COMPSYS 303: Microcomputers and Embedded Systems

Lecture Notes

Nicholas Russell

Saturday, October 5, 2024

Table of contents

Introduction

These notes provide a detailed breakdown of the lectures for COMPSYS 303. Each lecture is represented in its own document, allowing for modular learning and easy reference.

Lectures Included

Lecture 1: Concurrency Overview of concurrency in embedded systems, challenges like race conditions and priority inversion, and real-world examples.

Lecture 2: SCCharts and Synchronous Concurrency Detailed discussion on SCCharts, the synchronous approach, and cyber-physical systems.

Lecture 3: Peripherals and Interfacing Types of I/O, data transfer mechanisms, and interrupt handling in embedded systems.

Lecture 4: Embedded Processors Overview of processor design in embedded systems, FSMD, types of processors, and synthesis from C code.

Lecture 5: SoC and SOPC Buses Discussion of bus design issues, synchronous vs. asynchronous buses, and bus arbitration.

Lecture 6: Embedded Control Systems 1 Real-time systems, sampling and quantization, aliasing, and worst-case execution time.

Lecture 7: Embedded Control Systems 2 Interaction with real-world peripherals, including PWM, stepper motors, and ADC.

Lecture 8: Embedded Control Systems 3 Control of physical processes, key metrics in control systems, and examples like cruise control.

Lecture 9: Industrial Automation 1 Introduction to industrial automation, PLCs, HMIs, and generations of automation.

Lecture 10: Industrial Automation 2 IEC 61131 standard, ladder logic features, and an example of a pneumatic cylinder control sequence.

Part I

Partha's Lectures

Lecture 1: Concurrency

1.1 Introduction to Embedded Systems and Concurrency

Embedded systems are a specialized class of computers designed to continuously interact with their environment in a real-time, often reactive manner. Unlike general-purpose computers, embedded systems are purpose-built to handle specific tasks under stringent conditions, including timing constraints. These systems are embedded as integral parts of larger devices and must operate dependably, often without human intervention. Examples include automotive control systems, medical devices like pacemakers, and consumer electronics such as washing machines.

When deadlines must be met without exception, the system is termed a **hard real-time system**; otherwise, it is a **soft real-time system**. In hard real-time systems, any missed deadline could result in catastrophic failure, such as in medical devices or automotive safety features. In contrast, soft real-time systems can tolerate occasional missed deadlines, although they may affect performance.

The concept of **concurrency** is crucial in embedded systems because these systems often involve multiple processes interacting simultaneously, such as a pacemaker continuously monitoring the heart and delivering pulses when needed. Concurrency enables these systems to effectively manage different tasks that need to be performed either synchronously or asynchronously. For example, while monitoring the heart, a pacemaker must also manage pulse delivery in real-time, demonstrating the need for concurrent tasks.

1.2 Key Concepts in Concurrency

Concurrency refers to multiple components or processes operating at the same time within a system. In embedded systems like pacemakers, various components (controllers, sensors, and timers) work concurrently to ensure precise timing between events. Concurrency becomes especially challenging when different tasks access shared resources simultaneously, potentially leading to issues such as **race conditions**, **deadlocks**, and **livelocks**.

- **Race Conditions:** These occur when two or more processes access shared data concurrently, and the final result depends on the timing of their access. A typical example is the producer-consumer problem, where one process generates data while another consumes it. If these processes do not manage shared data properly, inconsistencies may arise. For instance, if two processes increment a shared counter without proper synchronization, the final value of the counter may be incorrect due to overlapping access. This issue is exacerbated in systems with multiple cores or processors, where different threads can execute simultaneously.
- **Critical Sections and Mutual Exclusion:** A **critical section** is a part of a program that should only be executed by one process at a time to avoid data corruption. Ensuring **mutual exclusion**, where no two processes are in their critical sections simultaneously, is essential for preventing race conditions. Mutual exclusion can be achieved using synchronization mechanisms like **locks**, which ensure that only one process can access the critical section at any given time. Common techniques include **mutexes** and **binary semaphores**, both of which can lock shared resources to ensure orderly access. It is also important to minimize the time a process spends in a critical section to reduce the chances of blocking other processes.

Deadlocks and Livelocks:

- **Deadlocks** occur when processes are waiting on each other indefinitely, leading to a situation where no progress can be made. This typically happens in resource allocation scenarios where two or more processes form a circular chain of dependencies. A classic example is two processes holding a resource that the other needs to proceed, resulting in a stalemate.
- **Livelocks** are similar to deadlocks but differ in that processes keep changing state in response to each other without making any actual progress. Both issues are critical in concurrent systems and need to be addressed through careful design, such as avoiding circular waiting conditions, implementing resource hierarchies, or using **deadlock detection algorithms** that can detect and break deadlocks. Timeout mechanisms can also be used to release resources if a process waits too long.

1.3 Approaches to Manage Concurrency

Concurrency in embedded systems is managed through a combination of hardware and software synchronization mechanisms. Some of the methods discussed include:

Busy Waiting and Non-Busy Waiting:

- **Busy waiting** occurs when a process remains in the ready state, actively checking for a condition to be satisfied. This approach is inefficient in terms of CPU utilization, as the CPU is occupied with checking instead of performing useful work.
- **Non-busy waiting** solutions, such as putting a process in the blocked state while waiting for a resource, are more efficient for embedded systems where CPU resources are limited. Busy waiting is often used in systems where the wait time is expected to be very short, whereas non-busy waiting is preferred for longer wait times to free up CPU resources. In embedded systems, avoiding busy waiting is crucial because the CPU often needs to manage multiple tasks under strict timing constraints.
- **Test and Set Instruction:** The **Test and Set** instruction is an atomic operation used to achieve mutual exclusion by using a global variable to indicate whether a critical section is occupied. By using Test and Set, embedded systems can ensure that processes access shared resources in a predictable and coordinated manner. The atomicity of this instruction prevents race conditions by ensuring that the check-and-set operation is performed without interruption. However, Test and Set can lead to **busy waiting**, which may not be suitable for all embedded applications. In systems with limited processing power or battery life, busy waiting can degrade performance and reduce efficiency.

Semaphores and Mutex Locks:

- **Semaphores** are synchronization mechanisms used to manage concurrent access to shared resources. They allow processes to signal and wait for access, ensuring a controlled flow of execution. Semaphores can be **binary** (acting like a mutex) or **counting**, allowing a specified number of processes to access a resource concurrently. For example, counting semaphores are often used in producer-consumer scenarios where multiple producers or consumers need access to a shared buffer.

- **Mutex locks** are similar but are typically used to protect access to a particular data structure or variable, providing mutual exclusion in critical sections. Mutexes are often simpler to implement but are limited to binary states (locked or unlocked), whereas semaphores offer more flexibility for resource management.

Priority Inversion and Solutions:

- **Priority inversion** occurs when a higher-priority task is waiting for a resource held by a lower-priority task, which can lead to missed deadlines in real-time systems. A well-known example is the **Mars Pathfinder mission**, where a low-priority task holding a shared resource caused a higher-priority task to miss its deadline, resulting in system resets.
- **Solutions** to priority inversion include **priority inheritance**, where the lower-priority task temporarily inherits the higher priority to complete its task and release the resource. Another approach is **priority ceiling**, where a resource is assigned a priority equal to the highest priority of any task that may lock it, thus preventing priority inversion.

1.4 Examples and Applications

The lecture provided various examples of where concurrency is critical, including the automotive industry and robotics. For instance, the **2010 Toyota recall** was attributed to software issues in their anti-lock braking systems, highlighting the importance of deterministic behavior in safety-critical systems. This incident shows how improperly managed concurrency can lead to life-threatening failures. Similarly, **Tesla's accidents** were referenced, emphasizing the challenges of building reliable autonomous systems. These examples underscore the need for proper concurrency management to ensure system reliability and safety.

In embedded systems such as a **pacemaker**, concurrency is an intrinsic characteristic. The pacemaker must concurrently monitor the heart and trigger electrical pulses without delay, thus operating as a real-time reactive system. Concurrency, in this context, is about ensuring different components work seamlessly together to meet life-critical deadlines. The pacemaker example illustrates how timing constraints and concurrent interactions must be carefully managed to avoid potentially fatal outcomes. The system must ensure that sensing, decision-making, and actuation are carried out with precise timing to maintain patient safety.

1.5 Problems with Concurrency and Solutions

Concurrency can lead to serious issues if not properly managed.

