Answer 1)
1.1)
Let the total number of instructions = n
Without forwarding, NOPS per instruction = 0.4
Without forwarding, clock cycle time = 250 ps
With the implicit assumption that the average-CPI of the program is 1,
Execution Time Old (T_old)

= 1.4 * n * 250 ps
= 350n ps

Adding forwarding hardware, NOPS per instruction = 0.05
Adding forwarding hardware, clock cycle time = 300 ps
Execution Time New (T_new)

= 1.05 * n * 300 ps
= 315n ps

Therefore,

Speedup = T_old/T_new
= 350/315
= 1.11 times

1.2)
Adding forwarding hardware, let the NOPS per instruction be p

For the new processor to be faster,

(1 + p) *n*300 <= 350n
1+p<=1.167
p <= 0.167

Therefore, the average NOPS per instruction with forwarding should **be less than 16.7%** of the total number of instructions

1.3)
Replacing 0.4 in 1.2 with x

(1 + p)*n*300 <= (1 + x)*n*250
300+300p<=250+250x
300p <= 250x - 50
p <= (250x - 50)/300

1.4)
When x = 0.075
p <= (250 * 0.075 - 50)/300
p <= -0.1042

We see that, when x = 0.075 NOPS per instruction, there are very few NOPS to begin with for forwarding hardware to improve

Another way to look at this is, adding the forwarding hardware, even if we get rid of all the NOPS, the execution time will be a minimum 300n. But with x=0.075, the execution time without forwarding hardware is 250*1.075n = 268.75n which is lesser with forwarding hardware. Hence, it doesn't make sense to add the hardware.

1.5)
We see that if 250x-50<=0, there will be no use of forwarding hardware
=> 250x <= 50
=> x <= 0.2

Therefore, when NOPS without forwarding is 0.2 per instruction or lesser, it is pointless to add forwarding hardware.

Answer 2)
Given: Instruction Memory and Data Memory are the same

2.1)
The first two instructions use Data Memory to load/store. The third instruction sub doesn't use Data Memory. Hence, even though we can't access instruction memory in cycles C4 and C5, we can access it in C6. We need to introduce two NOP instructions before the beqz instruction to mitigate this hazard.

	Clock Cycle											
Instruction	**C1**	**C2**	**C3**	**C4**	**C5**	**C6**	**C7**	**C8**	**C9**	**C10**	**C11**	**C12**
sd	IF	ID	EX	**MEM**	~~WB~~							
ld		IF	ID	EX	**MEM**	WB						
sub			IF	ID	EX	~~MEM~~	WB					
beqz				...	...	IF	ID	EX	~~MEM~~	~~WB~~		
add							IF	ID	EX	~~MEM~~	WB	
sub								IF	ID	EX	~~MEM~~	WB

2.2)
To remove the structural hazards above, we need to make sure that there are no instructions in the Instruction Fetch IF stage when load and store instructions are using the Data Memory MEM. The only way to do that is to move the load/store instructions at the end. But we can do that only if the branch label isn't using register x29 or modifying the words stored at offsets 8 and 12 of MEM x16.

In general, every third instruction after a load/store instruction will face a structural hazard in this processor and it will be impossible to mitigate it. So, we would need to introduce NOPs in this case.

2.3)
To address this structural hazard in the hardware, we need to have separate memory arrays for Instruction and Data Memory.

If we want to handle this in the code, as stated in 2.2, every third instruction after a load/store should be a NOP.

In the given set of instructions, we can mitigate it by adding two NOPs before beqz. The set of instructions then become:

```
sd x29, 12(x16)
ld x29, 8(x16)
sub x17, x15, x14
NOP
NOP
beqz x17, label
add x15, x11, x14
sub x15, x30, x14
```

But, it is better to handle this in hardware, as adding NOP will significantly increase the execution time.

2.4)
Let the total number of instructions be n

Given the instruction mix, there are a total of 36% instructions which are load/store and require Data Memory.

Hence, we can approximate that 0.36n NOPs would be added since for every load/store instruction we need to add 1 NOP after 3 instructions.

Answer 3)
The pipeline has only 4 stages now: IF, ID, EX/MEM, WB

3.1)
Since the clock cycle time is dependent on the latency of the slowest ones of the 5 stages : IF, ID, EX, MEM and WB, combining the EX and MEM stage would have **no effect on the clock cycle time**.

3.2)
The improvement in performance can be attributed to two reasons:

1. Since, the pipeline only has 4 stages now, **the latency of an average instruction would decrease,** reducing the overall execution time of the program.
2. Also, data hazards stemming from load/store instructions and an instruction that uses the data memory can be removed. We **won't have to add NOP instructions which reduces the penalty time in Execution Time from the data hazards**.

