System Skill Final Quiz

Date:Thursday, July 21st, 2022
Due: Friday, July 22nd, 2022 at 11.59PM
Instructor: Rachata Ausavarungnirun

Problem 1 (25 Points):	
Problem 2 (10 Points):	
Problem 3 (20 Points):	
Problem 4 (20 Points):	
Problem 5 (30 Points):	
Extra Credit (10 Points):	
Total (100+15 Points):	

Instructions:

1. This is a 38-hour exam. **If you get 100, you get a full score. Any points above 100 goes to your extra credit at the conversion rate of 50% per point.**
2. Submit your work as a pdf file on Canvas.
3. Clearly indicate your final answer for each conceptual problem.
4. **DO NOT CHEAT.** If we catch you cheating in any shape or form, you will be penalized based on **my plagiarism policy** ($N*$ 10% of your total grade, where N is the number of times you plagiarized previously). This also includes asking the internet for the answer to our questions.

Tips:

- **Read everything.** Read all the questions on all pages first and formulate a plan.
- **Be cognizant of time.** It is a sad day if you click submit when the submission site close.
- **Canvas allows resubmission.** I will take a look at the last version you submit.
- **Show work when needed.** You will receive partial credit at the instructors' discretion.

1. Assembly and ISA! [25 points]

During the semester, we learn how x86 can be assembled and deassembled into assembly code and binaries in assignment 4 task 2. In this question, one of your TAs would like to be cool and design his own ISAs. Consider the following 16-bit MUIC-IS-COOL-ISA with the following features.

- The ISA is byte addressable and there are 8 16-bit registers from R0 to R7.
- The machine stop its execution whenever the decoder observes the instruction `JMP 0`, in which case it finish all remaining instructions in the pipeline.
- There are 3 status bits: negative, zero, overflow. **The negative bit** is set to true if the destination register yield a negative result, in which case value of the destination register is the leftmost 16 bits. **The zero bit** is set to true if the destination register yield zero. **The overflow bit** is set to true if the destination register overflows (i.e., result in a number higher than 16 bits, in which case the destination register stores the leftmost 16 bits value).

Instruction Type	**Opcode**	**Op1**	**Op2**	**Op3**	**Unused**
Bits location	Higher bits <----------> Lower bits				
Compute R Rs1, Rs2, Rd	4 bits	3 bits	3 bits	3 bits	3 bits
Compute I Rs1, Rd, IMM	4 bits	3 bits	3 bits	6 bits	None
Memory Type 1 Rd, Rs, IMM	4 bits	3 bits	3 bits	6 bits	None
Memory Type 2 Rd, IMM	4 bits	3 bits	9 bits		None
Cond. Type 1 IMM	4 bits	12 bits			None
Cond. Type 2 Rd	4 bits	3 bits			6 Bits
Cond. Type 3 Rs1, Rs2, Rd	4 bits	3 bits	3 bits	3 bits	3 Bits

Table 1: Bit pattern for each instruction types. The most significant bit is on the leftmost side and the least significant bit is on the rightmost side.

Instruction	Opcode	Op1	Op2	Op3	Description
ADD	0000	Rs1	Rs2	Rd	$R_d <= R_{s1} + Rs2$
ADDI	0001	Rs1	Rd	IMM	$R_d <= R_{s1} + IMM$
SUB	0010	Rs1	Rs2	Rd	$R_d <= R_{s1} - Rs2$
SUBI	0011	Rs1	Rd	IMM	$R_d <= R_{s1} - IMM$
AND	0100	Rs1	Rs2	Rd	$R_d <= R_{s1} and Rs2$
OR	0101	Rs1	Rs2	Rd	$R_d <= R_{s1} or Rs2$
XOR	0110	Rs1	Rs2	Rd	$R_d <= R_{s1} xor Rs2$
LD	0111	Rd	Rs	IMM	$R_d <= mem[Rs + IMM]$
LDI	1000	Rd	IMM		$R_d <= IMM$
ST	1001	Rd	Rs	IMM	$R_s => mem[Rd + IMM]$
JMP	1010	IMM			$PC <= IMM$
JMPR	1011	Rd			$PC <= Rd$
BLT	1100	Rs1	Rs2	Rd	If $Rs1 < Rs2$ then $PC <= Rd$ else $PC <= PC + 2$
BGT	1101	Rs1	Rs2	Rd	If $Rs1 > Rs2$ then $PC <= Rd$ else $PC <= PC + 2$
BNE	1110	Rs1	Rs2	Rd	If $Rs1 = Rs2$ then $PC <= Rd$ else $PC <= PC + 2$
LDPC	1111	Rd			$Rd <= PC$

