CS 61C Spring 2024

Yan, Yokota Midterm

Solutions last updated: Monday, March 18th, 2024

PRINT Your Name: ______________________________

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:	14	7.5	7	20	23.5	12	16	0	100

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

○ Unselected option (completely unfilled)

● Only one selected option (completely filled)

✓ Don't do this (it will be graded as incorrect)

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

☐ You can select

■ multiple squares

■ (completely filled)

Anything you write outside the answer boxes or you ~~cross out~~ will not be graded. If you write multiple answers, your answer is ambiguous, or the bubble/checkbox is not entirely filled in, we will grade the worst interpretation. For coding questions with blanks, you may write at most one statement per blank and you may not use more blanks than provided.

If an answer requires hex input, you must only use capitalized letters (`0xDEADBEEF` instead of `0xdeadbeef`). For hex and binary, please include prefixes in your answers unless otherwise specified, and do not truncate any leading 0's. For all other bases, do not add any prefixes or suffixes.

Write the statement below in the same handwriting you will use on the rest of the exam.

I have neither given nor received help on this exam (or quiz), and have rejected any attempt to cheat; if these answers are not my own work, I may be deducted up to 0x0123 4567 89AB CDEF points.

SIGN your name: ______________________________

Q1 *Potpourri* **(14 points)**

For Q1.1-Q1.2, convert the 8-bit binary `0b1100 1111` to decimal, treating it as...

Q1.1 (1 point) ...an **unsigned** integer.

Solution: 207

Q1.2 (1 point) ...a **two's complement** integer.

Solution: -49

Q1.3 (1 point) Convert the 12-bit number 4727_8 to hexadecimal.

Solution: `0x9D7`

`0o4727 = 0b100 111 010 111 = 0b1001 1101 0111 = 0x9D7`

Q1.4 (1.5 points) Suppose there are 615 students enrolled in CS61C. What is the minimum number of bits needed to uniquely identify each student? Express your answer in decimal form.

Solution: 10 bits

To be able to represent 615, we need to be able to represent the next highest power of two, which is $1024 = 2^{10}$.

Q1.5 (1.5 points) An IEEE-754 double-precision floating point number can represent every integer that a 32-bit two's complement number can.

[✓ (A) True] (B) False

Solution: In a 32-bit two's complement number, there are effectively 31 significand bits (bits 0 through 30 inclusive) and one sign bit (bit 31). In an IEEE-754 double-precision floating point number, there are 52 significand bits, which means that we can represent any 32-bit two's complement number.

For Q1.6-Q1.7, consider a **16-bit** floating point format that follows the IEEE-754 standard, with 1 sign bit, 5 exponent bits (with a bias of -15) and 10 significand bits.

Q1.6 (1.5 points) Convert 6.25 into hexadecimal in this floating point format. If it cannot be represented, write "None".

Solution: 0x4640

The sign bit should be 0 as we're working with a positive integer. Converting 6.25 into binary gives us `0b110.01 = 0b1.1001 * 22^`. This means that our exponent is 2, which would be stored as 129 (`0b10001`) due to the bias. The mantissa would start with `0b1001` and the remaining bits would be 0s.

`0x 0 10001 1001000000 = 0x4640`

Q1.7 (1.5 points) What is the smallest positive value this format can support? Express your answer as 2^n where n is an integer.

Solution: 2^{-24}

The LSB of the significand represents 2^-10, and the exponent associated with denormalized numbers is $2^{\text{Bias}+1} = 2^{-14}$. Thus, the smallest representable number is $\left(2^{-10}\right)\left(2^{-14}\right) = 2^{-24}$.

Q1.8 (1.5 points) All labels in an assembly file can be referenced from other assembly files.

(A) True [✓ (B) False]

Solution: Labels can only be referenced if there's a `.globl` directive for it.

Q1.9 (1.5 points) The first step of CALL will take in a C file and turn it into an object file.

(A) True [✓ (B) False]

Solution: The first step of CALL is the compiler, and it compiles code from a higher-level language into an assembly language, such as RISC-V.

Q1.10 (2 points) Convert the following RISC-V machine code into its corresponding instruction. If there is an immediate value, express it in decimal form. Provide the appropriate register names, not numbers, where necessary (i.e.: `s5` instead of `x21`).

