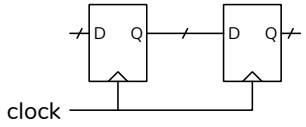
Interfaces

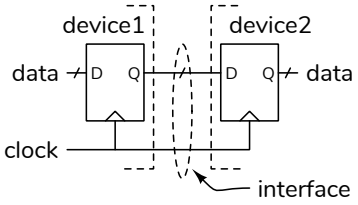
Digital circuits are used to transfer data between modules and devices. This lecture describes the operation and design of some common interfaces.
After this lecture you should be able to: classify an interface as serial or parallel and uni- or bi-directional and explain the advantages of each; ; determine when data is transferred over a ready/valid interface; draw the schematic or write the Verilog for an SPI transmitter or receiver; convert data transmitted over an SPI interface to the interface waveform(s) and extract the data from these waveforms.

Parallel Interfaces

We've seen how data can be transferred between two flip-flops by connecting the Q output of one flip-flop to the D input of another and using a common clock:



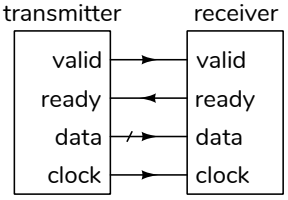
If the two flip-flops are on different devices – whether two IC packages or two pieces of equipment – we can connect them this way to transfer data between them:



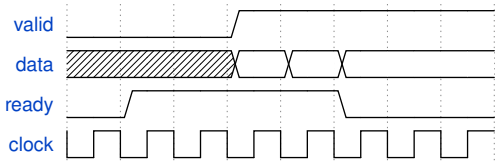
This is the simplest type of interface between two devices and can transfer any number of bits in parallel on the same clock edge.

Ready/Valid Interfaces

A transmitting device may not have data to transfer on every clock edge. Or the receiving device may not be ready to accept data on every clock edge. In this case two "handshaking" signals can control the transfer of data between devices.



The transmitting device asserts a **`valid`** output when its data output is valid. The receiving device asserts a **`ready`** output when it is ready to accept data on the next clock edge[1]. Data is transferred only on clock edges where both **`valid`** and **`ready`** are asserted.



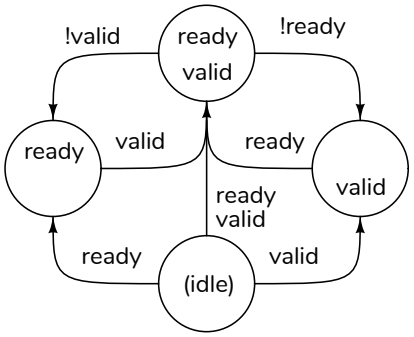
Exercise 1: Mark the clock edges where data is transferred.

Typically, the transmitter and receiver are controlled by state machines and the **`valid`** and **`ready`** outputs are asserted in specific state(s).

A device or module with an output will assert **`valid`** when it enters a state in which its data output is valid. It will stay in this state until the other device has asserted **`ready`** which indicates that the data has been accepted.

A device with a data input will assert **`ready`** when it is in a state where it can accept data on the next rising clock edge. It will stay in this state until a clock edge where **`ready`** is asserted which will cause the data to be loaded into a register in the device.

The state machine for a ready/valid interface is:



[1] This is sometimes called a FIFO (First-In First-Out) interface and can also be used between modules.

This interface requires that both devices use the same clock.

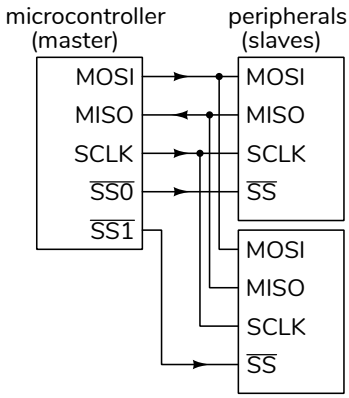
Serial Interfaces

The bits of a word can also be transferred over an interface sequentially (serially), typically one bit at a time. Although serial interfaces are more complex, this is often more than offset by lower costs due to fewer IC pins, smaller connectors, less PCB area, and lower cost cables.

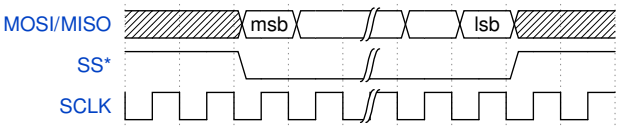
Example: SPI

The Serial Peripheral Interface (SPI, pronounced "ehs-pea-eye" or "spy")) is a common serial interface between a "master" (typically a microcontroller) and a "slave" (typically a peripheral IC). Applications include LCD controllers and SD cards.

The SPI interface has one data signal in each direction (named **MOSI** and **MISO**), a clock signal (**SCLK**) and a (typically active-low) slave-select ($\overline{\text{SS}}$) signal.

