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:

- ■ $(A+!B)(A+!C)$
- ■ $!B!C + A$
- □ $A+!(B + C) + (!B+!C)(!B + C)$
- □ $(A + B)(A + C)(!BC)$

Solution: First two answers.
Grading: All-or-nothing.
Explanation: TODO

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

Q1.2 (1 point) `0xFF40`

Solution: -192
Grading: All-or-nothing. No credit for equivalent expressions.
Explanation: `0xFF40` = `0b1111 1111 0100 0000` We know that the resulting decimal representation will be negative, since the MSBit is a 1. Thus, we flip the bits and add 1 to obtain the positive version of the decimal representation, obtaining `0b0000 0000 1011 1111`$+1 =$ `0b0000 0000 1100 0000` $= 128 + 64 = 192$. Thus our answer is -192.

Q1.3 (1 point) `0x009C`

Solution: 156
Grading: All-or-nothing. No credit for equivalent expressions.
Explanation: `0x009C` = `0b0000 0000 1001 1100`. This is a positive 2's complement number since the MSBit is a 0. Thus, we can interpret this value the same way as unsigned, obtaining `0b0000 0000 1001 1100` $= 128 + 16 + 8 + 4 = 156$

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

Solution: -2048

Grading: All-or-nothing.

Explanation: A 12-bit *unsigned* number representation has a representable range of $[0, 4096)$. A biased 12-bit representation with bias of -2047 has a representable range of $[0 + (-2047), 4096 + (-2047)) = [-2047, 2049)$. Thus, -2048 is not representable.

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.

```
 1 typedef struct file_item {
 2     char *name;
 3     bool is_folder;
 4     file_item_data data;
 5 } file_item;
 6
 7 typedef union file_item_data {
 8    char contents[X];
 9    struct file_item* children[16];
10 } file_item_data;
11
12 // Copies all characters from src to dest including the NULL terminator
13 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?

Solution: 63

Grading: +2 for correct answer; +1 for 64 (forgot to consider null terminator)

Explanation: The size of a union is determined by the largest element it can represent (disregarding alignment rules, which we don't consider here). In a 32-bit system, pointers are 32 bits (4 bytes). `children` is an array of 16 points, which takes up $16 \times 4 = 64$ bytes. Thus, we have to ensure `char contents[X]` does not exceed 64 bytes. Since we know by default 1 byte is defined to be the size of a `char`, `X` $= 64$ fulfills our goal. `strlen(contents)` thus would return 63 since it does not count the null terminator in string length.

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.

```
 1 /* Creates a file with the given name and contents,
 2    and returns a pointer to that file. */
 3 file_item* create_file(char* contents, const char *name) {
 4     file_item* new_file = (file_item*) calloc(1, sizeof(file_item));
 5     new_file->name = name;
 6     strcpy(new_file->data.contents, contents);
 7     return new_file;
 8 }
 9 /* Creates a folder with the given name and no children,
10    and returns a pointer to that folder. */
11 file_item* create_folder(const char *name) {
12     file_item* new_dir = (file_item*) calloc(1, sizeof(file_item));
13     new_dir->name = name;
14     new_dir->is_folder = true;
15     return new_dir;
16 }
```

Solution: See above.

Grading: see Gradescope assignment for full partial credit; full credit was given for different implementations that are fully correct with no memory leaks and did not add any additional lines.

`create_file` Explanation: For `create_file`, we can directly set the `name` field because we're just setting the field's value to the pointer value passed in from the `const char *name` argument. `strcpy(new_file->name, name)` is not valid without dynamically allocating space for `name` because it's relying on additional memory to copy `name` into, as opposed to assigning a pointer (thus, it's not assigning `name` correctly, which is one of the partial credits). Points were also taken off if the answer allocated space as follows: `char* name_mem = (char*) calloc(strlen(name))` because this does not allow the one extra byte for the null terminator to be copied over. `char* name_mem = (char*) calloc(sizeof(name))` is also incorrect since `sizeof` will return 4, since it's taking the size of the pointer.

We cannot directly assign `contents` with the assignment operator (=) because we need to ensure the code still works even if input strings are freed later on (per the instructions). (It also doesn't work because you're assigning the static type `char contents[X]` to a `char*`; however the reverse is fine.) `name` does not share this problem because it's typed as `const char` which means it can't be freed. Thus, we can either explicitly dynamically allocate space of `strlen(contents) + 1` and then `strcpy(data.contents, contents)` or directly copy over the data. To dynamically allocate space, we can allocate a new variable, `file_item_data* new_data = (file_item_data*) malloc(sizeof(file_item_data*))`. For both, we can then `strcpy the contents`. For the second option, for the solution to be fully correct, something like `new_file->data = *new_data; free(new_data)` must be used in order to correctly copy the struct into the original `data` field's location in memory and free the temporary struct to not have memory leaks before returning the file.

`create_folder` Explanation: the two major fields to set here are the `name` and `is_folder` fields, respectively by doing `new_dir->name = name` and `new_dir->is_folder = true`.

To set `new_dir->data.children` to `NULL` correctly, we can either `calloc` the entire struct, since all 0 bits is equivalent logically to `NULL`. Alternatively, we can set the fields manually if we used `malloc` to allocate the new folder as follows:

