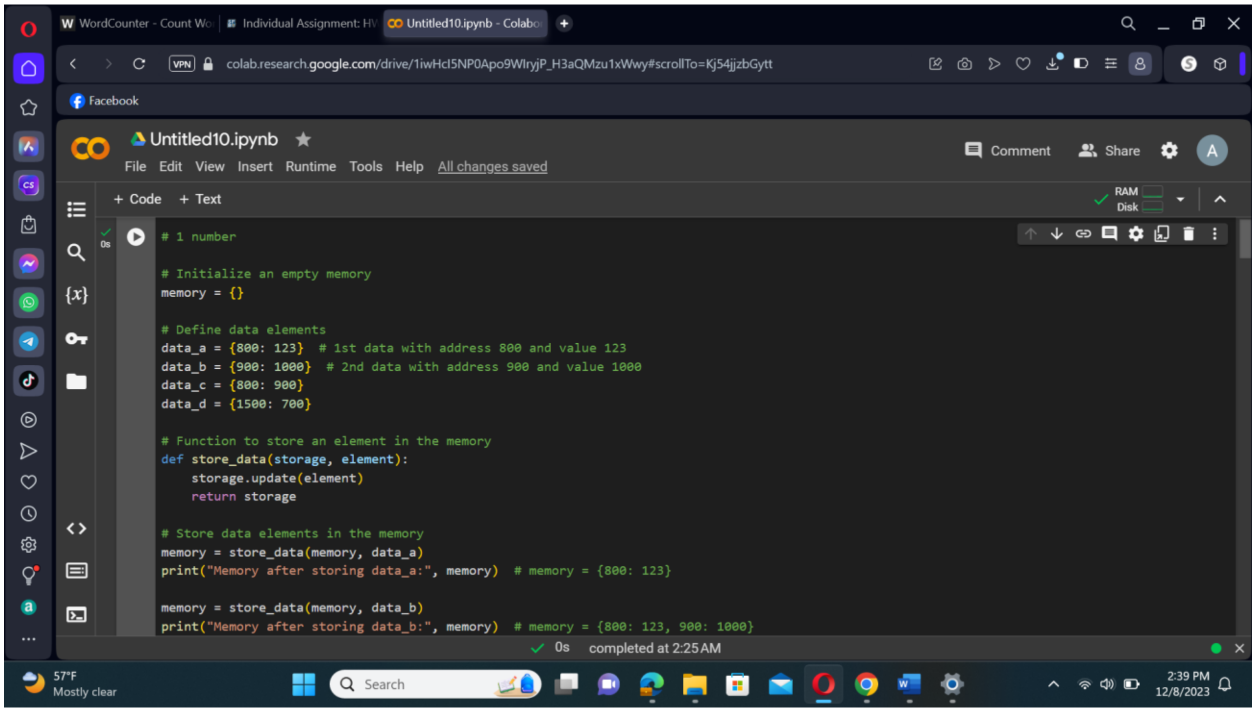
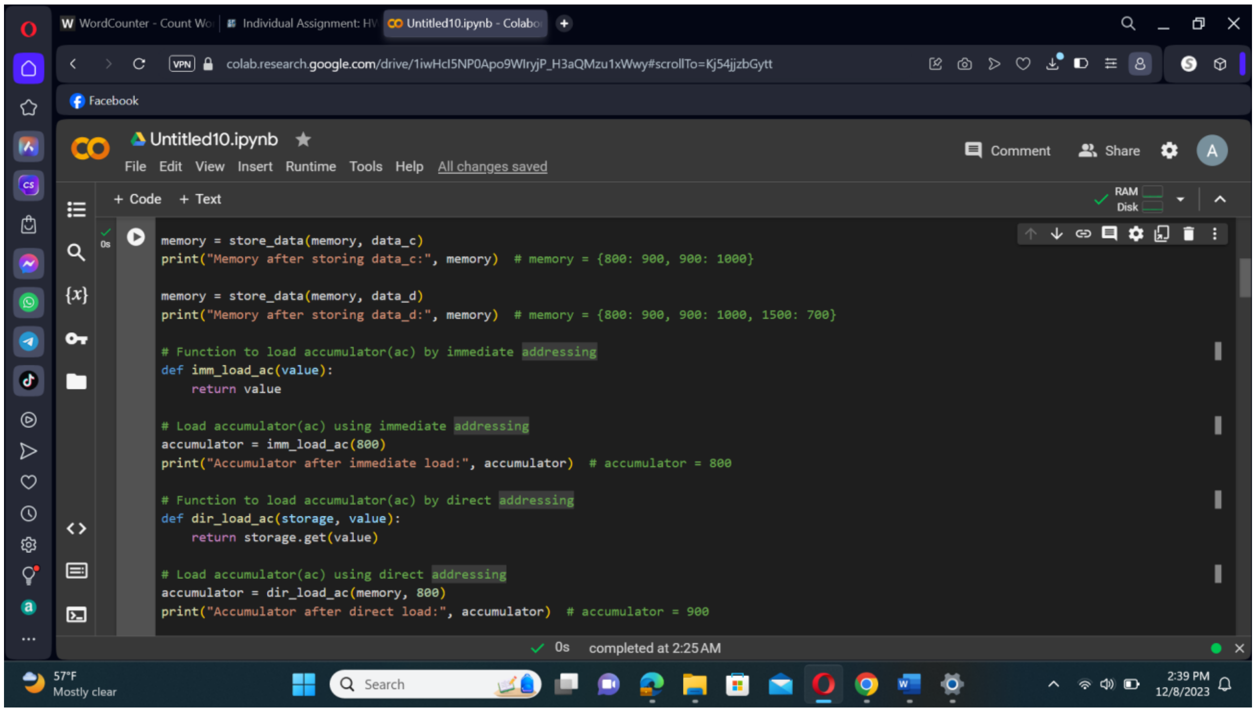
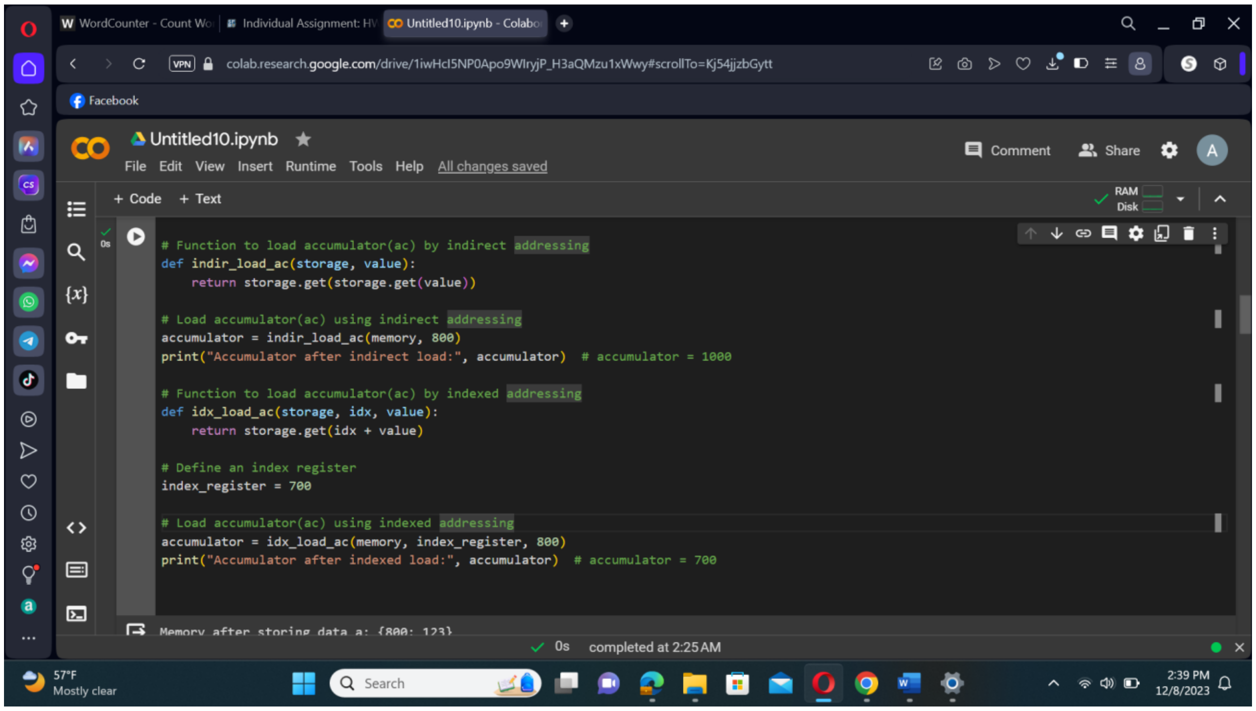
1.In MARIE architecture ISA, there are several different types of addressing modes, such as immediate addressing, direct addressing, indirect addressing and indexed addressing. Please complete the following functions in Python to simulate how the different addressing modes work.



```python
# 1 number

# Initialize an empty memory
memory = {}

# Define data elements
data_a = {800: 123}  # 1st data with address 800 and value 123
data_b = {900: 1000}  # 2nd data with address 900 and value 1000
data_c = {800: 900}
data_d = {1500: 700}

# Function to store an element in the memory
def store_data(storage, element):
    storage.update(element)
    return storage

# Store data elements in the memory
memory = store_data(memory, data_a)
print("Memory after storing data_a:", memory)  # memory = {800: 123}

