Homework Assignment #3
SAMPLE SOLUTION

Q1. (10 points) Please briefly discuss what structural hazard and data hazard are. Please show a sample solution to overcoming each of these hazards.

ANSWER: Structural hazard refers to a situation that a required resource for the pipeline execution is busy.

Data hazard refers to a situation that required data is not ready yet for the pipeline execution, which means the pipeline needs to wait for previous instructions to complete its data read/write.

Structural hazard can be addressed with replicating or separating required resources; for instance, L1 cache is usually separate as L1 instruction cache and L1 data cache.

Data hazard can be addressed with forwarding or bypassing technique.

Q2. (15 points) In this exercise, we examine how pipelining affects the clock cycle time of the processor. Problems in this exercise assume that individual stages have the following latencies:

IF	ID	EX	MEM	WB
250 ps	350 ps	150 ps	300 ps	200 ps

Also, assume that instructions executed by the processor are broken down as follows:

ALU/Logic	Jump/Branch	Load	Store
45%	20%	20%	15%

Lec 10-13

a. What is the clock cycle time in a pipelined and non-pipelined processor?

ANSWER: In a pipelined processor, the clock cycle time would be decided by the longest stage, i.e., the ID stage in this question, 350 ps.

In a non-pipelined, single-cycle processor, the clock cycle time would be decided by the total time required, i.e., 250ps+350ps+150ps+300ps+200ps = 1250ps.

b. What is the total latency of an `lw` instruction in a pipelined and non-pipelined processor?

ANSWER: "Latency" refers to the time between the start of the instruction and the end of the instruction, and an `lw` instruction ends till the result writes back to the register.

In a pipelined processor, since the clock cycle time is 350ps, and each stage would take 350ps, and thus the latency would be 350ps * 5 = 1750ps.

In a non-pipelined single-cycle processor, it would take 1250ps as one cycle takes 1250ps and the `lw` instruction will complete in one cycle.

c. If we can split one stage of the pipelined datapath into two new stages, each with half the latency of the original stage, which stage would you split and what is the new clock cycle time of the processor?

ANSWER: Since the ID stage has the longest latency, the ID stage should be split into two new stages, and each of these two new stages would have 350/2=175ps latency. Now, the longest stage would become the MEM stage with 300ps latency. As the clock cycle time is decided by the longest stage, the new clock cycle time becomes 300ps.

Q3. (5 points) What is the minimum number of cycles needed to completely execute *n* instructions on a CPU with a *k* stage pipeline? Justify your formula.

ANSWER:

ANSWER: As we have a *k*-stage pipeline, it takes *k* cycles for the first instruction to complete. Then, the pipeline is full, and it takes *n* - 1 cycles for the next *n* - 1 instructions to complete. So, the total number of cycles needed to completely execute *n* instructions will be:

$k + (n - 1) = k + n - 1$ cycles.

Q4. (5 points) Assume that x11 is initialized to 11 and x12 is initialized to 22. Suppose you executed the code below on a version of the pipeline that does NOT handle data hazards (i.e., the programmer is responsible for addressing data hazards by inserting NOP (no operations) instructions where necessary). What would the final values of registers x13 and x14 be?

```
addi x11, x12, 5
add x13, x11, x12
addi x14, x11, 15
```

ANSWER: The key to answer this problem is that values are only written back to the destination register when they are in WB (write-back) stage. Therefore, if the pipeline does not handle data hazards, an instruction that uses the result written in an earlier instruction may not get the correct result. For this question, as shown below, the first instruction addi x11, x12, 5 will change x11 to 27 (22 + 5 =27), but the register x11 is only updated in Cycle 5 (in the first-half cycle).

Instruction	Cycle1	Cycle2	Cycle3	Cycle4	Cycle5	Cycle6	Cycle7	
addi **x11**, x12, 5	IF	ID	EX	MEM	WB			
add x13, **x11**, x12		IF	ID	EX	MEM	WB		
addi x14, **x11**, 15			IF	ID	EX	MEM	WB	

Hence, for the instruction add x13, x11, x12, during the decode and register read stage (i.e., ID stage, in Cycle3), x11=11 and x12=22. As such, x13 will get the value of 33 (11 + 22 = 33).

