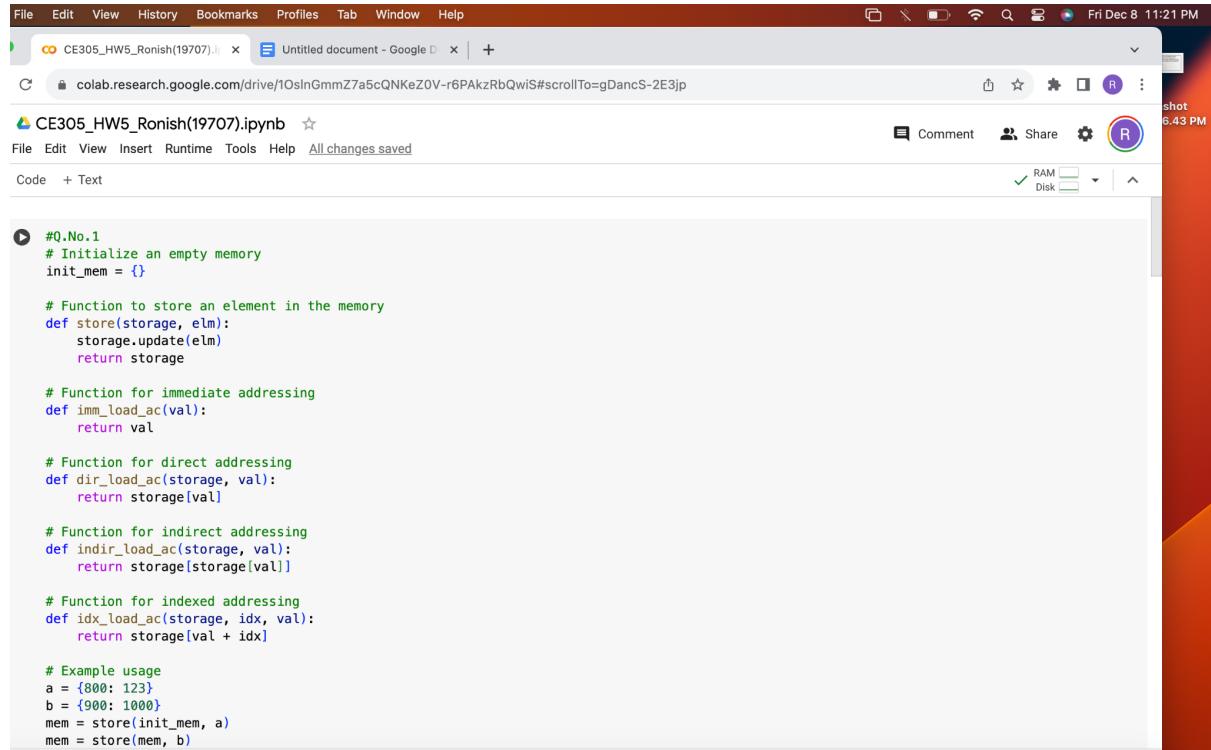


Q.no.1) In MARIE architecture ISA, there are several different types of addressing modes, such as immediate addressing, direct addressing, indirect addressing and indexed addressing. Please complete the following functions in Python to simulate how the different addressing modes work.



```

File Edit View History Bookmarks Profiles Tab Window Help
File colab.research.google.com/drive/1OslnGmmZ7a5cQNKeZ0V-r6PAkzRbQwiS#scrollTo=gDancS-2E3jp
CE305_HW5_Ronish(19707).ipynb Untitled document - Google Colab + Fri Dec 8 11:21 PM
File Edit View Insert Runtime Tools Help All changes saved
Code + Text
Comment Share R RAM Disk
# No.1
# Initialize an empty memory
init_mem = {}

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

# Function for immediate addressing
def imm_load_ac(val):
    return val

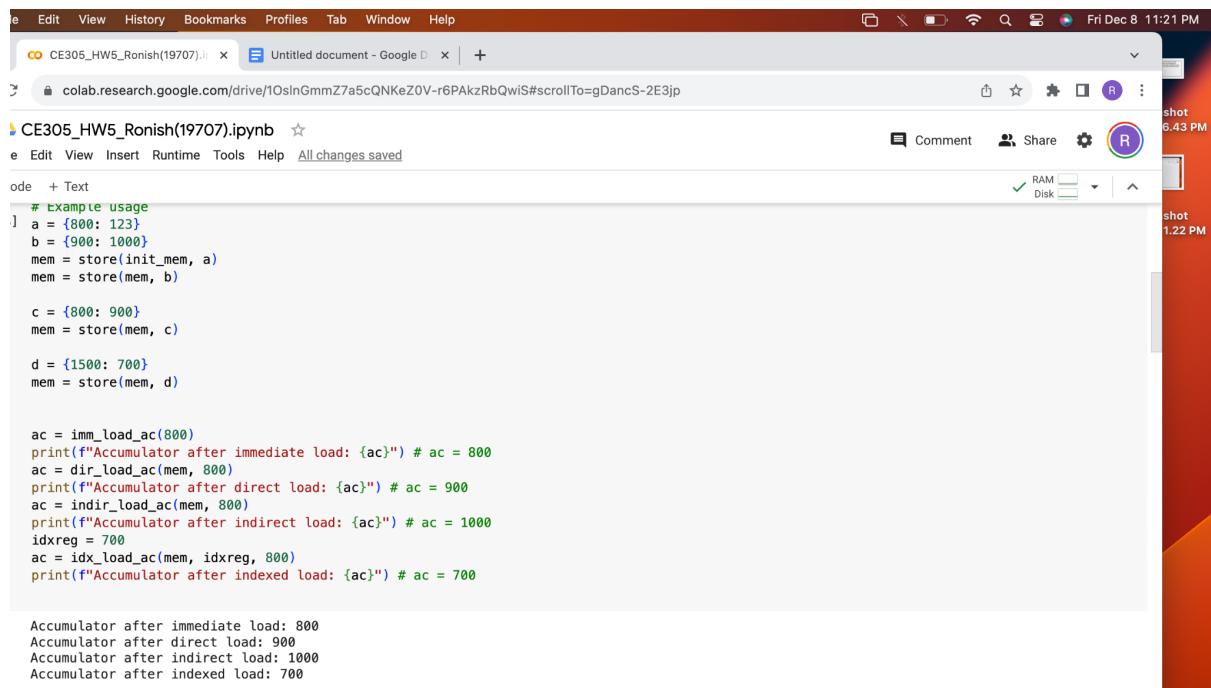
# Function for direct addressing
def dir_load_ac(storage, val):
    return storage[val]

# Function for indirect addressing
def indir_load_ac(storage, val):
    return storage[storage[val]]

# Function for indexed addressing
def idx_load_ac(storage, idx, val):
    return storage[val + idx]

# Example usage
a = {800: 123}
b = {900: 1000}
