1a For single cycle processing, the cycle time is given as the time of the longest instruction. This because each instruction must be computed within a single cycle and cycle time can't change (It actually can, but it's extremely inefficient). Therefore, the single cycle clock cycle time is 450 ps

For pipelined datapath, the cycle time is the length of the longest stage among all instructions. In this case, it is the lw MEM_2 stage with 150ps. So, the pipelined cycle time is 150ps

Multicycle datapath has the same cycle time as pipelined for the same reason and should also be 150 ps.

1b First calculate how long the pipelined instruction is. Every pipelined instruction runs every stage at the same speed as the clock cycle. Then the timing is as follows: IF ID EX_1 EX_2 WB, MEM_1 MEM_2 WB_2

150 150 150 150 150 150 150 150 = 1200 ps

While this is long, it only takes an additional 150 ps to run the next instruction Next, calculate the time for single cycle. Cycle time is already given as 450 ps. So every instruction runs in 450ps. While this may seem short, it takes an extra 450 ps. It only takes 4 instructions for single cycle to surpass pipelined in time for execution. To answer the question, subtract pipelined by single cycle to get 750 ps faster.

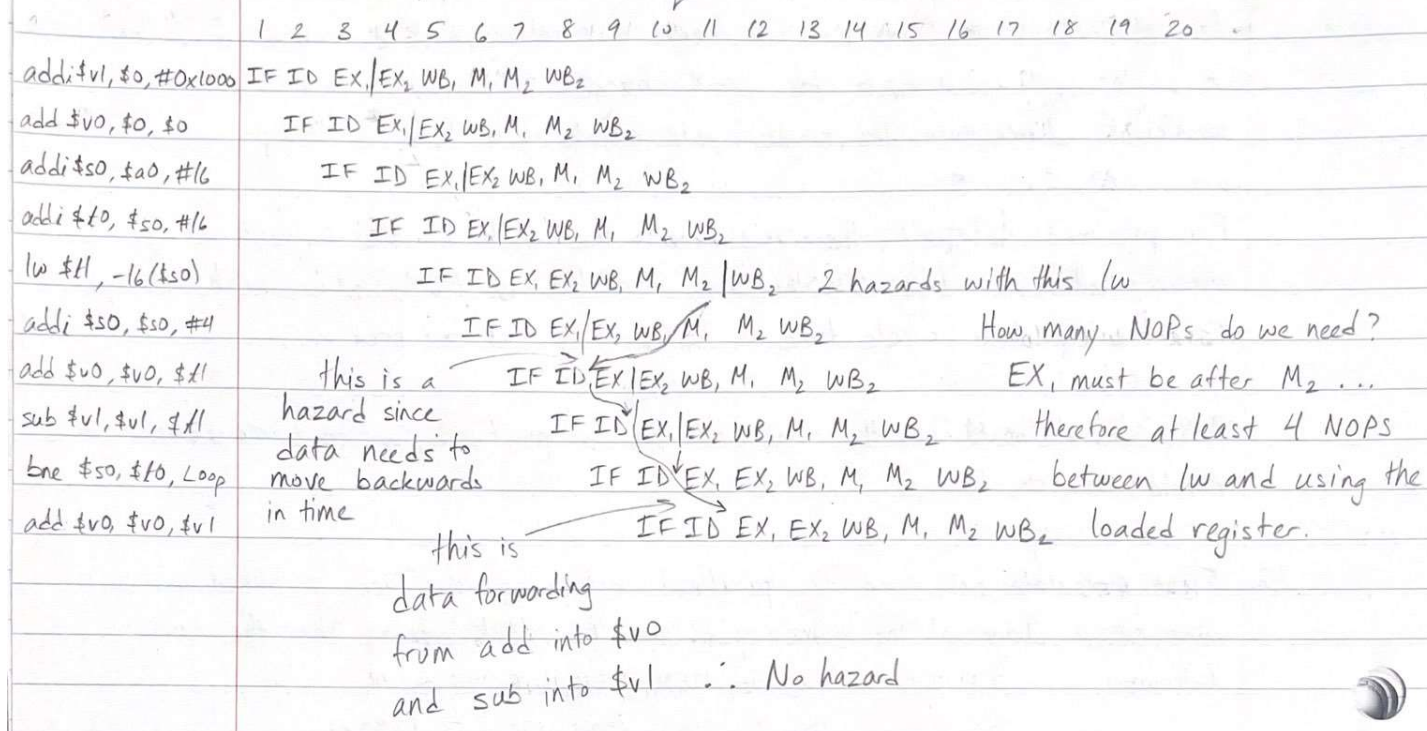
Lastly, we need to calculate multicycle time to completion. Multicycle and pipelined have the same clock cycle time, but multicycle only runs one instruction at a time. The benefit of multicycle is that it can skip unused stages, so for lw, the execution is as follows:

IF ID EX_1 EX_2 WB, MEM_1 MEM_2 WB_2

150 150 150 — — 150 150 150 = 900 ps

So multicycle is 300ps faster than pipelined for one instruction.

1c Draw the pipeline for this question



No branch delay hazard because the add $v0, $v0, $v1 instruction is killed by the branch if the branch is taken before writeback happens. Only 2 hazards, both read after write with register $t1 in the lw

1d

```
//Start with initialization
addi $v1, $0, #0x1000
add  $v0, $0, $0
addi $s0, $a0, #16
addi $t0, $s0, #16
//now loop uncroll. It is 4 copies because $t0 is 16 away from $s0
//and $s0 increments by 4
//start with loads, each load acts as a delay slot for previous loads
lw  $t1, -16($s0)
lw  $t2, -12($s0)      <- start adding new registers
lw  $t3, -8($s0)
lw  $t4, -4($s0)
addi $s0, $s0, 16      //increment $s0 for more delay
add $v0, $v0, $t1      //at this point only $t1 is ready, so just add it into $v0
sub $v1, $v1, $t1      //now $t2 is ready, but it's better to combine with $t3
add $t5, $t2, $t3      //combine $t2, $t3
add $t5, $t5, $t4      //combine with $t4
add $v0, $v0, $t5      //put them into $v0 and $v1
sub $v1, $v1, $t5
//now execute the line after the loop ends
add $v0, $v0, $v1
```

So, this code removes the hazards while loop uncrolling. However, during the midterm, follow the homework answers for loop unrolling. This is a more advanced question.

To compute $\overline{CPI}$, you simply need the number of cycles and the number of instructions. The first instruction, as discussed before, finishes in 8 cycles. Every instruction after is only one additional cycle. So the total number of cycles = 8 + (#ins - 1) = 23 cycles

#ins = 16 ins

Then $\overline{CPI} = \frac{23}{16}$

1e Using dual issue on the previous code with this format

ALU Branch	ANY	
addi $v1, $0, #0x1000	add $s0, $a0, #16	Start at the beginning
add $v0, $0, $0	lw $t1, -16($s0)	Swap $v0 and $s0, so we can load
addi $t0, $s0, #16	lw $t2, -12($s0	
lw $t4, -4($s0)	lw $t3, -8($s0)	
addi $s0, $s0, 16	NOP	$t1 isn't ready yet, but $s0 is
addi $s0, $s0, #16	NOP	$t1 still not ready
add $v0, $v0, $t1	sub $v1, $v1, $t1	$t1 ready now
NOP	NOP	$t2 ready now, going solely based
add $t5, $t2, $t3	NOP	on the previous code, nothing can run
add $t5, $t5, $t4	NOP	all loads available here
add $v0, $v0, $t5	sub $v1, $v1, $t5	
add $v0, $v0, $v1	NOP	

The new CPI will have the same number of instructions as before.
So 16 instructions. The number of cycles is now only dependent on the number of instructions run in each issue. Both sides have 12 instructions so the total number of cycle = 19. Then $\overline{CPI} = \frac{19}{16}$

1f Spatial locality defines that memory close to recently used memory will likely be used. It allows for multiple values close to each other to be brought up from memory to cache.
Temporal locality defines that memory recently used will likely be used again. Therefore, a recently used value will be brought up to cache since it will likely be used again.
So, for this question spatial locality could result in more hits.

