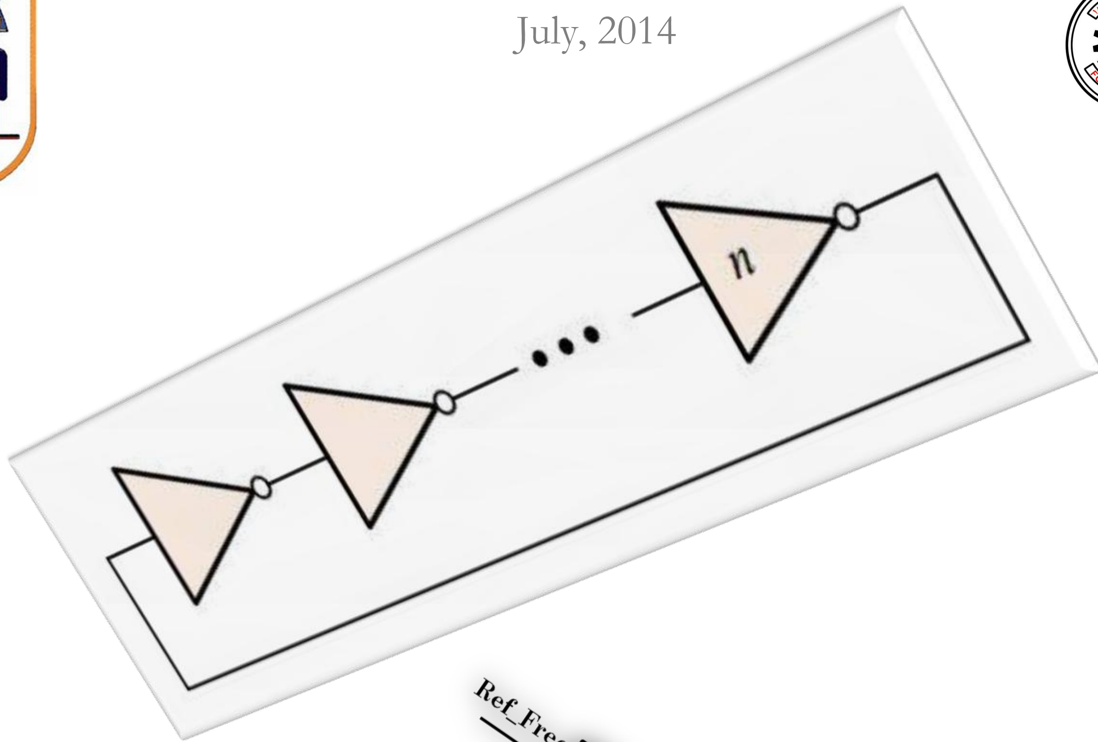
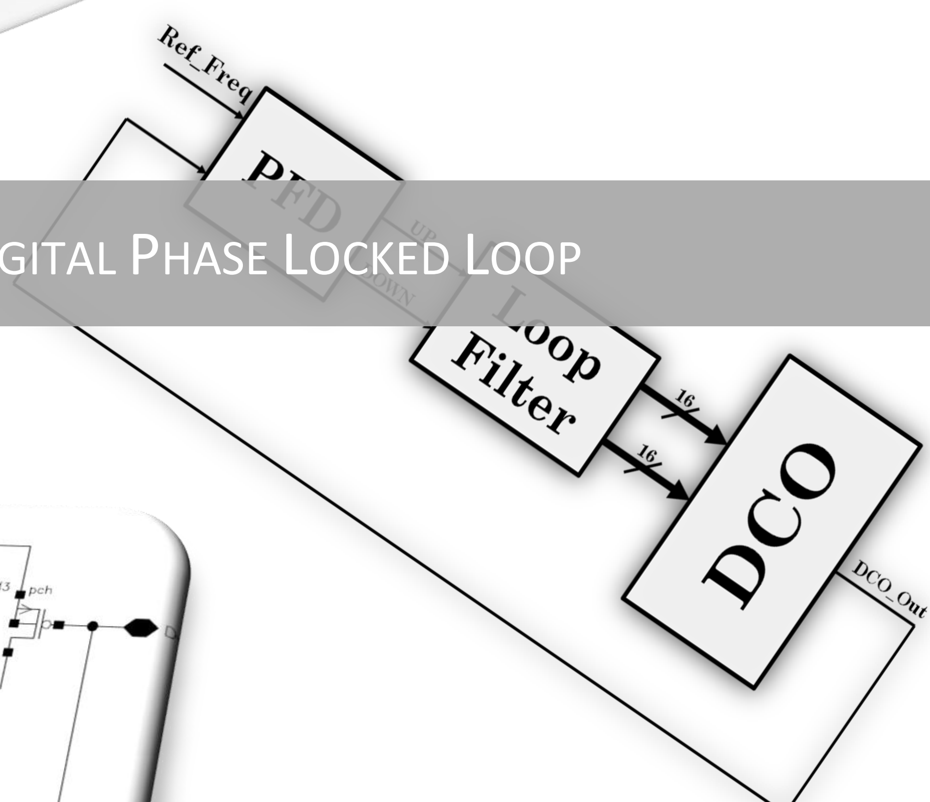
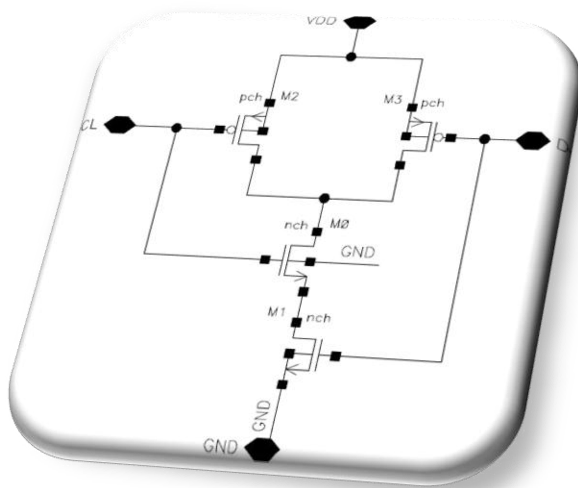
July, 2014



[ADPLL]

All Digital Phase Locked Loop





Faculty Of Engineering ,Cairo University

LOW POWER, SMALL AREA ALL DIGITAL PHASE LOCKED LOOP (ADPLL)

By

Ammar Mohammad Ibrahim

Ammar Mohammad Hussein

Mohammad Abdel-Lateef Abdel-Tawab

A thesis submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science in Electronics and Electrical Communications Department, Faculty of Engineering

Cairo University

Supervised by
Dr. Hassan Mostafa

July, 2014

ABSTRACT

ADPLL

The objective of the thesis is to design an All Digital Phase Locked Loop (ADPLL) with low power. The design consists of three main blocks: Digitally Controlled Oscillator (DCO), Phase Detector (PD) and Loop Filter (LF). The DCO is considered as the heart of the ADPLL as it consumes the most power for the whole system. The design went through two different approaches, standard cells and custom cells .This design can be used in Clock and Data Recovery (CDR) system as an application.

This thesis presents a low power all digital phase locked loop (ADPLL) in 65 nm CMOS process with 1.2 V power supply. It operates in the frequency range of 100 – 300 MHz. The ADPLL uses a digitally controlled oscillator with two stages, fine tuning stage and coarse tuning stage. The source of oscillation for this DCO is the ring oscillator.

The proposed ADPLL uses also a phase-frequency detector (PFD) and shift registers for the loop filter. It achieved power consumption at 200 MHz of 0.6 mW and a lock time of 1 uS.

The last design step of the ADPLL is the layout, some modification applied to the layout to satisfy the required specifications, at the end of this thesis a comparison between the required and the achieved specifications in schematic and layout level.

Design considerations of the ADPLL circuit components and implementation using Cadence, Synopsys and Mentor tools are presented; the AMS tool is used frequently in the standard cells flow.

TABLE OF CONTENTS

List of Figures v
Acknowledgments vii
Acronyms viii
Chapter 1: Introduction 1
Chapter 2: Custom Cells Approach 5
2.1 DCO 5
2.1.1 Ring Oscillator 6
2.1.2 Fine stage 11
2.1.3 Coarse stage 16
2.1.4 Conclusion and final results 20
2.2 PFD 24
2.3 Loop Filter 26
2.4 Overall Design 28
2.4.1 Extreme Reference Frequency 28
2.4.2 Intermediate Reference Frequency 29
2.4.3 Frequency step response 31
2.4.4 Jitter calculation 33
Chapter 3: Standard Cells Approach 34
3.1 PFD 34
3.2 Loop Filter 36
3.3 PFD and Loop Filter 37
3.3 Overall Design 38
Chapter 4: Layout 43
4.1 DCO 43
4.1.1 DCV 43
4.1.2 DCV2 44
4.1.3 Ring Oscillator 45
4.1.4 Complete DCO 46
4.2 PFD 47
4.3 Loop Filter 49
4.4 Overall Design 50
4.5 Conclusion 51
References 52
Appendix A: AMS Tutorial 53
Appendix B: Logic Synthesis 107
Appendix C: Importing Synthesized Design into Cadence Composer 124
Appendix D: Standard Cell Placement and Routing 138
Appendix E: Power Calculation 161

LIST OF FIGURES

Figure 1.1: Block diagram of the PLL 1
Figure 1.2: The Overall block diagram of the ADPLL 2
Figure 1.3: Jitter Illustration 3
Figure 2.1: DCO symbol view 5
Figure 2.2: DCO internal structure 6
Figure 2.3: Ring oscillator block diagram 6
Figure 2.4: A Schematic view for the inverter 7
Figure 2.5: Simulation results of the inverter 8
Figure 2.6: Simulation results for the ring oscillator without delay cells 8
Figure 2.7: A schematic view of the HDC 9
Figure 2.8: The ring oscillator with HDC cells 10
Figure 2.9: Simulation results for the ring oscillator with HDC cells 10
Figure 2.10: A schematic view for the DCV cell 11
Figure 2.11: The gate capacitance of NAND gate 12
Figure 2.12: Simulation results for the NAND based DCV Cell 13
Figure 2.13: A block diagram for the DCV building block 13
Figure 2.14: DCV array internal structure 14
Figure 2.15: A symbol view for the DCV array 15
Figure 2.16: Period step Vs. frequency steps 16
Figure 2.17: A schematic view for the DCV2 cell 17
Figure 2.18: DCV2 block 18
Figure 2.19: DCV2 array 19
Figure 2.20: A symbol view for the DCV2 array 20
Figure 2.21: Different output periods of the DCO 21
Figure 2.22: Different output waveforms of the DCO 22
Figure 2.23: Period steps versus code 22
Figure 2.24: Frequency steps versus code 23
Figure 2.25: PFD schematic 24
Figure 2.26: State diagram of the PFD 25
Figure 2.27: PFD simulation results 25
Figure 2.28: Digital control signals used to switch a set of varactors 26
Figure 2.29: Schematic view of the shift register 27
Figure 2.30: Reference = 100MHz 28
Figure 2.31: DCO delay 29
Figure 2.32: Reference= 250MHz 29
Figure 2.33: Effect of using counter 30
Figure 2.34: Frequency counter 31
Figure 2.35: frequency step response 32
Figure 2.36: Eliminating the oscillations 32
Figure 2.37: Eye-diagram of the ADPLL 33
Figure 3.1: PFD schematic 35
Figure 3.2: Loop Filter schematic 36
Figure 3.3: PFD and Loop Filter 37
Figure 3.4: Overall ADPLL (standard cells) 38

Figure 3.5: AMS of the overall ADPLL without the counter....38
Figure 3.6: AMS simulation of the overall ADPLL with counter....39
Figure 3.7: AMS of the ADPLL with counter....40
Figure 3.8: The standard cells of both PFD and Loop Filter....41
Figure 3.9: The transistor level of the standard D flip flop in cadence. 42
Figure 4.1: DCV layout....43
Figure 4. 2: DCV block layout....44
Figure 4. 3: DCV array layout....44
Figure 4. 4: DCV2 layout....44
Figure 4. 5: DCV2 block layout....45
Figure 4. 6: DCV2 array layout....45
Figure 4. 7: Ring Oscillator layout....45
Figure 4. 8: Final DCO layout....46
Figure 4. 9: DCO operating range in layout....47
Figure 4. 10: PFD layout....47
Figure 4.11: PFD pre-layout simulation....48
Figure 4.12: PFD post-layout simulation....48
Figure 3.13: Shift Register layout....49
Figure 4.14: Pre-layout simulation of the Shift Register....49
Figure 4.15: Post-layout simulation of the Shift Register....50
Figure 4.16: The overall layout....50
Figure 4.17: Overall post layout simulation....50

ACKNOWLEDGMENTS

First of all we would like to thank ALLAH for his mercy supporting us through this project. We wish to express sincere appreciation to **Dr. Hassan Mostafa** for his assistance in the preparation of this project. In addition, special thanks to our teaching assistance **Eng. Mahmoud Nagib Sawaby** for helping us with the needs and ideas of the standard cells approach and to **Eng. Mohammad Wagih Emam** for his time and effort for answering our questions and helping deal with Cadence toolkit. Thanks also to our friend **Eng. Islam Abdou** for installing the AMS tool in Cadence.

ACRONYMS

ADPLL	All Digital Phase Locked Loop
AMS	Analog Mixed Simulation
DCO	Digitally Controlled Oscillator
DCV	Digitally Controlled Varactor
DFF	D Flip Flop
DPLL	Digital Phase Locked Loop
HDC	Hysteresis Delay Cell
LF	Loop Filter
LPLL	Linear Phase Locked Loop
PD	Phase Detector
PFD	Phase Frequency Detector
PLL	Phase Locked Loop
SR	Shift Register
VCO	Voltage Controlled Oscillator

Chapter 1

INTRODUCTION

The PLL represents one of the most active topics in signal processing and communication theory. The initial ideas started as early as 1919 in the context of synchronization of oscillators. The theory of phase-locked loop was based on the theory of feedback amplifiers. The PLL contributed significantly to communications and motor servo systems. Due to the rapid development of integrated circuits (IC's) since the 1970's, PLLs are widely used in modern signal processing and communication systems, and it is expected that PLL will contribute to improvement in performance and reliability of future communication systems. The applications of PLLs include filtering, frequency synthesis, motor speed control, frequency modulation, demodulation, signal detection, frequency tracking and many other applications.

The PLL consists of three main blocks VCO, Loop Filter and Phase detector as shown in figure 1.1.

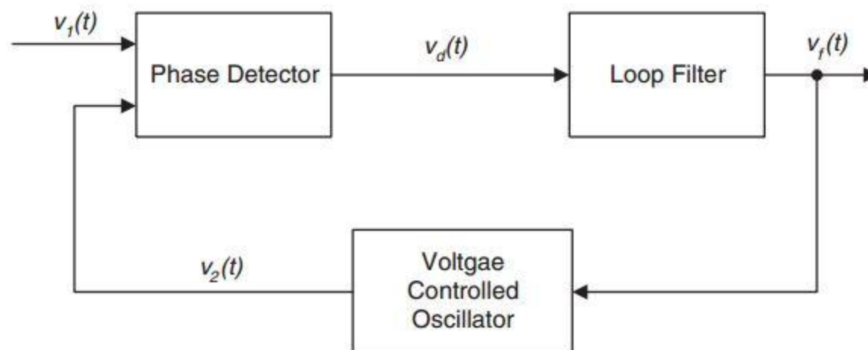


Figure 1.1: Block diagram of the PLL

There are many types of PLL according to the internal blocks and designing techniques as following:

1. LPLL: Linear Phase Locked Loop which contains a VCO and RC circuit for the Loop Filter block and uses a multiplier to detect the phase difference between the reference frequency and the VCO output frequency.
2. DPLL: Digital Phase Locked Loop was the very first digital PLL; it was in effect a hybrid device ONLY the phase detector was built as a digital block like EXOR.
3. ADPLL: All Digital Phase Locked Loop in which all the blocks are built as digital blocks.

ADPLL consists of the same three main blocks mentioned previously except for the VCO; it will be replaced by the DCO as shown in figure 1.2

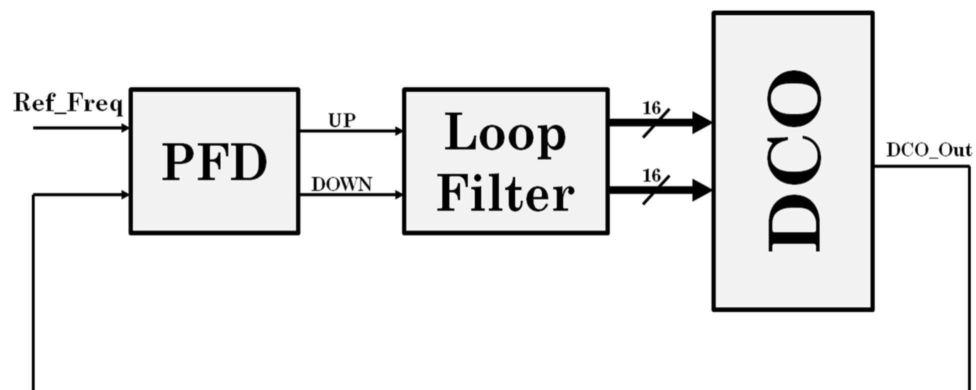


Figure 1.2: The Overall block diagram of the ADPLL

PLL in general has its own parameters such as:

- **The operating frequency range:** the range of frequencies that PLL can lock on them.
- **The lock time:** the time which PLL needs to lock on the reference frequency.
- **The Jitter:** undesired deviation from the true periodicity of an assumed periodic signal.

What is Jitter?

Jitter is the undesired deviation from true periodicity of an assumed periodic signal, Deviation (expressed in ± ps) can occur on either the leading edge or the trailing edge of a signal. Jitter may be induced and coupled onto a clock signal from several different sources and is not uniform over all frequencies.

Period of ring oscillator vibrates in a random manner T=T+T` where T` is a random value. In high-quality circuits range of T` is relatively small compared to T. This variation in oscillator period is called jitter. Local temperature effects cause the period of a ring oscillator to wander above and below the long-term average period when the local silicon is cold, the propagation delay is slightly shorter, causing the ring oscillator to run at a slightly higher frequency, which eventually raises the local temperature. When the local silicon is hot, the propagation delay is slightly longer, causing the ring oscillator to run at a slightly lower frequency, which eventually lowers the local temperature.

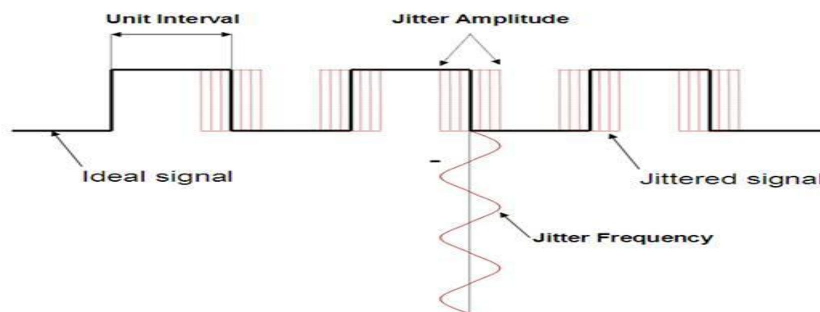


Figure 1.3: Jitter Illustration

We considered the DCO as the first design stage, because it consumes about 50% of the system power and covers the most area of the whole system area. The design went through two approaches the custom cells and the standard cells. The DCO is done in the custom approach however; the rest of the design went through both approaches.

The proposed ADPLL has the following specifications:

- Power < 1 mW.
- Area < 0.01 mm^2.
- Frequency Range from 100 MHz to 300 MHz.
- Lock time < 10 μs.
- Peak to Peak Jitter < 20 ps.
- R.M.S. Jitter < 5 ps.

In the following chapters we are going to discuss the design steps in details for each block of the ADPLL to satisfy these requirements.

Frequently Asked Question about ADPLL:

- Why digital? What is the problem of the analog (linear) one?
    - Basically, the ADPLL consumes less power than the linear PLL.
    - ADPLL can be easily scaled down to another technology.
    - Linear PLL needs an off chip components such as capacitors and resistors (for the loop filter) which do not have a fixed and stable value because they may suffer from aging.
    - ADPLL covers less area than linear PLL.

C h a p t e r   2

CUSTOM CELLS APPROACH

2.1 Digitally Controlled Oscillator

The digitally controlled oscillator (DCO) is considered the heart of the PLL as it controls the overall system performance and consumes the most power and area of the whole design. The proposed DCO follows a full custom design approach to make it easier to control its area and power. It consists of three main blocks:

1. A ring oscillator

2. A fine tuning stage (DCV Array)

3. A Coarse tuning stage (DCV2 Array)

Figure 2.1 is the symbol view of the DCO.

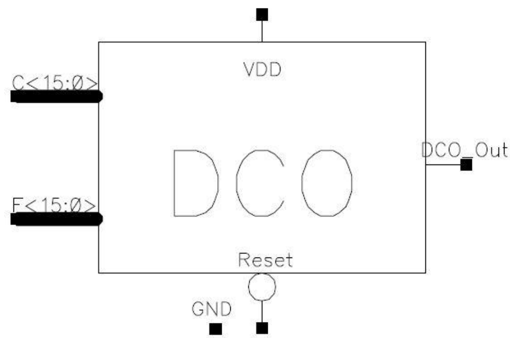


Figure 2.1: DCO symbol view

Another figure for the internal block diagram of the DCO is figure 2.2, figuring out its three main blocks.

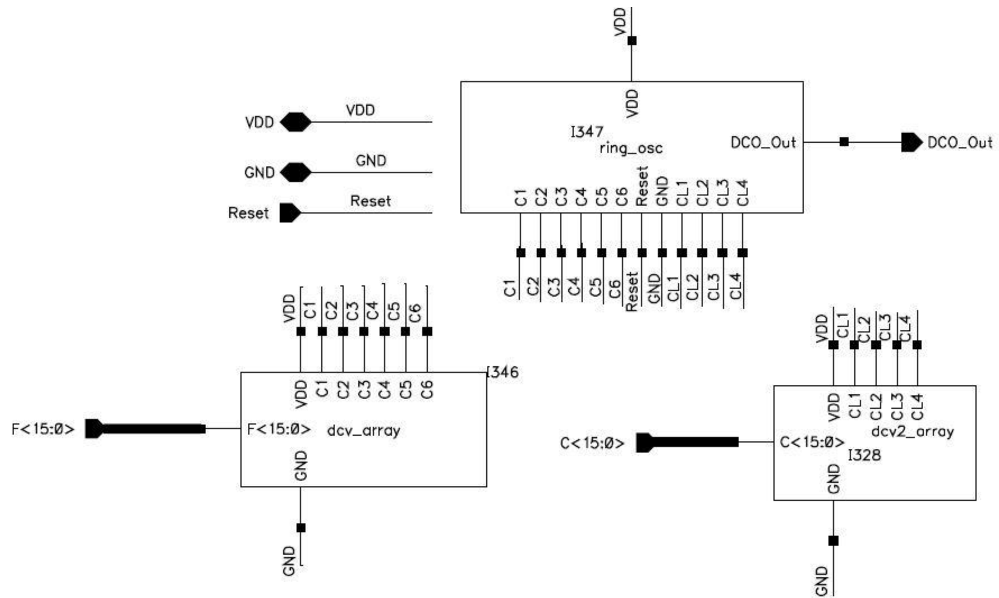


Figure 2.2: DCO internal structure

We will start our discussion by investigating the internal structure of the ring oscillator followed by the fine tuning stage (DCV Array) and finally the coarse tuning stage (DCV2 Array).

2.1.1 Ring Oscillator

The ring oscillator is the source of oscillation for the DCO. It consists of an odd number of inverters in a cascaded configuration with a feedback from the output to the input. Figure 2.3 is a block diagram for the proposed ring oscillator.

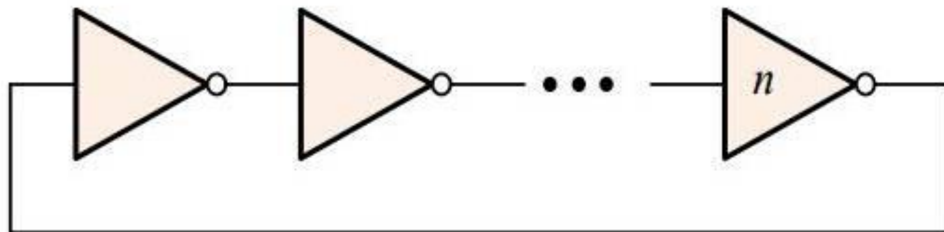


Figure 2.3: Ring oscillator block diagram

The design of the ring oscillator follows a full custom approach. The following is the schematic view (figure 2.4) of the basic cell for the ring oscillator, the inverter.

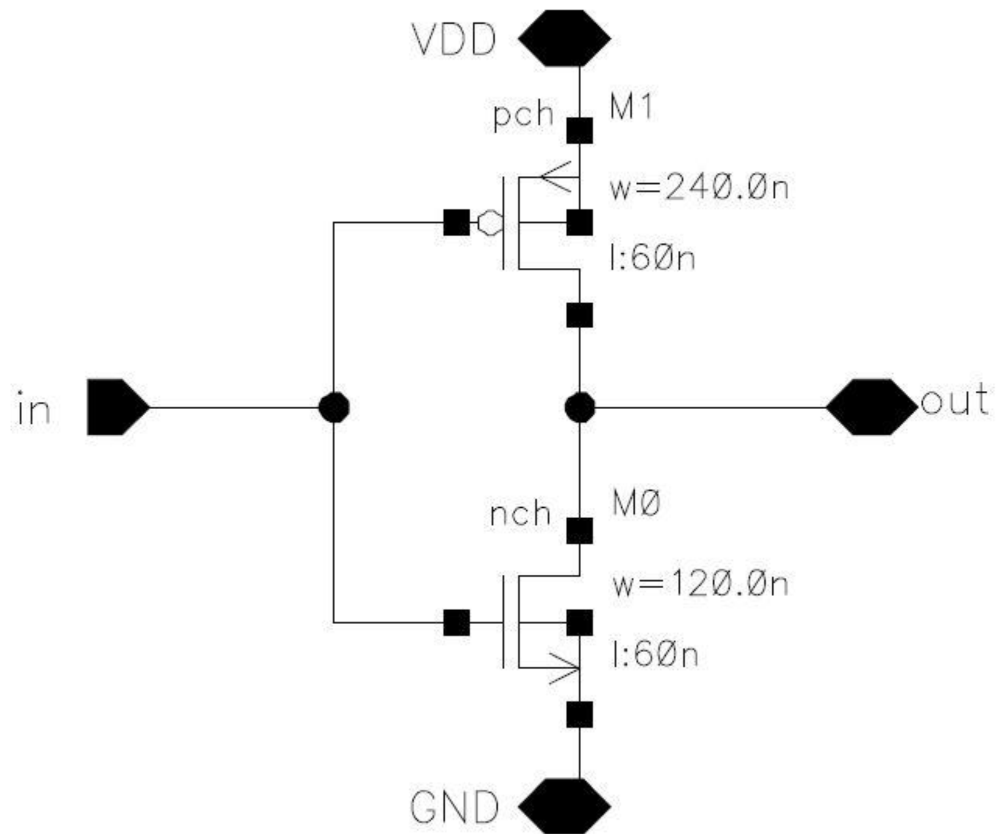


Figure 2.4: A Schematic view for the inverter

The simulation result for the inverter can be found in figure 2.5.

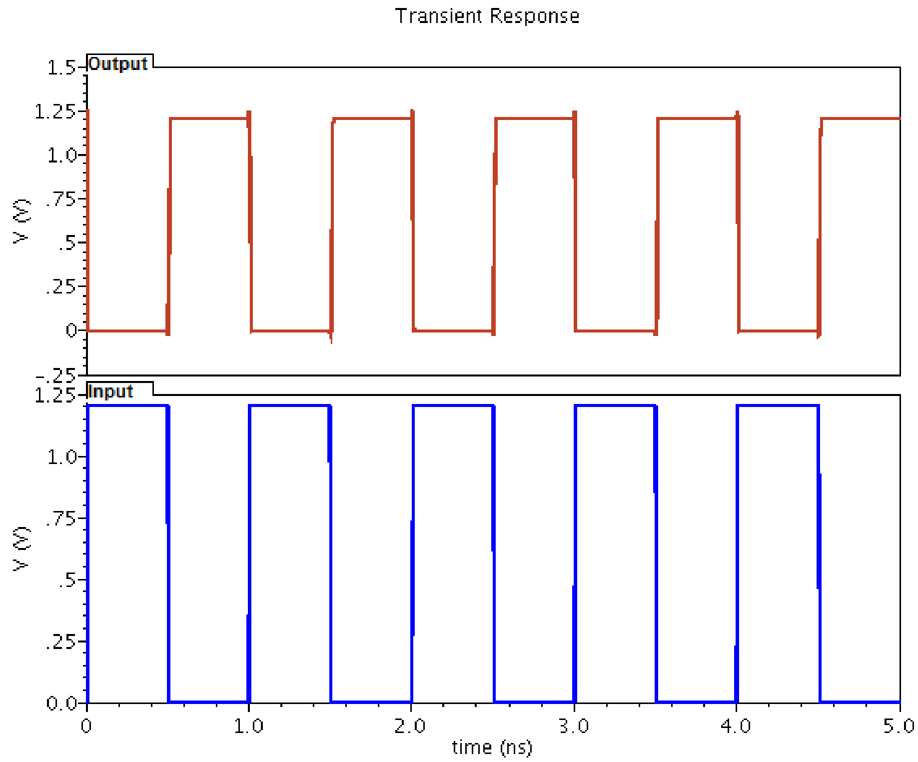


Figure 2.5: Simulation results of the inverter

Next step is to simulate the ring oscillator as cascaded inverters without any delay cells. Simulation result for this step is shown in figure 2.6.

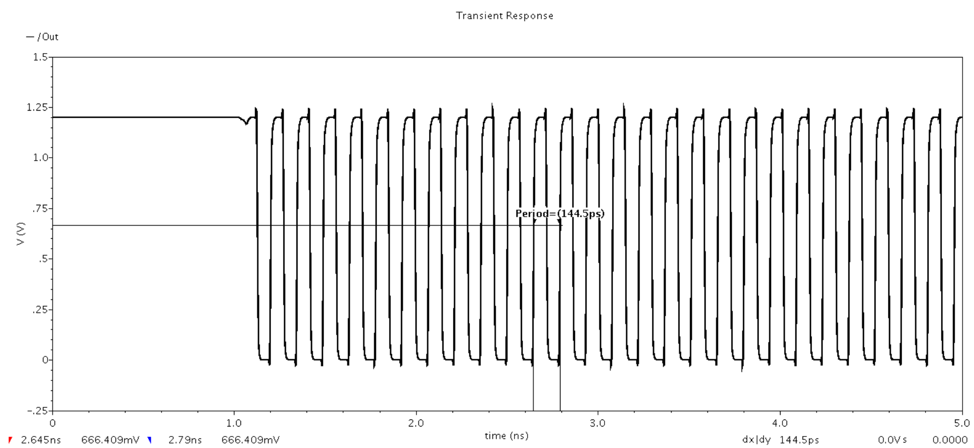


Figure 2.6: Simulation results for the ring oscillator
without delay cells

The problem with the simulation results of this ring oscillator is that, the output frequency is in range of GHZ (Period = 144.5 ps) not MHZ and our lock range is from 100 MHZ to 300 MHZ. To solve this problem, we added delay cells to the internal nodes of the ring oscillator. The delay cells to be added to the ring oscillator are Hysteresis Delay Cells (HDC). Each HDC cell consists of two cross coupled inverters. The schematic view of the HDC cell can be found in the figure 2.7.

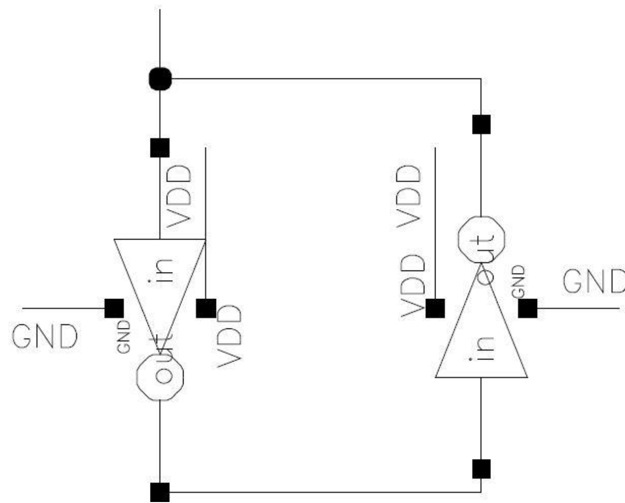


Figure 2.7: A schematic view of the HDC

The schematic view of the ring oscillator with HDC cells attached to it, is in figure 2.8.

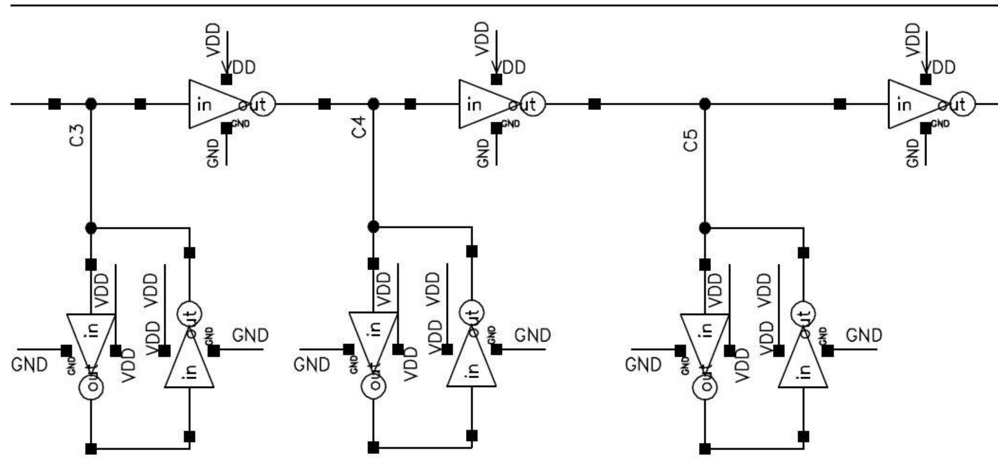


Figure 2.8: The ring oscillator with HDC cells

The simulation result for this modified ring oscillator is in figure 2.9 and it increased the period of oscillation from 144.5 ps to 678.3 ps.

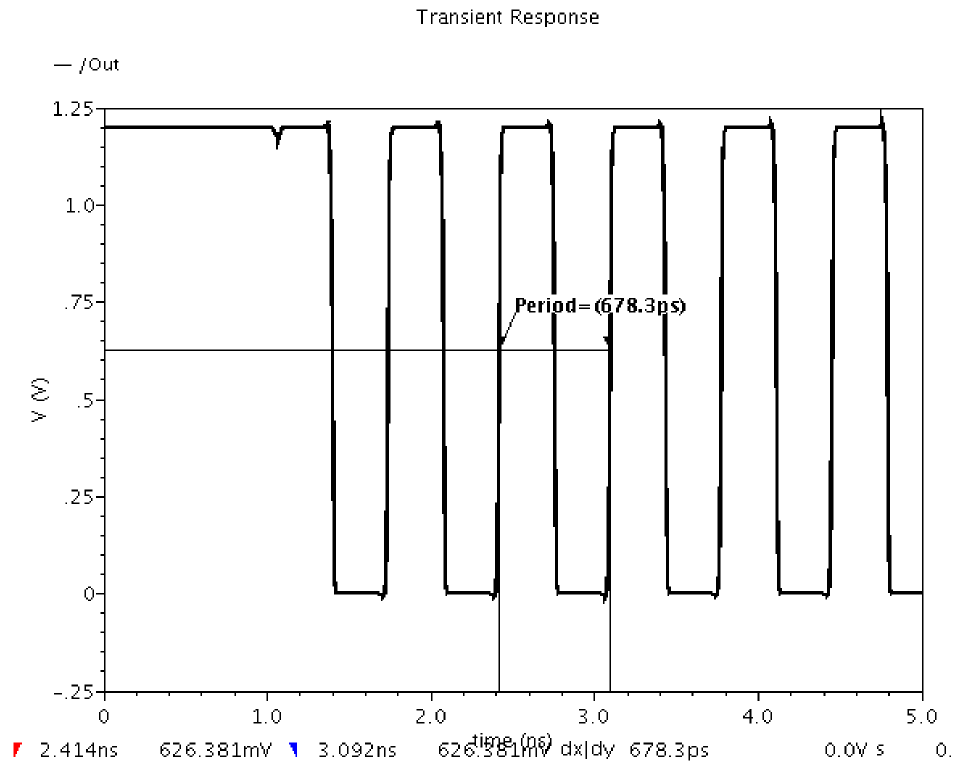


Figure 2.9: Simulation results for the ring oscillator
with HDC cells

2.1.2 A fine tuning stage (DCV Array)

For the proposed DCO, we use a fine tuning stage to give a step change in the period of oscillation of about 48 ps. The DCV cell is a NAND based delay cell. A schematic view of this DCV cell can be found in figure 2.10.

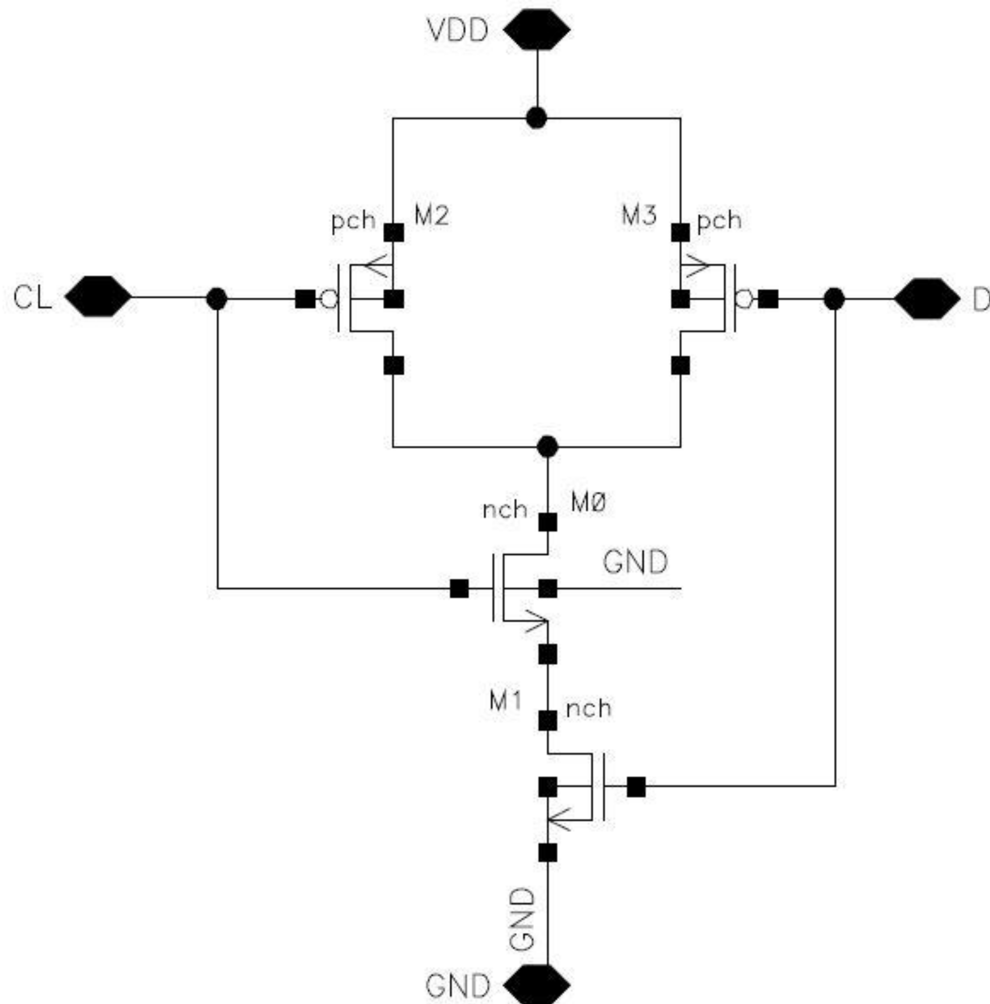


Figure 2.10: A schematic view for the DCV cell

The idea of operation of this cell is that, the gate capacitance seen from node CL (It refers to the load capacitance and it is connected to the output node of each inverter in the ring oscillator) can be changed according to the gate voltage applied to the node D (it refers to digital input bit of the DCO). The formula of

the resulting gate capacitance for this NAND based cell in both cases (when D is high and when it is low) is as follows in figure 2.11:

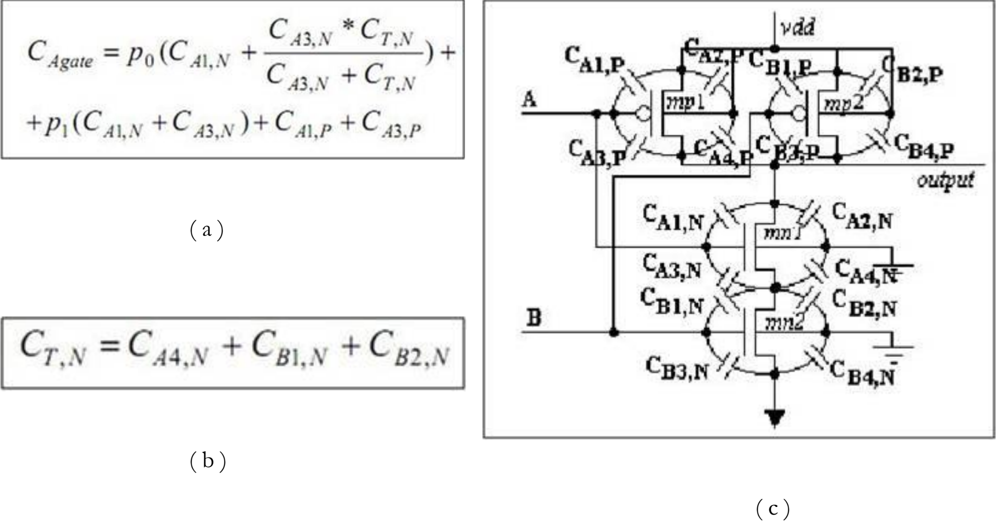


Figure 2.11: The gate capacitance of NAND gate

Where p0 is probability second input (B) to be equal the ZERO and p1 is probability second input to be equal the ONE (p0 + p1 = 1). The simulation result in figure 2.12 illustrates changing the gate capacitance with the gate voltage (CL) in two cases, when D is high and when it is low. From this result, it seems that we can achieve high capacitance for the case when the digital input bit is high (D=1) and we can get a low capacitance when it is low (D=0). Increasing the load capacitance for each node of the ring oscillator output means increasing the delay as the value of RC constant will be increased. For the ring oscillator and for a typical inverter, the propagation delay can be calculated from the following formula:

$$T_p = 0.69 C_L \left(\frac{R_{\text{eqp}} + R_{\text{eqn}}}{2} \right)$$

Where CL is the output capacitance of the inverters of ring oscillator which is the gate capacitance of the delay cells. Reqp and Reqn are the equivalent resistances of the PMOS and NMOS transistors respectively.

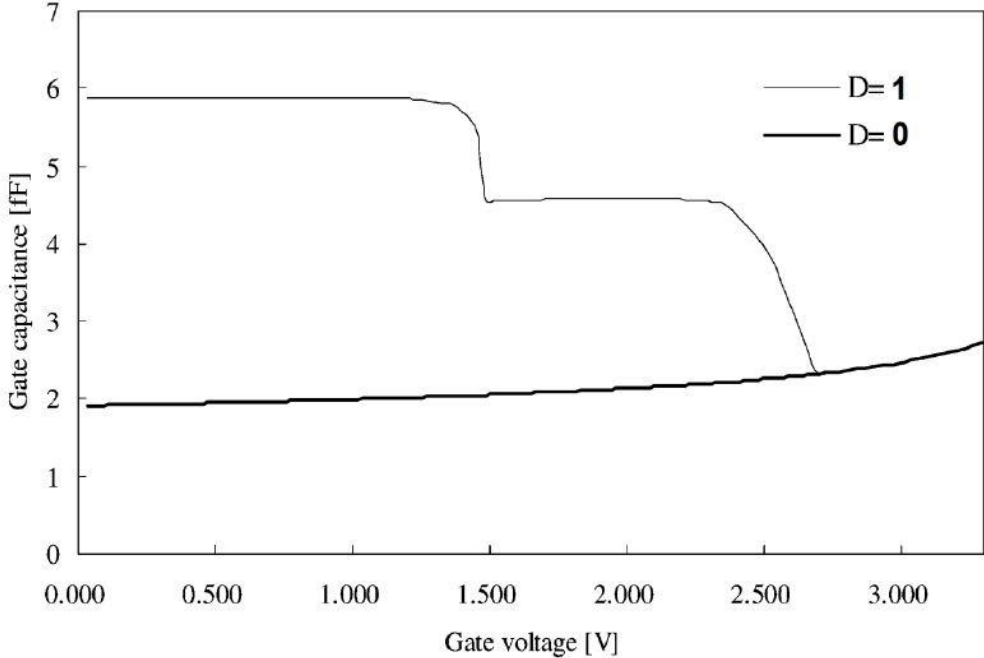


Figure 2.12: Simulation results for the NAND based DCV Cell

We use this NAND based DCV cell as a building element for a DCV block in the fine tuning stage. Every twelve DCV cells are connected to a single input which is the digital input bit (D). The output of this block consists of six nodes from C1 to C6. These nodes are connected to the corresponding outputs of six inverters in the ring oscillator. A block diagram for the DCV block is in figure 2.13.

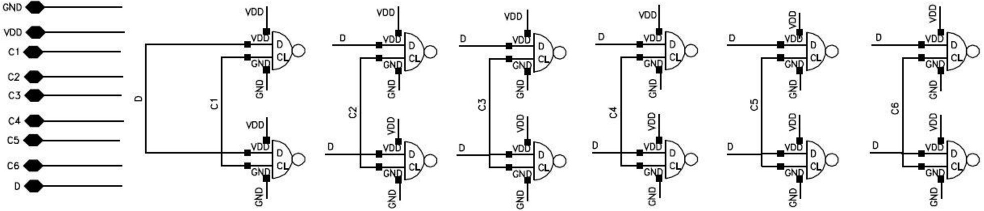


Figure 2.13: A block diagram for the DCV building block

We then use this DCV block to construct the DCV array consisting of sixteen DCV blocks. All outputs of the sixteen blocks (C1 to C6) are connected to the same six nodes of the ring oscillator but each input from the DCV blocks is connected to a different external digital bit so, for the DCV array of the fine tuning stage we have a digital word of sixteen bits. Figure 2.14 is the internal structure of the DCV array:

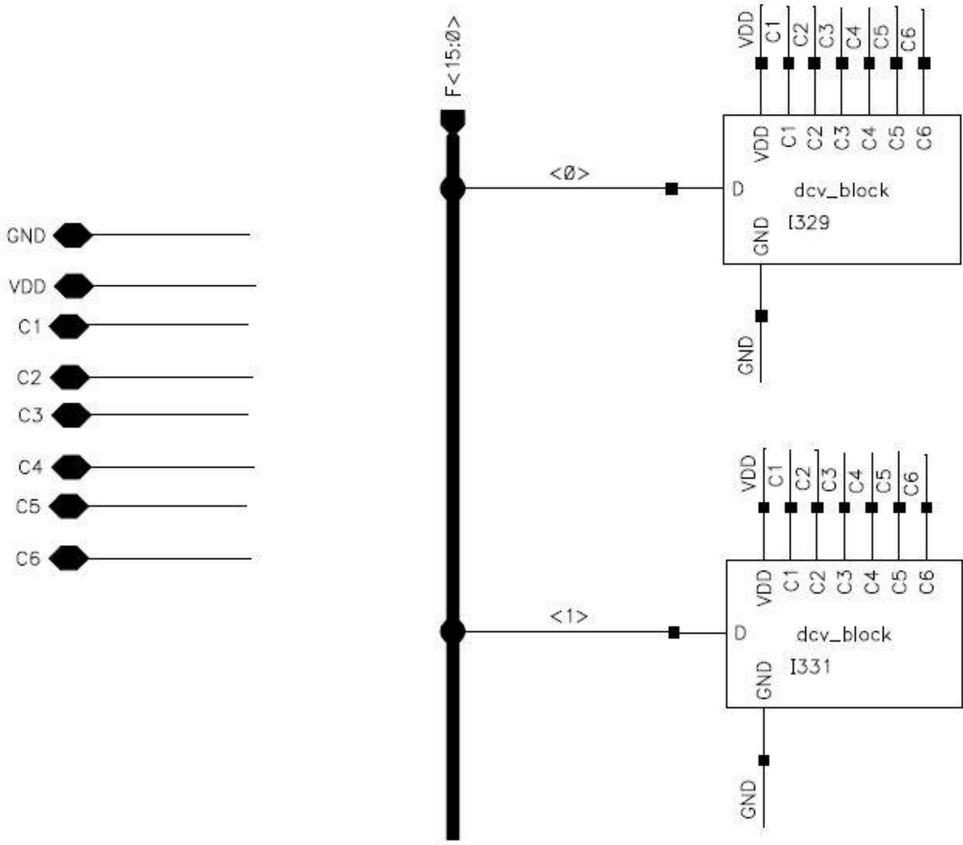


Figure 2.14: DCV array internal structure

A symbol view of this DCV array is shown below in figure 2.15:

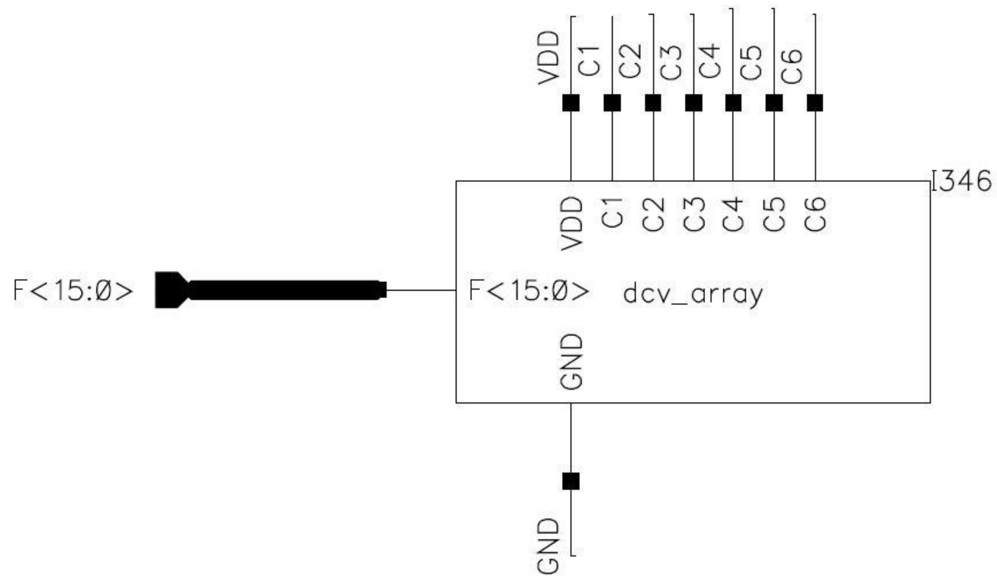


Figure 2.15: A symbol view for the DCV array

An important thing to notice here is that, although we achieved a fixed step in period of about 48ps for fine stage, the step in output frequency is not fixed because the relation between the frequency and period is not linear. For the same period step we get many frequency steps depending on the location of this period step in time access. For example, our DCO period range is from 3.3 ns to 10 ns consider adding a period step to the first period in range, from 3.3 ns to 3.348 ns (3.3ns+48ps), 3.3 ns corresponds to a frequency of 303 MHZ and 3.348 ns corresponds to a frequency of 298.686 MHZ so a period step of 48 ps from 3.3 ns to 3.348 ns causes a frequency step of 4.314 MHZ. Let's consider the same period step added to another period in another location in time access, for the last period of output oscillation from the DCO which is 10 ns, the period before this one is 9.952 ns (10 ns – 48 ps), 9.952 ns corresponds to a frequency of 100.482 MHZ and 10 ns corresponds to a frequency of 100 MHZ, so the same period step of 48 ps when added to the period 9.952 ns we get a frequency step of 0.482 MHZ. For these two cases we get two different frequency steps of 4.314 MHZ and 0.482 MHZ although the period step of fine stage is constant. Conclusion is that, although the period step for fine stage is constant we will get different frequency steps (not fixed) because the relation between frequency and

period is not linear. Figure 2.16 illustrates why we get different frequency steps for the same period step.

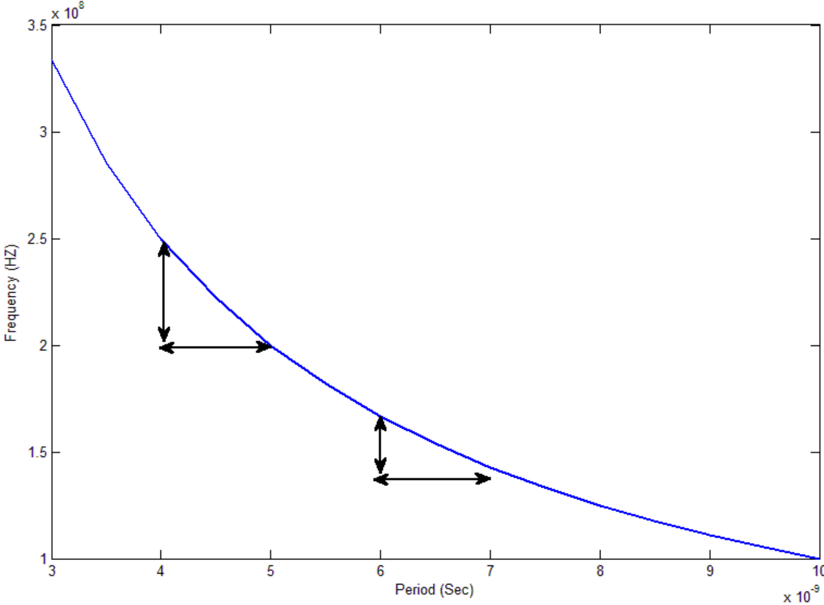


Figure 2.16: Period step Vs. frequency steps

2.1.3 A coarse tuning stage (DCV2 Array)

After using the fine tuning DCV array we achieved a relatively small step change in output period of the DCO and consequently a relatively small frequency steps. For the coarse stage we need to get larger frequency steps to reduce the lock time of the PLL, to achieve these larger frequency steps we have to use delay cells with larger period steps than the fine tuning stage (48 ps), so we used another delay cell to get this larger period step. The delay cell used for coarse tuning stage is also based on the NAND configuration but with a transmission gate in the beginning. The enable line for this transmission gate is the external digital input bit (D) and the input to it is the load capacitance node (CL) which is connected to the output of each inverter in the ring oscillator. Figure 2.17 is a schematic view for this delay cell.

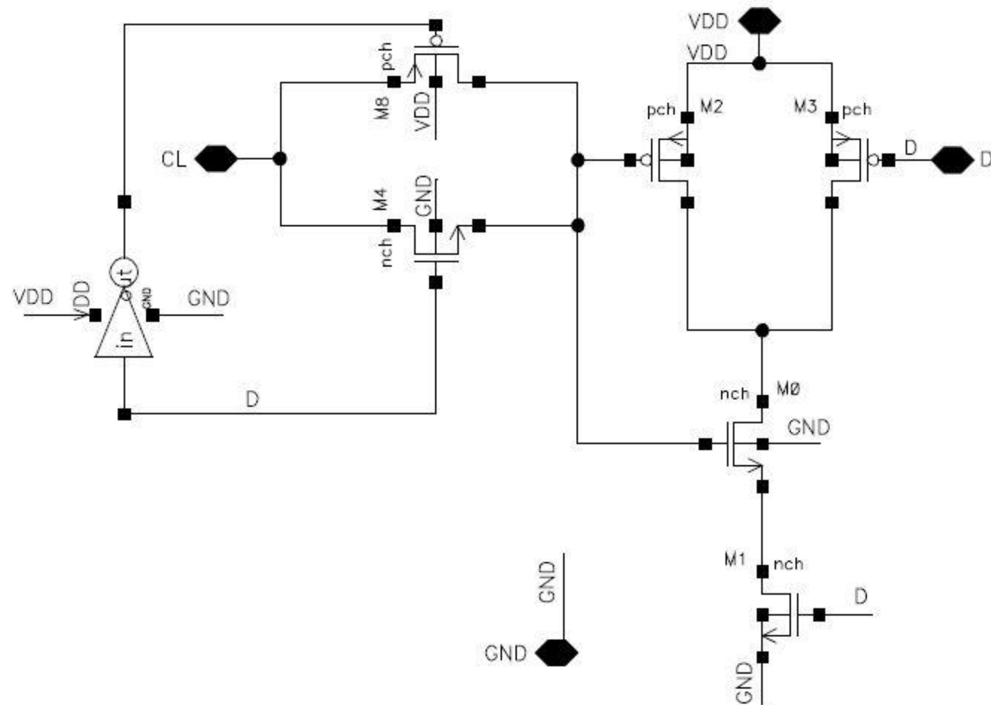


Figure 2.17: A schematic view for the DCV2 cell

When the external input is low (D=0), the CL node will be disconnected from the NAND cell and introduces low capacitance and consequently low delay. When the external input is high (D=1), the CL node is now connected to the NAND cell and can see the gate capacitance of it. The two cases of D=0 and D=1 here are different from those in fine tuning stage, as in fine tuning stage the CL node was connected to NAND cell in both cases that's why the period step in fine stage was relatively small. In coarse stage, the CL node is connected only when D=1 so we can say that, in coarse stage the node CL can see the capacitance of the NAND cell or it cannot see it, so the period step here is larger than the period step in fine stage. For coarse stage we achieved a period step of 380 ps. Another advantage for using the DCV2 cell is that, it helped us increase the largest output period of the DCO (10 ns) without affecting the smallest period (3.3 ns) by changing the sizing of the NAND cell in this DCV2 cell

We use this NAND based DCV2 cell as a building element for a DCV2 block in the coarse tuning stage. Every eight DCV2 cells are connected to a single input

which is the digital input bit (D). The output of this block consists of four nodes from CL1 to CL4. These nodes are connected to the corresponding outputs of four inverters in the ring oscillator. These four inverters are following the six inverters used in the fine stage. A block diagram for the DCV2 block is in figure 2.18.

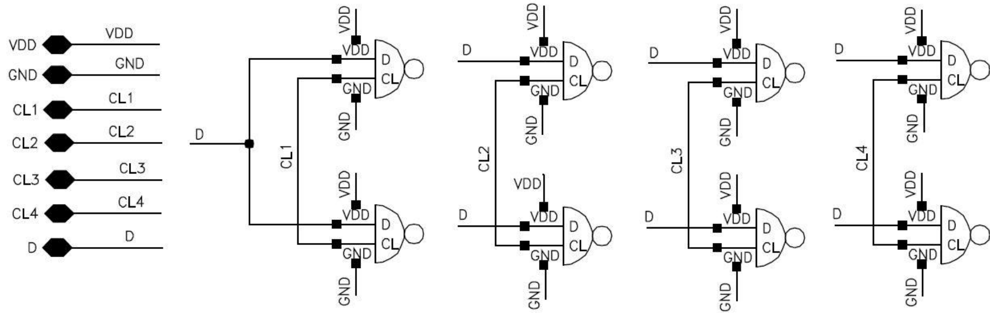


Figure 2.18: DCV2 block

We then use this DCV2 block to construct the DCV2 array consisting of sixteen DCV2 blocks. All outputs of the sixteen blocks (CL1 to CL4) are connected to the same four nodes of the ring oscillator but each input DCV2 blocks is connected to a different external digital bit so, for the DCV2 array of the coarse tuning stage we have a digital word of sixteen bits. The following figure is the internal structure of the DCV2 array (Figure 2.20).

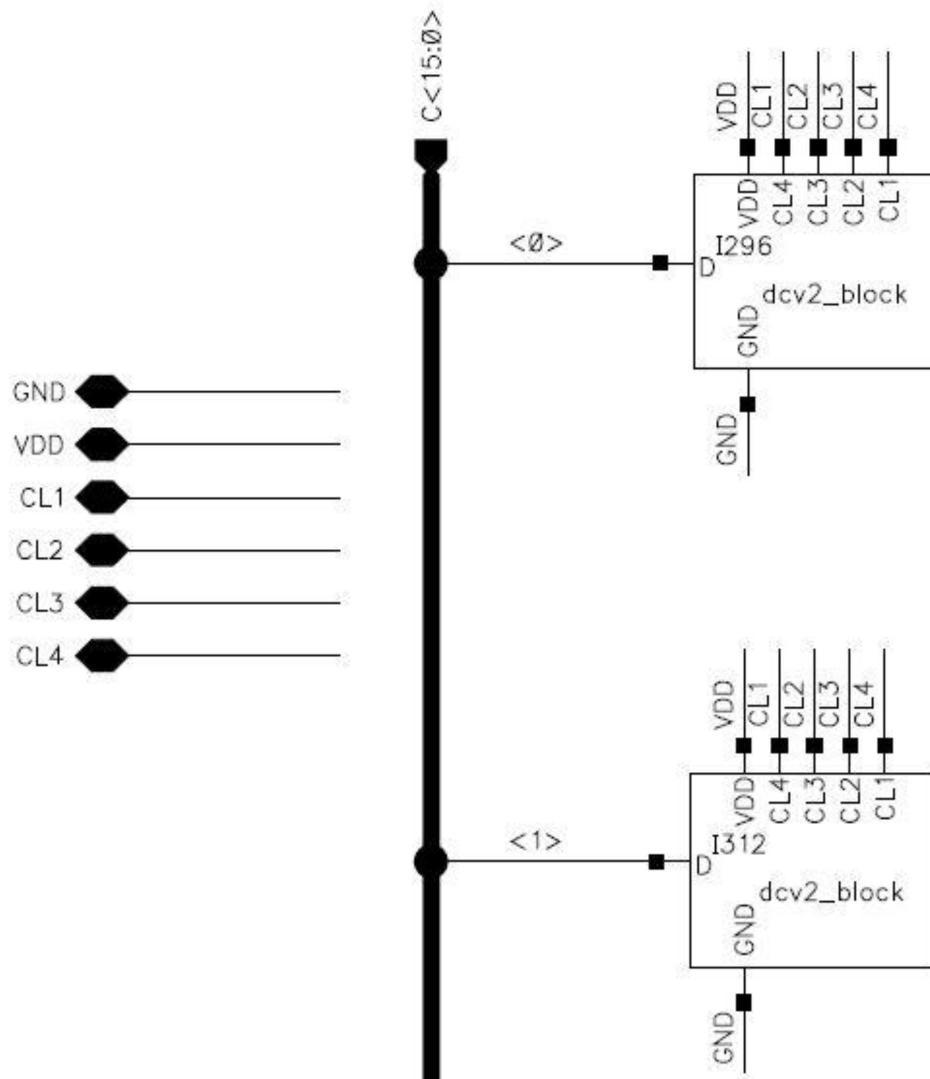


Figure 2.19: DCV2 array

A symbol view of this DCV2 array is shown in figure 2.20.