mem = store(init_mem, a)
mem = store(mem, b)

```



```

File Edit View History Bookmarks Profiles Tab Window Help
File colab.research.google.com/drive/1OslnGmmZ7a5cQNKeZ0V-r6PAkzRbQwiS#scrollTo=gDancS-2E3jp
CE305_HW5_Ronish(19707).ipynb Untitled document - Google Colab + Fri Dec 8 11:21 PM
File Edit View Insert Runtime Tools Help All changes saved
Code + Text
# example usage
a = {800: 123}
b = {900: 1000}
mem = store(init_mem, a)
mem = store(mem, b)

c = {800: 900}
mem = store(mem, c)

d = {1500: 700}
mem = store(mem, d)

ac = imm_load_ac(800)
print(f"Accumulator after immediate load: {ac}") # ac = 800
ac = dir_load_ac(mem, 800)
print(f"Accumulator after direct load: {ac}") # ac = 900
ac = indir_load_ac(mem, 800)
print(f"Accumulator after indirect load: {ac}") # ac = 1000
idxreg = 700
ac = idx_load_ac(mem, idxreg, 800)
print(f"Accumulator after indexed load: {ac}") # ac = 700

Accumulator after immediate load: 800
Accumulator after direct load: 900
Accumulator after indirect load: 1000
Accumulator after indexed load: 700

```

Q.No.2) If the main memory consists of 214 words, 211 blocks will be created in it, each block holds 8 words, and cache has $16 = 24$ 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.

