SPI Serial I/O

ECE 476 Advanced Embedded Systems

Jake Glower - Lecture #7

Please visit Bison Academy for corresponding
lecture notes, homework sets, and solutions

Introduction:

So far, we've used parallel I/O

- Each pin controls one LED
- Each pin reads one button

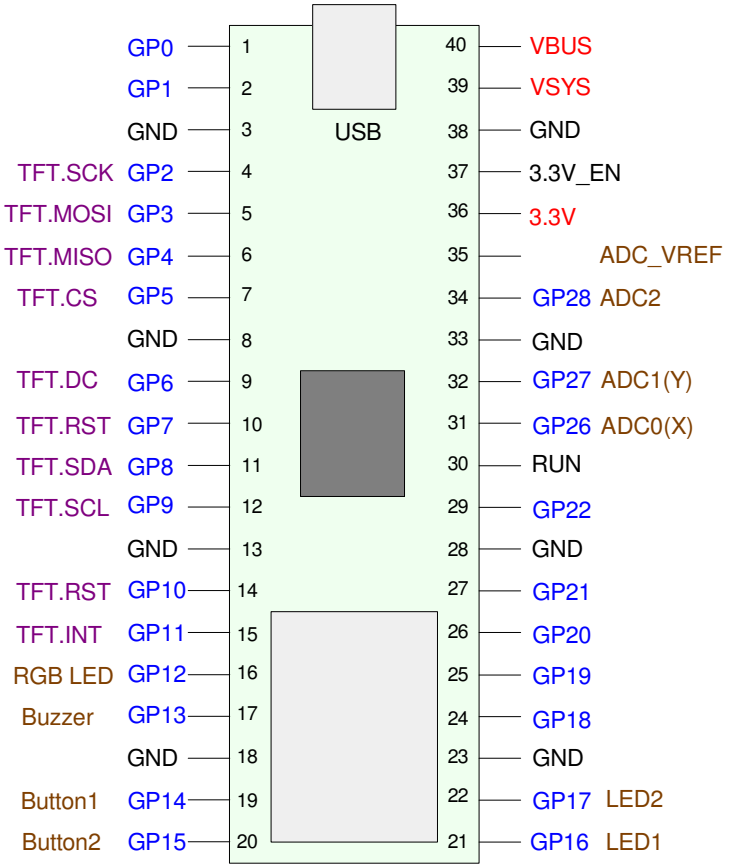
Problem:

- This uses up a lot of I/O pins

Some pins have other funcitons

- GP0/1: Serial port (UART0)
- GP2..11: LCD graphic display
- GP13..17: Buzzer, LEDs, buttons
- GP26..28: Analog inputs

There just aren't a lot of unused pins



Serial vs. Parallel

Fortunately, there is a way to get more I/O pins

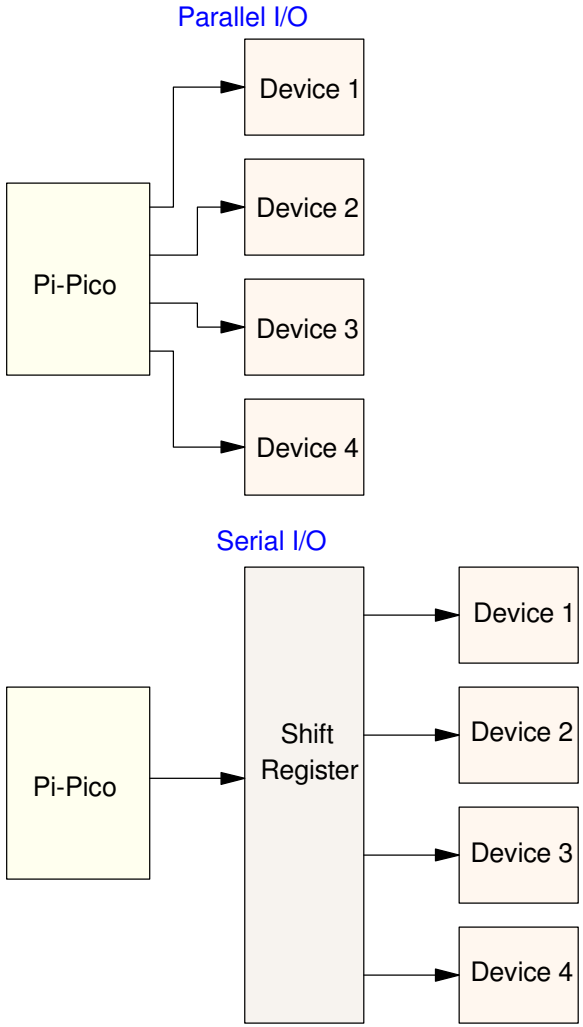
- Use serial I/O rather than parallel

Parallel I/O:

- (previous lectures)
- Each GPIO pin drives one device
- Communications takes one clock

Serial I/O:

- (this lecture)
- Each GPIO pin drives many devices
- Communications takes several clocks



SPI & I2C Serial Communications

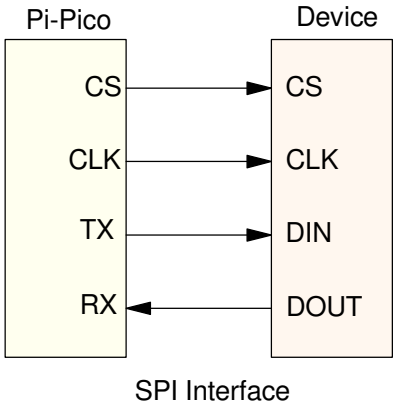
The two main forms of serial communications are SPI and I2C.

Both forms are similar

- SPI came from Motorola
- I2C came from Phillips Semiconductors.

SPI:

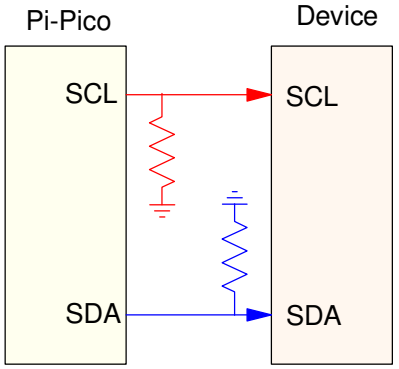
- 3 or 4 wire interface
- Capable of full-duplex communications
- Capable of 40+ Mbps communications
- Supported by the Pi-Pico



SPI Interface

I2C

- 2 wire interface
- Limited to 400k bps
- Supported by the Pi-Pico



I2C Interface

SPI Ports on a Pi-Pico

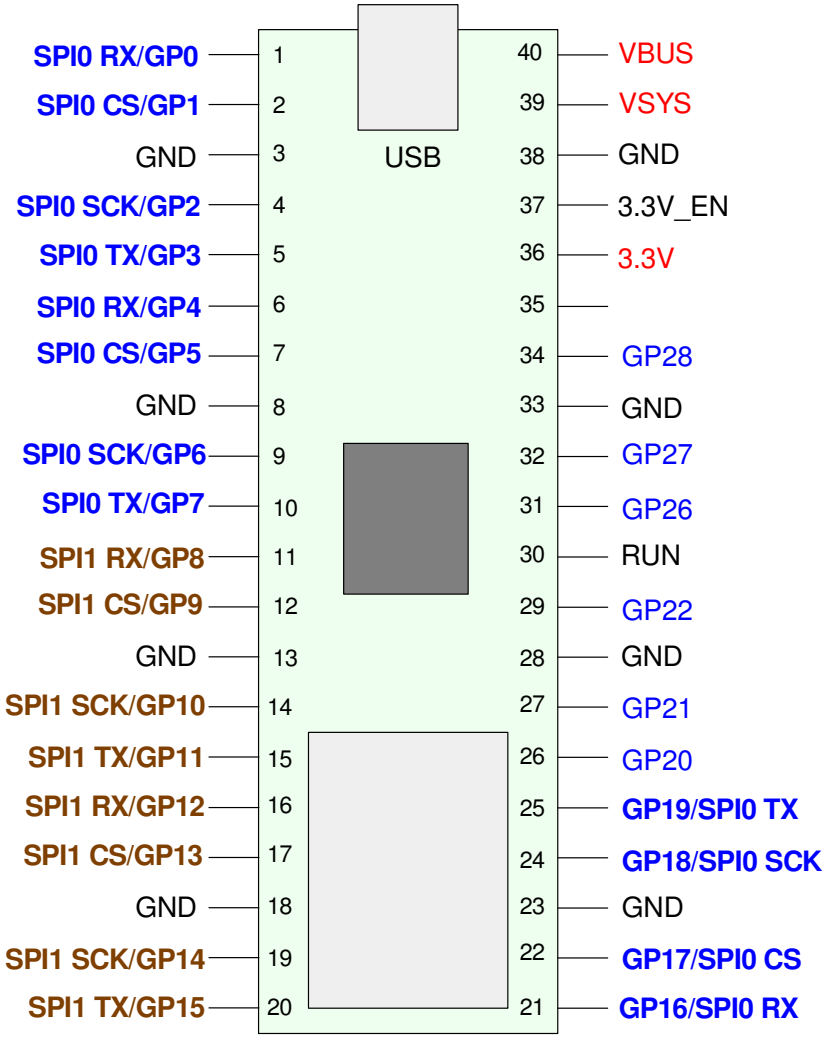
The Pico has two SPI ports

- SPI0
- SPI1

Each SPI port has four pins

- TX (MOSI)
- RX (MISO)
- CS
- SCK

These can be assigned to several pins



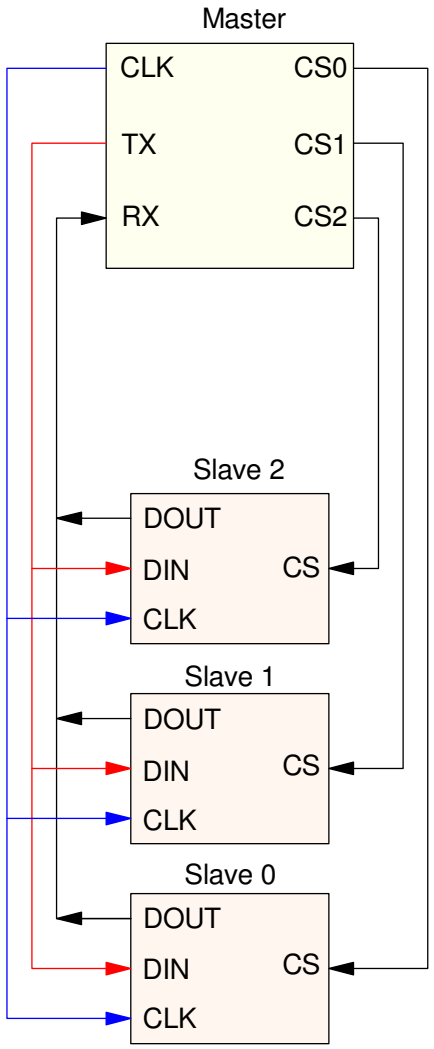
SPI Communications

Four signals (plus common ground):