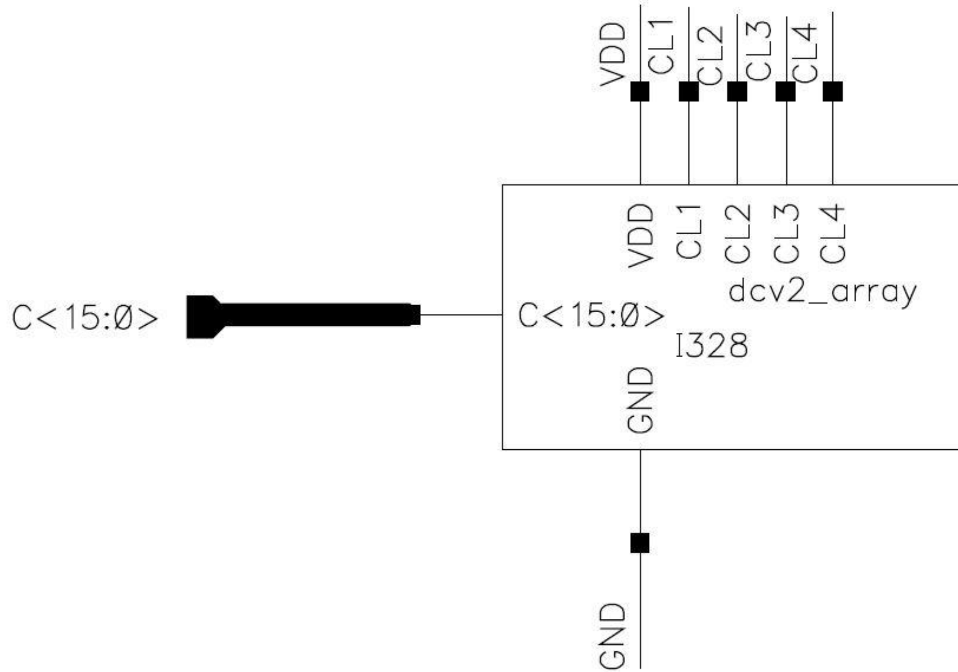


Figure 2.20: A symbol view for the DCV2 array

2.1.4 Conclusion and final results

In this section we introduced the proposed DCO which is consisting of three main blocks:

1. A ring oscillator
2. A fine tuning stage
3. A coarse tuning stage

The output period for this DCO ranges from 3.3 ns to 10 ns which is equivalent to a lock range from 100 MHZ to 300 MHZ. We used HDC cells to add a fixed delay to the ring oscillator and DCV cells for both fine and coarse tuning stages to add a programmable delay. The fine tuning stage gives a small step (48 ps) in period and consequently small steps in frequency. The coarse tuning stage gives large period step (380 ps) and consequently large steps in output frequency.

The following table in figure 2.21 shows different output periods of the DCO according to different values for the digital words of both fine and coarse tuning stages.

Control bits Fine Word Coarse Word	Period ns	Frequency MHz	Period step ns	Freq. step MHz
00000000 00000000 00000000 00000000	3.289	304.878	----------------	------------------
00000000 00000000 00000000 00000001	3.669	272.553	0.38	32.325
10000000 00000000 00000000 00000001	3.718	268.961	0.049	3.592
10000000 00000000 00000000 00000011	4.1	243.902	0.382	25.059
11000000 00000000 00000000 00000011	4.147	241.138	0.047	2.764
11000000 00000000 00000000 00000111	4.53	220.75	0.383	20.388
11100000 00000000 00000000 00000111	4.578	218.435	0.048	2.315
11100000 00000000 00000000 00001111	4.961	201.572	0.383	16.863
11110000 00000000 00000000 00001111	5.009	199.641	0.048	1.931
11110000 00000000 00000000 00011111	5.38973	185.538	0.38073	14.103
11111000 00000000 00000000 00011111	5.43695	183.927	0.04722	1.611
11111000 00000000 00000000 00111111	5.82148	171.778	0.38453	12.149
11111100 00000000 00000000 00111111	5.869	170.387	0.04752	1.391
11111100 00000000 00000000 01111111	6.25404	159.897	0.38504	10.49
11111110 00000000 00000000 01111111	6.29857	158.766	0.04453	1.131
11111110 00000000 00000000 11111111	6.68375	149.617	0.38518	9.149
11111111 00000000 00000000 11111111	6.73115	148.563	0.0474	1.054
11111111 00000000 00000001 11111111	7.1095	140.657	0.464	7.906
11111111 10000000 00000001 11111111	7.1607	139.651	0.0512	1.006
11111111 10000000 00000011 11111111	7.5419	132.593	0.3812	7.058
11111111 11000000 00000011 11111111	7.59375	131.687	0.05185	0.906
11111111 11000000 00000111 11111111	7.9758	125.379	0.38205	6.308
11111111 11100000 00000111 11111111	8.0248	124.614	0.049	0.765
11111111 11100000 00001111 11111111	8.41188	118.879	0.38708	5.735
11111111 11110000 00001111 11111111	8.4583	118.227	0.04642	0.652
11111111 11110000 00011111 11111111	8.84562	113.05	0.38732	5.177
11111111 11111000 00011111 11111111	8.8918	112.463	0.04618	0.587
11111111 11111000 00111111 11111111	9.2775	107.788	0.3857	4.675
11111111 11111100 00111111 11111111	9.3259	107.228	0.0484	0.56
11111111 11111100 01111111 11111111	9.7117	102.969	0.3858	4.259
11111111 11111110 01111111 11111111	9.7605	102.454	0.0488	0.515
11111111 11111110 11111111 11111111	10.146	98.561	0.3855	3.893
11111111 11111111 11111111 11111111	10.194	98.097	0.048	0.464

Figure 2.21: Different output periods of the DCO

The simulation result in figure 2.22 is different output waveforms for the DCO.

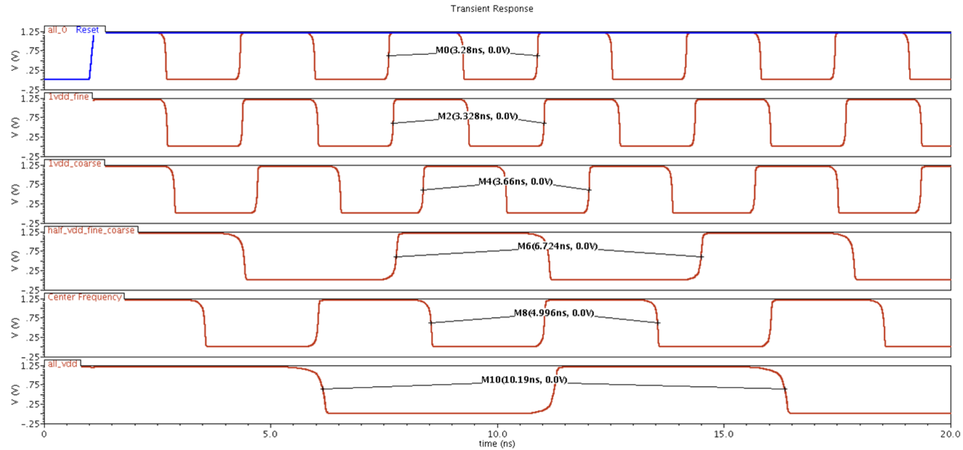


Figure 2.22: Different output waveforms of the DCO

Figure 2.23 is the period steps versus the digital code according to the above table, one coarse step change followed by one fine step change. From this figure, one can easily notice the coarse step is larger than the fine step. The start point of our range is 3.3 ns and the end point is 10 ns corresponds to the lock range for the proposed DCO (100 MHZ to 300 MHZ).

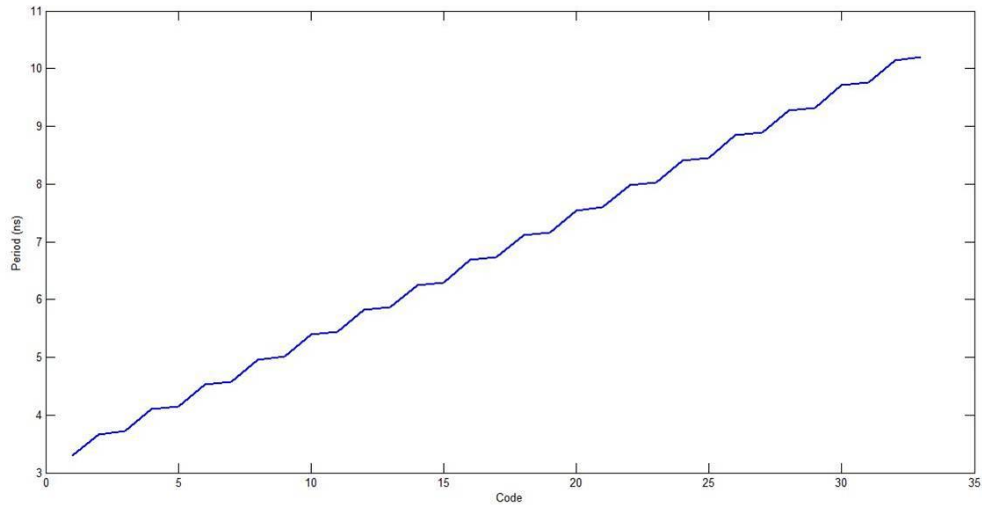


Figure 2.23: Period steps versus code

Figure 2.24 is the output frequency versus the 32-bit digital input word (16-bit word for fine stage and 16-bit word for coarse stage).

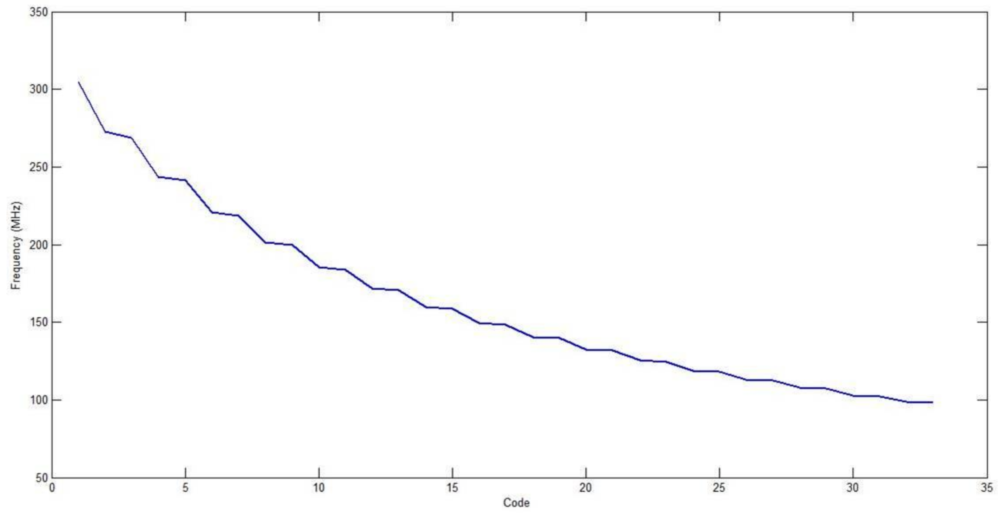


Figure 2.24: Frequency steps versus code

2.2 PFD

A phase detector is a circuit capable of delivering an output signal that is proportional to the phase difference between its two input signals Ref_Freq and DCO_Out as mentioned in figure 1.2. When the PLL moved into digital territory, digital phase detectors become popular, such as EXOR gate, the edge-triggered JK-flip flop, and the so-called phase-frequency detector (PFD).The PFD differs greatly from the other phase detector types as its name implies, its output signal depends not only on phase error but also on frequency error when the PLL has not yet acquired lock. The PFD is built from two D-flip flops, whose outputs are denoted UP and DOWN(DN) as shown in figure 2.25, these two signals are the digital representation of the phase/frequency error. The PFD can be in one of four states:

- UP=0, DN=0
- UP=1, DN=0
- UP=0, DN=1
- UP=1, DN=1

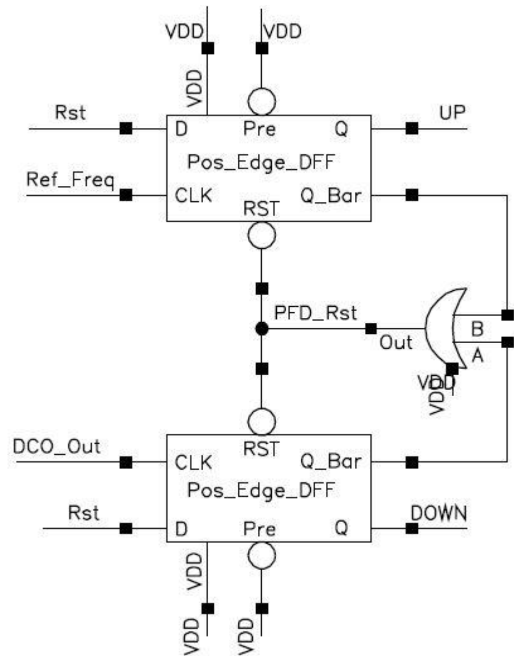


Figure 2.25: PFD schematic

The fourth state is inhibited, however, by an additional gate. Whenever both flip flops are in the 1 state, a logic low level appears at their reset inputs, which reset both flip flops. We assign the symbols -1, 0, and 1 to these three states :

- UP=0, DN=0 → state -1
- UP=1, DN=0 → state 0
- UP=0, DN=1 → state 1

The actual state of the PFD is determined by the positive-going transients of the signals Ref_Freq and DCO_Out, as explained by the state diagram in figure 2.26, a positive transition of Ref_Frq forced the PFD to go into its next higher state, unless it is already in the 1 state. In analogy, a positive edge of DCO_Out forces the PFD into its next lower state, unless it is already in the -1 state.

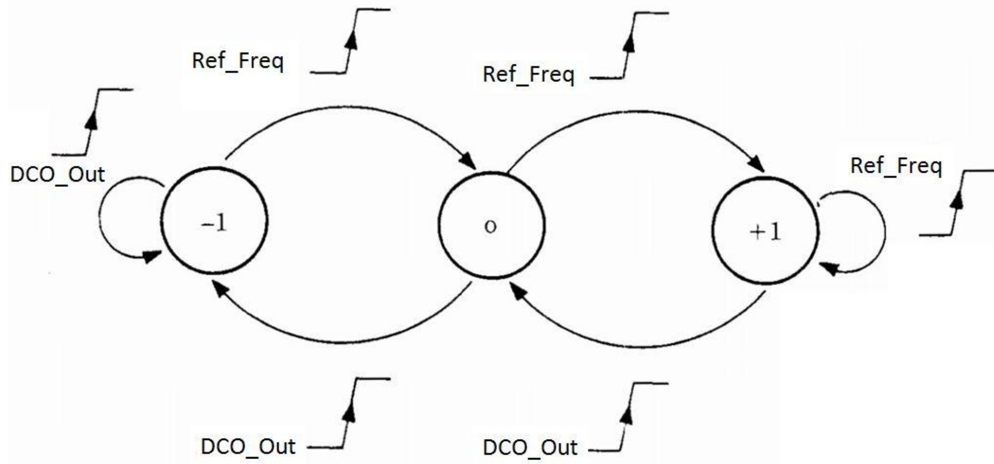


Figure 2.26: State diagram of the PFD

To see how the PFD works in a real PLL system, we consider the waveforms in figure 2.27,this figure shows the three cases:

a) First 25ns shows the case where Ref_Freq leads, therefore the PFD toggles between states 0 and 1.
b) If Ref_Freq lags as in the next 25ns, the PFD now toggles between states -1 and 0.
c) The signals Ref_Freq and DCO_Out are 'exactly' in phase; both positive edges occur at the same time; hence the PFD will stay in state 0 forever.

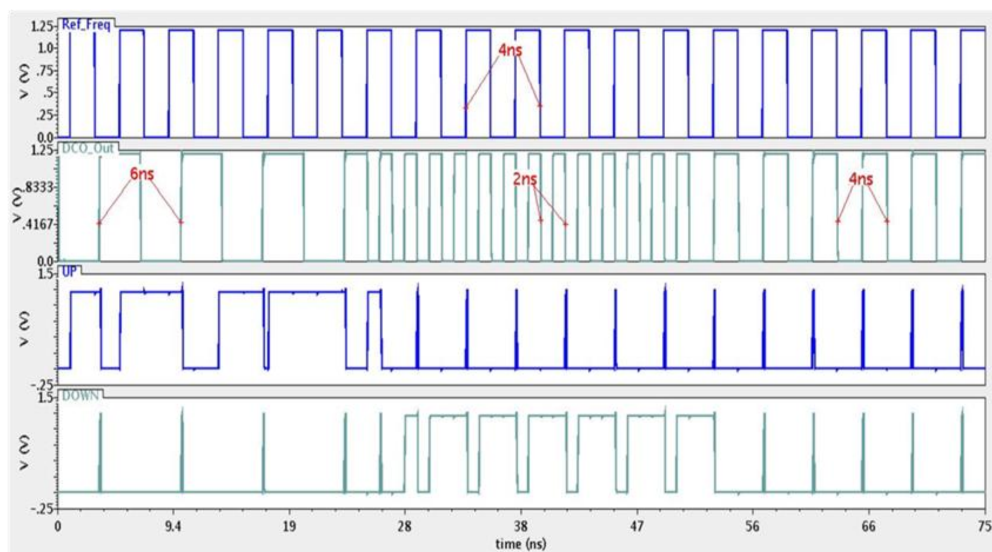


Figure 2.27: PFD simulation results

2.3 Loop Filter:

The Loop filter stage controls the output capacitance by changing the number of DCV cells that are turned on, as shown in figure 2.28 the digital control signals is used to increase/decrease the DCO frequency for a certain period of time by reducing /increasing the capacitance. If the input to the DCV is '1', it provides more capacitive load at the output. If more number of cells are on (input is '1'), then it acts as more capacitive load on the ring oscillator which reduces the DCO frequency.

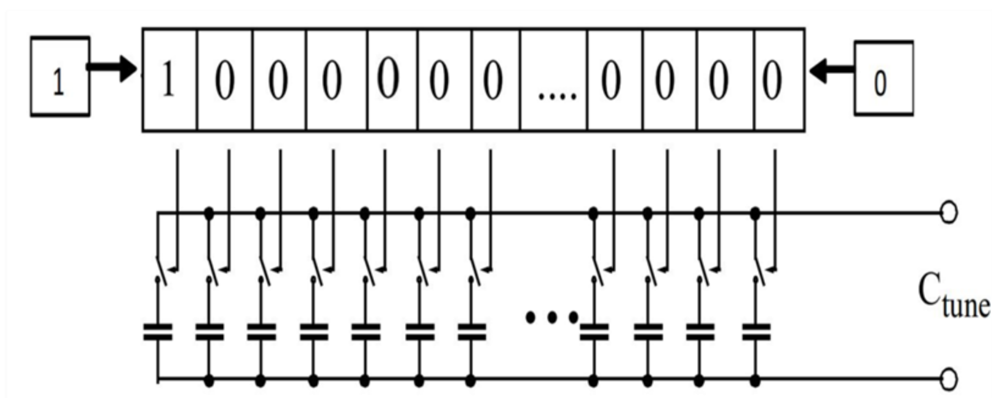


Figure 2.28: digital control signals used to switch a set of varactors

In order to control each of the fine DCV array and the coarse DCV array individually, we have used two 16-bits bi-directional loadable shift registers and here part of the schematic view for each shift register in figure 2.29

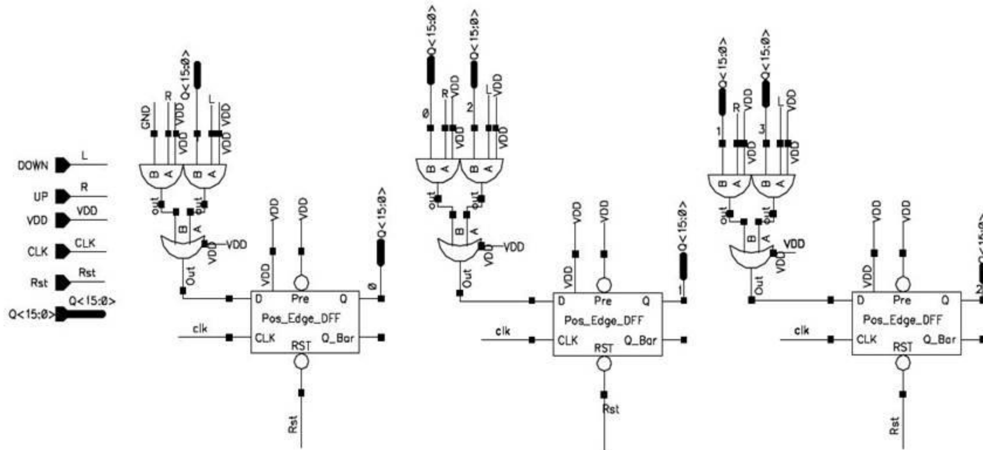


Figure 2.29: Schematic view of the shift register

Initially, 8 DCV cells of each array are on, this achieved by using asynchronous Reset and Preset signals. Depending on the up/down signals from PFD, the frequency is either increased or decreased. When phase and frequency acquisition starts, if the output of PFD is up, then the contents of the shift register are left shifted and bit ‘0’ is pushed into Q<15> and hence the capacitive load decreases and the frequency increases. Likewise, if it is down, the contents of the shift register are right shifted and bit ‘1’ is pushed into Q<0>. This reduces the frequency of the DCO as the capacitive loading at the output increases.

2.4 Overall Design

In this section the simulation results of the whole system will be introduced.

2.4.1 Extreme Reference frequency

First let the input (reference frequency) signal be an extreme, let's say the minimum frequency in the desired range (100-300MHz) as in figure 2.30.And by knowing

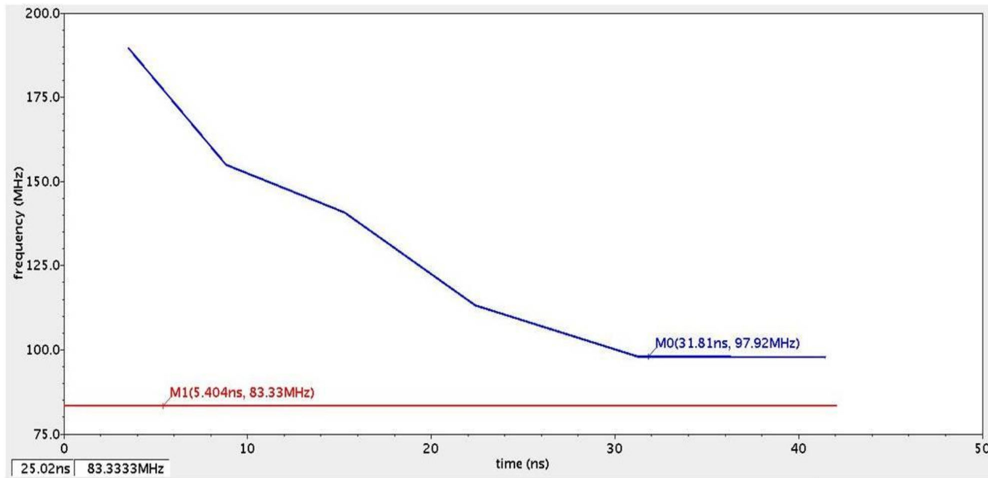


Figure 2.30: Reference = 100MHz

that the DCO frequency initially equals to the center frequency (200MHz), then the shift register should get only DOWN pulses from the PFD, which activates all the DCV cells and introduces the lowest frequency .now what about the lock time ? Actually this depends on two factors, the:

i. CLK used for the loop filter: this means the rate of changing in DCO frequency at a certain time. It's clear that we need to increase the frequency of this CLK to get smaller lock time.

ii. DCO Delay : this means time needed by the DCO to change its frequency after one step delay as shown in figure 2.31, where one DCV cell is deactivated after two cycles, the effect of this step appears after exactly two cycles, this time is considered as the DCO delay, thus the CLK above in part (i) should take in consideration this delay to get the true UP/DN pulse after the new change in the DCO frequency.

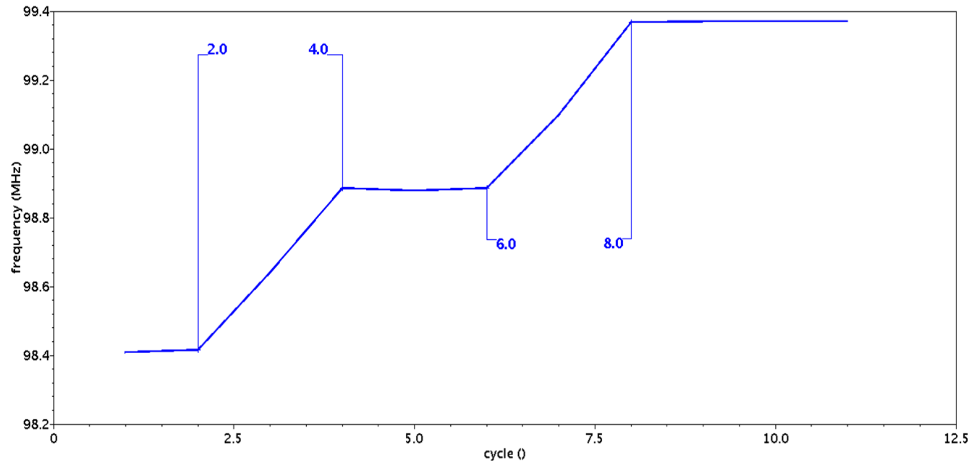


Figure 2.31: DCO delay

2.4.2 Intermediate Reference frequency

Now let's consider this case, reference = 250MHz as below in figure 2.32.

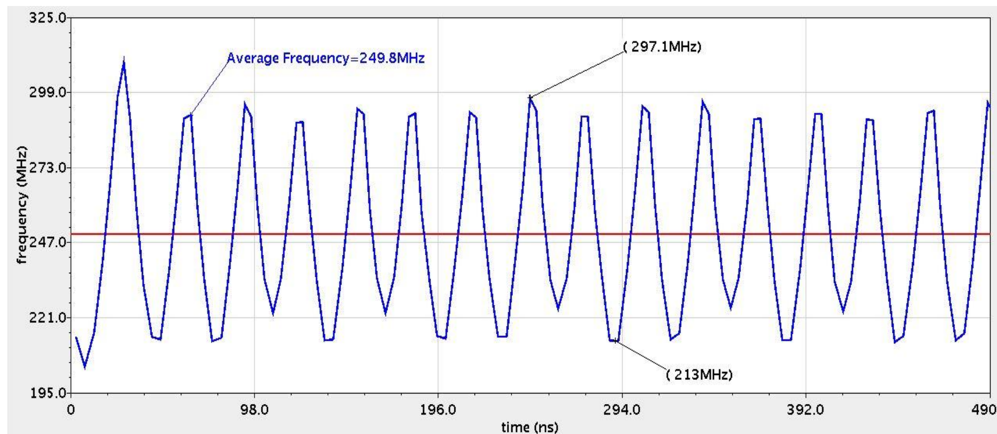
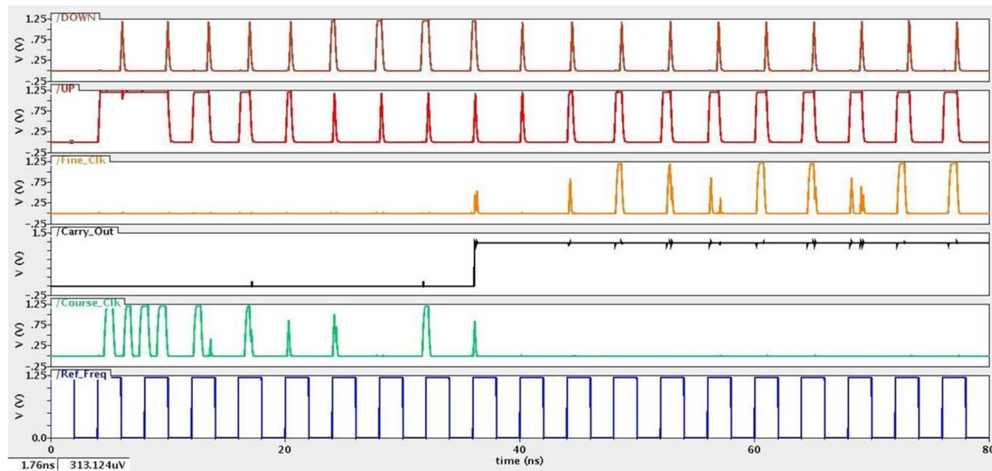


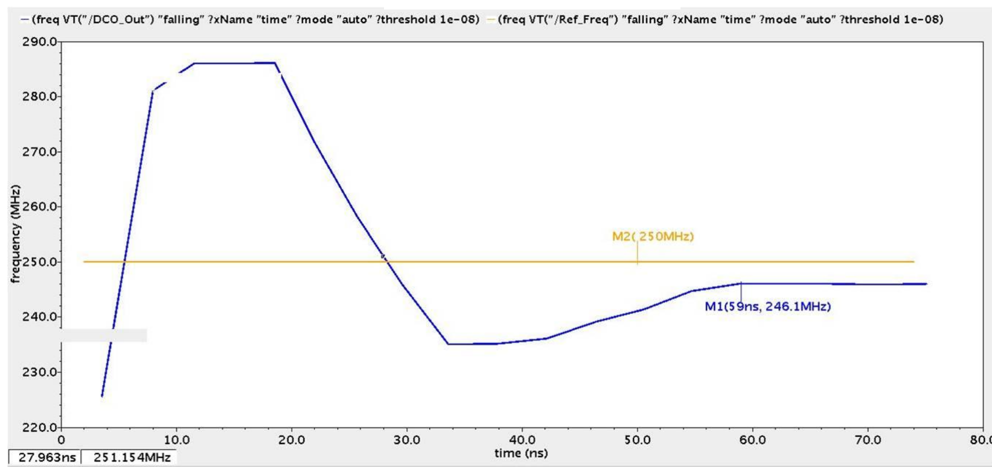
Figure 2.32: Reference= 250MHz

It's clear that average frequency is almost equals to the reference, but the problem that appears here due to the wide steps of the course stage (up to 31MHz), which results to these oscillations with a very high peak-to-peak value. Simply to solve this problem, the coarse stage must be stopped after being operating separately from the fine stage. This is applied using a 4bit Counter to count the maximum number of steps needed by the coarse stage, which is 16 at the worst case, after that a carry signal is used to turn off the coarse shift register and activates the fine one. The simulation results of this idea are

shown in figures 2.33a, 2.33b and 2.33c.



a) Control signals



b) Frequency response

Course stage		
Course SH_Reg	Fine SH_Reg	Frequncy (MHz)
00000000 00000001	00000000 00001111	260.28
00000000 00000011	00000000 00001111	236.85

Fine stage		
Course SH_Reg	Fine SH_Reg	Frequncy (MHz)
00000000 00000011	00000000 00000001	245.21
00000000 00000011	00000000 00000000	246.13

c) Digital words of Both SRs

Figure 2.33: Effect of using counter

In this example the reset signal of the whole system was designed to start the DCO oscillations at the center frequency, this is achieved by activating four DCV cells from each array, to calculate this frequency as explained in section 2.1 :

Period (ps) = 3270 + Coarse_1's × (380) + Fine_1's × (48) = 5ns (200MHz).

Through the first stage the fine SR is stopped as shown in figure 2.33a and 2.33c, and at the end of this stage the DCO frequency was undecided between two frequencies around the reference corresponding to the digital words which clarified in figure 2.33c, after that the course stage stopped using the carry signal at 38n as shown in figure 2.33a.At the same time the fine stage started, this can be observed through the frequency response in figure 2.33b ,where the very small slopes appears after the 38ns.

2.4.3 Frequency step response

It was necessary to consider in our design that the Reference signal frequency may be changed during the loop, because this PLL targets a low power clock and data recovery system. But in order to detect this change, the loop filter won't be a simple shift registers as it now because a certain controlling circuit must be added. Actually we have replaced the coarse SH by a frequency counter circuit which is one of Digital Instruments that can be used to measure signal frequency and period, the basic idea is illustrated in the following figure:

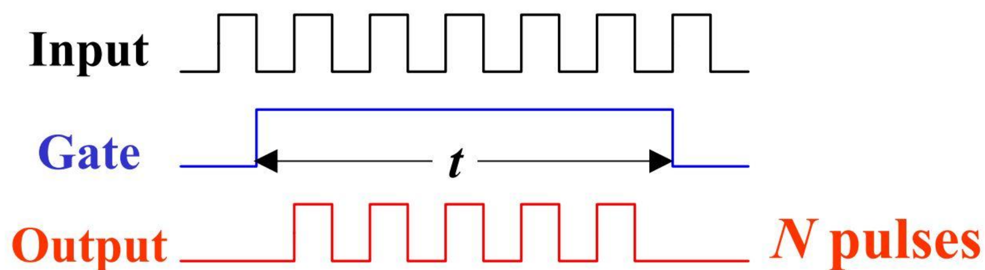


Figure 2.34: Frequency counter

We have implemented this idea by generating the Gate signal (figure 2.34) from the reference frequency, and the input clock source (time base) signal which used to trigger the counter from a ring oscillator which already discussed in section 2.1.1.

After counting the N pulses mentioned in figure 2.34, this number is therefore mapped to the 16 bit to control the coarse DCV array. Note that at the end of each counting period, the counted value should mapped synchronously and the counter should cleared. In the following the whole system simulation result after using the frequency counter :

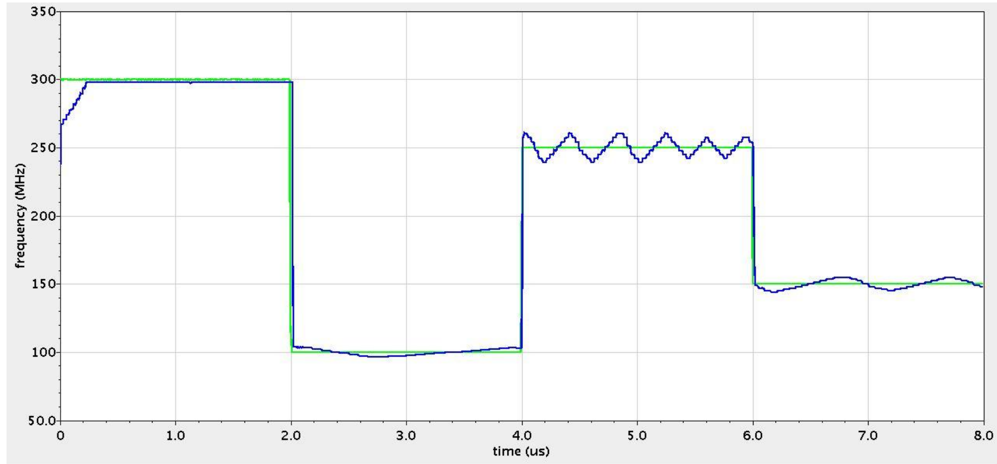


Figure 2.35: frequency step response

After this modification on the system to become capable of reacting with the step system response, the consumed power is increased from 0.25 to **0.6** mWatt due to the high frequency clock added. Also the lock time at the worst case does not exceed 300ns. Now there is one more enhancement needed

The last improvement needed, is to reduce the oscillation of the fine stage around the reference, this is achieved by slowing its clock frequency. Figure 2.36 shows the result after this modification.

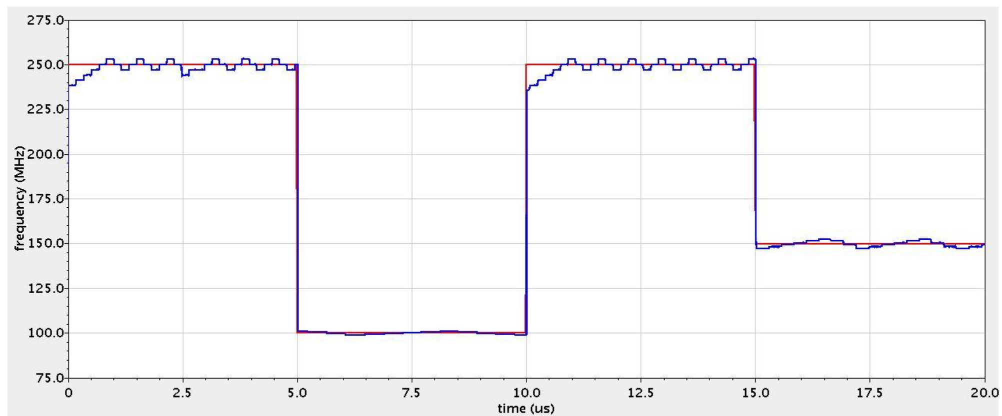


Figure 2.36: Eliminating the oscillations

As expected, slowing the fine SR led to the expansion of the time lock until the time of 1us as figure 2.36 shows.

2.4.4 Jitter calculation :

Figure 2.37 shows the eye-diagram of DCO output clock when locked at 300 MHz plotted using Cadence tools. In this eye-diagram, each and every cycle of the DCO output clock are overlapped on one clock period (after the DCO clock is locked to the reference) and the maximum deviation that can be obtained from the graph is measured as peak-to-peak jitter. Number of cycles that are taken into account are 100. The delay is measured at 50% voltage levels and the period jitter determines how noisy and stable the oscillator output signal is.

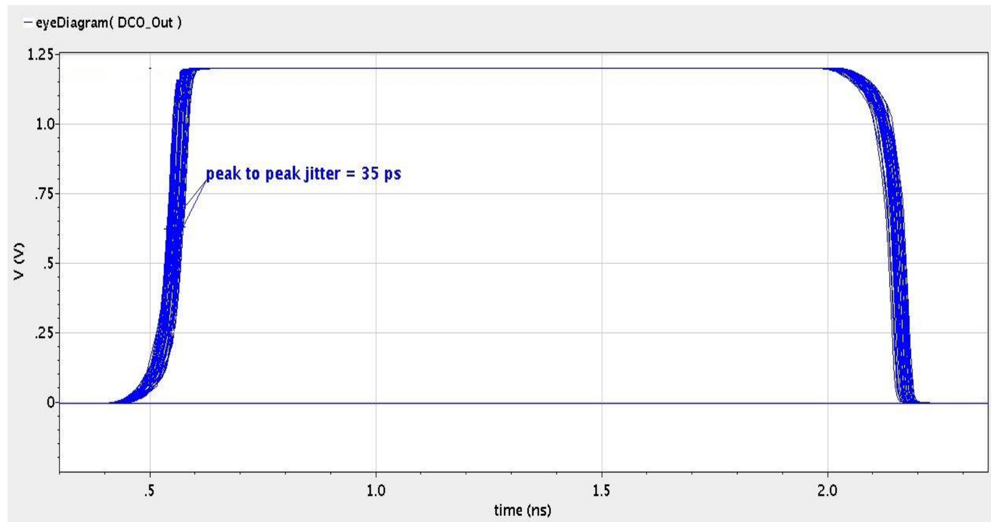


Figure 2.37: Eye-diagram of the ADPLL

The peak-to-peak jitter for this implementation when the feedback signal is locked at 300 MHz is 35 ps.

Chapter 3

STANDARD CELLS APPROACH

Standard Cells Approach means that the targeted block will be written with one of the Hardware Description Language HDL codes such as Verilog or VHDL, and then can be translated into an hardware circuit using Standard Cells library. Thanks to AMS we are able to simulate and test the targeted block with analog blocks in cadence environment.

One big advantage of using such approach is that the designer is not have to deal with the block gates at the transistor level and check the sizing of the logic gates . This approach also made the layout step very easy and effective in area.

3.1 PFD

As mentioned before, the Phase and Frequency Detector (PFD) will be used as a phase detector to detect the phase and the frequency difference between the reference signal and the output signal of the DCO.

First of all, we will write the code of our PFD using Verilog programming language , we will design a block which has:

- Two input ports (reference signal and DCO output).
- Two output ports (Up and Down).

The Up signal indicates that the system should increase the DCO frequency (i.e. the reference frequency is higher than the DCO frequency) and the Down signal indicates that the system should decrease the DCO frequency.

```
module dff (input d,clk,reset,output reg q);
always@(posedge clk,negedge reset)
if(~reset)
q<=1'b0;
else
q<=d;
endmodule

module pfd (input refSignal,dcoSignal,output up,dn);
wire intReset;
assign intReset=~(up&dn);
dff up_dff(1'b1,refSignal,intReset,up);
dff dn_dff(1'b1,dcoSignal,intReset,dn);
endmodule
```

And using Synplify PRO we translated our PFD into the corresponding standard cells schematic as shown in figure 3.1

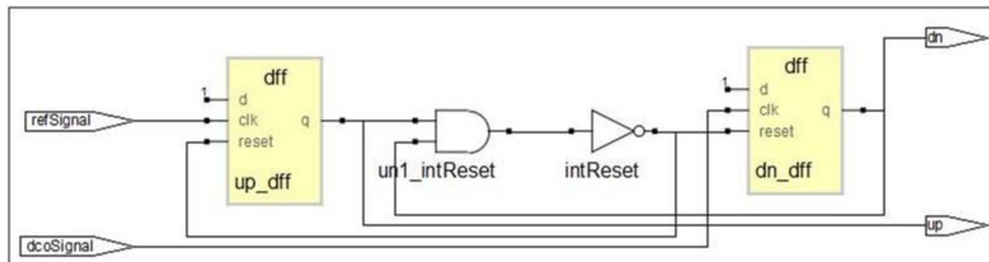


Figure 3.1: PFD schematic

This PFD can be imported as Verilog code in cadence also , see appendix A for more details.

3.2 Loop Filter

The Loop Filter used here is simply a 16 bits shift register ,following the same procedure of PFD ,we got the schematic view of the Loop Filter as shown in figure 3.2.

```
module loopFilter ( input up,dn,reset, output reg [15:0] q);
wire ored;
assign ored = up | dn;
always @(posedge ored, negedge reset)
if(~reset) //active low reset(level sensitive)
q<= 16'b11110000_00000000;
else if(up & ~dn)
q <= {q[14:0],1'b0};
else if(dn & ~up)
q <= {1'b1,q[15:1]};
endmodule
```

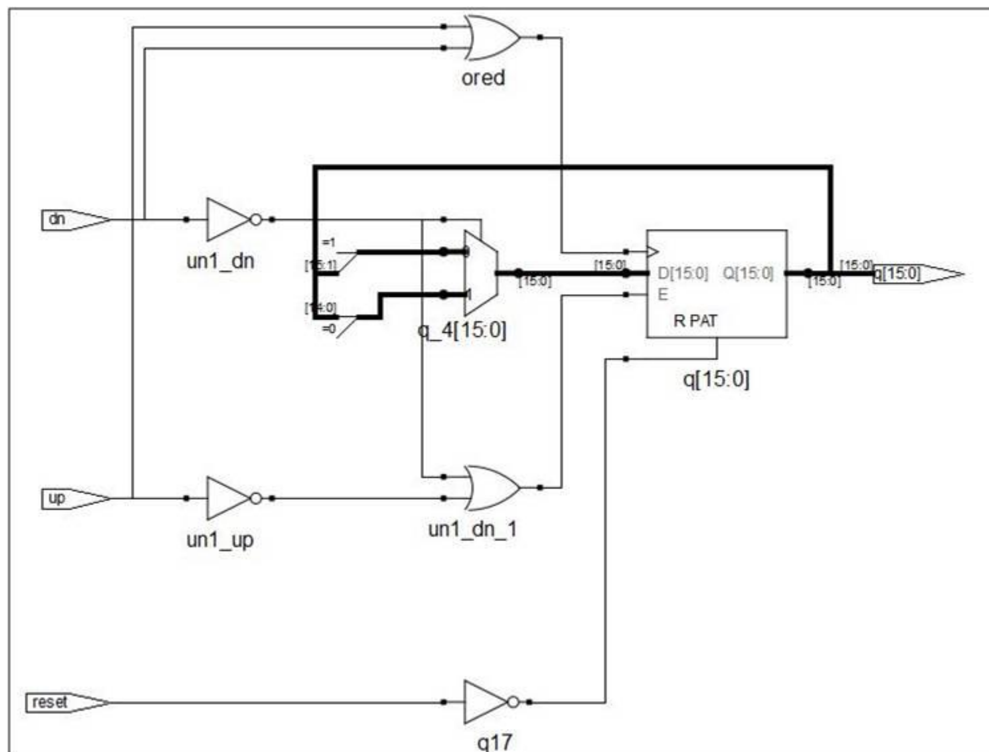


Figure 3.2: Loop Filter schematic

We used two Loop Filters, one as a Coarse and the other as a fine, the Coarse one is responsible for the large step change in the DCO output frequency while the fine one is responsible for the small step change.

3.3 PFD and Loop Filter

Now, we have PFD and Loop Filter as functional blocks, so we can connect them together as shown in figure 3.3.

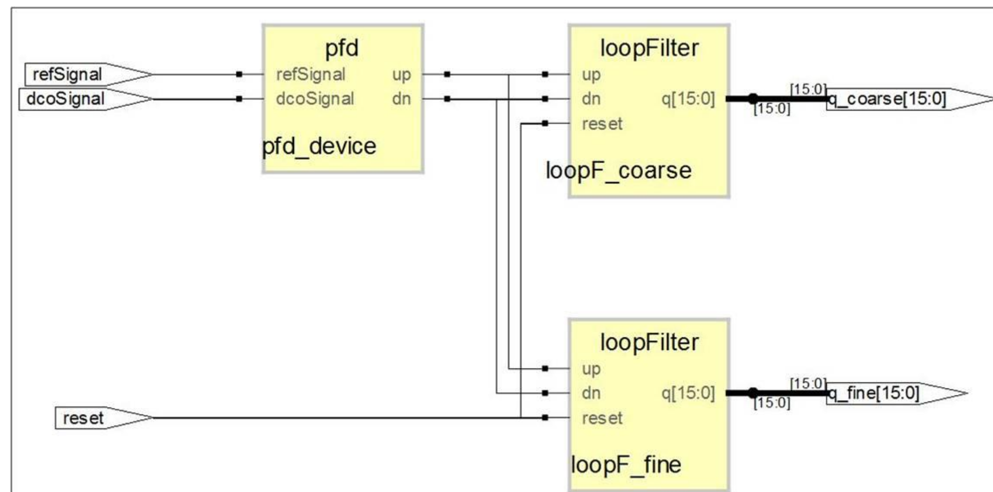


Figure 3.3: PFD and Loop Filter

Now , we have our functional block (PFD + Loop Filter) so we can import it to cadence environment and connect it with the custom designed DCO.

3.4 Overall Design

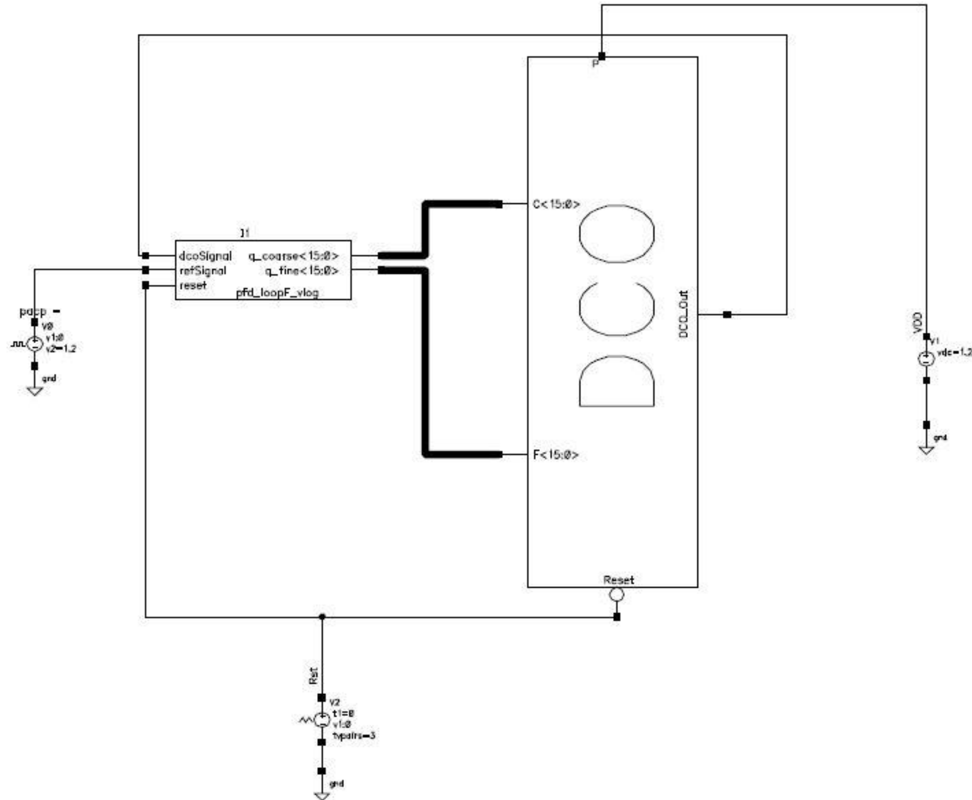


Figure 3.4: Overall ADPLL (standard cells)

Using AMS we can simulate analog and digital(functional) blocks together and check the functionality of our ADPLL and the result was as shown in figure 3.5

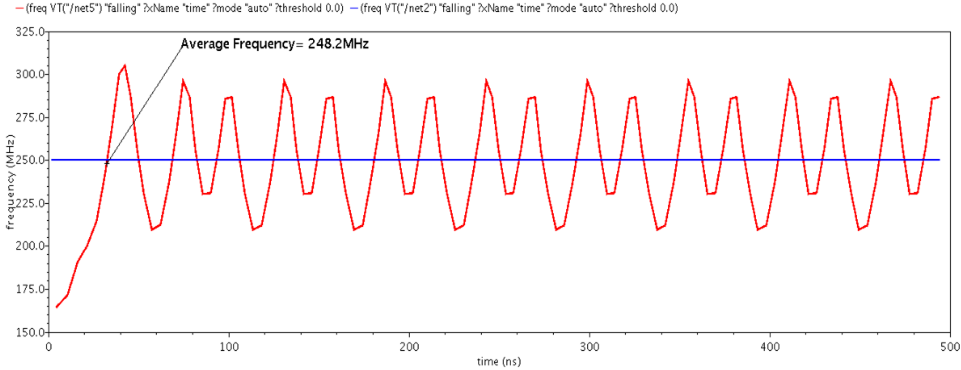


Figure 3.5: AMS of the overall ADPLL without the counter

As we can see that the DCO output frequency is oscillating around the reference frequency and that is due to the coarse large steps, so we need to stop the coarse shift register to eliminate the large oscillations.

4 bits counter is used for that purpose and when the counter carry bit activated the coarse shift register is turned off and the fine one is turned on as shown in figure 3.6

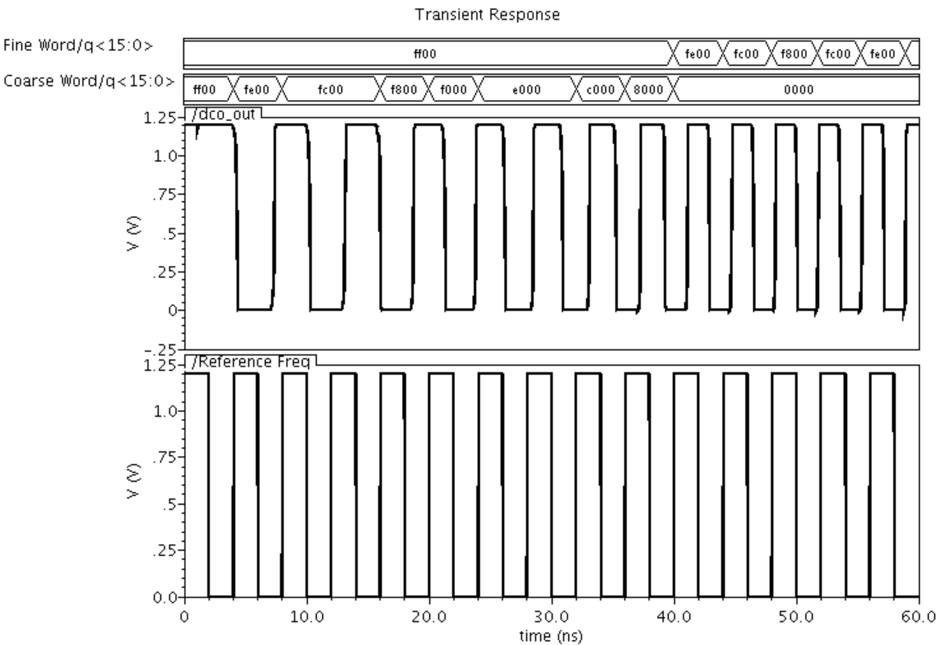


Figure 3.6: AMS simulation of the overall ADPLL
with counter

And the result of the Overall ADPLL as a functional (behavioral) block was as shown in figure 3.7.

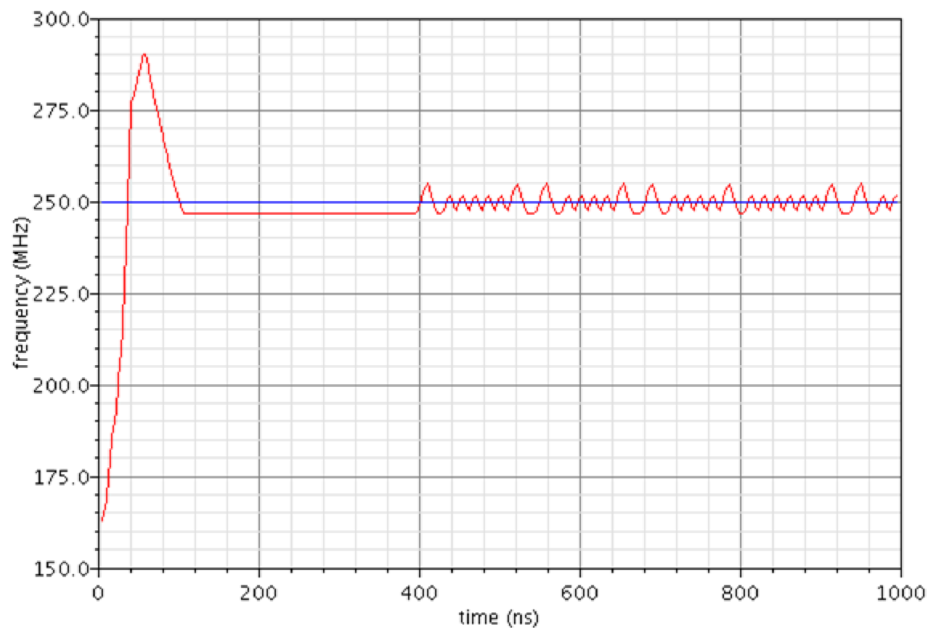


Figure 3.7: AMS of the ADPLL with counter.

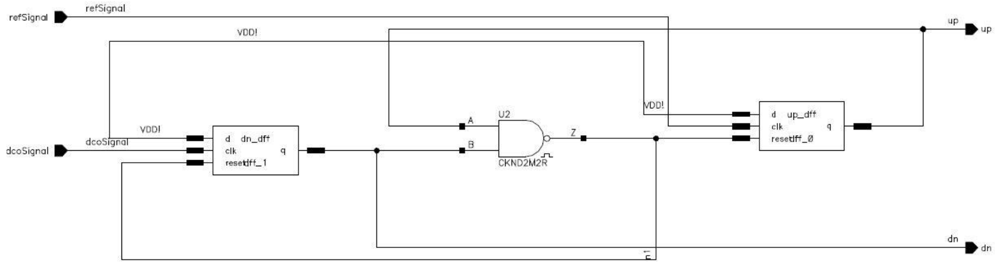
From the above figure we found that the lock time is less than 150 ns which means that our lock time restriction (lock time < 10 us) is satisfied.

But, the problem is, the above design assuming the reference frequency is fixed and will not exposed to a step change.

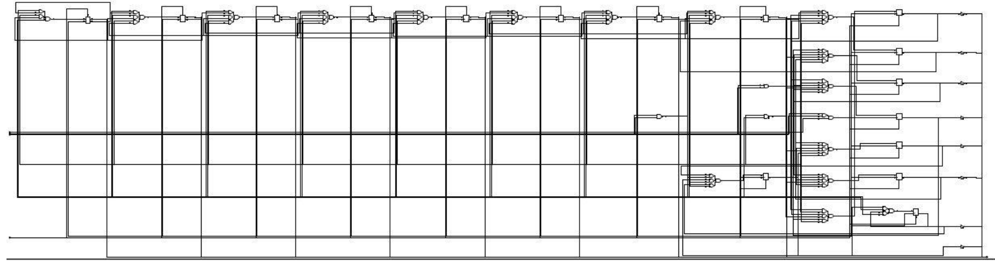
All the above simulation results is done considering the functional behavioral of the PFD and the Loop Filter (as code only).

We did the technology mapping using Design Compiler by converting the functional Verilog code of the PFD and the Loop Filter into a netlist to generate a mapped code which contains the needed standard cells to achieve the block functionality.

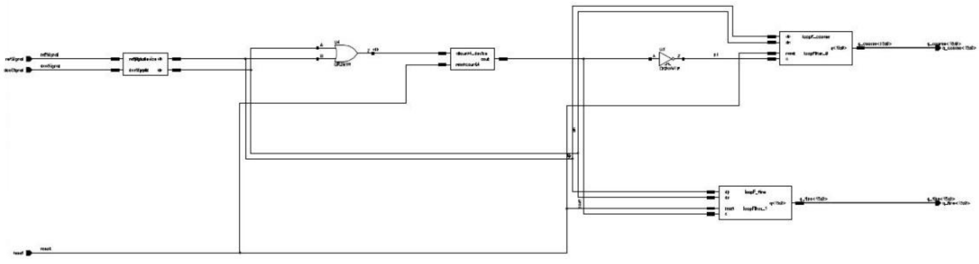
We extracted the transistor level schematic of the PFD and the Loop Filter successfully using the standard cells of UMC65nm digital kit as shown in figure3.8.



a) Imported schematic of PFD in cadence



b) Imported schematic of Loop Filter in cadence



c) Imported schematic of PFD and Loop Filter together in cadence

Figure 3.8: The standard cells of both PFD and Loop Filter

As an example we took a snap shot of one standard block inside the PFD schematic in figure 3.8.a. above

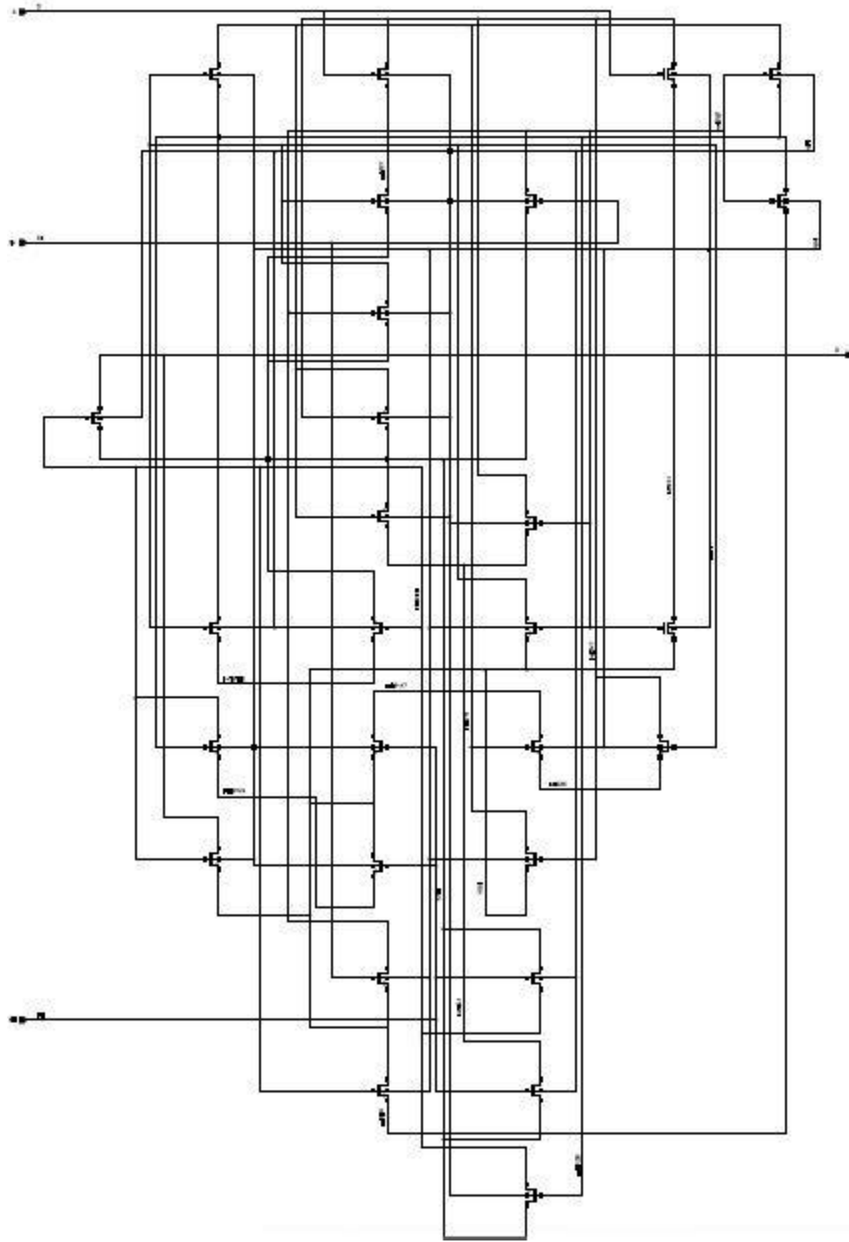


Figure 3.9: The transistor level of the standard D flip flop in cadence.

