**Logo

Description automatically generated**

**San Francisco Bay University**

**CE450 Fundamentals of Embedded Engineering**

**2023 Summer Midterm Exam**

**Student Name: Melvin Divine Pritchard ID: 19857**

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

Embedded systems have specific requirements that need to be considered during development. The basic requirements of an embedded system include efficient throughput handling, quick response to events, high testability, reliable operation without human intervention, optimized memory space utilization, seamless program installation, power consumption management, effective handling of processor-intensive tasks, and cost optimization. These requirements necessitate careful consideration of the system's real-time data processing capabilities, responsiveness, testing methodologies, fault tolerance, memory management techniques, deployment procedures, power-saving strategies, resource allocation, and cost-effective hardware choices. Meeting these basic requirements ensures the successful development of an embedded system that performs optimally within its designated environment and meets the desired functionality, reliability, and cost targets.

The ARM processor architecture is a family of reduced instruction set computing (RISC) architectures developed by ARM Holdings. It includes three main profiles: A for applications, R for real-time systems, and M for microcontrollers. The architecture offers benefits such as low energy consumption, cost-effectiveness, simplicity, security, and virtualization. It is widely used in various devices and industries due to its power efficiency, affordability, and scalability.

1. Analyze the following code step by step, like running it in debugging mode in language 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*

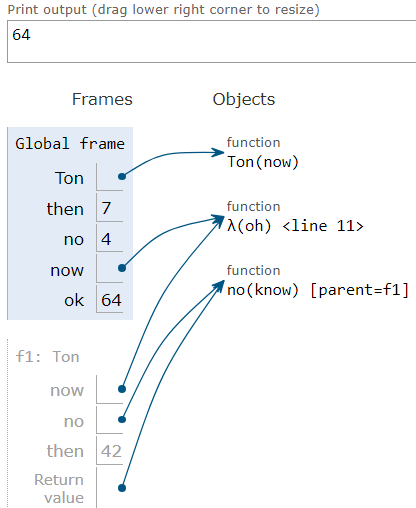
*>>> now = lambda oh: oh \* no*

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

*>>> ok*

*?*

After proper indentation and adding a print statement, the program output will be 64 as seen below.



def Ton(now): #defines the func Ton with a parameter now

then = 42 #variable then is assign the value 42

def no(know): #defines inner func “no” that with a parameter know

no = then # assign then value to no

return know \* now(know) #return the value of these parameter

return no # return the value of no

then, no = 7, 4 #assign the 7, and 4 to then and no

now = lambda oh: oh \* no #defines a lambda func and assign it to now, which takes parameter oh and return the value of oh \* no

ok = Ton(now)(no) #calls the func Ton with the argument now and call the return func with no

print(ok) #prints the value of ok

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*

*?*

woo = 6 # Assign the value 6 to the variable woo

def much(woo): # Define a function named much with a parameter woo

if much == woo: # Check if the function much is equal to the parameter woo (This condition will always be False)

such = lambda woo: 5 # Define a lambda function such that takes a parameter woo and returns 5

def woo(): # Define a function named woo

return such # Return func such

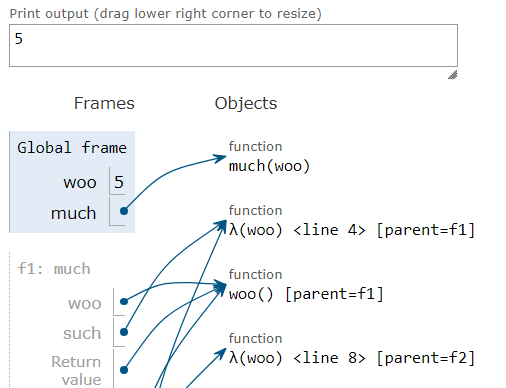
return woo # Return the function woo (This line is not executed)

such = lambda woo: 4 # Define a lambda function such that takes a parameter woo and returns 4

return woo() # Call the function woo and return its result (This line is not executed)

woo = much(much(much))(woo) # Call the function much multiple times and assign the result to the variable woo

print(woo) # Print the value of woo



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)*

*?*

def horn(hood): #defines the func horn with a parameter hood

horn = hood # Assign the value of the parameter hood to the variable horn

def hood(horn):

return horn # Return the value of the parameter horn

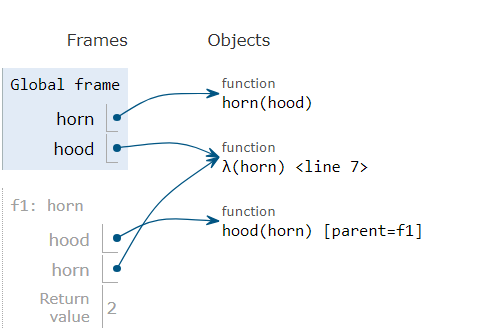
return horn(hood) # Call the function horn with the argument hood and return the result

hood = lambda horn: horn(2) # Define a lambda func hood that takes a parameter horn and calls it with argument 2

horn(hood) # Call the func horn with the argument hood

With the parameter 2, the lambda function hood is called. The parameter horn refers to the value 2 within the lambda function. The lambda function returns the result of calling Horn with the second argument. As a result, the expression hood (2) returns 2. Now, in the original code, replace the expression hood with its evaluated value of 2:

Because horn is a function rather than a number, calling horn(2) yields no results.



