Australian National University

Week 4
Finite-State Machines
Video Lecture

ENGN4213/6213

Digital Systems and Microprocessors

What’s in this lecture

- Finite-State Machine (FSM) concepts and definitions
- Representation of state machines
- Designing state machines

Australian National University

Resources

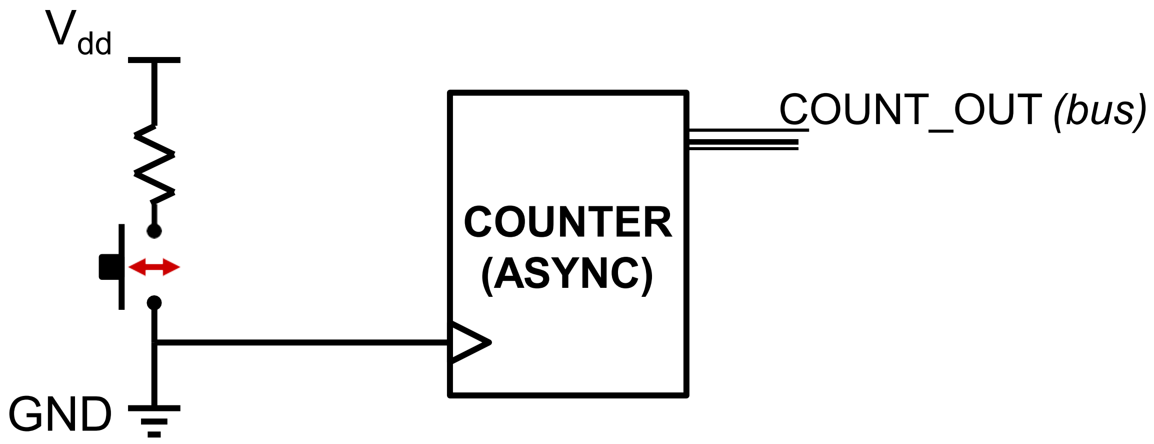
- Wakerly (5 edition) Ch.9: introduction to some general concepts and definitions
    - Wakerly Ch. 9.1-9.2 for analysis of state machines
    - Wakerly Ch. 9.3 for FSM design with state tables
    - Wakerly Ch. 9.4 for FSM design with state diagrams
    - Wakerly Ch. 9.6 for FSM design with Verilog
- Wakerly (5 edition) Ch.12 for FSM design with examples
    - Wakerly Ch. 12.7 for an example of a FSM machine implementing a “guessing game”
    - – (not strictly required: read it if it helps your understanding)
    - Wakerly Ch.12.9 for decomposing state machines

The concept of ***state***

- We have already learned that in a *sequential circuit*, the output (y) is a function (f_S) of current and past inputs (u)
- This is opposed to *combinational circuits*, where inputs and outputs are related through a boolean function of some sort (f_c) (*truth table*).
- Mathematically, this could be written as
  - For a combinational circuit $y(t)=f_C(u(t))$
  - For a sequential circuit $y(t)=f_S(u(t),u(t-1),u(t-2),\ldots)$
- But how many time steps of input history are required to describe a system completely?
  - This may vary depending on the system.
  - For some systems a full input history might be required, making for a rather intractable description

The concept of ***state*** (2)

- **Example:**
  - consider a counter circuit: every time a push button is pressed the counter is incremented by 1.



  - the output of such a system depends on the full history of inputs (how many times the button has been pushed). Also, since the timing of the button pushes is unknown, how can f_S be established unambiguously?
  - A better way of describing sequential systems is required. That's where the concept of ***state*** becomes useful.

The concept of ***state*** (3)

- Your textbook (Ch.9) gives a very good **definition** of the concept of *state:*

  The state of a sequential circuit is a collection of state variables whose values at any time contain all the information about the past necessary to account for the circuit's future behaviour.