```
15      ...
16      for (int i = 0; i < 16; i++) {
17           new_dir->data.childreni = NULL;
18      }
19  ...
```

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:

```
 1 jitter:
 2   # BEGIN PROLOGUE
 3   addi sp sp <BLANK 1>
 4   # (multiple lines omitted)
 5   # END PROLOGUE
 6   mv s0 a0
 7   mv s1 a1   # Hold beginning of output arr
 8   mv s2 a1
 9   mv s3 a2   # Hold counter
10 loop:
11   beq s3 x0 end
12   lw a0 0(s0)
13   jal ra noisen
14   sw a0 0(s1)
15   addi s0 s0 4
16   addi s1 s1 4
17   addi s3 s3 -1
18   j loop
19 end:
20   mv a0 s2
21   # BEGIN EPILOGUE
22   # (multiple lines omitted)
23   addi sp sp <BLANK 2>
24   # END EPILOGUE
25   ret
```

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

`<BLANK 1>`

`<BLANK 2>`

Solution:

Blank 1: -20

Blank 2: 20

Grading: All-or-nothing per blank.

Explanation: We need to make space for 5 registers on the stack, and each register is 4 bytes long. (See next subpart for the 5 registers.)

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

Solution: `s0`, `s1`, `s2`, `s3`, `ra`.

Grading: 0.2 points for each correct answer. -0.2 points for each additional incorrect register.

Explanation: The function modifies `s0`, `s1`, `s2`, `s3`. According to calling convention, the function must leave saved registers unchanged, so we have to save their values at the start of the function, and restore their values at the end of the function.

Also, line 13 of the function (`jal ra noisen`) modifies `ra`. According to calling convention, `ra` must stay unchanged throughout the function (so that we know what address/instruction to return to after the end of the function), so we also need to save its value at the start of the function, and restore its value at the end of the function.

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 _____
```

Solution:

```
1 la t0, loop
2 jalr x0, t0, 0 # jr t0
```

Grading: +3 for correct `la` usage, +3 for correct `jalr` with offset 0 (given correct usage of `la`)

Explanation: `j loop` is just `jal x0, loop`. `jal` instructions are PC-relative so to translate it to an absolute address, you have to load the address in code for the `loop` label using the `la` (load address) instruction into any register. Then, you have to use `jalr` with no return address (since `j` does not save a return address) on the address loaded into a register with no offset to execute starting from `loop`.

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.

```
 1 noisen:
 2   addi sp sp -8 # Prologue
 3   sw ra 0(sp)
 4   sw s0 4(sp)
 5   mv s0 a0
 6   jal ra randomBool
 7   add s0 s0 a0
 8   lb t0 3(ra)  # one RISC-V instruction
 9   xori t0 t0 64 # some immediate
10   sb t0 3(ra)  # one RISC-V instruction
11   mv a0 s0 # Epilogue
12   lw ra 0(sp)
13   lw s0 4(sp)
14   addi sp sp 8
15   ret
```

Solution: See above.

