Jose Franco Baquera

April 30, 2019

Assignment 5

CS 473: Architectural Concepts I

1. The following table provides parameters for different caches. C is the cache size (bytes), B is the block size (bytes) and E is the associativity. Assume a 32-bit address space. For each cache, calculate the number of cache sets (S), number of tag bits (t), number of set index bits (s) and number of block offset bits (b).

Cache	C	B	E	S	t	s	b
1	1024	4	4	-	-	-	-
2	1024	4	256	-	-	-	-
3	1024	8	1	-	-	-	-
4	1024	8	128	-	-	-	-

Work Shown for Cache 1:

We know the following: Cache Size = 1024 Bytes
Block Size = 4 Bytes
E (Associativity/Number of Lines per Set) = 4
Address Space = 32 bits

Finding S:

We can use the equation C = S x E x B => S = C/(E x B)
=> S = 1024 ~~Bytes~~/(4 ~~Bytes~~ x 4)
=> S = 1024/16 => S = **64**

Finding t:

We note that the address space is 32 bits. Therefore, the number of tag field bits is equal to 32 – s – b = 32 – 6 – 2 = 32 – 8 = **24**

Finding s (number of set index bits):

The cache is 64 = 2^6 Blocks. Therefore, the total number of set index bits is 6. Another way to find this is by the following: s = $\log_2(64)$ = **6**

Finding b (block offset bits):

Block size = 4 Bytes = 2^2 Bytes. Therefore, the byte offset is 2 bits. Another way to find this is by the following: b = $\log_2(4)$ = **2**

Work Shown for Cache 2:

We know the following: Cache Size = 1024 Bytes
Block Size = 4 Bytes
E (Associativity/Number of Lines per Set) = 256
Address Space = 32 bits

Finding S:

We can use the equation C = S x E x B => S = C/(E x B)

=> S = 1024 ~~Bytes~~/(4 ~~Bytes~~ x 256)
=> S = 1024/1024 => S = **1**

Finding t:
We note that the address space is 32 bits. Therefore, the number of tag field bits is equal to 32 – s – b = 32 – 0 – 2 = 32 – 2 = **30**

Finding s (number of set index bits):
The cache is 1 = 2^0 Blocks. Therefore, the total number of set index bits is 0. Another way to find this is by the following: s = $\log_2(1)$ = **0**

Finding b (block offset bits):
Block size = 4 Bytes = 2^2 Bytes. Therefore, the byte offset is 2 bits. Another way to find this is by the following: b = $\log_2(4)$ = **2**

Work Shown for Cache 3:

We know the following: Cache Size = 1024 Bytes
Block Size = 8 Bytes
E (Associativity/Number of Lines per Set) = 1
Address Space = 32 bits

Finding S:
We can use the equation C = S x E x B => S = C/(E x B)
=> S = 1024 ~~Bytes~~/(8 ~~Bytes~~ x 1)
=> S = 1024/8 => S = **128**

Finding t:
We note that the address space is 32 bits. Therefore, the number of tag field bits is equal to 32 – s – b = 32 – 7 – 3 = 32 – 10 = **22**

Finding s (number of set index bits):
The cache is 128 = 2^7 Blocks. Therefore, the total number of set index bits is 7. Another way to find this is by the following: s = $\log_2(128)$ = **7**

Finding b (block offset bits):
Block size = 8 Bytes = 2^3 Bytes. Therefore, the byte offset is 3 bits. Another way to find this is by the following: b = $\log_2(8)$ = **3**

Work Shown for Cache 4:

We know the following: Cache Size = 1024 Bytes
Block Size = 8 Bytes
E (Associativity/Number of Lines per Set) = 128
Address Space = 32 bits

Finding S:
We can use the equation C = S x E x B => S = C/(E x B)
=> S = 1024 ~~Bytes~~/(8 ~~Bytes~~ x 128)
=> S = 1024/1024 => S = **1**

Finding t:
We note that the address space is 32 bits. Therefore, the number of tag field bits is equal to 32 – s – b = 32 – 0 – 3 = 32 – 3 = **29**

Finding s (number of set index bits):
The cache is 1= 2^0 Blocks. Therefore, the total number of set index bits is 0. Another way to find this is by the following: s = $\log_2(1)$ = **0**

