University of Illinois at Urbana-Champaign
Dept. of Electrical and Computer Engineering

ECE 120: Introduction to Computing

Finite State Machines (FSMs)

A Finite State Machine (FSM) Models a System

A model of a system
- system moves among a finite set of states
- motion based on external inputs
- produces external outputs

Examples include:
- coin/bill-operated machines,
- many vehicle control systems, and
- computers executing programs.

An FSM Consists of Five Parts

1. a finite set of states **(bits)**
2. a set of possible inputs **(bits)**
3. a set of possible outputs **(bits)**
4. a set of transition rules **(Boolean expressions)**
5. methods for calculating outputs **(Bool. expr's)**

When implemented as a digital system, all parts of an FSM must be mapped to … **bits!**

A Digital FSM Must be Complete

We implement FSMs as clocked synchronous sequential circuits. (So state ID bits are stored in flip-flops.)

Given **any state** and **any combination of inputs**, a **transition rule** from the given state to a next state **must be defined**.

Self-loops–transitions from a state to itself–are acceptable.

Use Keyless Entry as a Motivating Example

meaning	state	driver's door	other doors	alarm on?
vehicle locked	LOCKED	locked	locked	no
driver door unlocked	DRIVER	unlocked	locked	no
all doors unlocked	UNLOCKED	unlocked	unlocked	no
alarm sounding	ALARM	locked	locked	yes

Table is a **list of abstract states**.

A List of Abstract States Need Only List States

In a list of abstract states,
- we can just list the states.
- Adding human meanings is optional (good to have if state names are generic).

Including outputs
- is also optional,
- and implies that **outputs depend only on state**.*

*An extra assumption that we will always make in our class.

An Abstract Next-State Table Captures Expected Behavior

To specify transitions, we use **a next-state table**, which maps combinations of states and inputs into next states.

This is an **abstract next-state table**.

state	action/input	next state
LOCKED	push "unlock"	DRIVER
DRIVER	push "unlock"	UNLOCKED
(any)	push "lock"	LOCKED
(any)	push "panic"	ALARM

Abstract Next-State Table Does Not Answer All Questions

We wrote transitions for typical use cases, but **the table can be incomplete, ambiguous, and even inconsistent**.

For example, what happens if the user pushes "lock" and "unlock" at the same time?

state	action/input	next state
LOCKED	push "unlock"	DRIVER
DRIVER	push "unlock"	UNLOCKED
(any)	push "lock"	LOCKED
(any)	push "panic"	ALARM

Many Design Decisions are Usually Needed

All such **design decision questions should eventually be considered, and preferably answered.**

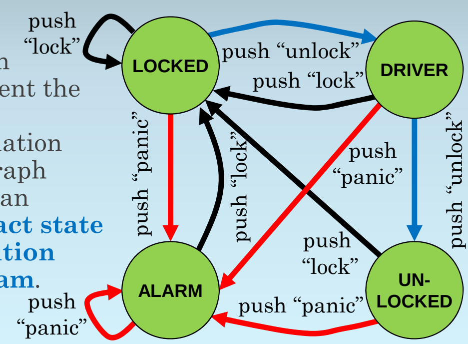
Be aware: **any digital logic implementation will define answers**.

Only when any possible answer is acceptable should you make use of "don't cares."

Typically, you should **review the final implementation** to determine how any questions left open are answered.

Abstract State Transition Diagram: the Same Information

We can represent the same information as a graph called an **abstract state transition diagram**.



It's Time to Make Our Design Complete and Concrete

The abstract next-state table and the abstract state transition diagram (can) **contain exactly the same information**.

They answer the same questions.

And **neither is complete**.

So. It's time for … **bits!**

Let's Start with the State Identifiers

How many bits do we need to identify a state?

There are 4 states.

$\lceil \log_2(4) \rceil = 2$ **bits.**

Call them S_1S_0.

"S" is for "S(tate)."

All Outputs and Inputs Must Also Use Bits

What about outputs?

D driver door; 1 means unlocked

R remaining doors; 1 means unlocked

A alarm; 1 means alarm is sounding

And inputs?

U unlock button; 1 means it's been pressed

L lock button; 1 means it's been pressed

P panic button; 1 means it's been pressed

We Next Choose a Representation for States

Now we can choose a representation for states and rewrite our list of states.

The order of states in the list doesn't matter.

meaning	state	S_1S_0	D	R	A
vehicle locked	LOCKED	00	0	0	0
driver door unlocked	DRIVER	10	1	0	0
all doors unlocked	UNLOCKED	11	1	1	0
alarm sounding	ALARM	01	0	0	1

Choice of Representation Affects Amount of Logic Needed

As you may realize
- from your experience with bit-sliced designs,
- **the representation does matter** (for the amount of logic needed).

We will talk more later about ways to choose.

Use $S_1^+S_0^+$ to Denote the Next State (in Next Clock Cycle)

The +'s in $S_1^+S_0^+$ indicate that these are values **in the next clock cycle**.

Let's rewrite the next-state table with bits.
- The table gives us $S_1^+S_0^+$ as a function of current state S_1S_0 and inputs **ULP**.
- Such tables typically use binary order for states (vertical) and inputs (horizontal).
- We use Grey code order on both axes for convenience (in copying to K-maps).

