CS 61C
Summer 2022

Caroline, Justin, Peyrin
Final

Print your name: ______________________________, ______________________________
(last) (first)

Print your student ID: ____

You have 110 minutes. There are 8 questions of varying credit (100 points total).

Question:	1	2	3	4	5	6	7	8	Total
Points:	7	16	20	6	15	7	16	13	100

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 unless otherwise specified. For all other bases, do not add any prefixes or suffixes.

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.

Sign your name: ____

Questions begin on the next page.

Q1 *Potpourri* (7 points)

Q1.1 (4 points) Select the 2 Boolean expressions from below that are equivalent to each other:

☐ $(A+!B)(A+!C)$

☐ $!B!C + A$

☐ $A+!(B + C) + (!B+!C)(!B + C)$

☐ $(A + B)(A + C)(!BC)$

Convert the following 2-byte hex numbers to decimal using signed 2's complement.

Q1.2 (1 point) `0xFF40`

Q1.3 (1 point) `0x009C`

Q1.4 (1 point) Consider a 12-bit biased number representation scheme with a bias of -2047. Which of the following is not a representable number in this scheme?

○ 0

○ 2048

○ -2048

○ -1

Q2 *C: Filed Away* **(16 points)**

You are bored over summer break so you decide to write up a file system in C!

The struct `file_item` represents a file or a folder. The `data` union holds either the contents of the file (a string), or an array of pointers to children `file_item`s.

In this question, assume that pointers are 4 bytes long.

```
typedef struct file_item {
    char *name;
    bool is_folder;
    file_item_data data;
} file_item;

typedef union file_item_data {
   char contents[X];
   struct file_item* children[16];
} file_item_data;

// Copies all characters from src to dest including the NULL terminator
char *strcpy(char *dest, const char *src);
```

We set `X` to be the largest possible value that doesn't increase the union size. What is the `strlen` of the longest string we can store in a single file?

Fill in the code on the next page to create files and folders. Your code must still work even if the input strings are freed later on. Assume that the input strings will fit inside of a `file_item_data` union.

You may use fewer lines than provided, but you may not add more lines.

```
/* Creates a file with the given name and contents,
   and returns a pointer to that file. */
file_item* create_file(char* contents, const char *name) {

    ____________________________________________________________

    ____________________________________________________________

    ____________________________________________________________

    ____________________________________________________________

    ____________________________________________________________

    ____________________________________________________________

    ____________________________________________________________

    ____________________________________________________________

    ____________________________________________________________

    ____________________________________________________________
}
/* Creates a folder with the given name and no children,
   and returns a pointer to that folder. */
file_item* create_folder(const char *name) {

    ____________________________________________________________

    ____________________________________________________________

    ____________________________________________________________

    ____________________________________________________________

    ____________________________________________________________

    ____________________________________________________________

    ____________________________________________________________

    ____________________________________________________________

    ____________________________________________________________

    ____________________________________________________________
}
```

(Question 2 continued…)

Q3 *Error-Introducing Code* **(20 points)**

In machine learning, some data scientists add random noise to a training dataset in order to improve their models. Here, we will take a dataset and transform it into usable data in RISC-V!

The function `jitter` is defined as follows:

- Inputs:
  - `a0` holds a pointer to an integer array.
  - `a1` holds a buffer large enough for `n` integers (which does not overlap with the array in `a0`).
  - `a2` holds `n`, the length of the arrays.
- Output:
  - `a0` holds a pointer to the buffer originally in `a1`. The buffer is filled with the result of calling `noisen` on each element in the `a0` array.

The function `noisen` is defined as follows:

- Input: `a0` holds an integer.
- Output: `a0` returns the integer with noise added.

Eric has provided the correct implementation of `jitter` below, following calling convention:

```
jitter:
  # BEGIN PROLOGUE
  addi sp sp <BLANK 1>
  # (multiple lines omitted)
  # END PROLOGUE
  mv s0 a0
  mv s1 a1   # Hold beginning of output arr
  mv s2 a1
  mv s3 a2   # Hold counter
loop:
  beq s3 x0 end
  lw a0 0(s0)
  jal ra noisen
  sw a0 0(s1)
  addi s0 s0 4
  addi s1 s1 4
  addi s3 s3 -1
  j loop
end:
  mv a0 s2
  # BEGIN EPILOGUE
  # (multiple lines omitted)
  addi sp sp <BLANK 2>
  # END EPILOGUE
  ret
```

Q3.1 (1 point) To follow calling convention, what numbers should go in the blanks?

`<BLANK 1>` [ ]

`<BLANK 2>` [ ]

Q3.2 (1 point) List all registers that Eric needs to save on the stack in order to follow calling convention.

[ ]

Q3.3 (6 points) Write a sequence of at most two instructions or pseudoinstructions that are equivalent to the `j loop` instruction.