3.3)
However, the removal of offset from load/store instructions leads to an **additional addi instruction to calculate the offset**. If the instruction mix is such that there are large number of load/store instructions, it will get doubled because of the additional addi instruction. This increases the execution time of the program and degrades the performance.

Answer 4)
Analyzing the instruction stages at different clock cycles

Choice 1:

	Clock Cycle							
Instruction	**C1**	**C2**	**C3**	**C4**	**C5**	**C6**	**C7**	**C8**
ld x11, 0(x12)	IF	ID	EX	**MEM**	WB			
add x13, x11, x14		IF	ID	**EX**	...	~~MEM~~	WB	
or x15, x16, x17			IF	ID	...	EX	~~MEM~~	WB

The content of x11 becomes available in the first instruction after the completion of C4. However, the second instruction - add requires x11 in its EX stage which is in C4 as well. So, there's a load-use-data hazard.

Choice 2:

	Clock Cycle							
Instruction	**C1**	**C2**	**C3**	**C4**	**C5**	**C6**	**C7**	**C8**
ld x11, 0(x12)	IF	ID	EX	**MEM**	WB			
add x13, x11, x14		IF	ID	...	**EX**	~~MEM~~	WB	
or x15, x16, x17			IF	...	ID	EX	~~MEM~~	WB

The content of x11 becomes available in the first instruction after the completion of C4. Now, the second instruction - add requires x11 in its EX-stage which is in C5. Thus, adding a stall in C4 in the second instruction effectively removes the data hazard.

Hence, **Choice 2** best describes the use of pipeline's hazard detection unit.

Answer 5)
Given:
Perfect Branch Prediction is used - This means we do not lose any cycles due to branch hazards. The branch is always predicted and taken correctly.
Full forwarding support is assumed - Therefore, data hazards that can be resolved with forwarding do not stall the pipeline.
Branches are resolved in the EX-stage as opposed to the ID stage

5.1)
Pipeline execution diagram for first two iterations of the loop.
To resolve the data hazard for the add instruction, stalls are introduced.

	Clock Cycle															
INST	**C1**	**C2**	**C3**	**C4**	**C5**	**C6**	**C7**	**C8**	**C9**	**C10**	**C11**	**C12**	**C13**	**C14**	**C15**	**C16**
loop: ld x10, 0(x13)	IF	ID	EX	**MEM**	WB											
ld x11, 8(x13)		IF	ID	EX	**MEM**	WB										
add x12, x10, x11			IF	ID	...	EX	~~MEM~~	WB								
subi x13, x13, 16				IF	...	ID	EX	~~MEM~~	WB							
bnez x12, loop					...	IF	ID	EX	~~MEM~~	~~WB~~						
loop: ld x10, 0(x13)							IF	ID	EX	**MEM**	WB					
ld x11, 8(x13)								IF	ID	EX	**MEM**	WB				
add x12, x10, x11									IF	ID	...	EX	~~MEM~~	WB		
subi x13, x13, 16										IF	...	ID	EX	~~MEM~~	WB	
bnez x12, loop											...	IF	ID	EX	~~MEM~~	~~WB~~
All Stages Utilitzed?	–	–	–	–	N	N	N	N	N	N	N	N	–	–	–	–

5.2)
There are no clock cycles where the pipeline is fully utilized. Stages not used in an instruction are striked through.

Answer 6)
6.1)
Case1: EX to 1st only
add x10, x13, x14
or x11, x10, x8
sub x15, x16, x17

Case 2: MEM to 1st only
ld x10, 0(x13)
or x11, x10, x8
sub x15, x16, x17

Case 3: EX to 2nd only
add x10, x13, x14
or x11, x9, x8
sub x15, x10, x17

Case 4: MEM to 2nd only
ld x10, 0(x13)
or x11, x16, x8
sub x15, x10, x17

Case 5: EX to 1st and 2nd
add x10, x13, x14
or x11, x10, x8
sub x15, x10, x17

6.2)
Case 1 : EX to 1st only
add x10, x13, x14
NOP
NOP
or x11, x10, x8
sub x15, x16, x17

Case 2 : MEM to 1st only
ld x10, 0(x13)
NOP
NOP
or x11, x10, x8
sub x15, x16, x17

Case 3 : EX to 2nd only
add x10, x13, x14
or x11, x9, x8
NOP
sub x15, x10, x17

Case 4 : MEM to 2nd only
ld x10, 0(x13)
or x11, x16, x8
NOP
sub x15, x10, x17

Case 5 : EX to 1st and 2nd
add x10, x13, x14
NOP
NOP
or x11, x10, x8
sub x15, x10, x17

6.3)
In the cases mentioned in 6.2, if we analyze the sequence of instructions independently, the number of NOPs to be added would be greater than the number of NOPs actually introduced to avoid data hazard.

