# CS61C F20 Final


### Instructors: Dan Garcia, Borivje Nikolic

### Head TAs: Stephan Kaminsky, Cece McMahon


### Question Breakdown

**Zone 1-1: Quest Clobber (*10 pts*, *30 minutes*)**
**Zone 2-1: CALL (*2 pts*, *90 minutes total for Zone 2*)**
**Zone 2-2: Floating Point (*5 pts*, *90 minutes total for Zone 2*)**
**Zone 2-3: Circuit to Expression (*6 pts*, *90 minutes total for Zone 2*)**
**Zone 2-4: Datapath (*6 pts*, *90 minutes total for Zone 2*)**
**Zone 2-5: RISC-V (*11 pts*, *90 minutes total for Zone 2*)**
**Zone 3-1: VM/Caches (*9 pts*, *60 minutes total for Zone 3*)**
**Zone 3-2: Parallelism (*8 pts*, *60 minutes total for Zone 3*)**
**Zone 3-3: Potpourri (*3 pts*, *60 minutes total for Zone 3*)**

## Section 1: Quest Clobber

### 1: Quest Clobber (*10 pts*)

**Part A — 2 pts** Recall that an 8-bit bias-encoded number normally has a bias of -127 so that roughly half the numbers are negative and half are positive, but there's one more positive than negative number. Using an equivalent scheme for choosing the bias, what base 14 number XXXXXX represents 0? (That is, your answer needs to have 6 base-14 characters.)

Variants: Different bases (all even, so the above process works), and different number of digits.

**Part B — 8 pts** We want to write a helper function to return a memory address aligned to $2^{20}$-byte boundaries. We also need to "return" the original malloced amount so we can free it later. We want to `malloc` the *fewest extra bytes possible*, and then do something to the pointer to align it. Fill in the code to complete it; don't worry about typecast warnings/errors (we removed casting for simplicity).

```
void *malloc_2totheN_aligned(size_t size, void /* CODE INPUT 1 */) {
    size_t offset = /* CODE INPUT 2 */;
    /* CODE INPUT 3 */ = malloc(size + offset); //smallest possible
    return ((/* CODE INPUT 4 */ + offset) /* CODE INPUT 5 */ /* CODE INPUT 6 */); //align
}

int main(int argc, char *argv[]) {
    void *head, *aligned;
    aligned = malloc_2totheN_aligned(59, /* CODE INPUT 7 */
    printf("head = 0x%p\n",head); // %p is what we use to print out a pointer
    printf("aligned = 0x%p\n",aligned);
    free(head);
}

unix% a.out
head    = 0x12345678
aligned = 0x[HEX INPUT HERE]
```

Variants: Different alignment requirement. This changed the exponent used, and changed how many bits turn into zeros in the hex input.

## Section 2: Midterm Clobber (*30 pts*)

### 1: CALL (*2 pts*)

Variants: Students were given a random set of eight of the following parts, and were asked to determine which part of CALL bests matches the statements.

Choose which of **Compiler, Assembler, Linker, Loader** best matches each statement.

- Its input may contain pseudoinstructions.
- It copies instructions into its address space. text and data segments.
- Its job is to read the relocation table.
- Its input is executable code.
- One of its jobs is to take the text segments from .o files and concatenate them together.
- Its output is true assembly only.
- It reads directives.
- Output file suffix is `.o`.
- Its input is object code files (among other things).
- It sets the PC.
- This job is typically part of the operating system.
- One of its jobs is to take the code segments from .o files and concatenate them together.
- It creates a new address space for the program large enough to hold - Input file suffix is `.s`.
- It deals with the syntax of C.
- Its output is an object file.
- Its output is machine language.
- Input file suffix is `.c`.
- This stage makes two passes over its input.
- Input file suffix is `.o`.
- Typical input file is a.out.
- Its output may contain pseudoinstructions.
- Its input is information tables (among other things).
- It is often called the 'bottleneck' of the development process.
- Its output is a fully running program.
- Typical output file is `a.out`.
- It deals with the semantics (i.e. meaning) of C.
- Its job is to resolve references and fill in the absolute addresses.
- It replaces pseudoinstructions.
- Output file suffix is `.s`.
- Its output is two information tables.
- People sometimes do this stage by hand for optimization.
- Its output is an executable.
- It initializes machine registers.

### 2: Floating Point (*3 pts*)

Consider a `NON-WORD-LENGTH`-bit floating point number with the following components: 1 sign bit, 13 exponent bits, and 19 mantissa/significand bits structured otherwise in IEEE754 Floating Point standard format.

$$SEEEEEEEEEEEEMMMMMMMMMMMMMMMMMMM$$

All other properties of IEEE754 apply (bias, denormalised numbers, $\infty$, NaNs, etc…). The bias is the usual $-(2^{E-1}-1)$, where here would be $-4095$.

