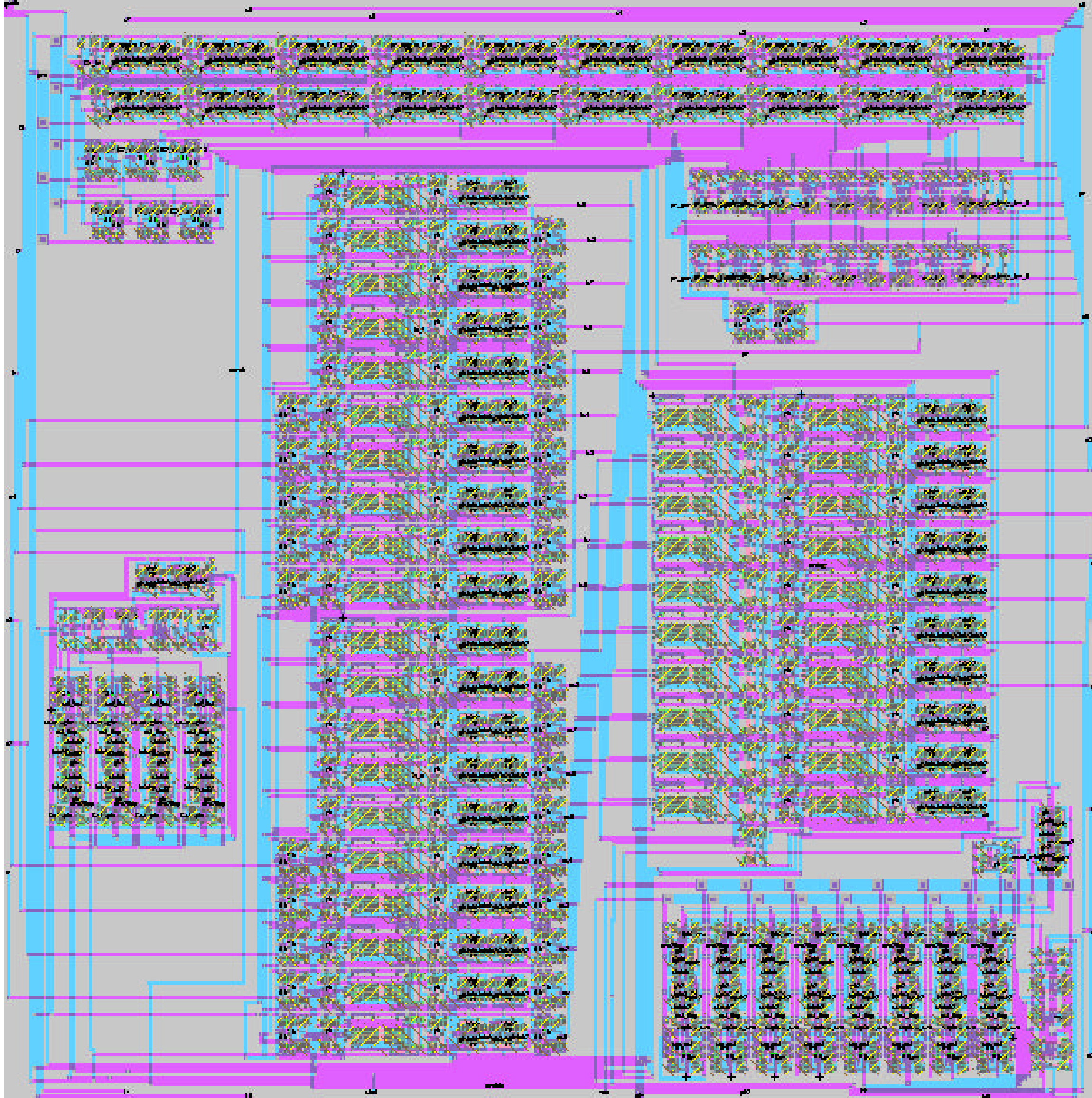
"A Single Satellite/Offset GPS Searcher"



Sergio Rodrigues and Fernando Mattos

E158 – VLSI
Final Project Report

April 10, 2001

ABSTRACT

In this project we developed a single satellite/offset GPS searcher using the 1.5 µm, 5V MOSIS process. A GPS (Global Positioning System) searcher is an important module in a GPS receiver that constantly analyzes antenna data and identifies which satellites are available in the user's location and their corresponding power.

Our design allows searching for a single satellite at a single offset at a time. This is quite simple in comparison to available receivers in the market, but we were limited by the required size of the chip for this project.

This searcher will be manufactured with the MOSIS 1.5µm, 2 metal, 1 poly process on a 2.2mm by 2.2mm die and placed in the tinyCHIP 40 pin DIP package.

TABLE OF CONTENTS

ABSTRACT iii
LIST OF FACETS – SCHEMATICS v
LIST OF FACETS – LAYOUT vi
INTRODUCTION 7
ARCHITECTURE 9
PN Generator 9
BDS 12
Coherent Accumulator Block 12
Noncoherent Accumulator Block 13
Coherent Counter Block 13
Noncoherent Counter Block 13
PINOUT 14
FLOORPLAN 16
SIMULATION 18
Simulation Results 20
POSTFABRICATION TESTING 23
List of Necessary Changes to DN2000K10 board 23
VERIFICATION RESULTS 25
SCHEMATICS 29
LAYOUT 45
APPENDIX A : vlsi_searcher.m SOURCE CODE 63
APPENDIX B : vlsi.m SOURCE CODE 65

LIST OF FACETS – SCHEMATICS

Basic Cells **29**
inv 29
nor2 29
or2 29
nand2 30
and2 30
nand3 30
mux2 31
mux4 31
fulladder 31
latch 32
flop 32
xor2 32
PN Generator Cells **33**
pn_gen_or_flop 33
pn_gen_10_lfsr 33
pn_gen_mux 34
pn_gen 35
Datapath Cells **36**
coh_add_and_inv 36
coh_adder_1bit 36
accumulator_reg 36
datapath_energy_1bit 37
Datapath_energy 37
datapath_coherent_1bit 38
datapath_coherent 38
datapath_bds 39
datapath_iq 40
Datapath 41
Coherent/Non Coherent Dwell Counter Cells **42**
Counter_reg_reset 42
4-bit counter 42
8-bit counter 43
coh_dwell_counter 43
noncoh_dwell_counter 43
Searcher **44**

LIST OF FACETS – LAYOUT

Basic cells **45**
inv/nand2/or2 45
mux2 45
mux 4 46
fulladder 46
latch 47
flop 48
xor2 49
PN Generator Cells **50**
pn_gen_or_flop 50
pn_gen_10_lfsr 50
pn_gen_mux 50
Datapath Cells **52**
coh_adder_1bit 52
accumulator_reg 53
datapath_energy_1bit 53
Datapath_energy 54
datapath_coherent 55
datapath_bds 56
datapath_iq 57
Datapath 58
Coherent/Noncoherent Facets **59**
Counter_reg_reset 59
4-bit counter 59
8-bit counter 60
coh_dwell_counter 61
noncoh_dwell_counter 62

INTRODUCTION

The GPS searcher we designed has the same basic functionality of any other searcher in the market, but allows the user to search for only one satellite starting at only one offset. This may seem like a feature that makes it useless, but several of these searchers could be placed in parallel with different start signals, allowing the user to search for different satellites at different offsets.

GPS searchers look for available satellites by correlating a known pseudorandom noise (PN) code with the incoming signal. The PN code is generated with LFSRs, which are then synchronized with the other searcher modules to provide the PN code as it is needed.

The searcher reads the incoming data as two 4-bit signals: I (for in-phase) and Q (for quadra-phase). In simpler words, these two signals are the output of a 16-bit A/D converter of the antenna signal. In general, the antenna data is sampled with at least 4 samples. However, our chip does not analyze all samples. The user must choose the sample to be analyzed and provide that to the chip.

During a search, the searcher performs coherent accumulations and noncoherent accumulation. In coherent accumulation, the correlation results of I and Q are accumulated independently for a certain amount of chips (defined by the input N). The value of N ranges from 0 to 3, to indicate 17, 33, 65 or 129 accumulations. During noncoherent accumulation, the absolute values of the independent coherent results are summed together. Noncoherent accumulation is necessary because of the natural difference in frequency between the searcher and the satellite transmitting the data. Our

searcher performs L noncoherent accumulations (provided by user). The input L can vary from 0 to 3, indicating 1 to 4 noncoherent accumulations.