memory = store_data(memory, data_b)
print("Memory after storing data_b:", memory)  # memory = {800: 123, 900: 1000}
```

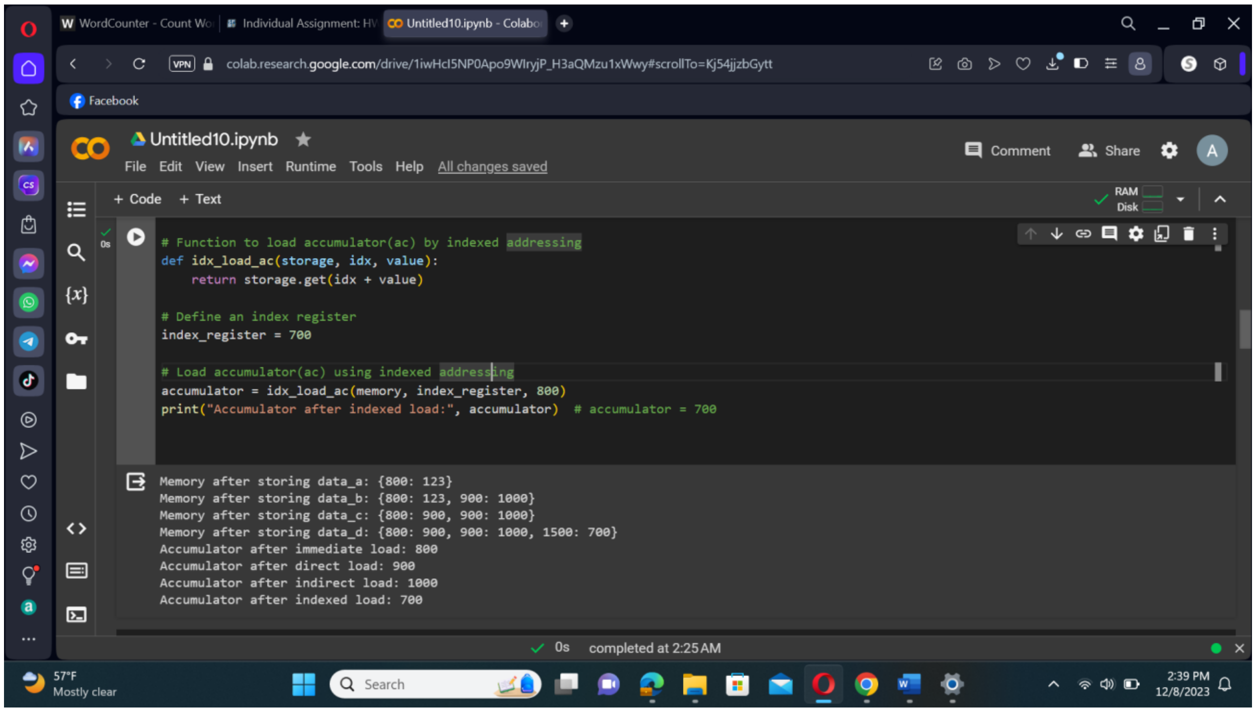


```python
memory = store_data(memory, data_c)
print("Memory after storing data_c:", memory)  # memory = {800: 900, 900: 1000}

memory = store_data(memory, data_d)
print("Memory after storing data_d:", memory)  # memory = {800: 900, 900: 1000, 1500: 700}

# Function to load accumulator(ac) by immediate addressing
def imm_load_ac(value):
    return value

# Load accumulator(ac) using immediate addressing
accumulator = imm_load_ac(800)
print("Accumulator after immediate load:", accumulator)  # accumulator = 800

# Function to load accumulator(ac) by direct addressing
def dir_load_ac(storage, value):
    return storage.get(value)

# Load accumulator(ac) using direct addressing
accumulator = dir_load_ac(memory, 800)
print("Accumulator after direct load:", accumulator)  # accumulator = 900
