

# 2017 Exam - Computer Architecture

## CS3021

---

Two Hour Exam

Answer 3 out of 4 Questions

20 marks per question

40 mins per question

---

## Question 1

*i) What is a pipelined processor? What are the benefits of pipelining? Explain the organization and operation of the DLX five stage execution pipeline.*

A pipelined processor is a processor that allows for its execution cycle to be broken down into multiple different stages of execution. These executions are then executed in the flow of a consecutive pipeline which allows for multiple execution stages to be executed concurrently thus increasing the overall performance time of execution. Each instruction is broken into a series of small steps and executed in parallel. The clock rate is thus the time taken for the longest step to be completed.

### DLX 5 Stage Pipeline

- **IF** - Instruction Fetch
- **ID** - Instruction Decode & Register Fetch
- **EX** - Execution & Effective Address Calculation
- **MA** - Memory Access
- **WB** - Write Back

<i>i</i>	IF	ID	EX	MA	WB				
<i>i+1</i>		IF	ID	EX	MA	WB			
<i>i+2</i>			ID	ID	EX	MA	WB		
<i>i+3</i>				IF	ID	EX	MA	WB	
<i>i+4</i>					IF	ID	EX	MA	WB

*ii) What are data hazards? Describe two techniques to prevent stalls which can be used to overcome data hazards in the DLX pipeline. Show, using diagrams, how the data hazards in the following code sequence are overcome by the DLX CPU.*

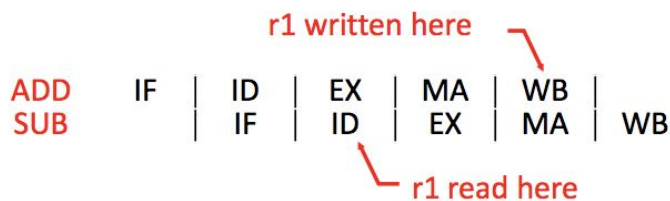
### Data Hazards

Data hazards occur when instruction  $x+1$  depends on a value used/changed in instruction  $x$ , for example:

---

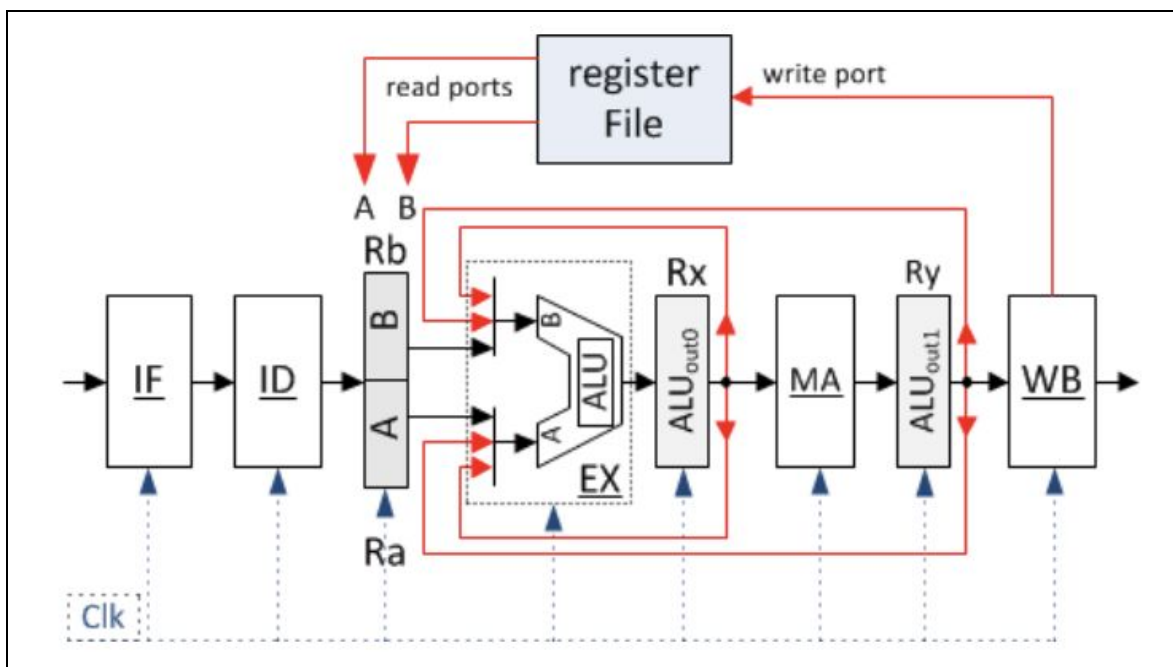
$r1 = r2 + r3$       [ADD]  
 $r4 = r1 - r5$       [SUB]

The second instruction must wait for the first to complete and be written to memory before it can be safely used. If this is not accounted for the second instruction will be executed in the pipeline before the desired result is written to r1 from instruction 1.