Deadlocks

- **Deadlocks** occur when concurrent processes are waiting on each other to release resources, resulting in a state where no process can proceed. To prevent deadlocks, proper scheduling mechanisms must be implemented to ensure that resource allocation does not lead to circular waiting. **Deadlock prevention techniques** include resource ordering, where resources are always requested in a specific order, and **deadlock detection algorithms** that periodically check for cycles in the resource allocation graph. Another approach is **deadlock avoidance**, such as using the **Banker's algorithm**, which allocates resources in a way that ensures a safe state is always maintained.

Fairness and Scheduling

- **Fairness** in resource allocation means that every process eventually gets a chance to execute. Embedded systems often use a **scheduler** to manage processes, ensuring tasks are completed in a timely and fair manner. The role of the scheduler is critical to prevent **starvation**, where some tasks never get executed due to others continuously consuming resources.
- Common **scheduling** algorithms include **round-robin**, **priority-based**, and **rate-monotonic scheduling**, each with its own advantages and trade-offs depending on the system requirements. **Round-robin scheduling** is simple and ensures fairness by giving each process a fixed time slice, while **priority-based scheduling** allows more critical tasks to execute first but can lead to starvation without priority aging.

Starvation

- **Starvation** occurs when a process is perpetually denied the resources it needs to execute, often because other higher-priority processes continuously take precedence. To mitigate starvation, **priority aging** can be used, where the priority of a waiting process is gradually increased over time to ensure it eventually gets the necessary resources. This technique ensures that lower-priority processes are not starved indefinitely, especially in systems with many competing processes.

1.6 Conclusion

Concurrency is an essential aspect of embedded systems, particularly for reactive, real-time applications. This lecture emphasized the importance of understanding the mechanisms—such as **mutual exclusion**, **semaphores**, and **scheduling**—that help manage concurrent processes effectively. For embedded systems like automotive controls, pacemakers, or industrial automation, **deterministic behavior** and avoiding race conditions are paramount. Properly handling concurrency ensures that embedded systems can meet their timing constraints, maintain data consistency, and operate safely under all conditions.

Next steps in the lecture will involve exploring **reactive systems** and how **synchronous concurrency** can be implemented to further ensure the reliability of embedded systems. Understanding these concepts is vital for designing embedded systems that are safe, reliable, and efficient. Future topics will also cover specific synchronization protocols and **real-time operating system (RTOS)** features that support concurrency in embedded environments. Additionally, understanding how to leverage hardware features, such as **interrupt controllers** and **timers**, will be crucial for achieving precise control over concurrent tasks in embedded systems.

Lecture 2: SCCharts and Synchronous Concurrency

2.1 Introduction to Synchronous Concurrency and SCCharts

Synchronous Concurrency (SC) is a method used to manage the complexity of Cyber-Physical Systems (CPS) by providing deterministic behavior and efficient handling of concurrent processes. SCCharts are a powerful visual formalism used to model concurrent, reactive, and real-time behaviors in embedded systems. They offer a way to design systems that react to changes in the environment while maintaining reliable and predictable execution.

2.2 The Synchronous Approach

The **synchronous approach** is a paradigm in which all concurrent processes are assumed to execute in lockstep relative to a global logical clock. This provides a deterministic and predictable execution model, which is highly beneficial for embedded systems that operate in safety-critical environments.

Key aspects of the synchronous approach include:

- **Synchrony Hypothesis:** This hypothesis assumes that the system's reactions occur infinitely faster than the environment, leading to a **zero-delay model**. This means that the reaction to an input is instantaneous from the system's perspective, ensuring a deterministic response.
- **Synchronous Threading and Broadcast Communication:** All concurrent threads execute synchronously, meaning that outputs generated by one component are immediately visible to all other components. This approach resembles **synchronous circuits**, where each clock tick coordinates all activities, providing atomic and instantaneous reactions.
- **Atomicity and Instantaneity of Reactions:** The synchronous approach treats reactions as atomic, meaning that no intermediate states are visible

during execution. This helps prevent issues like race conditions, which are prevalent in asynchronous systems.

- **Reaction to Absence:** The system can react not only to events that occur but also to the absence of expected events, allowing for a more comprehensive model of environmental interaction.

2.3 SCCharts Overview

SCCharts are an extension of Statecharts, introduced by David Harel, that add the benefits of the synchronous paradigm to the powerful modeling features of Statecharts. SCCharts are used to describe complex systems involving multiple states and concurrent processes.

2.3.1 Similarities and Differences Between SCCharts and Statecharts

- **Similarities:** Both SCCharts and Harel's Statecharts share common elements such as states, transitions, signals/events, and modularity. They both support **hierarchy**, **parallelism**, and **broadcast communication** between states, which allows for more compact and expressive models.
- **Differences:** SCCharts operate within a **synchronous framework** and guarantee deterministic behavior, which contrasts with the often non-deterministic nature of Statecharts. SCCharts do not interpret events for simulations, which eliminates hidden behaviors and allows for more precise analysis. SCCharts also do not support **inter-level transitions** and provide deterministic handling of concurrent events, which simplifies debugging and validation.

2.4 Boolean Mealy Machine (BMM)

An SCChart can be represented using a **Boolean Mealy Machine (BMM)**. A BMM is defined as a tuple $M = \langle Q, q_0, I, O, T \rangle$, where:

- Q is the set of states.
- q_0 is the initial state.

- I is the set of inputs.
- O is the set of outputs.
- T is the transition relation, involving states and boolean conditions on inputs.

BMMs provide a formal basis for understanding how SCCharts operate, with transitions being defined based on the current state and inputs. The key properties of BMMs are **determinism** and **reactivity**:

- **Determinism**: At most one transition is enabled for any valid combination of inputs from the environment. This ensures that the system behaves predictably regardless of concurrent events.
- **Reactivity**: At least one transition is enabled for any valid combination of inputs, ensuring the system can always respond appropriately to environmental changes.

2.5 The VABRO Example

The **Valued ABRO (VABRO)** example, which is an adaptation of Berry's **ABRO**, is used to demonstrate the synchronous programming model. In this example, the system generates an output when two specific inputs have occurred. The model incorporates **strong pre-emption**, allowing the behavior to be reset when a particular condition is met. VABRO demonstrates how SCCharts can handle complex state-based logic, including counting events and generating output values based on conditions.

2.6 Constructiveness in Synchronous Systems

Constructiveness is an important concept in synchronous systems, ensuring that concurrent processes do not interfere with each other's operation. **Berry's Constructiveness** requires that all reads happen only after writes to shared entities are completed, and that there is at most one write to any shared entity during a reaction. This ensures consistency and prevents conflicting updates during a single tick of the logical clock.

Sequential Constructiveness extends Berry's constructiveness by allowing sequential updates to shared variables under certain conditions. The **Init-Update-Read (IRU) protocol** is used to manage variable updates in the following order:

1. **Initialization**: Concurrent threads can make confluent initializations to shared variables, provided they assign the same value.
2. **Update**: Threads update shared variables using the same update function, ensuring confluent updates.
3. **Read**: Finally, threads are allowed to read the updated value, guaranteeing consistency.

2.7 Timed SCCharts and Real-Time Systems

Timed SCCharts add temporal constraints to SCCharts, enabling the modeling of **real-time systems** where timing is crucial. These models can compute the **Worst Case Reaction Time (WCRT)** and adjust the tick length accordingly to ensure the system meets its timing requirements. By counting ticks, timed SCCharts allow real-time systems to be modeled effectively, ensuring that actions are taken within the required time frame.

Dynamic Ticks are also introduced to allow the system to adjust tick lengths dynamically, providing more flexibility in real-time operation. This flexibility is essential for systems that need to adapt to changing conditions, such as varying processing loads or external inputs that influence timing requirements.

2.8 Designing Real-Time Systems with SCCharts

The synchronous approach is particularly suitable for **real-time systems**, where deterministic behavior is essential for safety and reliability. By using SCCharts:

- **Concurrency is managed in a deterministic manner**, ensuring that all processes execute predictably without interference or unintended interactions.
- **Pre-emption and Suspension** are modeled explicitly, allowing designers to represent high-priority tasks interrupting lower-priority tasks, or suspending ongoing activities until a condition is met.

- The **generated code from SCCharts** compiles the concurrent specifications into sequential code, which is efficient and suitable for **bare metal implementations** without requiring an operating system. This makes SCCharts an excellent choice for systems with constrained resources, such as microcontrollers in embedded environments.

2.9 Conclusion

Synchronous concurrency, as implemented using SCCharts, provides a robust framework for designing **deterministic, reactive, and real-time embedded systems**. By combining the power of Statecharts with the guarantees of the synchronous model, SCCharts offer a visual and formal way to specify complex concurrent behavior while ensuring predictable execution. The **synchronous approach** is well-suited for **safety-critical CPS** applications, such as medical devices, automotive systems, and robotics, where the cost of failure is high.

