EE435 Final Project Report
12-Bit SAR ADC

Presented By:

Xilu Wang, Aaron Pederson, Tao Chen

Department of Electrical and Computer Engineering
Iowa State University

5/11/2015

This page intentionally left blank.

Table of Contents

Project Requirement

Process:	MOSIS ON 0.5 um
Supply:	0V and 5V
A_{in} range:	>= 2.5V, with V_{reftop} and V_{refbot} to be defined by designer if necessary. Default values are 5V and 0V.
Resolution:	12 bits
INL:	±1 LSB
DNL:	±1 LSB
Speed:	Not specified, but should be > 100 KSPS, target 500 KSPS
Power:	Not specified, but should be < 20 mW
Area:	Not specified, but should be < 1 mm^2
Architecture:	Segmented cap array (4 bit binary MSB array + scale down + 8 bit binary LSB array) + high resolution comparator + SAR + logic control

Required pins:
AVDD, AGND (default: these will be used as Vreftop and Vrefbot. If you prefer separate Vreftop and Vrefbot, you can use additional pins.)
DVDD, DGND
Ain
CLK: high speed clock, from which all other clock signals to be generated
OE (output enable)
D0-D11; D0=LSB, D11=MSB; should be in high-z mode when OE is low

Design and Performance to be verified in report and presentation:

- Brief description of system level operation of SAR ADC (including block diagram, SA search algorithm, timing diagram, charge conservation)
- Description of key components (CDAC, comparator, SAR logic, switches), key constraints, and sizing decisions (including ratio critical issues)
- Power, area, speed trade-off decisions
- DC analysis of all key node voltages and branch currents
- Transient analysis of key node voltages in key building blocks during key transitions
- Converter Static Performance: INL, DNL, gain error, offset
- Converter Dynamic Performance: SFDR, SNR, THD, SINAD, ENOB at 3 different frequencies: near 90% Nyquist, near 20% Nyquist, and near 5% Nyquist. Make sure to use coherent sampling. Use 2^13 samples for FFT.

Abstract: In this project report, we are going to present you how the SAR ADC operates by using the binary search algorithm to convert analog input voltage into binary codes. This report will explain some of the key building blocks of the ADC, such as capacitive DAC, high-speed comparator.

1. Introduction

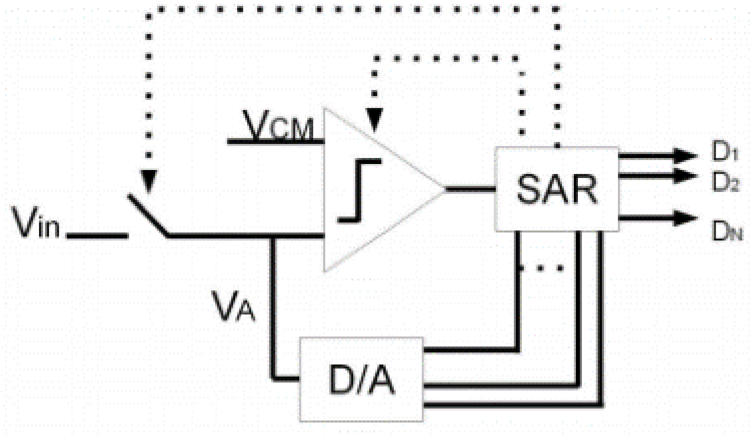


Figure 1. Simplified N-bit SAR architecture.

SAR is frequently used for medium-to-high-resolution applications with sample rates under 5 Msps.The SAR architecture allows for high-performance, low-power ADCs to be packaged in small form factors for today's demanding applications.

Our SAR ADC is implemented by the circuit in figure [1]. There is capacitive DAC which is used to generate analog output voltage; comparator, comparing input voltage with V_{DAC}, and output logic high or low. SAR logic unit is used to output binary codes.

2. C-DAC

2.1 Design Capacitor Array and Switches

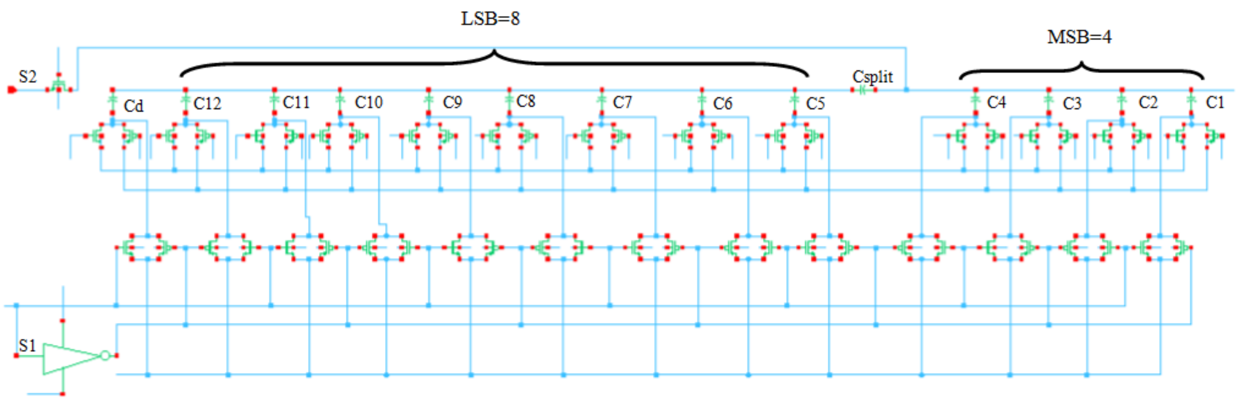


Figure 2. Capacitor array.

A. Capacitor Array Design Strategy

For a 12-bit SAR ADC, we will have 8-bit LSB and 4-bit MSB. Instead of making completely parallel array. A two-stage weighted capacitor array is used here. By adding a split capacitor C_{split} into the circuit, we can reduce the size of the capacitor. If we don't use C_{split} the LSB capacitor might be too small to fabricate.

The sizes for the capacitor array, in referring to Figure [2], from left to right:

	Capacitor	Size (pF)
	C_{dummy}	1/128
LSB	C12	1/128
LSB	C11	1/64
LSB	C10	1/32
LSB	C9	1/16
LSB	C8	1/8
LSB	C7	1/4
LSB	C6	1/2

LSB	C5	1
	C_{split}	130fF
MSB	C4	1/8
MSB	C3	1/4
MSB	C2	1/2
MSB	C1	1
	Ctotal	2

Table 1. Capacitor array sizes.

$$C_{split} = \frac{C_{LSB} \cdot C_4}{C_4 + C_4} = 130\, fF$$

B. DAC Switches Logic

Transmission gates are used to reduce charge injection effect, because NMOS and PMOS have inverted polarity. The positive charge in NMOS can cancel out negative charge in PMOS. Below is our strategies to size the switches.

In our book, the strategy is to have all switches to have the same R_{on}. However, this causes possible transient when switches are toggled. In our case, we will size the switches proportionally to the capacitor sizes.

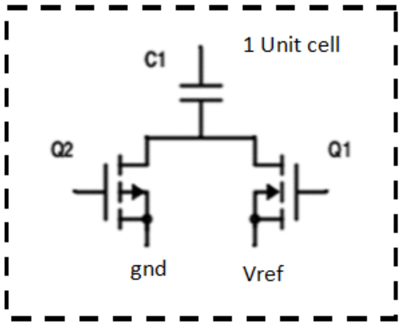


Figure 3 Unit cell.

The figure above represent one unit cell. General steps are:

1. Size Q2 & Q1 such that R_{on} (when gate is VDD)= R_{op} (When gate is gnd)

2. Use multiple unit cells for each bit. For example, our LSB is 8 bits and MSB is 4 bits, so the distribution will be (Change unit cell is same as changing multiplier):

	Capacitor	Size
	Bit Dummy	1 unit
LSB	Bit 12	1 unit
LSB	Bit 11	2 units
LSB	Bit 10	4 units
LSB	Bit 9	8 units
LSB	Bit 8	16 units
LSB	Bit 7	32 units
LSB	Bit 6	64 units
LSB	Bit 5	128 units
MSB	Bit 4	16 units
MSB	Bit 3	32 units
MSB	Bit 2	64 units
MSB	Bit 1	128 units

Table 2 Unit cell sizes.

Since resistance value is proportional to 1/size. By sizes above we could make sure $R_1 \cdot C_1 = R_{2N}C_2 = R_{2P}C_2$, then when switch changes there is no RC settling transient at Vx.

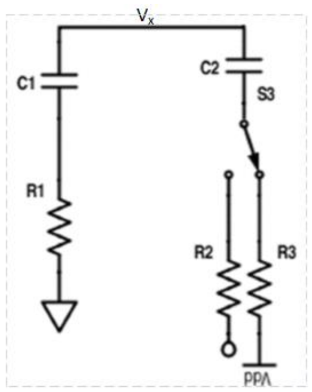


Figure 3 Size switches.

For 12 bit ADC, for each bit, we need one T (clock period), so total we have 12 bit = 12T. This is the time required (Sampling time) to convert V_{in} to all capacitance for tracking.

For sampling rate requirement ≥100 KSPS, $T_s = 15T = \frac{1}{100k} = 10\mu s$.

