INDIAN INSTITUTE OF TECHNOLOGY KHARAGPUR
Final Test Solution - 2022

Date: 13-04-22 **Timing: 8 am - 9:30 am; Total Marks =** 60
Subject No : CS60003 **HIGH PERFORMANCE COMPUTER ARCHITECTURE**
Department/Centre/School : Computer Science and Engineering
Answer all questions. In case of reasonable doubt, make practical assumptions and state them upfront. Marks will be deducted for claims without proper reasoning. Write your answers (with all analysis, justification, calculation etc) in loose sheets, scan them and upload in moodle within the allotted time.

1. Consider the instruction sequence that executes according to Tomasulo's algorithm with Re-order Buffer (RoB) and Load-Store Queue (LSQ).

Instruction	**Operands**
L.D	F1, 0(R2)
ADD.D	F3, F1, F5
L.D	F2, 100(R2)
MUL.D	F1, F2, F3
S.D	F5, 0(R2)
ADDI	R4, R2, #8
L.D	F1, 0(R2)
ADD.D	F3, F1, F5

Problem Description: Please consider the following assumptions during computation.

- Instruction Issue and Commits are in-order. However, execution can be out-of-order.
- Only a single instruction is issued per cycle.
- In case of RAW hazards, the operand's value are ready after the value is written in ROB from Common Data Bus (CDB).
- The size of Reorder Buffer is infinite. This will eliminate any stalls due to the ROB being full.
- At most one instruction can be committed in each cycle.
- There is no cache miss.
- Assume CDB is wide enough for handling multiple instructions. In case of race condition in the CDB, multiple concurrent result updates can be broadcast in a single cycle.
- The initial values of the registers are: $R2 = 100$, $R3 = 20$, $F1 = 50$, $F5 = 25$, $F3 = 60$. Assume that the initial values in the memory addresses 100 and 200 are 0 and 4 respectively.
- We have 4 functional units as follows:
  - **Integer ALU:** All integer operations are performed here. Needs 1 cycle for execution. It has 2 reservation stations. For the conditional branch, the instructions are resolved in 1 cycle only.
  - **FP Adder:** Needs 4 cycles in execution stage. Has 3 reservation stations.
  - **FP Multiplier:** Needs 8 cycles in execution stage. Has 2 reservation stations.
  - **Load Store Unit:** For calculating effective addresses, the load and store instructions need 1 cycle in execution stage. They also requires 2 cycles for cache access. We are considering only 1 load and store unit in this problem. It has a buffer size of 5. Also, **the store-to-load forwarding is enabled and takes only 1 cycle.**

Question: Trace the execution by showing the **Reservation Station Status**, **LSQ status**, **RAT**, **ROB** and **Register Status** at the end of cycles 8, 10 and 16.

Provide the Final table of **Instruction Status** identifying the cycles when each of the different stages of the instruction **start**, e.g. the cycle count for start of execution of S.D should be the entry for 5th row in "Execute" column. (Use the table template provided). **[4 × 5 = 20]**

Solution The final table is given below:

Instructions	**Instruction Status**			
	Issue	**Execute**	**CDB ⇒ ROB**	**Commit**
L.D F1, 0(R2)	**1**	**2**	**5**	**6**
ADD.D F3, F1, F5	**2**	**6**	**10**	**11**
L.D F2, 100(R2)	**3**	**4**	**7**	**12**
MUL.D F1, F2, F3	**4**	**11**	**19**	**20**
S.D F5, 0(R2)	**5**	**6**	**7**	**21**
ADDI R4, R2, #8	**6**	**7**	**8**	**22**
L.D F1, 0(R2)	**7**	**8**	**11**	**23**
ADD.D F3, F1, F5	**8**	**12**	**16**	**24**

2. Consider the following program.

```
loop:
        LW   R2, 0(R1);
        ADD  R2, R2, R3;
        SW   R2, 0(R1);
        ADDI R1, R1, -4;
        BNE  R1, R5, loop;
```

Consider a 4-issue, in-order CPU with perfect Branch Predictor which executes every instruction in single cycle. An optimizing compiler can do the following for the code : intelligent instruction scheduling, unroll by a factor of 1, replace two consecutive LW instructions loading from consecutive locations X(R1), X-4(R1) in memory to registers R2 and R3 with a single optimized load instruction LW-opt given as "LW-opt R2, R3, X(R1)" which also takes a single cycle. Write the optimized output code for this compiler and calculate CPI for the same showing a cycle based analysis. **[5]**

