Wawrzynek, Weaver
Fall 2021

CS 61C

Final

PRINT your name: _____, _____
(last) (first)

PRINT your student ID: _____

Read the following honor code and sign your name.

> I understand that I may not collaborate with anyone else on this exam, or cheat in any way. I am aware of the Berkeley Campus Code of Student Conduct and acknowledge that academic misconduct will be reported to the Center for Student Conduct and may further result in, at minimum, negative points on the exam and a corresponding notch on Nick's Stanley Fubar demolition tool.

SIGN your name: _____

You have 170 minutes. There are 9 questions of varying credit (100 points total).

For questions with **circular bubbles**, you may select only one choice.

- ◯ Unselected option (completely unfilled)
- ● Only one selected option (completely filled)

For questions with **square checkboxes**, you may select one or more choices.

- ■ You can select
- ■ multiple squares (completely filled).

Anything you write that you ~~cross out~~ will not be graded. Anything you write outside the answer boxes will not be graded.

If an answer requires hex input, make sure you only use capitalized letters! For example, `0xDEADBEEF` instead of `0xdeadbeef`. Please include hex (`0x`) or binary (`0b`) prefixes in your answers. For all other bases, do not add the suffix or prefixes.

This page intentionally left with only one sentence.

Q1 *Potpourri* **(12 points)**

Q1.1 (1 point) True or False: The assembler translates code from a human-readable language (such as C) to an assembly language (such as RISC-V assembly).

◯ True ◯ False

Q1.2 (1 point) True or False: Jumps made to statically linked libraries are fully resolved in the linker.

◯ True ◯ False

Q1.3 (1 point) True or False: The OS allows for higher reliability; if a program has a bug, only that program will crash, instead of the entire system.

◯ True ◯ False

Q1.4 (1 point) True or False: For high-performance network devices, polling tends to be used when there's a low data rate, while interrupts tend to be used when there's a high data rate.

◯ True ◯ False

Q1.5 (1 point) True or False: A multithreaded program is considered correct as long as at least one order of the threads yields the correct answer, since we can force the scheduler to follow that thread order.

◯ True ◯ False

Q1.6 (1 point) True or False: It is cheaper to locate warehouse-scale computers in a cooler climate, in order to reduce total energy consumption.

◯ True ◯ False

We decide to set up 10 1 TiB disks together in a single RAID configuration. What is the effective amount of storage we have if we decide to use:

Q1.7 (0.5 points) RAID 0?

[ ] TiB

Q1.8 (0.5 points) RAID 5?

[ ] TiB

Q1.9 (1 point) We run the following code on two threads.

```
1  int y = 0;
2  int x = 10;
3  #pragma omp parallel
4  {
5    while(x>0)
6    {
7      y = y+1;
8      x = x-1;
9    }
10 }
```

What is the smallest possible value y can contain after this runs?

Q1.10 (1 point) Justin purchased his HP Pavilion 15t-cs300 laptop 1,000 days ago. During this time, it has broken twice, and had to be repaired. Each repair took 10 days to complete, during which time the laptop was unusable. What is the mean time to failure (MTTF) of Justin's laptop, in days?

days

Q1.11 (1 point) What is the availability of Justin's laptop?

Q1.12 (1 point) We've devised an error-correcting code which is able to fix 1 bit errors. If 0x61C is a valid codeword, which of the following can NOT be a valid codeword, regardless of the error-correcting scheme we have? Select all that apply.

- ☐ `0x71C`
- ☐ `0xC16`
- ☐ `0x51C`
- ☐ `0x16C`
- ☐ `0x70D`
- ☐ None of the above

Q1.13 (1 point) A program originally takes 1 second to run. We manage to parallelize 90% of our code to be 10 times faster, at the cost of 10 milliseconds of overhead. How many times faster is our new code?

Q2 *Rounding Errop* **(14 points)**

Note: we think Q2 and Q3 are harder questions. Feel free to skip them and come back later.

