VIT

Vellore Institute of Technology

(Deemed to be University under section 3 of UGC Act, 1956)

Name: Apurba Koirala

Reg no: 22BCE3799

Subject Code: BCSE305L

Course Title: Embedded Systems

Guided by: Dr. M.Narayana Moorthi

Digital Assignment I

(1) What are the design challenges of Embedded System? Compare and contrast topdown approach and bottom –up approach.

Design Challenges of Embedded Systems

Embedded systems face several design challenges due to their specialized nature and application requirements. Real-time constraints are a significant challenge, as these systems often need to perform tasks within strict time limits. Resource limitations, such as restricted memory, processing power, and energy, further complicate design, requiring optimization at every level. Power consumption is another critical consideration, especially for battery-operated devices, where efficiency directly impacts operational longevity. Cost constraints also play a crucial role, as embedded systems often need to balance performance and functionality while adhering to tight budgets. Additionally, system reliability and stability are paramount, particularly in applications where failure is not an option.

Designers must also address interfacing challenges, ensuring seamless communication between hardware peripherals and the processor. Security is an increasing concern, as embedded systems must protect against unauthorized access and cyber threats. Scalability and upgradability are also important, enabling the system to adapt to future requirements without significant redesign. Finally, environmental factors and durability must be considered, as embedded systems often operate in harsh or demanding conditions, requiring robust and reliable designs.

Comparison of Top-Down and Bottom-Up Design Approaches

In embedded system design, the **top-down approach** and **bottom-up approach** represent two distinct methodologies. The top-down approach begins with the overall system design and breaks it down into smaller subsystems and components. This method focuses on system-level functionality and ensures a clear understanding of the system's goals and architecture. It moves from high-level abstraction to detailed implementation, offering design consistency and clarity. However, it can sometimes overlook specific component-level issues early in the process and relies on a strong initial system definition.

In contrast, the bottom-up approach starts with the design of individual components or modules, which are later integrated into the complete system. This approach emphasizes component-level functionality, allowing for the reuse of well-defined modules and enabling parallel development. However, it may face integration challenges, and critical system-level requirements might be missed during the early stages.

The top-down approach is ideal for complex systems where overall functionality is critical, such as designing an entire embedded system for a smart appliance. On the other hand, the bottom-up approach suits projects with well-defined, reusable components, like developing a sensor interface module. In practice, embedded system design often combines these approaches to leverage their strengths, ensuring both system-level coherence and efficient component integration.

(2) How to measure the execution time and speed of a program running on a processor?

Measuring the execution time and speed of a program running on a processor is a vital aspect of performance analysis and optimization, especially in scenarios where efficiency and precision are critical. There are several methods available, depending on the level of detail required and the tools at hand. One commonly used technique is leveraging the built-in timers integrated into modern processors. These timers, such as the time-stamp counter (TSC) on x86 architectures or the performance monitoring units (PMUs) on ARM processors, can provide high-resolution measurements by counting clock cycles or tracking elapsed time. The process involves recording the timer value before the program starts execution, running the program or code segment, and then recording the timer value after completion. The difference between the start and end values represents the execution time, typically in nanoseconds or microseconds.

Another widely used approach is employing software libraries available in programming languages. These libraries simplify time measurement by providing built-in functions that record and compute the duration of code execution. For instance, in C++, the <chrono> library is commonly used, while in Python, the time module or timeit library offers similar functionality. These libraries are particularly useful for measuring execution times in user-level programs, allowing developers to focus on optimizing specific portions of code.

Profiling tools are also essential for detailed performance analysis. Tools such as Gprof, Valgrind, Perf, and Intel VTune Profiler go beyond measuring raw execution time by analyzing resource usage, identifying bottlenecks, and offering insights into the program's behavior at the system level. They can measure not only the total execution time but also the time spent in individual functions or modules, making them invaluable for complex applications. For Windows-based systems, Visual Studio Profiler provides a robust set of features to profile programs and understand their performance dynamics.

In embedded systems, where precise timing and minimal overhead are crucial, hardware-based methods are often preferred. Hardware debuggers or oscilloscopes can directly measure execution time by observing the behavior of specific signals. For example, toggling a GPIO pin before and after a code segment can provide a clear start and stop signal, which can then be measured using an oscilloscope or logic analyzer. These methods are particularly useful for systems that operate in real-time environments or have strict timing constraints.

Additionally, the speed of a program, often expressed in terms of instructions per second or cycles per instruction (CPI), provides another layer of performance analysis. CPI is calculated by dividing the total clock cycles by the number of instructions executed. The execution time can be further derived using the formula that relates CPI, instruction count, and clock speed. Tools like Perf and Intel VTune Profiler can measure these metrics, offering insights into the processor's efficiency and the program's computational intensity.

Simulation tools like QEMU and Simics are valuable during the development phase, especially for systems that are not yet fully implemented in hardware. These tools allow developers to simulate the processor environment, measure performance metrics, and make adjustments to optimize execution. Such simulations are particularly useful for analyzing the behavior of programs in systems with complex architectures.

To ensure accurate measurements, it is important to follow certain best practices. First, the measurement process itself should introduce minimal overhead to avoid distorting the results. Multiple runs of the program should be performed to account for variability due to factors like caching, operating system scheduling, or other environmental influences. Furthermore, measurements should be context-aware, as factors like background processes or hardware contention can affect timing results. Combining these methods and adhering to best practices ensures a comprehensive understanding of the program's execution time and speed, enabling targeted optimizations and improved performance.

(3) Develop a requirements description for a device suitable for household appliance or computer peripheral.

