Surname (COGNOME)	SOLUTION
Name	
POLIMI Personal Code	
Signature	

Politecnico di Milano, 13 June, 2023

Course on Advanced Computer Architectures

Prof. C. Silvano

EX1	(5 points)	
EX2	(5 points)	
EX3	(5 points)	
Q1	(5 points)	
Q2	(5 points)	
QUIZZES	(8 points)	
TOTAL	(33 points)	

EXERCISE 1 – TOMASULO (5 points)

Let's consider the following assembly code. to be executed on a CPU with dynamic scheduling based on **TOMASULO algorithm** with all cache HITS, a single Common Data Bus and:

- *2 RESERVATION STATIONS **(RS1, RS2)** with 1 LOAD/STORE unit **(LDU1)** with latency **4***
- *2 RESERVATION STATION **(RS3, RS4)** with 1 ALU/BR unit **(ALU1)** with latency **2***

Please complete the following table:

INSTRUCTION	ISSUE	START EXEC	WRITE RESULT	Hazards Type	RSi	UNIT
`I1: lw $f0,0($r1)`	1	2	6		RS1	LDU1
`I2: lw $f2,4($r1)`	2				RS2	LDU1
`I3: lw $f4,8($r1)`						
`I4: lw $f6,C($r1)`						
`I5:fadd $f8,$f0,$f2`						
`I6:fadd $f10,$f4,$f6`						
`I7:fadd $f10,$f8,$f10`						
`I8: sw $f10,4($r1)`						

Calculate the CPI:

CPI = ____

Feedback:

INSTRUCTION	ISSUE	START EXEC	WRITE RESULT	Hazards Type *(*)*	RSi	UNIT
`I1: lw $f0,0($r1)`	1	2	6		RS1	LDU1
`I2: lw $f2,4($r1)`	2	**7**	11	**STR. LDU1**	RS2	LDU1
`I3: lw $f4,8($r1)`	**7**	**12**	16	**STR. RS1 + STR. LDU1**	RS1	LDU1
`I4: lw $f6,C($r1)`	**12**	**17**	21	**STR. RS2 + STR.LDU1**	RS2	LDU1
`I5:fadd $f8,$f0,$f2`	13	14	**17**	**STR. CDB/RF WRITE**	RS3	ALU1
`I6:fadd $f10,$f4,$f6`	14	**22**	24	**RAW $f6**	RS4	ALU1
`I7:fadd $f10,$f8,$f10`	**18**	**25**	27	**STRUCT RS3 + RAW Sf10**	RS3	ALU1
`I8: sw $f10,4($r1)`	19	**28**	32	**RAW $f10**	RS1	LDU1

CPI = # clock cycles / IC = 32 / 8 = 4

(*) Only the hazards that have caused the introduction of some stalls are reported in the table. Other potential hazards are already solved.

In Tomasulo, the WAW and WAR hazards are already solved by Register Renaming

EXERCISE 2 – MESI PROTOCOL (5 points)

Let's consider the following access patterns on a **4-processor** system with a direct-mapped, **write-back cache** with one cache block per processor and a two-cache block memory.

Assume the **MESI protocol** is used, with **write-back** caches, **write-allocate,** and **write-invalidate** of other caches.

Please complete the following table:

Cycle	After Operation	P0 cache block state	P1 cache block state	P2 cache block state	P3 cache block state	Memory at bl. 0 up to date?	Memory at bl. 1 up to date?
0	**P0: write Bl. 1**	Mod (1)	Invalid	Invalid	Invalid	Yes	No
1	**P2: Read Bl. 0**						
2	**P3: Read Bl. 0**						
3	**P1: Write Bl. 0**						
4	**P2: Write Bl. 1**						
5	**P3: Read Bl. 0**						
6	**P0: Read Bl. 1**						
7	**P2: Write Bl. 0**						

Feedback:

Cycle	**After Operation**	**P0 cache block state**	**P1 cache block state**	**P2 cache block state**	**P3 cache block state**	**Memory at bl. 0 up to date?**	**Memory at bl. 1 up to date?**
0	**P0: write Bl. 1**	Mod (1)	Invalid	Invalid	Invalid	Yes	No
1	**P2: Read Bl. 0**	Mod (1)	Invalid	Excl (0)	Invalid	Yes	No
2	**P3: Read Bl. 0**	Mod (1)	Invalid	Shared (0)	Shared (0)	Yes	No
3	**P1: Write Bl. 0**	Mod (1)	Mod (0)	Invalid	Invalid	No	No
4	**P2: Write Bl. 1**	Invalid	Mod (0)	Mod (1)	Invalid	No	No
5	**P3: Read Bl. 0**	Invalid	Shared (0)	Mod (1)	Shared (0)	Yes	No
6	**P0: Read Bl. 1**	Shared (1)	Shared (0)	Shared (1)	Shared (0)	Yes	Yes
7	**P2: Write Bl. 0**	Excl (1)	Invalid	Mod (0)	Invalid	No	Yes

EXERCISE 3 – CACHE MEMORIES (5 points)

Let's consider 32-block main memory with a **4-way set associative** 8-block cache based on a ***write allocate*** with ***write-back*** protocol.

The addresses are expressed as:
Memory Address: [0, 1, 2, ………. 31] $_{10}$
Cache Address: [a, b, c, d, e, f, g, h]
and Cache Tags are expressed as binary numbers.

At cold start the cache is empty, then there is the following sequence of memory accesses.
Please complete the following table:

	Type of memory access	Memory Address	HIT/MISS Type	Cache Tag	Cache Address	Set	Write in memory
1	Read	[16] $_{10}$	Cold-start Miss				
2	Write	[16] $_{10}$	Hit				
3	Read	[14] $_{10}$					
4	Read	[07] $_{10}$					
5	Write	[14] $_{10}$					
6	Write	[16] $_{10}$					
7	Read	[12] $_{10}$					
8	Read	[25] $_{10}$					
9	Write	[10] $_{10}$					
10	Read	[02] $_{10}$					
11	Write	[06] $_{10}$					
12	Read	[30] $_{10}$					

Feedback

The 4-way set-associative cache has 8-blocks [a, b, c, d, e, f, g, h] organized in 2 sets, where each set contains 4 blocks as follows: **Set_0: [a , b, c, d]; Set_1: [e, f, g, h]**

Being a set-associative cache, the block replacement policy in each set uses the **LRU algorithm**.

Memory Address Mapping:

Set_0 [a , b, c, d]	Set_1 [e, f, g, h]
0	1
2	3
4	5
…	…
30	31

	Type of memory access	Memory Address	HIT/MISS Type	Cache Tag	Cache Address	Set	Write in memory
1	Read	$[16]_{10}$	Cold-start Miss	$[1000]_2$	a	$[0]_{10}$	No
2	Write	$[16]_{10}$	Hit	$[1000]_2$	a	$[0]_{10}$	No
3	Read	$[14]_{10}$	Cold-start Miss	$[0111]_2$	b	$[0]_{10}$	No
4	Read	$[07]_{10}$	Cold-start Miss	$[0011]_2$	e	$[1]_{10}$	No
5	Write	$[14]_{10}$	Hit	$[0111]_2$	b	$[0]_{10}$	No
6	Write	$[16]_{10}$	Hit	$[1000]_2$	a	$[0]_{10}$	No
7	Read	$[12]_{10}$	Cold-start Miss	$[0110]_2$	c	$[0]_{10}$	No
8	Read	$[25]_{10}$	Cold-start Miss	$[1100]_2$	f	$[1]_{10}$	No
9	Write	$[10]_{10}$	Cold-start Miss	$[0101]_2$	d	$[0]_{10}$	No
10	Read	$[02]_{10}$	Conflict Miss	$[0001]_2$	b	$[0]_{10}$	Yes Wr. in $M[14]_{10}$
11	Write	$[06]_{10}$	Conflict Miss	$[0011]_2$	a	$[0]_{10}$	Yes Wr. in $M[16]_{10}$
12	Read	$[30]_{10}$	Conflict Miss	$[1111]_2$	c	$[0]_{10}$	No

QUESTION 1: MULTITHREADING (5 points)

Let's consider the different multithreading techniques used to manage thread-level parallelism by hardware processors. *Answer to the following questions:*