LAYOUT

The last step of the design flow is the layout, first of all DCO is considered as the core of the ADPLL so as we started the design in the schematic scope with the DCO , we will start with the DCO in the layout scope and we believe that the range will be changed due to the capacitance and the resistance added by the layout (i.e. more delay).

4.1 DCO

The proposed DCO depends mainly on the delay cells (HDCs and DCVs) and because of the layout, the delay is not the same as schematic any more.

4.1.1 DCV

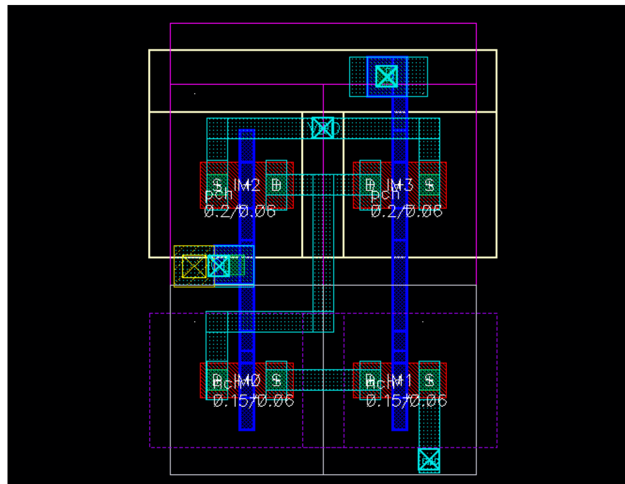


Figure 4.1: DCV layout

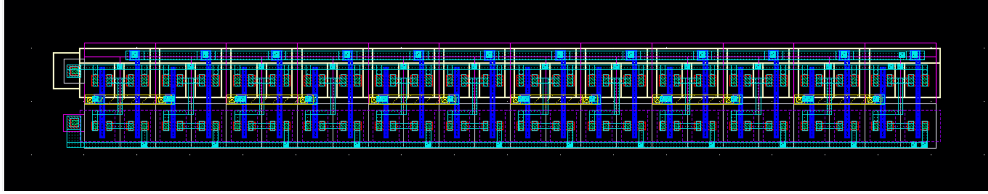


Figure 4. 2: DCV block layout

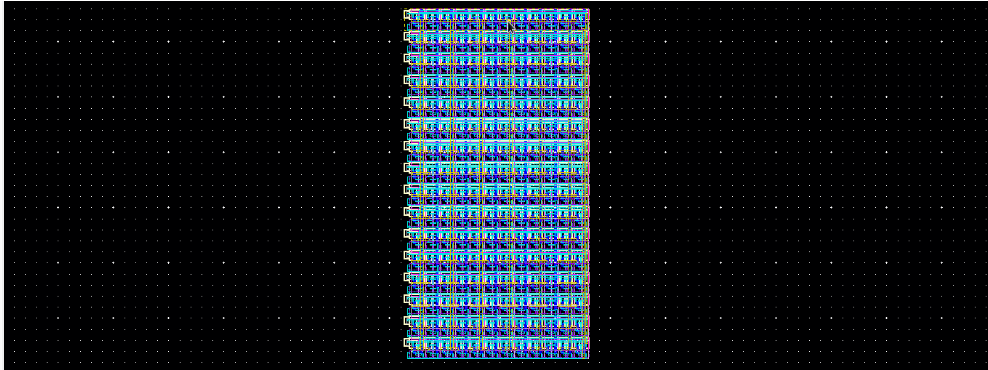


Figure 4. 3: DCV array layout

4.1.2 DCV2

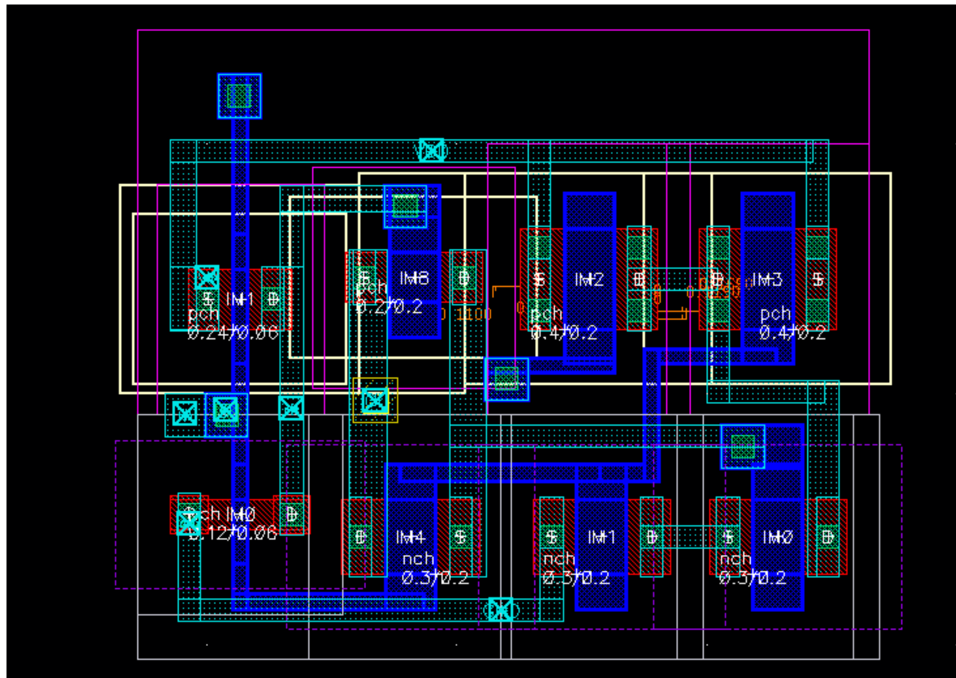


Figure 4. 4: DCV2 layout

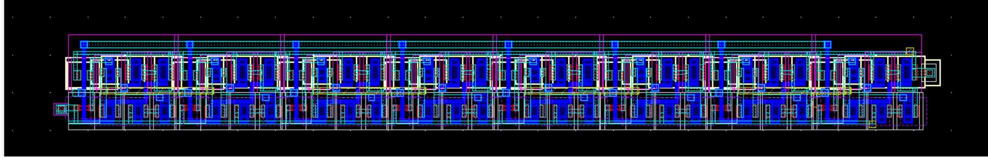


Figure 4. 5: DCV2 block layout.

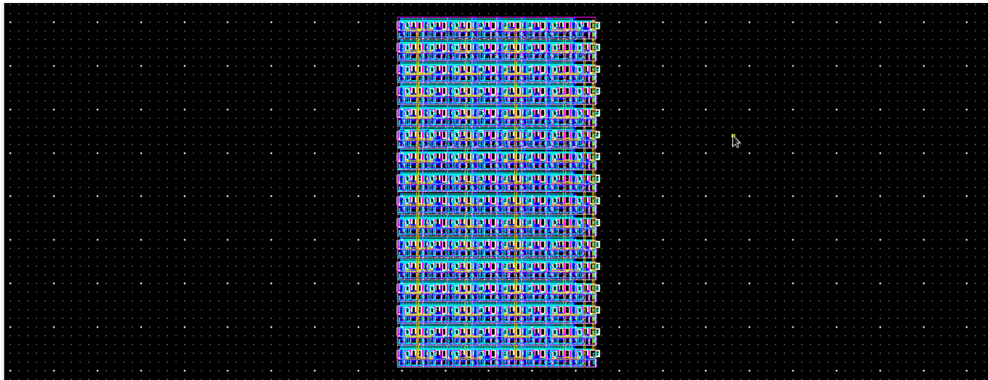


Figure 4. 6: DCV2 array layout

4.1.3 Ring Oscillator

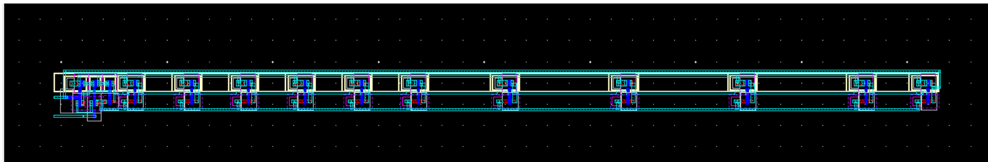


Figure 4. 7: Ring Oscillator layout

4.1.4 Complete DCO

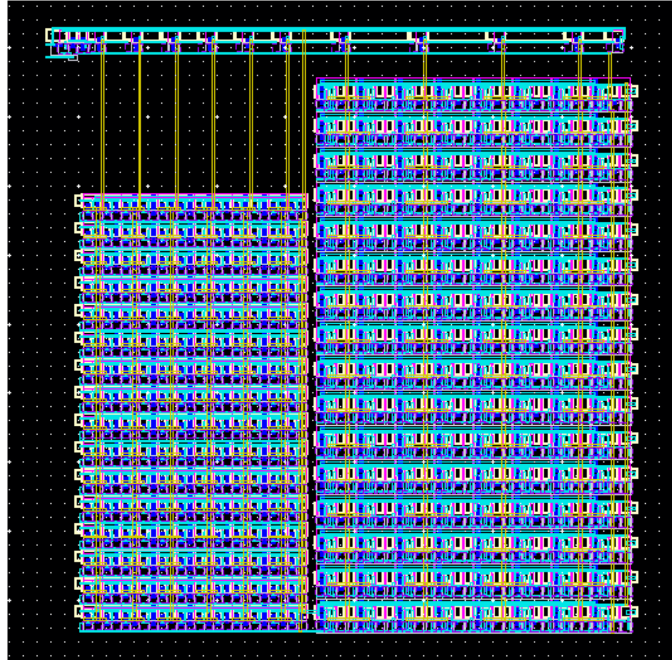


Figure 4. 8: Final DCO layout

The DCO range was from 3.33 ns to 10 ns of period (i.e. from 100 MHz to 300 MHz), but after constructing the layout directly without any modification to the sizing of DCVs or removing any fixed delay cells , the DCO range was from 11 ns to 16 ns (i.e. from 62.5 MHz to 90 MHz) which is out of our required range, so we had to remove the fixed delay cells (i.e. HDCs) but the range was still not satisfied, so we started to modify the length of the transistors in the DCV2 cells because the problem was with the lower bound of the required range (i.e. 100 MHz) and also increase the supply voltage source from 1.2v to 1.5v to satisfy the required range, and finally we have got 3.236 ns to 9.952 ns (i.e. 100.48 MHz to 309.02 MHz) as shown in figure 4.9.

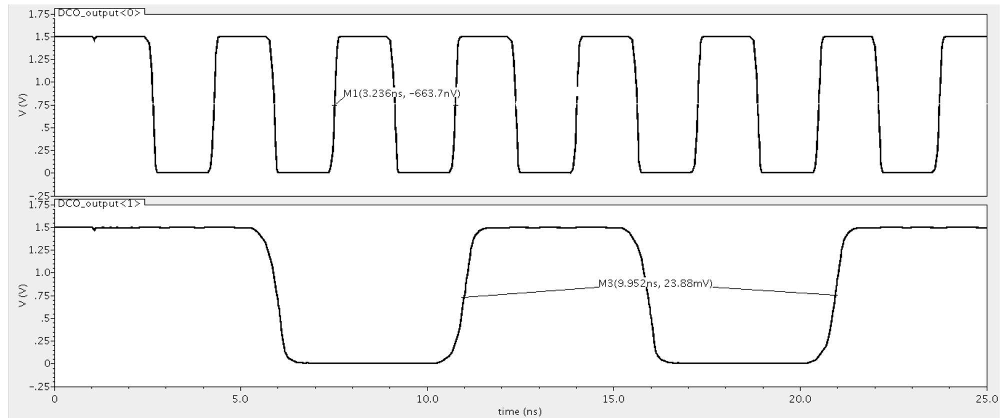


Figure 4. 9: DCO operating range in layout.

4.2 PFD

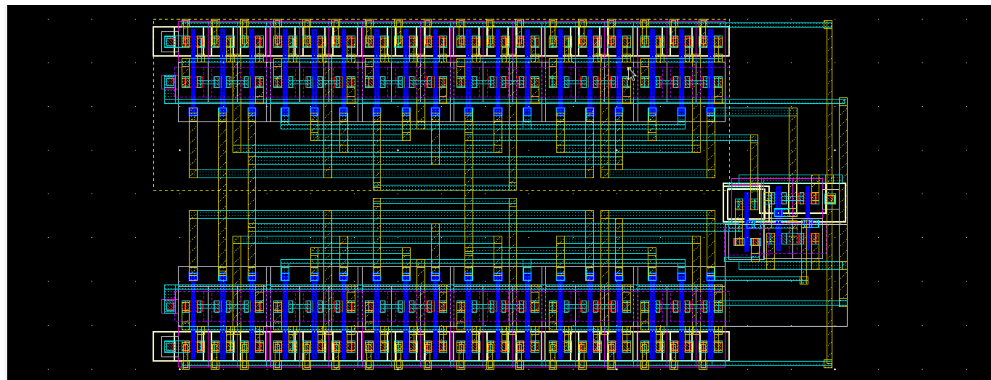


Figure 4. 10: PFD layout

The proposed PFD consists of two D flip flop and OR gate. At the first time , when simulating the layout directly with power supply voltage 1.2v , we found that the functionality of the PFD is not working correctly, so we decreased the supply voltage to 1v and the PFD worked properly as shown in figure 4.10 and figure 4.11.

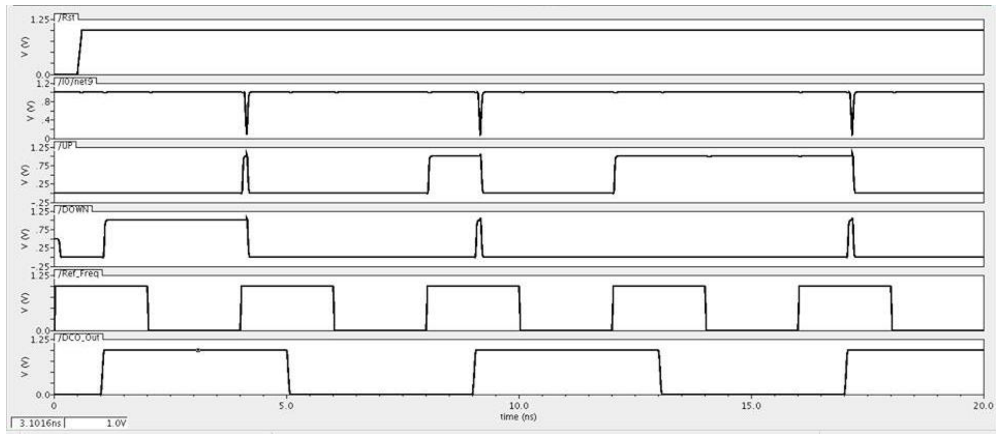


Figure 4.11: PFD pre-layout simulation.

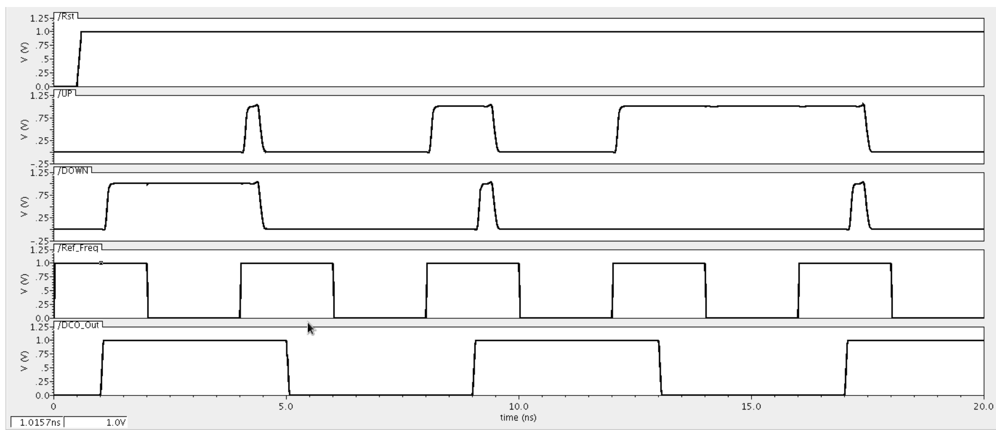


Figure 4.12: PFD post-layout simulation.

4.3 Loop Filter

As mentioned earlier , the proposed Loop Filter in ADPLL is a shift register, which contains D FF's and some combinational blocks.

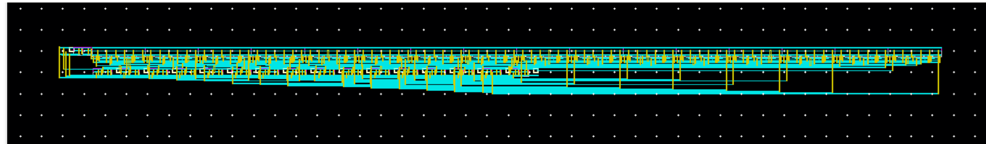


Figure 3.13: Shift Register layout

The post-layout simulation result was different from the pre-layout simulation due to the delay produced by the layout capacitance and resistance as shown in figure 4.12 and figure 4.13, but that is not a big deal. As we can see in figure 4.12 in the time period 2.5ns to 5ns the shift occurs at the positive edge of the CLK signal because the up and down signal are different , but in the same time slot in figure 4.13 the shift occurred once due to the delay produced by the layout.

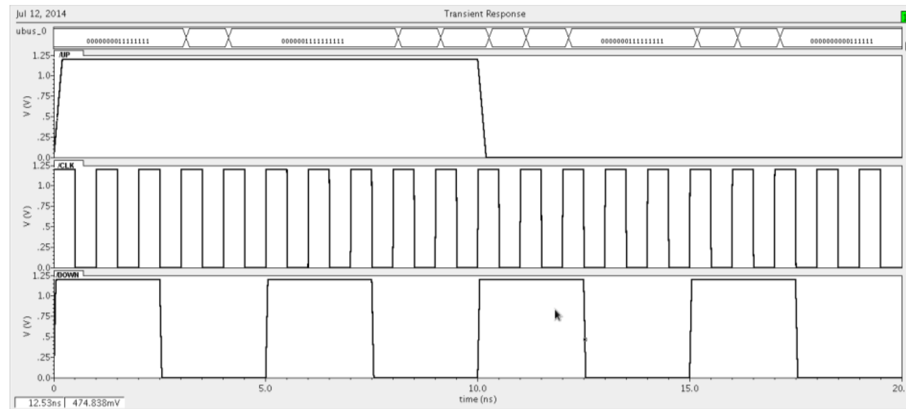


Figure 4.14: Pre-layout simulation of the Shift Register

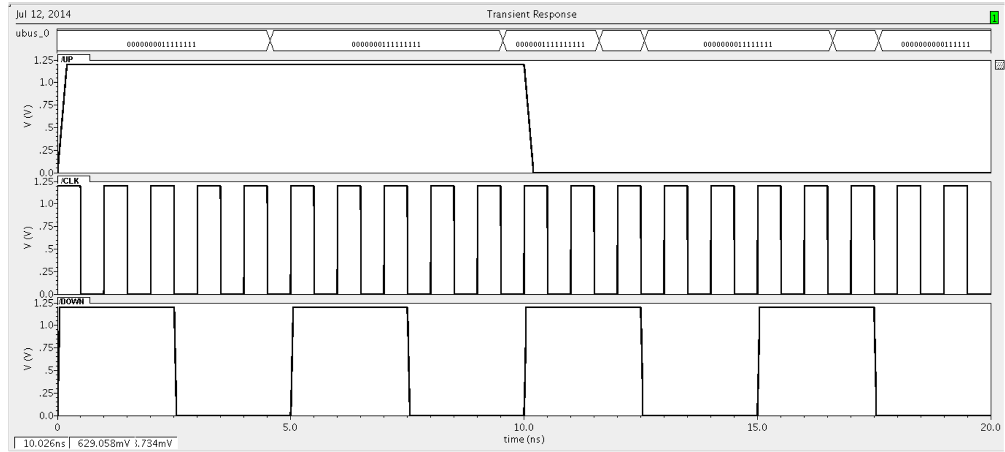


Figure 4.15: Post-layout simulation of the Shift Register

4.4 Overall Design

After constructing the layout of each block in the system, the layout of the overall design can be constructed as shown in figure 4.16

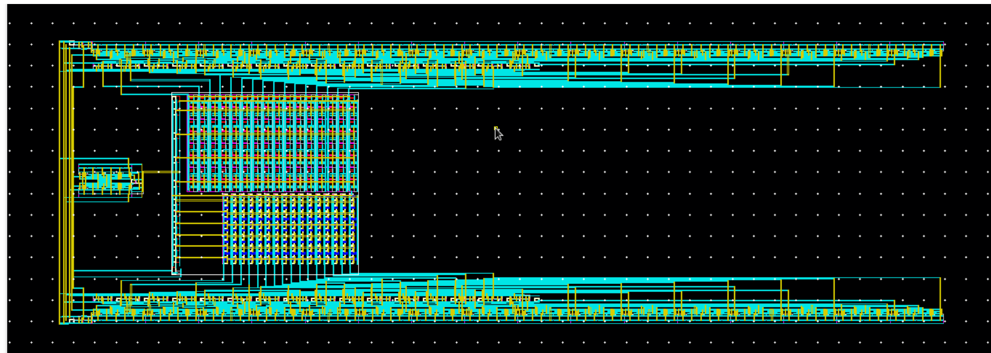


Figure 4.16: The overall layout

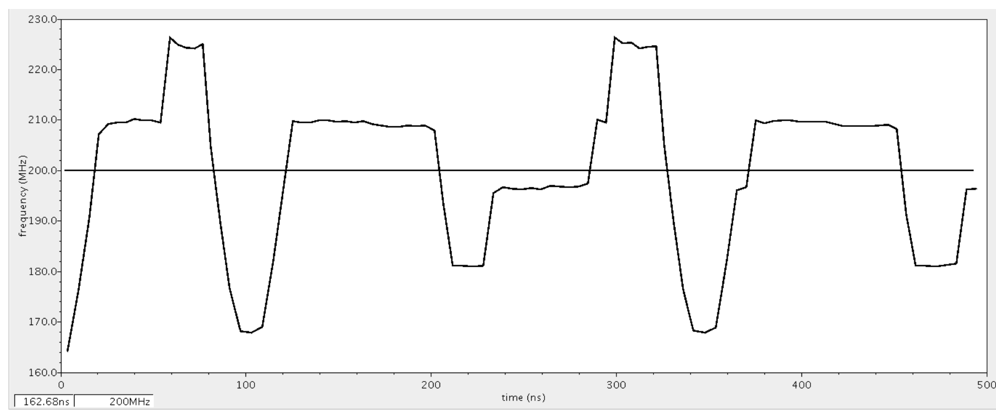


Figure 4.17: Overall post layout simulation

4.5 Conclusion

A comparison between the Required Specifications and Achieved Specifications is held as shown below

Required Specifications	***Achieved Specifications***
Power < 1 mW	0.6 mW
Area < 0.01 mm^2	From Layout= 0.0086mm^2
Lock time < 10 μs	1 μs
P-to-P jitter < 20 ps	35 ps

References

[1] Roland E. Best, “Phase Locked Loops, Design, simulation and applications 5th Edition” New York McGraw-Hill, 2003.

[2] Duo Sheng, Ching-Che Chung and Chen-Yi Lee, "An Ultra-Low-Power and Portable Digitally Controlled Oscillator for SoC Applications" IEEE TRANSACTIONS ON CIRCUITS AND SYSTEMS—II: EXPRESS BRIEFS, VOL. 54, NO. 11, NOVEMBER 2007.

[3] SALEH R. AL-ARAJI, ZAHIR M. HUSSAIN and MAHMOUD A. AL-QUTAYRI ,"DIGITAL PHASE LOCK LOOPS, Architectures and Applications" Springer,2006.

[4] Pao-Lung Chen, Ching-Che Chung, and Chen-Yi Lee "A Portable Digitally Controlled Oscillator Using Novel Varactors " IEEE TRANSACTIONS ON CIRCUITS AND SYSTEMS—II: EXPRESS BRIEFS, VOL. 52, NO. 5, MAY 2005.

[5] João Baptista Martins, Ricardo Reis and José Monteiro, "Capacitance and Power Modeling at Logic-Level".

[6] CADENCE, “Virtuoso AMS Designer Environment Tutorials”, 2008.

[7] CADENCE, “Virtuoso AMS Designer Simulator User Guide”, 2006.

[8] CADENCE, “Virtuoso AMS Environment User Guide”, 2006.

[9] Alain Vachoux, “Top-Down Digital Design Flow” Version 6.0, October 2011.

[10] Ahmed Ahmed, Hussein Mohamed, Khaled Ebrahim, Khaled Mohamed and Mohamed Sherif "All Digital Phase Locked Loop (ADPLL)", July 2013

Appendix A

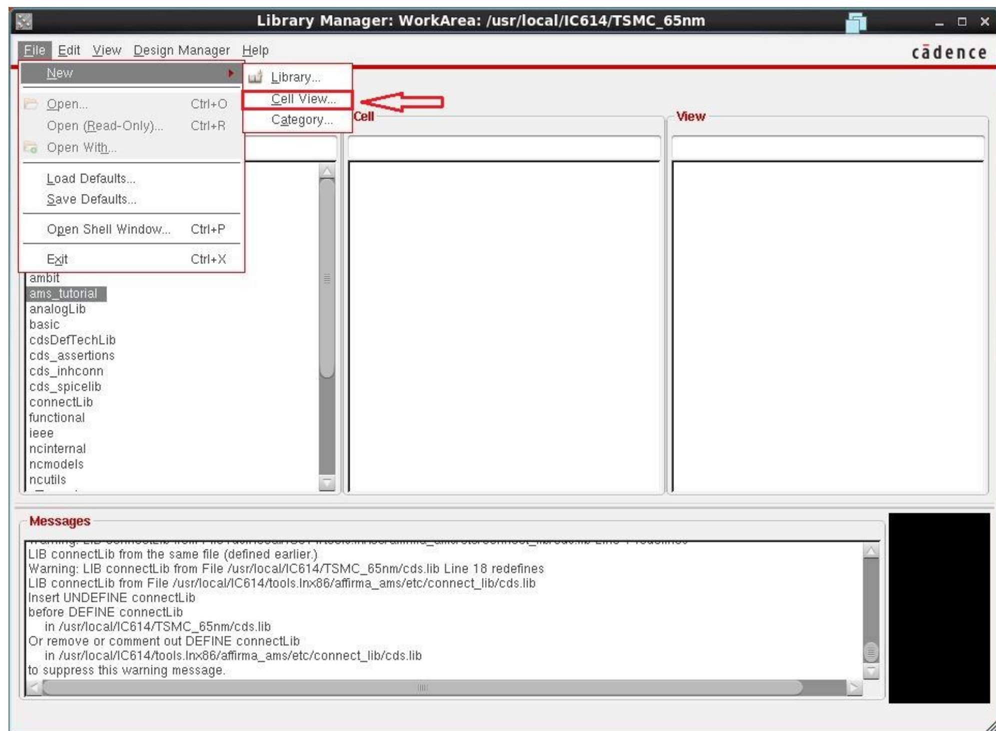
AMS TUTORIAL

Through this tutorial you will learn how to simulate a system containing digital Verilog blocks, digital VHDL blocks and analog blocks. This tutorial is mainly divided into two parts, part 1 and part 2. For part 1, we are going to simulate a digital Verilog inverter with an analog inverter and compare the outputs of them, and then we put them in a cascaded configuration (analog inverter after digital Verilog inverter) to work together as a buffer. The idea of this cascaded configuration is to make sure that, the connect rules between the digital block and analog block are established correctly. For part 2, we are going to simulate another design contains three main blocks, a digital Verilog 4-bit counter, an analog 4-bit inverter and a digital VHDL 4-bit inverter. This design is organized as follows; an external clock signal and reset signal are applied to the Verilog 4-bit counter. The output of the Verilog 4-bit counter is labeled as count_out<3:0> and is applied as an input to the next block which is the analog 4-bit inverter. The output of the analog 4-bit inverter is labeled as vhdl_inv<3:0> and is applied as an input to the last block which is VHDL 4-bit inverter. The output of the VHDL 4-bit inverter is labeled as vhdl_out<3:0>. We are interested in these signals, count_out<3:0>, vhdl_inv<3:0> and vhdl_out<3:0>. If everything is correct, the final output, vhdl_out<3:0> will be the same as the counter output count_out<3:0>. A block diagram for this design can be found in part 2 section.

Part1:

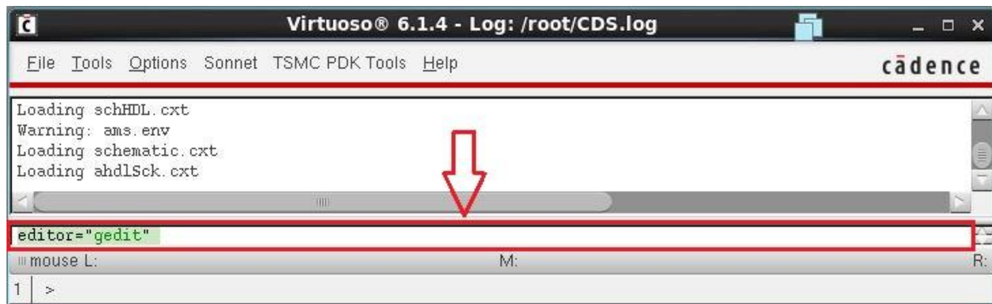
The following steps are to simulate both the digital Verilog inverter and the analog inverter and compare both outputs. Another configuration for these two inverters is to put them in cascade to work as a buffer. We use a library called ams_tutorial to include all circuits in this tutorial.

Firstly, we will make a cell view for the digital Verilog inverter.

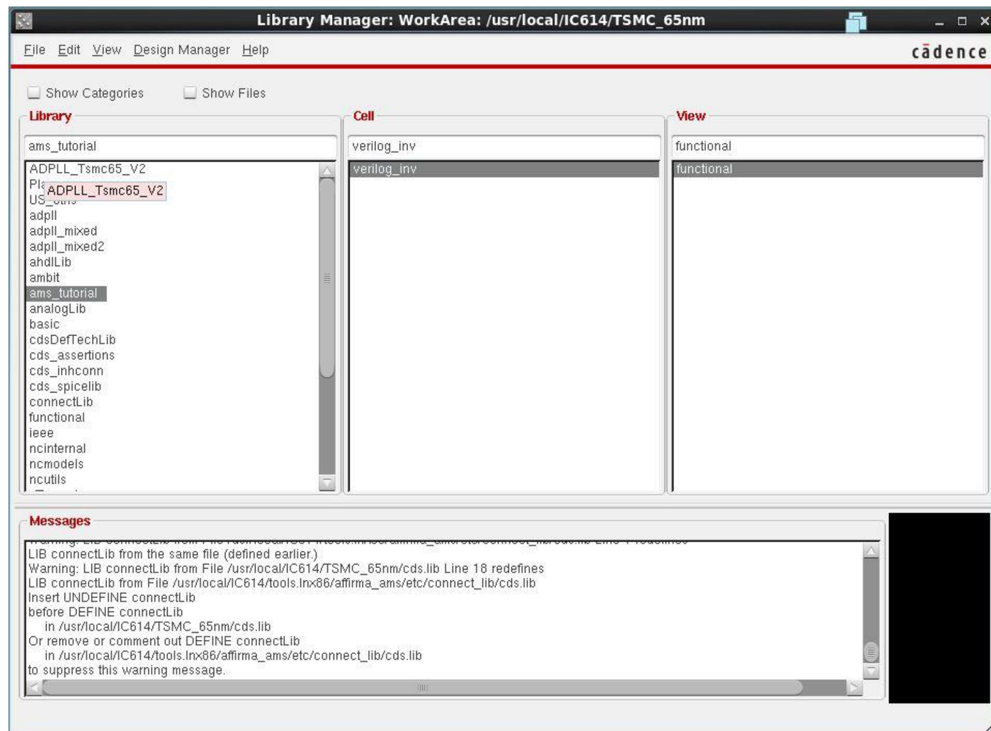


After pressing the OK button, you'll get this menu

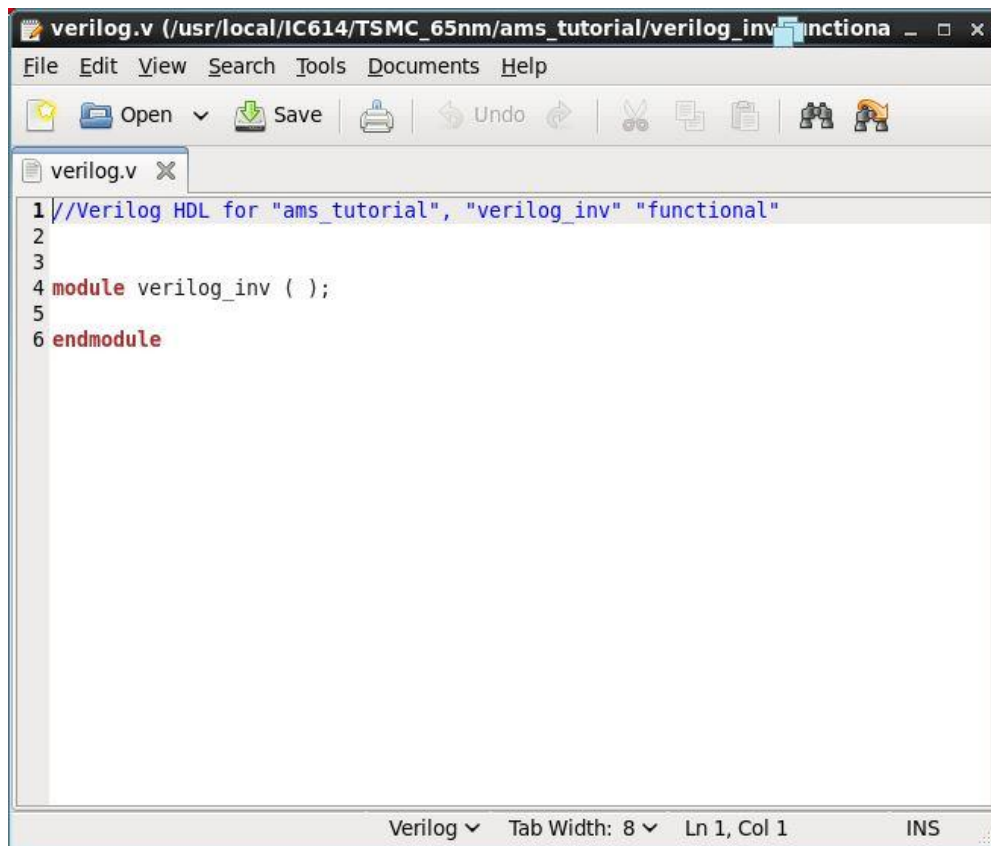
It is a little difficult to edit your code in this text editor so, we will use another text editor called "gedit" by typing the command **editor="gedit"** in the CIW window.



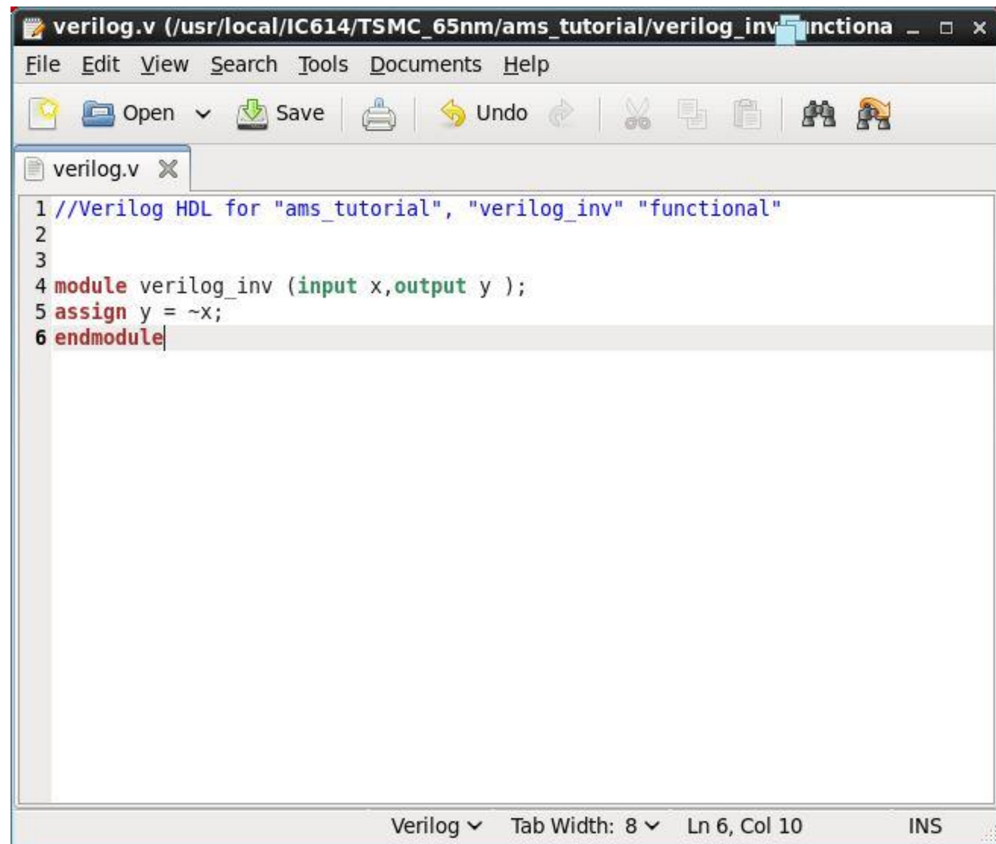
You can now see your cell view and double click on it to edit the code using gedit text editor



A new window will open up after double clicking your cell view as follows:

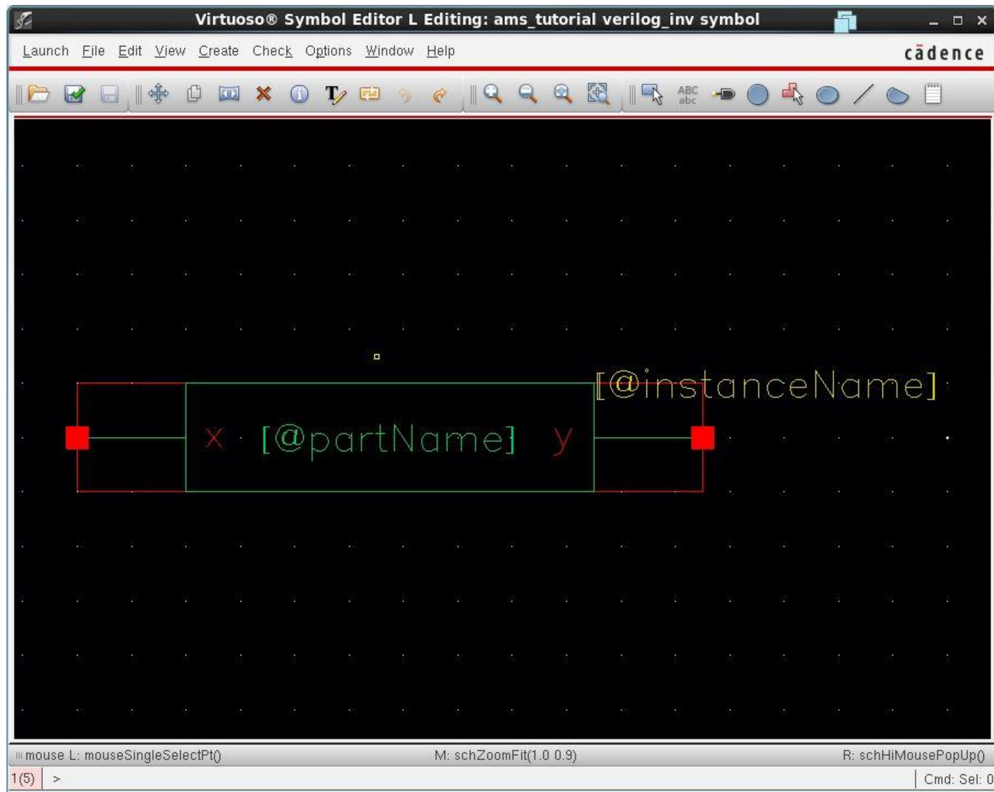


Edit the code and save it.

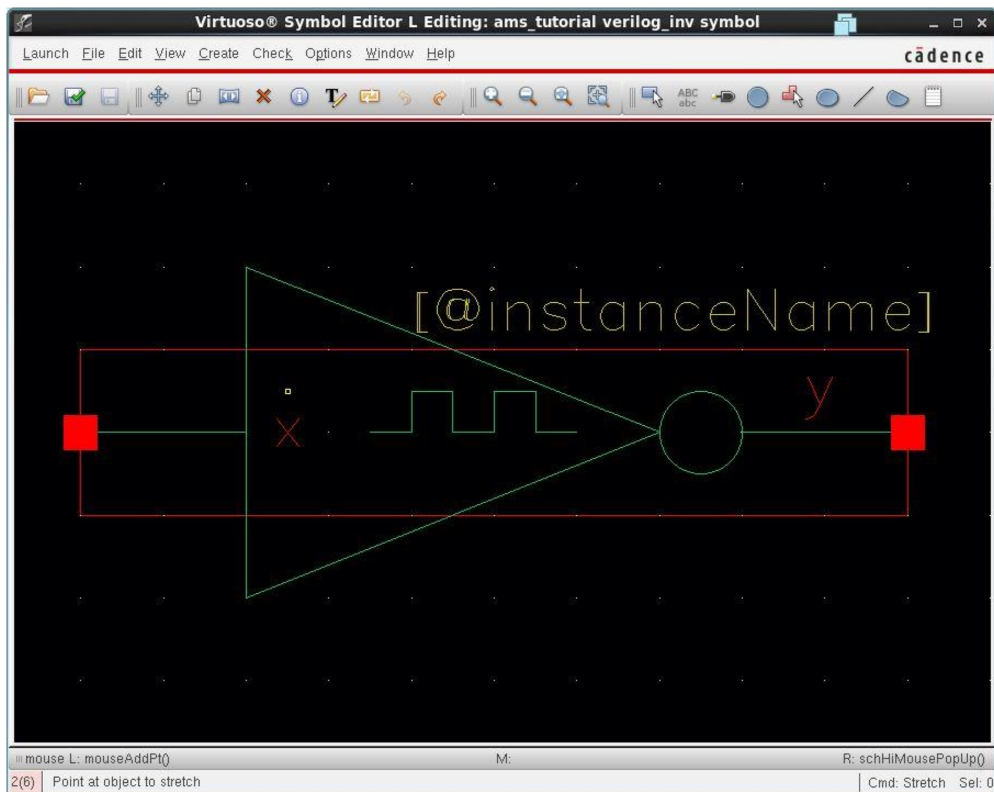


After closing it, you'll get a menu asking to create a symbol view for your circuit, press "yes".

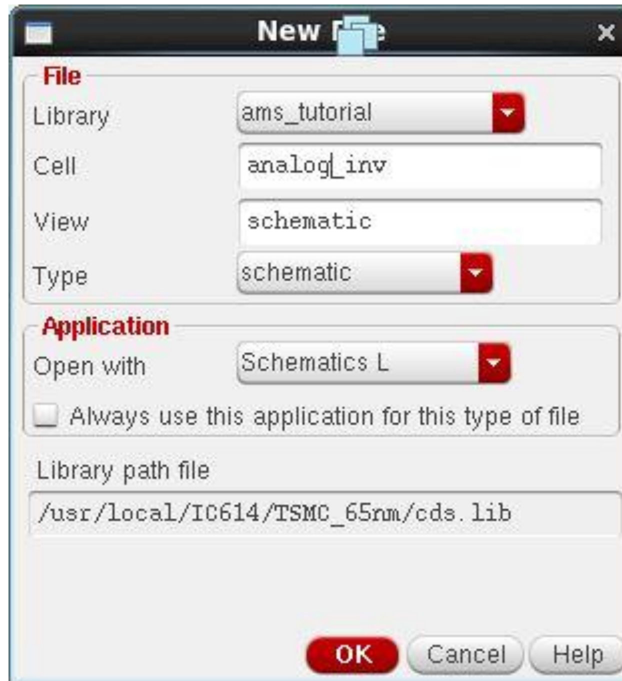
You can use the created symbol as is or edit it.

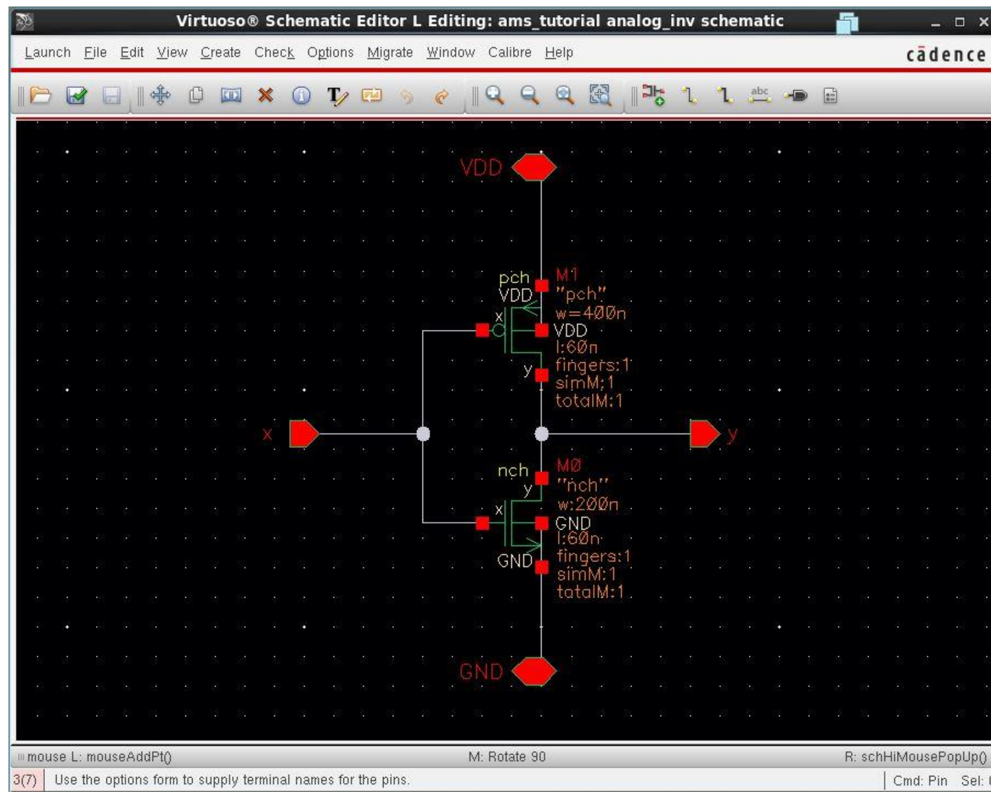


We will edit it to take the form of an inverter (optional step).

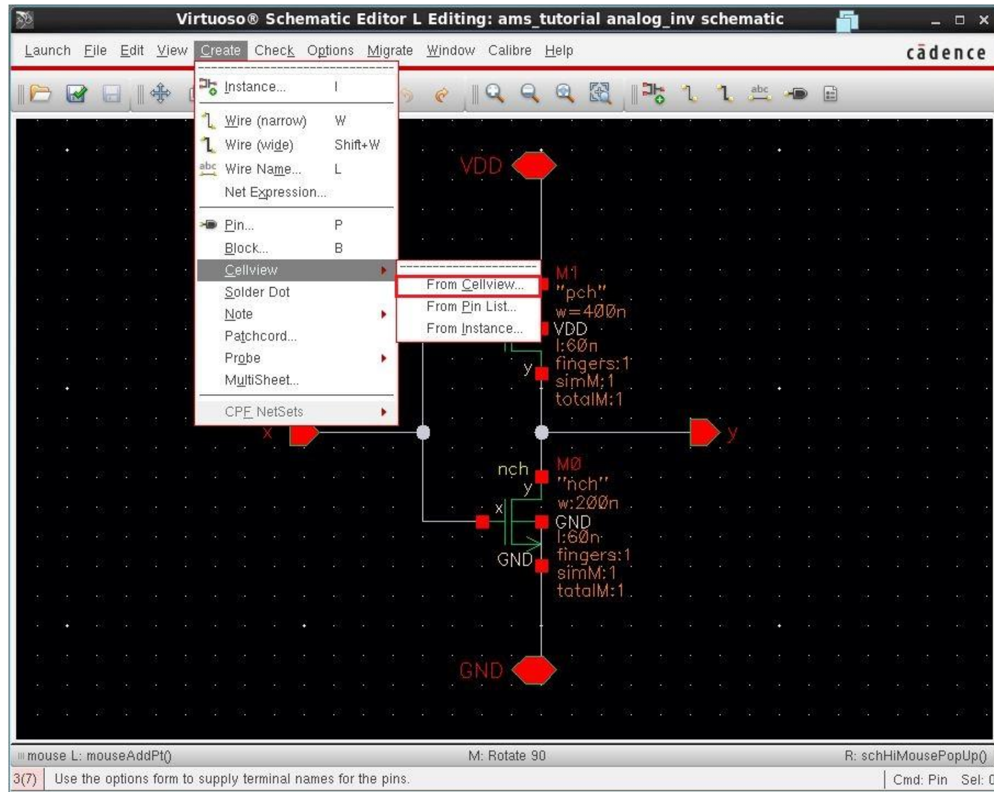


We put a something like square wave inside the symbol view to differentiate between it and the analog inverter. Now it is time to create the analog inverter.



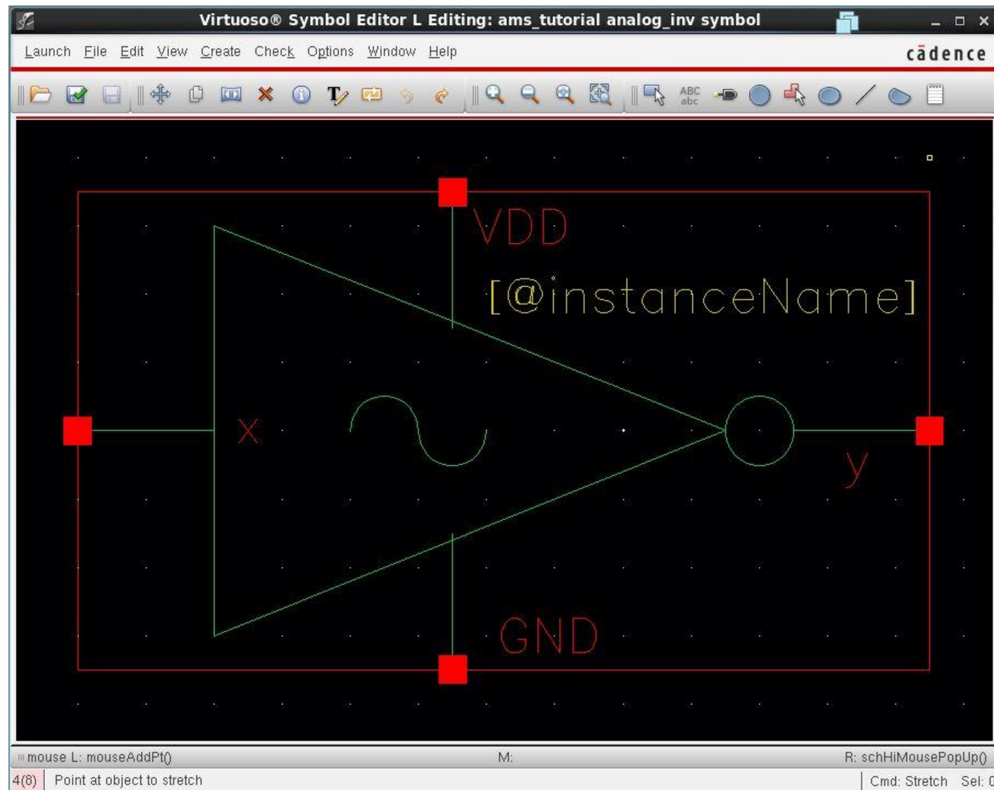


To create a symbol choose, **Create > cellview > from cellview** as follows

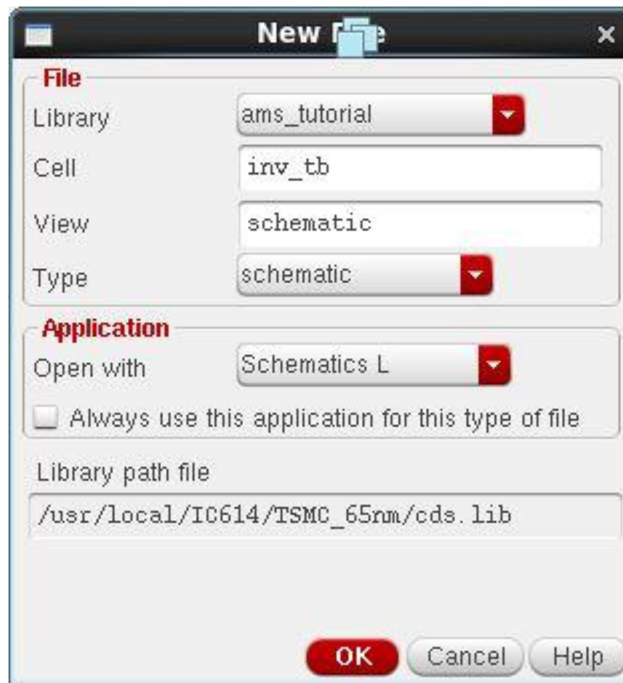


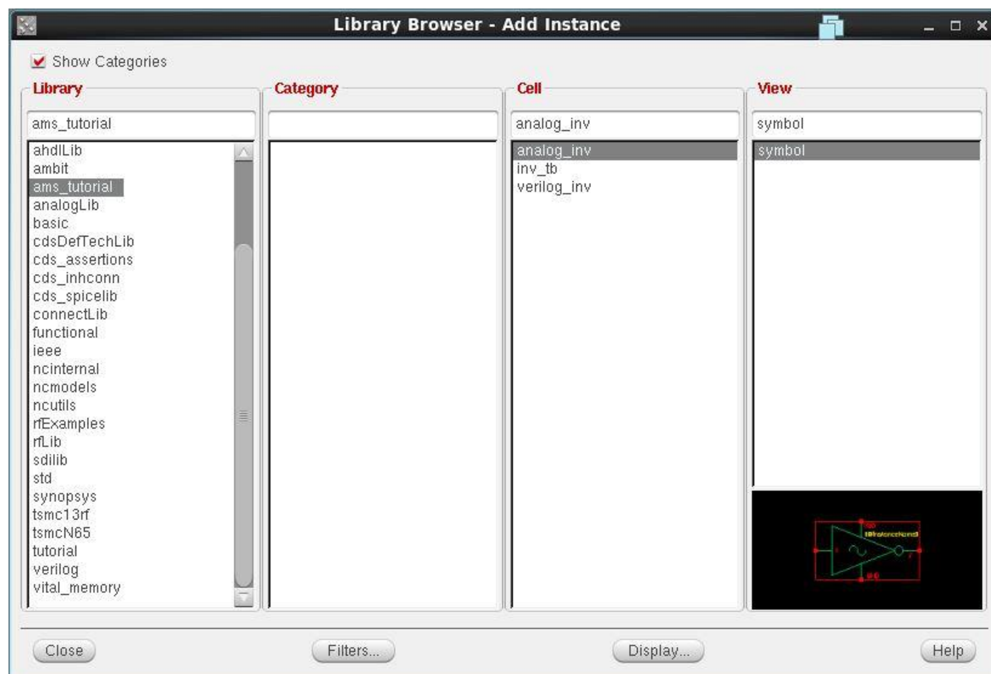
Press ok for the next window

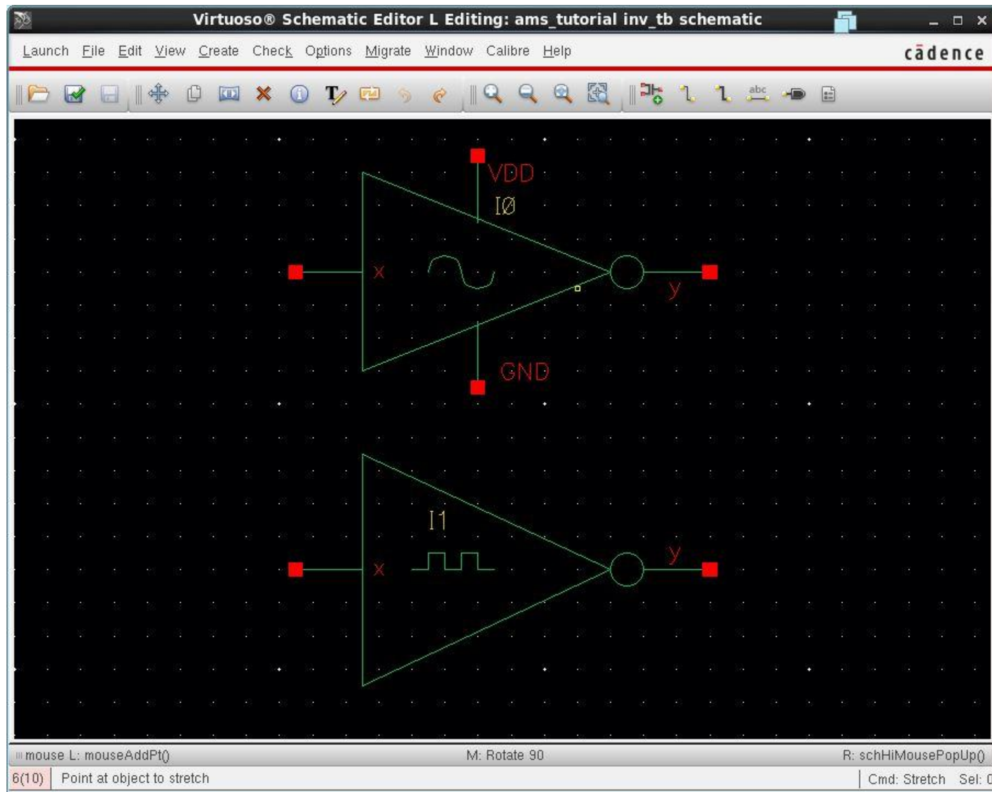
We will edit the symbol view to looks like the following (optional). We put a something like a sine wave inside the symbol to differentiate it from the digital inverter.



Next step is to make a test bench for a circuit containing both the digital Verilog inverter and the analog one.

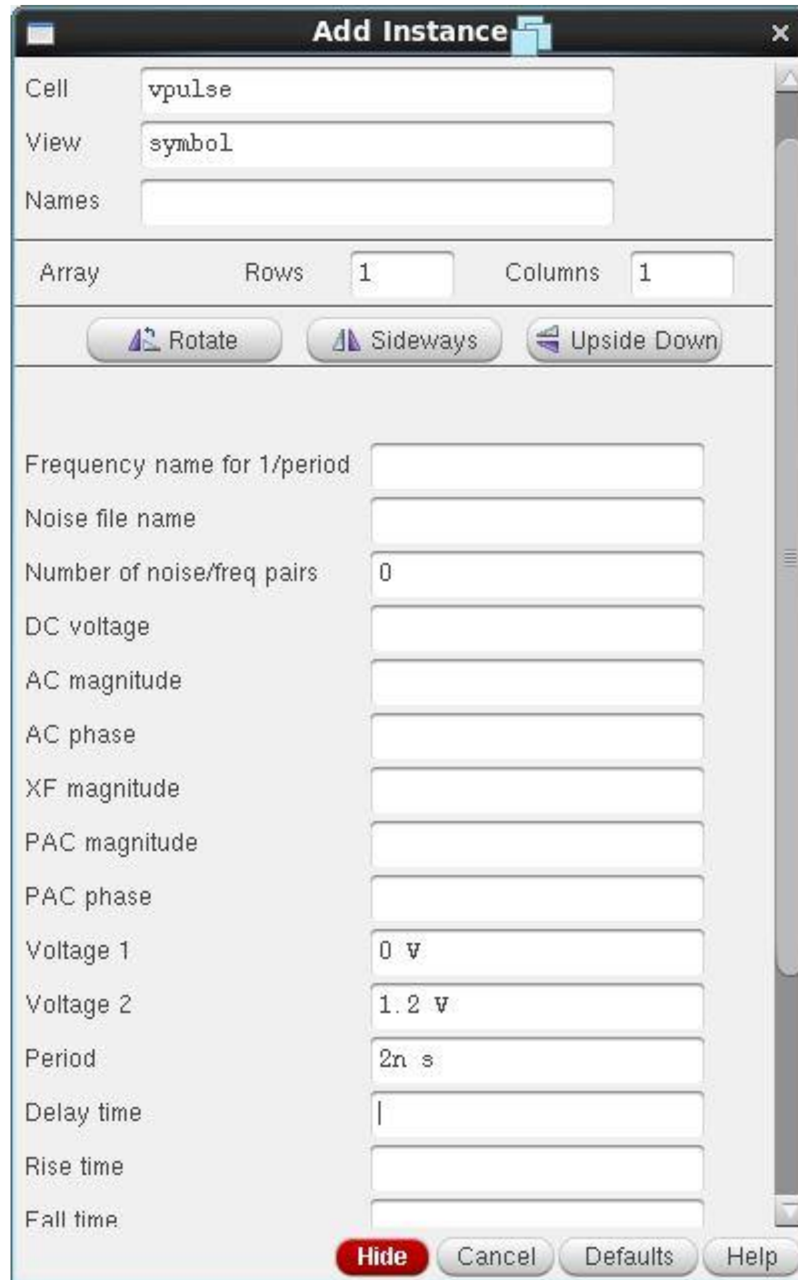




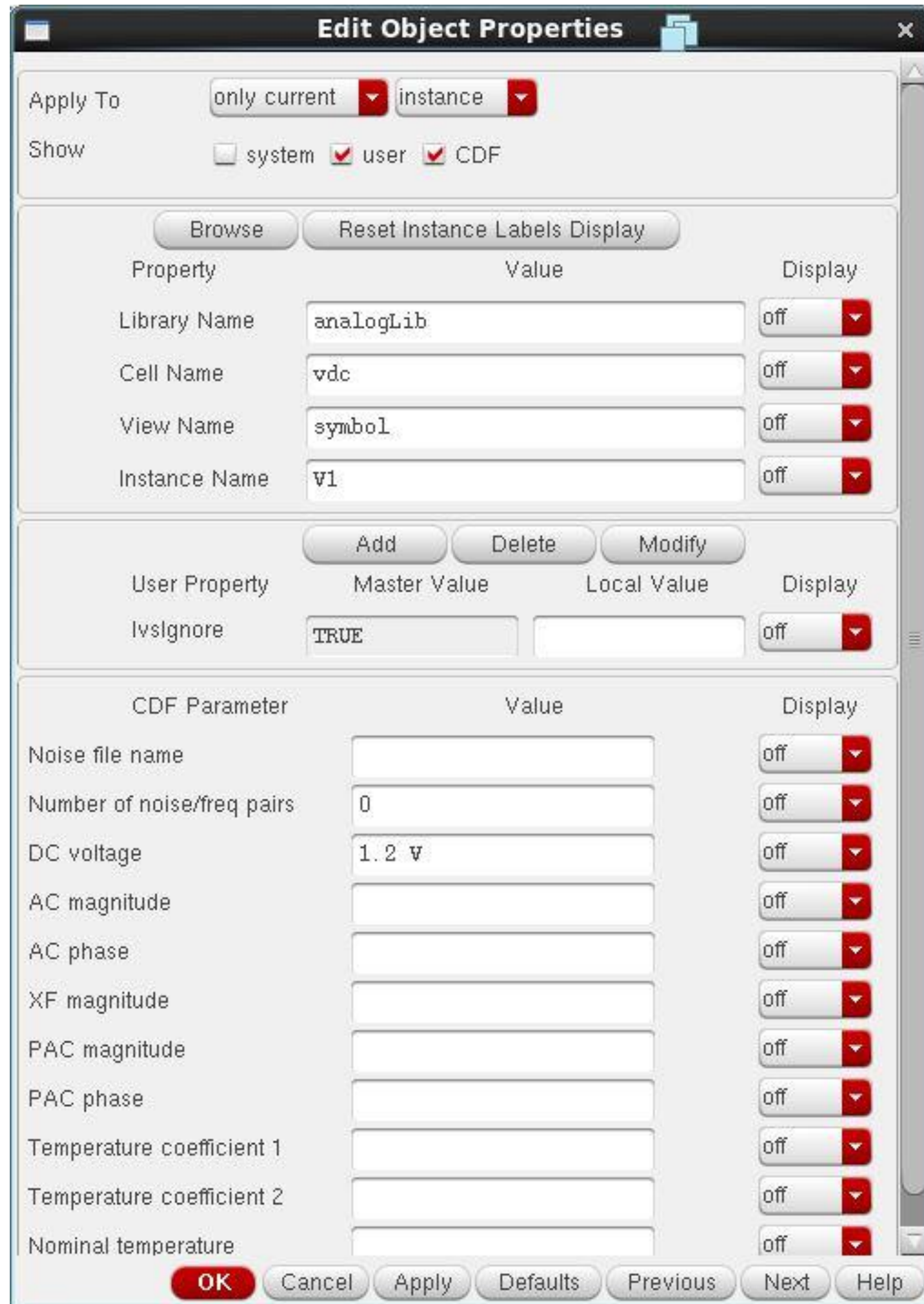


Put labels for input signal, Verilog output and analog output signals.

