1. Pre-check

This section is designed as a check to help you determine whether you understand the concepts covered in class. Please answer "true/false" to the following questions and include an explanation.

1.1. Introducing “pipeline” registers to the Datapath induces higher latency and throughput in the execution of instructions.
TRUE! Latency is the time it takes for an instruction to complete, while throughput is the number of instructions processed per unit of time. The introduction of pipelining in a processor results in higher throughput because more instructions are executed at once. Latency is also higher because each individual instruction takes longer from start to finish.

1.2. "Data hazards" typically result in three pipeline stalls if we don’t use the transfer unit.
TRUE! The next instruction must wait for the preceding instruction to complete the EX, MEM, and WB stages before it can use the EX stage of the pipeline.

1.3. All "data hazards" can be resolved using the “transfer unit”.
FALSE! The hazards that may arise following a read instruction (i.e. `lw`, `...`) cannot be completely resolved with the “transfer unit” because the data is only known at the MEM stage. Therefore, a stall will occur in this case.

1.4. Suspending the pipeline is the only way to resolve the "branch hazards."
FALSE! Pipeline suspension is one way to resolve branch hazards. Other more advanced techniques attempt to predict the path the instruction will take after a branch instruction and flush the pipeline if the prediction turns out to be wrong.

2. Pipeline Registers

2.1. In order to convert a single-cycle processor into a pipelined processor, registers are introduced between the different stages of the data path. What is the role of these registers?
In a pipelined data path, the values for each stage must be passed at each clock cycle. Each stage of the pipeline only operates on a small set of values, but those values must be correct for the instruction being processed. Consider the load instruction "`lw`": if the instruction is in the EX stage, then the values in that stage should look like the values of the EX stage in a single-cycle data path (i.e. the values of the `rs` and `rt` bit fields as well as the immediate "`imm`" must be as if the "`lw`" instruction was the only executing instruction in the entire path).

This also includes “the control signals”: Indeed, the appropriate control signals are generated during the “instruction decoding” stage and to ensure that each stage has the correct signals for its execution, these signals must be passed through the pipeline registers too.

2.2. Why do we need to save the “control signals” multiple times in the pipeline?
Because each pipeline stage must receive the correct control signals for the instruction currently at that stage.

3. Performance Analysis

Before continuing with the questions in this tutorial, please have a look at the document "**Timing Constraints for digital circuits**" (available on the course website) for a quick note on the time keywords (i.e. t_{cpq}, t_{pd}, …) used here.

Let's assume the following time durations for our pipelined CPU:

t_{cpq} register :	30 ps ,	t_{setup} register :	20 ps
t_{pd} Mux:	25 ps ,	t_{pd} ALU:	200 ps
t_{cpq} Mem:	250 ps ,	t_{setup} MEM:	200 ps
t_{cpq} RegFile:	150 ps ,	t_{setup} RegFile:	20 ps

3.1. With the delays shown above for each of the Datapath components, what would be the fastest possible clock time for a single-cycle Datapath?

$$T_c \geq t_{cpq}(PC) + t_{cpq}(IMEM) + t_{cpq}(RegFile) + t_{pd}(Mux) + t_{pd}(ALU) + t_{cpq}(DMEM) + t_{pd}(Mux) + t_{setup}(RegFile)$$

$$T_c \geq 950ps$$

$$f_c \leq \frac{1}{T_c} \cong 1.05\ GHz$$

Note that the Mux delay before the "Registers File" is omitted here because in the case of a load instruction (*i.e.* our critical path), the output of this component is required much later in the data path (i.e. after WB). This leaves plenty of time for the output of this Mux to stabilise for the "Write Register" input of the "Registers File ".

