Examination of SID noise waveform

Sampling the waveform

The waveforms of the SID in the c64 and c128 can be examined, because the SID provides a 8-bit output register of the waveform of voice 3 in register $1b.

The exact waveform can also be examined from the “start” of the waveform, because the test-bit (bit 3 in register $12 for voice 3) can be used to reset the random-waveform.

To examine the data, we want to be able to sample the output in a consistent way. First step is to establish the sampling-rate to be used. Of course the sampling-rate must depend on the frequency used in registers $0e and $0f, so our job is to determ ine the dependency between the frequency and the waveform “wavelength” in cycles, i.e. the number of cycles between the waveform-values change.

To this end, we want to read the waveform output of register $1b as fast as possible. With a REU it is possible to sample the value every cycle, and with the program “Cyclewise”, which samples $10000 bytes of the waveform output into bank 0 of a REU, I have collected the data in table 1 which are valid for the noise waveform. The number in the first column is the frequency put into registers $0e and $0f. The second number is the number of cycles each value of the waveform at least lasted, while the third number is the number of cycles the first value ($fe) lasted when the sampling is done using the construct:

```
sta $d412 ;Start waveform
stx $df01 ;Start sampling
```

The STA and STX instructions take 4 cycles, but the data in the table suggests that the sampling is delayed only 3 cycles. Furthermore the first value count is probably 1.5 times longer than the waveform.

Table 1: Frequency, wavelength and initial delay

```
Frequency     Wavelength    1st value cnt
-----------------------------------------------
  $ffff       $10.001000      $16
  $C000       $15.555555      $1D
  $AAAA       $18.001800      $22
  $8000       $20          $2D
  $6000       $2A.AAAAAB      $3D
  $4000       $40          $6D
  $3222       $51.B3F644      $78
  $3000       $55.555555      $7D
  $1000       $100          $17D
  $0100       $1000          $17FD
```

This leads to the conclusion that the frequencies can generated with a loop like this for the noise waveform:

```
void Frequency-generator(long freq) {
  long delay=0x180000;    /* C-notation for $180000 */
  long cycle=0;
  for (;;;) {         /* Repeat forever */
    delay= delay-freq;
    if (delay0) {
      delay= delay+0x100000;  /* C-notation for $100000 */
      waveform_output= calculate new value for waveform;
    };
    waveform[cycle]= waveform_output;
    cycle= cycle+1;
  }
}
```

Furthermore, we notice that the highest frequency that is 100% cycle-aligned is $8000, which gives us a wavelength of 32 cycles. In the following discussion this frequency is assumed to be used.

Notice that the above program is not garanteed to be 100% correct, since I haven't tested it. Specifically is the value of $180000 a quick hack and it might be $17ffff just as well.

The noise-waveform

The next step is to determine whether the noise waveform loops, since this knowledge is useful in respect to an algorithm based on manipulating an internal register. If the algorithm is based upon manipulation an internal register (by doing for instance shifting and exclusive-or), the values _have_ to restart sometime, because an internal register on n bits can only hold 2^n different values. In other words, we will try to establish whether the output stream from $d41b will repeat itself and if it doesn't in a set period of time, say n clockcycles, we will know that the internal register will be of at least log2(n/32) bit s, since the value changes each 32 clockcycles (note: log2(x) is the same as log(x)/log(2)).

The program “Loopchecker” checks for loops with an algorithm like this:

```
void Loopchecker() {
  start_noise-waveform();

  /* We want to sample *inside* the potential loop */
  wait_a_while();

  /* Sample 256 values */
  for (i=0;i256;i=i+1)
    data[i]= peek($d41b);

  /* Sample until those 256 values arrive again */
  do {
    i=0;
    while (peek($d41b) = data[i]) do
      i=i+1;
  while (i256);
```

```
  end;
```

This program will terminate if a sequence of 256 recorded bytes will appear again later. If the 256 bytes reappear, we can be quite certain that the waveform loops, since the chance of this happening with a totally random source is infinitely small (well below 1e-500). Marko Makela and I (Asger Alstrup) hacked a loopchecker together over the IRC, and our results with a 16 cycle sampling-rate was that the computer terminated after approximately 2 minutes and 15 seconds. However, our results weren't consistent, partly because of the 16 cycle sampling frequency, which isn't completely cycle-aligned and partly because the reseting of a waveform with the TEST-bit does not reset the waveform immediately, but rather $2000-$8000 cycles later (this figure varies greatly, d oes anybody know of a way to reset the waveform fast?).

