Chapter V
Binary logic circuits

Computer systems are built from binary logic circuits: circuits in which each signal has only two allowed values, 0 and 1, represented by two different voltages on a wire, measured relative to a reference “ground” voltage. It is possible to build simple computers using analogue circuits, where the voltage on a wire is directly proportional to a continuous value to be computed with. A major problem with that approach is that all wires in an electronic circuit pick up rapidly varying *noise*, which alters the voltage on the wire. In an analogue circuit, that changes the value represented, causing errors in the calculation. In a digital circuit, as long as the noise voltage is small compared to the difference between the voltages which represent 0 and 1, the intended 0 or 1 value on the wire can still be distinguished without error.

There are two kinds of binary logic circuits: *combinational* and *sequential*.

V.1 Combinational logic circuits

A general logic circuit has a number of inputs and a number of outputs, each of which is a binary logic signal. In a combinational logic circuit, the values of the outputs depend only on the current values of the inputs. The circuit has no memory, and the output does not depend on the history of the inputs.

The simplest combinational logic circuit has one input and one output, and the only interesting circuit of this type is the *inverter* (sometimes called a “not gate”), whose output is the inverse of its input. When the input is 0, the output is 1, and vice versa.

Two-input/one-output circuits are a little more interesting. These include the 2-input AND gate, whose output is 1 only if both inputs are 1, and the OR gate, whose output is 1 if either or both inputs are 1. The output of a NAND gate is the inverse of that of an AND gate, and similarly a NOR gate produces the inverse of an OR gate. All these gates can be made with more than two inputs; for example a 4-input NOR gate has a single output, which is 1 if all the inputs are 0, and 0 otherwise.

NAND and NOR gates are called *universal gates* because they can realise any logic operation when connected appropriately. For example, to create an inverter from a NAND gate, one only needs to set the unused input(s) to logic 1.

A useful notation for describing, and reasoning about, combinational logic circuits is the notation of Boolean algebra. Boolean algebra deals with variables that have two possible values, false and true (or 0 and 1). The Boolean notation for an AND function of two values a and b is $a.b$ (alternatively $a \wedge b$), and for the OR of a and b, the notation is $a + b$ (alternatively $a \vee b$). The inverse of a value is represented by placing a bar over the value's notation, for example $\overline{a}$. A 3-input NAND gate with inputs a, b and c, and output z can be described as $z = \overline{a.b.c}$

Multiplexer

One of the simplest combinational circuit used in a processor is the two-input multiplexer; it selects between two 1-bit inputs, steering one of them to its 1-bit output. A one-bit control input determines which 1-bit value is selected.

Truth tables

A general way to specify the function of a combinational logic circuit is using a *truth table*, which lists the value of the output(s) for all possible values of the inputs. The truth table for a n-input circuit will have 2^n rows.

The truth table for a two-input multiplexer is as follows. i_0 and i_1 are the two data inputs, c is the control input, and z is the output. The output takes the value of i_0 if the control input c is 0, and takes the value of i_1 if c is 1.

i_0	i_1	c	z
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	1

By looking at the rows in the truth table where the output is a 1, it is possible immediately to write down a Boolean expression for the output of the circuit:

$$z = \overline{i_0}.i_1.c + i_0.\overline{i_1}.\overline{c} + i_0.i_1.\overline{c} + i_0.i_1.c$$

Each of the four *AND terms* in this expression (e.g. $\overline{i_0}.i_1.c$) has the value 1 only for inputs corresponding to a single line in the truth table. There are four AND terms, one for each line in the truth table where the output is 1, and so the full output is obtained by ORing these terms together. This expression gives us a circuit for the multiplexer, consisting of three inverters to generate $\overline{i_0}$, $\overline{i_1}$ and $\overline{c}$, four three-input AND gates to generate the AND terms, and a four-input OR gate to OR together the outputs of the AND gates.

However, the Boolean expression above can be simplified by noting that $\overline{i_0}.i_1.c+i_0.i_1.c = (\overline{i_0}+i_0).i_1.c = i_1.c$ and $i_0.\overline{i_1}.\overline{c}+i_0.i_1.\overline{c} = i_0.(\overline{i_1}+i_1).\overline{c} = i_0.\overline{c}$, so $z = i_1.c + i_0.\overline{c}$. This is as expected, since if c is 0, z is only 1 if i_0 is 1, while if c is 1, z is 1 if i_1 is 1. This simpler Boolean expression represents a circuit consisting of one inverter, two 2-input AND gates, and a 2-input OR gate.