Future discussions will delve deeper into **synchronous pre-emption**, **timed automata**, and how SCCharts can be extended to model more complex temporal behaviors. These tools and concepts are crucial for ensuring that CPS can meet their real-time requirements in a safe and reliable manner.

Part II

Avinash's Lectures

Lecture 3: Peripherals and Interfacing

3.1 Introduction to Peripherals and Interfacing in Embedded Systems

Embedded systems require effective mechanisms for communicating with various external devices, known as **peripherals**. These peripherals, which include sensors, actuators, timers, and communication modules, interact with the processor through specialized interfaces (Slide 4). **Interfacing** is critical in embedded systems as it determines how well the processor can manage peripherals to meet performance and reliability requirements. The lecture discussed various types of I/O, including memory-mapped and direct I/O, as well as data transfer methods like polling, interrupts, and Direct Memory Access (DMA).

3.2 Types of I/O Interfaces

- **Memory-Mapped I/O**: In this type of I/O, the processor treats peripheral registers as memory locations within the same address space (Slide 7). The advantage of this approach is that the same instructions can be used for both memory and I/O operations, simplifying program design. However, memory-mapped I/O also shares address lines with memory, which means it can impact addressable memory space.
  - Example: The **Nios II processor** from Altera uses memory-mapped I/O to communicate with peripherals via Avalon buses, where specific addresses map to different peripheral registers, such as control, status, or data registers (Slide 7). Functions like `IORD()` and `IOWR()` are used to read from and write to these registers.
- **Direct I/O**: Here, peripherals are addressed using separate I/O ports, distinct from the memory address space (Slide 8). This method requires specialized I/O instructions, such as `IN` and `OUT` on x86 processors, making it more suitable for systems where efficient separation of I/O and memory space is required. In **direct I/O**, the processor has separate control signals specifically for enabling devices, which prevents interference with memory operations.

3.3 Data Transfer Methods

- **Programmed I/O**: In this mode, the processor executes a program to directly manage data transfer between itself and the I/O device (Slide 11). While simple to implement, it is inefficient since the processor is blocked during the entire transfer process, often resulting in idle CPU cycles while waiting for the peripheral to be ready.
- **Interrupt-Driven I/O**: This method leverages **interrupts** to initiate data transfer, which improves efficiency by allowing the processor to perform other tasks until the I/O device signals readiness (Slide 14). When data is ready, the peripheral asserts an interrupt signal, causing the processor to execute an **Interrupt Service Routine (ISR)**. The ISR then handles the data transfer, enabling better utilization of CPU resources compared to polling.
  - **Fixed and Vectored Interrupts**: With **fixed interrupts**, the address of the ISR is built into the microprocessor, while **vectored interrupts** allow the peripheral to specify the address of its ISR, providing more flexibility (Slide 16-24). Interrupts improve system efficiency but add complexity due to the need for context saving and careful ISR design.
- **DMA (Direct Memory Access)**: DMA involves an external controller taking over the system bus to manage data transfer between a peripheral and memory without processor involvement, freeing up the CPU for other operations (Slide 37). **Block DMA** transfers data in large blocks, while **cycle-stealing DMA** transfers a few bytes at a time, relinquishing the bus to the CPU when needed (Slide 52).
  - **Example**: In a **peripheral-to-memory transfer using DMA**, the DMA controller initiates a request to gain control of the bus, then proceeds to transfer data while the CPU resumes its normal execution. The CPU only stalls when it requires access to the bus, significantly improving system performance by overlapping computation and data transfer (Slide 46-50).

3.4 Timer and Counter Peripherals

Timers and counters are integral parts of embedded systems for handling periodic events and measuring intervals (Slide 53).

- **Timers**: Timers are used to measure time intervals or generate timed output events. For example, a **16-bit timer** counting clock pulses with a 10 ns period can generate an event after a set duration. Timers can be used in applications like controlling traffic lights or measuring a car's speed.
- **Counters**: Similar to timers but used for counting external pulses, such as counting cars passing over a sensor. **Programmable timers** allow setting a terminal count value to specify when an event should occur, which is critical for tasks like interval timing or watchdog functionalities (Slide 54).
- **Example of Reaction Timer**: A **reaction timer** measures the user's response time to a visual stimulus (Slide 56). A timer is started when an indicator light turns on, and the user's reaction time is recorded by counting the number of timer overflows until a button is pressed. This showcases the use of timers to measure user interaction intervals.

3.5 UART Communication

Universal Asynchronous Receiver/Transmitter (UART) is a widely used peripheral for serial communication in embedded systems. It converts parallel data from the processor to serial data for transmission and vice versa (Slide 58).

- **Data Transmission and Reception**: UART communication uses start, data, and stop bits to frame each byte of information, with an optional parity bit for error detection. UARTs require synchronization mechanisms like **baud rate** settings to ensure the transmitter and receiver operate at the same speed (Slide 58).
- **Registers and Flow Control**: UARTs often use multiple registers, such as **control**, **status**, **rxdata**, and **txdata** (Slide 60). For example, a transmitter may set the **Request to Send (RTS)** bit in the control register and then poll the **Clear to Send (CTS)** bit to determine if the receiver is ready. Double buffering in the transmitter and receiver allows for more efficient communication by enabling one data transfer while another is in progress.
- **Interrupt Handling in UARTs**: The UART peripheral can generate interrupts when new data is received or the transmission buffer is ready for new data, thus enabling efficient communication without busy-waiting (Slide 64). The ISR reads the **rxdata** register and processes the incoming data, which

is crucial in scenarios where timely responses are needed, such as in real-time data logging.

3.6 Peripheral Interfacing Techniques

Interrupt Handling in Nios II Processors: The **Nios II processor** uses an exception handler provided by the Hardware Abstraction Layer (HAL) to differentiate between software exceptions and hardware interrupts (Slide 31). This approach ensures efficient response handling based on the nature of the interrupt. A typical function for registering an interrupt in Nios II is `alt_irq_register()`, where the handler function and device ID are specified (Slide 32).

Direct Memory Access for High-Speed Transfers: DMA is employed when dealing with high-speed peripherals that need to transfer large data blocks. For example, transferring data from a tape drive or video feed requires a DMA controller to handle the high volume of data without overloading the CPU (Slide 37). The use of **DMA controllers** helps avoid bottlenecks in data transfer and allows parallel processing, which is essential in high-performance embedded systems.

3.7 Practical Considerations and Limitations

Limitations of Programmed I/O: Programmed I/O is unsuitable for high-speed peripherals because it involves significant overhead, where the CPU needs to execute multiple instructions for each word of data transferred (Slide 36). This leads to inefficiencies, particularly when handling peripherals that require large amounts of data to be processed rapidly, such as audio or video streams.

DMA vs Programmed I/O: The main advantage of **DMA** over programmed I/O is its ability to handle large data transfers without constant CPU intervention, reducing the overhead on the main processor (Slide 38). This is particularly useful for applications like transferring video frames to memory or storing data from high-speed sensors.

3.8 Conclusion

Peripherals and interfacing are fundamental components of embedded systems, enabling interaction with the external world. The lecture covered various data transfer methods, highlighting their advantages and trade-offs. Effective use of **interrupts**, **DMA**, and **timers** allows embedded systems to operate efficiently, especially in real-time applications where timely responses are critical. Understanding the different modes of I/O—such as memory-mapped versus direct I/O—and the specific requirements for handling data transfers through **polling**, **interrupts**, or **DMA** helps in designing robust and responsive embedded systems.

Future discussions will focus on specific interfacing techniques for more complex peripherals, such as **SPI** and **I2C** communication protocols, as well as how to optimize **interrupt-driven I/O** to minimize latency and maximize throughput in resource-constrained environments.

Lecture 4: Embedded Processors

4.1 Introduction to Embedded Processors

Embedded processors are specialized computation engines designed to implement a system's desired functionality. Unlike general-purpose processors, embedded processors can be optimized for specific applications, balancing performance, power consumption, and area constraints (Slide 5). There are three main types of embedded processors: **general-purpose processors**, **single-purpose processors**, and **application-specific processors**. Each has its advantages and trade-offs, depending on the intended application.

4.2 Types of Embedded Processors

4.2.1 General-Purpose Processors

General-purpose processors (also known as microprocessors) are programmable devices used in a wide variety of applications. They contain a **program memory**, a **datapath** with a large register file, and a **general ALU** (Slide 7). Some notable features include:

- **High Flexibility**: General-purpose processors can run different software programs, making them highly adaptable.
- **Low Non-Recurring Engineering (NRE) Costs**: These processors can be developed for a broad market, leading to lower initial costs.
- **Examples**: The **Pentium** processor is a well-known example, but there are many other general-purpose processors used in embedded systems.