When working with floating point arithmetic, it is often the case that the exact result can't be stored in the floating point format. In this case, IEEE-754 defines the following rounding rule, which is commonly used: Compute the value precisely, then round to the nearest floating point number. In the event that the number is exactly halfway between two floating point numbers, round to the number with a least significant bit of 0.

For example, if we had a 10-bit minifloat with 5 exponent bits (and standard bias of -15) and 4 significand bits, the numbers 32 and 34 would be precisely representable (with no other representable numbers between them). When evaluating 32 + 0.5 = 32.5, we would round down to 32, while evaluating 34 - 0.5 = 33.5 would round up to 34. The expression 16 +17 = 33 would round to 32, because 32's binary representation as a 10-bit float is `0b0 10100 0000` (which has a 0 as its least significant bit), and 34's binary representation as a 10-bit float is `0b0 10100 0001`.

You may assume that for any two adjacent floating point numbers, one will have a LSB of 1 and the other will have a LSB of 0. Further, you may assume for this question that you will not need to round to infinity. Assume that any division in this question is float division (not integer division).

For the following questions, we will work with a 10-bit floating point representation that follows all conventions of IEEE-754 (including NaNs, denorms, etc.) but with 5 exponent bits (and standard bias of -15) and 4 significand bits.

What is the rounded values of the following (decimal) floating point numbers? You may express your answer either as an decimal value, or as an odd integer multiplied by a power of 2:

Q2.1 (3.5 points) 37

Q2.2 (3.5 points) 1/3 (whose binary representation is 0b0.0101 0101…)

We compute the following infinite sums under this floating point system, using left-association for addition (that is, a+b+c is evaluated in the order ((a+b)+c)), rounding after each addition. Eventually, this converges to some value, after which any further iterations don't change the sum. What is that value? You may express your answer either as an decimal value, or as an odd integer multiplied by a power of 2.

Q2.3 (3.5 points) $1 + (1/2) + (1/4) + \ldots$

Q2.4 (3.5 points) $2 + 2 + 2 + 2 + \ldots$

Q3 *Wait, why was this RISC-y, anyway?* **(14 points)**

Note: we think Q2 and Q3 are harder questions. Feel free to skip them and come back later.

Recall the definition of the function `verifypassword`:

The function `verifypassword` is defined as follows:

- Input: No register input; however, the function receives a string input from `stdin`.
- Output: `a0` returns 1 if the input from `stdin` is exactly `"secretpass"`, and 0 otherwise.

You have access to the following labels defined externally:

- Password: a pointer to a statically-stored string `"secretpass"`
- Get20chars: A function defined as follows:
  - Input: `a0` is a pointer to a buffer
  - Effect: Reads characters from `stdin`, and fills the buffer pointed to by `a0` with the read data, null-terminating the string. Your code may assume that the input is at most 19 characters, not including the null-terminator.
  - Output: None

You are a hacker, and you're currently trying to target the implementation of `verifypassword` presented on the midterm (copied below):

```
1  verifypassword:
2      addi sp, sp, -24 # Space for:
3      sw ra 20(sp)     #     ra
4      mv a0 sp         #     20-byte buffer
5      jal ra Get20chars
6      la t0 Password
7      mv t1 sp
8  Loop:
9      lb t2 0(t0)
10     lb t3 0(t1)
11     bne t2 t3 Fail
12     beq t2 x0 Pass
13     addi t0 t0 1
14     addi t1 t1 1
15     j Loop
16 Pass:
17     addi a0 x0 1
18     j End
19 Fail:
20     mv a0 x0
21 End:
22     lw ra 20(sp)
23     addi sp sp 24
24     jr ra
```

During the course of your testing, you discovered an interesting fact: the function `Get20Chars` doesn't actually work as intended! Instead of truncating at the 20th character, `Get20Chars` continues to write data until the first null terminator in its input. As before, verifypassword is located at `0x1000` and `Get20Chars` is located at `0x0F00`. Further, assume that the stack pointer is located at `0xBFFF F800` at the start of `verifypassword`, our page size is 4 KiB, and that we are currently working on a little-endian system.