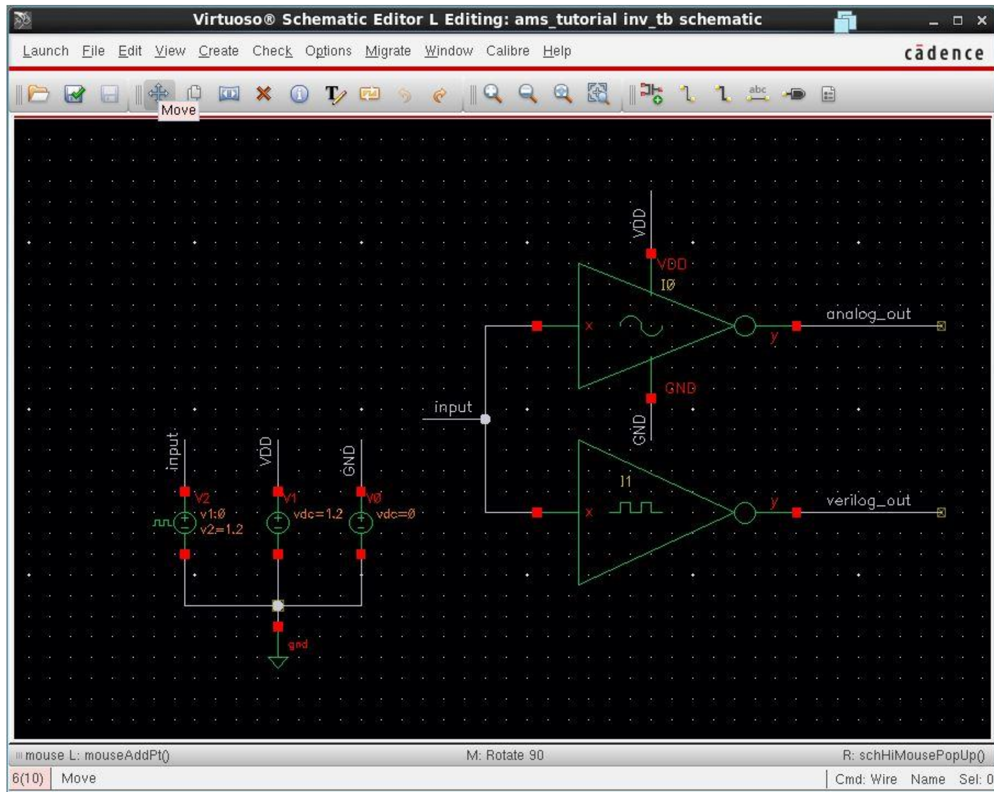
For the input source, a good choice is a periodic signal to test both, high state and low state. We use Vpulse as an input signal.



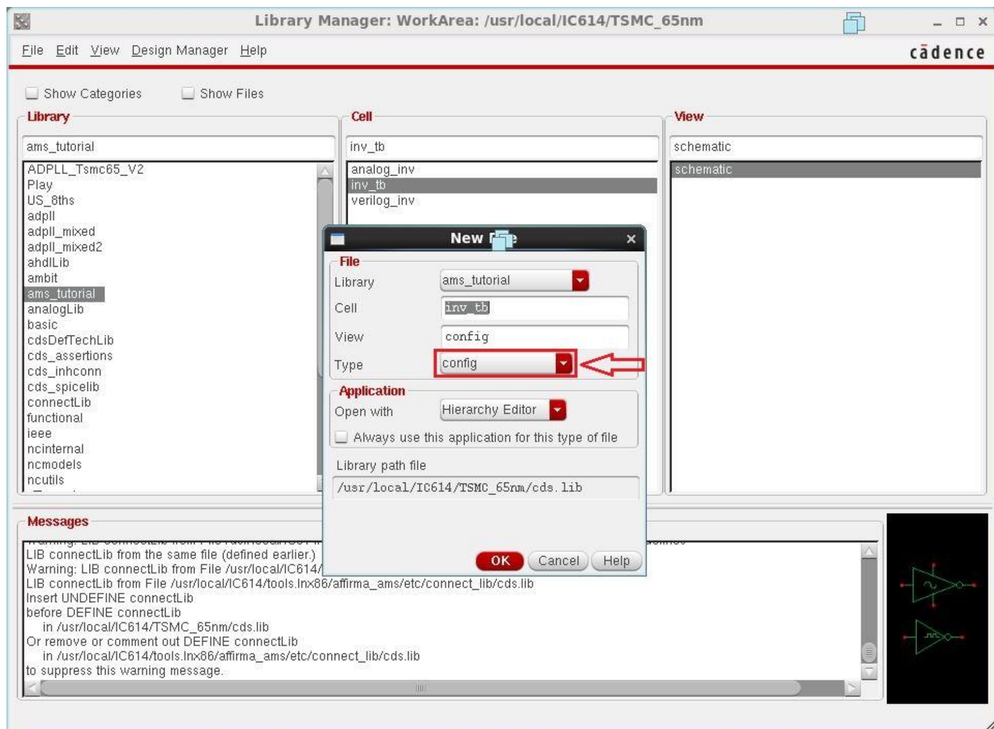
For this tutorial we use Vsupply = 1.2 v. We will use this value in connect rules also.



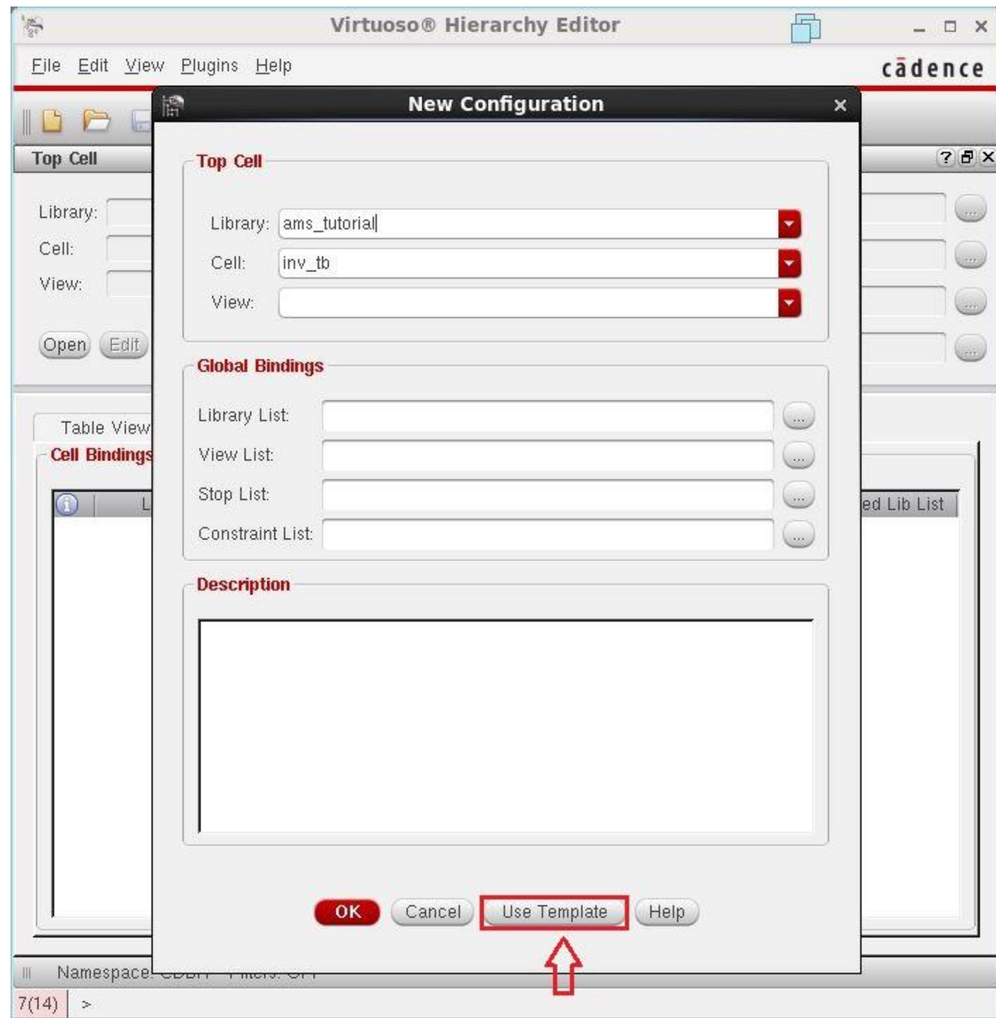
The final schematic after adding labels and sources looks like the following



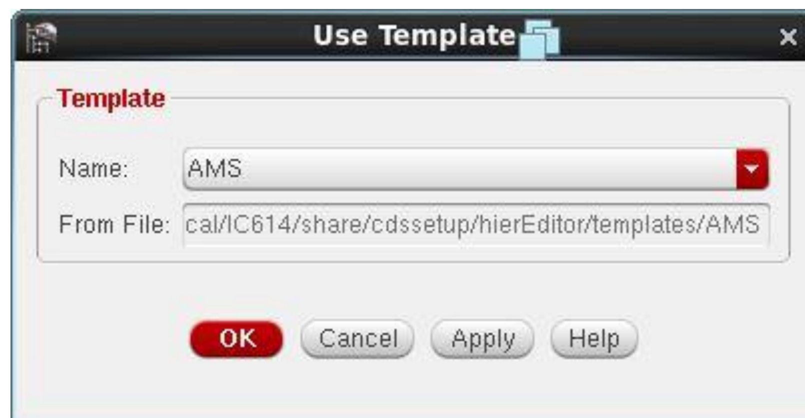
Close this schematic and create another cell view for it of type **config**. Make sure to choose type **config** (important).



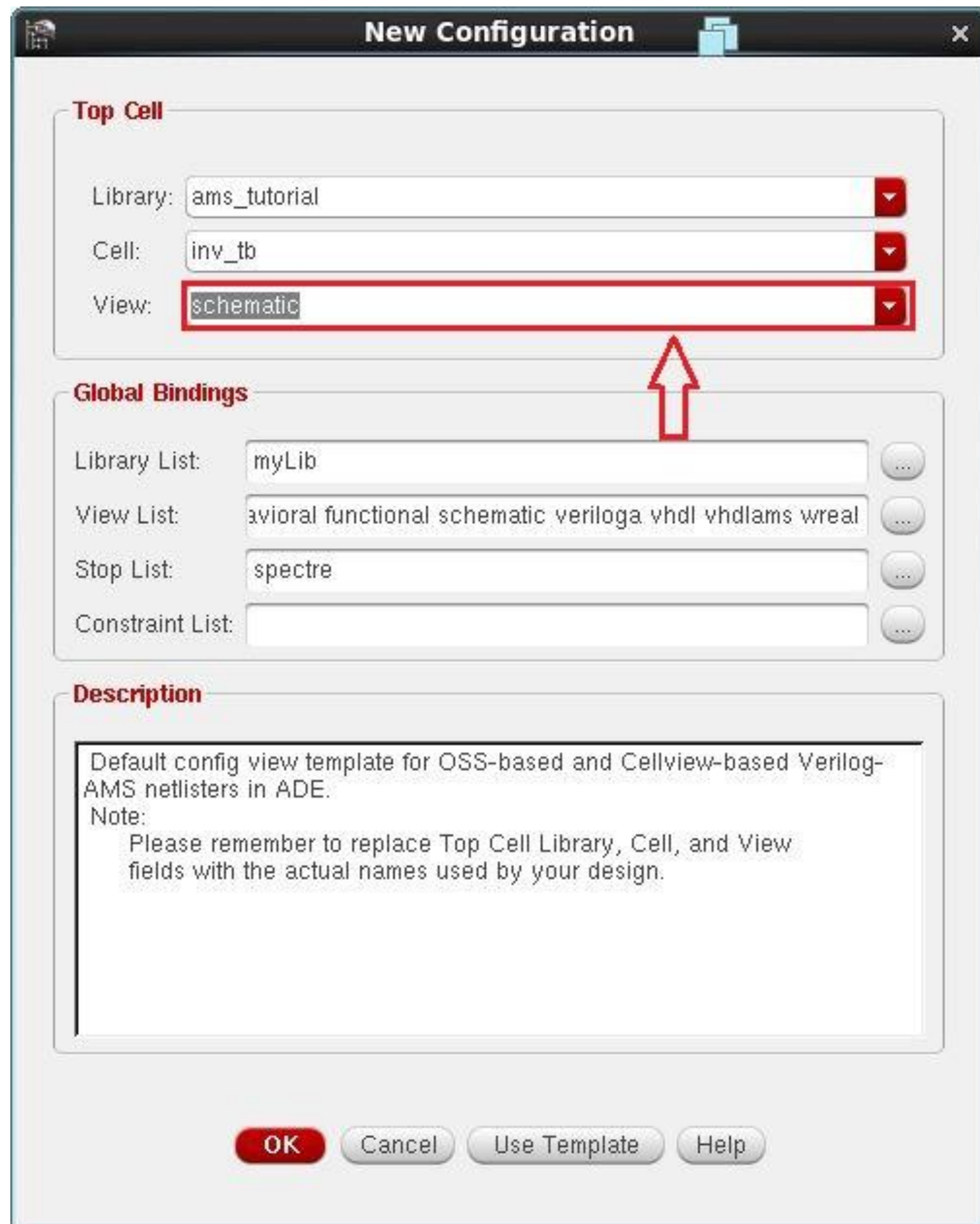
You will get a menu like the following



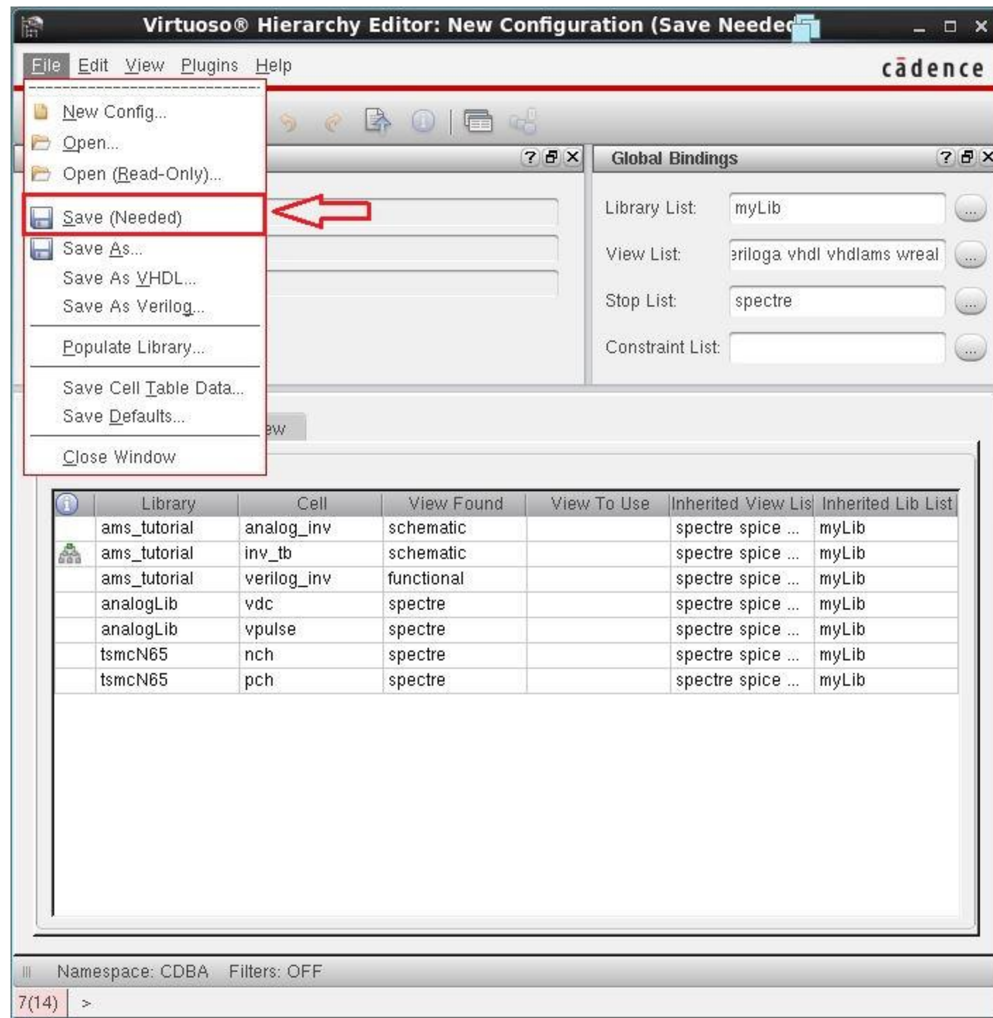
Press **Use Template** and choose **AMS** for **Name** field from the following menu.



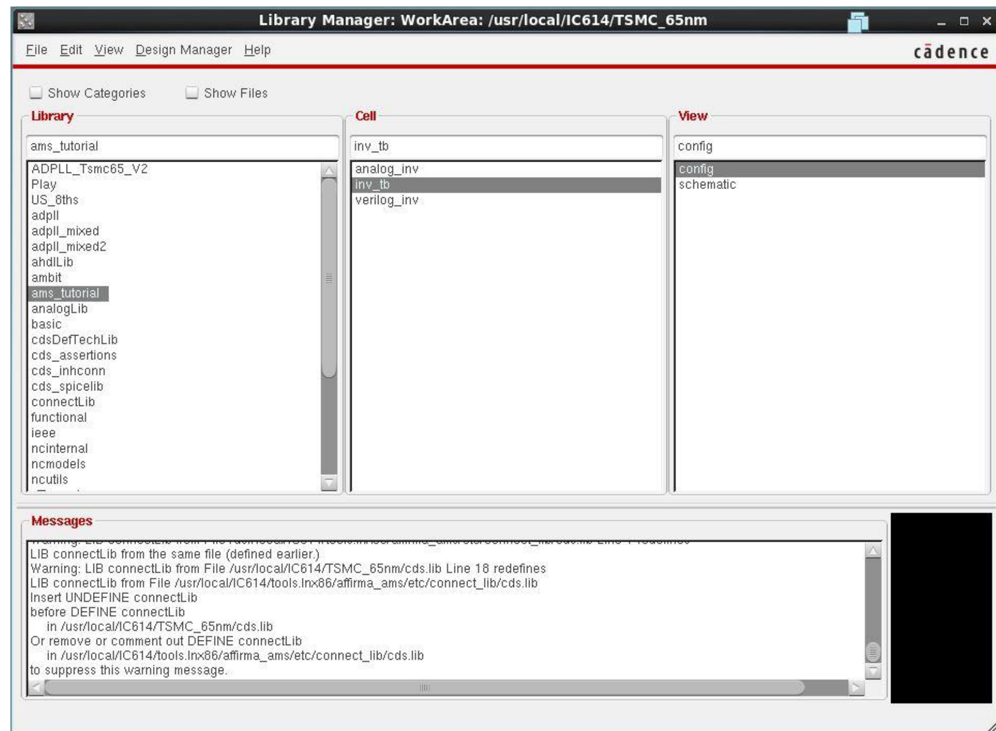
Press **OK** and choose **view** as **schematic** for the section **Top Cell** like the following.



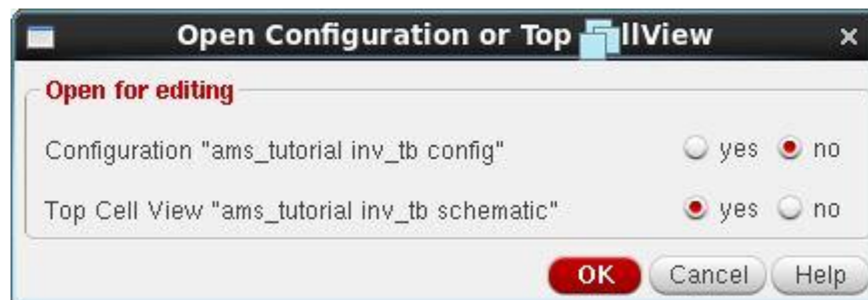
After pressing **OK** you will get the following menu. Choose **File>Save**



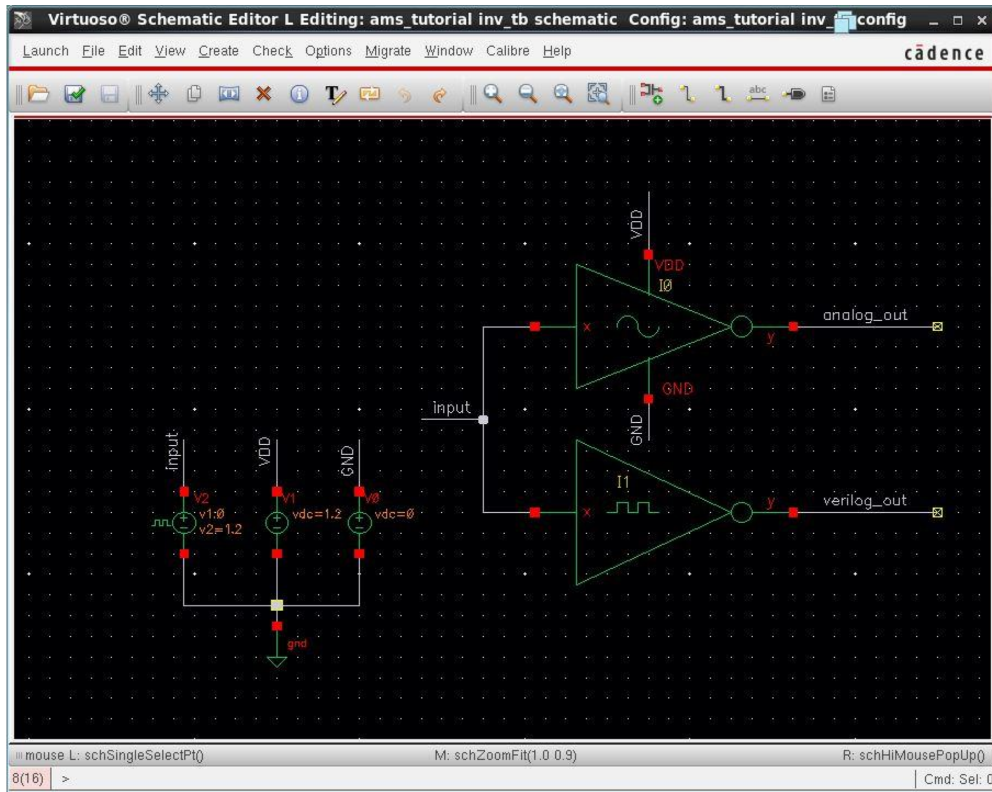
Notice that, **inv_tb** is now having two views, **config** and **schematic**.



Double click on **config** and press **OK** from the following menu

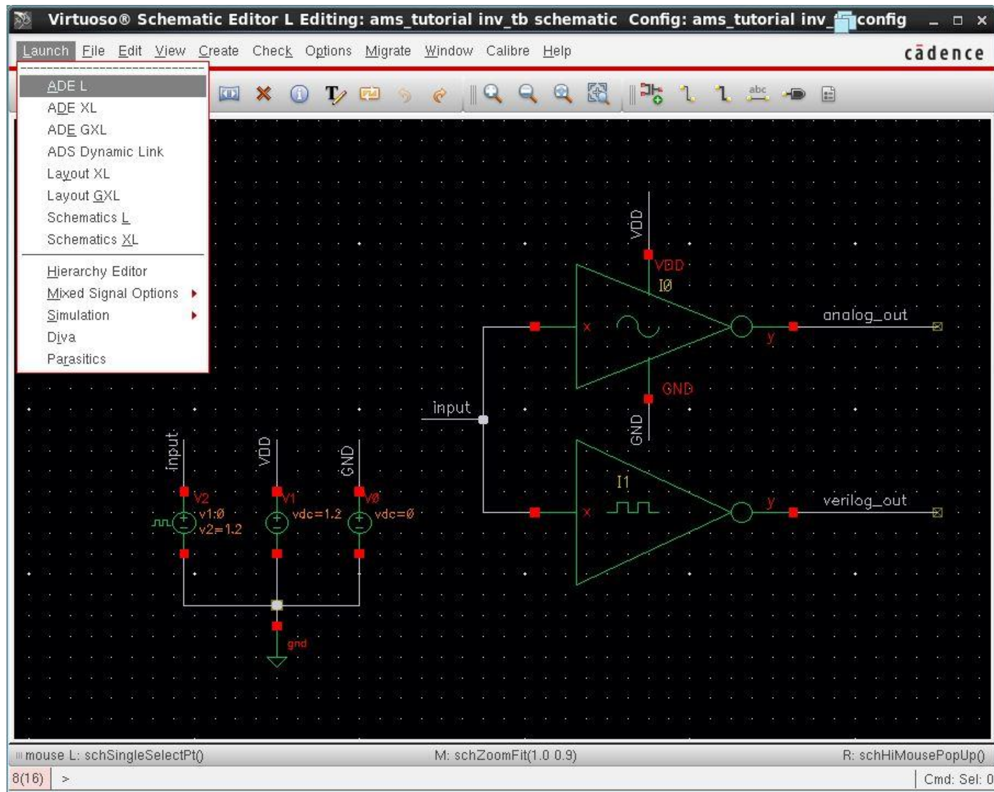


It will open the **config** view of your circuit. **Choose check and Save.**

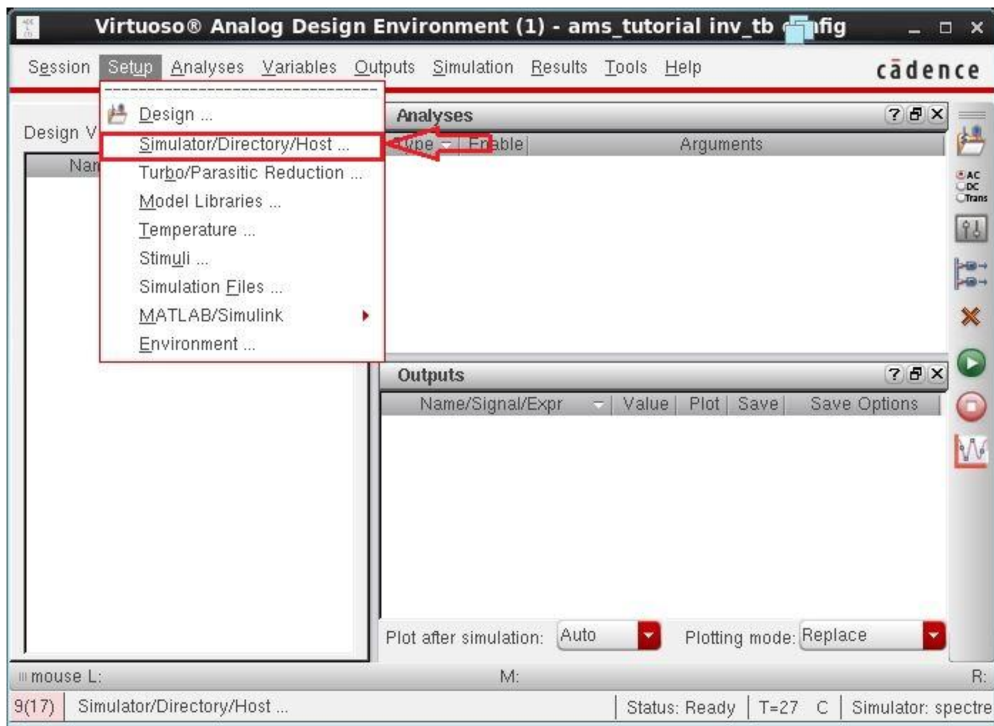


Then Launch > IDE L to start simulating your circuit

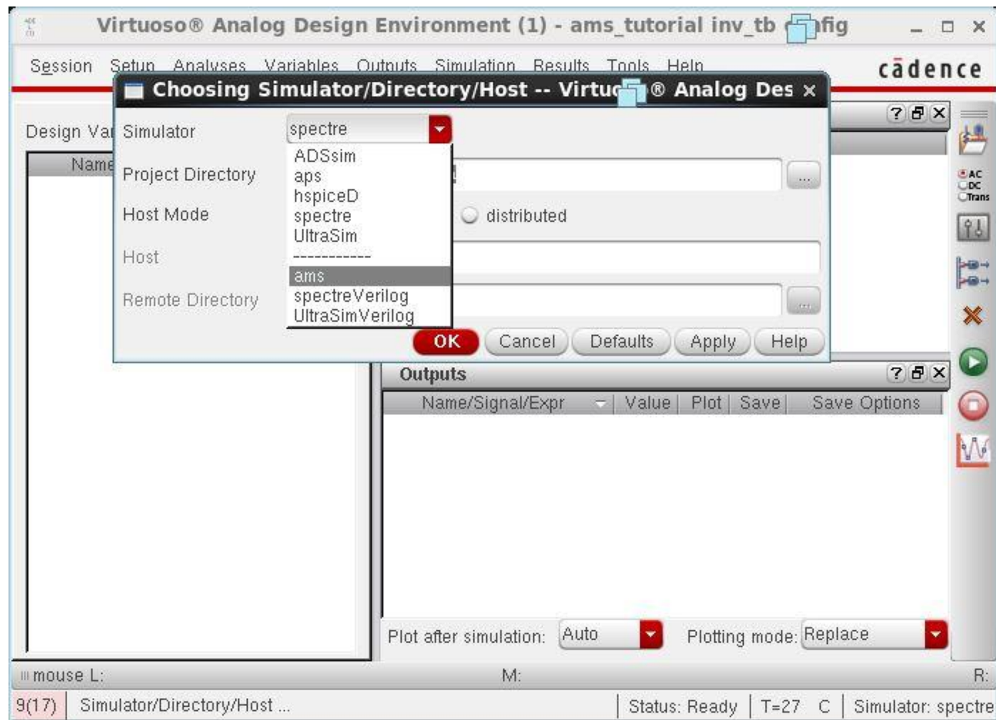
(This must be done from the **config** view not schematic view)



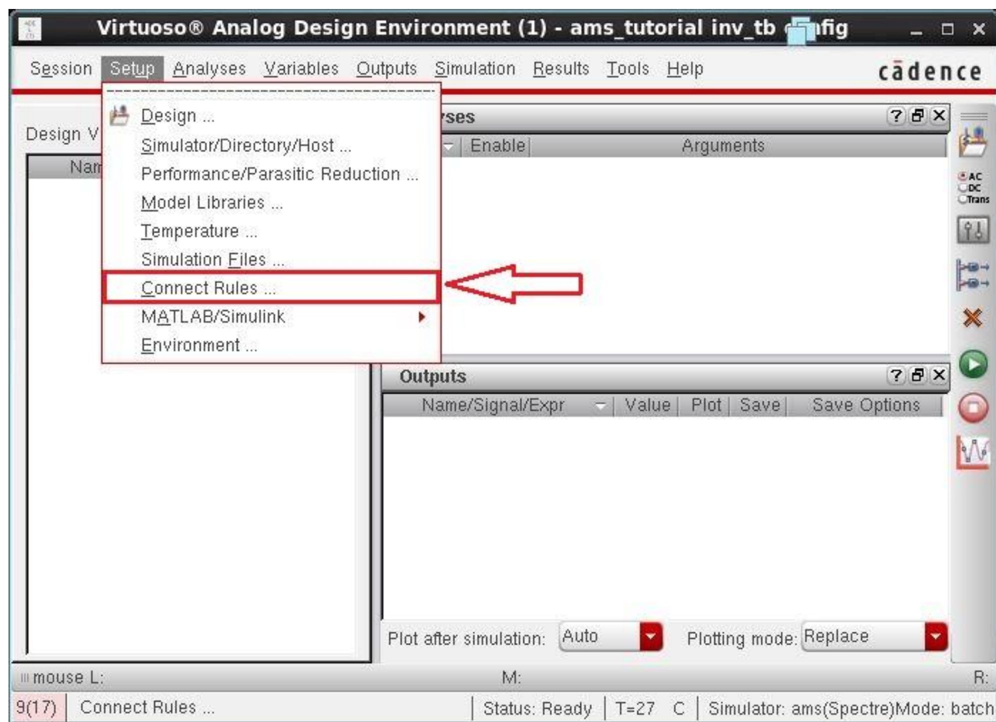
Then choose **Setup>Simulation/Directory/Host..** from the following menu



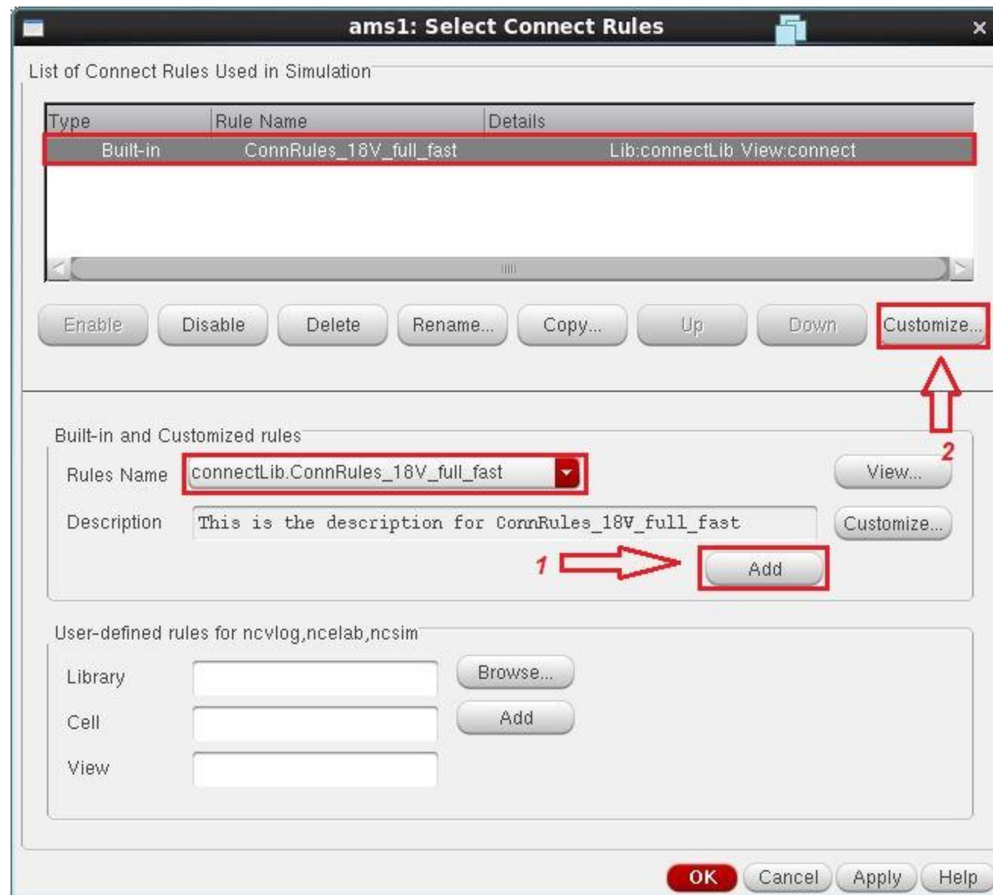
Choose simulator as **ams** and press **OK**



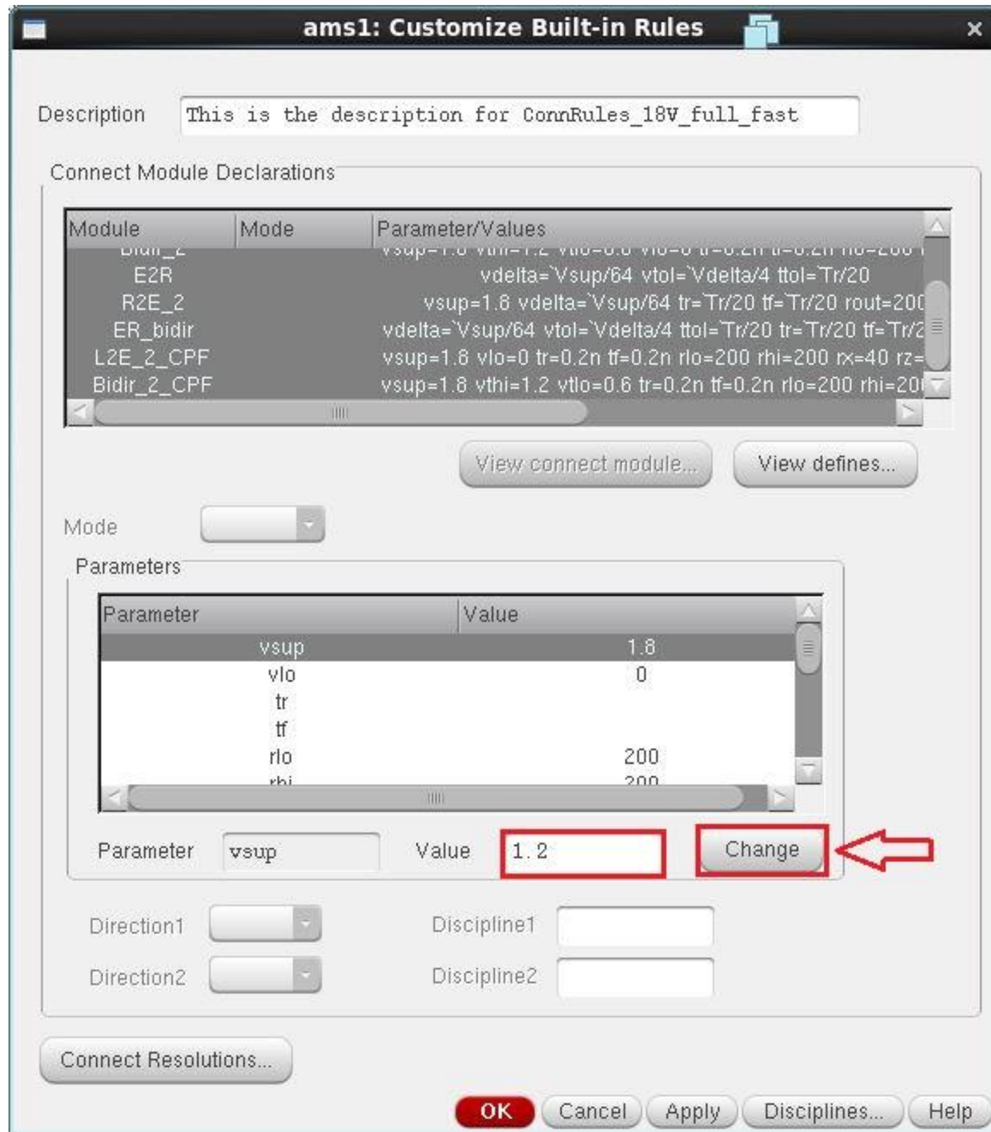
The next step is to edit the connection rules. Choose **Setup>Connect Rules**

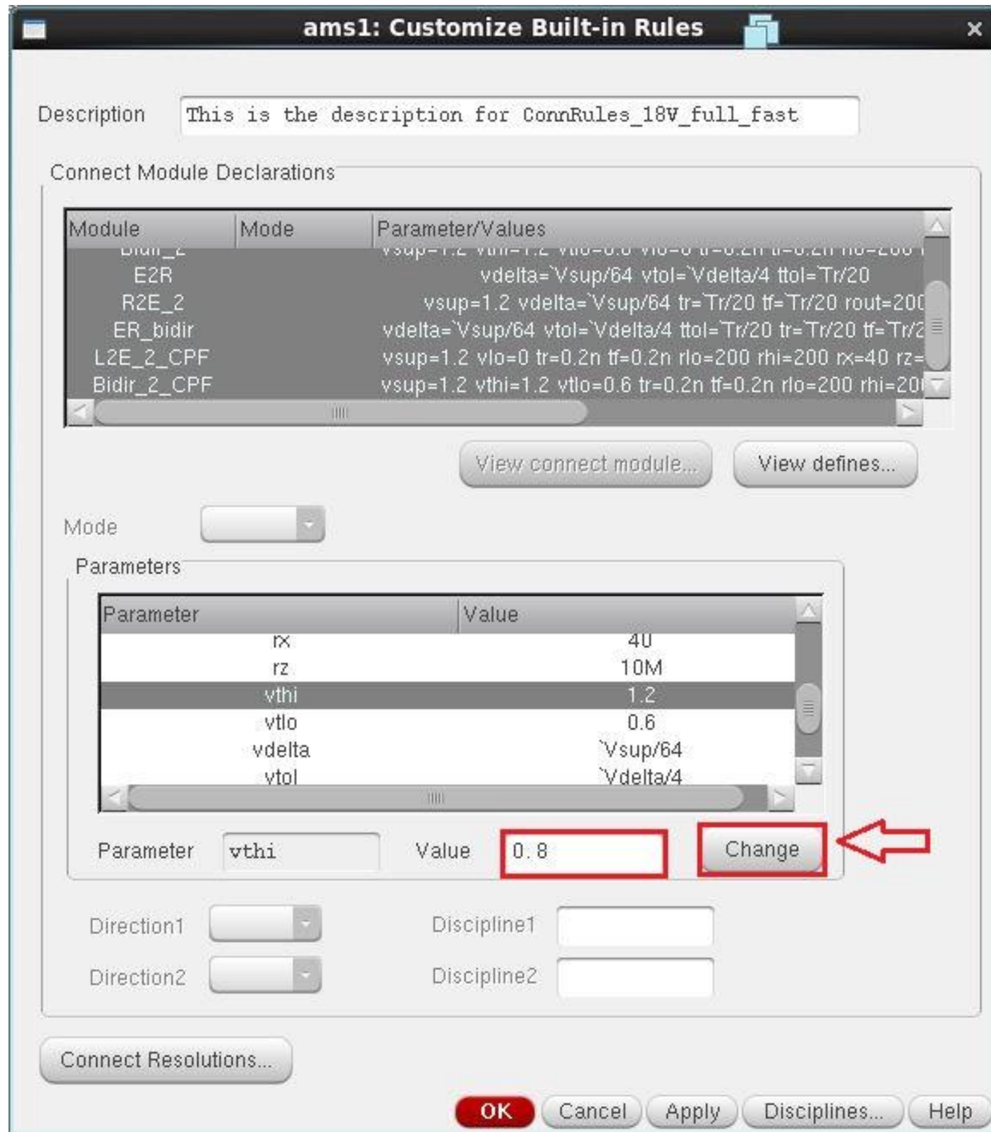


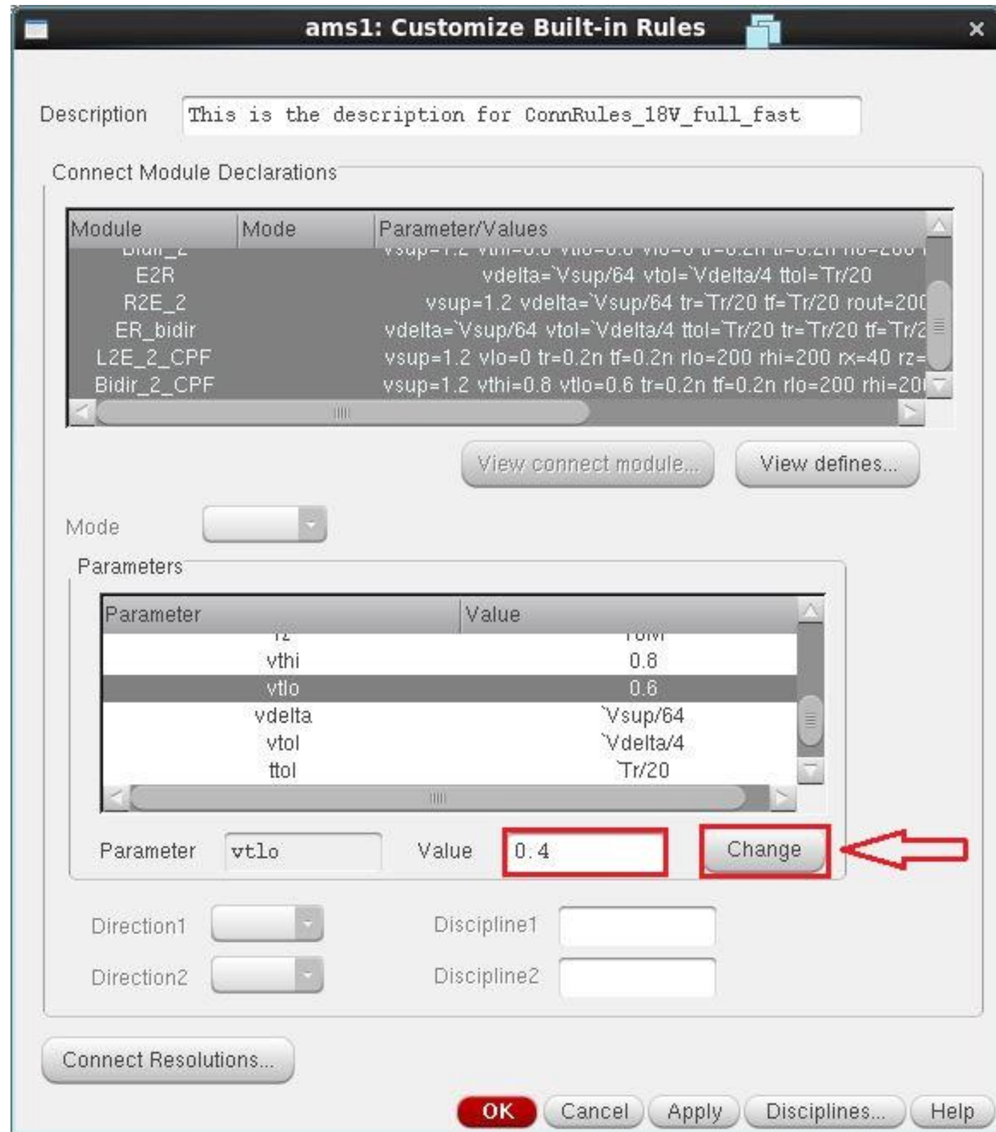
You will get a menu looks like the following.



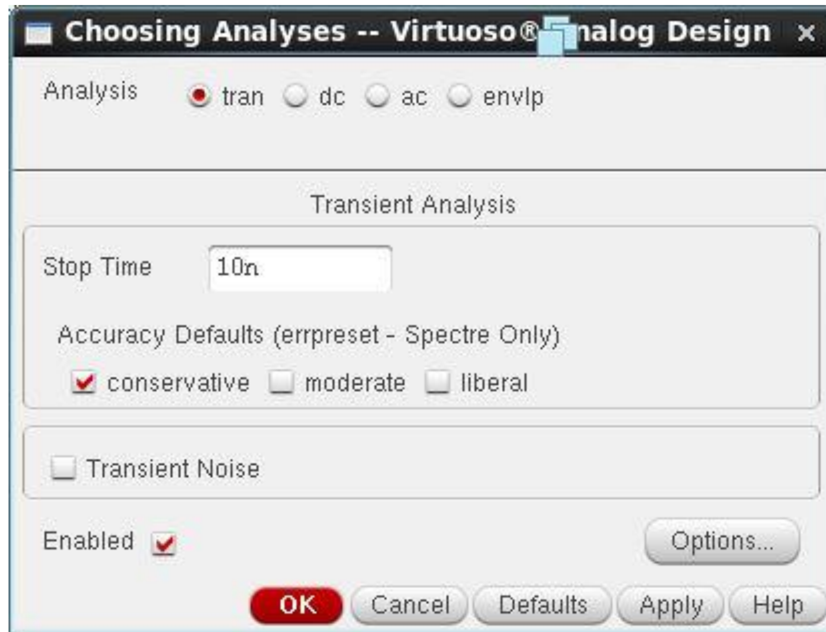
For this tutorial we use Vsupply=1.2v which is not included in the attached rules so we will choose any one and edit it using the **Customize** button. We will set **Vsup** to **1.2v** instead of **1.8v** and **vthi** to 0.8v (2*Vsup/3) and **vtlo** to 0.4v (Vsup/3). Do not forget to press the button **Change** after changing any value of these values. Vthi means threshold value for high logic and vtlo means threshold value for low logic. The range between vtlo and vthi is called the forbidden zone.


ams1: Customize Built-in Rules
Description
This is the description for ConnRules_18V_full_fast
Connect Module Declarations
Module
Mode
Parameter/Values
E2R
vdelta=`Vsup/64 vtol=`Vdelta/4 ttol=`Tr/20
R2E_2
ER_bidir
L2E_2_CPF
Bidir_2_CPF
View connect module...
View defines...
Mode
Parameters
Parameter
Value
vsup
1.8
vlo
0
tr
tf
rlo
200
Parameter
vsup
Value
1.2
Change
Direction1
Discipline1
Direction2
Discipline2
Connect Resolutions...
OK
Cancel
Apply
Disciplines...
Help

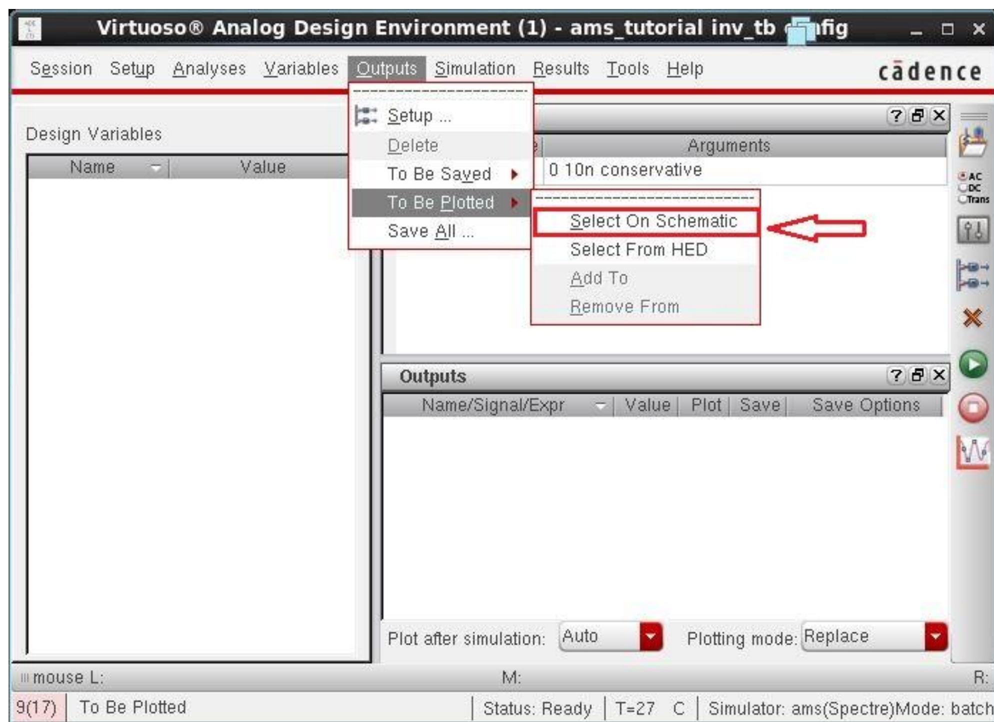

ams1: Customize Built-in Rules
Description
This is the description for ConnRules_18V_full_fast
Connect Module Declarations
Module
Mode
Parameter/Values
E2R
vdelta=`Vsup/64 vtol=`Vdelta/4 ttol=`Tr/20
R2E_2
vsup=1.2 vdelta=`Vsup/64 tr=`Tr/20 tf=`Tr/20 rout=20
ER_bidir
vdelta=`Vsup/64 vtol=`Vdelta/4 ttol=`Tr/20 tr=`Tr/20 tf=`Tr/2
L2E_2_CPF
vsup=1.2 vlo=0 tr=0.2n tf=0.2n rlo=200 rhi=200 rx=40 rz=
Bidir_2_CPF
vsup=1.2 vthi=1.2 vtlo=0.6 tr=0.2n tf=0.2n rlo=200 rhi=20
View connect module...
View defines...
Mode
Parameters
Parameter
Value
rz
10M
vthi
1.2
vtlo
0.6
vdelta
`Vsup/64
vtol
`Vdelta/4
Parameter
vthi
Value
0.8
Change
Direction1
Discipline1
Direction2
Discipline2
Connect Resolutions...
OK
Cancel
Apply
Disciplines...
Help



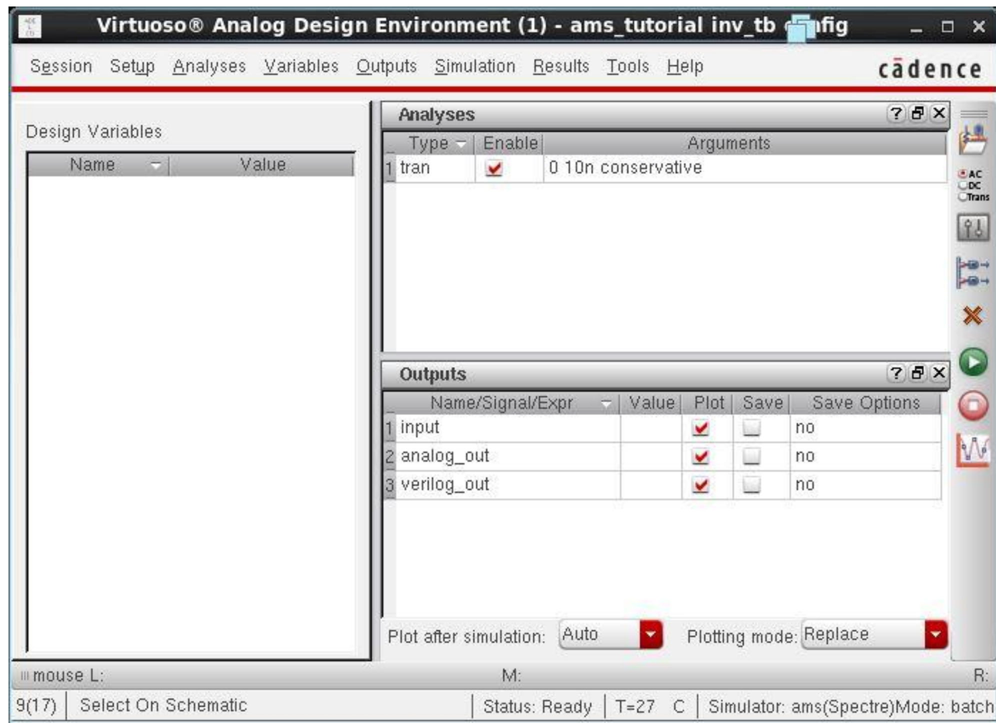
Then choose analysis and we will simulate the circuit for 10ns



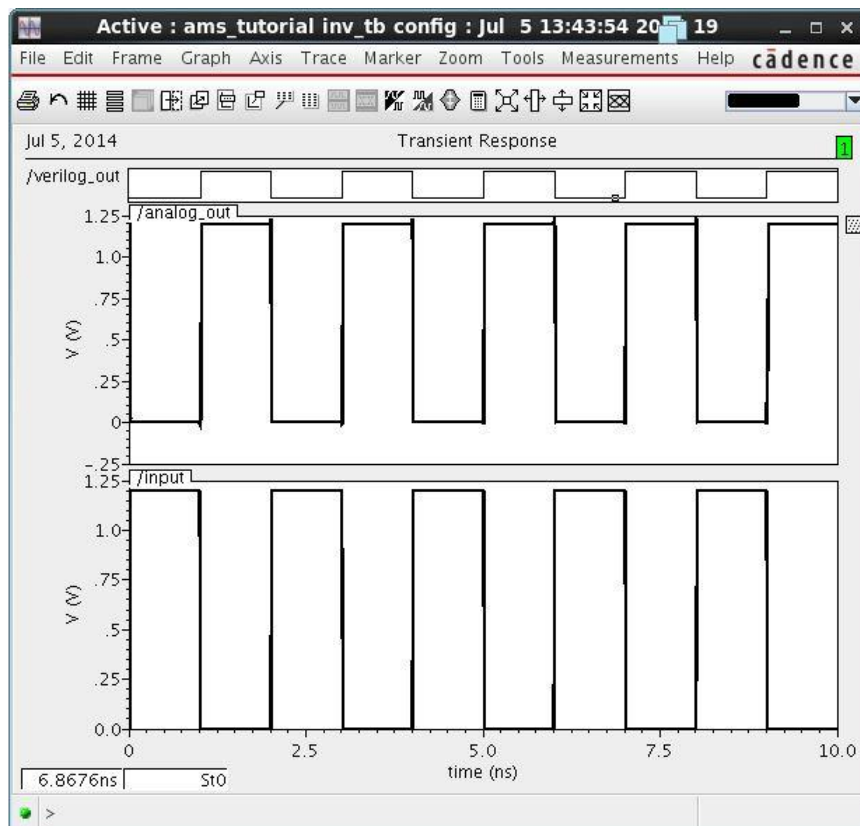
Then select the signals to be plotted **Outputs>To Be Plotted>Select On Schematic**



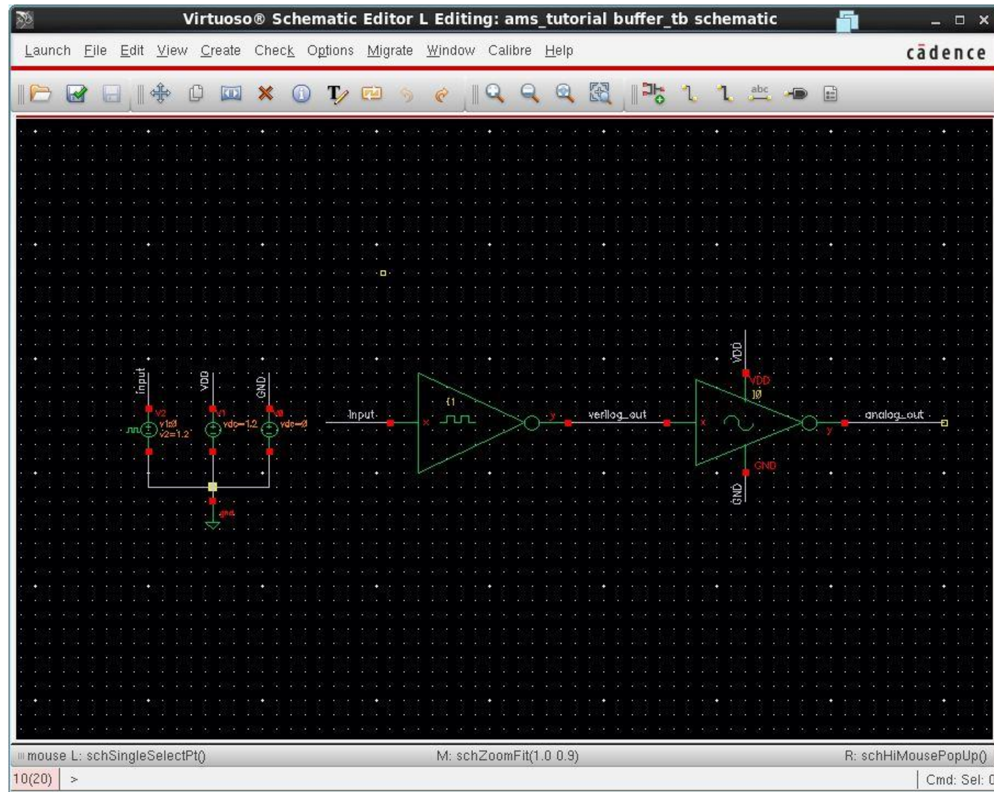
We are interested in these signals, input, analog_out and Verilog_out.



