COMSM1302
Overview of Computer Architecture

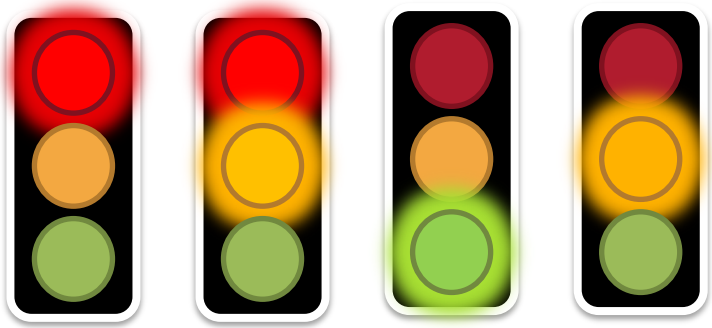
Lecture 7
Simple controllers based on Finite State Machines

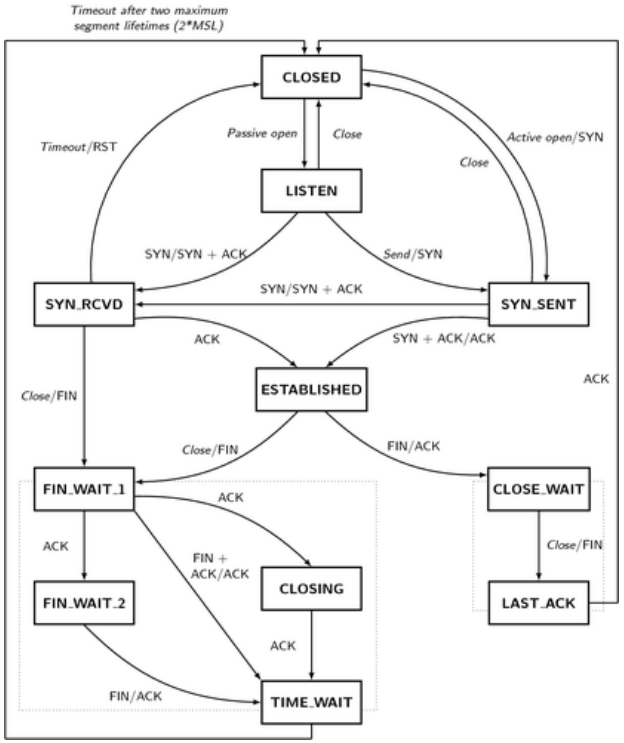
In this lecture (and the next two)

Foundations	• Data representation, logic, Boolean algebra.
Building blocks	• Transistors, transistor based logic, simple devices, storage.
Modules	• Memory, **simple controllers, FSMs**, processors and execution.
Programming	• Machine code, assembly, high-level languages, compilers.
Wrap-up	• Operating systems, energy aware computing.

In this lecture

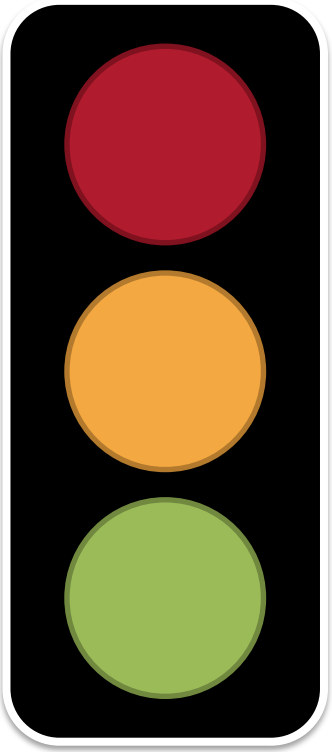
- Finite State Machine (FSM) design
- Building our first FSM
- Simplification of Boolean functions





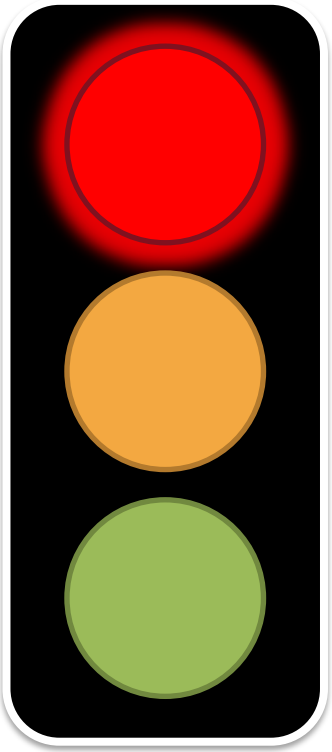
Traffic light

- A (UK) traffic light has three outputs.
  - Red
  - Amber
  - Green
- Red means stop.
- Green means go (if it is safe to do so).
- Amber warns of a change taking place.



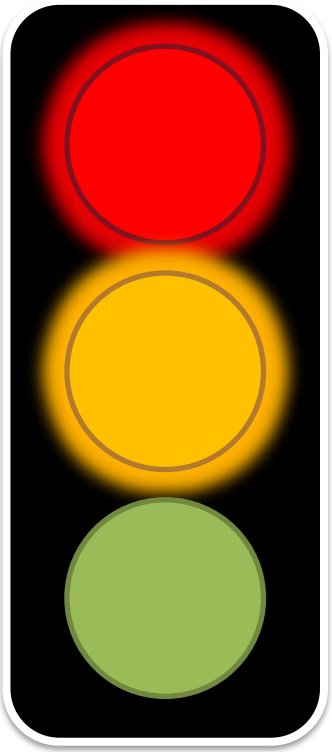
Traffic light

- Assume initially, it is red.



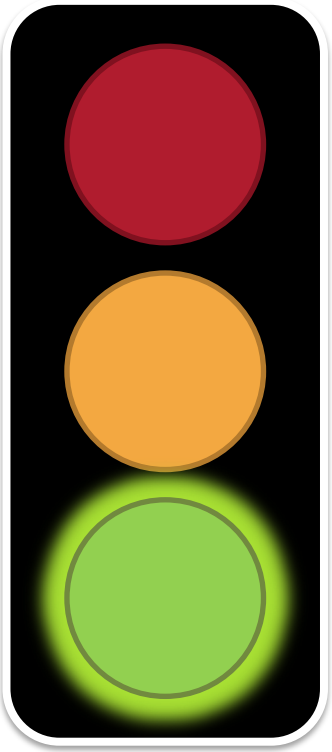
Traffic light

- Next, red and amber signify a change.
- This still means stop but you can prepare to go.
  - Why red and amber and not just amber?



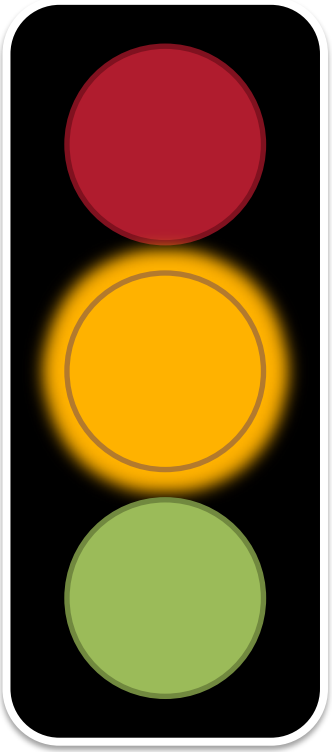
Traffic light

- Now we transition to green.



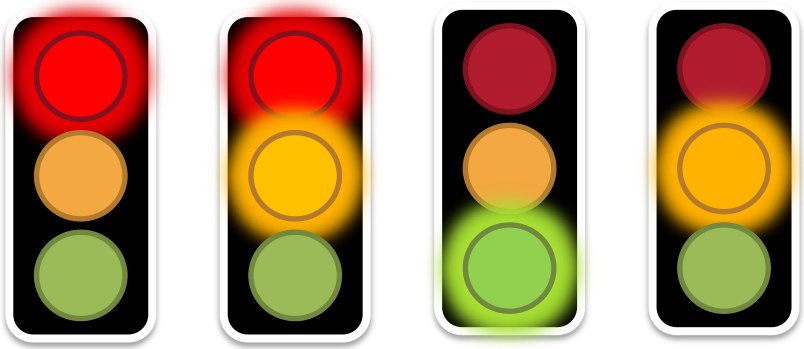
Traffic light

- Next, amber alone signifies that we are transitioning towards “stop”.



Traffic light FSM

- The traffic light system can be represented as a very simple Finite State Machine. It has:
  - Four states
  - Very straightforward transitions between them.



- **What limitations does this very simple FSM have that would make this a poor traffic light?**

Finite State Machines

An FSM consists of:

- *Inputs* (from the environment);
- *Outputs* (to the environment);
- Internal *state* of the system stored in memory;
- *Next state logic;*
- *Output logic.*

Finite State Machines

An FSM consists of:

- *Inputs* (from the environment);
- *Outputs* (to the environment);
- Internal *state* of the system stored in memory;
- *Next state logic;*
- *Output logic.*

- An FSM can be in exactly one of a finite number of states at any given time.
- The FSM can change (i.e. transition) from one state to another state in response to some inputs. Such a state change is called a transition.

Next state and Output functions

The *next state* and the *output* can be determined based on the *input* and the *current state* using combinatorial logic.

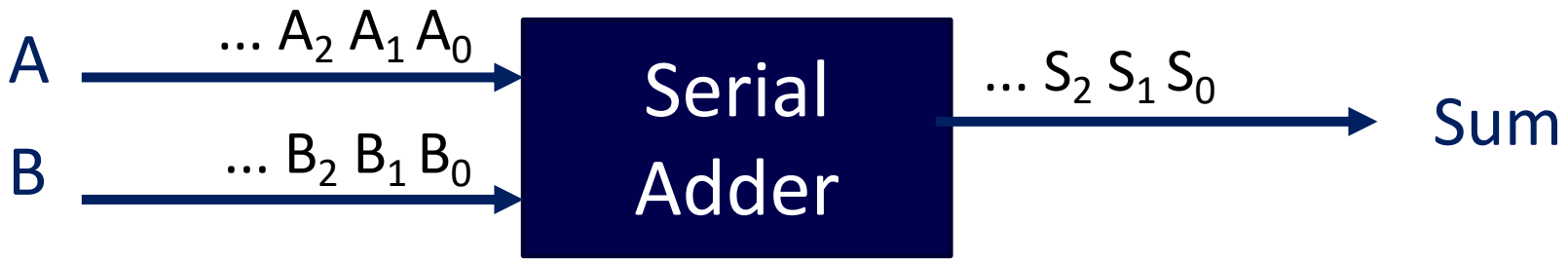
There are two functions:

- *Next state function* δ *(Input, Current State)*
- *Output function* ω *(Input, Current State)*

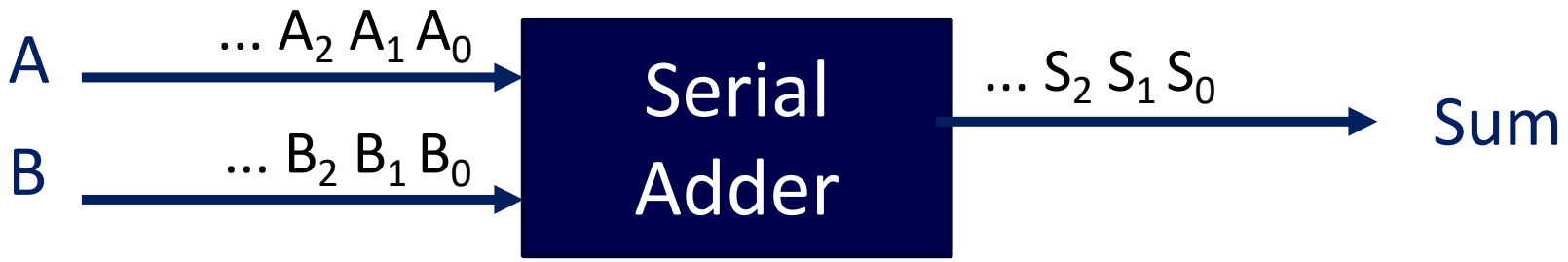
This is called a Mealy machine.

THE SERIAL BINARY ADDER

FSM example: Serial (binary) Adder



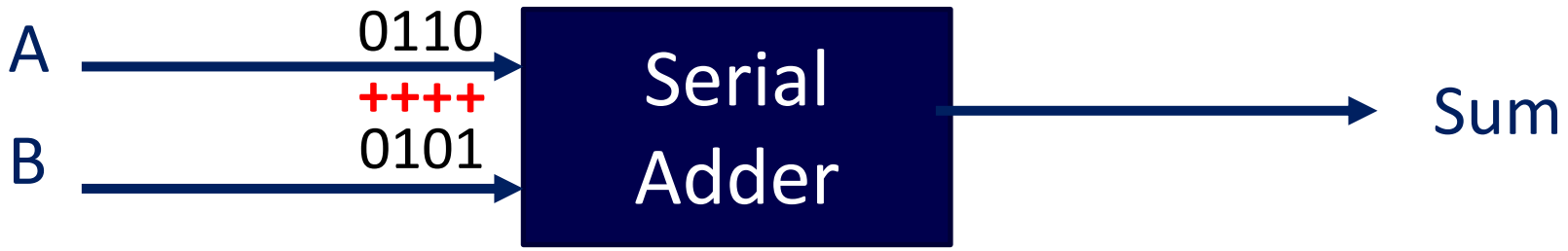
FSM example: Serial (binary) Adder



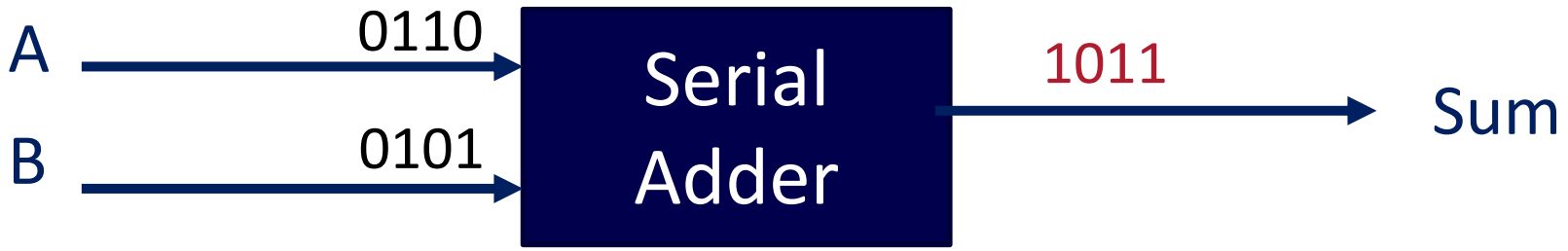
Inputs: A_i and B_i

Output: Sum_i

FSM example: Serial Adder



FSM example: Serial Adder

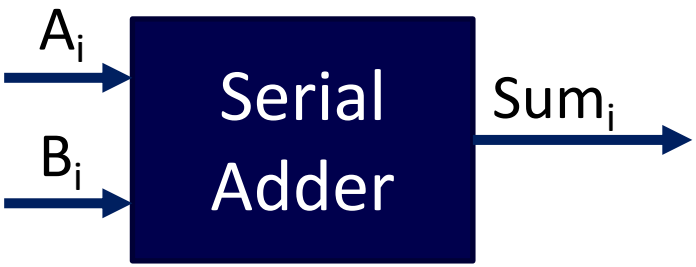


Inputs: A = 0110 = 6 and B = 0101 = 5

Output: Sum_i = 1011 = 11 = 6+5

States:

FSM example: Serial Adder

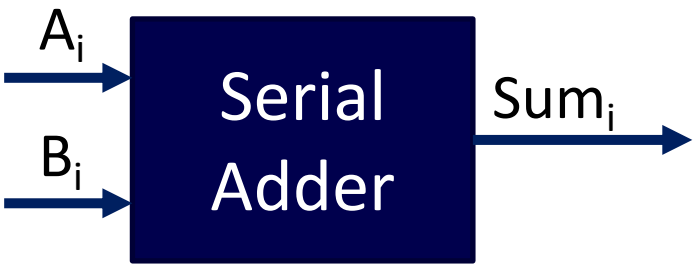


Inputs: A_i and B_i

Output: Sum_i

States: NoCarry
Carry

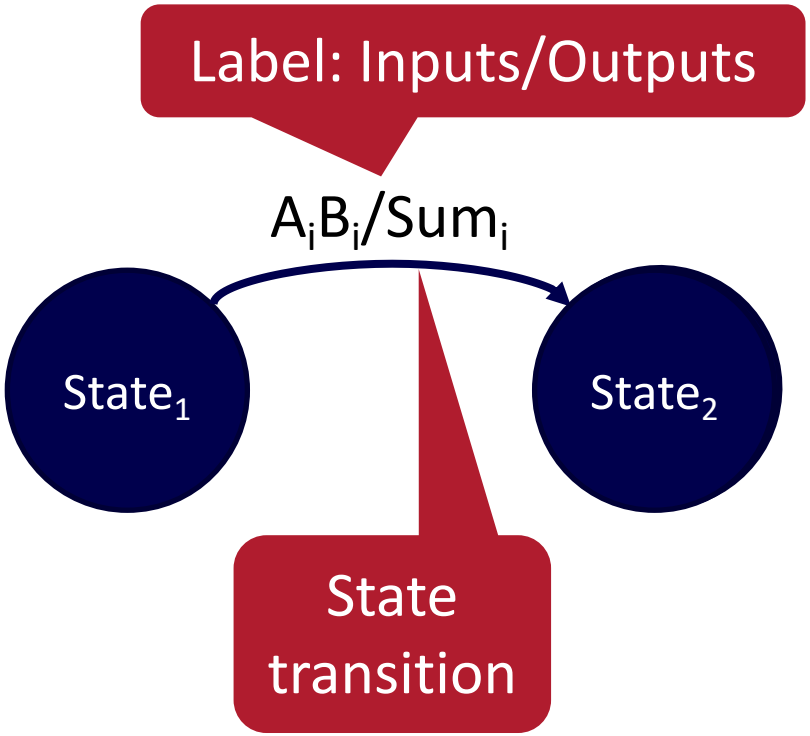
FSM example: Serial Adder



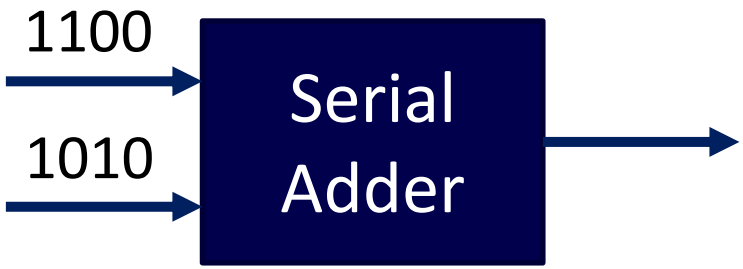
Inputs: A_i and B_i

Output: Sum_i

States: NoCarry
Carry



FSM example: Serial Adder



Label: Inputs/Outputs

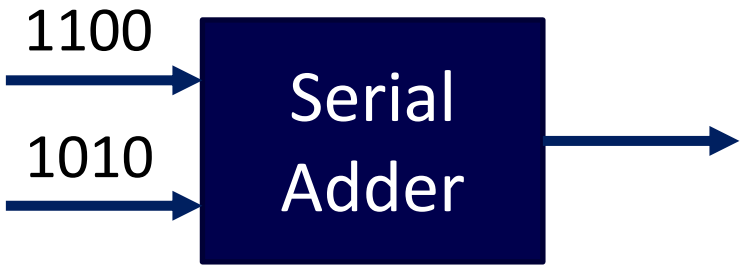
A_iB_i/Sum_i

Inputs: A_i and B_i

Output: Sum_i

States: NoCarry

FSM example: Serial Adder



Label: Inputs/Outputs

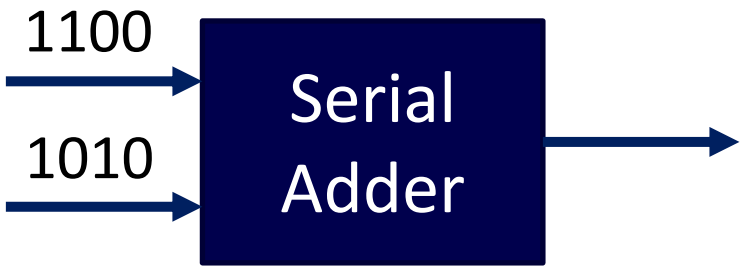
A_iB_i/Sum_i

Inputs: A_i and B_i

Output: Sum_i

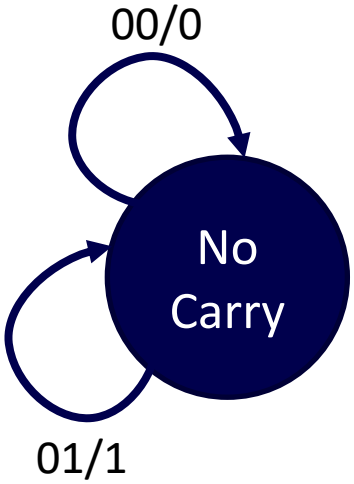
States: NoCarry
Carry

FSM example: Serial Adder



Label: Inputs/Outputs

A_iB_i/Sum_i

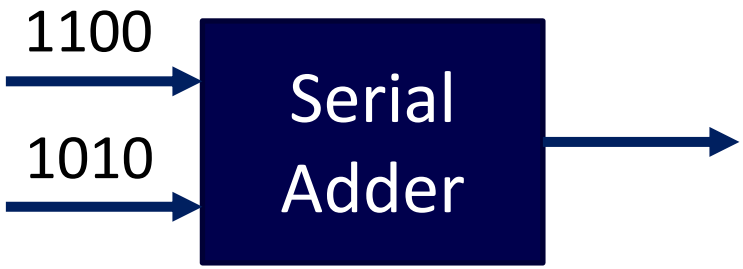


Inputs: A_i and B_i

Output: Sum_i

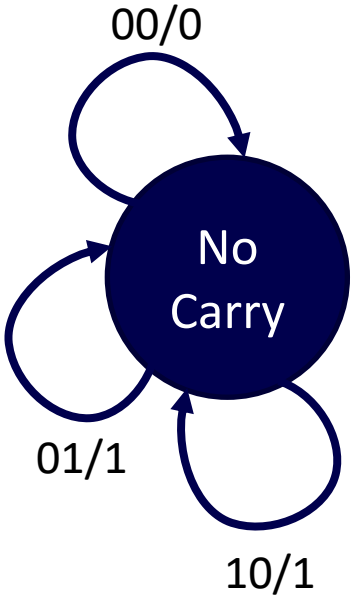
States: NoCarry
Carry

FSM example: Serial Adder



Label: Inputs/Outputs

A_iB_i/Sum_i

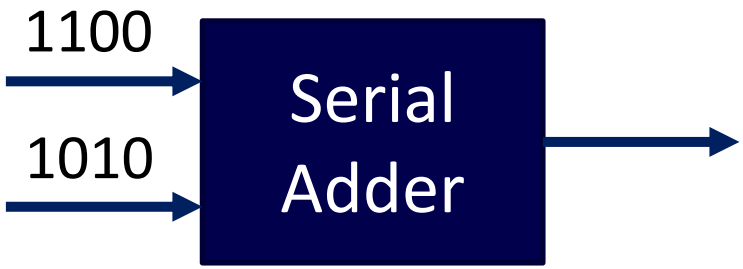


Inputs: A_i and B_i

Output: Sum_i

States: NoCarry
Carry

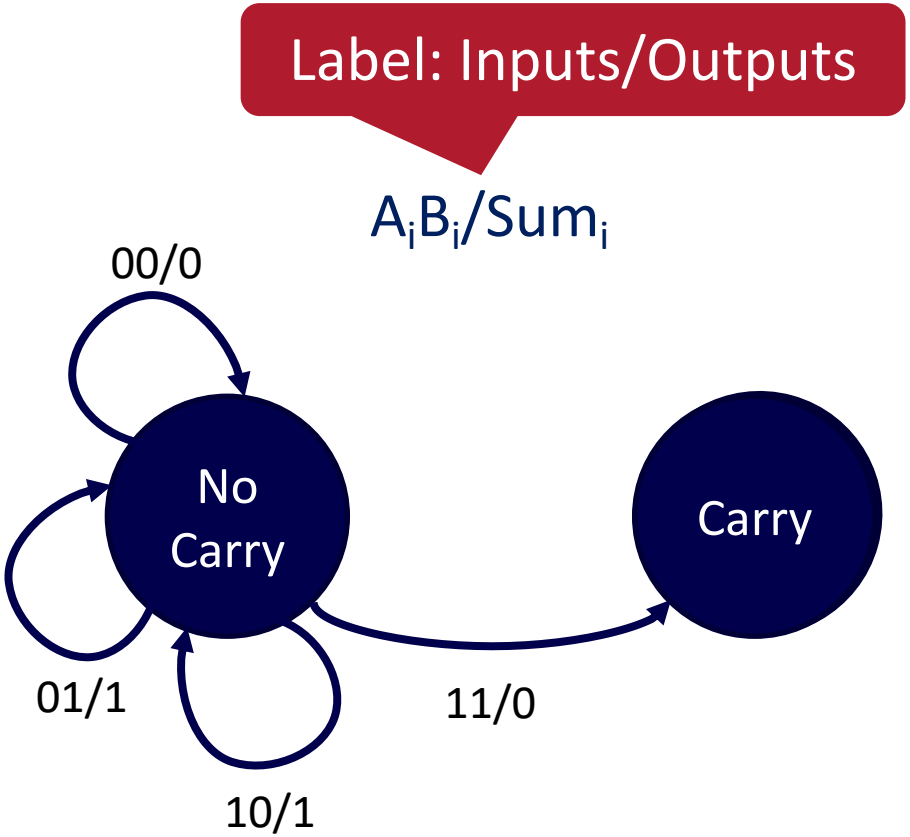
FSM example: Serial Adder