Recall the median of a set of numbers is the "middle number", if you sorted them.

E.g., the median of {1, 2, 5, 100, 1000} is 5.

**Part A — 3 pts** What is the median of the positive non-NaN floats? (including +0, denorms, and $\infty$, which is an odd number of numbers) Write your answer as a decimal number, like 7.65; something a first grader could understand.

**Part B — 2 pts** Of all the numbers you can represent with this floating-point format, what is the largest odd number?

Different versions had different numbers of exponent and mantissa bits.

**3: SDS (*6 pts*)**

**Part A *Circuit to Expression* — 3 pts**
Please choose the correct boolean expression for the circuit given. NOTE: see all versioned diagrams in the Appendix.

**Part B *FSM Blinding Lights* — 3 pts** You fall in love with a new pop hit, *Blinding Lights*. To make your own blinking lights, come up with a finite state machine that turns static input into "blinking" outputs. Based on two examples of input and output boolean sequences, **fill in the state transition table for the finite state machine below**.

*Version 1*
`Input 1: 111111111111111`
`Output 1: 001001001001001`
`Input 2: 001011011101111`
`Output 2: 000000000100010`

| current state | input | next state | output |
|---|---|---|---|
| 00 | 0 | YOUR ANS HERE | YOUR ANS HERE |
| 00 | 1 | YOUR ANS HERE | YOUR ANS HERE |
| 01 | 0 | YOUR ANS HERE | YOUR ANS HERE |
| 01 | 1 | YOUR ANS HERE | YOUR ANS HERE |
| 10 | 0 | YOUR ANS HERE | YOUR ANS HERE |
| 10 | 1 | YOUR ANS HERE | YOUR ANS HERE |

*Version 2*
`Input 1: 111111111111111`
`Output 1: 0100100100100100`
`Input 2: 001011011101111`
`Output 2: 000001001000100`

| current state | input | next state | output |
|---|---|---|---|
| 00 | 0 | YOUR ANS HERE | YOUR ANS HERE |
| 00 | 1 | YOUR ANS HERE | YOUR ANS HERE |
| 01 | 0 | YOUR ANS HERE | YOUR ANS HERE |
| 01 | 1 | YOUR ANS HERE | YOUR ANS HERE |
| 10 | 0 | YOUR ANS HERE | YOUR ANS HERE |
| 10 | 1 | YOUR ANS HERE | YOUR ANS HERE |

*Version 3*
`Input 1: 111111111111111`
`Output 1: 0110110110110110`
`Input 2: 001011011101111`
`Output 2: 000001001100110`

| current state | input | next state | output |
|---|---|---|---|
| 00 | 0 | YOUR ANS HERE | YOUR ANS HERE |
| 00 | 1 | YOUR ANS HERE | YOUR ANS HERE |
| 01 | 0 | YOUR ANS HERE | YOUR ANS HERE |
| 01 | 1 | YOUR ANS HERE | YOUR ANS HERE |
| 10 | 0 | YOUR ANS HERE | YOUR ANS HERE |
| 10 | 1 | YOUR ANS HERE | YOUR ANS HERE |

*Version 4*
`Input 1: 111111111111111`

Output 1: 100100100100100
Input 2: 001011011101111
Output 2: 001010010001001

| current state | input | next state | output |
|---|---|---|---|
| 00 | 0 | YOUR ANS HERE | YOUR ANS HERE |
| 00 | 1 | YOUR ANS HERE | YOUR ANS HERE |
| 01 | 0 | YOUR ANS HERE | YOUR ANS HERE |
| 01 | 1 | YOUR ANS HERE | YOUR ANS HERE |
| 10 | 0 | YOUR ANS HERE | YOUR ANS HERE |
| 10 | 1 | YOUR ANS HERE | YOUR ANS HERE |

*Version 5*
Input 1: 111111111111111
Output 1: 101101101101101
Input 2: 001011011101111
Output 2: 001010010101011

| current state | input | next state | output |
|---|---|---|---|
| 00 | 0 | YOUR ANS HERE | YOUR ANS HERE |
| 00 | 1 | YOUR ANS HERE | YOUR ANS HERE |
| 01 | 0 | YOUR ANS HERE | YOUR ANS HERE |
| 01 | 1 | YOUR ANS HERE | YOUR ANS HERE |
| 10 | 0 | YOUR ANS HERE | YOUR ANS HERE |
| 10 | 1 | YOUR ANS HERE | YOUR ANS HERE |

*Version 6*
Input 1: 111111111111111
Output 1: 1101101101101110
Input 2: 001011011101111
Output 2: 001011011001101

| current state | input | next state | output |
|---|---|---|---|
| 00 | 0 | YOUR ANS HERE | YOUR ANS HERE |
| 00 | 1 | YOUR ANS HERE | YOUR ANS HERE |
| 01 | 0 | YOUR ANS HERE | YOUR ANS HERE |
| 01 | 1 | YOUR ANS HERE | YOUR ANS HERE |
| 10 | 0 | YOUR ANS HERE | YOUR ANS HERE |
| 10 | 1 | YOUR ANS HERE | YOUR ANS HERE |

Versions had the same path, but different outputs for state transitions.