Finding b (block offset bits):

Block size = 8 Bytes = 2^3 Bytes. Therefore, the byte offset is 3 bits. Another way to find this is by the following: b = $\log_2(8)$ = **3**

Final Answer: Please Look at the Following Table

Cache	C	B	E	S	t	s	b
1	1024	4	4	**64**	**24**	**6**	**2**
2	1024	4	256	**1**	**30**	**0**	**2**
3	1024	8	1	**128**	**22**	**7**	**3**
4	1024	8	128	**1**	**29**	**0**	**3**

2. In general, cache access time is proportional to capacity. Assume that main memory accesses take 70 ns and that memory accesses are 36% of all instructions. The following table shows data for L1 caches attached to each of two processors, P1 and P2.

	L1 Size	L1 Miss Rate	L1 Hit Time
P1	2 KiB	8.0%	0.66 ns
P2	4 KiB	6.0%	0.90 ns

a. Assuming that the L1 hit time determines the cycle times for P1 and P2, what are their respective clock rates?

- **P1 Clock Rate**
  P1 Clock Rate = 1/Clock Cycle Time = 1/0.66 ns = 1.51515 GHz
  P1 Clock Rate = 1.51515 GHz

- **P2 Clock Rate:**
  P2 Clock Rate = 1/Clock Cycle Time = 1/0.90 ns = 1.11111 GHz
  P2 Clock Rate = 1.11111 GHz

b. What is the Average Memory Access Time for P1 and P2?

NOTE: According to the slides, AMAT = Hit time + Miss rate × Miss penalty

- **P1 Average Memory Access Time (AMAT)**
  P1 AMAT = L1 Hit Time + (L1 Miss Rate x Memory Access Time)
  = 0.66 ns + (0.08 x 70 ns) = 0.66 ns + 5.6 ns
  = 6.26 ns
  P1 Average Memory Access Time = 6.26 ns

- **P2 Average Memory Access Time (AMAT)**

  P2 AMAT = L1 Hit Time + (L1 Miss Rate x Memory Access Time)
  = 0.90 ns + (0.06 x 70 ns) = 0.90 ns + 4.2 ns
  = 5.1 ns

  P2 Average Memory Access Time = 5.1 ns

c. Assuming a base CPI of 1.0 without any memory stalls, what is the total CPI for P1 and P2? Which processor is faster?

- **Total CPI for P1**

  The total CPI for P1 can be computed with the following equation:
  Total CPI = Base CPI + (Miss Penalty *Miss Rate)+(Miss Penalty * Miss Rate * % of Memory Instructions)
  We note that Miss Penalty (cycles) = Memory Access Time / Hit Time and that Hit Time in this example determines the processor's clock cycle time. Therefore, the total CPI for P1 is equal to the following:
  = 1 + (70~~ns~~/0.66~~ns~~)*0.08 + (70~~ns~~/0.66~~ns~~)*0.08*.36
  = 1+ 8.484848485+ 3.054545455
  = 12.53959594

  Total CPI for P1 = 12.53959594

- **Total CPI for P2**

  The total CPI for P2 can be computed with the following equation:
  Total CPI = Base CPI + (Miss Penalty *Miss Rate)+(Miss Penalty * Miss Rate * % of Memory Instructions)
  We note that Miss Penalty (cycles) = Memory Access Time / Hit Time and that Hit Time in this example determines the processor's clock cycle time. Therefore, the total CPI for P2 is equal to the following:
  = 1 + (70~~ns~~/0.90~~ns~~)*0.06+(70~~ns~~/0.90~~ns~~)*0.06*.36
  = 1 +4.66666667+1.68
  = 7.3467

  Total CPI for P2 = 7.3467

- **Faster Processor**

  We note that processor P2 has a lower total CPI than P1. Therefore, P2 is the faster processor.

For the next three problems, we will consider the addition of an L2 cache to P1 to presumably make up for its limited L1 cache capacity. Use the L1 cache capacities and hit times from the previous table when solving these problems. The L2 miss rate indicated is its **local** miss rate.

L2 Size	L2 Miss Rate	L2 Hit Time
1 MiB	95%	5.62 ns

d. What is the AMAT for P1 with the addition of an L2 cache? Is the AMAT better or worse with the L2 cache?

