Computer Architecture II Final Examination

Question 1:

(a)

(i)

(1) **Explain the structure of the main memory address in terms of tag, index and word.**

To explain the structure of the main memory address in terms of tag, index and word, we can use calculations to find this. Since there is 1 byte in a cache block, the WORD field must contain 0 bits (2^0 = 1 byte). To determine the number of bits in the INDEX field, we need to determine the number of sets. Each set contains 1 cache block (direct mapping) so a set contains 1 byte. There are 8 bytes in the entire cache, so there are 8 bytes/1 byte = 8 sets. Thus the INDEX field contains 3 bits (2^3 = 8 sets). Finally, since 2^8 = 256 bytes, we have a total of 8 bits, of which 3 is used by the index. We have 5 bits remaining for the TAG. Thus a main memory address is decomposed as shown below.

TAG 5 bits	INDEX 3 bits	WORD 0 bits

(2) **Suppose the cache is empty at the beginning. By showing the relevant steps, deter-mine the hits and misses from the address references in equation (1) and evaluate the final content of the cache**

We can determine the hits and misses from the address references and get the final content of the cache with a table. Note: The X stands for the 0 bytes allocated to the WORD value. In reality, this would be non-existent.

Address	**Binary Address**	**Hit/Miss**	**Set**	**Explain**
5	**00000|101|X**	**Miss**	**(101) = 5**	**Set 5 currently is empty.**
2	**00000|010|X**	**Miss**	**(010) = 2**	**Set 2 is currently empty.**
30	**00011|110|X**	**Miss**	**(110) = 6**	**Set 6 is currently empty.**
20	**00010|100|X**	**Miss**	**(100) = 4**	**Set 4 is currently empty.**
18	**00010|010|X**	**Miss**	**(010) = 2**	**Set 2 had the address 2, not 18.**
1	**00000|001|X**	**Miss**	**(001) = 1**	**Set 1 is currently empty.**

6	**00000|110|X**	**Miss**	**(110) = 6**	**Set 6 has the address 30, not 6.**
2	**00000|010|X**	**Miss**	**(010) = 2**	**Set 2 has the address 18, not 2.**
25	**00011|001|X**	**Miss**	**(001) = 1**	**Set 1 has the address 1, not 25.**
1	**00000|001|X**	**Miss**	**(001) = 1**	**Set 1 has the address 25, not 1.**
32	**00100|000|X**	**Miss**	**(000) = 0**	**Set 0 is currently empty.**
5	**00000|101|X**	**Hit**	**(101) = 5**	**Set 5 does have the address 5!**
8	**00001|000|X**	**Miss**	**(000) = 0**	**Set 0 has the address 32, not 8.**
32	**00100|000|X**	**Miss**	**(000) = 0**	**Set 0 has the address 8, not 32.**
1	**00000|001|X**	**Hit**	**(001) = 1**	**Set 1 does have address 1!**
3	**00000|011|X**	**Miss**	**(011) = 3**	**Set 3 is currently empty.**
11	**00001|011|X**	**Miss**	**(011) = 3**	**Set 3 has the address 3, not 11.**
17	**00010|001|X**	**Miss**	**(001) = 1**	**Set 1 has the address 1, not 17**
3	**00000|011|X**	**Miss**	**(011) = 3**	**Set 3 has the address 11, not 3.**

Final content of the cache:

Set	**Contents**

0	**00100|000|X**
1	**00010|001|X**
2	**00000|010|X**
3	**00000|011|X**
4	**00010|100|X**
5	**00000|101|X**
6	**00000|110|X**
7	**Empty**

(ii) **Repeat Q.1.i using direct mapping with 4-byte blocks**

(1) **Explain the structure of the main memory address in terms of tag, index and word.**

To explain the structure of the main memory address in terms of tag, index and word, we can use calculations to find this. Since there are 4 bytes in a cache block, the WORD field must contain 2 bits (2^2 = 4 bytes). To determine the number of bits in the INDEX field, we need to determine the number of sets. Each set contains 1 cache block (direct mapping) so a set contains 4 bytes. There are 8 bytes in the entire cache, so there are 8 bytes/4 byte = 2 sets. Thus the INDEX field contains 1 bit (2^1 = 2 sets). Finally, since 2^8 = 256 bytes, we have a total of 8 bits, of which 1 is used by the index and 2 bits for the word. We have 5 bits remaining for the TAG. Thus a main memory address is decomposed as shown below.

