12. Finite-State Machines

12.1 Introduction

This chapter introduces finite-state machines, a primitive, but useful computational model for both hardware and certain types of software. We also discuss regular expressions, the correspondence between non-deterministic and deterministic machines, and more on grammars. Finally, we describe typical hardware components that are essentially physical realizations of finite-state machines.

Finite-state machines provide a simple computational model with many applications. Recall the definition of a Turing machine: a finite-state controller with a movable read/write head on an unbounded storage tape. If we restrict the head to move in only one direction, we have the general case of a finite-state machine. The sequence of symbols being read can be thought to constitute the input, while the sequence of symbols being written could be thought to constitute the output. We can also derive output by looking at the internal state of the controller after the input has been read.

Finite-state machines, also called *finite-state automata* (singular: *automaton*) or just finite *automata* are much more restrictive in their capabilities than Turing machines. For example, we can show that it is not possible for a finite-state machine to determine whether the input consists of a prime number of symbols. Much simpler languages, such as the sequences of well-balanced parenthesis strings, also cannot be recognized by finite-state machines. Still there are the following applications:

- Simple forms of pattern matching (precisely the patterns definable by "regular expressions", as we shall see).

- Models for sequential logic circuits, of the kind on which every present-day computer and many device controllers is based.

- An intimate relationship with directed graphs having arcs labeled with symbols from the input alphabet.

Even though each of these models can be depicted in a different setting, they have a common mathematical basis. The following diagram shows the context of finite-state machines among other models we have studied or will study.

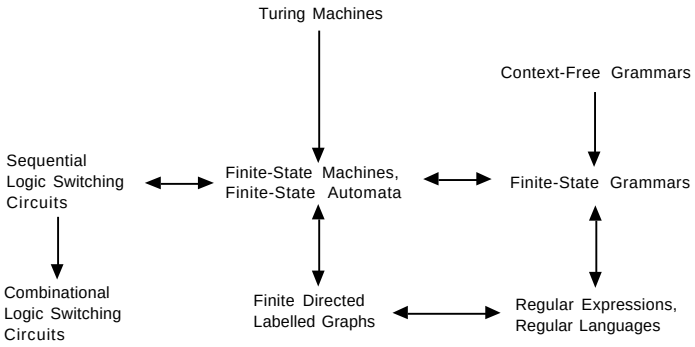


Figure 189: The interrelationship of various models with respect to computational or representational power. The arrows move in the direction of restricting power. The bi-directional arrows show equivalences.

Finite-State Machines as Restricted Turing Machines

One way to view the finite-state machine model as a more restrictive Turing machine is to separate the input and output halves of the tapes, as shown below. However, mathematically we don't need to rely on the tape metaphor; just viewing the input and output as sequences of events occurring in time would be adequate.

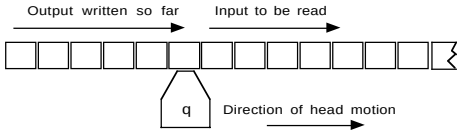


Figure 190: Finite-state machine as a one-way moving Turing machine

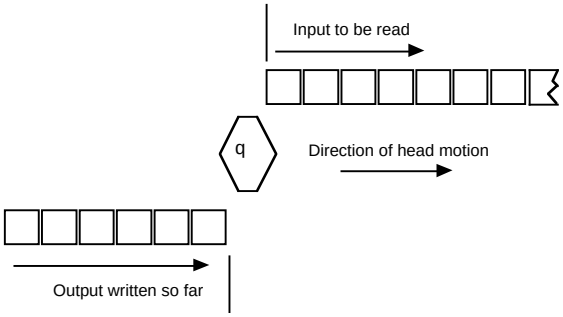


Figure 191: Finite-state machine as viewed with separate input and output

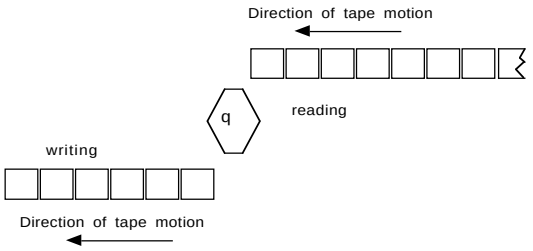


Figure 192: Finite-state machine viewed as a stationary-head, moving-tape, device

Since the motion of the head over the tape is strictly one-way, we can abstract away the idea of a tape and just refer to the input *sequence* read and the *output* sequence produced, as suggested in the next diagram. A machine of this form is called a ***transducer***, since it maps input sequences to output sequences. The term *Mealy machine*, after George H. Mealy (1965), is also often used for transducer.

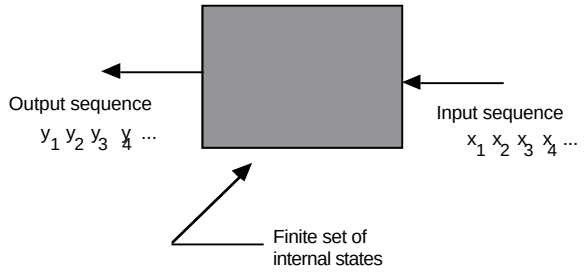


Figure 193: A transducer finite-state machine viewed as a tapeless "black box" processing an input sequence to produce an output sequence

On the other hand, occasionally we are not interested in the sequence of outputs produced, but just an output associated with the current state of the machine. This simpler model is called a *classifier*, or *Moore machine*, after E.F. Moore (1965).

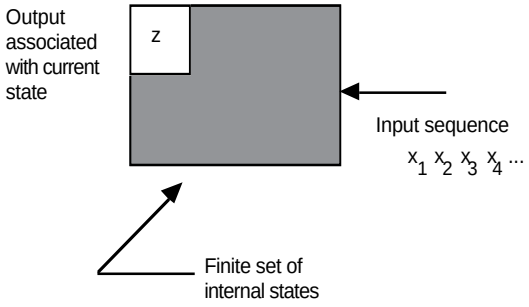


Figure 194: Classifier finite-state machine.
Output is a function of current state, rather than being a sequence

Modeling the Behavior of Finite-State Machines

Concentrating initially on transducers, there are several different notations we can use to capture the behavior of finite-state machines:

- As a functional program mapping one list into another.
- As a restricted imperative program, reading input a single character at a time and producing output a single character at a time.
- As a feedback system.
    - Representation of functions as a table
    - Representation of functions by a directed labeled graph

For concreteness, we shall use the sequence-to-sequence model of the machine, although the other models can be represented similarly. Let us give an example that we can use to show the different notations:

Example: An Edge-Detector

The function of an edge detector is to detect transitions between two symbols in the input sequence, say 0 and 1. It does this by outputting 0 as long as the most recent input symbol is the same as the previous one. However, when the most recent one differs from the previous one, it outputs a 1. By convention, the edge detector always outputs 0 after reading the very first symbol. Thus we have the following input output sequence pairs for the edge-detector, among an infinite number of possible pairs:

input	output
0	0
00	00
01	01
011	010
0111	0100
01110	01001
1	0
10	01
101	011
1010	0111
10100	01110
etc.	

Functional Program View of Finite-State Machines

In this view, the behavior of a machine is as a function from lists to lists.

Each state of the machine is identified with such a function.

The initial state is identified with the overall function of the machine.

The functions are interrelated by mutual recursion: when a function processes an input symbol, it calls another function to process the remaining input.

Each function:

looks at its input by one application of *first*,

produces an output by one application of *cons*, the first argument of which is determined purely by the input obtained from first, and

calls another function (or itself) on rest of the *input*.

We make the assumptions that:

The result of cons, in particular the first argument, becomes partially available even before its second argument is computed.

Each function will return NIL if the input list is NIL, and we do not show this explicitly.

Functional code example for the edge-detector:

We will use three functions, `f`, `g`, and `h`. The function `f` is the overall representation of the edge detector.

```
f([0 | Rest]) => [0 | g(Rest)];
f([1 | Rest]) => [0 | h(Rest)];
f([]) => [];

g([0 | Rest]) => [0 | g(Rest)];
g([1 | Rest]) => [1 | h(Rest)];
g([]) => [];

h([0 | Rest]) => [1 | g(Rest)];
h([1 | Rest]) => [0 | h(Rest)];
h([]) => [];
```

Notice that `f` is never called after its initial use. Its only purpose is to provide the proper output (namely `0`) for the first symbol in the input.

Example of `f` applied to a specific input:

f([0, 1, 1, 1, 0]) ==> [0, 1, 0, 0, 1]

An alternative representation is to use a single function, say `k`, with an extra argument, treated as just a symbol. This argument represents the *name* of the function that would have been called in the original representation. The top-level call to `k` will give the initial state as this argument:

```
k("f", [0 | Rest]) => [0 | k("g", Rest)];
k("f", [1 | Rest]) => [0 | k("h", Rest)];
k("f", []) => [];

k("g", [0 | Rest]) => [0 | k("g", Rest)];
k("g", [1 | Rest]) => [1 | k("h", Rest)];
k("g", []) => [];

k("h", [0 | Rest]) => [1 | k("g", Rest)];
k("h", [1 | Rest]) => [0 | k("h", Rest)];
k("h", []) => [];
```

The top level call with input sequence `x` is `k("f", x)` since `"f"` is the initial state.

Imperative Program View of Finite-State Machines

In this view, the input and output are viewed as streams of characters. The program repeats the processing cycle:

read character,
select next state,
write character,
go to next state

ad infinitum. The states can be represented as separate "functions", as in the functional view, or just as the value of one or more variables. However the allowable values must be restricted to a finite set. No stacks or other extendible structures can be used, and any arithmetic must be restricted to a finite range.

The following is a transliteration of the previous program to this view. The program is started by calling `f()`. Here we assume that `read()` is a method that returns the next character in the input stream and `write(c)` writes character `c` to the output stream.

```
void f()          // initial function
{
switch( read() )
  {
  case '0': write('0'); g(); break;
  case '1': write('0'); h(); break;
  }
```

```
}

void g()    // previous input was 0
{
switch( read() )
  {
  case '0': write('0'); g(); break;
  case '1': write('1'); h(); break; // 0 -> 1 transition
  }
}

void h()    // previous input was 1
{
switch( read() )
  {
  case '0': write('1'); g(); break; // 1 -> 0 transition
  case '1': write('0'); h(); break;
  }
}
```

[Note that this is a case where all calls can be "tail recursive", i.e. could be implemented as *goto*s by a smart compiler.]

The same task could be accomplished by eliminating the functions and using a single variable to record the current state, as shown in the following program. As before, we assume `read()` returns the next character in the input stream and `write(c)` writes character c to the output stream.

```
static final char f = 'f';                      // set of states
static final char g = 'g';
static final char h = 'h';

static final char initial_state = f;

main()
{
char current_state, next_state;
char c;

current_state = initial_state;
```

```
while( (c = read()) != EOF )
  {
  switch( current_state )
    {
    case f:                    // initial state
      switch( c )
      {
      case '0': write('0'); next_state = g; break;
      case '1': write('0'); next_state = h; break;
      }
      break;

    case g:                    // last input was 0
      switch( c )
      {
      case '0': write('0'); next_state = g; break;
      case '1': write('1'); next_state = h; break; // 0 -> 1
      }
      break;

    case h:                    // last input was 1
      switch( c )
      {
      case '0': write('1'); next_state = g; break; // 1 -> 0
      case '1': write('0'); next_state = h; break;
      }
      break;
    }
  current_state = next_state;
  }
}
```

Feedback System View of Finite-State Machines

The feedback system view abstracts the functionality of a machine into two functions, the next-state or state-transition function F, and the output function G.

F: States X Symbols → States state-transition function

G: States X Symbols → Symbols output function

The meaning of these functions is as follows:

F(q, σ) is the state to which the machine goes when currently in state q and σ is read

G(q, σ) is the output produced when the machine is currently in state q and σ is read

The relationship of these functions is expressible by the following diagram.

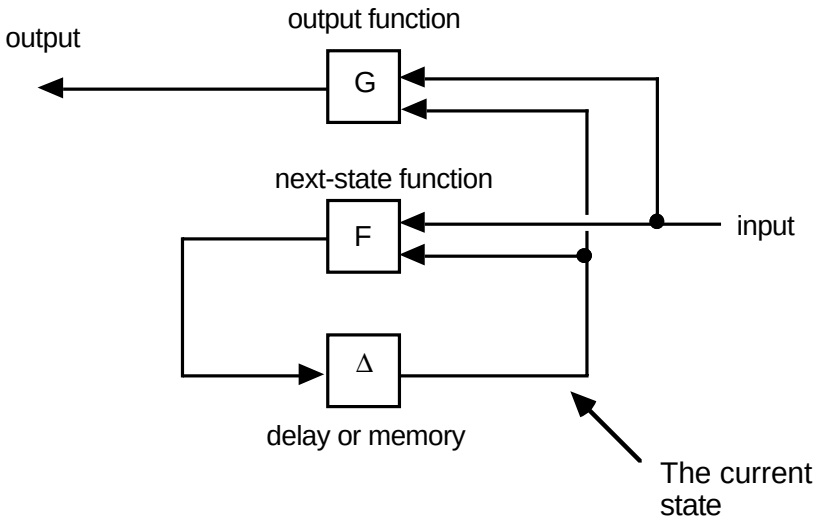


Figure 195: Feedback diagram of finite-state machine structure

From F and G, we can form two useful functions

F*: States X Symbols* → States extended state-transition function

G*: States X Symbols* → Symbols extended output function

where Symbols* denotes the set of all *sequences* of symbols. This is done by induction:

$F^*(q, \lambda) = q$

$F^*(q, x\sigma) = F(F^*(q, x), \sigma)$

$G^*(q, \lambda) = \lambda$

$G^*(q, x\sigma) = G^*(q, x)\, G(F^*(q, x), \sigma)$

In the last equation, juxtaposition is like cons'ing on the right. In other words, F*(q, x) is the state of the machine after all symbols in the sequence x have been processed, whereas G*(q, x) is the sequence of outputs produced along the way. In essence, G* can be regarded as the *function computed by* a transducer. These definitions could be transcribed into rex rules by representing the sequence xσ as a list [σ | x] with λ corresponding to [ ].

Tabular Description of Finite-State Machines

This description is similar to the one used for Turing machines, except that the motion is left unspecified, since it is implicitly one direction. In lieu of the two functions F and G, a

finite-state machine could be specified by a single function combining F and G of the form:

States X Symbols → States X Symbols

analogous to the case of a Turing machine, where we included the motion:

States X Symbols → Symbols X Motions X States

These functions can also be represented succinctly by a table of 4-tuples, similar to what we used for a Turing machine, and again called a *state transition table*:

$State_1, Symbol_1, State_2, Symbol_2$

Such a 4-tuple means the following:

> If the machine's control is in $State_1$ and reads $Symbol_1$, then machine will write $Symbol_2$ and the next state of the controller will be $State_2$.

The state-transition table for the edge-detector machine is:

	current state	**input symbol**	**next state**	**output symbol**
start state	f	0	g	0
	f	1	h	0
	g	0	g	0
	g	1	h	1
	h	0	g	1
	h	1	h	0

Unlike the case of Turing machines, there is **no particular halting convention**. Instead, the machine is always read to proceed from whatever current state it is in. This does not stop us from assigning our own particular meaning of a symbol to designate, for example, end-of-input.

Classifiers, Acceptors, Transducers, and Sequencers

In some problems we don't care about generating an entire sequence of output symbols as do the **transducers** discussed previously. Instead, we are only interested in categorizing each input sequence into one of a finite set of possibilities. Often these possibilities can be made to derive from the current state. So we attach the result of the computation to the state, rather than generate a sequence. In this model, we have an output function

c: Q → C

which gives a category or class for each state. We call this type of machine a **classifier** or **controller**. We will study the conrtoller aspect further in the next chapter. For now, we focus on the classification aspect. In the simplest non-trivial case of classifier, there are two categories. The states are divided up into the "accepting" states (class 1, say) and the "rejecting" states (class 0). The machine in this case is called an **acceptor** or **recognizer**. The sequences it accepts are those given by

$$c(F^*(q_0, x)) = 1$$

that is, the sequences *x* such that, when started in state q_0, after reading x, the machine is in a state q such that c(q) = 1. The set of all such x, since it is a set of strings, is a **language** in the sense already discussed. If A designates a finite-state acceptor, then

$$L(A) = \{ x \text{ in } \Sigma^* \mid c(F^*(q_0, x)) = 1\}$$

is the **language accepted by** A.

The structure of a classifier is simpler than that of a transducer, since the output is only a function of the state and not of both the state and input. The structure is shown as follows:

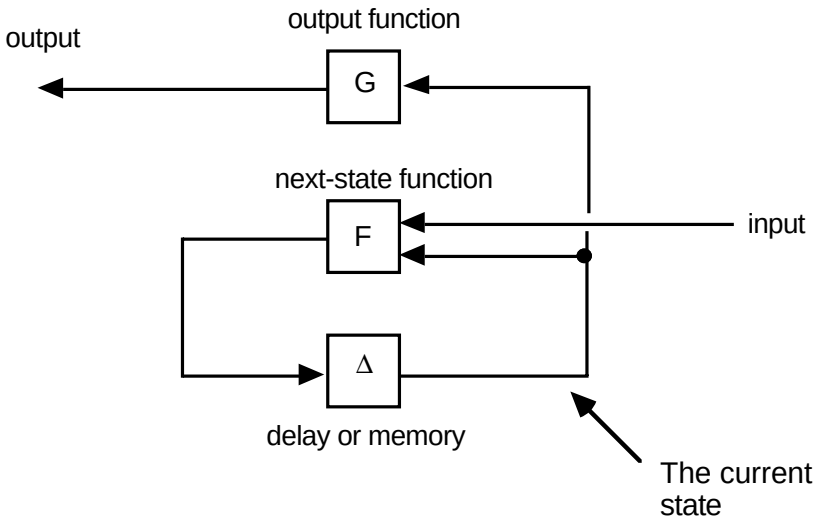


Figure 196: Feedback diagram of classifier finite-state machine structure

A final class of machine, called a **sequencer** or **generator**, is a special case of a transducer or classifier that has a single-letter input alphabet. Since the input symbols are unchanging, this machine generates a fixed sequence, interpreted as either the output sequence of a transducer or the sequence of classifier outputs. An example of a sequencer is a MIDI (Musical Instrument Digital Interface) sequencer, used to drive electronic musical instruments. The output alphabet of a MIDI sequencer is a set of 16-bit words, each having a special interpretation as pitch, note start and stop, amplitude, etc. Although

most MIDI sequencers are programmable, the program typically is of the nature of an initial setup rather than a sequential input.

