THE UNIVERSITY OF TEXAS AT DALLAS

ERIK JONSSON SCHOOL OF ENGINEERING & COMPUTER SCIENCE

DATA CONVERTERS
(EECT 7327 - SPRING 2018)

HOMEWORK-2

A 100 MSPS Segmentation DAC with DEM Thermometer code and Gray code structure

PROJECT REPORT SUBMITTED BY:

DEVANG SANKHALA	**Net-id: dgs150030**
BHARGAV PATEL	**Net-id : bbp160230**

TABLE OF CONTENTS

1. INTRODUCTION:

Pressure to reduce cost in mass market communication devices such as cable modems and digital cable set-top boxes has created a need for embedded high-speed high resolution DACs. DACs are widely used in automatic test equipment(ATE) systems such as arbitrary waveform generators (AWG) where clean DAC output is required for high frequency signal generation. Since with performance of DAC area is also an important parameter, lots of research has been made to minimize area and make optimum performance of DAC. It is well known to have a segmentation technique to improve parameters of DAC such as resolution, area and largest to smallest component spread. Since MSBs change significantly affects performance we can have a sub-DACs by which generations of MSB and LSBs can be separated. Current steering DACs are most popular for high frequency applications.

When it comes to the binary DAC, a disadvantage cannot be ignored in many applications, which is the glitch affecting the performance of D/A conversion. Changing a binary code from a value to its next value, for example, a binary code is changing between 011 (3decimal) and 100 (4 decimal). Then all three bits have to be reversed. Applications where DACs work at a value near the middle point, in which a number of switching can lead to noise and performance degradation. For example, when input data is converting from 0111 to 1000, the switching waveform may experience a transient state of 0000 instead of directly change from 0111 to 1000. As a result, the DAC output might show a deep spike. This glitch effect will be a serious problem for some applications such as graphic display as well as some analog /mixed-signal/RF device testing.

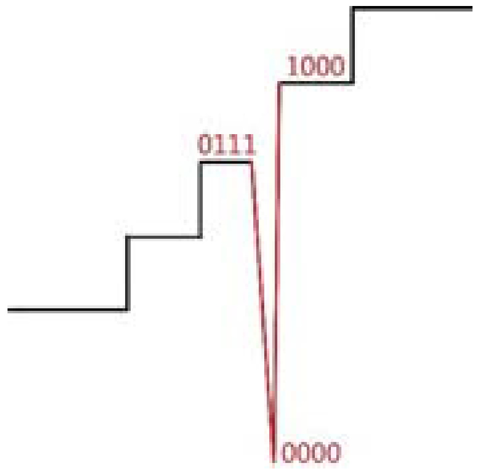


Fig.1 Change of MSB causing Glitch in conversion output

Table 1. Comparison of binary and Gray code

Decimal numbers	Natural Binary Code	4-bit Gray Code
0	0000	0000
1	0001	0001
2	0010	0011
3	0011	0010
4	0100	0110
5	0101	0111
6	0110	0101
7	0111	0100
8	1000	1100
9	1001	1101
10	1010	1111
11	1011	1110
12	1100	1010
13	1101	1011
14	1110	1001
15	1111	1000

Gray-code (reflected binary code) is a binary system where two successive values differ in only one bit . So a Gray-code driven DAC is capable to reduce the glitch. The conversion between binary-code and

Gray-code can be easily realized by XOR logic as followed (in 4-bit case): Binary code: B3, B2, B1, B0 and Gray code: G3, G2, G1, G0

$$G2=B3 \oplus B2,\ G1=B2 \oplus B1,\ G0=B1 \oplus B0$$
$$B3=G3,\ B2=G3 \oplus G2, B1=G3 \oplus G2 \oplus G1, B0=G3 \oplus G2 \oplus G1 \oplus G0$$

Segmented DAC is combination of unit-Element and Binary-Weighted or Gray-Weighted DAC. As shown in Fig.2, The DAC is divided into two sub-DAC's, one for the MSB's and one for the LSB's. Thermometer coding is used in the MSB where the glitches has to be minimum and accuracy has to be maximum. The LSB section can either be done using the binary-weighted or Gray-coded approach. Here Gray coded approach has been taken for implementation since it has minimum glitch performance compared to binary weighted implementation as decribed above.

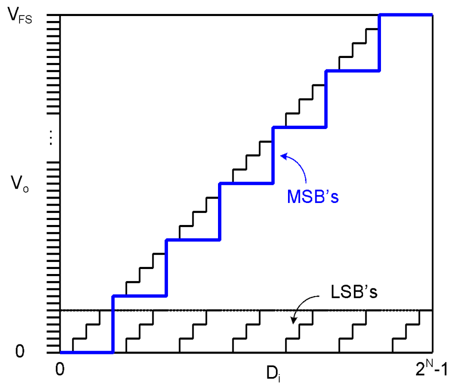


Fig.2 Segmented DAC conversion

2. GRAY CODE CURRENT STEERING DAC:

Gray-code input current-steering DAC replaces the switch array (which conventional binary weighted DAC implements) with dpdt array and sets one more binary-distributed current sources array. As for the output, the subtraction of Iout- and Iout+ called Iout is defined as the output.

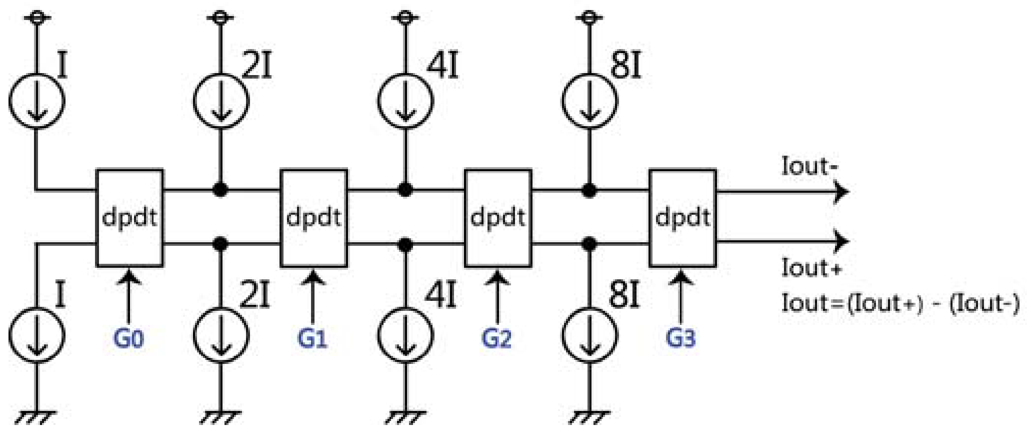


Fig.3 Gray code input current steering DAC

In Gray-code input current-steering DAC is input binary-code are converted to Gray-code through XOR logic, and Gray-code will steer the dpdt to guide the binary-code weighted current sources. Thus Gray-code will control the switching of current sources while rest parts are compatible to conventional binary-code DAC architecture.

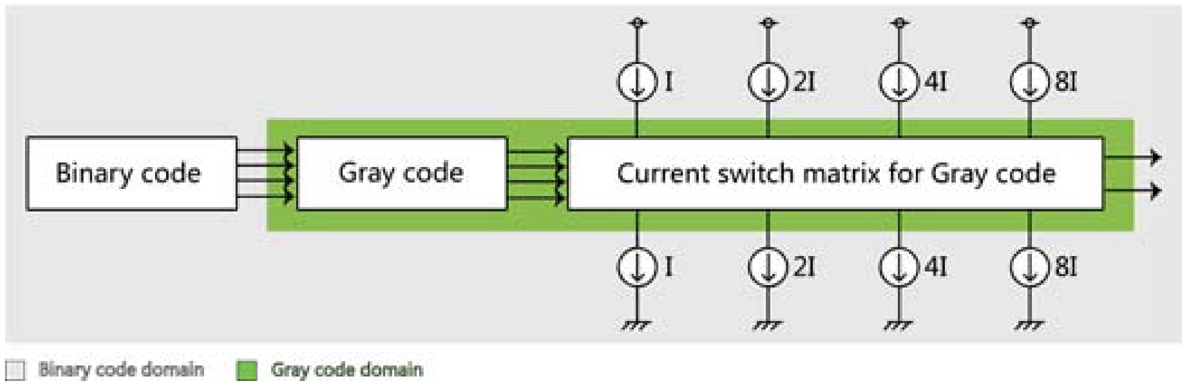


Fig.4 Architecture of Gray Code input current mode DAC

Fig. 8 shows an example of how the Gray-code input current-steering DAC operation when the input data=5. Three of the dpdt switches are set to be in cross connection state and one dpdt switch is in parallel connection state. Thus Iout-=(2I+8I)- (1I+4I)=5I, Iout-=(1I+4I)-(2I+8I)=-5I and Iout=(Iout+)-(Iout-)=-10I.