The “Loopchecker” program given below fixes these errors, and if you run this program, the computer will terminate after approximately 4 minutes and 32 seconds or 272 seconds. Firstly this means that the waveform repeats. Secondly, when the waveform changes every 32 cycles, this means that the length of the loop is approximately

```
(272 sec * 980000 cyc/sec)/32 cyc/bytes ~= 8.0 Megabytes.
```

Thirdly, this implies an internal register length of log2(8.0 MB)= 23 bits. And finally, we must admit that it would be a difficult task to sample the entire 8 MB on a c64 with only 64KB memory. It is possible to sample the lot in chunks and saving it on multiple disks, but this is not a trivial task. Notice that the data probably can't be packed much since they are random. It should be mentioned that by changing the delayloop after the noisewaveform is selected, it can be demonstrated that the waveform indeed loops after 8 megabytes of data nomatter where in the waveform cycle you are.

With the implication of an internal register of 23 bits, the next step is to find a pattern in the data which might hint the algorithm used to produce the data. Since we know a shifting and eor scheme can be used, it might be useful to take a look at the data in binary:

```
11111110
11111100
11111100
11111100
11111000
11111000
11111000
11111000
11110000
11110000
11100000
11100000
11100000
11000000
11000000
11000000
11000000
10000001 New value for bit 0!
10000001
00000011
00000011
```

```
00000011
00000110
00000110
00000100
00000100
00001100
00001000
00011000
00011000
00011000
00110000
00110000
...
```

It should be quite clear that some kind of shifting scheme is used. Further examination of the data suggests that the internal register indeed is on 23 bits, where the mapping between the 8 bits in the output and the internal 23 bit register is like this:

```
Internal register bit number
    22 21 20 19 18 17 16 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1
Output bit from register $1b
     7     6           5        4     3           2        1     0
```

The first data from the output can be reproduced with this layout if the internal register is leftshifted, and the initial value of the internal register is:

```
     1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  0  0  0
```

Now we need to explain the new bits appearing in bit 0 of the data. Even further examination of the data implies that an eor-gate is used to feed bit 0 of the internal register. I have found that the bits 22 and 17 of the internal register provides the feed for bit 0 through an eor-gate, which gives us the entire mechanism:

```
22-21-20-19-18-17-16-15-14-13-12-...-8-7-6-5-4-3-2-1-0
|                  |                                  |
+---->----->-----> eor ->----->----->----->----->----->----->----->---->+
```

This can also be expressed as a C-program like this:

```
/* Test a bit. Returns 1 if bit is set. */
long bit(long val, byte bitnr) {
  return (val & (1<<bitnr))? 1:0;
}


/* Generate output from noise-waveform */
void Noisewaveform {
  long bit22;   /* Temp. to keep bit 22 */
  long bit17;   /* Temp. to keep bit 17 */
```

```
  long reg= 0x7ffff8; /* Initial value of internal register*/

  /* Repeat forever */
  for (;;;) {

    /* Pick out bits to make output value */
    output = (bit(reg,22) << 7) |
         (bit(reg,20) << 6) |
         (bit(reg,16) << 5) |
         (bit(reg,13) << 4) |
         (bit(reg,11) << 3) |
         (bit(reg, 7) << 2) |
         (bit(reg, 4) << 1) |
         (bit(reg, 2) << 0);

    /* Save bits used to feed bit 0 */
    bit22= bit(reg,22);
    bit17= bit(reg,17);

    /* Shift 1 bit left */
    reg= reg << 1;

    /(* Feed bit 0 */
    reg= reg | (bit22 ^ bit17);
  };
};
```

Every loop in the above program provides a new value for the noise waveform in the variable “output”. Notice that the value will repeat for several cycles, just like the real SID. All in all, I find that the above program can sufficiently reproduce t he output from the noise-waveform in the SID.