For each bit max settling time = T, $\log_{10}(2^{12}) = 3.6$,

$T \geq 3.6RC \Rightarrow R \leq \frac{T}{3.6C}$,

for example, if $C_{tot} = 2pF$ divided into 64 units, $C_i = \frac{2pF}{64}$,

$$R_{onN} = R_{onP} \leq \frac{{}^{10}\!/\!_{15}\,\mu s}{8.3 \cdot \frac{2pF}{64}},$$

$$R_{onN} = \frac{1}{\frac{\mu_n C_{ox} W}{L_{\min}}(V_{DD} - V_{TN})},\quad R_{onP} = \frac{1}{\frac{\mu_p C_{ox} W}{L_{\min}}(V_{DD} - |V_{Tp}|)}$$

C. Sampling and Hold

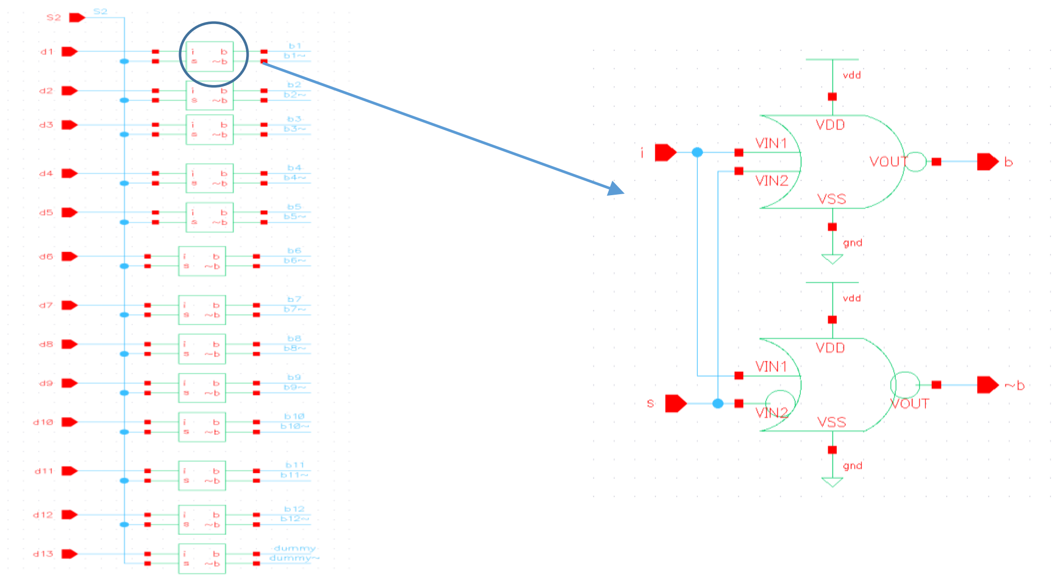


Figure 4 Logical gates used for control sampling and holding.

2.2 Test Capacitor Array and Switches

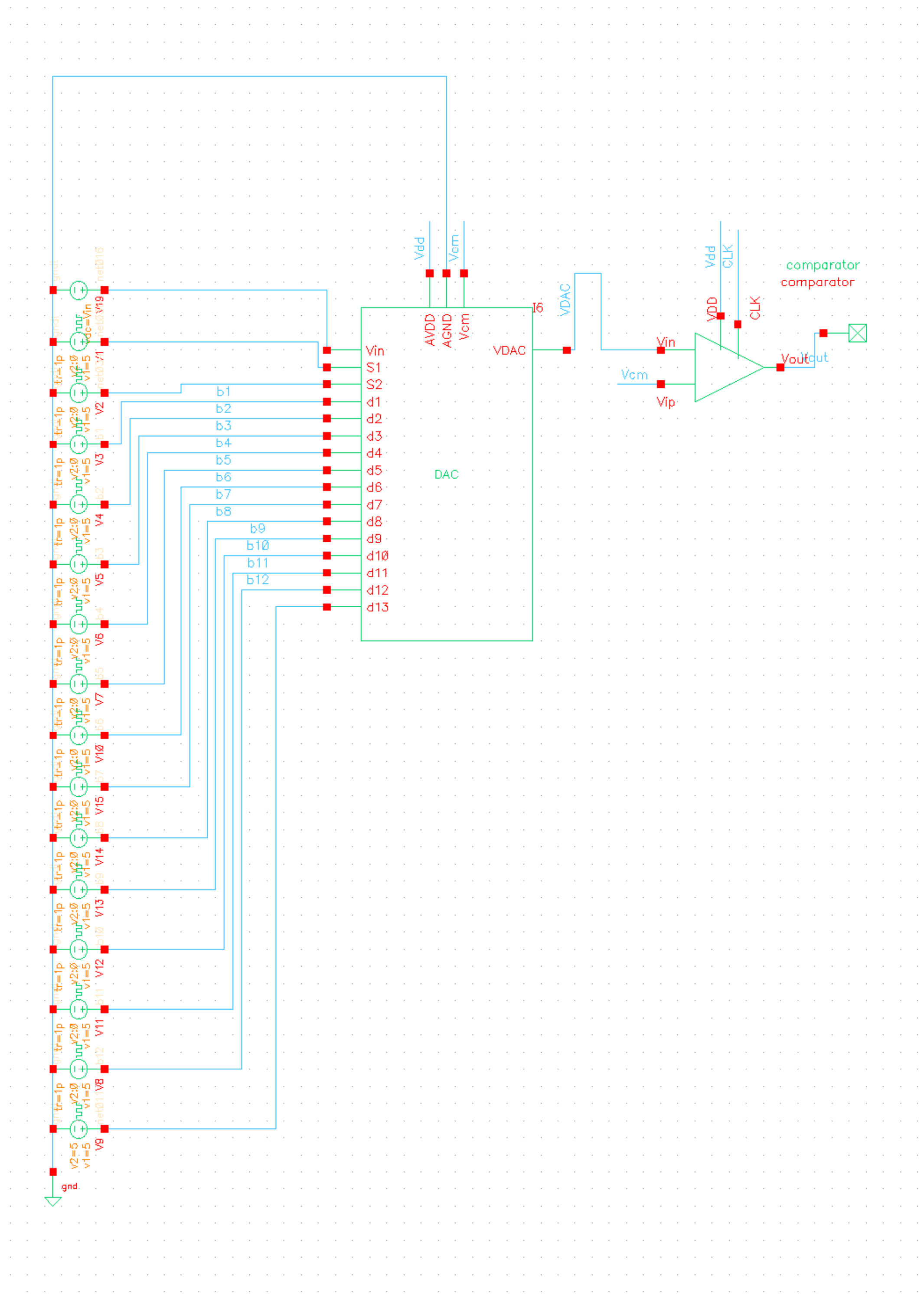


Figure 5 DAC testing bench.

A. DAC Testing

After constructing DAC, we set up a test bench like above. We need to make sure our DAC is working properly before proceed to SAR logic and comparison stage. The strategy here is to give each pin a impulse signal with different delay so that all the input signal can can cover to the range from 000000000000 to 111111111111.

To test the performance of the DAC, we use the “vpluse” as input for each bit, and certain period for each “vplus”. From bit 1 to bit 12, which is from top to bottom (In referring Figure [5]), the periods are 4096*tCLK, 2048*tCLK, 1024*tCLK, ⋯⋯, 2*tCLK. tCLK is clock period which is $0.6\mu s$.

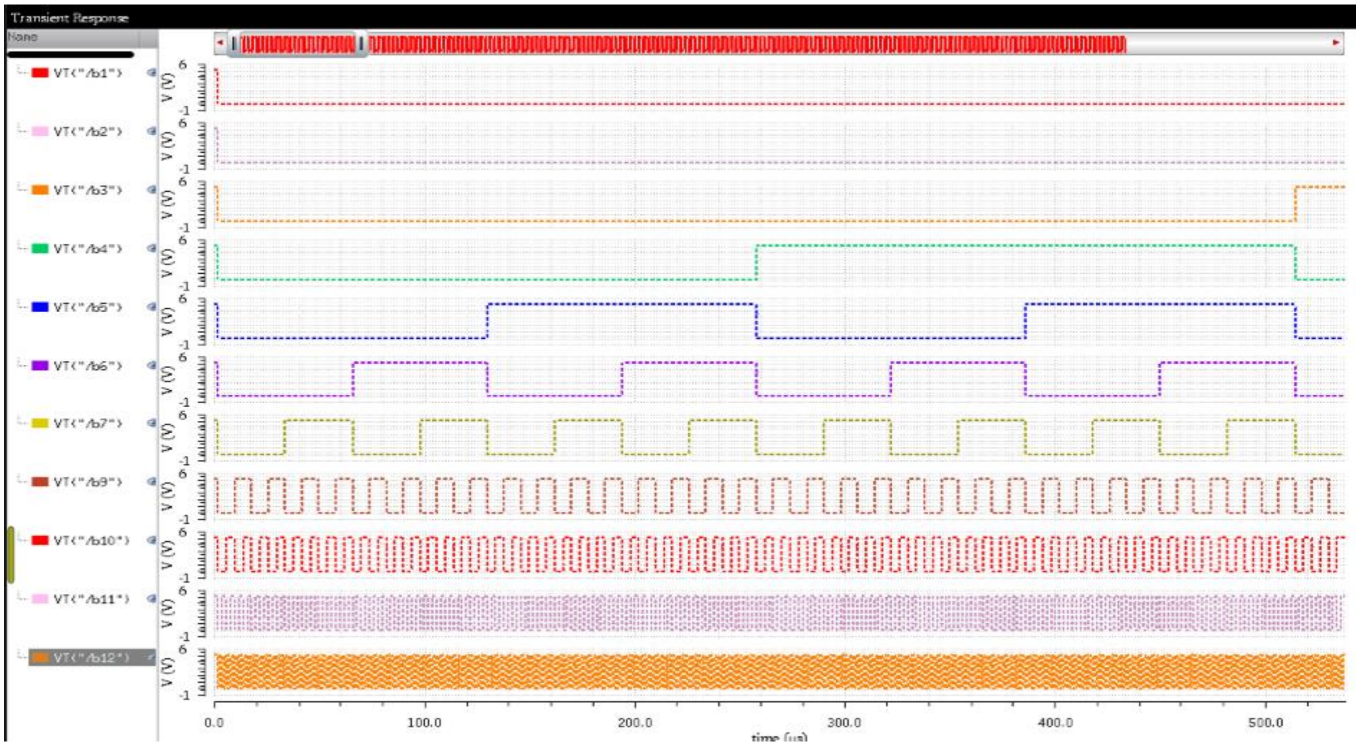


Figure 6 Testing signals.

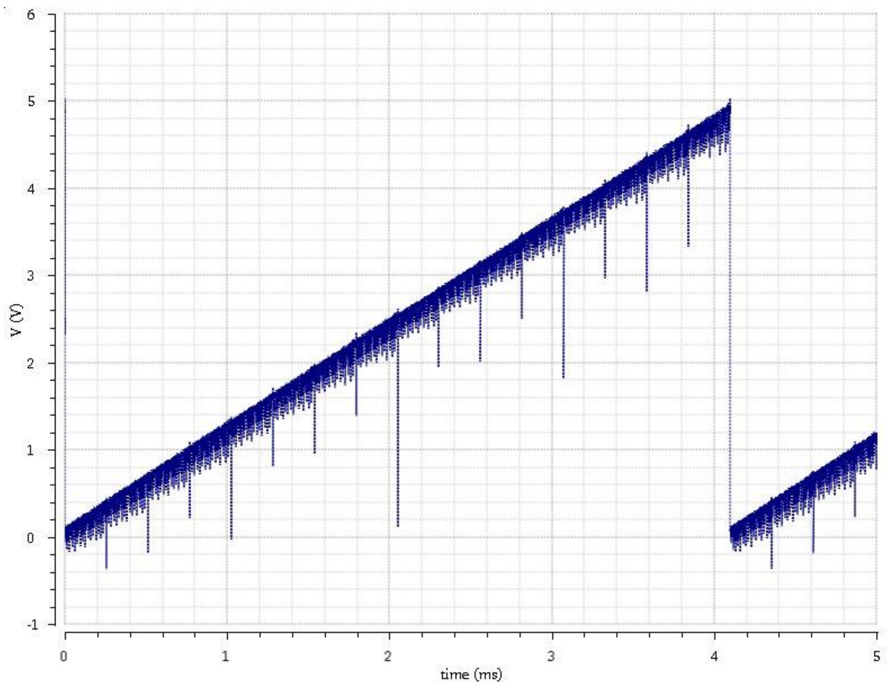


Figure 7 DAC output before sizing the switches.

