Sample Inputs to test :

Commands :

1. init <size> : Initialize physical memory size
2. config cache <L1|L2> : Configure Cache (ex: config cache L1 2048 64 2)
3. set allocator <type> : Set allocator (first, best, worst, buddy)
4. set policy <type> : Set VM replacement policy (FIFO, LRU)
5. malloc <size> : Allocate virtual memory block
6. free <id> : Free memory block
7. read <virtual_addr> : Read Address (Read)
8. write <virtual_addr> : Write Address (Sets Dirty Bit)
9. dump : memory dump
10. stats : Show All Stats
11. exit : Exit

Test 1. Allocation / Deallocation :

```
# Reset memory
init 1024

# A. FIRST FIT (Default)
set allocator first
malloc 100   # ID 0
```

```
malloc 200   # ID 1
malloc 300   # ID 2
free 1       # Create a hole of 200 bytes in the middle
malloc 150   # Should take the first hole (ID 1's old spot)
dump         # Verify ID 3 is at address 100

# B. BEST FIT
# Reset for clean slate
init 1024
set allocator best
malloc 100   # ID 0
malloc 200   # ID 1
malloc 300   # ID 2
malloc 100   # ID 3
free 1       # Hole of 200 bytes
free 3       # Hole of 100 bytes
malloc 90    # Request 90 bytes
# Logic: First Fit would take the 200 hole. Best Fit MUST
take the 100 hole.
dump         # Verify the new block is in the smaller hole

# C. WORST FIT
init 1024
set allocator worst
malloc 100
malloc 200
malloc 100
free 1       # Hole of 200
free 2       # Hole of 100 (at end)
```

malloc 50   # Request 50
Logic: Worst Fit should take the largest hole (200), not the 100.
dump
stats

Test 2. Buddy Allocator :

init 1024

set allocator buddy

1. Allocation (Splitting)

malloc 10   # Request 10 -> Rounds to 16. Splits 1024->512->256...->16

malloc 100   # Request 100 -> Rounds to 128.

malloc 200   # Request 200 -> Rounds to 256.

dump   # Check the "Order" list. You should see used blocks of sizes 16, 128, 256.

2. Deallocation (Merging)

We need to free blocks that are "Buddies" to see them merge.

If malloc 10 got ID 0, it used the first 16-byte block.

free 0

This should trigger a merge up the chain if its buddy is free.

dump

3. Stats & Internal Fragmentation

stats # Look for "Internal Fragmentation" (e.g. Req 10 vs Alloc 16 = 6 bytes lost)

<u>Test 3. Cache Implementation :</u>

<u>Read Operation :</u>

- L1 Hit :

config cache L1 1024 64 2

init 1024

set policy LRU

read 0x00 # Miss all (RAM fetch)

read 0x00 # L1 Hit

- L2 Hit :

1. Initialize large memory (so we don't run out of frames)

init 65536

set policy LRU

2. Load the First Block (Target) -> Frame 0

read 0x0

(Maps to L1 Set 0. L1 holds: [0x0])

```
# 3. BURN FRAMES 1-7 (Fill Frames 1,2,3,4,5,6,7)
# We read dummy addresses just to consume these frames.
read 0x40
read 0x80
read 0xC0
read 0x100
read 0x140
read 0x180
read 0x1C0

# 4. Load Collision Block #1 -> Frame 8
# This corresponds to Physical Address 0x200.
# 0x200 maps to L1 Set 0!
read 0x200
# (L1 Set 0 now holds: [0x0, 0x200]) -> FULL!

# 5. BURN FRAMES 9-15
read 0x240
read 0x280
read 0x2C0
read 0x300
```

read 0x340

read 0x380

read 0x3C0

6. Load Collision Block #2 -> Frame 16

This corresponds to Physical Address 0x400.

