CS61C F20 Final Solutions

Instructors: Dan Garcia, Borivje Nikolic

Head TAs: Stephan Kaminsky, Cece McMahon

If you believe there are any mistakes, please let staff know ASAP! All diagrams are located at the end of the PDF for reference's sake.

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.)

Solutions

Answer: 6DDDDD. For now, let's pretend that 0 is negative; that way, we want exactly half the numbers to be negative, and half to be positive, and all numbers are positive or negative. If we do that, then we want the cutoff from negative to positive to be right in the middle of all possible numbers; this is between the numbers 6DDDDD and 700000. Our 0 would be the number right before this cutoff, and is thus 6DDDDD.

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]
```

Solutions

`CODE INPUT 1: **head`

`CODE INPUT 2: 1 << 20 - 1`

`CODE INPUT 3: *head`

`CODE INPUT 4: *head`

`CODE INPUT 5: &`

`CODE INPUT 6: ~(1 << 20 - 1)`

`CODE INPUT 7: &head`

`[HEX INPUT HERE]: 12400000`

In order for this code to work, we need at least one contiguous group of size bits, starting at an aligned address. We don't have any control over our initial head position, so we need to add an extra few bytes to ensure our buffer contains such a block. The specified amount makes it so that exactly one such buffer is guaranteed to exist; we then shift our return value to exactly the right spot. We need to send in a double pointer so that we modify a pointer outside the program; if we had a single pointer, the change we make to head in the function would not appear outside the function, because C is pass-by-value.

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.

Solutions

Compiler

- Its output may contain pseudoinstructions.
- People sometimes do this stage by hand for optimization.
- It deals with the syntax of C.
- It deals with the semantics (i.e. meaning) of C.
- Input file suffix is `.c`.
- Output file suffix is `.s`.

Assembler

- Its output is two information tables.
- Its input may contain pseudoinstructions.
- Its output is true assembly only.
- It reads directives.
- It replaces pseudoinstructions.
- Its output is an object file.
- Its output is machine language.
- This stage makes two passes over its input.
- Input file suffix is `.s`.
- Output file suffix is `.o`.

Linker

- Its input is object code files (among other things).
- Its input is information tables (among other things).
- Its output is an executable.
- One of its jobs is to take the text segments from .o files and concatenate them together.
- One of its jobs is to take the code segments from .o files and concatenate them together.
- Input file suffix is `.o`.
- Typical output file is `a.out`.
- It is often called the 'bottleneck' of the development process.
- Its job is to read the relocation table.
- Its job is to resolve references and fill in the absolute addresses.

Loader

- Its input is executable code.
- Its output is a fully running program.
- Typical input file is a.out.
- This job is typically part of the operating system.
- It creates a new address space for the program large enough to hold text and data segments.
- It copies instructions into its address space.
- It initializes machine registers.
- It sets the PC.

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.

$$SEEEEEEEEEEEEEMMMMMMMMMMMMMMMMMMM$$

All other properties of IEEE754 apply (bias, denormalised numbers, ∞, 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 ∞, which is an odd number of numbers) Write your answer as a decimal number, like 7.65; something a first grader could understand.

Solutions

1.5

Explanation

One useful thing to note is that floating point numbers maintain the order of sign-magnitude numbers; for example, 0x00000000 < 0x00000001 < 0x00000002 < 0x0FFFFFFF< 0x10000000 < 0x10000001, if we interpret the hexadecimal as an IEEE standard floating point number. We can thus find the median by looking at the bit patterns, finding the median of a sign-magnitude number, and converting that to floating point. The smallest number is 0x000000000, and the largest number is 0x0FFF80000 (+∞). The median of these two numbers (treating them as sign-magnitude numbers) is (0x000000000+0x0FFF80000)/2 = 0x07FFC000, which we convert to floating point as 1.5.

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

Solutions

`2**(20) - 1`

Explanation

An odd number has a "1" in the ones place, so we need the ones place within our mantissa. The largest number we can get that way would be if the last bit in our mantissa was the ones place, and every other bit in the mantissa was a one. This number is 2^20-1. We verify that we can have an exponent of +19, so this is possible.

Different versions had different numbers of exponent and mantissa bits. The exponent was always selected such that the mantissa was "longer", so the answer to part B would rely only on the mantissa length (exponent is significand + 1). The answer to part A was always 1.5, regardless of the mantissa and exponent length.

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.

Solutions

Version 1: `x(~y + z)`
Version 2: `y(~z + x)`
Version 3: `~(z + y) + xyz`
Version 4: `~xz`
Version 5: `~(z + y)`
Version 6: `~zy`

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**.

Solutions

Version 1