To find the AMAT for P1 with the addition of an L2 cache, we can still use the fact that AMAT = Hit time + Miss rate × Miss penalty. For this particular problem, we need to modify this equation slightly. That is, the AMAT with the addition of an L2 cache is the following:
L1 Hit Time + L1 Miss rate*(L2 Hit time + L2 Miss rate*Memory Access Time)

AMAT = 0.66 ns + 0.08*(5.62 ns + 0.95*70 ns) = 0.66 ns + 0.08*72.12 ns
= 0.66 ns + 5.7696 ns
= 6.4296 ns

The AMAT of P1 with the addition of an L2 cache is 6.4296 ns, which is an increase from the original AMAT. Therefore, the AMAT of P1 is WORSE with the L2 cache.

e. Assuming a base CPI of 1.0 without any memory stalls, what is the total CPI for P1 with the addition of an L2 cache?

We need to find the total CPI for P1 with the addition of an L2 cache. We note that we originally used the equation Total CPI = Base CPI + Miss Penalty * Miss Rate + Miss Penalty * Miss Rate * % of Memory Instructions when finding the total CPI without the L2 cache. We can modify this equation to the following in order to take into account cache L2:
Total CPI =>
= BaseCPI + L1MissRate*L2HitCycles + L1MissRate*L2MissCycles+ %ofMemoryInstructions *L1MissRate *L2HitCycles + %ofMemoryInstructions*L1MissRate*L2MissCycles
= BaseCPI+L1MissRate(L2HitCycles +L2MissCycles)+%ofMemoryInstructions*L1MissRate*(L2HitCycles+L2MissCycles)
= 1 + 0.08*(5.62~~ns~~/0.66~~ns~~ + 0.95*(70~~ns~~/0.66~~ns~~)) + 0.36*0.08*(5.62~~ns~~/0.66~~ns~~ + 0.95*(70~~ns~~/0.66~~ns~~))
= 1 + 0.08*109.2727273 + 0.36*0.08*109.2727273
= 1 + 8.741818184+ 3.147054545
= 12.88887273
= (approx.) 12.89

The total CPI for P1 with the addition of an L2 cache = 12.89

f. Which processor is faster, now that P1 has an L2 cache? If P1 is faster, what miss rate would P2 need in its L1 cache to match P1's performance? If P2 is faster, what miss rate would P1 need in its L1 cache to match P2's performance?

We note the following:
Total CPI of P1 with L2 cache = 12.89
Total CPI of original P2 = 7.3467
Therefore, processor P2 is still faster.

Processor P2 is still faster.

Now we have to find the miss rate that P1 (with L2 cache) needs in its L1 cache to match P2's performance.

We note that P2 Total CPI Time = 7.3467
Now we note that the required miss rate for P1 (call it variable W) in its L1 cache to match P2's performance must satisfy the following equation:

$$(1 + W*109.2727273+0.36*W*109.2727273) <= 7.3467$$

It is important to note that the "left-hand side" of the equation was copied from part e) of this problem.
Using algebra, we get the following:
W <= (7.3467– 1)/(0.36*109.2727273+109.2727273)
W <= 0.0427068244
W <= 0.04271= 4.271%

The miss rate of L1's cache (i.e. W) in P1 should be at most 0.04271= 4.271% in order to match P2's performance.

3. The 16 × 16 integer matrix A is stored in memory in row-major order. Assume that a small fully associative cache stores 4 blocks and each block has 4 words. To summate the matrix elements by iterating row by row or column by column as the following two pieces of code. What are the respective cache misses?

```
//Row by row
int sum = 0;
for (int i=0; i<16; ++i)
    for (int j=0; j<16; ++j)
        sum += A[i][j];

//Column by column
int sum = 0;
for (int j=0; j<16; ++j)
    for (int i=0; i<16; ++i)
        sum += A[i][j];
```

