

EE3220 System on Chip Design

Lecture Note II

Integrating System-on-Chip Components

Review & Preview

- Lecture 1: Course Overview
- Lecture 2: Intro to Modern Microprocessor (CPU Architecture)
- Lecture 3: Embedded System Software (Software Architecture)
- Lecture 4: ARM Cortex-M4 (ARM Hardware)
- Lecture 5: From C to ARM Assembly (C Language, ARM Firmware)
- Lecture 6: Team Based Learning (USB Firmware)
- Lecture 7: AAPCS (ARM Firmware)
- Lecture 8: Exceptions & Interrupts (ARM Firmware)
- Lecture 9: Intro to HDL & FPGA Design (HDL Design)
- Lecture 10: FPGA based SoC (FPGA based SoC)
- **Lecture 11: Integrating SoC Components (SoC Integration)**
- Lecture 12: Team Based Learning (Wi-Fi SoC)
- Lecture 13: RISC-V Architecture (RISC-V, Guest Speaker)

Overview

- Quiz 2 Results
- SoC Components & Integration
- SoC Peripherals (GPIO, USART, Timer)
- SoC Bus Architectures
- DMA Controllers for Efficient Data Transfer

Overview

- Quiz 2 Results
- SoC Components & Integration
- SoC Peripherals (GPIO, USART, Timer)
- SoC Bus Architectures
- DMA Controllers for Efficient Data Transfer

Quiz 2 Stats

- Average: ~57 (/100)
- Standard Dev.: ~11 (/100)
- Highest Score: 88 (/100)
- > 57 (Average) : DOING WELL
- < 46 (Average - Standard Dev.): TIME TO **CATCH UP**

Multiple Choices Questions

4. Which C data type is best suited for storing memory-mapped hardware registers?

- A. int
- B. volatile int
- C. const int
- D. static int

Ans: B

15. What is the primary function of a watchdog timer in an embedded system?

- A. Managing network communication
- B. Resetting the system if software hangs
- C. Handling floating-point calculations
- D. Improving power efficiency

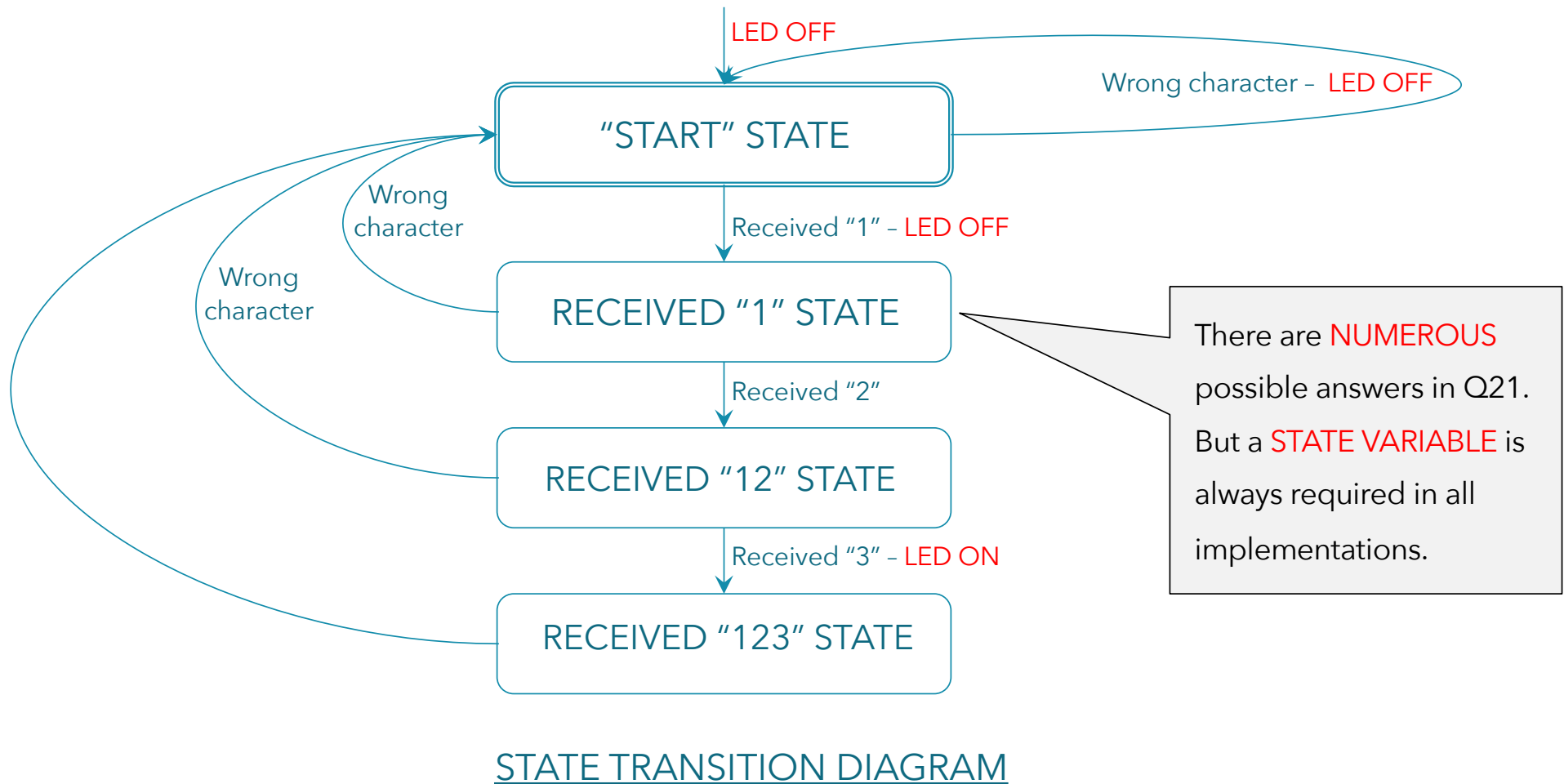
Ans: B

18. In Verilog, which statement regarding blocking and non-blocking assignments is correct?

- A. Non-blocking assignments are slower than blocking assignments
- B. Blocking assignments occur sequentially, non-blocking assignments occur concurrently
- C. Non-blocking assignments are faster than blocking assignments
- D. Blocking and non-blocking assignments are the same

Ans: B

Question 21



Question 21

```
#include "mbed.h"
static BufferedSerial pc(USBTX, USBRX);
int main(){
    DigitalOut led(LED1);
    char command;
    char passphrase[3] = {'1','2','3'};
    int index = 0;
    led = 0;
    while (true){
        pc.read(&command, 1);
        if (command == passphrase[index]) {
            index++;
            if (index == 3) {
                index = 0;
                led = 1;
            } else {
                led = 0;
            }
        } else {
            index = 0;
            led = 0;
        }
    }
}
```

Most practical problems require **STATE VARIABLE(S)** and **STATE MACHINE(S)**.
C variables are needed to define the state variable(s).

Question 22

```
always @(posedge clk) begin
    if (!rst_n) begin
        sum    <= 1'b0;
        carry  <= 1'b0;
    end else begin
        sum    <= a ^ b;
        carry  <= a & b;
    end
end
```

Unlike in asynchronous reset, the sequential logic in synchronous reset is evaluated only at every rising clock edge. In other words, the logic is not sensitive to small reset glitches.

The "sum" and "carry" registers are evaluated concurrently, regardless whether blocking or non-block assignment is used.

Using non-blocking assignments (`<=`) in the following example is incorrect.

```
sum    = a ^ b;
carry  = a & b & ~sum;
```

Question 23

Beginning of the Interrupt Service Routine (ISR)

1. SP points to MSP
2. LR = EXC_RETURN
3. PC = interrupt vector
4. Update PSP after PUSH

xPSR = 0x21000000 | exception_no

R0	0x20
R1	0x21
R2	0x22
R3	0x23
R4	0x24
R5	0x25
R6	0x26
R7	0x27
R8	0x28
R9	0x29
R10	0x30
R11	0x31
R12	0x32
R13 (SP)	0x30001000
R14 (LR)	0xFFFFFFFF
R15 (PC)	0x0000C000
xPSR	0x21000000
MSP	0x30001000
PSP	0xC0002FE0

Address	Content
0x30000FE0	0x100
0x30000FE4	0x101
0x30000FE8	0x102
0x30000FEC	0x103
0x30000FF0	0x104
0x30000FF4	0x105
0x30000FF8	0x106
0x30000FFC	0x107
0x30001000	0x108
0x30001004	0x109
0x30001008	0x110
0x3000100C	0x111
0x30001010	0x112
0x30001014	0x113
0x30001018	0x114
0x3000101C	0x115
0x30001020	0x116
0x30001024	0x117

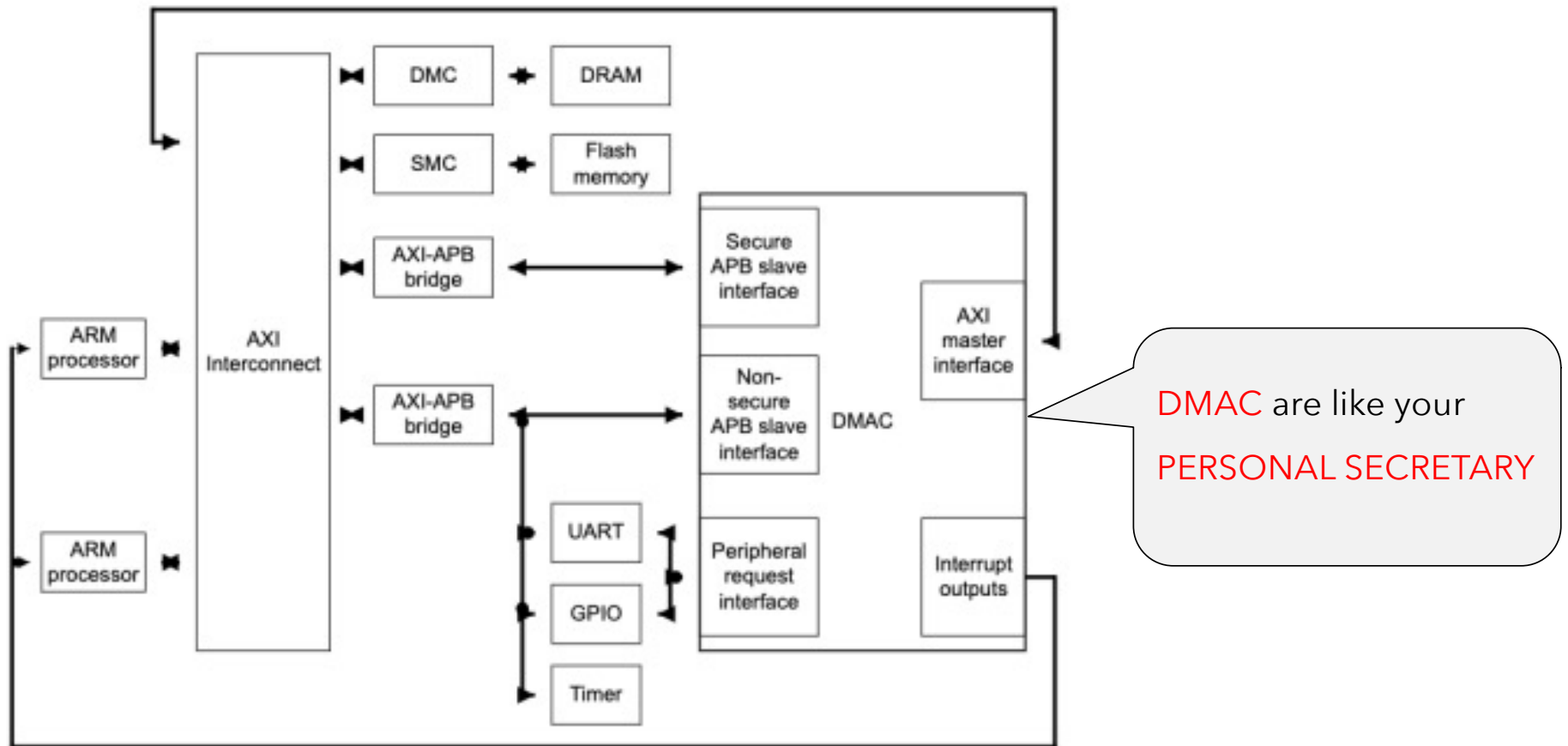
Address	Content
0xC0002FE0	0x20
0xC0002FE4	0x21
0xC0002FE8	0x22
0xC0002FEC	0x23
0xC0002FF0	0x32
0xC0002FF4	0x0000B000
0xC0002FF8	0x0000B800
0xC0002FFC	0x21000000
0xC0003000	0x208
0xC0003004	0x209
0xC0003008	0x210
0xC000300C	0x211
0xC0003010	0x212
0xC0003014	0x213
0xC0003018	0x214
0xC000301C	0x215
0xC0003020	0x216
0xC0003024	0x217

Push R0, R1, R2, R3,
R12, LR, PC, xPSR

Overview

- Quiz 2 Results
- SoC Components & Integration
- SoC Peripherals (GPIO, USART, Timer)
- SoC Bus Architectures
- DMA Controllers for Efficient Data Transfer

System-on-Chip Block Diagram



What are SoC Components?

