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

ECE 120: Introduction to Computing

FSM Design Process

We Follow a Six-Step Process to Design an FSM

Here is a structured methodology that you can use to develop an FSM.

There are six steps:

1. develop an abstract model
2. specify I/O behavior
3. complete the specification
4. choose a state representation
5. calculate logic expressions
6. implement with flip-flops and gates

Step 1: Develop an Abstract Model

First, we translate our ideas and thoughts
- from human language
- into a model with states and desired behavior.

For now, just capture intended use (no need to be thorough nor complete).

What are the different states of the system?

How do we expect it to move amongst these states?

Step 2: Specify I/O Behavior

Start to formalize a little by specifying input and output behavior.

Input and output must consist of bits.

How many inputs are needed?

What representation is used?

And the same questions for outputs.

Sometimes, the FSM I/O must match other systems, so representations (using bits) are already defined.

Step 3: Complete the Specification

We are now ready to resolve any ambiguities

- by making design decisions
- (in other words, choosing behavior).

Implicit assumptions should also be made clear and written down.

We may choose to leave some behavior as "don't care," but such a decision should be made carefully (and checked later).

Step 4: Choose a State Representation

The FSM state representation will affect logic for both next states and outputs.

Some ways to choose

- match state to output (output patterns must be unique),
- map states to hypercube such that transitions are mostly along edges, or
- use human meaning for state bits.

The last is a good way to choose because it separates bits into meaningful groups that may not affect each other (thus simplifying logic).

Step 5: Calculate Logic Expressions

Once you have completed the specification of state IDs, next states, and outputs in bits, all that's left is to build combinational logic.

If you have a lot of variables, breaking the truth tables up may help.

State bits that have human meaning also helps to simplify here: bits may be ignored if they are not relevant.

Step 6: Implement with Flip-Flops and Gates

State bits are stored in flip-flops.*

Next-state and output logic are built in the same way that you build any other combinational logic.

There's nothing special about it.

Hook the next-state logic outputs to the D inputs of the flip-flops.

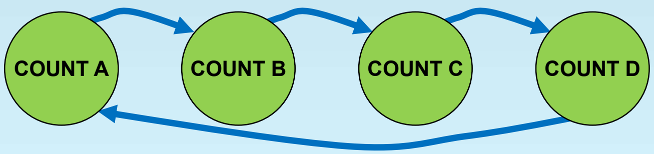
Output bits are functions of the flip-flop state.

*Registers, shift registers, and counters are fine, too. We'll use those in a week or so.

A Quick Example: A 2-Bit Gray Code Counter

Let's design a 2-bit Gray code counter using our methodology.

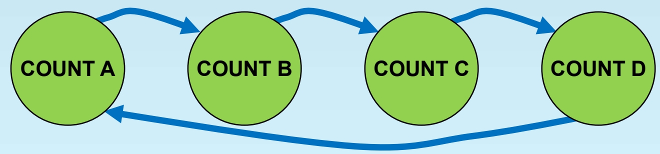
1. Our abstract model? A counter that goes through four states. Like this:



Defining I/O and Completing the Specification

2. Inputs: none (it's a counter).

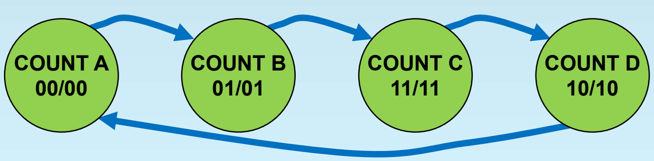
   Outputs? 00, then 01, then 11, then 10, then back to 00.
3. No inputs, so … specification is complete!



Choose Output Bits as the State IDs

4. What about the representation?

   The outputs are unique, so let's use them as state IDs as well. Then we need no output logic.



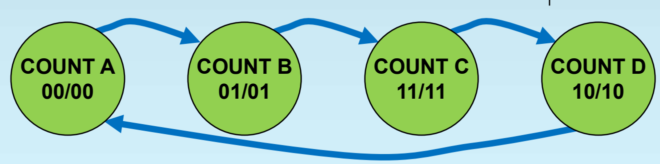
Solve the Next-State Equations

5. Now we can write equations from a truth table.

   $S_1^+ = S_0$

   $S_0^+ = S_1'$

S_1	S_0	S_1^+	S_0^+
0	0	0	1
0	1	1	1
1	0	0	0
1	1	1	0



Implement Using Two Flip-Flops

6. Finally, we can implement, as shown below.

$S_1^+ = S_0$
$S_0^+ = S_1'$