Grading: +6 points was given to everyone due to logistical and fire alarm issues during the exam. +2 was given for an attempt to load from/store to the `add` instruction. +1 was given to loading/storing relative to `ra`. +1 was given to accessing the correct instruction in IMEM. +0.5 was given to accessing the correct part of the instruction (`lb`/`sb` with offset 3). +1.5 was given to flipping the correct bit to switch from `add` to `sub`.

Explanation: This was an extremely difficult question, so don't worry if you weren't sure how to approach this question. This question relies on the concept of modifying unprotected instruction memory, to create self-modifying code. The first thing to notice is that line 7 is currently set to add the random bool to the data; to alternate to a subtraction, we would need to change this instruction to a `sub` instruction. The second thing to notice is that `add` and `sub` in RV32 is one bit off from each other, given usage of the same registers. This occurs in the `funct7` with `add` using `000 0000` and `sub` using `010 0000` (this was by design, as add and subtract are so close in function). Thus, our general approach can be described as follows:

- Load the `add`/`sub` instruction into a temporary register
- Flip the correct bit in the `func7` to change the instruction into a `sub`/`add` instruction
- Store the instruction back in its original spot

Two major challenges arise from this approach: Finding a pointer to that instruction, and flipping the correct bit.

In order to load the `add/sub` instruction, we need to find a pointer to the code; further, we do not have sufficient lines to run a `la` instruction, so this address must already be stored in a current register. One insight here is that the `ra` register is currently pointing to a location in the code; more precisely, we called `jal ra randomBool` on line 6. This guarantees that `ra` was set to point to line 7, which is incidentally exactly the line we need to load. Note that this could also work even if there were extra lines between the function call and the `add/sub`, using a nontrivial offset.

The second concern is to determine how to flip the correct bit. Note that `add/sub` differ in the bit at the 30th position of our instruction. If we attempt to load the entire instruction and flip the bit with a single `xori`, we would require an immediate of 0x4000 0000, which is greater than our maximum immediate of 2047. Thus, we instead need to load part of the instruction, using a `lh` or `lb`. If we use `lh` to load the top half of the instruction, we would have to modify the 14th bit position, yielding an immediate of 0x4000 = 16384; thus, we must use a `lb` instruction to load the top byte of the instruction. This would allow us to modify the 6th bit position, which corresponds to an immediate of 0x40 = 64.

Thus, the final solution is as follows:

- Load the top byte of the `add/sub` instruction into a temporary register, using the `ra` as a code pointer: lb t0 3(ra)
- Flip the correct bit in the `func7` to change the instruction into a `sub/add` instruction: xori t0 t0 64
- Store the instruction back in its original spot: sb t0 3(ra)

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.

Solution: `0x4040 0000`
First write 3 in binary scientific notation: $3 = \texttt{0b}1.1 = \texttt{0b}1.1 \times 2^1$.
Sign bit is `0b0` since the number is positive.
Exponent is 1. To encode in biased notation, we subtract the bias (for an 8-bit exponent, use the default bias of $-(2^{8-1}-1) = -127$) to get $1-(-127) = 128$. Then encode in 8-bit unsigned binary to get `0b1000 0000`.
Mantissa is `0b1` (dropping the implicit 1). Adding trailing zeros until it's 23 bits long gives `0b100 0000 0000 0000 0000 0000`.
Putting sign, exponent, mantissa together:
`0b0 1000 0000 100 0000 0000 0000 0000 0000`
Regrouping bits:
`0b0100 0000 0100 0000 0000 0000 0000 0000`
Converting to hex: `0x4040 0000`

