

Design of Digital Circuits and Systems Communication

Instructor: Justin Hsia

Teaching Assistants:

Colton Harris

Deepti Anoop

Gayathri Vadhyan

Jared Yoder

Lancelot Wathieu

Matthew Hung

Relevant Course Information

- ❖ Homework 5 due tomorrow

- ❖ Lab 5 due next week (5/17)
 - Hardest/longest lab
 - You will need to use the VGA interface on LabsLand

- ❖ Quiz 4 next Thursday (5/16)
 - Algorithm to Hardware – ASMD and datapath drawing
 - 40 minutes

Disclaimers

- ❖ This topic (communication) won't be used on any assignment this quarter
- ❖ It encompasses both FPGAs and computers and some analog considerations, but is definitely relevant to digital design
 - Could test/build communication drivers on FPGAs, but often included in microcontrollers
 - For more, take EE/CSE474: Embedded Systems

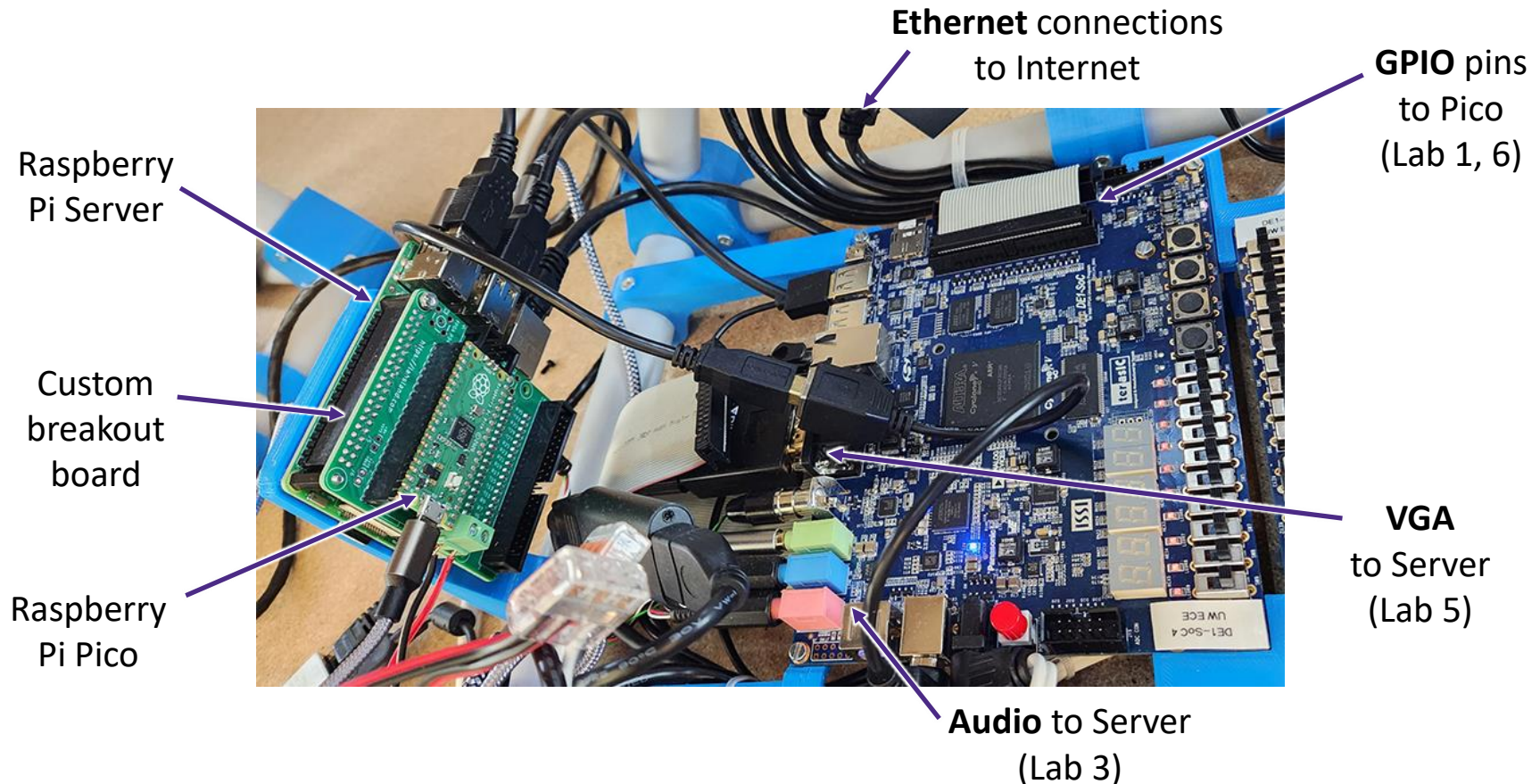
Communication

- ❖ “The transmission, reception, and processing of information between two or more locations with the use of electronic circuits.”
 - Includes a lot more than what we’ve discussed so far
 - Want this to be *general* so you don’t have to build custom circuits (like the CDC ones we saw) for every use
 - However, many communication schemes are/were created for specific applications