### 4: Datapath, Control, and Pipelining (*6 pts*)

Consider a standard single-cycle datapath that implements the RV32I instruction set. It also implements all three store instructions (`sw`, `sh`, `sb`), and all three load instructions (`lw`, `lh`, `lb`). As in Project 3B, the load and store instructions are considered to be valid when they **do not exceed the boundaries of a contiguous word in memory**. In other words, all valid `sw`s and `lw`s must be word-aligned and `sh`s and `lh`s must be within a single word. Memory is 32-b wide.

**Part A — 5 pts** Your task is to derive the logic that throws an `**InvalidMemoryAccess**` exception (returns `True`) if and only if the input instruction is a sw instruction, and the memory access is not valid per the description above. `Addr[31:0]` is the ALU output that addresses the data memory and `Instr[31:0]` is the current instruction. Write the Boolean logic expression that implements `**InvalidMemoryAccess**`. Format your answer using instruction bits and

boolean operators.

*Version 1*—`sw`
*Version 2*—`sh`
*Version 3*—`lw`
*Version 4*—`lh`

**Part B — 1 pt** If the datapath is implemented as a 5-stage pipeline, what is the earliest phase of execution in which the exception can be detected?

Versions: Different versions had different instruction requests; this affected the required opcode/func3 and the addresses which would cause an invalid memory access.

### 5: RISC-V (*11 pts*)

`mystery(a0)` is a stream-like procedure that returns `a0 + (the number of times the function has been run so far)`.

For example (in Python notation):

```
>>> mystery(100)
100 # The function has been run 0 times before, so we return 100+0 = 100
>>> mystery(100)
101 # The function has been run 0 times before, so we return 100+1 = 101
>>> mystery(1000)
1002 # The function has been run 2 times before, so we return 1000+2 = 1002
```

**Part A *Self-modifying RISC-V* — 7 pts** **Fill in the blanks to complete the function `mystery`.** Do **NOT** include any part of the line that's already written. Your code should work as many times as possible. Assume that there is no memory protection (so you can modify any addresses), the system is little-endian, and that you can only load and store within single words (the same restriction as stores and loads in Project 3). **Do not include commas or uppercase symbols, and write out all immediates.** For example, if your answer to a code input is `lb a0 10(a0)`, the answers `lb a0, 0xa(a0)` and `lb a0 0x10(a0)` would be marked as incorrect! Additionally, if a line of code is partially filled in, only include the missing component. For example, `CODE INPUT 3` should not include the `addi t1 t1`.

```
mystery:    addi a0 a0 0
        la t0 # CODE INPUT 1
        # CODE INPUT 2
        addi t1 t1 # CODE INPUT 3
        # CODE INPUT 4
        ret
```

Versions: Different t registers were used.

**Part B — 2 pts** **How many times can you call `mystery` before it stops working** (according to our definition)?

**Part C — 2 pts** You run `mystery(0)` one more time after having it work perfectly for the number of times you call it in part (b). **What is the return value (in hex)?** Do **NOT** prefix your solution with `0x`. Please pad your answer to a full 4 bytes when submitting if necessary. **Hex digits which are letters can be written in either upper- or lowercase. For example, if the answer is `0x0000BEEF`, the answers `0000BEEF` and `0000beef` will be accepted. Submitting `0x0000BEEF` or `BEEF` will result in your answer being marked as incorrect**!

## Section 3: Post-Midterm (*20 pts*)

### 1: VM and Caches (*9 pts*)

We are given a system with physical memory of size **4 GiB** and virtual memory of **16 GiB** and a page size of **8 KiB**. Our page table system has one level with no TLB.

***Part A* — *2 pts***

1. How many bits are in the page offset?
2. How many bits are in the VPN?
3. How many bits are in the physical address?
4. How many bits are in the PPN?

Versions: Different versions had different physical/virtual memory sizes, and different page sizes.

**Part B — 6 pts** Let's consider cache design, independent of the VM system. Our system now has a large direct-mapped cache that starts cold.