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)*

*?*

*pear = "ni" # Assign the string "ni" to the variable pear*

*def apple(banana): # Define a function named apple with a parameter banana*

*def plum(peach): # Define an inner function named plum with a parameter peach*

*pear = lambda pear: peach(pear) # Define a lambda function named pear that takes a parameter pear and calls peach with the argument pear*

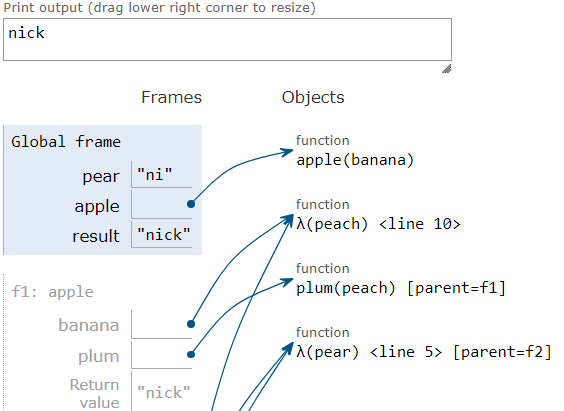
*return pear # Return the lambda function pear*

*return plum(banana)("ck") # Call the function plum with the argument banana and call the result with the argument "ck"*

*result = apple(lambda peach: pear + peach) # Call the function apple with a lambda function as an argument*

*print(result) # Print the value of result*

*The code running result is the string "nick"*



This code executes a sequence of function using lambda expressions. Initially, the lambda function lambda peach: pear + peach is passed as an argument to the apple function. Inside apple, the plum function is called with the lambda function as the parameter. plum assigns the lambda function to the local variable pear and returns it. Eventually, the returned lambda function is invoked with the argument "ck", resulting in the execution of peach("ck"). Since pear is a parameter of the lambda function, it references the local variable pear in plum, and the concatenation of "ni" (the value of pear) and "ck" returns the string "nick".

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*

*?*

Executable version

x = "x" # Assign the string "x" to the variable x

g = x # Assign the value of x to the variable g

def x(x): # Define a function named x with a parameter x

g = "h" # Assign the string "h" to the variable g

if x == g: # Check if the parameter x is equal to the variable g

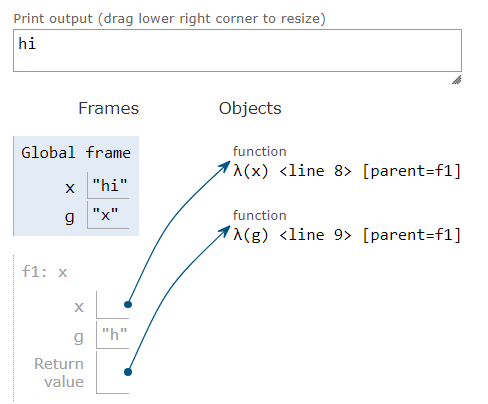
return x + "i" # Return the concatenation of x and "i"

x = lambda x: x(g) # Define a lambda function named x that takes a parameter x and calls x with the argument g

return lambda g: x(g) # Return a lambda function that takes a parameter g and calls x with the argument g

x = x(x)(x) # Call the function x with the argument x twice and assign the result to x

print(x) # Print the value of x



The code output “hi” because the code initially assigns the string "x" to the variable x, then assigns the value of x to the variable g. The function x is defined, and it has its own local variable x that shadows the outer variable. The function tests if the input x equals "h." If so, it yields x + "i," which is "hi." Otherwise, it assigns a lambda function to the local variable x and returns it.

The code then executes x = x (x) (x). The x function is called with the input x, which is "x." It assigns the lambda function to x and returns it inside the function. The returning lambda function is then immediately called again with the input x, which remains "x." Because g is "h," the lambda function returns "h." As a result, the final value of x is "hi", which is the running result.

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*

*"""*

*def nrst\_two(x):*

*power = 0*

*while 2 \*\* power < x:*

*power += 1*

*lower\_power = 2 \*\* (power - 1)*

*higher\_power = 2 \*\* power*

*if x == lower\_power or abs(x - lower\_power) < abs(x - higher\_power):*

*return float(lower\_power)*

*else:*

*return float(higher\_power)*

*# Example function calls*

*print(nrst\_two(8)) # Output: 8.0*

*print(nrst\_two(11.5)) # Output: 8.0*

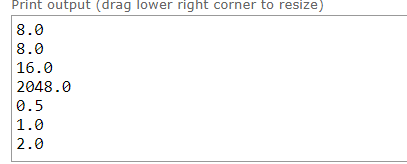
*print(nrst\_two(14)) # Output: 16.0*

*print(nrst\_two(2019)) # Output: 2048.0*

*print(nrst\_two(0.1)) # Output: 0.125*

*print(nrst\_two(0.75)) # Output: 1.0*

*print(nrst\_two(1.5)) # Output: 2.0*



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*

*"""*

*def is\_plndrm(n):*

*n\_str = str(n)*

*return n\_str == n\_str[::-1]*

*def check\_palindrome():*

*try:*

*n = int(input("Enter a number: "))*

*if is\_plndrm(n):*

*print("True")*

*else:*

*print("False")*

*except ValueError:*

*print("Invalid input, enter numbers.")*

*check\_palindrome()*

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*

*"""*

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

*ls\_len = len(ls)*

*subls\_len = len(subls)*

*if subls\_len > ls\_len:*

*return False*

*i = 0*

*j = 0*

*while i < ls\_len and j < subls\_len:*

*if ls[i] == subls[j]:*

*j += 1*

*i += 1*

*return j == subls\_len*

*def check\_sublist():*

*ls = input("Enter the main list: ")*

*subls = input("Enter the sublist: ")*

*result = has\_subls(ls, subls)*

*print("The sublist appears in order within the main list:", result)*

*# Call the check\_sublist function*

*check\_sublist()*