Lecture 11: Registers

Registers are used widely in computer systems for a variety of purposes such as the address register, the program counter &c. which you will meet (or have already met) in other parts of the course. The purpose of this lecture is to introduce you to the different ways registers can be constructed to achieve the functionality required in a computer.

We have already seen one example of a register, namely the bank of flip-flops that store the state of a synchronous circuit. A set of *n* flip flops can store any binary number in the range 0 to 2^{n-1}, or $-(2^{n-1}-1)$ to $+(2^{n-1}-1)$, depending on how we choose to represent that number. In the case of the state register that we have already seen, we are concerned with the simple functionality of loading and storing a binary number - the state - and so we need nothing more than a set of D-type flip flops with a common clock.

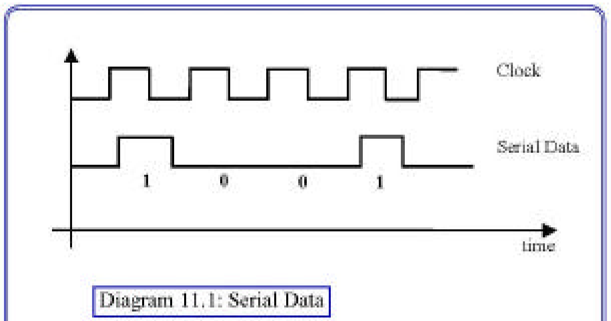

Diagram 11.1: Serial Data

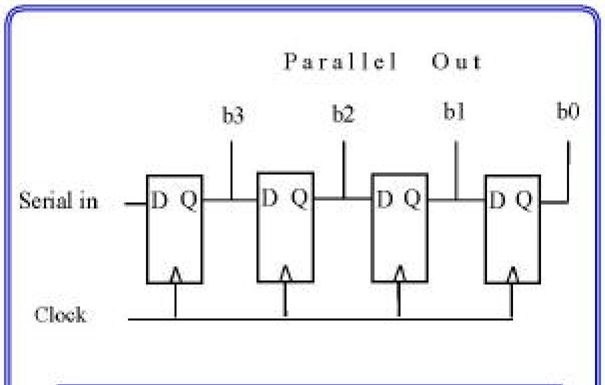

Diagram 11.2: Serial to Parallel Conversion

Frequently however, information in digital computers is in serial form. That means that, in contrast to the state register, where a number is read from the bits at one instant of time, the bits of a number are a function of time, and arrive one after the other on successive falling edges of a clock. The timing Diagram 11.1 shows a serial bit stream representing a binary number with four bits. In practice, most of the processing carried out inside a computer is done in parallel, to increase the speed, whereas most of the transfer of information between computers is carried out in serial to reduce the cost of the cables. (A serial signal can be carried on a cable containing a pair of wires, which is clearly going to be cheaper than a cable containing nine wires for transmitting an eight bit number). So one important task is to convert serial information into parallel form, and this is done using a shift register. Shift registers can be most simply be constructed by joining up a set of D-type flip flops as shown in Diagram 11.2. On each successive clock falling edge, the data bit (Q) on a flip flop is loaded onto the flip flop on its right. Thus successive serial bits arriving at the input travel through the register in four clock pulses.