Description of Finite-State Machines using Graphs

Any finite-state machine can be shown as a graph with a finite set of nodes. The nodes correspond to the states. There is no other memory implied other than the state shown. The start state is designated with an arrow directed into the corresponding node, but otherwise unconnected.

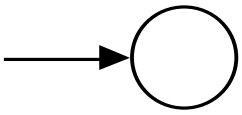


Figure 197: An unconnected in-going arc indicates that the node is the start state

The arcs and nodes are labeled differently, depending on whether we are representing a transducer, a classifier, or an acceptor. In the case of a **transducer**, the arcs are labeled σ/δ as shown below, where σ is the input symbol and δ is the output symbol. The state-transition is designated by virtue of the arrow going from one node to another.

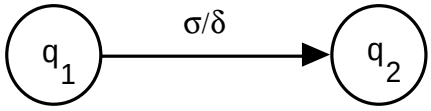


Figure 198: Transducer transition from q_1 to q_2, based on input σ, giving output δ

In the case of a **classifier**, the arrow is labeled only with the input symbol. The categories are attached to the names of the states after /.

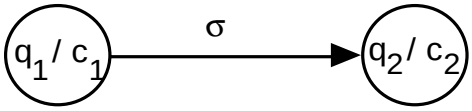


Figure 199: Classifier transition from q_1 to q_2, based on input σ

In the case of a **acceptor**, instead of labeling the states with categories 0 and 1, we sometimes use a double-lined node for an accepting state and a single-lined node for a rejecting state.

Figure 200: Acceptor, an accepting state

Transducer Example

The edge detector is an example of a transducer. Here is its graph:

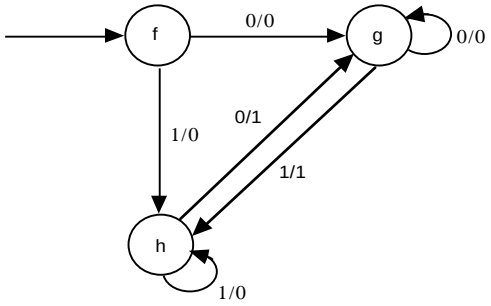


Figure 201: Directed labeled graph for the edge detector

Let us also give examples of classifiers and acceptors, building on this example.

Classifier Example

Suppose we wish to categorize the input as to whether the input so far contains 0, 1, or more than 1 "edges" (transitions from 0 to 1, or 1 to 0). The appropriate machine type is classifier, with the outputs being in the set {0, 1, more}. The name "more" is chosen arbitrarily. We can sketch how this classifier works with the aid of a graph.

The construction begins with the start state. We don't know how many states there will be initially. Let us use a, b, c, ... as the names of the states, with *a* as the start state. Each state is labeled with the corresponding class as we go. The idea is to achieve a finite closure after some number of states have been added. The result is shown below:

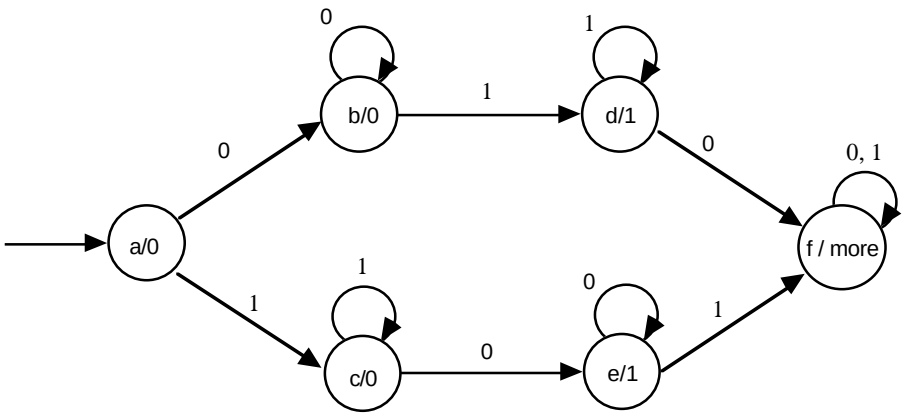


Figure 202: Classifier for counting 0, 1, or more than 1 edges

Acceptor Example

Let us give an acceptor that accepts those strings with exactly one edge. We can use the state transitions from the previous classifier. We need only designate those states that categorize there being one edge as accepting states and the others as rejecting states.

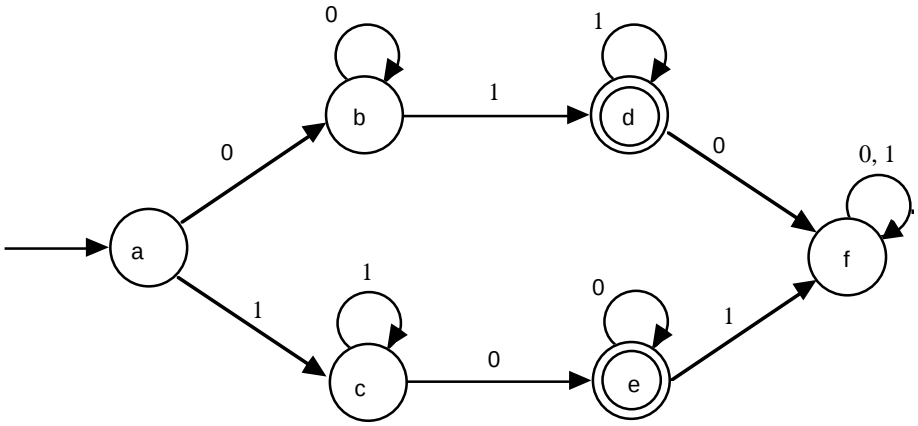


Figure 203: Acceptor for strings with exactly one edge. Accepting states are d and e.

Sequencer Example

The following sequencer, where the sequence is that of the outputs associated with each state, is that for a naive traffic light:

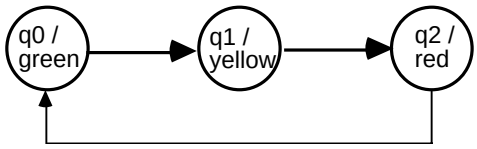


Figure 204: A traffic light sequencer

Exercises

1 •• Consider a program that scans an input sequence of characters to determine whether the sequence as scanned so far represents either an integer numeral, a floating-point numeral, unknown, or neither. As it scans each character, it outputs the corresponding assessment of the input. For example,

Input Scanned	Assessment
1	integer
+	unknown
+1	integer
+1.	floating-point
1.5	floating-point
1e	unknown
1e-1	floating-point
1e.	neither

Describe the scanner as a finite-state transducer using the various methods presented in the text.

2 •• Some organizations have automated their telephone system so that messages can be sent via pushing the buttons on the telephone. The buttons are labeled with both numerals and letters as shown:

1	**2** **a b c**	**3** **d e f**
4 **g h i**	**5** **j k l**	**6** **m n o**
7 **p r s**	**8** **t u v**	**9** **w x y**
*****	**0**	**#**

Notice that certain letters are omitted, presumably for historical reasons. However, it is common to use * to represent letter q and # to represent letter z. Common schemes do not use a one-to-one encoding of each letter. However, if we wanted such an encoding, one method would be to use two digits for each letter:

> The first digit is the key containing the letter.
>
> The second digit is the index, 1, 2, or 3, of the letter on the key. For example, to send the word "cat", we'd punch:
>
> ```
> 2 3   2 1   8 1
>  c     a     t
> ```
>
> An exception would be made for 'q' and 'z', as the only letters on the keys '*' and '#' respectively.

Give the state-transition table for communicating a series of any of the twenty-six letters, where the input alphabet is the set of digits {1, ..., 9, *, #} and the output alphabet is the set of available letters. Note that outputs occur only every other input. So we need a convention for what output will be shown in the transducer in case there is no output. Use λ for this output.

3 •• The device sketched below is capable of partially sensing the direction (clockwise, counterclockwise, or stopped) of a rotating disk, having sectors painted alternating gray and white. The two sensors, which are spaced so as to fit well within a single sector, regularly transmit one of four input possibilities: wg (white-gray), ww (white-white), gw (gray-white), and gg (gray-gray). The sampling rate must be fast enough, compared with the speed of the disk, that artifact readings do not take place. In other words, there must be at least one sample taken every time the disk moves from one of the four input combinations to another. From the transmitted input values, the device produces the directional

information. For example, if the sensors received wg (as shown), then ww for awhile, then gw, it would be inferred that the disk is rotating clockwise. On the other hand, if it sensed gw more than once in a row, it would conclude that the disk has stopped. If it can make no other definitive inference, the device will indicate its previous assessment. Describe the device as a finite-state transducer, using the various methods presented in the text.

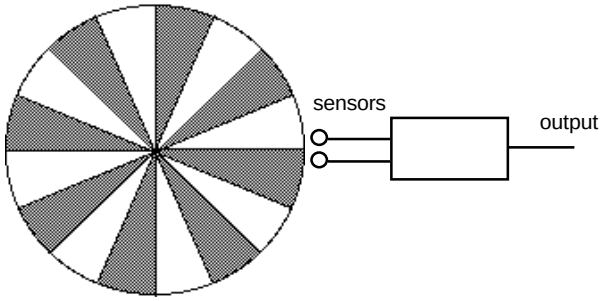


Figure 205: A rotational direction detector

4 •• Decide whether the wrist-watch described below is best represented as a classifier or a transducer, then present a state-transition diagram for it. The watch has a chronograph feature and is controlled by three buttons, *A*, *B*, and *C*. It has three display modes: the time of day, the chronograph time, and "split" time, a saved version of the chronograph time. Assume that in the initial state, the watch displays time of day. If button *C* is pressed, it displays chronograph time. If *C* is pressed again, it returns to displaying time of day. When the watch is displaying chronograph time or split time, pressing *A* starts or stops the chronograph. Pressing B when the chronograph is running causes the chronograph time to be recorded as the split time and displayed. Pressing B again switches to displaying the chronograph. Pressing B when the chronograph is stopped resets the chronograph time to 0.

5 ••• A certain vending machine vends soft drinks that cost $0.40. The machine accepts coins in denominations of $0.05, $0.10, and $0.25. When sufficient coins have been deposited, the machine enables a drink to be selected and returns the appropriate change. Considering each coin deposit and the depression of the drink button to be inputs, construct a state-transition diagram for the machine. The outputs will be signals to vend a drink and return coins in selected denominations. Assume that once the machine has received enough coins to vend a drink, but the vend button has still not been depressed, that any additional coins will just be returned in kind. How will your machine handle cases such as the sequence of coins 10, 10, 10, 5, 25?

6 ••• Consider the problem of controlling traffic at an intersection such as shown below.

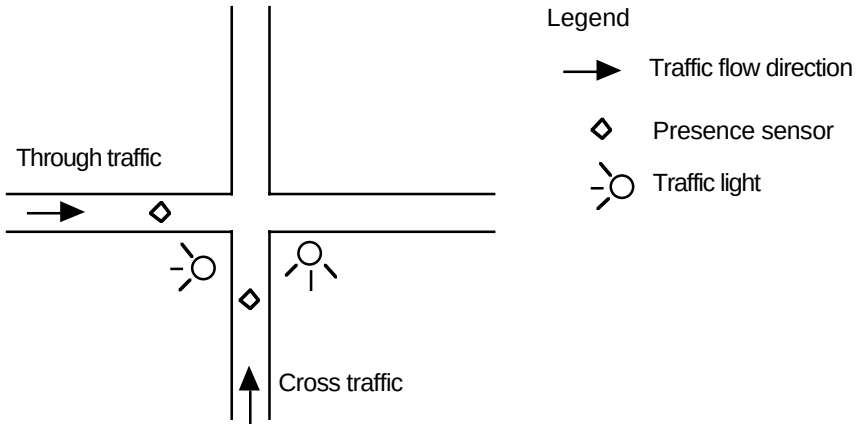


Figure 206: A traffic intersection

Time is divided into equal-length intervals, with sensors sampling the presence of traffic just before the end of an interval. The following priority rules apply:

1. If no traffic is present, through-traffic has the right-of-way.

2. If through-traffic is still present at the end of the first interval during which through-traffic has had the right-of-way, through-traffic is given the right-of-way one additional interval.

3. If cross-traffic is present at the end of the second consecutive interval during which through-traffic has had the right-of-way, then cross-traffic is given the right-of-way for one interval.

4. If cross-traffic is present but through-traffic is absent, cross-traffic maintains the right-of-way until an interval in which through-traffic appears, then through-traffic is given the right-of-way.

Describe the traffic controller as a classifier that indicates which traffic has the right-of-way.

7 ••• A bar code represents bits as alternating light and dark bands. The light bands are of uniform width, while the dark bands have width either equal to, or double, the width of the light bands. Below is an example of a code-word using the bar code. The tick marks on top show the single widths.

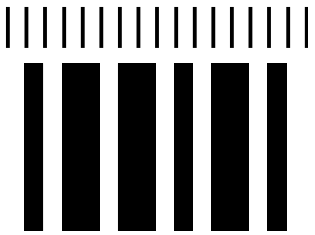


Figure 207: A bar code scheme

Assume that a bar-code reader translates the bands into symbols, L for light, D for dark, one symbol per single width. Thus the symbol sequence for the code-word above would be

```
L D L D D L D D L D L D D L D L
```

A bar pattern represents a binary sequence as follows: a 0 is encoded as LD, while a 1 is encoded as LDD. A finite-state transducer M can translate such a code into binary. The output alphabet for the transducer is {0, 1, _, end}. When started in its initial state, the transducer will "idle" as long as it receives only L's. When it receives its first D, it knows that the code has started. The transducer will give output 0 or 1 as soon it has determined the next bit from the bar pattern. If the bit is not known yet, it will give output _. Thus for the input sequence above, M will produce

```
_ _ 0 _ 1 _ _ 1 _ _ 0 _ 1 _ _ 0
L D L D D L D D L D L D D L D L
```

where we have repeated the input below the output for convenience. The transducer will output the symbol *end* when it subsequently encounters two L's in a row, at which point it will return to its initial state.

a. Give the state diagram for transducer M, assuming that only sequences of the indicated form can occur as input.

b. Certain input sequences should not occur, e.g. L D D D. Give a state-transition diagram for an acceptor A that accepts only the sequences corresponding to a valid bar code.

8 •• A gasoline pump dispenses gas based on credit card and other input from the customer. The general sequence of events, for a single customer is:

Customer swipes credit card through the slot.

Customer enters PIN (personal identification number) on keypad, with appropriate provisions for canceling if an error is made.

Customer selects grade of gasoline.

Customer removes nozzle.

Customer lifts pump lever.

Customer squeezes or releases lever on nozzle any number of times.

Customer depresses pump lever and replaces nozzle.

Customer indicates whether or not a receipt is wanted.

Sketch a state diagram for modeling such as system as a finite-state machine.

Inter-convertibility of Transducers and Classifiers (Advanced)

We can describe a mathematical relationship between classifiers and transducers, so that most of the theory developed for one will be applicable to the other. One possible connection is, given an input sequence x, record the outputs corresponding to the states through which a classifier goes in processing x. Those outputs could be the outputs of an appropriately-defined transducer. However, classifiers are a little more general in this sense, since they give output even for the empty sequence λ, whereas the output for a transducer with input λ is always just λ. Let us work in terms of the following equivalence:

> A transducer T started in state q_0 is equivalent to a classifier C started in state q_0 if, for any non-empty sequence x, the sequence of outputs emitted by T is the same as the sequence of outputs of the states through which C passes.

With this definition in mind, the following would be a classifier equivalent to the edge-detector transducer presented earlier.

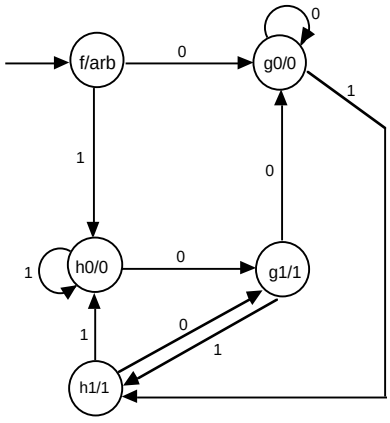


Figure 208: A classifier formally equivalent to the edge-detector transducer

To see how we constructed this classifier, observe that the output emitted by a transducer in going from a state q to a state q', given an input symbol σ, should be the same as the output attached to state q' in the classifier. However, we can't be sure that all transitions into a state q' of a transducer produce the same output. For example, there are two transitions to state *g* in the edge-detector that produce 0 and one that produces 1, and similarly for state h. This makes it impossible to attach a fixed input to either g or h. Therefore we need to "split" the states g and h into two, a version with 0 output and a version with 1 output. Call these resulting states g0, g1, h0, h1. Now we can construct an output-consistent classifier from the transducer. We don't need to split f, since it has a very transient character. Its output can be assigned arbitrarily without spoiling the equivalence of the two machines.

The procedure for converting a classifier to a transducer is simpler. When the classifier goes from state q to q', we assign to the output transition the state output value c(q'). The following diagram shows a transducer equivalent to the classifier that reports 0, 1, or more edges.

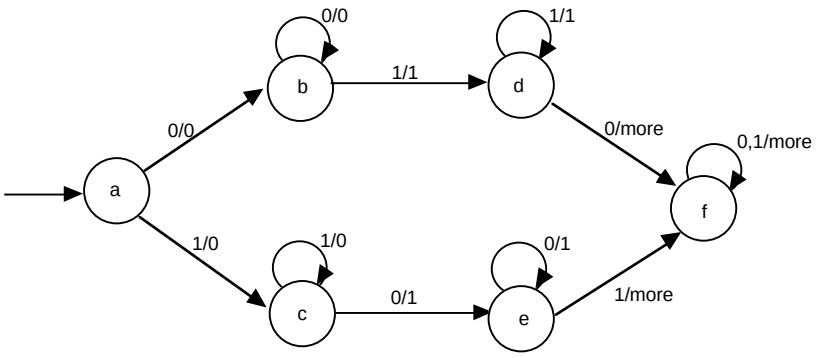


Figure 209: A transducer formally equivalent to the edge-counting classifier

Exercises

1 •• Whichever model, transducer or classifier, you chose for the wrist-watch problem in the previous exercises, do a formal conversion to the other model.

Give a state-transition graph or other equivalent representation for the following machines.

