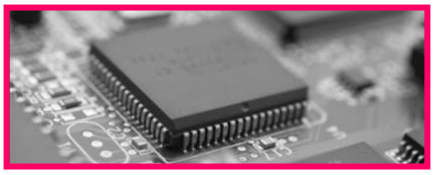
National and Kapodistrian University of Athens
Department of Informatics and Telecommunications
COURSE: **COMPUTER ARCHITECTURE II**



Handout 2 – Cache Memory

FALL SEMESTER 2021 – 2022

Objective

The goal of this project is to **design a "family" of three microprocessors** that differ in performance and cost for the same computing task. The processors will be designed and evaluated in the QtMips simulator.

The computing task consists of two distinct steps (sort and search):

(a) An initially unsorted array of 40 000 positive integers (n[1], … , n[40 000]) of size 1 byte each is to be sorted in ascending order. A check must be made that the numbers are indeed positive and the program must stop if a negative number or zero is found. The array resides in the data section of your program and no I/O is required. The sorting should use the quicksort algorithm.

(b) The sorted array of 1-byte integers is searched for a specific number (also found in the data section of your program). The search should use the binary search algorithm. If the number is found then the first element of the table should contain its position (1 to 40 000). If the number is not found then the first element of the array should have the value 0.

The three processors share a common design at the core of the pipeline: they have a 2-bit branch predictor with a 5-bit BHT accessed and branch resolution in the EX stage and a full feed-forward hazard unit. They differ in the memory system.

The **first processor** (codename: turtle) is a cheap version of the family with the minimum possible cost, which means that it does not use a cache. Main memory access takes 40 clock cycles. The cost of the processor is 20 euros and its clock rate is 500MHz.

The **second processor** (codename: rabbit) is an average version of the family that uses only one cache level (L1) for instructions and data. Accessing main memory in this processor also takes 40 cycles. The instruction/program cache (L1 program cache in QtMips terminology) is 8KB in size and can contain 4, 8, 16, or 32 words per block. You can choose any associativity and any replacement policy you wish. The data cache (L1 data cache) can be between 4, 8, or 16KB in size and can contain 4, 8, 16, or 32 words per block. You can again choose any associativity and any replacement policy you wish. The write and allocation policy must be write back and write allocate. The cost of this processor is 20, 25, or 30 euros higher than the previous one if the data cache is 4, 8, or 16KB in size, respectively. The clock rate is 500 MHz if the correlation is 1 (direct mapped) but is reduced by 10 MHz for each doubling of the correlation.

The **third processor** (codename: puma) is an advanced version of the family that uses two levels of cache memory (separate L1 for instructions and data and a single L2). Main memory access in this processor also takes 40 cycles and L2 access takes 5 cycles. The L1 caches are the same as the previous processor (you'll keep the design you came up with – hence the cost and clock rate). The L2 cache is 16, 32, or 64 KB in size and has the same block size as the previous processor's L1 data cache. You can choose any associativity and replacement policy you want for L2. The write and allocation policy must be write back and write allocate. The cost of this processor is 50, 75, or 100 euros higher than the previous one if the L2 cache is 16, 32, or 64 KB in size, respectively. The clock rate is equal to that of the previous processor.

- Complete the table below with the **attributes of your design** for each of the three processors.
- Measure the **execution time** of your program in your three designs using the number of clock cycles from QtMips and the clock rate of your design. If you use different data sets describe how you get your measurements (you can use average values for example).
- Calculate the **performance-to-cost ratio** of your program in all three of your designs (as you calculated in the first project).

