**Logo

Description automatically generated**

**San Francisco Bay University**

**CE450 Fundamentals of Embedded Engineering**

**2024 Spring Midterm Exam**

**Student Name: MD MANIRUZZAMAN ID: 20099**

1. What is embedded system basic requirements? What is ARM processor architecture?

**Answer:**

**Embedded system basic requirements:**

An embedded system's basic requirements can change based on the application and domain, but these systems need the following basic requirements:

1. Real time operation:

A lot of embedded systems must react to inputs or changes in their environment in real-time, which means they must do it in a set amount of time. For applications like automotive control systems, where delayed reactions could cause accidents, real-time processing is essential.

1. Resource Constraints:

Memory, processing speed, and energy consumption are frequently constrained in embedded systems. Since many of these systems are battery-powered or require extended periods of time without maintenance, it is imperative that these resources be used as efficiently as possible.

1. Reliability and Stability:

Because embedded systems are often used in critical applications (industrial control systems, medical devices), they need to be extremely stable and reliable. These systems must have the capacity to correct mistakes on their own without the need for human intervention because they frequently operate around the clock.

1. Customizability and Scalability:

Scalability and customization are essential for embedded systems to support requirements for future upgrades and modifications.

1. Security:

Security becomes a vital requirement as more embedded systems are linked to other networks or IoT devices. These systems must be shielded from cyberattacks, illegal access, and data breaches.

1. Cost-effectiveness:

Maintaining a low cost per unit is crucial because embedded systems are frequently produced in large quantities. This includes the cost of development and maintenance throughout the system's life cycle in addition to the component costs.

1. User Interface and Experience:

Embedded systems that engage with users, like consumer electronics and interactive kiosks, must offer an intuitive user interface and a satisfying user experience.

1. Environmental Conditions:

The environmental conditions of an embedded system's application, such as high or low temperatures, high humidity, or vibration, must be taken into consideration during the design process.

**ARM processor architecture:**

The Arm architecture defines a set of guidelines that control the hardware's behavior during the execution of a specific instruction. It is an agreement outlining the terms of interaction between the software and the hardware.

A family of RISC computer processor architectures, the ARM family is well-known for its power efficiency, which makes it ideal for embedded and mobile devices.

The ARM processor architecture includes the following important features:

RISC Principles:

The simplicity and efficiency of RISC architecture serve as the foundation for the design of ARM processors.

Power Efficiency:

The power efficiency of ARM architecture is one of its defining characteristics, making it perfect for battery-operated devices like smartphones, tablets and other portable electronics.

Licensing Model:

Instead of producing processors, ARM Holdings licenses the ARM architecture to other businesses, who then use the ARM cores in their own product designs.

Cores and Families:

ARM architecture is divided into multiple families and generations, such as ARMv7, ARMv8, and others. There are various cores within these families, such as Cortex-A for high-performance applications, Cortex-R for real-time applications, and Cortex-M for microcontroller applications.

64-bit Support:

64-bit computing was made possible by ARMv8, which gave ARM processors the ability to process larger data sets and perform better in specific kinds of applications.

Customization and Extensions:

The ARM architecture, licensees can differentiate their products by customizing their implementations to some extent.

Ecosystem:

It is easier to design and implement ARM-based solutions to the broad adoption of ARM processors, which has produced a robust ecosystem of software, operating systems, development tools, and IP components.

For most mobile devices and an increasing number of other applications, the ARM architecture is the preferred choice due to its exceptional power efficiency, responsiveness, and adaptability.

---------------------------------------------------------------------------------------------------------------------------

1. Analyze the following code step by step, like running it in debugging mode in Python compiler. And then give the result through line-by-line explanation

*def Ton(now):*

*then = 42*

*def no(know):*

*no = then*

*return know \* now(know)*

*return no*

*>>> then, no = 7, 4*

*>>> now = lambda oh: oh \* no*

*>>> ok = Ton(now)(no)*

*>>> ok*

*?*

**Answer:**

def Ton(now):

    then = 42

    def no(know):

        return know \* now(know)

    return no

then, no = 7, 4

now = lambda oh: oh \* no

ok = Ton(now)(no)

print(ok)

# output : 64

**Line by line:**

* Define a function named “Ton” with single argument is “now”.
* Assign value of “then” and “no” are “7” and “4” respectively.
* Lambda function execute “now” and multiply it’s input by global “no” = 4
* Call the function “Ton(now)” take the value of “then” = 42
* Execute the function “def no(know)” and return the value “no” nested function.”
* Within nested function “no” , “know” is set to 4, value pass the function.
* Now(know) called within the nested function “no”, and execute lambda function “now” with “oh” set the value to 4. And calculate: 4\*4 = 16
* The nested function “no” then calculate “know” \* now(know) and calculation will be: 4\*16 = 64
* Result is ok = Ton(now)(no) is 64

