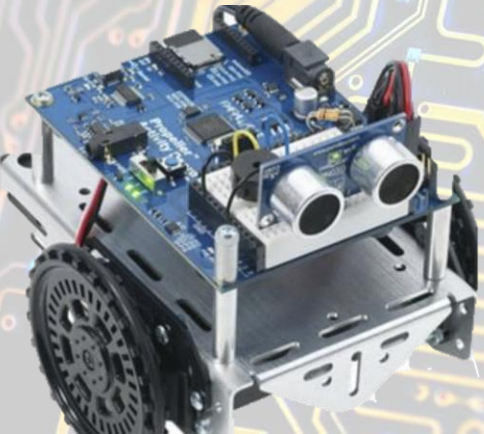
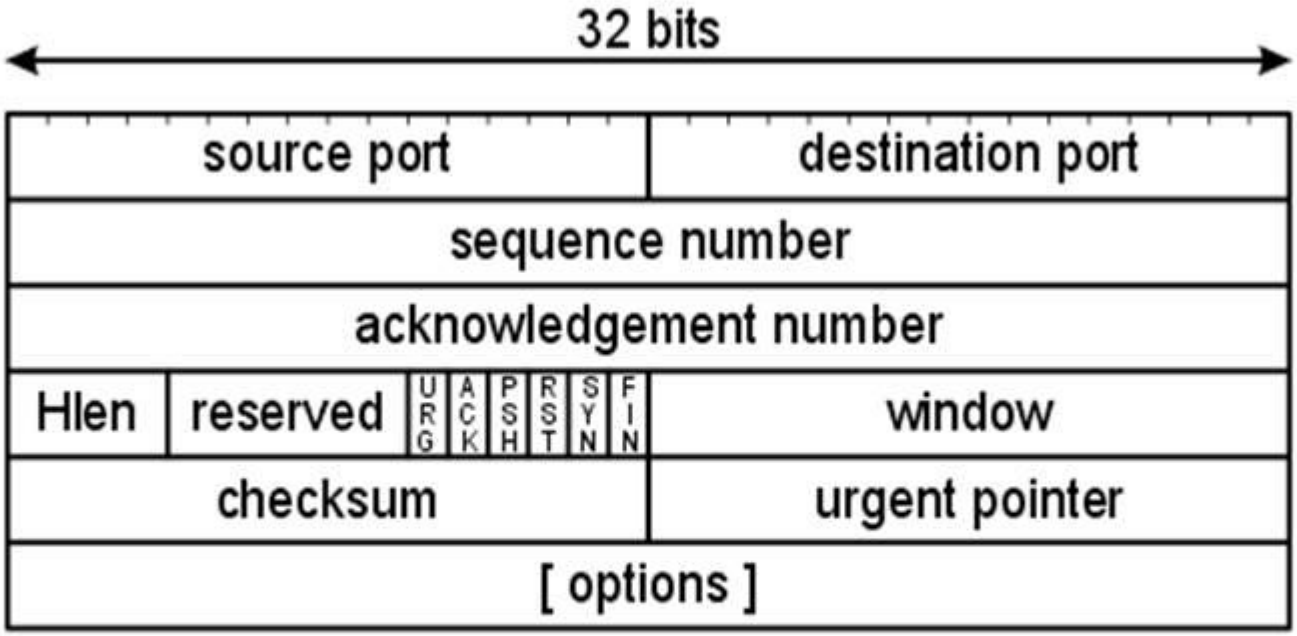
EC535 Introduction to Embedded Systems



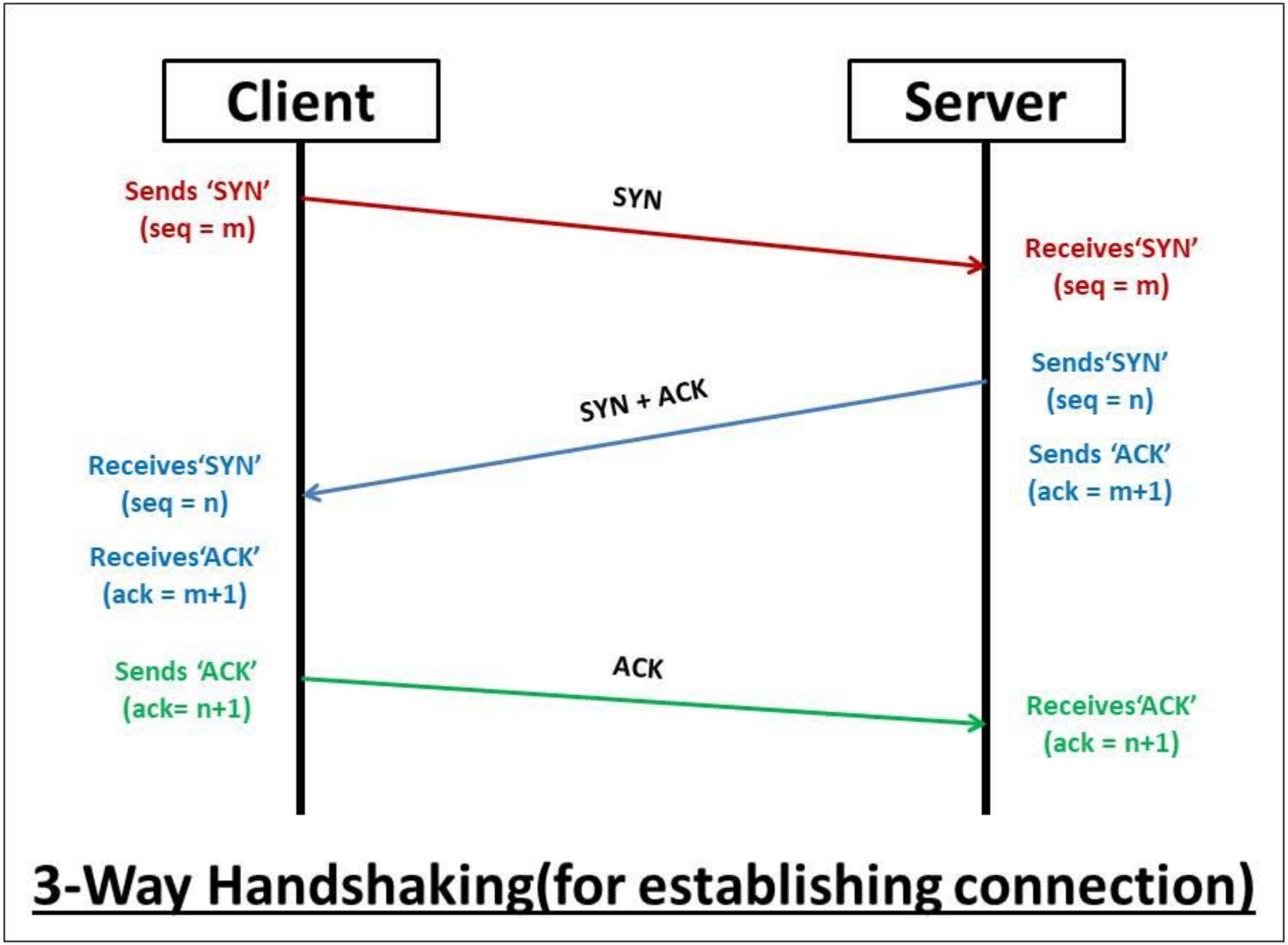
Extra Sensors

Name	Count
Cameras	2
eZ430 Smartwatch	2
Thermocouples	1
Motion Sensors	5
Dual Ball Bearing	4
Fingerprint Reader	2
Wireless router	1
Temperature and Humidity Sensor Module (Gowoops)	5
Bluetooth Headphones	1
ultrasonic distance sensor (ElecRight HC-SR04)	5

