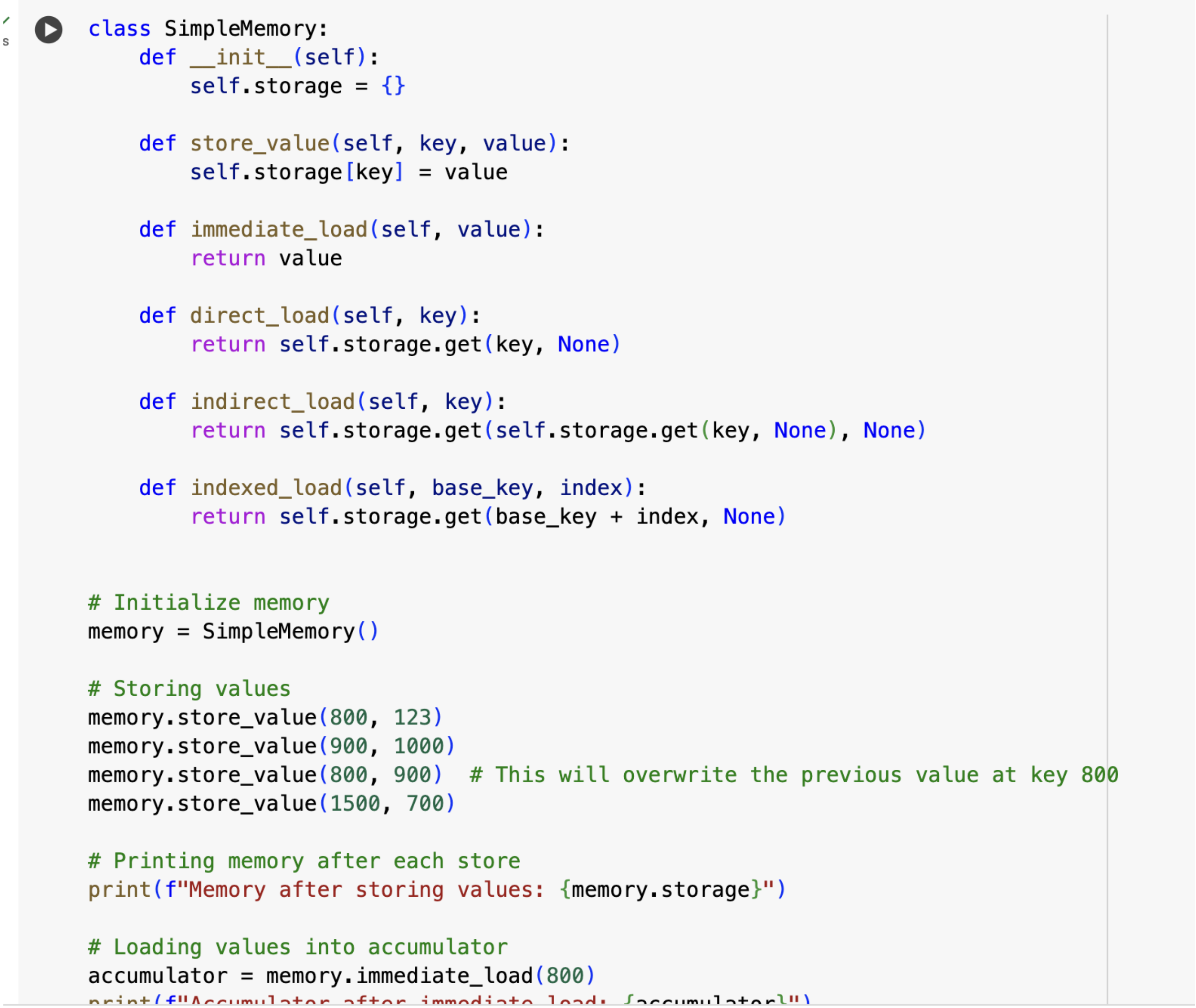
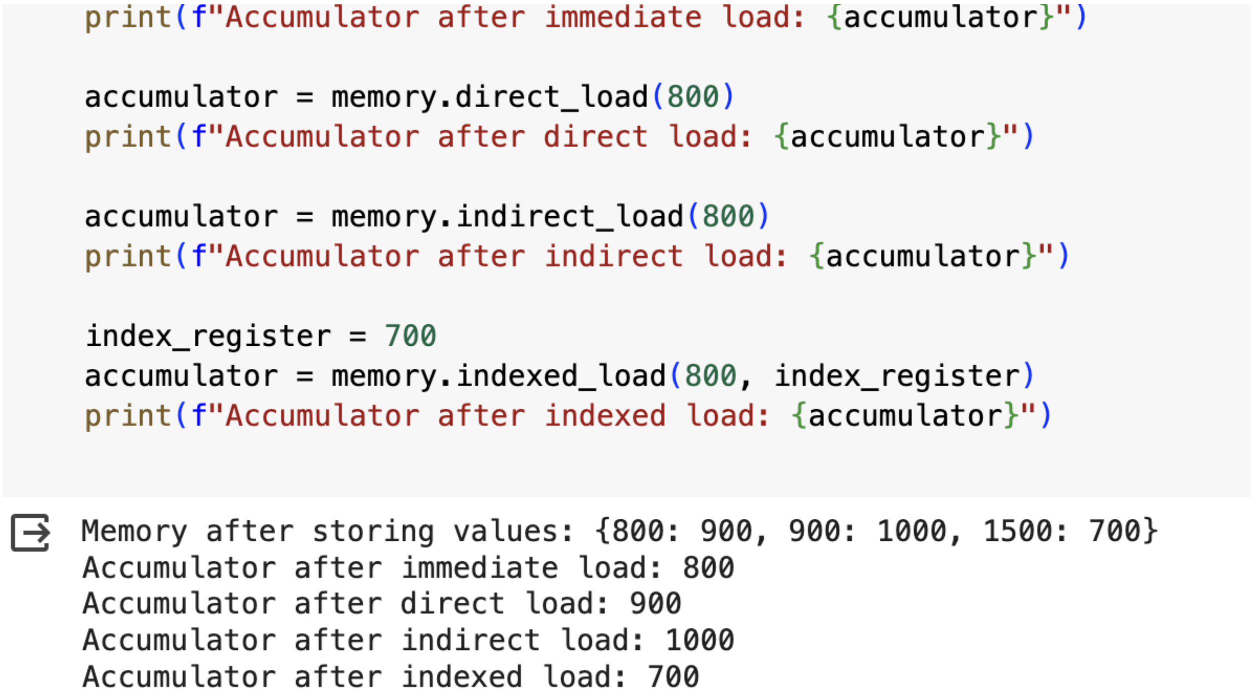
REWATI RAMAN KARKI(19856)
COMPUTER ORGANIZATION
1.



