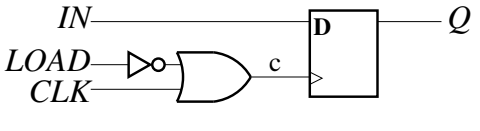
ECE120: Introduction to Computer Engineering

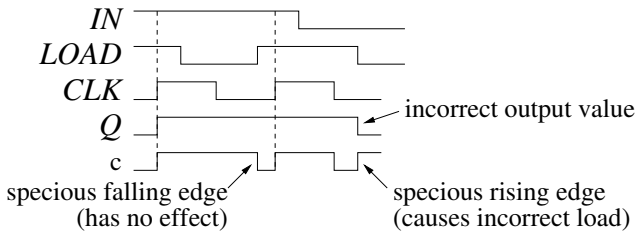
Notes Set 2.7 Registers

This set of notes introduces registers, an abstraction used for storage of groups of bits in digital systems. We introduce some terminology used to describe aspects of register design and illustrate the idea of a shift register. The registers shown here are important abstractions for digital system design.

2.7.1 Registers

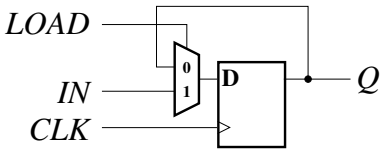
A **register** is a storage element composed from one or more flip-flops operating on a common clock. In addition to the flip-flops, most registers include logic to control the bits stored by the register. For example, D flip-flops copy their inputs at the rising edge of each clock cycle, discarding whatever bits they have stored before the rising edge (in the previous clock cycle). To enable a flip-flop to retain its value, we might try to hide the rising edge of the clock from the flip-flop, as shown to the right.



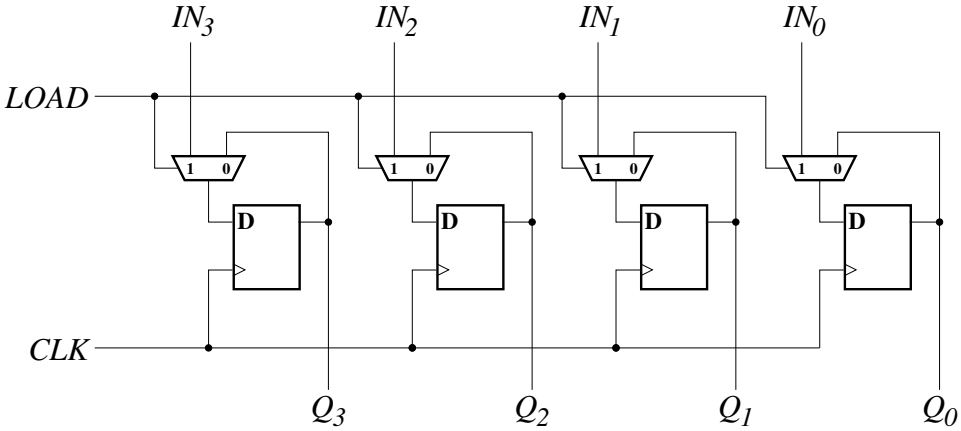


The *LOAD* input controls the clock signals through a method known as **clock gating**. When *LOAD* is high, the circuit reduces to a regular D flip-flop. When *LOAD* is low, the flip-flop clock input, *c*, is held high, and the flip-flop stores its current value. The problems with clock gating are twofold. First, adding logic to the clock path introduces clock skew, which may cause timing problems later in the development process (or, worse, in future projects that use your circuits as components). Second, in the design shown above, the *LOAD* signal can only be lowered while the clock is high to prevent spurious rising edges from causing incorrect behavior, as shown in the timing diagram.

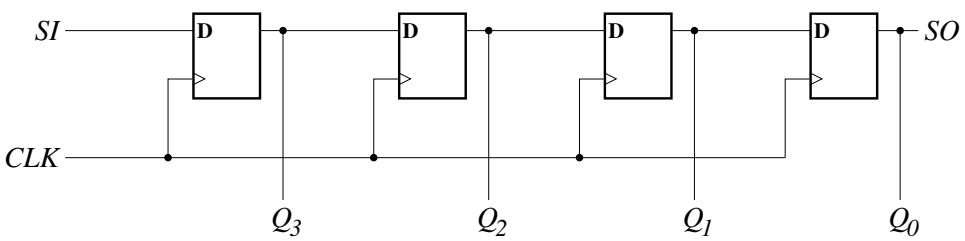
A better approach is to use a mux and a feedback loop from the flip-flop's output, as shown in the figure to the right. When *LOAD* is low, the mux selects the feedback line, and the register reloads its current value. When *LOAD* is high, the mux selects the *IN* input, and the register loads a new value. The result is similar to a gated D latch with distinct write enable and clock lines.



