Virtually Indexed Virtually Tagged (VIVT)

HIGH SPEED COMPUTER ARCHITECTURE

Group 4

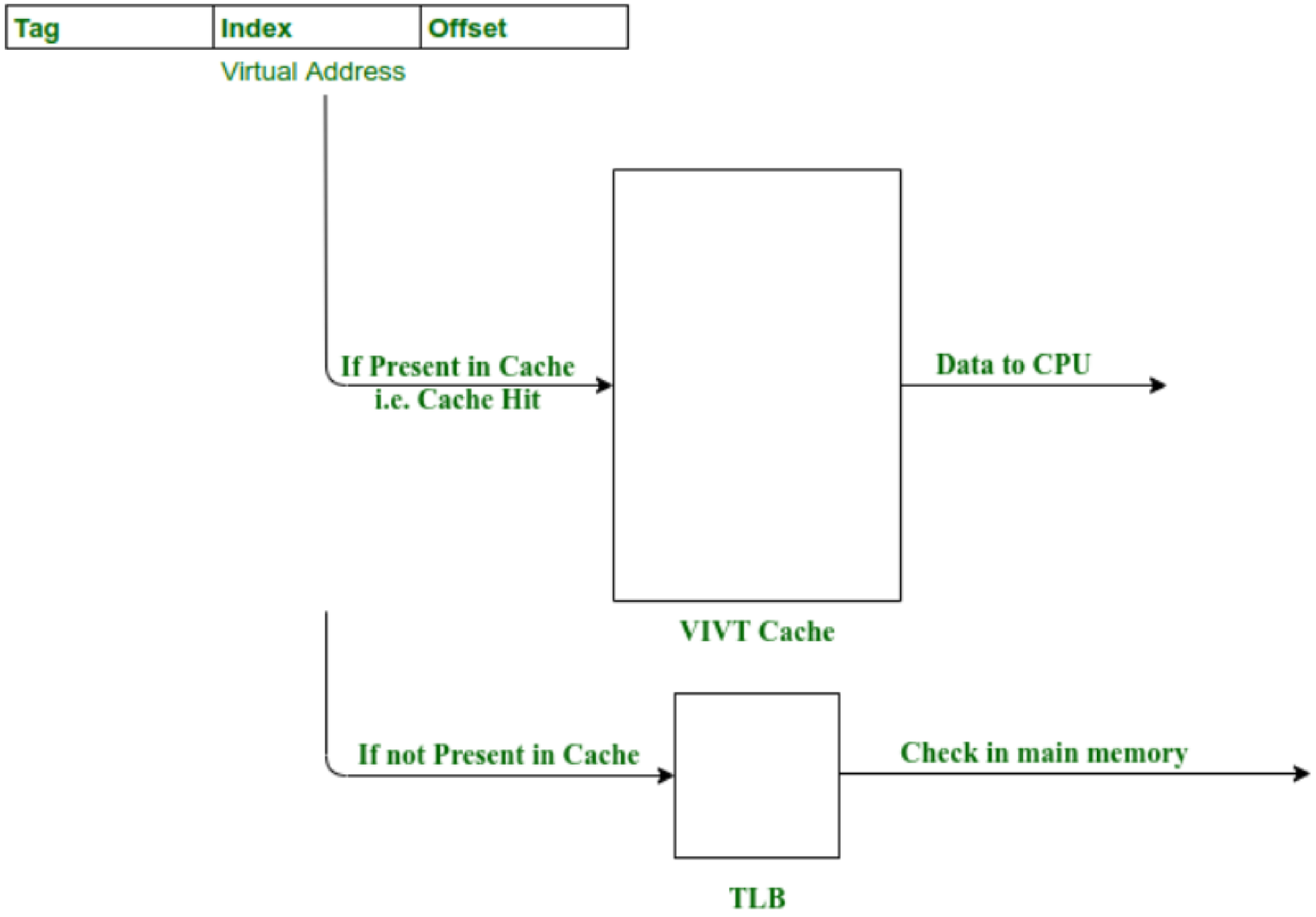
Kuntal Panchal AU2040141
Jay Shapariya AU2040208
Kanvi Patel AU2040235
Deval Patel AU2040017