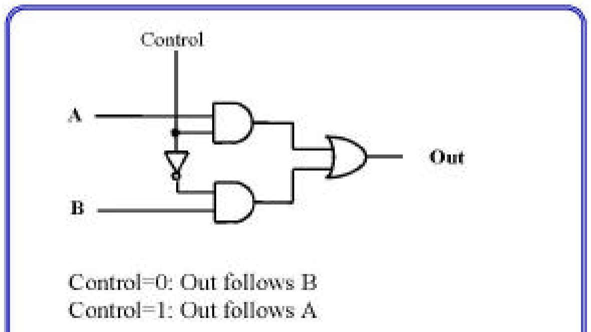

Diagram 11.3: Two input multiplexor

It is clear that parallel to serial conversion can be achieved in an analogous manner, provided that we can load the parallel data into the register. To do this we need to add some further control circuitry which determines the function of the register. In the simplest case we need to allow each flip flop to select its input either from the previous stage (for serial/parallel conversion) or from an input bit for parallel load. The circuit that achieves this selection is called a multiplexor, and is shown in Diagram 11.3. We can add this to our previous register circuit to create a general purpose serial to parallel and parallel to serial converter as shown in Diagram 11.4. Clearly repetitive structures of this type can be of any length, and it should be noted that the time taken to load serial data will increase with length. In fact, the inclusion of a serial load facility complicates our previous synchronous design methodology, and therefore it is usual to use a separate clock from the system clock to convert between serial and parallel systems, and to use other control lines to indicate when a conversion is complete and the register can be read.

Note the convention that is normally used that registers are labelled such that the most significant bit has the highest index (PO3 above), and drawn with the most significant bit on the left so that they conform to the same conventions used to write a binary number. In Diagram 11.4 the bits arrive least significant first, though this is not a universal convention.

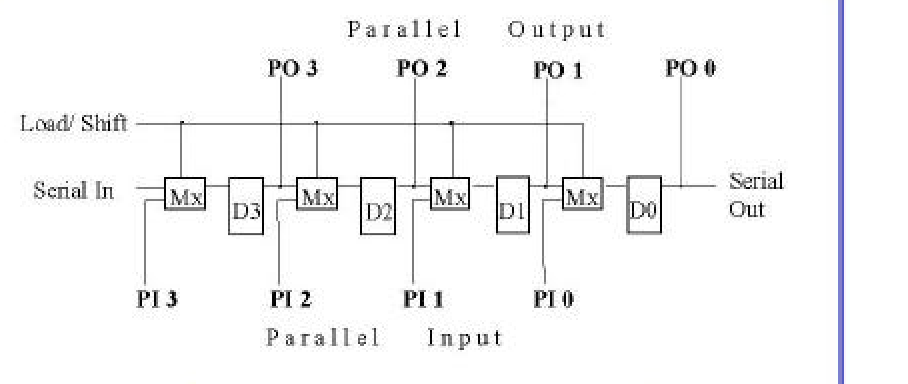

Diagram 11.4: Serial/Parallel and Parallel/serial convertor

Serial to parallel conversion is not the only use of shift registers. It is well known that shifting a register by one place to the left, and filling the bottom bit with zero, is equivalent to multiplying the number by two (providing the register is long enough to represent the result), and conversely, shifting right is equivalent to dividing by two. Thus, in a general purpose shift register we might identify four modes in which we would expect it to function:

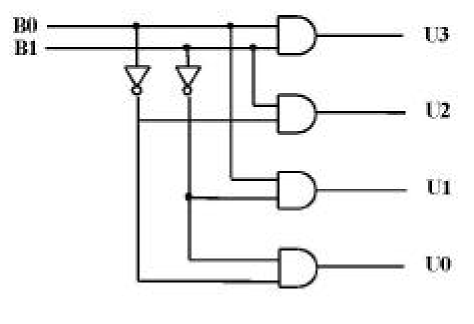

Diagram 11.5: Two bit Binary to Unary decoder

Hold	00
Shift Right	01
Shift Left	10
Parallel Load	11

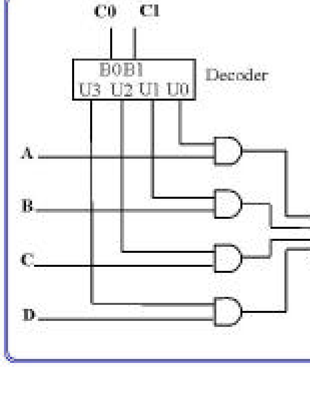

Diagram 11.6: Four line Multiplexer

We can design a register to do this simply by adding functionality to the multiplexor. If we represent each of our four modes by the binary number as shown above, then we require a binary to unary converter for the control of our multiplexor. The circuit for this is diagram 11.5. It will be clear that it is only possible for one of the four output lines to be 1 at any one time. Using this as a component in its own right, we can design the four way multiplexer shown in Diagram 11.6. This is really a four position switch, with the position selected by the binary input on lines C0 and C1. In turn, this component can be incorporated into the general purpose shift register of diagram 11.7. Here just one stage is shown, and can be duplicated to form a register of any desired length. The multiplexer control inputs are not drawn in for simplicity, but will be the same for all stages of the register.