1g It will not affect hit rate because the program accesses different locations in memory. This means they will all miss initially irregardless of cache structure.

2a

```
L.d   $f5, 8($f1)
L.d   $f2, 0($f1)
add.d $f3, $f2, $f2
daddui $f4, $0, #4.8
mul.d $f3, $f2, $f3
add.d $f2, $f5, $f3
```

Since processor format is not given, assume execution time of instructions approach ∞.

Then data dependencies are any registers that match going in this direction: ↘

Anti dependencies go the opposite way and are not real hazards. They only have the potential to be hazards with out of order processing. They are in this direction: ↙

Output dependencies are write after write hazards and go in this direction: ↓

Data dependencies:

$f5 L1 L6
$f2 L2 L3
$f2 L2 L5 Not
$f3 L3 L5 fixable
$f3 L5 L6

fixable

Anti dependencies	Output dependencies
$f2 L3 L6	$f2 L2 L6
$f2 L5 L6	$f3 L3 L5

2b Fixing output dependencies

```
L.d   $f5, 8($f1)
L.d   $f2, 0($f1)
add.d $f3, $f2, $f2
daddui $f4, $0, #4.8
mul.d $f9, $f2, $f3
add.d $f10, $f5, $f9
```

This ended up fixing the anti dependencies as well.

← This must change to $f9 because it must match the previous reference of $f3. Data dependencies can't be fixed

2c Cycle #1

	ADD/SUB						MUL/DIV					
	Ref	Tag	Ref	Tag	V		Ref	Tag	Ref	Tag	V	
1						4						
2						5						
3												

Reg	Busy	Data
$f1	N	1004
$f2	Y	M(1004)
$f3	N	26
$f4	N	12.2
$f5	Y	M(100c)

Load doesn't exactly go in a reservation station
We just say that it is busy at M(addr) instead
The first 2 loads are issued and begin executing in the first cycle
It will take until the end of Cycle #4 for the values to load

Cycle #2

	ADD/SUB					MUL/DIV				
	Ref	Tag	Ref	Tag	V	Ref	Tag	Ref	Tag	V
	26	0	—	M(1004)						
Ex left: 2	0	0	4.8	0						

$f1	N	1004
$f2	Y	M(1004)
$f3	Y	S1
$f4	Y	S2
$f5	Y	M(100c

The 2 add instructions are issued. The first add has to wait for the load, the second one is not dependent and can start execution.

Cycle #3

	ADD/SUB					MUL/DIV				
	Ref	Tag	Ref	Tag	V	Ref	Tag	Ref	Tag	V
	26	0	—	M(1004)		—	M(1004)	—	1	
Ex left: 1	0	0	4.8	0						
	—	M(100c)	—	4						

$f1	N	1004
$f2	Y	S3
$f3	Y	S4
$f4	Y	S2
$f5	Y	M(100c)

The last 2 instructions are issued. Make sure you update the reservation station reference tags before updating the register map data.

In cycle 4, the second add instruction will complete, but the data will only be ready in cycle 5. Same for the l.d instructions. So, cycle 4 will look identical to cycle 3.

Cycle #4

ADD/SUB				
Ref	Tag	Ref	Tag	V
26	0	–	M(1004)	
0	0	4.8	0	
–	M(100c)	–	4	

Ex left: 0

MUL/DIV				
Ref	Tag	Ref	Tag	V
–	M(1004)	–	1	

$f1	N	1004
$f2	Y	S3
$f3	Y	S4
$f4	Y	S2
$f5	Y	M(100c)

In cycle #5, we need to update values in both reservation stations and the register file for the contents of M(1004), M(100c), and $f4.