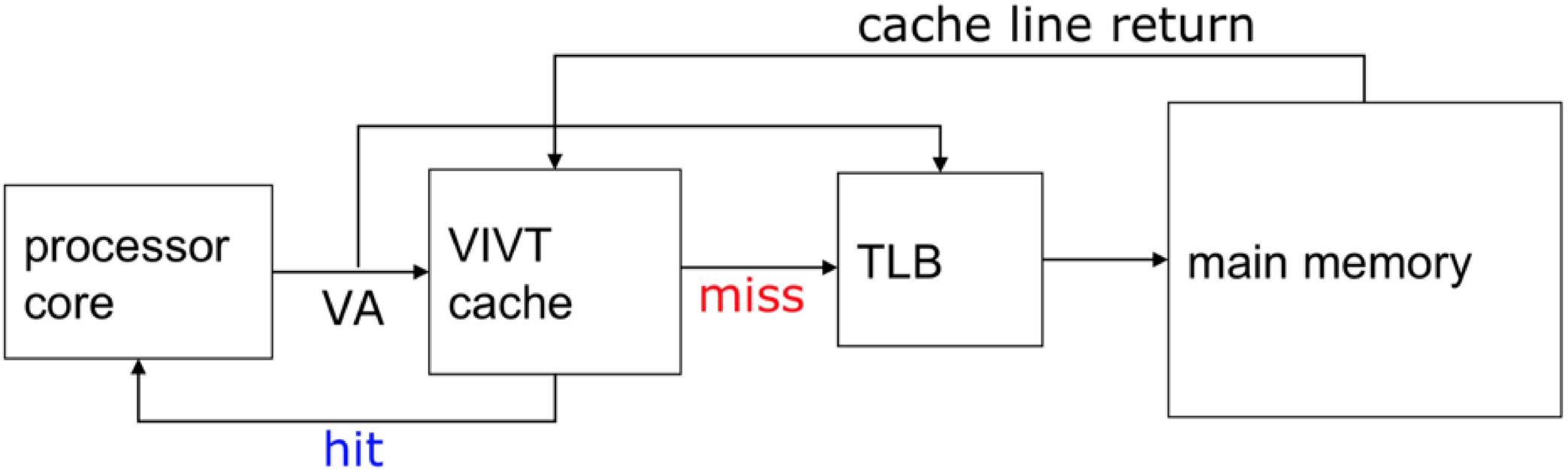
Introduction

- Virtually Indexed Virtually Tagged (VIVT) is a type of cache memory organization used in computer systems.
- It is a compromise between the Virtually Indexed Physically Tagged (VIPT) and Virtually Tagged Physically Indexed (VTPI) cache architectures.
- VIVT caches use virtual addresses for indexing and virtual tags for matching cache lines.
- They are commonly found in modern CPUs, including those in desktops, laptops, and mobile devices.

How It works

1. CONVERT VIRTUAL ADDRESS TO CACHE INDEX
2. COMPARE THE VIRTUAL TAG WITH CACHE TAG
3. RETRIEVE OR UPDATE DATA
4. UPDATE THE CACHE


Tag
Index
Offset
Virtual Address
If Present in Cache
i.e. Cache Hit
Data to CPU
VIVT Cache
If not Present in Cache
Check in main memory
TLB


cache line return
processor core
VA
VIVT cache
miss
TLB
main memory
hit

Advantages

- **Faster Cache Access:** In a VIVT cache, virtual address translation and cache indexing can be performed simultaneously, resulting in faster cache access times.
- **Reduced Power Consumption:** A VIVT cache requires less power consumption compared to other cache organizations, as it does not need to store the physical address tags.
- **Reduced Complexity:** The cache indexing logic in a VIVT cache is simpler, as it only uses the virtual address to determine the cache index.
- **Reduced Memory Access:** A VIVT cache requires fewer memory accesses during virtual-to-physical address translation.

Limitations

- **Increased Complexity:** VIVT cache requires additional logic for maintaining consistency between the cache and main memory, as the cache may contain stale data due to its virtual indexing.
- **Increased Cache Miss Penalty:** In a VIVT cache, if the cache index for a virtual address is not found in the cache, a physical address translation is required to locate the data, resulting in an increased cache miss penalty.
- **Increased Hardware Overhead:** Implementing a VIVT cache requires additional hardware support to manage the cache tags and ensure data consistency, which may increase hardware overhead and design complexity.
- **Increased Complexity in Multicore Systems:** In a multicore system, a VIVT cache requires additional coordination to maintain consistency between the caches of different cores.