- ❖ The goal for today is to introduce you to communication considerations via examples

Aside: LabsLand

- ❖ Made possible by communication between the DE1-SoC, a Raspberry Pi Pico, and a Raspberry Pi Server:



Communication Considerations

- ❖ **Bandwidth:** number of wires and what mix of serial/parallel
- ❖ **Speed:** bits/bytes/words per second
 - Baud (Bd) is the unit for symbol changes per second
- ❖ **Timing methodology:** synchronous or asynchronous
- ❖ **Number of devices:** sources and destinations
 - **Arbitration scheme:** daisy-chain, centralized, distributed
- ❖ **Protocols:** provide some guarantees as to correctness, may include error detection or correction

Serial vs. Parallel Communication

- ❖ **Serial communication** involves sending *data* over a single wire, separated in time
 - Often includes other wires for *control signals*
- ❖ **Parallel communication** involves sending multiple bits of data simultaneously over multiple wires
- ❖ Discuss with your neighbor(s):
 - Which type do you think is more prevalent in computer systems and why (pros/cons)?

Serial vs. Parallel Communication

- ❖ **Serial** is actually more common in modern systems!
 - **Fewer wires** required → less costly, less power, less space, no clock skew

Parallel ATA



Serial ATA

- High switching speeds leads to **crosstalk in parallel data bits**
→ serial can do longer distances and higher transmission speeds (*i.e.*, clock rate, not necessarily data rate)
 - However, serial requires more **processing** to convert between serial and parallel form
- ❖ **Parallel** found within ICs and computer systems (system, memory, and hard drive busses) or specialized devices (*e.g.*, older printer ports)

Timing Methodology

- ❖ Analogous to our clock domain crossing discussion
- ❖ **Synchronous:** clock signal is sent along one of the communication wires
 - Recall: *known* relationship between clock signals
- ❖ **Asynchronous:** clock signal is not sent
 - Recall: *unknown* relationship between clock signals
 - Open-loop solutions typically involves oversampling by the receiver
 - Closed-loop solutions include the various synchronizers previously discussed

Number of Devices

- ❖ Single source – single destination
 - Easy and cheap (point-to-point, no tri-stating necessary)
- ❖ Single source – multiple destination
 - Physical fanout limitations
 - Need an addressing scheme to direct data to particular destination
- ❖ Multiple source – multiple destination
 - With multiple potential drivers, need tri-stating as well as collision detection
 - Fairness considerations (*e.g.*, priority scheme, arbitration between senders)

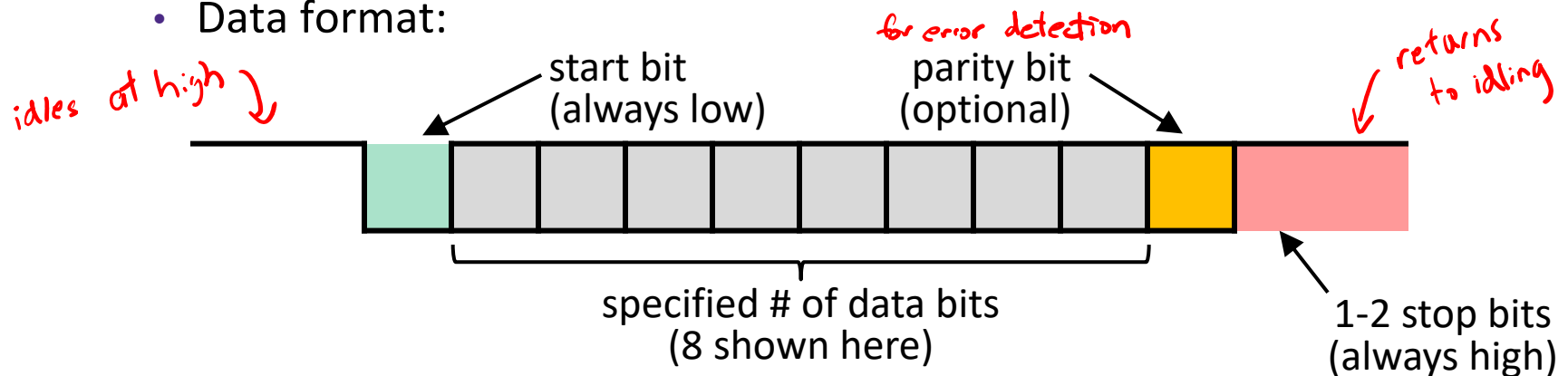
Some Serial Communication Schemes

- ❖ UART
 - Usually asynchronous, point-to-point
- ❖ SPI
 - Synchronous, single source – single destination
- ❖ I2C
 - Synchronous, single source – multiple destination
- ~~❖ USB~~
- ~~❖ Ethernet~~
 - Multiple source – multiple destination

UART

❖ Universal Aynchronous Receiver-Transmitter

- Hardware device that implements this asynchronous serial communication *interface* (electrical details missing)
 - UART is usually part of a microcontroller chip alongside an external driver circuit that converts the UART output to a specific standard (e.g., RS-232)
- Configurable *serial frame*:
 - Transmission speed (e.g., 9600 baud)
 - Data format:



DE1-SoC UART Receiver Design

❖ Design notes:

- Data: input is serial, output is parallel (assume 8 bits/frame)
- DE1 running on CLOCK_50 but UART receiver must account for variable baud rate
- Want to sample in *middle* of data bit

❖ Design questions (datapath):

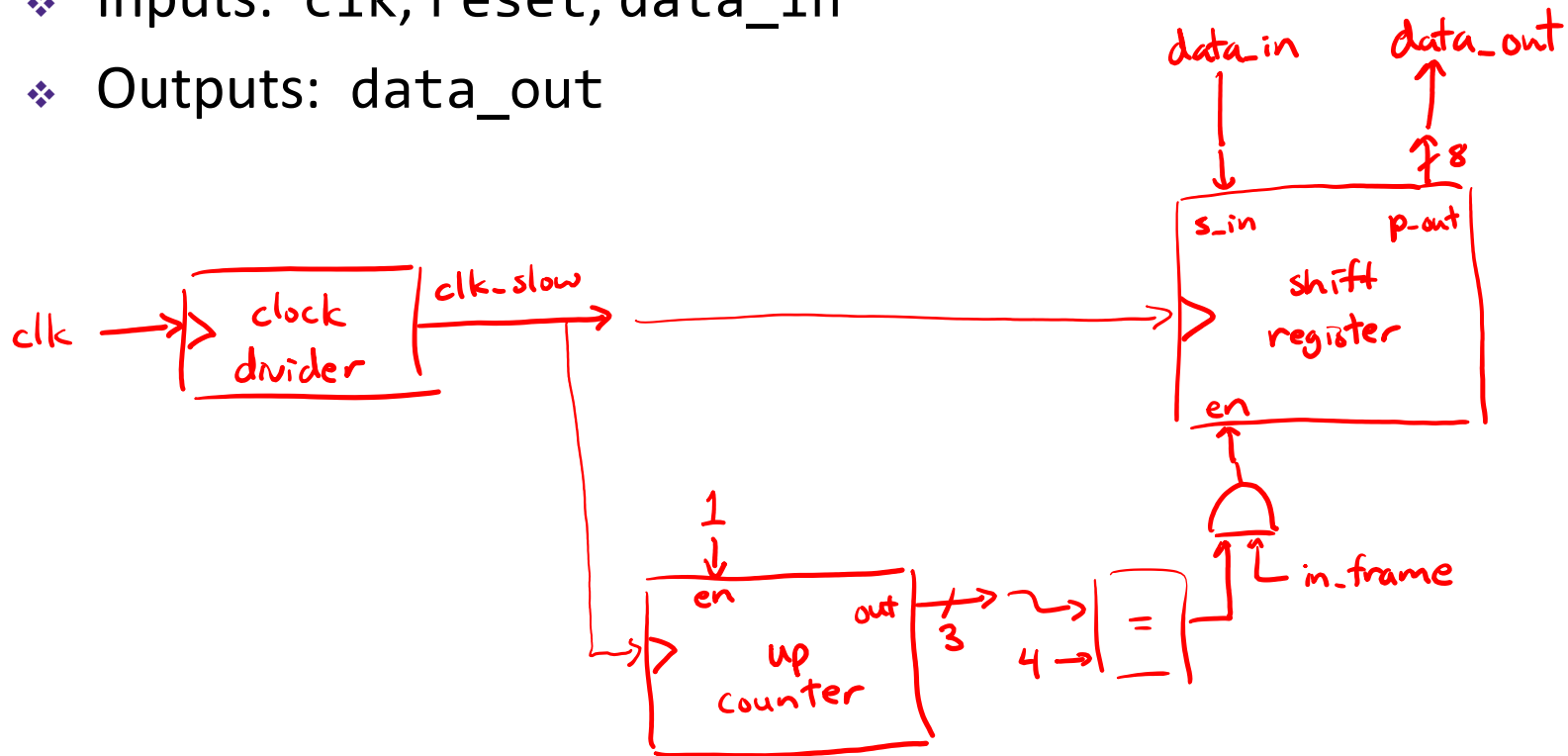
- What digital component will help us *convert from serial to parallel*? *a shift register*
- If the desired transmission rate is X Hz, which digital components can help us decide when to sample?