General-purpose processors are ideal for systems that need versatility but do not have extreme performance or power constraints.

4.2.2 Single-Purpose Processors

Single-purpose processors are digital circuits designed to execute only one specific program, such as a **coprocessor**, **accelerator**, or **peripheral** (Slide 8). Key characteristics include:

- **Minimal Components**: They contain only the elements necessary for their task, often without program memory, which reduces size and power consumption.
- **Benefits**: Single-purpose processors are **fast**, **small**, and **low power** due to their dedicated nature. Since there are no extra components, they achieve high efficiency in both speed and energy.
- **Examples**: Hardware implementations of functions like **Greatest Common Divisor (GCD)** are often realized with single-purpose processors.

These processors are well-suited for real-time, deterministic tasks that require minimal latency and maximum energy efficiency.

4.2.3 Application-Specific Processors

Application-specific processors are programmable processors optimized for a particular class of applications that share common characteristics (Slide 9). They combine aspects of both general-purpose and single-purpose processors:

- **Program Memory and Optimized Datapath**: Application-specific processors include a program memory and a **customized datapath** tailored to the target applications, such as **DSPs** (Digital Signal Processors).
- **Trade-Off**: They offer some flexibility compared to single-purpose processors, while providing better performance and efficiency than general-purpose processors.
- **Benefits**: These processors provide a balance of flexibility, performance, size, and power consumption, making them ideal for use in systems that need customization without sacrificing too much generalizability.

4.3 Finite-State Machine with Datapath (FSMD)

One of the most common approaches in custom embedded processor design is the use of a **Finite-State Machine with Datapath (FSMD)**. FSMD is a model that integrates control and data paths, allowing complex behaviors to be implemented systematically (Slide 17).

- **FSMD Structure**: An FSMD consists of:
  - **Controller**: A finite-state machine that determines the sequence of operations.
  - **Datapath**: Composed of **registers**, **ALUs**, and **other functional units** used to manipulate data (Slide 19).
- **Design Flow**: Typically, a design starts by creating an algorithm, converting it to a state machine, and then mapping it to an FSMD (Slide 18). This involves defining states, transitions, and operations that control how data is processed.
- **Example: GCD Calculation**: The **Greatest Common Divisor (GCD)** example demonstrated in the lecture involves defining a simple algorithm and converting it to an FSMD. The states of the GCD FSM correspond to different stages of the computation, and the datapath manages the subtraction operations required to compute the GCD (Slide 17).

4.4 Custom Processor Design Techniques

The design of custom embedded processors involves several key techniques for optimizing performance, size, and power:

- **Datapath Design**: Involves creating registers for each declared variable and functional units for each arithmetic operation (Slide 19). The registers, ALU, and functional units are connected based on the data flow in the original algorithm.
- **Controller Design**: The controller is responsible for sequencing the operations in the datapath. Complex actions in the FSMD are broken down into manageable control steps to ensure deterministic behavior (Slide 20).

- **Splitting into Controller and Datapath**: Splitting the controller and datapath allows for better modularity and optimization of the design. The controller handles decision-making, while the datapath focuses on arithmetic and logical operations (Slide 21).

4.5 Optimization Strategies for Embedded Processors

Optimization is the process of refining the design to achieve the best possible trade-off among different design metrics, such as speed, area, and power consumption (Slide 24).

- **Program Optimization**: Involves analyzing the original program for inefficiencies. The use of more efficient arithmetic operations or changing the sequence of operations can significantly improve the overall performance. For example, replacing subtraction operations with modulo in the GCD algorithm sped up the calculation and reduced loop iterations (Slide 26).
- **FSMD Optimization**: The FSMD can be optimized by merging or separating states to improve efficiency (Slide 27). **State merging** can be done for states with constant transitions, whereas **state separation** can help reduce hardware complexity for states requiring complex operations.
- **Datapath Optimization**: **Sharing functional units** between states can reduce the area by avoiding redundant hardware. **Multi-functional units** like ALUs can be used for different operations in different states, optimizing resource usage (Slide 29).
- **FSM Optimization**: Optimizing the finite-state machine involves techniques like **state encoding** (assigning efficient binary codes to states) and **state minimization** (merging equivalent states to reduce complexity) (Slide 30).

4.6 Custom Single-Purpose Processor Design: An Example

The lecture provided an example of designing a custom single-purpose processor for a simple **bridge** that converts two 4-bit inputs into an 8-bit output (Slide 31).

- **RT-Level Design**: The Register-Transfer (RT) level model is used to specify how data is transferred and manipulated in each clock cycle. The design includes both a **controller FSM** and a **datapath** with registers and functional units (Slide 32).
- **State Diagram and Data Flow**: The RT-level processor's behavior is defined through state diagrams and data flow diagrams, demonstrating how each input is processed and combined into an output. The controller is responsible for sequencing these operations, while the datapath handles the actual data manipulation.

4.7 C to Hardware: Synthesis and Implementation

Another important aspect covered in the lecture was the conversion from **C code to hardware**. This is known as **High-Level Synthesis (HLS)** and is used to automatically convert behavioral C code into corresponding hardware designs (Slide 14).

- **Motivation**: The main advantage is achieving maximum performance for specific applications without having to manually write hardware descriptions in HDL (Hardware Description Language). This approach allows software developers to leverage existing C code and translate it into efficient hardware implementations.
- **Components**: The translation involves identifying data and control instructions and converting them into datapath elements like **ALUs**, **registers**, and **control logic**. This is then used to create a hardware implementation of the algorithm specified in C (Slide 15).

4.8 Evolution of Embedded Processors

Embedded processors have evolved significantly to meet the demands of modern applications (Slide 10).

- **Single-Purpose to Application-Specific Processors**: Initially, systems relied on **single-purpose processors** for dedicated tasks. These processors provided efficient solutions but lacked flexibility.

- **Application-Specific Processors**: As the complexity of embedded applications increased, the need for processors that strike a balance between performance and flexibility led to the development of **application-specific processors**. These processors retain the programmability of general-purpose processors while including optimizations for particular tasks.
- **Customizable Processors**: Modern approaches include the use of **customizable processors** that allow designers to optimize hardware for a specific application domain without losing the general flexibility of programmable processors.

4.9 Conclusion

The lecture provided a comprehensive overview of **embedded processors** and their role in implementing complex systems. Custom embedded processors, including general-purpose, single-purpose, and application-specific types, each have their unique trade-offs. Techniques such as **FSMD modeling**, **controller and datapath splitting**, and **high-level synthesis** were introduced to demonstrate the design and optimization of these processors.

Future discussions will delve deeper into **real-time constraints**, **hardware-software co-design**, and techniques for optimizing **multi-core embedded systems** to meet stringent performance and power requirements. Understanding these concepts is essential for designing embedded processors that can efficiently address the challenges posed by modern applications.

Lecture 5: SoC and SOPC Buses

5.1 Introduction to SoC and SOPC

System-on-Chip (SoC) and **System-on-Programmable-Chip (SOPC)** are key technologies in modern embedded systems. These technologies combine multiple functional blocks, such as processors, memory, and peripherals, onto a single chip. They use **buses** to facilitate communication between these components (Slide 1). Buses are fundamental to the functionality of SoC and SOPC designs, as they determine how different modules interact, ensuring reliable data transfer and system performance.

5.2 Internal vs External Buses

Buses can be broadly classified into **internal** and **external** types based on their scope within a system (Slide 9):

- **Internal Buses**: These are used within a single chip or a tightly integrated board to connect different functional blocks. Examples include:
  - **PCI** (Peripheral Component Interconnect)
  - **AGP** (Accelerated Graphics Port)
  - **PCMCIA**
  - **AMBA** (Advanced Microcontroller Bus Architecture) for ARM-based systems
  - **Avalon** buses used in Altera's SOPC designs
- **External Buses**: These are used to connect different chips or boards. Examples include:
  - **USB** (Universal Serial Bus)
  - **FireWire**

5.3 Bus Design Issues

The design of a bus impacts system performance, scalability, and cost. Several factors must be considered when designing a bus (Slide 14):

- **Bus Width**: The width of the data and address buses is a critical determinant of the system's performance. A wider bus can transfer more data per cycle but is more expensive to implement. For example, a Pentium processor has a 32-bit instruction set architecture (ISA) but a 64-bit data bus, while an Itanium processor has a 128-bit data bus, offering significantly more bandwidth for data transfers (Slides 15-16).
- **Bus Type**: Buses can be **dedicated** or **multiplexed**:
  - **Dedicated Buses** have separate lines for data and address information, providing better performance but increasing costs.
  - **Multiplexed Buses** share the same lines for data and address, which reduces cost and resource usage but may introduce delays in data transmission (Slide 17).

