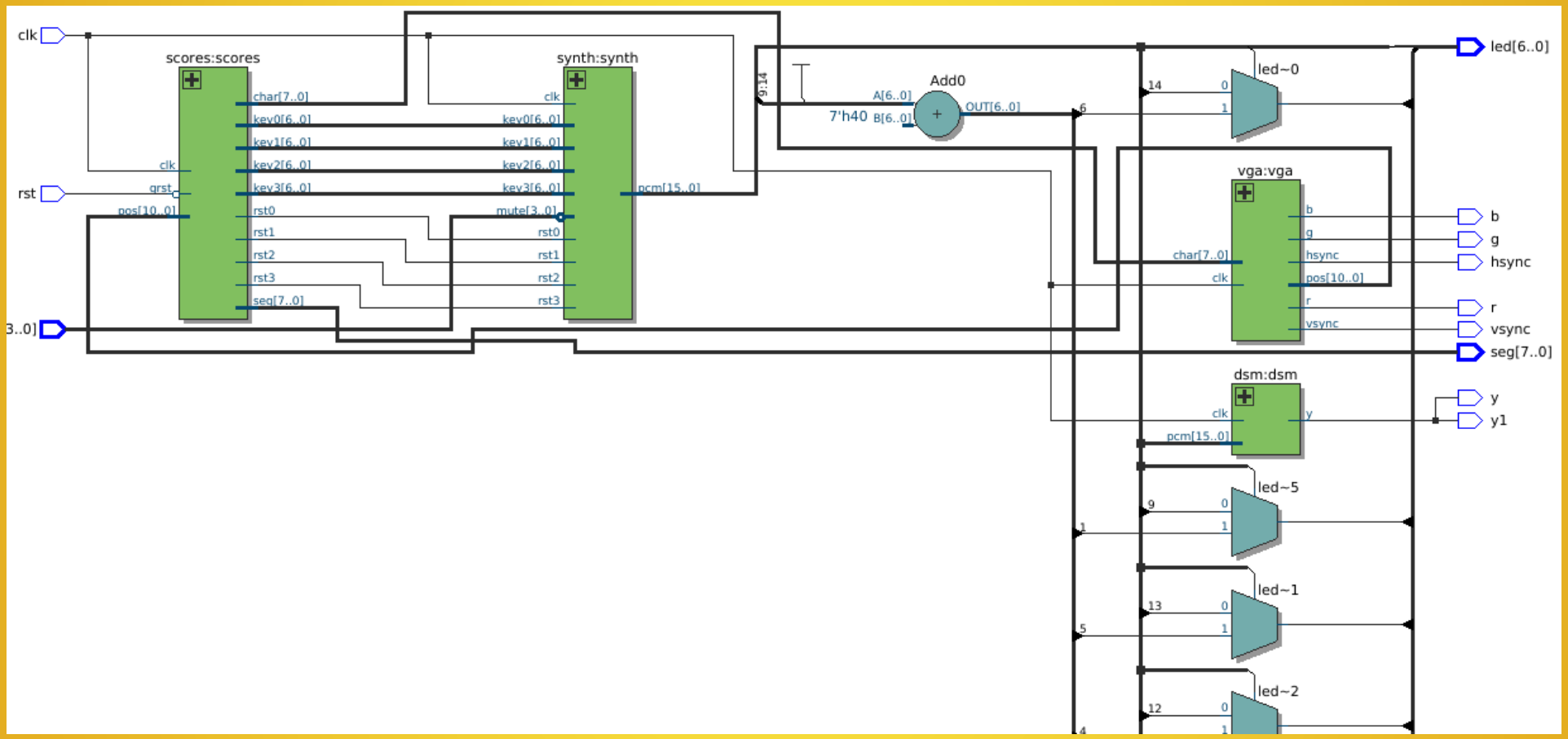
MIDI Sequencer with VGA controller on an FPGA

Star Brilliant
4 July 2016

Overall schematic



Score → Synth → Delta-Sigma Mod, VGA, LED, 7-seg tube

Resource occupation

Flow Status	Successful - Mon Jul 4 00:43:07 2016
Quartus II 64-Bit Version	13.1.0 Build 162 10/23/2013 SJ Web Edition
Revision Name	top
Top-level Entity Name	top
Family	Cyclone III
Device	EP3C5E144C8
Timing Models	Final
Total logic elements	1,780 / 5,136 (35 %)
Total combinational functions	1,743 / 5,136 (34 %)
Dedicated logic registers	497 / 5,136 (10 %)
Total registers	497
Total pins	28 / 95 (29 %)
Total virtual pins	0
Total memory bits	130,048 / 423,936 (31 %)
Embedded Multiplier 9-bit elements	1 / 46 (2 %)
Total PLLs	0 / 2 (0 %)

Only one multiplier, no PLLs
16 KiB memory for 4,000 notes on 4 tracks

Inputs & Outputs

Inputs:

- Clock: **clk**, 20 MHz (PIN_22)
- Reset: **rst**, Button 7
- Track mute: **mute**, Button 6..3