3.2. What would be the fastest possible clock frequency for a pipelined data path?

IF : $t_{cpq}(PC) + t_{cpq}(IMEM) + t_{setup}(Regs\_IF{:}ID) = 300\ ps$

ID : $t_{cpq}(Regs\_IF{:}ID) + t_{cpq}(RegFile) + t_{setup}(Regs\_ID{:}EX) = 200\ ps$

EX : $t_{cpq}(Regs\_ID{:}EX) + t_{pd}(Mux) + t_{pd}(UAL) + t_{setup}(Regs\_EX{:}MEM) = 275\ ps$

MEM : $t_{cpq}(Regs\_EX{:}MEM) + t_{cpq}(DMEM) + t_{setup}(Regs\_MEM{:}WB) = 300\ ps$

WB : $t_{cpq}(Regs\_MEM{:}WB) + t_{pd}(Mux) + t_{setup}(RegFile) = 75\ ps$

$$max(IF, ID, EX, MEM, WB) = 300\ ps \quad \Rightarrow \quad f_c = \frac{1}{300\ ps} \cong 3.33\ GHz$$

3.3. What is the speedup after converting the single-cycle Datapath to a pipelined Datapath? Why is the speedup less than 5?

A speedup of $\frac{950\ ps}{300\ ps} \cong 3.17$ times. The speedup is less than 5 because the transfer registers (i.e. the pipeline registers) introduce additional delays (t_{cpq} and t_{setup}) to the Datapath. On the other hand, it is necessary to set the clock to the maximum time required among the five stages.

Note: Resolving potential hazards require additional circuitry that would further impact the speedup.

4. Execution hazards

Converting a single-cycle data path to a pipelined version introduces three types of execution hazards: structural hazards, data hazards, and branch hazards.

Structural hazards

They occur when multiple instructions need to use the same resource in the data path at the same time. There are two main causes of structural hazards:

RegFile Access: The "registers file" is accessed both during the ID stage, when it is read, and during the WB stage, when it is modified (i.e. written to). This problem is solved by having separate read and write ports. In the case of simultaneous reading and writing to the same register, one writes to the register during the first half of the clock cycle and reads from it during the second half. This is also known as "double pumping."

Memory Access: Memory holds both the instructions of the program to be executed (accessed during the IF stage) and the data to be processed (required during the MEM stage). Having a separate instruction memory (IMEM) and data memory (DMEM) resolve the problem of simultaneous access to this resource by different instructions in the pipeline. This design is based on an old architecture called "Harvard architecture" and is implemented today by using separate memory caches for instructions and data (cf. the lecture on memory caches).

☞ **Structural hazards can always be solved by adding more hardware components.**

Data hazards

These hazards are caused by data dependencies between instructions. In particular, a data hazard occurs when an instruction reads from a register before a previous instruction has finished writing to that register.

Transfer Unit

Most data hazards can be resolved by transfer, i.e., when the output of the EX stage or the MEM stage is transferred directly to the EX stage of the subsequent instruction.

4.1. Point out the data hazards in the code below and indicate how data transfer could be used to resolve them.

Cycles / Instructions	C1	C2	C3	C4	C5	C6	C7
`1. addi $t0, $a0, -1`	IF	ID	EX	MEM	WB		
`2. and $s2, $t0, $a0`		IF	ID	EX	MEM	WB	
`3. sltiu $a0, $t0, 5`			IF	ID	EX	MEM	WB

There are two data hazards: Between instructions 1 and 2 and between instructions 1 and 3. The first hazard could be solved by transferring the result of the EX stage in C3 to the input of the EX stage in C4. The second hazard could be solved by transferring the result of the EX stage in C3 to the beginning of the EX stage in C5.

4.2. How many instructions after an `addi` statement could be affected by data hazards related to that statement?

Three instructions. Consider instruction 1 in question 4.1. Any subsequent statement that would read register `$t0` (i.e. during the ID stage) in the C3, C4, or C5 clock cycles will not have the correct value of `$t0` because the `addi` statement will only update it in cycle C5. If, however, the Regfile registers possess the "double pumping" feature, then only two instructions will be affected because the ID stage of instruction 4 could be executed at the same time as the WB stage of instruction 1.

Pipeline suspension

4.3. Identify the data hazards in the code below. Why can't one of these hazards be resolved with the transfer unit? What can we do to resolve this issue?

