Why are We Talking About Finite State Machines

Engineers use both science and mathematics to make sure that their products will work.

The science of software is based on the finite state machine (FSM) model.

You will be learning about this model in several other courses. It plays a central role.

In this course, we will focus on using FSMs as a design tool, not for theoretical reasons.

The FSM model allows us to make sure that we have not overlooked important cases.

The FSM model enables us to connect the mathematics that we use to the real world.

There are some interesting theoretical results about the power of finite state machines; these are discussed in other courses. We will look at them as a useful tool for software designers.

What is the State of a System?

State is primitive concept. It can only be defined indirectly.

A description of the state is a description of the condition of that machine that includes all *information needed to predict the machine's future response to any external stimuli. (input)*

Two identical machines, both in a specified state will respond identically.

Certain aspects of the state (e.g. temperature, location) may be irrelevant to the behaviour that we are observing; these aspects can be ignored.

The finite state model is the way that we ignore those irrelevant aspects of the physical state.

The Distinction Between Finite and Infinite State Machines

Physical devices have an infinite number of states (ignoring quantum physics)

No upper bound to the number of states

Digital Devices can be treated as if the number of states was finite.

This is an abstraction.

We abstract from (ignore) the period of transition between stable states.

Digital machines must be built so that we can safely do that.

That's not easy. It is one of the main tasks of computer engineers. As software engineers we can usually assume that they have done their job correctly.

Example: A Chair on the Floor

A chair is a physical device with an effectively infinite number of states.

We may want to ignore, temperature, weight, small scratches, position of the centre of gravity on this earth, etc.

If we only care about its position relative to the floor (i.e. its attitude), it still has an infinite number of states, but ...

Only a finite number of those are *stable*.

Often we can ignore the transition time, the time in which the chair is moving between those stable states.

If we can ignore that transition time, and if we only care about the attitude relative to the floor, we can analyse the chair as if it were a finite state machine.

Digital Computers are Finite State Machines

All digital computers have been designed to behave as if they were finite state machines.

They are built from a finite number of components; each component has a finite number of stable states.

When reality shows through, we call a technician!

Three Important Characteristics of FSMs

(1) "Almost right" has no real meaning.

- "Almost right" assumes continuity.
- With digital machines "almost" means "not".

(2) You <u>can</u> get things exactly right.

- You cannot cut a string exactly in half.
- You can get software exactly right.

(3) Most of the mathematics that you have learned (e.g. arithmetic, calculus) must be applied with care.

- Differentiation assumes continuity
- We can only approximate continuity

<u>This makes some fundamental changes in the nature of engineering. The concept of tolerance must be refined. Differential calculus must be used with caution.</u>

Why are Modern Computers Digital?

There was once competition between digital and analogue computers.

Norbert Wiener and John von Neumann - had two conflicting visions.

Wiener's analogue computers were a *model* of the system under study.

Behaved analogously - circuits had the same differential equations as the system modelled.

Digital Computer - calculated approximate solutions to those equations.

Digital Computer - in many ways harder to use

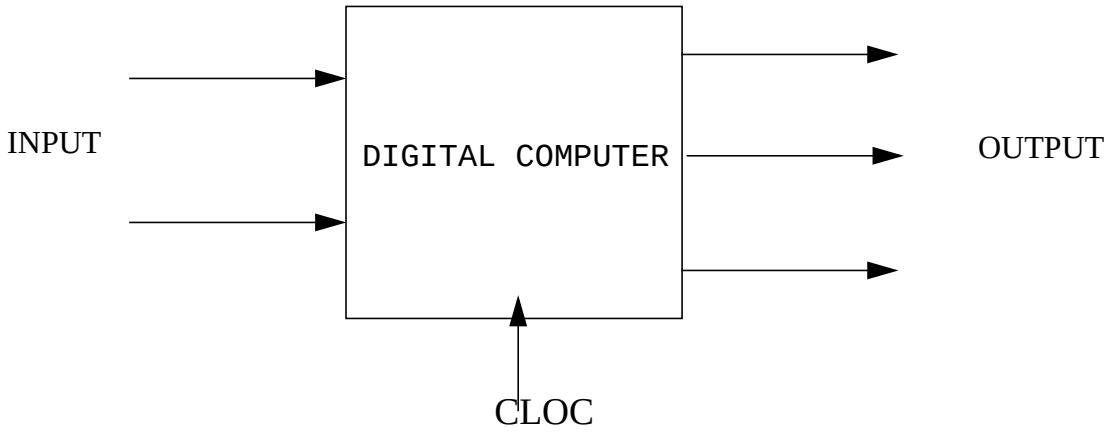
- You have to figure out how to solve the equations.
- You have to deal with a machine that is fundamentally different from the system of interest.