Table 2: All instructions in the MUIC-IS-COOL-ISA. Rs1 is the input source 1, Rs2 is the input source 2, Rd is the destination (output), and IMM is the immediate (constant value). Register bits are denoted based on their register ID. For example, if Rs1 is R3, it will have the value equal to 3 in the appropriate register field in the binary instruction.

(a) Now that we have establish the ISA specification. (15 points)

Assume PC starts at `0x30`. What is the code (in MUIC-IS-COOL assembly) from the memory snapshot below. **Note that for this memory snapshot, the bits within the data word in the table below is sorted using [highest bit – lowest bit] format (i.e., if the data word is `0x1234`, then the word is 0b'0001 0010 0011 0100).**

Address	Values (in hex) [Lowest address – Highest address]
0x00	`00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f`
0x10	`00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f`
0x20	`00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f`
0x30	`80 00 8a 00 8c 14 fe f2 72 14 0a 6b 10 02 e1 ba`
0x40	`91 40 a0 00 e2 ff 01 0f ff 2e be ef 24 31 a0 00`
0x50	`19 15 12 0a 6b 3a 4b 12 91 ac ff fe 3c 3d 3e 4f`
0x60	`12 50 62 8a 5e 5f df ea 99 ac 74 6b 91 44 33 ef`
0x70	`70 71 72 73 74 75 76 77 78 79 7a 7b 7c 7d 7e 7f`
0x80	`80 81 82 83 84 85 86 87 88 89 8a 8b 8c 8d 8e 8f`
0x90	`90 91 92 93 94 95 96 97 98 99 9a 9b 9c 9d 9e 9f`
0xa0	`80 00 8a 00 8c 14 fe ff 72 14 0a 6b 10 02 e1 ba`
0xb0	`91 40 8a 00 8c 14 fe ff 74 13 02 ba 6b 12 4b 31`
0xc0	`80 00 8a 00 8c 14 fe ff 72 14 0a 6b 10 01 e1 ba`
0xd0	`91 40 8a 00 8c 14 fe ff 74 13 02 ba 6b 12 4b 31`
0xe0	`70 00 8a 00 8c 14 fe ff 72 14 0a 6b 10 03 e1 ba`
0xf0	`91 40 8a 00 8c 14 fe ff 74 13 02 ba 6b 12 4b 31`

Instruction	Bits Pattern
LDI R0 0	10000000 00000000
LDI R5 0	10001010 00000000
LDI R6 0x14	10001100 00010100
LDPC R7	1111111x xxxxxxxx
LD R1 R0 0x14	01110010 00010100
ADD R5 R1 R5	00001010 01101xxx
ADDI R0 R0 2	00010000 00000010
BNE R0 R6 R7	11100001 10111xxx
ST R0 R5 0x00	10010001 01000000
JMP 0	10100000 00000000

(b) What are the values of all the registers inside the register files after the program finishes? You can put in XX for an unknown value. (10 points)

