PROJECT

Computer Architecture: Instruction Set Architecture

Ultra Super Calculation Computer

You are a software engineer and the new Ultra Super Calculation Computer (USCC) has just arrived in your office to be put online for use by users at USCC Headquarters. Every day the USCC will receive tons of data in the form of binary numbers and it is expected to perform specific calculations on this data but no one has told you how to do it!

Lucky for you, the engineers over at USCC headquarters have just sent you a copy of the Instruction Set Architecture that they are using. Finally, we can figure out what all those 1's and 0's have meant!

Your job today is to write the code for the CPU that will support the functions required by the ISA. Based on the design specification in the code editor window, you know you are asked to perform five functions:

- Add
- Subtract
- Multiply
- Divide
- Display a history of calculations

These five functions will also require several other support functions to be written as well so that we can access different parts of our computer hardware. We must also be able to:

- Read and split up our incoming data
- Store a binary number to a register
- Access what is stored in the register
- Allocate some registers for a 'history' of our calculations
- Store/Load from the history when needed

Let's get to it!

Tasks

37/37 complete

Mark the tasks as complete by checking them off

Calculator Setup

1.

Familiarize yourself with the ISA template that was sent down from USCC Headquarters, it's in the code editor to the right. You may need to adjust the size of the code editor to view some of the tables and instructions properly. Do that now if you have to.

Your CPU will receive instructions in the form of a Python string and not an actual number. Also, you use two built-in functions that you are familiar with from the Binary lesson, int() and bin(), check out the documentation to brush up on those functions if need be.

2.

Your first task is to create a Python class that you will use to instantiate your calculator when it is to be used.

This will allow each of the users at USCC HQ the ability to have their own personalized calculator.

Just below the last comment, create the class UltraSuperCalculator.

Within the class, define the __init__() constructor method. It will have two parameters: self and name and return None.

Create the instance variable name and set it equal to the argument name. When a USCC user creates their calculator, this is where they will store their name.

Hint

```
class UltraSuperCalculator:
 def __init__(self, name) -> None:
  # Instance variables begin with `self` indicating
  # they are accessed by references to a specific instance
  self.name = name
```

Copy to Clipboard

3.

You've got a name! Now let's set up your registers according to the ISA documentation. Still within the __init__ method, create two more instance variables, number_registers and history_registers.

Both sets of registers will be represented by lists of zeros equal to the number of registers allocated in the documentation. Remember, normally your registers are hardwired memory directly on the CPU used for quick access.

These lists will simulate our empty registers when the calculator is first turned on.

Hint

```
self.number_registers = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
self.history_registers = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

Copy to Clipboard

4.

Now that you've defined your registers you will need to be able to access those registers easily. Your CPU can only perform functions on data stored in the registers. Since these are simulated as Python lists, you will use the list indexes to store and retrieve data for the operations.

Create three more instance variables inside the __init__() method: numbers_index, history_index, and temp_history_index.

In the ISA documentation, number_registers[0] is always 0. Since we will never change it, we will begin our indexing at 1. Set numbers_index equal to 1.

Set both history indexes equal to 0.

Hint

```
# Used to index through the numbers "registers"
self.numbers_index = 1

# Used to index through the history "registers"
self.history_index = 0
self.temp_history_index = 0
```

Copy to Clipboard

5.

To finish off your constructor method, declare your final instance variable: user_display and set it equal to an empty string. This will be the data that you will return to the user.

Hint

```
self.user_display = ''
```

Copy to Clipboard

Display Data to the User

6.

Now it's time to create a method that will update the terminal window for the user per the System Design parameters.

Define a method update_display() that accepts two parameters, self and to_update. The method will update the instance variable user_display to equal to_update and then print the user_display to the console.

Hint

```
def update_display(self, to_update):
  self.user_display = to_update
  print(self.user_display)
```

Copy to Clipboard

7.

Now that you've created a way to display information to the USCC user, let's set a welcome message for them when they start the calculator.

Go back inside the constructor method, at the bottom, and add a call to your newly created update_display() method.

Pass in your own custom message to the user, try to include their name if you can!

Hint

```
# Inside __init__ method
# Sets inital output message
self.update_display(f"Welcome to {self.name}'s Calculator!")
```

Copy to Clipboard

8.

It’s time to test out our class to make sure that everything works as created.

Below and outside of your class, create a new variable and instantiate your class within it. Give your calculator any name you choose, it should print out the welcome message to the console when you click **Save**.

Hint