`0x0655 0F63`

Solution: `beq a0 t0 126`

Q2 *C-narios* (7.5 points)

Each of the following scenarios represents a bug in a program. For each of the scenarios, please indicate whether the bug is caused by an arithmetic overflow, precision loss, or another reason.

If you choose "Arithmetic Overflow" or "Precision Loss", please indicate the exact C type of the variable(s) that are involved from the following list of types:
`int8_t, uint8_t, int32_t, uint32_t, float, double`.

If you choose "Other", describe the likely bug **using at most 5 words**; no complete sentences needed.

Q2.1 (1.5 points) Completing level 255 in a game resets you to level 0.

[✓ (A) Arithmetic Overflow] (B) Precision Loss (C) Other

Solution: This is most likely an overflow of a `uint8_t`, as they have a range of 0 to 255. Many classic arcade games have bugs of this nature, such as Tetris, Pac-Man, and Pokemon Red.

Q2.2 (1.5 points) A rectangular platform that is designed to move in a sinusoid up-and-down pattern in a game slowly drifts upwards over the course of several days.

(A) Arithmetic Overflow [✓ (B) Precision Loss] (C) Other

Solution: In any floating point computation, the value of the float needs to be rounded to the nearest representable float after each computation; this introduces some rounding error at each time tick. If rounding behavior consistently favors one direction (e.g. rounding towards zero always), then a function that is meant to be a stable sinusoid will steadily drift in one direction due to precision loss. Each calculation can have a relative error at most the smallest distance between representable numbers in the floating point scheme; for floats, this is $1/2^{23}$, so we would expect floating point error to become significant after a few million cycles of the sinusoid; taking into account the fact that we likely don't get the maximum error each cycle, this can reasonably take several days. For doubles, the relative error is bounded by $1/2^{52}$ instead, which would require at least several quadrillion calculations. Even if the platform was moving up and down 1000 times a second, it would take around 30 years of operation before the error becomes noticeable. Thus it is unlikely that precision loss on a double alone would cause this error.

This particular bug exists in certain versions of Super Mario 64, and is caused specifically by doubles being converted to floats (and getting rounded to 0) repeatedly. Notably, this bug allows for the entire game to be completed without a single A button press.

Q2.3 (1.5 points) A game's score counter behaves unexpectedly when the score exceeds $\approx 10^{308}$.

[✓ (A) Arithmetic Overflow] (B) Precision Loss (C) Other

Solution: Since the bug occurs when a sufficiently large number is hit, this is likely an overflow error. However, most integer formats overflow far earlier than 10^{308}; a 32-bit unsigned integer maxes out at $2^{32} \approx 4$ billion. We thus note that the overflow must occur on a floating point format. For floats, overflow should occur at the maximum possible exponent of approximately 2^{127}, while for doubles, the overflow should occur at the maximum exponent of 2^{1023}. Noting that $2^{10} \approx 10^3$, we should expect a `float` overflow around 10^{38}, and a `double` overflow around 10^{307}. Thus, this is likely a `double` overflow.

This occurs in many incremental/idle games, as their scoring system tends to reach these ludicrously high values.

Q2.4 (1.5 points) When a program outputs a string, it infrequently prints seemingly random, corrupted characters after the expected string.

(A) Arithmetic Overflow (B) Precision Loss [✓ (C) Other]

Solution: This is most likely due to failing to properly null-terminate a string. If a string is not properly null-terminated, any print call will continue to read data as if it was string data until seeing a null terminator. However, 0 bytes are far more common than nonzero bytes, so in most cases, the next byte happens to be by chance a null terminator, masking this bug.

This is also a common symptom of a potential security vulnerability, since if a program can print random data as if it was a string, then that data can be disclosed to a user even if that data should not have been disclosed. See CS 161 for more details!

Q2.5 (1.5 points) After leaving a program open for several days, the program and all other programs running concurrently on your computer start to slow down.

(A) Arithmetic Overflow (B) Precision Loss [✓ (C) Other]

Solution: The most likely cause of this is due to excessive resource usage by the program, causing it and all other programs using that resource to slow down. Since this develops gradually over time, it is likely that this resource gets steadily more used as time passes. Given this, the most likely root cause is a memory leak, as most other computer resources generally don't steadily increase in use, and a memory leak would cause the program to increase its memory use over time.