Attribute	First Processor (*turtle*)	Second Processor (*rabbit*)	Third Processor (*puma*)
L1 Program Cache (Size, Block Size, Associativity, Replacement Policy)	-	Size: 8KB Block Size: 32 Direct Mapped LRU	Μέγεθος: 8KB Block Size: 32 Direct Mapped LRU
L1 Data Cache (Size, Block Size, Associativity, Replacement Policy)	-	Μέγεθος: 4KB Block Size: 16 Two-way set LRU	Μέγεθος: 4KB Block Size: 16 Two-way set LRU
L2 Cache (Size, Block Size, Associativity, Replacement Policy)	-	-	Μέγεθος: 16KB Block Size: 32 Direct Mapped LRU
Cost	20 €	40 €	90 €
Frequency	500MHz	490MHz	490MHz
Cycles	505.245.286	11.253.032	11.166.757
Execution Time	1,01049 seconds	0,02296 seconds	0,02278 seconds
Cost Performance	0,04948	1,089	0,487

Implementation (Quicksort & Binary Search)

Initially, I check if there is any number in the array that is zero or negative to stop the execution of the program as is required. By having this check here instead of every time during the partition it saves cycles. In Quicksort I initially allocate space on the stack to hold the array of pointers that I will need for each recursive call. Then I partition the array and recursively call the quicksort for the two parts of the table before and after the pivot. At the end of the process, that is, as soon as the high index becomes smaller than low, I restore the values of the variables I had on the stack and return to the point where quicksort was called.

In the partition, I want the leftmost element to be the pivot and the "i" pointer that initially points to low should be increased until it encounters an element of the array greater than or equal to the pivot and correspondingly the "j" pointer should be decreased until it finds an element smaller than or equal to the pivot. Then it goes to the reversi label which simply reverses them. Otherwise, it continues this process until "i" becomes greater than "j". If that happens I simply reverse the "j" element with the pivot.

When the Quicksort is finished, then for the Binary Search I keep the first and last index of the element of the array. In this way and using shift right logical for division by two I go every time to the middle element (which will be middle = left + (right - left) / 2) and check if it is equal to the element I am looking for. If it is, I place it in the place of the memory I have allocated for it. If it's smaller, I make the left index equal to the middle one + 1 and if it's bigger, the right index is correspondingly smaller by 1 than the middle one.

Testing and Measurements for the First Processor (Turtle)

At first, I tested and improved the above program till the point that it worked with as few cycles as possible for the array of 40,000 integers and at the same time didn't require a lot of space on the stack, since swapping elements should be done "in place". Testing without pipeline to see that it runs correctly by looking at the memory section it took 11,090,752 cycles (for the array in the code) with a CPI of 1.17202. Also, with an ideal memory and pipelined with a 2-bit branch predictor with a 5-bit accessed BHT and branch resolution in the EX stage and a forwarding hazard unit it required 11 million cycles (to run 9 million commands).

With the main memory access taking 40 cycles as requested for the Turtle (as in the rest of the processors) I report the measurements I have obtained with the particular 40,000 position array I have in the code.

Cycles	CPI	Instructions	RAM Stalls	Control Hazard Stalls	Data Hazard Stalls
505.245.286	53.392	9.462.932	494.154.534	706.185	921.635

For the first processor the program took, as expected, a lot of time to complete its execution. After I verified that the results were correct in the memory section with the requested pipeline design, I calculated the execution time as well as the performance to cost ratio.

Time = Cycles * Seconds/Cycle = 505,245,286 * 2 * (10^-9) = 1.01049 seconds

Performance = 1 / (Time * Cost) = 1 / (1.01049 * 20) = 0.04948

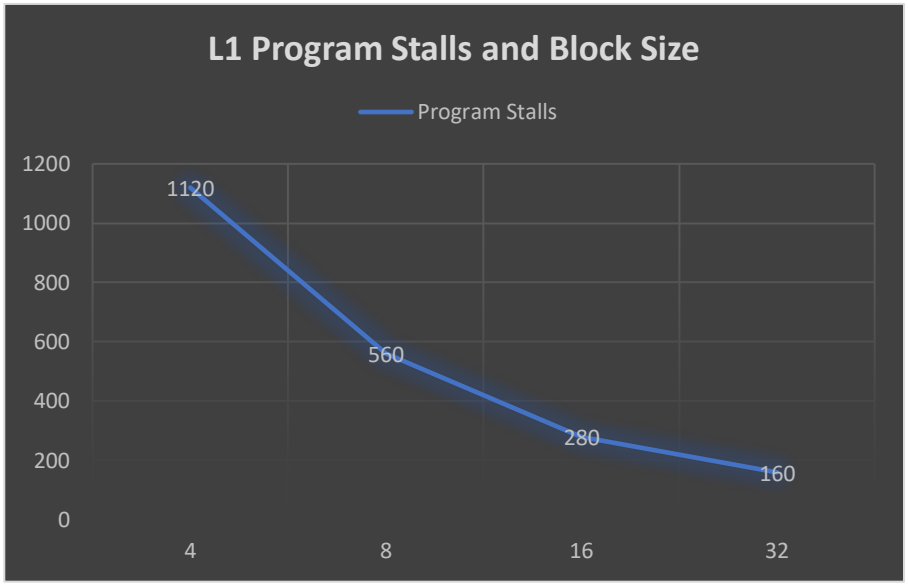
Parameter Selection for the Second Processor (Rabbit)