- SCK: The clock line
- TX (MOSI): Master Out, Slave In (write bus)
- RX (MISO): Master In, Slave Out (read bus), and
- CS: Chip Select

SCK, TX, and RX can be shared

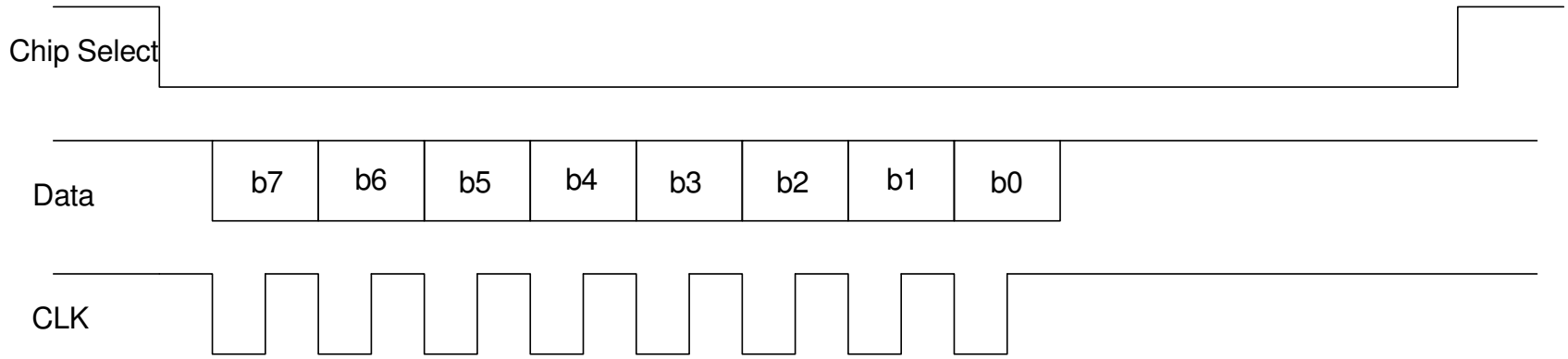
- CS needs to be different for each device



SPI Data Format

Typically, an SPI message proceeds as follows:

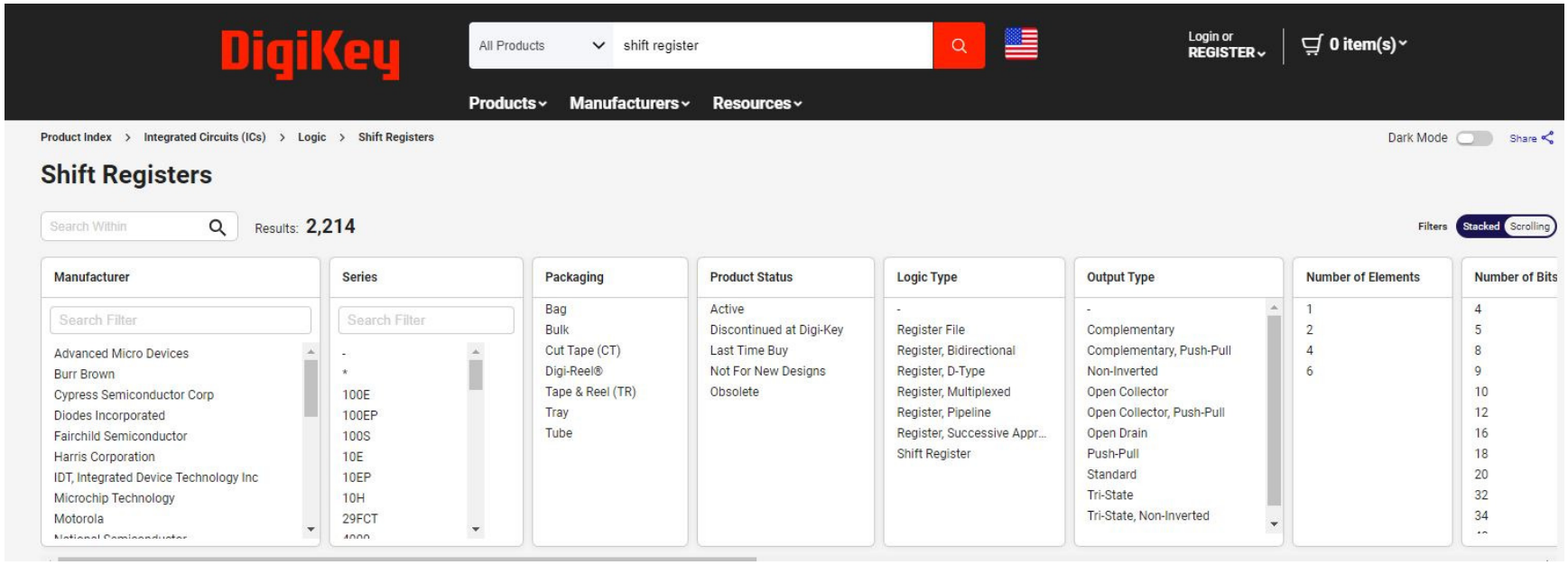
- Chip Select goes low to start a message
- Bits are sent out on the MOSI line, one by one,
- Bits are read on the MISO line, one by one, and
- Each bit is synchronized by a clock line (sent by the master)
- At the end of the message, Chip Select goes high



Typical signals on an SPI bus with 8-bits of data

Serial Input: 74HC165

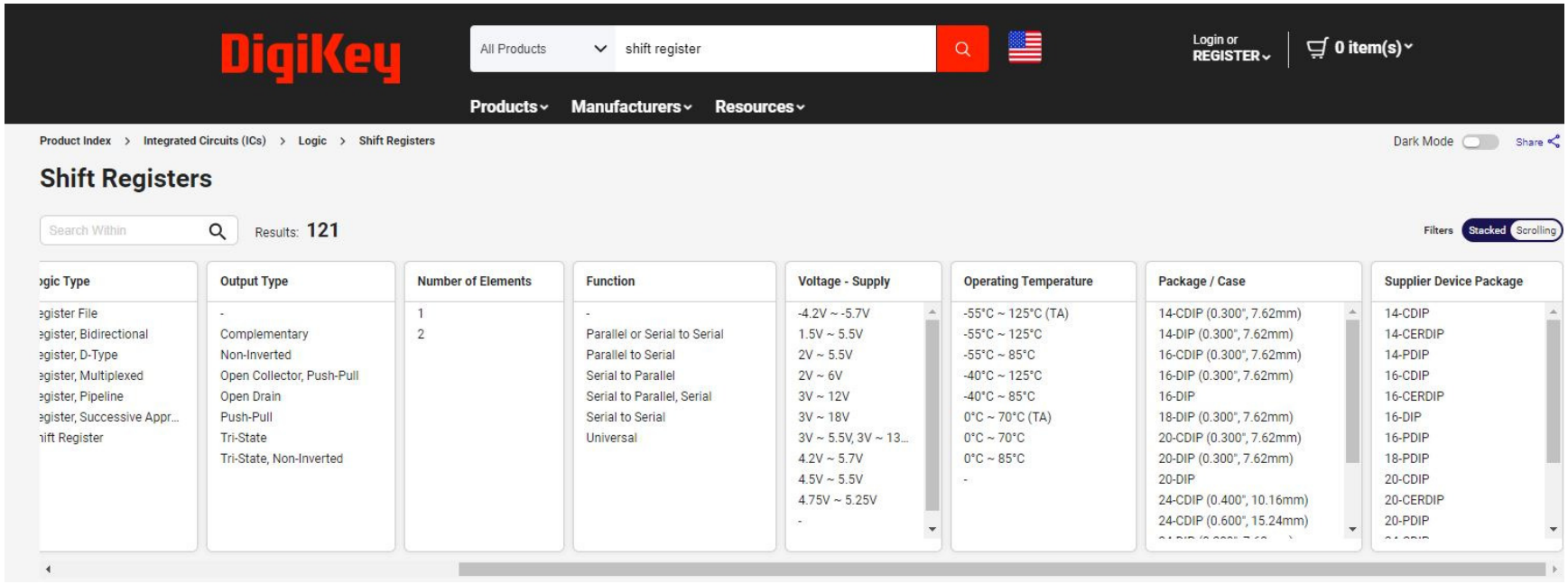
Starting out, lets look at having a Pi-Pico read eight binary inputs just a 3-wire SPI interface (since this is a read operation, the MOSI line isn't needed). If you search Digikey using the term *shift register*, you'll get 2214 hits (as of April 1, 2024).



Narrow the search as

- In-Stock
- Active
- 8-bits
- Through Hole

and you're down to 121 hits.



Narrow to *Parallel to Serial* (serial input) and you now have a manageable number of options. One that looks promising is a 74HC165. Select this one and pull up the datasheets

SNx4HC165 8-Bit Parallel-Load Shift Registers

1 Features

- Wide Operating Voltage Range of 2 V to 6 V
- Outputs Can Drive Up to 10 LSTTL Loads
- Low Power Consumption, 80-μA Maximum I_{CC}
- Typical t_{pd} = 13 ns
- ±4-mA Output Drive at 5 V
- Low Input Current of 1 μA Maximum
- Complementary Outputs
- Direct Overriding Load (Data) Inputs
- Gated Clock Inputs
- Parallel-to-Serial Data Conversion
- On Products Compliant to MIL-PRF-38535, All Parameters Are Tested Unless Otherwise Noted. On All Other Products, Production Processing Does Not Necessarily Include Testing of All Parameters.