Solution:

```
loop:
        LW-opt   R2, R10, 0(R1);
        ADD  R2, R2, R3;
        ADD  R10, R10, R3;
        ADDI R1, R1, -8;
        SW   R2, 8(R1);
        SW   R10, 4(R1);
        BNE  R1, R5, loop;
```

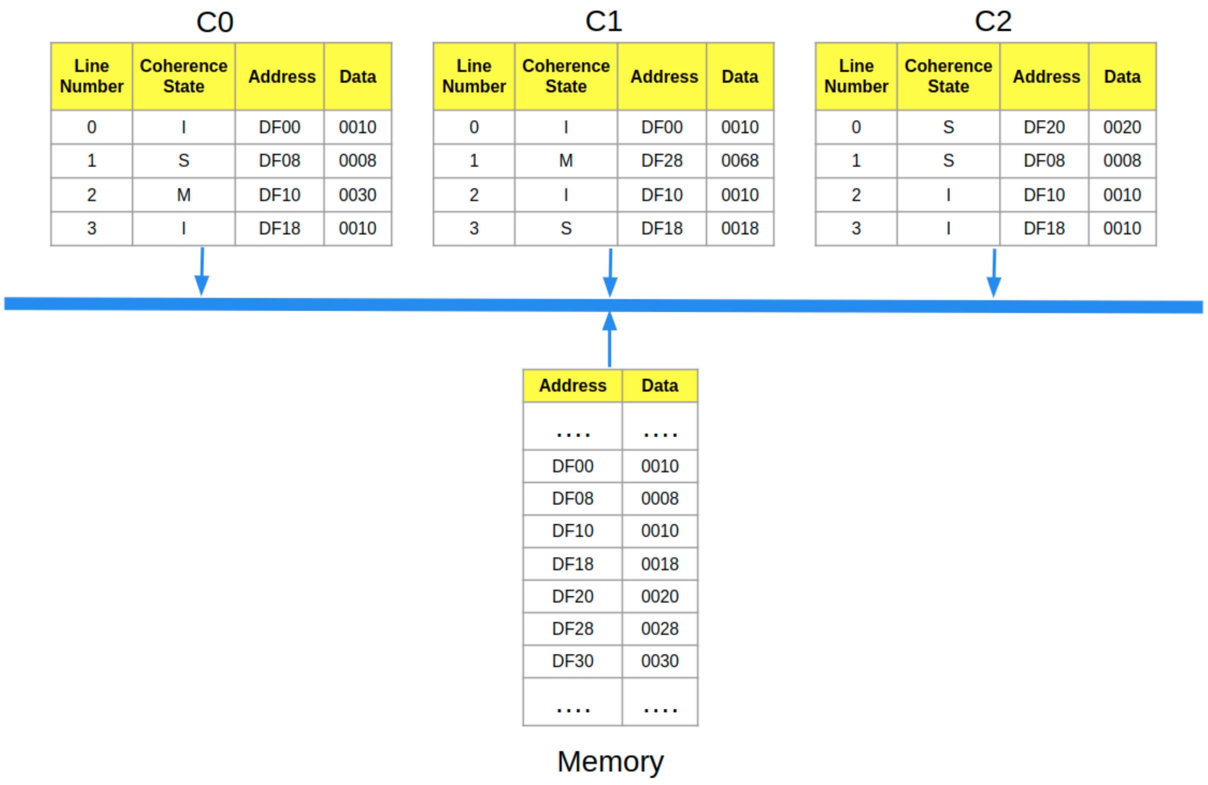
CPI: $\frac{2}{7} = 0.2857$

3. Consider the following multicore processor scenario:

- Number of cores is three, each core is equipped with a single private cache, caches are direct mapped
- Each cache has four lines and each line has two bytes
- Caches employ invalidation based snooping coherence protocol
- CPU read and write local cache hits generate no stall cycles.

- CPU read and write misses in cache generate ***175*** stall cycles if they need to be satisfied by memory. CPU read and write misses generate ***55*** stall cycles if they are satisfied by other caches.
- CPU write hits that generate an invalidate incur ***35*** stall cycles.
- A write-back of a block, due to either a conflict or another processor's request to a M block, incurs an additional ***17*** stall cycles.

As depicted in Figure 1, to simplify the illustration, the cache-address tags contain the full address.



