Q1.A

512KB = $2^9 * 2^{10} = 2^{19}$

32word * 4 = 128bytes = 2^7

Calculate index = 19-7 = 12

Tag = 32 – 12 – 7 = 13bit

13-bit (Tag)	12-bit (index)	7-bit (offset)

Field	Size (bits)
Cacheline Offset	7
Cacheline Index	12
Tag	13

Q1.B

512KB = $2^9 * 2^{10} = 2^{19}$

32word * 4 = 128bytes = 2^7

Index = 0 (since this is a Fully-Associative cache)

Tag = 32 – 0 – 7 = 25bit

25-bit (Tag)	0-bit (index)	7-bit (offset)

Field	Size (bits)
Cacheline Offset	7
Cacheline Index	0
Tag	25

Q1.C

512KB = $2^9 * 2^{10} = 2^{19}$

32word * 4 = 128bytes = 2^7

Index = 19-7(bit-offset) – 3(2^3 way set associative) = 9

Tag = 32-7-9 = 16

16-bit (Tag)	9-bit (index)	7-bit (offset)

Field	Size (bits)
Cacheline Offset	7
Cacheline Index	9
Tag	16

Q1.D

Address **After 2-bit offset**

0x128 0000 0001 0010 1000 (cache offset)

0xF40 0000 1111 0100 0000 (cache offset)

0xC00024 1100 0000 0000 0000 0010 0100 (cache offset)

0x014 0000 0000 0001 0100 (cache offset)

0x1000F44 0001 0000 0000 0000 1111 0100 0100 (cache offset)

Address	Request	Cachline Index	Valid	Modified	Tag	Data	Caused replaced?	Write-back to Memory?
0x128	read	0x2	1	0	0x0	M[0x100]	No	No
0xF40	write	0x1E	1	1	0x0	D[0xF00]	No	No
0xC00024	read	0x0	1	0	0x18	D[0xC00000]	No	No
0x014	write	0x0	1	1	0x0	D[0x00]	Yes	No
0x1000F44	read	0x1E	1	0	0x20	D[1000F00]	Yes	Yes

Q1.E

Address	Request	Set Index	Valid	Modified	Tag	Data	Caused replaced?	Write-back to Memory?
0x128	read	0x2	1	0	0x0	M[0x100]	No	No
0xF40	write	0x1E	1	1	0x0	D[0xF00]	No	No
0xC00024	read	0x0	1	0	0xC0	D[0xC00000]	No	No
0x014	write	0x0	1	1	0x0	D[0x00]	No	No
0x1000F44	read	0x1E	1	0	0x100	D[1000F00]	No	No

Question 1.F Overhead

Direct-mapped:

32*4*8bit = 1024 bits

tag + valid + modified = 13 + 1 + 1 = 15

2^19 / 2^7 = 2^12 number of cachelines * 15 = 61440 = 60Kb / 8 = 7.5KB

8-way associative cache:

32*4*8bit = 1024 bits

Tag + valid + modified = 16 + 1 + 1 = 18

2^19 / 2^7 * 2^3 = 2^9 number of cachelines * 18 = 9216 = 9Kb / 8 = 1152B = 1.1KB

8-way associative has less overhead.

Q2.A

Component	Delay Equation	Delay
addr_reg_M0	2	2
tag_decoder	2+2.2	6
tag_mem	12+ceiling[(4+27)/32]	13
tag_cmp	2+3.ceiling[$\log_2$ 26]	17
tag_and	2	2
data_decoder	2+2.2	6
data_mem	12+ ceiling[(8+128)/32]	17
data_mux	1+ 2.ceiling[$\log_2$ 4]	5
rdata_reg_M1	1	1
Total		69

Total delay time of the two-way set-associative cache is greater than the delay time of the direct-mapped cache, because there is an additional decoder in the set-associative cache in its critical path between the tag_and and data_mem (which is data_decoder). So, that's the reason for having a small delay in the two-way set-associative cache.

Q2.B

Direct-Mapped Cache:

Component	Delay Equation	Delay
addr_reg_M0	2	2
idx_mux	1+2.ceiling[$\log_2$ 2]	3
data_decoder	2+2.3	8
data_mem	12 + ceiling[(8 + 128)/32]	17
data_mux	1+2.ceiling[$\log_2$ 4]	5
rdata_reg_M1	1	1
Total		36

Two-way set-associative cache:

Component	Delay Equation	Delay
addr_reg_M0	2	2
idx_mux	1+2.ceiling[$\log_2$ 2]	3
data_decoder	2+2.2	6
data_mem	12 + ceiling[(4 + 128)/32]	17
data_mux	1+2.ceiling[$\log_2$ 4]	5
way_mux	1+2.ceiling[$\log_2$ 2]	3
rdata_reg_M1	1	1
Total		37

Total delay time of the two-way set-associative cache is greater than the delay time of the direct-mapped cache, because there is an additional mux in its critical path that it could potentially choose between two ways. So, this mux causes the two-way set-associative cache to have a greater delay time compare to the direct-mapped cache.

