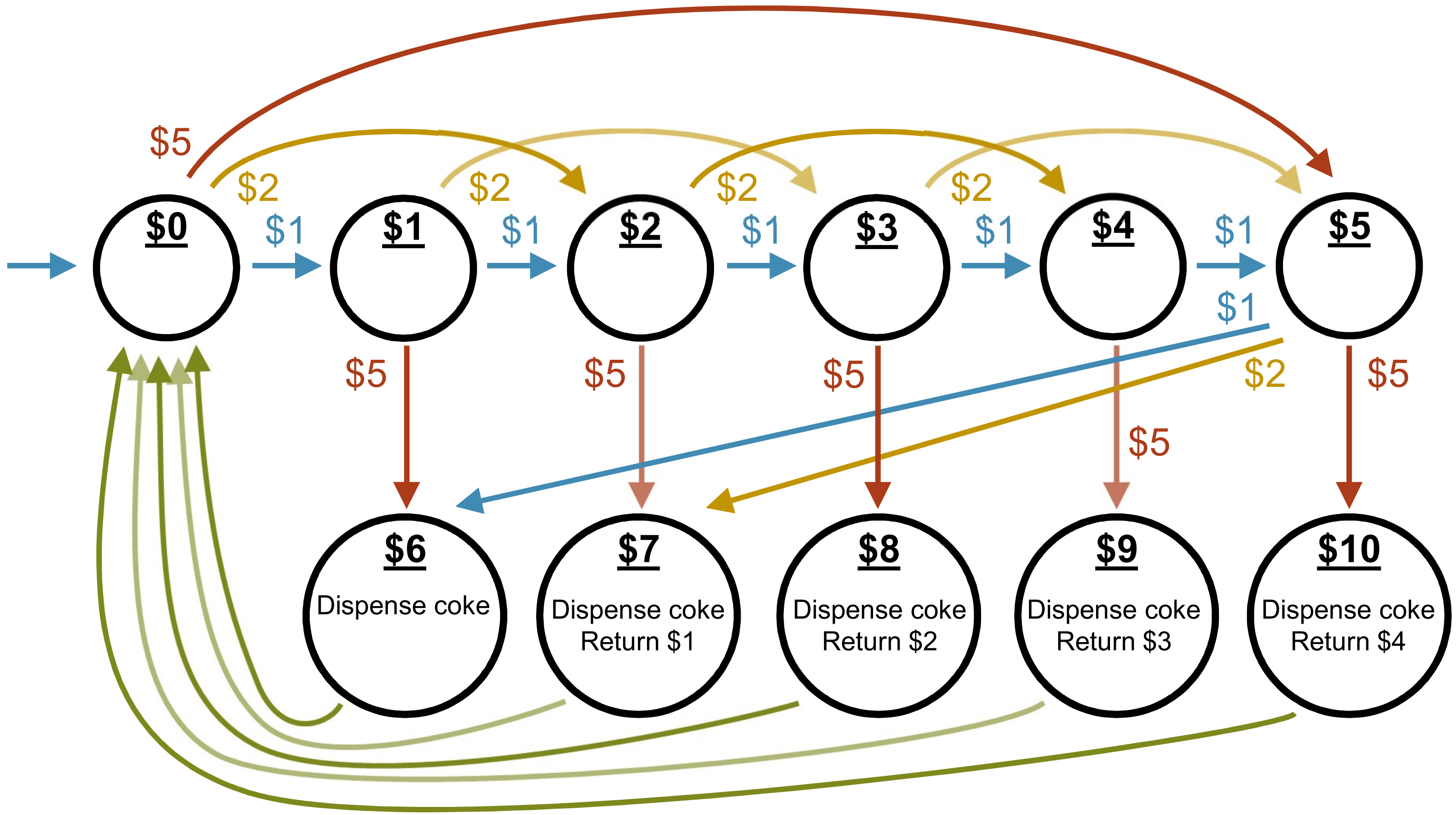


Vending Machine More States

- If \$5 is already deposited, what happens if an extra \$1 is deposited?
  - Add transition and another state
  - Add an **output action**
  - Add **transition** back