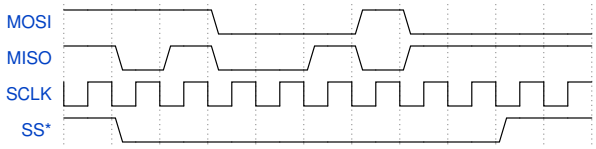


The following timing diagram shows the operation of the bus:



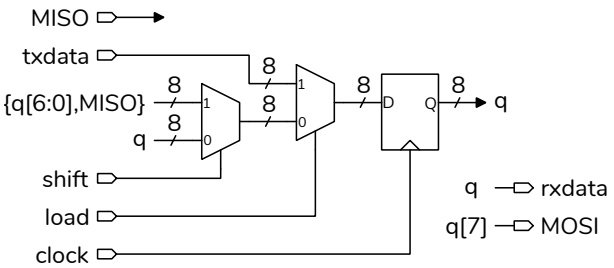
The data transfer begins when the master asserts $\overline{\text{SS}}$. On the following clock edges[2] one bit is transferred. Typically, multiples of 8 bits are transferred, most-significant bit (m.s.b.) first. $\overline{\text{SS}}$ is de-asserted when the transfer is done. Note that the SPI interface transfers the same amount of data in each direction.

[2]SPI interfaces can be configured so that the data is sampled on either the rising or falling edge of **SCLK**.



Exercise 2: The diagram above shows a transfer over an SPI bus. How many bits of data are transferred? What is the value, in decimal, of the data transferred from the master to the slave? From the slave to the master?

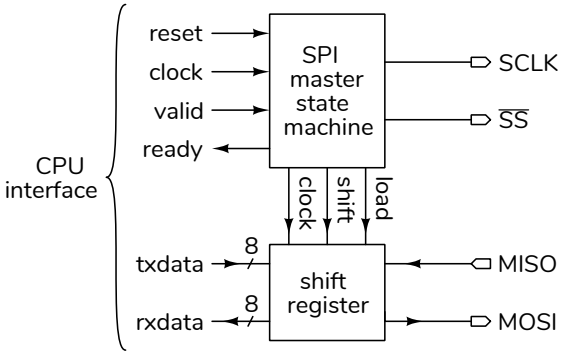
An SPI master interface can be implemented with a shift register that has parallel inputs and outputs as shown below:



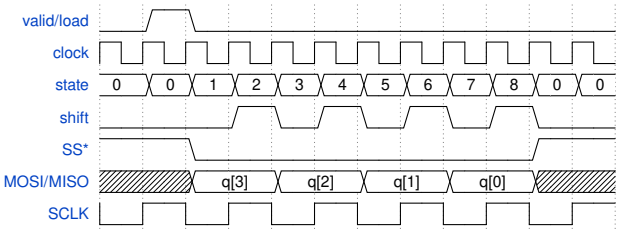
The shift register loads the eight **MOSI** bits to be transmitted from **txdata** when **load** is asserted. When **shift** is asserted a **MISO** bit shifts in on the right and a **MOSI** bit shifts out on the left. At the end of the transfer the eight received **MISO** bits can be read in parallel from the shift register on **rxdata**.

An example of an interface to a CPU is shown below. It has two 8-bit parallel data signals (**rxdata** and **txdata**) and two control signals: **ready**, set true when the interface can accept another byte and **valid**, set true when another byte can be transmitted.

The controller is a state machine that sequences through 16 states to transmit a byte with two states for each bit (one for **SCLK** high and one for **SCLK** low).



The diagram below shows the waveforms for an abbreviated (4-bit) transfer over an SPI interface:



Exercise 3: Based on the diagram above, write a state transition table for an SPI interface controller that transfers four bits at a time. Include an idle state. In which states are `SCLK` and $\overline{\texttt{SS}}$ asserted?

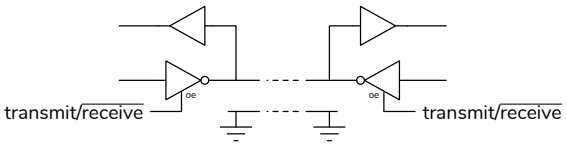
Note that `SCLK` is the clock signal for the interface, not for the interface's logic circuits.

The slave SPI interface will also be implemented with a state machine that synchronises to the transmitter using $\overline{\texttt{SS}}$. Note that both master and slave must be configured for the same bit order and for whether `MOSI`/`MISO` and `SS*` change on the rising or falling edge of `SCLK`.

Bi-Directional Interfaces

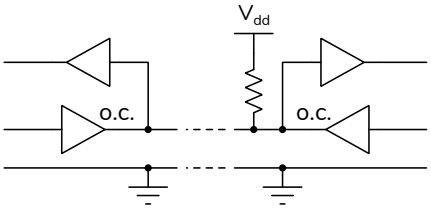
We can further reduce the number of conductors required by using the same ones to transmit data in both directions.

One way is by using tri-state outputs that are alternately enabled so that only one side of the interface is configured as an output at any time:



this is the approach used by USB.

Another way is by using open-collector outputs so that any device can pull the bus low in a "wired-OR" configuration:



This interface allows multiple devices to share the same wiring. However, the RC time constant on the rising edge of the signal limits the possible data rates.