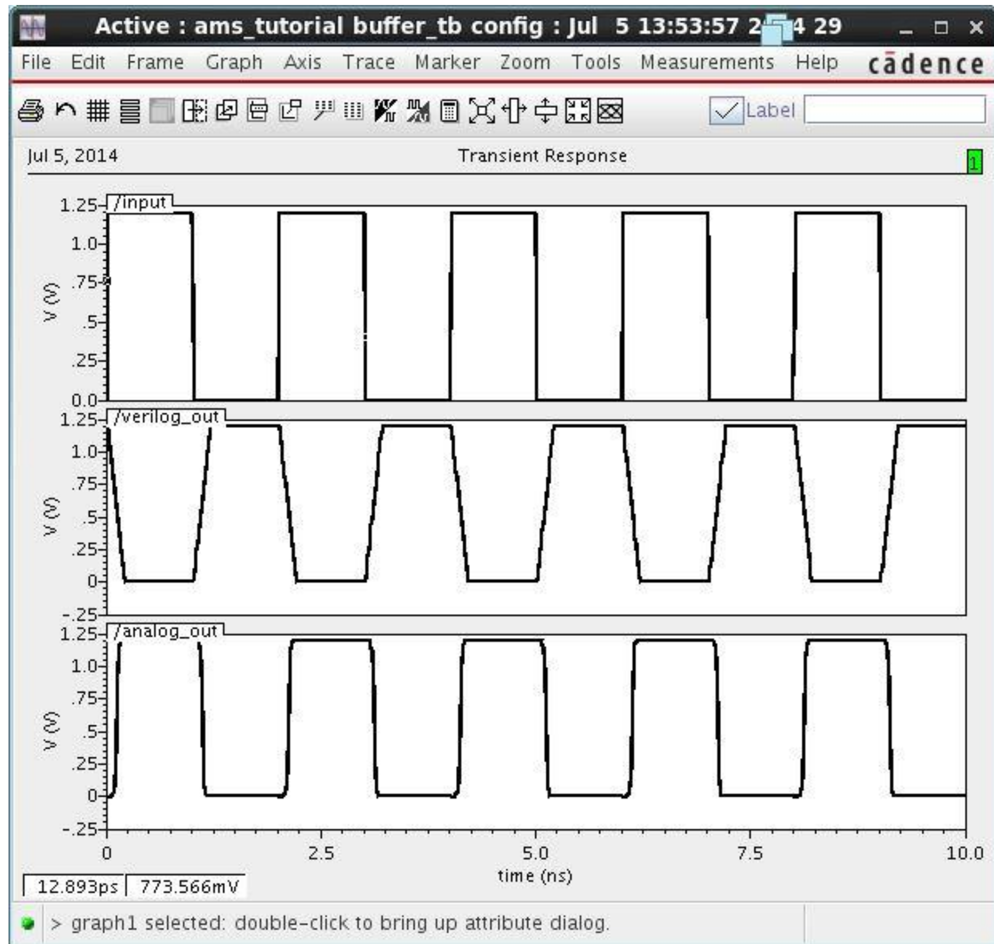
Then Run Simulation. The following are the results from both the analog inverter and the Verilog digital inverter and they are the same.



We will now try the cascaded configuration of both inverters to act as a buffer.

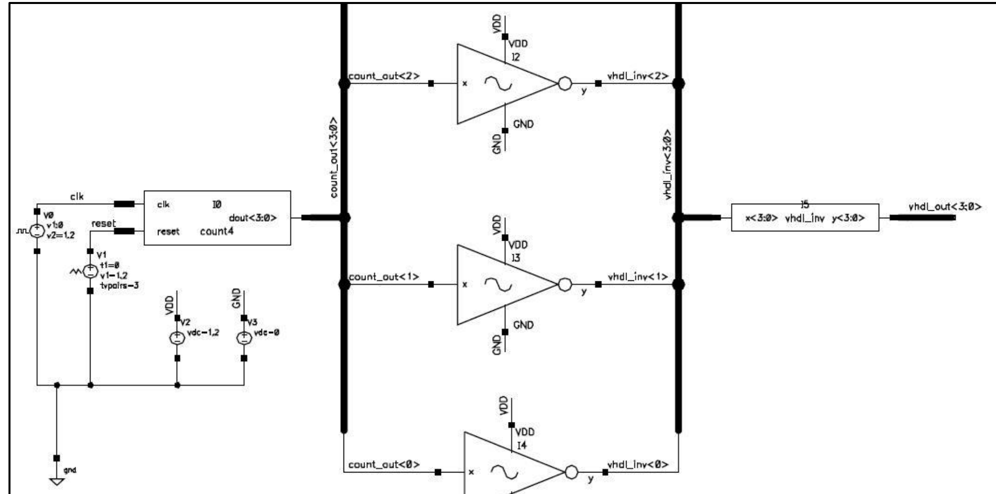


The simulation result can be found in the following figure. One can simply notice that, the analog_out signal is the same as the input signal. (The operation of a typical buffer).



Part2:

In this part we are going to simulate a design consisting of 3 main blocks, digital Verilog 4-bit counter, analog 4-bit inverter and digital VHDL 4-bit inverter. The block diagram of this circuit is as follows

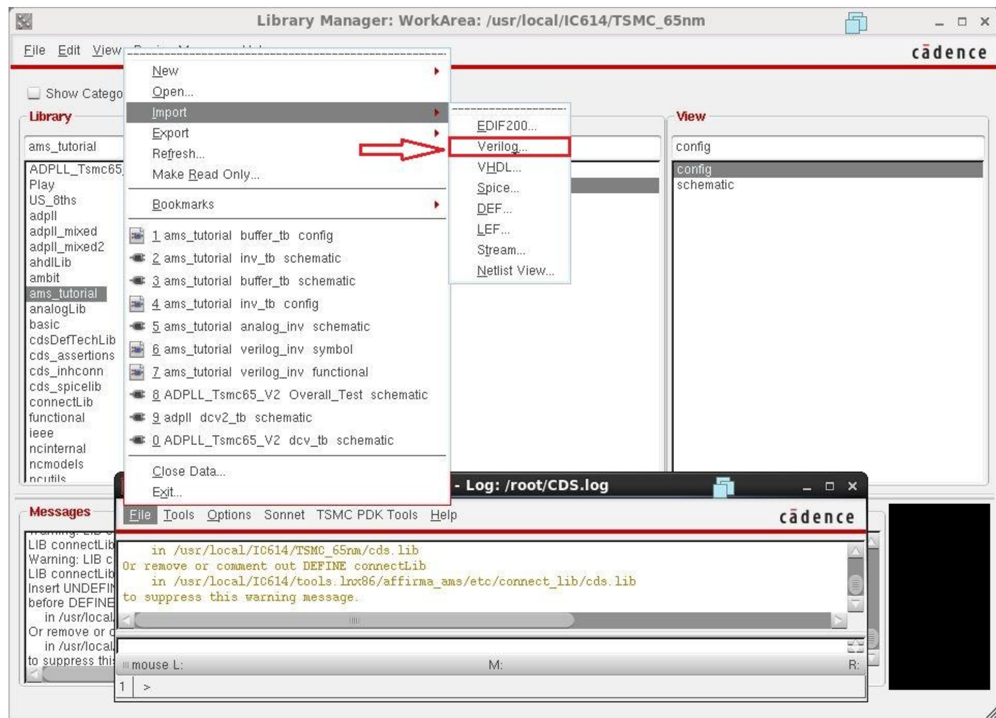


The analog inverters are taken **from Part 1** and the codes for Verilog counter and VHDL inverter is as follows

```
module count4(input clk,reset,output[3:0] dout);
reg[3:0] count_reg;
wire[3:0] count_next;
//state register
always@(posedge clk,posedge reset)
   if(reset)
   count_reg <= 0;
   else
   count_reg <= count_next;
//next state logic
assign count_next = count_reg+1;
//output logic
assign dout = count_reg;
endmodule
```

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
entity vhdl_inv is
port (x: in STD_LOGIC_VECTOR (3 downto 0);
y: out STD_LOGIC_VECTOR (3 downto 0));
end;
architecture behavioral of vhdl_inv is
begin
y <= not x;
end;
```

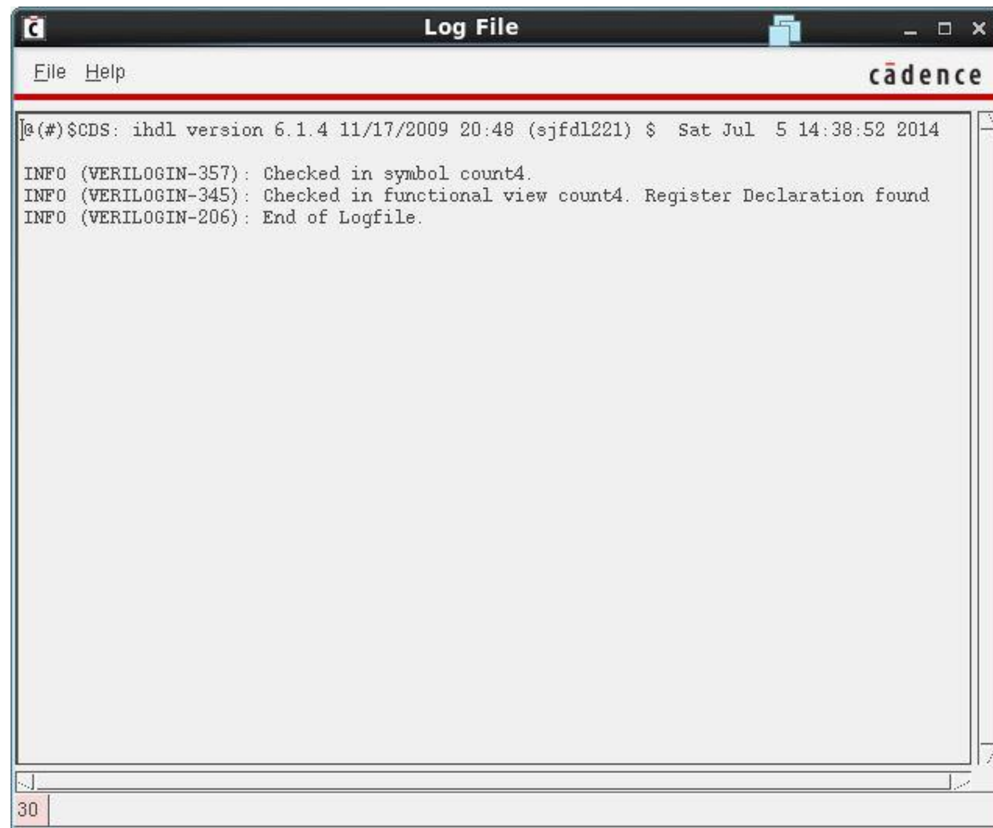
We will start by importing both the Verilog counter and the VHDL inverter into virtuoso. The following figure is to import the Verilog counter.



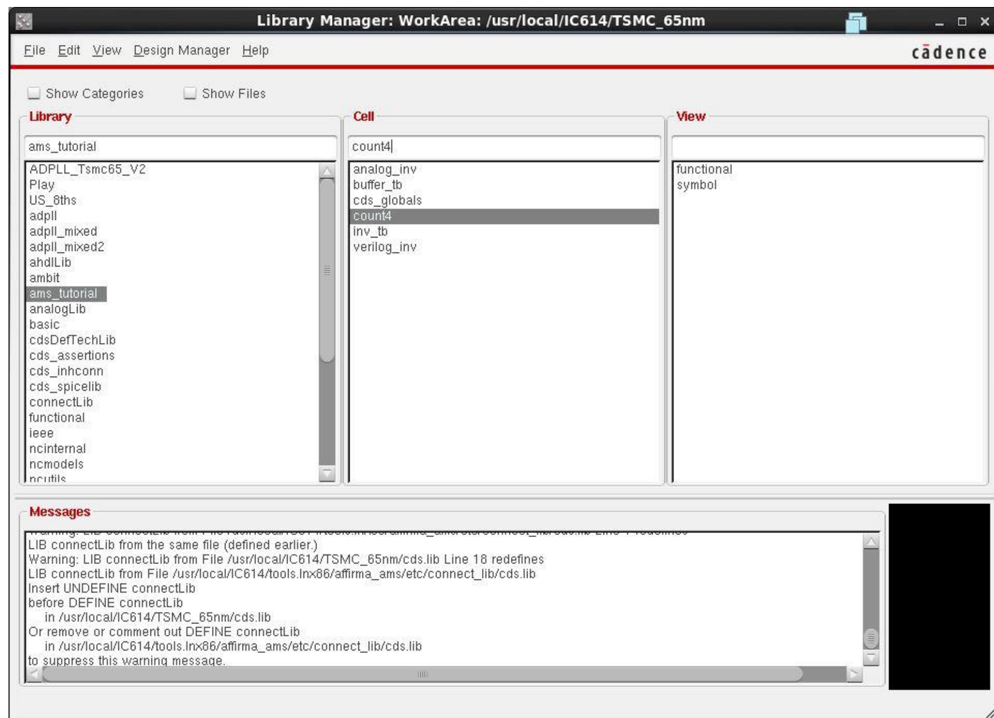
Choose Target Library and Verilog file as follows

You will get a menu like the following after pressing **OK**. You may press **Yes** to view the logfile.

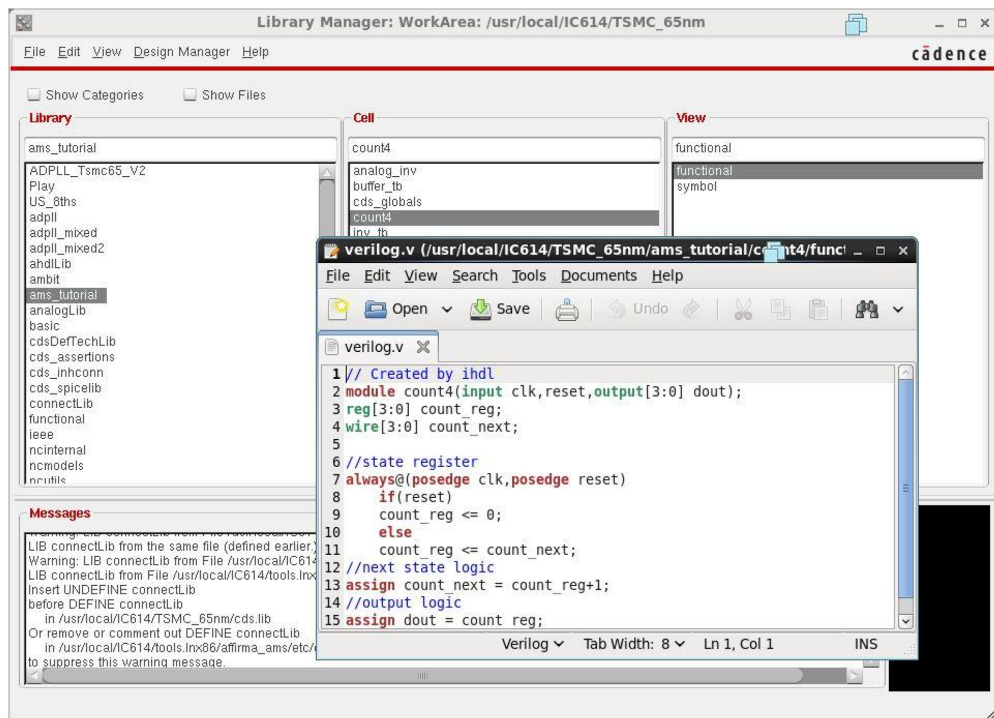
The logfile looks like the following. Reporting that your import process has been completed.



Now check your target library (**ams_tutorial** in out example) to find the Verilog counter **count4** with two views, **functional** (The Verilog code) and **symbol.**



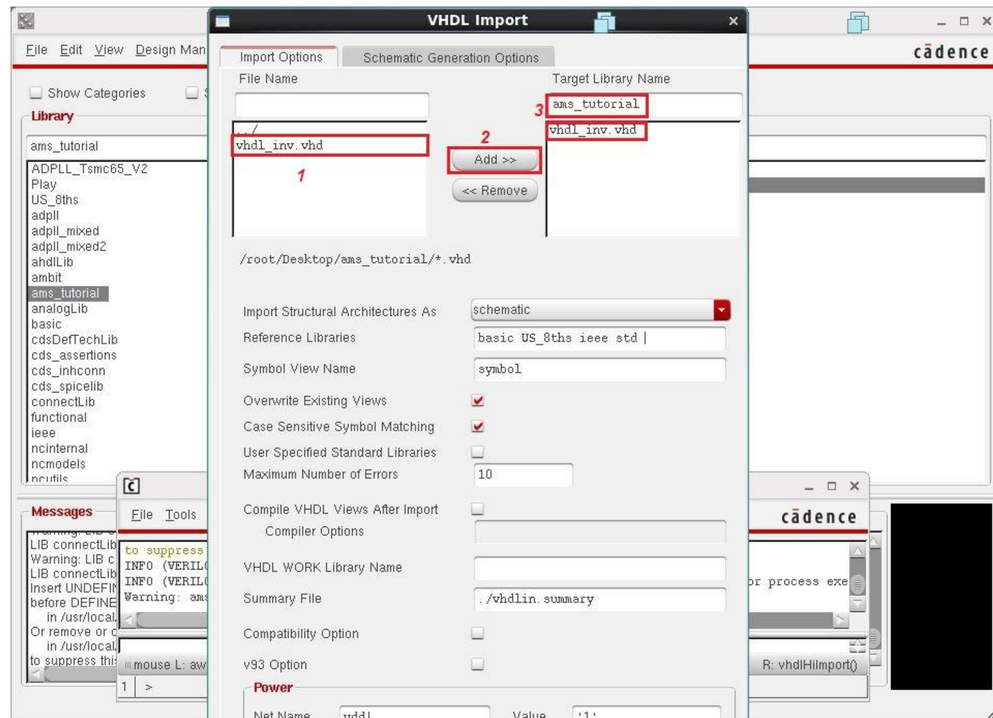
Double click on functional to check the code.



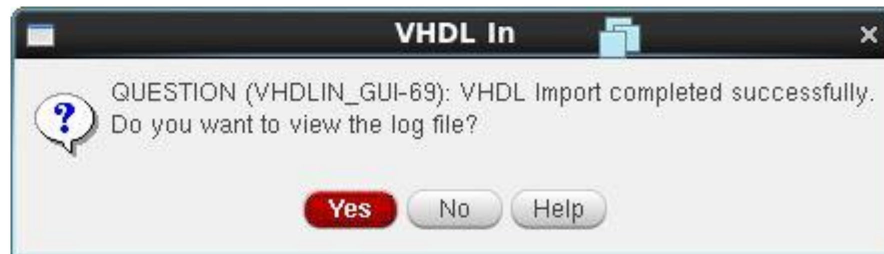
Double click on symbol view to check the symbol. You may need to edit it, we will just leave it as is.

Next step is to import the VHDL 4-bit inverter.

Choose your target library (ams_tutorial in our example) and VHDL file. Leave all other fields as they are.

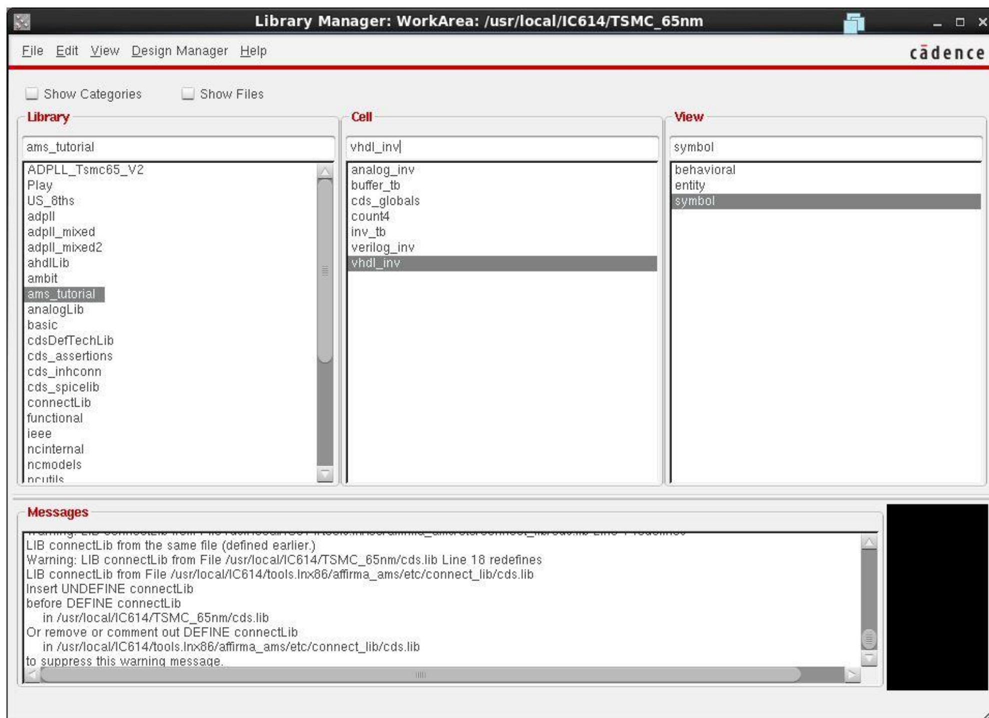


After pressing **OK** you will get the following menu, press **Yes** to check the logfile.

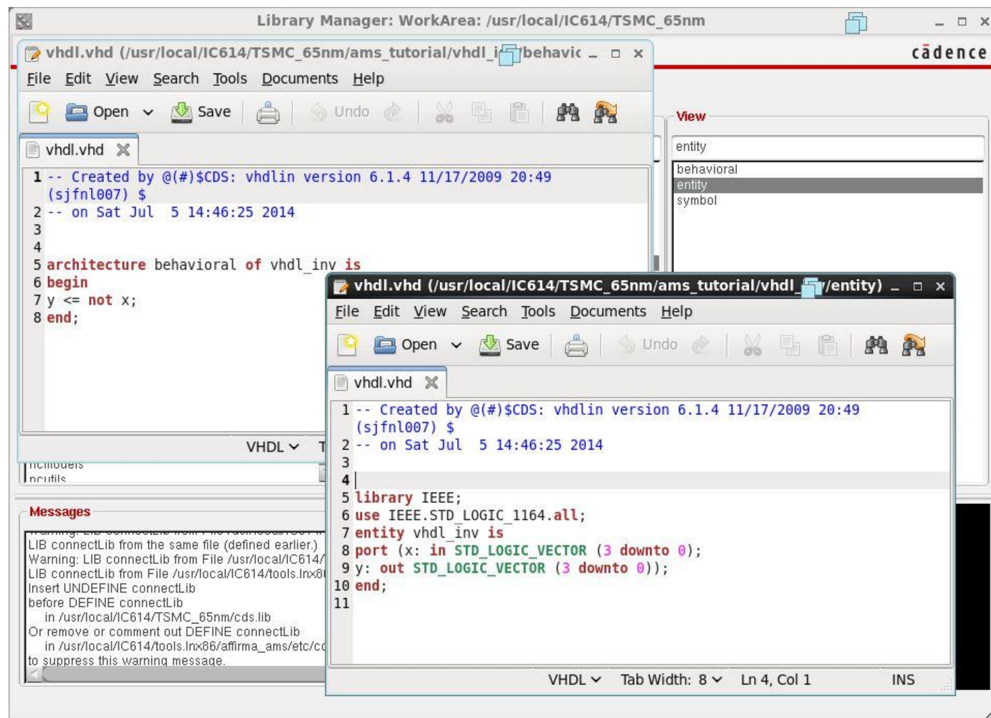


```
VHDL ToolBox Log File
File  Help                                                              cādence

INFO (VHDLIN-284): VHDL In Run Summary
@(#)$CDS: vhdlin version 6.1.4 11/17/2009 20:49 (sjfnl007) $  Sat Jul  5 14:46:25 2014

INFO (VHDLIN-238): Processing VHDL source file: /root/Desktop/ams_tutorial//vhdl_inv.vhd.
INFO (VHDLIN-264): Done.
INFO (VHDLIN-244): Vhdl Design Unit: vhdl_inv : Entity
INFO (VHDLIN-243): Created symbol view of type symbol.
INFO (VHDLIN-245): Created entity view of type Vhdl
INFO (VHDLIN-251): Vhdl Design Unit: vhdl_inv.behavioral : Architecture
INFO (VHDLIN-245): Created behavioral view of type Vhdl
INFO (VHDLIN-255):        -> (/root/Desktop/ams_tutorial//vhdl_inv.vhd,9) Behavioral expression.


*****************************************************************

INFO (VHDLIN-229): Number of file(s) processed in this round is 1
*****************************************************************
INFO (VHDLIN-231): Number of file(s) successfully imported in this round is 1
*****************************************************************