Try as we might to avoid this bug, memory leaks often occur even in production code. In general, this type of bug is why it is sometimes recommended to try and restart things if they start getting slow.

Q3 `void *cf0` **(7 points)**

C's standard library has a built-in **`qsort`** function that implements the quicksort sorting algorithm. Here is an excerpt from its **`man`** pages.

```
void qsort(void *base, size_t nmemb, size_t size,
           int (*compar)(const void *, const void *));

DESCRIPTION
   The qsort() function sorts an array with nmemb elements of size size.
   The base argument points to the start of the array.

   The contents of the array are sorted in ascending order according to
   a comparison function pointed to by compar, which is called with two
   arguments that point to the objects being compared.

   The comparison function must return an integer less than, equal to,
   or greater than zero if the first argument is considered to be
   respectively less than, equal to, or greater than the second. If two
   members compare as equal, their order in the sorted array is undefined.
```

Implement the **`sort_matrices`** function, which sorts a **`list`** of **`matrix`** structs of length **`list_len`** by their **`size`** in ascending order using the **`qsort`** function. You'll need to implement your own comparison function **`compare_matrices`** to do this. Assume appropriate C standard libraries are already imported.

```
typedef struct {
  int **data;
  size_t size;
} matrix;

int compare_matrices (const void *p, const void *q) {

  return ((matrix*) p)->size - ((matrix*) q)->size;
                             Q3.1
}

void sort_matrices (matrix *list, size_t list_len) {

  qsort(list, list_len,
        Q3.2  Q3.3

        sizeof(matrix), compare_matrices);
            Q3.4              Q3.5
}
```

Q4 *FCVT.S.WU* **(20 points)**

Implement `magnitude`, a RISC-V function, as follows:

- Input `a0`: a **nonzero** unsigned integer
- Returns in `a0`: the **index** of the most significant bit that is 1 in the binary representation of `a0`. The **least** significant bit is at index 0.

For example, `magnitude` of 2 (`0b10`) returns 1, and `magnitude` of 727 (`0b10 1101 0111`) returns 9.

Ensure that your implementation follows calling convention.

```
magnitude:
    li t0 0
    li t1 1
loop:
    beq t1 a0 end

    addi t0 t0 1
                    Q4.1

    srli a0 a0 1
                    Q4.2
    j loop
end:

    mv a0 t0
                    Q4.3
    jr ra
```

Solution: If our input is 1, then we immediately break to `end`, as we have 0 in `t0` already, and the value returned should be 0. Otherwise, we increment a counter in `t0` and shift `a0` right by one. Finally, we move our counter into `a0` to return it.

Note: `srai` does not work in this question as we will never reach `a0 == 1` if the MSB of the argument is 1.

Implement `convert`, a RISC-V function, as follows:

- Input `a0`: a **nonzero** unsigned integer
- Returns in `a0`: the IEEE-754 single-precision floating point representation of `a0`, **rounded down** if there is no exact representation.

For example, the integer 2 (`0x00000002`) should be converted to its corresponding floating point representation `0x40000000`. The integer 268435471 (`0x1000000F`) has no exact floating point representation; instead, the representation `0x4d800000` (rounded towards zero) should be returned.

You may assume that `magnitude` is implemented correctly and behaves as specified in the first part, regardless of your implementation above. You may not assume any specific implementation of `magnitude`, and you may not modify any `s` registers except for `s4`.

```
convert:
    # prologue omitted

    mv s4 a0
    jal ra magnitude

    addi t1 x0 32                             # set significand

    sub  t1 t1 a0____________________________
                     Q4.4

    sll  s4 s4 t1____________________________
                     Q4.5

    srli s4 s4 9_____________________________
                     Q4.6

    addi a0 a0 __________127_________        # set exponent
                         Q4.7

    slli a0 a0 ___________23_________
                         Q4.8

    add  a0 a0 s4____________________________
                     Q4.9

    # epilogue omitted
    jr ra
```

Solution: When we initialize `t1` to 32 and subtract the magnitude, we get the number of bits before (and including) the first 1 bit. Shifting left by this amount aligns the first bit of the would-be significand all the way to the left, while removing the implicit 1, then we shift right by 9 into position. The return value of the `magnitude` function is also the exponent we need to multiply by; we add the bias 127 then shift left by 23 to place it properly. Finally, we combine the values.