- System-on-Chip:
 - Integration of all required components into a single chip
- Key Components:
 - Processor Cores – ARM, RISC-V, etc.
 - Memory Blocks – ROM, SRAM, Flash, etc.
 - BUS INTERFACES – AXI, AHB, APB
 - DMA CONTROLLERS – Efficient Data Transfer
 - PERIPHERALS – GPIO, UART, SPI, I2C, etc.
 - Custom IP Blocks – DSPs, Accelerators, etc.
 - Debug Modules – JTAG, SWD

Integrating SoC Components

- What does Integration Involve?
 - Connecting CPU, memory, and peripherals via **SHARED BUSES**
 - Mapping peripheral registers into memory address space
- Importance of Bus **ARCHITECTURE**
 - Acts as the backbone of communication within the SoC
 - Determines data transfer speed, latency, and access arbitration
- Integration Strategies
 - Isolate **LOW-SPEED** peripherals on APB, **HIGH-SPEED** on AXI
 - Use AXI interconnects to support multiple masters/slaves
 - Support for custom IP via standard AXI interfaces
- Design **CONSIDERATIONS**
 - Cost, power, bandwidth and latency requirements
 - Design for testing and manufacturing

Overview

- Quiz 2 Results
- SoC Components & Integration
- SoC Peripherals (GPIO, USART, Timer)
- SoC Bus Architectures
- DMA Controllers for Efficient Data Transfer

SoC Peripheral Examples

PERIPHERAL	FUNCTION
GPIO	Digital input/output
ADC / DAC	Analog-to-digital input / digital-to-analog output
USART	Synchronous & asynchronous serial communication
SPI / I2C	Synchronous serial buses
Timers / PWM	Generate delays or periodic pulses
Watchdog Timer	Auto-reset on system freeze

(1) Simple SoC Peripherals

- Memory-Mapped GPIOs / ADCs / DACs

- GPIOs and ADC / DAC registers are accessed like memory

- Example:

- `(volatile uint32_t*) (0x40011000) = 0x1;`

- GPIOs / ADCs / DACs are usually connected to Advanced Peripheral Bus (APB)

- Simple Interface: GPIOs / ADCs / DACs require basic read/write access

- Low Power: GPIOs / ADCs / DACs are low speed, low power peripherals

- No Pipelining / Bursts: GPIOs / ADCs / DACs require simple read/write accesses

- Not Latency Sensitive: GPIOs / ADCs / DACs tolerate delays

(2) USART: Why Serial Communication?

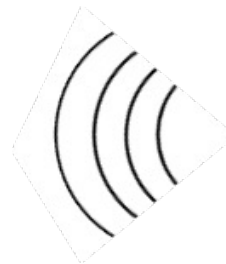
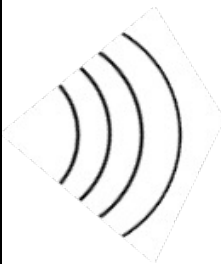
- Common Serial Communication IPs:
 - USART: Universal Synchronous and Asynchronous Receiver and Transmitter
 - UART: Universal Asynchronous Receiver and Transmitter
 - SPI: Serial Peripheral Interface
 - I2C/IIC: Inter-Integrated Circuit
- Communication Performance is proportional to **DATA WIDTH**.
 - However, it is sometimes not a good idea to use parallel communication.
 - **COSTS**: More wires and larger connectors needed for parallel communication
 - **RELIABILITY**: More connectors mean more connector failures
 - **TIMING**: Some bits may arrive too early due to variations across conductors (skews)
 - **COMPLICATIONS**: Undesirable to have too many EMI transmitters + receivers

(2) USART: Comparing Different Serial Communications

FEATURE	UART	USART	SPI
Clock Signal	No	Optional (Sync or Async)	Yes (always)
Communication	ASYNCHRONOUS (Peer-to-Peer)	ASYNC (Peer-to-Peer) + SYNC (Master-Slave)	SYNCHRONOUS (Master-Slave)
Full Duplex	Yes	Yes	Yes
Data Format	Start/Stop bits, parity	Start/Stop + optional clock	Clocked bitstream, no framing
Wires Needed	2 (TX, RX)	2 (TX, RX) or 3 (TX, RX, CLK)	3+ (MISO, MOSI, SCLK, SS)
Speed	Low to medium	Medium to high	High
Typical Use	Terminal comm, GPS	More flexible comms	Sensors, SD cards, displays

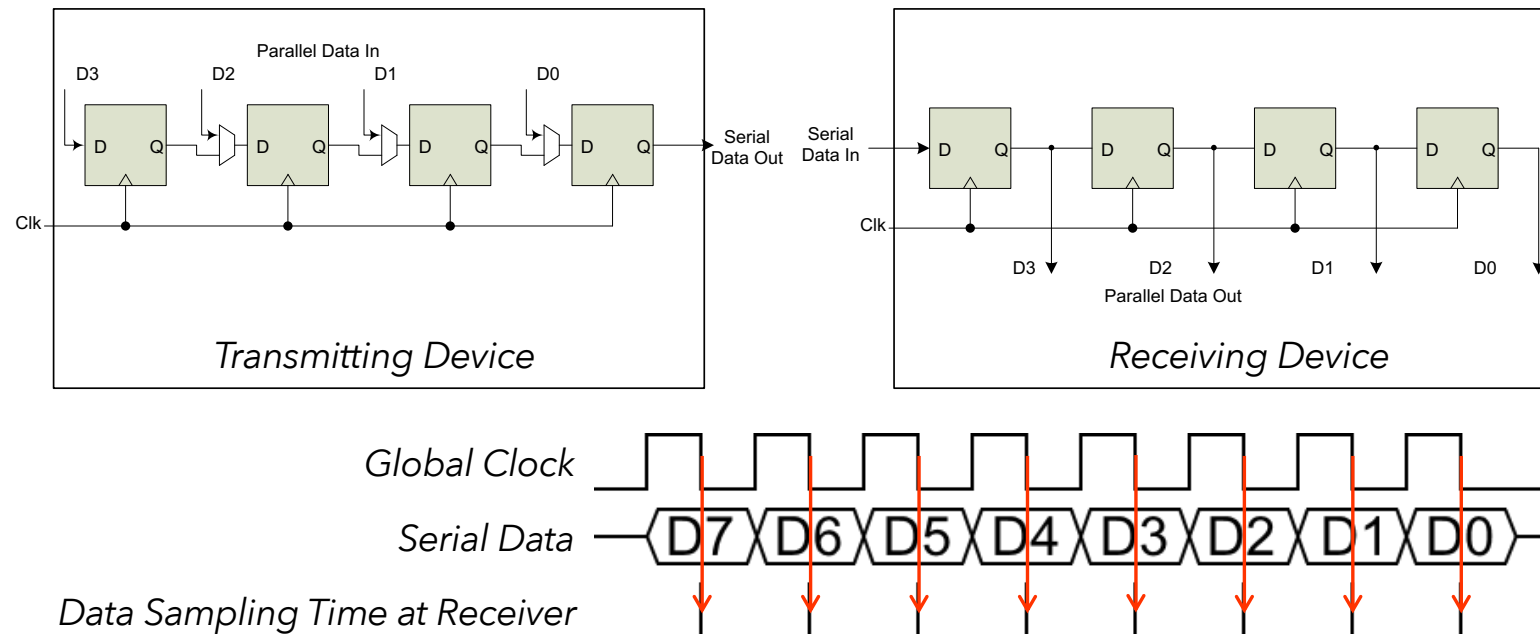
(2) USART: Serial Communication in Voyager Spacecraft

- Launched in 1977: Binary Phase-Shift Keying (BPSK)
- Constraints: Reliability, Power, Size, Weight, Reliability.
- **UPLINK** Communications to Voyager via **S-BAND**
 - 2.3GHz, 16-bps command rate
- **DOWNLINK** Communications from Voyager via **X-BAND**
 - 8.4GHz, 160 bps normally and 1.4 kbps for high-rate plasma wave data playback
- Both data transmit and receive at Voyager use 3.7-meter High-Gain Antenna



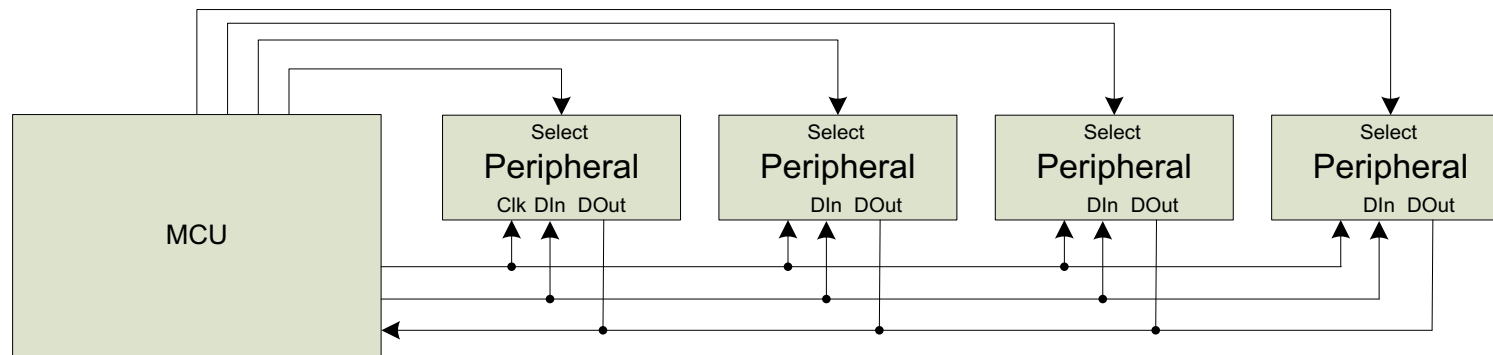
(2) USART H/W: Synchronous Serial Data Transmission

- Use Shift Registers and Clock Signal to convert between Serial and Parallel formats
 - Synchronous: "Global" clock signal is provided by the master device



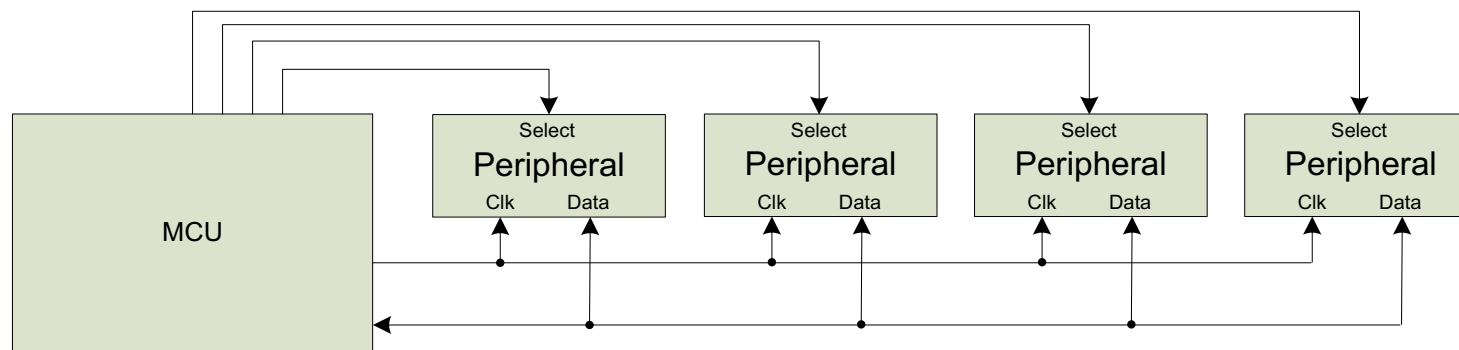
(2) USART H/W: **SYNCHRONOUS** Full-Duplex Serial Bus