2 •• **MB2 (multipy-by-two)** This machine is a transducer with binary inputs and outputs, both **least-significant bit first**, producing a numeral that is twice the input. That is, if the input is $...x_2x_1x_0$ where x_0 is input first, then the output will be ... $x_2x_1x_00$ where 0 is output first, then x_0, etc. For example:

input	output	input decimal	output decimal
0	0	0	0
01	10	1	2
011	110	3	6
01011	10110	11	22
101011	010110	43	*incomplete*
0101011	1010110	43	86

first bit input ^

Notice that the full output does not occur until a step later than the input. Thus we need to input a 0 if we wish to see the full product. All this machine does is to reproduce the input delayed one step, after invariably producing a 0 on the first step. Thus this machine could also be called a **unit delay machine**.

Answer: Since this machine "buffers" one bit at all times, we can anticipate that two states are sufficient: r0 "remembers" that the last input was 0 and r1 remembers that the last input was 1. The output always reflects the state before the transition, i.e. outputs on arcs from r0 are 0 and outputs on arcs from r1 are 1. The input always takes the machine to the state that remembers the input appropriately.

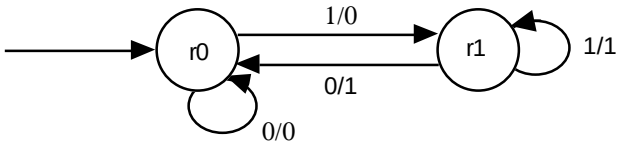


Figure 210: A multiply-by-2 machine

3 •• **MB2^n (multiply-by-2^n, where n is a fixed natural number)** (This is a separate problem for each n.) This machine is a transducer with binary inputs and outputs, both **least-significant bit first**, producing a numeral that is 2^n times as large the input. That is, if the input is ...$x_2x_1x_0$ where x_0 is input first, then the output will be ... $x_2x_1x_00$ where 0 is output first, then x_0, etc.

4 •• **Add1** This machine is a transducer with binary input and outputs, both **least-significant bit first**, producing a numeral that is 1 + the input.

Answer: The states of this machine will represent the value that is "carried" to the next bit position. Initially 1 is "carried". The carry is "propagated" as long as the input bits are 1. When an input bit of 0 is encountered, the carry is "absorbed" and 1 is output. After that point, the input is just replicated.

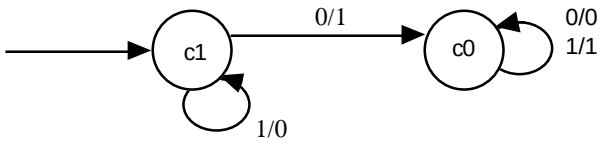


Figure 211: An add-1 machine

5 •• **W2 (Within 2)** This is an acceptor with input alphabet {0, 1}. It accepts those strings such that *for every prefix* of the string, the difference between the number of 0's and the number of 1's is always within 2. For example, 100110101 would be accepted but 111000 would not.

6 ••• **Add3** This machine is a transducer with binary input and outputs, both **least-significant bit first**, producing a numeral that is 3 + the input.

7 ••• **Add-n, where n is a fixed natural number** (This is a separate problem for each n.) This machine is a transducer with binary input and outputs, both **least-significant bit first**, producing a numeral that is n + the input.

8 •• **Binary adder** This is a transducer with binary inputs occurring in pairs. That is, the input alphabet is all pairs over {0, 1}: {(0, 0), (0, 1), (1, 0), (1, 1)}. The inputs are to be interpreted as bits of two binary numerals, least-significant bit first as in the previous problem. The output is a numeral representing the sum of the inputs, also least-significant bit first. As before, we need to input a final (0, 0) if we wish to see the final answer.

		decimal value of	
input	output	input	output
(0, 0)	0	0, 0	0
(0, 1)	1	0, 1	1
(0, 0)(1, 1)	10	1, 1	2
(0, 0)(1, 1)(0, 0)	100	2, 2	4
(0, 0)(1, 1)(1, 1)	110	3, 3	6
first input pair ^			

Answer: Apparently only the value of the "carry" needs to be remembered from one state to the next. Since only two values of carry are possible, this tells us that two states will be adequate.

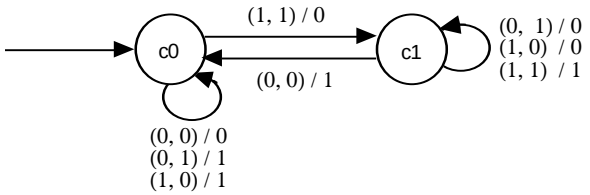


Figure 212: Serial binary adder, least-significant bits first

9 ••• **MB3 (multiply-by-three)** Similar to MB2, except the input is multiplied by 3. For example

		decimal value of	
input	output	input	output
0	0	0	0
01	11	1	3
010	110	2	6
001011	100001	11	33

Note that two final 0's might be necessary to get the full output. Why?

10 •••• **MBN (multiply-by-n, where n is a fixed natural number)** (This is a separate problem for each n.) This machine is a transducer with binary input and outputs, both **least-significant bit first**, producing a numeral that is n times the input.

11 ••• **Binary maximum** This is similar to the adder, but the inputs occur **most-significant digit first** and both inputs are assumed to be the same length numeral.

		decimal value of	
input	output	input	output
(0, 0)	0	0, 0	0
(0, 1)	1	0, 1	1
(0, 1)(1, 1)	11	1, 3	3
(0, 1)(1, 1)(1, 0)110	3, 6	6	
(1, 1)(1, 0)(0, 1)110	6, 5	6	

^ first input pair

12 •• **Maximum classifier** This is a classifier version of the preceding. There are three possible outputs assigned to a state: {tie, 1, 2}, where 1 indicates that the first input sequence is greater, 2 indicates the second is greater, and 'tie' indicates that the two inputs are equal so far.

input	class
(1, 1)	tie
(1, 1)(0, 1)	2
(1, 1)(0, 1)(1, 1)	2
(1, 0)(0, 1)(1, 1)	1

13 •• **1DB3 (Unary divisible by 3)** This is an acceptor with input alphabet {1}. It accepts exactly those strings having a multiple of three 1's (including λ).

14 ••• **2DB3 (Binary divisible by 3)** This is an acceptor with input alphabet {0, 1}. It accepts exactly those strings that are a numeral representing a multiple of 3 in binary, least-significant digit first. (Hint: Simulate the division algorithm.) Thus the accepted strings include: 0, 11, 110, 1001, 1100, 1111, 10010, ...

15 ••• **Sequential combination locks** (an infinite family of problems): A single string over the alphabet is called the "combination". Any string *containing* this combination is accepted by the automaton ("opens the lock"). For example, for the combination 01101, the acceptor is:

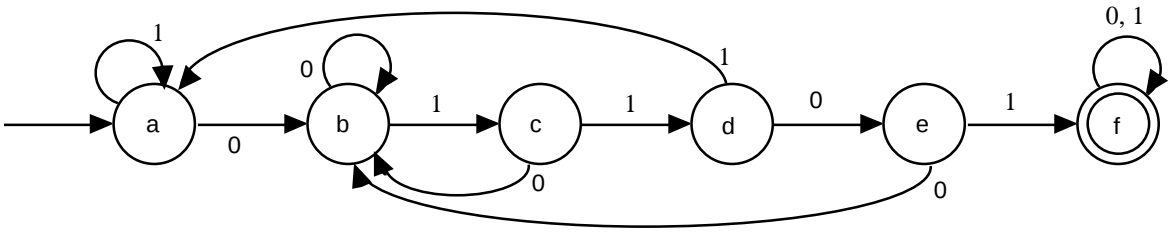


Figure 213: A combination lock state diagram

> The tricky thing about such problems is the construction of the backward arcs; they do not necessarily go back to the initial state if a "wrong" digit is entered, but only back to the state that results from the longest usable suffix of the digits entered so far. The construction can be achieved by the "subset" principle, or by devising an algorithm that will produce the state diagram for any given combination lock problem. This is what is done in a string matching algorithm known as the "Knuth-Morris-Pratt" algorithm.

Construct the state-diagram for the locks with the following different combinations: 1011; 111010; 010010001.

16 ••• Assume three different people have different combinations to the same lock. Each combination enters the user into a different security class. Construct a classifier for the three combinations in the previous problem.

17 ••• The preceding lock problems assume that the lock stays open once the combination has been entered. Rework the example and the problems assuming that the lock shuts itself if more digits are entered after the correct combination, until the combination is again entered.

12.2 Finite-State Grammars and Non-Deterministic Machines

An alternate way to define the language accepted by a finite-state acceptor is through a grammar. In this case, the grammar can be restricted to have a particular form of production. Each production is either of the form:

$$N \rightarrow \sigma M$$

where N and M are auxiliaries and σ is a terminal symbol, or of the form

$$N \rightarrow \lambda$$

recalling that λ is the empty string. The idea is that **auxiliary symbols are identified with states**. The start state is the start symbol. For each transition in an acceptor for the language, of the form

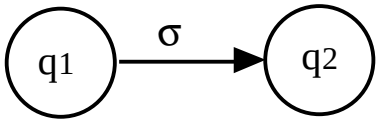


Figure 214: State transition corresponding to a grammar production

there is a corresponding production of the form

$$q_1 \rightarrow \sigma q_2$$

In addition, if q_2 happens to be an accepting state, there is also a production of the form.

$$q_2 \rightarrow \lambda$$

Example: Grammar from Acceptor

For our acceptor for exactly one edge, we can apply these two rules to get the following grammar for generating all strings with one edge:

The start state is *a*. The auxiliaries are {a, b, c, d, e, f}. The terminals are {0, 1}. The productions are:

a ⟶ 0b	c ⟶ 0e
a ⟶ 1c	c ⟶ 1c
b ⟶ 0b	e ⟶ 0e
b ⟶ 1d	e ⟶ 1f
d ⟶ 0f	e ⟶ λ
d ⟶ 1d	f ⟶ 0f
d ⟶ λ	f ⟶ 1f

To see how the grammar derives the 1-edged string 0011 for example, the derivation tree is:

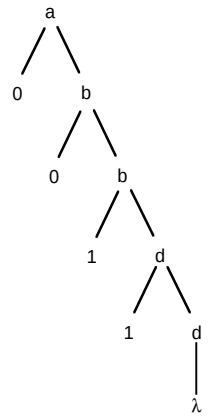


Figure 215: Derivation tree in the previous finite-state grammar, deriving the 1-edged string 0011

While it is easy to see how a finite-state grammar is derived from any finite-state acceptor, the converse is not as obvious. Difficulties arise in productions that have the same lefthand-side with the same terminal symbol being produced on the right, e.g. in a grammar, nothing stops us from using the two productions

a ⟶ 0b
a ⟶ 0c

Yet this would introduce an anomaly in the state-transition diagram, since when given input symbol 0 in state a, the machine would not know to which state to go next:

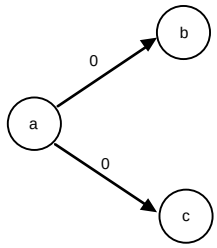


Figure 216: A non-deterministic transition

This automaton would be regarded as non-deterministic, since the next state given input 0 is not determined. Fortunately, there is a way around this problem. In order to show it, we first have to define the notion of "acceptance" by a non-deterministic acceptor.

> A non-deterministic acceptor **accepts** a string if there is some path from a starting state to an accepting state having a sequence of arc labels equal to that string.

We say *a* starting state, rather than *the* starting state, since a non-deterministic acceptor is allowed to have multiple starting states. It is useful to also include λ transitions in non-deterministic acceptors. These are arcs that have λ as their label. Since λ designates the empty string, these arcs can be used in a path but do not contributed any symbols to the sequence.

Example: Non-deterministic to Deterministic Conversion

Recall that a string in the language generated by a grammar consists only of terminal symbols. Suppose the productions of a grammar are (with start symbol a, and terminal alphabet {0, 1}):

$a \rightarrow 0d$	$b \rightarrow 1c$
$a \rightarrow 0b$	$b \rightarrow 1$
$a \rightarrow 1$	$d \rightarrow 0d$
$c \rightarrow 0b$	$d \rightarrow 1$

The language defined by this grammar is the set of all strings ending in 1 that either have exactly one 1 or that consist of alternating 01. The corresponding (non-deterministic) automaton is:

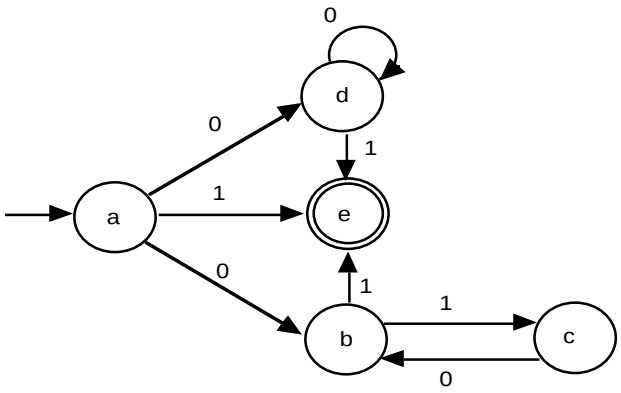


Figure 217: A non-deterministic automaton that accepts the set of all strings ending in 1that have exactly one 1 or consist of an alternating 01's.

There are two instances of non-determinism identifiable in this diagram: the two 0-transitions leaving a and the two 1-transitions leaving b. Nonetheless, we can derive from this diagram a corresponding deterministic finite-automaton. The derivation results in the deterministic automaton shown below.

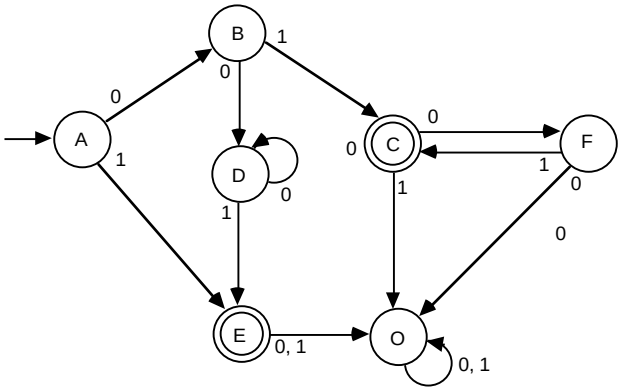


Figure 218: A deterministic automaton that accepts the set of all strings ending in 1 that have exactly one 1 or consist of an alternating 01's.

We can derive a deterministic automaton D from the non-deterministic one N by using *subsets* of the states of N as states of D. In this particular example, the subset associations are as follows:

A ~ {a}
B ~ {b, d}
C ~ {c, e}
D ~ {d}
E ~ {e}
F ~ {b}
O ~ {}

General method for deriving a deterministic acceptor D from a non-deterministic one N:

The **state set** of D is the set of all *subsets* of N.

The **initial state** of D is the set of all initial states of N, together with states reachable from initial states in N using only λ transitions.

There is a **transition** from a set S to a set T of D with label σ (where σ is a single input symbol).

T = {q' | there is a q in S with a sequence of transitions from q to q'
corresponding to a one symbol string σ}

The reason we say *sequence* is due to the possibility of λ transitions; these do not add any new symbols to the string. Note that λ is not regarded as an input symbol.

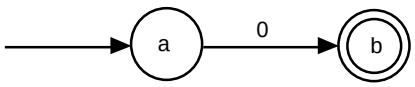
The **accepting states** of the derived acceptor are those that contain at least one accepting state of the original acceptor.

In essence, what this method does is "compile" a breadth-first search of the non-deterministic state graph into a deterministic finite-state system. The reason this works is that the set of all subsets of a finite set is finite.

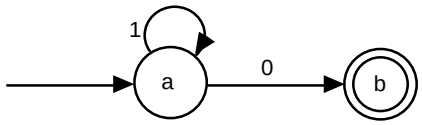
Exercises

Construct deterministic acceptors corresponding to the following non-deterministic acceptors, where the alphabet is {0, 1}.

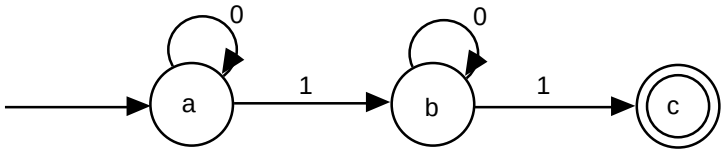
1 •



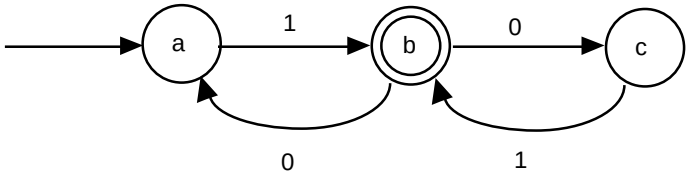
2 •



3 ••



4 ••



12.3 Meaning of Regular Expressions

From the chapter on grammars, you already have a good idea of what regular expressions mean already. While a grammar for regular expression was given in that chapter, for purposes of giving a meaning to regular expressions, it is more convenient to use a grammar that is intentionally ambiguous, expressing constructs in pairs rather than in sequences:

$R \rightarrow$ 'λ'
$R \rightarrow \varnothing$
$R \rightarrow \sigma$, for each letter σ in A
$R \rightarrow R\ R$ // juxtaposition
$R \rightarrow (\ R \mid R\ )$ // alternation
$R \rightarrow (R)^*$ // iteration

To resolve the ambiguity in the grammar, we simply "overlay" on the grammar some conventions about precedence. The standard precedence rules are:

* binds more tightly than either juxtaposition or |

juxtaposition binds more tightly than |

We now wish to use this ambiguous grammar to assign a meaning to regular expressions. With each expression E, the meaning of E is a language, i.e. set of strings, over the alphabet A. We define this meaning recursively, according to the structure of the grammar:

Basis:

- L(λ) is {λ}, the set consisting of one string, the empty string λ.
- L($\varnothing$) is $\varnothing$, the empty set.
- For each **letter** σ in A, and L(σ) is { σ }, the set consisting of one string of one letter, σ.

Induction rules:

- L(RS) = L(R)L(S), where by the latter we mean the set of all strings of the form of the concatenation rs, where r ∈ L(R) and s ∈ L(S).
- L(R | S) = L(R) ∪ L(S).
- L(R*) = L(R)*

To clarify the first bullet, for any two languages L and M, the "set concatenation" LM is defined to be {rs | r ∈ L and s ∈ M}. That is, the "concatenation" of two sets of strings is the set of all possible concatenations, one string taken from the first set and another taken from the second. For example,

{0}{1} is defined to be {01}.

{0, 01}{1, 10} is defined to be {01, 010, 011, 0110}.

{01}{0, 00, 000, ...} is defined to be {010, 0100, 01000, ...}.

To explain the third bullet, we need to define the * operator on an arbitrary language. If L is a language, the L* is defined to be (using the definition of concatenation above)

{λ} ∪ L ∪ LL ∪ LLL ∪ ...

