Istanbul Technical University

Department of Computer Engineering

BLG 322E - COMPUTER ARCHITECTURE

Assignment 4

Due Date: Wednesday, May 23, 2025, 23:59.

- Please **write your full name** (first name and last name) **and Student ID** at the top of your solution.
- Please show ALL your work. Answers with no supporting explanations or work will not receive any partial credit. Your homework is not just a final report of your results; we want to see your steps. Upload all the papers you worked on to get to the solution.
- Please follow all formatting instructions exactly (tables, address notation, naming conventions, etc.). Use the assignment file (.docx) directly by filling in blank (red) fields or copy them faithfully in your own report.
- **Submissions:** Submit your solution as a PDF file to Ninova before the deadline.
- **No late submissions** will be accepted. Do not send your solutions by e-mail. We will only accept files that have been uploaded to the official Ninova e-learning system before the deadline. Do not risk leaving your submission to the last few minutes.
- **Consequences of plagiarism/cheating:** Assignments have to be done individually. Any cheating will be subject to disciplinary action.
- **AI usage:** Language and reasoning models (ChatGPT, o1, o3, Claude, etc.) are tested on the questions. They hallucinate on some aspects of questions and give the wrong answers. You can use them to test alternative ideas, but you should always check the results.

If you have any questions, please e-mail **Mustafa Abdullah Hakkoz (hakkozm22@itu.edu.tr) or use the message board in Ninova.**

Student Name: Mustafa Can Çalışkan
Student No: 150200097

Assignment 4: Cache-Memory Analysis and Access Time

A computer system has the following characteristics:

Parameter	Value
Main memory	**512 KByte** (19-bit physical addresses)
L1 data-cache	**512 bytes, 4-way set-associative**
Block (line) size	**16 bytes**
Replacement	**LRU** with a 2-bit ageing counter (00 = oldest … 11 = most-recent)
Read policy	Read-through
Write policy	Write-through + Write-allocate
Transfers	Whole 16-byte blocks move between memory and cache

The cache is **initially empty**. The CPU executes the code below.

Program in c (outer loop runs exactly two times)

```
for (int i = 1; i <= 2; ++i)
{
   /* 1. A - forward scan (16 elements)*/
   for (int j = 0; j < 16; ++j)        // A[0]...A[15]
      t += A[j];

   /* 2. B - backward scan (16 elements)*/
   for (int j = 15; j >= 0; --j)       // B[15]...B[0]
      t += B[j];

   /* 3. C - forward scan (24 elements) */
   for (int j = 0; j < 24; ++j)        // C[0]...C[23]
      t += C[j];

   /* 4. D - ping-pong scan, starts->both sides, ends->centre (32 elements) */
   for (int j = 0; j < 16; ++j) {
      t += D[j];
      t += D[31-j];
   }

   /* 5. E - stride-2 scan (48 elements)*/
   for (int j = 0; j < 48; j += 2)     // E[0],E[2]...E[46]
      t += E[j];
}
```

Array	Base address	Size
A	0x0000 A0	16 B
B	0x0040 20	16 B
C	0x0150 10	24 B
D	0x0200 90	32 B
E	0x0280 80	48 B

All elements are **one byte**; the loops **read only**.

Use timing symbols:

- t_c – cache-hit time
- t_m – main-memory access time
- t_b – block-transfer time
  $t_b = 2t_m = 10t_c$,
  $t_b > 0$

Tasks

a) **(5 pts)** Into which fields does the cache controller divide the physical address? Specify field names and bit widths (Tag, Set Index, Block Offset/Word bits).

Table-1: Address-field breakdown

Field	Bits	Reason
Block Offset (Word bits)	4	The block line size is 16. 4 bits are enough to represent offset.
Set Index	3	512/16 = 32 frame. Each frame consists of 4 blocks (4-way). 32/4 = 8, meaning that 3 bits are enough to represent set offset.
Tag	12	Physical address size – block offset – set index = remaining = 12.

b) (5 pts) Describe the **tag-store memory**:

- number of sets (rows)
- number of ways (frames) per set
- tag width per way
- total tag-memory size in **bytes** (include status bits: 1 valid bit, 1 dirty bit, and 2 LRU-ageing bits per frame).