5.4 Bus Operations

Bus operations define how data is transferred between components (Slide 18). Common bus operations include:

- **Read and Write**: Basic data transfer operations.
- **Block Transfer**: A series of contiguous memory locations are read or written in sequence, useful for operations like filling a cache line.
- **Read-Modify-Write**: Often used for critical sections where multiple operations must be performed atomically.
- **Interrupt Operation**: A mechanism for peripherals to request the processor's attention, triggering an interrupt service routine.

5.5 Synchronous vs Asynchronous Buses

5.5.1 Synchronous Bus

A **synchronous bus** uses a **bus clock signal** to synchronize all actions on the bus (Slide 19). Characteristics of synchronous buses include:

- **Clock Edge Coordination**: All data transfers occur relative to a clock signal, simplifying timing design.
- **Wait States**: A synchronous bus may operate with **wait states** if the target device is not ready for a data transfer. Without wait states, data transfers occur at the maximum clock speed, but adding wait states ensures reliable operation if the target is slower (Slides 20-24).
- **Block Transfers**: Synchronous buses also support **block transfers**, which allow multiple data units to be transferred in a single bus transaction, improving throughput (Slide 24).

5.5.2 Asynchronous Bus

An **asynchronous bus** does not use a clock signal to synchronize actions. Instead, it relies on **handshaking signals** to control data transfer between the master and slave (Slide 25). Features include:

- **Handshaking Protocol**: Two synchronization signals, **Master Synchronization (MSYN)** and **Slave Synchronization (SSYN)**, ensure that the sender and receiver are ready for each data transfer.
- **No Clock Dependency**: This allows asynchronous buses to be more flexible, as they do not require a fixed clock frequency. However, implementing the handshaking mechanism can be more complex compared to synchronous buses.

5.6 Bus Arbitration

In systems with multiple masters (e.g., CPUs, DMA controllers), **bus arbitration** is required to manage access to the bus (Slide 27). Arbitration mechanisms include:

- **Static Arbitration**: Allocates the bus in a fixed, predetermined manner. While easy to implement, this approach can lead to poor utilization and potential bus monopolization by one master (Slide 28).
- **Dynamic Bus Arbitration**: Allocates the bus only when a master requests it, allowing more efficient use of resources. Masters use **bus request** and **bus grant** lines to signal their need for the bus and receive permission to use it. Dynamic arbitration supports various allocation policies (Slides 28-40):
  - **Fixed Priority**: Assigns a fixed priority to each master, but may lead to **bus hogging** by high-priority devices.
  - **Rotating Priority**: Changes the priority of masters dynamically to prevent starvation. The **lowest priority** is assigned to the master that just used the bus, ensuring fair access.
  - **Fair Policies**: Ensure that no master is starved of the bus. These policies can be based on **time windows** to guarantee a maximum delay before a bus request is granted.
- **Bus Release Policies**: Determine when the current master releases the bus. Two types of policies are:
  - **Non-Preemptive Release**: The master voluntarily releases the bus after completing the current transaction. This is easy to implement but can lead to inefficient bus usage if the master holds the bus unnecessarily (Slide 40).
  - **Preemptive Release**: Forces the current master to release the bus, which can be useful for handling urgent requests from other masters.

5.6.1 Bus Arbitration Implementation

Centralized Bus Arbitration: Uses a **daisy-chaining** method where the bus grant signal is passed down a chain of masters. It is easy to implement but has several drawbacks, such as fixed priority and increased arbitration time proportional to the number of masters (Slide 37).

Independent Requests: Each master has a dedicated bus request line to the central arbiter. This approach is more fault-tolerant and allows for **constant arbitration time**, but requires a higher number of control signals (Slide 39).

Hybrid Arbitration Schemes: Combine features of daisy-chaining and independent requests. For example, bus masters can be grouped into **classes**, with independent arbitration at the class level and daisy-chaining within each class. This provides a balance between cost and performance (Slide 40).

5.7 System Bus Components

The system bus consists of the following components:

- **Address Bus**: Carries the addresses from the master to the slave, specifying the location for read or write operations.
- **Data Bus**: Transfers actual data between components.
- **Control Bus**: Carries control signals, such as read/write enable, to coordinate the operations on the address and data buses (Slide 12).

Buses can be either **dedicated** or **multiplexed**, **synchronous** or **asynchronous**, depending on the system's requirements for speed, cost, and complexity (Slide 12).

5.8 Learning Outcomes

By understanding the different types of buses, operations, arbitration schemes, and bus components, one can design effective **SoC and SOPC** systems that balance performance, cost, and complexity. Key takeaways include:

- The importance of selecting appropriate **bus width** to match system requirements while considering cost implications.
- Differences between **synchronous** and **asynchronous** buses and the trade-offs involved in choosing one over the other.
- The necessity of **bus arbitration** in multi-master systems to ensure fair access and system efficiency.
- How different arbitration policies impact bus access, latency, and fairness.

The **building blocks of an SoC or SOPC**—including **masters (e.g., processors, DMA controllers)**, **slaves (e.g., peripherals)**, **buses**, **decoders**, and **arbiters**—all work together to create a cohesive, functional system capable of handling complex tasks (Slide 53).

5.9 Conclusion

The lecture provided a comprehensive overview of **bus design and operations** in SoC and SOPC systems. Understanding bus types, operations, arbitration, and implementation is crucial for designing efficient and scalable embedded systems. These principles are applied to ensure that various components within a SoC or SOPC can communicate effectively while meeting performance requirements.

Future discussions will explore **on-chip interconnects**, **crossbars**, and **network-on-chip (NoC)** architectures, which are becoming increasingly relevant for **multi-core SoC designs** that demand higher data throughput and efficient parallel processing.

Part III

Nathan's Lectures

Lecture 6: Embedded Control Systems 1

6.1 Introduction to Embedded Control Systems

Embedded control systems are crucial for industrial automation and other real-time applications that require precise, timely responses. These systems often control physical processes, ensuring safety and reliability through careful computational analysis of execution times. This lecture focused on various elements of embedded control, such as real-time system properties, sampling and quantization, computational delays, and worst-case execution time (WCET) analysis (Slides 2-3).

6.2 Real-Time Systems

Definition: Real-time systems are used in scenarios where timing is critical, such as industrial automation and safety-critical applications. These systems must perform their functions not only correctly (functional correctness) but also within specific timing constraints (Slide 3).

Examples: A robotic arm must stop within 10 ms of detecting human intrusion to prevent accidents. This combination of functional and timing correctness is what defines a real-time system.

6.3 Sampling and Quantization

Sampling: Sampling refers to taking discrete snapshots of an analog signal over time. This process is fundamental in digitizing signals like video, audio, or physical voltages. The higher the sampling rate, the more accurately the original signal is represented (Slides 4, 8).

Quantization: Due to the discrete nature of computer data, real values must be approximated to the nearest value in a finite set. Quantization replaces real-world continuous values with discrete levels, introducing **quantization error** (Slide 5).

- **Quantization Levels**: For an n-bit Analog-to-Digital Converter (ADC), there are 2^n levels, which determine how finely the signal is represented. For example, an 8-bit ADC has 256 quantization levels.
- **Resolution**: The resolution of an ADC determines the voltage represented by each digital increment. For instance, a 10-bit ADC with a High Reference Voltage (HRV) of 5V and Low Reference Voltage (LRV) of 0V has a resolution of approximately 0.00488V per step (Slide 6).

6.4 Aliasing

Aliasing occurs due to insufficient sampling rates. If the sampling rate is less than twice the frequency of the signal (the **Nyquist frequency**), higher-frequency signals appear indistinguishable from lower frequencies, leading to errors (Slides 9-11).

- For example, if a signal is sampled at 2.5 Hz, signals at 0.5 Hz, 3 Hz, and 5.5 Hz appear the same, leading to ambiguity.

6.5 Computation Delays and Timing Variances

Computation Delays: In real-time systems, delays can occur due to inherent processing overhead, causing sampling or actuation to happen later than expected. Accurate delay characterization is needed to ensure that systems meet timing requirements (Slide 12).

- **Sources of Timing Variance**:
  - **Unbounded Software Constructs**: Unpredictable branches, loops, recursion, and dynamic memory allocations can introduce delays (Slide 13).
  - **Speculative Hardware**: Features like pipelines, memory hierarchies (caches), and branch predictors in modern processors can lead to non-deterministic execution times.

6.6 Worst-Case Execution Time (WCET)

WCET is the maximum time required for a system to complete a given task (Slide 14). Understanding WCET is critical for ensuring that an embedded control system meets its real-time constraints.