https://rb.gy/e25ejg

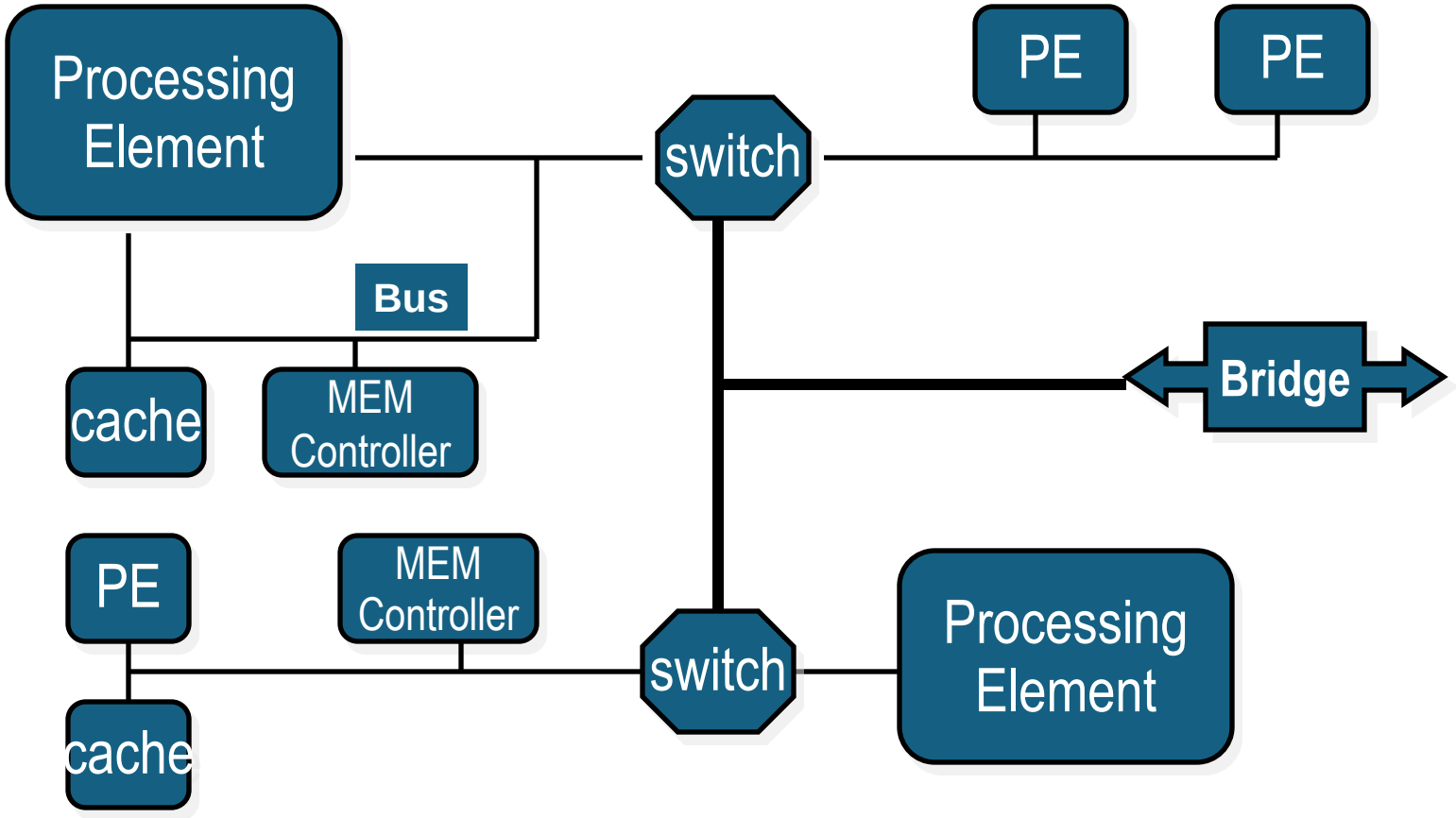

32 bits
source port
destination port
sequence number
acknowledgement number
Hlen
reserved
URG
ACK
PSH
RST
SYN
FIN
window
checksum
urgent pointer
[options]

TCP Header



3-Way Handshaking(for establishing connection)

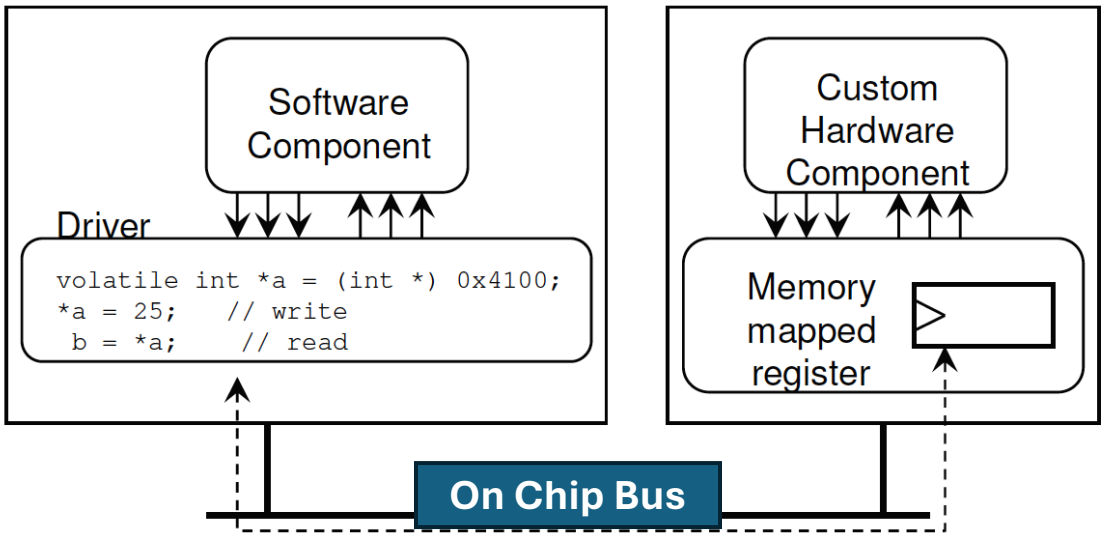
System Elements



PEs may be CPUs or ASICs.

Memory Mapped Interfaces

- Memory Mapped Register
  - Communicating hardware & software components
  - Register at the interface of the hardware component read/written from the software component
  - Attached to the on-chip bus

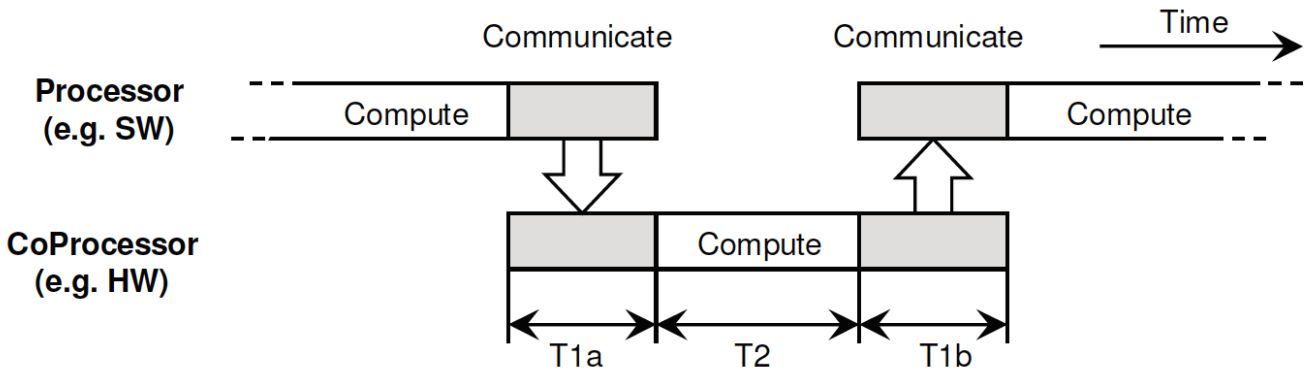



```
int *status = (int *) 0x4000;

while (*status == 0); // wait for flag to

Compiler may do:
if (*status == 0) {
  while (true); // infinite loop!
}
```

```
volatile int * a = (int *) address_to_access;
```

Tightly vs Loosely Coupled Design

- Computation to communication ratio affects the design choice



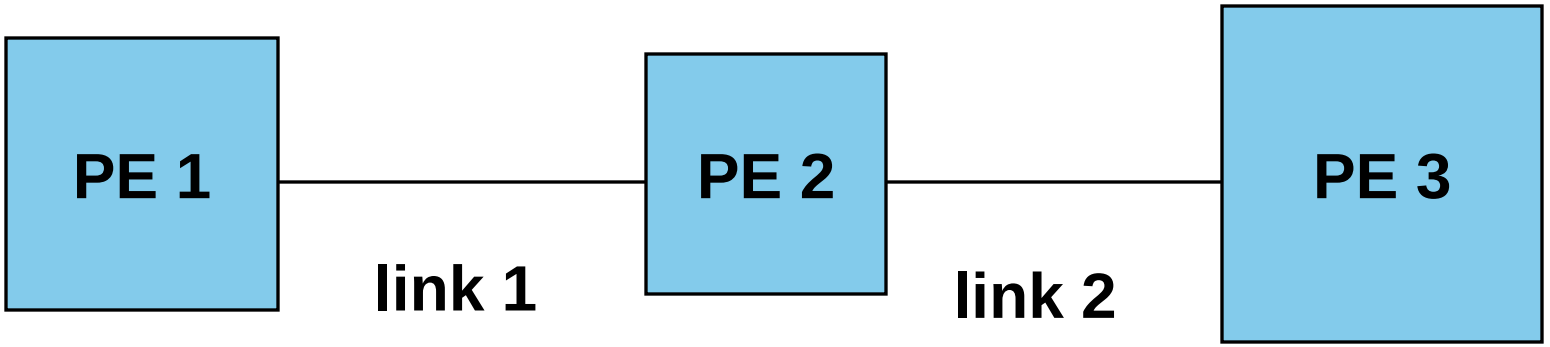
$T_{compute} = T2$

$T_{communication} = T1a + T1b$

- If $T_{compute} / T_{communication} >> 1$ → Loosely coupled design
- If $T_{compute} / T_{communication} <= 1$ → Tightly coupled design