John von Neumann won! Why?

- greater reliability
- accuracy that is limited only by the size (number of elements)

The Finite State Machine Model

The most common view of the finite state machine i looks like a black box with wires coming in to it, called the inputs, and wires going out of it, called the output.



The machine changes its state at discrete “points” in time. The clock determines when state changes can happen.

The next state and the output are *determined* by present state input.

That is all that any digital machine does!

It is a boring, and endless, cycle.

When is a FSM non-deterministic

For non-deterministic machines, replace "*determined*" by "*constrained*".

The non deterministic model allows us to leave some things unspecified.

- because we don't know, or
- because we don't care.

Applications of the non-deterministic model:

- Designing families of programs. - Leaving things "open".
- Designing programs to be as economical as possible (by don't care conditions)
- Designing programs where we are uncertain (don't know conditions).

How Can We Describe FSMs?

If you want to describe a finite state system, here is the procedure that you should follow:

(1) Enumerate the set of states of the machine.

(2) Enumerate all of the possible input and output conditions. (input alphabet, output alphabet).

(3) Describe two functions/relations.

- The NS function/relation describes the next state for (s,i).
- The OUT function/relation describes the output for (s,i).

For small, simple machines, use a table.

"In theory" you can always use a table, but in practice we will have to use more powerful (convenient) methods.

Nonetheless, the state transition table underlies all of our methods of describing computers and programs.

Example: Programming a Chinese Abacus

First attempt: A procedure in the form of rules:

- If there are two lower beads up and you are adding two, move two beads up.
- If there are 4 lower beads up and you are adding 3, move 1 of the upper beads up and then 2 of the lower beads down.
- If both upper beads are up, move both upper beads down and add one to the column to the left.
- . . . . . .

This will take a lot of rules.

There may be ambiguities.

There may be missing cases.

We can instruct the operator by a table so that we can be sure we covered everything.

Being systematic is the key to good programming.

Assigning (arbitrary) Numbers to States.

We must make sure we get them all!

State Assignments

State	Upper	Lower	value
1	0	0	0
2	0	1	1
3	0	2	2
4	0	3	3
5	0	4	4
6	0	5	5
7	1	0	5
8	1	1	6
9	1	2	7
10	1	3	8
11	1	4	9
12	1	5	10
13	2	0	10
14	2	1	11
15	2	2	12
16	2	3	13
17	2	4	14
18	2	5	15

An Instruction Table for the Abacus

What to do with the beads for this column || What to add to left column

NS	1	2	3	4	5	6	7	8	9
1	2	3	4	5	6	8	9	10	11
2	3	4	5	6	8	9	10	11	12
3	4	5	6	8	9	10	11	12	14
4	5	6	8	9	10	11	12	14	15
5	6	8	9	10	11	12	14	15	16
6	8	9	10	11	12	14	15	16	17
7	8	9	10	11	12	14	15	16	17
8	9	10	11	12	14	15	16	17	18
9	10	11	12	14	15	16	17	18	8
10	11	12	14	15	16	17	18	8	9
11	12	14	15	16	17	18	8	9	10
12	14	15	16	17	18	8	9	10	11
13	14	15	16	17	18	8	9	10	11
14	15	16	17	18	8	9	10	11	12
15	16	17	18	8	9	10	11	12	14
16	17	18	8	9	10	11	12	14	15
17	18	8	9	10	11	12	14	15	16
18	8	9	10	11	12	14	15	16	17

OUT	1	2	3	4	5	6	7	8	9
1	0	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0	0
8	0	0	0	0	0	0	0	0	0
9	0	0	0	0	0	0	0	0	1
10	0	0	0	0	0	0	0	1	1
11	0	0	0	0	0	0	1	1	1
12	0	0	0	0	0	1	1	1	1
13	0	0	0	0	0	1	1	1	1
14	0	0	0	0	1	1	1	1	1
15	0	0	0	1	1	1	1	1	1
16	0	0	1	1	1	1	1	1	1
17	0	1	1	1	1	1	1	1	1
18	1	1	1	1	1	1	1	1	1

This “program” (or set of programs) turns the abacus into a new machine. It is a finite state machine and its behaviour is described by the above table. The old commands were in terms of moving individual beads. The new commands are addition commands.