Both of these Boolean expressions for the multiplexer circuit are in "sum of products" form; that is, they consist of a number of AND terms ORed together. The simpler expression is the simplest possible sum of products expression for this particular truth table, and represents a good circuit for this function. Sum of products circuits are suitable for many combinational logic functions. They have the advantage that they can be generated directly and automatically from any truth table, and then simplified to the minimum number of, and simplest possible, AND terms. There are logic simplification algorithms which guarantee an optimal sum of products simplification, and logic design software is available to do this. Unfortunately, the unsimplified sum of products expression usually has of the order of 2^n AND terms (where n is the number of inputs to the function), and some functions do not simplify well. In those cases, a sum of products circuit uses far too many gates, and a more complex and more inspired design must be used instead.

V.1.1 Arithmetic circuits

Since numbers can be represented in binary and we have circuits available which operate on binary values, we can build circuits which perform arithmetic operations. An adder is a very common building block in processors, so we will examine how one can be designed.

Unfortunately, the adder is an example of a circuit for which a sum of products design is not suitable. A 32-bit adder has 64 inputs and 33 outputs. The least significant output bit depends only on the least significant bit of the two input numbers, but the most significant output bit depends on all 32 bits of each input number; it requires a logic circuit with 64 inputs.

A 32-bit adder circuit can be based on a circuit, called a *full adder*, which adds three one-bit values to produce a two-bit result consisting of a *sum* bit and a *carry* bit. The truth table is as follows:

a	b	c	$carry$	sum
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Using sum of products design we get the following expressions for the outputs:

$$sum = \overline{a}.\overline{b}.c + \overline{a}.b.\overline{c} + a.\overline{b}.\overline{c} + a.b.c$$

which cannot be simplified, and $carry = \overline{a}.b.c + a.\overline{b}.c + a.b.\overline{c} + a.b.c$, which simplifies to:

$$carry = b.c + a.c + a.b$$

A 32-bit adder can now be made by chaining together 32 full adders, with the *carry* output of each connected to the *c* input of the next (the c_0 input is usually just fed with 0):

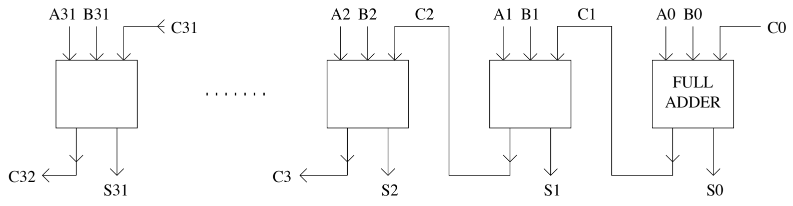

Figure V.1: Ripple carry adder.

V.1.2 Propagation delays

Real gates do not operate instantaneously. There is a *propagation delay* after an input changes, before the new input is reflected in the output value. The propagation delay of a gate depends on factors such as the design of the transistors in the gate, and on the number of subsequent gate inputs fed by the gate's output. The propagation delay for a simple gate is less than 100ps (1ps = 1 picosecond = 10^{-12}s) in current manufacturing technologies.

Longest path determines the speed

Since everyone wants their computer to go as fast as possible, designing logic circuits to have minimum propagation delay is very important in computer design and the key to this is to minimise the number of gates an input change must propagate through to reach the output of a circuit.

In the above adder circuit, for example, the longest path from an input to an output is 65 gates long: from the least significant bits of the two input numbers, along the chain of carries, through an AND and an OR gate in each of 31 full adders, and then through an inverter, an AND and an OR gate in the most significant full adder, to generate the output bit s_{31}. The output of the 32-bit adder cannot be relied on to be correct until 65 gate delays after the two input numbers are presented.

This particular type of adder is known as a *ripple carry adder*, because of the way the carries ripple relatively slowly along the chain of full adders. If such a circuit was used in a real processor datapath, it would certainly be the limiting factor on the clock speed, and would slow the processor down significantly. In fact, adders can be made much faster by adding extra circuitry to speed up the rate at which the carries pass along the chain, but this is beyond the scope of this course.

V.2 Sequential Logic Circuits

In a sequential logic circuit, the outputs depend not just on the current values of the inputs, but also on past values of the inputs. The circuit has memory. Sequential circuits can do two things that combinational circuits cannot: they can recognise sequences of inputs and they can generate sequences of outputs.

A sequential logic circuit is made by adding feedback to a combinational logic circuit:

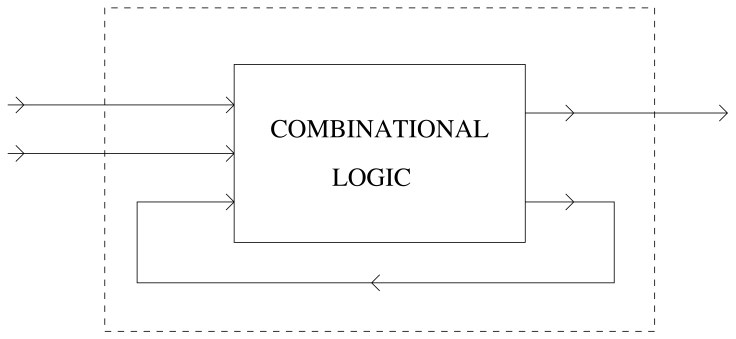

Figure V.2: Sequential logic.