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 li t4 0x4080 0000
2 li t3 0x4040 0000
3 fmv.w.x f4 t4
4 fmv.w.x f3 f3
5 fdiv.s f1 f4 f3
```

Solution: Put the bit patterns for 4 (in floating-point) and 3 (in floating-point) into two integer registers. For clarity, we used `t4` to hold 4 and `t3` to hold 3 in our solution.
Then, move these bit patterns into floating-point registers. For clarity, we used `f4` to hold 4 and `f3` to hold 3 in our solution.
Finally, use floating-point division to compute $4/3$ and put the result in `f1`.
Note that something like `li t4 4` and `li t4 3` would not work; these would put the bit patterns `0x0000 0004` and `0x0000 0003` into floating-point registers, but when interpreted in floating-point, these bit patterns do not represent the numbers 4 and 3.
Note that we have to use `li t4 0x4080 0000` or `lui x4 0x40800` to change the upper 20 bits of the register. Using an instruction like `addi` won't work because I-type immediates are only 12 bits long.
Some alternate solutions do exist, such as computing $1 + 1/3$, or somehow computing that `0x3FAA AAAB` is the floating-point number closest to $4/3$ (how you'd compute this is out-of-scope) and hard-coding this number into `f1`.

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.

Solution: `0x43`
Grading: +1.5 for each correct byte, +3 for fully correct.
Explanation: `0b 1001110` gives us: $p_1 = 1$, $p_2 = 0$, $d_1 = 0$, $p_4 = 1$, $d_2 = 1$, $d_3 = 1$, and $p_4 = 0$. Going through the parity:

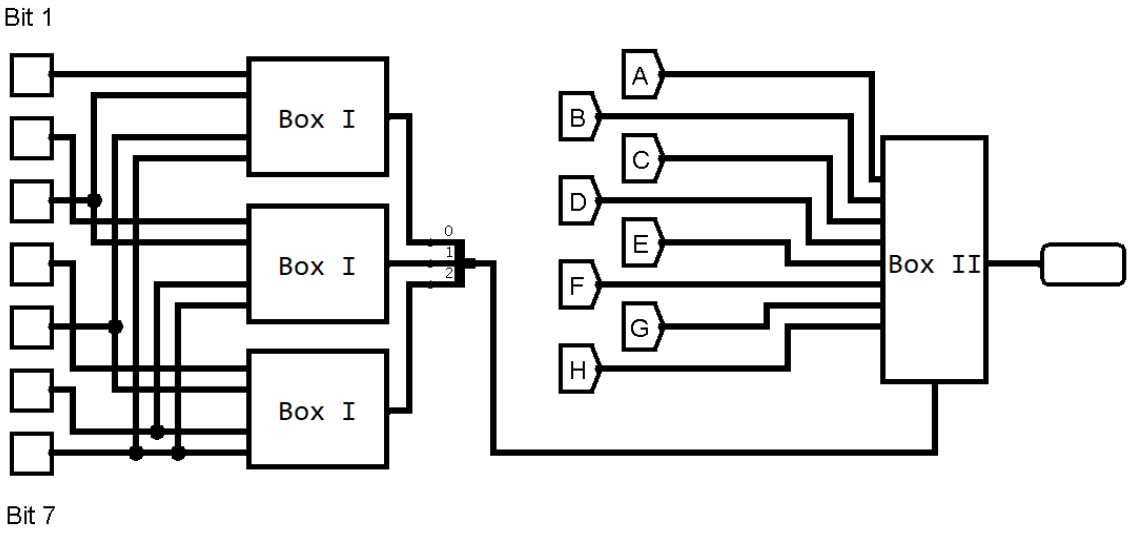
- Parity 1: $p_1 \oplus d_1 \oplus d_2 \oplus d_4 = 1 \oplus 0 \oplus 1 \oplus 0 = 0$
- Parity 2: $p_2 \oplus d_1 \oplus d_3 \oplus d_4 = 0 \oplus 0 \oplus 1 \oplus 0 = 1$
- Parity 4: $p_4 \oplus d_2 \oplus d_3 \oplus d_4 = 1 \oplus 1 \oplus 1 \oplus 0 = 1$

To find the erroneous bit, we sum up the parity indices where it was incorrect (not even parity, AKA the XOR returned 1): $2 + 4 = 6$ so we flip bit 6, giving us the bitstring `0b 1001100`. We can check this by substituting back $d_3 = 0$ into our parity calculations, which since we're now flipping 1 to 0, gives us an XOR or 0 on both parity 2 and parity 4. Thus our final datastring is `0b 0100`, or `0x4`.
We do the same thing for `0b 1000011`, giving us $p_1 = 1$, $p_2 = 0$, $d_1 = 0$, $p_4 = 0$, $d_2 = 0$, $d_3 = 1$, and $p_4 = 1$. Going through the parity:

- Parity 1: $p_1 \oplus d_1 \oplus d_2 \oplus d_4 = 1 \oplus 0 \oplus 0 \oplus 1 = 0$
- Parity 2: $p_2 \oplus d_1 \oplus d_3 \oplus d_4 = 0 \oplus 0 \oplus 1 \oplus 1 = 0$
- Parity 4: $p_4 \oplus d_2 \oplus d_3 \oplus d_4 = 0 \oplus 0 \oplus 1 \oplus 1 = 0$

The parity is correct so our datastring is `0b 0011` or `0x3`.
Thus our final data byte is `0x43`.

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)

Solution: Cascaded XOR gates can be used to determine the parity of the input bits; XOR returns 1 if the number of input "1" bits is odd, and 0 if the number of input "1" bits is even. This matches our definition of parity.
Note: On the paper exam, the question prompt was mistakenly printed as "Select 1 choice" but this was clarified at the start of the exam.

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)

Solution: We need to output the corrected data based on the parity bits; this can only be done using a mux. Note: On the paper exam, the question prompt was mistakenly printed as "Select 2 choices" but this was clarified at the start of the exam.

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

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