- Now can use two serial data lines - one for reading, one for writing.
 - FULL-DUPLEX COMMUNICATION**: Allows simultaneous send and receive



(2) USART H/W: **SYNCHRONOUS** Half-Duplex Serial Bus

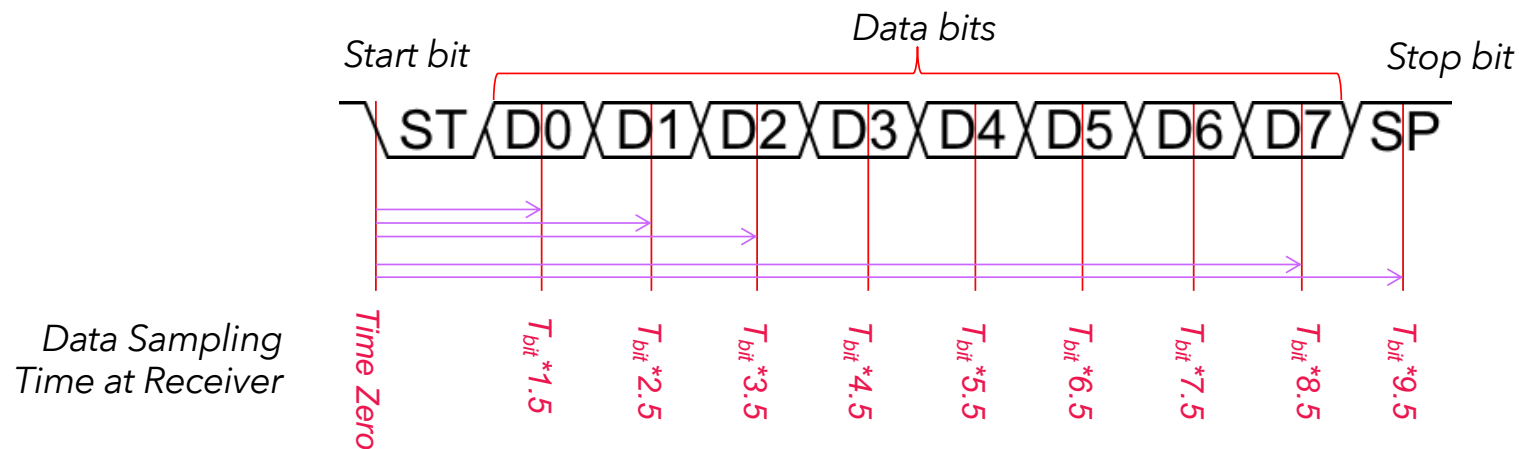
- Share the serial data line in half-duplex communication
 - **HALF-DUPLEX COMMUNICATION**: Does not allow simultaneous send and receive
- How about Simplex Communication?
 - **SIMPLEX COMMUNICATION**: Allows only one direction



(2) USART H/W: ASYNCHRONOUS Serial Communication

■ ELIMINATE CLOCK

- Transmitter and receiver must generate clocks locally
- Transmitter must add start bit to indicate start of each data frame
- Receiver detects leading edge of start bit
 - Sample data line to extract each data bit N at time $T_{\text{bit}} \times (N+1.5)$
 - Stop bit is also used to detect some timing errors



(2) USART H/W: **ASYNCHRONOUS** Serial Communication

- Data frame fields

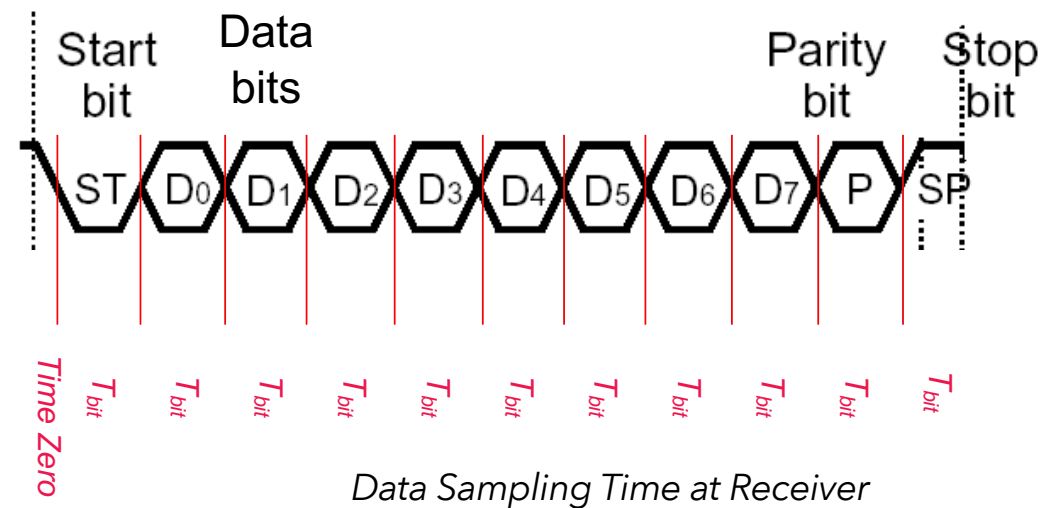
- Start bit: 1 bit
- Order: LSB or MSB first
- Data Size: 7, 8, 9 bits
- Parity bit: Optional
- Stop bit: 1 or 2 bits



- All devices must use the same communication parameters
 - Baud rate: 300, 600, 1200, 2400, 9600, 14400, 19200 bps
- Sophisticated network protocols embed more information in each data frame
 - Medium access control: Arbitrate to transmit when there are multiple nodes are on bus
 - Addressing info: Determine the target destination node
 - Efficiency: Larger data payload
 - Data integrity: Stronger error detection or error correction information
 - Request for immediate response ("in-frame")

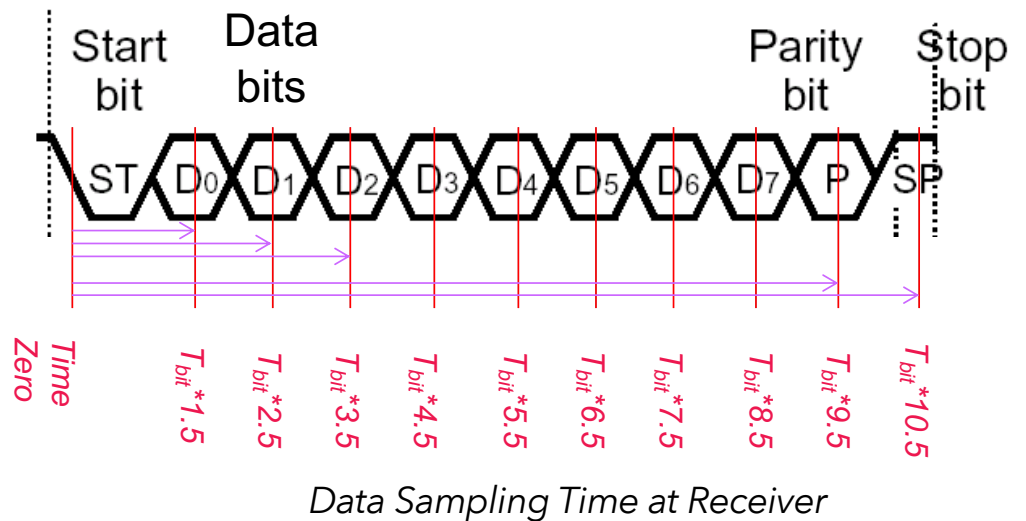
(2) USART H/W: Transmitter Basics

- When there is no data to send:
 - Keep sending 1 (stop bit) to indicate bus idle
- When there is a word to send:
 - Send a 0 (start bit) to indicate the start of the word
 - Send each data bit in the word (use a shift register)
 - Send the parity bit
 - Send one or more stop bits to indicate the end



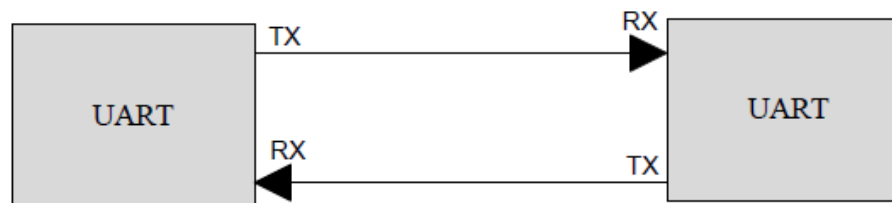
(2) USART H/W: Receiver Basics

- Wait for the falling edge (Start bit)
 - Then wait $\frac{1}{2}$ bit time
- Do the following for all data bits in the word
 - Wait 1 bit time
 - Read the data bit and shift it into a receive buffer (shift register)
- Read the bit (Parity bit)
 - Check if the Parity bit is correct
- Read the bits (Stop bits)
 - Check if the Stop bit(s) are correct



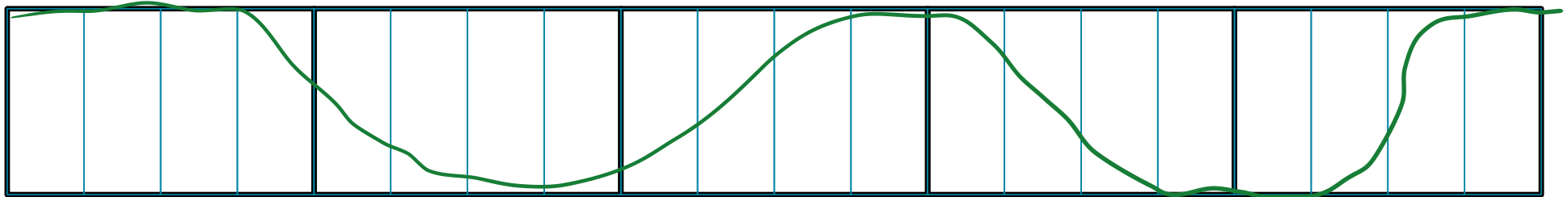
(2) USART H/W: Protocol Agreement

- Transmitter and receiver must agree on the **SAME PROTOCOL**
 - Order of data bits
 - Number of data bits
 - What a start bit is (1 or 0)
 - What a parity bit is (even, odd, none)
 - What a stop bit is (1 or 0)
 - How long a bit lasts
- Transmitter and receiver clocks must be similar, as the beginning of the start bit is the only reference



(2) USART H/W: Input Data Oversampling

- When receiving, USART *oversamples* incoming data line
 - Extra samples allow voting, improving noise immunity
 - Better synchronization to incoming data, improving noise immunity

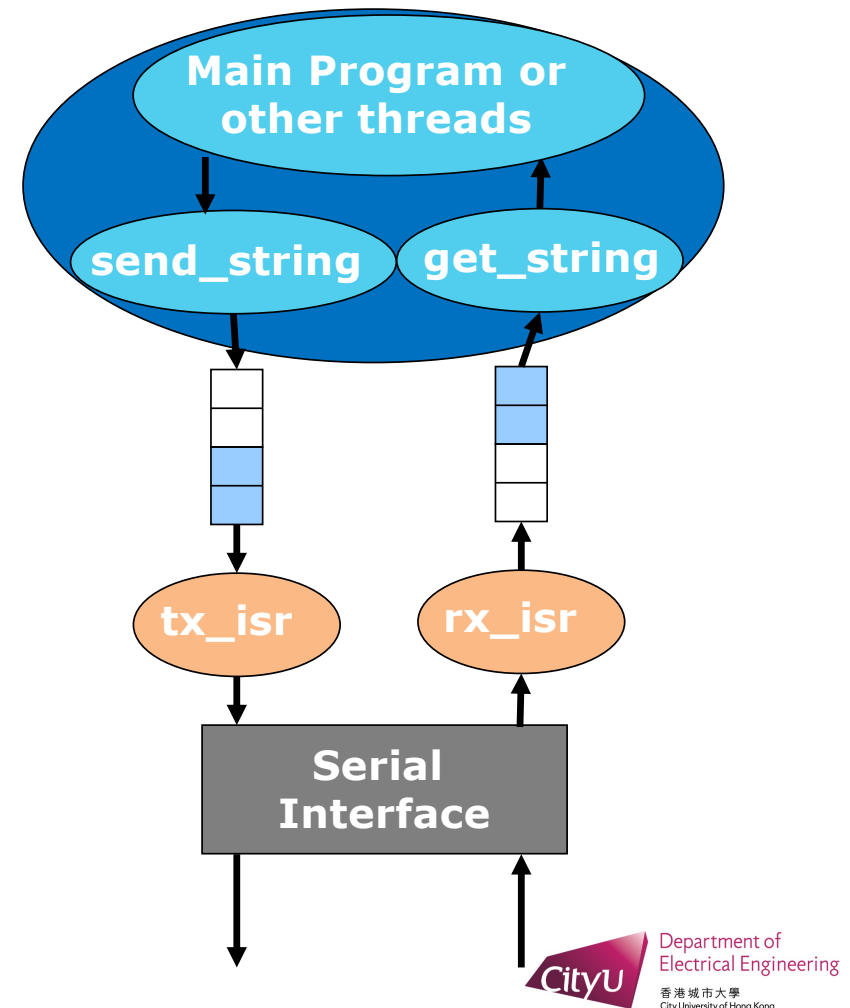


(2) USART S/W: Software Perspectives

- Communication is **ASYNCHRONOUS** to Program
 - Don't know what code the Program is executing:
 - When will the next item arrive?
 - When will the current outgoing item complete transmission?
 - When will an error occur?
 - Need to synchronize between Program and serial communication interface
- Design Options:
 - Polling (wait until data is received): Simple, inefficient use of processor time
 - Interrupt (CPU interrupts Program when data is received): Efficient, but more complex