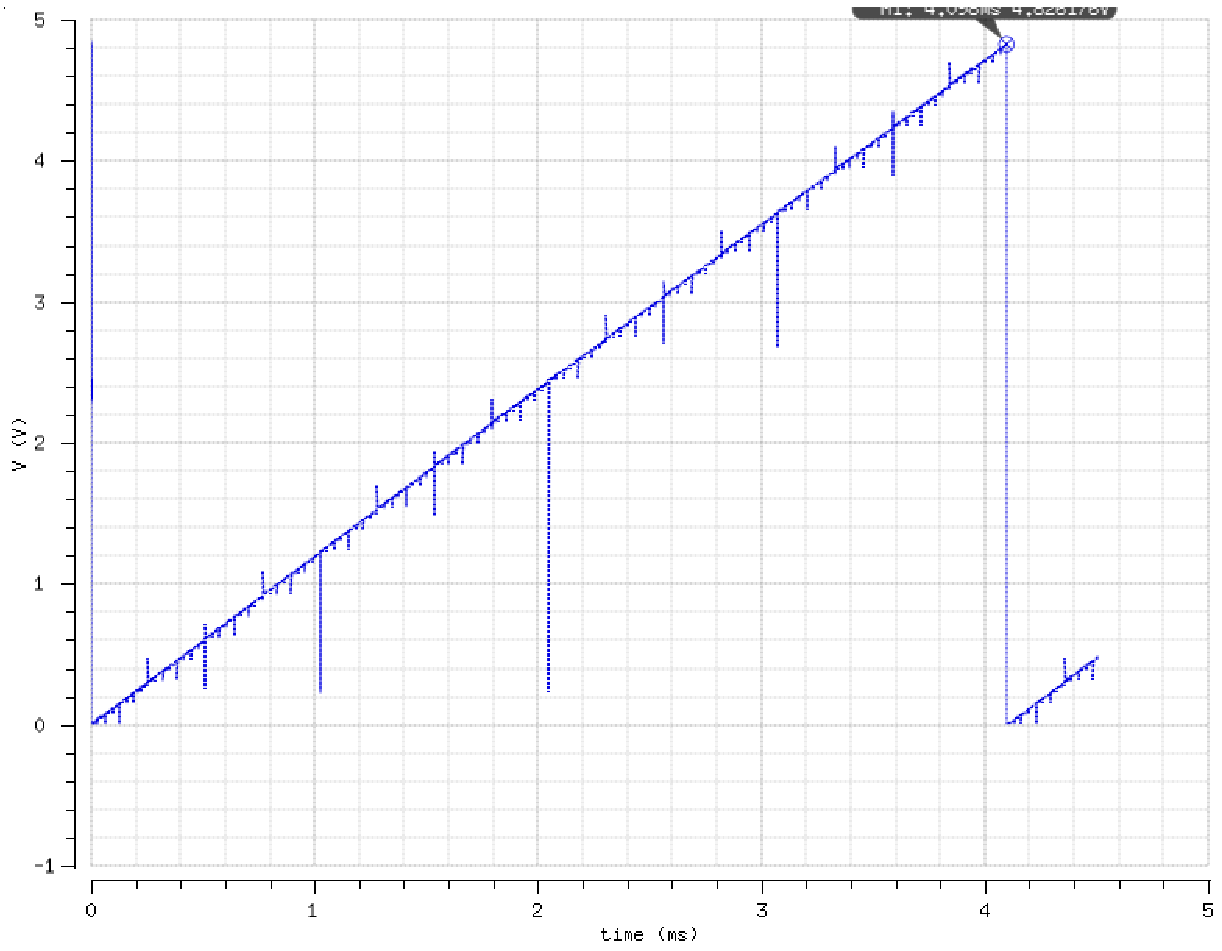


Figure 8 DAC output after sizing the switches.

As we can see, before we size the switches, in the Figure [9], there were lots of glitches occurred on the line. But after we sized our switches, the glitches were reduced effectively. One confusion is the DAC didn't go to 5V at the end, it only went to 4.8V as highest. Another problem need to be mentioned here is that there is an offset on the DAC, and the way we used to reduce the offset temporarily was find a good common mode voltage V_{cm} for DAC. We set V_{cm} as variable and swept it, in the meantime, we plot the DAC output.

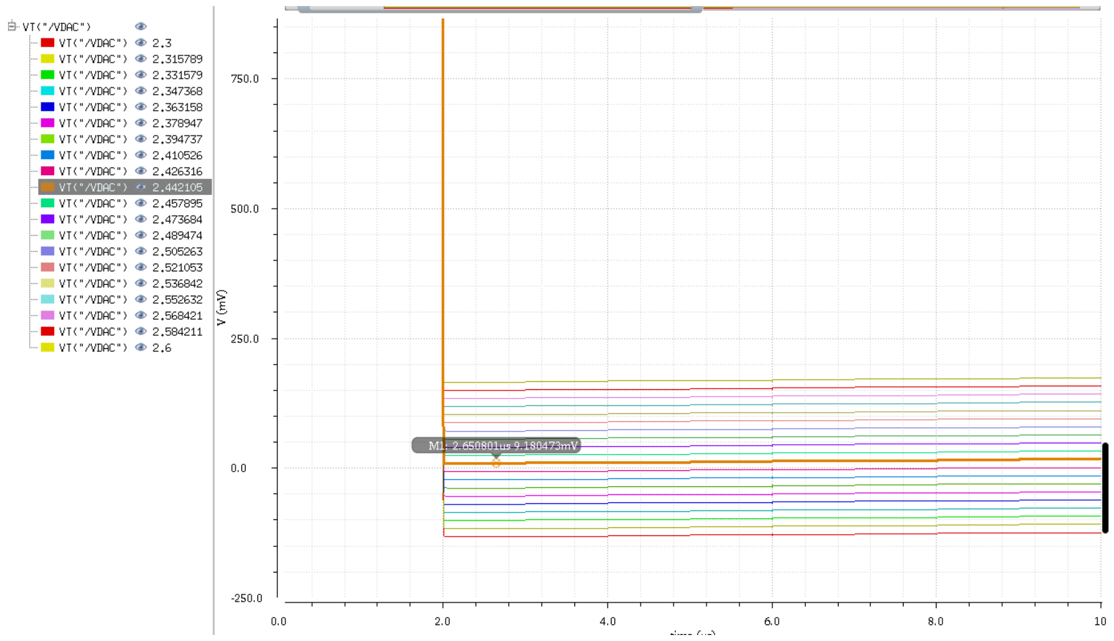


Figure 9 Find best Vcm to reduce offset voltage.

As you can see, the best V_{cm} occurs when V_{cm} is 2.44V, at which we have the smallest offset 9.18mV. But changing V_{cm} to adjust the offest is not the best way to elimate offest, instead, we should use offset cancellation to cancel the offset. It will be discussed in the compatator part later.

B. D-Flip Flop (DFF) Timing

To make our switching time more accurate, we proposed to use DFF to replace the NOT gate in the DAC, see Figure [12,13]. The reason we do that is because the NOT gate cause the time delay on the signal, but in our circuit, we want two signals S_1 and $\bar{S}_1$ feed into the switches at the same time, and DFF can make that happen.

The difference in clock edges that an inverter causes may cause the PMOS transistor of the transmission gate to be switched at a later time than the NMOS. To prevent this from happening, we decided to try and use a synchronous D flip-flop. The DFF in the libraries don’t provide a DFF with a Q and Q~ that we need. So we had to create a DFF from the gate level.

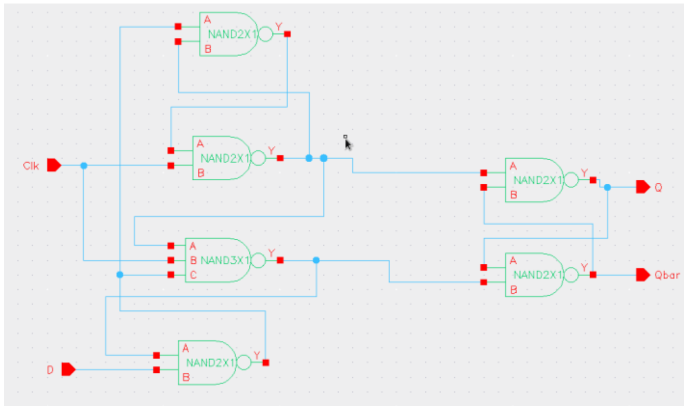


Figure 10 DFF configuration.

Connected as a clock divider, we can use this to generate our inverted signals for the switches.

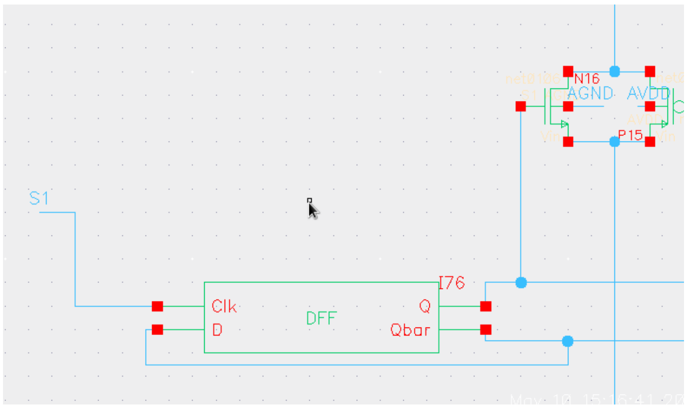


Figure 11 Improved switch connect with DFF.