For example: `0x61C = 1564 = 0b0110 0001 1100`

`magnitude(0x61C) = 10`, and `32 - 10 = 22`. The shifts produce values that look like this:

```
     original:   0b0000 0000 0000 0000 0000 0110 0001 1100
left shift 22:   0b1000 0111 0000 0000 0000 0000 0000 0000
right shift 9:   0b0000 0000 0100 0011 1000 0000 0000 0000
```

Adding the bias to the exponent/magnitude gives `127 + 10 = 137 = 0b1000 1001`. Shifting left by 23 and adding the significand gives:

```
     exponent:   0b0100 0100 1000 0000 0000 0000 0000 0000
+ significand:   0b0000 0000 0100 0011 1000 0000 0000 0000
            =    0b0100 0100 1100 0011 1000 0000 0000 0000
```

which is precisely the float representation of 1564.

Q4.10 (2.5 points) Which registers need to be saved in the prologue and restored in the epilogue for `convert` to satisfy calling convention?

- [✓ (A) `s4`]
- [✓ (B) `ra`]
- ☐ (C) `a0`
- ☐ (D) `t1`
- ○ (E) None

Solution: `s4` and `ra`. `ra` is needed since we call the function `magnitude`.

Q5 *Pets* **(23.5 points)**

The `Pets` struct is defined as follows:

```
typedef struct {
  uint32_t count; // The number of pets in this struct
  char **names;   // An ordered list of each pet's name
} Pets;
```

The function `void add_pet(Pets *p, char *name)` is defined as follows:

- `Pets *p`: A valid pointer to a `Pets` struct.
- `char *name`: The pet's name as a properly null-terminated string.

You may assume the following:

- For any `Pets` struct, `count` is initialized to 0 and `names` is initialized to `NULL`.
- Dynamic memory allocation will never fail.
- All relevant standard libraries have been imported.

Below is an example of the behavior of `add_pet`:

```
int num_cats = 3;
int main () {
  Pets dogs = {0, NULL};
  char dog[] = "Harold";
  add_pet(&dogs, dog);
  dog[0] = 'D';
  dog[1] = 'a';
  dog[2] = 'v';
  dog[3] = 'e';
  dog[4] = '\0';
  add_pet(&dogs, dog);
  printf("%d\n", dogs.count); // output is 2
  printf("%s\n", (dogs.names)[0]); // output is Harold
  printf("%s\n", (dogs.names)[1]); // output is Dave
  return 0;
}
```

Useful C stdlib function prototypes:

```
void *malloc(size_t size);
void free(void *ptr);
void *calloc(size_t num_elements, size_t size);
void *realloc(void *ptr, size_t size);

size_t strlen(char *s);
char *strcpy(char *dest, char *src);
```

Implement **`add_pet`** to match the described behavior above. Note: `realloc(NULL, n)` is equivalent to `malloc(n)`.

```
void add_pet(Pets *p, char *name) {

  int name_len = strlen(name);
                        Q5.1

  p->names = realloc(p->names,(p->count + 1) * sizeof(char*));
                                   Q5.2

  char *name_copy = malloc(name_len + 1);
                           Q5.3

  strcpy(name_copy, name);
           Q5.4     Q5.5

  *(p->names + p->count) = name_copy;
          Q5.6               Q5.7
  p->count = p->count + 1;
}
```

Solution: We need to reallocate memory for the list, **`p->names`** to add another name, copy **`*name`** into a new memory location and update the list count to implement **`add_pets`**.

For each of the following symbols from the example, choose which section of memory it would live in.

Q5.8 (1.5 points) **`num_cats`** (defined on line 1)

(A) Code [✓ (B) Static] (C) Stack (D) Heap

Solution: **`num_cats`** is in static memory because it's a variable defined outside of a function.

Q5.9 (1.5 points) **`add_pet`**

[✓ (A) Code] (B) Static (C) Stack (D) Heap

Solution: **`add_pet`** is in code since it's a function.