---------------------------------------------------------------------------------------------------------

1. As above required, analyze the following code.

*woo = 6*

*def much(woo):*

*if much == woo:*

*such = lambda woo: 5*

*def woo():*

*return such*

*return woo*

*such = lambda woo: 4*

*return woo()*

*>>> woo = much(much(much))(woo)*

*>>> woo*

*?*

**Answer:**

woo = 6

def much(woo):

    if much == woo:

        such = lambda woo: 5

        def woo():

            return such

        return woo

    such = lambda woo: 4

    return woo()

woo = much(much(much))(woo)

print(woo)

# Output : 5

**Explain:**

* Declare variable “woo” has an integer value = 6
* Define a function “much” with parameter “woo”
* Condition checking for object “much” is equal to parameter “woo” and will always evaluate to “False” under execution because “much” is a function object and “woo” is integer.
* Execute woo = much(much(much))(woo) where, “much” as an argument and noy align. Until “much” excepts a integer value, passing “much” to itself not satisfy any logical way.
* Inner most “much(much)” will execute first, until “much” == “woo”
* Calling the function “(woo)” where woo=6 and passed the argument
* Then print(woo) will execute and print the output = 5

-------------------------------------------------------------------------------------------------------

1. What is the running result after analyzing step by step as required in question*#1*?

*def horn(hood):*

*horn = hood*

*def hood (horn):*

*return horn*

*return horn(hood)*

*hood = lambda horn: horn(2)*

*>>> horn (hood)*

*?*

**Answer:**

def horn(hood):

    horn = hood

    def hood(horn):

        return horn

    return horn(hood)

hood = lambda horn: horn(2)

print(horn(hood))

# Output : 2

Explanation:

* Define a function “horn” where parameter is “hood”.
* Call the function “horn(hood)”, then pass the lambda function “hood” as an argument to the “horn” function.
* In the “horn” function the argument “horn” is assigned to local variable “horn”.
* The “horn" function returns the function “hood” which takes the argument “horn”
* Since called “horn” with the lambda function “hood” as an argument, the function “hood” defined with lambda function “hood” as it is an argument.
* Execute the inner function “hood” as “lambda horn: horn(2)”.
* Call result = horn(hood), actually call result = lambda horn : horn (2) (2).
* Lambda function “lambda horn: horn(2) returns the function “horn” with the argument “2”.
* The argument “horn” is the function “lambda horn: horn (2)” the expression “horn(2)” returns the function “lambda horn: horn(2)” again. So it return the result by calling “horn” with the argument “2”
* Then, “lambda horn: horn(2)(2) returns the integer “2” and the Output = 2

------------------------------------------------------------------------------------------------------------

1. What is the running result? And explain why as above

*pear = "ni"*

*def apple(banana):*

*def plum(peach):*

*pear = lambda pear: peach(pear)*

*return pear*

*return plum(banana)("ck")*

*>>>apple(lambda peach: pear + peach)*

*?*

**Answer:**

pear = "ni"

def apple(banana):

    def plum(peach):

        pear = lambda pear: peach(pear)

        return pear

    return plum(banana)("ck")

print(apple(lambda peach: pear + peach))

# Output : “nick”

**Explanation:**

- Define a string “ni” with variable “pear”

- Define a function “apple” with argument “banana”.

- inside the “apple” function defined another function “plum” with argument “peach”

- “plum” function defined a lambda function “pear” with argument “pear” and return the result by calling “peach” with the argument “pear”.

The “plum” function returns the lambda function “pear” when called with argument “banana” and “apple” function then call lambda function “pear” with argument “ck” and return the result. Finally, print the output by calling “apple” with lambda function “lambda peach: pear + peach” as argument.

Step by step:

* When call “apple(lambda peach: pear + peach” passed the lambda function as an argument to “apple” function.
* In the apple function the argument “banana” is the lambda function “lambda peach: pear + peach”. Then “apple” function defines inner function “plum” with argument “banana” and inner function will be “plum” as “def plum(peach): return lambda pear: peach(pear)”.
* The “apple” function then call “plum(banana)(“ck”) and lambda function return concentration of “pear” and “peach” when called.
* The argument “peach” is the string “ck” the lambda function “lambda peach: pear+peach” return “nick”, so, the expression “plum(lambda peach: pear+peach)(“ck”) is evaluate to “nick”.
* The inner function “plum” returns the lambda function “pear” when call with argument “nick”
* Then, the lambda function “pear” as “lambda pear: (lambda pear: “nick”)(pear)”
* When call “(lambda pear: “nick”)(“ni”), since “pear” is a string “ni”
* The lambda function “lambda pear: “nick” returns the string “nick” when called “ni”
* So, the expression “ lambda pear: (lambda pear: “nick”)(pear) returns the string “nick” as output.

