San Francisco Bay University

CE305 - Computer Organization
2023 Fall Homework #5

Due day: 12/8/2023

Kritika Regmi
ID:19702

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.

ANSWER:

```python
#Answer of no:1
init_mem = {}  # Empty memory at the very beginning
a = {800: 123}  # 1st data with address 800 and value 123
b = {900: 1000}  # 2nd data with address 900 and value 1000

def store(storage, elm):
    storage.update(elm)
    return storage

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

c = {800: 900}
mem = store(mem, c)  # mem = {800: 900, 900: 1000}

d = {1500: 700}
mem = store(mem, d)  # mem = {800: 900, 900: 1000, 1500: 700}

def imm_load_ac(val):
    return val

ac = imm_load_ac(800)  # ac = 800
```

```python
def dir_load_ac(storage, val):
    return storage.get(val, 0)

ac = dir_load_ac(mem, 800)  # ac = 900

def indir_load_ac(storage, val):
    indir_addr = storage.get(val, 0)
    return storage.get(indir_addr, 0)

ac = indir_load_ac(mem, 800)  # ac = 1000

def idx_load_ac(storage, idx, val):
    idx_addr = storage.get(val, 0)
    return storage.get(idx_addr + idx, 0)

idxreg = 700
ac = idx_load_ac(mem, idxreg, 800)  # ac = 700

print("Final Memory State:", mem)
print("Final Accumulator Value:", ac)
```

OUTPUT:

```
Final Memory State: {800: 900, 900: 1000, 1500: 700}
Final Accumulator Value: 0
```

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.

ANSWER: This program defines functions for storing data in memory and mapping the memory to cache in a direct-mapped manner. It also checks if a given address is present in the cache or not.

```
#Answer of no: 2
init_mem = {}  # Empty memory at the very beginning

# Take memory address as the key and all values in the block as the key's value
a = {"00000110101000": [0, 1, 2, 3, 4, 5, 6, 7]}
b = {"00001110101000": [10, 11, 12, 13, 14, 15, 16, 17]}

def store(storage, elm):
    storage.update(elm)
    return storage

mem = store(init_mem, a)
mem = store(mem, b)

# Cache line format: [tag (7 bits), values of 8 words, valid (1 bit)]
cache = {"0000": ["0000000", [0, 0, 0, 0, 0, 0, 0, 0], 0],
         "0001": ["0000000", [0, 0, 0, 0, 0, 0, 0, 0], 0],
         "0010": ["0000000", [0, 0, 0, 0, 0, 0, 0, 0], 0],
         "0011": ["0000000", [0, 0, 0, 0, 0, 0, 0, 0], 0],
         "0100": ["0000000", [0, 0, 0, 0, 0, 0, 0, 0], 0],
         "0101": ["0000000", [0, 0, 0, 0, 0, 0, 0, 0], 0],
         "0110": ["0000000", [0, 0, 0, 0, 0, 0, 0, 0], 0],
         "0111": ["0000000", [0, 0, 0, 0, 0, 0, 0, 0], 0],
         "1000": ["0000000", [0, 0, 0, 0, 0, 0, 0, 0], 0],
         "1001": ["0000000", [0, 0, 0, 0, 0, 0, 0, 0], 0],
         "1010": ["0000000", [0, 0, 0, 0, 0, 0, 0, 0], 0],
         "1011": ["0000000", [0, 0, 0, 0, 0, 0, 0, 0], 0],
         "1100": ["0000000", [0, 0, 0, 0, 0, 0, 0, 0], 0],
         "1101": ["0000000", [0, 0, 0, 0, 0, 0, 0, 0], 0],
         "1110": ["0000000", [0, 0, 0, 0, 0, 0, 0, 0], 0],
         "1111": ["0000000", [0, 0, 0, 0, 0, 0, 0, 0], 0]}
```

```
adr1 = "00000110101010"  # hex address: 1AA

def dir_map_cache(cache, adr, storage):
    tag = adr[:7]
    block = adr[7:11]
    word = adr[11:]

    block_data = storage.get(adr, [0] * 8)

    cache_line = cache[block]
    cache_tag = cache_line[0]

    if cache_tag == tag:  # Hit
        cache_line[1] = block_data
        cache_line[2] = 1
    else:  # Miss
        if cache_line[2] == 1:  # If the line is valid, write back to memory
            storage[cache_tag + block] = cache_line[1]
        cache[block] = [tag, block_data, 1]

    return cache

cache = dir_map_cache(cache, adr1, mem)

adr2 = "00001110101010"  # hex address: 3AA

cache = dir_map_cache(cache, adr2, mem)

c = {"00001110111000": [20, 21, 22, 23, 24, 25, 26, 27]}
mem = store(mem, c)

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

def check_cache(cache, adr):
    block = adr[7:11]
    cache_line = cache[block]

    if cache_line[2] == 1:  # If the line is valid
        print("Hit")
    else:
        print("Miss")

check_cache(cache, adr1)
```

```
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.

ANSWER:

```
#Answer of No:3
init_mem = {}  # Empty memory at the very beginning

# Take memory address as the key and all values in the block as the key's value
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]}

def store(storage, elm):
    storage.update(elm)
    return storage

mem = store(init_mem, a)
mem = store(mem, b)
mem = store(mem, c)
mem = store(mem, d)
mem = store(mem, e)

# Initialize cache
# Cache format: key -> block label
#               value -> tag(11 bits), values of 8 words, valid(1 bit)
# Assume that there are only 4 cache lines
cache = {"blk0": ["00000000000", [0, 0, 0, 0, 0, 0, 0, 0], 0],
         "blk1": ["00000000000", [0, 0, 0, 0, 0, 0, 0, 0], 0],
         "blk2": ["00000000000", [0, 0, 0, 0, 0, 0, 0, 0], 0],
         "blk3": ["00000000000", [0, 0, 0, 0, 0, 0, 0, 0], 0]}
```

```python
def fully_ass_cache(cache, adr, storage):
    tag = adr[:11]
    block_data = storage.get(adr, [0] * 8)

    # Check for an available cache line with a valid bit of 0
    for block, block_info in cache.items():
        if block_info[2] == 0:
            cache[block] = [tag, block_data, 1]
            return cache

    # If all cache lines are occupied, evict the least recently used line (LRU)
    lru_block = min(cache, key=lambda x: cache[x][2])
    cache[lru_block] = [tag, block_data, 1]

    return 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)
```

PRINTING AND OUTPUT:

```python
print("Final Cache State:")
for block, block_info in cache.items():
    print(f"{block}: {block_info}")
```

```
Final Cache State:
blk0: ['01111110101', [0, 0, 0, 0, 0, 0, 0, 0], 1]
blk1: ['00001110101', [0, 0, 0, 0, 0, 0, 0, 0], 1]
blk2: ['00011110101', [0, 0, 0, 0, 0, 0, 0, 0], 1]
blk3: ['00111110101', [0, 0, 0, 0, 0, 0, 0, 0], 1]
```