This can be resolved by using a pipeline which forwards/bypasses writing to the register file and immediately passes the result of instruction 1 back to the ALU to be used in instruction 2.

### Pipeline Forwarding



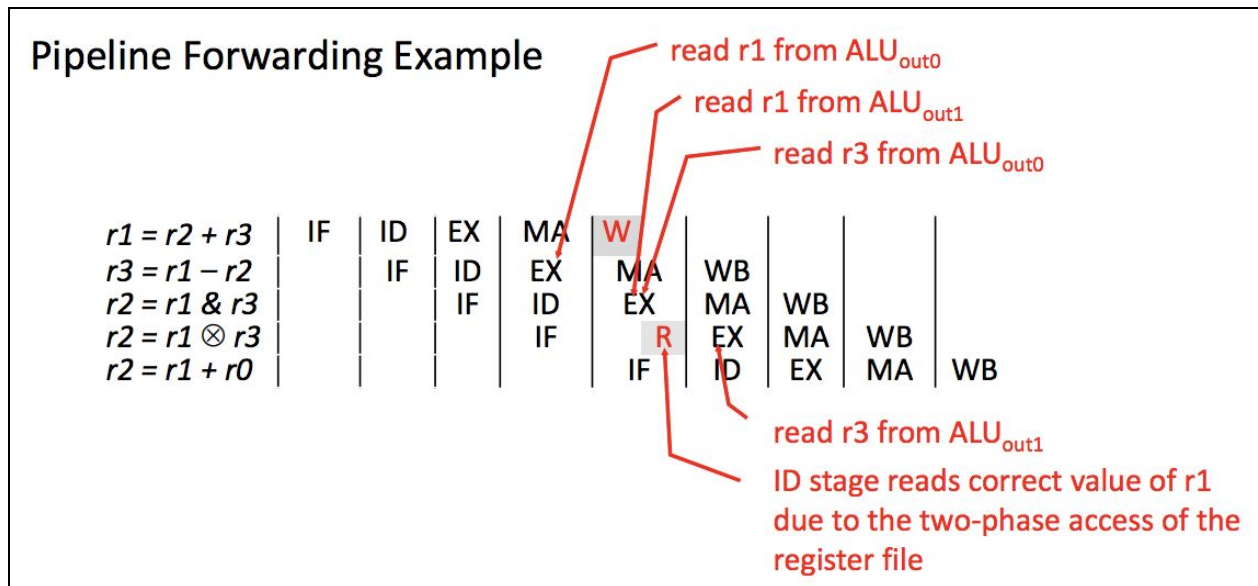
The ALU results from the “previous” two instructions can be forwarded to the ALU inputs as seen in the above diagram.

ALU<sub>out0</sub> and ALU<sub>out1</sub> are tagged as the destination register from each operation respectively.

The EX stage of the pipeline checks for source register of the instruction in the order:

1. ALU<sub>out0</sub>
2. ALU<sub>out1</sub>
3. A/B

### Pipeline Forwarding Example




The first instruction writes to r1 and the next 4 instructions use r1 as a source operand and as a result access this via ALU<sub>out0</sub> and ALU<sub>out1</sub>.

Second instruction writes to r3 which is used as a source operand by the third and fourth instructions and as a result are also accessed via ALU<sub>out0</sub> and ALU<sub>out1</sub>.

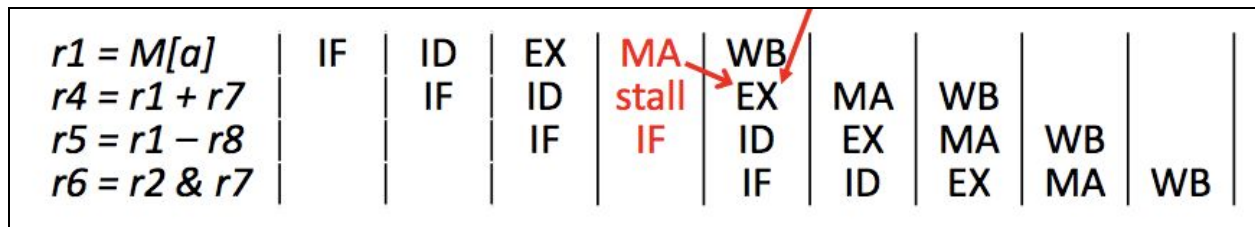
iii) What are load hazards? Explain why load hazards arise in the 5 stage DLX pipeline and its effect in terms of stall cycles. How can you avoid such load hazards?