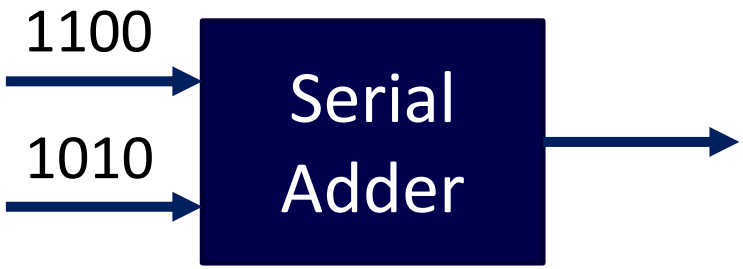
Inputs: A_i and B_i

Output: Sum_i

States: NoCarry
Carry



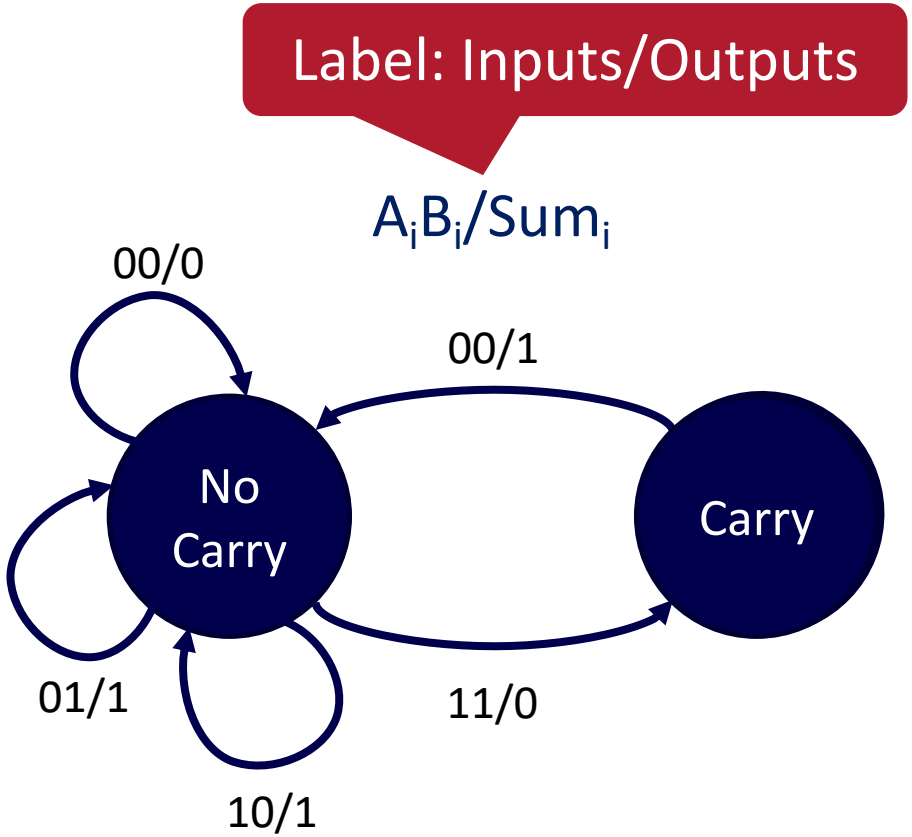
FSM example: Serial Adder



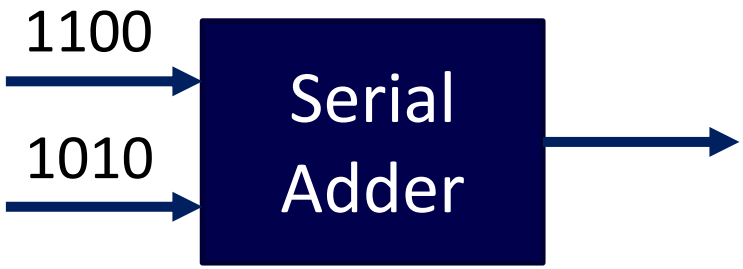
Inputs: A_i and B_i

Output: Sum_i

States: NoCarry
Carry



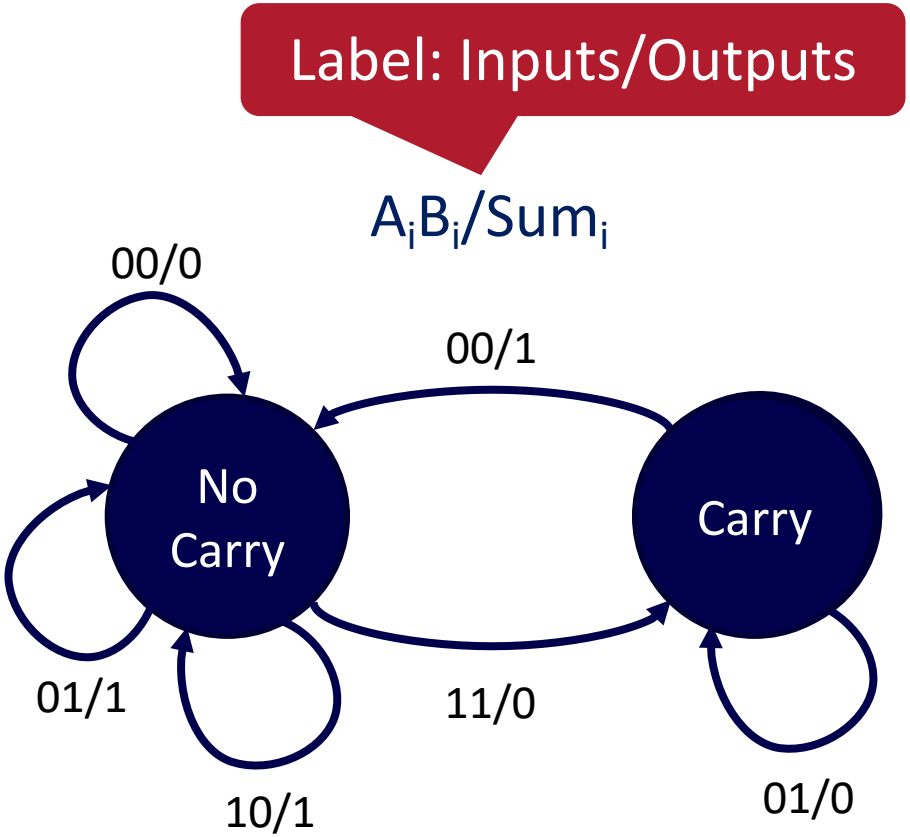
FSM example: Serial Adder



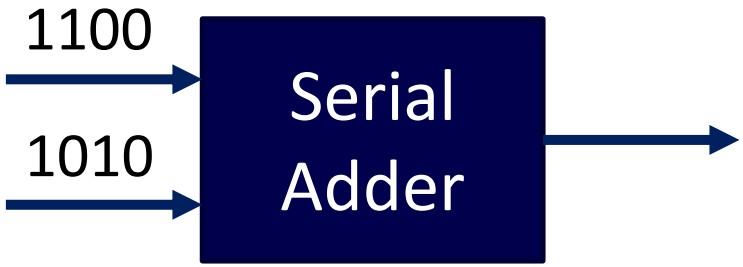
Inputs: A_i and B_i

Output: Sum_i

States: NoCarry
Carry



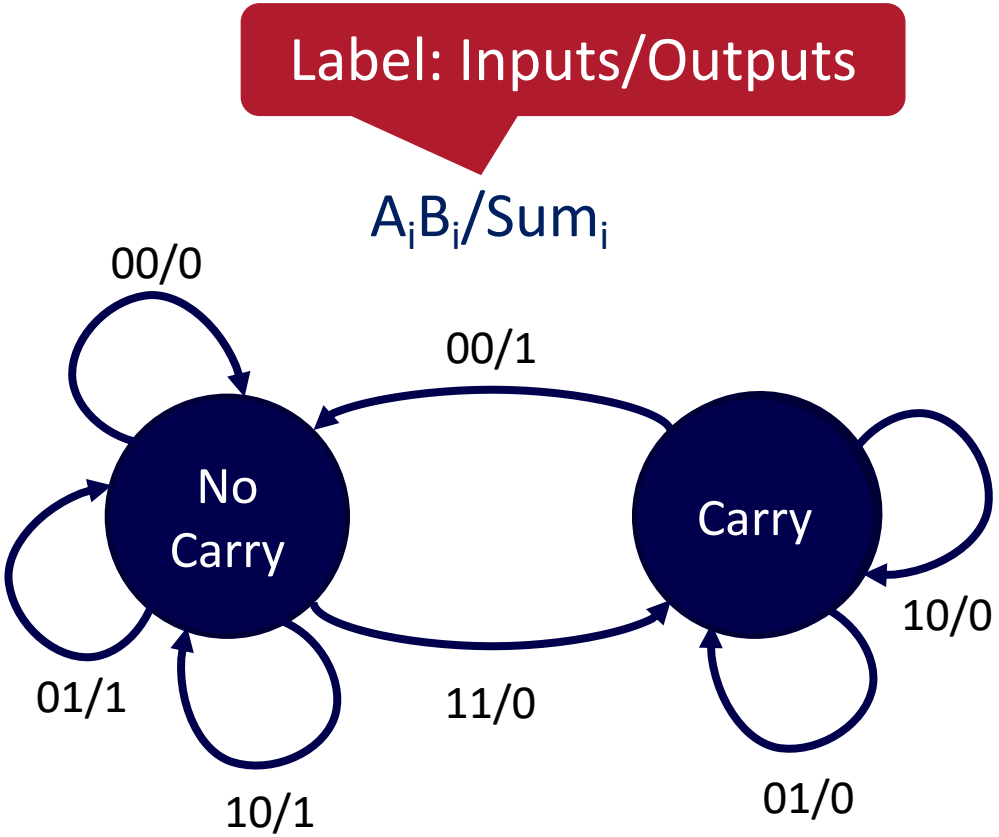
FSM example: Serial Adder



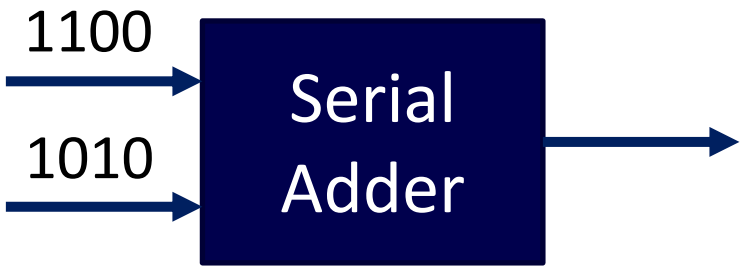
Inputs: A_i and B_i

Output: Sum_i

States: NoCarry
Carry



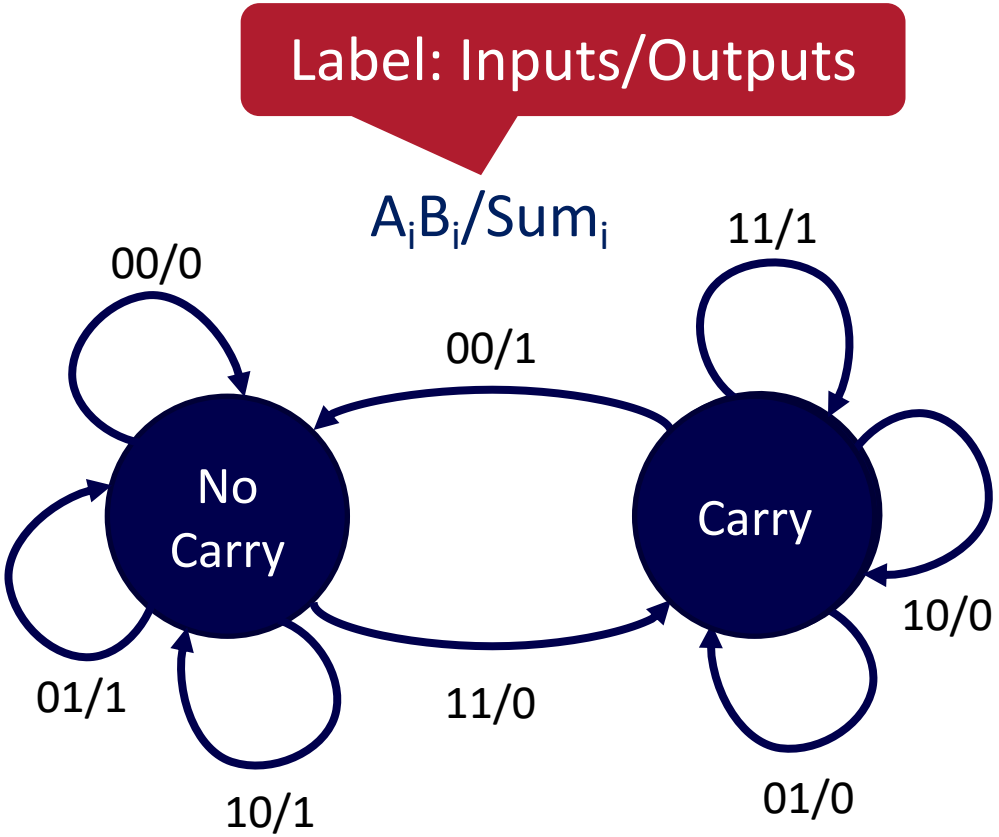
FSM example: Serial Adder



Inputs: A_i and B_i

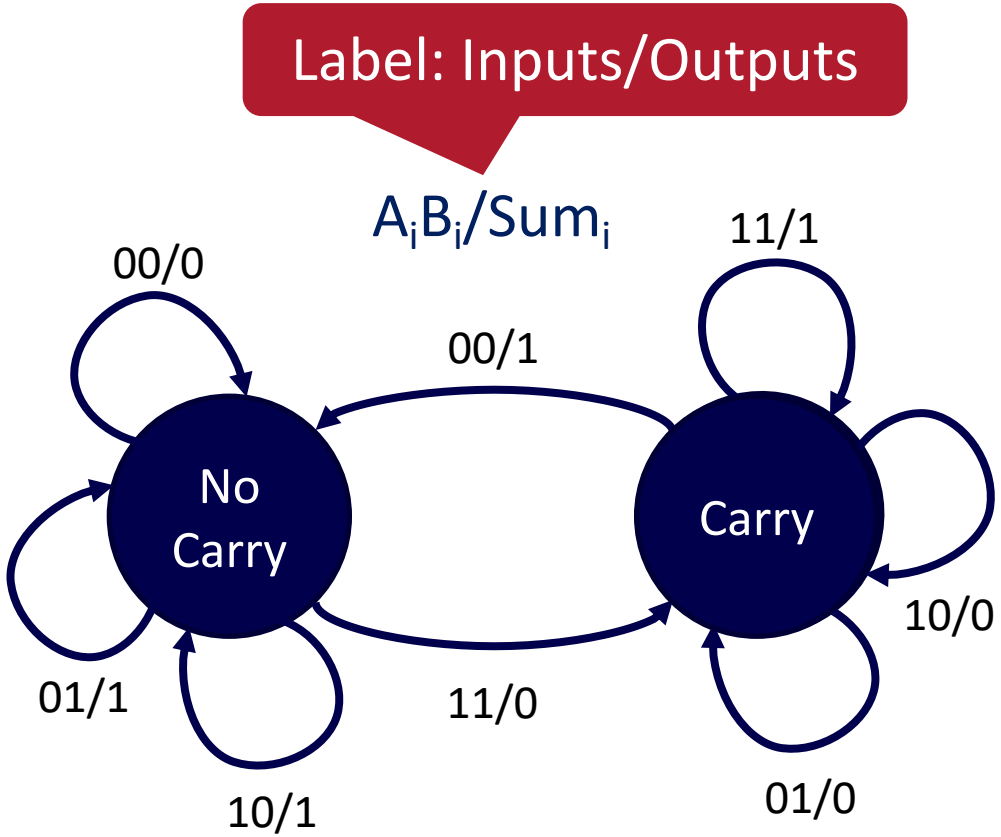
Output: Sum_i

States: NoCarry, Carry



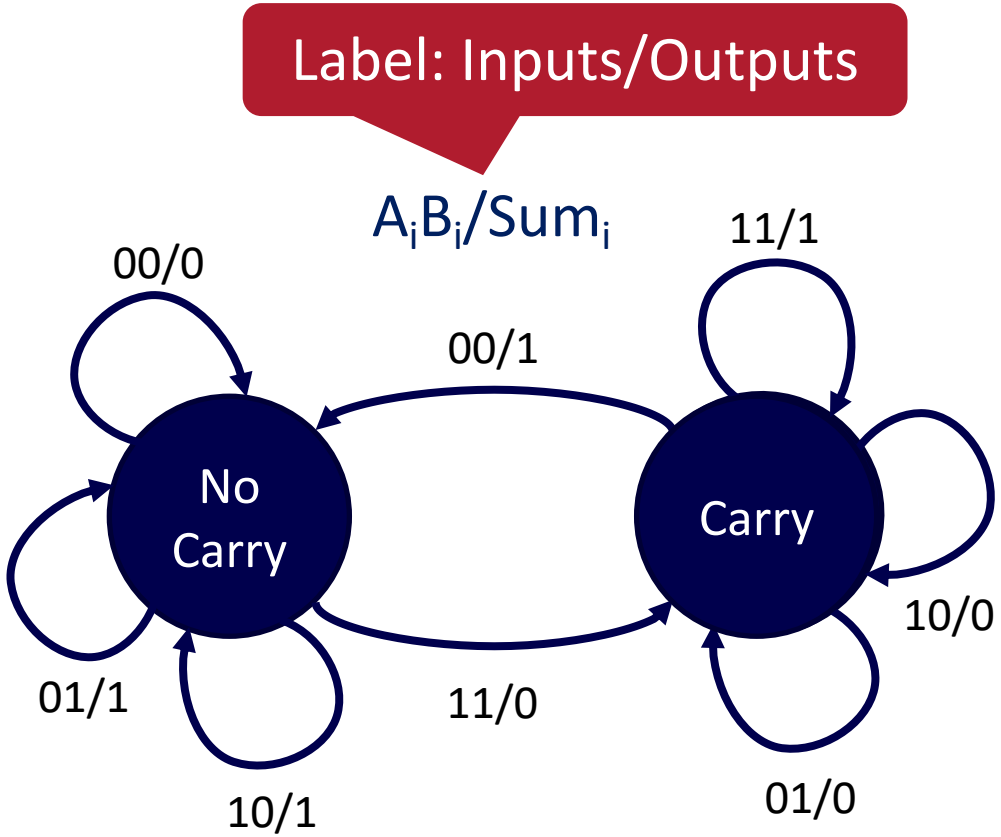
FSM example: Serial Adder

Current State	Input		Next State	Output
S	A_i	B_i	S_{next} (S')	Sum_i



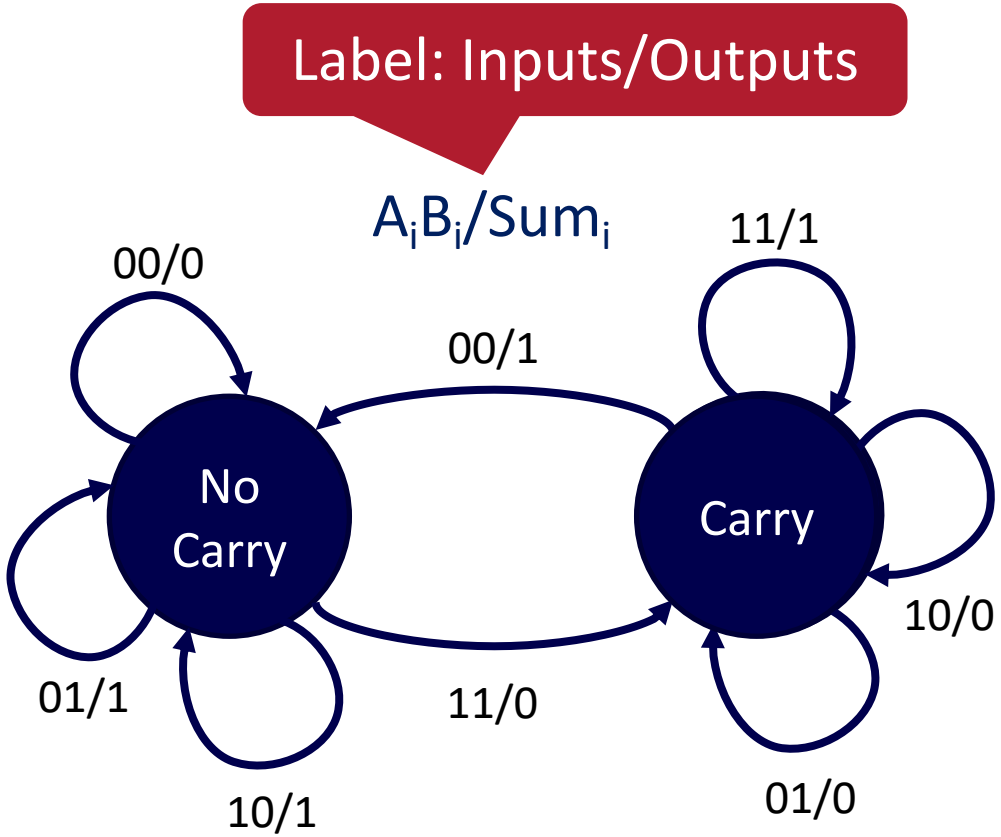
FSM example: Serial Adder

Current State	Input		Next State	Output
S	A_i	B_i	S_{next} (S')	Sum_i
NoCarry	0	0		
NoCarry	0	1		
NoCarry	1	0		
NoCarry	1	1		