- In the counter example, a valid state variable ***s*** is the value of the count. The output can be unambiguously described as a function of the current count and input: $y(t) = f_S(s(t),u(t)) = s(t) + u(t)$

The concept of ***state*** (4)

- You have already encountered the concept of state in your earlier studies if you have learned about ***dynamical systems***. You will most likely know of state-space models. For a linear discrete-time dynamical system a common formulation is:

  s(t+1) = As(t) + Bu(t) ***next state equation***

  y(t) = Cs(t) + Du(t) ***output equation***

- Indeed, we will use "next-state" concepts in our treatment of sequential circuits. A sequential circuit **is** a dynamical system (although not necessarily a linear system).

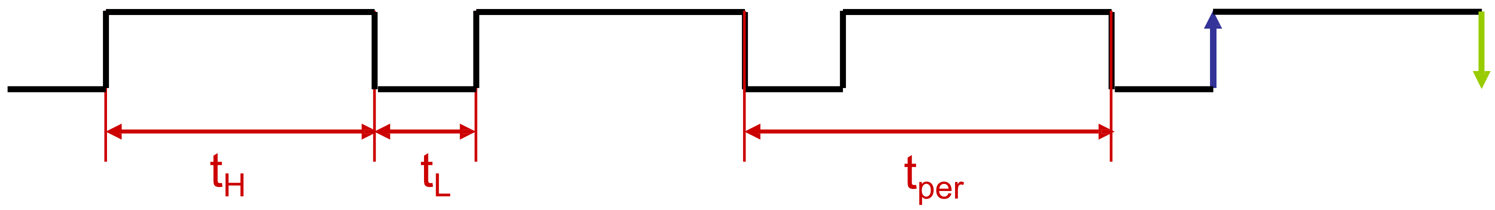
Finite-State Machines

- In a sequential logic circuit, the state variables are **binary variables**.
  - it means that a circuit with n state variables will have 2^n possible states

- In sequential circuits the number of variables can be large but is finite. Hence the name ***finite-state machine.***
  - All sequential logic circuits can be interpreted as FSM
  - As your textbook suggests it is possible to make a theoretical case for a non-finite-state machine (e.g. a Turing machine), but for this course we are interested in practical, finite-state designs.

Clock signals (synchronous systems)

- We have already talked about these, just a few more specifications:



  - Clock period = t_{per} , frequency = 1/ t_{per}
  - For a synchronous system driven by an "***active high***" clock:
    - **State transitions** occur at ***rising*** clock edges (↑)
    - The **duty cycle** of the clock is calculated as t_H/t_{per}
  - For a synchronous system driven by an "***active low***" clock:
    - **State transitions** occur at ***falling*** clock edges (↓)
    - The **duty cycle** of the clock is calculated as t_L/t_{per}

State-machine description

- **Mathematical description**
  - **Mealy machines**
    - The output at time t depends on both the state and input at time t