33
```

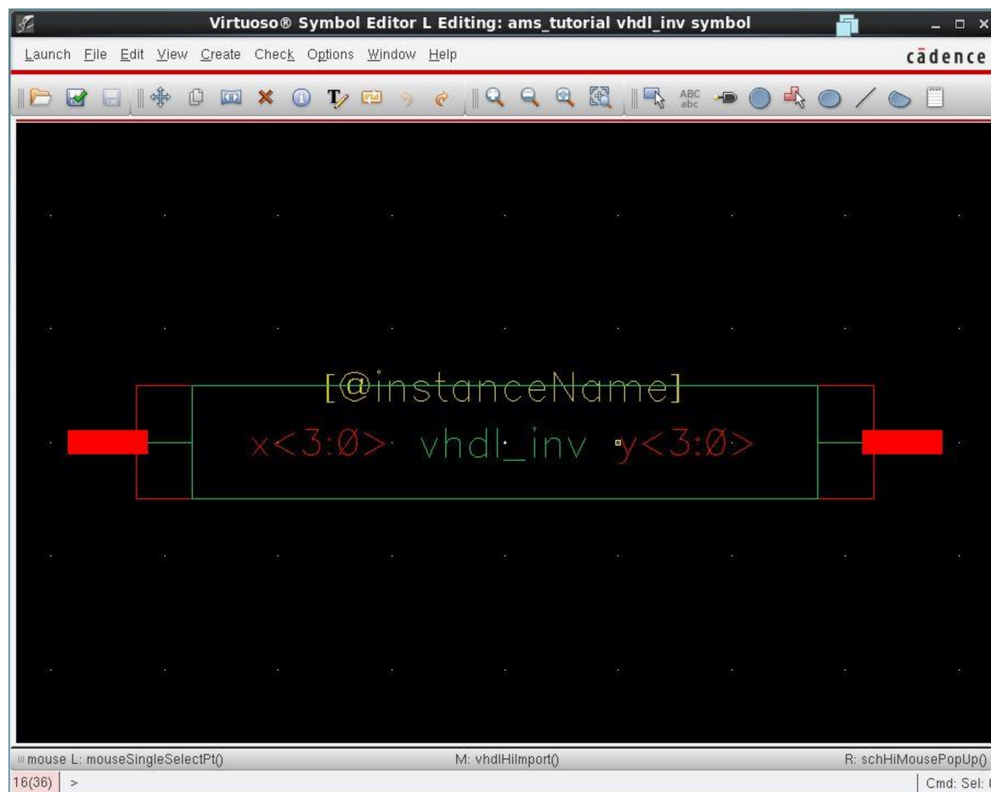
Now you can check your cell, named **vhdl_inv**.



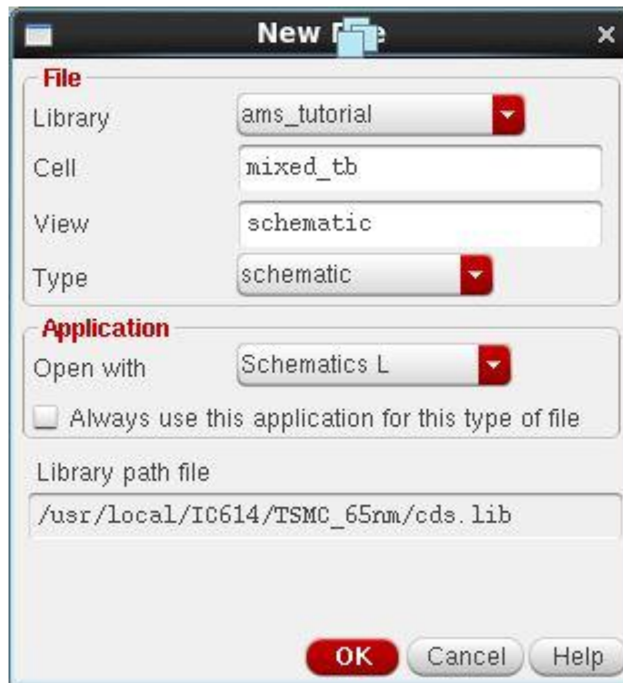
Double click on behavioral view and entity view to check the code.

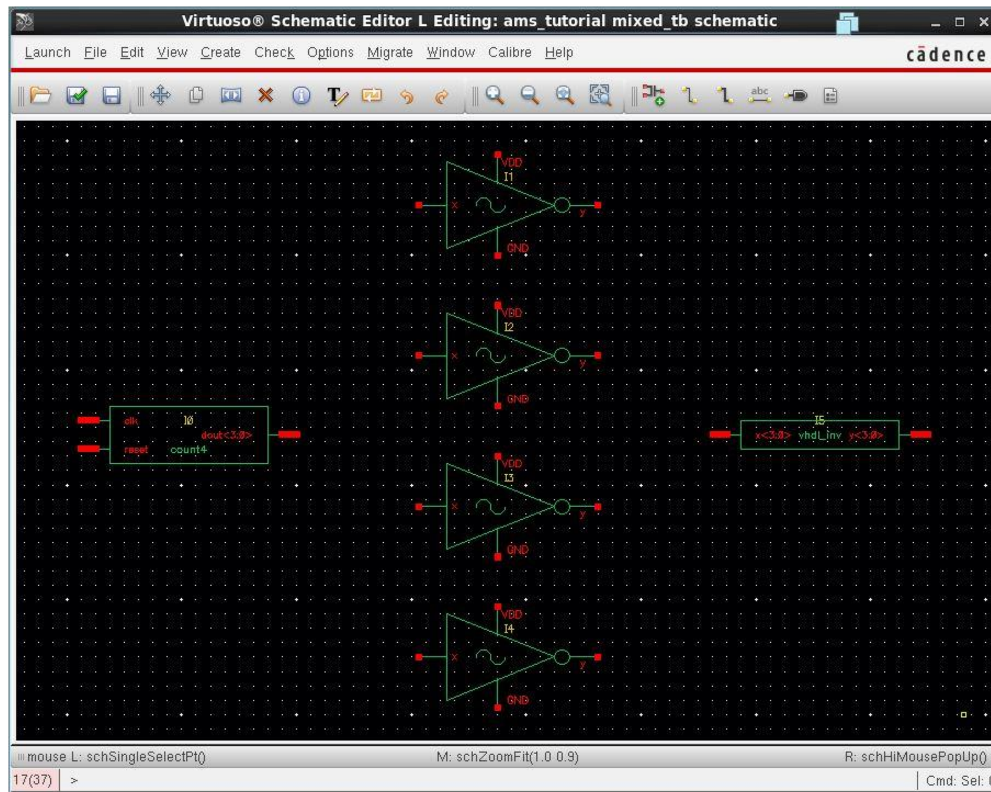


You can edit the symbol. We will use it as it is.

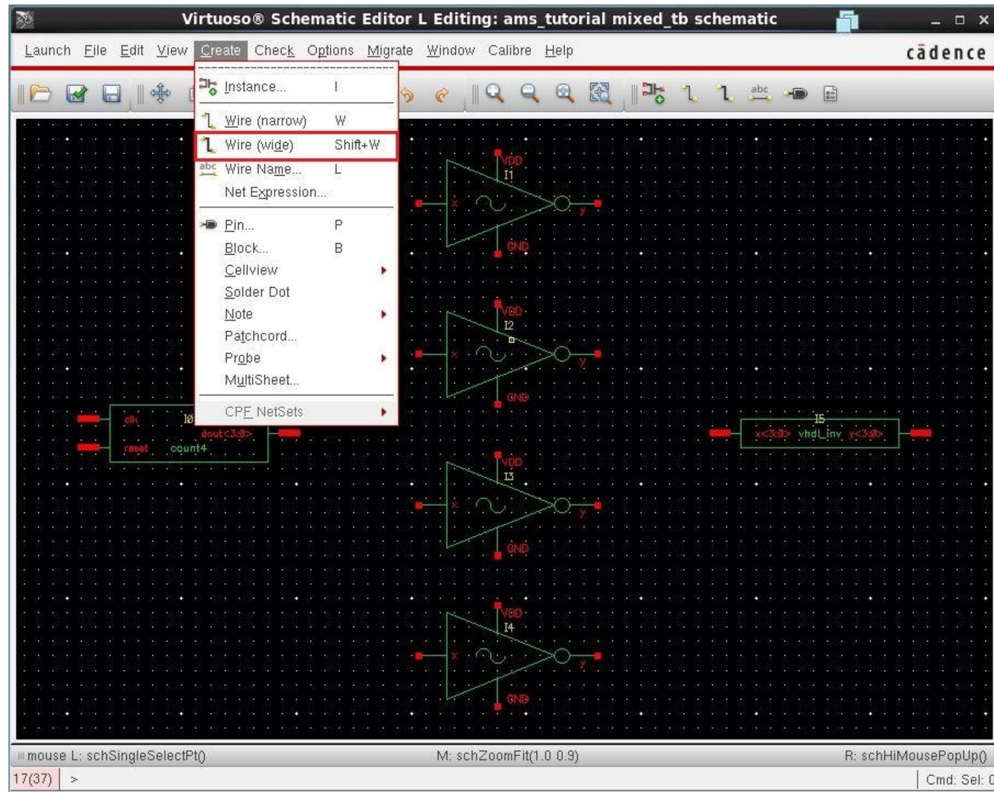


Next step is to make the test bench for the block diagram mentioned in the beginning of this section. We will name it **mixed_tb** (optional).

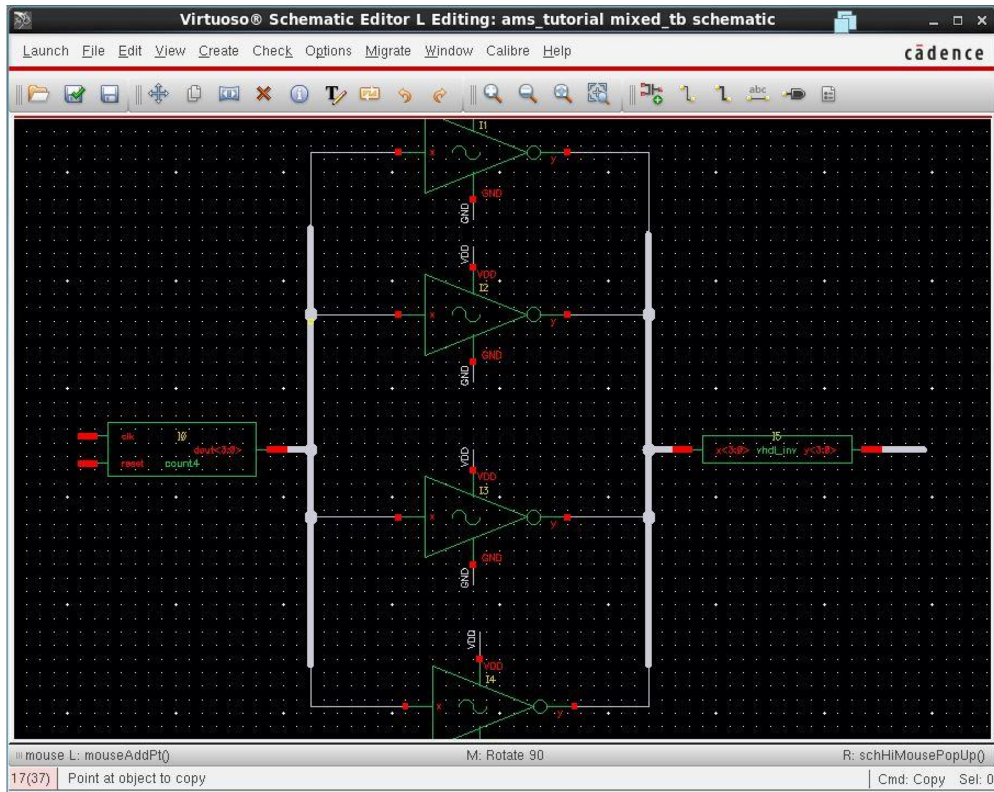




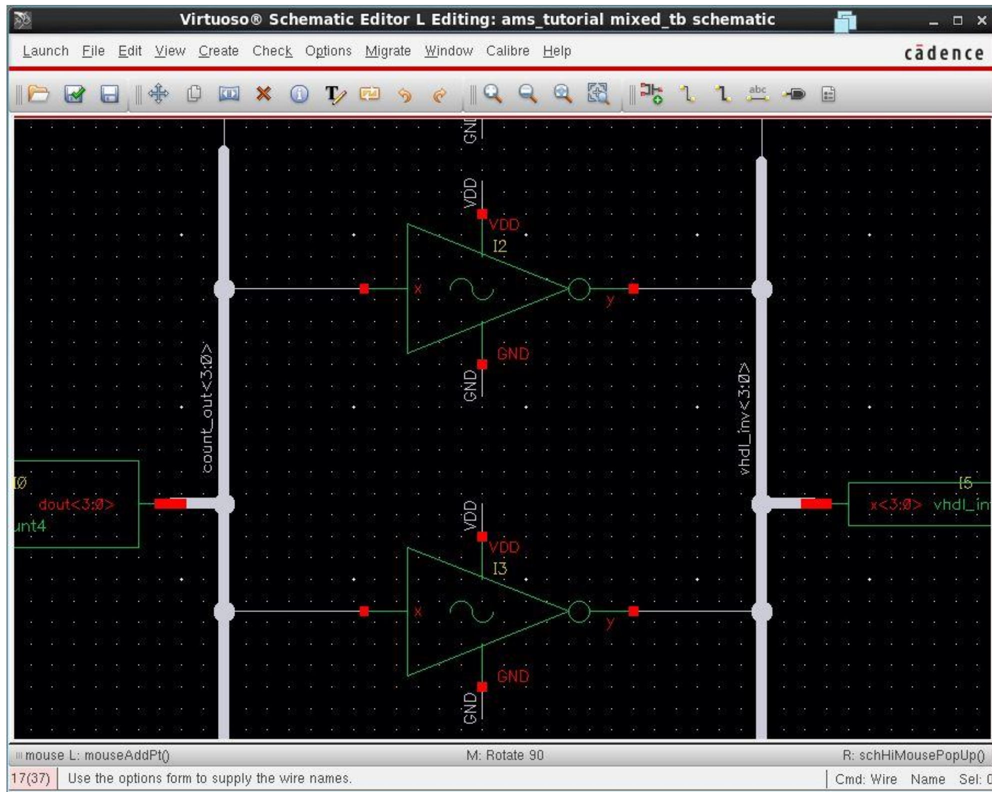
Choose **create>wire (wide)** to create buses.



Connect these blocks together as follows

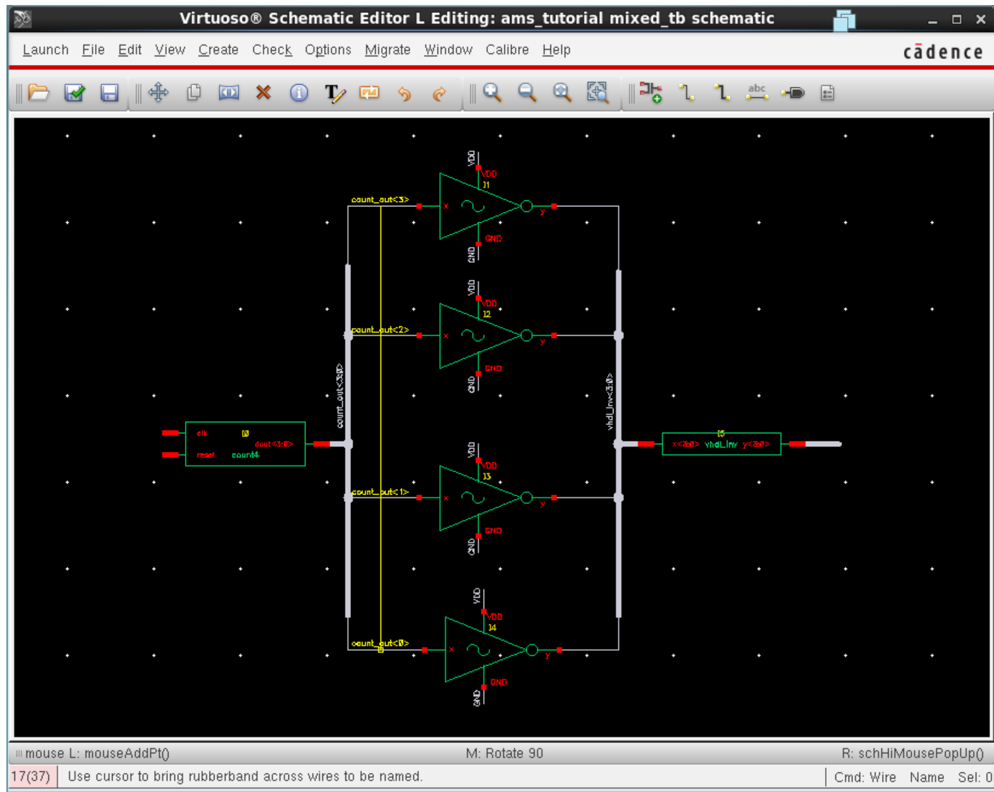


Add labels for buses by pressing l.

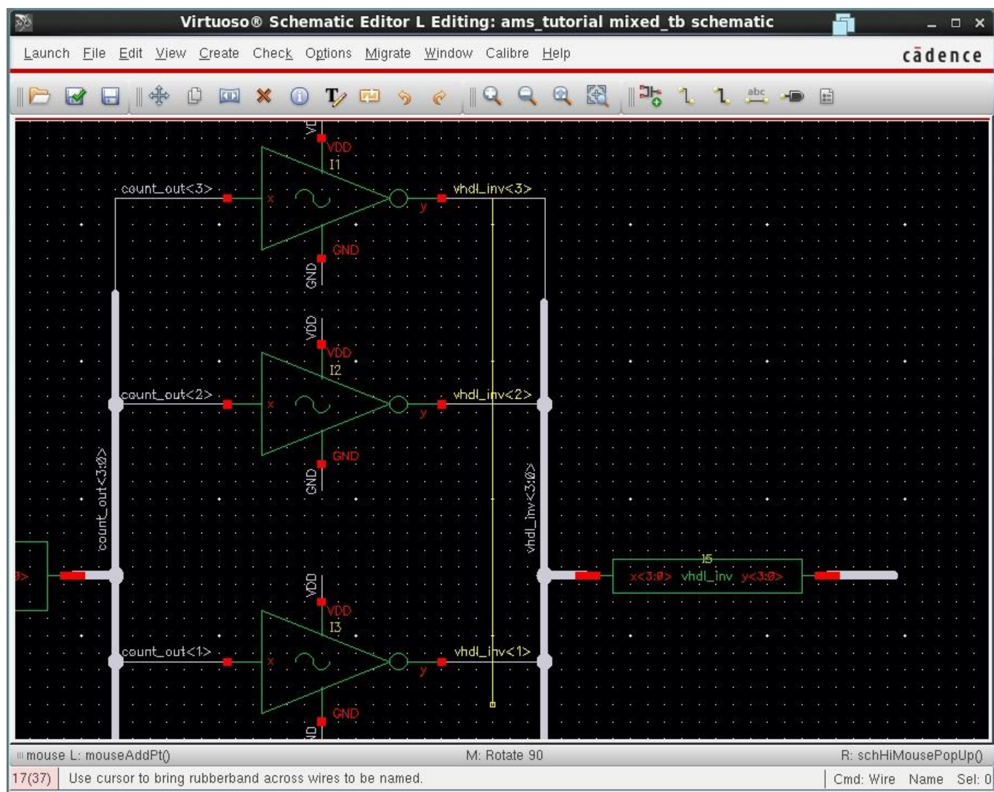


Add labels for the wires connected to the buses.

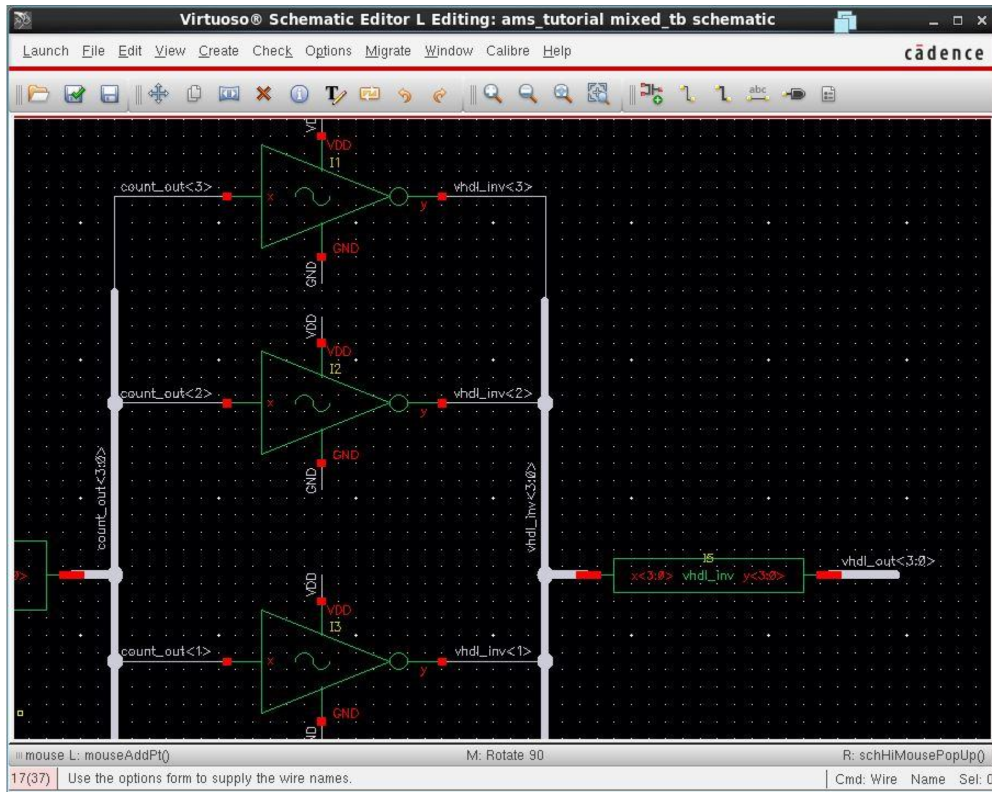
After pressing **OK,** left click on the first wire and move to the last wire across the wires between them and finally left click on the last wire.



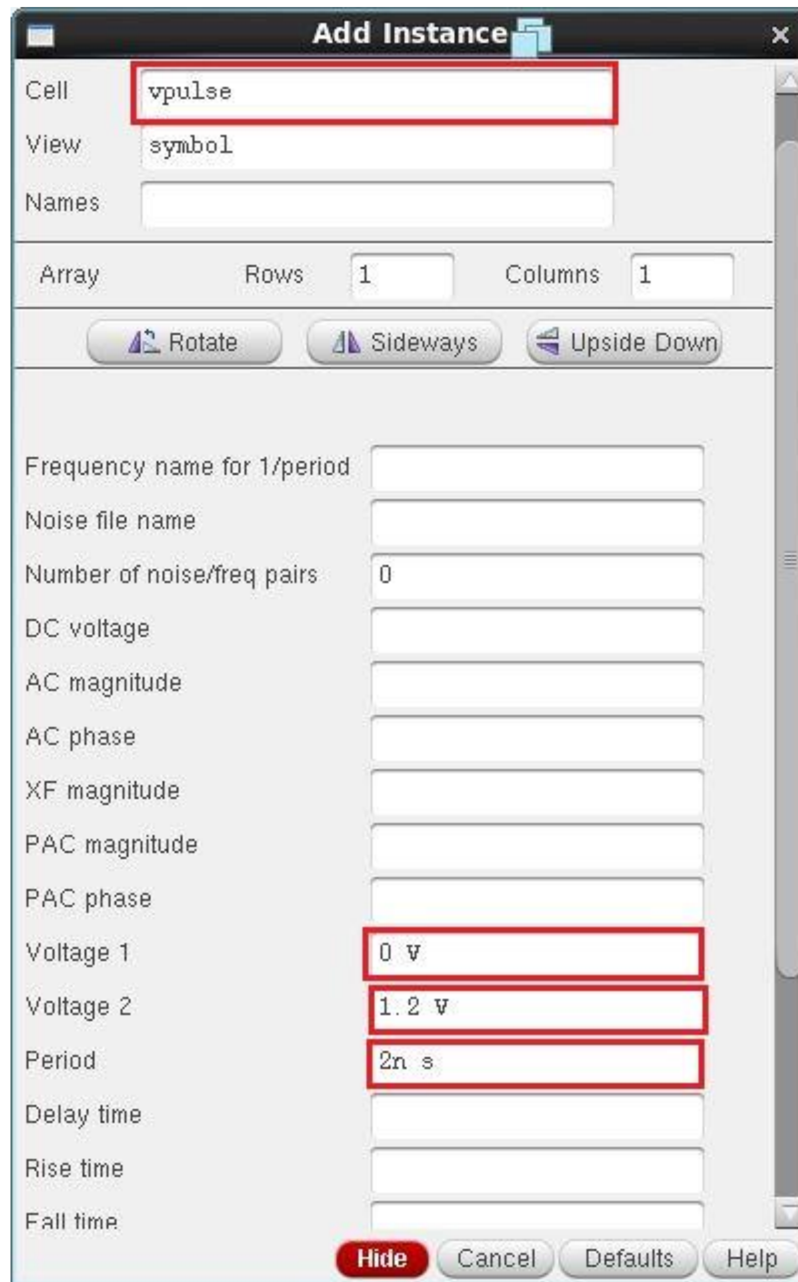
Virtuoso® Schematic Editor L Editing: ams_tutorial mixed_tb schematic
Launch File Edit View Create Check Options Migrate Window Calibre Help
cādence
mouse L: mouseAddPt()
M: Rotate 90
R: schHiMousePopUp()
17(37)
Use cursor to bring rubberband across wires to be named.
Cmd: Wire Name Sel: 0



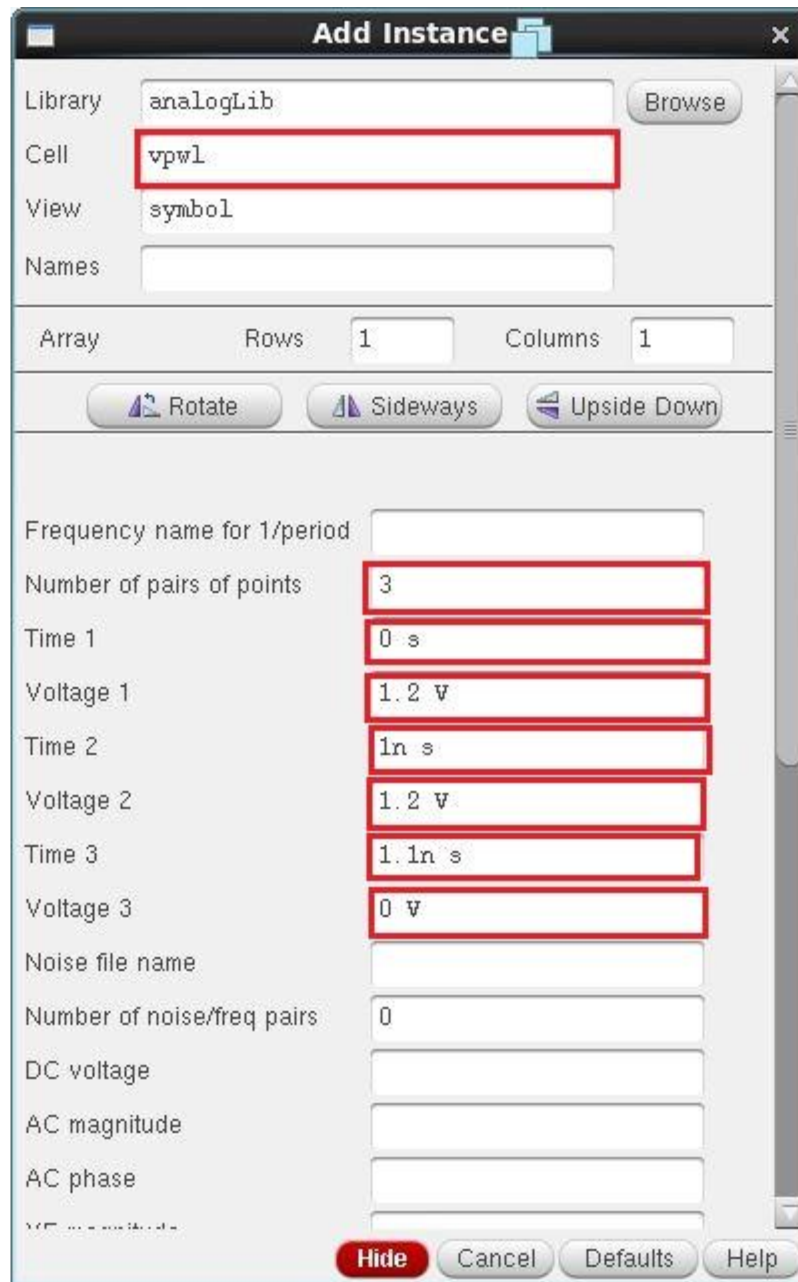
Virtuoso® Schematic Editor L Editing: ams_tutorial mixed_tb schematic
Launch File Edit View Create Check Options Migrate Window Calibre Help
cādence
VDD
I1
count_out<3>
vhdl_inv<3>
GND
I2
count_out<2>
vhdl_inv<2>
count_out<3:0>
vhdl_inv<3:0>
I5
x<3:0> vhdl_inv y<3:0>
I3
count_out<1>
vhdl_inv<1>
mouse L: mouseAddPt()
M: Rotate 90
R: schHiMousePopUp()
17(37)
Use cursor to bring rubberband across wires to be named.
Cmd: Wire Name Sel: 0



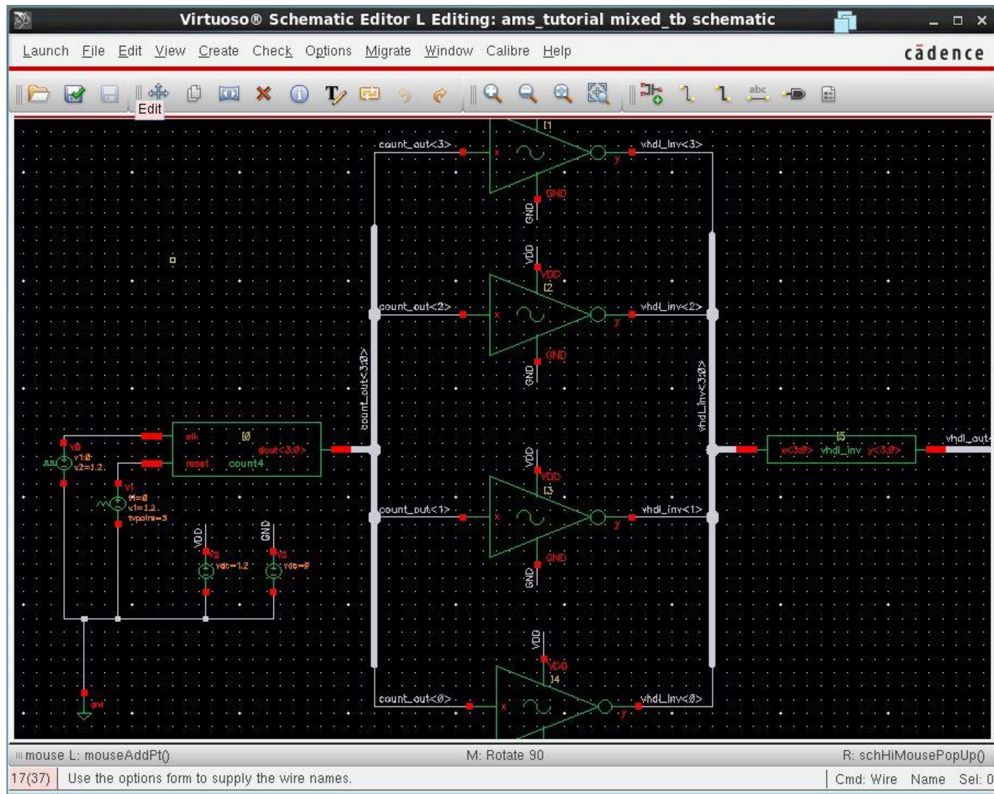
Add sources to your design clk, reset, VDD and GND. For the clk we use Vpulse with period 2ns.



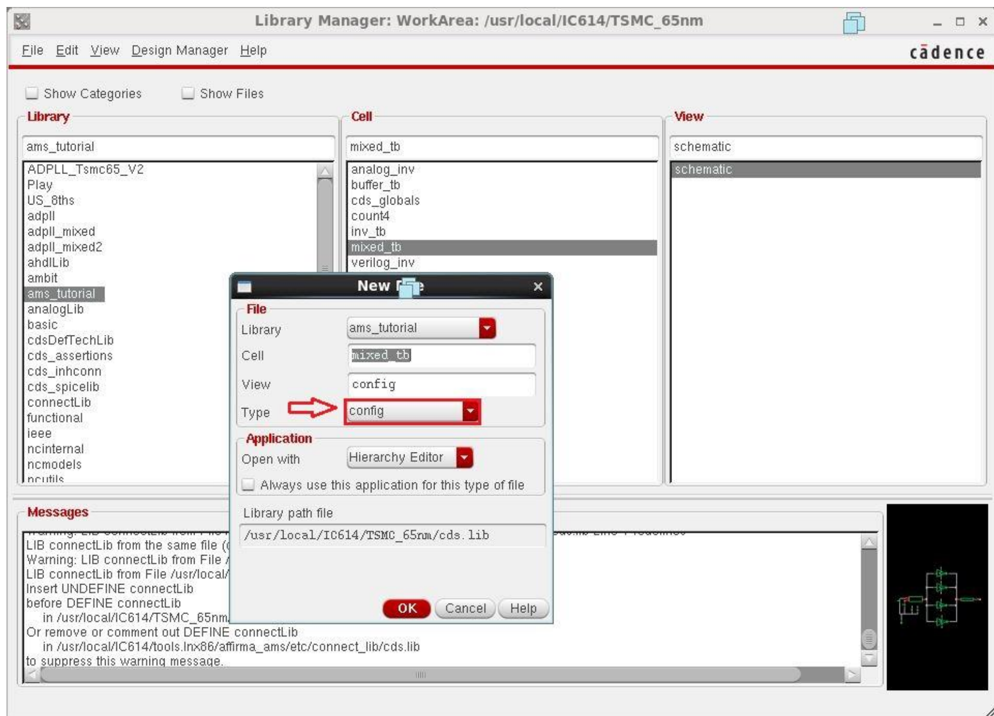
For the reset signal, we use Vpwl with the following configuration for an active high reset.

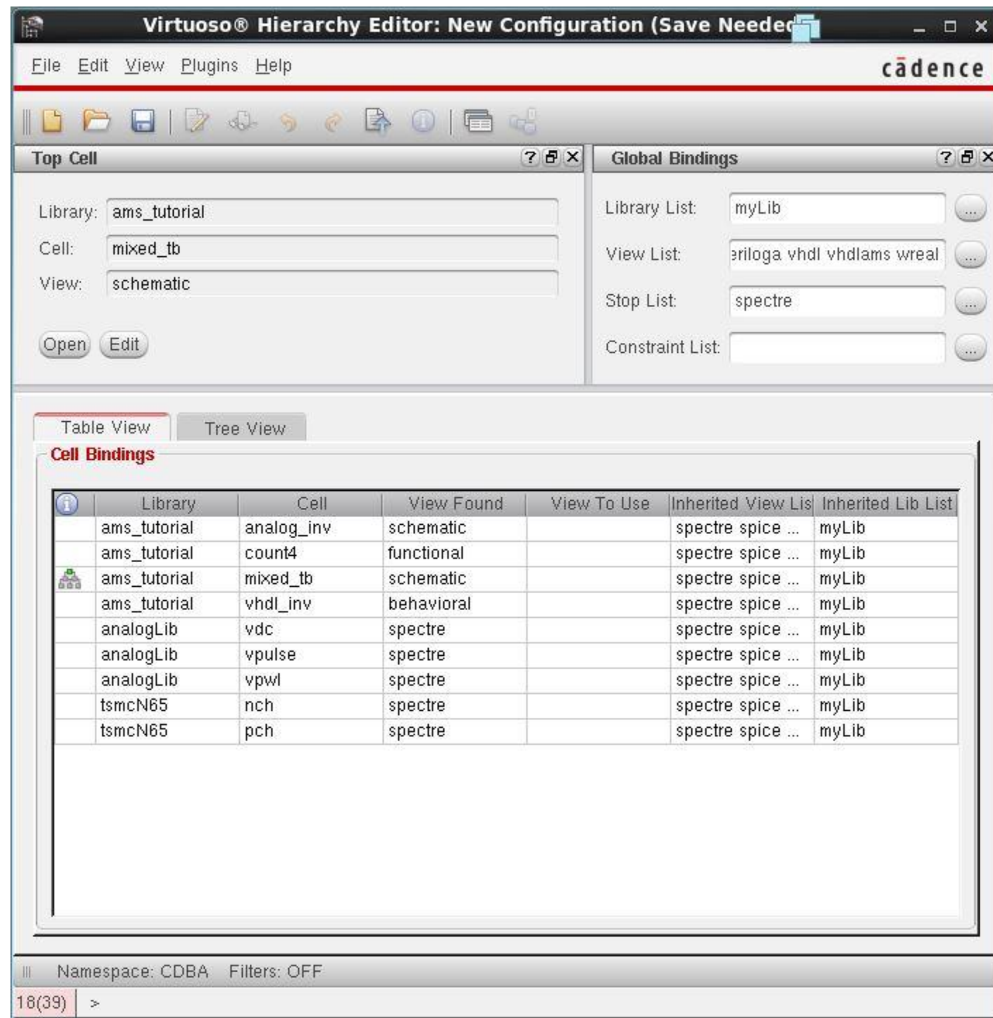


The final schematic after adding sources and labels is as follows

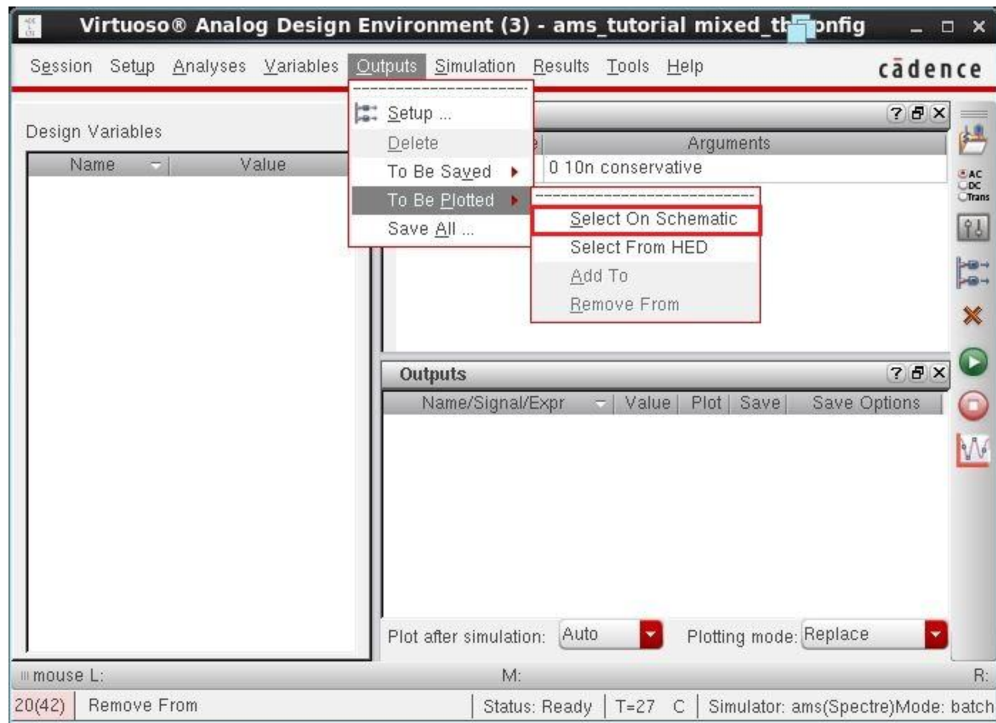


Check and Save then close this schematic and create a **config** view for it as we did for **Part 1**.

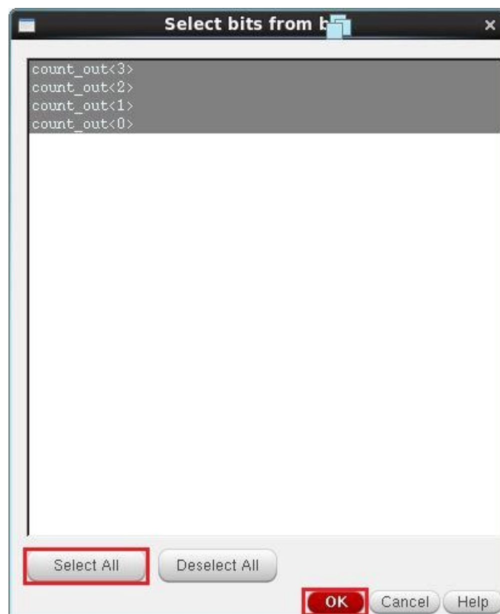


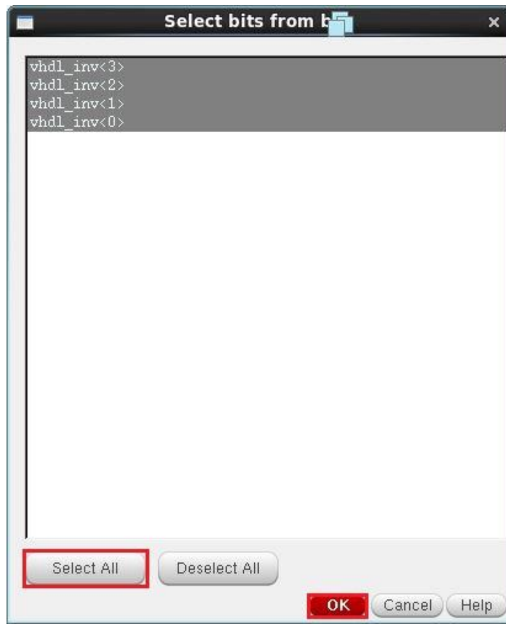


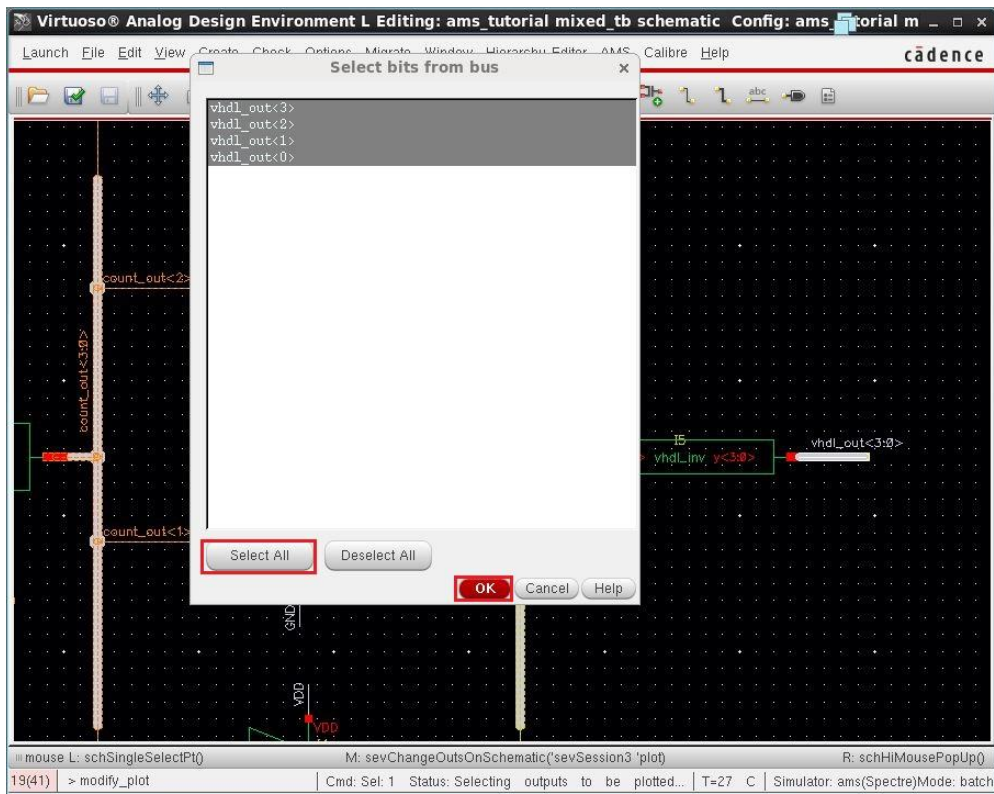
Choose the **ams simulator** as we did for **Part 1** and edit **connect rules** then select signal to be plotted.



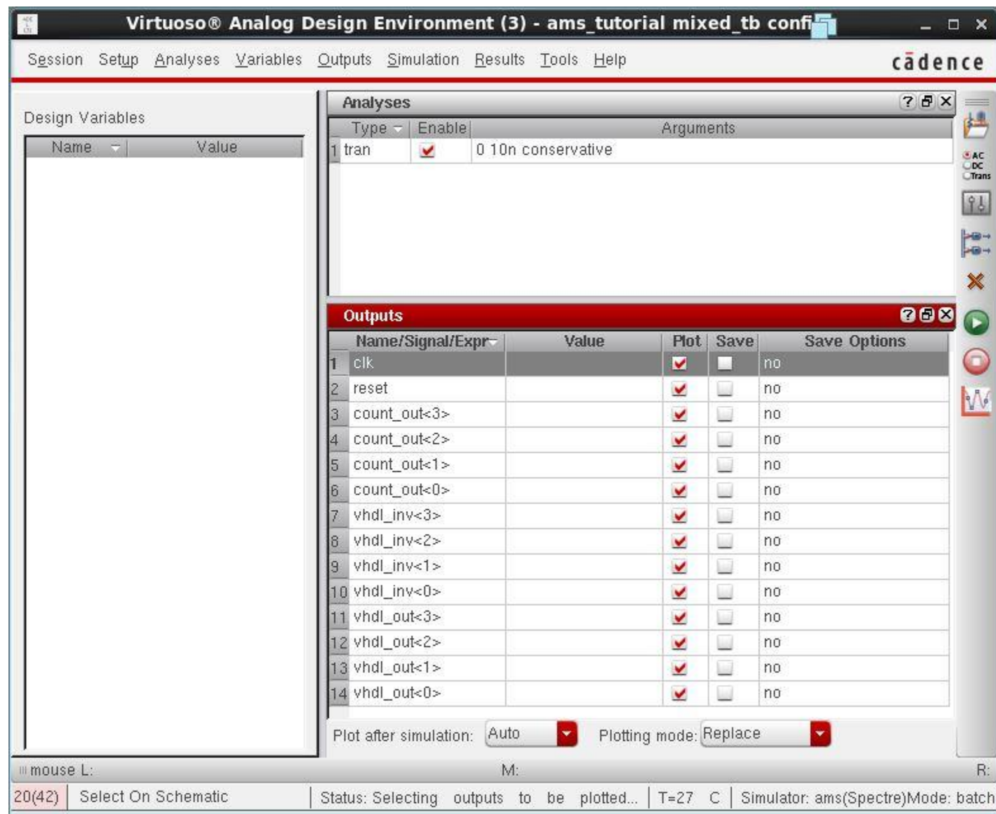
We are interested in plotting these signals, count_out<3:0>, vhdl_inv<3:0> and vhdl_out<3:0>.



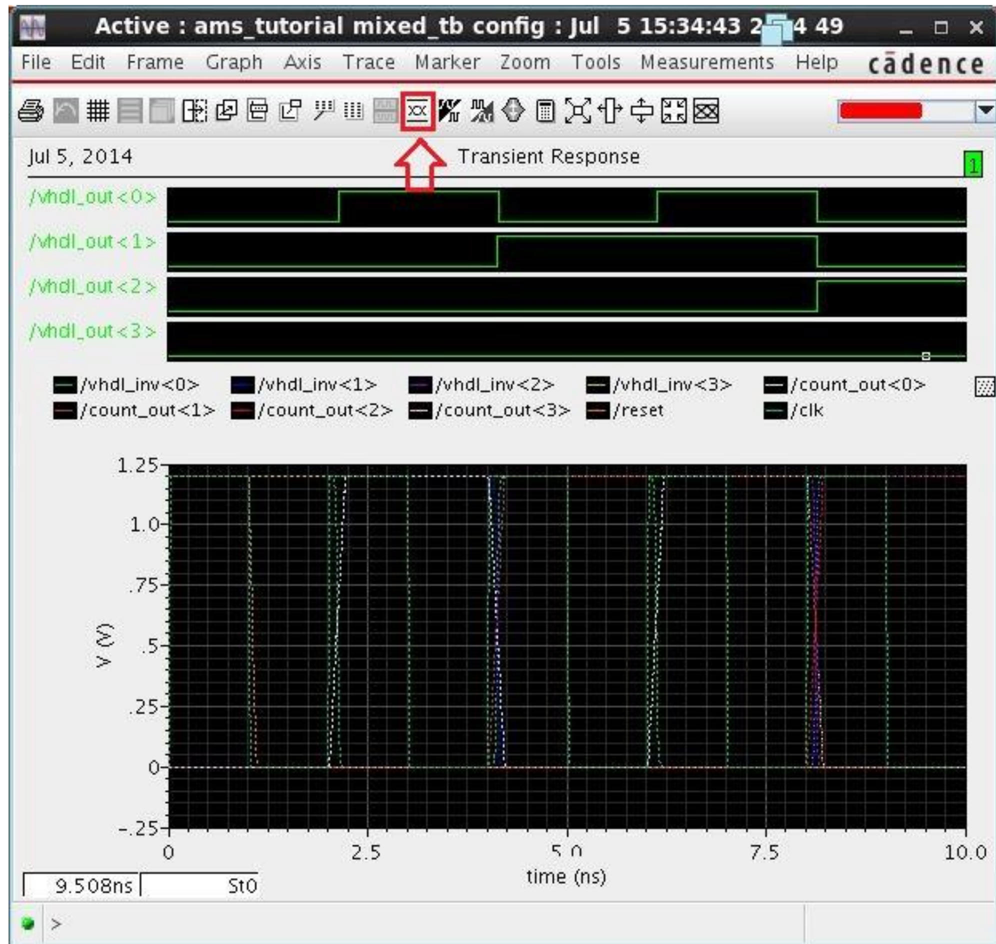




Then **Run Simulation**.



From the results figure, we can select the digital signals of vhdl_out<3:0> and convert them to a bus.



Create Bus

Help

Bit	Signal	Results Directory
0 (lsb)	/vhdl_out<0>	/root/simulation/mi...
1	/vhdl_out<1>	/root/simulation/mi...
2	/vhdl_out<2>	/root/simulation/mi...
3 (msb)	/vhdl_out<3>	/root/simulation/mi...

alpha Sort

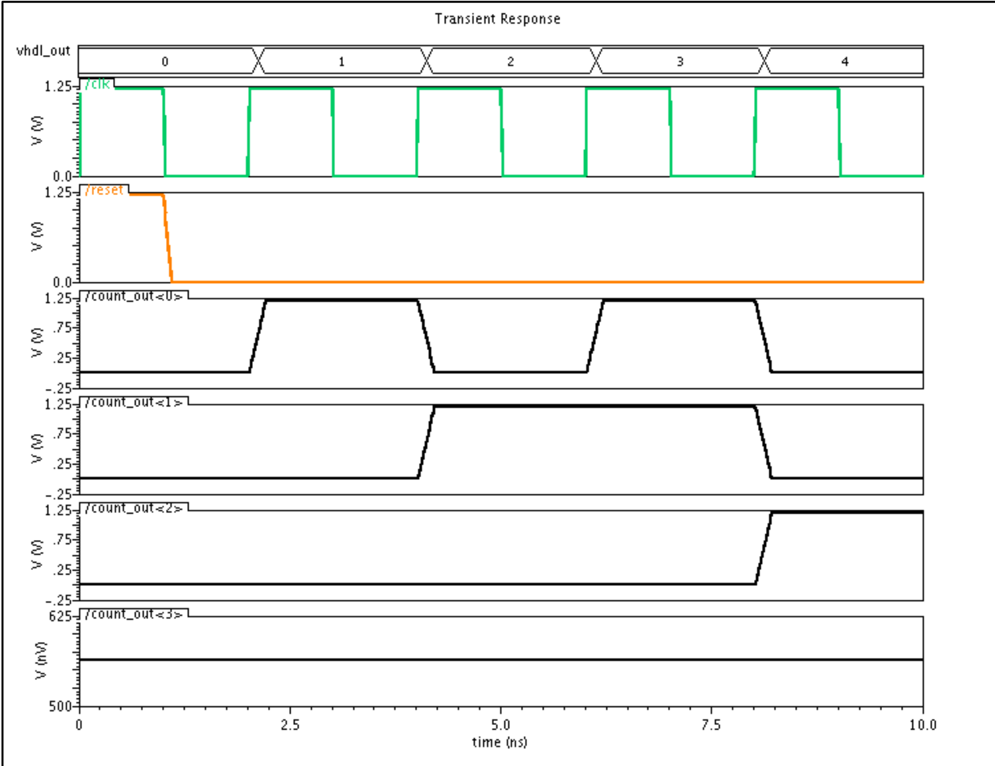
Bus Name

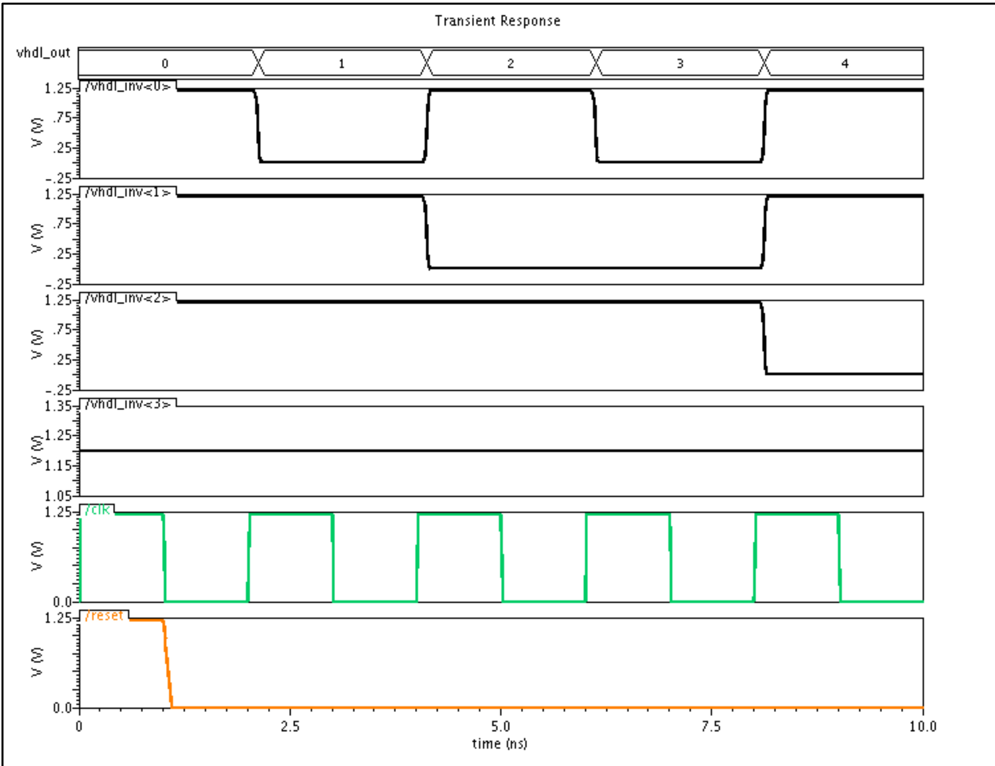
Radix Hex

Plot Mode Replace

OK Cancel Apply

Simulation results are as follows





For more details, please check the following videos:

http://youtu.be/QzHU-FyISIo

http://youtu.be/AN641gYyFj4

http://youtu.be/SLAEHGnrwuE

Conclusion

In this tutorial we have learned how to use AMS simulator to simulate digital Verilog codes with digital VHDL code and analog blocks. This tutorial assumes that, you have the AMS tool integrated with Cadence Virtuoso. For further information, please check the video notes of this tutorial in the reference page.

Appendix B

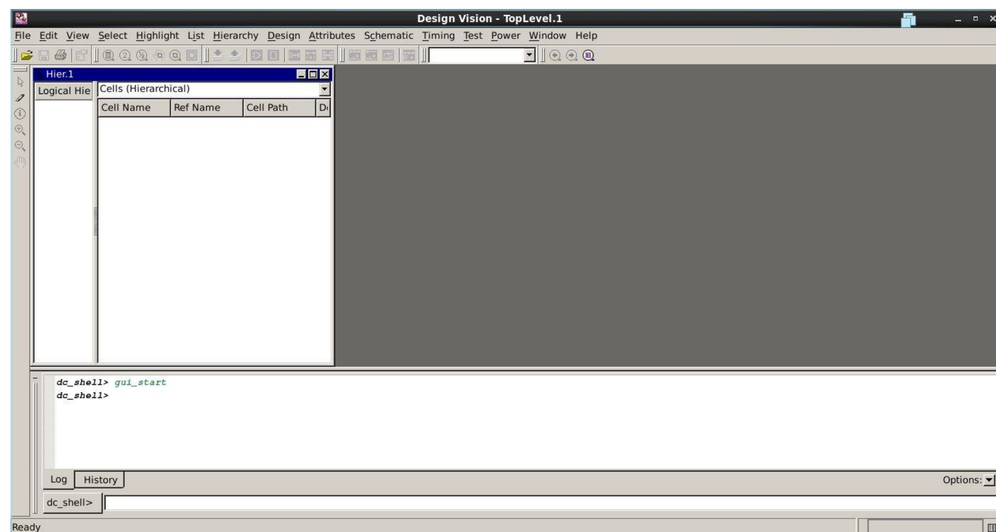
LOGIC SYNTHESIS

This tutorial presents the main steps to perform the logic synthesis of a digital Verilog 4-bit counter with the Synopsys Design Vision and Design Compiler tools. We divide this tutorial into two parts, part 1 and part 2. For part 1, we make all the steps using the Design Vision graphical environment. For part 2, we make the same steps using a TCL (Tool Command Language) script.

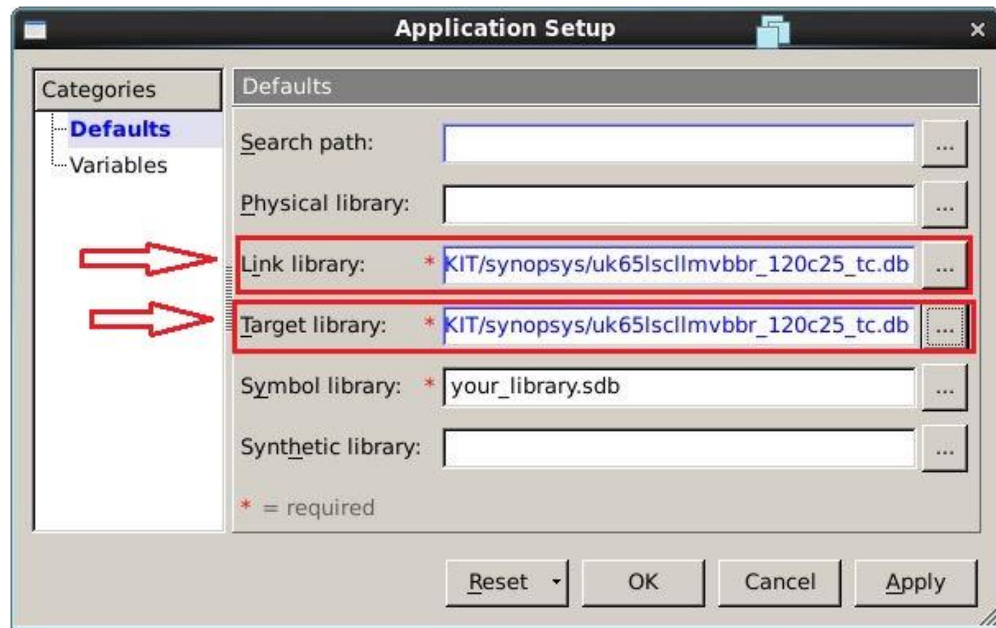
Part 1:

First step is to start the Design Vision graphical environment. In command window type:

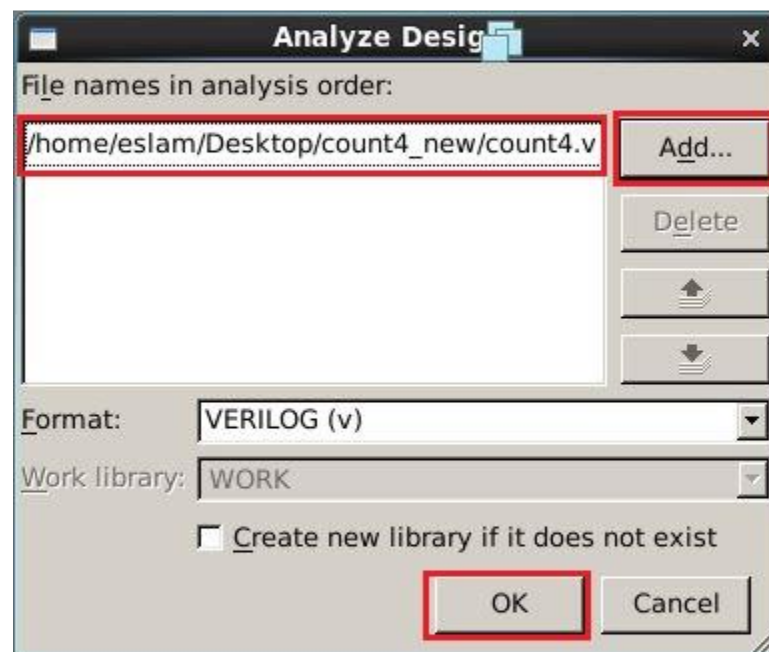
dc_shell and press enter then type *gui_start*. You should get the following menu:



Next step is to edit setup menu to add your target library and link library; the libraries to which your design will be mapped. To do this step choose **File-> Setup** and from the following menu remove default libraries and add yours.



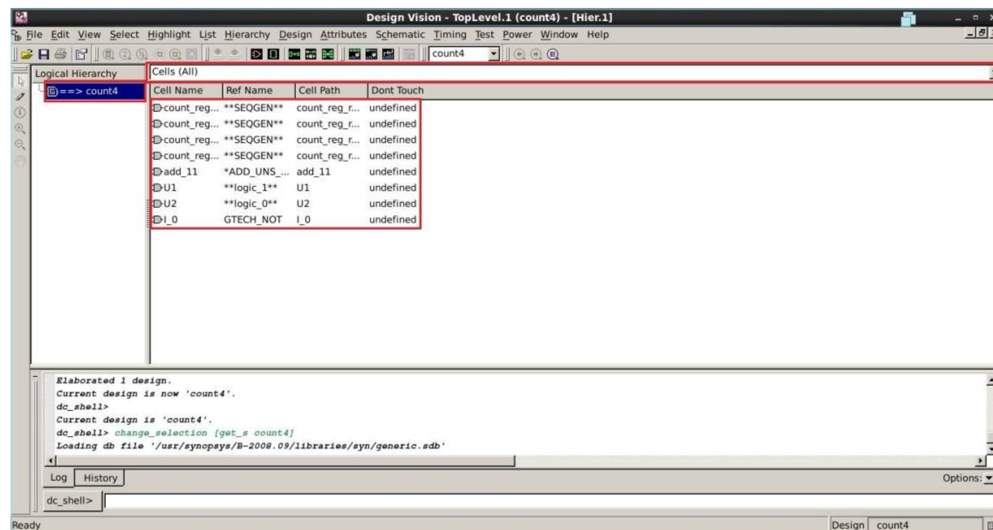
The following step is to analyze your Verilog code to check it and see if it is synthesizable. Choose **File-> Analyze** from the main menu and using the **Add** button select all your Verilog sources to be analyzed then press **OK**.

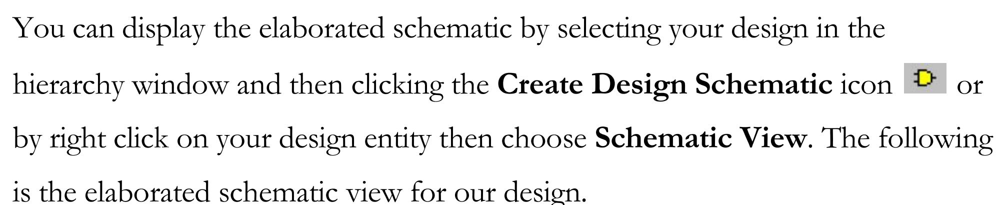


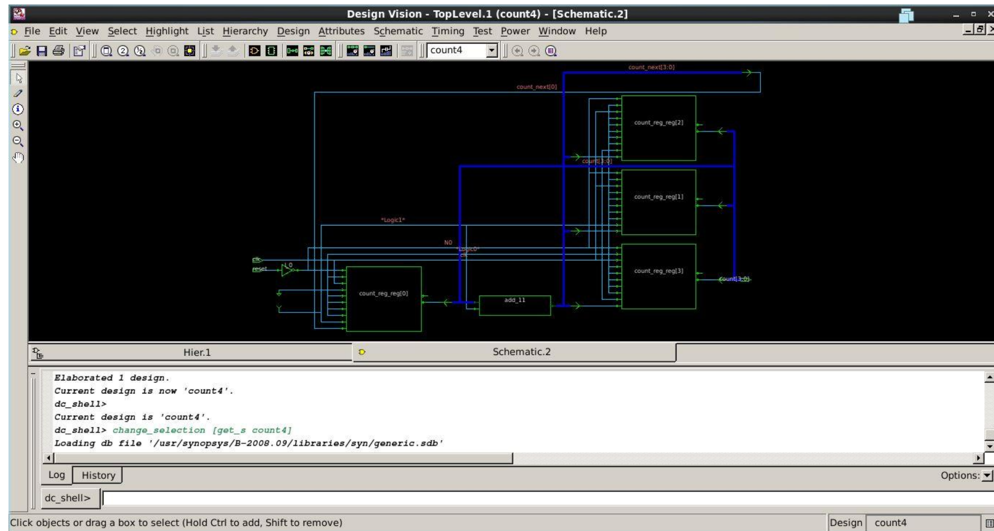
Next step is to elaborate your design. The elaboration phase performs a generic pre-synthesis of the analyzed model. It essentially identifies the registers that will be inferred.

To do this step choose **File -> Elaborate** from the main menu and from the elaboration menu select the top level file of your design and press **OK** as seen in the following figure.

After pressing OK in the elaboration menu, the main menu will be updated with your design which is count4 in this tutorial.

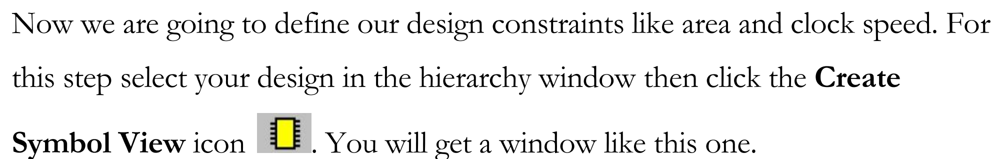


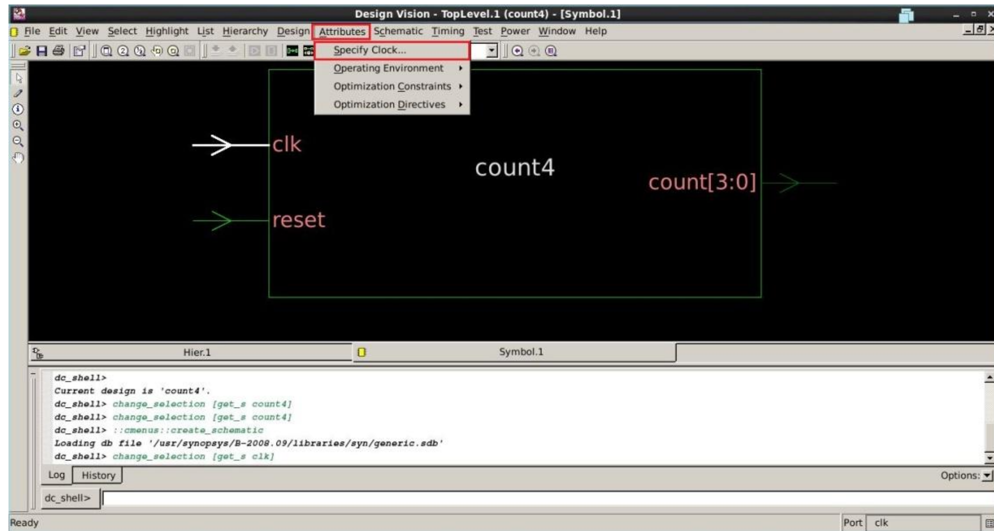
You can display the elaborated schematic by selecting your design in the hierarchy window and then clicking the **Create Design Schematic** icon  or by right click on your design entity then choose **Schematic View**. The following is the elaborated schematic view for our design.



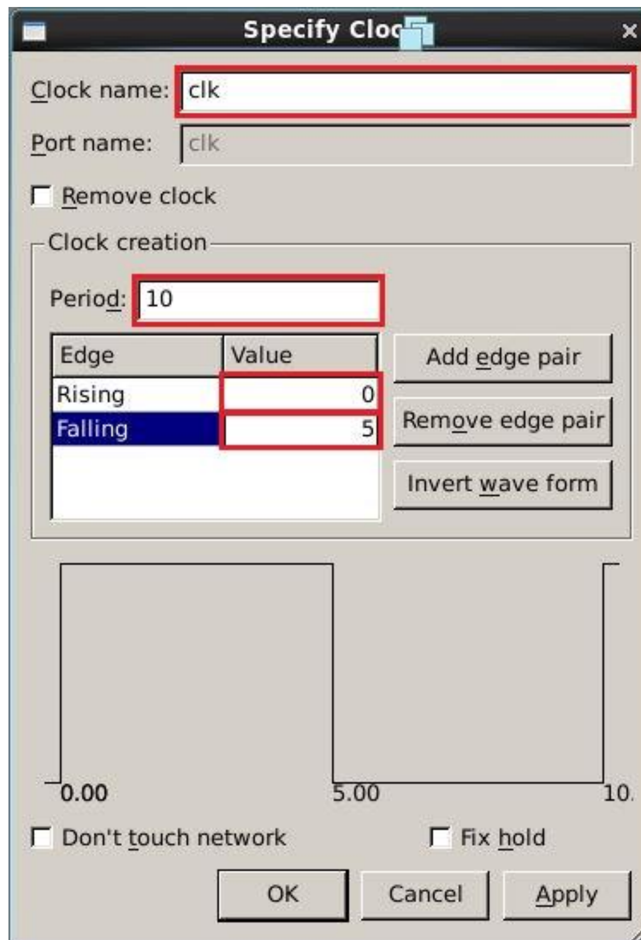
It is a good idea to save your design to this point. Choose **File -> Save as** and from the following window choose a name and click **Save.**

You can type this command in Design Vision to generate a report of the hierarchy of the design, *report_hierarchy*. You can also use this command to report all used cells and operating voltage of your design, *report_design*.

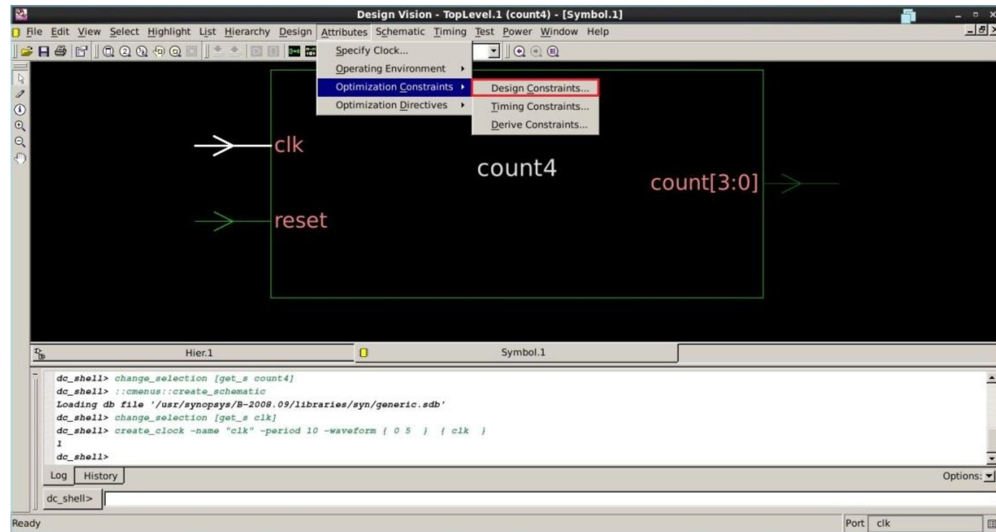
Now we are going to define our design constraints like area and clock speed. For this step select your design in the hierarchy window then click the **Create Symbol View** icon . You will get a window like this one.

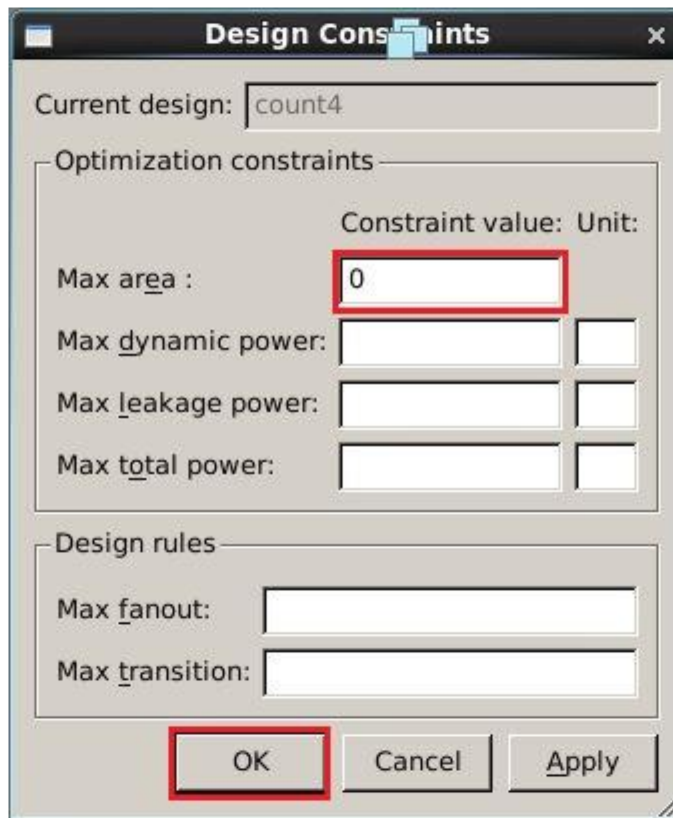


Now select your clock signal from the symbol view and choose **Attributes -> Specify Clock** to define the clock period and its duty cycle. For our example we choose a clock period of 10 ns and duty cycle of 50% like the following figure.

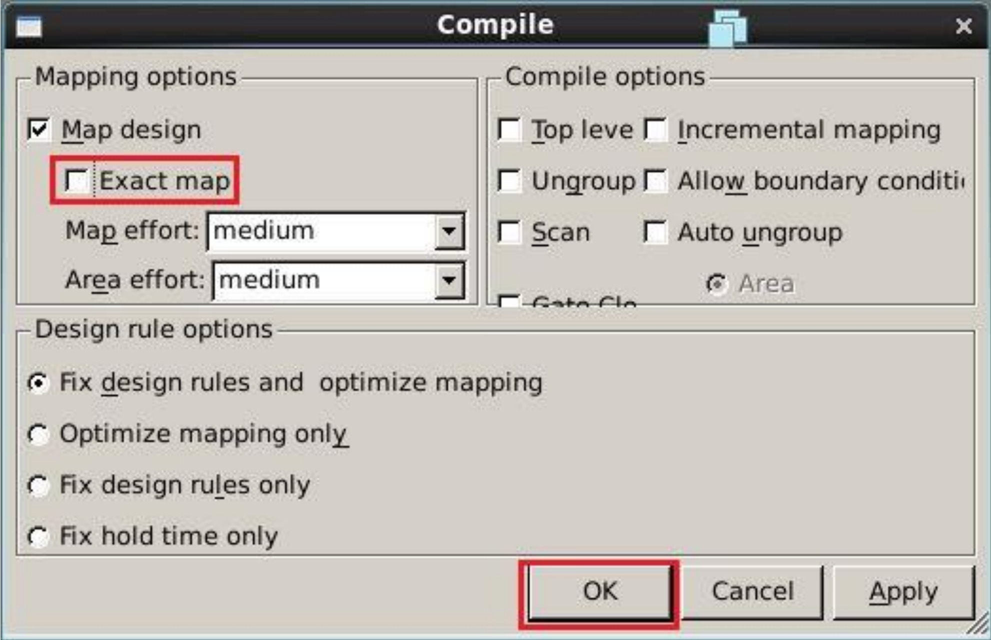


Next step is to define the constraints on design area by choosing **Attributes -> Optimization Constraints -> Design Constraints** from Design Vision main window. We choose area to be equal to zero to get the minimum area for our design.

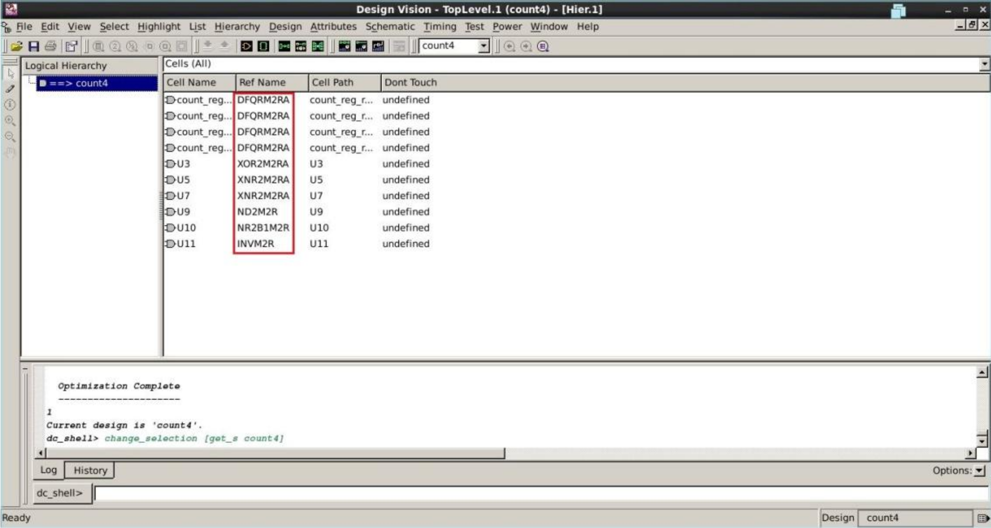




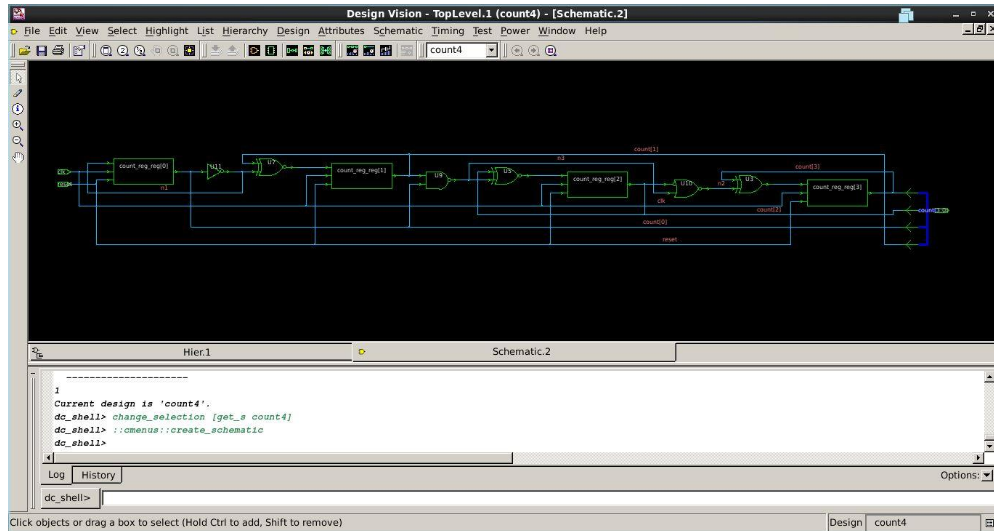
Next step is to compile your design. After this step your design will be mapped to real standard cells from your target technology library. From the main window of Design Vision, choose **Design -> Compile Design,** you will get a menu like the following figure, click **OK**.



As seen in the following figure, your design is now mapped to real standard cells.

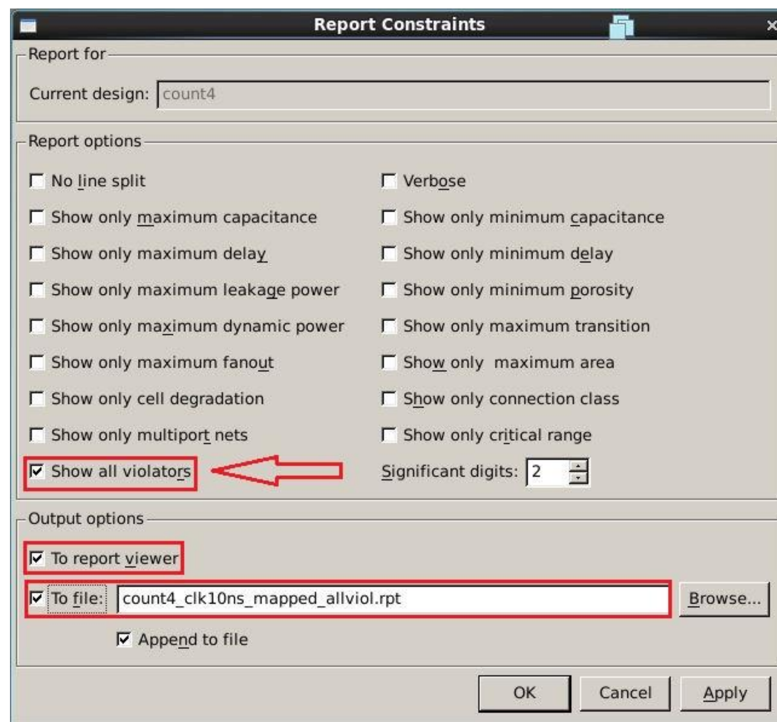


You can check the schematic view of your design now to see it in terms of standard cells.

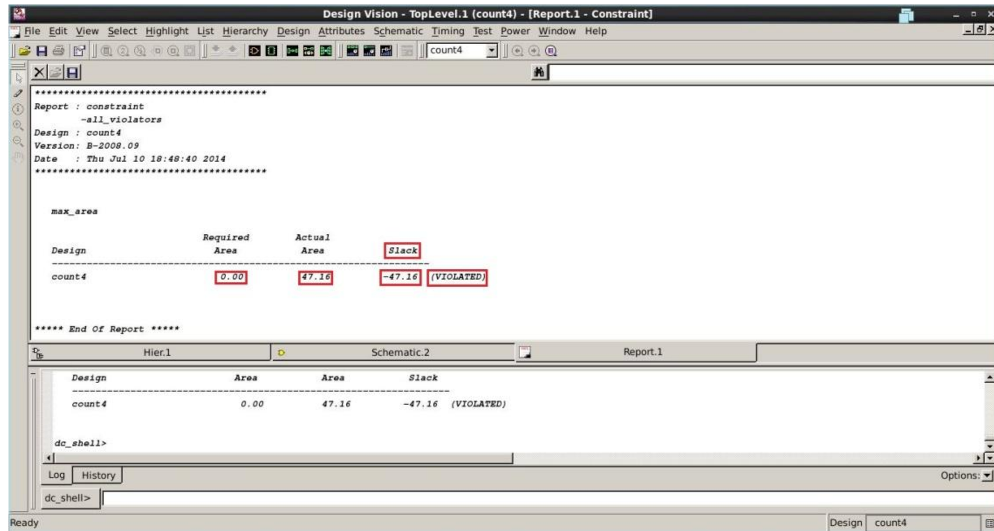


A good idea is to save your design now using File -> Save As and choose an appropriate name, for use we name it count4_clk10ns_mapped.ddc.

To report your design constraints to check if there are any violations on your design constraints, choose **Design -> Report Constraints**. Continue like the following figure.

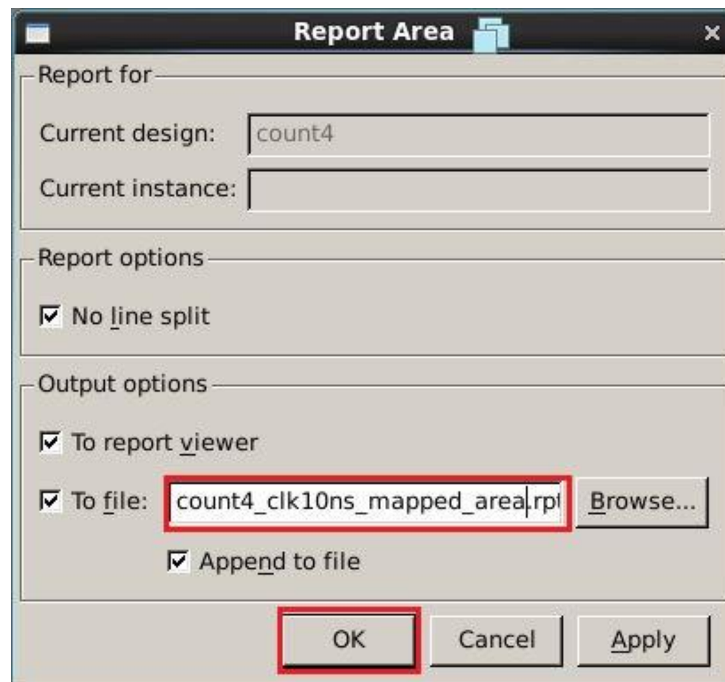


You will get a report like the following.

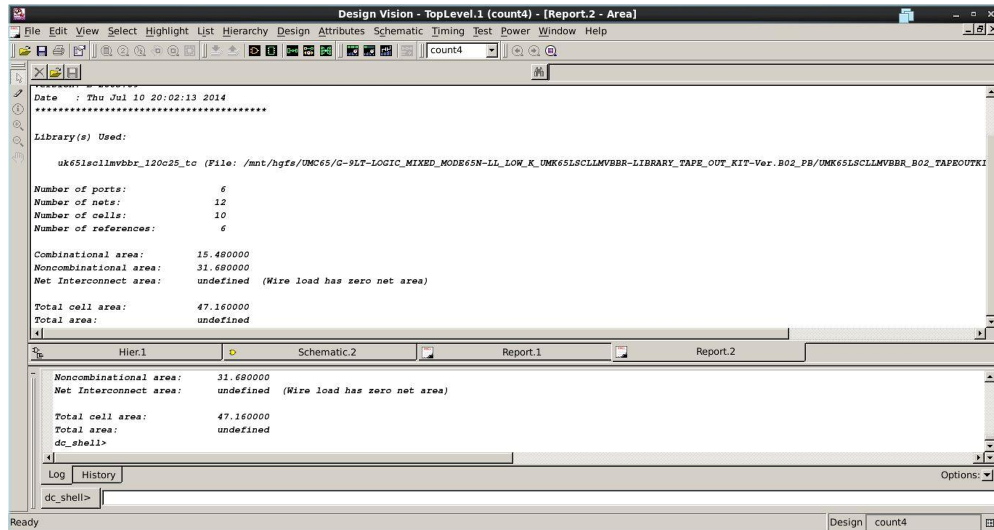


This report says that, area is violated and this is expected because we set the max area earlier to zero to achieve a minimum area, now this minimum area can be seen under the field of **Actual Area** in this report and it is 47.16 um^2 in our case. **Slack** equals to **Required Area** minus the **Actual Area** achieved by Design Vision, one would like always to get a positive slack. Positive slack means that, constraint is met but negative slack means constraint is violated.

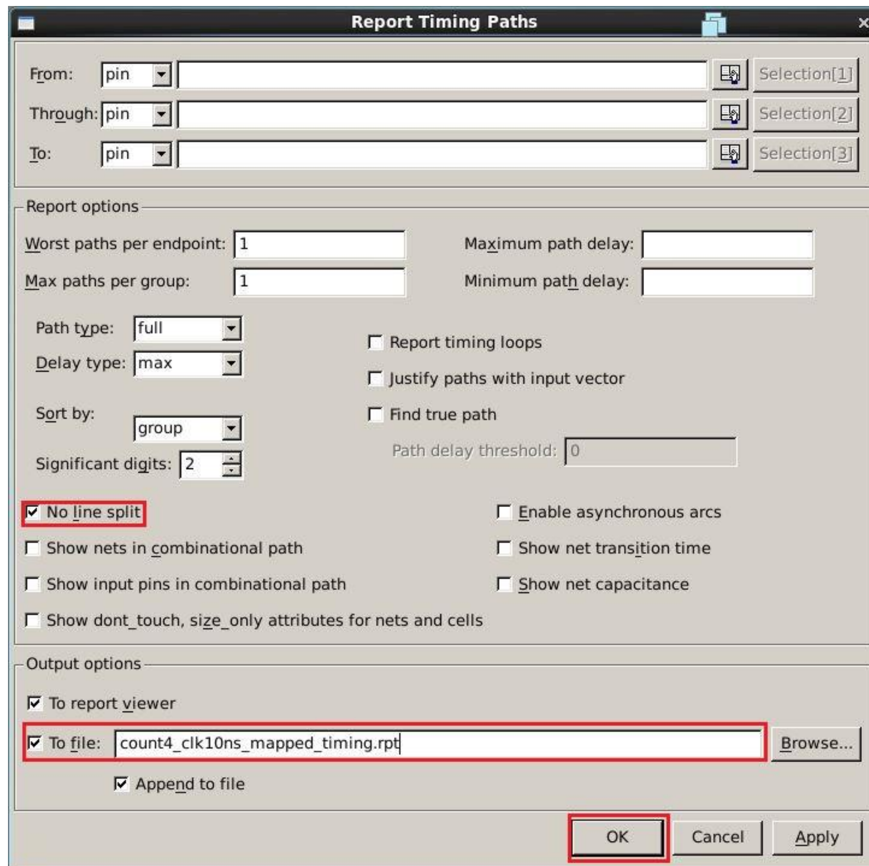
Choose **Design -> Report Area** to check the overall area of your design.



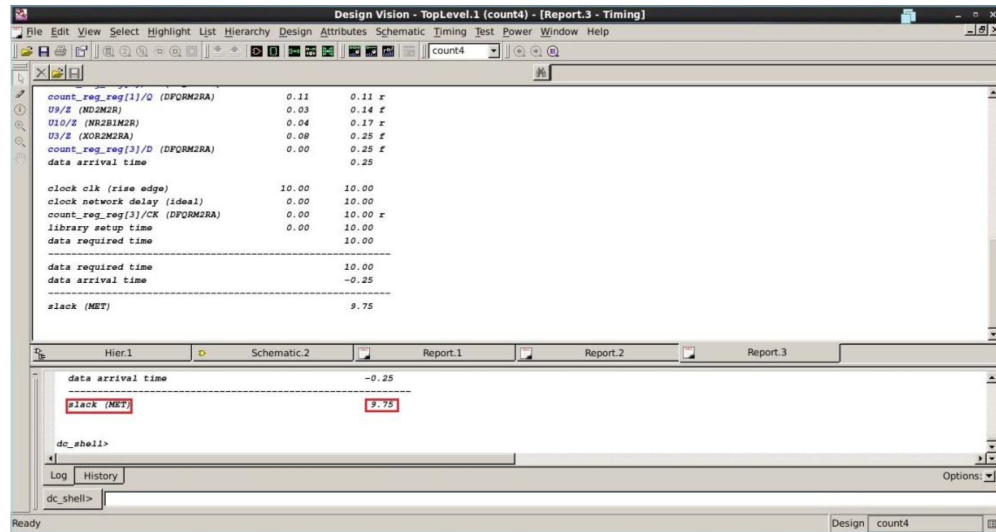
After clicking **OK** you will get a report for your design area like the following figure.



To check timing of your design, choose **Timing -> Report Timing Path.**



You will get a timing report like the following one; from this figure we notice that our timing slack is positive and equals to 9.75 ns i.e. timing constraint is met and we can reduce the required clock period by 9.75 ns. Now we know that min clock period for our design is, 10 ns - 9.75 ns = 0.25 ns.



A good way to visualize your timing paths and determine the critical path can be done using **Timing -> Path Slack.** Choose OK from the following menu.

Path Slack

From: pin Selection[1]
Through: pin Selection[2]
To: pin Selection[3]

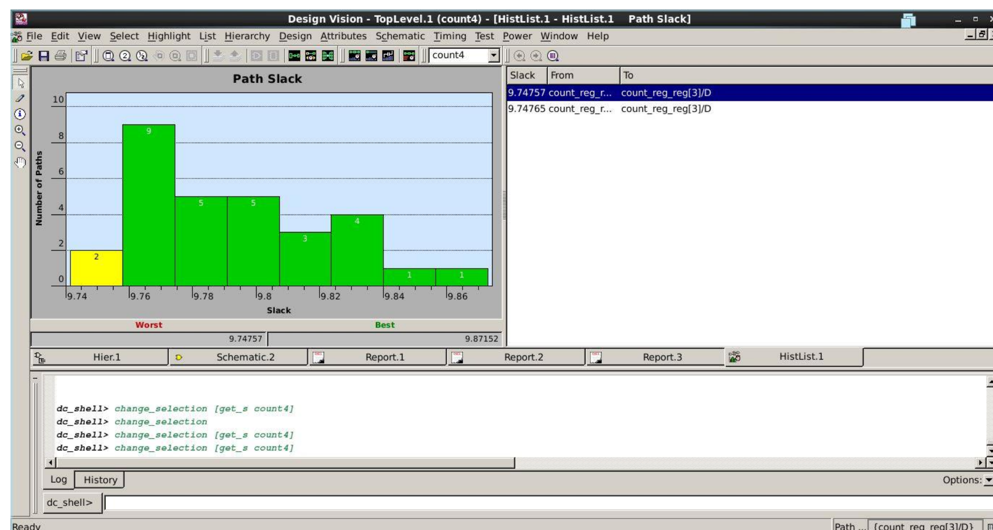
Nworst paths: 10 Max paths: 50
Group name: Delay type: max
Enable preset clear arcs Include hierarchical pins

Number of bins: 8
Value range per bin:
<= Slack <=
Lower bound strict Upper bound strict

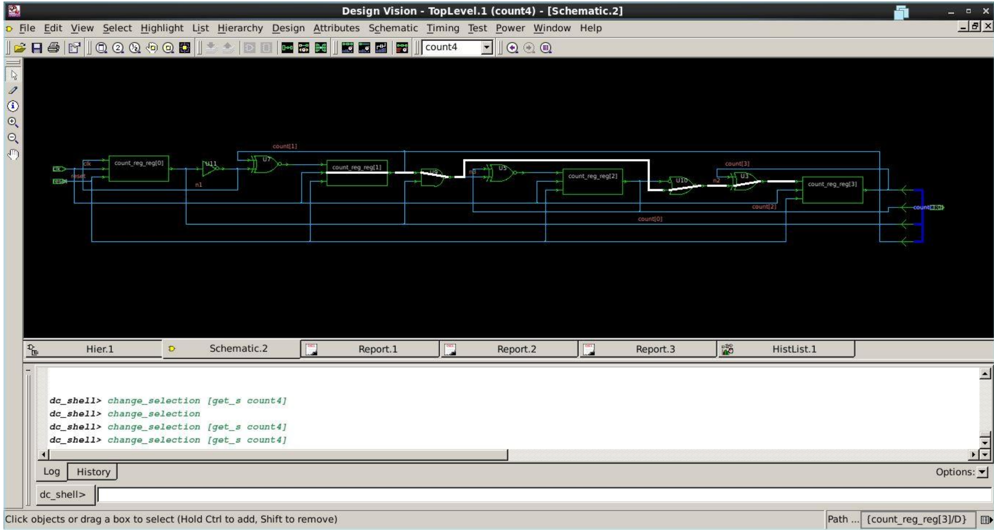
Histogram title: Path Slack
X-axis title: Slack
Y-axis title: Number of Paths

OK Cancel Apply

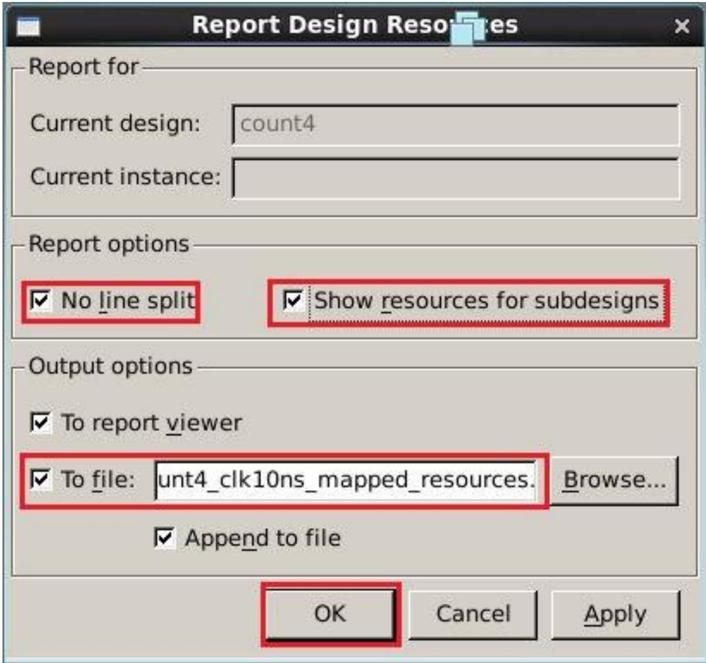
You will get a histogram for slacks of all timing paths in your design like the following one.



In this histogram, the path with the smallest slack is the worst (critical) path in your design. In our example, the worst paths have a slack of 9.74 ns. When selecting any path from this histogram you can see the equivalent path on your circuit by moving to your schematic view. The selected path above is shown in schematic as follows.



Other reports can be useful for you like resources report, you can get it from **Design -> Report Design Resources** as following.

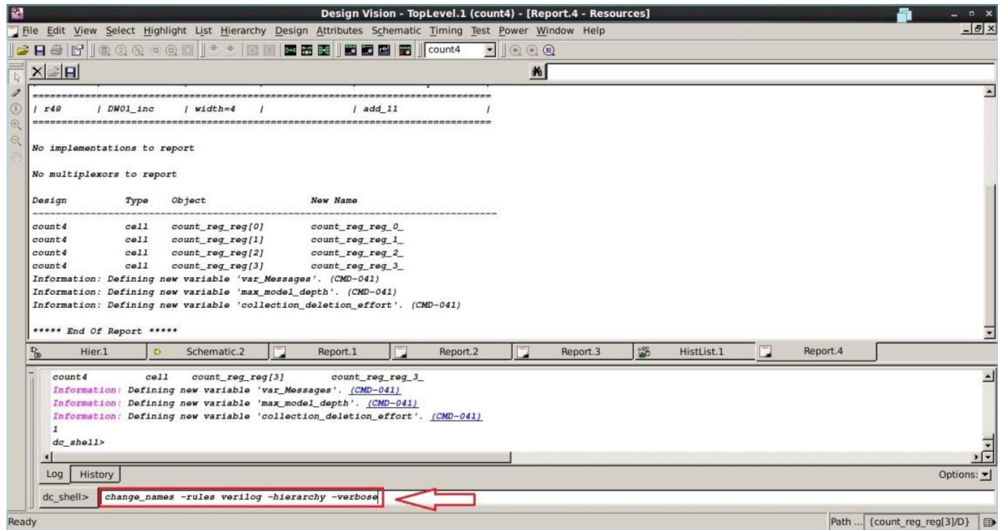


In the following steps, we will export the netlist file and all needed files to be used in the following design stages like automatic place and route using SOC Encounter.

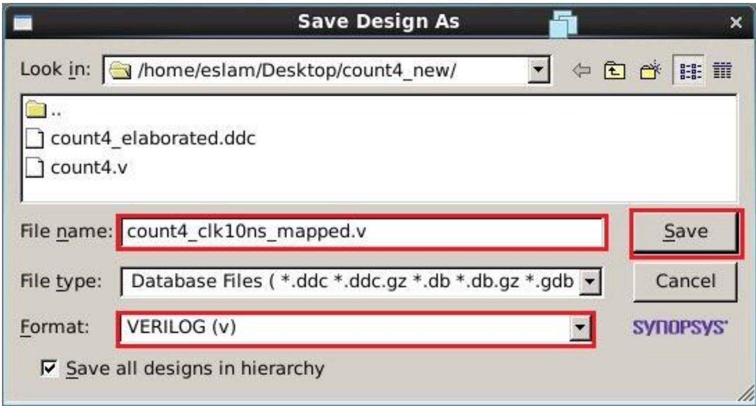
Before exporting the netlist file you have to type the following command in Design Compiler command line.

change_names -rules verilog -hierarchy -verbose

This command is to apply some verilog naming rules to your design before exporting the gate level netlist file.



Now, save your netlist verilog file using **File -> Save As** then type an appropriate name and change the format to verilog like the following figure.



The output gate level netlist file from this step is as follows.