------------------------------------------------------------------------------------------------------------

1. What would Python print? And explain why as above

*x = "x"*

*g = x*

*def x(x):*

*g = "h"*

*if x == g:*

*return x + "i"*

*x = lambda x: x(g)*

*return lambda g: x(g)*

*>>> x = x(x)(x)*

*>>> x*

*?*

**Answer:**

x = "x"

g = x

def x(x):

    g = "h"

    if x == g:

        return x + "i"

    x = lambda x: x(g)

    return lambda g: x(g)

x = x(x)(x)

print(x)

# output: hi

**Explanation:**

* x = "x": Assigns the string "x" to the variable x.
* g = x: Assigns the value of x (which is "x") to the variable g.
* This function is named x and takes a parameter also named x.
* Inside the function:
* g = "h": Assigns the string "h" to the variable g.
* if x == g: Checks if the parameter x is equal to the local variable g.
* If they are equal, it returns the concatenation of x and "i".
* Otherwise:
* x = lambda x: x(g): Defines a lambda function that takes a parameter x and returns x(g).
* Returns another lambda function that takes a parameter g and calls the lambda function defined previously with the argument g.
* x = x(x)(x): Assigns the result of calling the x function with itself as the argument twice to the variable x.
* First, it calls x(x), which goes through the function definition and returns a lambda function.
* Then, it calls the returned lambda function with x again, which results in a lambda function assigned to x.
* print(x): Prints the value of x, which is a lambda function.
* In summary, the code defines a function x that returns a lambda function under certain conditions. It then calls this function recursively with itself as arguments and assigns the result to x. Finally, it prints the resulting lambda function.

---------------------------------------------------------------------------------------------------------

1. Define a function with one argument like a positive number *x* and returns the powers of two, which is nearest to *x*. If *x* is exactly between two powers of two, return the larger.

*def nrst\_two(x):*

*"""*

*>>> nrst\_two(8) # 2^3 is 8*

*8.0*

*>>> nrst\_two (11.5) # 11.5 is closer to 8(=2^3) than 16(=2^4)*

*8.0*

*>>> nrst\_two (14) # 14 is closer to 16 than 8*

*16.0*

*>>> nrst\_two (2019) # 2^10 = 1024; 2^11=2048.0*

*2048.0*

*>>> nrst\_two (0.1)*

*0.125*

*>>> nrst\_two (0.75) # Tie between ½(=) and 1(=)*

*1.0*

*>>> nrst\_two (1.5) # Tie between 1 and 2(=)*

*2.0*

*"""*

***Answer:***

import math

def nrst\_two(x):

    lower\_power = 2 \*\* math.floor(math.log2(x))

    upper\_power = 2 \*\* math.ceil(math.log2(x))

    if x - lower\_power < upper\_power - x:

        return lower\_power

    else:

        return upper\_power

print(nrst\_two(8))

print(nrst\_two(11.5))

print(nrst\_two(14))

print(nrst\_two(2019))

print(nrst\_two(0.1))

print(nrst\_two(0.75))

print(nrst\_two(1.5))

*# output:*

*8*

*8*

*16*

*2048*

*0.125*

*1*

*2*

***Explanation:***

*import math: This imports the math module, which provides mathematical functions and constants for various operations.*

*def nrst\_two(x):: This defines a function named nrst\_two that takes a single argument x.*

*lower\_power = 2 \*\* math.floor(math.log2(x)): This line calculates the lower power of 2 that is less than or equal to the given number x. It first computes the base-2 logarithm of x using math.log2(x), then takes the floor of this logarithm using math.floor() to get the largest integer less than or equal to the logarithm, and finally raises 2 to the power of this result using 2 \*\*. This gives us the nearest power of 2 that is less than or equal to x.*

*upper\_power = 2 \*\* math.ceil(math.log2(x)): This line calculates the upper power of 2 that is greater than or equal to the given number x. It computes the base-2 logarithm of x using math.log2(x), then takes the ceiling of this logarithm using math.ceil() to get the smallest integer greater than or equal to the logarithm, and finally raises 2 to the power of this result using 2 \*\*. This gives us the nearest power of 2 that is greater than or equal to x.*