① use clock_divider to get a second clock of approximately $8-16X$ Hz

② use a counter to decide which clock cycle to sample after start bit received

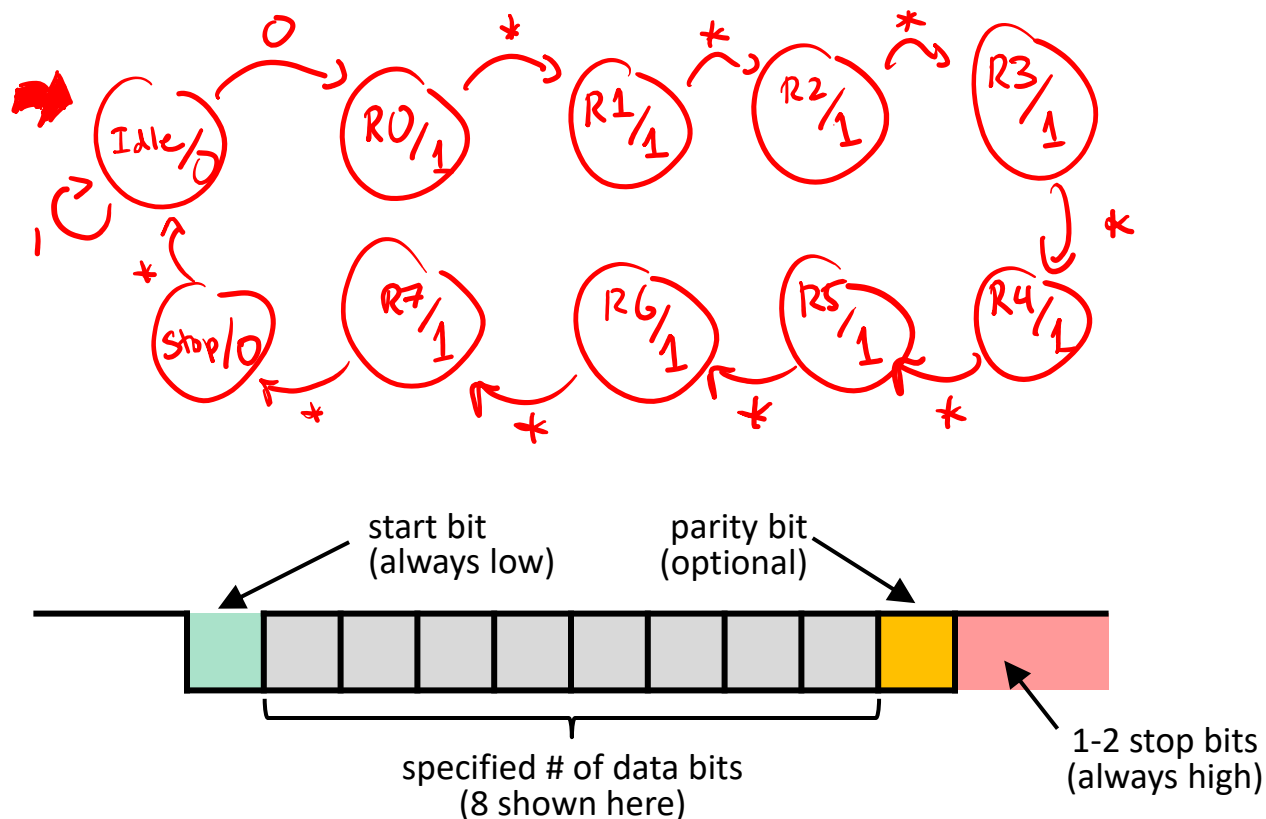
DE1-SoC UART Receiver Block Diagram

- ❖ Assume we are oversampling by 8x
- ❖ Inputs: clk, reset, data_in
- ❖ Outputs: data_out