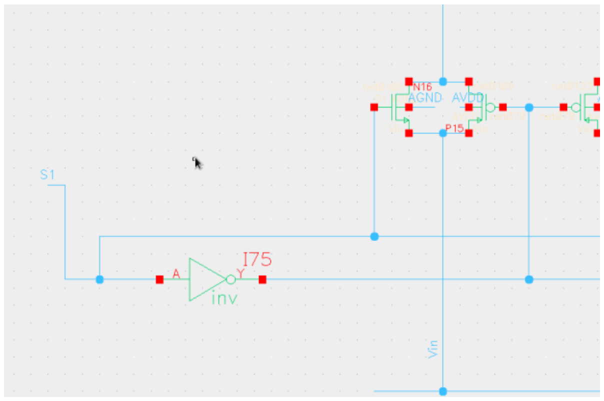


Figure 12 Original switch connection with NOT gate.

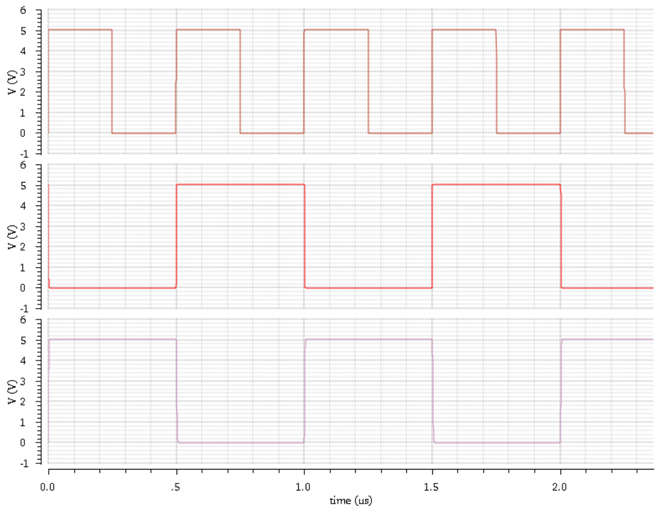


Figure 13 Transient output of DFF.

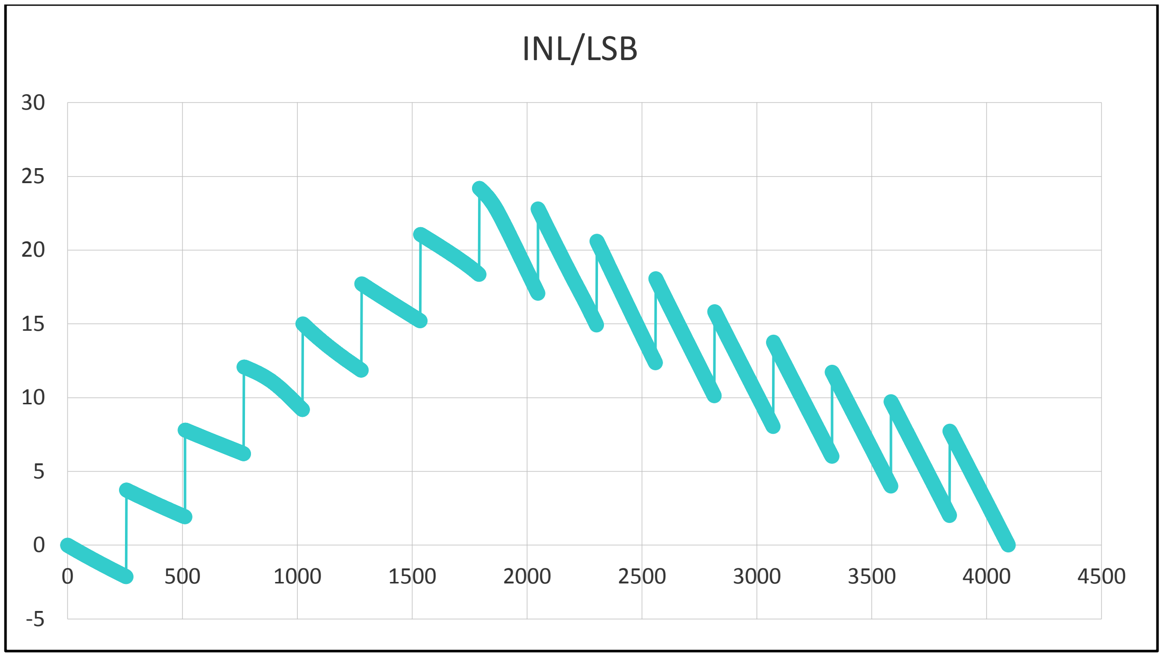


Figure 14 DAC INL.

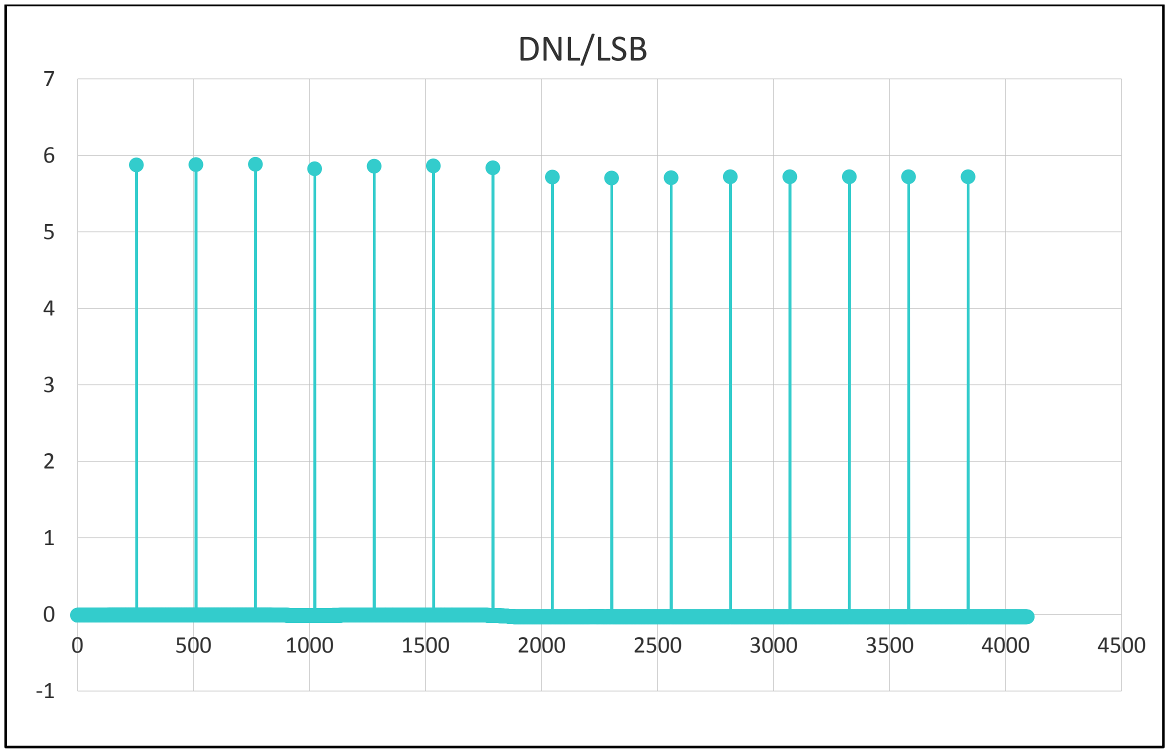
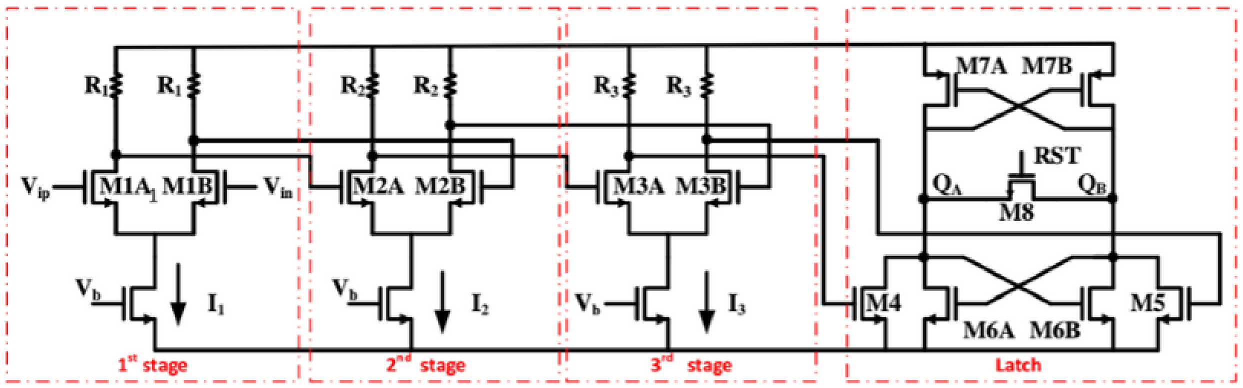


Figure 15 DAC DNL.

3. Comparator

3.1 Design Comparator

A comparator consists 3 stages of pre-amplifier, and a latch. For the pre-amplifier stages, we will scale down the power by factor of 3 between each stage while main the gain to be the same.



- 3-stages pre-amps (Reduce kickback noise from QA).
- Size: $I_1 = 2I_2 = 9I_3$, $R_3 = 3R_2 = 9R_1$,
- Gain: $A_V = g_m R_1 = \frac{I_1}{V_{eff1}} \frac{V_{IR}}{I_1/2} = \frac{2V_{IR}}{V_{eff1}} = \frac{5}{V_{eff1}}$, 19dB / stage
- Latch, M8 is switch

There are trade-offs between pre-amp gain and comparator speed, if gain is large, the speed will slow down.

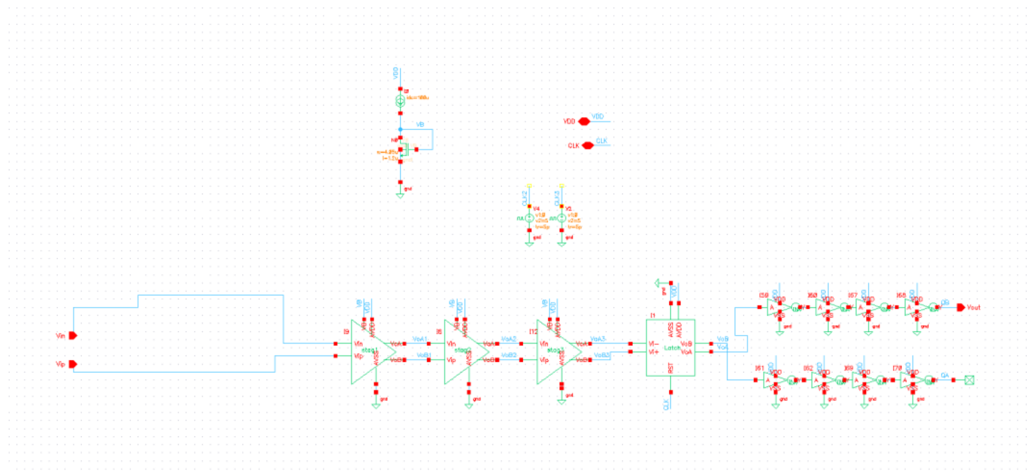


Figure 16 Comparator schematic.