*if x - lower\_power < upper\_power - x:: This line checks if the difference between x and the lower power of 2 is less than the difference between the upper power of 2 and x. This comparison determines which of the two powers of 2 (lower\_power or upper\_power) is closer to x.*

*return lower\_power: If the lower power of 2 is closer to x, this line returns the lower power of 2.*

*else:: If the upper power of 2 is closer to x, this line specifies what to do next.*

*return upper\_power: If the upper power of 2 is closer to x, this line returns the upper power of 2.*

*The print() function calls nrst\_two with various arguments and prints the results.*

*----------------------------------------------------------------------------------------------------------*

1. Create a function in recursion to check whether input argument is palindrome if it reads the same forwards and backwards.

*def is\_plndrm(n):*

*"""*

*>>> is\_plndrm (45654)*

*True*

*>>> is\_plndrm (42)*

*False*

*>>> is\_plndrm (2019)*

*False*

*>>> is\_plndrm (10101)*

*True*

*"""*

**Answer:**

def is\_plndrm(n):

    n\_str = str(n)

    return n\_str == n\_str[::-1]

print(is\_plndrm(45654))

print(is\_plndrm(42))

print(is\_plndrm(2019))

print(is\_plndrm(10101))

*# Output:*

*True*

*False*

*False*

*True*

***Explanation:***

The first line converts n to a string and assigns it to a variable n\_str. This is done because strings have a built-in method to reverse themselves, while numbers do not.

The second line returns the result of comparing n\_str with n\_str[::-1]. The expression n\_str[::-1] reverses the string by using a negative step of -1 in slicing. For example, if n\_str is "123", then n\_str[::-1] is "321".

If n\_str and n\_str[::-1] are equal, it means that n is a palindrome, so the function returns True. Otherwise, it returns False.

1. Define a function "*has\_subls*" in recursive call with two lists as arguments, *ls* and s*ubls*, and returns if the elements of *subls* appear in order anywhere within *ls*.

*def has\_subls (ls, subls):*

*"""Returns if the elements of subls appear in order anywhere within list ls.*

*>>> has\_subls ([], [])*

*True*

*>>> has\_subls ([3, 3, 2, 1], [])*

*True*

*>>> has\_subls ([], [3, 3, 2, 1])*

*False*

*>>> has\_subls ([3, 3, 2, 1], [3, 2, 1])*

*True*

*>>> has\_subls ([3, 2, 1], [3, 2, 1])*

*True*

*"""*

***Answer:***

def has\_subls(ls, subls):

    if not subls:

        return True

    if not ls:

        return False

    if ls[0] == subls[0]:

        return has\_subls(ls[1:], subls[1:])

    else:

        return has\_subls(ls[1:], subls)

print(has\_subls([], []))

print(has\_subls([3, 3, 2, 1], []))

print(has\_subls([], [3, 3, 2, 1]))

print(has\_subls([3, 3, 2, 1], [3, 2, 1]))

print(has\_subls([3, 2, 1], [3, 2, 1]))

*# output:*

*True*

*True*

*False*

*True*

*True*

***Explanation:***

The has\_subls function takes two arguments: ls and subls. The function checks if the sublist subls appears in order within the list ls. The function returns True if subls appears in ls, and False otherwise.

In the first example, print(has\_subls([3, 3, 2, 1], [])), the function checks if the empty list [] appears in the list [3, 3, 2, 1]. Since every list contains the empty list, the function returns True.

In the second example, print(has\_subls([], [3, 3, 2, 1])), the function checks if the list [3, 3, 2, 1] appears in the empty list []. Since the empty list does not contain any elements, the function returns False.

In the third example, print(has\_subls([3, 3, 2, 1], [3, 2, 1])), the function checks if the list [3, 2, 1] appears in the list [3, 3, 2, 1]. Since the sublist [3, 2, 1] appears in order within the list [3, 3, 2, 1], the function returns True.

In the fourth example, print(has\_subls([3, 2, 1], [3, 2, 1])), the function checks if the list [3, 2, 1] appears in the list [3, 2, 1]. Since the sublist [3, 2, 1] appears in order within the list [3, 2, 1], the function returns True.

Note that the has\_subls function checks if the sublist appears in order within the list. Therefore, the sublist [2, 3] would not be considered a match in the list [3, 2, 1], even though it contains the same elements. The function would return False for print(has\_subls([3, 2, 1], [2, 3])).