We run the following code:

```
#define MAX = 192
uint32_t A[MAX];
uint32_t dummy, i; /*stored in registers*/
for (uint32_t i = 0; i < MAX; i++) {
    dummy = A[i];
}
```

1. What is the miss rate for the cache having a blocksize = 64 B?
2. We add an L2 cache. After running a new program, we measure the following system parameters:
   - **L1 cache reference:** Hit time is 2 B clock cycles and hit rate is 0.8 B
   - **L2 cache reference:** Hit time is 10 B clock cycles and hit rate is 0.98 B
   - **Memory reference:** Hit time is 1000 B clock cycles

Calculate the AMAT of this system in terms of CPU clock cycles.

Versions: Different versions changed the block size, array size, and cache hit times and hit rates.

**Part C — 1 pt** Which components of the system are accessed during a context switch? Select all that apply.

- Register File
- Main Memory
- Virtual Memory
- PTBR
- Active Page Tables
- Inactive Page Tables
- Disk
- PC
- TLB

### Parallelism (*8 pts*)

**Part A *Amdahl's Law* — 1 pt** How much does 99% of a program need to be sped up by such that the overall program is sped up by 75x?

Versions: Different numbers were selected for the section to be sped up, and the required speedup.

**Part B *TLP* — 3 pts** What are the smallest and largest final values of **a** if the code were running on two concurrent hardware threads?

*Sample Version*

```
int a = 2
#pragma omp parallel
{
    a *= 2
    a += 1
}
```

Versions: Different versions had different values to multiply/add, and different starting points for a.

**Part C *DLP* — 1 pt** Select all that is true about DLP.

- A major bottleneck for DLP speedup is due to register size.
- It is useful for control-dependent code.
- MIMD falls under the category of DLP.
- It always performs best when the data is repetitive.
- It can handle multiple streams of instructions.
- MISD falls under the category of DLP.
- The central idea is scalar loading and calculation.
- A major bottleneck for DLP speedup is due to register speed.
- Intrinsics used for DLP calculations are architecture-dependent.
- DLP code can detect the end of a data stream.
- Unaligned instructions perform better than aligned instructions overall (`storeu`/`loadu` vs `store`/`load`).
- Vectors are indexable.
- It performs best when the task is repetitive.

**Part D *MapReduce/Spark* — 3 pts** We want to analyze a corpus of text (think of it as a large bucket of lowercase words, with no punctuation) and return a list of tuples in which the "keys" (i.e., tuple first elements) are **every last letter encountered**, and the "values" (i.e., tuple second elements) are **the unique words that end with that letter**. Select the operators and fill in the blanks of the code below.

```
unix% **cat words.txt**
this is the best course in the world
unix% **pyspark**
Welcome to Spark version 3
>>> **{"cal"[0], "cal"[-1], len("cal")}.union({3,4})** ## Just a refresher
{3, 'c', 4, 'l'} ## note 3 is not listed twice since sets elements are unique
>>> **W = sc.textFile("words.txt")**
>>> **Fab = lambda a, b: CODE_INPUT_1
>>> Fline = lambda line: line.split()
>>> Fw = lambda w: (CODE_INPUT_2)
>>> W.CODE_INPUT_3(CODE_INPUT_4).CODE_INPUT_5(CODE_INPUT_6).CODE_INPUT_7(CODE_INPUT_8).collect()

[('s', {'this', 'is'}), ('d', {'world'}), ('e', {'course', 'the'}), ('t', {'best'}), ('n', {'in'})]
```

**Potpourri (*3 pts*)**

**Part A *Dependability RAID* — 1 pt** Select all versions of RAID for which `VERSION`:

*Versions*

1. "the following statement is true: redundancy through parity."
2. "a pro is having a small overhead."
3. "fast small reads are a pro."
4. "fast small writes are a pro."
5. "higher throughput is a pro."
6. "parity is bit-striped."
7. "parity is byte-striped."
8. "parity is block-level striped."
9. "disks play a part in increasing reliability."

**Part B *ECC* — 1 pt** Given the following bitstring `0b010011000`, calculate the encoded bitstring to perform single-bit error correction with odd parity. Give your final answer in binary with the fewest number of bits needed to recover the

given correct bitstring should a 1-bit error occur in the given bitstring without the `0b` prefix. For example, if your answer is `0b101000101`, you will only receive credit for writing `101000101` and **NOT** `0b101000101`, `0b0000101000101`, and similar answers.

Versions: Different versions had different bitstring lengths and patterns as well as varied parity.

**Part C *OS/IO* — 1 pt** *Version 1*

Which of the following is true about the operating system?