Q3.1 (2 points) Our first step in exploiting this program is to find an input that changes the program flow. Submit a string that, if inputted in `stdin`, will cause `verifypassword` to return to the address `0xDEAD BEEF`. You may use the syntax `"A"*10` to denote a string consisting of 10 letter `"A"`s, and `"0xAB"` to signify the ASCII character corresponding to byte value `0xAB` (so "B" == `"0x42"`). For example, the answer `"A"*15+"0x42 0x42"+"C"` would correspond to the string `"AAAAAAAAAAAAAAABBC"`. (Hint: What gets changed if we write more than 20 characters?)

Q3.2 (2 points) Now that we can move the program counter to an arbitrary location, we would like to jump to some RISC-V code that we've written. In order to do this, we decide to jump to the start of the buffer on the stack. What is the maximum number of RISC-V standard instructions we can inject into this buffer?

Q3.3 (4 points) Regardless of your previous answer, assume that we can put up to 5 instructions in the buffer. Unfortunately, that's not really enough instructions to do much. Instead, we decide to inject code that lets us run longer programs, instead of only being limited to 5 instructions.

Complete the following 5-line code which does the above. You may use pseudoinstructions, as long as they resolve to exactly one instruction. Each blank is worth 1 point.

```
# Allocate a buffer of 256 bytes, which does not overlap with any data
# we already are using (such as the instructions injected in part 1)


1: addi _____________________________

# Set the argument of Get20Chars to the start of the allocated buffer


2: mv _________ sp

# Set t1 so the next instruction jumps to Get20Chars


3: lui _____________________________

4: jalr ra t1 -256 # Call Get20Chars

# Jump to the start of the buffer


5: ___________________________________
```

Q3.4 (2 points) Translate the instruction `jalr ra t1 -256` to its hexadecimal machine language encoding.

(Binary answers will not be awarded credit.)

0x

Q3.5 (2 points) Briefly explain in 10 words or fewer why we cannot use this instruction in our injected code. (Hint: What would `Get20Chars` do if you tried to send this instruction as input?)

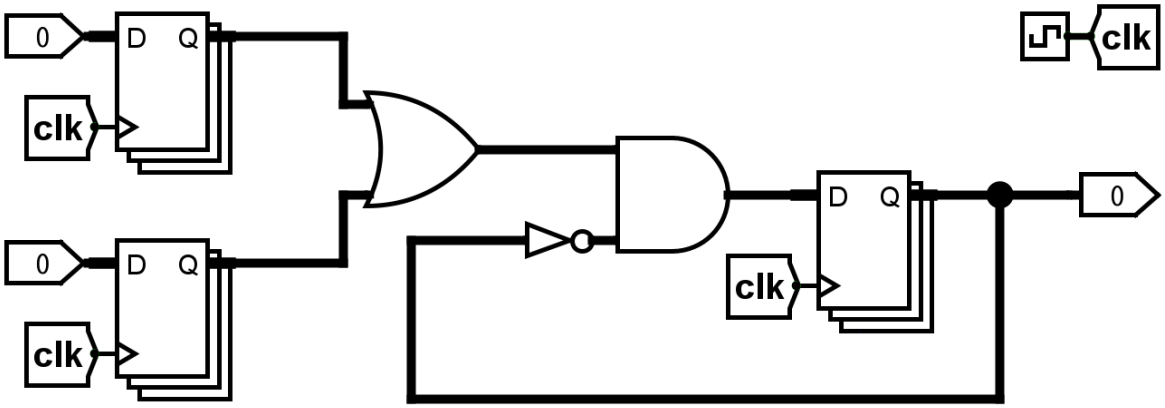
Q3.6 (2 points) Which of the following jump instructions can we use in our injected code? Don't worry about these lines not properly calling `Get20Chars`; we just want a valid RISC-V jump without running into the problem identified in part 5 (Hint: use the conversion you already did in part 4).

Note that +3840 == `0x0000 0F00`.