How to Fill in the Next-State Table

Where should we start?

current state S_1S_0	ULP: 000	001	011	010	110	111	101	100
00								
01								
11								
10								

What about multiple buttons?

Let's make some design decisions first…

Completing the Design Requires Decisions

To fill in the next-state table
- starting with only the abstract design,
- we **need to make many design decisions**,
- including some that we haven't even recognized yet.

For example,
- What happens when the user presses more than one button?
- What happens when the user presses "unlock" in the **UNLOCKED** state?

Make Design Decisions Early When Possible

Let's try to **make decisions first**.

Design decisions can shape the design, and **may conflict with one another**.

Making decisions early and writing them down ensures that
- any **issues are raised early**, and that
- **known decisions are not overlooked**
- (in which case the final design answers them implicitly, with no human guidance).

Start by Deciding How to Handle Multiple Buttons

We're going to start by **prioritizing the buttons**.

Our rules:
- Panic has priority!
- Lock has second priority.
- Unlock only matters when neither of the others is pressed.

Start with the Panic Button (Highest Priority)

The next-state table gives us $S_1^+S_0^+$.

current state S_1S_0	ULP 000	001	011	010	110	111	101	100
00		01	01			01	01	
01		01	01			01	01	
11		01	01			01	01	
10		01	01			01	01	

panic button pushed

Continue with the Lock Button (Second Priority)

The next-state table gives us $S_1^+S_0^+$.

current state S_1S_0	ULP 000	001	011	010	110	111	101	100
00		01	01	00	00	01	01	
01		01	01	00	00	01	01	
11		01	01	00	00	01	01	
10		01	01	00	00	01	01	

lock button pushed

No Buttons? No Change. All Self-Loops

What if the user pushes nothing?

current state S_1S_0	ULP 000	001	011	010	110	111	101	100
00	00	01	01	00	00	01	01	
01	01	01	01	00	00	01	01	
11	11	01	01	00	00	01	01	
10	10	01	01	00	00	01	01	

no buttons pushed

Finally, Unlock … But are We Done?

Two transitions were defined for Unlock.

current state S_1S_0	ULP 000	001	011	010	110	111	101	100
00	00	01	01	00	00	01	01	10
01	01	01	01	00	00			
11	11	01	01	00	00			
10	10	01	01	00	00	01	01	11

from LOCKED

What about these?

from DRIVER

We Have More Design Decisions to Make!

What should happen if we press "unlock" when the car is already fully unlocked (in the **UNLOCKED** state)?

Maybe just stay **UNLOCKED**.

What should happen if we press "unlock" while the alarm is sounding?
- Continue to lock out an attacker / thief?
- Or open the doors so that the owner can climb inside quickly?

Let's Implement Our Decisions

Ignore Unlock in both other cases.

current state S_1S_0	ULP 000	001	011	010	110	111	101	100
00	00	01	01	00	00	01	01	10
01	01	01	01	00	00	01	01	01
11	11	01	01	00	00	01	01	11
10	10	01	01	00	00	01	01	11

from ALARM

from UNLOCKED

The Rest You Know How to Do

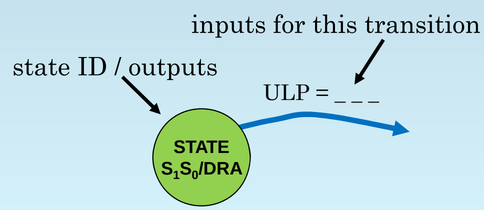
The rest is K-maps, expressions, and logic.

1. Express S_1^+ and S_0^+ in terms of S_1, S_0, **U**, **L**, and **P**.
2. Express **D**, **R**, and **A** in terms of S_1, S_0.
3. Build the combinational logic.
4. Put the next state expressions S_1^+ and S_0^+ into the **D** inputs of two flip-flops.

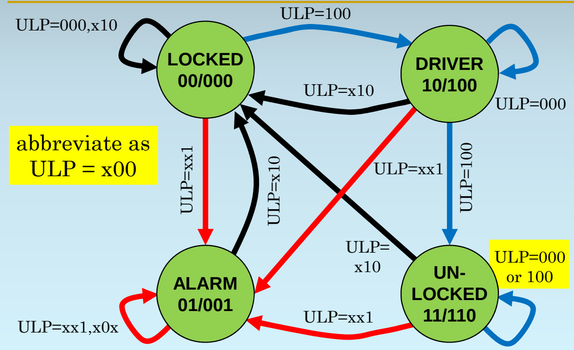
You should do it as an exercise. Break up the truth tables or use 5-variable K-maps.

One Last Tool: the Complete State Transition Diagram

The complete **state transition diagram** contains the information in both the state list and the next-state table.



Complete State Transition Diagram



Be Careful with Input Abbreviations

Input abbreviations can render a state transition diagram

- incomplete (if labels fail to cover all input combinations), or
- inconsistent (if labels indicate multiple next states).

For example,

- self-loop from **ALARM** labeled **ULP=xx1,x0x**:
- the patterns **x01** match both labels!
- In this case, these two combinations go to the same next state, so it's ok.

