1. Answer the following questions based on the following table. All different datapaths support the following four instruction types with listed delay of each component. Assume no hazard is detected in the pipelined datapath. Please explain your answer.

Instruction class	Instruction fetch	Register read	ALU operation	Data access	Register write	Total time
Load word (`lw`)	200 ps	100 ps	200 ps	200 ps	100 ps	800 ps
Store word (`sw`)	200 ps	100 ps	200 ps	200 ps		700 ps
R-format (`add`, `sub`, `and`, `or`, `slt`)	200 ps	100 ps	200 ps		100 ps	600 ps
Branch (`beq`)	200 ps	100 ps	200 ps			500 ps

a) What should clock cycle time be for the single-cycle, multi-cycle and pipelined datapaths? Please explain your answer.
- Single-cycle: 800 ps (Obtained from the total time, and in this case *lw* is considered as it's the instruction with the largest latency)
- Multi-cycle: 200 ps (Obtained from the maximum of the largest latency contributed by each component/unit)
- Pipelined: Same as that of multi-cycle (assuming that the intermediate registers have 0 access time)

b) What is the latency (defined as the delay from when the instruction enters the datapath until it finishes) of an R-format instruction in the single-cycle, multi-cycle and pipelined datapaths?
- Single-cycle: 800 ps: As decided from the previous case, no matter what type of instruction it is, it takes the time equal to the *lw* instruction.
- Multi-cycle: 800 ps- Equal to the total time the type of instruction takes, which is 4 clock cycles.
- Pipelined: 1000 ps – As the instruction also has to go through the data access stage. So 200 ps * 5 = 1000 ps

c) What is the latency (defined as the delay from when the instruction enters the datapath until it finishes) of load word (lw) instruction in the single-cycle, multi-cycle and pipelined datapaths?
- Single-cycle: 800 ps: As decided from a)
- Multi-cycle: 1000 ps – Equal to the total time of *lw* instruction, which is 5 clock cycles.
- Pipelined: 1000 ps – The instruction has to go through all the stages. 200 ps * 5 = 1000 ps.

d) What is the averaged throughput (defined as the number of instructions executed in 1 nanosecond in this case) of the single-cycle, multi-cycle and pipelined datapaths? Assume the frequency of four different instruction types are the same.
- Single-cycle: One instruction is completed for every 800 ps.
  Therefore $throughput = \frac{1}{800\ ps} = 1.25 * 10^9\ instructions/sec$
- Multi-cycle: As the frequency of occurrence of each instruction is the same, the average execution time of the instructions $= \frac{(5+4+4+3)200\ ps}{4} = 800\ ps$
  Therefore $throughput = \frac{1}{800\ ps} = 1.125 * 10^9\ instructions/sec$
- Pipelined: The following calculation assumes the case when the pipeline is full, in which case an instruction is completed in each clock cycle (200 ps)
  Therefore $throughput = \frac{1}{200\ ps} = 5 * 10^9\ instructions/sec$

2. Suppose that an unpipelined processor has a cycle time of 25 ns, and its datapath consists of modules

with the latencies of 2, 3, 4, 7, 3, 2, and 4 ns (in this specific order). In pipelining this processor, it is not possible to rearrange the order of the modules (for example, putting the register read stage before the instruction decode stage) or to divide a module into multiple pipeline stages (for complexity reasons). Note that, this is not standard five-stage pipeline design.

a) What is the minimum clock cycle time that can be achieved by pipelining this processor? Please explain your answer.
   As we cannot ignore any of the pipeline stages, the minimum clock cycle time is equal to the maximum time contributed by each of the stages. So, in this case 7 ns.

b) If you are limited to a 2-stage pipeline, what is the minimum clock cycle time? Please explain your answer.
   The approach to this solution would be to combine a few of the stages into one single combinational stage.
   Case-I: (2 + 3 + 4 + 7) | (3 + 2 + 4) = 16 | 9. In this case, the clock cycle time would be 16 ns
   Case-II: (2 + 3 + 4) | (7 + 3 + 2 + 4) = 9 | 16. Even in this case, the clock cycle time would be 16 ns
   Either of these two cases can be used to achieve a 2-stage pipeline with minimum clock cycle time = 16 ns