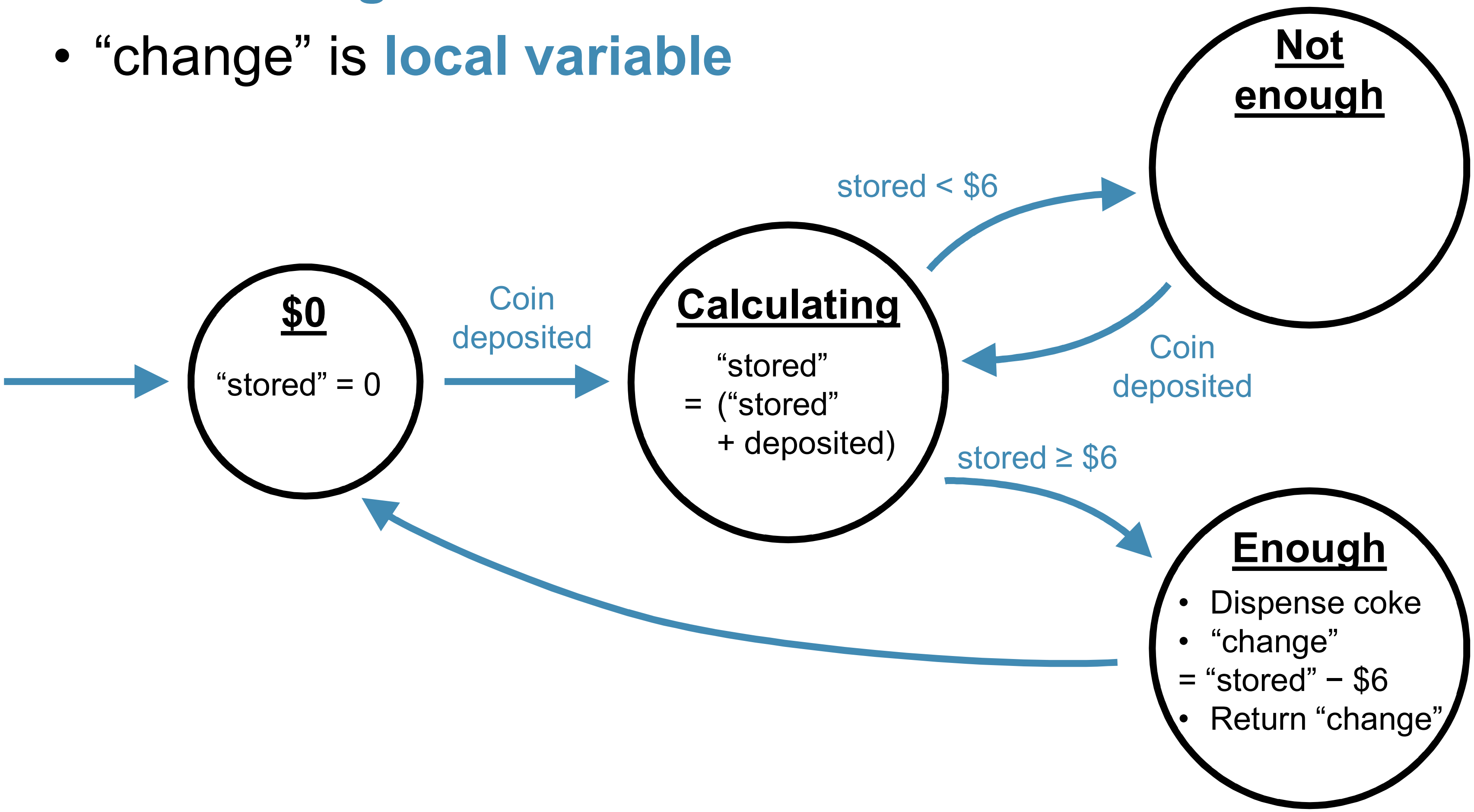


Vending Machine State Diagram



Alternative State Diagram

- “stored” is **global variable**
- “change” is **local variable**

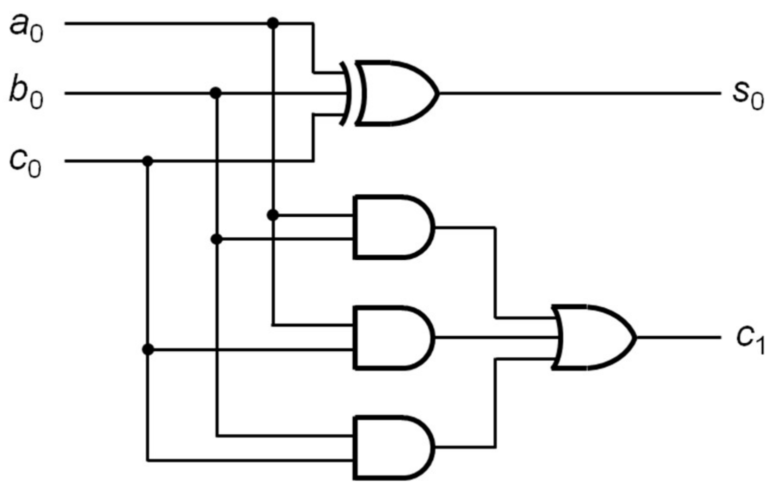


FSM Design

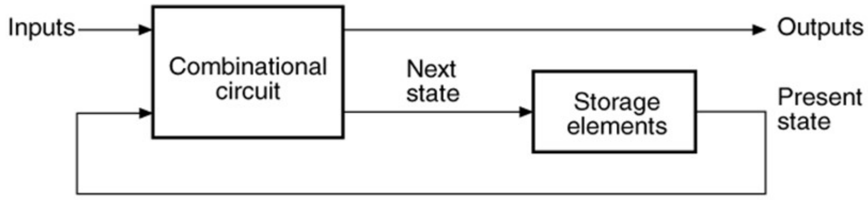
- There are **multiple solutions** to a problem (i.e., no fixed answer).
- The **implementation** will depend on your FSM **design**.
- Trade off between
  - Complexity
  - Regularity
  - Number of states
  - Number of variables