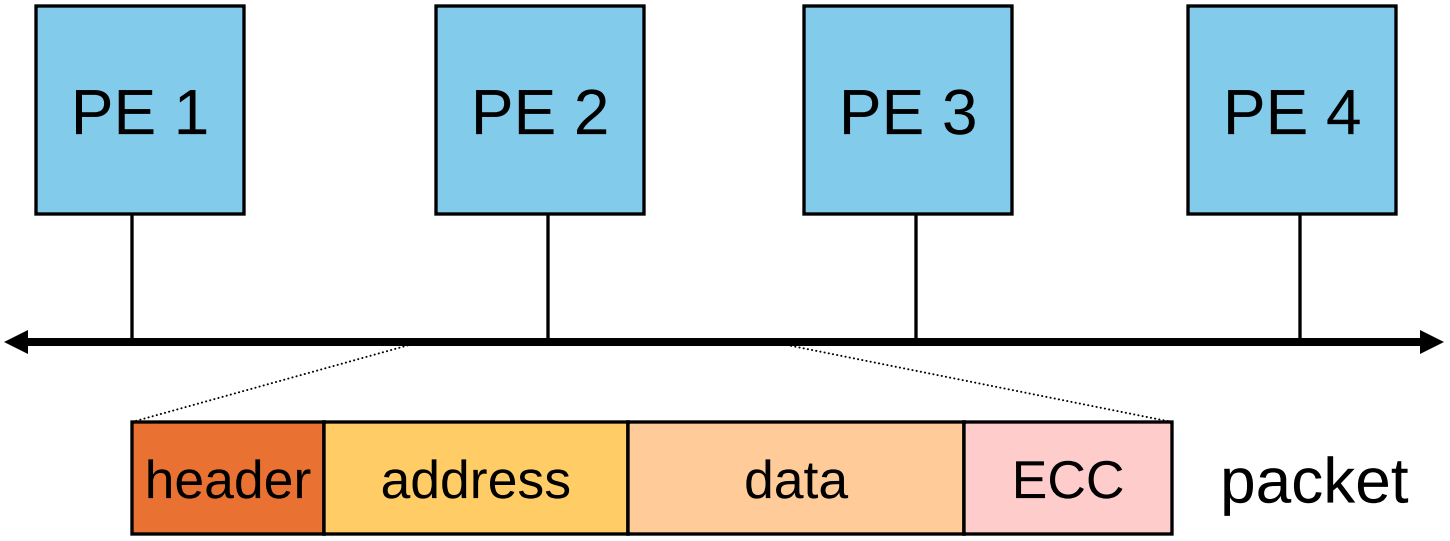
Point-to-point networks

- One source, one destination, no data switching
  - For example: RS232 serial port, IEEE 1284 parallel port



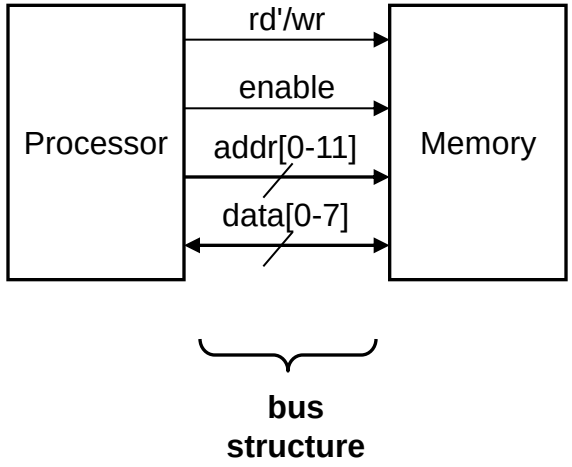
Bus networks

- Common physical connection:
  - Can be parallel or serial



A simple bus

- Wires:
  - Uni-directional or bi-directional
- Bus
  - Set of wires with a single function
    - Address bus, data bus
  - Or, entire collection of wires
    - Address, data and control
    - Associated protocol: rules for communication

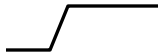


Timing Diagrams

Timing Diagrams

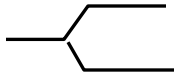
- Most common method for describing a communication protocol
- Time proceeds to the right on x-axis

- Control signal: low or high



  - May be active low (e.g., go', /go, or go_L)
  - Use terms *assert* (active) and *deassert*
  - Asserting go' means go=0

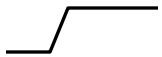
- Data signal: not valid or valid



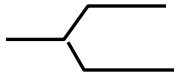
Timing Diagrams

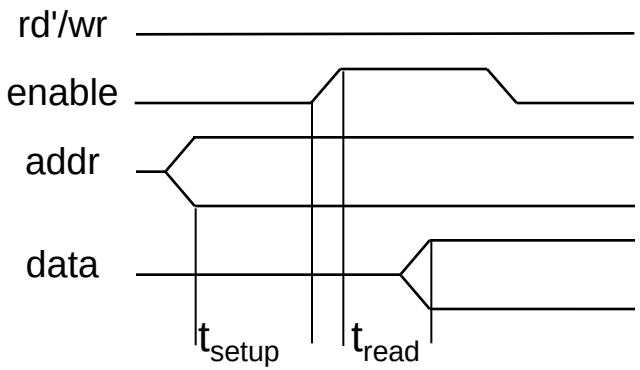
- Most common method for describing a communication protocol
- Time proceeds to the right on x-axis

- Control signal: low or high
  - May be active low (e.g., go', /go, or go_L)
  - Use terms *assert* (active) and *deassert*
  - Asserting go' means go=0



- Data signal: not valid or valid



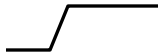


example read protocol

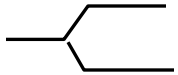
Timing Diagrams

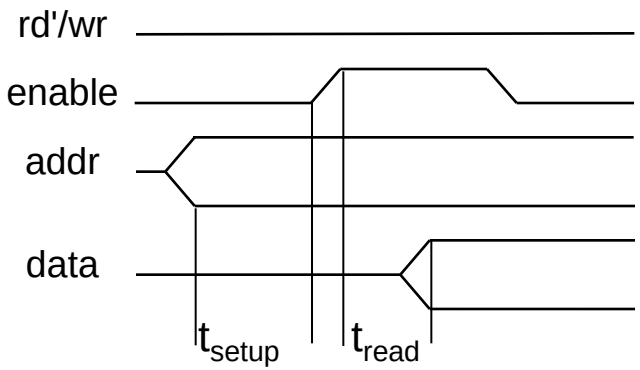
- Most common method for describing a communication protocol
- Time proceeds to the right on x-axis

- Control signal: low or high
  - May be active low (e.g., go', /go, or go_L)
  - Use terms *assert* (active) and *deassert*
  - Asserting go' means go=0

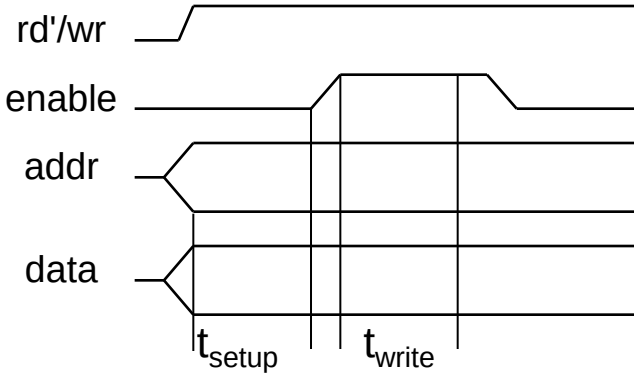


- Data signal: not valid or valid





example read protocol

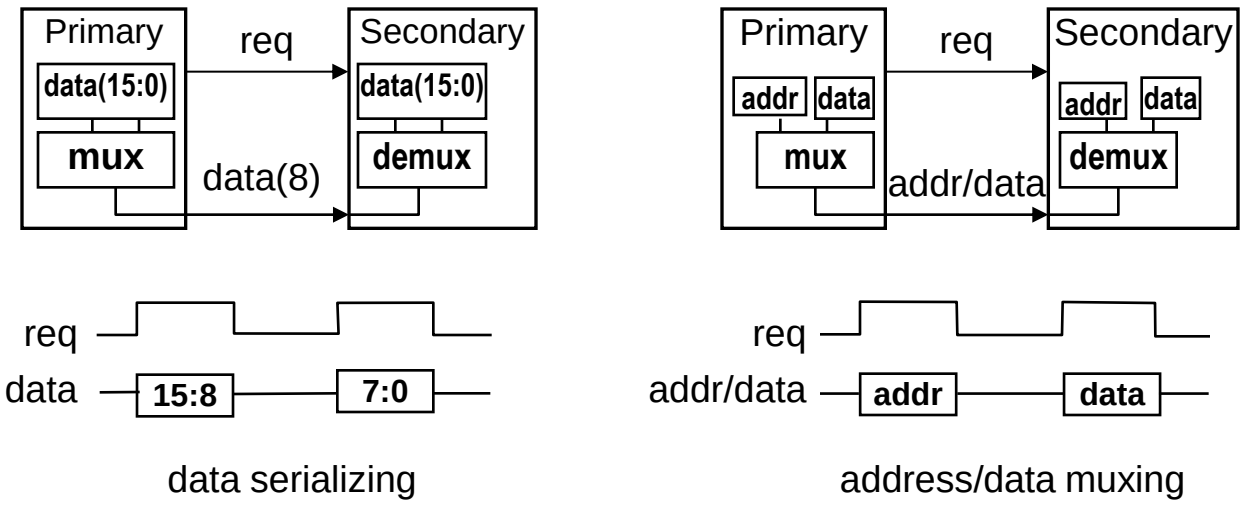


example write protocol

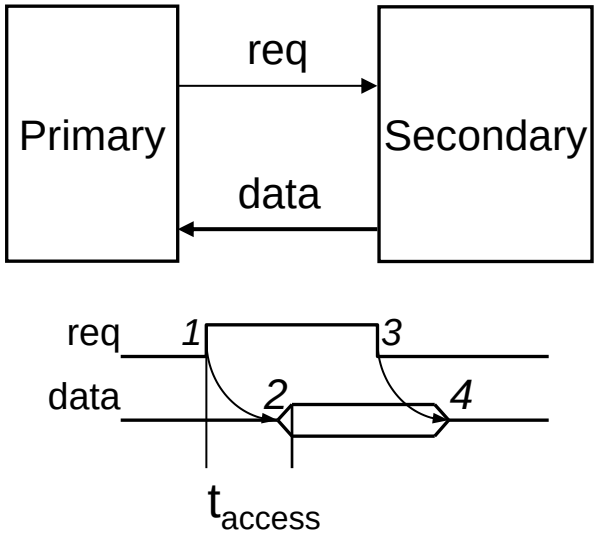
Basic protocol concepts

- Actor: *primary* initiates, *secondary* responds
- Direction: sender, receiver
- Addresses: special kind of data
  - Specifies a location in memory, a peripheral, or a register within a peripheral
- Time multiplexing
  - Share a single set of wires for multiple pieces of data
  - Saves wires at expense of time