Another Instruction Table for the Abacus

What to do with the beads for this column || What to add to left column

NS	1	2	3	4	5	6	7	8	9
1	2	3	4	5	7	8	9	10	11
2	3	4	5	7	8	9	10	11	13
3	4	5	7	8	9	10	11	13	14
4	5	7	8	9	10	11	13	14	15
5	7	8	9	10	11	13	14	15	16
6	8	9	10	11	13	14	15	16	17
7	8	9	10	11	13	14	15	16	17
8	9	10	11	13	14	15	16	17	18
9	10	11	13	14	15	16	17	18	8
10	11	13	14	15	16	17	18	8	9
11	13	14	15	16	17	18	8	9	10
12	14	15	16	17	18	8	9	10	11
13	14	15	16	17	18	8	9	10	11
14	15	16	17	18	8	9	10	11	13
15	16	17	18	8	9	10	11	13	14
16	17	18	8	9	10	11	13	14	15
17	18	8	9	10	11	13	14	15	16
18	8	9	10	11	13	14	15	16	17

OUT	1	2	3	4	5	6	7	8	9
1	0	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0	0
8	0	0	0	0	0	0	0	0	0
9	0	0	0	0	0	0	0	0	1
10	0	0	0	0	0	0	0	1	1
11	0	0	0	0	0	0	1	1	1
12	0	0	0	0	0	1	1	1	1
13	0	0	0	0	0	1	1	1	1
14	0	0	0	0	1	1	1	1	1
15	0	0	0	1	1	1	1	1	1
16	0	0	1	1	1	1	1	1	1
17	0	1	1	1	1	1	1	1	1
18	1	1	1	1	1	1	1	1	1

This is another “program” in which we move the five lower beads down and replace them with an upper bead whenever we can. It works just as well, gets the same answers.

Another Instruction Table for the Abacus

What to do with the beads for this column || What to add to left column

NS	1	2	3	4	5	6	7	8	9
1	2	3	4	5	7	8	9	10	11
2	3	4	5	7	8	9	10	11	1
3	4	5	7	8	9	10	11	1	2
4	5	7	8	9	10	11	1	2	3
5	7	8	9	10	11	1	2	3	4
6	8	9	10	11	1	2	3	4	5
7	8	9	10	11	1	2	3	4	5
8	9	10	11	1	2	3	4	5	7
9	10	11	1	2	3	4	5	7	8
10	11	1	2	3	4	5	7	8	9
11	1	2	3	4	5	7	8	9	10
12	2	3	4	5	7	8	9	10	11
13	2	3	4	5	7	8	9	10	11
14	3	4	5	7	8	9	10	11	13
15	4	5	7	8	9	10	11	13	14
16	5	7	8	9	10	11	13	14	15
17	5	8	9	10	11	13	14	15	16
18	8	9	10	11	13	14	15	16	17

OUT	1	2	3	4	5	6	7	8	9
1	0	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0	1
3	0	0	0	0	0	0	0	1	1
4	0	0	0	0	0	0	1	1	1
5	0	0	0	0	0	1	1	1	1
6	0	0	0	0	1	1	1	1	1
7	0	0	0	0	1	1	1	1	1
8	0	0	0	1	1	1	1	1	1
9	0	0	1	1	1	1	1	1	1
10	0	1	1	1	1	1	1	1	1
11	1	1	1	1	1	1	1	1	1
12	1	1	1	1	1	1	1	1	1
13	1	1	1	1	1	1	1	1	1
14	1	1	1	1	1	1	1	1	1
15	1	1	1	1	1	1	1	1	1
16	1	1	1	1	1	1	1	1	1
17	1	1	1	1	1	1	1	1	1
18	1	1	1	1	1	1	1	1	1

This version does a “carry” as soon as possible. It doesn’t really use all of its states and it has a reduced capacity. The maximum value of a digit is 9.

Note that the highest number in rows 1 - 11 is 11. The first 11 rows describe a “*terminal submachine*”. Are the other rows needed?

Why are we discussing this?

We don't really care about the abacus.

Tables like this are our way of being systematic, being sure we cover all our cases, being precise, and reducing a complex problem to a simple one.

Designing finite state machines like this is a simple form of programming.

Some of the best programs are table driven.

Even if we don't use the table directly, making tables like this is a systematic way of covering all of the cases and thinking about a problem a little bit at a time.

Of course, we will have to extend this technique to allow us to deal with more cases.

This is the "fall back" technique. We use it when nothing else works. It is the basis for all other techniques.

An example: the “ABA” acceptor

This is a description of a simple machine that will output “1” only if the most recent three inputs were “A”, “B”, “A”.

(initial state = 1)

NS	A	B	C
1	2	1	1
2	2	3	1
3	2	1	1

OUT	A	B	C
1	0	0	0
2	0	0	0
3	1	0	0

The state set is {1, 2, 3}.

The input set is {A, B, C}.