Types of Logic Circuits

- **Combinational Logic**
  - Memoryless
  - Outputs determined by current values of inputs

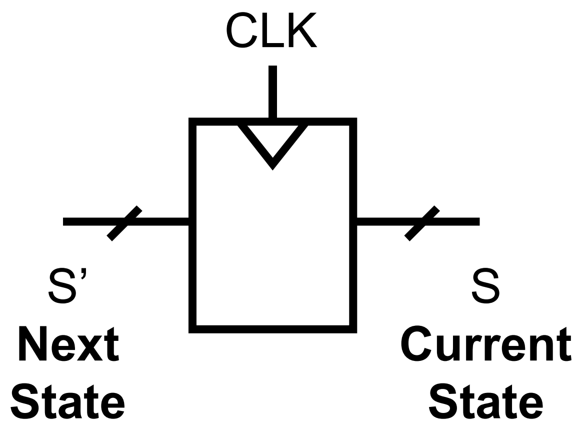


- **Sequential Logic**
  - Has memory
  - Outputs determined by previous and current values of inputs



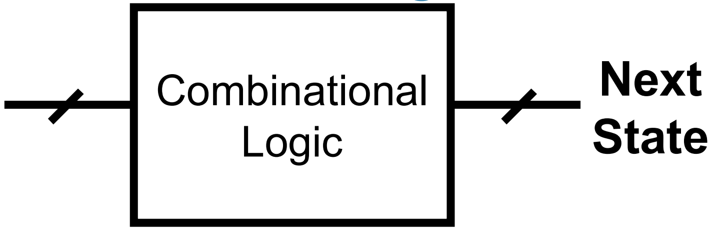
FSM Structures

- Consists of:
  - **State register**
    - Stores current state
    - Loads next state at clock edge

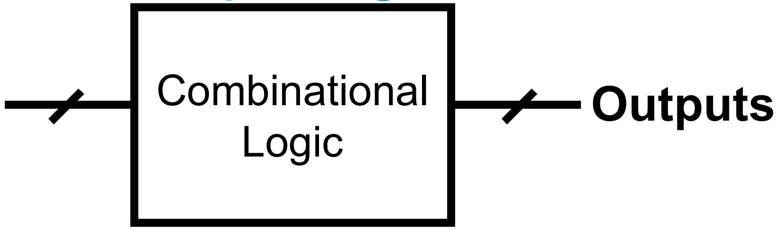


  - **Combinational logic**
    - Computes the next state
    - Computes the outputs