Time-multiplexed data transfer



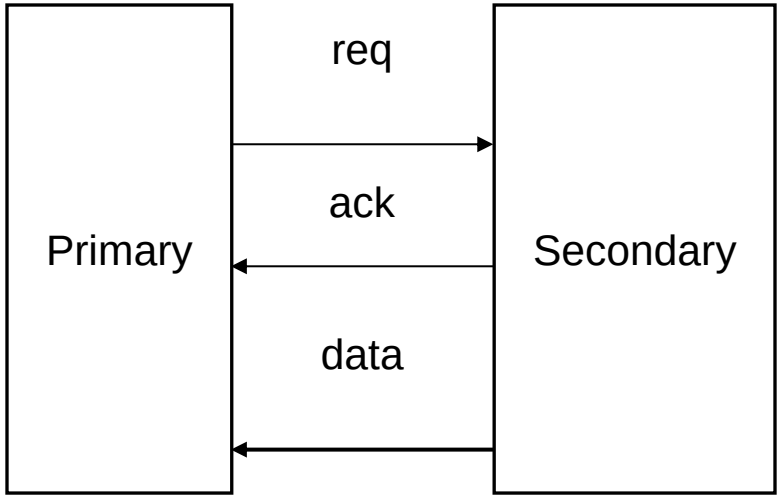
Basic protocol concepts: control methods

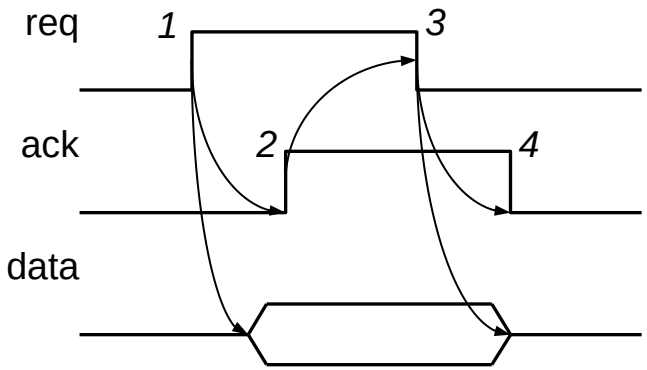


1. Primary asserts *req* to receive data
2. Secondary puts data on bus **within time t_{access}**
3. Primary receives data and deasserts *req*
4. Secondary ready for next request

Strobe protocol

Basic protocol concepts: control methods

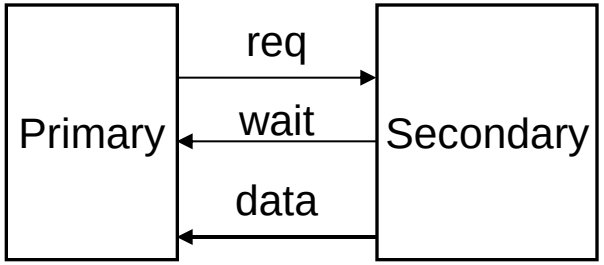


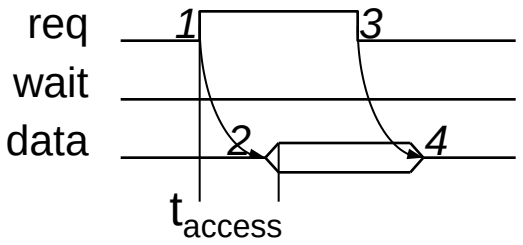


1. Primary asserts *req* to receive data
2. Secondary puts data on bus
and asserts *ack*
3. Primary receives data and deasserts *req*
4. Secondary ready for next request

Handshake protocol

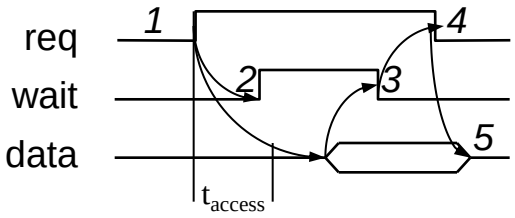
A strobe/handshake compromise





1. Primary asserts *req* to receive data
2. Secondary puts data on bus **within time t_{access}** (wait line is unused)
3. Primary receives data and deasserts *req*
4. Secondary ready for next request

Fast-response case

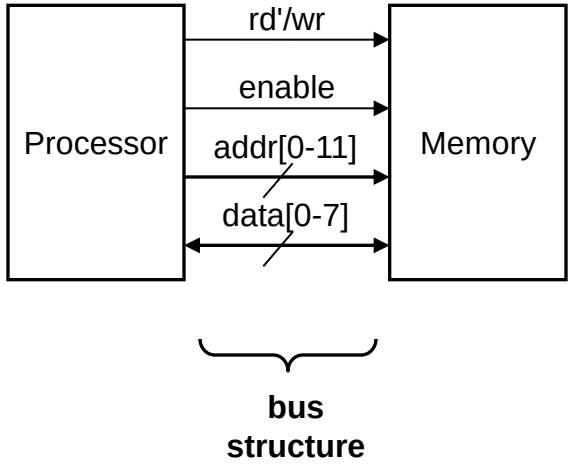


1. Primary asserts *req* to receive data
2. Secondary can't put data within **t_{access}**, **asserts *wait*** ack
3. Secondary puts data on bus and **deasserts *wait***
4. Primary receives data and deasserts *req*
5. Secondary ready for next request

Slow-response case

A simple bus

- Wires:
  - Uni-directional or bi-directional
- Bus
  - Set of wires with a single function
    - Address bus, data bus
  - Or, entire collection of wires
    - Address, data and control
    - Associated protocol: rules for communication

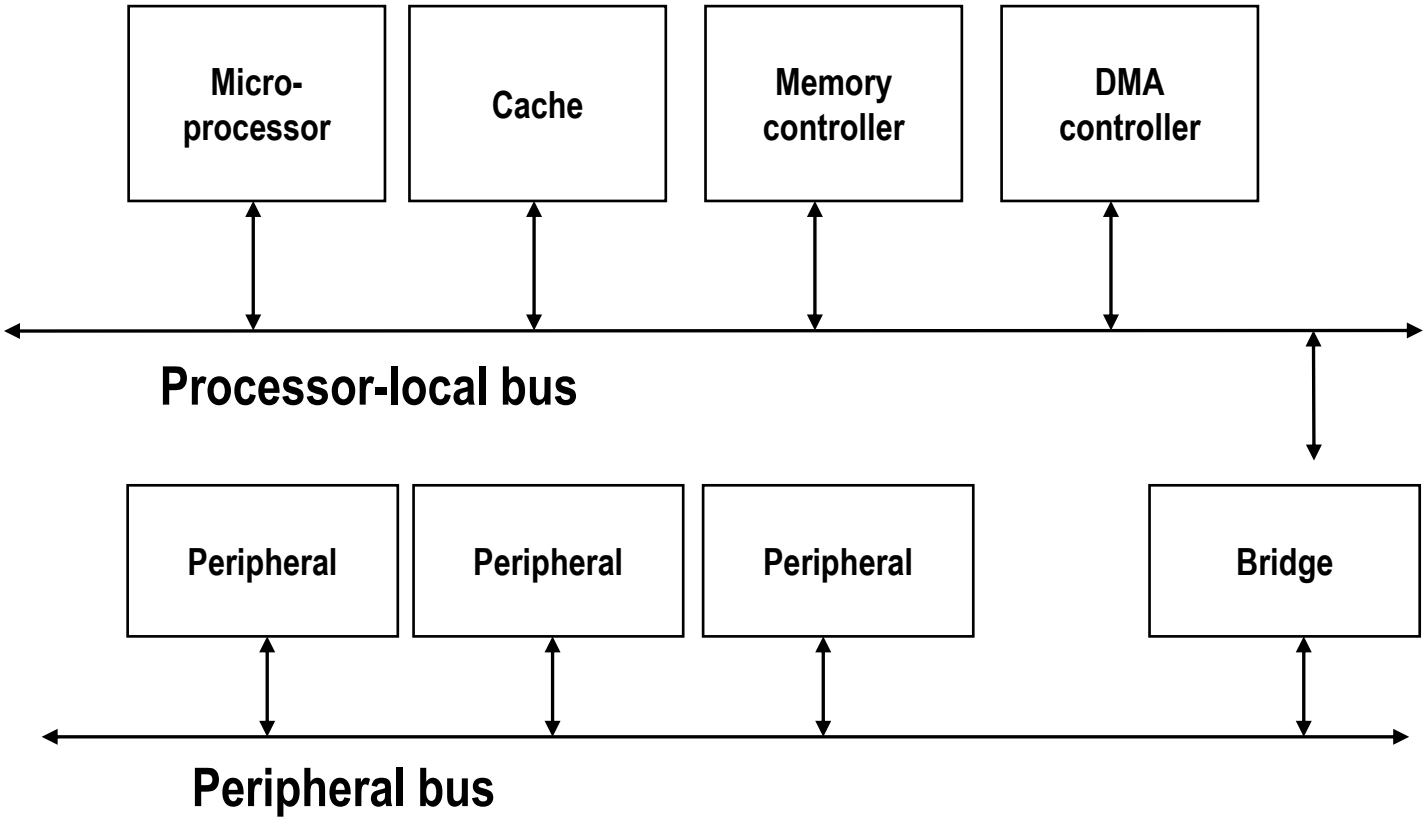


Classification based on the physical layer

- ***Parallel communication***
  - Physical layer capable of transporting multiple bits of data
- ***Serial communication***
  - Physical layer transports one bit of data at a time
- ***Wireless communication***
  - No physical connection needed for transport at physical layer

Multilevel bus architectures

- Don't want **one bus for all communication**
  - Peripherals would not need high-speed, processor-specific bus interface
    - excess gates, power consumption, and cost; less portable
  - Too many peripherals slows down bus



who gets access first?