Q5.10 (1.5 points) `dogs.names`

○ (A) Code ○ (B) Static [✓ (C) Stack] ○ (D) Heap

Solution: `dogs.names` is on the stack since `dogs` was declared on the stack, and `names` is a member in the `Pets` struct.

Q5.11 (1.5 points) `(dogs.names)[1]`

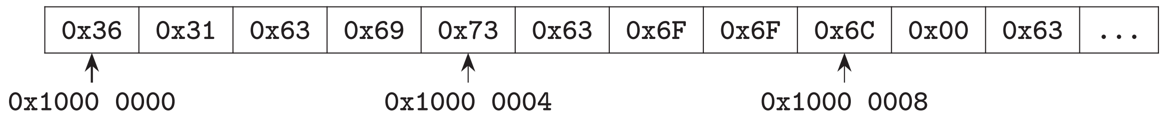
○ (A) Code ○ (B) Static ○ (C) Stack [✓ (D) Heap]

Solution: `dogs.names[1]` is on the heap as `add_pet` should make a copy of the `name` passed in on the heap and add it to the `names` list in the `Pets` struct.

For Q5.12-Q5.14, assume we have a **big-endian** system, and the code below has been run.

```
char course[] = {'6', '1', 'c'};
uint64_t *q = (uint64_t *) course;
uint32_t *p = (uint32_t *) q;
```

`course` is located at address `0x1000 0000`. Memory starting at `0x1000 0000` is shown below:



Q5.12 (1.5 points) What is the value of `strlen(course)`?

○ (A) 3
○ (B) 4
[✓ (C) 9]
○ (D) 10
○ (E) 11
○ (F) 12
○ (G) None of the above
○ (H) Compiler/runtime error

Solution: `strlen` reads memory one byte at a time until it hits a null terminator (`0x00`). In this case, there are 9 bytes (from address `0x1000 0000` in memory) before `0x00` is reached at address `0x1000 0009`.

Q5.13 (1.5 points) What is the value of `*q`?

- ○ (A) `0x3613 3696 3736 F6F6`
- ● (B) `0x3631 6369 7363 6F6F`
- ○ (C) `0x6963 3136 6F6F 6373`
- ○ (D) `0x6F6F 6373 6963 3136`
- ○ (E) `0x7363 6F6F 3631 6369`
- ○ (F) None of the above

Solution: Evaluate the sequence of 8 bytes stored in address `0x1000 0000` to `0x1000 0007` in big-endian.

Q5.14 (1.5 points) What is the value of `p + 1`?

- ○ (A) `0x3163 6973`
- ○ (B) `0x6F6F 6373`
- ○ (C) `0x7363 6F6F`
- ○ (D) `0x1000 0001`
- ● (E) `0x1000 0004`
- ○ (F) None of the above

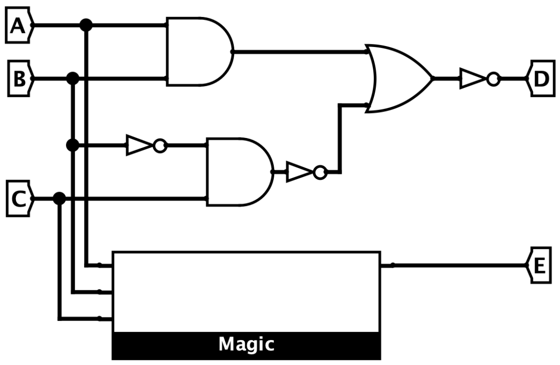
Solution: p + 1 evaluates to p + sizeof(p) = 0x1000 0000 + 4 = 0x1000 0004

This page intentionally left (mostly) blank.

The exam continues on the next page.

Q6 *Logical Logisim* (12 points)

Q6.1 (2 points) Given the following circuit with three inputs (A, B, and C), fill in the truth table for output D. You may ignore the Magic subcircuit and output E for Q6.1.



