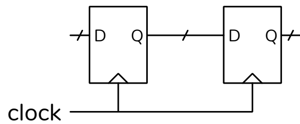
Interfaces

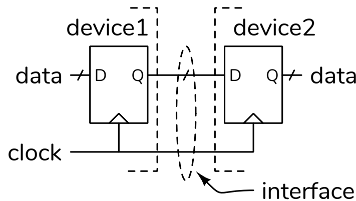
This lecture describes interfaces between digital systems.
After this lecture you should be able to: classify an interface as serial or parallel, synchronous or asynchronous, uni- or bi-directional and explain the advantages of each; draw the schematic or write the Verilog for a synchronous serial transmitter or receiver; extract the data transmitted over an SPI interface from the interface 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 we can use a cable to connect them and this allows us to transfer data between devices:



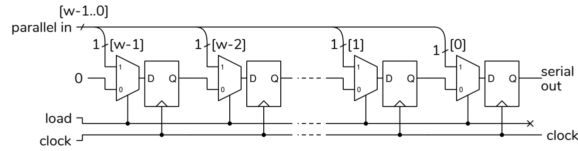
This type interface is the simplest interface between two devices. We can transfer any number of bits on the same clock edge.

An examples is the 8-bit parallel printer port that was used by early personal computers. This interface used the falling edge of a **STROBE** signal as the clock and an active-high **BUSY** signal that indicated the printer was unable to accept another character.

Serial Interfaces

We can reduce the width of the data bus between the two devices down to 1 bit. This reduces the number of conductors required in the interface cable.

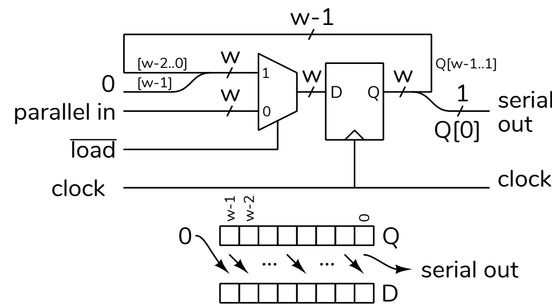
Words of more than one bit can be transferred over the interface sequentially (serially). The conversion from a word whose w bits are available in parallel to a sequence of bits can be done using a shift register whose flip-flops can be loaded in parallel as shown below:



This circuit loads w bits into the flip-flops in parallel when the **load** input is asserted and otherwise shifts one bit per clock cycle to the **serial out** output.

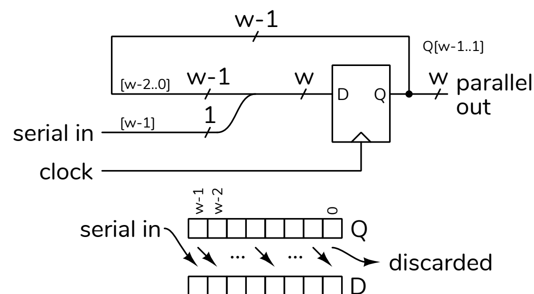
Exercise 1: How could you modify the circuit so that the bits are transmitted in order from the most-significant to the least-significant bit?

We can also draw this as:



where the w-bit register is loaded from the parallel input when $\overline{\textbf{load}}$ is asserted or from the current register value shifted right by one bit. The least-significant bit of the register is the serial output.

At the receiving side of the interface the serial stream of bits needs to be converted back to a w-bit word. We can do this with another shift register:



in this case the serial input bits are shifted into the w-bit shift register through the most-significant bit.

The example below shows how the value `1101` is transferred over the interface. The columns show the values on the $\overline{\texttt{load}}$ signal, the parallel input, the transmitter register, the receiver register, and a **`valid`** signal that indicates that the value in the receiver register is valid. The lines correspond to register values between rising edges of the clock. **`aaaa`** is the previous value transferred over the interface and **`bbbb`** is the next value to be transferred.

$\overline{\texttt{load}}$	parallel in	transmit register	receive register	valid
0	1101	000a	aaa-	0
1	----	1101	aaaa	1
1	----	0110	1aaa	0
1	----	0011	01aa	0
0	bbbb	0001	101a	0
1	----	bbbb	1101	1

The word to be transmitted is placed on the parallel input and $\overline{\texttt{load}}$ is set low. After the next rising clock edge this value has been loaded into the transmit register and the first (least-significant) bit is available on the serial output. On each subsequent rising clock edge the bit on the serial interface is stored in the receiver register and another bit is placed on the serial output. This continues until all the bits are loaded into the receiver register. When the final bit is captured the **`valid`** output indicating that the register contents are valid is asserted. This could be used to load the word into another register.

A complete design requires a state machine to set the $\overline{\texttt{load}}$ signal low when there is a word at the parallel input ready to be transmitted and to generate a **`done`** status signal when the word has been transmitted so that another word can be sent.