Arbitration: Priority arbiter

Arbitration: Priority arbiter

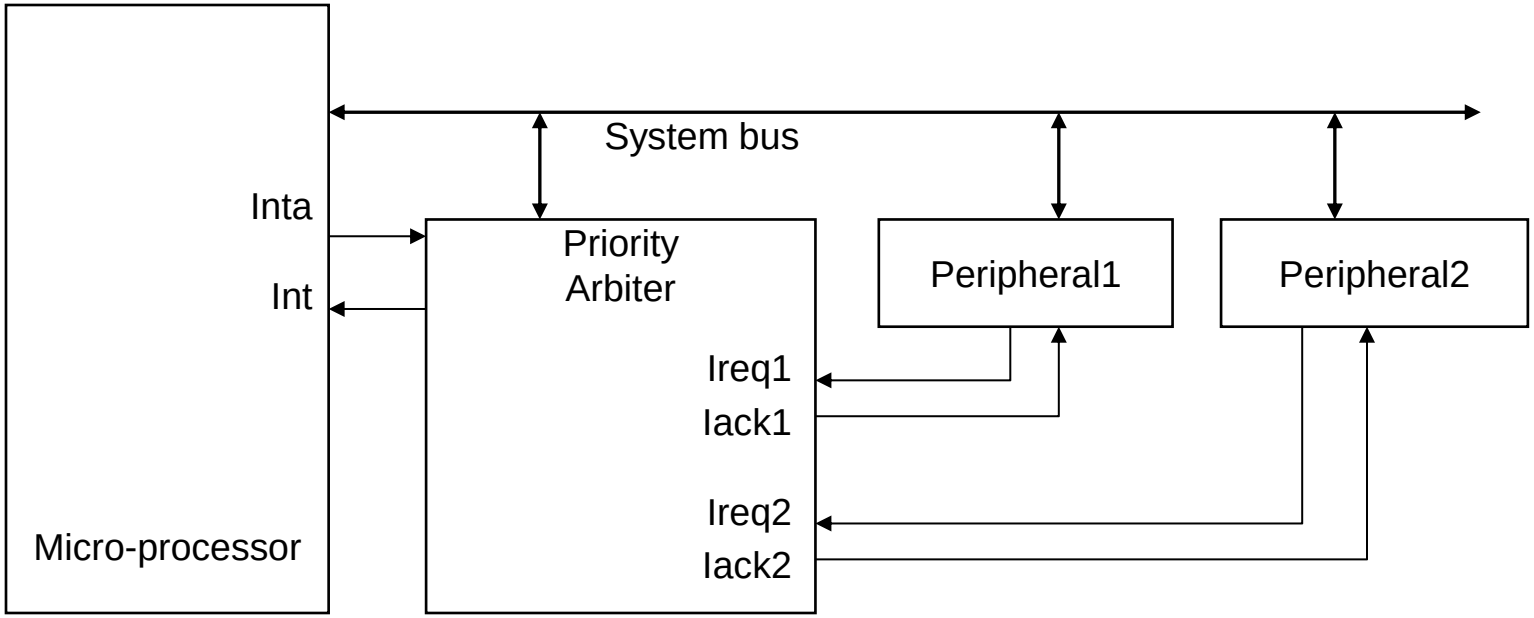
- Multiple peripherals request service from single resource (e.g., microprocessor, memory access controller) simultaneously.

Arbitration: Priority arbiter

- Multiple peripherals request service from single resource (e.g., microprocessor, memory access controller) simultaneously.
- **Priority arbiter**
  - Single-purpose processor
  - Peripherals make requests to arbiter, arbiter makes requests to resource
  - Arbiter connected to system bus for ***configuration only***

Arbitration: Priority arbiter

- Multiple peripherals request service from single resource (e.g., microprocessor, memory access controller) simultaneously.
- **Priority arbiter**
  - Single-purpose processor
  - Peripherals make requests to arbiter, arbiter makes requests to resource
  - Arbiter connected to system bus for ***configuration only***



Arbitration: Priority arbiter

Types of priority

Arbitration: Priority arbiter

Types of priority

- Fixed priority
- Rotating priority (round-robin)

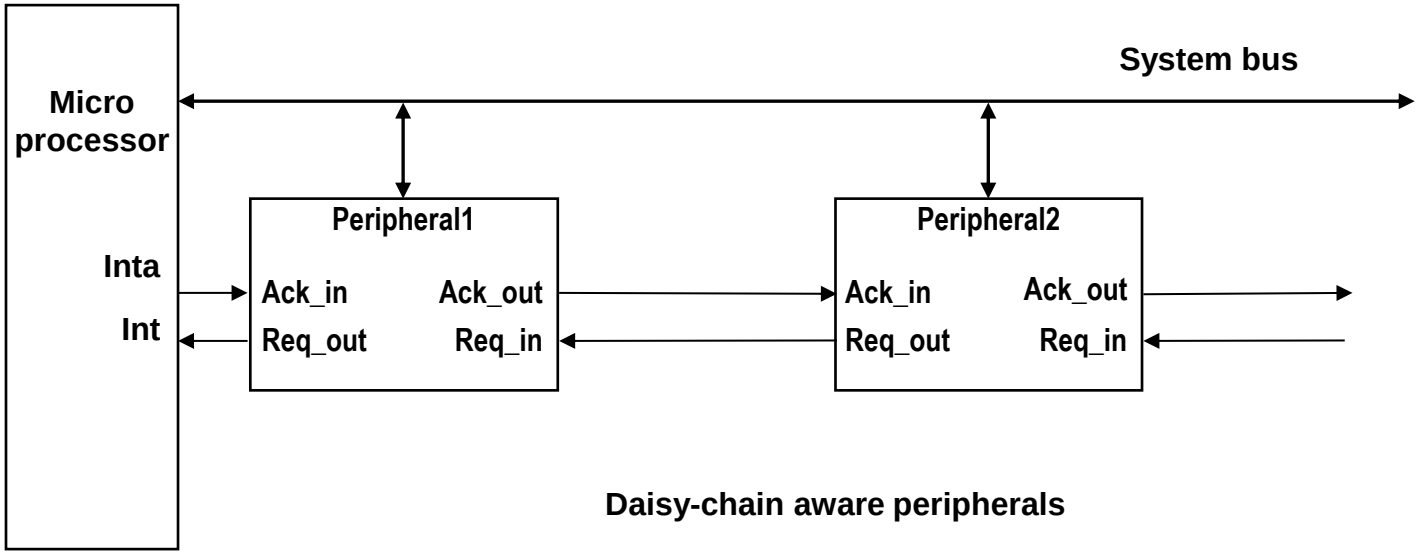
Must we have an arbiter?

Arbitration: Daisy-chain arbitration

- Arbitration done by peripherals
- Peripherals connected to each other in daisy-chain manner

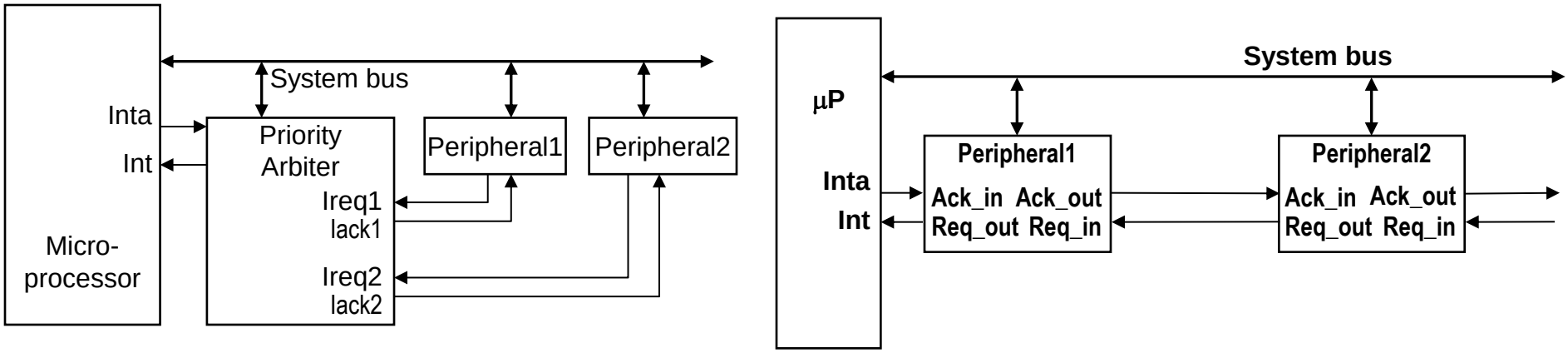
Arbitration: Daisy-chain arbitration

- Arbitration done by peripherals
- Peripherals connected to each other in daisy-chain manner



Arbitration: Daisy-chain arbitration

- Pros/cons
  - Easy to add/remove peripheral - no system redesign needed
  - Does not support rotating priority
  - One broken peripheral can cause loss of access to other peripherals