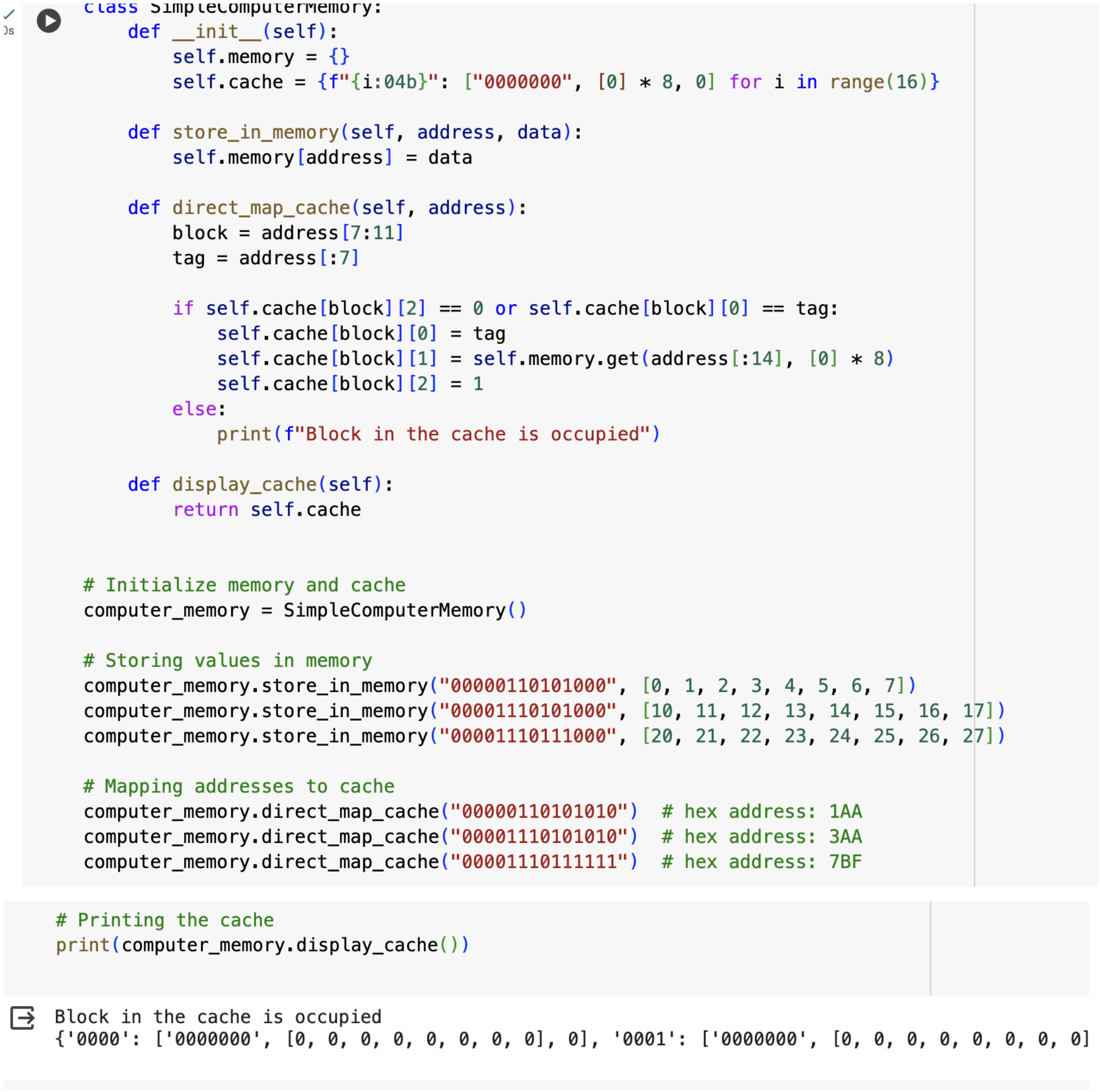
```python
class SimpleMemory:
    def __init__(self):
        self.storage = {}

    def store_value(self, key, value):
        self.storage[key] = value

    def immediate_load(self, value):
        return value

    def direct_load(self, key):
        return self.storage.get(key, None)

    def indirect_load(self, key):
        return self.storage.get(self.storage.get(key, None), None)

    def indexed_load(self, base_key, index):
        return self.storage.get(base_key + index, None)


# Initialize memory
memory = SimpleMemory()

# Storing values
memory.store_value(800, 123)
memory.store_value(900, 1000)
memory.store_value(800, 900)  # This will overwrite the previous value at key 800
memory.store_value(1500, 700)

# Printing memory after each store
print(f"Memory after storing values: {memory.storage}")

# Loading values into accumulator
accumulator = memory.immediate_load(800)
print(f"Accumulator after immediate load: {accumulator}")
```



```
print(f"Accumulator after immediate load: {accumulator}")

accumulator = memory.direct_load(800)
print(f"Accumulator after direct load: {accumulator}")

accumulator = memory.indirect_load(800)
print(f"Accumulator after indirect load: {accumulator}")

index_register = 700
accumulator = memory.indexed_load(800, index_register)
print(f"Accumulator after indexed load: {accumulator}")
```

```
Memory after storing values: {800: 900, 900: 1000, 1500: 700}
Accumulator after immediate load: 800
Accumulator after direct load: 900
Accumulator after indirect load: 1000
Accumulator after indexed load: 700
```

2.