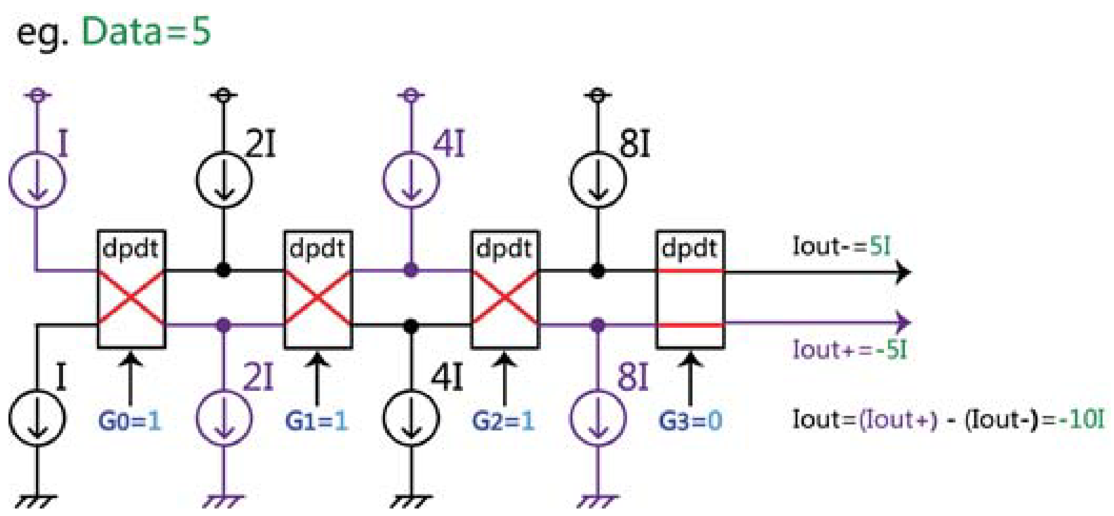


Fig. 5 A Gray-Code input Current-mode DAC. (For Data=5)

Implementation of Gray-Code input Current-mode DAC :

For a **12-bit DAC**, segmentation of '**8+4**' with **8 bits** as MSB section of **Thermometer coded** current steering DAC and **4 bits** as LSB section of **Gray code** input current-mode DAC. **Load** for this DAC has been taken as **20mA**.
Implementation of current-steering Gray-code DAC starts with a synchronous binary-code initiator. Then the binary-code converted to Gray-code by XOR logic and can be stored at Latch array in hardware implementation. After D/A conversion, Iout can be
obtained from Iout- and Iout+ by passing through a suitable load. Here, load for DAC is taken as 20mA, hence

$$LSB_{SegDAC} = \frac{I_{load}}{2^{N}} = \frac{20\ mA}{2^{12}} = 5\mu A$$

$$LSB_{thermoDAC} = \frac{I_{load}}{2^{N_{thermo}}} = \frac{20\ mA}{2^{8}} = 80\mu A$$

$$LSB_{GrayDAC} = \frac{LSB_{thermoDAC}}{2^{N_{gray}}} = \frac{80\mu A}{2^{4}} = 5\ \mu A$$

DPDT switch Implementation:

By considering 20mV drop across each switch in ON-state, sizing of switches have been decide.

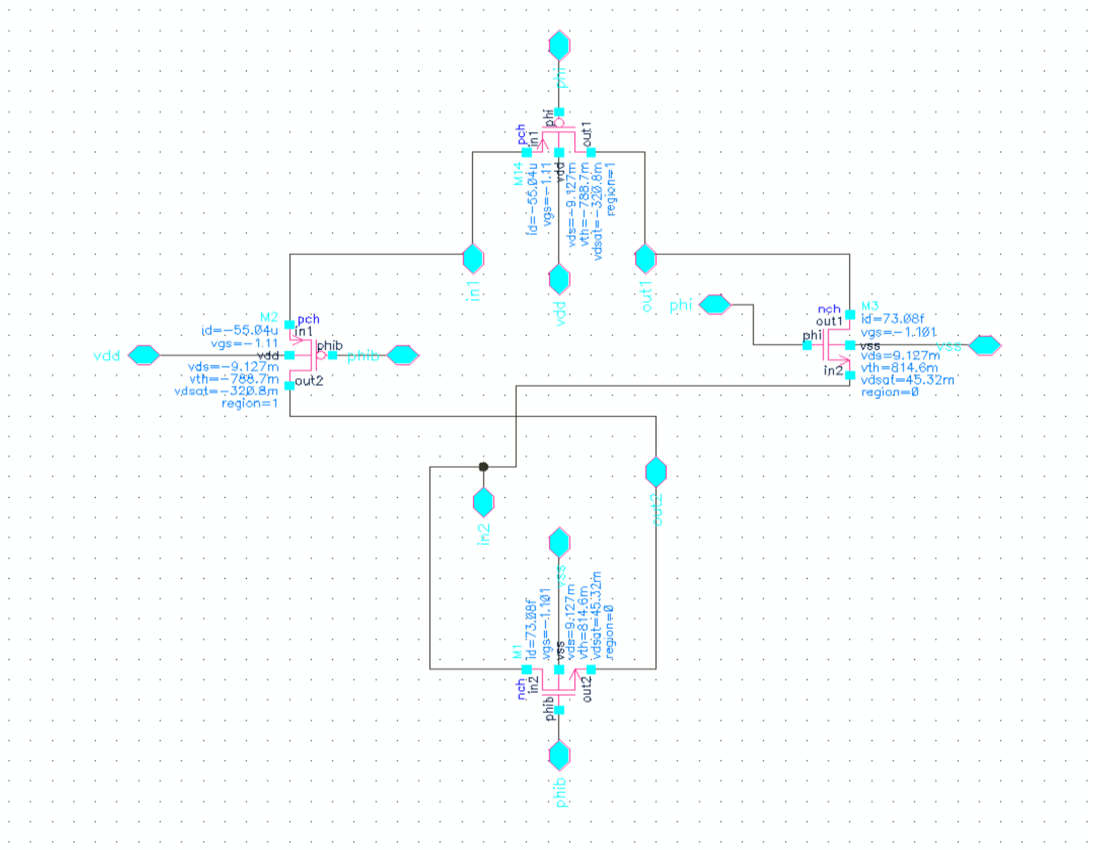


Fig.6 DPDT switch implementation at MOSFET level

Block were implemented for Generation of synchronous binary counter and conversion of Binary to Gray code with VerilogA. 4-bit implementation of Gray Coded DAC is given in Fig.7. Here Current sources are taken with current mirror for 4 bits depending upon scale.

Circuit Level Implementation of 4 bit Gray Coded DAC:

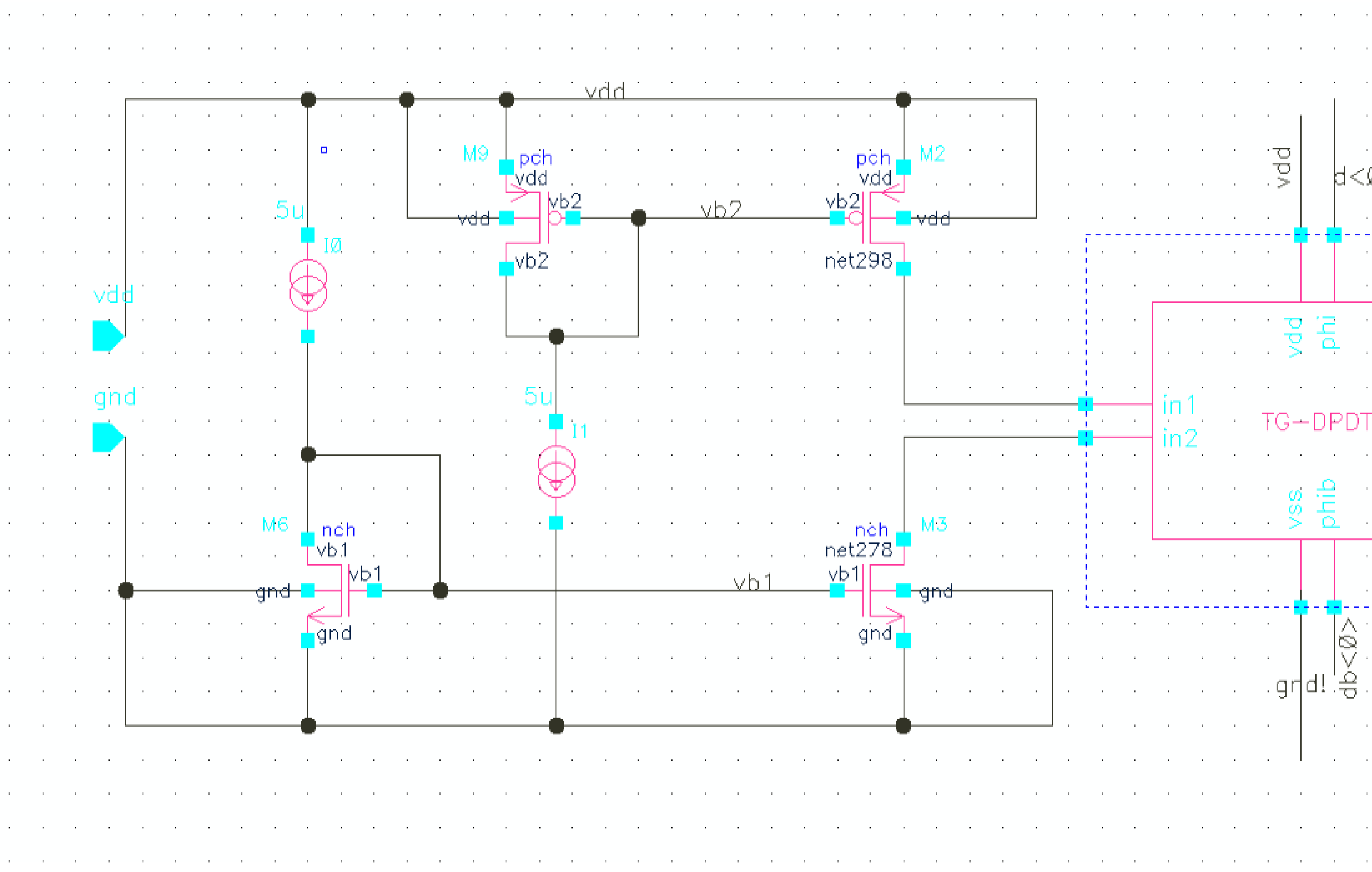


Fig.7 Bias Circuit and First stage of Gray coded DAC (4-bit)