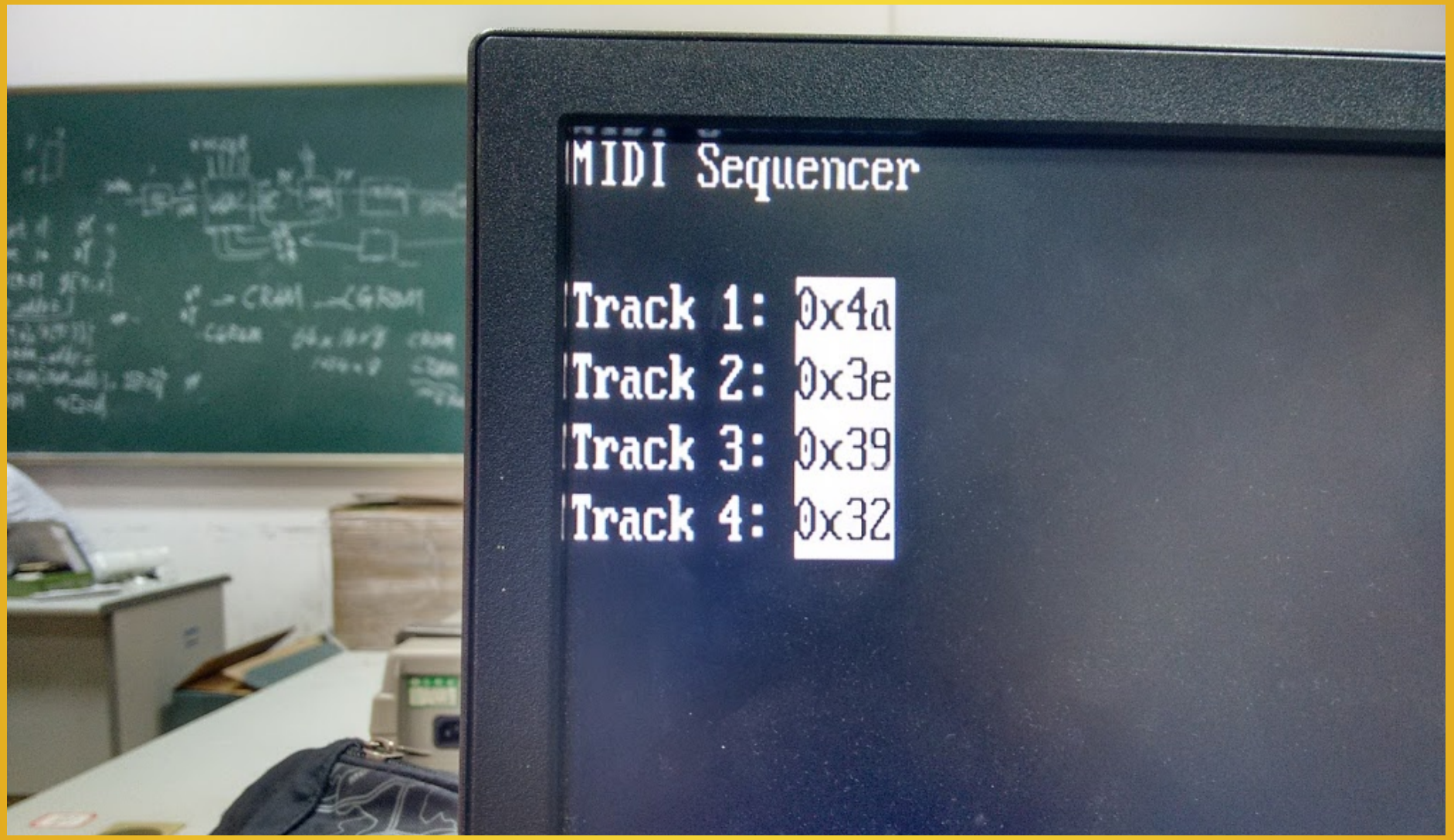
Outputs:

- Direct Stream Signal (DSD): **y**, Beeper (PIN_11)
- VGA: **r, g, b, hsync, vsync**
- Volume indicator: **led**, LED 6..0
- Score cursor: **seg**, 7-seg tube