- ☐ `jalr ra t2 -256`
- ☐ `jalr ra t0 16`
- ☐ `ret`
- ☐ `jalr s0 x0 3840`
- ☐ `jalr x0 t1 -256`
- ☐ `jalr s0 t1 -256`
- ☐ None of the above

Q4 *Bit of a Delay* **(10 points)**

Consider the following circuit. Assume that AND and OR gates have a delay of 8 ps (picoseconds), NOT gates have a delay of 4 ps, and all registers have a setup time constraint of 6 ps and clock-to-Q delay of 3 ps. Assume all wires are ideal, i.e. they have zero delay.



Q4.1 (2 points) What is the largest combinational delay of all paths in this circuit, in picoseconds?

ps

Q4.2 (2 points) What is the smallest combinational delay of all paths in this circuit, in picoseconds?

ps

Q4.3 (2 points) What is the maximum possible hold time constraint for registers to function properly in this circuit, in picoseconds?

ps

Q4.4 (2 points) What is the minimum allowable clock period for this circuit to function properly, in picoseconds?

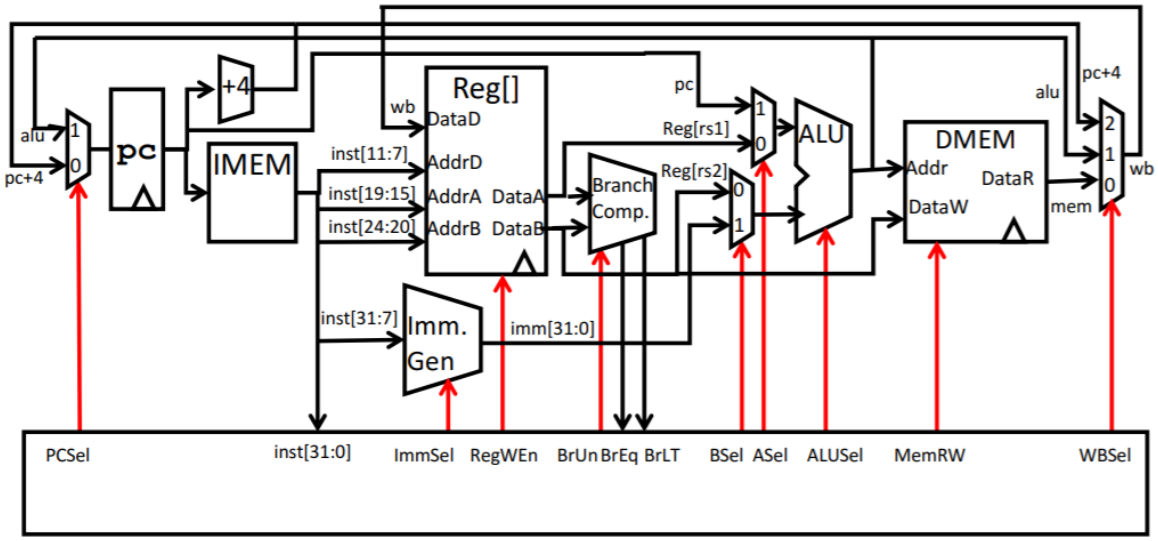
ps

Q4.5 (2 points) What is the maximum allowable clock frequency for this circuit to function properly, in gigahertz?

GHz

Q5 *Big Mac* **(10 points)**

Below is the standard RISC-V CPU used in Project 3.