At the end of the search, out chip outputs a *done* interrupt and the energy value found for that satellite.

ARCHITECTURE

We divided this chip into 6 functional blocks: PN generator, BDS (2 instances), coherent accumulator (2 instances), noncoherent accumulator, coherent counter, and noncoherent counter. The figure below shows the interface between the blocks:

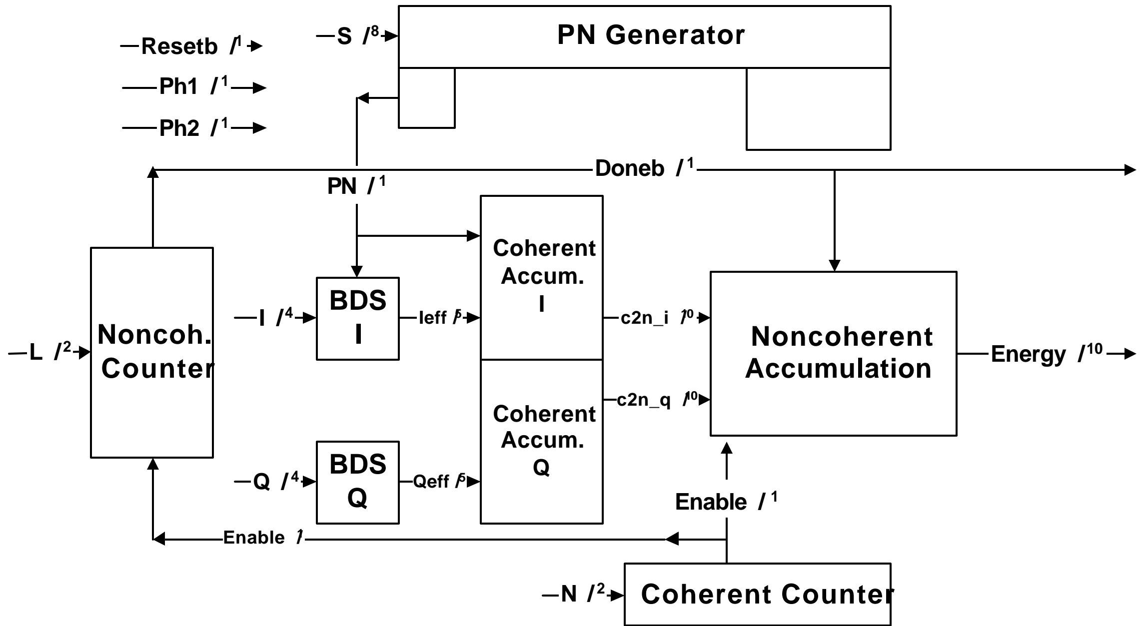


Figure 1: Top-level interface between searcher modules

PN Generator

The function of the PN generator is to produce the pseudorandom noise code for the GPS satellite of interest. It performs the operations described in the following figure:

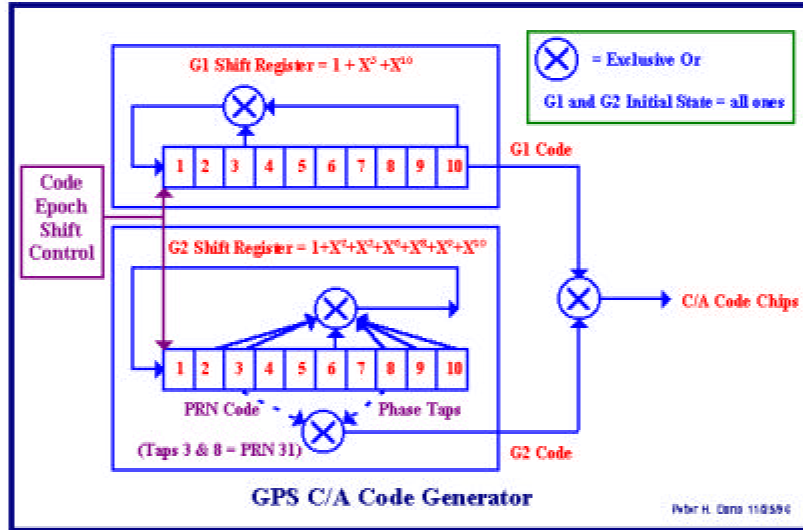


Figure 2: GPS PN code generation

This module contains two 10-bit LFSRs and two 10-bit MUXs. The LFSRs are rows with 10 flip-flops, called G1 and G2. All flops in G1 and G2 reset to high. The input to the first flop in G1 is the XOR of the values in flops 3 and 10. The input to the first flop in G2 is the XOR of the values in flops 2, 3, 6, 8, 9 and 10. This is common to all satellites.

The output of G1 is simply the value in flop 10. On the other hand, the output of G2 is the XOR of the values in two flip-flops. The two flops chosen are called the *taps*, and vary for each of the 32 available satellites. For example, satellite 1 uses taps 2 and 6, while satellite 7 uses taps 2 and 10. This process generates distinct PN codes for each satellite. At the end, the outputs of G1 and G2 are XORed, generating the PN code. This process is clocked, so we obtain one bit of PN code for every clock.

To choose which taps will be XORed in G2, we used two 10-bit MUXs. Each 10-bit MUX is responsible for choosing one of the two taps. The figure below shows the 10-bit MUX used:

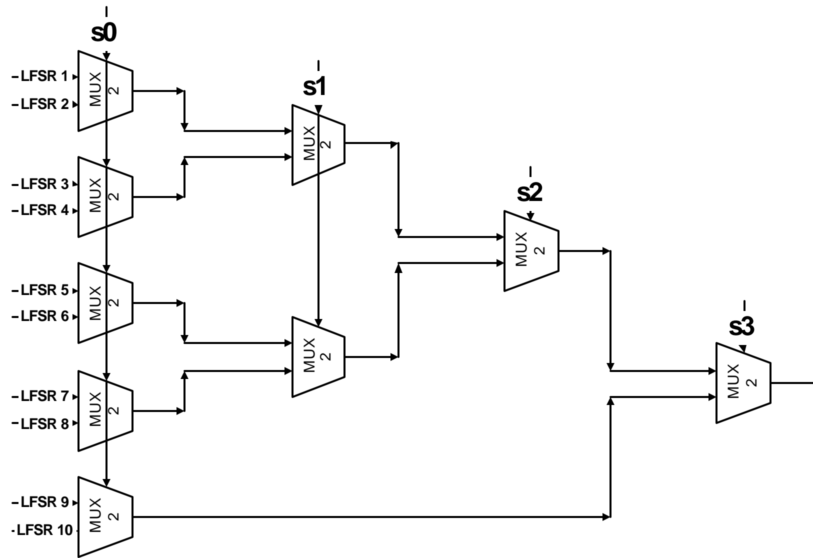


Figure 3: 10-bit MUX used for choosing taps for each satellite

The inputs s3 ~ s0 choose TAP 1, while inputs s7 ~ s4 choose TAP 2. Also, because of the 2-bit MUXs used, the inputs should be inverted. For example, for satellite 1 we want TAP 1 to be 2, and TAP 2 to be 6. Thus, s3 ~ s0 should be 1110, and s7 ~ s4 should be 1010. The table below contains the required input for each satellite:

Table 1: Required input for selecting satellite

Satellite ID	Satellite code (S7 ~ S0)	Satellite ID	Satellite code (S7 ~ S0)
1	10101110	17	11001111
2	10011101	18	10111110
3	10001100	19	10101101
4	01111011	20	10011100
5	01111111	21	10001011
6	01101110	22	01111010
7	10001111	23	11011111
8	01111110	24	10101100
9	01101101	25	10011011
10	11011110	26	10001010
11	11001101	27	01111001
12	10101011	28	01101000
13	10011010	29	10101111
14	10001001	30	10011110
15	01111000	31	10001101
16	01100111	32	01111100

BDS

The BDS block is quite simple. The I and Q inputs are first converted into 5-bit inputs by making the LSB equal to 1. Thus, the inputs represent values ranging from –7.5(10001b) to +7.5 (01111b). The PN code represents what we expect the I and Q values to be. It PN is zero (low), that means we expect the input to be positive. If PN is one (high), we expect the input to be negative. The BDS block outputs a number that represents how much I and Q match the PN code. The more positive the number, the better the match. Thus, if PN is zero and I is +7.5, it outputs +7.5 (a perfect match). If PN is one and I is –7.5, it also outputs +7.5 (a perfect match). However, if PN is one and I is +7.5, then the BDS block outputs –7.5 (a perfect mismatch). These values are then accumulated by the other blocks in the searcher, resulting in a final value that represents how much the entire incoming signal matches the PN code.