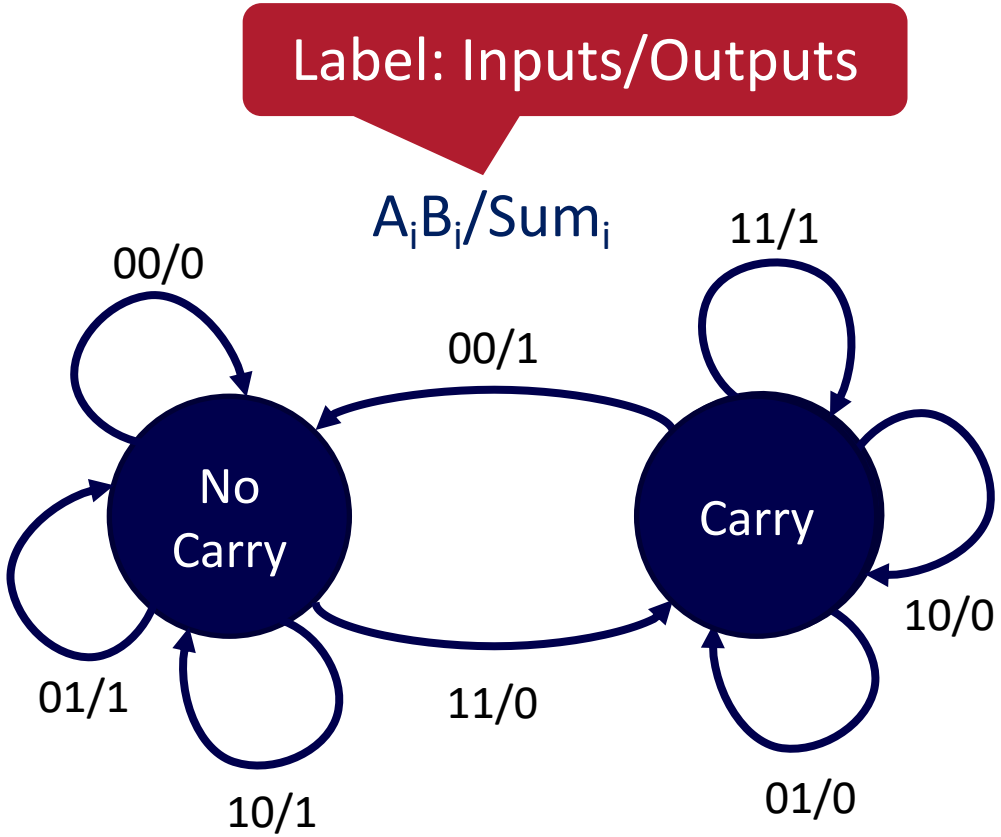
FSM example: Serial Adder

Current State	Input		Next State	Output
S	A_i	B_i	S_{next} (S')	Sum_i
NoCarry	0	0	NoCarry	0
NoCarry	0	1	NoCarry	1
NoCarry	1	0	NoCarry	1
NoCarry	1	1	Carry	0
Carry	0	0		
Carry	0	1		
Carry	1	0		
Carry	1	1		



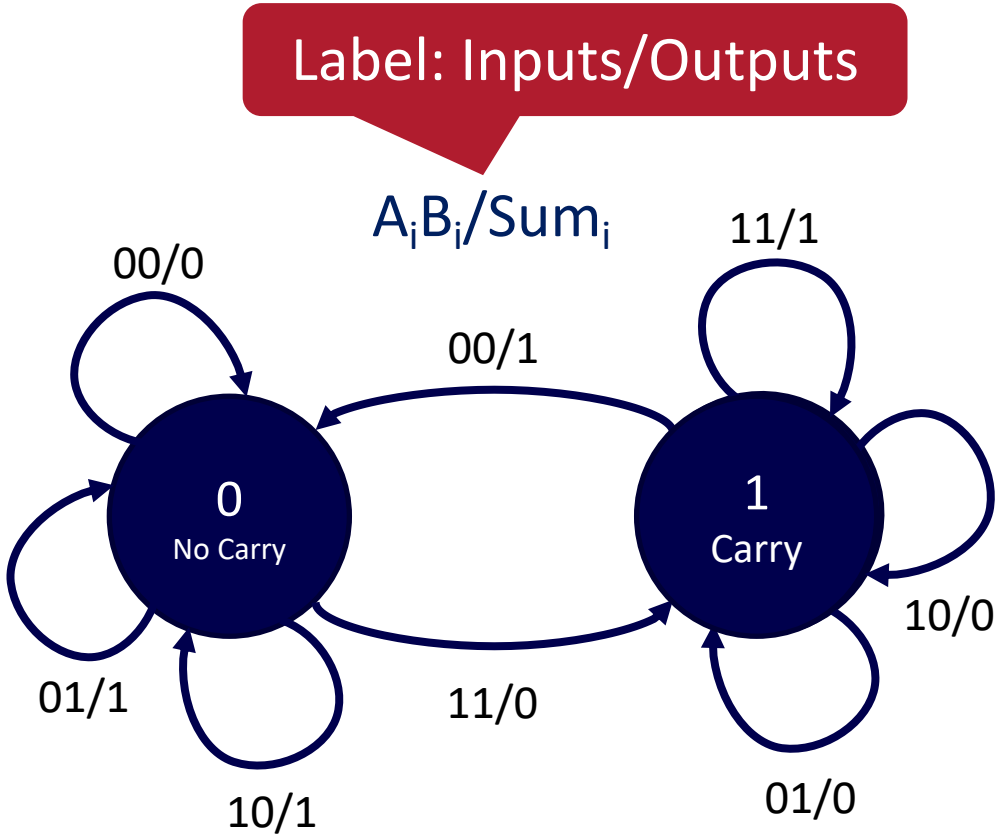
FSM example: Serial Adder

Current State	Input		Next State	Output
S	A_i	B_i	S_{next} (S')	Sum_i
NoCarry	0	0	NoCarry	0
NoCarry	0	1	NoCarry	1
NoCarry	1	0	NoCarry	1
NoCarry	1	1	Carry	0
Carry	0	0	NoCarry	1
Carry	0	1	Carry	0
Carry	1	0	Carry	0
Carry	1	1	Carry	1



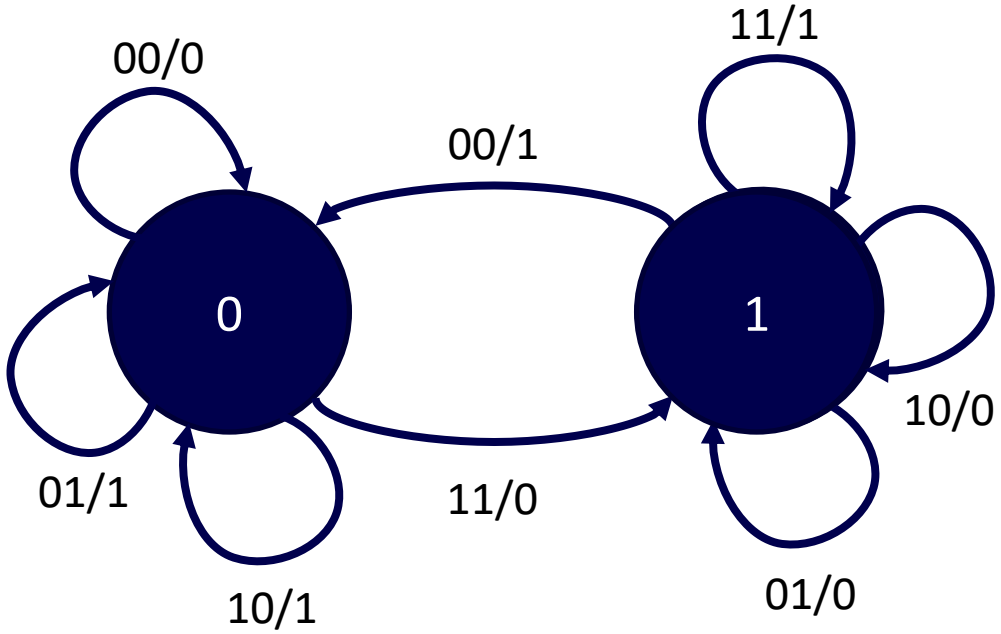
FSM example: Serial Adder

Current State	Input		Next State	Output
S	A_i	B_i	S_{next} (S')	Sum_i
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1



FSM example: Serial Adder

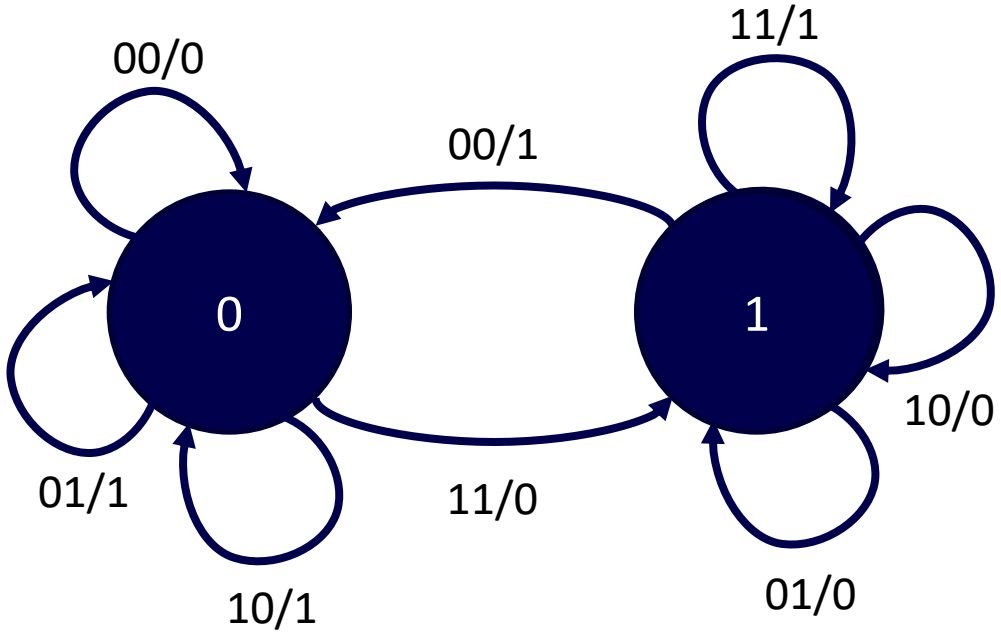
Current State	Input		Next State	Output
S	A_i	B_i	S_{next} (S')	Sum_i
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1



S_{next} =

FSM example: Serial Adder

Current State	Input		Next State	Output
S	A_i	B_i	S_{next} (S')	Sum_i
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

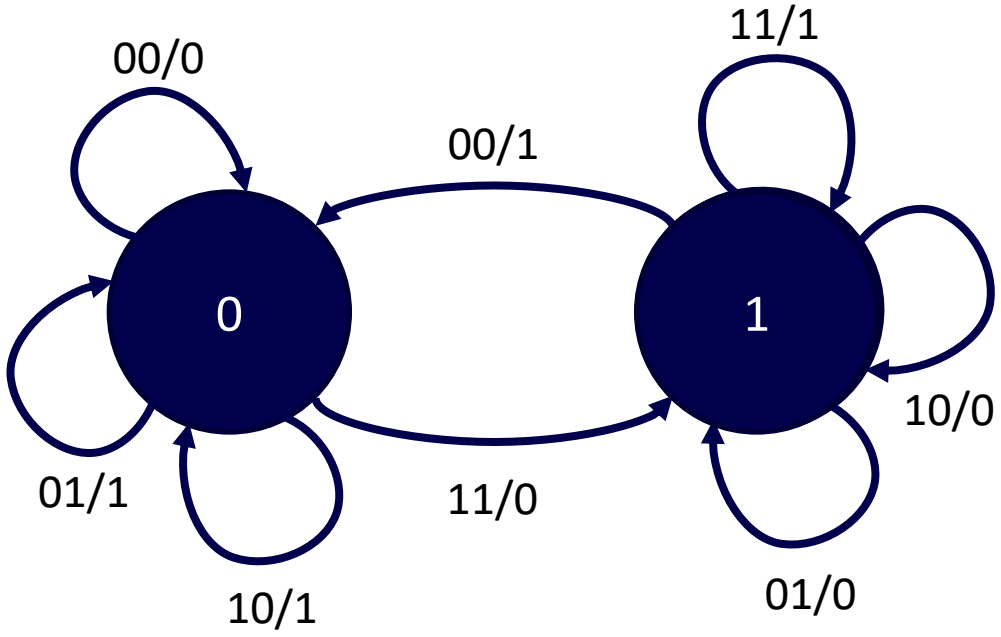


S_{next} =

Sum_i =

FSM example: Serial Adder

Current State	Input		Next State	Output
S	A_i	B_i	S_{next} (S')	Sum_i
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1



$S_{next} = \neg S \wedge A_i \wedge B_i \vee S \wedge \neg A_i \wedge B_i \vee S \wedge A_i \wedge \neg B_i \vee S \wedge A_i \wedge B_i$

$Sum_i = \neg S \wedge \neg A_i \wedge B_i \vee \neg S \wedge A_i \wedge \neg B_i \vee S \wedge \neg A_i \wedge \neg B_i \vee S \wedge A_i \wedge B_i$