Cycle #5

ADD/SUB				
Ref	Tag	Ref	Tag	V
26	0	2.6	0	
–	–	–	–	4.8
0.6	0	–	4	

Ex left: 2

MUL/DIV				
Ref	Tag	Ref	Tag	V
2.6	0	–	1	

$f1	N	1004
$f2	Y	S3
$f3	Y	S4
$f4	N	4.8
$f5	N	0.6

Cycles 6 and 7 will be identical to Cycle #5 since both instructions are waiting on operation 1 to finish first

Cycle 6, 7

ADD/SUB				
Ref	Tag	Ref	Tag	V
26	0	2.6	0	
0.6	0	–	4	

MUL/DIV				
Ref	Tag	Ref	Tag	V
2.6	0	–	1	

$f1	N	1004
$f2	Y	S3
$f3	Y	S4
$f4	N	4.8
$f5	N	0.6

In cycle 8, update values for the results of S1

Cycle #8

ADD/SUB				
Ref	Tag	Ref	Tag	V
–	–	–	–	28.6
0.6	0	–	4	

Ex left: 6

MUL/DIV				
Ref	Tag	Ref	Tag	V
2.6	0	28.6	0	

$f1	N	1004
$f2	Y	S3
$f3	Y	S4
$f4	N	4.8
$f5	N	0.6

I'm not going to show the rest of Tomasulo's because it's too much writing. After Cycle #8, cycles 9, 10, 11, 12, 13, and 14 will be used to execute the multiply instruction. Then on cycle 15, it will write the result (74.36) into both $3 and $f3. Cycles 15, 16, and 17 will be used to execute the last add instruction. Cycle 18 will be the last cycle where the result (74.96) is written into $f2.

3 Cache size = 1 kB Main memory = 16 kB Block size = 4 bytes
direct mapped byte addressing

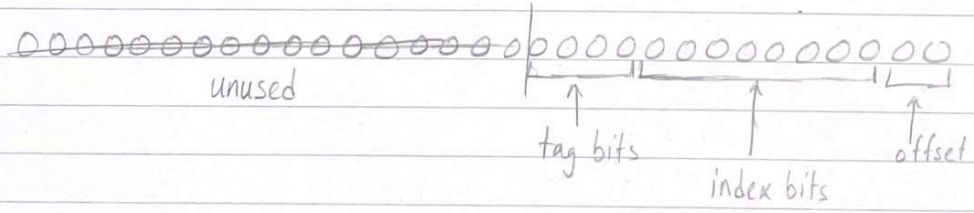
address bits = $\log_2$(main_memory_size)
$\log_2$(16 kB) ⟶ same thing as 16×2^{10}.
14 bits

index bits = $\log_2\left(\frac{\#\text{blocks}}{\text{associativity}}\right)$ ⟶ Since it is direct mapped, the number of blocks = cache size / block size and associativity = 1

index bits = $\log_2\left(\frac{1\text{kB}}{4}\right)$

10 − 2
8 bits

offset bits = $\log_2$(block size)
$\log_2$(4)
2 bits



4

INS	%	Cycles
LW/SW	25%	6
ALU	70%	2
Branch	5%	3

Ins miss = 5% penalty 5
L1 cache miss = 50% penalty 10
L2 cache miss = 25% penalty 40

$\overline{CPI} = 6(0.25) + 2(0.7) + 3(0.05)$

$\overline{CPI} = 1.5 + 1.4 + 0.15$

$\boxed{\overline{CPI} = 3.05}$

Use an arbitrary number of instructions to help calculate the second part. I like to use 1,000

Of 1000 instructions, 250 LW/SW, 700 ALU, and 50 branch.
Of 1000 instructions, 50 will miss with penalty 5 = +250 cycles

Of 250 LW/SW, 125 will miss = +1250
Of 125 that missed, 31.25 will miss again = +1250

Total time = $250(6) + 700(2) + 50(3) + 250 + 1250 + 1250$
$= 1500 + 1400 + 150 + 250 + 2500$
$= 2900 + 400 + 2500$
$= 5800$

$\frac{5800}{1000} = 5.8$ $\boxed{\text{new } \overline{CPI} = 5.8}$