Our implementation is actually quite simple. We simply XOR the inputs (I and Q) with PN to generate the outputs Ieff and Qeff. The next block (coherent accumulator) uses PN as a carry-in input to its accumulators (see below). Thus, if PN is one and I is –7.5 (10001), then the BDS block inverts the bits in I (to 01110) and the coherent accumulator block adds PN as a carry-in, resulting in 01111 (+7.5).

Coherent Accumulator Block

This block accumulates the values received from the BDS block. The accumulated value is registered in flip-flops. It is then used as an input to the noncoherent accumulator block when it detected an interrupt from the coherent counter. Once the value has been transmitted, the flip-flops are reset to zero to start a new accumulation.

Noncoherent Accumulator Block

This block sums the coherent accumulator values for I and Q, and then accumulates it. The accumulated value is the instantaneous energy of the signal. Once it detects the *done* interrupt from the noncoherent counter, it refrains from accumulating any further. The energy is then ready for the user.

Coherent Counter Block

This block counts to 17, 33, 65 or 129 for N = 0, 1, 2 or 3 respectively. Once it reaches the requested value, it resets to zero and outputs an interrupt to the coherent accumulator block. This interrupt makes the coherent value be released to the noncoherent accumulator, and it resets the coherent accumulator registers in the next cycle.

Noncoherent Counter Block

This block counts to (L + 1), where L can range from 0 to 3. However, it uses the interrupt from the coherent counter as a carry-in, such that it only increments when the coherent counter outputs the interrupt. Once this block reaches the value in L, it outputs a *done* interrupt that can be detected by the user (indicating that the energy value is ready) and keeps the noncoherent accumulator block from accumulating any further.

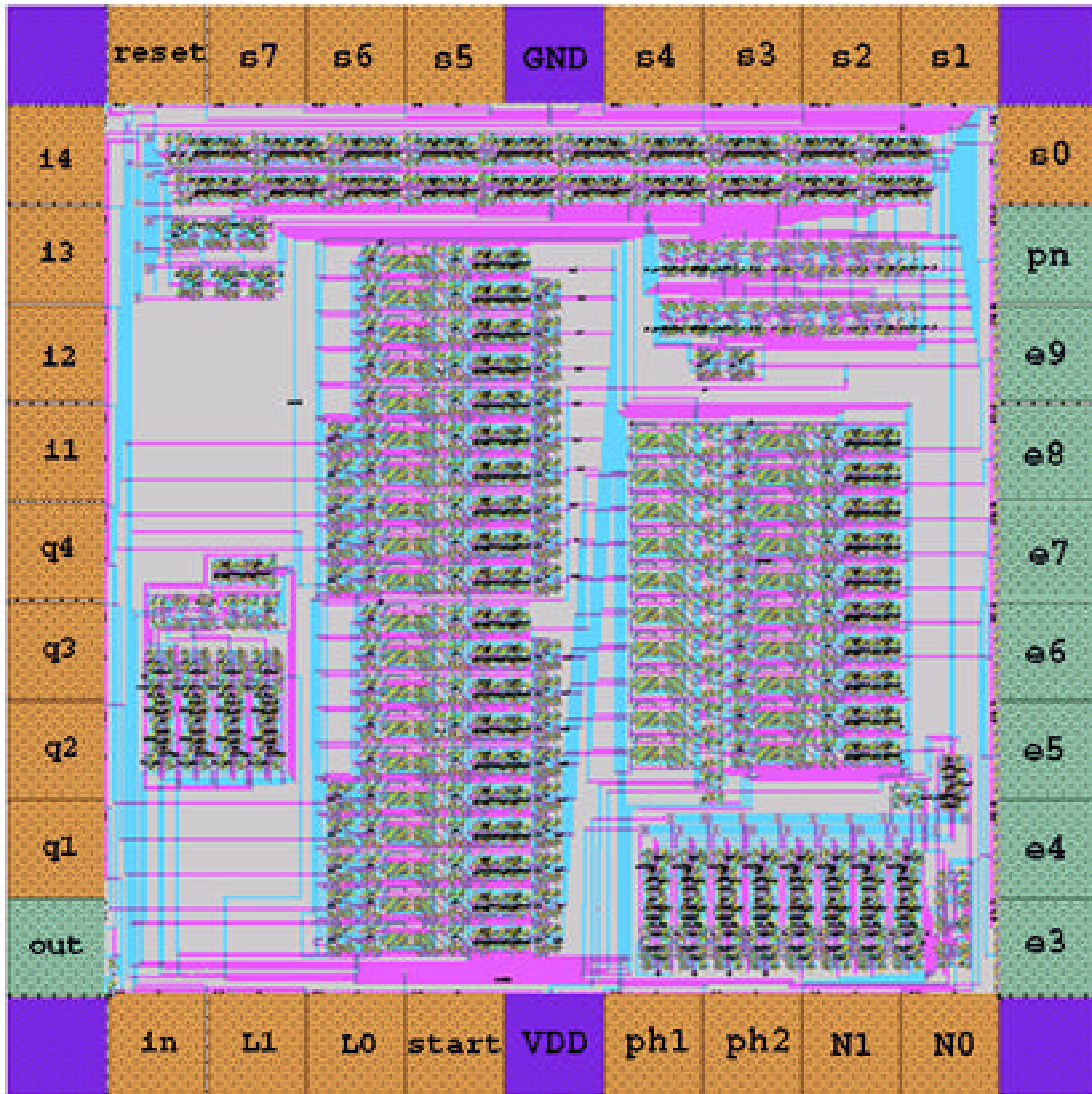
PINOUT TABLE

Table 2: Chip pinout

Name	I/O	Length (bits)	Explanation
I	Input	4 (i4-i0)	Input I data
Q	Input	4 (q4-q0)	Input Q data
Start	Input	1	Start search
Reset	Input	1	Abort search
N	Input	2 (N1,N0)	# of coherent accums. (00 = 17, 01 = 33, 10 = 65, 11 = 129)
L	Input	2 (L1,L0)	# of noncoherent accums.
Sat. ID	Input	8 (s7-s0)	Satellite code (0 ~ 31)
Energy	Output	7 (e9-e3)	Final energy value
Done	Output	1	Search done interrupt
ph1	Input	1	Clock for phase 1
ph2	Input	1	Clock for phase 2
in	Input	1	Input signal to test inverter on chip
out	Output	1	Output signal from inverter

Total = 34 pins

PINOUT DIAGRAM



orange ➔ input
green ➔ output
purple ➔ Vdd/Gnd

FLOORPLAN

The following figure indicates the general location of each functional block within the chip and their corresponding sizes.