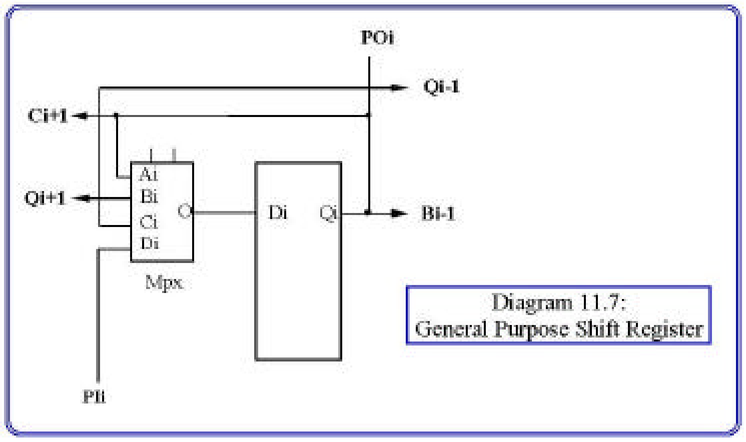

Diagram 11.7:
General Purpose Shift Register

We have already met registers in another form, namely counters, and we have seen that it is possible to design them to operate in synchronisation with the system clock. It is sometimes useful to build counters for the specific purpose of dividing clocks. This is required when we have different timing requirements, such as we noted when considering serial to parallel conversion, and the requirement turns up in other circumstances. To see how this works in practice, consider the one bit counter shown in Diagram 11.8. Here the value of D is taken directly from Q', and therefore at each clock pulse the flip flop will simply change state. Now if we look at the timing diagrams, we see that Q will output a square wave which will be exactly half the frequency of the clock. We can therefore consider the circuit a clock divider. Any counter can be used in this way, remembering that we can always design a synchronous counter to count to any number and then reset to zero. Thus, if we wish to divide a clock by say seven, we can do so by the circuit corresponding to the Moore machine of Diagram 11.9, though the output, which is the divided clock, cannot be made to have an equal mark to space ratio. This is not usually a problem, since in any clock the timing of the falling edge is all that we are concerned with. One obvious application of clock dividers is in wrist watches or quartz clocks. Here, the regulator is a quartz crystal which has a characteristic property that allows a very accurate square wave of around 1MHz to be produced. However, the stepper motor which drives the second hand needs one pulse every second, and hence we need to interpose a circuit which divides the clock by 10^6. This can be done with a synchronous counter with 20 stages, but, such a counter would have a lot of complex state sequencing logic. An alternative is to use a set of successive dividers. For example, we could design a circuit to divide by 256 by concatenating two