C0

Line Number	Coherence State	Address	Data
0	I	DF00	0010
1	S	DF08	0008
2	M	DF10	0030
3	I	DF18	0010

C1

Line Number	Coherence State	Address	Data
0	I	DF00	0010
1	M	DF28	0068
2	I	DF10	0010
3	S	DF18	0018

C2

Line Number	Coherence State	Address	Data
0	S	DF20	0020
1	S	DF08	0008
2	I	DF10	0010
3	I	DF18	0010

Address	Data
....	....
DF00	0010
DF08	0008
DF10	0010
DF18	0018
DF20	0020
DF28	0028
DF30	0030
....	....

Memory

Figure 1: Multi-core system depicting caches and memory

For the multi-core system illustrated in Figure 1, consider the execution of a sequence of operations. We use the following symbols for this purpose.
C#core: R, <address> implies a read by #core from <address> and
C#core: W, <address> $\leftarrow$ ***<value written>*** implies a write by #core to <address> of value =<value written>.
Read and write operations are for 1 byte at a time. For each of the read/write sequences given in the following cases, assume the initial state of the system same as the one in Figure 1. In each case, identify total stall cycles and sequence of state transitions with both MSI protocol and MESI protocol. Assume state transitions that require zero interconnect transactions incur no additional stall cycles.

An example is given below.

C0: R, DF00 : Read miss, satisfied in memory, MSI: S, MESI: E for DF00 in C0
C1: W, DF00 $\leftarrow$ ***60*** : Write miss, satisfied in memory regardless of protocol. State : M
Both MSI and MESI will incur 175 + 175 = 350 stall cycles

Similarly, do for the following cases.

(1) ***C0: R, DF00***
C0: W, DF00 $\leftarrow$ ***40***

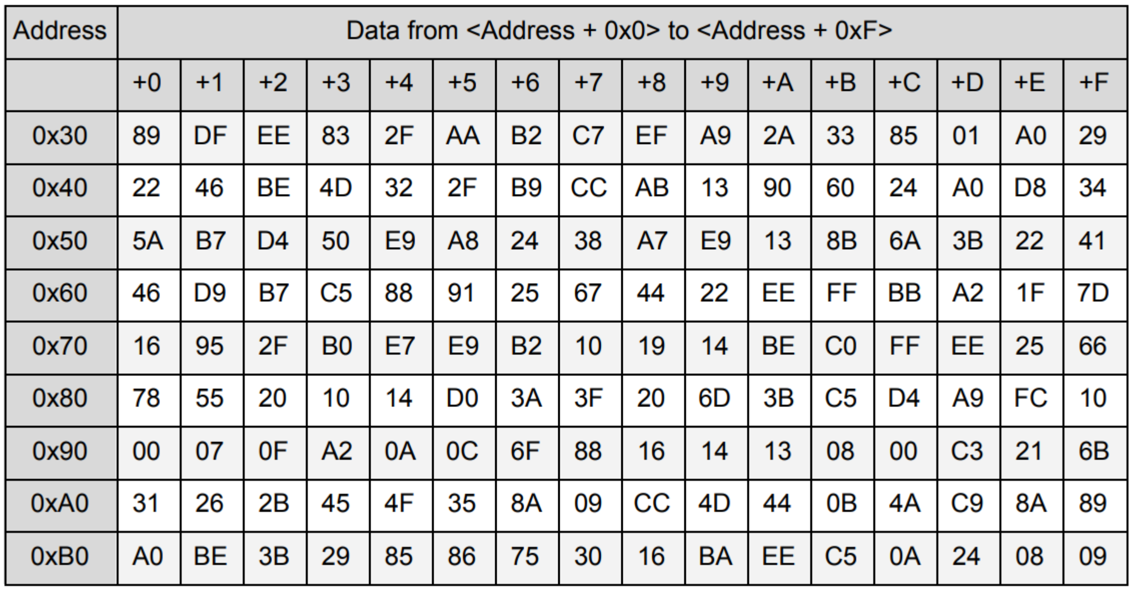
(2) ***C0: R, DF00***
C0: W, DF00 $\leftarrow$ ***60***
C1: W, DF00 $\leftarrow$ ***40***

For both protocols, write (very brief) explanations of your stall calculations in all cases. **[2 $\times$ 5 = 10]**
Answer:

(1) ***C0: R, DF00;*** Read miss, satisfied in memory, no sharers MSI: S, MESI: E
C0: W, DF00 $\leftarrow$ ***40;*** MSI: send invalidate, MESI: silent transition from E to M
MSI: 175 + 35 = 210 stall cycles
MESI: 175 + 0 = 175 stall cycles