Next State Logic

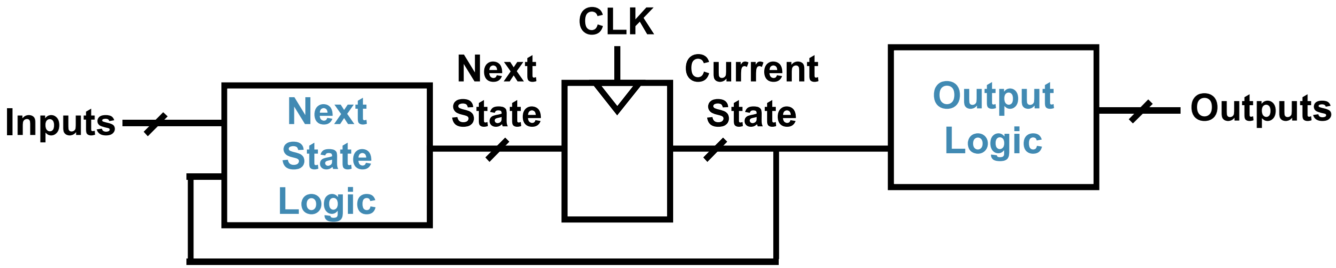


Output Logic

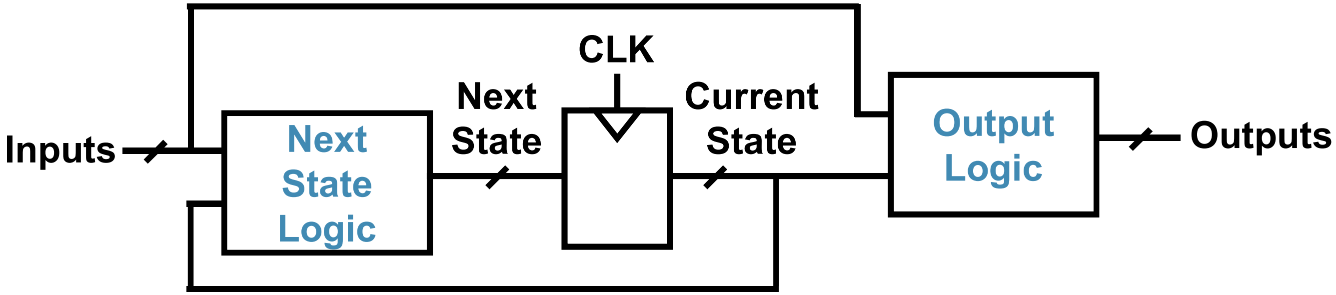


Moore and Mealy FSMs

- Next state determined by current state and inputs.
- Two types of finite state machines differ in output logic:
  - **Moore FSM**: outputs depend only on current state.



  - **Mealy FSM**: outputs depend on current state and inputs.

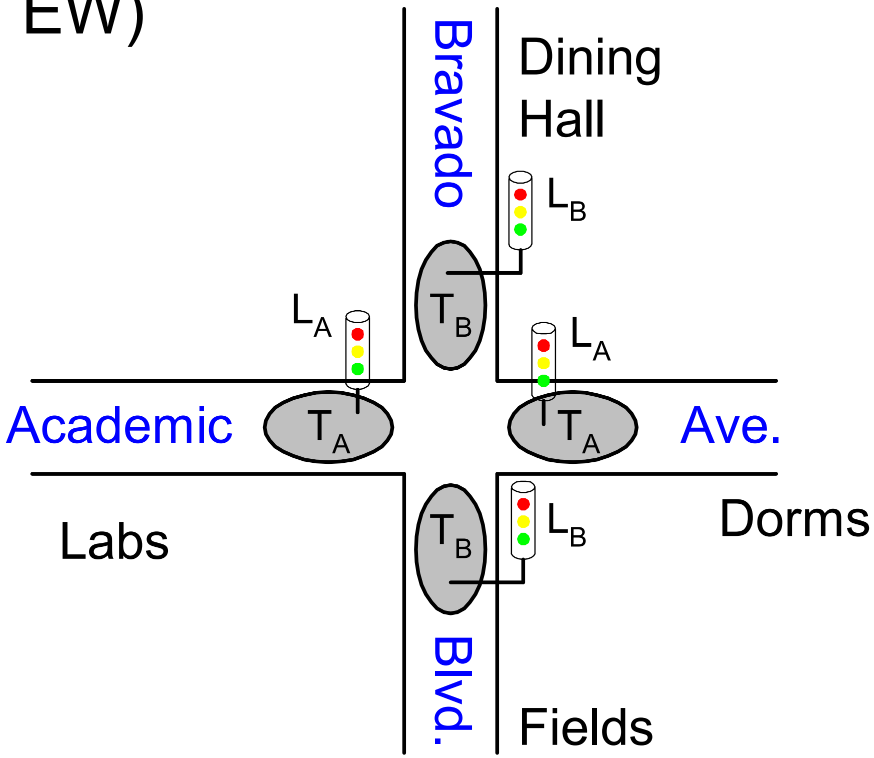


FSM Design Procedure

- Identify inputs and outputs.
- Sketch state transition diagram.
  - Each **state** is denoted by a **circle**.
  - Each **arrow** (between two circles) denotes a **transition** of the sequential circuit (a row in state table).
  - Label the arrow with the **transition condition**.
- Write state transition table consisting of all possible binary combinations of **present states**, **inputs**, **next states** and **outputs**.

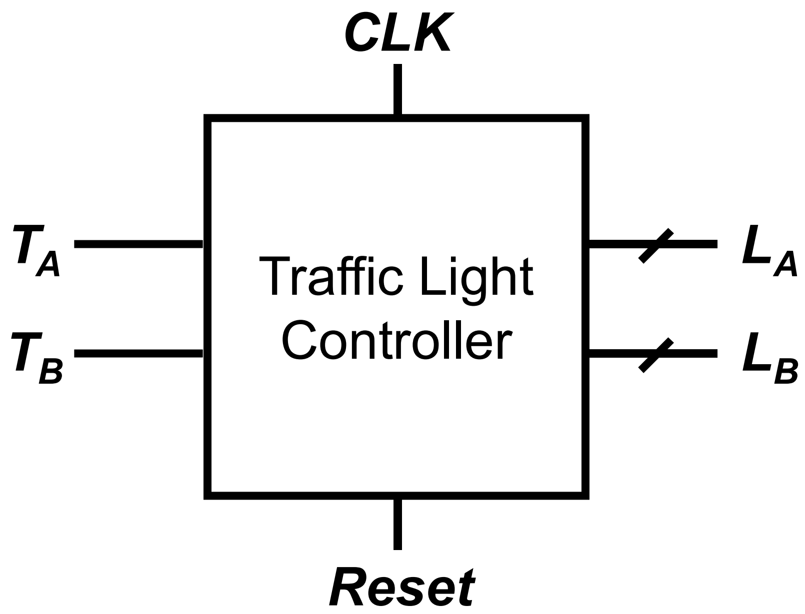
FSM Example (1)

- **Traffic light controller** (in US)
  - **Traffic sensors**: T_A, T_B (TRUE when there is traffic)
  - **Lights**: L_A, L_B
- Two sets of lights (NS and EW)
  - When EW (L_A) is green, NS (L_B) is red.
  - EW light (L_A) will stay green as long as there is EW traffic (T_A).