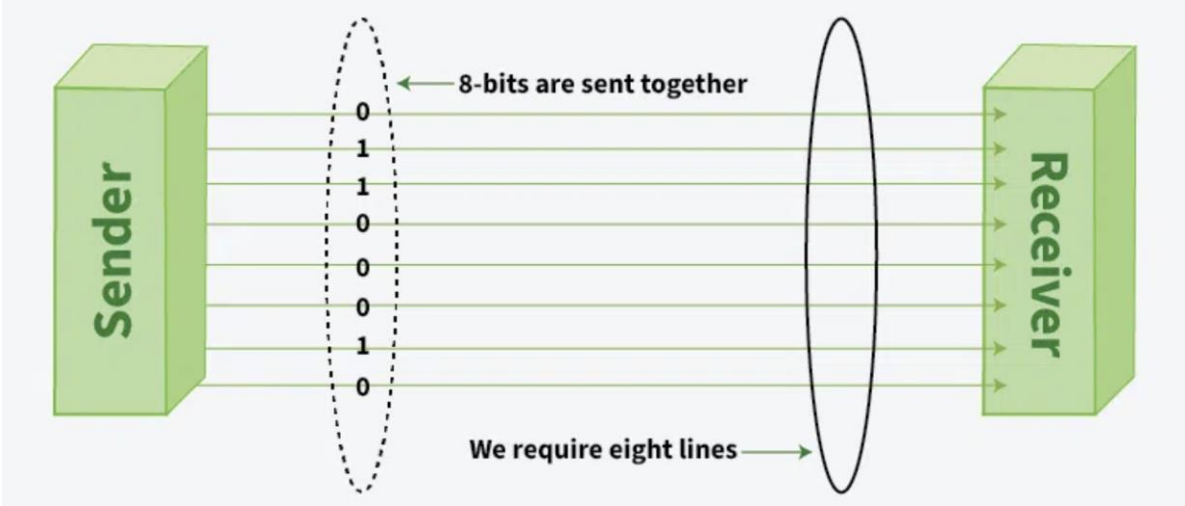


Network-oriented arbitration

- When ***multiple microprocessors*** share a bus (sometimes called a network)
  - Separate processors may try to write simultaneously causing collisions
    - Data must be resent
    - Don’t want to start sending again at same time
      - statistical methods can be used to reduce chances
- Typically used for connecting ***multiple distant chips***
  - Trend – use to connect multiple on-chip processors

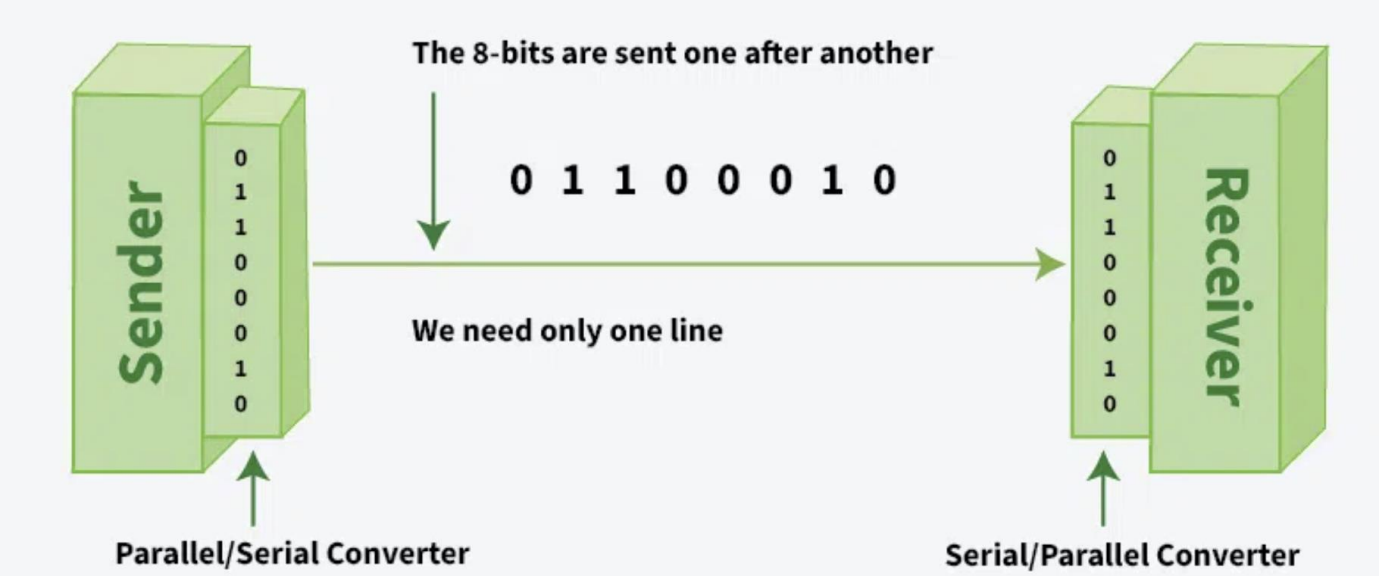
Classification based on the physical layer

- ***Parallel communication***
  - Physical layer capable of transporting multiple bits of data



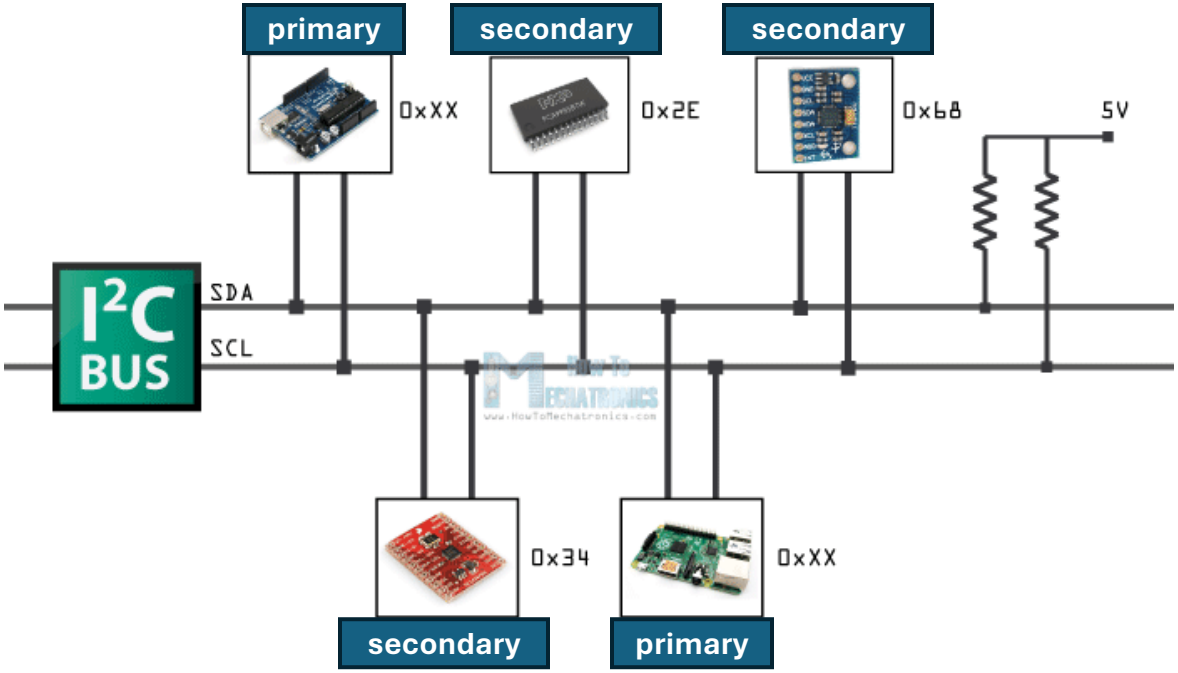
Classification based on the physical layer

- ***Serial communication***
  - Physical layer transports one bit of data at a time