- The OS is considered more trusted than the user level.
- It can be responsible for VA to PA translation.
- The user can request the OS to do certain operations through syscalls.
- The OS is usually trusted.
- The OS is not the first program that runs when a device starts up.
- The OS uses the same virtual memory space as the user does.
- OSes are not responsible for protecting different processes as it's part of the process's job to know what is accessible and what is not.
- The OS has access to the same physical memory space as the user does.
- The role of the OS is to load, run, manage, and combine multiple programs optimally.

*Version 2*

What operation(s) can you use to continuously poll? Assume each choice is the sole instruction responsible for changing code flow in polling code.

- `bne`
- `blt`
- `jr`
- `beq`
- `jal`
- `neg`
- `bge`
- `j`

*Version 3*

What is true about polling and what are benefits of using it over interrupts? Select all that apply.

- Always causes overall lower latency.
- Deterministic response time per event.
- Has lower overhead when data is unavailable.
- Can do other tasks simultaneously.
- Used to send checksum packets.
- Allows CPU to sleep.
- Expensive when there's lots of IO.
- Allows for "always-busy" devices to continuously send input.
- Once data starts coming in, polling always stops and the system switches to DMA (direct memory access).

*Version 4*

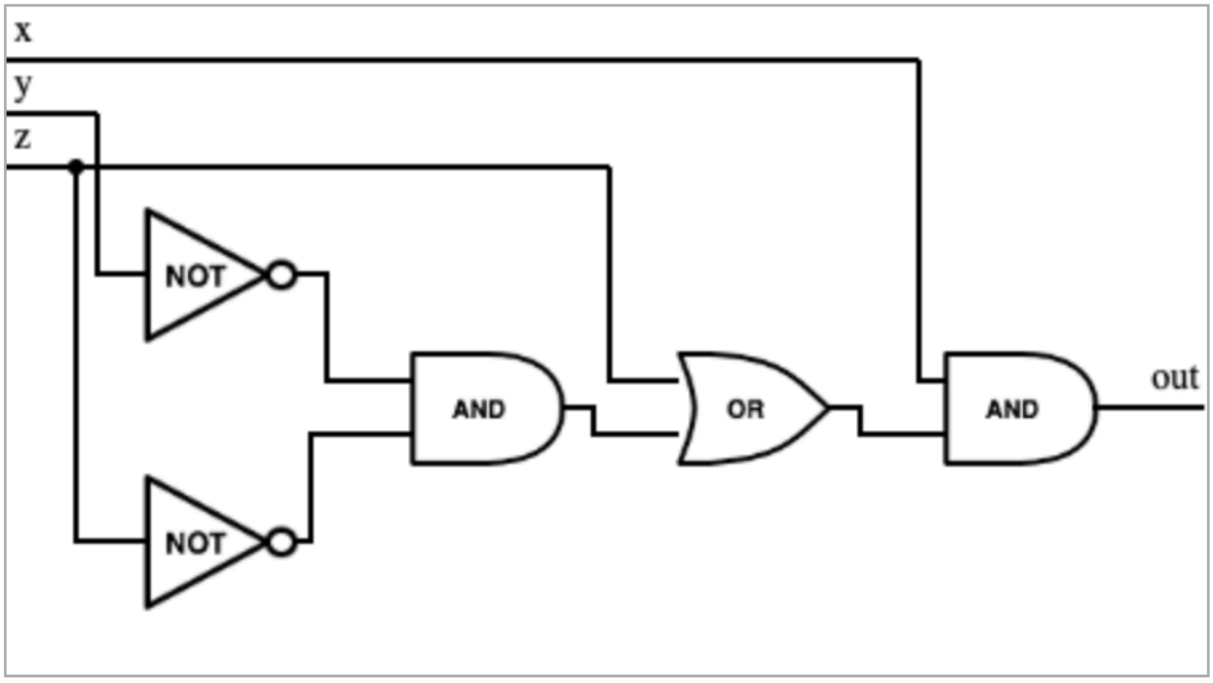
What is true about interrupts and what are benefits of using it over polling? Select all that apply.

- Preferred for devices that require less frequent servicing by the CPU.
- Frees up CPU for other events when no event is in progress.
- CPU automatically checks for status of input from device.
- Allows for a deterministic response time.
- Higher latency per event.
- No additional overhead time.
- Always has better throughput.
- Requires the device to signal for the CPU to handle input.
- Sets control and data registers.