The output set is {0, 1}.

The ABA or BAB Acceptor (initial state = 1)

This is a description of a machine that will output "1" if and only if the most recent three inputs were: "A", "B", "A" and a "2" if and only if the most recent three inputs were: "B", "A","B".

NS	A	B	C
1	2	3	1
2	2	4	1
3	5	3	1
4	5	3	1
5	2	4	1

OUT	A	B	C
1	0	0	0
2	0	0	0
3	0	0	0
4	1	0	0
5	0	2	0

The state set is {1, 2, 3, 4,5}.

The input set is {A, B, C}.

The output set is {0, 1, 2}.

Another ABA Acceptor - an Example of Nondeterminism (initial state = 1)

NS	A	B	C
1	{2,4}	1	1
2	{2,4}	3	1
3	{2,4}	1	1
4	{2,4}	3	1

OUT	A	B	C
1	0	0	0
2	0	0	0
3	1	0	0
4	0	0	0

Applying Finite State Machines

In theory, any computer can be described by such a table, but theory is not our concern in this class.

We are interested in finite state machines because they allow us to analyse the behaviour of our programs completely.

In practice, the tables are much too large.

However,

- We know that computers can in principle be described completely and analysed.
- We know that there is no magic, no giant brain.
- We know that we can be systematic in our analysis.
- We know that we can be complete in our analysis.

If we can learn to describe classes of states and classes of inputs, we can describe bigger machines. Eventually tables of this sort will be the way that we write specifications for programs.

Equivalent Finite State Machines

This ABA Acceptor is equivalent to the previous one.

initial state = 1

NS	A	B	C
1	3	1	1
2	3	1	1
3	3	2	1

OUT	A	B	C
1	0	0	0
2	1	0	0
3	0	0	0

Two finite state machines are equivalent if you could not tell one from the other if the only things you can observe were the input and the output.

The number of states is not visible outside of the box.

Two equivalent machines need not have the same number of states.

Equivalent Finite State Machines

This machine too is an ABA acceptor.

initial state = l

NS	A	B	C
1	4	1	1
2	3	1	1
3	4	2	1
4	3	2	1

OUT	A	B	C
1	0	0	0
2	1	0	0
3	0	0	0
4	0	0	0

All three machines are equivalent.

Equivalent machines are indistinguishable from outside.

In this machine, states 3 and 4 are equivalent.

Equivalent states are indistinguishable from outside.

Machines where no two states are equivalent are called *minimal*.

Minimal machines are not necessarily better in any way!

Software designers often build faster programs by having extra, equivalent, states.

Designing a digital machine

The following pages illustrate how we apply the following procedure:

(1) List the input and output values.

(2) List the historical conditions that might be relevant. these your initial set of states.

(3) Form two tables with one row for each state and one column for each input value.

(4) In the first table, list the next states for each state/ input combination. Add states if needed.

(5) In the second table, list the outputs for each state/ input combination. Add states if needed.

The result may not be minimal!

There are procedures for reducing machines to minimal machines.

You will learn about these procedures in your logic design course.

Example: Designing an ABA Machine

The Input values are “A”, “B”, “C”. The output values are “0”, “1”.

For the “Historical Conditions”, Do we need the last three inputs, i.e.: AAA, AAB, AAC, ABA, ABB, ABC, ACA, ACB, ACC, BAA, ...
or only?

AA, AB, AC, BA, BB, BC, CA, CB, CC

We name the latter 1 ... 9 and complete the tables below:

Initial State = 9

NS	A	B	C
1 (AA)	1	2	3
2 (AB)	4	5	6
3 (AC)	7	8	9
4 (BA)	1	2	3
5 (BB)	4	5	6
6 (BC)	7	8	9
7 (CA)	1	2	3
8 (CB)	4	5	6
9 (CC)	7	8	9

OUT	A	B	C
1	0	0	0
2	1	0	0
3	0	0	0
4	0	0	0
5	0	0	0
6	0	0	0
7	0	0	0
8	0	0	0
9	0	0	0

Example: Designing an ABA Machine (continued)

:

Initial State = 9

NS	A	B	C
1 (AA)	1	2	3
2 (AB)	4	5	6
3 (AC)	7	8	9
4 (BA)	1	2	3
5 (BB)	4	5	6
6 (BC)	7	8	9
7 (CA)	1	2	3
8 (CB)	4	5	6
9 (CC)	7	8	9

OUT	A	B	C
1	0	0	0
2	1	0	0
3	0	0	0
4	0	0	0
5	0	0	0
6	0	0	0
7	0	0	0
8	0	0	0
9	0	0	0

states 1, 4, 7 are identical.