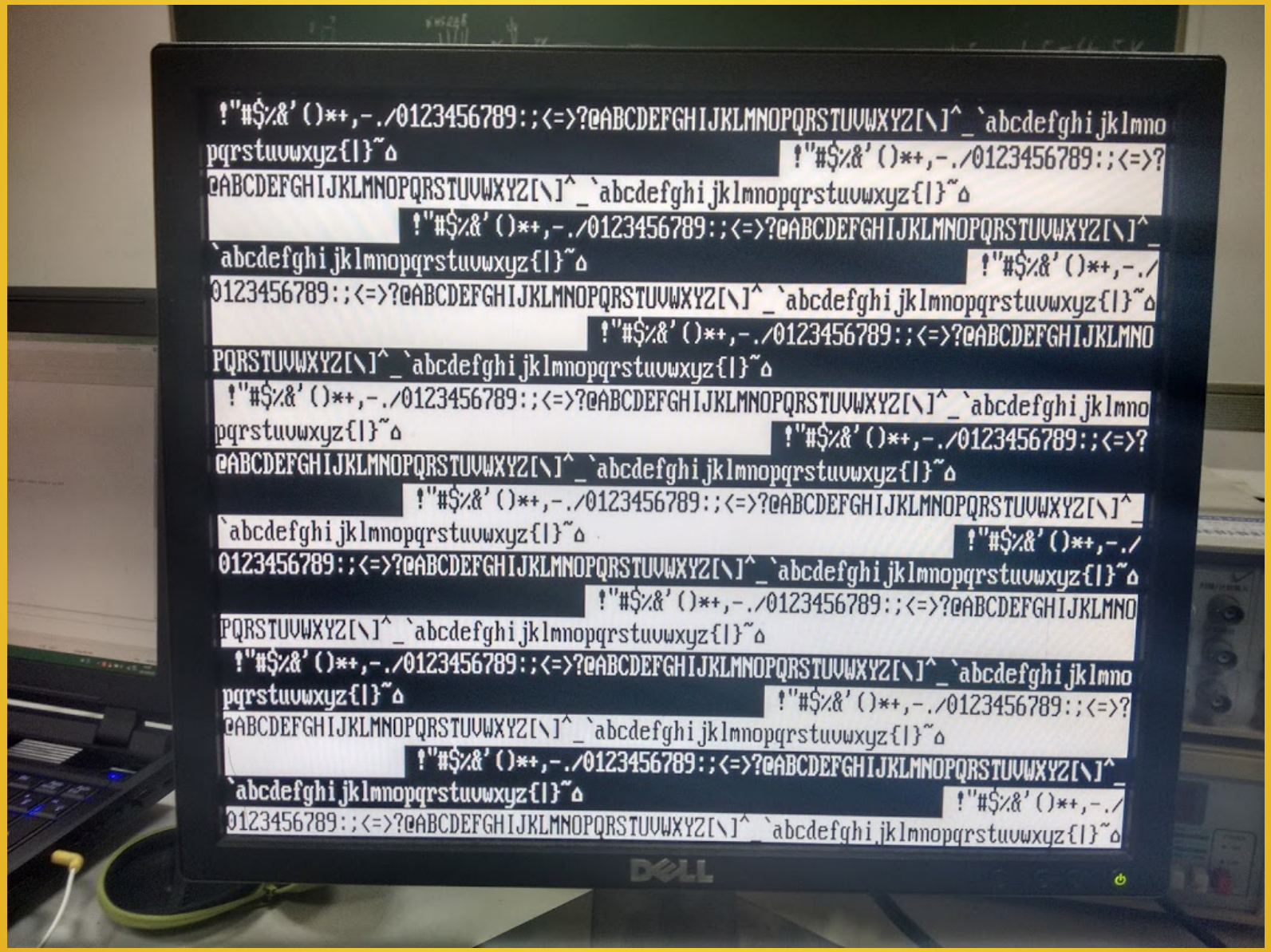
Contents

- VGA controller
- MIDI score controller
- MIDI synth
- Delta-Sigma modulator

Flexible VGA output

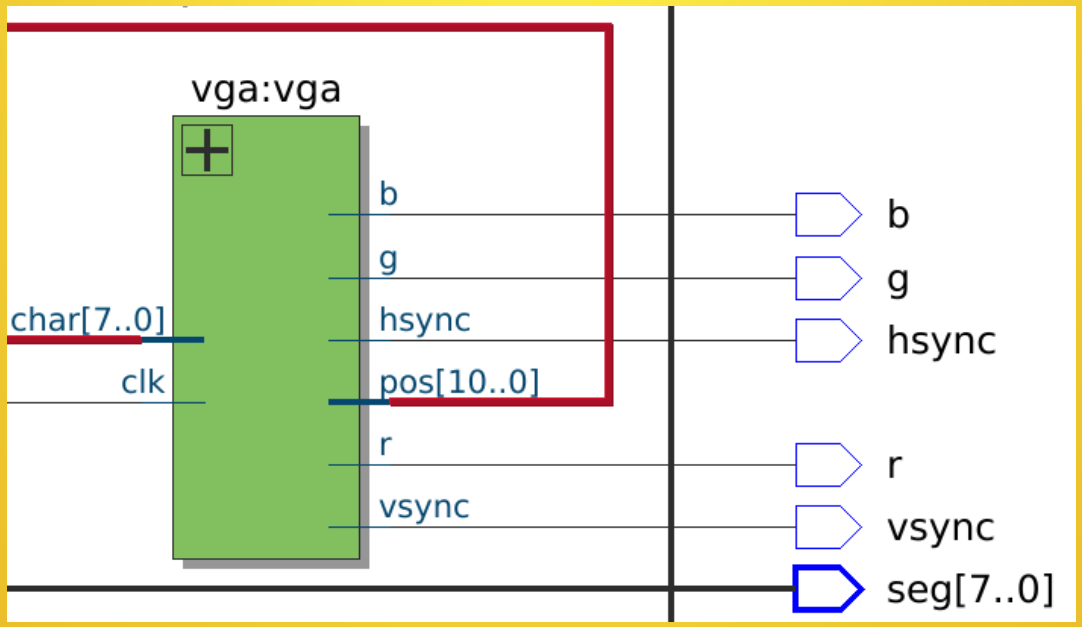


Full ASCII support



Details about VGA output

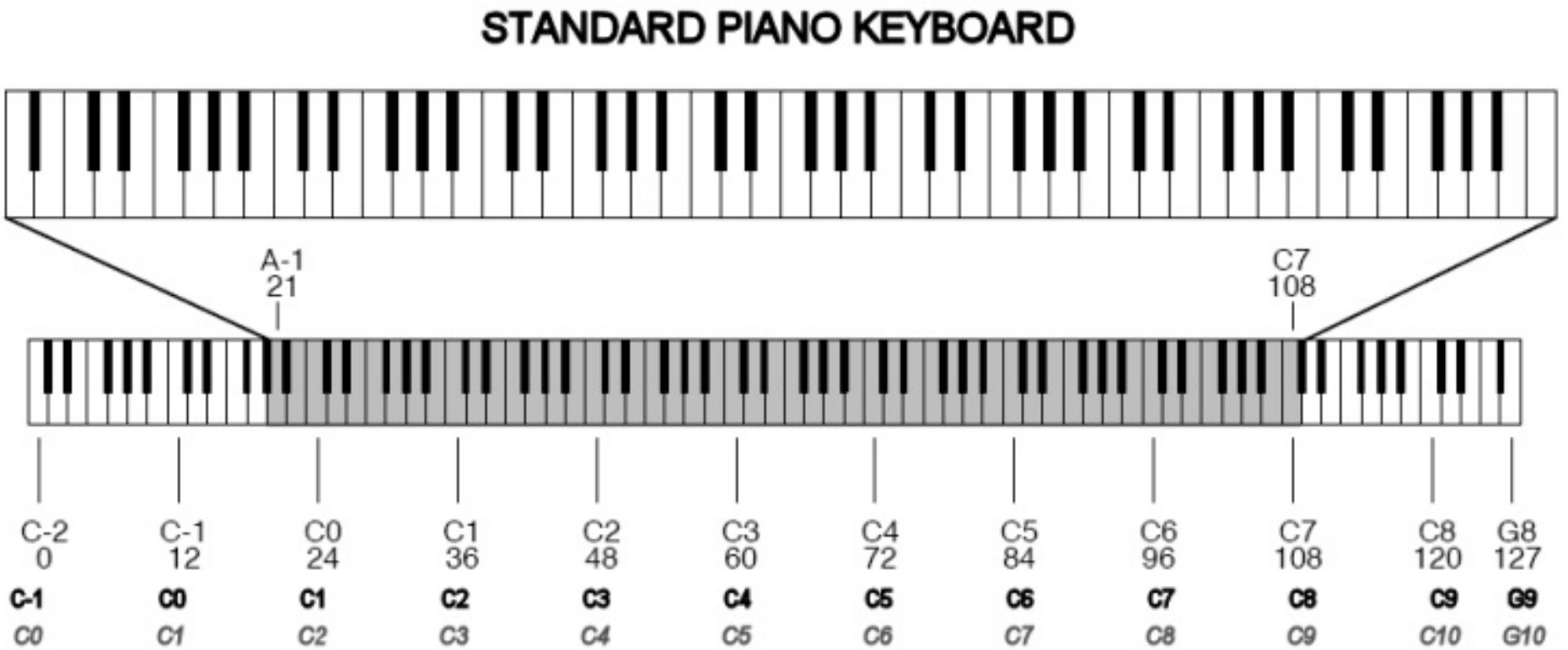
- Runs at 640×400 resolution
- 80×25 characters, 8×16 IBM-compatible bitmap font
- Pixel clock is 20 MHz, thus requiring no PLLs
- Really easy-to-use interface



