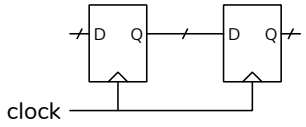
Interfaces

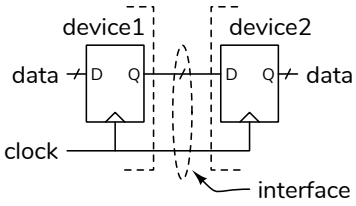
Digital circuits are used to transfer data between modules and devices. This lecture describes the operation and design of some common interfaces.
After this lecture you should be able to: classify an interface as serial or parallel and uni- or bi-directional and explain the advantages of each; ; determine 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:



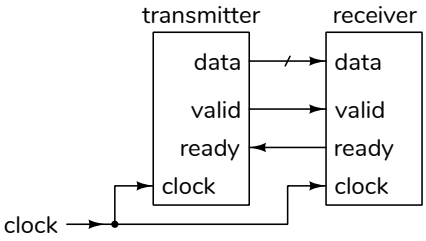
If the two flip-flops are on different devices – whether two IC packages or two pieces of equipment – we can connect them this way to transfer data between them:



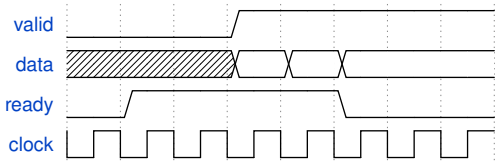
This is the simplest type of interface between two devices and can transfer any number of bits in parallel on the same clock edge.

Ready/Valid Interfaces

There may not be new (valid) data to transfer on every clock edge and the receiving device may not be ready to accept data on every clock edge. Two "handshaking" signals, often called valid and ready, can be used to control the transfer of data[1]:



- 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.
- Data is only transferred on clock edges where *both* **`valid`** and **`ready`** are asserted.



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

The **`valid`** and **`ready`** outputs are generated by the state machines that control the transmitter and receiver.

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

Note that this interface requires that both devices use the same clock.

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

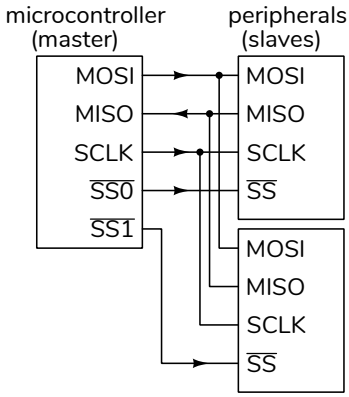
Serial Interfaces

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

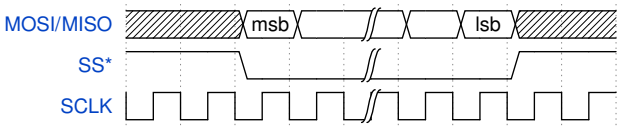
Example: SPI

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.

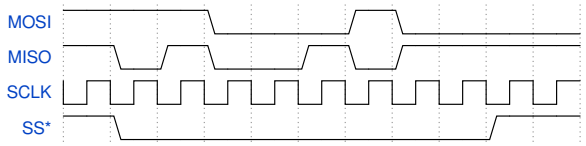


The following timing diagram shows the operation of the bus:



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**.

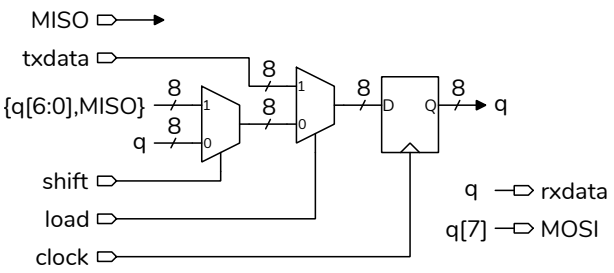


Exercise 2: The diagram above shows a transfer over an SPI bus. How many bits of data are transferred? What is the value, in decimal, of the data transferred from the master to the slave? From the slave to the master?

Implementing an SPI Interface

An SPI interface is typically connected to a CPU over a parallel interface with ready/valid handshaking. We can divide the design of such an interface into two parts: a "datapath" that operates on the bits that are being transferred and a "controller" that controls the operation of the datapath.

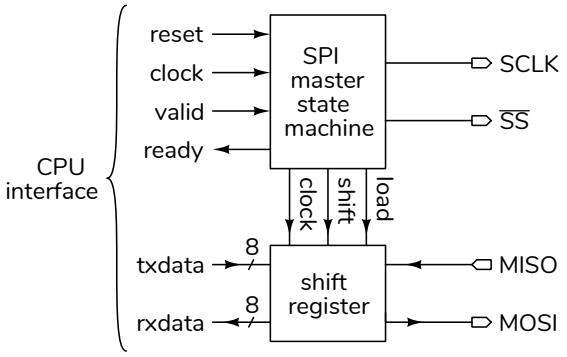
The datapath of an SPI master interface can be implemented with a shift register that has parallel inputs and outputs as shown below:



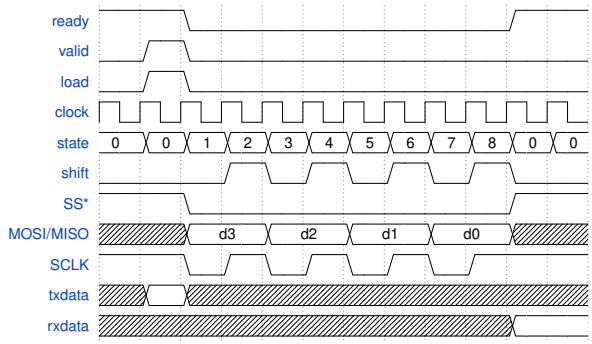
The shift register loads the eight **MOSI** bits to be transmitted from **txdata** when **load** is asserted. When **shift** is asserted a **MISO** bit shifts in on the right and a **MOSI** bit shifts out on the left. At the end of the transfer the eight received **MISO** bits can be read in parallel 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: a **ready** output that is set true when the interface can accept another byte and a **valid** input that is set true when **txdata** holds the next byte to 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 a 4-bit SPI transmit 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 $\overline{\text{SS}}$ 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 $\overline{\text{SS}}$. 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**.

Asynchronous Interfaces

We can simplify the interface further by omitting the clock. This requires that the clock signal be regenerated at the receiver so that the bits can be sampled and shifted in at the correct time.

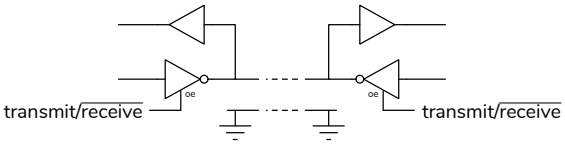
The receiver uses an internal clock running at approximately the same frequency as the transmitter. But it must periodically re-synchronize its clock with the transmitter clock to ensure the two clock's edges remain aligned. To do this the receiver looks for changes in the input data signal in-between its clock edges.

Accurate synchronization thus requires periodic changes in data signal level, even if the data itself does not contain transitions (e.g. it's always low or always high). There are various ways of ensuring this, including "RS-232", Manchester, and Differential coding.

Bi-Directional Interfaces

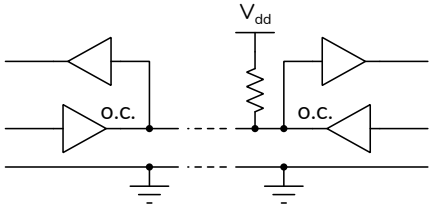
We can further reduce the number of conductors required by using the same ones to transmit data in both directions.

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. This is the approach used by the I^2C "Inter-IC" bus.