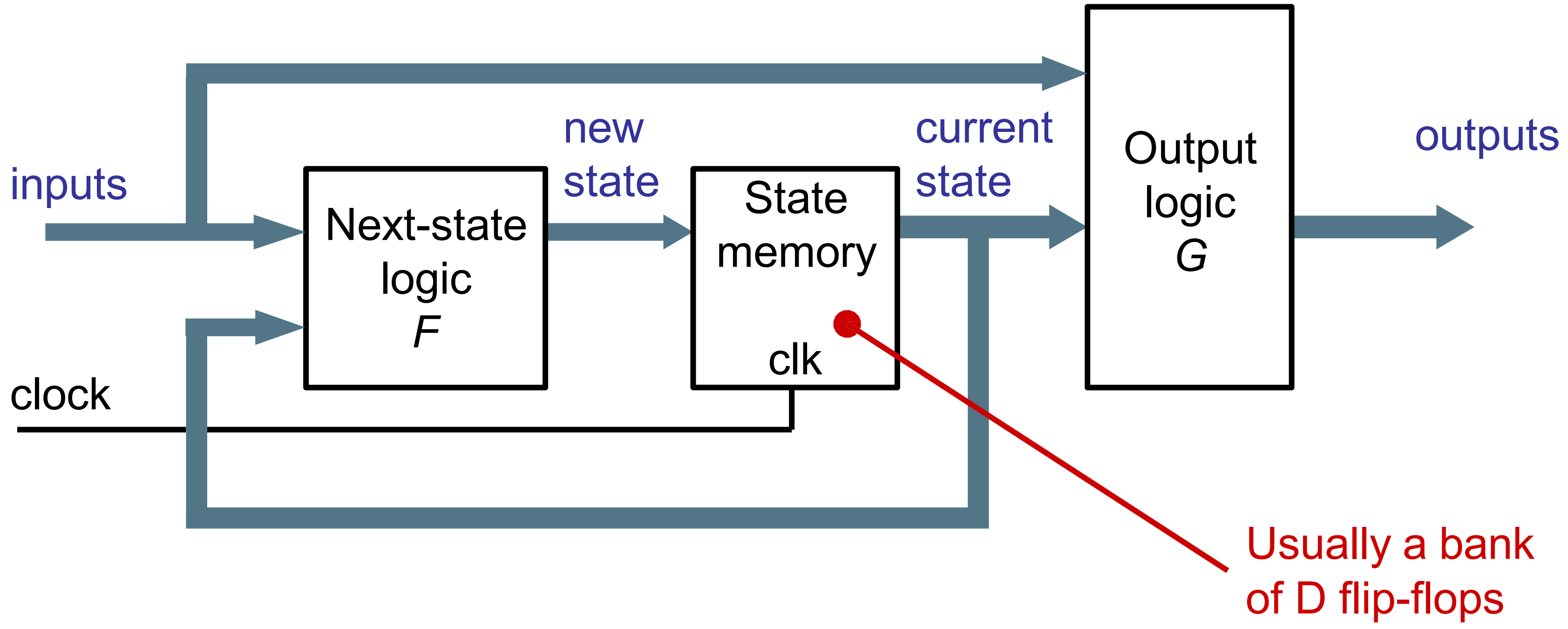
$s(t+1) = F(s(t),u(t))$	***next state equation***
$y(t) = G(s(t),u(t))$	***output equation***

  - **Moore machines**
    - The output at time t depends