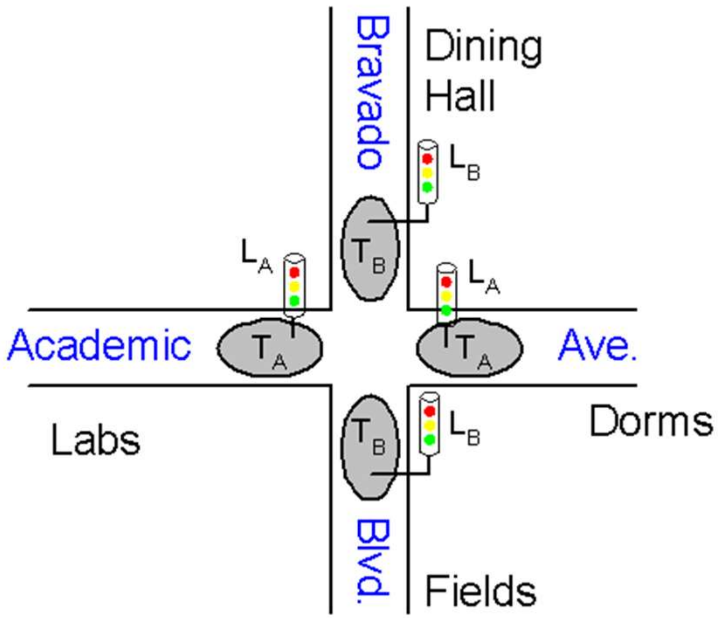
FSM Example (2)

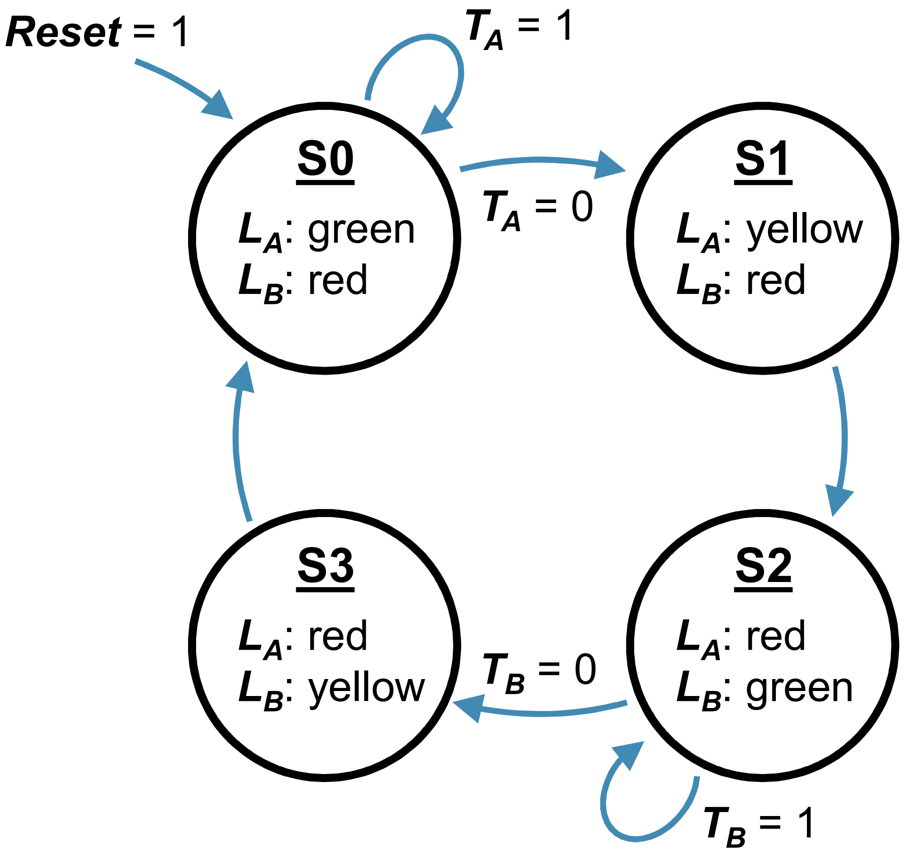
- **Inputs**: CLK, $Reset$, T_A, T_B
- **Outputs**: L_A, L_B



FSM Example (3)

- **State Transition Diagram**
  - **Moore FSM**: outputs labeled in each state
  - **States**: Circles
  - **Transitions**: Arrows