```
Input 1: 111111111111111
Output 1: 001001001001001
Input 2: 001011011101111
```

`Output 2: 000000000100010`

current state	input	next state	output
00	0	00	0
00	1	01	0
01	0	00	0
01	1	10	0
10	0	00	0
10	1	00	1

Version 2
`Input 1: 111111111111111`
`Output 1: 010010010010010`
`Input 2: 001011011101111`
`Output 2: 000001001000100`

current state	input	next state	output
00	0	00	0
00	1	01	0
01	0	00	0
01	1	10	1
10	0	00	0
10	1	00	0

Version 3
`Input 1: 111111111111111`
`Output 1: 011011011011011`
`Input 2: 001011011101111`
`Output 2: 000001001100110`

current state	input	next state	output
00	0	00	0
00	1	01	0
01	0	00	0
01	1	10	1
10	0	00	0
10	1	00	1

Version 4
`Input 1: 111111111111111`
`Output 1: 100100100100100`
`Input 2: 001011011101111`
`Output 2: 001010010001001`

current state	input	next state	output
00	0	00	0
00	1	01	1
01	0	00	0
01	1	10	0
10	0	00	0

current state	input	next state	output
10	1	00	0

Version 5

```
Input 1: 111111111111111
Output 1: 101101101101101
Input 2: 001011011101111
Output 2: 001010010101011
```

current state	input	next state	output
00	0	00	0
00	1	01	1
01	0	00	0
01	1	10	0
10	0	00	0
10	1	00	1

Version 6

```
Input 1: 111111111111111
Output 1: 110110110110110
Input 2: 001011011101111
Output 2: 001011011001101
```

current state	input	next state	output
00	0	00	0
00	1	01	1
01	0	00	0
01	1	10	1
10	0	00	0
10	1	00	0

Explanation

We note that when a constant input is received, the output repeats with a period of 3. That means that we need to find a loop of three states with arrows to each other, and that is the "path" taken by a constant input of ones. Fortunately, we find such a path in the diagram, and we can thus fill in the next state and output of each state for input 1. We then note that the other arrow must be for input 0, and also note that all remaining arrows go to state 00; in other words, if we see a 0, we go to state 00, regardless of our current state. We can thus use the second input stream to determine what output we expect on getting a 0 at each state.
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**`.

Solutions

Version 1—sw: `(!Instr[6] \xd7 Instr[5] \xd7 !Instr[4] \xd7 !Instr[3] \xd7 !Instr[2] \xd7 Instr[1] \xd7 Instr[0]) \xd7 (!Instr[14] \xd7 Instr[13] \xd7 !Instr[12]) \xd7 (Addr[1] + Addr[0])`
Version 2—sh: `(!Instr[6] \xd7 Instr[5] \xd7 !Instr[4] \xd7 !Instr[3] \xd7 !Instr[2] \xd7 Instr[1] \xd7 Instr[0]) \xd7 (!Instr[14] \xd7 !Instr[13] \xd7 Instr[12]) \xd7 (Addr[1] \xd7 Addr[0])`
Version 3—lw: `(!Instr[6] \xd7 !Instr[5] \xd7 !Instr[4] \xd7 !Instr[3] \xd7 !Instr[2] \xd7 Instr[1] \xd7 Instr[0]) \xd7 (!Instr[14] \xd7 Instr[13] \xd7 !Instr[12]) \xd7 (Addr[1] + Addr[0])`
Version 4—lh: `(!Instr[6] \xd7 !Instr[5] \xd7 !Instr[4] \xd7 !Instr[3] \xd7 !Instr[2] \xd7 Instr[1] \xd7 Instr[0]) \xd7 (!Instr[14] \xd7 !Instr[13] \xd7 Instr[12]) \xd7 (Addr[1] \xd7 Addr[0])`

Explanation

The first part and second part return true whenever we have the correct opcode and func3; this makes it so that the error is called only when we have the correct instruction. For word instructions, we need to make sure that our address has 0 for the bottom two bits, and for half-word instructions, we need to make sure that the bottom two bits aren't 11. We need to `\xd7` everything together, since an error is called if the opcode is sw AND the func3 matches sw AND the address is bad.

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?

Solutions
Execute

Explanation
Regardless of the version, the error relies on the current operation, and on the address accessed. We can find the operation by the ID stage, but we only know the address after we use the ALU to add in the immediate. As such, we need to wait for the end of the execution phase.
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
```

Solutions

```
CODE INPUT 1: mystery
CODE INPUT 2: lh t1 2(t0)
CODE INPUT 3: 16
CODE INPUT 4: sh t1 2(t0)
```

Completed Code:

```
mystery:    addi a0 a0 0
        la REG1 **mystery**
        **lh t1 2(t0)**
        addi t1 t1 **16**
        **sh t1 2(t0)**
        ret
```

Explanation
This question is quite tricky. In order for this to work, we need to modify our RISC-V code somehow to store the number of times the function is run; the best place to do that is in the immediate of the addi instruction (we want to make it so that the next time this is run, that line is `addi a0 a0 1`). In order to do this, we need to check our RISC-V green sheet to see which bits store the immediate on an `addi` instruction. We see that the `addi` instruction stores its immediate in `inst[20:31]`, so we need to load this chunk of data into our register, and modify it. We have two choices for this: `lh` and `lw` (plus unsigned versions, but since we `sh`/`sw` later, this doesn't affect our code). If we choose `lh`, then we need to increment our 4th bit, so we `addi 16` and store. If we choose `lw`, we would need to increment our 20th bit, which would be an `addi` of `1048576`. However, this is not possible with an `addi` instruction; an `addi` instruction only has 12 bits of immediate, so it can't contain an immediate value of 1 million. Therefore, we are forced to use `lh`.
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)?