3, 6, and 9 are identical.

5 and 8 are identical.

Example: Designing an ABA Machine (continued)

This leads to a new reduced table:

Initial State = 3

NS	A	B	C
1	1	2	3
2	1	4	3
3	1	4	3
4	1	4	3

OUT	A	B	C
1	0	0	0
2	1	0	0
3	0	0	0
4	0	0	0

Now, states 3 and 4 are identical. The initial state can be either 3 or 4.

The final result of our "design":.

Initial State = 3

NS	A	B	C
1	1	2	3
2	1	3	3
3	1	3	3

OUT	A	B	C
1	0	0	0
2	1	0	0
3	0	0	0

Note that we came up with this design as a set of simple, systematic steps.

There are more imaginative ways to approach this problem but this one will always work.

Designing Cruise Control Software:

Inputs

1	Switch system on
2	Switch system off
3	Lock current speed
4	Suspend control
5	Resume control

States

1	System off
2	System on, no speed stored
3	System has speed stored but is not engaged
4	System has speed and is engaged

Next State

NS	1	2	3	4	5
1	2	1	1	1	1
2	2	1	3/4?	2	2
3	3	1	3/4?	3	4
4		1	4	3	4

Note: Actual speed control is a separate (simulated) analog control system.

An alternate design might do more error detection.

Networks of Finite State Machines

Nobody can design a real computer system as a single finite state machine!

The state table would be far too large.

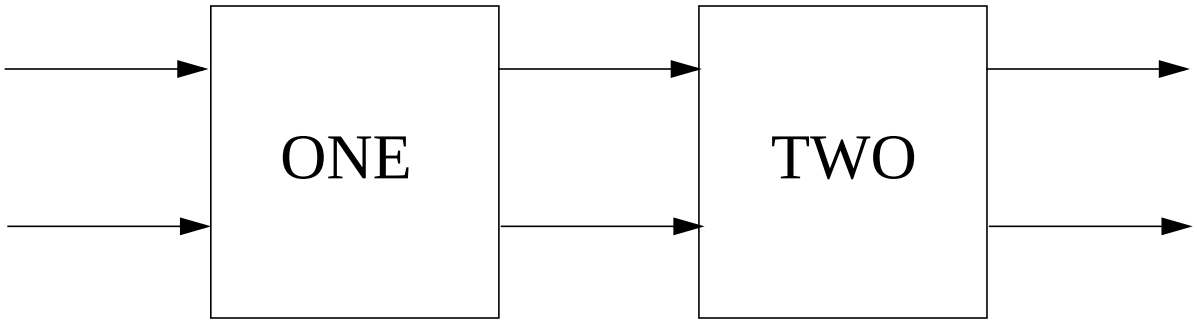
The secret to designing both hardware and software is *modularisation*.

Modularisation means designing something as a set of components, each of which can be designed without knowledge of the details of the others.

We can build networks of finite state machines in which:

- the output of one machine is the input to another,
- inputs are “split” and sent to more than one machine.
- outputs from several machines are combined to be considered as a single output.

Machines connected in “series”.



In this example, the inputs and outputs are pairs of elements.

The output alphabet of machine ONE must be the same as the input alphabet of machine TWO.