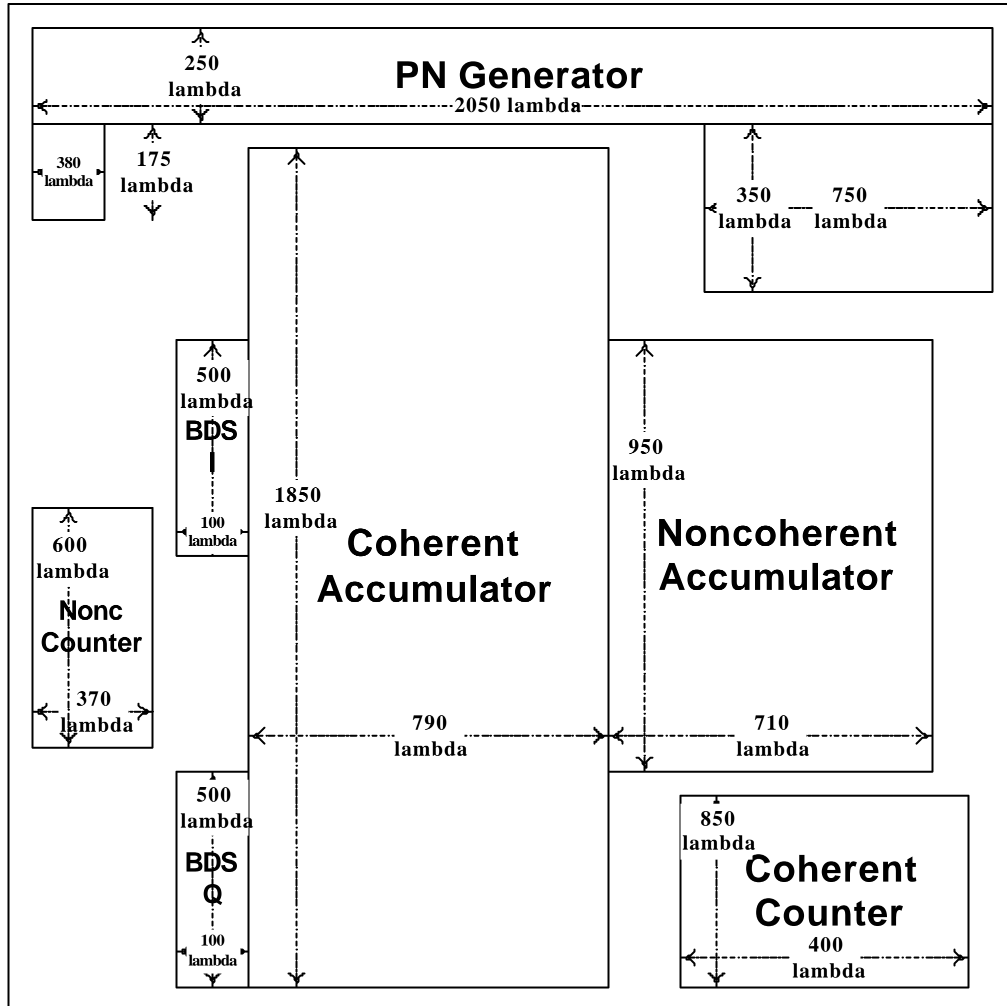


Figure 4: Chip floorplan

Table 3: Occupied area/ Time spent

Functional Block	Total Area (sq. lambda)	Time Spent (hours)
Flop	12,900	1
Xor2	6,300	2
Pn_gen_or_flop	18,900	1
Pn_gen_10_lfsr	185,000	2
Pn_gen_mux	63,500	2
Pn_gen	841,500	6
Coh_adder_1bit	20,490	0
Accumulator_reg	37,000	1
Datapath_energy_1bit	63,000	1
Datapath_energy	663,000	4
Datapath_coherent_1bit	44,200	1
Datapath_coherent	467,000	2
Datapath_bds	33,000	1
Datapath_iq	500,000	1
Datapath	2,005,000	3
Counter_reg_reset	31,500	1
4bit_counter	130,000	1
8bit_counter	293,000	1
Noncoherent Counter	222,000	2
Coherent Counter	340,000	2
Searcher (w/ connections to pads)	4,650,000 (4,940,000)	4 (10)

Total Occupied Area = 4,650,000 sq. lambda

SIMULATION

To simulate this design, we used MATLAB to produce test vectors. Below are the steps necessary to run the MATLAB simulation:

- Use the MATLAB code developed for the Texas Instruments clinic project 2000/2001 to generate GPS data. The command is:

  Searcher(satid, power, offset)

  Thus, to simulate satellite 30 at offset 5, type

  Searcher(30, 1, 5)

  This will generate a data file called rci_rxif_iq_0.in.

- Run the MATLAB code vlsi_searcher.m. This code performs several functions:
  - Conv_vlsi.m **:** Converts the rci_rxif_iq_0.in file to iq_data.in. The function keeps one sample out of the 8 samples in the former file;
  - Runs simulation: Using loops for the coherent and noncoherent counters, it reads the iq_data.in file and generates the expected energy output of our chip;
  - Vlsi.m: This function creates a *searcher.cmd* file that must be copied to the IRSIM directory to run the simulation. The function takes care of setting all parameters and sending the data from the iq_data.in file.

  The syntax is: vlsi_searcher(satid, N, L, offset)

  Thus, to simulate satellite 30 at offset 5 (with 129 coherent accumulations and 4 noncohorent accumulations), type

Vlsi_searcher(30,3,3,5)

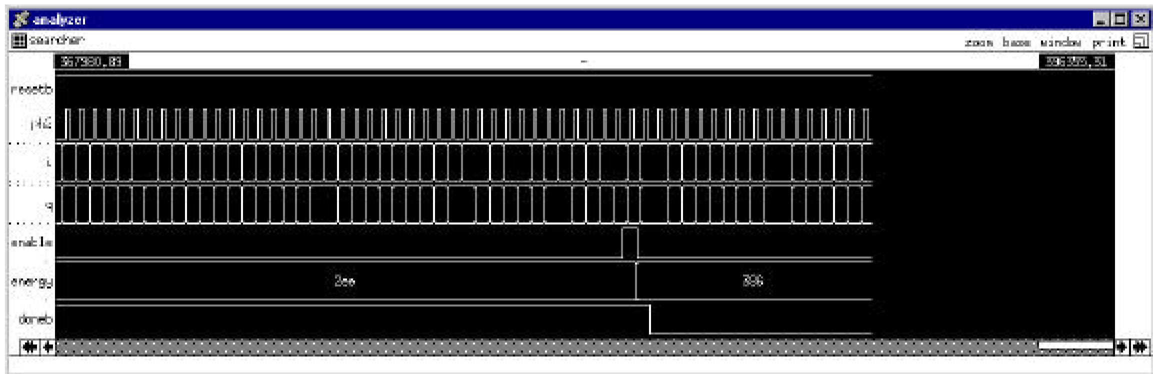
- Finally, copy the generated *searcher.cmd* file and run it with IRSIM.

NOTE: See appendix A for the source code of vlsi.m and vlsi_searcher.m.

Simulation Results

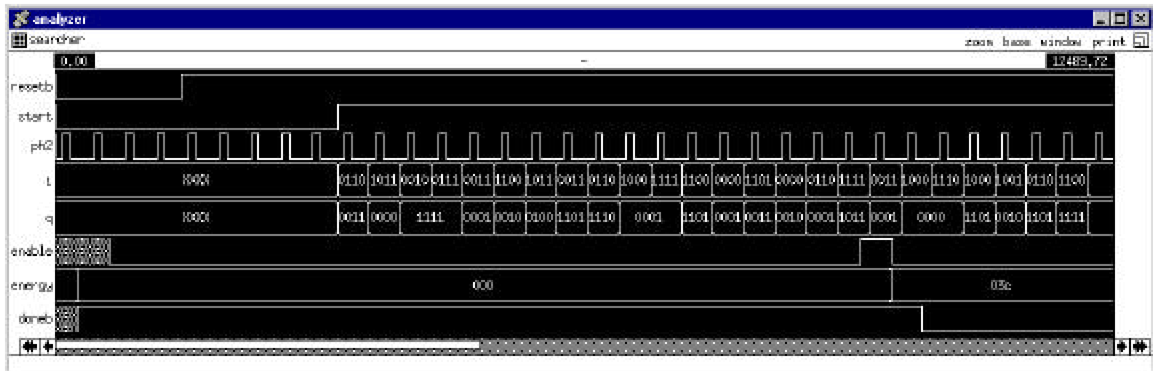
Test 1:

Variable/Result	Value
Satellite ID	1
Offset	0
N (# of coherent accumulations)	3
L (# of noncoherent accumulations)	3
MATLAB result	902 (dec) = 386 (hex)



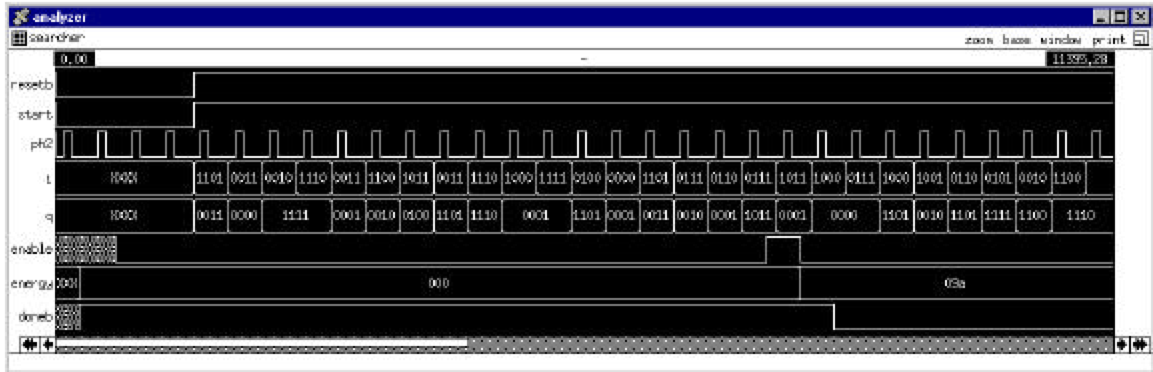
Test 2:

Variable/Result	Value
Satellite ID	30
Offset	5
N (# of coherent accumulations)	0
L (# of noncoherent accumulations)	0
MATLAB result	60 (dec) = 3C (hex)



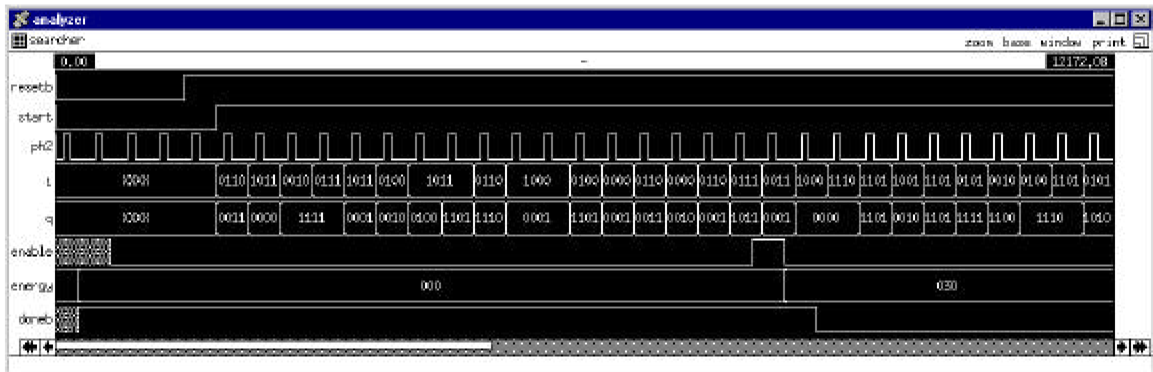
Test 3:

Variable/Result	Value
Satellite ID	5
Offset	0
N (# of coherent accumulations)	0
L (# of noncoherent accumulations)	0
MATLAB result	154 (dec) = 9A (hex)



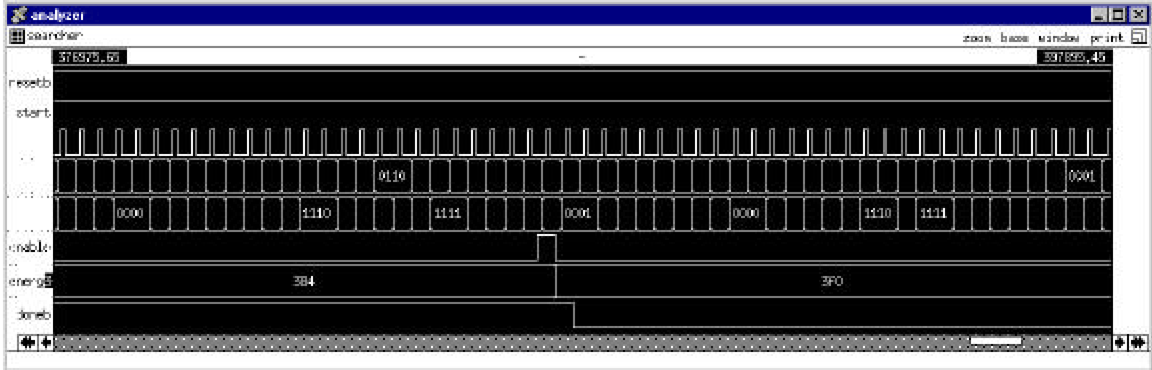
Test 4:

Variable/Result	Value
Satellite ID	5
Offset	1
N (# of coherent accumulations)	0
L (# of noncoherent accumulations)	0
MATLAB result	48 (dec) = 30 (hex)



Test 5:

Variable/Result	Value
Satellite ID	5
Offset	9
N (# of coherent accumulations)	3
L (# of noncoherent accumulations)	3
MATLAB result	1008 (dec) = 3F0 (hex)



POSTFABRICATION TESTING

We (Sergio!) will use the DN2000K10 board to test the chip when it returns from fabrication. We decided to use this board since it is well documented and the C++ interface code is already capable of sending I and Q data from the rci_rxif_iq_0.in file. We realize the board is overkill for the project, but it seems to be the simpler way.

The Virtex FPGA in the board will be used as a buffer to our chip, passing the input signals from the PCI interface to the chip. As for the output signals, we can simply verify them with a logic analyzer.

We are using 2 pads in the chip to verify that the chip was manufactured correctly: input *in* and output *out*. The output is the inverse of the input. Sergio will check this at first to make sure the chip is good.

List of Necessary Changes to DN2000K10 board

- **PCI Interface Verilog code:**
    - Sergio can use the code provided as an example in the TI project final report;
    - Create the following inputs:
        - ph1, ph2, resetb, start, i, q, s, N and L
    - Note: Outputs will go directly to logic analyzer
- **C++ Interface code:**
    - Modify the send_IQ() function to:
        - Read the iq_data.in file (instead of rci_rxif_iq_0.in);
        - Send 4 MSBs for I and Q (instead of all 5 bits);
        - Produces ph1 and ph2 (instead of a single clock).

- **Batch File:**
    - Create a batch file that programs satid, N and L automatically.

NOTE: Fernando Mattos will be available for contact via email at fbmattos@earthlink.net or Fernando_mattos@hmc.edu.

VERIFICATION RESULTS

Electric

DRC

Checking... (type Windows-C to abort)
Checking facet searcher{lay}
   No errors found (20.97 seconds so far)
0 errors found (took 20.97 seconds)

ERC

Checking electrical rules... (type Windows-C to abort)
Farthest distance from a P-Well contact is 58.6006
Farthest distance from a N-Well contact is 78.3662
1997 nodes of type P-Transistor
1997 nodes of type N-Transistor
1999 networks found

No ERC errors found

NCC

--- Comparing facet inv{lay} (2 components, 2 nets) with facet inv{sch} (2 components, 2 nets)
Facets are equivalent (took 0 seconds)
--- Comparing facet tri{lay} (4 components, 6 nets) with facet tri{sch} (4 components, 6 nets)
Facets are equivalent (took 0 seconds)
--- Comparing facet mux2{lay} (2 components, 5 nets) with facet mux2{sch} (2 components, 5 nets)
Facets are equivalent (took 0 seconds)
--- Comparing facet mux4{lay} (3 components, 11 nets) with facet mux4{sch} (3 components, 11 nets)
Facets are equivalent (took 0 seconds)
--- Comparing facet latch{lay} (12 components, 8 nets) with facet latch{sch} (12 components, 8 nets)
Facets are equivalent (took 0 seconds)
Cannot find schematic view of facet latch2{lay}
--- Comparing facet flop{lay} (2 components, 5 nets) with facet flop{sch} (2 components, 5 nets)
Facets are equivalent (took 0 seconds)
--- Comparing facet nand2{lay} (4 components, 4 nets) with facet nand2{sch} (4 components, 4 nets)

Facets are equivalent (took 0 seconds)
--- Comparing facet and2{lay} (2 components, 4 nets) with facet and2{sch} (2 components, 4 nets)
Facets are equivalent (took 0 seconds)
--- Comparing facet xor2{lay} (7 components, 5 nets) with facet xor2{sch} (7 components, 5 nets)
Facets are equivalent (took 0 seconds)
--- Comparing facet counter_reg_reset{lay} (4 components, 8 nets) with facet counter_reg_reset{sch} (4 components, 8 nets)
Facets are equivalent (took 0 seconds)
--- Comparing facet 8bit_counter{lay} (8 components, 19 nets) with facet 8bit_counter{sch} (8 components, 19 nets)
Facets are equivalent (took 0 seconds)
--- Comparing facet coh_dwell_counter{lay} (8 components, 20 nets) with facet coh_dwell_counter{sch} (8 components, 20 nets)
Facets are equivalent (took 0 seconds)
--- Comparing facet nand3{lay} (6 components, 6 nets) with facet nand3{sch} (6 components, 6 nets)
Facets are equivalent (took 0 seconds)
--- Comparing facet coh_adder_and_inv{lay} (2 components, 5 nets) with facet coh_adder_and_inv{sch} (2 components, 5 nets)
Facets are equivalent (took 0 seconds)
--- Comparing facet fulladder{lay} (28 components, 17 nets) with facet fulladder{sch} (28 components, 17 nets)
Facets are equivalent (took 0 seconds)
--- Comparing facet accumulator_reg{lay} (4 components, 11 nets) with facet accumulator_reg{sch} (4 components, 11 nets)
Facets are equivalent (took 0 seconds)
--- Comparing facet coh_adder_1bit{lay} (2 components, 8 nets) with facet coh_adder_1bit{sch} (2 components, 8 nets)
Facets are equivalent (took 0 seconds)
--- Comparing facet datapath_energy_1bit{lay} (2 components, 13 nets) with facet datapath_energy_1bit{sch} (2 components, 13 nets)
Facets are equivalent (took 0 seconds)
--- Comparing facet datapath_energy{lay} (11 components, 58 nets) with facet datapath_energy{sch} (11 components, 58 nets)
Facets are equivalent (took 0 seconds)
--- Comparing facet datapath_bits5to8{lay} (2 components, 10 nets) with facet datapath_bits5to8{sch} (2 components, 10 nets)
Facets are equivalent (took 0 seconds)
--- Comparing facet datapath_bit9{lay} (4 components, 11 nets) with facet datapath_bit9{sch} (4 components, 11 nets)
Facets are equivalent (took 0 seconds)
--- Comparing facet datapath_coherent{lay} (10 components, 35 nets) with facet datapath_coherent{sch} (10 components, 35 nets)
Facets are equivalent (took 0 seconds)