Solutions

2048

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!**

Solutions

FFFFF800

Explanation
Our limit here is that `addi` instructions only have 12 bits of immediate, so there are only 2^12 possible immediates. However, `addi` instructions store their immediate in two's complement (otherwise negative immediates would be impossible); as such, their range in the positive direction is limited to +2047, which is used on the 2048th iteration of mystery. On the 2049th iteration of mystery, our immediate becomes -2048, so our answer to part c is -2048 = `0xFFFFF800`.

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.

NOTE: answers are given with respect to the sample version's parameters; some answers are given in general form as well.

Part A* — *2 pts
Please choose the correct answer for the following questions.

1. How many bits are in the page offset? `OFFSET_BITS` $= \log_2(PGSIZE) = 13$
2. How many bits are in the VPN? $\log_2(\frac{VIRTMEM}{PGSIZE}) =$ `VIRT_ADDR_BITS - OFFSET_BITS` $=$ 34-13 $= 21$
3. How many bits are in the physical address? $\log_2(PHYSMEM) = 32$
4. How many bits are in the PPN? `PHYS_ADDR_BITS` $= \log_2(\frac{PHYSMEM}{PGSIZE}) =$ `PHYS_ADDR_BITS - OFFSET_BITS` $=$ 32-13 $= 19$

Versions: Different versions had different physical/virtual memory sizes, and different page sizes. The effect on the answer is listed in the above questions.

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 = <span style="color:red">192</span>
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.

Solutions

1. $\frac{1}{16}$
2. `L1_REF + (L2_REF + MEM_REF * (1 - L2_HR)) * (1 - L1_HIT) = 8 cycles`

Explanation For part 1, we miss once every time we bring in a new block, and we access every word in that block exactly once. Because we never return to a block and the array size is larger than the block size, the cache size and the array size don't matter, and the miss rate is dependent only on the block size.
For part 2, we compute the hit rate. 100% of the time, we have to go through the L1 cache, so that adds 2 cycle. 20% of the time, we go through the L2 cache and spend an extra 10 cycles, so that adds 2 cycles on average. 20% * 2% of the time we miss on the L2 cache, and spend an extra 1000 cycles, so that adds 4 cycles on average. Our answer is thus 8 clock cycles.

Versions: Different versions changed the block size, array size, and cache hit times and hit rates. Numbers were always selected such that the array size was a positive integer multiple of the block size, and such that the AMAT would be a positive integer.

Part C — 1 pt Which components of the system are accessed during a context switch? Select all that apply (bolded is true).

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

Explanation

During a context switch, we need to move to a new program. Each program has its own TLB, PC, PTBR, and Register File, so we need to update them to the values of the new program. The other sections are shared between programs, so we don't directly access them when doing a context switch.

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?

Solution

297x

Explanation

Imagine that we start off with a program that runs in 3000 ns. In order to speed this up by 75x, we need to reduce the runtime to 40 ns. Out of the 3000 ns, 30 ns is in the 1% of the program that can't be sped up, so we need to speed up the rest of the program from 2970 ns to 10 ns. We can thus see that we need to speed up the program by 297x.

Side note: We selected the initial runtime of 3000 ns because that made the math easier. This choice was somewhat arbitrary (3000 is divisible by both 100 and 75, so we get nice round numbers in our analysis); any number selected for the initial runtime would have resulted in the same analysis, though most other numbers would have resulted in some annoying fractional components. Alternatively, you could use Amdahl's law directly to yield an algebra sentence; this does tend to yield worse numbers, though, so it becomes much easier to make an algebra mistake.

Versions: Different numbers were selected for the section to be sped up, and the required speedup. These numbers were selected such that the final answer would always be an integer.

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?

NOTE: the solutions below show a sample calculation for one of multiple versions. Please read the explanation to understand your version's answers.

Sample Version

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

Solutions
Minimum Value: 5
Maximum Value: 11

Explanation
To minimize the value, we want to reduce the operations being done, since each operation increases the value of a. The best case for this is if thread 1 read the value of a first, then thread 2 did everything, then thread 1 finished working. This causes thread 2 to look like it did nothing, so the value of a becomes 5.
To maximize the value, we want to do all the operations. There are two possible orders; we can do *+*+, or **++. Checking the value of these yields 11 and 10, respectively, so we choose our maximum of 11.