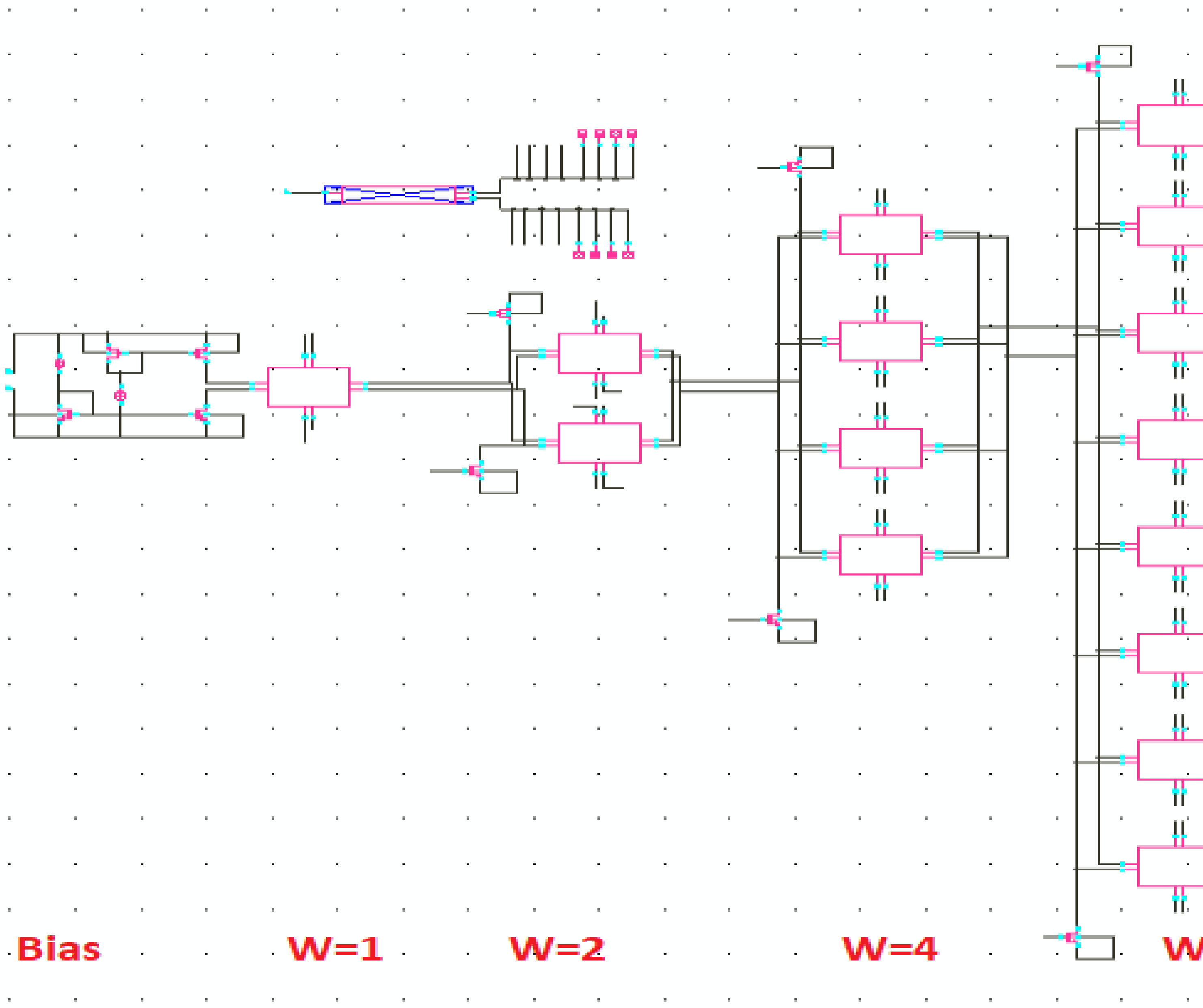


Fig.8 Schematic of 4 bit Gray Coded DAC

Simulation Results:

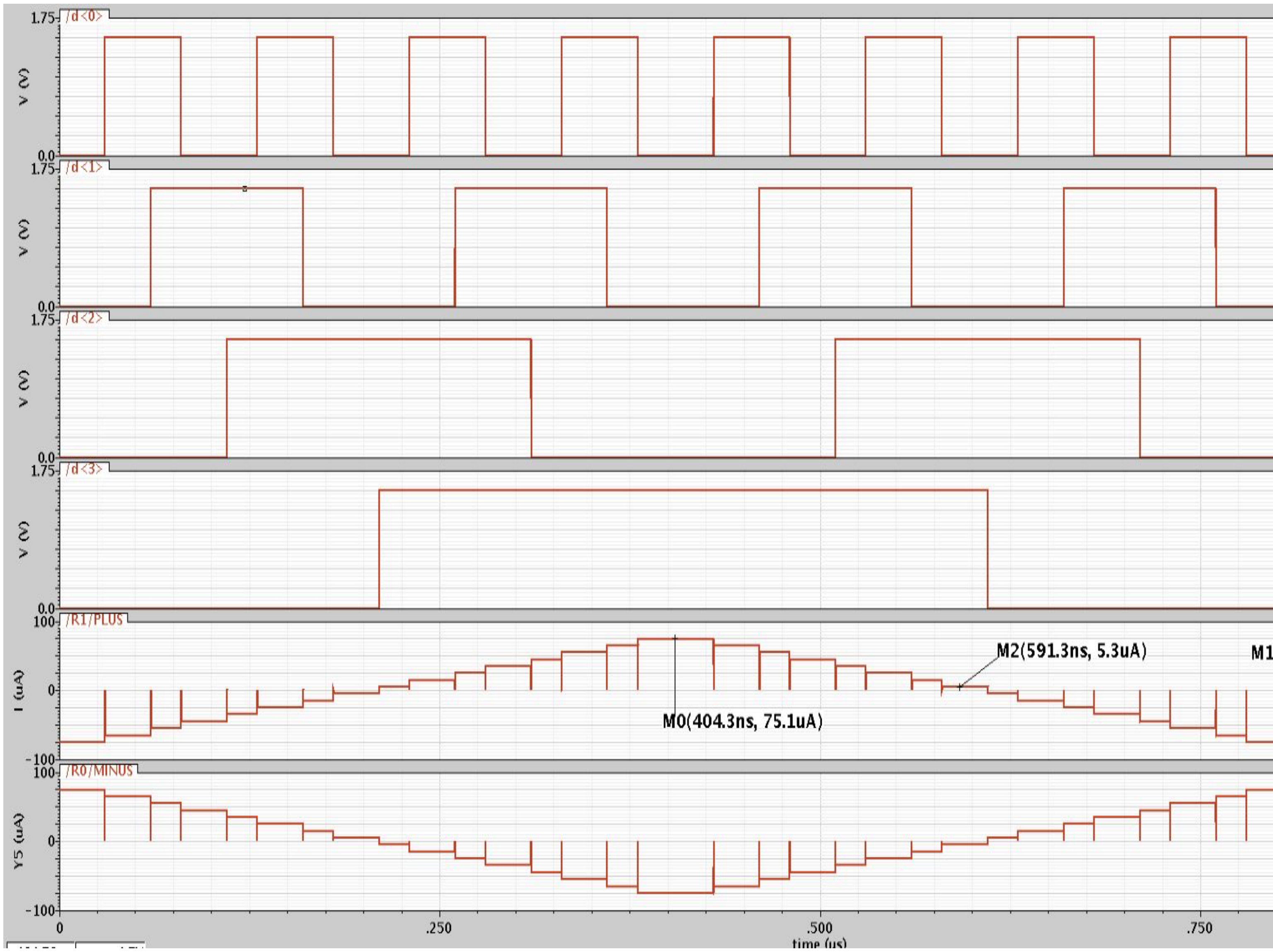


Fig.9. 4 bit Gray code conversion Output

3. SEGMENTED DAC USING RANDOMIZER AND GRAY CODE:

Thermometer Coded DAC:

Fig. 4(a) shows an example of a 10-bit thermometer-coded DAC. There are unit current sources. Each unit current source is connected to a switch controlled by the signal coming from the binary-to-thermometer decoder. When the digital input increases by 1 LSB, one more current source is switched from the negative to the positive side. Assuming positive only current sources, the analog output is always increasing as the digital input increases. Hence, monotonicity is guaranteed using this architecture.

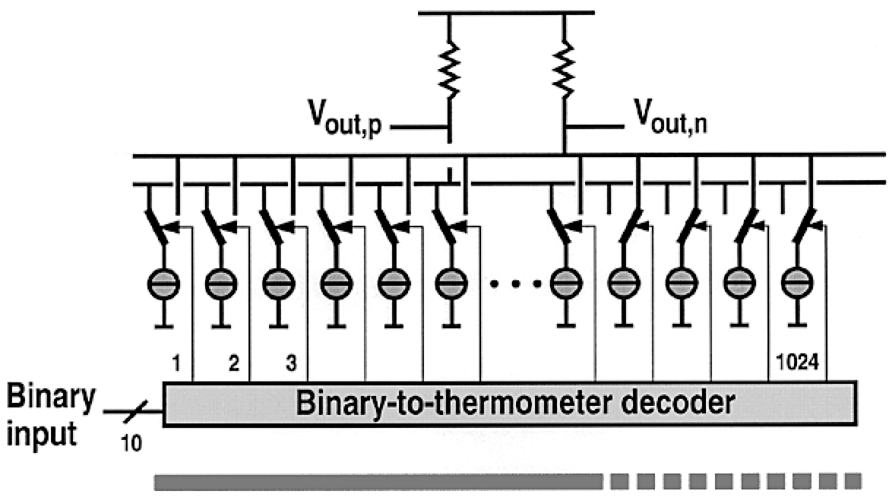


Fig.10 Thermometer Coded DAC

At the mid code, a 1-LSB transition (0 111 111 111 → 1 000 000 000), causes only one current source to switch as the digital input only increases by one. This will greatly reduces the glitch problem. Glitches hardly contribute to nonlinearity in the thermometer-coded architectures. This is because the magnitude of a glitch is proportional to the number of switches that are actually switching.

Implementation of **8-bit** Thermometer DAC using DEM:

The design of high dynamic performance current-steering DAC especially with high spurious-free dynamic range (SFDR) value, requires significant effort in minimizing mismatch-induced nonlinearities . The foreground calibration technique is one of the mismatch compensation methods . By making use of the measured mismatch values, the linearity is improved by providing tuning current or rearranging the sequence of the current sources. However, this technique suffers from deteriorated dynamic performance at high frequency and large area penalty.

In contrast, the background calibration technique, especially the dynamic-element-matching (DEM) technique, effectively suppress the negative effects of mismatch induced nonlinearities on the DAC's dynamic performance without the above-mentioned drawbacks .

In a DEM DAC, the unary current sources are randomly selected. The harmonic distortions are reduced due to the averaging of mismatch.