```
your_calculator = UltraSuperCalculator("Your Name")
```

Copy to Clipboard

Storing and Loading to Registers

9.

According to the ISA, the CPU can only operate on its registers and there is no allowance for direct access to memory locations or passed in values. This is common in smaller systems like the USCC.

Back inside your class, define a new empty method, store_value_to_register() that has two parameters, self and value_to_store.

10.

You need to ensure that when you store values to your numbers register you meet two requirements:

- Never overwrite the constant 0 stored at index 0
- If all your registers are full, begin overwriting the oldest registers

Inside the store_value_to_register() method, create an if statement that checks if numbers_index is greater than 21. If it is, set it equal to 1.

This accomplishes both register parameters by ensuring our index never goes to 0 and then we loop through the indexes from 1 to 21.

Hint

```
if (self.numbers_index > 21):
 self.numbers_index = 1
```

Copy to Clipboard

11.

The value_to_store parameter will be in the form of a binary number since it is received directly from an instruction.

Convert this to an int and store it into the numbers_index position of number_registers.

This will allow you to quickly access these numbers from the register so you can operate on them.

Hint

```
self.number_registers[self.numbers_index] = int(value_to_store, 2)
```

Copy to Clipboard

12.

One of the important concepts of working with registers and computer hardware is remembering what is stored where. Still inside the store_value_to_register() method, add a print() statement that tells the user what value was stored along with the register address.

Once this is complete, increase the numbers_index by one so that the next number we save will go into the next available register.

Hint

```
print(f"Value: {int(value_to_store,2)} stored in Register: {self.numbers_index}.")
self.numbers_index += 1
```

Copy to Clipboard

13.

Create an empty method, load_value_from_register(), that will be used to load a value from the register for use inside the actual calculation methods. It will have two parameters, self and register_address.

14.

Inside the load_value_from_register() method, create a local variable index that converts the register_address binary number to an int.

Hint

```
index = int(register_address, 2)
```

Copy to Clipboard

15.

Create another local variable, int_value, and set it equal to the number stored in number_registers at the index. Type-cast the value to an int just in case there is a transcription problem on your local computer.

Return int_value at the end of the method.

Hint

```
int_value = int(self.number_registers[index])
return int_value
```

Copy to Clipboard

16.

Define a new method, store_to_history_register() that accepts two parameters, self and result_to_store.

17.

Similar to the store_value_to_register() method, create an if statement that ensures our history_index is not greater than 9, if the condition is true, set it equal to 0.

Hint

```
if (self.history_index > 9):
  self.history_index = 0
```

Copy to Clipboard

18.

Convert the result_to_store parameter to a binary number and store it inside the history_registers list at the history_index.

Hint

```
self.history_registers[self.history_index] = bin(result_to_store)
```

Copy to Clipboard

19.

Increase the history_index by 1 and then set the temp_history_index equal to the history_index. By doing this, you will ensure that after each calculation the history starts at the right location.

Hint

```
self.history_index += 1
self.temp_history_index = self.history_index
```

Copy to Clipboard

Creating the Main Function Methods

20.

Declare a new method, add(), that accepts three parameters: self, address_num1, and address_num2.

Remember that you can only operate on numbers stored in the registers, so you are passing in registry addresses to the method so it can fetch the data.

21.

Next, create two local variables, num1 and num2, and set them equal to the values from the register. This will require a call to load_value_from_register().

Hint

```
num1 = self.load_value_from_register(address_num1)
num2 = self.load_value_from_register(address_num2)
```

Copy to Clipboard

22.

Now that you've retrieved the numbers from memory, create another local variable, calculated_value, and set it equal to the sum of num1 and num2.

This value will also be used to update the user display and store to the history register. At the end of the method, return the calculated_value back out of the method.

Hint

```
calculated_value = num1 + num2
return calculated_value
```

Copy to Clipboard

23.

The multiply() and subtract() methods will look nearly identical to the add() method, except when you create calculated_value, you will need to multiply and subtract instead of add.

Go ahead and create those two methods.

Once complete, create the divide() method as well, but leave it empty.

Hint

Copy and paste the add() method, change the method names, and then change the line calculated_value = num1 + num2 to calculated_value = num1 * num2 and calculated_value = num1 - num2, respectively.

24.

In the divide() method we have to do something a little different to account for a 'Divide By Zero' exception. To handle this, add an if-else statement to your method to catch the error before the compiler does.

Inside the divide() method create the num1 and num2 variables. Finally, create calculated_value below and set it equal to 0.

You will return 0 as the result of the calculation to indicate that a Divide By Zero exception occurred.

Hint