VGA controller interface

```
always @(posedge clk)
    case(pos)
        /* Row 0 */
        0: char <= "M";
        1: char <= "I";
        2: char <= "D";
        3: char <= "I";
        5: char <= "S";
        6: char <= "e";
        7: char <= "q";
        8: char <= "u";
        9: char <= "e";
        10: char <= "n";
        11: char <= "c";
        12: char <= "e";
        13: char <= "r";
        /* Row 2, 3, 4, 5 */
        2*80, 3*80, 4*80, 5*80: char <= "T";
        2*80+1, 3*80+1, 4*80+1, 5*80+1: char <= "r";
        2*80+2, 3*80+2, 4*80+2, 5*80+2: char <= "a";
        2*80+3, 3*80+3, 4*80+3, 5*80+3: char <= "c";
        2*80+4, 3*80+4, 4*80+4, 5*80+4: char <= "k";
        2*80+6: char <= "1";
        3*80+6: char <= "2";
        4*80+6: char <= "3";
        5*80+6: char <= "4";
        2*80+7, 3*80+7, 4*80+7, 5*80+7: char <= ":";
```

128-key MIDI tuning



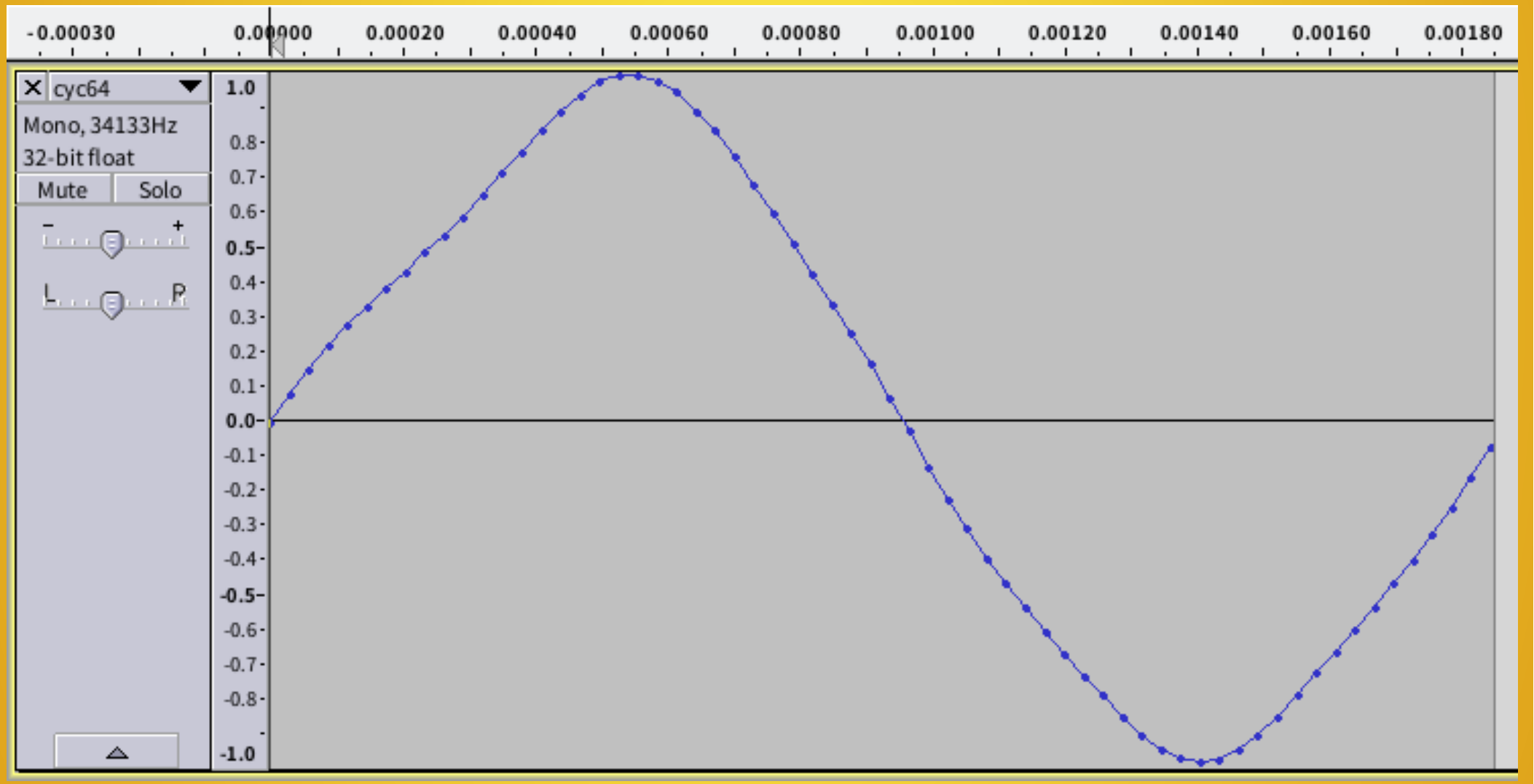
MIDI score format

- Converted from Standard MIDI Format (SMF) version 1
- Song 0 title: *Senbonzakura*
- Song 1 title: *Nyan nyan cat*
- ~ 4,000 notes, 4 tracks
- Stored into “note%d.hex” and “time%d.hex”
- Notes are stored in standard MIDI key names, 7 bits
- Times are stored in units of “ticks”, 16 bits
- 480 ticks per beat, 143 beats per minute (BPM)