Solution: A, B, C, and E. These correspond to a parity bit value of 0, 1, 2, and 4, respectively. The 1, 2, and 4 cases occur when a bit error occurred in a parity bit; thus, the actual data is correct, so we output the data uncorrected. The 0 case is when no error is detected, so this should also output the data uncorrected. The remaining cases happen when one of the data bits got corrupted, which thus needs to be fixed before returning.

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:
Latest:

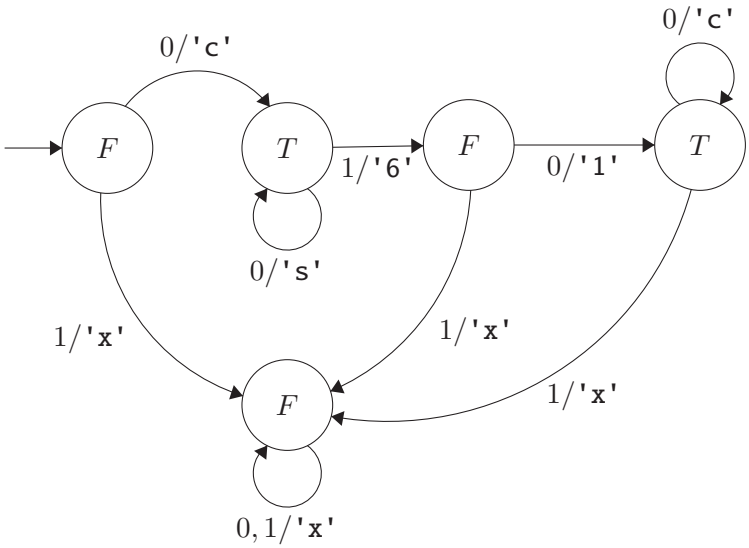
Solution:
Earliest: 6 ns. The values of A-H are computed at 1 ns. Then after the 5 ns delay of Box 2, the output changes.
Latest: 8 ns. The inputs change at 0 ns. Then after the 3 ns delay of Box 1, the bottom input to Box II changes. Then after the 5 ns delay of Box 2, the output changes.

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.

Solution: Points were given for either answer with sufficient justification.
Errors in the IMEM are often catastrophic (since a changed instruction can cause significant changes to a program's behavior), while DMEM errors can generally be corrected in software by computing the target value three times and returning the most common result. Further, the IMEM is in use more often than the DMEM, since every instruction uses the IMEM, while the DMEM is used only for loads and stores.
Errors in the DMEM, on the other hand, tend to be more common. Generally speaking, DMEM data is designed to be modifiable, with write support necessary. Data is more likely to be corrupted during a write or in volatile memory, so we can expect the DMEM to have more errors. Further, not all binary values correspond to a valid instruction, so there is already some error detection built-in to the IMEM; thus, it is more likely that an IMEM error gets caught immediately.
Overall, IMEM errors tend to be more catastrophic, while DMEM errors tend to be more frequent.

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

Solution: `css61c`
Grading: All-or-nothing.

Q6.2 (1 point) 0110

Solution: `c6xx`
Grading: All-or-nothing.

Q6.3 (1 point) 010000

Solution: `c61ccc`
Grading: All-or-nothing.

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.

Solution: An input is accepted if it contains at most one 1 bit, and if neither the leading nor trailing bit is a 1. All other inputs are not accepted.
We can deduce this from the FSM by noting that to reach the first T state on the left, we have to start by inputting a 0. Then, any further 0s keep us in the T state.
At this point, we could optionally choose to input a 1, which moves us right into the F state, as long as we then input another 0 to move us into the second T state on the right. Once we're in the second T state on the right, we can add any further 0s to keep us in the T state.

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.

Solution: 2^{13} bytes.
The virtual memory address and the physical memory address share the lower 13 bits. The page size cannot be larger, because that would mean the the addresses would refer to a different offset within the page.

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

Solution: 2^{28} bytes. The virtual memory address is 28 bits long if all leading 0s are removed, so the virtual memory must have at least 2^{28} bytes.

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?

Solution: Physical memory is 24 bits, offset is 12 bits, leaves 12 bits of PPN + 4 bits of metadata = 16 bits.

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	0x001EEF	○ TLB Hit ○ TLB miss, page table hit ● Page Fault
0x0000061C	2	0x00261C	○ TLB Hit ○ TLB miss, page table hit ● Page Fault
0xDEADB61C	2	0x00161C	● TLB Hit ○ TLB miss, page table hit ○ Page Fault
0xDEADB61B	1	0x00061B	○ TLB Hit ● TLB miss, page table hit ○ Page Fault