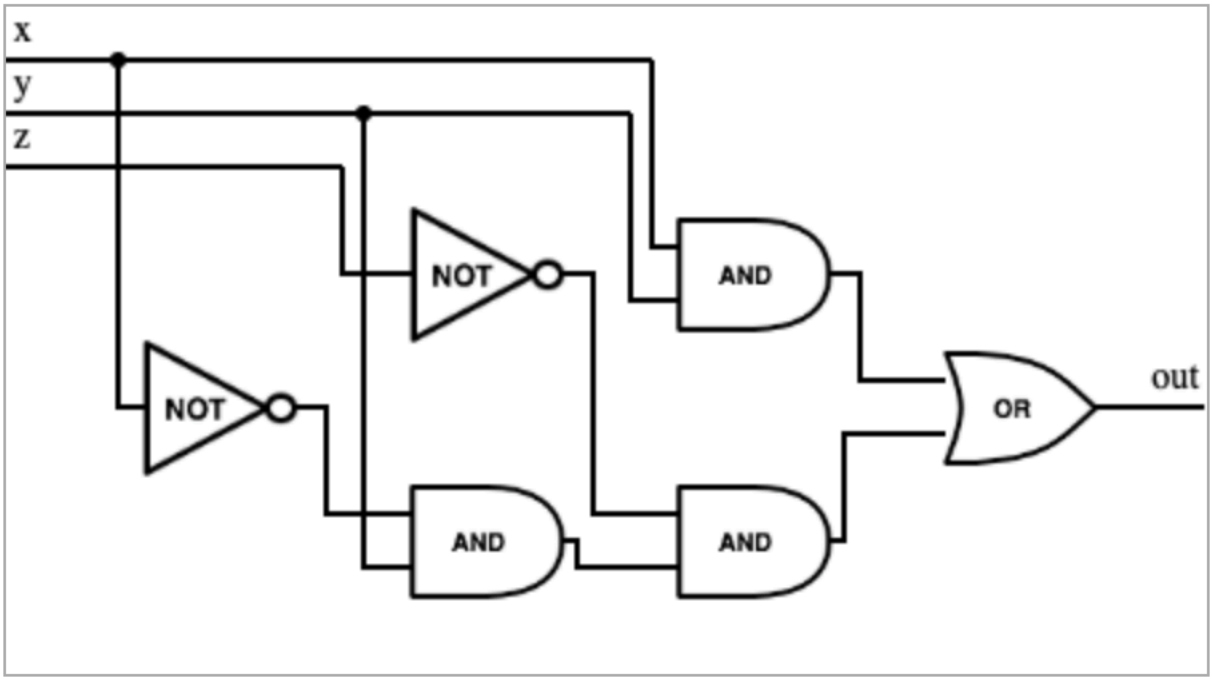
# Appendix: Relevant Diagrams

## Zone 2-3: Midterm Clobber — SDS

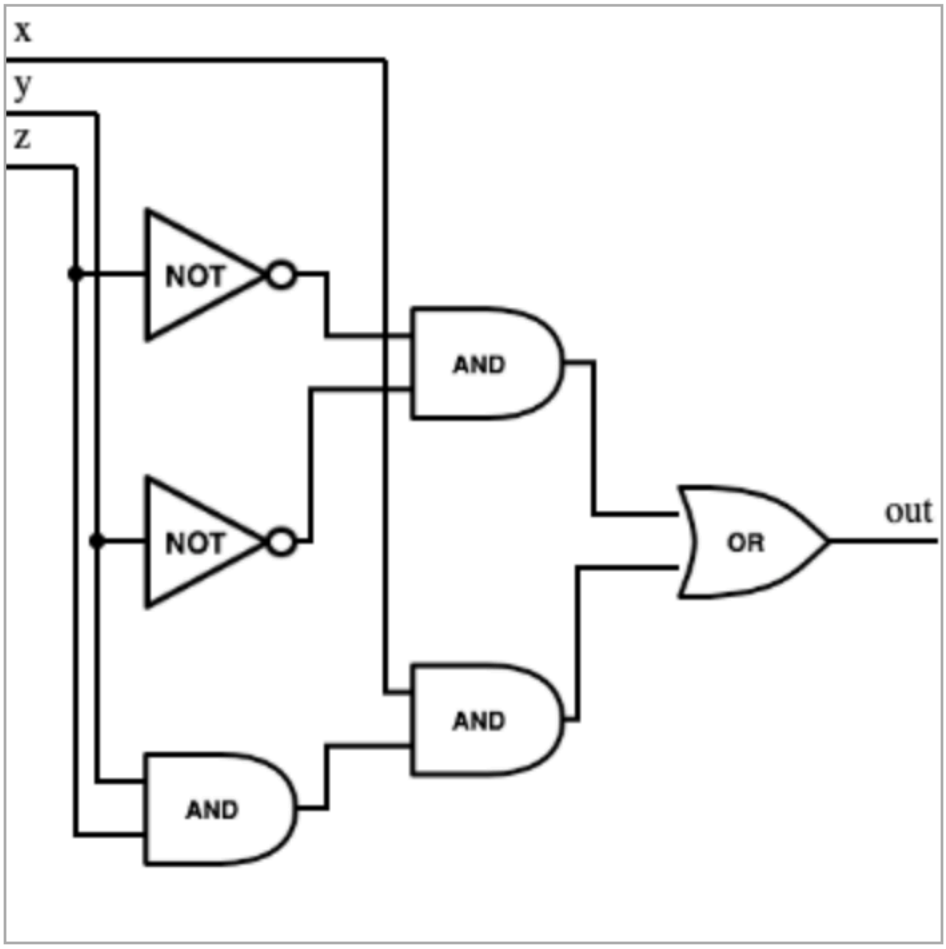