FSM transition tables

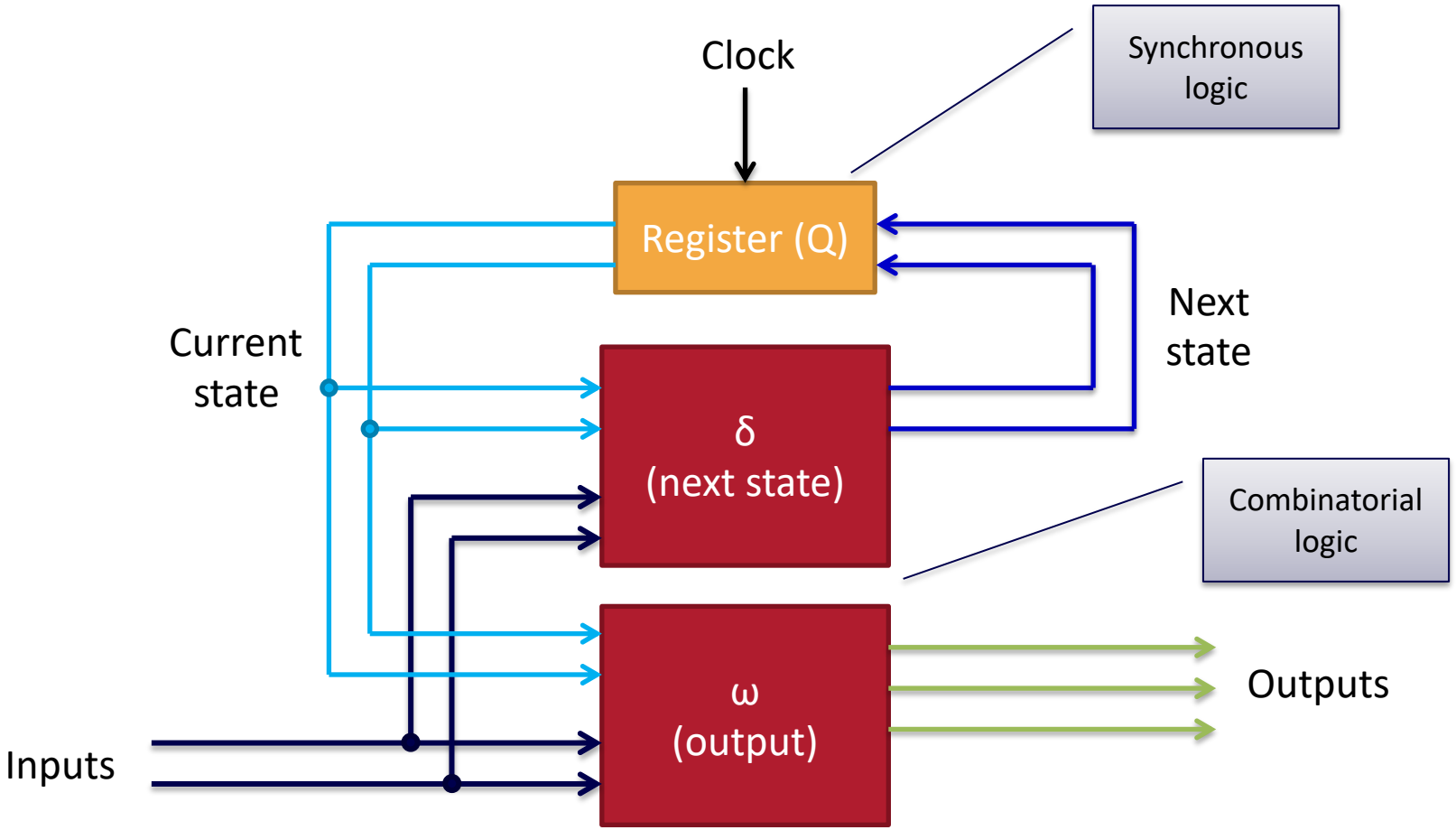
- Tables can be used to represent the logic controlling FSMs
  - The current state and inputs determine the next state and the outputs.
  - A state transition function, δ, chooses the next state.
  - An output function, ω, determines the outputs.

It's a truth table!

Current state		Inputs		Next state, δ		Outputs, ω	
Q1	Q0	I1	I0	nQ1	nQ0	O1	O0
0	0	0	0	0	0	0	0
0	0	0	1	1	0	0	0
…							
1	1	1	1	1	0	1	0

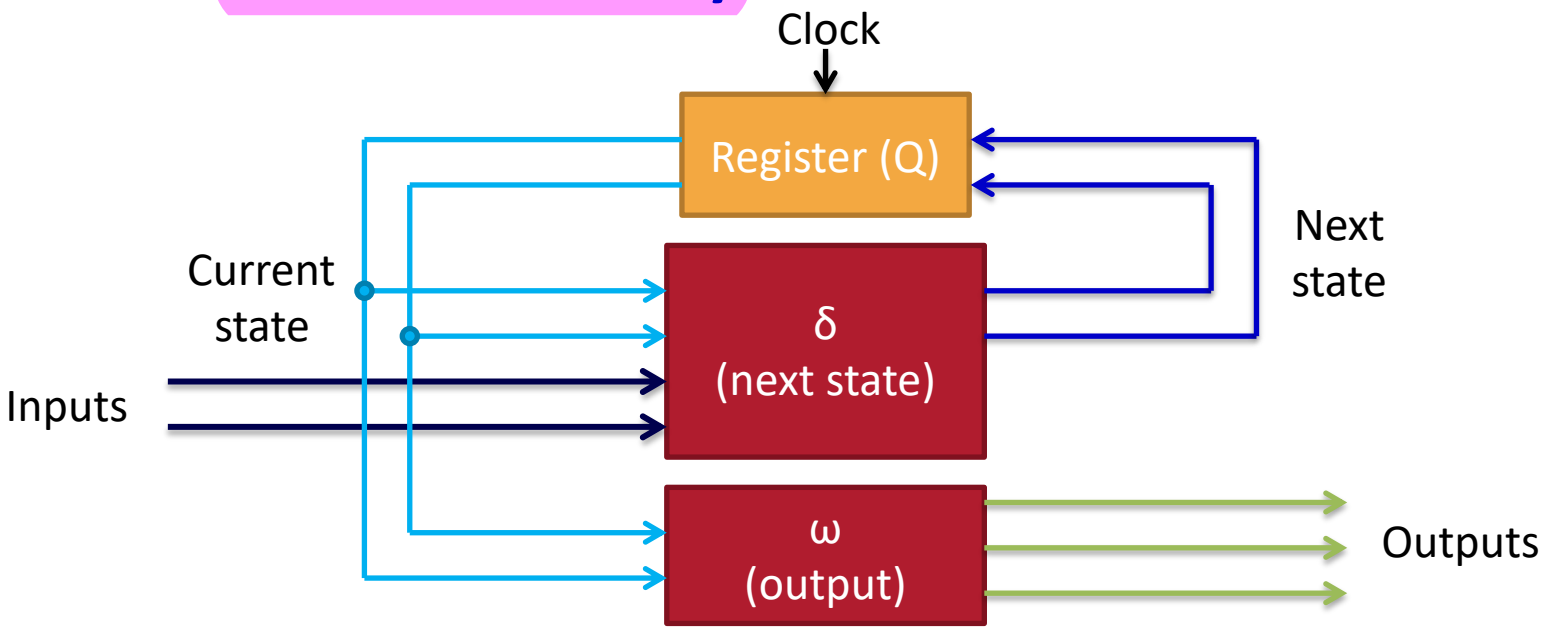
TYPES OF FINITE STATE MACHINES

FSM block level view



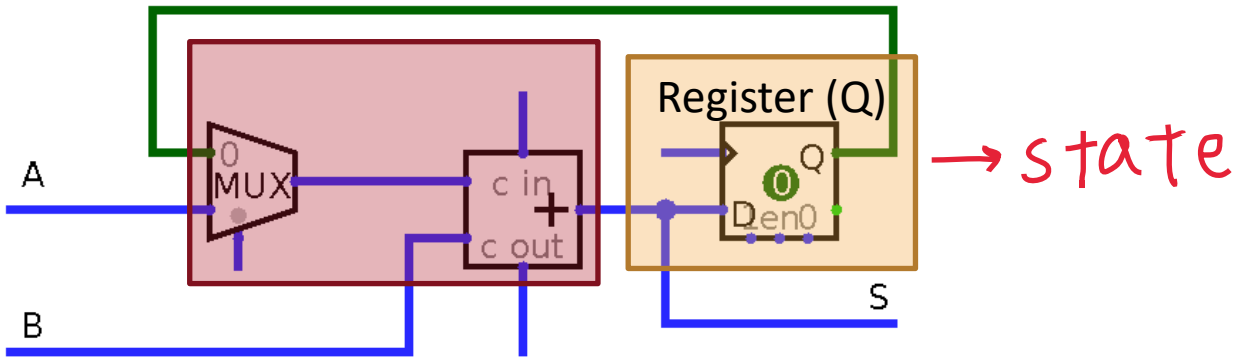
Moore vs. Mealy

- The previous slide showed a **Mealy Machine**.
  - The output was a function of **inputs and state**.
- A **Moore Machine** is similar, but outputs are a function of the **current state only**.



FSMs are Computers

- We looked at FSMs as early as Lecture 5!



- A modern microprocessor is an FSM
  - It has a ***massive*** number of states
  - But it's still **finite**…

COUNTER MACHINES

Counter machines

- Not quite a general purpose computer, but counter machines are an essential part of computing.
  - Timers.
  - Incremental progression of time.

- A counter machine is… a Finite State Machine.

- **Example: 3-bit counter**
  - **How many states does a 3-bit counter have?**
  - **What is the state transition diagram?**
  - **What is the state transition function?**

Counter machine

Your task is to experiment with a **3-bit counter machine.**

- Develop a **basic version** for which you clearly specify the input and output and the state (encoding).
  - Note that there are different options for the output, e.g.
    - the output is a single bit that tells you whether the input has been on, or
    - the output is a 3-bit value that tells you how often the input was enabled, i.e. the value of the counter.
- Extend your basic version with a **second input** that resets the counter to ZERO each time it is enabled (no matter whether or not the original COUNTER input is enabled); this is a **RESET** input.
- Optional:
  - Develop a new version of your counter that **counts up in multiples of two.** ☺
  - Develop a version of your counter that **counts forwards and backwards by one.** ☺

Congratulate yourself once you've got to here.

Well done for doing extra! ☺

IMPLEMENTING FINITE STATE MACHINES

Implementing FSMs

- Registers to store state, i.e. the state holding logic
  - How many bits are needed?
- Logic to implement δ (next state) and ω (output) functions
  - How much logic do we need?
  - How can we realize this logic?
  - Is this the optimal solution?

Discuss these questions further during the workshop

Optimising FSM logic

There are several things to consider in the design of an FSM to ensure that it is both **correct** and **efficient**.

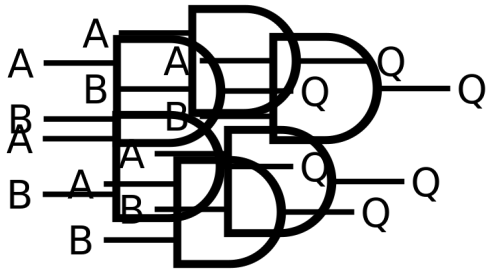
- Choosing the type of machine (Moore/Mealy)
  - Determines how states and output functions are computed.
  - Ensures outputs happen exactly when intended.
- Selecting states
  - Minimise the number of states to keep the **state space** small.
  - Good state design will facilitate construction of FSM.
- Simplification of δ (next state) and ω (output) functions.
  - Fewer gates required.
  - Cheaper, smaller, faster.

Simplifying Boolean expressions

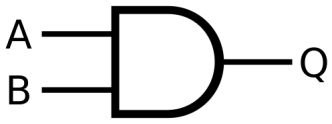
- Recall lectures 2 and 3.
- We used **axioms** in Boolean algebra to show functional completeness of NAND and NOR. Example:

$$\neg(\neg(A \wedge B)) \equiv A \wedge B$$

- The **right side** of the above expression is a simplified form of the **left side**.
- *If* we can simplify complicated Boolean expressions, we can build a **smaller combinatorial circuit** to implement it.



Why use lots of gates…



…when one is all you need?

Karnaugh Maps

- Karnaugh Maps are useful for identifying minimal forms of Boolean functions.

Karnaugh Maps

- Karnaugh Maps are useful for identifying minimal forms of Boolean functions.
- What are some possible ways of expressing the below function, F(A, B) ?

A	B	F(A,B)
0	0	0
0	1	0
1	0	1
1	1	1

F(A,B) =

Karnaugh Maps

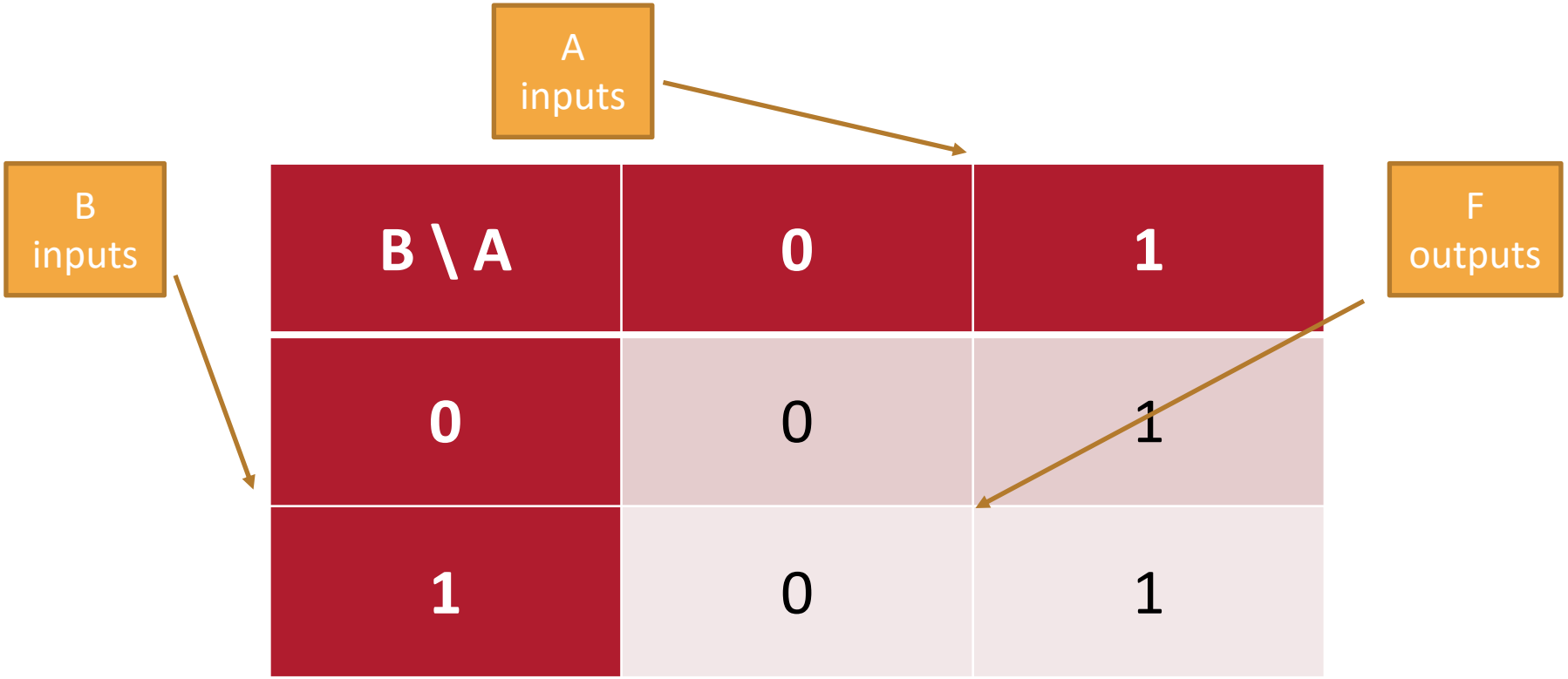
- Karnaugh Maps are useful for identifying minimal forms of Boolean functions.
- What are some possible ways of expressing the below function, F(A, B) ?