TAG 5 bits	INDEX 1 bits	WORD 2 bits

(2) Suppose the cache is empty at the beginning. By showing the relevant steps, deter-mine the hits and misses from the address references in equation (1) and evaluate the final content of the cache

Address	**Binary Address**	**Hit/Miss**	**Set**	**Explain**
5	**00000|1|01**	**Miss**	**1**	**Set 1 currently is empty.**
2	**00000|0|10**	**Miss**	**0**	**Set 0 is currently empty.**
30	**00011|1|10**	**Miss**	**1**	**Set 1 does not have address 30.**
20	**00010|1|00**	**Miss**	**1**	**Set 1 does not have address 20.**

18	**00010|0|10**	**Miss**	**0**	**Set 0 does not have address 18.**
1	**00000|0|01**	**Miss**	**0**	**Set 0 does not have address 1.**
6	**00000|1|10**	**Miss**	**1**	**Set 1 does not have address 6**
2	**00000|0|10**	**Miss**	**0**	**Set 0 does not have address 2**
25	**00011|0|01**	**Miss**	**0**	**Set 0 does not have address 25**
1	**00000|0|01**	**Miss**	**0**	**Set 0 does not have address 1**
32	**00100|0|00**	**Miss**	**0**	**Set 0 does not have address 32**
5	**00000|1|01**	**Hit**	**1**	**Set 1 does have the address 5 because of spatial locality (has addresses 4 addresses 3,4,5,6).**
8	**00001|0|00**	**Miss**	**0**	**Set 0 does not have the address 8**
32	**00100|0|00**	**Miss**	**0**	**Set 0 does not have the address 32**
1	**00000|0|01**	**Miss**	**0**	**Set 0 does not have the address 1**
3	**00000|0|11**	**Hit**	**0**	**Set 0 does have the address 3 because of spatial locality (has 4 addresses 1,2,3,4)**
11	**00001|0|11**	**Miss**	**0**	**Set 0 does not have the address 11.**
17	**00010|0|01**	**Miss**	**0**	**Set 0 does not have the**

				address 17.
3	**00000|0|11**	**Miss**	**0**	**Set 0 does not have the address 3.**

Final content of the cache:

Set	**Contents**
0	**00000|1|01, 00000|1|10, 00000|1|11, 00000|1|00,**
1	**00000|0|11, 00000|0|10, 00000|0|00, 00000|0|01**

(iii) **Repeat Q.1.i using two-way set-associative mapping with least recenty used (LRU) replacement algorithm for 2-byte blocks**

(1) Explain the structure of the main memory address in terms of tag, index and word.

To explain the structure of the main memory address in terms of tag, index and word, we can use calculations to find this. The cache size is 8 bytes. The block size is 2 bytes. The block size is equal to the line size. Therefore, the line size is also 2 bytes. Cache size/line size - number of lines, 8 bytes/2 bytes = 4 lines. 2 way set associative means we do the number of lines divided by 2. Therefore, 4/2 = 2 sets.

Since there are 2 bytes in a cache block, the WORD field must contain 1 bit (2^1 = 2 bytes). To determine the number of bits in the INDEX field, we need to determine the number of sets. We already determined above that we have 2 sets with 2 way set associative mapping here. Thus the INDEX field contains 1 bit (2^1 = 2 sets). Finally, since 2^8 = 256 bytes, we have a total of 8 bytes, of which 1 is used by the index and 1 bit for the word. We have 6 bytes remaining for the TAG. Thus a main memory address is decomposed as shown below.

TAG 6 bits	INDEX 1 bit	WORD 1 bit

(2) Suppose the cache is empty at the beginning. By showing the relevant steps, deter-mine the hits and misses from the address references in equation (1) and evaluate the final content of the cache

Address	**Binary Address**	**Hit/Miss**	**Set**	**Explain**
5	**000001|0|1**	**Miss**	**0**	**Set 0 is currently empty.**
2	**000000|1|0**	**Miss**	**1**	**Set 1 is currently empty.**