```



```
# Function to load accumulator(ac) by indirect addressing
def indir_load_ac(storage, value):
    return storage.get(storage.get(value))

# Load accumulator(ac) using indirect addressing
accumulator = indir_load_ac(memory, 800)
print("Accumulator after indirect load:", accumulator)  # accumulator = 1000

# Function to load accumulator(ac) by indexed addressing
def idx_load_ac(storage, idx, value):
    return storage.get(idx + value)

# Define an index register
index_register = 700

# Load accumulator(ac) using indexed addressing
accumulator = idx_load_ac(memory, index_register, 800)
print("Accumulator after indexed load:", accumulator)  # accumulator = 700
```

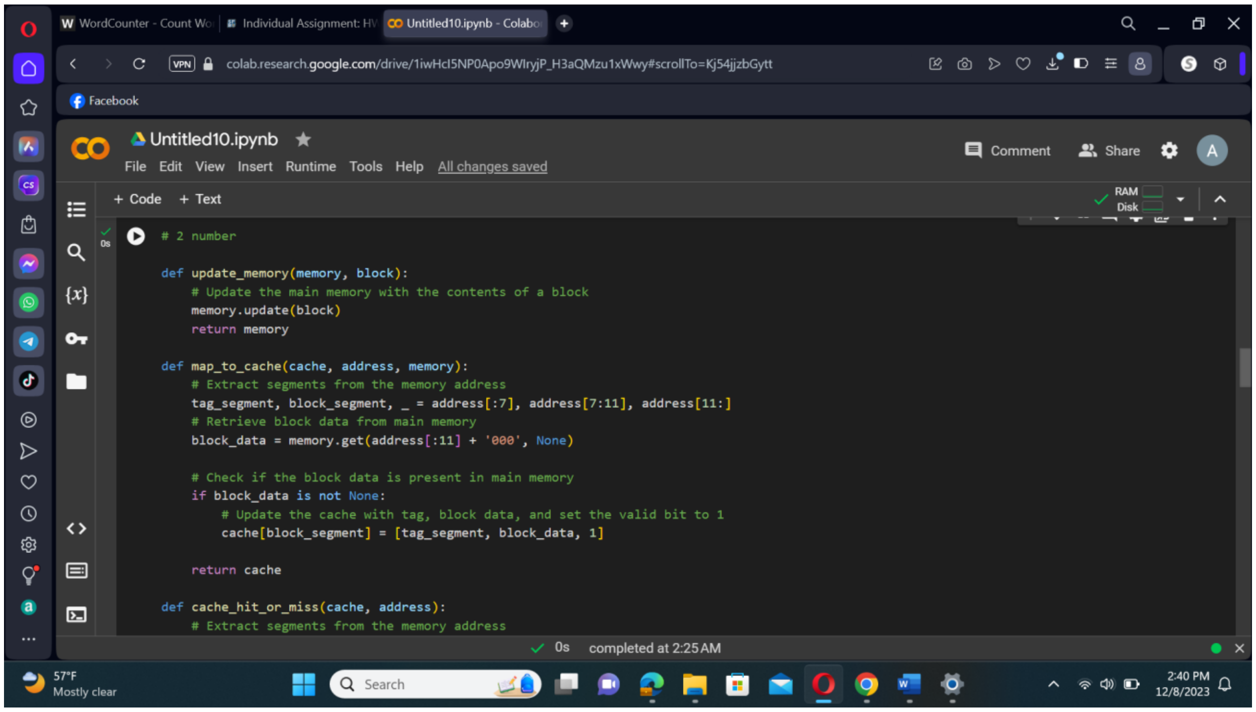
```
Memory after storing data_a: {800: 123}
```



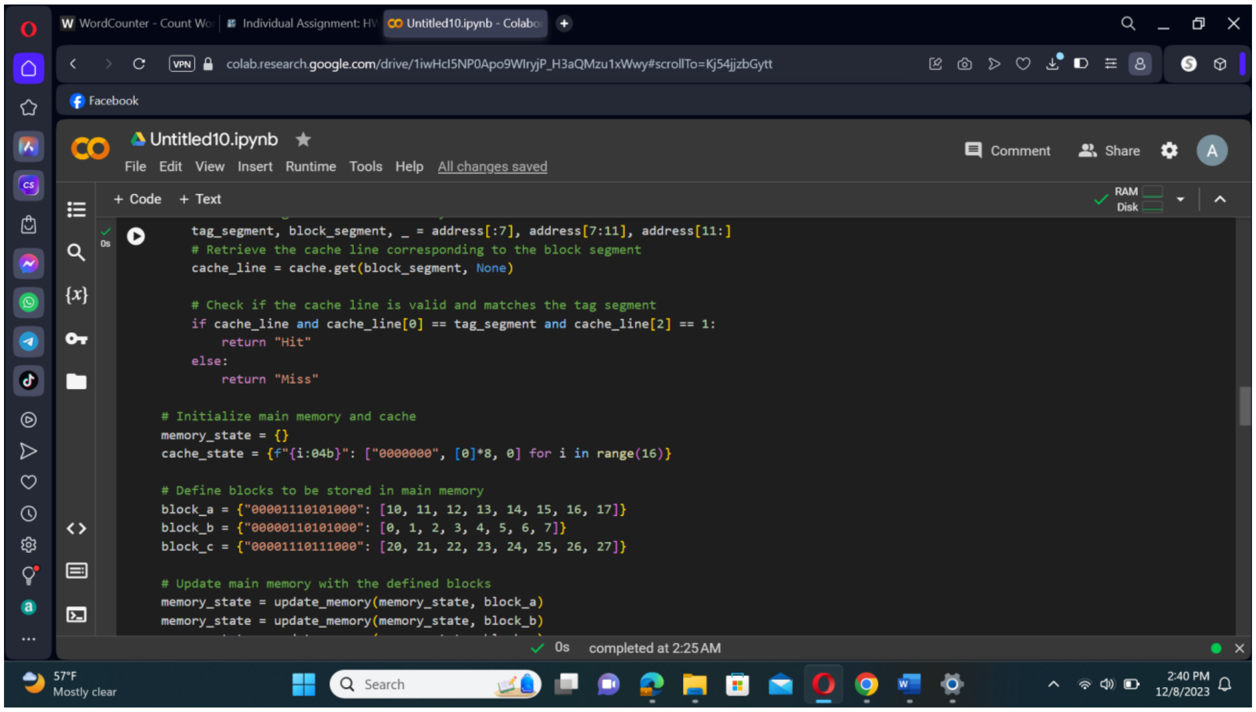
```
# Function to load accumulator(ac) by indexed addressing
def idx_load_ac(storage, idx, value):