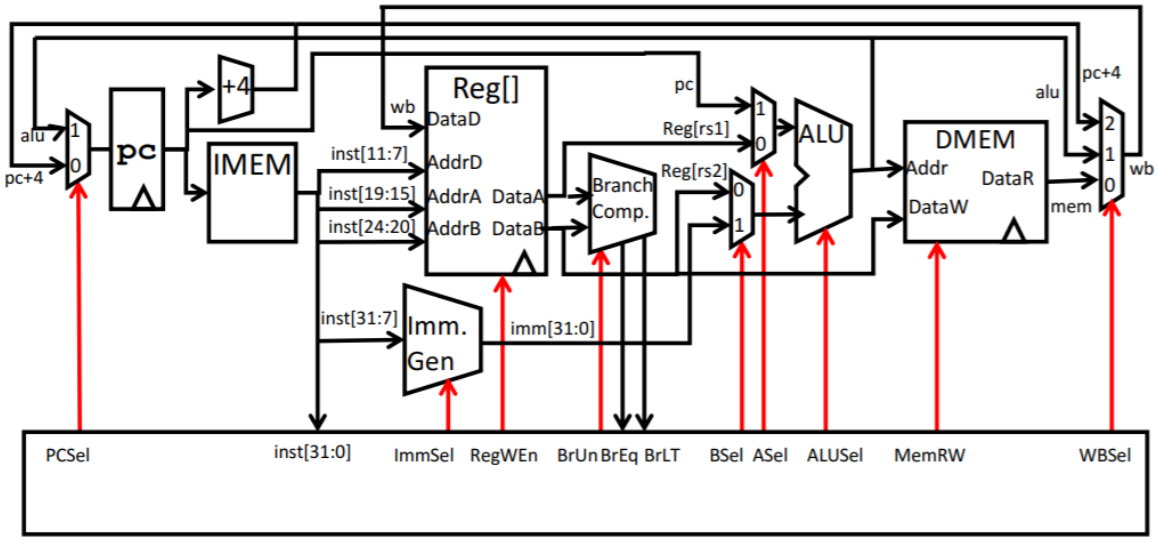


Q5.1 (4.5 points) For the instruction "lw", what are the control signals used? If a control signal doesn't matter, select "Don't Care".

Select one option per box. Each box is worth 0.5 points.

PCSel	◯ PC+4	◯ ALU	◯ Don't Care	
ImmSel	◯ I-type	◯ S-type	◯ B-type	◯ U-type
	◯ J-type	◯ Don't Care		
ASel	◯ PC	◯ `rs1`	◯ Don't Care	
BSel	◯ Imm	◯ `rs2`	◯ Don't Care	
RegWEn	◯ 1	◯ 0	◯ Don't Care	
BrUn	◯ 1	◯ 0	◯ Don't Care	
ALUSel	◯ `add`	◯ `sll`	◯ `slt`	◯ `xor`
	◯ `srl`	◯ `or`	◯ `and`	◯ `mul`
	◯ `mulh`	◯ `sub`	◯ `sra`	◯ `bsel`
	◯ Don't Care			
MemRW	◯ Read	◯ Write	◯ Don't Care	
WBSel	◯ ALU	◯ MEM	◯ PC+4	◯ Don't Care

The CPU is reproduced here for convenience.



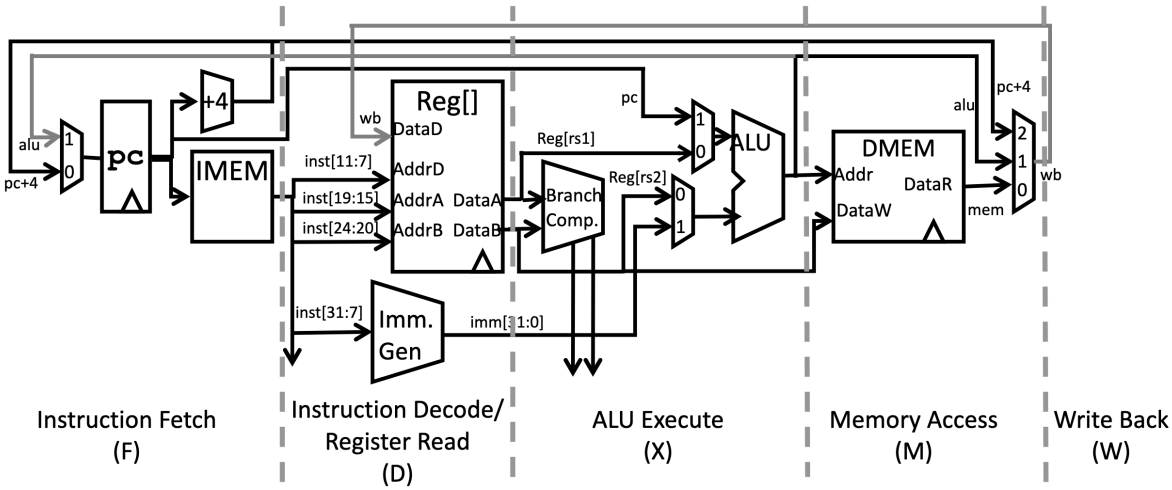
Q5.2 (3 points) We want to add a new instruction `mac` (multiply and accumulate) to our CPU:

```
mac rd, rs1, rs2
```

Set `rd` to `rd + (rs1 * rs2)`.