only on the state at time t

$s(t+1) = F(s(t),u(t))$	***next state equation***
$y(t) = G(s(t))$	***output equation***

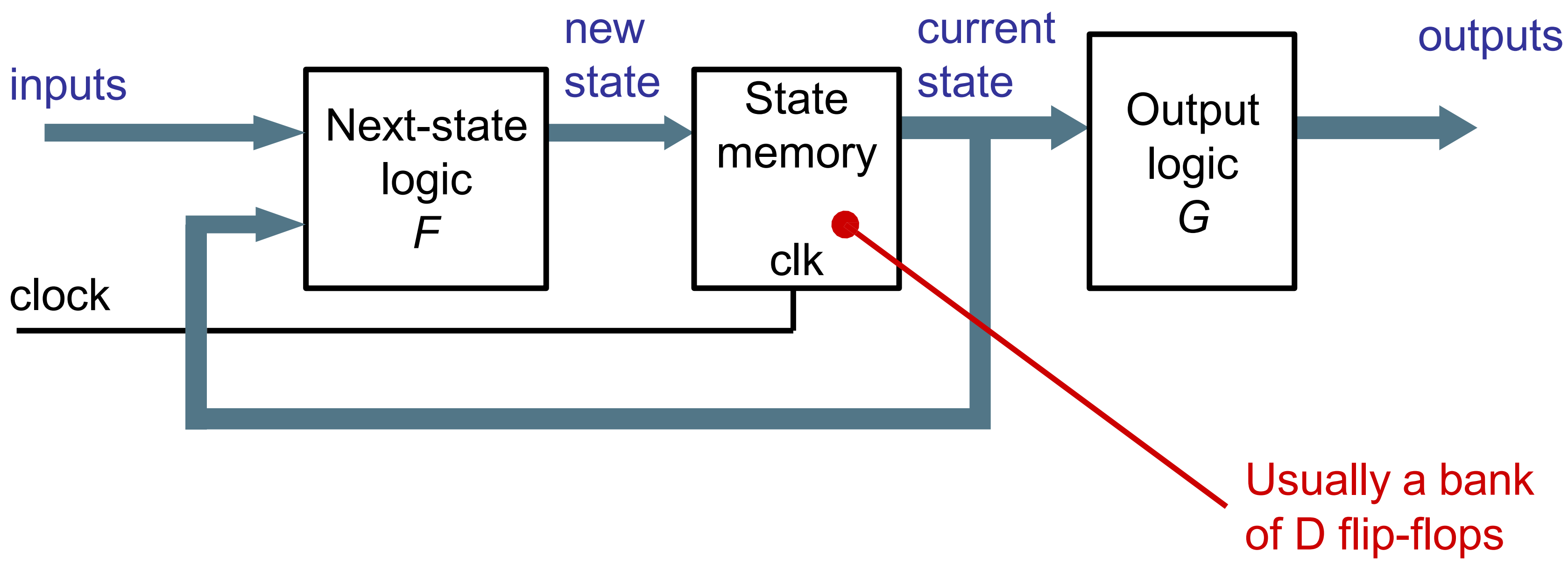
State-machine description (2)

