Full Name: Zachary Wilcox

A-number: A02080016

ECE 5720, Fall 2020

December 14, 2020

Instructions:

- Write your A-number on top of every sheet.
- Make sure that your exam is not missing any sheets, then write your full name on the front.
- **The question paper has been reviewed several times to remove any ambiguity. If you still feel a particular problem is ambiguous, then state your assumption clearly and proceed to solve the problem.**
- Write your answers in the space provided below the problem. You may attach extra sheet if necessary. Good luck!

1 (20):
2 (10):
3 (10):
4 (35):
5 (15):
TOTAL (90):

A02090016

Problem 1. (2+5+5+2+8 points):

Consider the source code for the function `funny_recursion()` as given below:

```
int main()
{
textint myArray[]=
{4,6,8,10, 12, 16};
textint val =
funny_recursion(myArray,
4);
textreturn 0;
}
```

```
int funny_recursion(int *a, int N)
{
    if (N <= 1) return 1;
    int temp = a[0] + a[1];
    int val = funny_recursion(a + 1, N -
1);
        return val + temp;
    }
}
```

The assembly code for the function `funny_recursion()` is given below:

```
funny_recursion:
    0x40076d <+0>: mov    $0x1,%eax
    0x400772 <+5>: cmp    $0x1,%esi
    0x400775 <+8>: jle    0x40078c
    0x400777 <+10>: push   %rbx
    0x400778 <+11>: mov    (%rdi),%ebx
    0x40077a <+13>: add    0x4(%rdi),%ebx
    0x40077d <+16>: sub    $0x1,%esi
    0x400780 <+19>: add    $0x4,%rdi
    0x400784 <+23>: callq  0x40076d <funny_recursion>
    0x400789 <+28>: add    %ebx,%eax
     ------------------------------------> Breakpoint
    0x40078b <+30>: pop    %rbx
    0x40078c <+31>: repz retq
```

4
3
2
1 → ret 1

10
14
18

A. During a recursive call to the function *funny_recursion*, what return address is pushed into the stack?

0x400789

question continued on the next page

A02090016

B. During execution, when the first call from *main* enters the function *funny_recursion*, the value of the rsp register was: **0x7fffffffe1d8**. *Now assume that the code reaches the marked breakpoint for the first time (same assumption for part C through E)*, with rip at **0x40078b**. What is the value of rsp at that point? Be sure to show your work how you got to the answer.

rbx = 8 bytes

e1

0x7f...fe1b9

C. What is being stored in rbx and what will it be its value at the breakpoint?

temp,

42

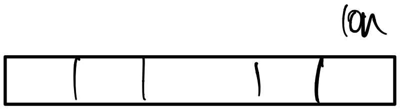
D. At the breakpoint, what do you expect the value of rsi?

1

E. Below is an incomplete snapshot of the stack at the breakpoint shown in *funny_recursion*. Assume that the register rdi has the value **0x7fffffffe1ec**. Fill in the values on the right hand column *in hex*. If you feel an entry cannot be determined, mark it **Cannot be determined** :

0x7fffffffe1e0	0x 7f...fe1d8
0x7fffffffe1e4	0x 0
0x7fffffffe1e8	0x 0x400789
0x7fffffffe1ec	0x A
0x7fffffffe1f0	0x 0x400789
0x7fffffffe1f4	0x E
0x7fffffffe1f8	0x 0x400789
0x7fffffffe1fc	0x F2

A02090016



Problem 2. (3+7 points):

This problem is on a machine with a 5-stage pipeline we studied in class, and a cycle time of 10ns. Assume that you are executing a program where a fraction f, of all instructions are branch instructions, out of which 20% are mis-predicted. Furthermore, assume that we can get the actual CPI in this machine by *only* adjusting for the overheads from branch mis-predictions.

Part A. What is the total execution time of N instructions, in terms of f ?

f = branch 20% miss

.2f = missed. assuming 2cycle penalty = 20ns

20 * .2f = 4f

$\boxed{t = N + N(4f) + 40 \text{ ns}}$

Part B: In a different impementation of the above pipeline, you are considering implementing a machine learning based branch predictor, with a miss-prediction probability of α (i.e., given a branch instruction, the predictor will mis-predict with a probability of α). However, implementing this miss-predictor will increase the time to complete the fetch stage to 13ns. There are now two options: (a) add another *fetch* stage, so that there are two fetch stages and operate this new 6-stage pipeline at the cycle time of 10ns; or (b) increase the pipeline cycle time to 13ns so that the new fetch stage can fit in, keeping the pipeline depth at 5. For a program mix with the characteristics mentioned in Part A, when is the first option better than the second? Your answer should be based on the values of f and α.