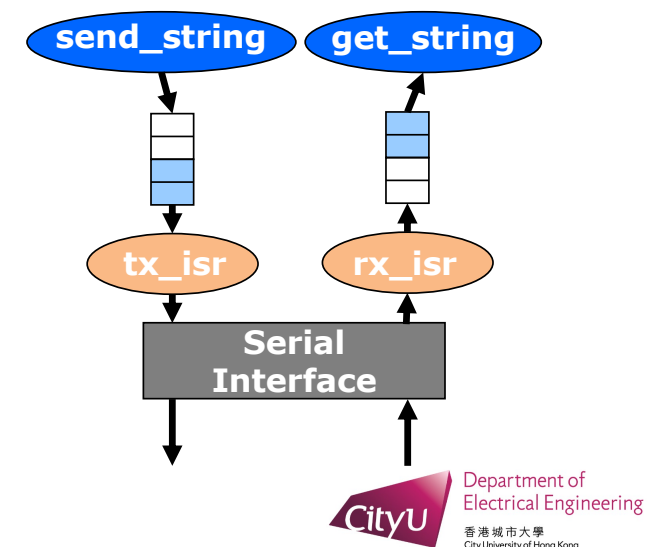
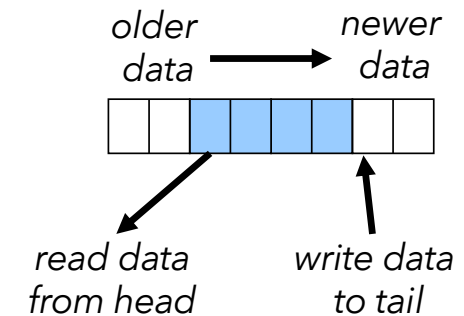
(2) USART S/W: Serial Communications & Interrupts

- Provide multiple threads of control in the program
 - Main program and associated subroutines
 - Transmit ISR: Executes when the interface is ready to send another character
 - Receive ISR: Executes when the interface receives a character
 - Error ISR(s): Executes if an error occurs
- Need a way of buffering information between threads
 - Circular queue with head and tail pointers
 - One for transmit buffer and one for receive buffer



(2) USART S/W: Code to Implement Queues

- Enqueue at tail: tail is the index of the next free entry
- Dequeue from head: head is the index of the item to remove
- Queue size is initialized and stored in size
- One queue per direction
 - Transmit ISR unloads `tx_q`
 - Receive ISR loads `rx_q`
- Other threads load `tx_q` and unload `rx_q`
- Need to wrap pointer at end of buffer to make it circular,
 - Use `&` (bitwise and) if queue size is a power of 2
 - Use `%` (mod, remainder) operator if not a power of 2
- Queue is empty if: `(tail == head)`
- Queue is full if: `((tail + 1) % size) == head)`



(2) USART S/W: Defining the Queues (TX, RX)

```
typedef struct {  
    uint8_t *data; //!< Array of data, stored on the heap.  
    uint32_t head; //!< Index in the array of the oldest element.  
    uint32_t tail; //!< Index in the array of the youngest element.  
    uint32_t size; //!< Size of the data array.  
} Queue;
```

(2) USART S/W: Initialization and Status Inquiries

```
int queue_init(Queue *queue, uint32_t size) {
    queue->data = (uint8_t*) malloc(sizeof(uint8_t) * size);
    queue->head = 0;
    queue->tail = 0;
    queue->size = size;
    return (queue->data != 0); // return malloc result
}

int queue_is_full(Queue *queue) {
    return ((queue->tail + 1) % queue->size) == queue->head;
}

int queue_is_empty(Queue *queue) {
    return queue->tail == queue->head;
}
```

(2) USART S/W: Enqueue and Dequeue

```
int queue_enqueue(Queue *queue, uint8_t item) {
    if (!queue_is_full(queue)) {
        queue->data[queue->tail++] = item;
        queue->tail %= queue->size;
        return 1;
    }
    return 0;
}

int queue_dequeue(Queue *queue, uint8_t *item) {
    if (!queue_is_empty(queue)) {
        *item = queue->data[queue->head++];
        queue->head %= queue->size;
        return 1;
    }
    return 0;
}
```

(2) USART S/W: Using the Queues

- Sending data:
 - `queue_enqueue(&queue, c)`
- Receiving data:
 - `queue_dequeue(&queue, &c)`

(2) USART S/W: Use Model

- When can we **TRANSMIT** data?
 - Transmit peripheral must be ready for data
 - Can poll the status register
 - Or we can use an interrupt, in which case we will need to queue up data
- When can we **RECEIVE** a byte?
 - Receive peripheral must have data
 - Can poll the status register
 - Or we can use an interrupt, and again we will need to queue the data

(2) USART S/W: Software for Polling Serial Port

```
void test_polled() {  
    uart_init(9600);  
    uart_enable();  
  
    while(1) {  
        uart_tx(uart_rx()); // echo back the received character  
    }  
}
```

(2) USART S/W: Display Data on LCD

```
// Assume LCD screen is 8x2 character LCD
line = col = 0;
while (1) {
    c = uart_rx();
    lcd_set_cursor(col, line);
    lcd_put_char(c);
    col++;
    if (col > 7) {
        col = 0;
        line++;
        if (line > 1) {
            line = 0;
        }
    }
}
```

(2) USART S/W: Interrupt-Driven S/W

- Use interrupts:
 - 1st Step: Initialize peripheral to generate interrupts
 - 2nd Step: Create an ISR with a receive callback
 - 3rd Step: Enable the peripheral

(2) USART S/W: Interrupt Handler

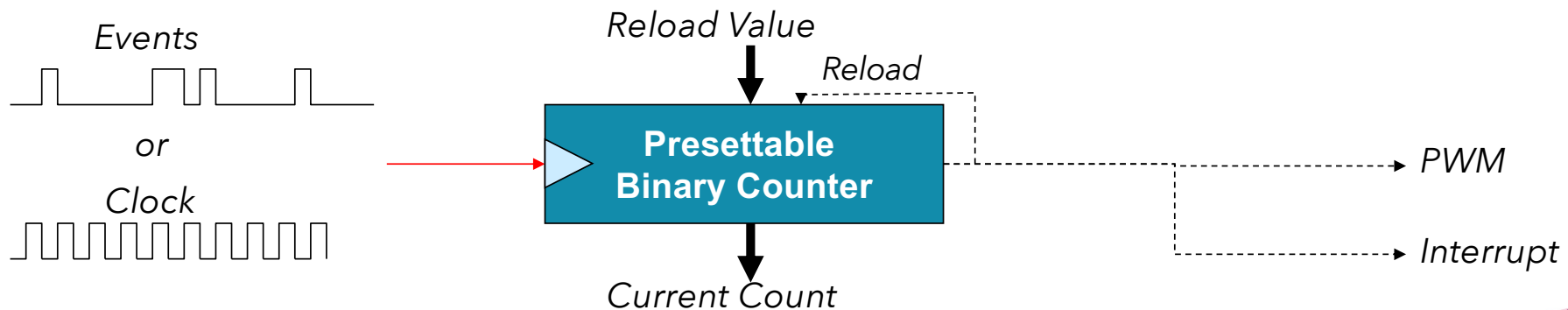
```
Queue rx_queue;

void uart_rx_isr(uint8_t rx) {
    // Store the received character
    queue_enqueue(&rx_queue, rx);
}

int main() {
    queue_init(&rx_queue, 128);
    uart_init(9600);
    uart_set_rx_callback(uart_rx_isr);
    uart_enable();
    ...
}
```

(3) Timer Peripheral: Introduction

- Common peripheral for MCU
- Based on presetable binary counter, enhanced with configurability
 - Count value can be read and modified by MCU
 - Count direction can be set to UP or DOWN
 - Counter's clock source can be selected
 - Counter's overflow/underflow action can be selected
 - Generate interrupt or reload counter or toggle output



(3) Timer Peripheral: Types of Timer Peripherals

1. Interrupt Timer

- Generates periodically interrupts or trigger DMA transfers

2. Timer Controlled I/O Module

- Connected to I/O pins, has input capture and output compare support
- Generates PWM signals and interrupt requests

3. Real-Time Clock

- Powered by external 32.768 kHz crystal to track elapsed time in seconds
- Sets alarm or generates 1Hz output signal and wakes up system with interrupt

4. SYSTICK

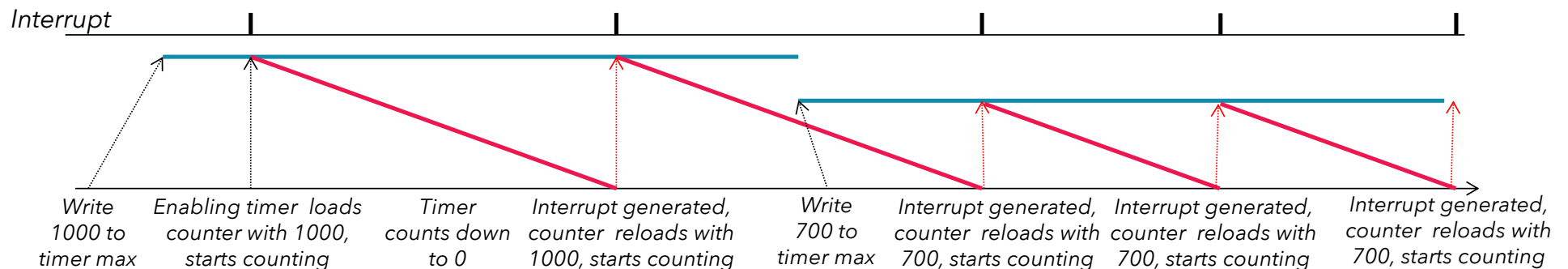
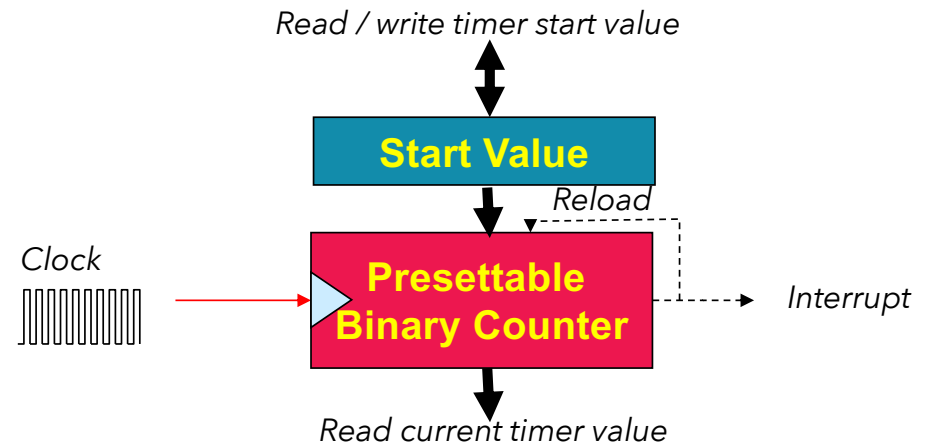
- Part of CPU core's peripherals
- Generates periodic interrupt

(3) Timer Peripheral: Timer Software Configuration

- Setup timer, set to tick at 10 Hz
 - `timer_init(CLK_FREQ / 10);`
- Set interrupt
 - `timer_set_callback(timer_isr);`
- Enable module
 - `timer_enable();`
- Disable module
 - `timer_disable();`