That is, L* consists of all strings formed by concatenating zero or more strings, each of which is in L.

Regular Expression Examples over alphabet {a, b, c}

Expression	**Set denoted**
a | b | c	The set of 1-symbol strings {"a", "b", "c"}
λ | (a | b | c) | (a | b | c)(a | b | c)	The set of strings with two or fewer symbols
a*	The set of strings using only symbol *a*
a*b*c*	The set of strings in which no *a* follows a *b* and no *a* or *b* follows a *c*
(a | b)*	The set of strings using only *a* and *b*.
a* | b*	The set of strings using only *a* or only *b*
(a | b | c)(a | b | c)(a | b | c)*	The set of strings with at least two symbols.
((b | c)* ab (b | c)*)*	The set of strings in which each *a* is immediately followed by a *b*.
(b | c)* | ((b | c)* a (b | c) (b | c)*)* (λ | a)	The set of strings with no two consecutive *a*'s.

Regular expressions are finite symbol strings, but the sets they denote can be finite or infinite. Infinite sets arise only by virtue of the * operator (also sometimes called the Kleene-star operator).

Identities for Regular Expressions

One good way of becoming more familiar with regular expressions is to consider some identities, that is equalities between the sets described by the expressions. Here are some examples. The reader is invited to discover more.

For any regular expressions R and S:

$R \mid S = S \mid R$

$R \mid \varnothing = R$ $\qquad$ $\varnothing \mid R = R$

$R\lambda = R$ $\qquad$ $\lambda R = R$

$R \varnothing = \varnothing$ $\qquad$ $\varnothing R = \varnothing$

$\lambda^* = \lambda$

$\varnothing^* = \lambda$

$R^* = \lambda \mid RR^*$

$(R \mid \lambda)^* = R^*$

$(R^*)^* = R^*$

Exercises

1 •• Determine whether or not the following are valid regular expression identities:

$\lambda\varnothing = \lambda$

$R (S \mid T) = RS \mid RT$

$R^* = \lambda \mid R^*R$

$RS = SR$

$(R \mid S)^* = R^* \mid S^*$

$R^*R = RR^*$

$(R^* \mid S^*)^* = (R \mid S)^*$

$(R^*S^*) = (R \mid S)^*$

For any n, $R^* = \lambda \mid R \mid RR \mid RRR \mid .... \mid R^{n-1} \mid R^nR^*$, where R^n is an abbreviation for RR....R. (n times)

2 ••• Equations involving languages with a language as an unknown can sometimes be solved using regular operators. For example,

$S = RS \mid T$

can be solved for unknown S by the solution $S = R^*T$. Justify this solution.

3 ••• Suppose that we have grammars that respectively generate sets L and M as languages. Show how to use these grammars to form a grammar that generates each of the following languages

$L \cup MLM$ L^*

Regular Languages

The regular operators (|, *, and concatenation) are applicable to any languages. However, a special name is given to languages that can be constructed using only these operators and languages consisting of a single string, and the empty set.

Definition: A language (set of strings over a given alphabet) is called *regular* if it is a set of strings denoted by some regular expression. (A regular language is also called a *regular set*.)

Let us informally explore the relation between regular languages and finite-state acceptors. The general idea is that the regular languages exactly characterize sets of paths from the initial state to *some* accepting state. We illustrate this by giving an acceptor for each of the above examples.

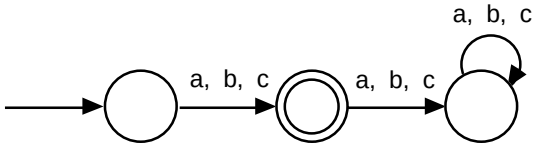


Figure 219: Acceptor for a | b | c

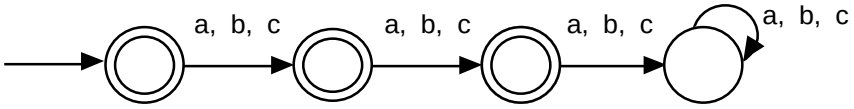


Figure 220: Acceptor for λ | (a | b | c) | (a | b | c)(a | b | c)

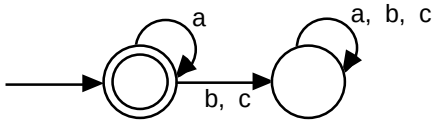


Figure 221: Acceptor for a*

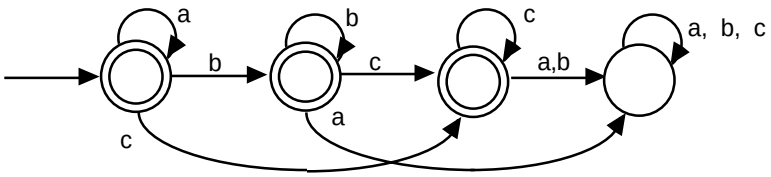


Figure 222: Acceptor for a*b*c*

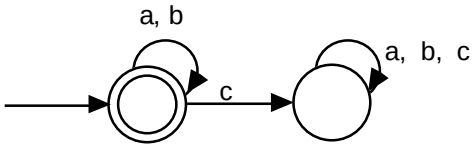


Figure 223: Acceptor for (a | b)*

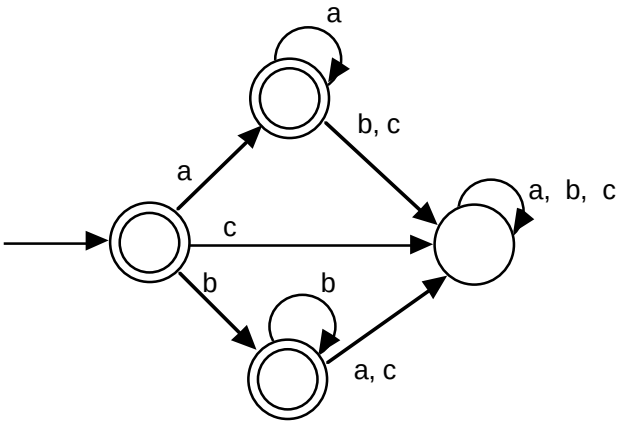


Figure 224: *Acceptor for* a* | b*

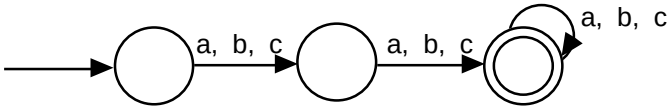


Figure 225: Acceptor for (a | b | c)(a | b | c)(a | b | c)*

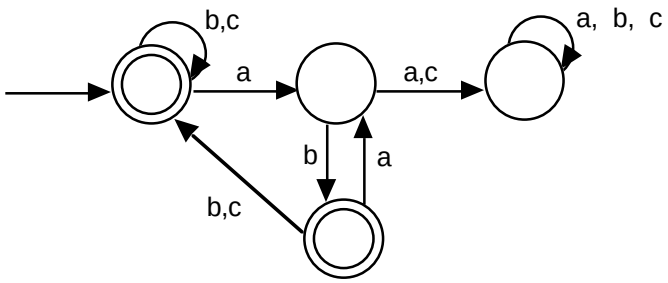


Figure 226: Acceptor for ((b | c)* ab (b | c))*

As we can see, the connection between regular expressions and finite-state acceptors is rather close and natural. The following result makes precise the nature of this relationship.

> **Kleene's Theorem** (Kleene, 1956) A language is regular iff it is accepted by some finite-state acceptor.

The "if" part of Kleene's theorem can be shown by an algorithm similar to Floyd's algorithm. The "only if" part uses the non-deterministic to deterministic transformation.

The Language Accepted by a Finite-State Acceptor is Regular

The proof relies on the following constructive method:

> Augment the graph of the acceptor with a single distinguished starting node and accepting node, connected via λ-transitions to the original initial state and accepting states in the manner shown below. The reason for this step is to isolate the properties of being initial and accepting so that we can more easily apply the transformations in the second step.

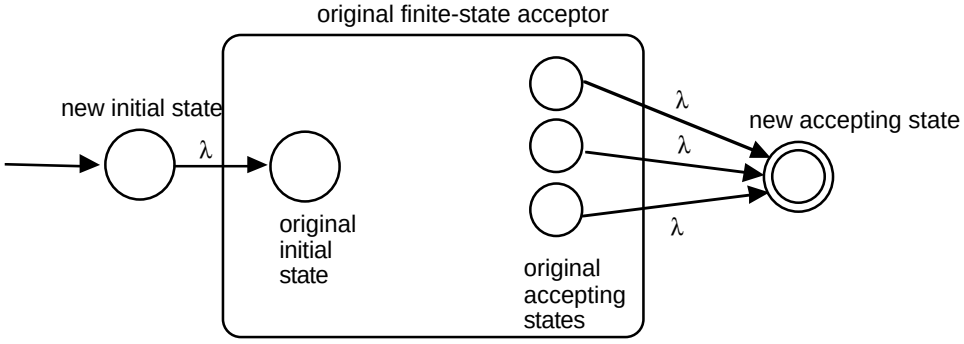


Figure 227: Modifying acceptor in preparation for deriving the regular expression

One at a time, eliminate the nodes in the original acceptor, preserving the set of paths through each node by recording an appropriate regular expression between each pair of other nodes. When this process is complete, the regular expression connecting the initial state to the accepting state is the regular expression for the language accepted by the original finite-state machine.

To make the proof complete, we have to describe the node elimination step. Suppose that prior to the elimination, the situation is as shown below.

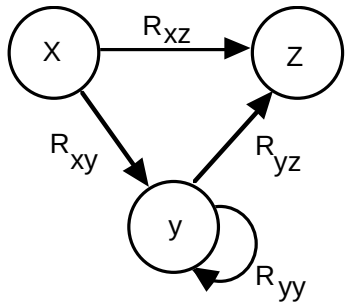


Figure 228: A situation in the graph before elimination of node y

Here x and z represent arbitrary nodes and y represents the node being eliminated. A variable of the form R_{ij} represents the regular expression for paths from i to j using nodes previously eliminated. Since we are eliminating y, we replace the previous expression R_{xz} with a new expression

$$R_{xz} \mid R_{xy} R_{yy}^{*} R_{yz}$$

The rationale here is that Rxz represents the paths that were there before, and $R_{xy} R_{yy}^{*} R_{yz}$ represents the paths that went through the eliminated node y.

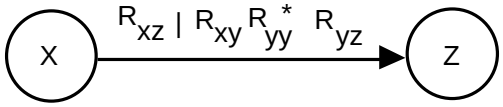


Figure 229: The replacement situation after eliminating node y

The catch here is that we must perform this updating for every pair of nodes x, z, including the case where x and z are the same. In other words, if there are m nodes left, then m^2 regular expression updates must be done. Eliminating each of n nodes then requires $O(n^3)$ steps. The entire elimination process is very similar to the Floyd and Warshall algorithms discussed in the chapter on Complexity. The only difference is that here we are dealing with the domain of regular expressions, whereas those algorithms dealt with the domains of non-negative real numbers and bits respectively.

Prior to the start of the process, we can perform the following simplification:

> Any states from which no accepting state is reachable can be eliminated, along with arcs connecting to or from them.

Example: Regular Expression Derivation

Derive a regular expression for the following finite-state acceptor:

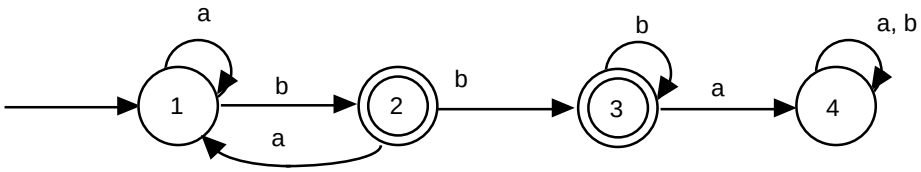


Figure 230: A finite-state acceptor from which a regular expression is to be derived

First we simplify by removing node 4, from which no accepting state is reachable. Then we augment the graph with two new nodes, 0 and 5, connected by λ-transitions. Notice that for some pairs of nodes there is no connection. This is equivalent to the corresponding regular expression being ∅. Whenever ∅ is juxtaposed with another regular expression, the result is equivalent to ∅. Similarly, whenever λ is juxtaposed with another regular expression R, the result is equivalent to R itself.

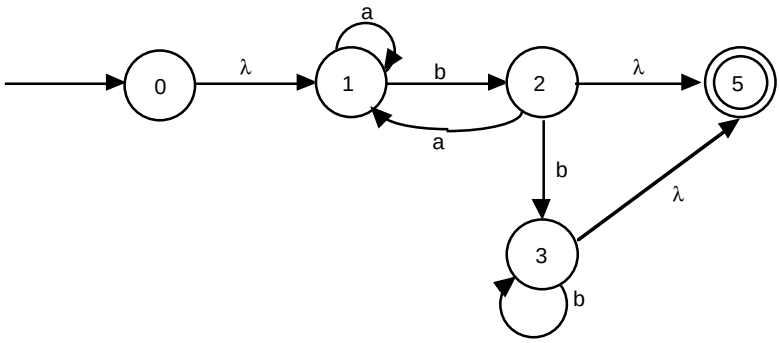


Figure 231: The first step in deriving a regular expression. Nodes 0 and 5 are added.

Now eliminate one of the nodes 1 through 3, say node 1. Here we will use the identity that states λa*b = a*b.

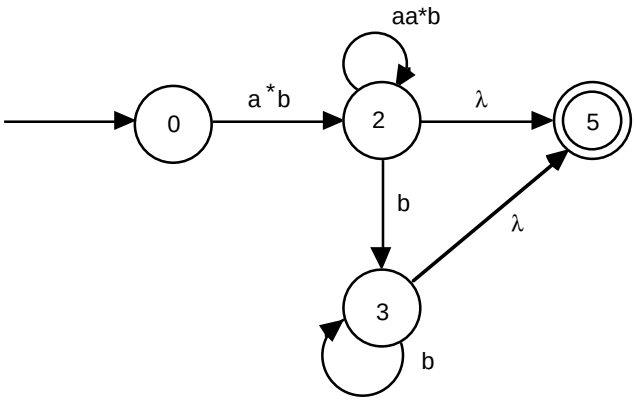


Figure 232: After removal of node 1

Next eliminate node 2.

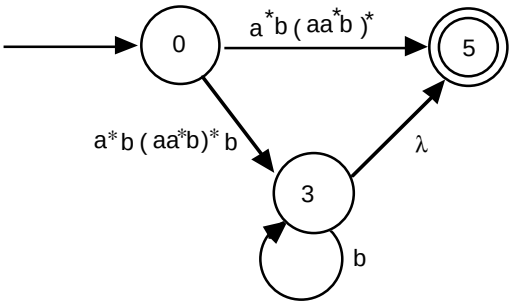


Figure 233: After removal of node 2

Finally eliminate node 3.

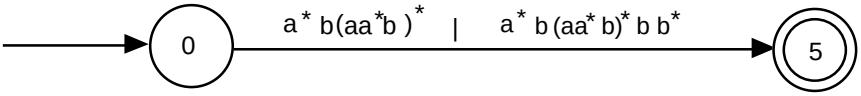


Figure 234: After removal of node 1

The derived regular expression is

$$a^*b(aa^*b)^* \mid a^*b(aa^*b)^*bb^*$$

Every Regular Language is Accepted by a Finite-State Acceptor

We already know that we can construct a deterministic finite-state acceptor equivalent to any non-deterministic one. Hence it is adequate to show how to derive a non-deterministic finite-state acceptor from a regular expression. The paths from initial node to accepting node in the acceptor will correspond in an obvious way to the strings represented by the regular expression.

Since regular expressions are defined inductively, it is very natural that this proof proceed along the same lines as the definition. We expect a basis, corresponding to the base cases λ, $\varnothing$, and σ (for σ each in A). We then assume that an acceptor is constructable for regular expressions R and S and demonstrate an acceptor for the cases RS, R | S, and R^*. The only thing slightly tricky is connecting the acceptors in the inductive cases. It might be necessary to introduce additional states in order to properly isolate the paths in the constituent acceptors. Toward this end, we stipulate that

(i) the acceptors constructed shall always have a single initial state and single accepting state.

(ii) no arc is directed from some state into the initial state

Call these **property P**.

Basis: The acceptors for λ, $\varnothing$, and σ (for σ each in A) are as shown below:

Figure 235: Acceptor for $\varnothing$ with property P

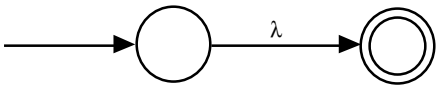


Figure 236: Acceptor for λ (the empty sequence) with property P

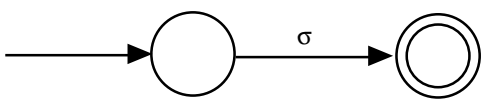


Figure 237: Acceptor for σ (where $\sigma \in$ A) with property P

Induction Step: Assume that acceptors for R and S, with property P above, have been constructed, with their single initial and accepting states as indicated on the left and right, respectively.

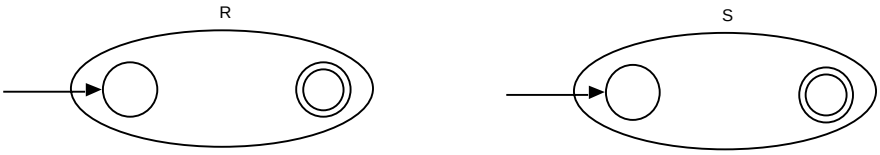


Figure 238: Acceptors assumed to exist for R and S respectively, having property P

Then for each of the cases above, we construct new acceptors that accept the same language and which have property P, as now shown:

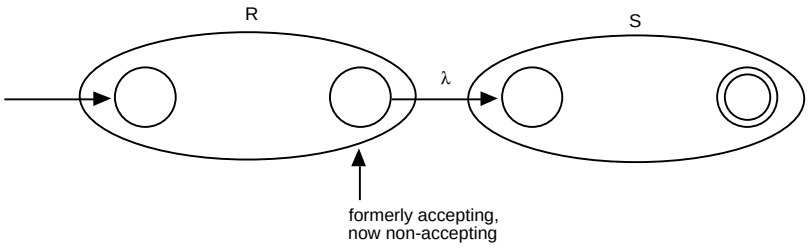


Figure 239: Acceptor for RS, having property P

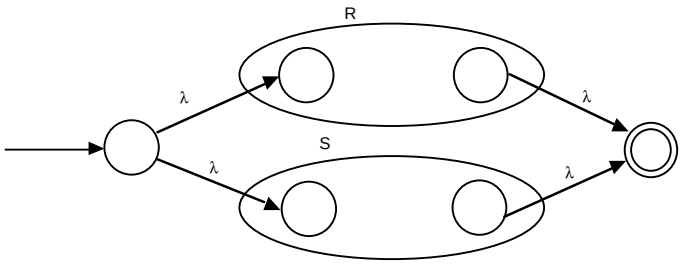


Figure 240: Acceptor for R | S, having property P

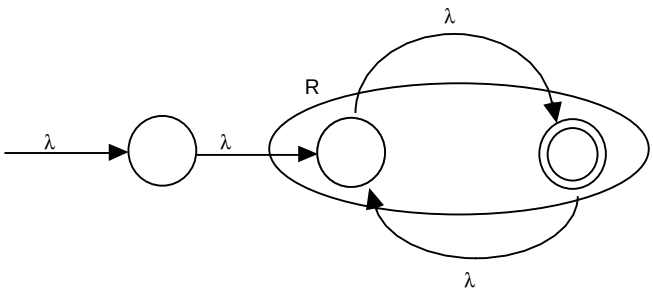


Figure 241: Acceptor for R*, having property P

Regular expressions in UNIX®

Program *egrep* is one of several UNIX® tools that use some form of regular expression for pattern matching. Other such tools are *ed*, *ex*, *sed*, *awk*, and *archie*. The notations appropriate for each tool may differ slightly. Possible usage:

egrep *regular-expression filename*

searches the file line-by-line for lines containing strings matching the regular-expression and prints out those lines. The scan starts anew with each line. In the following description, 'character' means excluding the newline character:

A single character not otherwise endowed with special meaning matches that character. For example, 'x' matches the character x.