	Coarse-grained Multithreading	**Fine-grained Multithreading**	**Simultaneous Multithreading**
Explain the main concepts for each technique.	Whenever a thread is stalled (for any reason, such as a dependence, a cache miss, etc.), the processor switches to execute another thread, that uses the processor resources. The processor must be able to switch threads by hardware context switch.	In fine-grained MT, the processor switches between threads on each instruction by hardware context switch. Execution of multiple threads is interleaved in a sort of round-robin fashion, skipping any thread that is stalled at that time.	Suitable for multiple-issue dynamic scheduled processors (such as 4-issues superscalar processors). Simultaneously schedule instructions execution from several threads in the same cycle in order to maximize the use of parallel functional units in multiple-issues processors. SMT exploits both ILP and TLP.
For each technique, make ***an example*** *with up to 4 threads (T1, T2, T3, T4) on a dual-issue processor*	C0 T1 T1 C1 T1 C2 T1 T1 C3 T1 T1 C4 T1 C5 T2 T2 C6 T2 T2 C7 T2 C8 T3 C9 T3 T3	C0 T1 T1 C1 T2 C2 T3 T3 C3 T4 T4 C4 T1 C5 T3 T3 C6 T4 T4 C7 T1 T1 C8 T2 T2 C9 T3	C0 T1 T2 C1 T3 T4 C2 T1 T2 C3 T3 T3 C4 T1 T2 C5 T1 C6 T1 T2 C7 T3 T4 C8 T1 T2 C9 T3 T4
Let's consider a dual-issue SMT processor that can manage up to 4 threads			*How much is the* ***ideal CPI?*** Ideal CPI = 0.5 *How much is the* ***ideal per-thread CPI?*** Ideal per-thread CPI = 2

What are the main benefits for each technique?	Go beyond the limits of ILP to get better performance (this benefit is also valid for fine-grained and SMT) It can hide the penalty of long stalls because, as a example, during an L2 cache miss, it can use the stall cycles to switch to execute another thread. Requires a simpler switching logic with lower overhead than fine-grained and SMT. Requires a less frequent HW context switch than fine-grained and SMT. Less impact on single thread execution time: an individual thread might complete execution earlier than in fine-grained and SMT.	It can hide both short and long stalls because instructions from other threads are executed when one thread stalls. The frequent switching among threads helps to reduce uniformly the penalty due to stalls because a control/data dependency can be solved during the cycles used by another thread. More fairness: All threads can begin/continue their execution quite uniformly, so there is no individual thread “left behind”. Fine-grain (but also coarse-grain) can be applied even to a simple single issue processor core that can be combined in a multicore.	Suitable to maximize the use of parallel functional units available in the multiple-issues by different threads. Exploit more parallelism than coarse- and fine-grained MT.
What are the main drawbacks for each technique?	The new thread has a pipeline startup overhead period to empty the pipeline before starting the new thread. This causes a loss of throughput. Less fairness than fine-grain MT: Some threads might not be able to begin/continue execution for some time. If applied to multiple-issue processors, ILP limitations could lead to empty issue slots at each clock cycle (same for fine-grained MT)	Compared to coarse-grain MT, it slows down the execution of individual threads, since a thread will be delayed by another thread even if it is ready to execute. Higher overhead due to more frequent HW context switch.	In large multiple-issue processors (such as 4-issue) requires complex dynamic control logic to keep busy the multiple-slots. Higher overhead and power consumption. Obviously, it is limited by the number of issues available in the processor.

QUESTION 2: BRANCH PREDICTION (5 points)

Let's consider the following code where $R0 has been initialized to 0:

```
INIT:   ADDI $R1, $R0, 0
        ADDI $R2, $R0, 80
        ADDI $R3, $R0, 0
        ADDI $R4, $R0, 40

LOOP1: LD $F0, 0($R1)
        FADD $F2, $F0, $F0
        SD $F2, 0($R1)

        ADDI $R3, $R0, 0
LOOP2: LD $F4, 0($R3)
        FADD $F6, $F4, $F4
        SD $F6, 0($R3)
        ADDI $R3, $R3, 4
BR2:    BNE $R3, $R4, LOOP2

        ADDI $R1, $R1, 4
BR1:    BNE $R1, $R2, LOOP1
```

Let's assume to have a **1 entry 1-bit BHT** as dynamic branch predictor (where the two branch instructions **collide**) initialized as ***Taken.*** *Answer to the following questions:*