For a reduced page fault rate, I considered that the LRU replacement policy is optimal for the data cache. This is because of the way partition works. The integers that have been already examined are the ones that will not be needed later on, so replacing the least recently used page is best in this program.

For the instruction cache I thought it wise to have a large block size to reduce the miss rate, since the same instructions will be executed over and over again. Also with a direct mapped associativity since they are not replaced, due to the fact that the same program is always executed. So with 4 possible block sizes:

Block Size	L1 Program Stalls	L1 Program Hit	L1 Program Miss
4	1120	10.043.286	28
8	560	10.043.372	14
16	280	10.043.379	7
32	160	10.043.386	4

After the program is in the L1 data cache then the misses stop and the hit rate is 100% for all. So a better evaluation to compare is going to be the program stalls.



Thus, it's apparent that a large block size is better in this particular cache. Since I will be fetching the instructions directly into memory (since this requires 40 cycles) I chose 32 for the block size of the L1 program cache. So before adding the data cache I have 103.509.446 cycles, with CPI 10.9384 and ram stalls 92.418.534 (which of course are due to the absence of L1 data cache).

Using Quicksort, it will be appropriate when it brings words from the data to the cache to also bring the adjacent ones since they will be examined later. In Binary Search this is not useful, but since it is fast and finishes in O(logn) time compared to O(nlogn) of Quicksort, it is not so important. So keeping the settings of the L1 program cache that I chose before (32block size, direct mapping) for the first tests of the L1 data cache with direct mapping and a block size of 32 for the 4 possible sizes of the L1 data cache, I got:

L1 Data Cache Parameters	Cycles	CPI	L1 Data Stalls	L1 Data Hit	L1 Data Miss (& Low memory Reads)	L1 Low Memory Writes	L1 Hit Rate
Size: 4KB **Block Size: 32** **Direct Mapping**	11.689.152	1.23526	598.240	2.797.140	14.956	5.968	99.468%
Size: 8KB **Block Size: 32** **Direct Mapping**	11.359.232	1.20039	921.635	2.805.388	6708	3.297	99.761%
Size: 16KB **Block Size: 32** **Direct Mapping**	11.104.500	1.17348	12.720	2.811.772	318	318	99.989%

The instructions are 9,462,932 and increasing the size of the data cache predictably reduces the cycles required to execute them. To examine whether this decrease in execution time is worth the increase in cost, I will compare the cost performance ratio with these settings. For 500MHz (meaning that 2 nanoseconds are required for each cycle) I have:

PerformanceCost4KB = 1 / (11,689,152 * 2 * (10^-9) * 40) = 1.0694

PerformanceCost8KB = 1 / (11,359,232* 2 * (10^-9) * 45) = 0.978

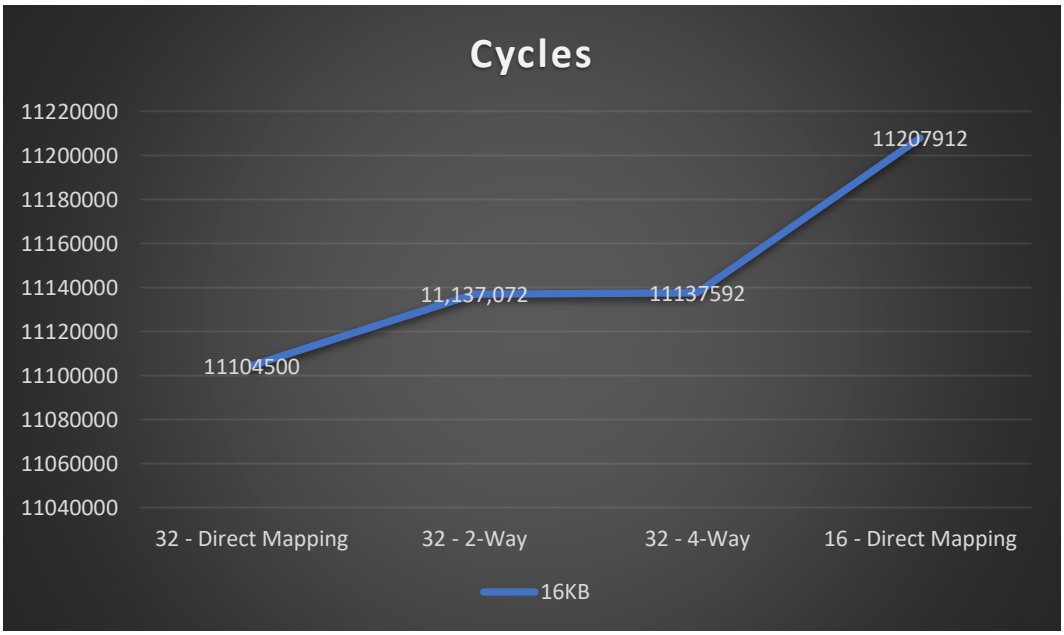
PerformanceCost16KB = 1 / (11,104,500 * 2 * (10^-9) * 50) = 0.9005

Although it requires fewer cycles, the cache with 16KB is not so much faster as to justify the increased cost. So, the one that will give me the best performance for cost is the 4KB cache. However, the fact that the one with 16KB is faster can be a good basis to select the settings for the 4KB cache I will use.

This way I can decide on the compromises I don't need to make. To compare I'll take measurements with a decrease in block size while keeping the direct mapping and also with an increase in the set associativity while keeping the block size constant:

L1 Data Cache Parameters	Cycles	CPI	L1 Data Stalls	L1 Data Hit	L1 Data Miss (& Low memory Reads)	L1 Low Memory Writes	L1 Hit Rate
Size: 16KB **Block Size: 32** **2-Way Set Associative**	11.137.072	1.17692	46.160	2.810.942	1.154	879	99.959%
Size: 16KB **Block Size: 32** **4-Way Set Associative**	11.137.592	1.17697	46.680	2.810.929	1.167	889	99.959%
Size: 16KB **Block Size: 32** **Direct Mapping**	11.104.500	1.17348	12.720	2.811.772	318	318	99.989%
Size: 16KB **Block Size: 16** **Direct Mapping**	11.207912	1.1844	117.000	2.809.171	2925	2134	99.896%

By keeping the block size at 32 and having 2-way associativity and then 4-way I saw no improvement in time. While reducing the block size with direct mapping display the cycles increased.



Decreasing the block size of the data cache predictably brings an increase in required cycles, if the associativity does not double at the same time.

With that in mind, for the next metrics I tested for a 4KB memory with 2 or 4 way associativity for lower block size than 32. If the block size is too small, then it won't be efficient. I measured with a block size of 4 for comparison, but mostly focused on block size 16 with different associativity. Finally, once I found the ideal settings for 4KB for a comparison base, I did the corresponding tests with these parameters for 8KB and 4KB:

L1 Data Cache Parameters	Cycles	CPI	L1 Data Stalls	L1 Data Hit	L1 Data Miss (& Low memory Reads)	L1 Low Memory Writes	L1 Hit Rate
Size: 4KB Block Size: 4 Direct Mapping	11.849.672	1.25222	758.760	2.793.127	18.969	15.054	99,325%
Size: 4KB Block Size: 32 Direct Mapping	11.689.152	1.23526	598.240	2.797.140	14.956	5.968	99,468%
Size: 4KB Block Size: 8 2-Way Set	11.412.032	1.20597	321.120	2.804.068	8.028	6.803	99,715%
Size: 4KB Block Size: 16 2-Way Set	11.253.032	1.18917	162.120	2.808.043	4.053	3.441	99.856%
Size: 4KB Block Size: 8 4-Way Set	11.414.672	1.20625	323.760	2.804.002	8.094	6.860	99.712%
Size: 4KB Block Size: 16 Direct Mapping	11.507.992	1.21611	417.080	2.801.669	10.427	5.758	99.629%
Size: 4KB Block Size: 16 4-Way Set	11.183.752	1.18185	92.840	2.809.775	2.321	1.769	99.917%
Size: 16KB Block Size: 16 2-Way Set	11.182.872	1.18176	91.960	2.809.797	2.299	1.751	99.918%
Size: 8KB Block Size: 16 2-Way Set	11.213.752	1.18502	122.840	2.809.025	3.071	2.480	99.891%

As expected decreasing block size greatly increases execution time, while increasing associativity after a point also increases cycles (while increasing access time). So the ideal block size I found after the measurements is 16 words. And it runs in fewer cycles with 4-way set associativity, but even 2-way is close enough. And since doubling the associativity lowers the clock rate by 10 MHz I'll compare the cost performance ratio for 4KB with a block size of 16 to see which one is better as well as for 8 and 16 KB with the same settings:

L1 Data Cache Parameters	Cycles	Time	Cost	Performance -to-Cost
Size: 4KB **Block Size: 16** **2-Way Set**	11.253.032	0.02296	40	1.089
Size: 4KB **Block Size: 16** **Direct Mapping**	11.507.992	0.02302	40	1.086
Size: 4KB **Block Size: 16** **4-Way Set**	11.183.752	0.02326	40	1.074
Size: 16KB **Block Size: 16** **2-Way Set**	11.182.872	0.02281	50	0.8767

As can be seen from the results of the table above I have the best performance to cost with the 4KB cache with a block size of 16 when the set associativity is 2-way. And in fact the CPI I measured 1.18917 is close to the measurement I made with ideal memory which was 1.172.

So the parameters for the Rabbit processor will be:
Frequency: 490MHz

L1 Program Cache
Size: 8KB
Sets: 64
Block Size: 32 words
Associativity: Direct Mapped

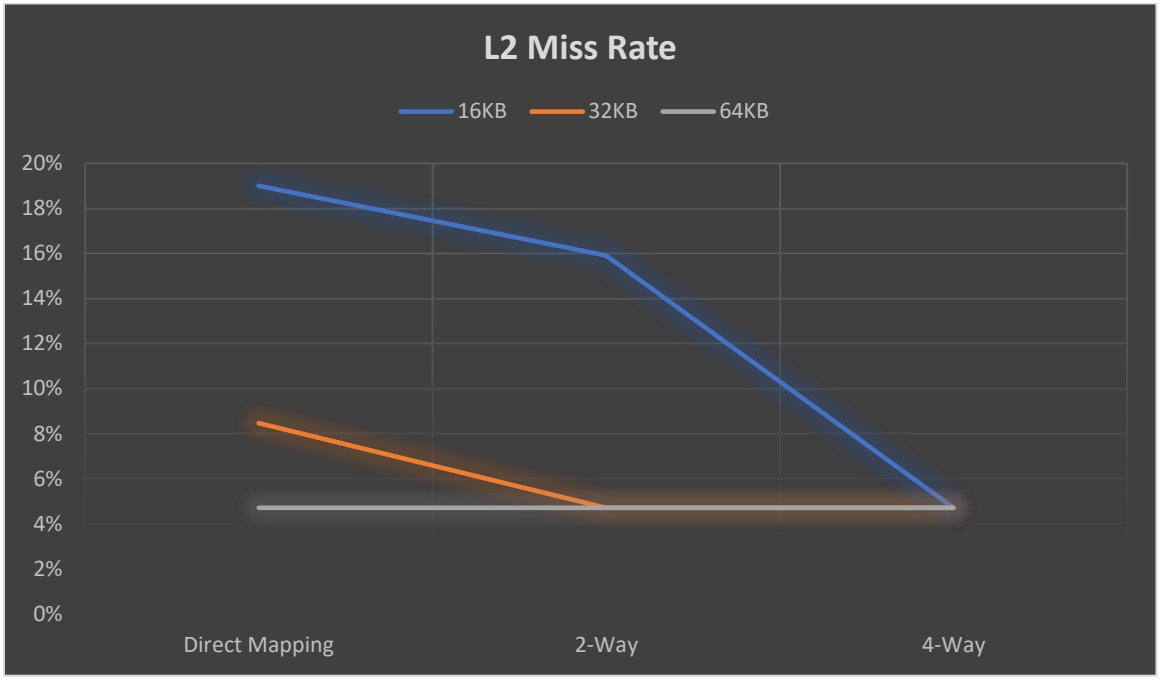
L1 Data Cache
Size: 4KB
Sets: 32
Block Size: 16 words
Associativity: 2 – way