- **Block-Diagram representation**
  - **Mealy machines**



State-machine description (3)

- **Block-Diagram representation**
  - **Moore machines**



Finite state machine structure

- The **state memory** is the true sequential part of a design. Generally implemented as n flip flops for 2^n states. (*why?*)
- The flip-flops undergo their transitions (change of state) on the active edge of a common clock – **synchronous state machine**
- The next state is determine by the **next state logic F** which is a function of the inputs and the current state.
- The **output logic G** determines the output as a function of the current state and (Mealy machine) the inputs.
- Thus we see that **a synchronous FSM consists of two combinational blocks** (the input and output logic blocks) **and one sequential block** (the state memory block)

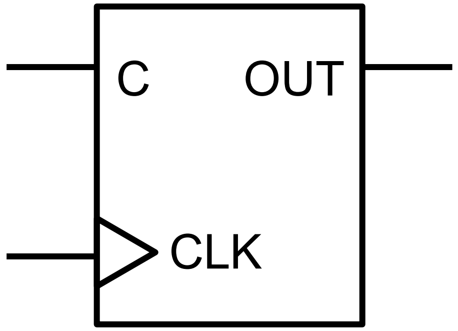
State diagrams

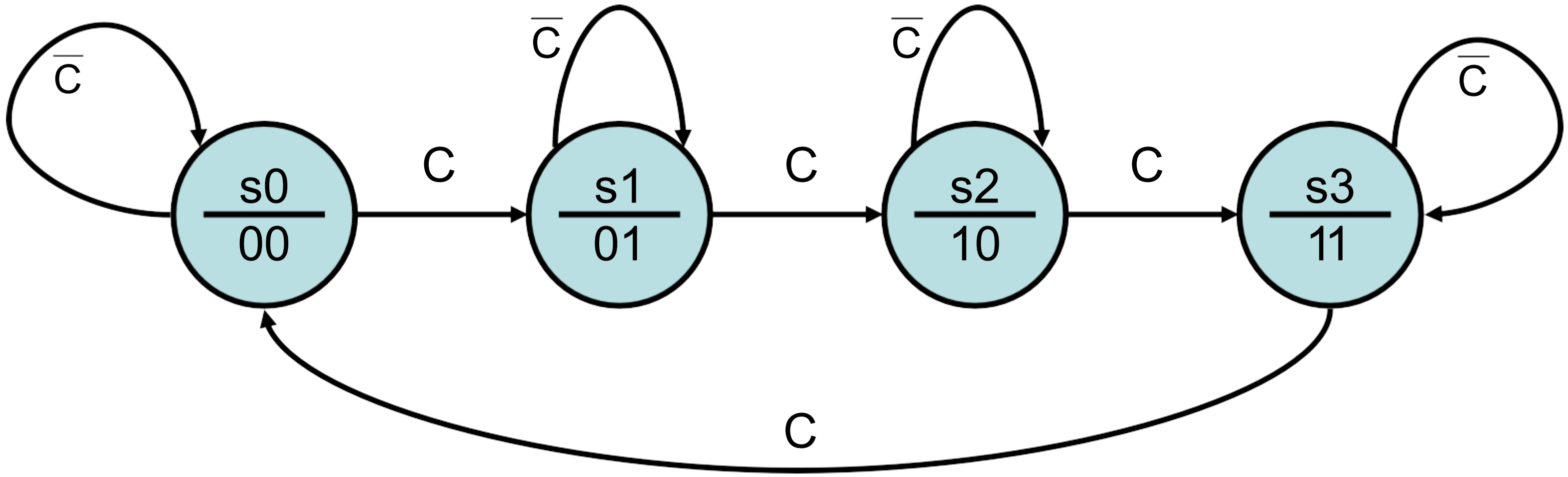
- **A way to represent the FSM in one diagram**
  - **States** are symbolised by **a bubble** in the form of a circle or a box.
  - **Loops or branches** between the state bubbles represent **state transitions** with the input condition for the transition written on them
  - For a *Mealy machine* the outputs depend on both the input and the state. Thus they are written underneath the inputs on the loops
  - The outputs of a *Moore machine* can be written inside the state bubbles

State diagrams (2)

- **Example:**
  a 2-bit counter (state=current count)

  A Moore or Mealy machine?





Tables

- A FSM machine can also be described using tables:
  - **transition tables,**
  - **state tables**
  - **state/output tables**
- **Transition tables** represent the relationship between the current state value and next state value as a function of the input
- **State tables** are very similar but more intuitive for complex states as the states are assigned a mnemonic description
- **State/output tables** also include the relationship between the state and the output. These can become complicated for Mealy machines as the output is also a function of the input.
- You can draw one or more of these tables, or condense them together as one single *input* | *next state* | *output* table.