6 stage 10ns or 5 stage 13ns

$N + 40 + N(4f)$

$N + 50 + 20 \cdot \alpha \cdot N \cdot f$

$N + 52 + 26 \cdot \alpha \cdot N \cdot f$

α

A is better when $\alpha \cdot f < \frac{1}{3} \cdot N$

A02090016

Problem 3. (5+5 points):

A. There are performance benefits in ensuring that the cache index bits (bits used to select one of the available cache sets) of the physical address falls within the page offset bits in the virtual address. Justify or contradict with appropriate reasoning.

the Page offset remains the same between the VA and the PA, as well as the Page size, so having the CI be within the VPO means that no extra processing will be required to get the index

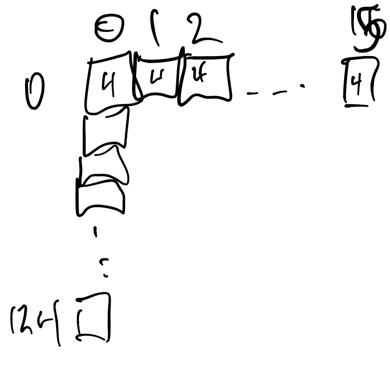
B. Given a page size of 8KB, and that the associativity of the cache can only be up to 16-way, what is maximum cache size that will still ensure that the cache index bits of the physical address falls within the page offset bits in the virtual address? You can assume that memory is byte addressable, and each cache line is 64 bytes.

L = 64 bytes   P_Size = 8000   = 13 bits

CO = 4 bits

CI = 9 bits = 512 lines

125 lines



$512_{\text{lines}} * 64 \text{ bytes} = \boxed{32{,}768 \text{ bytes}}$

A02090016

4. Caches – 35 pts total (14/A, 6/B, 15C)

A. You are given a direct-mapped cache of total size 256 bytes, with cache block size of 16 bytes. The system's page size is 4096 bytes. The following C array has been declared and initialized to contain some values:

```
int x[2][64];
```

i. How many sets will the cache have?

16

ii. How many bits will be required for the cache block offset?

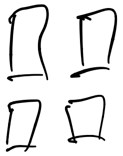
4

iii. If the physical addresses are 22 bits, how many bits are in the cache tag?

22-8= 14 bits $\underbrace{4 \quad 4}$

iv. Assuming that all data except for the array **x** are stored in registers, and that the array **x** starts at address 0x0. Give the miss rate (as a fraction or a %) and total number of misses for the following code, assuming that the cache starts out empty:

```
int sum = 1;
int i;
for (i = 0; i < 64; i++) {
   sum += x[0][i] + x[1][i];
}
```



Miss Rate: 25% Total Number of Misses: 32

v. What if we maintain the same total cache size and cache block size, but increase the associativity to 2-way set associative. Now what will be the miss rate and total number of misses of the above code, assuming that the cache starts out empty?

Miss Rate: 12.5% Total Number of Misses: 16

A02090016

4. (cont.)

B. Given the following access results in the form (address, result) on an empty cache of total size 16 bytes, what can you infer about this cache's properties? Assume LRU replacement policy. **Circle all that apply**.

(0, Miss), (8, Miss), (0, Hit), (16, Miss), (8, Miss)

8

X a. The block size is greater than 8 bytes

[✓ b. The block size is less or equal to 8 bytes]

16

[✓ c. This cache has only two sets]

X d. This cache has more than 8 sets

[✓ e. This cache is 2-way set associative]

X f. The cache is 4-way set associative

g. Using an 8 bit address, the tag would be 4 bits

[✓ h. Using an 8 bit address, the tag would be greater than 4 bits]

i. None of the above

1

A02090016

4. (Cont)

C. Given the following 2-way set-associative cache and its contents in a system with a 10-bit address:

CI=2

CO=3

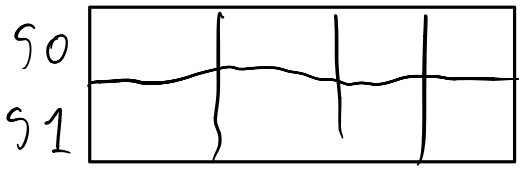
Index	Tag	V	B0	B1	B2	B3	B4	B5	B6	B7	Tag	V	B0	B1	B2	B3	B4	B5	B6	B7
0	07	1	99	1F	34	56	99	1F	34	56	11	1	DE	AD	BE	EF	DE	AD	BE	EF
1	03	1	27	A4	C5	23	00	00	00	01	1C	1	1F	2E	11	09	1F	2E	11	09
2	01	1	54	21	65	78	54	21	65	78	0F	0	CA	FE	12	34	CA	FE	12	34
3	0F	1	01	02	03	04	05	06	07	08	1C	0	12	34	56	78	13	24	57	68

What are the results of the following read operations (specify whether it is a hit or miss and the value if is determinable from the information given, otherwise just write ND for non-determinable)? Assume the cache uses a LRU replacement policy and that reads are executed in the order given below (addresses are given in hex).