	LOOP 1	**LOOP2**	**GLOBALLY**
How many iterations are executed for each loop?	The outer loop LOOP1 is executed **20 times**.	The inner loop LOOP2 is executed **10 times for each iteration of LOOP1.** Globally LOOP2 is executed (10 x 20) = **200 times.**	▨
How many times the BHT has been used?	The BHT has been used **20 times for BR1** (once per iteration).	The BHT has been used **200 times for BR2** (once per iteration).	Globally, the BHT has been used 220 times (20 + 200). ***=>220 predictions done***

*Assuming the BHT initialized as **Taken,** when there is the **first misprediction?***		At the first iteration of LOOP1, being the predictor initialized as Taken, we have the first misprediction at the last iteration (exit) of the BR2.	
At the first iteration of the outer LOOP1, is there a correct prediction or a misprection?	Exiting from the inner LOOP2 with the NT prediction, this generates a **misprediction** at the first iteration of LOOP1.		
When re-entering in each loop, the value of the prediction bit is Taken or Not Taken?	When re-entering in LOOP1, the predictor is **Taken**.	When re-entering in LOOP2, the predictor is **Taken**.	
How many mispredictions are we going to observe locally in the inner LOOP2?		Considering LOOP2 in isolation, we have **1 misprediction out of 10 predictions** (local misprediction rate 10% for the inner LOOP2).	
How many mispredictions are we going to observe?	Exiting from the inner LOOP2 as Not Taken, this generates a misprediction on the BR1 for 19 iterations (except for the last iteration). ⇨ for the outer LOOP1, we have **19 mispredictions.**	We have 1 misprediction for the BR2 only when exiting from LOOP2, this is for the 20 iterations of the outer LOOP1 => for the inner LOOP2, we have **20 mispredictions.**	Globally there are (19 + 20) = **39 mispredictions.**
How much is the global misprediction rate?			There are 39 mispredictions out of 220 predictions => **39/220 % = 17.72%** global misprediction rate.

MULTIPLE-CHOICE QUESTIONS: (8 points)

Question 3 (format True – False)
A multiple-issue processor has more functional units in parallel than a single thread can use.
(SINGLE ANSWER)
1 point

Answer 1: TRUE **(TRUE)**
Answer 2: FALSE

Question 4 (format Multiple Choice – Single answer)
Let's consider a directory-based protocol for a distributed shared memory system with 4 Nodes (N0, N1, N2, N3) where we consider the block B0 in the directory of N0:
Directory N0 Block B0 | State: Shared | Sharer Bits: 1001 |

*During a **Write Miss** on B0 from N1, please indicate which are the home node, the local node and the remote node(s):*
(SINGLE ANSWER)
1 point

Answer 1: N0 home node, N1 local node; N3 remote node.
Answer 2: N0 home node; N3 local node; N1 remote node.
Answer 3: N0 home node; N1 local node; N0 and N3 remote nodes. **(TRUE)**
Answer 4: N1 home node; N0 local node; N3 remote node.
Answer 5: N1 home node; N1 local node; N3 remote node.

Question 5 (format Multiple Choice – Multiple answer)

Let's consider the following loop:

```
for (i=1; i<=100, i++) {
     X[i] = X[i] + X[i-1];    /*S1*/
     Z[i] = X[i] + Y[i-1]     /*S2*/
     Q[i] = X[i] + Q[i-1]     /*S3*/
}
```

How many loop-carried dependencies are in the code?
(MULTIPLE ANSWERS)
1 point

Answer 1: One in S1 because X[i] depends on X[i-1]; (**TRUE**)
Answer 2: One in S2 because Z[i] depends on Y[i-1];
Answer 3: One in S2 because Z[i] depends on X[i];
Answer 4: One in S3 because Q[i] depends on Q[i-1]; (**TRUE**)

Feedback:
The dependence of Z[i] on Y[i-1] in S2 is not a loop-carried dependence because the vector Y[ ] is never modified in the loop. Also the dependences of Z[i] on X[i] in S2 and Q[i] on X[i] in S3 are not loop-carried dependences.