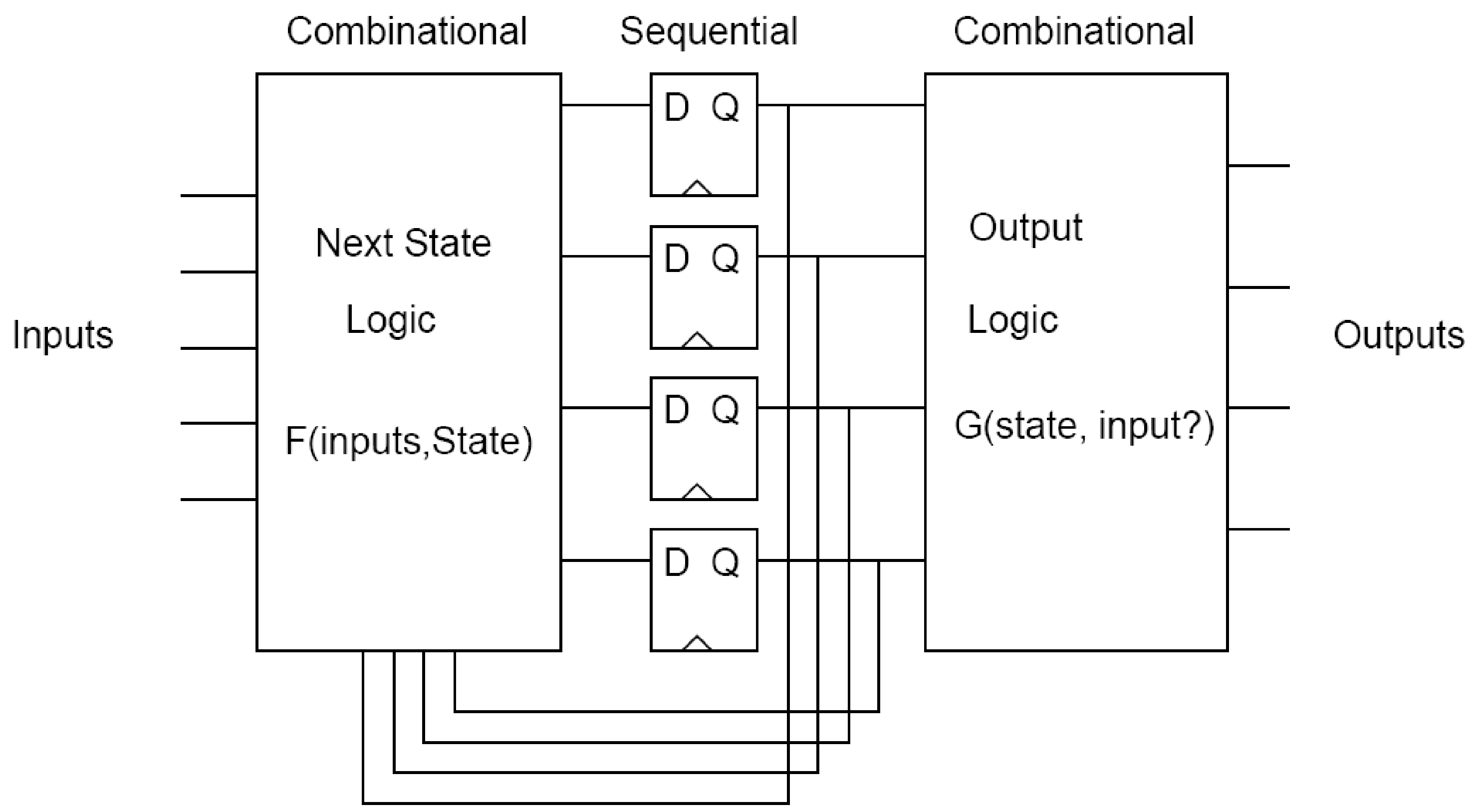
Tables (2)

- **The counter example (again):**

Transition table			State table			State/output table	
State value **S**	Input C		State name	Input C		State name	Output
	0	1		0	1		OUT
00	00	01	s0	s0	s1	s0	00
01	01	10	s1	s1	s2	s1	01
10	10	11	s2	s2	s3	s2	10
11	11	00	s3	s3	s0	s3	11
	Next state **S***			Next state			

- ***Note:*** *the notation S* indicates the value of S "at the next time step" (next clock cycle).*

Schematic representation of a FSM



Finite-state machines in Verilog

- The transition between states is a sequential operation (in the sense of a **sequential circuit**)
- In a FSM the actions that produce outputs from the inputs are always **combinational**

- From a coding perspective
- For state transitions, use a **sequential `always` block**
- For next state logic, use a **combinational `always` block**
- For output logic, use a **combinational `always` block** or **`assign` statements**

Australian National University

Two-state *Moore* machine in Verilog

```
module FSM
(input wire clk, rst, input,
output reg out);
parameter S0 = 1'b0, S1 = 1'b1;
reg state, Snext;
always @(posedge clk)
//STATE MEMORY - SEQUENTIAL
 if(rst)
  state <= S0;
 else state <= Snext;

always @(*) begin
//NEXT STATE LOGIC -COMBINAT.
 case(state)
  S0: if(input) Snext = S1;
      else Snext = S0;
  S1: if(input) Snext = S0;
      else Snext = S1;
  default: Snext = S0;
 endcase
end

always @(*) begin
//OUTPUT LOGIC -COMBINAT
 case(state)
  S0: out = f(state);
  S1: out = g(state);
  default: out = h(state);
//note: f,g,h arbitrary
 endcase
end

endmodule
```