(2) ***C0: R, DF00;*** Read miss, satisfied in memory, no sharers MSI: S, MESI: E
C0: W, DF00 $\leftarrow$ ***60;*** MSI: send invalidate, MESI: silent transition from E to M
C1: W, DF00 $\leftarrow$ ***40;*** Write miss, C0's cache, writeback data to memory
MSI: 175 + 35 + 55 + 17 = 282 stall cycles
MESI: 175 + 0 + 55 + 17 = 247 stall cycles

4. A mission-critical application (PID=1097) running on a system in IITKGP has just crashed. Dr Dyuti, as a systems engineer, has been given the responsibility to recover some of the sensitive data, and he needs help. In order to assist him, you need to first find out the value of the *page table base register* (PTBR). Fortunately, the autolog system saved a portion of the physical memory right before the crash happened. The below table shows a snapshot of the physical memory (bytes given in hexadecimal).



Address	Data from <Address + 0x0> to <Address + 0xF>															
	+0	+1	+2	+3	+4	+5	+6	+7	+8	+9	+A	+B	+C	+D	+E	+F
0x30	89	DF	EE	83	2F	AA	B2	C7	EF	A9	2A	33	85	01	A0	29
0x40	22	46	BE	4D	32	2F	B9	CC	AB	13	90	60	24	A0	D8	34
0x50	5A	B7	D4	50	E9	A8	24	38	A7	E9	13	8B	6A	3B	22	41
0x60	46	D9	B7	C5	88	91	25	67	44	22	EE	FF	BB	A2	1F	7D
0x70	16	95	2F	B0	E7	E9	B2	10	19	14	BE	C0	FF	EE	25	66
0x80	78	55	20	10	14	D0	3A	3F	20	6D	3B	C5	D4	A9	FC	10
0x90	00	07	0F	A2	0A	0C	6F	88	16	14	13	08	00	C3	21	6B
0xA0	31	26	2B	45	4F	35	8A	09	CC	4D	44	0B	4A	C9	8A	89
0xB0	A0	BE	3B	29	85	86	75	30	16	BA	EE	C5	0A	24	08	09

Figure 2: Physical Memory before crash (bytes given in hexadecimal notation)

Process 1097's memory accesses are reproduced below. You may assume that no other processes were running in the system and the process made no other memory access besides those shown in the table below.

More precisely, assume that the first memory access occurred at 13ms before the crash and when the process started, the page table was empty.

Dr Dyuti have given you the following information regarding the system:

- Virtual addresses are 10 bits (it's a 10-bit byte addressable system)

Time before the crash (in ms)	**Virtual Address (in hex)**	**Virtual Address (in binary)**	**Page Fault Occured?**	**Data Retreived**
13	0x8A	0010001010	Yes	0xEE
8	0x92	0010010010	Yes	0xBE
3	0x83	0010000011	No	0xC5
1	0xAC	0010101100	Yes	0x85

Table 1: Virtual Memory Addresses accessed by process 1097

- Physical memory is of 256 bytes
- Single-level page table
- All sizes are in powers of 2
- Page Table Entries (PTE) are 8 bits and of the form:

$\leftarrow$ 8 bits $\rightarrow$		
$\langle$ unused $\rangle$	$\langle$ valid bit $\rangle$	$\langle$ PFN $\rangle$

The valid bit is 1-bit in size and the Page Frame Number (PFN) bits and unused bits comprises the rest 7 bits. Now, try to find answers to the following questions which will eventually lead you to find the PTBR and thus the starting address of the page table.

(a) What is the page size? Dr Dyuti did some initial investigations and found out four possible values. Your job is to find the correct one. Please explain your choice.
Hint: Observe the page fault history from Table 1.

A. 4 bytes. B. 8 bytes. C. 16 bytes. D. 32 bytes.

(b) How many page table entries (PTE) are there in the page table?

(c) How many bits compose a page frame number (PFN)?

(d) How are virtual addresses and physical addresses composed? Fill the following tables

Virtual Page Number	**Page Offset**

Physical Frame Number	**Page Offset**

(e) For each of the four virtual address accesses, write down the Page Frame Number (physical frame number) in the column provided in the below table.
Hint: Note that the data for the virtual address in 0x8A is 0xEE. Search for that value in the physical memory and remember the translation method from virtual address to physical address.