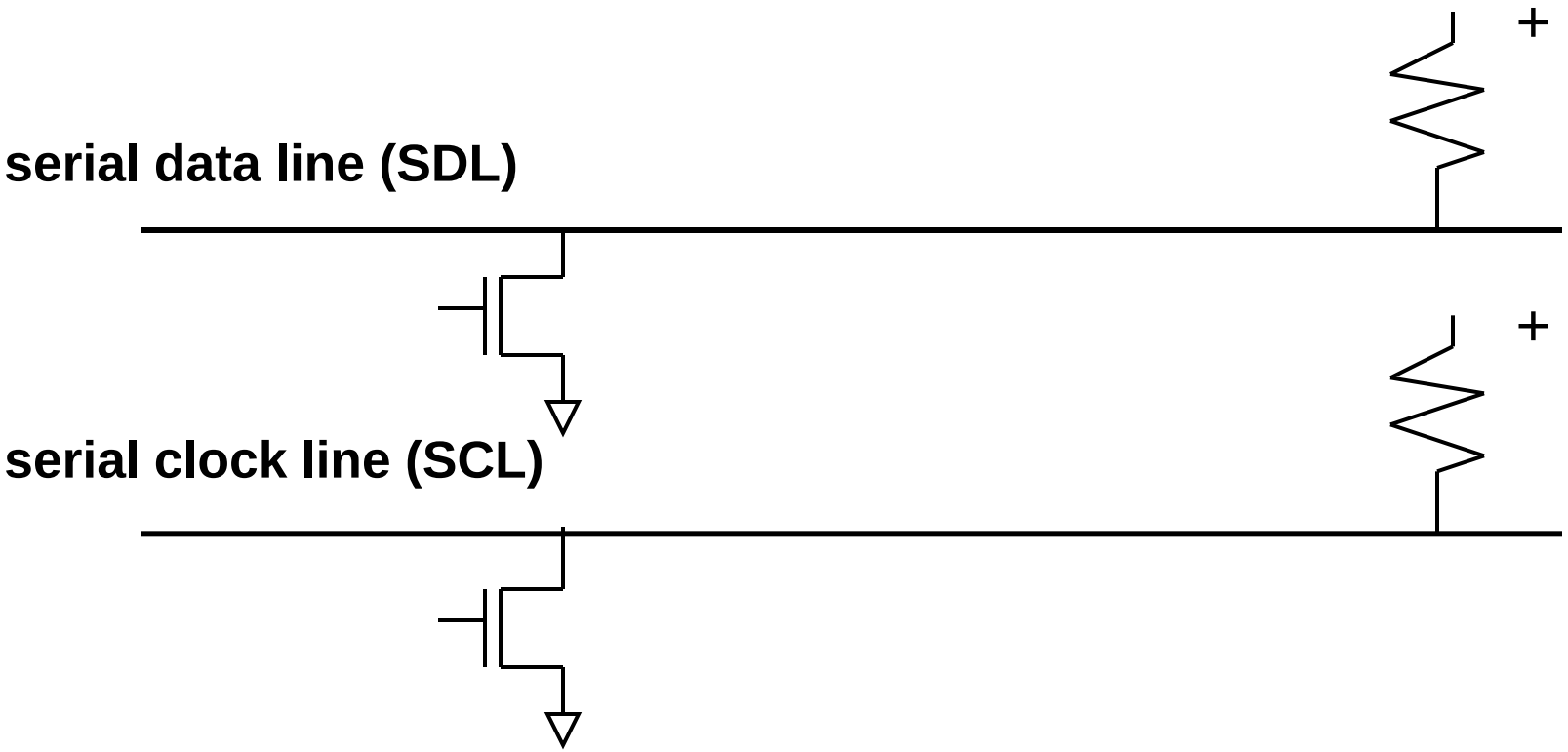
Serial protocols: I²C

- I²C (Inter-IC)
  - Two-wire serial bus protocol originally developed by Philips
  - Data transfer rates up to 100 kbits/s and 7-bit addressing
  - Common devices capable of interfacing to I²C bus:
    - EPROMS, Flash, LCD controller, some RAM memory, real-time clocks, watchdog timers, and microcontrollers



I^2C electrical interface

- Open drain/collector interface:



Wired AND logic

Example

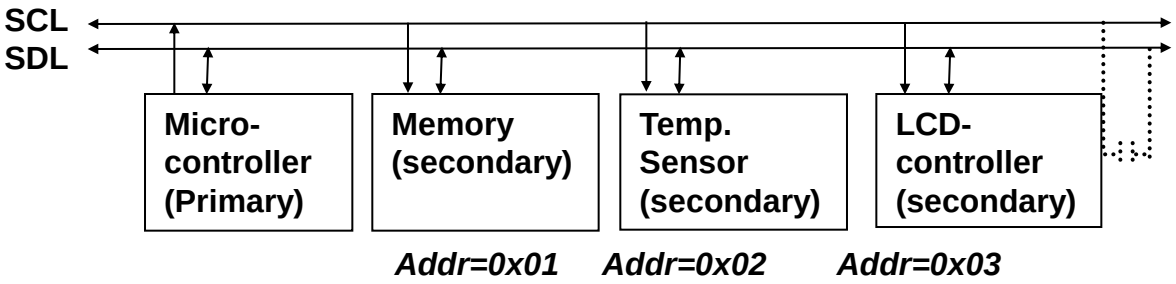
I^2C Overview (simplified):

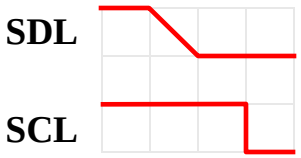
1. **START condition** (SDA goes low while SCL is high)
2. **7-bit address** is sent (MSB first)
3. **1-bit Read/Write flag**
4. **ACK** from the receiver
5. Then comes **data bytes** (if any), followed by another ACK after each one.
6. **STOP condition** (SDA goes high while SCL is high)

I^2C bus arbitration

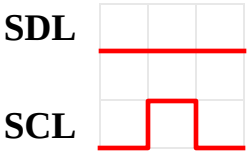
- Sender ***listens*** while sending address.
- When sender hears a ***conflict***, it stops signaling.
  - The one sending ‘1’ will hear conflict due to wired-and logic.
  - The one sending ‘0’ will not hear conflict.
- No waste of time on bus since the high-priority sender proceeds

