After the exam, indicate on the line above where you fall in the emotion spectrum between “sad” & “smiley”...

<table>
<thead>
<tr>
<th>Last Name</th>
<th>Answer Key</th>
</tr>
</thead>
<tbody>
<tr>
<td>First Name</td>
<td></td>
</tr>
<tr>
<td>Student ID Number</td>
<td></td>
</tr>
<tr>
<td>Login cs61c-</td>
<td></td>
</tr>
<tr>
<td>Login First Letter (please circle) abcdefghijklmnopqrstuvwxyz</td>
<td></td>
</tr>
<tr>
<td>Login Second Letter (please circle) abcdefghijklmnopqrstuvwxyz</td>
<td></td>
</tr>
<tr>
<td>The name of your SECTION TA (please circle) Anirudh, Brian, Eric, Ian, Paul, Sean</td>
<td></td>
</tr>
<tr>
<td>Name of the person to your Left</td>
<td></td>
</tr>
<tr>
<td>Name of the person to your Right</td>
<td></td>
</tr>
<tr>
<td>All the work is my own. I had no prior knowledge of the exam contents nor will I share the contents with others in CS61C who have not taken it yet. (please sign)</td>
<td></td>
</tr>
</tbody>
</table>

Instructions (Read Me!)

- This booklet contains 8 numbered pages including the cover page. Put all answers on these pages; don’t hand in any stray pieces of paper.
- Please turn off all pagers, cell phones & beepers. Remove all hats & headphones. Place your backpacks, laptops and jackets at the front. Sit in every other seat. Nothing may be placed in the “no fly zone” spare seat/desk between students.
- You have 180 minutes to complete this exam. The exam is closed book, no computers, PDAs or calculators. You may use two pages (US Letter, front and back) of notes and the green sheet.
- There may be partial credit for incomplete answers; write as much of the solution as you can. We will deduct points if your solution is far more complicated than necessary. When we provide a blank, please fit your answer within the space provided. “IEC format” refers to the mebi, tebi, etc prefixes.
- **You must complete ALL THE QUESTIONS, regardless of your score on the midterm.** Clobbering only works from the Final to the Midterm, not vice versa. You have 3 hours... relax.

<table>
<thead>
<tr>
<th>Question</th>
<th>M1</th>
<th>M2</th>
<th>M3</th>
<th>Ms</th>
<th>F1</th>
<th>F2</th>
<th>F3</th>
<th>F4</th>
<th>Fs</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Minutes</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>60</td>
<td>30</td>
<td>30</td>
<td>30</td>
<td>30</td>
<td>120</td>
<td>180</td>
</tr>
<tr>
<td>Points</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>30</td>
<td>22</td>
<td>23</td>
<td>22</td>
<td>23</td>
<td>90</td>
<td>120</td>
</tr>
<tr>
<td>Score</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>30</td>
<td>22</td>
<td>23</td>
<td>22</td>
<td>23</td>
<td>90</td>
<td>120</td>
</tr>
</tbody>
</table>
M1) Hacker’s Delight (10 pts, 20 min)

a) The function mystery lives at address 0xF0000000 (but mystery obviously doesn’t know that). Fill in the blanks with TAL MIPS so that mystery returns the address of itself in $v0. *(Hint: you should do this part without calls to lw or sw!*)

```
mystery:  addu $t0 $ra $0
          jal here
          ___________________
          addiu $v0 $ra -8
          here:     ___________________
          jr $t0
```

b) Let’s assume you wrote part (a) correctly. Fill in the blanks so that you get the following behavior. LookupInGreenSheet() returns the 32 bits corresponding to the MIPS function.

```
main() {
    unsigned int *A, Baseline, mystery();
    A = (unsigned int *) mystery(NULL); // argument doesn’t matter, not used
    A[0] = LookupInGreenSheet("_____________");  // $v0 $v0 4($a0)
    A[1] = LookupInGreenSheet("addiu ___________");  // $v0 $v0 1
    A[2] = LookupInGreenSheet("_____________");  // sw $v0 4($a0)
    Baseline = mystery(A)-2; // Consider mystery’s first return val (minus 2) as baseline
    printf("%d\n", mystery(A) - Baseline);  // 4 is printed
    printf("%d\n", mystery(A) - Baseline);  // 8 is printed
    printf("%d\n", mystery(A) - Baseline);  // 16 is printed
    printf("%d\n", mystery(A) - Baseline);  // 32 is printed
    // etc
}
```
M2) Cache Money, y’all (10 pts, 20 min)

The following C code is run on a 32-bit MIPS machine with a 4 KiB L1 cache with 10 offset bits. Vectors A, B, C live in different places of memory, are of equal size (n is a power of 2 >> cache size), and are all block aligned.