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. (bolded is true)

- It is useful for control-dependent code.
- 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.
- 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.**
- **MIMD falls under the category of DLP.**
- **A major bottleneck for DLP speedup is due to register size.**
- **Intrinsics used for DLP calculations are architecture-dependent.**

Control-dependent code does not work well with DLP techniques because it is often dependent on previous values so handling multiple consecutive calculations raises the potential of unexpected behaviour.
It does not always have to be the case; it often performs well when the data is handled in a repetitive manner.
DLP handles multiple streams of **data**, not instructions.
MISD stands for "Multiple Instructions, Single Data" which does not fall under DLP's domain.
The central idea behind DLP is vectorised loading, storing, and manipulation of data.
Register speed is negligible in these scenarios; a large bottleneck becomes register sizes as it determines how much data we can handle at any one time.
DLP code cannot detect when a stream has ended; the user needs to take that into account and handle the tail case due to that behaviour.
Unaligned instructions refer to aligning memory addresses/pointers passed to a specific multiple. When the data is aligned before it's passed into processes with intrinsics, it doesn't have to do any alignment afterwards which reduces that overhead. Unaligned instructions, though easier on the user, have that additional overhead before it can process the data.
Vectors are **not** indexable.
MIMD stands for "Multiple Instructions, Multiple Data" which falls under DLP's domain.
Intrinsics are very architecture-dependent; the ones used in 61C rely on x86 architecture.

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.

NOTE: the bolded portions are parameterised; the solutions are using a sample.

```
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'})]
```

Solutions
`CODE_INPUT_1`: `a.union(b)`
`CODE_INPUT_2`: `w[-1],{w}`
`CODE_INPUT_3`: `flatMap`
`CODE_INPUT_4`: `Fline`
`CODE_INPUT_5`: `map`
`CODE_INPUT_6`: `Fw`
`CODE_INPUT_7`: `reduceByKey`
`CODE_INPUT_8`: `Fab`

```
>>> **Fab = lambda a, b: a.union(b)
>>> Fline = lambda line: line.split()
>>> Fw = lambda w: (w[-1],{w})
>>> W.flatMap(Fline).map(Fw).reduceByKey(Fab).collect()
```

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."

Solutions

1. 2, 3, 4, 5
2. 0, 2, 3
3. 0, 1, 4
4. 0, 1, 5
5. 4, 5
6. 2
7. 3
8. 4, 5
9. 1, 2, 3, 4, 5

Part B *ECC* — 1 pt Given the following bitstring `DATASTRING`, calculate the encoded bitstring to perform single-bit error correction with `TYPE` 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.

`DATASTRING`: see PraireLearn for your version; sample solution will use `0b010011000`

`TYPE`: even/odd parity; sample solution will use **odd** parity

Solutions
101100011000

Explanation

The first thing to note is what type of parity we have. In this case, we're working with odd parity which is opposite to what was taught in lecture. This just means the for each parity bit and the bits marked by it, when XORed, the output bit should be 1. For even parity, the output should be 0.
We start with our bitstring, in this case, `010011000`, and calculate the number of parity bits we need to cover the entire string. Letting m represent the total number of parity bits we need, we can calculate $2^m - m - 1$ which should be the minimum value greater than or equal to our datastring length. In this case, $2^4 - 4 - 1 = 11 \geq 8$ so we need 4 parity bits.
We insert a temporary filler bit at each position our parity bits will reside in the encoded string; in our case, our encoded string should look like `P1 P2 D1 P3 D2 D3 D4 P4 D5 D6 D7 D8`. Occasionally you'll see parity bits numbered by the value of the bit they cover in each index, which would look like `P1 P2 D1 P4 D2 D3 D4 P8 D5 D6 D7 D8` as opposed to the former which is numbered based on `index + 1`.
To calculate the parity bits, we go through each index of the temporary encoded string, left to right, 1-indexing and check whether the position that the parity bit covers in the binary representation of the bit's location has a 1 or not. After going through the entire partially encoded bitstring, you total up the bits and a simple way to check for the current XOR is to total up the values and modulo by 2. If the result is equal to your parity, then that parity bit can be `0`, otherwise it should be `1`.
As an example, `P1` watches the 0th position of an index's binary representation: $1 = 0001$ so it gets counted into the sum as 1; $2 = 0010$ does not so it's counted as 0; $3 = 0011$ gets counted for `P1` **and** `P2` as both positions have 1, and so on and so forth.

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

Part C *OS/IO* — 1 pt **Bolded** selections are true.

Version 1

Which of the following is true about the operating system?

- **The OS is considered more trusted than the user level.**
- **The user can request the OS to do certain operations through syscalls.**
- **The OS is usually trusted.**
- **The OS uses the same virtual memory space as the user does.**
- **The OS has access to the same physical memory space as the user does.**
- **It can be responsible for VA to PA translation.**
- The role of the OS is to load, run, manage, and combine multiple programs optimally.
- The OS is not the first program that runs when a device starts up.
- 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.