3 Description

The SNx4HC165 devices are 8-bit parallel-load shift registers that, when clocked, shift the data toward a serial (Q_H) output. Parallel-in access to each stage is provided by eight individual direct data (A−H) inputs that are enabled by a low level at the shift/load (SH/$\overline{LD}$) input. The SNx4HC165 devices also feature a clock-inhibit (CLK INH) function and a complementary serial ($\overline{Q}_H$) output.

Clocking is accomplished by a low-to-high transition of the clock (CLK) input while SH/LD is held high and CLK INH is held low. The functions of CLK and CLK INH are interchangeable. Because a low CLK and a low-to-high transition of CLK INH also accomplish clocking, CLK INH must be changed to the high level only while CLK is high. Parallel loading is inhibited when SH/$\overline{LD}$ is held high. While SH/$\overline{LD}$ is low, the parallel inputs to the register are enabled independently of the levels of the CLK, CLK INH, or serial (SER) inputs.

Page #14 of the data sheets tell you

- This device operated at up to 35MHz, and
- It gives the wiring diagram,

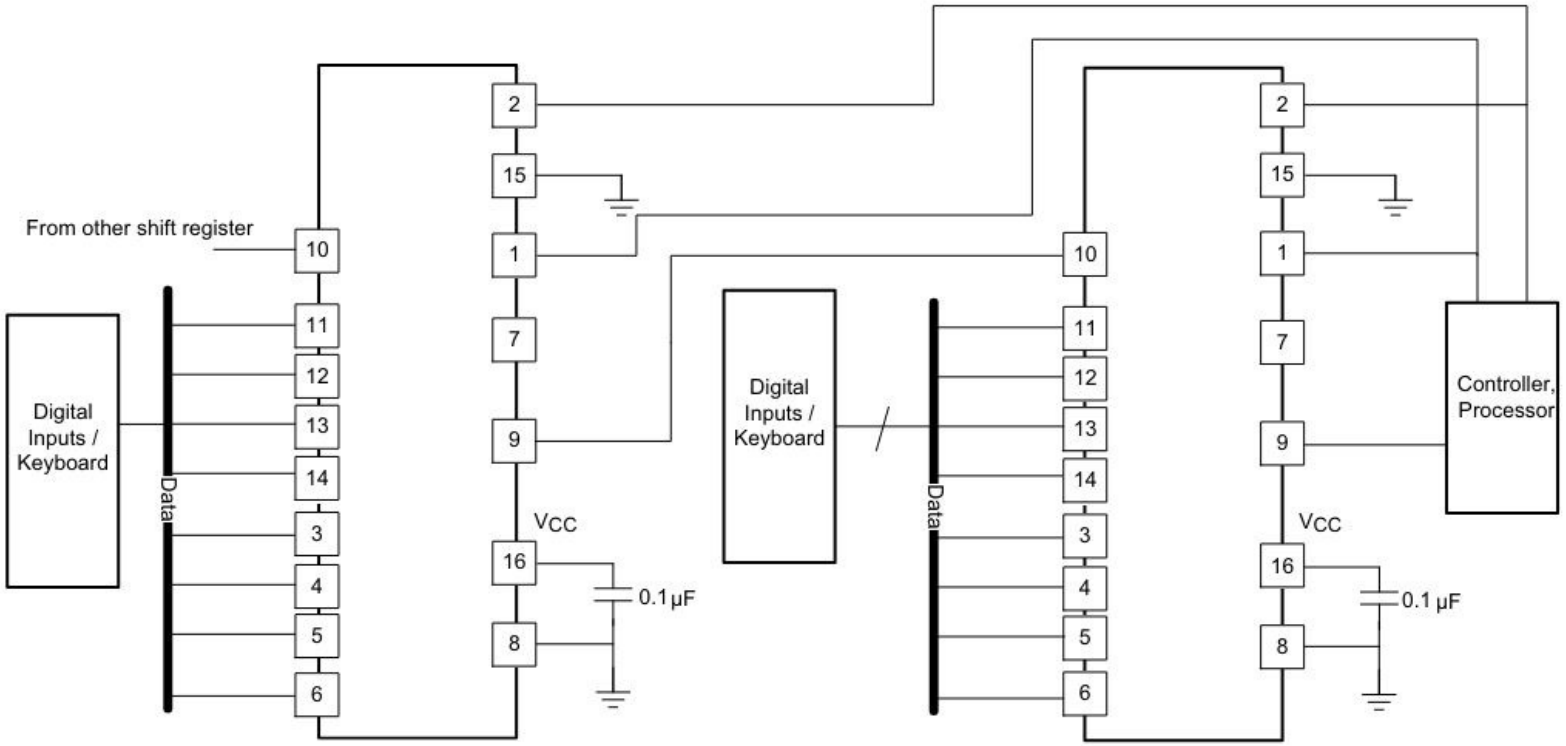
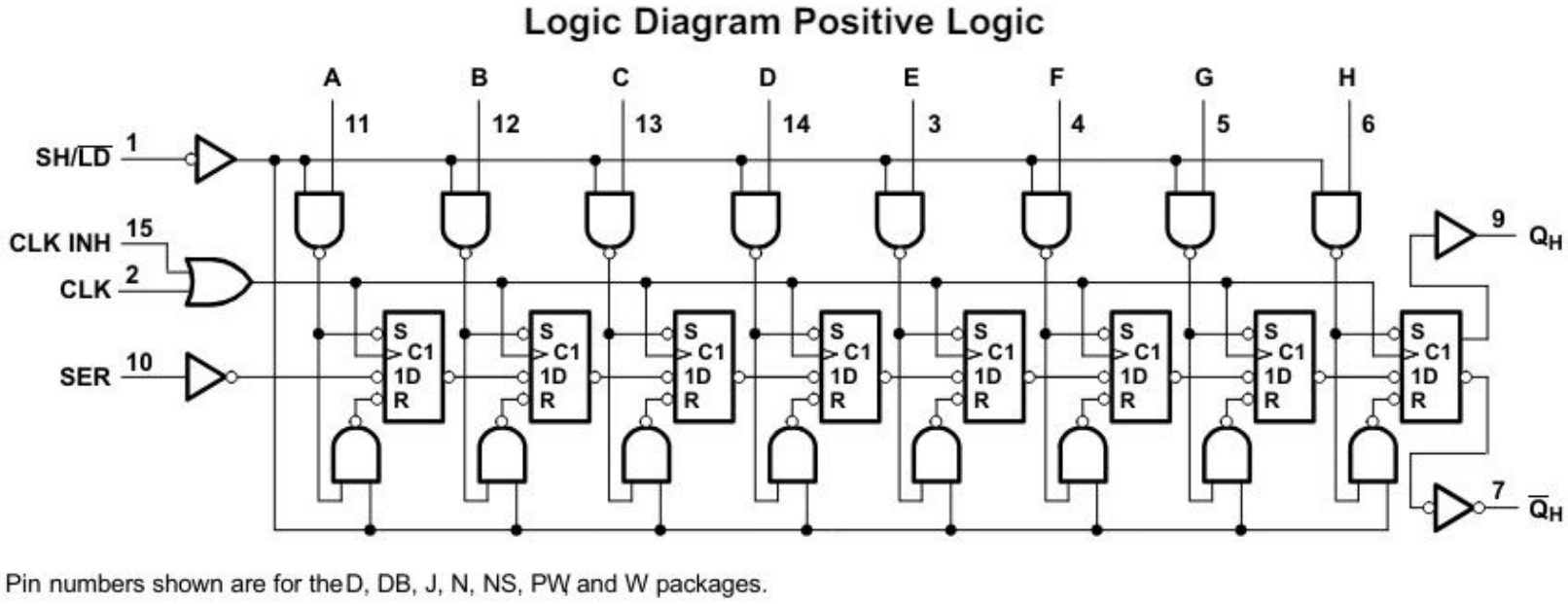


Figure 5. Typical Application Diagram for SN74HC165

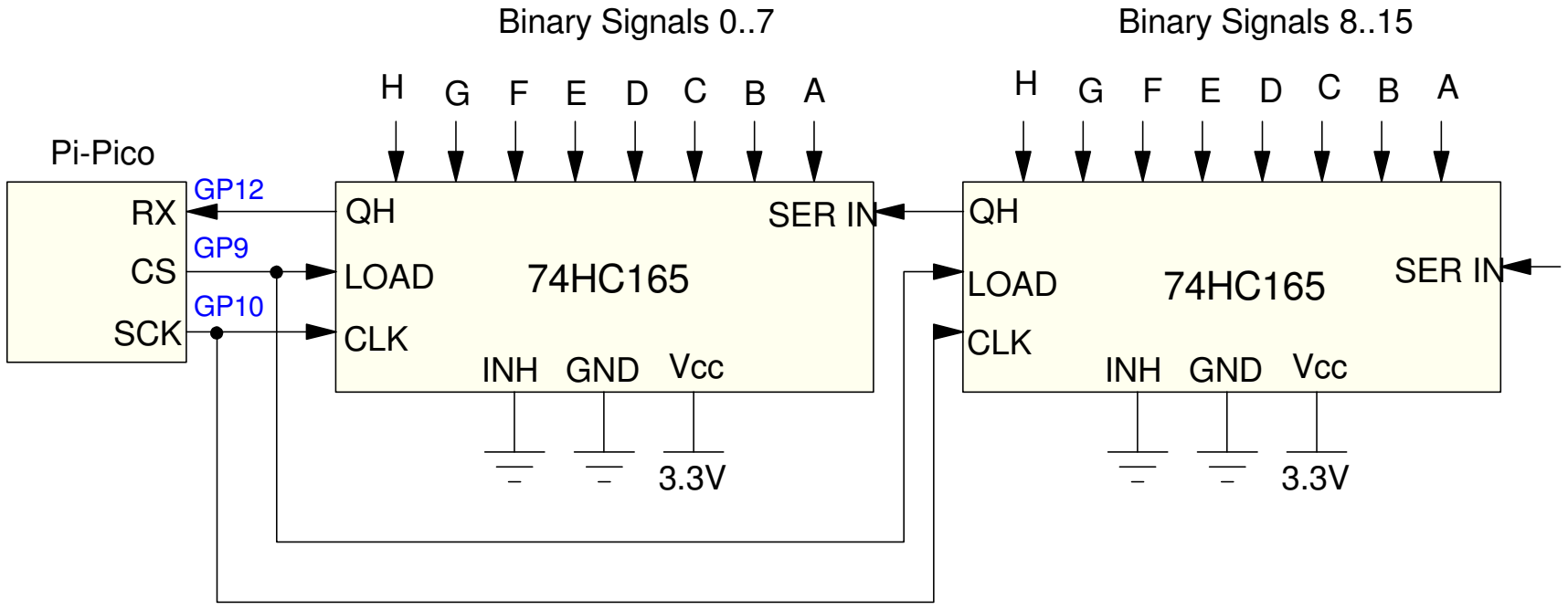
Data sheets shows you the schematic of the device, along with a timing diagram.