Similarly, for the instruction addi x14, x11, 15, during the decode and register read stage (i.e., ID stage, in Cycle4), x11=11. As such, x14 will get the value of 26 (11 + 15 = 26).

Q5. (5 points) Add NOP instructions to the code below so that it will run correctly on a pipeline that does not handle data hazards.

```
addi x11, x12, 5
add x13, x11, x12
addi x14, x11, 15
add x15, x13, x12
```

ANSWER: if a pipeline does not handle data hazards, and if a data hazard exists, then the earliest cycle for an instruction in the ID stage to read the register with correct result written from an earlier instruction is the cycle when the earlier instruction in the WB stage. This is because write happens in the first-half cycle and read happens in the second-half cycle. So, if a register is written and read in the same cycle, the read can get the correct result from the write. Therefore, as shown below, we need insert 2 NOP instructions between the first and second instructions. We also need to insert 1 NOP instruction between the third and fourth instructions. The dependencies (or data hazards) are also highlighted below.

	CC1	CC2	CC3	CC4	CC5	CC6	CC7	CC8	CC9	CC10	CC11
addi **x11**, x12, 5	IF	ID	EXE	MEM	**WB**						
NOP		NOP	NOP	NOP	NOP	NOP					
NOP			NOP	NOP	NOP	NOP	NOP				
add **x13**, **x11**, x12				IF	**ID**	EXE	MEM	**WB**			
addi x14, **x11**, 15					IF	**ID**	EXE	MEM	WB		
NOP						NOP	NOP	NOP	NOP	NOP	
add x15, **x13**, x12							IF	**ID**	EXE	MEM	WB

Q6. (15 points) Assume we have the following sequence of instructions, and assume that it is executed on a 5-stage pipelined datapath:

```
add x15, x12, x11
lw x13, 8(x15)
lw x12, 0(x2)
or x13, x15, x13
sw x13, 0(x15)
```

a. If there is no forwarding or hazard detection, insert NOPs to ensure correct execution and draw the pipeline execution diagram.

ANSWER: As discussed in Q7, if a pipeline does not handle data hazards, and if a data hazard exists, then the earliest cycle for an instruction in the ID stage to read the register with correct result written from an earlier instruction is the cycle when the earlier instruction in the WB stage. Therefore, we need to insert NOPs as shown below to ensure correct execution. The dependencies (or data hazards) are highlighted.

add **x15**, x12, x11	IF	ID	EX	MEM	WB									
NOP		*	*	*	*	*								
NOP			*	*	*	*	*							
lw **x13**, 8(**x15**)				IF	ID	EX	MEM	WB						
lw x12, 0(x2)					IF	ID	EX	MEM	WB					
NOP						*	*	*	*	*				
or **x13**, x15, **x13**							IF	ID	EX	MEM	WB			
NOP								*	*	*	*	*		
NOP									*	*	*	*	*	
sw **x13**, 0(x15)										IF	ID	EX	MEM	WB

b. Now, change and/or rearrange the code to minimize the number of `NOPs` needed. You can assume register x17 can be used to hold temporary values in your modified code.

ANSWER: In this sequence of instructions, the second instruction depends on the first instruction, the fourth instruction depends on the second instruction, and the fifth instruction depends on the fourth instruction. Only the third instruction can be reordered but reordering the third instruction cannot reduce the number of NOPs to be less than 5 in this case.

c. Now assume we schedule the above reordered instructions to be executed on a 2-issue statically scheduled processor, as shown below, draw the pipeline execution diagram again.