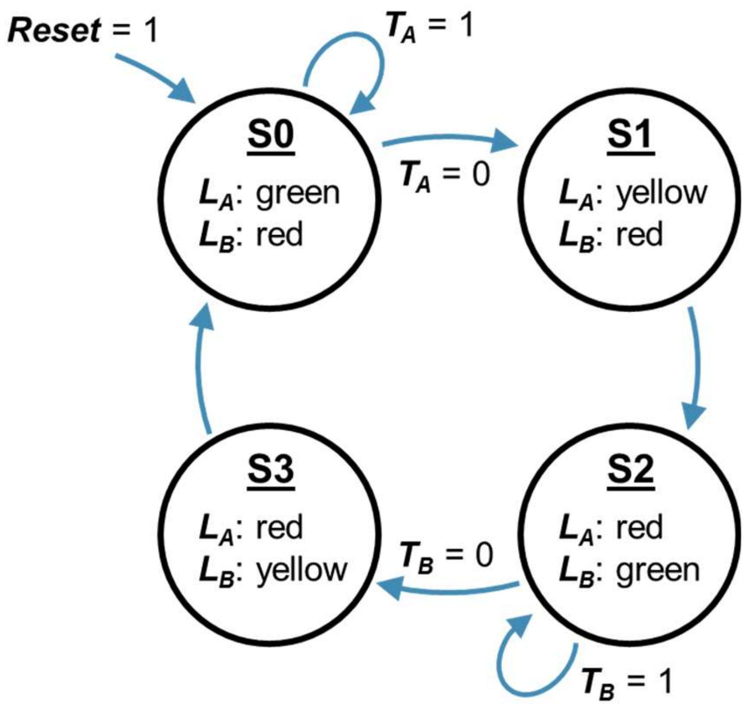


FSM Example (4)

- **State Transition Table**

Current State	Inputs		Next State
S	T_A	T_B	S^+
S0	0	X	S1
S0	1	X	S0
S1	X	X	S2
S2	X	0	S3
S2	X	1	S2
S3	X	X	S0

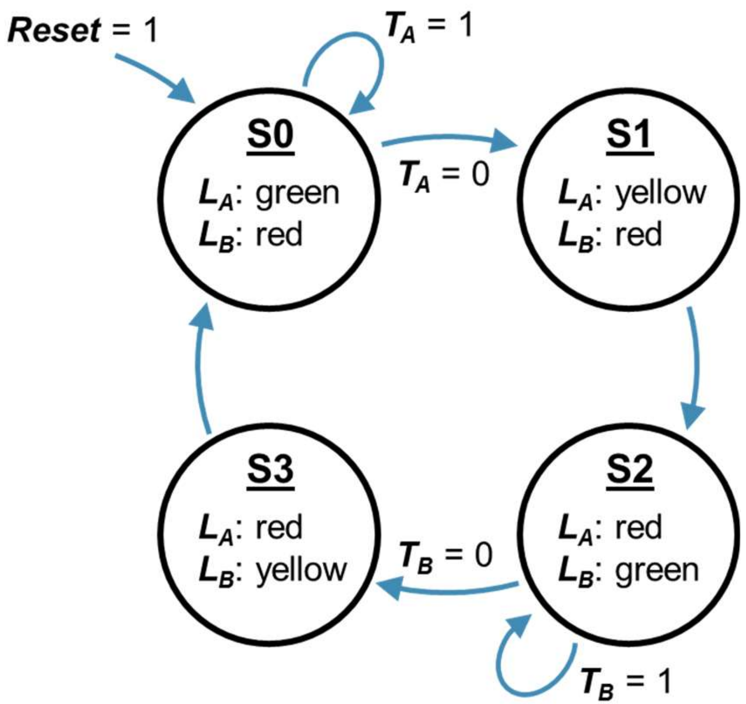
X = don't care



FSM Example (5)

- **Output Table**

Current State	Outputs	
S	L_A	L_B
S0	green	red
S1	yellow	red
S2	red	green
S3	red	yellow