Address	Values (in Decimal)
R0	20
R1	30
R2	XX
R3	XX
R4	XX
R5	198
R6	20
R7	166

16 bits → 2 bytes = +2

Address	Values (in hex) [Lowest address – Highest address]
0x00	00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f
0x10	00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f
0x20	00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f
0x30	80 00 | 8a 00 | 8c 14 | fe f2 | 72 14 | 0a 6b | 10 02 | e1 ba (0x36, 0x38, 0x3a, 0x3c, 0x3e)
0x40	91 40 | a0 00 | e2 ff 01 0f ff 2e be ef 24 31 a0 00 (0x40, 0x42)
0x50	19 15 12 0a 6b 3a 4b 12 91 ac ff fe 3c 3d 3e 4f
0x60	12 50 62 8a 5e 5f df ea 99 ac 74 6b 91 44 33 ef
0x70	70 71 72 73 74 75 76 77 78 79 7a 7b 7c 7d 7e 7f
0x80	80 81 82 83 84 85 86 87 88 89 8a 8b 8c 8d 8e 8f
0x90	90 91 92 93 94 95 96 97 98 99 9a 9b 9c 9d 9e 9f
0xa0	80 00 8a 00 8c 14 fe ff 72 14 0a 6b 10 02 e1 ba
0xb0	91 40 8a 00 8c 14 fe ff 74 13 02 ba 6b 12 4b 31
0xc0	80 00 8a 00 8c 14 fe ff 72 14 0a 6b 10 01 e1 ba
0xd0	91 40 8a 00 8c 14 fe ff 74 13 02 ba 6b 12 4b 31
0xe0	70 00 8a 00 8c 14 fe ff 72 14 0a 6b 10 03 e1 ba
0xf0	91 40 8a 00 8c 14 fe ff 74 13 02 ba 6b 12 4b 31

A

	Instruction	Bits Pattern	
1	LDI R0 0	10000000 00000000	R0 = 0
2	LDI R5 0	10001010 00000000	R5 = 0
3	LDI R6 0x14	10001100 00010100	R6 = 20
4	LDPC R7	1111111x xxxxxxxx	R7 = 56
5	LD R1 R0 0x14	01110010 00010100	R1 = 20
6	ADD R5 R1 R5	00001010 01101xxx	R5 = 20
7	ADDI R0 R0 2	00010000 00000010	R0 = 2
8	BNE R0 R6 R7	11100001 10111xxx	R7 = 66
9	ST R0 R5 0x00	10010001 01000000	R5 =
10	JMP 0	10100000 00000000	

Instruction	Opcode	Op1	Op2	Op3	Description
ADD	0000	Rs1	Rs2	Rd	$R_d <= R_{s1} + Rs2$
ADDI	0001	Rs1	Rd	IMM	$R_d <= R_{s1} + IMM$
SUB	0010	Rs1	Rs2	Rd	$R_d <= R_{s1} - Rs2$
SUBI	0011	Rs1	Rd	IMM	$R_d <= R_{s1} - IMM$
AND	0100	Rs1	Rs2	Rd	$R_d <= R_{s1} and Rs2$
OR	0101	Rs1	Rs2	Rd	$R_d <= R_{s1} or Rs2$
XOR	0110	Rs1	Rs2	Rd	$R_d <= R_{s1} xor Rs2$
LD	0111	Rd	Rs	IMM	$R_d <= mem[Rs + IMM]$
LDI	1000	Rd	IMM		$R_d <= IMM$
ST	1001	Rd	Rs	IMM	$R_s => mem[Rd + IMM]$
JMP	1010	IMM			$PC <= IMM$
JMPR	1011	Rd			$PC <= Rd$
BLT	1100	Rs1	Rs2	Rd	If $Rs1 < Rs2$ then $PC <= Rd$ else $PC <= PC + 2$
BGT	1101	Rs1	Rs2	Rd	If $Rs1 > Rs2$ then $PC <= Rd$ else $PC <= PC + 2$
BNE	1110	Rs1	Rs2	Rd	If $Rs1 = Rs2$ then $PC <= Rd$ else $PC <= PC + 2$
LDPC	1111	Rd			$Rd <= PC$