0x400 maps to L1 Set 0!

read 0x400

(L1 Set 0 is full. LRU Policy evicts the oldest: 0x0)

(0x0 is kicked out of L1, but it stays in L2)

7. THE MOMENT OF TRUTH

Read 0x0 again.

read 0x0

WRITE OPERATION :

- **To see writing back to memory on eviction :**

1. Setup: Large RAM, LRU Policy

init 65536

set policy LRU

2. The Target: Write to Virtual Address 0x0

```
# MMU maps this to Frame 0 (Physical 0x0).

# Cache puts this in Set 0.

# Action: It sets the 'Dirty' bit to TRUE.

write 0x0


# 3. The "Frame Burner" (Frames 1-7)

# We need to use up Frames 1-7 so the MMU will give us Frame 8 next.

# These go into Sets 1-7, so they don't interfere with our target in Set 0.

read 0x40

read 0x80

read 0xc0

read 0x100

read 0x140

read 0x180

read 0x1c0


# 4. Collision #1: Read Virtual 0x200

# MMU maps this to Frame 8 (Physical 0x200).

# Physical 0x200 maps to Set 0.

# L1 Set 0 now holds two items: [0x0 (Dirty), 0x200 (Clean)].
```

```
read 0x200

# 5. The "Frame Burner" Round 2 (Frames 9-15)
# Use up frames so the MMU gives us Frame 16 next.
read 0x240
read 0x280
read 0x2c0
read 0x300
read 0x340
read 0x380
read 0x3c0

# 6. The Trigger: Read Virtual 0x400
# MMU maps this to Frame 16 (Physical 0x400).
# Physical 0x400 maps to Set 0.
# Set 0 is FULL (Size 2). It must evict the oldest block (0x0).
# Since 0x0 is DIRTY, it must trigger a write-back!
read 0x400
```

- **To see dirty block getting hit** :

  ```
  init 1024
  write 0x00 # reading RAM , dirty bit = true

  write 0x00 # L1 Hit , marked to dirty = true
  ```

Test 4. Virtual Memory Implementation :

Part 1: FIFO Policy Test (First-In, First-Out)

We will verify that the "oldest loaded" page gets kicked out, regardless of how popular it is.
Bash

```
# 1. Setup: Tiny RAM (192 bytes = exactly 3 Frames of 64 bytes)
init 192
set policy FIFO

# 2. Fill the RAM (Load Pages 0, 1, 2)
# Frame 0 gets VPN 0
read 0x0
# Frame 1 gets VPN 1
read 0x40
# Frame 2 gets VPN 2
read 0x80
# RAM State: [VPN 0 (Oldest), VPN 1, VPN 2 (Newest)]

# 3. Read Page 0 again (Make it "popular")
# In FIFO, this DOES NOT save it. It was still loaded first.
read 0x0

# 4. Force Eviction (Load Page 3)
# RAM is full. FIFO looks at load time. VPN 0 is oldest.
```

read 0xC0

5. Verify Eviction
Read Page 0 again.
IF LOGIC IS CORRECT: It should say "Page Fault" (it was kicked out).
read 0x0

Part 2: LRU Policy Test (Least Recently Used)

Now we verify that the "least popular" page gets kicked out, even if it was loaded recently.

```
# 1. Reset: Tiny RAM again

init 192

set policy LRU


# 2. Fill RAM (Load Pages 0, 1, 2)

read 0x0    # (Timestamp: 1)

read 0x40   # (Timestamp: 2)

read 0x80   # (Timestamp: 3)


# 3. "Touch" the Oldest Pages (Make them popular)

read 0x0    # VPN 0 Timestamp updates to 4 (Saved!)

read 0x40   # VPN 1 Timestamp updates to 5 (Saved!)
```

```
# Current State:
# VPN 0: Used at time 4
# VPN 1: Used at time 5
# VPN 2: Used at time 3 (Least Recently Used!)

# 4. Force Eviction (Load Page 3)
# LRU should skip 0 and 1, and kill VPN 2.
read 0xC0

# 5. Verify
# Read VPN 0 -> Should be a HIT (it was saved).
read 0x0
# Read VPN 2 -> Should be a PAGE FAULT (it was the victim).
read 0x80
```

****