30	**000111|1|0**	**Miss**	**1**	**Set 1 does not have address 30.**
20	**000101|0|0**	**Miss**	**0**	**Set 0 does not have address 20.**
18	**000100|1|0**	**Miss**	**1**	**Set 1 does not have address 18.**
1	**000000|0|1**	**Miss**	**0**	**Set 0 does not have address 1.**
6	**000001|1|0**	**Miss**	**1**	**Set 1 does not have address 6**
2	**000000|1|0**	**Hit**	**1**	**Set 1 does have address 2 (It has 000000|1|0, 000000|1|1, 000001|1|0, 000001|1|1)**
25	**000110|0|1**	**Miss**	**0**	**Set 0 does not have address 25**
1	**000000|0|1**	**Miss**	**0**	**Set 0 does not have address 1**
32	**001000|0|0**	**Miss**	**0**	**Set 0 does not have address 32**
5	**000001|0|1**	**Miss**	**0**	**Set 0 does not have address 5.**
8	**000010|0|0**	**Miss**	**0**	**Set 0 does not have the address 8**
32	**001000|0|0**	**Miss**	**0**	**Set 0 does not have the address 32**
1	**000000|0|1**	**Miss**	**0**	**Set 0 does not have the address 1**
3	**000000|1|1**	**Hit**	**1**	**Set 1 does have address 3, (It has 000000|1|0, 000000|1|1, 000001|1|0, 000001|1|1)**

11	**000010|1|1**	**Miss**	**1**	**Set 1 does not have the address 11.**
17	**000100|0|1**	**Miss**	**0**	**Set 0 does not have the address 17.**
3	**000000|1|1**	**Miss**	**1**	**Set 1 does not have the address 3.**

Final content of the cache:

Set	**Contents**
0	**000100|0|1, 000100|0|0**
1	**000000|1|1, 000000|1|0**

(iv) Explain in general the advantages and disadvantages of using direct mapping and two-way set-associative cache? For the Q1, part i, ii and iii, compute the miss and hit ratios and comment on which mapping strategy do you prefer from the ratios?

Let's first talk about the advantages and disadvantages of using direct mapping.

Advantages:

1. You only need to compare one tag.
2. It is relatively easy to implement, with each set only having 1 cache block.

Disadvantages:

1. Multiple access to blocks mapped to the same line will cause constant misses and blocks will continuously be swapped.
2. A number of locations are mapped to one fixed location.

Now let's talk about the advantages and disadvantages of using two-way set associative mapping.

Advantages:

1. Since multiple access to blocks that aren't mapped to the same line will avoid the constant misses that direct mapping would offer.
2. A number of locations are not mapped to a fixed location, unlike direct mapping.

Disadvantages:

1. Harder to implement, since each set doesn't only have 1 cache block.
2. You need to compare more than one tag.

The miss and hit ratio is calculated by getting the number of misses/memory accesses, and hits/memory accesses.

Part i:

Hits = 2, Total memory accesses = 19, hence 2/19 = 0.11

Misses = 17, Total memory accesses = 19, hence 17/19 = 0.90

Part ii:

Hits = 2, Total memory accesses = 19, hence 2/19 = 0.11

Misses = 17, Total memory accesses = 19, hence 17/19 = 0.90

Part iii:

Hits = 2, Total memory accesses = 19, hence 2/19 = 0.11

Misses = 17, Total memory accesses = 19, hence 17/19 = 0.90

From the ratios, they are all the same, but I would prefer two way set associative mapping as multiple access to blocks that aren't mapped to the same line will avoid the constant misses that direct mapping would offer and a number of locations are not mapped to a fixed location, unlike direct mapping. This would give more of a chance of a better hit ratio with different addresses.

(b) Consider a 32-bit microprocessor that has an on-chip 16-K Byte four-way set associative cache. Assume that the cache has a line size of four 32-bit words. How is the 32-bit address divided into fields of tag, set index and offset? Show complete working.

So it is a 32 bit microprocessor, and has a 16-K Byte four way set associative cache. The cache line size is 4 32 bit words. To get the offset, we can use the fact that we have 4 bytes per line. Since $2^2 = 4$ bytes, we can say that we need 2 bits for the offset. For the index, we need to use the fact that it is a four way set associative cache. This means that we have 4 lines per set. Therefore we have 16 byte sets. Hence the number of sets is 16-K Byte/16 byte = 1024 sets, meaning 10 bits for the index. We have 32 bits in total as said in the question, of which 2 bits is for the offset, and 10 bits is for the set index. Therefore we have (32-2-10) 20 bits for the tag bits.