(3) Timer Peripheral: Interrupt Timer

- Load start value from register
- Counter counts down with each clock pulse
- When timer value reaches zero
 - Generates interrupt
 - Reloads timer with start value



(3) Timer Peripheral: Timer Controlled I/O Module

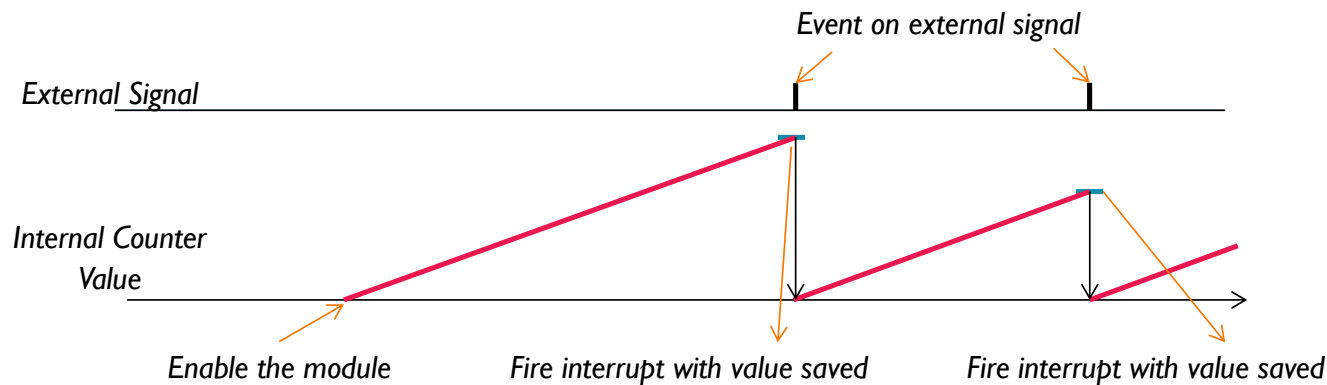
- Core Counter
 - Clock Options: External or Internal
 - Pre-scaler to divide clock
 - Can reload with set value, or overflow and wrap around
- Number of Channels ("N" Channels) per Module
 - 3 Timer Modes:
 - Capture Mode (IN): Capture timer's value when input signal changes
 - Output Compare (OUT): Change output signal when timer reaches certain value
 - PWM (OUT): Generate pulse-width-modulated signal.
 - Possible triggering of interrupt, hardware trigger on overflow
 - One I/O pin per Channel

(3) Timer Peripheral: I/O Modes

- Input Capture Mode (IN)
 - Capture timer's value when input signal changes
 - Rising edge, falling edge, both
 - How long after I started the timer did the input change?
 - Measure time delay
- Output Compare Mode (OUT)
 - Modify output signal when timer reaches specified value
 - Set, clear, pulse, toggle (invert)
 - Make a pulse of specified width or after specified delay
- Pulse Width Modulation (OUT)
 - Make a series of pulses of specified width and frequency

(3) Timer Peripheral: Input Capture Mode

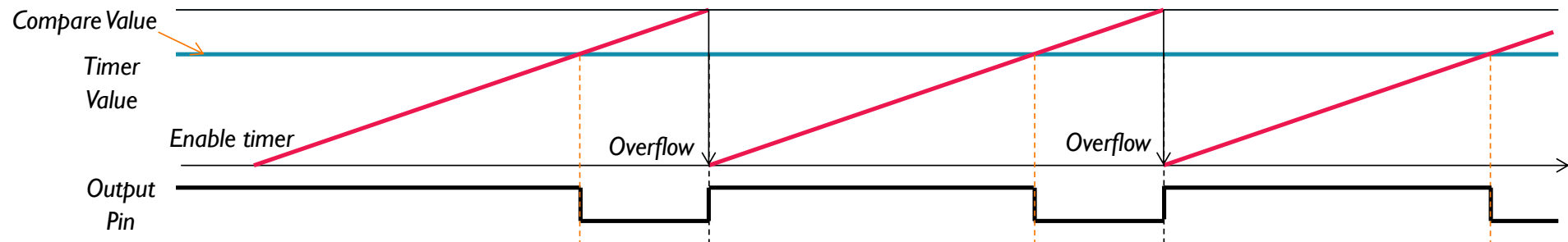
- I/O pin operates as input on edge
- When valid edge is detected on pin...
 - Current value of counter is stored
 - Interrupt is activated



(3) Timer Peripheral: PWM Mode

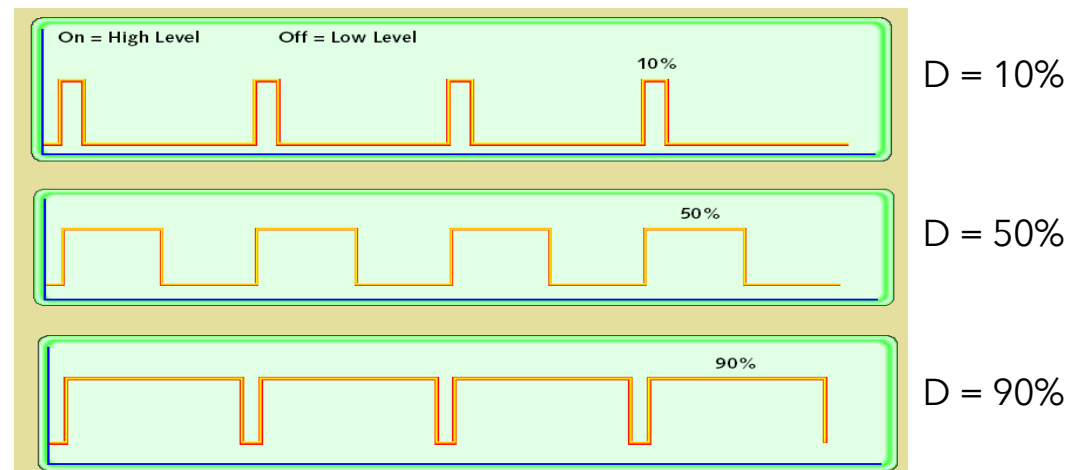
- PWM duty cycle proportional to compare value
 - Period = max timer value
 - Pulse width = compare value

$$\text{Duty Cycle } (D) = \frac{\text{Compare Value}}{\text{Max Value}} \cdot 100\%$$



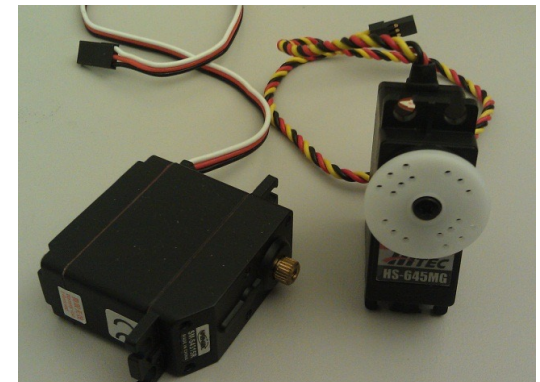
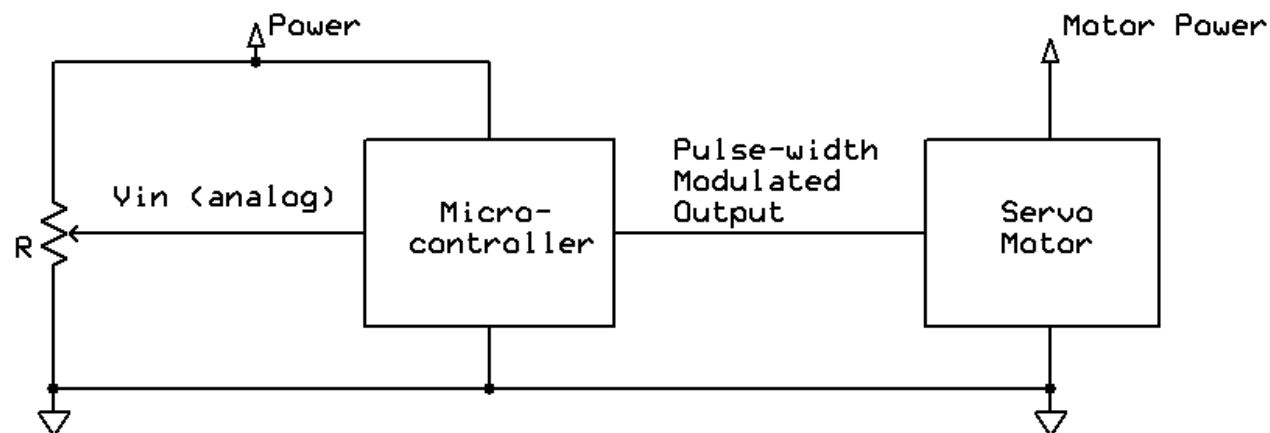
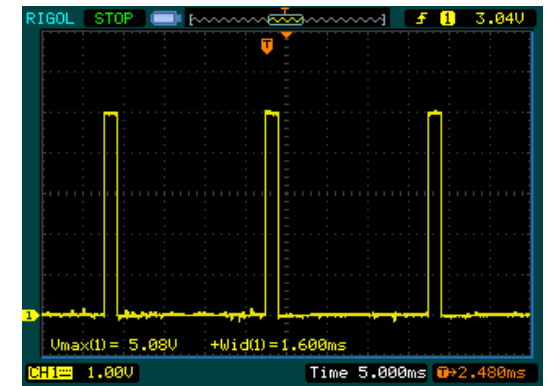
(3) Timer Peripheral: Pulse-Width Modulation (PWM)

- PWM Use Models
 - Digital Power Amplifiers are more efficient than Analog Power Amplifiers (**CLASS-D AMPLIFIER**)
 - Applications: Motor Control, Inverter, SMPS (Buck/Boost/Flyback/Forward), DAC, Dimmer
 - Load responds slowly to averaged PWM signal
 - Digital communication is less sensitive to noise than analog methods
 - PWM provides a digital encoding of an analog value
 - Less vulnerable to noise
- PWM Signal Characteristics
 - Modulation Frequency (f)
 - Period = $1/f$
 - T_{ON} vs T_{OFF}
 - Duty-cycle (D) = $T_{ON} / (T_{ON} + T_{OFF})$
- Power Converter Examples:
 - $V_{OUT} = \sim V_{IN} \times D$ (Buck Converter)
 - $V_{OUT} = \sim V_{IN} / (1-D)$ (Boost Converter)



(3) Timer Peripheral: PWM Driving Servo Motor

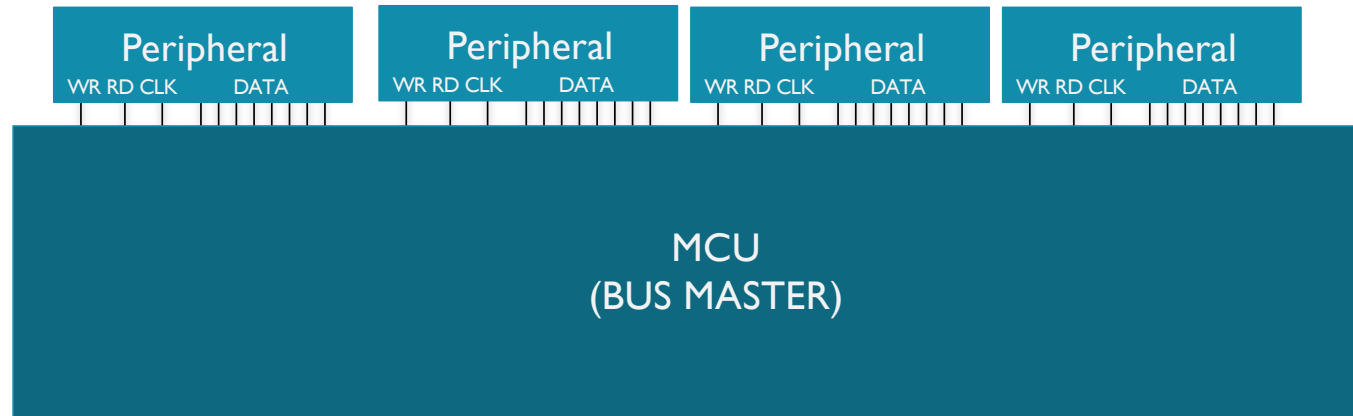
- MCU PWM signal
 - Realtime Servo Motor Control (Servo: Slave)
 - PWM period = 20ms
 - Pulse width varies from 1ms to 2ms
 - Duty cycle varies from 5% to 10%