Applications

- **High-performance computing:** VIVT systems are commonly used in high-performance computing environments such as supercomputers, where fast access to data is crucial.
- **Embedded systems:** VIVT systems are used in embedded systems such as mobile phones, where the small size and low power consumption of the cache is important.
- **Real-time systems:** VIVT systems are used in real-time systems such as automotive applications, where fast access to data is necessary.
- **Video processing:** VIVT systems are used in video processing applications, such as in digital TVs and video game consoles, where the fast access to data is essential for smooth video playback.

1. Initialization of Variables:

```
# Initialize the cache as a dictionary of cache lines
cache = {}
cache_size = int(input("Enter cache size in bytes: "))
block_size = int(input("Enter block size in bytes: "))
num_blocks = cache_size // block_size

# Initialize the LRU counter and the cache hit/miss counters
lru_counter = 0
cache_hits = 0
cache_misses = 0
```

2. Cache Lookup Function

If cache hits-->

```python
# Define the cache lookup function
def cache_lookup(virtual_address, operation):
    global lru_counter, cache_hits, cache_misses, cache

    # Extract the tag and index from the virtual address
    tag = virtual_address // cache_size

    index = (virtual_address % cache_size) // block_size


    # Check if the cache line is present in the cache
    if index in cache and cache[index]['tag'] == tag:
        # Increment the LRU counter for all cache lines with a lower counter value
        for i in cache:
            if cache[i]['lru'] < cache[index]['lru']:
                cache[i]['lru'] += 1
        # Reset the LRU counter for the hit cache line
        cache[index]['lru'] = 0
        # Update the dirty bit if the operation is a write
        if operation == 'write':
            cache[index]['dirty'] = True
        # Increment the cache hit counter and return the data
        cache_hits += 1
        return cache[index]['data']
```

3. Cache Miss Case

```
else:
    # Increment the LRU counter for all cache lines
    for i in cache:
        cache[i]['lru'] += 1
    # Check if the cache is full
    if len(cache) < num_blocks:
        # Insert the new cache line
        cache[index] = {'tag': tag, 'data': '0x' + hex(virtual_address)[2:].zfill(8), 'dirty': False, 'lru': 0}
    else:
        # Find the LRU cache line and replace it
        max_lru = max([cache[i]['lru'] for i in cache])
        for i in cache:
            if cache[i]['lru'] == max_lru:
                # Update the main memory if the cache line is dirty
                if cache[i]['dirty']:
                    main_memory_address = cache[i]['tag'] * cache_size + i * block_size
                    main_memory_data = cache[i]['data']
                    print(f"Updating main memory at 0x{hex(main_memory_address)[2:].zfill(8)} with data {main_memory_data}")
                # Replace the cache line
                cache[i] = {'tag': tag, 'data': '0x' + hex(virtual_address)[2:].zfill(8), 'dirty': False, 'lru': 0}
                break
    # Update the dirty bit if the operation is a write
    if operation == 'write':
        cache[index]['dirty'] = True
    # Increment the cache miss counter and return the data
    cache_misses += 1
    return '0x' + hex(virtual_address)[2:].zfill(8)
```

2. Cache Lookup Function

```
# Test the cache lookup function with sample virtual addresses
virtual_address = int(input("Enter virtual address: "), 16)
operation = input("Enter operation (read or write): ")
data = cache_lookup(virtual_address, operation)
print(f"Data: {data}")
print(f"Cache hits: {cache_hits}")
print(f"Cache misses: {cache_misses}")
```

5. Looping and User Interaction

```
while True:
  virtual_address = int(input("Enter virtual address (in hex): "), 16)
  operation = input("Enter operation (read/write): ")
  # Perform the cache lookup
  data = cache_lookup(virtual_address, operation)

  # Print the results
  print(f"Virtual address: 0x{hex(virtual_address)[2:].zfill(8)}")
  print(f"Operation: {operation}")
  print(f"Data: {data}")
  print(f"Cache hits: {cache_hits}")
  print(f"Cache misses: {cache_misses}")
  print("Cache contents:")
  for i in range(num_blocks):
      if i in cache:
          print(f"Block {i}: tag={cache[i]['tag']}, data={cache[i]['data']}, dirty={cache[i]['dirty']}, LRU={cache[i]['lru']}")
      else:
          print(f"Block {i}: empty")
  print()

  # Ask the user if they want to continue
  choice = input("Do you want to continue (yes/no)? ")
  if choice.lower() != "yes":
      break
  print("Exiting the program...")
```