Instruction Type	Opcode	Op1	Op2	Op3	Unused
Bits location	Higher bits <-----------> Lower bits				
Compute R Rs1, Rs2, Rd	4 bits	3 bits	3 bits	3 bits	3 bits
Compute I Rs1, Rd, IMM	4 bits	3 bits	3 bits	6 bits	None
Memory Type 1 Rd, Rs, IMM	4 bits	3 bits	3 bits	6 bits	None
Memory Type 2 Rd, IMM	4 bits	3 bits	9 bits		None
Cond. Type 1 IMM	4 bits	12 bits			None
Cond. Type 2 Rd	4 bits	3 bits			6 Bits
Cond. Type 3 Rs1, Rs2, Rd	4 bits	3 bits	3 bits	3 bits	3 Bits

What are the values of all the registers inside the register files after the program finishes? You can put in XX for an unknown value. (10 points)

B

Address	Values (in Decimal)
R0	20
R1	30
R2	XX
R3	XX
R4	XX
R5	198
R6	20
R7	166

1. 80 00 → 1000 0000 0000 0000 (opcode, R0, 0)
(memory type 2)
∴ LDI R0 0

2. 8a 00 → 1000 1010 0000 0000 (opcode, R5, 0)
(memory type 2)
∴ LDI R5 0

3. 8C 14 → 1000 1100 0001 0100 (opcode, R6, 14)
∴ LDI R6 0x14

4. fe f2 → 1111 1110 1111 0010 (opcode, R7, unused)
(Cond. Type 2)
∴ LDPC R7

5. 72 14 → 0111 0010 0001 0100 (opcode, R1, R0, 0x14)
(Memory Type 1)
∴ LD R1 R0 0x14

6. 0a 6b → 0000 1010 0110 1011 (opcode, R5, R1, R5, unused)
(compute R)
∴ ADD R5 R1 R5

7. 10 02 → 0001 0000 0000 0010 (R0, R0, R2)
∴ ADDI R0 R0 R2

8. e1 ba → 1110 0001 1011 1010 (R0, R6, R7, unused)
∴ BNE R0 R6 R7

9. 91 40 → 1001 0001 0100 0000 (R0, R5, R0)
∴ ST R0 R5 0x00

* PC = Address of the next instruction right after call

56

$\log_2(16) = 4$

10100110

10100110

2. Code Size [10 points]

For the following code, please fill in the number (in **hexadecimal base**) for the address of each instruction.

Address	Instruction (in binary)	Instruction (in Assembly)
5fa:	55	push %rbp
5fb	48 89 e5	mov %rsp,%rbp
5fe	89 7d fc	mov %edi,-0x4(%rbp)
601	89 75 f8	mov %esi,-0x8(%rbp)
604	8b 55 fc	mov -0x4(%rbp),%edx
607	8b 45 f8	mov -0x8(%rbp),%eax
60a	01 d0	add %edx,%eax
60c	5d	pop %rbp
60d	c3	retq
60e:	55	push %rbp
60f	48 89 e5	mov %rsp,%rbp
612	48 83 ec 08	sub $0x8,%rsp
616	89 7d fc	mov %edi,-0x4(%rbp)
619	89 75 f8	mov %esi,-0x8(%rbp)
61c	8b 55 f8	mov -0x8(%rbp),%edx
61f	8b 45 fc	mov -0x4(%rbp),%eax
622	89 d6	mov %edx,%esi
624	89 c7	mov %eax,%edi
626	e8 cf ff ff ff	callq 5fa

a = 10
b = 11
c = 12
d = 13
e = 14
f = 15

3. Jump Table [20 points]

In this question, consider the following assembly codes below. Fill in the rest of the C code for each of the switch cases. Write "NOTHING HERE" if the space should be left blank or if that line of code should not exist (i.e., the program does not support to modify `result` at that line).