We can use this extended flip-flop as a bit slice for a multi-bit register. A four-bit register of this type is shown to the right. Four data lines—one for each bit—enter the registers from the top of the figure. When *LOAD* is low, the logic copies each flip-flop's value back to its input, and the *IN* input lines are ignored. When *LOAD* is high, the muxes forward each *IN* line to the corresponding flip-flop's *D* input, allowing the register to load the new 4-bit value. The use of one input line per bit to load a multi-bit register in a single cycle is termed a **parallel load**.



2.7.2 Shift Registers

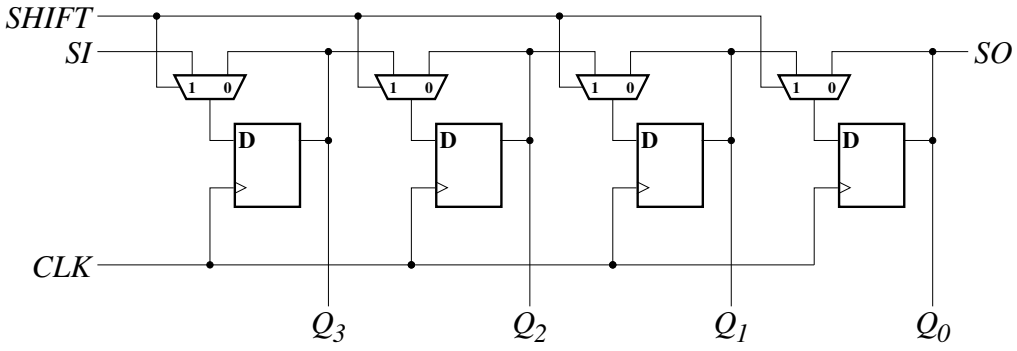


Certain types of registers include logic to manipulate data held within the register. A **shift register** is an important example of this type. The simplest shift register is a series of D flip-flops, with the output of each attached to the input of the next, as shown to the right above. In the circuit shown, a serial input SI accepts a single bit of data per cycle and delivers the bit four cycles later to a serial output SO. Shift registers serve many purposes in modern systems, from the obvious uses of providing a fixed delay and performing bit shifts for processor arithmetic to rate matching between components and reducing the pin count on programmable logic devices such as field programmable gate arrays (FPGAs), the modern form of the programmable logic array mentioned in the textbook.

An example helps to illustrate the rate matching problem: historical I/O buses used fairly slow clocks, as they had to drive signals and be arbitrated over relatively long distances. The Peripheral Control Interconnect (PCI) standard, for example, provided for 33 and 66 MHz bus speeds. To provide adequate data rates, such buses use many wires in parallel, either 32 or 64 in the case of PCI. In contrast, a Gigabit Ethernet (local area network) signal travelling over a fiber is clocked at 1.25 GHz, but sends only one bit per cycle. Several layers of shift registers sit between the fiber and the I/O bus to mediate between the slow, highly parallel signals that travel over the I/O bus and the fast, serial signals that travel over the fiber. The latest variant of PCI, PCIe (e for "express"), uses serial lines at much higher clock rates.

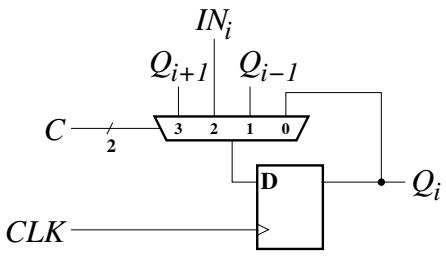
Returning to the figure above, imagine that the outputs Q_i feed into logic clocked at $1/4^{th}$ the rate of the shift register (and suitably synchronized). Every four cycles, the flip-flops fill up with another four bits, at which point the outputs are read in parallel. The shift register shown can thus serve to transform serial data to 4-bit-parallel data at one-quarter the clock speed. Unlike the registers discussed earlier, the shift register above does not support parallel load, which prevents it from transforming a slow, parallel stream of data into a high-speed serial stream. The use of **serial load** requires N cycles for an N-bit register, but can reduce the number of wires needed to support the operation of the shift register. How would you add support for parallel load? How many additional inputs would be necessary?