### Load Hazards

When values are loaded from memory into a given register and are then used in the next instruction it produces a load hazard as the processor must stall for 1 cycle while the value is loaded into the register. Consider the following instructions:

	<code>r1 = M[a]</code>	<code>// load</code>
	<code>r4 = r1 + r7</code>	<code>// add</code>
	<code>r5 = r1 - r8</code>	<code>// subtract</code>
	<code>r6 = r2 &amp; r7</code>	<code>// and</code>

Here r1 is used on instruction 2 and as a result instruction 2 is dependent on the resulting value in r1 and must wait for it to be loaded.



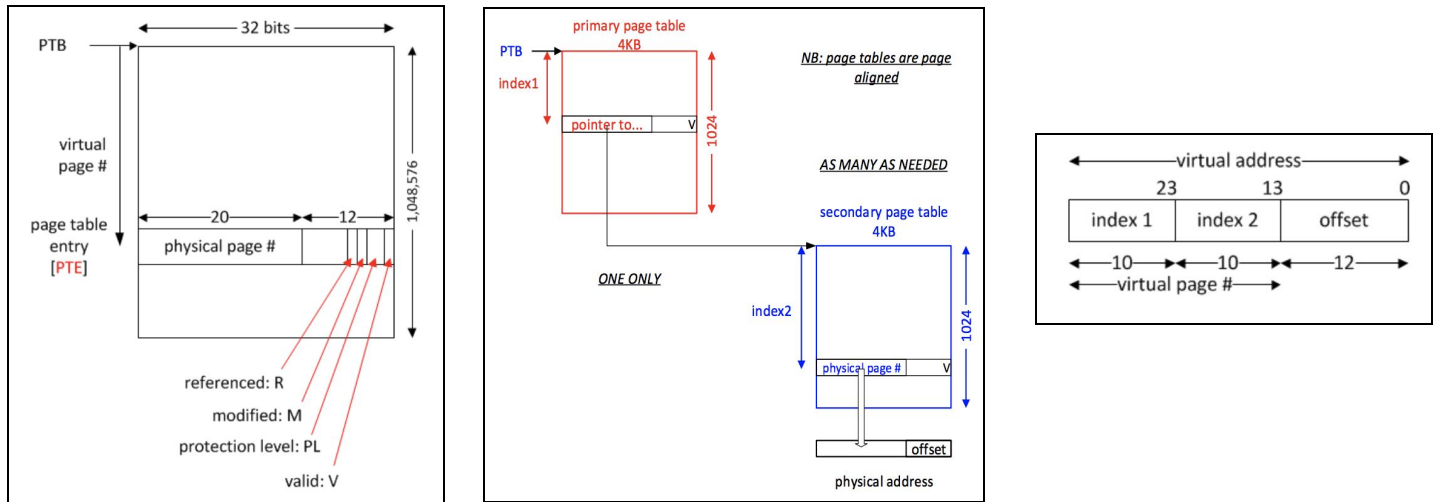
To avoid this it is often possible to reschedule the instruction causing the delay. For example with the above program instruction 4 can be swapped with instruction 2 and then there will be no data dependency issues and thus no stall.

## Question 2

i) How are virtual addresses converted into physical addresses by a two level page table? Assume a page table structure as per an IA32 based CPU. Draw diagrams to illustrate your answer.

The Memory Management Unit (MMU) converts virtual addresses generated by the CPU into physical addresses in memory using page tables. Address space is divided into fixed sized pages. Each process runs in its own 4GB virtual address space which has pages mapped by the MMU to memory.

Virtual page numbers are converted to physical page numbers using a page-table lookup. The virtual page number is used as an index into a page table stored in physical memory. The page table base register (PTB) contains the physical address of the page table of the current process. 4MB physical memory needed to hold the page table of every process



Using the virtual address and the primary page (`primarytable[index1]`) table we can find the address of the secondary page table and from this can get the physical address (`secondarytable[index2]`) of the data in memory.