```
def divide(self, address_num1, address_num2):
 num1 = self.load_value_from_register(address_num1)
 num2 = self.load_value_from_register(address_num2)
 calculated_value = 0
```

Copy to Clipboard

25.

Next, add an if statement below calculated_value that:

- Checks if num2 is not equal to 0
  - Inside the if statement, set calculated_value equal to int(num1 / num2)
- Add an else clause that prints an error message to the user
  - Example: print(f"Division by 0 error: {num1}/{num2}.")

Hint

```
if (num2 != 0):
  calculated_value = int(num1 / num2)
else:
  print(f"Division by 0 error: {num1}/{num2}.")
```

Copy to Clipboard

26.

Finally, return calculated_value out of the method.

Hint

```
return calculated_value
```

Copy to Clipboard

27.

The last functional method is the get_last_calculation() method. Create this method with only self as the parameter.

28.

Inside the get_last_calculation() method, the first thing you will do is decrease the temp_history_index value by 1 in order to look backward in history.

Hint

```
self.temp_history_index -= 1
```

Copy to Clipboard

29.

After the decrease, create a local variable, last_value. It will contain two parts, a custom message, and the last calculated value. USCC didn't specify a message, so choose one that you like, here are some suggestions:

- "The last calculated value was: "
- "Last result = "

The second part of last_value will be the value from history_registers that is located at temp_history_index. Use string interpolation to include the value into your message.

Remember that the value stored in the register is in binary and will need to be cast to an int before display.

Hint

This one was a little complicated, take a look at our sample string below to help you out!

```
last_value = f"The last calculated value was: {int(self.history_registers[self.temp_history_index], 2)}"
```

Copy to Clipboard

30.

The last line in the method will be a call to update_display and pass in the last_value variable.

Hint

```
self.update_display(last_value)
```

Copy to Clipboard

Reading the Binary Data

31.

Data from USCC HQ is coming at you in 32-bit long strings, in accordance with the ISA. Declare a method, binary_reader(), that has two parameters, self and instruction.

All data sent from the user will get processed through this method so you will need to create some error handling and instruction manipulation as you continue to build it out.

32.

The first thing to check is that the incoming data is actually a 32-bit instruction. Create an if statement that checks whether or not the instruction is 32-bits in length. If it isn't, update the user display with "Invalid Instruction Length" and return out of the method.

Hint

```
if (len(instruction) != 32):
 self.update_display("Invalid Instruction Length")
 return
```

Copy to Clipboard

33.

Now that you know the instruction is 32-bits long, create five local variables inside the binary_reader() method:

- opcode -source_one -source_two -store -function_code

Set each one equal to the correct characters from the instruction argument. Use the ISA documentation.

Hint

To split a string use: stringName[starting_index:ending_index], remembering that the ending index is not included in the resulting string.

```
opcode = instruction[0 : 6]
source_one = instruction[6 : 11]
source_two = instruction[11 : 16]
store = instruction[16 : 26]
function_code = instruction[26:]
```

Copy to Clipboard

34.

The first part of the instruction you want to read is the OPCODE. Create an if-else statement that checks the following conditions:

- if opcode equals '000001': call store_value_to_register(store) and return
- elif opcode equals '100001': call get_last_calculation() and return
- elif opcode DOES NOT equal '000000': call update_display("Invalid OPCODE") and return

This will ensure that the only instructions with an opcode of '000000' will make it to the next lines of code.

To check your conditionals are working as intended, see the hint for some tests to run.

Hint

```
if (opcode == '000001'):
  self.store_value_to_register(store)
  return
elif (opcode == '100001'):
  self.get_last_calculation()
  return
elif (opcode != '000000'):
  self.update_display("Invalid OPCODE")
  return
```

Copy to Clipboard

Copy and paste the following lines at the bottom of your program, outside your class, and after your initialization. Change the your_calc_name to what you named your calculator.