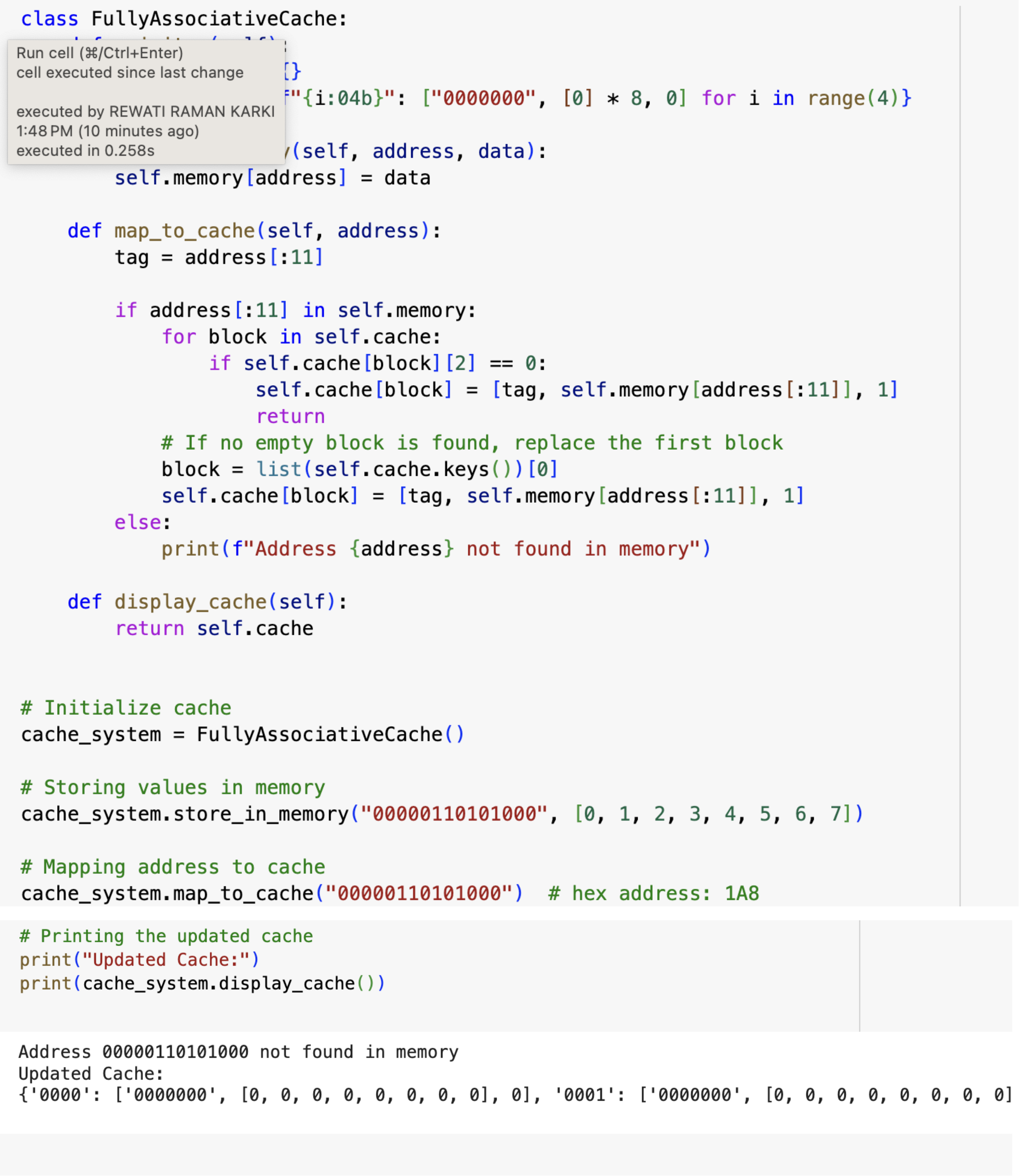


```
class SimpleComputerMemory:
    def __init__(self):
        self.memory = {}
        self.cache = {f"{i:04b}": ["0000000", [0] * 8, 0] for i in range(16)}

    def store_in_memory(self, address, data):
        self.memory[address] = data

    def direct_map_cache(self, address):
        block = address[7:11]
        tag = address[:7]

        if self.cache[block][2] == 0 or self.cache[block][0] == tag:
            self.cache[block][0] = tag
            self.cache[block][1] = self.memory.get(address[:14], [0] * 8)
            self.cache[block][2] = 1
        else:
            print(f"Block in the cache is occupied")

    def display_cache(self):
        return self.cache


# Initialize memory and cache
computer_memory = SimpleComputerMemory()

# Storing values in memory
computer_memory.store_in_memory("00000110101000", [0, 1, 2, 3, 4, 5, 6, 7])
computer_memory.store_in_memory("00001110101000", [10, 11, 12, 13, 14, 15, 16, 17])
computer_memory.store_in_memory("00001110111000", [20, 21, 22, 23, 24, 25, 26, 27])

# Mapping addresses to cache
computer_memory.direct_map_cache("00000110101010")  # hex address: 1AA
computer_memory.direct_map_cache("00001110101010")  # hex address: 3AA
computer_memory.direct_map_cache("00001110111111")  # hex address: 7BF
```

```
# Printing the cache
print(computer_memory.display_cache())
```

```
Block in the cache is occupied
{'0000': ['0000000', [0, 0, 0, 0, 0, 0, 0, 0], 0], '0001': ['0000000', [0, 0, 0, 0, 0, 0, 0, 0]
```

3.



Run cell (⌘/Ctrl+Enter)