    return storage.get(idx + value)

# Define an index register
index_register = 700

# Load accumulator(ac) using indexed addressing
accumulator = idx_load_ac(memory, index_register, 800)
print("Accumulator after indexed load:", accumulator)  # accumulator = 700
```

```
Memory after storing data_a: {800: 123}
Memory after storing data_b: {800: 123, 900: 1000}
Memory after storing data_c: {800: 900, 900: 1000}
Memory after storing data_d: {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
```

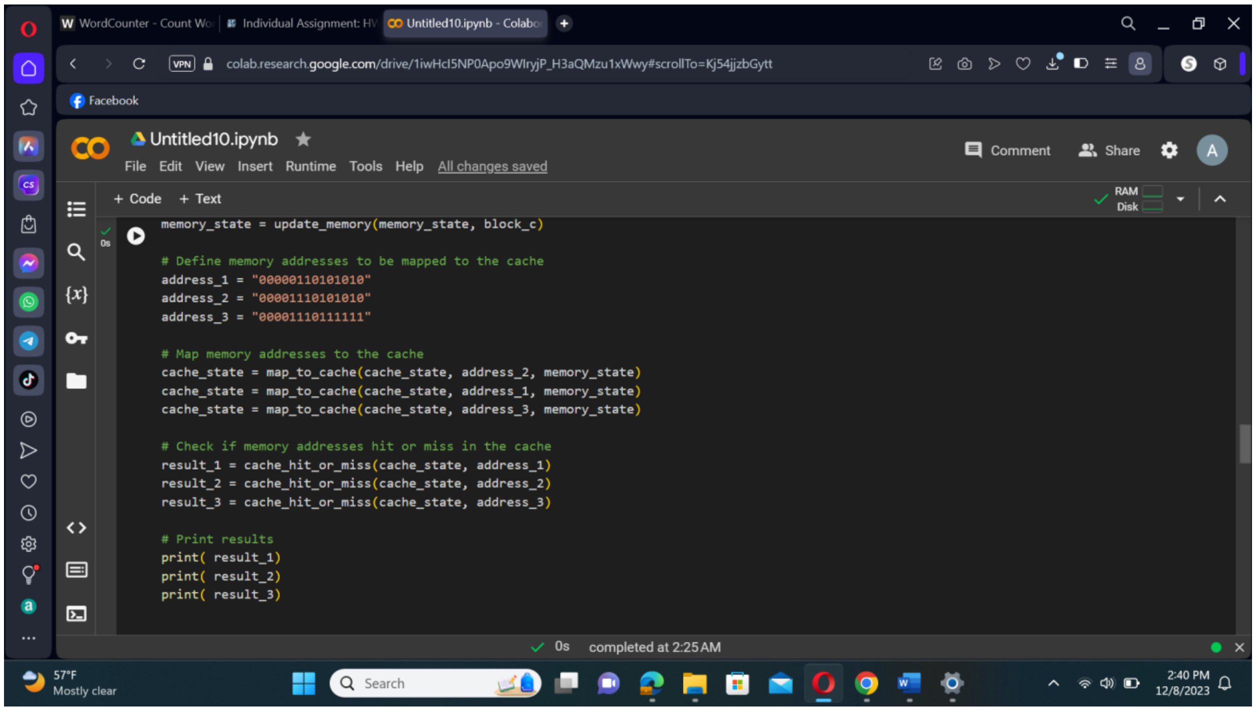
2.If the main memory consists of 2^14 words, 2^11 blocks will be created in it, each block holds 8 words, and cache has 16 = 2^4 blocks, any memory address can be separate as following segments for Tag, Block and Word. In direct mapped cache, the whole block can be directly mapped to the cache line based on the values of 4-bit in Block segment. Please complete the following functions in Python program.