__Row by Row Algorithm__

We assume that the cache is empty before the row by row algorithm begins. Furthermore, assume that the matrix is stored in "row-major" order. We note the following *"memory" matrix*. The red numbers represent misses while the blue numbers represent hits. Furthermore, green represents cache block 1, purple represents cache block 2, orange represents cache block 3, and yellow represents cache block 4.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48
49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64
65	66	67	68	69	70	71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90	91	92	93	94	95	96
97	98	99	100	101	102	103	104	105	106	107	108	109	110	111	112
113	114	115	116	117	118	119	120	121	122	123	124	125	126	127	128
129	130	131	132	133	134	135	136	137	138	139	140	141	142	143	144
145	146	147	148	149	150	151	152	153	154	155	156	157	158	159	160
161	162	163	164	165	166	167	168	169	170	171	172	173	174	175	176
177	178	179	180	181	182	183	184	185	186	187	188	189	190	191	192
193	194	195	196	197	198	199	200	201	202	203	204	205	206	207	208
209	210	211	212	213	214	215	216	217	218	219	220	221	222	223	224
225	226	227	228	229	230	231	232	233	234	235	236	237	238	239	240
241	242	243	244	245	246	247	248	249	250	251	252	253	254	255	256

For example, let us simulate the first iteration of the inner loop:

Cache:

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

We note that i = 0.
j = 0, 1, 2, 3 (miss, hit, hit, hit)
j = 4, 5, 6, 7 (miss, hit, hit, hit)
j = 8, 9, 10, 11 (miss, hit, hit, hit)
j = 12, 13, 14, 15 (miss, hit, hit, hit)

When i = 1, 2, 3,….,15 we will have a similar scenario. As the matrix demonstrates, there will be one miss for every four elements. Since there are 256 total elements and 1 miss per 4 new elements, then the respective total cache misses for the row by row summation equals to 256/4 = 64.

Row by row will have a total of 64 cache misses.

Column by Column Algorithm

We assume that the cache is empty before the column by column algorithm begins. Furthermore, assume that the matrix is still stored in "row-major" order. We note the following *"memory" matrix*. The red numbers represent misses while the blue numbers represent hits. Furthermore, green represents cache block 1, purple represents cache block 2, orange represents cache block 3, and yellow represents cache block 4.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48
49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64
65	66	67	68	69	70	71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90	91	92	93	94	95	96
97	98	99	100	101	102	103	104	105	106	107	108	109	110	111	112
113	114	115	116	117	118	119	120	121	122	123	124	125	126	127	128
129	130	131	132	133	134	135	136	137	138	139	140	141	142	143	144
145	146	147	148	149	150	151	152	153	154	155	156	157	158	159	160
161	162	163	164	165	166	167	168	169	170	171	172	173	174	175	176
177	178	179	180	181	182	183	184	185	186	187	188	189	190	191	192
193	194	195	196	197	198	199	200	201	202	203	204	205	206	207	208
209	210	211	212	213	214	215	216	217	218	219	220	221	222	223	224
225	226	227	228	229	230	231	232	233	234	235	236	237	238	239	240
241	242	243	244	245	246	247	248	249	250	251	252	253	254	255	256

For example, let us simulate the first iteration of the inner loop (Note: Numbers that are black in color were “irrelevant.” That is, they were not hits nor misses.):

Cache:

We note that j = 0.

~~1~~	~~2~~	~~3~~	~~4~~
~~17~~	~~18~~	~~19~~	~~20~~
~~33~~	~~34~~	~~35~~	~~36~~
~~49~~	~~50~~	~~51~~	~~52~~

i = 0 (miss)
i = 1 (miss)
i = 2 (miss)
i = 3 (miss)

~~65~~	~~66~~	~~67~~	~~68~~
~~81~~	~~82~~	~~83~~	~~84~~
~~97~~	~~98~~	~~99~~	~~100~~
~~113~~	~~114~~	~~115~~	~~116~~

i = 4 (miss)
i = 5 (miss)
i = 6 (miss)
i = 7 (miss)

~~129~~	~~130~~	~~131~~	~~132~~
~~145~~	~~146~~	~~147~~	~~148~~
~~161~~	~~162~~	~~163~~	~~164~~
~~177~~	~~178~~	~~179~~	~~180~~

i = 8 (miss)
i = 9 (miss)
i = 10 (miss)
i = 11 (miss)

193	194	195	196
209	210	211	212
225	226	227	228
241	242	243	244

i = 12 (miss)
i = 13 (miss)
i = 14 (miss)
i = 15 (miss)

When j = 1, 2, 3,….,15 we will have a similar scenario. As the matrix demonstrates, there will be one miss for every one element. Since there are 256 total elements and 1 miss per 1 new element, then the respective total cache misses for the column by column summation equals to 256.