AddVectors(uint8_t *A, uint8_t *B, uint8_t *C, int n) {    // sizeof(uint8_t) = 1
    for (int i = 0; i < n; i++)
        C[i] = A[i] + B[i];
}

0   3n   1023   1

a) If the cache is **direct mapped**, what is the lowest _____:_____ and highest _____:_____ hit:miss ratio?

0   3n

b) If it is **2-way set-associative**, what is the lowest _____:_____ hit:miss ratio?

1023   1

c) If it is **4-way set-associative (with LRU replacement)**, what is the lowest _____:_____ hit:miss ratio?

d) If it is **4-way set-associative (with LRU replacement)**, and we change the interpretation of bits from “T-I-O” (tag-index-offset) to “T-O-I”, and don’t tell the cache, what is the worst that could happen? (in one sentence)

**Nothing, since there’s no I**

e) If it is **4-way set-associative (with LRU replacement)**, and we change the interpretation of bits from “T-I-O” (tag-index-offset) to “O-I-T”, and don’t tell the cache, what is the worst that could happen? (in one sentence)

We could get the wrong values, since a cache block is loaded assuming adjacent memory slots, and the tags (now the lower-order 22 bits) of A[0] and B[0] could match, so when it read B[0] it would think the block is already loaded and read some other byte of that block thinking it was B when it was really A’s data.
M3) Got some numbers, down by the C... (10 pts, 20 mins)

a) Fill in the blanks of this C code to write a ‘1’ to every byte of a single 2 KiB page on the heap, don’t write anything to any other pages. You may assume that there are a few available heap pages. Use as little memory as possible (you might need to ask for more than 2 KiB). memset is not allowed, you can’t allocate anything already allocated on the heap, and two consecutive memory requests may not be near each other.

```c
#define PAGE 0x__________ // 2 KiB in hex

void TouchEveryPageByte() {
    uint8_t *ptr, *tmp;
    ptr = (uint8_t *) malloc (2 * PAGE);
    tmp = (ptr | (PAGE-1)) + 1;
    for (int i = 0; i < PAGE; i++)
        *(tmp+i) = 1;
}
```

b) Here are 3 different numerical encodings of 32 bits.

<table>
<thead>
<tr>
<th>Float</th>
<th>Fixed point</th>
<th>Rational</th>
</tr>
</thead>
<tbody>
<tr>
<td>float</td>
<td>XXXXXXXXXXXXXXXXXXX.YYYYYYYYYYYYYYYYYYYYYYY ...where xs are interpreted as 2s complement, ys are interpreted the standard way bits on the right of a fixed-point representation are interpreted.</td>
<td>NNNNNNNNNNNNNNNDDDDDDDDDDDDDDDDDDDD ...where ns are interpreted as a biased numerator (the bias is set in the usual way so that roughly half the numerators are positive, half are negative), and ds as an unsigned denominator. A denominator of 0 means infinity; if both num and denom are zero it’s a NaN. This # basically looks like: NNNNNNNNNNNNNNNN &lt;- biased --------- DDDDDDDDDDDDDDDD &lt;- unsigned</td>
</tr>
</tbody>
</table>

Rank these three in terms of the following categories (ties are allowed). We did the first two for you.

<table>
<thead>
<tr>
<th>Category</th>
<th>Float</th>
<th>Fixed Point</th>
<th>Rational</th>
</tr>
</thead>
<tbody>
<tr>
<td>Distance from “A” of first letter in name</td>
<td>1st</td>
<td>1st</td>
<td>2nd</td>
</tr>
<tr>
<td>Fewest # of letters in name</td>
<td>1st</td>
<td>3rd</td>
<td>2nd</td>
</tr>
<tr>
<td>Fewest number of zeros</td>
<td>2 (2)</td>
<td>1 (1)</td>
<td>3 (2^16)</td>
</tr>
<tr>
<td>Smallest positive number</td>
<td>1 (2^-149)</td>
<td>2 (2^-16)</td>
<td>2 (2^-16)</td>
</tr>
<tr>
<td>Closest representable number to -1/3</td>
<td>2 (23 bits close)</td>
<td>3 (16 bits close)</td>
<td>1 (exact)</td>
</tr>
<tr>
<td>Actual number farthest to the left on the number line</td>
<td>1 ~(− 2^127)</td>
<td>2 (~− 2^15)</td>
<td>3 (~− 2^15−1)</td>
</tr>
</tbody>
</table>
F1) **Code, Earl Gray, Hot... (22 pts, 30 mins)**