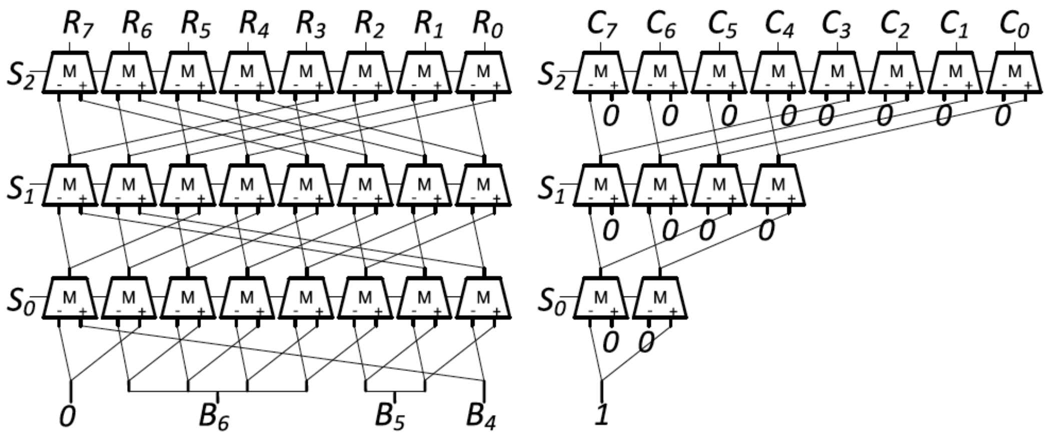

R_7 R_6 R_5 R_4 R_3 R_2 R_1 R_0
C_7 C_6 C_5 C_4 C_3 C_2 C_1 C_0
S_2
S_1
S_0
M
0
B_6
B_5
B_4
1

Fig.11 Schematics of MSB rotator and generation of Row-Columns by DEM.

6-bit DAC example is presented for illustrating the thermo-coded current DAC structure and the corresponding DEM implementation. The total six bits B5- B0 are split into three MSB and three LSB bits for segmentation.
For 8-bit Thermometer DAC this logic has been implemented in VerilogA block and random numbers for Row and Columns are generated so that out of 256 cells depending upon weightage of given thermometer code, cells would be selected for positive current generation as well as for negative current generation. Implemented block for Randomizer is shown in Fig.9.

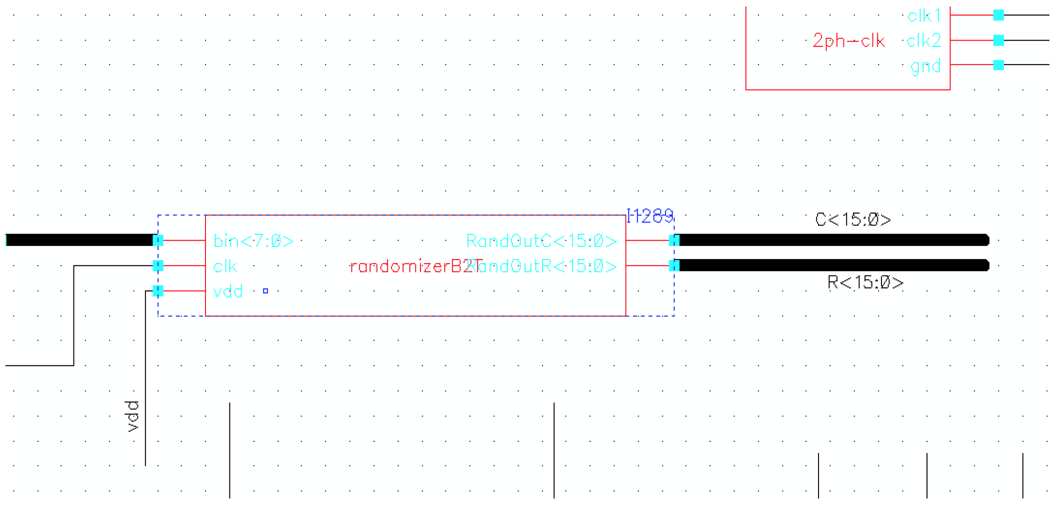


Fig.12 Randomizer Block

Current Cell Design:

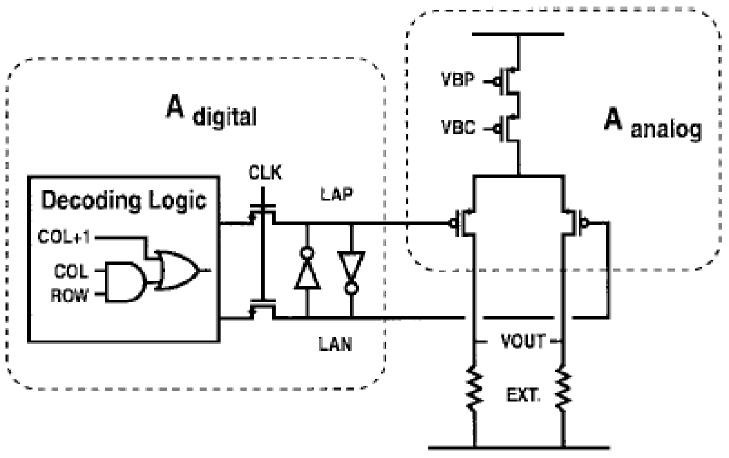


Fig.13 Unit Current Cell

Fig. 10 shows the circuit of one unit current cell. It consists of an analog part and a digital part. The analog part consists of a differential switch and a cascoded current source. The digital part consists of a decoding logic and a latch. The decoding logic is equivalent to an AND –OR gate function and the latch is essential for timing synchronization, as all the current cells should switch at the same time.

Circuit Implementation of cascoded current source:

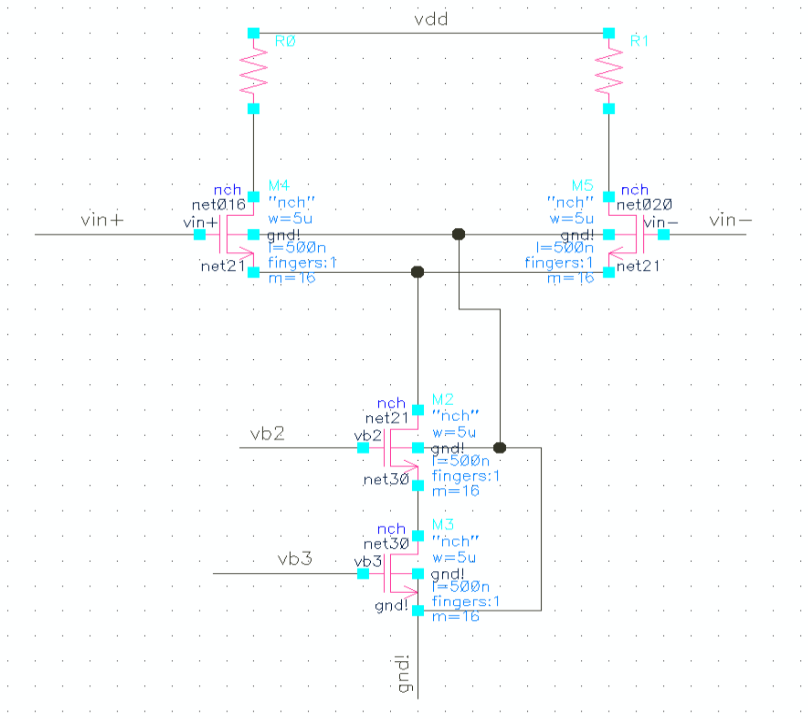


Fig. 14 Cascoded Current source (Differential Output)

MOSFET Sizing for Cascoded current source:

MOSFET	W (μm)	L (μm)	M(multiplier)
M2	5	0.5	16
M3	5	0.5	16
M4	5	0.5	16
M5	5	0.5	16

LSB current for 8 bit Thermometer DAC would be 80 μA. Current Source is Designed with 80 μA diff. switching output current.

Switching Current Waveforms:

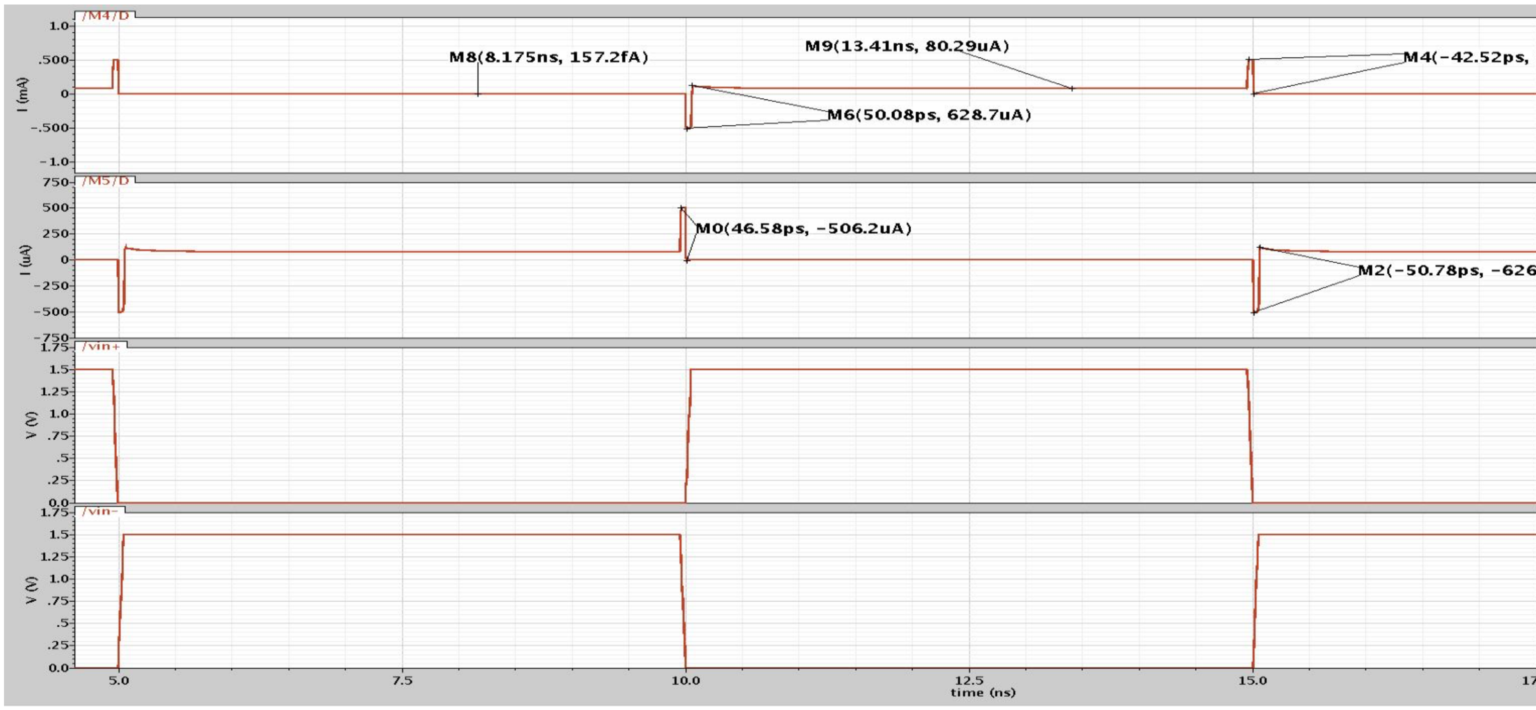


Fig. 15 Current source Switching Waveforms

Decoding Logic is equivalent to AND –OR Logic and Every unit cell will have a latch driving the current source input transistors. Circuit Implementation of AND-OR logic and latch is given in Fig.13.