*ii) What is the advantage of using a two level page table compared with a single level page table? Comment on the amount of memory needed for a small process and a maximum sized process using a single and two level page table structure.*

To reduce the size of the page table structure an n-level look up table can be used. This means that the larger the process, the more memory is needed for its tables. However, the smaller the process the less memory that is needed.

Each process has one 4MB primary page table and multiple 4KB secondary, tertiary... page tables. Secondary page tables are created on demand as a process' size exceeds that of the capability of a single secondary table and needs more address spaces. This way, small

---

processes never over-use memory and memory is utilized efficiently for both small processes and maximum sized processes.

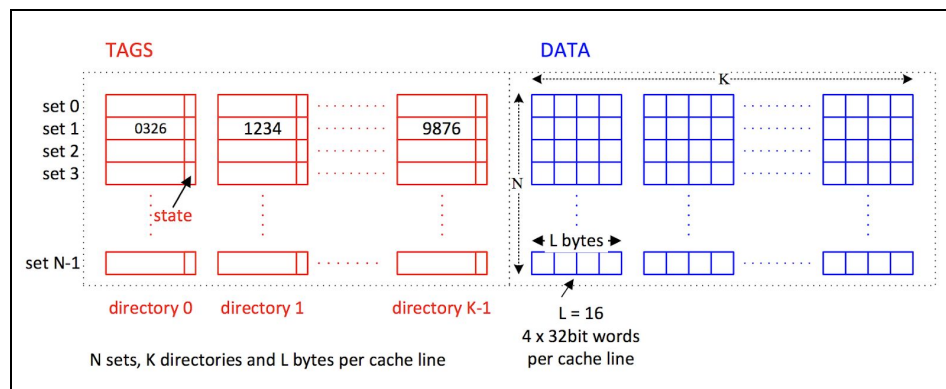
A process in a 32 bit system can theoretically use each of the possible 1 million 20 bit page locations in its table if necessary. A single 32-bit page table therefore would need 4MB of RAM (32bit = 4 bytes, 4 bytes \* 1 million rows = 4MB). A two level page table is far more memory usage efficient as the virtual page address is split into two addresses (index1 - primary table index points to secondary page table AND index2 - secondary table index points to address in memory). This means that each primary and secondary table is now 1024 lines long. The same capabilities apply as the 1024 primary table addresses can map to 1024 secondary table addresses ( $1024^2 = 1$  million). More secondary tables after the first one only need to be created as they are required thus saving space in RAM.

	min process (i)	max process (ii)
1 level page table	4MB	4MB
2 level page table	4KB + 4KB (1 primary level table always present, smallest process will need at least one secondary)	4KB + 4MB (1 primary level table + 1024 secondary level tables)

*ii) A user process is created which has 0x7000 bytes of code, 0x1678 bytes of initialised data, 0x2888 bytes of uninitialised data and 648 bytes of stack data copied from its parent. Draw a diagram that shows the structure and size of the 2-level page table which is initially created for the process. What size (in pages) is the initial memory footprint of the process?*

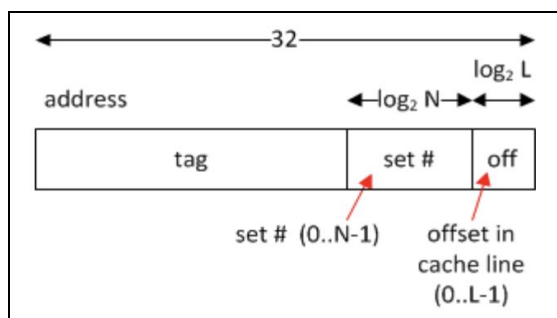
### Question 3

i) With the aid of a diagram explain how a cache organisation can be characterised by three constants, LKN. Explain in detail, how a data item is accessed in an LKN cache. Why does a cache normally reduce the effective memory access time? [6 marks]



- **N** sets of cache lines/tags
- **K** directories
- **L** bytes per cache line ( $L=16 \rightarrow 4 \times 32\text{bit words}$  per cache line)

When referencing/searching a K-Way cache, each cache address is mapped to a particular set (0-N).



[tag] [set #] [offset]

If an address maps to set 1, the set 1 tags of all K directories are compared with the incoming address tag simultaneously (fully associative).