Schematics for a 74HC165 Shift Register

This is enough to set up the hardware

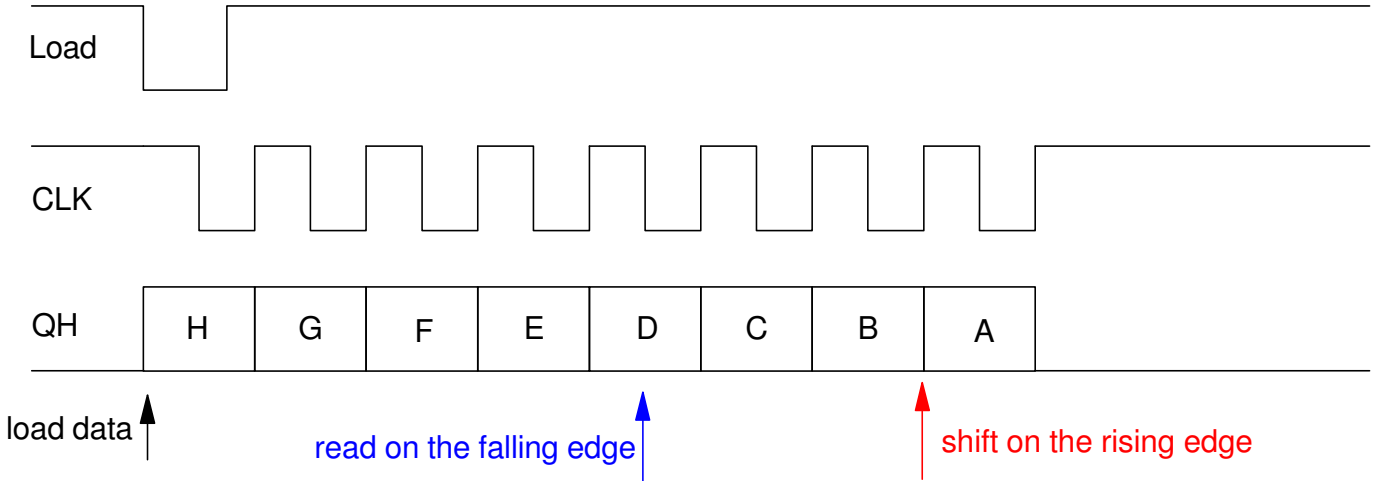
- Each 74HC165 adds eight binary inputs
- You can cascade these to get more binary inputs
- Using just three GPIO pins on the Pi-Pico



Wiring for a 74HC165: Parallel to Series IC

To read 8 binary inputs

- Pull LOAD low then high
- Pulse the clock low.
- Read in the first bit (H),
- Pulse the clock high. This shifts the data in the shift register
- repeat 8x for reading 8 bits, 16x to read 16 bits



Bit-Banging:

Set/clear each bit manually

- Hold for 10ms
- Time can be dropped to 1us

Net result (time changed to 1us)

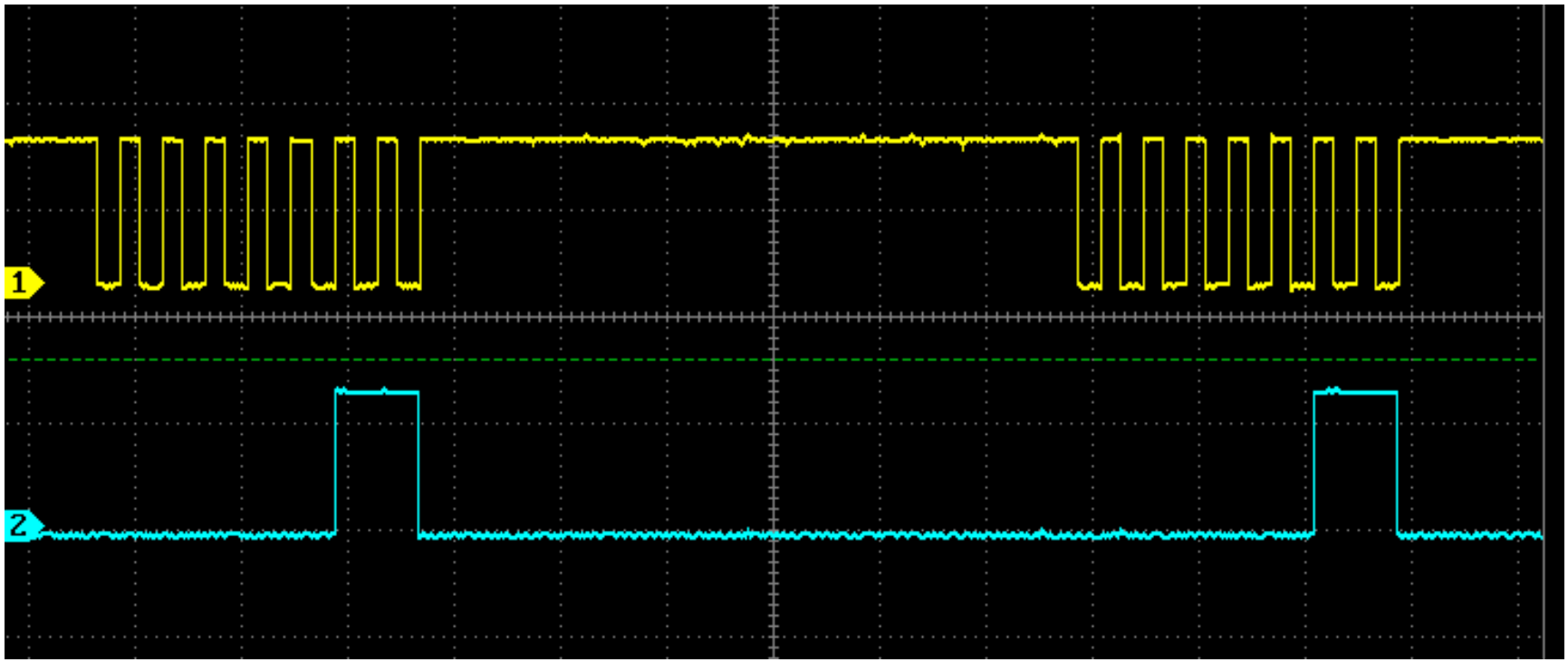
- Read 8 bits of data in 18us
- Read 16 bits of data in 34us

All while using just two wires.

```
from machine import Pin
from time import sleep_ms, sleep_us

CLK = Pin(10, Pin.OUT)
DIN = Pin(11, Pin.IN, Pin.PULL_UP)
LATCH = Pin(9, Pin.OUT)

def HC165():
    LATCH.value(1)
    CLK.value(1)
    sleep_ms(10)
    LATCH.value(0)
    sleep_ms(10)
    LATCH.value(1)
    # data is latched - now shift it in
    X = 0
    for i in range(0,8):
        CLK.value(0)
        sleep_ms(100)
        X = (X << 1) + DIN.value()
        CLK.value(1)
        sleep_ms(100)
        print(i, X)
    return(X)

while(1):
    Y = HC165()
    print(Y)
```

Oscilloscope Results

- Oscilloscopes are your friend
- They let you see what's really happening
- Data (blue) is shifted on the rising edge of CLK (yellow)
- Data should be read on the falling edge of CLK



Bit-Banging vs. Hardware

Bit-banging has some advantages:

- You have complete control of each signal
- You can use any I/O pins for the SPI communications

There *are* alternatives, however.

SPI communications has become a defacto standard

- Python supports SPI communications
- Pi-Pico supports SPI communications
- (I2C as well...)

Using these features simplifies and speeds up the code

SPI Port via Hardware

To set up a SPI port in Python, the function SPI in *machine* is used:

```
from machine import Pin, SPI

spi = SPI(1, baudrate=1000,polarity=0,phase=0,bits=8,sck=10,mosi=11,miso=12)

rxdata = spi.read(2, 0x42)
```

- *baud rate* sets the speed of the SPI communications (up to 30MHz for the LS165)
- *bits* tells you how many bits per message (8 or 16 for this example(
- *sck, mosi, miso* are the pins used for the SPI communications interface.
- *spi.read(2, 0x42)* reads in two bytes while sending out 0x42 on the MOSI line

(The MOSI line isn't used in this example - but could be used to drive a 74HC594 in the next section).