The screenshot shows a Google Colab notebook titled "CE305_HW5_Ronish(19707).ipynb". The code defines three functions: store, dir_map_cache, and check_cache. The store function takes storage, block_address, and block_values as arguments and updates the storage. The dir_map_cache function takes cache, adr, and storage as arguments, extracts tag, block, and word from the address, and updates the cache if the block is present. The check_cache function takes cache and adr as arguments and returns "Hit" if the tag matches and the value is 1, or "Miss" otherwise. The notebook also includes initialization code for memory and cache, and example usage with address "00000110101000".

```
#0.No.2
def store(storage, block_address, block_values):
    storage[block_address] = block_values
    return storage

def dir_map_cache(cache, adr, storage):
    tag = adr[:7]      # 7-bit tag
    block = adr[7:11]  # 4-bit block
    word = adr[11:]   # 3-bit word

    block_address = tag + block

    if block_address in storage:
        cache[block] = [tag, storage[block_address], 1]  # Update cache line
    return cache

def check_cache(cache, adr):
    tag = adr[:7]      # 7-bit tag
    block = adr[7:11]  # 4-bit block

    if cache[block][0] == tag and cache[block][2] == 1:
        return "Hit"
    else:
        return "Miss"

# Initialize main memory and cache
init_mem = {}
cache = {format(i, '04b'): ["0000000", [], 0] for i in range(16)}

# Example usage
a_address = "00000110101000"
```

The screenshot shows a Google Colab notebook titled "CE305_HW5_Ronish(19707).ipynb". The code demonstrates the use of the functions defined in the previous screenshot. It starts by initializing memory and cache, then stores values at addresses 1 through 7. It then checks the cache for address 1, which is a hit. It then stores values at addresses 10 through 17, and checks the cache for address 2, which is a miss. Finally, it stores values at addresses 20 through 27 and checks the cache for address 3, which is a miss. The code uses hex addresses and prints the results of each cache check.

```
# Example usage
a_address = "00000110101000"
a_values = [0, 1, 2, 3, 4, 5, 6, 7]
mem = store(init_mem, a_address[:11], a_values)

adr1 = "00000110101010"
cache = dir_map_cache(cache, adr1, mem)

print("Check cache for adr1:", check_cache(cache, adr1))

# Storing additional blocks in main memory
b_address = "00001110101000"
b_values = [10, 11, 12, 13, 14, 15, 16, 17]
mem = store(mem, b_address[:11], b_values)

adr2 = "00001110101010"
cache = dir_map_cache(cache, adr2, mem)

print("Check Cache for adr2:", check_cache(cache, adr2))

c_address = "00001110111000"
c_values = [20, 21, 22, 23, 24, 25, 26, 27]
mem = store(mem, c_address[:11], c_values)

adr3 = "00001110111111" # hex address: 7BF
cache = dir_map_cache(cache, adr3, mem)

print("Check Cache for adr3:", check_cache(cache, adr3)) # Hit or Miss
```

```

CE305_HW5_Ronish(19707).ipynb
c_address = "00001110111000"
c_values = [20, 21, 22, 23, 24, 25, 26, 27]
mem = store(mem, c_address[:11], c_values)

adr3 = "00001110111111" # hex address: 7BF
cache = dir_map_cache(cache, adr3, mem)

print("Check Cache for adr3:", check_cache(cache, adr3)) # Hit or Miss

Check cache for adr1: Hit
Check Cache for adr2: Hit
Check Cache for adr3: Hit

```

Q.no.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.

```

#Q.no.3
# Initialize main memory
init_mem = {}

# Function to store elements in the main memory
def store(storage, elm):
    for tag, block in elm.items():
        storage[tag] = block
    return storage

# Initialize cache
# Each cache line has a structure: [tag, data block, valid bit]
cache = {
    "blk0": ["000000000000", [0,0,0,0,0,0,0,0], 0],
    "blk1": ["000000000000", [0,0,0,0,0,0,0,0], 0],
    "blk2": ["000000000000", [0,0,0,0,0,0,0,0], 0],
    "blk3": ["000000000000", [0,0,0,0,0,0,0,0], 0]
}

# Function to simulate fully associative cache behavior
def fully_ass_cache(cache, adr, storage):
    tag = adr[-3] # The tag is the address without the last 3 bits
    word_index = int(adr[-3:], 2) # The word index within the block

    # Check if data is already in the cache
    for block in cache.values():
        if block[0] == tag and block[2] == 1:
            # Cache hit, return the value
            return block[1][word_index]