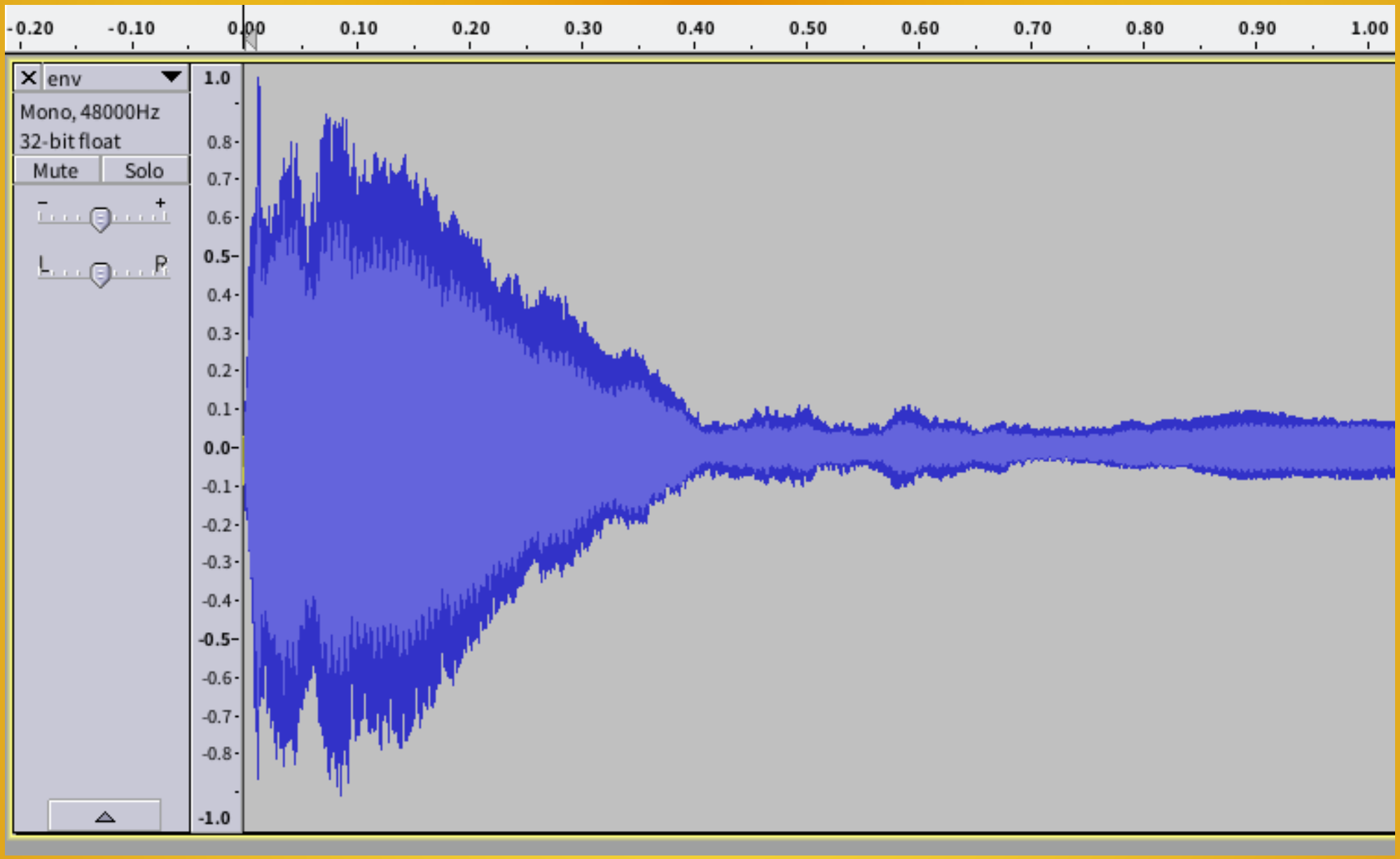
Piano sound sample

- Converted the piano sound sample from an SF2 format soundfont library
- Extracted into 2 parts: “cyc”, “env”
- Each stored in 64 bytes
- cyc × env = PCM