A	B	F(A,B)
0	0	0
0	1	0
1	0	1
1	1	1

$F(A,B) = (A \wedge \neg B) \vee (A \wedge B)$

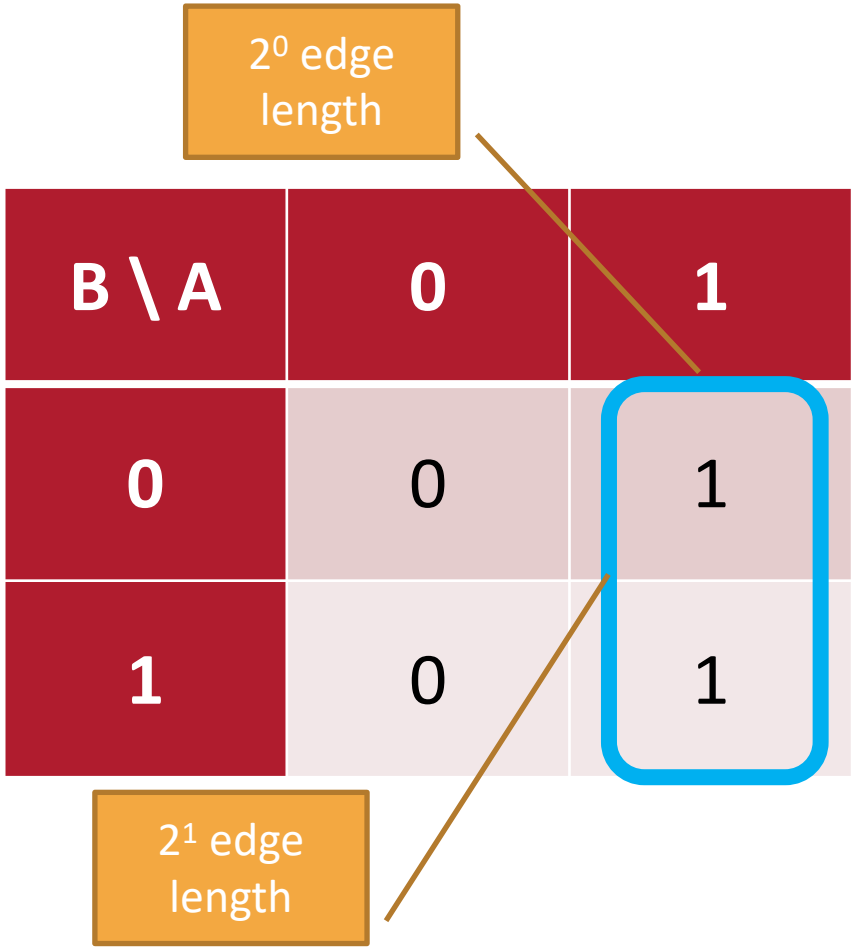
K-map example

A	B	F(A,B)
0	0	0
0	1	0
1	0	1
1	1	1



K-map example

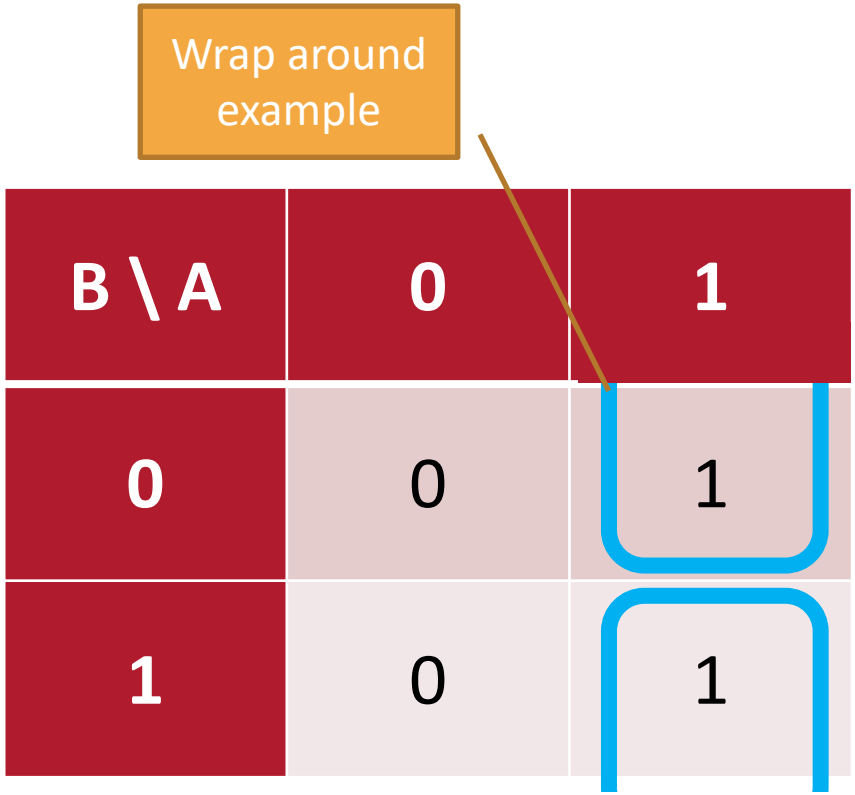
A	B	F(A,B)
0	0	0
0	1	0
1	0	1
1	1	1



- Minterm form (DNF)
- Form groups of 1s where the length of the edges are 2^n
- The edges may **wrap around**.

K-map example

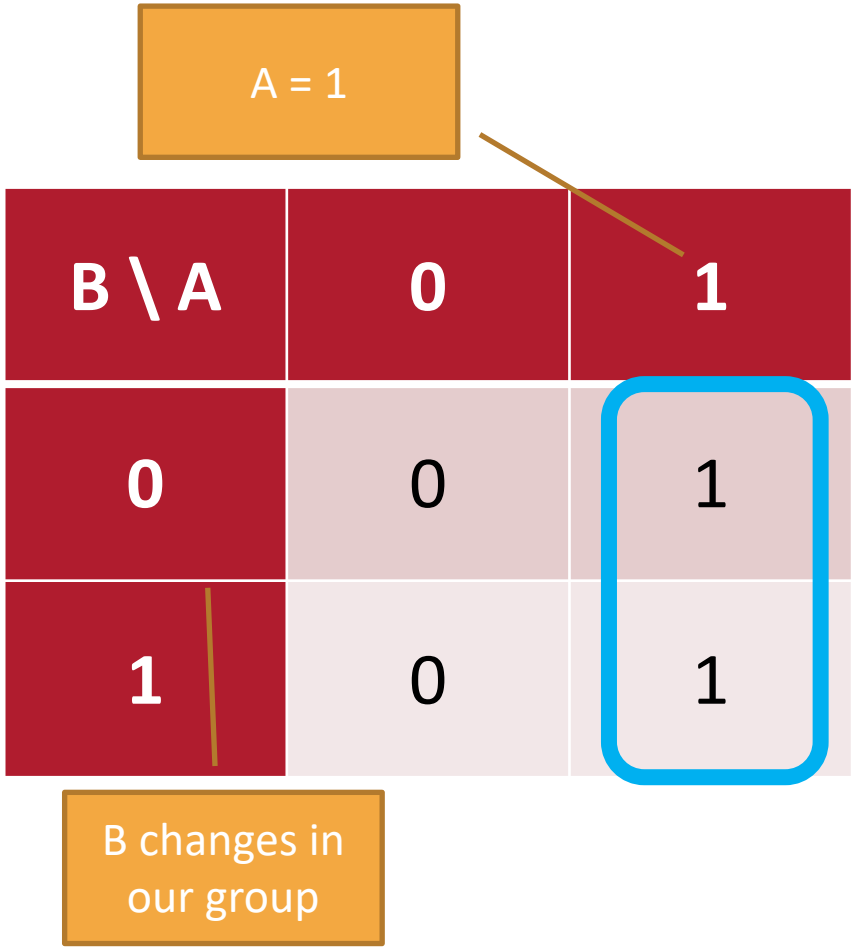
A	B	F(A,B)
0	0	0
0	1	0
1	0	1
1	1	1



- Minterm form (DNF)
- Form groups of 1s where the length of the edges are 2^n
- The edges may **wrap around**.

K-map example

A	B	F(A,B)
0	0	0
0	1	0
1	0	1
1	1	1



- For each group, write down the variables that do not change, in that group.
- Express in minterm (sum of products) form.

F = A

K-map bigger example

A	B	C	D	F
0	0	0	0	0
0	0	0	1	0
0	0	1	0	1
0	0	1	1	0
0	1	0	0	0
0	1	0	1	1
0	1	1	0	1
0	1	1	1	1
1	0	0	0	1
1	0	0	1	0
1	0	1	0	1
1	0	1	1	0
1	1	0	0	0
1	1	0	1	1
1	1	1	0	1
1	1	1	1	1

Note the order of values (Gray coding)

Gray code: two successive values differ in only one binary digit.

CD\AB	00	01	11	10
00	0	0	0	1
01	0	1	1	0
11	0	1	1	0
10	1	1	1	1

K-map bigger example

A	B	C	D	F
0	0	0	0	0
0	0	0	1	0
0	**0**	**1**	**0**	**1**
0	0	1	1	0
0	1	0	0	0
0	**1**	**0**	**1**	**1**
0	**1**	**1**	**0**	**1**
0	**1**	**1**	**1**	**1**
1	**0**	**0**	**0**	**1**
1	0	0	1	0
1	**0**	**1**	**0**	**1**
1	0	1	1	0
1	1	0	0	0
1	**1**	**0**	**1**	**1**
1	**1**	**1**	**0**	**1**
1	**1**	**1**	**1**	**1**

Note the order of values (Gray coding)

Gray code: two successive values differ in only one binary digit.

CD\AB	00	01	11	10
00	0	0	0	1
01	0	1	1	0
11	0	1	1	0
10	1	1	1	1

K-map bigger example

CD\AB	00	01	11	10
00	0	0	0	1
01	0	1	1	0
11	0	1	1	0
10	1	1	1	1

K-map bigger example

CD\AB	00	01	11	10
00	0	0	0	1
01	0	1	1	0
11	0	1	1	0
10	1	1	1	1

Overlapping groups is OK… and **necessary** to get the simplest expression.

K-map bigger example

CD\AB	00	01	11	10
00	0	0	0	1
01	0	1	1	0
11	0	1	1	0
10	1	1	1	1

Overlapping groups is OK... and **necessary** to get the simplest expression.

$F = (A \wedge (\neg B) \wedge (\neg D)) \vee (B \wedge D) \vee (C \wedge (\neg D))$

$F = A\neg B\neg D + BD + C\neg D$

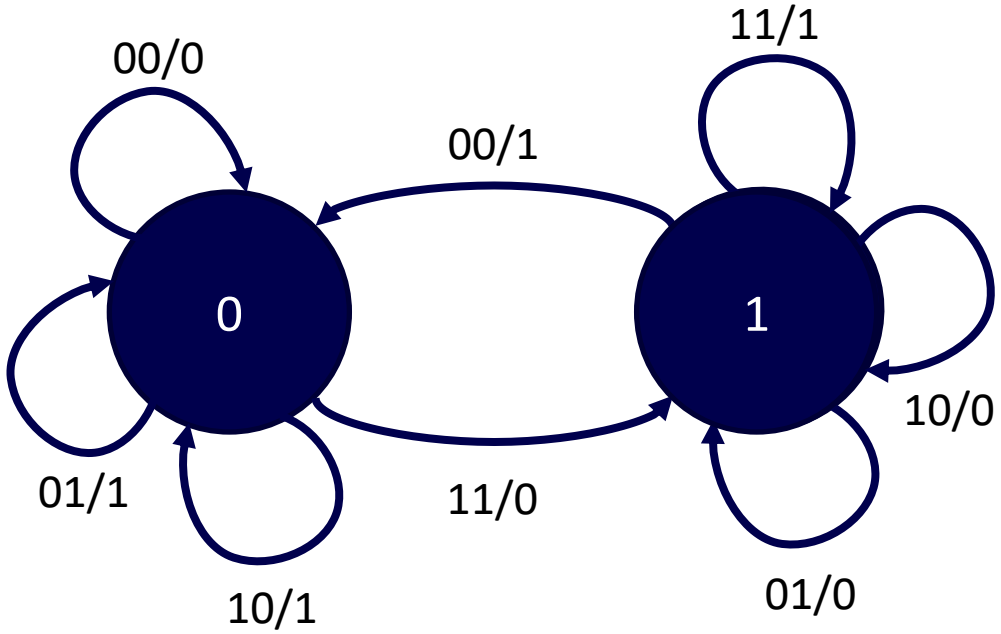
Simplification algorithmically

- Quine-McCluskey algorithm can be used to simplify Boolean functions.
- Functionally equivalent to Karnaugh Maps.
- Easier to implement as an algorithm.
  - Can be used in logic synthesis tools to produce efficient implementations of logic functions.