3. The following codes run on the five-stage pipelined datapath. For each piece of code, answer the following questions.

a) Without forwarding, insert the necessary number of 'nops' for the code to execute correctly.
b) With the forwarding unit available (only supports forwards MEM-> EX and WB->EX), show how the code will execute. Mention the data that's forwarded between the stages. Use 'nops' only when necessary.

```
Code 1:
sub t1, t2, t3
add t1, t4, t5
or t4, t2, t6
```

There is a WAW dependency between *sub* and *add* instructions. Also, a WAR dependency between *add* and *or* instructions.

a) No *nops* are necessary
b) No data needs to be forwarded and no *nops* necessary

```
Code 2:
and t2, t5, t1
sub t3, t2, t0
nor t7, t1, t5
```

There is a RAW dependency between *and* and *sub* instructions. *nor* is completely independent of the above two instructions.

a) The value of t2 won't be available until the WB stage of *and* is completed. So, until then, the ID stage of *sub* must wait so that the correct value of t2 can be used by *sub*. Hence we have to delay the instruction with two clock cycles by inserting *nops* as shown below:

```
and t2, t5, t1 IF ID EX ME WB
nop              **
```

```
nop                  **
sub t3, t2, t0          IF ID EX ME WB
nor t7, t1, t5             IF ID EX ME WB
```

By doing so, the WB stage of and will now be synchronized with the ID stage (assuming that registers can be written and read in the same clock cycle).

b) When forwarding is enabled, the value can be forwarded from MEM->EX, in which case no *nops* are necessary, as the data would be available in the MEM stage itself.

```
and t2, t5, t1 IF ID EX ME WB
sub t3, t2, t0    IF ID EX ME WB ; t2 forwarded from MEM->EX
nor t7, t1, t5       IF ID EX ME WB
```

Code 3:

```
and t2, t5, t1
sub t3, t2, t0
nor t7, t2, t1
```

In this, there is a RAW dependency between the *and* and *sub* and *nor*.

a) *nops* have to be inserted in the following way:

```
and t2, t5, t1    IF ID EX ME WB
nop                  **
nop                     **
sub t3, t2, t0             IF ID EX ME WB
nor t7, t2, t1                IF ID EX ME WB
```

b) With register forwarding in place, there is no need for *nop:*

```
and t2, t5, t1    IF ID EX ME WB
sub t3, t2, t0       IF ID EX ME WB ; t2 forwarded from MEM->EX
nor t7, t2, t1          IF ID EX ME WB ; t2 forwarded from WB->EX
```

Code 4:

```
lw t1, 22(t0)
and t2, t1, t3
```

In this, there is a RAW dependency between the two instructions.

a) *nops* has to be inserted in the following way:

```
lw t1, 22(t0)     IF ID EX ME WB
nop                  **
nop                     **
and t2, t1, t3             IF ID EX ME WB
```

b) A single *nop* is required here, as the data can be forwarded from WB->EX stage, as the data won't be ready until we reach WB stage

```
lw t1, 22(t0)     IF ID EX ME WB
nop                  **
and t2, t1, t3          IF ID EX ME WB ; t1 forwarded from WB->EX stage
```

```
Code 5:
lw t1, 22(t0)
sub t6, t0, t2
xor t3, t1, t5
```

There is a RAW hazard between the first and the third instructions.

a) A single *nop* should suffice in this case:

```
lw t1, 22(t0)      IF ID EX ME WB
sub t6, t0, t2        IF ID EX ME WB
nop                      **
xor t3, t1, t5              IF ID EX ME WB
```

b) With forwarding enabled, the data can be forwarded from WB->EX stage as shown, without introducing any *nops*

```
lw t1, 22(t0)      IF ID EX ME WB
sub t6, t0, t2        IF ID EX ME WB
xor t3, t1, t5           IF ID EX ME WB ; t1 forwarded from WB->EX
```