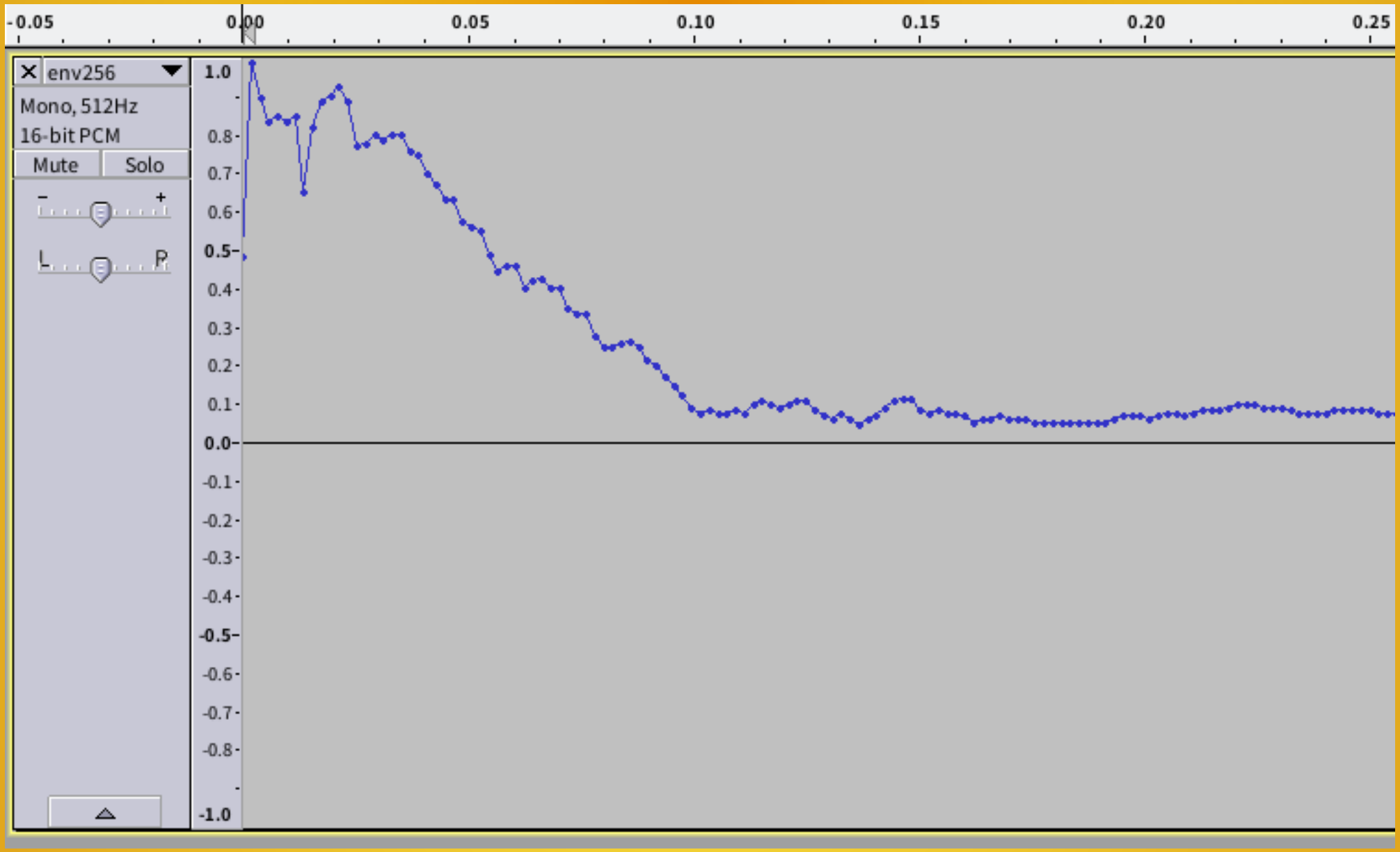
Sample cyc


-0.00030
0.00000
0.00020
0.00040
0.00060
0.00080
0.00100
0.00120
0.00140
0.00160
0.00180
cyc64
Mono, 34133Hz
32-bit float
Mute
Solo
1.0
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0.0
-0.1
-0.2
-0.3
-0.4
-0.5
-0.6
-0.7
-0.8
-1.0

Source envelope



Sample env


-0.05
0.00
0.05
0.10
0.15
0.20
0.25
env256
Mono, 512Hz
16-bit PCM
Mute
Solo
1.0
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0.0
-0.1
-0.2
-0.3
-0.4
-0.5
-0.6
-0.7
-0.8
-1.0

Time division multiplier mux

- Seperate clock into 4 rounds
- Each track use the multiplier turn by turn
- Verilog “generator” feature

Time division multiplier mux

```
reg signed [7:0] cyci;
reg signed [7:0] envi;
wire signed [15:0] pcmi = cyci * envi;
always @(negedge clk) begin
    cyci <= cycle[cyccnt[clk_div]];
    envi <= {1'b0, env[envcnt[clk_div]]};
end

reg signed [15:0] pcms [3:0];
genvar i;
generate
    for(i = 0; i < 4; i = i+1) begin : pcms_for
        always @(negedge clks[i])
            pcms[i] <= mute[i] ? 0 : pcmi;
    end
endgenerate

assign pcm = {pcms[0][15], pcms[0][15], pcms[0][15:2]}
           + {pcms[1][15], pcms[1][15], pcms[1][15:2]}
           + {pcms[2][15], pcms[2][15], pcms[2][15:2]}
           + {pcms[3][15], pcms[3][15], pcms[3][15:2]};
```

Delta-Sigma Modulator

- Simulate arbitary wave using square wave
- Convert from PCM to PWM
- A kind of PWM, better SNR than SPWM