Column by column will have a total of 256 cache misses.

4. Virtual memory uses a page table to track the mapping of virtual addresses to physical addresses. A stream of virtual addresses as seen on a system is as following:

4969, 3227, 14916

Assume 4 KiB pages, a 4-entry fully associative TLB, true LRU replacement, and the most recently used physical pages are 4, 6, and 12. If pages must be brought in from disk, increment the next largest page number. The initial TLB and page table are below:

Valid	Tag	Physical Page Number
1	11	12
1	7	4
1	3	6
0	4	9

Figure 1: Initial TLB

Valid	Physical Page or in Disk
1	5
0	Disk
0	Disk
1	6
1	9
1	11
0	Disk
1	4
0	Disk
0	Disk
1	3
1	12

Figure 2: Initil Page Table

a. Given the address stream above, compute the virtual page number of each reference, and show how the TLB and page table are updated. Also list for each reference if it is a hit in the TLB, or a page fault.

First, we must convert the three virtual addresses to binary numbers and find their corresponding virtual page numbers. It is important to note that the corresponding virtual page numbers are the numbers represented by the upper 20 bits [31 – 12]. That is, since each page is 4 KiB = 2^{12} Bytes, then the lower 12 bits of the virtual addresses are the page offsets and can be ignored.

Virtual Address (Decimal)	**Virtual Address 32 Bits (Binary)**	**Virtual Page Number [31-12] (Decimal)**
4969	0b0000-0000-0000-0000-0001-0011-0110-1001	**1**
3227	0b0000-0000-0000-0000-0000-1100-1001-1011	**0**
14916	0b0000-0000-0000-0000-0011-1010-0100-0100	**3**

Equivalent virtual page numbers (in decimal) are the following:
4969 => 1
3227 => 0
14916 => 3

Now, we must check if each reference is a TLB hit, TLB miss/Page Table Hit, or page fault. A TLB hit occurs when the virtual page number matches a valid tag number in the TLB table, a TLB miss/Page Table Hit occurs when the virtual page number does not match a tag number in the TLB table (or is not valid) but does match a valid index in the page table, and a page fault occurs when the virtual page number does not match a tag number in the TLB table (or is not valid) and does not match an index in the page (or is not valid). Because of this, we get the following:

Reference	**Result**
4969	**Page Fault (i.e. TLB Miss and Page Table Miss)**
3227	**TLB Miss but Page Table Hit**
14916	**TLB Hit**

References and their result:
4969 => Page Fault (i.e. TLB Miss and Page Table Miss)
3227 => TLB Miss but Page Table Hit
14916 => TLB Hit

We first do the final TLB table. Since the most recently used physical pages are 4, 6, and 12 (in that order), then we have the initial TLB as the following (NOTE: An LRU = 4 means "most recently used" and an LRU = 1 means "least recently used".)

INITIAL *(NOTE: All numbers are in decimal base-10):*

Valid	**Tag**	**Physical Page Number**	**LRU**
1	11	12	2
1	7	4	4
1	3	6	3
0	4	9	1

AFTER 4969 => Page Fault
(NOTE: All numbers are in decimal base-10):

Valid	**Tag**	**Physical Page Number**	**LRU**
1	11	12	1
1	7	4	3
1	3	6	2
1	1	13	4

AFTER 3227 => TLB Miss but Page Table Hit
(NOTE: All numbers are in decimal base-10):

Valid	**Tag**	**Physical Page Number**	**LRU**
1	0	5	4
1	7	4	2
1	3	6	1
1	1	13	3

After 14916 => TLB Hit
(Final Updated TLB Table):

Valid	Tag	Physical Page Number	LRU
1	**0**	**5**	3
1	**7**	**4**	1
1	**3**	**6**	4
1	**1**	**13**	2

(NOTE: All numbers are in decimal base-10. Furthermore, the higher the LRU, the more recently the row was used.)

Lastly, we have to display how the page table was updated. It is important to note that only index one is changed.

