Figure 1.1: A shift register

Figure 1.2: A binary adder

1.1 Finite-state machines

Clocked circuits. The most obvious element of ordinary computations missing from Boolean circuits is *repetition*. Repetition requires *timing* of the work of computing elements and *storage* of the computation results between consecutive steps. Let us look at the drawings of the circuit designer again. We will see components with one ingoing edge, of the form like in Figure 1.1, called **shift registers**. The shift registers are controlled by one central **clock** (invisible on the drawing). At each clock pulse, the assignment value on their incoming edge jumps onto their outgoing edges and becomes the value "in" the register.

A **clocked circuit** is a directed graph with numbered nodes where each node is either a Boolean component, a shift register, or an input or output node. There is no directed cycle going through only Boolean components.

How to use a clocked circuit for computation? We write some initial values into the shift registers and input edges, and propagate the assignment using the Boolean components, for the given clock cycle. Now we send a clock pulse to the register, and write new values to the input edges. After this, the new assignment is computed, etc.

(1.1) **Example** An adder. This circuit (see Figure 1.2) computes the sum of two binary numbers x, y. We feed the digits of x and y beginning with the lowest-order ones, at the input nodes. The digits of the sum come out on the output edge. $\diamondsuit$

How to compute a *function* with the help of such a circuit? Here is a possibility. We enter the input, either parallelly or serially (in the latter case, we signal at extra input nodes when the first and last digits of the input are entered). Now we run the circuit, until it signals at an extra output node when the output can be received from the other output nodes. After that, we read out the output, again either parallelly or serially (in the latter case, one more output node is needed to signal the end).

In the adder example above, the input and output are processed serially, but for simplicity, we omitted the extra circuitry for signaling the beginning and the end of the input and output.

Finite-state machines. The clocked circuit's future responses to inputs depend only on the inputs and the present contents of its registers. Let $s_1, \ldots, s_k$ be this content. To a circuit with n binary inputs and m binary outputs let us denote the inputs at clock cycle t by $x^t = (x_1^t, \ldots, x_n^t)$, the state by $s^t = (s_1^t, \ldots, s_k^t)$, and the output by $y^t = y_1^t, \ldots, y_k^t)$. Then to each such circuit, there are two functions

$$\begin{aligned} \lambda &: \{0,1\}^k \times \{0,1\}^n \to \{0,1\}^m, \\ \delta &: \{0,1\}^k \times \{0,1\}^n \to \{0,1\}^k \end{aligned}$$

such that the behavior of the circuit is described by the equations

$$\begin{aligned} y^t &= \lambda(s^t, x^t), \\ s^{t+1} &= \delta(s^t, x^t). \end{aligned} \tag{1.2}$$

From the point of view of the user, only these equations matter, since they describe completely, what outputs to expect from what inputs. A device described by equations like (1.2) is called a **finite-state machine**, or **finite automaton**. The finiteness refers to the finite number of possible values of the state s^t. This number can be very large: in our case, 2^k.

(1.3) **Example** For the binary adder, let u^t and v^t be the two input bits at time t, let c^t be the content of the carry, and w^t be the output at time t. Then the equations (1.2) now have the form

$$\begin{aligned} w^t &= u^t \oplus v^t \oplus c^t, \\ c^{t+1} &= \mathrm{Maj}(u^t, v^t, c^t). \end{aligned}$$

$\Diamond$

Not only every clocked circuit is a finite automaton, but every finite automaton can be implemented by a clocked circuit.

(1.4) **Theorem** *Let*

$$\begin{aligned} \lambda &: \{0,1\}^k \times \{0,1\}^n \to \{0,1\}^m, \\ \delta &: \{0,1\}^k \times \{0,1\}^n \to \{0,1\}^k \end{aligned}$$

be functions. Then there is a clocked circuit with input $x^t = (x_1^t, \ldots, x_n^t)$, *state* $s^t = (s_1^t, \ldots, s_k^t)$, *output* $y^t = y_1^t, \ldots, y_k^t)$ *whose behavior is described by the equations* (1.2),

Figure 1.3: Circuit and state-transition diagram of a memory cell

Proof The circuit has k shift registers, which at time t will contain the string s^t, and n input nodes, where at time t the input x^t is entered. It contains two Boolean circuits. The input nodes of both of these Boolean circuits are the shift registers and the input nodes of the whole clocked circuit. The output nodes of the first Boolean circuit B are the output nodes of the clocked circuit. We choose this circuit to implement the function $\lambda(s, x)$. The output nodes of the other circuit C are the inputs of the shift registers. This Boolean circuit is chosen to implement the function $\delta(s, x)$. Now it is easy to check that the clocked circuit thus constructed obeys the equations (1.2). ■

Traditionally, the state-transition functions of a finite-state machine are often illustrated by a so-called **state-transition diagram**. In this diagram, each state is represented by a node. For each possible input value x, from each possible state node s, there is a directed edge from node s to node $\delta(s, x)$, which is marked by the pair $(x, \lambda(s, x))$.

(1.5) **Example** A memory cell (flip-flop, see Figure 1.3). This circuit has just one register with value s which is also the output, but it has two input lines x and c. Line x is the information, and line c is the control. The equations are

$$\lambda(s, x, c) = \delta(s, x, c) = c \wedge x \vee \neg c \wedge s.$$

Thus, as long as $c = 0$ the state s does not change. If $c = 1$ then s is changed to the input x. The figure shows the circuit and the state transition diagram. ♢

1.2 A realistic finite computer

Clocked circuits contain all essential elements needed to describe the work of today's computers symbolically.

In the fip-flop example, we have seen how to build memory elements. Combining the memory elements with a selection mechanism using a decoder, we can build a **random-access memory**, storing e.g. 2^{16} (64K) words with 16 bits each.

We can consider as the program in the broader sense, everything found in the memory at the beginning of the computation. The result of the

computation can now be represented as $f_p(x)$, where p is the program and x is the input. Different choices of p define different functions f_p. Of course, a finite machine will be able to compute only a finite number of functions, and even these only on inputs x of a limited size.

There is a format for programs and programmable machines that has proved useful. There are some *operations*, i.e. small functions, used especially often. (E.g. multiplication, comparison, memory fetch and store, etc.) We build small circuits for all of these operations and arrange the circuits in an *operation table*, accessed by a decoder, just as a random- access memory. Most operations used in actual computers can be easily simulated using just 3-4 basic ones.

An **instruction** consists of the **address** of an operation in the operation table, and up to three **parameters**: arguments of the operation. A **program** is a series of instructions. It is stored in the random access memory.

A special circuit, the **instruction-interpreter**, stores a number called the **instruction pointer** that contains the address of the instruction currently executed. The interpreter goes through the following cycle:

1. It fetches the current instruction using the instruction-pointer.

2. It directs control to the circuit of the operation whose address is the first word of the instruction.

3. Upon return of the control, it adds 1 to the value of the instruction pointer.

Each operation returns control to the instruction-interpreter, but before that, it may change the value of the instruction-pointer ("goto instructions").

Just as in the RAM, it is enough to use two basic kinds of operation: **arithmetical assignment** and **flow control** (the latter involves a change in the instruction pointer). We can assign some special locations for input, for output, and for the instruction pointer. If the memory location involved in the assignment is input or output then the assignment involves an input- or output operation.