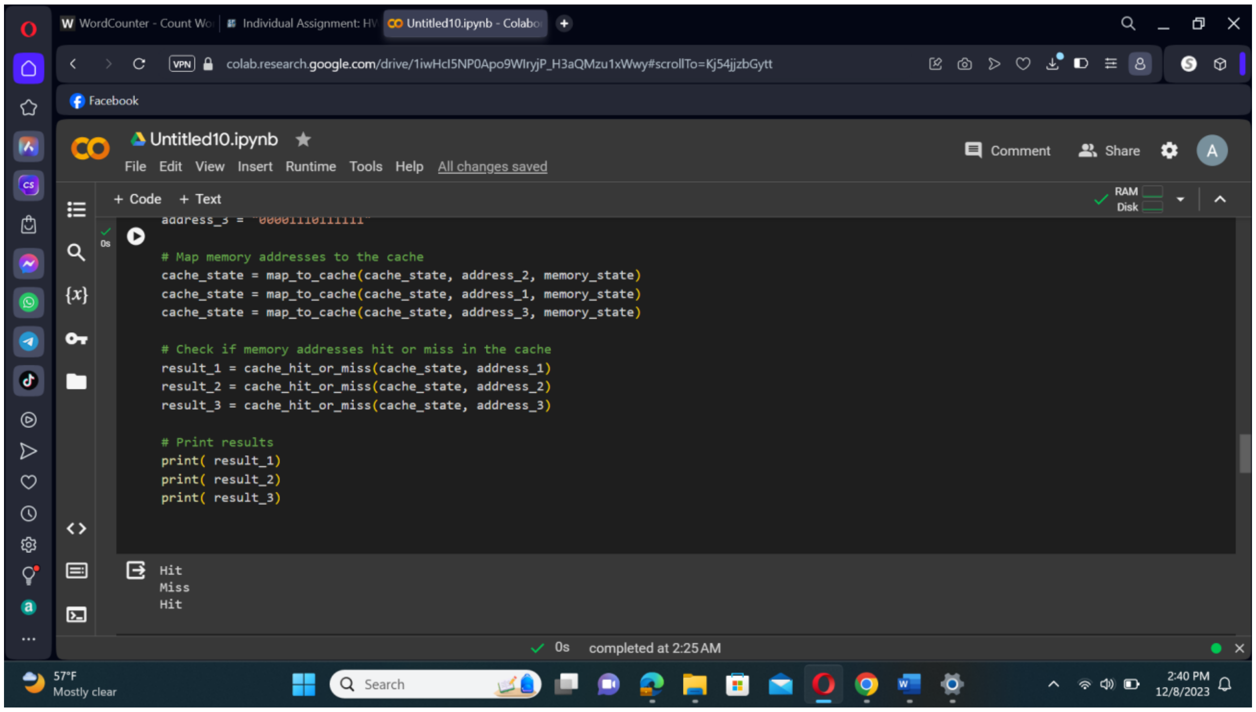
```python
# 2 number

def update_memory(memory, block):
    # Update the main memory with the contents of a block
    memory.update(block)
    return memory

def map_to_cache(cache, address, memory):
    # Extract segments from the memory address
    tag_segment, block_segment, _ = address[:7], address[7:11], address[11:]
    # Retrieve block data from main memory
    block_data = memory.get(address[:11] + '000', None)

    # Check if the block data is present in main memory
    if block_data is not None:
        # Update the cache with tag, block data, and set the valid bit to 1
        cache[block_segment] = [tag_segment, block_data, 1]

    return cache

def cache_hit_or_miss(cache, address):
    # Extract segments from the memory address
```



```python
    tag_segment, block_segment, _ = address[:7], address[7:11], address[11:]
    # Retrieve the cache line corresponding to the block segment
    cache_line = cache.get(block_segment, None)

    # Check if the cache line is valid and matches the tag segment
    if cache_line and cache_line[0] == tag_segment and cache_line[2] == 1:
        return "Hit"
    else:
        return "Miss"

# Initialize main memory and cache
memory_state = {}
cache_state = {f"{i:04b}": ["0000000", [0]*8, 0] for i in range(16)}

# Define blocks to be stored in main memory
block_a = {"00001110101000": [10, 11, 12, 13, 14, 15, 16, 17]}
block_b = {"00000110101000": [0, 1, 2, 3, 4, 5, 6, 7]}
block_c = {"00001110111000": [20, 21, 22, 23, 24, 25, 26, 27]}

# Update main memory with the defined blocks
memory_state = update_memory(memory_state, block_a)
memory_state = update_memory(memory_state, block_b)
```



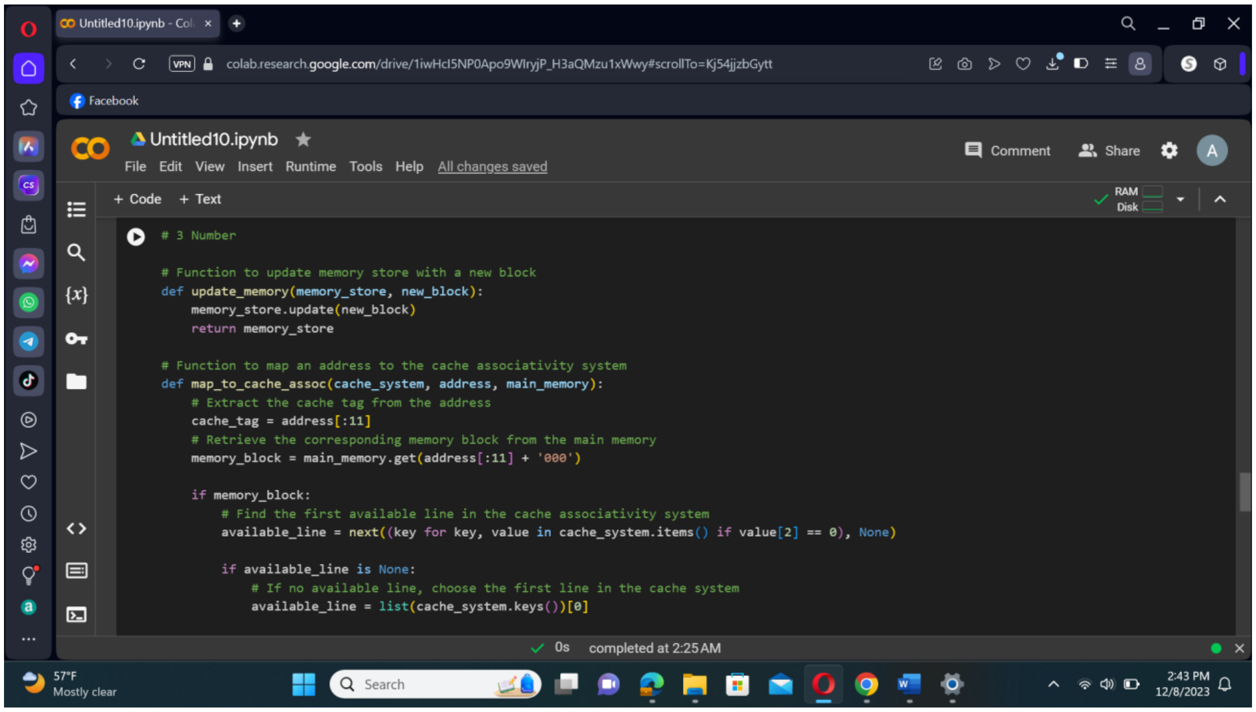