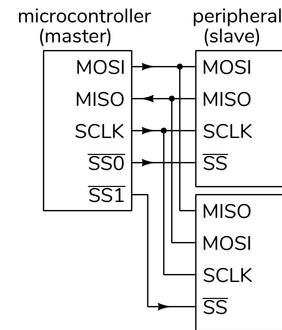
A state machine is also required at the receiver to determine that w bits have been shifted into the receiver register and assert the **`valid`** output.

The receiver's state machine must synchronise itself to the transmitter. The details will depend on the serial interface and the interface to the device. Both interfaces must also send the bits in the same order and transfer the same number of bits per word.

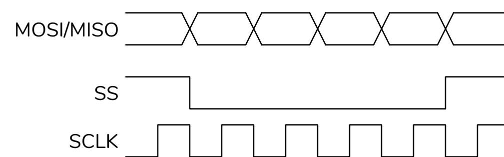
Example: SPI

The Serial Peripheral Interface (SPI) is a common interface between a microcontroller (typically the "master") and a peripheral IC (the "slave"). Applications include LCD controllers and SD cards.

The SPI interface has separate data in and data out lines (labelled MOSI and MISO), a clock signal and a slave-select (SS) signal.



The following timing diagram shows the operation of the bus:



The data transfer begins when the master asserts $\overline{\texttt{SS}}$. On the following clock edges[1] one bit is transferred in each direction. Typically, multiples of 8 bits are transferred, most-significant bit first. $\overline{\texttt{SS}}$ is deasserted when the transfer is done.

Asynchronous Interfaces

We can omit the clock from the interface to reduce the number of conductors required. 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 has an internal clock running at approximately the same frequency as the transmitter. But it must "synchronize" its clock with the transmitter clock so the clock edges are aligned. To do this the receiver uses a clock that operates at some multiple of the transmitter's clock and looks for changes in the input signal in-between it's clock edges.

Accurate synchronization thus requires periodic changes in data signal level, even if the data is constant (e.g. all zero). There are various ways of ensuring this:

[1]SPI interfaces can be configured so that the data and $\overline{\texttt{SS}}$ change on either the rising or falling edge of **`SCLK`**.

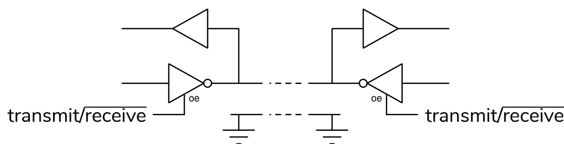
- sending a pair of bits of opposite level in-between words (the "stop" and "start" bits used in "RS-232" serial interfaces);
- encoding each bit as either a pair of 0-1 or 1-0 bits ("Manchester" coding) as used by the 10 Mbps 10BASE-T Ethernet;
- inserting an extra bit when long runs of the same polarity are detected such as the "bit stuffing" used by USB.

Most common serial interfaces, including the ones mentioned above, transmit the data bits least-significant bit first.

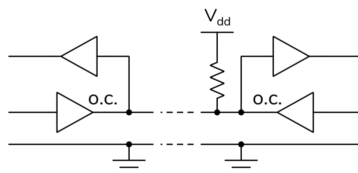
Some serial interfaces encode their bits differentially. For example USB uses an "NRZI" encoding where a zero data bit is transmitted as a change in level, and a "one" bit as no change.

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 (2.0). Another is by using open-collector outputs so that multiple devices can pull the bus low in a "wired-OR" configuration:



as is used with I^2C.

Often, one device is a master and "polls" the slave. After the poll the master turns off its driver and the slave turns on its driver for the duration of the response. However, there are also interfaces (e.g. I^2C, see below) where multiple devices can contend to become the bus master using an "arbitration" protocol.

Example: USB

Universal Serial Bus (USB) is a popular peripheral interface that uses an asynchronous bidirectional serial interface. Earlier versions ("USB 2.0") of the interface required only four conductors: two for a differential[2], bi-directional data signal, one for ground, and one for power (+5 V).

Other Interfaces

Addressable Interfaces

Instead of using $\overline{\texttt{SS}}$ lines to enable specific devices, it's possible to design interfaces where a device can be enabled by sending its address on the data lines before sending it data. The Inter-IC Communications (I^2C) protocol is an example. This bus allows IC's to be connected in parallel using two signals: **SDA** (data) and **SCL** (clock), both of which use open-collector bidirectional buses.

SerDes

A serializer/de-serializer (SerDes) is the name used for serial interfaces that operate at high (GHz) speeds. These are typically used for high-speed peripheral interfaces such as HDMI, PCIe and SATA or for fiber-optic communication links.

Device Descriptors

Some buses support retrieval of "descriptors" – blocks of data that identify a device. For example, USB peripherals contain a descriptor that identifies the device type (e.g. a keyboard), the manufacturer and the model.

[2] The data is encoded as the difference between the two signals rather than their voltage relative to ground.