3.2 Comparator Testing

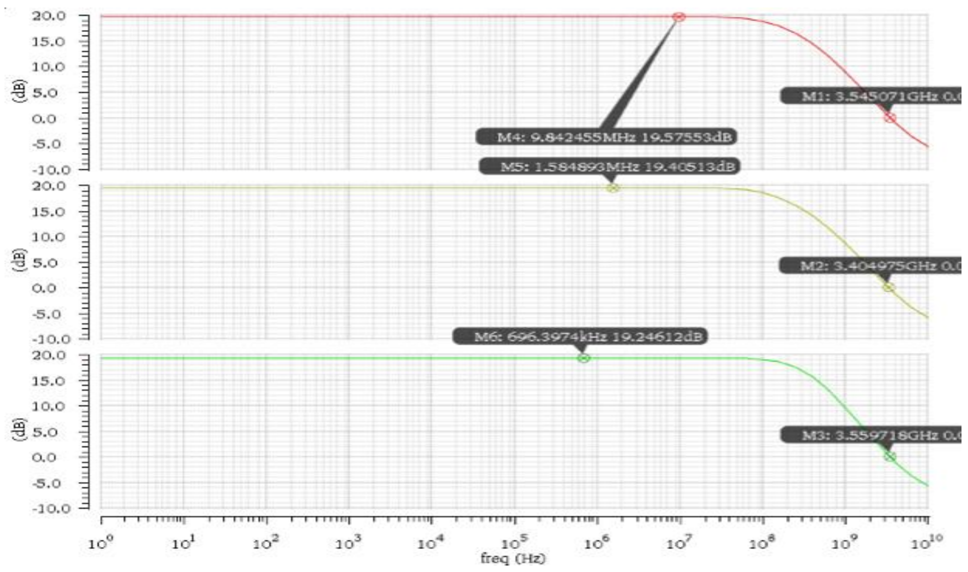


Figure 17 AC simulation.

From Top to bottom is stage 1, 2,3

Gain: 19.57dB, 19.4dB, 19.26dB

UGF: 3.5GHz, 3.4GHz, 3.6GHz

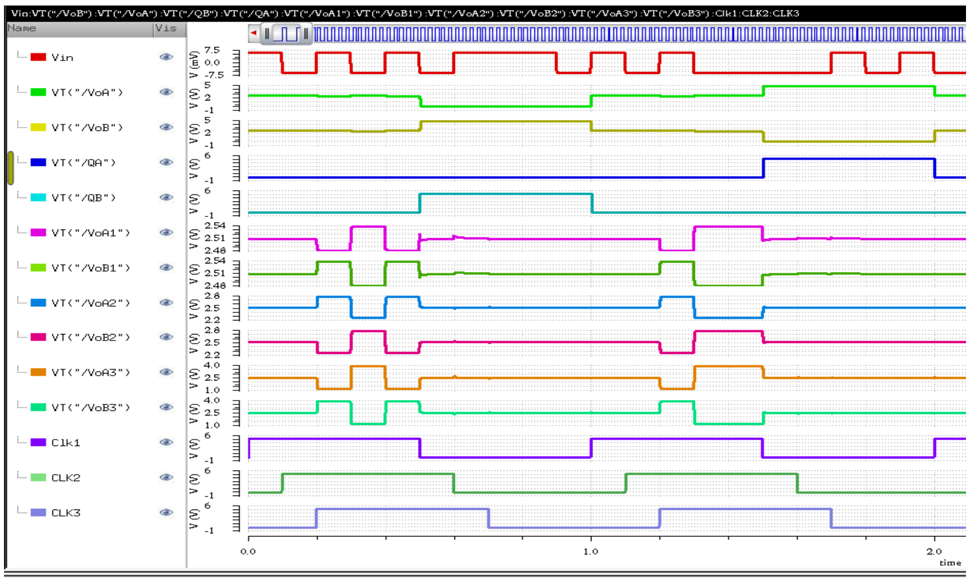


Figure 18 Transient simulation.

With 2.44V V_{cm}, the following testing is input VDAC, and measure comparator output.

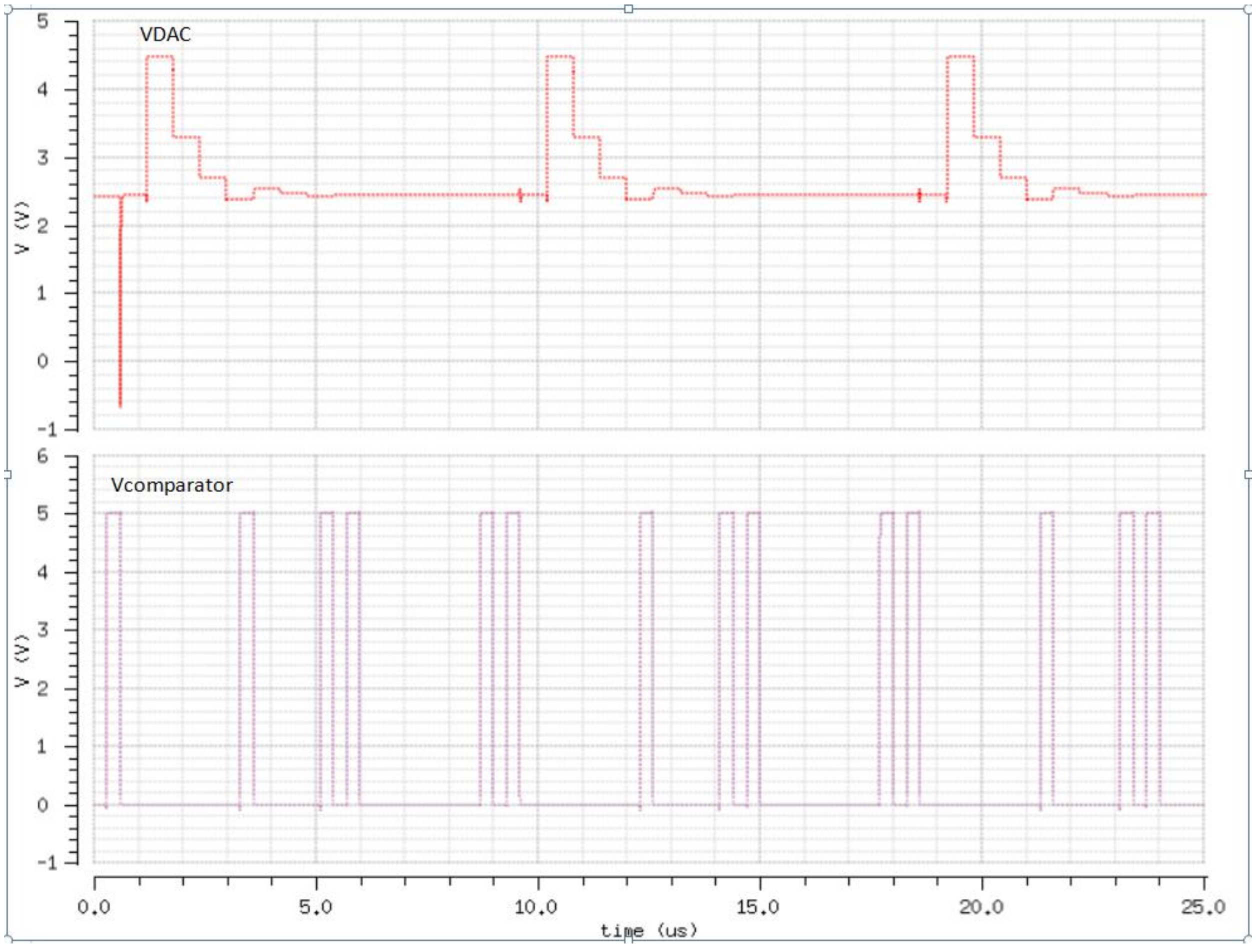
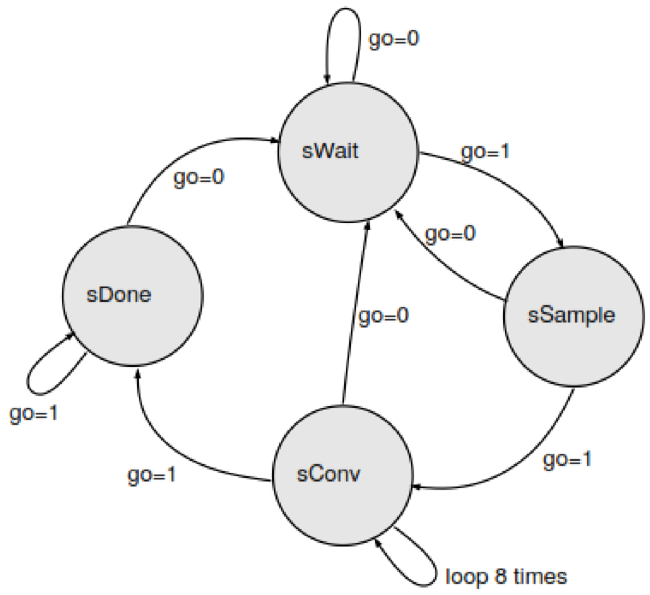


Figure 19 Comparator performance.