```
memory_state = update_memory(memory_state, block_c)

# Define memory addresses to be mapped to the cache
address_1 = "00000110101010"
address_2 = "00001110101010"
address_3 = "00001110111111"

# Map memory addresses to the cache
cache_state = map_to_cache(cache_state, address_2, memory_state)
cache_state = map_to_cache(cache_state, address_1, memory_state)
cache_state = map_to_cache(cache_state, address_3, memory_state)

# Check if memory addresses hit or miss in the cache
result_1 = cache_hit_or_miss(cache_state, address_1)
result_2 = cache_hit_or_miss(cache_state, address_2)
result_3 = cache_hit_or_miss(cache_state, address_3)

# Print results
print( result_1)
print( result_2)
print( result_3)
```



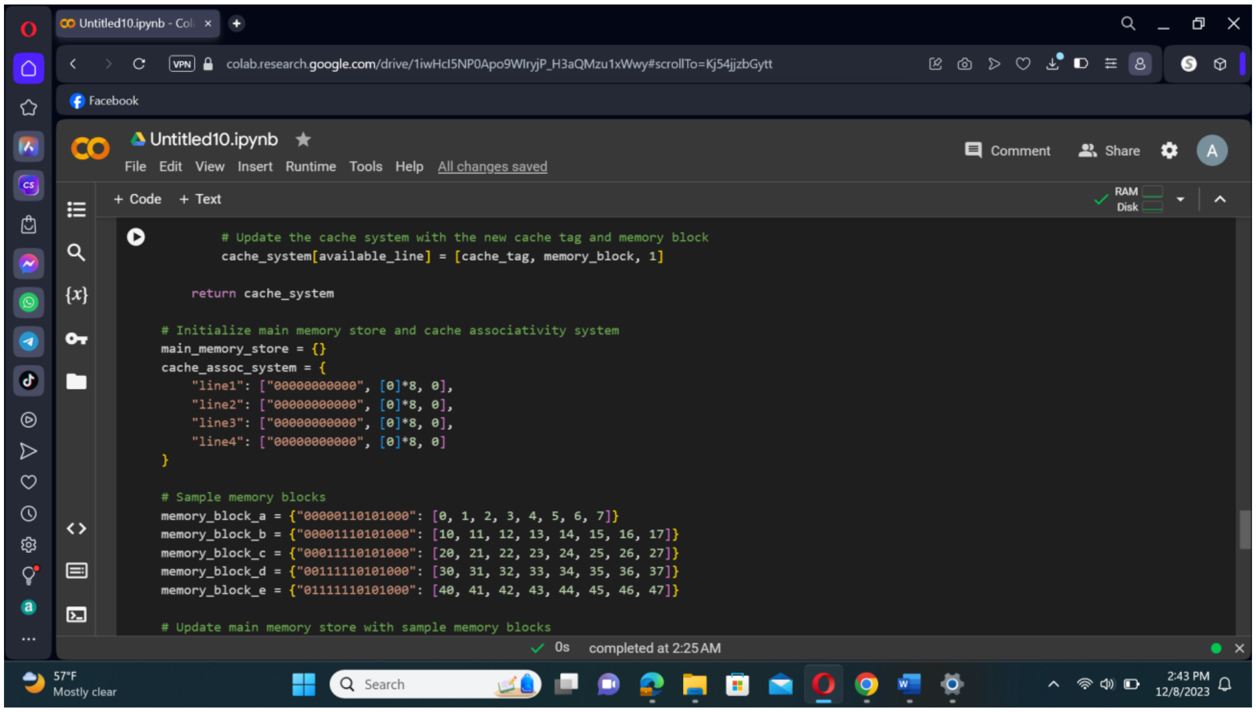
```
address_3 = "00001110111111"

# Map memory addresses to the cache
cache_state = map_to_cache(cache_state, address_2, memory_state)
cache_state = map_to_cache(cache_state, address_1, memory_state)
cache_state = map_to_cache(cache_state, address_3, memory_state)

# Check if memory addresses hit or miss in the cache
result_1 = cache_hit_or_miss(cache_state, address_1)
result_2 = cache_hit_or_miss(cache_state, address_2)
result_3 = cache_hit_or_miss(cache_state, address_3)

# Print results
print( result_1)
print( result_2)
print( result_3)
```

```
Hit
Miss
Hit
```

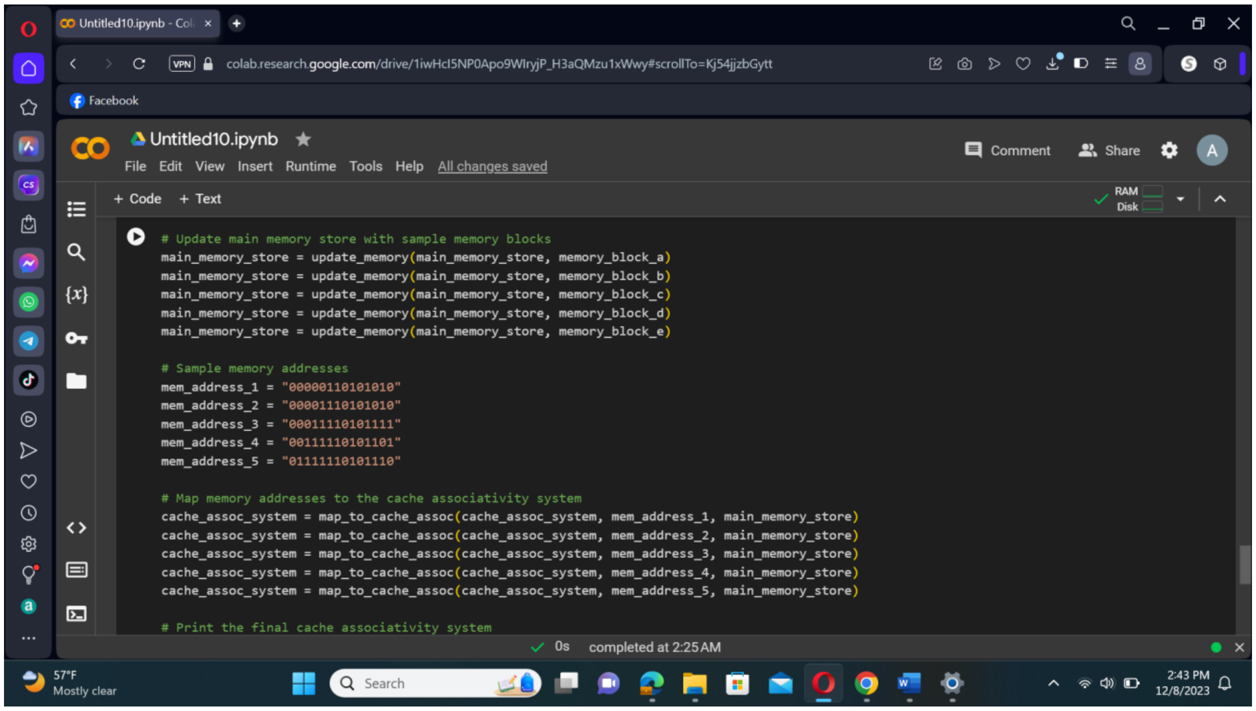
3. To avoid thrashing issue in direct mapped cache as above, the technique of fully associative cache will be taken. The 14-bit memory address can be separated as follows for Tag and word segments. Assuming that there are only 4 cache lines in the cache, the block in the main memory can be mapped to any cache line if the valid bit is 1 showing it is available.