0xDEADB61C	1	0x00061C	● TLB Hit ○ TLB miss, page table hit ○ Page Fault
0x0000061A	2	0x00261A	○ TLB Hit ● TLB miss, page table hit ○ Page Fault
0x0000061A	1	0x00361A	○ TLB Hit ○ TLB miss, page table hit ● Page Fault

Solution:

For this entire question, page size is 2^{12} bytes, so we need 12 bits to uniquely find a byte within a page. This means that the bottom 12 bits = 4 hex digits are the offset, which stay unchanged when translating from virtual to physical address.

We have 2^{32} bytes of virtual memory, so virtual addresses are 32 bits long. This leaves the top $32 - 12 = 20$ bits = 5 hex digits as the VPN.

We have 2^{24} bytes of physical memory, so physical addresses are 24 bits long. This leaves the top $24 - 12 = 12$ bits = 3 hex digits as the PPN.

Row 2: Process 2 is accessing virtual page `0xDEADB`. Process 2 has never requested this page before, so it's a page fault and a new physical page, `0x001`, is allocated. The TLB now maps `0xDEADB` to `0x001`.

Row 3: Process 2 is accessing virtual page `0x00000`. Process 2 has never requested this page before, so it's a page fault and a new physical page, `0x002`, is allocated. The TLB now maps `0xDEADB` to `0x001` and `0x00000` to `0x002`.

Row 4: Process 2 is accessing virtual page `0xDEADB`. We have this VPN in our TLB, so we have a TLB hit and we can translate the VPN to `0x001`.

Row 5: Process 1 is accessing virtual page `0xDEADB`. We switched processes, so our TLB is empty again. However, process 1 has requested page `0xDEADB` before in row 1, so we can look in the page table and translate VPN `0xDEADB` to PPN `0x000`. The TLB now maps `0xDEADB` to `0x000`.

Row 6: Process 1 is accessing virtual page `0xDEADB`. We have this VPN in our TLB, so we have a TLB hit and we can translate the VPN to `0x000`.

Row 7: Process 2 is accessing virtual page `0x00000`. We switched processes, so our TLB is empty again. However, process 2 has requested page `0x00000` before in row 3, so we can look in the page table and translate VPN `0x00000` to PPN `0x002`. The TLB now maps `0x00000` to `0x002`.

Row 8: Process 1 is accessing virtual page `0x00000`. We switched processes, so our TLB is empty again. Also, process 1 has never requested virtual page `0x00000` before. This is a page fault, and causes a new physical page, `0x003`, to be allocated.

Grading:

Row 2: We're looking for the highest unused page number. Since this is the first blank and the filled-in row used `0x000`, this page number must be `0x001` for credit.

Row 3: We're looking for the highest unused page number. This might be `0x001` if the student put `0x000` in row 2, or `0x002`, but no higher than that.

Row 4: We're looking for the same page number used in Row 2, which could be `0x000` or `0x001`.

Row 5: We're looking for the same page number used in Row 1, which must be `0x000`.

Row 6: We're looking for the same page number used in Row 1, which must be `0x000`.

Row 7: We're looking for the same page number used in Row 3, which could be `0x001` or `0x002`.

Row 8: We're looking for the highest unused page number, which could be `0x001` (unlikely), `0x002`, and `0x003`.

Online exams had an alternate row 7, where the address was `0xDEADB61A`. Alternate solution: `0xDEADB61A` maps to `0x00161A`. This page was previously seen mapping to physical page 1, in Row 4. This is a TLB miss, page table hit (like the original version).

Online exams had an alternate row 8, where the address was `0xDEADB61A`. Alternate solution: `0xDEADB61A` maps to `0x00061A`. This page was previously seen mapping to physical page 0, in Row 1. This is a TLB miss, page table hit (no longer a 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.