Explanation

The OS is generally considered to be the most secure level of a system. The user is considered to be the least. There is a level of security below the OS however (check out enclaves if you're curious)!
The user can never directly access the OS's domain; however system calls, or syscalls, act as vesicles or tunnels through which safe accesses from user to OS domain can be achieved.
Usually as there are ways of setting components of the OS that can cause insecurity. (Do not try this at home.)
The OS does not map to the same physical addresses the userspace does but virtualisation allows it to "use" the same virtual memory.
Older versions of the OS handle virtual to physical address translation. Modern devices us MMUs (memory management units) as a bridge (learn more about this in 162)!
The OS **segregates** user programs; this is to enforce protection barriers and ensures no program is aware of anything but itself.
The OS is responsible for helping set the up system when booting.
One of the OS's major jobs is to help enforce protection, not only between itself and userspace but between different processes happening in userspace. This ensures that any damage done accidentally or through malicious users and/or programs is contained and does not affect the entire system. The process itself should never have knowledge beyond its virtualisation schema.

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.

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

Explanation

All branch instructions have the ability to control polling code. This is because one of the major jobs is to continuously check for whether the input signal or output signal bits are set to 1 or not, thus checking if the data is ready. The various jump and negate instructions cannot do this if they're the sole holders of responsibility. There are ways in which you can write code that handles polling with those instructions but it would require more lines and more instructions to change the code flow properly.

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

Explanation

Compared to interrupts, polling operations take comparatively shorter to complete than interrupts.
The purpose of polling is to ensure devices that are busy (e.g. your mouse, keyboard) can continuously send input and don't have tremendous lag.
Polling happens regularly on a cycle.
Checksum packets are used in internet architecture to handle data transfers; in order to properly ensure data packets are not corrupted or malicious, checksums are used to see if the data has been altered or if it's different from what's expected. It's ACKed by the OS if it's okay and if it's not, it's deleted. This happens regularly on a timer to ensure that the system is both not overloaded but also that the data is received.
When polling, nothing else can be done.
The CPU is being used in polling.
The tradeoff between CPU usage and data being received is much better when there is a lot of I/O.
It depends on the data and devices sending the input for the system to determine whether or not DMA is needed and used or not.

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.**
- **Requires the device to signal for CPU to handle input.**
- **Higher latency per event.**
- **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.
- No additional overhead time.
- Always has better throughput.
- Sets control and data registers.