Q2.C

Direct-Mapped Cache:

Component	Delay Equation	Delay
addr_reg_M0	2	2
tag_decoder	2+2.3	8
tag_mem	12+ceiling[(8+26)/32]	14
tag_cmp	2+3.ceiling[$\log_2$ 25]	17
tag_and	2	2
wr_en_M1	1	1
Total		44

Two-way set-associative cache:

Component	Delay Equation	Delay
addr_reg_M0	2	2
Tag_decoder	2+2.2	6
Tag_mem	12+ceiling[(4+27)/32]	13
Tag_cmp	2+3.ceiling[$\log_2$ 26]	17
Tag_and	2	2
Tag_or	2	2
Wr_en_M1	1	1
Total		43

Well, the critical path for the two-way set-associative cache is smaller than the critical path for the directed-mapped cache, because the value for tag_mem and tag_decoder have smaller (this is because of the different value of index) values.

Q3A.

AMAT = %InstructionAccess.(HitTimeFor_IL1 + MissRateFor_IL1 . (HitTimeFor_L2 + MissRateFor_L2 . (HitTimeFor_MainMemory))) + %Data_Access . (HitTimeFor_DL1 + MissRateFor_DL1 . (HitTimeFor_L2 + MissRateFor_L2. (HitTimeFor_MainMemory))) / (%InstructionAccess + %Data_Access)

AMAT = 1(1+0.08(6+0.3.(50))) + 0.25(1+0.15(6+0.3.(50))) / (1+0.25) = 2.974

Q3B.

We know that %25 of instructions are accessing data memory, we have:
CPI = 1 + memory stalls for fetching instruction (each memory access) + memory stalls for the data memory access

CPI = 1+52 + 0.25(52) = 66

Q3C.

Memory stalls = (MissRateFor_IL1 + MissRateFor_DL1) . (TimeHitFor_L2 + MissRateFor_L2 . (TimeToAccessMemoryFromL2))

Memory stalls = 0.08.(6) + 0.08.(0.30).(50) + 0.15.(0.25).(6) + 0.15.0.25.(0.30.50)
Memory stalls = 0.48 + 1.2 + 0.225 + 0.5625 = 2.4675

CPI = 1+2.4675= 3. 4675

Q4A.

4KB page size:

4KB page size = $2^{10}.2^2 = 2^{12}$ ➔ 12 bits
24 - 12 = 12 bits (this is for virtual pages)
Total page table size (4KB pages) = 2^{12}.(3) = 4096 * (2^1 + 2^0) = 12288B / 1000 = 12.288KB

16KB page size:

16KB page size = $2^4.2^{10} = 2^{14}$ ➔ 14 bits
24 - 14 = 10 bits (this is for virtual pages)
Total page table size (16KB pages) = 2^{10}. (2^1 + 2^0) = 3072 /1000 = 3.1KB

Q4B.

0x5D10 ➔ TLB hit, and also page table hit

Valid	Tag	Physical Address	LRU
0	0xB	0x1B	
1	0x5	0x15	1
1	0x3	0x13	
1	0x6	0x16	

0xB000 ➔ TLB hit, but page table miss

Valid	Tag	Physical Address	LRU
1	0xB	0x1B	2
1	0x5	0x15	1
1	0x3	0x13	
1	0x6	0x16	

0x7200 ➔ TLB miss, page table hit

Valid	Tag	Physical Address	LRU
1	0xB	0x1B	2
1	0x5	0x15	1
1	0x7	0x17	3
1	0x6	0x16	

0x6800 ➔ TLB hit, page table hit

Valid	Tag	Physical Address	LRU
1	0xB	0x1B	2
1	0x5	0x15	1
1	0x7	0x17	3
1	0x6	0x16	4

0x5800 ➔ TLB hit, page table hit

Valid	Tag	Physical Address	LRU
1	0xB	0x1B	2
1	0x5	0x15	5
1	0x7	0x17	3
1	0x6	0x16	4

0x0000 ➔ TLB miss, page table miss

Valid	Tag	Physical Address	LRU
1	0xB	0x1B	2
1	0x5	0x15	5
1	0x0	0x10	6
1	0x6	0x16	4

0x3100 ➔ TLB miss, page table hit

Valid	Tag	Physical Address	LRU
1	0xB	0x1B	2
1	0x5	0x15	5
1	0x0	0x10	6
1	0x3	0x13	7

Please note that the value of LRU represent least used entry. Smaller LRU represents the least used entry.

This is the updated page table:

Index	Valid	Physical Mapping
0	1	0x10
1	1	0x11
2	1	0x12
3	1	0x13
4	0	Disk
5	1	0x15
6	1	0x16
7	1	0x17
8	1	0x18
9	0	Disk
A	0	Disk
B	1	0x1B
C	0	Disk
D	0	Disk
E	0	Disk
F	1	0x1F

Q4C.