Assume that both a is in `%rdi` and b is in `%rsi`

source , dest

```
quiz3:
pushl %ebp
movl %esp, %ebp     ebp = esp
movl %rdi, %edx     edx = a
movl %rsi, %eax     eax = b
cmpl $8, %edx       = a
ja .L8
jmp *.L9(,%edx,4)   .L9 + edx*4
.section .rodata
.align 4
.align 4
.L9:
.long .L8 0
.long .L4 1
.long .L8 2
.long .L5 3
.long .L8 4
.long .L4 5
.long .L6 6
.long .L8 7
.long .L2 8
.text
1,5  .L4:
movl %edx, %eax     result
                    eax = a
jmp .L2
3    .L5:
decl %eax           b-1
jmp .L2
6    .L6:
incl %eax           b+1
jmp .L2
default case
0,2,4,7  .L8:
movl $-1, %eax      result = -1
.L2:
popl %ebp
ret
```

In the space below, fill in the blank to reflect the assembly code above.

```
int quiz3(int a, int b)
{
int result = ___b___;

int temp = ________; // Feel free to use this if you need to store
                     // any temp variable. Leave blank if not needed.

switch(___a___)
{

    case ___1___:
    case ___5___:
        result = ___a___;
        break;

    case ___3___:

        result = ___b-1___;
        break;

    case ___6___:

        result = ___b+1___;
        break;

    case ________:

        result = nothing here;
        break;

    default:

        result = ___-1___;
}
return result;
}
```

tag	set index	block offset
t bits	s bits	b bits

block size = B = 32

$b = \log_2 B = \log_2 32 = 5$

$s = \log_2 b = \log_2 S = \log_2 4 = 2$

$t = \ell - b - s = 16 - 5 - 2 = 9$

Initials:

---

4. Caching [20 points]

In this question, let's assume that we have a 16-bit system with a single level 5-way set associative cache with 4 sets, and a cache block size of 32 bytes.

How many bits are needed for the setID and the tags? Draw the breakdown of the tag/index/byte-in-block bits.

> 2 bits for setID, 9 bits tag, 5 for byte in block

What is the total size of this cache?

> $5 \times 32 \times 4 = 640$ bytes

For the following program, **assume that an integer is 4 bytes**.

```
int i; // Assume these variables are stored in the registers.
int a[65536]; // Assume that a = 0x1000
int b[65536]; // Assume that b = 0x8000000

for(i=0;i<2048;i++)
    a[i * __X__ ] = i;

for(i=0;i<2048;i++)
    b[i * __Y__ ] = a[i * __Z__ ]++;
```

$65536 \times 4 = 262{,}144$ bytes
but cache is only 640 bytes

Let's asssume that the array is initialized to zero. Is it possible for me to have a combination of X, Y and Z such that the cache hit rate is 0%. Why or why not? **Show your work.**

cache = 640 bytes $\to \frac{640}{4} = 160$ ints

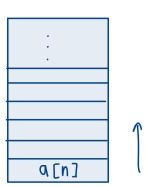
$Max_x = \frac{65536}{2048} = 32$

$x, y, z \leq 32$ otherwise will exceed arr size.

x, y, z cannot be 0 because $a[i*0] = a[0]$ so we will only have a[0] in cache (will always hit)

if $x, y, z > 32$ index of a, b will exceed 65536 (index out of range)

$1 \leq x, y, z \leq 32$



last 160 values of a.

Level 1	Level 2	Page Offset
10	10	12

page size

Page offset = $\log_2(4K)$ = 12 bits
1^{st} level = 10 bits
2^{nd} level = 10 bits
} 32 bit-address

5. Virtual Memory [30 points]

Let's create a simple **BIG endian** machine that utilize two-level page table with a 4KB page size (similar to what we learn in class), 4KB page table, and this processor also uses 32-bit address. Assuming the following:

$\log_2(4KB)$ = 12 bits

- Data in the memory and the page table root is at `0x10`.
- The status bits in the PTE for both levels are 12 bits, and the page table entries is 32-bit long, where the n most significant bits after the page offset are either used as the ID of the next page (for the first level) or the physical page number (for the second level).
- To get the address of the first entry in the second level of the page table, our machine will take the ID of the next page. Then, it appends this ID with m additional zero bits, where m is the number of bits required for the page table size. For example, if your page table size is 64 bytes and the ID is 5, m is 6. So, the next level page for this ID is at address 0x140.

