Registers: Load/Unload & shifters

Prof. Daniele Gorla

Registers

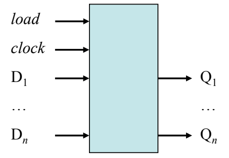
A *register* is a sequence of *n* sorted FFs (for *n* fixed) where we can store a sequence of *n* bits.

Issues realted to registers are:
- loading & unloading of infos
- functionalities provided by the register
- register interconnection

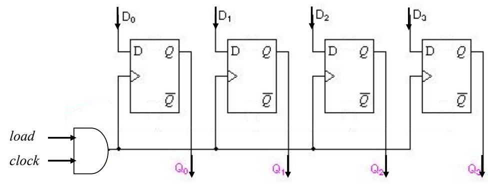
According to the type of data loading/unloading, we have:
- Parallel Input Parallel Output (PIPO)
- Serial Input Serial Output (SISO)
- Serial Input Parallel Output (SIPO)
- Parallel Input Serial Output (PISO)

2

Parallel Input Parallel Output

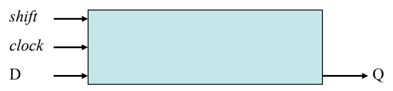


Specification: it riceives *n* data lines in input; the content of such lines is stored when the signal *load* holds 1 and when the clock is at a descending wave front (synchronous net). The stored value is unloadable after a time τ.

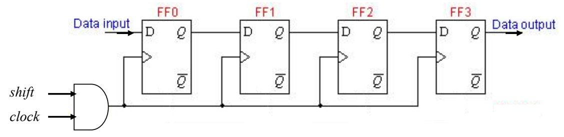


3

Serial Input Serial Output (1)



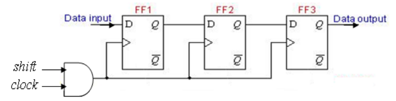
Specification: *(loading)* it riceives in input one data line, whose values are progressively stored at every descending wave front of the *clock*, provided that the *shift* signal holds 1. *(unloading)* The stored *n* values loaded from the input are unloaded along the only output line, one for every descending wave front of the *clock* when the *shift* holds 1.

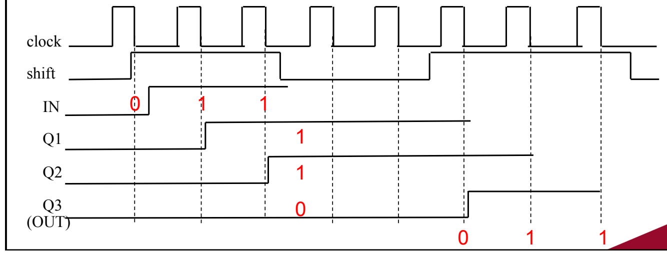


4

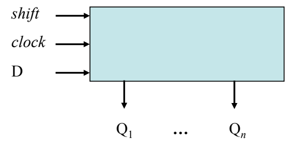
Serial Input Serial Output (2)

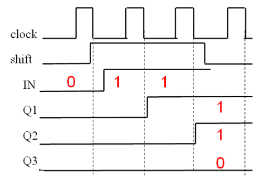
Behaviour: to store *n* bit, we have to keep *shift* at 1 for *n* descending wave fronts of the *clock*; furthermore, at every such front the input data line must provide the next bit to be stored. To unload the register, we have to keep *shift* at 1 for *n* wave fronts of the *clock* and retrieve bits from the output data line.



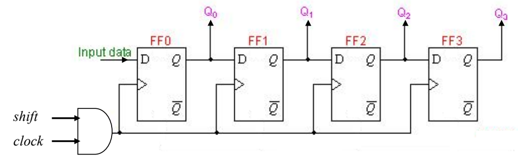


Serial Input Parallel Output

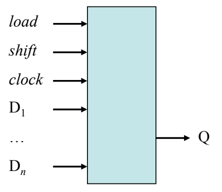




Specification: load like SISO, unload like PIPO

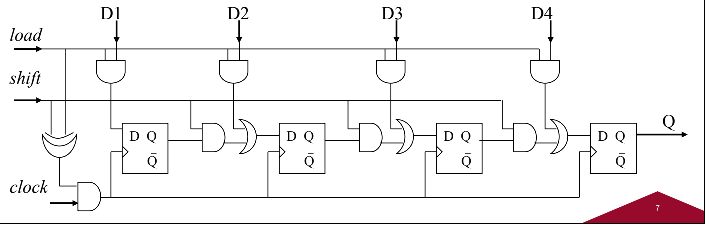


Parallel Input Serial Output (1)



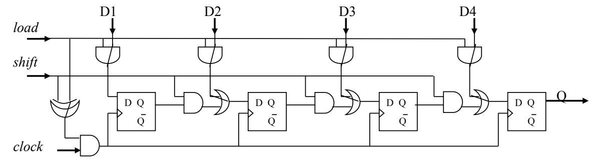
Specification: it stores the *n* inputs of the data lines whenever *load* holds 1 and the clock is at a descending wave front. The stored bits are unloaded whenever *shift* holds 1 (in *n* wave fronts of the clock).