Bits can mean anything! *Gray codes* are a way of ordering the binary encoding of symbols/values such that successive symbols/values only differ by *exactly one* bit in their representation. (Gray codes are useful for error correction in digital communication!) As an example, here’s one of many possible valid orderings of the four-bit Gray codes (with the consecutive differing bits underlined):

\[
\begin{align*}
0000, & \quad 0001, \quad 0011, \quad 0010, \quad 0110, \quad 0111, \quad 0101, \quad 0100, \quad 1100, \quad 1101, \quad 1111, \quad 1110, \quad 1010, \quad 1011, \quad 1001, \quad 1000
\end{align*}
\]

Any cycle of this sequence will also be a valid ordering, just so long as no code differs more than one bit from its neighboring codes. When two binary representations differ by exactly one bit, we call the pair *Gray adjacent*.

a) Annotate this cube such that each corner is Gray-adjacent to its 3 neighbors. (Corner a is given.)

```
 a 000  b ___  c ___  d ___  e ___  f ___  g ___  h ___
```

Now imagine a one-bit stream of data, from which we’re constantly assembling two-bit values. We use the bit received at time \( t \), \( b_t \), in conjunction with the previously streamed-in \( b_{t-1} \) to make the code “\( b_t b_{t-1} \)”.

Note that the most recent bit is the more significant. We want a finite state machine that indicates whether the two most recent two-bit values – that is to say “\( b_t b_{t-1} \)” and “\( b_{t-1} b_{t-2} \)”, are Gray code adjacent. (Take two zeros as the initial condition of the stream – if we stream in bit \( b_0 \) at \( t = 0 \), the three most recent bits we’ve seen will be taken as “\( b_000 \)” giving us the values “\( 00 \)” and “\( 00 \)”.) Here’s an example for the bit-stream 1101 (with our initial condition bits bracketed):

<table>
<thead>
<tr>
<th>Input</th>
<th>Most recent 3 bits</th>
<th>Most recent codes</th>
<th>Adjacent?</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1[00]</td>
<td>10 and 00</td>
<td>Yes</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1[10]</td>
<td>11 and 10</td>
<td>Yes</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0[11]</td>
<td>01 and 11</td>
<td>Yes</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1[01]</td>
<td>10 and 01</td>
<td>No</td>
<td>0</td>
</tr>
</tbody>
</table>

b) Let’s presume a state variable such that the three most recent bits are *Input*, (Current) \( S_2 \), (Current) \( S_1 \) (telling us that the next state bits must be *Input*, (Current) \( S_2 \)). Fill in the *Output* column of the table.

c) Add the state transition arrows, along with the input and output bits on each transition, to the state diagram below. One is done for you, as well as an arrow for the 00 initial condition:

```
  00
 /  \
/    \
1    11
  \
  01
```

a) Give the *simplest* Boolean expression for *Output* in terms of *Input*, (Current) \( S_2 \), and (Current) \( S_1 \). Show your work.
Consider a system with 1 GiB of physical memory, a 32-bit virtual address space, and 2 KiB pages.

a₁) How many bits to uniquely identify each page of virtual memory? ________ Physical memory? ________

\[ 2^{32} \text{ virt. addrs, } 2^{11} \text{ byte pages } \rightarrow 32 - 11 = 21 \text{ bits} \]

\[ 2^{30} \text{ phy. addrs, } 2^{11} \text{ byte pages } \rightarrow 30 - 11 = 19 \text{ bits} \]

a₂) How many entries does the page table contain? ________

\[ 2^{21} \]

The following code runs on the above system. \texttt{arr} is an integer array of size \texttt{ARRAY\_SIZE}. \texttt{SKIP} is a positive integer less than or equal to the page size. 32 page faults were detected while running the loop, all due solely to reading from \texttt{arr}.