Time before the crash (in ms)	**Virtual Address (in hex)**	**Page Fault Occured?**	**Data Retreived**	**Physical Frame Number (PFN)**
13	0x8A	Yes	0xEE	0x
8	0x92	Yes	0xBE	0x
3	0x83	No	0xC5	0x
1	0xAC	Yes	0x85	0x

(f) So finally, what is the value of the Page Table Base Register? This is the address of the start of the page table.
Hint: Remember, to access the PTE, the processor also does a memory access, which should give you the value of the start of the page table. **[3 + 1 + 1 + 2 + 5 + 3 = 15]**

Answer:

(a) Know page size is ≥ 16 because 0x83 does not cause page fault. Know page size < 32 since 0x92 does result in a page fault

(b) Page size = 16B $->$ Page offset = 4 bits. # PTE entries = VPN bits = VA - page offset = 2^6 entries

(c) P.A. = log(256) = 8 bits. P.A. - page offset = $8 - 4 = 4$ bits.

(d) (follows from 1-3)

(e)
1. 0x8A $->$ look for "A" column where data is 0xEE $->$ row 0x60 or 0xB0
2. 0x92 $->$ look for "2" column where data is 0xBE $->$ row 0x40
3. 0x83 $->$ look for "3" column where data is 0xC5 $->$ row 0x60. This also implies the correct physical page number for 0x8A
4. 0xAC $->$ look for "C" column where data is 0x85 $->$ row 0x30

(f) Look for row where offset $8, 9, A$ are $6, 4, 3$ respectively. This happens in row 0x90.

5. While handling the previous issue, Dr Dyuti realised that he needed a deeper understanding of the memory access hierarchy of a system so he does not have to rely on others' help in such cases. In pursuit of knowledge, he opened up his garage and brought back a rusty old machine from old dumps to test his understanding with an unknown system. Well, the poor little machine was beyond repair; however, he managed to get back the specifications from the old datasheet as follows.

System specifications:

- Single level cache memory
- Cache block size 8 bytes
- The cache uses allocate-on-write policy
- Cache is ***virtually addressed***
- Cache hit rate: 80%, cache access time: 1 cycle
- TLB hit rate: 90%, TLB access time: 1 cycle
- Page Table Miss Rate: 1% of TLB misses
- Main memory latency: 200 cycles
- Disk Latency: 100,000 cycles
- Assume the cache and TLB cannot be accessed in parallel
- Page table is one level and always available in memory
- Page table entries are not cacheable
- You can assume that maintenance of any other hardware or software data structures is negligible
- You can assume that the time it takes to forward data to the processor and update the cache, TLB, or Main Memory is negligible
- You can assume that even in the case of a miss in the cache, the access latency will be the same as a hit

Your job is to help him understand the memory access hierarchy by answering the following questions.

(a) Explain with the help of a flow diagram how the memory access process works with respect to memory hierarchy from processor to the disk.

(b) What is the average access latency for the described system? Show your work and write the average latency in cycles. **[5 + 5 = 10]**

Answer:

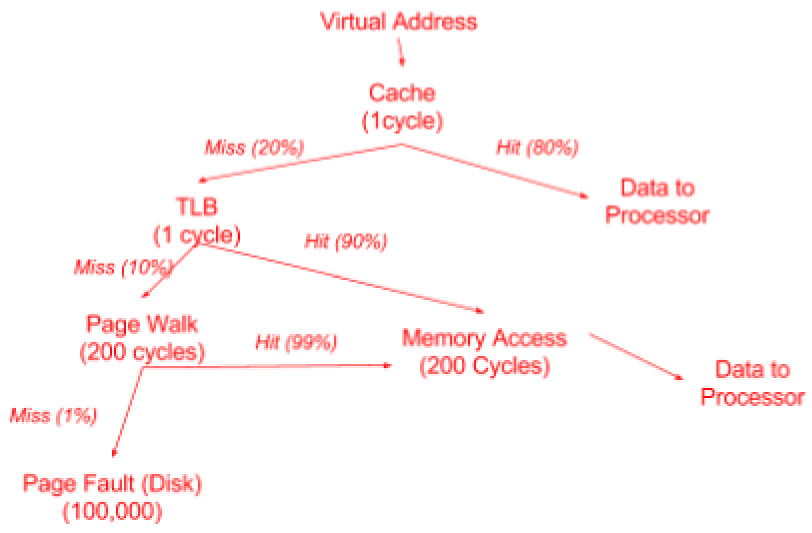


Figure 3: Memory access flow

(a) Please check Fig. 3.

(b) Average latency $= 1 + 0.20(1 + 0.10(200 + 0.01(100,000) + 0.99(200)) + 0.90(200)) = 65.16$ cycles

---