You must use a `jalr` instruction or `jalr` pseudoinstruction in at least one of the blanks. You may not use a `jal` instruction, branch instruction, or `jal` pseudoinstruction in any of the blanks.

1 ______________________________

2 ______________________________

Now, Eric wants to implement `noisen` to add some random offset to an integer `a0`. Unfortunately, Eric only has access to the `randomBool` function, which takes in no inputs and randomly returns either 1 or 0 in `a0`.

If Eric implemented `noisen` to return `a0+randomBool()`, the integer would always get shifted in the positive direction. Instead, Eric suggests implementing `noisen` so that `noisen` alternates between returning `a0+randomBool()` and returning `a0-randomBool()`.

Q3.4 (12 points) Fill in the blanks to complete Eric's suggested implementation of `noisen`.

Assume that you can read from and write to any memory addresses, in any section of memory.

```
noisen:
  addi sp sp -8 # Prologue
  sw ra 0(sp)
  sw s0 4(sp)
  mv s0 a0

  jal ra randomBool
  add s0 s0 a0

  ______________________________  # one RISC-V instruction

  xori t0 t0 ___________________  # some immediate

  ______________________________  # one RISC-V instruction

  mv a0 s0 # Epilogue
  lw ra 0(sp)
  lw s0 4(sp)
  addi sp sp 8
  ret
```

Q4 *Floating Loading* **(6 points)**

RISC-V has a floating-point extension to support IEEE-754 single-precision floats (1 sign bit, 8 exponent bits, 23 mantissa bits). The extension adds the instructions in the following table, as well as 32 floating-point registers (`f0` to `f31`) that each hold 32 bits.

Instruction	Name	Description
`fadd.s rd rs1 rs2`	Float ADD	`rd = rs1 + rs2`
`fsub.s rd rs1 rs2`	Float SUBtract	`rd = rs1 - rs2`
`fmul.s rd rs1 rs2`	Float MULtiply	`rd = rs1 * rs2`
`fdiv.s rd rs1 rs2`	Float DIVide	`rd = rs1 / rs2`
`fmv.w.x rd rs1`	MoVe from Integer	`rd = ((float) rs1)`

Note that the 4 floating-point arithmetic instructions can only operate on floating-point registers, and floating-point registers cannot be used with base instructions. For example, `fadd.s f0 t1 t2` and `addi f0 t1 5` are not valid instructions.

`fmv.w.x` takes the bitstring in an integer register `rs1`, and transfers the bitstring over to a floating-point register `rd`.

For example, the value 4 is represented in single-precision floating point numbers as `0x4080 0000`. If `t0` contained the hexadecimal value `0x4080 0000` and we ran `fmv.w.x f0 t0`, then `f0` would be set to the floating point value 4.0.

Q4.1 (2 points) Translate the value 3 into its single-precision floating point representation, in hexadecimal.

`0x`

Q4.2 (4 points) Write instructions to put the float (as close as possible to) `1.33333...` into a register `f1`. You may not use any floating-point instructions outside the five listed above.
Only one RISC-V instruction or pseudoinstruction is allowed per line. You may use fewer lines than provided, but you may not add more lines.

1 ____________________

2 ____________________

3 ____________________

4 ____________________

5 ____________________

Q5 *Error-Correcting Code* **(15 points)**

Recall that a Hamming Error Correcting Code can be used to fix single-bit errors. In situations where data is error-prone (such as on satellites, or in L1 caches), it is often useful to store blocks of data in Hamming codes, and internally fix errors during memory retrieval.

For this question, we will consider a (7,4) Hamming Code with even parity, which uses 7 total bits to store 4 data bits. Recall the following bit pattern for the (7,4) Hamming Code:

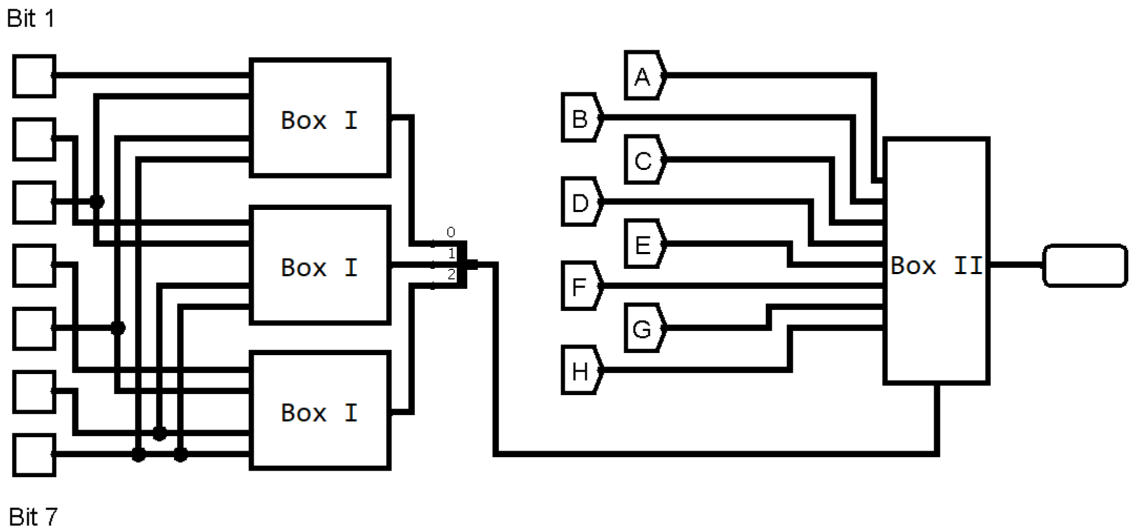
Bit	1	2	3	4	5	6	7
Transmitted Bit	p_1	p_2	d_1	p_4	d_2	d_3	d_4
p_1	✓		✓		✓		✓
p_2		✓	✓			✓	✓
p_4				✓	✓	✓	✓

We adopt the convention that bit 1 is the most significant bit of the data, and bit 7 is the least significant bit.

Q5.1 (3 points) In order to store a full byte, we concatenate two (7,4) Hamming codes, with the first 7 bits storing the most significant nibble of data, and the last 7 bits storing the least significant nibble. After storing a byte, we retrieve the following raw data (spacing has been added for readability):
`0b 1001110 1000011`
After error correction, what byte gets returned? Express your answer in hexadecimal.

0x

We construct a circuit that, given a (7,4) Hamming Code, outputs a corrected nibble of data. What components should be placed in the labeled boxes? Assume that multiple input gates are made by chaining two-input gates.



Q5.2 (2 points) What goes in Box I?

- ◯ AND gate
- ◯ OR gate
- ◯ XOR gate
- ◯ Adder gate
- ◯ Multiplier gate
- ◯ Multiplexer (with 0 input on top)
- ◯ Demultiplexer (with 0 input on top)
- ◯ Priority Encoder (with 0 input on top)

Q5.3 (2 points) What goes in Box II?

- ◯ AND gate
- ◯ OR gate
- ◯ XOR gate
- ◯ Adder gate
- ◯ Multiplier gate
- ◯ Multiplexer (with 0 input on top)
- ◯ Demultiplexer (with 0 input on top)
- ◯ Priority Encoder (with 0 input on top)

Q5.4 (2 points) Which four labels among A-H should have the same value?

- ☐ A
- ☐ B
- ☐ C
- ☐ D
- ☐ E
- ☐ F
- ☐ G
- ☐ H

Q5.5 (4 points) Assume that the component in Box I has a delay of 3 ns, the component in Box II has a delay of 5 ns, and that computing the values of labels A-H take 1 ns. If inputs arrive at time 0, what is the earliest and latest time the output can change in response?

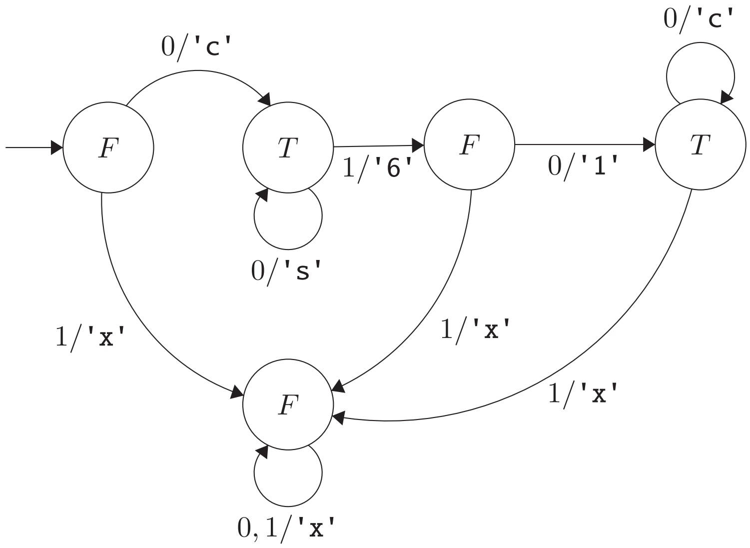
Earliest: ns

Latest: ns

Q5.6 (2 points) Is it more important to add this component around the IMEM or the DMEM? Explain your reasoning in 20 words or fewer.

Q6 ***00100*** **(7 points)**

Consider the following FSM:



On each transition, we receive a 1-bit input, and output a character.

For the following inputs, what string would the FSM output?