For example, in Case 4, if we take the first instruction ld which loads x10 in the first half of its WB stage (5th stage) and take the third instruction sub which uses x10 in the second half of its ID stage (2nd stage), we would need to introduce 2 NOP instructions. However, introducing only one NOP resolves the data hazard.

6.4)
We see that from 6.2 in case of no forwarding,
Number of NOPs introduced in Case 1 = 2
Number of NOPs introduced in Case 2 = 2
Number of NOPs introduced in Case 3 = 1
Number of NOPs introduced in Case 4 = 1
Number of NOPs introduced in Case 5 = 2

Hence the average NOP per instruction is:
= 0.05 * 2 + 0.2 * 2 + 0.05 * 1 + 0.1 * 1 + 0.1 * 2
= 0.1 + 0.4 + 0.05 + 0.1 + 0.2
= 0.85

Therefore, CPI for the program
= 1 + 0.85 = **1.85**

Now, percentage of cycles that are NOPs are
= 0.85/1.85
= 45.95 %

6.5)
In case we enable full forwarding,
Case 1 is resolved by forwarding from end of EX stage of add to beginning of EX stage of or
Case 2 cannot be resolved by forwarding because the output of MEM stage is required in the beginning of EX stage. We need to have atleast one NOP even with forwarding.
Case 3 is resolved by forwarding from the end of MEM stage of add to beginning of EX stage of sub
Case 4 is resolved in the same way as Case 3 by forwarding from the end of MEM stage of ld to beginning of EX stage of sub
Case 5 is resolved by forwarding
1) from the end of EX stage of add to beginning of EX stage of or
2) the end of MEM stage of add to beginning of EX stage of sub

So, the only RAW dependency that cannot be handled with forwarding is for Case 2. We need to introduce 1 NOP to mitigate the hazard.

Hence the average NOP per instruction is:
= 0.2 * 1
= 0.2

Therefore, CPI for the program = 1 + 0.2 **= 1.2**

Now, percentage of cycles that are NOPs are
= 0.2/1.2
= 16.67 %

6.6)
In case we enable EX/MEM (next-cycle) forwarding,
Case 1 is resolved by forwarding from end of EX stage of add to beginning of EX stage of or
Case 2 cannot be resolved by forwarding because the output of MEM stage is required in the beginning of EX stage. We need to have two NOPs even with EX/MEM forwarding.
Case 3 cannot be resolved by forwarding from the end of MEM stage of add to beginning of EX stage of sub. We need to have one NOP.
Case 4 cannot be resolved by forwarding from the end of MEM stage of ld to beginning of EX stage of sub. We need to have one NOP.
Case 5 is partially resolved by forwarding from the end of EX stage of add to beginning of EX stage of or But, from the end of MEM stage of add to beginning of EX stage of sub there's no forwarding, hence we need one NOP.

Hence the average NOP per instruction is:
= 0.2 * 2 + 0.05 * 1 + 0.1 * 1 + 0.1 * 1
= 0.65

Therefore, CPI for the program = 1 + 0.65 = **1.65**
Now, percentage of cycles that are NOPs are = 0.65/1.65
= **39.39 %**
In case we enable **MEM/WB (two-cycle) forwarding**,

Case 1 cannot be resolved by forwarding from end of EX stage of add to beginning of EX stage of or. But, we can forward the result from the MEM stage of add to the EX stage of or. Hence, we need 1 NOP.
Case 2 cannot be resolved by forwarding because the output of MEM stage is required in the beginning of EX stage. We need to have atleast one NOP even with MEM/WB forwarding.
Case 3 is resolved by forwarding from the end of MEM stage of add to beginning of EX stage of sub
Case 4 is resolved by forwarding from the end of MEM stage of ld to beginning of EX stage of sub.
Case 5 is partially resolved by forwarding from the end of EX stage of add to beginning of EX stage of sub But, from the end of EX stage of add to beginning of EX stage of or there's no forwarding, hence we need one NOP.

Hence the average NOP per instruction is:

= 0.05 * 1 + 0.2 * 1 + 0.1 * 1
= 0.35

Therefore, CPI for the program = 1 + 0.35 **= 1.35**

Now, percentage of cycles that are NOPs are
= 0.35/1.35
= 25.93 %

6.7)
In case of **No Forwarding**,
Clock Cycle = max(IF, ID, EX(no FW), MEM, WB)
= max(120ps, 100ps, 110ps, 120ps, 100ps)
= 120ps

Therefore, Execution Time per instruction = 1.85 * 120
= 222 ps

In case of **Full Forwarding**,
Clock Cycle = max(IF, ID, EX(full FW), MEM, WB)
= max(120ps, 100ps, 130ps, 120ps, 100ps)
= 130ps