- **Methods to Determine WCET**:
  - **Measured**: Simple to implement but may not explore all possible execution paths, thus providing non-guaranteed values.
  - **Static Analysis**: Uses an abstraction of the system to analyze all possible paths, ensuring coverage but requiring significant computational resources.

6.7 Static Timing Analysis

- **Control Flow Graph (CFG)**: Timing analysis starts with creating a **control flow graph** (CFG) to map program flow, representing different segments of code (Slides 18-19).
  - **Basic Blocks and Transitions**: CFG nodes are called **basic blocks**, which represent straight-line code without branches, and transitions show how the program moves between these blocks.
  - **Path Analysis**: Finding the **longest execution path** within the CFG is key to determining the worst-case scenario for execution time (Slide 20).
- **Timed CFG**: A timed CFG uses clock cycle estimates for different operations—computation, load, store, branch, and return—to determine execution duration (Slide 23).
  - Example values: **Load** and **Store** take 5 cycles, **Computation** takes 1 cycle, **Branch** (if taken) takes 3 cycles.

6.8 WCET Analysis Using Max-Plus Algebra

- The **Max-Plus** approach simplifies WCET analysis by using only max and plus operations to calculate the worst-case path cost (Slide 24).
  - For example, the WCET for two possible paths might be calculated as: $\text{wcet} = \max(19 + 14 + 10, 21 + 13 + 10) = 44$ clock cycles.

6.9 Integer Linear Programming (ILP)

- **ILP** is used for WCET analysis by defining an **objective function** to maximize execution time under given constraints (Slide 28-30).
  - The constraints are typically linear inequalities describing valid program transitions and flow consistency, allowing ILP to provide an upper bound for WCET.

6.10 Coding Guidelines for Time Predictability

- **Bounded Loops**: Always use bounded loops to guarantee an upper bound on execution time (Slide 36). For example, in the **GCD** function, the loop is bounded to ensure a predictable runtime.
- **Static Memory Allocation**: Prefer static allocation over **dynamic memory allocation** (e.g., `malloc`) to prevent runtime unpredictability caused by memory fragmentation or variable allocation times (Slide 38).
- **Avoiding Interrupts**: Where possible, avoid **interrupts** due to their impact on execution time variability. Instead, use polling or reactive processors that can queue interrupts predictably (Slide 39).
- **Fixed Point Arithmetic**: Use **fixed-point arithmetic** over floating-point for better predictability, as floating-point operations are not always supported natively and can have variable runtime characteristics.

6.11 PRET Philosophy

- The **PRET (Precision Timed)** philosophy emphasizes treating time as a first-class citizen, aiming for a **time-predictable system** (Slide 35). The objective is to achieve predictability in hardware, software-hardware interfaces, and the software itself, ensuring reliable operation within a time-constrained environment.

6.12 Conclusion

This lecture provided insights into the design and analysis of **embedded control systems**. Key topics included **real-time system requirements**, **sampling and quantization**, **computation delays**, **WCET analysis**, and **coding practices for time predictability**. Accurate timing analysis and predictable software execution are crucial for the development of safe and reliable embedded control systems. The concepts discussed lay the foundation for designing robust control systems that meet strict real-time requirements.

Lecture 7: Embedded Control Systems 2

7.1 Introduction to Embedded Control Systems

In this lecture, we focused on advanced topics in **embedded control systems**, including **PWM control**, **stepper motors**, **PID control**, and **ADC technologies**. These components play a critical role in achieving accurate and efficient control in real-time embedded applications (Slide 2).

7.2 Pulse-Width Modulation (PWM)

- **Pulse-Width Modulation (PWM)** is a technique used to simulate an analog signal using digital means by varying the duty cycle of a digital pulse (Slide 5).
  - **Duty Cycle**: The percentage of time the signal remains high during a cycle determines the average voltage output. A higher duty cycle implies higher average power delivered, which is crucial in controlling actuators like DC motors and LEDs (Slide 7).
  - **Fast PWM vs Phase-Correct PWM**: There are different types of PWM modes. **Fast PWM** operates at a higher frequency to minimize flickering, while **Phase-Correct PWM** maintains symmetry in the signal waveform, which is often better for controlling motors with less noise (Slide 8).

7.3 DC and Stepper Motors

7.3.1 DC Motors

- **DC Motors** are widely used for their simplicity and ease of control. Speed control is achieved by modulating the voltage applied to the motor, which is often implemented through PWM. The relationship between PWM duty

cycle and motor speed is usually non-linear, requiring careful tuning to achieve precise control (Slide 10).

- **Motor Driver Circuitry**: To drive DC motors, **H-bridges** are commonly used. An H-bridge allows for directional control by reversing the current flow through the motor, enabling forward and reverse motion (Slide 11).

7.3.2 Stepper Motors

- **Stepper Motors** are used when precise control of angular position is required. Each pulse sent to the motor corresponds to a fixed angular movement, allowing fine control over its rotation (Slide 12).
  - **Types of Stepper Motors**: Two main types include **unipolar** and **bipolar** stepper motors, each with distinct wiring configurations and driving requirements (Slide 13).
  - **Stepper Motor Control**: To control the stepper motor, sequences of pulses must be sent in the correct order to energize the windings and produce the desired stepping motion. This is typically achieved using a **stepper motor driver**, which abstracts the complexity of pulse timing (Slide 14).

7.4 PID Control

Proportional-Integral-Derivative (PID) Control is a control loop mechanism widely used in industrial control systems to achieve desired system behavior.

- **Proportional Control (P)**: The **proportional** term produces an output value proportional to the current error value. If the error is large, the control response is also large. This helps in reducing the response time but may lead to instability if used alone (Slide 17).
- **Integral Control (I)**: The **integral** term sums the error over time, addressing the **steady-state error** by accumulating corrections until the desired setpoint is achieved. This component helps eliminate any residual error that remains after the proportional action has been applied (Slide 18).

- **Derivative Control (D)**: The **derivative** term considers the rate of change of the error, providing a damping effect that reduces overshoot. It predicts future error and helps the system respond more smoothly, avoiding large fluctuations (Slide 19).
- **Tuning PID Parameters**: Tuning the PID parameters (K_p, K_i, and K_d) is crucial for optimal system performance. Methods like **Ziegler-Nichols** can be used to empirically determine the ideal values, balancing response speed, stability, and accuracy (Slide 20).

7.5 Analog-to-Digital Conversion (ADC)

- **Analog-to-Digital Converters (ADC)** are used to digitize analog signals so that they can be processed by digital controllers. ADCs are critical in embedded control systems where signals like temperature, pressure, or speed need to be sampled and processed (Slide 22).
  - **Quantization and Resolution**: The resolution of an ADC refers to the number of discrete levels used to represent the analog input. A higher resolution means better accuracy but increased cost and complexity. For instance, a 12-bit ADC provides 2^{12} quantization levels, which significantly improves signal representation over an 8-bit ADC (Slide 23).
  - **Sampling Rate**: The sampling rate must be at least twice the highest frequency of the input signal to prevent **aliasing** (Nyquist criterion). If the sampling rate is insufficient, aliasing can lead to distorted signals that compromise control accuracy (Slide 24).
- **Types of ADC Technologies**:
  - **Successive Approximation Register (SAR)**: A commonly used ADC type that offers a good balance between speed and accuracy. The SAR ADC works by iteratively approximating the input signal, which makes it suitable for medium-speed applications (Slide 25).
  - **Sigma-Delta ADC**: Provides very high resolution but at the cost of slower conversion speed, often used in applications where precision is more critical than speed, such as audio signal processing (Slide 26).

- **Flash ADC**: The fastest type of ADC, using a parallel set of comparators to convert the analog signal in a single step. However, it requires a large number of comparators, making it expensive and power-hungry, suitable for high-speed applications (Slide 27).

7.6 Practical Considerations in Embedded Control Systems

- **Noise Handling**: Analog signals are often susceptible to noise. Techniques such as **filtering** and **shielded cabling** are used to mitigate noise effects, ensuring that the ADC accurately captures the intended signal without distortion (Slide 29).
- **Anti-Aliasing Filters**: To prevent aliasing, **low-pass filters** are often used before the ADC to remove high-frequency components that could interfere with accurate sampling (Slide 30).

7.7 Case Study: Motor Speed Control

- **Objective**: The case study presented a practical example of motor speed control using a combination of PWM and PID control (Slide 32).
- **Implementation**: The motor speed is monitored using an **optical encoder**, which provides feedback to the PID controller. The PID controller adjusts the PWM duty cycle to minimize the speed error and maintain the desired speed despite varying load conditions (Slide 33).
- **Challenges**: Tuning the PID parameters was a challenge due to the non-linearities in motor response and the presence of external disturbances. Real-world control systems often require iterative tuning to balance response time, overshoot, and stability (Slide 34).