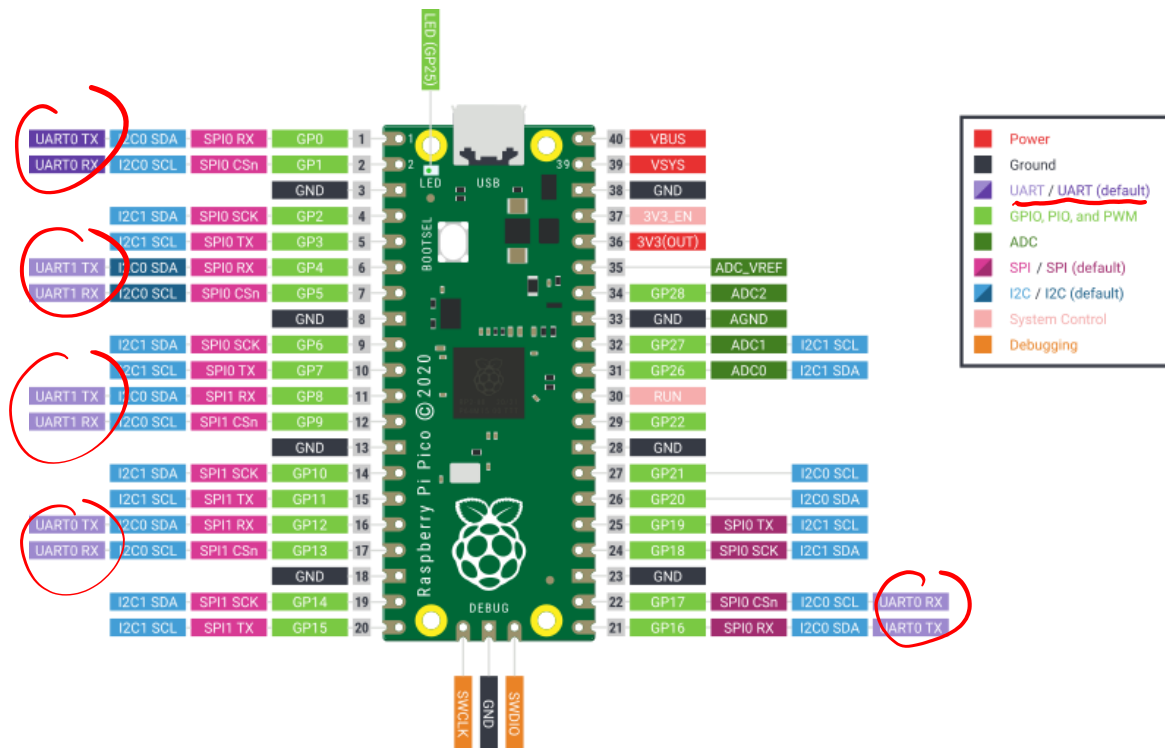
DE1-SoC UART Receiver State Diagram

- ❖ Assume 8 data bits, no parity bit, 1 stop bit
- ❖ Inputs: reset, sampled_bit
- ❖ Outputs: in_frame



Aside: LabsLand UART

- ❖ In Lab 6, you have the option of using two serial UART peripherals in LabsLand: N8 controller and joystick
 - Uses provided `serial_driver.sv` on one of the GPIO pins to talk to the Pico



serial_driver.sv Notes

- ❖ SPEED is related to baud/transmission rate
 - count/counter form the slow clock
 - Constantly receiving data – arbitrary wait before next read
- ❖ MAX_STEPS accounts for serial frame configuration
 - latch triggers request to Pico
 - pulse, latch are used to sample/shift
 - save is when save data (end of frame)

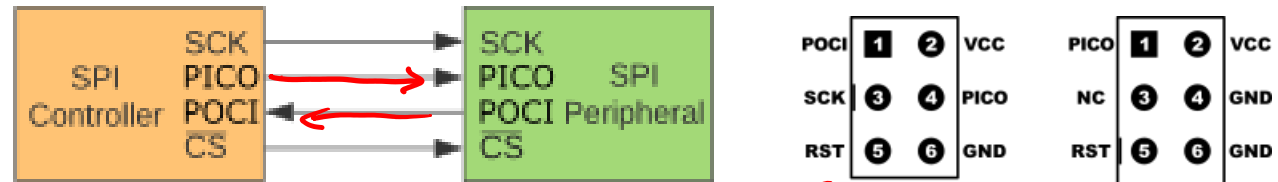
count	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
latch	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
pulse	0	0	0	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0
save	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1

Technology Break

Communication Terminology Note

- ❖ The following communication schemes have historically used the term “master” to describe a device that controls one or more “slaves”
 - This can be a problematic metaphor, particularly in the context of historical race relations in the United States
 - It can also be an inaccurate metaphor, as often the “master” device does not actually have a real control relationship over the “slave” device(s) or they may be swappable
 - More info: <https://muse.jhu.edu/article/215390/>
- ❖ Here, we will use proposed replacement names
 - Not uniformly accepted – there is plenty of current debate around naming

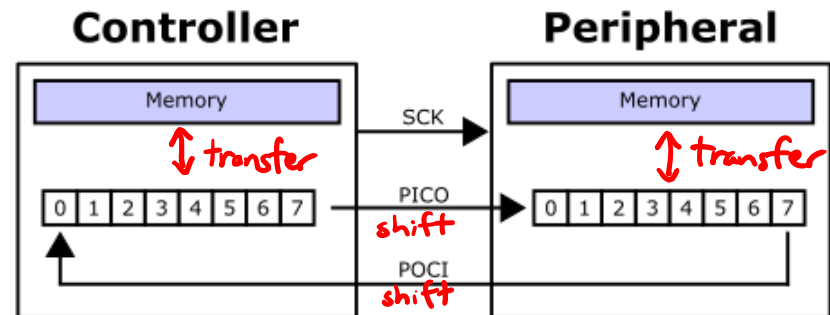
SPI



possible connector pin outs

Serial Peripheral Interface

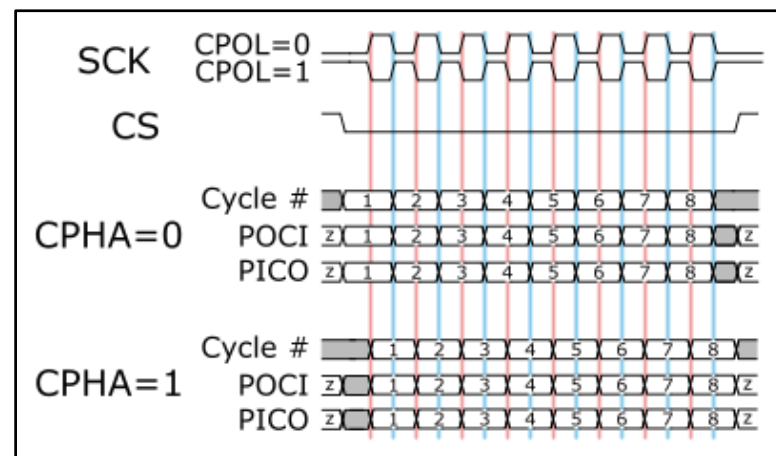
- A synchronous serial interface between one *controller* and one (of possibly many) *peripherals*
 - Requires ≥ 4 wires (as opposed to UART's 2)
 - SCK = serial clock, PICO = peripheral in/controller out, POCI = peripheral out/controller in, \overline{CS} = chip select (active low)
- Full duplex data transfer initiated by controller on a negative edge of the peripheral's chip select
 - Occurs one *word* (e.g., 8 bits) at a time, even if one party isn't sending that much



