Solver: CHEN BENJAMIN, HOANG VIET, WILLIS TEE TEO KIAN, YONG SHAN JIE

1)

a)

i) Data Dependencies:
Between I1 & I2
Between I2 & I3
Between I5 & I6
Total stalls required per iteration: 7 (2 from each data dependency, 1 to determine branch target)
Steady State CPI = $\frac{7+6\ cycles}{6\ instructions} = \frac{13}{6}$ = **2.1667**
Loop counts down from 0x0036 to 0 by addi 0xFFFF, 0x0036 in decimal is 54, hence total **54 loop iterations**.

ii) Let Way 1 be used for LW/SW instruction, and Way 2 for all other instructions.
We first lay out the key function of this loop, which is I1, I2, and I3, since we know there is data dependencies between these 3 instructions, they must be spaced at least two instructions apart.

Cycle	Way-1	Way-2
1	lw $t1, 0($s5)	
2		
3		
4		addi $t5, $t1, #0x0004
5		
6		
7	sw $t5, 0($s5)	

Next, we need to rid the data dependency between I5 and I6, by spacing them at least two instructions apart. We also know that I6 updates PC at Decode stage, hence to prevent additional stall, I6 is placed in cycle 6. Lastly, we insert the remaining instruction, I4, at any sensible location. (Note: Placing I4 in cycle 5 does not affect the SW instruction because during SW, the $s5 value is still the old $s5 value)

Cycle	Way-1	Way-2
1	loop: lw $t1, 0($s5)	addi $s6, $s6, #0xFFFF.
2	nop	nop
3	nop	nop
4	nop	addi $t5, $t1, #0x0004
5	nop	addi $s5, $s5,# 0x0008
6	nop	bgtz $s6, loop
7	sw $t5, 0($s5)	nop

New CPI: $\frac{7\ cycles}{6\ instructions} = \mathbf{1.1667}$

iii) It was clarified during exam that this question is a continuation of part ii and is to be done in 2-way superscalar processor. Same rule applies, that all data dependencies must be spaced 2 cycle apart.

Cycle	Way-1	Way-2
1	loop: lw $t1, 0($s5)	addi $s6, $s6, #0xFFFC.
2	lw $t2, 8($s5)	nop
3	lw $t3, 16($s5)	nop
4	lw $t4, 24($s5)	addi $t5, $t1, #0x0004
5	nop	addi $t6,$t2, #0x0004
6	nop	addi $t7, $t3, #0x0004
7	sw $t5, 0($s5)	addi $t8,$ t4, #0x0004
8	sw $t6, 8($s5)	addi $s5, $s5,# 0x0020
9	sw $t7, 16($s5)	bgtz $s6, loop
10	sw $t8, 24($s5)	nop

CPI: $\frac{10\ cycles}{15\ instructions} = \mathbf{0.6667}$

b) Ahmdal's law in formula:

$$\text{Speedup} = \frac{1}{(1-E)+\frac{E}{S}}$$

where E is the % of the program that can be sped up by a factor of S
For enhancement E, we have E = 0.6, S = 4
For enhancement E', let E' = x, S' = 2

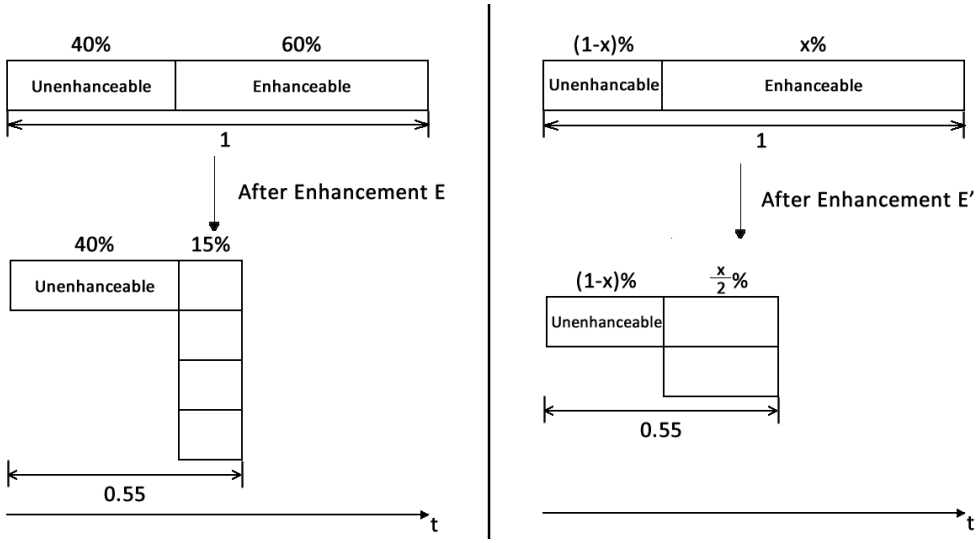

Figure 1. Visual representation of the two enhancements E and E'