Address to be read	**Tag (give bits)** 9-5	**Set (give bits)** 4-3	**Block Offset (give bits)** 2-0	**Hit or Miss (H or M)**	**Value read (or ND)**
0x389	11100	01	001	H	2E
0x30C	11000	01	100	M	ND
0x3BB	11101	11	011	M	ND
0x308	11000	01	000	M	ND
0x0E3	00111	00	011	H	56

A02090016

5. Virtual Memory (15 points)



Assume we have a virtual memory detailed as follows:

- 16 KiB Virtual Address Space,
- 4 KiB Physical Address Space,
- a TLB with 8 entries that is 4-way set associative with LRU replacement
- 128 B page size

a) [5 pts] How many bits will be used for:

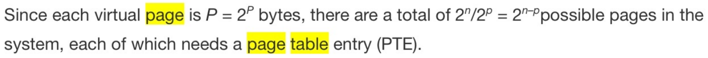
Page offset? 7

Virtual Page Number (VPN)? 7 Physical Page Number (PPN)? 5

TLB index? 1 TLB tag? 6

b) [1 pt] How many TOTAL entries are in this page table?
(It is fine to leave your answer as powers of 2).

128 (2^7)



Since each virtual page is $P = 2^p$ bytes, there are a total of $2^n/2^p = 2^{n-p}$ possible pages in the system, each of which needs a page table entry (PTE).

A02090016

5. (cont.) The current contents of the TLB and (partial) Page Table are shown below:

TLB

Set	Tag	PPN	Valid	Tag	PPN	Valid	Tag	PPN	Valid	Tag	PPN	Valid
0	16	-	0	2F	05	1	2A	04	1	04	1B	1
1	06	0A	1	3C	01	1	01	15	0	12	-	0

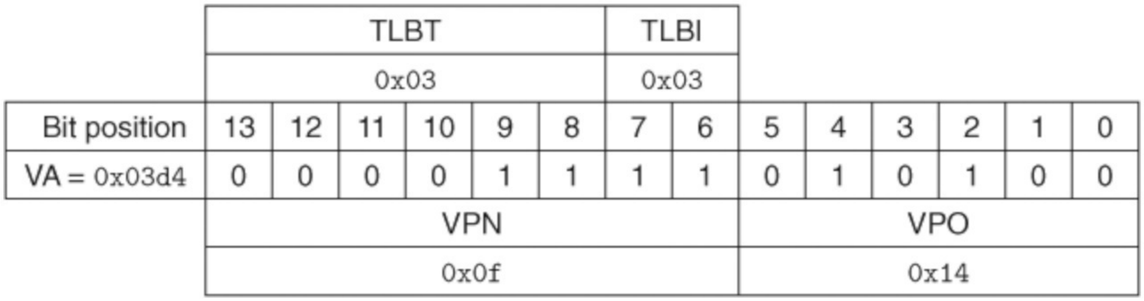
Page Table (only first 16 of the PTEs are shown)

VPN	PPN	Valid	VPN	PPN	Valid	VPN	PPN	Valid	VPN	PPN	Valid
00	1A	1	04	08	1	08	1B	1	0C	-	0
01	05	1	05	-	0	09	0A	1	0D	0A	1
02	16	1	06	1C	1	0A	04	1	0E	01	1
03	15	1	07	-	0	0B	1E	1	0F	-	0

c) [9 pts] Determine the physical address, TLB miss or hit, and whether there is a page fault for the following virtual address accesses (write "Y" or "N" for yes or no, respectively, in the TLB Miss? And Page Fault? columns). If you can't determine the PPN and/or physical address and/or TLB miss and/or Page Fault, simply write ND (for non-determinable) in the appropriate entry in the table.

Virtual Address	VPN	TLBT	TLBI	PPN	Physical Address	TLB Miss?	Page Fault?
0x046A	8	10	0	1B	0x DEA	Y	no
0x1A8F	9F	4F	1	ND	ND	Y	Y
0x027E	4	10	0	08	0x 47E	Y	N

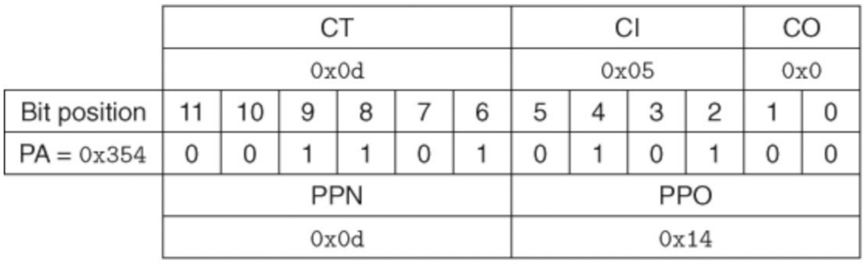
6 1



	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	TLBT						TLBI							
	0x03						0x03							
Bit position	13	12	11	10	9	8	7	6	5	4	3	2	1	0
VA = 0x03d4	0	0	0	0	1	1	1	1	0	1	0	1	0	0
	VPN								VPO					
	0x0f								0x14					

7 7

14 total



	CT						CI				CO	
	0x0d						0x05				0x0	
Bit position	11	10	9	8	7	6	5	4	3	2	1	0
PA = 0x354	0	0	1	1	0	1	0	1	0	1	0	0
	PPN						PPO					
	0x0d						0x14					

5