cell executed since last change

executed by REWATI RAMAN KARKI
1:48 PM (10 minutes ago)
executed in 0.258s

```
class FullyAssociativeCache:
                                        :
                                        {}
                                        f"{i:04b}": ["0000000", [0] * 8, 0] for i in range(4)}

                                        (self, address, data):
        self.memory[address] = data

    def map_to_cache(self, address):
        tag = address[:11]

        if address[:11] in self.memory:
            for block in self.cache:
                if self.cache[block][2] == 0:
                    self.cache[block] = [tag, self.memory[address[:11]], 1]
                    return
            # If no empty block is found, replace the first block
            block = list(self.cache.keys())[0]
            self.cache[block] = [tag, self.memory[address[:11]], 1]
        else:
            print(f"Address {address} not found in memory")

    def display_cache(self):
        return self.cache


# Initialize cache
cache_system = FullyAssociativeCache()

# Storing values in memory
cache_system.store_in_memory("00000110101000", [0, 1, 2, 3, 4, 5, 6, 7])

# Mapping address to cache
cache_system.map_to_cache("00000110101000")  # hex address: 1A8
```

```
# Printing the updated cache
print("Updated Cache:")
print(cache_system.display_cache())
```

```
Address 00000110101000 not found in memory
Updated Cache:
{'0000': ['0000000', [0, 0, 0, 0, 0, 0, 0, 0], 0], '0001': ['0000000', [0, 0, 0, 0, 0, 0, 0, 0]
```