Parameter Selection for the Third Processor (Puma)

The L1 cache is a small and fast memory, while L2 must be large, even if it is slow. To verify this and find the optimal parameters I'll keep the previous settings for L1 data cache and L1 program cache and I will first test the three sizes with direct mapping and then with 2 and 4 way set associativity:

L2 Parameters	Cycles	CPI	L2 Stalls	L2 Hit Rate	Time	Cost	Performance / Cost
Size: 16KB Block Size: 32 Direct Mapping	11.166.757	1.17942	50.640	80.995%	0.02278	90	0.487
Size: 32KB Block Size: 32 Direct Mapping	11.135.157	1.17658	23.080	91.531%	0.02272	115	0.383
Size: 64KB Block Size: 32 Direct Mapping	11.123.877	1.17552	12.840	95.292%	0.0227	140	0.314
Size: 16KB Block Size: 32 2-Way	11.157.517	1.17907	46.400	84.076%	0.0232	90	0.478
Size: 32KB Block Size: 32 2-Way	11.123.877	1.17552	12.840	95.292%	0.0231	115	0.376
Size: 64KB Block Size: 32 2-Way	11.123.877	1.17552	12.840	95.292%	0.0231	140	0.308
Size: 16KB Block Size: 32 4-Way	11.123.877	1.17552	12.840	95.292%	0.0235	90	0.471
Size: 32KB Block Size: 32 4-Way	11.123.877	1.17552	12.840	95.292%	0.0235	115	0.368
Size: 64KB Block Size: 32 4-Way	11.123.877	1.17552	12.840	95.292%	0.0235	140	0.302

It is obvious that in all measurements for the three sizes the improvements with the L2 cache make the processor have a CPI very close to that of the ideal memory (which is 1.172). Also while the jump in performance from Turtle to Rabbit was big, the jump to the third Puma processor is much smaller.



The reduction in cycles is also small as the associativity is doubled for the 16KB cache, while in the case of the 32KB there is no change. For this reason, for the L2 cache I will again choose the smallest memory available, i.e. 16KB, which, while it has similar cycles to the other two sizes, is much more economical and has the best performance-to-cost ratio. Also because the 10MHz clock rate penalty from increasing associativity makes the program run slower even though the required cycles are reduced, I will choose direct-mapped associativity.

With the addition of the L2 cache the cycles are close to that of an ideal memory, but not much different to the second processor. This small improvement in execution time (0.02278 vs. 0.02296 seconds) combined with the large increase in cost results in a worse cost performance ratio than before (0.487 vs. 1.089). However, this reflects the reality that after a point small increases in speed, cost much more. Nonetheless, these processors are intended for systems where cost is not a factor and performance is more important.

So the parameters for the Puma processor will be:

Frequency: 490MHz

L1 Program Cache

Size: 8KB

Sets: 64

Block Size: 32 words

Associativity: Direct Mapped

L1 Data Cache

Size: 4KB

Sets: 32

Block Size: 16 words

Associativity: 2 – way

L2 Unified Cache

Size: 16KB

Sets: 128

Block Size: 32 words

Associativity: Direct Mapped