```

File Edit View History Bookmarks Profiles Tab Window Help

Untitled document - Google Colab

colab.research.google.com/drive/1OslnGmmZ7a5cQNKeZ0V-r6PAkzRbQwiS#scrollTo=gDancS-2E3jp

CE305_HW5_Ronish(19707).ipynb

Comment Share R

RAM Disk

Code + Text

```
# Cache miss, load the data block from the main memory
if tag in storage:
    # Find the next available block or replace the first one
    for key in cache:
        if cache[key][2] == 0:
            cache[key] = [tag, storage[tag], 1]
            return cache[key][1][word_index]
    cache["blk0"] = [tag, storage[tag], 1] # If all blocks are used, replace the first one

return cache

# Storing data in memory
a = {"00000110101000": [0, 1, 2, 3, 4, 5, 6, 7]}
b = {"00001110101000": [10, 11, 12, 13, 14, 15, 16, 17]}
c = {"00011110101000": [20, 21, 22, 23, 24, 25, 26, 27]}
d = {"00111110101000": [30, 31, 32, 33, 34, 35, 36, 37]}
e = {"01111110101000": [40, 41, 42, 43, 44, 45, 46, 47]}

mem = store(init_mem, a) # Storing 'a' in memory
mem = store(init_mem, b) # Storing 'b' in memory
mem = store(init_mem, c) # Storing 'c' in memory
mem = store(init_mem, d) # Storing 'd' in memory
mem = store(init_mem, e) # Storing 'e' in memory

# Accessing cache
adr1 = "00000110101010" # hex address: 1AA
cache = fully_ass_cache(cache, adr1, mem)

adr2 = "00001110101010" # hex address: 3AA
cache = fully_ass_cache(cache, adr2, mem)
```

shot 6.43 PM
shot 1.22 PM
shot 1.32 PM
shot 1.43 PM
shot 1.54 PM
shot 2.00 PM
shot 2.10 PM

File Edit View History Bookmarks Profiles Tab Window Help

Untitled document - Google Colab

colab.research.google.com/drive/1OslnGmmZ7a5cQNKeZ0V-r6PAkzRbQwiS#scrollTo=gDancS-2E3jp

CE305_HW5_Ronish(19707).ipynb

Comment Share R

RAM Disk

Code + Text

```
de + Text

mem = store(init_mem, a) # Storing 'a' in memory
mem = store(init_mem, b) # Storing 'b' in memory
mem = store(init_mem, c) # Storing 'c' in memory
mem = store(init_mem, d) # Storing 'd' in memory
mem = store(init_mem, e) # Storing 'e' in memory

# Accessing cache
adr1 = "00000110101010" # hex address: 1AA
cache = fully_ass_cache(cache, adr1, mem)

adr2 = "00001110101010" # hex address: 3AA
cache = fully_ass_cache(cache, adr2, mem)

adr3 = "00011110101111" # hex address: 7AF
cache = fully_ass_cache(cache, adr3, mem)

adr4 = "00111110101101" # hex address: FAD
cache = fully_ass_cache(cache, adr4, mem)

adr5 = "01111110101110" # hex address: 1FAE
cache = fully_ass_cache(cache, adr5, mem)

# Print the cache to see the result
print(cache)
```

{'blk0': ['000000000000', [0, 0, 0, 0, 0, 0, 0, 0], 0], 'blk1': ['000000000000', [0, 0, 0, 0, 0, 0, 0, 0], 0], 'blk2': ['000000000000', [0, 0, 0, 0, 0, 0, 0, 0], 0]}

shot 6.43 PM
shot 1.22 PM
shot 1.32 PM
shot 1.43 PM
shot 1.54 PM
shot 2.00 PM
shot 2.10 PM