A household appliance or computer peripheral device must be designed with a clear understanding of its purpose, target audience, and functional requirements. The primary requirement is functionality, which ensures the device performs its intended tasks reliably and efficiently. For instance, a smart thermostat should accurately measure room temperature, provide user-friendly controls for adjusting settings, and support scheduling to optimize energy use. Similarly, a computer peripheral like a wireless mouse must offer precise cursor movement, ergonomic design for user comfort, and reliable wireless connectivity.

Ease of use is another crucial requirement, as the device should cater to a diverse range of users, including those with limited technical expertise. Intuitive interfaces, clear labeling, and minimal setup requirements enhance the user experience. For smart appliances, integration with home automation systems, such as support for voice commands through virtual assistants or compatibility with smart home hubs, adds significant value. For peripherals, plug-and-play functionality without the need for extensive configuration is desirable.

Durability and safety are essential, especially for household appliances that operate in dynamic environments. Devices must adhere to safety standards, such as protection against electrical hazards and overheating, and be robust enough to withstand daily wear and tear. For peripherals, durability ensures the device remains operational despite frequent use. Additionally, devices should be energy-efficient to minimize environmental impact and reduce operational costs. For battery-powered devices, optimizing power consumption to extend battery life is a critical consideration.

Connectivity and compatibility also play a significant role in determining the success of a device. For smart appliances, support for Wi-Fi, Bluetooth, or other communication protocols is essential for remote control and monitoring. For computer peripherals, compatibility with various operating systems and devices ensures widespread usability. Furthermore, software updates or firmware upgrades should be seamlessly deliverable to keep the device secure and functional over time.

Lastly, affordability and aesthetics contribute to the appeal of the device. While maintaining high standards of functionality and durability, the device should remain cost-effective for the target market. A visually appealing design that aligns with modern household or workspace aesthetics further enhances its desirability. By meeting these requirements, the device can effectively address user needs and stand out in a competitive market.

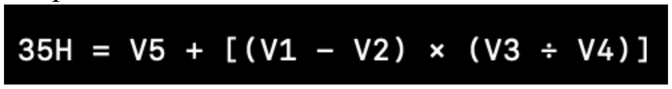
(4) Write an assembly code to implement the following expressions. X = a+ (b-c)*(e/f) using 8051 embedded c and assembly code?

For the given expression,

Assembly code:

```
ORG 0000H
   MOV A, 31H
   MOV R0, 32H
   SUBB A, @R0
   MOV R1, A

   MOV A, 33H
   MOV B, 34H
   DIV AB
   MOV R2, A

   MOV A, R1
   MOV B, R2
   MUL AB
   MOV R3, A

   MOV A, 30H
   ADD A, R3
   MOV 35H, A

   SJMP $

END
```

Output:

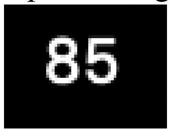


```
35H = V5 + [(V1 − V2) × (V3 ÷ V4)]
```

8051 Embedded C code:

```
#include <stdio.h>
#include <reg51.h>

void main() {
   unsigned char a = 10, b = 20, c = 5, e = 40, f = 8;
   unsigned char X;

   unsigned char temp1 = b - c;
   unsigned char temp2 = e / f;
   unsigned char result = temp1 * temp2;
   X = a + result;

   while (1);
}
```

Ouput:
Upon adding the print statement:



```
85
```

(5) Compare and contrast the following microcontrollers : 8051 , PIC and ARM

The 8051, PIC, and ARM microcontrollers are widely used in embedded systems, each with distinct features and capabilities suited to different applications. The 8051 is based on an 8-bit architecture with a Harvard structure, offering separate program and data memory. It operates at low clock speeds, typically between 12 MHz and 33 MHz, and is suitable for simple tasks requiring basic processing power. The 8051 provides limited memory, with up to 64 KB of program memory and 256 bytes of data memory, and it is ideal for cost-sensitive and small-scale applications such as basic automation and timers. However, it lacks advanced power-saving modes and debugging features, which makes it less suitable for modern, complex systems.

PIC microcontrollers, on the other hand, come in 8-bit, 16-bit, and 32-bit architectures and are based on a RISC (Reduced Instruction Set Computing) design. They are known for their efficient code execution and pipelining capabilities. PIC microcontrollers offer a wide range of performance options, with clock speeds up to 64 MHz or higher, and provide flexible memory configurations, from a few kilobytes to several megabytes, depending on the model. Their advanced power-saving features make them a popular choice for power-sensitive applications such as motor control, industrial automation, and

IoT devices. The development ecosystem for PIC, including tools like MPLAB IDE and XC compilers, further enhances its versatility.

ARM microcontrollers are built on 32-bit or 64-bit RISC architectures and are designed for high-performance applications. They often operate at clock speeds exceeding 1 GHz, making them ideal for resource-intensive tasks. ARM microcontrollers feature large on-chip memory and support for external memory interfaces, enabling them to handle complex applications that require significant storage and processing power. They also provide sophisticated power management features, balancing high performance with energy efficiency. The extensive development ecosystem for ARM, which includes tools like Keil, IAR Embedded Workbench, and GCC, along with a large developer community, ensures robust support for developers. ARM microcontrollers are widely used in advanced applications such as smartphones, automotive systems, and IoT devices.

In terms of cost, the 8051 is the most affordable option, making it attractive for budget-constrained projects. PIC microcontrollers are moderately priced, offering a balance between cost and performance. ARM microcontrollers are generally more expensive, particularly for high-performance models, but their capabilities make them cost-effective for advanced applications. Overall, the choice between these microcontrollers depends on the specific application requirements, such as processing power, memory, energy efficiency, and cost constraints. While the 8051 is suitable for basic tasks, PIC excels in mid-range applications, and ARM dominates in high-performance, complex systems.