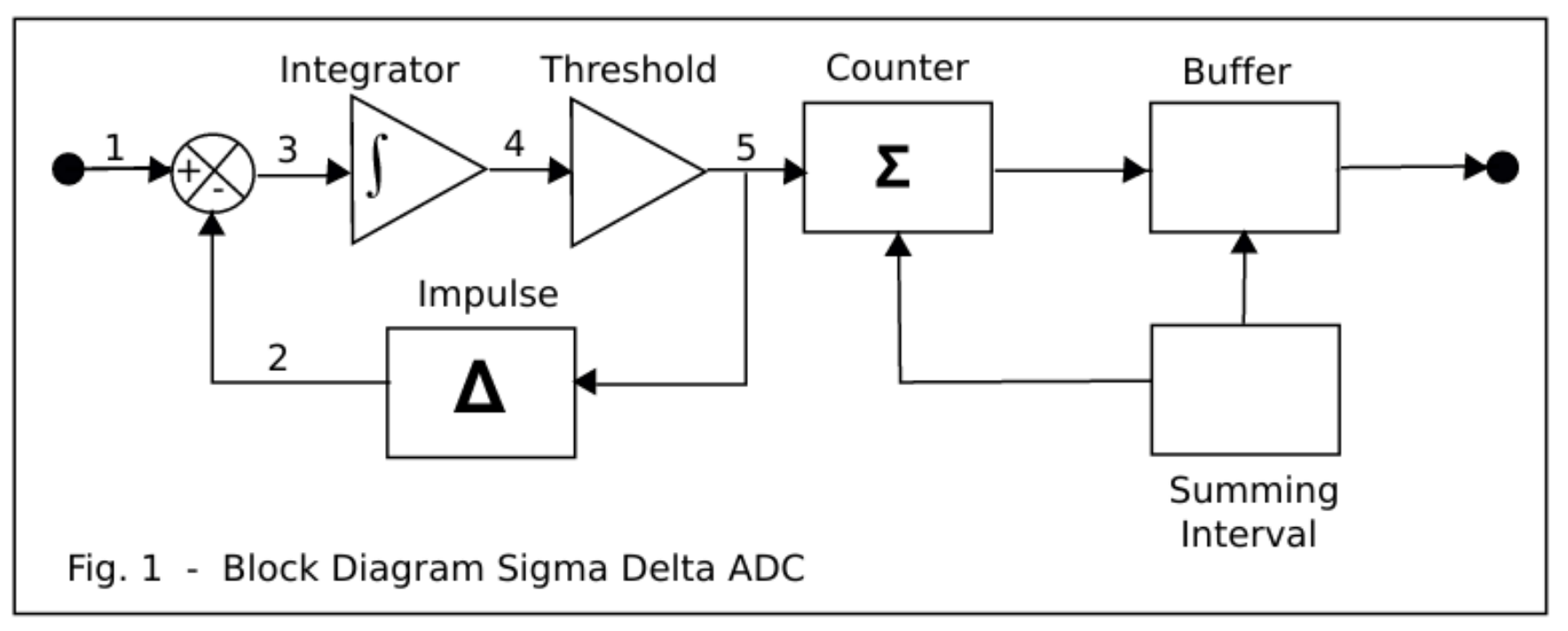
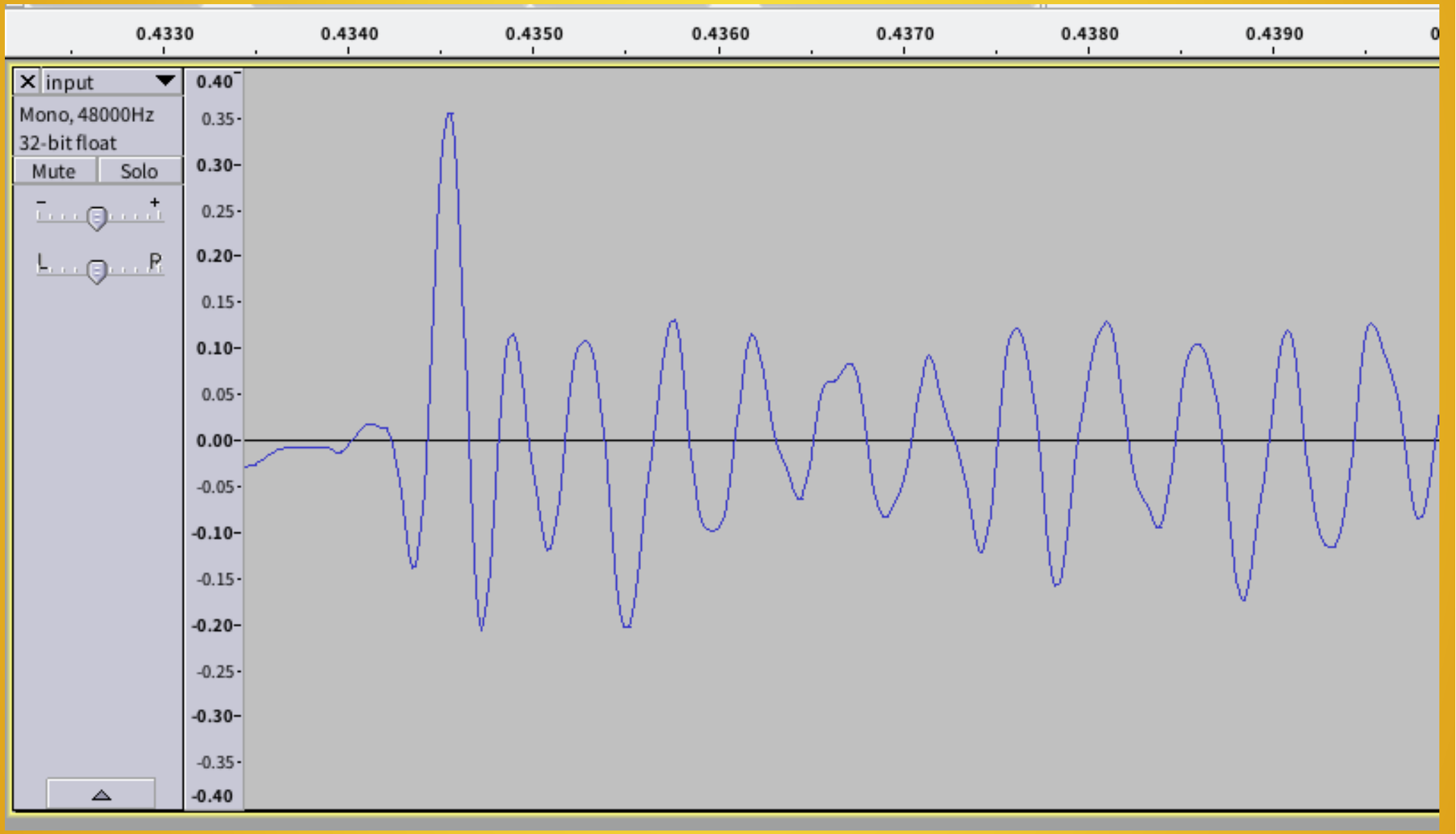
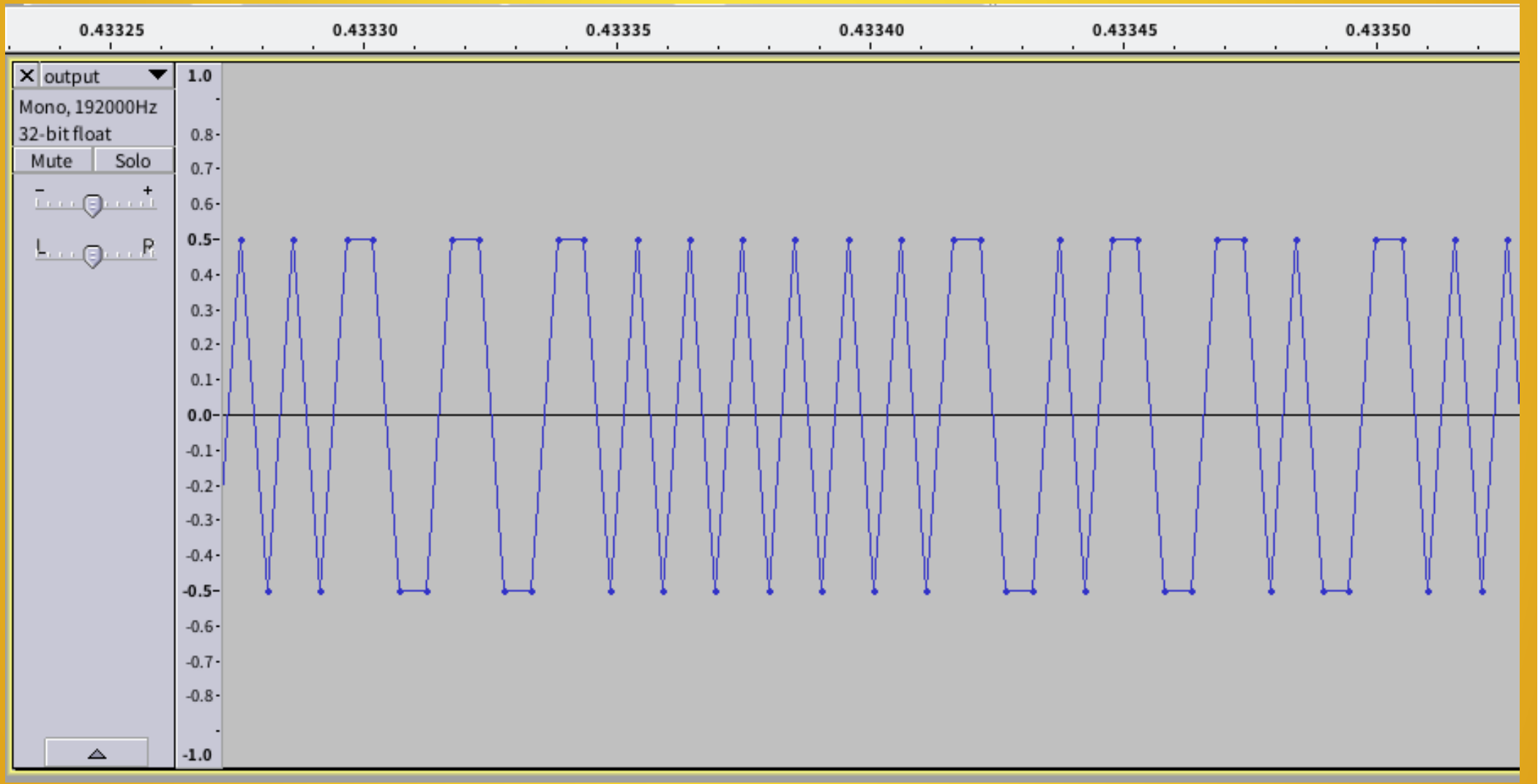