REMARK: *load* and *shift* cannot both hold 1 at the same moment.

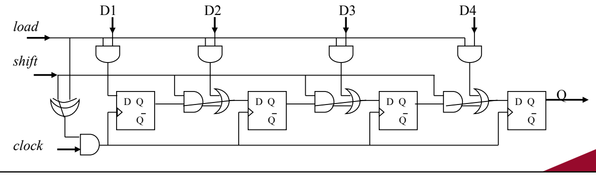


Parallel Input Serial Output (2)

- *shift* = *load:* their XOR is 0, so the register is in sleeping mode
- *shift* = 0, *load* = 1:



- *shift* = 1, *load* = 0:



Registers with Operations

SAPIENZA Università di Roma Dipartimento di Informatica

Independently of the kind of loading/unloading, a register can provide functionalities (operations on the stored bits).

There are 2 fundamental kinds of operations that a register can do:

- Shift
- Count

In this class we shall present the shifters, in the next one the counters.

9

Usage of shifters

SAPIENZA Università di Roma Dipartimento di Informatica

Multiplication and division by 2 is very easy and efficient with shifters:

$$\underbrace{00001000_2 \times 10_2 = 00010000_2}_{\text{left-shift, 1 position}} \qquad \underbrace{00001000_2 \div 10_2 = 00000100_2}_{\text{right-shift, 1 position}}$$

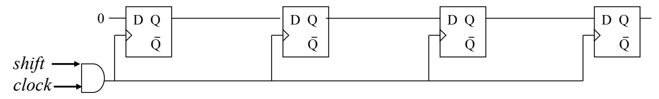
OBS.1: we loose the most/less signifying bit.
OBS.2: we replace the lost bit with a 0 at the opposite side of the register.

10

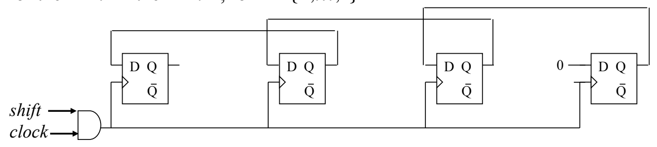
Basic Shifters

SAPIENZA Università di Roma Dipartimento di Informatica

Right shifter:
At every descending wave front of the *clock* when *shift* holds 1, it shifts the content of the FF *i* in the FF *i*+1, for i ∈ {1,…,*n*-1}



Left shifter:
At every descending wave front of the *clock* when *shift* holds 1, it shifts the content of the FF *i* in the FF *i*-1, for i ∈ {2,…,*n*}



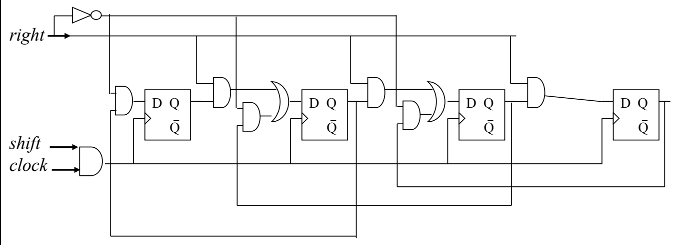
REMARK: we loose the content of FF *n* (right shifter) or of the first FF (left shifter).
The first FF (right shifter) and the FF *n* (left shifter) now contain 0.

11

Bidiretional Shifter

SAPIENZA Università di Roma Dipartimento di Informatica

It behaves like a left/right-shifter according to a control bit *right*: if such a bit holds 1, we have a right shift, otherwise a left one:



12

Shifter with bit holding

If we work with integers in 2-complement, shifting as seen before desn't work anymore to compute a division by 2:

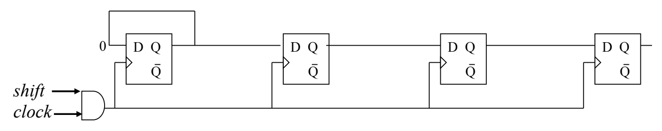
$11111110_{2compl} = -2_{10}$ $11111111_{2compl} = -1_{10}$
$11111110_{2compl} \div 10_2 = 11111111_{2compl}$

By contrast, performing a right shift of 11111110_{2compl} would give us $01111111_{Ca2} = 127_{10}$!!