Since we have a two-way set-associative cache, we would need 1 bit to index it. We have the original table below:

Valid	Tag	Physical Address
0	0xB	0x1B
1	0x5	0x15
1	0x3	0x13
1	0x6	0x16

Basically cacheline offset will not change, but there will 1-bit taken from the tag to determine where that tag will be sitting inside the set. If the 1-bit index is 0 the tag goes to set 0, if the 1-bit index is 1 the tag goes to set 1:

0xB = 0000 1011 ➔ (since we are dealing with 2-way 1bit for index) 0000 101**1**

Therefore the new tag would be changed to ➔ 0000 0101 which is 0x5 and since the 1-bit is 1, this tag will be in the set 1.

Same reason holds for the rest of the tag values inside the table.

0x5 = 0000 0101 ➔ 0000 0010 = 0x2

0x3 = 0000 0011 ➔ 0000 0001 = 0x1

0x6 = 0000 0110 ➔ 0000 0011 = 0x3

Therefore the updated TLB table is:

Valid	Tag	Physical Address	Set	LRU
0	0x5	0x1B	1	
1	0x2	0x15	1	
1	0x1	0x13	0	
1	0x3	0x16	0	

Assume the first two cachelines are in set 1 and the last two cachelines are in set 0.

0x5D10 ➔ TLB hit, and page table hit (5 ➔ 0101 last bit is for index, so new tag is 0x2)

Valid	Tag	Physical Address	Set	LRU
0	0xB	0x1B	1	
1	0x2	0x15	1	1
1	0x3	0x13	0	
0	0x6	0x16	0	

0xB000 ➔ TLB miss, page table hit (0xB ➔ 1011 -> 0101 which is 0x5)

Valid	Tag	Physical Address	Set	LRU
1	0x5	0x1B	1	2
1	0x2	0x15	1	1
1	0x3	0x13	0	
1	0x6	0x16	0	

0x7200 ➔ TLB miss, page table hit (0x7 ➔0x3)

This is a miss, because based in its index 1 (0111 ➔ 0011 (index =1)) it does not exist in set 1.

Valid	Tag	Physical Address	Set	LRU
1	0x5	0x1B	1	2
1	0x2	0x15	1	1
1	0x3	0x13	0	
1	0x6	0x16	0	

0x6800 ➔ TLB hit, page table hit

Valid	Tag	Physical Address	Set	LRU
1	0x5	0x1B	1	2
1	0x2	0x15	1	1
1	0x3	0x13	0	3
1	0x6	0x16	0	

0x5800 ➔ TLB hit, page table hit (0101 ➔ 0010)

Valid	Tag	Physical Address	Set	LRU
1	0x5	0x1B	1	2
1	0x2	0x15	1	4
1	0x3	0x13	0	3
1	0x6	0x16	0	

0x0000 ➔ TLB miss, page table miss

Valid	Tag	Physical Address	Set	LRU
1	0x5	0x1B	1	2
1	0x2	0x15	1	4
1	0x3	0x13	0	3
1	0x0	0x10	0	5

0x3100 ➔ TLB miss, page table hit (0011 ➔ 0001)

Valid	Tag	Physical Address	Set	LRU
1	0x1	0x10	1	7
1	0x2	0x15	1	4
1	0x3	0x13	0	6
1	0x0	0x10	0	5

Index	Valid	Physical Mapping
0	1	0x10
1	1	0x11
2	1	0x12
3	1	0x13
4	0	Disk
5	1	0x15
6	1	0x16
7	1	0x17
8	1	0x18
9	0	Disk
A	0	Disk
B	0	Disk
C	0	Disk
D	0	Disk
E	0	Disk
F	1	0x1F

Q4D.

Since we are dealing with 16KB page size, therefore we have ➔ $2^4 * 2^{10} = 2^{14}$ which has 2 more bits for the page offset. 2 more bits for offset will result in having a shorter tag , because of the following: 24 -14 = 10 which is 2bits less than what we had before (24-12 = 12 for tag). Therefore we need to modify the TLB table as shown below:

0xB = 0000 1011 ➔ (taking the last 2bits of tag, we have) 0000 0010 = 0x2

0x5 = 0000 0101 ➔ (taking the last 2bits of tag, we have) 0000 0001 = 0x1

0x3 = 0000 0011 ➔ (taking the last 2bits of tag, we have) 0000 0000 = 0x0

0x6 = 0000 0110 ➔ (taking the last 2bits of tag, we have) 0000 0001 = 0x1

Valid	Tag	Physical Address
0		
1	0x1	0x11
0		
1	0x1	0x11

One of the two tags should be removed, since we have two same tags, therefore we remove

1	0x1	0x11

from the TLB table. So we have the table below:

Valid	Tag	Physical Address
0		
0		
0		
1	0x1	0x11

Q4E.

We could reduce the TLB misses by having bigger pages, which it ultimately result in performance improvement. But the downside of having larger pages is that we would be wasting more memory. Because sometimes a program may need just a small space (less memory), but there is a lot more space defined for it, which is not really needed.