```
your_calc_name.binary_reader("1234567812345678123456781234567")
your_calc_name.binary_reader("12345678123456781234567812345678")
```

Copy to Clipboard

You should get some error messages, an invalid length message, and an invalid opcode message.

35.

The next part of the instruction your binary_reader() method needs to read is the function.

Create a local variable, result, and set it equal to 0.

After the result, add an if-else statement that reassigns result to the value of each of the four function_code functions. Add an else clause that catches all incorrect function_codes and sends an error message to the user before returning.

This instruction will be vague on purpose to challenge you to create your own logic. Don't forget to pass source_one and source_two into your functional methods for each function code.

Hint

```
result = 0

if (function_code == '100000'):
  result = self.add(source_one, source_two)
```

```
elif (function_code == '100010'):
  result = self.subtract(source_one, source_two)
elif (function_code == '011000'):
  result = self.multiply(source_one, source_two)
elif (function_code == '011010'):
  result = self.divide(source_one, source_two)
else:
  self.update_display("Invalid Function")
  return
```

Copy to Clipboard

After you've finished your code, paste the following at the bottom of your program and run it, you should see your error message pop up.

your_calc_name.binary_reader("00000078123456781234567812345678")

36.

The final step to complete your calculator is to store the result to the history register and update the user display with the message that gives them the result.

Hint

```
self.store_to_history_register(result)
self.update_display(f"The result is: {result}")
```

Copy to Clipboard

Using Your Calculator

37.

First things first, delete the code that you inserted to test your errors. This is the code that was placed at the bottom and outside of your class. Don't delete the line that instantiated your calculator.

Then, use the binary_reader() method to intake some binary data. Look at your opcodes and function codes from the ISA and do some calculations. Remember you need to store values in the register before you can operate on them.

Check the hints for some examples.

Hint

```
# Adds 5 and 10 to number registers
your_calc_name.binary_reader("00000100000000000000000101000000")
your_calc_name.binary_reader("00000100000000000000001010000000")

# Adds/Subtracts/Multiplies/Divides 5 and 10 from registers
your_calc_name.binary_reader("00000000001000100000000000100000")
your_calc_name.binary_reader("00000000001000100000000000100010")
your_calc_name.binary_reader("00000000001000100000000000011000")
your_calc_name.binary_reader("00000000001000100000000000011010")

# Gets the last three calculations
```

your_calc_name.binary_reader("10000100000000000000000000000000")
your_calc_name.binary_reader("10000100000000000000000000000000")
your_calc_name.binary_reader("10000100000000000000000000000000")

calculator.py

```
#-------------------------------------------
# USCC Headquarter's Instruction Set Architecture
#  System Design:
#   - Four function calculator
#   - Can only operate on numbers stored in registers
#   - Processor receives binary data as 32-bit strings
#   - Returns results to the terminal
#   - Can operate on 10-bit numbers (0 thru 1023)
#   - Results can be negative (5 - 10 = -5)
#  Instruction format:
#   - 32 bit's in length
#   - Binary data will come to the CPU as a string
#   - Registers (32 total on CPU, 0-indexed)
#     - 0 thru 21:  Available for number storage
#       - 0: Constant 0
#     - 22 thru 31: Available for history storage
# +=======+=======+=======+=======+=======+=======+=======+=======+
# | 0: 0  | 1:    | 2:    | 3:    | 4:    | 5:    | 6:    | 7:    |
# +-------+-------+-------+-------+-------+-------+-------+-------+
# | 8:    | 9:    |10:    |11:    |12:    |13:    |14:    |15:    |
# +-------+-------+-------+-------+-------+-------+-------+-------+
# |16:    |17:    |18:    |19:    |20:    |21:    |22: H0 |23: H1 |
# +-------+-------+-------+-------+-------+-------+-------+-------+
# |24: H2 |25: H3 |26: H4 |27: H5 |28: H6 |29: H7 |30: H8 |31: H9 |
# +=======+=======+=======+=======+=======+=======+=======+=======+
#   - Bits 0-5 are OPCODEs
#     - use variable 'opcode' in program
#   - Bits 6-10 & 11-15 are source register locations
```

```
#    - use variables 'source_one' and 'source_two' in program
#   - Bits 16-25 are reserved for adding a new value to the registers
#    - use variable 'store' in program
#   - Bits 26-31 are functions
#    - use variable 'function_code' in program
# +--------+----------+------------------------------------+
# | OPCODE | FUNCTION | Definition                         |
# | 000000 |  100000  | Add two numbers from registers      |
# | 000000 |  100010  | Subtract two numbers from registers |
# | 000000 |  011000  | Multiply two numbers from registers |
# | 000000 |  011010  | Divide two numbers from registers   |
# | 000001 |  000000  | Store value to next register        |
# | 100001 |  000000  | Return previous calculation         |
# +--------+----------+------------------------------------+