TAG 20 bits	SET INDEX 10 bits	OFFSET 2 bits

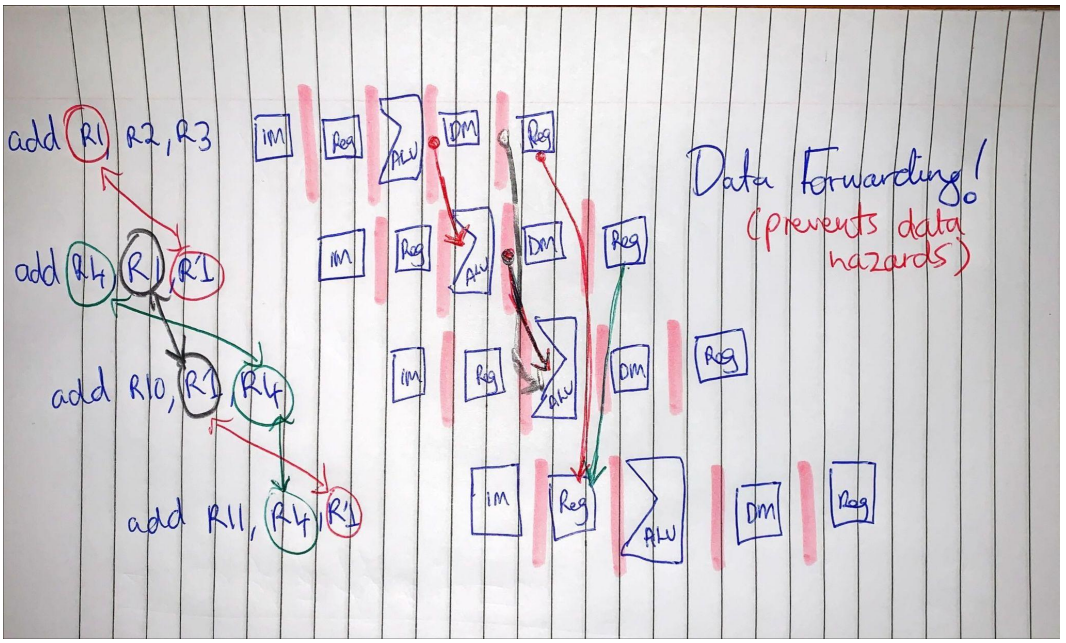
(c) What are data hazards? Describe two techniques (which prevent stalls) that can be used to overcome data hazards in the DLX/MIPS pipeline. Show, using diagrams, how the data hazards in the following code sequence are overcome by the DLX/MIPS CPU

```
i.   r1  = r2 + r3
ii.  r4  = r1 + r1
iii. r10 = r1 + r4
iv.  r11 = r4 + r1
```

Data hazards occur when instructions that have a data dependency use data when they are in different stages of a pipeline. This usually results in the instructions being executed being wrong and resulting in wrong values.

Two techniques to overcome data hazards:

1. We can delay the second instruction after the first instruction which would cause a data dependency, by adding a NOP (no operation) instruction. This slows down the execution but results in the data dependency being removed, and getting each instruction enough time to execute before passing it onto the next instructions using it's parameters.
2. Instead of waiting for the data to be written to a register, and then using a NOP etc, we can simply forward the result to the execution stage (before the ALU), entailing the use of multiplexers and control logic for the implementation.



As you can see above in my diagram, we can see how the data hazards in the code sequence are handled. We can see how after we calculate R1 after the ALU in the first instruction, it is forwarded to the next instruction before it's ALU stage so it has the value R1. Then after the ALU stage in the second instruction, we pass R4 to the next instruction before it's ALU stage so it has R4. By now, R1 has been fully calculated by the first instruction so we can pass it here before the ALU stage as well. Finally, the instruction calculating R4 has been completed now, so we can pass that to the REG stage/or before the ALU stage of the last instruction so it has R4. We can do the same with R1 from the first instruction. This way, with data forwarding, we can avoid data hazards and consequently the need for NOPs.

(d)

Correct the functionality:

```
00 LD R2, R0, 00
04 LD R1, R0, 01      ← switched
08 SRLi R2, R2, 01
0C SUBi R1, R1, 01
10 BNE2 R1, 09        ← changed
14 LD R3, R0, 02
18 ADD R2, R2, R3
1C HALT
```

The program will take
1+1+1+1+1+1+1+1+1+1+1+1+1 = 13 clock cycles

and
1+1+1+1+1+1+1+1 = 8 instructions executed

The numbers are different, because there are 4 stalls at the beginning of the program. This happens before the pipeline has been filled. The number of stalls is proportional to the stages in the pipeline, so 4 stalls here.