Instruction type	Pipeline Stages						
ALU/branch	IF	ID	EX	MEM	WB		
Load/store	IF	ID	EX	MEM	WB		
ALU/branch		IF	ID	EX	MEM	WB	
Load/store		IF	ID	EX	MEM	WB	
ALU/branch			IF	ID	EX	MEM	WB
Load/store			IF	ID	EX	MEM	WB

ANSWER: please find below.

add **x15**, x12, x11	IF	ID	EX	MEM	WB									
lw x12, 0(x2)	IF	ID	EX	MEM	WB									
NOP		*	*	*	*	*								
NOP		*	*	*	*	*								
NOP			*	*	*	*	*							
NOP			*	*	*	*	*							

NOP				*	*	*	*	*						
lw **x13**, 8(**x15**)				IF	ID	EX	MEM	WB						
NOP					*	*	*	*	*					
NOP					*	*	*	*	*					
NOP						*	*	*	*	*				
NOP						*	*	*	*	*				
or **x13**, x15, **x13**							IF	ID	EX	MEM	WB			
NOP							*	*	*	*	*			
NOP								*	*	*	*	*		
NOP								*	*	*	*	*		
NOP									*	*	*	*	*	
NOP									*	*	*	*	*	
NOP										*	*	*	*	*
sw **x13**, 0(x15)										IF	ID	EX	MEM	WB

Q7. (5 points) Please briefly discuss what the control hazard is. Please show a sample solution to overcoming the control hazard.

ANSWER: Control hazard refers to a situation that the control action (which instructions to run next) depends on the previous instruction, like in the case of branch instructions.

Control hazard can be addressed with branch prediction technique.

Q8. (10 points) An (m,n) correlating branch predictor uses the behavior of the most recent m executed branches to choose from 2^m predictors, each of which is an n-bit predictor. A two-level local predictor works in a similar fashion, but only keeps track of the past behavior of each individual branch to predict future behavior. There is a design trade-off involved with such predictors: correlating predictors require little memory for history, which allows them to maintain 2-bit predictors for a large number of individual branches (reducing the probability of branch instructions reusing the same predictor), while local predictors require substantially more memory to keep history and are thus limited to tracking a relatively small number of branch instructions. For this exercise, consider a (1,2) correlating predictor that can track four branches (requiring 16 bits) versus a (1,2) local predictor that can track two branches using the same amount of memory. For the following branch outcomes, provide each prediction, the table entry used to make the prediction, any updates to the table as a result of the prediction, and the final misprediction rate of each predictor. Assume that all branches up to this point have been taken. Initialize each predictor to the following:

Correlating predictor			
Entry	Branch	Last outcome	Prediction
0	0	T	T with one misprediction
1	0	NT	NT
2	1	T	NT
3	1	NT	T
4	2	T	T
5	2	NT	T
6	3	T	NT with one misprediction
7	3	NT	NT

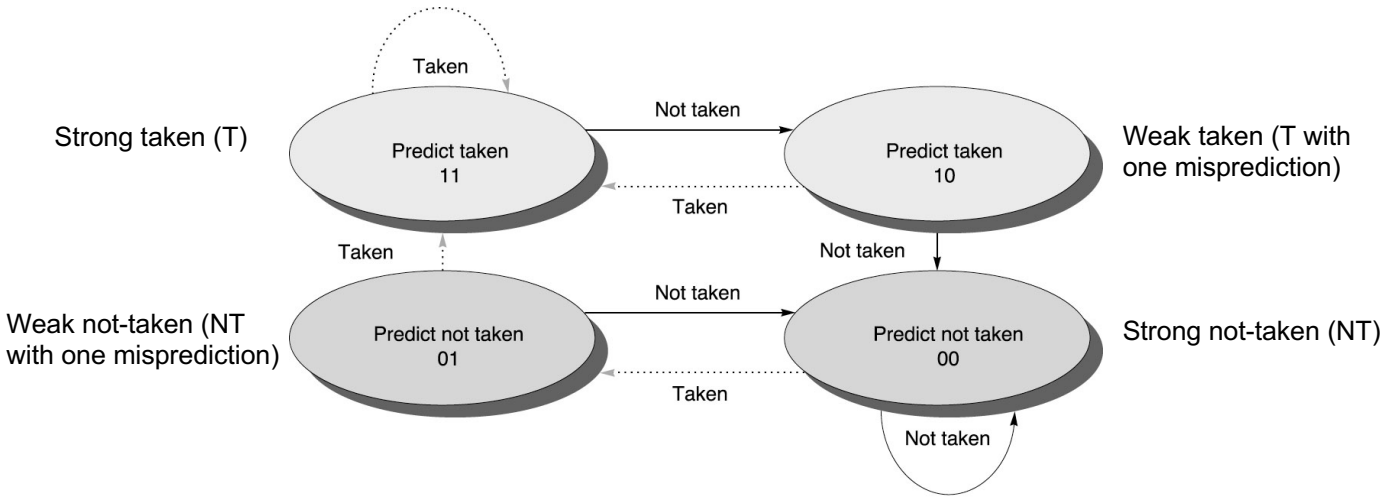
Local predictor			
Entry	Branch	Last 2 outcomes (right is most recent)	Prediction
0	0	T,T	T with one misprediction
1	0	T,NT	NT
2	0	NT,T	NT
3	0	NT	T
4	1	T,T	T
5	1	T,NT	T with one misprediction
6	1	NT,T	NT
7	1	NT,NT	NT