```
module count4 ( clk, reset, count );
output [3:0] count;
input clk, reset;
wire   n1, n2, n3;
wire   [3:0] count_next;

DFQRM2RA count_reg_reg_0_ ( .D(n1), .CK(clk), .RB(reset), .Q(count[0]) );
DFQRM2RA count_reg_reg_1_ ( .D(count_next[1]), .CK(clk), .RB(reset), .Q(
        count[1]) );
DFQRM2RA count_reg_reg_2_ ( .D(count_next[2]), .CK(clk), .RB(reset), .Q(
        count[2]) );
DFQRM2RA count_reg_reg_3_ ( .D(count_next[3]), .CK(clk), .RB(reset), .Q(
        count[3]) );
XOR2M2RA U3 ( .A(count[3]), .B(n2), .Z(count_next[3]) );
XNR2M2RA U5 ( .A(count[2]), .B(n3), .Z(count_next[2]) );
XNR2M2RA U7 ( .A(count[1]), .B(n1), .Z(count_next[1]) );
ND2M2R U9 ( .A(count[1]), .B(count[0]), .Z(n3) );
NR2B1M2R U10 ( .NA(count[2]), .B(n3), .Z(n2) );
INVM2R U11 ( .A(count[0]), .Z(n1) );
endmodule
```

This file will be used in appendix C and is also used as an input to the automatic place and route tool, SOC Encounter to generate the layout for your design.

Next step is the post synthesis timing data extraction. In this step we extract the standard delay format file which can be used in post synthesis simulation. To get this file, type the following command in the command line of Design Vision.

write_sdf -version 2.1 coun4_clk10ns_mapped_vlog.sdf

You can fine this file and all report files you've generated earlier in your working directory.

The following command is to generate a file includes all your design constraints in TCL format and will be used in standard cell placement and routing (SOC Encounter).

write_sdc -nosplit count4_clk110ns_mapped.sdc

For more details, please check the following video:

http://youtu.be/pD85Hnsi2cc

Part 2:

The same steps of part 1 can be done by using this TCL script. The code is self documented, please read it carefully and edit all needed fields according to your design.

```
# Design Setup (change library files according to your technology)
set link_library /home/eslam/Desktop/synopsys/uk65lscllmvbbr_120c25_tc.db
set target_library /home/eslam/Desktop/synopsys/uk65lscllmvbbr_120c25_tc.db
#Analyze
analyze -format verilog {/home/eslam/Desktop/synopsys/count4.v}
#elaborate (count4 is the name of the top level module)
elaborate count4 -architecture verilog -library DEFAULT -update
write -hierarchy -format ddc -output /home/eslam/Desktop/synopsys/count4.ddc
#timing & area constraints (clk is the clock name in my verilog file-edit according to
your design (ns))
create_clock -name "clk" -period 10 -waveform { 0 5  } { clk  }
set_max_area 0
#compile design
compile
#export design (reports and netlist and timing files)
write -hierarchy -format ddc -output
/home/eslam/Desktop/synopsys/count4_clk10ns_mapped.ddc
#generate design reports
report_constraint -nosplit -all_violators > /home/eslam/Desktop/synopsys/allviol.rpt
report_area > /home/eslam/Desktop/synopsys/area.rpt
report_timing > /home/eslam/Desktop/synopsys/timing.rpt
report_resources -nosplit -hierarchy > /home/eslam/Desktop/synopsys/resources.rpt
report_reference -nosplit -hierarchy > /home/eslam/Desktop/synopsys/references.rpt
report_hierarchy > hierarchy.rpt
report_design > design.rpt
#add some verilog naming rules before exporting the gate level netlist file
change_names -rules verilog -hierarchy -verbose
write -hierarchy -format verilog -output
/home/eslam/Desktop/synopsys/count4_clk10ns_mapped.v
write_sdf -version 2.1 count4_mapped_vlog.sdf
write_sdc -nosplit count4_vlog.sdc
puts "Finished"
```

For more details, please check the following video:
http://youtu.be/gH6ZMh3IQBI

Conclusion

Using this tutorial we've learned how to convert our functional verilog code into a gate level netlist file, i.e. a file containing standard cells from our technology library and learned how to define design constraints like timing and area constraints and how to analyze timing paths using a timing histogram and finally how to generate all needed reports and files which can be used in next design steps like automatic place and route.

Appendix C

IMPORTING SYNTHESIZED DESIGN INTO CADENCE COMPOSER SCHEMATIC VIEW

In this tutorial we are going to simulate a digital 4-bit Verilog counter to test its function first without any timing analysis then we will use the synthesized version (output of Design Compiler) of this counter and import it into CADENCE Composer as a schematic view containing all standard cells needed for this counter from our technology library, so in this part we will simulate the 4-bit counter in two levels of design:

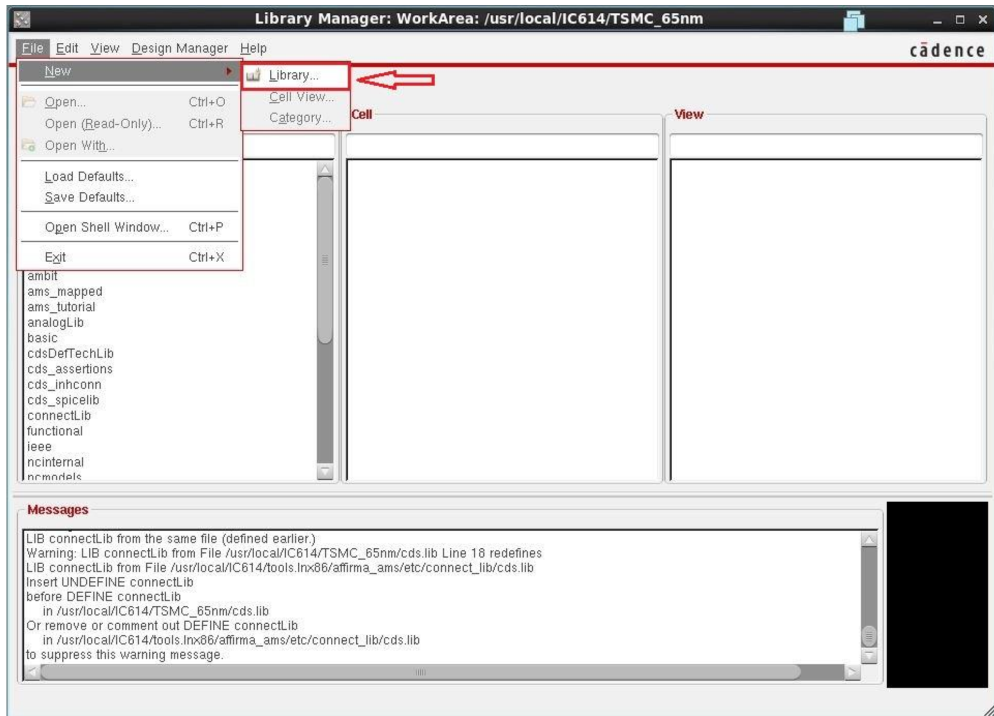
1. Pre-synthesis functional Verilog simulation using AMS simulator
2. Post-synthesis transistor level simulation using Spectre simulator

1. Pre-synthesis functional Verilog simulation using AMS simulator

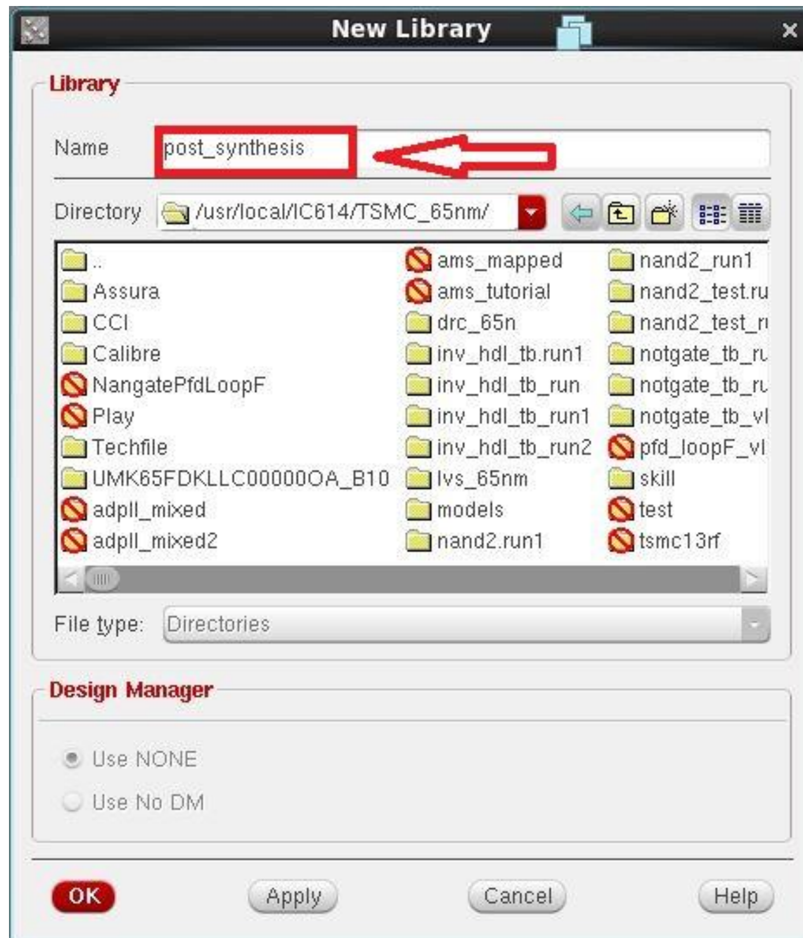
The following is the digital 4-bit Verilog code used for this tutorial:

```
module count4(input clk,reset,output[3:0] count);
reg[3:0] count_reg;
wire[3:0] count_next;

//state register
always@(posedge clk,negedge reset)
if(~reset)
count_reg <= 0;
else
count_reg <= count_next;
//next state logic
assign count_next = count_reg+1;
//output logic
assign count = count_reg;
endmodule
```

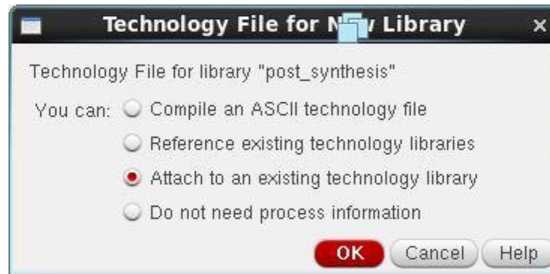
For this tutorial, we will make new work library and name it post_synthesis as following:



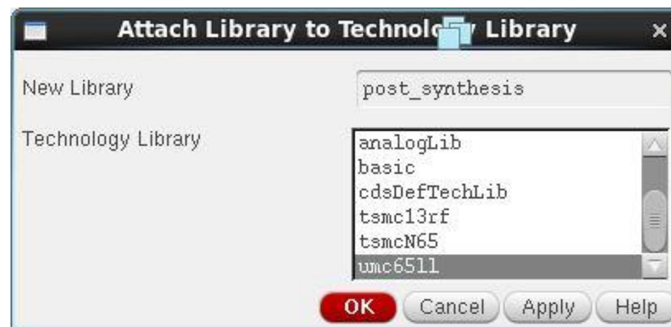
Library Manager: WorkArea: /usr/local/IC614/TSMC_65nm
File Edit View Design Manager Help
cādence
New
Library...
Cell View...
Category...
Open... Ctrl+O
Open (Read-Only)... Ctrl+R
Open With...
Load Defaults...
Save Defaults...
Open Shell Window... Ctrl+P
Exit Ctrl+X
Cell
View
ambit
ams_mapped
ams_tutorial
analogLib
basic
cdsDefTechLib
cds_assertions
cds_inhconn
cds_spicelib
connectLib
functional
ieee
ncinternal
Messages
LIB connectLib from the same file (defined earlier.)
Warning: LIB connectLib from File /usr/local/IC614/TSMC_65nm/cds.lib Line 18 redefines
LIB connectLib from File /usr/local/IC614/tools.lnx86/affirma_ams/etc/connect_lib/cds.lib
Insert UNDEFINE connectLib
before DEFINE connectLib
in /usr/local/IC614/TSMC_65nm/cds.lib
Or remove or comment out DEFINE connectLib
in /usr/local/IC614/tools.lnx86/affirma_ams/etc/connect_lib/cds.lib
to suppress this warning message.



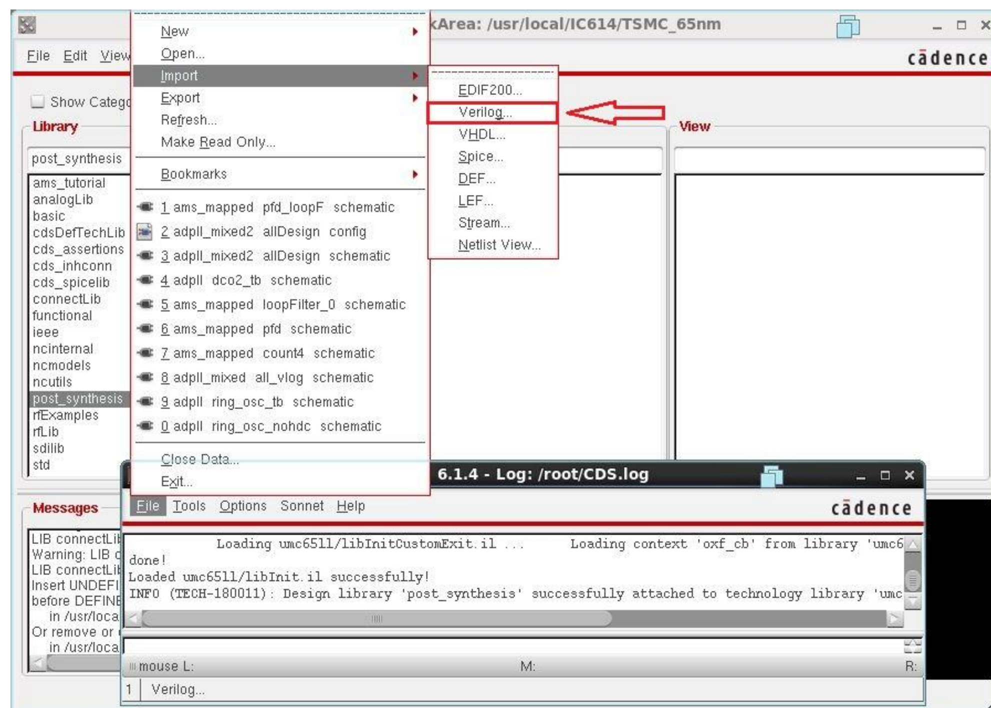
New Library
Library
Name
post_synthesis
Directory
/usr/local/IC614/TSMC_65nm/
..
Assura
CCI
Calibre
NangatePfdLoopF
Play
Techfile
UMK65FDKLLC00000OA_B10
adpll_mixed
adpll_mixed2
ams_mapped
ams_tutorial
drc_65n
inv_hdl_tb.run1
inv_hdl_tb_run
inv_hdl_tb_run1
inv_hdl_tb_run2
lvs_65nm
models
nand2.run1
nand2_run1
skill
test
tsmc13rf
File type:
Directories
Design Manager
Use NONE
Use No DM
OK
Apply
Cancel
Help

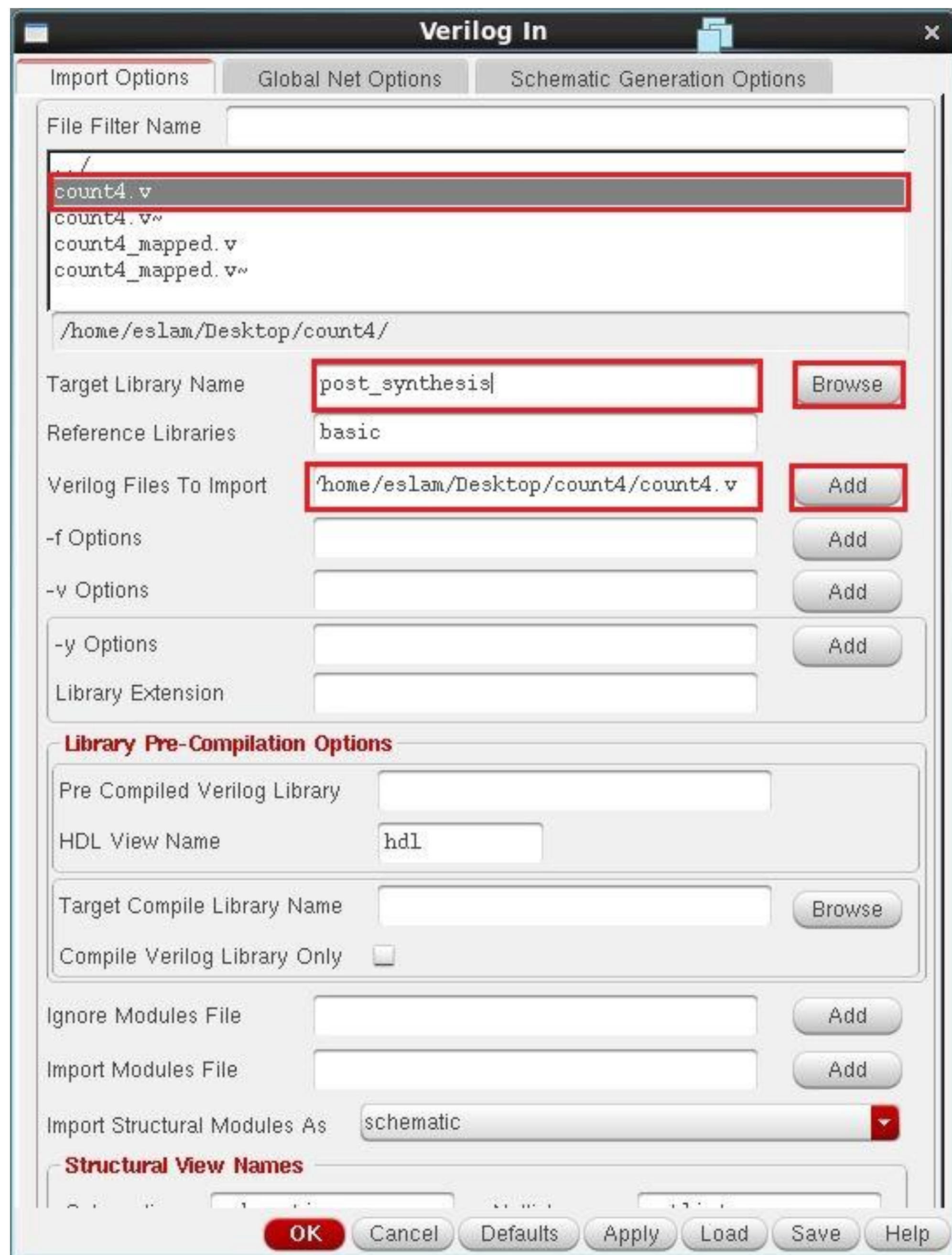


We should attach this work library to our technology library which is **umc65ll** in our tutorial.

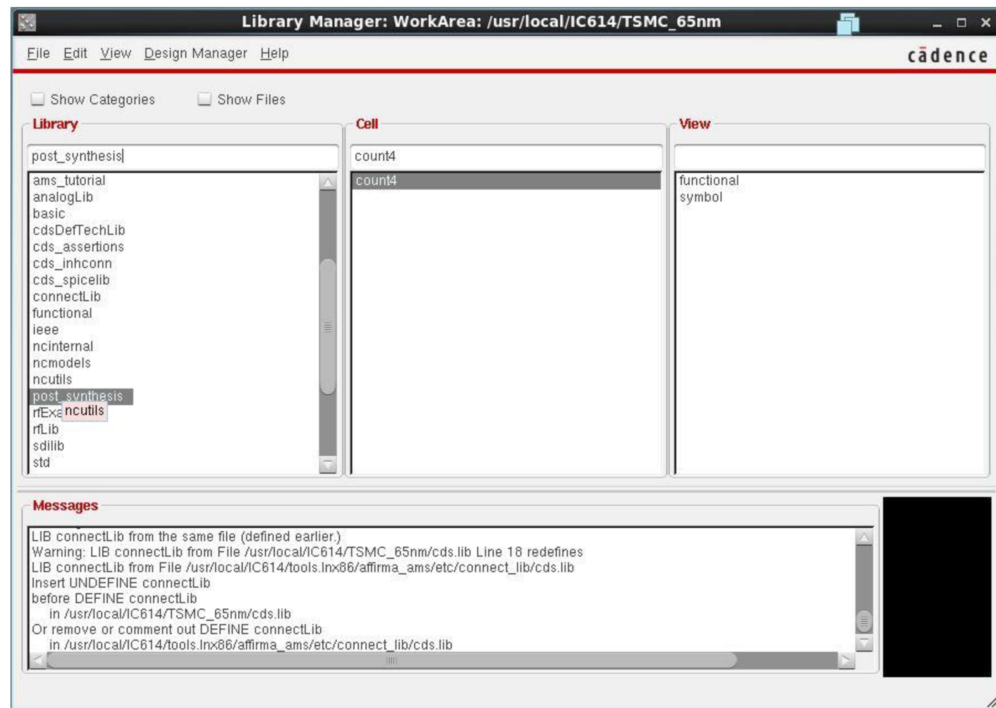


We make similar steps to those done in appendix A to import this Verilog code into cadence as following:

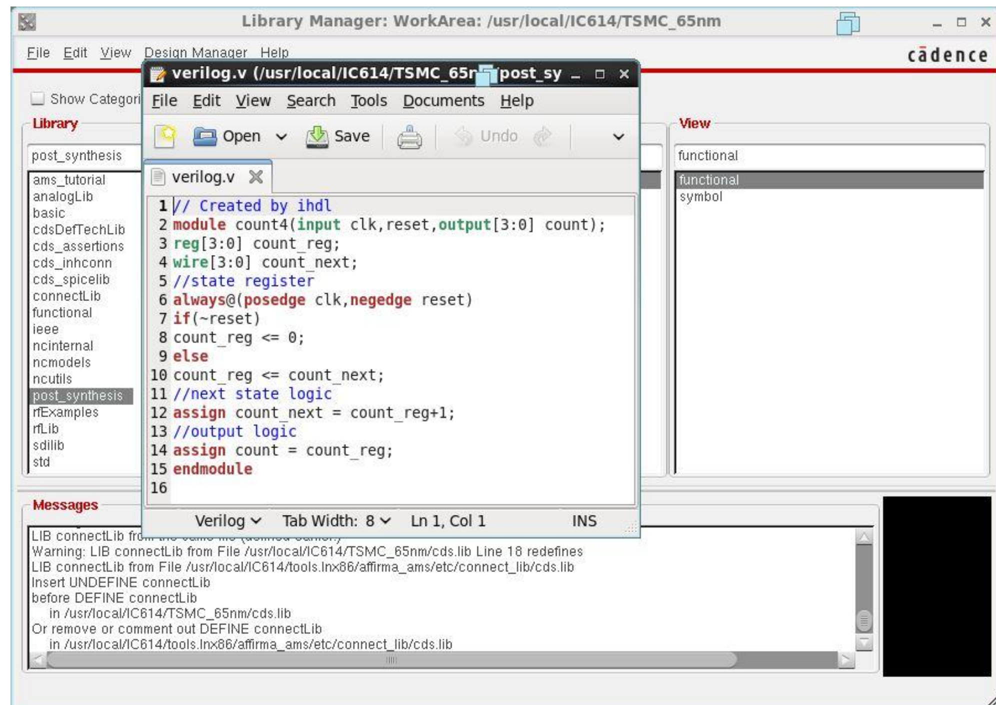




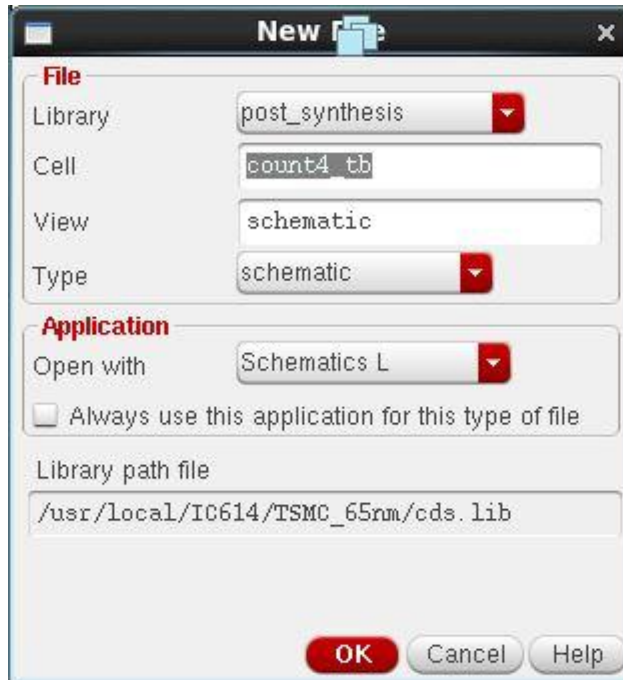
After ending this process, you will get a functional and symbol view for your counter.

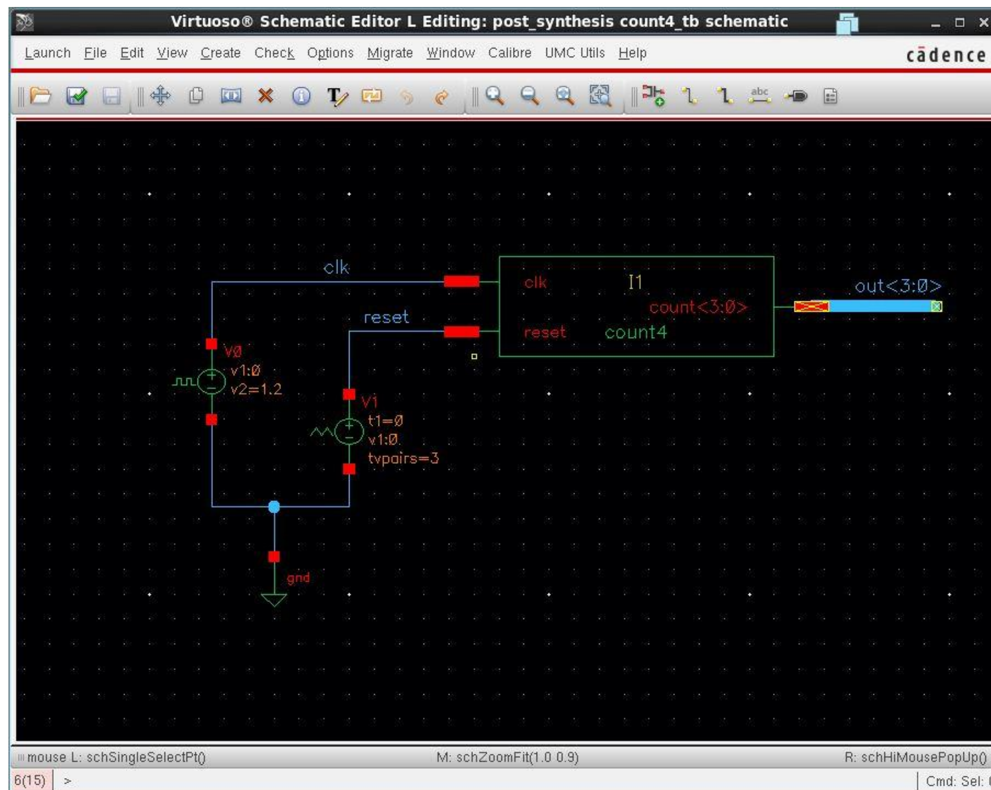


Double click on functional view to check your code.

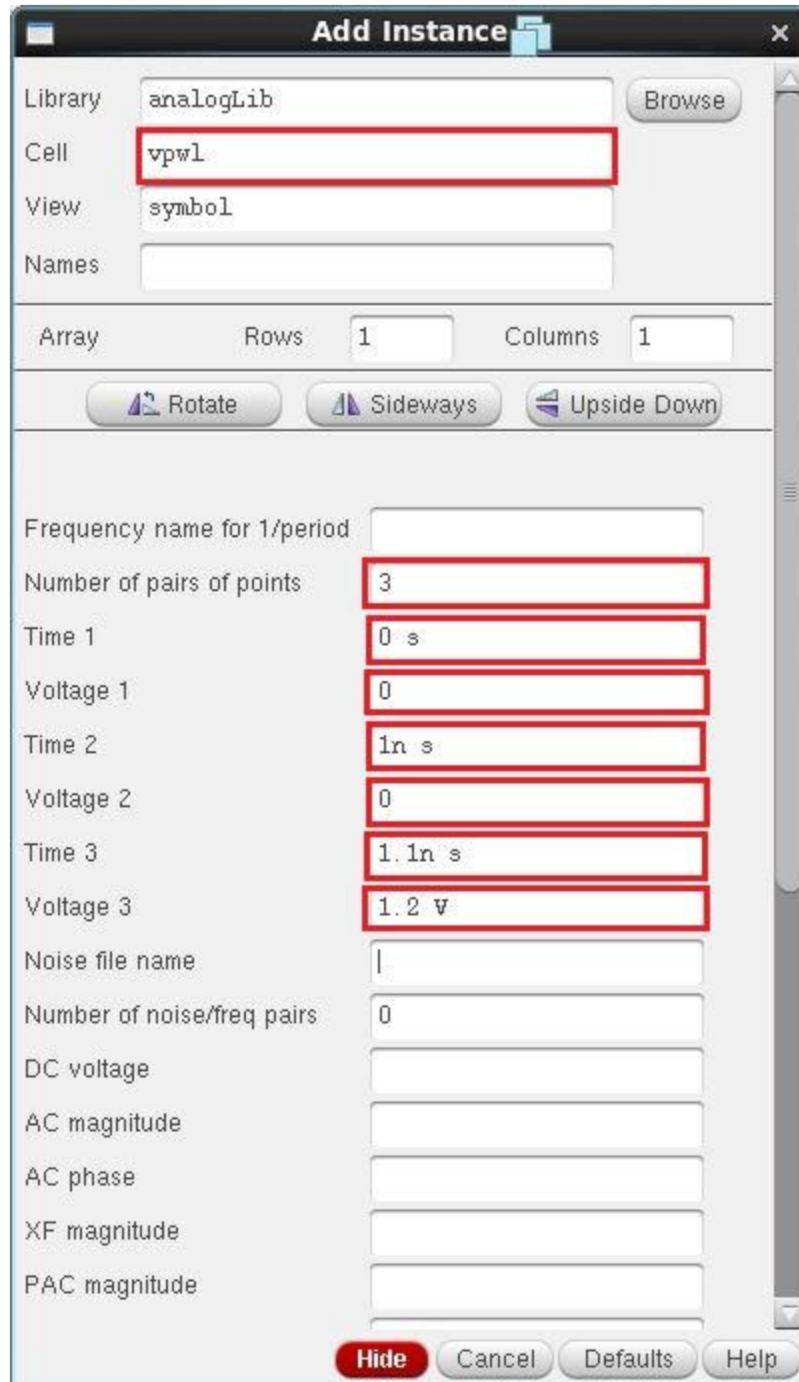


Now, it's time to make a test bench for this code to generate simulations results for it.





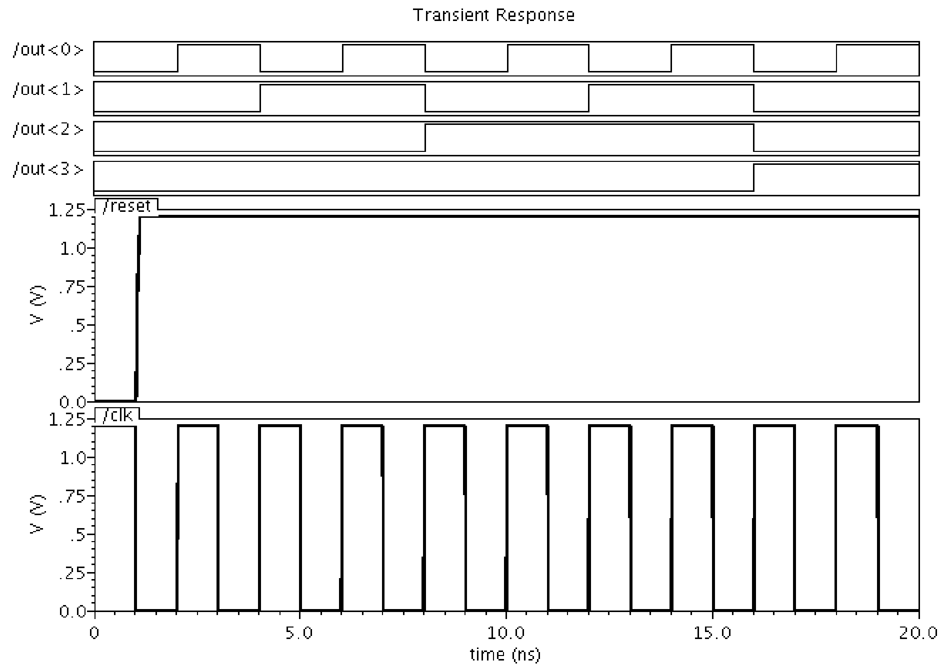
For this code we use an active low reset signal, this is how to get this reset signal using library sources:



Use similar steps to that in appendix A to create new config view for your test bench and don't forget to change the simulator to AMS and edit the connection rules.

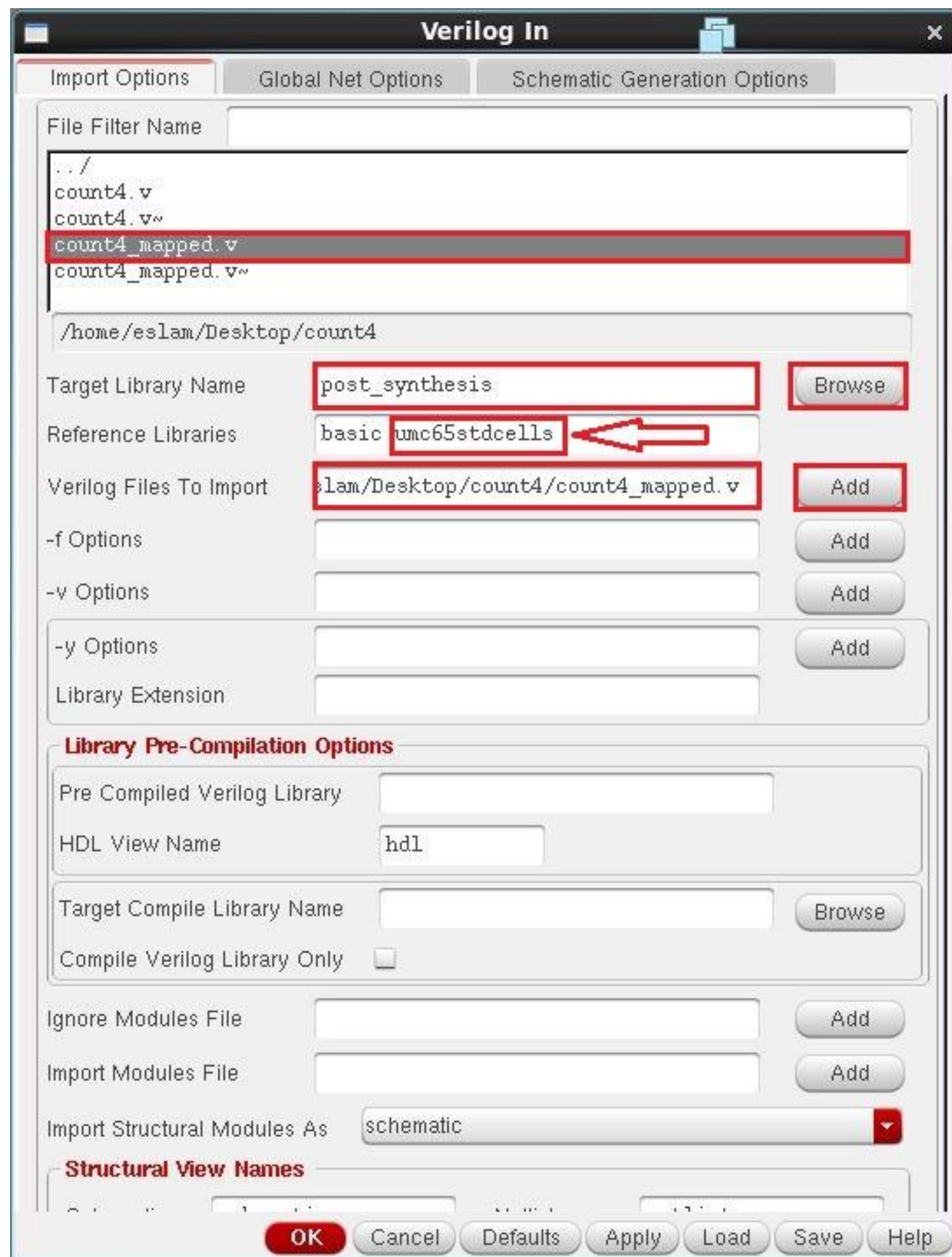
You must start the simulation from this config view as we noticed in appendix A.

The simulation results for this Verilog 4-bit counter are in the following figure:



2. Post-synthesis transistor level simulation using Spectre simulator

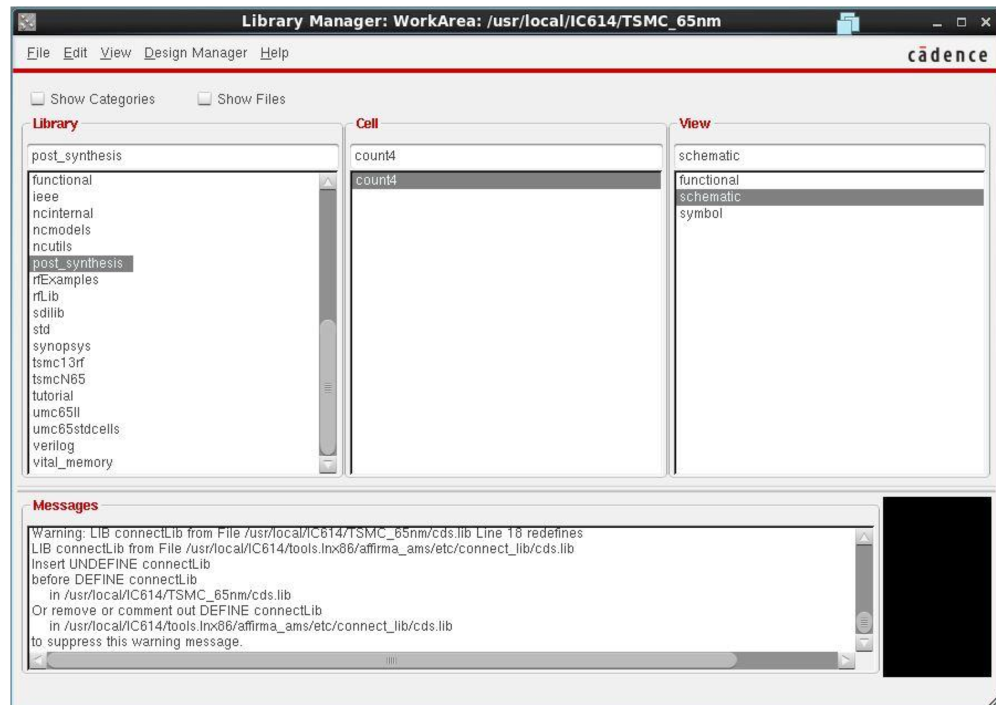
Now, we are going to import our synthesized Verilog code (standard cells description of our counter – output of Design Compiler) into CADENCE composer and simulate it in the transistor level using Spectre simulator. We use similar steps as we did to import the functional Verilog code but the only different part in import menu is that, you must specify your standard cells library (for us, it is **umc65stdcells)** as shown in the following figure :



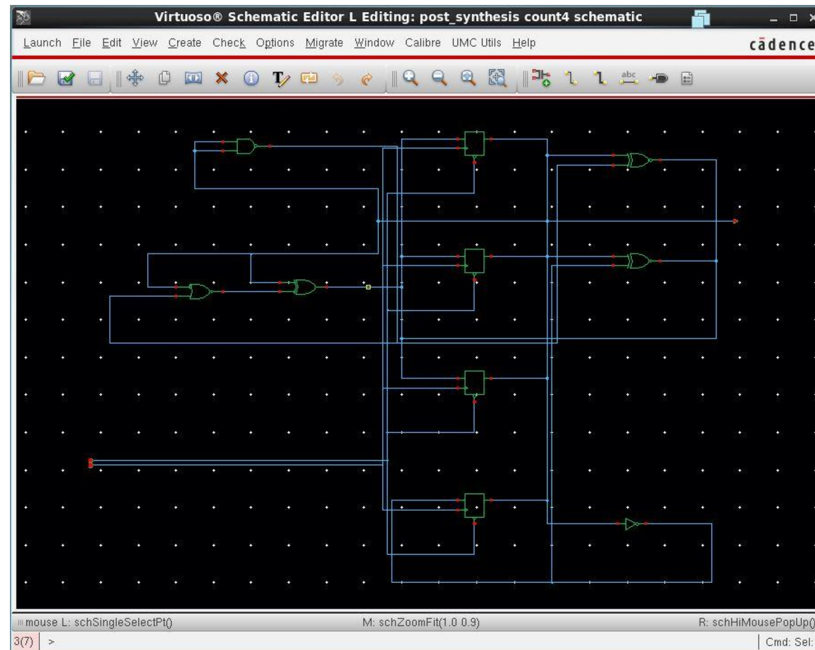
Transistors used in each standard cell in **umc65stdcells** are taken from the technology library we've included earlier, the **umc65ll.** You can have a look on the mapped code of this counter in the following figure:

```
module count4 ( clk, reset, count );
output [3:0] count;
input clk, reset;
wire   n1, n2, n3;
wire   [3:0] count_next;

