Mess Detector

Video 1 = Earth's channel
Video 2 = Matthew's channel

Video 1 Outline:

1. Intro
    - Mega long ~~brief~~ showcase(see List of Impressive Things)
    - What is the mess detector?
    - How did it come to be?
    - What are the uses? lol
2. Show reversal formula and highlight which parts of the mess detector are what.

Video 2 Outline:

1. Explain LCG
    - Animated LCG formula
    - Some properties(?)
2. Explain LCG reversal (lattices)
    - *steal half the info from original docs*
3. Explain how we applied this to the mess detector
    - Step by step walkthrough of all the main components

List of Impressive Things:

Simulation Program
The mod for testing (10000 successes!)
50 km of Redstone
22 km of comparators
Tnt blast ikr

Video 1 Script:

1. Hype intro
    a. 50km of redstone
    b. 22km of comparators
    c. Reversing RNG using only redstone
    d. 100% success rate, tested 10,000 times
2. Global counting/control device (1, 46, -7)
    a. Counts to 7, to launch 7 TNT

3. Item dispensers (32, 48, -4)
    a. Advancers, to make calculation a bit easier
4. TNT launcher (0, 40, 0)
    a. Follow the TNT to the edge
5. Detectors (ender chests by the edge)
    a. TNT on ender chests to stop explosion destroying blocks, snow is same height as ender chests
    b. De-duplicator step 1: pulse extenders (looks like a checkerboard of repeaters)
    c. De-duplicator step 2: synchronize with D-flip-flop (sideways rail with instant rail line)
    d. De-duplicator step 3: merge with comparators
    e. Corner special cases (there's instant wire that's only in the corners)
6. Lookup Table
    a. Budded lines, instant (10 parallel budded lines round the whole thing)
    b. Block event delay in the corners to prevent update suppression
    c. Line is re-budded by the slimestone
    d. Goes through a D-flip-flop and is encoded to hexadecimal, then sent to the calculator for further processing (-132, 19, -153)
    e. Result is a lower bound of the seed, divided by 2^{38}
7. Signal distributed to 7 identical "cores"
8. Matrix multiplication by mm1 ($2^{48}\boldsymbol{M}^{-1}$)
    a. Input value is multiplied by a different value each TNT shot, constants also different in each core (constants stored at -110, 122, -26)
    b. Correct sign applied after the multiplier with the conditional negator (well, almost, partly factored into bvec) (-89, 115, 31)
    c. Cumulative adder at the end (pink bit)
    d. Addition of bvec ($\boldsymbol{b}$ with several other things factored in) due to cumulative adders not being initialized to 0
9. Right shift by 10 (grey bit after the pink bit)
    a. Implements the floor part of the formula
    b. Multiplication by 2^{38} cancels with the implicit division in the lookup table, division by 2^{48} cancels the implicit multiplication in mm1, overall a division by 2^{10}
10. Dot product with m6 (7th column of $\boldsymbol{M}$, $\boldsymbol{M}_6$ with 0-indexing)

    a. Cycle through the 7 values, multiply them with constants individually (-103, 125, 88), (-115, 122, 129)
    b. Uses special 48-bit multiplier, slow but more compact (the big purple beast in between them)
    c. Cumulative adder at the end (11, 113, 73)
11. Prediction
    a. Simulate the LCG with a 48-bit multiplier (105, 129, 8) and adder (69, 113, 63)
    b. Store output and compare with the next result (58, 105, 64)

Video 2 Script:

1. Intro
    a. Mess detector reverses Math.random() ingame using only redstone
    b. Done with 7 random TNT angles and the lattice technique
2. Explain LCG with the (21, 9, 100) example
    a. “Knocking off digits” is modulus
    b. Java LCG works the same way but with bigger numbers in hexadecimal (0x5deece66d, 0xb, 2^{48})
    c. Math.random() uses Random.nextDouble(). nextDouble() uses two LCG calls, to produce a 53-bit number, which when divided by 2^{53} gives a value between 0 and 1. The top 26 bits of the first call make the top 26 bits of the 53-bit number, and the top 27 bits of the second call make the remaining 27 bits of the 53-bit number.
    d. A nice property of LCGs is that you can compose (“combine”) 2 LCGs to produce a single LCG which is equivalent of advancing the seed by both LCGs. Let $LCG_a = (m_a, b_a, 2^{48})$, $LCG_b = (m_b, b_b, 2^{48})$, then $LCG_a.LCG_b = LCG_b.LCG_a = (m_am_b, m_ab_b + b_a, 2^{48})$, since $m_a(m_bx + b_b) + b_a = m_am_bx + (m_ab_b + b_a)$.
3. Explain Lattice LCG reversal
    a. LCGs have several weaknesses, but the main one the mess detector exploits is that plotting 2 calls of an LCG in 2d space produces a lattice (“slanted grid”), or, more generally, you can plot n calls in n-dimensional space.
    b. An n-d lattice with a point on the origin is a linear transformation of the set of integer points, Z^n. A general lattice has a translation from the origin as well, with the addition of an n-d vector.