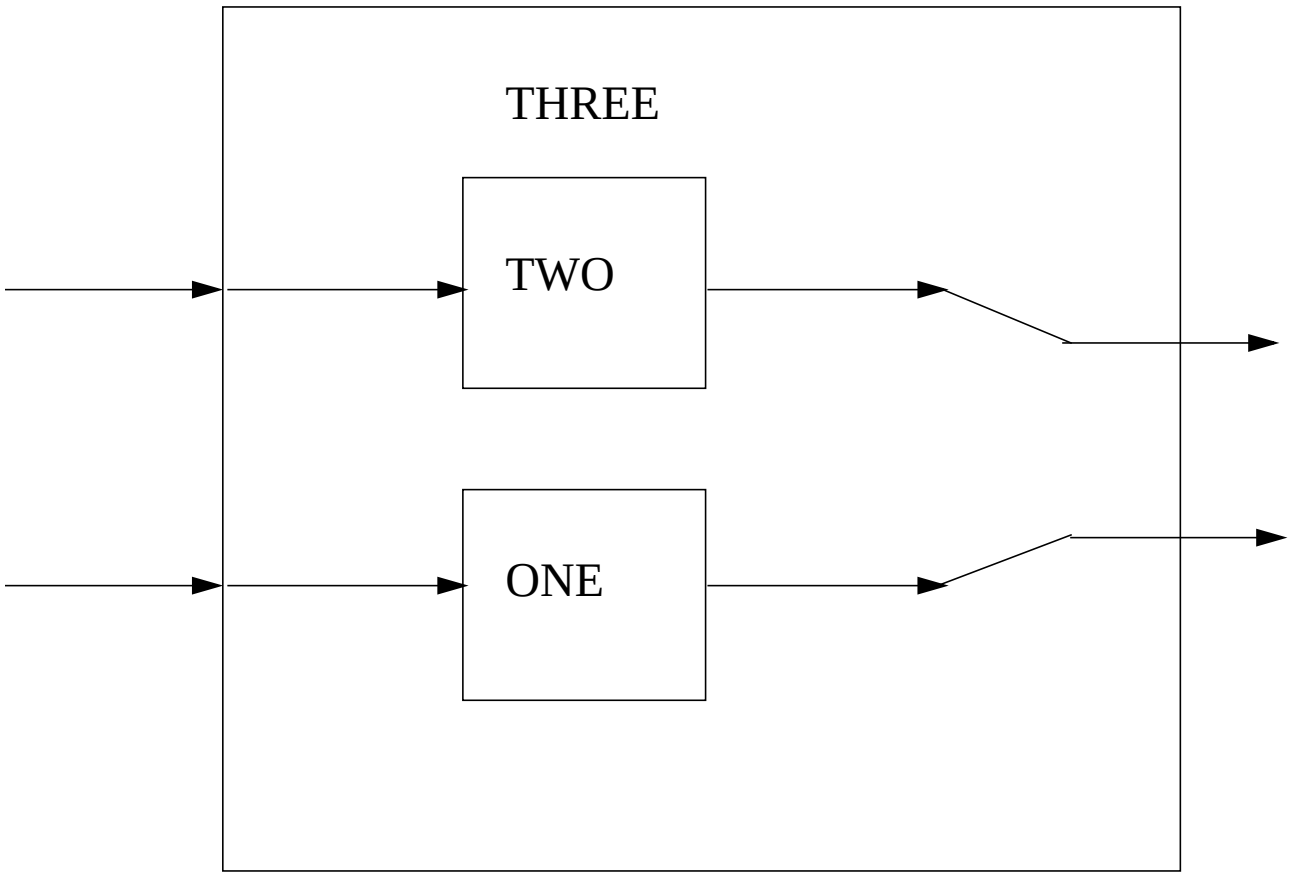
We assume that the machine clocks are synchronised. Two machines are *synchronised* if they always change state at the same time.

The maximum number of network states is the product of the number of states of the two machines. It can be less if some combinations cannot happen.

Other ways to build networks

An input to the network (or the output of one of the machines can be “split”, i.e. go to several inputs.

Several machines may be combined (by putting a box around them as in the following example



Why do we want Networks?

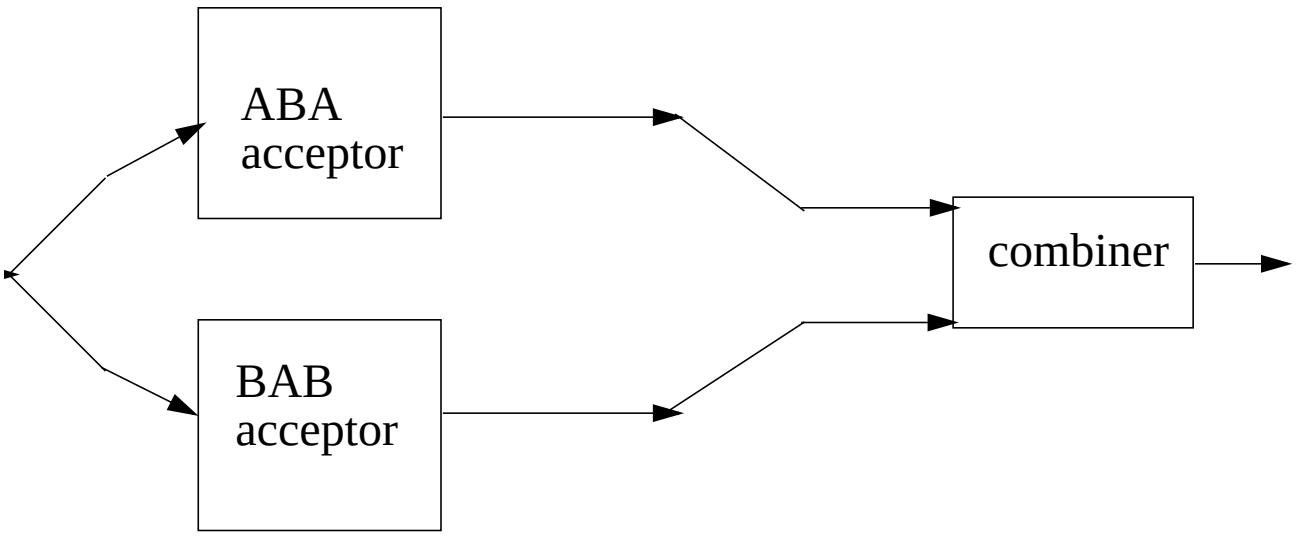
Most real finite state machines many states to enumerate.