I²C bus structure

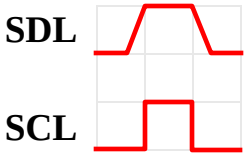




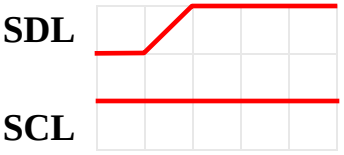
Start condition



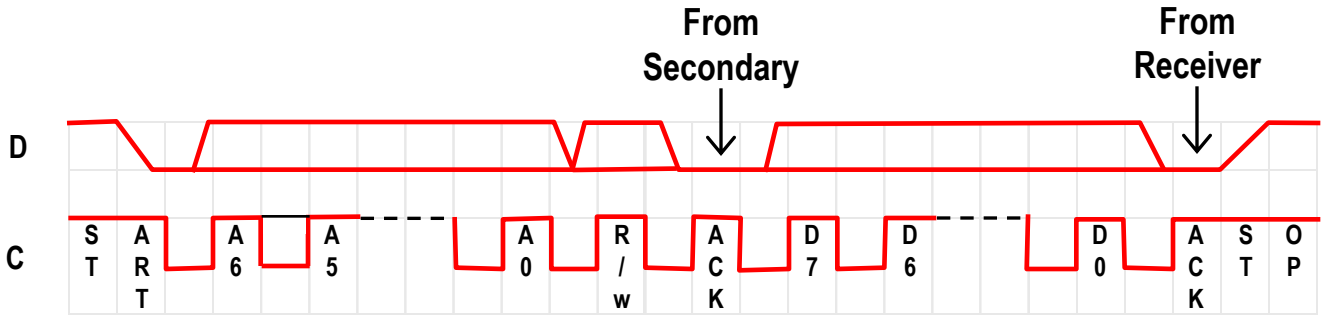
Sending 0



Sending 1

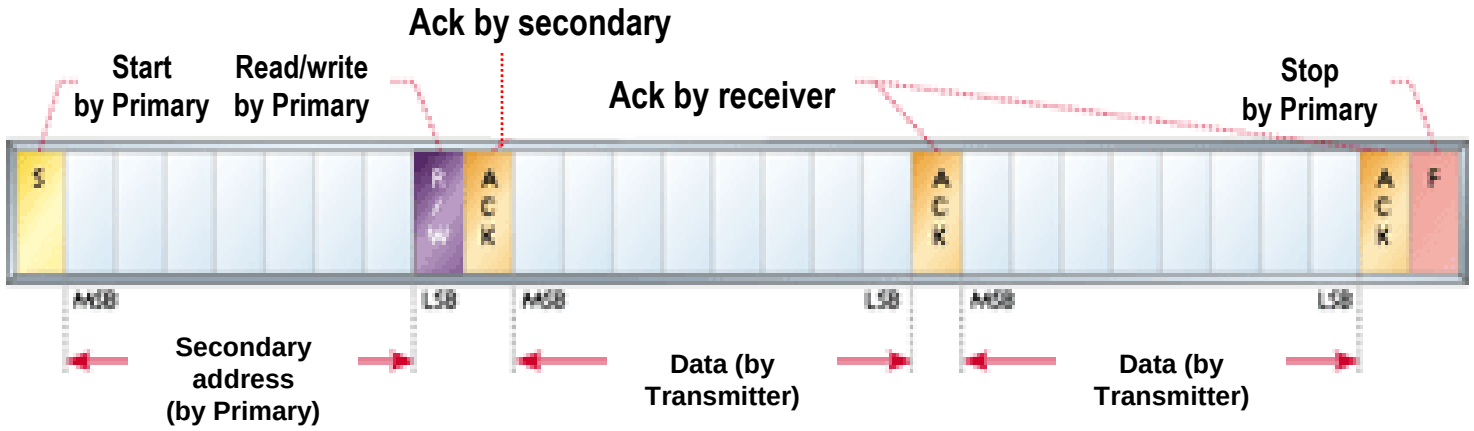


Stop condition

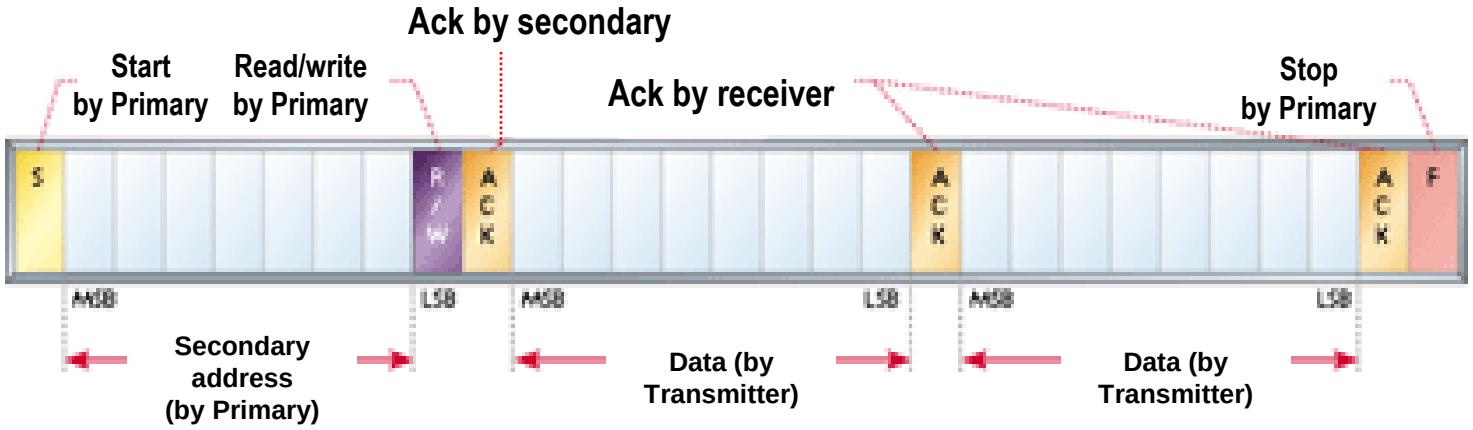


Typical read/write cycle

I²C Protocol



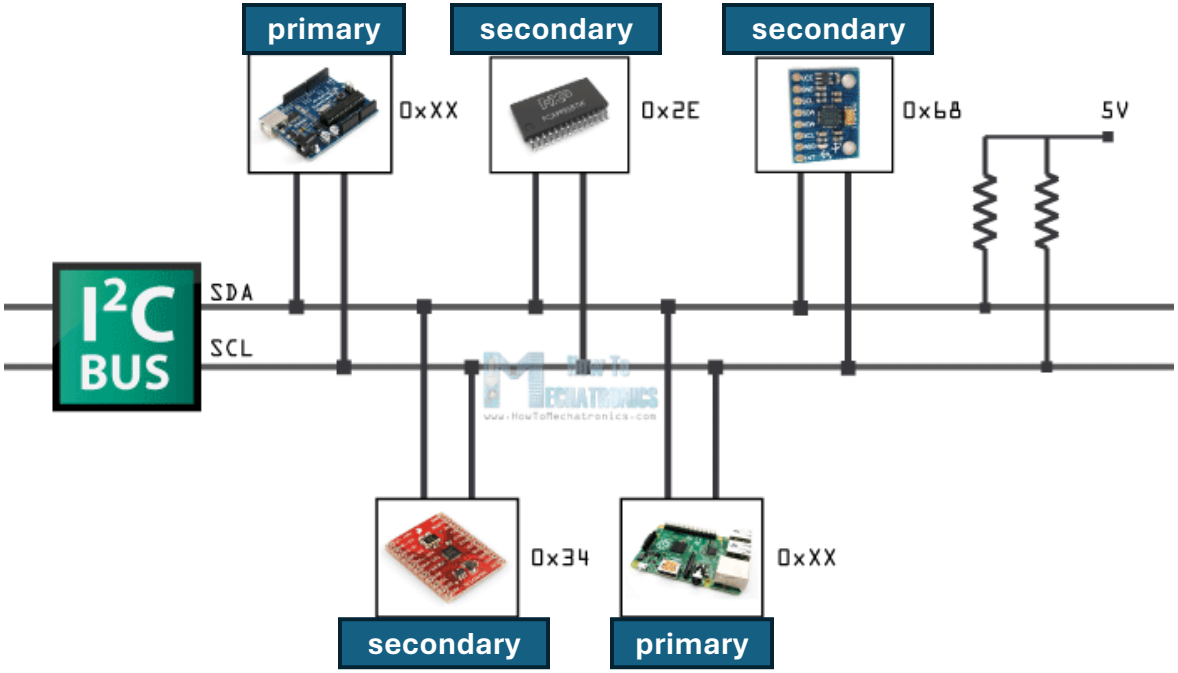
1. Primary sends start condition (S) and controls the clock signal
2. Primary sends a unique 7-bit Secondary device address
3. Primary sends read/write bit (R/W) – 0 - Secondary receive, 1 - Secondary transmit
4. Secondary sends acknowledge bit (ACK)
5. Transmitter (Primary or Secondary) transmits 1 byte of data



1. Primary sends start condition (S) and controls the clock signal
2. Primary sends a unique 7-bit Secondary device address
3. Primary sends read/write bit (R/W) – 0 - Secondary receive, 1 - Secondary transmit
4. Secondary sends acknowledge bit (ACK)
5. Transmitter (Primary or Secondary) transmits 1 byte of data
6. Receiver issues an ACK bit for the byte received
7. Repeat 5 and 6 if more bytes need to be transmitted.
8. a) For write transaction (Primary transmitting), Primary issues stop condition (P) after last byte of data.
8. b) For read transaction (Primary receiving), Primary does not acknowledge final byte, just issues stop condition (P) to tell the Secondary that the transmission is done.

Serial protocols: I²C

- I²C (Inter-IC)
  - Two-wire serial bus protocol originally developed by Philips
  - Data transfer rates up to 100 kbits/s and 7-bit addressing
  - Common devices capable of interfacing to I²C bus:
    - EPROMS, Flash, LCD controller, some RAM memory, real-time clocks, watchdog timers, and microcontrollers



Open-Drain

- Only 0 is actively pulled low
- 1 is just a passive *pull-up resistor*

In push-pull, both logic levels, high and low, actively driven by transmitter:

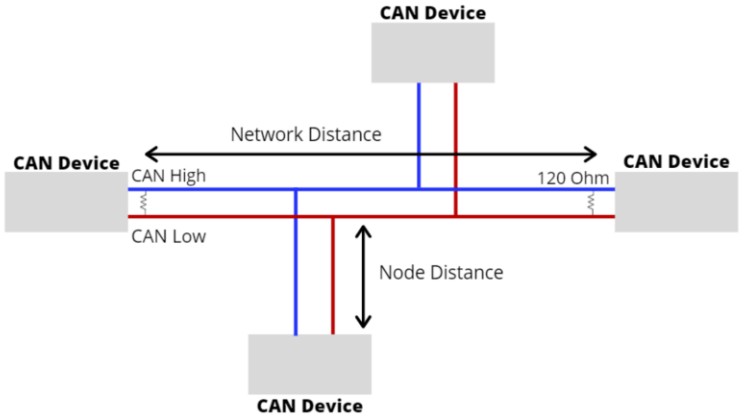
- To send a 1, the transmitter *pushes the line high* (connects it to Vcc).
- To send a 0, it *pulls the line low* (connects it to GND).

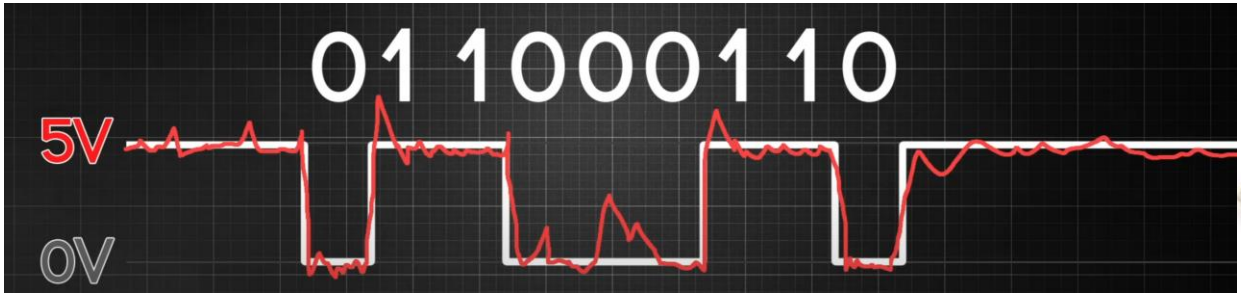
Summary:

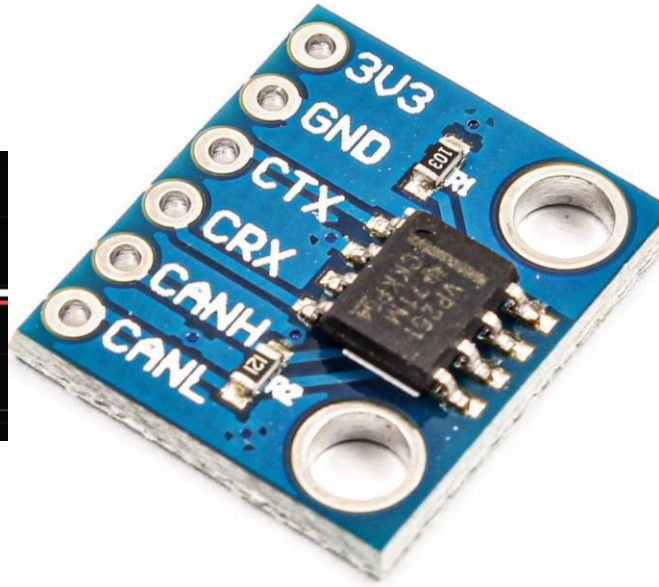
Feature	**Open-Drain (I^2C)**	**Push-Pull (I^3C)**
HIGH signal	Passive (via resistor)	Actively driven
LOW signal	Actively pulled down	Actively pulled down
Signal rise time	Slow (RC delay)	Fast (no pull-up delay)
Power efficiency	Lower	Higher

Another Serial Protocol: CAN

- CAN (Controller area network)
  - Developed by Robert Bosch GmbH
  - Originally for communication among components of cars
  - Applications now using CAN include:
    - elevator controllers, copiers, telescopes, production-line control systems, and medical instruments
  - Protocol very similar to I^2C but with better error detection capability
    - Covering longer distance







Hierarchical Serial Buses

- Each physical link connects two ends
- Can connect multiple devices as a tree
- Firewire
  - 6 pins: Vdd (30v, 45W), GND, A+, A-, B+, B-
  - 4 pins: A+, A-, B+, B-
  - One signal pair for strobe, the other for data
- USB
  - 4 pins: Vdd, GND, D+, D-