Using the hardware SPI port

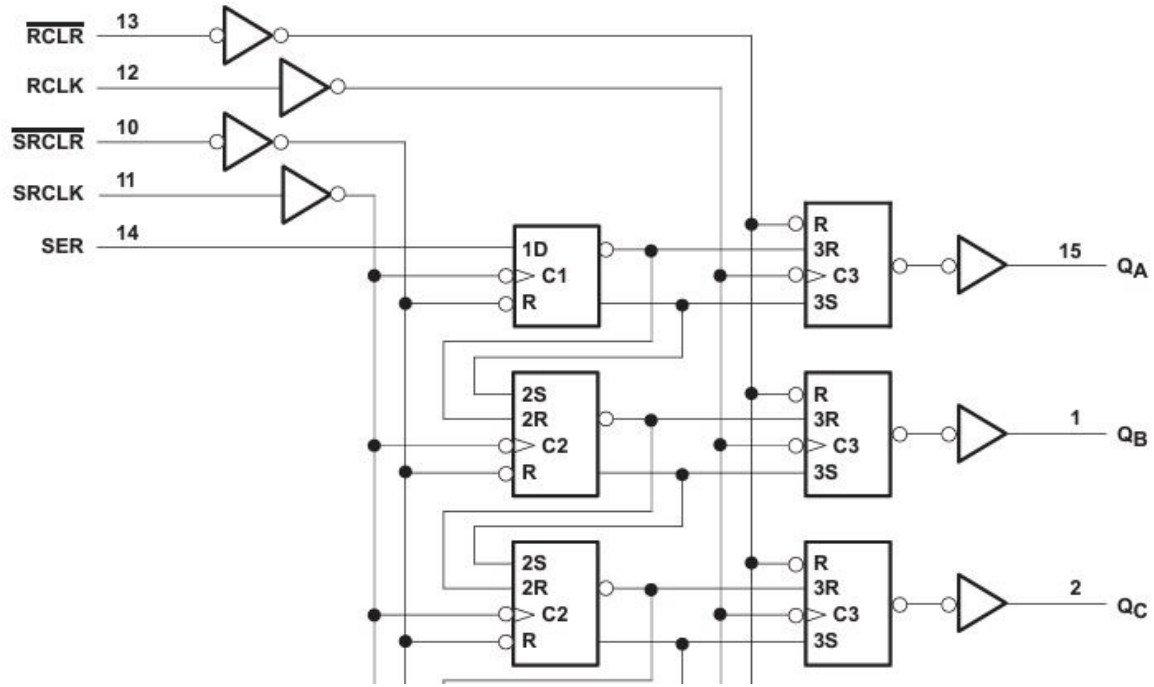
- Data sent at 10MHz
- Takes 3.6us to send 16 bits
- Plus 2us to latch high then low

SPI is faster and easier if you use the built-in SPI features

```
from machine import Pin
from time import sleep, sleep_ms, sleep_us

spi = SPI(1, baudrate=10_000_000, polarity=0,
phase=0, bits=8, sck=10, mosi=11, miso=12)

Button = Pin(20, Pin.IN, Pin.PULL_UP)
LATCH = Pin(13, Pin.OUT)

def LS165():
    LATCH.value(1)
    sleep_us(1)
    LATCH.value(0)
    sleep_us(1)
    LATCH.value(1)
    # data is latched - now shift it in
    rxdata = spi.read(2, 0x42)
    return(rxdata)


while(1):
    Y = LS165()
    print(Y)
    sleep(0.1)
```

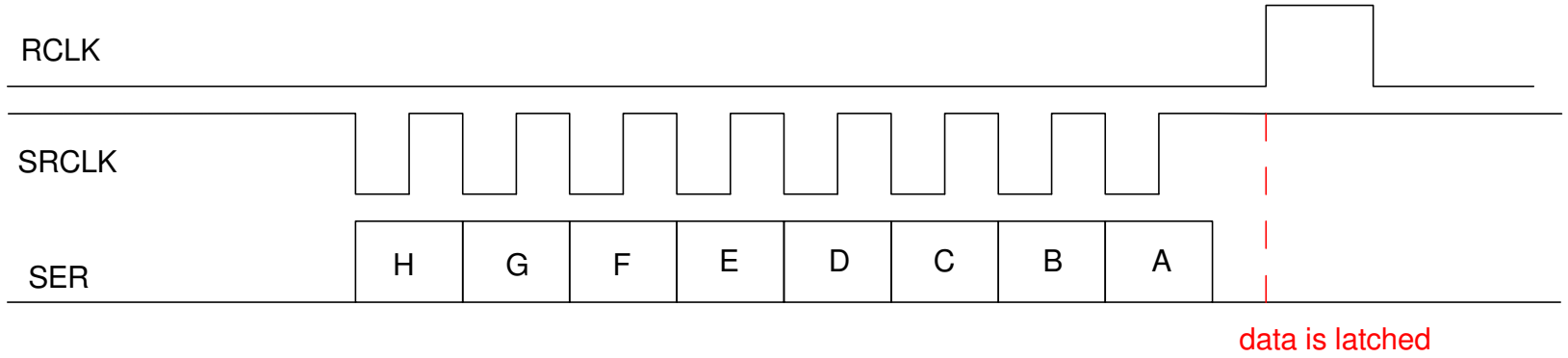
Serial Output: 74HC594

You can also do serial output with a Pi-Pico. To do so, first find a serial-in, parallel-out shift register. A 74HC594 is one such candidate.



74HC594 Serial-In, Parallel Out Shift Register

The corresponding timing diagram looks like this:

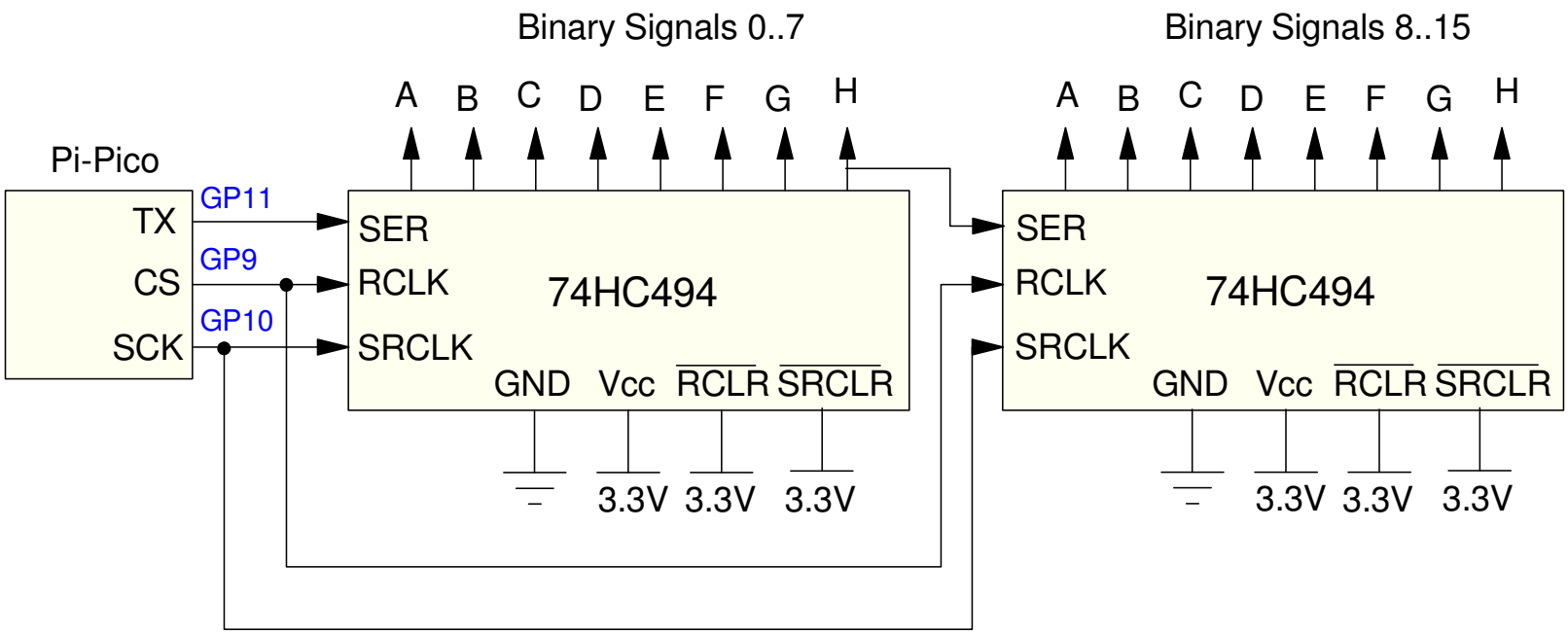


Translating....

- Start with RCLK = 0 and SRCLK = 1
- Send the first bit to the SER line (MOSI) and pulse the clock low then high
- Send the following bits to the SER line, pulsing the clock each time
- When done, pulse RCLK high then low to latch the outputs of the shift register to the outputs

At that point, the output pins are ready.

In terms of hardware, two 74HC594's could be used to output 16 bits using just 3 pins on the Pi-Pico:



Setting up 16 binary outputs using two 74HC594 series-in, parallel-out shift registgers.

Note:

- There's no limit to how many shift registers you can cascade.
- You could use 5V for the 74HC494 shift registers.
  - The 74HC494 doesn't send any signals to the Pi-Pico