OPTIMISING
THE SERIAL BINARY ADDER

FSM example: Serial Adder

Current State	Input		Next State	Output
S	A_i	B_i	S_{next} (S')	Sum_i
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1



$S_{next} = \neg S \wedge A_i \wedge B_i \vee S \wedge \neg A_i \wedge B_i \vee S \wedge A_i \wedge \neg B_i \vee S \wedge A_i \wedge B_i$

$Sum_i = \neg S \wedge \neg A_i \wedge B_i \vee \neg S \wedge A_i \wedge \neg B_i \vee S \wedge \neg A_i \wedge \neg B_i \vee S \wedge A_i \wedge B_i$

FSM example: Serial Adder

Current State	Input		Next State	Output
S	A_i	B_i	S_{next} (S')	Sum_i
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

S_{next}

$S\backslash A_iB_i$	00	01	11	10
0				
1				

Sum_i

$S\backslash A_iB_i$	00	01	11	10
0				
1				

$$S_{next} = \neg S \wedge A_i \wedge B_i \vee S \wedge \neg A_i \wedge B_i \vee S \wedge A_i \wedge \neg B_i \vee S \wedge A_i \wedge B_i$$

$$Sum_i = \neg S \wedge \neg A_i \wedge B_i \vee \neg S \wedge A_i \wedge \neg B_i \vee S \wedge \neg A_i \wedge \neg B_i \vee S \wedge A_i \wedge B_i$$

FSM example: Serial Adder

Current State	Input		Next State	Output
S	A_i	B_i	S_{next} (S')	Sum_i
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

S_{next}

$S\backslash A_iB_i$	00	01	11	10
0	0	0	1	0
1	0	1	1	1

Sum_i

$S\backslash A_iB_i$	00	01	11	10
0	0	1	0	1
1	1	0	1	0

$$S_{next} = \neg S \wedge A_i \wedge B_i \vee S \wedge \neg A_i \wedge B_i \vee S \wedge A_i \wedge \neg B_i \vee S \wedge A_i \wedge B_i$$

$$Sum_i = \neg S \wedge \neg A_i \wedge B_i \vee \neg S \wedge A_i \wedge \neg B_i \vee S \wedge \neg A_i \wedge \neg B_i \vee S \wedge A_i \wedge B_i$$

FSM example: Serial Adder

Current State	Input		Next State	Output
S	A_i	B_i	S_{next} (S')	Sum_i
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

S_{next}

$S\backslash A_iB_i$	00	01	11	10
0	0	0	1	0
1	0	1	1	1

Sum_i

$S\backslash A_iB_i$	00	01	11	10
0	0	1	0	1
1	1	0	1	0

$$S_{next} = \neg S \wedge A_i \wedge B_i \vee S \wedge \neg A_i \wedge B_i \vee S \wedge A_i \wedge \neg B_i \vee S \wedge A_i \wedge B_i$$

$$Sum_i = \neg S \wedge \neg A_i \wedge B_i \vee \neg S \wedge A_i \wedge \neg B_i \vee S \wedge \neg A_i \wedge \neg B_i \vee S \wedge A_i \wedge B_i$$

FSM example: Serial Adder

Current State	Input		Next State	Output
S	A_i	B_i	S_{next} (S')	Sum_i
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

S_{next}

$S\backslash A_iB_i$	00	01	11	10
0	0	0	1	0
1	0	1	1	1

Sum_i

$S\backslash A_iB_i$	00	01	11	10
0	0	1	0	1
1	1	0	1	0

$$S_{next} = \neg S \wedge A_i \wedge B_i \vee S \wedge \neg A_i \wedge B_i \vee S \wedge A_i \wedge \neg B_i \vee S \wedge A_i \wedge B_i$$

$$=$$

$$Sum_i = \neg S \wedge \neg A_i \wedge B_i \vee \neg S \wedge A_i \wedge \neg B_i \vee S \wedge \neg A_i \wedge \neg B_i \vee S \wedge A_i \wedge B_i$$

FSM example: Serial Adder

Current State	Input		Next State	Output
S	A_i	B_i	S_{next} (S')	Sum_i
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

S_{next}

$S\backslash A_iB_i$	00	01	11	10
0	0	0	1	0
1	0	1	1	1

Sum_i

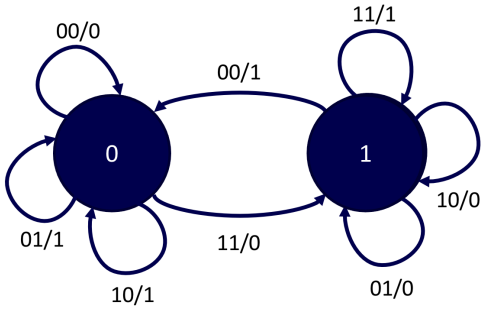
$S\backslash A_iB_i$	00	01	11	10
0	0	1	0	1
1	1	0	1	0

$$S_{next} = \neg S A_i B_i + S \neg A_i B_i + S A_i \neg B_i + S A_i B_i$$
$$= A_i B_i + S B_i + S A_i$$

$$Sum_i = \neg S \neg A_i B_i + \neg S A_i \neg B_i + S \neg A_i \neg B_i + S A_i B_i$$

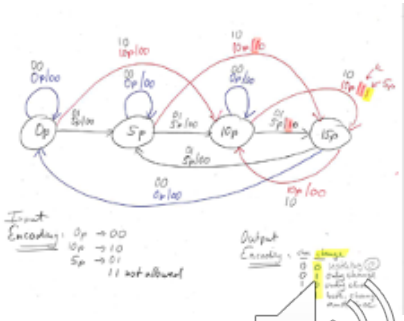
Summary

- FSMs
  - State diagrams and state transition tables
  - Next state & output functions
  - Moore/Mealy machines
- Boolean function optimisation
  - Karnaugh maps
- Workshop on FSMs
  - Counter machine: 3-bit counter
  - Vending machine control logic



S_{next}

$S\backslash A_iB_i$	00	01	11	10
0	0	0	1	0
1	0	1	1	1



So far in this unit

Foundations	• Data representation, logic, Boolean algebra.
Building blocks	• Transistors, transistor based logic, simple devices, storage.
Modules	• Memory, **simple controllers, FSMs**, processors and execution.
Programming	• Machine code, assembly, high-level languages, compilers.
Wrap-up	• Operating systems, energy aware computing.

Next in this unit

Foundations	• Data representation, logic, Boolean algebra.
Building blocks	• Transistors, transistor based logic, simple devices, storage.
Modules	• Memory, simple controllers, FSMs, **processors and execution.**
Programming	• Machine code, assembly, high-level languages, compilers.
Wrap-up	• Operating systems, energy aware computing.

VENDING MACHINE FSM DEVELOPMENT TASK

The slides that follow illustrate some solutions previously developed for this task.

***

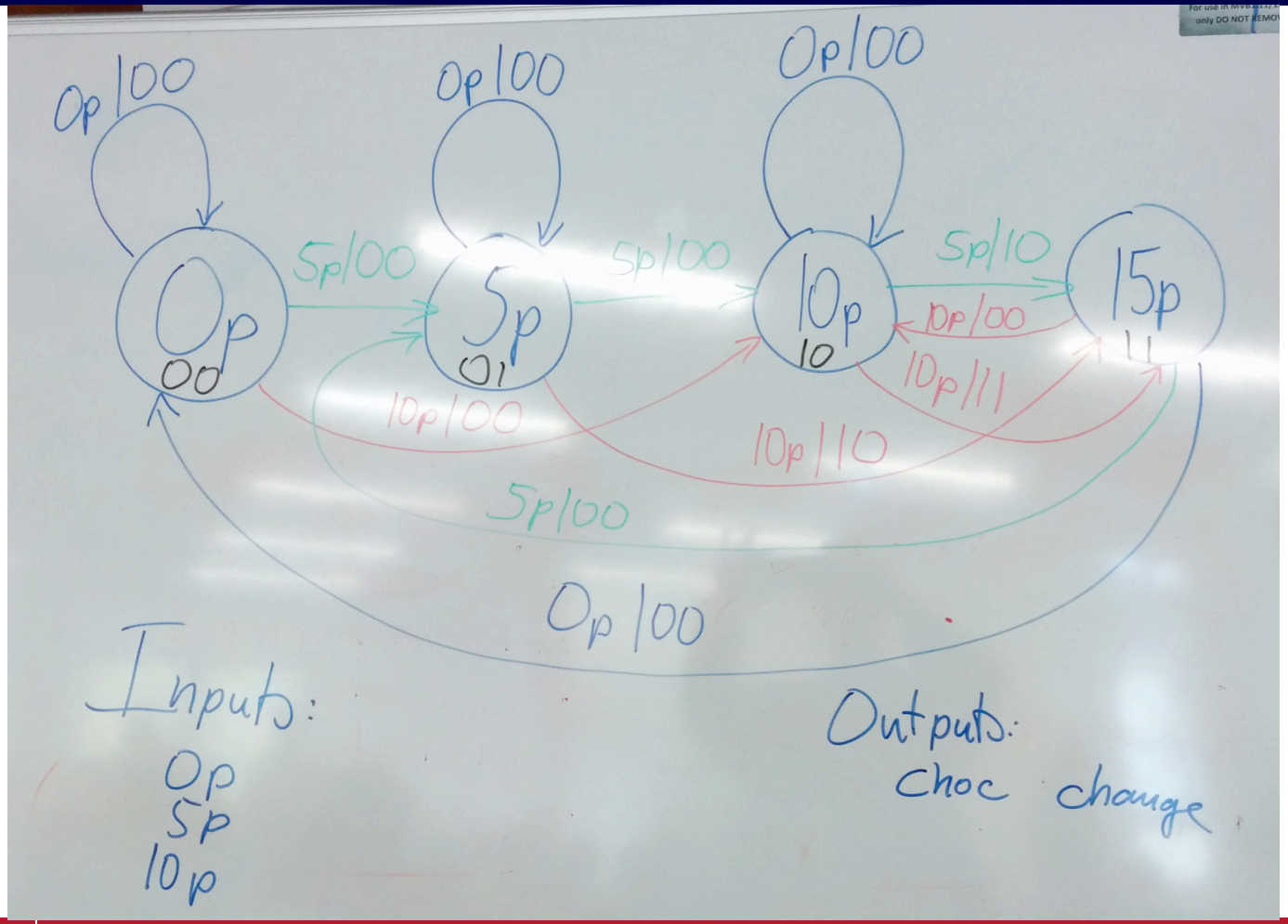
You may use these for inspiration or for comparison, but please do develop your own solution before looking at the next slide(s).

Vending Machine FSM

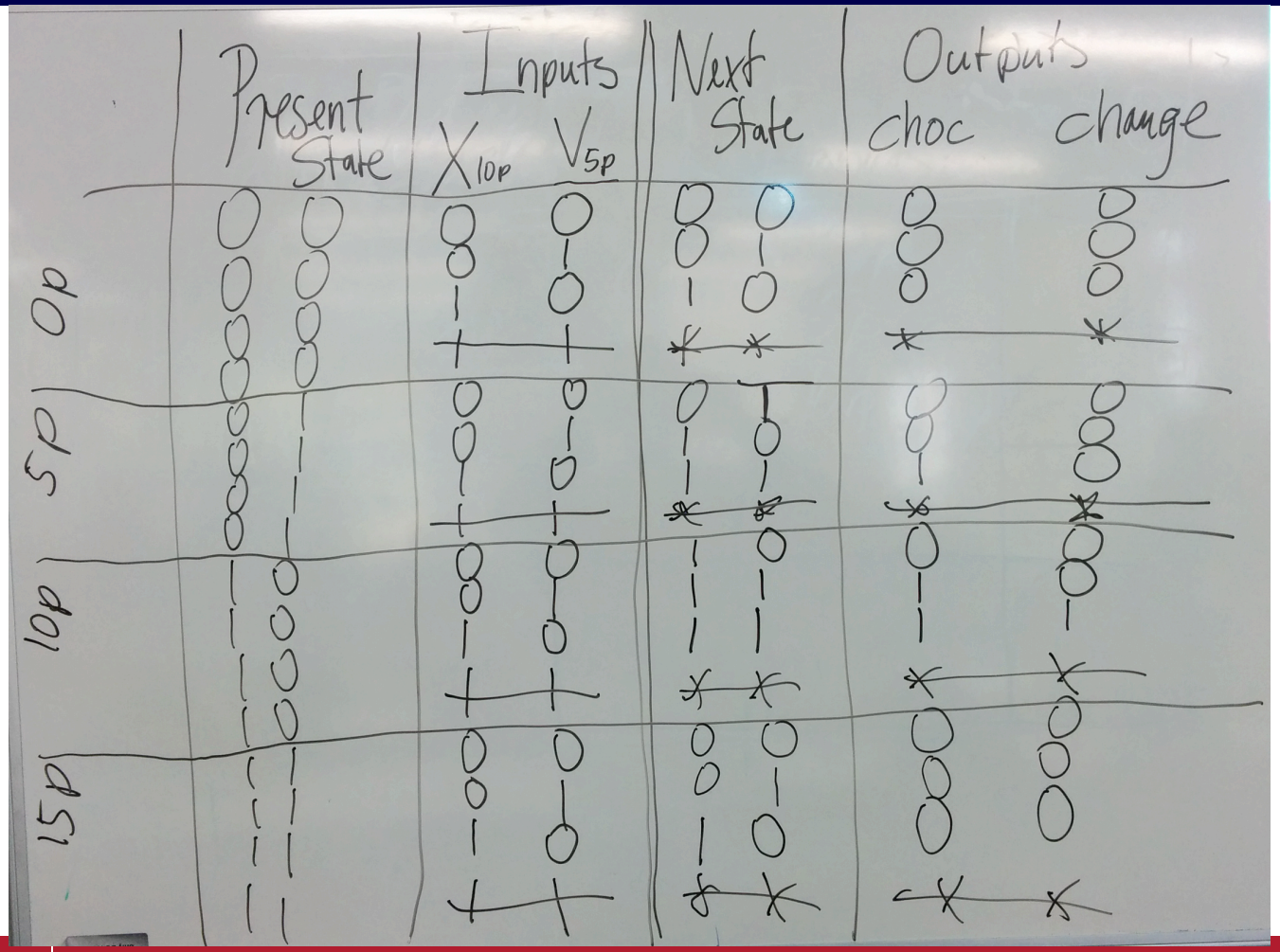
- Vending machine
  - gives you a choc bar for 15p
  - takes 5p and 10p coins, one at a time
  - gives 5p change
- FSM design for a simple vending machine including
  - State-transition diagram
  - State-transition table
  - Next state logic and output logic
  - Optimization via Karnaugh maps

This is your task to develop during the workshop later.
Please bring something to write!