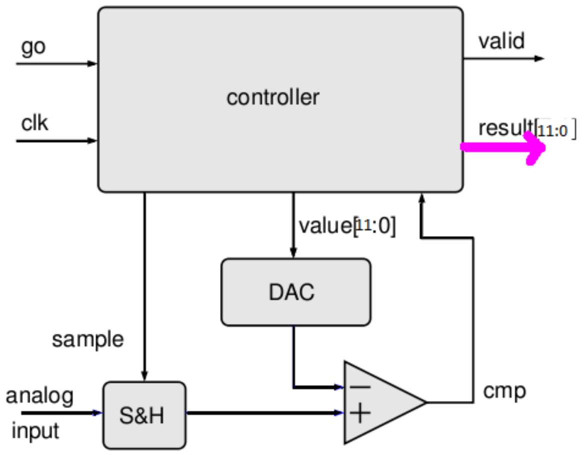
4. SAR Logic



Four States

1. wait [0,0]

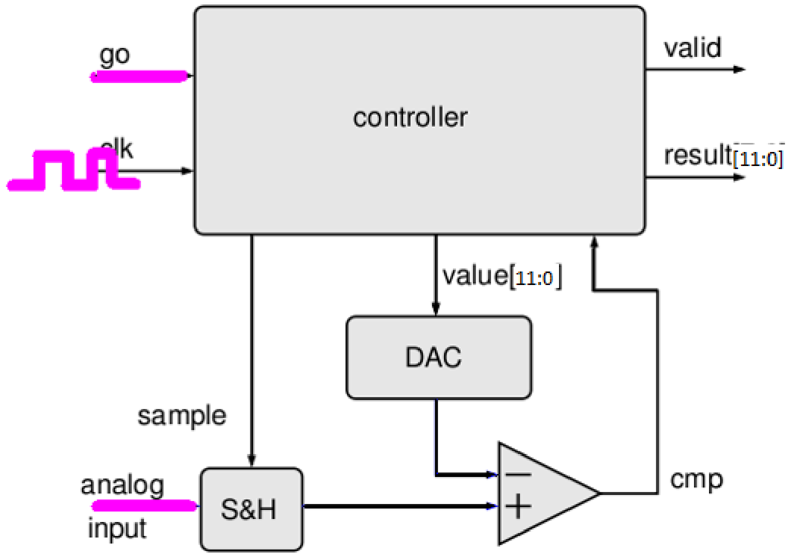
This is the stage in between outputting a result and waiting for go to be triggered.



2. sample [0,1]

Input Sampled

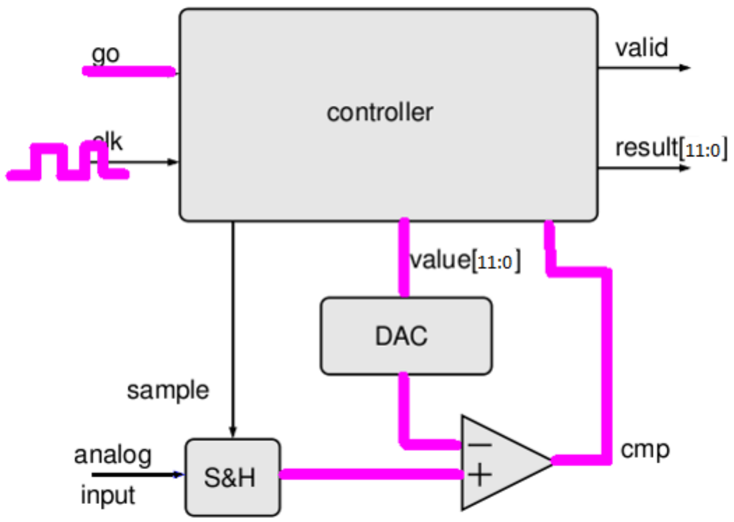
Result is held until process completes.



3.convert [1,0]

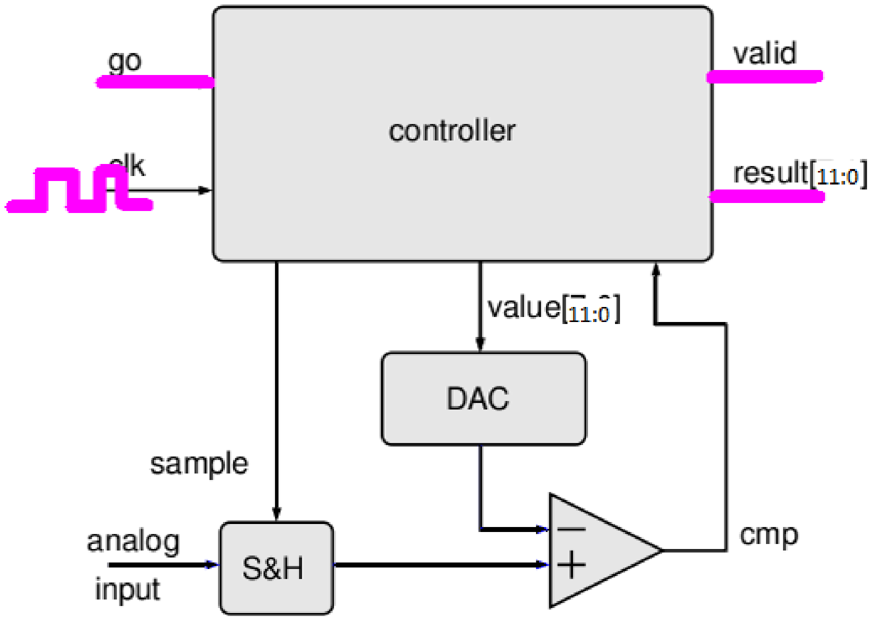
Internal Loop to find correct value relative to Vin.

Result from last conversion is still held until process completes.



4. done [1,1]

Output is correct and loop ends. Only stays in this state for a few clock cycles until the process repeats or until go is no longer active.