$\log_2 64 = 6$

101000000 = 320 = 0x140

Address	**Values (in hexadecimal) [Lowest byte – Highest byte]**			
0x00	00 10 20 30	40 50 60 70	80 90 a0 b0	c0 d0 e0 f0
0x10	10 11 12 13	14 15 16 17	18 19 1a 1b	08 00 00 00
0x20	19 15 12 0a	6b 3a 60 70	19 15 12 dd	6b d0 e0 f0
0x30	30 31 ee 33	34 35 36 37	00 10 0e aa	3c 3d 3e 3f
0x40	00 10 20 30	40 50 60 70	80 90 a0 b0	c0 d0 e0 f0
0x50	19 15 12 0a	6b 3a 4b 12	91 ac ff fe	3c 3d 3e 4f
0x60	12 50 62 8a	5e 5f df ea	99 ac 74 6b	91 44 33 ef
0x70	70 71 72 73	74 75 76 77	78 79 7a 7b	7c 7d 7e 7f
0x80	91 40 8a 00	8c 14 fe ff	74 13 02 ba	6b 12 4b 31
0x90	90 91 92 93	94 95 96 97	98 99 9a 9b	9c 9d 9e 9f
0xa0	80 00 8a 00	8c 14 fe ff	72 14 0a 6b	10 02 e1 ba
0xb0	30 31 32 33	34 35 36 37	38 39 3a 3b	3c 3d 3e 3f
0xc0	80 00 8a 00	8c 14 fe ff	72 14 0a 6b	10 01 e1 ba
0xd0	91 40 8a 00	8c 14 fe ff	74 13 02 ba	6b 12 4b 31
0xe0	70 00 8a 00	8c 14 fe ff	72 14 0a 6b	10 03 e1 ba
0xf0	91 40 8a 00	8c 14 fe ff	74 13 02 ba	6b 12 4b 31
0x100000	00 10 20 30	40 50 60 70	80 90 a0 b0	c0 d0 e0 f0
0x100010	10 11 12 13	14 15 16 17	18 19 1a 1b	1c 1d 1e 1f
0x100020	00 10 20 30	40 50 60 70	80 90 a0 b0	c0 d0 e0 f0
0x100030	30 31 32 33	34 35 36 37	38 39 3a 3b	3c 3d 3e 3f
0x100040	00 10 20 30	40 50 60 70	80 90 a0 b0	c0 d0 e0 f0
0x100050	19 15 12 0a	6b 3a 4b 12	91 ac ff fe	3c 3d 3e 4f
0x100060	12 50 62 8a	5e 5f df ea	99 ac 74 6b	91 44 33 ef
0x100070	70 71 72 73	74 75 76 77	78 79 7a 7b	7c 7d 7e 7f
0x8000000	91 40 8a 00	8c 14 fe ff	74 13 02 ba	6b 12 4b 31
0x8000010	90 91 92 93	94 95 96 97	98 99 9a 9b	9c 9d 9e 9f
0x8000020	80 00 8a 00	8c 14 fe ff	72 14 0a 6b	10 02 e1 ba
0x8000030	30 31 32 33	34 35 36 37	38 39 3a 3b	3c 3d 3e 3f
0x8000040	80 00 8a 00	8c 14 fe ff	72 14 0a 6b	10 01 e1 ba
0x8000050	91 40 8a 00	8c 14 fe ff	74 13 02 ba	6b 12 4b 31
0x8000060	70 00 8a 00	8c 14 fe ff	72 14 0a 6b	10 03 e1 ba
0x8000070	91 40 8a 00	8c 14 fe ff	74 13 02 ba	6b 12 4b 31

0|1|2|3
$\underbrace{\quad}_{20}$ $\underbrace{\quad}_{12}$

10111000

4×31
= 124

08 00 0000

6b 12 4b 31

6b 12 4~~ff~~ → PA

(a) What is the physical address for a virtual address `0x0000beef`? Put in **Not enough information** if the table does not provide enough information to get the physical address.