The character '.' matches any character.

A regular expression followed by an * (asterisk) matches a sequence of 0 or more matches of the regular expression.

Effectively a regular expression used for searching is preceded and followed by an implied .*, meaning that any sequence of characters before or after the string of interest can exist on the line. To exclude such sequences, use ^ and $:

The character ^ matches the beginning of a line.

The character $ matches the end of a line.

A regular expression followed by a + (plus) matches a sequence of 1 or more matches of the regular expression.

A regular expression followed by a ? (question mark) matches a sequence of 0 or 1 matches of the regular expression.

A \ followed by a single character other than newline matches that character. This is used to escape from the special meaning given to some characters.

A string enclosed in brackets [] matches any single character from the string. Ranges of ASCII character codes may be abbreviated as in a-z0-9, which means all characters in the range a-z and 0-9. A literal - in such a context must be placed after \ so that it can't be mistaken as a range indicator.

Two regular expressions concatenated match a match of the first followed by a match of the second.

Two regular expressions separated by | or newline match either a match for the first or a match for the second.

A regular expression enclosed in parentheses matches a match for the regular expression.

The order of precedence of operators at the same parenthesis level is [] then *+? then concatenation then | and newline.

Care should be taken when using the characters $ * [ ] ^ | ( ) and \ in the expression as they are also meaningful to the various shells. **It is safest to enclose the entire expression argument in single quotes**.

Examples: UNIX Regular Expressions	
Description of lines to be selected	**Regular Expression**
containing the letters `qu` in combination	`qu`
beginning with `qu`	`^qu`
ending with `az`	`az$`
beginning with `qu` and ending with `az`	`^qu.*az$`
containing the letters `qu` or `uq`	`uq|qu`
containing two or more `a`'s in a row	`a.*a`
containing four or more `i`'s	`i.*i.*i.*i`
containing five or more `a`'s and `i`'s	`[ai].*[ai].*[ai].*[ai].*[ai]`
containing `ai` at least twice	`(ai).*(ai)`
containing `uq` or `qu` at least twice	`(uq|qu).*(uq|qu)`

Exercises

Construct deterministic finite-state acceptors for the following regular expressions:

1 • 0*1*

2 • (0*1*)*

3 •• (01 | 011)*

4 •• (0* | (01)*)*

5 •• (0 | 1)*(10110)(0 | 1)*

6 ••• The regular operators are concatenation, union (|), and the * operator. Because any combination of regular languages using these operators is itself a regular language, we say that the regular languages are *closed under* the regular operators. Although intersection and complementation (relative to the set of all strings, Σ^*) are not included among the regular languages, it turns out that the regular languages are closed under these operators as well. Show that this is true, by using the connection between regular languages and finite-state acceptors.

7 ••• Devise a method for determining whether or not two regular expressions denote the same language.

8 •••• Construct a program that inputs a regular expression and outputs a program that accepts the language denoted by that regular expression.

9 ••• Give a UNIX regular expression for lines containing floating-point numerals.

12.4 Synthesizing Finite-State Machines from Logical Elements

We now wish to extend techniques for the implementation of functions on finite domains in terms of logical elements to implementing finite-state machines. One reason that this is important is that digital computers are constructed out of collections of finite-state machines interconnected together.As already stated, the input sequences for finite-state machines are elements of an infinite set Σ^*, where Σ is the input alphabet. Because the output of the propositional functions we studied earlier were simply a *combination* of the input values, those functions are called **combinational**, to distinguish them from the more general functions on Σ^*, which are called **sequential**.

We will show how the implementation of machines can be decomposed into combinational functions and memory elements, as suggested by the equation

Sequential Function = Combinational Functions + Memory

Recall the earlier structural diagrams for transducer and classifiers, shown as "feedback" systems. Note that these two diagrams share a common essence, namely the next-state portion. Initially, we will focus on how just this portion is implemented. The rest of the machine is relatively simple to add.

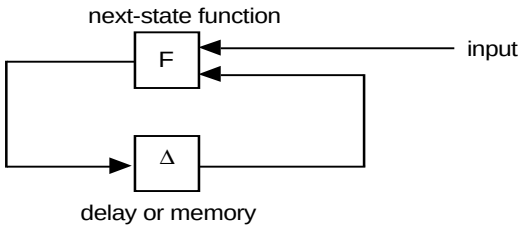


Figure 242: The essence of finite-state machine structure

Implementation using Logic Elements

Before we can "implement" such a diagram, we must be clearer on what items correspond to *changes* of input, output, and state. The combinational logical elements, such as AND-gates and OR-gates, as discussed earlier are abstractions of physical devices. In those devices, the logical values of 0 and 1 are interpretations of physical states. The output of a device is a function of its inputs, *with some qualification.* No device can change state instantaneously. When the input values are first presented, the device's output might be in a different state from that indicated by the function. There is some *delay time* or *switching time* associated with the device that must elapse before the output stabilizes to the value prescribed by the function. Thus, each device has an inherent *sequential* behavior, even if we choose to think of it as a combinational device.

Example Consider a 2-input AND-gate. Suppose that a device implements this gate due to our being able to give logical values to two voltages, say LO and HI, which correspond to 0 and 1 respectively. Then, observed over time, we might see the following behavior of the gate in response to changing inputs. The arrows in the diagram indicate a causal relationship between the input changes and output changes. Note that there is always some delay associated with these changes.

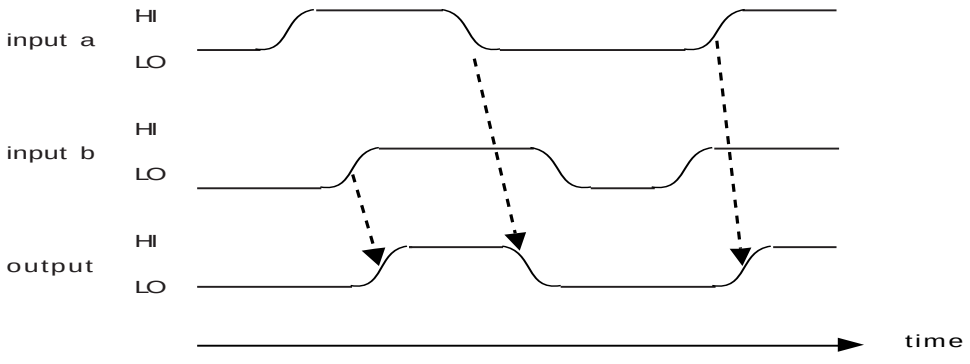


Figure 243: Sequential behavior of an AND-gate

Modeling the sequential behavior of a device can be complex. Computer designers deal with an abstraction of the behavior in which the outputs can only change at specific instants. This simplifies reasoning about behaviors. The abstract view of the AND-gate shown above can be obtained by straightening all of the changes of the inputs and outputs, to make it appear as if they were instantaneous.

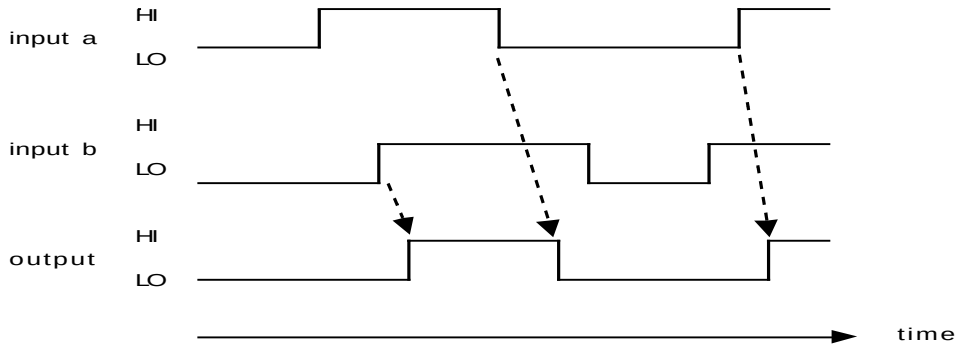


Figure 244: An abstraction of the sequential behavior of an AND-gate

Quantization and Clocks

In order to implement a sequential machine with logic gates, it is necessary to select a scheme for quantizing the values of the signal. As suggested by the preceding diagram, the signal can change *continuously*. On the other hand, the finite-state machine abstraction requires a series of discrete input and output values. For example, as we look at input *a* in the preceding diagram, do we say that the corresponding sequence is 0101 based on just the input changes? If that were the case, what would be the input corresponding to sequence 00110011? In other words, how do we know that a value that stays high for some time is a single 1 or a series of 1's? The most common means of resolving this issue is to use a system-wide clock as a timing standard. The clock "ticks" at regular intervals, and the value of a signal can be *sampled* when this tick occurs.

The effect of using a clock is to superimpose a series of tick marks atop the signals and agree that the discrete valued signals correspond to the values at the tick marks. Obviously this means that the discrete interpretation of the signals depends on the clock interval. For example, one quantization of the above signals is shown as follows:

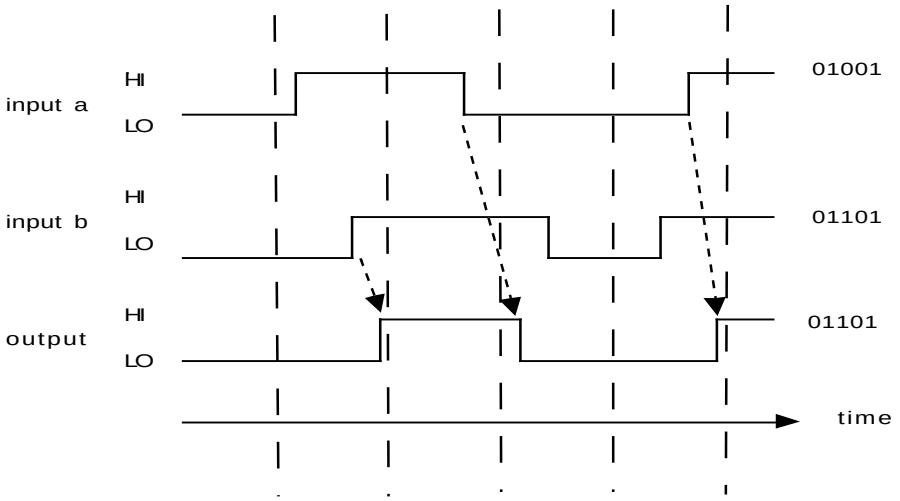


Figure 245: AND-gate behavior with one possible clock quantization

Corresponding to the five ticks, the first input sequence would be 01001, the second would be 01101, and the output sequence would be 01101. Notice that the output is not quite the AND function of the two inputs, as we might have expected. This is due to the fact that the second output change was about to take place when the clock ticked and the previous output value carried over. Generally we avoid this kind of phenomenon by designing such that the changes take place between ticks and at each tick the signals are, for the moment, stable.

The next figure shows the same signals with a slightly wider clock interval superimposed. In this instance, no changes straddle the clock ticks, and the input output sequences appear to be what is predicted by the definition of the AND function.

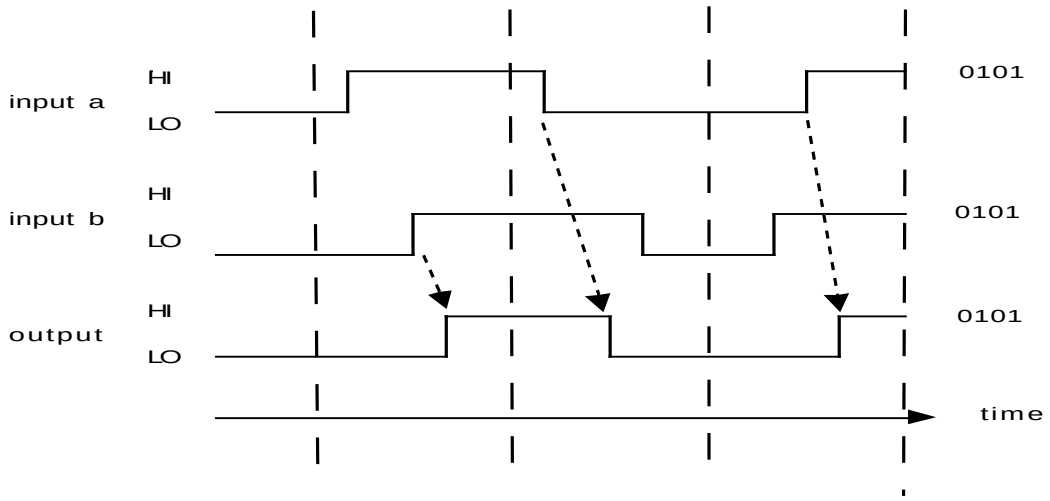


Figure 246: AND-gate behavior with wider quantization

Flip-Flops and Clocks

As stated above, in order to maintain the effect of instantaneous changes when there really is no such thing, the inputs of gates are only *sampled* at specific instants. By using the value of the sample, rather than the true signal, we can approach the effect desired. In order to hold the value of the sample from one instant to the next, a memory device known as a **D flip-flop** is used.

The D flip-flop has two different kinds of inputs: a signal input and a clock input. Whenever the clock "ticks", as represented, say, by the *rising* edge of a square wave, the signal input is sampled and held until the next tick. In other words, the flip-flop "remembers" the input value until the next tick; then it takes on the value at that time.

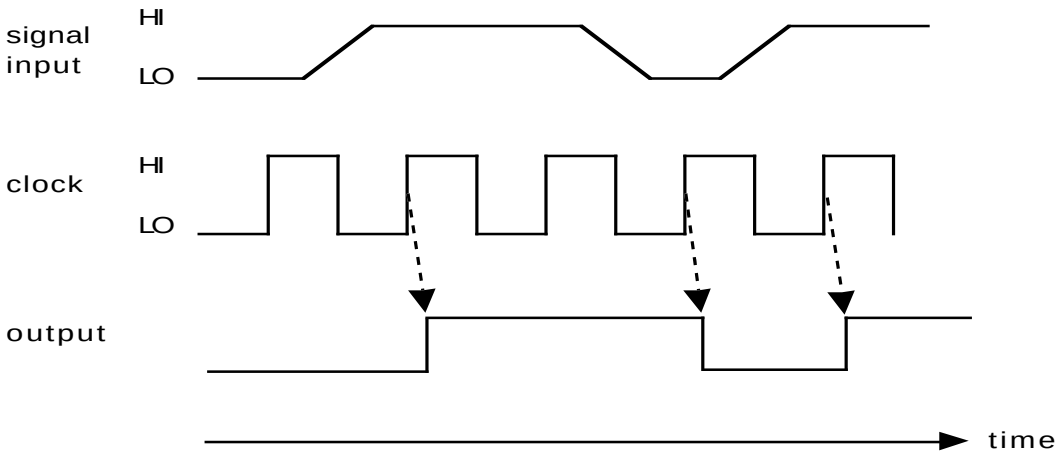


Figure 247: D flip-flop behavior:
The output of the flip-flop changes only in response to the rising edge of the clock, and will reflect the value of the signal input at that time.

Note that the signal input can change in between clock ticks, but it should not be changing near the same time. If it does, a phenomenon known as "meta-stability" can result, which can upset the abstraction as presented. We shall say more about this later.

Clock-based design is called ***synchronous design***. This is not the only form of design, but it is certainly the most prevalent, with at least 99% of computer design being based on this model. We will indicate more about the reasons for this later, but for now, synchronous design is the mode on which we will concentrate.

In synchronous design, the inputs to a device will themselves be outputs of flip-flops, and will change after the clock ticks. For example, the following diagram shows an AND-gate in the context of D flip-flops controlled by a common clock. The inputs to a and b are not shown. However, we assume that they change between clock ticks and thus the outputs of a and b will change right after the clock tick, as will the output c.

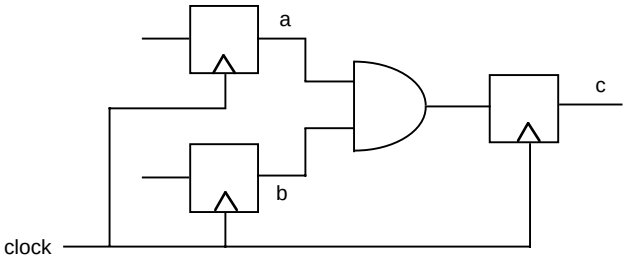


Figure 248: An AND-gate in a synchronous system.
Inputs to the flip-flops with outputs a and b are not shown.

The next diagram shows a sample behavior of the AND-gate in a synchronous system. This should be compared with the abstracted AND-gate presented earlier, to verify that synchronous design implements the abstract behavior.