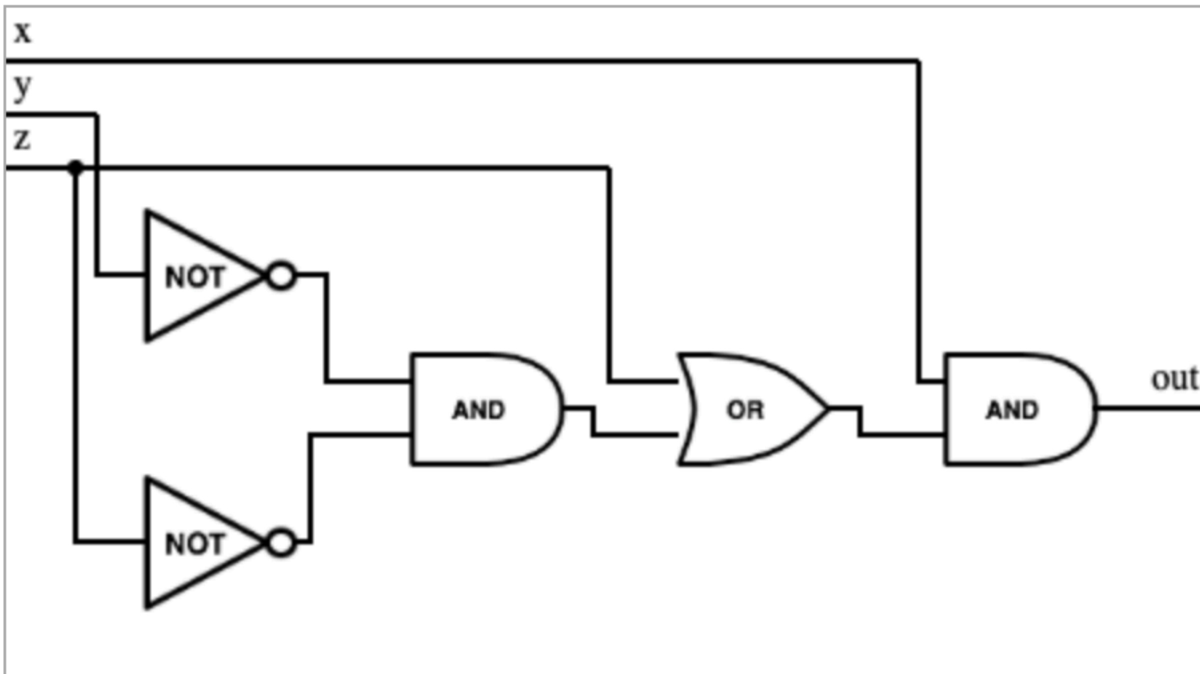
Explanations

This is the opposite of polling as it addresses devices that are less frequently visited due to potentially long response times or infrequent usage. This is to minimise the overhead in access times and so that the system doesn't sink resources into devices that does not frequently have data.
Interrupts are written in separate code and have to be triggered by the device to signal it's ready for checking and data processing; without the signal, the device will wait infinitely.
It takes on average much longer per case than polling does.
In between handling and receiving interrupts, the CPU is free to do whatever else it needs to do and will only service the device when needed.
The CPU needs to be signalled to service the device in question.
The response time is non-deterministic as it will depend on how the system wishes to handle interrupts; some systems will require the CPU to be given to the device immediately whilst other schemas might have the device wait until the CPU has finished processing the task at hand.
There is a substantial overhead time as not only does the interrupt code have to process but the system is effectively performing a context switch every time it gets signalled.
It usually has worse throughput time as it takes both longer and happens less frequently than polling does.
The interrupt code itself will not set the data and control register bits as that's unique to polling code to perform its own version of the signal to the system data is ready. During the context switch, various registers will be set but that is out of the interrupt's domain.