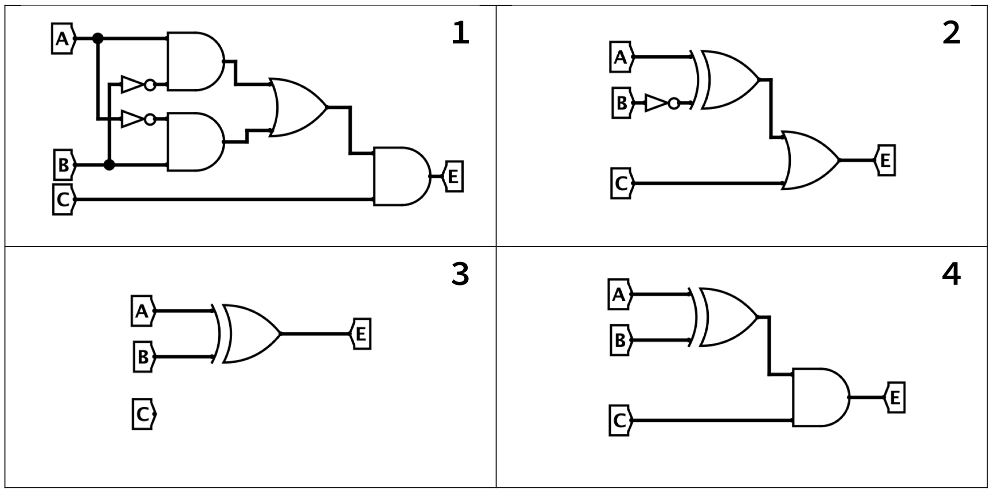
A	B	C	D	E
0	0	0	0	0
0	0	1	1	0
0	1	0	0	0
0	1	1	0	1
1	0	0	0	0
1	0	1	1	0
1	1	0	0	1
1	1	1	0	1

Q6.2 (3 points) Referencing the truth table and circuit above, write a boolean algebra expression in terms of A, B, and C that is equivalent to the behavior of the "Magic" subcircuit (i.e. output E). For full credit, you may use at most 2 operators. You may **only** use NOT (~), AND (&), OR (|), and each count as one operator. We will assume standard C operator precedence, so use parentheses when uncertain.

Solution: `B & (C | A)`

Q6.3 (3 points) Select all circuits that are logically equivalent to the following expression. If you select "None of the above," you cannot select other options.

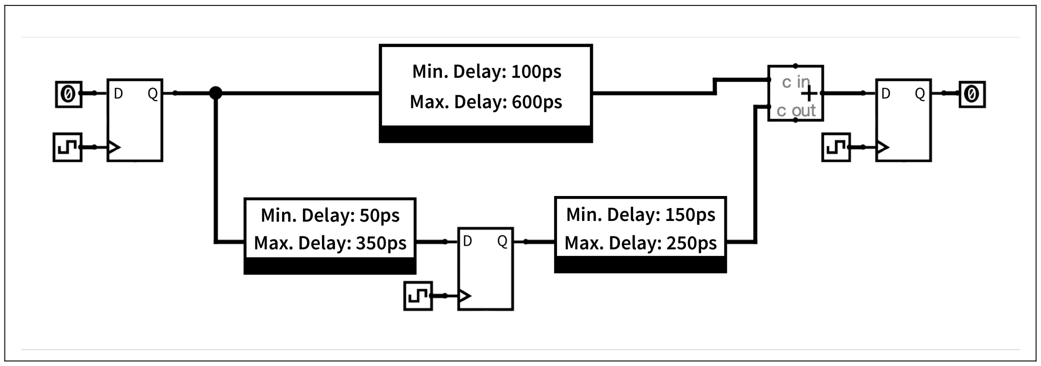
```
E = (A & ~B & C) | (~A & B & C)
```



■ (A) 1 ☐ (B) 2 ☐ (C) 3 ■ (D) 4

○ (E) None of the above

Consider the following SDS circuit and combinational delays. The adder block has a delay of 50ps. Each register has a clk-to-q delay of 20ps and a setup time of 30ps. You may assume registers are triggered on the rising edge and there is only one clock signal.



Q6.4 (2 points) What is the minimum allowable clock period?

Solution: 700 ps ($t_{\text{clk-to-q}} + t_{\text{top CL}} + t_{\text{adder}} + t_{\text{setup}} = 20 + 600 + 50 + 30 = 700$)

Q6.5 (2 points) What is the maximum hold time for our registers in order for the circuit to have well-defined behavior?