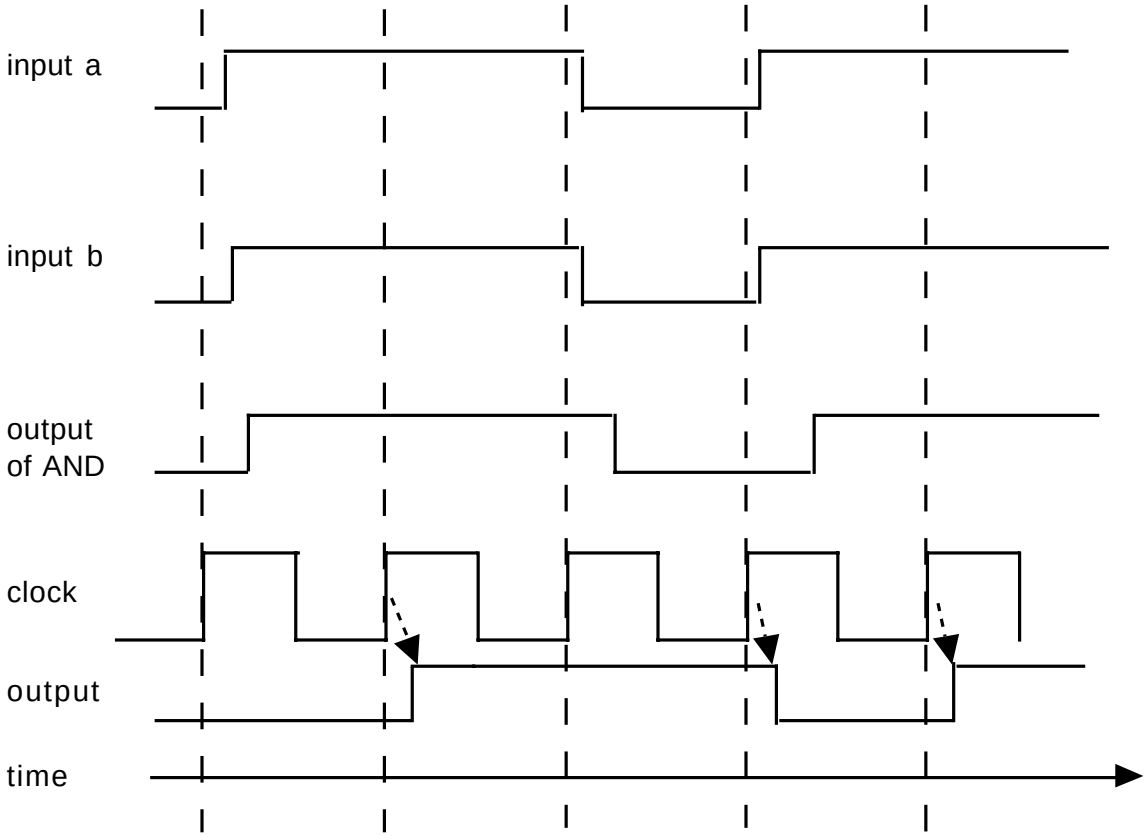


Figure 249: Example behavior of the AND-gate in a synchronous system. Note that the output only changes right after the rising edge of the clock.

The assumption in synchronous design is that the inputs to a device are held constant during each clock interval. During the interval itself, the device has an opportunity to change to the value represented by its logical function. In fact, the length of the interval is chosen in such a way that the device can achieve this value by the end of the interval. At the end of the interval, the output of the device will thus have stabilized. It can then be used as the input to some other device.

Closing the Loop

The previous example of an AND-gate in the context of three flip-flops can be thought of as a simple sequential machine. The state of the machine is held in the output flip-flop c. Thus, the current state always represents the logical AND of the inputs at the previous clock tick. In general, the state is a function of not just the inputs, but also the previous state. This can be accomplished by using the output of the state flip-flop to drive the next

state. An example obtained by modifying the previous example is shown below. Here we connect the output c to the place where the input a was.

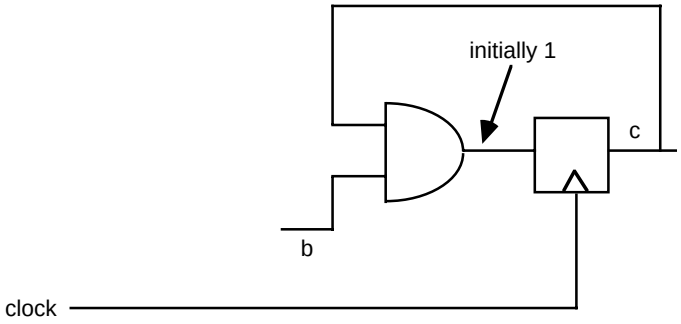


Figure 250: A sequential machine that remembers if b were ever 0

Suppose that we observe this machine from a point at which the output flip-flop c is 1. At the next clock tick, if b is 1, then the flip-flop will stay at 1. However, if b is 0, then the flip-flop will be switched to 0. Once flip-flop c is at 0, it will stay there forever, because no input value *anded* with 0 will ever give 1. The following diagram shows a possible behavior.

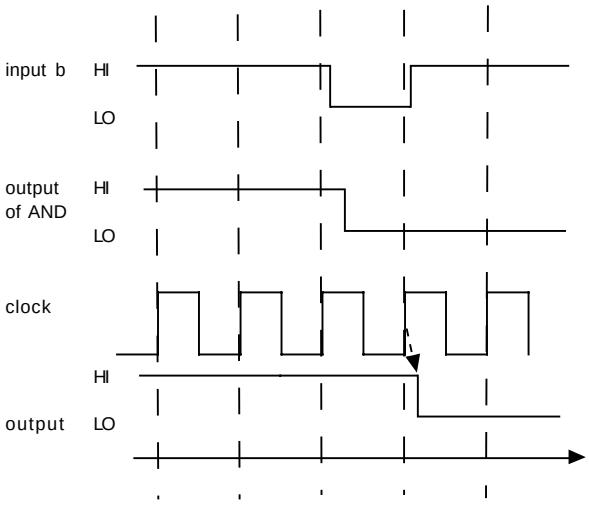


Figure 251: Example behavior of the previous sequential machine

The timing diagram above shows only one possible behavior. To capture all possible behaviors, we need to use the state-transition diagram, as shown below. Again, the state in this case is the output of flip-flop c.

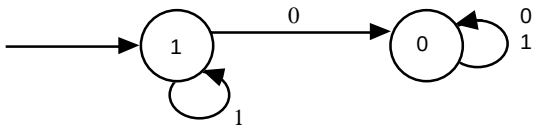


Figure 252: The state diagram for a machine that remembers if it ever saw 0

The state-transition structure of this machine is the same as that of a transducer that adds 1 to a binary representation, least-significant-bit-first:

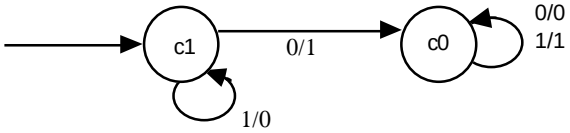


Figure 253: The add-1 transducer

Before giving the general method for synthesizing the logic from a state-transition behavior, we give a couple more examples of structure vs. function.

Sequential Binary Adder Example

This is the essence of the sequential binary adder that adds a pair of numerals together, least-significant bit first. Its state remembers the carry. We present both the full transducer and the abstracted state-transition behavior.

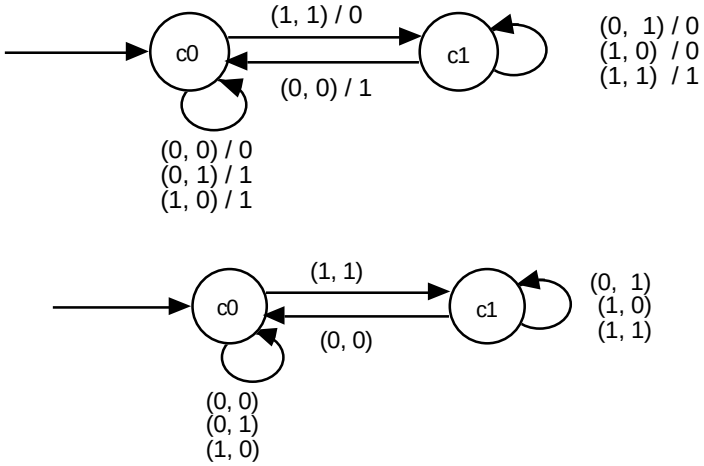


Figure 254: Transducer and state-transition behavior for the binary adder

The machine has 2 inputs that are used in parallel, one for each bit of the two addends. Assuming that we represent the carry by the 1 or 0 value of the output flip-flop, the structure of the adder can be realized as follows:

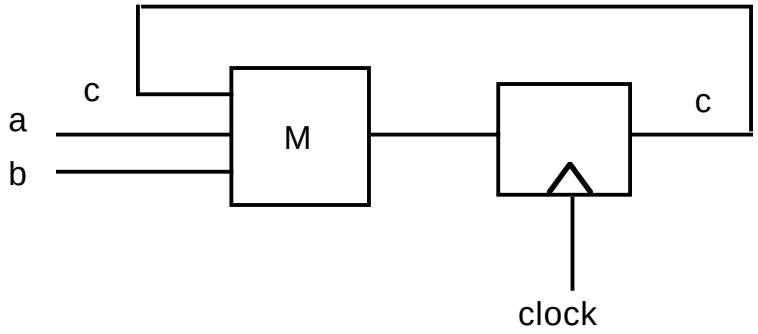


Figure 255: State-transition implementation of the binary adder

The box marked M is the majority combination function, as given by the following table:

a	**b**	**c**	**F**
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

As we can see, the output of function M is 1 iff at least two out of three inputs are 1. These combinations could be described by giving the minterm form, or the simplified form:

$$F(a, b, c) = ab + ac + bc$$

Were we to implement this M using AND- and OR- gates, the result would appear as:

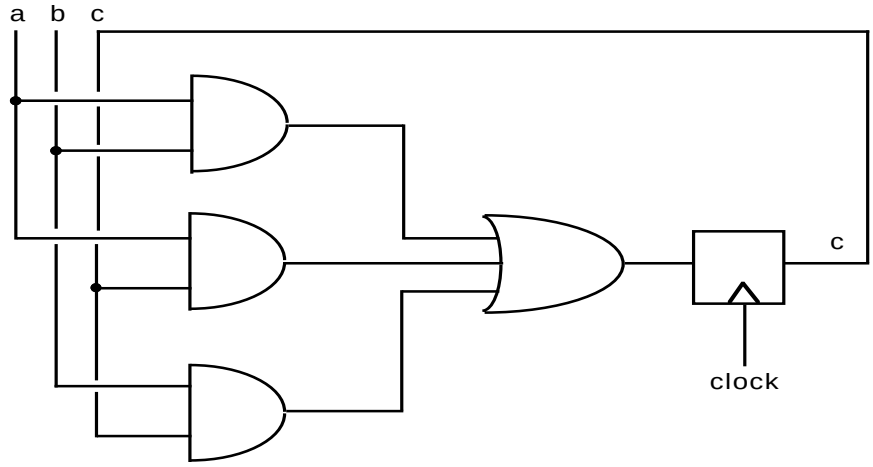


Figure 256: The binary adder state-transition behavior implemented using combinational gates and a flip-flop

Combination Lock Example

This example, a combination lock with combination 01101, was given earlier:

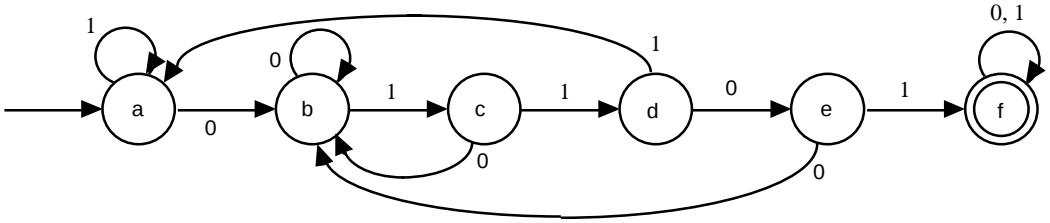


Figure 257: Combination lock state transitions

Suppose we encode the states by using three flip-flops, u, v, and w, as follows:

	flip-flops		
state name	**u**	**v**	**w**
a	0	0	0
b	0	0	1
c	0	1	0
d	0	1	1
e	1	0	0
f	1	1	1

From the tabular form for the state-transition function:

	next state as a function of input	
current state	**0**	**1**
a	b	a
b	b	c
c	b	d
d	e	a
e	b	f
f	f	f

we *transcribe* the table by substituting (e.g. by using a text editor) flip-flop values for each state. This is the same process we used in implementing combinational logic functions.

	next uvw as a function of input	
current uvw	**0**	**1**
000	001	000
001	001	010
010	001	011
011	100	000
100	001	111
111	111	111

For each flip-flop, we derive the next-state in terms of the current one simply by separating this table:

	next u as a function of input	
current uvw	**0**	**1**
000	0	0
001	0	0
010	0	0
011	1	0
100	0	1
111	1	1

Letting x represent the input, from this table, we can see that

next u = u'vwx' + uv'w'x + uvwx' + uvwx

(using the minterm form),but a simpler version derived from considering "don't cares" is:

next u = vwx' + ux

	next v as a function of input	
current uvw	**0**	**1**
000	0	0
001	0	1
010	0	1
011	0	0
100	0	1
111	1	1

From this table, we can derive:

next v = u'v'wx + u'vw'x + uv'w'x + uvw

	next w as a function of input	
current uvw	**0**	**1**
000	1	0
001	1	0
010	1	1
011	0	0
100	1	1
111	1	1

From this table, we can derive the simplified form:

next w =v'x' + vw' + u

Putting these together, we can realize the combination lock as shown on the next page.

12.5 Procedure for Implementing a State-Transition Function

To implement a state-transition function for a finite-state machine in terms of combinational logic and flip-flops:

1. Choose encodings for the input alphabet Σ and the state set Q.

2. Transcribe the table for the state-transition function F: $Q \times \Sigma \rightarrow Q$ into propositional logic functions using the selected encodings.

3. Implement the transcribed F functions from logical elements

4. The functions thus implemented are used as inputs to a bank of D flip-flops, one per bit in the state encoding.

Inclusion of Output Functions

In order to synthesize a finite-state machine having output, we need to augment the state-transition implementation with an output function implementation. Fortunately, the output function is simply a combinational function of the state (in the case of a classifier or acceptor) or of the state and input (in the case of a transducer).

Example: Inclusion of Output for the Combination Lock Example

We see that the lock accepts only when in state f. Equating acceptance to an output of 1, we see that the lock produces a 1 output only when uvw = 1. Therefore, we need only add an AND-gate with inputs from all three flip-flops to get the acceptor output. The complete lock is shown below.

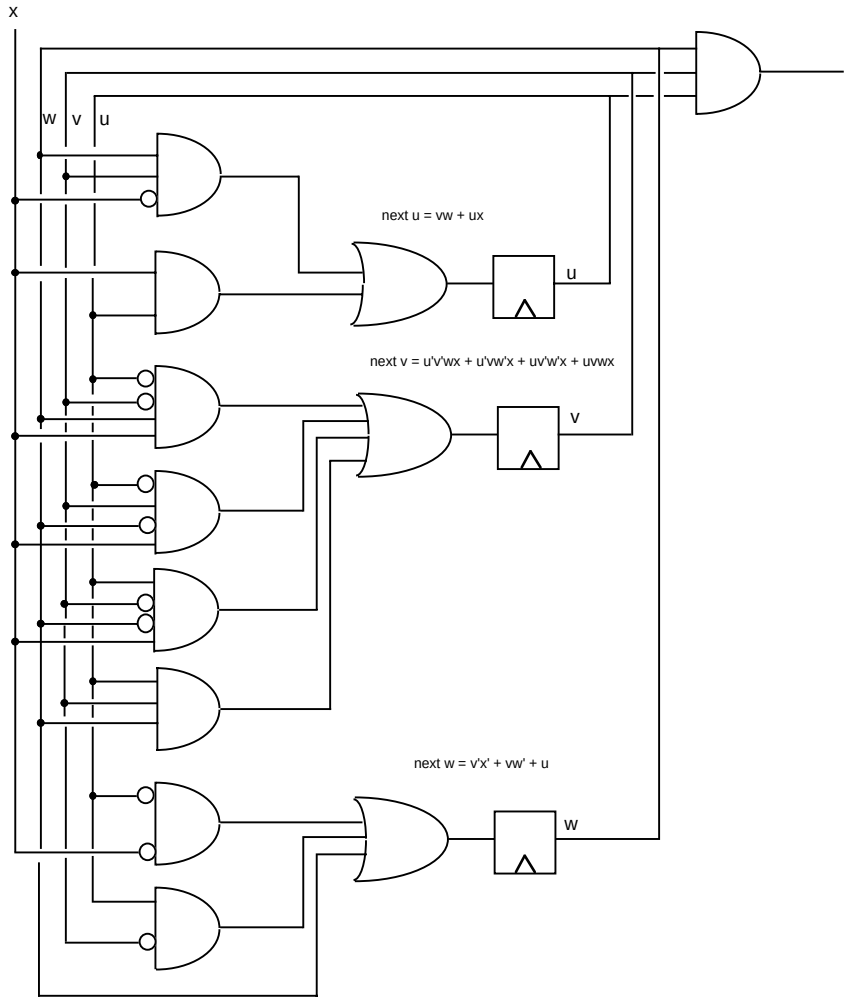


Figure 258: Implementation of a combination lock using flip-flops and gates

Example: Binary Adder with Output

The binary adder is an example of a transducer. The output value is 1 when the one or three of the two inputs and the carry are 1. When none or two of those values are 1, the output is 0. This functionality can be represented as a 3-input exclusive-OR gate, as shown in the figure, but this gate can also be further implemented using AND-, OR-, and NOT- gates as always. Typically the output would be used as input to a system in which this machine is embedded.

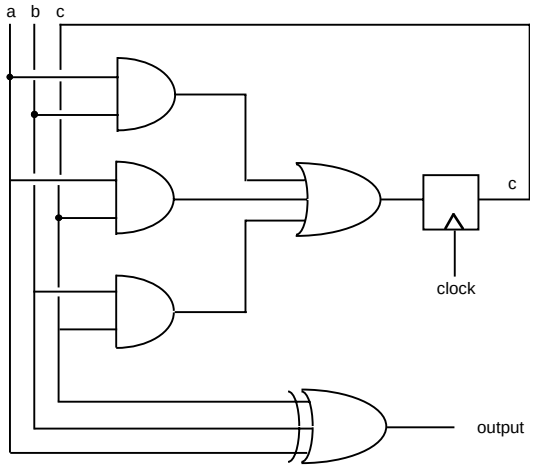


Figure 259: Implementation of a sequential binary adder

12.6 Inside Flip-Flops

In keeping with our desire to show a relatively complete vertical picture of computer structure, we briefly go into the construction of flip-flops themselves. Flip-flops can be constructed from combinational logic, assuming that such logic has a delay between input changes and output changes, as all physical devices do. Flip-flops can be constructed from a component that realizes the memory aspect, coupled with additional logic to handle clocking. The memory aspect alone is often referred to as a **latch** because it holds or "latches" the last value that was appropriately signaled to it.