\begin{verbatim}
for(int i = 0; i < ARRAY\_SIZE; i+=SKIP)
    total += arr[i];
\end{verbatim}

b₁) What’s the smallest possible value of \texttt{ARRAY\_SIZE}? ________

\[ 2 + 30 \times 2^8 \]

b₂) The TLB for this system has four entries, each listing a single VPN-PPN pair, and it uses an LRU replacement policy. In the middle of running the above code, another process was given CPU time – but for some reason an entry of the TLB didn’t flush during the switch to this process. This unflushed entry corresponds to a page filled with elements of \texttt{arr}. Both processes executed without raising an exception. Why might \texttt{total} be incorrect? At worst, how many incorrect values were added to \texttt{total}?

The other process might have performed writes to memory at virtual addresses that match those used by array elements living on the unflushed page. At worst, \(\frac{2\text{KiB}}{\text{SKIP}}\) incorrect values were added.

b₃) Assume we are using RAID 1 for this system’s disk, and the above scenario unfolds.

How many pages on disk would have been modified in the best case? ________ Worst case? ________

None

64

c) In which situations might some form of DMA be used? Check all that apply.

- Handling a page fault  \(\times\)
- Communication between servers in a datacenter  \(\times\)
- Playing music from a file already in memory  \(\checkmark\)
- Scrolling through a (fully downloaded) web page  \(\times\)

d) Explain (in the provided space) how, why or when the statements below are true:

<table>
<thead>
<tr>
<th>Statement</th>
<th>Justification</th>
</tr>
</thead>
<tbody>
<tr>
<td>Polling can be faster than interrupt-driven I/O.</td>
<td>Polling has no overhead in the way that interrupt handling requires a context shift</td>
</tr>
<tr>
<td>High frequency polling is undesirable.</td>
<td>High frequency polling requires lots of wasted CPU time and power waiting for the I/O data to become valid</td>
</tr>
<tr>
<td>Low frequency polling can make data transfer slow.</td>
<td>If the device fills up its buffer faster than it’s polled out, data needs resending</td>
</tr>
</tbody>
</table>
I/O interrupts can be combined with I/O polling to get the best of both. Interrupts can tell the system when a burst of I/O data has begun, and polling can be used subsequently until the burst of data is over (requiring less interrupt overhead losses).

We have a system to which we can instantaneously add and remove cores – adding more cores never leads to slowdown from things like false sharing, thread overhead, context switching, etc. When the program $foo$ is executed to completion with a single core in the system, it completes in 10 minutes. When $foo$ is run with a total of three cores in the system, it completes in 6 minutes. How long would it take to execute $foo$ on this magical system as the number of cores approaches infinity?

The speed up from 1 to 3 nodes suggests 60% of $foo()$ is parallelizable, so Amdahl’s law suggests a maximum speed up of 2.5x over the single core in the limit – meaning $foo$ would take 4 minutes.

**F3) Datapathology ... (22 pts, 30 mins)**

Consider the single cycle datapath as it relates to a new MIPS instruction, `save and duplicate`:

```
sdup rt, rs, imm
```

The instruction does the following:
1) Stores the value in $rs$ into memory at the address stored in $sp$, offset by $imm$.
2) Copies the value in $rs$ into $rt$.

**Ignore pipelining for parts (a)-(c).**

a) Write the RTL corresponding to `sdup rt, rs, imm`

```
R[rt] ← R[rs]; Mem[R[$sp] + imm] ← R[rs]
```

b) Change as little as possible in the datapath above (draw your changes right in the figure) to enable `sdup`. List all your changes below. Your modification may use muxes, wires, constants, and new control signals, but nothing else. (You may not need all four provided boxes.)

<table>
<thead>
<tr>
<th>(i)</th>
<th>Add a mux to select rs or $sp$ for Ra</th>
</tr>
</thead>
<tbody>
<tr>
<td>(ii)</td>
<td>Add a mux to select either rs or rt for Rb</td>
</tr>
<tr>
<td>(iii)</td>
<td>Instead of the busW line, add a mux to select between busB and (new line) busW</td>
</tr>
<tr>
<td>(iv)</td>
<td>A new control signal <code>sdup</code> to control all the above muxes</td>
</tr>
</tbody>
</table>

c) We now want to set all the control lines appropriately. List what each signal should be, either by an intuitive name or {0, 1, “don’t care“}. Include any new control signals you added. Don’t allow $sdup$ to access any memory below the stack pointer.
d) Now consider `sdup` run on the 5-stage MIPS pipeline. Consider the following pairs of instructions (i) through (iii) independently. Circle each pair that poses a data hazard.