What changes would we need to make to our datapath in order for us to implement this instruction (with as few changes as possible)? Select all that apply.

- ☐ Add a new instruction format
- ☐ Add a new immediate type for the ImmGen
- ☐ Add a new rs3 input to RegFile
- ☐ Add a new output to RegFile for a third register value
- ☐ Add a new input to AMux and update the relevant selectors/control logic
- ☐ Add a new input to BMux and update the relevant selectors/control logic
- ☐ Add a new ALU input for a third register input
- ☐ Add a new ALU operation and update the relevant selectors/control logic
- ☐ Add a new input to WBMux and update the relevant selectors/control logic
- ☐ None of the above



Q5.3 (2.5 points) Write a sequence of instructions that causes a hazard in a completely unoptimized 5-stage pipeline (no forwarding, no branch prediction, no synchronous read/writes, etc), but which would not cause a hazard if all mac instructions were changed to mul instructions. If no such sequence exists, write "Not Possible."

Q6 *Boolean Code Golf* **(10 points)**

For this question, consider each truth table of inputs and expected outputs. Write Boolean expressions that, given the inputs W, Y, and Z, evaluate to the given output.

Your answer should consist of the following characters:

W, Y, Z	The inputs
∼	NOT
|	OR
&	AND
∧	XOR
()	Parentheses
1, 0	Constants

Each question specifies a par score, which is the target number of Boolean operations to use. For full credit, your solution must use at most the par score number of operations (∼ , |, &, and ∧ each count as one operation). Partial credit will be awarded for fully correct solutions that slightly exceed the par score. The par score is not necessarily the minimum number of operations required.

Operator precedence will follow standard C operator precedence. We will NOT offer partial credit for assuming incorrect operator precedence, so use parentheses when uncertain.

Q6.1 (2.5 points) Par 1 (1 Boolean operation)

W	Y	Z	Out
0	0	0	1
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	0

Q6.2 (2.5 points) Par 2

W	Y	Z	Out
0	0	0	1
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	0

Q6.3 (2.5 points) Par 3

An X is used to signify that either 1 or 0 can be outputted for the corresponding input.

W	Y	Z	Out
0	0	0	X
0	0	1	X
0	1	0	0
0	1	1	1
1	0	0	X
1	0	1	X
1	1	0	1
1	1	1	0

Q6.4 (2.5 points) Par 4

W	Y	Z	Out
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

Q7 *<insert obligatory money pun here>* **(10 points)**

A program is run on a byte-addressed system with a single-level cache, where memory addresses are 10 bits long. After a while, the entire cache has the following state:

Index	Tag 1	Valid 1	Tag 2	Valid 2
0b00	0b1011	1	0b1101	1
0b01	0b0011	1	0b0010	1
0b10	0b1110	1	0b0111	0
0b11	0b1111	0	0b0001	0

Q7.1 (1 point) What is the associativity of the cache?

Q7.2 (1.5 points) What is the T:I:O breakdown of memory addresses?

T	I	O

Q7.3 (1.5 point) How many bytes of data can this cache contain?

Q7.4 (6 points) For each of the following memory accesses, determine if each access would be a hit or miss based on the cache state shown above, and if it's a miss, classify the possible miss type(s). If multiple miss types are possible depending on prior memory accesses, select all possible miss types.

Note: For this question, each memory access should be considered in isolation. In particular, do not update the cache state after each memory access.