To implement this system, we decided to synthesize it from verilog code from an online source, the Computer Laboratory, University of Cambridge.
The synthesized code is provided below.

```
// Generated by Cadence Encounter(R) RTL Compiler v12.10-s012_1

// Verification Directory fv/controller

module controller(clk, go, valid, result, sample, value, cmp);
  input clk, go, cmp;
  output valid, sample;
  output [11:0] result, value;
  wire clk, go, cmp;
  wire valid, sample;
  wire [11:0] result, value;
  wire [11:0] mask;
  wire [1:0] state;
  wire n_0, n_1, n_14, n_16, n_17, n_18, n_20, n_21;
  wire n_22, n_23, n_24, n_25, n_26, n_27, n_28, n_29;
  wire n_30, n_31, n_32, n_33, n_34, n_35, n_36, n_37;
  wire n_38, n_39, n_40, n_41, n_42, n_43, n_44, n_45;
  wire n_46, n_47, n_48, n_49, n_50, n_51, n_52, n_53;
  wire n_54, n_55, n_56, n_57, n_58, n_59, n_60, n_61;
  wire n_62, n_63, n_64, n_65, n_66, n_67, n_68, n_69;
  wire n_70, n_71, n_72;
  DFFPOSX1 \result_reg[5] (.CLK (clk), .D (n_61), .Q (result[5]));
  DFFPOSX1 \result_reg[6] (.CLK (clk), .D (n_72), .Q (result[6]));
  DFFPOSX1 \result_reg[7] (.CLK (clk), .D (n_71), .Q (result[7]));
  DFFPOSX1 \result_reg[8] (.CLK (clk), .D (n_70), .Q (result[8]));
  DFFPOSX1 \result_reg[9] (.CLK (clk), .D (n_69), .Q (result[9]));
  DFFPOSX1 \result_reg[0] (.CLK (clk), .D (n_68), .Q (result[0]));
  DFFPOSX1 \result_reg[10] (.CLK (clk), .D (n_67), .Q (result[10]));
  DFFPOSX1 \result_reg[11] (.CLK (clk), .D (n_66), .Q (result[11]));
  DFFPOSX1 \result_reg[1] (.CLK (clk), .D (n_65), .Q (result[1]));
  DFFPOSX1 \result_reg[2] (.CLK (clk), .D (n_64), .Q (result[2]));
  DFFPOSX1 \result_reg[3] (.CLK (clk), .D (n_63), .Q (result[3]));
  DFFPOSX1 \result_reg[4] (.CLK (clk), .D (n_62), .Q (result[4]));
  DFFPOSX1 \mask_reg[2] (.CLK (clk), .D (n_58), .Q (mask[2]));
  DFFPOSX1 \mask_reg[3] (.CLK (clk), .D (n_46), .Q (mask[3]));
  DFFPOSX1 \mask_reg[4] (.CLK (clk), .D (n_45), .Q (mask[4]));
  DFFPOSX1 \mask_reg[5] (.CLK (clk), .D (n_42), .Q (mask[5]));
  DFFPOSX1 \mask_reg[6] (.CLK (clk), .D (n_39), .Q (mask[6]));
  DFFPOSX1 \mask_reg[7] (.CLK (clk), .D (n_37), .Q (mask[7]));
  DFFPOSX1 \mask_reg[10] (.CLK (clk), .D (n_56), .Q (mask[10]));
  DFFPOSX1 \mask_reg[9] (.CLK (clk), .D (n_57), .Q (mask[9]));
  DFFPOSX1 \mask_reg[8] (.CLK (clk), .D (n_60), .Q (mask[8]));
  DFFPOSX1 \mask_reg[0] (.CLK (clk), .D (n_55), .Q (mask[0]));
  DFFPOSX1 \mask_reg[1] (.CLK (clk), .D (n_54), .Q (mask[1]));
  INVX1 g1212(.A (n_53), .Y (n_72));
```

```
INVX1 g1214(.A (n_50), .Y (n_71));
INVX1 g1216(.A (n_49), .Y (n_70));
INVX1 g1218(.A (n_48), .Y (n_69));
INVX1 g1220(.A (n_47), .Y (n_68));
INVX1 g1222(.A (n_44), .Y (n_67));
INVX1 g1224(.A (n_43), .Y (n_66));
INVX1 g1226(.A (n_41), .Y (n_65));
INVX1 g1228(.A (n_40), .Y (n_64));
INVX1 g1230(.A (n_38), .Y (n_63));
INVX1 g1232(.A (n_36), .Y (n_62));
INVX1 g1234(.A (n_59), .Y (n_61));
INVX1 g1200(.A (n_34), .Y (n_60));
AOI22X1 g1235(.A (mask[5]), .B (n_52), .C (result[5]), .D (n_51), .Y
        (n_59));
INVX1 g1188(.A (n_27), .Y (n_58));
INVX1 g1202(.A (n_31), .Y (n_57));
INVX1 g1204(.A (n_30), .Y (n_56));
INVX1 g1206(.A (n_29), .Y (n_55));
INVX1 g1208(.A (n_28), .Y (n_54));
DFFPOSX1 \mask_reg[11] (.CLK (clk), .D (n_22), .Q (mask[11]));
AOI22X1 g1213(.A (mask[6]), .B (n_52), .C (result[6]), .D (n_51), .Y
        (n_53));
AOI22X1 g1215(.A (mask[7]), .B (n_52), .C (result[7]), .D (n_51), .Y
        (n_50));
AOI22X1 g1217(.A (mask[8]), .B (n_52), .C (result[8]), .D (n_51), .Y
        (n_49));
AOI22X1 g1219(.A (mask[9]), .B (n_52), .C (result[9]), .D (n_51), .Y
        (n_48));
AOI22X1 g1221(.A (mask[0]), .B (n_52), .C (result[0]), .D (n_51), .Y
        (n_47));
INVX1 g1190(.A (n_26), .Y (n_46));
INVX1 g1192(.A (n_25), .Y (n_45));
AOI22X1 g1223(.A (mask[10]), .B (n_52), .C (result[10]), .D (n_51),
        .Y (n_44));
AOI22X1 g1225(.A (mask[11]), .B (n_52), .C (result[11]), .D (n_51),
        .Y (n_43));
INVX1 g1194(.A (n_24), .Y (n_42));
AOI22X1 g1227(.A (mask[1]), .B (n_52), .C (result[1]), .D (n_51), .Y
        (n_41));
AOI22X1 g1229(.A (mask[2]), .B (n_52), .C (result[2]), .D (n_51), .Y
        (n_40));
INVX1 g1196(.A (n_23), .Y (n_39));
AOI22X1 g1231(.A (mask[3]), .B (n_52), .C (result[3]), .D (n_51), .Y
        (n_38));
INVX1 g1198(.A (n_35), .Y (n_37));
AOI22X1 g1233(.A (mask[4]), .B (n_52), .C (result[4]), .D (n_51), .Y
        (n_36));
AOI22X1 g1199(.A (mask[7]), .B (n_33), .C (mask[8]), .D (n_32), .Y
        (n_35));
AOI22X1 g1201(.A (mask[8]), .B (n_33), .C (mask[9]), .D (n_32), .Y
        (n_34));
AOI22X1 g1203(.A (mask[9]), .B (n_33), .C (mask[10]), .D (n_32), .Y
        (n_31));
AOI22X1 g1205(.A (mask[10]), .B (n_33), .C (mask[11]), .D (n_32), .Y
            (n_30));
  AOI22X1 g1207(.A (mask[0]), .B (n_33), .C (mask[1]), .D (n_32), .Y
            (n_29));
  AOI22X1 g1209(.A (mask[1]), .B (n_33), .C (mask[2]), .D (n_32), .Y
            (n_28));
  AOI22X1 g1189(.A (mask[2]), .B (n_33), .C (mask[3]), .D (n_32), .Y
            (n_27));
  DFFPOSX1 \state_reg[1] (.CLK (clk), .D (n_21), .Q (state[1]));
  AOI22X1 g1191(.A (mask[3]), .B (n_33), .C (mask[4]), .D (n_32), .Y
            (n_26));
  AOI22X1 g1193(.A (mask[4]), .B (n_33), .C (mask[5]), .D (n_32), .Y
            (n_25));
  AOI22X1 g1195(.A (mask[5]), .B (n_33), .C (mask[6]), .D (n_32), .Y
            (n_24));
  AOI22X1 g1197(.A (mask[6]), .B (n_33), .C (mask[7]), .D (n_32), .Y
            (n_23));
  OAI21X1 g1238(.A (n_1), .B (n_32), .C (n_51), .Y (n_22));
  DFFPOSX1 \state_reg[0] (.CLK (clk), .D (n_20), .Q (state[0]));
  INVX1 g1236(.A (n_33), .Y (n_21));
  AND2X2 g1240(.A (n_32), .B (cmp), .Y (n_52));
  NAND2X1 g1237(.A (go), .B (n_16), .Y (n_33));
  INVX1 g1244(.A (n_18), .Y (n_20));
  NAND2X1 g1242(.A (go), .B (sample), .Y (n_51));
  AND2X2 g1243(.A (n_17), .B (state[1]), .Y (n_32));
  OAI21X1 g1245(.A (mask[0]), .B (n_14), .C (n_17), .Y (n_18));
  HAX1 g1241(.A (state[0]), .B (state[1]), .YC (valid), .YS (n_16));
  AND2X2 g1246(.A (n_14), .B (state[0]), .Y (sample));
  NOR2X1 g1247(.A (n_0), .B (state[0]), .Y (n_17));
  OR2X1 g1256(.A (result[7]), .B (mask[7]), .Y (value[7]));
  OR2X1 g1259(.A (result[11]), .B (mask[11]), .Y (value[11]));
  OR2X1 g1250(.A (result[1]), .B (mask[1]), .Y (value[1]));
  OR2X1 g1253(.A (result[0]), .B (mask[0]), .Y (value[0]));
  OR2X1 g1248(.A (result[5]), .B (mask[5]), .Y (value[5]));
  OR2X1 g1255(.A (result[3]), .B (mask[3]), .Y (value[3]));
  OR2X1 g1257(.A (result[10]), .B (mask[10]), .Y (value[10]));
  OR2X1 g1251(.A (result[9]), .B (mask[9]), .Y (value[9]));
  OR2X1 g1254(.A (result[8]), .B (mask[8]), .Y (value[8]));
  OR2X1 g1258(.A (result[6]), .B (mask[6]), .Y (value[6]));
  OR2X1 g1249(.A (result[4]), .B (mask[4]), .Y (value[4]));
  OR2X1 g1252(.A (result[2]), .B (mask[2]), .Y (value[2]));
  INVX1 g1261(.A (state[1]), .Y (n_14));
  INVX1 g1260(.A (mask[11]), .Y (n_1));
  INVX1 g1262(.A (go), .Y (n_0));
endmodule
```

Figure 20 Synthesized SAR logic controller.

5. Conclusion

To wrap up for some of the design requirements and final ADC performance. We meet the requirement for sampling speed, which is 100 KSPS, because our clock time is $0.6\mu s$. Our power consumption is 76mW, which was not meet the requirement. And we also didn’t have time to do the simulations on ADC’s static and dynamic performance. But, we can still explain some of the performance of our ADC.

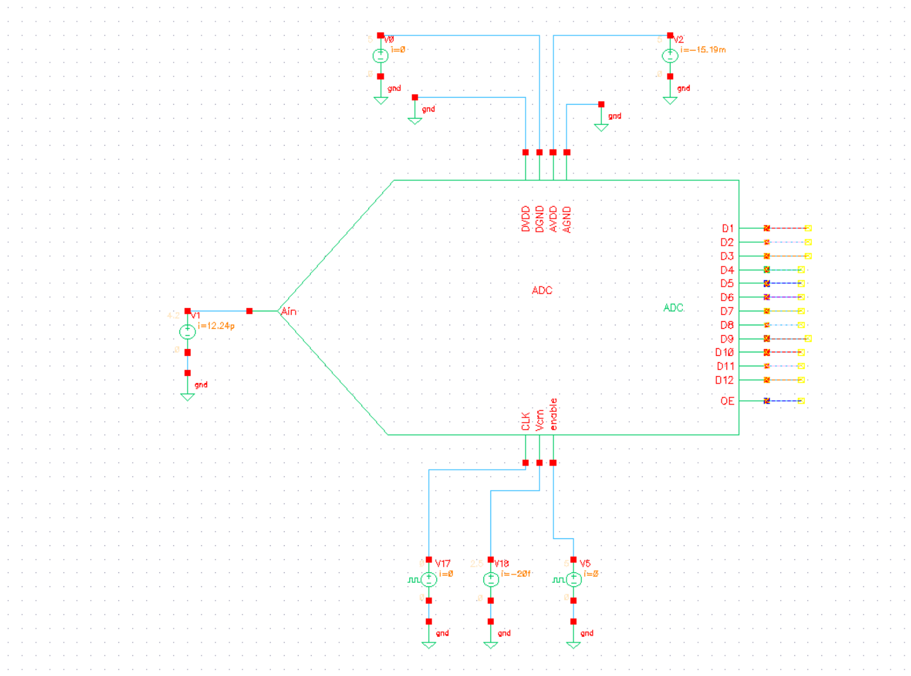


Figure 21 ADC.

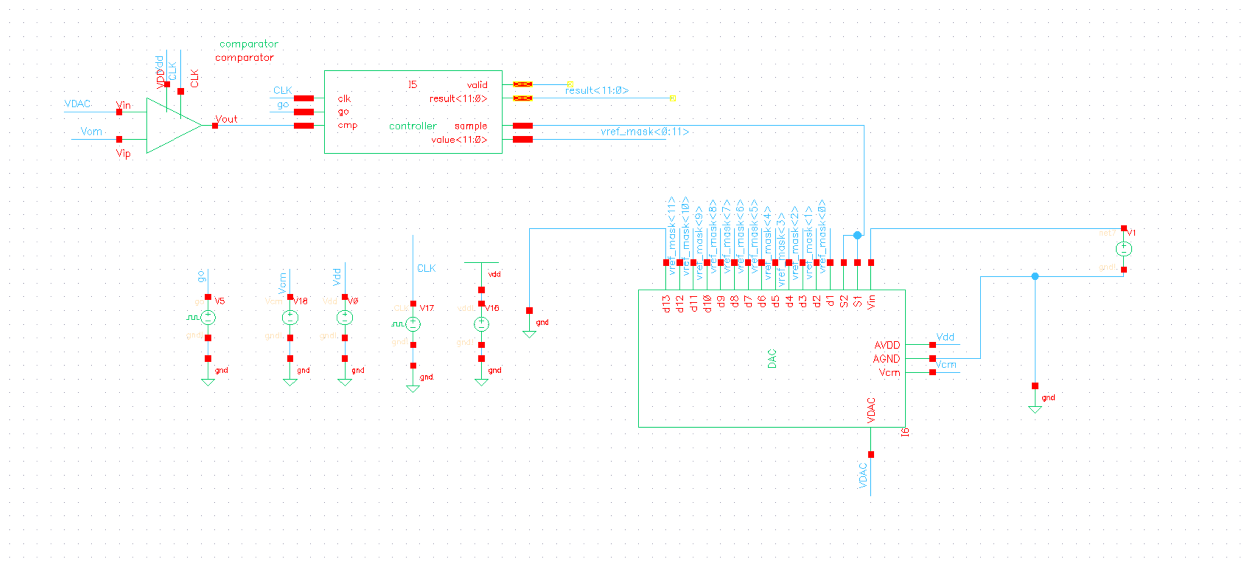


Figure 22 ADC testbench prototype.