In order to achieve the same speed up:

$$\frac{1}{(1-0.6)+\frac{0.6}{4}} = \frac{1}{0.55} = \frac{1}{(1-x)+\frac{x}{2}}$$

Solve for x, x = 0.9

Hence, percentage enhanceable for Enhancement E' should be 90%
(We later learned that the word 'enhanceable' does not exist, but you get the idea)

2)

a)

i) Taking an example, **SW $t5, 0($s5)**

1. I-mem (500ps)
2. There is no need to consider the Mux before Reg File, since the data travels to Read Register 1 and towards Sign-extend. As Multiplexing and Reading Reg data happens in parallel, we choose the one with higher latency - Reg File (200ps). Reg File also happens concurrently with control signals, sign extend & ALUSrc Mux
3. ALU (250ps)
4. D-Mem (500ps)

Therefore, 500 + 200 + 250 + 500 = **1450ps**

Taking an example, **BEQ $t1, $t0, LOOP**

1. I-mem (500ps)
2. Reg Read (200ps)
3. ALUSrc Mux (10ps), this is required since 'Read data 2' is used
4. ALU (250ps)
5. AND Gate (5ps)
6. Mux before PC (10ps)
7. PC (100ps)

Therefore, 500 + 200 + 10 + 250 + 5 + 10 + 100 = **1075ps**

There is no need to consider the timing for Adder (for branch) since it takes a shorter time compared to going from I-mem to the AND Gate.

ii) LW is known to be the instruction that passes through all 5 stages of the pipeline, hence it will have highest latency. Therefore, the maximum frequency is based on the latency of LW instruction.

Taking an example, **LW $t5, 0($s5)**

1. It takes the same path as SW and continues from D-MEM (1450ps)
2. MemtoReg Mux (10ps)
3. Reg Write (200ps)

Therefore, 1450ps + 10 + 200 = <u>1660ps</u>

Maximum possible frequency = 1 / 1660ps = <u>602.41MHz</u>

iii) Voltage scale down means frequency scale down as well.

Maximum possible frequency $f_{max} \propto \frac{[V-V_{th}]^2}{V}$

V_{th} is negligible compared to V, hence Let $V_{th} = 0$.

$f_{max} \propto V \Rightarrow 602.41MHz \propto 2.5V$

$\frac{f_{new}}{2.5} MHz \propto 1V$
$f_{new} = 204.96MHz$

b) Conditional Branch Instructions: PC-Relative Addressing
Unconditional Branch Instructions: Pseudo Direct Addressing
Each instruction is 32 bits, occupying 4 address space, hence offset must multiply by

Conditional Branching

nPC= PC + (immediate * 4) + 4
I-type instruction, therefore there is only 16-bits for offset.
Maximum possible offset: 0b0111 1111 1111 1111 -> 0x7FFF (**Remember sign bit**!!)
After left-shift by 2 bits/multiply by 4, expressed in 32 bits: 0x0001 FFFC
Minimum possible offset: 0b1000 0000 0000 0000 -> 0x8000
After left-shift by 2 bits/multiply by 4, expressed in 32 bits: 0xFFFE 0000
Maximum Possible Address: 0x2000 6F0C + 0x0001 FFFC + 0x4 = **0x2002 6F0C**
Minimum Possible Address: 0x2000 6F0C + 0xFFFE 0000 + 0x4 = **0x1FFE 6F10**

Unconditional Branching

nPC = PC[31:28] :: (immediate * 4) //:: refers to concatenation
J-type instruction, therefore there is 26-bits for offset. Since the **offset value does not go through Sign Extension**, there is no need for a signed-bit (i.e. offset is unsigned)
Largest possible offset: 0b11 1111 1111 1111 1111 1111 1111 -> 0x3FF FFFF
After left-shift by 2 bits/multiply by 4, expressed in 28 bits: 0xFFF FFFC
Smallest possible offset: 0b00 0000 0000 0000 0000 0000 0000 ->0x000 0000
After left-shift by 2 bits/multiply by 4, expressed in 28 bits: 0x000 0000
Maximum Possible Address: 0x2::FFF FFFC = **0x2FFF FFFC**
Minimum Possible Address: 0x2::F00 0000 = **0x2000 0000**

c) Since earlier outcome is all T, the start state would be in state '11'