0x0000 beef

0000 0000 0000 0000 1011 1110 1110 1111

4K

$\log_2(N) = ?$

c

(b) What is the physical address for a virtual address `0x00803fff`? Put in **Not enough information** if the table does not provide enough information to get the physical address.

0000 0000 0011 1111 1111 1111

level 1 level 2 = 3

(c) What is the physical address for a virtual address `0x0000beef` if this system were to use a single level 16KB page instead? Put in **Not enough information** if the table does not provide enough information to get the physical address.

0x0000beef

$\log_2(16KB) = 14$ bits

$\underbrace{0000\ 0000\ 0000\ 10}_{2}\underbrace{11\ 1110\ 1110\ 1111}_{\text{page offset}}$

0x18191a1b
↓

PPN = 0001 1000 0001 1001 00 | 0001 1000 0001 1001 0011 1110 1110 1111

1 8 1 9 3 E E F

PO = 11 1110 1110 1111

= 0x18193EEF

(d) Assuming that the memory access takes 100 cycles to access DRAM, the system has 4-level page table (i.e., a page walk have to access the memory 4 times before it can access its data), an TLB access takes 1 cycle, and a L1 cache access to the set takes 1 cycle and the tag comparison in the L1 cache takes another 1 cycle. How long does it takes to load a data that has a TLB miss and a L1 cache hit? Feels free to explain your answer.

check TBL → 400 + 1

L1 → 1

402

2401

6. Extra Credit: 0x44444444 [10 points]

Do not attempt this until you are done with other questions.

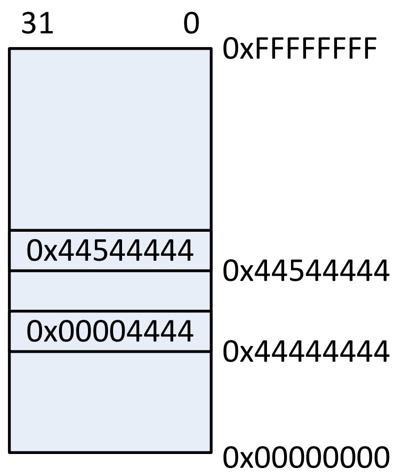
A 32-bit processor implements paging-based virtual memory using a single-level page table. The following are the assumptions about the processor's virtual memory.

- A page table entry (PTE) is 4-bytes in size.
- A PTE stores the physical page number in the least-significant bits.
- The base address of the page tables is page-aligned.

The following figure shows the physical memory of the processor at a particular point in time.

Virtual address space

$= m$-bit or 2^m bytes $= 2^{32}$ bytes



4GB Physical Memory

At this point, when the processor executes the following piece of code, it turns out that the processor accesses the page table entry residing at the physical address of `0x44444444`.

```
char *ptr = 0x44444444;
char val = *ptr; // val == 0x44
```

What is the page size of the processor? Show work in detail.

Log Table

N	$log_2 N$
1	0
2	1
4	2
8	3
16	4
32	5
64	6
128	7
256	8
512	9
1024 (1k)	10
2048 (2k)	11
4096 (4k)	12
8192 (8k)	13
16384 (16k)	14
32768 (32k)	15
62236 (64k)	16
131072 (128k)	17
262144 (256k)	18
524288 (512k)	19
1048576 (1M)	20
2097152 (2M)	21
4194304 (4M)	22
8388608 (8M)	23
16777216 (16M)	24

Stratchpad

Stratchpad