In this circuit, depending on the design of the combinational logic, the value on the output will depend both on the two inputs from the outside world, and the on internal feedback signal. With one feedback signal, the circuit can have two behaviours for any particular values of the inputs from the outside world: one with the feedback signal = 0, and one with the feedback signal = 1. The circuit then has two states, with different behaviours. A circuit with two feedback signals can have up to four different states, and one with n feedback signals, up to 2^n states. The state of the feedback signals at any time in such a circuit will, in general, depend on the sequence of earlier inputs: this is how the circuit remembers past inputs.

SR-Latch

As a concrete example, consider the circuit in figure V.3, the *SR latch*, made of two 2-input NOR gates, which is the simplest interesting sequential circuit. When the two inputs S and R are both 0, the circuit has two states, with the output Q equal to either 0 or 1. Both states are stable (work out the logic level on each internal signal to see this). Which state the circuit is in when S = R = 0 depends on the values on S and R immediately before they were both 0. For example, if S is 1 and R is 0, the output Q must be 1. If S then changes to 0, the output Q remains 1. On the other hand, if S is 0 and R is 1, the output Q must be 0, and if R then changes to 0, the output Q remains 0 (again, working out the logic levels on each signal will confirm this).

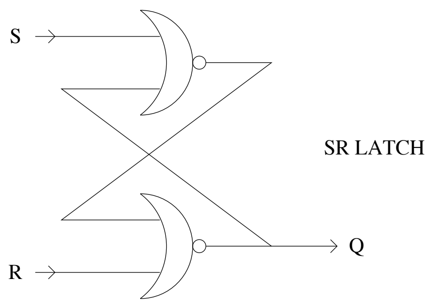

Figure V.3: The Set-Reset latch circuit.

The SR latch is used as a one-bit storage element. Most of the time, S and R are both kept at 0, and the latch is storing a bit, 1 or 0, which is available on Q. To set the stored bit to a 1, make a short 0→1→0 pulse on the S input. To reset the stored bit to a 0, make a short 0→1→0 pulse on the R input.

V.2.1 Synchronous sequential logic

The circuits considered so far are examples of *asynchronous* sequential logic circuits. In an asynchronous circuit, the state of the circuit, i.e. the value on the fed back signal(s), can change at any time in response to an input change. Asynchronous circuits are tricky to design, and so most sequential logic circuits are designed in a different way, as *synchronous* circuits. In a synchronous sequential logic circuit, changes of state are only allowed to happen at times synchronised to a special timing signal, called the *clock*, which is connected to all storage elements.

The simplest synchronous circuit is again a one-bit storage element, called a *D flip-flop*. The symbol for an *edge-triggered* D flip-flop is shown in figure V.4. In this flip-flop, the stored state Q only changes when the clock input makes a rising edge, i.e. a transition from 0 to 1. When that happens, Q takes on the value on the input D. At all other times, D is ignored and Q does not change.

We can now make a register used in a 32-bit processor simply by putting together 32 of these flip-flops, with the 32 clock inputs connected together to form a single register load control signal. The symbol for a four-bit register is shown in figure V.4. A new value is loaded into the register when the clock makes a rising edge.

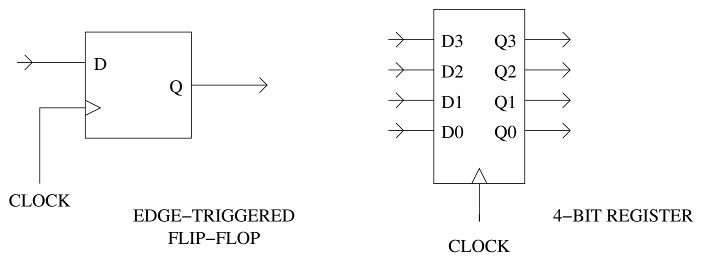

Figure V.4: D flip flop and 4-bit register.

A more general synchronous sequential logic circuit looks like this:

The feedback signals have a register placed in them, the *state register*. Changes on the inputs to the circuit change the *next state* signals, that feed the D inputs of the state register, but the actual state of the circuit, as fed back into the combinational logic from the Q outputs of the state register, will only change when the clock makes a rising edge. The usual way to use such a circuit it to have the clock pulsing regularly, so that the state updates at regular intervals, on each rising edge of the clock.