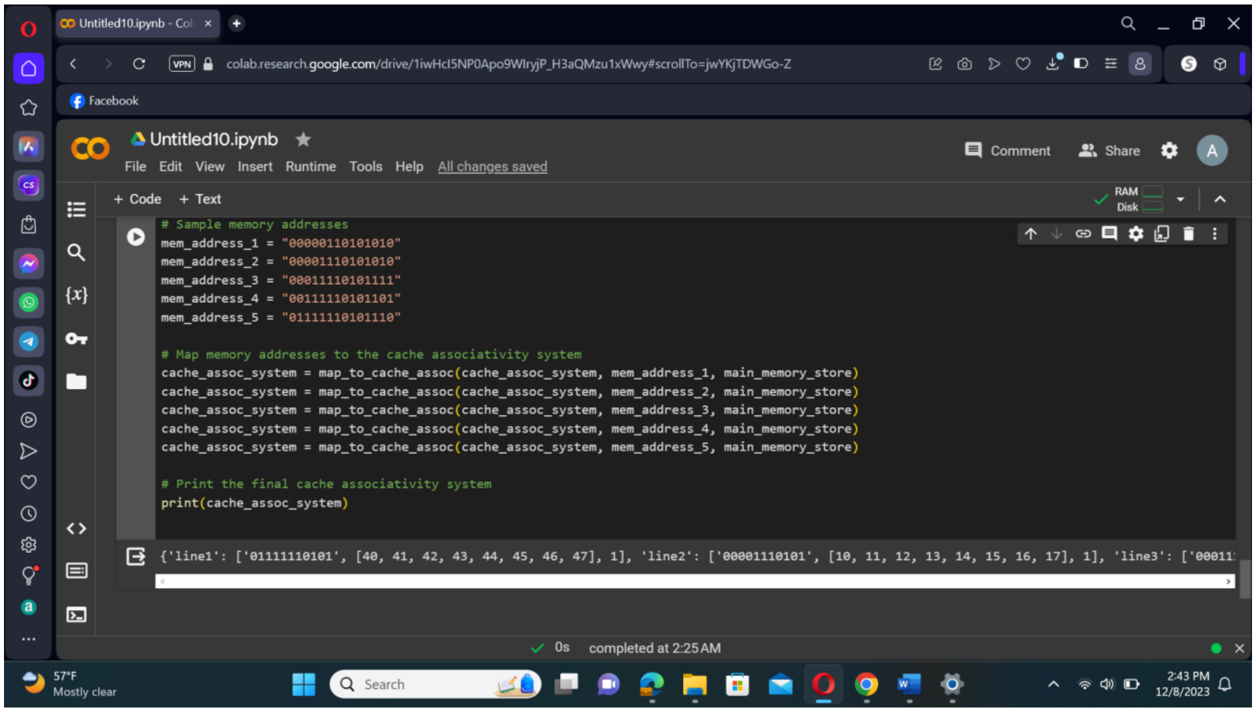
```
# 3 Number

# Function to update memory store with a new block
def update_memory(memory_store, new_block):
    memory_store.update(new_block)
    return memory_store

# Function to map an address to the cache associativity system
def map_to_cache_assoc(cache_system, address, main_memory):
    # Extract the cache tag from the address
    cache_tag = address[:11]
    # Retrieve the corresponding memory block from the main memory
    memory_block = main_memory.get(address[:11] + '000')

    if memory_block:
        # Find the first available line in the cache associativity system
        available_line = next((key for key, value in cache_system.items() if value[2] == 0), None)

        if available_line is None:
            # If no available line, choose the first line in the cache system
            available_line = list(cache_system.keys())[0]
```



```
        # Update the cache system with the new cache tag and memory block
        cache_system[available_line] = [cache_tag, memory_block, 1]

    return cache_system

# Initialize main memory store and cache associativity system
main_memory_store = {}
cache_assoc_system = {
    "line1": ["00000000000", [0]*8, 0],
    "line2": ["00000000000", [0]*8, 0],
    "line3": ["00000000000", [0]*8, 0],
    "line4": ["00000000000", [0]*8, 0]
}

# Sample memory blocks
memory_block_a = {"00000110101000": [0, 1, 2, 3, 4, 5, 6, 7]}
memory_block_b = {"00001110101000": [10, 11, 12, 13, 14, 15, 16, 17]}
memory_block_c = {"00011110101000": [20, 21, 22, 23, 24, 25, 26, 27]}
memory_block_d = {"00111110101000": [30, 31, 32, 33, 34, 35, 36, 37]}
memory_block_e = {"01111110101000": [40, 41, 42, 43, 44, 45, 46, 47]}

# Update main memory store with sample memory blocks
```

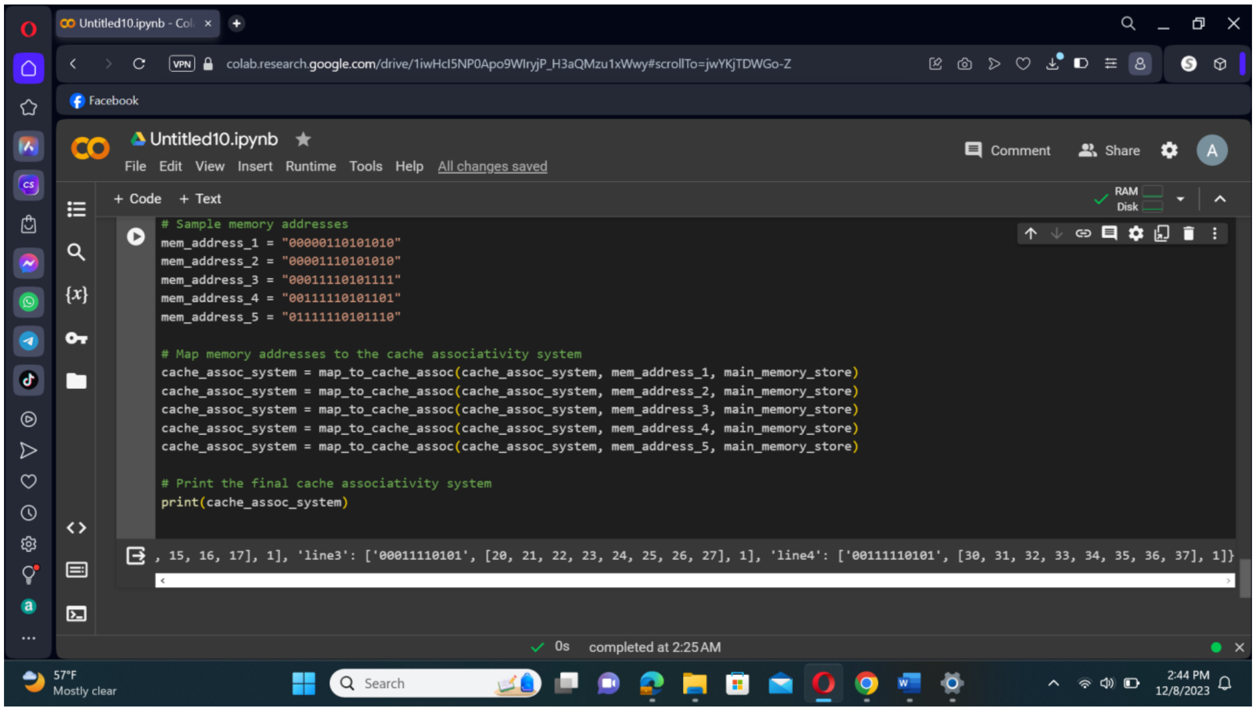


```
# Update main memory store with sample memory blocks
main_memory_store = update_memory(main_memory_store, memory_block_a)
main_memory_store = update_memory(main_memory_store, memory_block_b)
main_memory_store = update_memory(main_memory_store, memory_block_c)
main_memory_store = update_memory(main_memory_store, memory_block_d)
main_memory_store = update_memory(main_memory_store, memory_block_e)

# Sample memory addresses
mem_address_1 = "00000110101010"
mem_address_2 = "00001110101010"
mem_address_3 = "00011110101111"
mem_address_4 = "00111110101101"
mem_address_5 = "01111110101110"