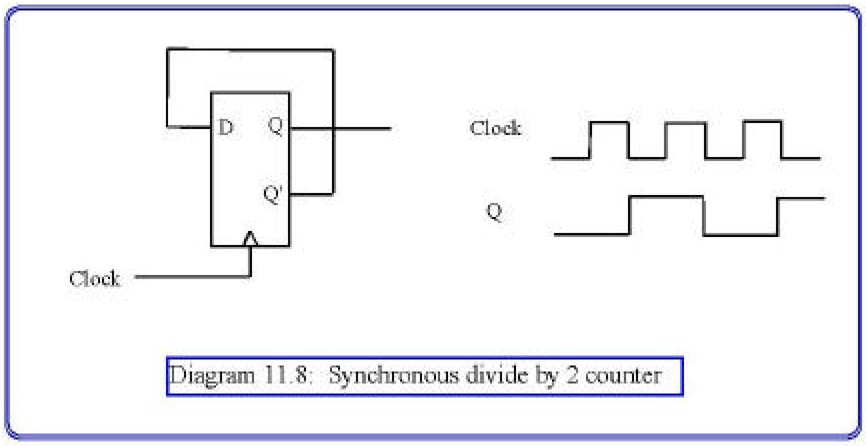

Diagram 11.8: Synchronous divide by 2 counter

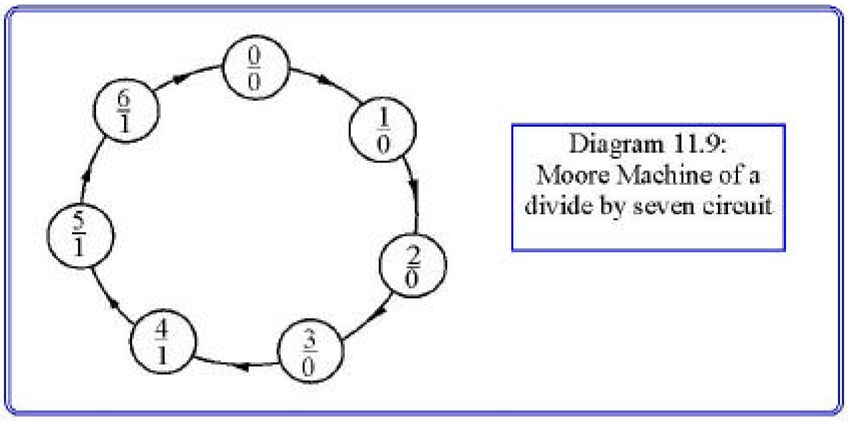

Diagram 11.9:
Moore Machine of a
divide by seven circuit

synchronous divide by sixteen circuits. In this configuration the clock input to the second stage is the output of the first stage as shown in Diagram 11.10. Note that, although a zero to sixteen counter requires a lot of extra logic if we use the D type flip flops, with the J-K flip flops the required sequencing logic is very simple. Hence, this strategy represents a viable way to achieve dividing by large numbers. Taking the above idea to its extreme, we could make our basic elements the divide by two circuit of Diagram 11.8, and this then creates a famous circuit called the ripple through counter, which is shown in Diagram 11.11. If a counter is to be limited to a particular maximum value, then the 'clear' input function normally provided can be used. Thus, if we wish to count to 11, then we would require four stages, and the 'clear' input to each stage would be determined by the minterm Q3•Q2•Q1'•Q0' , which in effect resets the counter to zero when it sees that the output is 12.

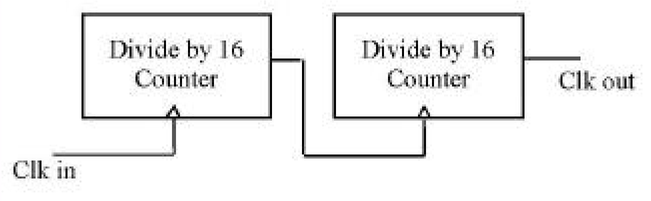

Diagram 11.10: Divide by 256 circuit

It is very important to note that the ripple through counter violates our principle of keeping the digital design synchronous. The correct state will only be present on the outputs at a short time after the counting clock pulse, which will vary depending on how far the change propagates. It is therefore very important to realise that this counter should be used only with extreme care, and only when there is no time critical functionality in the circuit.

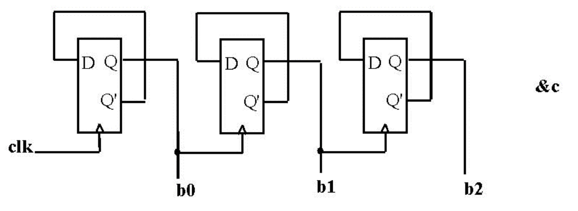

Diagram 11.11: The ripple through counter