To a first approximation, a latch can be constructed from two NOR gates, as shown below.

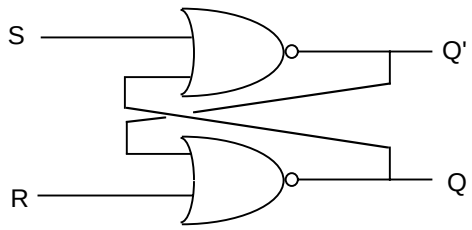


Figure 260: An electronic latch from opposing NOR-gates

In order for the latch to function properly, the inputs have to be controlled with a specific discipline; at this level, there is no clock to help us out. Let the state of the latch be represented by SRQQ'. Normally Q and Q' will be complementary, but there will be times at which they are not. Consider the state SRQQ' = 0010. Here we say the latch is "set". In the similar state 0001, the latch is "reset". The function of the inputs S and R is to put the latch into one of these two states. Specifically, when S is raised to 1, the latch should change to the set state, or stay in the set state if it was already there. Similarly, when R is raised to 1, the latch should change to the reset state. When the input that was raised is lowered again, the latch is supposed to stay in its current state.

We must first verify that the set and reset states are stable, i.e. not tending to change on their own. In 0010, the inputs to the top NOR gate are 01, making the output 0. This agrees with the value of Q' in 0010. Likewise, the inputs to the bottom NOR gate are 00, making the output 1. This agrees with the value of Q in 0010. Therefore 0010 is stable. Similarly, we can see that 0001 is also stable.

Now consider what happens if the latch is in 0010 (set) and R is raised. We then have state 0110. The upper NOR gate's output does not tend to change at this point. However, the lower NOR gate's output is driven toward 0, i.e. Q changes from 1 to 0. Following this, the upper NOR gate's output is driven toward 1, so Q' changes from 0 to 1. Now the latch is in state 0101. We can see this is stable. If R is now lowered, we have state 0001, which was already observed to be stable. In summary, raising R for sufficiently long, then lowering it, results in the reset state. Also, if the latch were in state 0001 when R is raised, then no change would take place and the latch would stay in state 0001 when R is lowered.

Similarly, we can see that raising S momentarily changes the latch to state 0010. So S and R are identified with the functions of setting and resetting the latch, respectively. Thus the latch is called a set-reset or SR latch. The following state diagram summarizes the behavior we have discussed, with stable states being circled and transient states left uncircled.

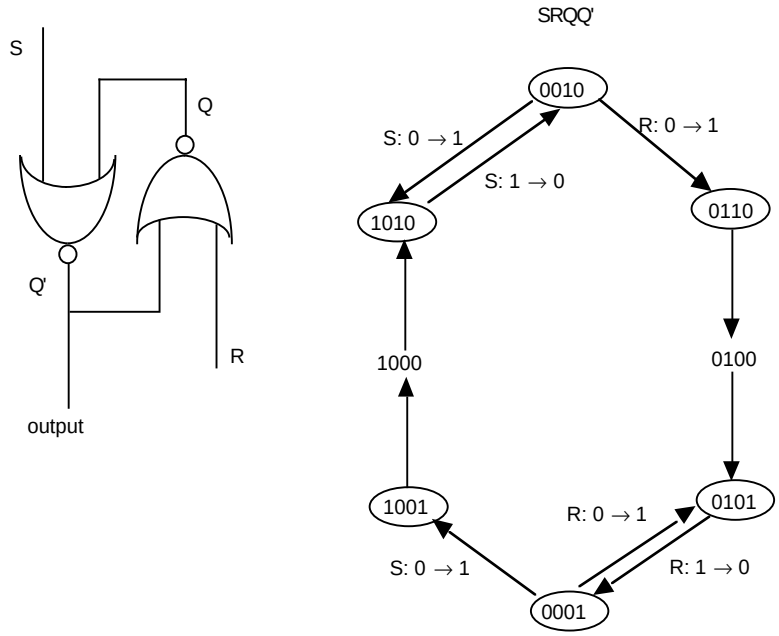


Figure 261: State-transition behavior of the electronic latch. States not outlined (1000, 0100) are unstable and tend to make autonomous transitions toward stable states as shown.

In describing the latch behavior, we dealt with the cases where only one of R or S is changing at a time. This constitutes a constraint under which the latch is assumed to operate. If this condition is not maintained, then the latch will not necessarily behave in a predictable fashion. We invite you to explore this in the exercises.

Next we show how a latch becomes a flip-flop by adding the clock element. To a first approximation, a flip-flop is a latch with some added gates so that one of S or R is only activated when the clock is raised. This approximation is shown below. However, we do not yet have a true flip-flop, but only a **clocked latch**, also called a **transparent latch**. The reason for this hedging is that if the input to the unit is changed while the clock is high, the latch will change. In contrast, in our assumptions about a D flip-flop, the flip-flop is supposed to only change depending on the value of the input at the leading edge of the clock, i.e. the flip-flop is supposed to be **edge-triggered**.

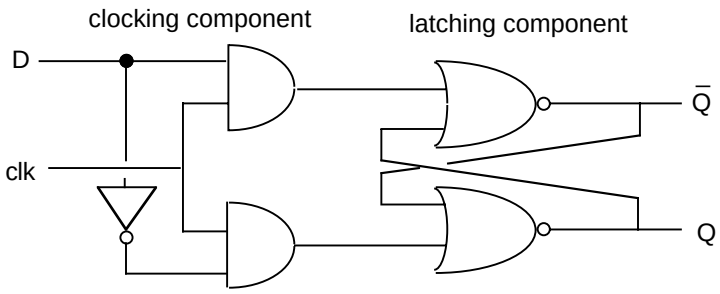


Figure 262: A "transparent" D latch

In order to get edge triggering, we need to latch the input at the time the clock rises, and then desensitize the latch to further changes while the clock is high. This is typically done using a circuit with a more complex clocking component, such as the one below. The assumption made here is that the D input is held constant long enough during the clock high for the flip-flops on the left-hand side to stabilize.

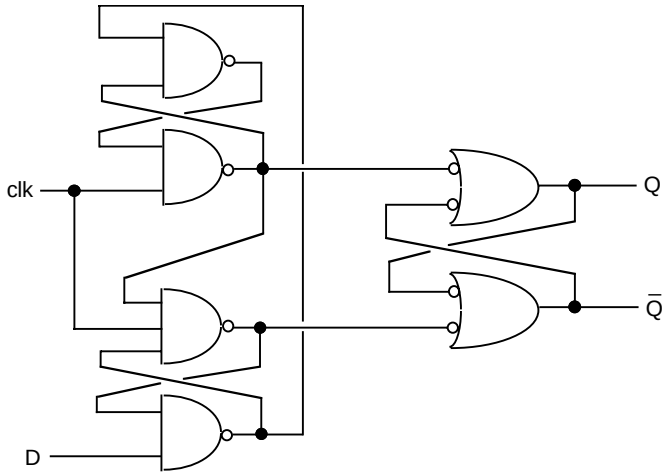


Figure 263: Edge-triggered D flip-flop using NAND gates

Exercises

1 • Explain in your own words why raising and lowering S changes the latch to 0010.

2 •• Explore the state transition behavior for states not shown in the diagram, in particular from the state 1100.

3 ••• By using a state diagram, verify that the edge-triggered D flip-flop is indeed edge-triggered.

12.7 The Progression to Computers

We have already seen hints of the relationship of finite-state machines to computers. For example, the control unit of a Turing machine looks very much like a finite-state transducer. In these notes, we use the "classifier" variety of finite-state machine to act as controllers for computers, introducing a type of design known as register-transfer level (RTL). From there, it is easy to explain the workings of a stored-program computer, which is the primary medium on which programs are commonly run.

Representing states of computers explicitly, as is called for in state diagrams and tables, yields state sets that are too large to be practically manageable. Not only is the number of states too big to fit in the memory of a computer that would be usable as a tool for analyzing such finite-state machines, it is also impossible for a human to understand the workings of a machine based on explicit representation of its states. We therefore turn to methods that combine finite-state machines with data operations of a higher level.

A Larger Sequential Adder

A machine that adds up, modulo 16, a sequence of numbers in the range 0 to 15, each represented in binary. Such a machine is depicted below.

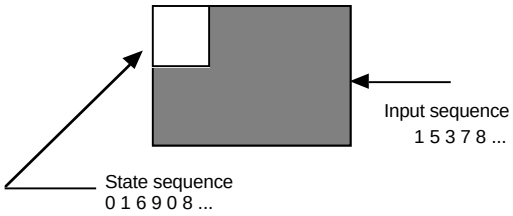


Figure 264: Sequence adding machine, modulo 16

We could show the state transition function for this machine explicitly. It would look like the following large addition table:

next state	input															
		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
current state	0	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	1	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	0
	2	2	3	4	5	6	7	8	9	10	11	12	13	14	15	0	1
	3	3	4	5	6	7	8	9	10	11	12	13	14	15	0	1	2
	4	4	5	6	7	8	9	10	11	12	13	14	15	0	1	2	3
	5	5	6	7	8	9	10	11	12	13	14	15	0	1	2	3	4
	6	6	7	8	9	10	11	12	13	14	15	0	1	2	3	4	5
	7	7	8	9	10	11	12	13	14	15	0	1	2	3	4	5	6
	8	8	9	10	11	12	13	14	15	0	1	2	3	4	5	6	7
	9	9	10	11	12	13	14	15	0	1	2	3	4	5	6	7	8
	10	10	11	12	13	14	15	0	1	2	3	4	5	6	7	8	9
	11	11	12	13	14	15	0	1	2	3	4	5	6	7	8	9	10
	12	12	13	14	15	0	1	2	3	4	5	6	7	8	9	10	11
	13	13	14	15	0	1	2	3	4	5	6	7	8	9	10	11	12
	14	14	15	0	1	2	3	4	5	6	7	8	9	10	11	12	13
	15	15	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

If we chose a much larger modulus than 16 the state table would be correspondingly larger, growing as the square of the modulus. However, the basic principle would remain the same. We can show this principle using a diagram like the finite-state machine diagram:

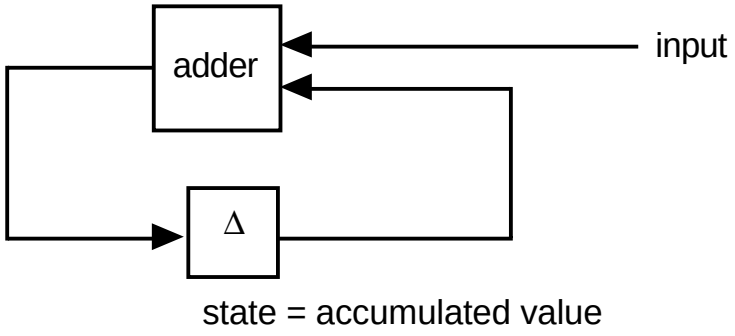


Figure 265: Diagram for the sequence adder.
The adder box adds two numbers together

As before, we can implement the adder in terms of combinational logic and the state in terms of a bank of D flip-flops. The combination logic in this is recognized as the adder module introduced in *Proposition Logic*.

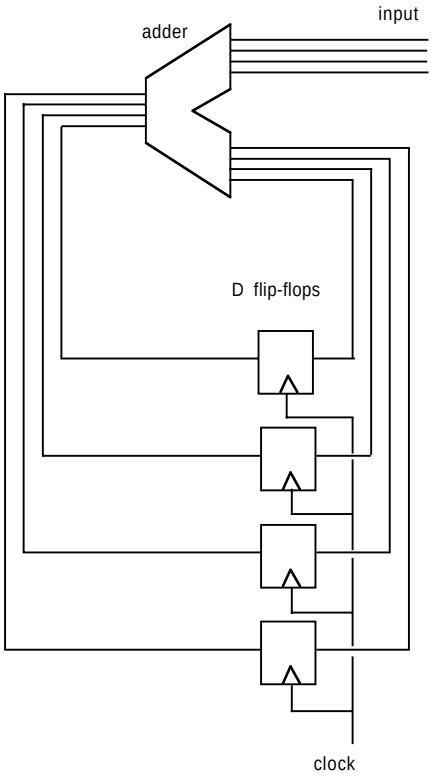


Figure 266: The sequence adder at the next level of detail

Registers

Now return to our diagram of the sequence adder. In computer design, it is common to group flip-flops holding an encoded value together and call them a **register**, as suggested by the following figure.

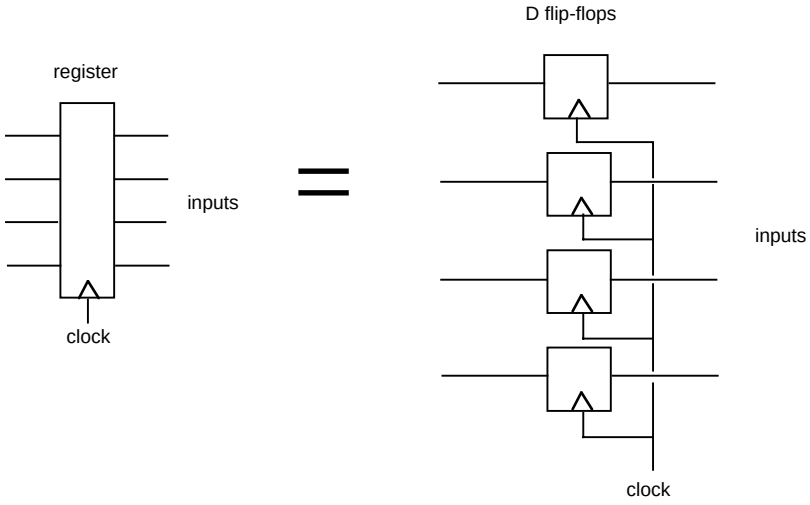


Figure 267: Expansion of a 4-bit register

Every time the clock ticks, the current logical values at the input side are stored into (or "gated into") the flip-flops. This shows a register of minimum functionality. Usually other functions are present. For example, we will often want to selectively gate information into the register. This can be accomplished by controlling whether or not the flip-flops "see" the clock tick. This can be done simply using an AND-gate. The control line is known as a "strobe": when the strobe is 1 and the clock ticks, the external values are gated into the register. When the strobe is 0, the register maintains its previous state.

Note that a register is essentially a (classifier) finite-state machine that just remembers its last data input. For example, suppose that we have a register constructed from two flip-flops. Then the state diagram for this machine is:

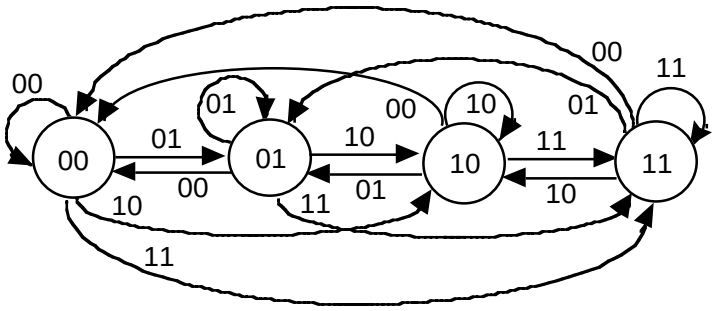


Figure 268: State-transition diagram for a 2-bit register.
Each input takes the machine to a state matching the input.

A typical use of this selective gating is in selective transfer from one register to another. The situation is shown below.

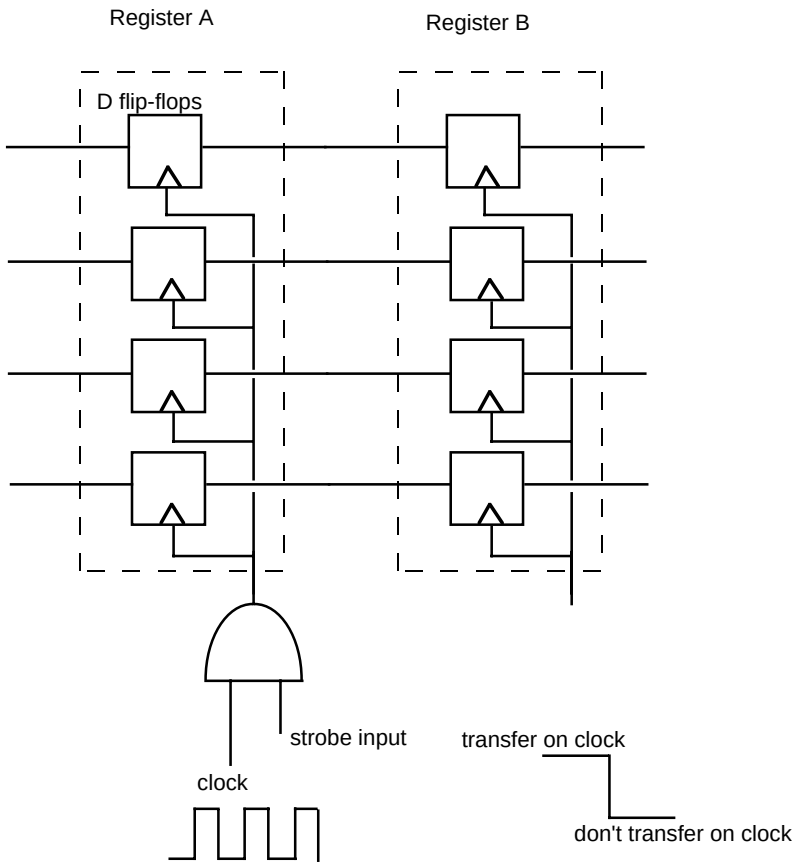


Figure 269: Transferring from one register to another using a strobe

In a similar fashion, the output of any combinational unit can be gated selectively into a register. Typically, the gate with the strobe is considered to be part of the register itself. In this case, the view of the register as a finite state-machine includes the strobe as one of the inputs. If the strobe value is 1, the transitions are as shown in the previous state diagram. If the strobe value is 0, the transitions are to the current state. For ultra-high-speed designs, strobing against the clock this way is not desirable, as it introduces an extra gate delay. It is possible to avoid this defect at the expense of a more complicated register design. We will not go into the details here.

Composition of Finite-State Machines

A computer processor, the kind you can buy, rather than an abstract computer like a Turing machine, is essentially a very large finite-state machine. In order to understand the behavior of such a machine, we must resort to a modular decomposition. We cannot hope to enumerate all the states of even a simple computer, even if we use all the resources in the universe.

There are several fundamental ways to compose finite-state machines. In each case, the overall machine has as its state set a subset of the Cartesian product of the state sets of the individual machines. Consider first the *parallel composition* of two machines, as shown below.