When it comes to the quality of the pseudo-random generated numbers, one thing is clear: Since all bit-patterns of the 23 bits are generated, all values in the 8 bit output will appear equally many times over time. Furthermore, I believe this particul ar scheme is recognized as one of the better ones for doing pseudo-random numbers, and it is based on polynomiums, much like the CRCs used for checksumming. But since the values are reproducible, you'll have to be careful if you plan to use the noise waveform as a source for random numbers for games and such. Firstly, only reset the waveform when the games is loaded. And wait with the sampling until the user presses a key or fire, which will provide an offset into the stream, so t he values will be different from each run. Furthermore, be sure to read the values at an appropriate sampling-rate, for instance every 32 cycle with a frequency of $8000. One prefered method is to do a table of say $2000 values before the game starts , so that voice 3 can be used in the game-music or for sound-effects in the game.

Programs in assembler

```
;"Cyclewise" - records noise-waveform each cycle using REU.
;-------------------------------------------------------

    * = $1000
```

```
freq    = $8000        ;Define frequency here

cycle   jsr init    ;Init screen

    lda #$08    ;Set testbit to reset the waveform.
    sta $d412

    jsr pause    ;Give it time to settle - nescessary on my machine.

    lda #<freq     ;Set frequency
    ldx #>freq
    sta $d40e
    stx $d40f

    ;Set up REU

    ldx #$d41b    ;Define REU read address to be $d41b.
    ldy #>$d41b
    stx $df02
    sty $df03

    lda #$00     ;Record into $0000
    sta $df04
    sta $df05
    sta $df06    ;in bank 0.

    sta $df07    ;Transfer $10000 bytes.
    sta $df08

    lda #$00    ;No interrupts from the REU.
    sta $df09

    lda #%10000000    ;Fix c64 adress at $d41b.
    sta $df0a

    lda #$80
    ldx #%10010000    ;REU command: Do transfer from c64->REU.

    ;start waveform and do the sampling

    sta $d412    ;Enable noise-waveform
    stx $df01    ;and start recording.
    jmp done    ;After sampling, wrap it up


;Loopchecker - checks for a loop in the noise-waveform.
;------------------------------------------------------

    * = $1100

block   = $2000
```

```
loop    jsr init    ;Init screen

    lda #$08    ;Set testbit to reset the waveform.
    sta $d412

    jsr pause    ;Give it time to settle - nescessary on my machine.

    lda #$00     ;Set frequency to $8000
    sta $d40e
    lda #$80
    sta $d40f

    lda #$80     ;Start noise-waveform
    sta $d412

    jsr pause    ;Wait a while, so we get well into the waveform

    inc $d020    ;Signal that the recording starts now

    ldx #0        ;Sample 256 bytes at 32 cycles
l1  lda $d41b
    sta block,x
    bit $ffff
    bit $ffff
    bit $ffff
    bit $ffff
    nop
    inx
    bne l1

    nop        ;Wait exactly 1 value
    bit $ea
    bit $ffff
    bit $ffff

l2  ldy #$1d    ;See if the 256 bytes repeat
    ldx #$00
l3  lda $d3ff,y    ;(Trick to get 5 cycles execution time)
    bit $ffff
    bit $ffff
    bit $ffff
    bit $ffff
    cmp block,x
    bne l2
    inx
    bne l3

    jmp done    ;And exit if they do

;Misc subroutines
;----------------
```

```
    * = $1800

    ;Init screen

init    sei         ;Secure accurate timing by
    lda #$00    ;disabling sprites
    sta $d015
in1 lda $d011    ;and screen.
    bpl in1
    and #$ef
    sta $d011
    rts

    ;Wait 50 frames so waveform can be reset

pause   ldx #50
p1  bit $d011
    bpl p1
p2  bit $d011
    bmi p2
    dex
    bne p1
    rts

    ;Wraps it up

done    lda $d011    ;Reenable screen
    ora #$10
    sta $d011

    cli         ;and exit.
    rts
```

Output from “cyclewise” with a frequency of $ffff (The data are correct: the wavelength is effectively 16 cycles for the first $10000 or so values):

Value (number of times) $fe ($16), $fc ($30), $f8 ($40), $f0 ($20), $e0 ($30), $c0 ($40), $81 ($20), $03 ($30), $06 ($20), $04 ($20), $0c ($10), $08 ($10), $18 ($30), $30 ($20) ...