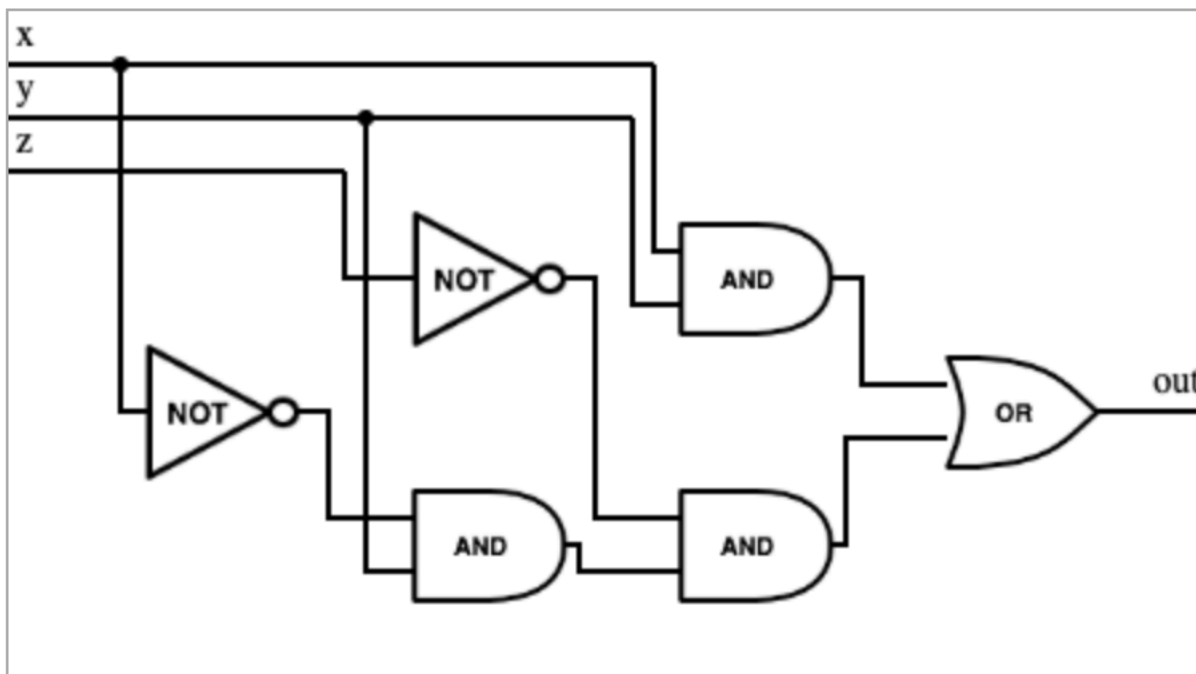
Appendix: Relevant Diagrams

Zone 2-3: Midterm Clobber — SDS

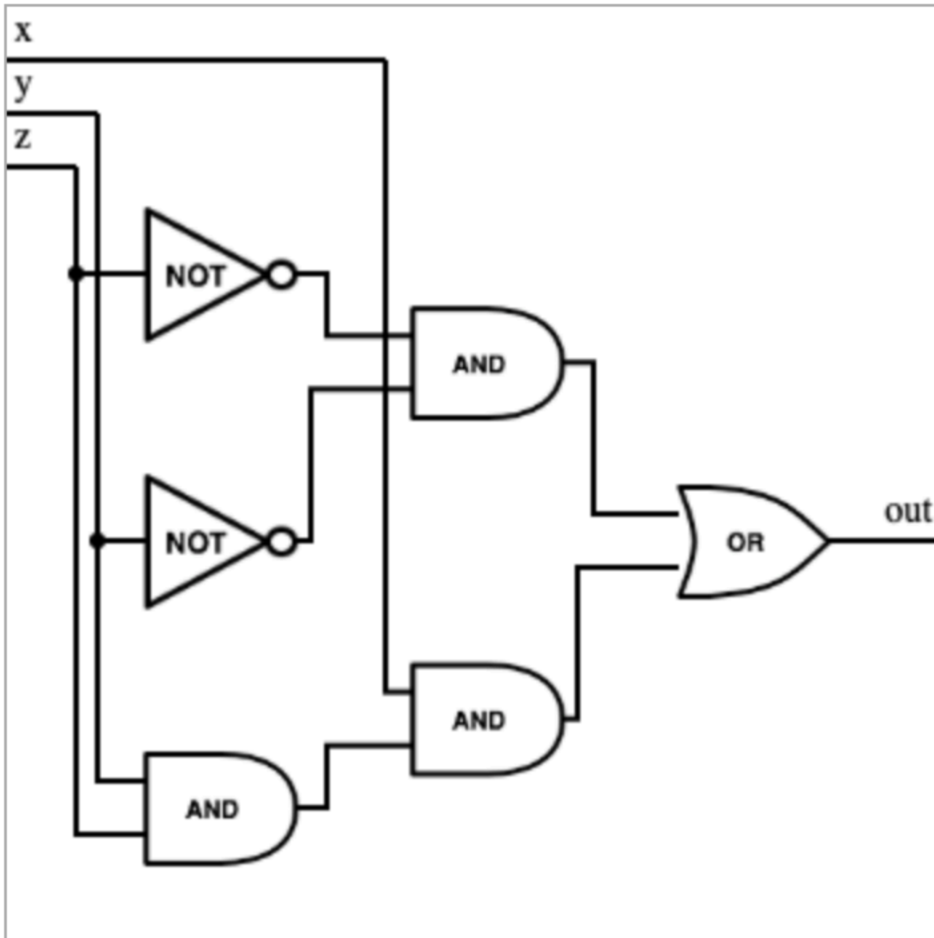
Version 1 Logic Gates



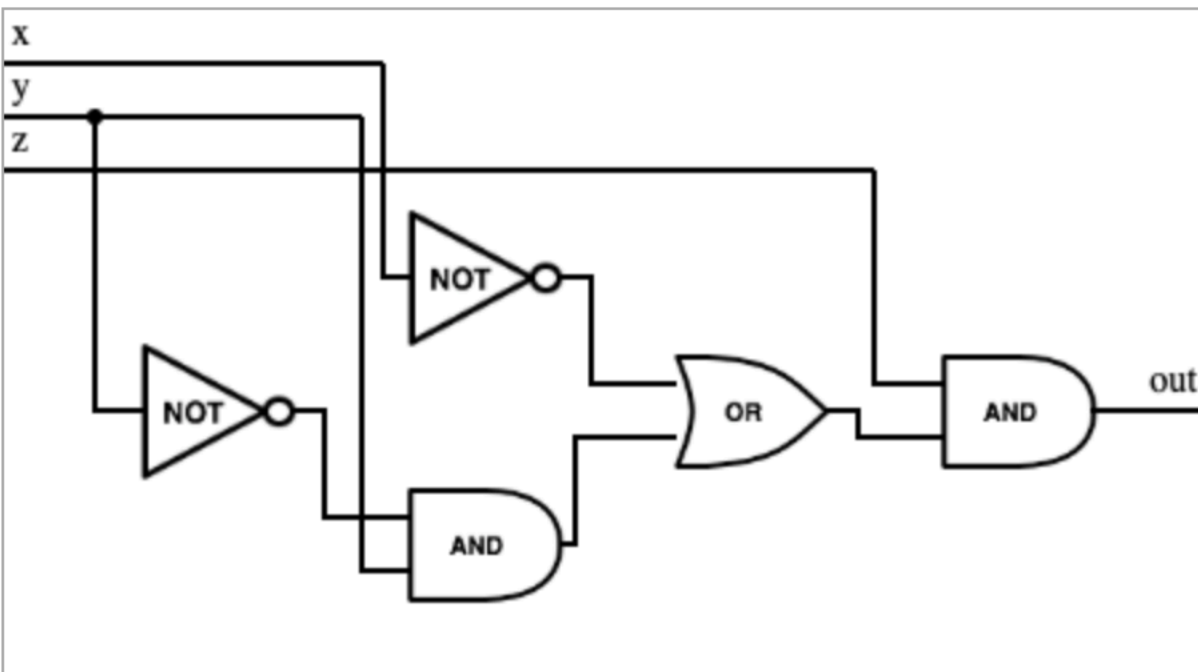