4. For the standard five-stage pipelined datapath, there are three different branch handling mechanisms below. Explain each strategy for the two approaches of implementing the 'beq' (i.e., in ID stage or in MEM stage). Also, for each strategy, answer the following questions:

- How many 'nops' are used if the branch is taken?
  a) ID: 2 nops
     MEM: 4 nops
  b) Assuming that the branch prediction strategy uses "taken", no nop will be needed.
  c) If sufficient number of independent instructions are available that can be safely executed, then no nop is needed.
- How many 'nops' are used of the branch is not taken?
  a) ID: 2 nops
     MEM: 4 nops
  b) ID: Assuming that the branch prediction strategy uses "taken", 2 nops will be needed.
     MEM: Assuming that the branch prediction strategy uses "taken", 4 nops will be needed.
  c) If sufficient number of independent instructions are available that can be safely executed, then no nop is needed.
- Is this solution always applicable? Explain.
  a) Yes, the solution is always applicable as it's a static strategy. We need not scan for further instructions.
  b) Yes, the solution is always applicable.
  c) Depends on the availability of independent instructions that can be safely executed with delayed branching.

a) *stall-on-branch* strategy for implementing the 'beq' instruction.
b) *branch prediction* strategy for implementing the 'beq' instruction.

c) *delayed branch* strategy for implementing the 'beq' instruction.

5. If the 'beq' instruction is implemented in the ID stage, how do the codes below execute correctly? Use forwarding and minimal number of 'nops'. Clearly place 'nops' within the codes and explain where do forwarding(s) take place.

```
Code 1:
sub t1, t2, t3
beq t1, t2, label
```

There is a RAW dependency between beq and sub on t1. No nop will be needed by utilizing EX->ID forwarding:

```
sub t1, t2, t3    IF ID EX ME WB
beq t1, t2, label    IF ID EX ME WB ; t1 is forward from EX to ID
```

```
Code 2:
sub t1, t2, t3
add t0, t2, t3
beq t1, t2, label
```

There is a RAW dependency between beq and sub on t1. No nop will be needed by utilizing MEM->ID forwarding:

```
sub t1, t2, t3    IF ID EX ME WB
add t0, t2, t3       IF ID EX ME WB
beq t1, t2, label       IF ID EX ME WB ; t1 is forward from MEM to ID
```

```
Code 3:
lw t1, 5(s0)
beq t1, t0, label
```

There is a RAW dependency between beq and lw on t1. One nop will be needed by utilizing MEM->ID forwarding:

```
lw t1, 5(s0)      IF ID EX ME WB
nop                  **
nop                     **
beq t1, t0, label          IF ID EX ME WB ; t1 is forward from WB to ID
```

6.) For five-stage pipelined datapath, please explain each line of the following code and fill "_____":

```
if ( EX/MEM.RegWrite==1 and
     EX/MEM.RegisterRd != 0 and
     EX/MEM.RegisterRd == ID/EX.RegisterRs )
then, forward from ____MEM_____(which stage?) to ___RS operand of ID/EX
stage_____(which operand of which stage?)
```

7. (For five-stage pipelined datapath, please explain each line of the following code and fill "_____":

```
if ( MEM/WB.RegWrite==1 and
     MEM/WB.RegisterRd != 0 and
     EX/MEM.RegisterRd != ID/EX.RegisterRs and
     MEM/WB.RegisterRd == ID/EX.RegisterRs )
then, forward from _____WB_____(which stage?) to ___RS operand of ID/EX
stage_______(which operand of which stage?)
```

8. The figure below shows how the forwarding unit sets the multiplexers to forward from the MEM or WB stages to the EX stage.

a) What are the values of "ForwardA" and "ForwardB" when the "sub" instruction in EX stage:

```
add $2, $1, $4
sub $3, $2, $1
```

ForwardA = 10
ForwardB = 00

b) What are the values of "ForwardA" and "ForwardB" when the "and" instruction in EX stage:

```
lw $2, 5($0)
sub $1, $3, $4
and $5, $2, $4
```

ForwardA = 01
ForwardB = 00