DFQRM2RA count_reg_reg_0_ ( .D(n1), .CK(clk), .RB(reset), .Q(count[0]) );
DFQRM2RA count_reg_reg_1_ ( .D(count_next[1]), .CK(clk), .RB(reset), .Q(
        count[1]) );
DFQRM2RA count_reg_reg_2_ ( .D(count_next[2]), .CK(clk), .RB(reset), .Q(
        count[2]) );
DFQRM2RA count_reg_reg_3_ ( .D(count_next[3]), .CK(clk), .RB(reset), .Q(
        count[3]) );
XOR2M2RA U3 ( .A(count[3]), .B(n2), .Z(count_next[3]) );
XNR2M2RA U5 ( .A(count[2]), .B(n3), .Z(count_next[2]) );
XNR2M2RA U7 ( .A(count[1]), .B(n1), .Z(count_next[1]) );
ND2M2R U9 ( .A(count[1]), .B(count[0]), .Z(n3) );
NR2B1M2R U10 ( .NA(count[2]), .B(n3), .Z(n2) );
INVM2R U11 ( .A(count[0]), .Z(n1) );
endmodule
```

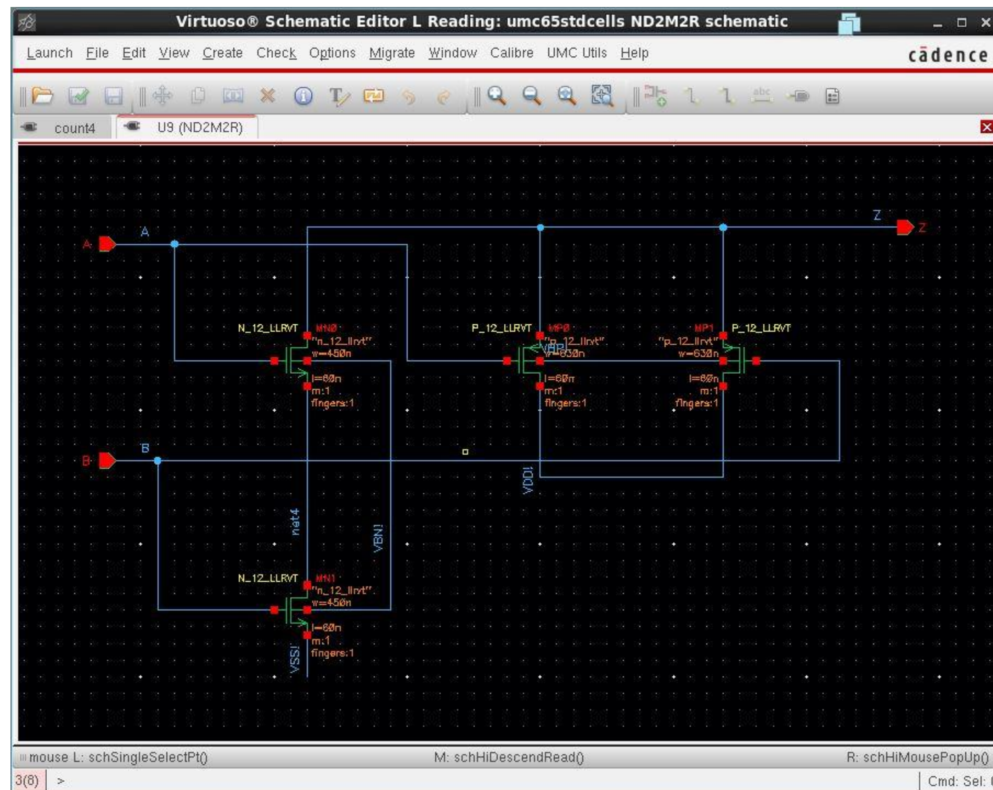
After ending this process successfully, there will be a schematic view added to your cell views of the counter.

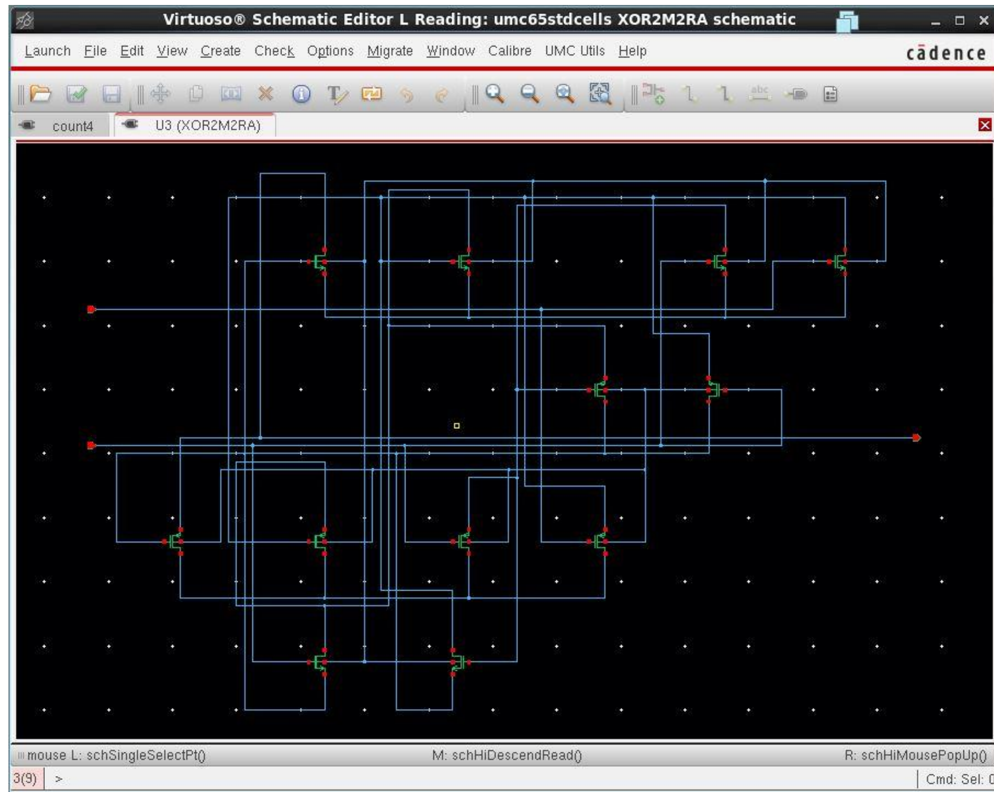


Double click on this schematic view to check the internal structure of your counter in terms of your standard cells.



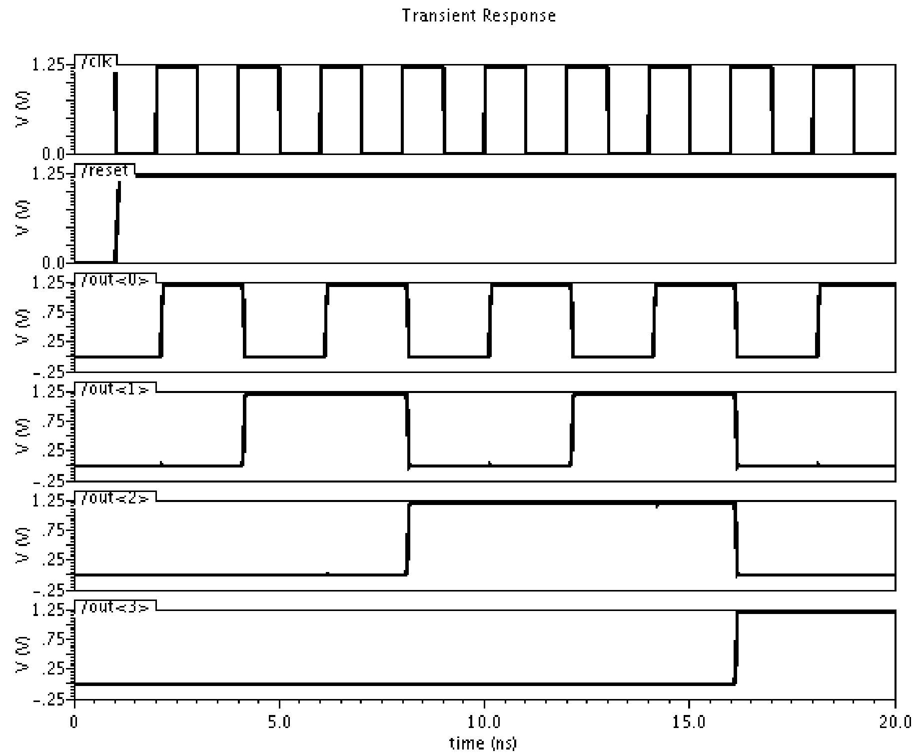
You can also check the internal structure of each standard cell in your counter by pressing **shift+E** on this cell. The following are the internal structures of two standard cells used in this counter:





Now, it is time to test this schematic. An important notice here is that; don't forget to connect power nodes (VDD and GND) and bulk nodes of transistors (VBN and VBP). We did not make this step when simulating the functional Verilog code because there were no transistors in this schematic and we were just simulating the code.

The simulation results for the transistor level schematic of the counter are in the following figure. One can easily notice, the same output as the functional Verilog code is achieved.



For more details please check this video:

http://youtu.be/FLjRAzKSvxc

3. Conclusion

In this tutorial, we have simulated a digital Verilog 4-bit counter in two different design levels, one is to simulate the pre-synthesis Verilog code to test its functionality only and another one is to simulate the post-synthesis code after being mapped to real standard cells from your technology library. Both simulations give the same results.

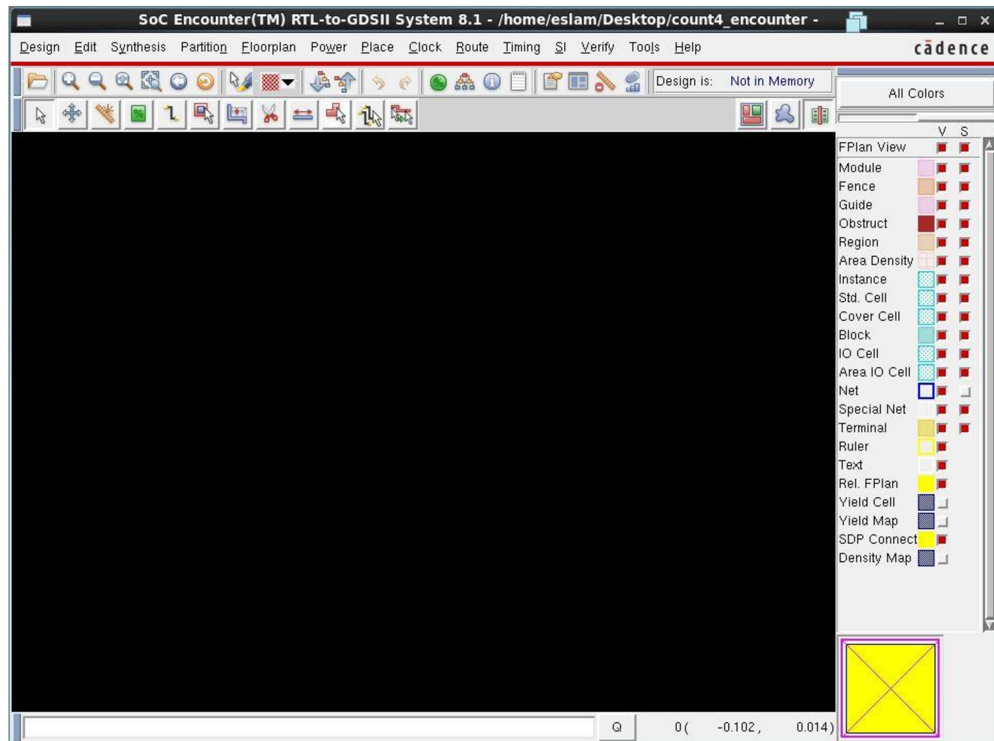
Appendix D

STANDARD CELL PLACEMENT AND ROUTING

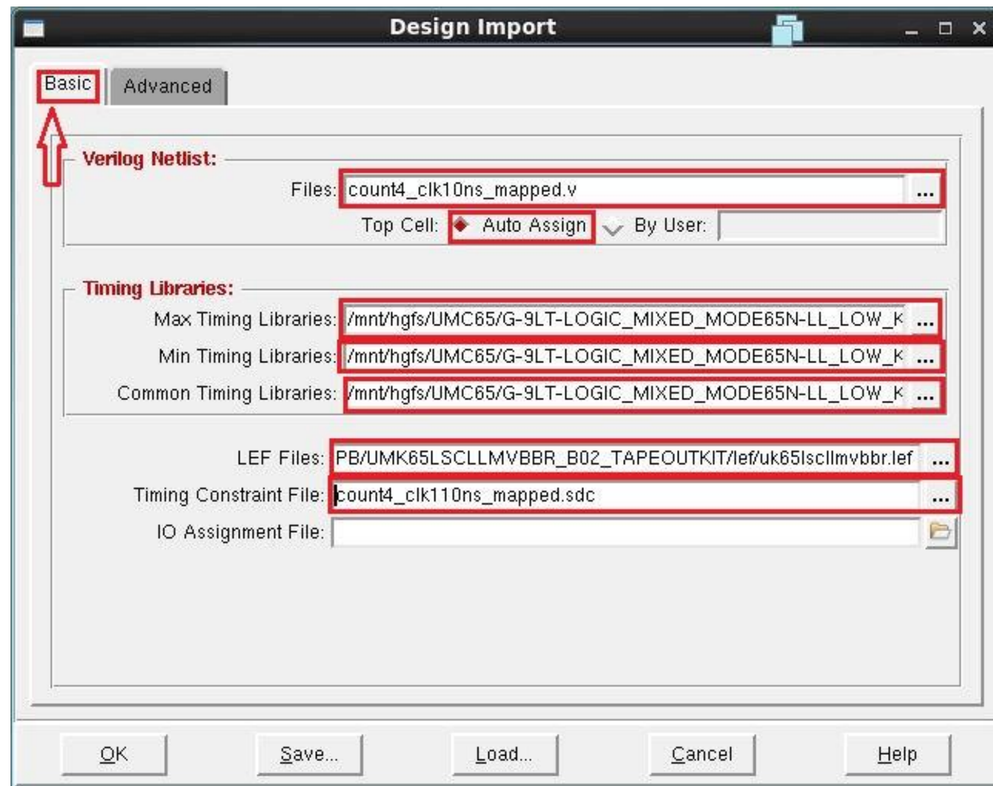
In this tutorial we will learn how to import the gate level netlist file (which we get from SYNOPSYS Design Vision) into CADENCE SOC Encounter to do the placement and routing of the standard cells used in our example, the 4-bit counter. You may import the final layout from this step into CADENCE layout editor to check it through DRC, LVS and PEX and make the post layout simulation. The tutorial is divided into two parts, part 1 and part 2. For part 1 we make all the steps using the graphical environment and for part 2, we make the same steps using a TCL (Tool Command Language) script.

Part 1:

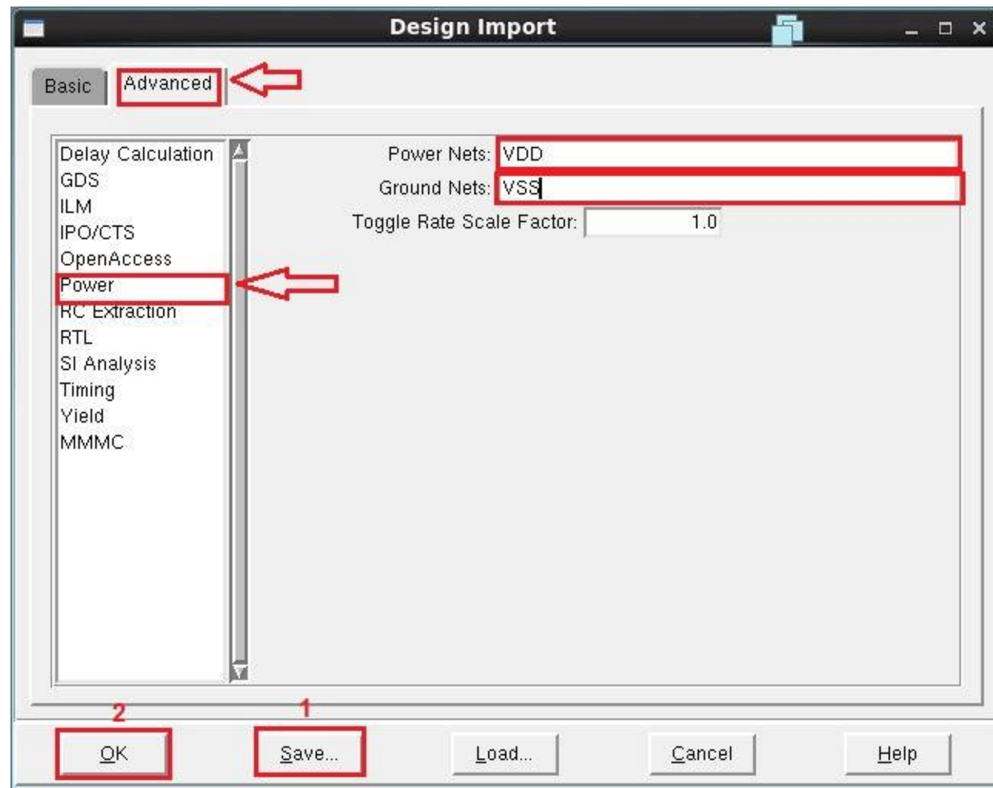
The starting window for SOC Encounter looks like the following figure.



Choose **Design -> Import Design**, the following window will show up. Fill it with your gate level netlist in the field of **Verilog Netlist** then choose your **Timing Libraries** as follows, for **Max Timing Libraries** choose the file with worst case conditions and for **Min Timing Libraries**, choose the file with best case conditions and finally **for Common Timing Libraries,** choose the file with typical case conditions. For **LEF files** you need to add the LEF files from your technology kit. For **Timing Constraints File**, add the SDC file which we got from Design Vision in logic synthesis step.

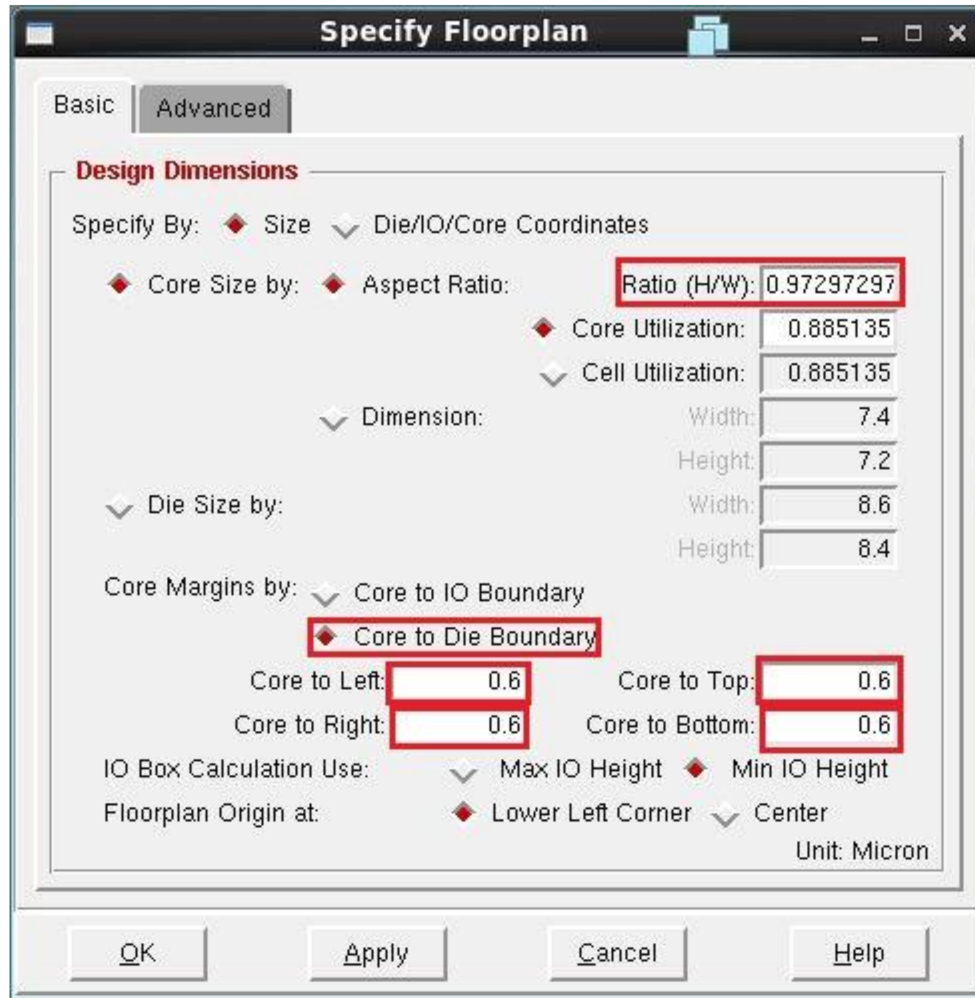


Now move to the **Advanced tab** and choose **Power** to add names for your power nets, make sure to type proper names like in you LEF file. To know the power net names in your technology kit you can check the LEF file and search for "power" to know the name for power net and ground net. For our technology library, net names are VDD and VSS. You may need to save this configuration to use it again instead of inserting all fields from the beginning, to do this step choose **Save** and the **OK**.

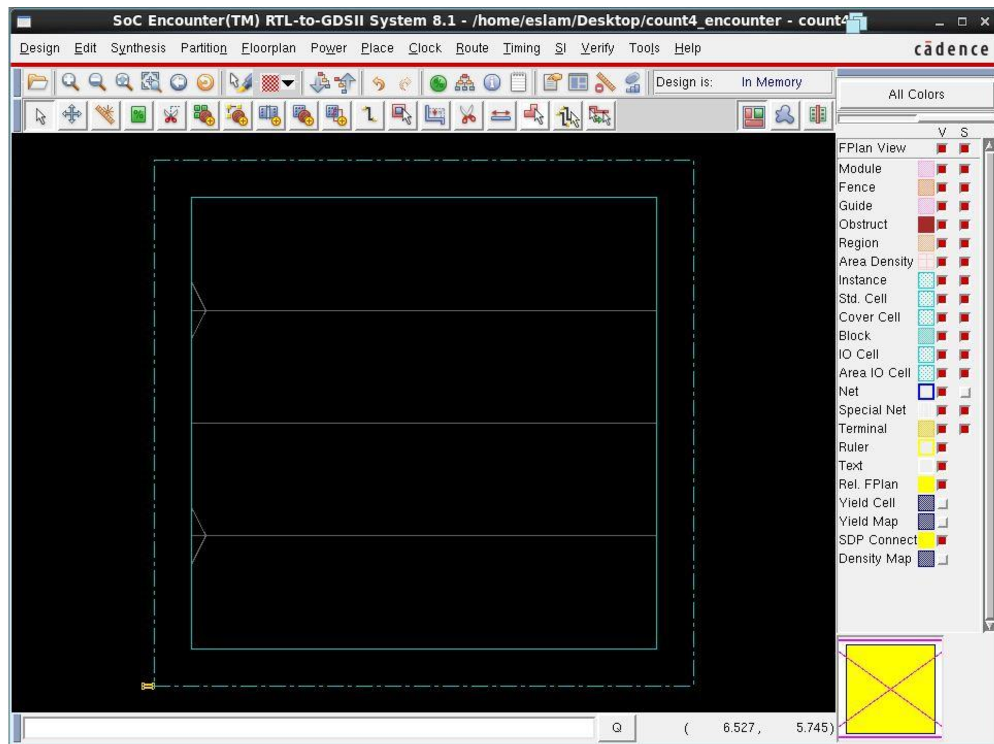


It is a good idea to save your design using **Design -> Save Design As -> SoCE.**

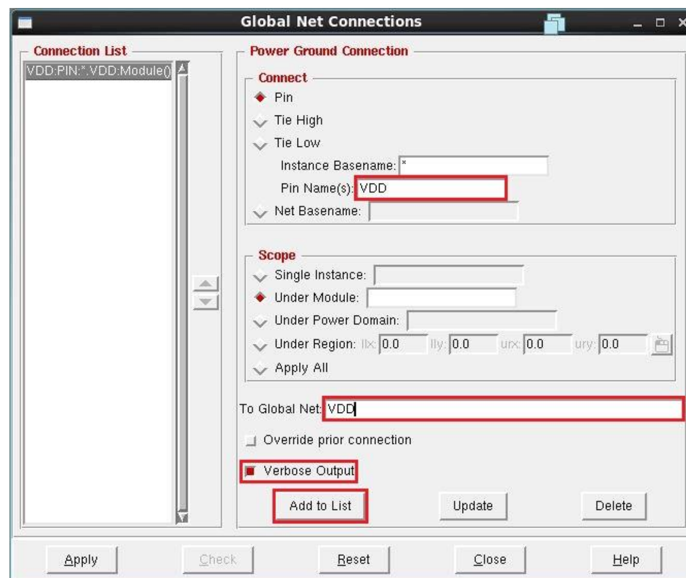
Next step is to specify the floorplan for your design, choose **Floorplan -> Specify Floorplan**. Define an **aspect ratio** of 1 and **core utilization** of 85% which means that 15% of the core area will be free for possible future cell replacements. Choose **core to die boundary** large enough to hold the power rings, we choose it as 0.6 microns. This is enough for this design as there will be one power ring and one ground ring of 0.1 micron and spacing between them of 0.1 micron also.



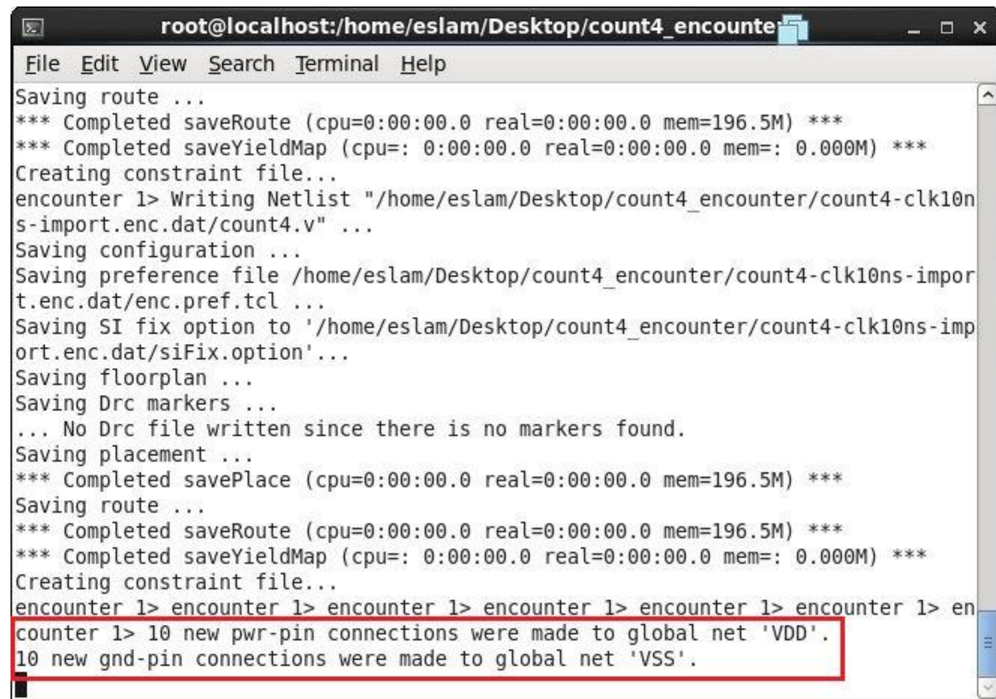
Your floor plan will look like the following one.



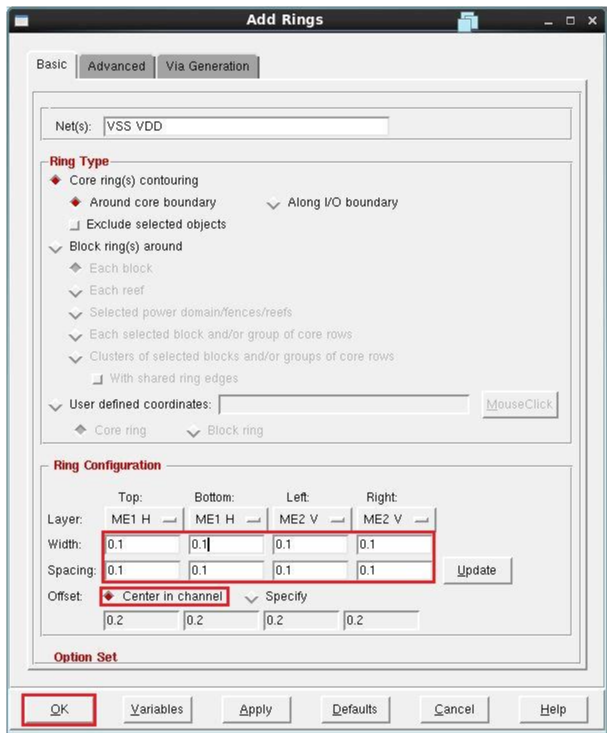
Choose **Floorplan -> Connect Global Nets** to connect power nets to your design. From the following menu type VDD in pin name and VDD in the field of **To Global Net** then click on **Add to List**. Make similar steps for VSS then **Apply** and **Close** this window.



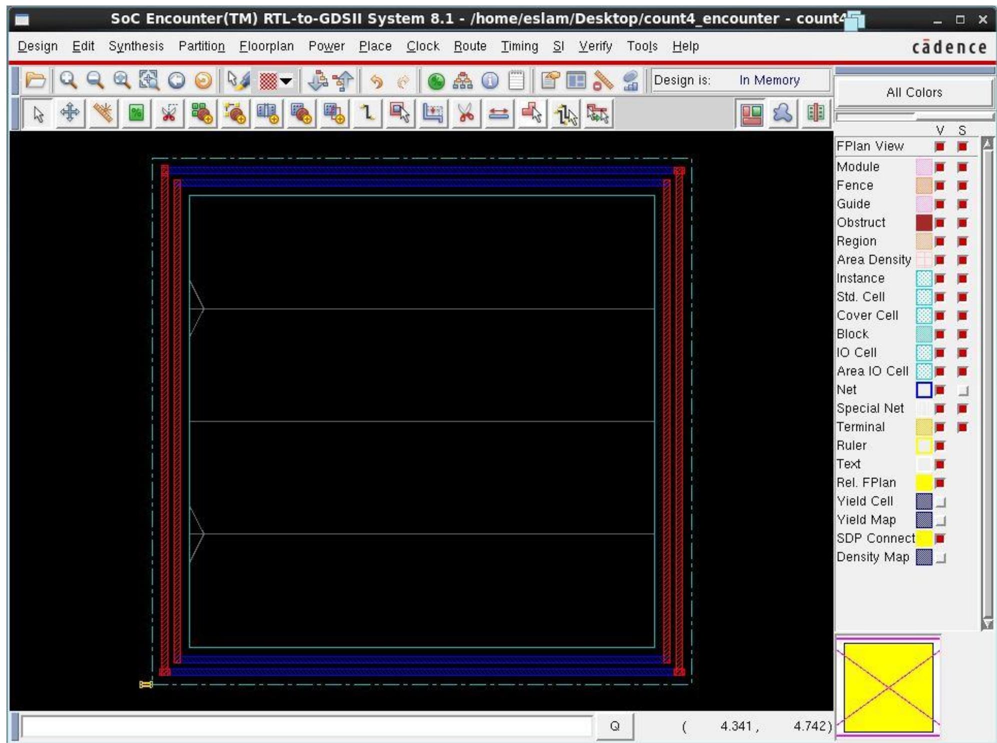
In the command window you will get a report of these connected power pins like the following figure.



Next step is to add power rings to your design. To do this, choose **Power -> Power Planning -> Add Rings** and the following window will show up. From this window, choose the width of your power rings and the spacing between them.

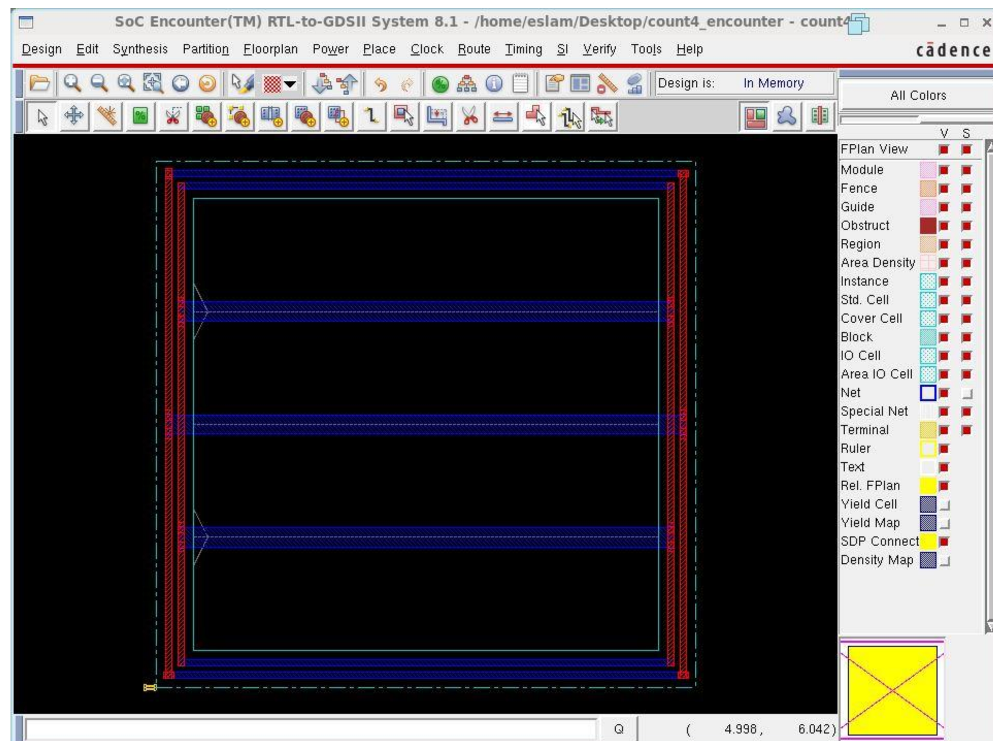


Your floorplan will now look like the following one.

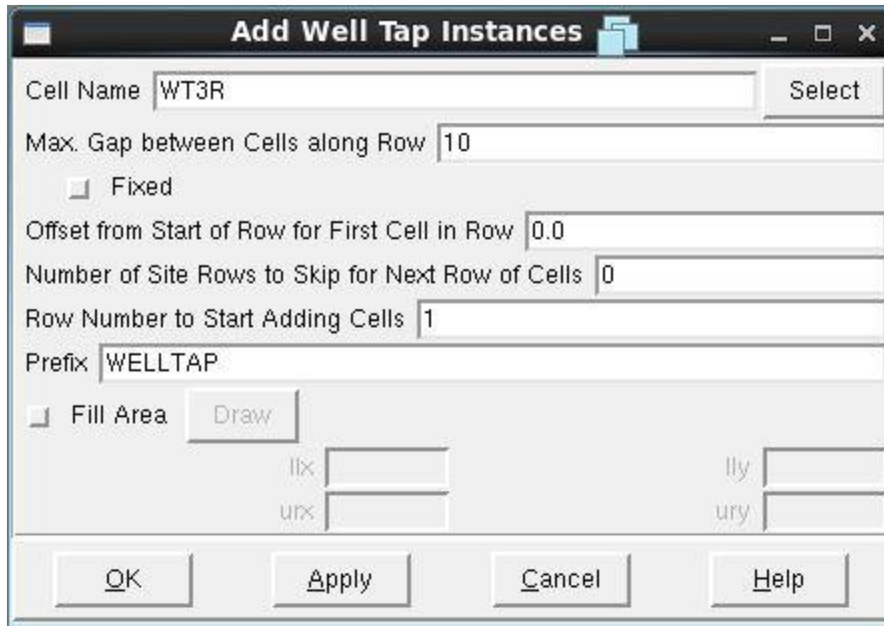


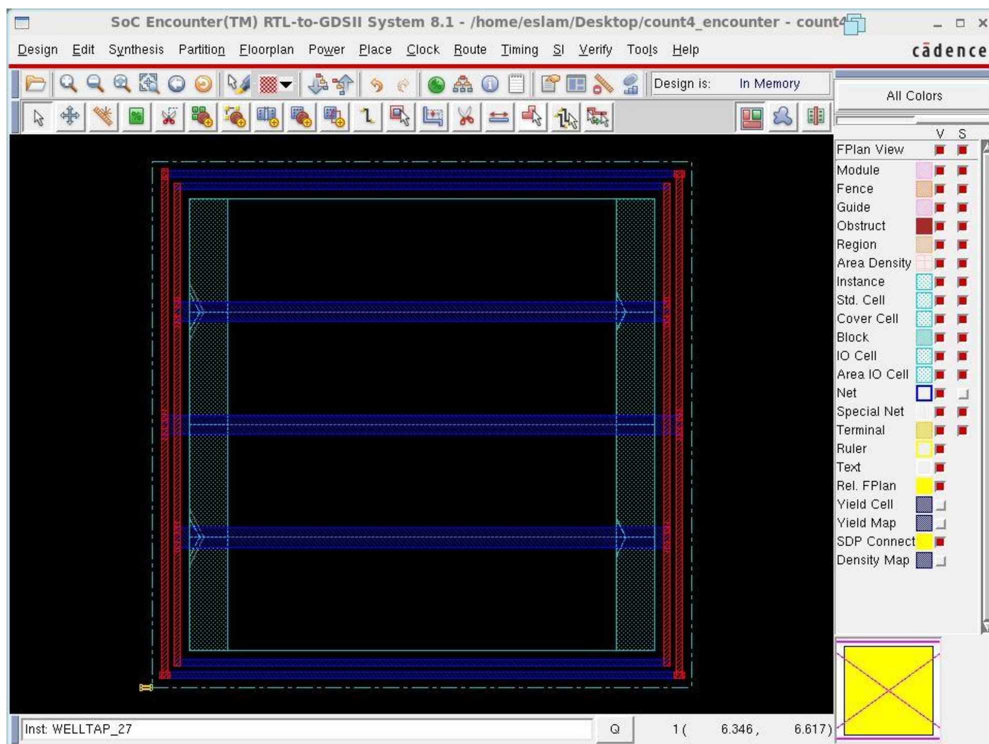
Save your design to this point using **Design -> Save Design AS -> SoCE** and choose an appropriate name, we choose count4_clk10ns-pring.enc.

Now, it is possible to route the power grid. Select **Route -> Special Route** then clock **OK**. After this step you will get the following.

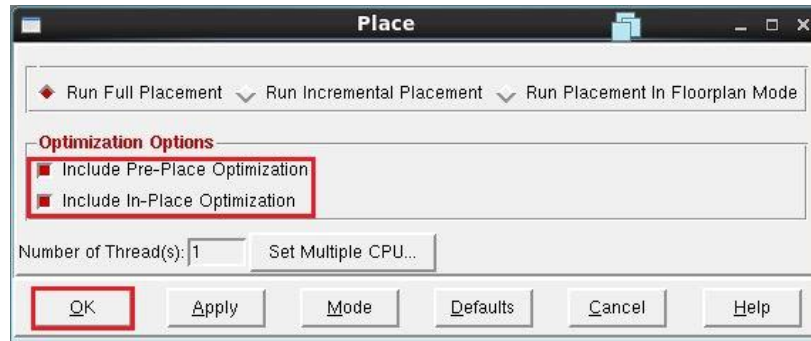


Next step is to add well taps to your design so that your VDD and GND are connected to substrate and n-wells respectively. This is to help tie them to your VDD and GND levels so that they don't drift too much. Choose **Place -> Physical Cells -> Add Well Tap.** Choose the well tap from your technology library and click **OK**.

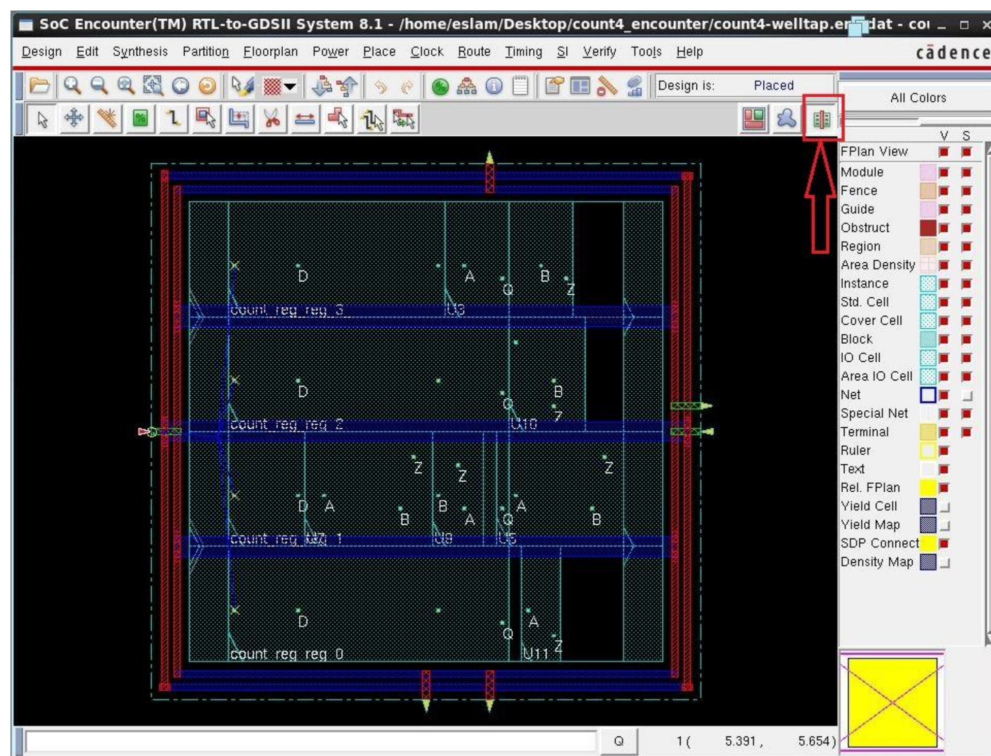




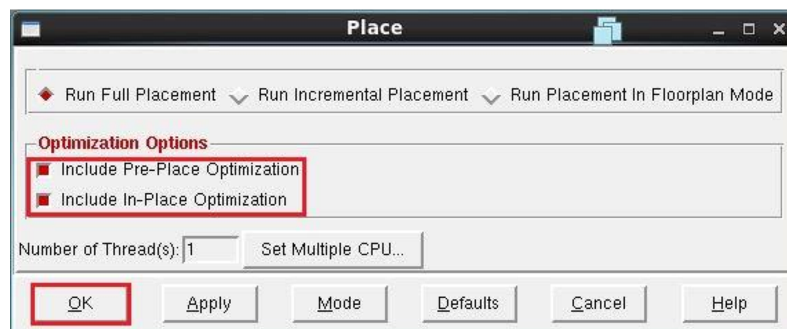
Next step is placing the standard cells. Choose **Place -> Standard Cells** and check the following menu then click **OK**.



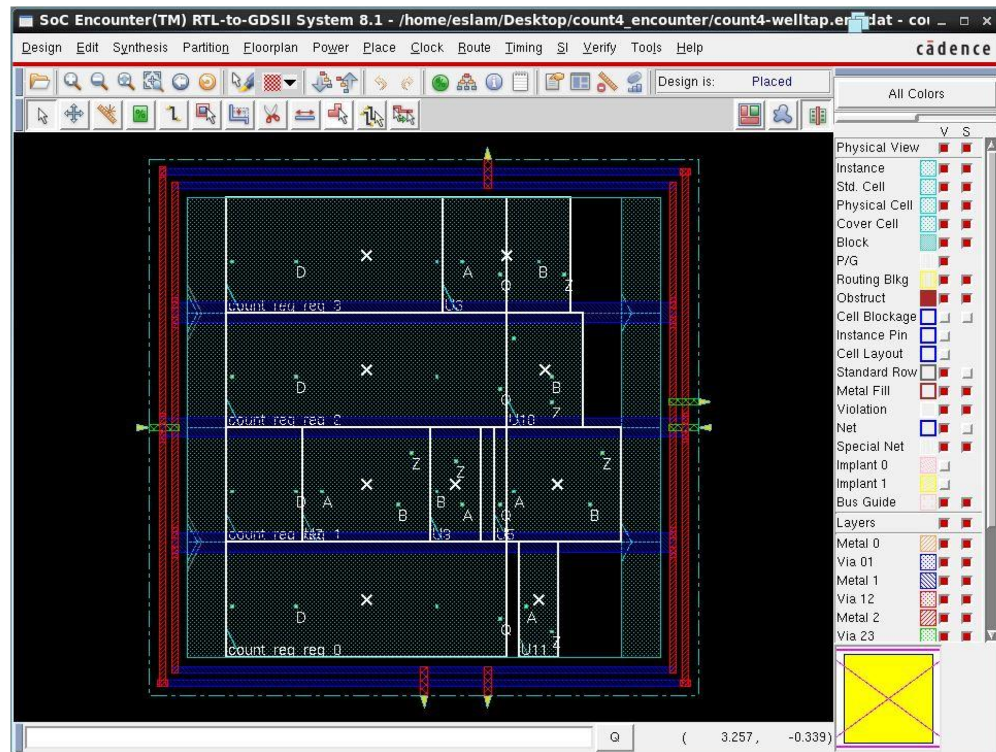
Your standard cells are now placed and you can check them.



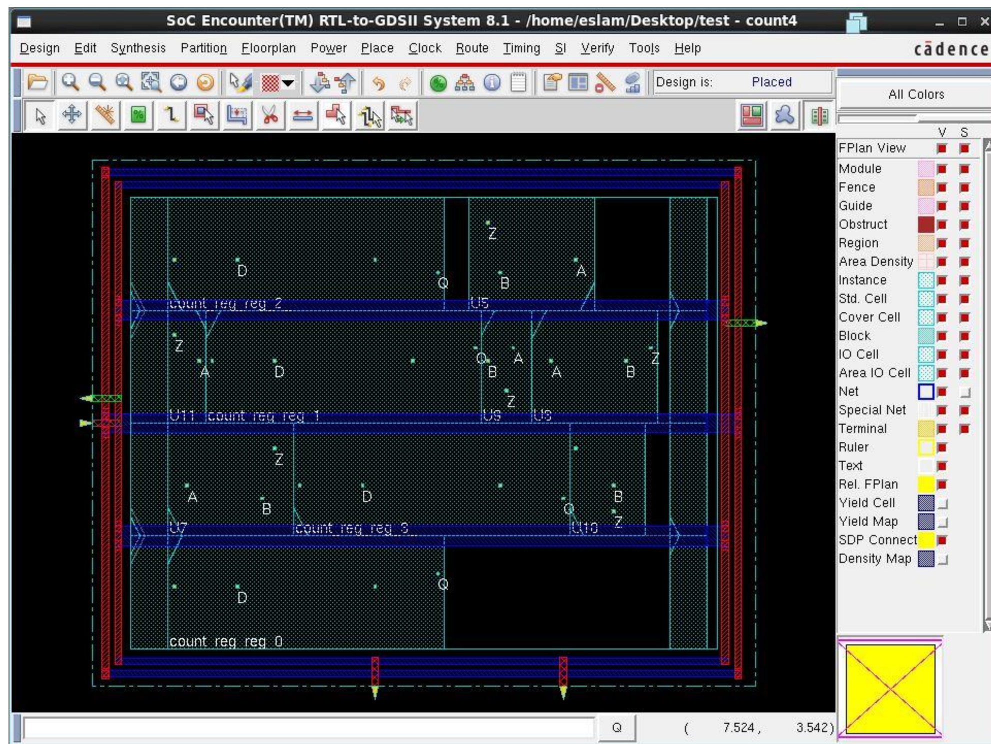
To check placement, choose **Place -> Check Placement** and click **OK**.



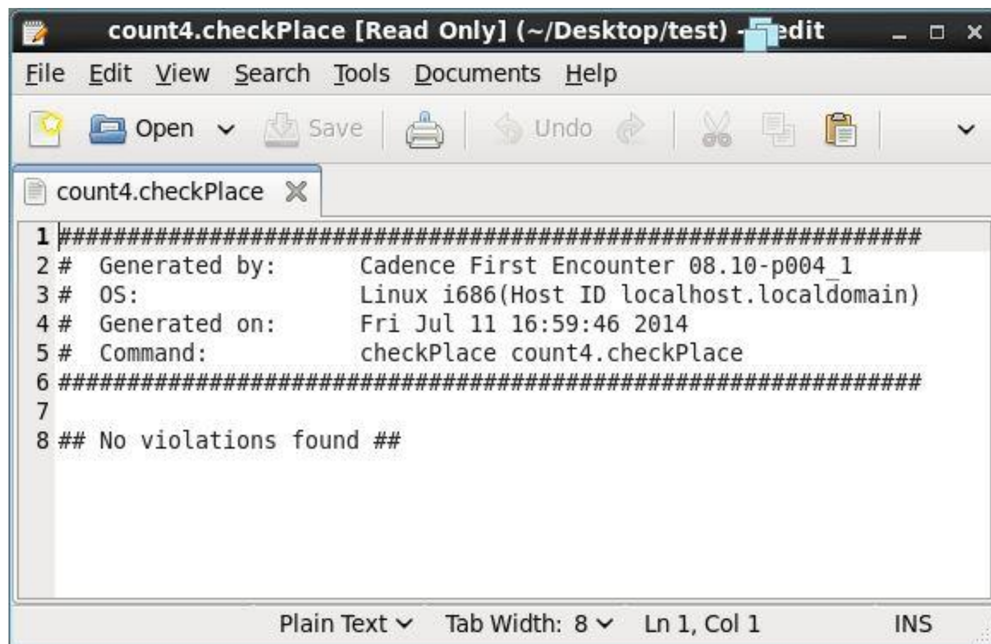
After clicking OK we got these violations, there was an overlapping between some standard cells.



To solve these violations we edited the aspect ratio again from **Floorplan -> Specify Floorplan** and the problem had been solved.

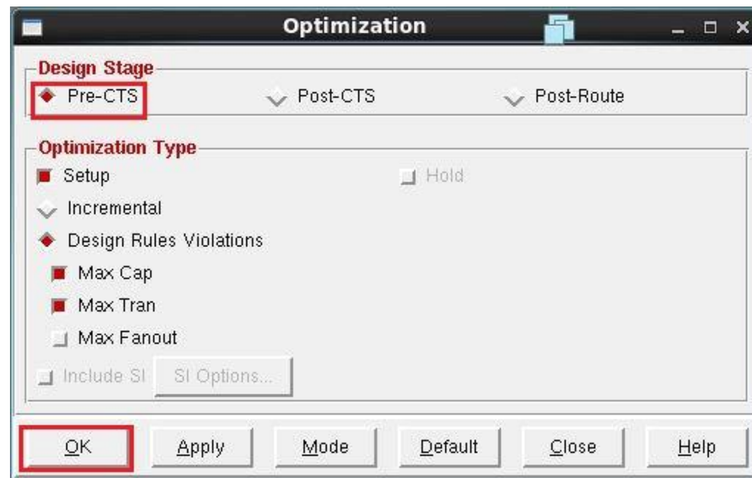


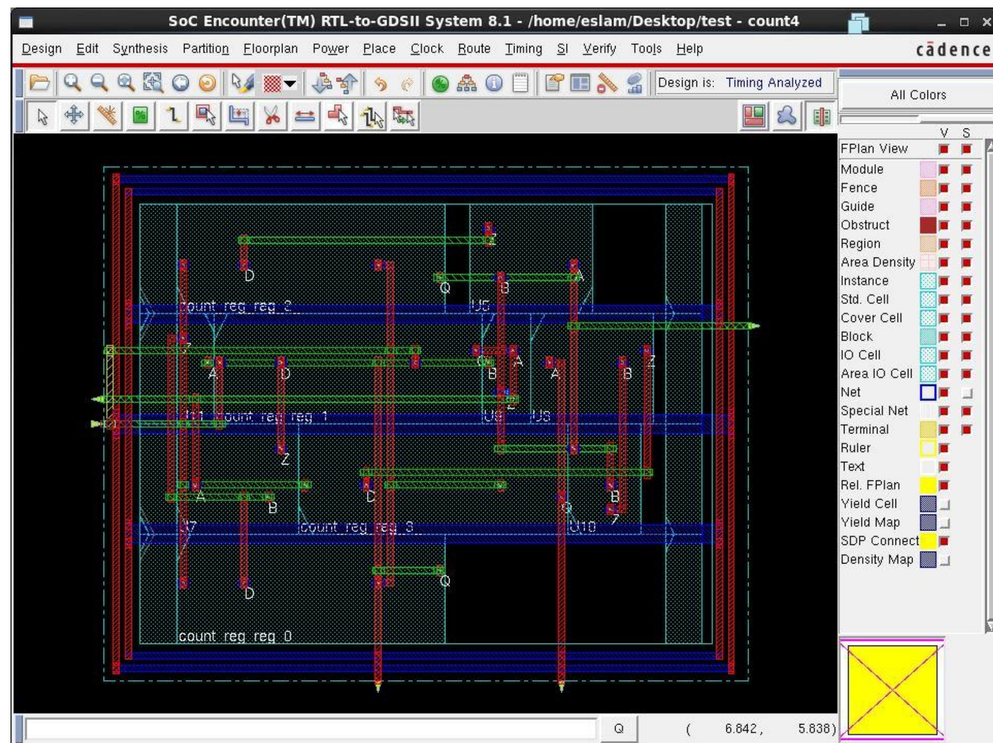
Check the file named .checkPlace in your work directory to make sure that there are no violations.



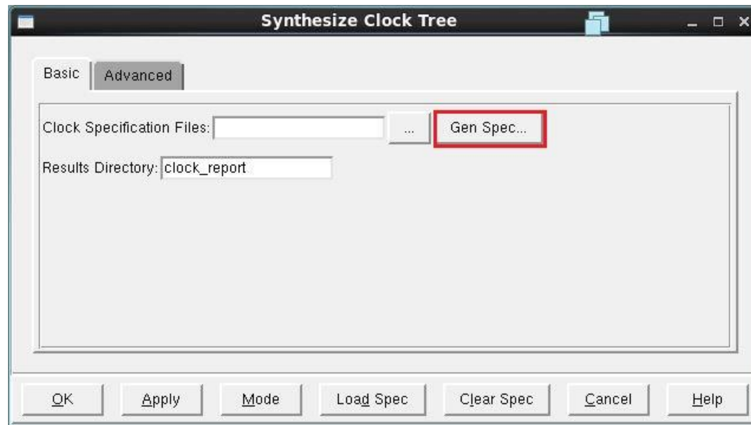
Now choose **Timing -> Optimize** to make the timing optimization for Pre-CTS.

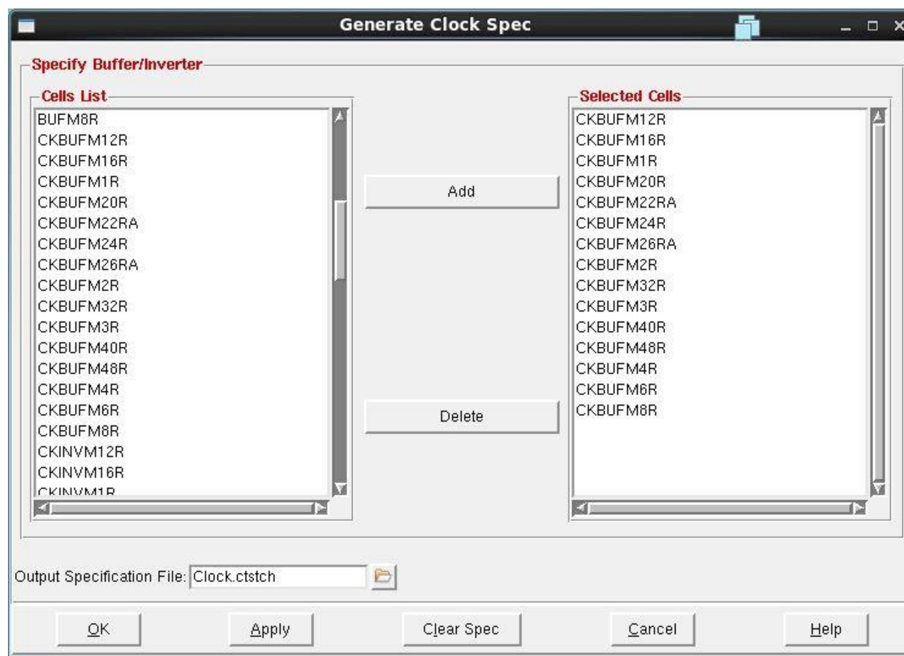
Click **OK** in the following menu.





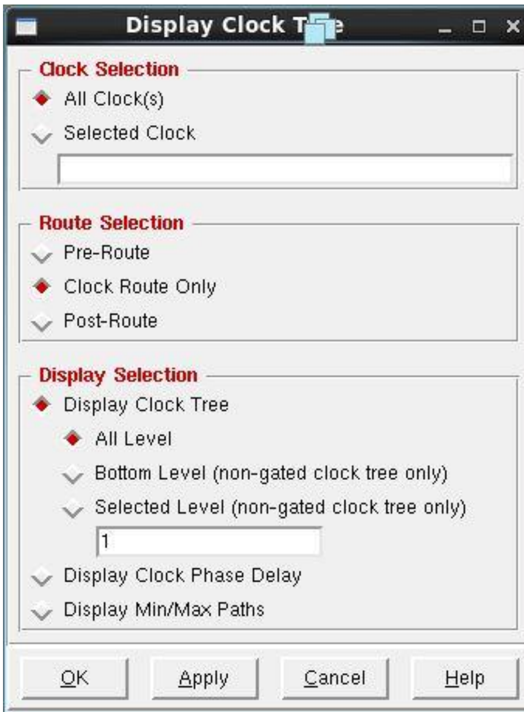
Now choose Clock -> Design Clock then click Gen Spec and select your CLKBUF cells.



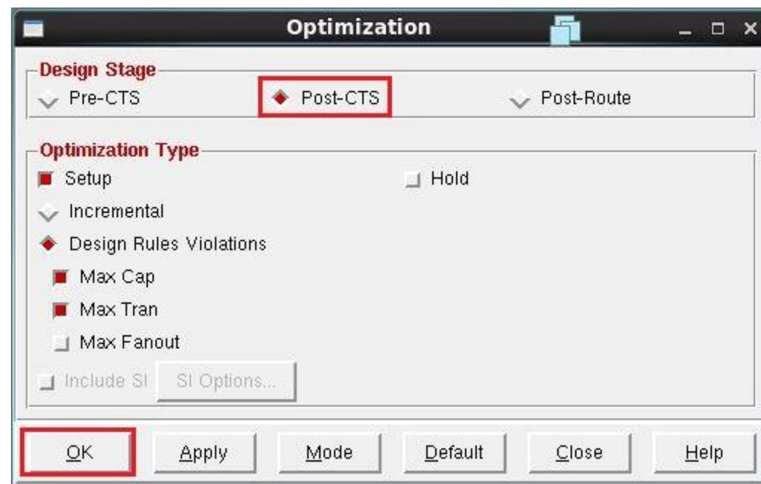


Then clock **OK**.

Now choose **Clock -> Display -> Display Clock Tree**. From the following menu you can display the clock phase delay.

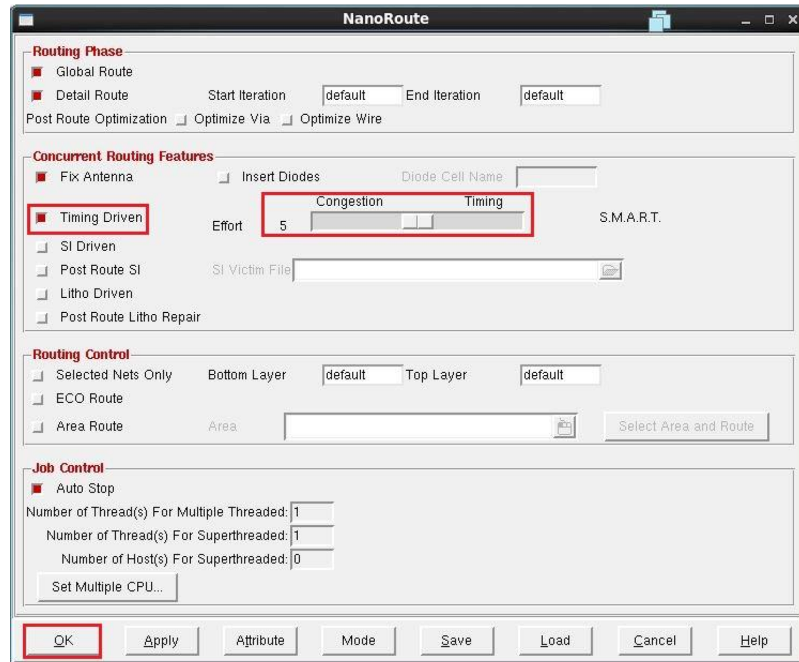


Now choose **Timing -> Optimize** then select **Post-CTS** and click **OK**.

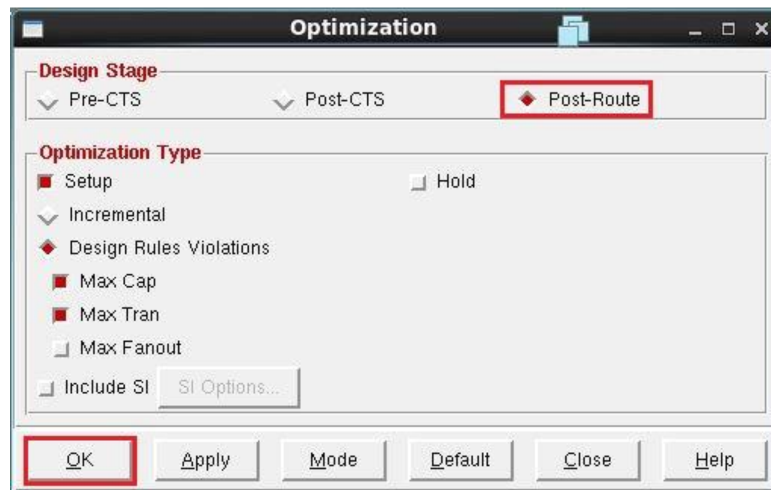


Open your terminal window and check the WNS field (Worst Negative Slack) which means the slack of the critical path in your design.

Now choose **Timing -> NanoRoute -> Route**. From the following menu check the required fields and click **OK**.



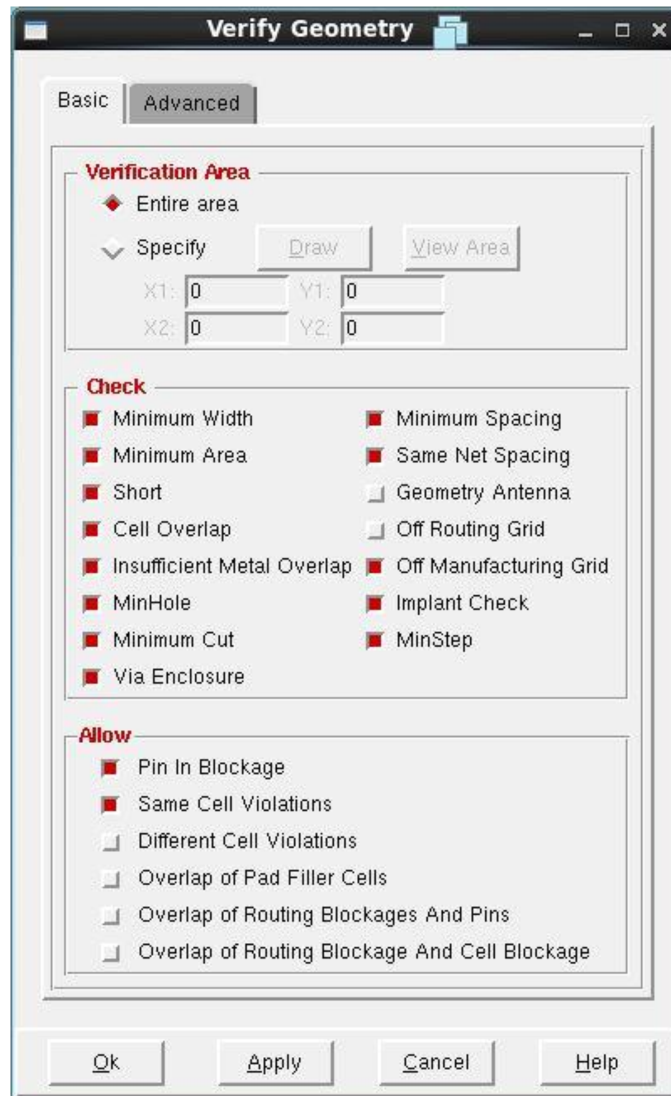
Next step is the post route timing optimization. Choose **Timing -> Optimize** and choose **Post-Route** then **OK**.



Next step is to add filler cells. Choose **Place -> Physical Cells -> Add Filler** and select filler cells from your technology library. Filler cells will fill remaining holes in the rows and ensure the continuity of power/ground rails and N+/P+ wells.

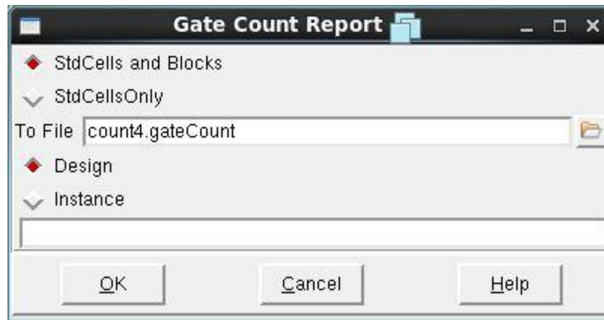
The following steps are to check your design. To check connectivity, choose **Verify -> Verify Connectivity** then press **OK** in the following window.

To check geometry of your design, choose **Verify -> Verify Geometry** then click **OK**.



You can now generate some useful reports. Choose **Design -> Report -> Netlist Statistics** and check your command window to see this report file.

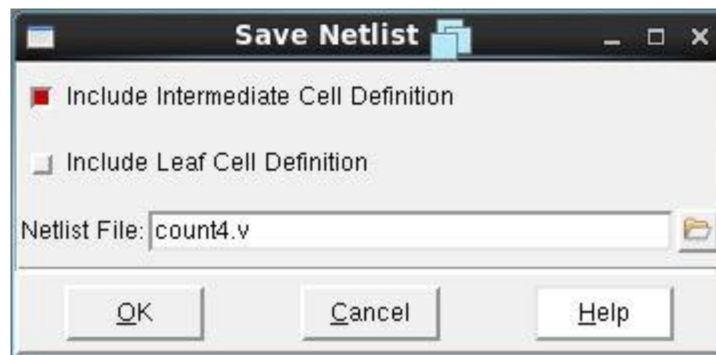
You can also check the number of gates used in your design by choosing **Design -> Report -> Gate Count** and click OK in the next window.



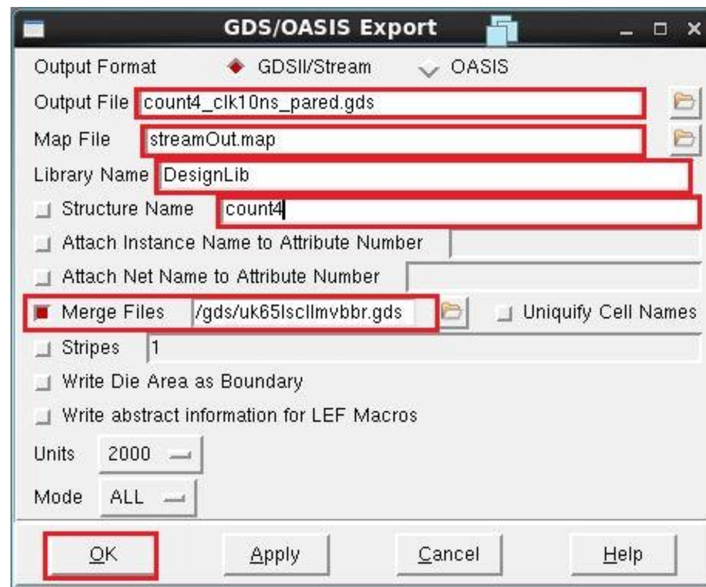
In command window, type ***report_timing*** to get a timing report for your design.

To export a standard delay format file type the following in command window
write_sdf count4_clk10ns_pared.sdf -version 2.1

The P+R netlist may be different from the imported netlist as cells may have been added or replaced during clock tree synthesis (CTS) and various timing optimization phases. To export this netlist choose **Design -> Save -> Netlist** and click **OK** in the following window.



The next step is to export this layout to a file which can be used in virtuoso layout editor, choose **Design -> Save -> GDS/OASIS**. Choose an appropriate **Map File**. The library name is the design library name in virtuoso. Choose merge files and merge it with the .gds files from your technology library.



For more details please check this video:

http://youtu.be/udPMw9_rZL0

Part 2:

The same steps of part 1 will be done here using this TCL script.

```
# Importing the Design
loadConfig Default.conf
setDrawView fplan
fit
saveDesign count4-import.enc
#Floorplanning the Design
floorPlan -r 1 0.85 0.6 0.6 0.6 0.6
saveDesign count4-fplan.enc
#Power Planning
clearGlobalNets
globalNetConnect VDD -type pgpin -pin VDD -inst * -module {} -verbose
globalNetConnect VSS -type pgpin -pin VSS -inst * -module {} -verbose
addRing \
-around core \
-nets {VSS VDD} \
-center 1 \
-width_bottom 0.1 -width_right 0.1 -width_top 0.1 -width_left 0.1 \
-spacing_bottom 0.1 -spacing_right 0.1 -spacing_top 0.1 -spacing_left 0.1 \
-layer_bottom ME1 -layer_right ME2 -layer_top ME1 -layer_left ME2 \
-bl 1 -br 1 -rb 0 -rt 0 -tr 0 -tl 0 -lt 1 -lb 1
#placing well taps
addWellTap -cell WT3R -maxGap 10 -skipRow 1 -startRowNum 2 -prefix WELLTAP
#special route
sroute \
-connect { blockPin corePin floatingStripe } \
-blockPin { onBoundary bottomBoundary rightBoundary } \
-allowJogging 1
saveDesign count4.enc
#Placing the standard cells
setPlaceMode -timingDriven true
placeDesign -prePlaceOpt
setDrawView place
checkPlace
optDesign -preCTS -outDir /home/eslam/Desktop/encounter
saveDesign count4-placed.enc
#Synthesizing a Clock Tree
createClockTreeSpec -output count4_spec.cts \
```

```
-bufferList CKBUFM12R CKBUFM16R CKBUFM1R CKBUFM20R CKBUFM22RA CKBUFM24R
CKBUFM26RA CKBUFM2R CKBUFM32R CKBUFM3R CKBUFM40R \
CKBUFM48R CKBUFM4R CKBUFM6R CKBUFM8R CKINVM12R CKINVM16R CKINVM1R
CKINVM20R CKINVM22RA CKINVM24R CKINVM26RA \CKINVM2R CKINVM32R CKINVM3R
CKINVM40R CKINVM48R CKINVM4R CKINVM6R CKINVM8R
clockDesign -specFile pfd_loopF_spec.cts \
        -outDir /home/eslam/Desktop/encounter
optDesign -postCTS -outDir /home/eslam/Desktop/encounter
saveDesign count4-cts.enc
#Routing the Design
setNanoRouteMode -routeWithTimingDriven true -routeTdrEffort 5
routeDesign
optDesign -postRoute -outDir /home/eslam/Desktop/encounter
saveDesign count4-routed.enc
#Design Finishing
addFiller \
  -cell { FIL16R FIL1R FIL2R FIL32R FIL4R FIL64R FIL8R FILE16R FILE32R FILE3R FILE4R
FILE64R FILE6R \FILE8R FILEP16R FILEP32R FILEP64R FILEP8 } \
  -prefix FIL
setDrawView place
saveDesign count4-filled.enc
#Checking the Design
verifyConnectivity -type all -report connectivity.rpt
verifyGeometry -report geometry.rpt
#Generating Reports
reportNetStat
reportGateCount -outfile gateCount.rpt
summaryReport -outdir /home/eslam/Desktop/encounter
#Design Export
write_sdf -version 2.1 -precision 4 count4_pared.sdf
saveNetlist -excludeLeafCell count4_pared.v
streamOut count4_pared.gds \
-mapFile streamOut_me_pinOnly.map \
-libName count4_pared \
-merge uk65lscllmvbbr.gds
```

For more details, please check the following video:

http://youtu.be/NEfx3igkzME

Conclusion

Through this tutorial, we started with a gate level netlist (The file we got from SYNOPSYS Design Vision) and followed all the steps to place and route the standard cells in this netlist file. The final output from this tutorial is a layout file and timing constraints files which can be used for post layout simulations.

Appendix E

POWER CALCULATION

The method to measure power using Cadence Spectre is described in this tutorial, the 2-bit inverter in the below figure is used as an example to show how power measurement is done in cadence spectre .

The equations we need to apply to Calculate the average power consumed are :

The Instantaneous Power :

P (t) = Power supply voltage (VDD) * current drawn from power supply at time

Then the Average Power

: $$\textbf{Average Power} = \frac{1}{Time\ Period} \int_{Time\ Period} \textbf{Instantaneous Power}$$

The following changes needs to be done for the measurement of the power drawn from the power supply .

1. Changes to the Existing Schematic :

- On the top-level of the schematic, add a Vdc source (from the analog library) and connect its positive terminal to the VDD.
- Select the Vdc source (a white box appears around the selected item), and press Q, an edit object properties window will appear Type he power supply value (which is 3v in this example) across DC Voltage and press OK.

Important note **:**This addition of the Vdc source has to be done only to the top-level of the design schematic and **SHOULD NOT** be done for each of the blocks in the project.

Now, Save the sheet (check and save) and go to the analog-environment window to perform the simulation.

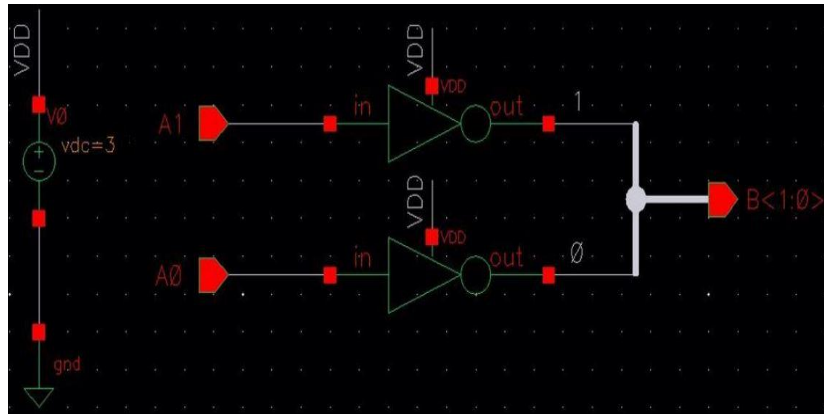


Figure 1

2. Simulation

- Make all the necessary set-up for the simulation
- To plot the current drawn from the VDD, select Output-> To Be Plotted - > Select on Schematic in the analog-environment window and then select the -ve terminal of the Vdc source in the schematic (because the current plotted for a certain node is the input to this node).
- A circle appears in the schematic as shown in figure 2, make sure the circle appears. If it does not appear, then you are plotting the voltage and not the current.

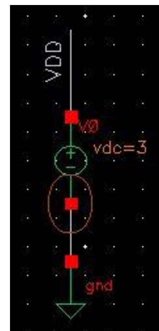


Figure 2

- Simulate the circuit and the plot of the output current from the VDD will be as shown in figure 3

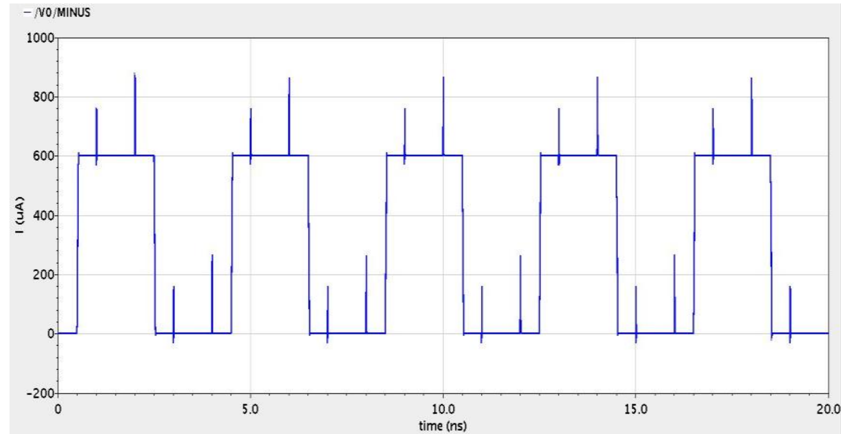


Figure 3

- Select the tools -> calculator from the analog-environment window
- To create TRAN current expression, check the (it) choice and then reselect the -ve terminal of the Vdc source in the schematic, after that select the function "**INTEG**" from the built-in functions. Figure 4 clarify this step.

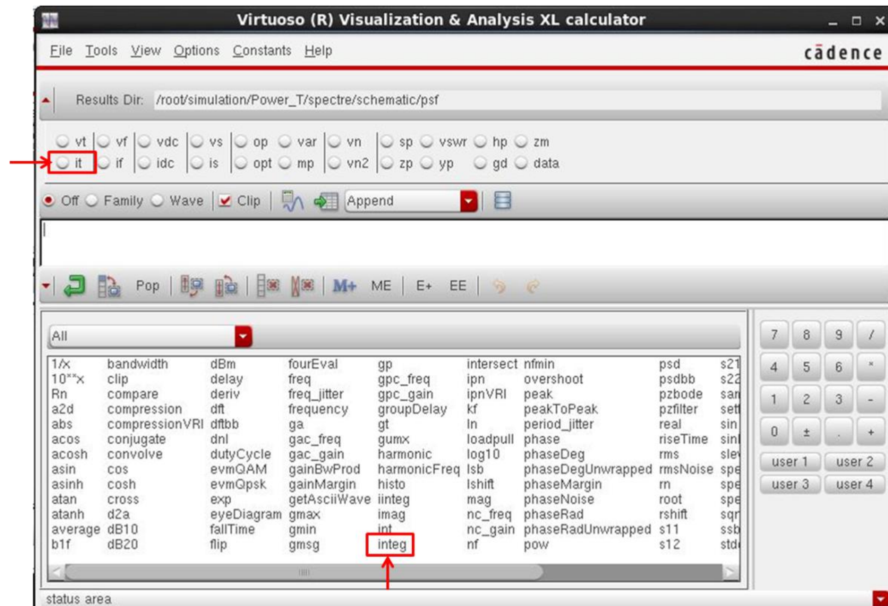


Figure 4

- The calculator window will appear as shown in figure 5 .This simulation (for the 2-bit inverter) is done from 0n to 20ns. Let's find the average power consumed by this circuit in this period, thus the integration should performed in this period (figure 5).In the Signal text box, multiply the current waveform by 3 and divide by 20ns (figure 5).Press OK.

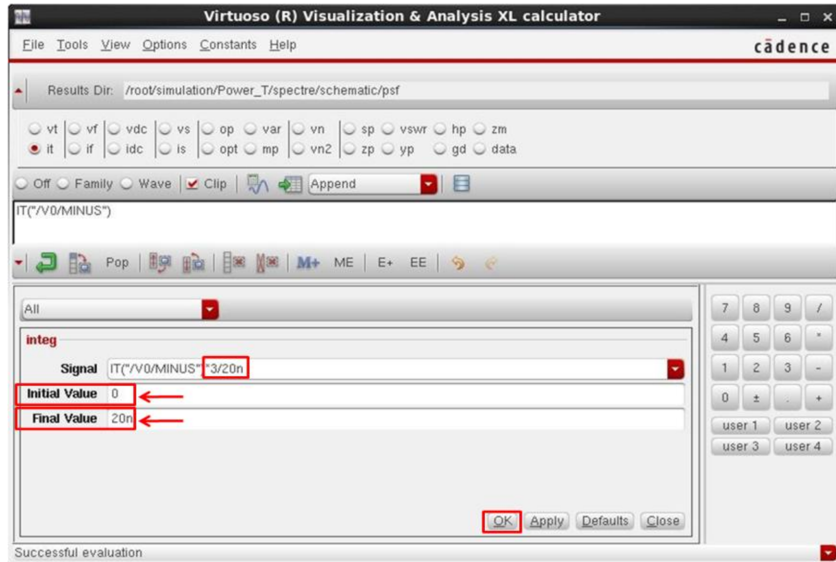
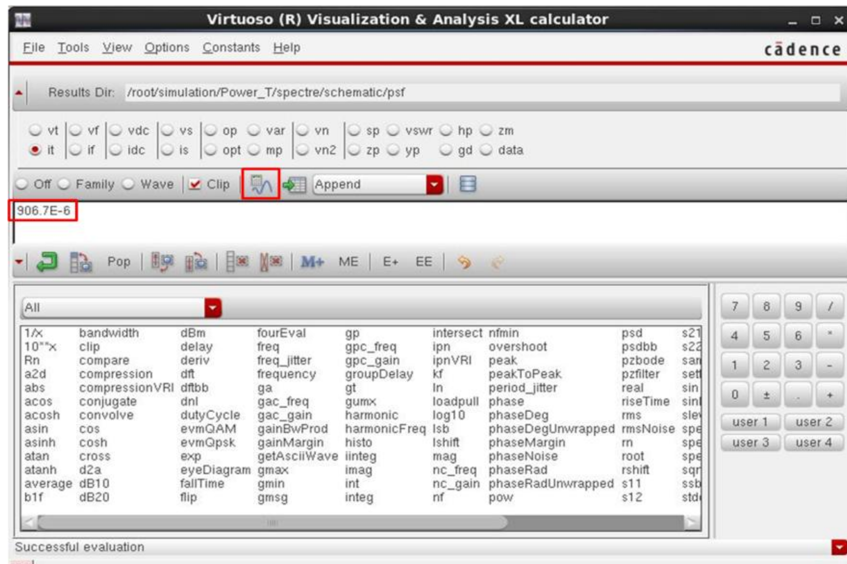


Figure 5

- The expression for power calculation appears in the result text-box. Press "EVAL" from the keypad in the calculator. The average power consumed by this circuit will be displayed in the result text-box, as shown in figure 6.



Figure