7.8 Conclusion

This lecture expanded on the foundational knowledge of embedded control systems by introducing practical components like **PWM**, **stepper motors**, **PID controllers**,

and **ADCs**. Understanding these components and their interactions is essential for designing robust and efficient control systems. Future lectures will focus on more advanced topics such as **real-time operating systems (RTOS)**, **sensor fusion**, and **model predictive control (MPC)**, which are critical for modern, complex embedded control applications.

Lecture 8: Embedded Control Systems 3

8.1 Introduction to Control Systems

In this lecture, we delve deeper into **control systems**, focusing on both **open-loop** and **closed-loop** configurations, as well as the terminology, benefits, and challenges associated with each approach (Slide 3). Control systems are used to ensure that the output of a physical system tracks a desired reference value by manipulating the inputs to the system. This lecture also highlighted the importance of stability, performance, and disturbance rejection in control systems.

> ❗ Important Note
>
> **THIS IS NOT THE SAME AS ELECTENG 332. THIS IS A DIFFERENT COURSE.**

8.2 Open-Loop vs Closed-Loop Control Systems

8.2.1 Open-Loop Control Systems

- **Definition**: In an **open-loop control system**, the controller adjusts the inputs to the plant without considering the actual output (Slide 14). It does not measure how well the output matches the desired reference.
- **Components**:
  - **Plant**: The physical system to be controlled (e.g., a car, a disk drive, or a heater) (Slide 12).
  - **Actuator**: A device that changes the parameters of the system, such as a throttle or motor.
  - **Reference**: The desired value of the output, such as speed or temperature.
- **Characteristics**:

  - **No Feedback**: Open-loop systems do not use feedback to correct their actions.
  - **Feed-Forward Control**: There may be delays in actual output changes.
  - **Predictable Output**: Open-loop systems are suitable when the plant output responds predictably to inputs and disturbances are minimal (Slide 14).

8.2.2 Closed-Loop Control Systems

- **Definition**: In a **closed-loop control system**, the controller continuously adjusts the inputs based on the difference between the measured output and the desired reference (Slides 15, 27).
- **Components**:
  - **Sensor**: Measures the plant output and provides feedback.
  - **Error Detector**: Calculates the error by comparing the output with the reference value.
  - **Feedback Loop**: This helps to adjust inputs to minimize the error.
- **Benefits**:
  - **Error Minimization**: By using feedback, the closed-loop system reduces tracking errors.
  - **Adaptation to Disturbances**: The system can react to changes, such as disturbances, to maintain desired performance.
  - **Stability**: Ensures that all control variables remain bounded, ideally making the error converge to zero over time (Slide 9).

8.2.3 Performance Metrics of Control Systems

- **Rise Time**: The time taken for the output to go from 10% to 90% of the desired value (Slide 10).
- **Peak Time**: The time taken to reach the first peak of the response.
- **Overshoot**: The percentage by which the output exceeds the final value.
- **Settling Time**: The time it takes for the output to reach and stay within 1% of the final value.
- These metrics help assess how well the output tracks changes in the reference input and how the system responds to disturbances.

8.3 Designing Control Systems

8.3.1 Open-Loop Control Design

- **Plant Modeling**: The plant model represents how the output changes based on inputs and the current state. Having a model helps in designing a better controller, although experimentation can be useful (Slide 18).
  - Example: In a car model, the speed is a function of the current speed and throttle position. The model can help predict how changes in the throttle angle affect the speed.
- **Controller Design**: A simple proportional controller can be used to manipulate the plant input. However, achieving the desired output requires tuning and careful consideration of system dynamics.

8.3.2 Closed-Loop Control Design

- **Proportional (P) Controller**: A proportional controller applies control by multiplying the tracking error by a gain value (Slide 28).
  - **Effects**: Adjusting the gain value (K) affects transient response, steady-state tracking, and disturbance rejection. However, increasing the gain can lead to oscillations (Slides 29-31).

- **Derivative (D) Controller**: Adds a control action based on the rate of change of the error, helping reduce overshoot and speed up convergence to the desired output (Slides 44-45).
- **Integral (I) Controller**: Addresses steady-state error by accumulating the error over time, ensuring that the output reaches the desired value eventually. However, it may introduce oscillations if not carefully tuned (Slides 49-50).
- **PID Controller**: A combination of Proportional, Integral, and Derivative components, the PID controller offers good steady-state performance, minimizes oscillations, and improves disturbance rejection (Slide 52).

8.4 Example Analysis of Controllers

- **Open-Loop vs Closed-Loop Performance**: Open-loop controllers struggle to handle disturbances, whereas closed-loop controllers can mitigate the effects of changes in the environment (Slide 42).
- **PD Controller**: By incorporating derivative action, the PD controller improves transient response, but does not change steady-state error (Slide 46).
- **PI Controller**: Useful for eliminating steady-state error by continually adjusting based on accumulated error (Slide 49).
- **PID Controller**: The combined PID control offers comprehensive benefits by balancing response time, minimizing steady-state error, and controlling oscillations.

8.5 Stability and Oscillation

- **Stability**: The stability of the control system is a critical objective, ensuring all control variables remain bounded (Slide 32).
  - The rate of decay of the effect of initial speed is controlled by the gain value (α). Setting a suitable gain is essential to avoid unbounded growth or excessive oscillations (Slides 33-34).
  - **Oscillation**: Negative values of the rate of decay parameter (α) can lead to oscillatory behavior, which must be avoided in most practical control applications (Slide 38).

8.6 Trade-Offs in Controller Design

- **Proportional Gain (K)**: Higher values can improve response time but lead to increased oscillations.
- **Disturbance Rejection**: To effectively reject disturbances, the proportional gain must be balanced to avoid compromising system stability (Slide 34).
- **Performance Objectives**: Reducing oscillation, achieving faster convergence, and minimizing steady-state error are often conflicting objectives, requiring careful tuning and consideration of the application requirements (Slide 40).

8.7 Conclusion

This lecture provided an overview of **open-loop** and **closed-loop control systems**, the key components involved, and the trade-offs between stability, response time, and error minimization. Understanding these concepts is crucial for designing robust and efficient control systems, whether using simple proportional control or more advanced **PID controllers**. Future discussions will explore more advanced control techniques, such as **state-space models** and **model predictive control (MPC)**, which are particularly useful for managing complex, multi-variable systems.

Lecture 9: Industrial Automation 1

9.1 Introduction to Industrial Automation

Industrial Automation (IA) refers to the use of computers, programmable logic controllers (PLCs), and other control systems to operate industrial machinery and processes without requiring human intervention (Slide 5). This step beyond mechanization aims to improve productivity, quality, flexibility, and safety while reducing operational costs and human errors.

9.2 Advantages and Drawbacks of Industrial Automation

9.2.1 Advantages

- **Productivity**: Automated systems can operate continuously, 24/7, significantly improving output compared to human-operated systems.
- **Quality and Waste Reduction**: Automation reduces the likelihood of human error, leading to better product quality and less waste. Sensors provide accurate, quantitative data, which contributes to enhanced consistency.
- **Flexibility**: Machines can be easily reprogrammed to handle new tasks, making automation well-suited for dynamic production environments where frequent changes are required.
- **Information Accuracy**: Automation employs sensors to gather quantitative data, reducing errors that arise from subjective human measurements.
- **Safety**: By automating dangerous tasks, robots and other automated equipment can work in environments that are unsafe for human workers, such as those involving hazardous chemicals or extreme temperatures.

9.2.2 Drawbacks

- **Initial Cost**: Setting up industrial automation systems involves high initial investment costs, including purchasing robots and making changes to existing

production lines.
- **Maintenance**: Ongoing maintenance is required to ensure that automated systems operate effectively. The need for specialized maintenance plans can increase overall costs.

9.3 Basic Components of Industrial Automation

- **Programmable Logic Controllers (PLCs)**: Hardened computers specifically designed for industrial use to control processes, machinery, and systems (Slide 8).
- **Numerical Controllers and Motion Controllers**: These devices are used to control specific tasks, such as the movement of mechanical parts.
- **Remote I/O and Communication Devices**: These components are essential for enabling communication between different parts of the automation system.
- **Industrial PCs**: Used for tasks that require more computational power or data processing capabilities than PLCs can provide.
- **Mechatronic Components**: Include drives, sensors, actuators, valves, etc., which interact with physical systems to achieve automation objectives.