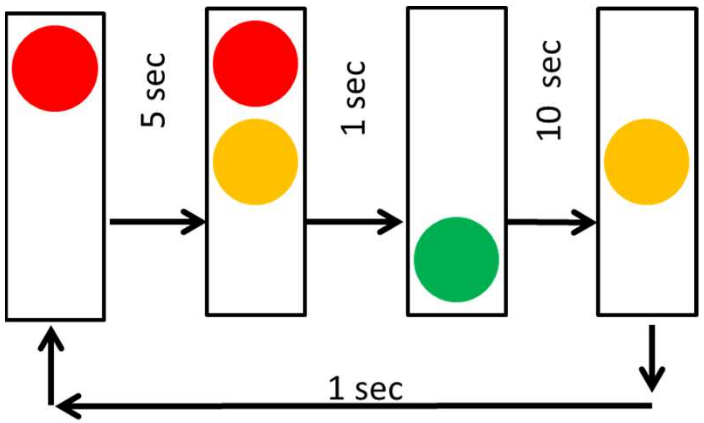


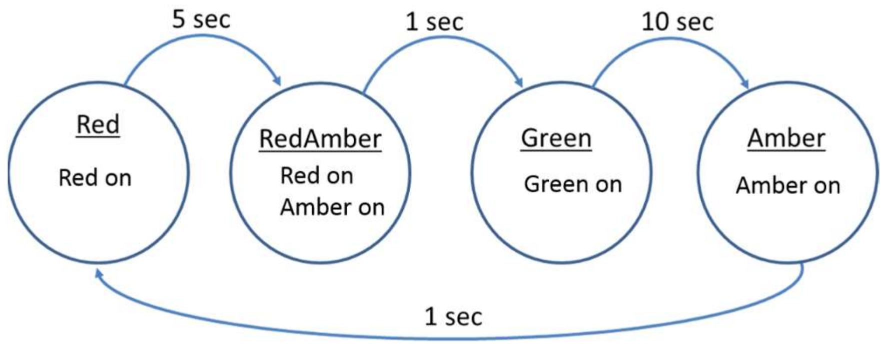
Implementation on Arduino

- From Lab 5 on, we use **Arduino** to implement the FSM.
- The **framework** of the FSM is provided for you.
- All you need to do is
  - Design your **states**, **transition**, **inputs** and **outputs**.
  - Construct your **state diagram**.
  - **Copy**, **paste** and **edit** the code.

Traffic Light Revisited

- **Traffic light controller** again (in HK)
  - In Lab 5, we start with "**time-driven**" pattern.

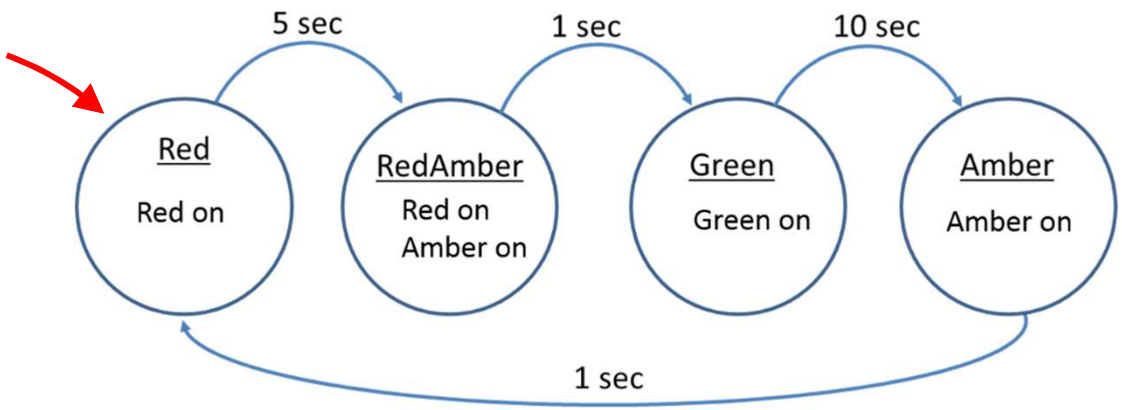




Traffic Light Initial State

- In **setup()**, the initialization points to "**S_DRed**" state.

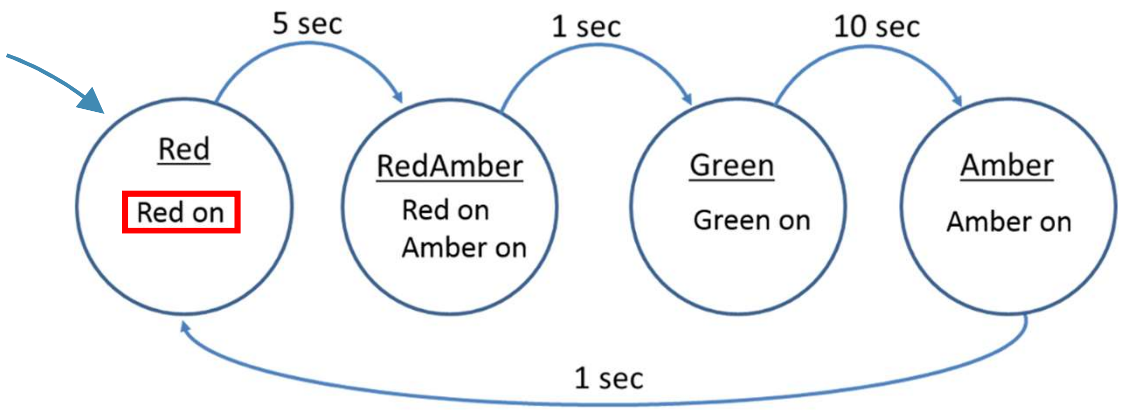
```
//===== Basically, no need to modify setup() and loop() ====
void setup()
{
  Serial.begin(115200);            //optional, for debug
  LEDDisplay.setBrightness(15); //optional
  FSM1.init(S_DRed);              // must have this line, you can change the first state of the FSM
}
```



Traffic Light Output

- Upon entering "**S_DRed**" state, turn on "**DRed**" (red light).