Question 6 (format Multiple Choice – Multiple answers)

Let's consider the following code sequence executed by a dynamic scheduled processor:

```
I1:  LD.D $FP1, 4($R2)
I2:  ADDI.D $FP0, $FP1, 4
I3:  SD.D $FP0, 0($R1)
I4:  LD.D $FP0, 4($R1)
I5:  ADDI $R1, $R1, 4
```

*Where are the **WAR hazards** in the code?*

(MULTIPLE ANSWERS)
1 point

Answer 1: 1 WAR on $FP1 between I2 and I1;
Answer 2: 1 WAR on $FP0 between I3 and I2;
Answer 3: 1 WAR on $FP0 between I4 and I3; (**TRUE**)
Answer 4: 1 WAR on $R1 between I5 and I3; (**TRUE**)
Answer 5: 1 WAR on $R1 between I5 and I4; (**TRUE**)

Question 7 (format Multiple Choice – Single answer)

Let's consider the following code executed by a Vector Processor with:

- *Vector Register File composed of 32 vectors of 8 elements per 64 bits/element;*
- *Scalar FP Register File composed of 32 registers of 64 bits;*
- *One Load/Store Vector Unit with operation chaining and memory bandwidth 64 bits;*
- *One ADD/SUB Vector Unit with operation chaining.*
- *One MUL/DIV Vector Unit with operation chaining.*

```
L.V V1, RX             # Load vector from memory address RX into V1
MULVS.D V1, V1, F0     # FP multiply vector V1 to scalar F0
ADDVV.D V2, V1, V1     # FP add vectors V1 and V1
MULVS.D V2, V2, F0     # FP multiply vector V2 to scalar F0
L.V V3, RY             # Load vector from memory address RY into V2
ADDVV.D V3, V2, V3     # FP add vectors V2 and V3
S.V V3, RZ             # Store vector V3 into memory address RZ
```

How many convoys? How many clock cycles to execute the code?

(SINGLE ANSWER)
2 points

Answer 1: 3 convoys; 24 clock cycles **(TRUE)**
Answer 2: 2 convoys; 16 clock cycles
Answer 3: 4 convoys; 32 clock cycles
Answer 4: 5 convoys; 40 clock cycles

Feedback: There are 3 convoys:

1) L.V V1, RX;	MULVS.D V1, V1, F0	ADDVV.D V2, V1, V1
2) L.V V3, RY	MULVS.D V2, V2, F0	ADDVV.D V3, V2, V3
3) S.V V3, RZ		

Question 8 (format Multiple Choice – Single answer)

Let's consider a ***Speculative Tomasulo*** *architecture with 2 load buffers (Load1, Load2), 2 multiply reservation stations (Mult1 and Mult2) and* ***6-entry ROB*** *(ROB0, ROB1, …, ROB5).*
Let's consider the following LOOP when the ROB is ***full*** *after the issue of the instructions of the first iteration (while the first load is executing a cache miss) and the start of the speculative issue of the second iteration.*

```
LOOP: LD $F0, 0 ($R1)
      MULTD $F4, $F0, $F2
      SD $F4, 0 ($R1)
      SUBI $R1, $R1, 8
      BNEZ $R1, LOOP   // branch prediction taken
```

In the Rename Table, what are the pointers used for $F0 and $F4?

(SINGLE ANSWER)
2 points

Answer 1: ROB5 for $F0 and ROB1 for $F4 **(TRUE)**
Answer 2: ROB5 for $F0 and ROB2 for $F4
Answer 3: ROB0 for $F0 and ROB1 for $F4
Answer 4: ROB0 for $F0 and ROB2 for $F4

Feedback:
The 6-entry ROB is FULL:

ROB#	**Instruction**	**Dest.**	**Ready**	
ROB0	LD $F0, 0 ($R1) (1^ iteration exec. cache miss)	$F0	No	HEAD
ROB1	MULTD $F4, $F0, $F2 (1^ iteration issued)	$F4	No	
ROB2	SD $F4, 0 ($R1) (1^ iteration issued)	MEM	No	
ROB3	SUBI $R1, $R1, 8 (1^ iteration issued)	$R1	No	
ROB4	BNEZ $R1, LOOP (1^ branch predicted as taken)		No	
ROB5	LD $F0, 0 ($R1) (2^ iteration issued speculatively)	$F0	No	

Rename Table:

$F0	ROB5
$F2	
$F4	ROB1

Therefore, in the Rename Table, $F0 points to ROB5 because the WAW $F0 is solved while $F4 points to ROB1 because the second MULTD has not yet been issued because the ROB is full. See also L07: Reorder Buffer & Speculation