Fig. 1 - Block Diagram Sigma Delta ADC

Delta-Sigma sample input



Delta-Sigma sample output



Delta-Sigma in MATLAB

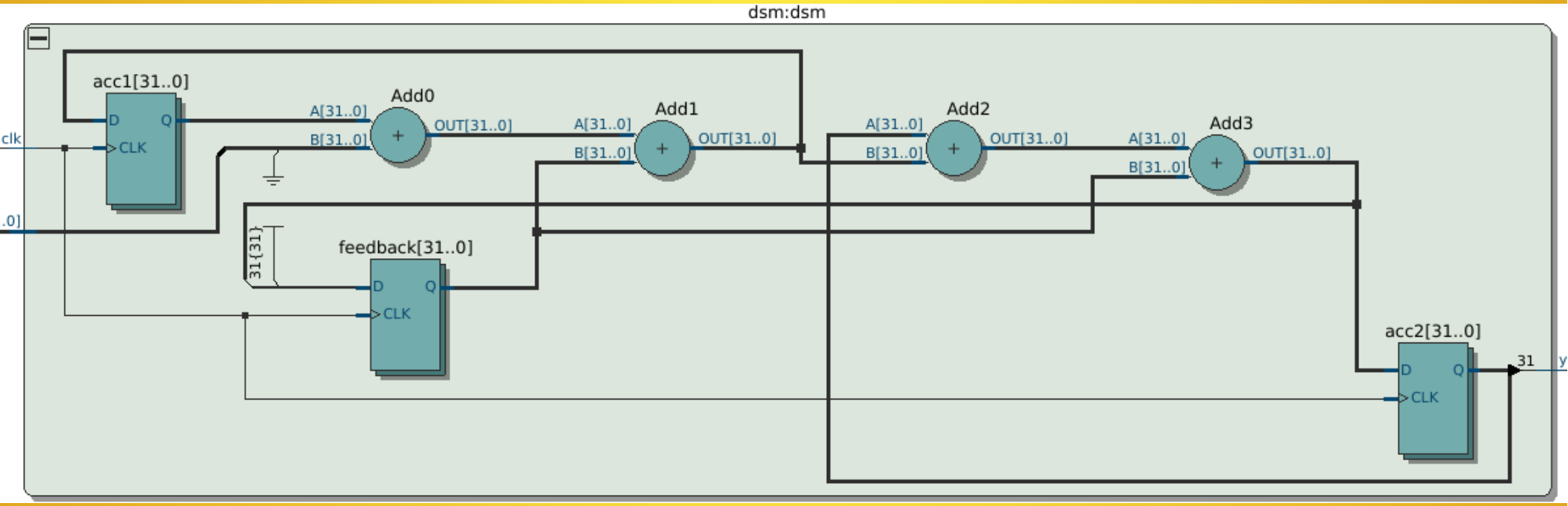
```
acc1 = 0;
acc2 = 0;
feed = 0;
for i = 2:length(wave_out)
    acc1 += wave_in(floor((i-1)*fs/sr)+1, 1) - feed;
    acc2 += acc1 - feed;
    feed = sign(acc2);
    wave_out(i, 1) = feed * 0.5;
    if mod(i, sr) == 0
        display([num2str(i/sr), total]);
    end
end
```

Delta-Sigma in Verilog

```
module dsm(
    input clk,
    input signed [15:0] pcm,
    output y
);
    wire signed[31:0] compress = {{16{pcm[14]}}, pcm[14:0], 1'b0};
    reg signed[31:0] acc1 = 0;
    reg signed[31:0] acc2 = 0;
    reg signed[31:0] feedback = 0;

    always @(posedge clk) begin
        acc1 = acc1 + compress + feedback;
        acc2 = acc2 + acc1 + feedback;
        feedback <= acc2[31] ? 32767 : -32767;
    end
    assign y = acc2[31];
endmodule
```

Delta-Sigma in RTL



Please check out the video

Thank you!

Star Brilliant
4 July 2016