74HC594 using Bit-Banging

CLK is held high & low for 1ms

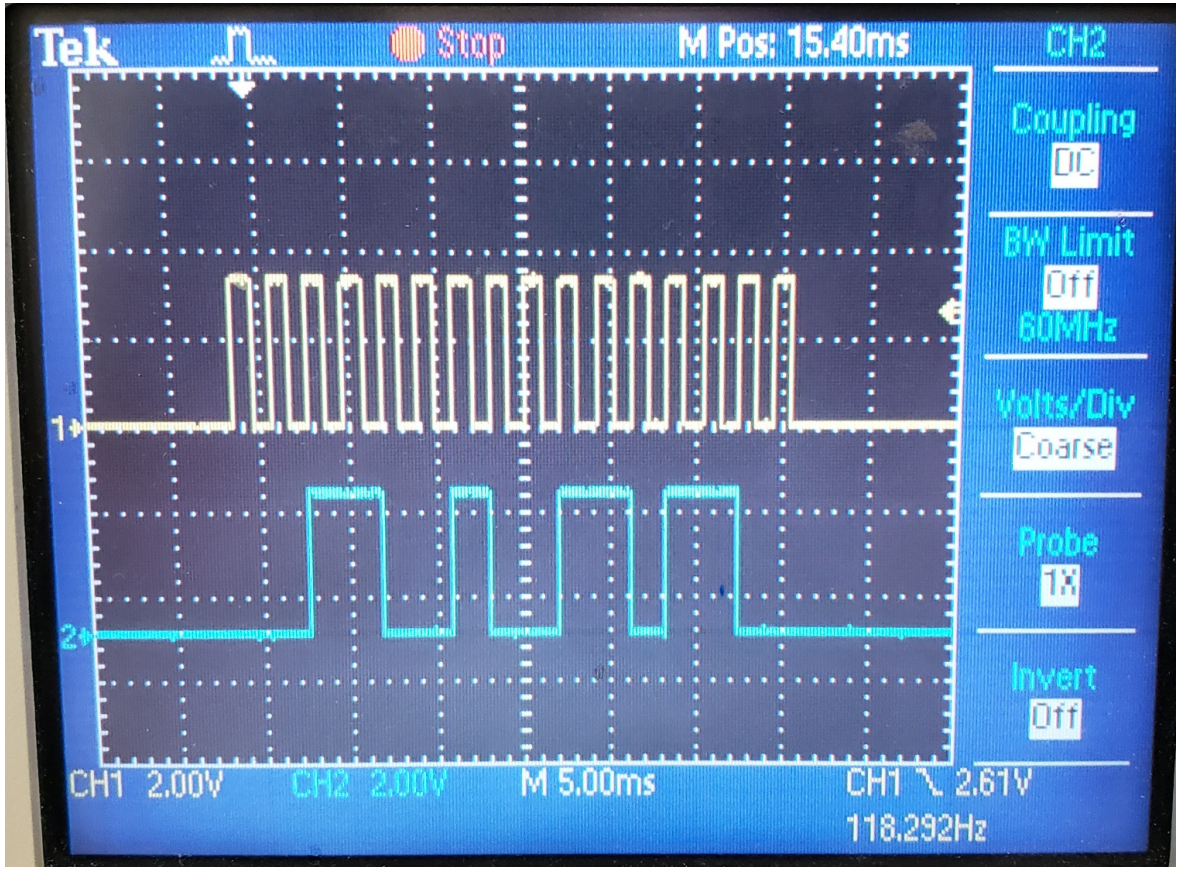
- Can be dropped to 1us

8 bits are sent out in 18.6ms

- change sleep times to 1us

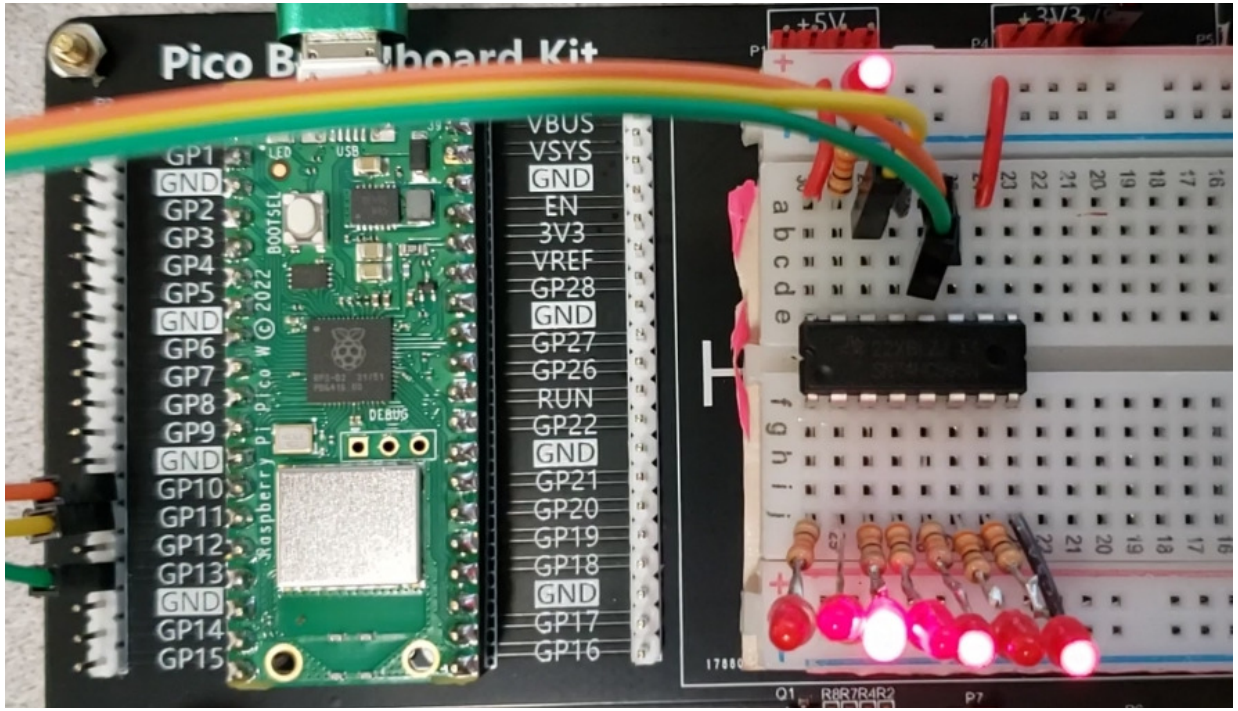
```
from machine import Pin
from time import sleep_ms, sleep_us

CLK = Pin(10, Pin.OUT)
DOUT = Pin(11, Pin.OUT)
LATCH = Pin(13, Pin.OUT)

def HC594(X):
    LATCH.value(0)
    CLK.value(1)
    sleep_ms(1)
    for i in range(0,8):
        if(X & (0x80 >> i)):
            DOUT.value(1)
        else:
            DOUT.value(0)
        CLK.value(0)
        sleep_ms(1)
        CLK.value(1)
        sleep_ms(1)
    LATCH.value(1)
    DOUT.value(0)
    sleep_ms(1)
    LATCH.value(0)

x = 0
while(1):
    x = (x + 1) & 0xFF
    HC594(x)
    sleep_ms(100)
```

If you look at the CLK and DATA lines on an oscilloscope, you can see the data being sent out:



Oscilloscope showing the CLK line (yellow) and DATA line (blue).

If you connect LEDs to the output pins of the 74HC594, you can see the data lines as well:



LEDs connected to a 74HC594 showing which pins are 1 and 0
The data is HGFE DCBA = 1010 1001

Repeat using SPI Hardware

- 140us transfer time
- vs. 18ms with bit-banging

code in blue added to measure execution time

- not needed

```
from machine import Pin, SPI
from time import sleep_ms, sleep_us, ticks_us

spi = SPI(1, baudrate=10_000_000, polarity=0,
phase=0, bits=8, sck=10, mosi=11, miso=12)
LATCH = Pin(13, Pin.OUT)

def HC594(X):
    LATCH.value(0)
    Y = bytearray([X])
    spi.write(Y)
    LATCH.value(1)
    sleep_us(1)
    LATCH.value(0)

x = 0
while(1):
    x = (x + 1) & 0xFF
    t0 = ticks_us()
    HC594(x)
    t1 = ticks_us()
    print(t1 - t0)
    sleep_ms(10)
```

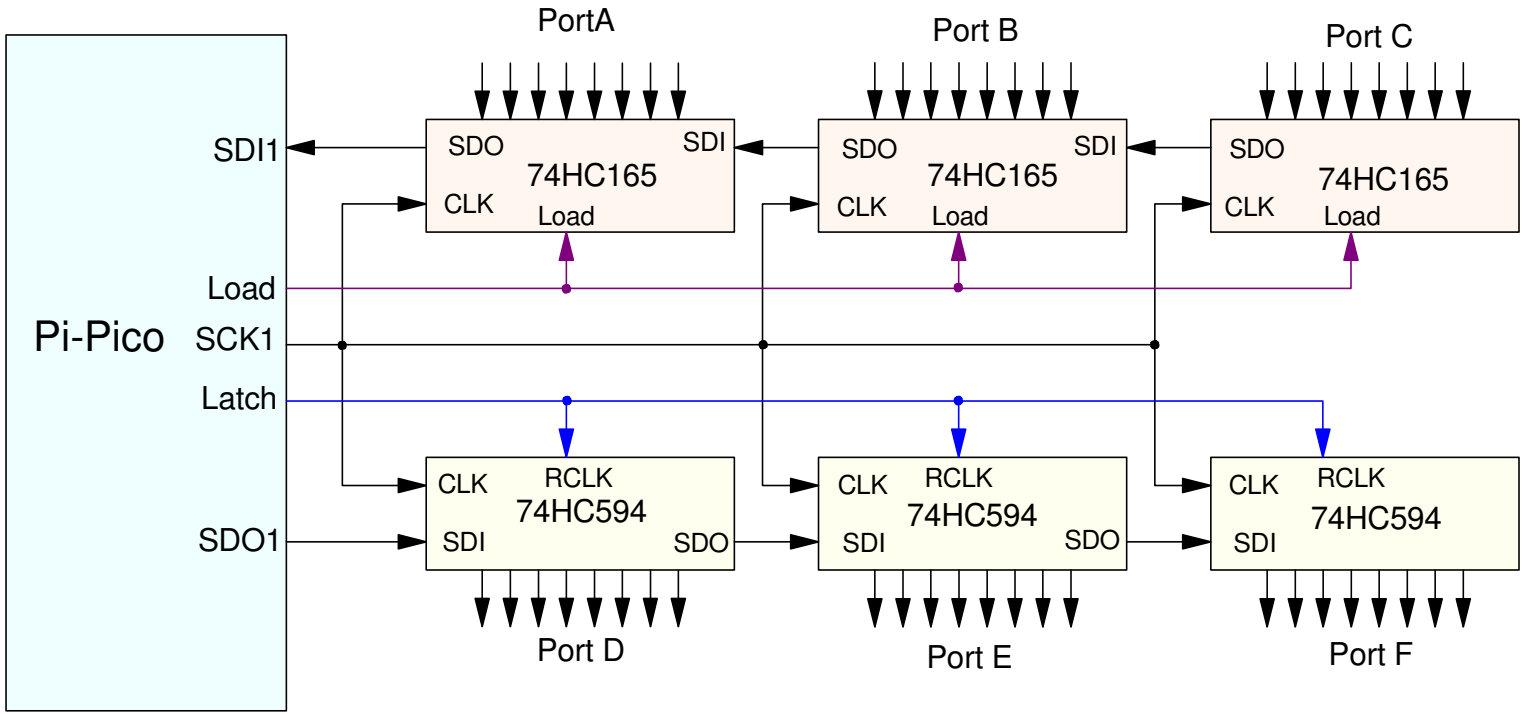
```
140
139
140
```

Why No Ports?