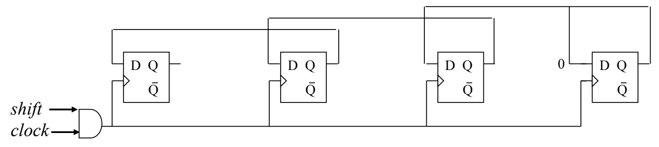
So, when right shifting numbers in 2-complement we have to hold the most signifying bit, in the sense that we don't have to always put it at 0 but we have to keep the previous bit.

Shifter with holding

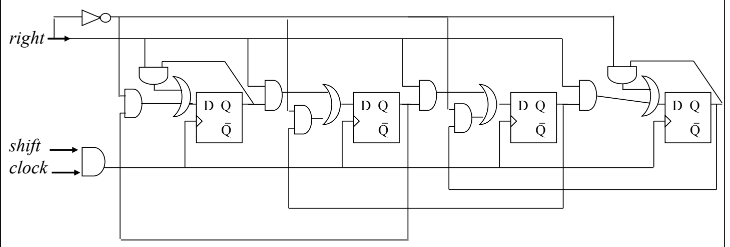
Right shifter:



Left shifter:

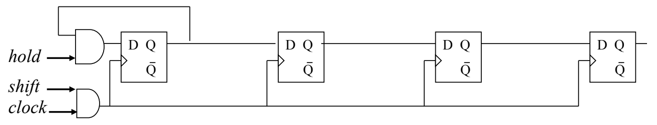


Bidiretional Shifter with holding



Shifter with/without holding

Right shifter:

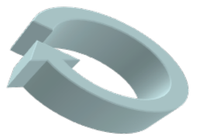


Similarly, we can design
- a left shifter with/without holding
- a bidirectional shifter with/without holding (in this case, we need both signals *hold* and *right*).

Circular Registers

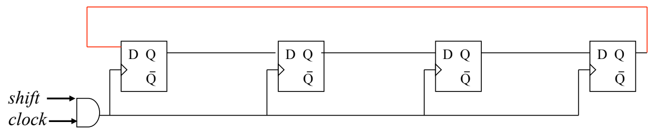
Another way not to loose the exiting bit during a shift is to store it from the other side of the register

In this way, the register is considered as circularly closed on itself:



Counterclockwise Circular Register

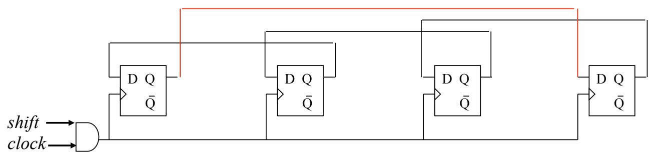
Information stored in the register turns from left to right:



At every descending wave front of the *clock* when *shift* holds 1, the content of FF *i* is moved into FF *i*+1, for $i \in \{1,\ldots,n\text{-}1\}$ and the content of FF *n* in the first FF.

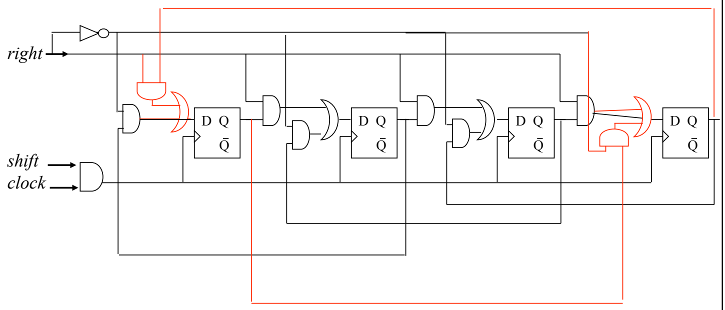
Clockwise Circular Register

Information stored in the register turns from right to left :



At every descending wave front of the *clock* when *shift* holds 1, the content of the FF *i* is moved into the FF *i*-1, for $i \in \{2,\ldots,n\}$ and the content of the first FF into FF *n*.

Bidiretional Circular Register


right
D Q
Q̄
shift
clock