Final Updated Page Table:

Valid	Physical Page or in Disk	Index
1	**5**	0
1	**13**	1
0	**Disk**	2
1	**6**	3
1	**9**	4
1	**11**	5
0	**Disk**	6
1	**4**	7
0	**Disk**	8
0	**Disk**	9
1	**3**	10
1	**12**	11

b. Repeat (a), but this time use 16 KiB pages instead of 4 KiB pages. What would be some of the advantages of having a larger page size? What are some of the disadvantages?

ASSUME THAT WE BEGIN AGAIN WITH THE ORIGINAL INITIAL TLB AND PAGE TABLES

We must repeat (a), but this time we use 16 KiB pages instead of 4 KiB pages. Since 16 KiB = 2^{14} Bytes, then the lower 14 bits of the virtual addresses are the page offsets and can be ignored.

Virtual Address (Decimal)	**Virtual Address 32 Bits (Binary)**	**Virtual Page Number [31-14] (Decimal)**
4969	0b0000-0000-0000-0000-0001-0011-0110-1001	**0**
3227	0b0000-0000-0000-0000-0000-1100-1001-1011	**0**
14916	0b0000-0000-0000-0000-0011-1010-0100-0100	**0**

Equivalent virtual page numbers (decimal) are the following:
4969 => 0
3227 => 0
14916 => 0

Now, we must check if each reference is a TLB hit, TLB miss/Page Table Hit, or page fault. A TLB hit occurs when the virtual page number matches a valid tag number in the TLB table, a TLB miss/Page Table Hit occurs when the virtual page number does not match a tag number in the TLB table (or is not valid) but does match a valid index in the page table, and a page fault occurs when the virtual page number does not match a tag number in the TLB table (or is not valid) and does not match an index in the page (or is not valid). Because of this, we get the following:

Reference	**Result**
4969	**TLB Miss but Page Table Hit**
3227	**TLB Hit**
14916	**TLB Hit**

References and their result:
4969 => TLB Miss but Page Table Hit
3227 => TLB Hit
14916 => TLB Hit

As the previous table demonstrates, if we use 16 KiB pages instead of 4 KiB pages, there will not be any page faults for the three given virtual addresses. We first do the final TLB table. Since the most recently used physical pages are 4, 6, and 12 (in that order), then we have the initial TLB as the following (NOTE:

An LRU = 4 means "most recently used" and an LRU = 1 means "least recently used".)

INITIAL *(NOTE: All numbers are in decimal base-10):*

Valid	Tag	Physical Page Number	LRU
1	11	12	2
1	7	4	4
1	3	6	3
0	4	9	1

AFTER 4969 => TLB Miss but Page Table Hit
(NOTE: All numbers are in decimal base-10):

Valid	Tag	Physical Page Number	LRU
1	11	12	1
1	7	4	3
1	3	6	2
1	0	5	4

AFTER 3227 => TLB Hit
(NOTE: All numbers are in decimal base-10):

Valid	Tag	Physical Page Number	LRU
1	11	12	1
1	7	4	3
1	3	6	2
1	0	5	4

After 14916 => TLB Hit
(Final Updated TLB Table):

Valid	Tag	Physical Page Number	LRU
1	**11**	**12**	1
1	**7**	**4**	3
1	**3**	**6**	2
1	**0**	**5**	4

(NOTE: All numbers are in decimal base-10. Furthermore, the higher the LRU, the more recently the row was used.)

Lastly, we have to display how the page table was updated. *It is important to note that nothing in the page table is changed if we use 16 KiB pages instead of 4 KiB pages for those specific three virtual addresses.*

Final Updated Page Table (No Change from Initial):

Valid	**Physical Page or in Disk**	**Index**
1	**5**	0
0	**Disk**	1
0	**Disk**	2
1	**6**	3
1	**9**	4
1	**11**	5
0	**Disk**	6
1	**4**	7
0	**Disk**	8
0	**Disk**	9
1	**3**	10
1	**12**	11

Advantages:

Some advantages of having a larger page size include the following:

- Lower TLB miss rates
- Lower page fault rates
- Require smaller page tables
- Lower searching times in the page table
- Higher TLB hit rates
- Might lead to less "overhead"

Disadvantages:

Some disadvantages of having a larger page size include the following:

- Higher fragmentation since larger paper sizes lead to using the same page number for different addresses
- Longer disk transfer rates
- Might lead to more page faults than a smaller page size *if system has no locality*
- Might lead to inefficient memory usage