Address				
0b 0011011011	☐ Hit	☐ Compulsory miss	☐ Capacity miss	☐ Conflict miss
0b 0011001101	☐ Hit	☐ Compulsory miss	☐ Capacity miss	☐ Conflict miss
0b 0110100010	☐ Hit	☐ Compulsory miss	☐ Capacity miss	☐ Conflict miss
0b 0010111100	☐ Hit	☐ Compulsory miss	☐ Capacity miss	☐ Conflict miss
0b 1110100010	☐ Hit	☐ Compulsory miss	☐ Capacity miss	☐ Conflict miss
0b 1111111111	☐ Hit	☐ Compulsory miss	☐ Capacity miss	☐ Conflict miss

Q8 *Deja-VM* **(10 points)**

One useful aspect of virtual memory is that it allows two distinct programs to try and access the same virtual memory addresses, and still get different data locations. Consider a system with 64 MiB of physical memory, which runs programs that have 4 GiB of virtual memory. Our page size is 4 KiB.

Q8.1 (1 point) How many virtual pages do we have? Express your answer as a power of 2.

Q8.2 (1 point) How many bits long is a physical address?

Q8.3 (1 point) How many bits long is a virtual page number?

Regardless of your answer to Q8.2, assume that physical addresses are 20 bits long. We run two programs (with no shared memory), which access the following virtual memory addresses in order. For each memory access, determine the physical address that gets accessed, writing your answer in hexadecimal.

Assume that no physical pages are in use prior to the first memory access, and that physical pages get assigned in order of physical page number (so page 0 is assigned first, then page 1, and so on).

Q8.4 (1 point) Program 1: `0xABCDEFAB`

Q8.5 (1 point) Program 1: `0x12345678`

Q8.6 (1 point) Program 1: `0xABCDD312`

Q8.7 (1 point) Program 2: `0xABCDEFAB`

Q8.8 (1 point) Program 2: `0x12345664`

Q8.9 (1 point) Program 1: `0x12345664`

Q8.10 (1 point) Program 2: `0xABCDEFAB`

Q9 *Testception 2, 3, 4, AND 5! Now simulcasting!* **(10 points)**

Fred's Factorization Factory has unveiled their latest product: an algorithm that factorizes an array of numbers provided. You want to test their factoring algorithm, so you decide to write the following function:

`int testFactor(uint32_t n, uint64_t *a, uint64_t *b, uint64_t *c);`

- n: The length of each list of integers. For simplicity, you may assume that n is a multiple of 4.
- a, b, c: Pointers to arrays of 64-bit integers.

`testFactor` returns 1 if, for all `i` from 0 to n-1, `a[i]*b[i] == c[i]`. Otherwise, it returns 0.

You have access to the following SIMD instructions:

- `_mm256 vectorLoad(void* ptr)`: Loads four `uint64_t` from `ptr` into a SIMD vector
- `void vectorStore(void* ptr, _mm256 mm)`: Stores the four `uint64_t` in `mm` at `ptr`
- `_mm256 vectorMul(_mm256 a, _mm256 b)`: Multiplies the values in `a` and `b`, and returns the result
- `_mm256 vectorSet0()`: Returns a vector containing only 0s.
- `_mm256 vectorOr(_mm256 a, _mm256 b)`: Computes the bitwise OR of the two vectors, and returns the result.
- `_mm256 vectorXor(_mm256 a, _mm256 b)`: Computes the bitwise XOR of the two vectors, and returns the result.

```
int testFactor(uint32_t n, uint64_t *a, uint64_t *b, uint64 *c)
{

    uint64_t output[4];


    _mm256 total = ____________________________________________;

    for(int i = 0; i < ___________________; i+= ___________________)
    {
        _mm256 adata = vectorLoad(a+i);
        _mm256 bdata = vectorLoad(b+i);
        _mm256 cdata = vectorLoad(c+i);

        _mm256 prod = ____________________________________________;

        _mm256 isequal = ____________________________________________;

        ____________________________________________________________;
    }
    vectorStore(output, total);

    return ____________________________________________________ ? 1 : 0;
}
```