--- Comparing facet datapath_bds{lay} (5 components, 11 nets) with facet datapath_bds{sch} (5 components, 11 nets)
Facets are equivalent (took 0 seconds)
--- Comparing facet datapath_iq{lay} (2 components, 25 nets) with facet datapath_iq{sch} (2 components, 25 nets)
Facets are equivalent (took 0 seconds)
--- Comparing facet datapath{lay} (3 components, 47 nets) with facet datapath{sch} (3 components, 47 nets)
Facets are equivalent (took 0 seconds)
--- Comparing facet 4bit_counter{lay} (4 components, 12 nets) with facet 4bit_counter{sch} (4 components, 12 nets)
Facets are equivalent (took 0 seconds)
--- Comparing facet nor2{lay} (4 components, 4 nets) with facet nor2{sch} (4 components, 4 nets)
Facets are equivalent (took 0 seconds)
--- Comparing facet or2{lay} (2 components, 4 nets) with facet or2{sch} (2 components, 4 nets)
Facets are equivalent (took 0 seconds)
--- Comparing facet noncoh_dwell_counter{lay} (8 components, 18 nets) with facet noncoh_dwell_counter{sch} (8 components, 18 nets)
Facets are equivalent (took 0 seconds)
--- Comparing facet pn_gen_or_flop{lay} (2 components, 6 nets) with facet pn_gen_or_flop{sch} (2 components, 6 nets)
Facets are equivalent (took 0 seconds)
--- Comparing facet pn_gen_10lfsr{lay} (10 components, 14 nets) with facet pn_gen_10lfsr{sch} (10 components, 14 nets)
Facets are equivalent (took 0 seconds)
--- Comparing facet pn_gen_mux{lay} (9 components, 23 nets) with facet pn_gen_mux{sch} (9 components, 23 nets)
Facets are equivalent (took 0 seconds)
--- Comparing facet pn_gen{lay} (13 components, 42 nets) with facet pn_gen{sch} (13 components, 42 nets)
Facets are equivalent (took 0 seconds)
--- Comparing facet searcher{lay} (4 components, 38 nets) with facet searcher{sch} (4 components, 38 nets)
Facets are equivalent (took 0 seconds)

NOTE: Flatten NCC did not work because of an Electric bug.

Gemini

Gemini 2.7.3 1994/10/12

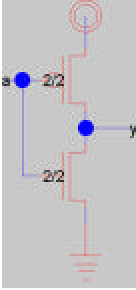
Graph "test3.sim": unit scale = 80, format 'SU' is unknown (assume MIT)
format = NULL (assume MIT)

```
format = NULL (assume MIT)
format = NULL (assume MIT)
format = NULL (assume MIT)

…  …  …
… … …

format = NULL (assume MIT)
format = NULL (assume MIT)
        Number of devices: 3267
        Number of nets: 1276

Graph "test_sch.sim": unit scale = 200, format 'SU' is unknown (assume MIT)
format = NULL (assume MIT)
        Number of devices: 3267
        Number of nets: 1276
4407 (96%) matches were found by local matching
All nodes were matched in 172 passes
```