Finally, there is a data dependency between LD R2, R0, 0 and SRLi R2, R2, 01. This causes an extra stall,

8 + 4 + 1 = 13

Question 3:

(a) **Consider the following state diagram and describe the behavior of each**

Diagram 1 to the left:

If you were in the predict not taken stage, if a branch instruction is taken (Taken on the diagram), but you do not change the history or your status (so if you are in a loop), you change the behaviour even if the next iteration is taken. For all branches taken in the diagram, you keep on maintaining the same state, and even if it's not taken for once, we do not change the state. You only change the state if it's not taken twice.

Diagram 2 on the right:

If you were in the predict not taken stage on the bottom right, if a branch instruction is taken, then we assume that it's only once, and it might not come again, so we will not change the behaviour. If there is a branch taken from the predict not taken on the bottom left, then we go to the predict taken status where we infinitely keep it that way if a branch is taken.But if it changes at all, we immediately change the state back to predict not taken.

(b) **Assume an instruction cache miss rate for a program is 2% and a data cache miss rate is 4%. If a processor has a CPI of 2 without any memory stalls and the miss penalty is 100 cycles for all misses, determine how much faster a processor would run with a perfect cache that never missed. The instructions which access memory, i.e., the load/store instructions, make up 36% of all instructions**

Instruction cache miss rate = 2%
Data cache miss rate = 4%
CPI = 2
Miss penalty = 100 cycles
Memory instructions = 36% of all instructions

If a processor ran with a perfect cache that never missed, then the instruction cache miss rate and data cache miss rate would be redundant, as we don't need them, since the miss rates for the instructions would now be 0%. CPI = Clock cycles per instruction = 2. Hence it takes 2 clock cycles for each instruction.

CPU Execution time = (CPU clock cycles + memory stall cycles)x clock cycle time
Let's say we ran 100 instructions:
For the nonperfect cache:
2% of the instructions, meaning 2 of them, would have an instruction miss in the cache.
4% of the instructions would have a data miss in the cache. Only 36% of the instructions are memory instructions that would be impacted by this. Therefore 36x0.04 = 1.44% of the instructions would have a data cache miss penalty.
Each penalty is 100 extra cycles. The total penalty would be (2+1.44) x 100 = 344 extra cycles.
Add this to the existing 2 cycles per instruction, 2x100 = 200.
200+344 extra cycles = 544.

For the perfect cache:
No misses happen at all. Let's say we have 100 instructions again.
No misses happen, so we don't have any penalty.
We have 2 cycles per instruction, 2x100 = 200.

The Speedup with the perfect cache would be 544/200 = 2.72.

(c) How does a 32-bit x86 architecture support an extended physical address page of more than 4GB. Explain in details with relevant details

A 32 bit x86 architecture supports an extended physical address page more than 4GB by using a 3 level paging scheme. It uses a physical address space for more than 4GB. It uses Page address extension to do so. By doing this, it can support an extended physical address page of more than 4GB.

(d) **How much space is allocated to a process in the extended physical address page?**

For pentium P6 and later it has a 36-bit address bus, but in terms of space, it has 64 GB of memory through extended physical addressing but only 4 GB of linear space available to a program.

(e) Briefly explain the segmentation and paging architecture of an x32 architecture

This architecture uses segmentation and paging. The segment size is up to 4GB, While the number of segments per process is up to 4GB. It uses 2 partitions for the logical address space. A private partition for up to 8K segments using a local descriptor table with each entry being 8 bytes. It also has a public partition for up to 8K segments using a global descriptor table with each entry being 8 bytes. It has 6 segment registers and six 8 microprogram registers in the tables.

For paging, the page size is 4KB/4MB and it has a 2 level paging scheme.

P1 = Index into the primary/outermost table (page directory)	P2 = Index into the secondary/innermost table (page table)	D → Address offset

(f) What will be the final value in EAX after these instructions execute?

The final value in EAX will be 5, as it is pushed onto the stack first. Therefore, it will be popped out last, into EAX, as 5, in the second pop EAX instruction.

(g) **Why is a register window overflow and underflow mechanism needed? Explain in detail the conditions under which register window underflow and overflow occur and the steps taken to resolve them**

A register window overflow and underflow mechanism is needed because if we have function calls with nesting, this can impact the program if functions nested too deeply. If there are no more register windows to allocate, this would cause a problem.

Register window overflow would happen if for example, we had 8 function calls with nesting, but with 9, there could be a register window overflow as seen with factorial(9) in the lectures.

Register window underflow would happen if for example, we had 8 function calls with nesting, but as we restore discarded registers, we return from each function, and we do not fully return to the original state we were in.