basically a 2-device ring buffer

SPI Details

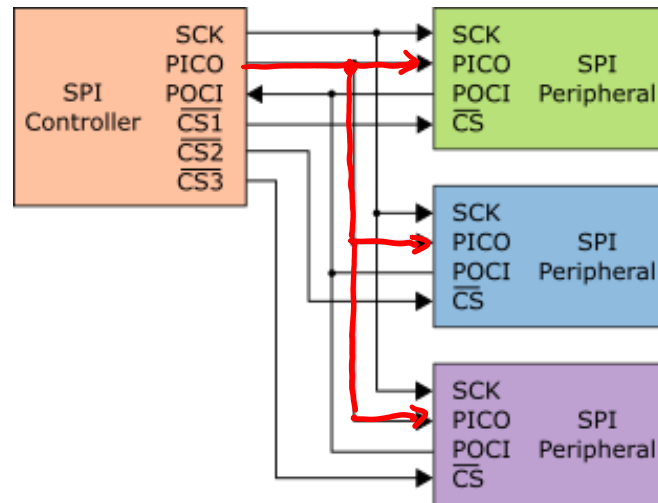
- ❖ We don't need to oversample values, but still want to read them in the middle
 - Defined modes based on **clock polarity (CPOL)** and **clock phase (CPHA)**
 - CPOL determines the "idling" state of the clock and which edges are considered *leading* vs. *trailing*
 - (0 or 1)
 - (idle → active)
 - (active → idle)
 - CPHA determines which edges are for data changes and which edges are for data capture
 - (leading or trailing)
 - (trailing or leading)
 - This means that we can typically run SPI at a faster rate than UART! *device clock doesn't have to account for oversampling the serial clock*



SPI Details

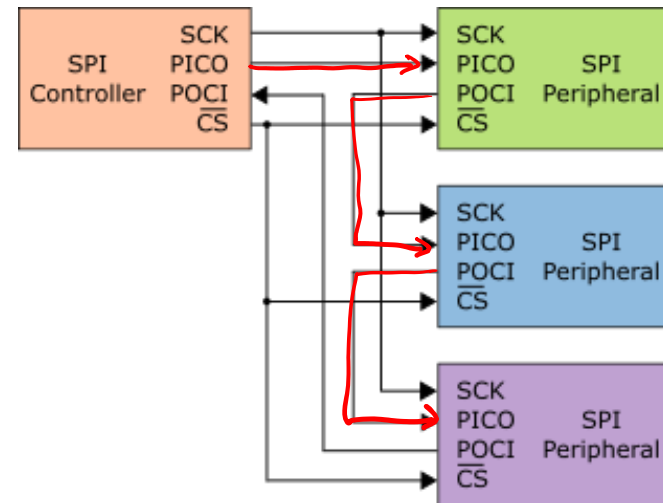
- ❖ Not great for communicating with multiple peripherals:

Independent Selection



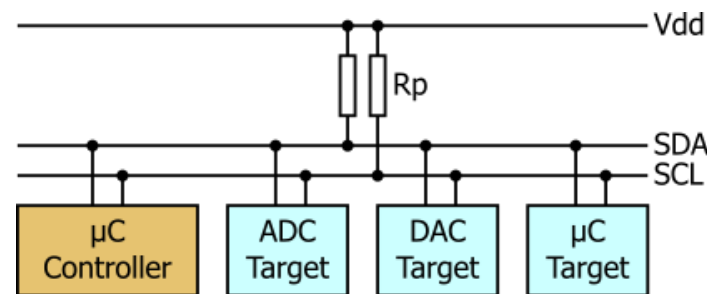
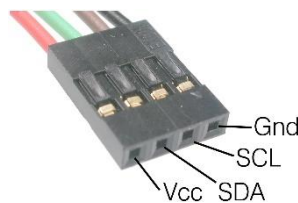
- Requires one chip select per peripheral *just for controller*
- Requires tri-stating *so no contention*
- Non-selected must ignore PICO and POCI signals

Daisy Chain



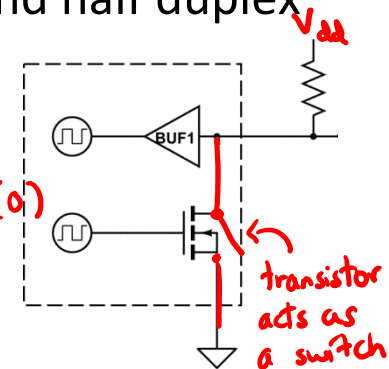
- Single chip select line needed
- Whole chain acts as a communication shift register
- A less common configuration