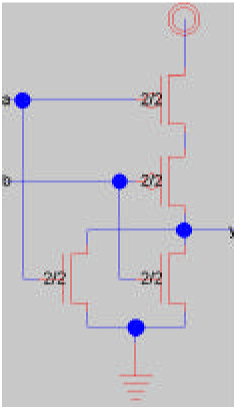
SCHEMATICS

Basic Cells

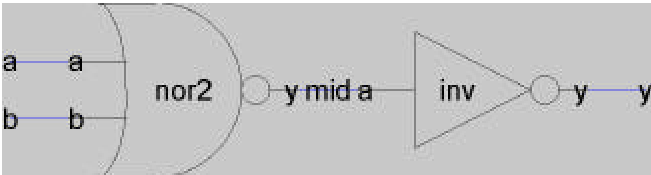
inv



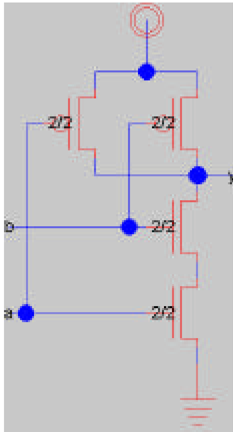
nor2



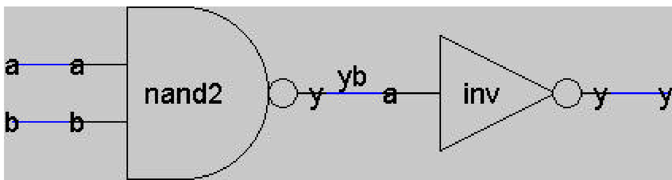
or2



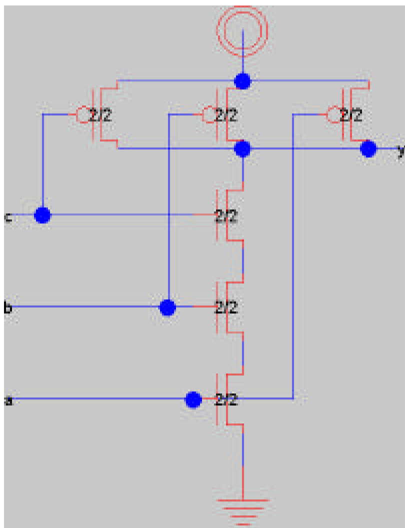
nand2



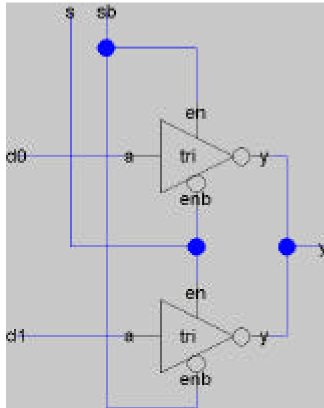
and2



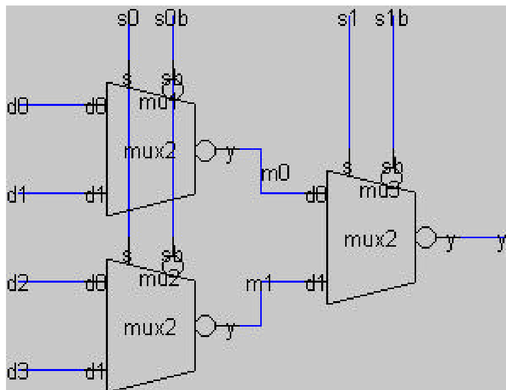
nand3



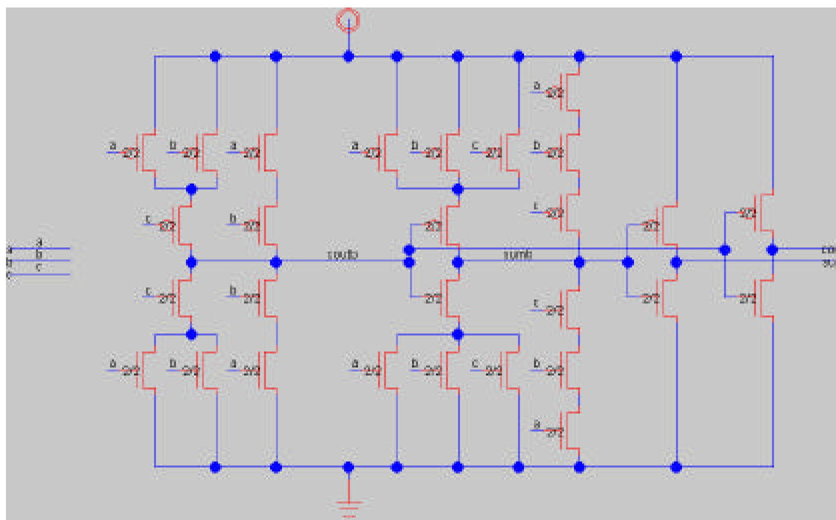
mux2



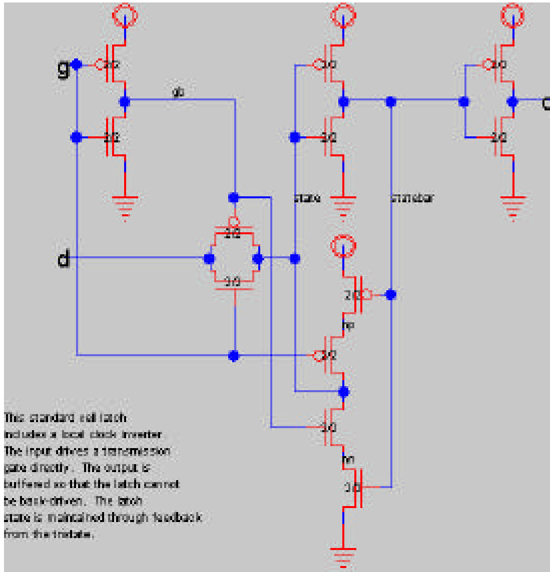
mux4



fulladder



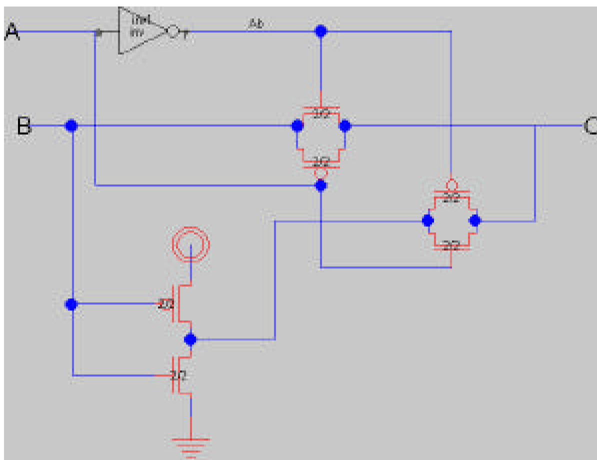
latch



flop

This facet creates a two-phase clocked "flop"