# Map memory addresses to the cache associativity system
cache_assoc_system = map_to_cache_assoc(cache_assoc_system, mem_address_1, main_memory_store)
cache_assoc_system = map_to_cache_assoc(cache_assoc_system, mem_address_2, main_memory_store)
cache_assoc_system = map_to_cache_assoc(cache_assoc_system, mem_address_3, main_memory_store)
cache_assoc_system = map_to_cache_assoc(cache_assoc_system, mem_address_4, main_memory_store)
cache_assoc_system = map_to_cache_assoc(cache_assoc_system, mem_address_5, main_memory_store)

# Print the final cache associativity system
```



```
# Sample memory addresses
mem_address_1 = "00000110101010"
mem_address_2 = "00001110101010"
mem_address_3 = "00011110101111"
mem_address_4 = "00111110101101"
mem_address_5 = "01111110101110"

# Map memory addresses to the cache associativity system
cache_assoc_system = map_to_cache_assoc(cache_assoc_system, mem_address_1, main_memory_store)
cache_assoc_system = map_to_cache_assoc(cache_assoc_system, mem_address_2, main_memory_store)
cache_assoc_system = map_to_cache_assoc(cache_assoc_system, mem_address_3, main_memory_store)
cache_assoc_system = map_to_cache_assoc(cache_assoc_system, mem_address_4, main_memory_store)
cache_assoc_system = map_to_cache_assoc(cache_assoc_system, mem_address_5, main_memory_store)

# Print the final cache associativity system
print(cache_assoc_system)
```

```
{'line1': ['01111110101', [40, 41, 42, 43, 44, 45, 46, 47], 1], 'line2': ['00001110101', [10, 11, 12, 13, 14, 15, 16, 17], 1], 'line3': ['00011
```



```python
# Sample memory addresses
mem_address_1 = "00000110101010"
mem_address_2 = "00001110101010"
mem_address_3 = "00011110101111"
mem_address_4 = "00111110101101"
mem_address_5 = "01111110101110"

# Map memory addresses to the cache associativity system
cache_assoc_system = map_to_cache_assoc(cache_assoc_system, mem_address_1, main_memory_store)
cache_assoc_system = map_to_cache_assoc(cache_assoc_system, mem_address_2, main_memory_store)
cache_assoc_system = map_to_cache_assoc(cache_assoc_system, mem_address_3, main_memory_store)
cache_assoc_system = map_to_cache_assoc(cache_assoc_system, mem_address_4, main_memory_store)
cache_assoc_system = map_to_cache_assoc(cache_assoc_system, mem_address_5, main_memory_store)

# Print the final cache associativity system
print(cache_assoc_system)
```

```
, 15, 16, 17], 1], 'line3': ['00011110101', [20, 21, 22, 23, 24, 25, 26, 27], 1], 'line4': ['00111110101', [30, 31, 32, 33, 34, 35, 36, 37], 1]}
```