Therefore, Execution Time per instruction = 1.2 * 130
= 156 ps

Hence, speedup = 222/156 = **1.42 times or 42%**

In case of **EX/MEM (next-cycle) forwarding**,
Clock Cycle = max(IF, ID, EX(FW from EX/MEM only), MEM, WB)
= max(120ps, 100ps, 120ps, 120ps, 100ps)
= 120ps

Therefore, Execution Time per instruction = 1.65 * 120
= 198 ps

Hence, speedup = 222/198 = **1.12 times or 12%**

In case of **MEM/WB (two-cycle) forwarding**,
Clock Cycle = max(IF, ID, EX(FW from MEM/WB only), MEM, WB)
= max(120ps, 100ps, 120ps, 120ps, 100ps)
= 120ps
Therefore, Execution Time per instruction =1.35*120 = 162 ps

Hence, speedup = 222/162 = **1.37 times or 37%**

6.8)
The full forwarding processor is the fastest with a CPI of 1.2 and clock cycle time of 120ps. The execution time per instruction is 156 ps.

If we added “timetravel” forwarding, that means we are effectively eliminating all hazards. Hence, the CPI remains 1 as we don’t have to introduce any NOP stalls. But, the latency of the EX stage becomes 130 (EX full FW) + 100 ps = 230 ps

Since, this is the highest latency among all the stages of the pipeline, this is also the clock cycle.

Hence, Execution time = 1 * 230
= 230 ps

Therefore, speedup = 156 / 230 = **0.68 times**

Hence, timetravel forwarding resulting in 0 hazards actually slows the processor from full forwarding.

Answer 7)
7.1)
In case of no forwarding,

```
add x15, x12, x11
NOP
NOP
ld x13, 4(x15)
ld x12, 0(x2)
NOP
or x13, x15, x13
NOP
NOP
sd x13, 0(x15)
```

First two NOPs are introduced, since x15 is set after WB stage of add and ld requires x15 in the beginning of execution stage

Next NOP is introduced because x13 is loaded after the WB stage of ld and or requires x13 in the EX stage.

Next two NOPs are introduced since x13 is calculated and available from the or instruction in its WB stage and sd requires x13 to store it in the ID stage.

7.2)
In case we try to rearrange the code, the only seems to be swap the two ld instructions. Doing that, reduces first set of NOP to 1 but increases the next set of NOP to 2. Hence, rearranging the code has no effect on the number of NOPs required.

7.3)
If we implement forwarding but forget the hazard detection unit responsible for load-use-data hazards, the program will run correctly. The only problem for no hazard detection unit would have been ld x13, 4(x15) where it is writing to x13 and or x13, x15, x13 where it is using x13, but there is already a ld instruction between them. So, there won't be any issues.

7.4)
Clock Cycle 1:
ForwardA = XX, ForwardB = XX
Since we don't have any instructions in EX stage yet

Clock Cycle 2:
ForwardA = XX, ForwardB = XX
Since we don't have any instructions in EX stage yet

Clock Cycle 3:
ForwardA = 00, ForwardB = 00
All the values are taken from the registers and there's no forwarding

Clock Cycle 4:
ForwardA = 10, ForwardB = 00
We need the result of the previous instruction as it is used in this cycle. So forwarding required

Clock Cycle 5:
ForwardA = 01, ForwardB = 01
We need the results of two previous instructions as it is used in this cycle as base registers

Clock Cycle 6:
ForwardA = 00, ForwardB = 10
We need the result of the ld instruction which was two instructions ago as it is used in the rs2

Clock Cycle 7:
ForwardA = 00, ForwardB = 10
We need data which is already written in the WB stage from the register file.

7.5)
In case of no forwarding, for the hazard detection unit to remove load-use-data hazards, we need to **know the value of rd that is the output of MEM/WB Register**. The instruction that is currently in the ID stage needs to be stalled if it depends on a value produced by (or forwarded from) the instruction in the EX or the instruction in the MEM stage. So we need to check the destination register of these two instructions. The Hazard unit already has the value of rd from the EX/MEM register as inputs, so we need only add the value from the MEM/WB register.

The three signals of the output of hazard detection unit - PCWrite, IF/IDWrite and Mix Control would suffice to introduce any stalls in the pipeline that are needed. Hence, **no need of any additional output signals.**

The value of rd from EX/MEM is needed to detect the data hazard between the add and the following ld. The value of rd form MEM/WB is needed to detect the data hazard between the first ld instruction and the or instruction.

7.6)
In 6.5, we dealt with full forwarding. Since we have stalls in C4 and C5 for the load registers to use the correct rd, for C4 and C5 the Mux Control should take input from the Control Unit.

Clock Cycle	PCWrite	IF/IDWrite	Mux Control
C1	1	1	0
C2	1	1	0
C3	1	1	0
C4	0	0	1
C5	0	0	1