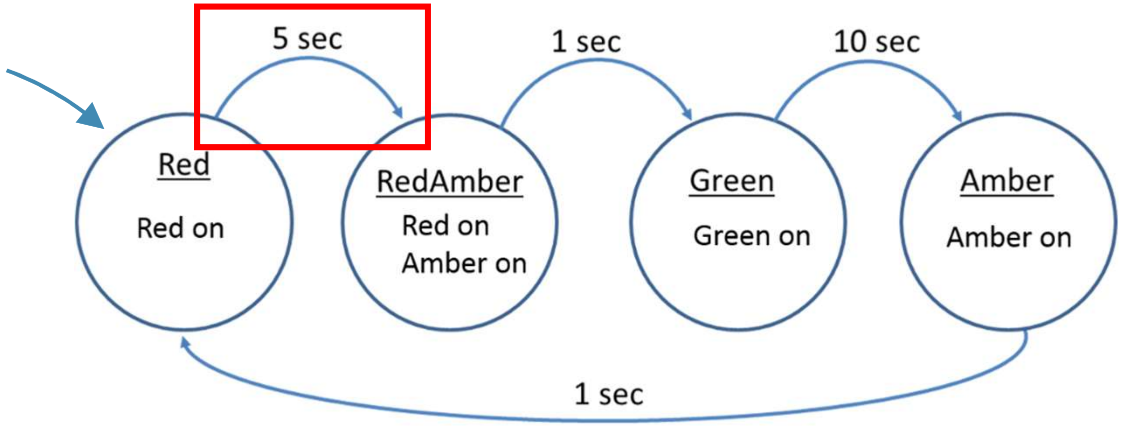
```
//================= Students add STATES below this line ====================
//----------start of state S_DRed -----
void S_DRed()    // This state the driver light RED lit.
{
  if(FSM1.doTask())
  {
    LEDDisplay.setValue(101); //show the current state on LED display module
    DRed.setHiLow(1);
    DAmber.setHiLow(0);
    DGreen.setHiLow(0);
    //PRed.setHiLow(0);
    //PGreen.setHiLow(1);
    //flash=0;
  }
  if (FSM1.getTime() >5000)  FSM1.transit(S_DRedAmber); //this state will be kept for 5sec.
  // if (SW14.getHiLow()==1 && FSM1.getTime() > 1000) FSM1.transit(S_DRedAmber);
}
```



Traffic Light Transition

- After **5000 ms**, transit to “**S_DRedAmber**” state.

```
//================= Students add STATES below this line ====================
//----------start of state S_DRed -----
void S_DRed()    // This state the driver light RED lit.
{
  if(FSM1.doTask())
  {
    LEDDisplay.setValue(101); //show the current state on LED display module
    DRed.setHiLow(1);
    DAmber.setHiLow(0);
    DGreen.setHiLow(0);
    //PRed.setHiLow(0);
    //PGreen.setHiLow(1);
    //flash=0;
  }
  if (FSM1.getTime() >5000)  FSM1.transit(S_DRedAmber); //this state will be kept for 5sec.
  // if (SW14.getHiLow()==1 && FSM1.getTime() > 1000) FSM1.transit(S_DRedAmber);
}
```



Traffic Light Adding State

- **Copy** and **paste** an existing state.
- Modify the **state name**.
- Modify its **output** (including LED state display).
- Add **transition condition** to other states.
- Modify **transition into this state**.

```
//================== Students add STATES below this line =====================
//----------start of state S_DRed -----
void S_DRed()    // This state the driver light RED lit.
{
  if(FSM1.doTask())
  {
    LEDDisplay.setValue(101); //show the current state on LED display module
    DRed.setHiLow(1);
    DAmber.setHiLow(0);
    DGreen.setHiLow(0);
    //PRed.setHiLow(0);
    //PGreen.setHiLow(1);
    //flash=0;
  }
  if (FSM1.getTime() >5000)  FSM1.transit(S_DRedAmber); //this state will be kept for 5sec.
  // if (SW14.getHiLow()==1 && FSM1.getTime() > 1000) FSM1.transit(S_DRedAmber);
}
```

Summary

- Finite-state machine (FSM) is a **model** to describe a machine.
- It has a **finite number of states**.
- The state changes (or **transits**) in response to the **inputs** (including time elapsed).
- Actions (or **outputs**) can be performed when a state is entered.
- The **predetermined behavior** of the machine depends on the sequence of events presented.
- A group of states can be reused, to apply a partial solution when similar problem is encountered.
- This "**divide-and-conquer**" technique is useful in tackling complex problem.