Branch PC (word address)	Outcome
454	T
543	NT
777	NT
543	NT
777	NT
454	T
777	NT
454	T
543	T

ANSWER:

For the correlating predictor that can track 4 branches, the branch address (i.e., the branch PC) will modulo by 4 to calculate the branch index. The branch index and the last (most recent one) branch outcome are used together as an entry to access the predictor table, to take the prediction from the table.

With a 2-bit predictor, the prediction result will be flipped after 2 miss predictions. In other words, "T" (or strong taken) will be changed to "T with one misprediction" (or weak taken) on a mis-prediction, then further changed to "NT" (strong not-taken) on another mis-prediction. On the other hand, "NT" (or strong not-taken) will be changed to "NT with one misprediction" (or weak not-taken) on a mis-prediction, then further changed to "T" (strong taken) on another mis-prediction. The state transition diagram is shown below.



Given this question, following the predictor table and 2-bit predictor state transition diagram, the correlating predictor will result in the below results.

Branch PC mod 4		Entry	Prediction	Outcome	Mispredict?	Table update
2	T	4	T	T	No	None
3	T	6	NT	NT	No	Change to "NT"
1	T	2	NT	NT	No	None
3	NT	7	NT	NT	No	None
1	NT	3	T	NT	Yes	Change to "T with one misprediction"
2	T	4	T	T	No	None
1	NT	3	T	NT	Yes	Change to "NT"
2	T	4	T	T	No	None
3	NT	7	NT	T	Yes	Change to "NT with one misprediction"

As such, the misprediction rate is 3 out of 9 branch instructions, i.e., 3 / 9 = 33.3%.

Similarly, following the local predictor table and 2-bit predictor state transition diagram, the local predictor will result in the below results.

Branch PC mod 2	Entry	Prediction	Outcome	Mispredict?	Table update
0	0	T	T	no	Change to "T"
1	4	T	NT	yes	Change to "T with one misprediction"
1	1	NT	NT	no	None
1	3	T	NT	yes	Change to "T with one misprediction"
1	3	T	NT	yes	Change to "NT"
0	0	T	T	no	None
1	3	NT	NT	no	None
0	0	T	T	no	None
1	5	T	T	no	Change to "T"

As such, the misprediction rate is also 3 out of 9 branch instructions, i.e., 3 / 9 = 33.3%.

Q9. (10 points) Please discuss what a superscalar processor is. Please discuss what are the advantages and disadvantages of a superscalar processor compared to a static multiple issue processor.

ANSWER: A superscalar processor is a multiple-issue processor with dynamic pipeline scheduling, which means the CPU hardware schedules instructions to avoid hazards, usually in an out-of-order execution fashion but commit results in order. This is in contrast to a static multiple issue processor, which relies on the compiler to schedule instructions.

The advantages of a superscalar processor include: first, the compiler doesn't need to have the knowledge of microarchitecture, as scheduling instructions is handled by the CPU, not by the compiler; second, it can handle cases where dependencies are unknown at compile time.