Two-state *Mealy* machine in Verilog

```
module FSM
(input wire clk, rst, input,
output reg out);
parameter S0 = 1'b0, S1 = 1'b1;
reg state, Snext;
always @(posedge clk)
//STATE MEMORY - SEQUENTIAL
 if(rst)
  state <= S0;
 else state <= Snext;

always @(*) begin
//NEXT STATE LOGIC -COMBINAT.
 case(state)
  S0: if(input) Snext = S1;
      else Snext = S0;
  S1: if(input) Snext = S0;
      else Snext = S1;
  default: Snext = S0;
 endcase
end

always @(*) begin
//OUTPUT LOGIC -COMBINAT
 case(state)
  S0: out = f(input,state);
  S1: out = g(input,state);
  default:
      out = h(input,state);
//note: f,g,h arbitrary
 endcase
end

endmodule
```

Designing a FSM – the key steps

1. Determine the inputs / outputs. **Determine the states and give them mnemonic names**
2. Draw up a state diagram and a next state table.
3. Render the inputs and outputs in binary format, adopt an encoding for the states (you can use Verilog `parameter` to your advantage).
4. Draw a **transition table (next-state value table)**
5. Draw an **output table**
6. Use Boolean algebra (Karnaugh maps where practical) to obtain **minimal next state and output combinational logic**.
7. Use the standard Verilog style to **code and simulate your design** and check for correct operation. Revise as appropriate.
8. Check for potential practical problems (e.g. non-ideal effects).

Determining the states

- Usually a design is described in terms of:
  “I want a system that takes in such and such inputs and does something (outputs) when something else (input sequence) happens.”
  - This can be turned into a state-based description using state tables and/or state diagrams.
  - State tables are more precise but state diagrams are often more intuitive
- You should try to come up with **as few states as possible**:
  - Saving resources / easier to manage designs
  - ***Equivalent states***, i.e., states associated with the same output and the same next-state transitions can be replaced by a single state.
- Use **meaningful names**!
- Once you have picked the states the number of required flip-flops is automatically derived

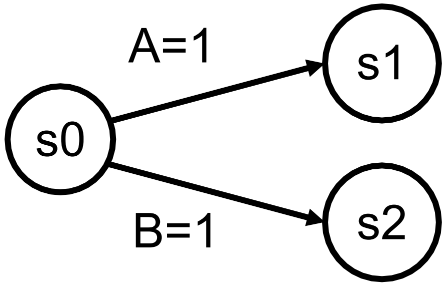
Assigning binary coding to the states

General rules to assign binary values to the states:

1. Choose an initial state which is easy to initialise (00..0)
2. Minimize the number of bit changes between adjacent states (Grey code can help)
3. Exploit symmetries: if a state or group of states are very similarly related, they can be assigned the same code with the exception of 1 bit.
4. If you have unused states, make use of the full range of binary numbers in order to achieve the above as much as possible
   - e.g., if you have 5 states (3 bits), don't stop at 101.
5. If each bit (or subgroup of bits) has a meaning, exploiting these meanings can lead to simpler next-state logic.

Designing good transition and output logic

- Karnaugh maps are a good tool when working with simple designs
  - See your textbook for an example of K-maps for more than 4 variables
- When transition/output equations are complicated you can simplify the expressions using Boolean algebra to some extent.
  - But it might be hard to come up with a minimal description
  - CAD tools (synthesisers in particular) do take some of the trouble out of this by carrying out their own complexity reduction algorithms.
- If you are working with state diagrams ensure that your diagram has no ambiguity before working on the transition logic!
  - Ambiguity can arise when different inputs can cause the system to evolve to different states.
    But what happens if both inputs are asserted at once?



What if both A and B are 1? Disambiguation required!

Summing up

- We have introduced **finite-state machine** concepts:
  - the concept of *state* and how it describes a system's dynamic behaviour
  - Mealy and Moore state machines
- We have seen how to describe state machines:
  - In terms of **state diagrams**
  - In terms of **tables (state, transition, output)**
- We have laid out some basic steps for FSM design
- All will be much clearer with real examples