Table-2:Tag-memory size calculation

Item	Calculation
Sets	512/16 = 32 frame. Each frame consists of 4 blocks (4-way). 32/4 = 8. There are 8 sets.
Ways (frames) per set	4-ways. 4 frame. It is given.
Tag bits per frame	12 bits. Found in part a).
Status bits per frame	1 valid bit + 1 dirty bit + 2 aging bits = 4 bits.
Bits per frame	12 (tag bits) + 4 (status bits) = 16.
Bits per row	16 * 4 (4-way) = 64. I assume that row indicates set.
Bytes per row	64 / 8 = 8 bytes.
Total tag memory	8 sets * 8 bytes = 64 bytes.

c) (30 pts) For the **first outer-loop iteration only** (i = 1), show **which memory blocks map to which set and way** after the entire iteration completes. Indicate **misses, hits, and replacements** by filling following table:

Table-3: Distinct blocks (example row for A already filled)

Array	Block label	Hex base	19-bit Tag | Set Idx | Word Number	Set #	1st byte of the block (Hit or Miss)
A	A_0	0x0000 A0	000 0000 0000 1 | 010 | 0000	2	M
B	B_0	0x0040 20	000 0100 0000 0 | 010 | 0000	2	H
C	C_0	0x015010	001 0101 0000 0 | 001 | 0000	1	M

C	C_1	0x015020	001 0101 0000 0 | 010 | 0000	2	M
D	D_0	0x020090	010 0000 0000 1 | 001 | 0000	1	M
D	D_1	0x0200A0	010 0000 0000 1 | 010 | 0000	2	H
E	E_0	0x028080	010 1000 0000 1 | 000 | 0000	0	M
E	E_1	0x028090	010 1000 0000 1 | 001 | 0000	1	M
E	E_2	0x0280A0	010 1000 0000 1 | 010 | 0000	2	M

NOTE: A single row in the table represents one 16-byte cache block. If an array spans several blocks, just insert additional rows to list those blocks. The example of array A is 16 bytes long, so it fits entirely in one block (A_0) and needs only that single row. If A were larger than 16 bytes you would add rows A_1, A_2, and so on for each extra block.

After filling table 3, illustrate the evolving LRU order for any set that receives more than four distinct blocks. Provide the **contents of the tag-store rows** you touched (tag+valid+dirt+ageing counter) at the end of this first iteration by filling following table:

Table-4: Cache snapshot after pass 1 (i==1) *(example row shows the format for set 0)*

Set	**Way0** Tag(3) in hex** | V(1)D(1)Age(2) bits	**Way1** Tag(3) in hex | V(1)D(1)Age(2) bits	**Way2** Tag(3) in hex | V(1)D(1)Age(2) bits	**Way3** Tag(3) in hex | V(1)D(1)Age(2) bits	**Removed blocks***
0	0x501 | 1011 (E_0)	–	–	–	–
1	0x2A0 | 1001 (C0)	0x401 | 1010 (D0)	0x501 | 1011 (E1)	-	-
2	0x080 | 1000 (B0)	0x2A0 | 1001 (C1)	0x401 | 1010 (D1)	0x501 | 1011 (E2)	A0

* Please write tag bits in hexadecimal and status bits (valid(1), dirty(1), ageing(2)) in binary. Use 11 to mark the most-recently-used (MRU) frame and 00 for the least-recently-used (LRU) frame.
** Blocks removed from the row of sets to make space for new ones. For the case of LRU, it is in the least used frame. Indicate them here.

d) (20 pts) Now let the outer loop finish all two passes.

i. How many **total replacements** occur during the entire run? Fill the table below and show your work.
ii. Compute the **overall hit ratio**. Calculate hits and misses on byte level and fill the table. Remember when you check for miss, you only need to look at first byte of the block. On the other hand, for counting hits, you need to look at all bytes of the block.

Table-5: Hit/Miss calculations

Metric	After pass 1 (i==1)	After pass 2 (i==2)	Total
Byte hits	103	107	210
Byte misses	9	5	14
Replacements	1 block (16 bytes)	5 block (80 bytes)	6
Hit ratio (bytes)	**0.92**	**0.96**	0.94

e) (10 pts) Propose a **new starting address for array E** (expressed in hexadecimal) that **reduces conflict misses** without changing any of the other addresses. Explain concisely **why** your choice helps.

Answer for part e:
New start address: 0x028030. Reason: At this address, the set number starts with 3, and when the entire array is fetched, it is distributed across sets 3, 4, and 5. This way, since no more than 4 blocks will be mapped to Set 2, the extreme conflict is resolved.

f) (10 pts) Using your miss count for pass 1 in **part c**, compute the **stall time.** Express the result in t_c.

$$T_1 = misses(t_m + t_b) + hits(t_c)$$

Answer for part f:
T1 = 9(5tc+10tc)+103tc=9(15tc)+103tc=238tc

g) (10 pts) Repeat the calculation after pass 2 (total time passed) and derive the **average time for memory access.**

$$Av.time = \frac{total\ time}{number\ of\ total\ byte\ accesses}$$

Answer for part g:
Total time = 14(5tc+10tc)+210tc=420tc Number of total byte access = 224 Av. Time = 1.88tc

h) (10 pts) Explain briefly why a FIFO policy would or would not change your numerical average time for this specific program.

Answer for part h:
If FIFO had been used, the result would not change. This is because each array is accessed only once and in order within the loop (in the order A B C D E), and since the same array is not accessed again within the same loop (the aging counters of the arrays decrease (getting older) according to their access order), the block that entered the cache first (the oldest one in LRU) is always evicted when replacement is occured. This causes LRU to behave like FIFO in this code.