Sequence	N	N	T	N	T	T
Current State	11	10	00	01	00	01
Taken?	Miss	Miss	Miss	Hit	Miss	Miss

Hit Rate: $\frac{1}{6} = 16.67\%$

3)

a) In Flynn's taxonomy:
Single instruction stream, single data stream (SISD)
Single instruction stream, multiple data streams (SIMD)
Multiple instruction streams, single data stream (MISD)
Multiple instruction stream, multiple data stream (MIMD)

Among these, SIMD architectures can make use of data-level parallelism efficiently as several operations on different data can be realized by one instruction. MIMD also can support data-

level parallelism, but at the cost of putting the same instruction on multiple stream, hence underutilization will arise.

b)

i) AMAT = Time for hit + (Miss Rate * Time for miss)
Finding X
1.93 = 1 + (X * 88)
X = **1.057%**
Finding Y
Y = 1 + (2.64% * 88)
Y = **3.323**
Finding Z
Z = 1 + (0.51% * 88)
Z = **1.449**

ii) With cache block of bigger size, access of bigger chunk of data will result in fewer misses than the case where block size is smaller, as it only requires fewer access to retrieve the entire chunk of data. For example, assume the block size is only 2 bytes, but the access requires 4 bytes of data, it will be required to do 2 accesses (2 x 2bytes = 4 bytes). This increases the chances of cache miss if the entire data (4 bytes) do not reside in the cache. But if the size is 4 bytes, the entire data chunk can be retrieved. Thus, this will reduce miss rate.

However, if we further increase block size, there would be fewer blocks given a fixed size cache. With fewer blocks, there would be fewer entries in the cache, hence there may be a higher increase in miss rate. Increasing the block size may also increase the number of conflict misses. There is a greater chance to displace a useful block from the cache. Hence as we keep increasing the block size, this effect will increase and outweigh the benefits brought about by bigger block size, hence a higher miss rate.

Additionally, as block size increase, it will also increase miss penalty, hence leading to a lower effective access time.

iii) Both cache miss rate and AMAT typically decrease when going from 16B to 64B block size, then increase when going from 64B to 256B. The only exception is the cache miss rate for cache size 256kB, where the miss rate only decreases when block size goes from 16B to 64B to 256B.

AMAT plays a more important role in maximizing cache performance because AMAT considers miss penalty, on top of cache miss rate, hence AMAT can look at cache performance from a larger perspective.

Take for example, for a cache size of 256kB, comparing block size of 64B and 256B, it is first seen that 256Bytes block size have better performance because it has a lower miss rate.

However, taking miss penalty into consideration, 64B Block size yields better performance over 256B block size.

4)

a) Architectural difference

	CPU	**GPU**
Main Aim	Low latency (Low execution time)	High Throughput
Instructions set	More suitable for sequential execution	Optimized for parallel execution
Scalability	Not easily scalable, there will be issues of cache coherency and atomicity in multicores system	Easily scalable by adding more GPU cores to the system

b)

i) N = 1 million
Each block runs 256 threads
We will need to Round up (1000000/256) = 3907
So, the line n will have this code:
`saxpy<<<3907,256>>>(N,a,d_X,d_Y);`

ii) Other functions needed:
1) *Before calling saxpy*
`cudaMemcpy(Host,Device,size of data,cudaMemcpyHostToDevice)`
for d_X and d_Y : this is to allocate memory in GPU
2) *After calling saxpy*
`cudaMemcpy(Device,Host,size of data,cudaMemcpyDeviceToHost)`
to copy back result from device to host
3) Calling free to release resource of d_X and d_Y

*It is worth to note that in practice, it is better to wrap these CUDA functions in an user-defined error checking function as CUDA is not able to print things on the console.
Your function can take in parameter cudaError_t err and check if err != cudaSuccess. This will help greatly in your GPU code debugging effort. (just for your information)

c)

i) 1 block with 256 threads to make use of **intra-block** information sharing, which is needed to compute the dot product. As data sharing is more efficiently implemented among threads rather than among blocks, this is the optimal configuration.

ii) There will be an issue of synchronization as thread 1 will try to sum up the component product before the actual results are available for computations.

An easy remedy is to insert `__syncthreads()` before line 7 to make sure that thread 0 will only execute the addition after all the other threads have carried out the multiplication.

--End of Answers--