- For each circled pair, explain how to resolve the hazard.
- If stalling is needed, provide the number of stall cycles required. (Please resolve the hazard with a minimal number of stall cycles.)
- If forwarding is needed, state after which stage the data is available and at which stage the data is needed.

<table>
<thead>
<tr>
<th>Instruction 1</th>
<th>Instruction 2</th>
<th>Instruction 3</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>add $sp $sp -8</code></td>
<td><code>add $t0 $t0 $t1</code></td>
<td><code>lw $t0 0($a1)</code></td>
</tr>
<tr>
<td><code>sdup $t0 $a0 0</code></td>
<td><code>sdup $t0 $a1 4</code></td>
<td><code>sdup $t1 $t0 0</code></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>RegDst</th>
<th>RegWr</th>
<th>nPC_sel</th>
<th>ExtOp</th>
<th>ALUSrc</th>
<th>ALUctr</th>
<th>MemWr</th>
<th>MemtoReg</th>
<th>sdup</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>+4</td>
<td>Zero</td>
<td>imm</td>
<td>add</td>
<td>1</td>
<td>X</td>
<td>1</td>
</tr>
</tbody>
</table>

8/10
Berkeley is looking at opening CS61C to the masses - but the glookup service is far too slow! In addition to ordering more servers, the inst staff have asked you to look at optimizing the glookup service. They've already determined that spinlocks are a performance issue, but there's no way to replace them. Perhaps you can make the locks themselves faster, though! You identify two possible definitions for a spinlock:

```c
typedef struct spinlock {
    char value; // 1 if the lock is currently held else 0
} spinlock_t;

typedef struct spinlock_padded {
    char value; // 1 if the lock is currently held else 0
    char padding[63]; // 63 bytes of padding (never used)
} spinlock_padded_t;
```

a.) Implement the generic `spin_lock` and `spin_unlock` functions in C, which will be called on pointers to `spinlock_t` and `spinlock_padded_t`. You use the following test-and-set function:

```c
int CAS(char *dest, char test, char value)
CAS compares the values of *dest and test:
• If they're equal, *dest is set to value, and CAS returns 1
• If not, CAS returns 0, changing nothing in the heap
```

State any assumptions you need to make.

```c
void spin_lock(void *lock) {
    while (! CAS(lock, 0, 1))  // Until the lock can be set...
        ;  // ... do nothing
}

void spin_unlock(void *lock) {
    (char*) lock = 0;
}
```

You evaluate the performance of both types of spinlocks using two different parallelism benchmarks. They are both locked and unlocked using the same generic `spin_lock` and `spin_unlock` procedures.

a.) On one benchmark, the padded spinlock worked faster. Why might this be? Describe what the program could be doing to cause this.

```
With unpadded locks, if two adjacent memory locations are associated with two different spin locks, two threads could try and poll each location sequentially, and false sharing can result. The larger padded lock causes each spinlock memory location to be spread apart from the others.
```

a.) On the other benchmark, the unpadded spinlock worked faster. Why might this be? Describe what the program could be doing to cause this.

```
If the benchmark allocates many spinlocks per thread, it can be faster to have a single thread’s locks all fit in the same cache blocks in to improve the cache hit rate.
```

Part of the glookup upgrade includes some convenient and speedy statistical reporting. The system has exactly one record for each student of the following format:

```
(Student Name, Student’s Advisor’s Name, Student’s Course Load)
```

where “course load” encodes the number of credits the student took in the most recent semester. To get a sense of which faculty are pushing students to take more or fewer classes, the administration have asked inst to ask you to write a MapReduce scheme for finding, for each advisor, the greatest course load any of that advisor’s students are taking. The output record should resemble: (Advisor Name, Max Course Load)

b.) Describe the process by which your mappers would read an input record and produce fodder for the reducers.
Please explicitly include the (key, value) pairs your map() function would emit.

For each record the mapper sees, remove the student’s name and just echo the advisor name (output key) and course load (output value)
(key, value): ( ______Student’s Advisor's Name______ , ______Course Load____________ )

b2) Describe how your reducers would transform the pairs received from the mappers into the output records.