### Version 1 Logic Gates



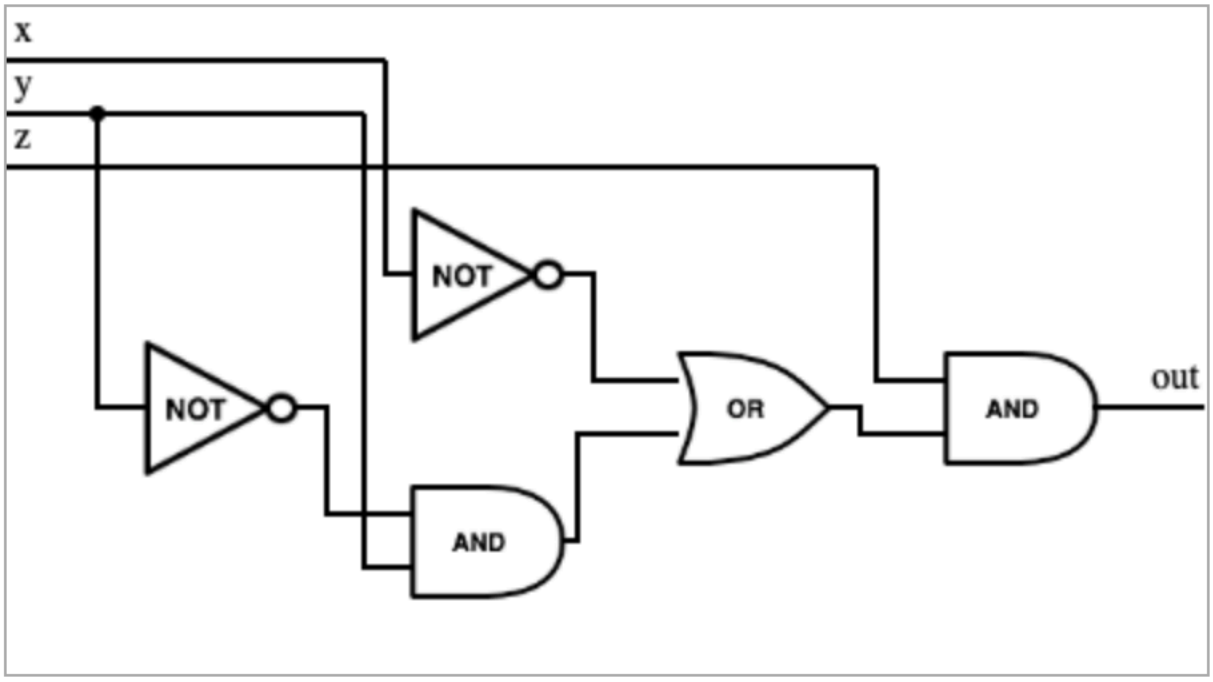

### Version 2 Logic Gates



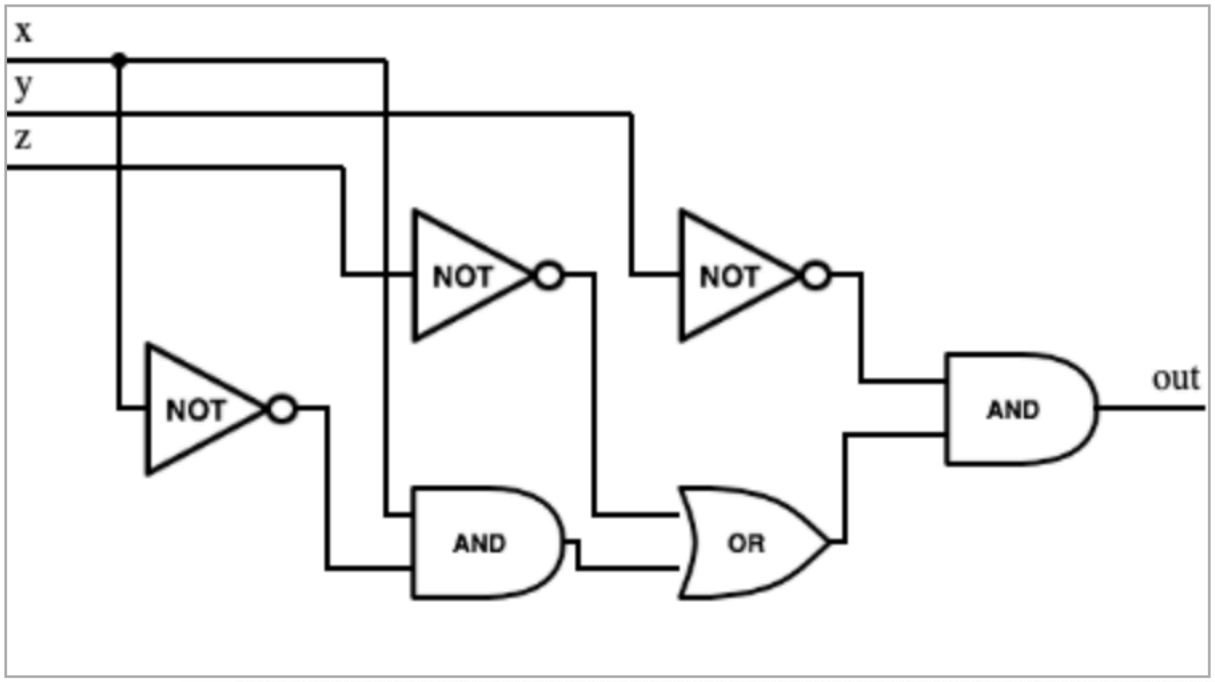
**Version 3 Logic Gates**