Solution: 20 + 50 = 70 ps ($t_{\text{clk-to-q}} + t_{\text{shortest CL}} = 20 + 50 \geq t_{\text{hold}}$)

Q7 *Datapathology* (16 points)

For this question, refer to the RISC-V Single Cycle Datapath from the CS 61C Reference Card.

For each of the following instructions, select the proper control signals and indicate which datapath components are used. If you select "None," you cannot select other options.

`bgeu t4 t5 end` (assume branch is taken)

Q7.1 (1 point) PCSel

(A) PC + 4 [✓ (B) ALU] (C) Don't Care

Q7.2 (1 point) ASel

[✓ (A) PC] (B) RegReadData1 (C) Don't Care

Q7.3 (1 point) BSel

[✓ (A) Imm] (B) RegReadData2 (C) Don't Care

Q7.4 (1 point) WBSel

(A) PC + 4 (B) ALU (C) Mem [✓ (D) Don't Care]

Q7.5 (1 point) Datapath Components

[✓ (A) Branch Comp] [✓ (B) Imm Gen] (C) DMEM (D) None

`auipc a0, 0x61C`

Q7.6 (1 point) PCSel

[✓ (A) PC + 4] (B) ALU (C) Don't Care

Q7.7 (1 point) ASel

[✓ (A) PC] (B) RegReadData1 (C) Don't Care

Q7.8 (1 point) BSel

[✓ (A) Imm] (B) RegReadData2 (C) Don't Care

Q7.9 (1 point) WBSel

(A) PC + 4 [✓ (B) ALU] (C) Mem (D) Don't Care

Q7.10 (1 point) Datapath Components

(A) Branch Comp [✓ (B) Imm Gen] (C) DMEM (D) None

Oh no! Jedi dropped your CPU and some of your datapath components are broken. You need to figure out which types of instructions are still guaranteed to function as expected. You may ignore `ebreak` and `ecall`. If you select "None," you cannot select other options.

Example: You should only select "I-type" if all I-type instructions are guaranteed to function as expected.

Q7.11 (2 points) PCSel is always "ALU"

- ☐ (A) R-Type
- ☐ (B) I-Type
- ☐ (C) S-Type
- ☐ (D) B-Type
- ☐ (E) U-Type
- [✓ (F) J-Type]
- ○ (G) None

Solution: J-Type instructions always jump.

Q7.12 (2 points) ASel is always "PC"

- ☐ (A) R-Type
- ☐ (B) I-Type
- ☐ (C) S-Type
- [✓ (D) B-Type]
- [✓ (E) U-Type]
- [✓ (F) J-Type]
- ○ (G) None

Solution: B-Type and J-Type instructions all use the PC register to calculate the target PC address. AUIPC uses the PC register. LUI doesn't need the PC register, but still functions properly.

Q7.13 (2 points) ImmGen always outputs `0x00000000`

- [✓ (A) R-Type]
- ☐ (B) I-Type
- ☐ (C) S-Type
- ☐ (D) B-Type
- ☐ (E) U-Type
- ☐ (F) J-Type
- ○ (G) None

Solution: R-Type instructions don't use ImmGen

Q8 *Coloring Book* (0 points)

These questions will not be assigned credit; feel free to leave them blank.

Q8.1 (0 points) What does the `FCVT.S.WU` instruction stand for?

> **Solution:** Floating-Point Convert to Single-Precision from Unsigned Word. This is the RISC-V floating-point extension's name for the operation that `convert` in question 4 implements!

Q8.2 (0 points) What does the `CVTSI2SS` instruction stand for?

> **Solution:** Convert Doubleword Integer to Scalar Single Precision Floating-Point Value. This is the equivalent instruction in x86 assembly.

Q8.3 (0 points) Which lecture contains a hidden animal and what was its species?

> **Solution:** Lecture 18, an adder is in a skipped slide.

Q8.4 (0 points) If there's anything else you want us to know, or you feel like there was an ambiguity in the exam, please put it in the box below.

For ambiguities, you must qualify your answer and provide an answer for both interpretations. For example, "if the question is asking about A, then my answer is X, but if the question is asking about B, then my answer is Y". You will only receive credit if it is a genuine ambiguity and both of your answers are correct. We will only look at ambiguities if you request a regrade.

> **Solution:** ¯_(ツ)_/¯