With serial I/O, you don't need (or want) ports

With the SPI interface and cascading IC's

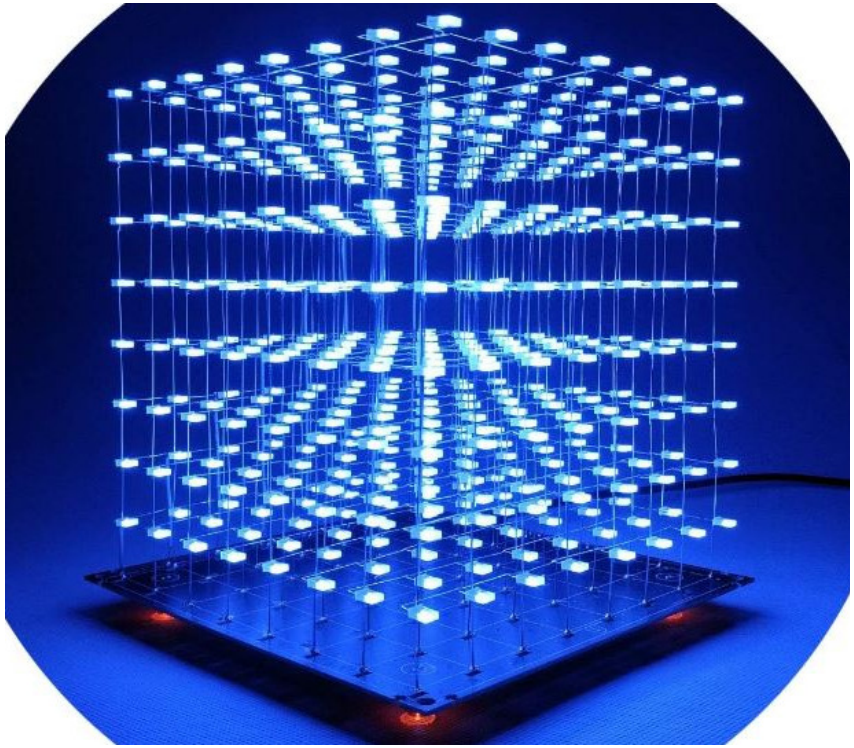
- You can read multiple ports (74HC165) and
- You can write to multiple ports (74HC594)
- All using just five pins



Fun with Series Outputs: LED Cube

8x8x8 LED Cube

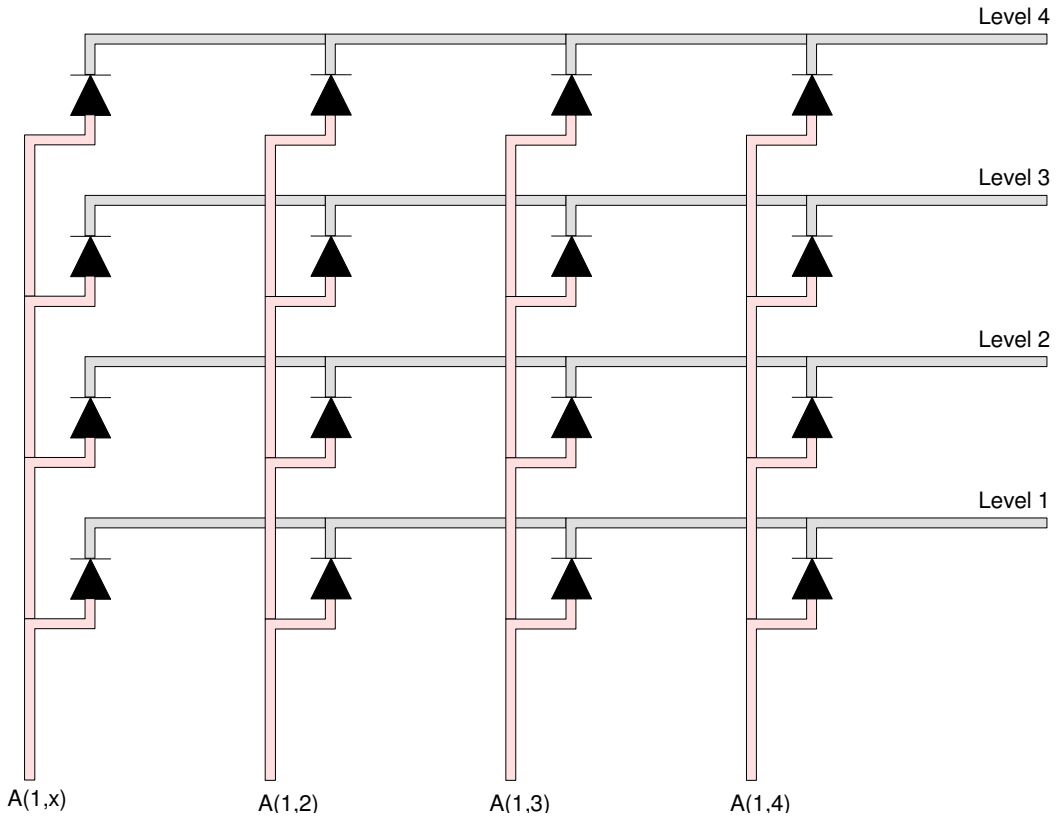
- Illustrate use of shift registers
- Control 256 LEDs with 11 output pins



4x4x4 LED Cube

Illustrate how an 8x8x8 cube works

- Start with a 4x4 LED array
- The anodes (+) connected together, going up and down
- The cathodes (-) connected together going left to right (the floor)