Vin	Code	ideal	real
1.0	001100110100	819.2	820
1.5	010011010000	1228.8	1232
2.0	011001100111	1638.4	1639
2.5	011111111111	2048	2047
3.0	100110011010	2457.6	2458
3.5	101100110001	2867.2	2865
4.0	110011001101	3276.8	3277
4.5	111001100100	3686.4	3684
5.0	111111111111	4096	4095

Table 3 Ideal and real ADC output.

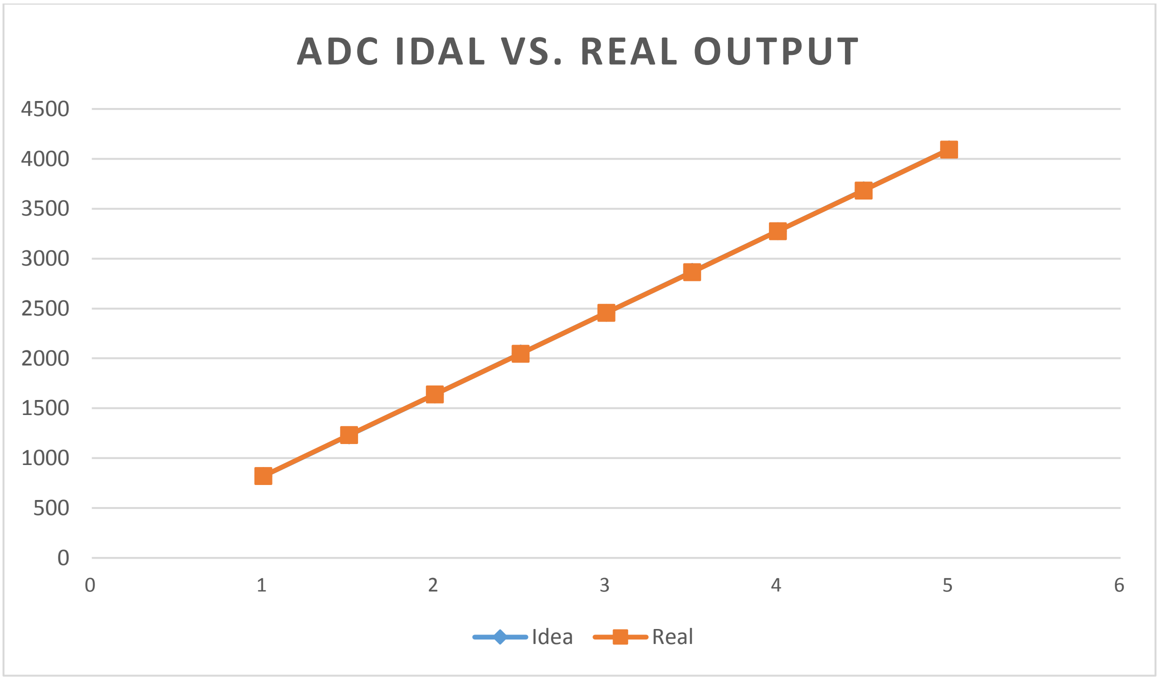


Figure 23 ADC ideal output vs. real output.

On this raw graph, the ADC is pretty accurate, but we need to thorough simulations with more data in the future.

Works Cited:

"Example: Successive Approximation Analog to Digital Converter (ADC)." Example: Successive Approximation Analog to Digital Converter (ADC) Description of Operation (n.d.): 2-www.cl.cam.ac.uk. Computer Laboratory, University of Cambridge. Web. 10 May 2015. <https://www.cl.cam.ac.uk/teaching/2002/ECAD/ExtraExamples/ADCcontroller.pdf>.

“Understanding SAR ADCs: Their Architecture and Comparison with Other ADCs”
http://www.maximintegrated.com/en/app-notes/index.mvp/id/1080

Appendices

INL DNL Test Code used for ADC:

```
%Code density/histogram test to calculate INL and DNL require a large number of samples.
%Step 1: Apply a close to full-scale sine wave (but not clipping) and find the mid-code
%for the applied signal.
%Step 2: Apply the same sine wave input, but slightly larger amplitude to clip the ADC
%slightly.
%Run the following program, enter the number of samples, resolution and mid-code from
%Step 1and continue.
%Copyright Au/Hofner, Maxim Integrated, 160 Rio Robles, Sunnyvale,
%CA94086
%This program is believed to be accurate and reliable. This program may get altered
%without prior notification.

filename=input('File Name or press ENTER for Listing Transfer HP16500C: ');
if isempty(filename)
   filename = 'listing';
end
fid=fopen(filename,'r');
numpt=input('Number of Data Points?  ');
numbit=input ('ADC Resolution?  ');
mid_code=input(Enter Mid-Code (Mean):  ');

for i=1:13,            %Discard 13 lines of redundant or header-related HP16500C data
   fgetl(fid);
end
[v1,count]=fscanf(fid,'%f',[2,numpt]);
fclose(fid);

v1=v1';
code=v1(:,2);
code_count=zeros(1,2^numbit);  %Code count

for i=1:size(code),
   code_count(code(i)+1)=code_count(code(i)+1) + 1;
end

%Routine to detect whether the ADC's input is clipping or not
```

```
if code_count(1) == 0 | code_count(2^numbit) == 0 | ...
  (code_count(1) < code_count(2)) | (code_count(2^numbit-1) > code_count(2^numbit))
   disp('Increase Sine-Wave Amplitude to Slightly Clip the ADC!!!');
   break;
end

A=max(mid_code,2^numbit-1-mid_code)+0.1; %Initial estimate of actual sine wave amplitude

vin=(0:2^numbit-1)-mid_code;        %distance of codes to mid code
sin2ramp=1./(pi* sqrt(A^2*ones(size(vin))-vin.*vin));              %sin2ramp*numpt is the expected
%Count each code; keep increasing estimate of A until the actual total number of counts from
%code 1 to 2^numbit-2 matches with that predicted by sin2ramp*numpt
while sum(code_count(2:2^numbit-1)) < numpt*sum(sin2ramp(2:2^numbit-1))
  A=A+0.1;
  sin2ramp=1./(pi* sqrt(A^2*ones(size(vin))-vin.*vin));
end

disp('You Have Applied a Sine Wave of (dBFS): ');
Amplitude=A/(2^numbit/2)
figure;
plot([0:2^numbit-1],code_count,[0:2^numbit-1],sin2ramp*numpt);
title('CODE HISTOGRAM - SINE WAVE');
xlabel('DIGITAL OUTPUT CODE');
ylabel('COUNTS');
axis([0 2^numbit-1 0 max(code_count(2),code_count(2^numbit-1))]);
code_countn=code_count(2:2^numbit-1)./(numpt*sin2ramp(2:2^numbit-1)); %End points discarded!
figure;
plot([1:2^numbit-2],code_countn);
title('CODE HISTOGRAM - NORMALIZED')
xlabel('DIGITAL OUTPUT CODE');
ylabel('NORMALIZED COUNTS');

dnl=code_countn-1;           %DNL=Vj+1-Vj-1LSB where Vj represents a transition point
                             %Vj+1-Vj is proportional to normalized code count

inl=zeros(size(dnl));
for j=1:size(inl')
   inl(j)=sum(dnl(1:j));       %INL,j=DNL,0+DNL,1+...+DNL,j
end
```

```
%INL still contains the offset and gain error!;

%INL with end-points fit, i.e. INL=0 at end-points the straight line joining the 2 end points
%[p,S]=polyfit([1,2^numbit-2],[inl(1),inl(2^numbit-2)],1);
%the best-fit straight line
[p,S]=polyfit([1:2^numbit-2],inl,1);
inl=inl-p(1)*[1:2^numbit-2]-p(2);

disp('End Points Eliminated for DNL and INL Calculations');
figure;
plot([1:2^numbit-2],dnl);
grid on;
title('DNL');
xlabel('DIGITAL OUTPUT CODE');
ylabel('DNL (LSB)');
figure;
plot([1:2^numbit-2],inl);
grid on;
title('INL (BEST END-POINT FIT)');
xlabel('DIGITAL OUTPUT CODE');
ylabel('INL(LSB)');
```

SAR Logic Verilog codes:

```
// ADC controller
module controller(clk,go,valid,result, sample,value,cmp);
input clk; // clock input
input go; // go=1 to perform conversion
output valid; // valid=1 when conversion finished
output [7:0] result; // 8 bit result output
output sample; // to S&H circuit
output [7:0] value; // to DAC
input cmp; // from comparitor
reg [1:0] state; // current state in state machine
reg [7:0] mask; // bit to test in binary search
reg [7:0] result; // hold partially converted result
// state assignment
parameter sWait=0, sSample=1, sConv=2, sDone=3;
// synchronous design
always @(posedge clk) begin
if (!go) state <= sWait; // stop and reset if go=0
else case (state) // choose next state in state machine
sWait : state <= sSample;
sSample :
begin // start new conversion so
state <= sConv; // enter convert state next
mask <= 8’b10000000; // reset mask to MSB only
result <= 8’b0; // clear result
end
sConv :
begin
// set bit if comparitor indicates input larger than
// value currently under consideration, else leave bit clear
if (cmp) result <= result | mask;
// shift mask to try next bit next time
mask <= mask>>1;
// finished once LSB has been done
if (mask[0]) state <= sDone;
end
sDone :;
endcase
end
assign sample = state==sSample; // drive sample and hold
assign value = result | mask; // (result so far) OR (bit to try)
assign valid = state==sDone; // indicate when finished
endmodule
```

SAR Logic Verilog testbench:

```
module testbench();
// registers to hold inputs to circuit under test, wires for outputs
reg clk,go;
wire valid,sample,cmp;
wire [7:0] result;
wire [7:0] value;
// instance controller circuit
controller c(clk,go,valid,result, sample,value,cmp);
// generate a clock with period of 20 time units
always begin
#10;
clk=˜clk;
end
initial clk=0;
// simulate analogue circuit with a digital model
reg [7:0] hold;
always @(posedge sample) hold = 8’b01000110;
assign cmp = ( hold >= value);
// monitor some signals and provide input stimuli
initial begin
$monitor($time, " go=%b valid=%b result=%b sample=%b value=%b cmp=%b
state=%b mask=%b",
go,valid,result,sample,value,cmp,c.state,c.mask);
#100; go=0;
#100; go=1;
#5000; go=0;
#5000; go=1;
#40; go=0;
#5000;
$stop;
end
endmodule
```

The End.