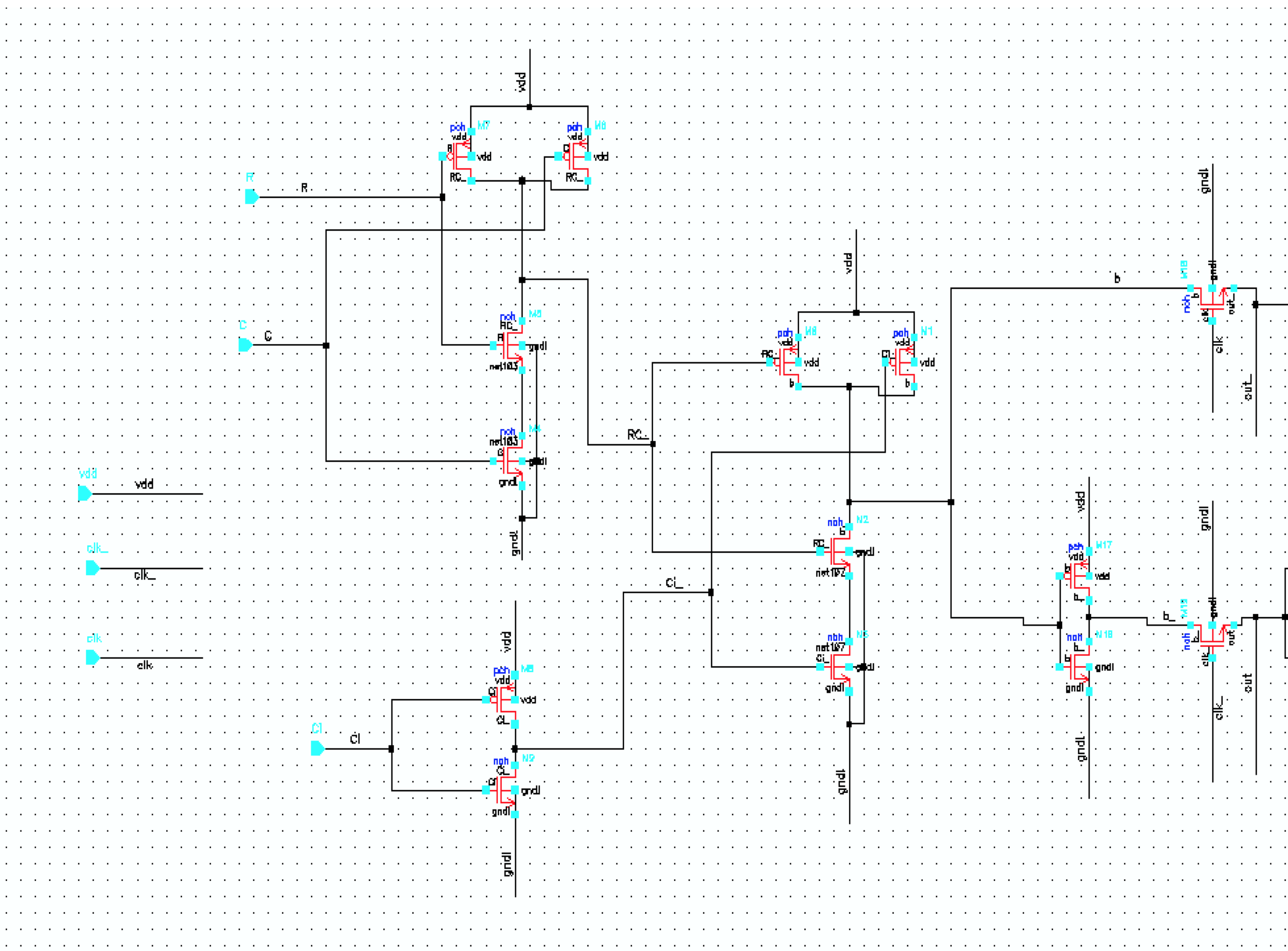


Fig.16 Circuit Implementation of AND-OR Logic and Latch (Digital Circuit)

MOSFET Sizing for Digital Circuit:

MOSFET	W (µm)	L (µm)	M(multiplier)
M1	5	0.35	100
M2	5	0.35	100
M3	5	0.35	90
M4	5	0.35	90
M5	5	0.35	180
M6	5	0.35	180
M7	5	0.35	200
M8	5	0.35	200
M9	5	0.35	200
M10	5	0.35	100
M11	5	0.35	4
M12	5	0.35	20
M13	5	0.35	20
M14	5	0.35	4
M15	5	0.35	20
M16	5	0.35	100
M17	5	0.35	8
M18	5	0.35	8

Sizing of MOSFET were decided based on Logical effort with load capacitance of 80fF which is input capacitance of Cascoded current source.

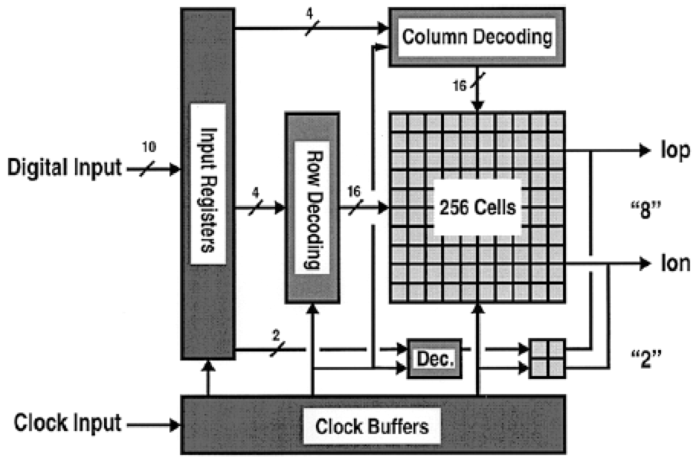


Fig. 17 Block Diagram of 8+2 Bit segmentation

With the help of shown Block diagram in Fig.14 of 8+2 bit segmentation, implementation of 8+4 bit segmentation was carried out. The digital inputs are first clocked into input registers. Then the first four MSB's are column decoded, the next four bits are row decoded, and the final two bits are sent to the decoding logic for the 2-bit LSB section.

Here, By implementation of DEM, Random numbers for Rows (R0-R15) and Columns (C0-C15) will be generated by Randomizer and will be given to driver cell comprised of digital Circuit and latch (driver). **Delay of Driver cell** was around **1ns** in Worst case delay measurements with 100MHz clock input.