Overview

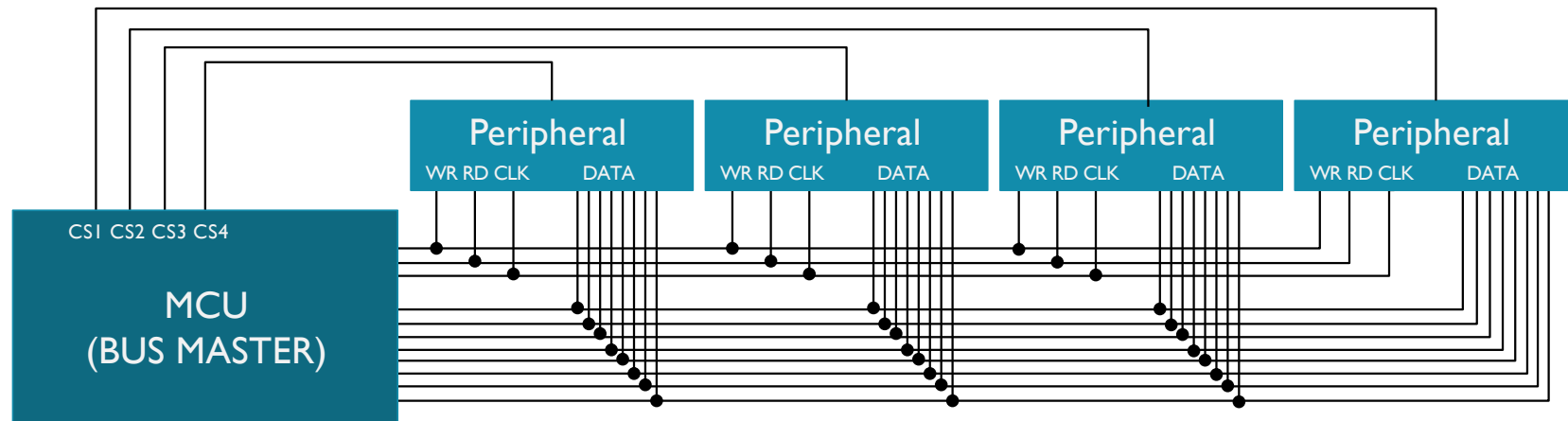
- Quiz 2 Results
- SoC Components & Integration
- SoC Peripherals (GPIO, UART, Timer)
- SoC Bus Architectures
- DMA Controllers for Efficient Data Transfer

Individual Parallel Bus Architecture



- Dedicated point-to-point connections
 - Parallel DATA, RD/WR, CLK lines between MCU (**MASTER**) and each Peripheral (**SLAVE**)
 - Fast and Simultaneous Data Transfers
 - Requires many Connections: Does **NOT SCALE** with too many Peripherals

Shared Parallel Bus Architecture



- All devices use buses to share data, read, write, clock signals
- There are fewer data lines in a shared parallel bus architecture
- Master uses individual chip select lines (CS1, CS2, CS3, CS4) to address each Slave
- Master can communicate with only one Slave at a time
- **PERFORMANCE** is proportional to **DATA WIDTH** and **BUS SPEED**
- **BUS ARBITRATION** is needed if there are **MULTIPLE MASTERS**

Shared Parallel Bus Architecture (Bus Master)

```
module bus_master (
    input clk, input rst, input [1:0] addr, input [7:0] write_data, input write_en, input read_en,
    output reg [7:0] read_data, output reg [3:0] cs, output reg wr, output reg rd,
    inout [7:0] data_bus
);
    reg [7:0] data_out;
    wire [7:0] data_in;

    assign data_in = data_bus;
    assign data_bus = (write_en) ? data_out : 8'bz;
    always @(*) begin
        cs = 4'b0000;
        cs[addr] = 1'b1;
    end

    always @(posedge clk or posedge rst) begin
        if (rst) begin
            wr <= 0; rd <= 0; data_out <= 8'b0; read_data <= 8'b0; // BUS IDLE
        end else begin
            if (write_en) begin
                wr <= 1; rd <= 0; data_out <= write_data;
            end else if (read_en) begin
                wr <= 0; rd <= 1; read_data <= data_in;
            end else begin
                wr <= 0; rd <= 0;
            end
        end
    end
end
endmodule
```

// COMBINATIONAL LOGIC

// TRI-STATE BUS

// CHIP SELECT

// SEQUENTIAL LOGIC

// BUS WRITE

// BUS READ

// BUS IDLE

Shared Parallel Bus Architecture (Simple Bus Slave)

```
module bus_slave (
    input clk,
    input cs,
    input wr,
    input rd,
    inout [7:0] data_bus
);

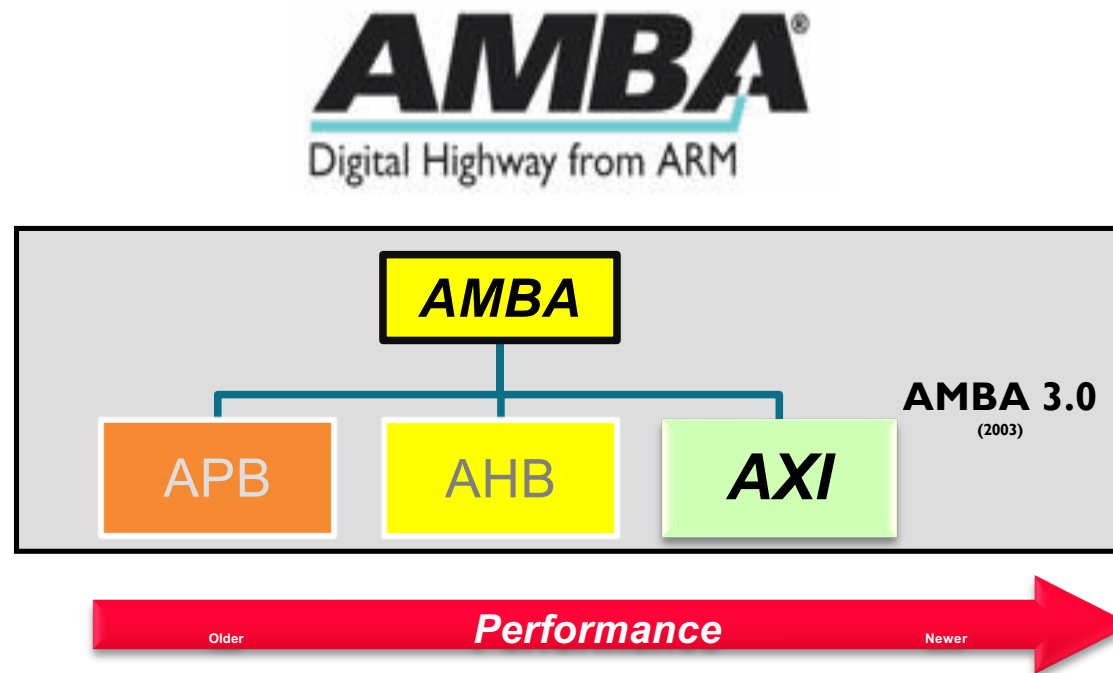
    reg [7:0] mem; // READ / WRITE ACCESS TO "mem"
    wire [7:0] data_in;
    reg [7:0] data_out;

    assign data_in = data_bus; // COMBINATIONAL LOGIC
    assign data_out = (cs && rd) ? data_out : 8'bz;

    always @(posedge clk) begin // SEQUENTIAL LOGIC
        if (cs && wr) begin
            mem <= data_in;
        end
    end

    always @(posedge clk) begin
        if (cs && rd) begin
            data_out <= mem;
        end
    end
endmodule
```

ARM SoC Bus Architectures: AMBA



AMBA: Advanced Microcontroller Bus Architecture
AXI: Advanced Extensible Interface

History on Advanced Microcontroller Bus Architecture

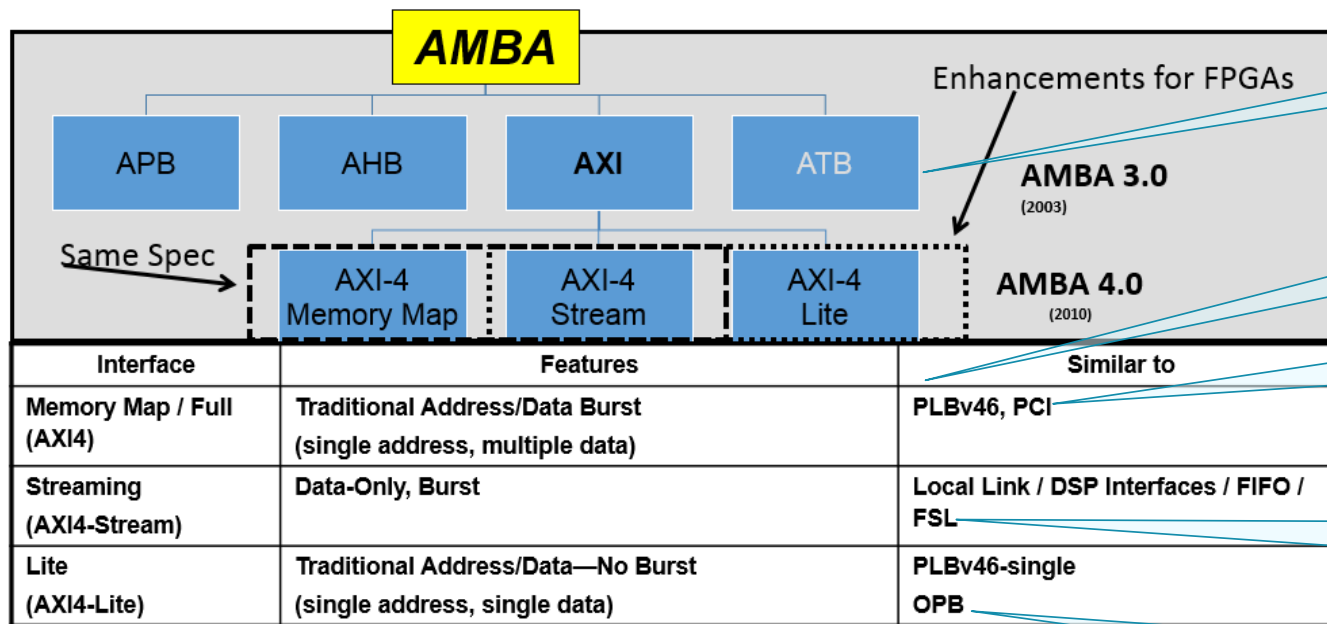
- AMBA 1 (Early version)
 - ASB (Advanced System Bus) and APB (Advanced Peripheral Bus)
- AMBA 2 (1999)
 - AHB (Advanced High-Performance Bus)
- AMBA 3 (2003)
 - AXI (Advanced eXtensible Interface)
- AMBA 4 (2010)
 - AXI4 and ACE (AXI Coherency Extensions)
- AMBA 5 (2013 onwards)
 - CHI (Coherent Hub Interface)

Comparison of AXI, AHB, APB

FEATURE	AXI	AHB	APB
Full Name	Advanced eXtensible Interface	Advanced High-Performance Bus	Advanced Peripheral Bus
Protocol Complexity	High	Medium	Low
Multiple Master Support	Yes	Yes	No
Multiple Outstanding Transactions	Yes	No	No
* Burst Support	Yes (up to 256 beats)	Yes (4/8/16 beats)	No
Pipelined	Yes (multiple channels)	Yes (address/data phase)	No
Clock Domain Support	Supports multiple clock domains	Single clock domain	Single clock domain
Use Case	High-speed data transfer	General memory access	Low-speed peripherals
Example Peripherals	DDR, DMA, GPU	SRAM, Flash	UART, GPIO, Timer

* Data transferred in one transaction = number of beats x data width

AXI as Part of AMBA Specifications



AMBA Trace Bus (ATB) is a point-to-point unidirectional bus protocol to transport trace data

Processor Local Bus (PLB) is an IBM bus protocol for PowerPC based embedded systems