Even with patience, the complexity would lead to errors.

We need to “divide and conquer” the complexity.

What we study now with FSMs, we will do later with programs.

Example: A “ABA or BAB” machine



The “Combiner” Machine

It is a one state machine. (combinatorial logic)

Its tables are:

NS	(0,0)	(1,0)	(0,1)	(1,1)
1	1	1	1	1

OUT	(0,0)	(1,0)	(0,1)	(1,1)
	0	1	2	*

“*” indicates “don't care”, a most important concept. We don’t care because we can never have both ABA and BAB at the same time.

We often write programs (design machines) in which certain cases do not arise.

To avoid “overspecification”, we indicate that the output in those cases can be anything that the constructor wants.

Summary

(1) We need better ways of describing large tables.

(2) Any computer can be viewed as a finite state machine - when the hardware is working properly

(3) *Theoretically*, one can design computers by state enumeration.

(4) This is a “fall back” approach.

(5) Decomposition is the key to mastering complexity.

(6) We want simple networks of simple machines.

(7) Always reduce a large programming problem to a sequence of simple ones.

(8) Picking the component machines is the critical step.

(9) Understanding the concept of state is the “science” of programming.

(10) Remember the true meaning of state: the state is all you need to know to predict the future behaviour of that machine. In a network, the states of the individual machines define classes of network states.

Dealing With Larger Machines

Make Assertions about classes or sets of states.

Example: In all states with an even number of widgets, ...

There is a well developed mathematical notation for talking about sets.

You will be learning about this in your other courses.

The next slide summarises common notation.

Notation for sets

{x,y,z} enumeration - the set containing x, y, z

| such that

{x | <condition>}The set of elements such that x satisfies the condition.

$A \subseteq B$ A is a subset of B (could be identical)

$A \subset B$ A is a subset of B and smaller than B. (A is a *proper* subset of B)

$A \cup B$ set of elements in either A or B

$A \cap B$ set of elements in both A and B

$A - B$ set of elements of A that are not in B

$-(B)$ set of elements in Universe not in B (the complement of B)

$X \in A$ X is an element of A

{} an empty set

Only combine sets from the same Universe.

Even empty sets must have an associated Universe.

Evaluating Predicate Expressions

If P and Q are predicate expressions,

(1) (a) $(\forall x_k, P)$ is ***true*** if P is true for all values of x_k in our Universe. Otherwise, it is ***false***.

(2) (b) $(\exists x_k, P)$ is ***true*** if P is true if there is a value of x_k in our Universe for which P is ***true***. Otherwise, it is ***false***.

(3) (c) $(P) \wedge (Q)$ is *true* if both P and Q are ***true***. Otherwise, it is ***false***.

(4) (d) $(P) \vee (Q)$ is ***true*** if either P or Q are ***true***. Otherwise, it is ***false***.

(5) (e) $\neg(P)$ is ***true*** if P is ***false***. Otherwise, it is ***false***.

(6) (f) $(P) \Rightarrow (Q)$ is *true* if either P is ***false*** or Q is ***true***. Otherwise, it is ***false***.

The symbols are read, “for all”, “there exists”, “and”, “or”, “not”, and “implies”.

Summary of Mathematical Terms

- A *relation* is a set of pairs (2-tuples).
- The set of values that appear as the first element of a pair is called the *domain* of that relation.
- The set of values that appear as the second element of a pair is called the relation's *range*.
- A *function* is a relation such that for any given element, x, in its domain, there is only one pair (x,y) in the function.
- If (a,b) is in the function F, "F(a)" means b, often called "*the value of F at a*"**.**
- Domains and ranges may include tuples. It may make sense to write "F((a,b))", "F((a,b,c))", and "F(F((a,b,c)))".
- Functions whose domain is smaller than the universe are called *partial functions*
- Many of the functions that arise in software development will be partial functions.
- A *predicate* is a function whose range contains no members other than ***true*** and ***false***.
- For any set, X, the *characteristic predicate* of X is a predicate whose domain is the universe from which X is drawn, and whose value, for b, is ***true*** if and only if b is a member of X.