If a match is found (**hit**), corresponding data returned offset within cache line. The K data lines in the set are also fully associative and accessed concurrently. If a match is not found (**miss**), read data from memory, place in cache line within set and update corresponding



cache tag (choice of K positions). This replacement depends on the cache line replacement strategy - Least Recently Used (LRU)

*ii) Compute the number of hits and misses if the following list of hexadecimal addresses is applied to (1) a 64 byte direct mapped cache with 16 bytes per line and (2) a 64 byte fully associative cache with 16 bytes per line.*

**(1)**  $L \times K \times N = 64$

$L = 16$

$K = 1$

$N = 4$

**Can ignore first 00 since all the same and the last 0 since don't care about offset**

HEX	BINARY	TAG	SET	OFFSET	HIT/MISS	CACHED AT
0x0000	00000000	00	00	-	MISS	N = 0
0x0010	00010000	00	01	-	MISS	N = 1
0x0020	00100000	00	10	-	MISS	N = 2
0x0030	00110000	00	11	-	MISS	N = 3
0x003C	00111100	00	11	-	HIT	N = 3
0x0050	01010000	01	01	-	MISS	N = 1
0x0010	00010000	00	01	-	MISS	N = 1
0x000C	00001100	00	00	-	HIT	N = 0
0x0010	00010000	00	01	-	HIT	N = 1
0x0050	01010000	01	01	-	MISS	N = 1
0x002C	00101100	00	10	-	HIT	N = 2
0x0010	00011000	00	01	-	MISS	N = 1
0x0050	01010000	01	01	-	MISS	N = 1
0x0020	00100000	00	10	-	HIT	N = 2
0x0010	00010000	00	01	-	MISS	N = 1
0x0000	00000000	00	00	-	HIT	N = 0

---

(2)  $L \times K \times N = 64$

$L = 16$

$K = 4$

$N = 1$

**Since  $N = 1$ , set always 0**

HEX	BINARY	TAG	SET	OFFSET	HIT/MISS	REPLACING LRU
0x0000	00000000	0000	0	-	MISS	
0x0010	00010000	0001	0	-	MISS	
0x0020	00100000	0010	0	-	MISS	
0x0030	00110000	0011	0	-	MISS	
0x003C	00110100	0011	0	-	HIT	
0x0050	01010000	0101	0	-	MISS	0000
0x0010	00010000	0001	0	-	HIT	
0x000C	00001100	0000	0	-	MISS	0010
0x0010	00010000	0001	0	-	HIT	
0x0050	01010000	0101	0	-	HIT	
0x002C	00101100	0010	0	-	MISS	0011
0x0010	00011000	0001	0	-	HIT	
0x0050	01010000	0101	0	-	HIT	
0x0020	00100000	0010	0	-	HIT	
0x0010	00010000	0001	0	-	HIT	
0x0000	00000000	0000	0	-	HIT	

LRU = 0000 / 0001 / 0010 / 0011 / 0101 / 0001 / 0000 / 0001 / 0101 / 0010 / 0001 / 0101 / 0010 / 0001 / 0000

---

iii) Would you expect an associative cache of size  $N$  and line size  $L$  to always have a higher hit rate than a direct mapped cache of size  $N$  and line size  $L$ . Explain the reasoning behind your answer.

Consider two caches:

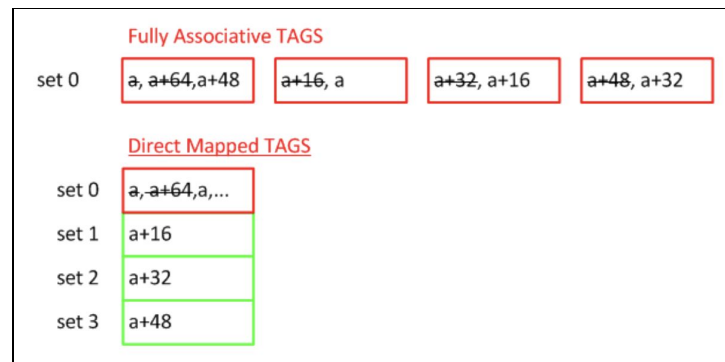
- $K = 4, N = 1, L = 16$  [64 byte fully associative]
- $K = 1, N = 4, L = 16$  [64 byte direct mapped]

$L * K * N$  all equal

Consider the following address sequences repeatedly querying cache:

$[a], [a+16], [a+32], [a+48], [a+64] \rightarrow 5 \text{ addresses}$

Our caches can contain 4 addresses, sequence comprises of 4



Fully Associative: Only 4 addresses can fit in the 4-way cache so, due to the LRU replacement policy, every access will be a miss.