Q6.1 (1 point) 000100

Q6.2 (1 point) 0110

Q6.3 (1 point) 010000

Q6.4 (4 points) If the FSM ends in a state labeled T after processing the entire input, then the input is accepted. Otherwise, the input is rejected.
Write a rule for determining whether an input will be accepted by this FSM.

Q7 *Virtual Memory* **(16 points)**

Assume the virtual memory address `0xABCD123` maps to the physical address `0x123123`.

Q7.1 (1.5 points) What is the maximum possible page size? Include units in your answer.

Q7.2 (1.5 points) What is the minimum possible size of virtual memory? Include units in your answer.

Q7.3 (2.5 points) This subpart is independent of the previous subparts.

We have 1 GiB of virtual memory, 24-bit physical addresses, a 4 KiB page size, and a single-level page table.

If a page table entry has 4 bits of metadata, how many bits is a page table entry, with no padding?

This subpart is independent of the previous subparts.
We have 4 GiB of virtual memory, 16 MiB of physical memory, a 4 KiB page size, and a single-level page table. We also have a 2-entry, fully-associative TLB with LRU replacement policy.
Assume that the TLB starts empty, and that physical pages are assigned in order starting with page 0 (e.g. page 0, page 1, etc.). Also assume that we are working with a single-core system that will context-switch between processes.

Q7.4 (10.5 points) Each row in the table represents a memory access. For each row, fill out the corresponding physical address, and whether the access causes a TLB hit, a TLB miss but page table hit, or a page fault.

Virtual Address	Process ID	Physical Address	Access Type
0xDEADBEEF	1	0x000EEF	○ TLB Hit ○ TLB miss, page table hit ○ Page Fault
0xDEADBEEF	2		○ TLB Hit ○ TLB miss, page table hit ○ Page Fault
0x0000061C	2		○ TLB Hit ○ TLB miss, page table hit ○ Page Fault
0xDEADB61C	2		○ TLB Hit ○ TLB miss, page table hit ○ Page Fault
0xDEADB61B	1		○ TLB Hit ○ TLB miss, page table hit ○ Page Fault
0xDEADB61C	1		○ TLB Hit ○ TLB miss, page table hit ○ Page Fault
0x0000061A	2		○ TLB Hit ○ TLB miss, page table hit ○ Page Fault
0x0000061A	1		○ TLB Hit ○ TLB miss, page table hit ○ Page Fault

Q8 *DLP Meets TLP* **(13 points)**

Grace wants to write a function called `vector_mul_positive`, defined as follows:

Inputs:

- `a`, a pointer to an integer array
- `b`, a pointer to an integer array
- `N`, the length of each array. You may assume that `N` is a multiple of 4.

Outputs:

- Compute the element-wise products, and output the sum of only the positive products.

Example input: `a=[-1, 2, 3, 4]`, `b=[5, 6, -7, 8]`, and `N = 4`.

Example output: $2 \times 6 + 4 \times 8 = 44$. Note we skip -1×5 and 3×-7 because these products are negative.

For this question, you may use these SIMD functions (`__m128i` represents packed signed 32-bit integers):

`__m128i vmul(__m128i a, __m128i b)`: Return the result of multiply values in `a` and `b`

`__m128i vset (int32_t x)`: Set the 4 signed 32–bit integers to `x`

`__m128i vload (__m128i* a)`: Return 128-bits of integer data stored at pointer `a`

`void vstore (__m128i* b, __m128i a)`: Store 128-bits of integer data from `a` into `b`

`__m128i vcmpgt (__m128i a, __m128i b)`: Compare `a` and `b` for greater-than and return result

`__m128i vadd (__m128i a, __m128i b)`: Return the addition if `a` and `b`

`__m128i vsub (__m128i a, __m128i b)`: Return the subtraction of `b` from `a`

`__m128i vxor (__m128i a, __m128i b)`: Return the bitwise XOR of `a` and `b`

`__m128i vor (__m128i a, __m128i b)`: Return the bitwise OR of `a` and `b`

`__m128i vand (__m128i a, __m128i b)`: Return the bitwise AND of `a` and `b`

Implement a parallelized version of `vector_mul_positive`. You may use fewer lines than provided, but you may not add more lines.

```
int32_t vector_mul_positive(int32_t *a, int32_t *b, int32_t N) {
    int32_t result[4];

    __m128i sum_v = ______________________________________________________;

    __m128i cond_v = _____________________________________________________;

    #pragma _______________________________________________________________

    for (______________________________________________________________) {

        __m128i curr_v1 = vload(________________________________________);

        __m128i curr_v2 = vload(________________________________________);

        __m128i mul = ___________________________________________________;

        __m128i tmp = ___________________________________________________;

        _________________________________________________________________;

        #pragma _________________________________________________________

        _________________________________________________________________;
    }

    _____________________________________________________________________;

    _____________________________________________________________________;
}
```

Nothing on this page will be graded.

Is there anything you want us to know?