CMSC611: Advanced Computer Architecture
Extra Credit Homework 2 Solutions

a) Hex address to Binary address:

$(5F852B24)_{16} = (0101\ 1111\ 1000\ 0101\ 0010\ 1011\ 0010\ 0100)_2$

Byte offset:	1 word = 4 bytes	$2^{\text{byte offset}} = 4$	Byte offset = 2 bits
Word offset:	1 word = 4 bytes = 32 bits	$2^{\text{word offset}}$ = Number of words in each block	
Index:	2^{index} = Cache size / (Block size × Number of set associativity)		
Tag:	Tag = 32 – Byte offset – Word offset – Index		

L1 instruction cache:

Byte offset:	1 word = 4 bytes	$2^{\text{byte offset}} = 4$	Byte offset = **2 bits**
Word offset:	256 bits = 32 bytes = 8 words	$2^{\text{word offset}} = 8$	Word offset = **3 bits**
Index:	2^{index} = Cache size / (Block size × Number of set associativity) = (32 × 1024) / (32 × 4) = 256		Index = **8 bits**
Tag:	Tag = 32 – Byte offset – Word offset – Index = 32 – 2 – 3 – 8 = **19 bits**		

L1 data cache:

Byte offset:	1 word = 4 bytes	$2^{\text{byte offset}} = 4$	Byte offset = **2 bits**
Word offset:	256 bits = 32 bytes = 8 words	$2^{\text{word offset}} = 8$	Word offset = **3 bits**
Index:	2^{index} = Cache size / (Block size × Number of set associativity) = (32 × 1024) / (32 × 8) = 128		Index = **7 bits**
Tag:	Tag = 32 – Byte offset – Word offset – Index = 32 – 2 – 3 – 7 = **20 bits**		

L2 cache:

Byte offset:	1 word = 4 bytes	$2^{\text{byte offset}} = 4$	Byte offset = **2 bits**
Word offset:	64 bytes = 16 words	$2^{\text{word offset}} = 16$	Word offset = **4 bits**
Index:	2^{index} = Cache size / (Block size × Number of set associativity) = (256 × 1024) / (64 × 8) = 512		Index = **9 bits**
Tag:	Tag = 32 – Byte offset – Word offset – Index = 32 – 2 – 4 – 9 = **17 bits**		

L3 cache:

Byte offset:	1 word = 4 bytes	$2^{\text{byte offset}} = 4$	Byte offset = **2 bits**
Word offset:	64 bytes = 16 words	$2^{\text{word offset}} = 16$	Word offset = **4 bits**
Index:	2^{index} = Cache size / (Block size × Number of set associativity) = (8 × 1024 × 1024) / (64 × 16) = 8192		Index = **13 bits**
Tag:	Tag = 32 – Byte offset – Word offset – Index = 32 – 2 – 4 – 13 = **13 bits**		

Cache level	Tag	Index	Offset		Size of read
			Word offset	Byte offset	
L1 instruction cache	0101 1111 1000 0101 001	0 1011 001	0 01	00	1 word
L1 data cache	0101 1111 1000 0101 0010	1011 001	0 01	00	1 word
L2 cache	0101 1111 1000 0101 0	010 1011 00	10 00	00	8 words
L3 cache	0101 1111 1000 0	101 0010 1011 00	00 00	00	16 words

Memory Read	Remainder of address	L3 sub-block	Byte offset in read	Size of Read
Memory (0)	0101 1111 1000 0101 0010 1011 00	00 0	000	2 words
Memory (1)	0101 1111 1000 0101 0010 1011 00	00 1	000	2 words
Memory (2)	0101 1111 1000 0101 0010 1011 00	01 0	000	2 words
Memory (3)	0101 1111 1000 0101 0010 1011 00	01 1	000	2 words
Memory (4)	0101 1111 1000 0101 0010 1011 00	10 0	000	2 words
Memory (5)	0101 1111 1000 0101 0010 1011 00	10 1	000	2 words
Memory (6)	0101 1111 1000 0101 0010 1011 00	11 0	000	2 words
Memory (7)	0101 1111 1000 0101 0010 1011 00	11 1	000	2 words

b) HT: hit time
MR: miss rate
MP: miss penalty

$$\text{Memory Access Cycle} = ceil(13.75\text{ns} \times 2.6\text{GHz}) = ceil(35.75) = 36 \text{ cycles}$$

$$\begin{aligned}
\text{Average Memory Access Time} &= \text{HT}_{\text{L1 data cache}} + \text{MR}_{\text{L1 data cache}} \times \left[\text{HT}_{\text{L2 cache}} + \text{MR}_{\text{L2 cache}} \times \left(\text{HT}_{\text{L3 cache}} + \text{MR}_{\text{L3 cache}} \times \text{MP}_{\text{L3 cache}} \right) \right] \\
&= 4 + 0.7\% \times \left[10 + 4\% \times \left(40 + 12\% \times (36 \times 8) \right) \right] \text{ cycles} \\
&= 4 + 0.7\% \times [10 + 4\% \times (40 + 12\% \times 288)] \text{ cycles} \\
&= 4 + 0.7\% \times [10 + 4\% \times 74.56] \text{ cycles} \\
&= 4 + 0.7\% \times 12.98 \text{ cycles} \\
&= 4 + 0.09 \text{ cycles} \\
&\approx 4.09 \text{ cycles}
\end{aligned}$$

$$\begin{aligned}
\text{Actual Access Time} &= \text{Access Time}_{\text{L1 data cache}} + \text{Access Time}_{\text{L2 cache}} + \text{Access Time}_{\text{L3 cache}} + \text{Access Time}_{\text{Memory}} \\
&= 4 + 10 + 40 + 36 \times 8 \\
&= 342 \text{ cycles} \\
&= \frac{342}{2.6} \text{ ns} \approx 131.5 \text{ ns}
\end{aligned}$$

c) The code with loops will benefit from the Loop Stream Detector. Generally, when a Core-based CPU is running a loop detected by LSD, the CPU doesn't need to fetch the required instructions again from the L1 instruction cache, because they have already been close to the decode unit in LSD. Instructions can be simply streamed out from LSD to the decode unit. In addition, the CPU actually turns off the fetch and branch prediction units while running a detected loop, making the CPU to avoid possible misprediction penalties and save some power.

On a Nehalem-based CPU, LSD has been moved to after the decode unit. So, instead of holding instructions, LSD holds micro-operations provided by the decode unit. When the CPU is running a loop, it doesn't need to decode the instructions present in the loop, because they have already been decoded and stored inside LSD. The CPU can now turn off the decode unit together with the fetch and branch prediction units when running a detected loop, making the CPU to be even more efficient than the Core-based CPU. Therefore, the location change of LSD improves the performance of the Nehalem-based CPU comparing to the Core-based CPU.