I²C/I2C



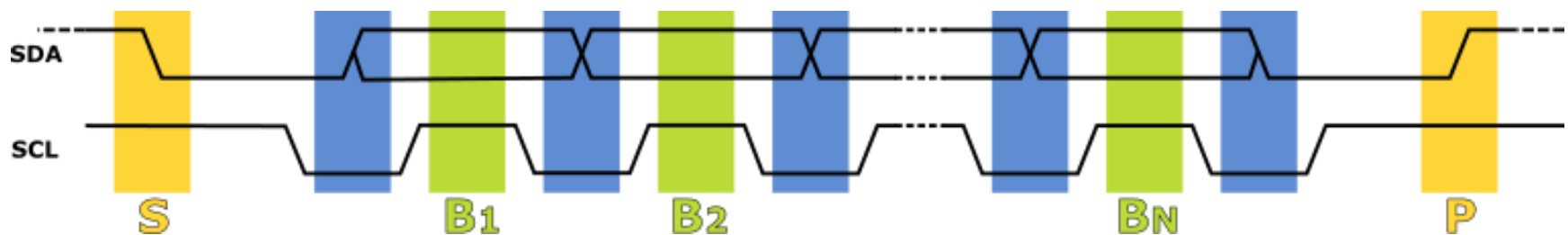
❖ Inter-Integrated Circuit Bus

- A synchronous serial interface between one *controller* and many *targets*
 - Only requires 2 lines (same as UART), but bidirectional and half duplex
 - SDA = serial data line, SCL = serial clock line
 - Pull-up resistors create “open drain” bidirectional I/O: when open, leave line floating at V_{DD} (1); when closed, drag line to GND (0)
- Can be used with many types of devices/targets
 - e.g., μ controllers, sensors, ADCs/DACs, memory modules, LCD drivers, real-time clocks
- Communications always initiated by controller, but both controller and target can send and receive data on SDA
 - More difficult to manage (more on this next)
 - Communications include acknowledgement ✓



I²C/I2C Details

- ❖ Communications bookended by special (2-wire) START and STOP signals: *(yellow regions)*
 - While SCL is high: START is SDA to low; STOP is SDA to high
 - Other “START” signals are ignored until STOP is seen
- ❖ SDA values changed on leading edges of clock and read on trailing edges of clock *(blue regions)* and *(green regions)* (like SPI)
- ❖ Every N bits (usually 8) is followed by an ACK (0) or NACK (1) sent by the *receiver* *controller opens switch target closes switch to acknowledge*
 - Communications on SDA switches directions for this bit



I²C/I2C Details

- ❖ Everything is listening on the same SDA, so how to differentiate?
 - ^(assuming N=8) First byte contains **7-bit address** followed by R/W bit; targets with different addresses will ignore this communication
 - 16 reserved addresses, so max of 112 targets
 - Actually possible to have multiple controllers on the same SDA line, so need an arbitration scheme in case multiple controllers start a communication at the same time
 - First controller to notice ground when trying to send a “1” stops
- ❖ Limited range because of bus capacitance and need for a common ground potential

reading or writing request?

Review Questions

❖ Which serial communication scheme(s):

- Has the most limited communication range?

UART

SPI

I²C

- Is not fully duplex?

UART

TX → RX
RX ← TX

SPI

multi-device
ring buffer

I²C

half duplex: sharing same SDA line

- Works best with multiple destinations?

UART

cannot

SPI

can, but clunky

I²C

addressing!

- Can have the fastest transmission rate?