c. If we have a lower and upper bound for what the seed could have been at each of the calls, we can construct an n-d box of constraints in this space, and if only one lattice point falls inside the box, it must represent the exact seeds.
d. Note: in this video we will use the row-vector convention; if you're used to column vectors, the main differences are that vectors are horizontal, and transformation matrices are post-multiplied rather than pre-multiplied.
e. From now on we will call the matrix, which transforms Z^n to the translated lattice, $\boldsymbol{M}$, and the vector which translates it $\boldsymbol{b}$. Thus, you can think of the overall formula of the lattice $\boldsymbol{L}$ to be $\boldsymbol{y} = \boldsymbol{xM} + \boldsymbol{b}$, $\boldsymbol{x} \in Z^n$, $\boldsymbol{y} \in \boldsymbol{L}$. Therefore, to map the lattice back to Z^n, $\boldsymbol{x} = (\boldsymbol{y} - \boldsymbol{b})\boldsymbol{M}^{-1}$.
f. There are many possible $\boldsymbol{M}$ and $\boldsymbol{b}$ which will produce the same lattice. The box of constraints tends to be massive, so ideally we want to pick an $\boldsymbol{M}$ such that after transforming the lattice and the box by $\boldsymbol{M}^{-1}$, the box ends up as small and even as possible, so that we have as few points to search as possible. Ideally, the box should end up smaller than the unit n-d cube, so that searching becomes a simple case of rounding. More on the choice of $\boldsymbol{b}$ later.
g. The algorithm for producing an even box is called LLL lattice reduction. It takes an arbitrary $\boldsymbol{M}_0$ and produces successively more orthogonal ("better") $\boldsymbol{M}_i$ until we reach the optimal matrix $\boldsymbol{M}$. Of course, we need to supply it with an initial basis matrix $\boldsymbol{M}_0$ which can be as simple as the top row being $(1, m, m^2, m^3, ...)$, and the other diagonal entries being 2^{48}.
h. It's sufficient to only know the maximum corner point of the box, $\boldsymbol{t}$, and its size. Were $\boldsymbol{M}^{-1}$ to contain only positive elements, the point we are searching for would be simply $\lfloor(\boldsymbol{t} - \boldsymbol{b})\boldsymbol{M}^{-1}\rfloor$, however, for every negative element of $\boldsymbol{M}^{-1}$, you have to subtract the box size from elements $\boldsymbol{t}$ during multiplication. This leads to a situation where it's no longer a normal matrix multiplication, but take a moment to convince yourself [animation will help] that if you go through the multiplication, you are out by a constant error vector $\boldsymbol{e}$ afterwards. Thus, the answer is $\lfloor(\boldsymbol{t} - \boldsymbol{b})\boldsymbol{M}^{-1} - \boldsymbol{e}\rfloor$. This can be rearranged to $\lfloor\boldsymbol{tM}^{-1} - (\boldsymbol{bM}^{-1} + \boldsymbol{e})\rfloor$. Let *bvec* = $-(\boldsymbol{bM}^{-1} + \boldsymbol{e})$, thus we

have ⌊$\boldsymbol{tM}^{-1}$ + *bvec*⌋. *bvec* will incorporate more error terms than this in the mess detector.

i. Once you have the correct point in Z^n space, you need the correct seed in ***L*** space. This is as simple as ⌊$\boldsymbol{tM}^{-1}$ + *bvec*⌋***M*** + ***b***. However, in the mess detector, we only care about the seed after reversal, which means we only want the final element of the resulting point vector. Therefore, you can simplify to ⌊$\boldsymbol{tM}^{-1}$ + *bvec*⌋ . *m6* + *b*, where *m6* is the rightmost column of ***M*** transposed into a row vector, and *b* is the final element of ***b***. This is where the choice of ***b*** becomes significant. We choose the ***b*** such that the last element of ***b*** is 0. This removes the need for the final addition. So the overall formula is ⌊$\boldsymbol{tM}^{-1}$ + *bvec*⌋ . *m6*.