9.4 PLCs and HMIs

- **PLCs**: PLCs are at the core of industrial automation, providing fast and reliable control. They are capable of synchronizing input from sensors and output to actuators, allowing for precise control of machinery (Slide 9).
  - **Characteristics of PLCs**: They are **fast** (providing response times in milliseconds), **durable** (built to withstand tough industrial conditions), **maintainable** (can be programmed using ladder logic), and **reliable** due to their simplicity and robustness (Slide 11).
- **Human-Machine Interface (HMI)**: HMIs are used to interact with PLCs, allowing operators to input and monitor system parameters, such as temperature or pressure settings (Slide 9).

9.5 Generations of Industrial Automation

9.5.1 Generation 1: Relay Ladder Circuits (Slide 15)

- **Hard-Wired Logic**: Early automation used hard-wired ladder logic circuits to control industrial equipment. These circuits were inflexible and required extensive rewiring to make changes.

9.5.2 Generation 2: Programmable Logic Controllers (PLCs) (Slide 16)

- **Introduction of PLCs**: PLCs revolutionized automation by providing programmable, hardened computers that replaced hard-wired circuits. They significantly improved the flexibility of automation systems, allowing for easier modification and expansion.

9.5.3 Generation 3: Multifunctional PLCs (Slide 17)

- **Modern PLCs**: Today's PLCs support multiple programming languages and feature a versatile architecture. They can be expanded and integrated into larger systems, providing more functionality beyond simple logic control.

9.5.4 Generation 4: Distributed Automation Solutions (Slide 20)

- **Distributed Systems**: Modern automation relies on distributed systems where a central processor interacts with sensors and actuators via networked field-area buses. This architecture offers enhanced scalability and modularity, enabling more complex automation solutions.

9.6 Challenges in Industrial Automation

9.6.1 Challenge 1: Flexible Modular Manufacturing (Slide 23)

- **Flexibility in Manufacturing**: The key challenge is to replace components of the manufacturing system without affecting the rest of the system. A modular approach enables seamless replacement and modification of conveyor sections or robotic workstations without the need for significant system reconfiguration.

9.6.2 Challenge 2: Dynamic Self-Configurability (Slide 27)

- **Self-Configurable Systems**: The dream of automation engineers is a system that can dynamically adapt and self-configure to handle different tasks without manual intervention. Achieving this level of automation requires sophisticated communication, middleware, and intelligent control systems.

9.7 Object-Oriented Design of Automation Software

- **Object-Oriented Design**: Designing software to match the physical structure of automation systems enhances modularity and flexibility. For example, structuring software components to represent individual physical units enables easy modification and adaptation (Slide 21).

9.8 Intelligent Agent-Based Automation

- **Agents in Automation**: The future of automation lies in using intelligent agents that can negotiate, recognize situations, make decisions, and coordinate operations autonomously (Slides 28-29).
  - **Agent Architecture**: The architecture involves controllers, processes, and agents that can autonomously interact and make decisions. This helps achieve more resilient and adaptive industrial processes.

9.9 Conclusion

Industrial Automation has undergone significant evolution, from hard-wired relay circuits to modern distributed and agent-based solutions. Each generation has brought greater flexibility, efficiency, and reliability to automation systems (Slide 30). Despite the challenges, such as the high initial cost and maintenance demands, industrial automation continues to advance, driven by the need for enhanced productivity, quality, and safety. The next stages will focus on improving self-configurability, modularity, and integrating intelligent, autonomous systems to enable seamless, adaptive manufacturing environments.

Lecture 10: Industrial Automation 2

10.1 Introduction to Industrial Automation Systems

This lecture focused on advanced elements of industrial automation, covering concepts such as the **IEC 61131 standard**, **Ladder Logic**, and control using **counters** and **timers**. These components are foundational in designing robust and flexible automated systems for industry (Slide 2).

10.2 IEC 61131 Standard

IEC 61131 is an international standard for programmable logic controllers (PLCs) that defines programming languages and structures used for industrial control (Slide 2). It supports a variety of programming approaches, including: - **Ladder Logic (LD)**: A graphical programming language that uses symbols to represent electrical relay logic, making it easy for engineers to visualize and debug control processes. - **Instruction List (IL)**: A low-level textual language similar to assembly, useful for describing control operations in a concise manner (Slide 3). - **Structured Text (ST)**: A high-level, block-structured language similar to traditional programming languages, which makes it ideal for complex logical operations (Slide 3). - **Function Block Diagrams (FBD)** and **Sequential Function Charts (SFC)**: Graphical representations that show the flow of control and parallel operations (Slide 3).

10.3 Ladder Logic and Its Elements

Ladder Logic represents control processes in a form that resembles a ladder. It has been used since the 1940s and is still popular today because of its visual representation, ease of debugging, and intuitive structure (Slides 4-7).

- **Basic Elements**:

  - **Contacts**: Represented as normally open or normally closed switches, contacts control the flow of "power" through each rung of the ladder diagram (Slide 5).
  - **Coils**: Act as outputs, controlling devices like motors or valves based on the conditions specified by the contacts (Slide 5).
  - **Connectors**: Elements arranged either in **series** (AND logic) or **parallel** (OR logic) to determine the conditions under which outputs are activated (Slide 5).

- **Ladder Logic Conventions**:
  - **Power Rails**: Vertical lines representing power flow from left to right across each "rung" of the ladder (Slide 6).
  - **Rungs**: Each rung represents an operation in the control process. Outputs are updated only after evaluating all inputs in the program cycle, ensuring deterministic behavior (Slide 6).
  - Devices are shown in their **normal condition** (inactive or off state) and can appear in multiple rungs as part of different control sequences (Slide 6).

10.4 Example: Pneumatic Cylinder Control

A simple example of using ladder logic to control a **pneumatic cylinder** was provided, showcasing its applicability in real-world automation tasks: - **Desired Behavior**: An LED indicates readiness. When a "START" button is pressed, the cylinder moves forward until fully extended, then returns to its default position. Sensors provide signals for "HOME" (default position) and "END" (fully extended) states (Slide 10). - **State Machine Representation**: The control process can be described using a **Moore state machine**, where inputs trigger state transitions and determine outputs. Boolean logic defines the state transitions, ensuring that the system follows a predictable sequence of actions (Slide 13).

10.5 Ladder Logic Timers and Counters

Timers and **counters** are used in ladder logic for managing time-dependent operations and counting events, such as sensor activations or production cycles.

10.5.1 Timers

- **On-Delay Timer (TON)**: The output turns on after a specified delay time once the input signal is activated. This is used for operations that need a fixed delay before activation, such as starting a motor after a set period (Slide 28).
- **Off-Delay Timer (TOF)**: The output turns off after a delay once the input signal is deactivated. This is useful for keeping an actuator active for a fixed period after the input stops (Slide 30).
- **Pulse Timer (TP)**: Produces a fixed-length output pulse when the input is activated. This is often used in signaling applications where a precise pulse duration is required (Slide 33).

10.5.2 Counters

- **Up-Counters (CTU)**: Count up from zero each time a rising edge of an input is detected until a preset value is reached (Slide 23).
- **Down-Counters (CTD)**: Count down from a preset value to zero, commonly used for decrementing remaining quantities, such as material in a hopper.
- **Resetting Counters**: A reset input brings the counter value back to zero, ensuring it starts counting again from the beginning (Slide 24).

10.6 Sequential Function Chart (SFC)

Sequential Function Charts (SFC) are another important tool for industrial automation, providing a high-level graphical representation of a system's operational sequence. SFCs show the flow from one step to another, detailing the conditions for each transition and enabling a clear view of parallel and sequential operations (Slide 39). - **Parallel Branches**: Allow multiple operations to occur simultaneously, synchronized by transition conditions. These branches are visually represented using double horizontal lines, making it easy to identify concurrent activities in the system (Slide 41).

10.7 Practical Considerations for Ladder Logic

- **Why Ladder Logic Is Still Popular**: Ladder logic is favored due to its **ease of use**, **visual clarity**, and the ability to **trace command flow** intuitively. It is particularly useful for maintenance personnel without deep programming knowledge, as its representation is similar to traditional relay-based control systems (Slide 22).
- **Counters and Timers**: Practical examples include **traffic light controllers** using timers to manage the sequence of lights, and counters to keep track of vehicles or system operations, ensuring safe and efficient process flows (Slide 36).

10.8 Conclusions

This lecture provided an in-depth overview of the tools and techniques used in **industrial automation**, such as **Ladder Logic**, **SFC**, **IEC 61131** programming languages, and the use of **counters** and **timers**. Understanding these concepts is crucial for implementing reliable, flexible, and maintainable automation systems. These methods allow for both the visual representation and precise control of industrial processes, making them fundamental for modern manufacturing environments (Slide 44).

In future discussions, we'll explore more advanced features of automation systems, such as integrating **networked embedded systems** and **agent-based control**, to further enhance flexibility and scalability in industrial applications.