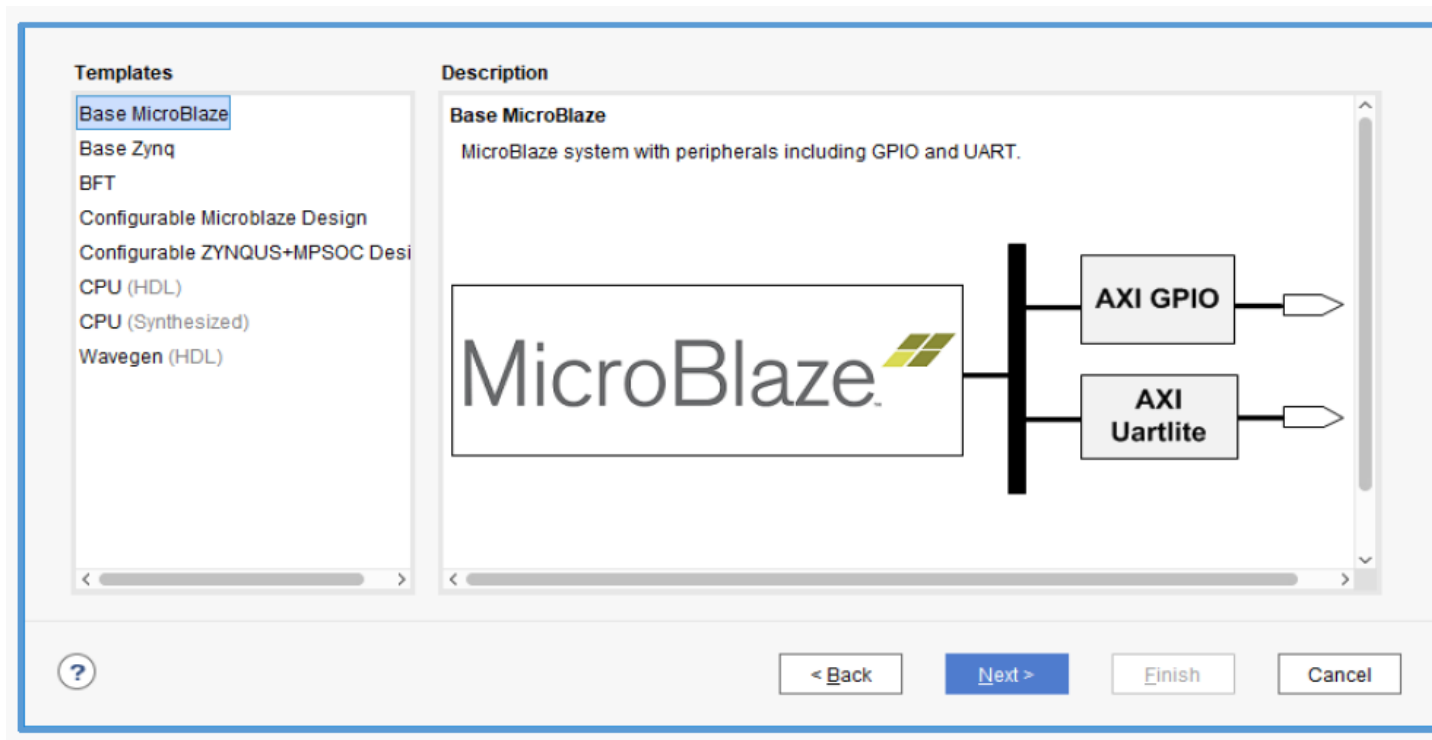
Peripheral Component Interconnect (PCI) is a standard interface to connect PC peripheral cards

Fast Simplex Link (FSL) is a point-to-point unidirectional bus protocol used in Xilinx embedded systems

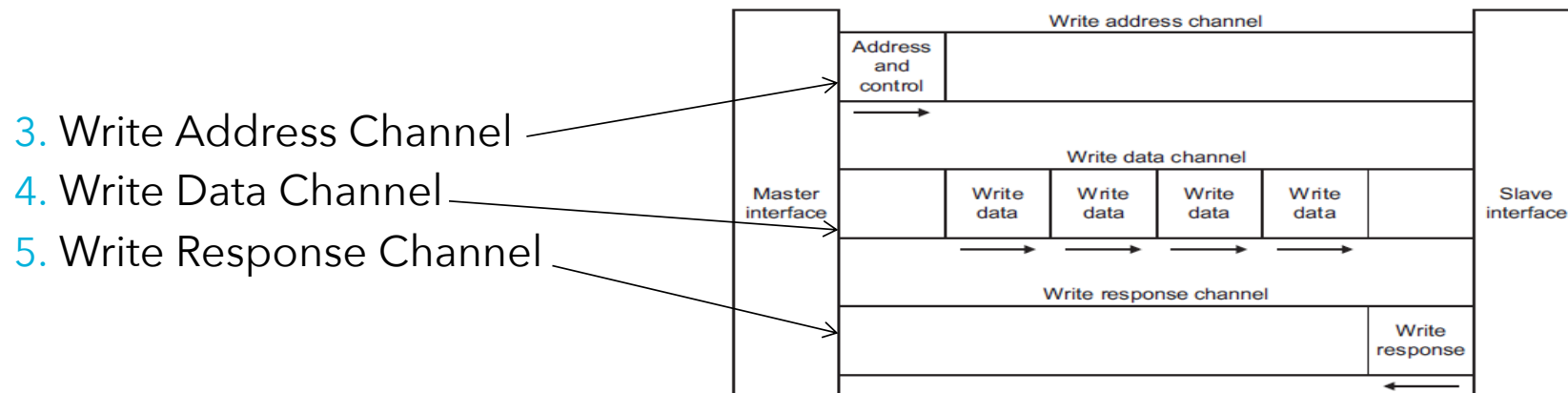
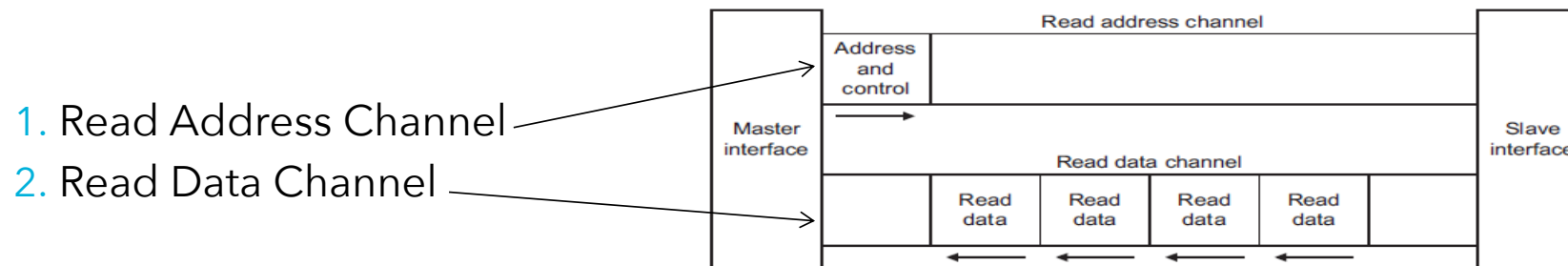
On-Chip Peripheral Bus (OCP) is medium speed bus for IBM PowerPC based embedded systems

MicroBlaze and AXI Bus Interface

- MicroBlaze is Xilinx's 32-bit RISC soft processor core, optimized for embedded applications on Xilinx devices.

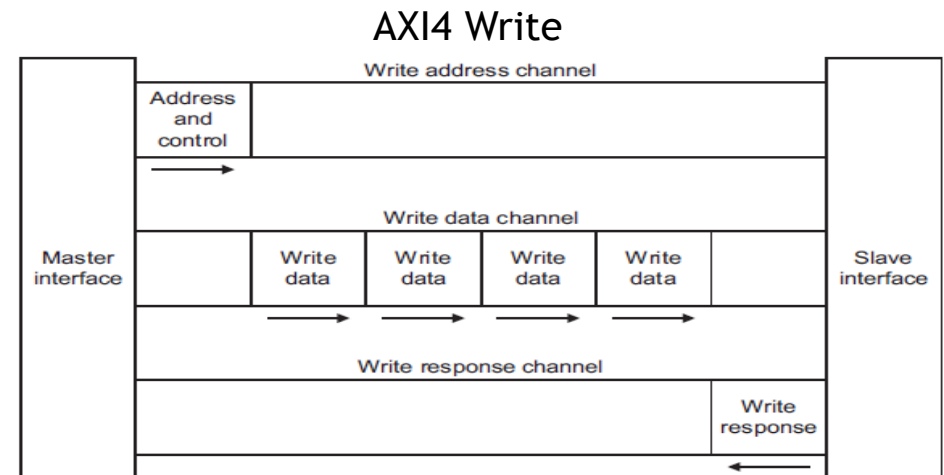
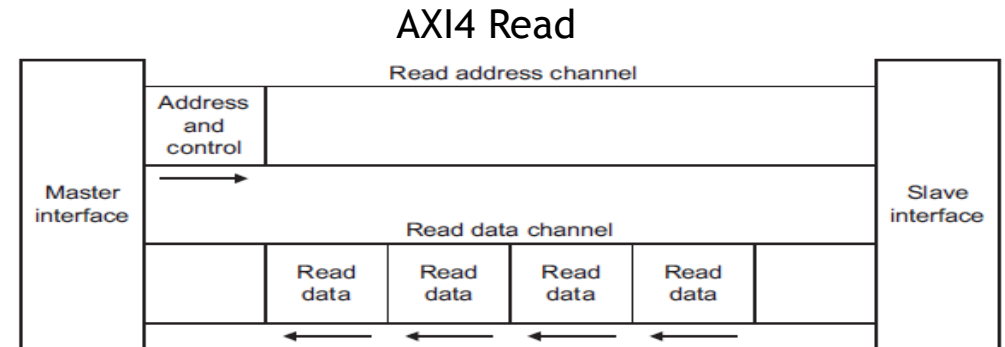


Basic AXI Signaling: 5 Channels



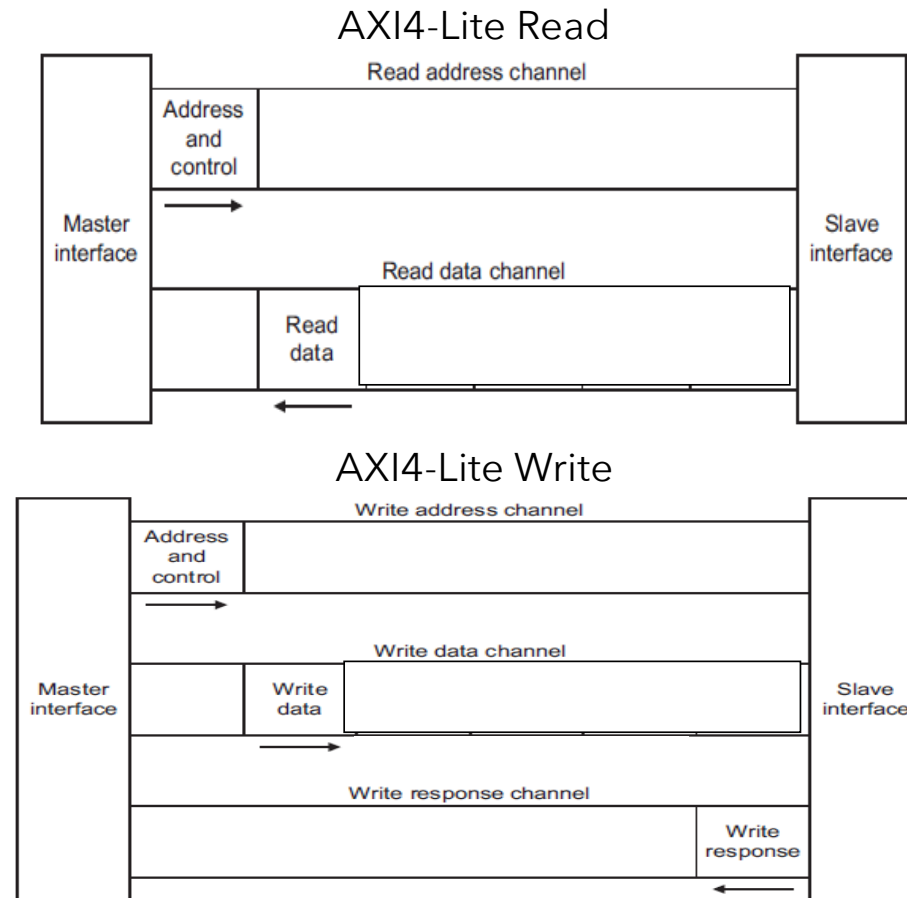
AXI Interface: AXI4

- Also Known As (AKA):
 - "Full AXI" or "AXI Memory Mapped"
- Single Address - Multiple Data
 - Support multiple data items (beats)
- Parameterizable Burst
 - 1-256 data beats
- Parameterizable Data Width
 - 32-1024 bits