The disadvantages include: first, a superscalar processor substantially increases the hardware complexity as CPU needs to perform out-of-order execution and commit results in order; second, the complexity in hardware can lead to higher power consumption and the power wall issue.

Q10. (20 points)

a. Please use the sequence of 4 instructions we discussed in class, i.e., below instructions, to illustrate how Tomasulo's dynamic scheduling algorithm issues these 4 instructions, i.e., how Tomasulo's algorithm works in the first issue step. Assume we have the exactly same hardware as discussed in the class and shown in Figure 3.10 in the textbook.

```
I1: fadd.d f2, f4, f1
I2: fmul.d f1, f2, f3
I3: fsub.d f4, f1, f2
I4: fadd.d f1, f2, f3
```

b. Based on Tomasulo's dynamic scheduling algorithm, what are the possible orders of the execution of the above 4 instructions? Please list all possible orders. Please give the final results of f1, f2, f3, and f4 registers for each possible order and explain how Tomasulo's algorithm results in dispatching this order and leads to the results of each register.

ANSWER:

a. Tomasulo's algorithm is a dynamic scheduling algorithm that allows for out-of-order execution of instructions. This means that instructions can be executed out of order, as long as the dependencies between instructions are satisfied.

Tomasulo's algorithm uses reservation stations to hold instructions that are waiting to be executed. When an instruction is dispatched to a reservation station, it is placed in the station until all of its operands are available. Once all of the operands are available, the instruction can be executed.

Since Tomasulo's algorithm allows for dispatch and execution of instructions in any order, as long as the dependencies between instructions are satisfied, it can lead to more efficient execution of instructions (as soon as their operands are available).

Given the sequence of 4 instructions in this question, these 4 instructions will be queued in the instruction queue, as shown below.

I4:	`fadd.d f1, f2, f3`
I3:	`fsub.d f4, f1, f2`
I2:	`fmul.d f1, f2, f3`
I1:	`fadd.d f2, f4, f1`

During the issue step, first, I1 is issued (an add). Since all three adder RSs (reservation stations) are available, Tomasulo's algorithm will take one RS, assuming to be RS1, for this instruction. Meantime, Tomasulo's algorithm looks at the Register Alias Table, which has empty entries for both operands f4 and f1, indicating the values of f4 and f1 are available for the registers. As such, Tomasulo's algorithm takes the values of f4 and f1 from the registers, and place into RS1, as shown below.

RS1:	ADD	4.0	1.0
RS2:			
RS3:			

Since the result of this add will go to f2, the Register Alias Table will be updated to change the entry of f2 to be RS1, indicating now the value of f2 should come from RS1, as shown below.

f1:	
f2:	RS1
f3:	
f4:	

Second, I2 is issued (a multiplication). Since all two multiplier RSs are available, Tomasulo's algorithm will take one RS, assuming to be RS4, for this instruction. Meantime, Tomasulo's algorithm again looks at the Register Alias Table for two operands of I2, f2 and f3, which has RS1 entry and an empty entry, respectively, indicating the value of f2 comes from RS1, and the value of f3 comes from the register. As such, Tomasulo's algorithm places RS1 for operand f2 and takes the values of f3 from the register (i.e., 3.0), and place these operands into RS4, as shown below.

RS4:	MUL	RS1	3.0
RS5:			

Since the result of this multiplication instruction will go to f1, the Register Alias Table will be updated to change the entry of f1 to be RS4, indicating now the value of f1 should come from RS4, as shown below.

f1:	RS4
f2:	RS1
f3:	
f4:	

Third, I3 is issued (a sub). Since there're two adder RSs (RS2 and RS3) are available, Tomasulo's algorithm will take one RS, assuming to be RS2, for this instruction. Meantime, Tomasulo's algorithm again looks at the Register Alias Table for two operands of I3, f1 and f2, which has RS4 and RS1, respectively, in their entries, indicating the value of f1 comes from RS4, and the value of f2 comes from RS1. As such, Tomasulo's algorithm places these operands and the operation (SUB) into the issue reservation station, RS2, as shown below.