V.2.2 A hardware FSM

Inf1A, Computation and Logic describes how a (deterministic) Finite State Machine (FSM) takes in a sequence of input symbols, and determines whether the sequence is accepted or rejected. A sequential circuit is essentially an FSM. We will design the control logic of a simple vending machine

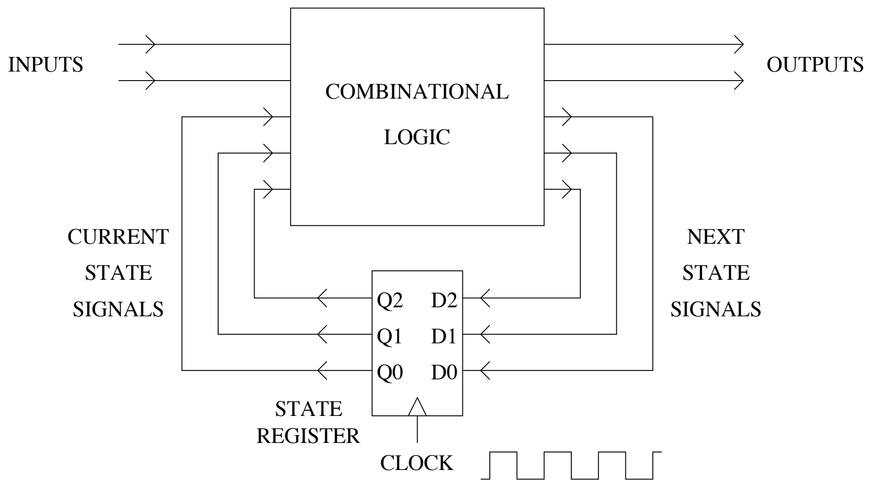

Figure V.5: Generic synchronous sequential circuit.

as an example.

Our vending machine accepts 10p and 20p coins (one at a time) and only sells one kind of chocolate bar which costs 30p. No change is given and only one chocolate can be bought at one time.

The machine has a sensor which detects what type of coin is paid in and has two outputs: $\mathtt{a}$ which is asserted when a 10p coin is detected and $\mathtt{b}$, which is asserted when a 20p coin is detected. One chocolate is released when the signal $\mathtt{z}$ is asserted.

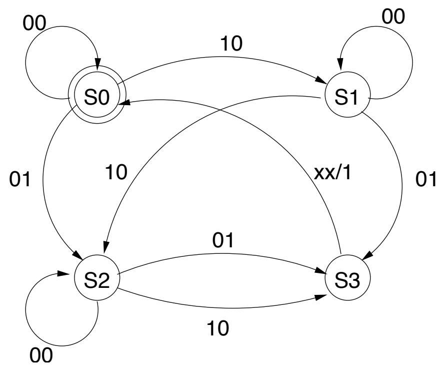

Figure V.6: Vending machine FSM.

Figure V.6 shows the state diagram of the machine. The initial state is S0, where no money has been paid into the machine. When the balance is

10p, the machine is in state S1. When the balance is 20p, the machine is in state S2. When the balance is 30p or more, the machine is state S3, asserts the output z to logic 1 and immediately returns to S0, to be ready for the next transaction.

Since there are four states, we need 2 bits to encode the current state: s_1, s_0. There are many ways to encode the states S0 – S3, but to keep things simple, we assume that S0 is represented as 00, S1 as 01,. . .

The sequential circuit implementing this FSM will look similar to the machine of figure V.5. The circuit is started in the state corresponding to the FSM start state, S0 (00) and the input sequence is presented on the inputs to the circuit, one per clock cycle. The circuit moves between states, one state transition per clock cycle. In this particular type of FSM, called *Moore* machine, the output signal depends only on the current state, not on the inputs.

In order to build the sequential circuit, we need to generate the boolean expressions for the output z and the next-state signals s'_1, s'_0. Here is the combined truth table:

Current state		10p	20p	Next state		
s_1	s_0	a	b	s'_1	s'_0	z
0	0	0	0	0	0	
0	0	0	1	1	0	0
0	0	1	0	0	1	
0	1	0	0	0	1	
0	1	0	1	1	1	0
0	1	1	0	1	0	
1	0	0	0	1	0	
1	0	0	1	1	1	0
1	0	1	0	1	1	
1	1	0	0	0	0	
1	1	0	1	0	0	1
1	1	1	0	0	0	

Note that the values for z are shown once per input state, because it depends only on the current state, not on the values of the inputs a, b. From the table we can extract the boolean expressions for all outputs and then build circuits as we did for the multiplexer.

The above procedure can be followed for generating a sequential circuit from a state diagram and is automated by computer design tools. When we will be looking at the control logic of the processor (the multicycle and pipelined versions), you should keep in mind that they are implemented by a form of FSM.