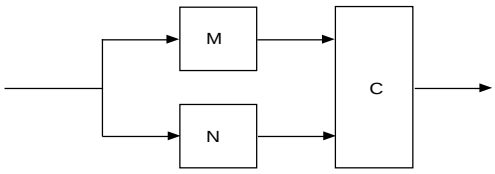


Figure 270: Parallel Composition of machines M and N

The two machines share a common input. They go through their transitions in "lock step" according to the clock. Unit C is combinational logic that combines the outputs produced by the machines but that does not have memory of its own.

To how the structure of this machine relates to the individual machines, let's show something interesting:

The intersection of two regular languages is regular.

Unlike the *union* of two regular languages, his statement does not follow directly from the definition of regular expressions. But we can show it using the parallel composition notion. Let M and N be the machines accepting the two languages in question. We will see how a parallel composition can be made to accept the intersection. What unit C does in this case is to form the logic product of the outputs of the machines, so that the overall machine accepts a string when, and only when, both component machines accept. This is, after all, the definition of intersection.

Example: Parallel Composition

Consider two acceptors used as examples earlier.

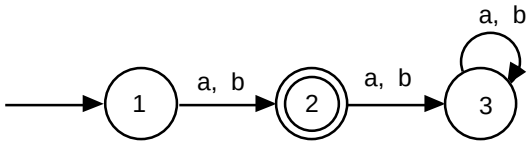


Figure 271: Acceptor for a | b

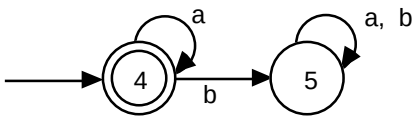


Figure 272: Acceptor for a*

To make an acceptor for the intersection of these two regular languages, we construct a machine that has as its state set the product {1, 2, 3} X {4, 5} = {(1, 4), (1, 5), (2, 4), (2, 5), (3, 4), (3, 5)}. We might not actually use all of these states, as some might not be reachable from the initial state, which is (1, 4) (the pair of initial states of each machine). There is just one accepting state, (2, 4), since it is the only one in which *both* components are accepting.

The following transitions occur in the product machine:

	input	
state	a	b
(1, 4)	(2, 4)	(2, 5)
(2, 4)	(3, 4)	(3, 5)
(2, 5)	(3, 5)	(3, 5)
(3, 4)	(3, 4)	(3, 5)
(3, 5)	(3, 5)	(3, 5)

We see that one state in the product, namely (1, 5), is not reachable. This is because once we leave 1, we can never return. The diagram for the product machine is thus shown in the following figure.

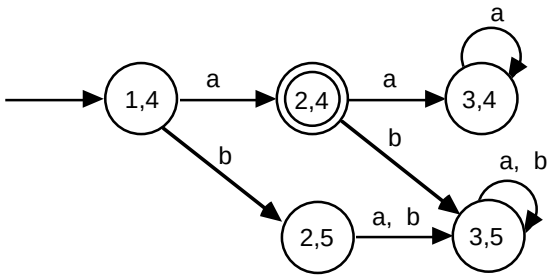


Figure 273: State diagram for the product of the preceding two acceptors

In a similar vein, we may construct other kinds of composition, such as ones in which one machine feeds the other, or in which there is cross-feeding or feedback. These are suggested by the following structural diagrams, but the experimentation with the state constructions is left to the reader.

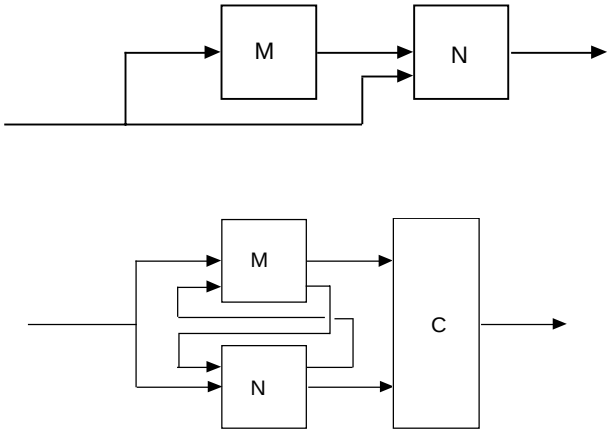


Figure 274: Examples of other machine compositions

Again, we claim that a computer is just one large composition of many smaller machines. A clear example of this will be seen in the next chapter.

Additional Capabilities of Registers

We saw in Part 1 how data can be transferred from one register to another as an array of bits in one clock tick. This is known as a **parallel transfer**. Another way to get data to or from a register is via a **serial transfer**, i.e. one bit at a time. Typically this is done by having the bits gated into one flip-flop and **shifting** the bits from one flip-flop to the next. A register with this capability is called a **shift register**. The following diagram shows how the shift register functionality can be implemented.

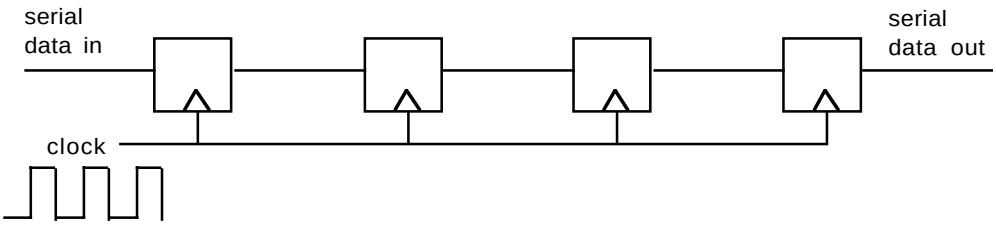


Figure 275: Shift register constructed from D flip-flops

Once the data have been shifted in serially, they can be transferred out in parallel. Also, data could be transferred in parallel and transferred out in serial. Thus the shift register can serve as a serial-parallel converter in both directions.

A shift register is an implementation of finite-state machine, in the same model discussed earlier, i.e. a bank of flip-flops serve as memory and a set of combinational functions compute the next-state function. The functions in this case are trivial: they just copy the

value from one flip-flop to the next. For the 4-flip-flop machine shown above, the transition table would be:

next state		**input**	
	current state	**0**	**1**
	0 0 0 _	0 0 0 0	1 0 0 0
	0 0 1 _	0 0 0 1	1 0 0 1
	0 1 0 _	0 0 1 0	1 0 1 0
	0 1 1 _	0 0 1 1	1 0 1 1
	1 0 0 _	0 1 0 0	1 1 0 0
	1 0 1 _	0 1 0 1	1 1 0 1
	1 1 0 _	0 1 1 0	1 1 1 0
	1 1 1 _	0 1 1 1	1 1 1 1

Here the _ indicates that we have the same next state whether the _ is a 0 or 1. This is because the right end bit gets "shifted off".

In order to combine functionalities of shifting and parallel input to a register, additional combinational logic has to be used so that each flip-flop's input can select either function. This can accomplished by the simple combinational circuit known as a **multiplexor**, as introduced in the Proposition Logic chapter. By using such multiplexors to select between a parallel input and the adjacent flip-flop's output, we can achieve a two-function register. The address line of each multiplexor is tied to a control line (which could be called a "strobe") that specifies the function of the register at a given clock.

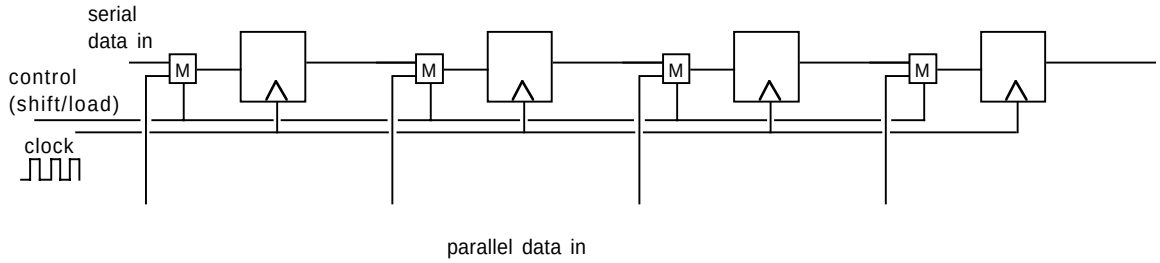


Figure 276: Structure of a shift register with two functions: serial data in and parallel data in

A commercial shift-register is typically constructed from a different type of flip-flop than the D, to simplify the attendant multiplexing logic. Thus the above diagram should be regarded as being for conceptual purposes. Multiplexors (also called **MUX**es) are more often found used in other applications, as will be discussed subsequently.

Buses and Multiplexing

Quite often in computer design we have the need to selectively transfer into a register from one of several other registers. One way to accomplish this is to put a multiplexor with one input per register from which we wish to transfer with the output connected to the register to which we wish to transfer. The address lines can then select one of the input registers. As before, the actual transfer takes place on a clock tick.

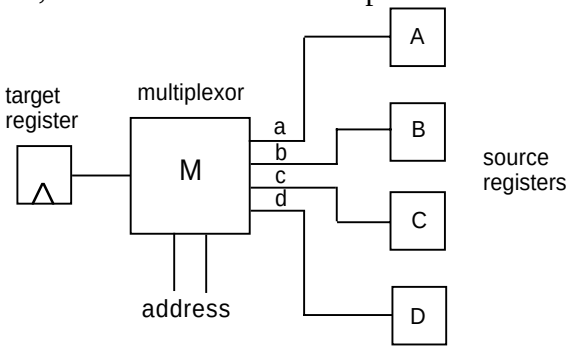


Figure 277: Selective transfer using a multiplexor (one bit shown)

This approach can be both expensive and slow (due to multiple levels of delay in the multiplexor) if the number of source registers is appreciable. An alternate approach is to use a **bus structure** to achieve the multiplexing. To a first approximation, a bus just consists of a single wire per bit. The understanding is that at most one source register will be providing input to the bus at any one time. Hence the bus at that time provides a direct connection to the target register. The problem is how to achieve this effect. We cannot use logic gates to connect the sources to the bus, since by definition these gates always have their output at either 0 or 1. If one gate output is 0 and the other is 1 and they are connected to a common bus, the result will be undefined logically (but the effect is similar to a short-circuit and could cause damage to the components).

A rather miraculous device known as a **three-state buffer** is used to achieve the bus interconnection. As its name suggests, its output can be either 0, 1, or a special third state known as "high impedance" or "unconnected". The effect of the third state is that the output line of the 3-state buffer is effectively not connected to the bus logically. Nonetheless, which of the three states the device is in is controlled electronically by two logical inputs: one input determines whether the output is in the unconnected state or not, and the other input has the value that is transmitted to the output when the device is not in the connected state. Therefore the following function table represents the behavior of the 3-state buffer:

control	data	output
0	0	unconnected
0	1	unconnected
1	0	0
1	1	1

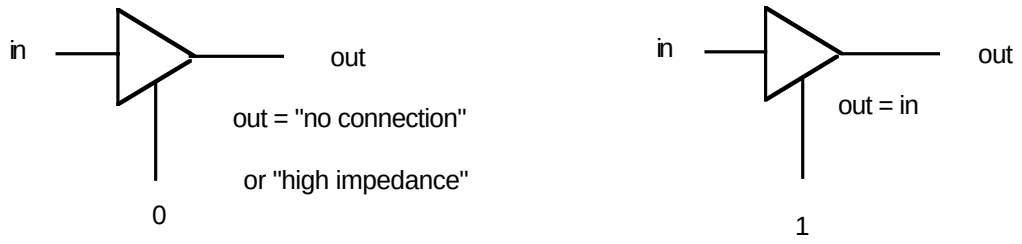


Figure 278: Three-state buffer behavior

The following figure illustrates how multiple source registers can be multiplexed using a bus and one 3-state buffer per source. Notice that the selection is done by one 3-state device being activated at a time, i.e. a "one-hot" encoding, in contrast to the binary encoding we used with a multiplexor. Often we have the control strobe in the form of a one-hot encoding anyway, but if not, we could always use a decoder to achieve this effect. Of course, if there are multiple bits in the register, we must have multiple bus wires and one 3-state buffer per bit per register.

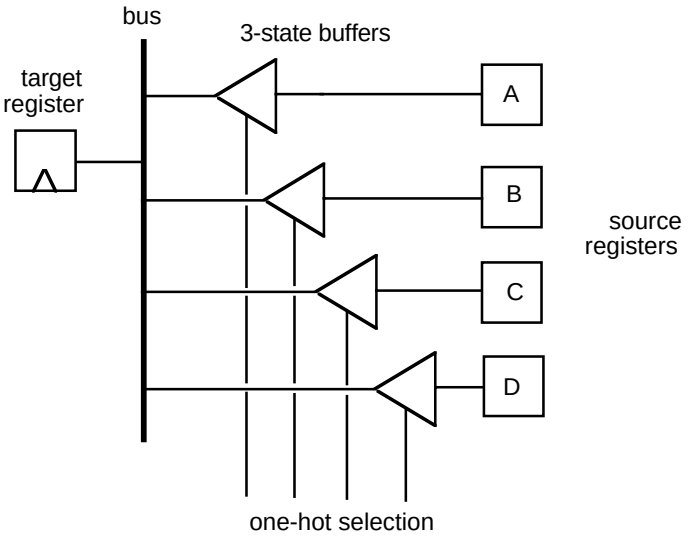


Figure 279: Multiplexing register sources using a bus and 3-state buffers

If there are multiple targets as well as sources, then we can control the inputs to the targets by selectively enabling the clock input to those registers. One contrast to the multiple-source case, however, is that we can "broadcast" the same input to multiple targets in a single clock tick. Put another way, the selection of the target register(s) can be done by a subset encoding rather than a one-hot encoding. This is shown in the next figure.

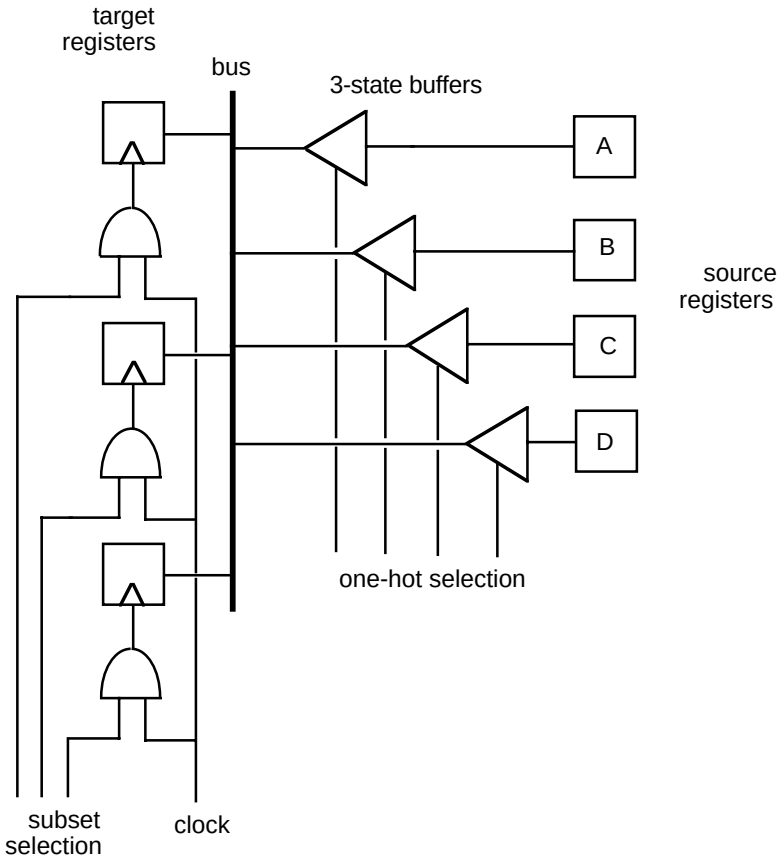


Figure 280: Multiple source and target selections on a bus

The other point to note about a bus is that inputs need not come from a register; they can come from the output of any combinational logic, such as an adder.

At a sufficiently coarse level of abstraction, buses are shown as a single data path, suppressing the details of 3-state devices, selection gates, etc. An example is the diagram of the ISC (Incredibly Simple Computer) in the next chapter.

Exercises

1 •• Show how to construct a 2-function shift-register that has the functions shift-left and shift-right.

2 ••• Show how to construct a 5-function shift-register, with functions shift-left, shift-right, parallel load, clear (set all bits to 0), and no-operation (all bits left as is).

3 ••• While the shift register described is convenient for converting serial data to parallel data, it is too slow to serve as an implementation of the shift instructions found in most computers, where it is desired to shift any number of bits in one clock interval. A combinational device can be designed that accomplishes this type of shift based on using the binary representation of the amount to be shifted

to shift in log N stages, shifting successive powers of two positions at each stage. Sketch the design of such a device based on 2-input multiplexors. In the trade, this is known as a "barrel shifter". Earlier, we referred to the principle on which the barrel shifter is based as **the radix principle**.

12.8 Chapter Review

Define the following terms:

acceptor
bus
classifier
clock
D flip-flop
edge-triggered
feedback system
finite-state machine
flip-flop
Kleene's theorem
latch
multiplexor
non-deterministic finite-state machine
parallel composition
quantization
regular expression
register
sequencer
shift register
stable state
synchronous
three-state buffer
transducer
Turing machine

Demonstrate how to convert a non-deterministic finite-state acceptor to a deterministic one.

Demonstrate how to derive a non-deterministic finite-state acceptor from a regular expression.

Demonstrate how to derive a regular expression from a finite-state machine.

Demonstrate how to synthesize a switching circuit from a finite-state machine specification.

12.9 Further Reading

Frederick C. Hennie, *Finite-State Models for Logical Machines*, Wiley, New York, 1968. [Further examples of finite-state machines and regular expressions. Moderate.]

S.C. Kleene, Representation of events in nerve nets and finite automata, pp 3-41 in Shannon and McCarthy (eds.), *Automata Studies*, Annals of Mathematics Studies, Number 34, Princeton University Press, 1956. [The original presentation of regular expressions and their connection with machines. Moderate.]

G.H. Mealy, *A method for synthesizing sequential circuits*, The Bell System Technical Journal, 34, 5, pp. 1045-1079, September 1955. [Introduces the Mealy model of finite-state machine. Moderate.]

Edward F. Moore, *Gedanken-experiments on sequential machines*, pp 129-153 in Shannon and McCarthy (eds.), *Automata Studies*, Annals of Mathematics Studies, Number 34, Princeton University Press, 1956. [Introduces the Moore model of finite-state machine. Moderate to difficult.]