Version 2 Logic Gates



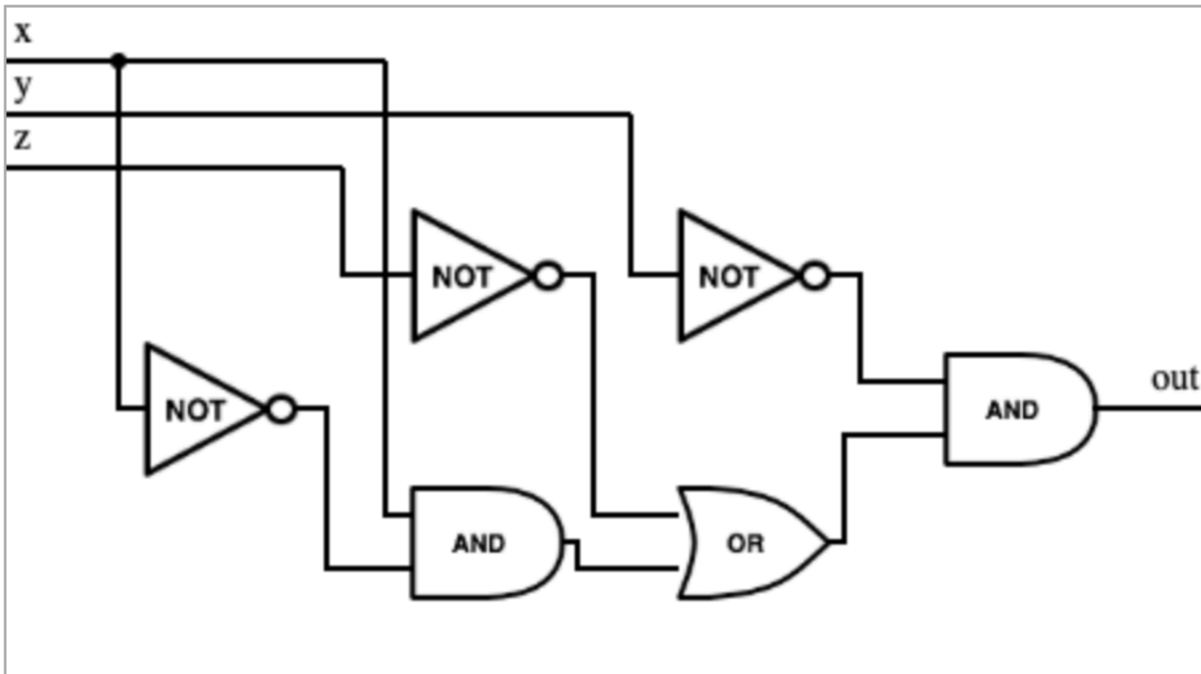
Version 3 Logic Gates



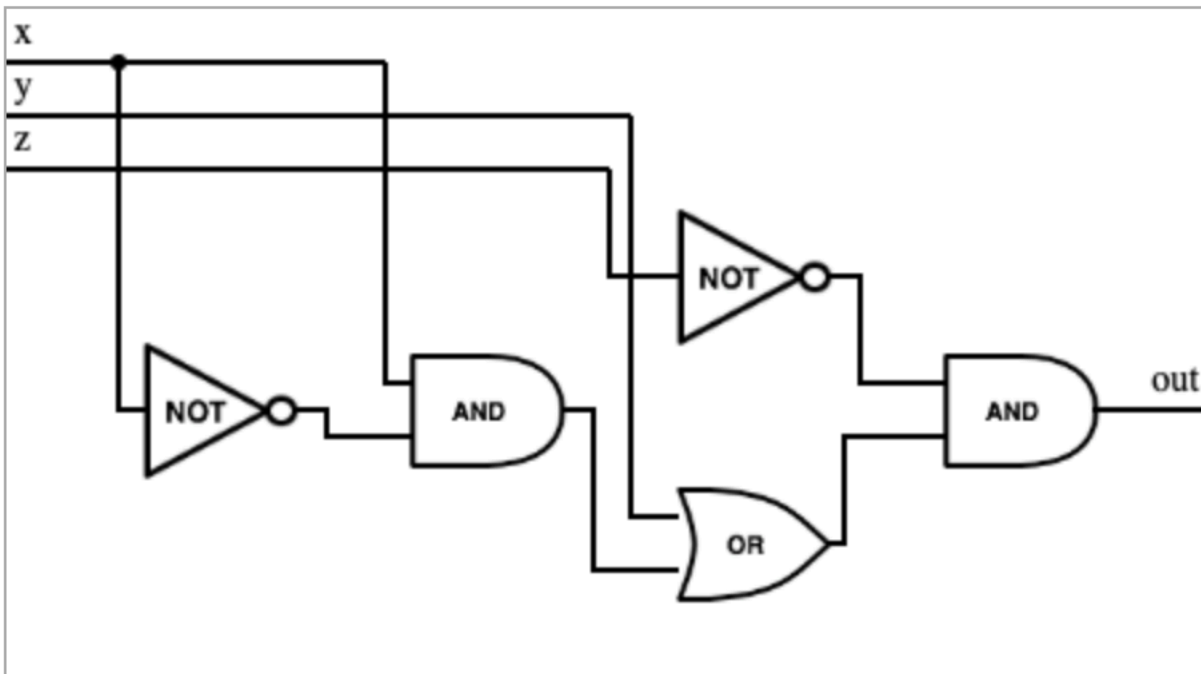
Version 4 Logic Gates



Version 5 Logic Gates



Version 6 Logic Gates



Blinding Lights FSM