```
1 int32_t vector_mul_positive(int32_t *a, int32_t *b, int32_t N) {
2     int32_t result[4];
3     __m128i sum_v = vset(0);
4     __m128i cond_v = vset(0);
5     #pragma omp parallel for
6     for (int i=0; i<N; i+=4) {
7         __m128i curr_v1 = vload(a+i);
8         __m128i curr_v2 = vload(b+i);
9         __m128i mul = vmul(curr_v1, curr_v2);
10        __m128i tmp = vcmpgt(mul, cond_v);
11        tmp = vand(tmp, mul);
12        #pragma omp critical
13        sum_v = vadd(sum_v, tmp);
14    }
15    vstore(result, sum_v);
16    return result[0] + result[1] + result[2] + result[3];
17 }
```

Solution:

Alternate solution: Line 4 is `vset(-1)`, and line 10 is `vcmpgt(mul, vmul(mul, cond_v)`. Instead of comparing the element-wise products to 0, you could compare the element-wise dot products to their inverses. If a number is greater than its inverse, the number must be positive (e.g. 1 is greater than -1, so 1 is positive, but -5 is not greater than 5, so -5 is negative).

Alternate solution: Line 6 is `int i=0; i<N/4; i++`, line 7 is `a+(i*4)`, and line 8 is `b+(i*4)`.

Alternate solution: Line 11 puts the result vector of `vand` into another vector variable (e.g. `tmp` or some newly-declared variable like `__m128i mask`, and line 13 uses that variable (e.g. `sum_v = vadd(sum_v, mask)`).

Nothing on this page will be graded.

Is there anything you want us to know?