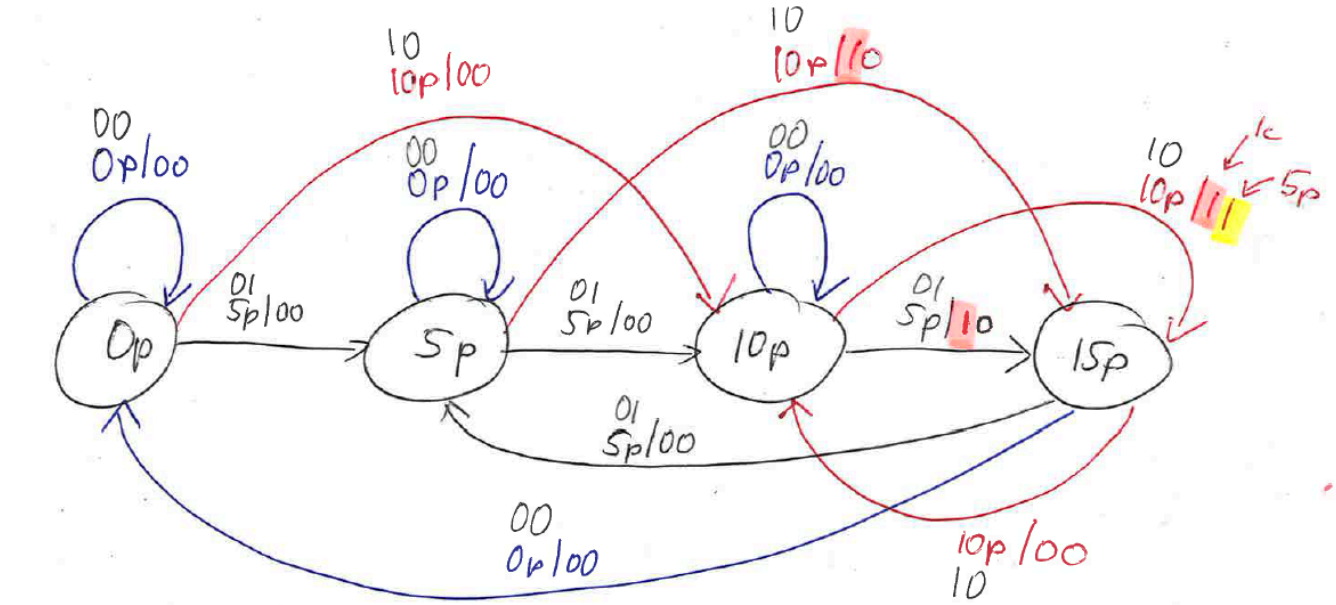
VM FSM State Transition Diagram



VM State Transition Table



VM State-Transition Diagram

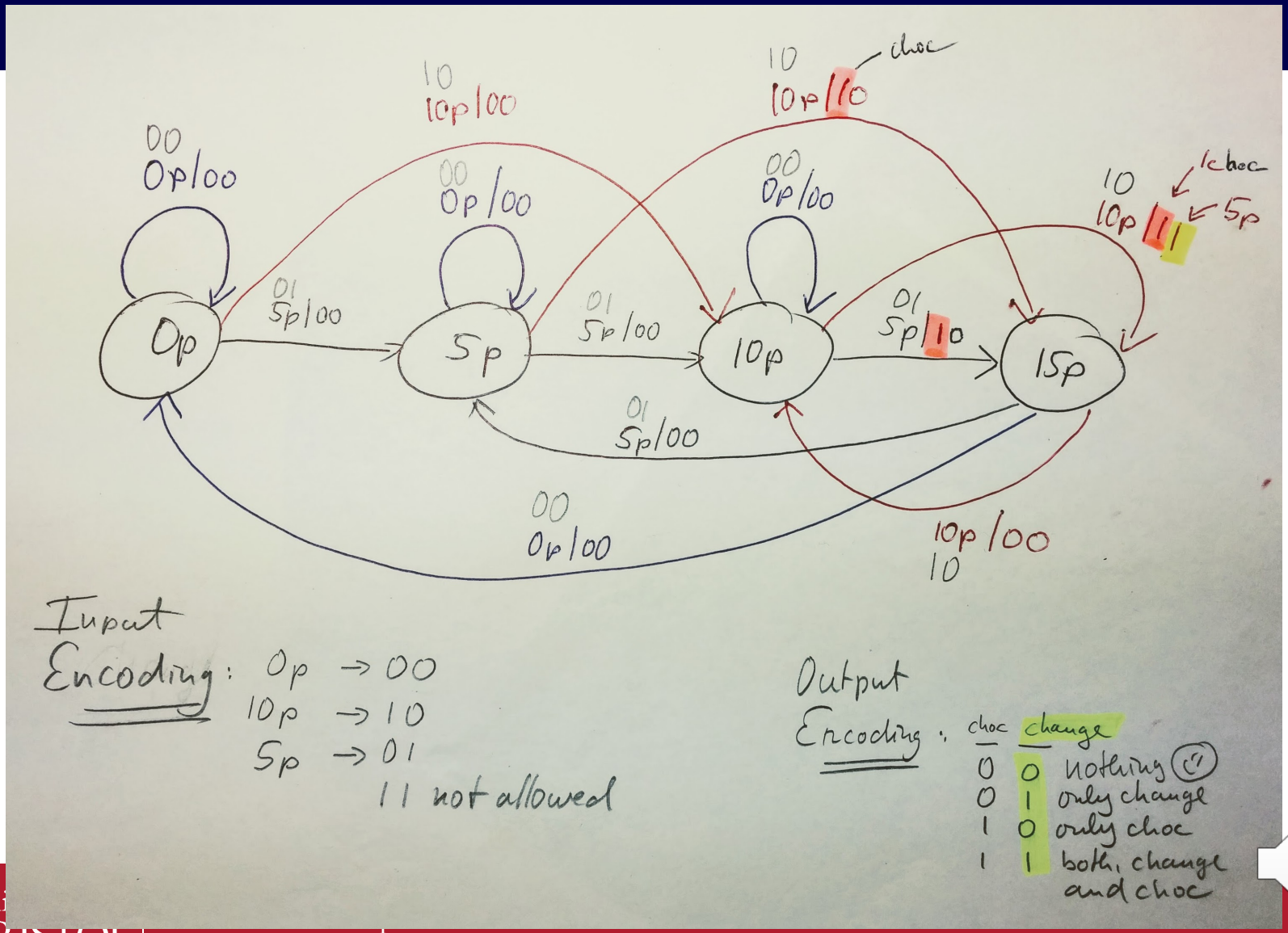


Input Encoding: 0p → 00
10p → 10
5p → 01
11 not allowed

Output Encoding:

choc	change	
0	0	nothing
0	1	only change
1	0	only choc
1	1	both, change and choc

VM State-Transition Diagram



VM State-Transition Table

State encoding using two bits, S_1S_0:

0p = 00

5p = 01

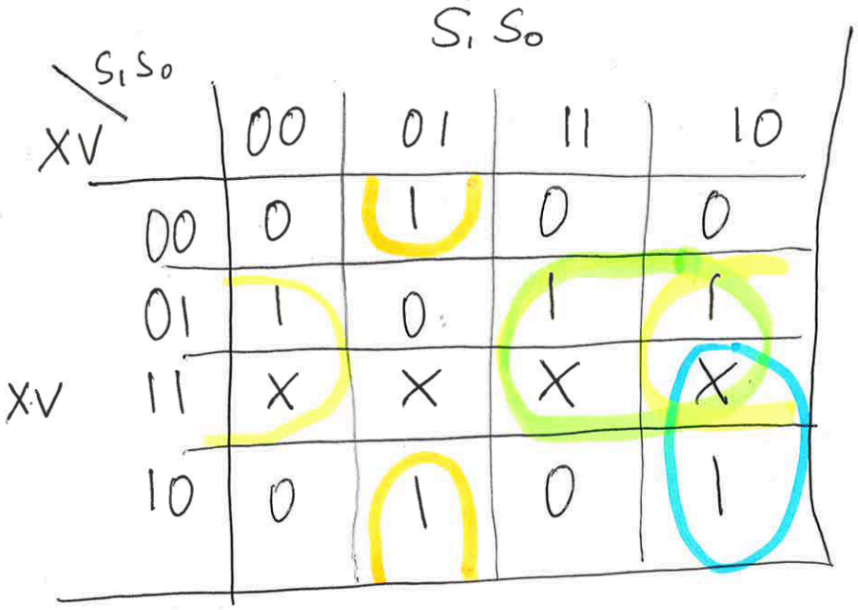
10p = 10

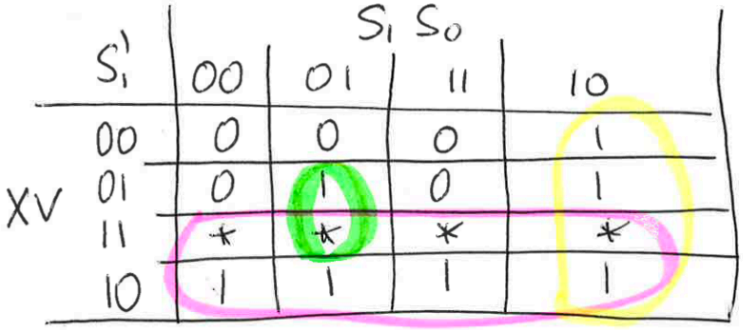
15p = 11

Vending Machine

	Present State S_1	Present State S_0	Input X=10p	Input V=5p	Next State S_1'	Next State S_0'	Output choc	Output change
0p	0	0	0	0	0	0	0	0
	0	0	0	1	0	1	0	0
	0	0	1	0	0	1	0	0
	0	0	1	1	X	X	—	—
5p	0	1	0	0	0	1	0	0
	0	1	0	1	0	1	0	0
	0	1	1	0	1	0	1	0
	0	1	1	1	X	X	—	—
10p	1	0	0	0	1	0	0	0
	1	0	0	1	1	1	1	1
	1	0	1	0	1	1	1	1
	1	0	1	1	X	X	—	—
15p	1	1	0	0	0	0	0	0
	1	1	0	1	0	1	0	0
	1	1	1	0	1	0	0	0
	1	1	1	1	X	X	—	—

VM Next State Logic Optimization





$$S_1' = X + S_1 \bar{S}_0 + \bar{S}_1 S_0 V$$

$$S_0' = \bar{S}_1 S_0 \bar{V} + S_1 \bar{S}_0 X + \bar{S}_0 V + S_1 V$$

A new vending machine

Q1	Q0	10p	5p	Q1'	Q0'	choc	change
0	0	0	0	0	0	0	0
0	0	0	1	0	1	0	0
0	0	1	0	1	0	0	0
0	0	1	1	x	x	x	x
0	1	0	0	0	1	0	0
0	1	0	1	1	0	0	0
0	1	1	0	0	0	1	0
0	1	1	1	x	x	x	x
1	0	0	0	1	0	0	0
1	0	0	1	0	0	1	0
1	0	1	0	0	0	1	1
1	0	1	1	x	x	x	x
1	1	?	?	?	?	?	?

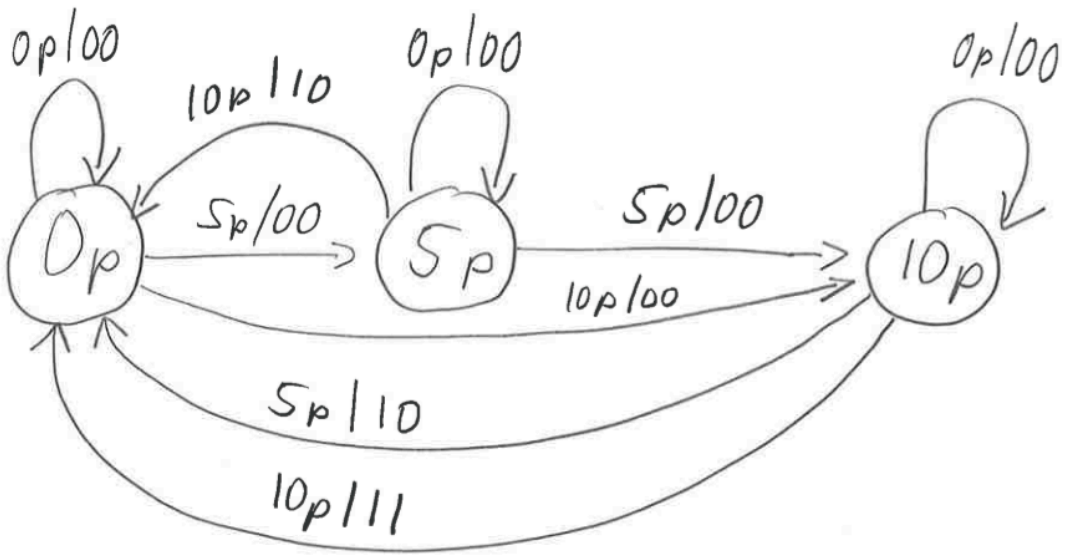
A new vending machine

Q1	Q0	10p	5p	Q1'	Q0'	choc	change
0	0	0	0	0	0	0	0
0	0		1	0	1	0	0
0							
0							
0							
0							
0							
0							
1							
1	0	0	1	0	0	1	0
1	0	1	0	0	0	1	1
1	0	1	1	x	x	x	x
1	1	?	?	?	?	?	?

- Draw the state-transition diagram for this FSM
- How does it differ from the FSM developed earlier?
- Can you formulate logic expressions for the next state and the output logic?

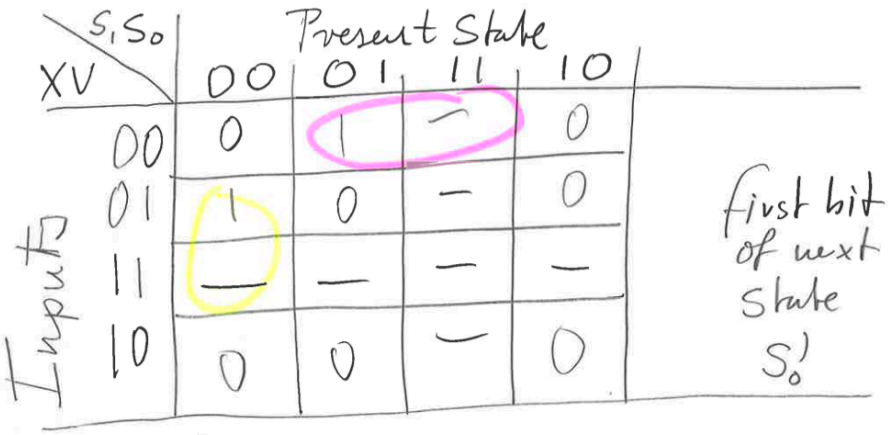
A new vending machine

State-transition diagram

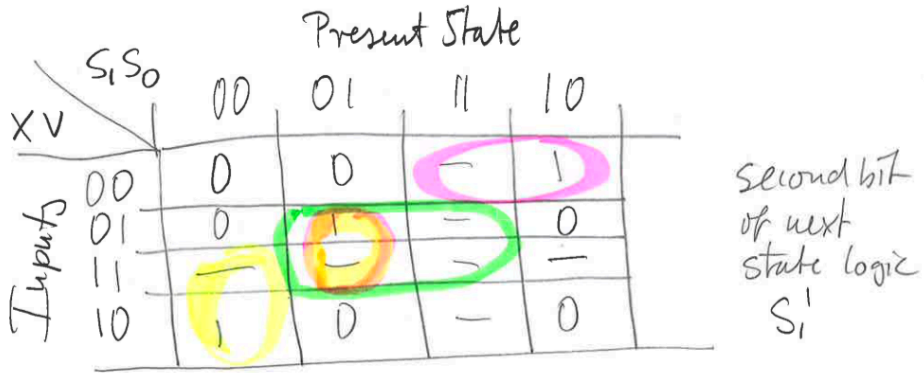


	Present State S_1 S_0	Inputs X_{10p} V_{5p}	Next State S_1' S_0'	Outputs choc change
0p	0 0	0 0	0 0	0 0
	0 0	0 1	0 1	0 0
	0 0	1 0	1 0	0 0
	0 0	1 1	—	—
5p	0 1	0 0	0 1	0 0
	0 1	0 1	1 0	0 0
	0 1	1 0	0 0	1 0
	0 1	1 1	—	—
10p	1 0	0 0	1 0	0 0
	1 0	0 1	0 0	1 0
	1 0	1 0	0 0	1 1
	1 0	1 1	—	—
(15p)	1 1	0 0	—	—
	1 1	0 1	—	—
	1 1	1 0	—	—
	1 1	1 1	—	—

Next State Logic Optimization



$$S_0' = \bar{S}_1\bar{S}_0V + S_0\bar{X}\bar{V}$$



$$S_1' = \bar{S}_1\bar{S}_0X + S_1\bar{X}\bar{V} + V\bar{S}_1S_0$$

$$S_1' = \bar{S}_1\bar{S}_0X + S_1\bar{X}\bar{V} + VS_0$$

We can do better!