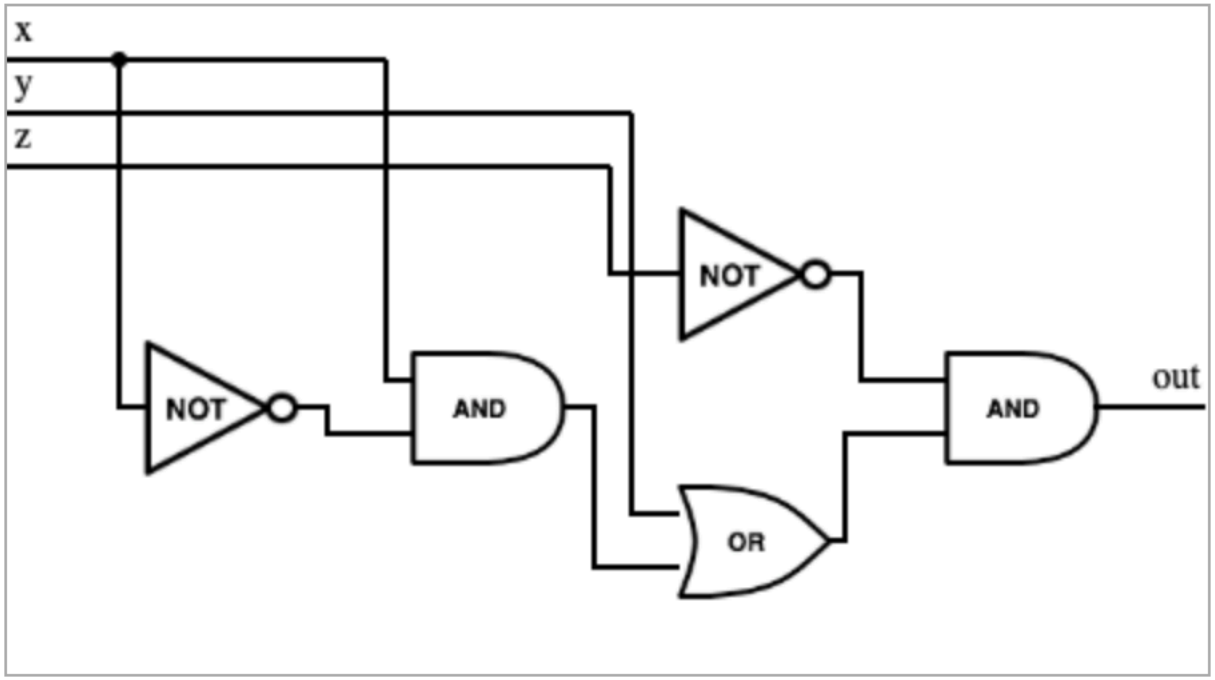

**Version 4 Logic Gates**



**Version 5 Logic Gates**




**Version 6 Logic Gates**



**Blinding Lights FSM**