1-2 stop bits

UART

SPI

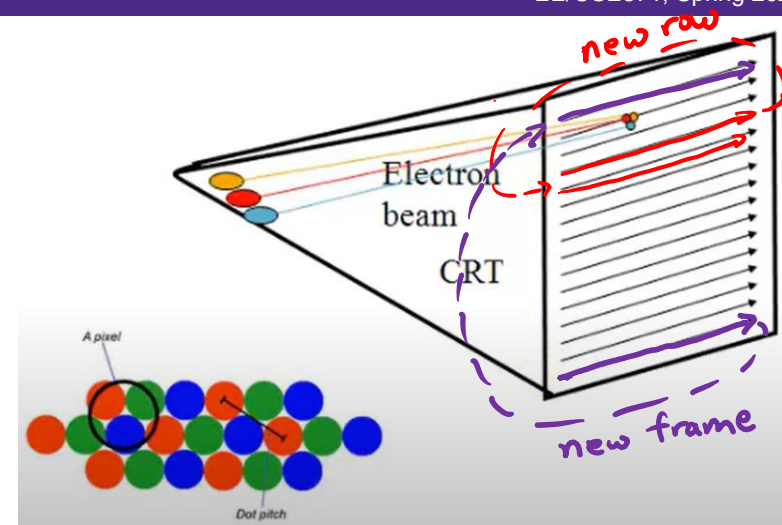
I²C

synchronous = no oversampling
less overhead & arbitration in SPI

VGA

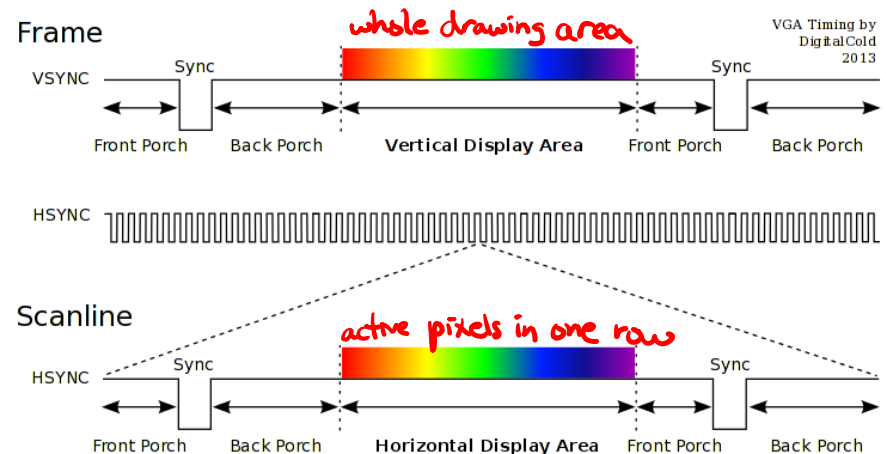
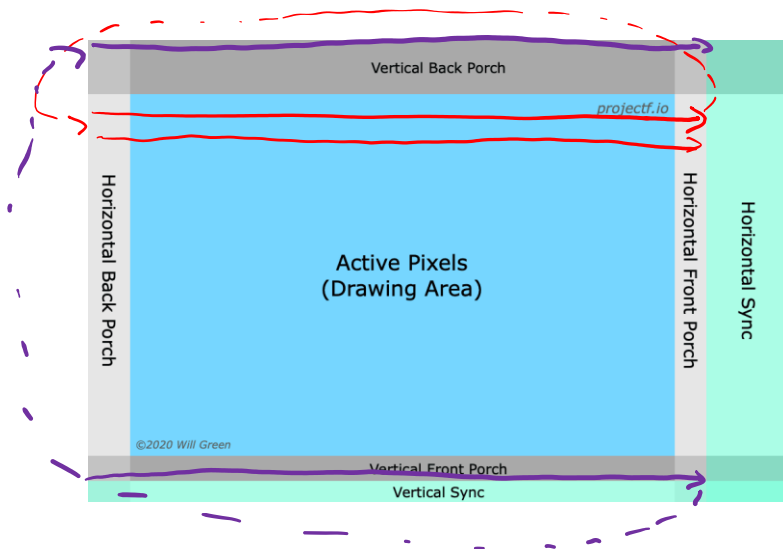
❖ Video Graphics Array

- Developed for cathode ray tube (CRT) displays, which scan across the monitor left-to-right and top-to-bottom
 - Data sent sequentially pixel-by-pixel, but this is **parallel communication** because each pixel contains red, green, and blue (RGB) data simultaneously
 - The electron guns need to reposition at the end of each row and frame
- Is a very confusing graphics standard, as it allows for different resolution, color, and timing specifications (with varying amounts of forgiveness on different monitors)



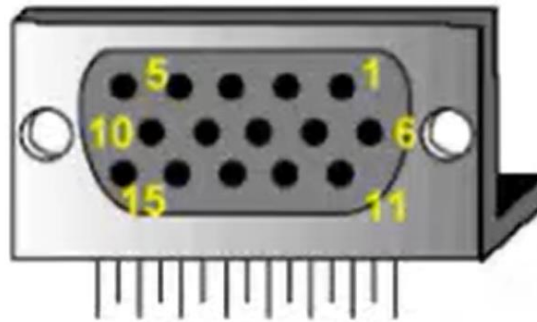
VGA Details

- ❖ The screen is not what you think!
 - Up to 640 x 480 pixels in drawing area, but surrounded by front and back porches
 - Pixel clock cycles through different pixel data during drawing area
 - ^{HSYNC} Horizontal and ^{VSYNC} vertical sync signals pulse to trigger reset at end of row and frame, respectively



VGA Details

- ❖ The pinout is analog:



Pin 1: Red	Pin 5: GND
Pin 2: Grn	Pin 6: Red GND
Pin 3: Blue	Pin 7: Grn GND
Pin 13: HS	Pin 8: Blu GND
Pin 14: VS	Pin 10: Sync GND

- Lines up mostly with the outputs of the VGA_framebuffer module:

* Outputs:	
* VGA_R	- Red data of the VGA connection
* VGA_G	- Green data of the VGA connection
* VGA_B	- Blue data of the VGA connection
* VGA_CLK	- VGA's clock signal
* VGA_HS	- Horizontal Sync of the VGA connection
* VGA_VS	- Vertical Sync of the VGA connection
* VGA_BLANK_n	- Blanking interval of the VGA connection
* VGA_SYNC_n	- Enable signal for the sync of the VGA connection