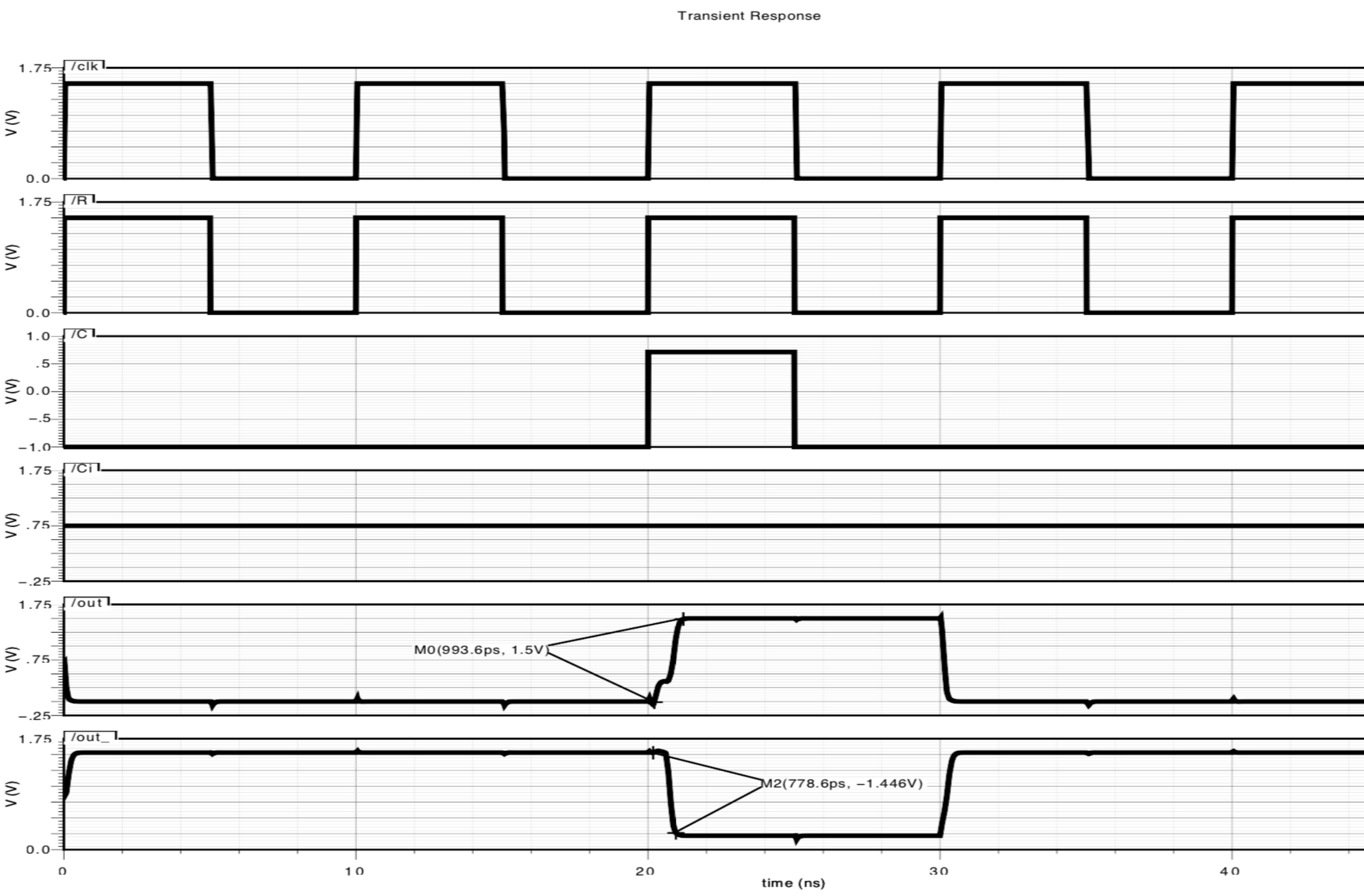


Fig.18 Driver Cell Output (R=1,C=1,Ci=0 with 100MHz of Clock)

With both possible options when driver cell is driving either positive current(I_{out+}) of negative current(I_{out-}) maximum power consumption within 100 clock cycle is 0.819 mW.

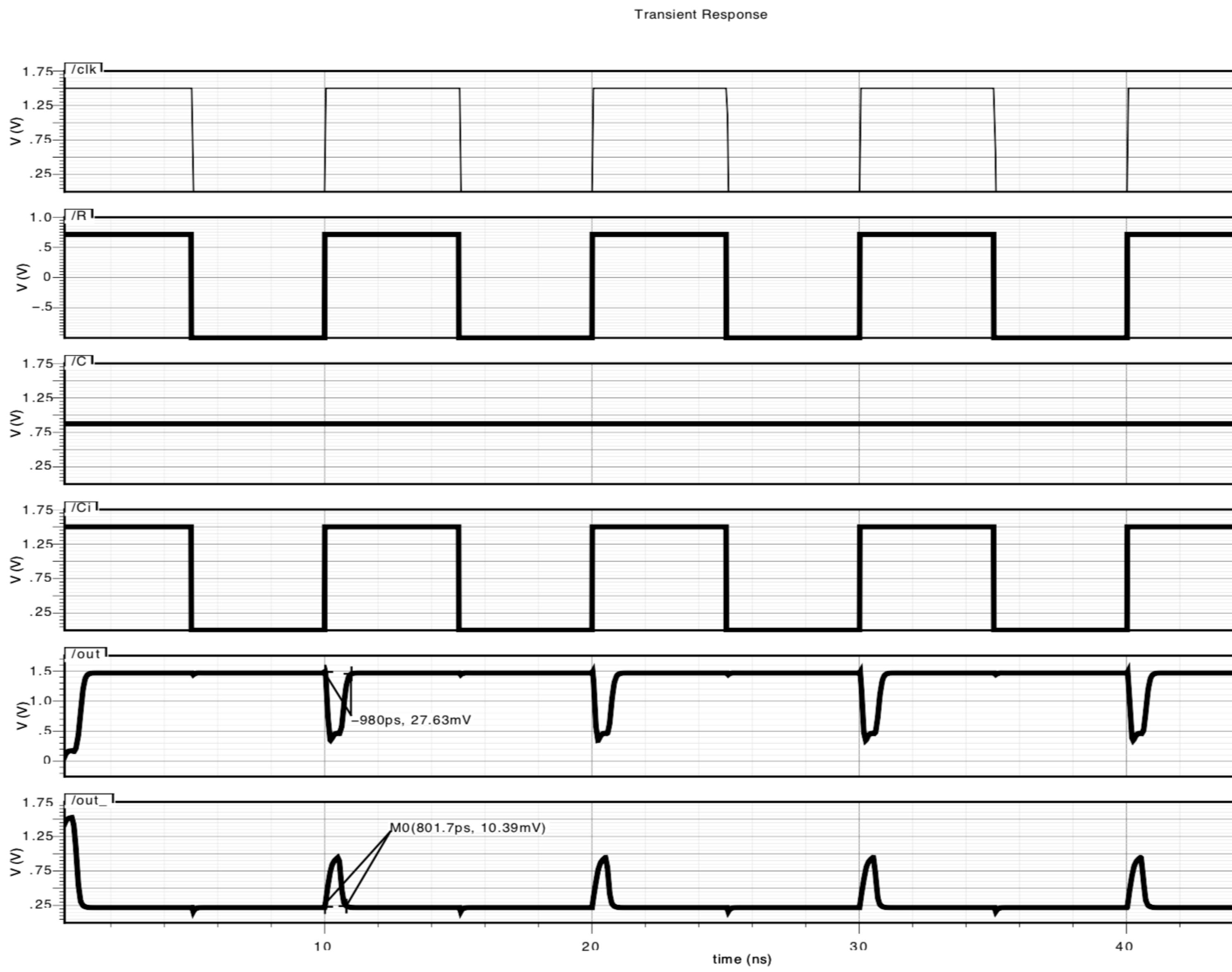


Fig.19 Driver Cell Output (R=1,C=0,Ci=1 with 100MHz Clock)

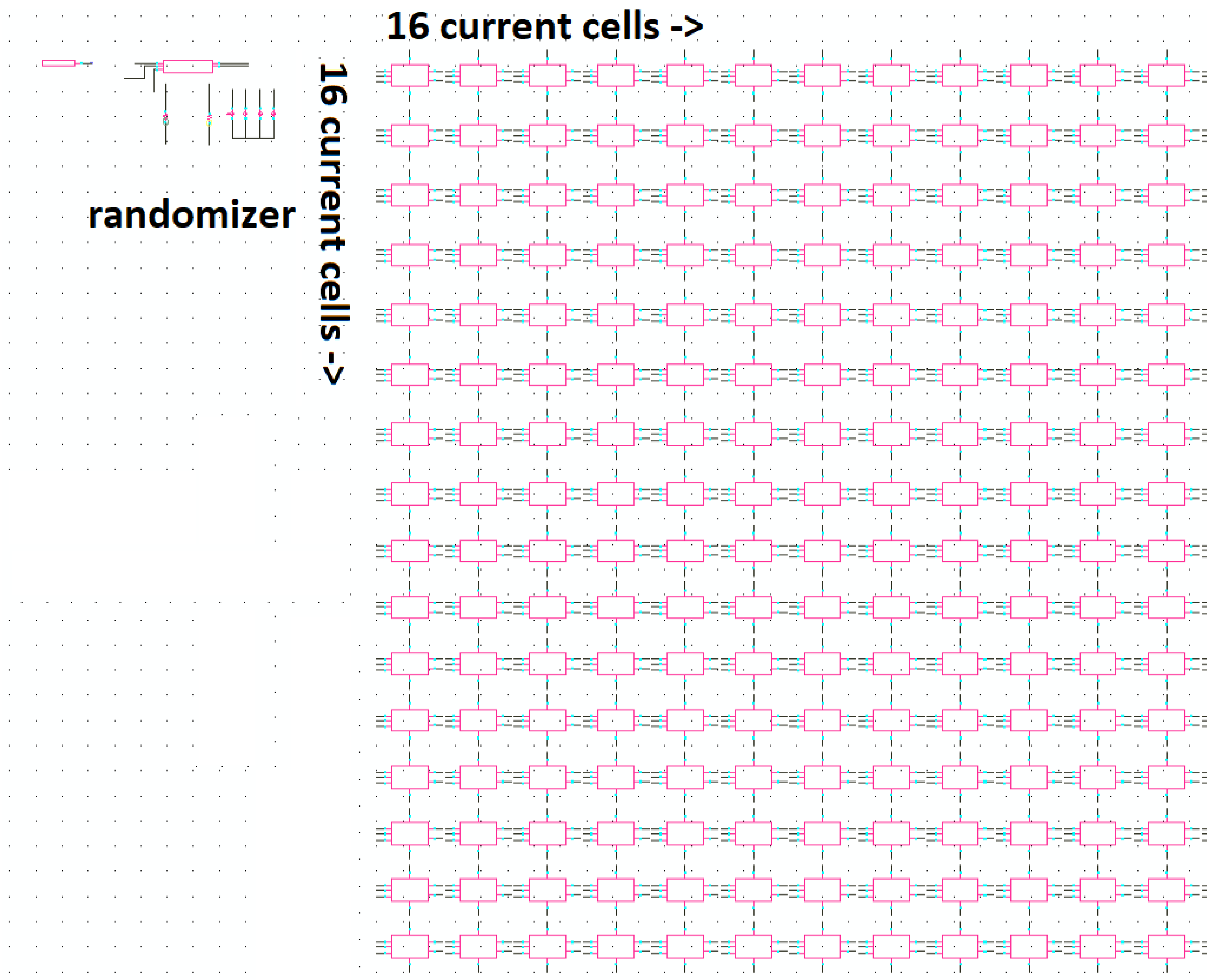


Fig.20 Schematic of 8 bit Thermometer DAC (256 Current Cell)