RS1:	ADD	4.0	1.0
RS2:	SUB	RS4	RS1
RS3:			

Since the result of this sub will go to f4, the Register Alias Table will be updated to change the entry of f4 to be RS2, indicating now the value of f4 should come from RS2, as shown below.

f1:	RS4
f2:	RS1
f3:	
f4:	RS2

Fourth and last, I4 is issued (an add). Since there's still an adder RS (RS3) is available, Tomasulo's algorithm will take that RS (i.e., RS3) for this instruction. Meantime, Tomasulo's algorithm again looks at the Register Alias Table for two operands of I4, f2 and f3, which has RS1 entry and an empty entry, respectively, indicating the value of f2 comes from RS1, and the value of f3 comes from the register. As such, Tomasulo's algorithm places these operands and the operation (ADD) into the issue reservation station, RS3, as shown below.

RS1:	ADD	4.0	1.0
RS2:	SUB	RS4	RS1
RS3:	ADD	RS1	3.0

Since the result of this add will go to f1, the Register Alias Table will be updated to change the entry of f1 to be RS3, instead of RS4, indicating now the value of f1 should come from RS3, as shown below.

f1:	RS3
f2:	RS1
f3:	
f4:	RS2

b. Continuing from a, after issuing all 4 instructions, the status of 5 RSs is shown below.

RS1:	ADD	4.0	1.0
RS2:	SUB	RS4	RS1
RS3:	ADD	RS1	3.0
RS4:	MUL	RS1	3.0
RS5:			

At this moment, only one instruction, I1, has both operands available. As such, the first instruction that can be executed or dispatched must be I1.

After I1 is executed, the result of 5.0, i.e., RS1 value, or f2 (as seen from the Register Alias Table), will be broadcasted to the registers and all RS. RS1 will be cleared as I1 finishes the execution. As such, all 5 RSs are updated and shown below:

RS1:			
RS2:	SUB	RS4	5.0
RS3:	ADD	5.0	3.0
RS4:	MUL	5.0	3.0
RS5:			

The registers' values are updated as shown below:

f1:	1.0
f2:	5.0
f3:	3.0
f4:	4.0

At this moment, there're two possibilities, either RS3 or RS4 is dispatched, as both instructions have two operands ready. For the first possibility, RS3 is dispatched, i.e., I4 is executed. After that, RSs are updated and shown below.

RS1:			
RS2:	SUB	RS4	5.0
RS3:			
RS4:	MUL	5.0	3.0

RS5:			

At this moment, only RS4 can be dispatched, as only RS4 has both operands ready. In other words, at this moment, we can only dispatch RS4 (or I2) first, then RS2 (or I3) next.
The final registers' values are as shown below:

f1:	8.0
f2:	5.0
f3:	3.0
f4:	10.0

Note that the final value of f1 comes from RS3, not RS4 anymore, according to the Register Alias Table.

For the second possibility, RS4 is dispatched, i.e., I2 is executed. After that, RSs are updated and shown below.

RS1:			
RS2:	SUB	15.0	5.0
RS3:	ADD	5.0	3.0
RS4:			
RS5:			

At this moment, since both RS2 and RS3 have two operands ready, they can be executed in any order, i.e., either RS2 (i.e., I3) then RS3 (i.e., I4), or RS3 (i.e., I4) then RS2 (i.e., I3).

For either order, the final registers' values are as shown below:

f1:	8.0
f2:	5.0
f3:	3.0
f4:	10.0

Therefore, there are totally 3 possible orders:
I1, I4, I2, I3
Or,
I1, I2, I3, I4
Or,
I1, I2, I4, I3

Apparently, the final results of f1, f2, f3, and f4 registers for each possible order are the same, and as shown below.

f1:	8.0
f2:	5.0
f3:	3.0
f4:	10.0

Again, note that the final value of f1 comes from RS3, not RS4, according to the Register Alias Table.

THE END.