Predicative Descriptions of Finite State Machines

Real computers have so many states that it is impractical to describe machines by enumerating the states.

Decomposing a machine into a network it is not always sufficient. Even simple, regular machines can have many states.

We have to find another way to describe the Next State and Output functions.

Because predicates can characterise sets, and hence functions, we can use predicate expressions to describe finite state machines.

These same mathematical concepts will be used to describe programs.

The ABA machine revisited.

Initial State = 1

NS	A	B	C
1	3	1	1
2	3	1	1
3	3	2	1

OUT	A	B	C
1	0	0	0
2	1	0	0
3	0	0	0

If we don't want to enumerate all the states and inputs we can describe the function by its characteristic predicate.

The following characterises the NS function:

$((NS = 1) \wedge$

$((((s = 1) \vee (s = 2)) \wedge ((i = B) \vee (i = C)))$

$\vee ((s = 3) \wedge (i = C)))) \vee$

$((NS = 2) \wedge (s = 3) \wedge (i = B)) \vee$

$((NS = 3) \wedge (i = A))$

The following characterises the OUT function:

$((OUT = 1) \wedge (s = 2) \wedge (i = A)) \vee$

$((OUT = 0) \wedge (\neg((s = 2) \wedge (i = A))))$

For this example, the table is simpler than the characteristic predicate.

Predicative Descriptions

When do they help?

- Help if and only if there is regularity that can be exploited.
- Help most when there are many states
- Can also be (helpfully) written in tabular form.

Most real programs meet these constraints.

This is why your logic course is so important.

Logic will allow you to describe complex systems precisely.

There are tools to help you check your descriptions and that will confirm that they have or do not have certain properties.

A Predicative Description of a Machine that Counts Up and Down Between 0 and 100

$((s < 100) \wedge (IN = 1) \wedge (NS = s + 1)) \vee$

$((s = 100) \wedge (IN = 1) \wedge (NS = 100)) \vee$

$((s > 0) \wedge (IN = 2) \wedge (NS = s - 1)) \vee$

$((s = 0) \wedge (IN = 2) \wedge (NS = 0))$

$((s < 100) \wedge (IN = 1) \wedge (OUT = s + 1)) \vee$

$((s = 100) \wedge (IN = 1) \wedge (OUT = 100)) \vee$

$((s > 0) \wedge (IN = 2) \wedge (OUT = s - 1)) \vee$

$((s = 0) \wedge (IN = 2) \wedge (OUT = 0))$

Descriptions of this sort are better than enumerative descriptions.

Descriptions of this sort are still hard to read and check.

Can we use tables to make things better?

Using tables to define functions.

The functions that describe software are defined "piecewise - different definitions under different conditions.

In this circumstance, one-dimensional expressions can be hard to read.

Conditional expressions with many cases become complex.

We can use a table to make things easier to read.

The columns headings are predicate expressions.

The row headings are the names of variables.

The column headings should be mutually exclusive.

The table entries are terms.

	w < 0	w = 0	w > 0
x=	x + w + q	x + 2 – q	x – w
y=	y + 2	x + y	x + y + 2
z=	z – w	z	z + w

Example of a Table Defining a Relation

$(\exists i, ((1 \leq i \leq n) \wedge (`A[i] = `x))$	$\neg(\exists i, ((1 \leq i \leq n) \wedge (`A[i] = `x))$

j' |
present'=

`A[j] = `x	true
true	**false**

A is a real Array

x is a real variable

i is an integer variable

j is an integer variable

present, a boolean variable, can have values "**true**", and "**false**".

We will discuss these tables more carefully later.

Tabular Description of the Counter

Normal Function Table Describing NS(s,IN)

NS =

	IN = 1	IN = 2
		H_1
s= 0	s + 1	0
0 < s < 100	s + 1	s - 1
s = 100	100	s - 1
H_2		**G**

Normal Function Table Describing OUT(s,IN)

OUT=

	IN = 1	IN = 2
		H_1
s= 0	s + 1	0
0 < s < 100	s + 1	s - 1
s = 100	100	s - 1
H_2		**G**

Note how the use of predicates and tabular notation helps in getting descriptions that are both precise and easy to read.

Themes in these lecture:

1. Finite State Machine Model is a good design tool.
2. Using the FSM model, we can be disciplined and check for complete coverage.
3. Logic and Tables makes it practical to do this.

Engineers work from specifications and produce well- documented products.

This should be no-less-true for software.

But, it is not “the usual way”.

Computers are fundamentally simple.

We have to work to keep them that way.

Divide work into small “modules”.

Encapsulate modules by specifications.

Keep the specification simpler than the program.