Cycles / Instructions	C1	C2	C3	C4	C5	C6	C7	C8
`1. addi $s0, $s0, 1`	IF	ID	EX	MEM	WB			
`2. addi $t0, $t0, 4`		IF	ID	EX	MEM	WB		
`3. lw $t1, 0($t0)`			IF	ID	EX	MEM	WB	
`4. add $t2, $t1, $0`				IF	ID	EX	MEM	WB

There are two data hazards in this code. The first hazard lies between statements 2 and 3 (register `$t0`), and the second between statements 3 and 4 (register `$t1`). The hazard between statements 2 and 3 can be solved with the transfer unit, but the problem between statements 3 and 4 cannot be solved with this unit (only). Indeed, even if the transfer unit were used, instruction 4 needs the result of instruction 3 at the beginning of cycle C6, but the value of `$t1` will only be available at the end of this cycle.

We can solve this problem by inserting a bubble into the pipeline between statements 3 and 4 (i.e., a **no-operation (nop)** instruction).

4.4. In a team of engineers, you are working on the design of a compiler for a MIPS processor (remember, it is the compiler that produces assembly language files). How could this compiler reorganise the instructions in Exercise 4.3 to minimise data hazards while ensuring the same result.

Because instruction 1 has no dependencies, rearrange the instructions as follows: 2–3–1–4.

Data Hazard Detection

Suppose we have the signals `rs`, `rt`, `RegWEn`, and `rd` for two instructions at times t and $t + 1$. We want to check if there is a data hazard between these statements. In this sense, we can verify whether the `rd` of the instruction at time t matches `rs` or `rt` of the instruction at time $t + 1$, thereby indicating a data hazard. We could then use this detection to decide which transfer paths to take or the hold period (if any) to apply to ensure correct execution of the instructions. In pseudocode, this gives:

```
if (rs(n + 1) == rd(n) || rt(n + 1) == rd(n) && RegWEn(n) == 1) {
    /* transfer the output of the ALU of instruction n */
}
```

Branch hazards

Branch hazards are caused by jump and branch instructions. Indeed, in a sequential execution without branching and without jumping, the next instruction to execute is located at the address given by PC + 4. In the case of jump instructions and for some branch instructions, the next PC is not PC + 4, but will be the result of the calculation performed in EX stage. A possible solution to this problem is to suspend the pipeline during a branch hazard, but this will negatively impact performance.

4.5. Besides suspending the pipeline, what can we do to resolve branch hazards?

We can try to predict which direction the branch will go, and when that prediction is incorrect, we "flush" the pipeline and continue with the correct instruction. A simple (and naive) method of prediction is to assume that jumps in branch instructions are never made.

4.6. How many execution hazards would there be in the MIPS code below if it were executed in a pipelined processor **without** a transfer unit? What is the type of each execution hazard? (Examine all possible instruction pairs for possible hazards).

How many times should the pipeline be paused to resolve possible data hazards? What about branch hazards?

Cycles / Instructions	C1	C2	C3	C4	C5	C6	C7	C8	C9
`1. sub $t1, $s0, 1`	IF	ID	EX	MEM	WB				
`2. or $s0, $t0, $t1`		IF	ID	EX	MEM	WB			
`3. sw $s1, 100($s0)`			IF	ID	EX	MEM	WB		
`4. beq $s0, $s2, L1`				IF	ID	EX	MEM	WB	
`5. add $t2, $0, $0`					IF	ID	EX	MEM	WB

We have four execution hazards: between statements 1 and 2 (data hazard on `$t1`), between statements 2 and 3 (data hazard on `$s0`), between instructions 2 and 4 (data hazard on `$s0`), and between statements 4 and 5 (branch hazard).

Assuming that we can read and write to the "Registers File" during the same clock cycle (i.e. using the “double pumping” feature), two bubbles are needed between statements 1 and 2, and two more bubbles between statements 2 and 3. No bubble is required for the branch hazard because it can be handled with branch prediction / pipeline flushing.