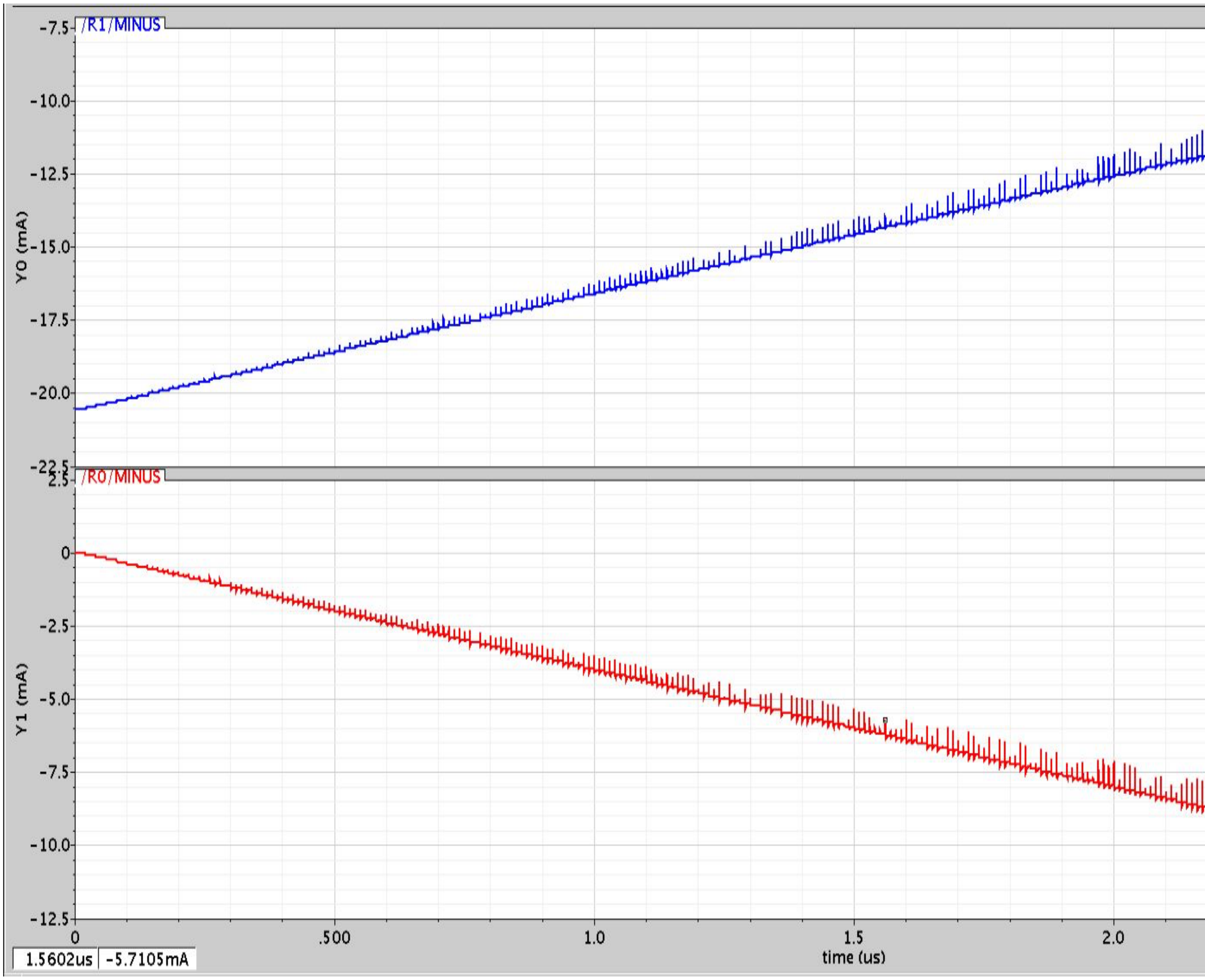
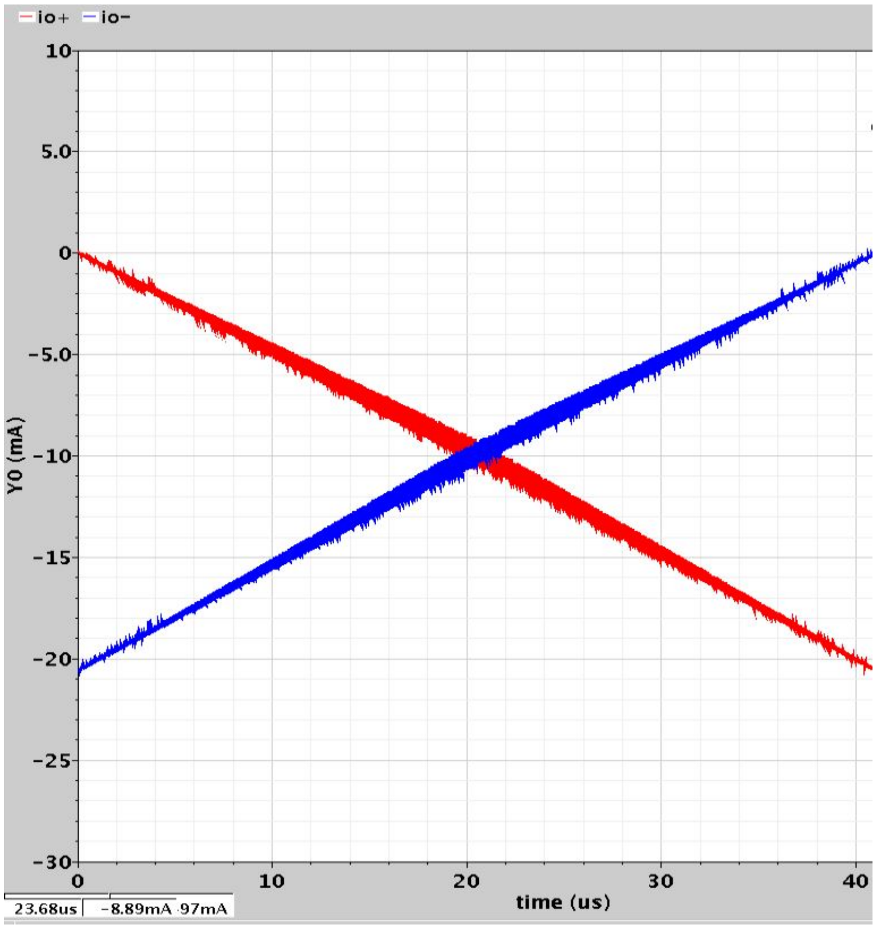


Fig.21 ThermometerDAC (8 Bit) Conversion output with 100MHz Clock

*DAC was initialized with all negative current sources ON(Iout- ≈ 20.48 mA)

4. <u>SIMULATION RESULTS:</u>



io+
io−
YO (mA)
time (us)
23.68us
−8.89mA
97mA

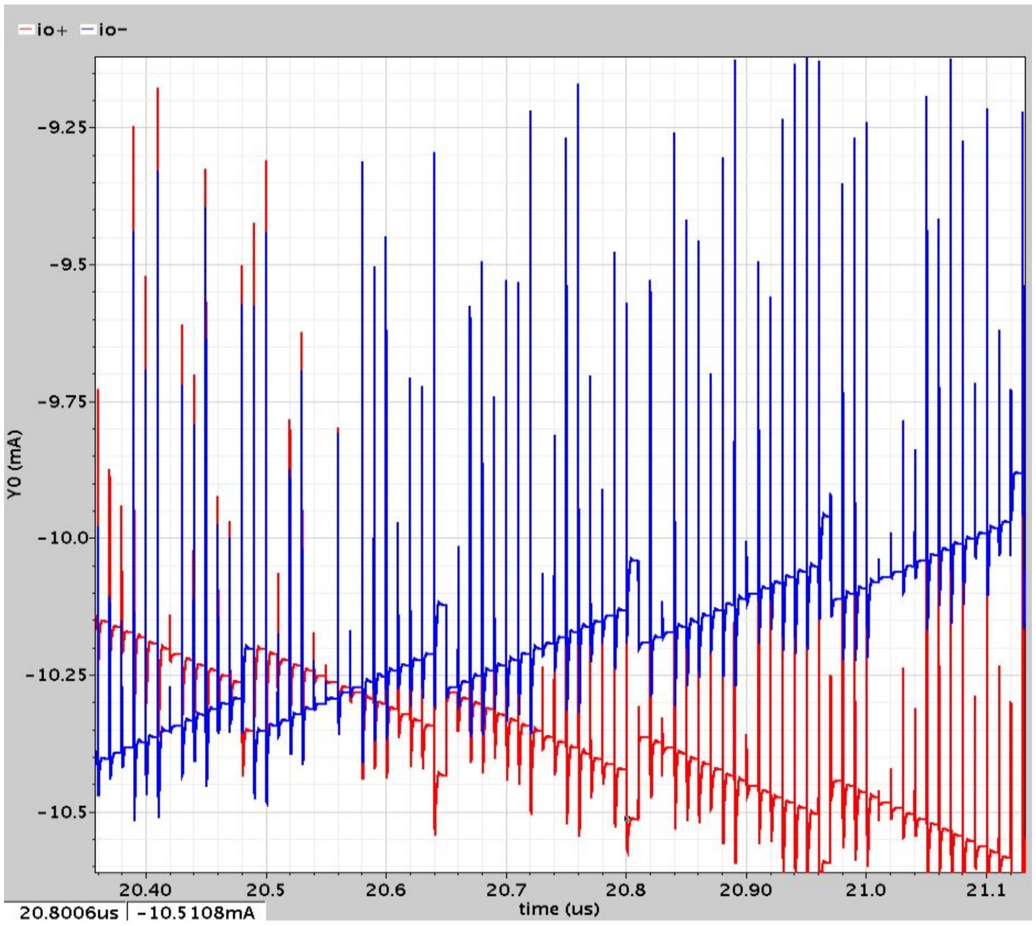
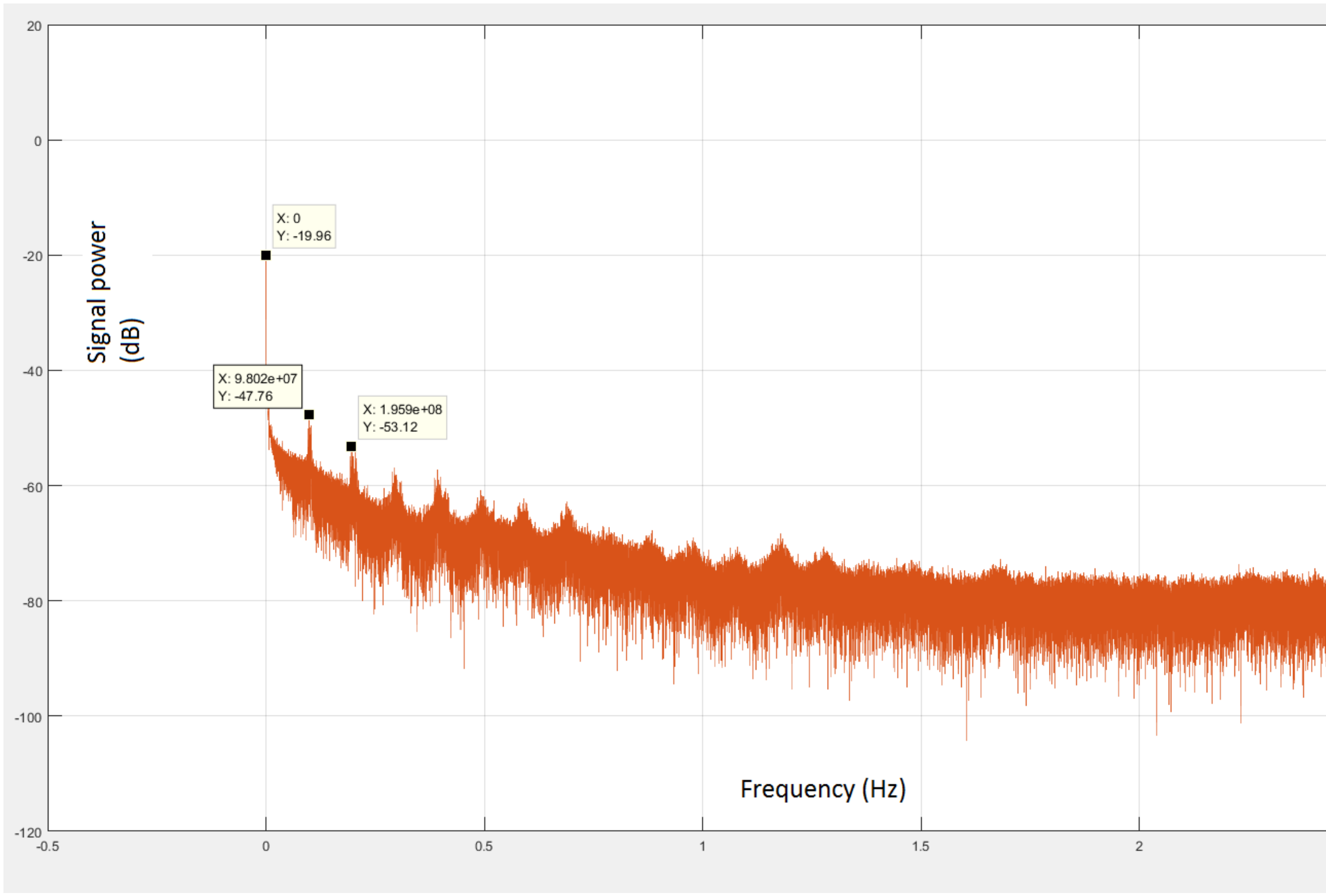