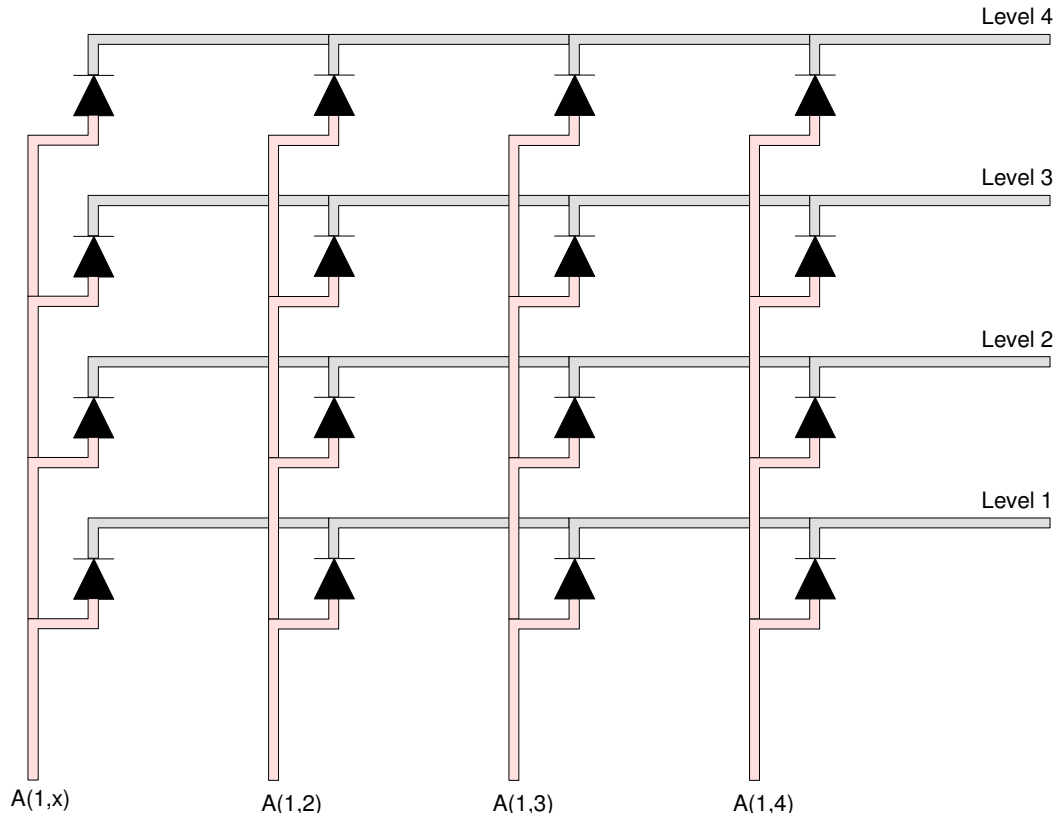


Add three more 4x4 LED arrays

- Creating a 4x4x4 cube

Short the levels together

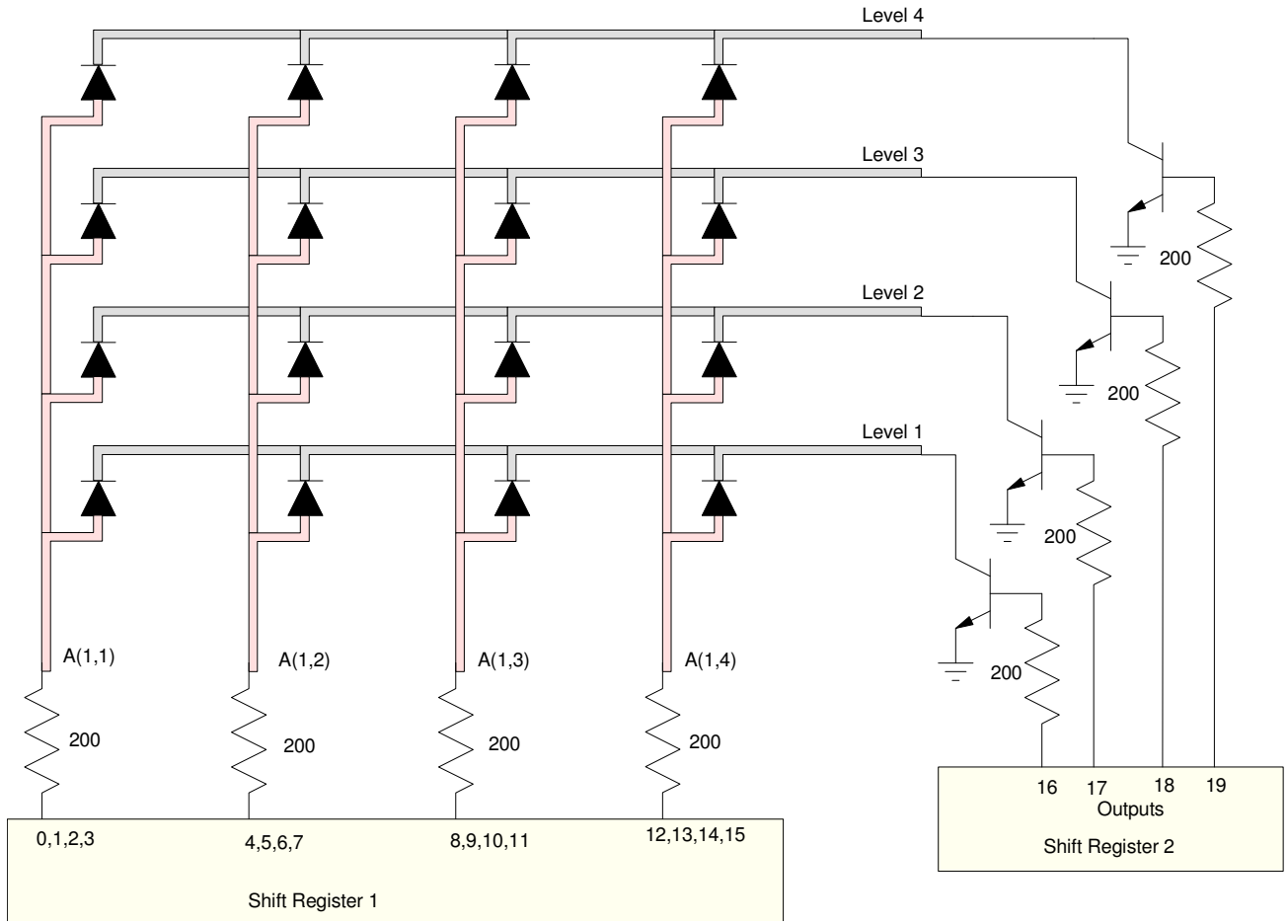
- Only one wire required to short
- More can be added for strength



Add resistors and transitors for the levels

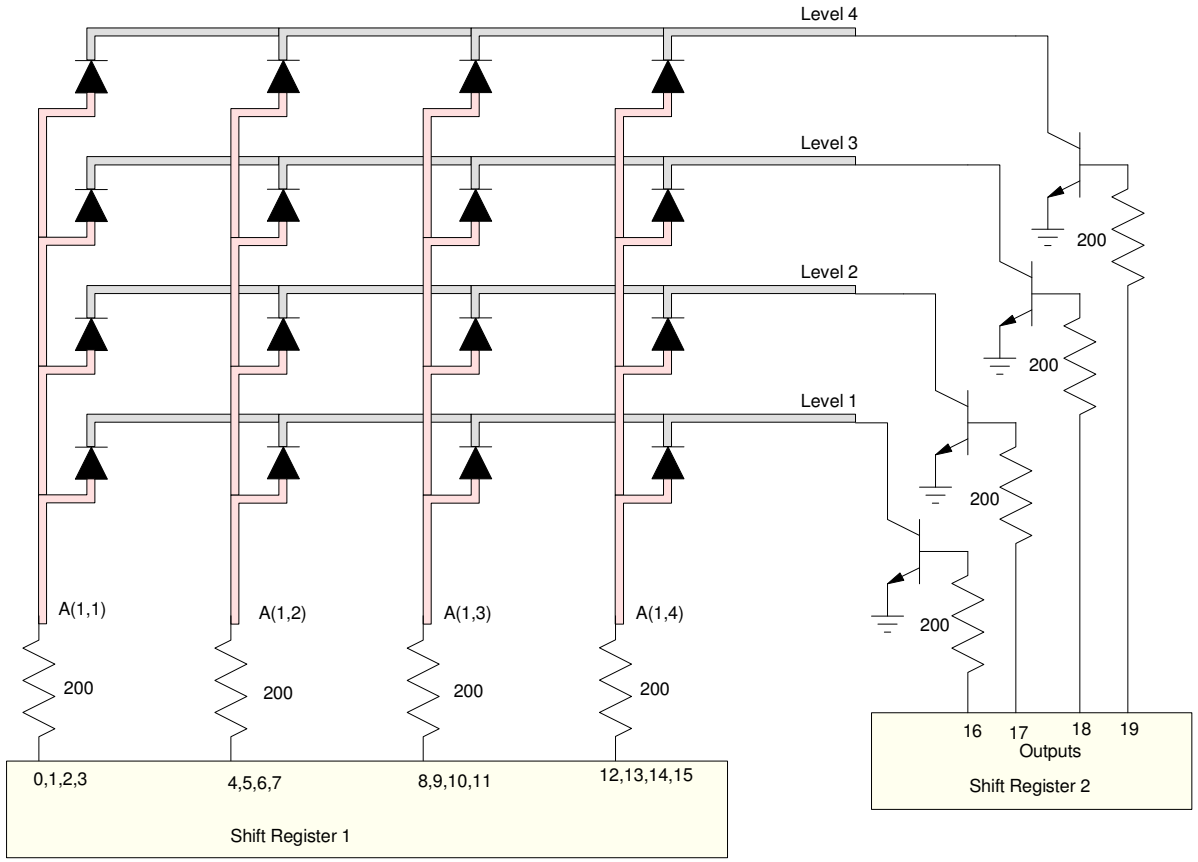
Power each element with outputs from shift registers

- 20 outputs total (three shift registers)



Software:

- Turn on Level 4 (Out19 = 1)
- Specify which LEDs on Level 4 are on
- Wait 2ms
- Repeat for each level



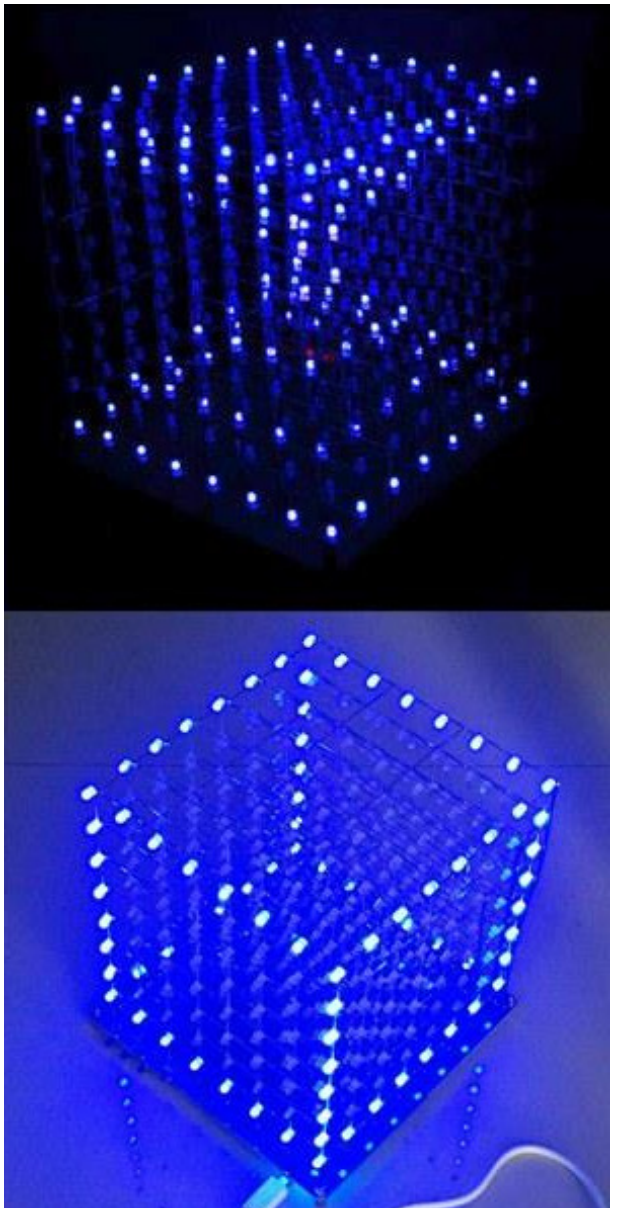
Net Result: LED Array

4x4x4 array:

- Each LED has a 25% duty cycle
- The SPI port uses 3 pins on the Pi-Pico
- The SPI port drives
  - 3 shift registers
  - 20 output pins (4x4 + 4)
  - 64 LEDs

With an 8x8x8 LED array

- Each LED has a 12.5% duty cycle
- The SPI port uses 3 pins on the Pi-Pico
- The SPI port drives
  - 3 shift registers
  - 72 output pins (8x8 + 8)
  - 256 LEDs



Summary:

Parallel I/O works

- Good if you're only driving a few things

Serial I/O really opens up possibilities

- SPI uses 3 or 4 wires
- Cascading shift registers gives an almost unlimited number of I/O pins

Driving the SPI port can be done

- Using bit-banging, or
- Using the built-in features of the Pi-Pico

The latter is 100x faster

References

Pi-Pico and MicroPython

- https://github.com/geeekpi/pico_breakboard_kit
- https://micropython.org/download/RPI_PICO/
- https://learn.pimoroni.com/article/getting-started-with-pico
- https://www.w3schools.com/python/default.asp
- https://docs.micropython.org/en/latest/pyboard/tutorial/index.html
- https://docs.micropython.org/en/latest/library/index.html
- https://www.fredscave.com/02-about.html

Pi-Pico Breadboard Kit

- https://wiki.52pi.com/index.php?title=EP-0172

Other

- https://docs.sunfounder.com/projects/sensorkit-v2-pi/en/latest/
- https://electrocredible.com/raspberry-pi-pico-external-interrupts-button-micropython/
- https://peppe8o.com/adding-external-modules-to-micropython-with-raspberry-pi-pico/
- https://randomnerdtutorials.com/projects-raspberry-pi-pico/
- https://randomnerdtutorials.com/projects-esp32-esp8266-micropython/