# Your code below here:
class UltraSuperCalculator:
  def __init__(self, name) -> None:
    self.name = name
    self.number_registers = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
    self.history_registers = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
    self.numbers_index = 1
    self.history_index = 0
    self.temp_history_index = 0
    self.user_display = ''
    self.update_display(f"Welcome to {self.name}'s Calculator!")

  def update_display(self, to_update):
    self.user_display = to_update
    print(self.user_display)

  def store_value_to_register(self, value_to_store):
```

```
  if (self.numbers_index > 21):
    self.numbers_index = 1

  self.number_registers[self.numbers_index] = int(value_to_store, 2)

  print(f"Value: {int(value_to_store, 2)} stored in Register: {self.numbers_index}.")
  self.numbers_index += 1

def load_value_from_register(self, register_address):
  index = int(register_address, 2)
  int_value = int(self.number_registers[index])
  return int_value

def store_to_history_register(self, result_to_store):
  if (self.history_index > 9):
    self.history_index = 0

  self.history_registers[self.history_index] = bin(result_to_store)

  self.history_index += 1
  self.temp_history_index = self.history_index

def add(self, address_num1, address_num2):
  num1 = self.load_value_from_register(address_num1)
  num2 = self.load_value_from_register(address_num2)
  calculated_value = num1 + num2
  return calculated_value

def multiply(self, address_num1, address_num2):
  num1 = self.load_value_from_register(address_num1)
  num2 = self.load_value_from_register(address_num2)
  calculated_value = num1 * num2
```

```
    return calculated_value

  def subtract(self, address_num1, address_num2):
    num1 = self.load_value_from_register(address_num1)
    num2 = self.load_value_from_register(address_num2)
    calculated_value = num1 - num2
    return calculated_value

  def divide(self, address_num1, address_num2):
    num1 = self.load_value_from_register(address_num1)
    num2 = self.load_value_from_register(address_num2)
    calculated_value = 0

    if (num2 != 0):
      calculated_value = int(num1 / num2)

    else:
      print(f"Division by 0 error: {num1}/{num2}.")

    return calculated_value

  def get_last_calculation(self):
    self.temp_history_index -= 1
    last_value = f"The last calculated value was: {int(self.history_registers[self.temp_history_index])}"
    self.update_display(last_value)

  def binary_reader(self, instruction):
    if (len(instruction) != 32):
      self.update_display("Invalid Instruction Length")
      return

    opcode = instruction[0 : 6]
```

```
    source_one = instruction[6 : 11]
    source_two = instruction[11 : 16]
    store = instruction[16 : 26]
    function_code = instruction[26:]

    if (opcode == '000001'):
      self.store_value_to_register(store)
      return
    elif (opcode == '100001'):
      self.get_last_calculation()
      return
    elif (opcode != '000000'):
      self.update_display("Invalid OPCODE")
      return

    result = 0

    if (function_code == '100000'):
      result = self.add(source_one, source_two)
    elif (function_code == '100010'):
      result = self.subtract(source_one, source_two)
    elif (function_code == '011000'):
      result = self.multiply(source_one, source_two)
    elif (function_code == '011010'):
      result = self.divide(source_one, source_two)
    else:
      self.update_display("Invalid Function")
    return


# Calling the class
your_calculator = UltraSuperCalculator('Andres')
```

'''

your_calculator.binary_reader("1234567812345678123456781234567")

your_calculator.binary_reader("12345678123456781234567812345678")

'''

'''

your_calculator.binary_reader("00000078123456781234567812345678")

'''

\# Adds 5 and 10 to number registers

your_calculator.binary_reader("00000100000000000000000101000000")

your_calculator.binary_reader("00000100000000000000001010000000")

\# Adds/Subtracts/Multiplies/Divides 5 and 10 from registers

your_calculator.binary_reader("00000000001000100000000000100000")

your_calculator.binary_reader("00000000001000100000000000100010")

your_calculator.binary_reader("00000000001000100000000000011000")

your_calculator.binary_reader("00000000001000100000000000011010")

\# Gets the last three calculations

your_calculator.binary_reader("10000100000000000000000000000000")

your_calculator.binary_reader("10000100000000000000000000000000")

your_calculator.binary_reader("10000100000000000000000000000000")

>> Output

Output-only Terminal

Output:

Welcome to Andres's Calculator!

Value: 5 stored in Register: 1.

Value: 10 stored in Register: 2.

The last calculated value was: 0

The last calculated value was: 0

The last calculated value was: 0