Direct Mapped: Since ONLY addresses  $[a]$  and  $[a+64]$  will conflict each other as they map to the same set, there will be 2 misses and 3 hits per cycle of 5 addresses.

---

## Question 4

*i) What is the cache coherency problem? Briefly explain the states and operation of the Firefly cache coherency protocol.*

The cache coherency problem is an issue regarding cache's and physical memory where a CPU might make changes to a value in cache. This change must be propagated to both physical memory and any other cache's that have access to this same shared address/value.

### **Firefly Protocol (Vivio)**

- Caches know if its cache lines are shared with other caches
- If cache line read other caches will assert a common SHARED bus line if they contain the same line
- Writes to !SHARED lines are write-back
- Writes to SHARED lines are write-through, other caches & memory will be updated
- When cache line no longer shared, it needs extra write-through cycle to find out that it is no longer shared
- At reset, a bootstrap program fills cache with addresses valid from memory

*ii) Describe the changes that occur in a three CPU multiprocessor system where each CPU has its own cache. Assume CPU 0, 1 and 2 all start with  $a0 = 0$  in set 1 and  $a1 = 0$  in set 2*

**t = 1** → CPU 0 reads  $a0$  which will be initially stored in its CPU and will not have to be fetched from memory. The shared flag remains the same and it is not dirty since it has not been modified.

**t = 2** → CPU 0 reads  $a2$  which is not present in set 1 and as a result will have to be read from memory. The value of  $a2$  is read from memory and the set is marked as NOT SHARED and NOT DIRTY.

**t = 3** → CPU 0 writes to  $a2$ . Since the value is not shared the value is marked as DIRTY and only changed within the CPU. The change is not transferred back to memory.

---

**t = 4** → CPU 0 writes to a2 once again. Similarly to above the value is only updated within the cache since it is NOT SHARED and DIRTY. No changes are propagated to other caches or memory.

**t = 5** → CPU 1 attempts to read a2. Since it is not in its CPU set's it must fetch from memory. CPU 0 sees that CPU 1 is attempting to read a value that is DIRTY within CPU 0. As a result it interrupts this read, passes the value to CPU 1. The flags in CPU 0 are then set to SHARED and NOT DIRTY similar to CPU1.

**t = 6** → CPU 2 now attempts to read a2. Since it is not in its CPU set's it must fetch from memory. CPU 0/1 see that CPU is attempting to read a value that it has in its local CPU. As a result it interrupts the read, passes the value to CPU 2. The flags in CPU 2 are then set to SHARED and NOT DIRTY.

**t = 7** → CPU 0 now attempts to write to a2. Since it is within its CPU set there is no need to read from memory. **BUT since it is shared with multiple registers, the write must be propagated to these registers and also updated within memory.**

**t = 8** → CPU 0 attempts to write to a2 again. Since it is within its CPU set there is no need to read from memory. Once again, these changes are propagated to the other CPU's and memory. The flags remain as SHARED and NOT DIRTY.

**t = 9** → CPU 0 attempts to read a0. Since a2 is the value in set 1 CPU 0 must fetch the value of a0 from memory. This is fetched and the flags in CPU 0 are set to NOT SHARED and NOT DIRTY.

**t = 10** → CPU 1 now attempts to read a0. Since a2 is the value in set 1 CPU 1 must fetch the value of a0 from memory. This is fetched and the flags in CPU 1 are set to SHARED and NOT DIRTY. The flags in CPU 0 are also changed to SHARED and NOT DIRTY since it now shares a0 with CPU 1.

**t = 11** → CPU 2 attempts to write to a2. Since a2 is already present in CPU 2 no value needs to be fetched from memory. The flags for set 1 of CPU 2 still SHARED as it doesn't know the other CPU's have forgotten it. As a result the write operation is propagated to memory and over the SHARED bus. The flags in CPU 2 are now set to NOT SHARED and NOT DIRTY.

---

t = 12 → CPU 2 attempts to write to a2. Since a2 is already present in CPU 2 and it is NOT DIRTY and NOT SHARED, the write operation can be performed locally without any data being sent to memory or any other CPUs. However, as a result of this the flags are then changed to DIRTY.