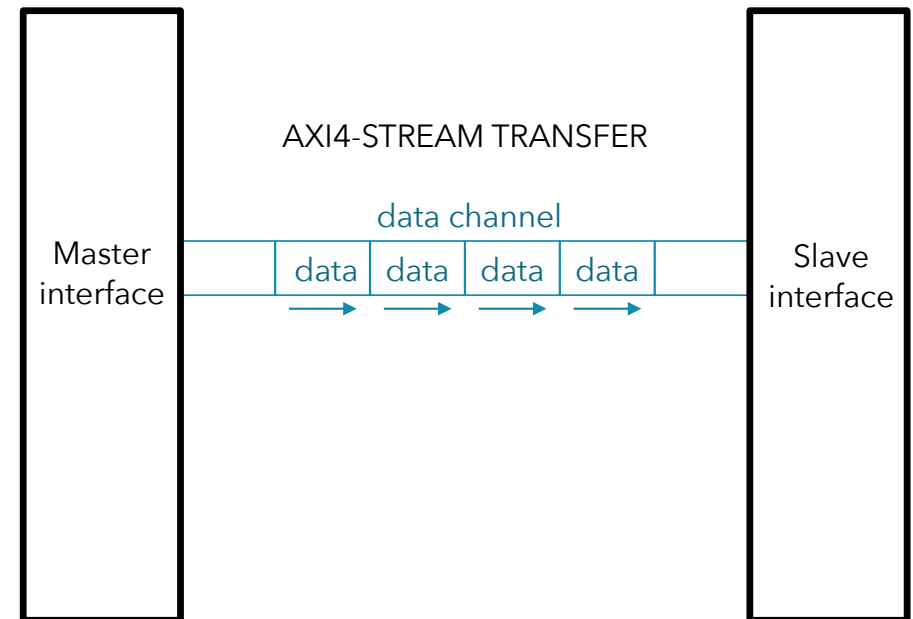
AXI Interface: AXI4-Lite

- **NO BURST**
- Data width 32 or 64 bits only
 - Xilinx IP supports 32-bits
- Efficient
 - Very small footprint
- AXI Interconnect
 - Bridging AXI4-Lite to AXI4



AXI Interface: AXI4-Stream

- No address channel, no read and write, always just master to slave
 - Effectively an AXI4 “write data” channel
- Unlimited burst length
 - AXI4 max: 256
 - AXI4-Lite: No burst
- Same signaling as AXI Data Channels
 - Allows merging, packing, width conversion
 - Supports sparse, continuous, aligned, unaligned streams



Streaming Applications

- AXI-Stream may not have packets
 - EX: Digital Up Converter (DUC), Digital Down Converter (DDC) in DSP (communications)
 - No concept of address
 - Free-running data
 - AXI4-Stream is optimized for a simple interface
- AXI-Stream may have packets
 - For example, AXI-Stream may represent PCIe traffic or Ethernet MAC traffic
 - These packets may contain different information
 - Usually, a bridge logic is needed

Overview

- Quiz 2 Results
- SoC Components & Integration
- SoC Peripherals (GPIO, USART, Timer)
- SoC Bus Architectures
- DMA Controllers for Efficient Data Transfer

What is a DMA Controller (DMAC)?

- Direct Memory Access Controller (DMAC):
 - H/W block that moves data between memory and peripherals **WITHOUT INVOLVING CPU** for each transfer.
- Key Purpose:
 - Offloads repetitive and high-speed data transfers from CPU
 - Improves system performance and power efficiency.
- Main Features:
 - Can act as bus master (initiates memory transactions)
 - Typically configured by the CPU via a slave interface
 - Supports various modes (single, burst, circular, scatter-gather)
 - Often used with high-throughput peripherals: USART, SPI, ADC, DAC

Typical DMA Use Cases

USE CASE	DESCRIPTION
UART RX/TX with DMA	CPU-free serial communication using interrupt or polling
Memory-to-Memory Transfer	Fast data copying without CPU load
ADC to RAM (Data Logging)	DMA moves sampled data from ADC into RAM buffer
RAM to DAC (Signal Output)	Real-time audio/video playback without CPU interruptions
SPI/Ethernet Packet Transfers	Receive/transmit packet buffers efficiently

DMA Transfer Modes

1. REGISTER-BASED DMA Controller:

- Single Transfer: Transfer one data item at a time
- Burst Transfer: Transfer multiple consecutive items per request
- Circular Transfer: Auto-restart after completion

2. SCATTER-GATHER DMA Controller:

- Linked List: Transfer a long list of non-consecutive items per request
- Supports **NON-CONTIGUOUS** memory writes and memory reads

How Register-Based DMAC Works?

1. CPU Configures DMA Controller **REGISTERS**:
 - **SOURCE ADDRESS**
 - **DESTINATION ADDRESS**
 - **TRANSFER SIZE & DIRECTION**
2. DMA Starts Transfer:
 - May be triggered by software, timer, or peripheral event (e.g. USART TX ready)
3. DMA Performs Data Transfer:
 - Reads from source, writes to destination
 - Uses AXI or AHB bus as master
4. Completion Handling:
 - DMA generates interrupt or flag when done
 - CPU can initiate next transfer or handle results

How Scatter-Gather DMAC Works?

1. CPU Creates **DESCRIPTOR LIST**:
 - Each entry contains: **SOURCE**, **DESTINATION**, **SIZE**, and **CONTROL FLAGS**
 - Stored in memory (linked list or array)
2. DMA Reads Descriptors:
 - Automatically follows the list
 - Fetches one descriptor after another without CPU intervention
3. DMA Transfer:
 - Handles multiple **NON-CONTIGUOUS** blocks
 - Useful for video/audio buffers, file I/O, or multi-frame streaming
4. Interrupt at End:
 - Optional interrupt after entire chain completes
 - Reduces CPU load significantly compared to polling per block

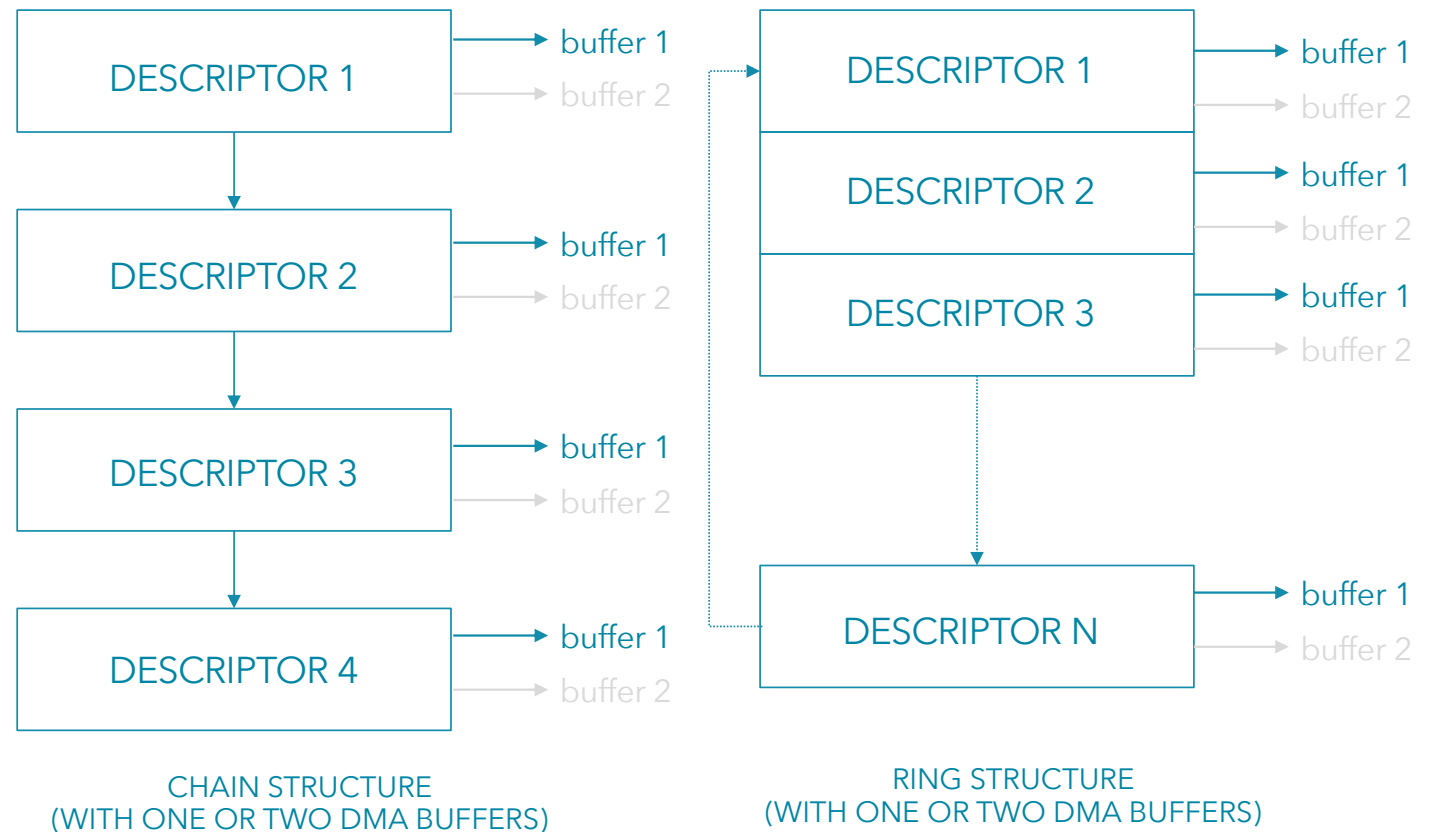
Descriptor and Descriptor List Structure

- Descriptor Structure

- Next Descriptor Pointer
- Buffer 0 Address
- Control (Buffer 0)
- Buffer 1 Address
- Control (Buffer 1)
- Status
- Metadata

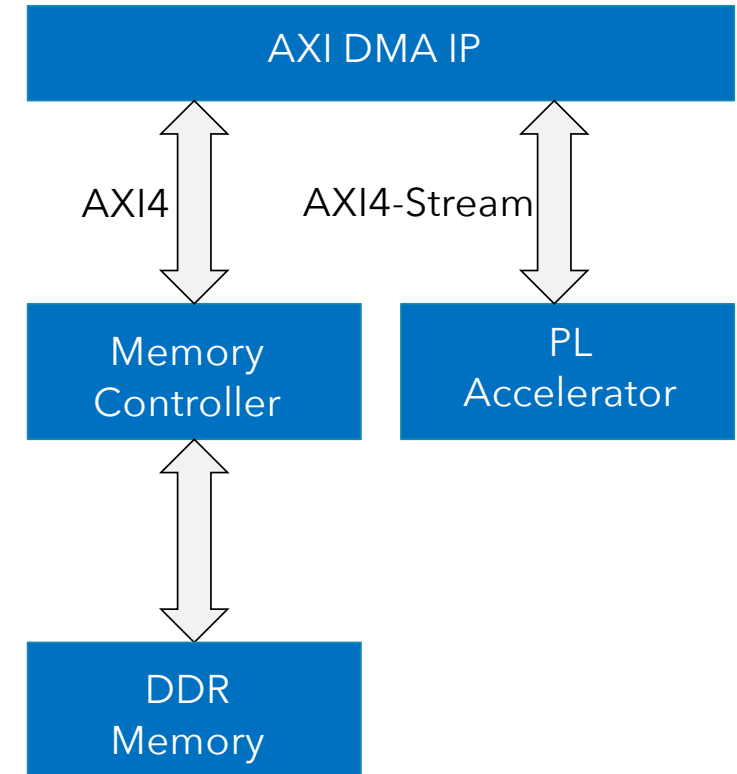
- Descriptor List Structure

- Chain, Ring, Flat
- Table, Tree



DMA in Action: Xilinx Zynq SoC DMA

- In Xilinx Zynq SoCs, DMA is commonly implemented using AXI DMA or Central DMA (CDMA) IP blocks.
- **AXI DMA** Features:
 - Connects to AXI interconnect
 - Supports **MEMORY-TO-STREAM** and **STREAM-TO-MEMORY** DMA transfers
 - Works with AXI4-Stream for high-speed peripherals
 - Can be configured by PS
- **CDMA** Features:
 - Simple **MEMORY-TO-MEMORY** DMA
 - Uses AXI4 full memory-mapped interface
 - Useful for general data copying within memory



Summary

- Reviewed Quiz 2 Results
- Reviewed SoC Components and Peripheral Integration
- Studied USART Hardware and Software
- Studied Timer Peripheral
- Studied AMBA buses
- Studied DMA transfers