The shift register above also shifts continuously, and cannot store a value. A set of muxes, analogous to those that we used to control register loading, can be applied to control shifting, as shown below.

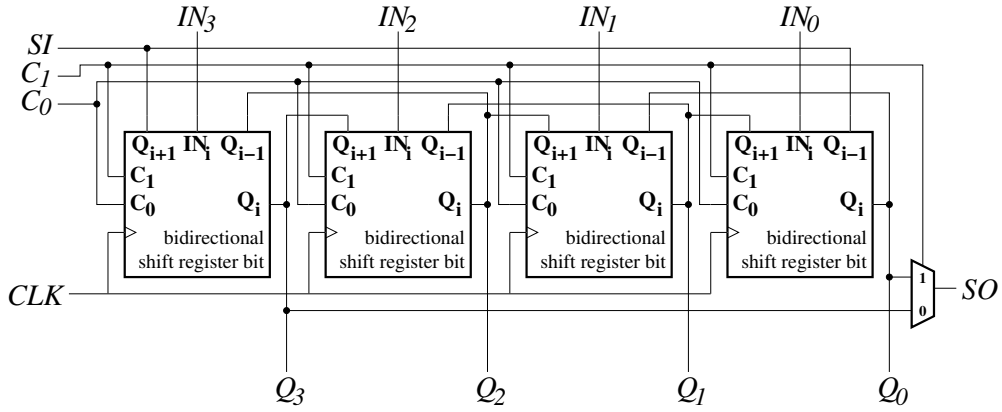


Using a 4-to-1 mux, we can construct a shift register with additional functionality. The bit slice at the top of the next page allows us to build a **bidirectional shift register** with parallel load capability and the ability to retain its value indefinitely. The two-bit control input C uses a representation that we have chosen for the four operations supported by our shift register, as shown in the table below the bit slice design.

The bit slice allows us to build N-bit shift registers by replicating the slice and adding a fixed amount of "**glue logic**." For example, the figure below represents a 4-bit bidirectional shift register constructed in this way. The mux used for the SO output logic is the glue logic needed in addition to the four bit slices. At each rising clock edge, the action specified by C_1C_0 is taken. When $C_1C_0 = 00$, the register holds its current value, with the register value appearing on $Q[3:0]$ and each flip-flop feeding its output back into its input. For $C_1C_0 = 01$, the shift register shifts left: the serial input, SI, is fed into flip-flop 0, and Q_3 is passed to the serial output, SO. Similarly, when $C_1C_0 = 11$, the shift register shifts right: SI is fed into flip-flop 3, and Q_0 is passed to SO. Finally, the case $C_1C_0 = 10$ causes all flip-flops to accept new values from $IN[3:0]$, effecting a parallel load.



C_1C_0	meaning
00	retain current value
01	shift left (low to high)
10	load new value (from IN)
11	shift right (high to low)



Several specialized shift operations are used to support data manipulation in modern processors (CPUs). Essentially, these specializations dictate the glue logic for a shift register as well as the serial input value. The simplest is a **logical shift**, for which SI is hardwired to 0: incoming bits are always 0. A **cyclic shift** takes SO and feeds it back into SI, forming a circle of register bits through which the data bits cycle.

Finally, an **arithmetic shift** treats the shift register contents as a number in 2's complement form. For non-negative numbers and left shifts, an arithmetic shift is the same as a logical shift. When a negative number is arithmetically shifted to the right, however, the sign bit is retained, resulting in a function similar to division by two. The difference lies in the rounding direction. Division by two rounds towards zero in most processors: $-5/2$ gives -2. Arithmetic shift right rounds away from zero for negative numbers (and towards zero for positive numbers): $-5 >> 1$ gives -3. We transform our previous shift register into one capable of arithmetic shifts by eliminating the serial input and feeding the most significant bit, which represents the sign in 2's complement form, back into itself for right shifts, as shown below. The bit shifted in for left shifts has been hardwired to 0.