Fig.22 12 bit segmented DAC conversion output

- **INL** measured was **+0.6 LSB**. **DNL** measured was **+0.3 LSB**.

FFT Analysis of Output :



SFDR for given output = 27.79 dB. ENOB= 9.68 bits ≈ 10 bits.

5. REFERNCES:

- iang, R., Adhikari, G., Sun, Y., Yao, D., Takahashi, R., Ozawa, Y., ... & Shiota, R. (2017, November). Gray-code input DAC architecture for clean signal generation. In *Intelligent Signal Processing and Communication Systems (ISPACS), 2017 International Symposium on* (pp. 669-674). IEEE.
- Lin, C. H., & Bult, K. (1998). A 10-b, 500-MSample/s CMOS DAC in 0.6 mm/sup 2. *IEEE Journal of Solid-State Circuits*, *33*(12), 1948-1958.
- Mao, W., Li, Y., Heng, C. H., & Lian, Y. (2018). High Dynamic Performance Current-Steering DAC Design With Nested-Segment Structure. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*.
- Class notes – Prof. Jin Liu, UTdallas.
- Class notes- Prof. Yun Chiu, UTdallas

APPENDIX:

Code for Measuring DNL using VerilogA block:

```
`include "discipline.h"
`include"constants.h"


module dac_dnl_12bit(vin, vd1, vd0);
electrical vin,  vd1, vd0;
parameter real tsettle=2n from (0:inf);
parameter real vlogic_high=1.5;//DGS modified
parameter real vlogic_low=0.0;
parameter integer log_to_file = 0;

//DGS modified
`define NUM_DAC_BITS 12
`define NUM_CODES 4096
//DGS modified

  integer out_file;

  real    vd_val[0:`NUM_DAC_BITS-1];

  real    code_vout[0:`NUM_CODES-1];
  real    dnl[0:`NUM_CODES-1];

  integer code_mask[0:`NUM_DAC_BITS-1];
  integer bit, mask;

  integer just_finished; // flag that says measurement finished during
                      //  this evaluation
  integer code;
```

```
real    tnext;

real    width_expect;

real    max_dnl;
integer max_dnl_code;


analog begin

  @ ( initial_step ) begin
    tnext = tsettle;
    max_dnl_code = -1;
    mask = 1;
    for (bit=0; bit < `NUM_DAC_BITS; bit=bit+1) begin
      code_mask[bit] = mask;
      mask = mask * 2;
      vd_val[bit] = vlogic_low;
    end
  end

  @ ( timer(tnext)) begin // at a measurement point

    // record the last measured value
    code_vout[code] = V(vin);

    code = code + 1;
    if (code < `NUM_CODES) begin
      tnext = tnext + tsettle;
    end else begin           // finished
      code = `NUM_CODES  - 1;
      just_finished = 1;
    end

    // to convert a code into a bit pattern
    for (bit=0;bit<`NUM_DAC_BITS; bit=bit+1) begin
      if (code & code_mask[bit]) begin
        vd_val[bit] = vlogic_high;
      end else begin
        vd_val[bit] = vlogic_low;
      end
    end
  end

  if (just_finished) begin  // calculate the dnl function
    just_finished = 0;
    width_expect = (code_vout[`NUM_CODES-1] - code_vout[0])
               / (`NUM_CODES - 1);
      for (code=0; code < `NUM_CODES-1; code = code+1) begin
        dnl[code] = (code_vout[code+1] - code_vout[code] - width_expect)
                / width_expect;
        if (max_dnl < abs(dnl[code])) begin
          max_dnl = abs(dnl[code]);
          max_dnl_code = code;
        end
      end

      $strobe("The maximum dnl measured is %f at code %d",max_dnl,
          max_dnl_code);

      // log the results to a file if desired
      //
      if (log_to_file) begin
        out_file = $fopen( "%C:r.dat" );
        $fstrobe(out_file,"# Generated by Spectre from `%M'");
        for (code=0; code < `NUM_CODES-1; code=code+1) begin
           $fstrobe(out_file,"%d\t%f",code,dnl[code]);
        end
        $fclose(out_file);
      end
    end

    V(vd0) <+ vd_val[0];
    V(vd1) <+ vd_val[1];


    @ ( final_step ) begin
      if (log_to_file) $fclose(out_file);
    end
  end
endmodule
```

Code for Measuring INL using VerilogA block:

```
include "discipline.h"
`include"constants.h"


module dac_inl_12bit(vin, vd1, vd0);
electrical vin, vd1, vd0;
parameter real tsettle=2n from (0:inf);
parameter real vlogic_high=1.5;
parameter real vlogic_low=0.0;
parameter integer log_to_file = 0;
```

```
`define NUM_DAC_BITS 12
`define NUM_CODES 4096

  integer out_file;

  real    vd_val[0:`NUM_DAC_BITS-1];

  real    code_vout[0:`NUM_CODES-1];
  real    inl[0:`NUM_CODES-1];

  integer code_mask[0:`NUM_DAC_BITS-1];
  integer bit, mask;

  integer just_finished; // flag that says measurement finished during
                     //  this evaluation
  integer code;

  real    tnext;

  real    width_expect;

  real    max_inl;
  integer max_inl_code;


  analog begin

   @ ( initial_step ) begin
      tnext = tsettle;
      max_inl_code = -1;
      mask = 1;
      for (bit=0; bit < `NUM_DAC_BITS; bit=bit+1) begin
        code_mask[bit] = mask;
      mask = mask * 2;
        vd_val[bit] = vlogic_low;
      end
    end

    @ (timer(tnext)) begin // at a measurement point

      // record the last measured value
      code_vout[code] = V(vin);

      code=code+1;
      if (code < `NUM_CODES) begin
        tnext = tnext + tsettle;
      end else begin          // finished
        code = `NUM_CODES  - 1;
        just_finished = 1;
      end

      // to convert a code into a bit pattern
      for (bit=0;bit<`NUM_DAC_BITS; bit=bit+1) begin
        if (code & code_mask[bit]) begin
          vd_val[bit] = vlogic_high;
        end else begin
          vd_val[bit] = vlogic_low;
        end
      end
    end

    if (just_finished) begin  // calculate the inl function
      just_finished = 0;
      width_expect = (code_vout[`NUM_CODES-1] - code_vout[0])
                / (`NUM_CODES - 1);
      for (code=0; code < `NUM_CODES; code=code+1) begin
        inl[code] = (code_vout[code] -  code*width_expect)/width_expect;
        if (max_inl < abs(inl[code])) begin
          max_inl = abs(inl[code]);
          max_inl_code = code;
        end
      end

      $strobe("The maximum inl measured is %f at code %d",max_inl,
            max_inl_code);

      // log the results to a file if desired
      //
      if (log_to_file) begin
        out_file = $fopen( "%C:r.dat" );
        $fstrobe(out_file,"# Generated by Spectre from `%M'");
        for (code=0; code < `NUM_CODES; code=code+1) begin
           $fstrobe(out_file,"%d\t%f\t%g",code,inl[code],code_vout[code]);
        end

        $fclose(out_file);
      end
    end
    V(vd0) <+ vd_val[0];
    V(vd1) <+ vd_val[1];

    @ ( final_step ) begin
      if (log_to_file) $fclose(out_file);
    end
  end
endmodule
```