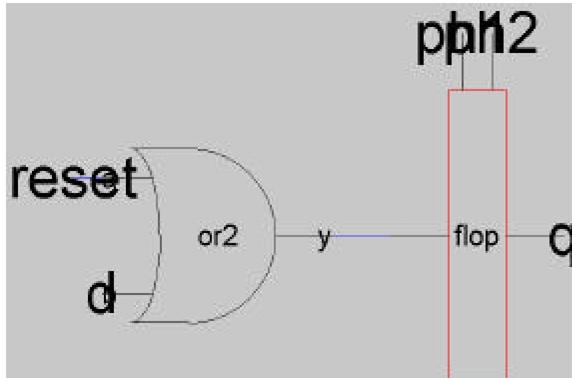
xor2



PN Generator Cells

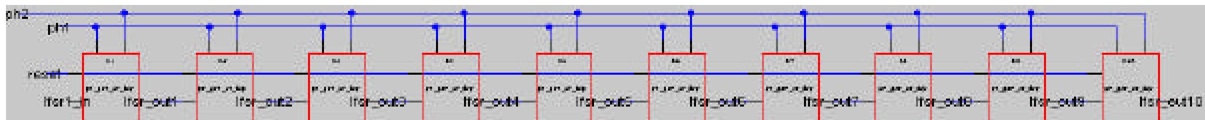
pn_gen_or_flop

This facet simply makes a resettable flip-flop



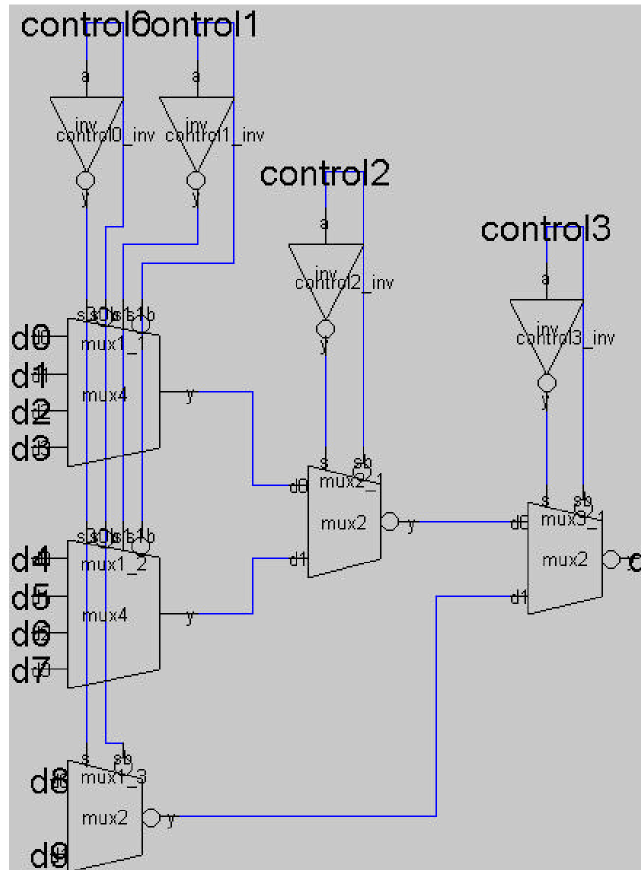
pn_gen_10_lfsr

10 resettable flops connected in series to create a 10-bit LFSR



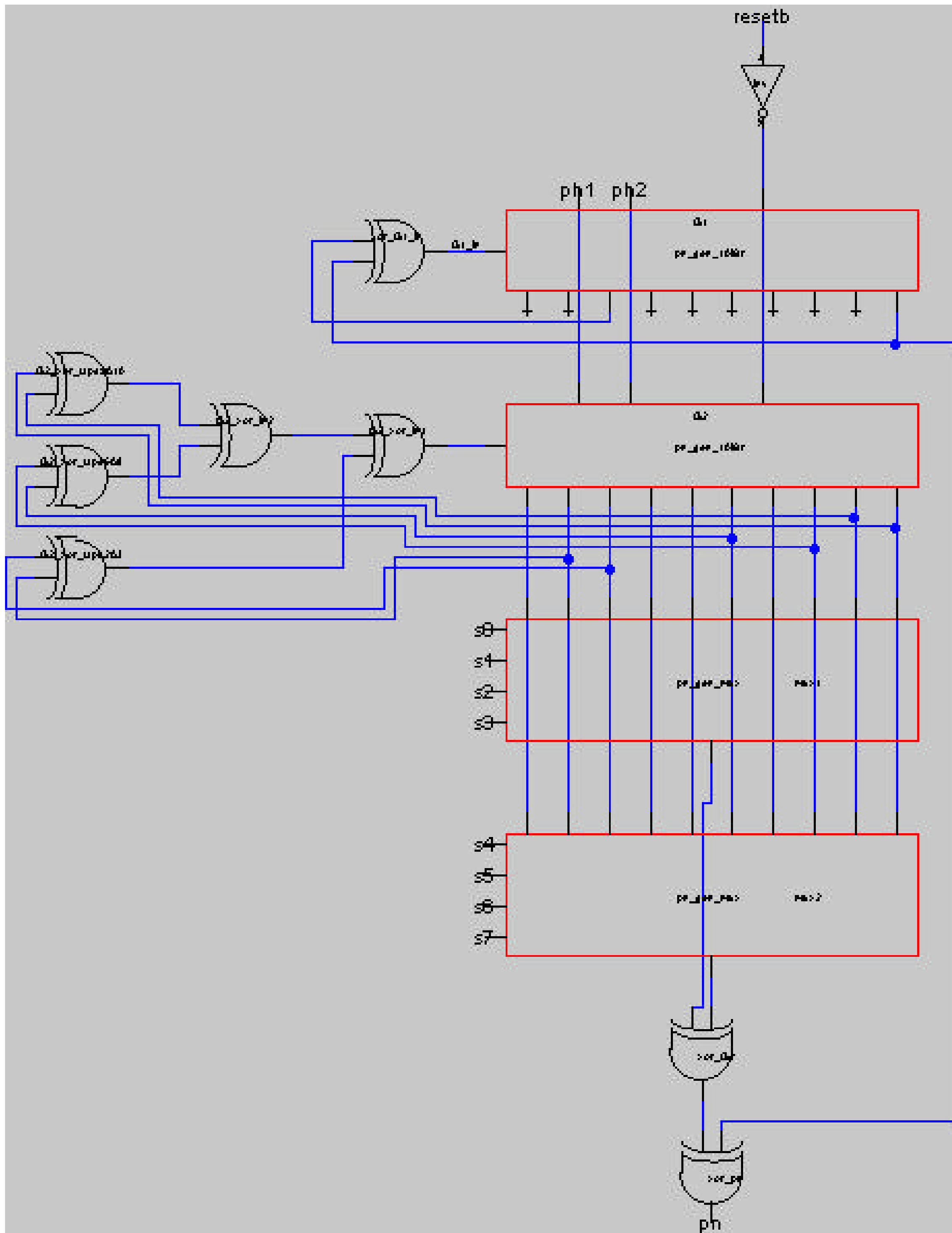
pn_gen_mux

mux with 4-bit select to tap 10 bits from the second LFSR



pn_gen

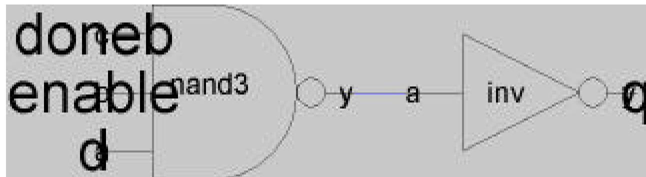
PN generator cell that performs the algorithm represented in Fig. 2 - The bottom two modules are the 10-bit muxes that tap from the second LFSR



Datapath Cells

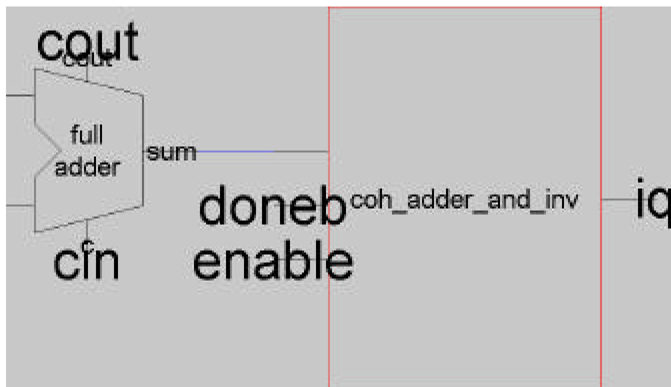
coh_add_and_inv

Noncoherent accumulations will not receive data unless the searcher is both "not done" and the enable interrupt is set

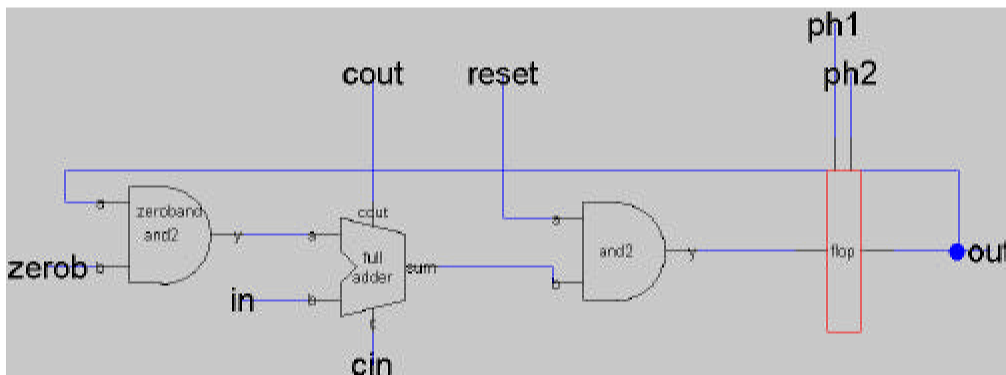


coh_adder_1bit

This facet simply makes a 1-bit adder with the above characteristics

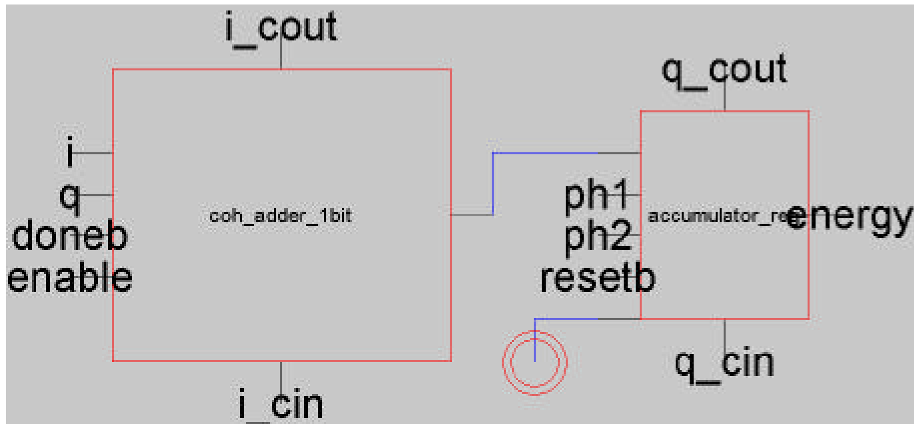


accumulator_reg



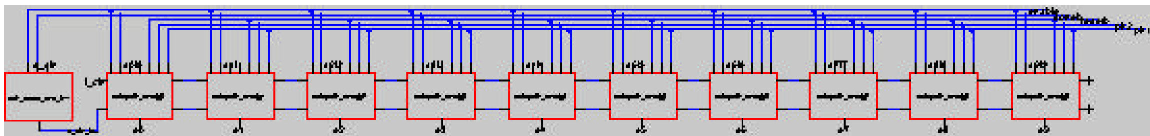
datapath_energy_1bit

Combining the coherent and noncoherent accumulations to simplify layout

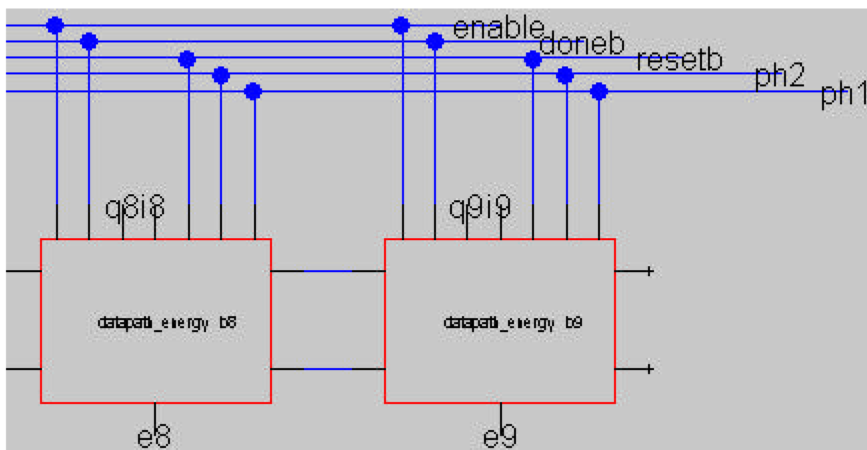


Datapath_energy

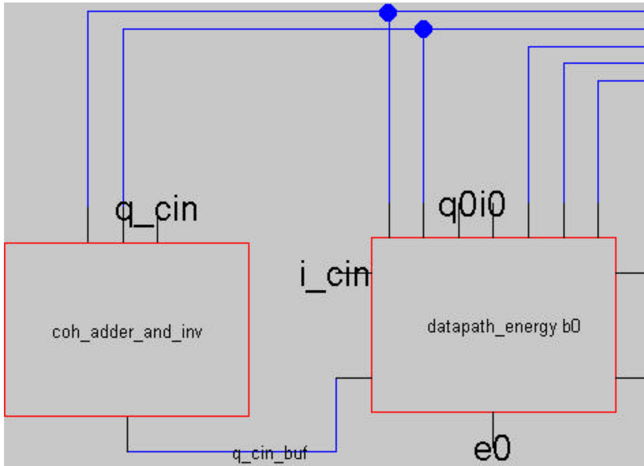
Connects 10 bits of datapath_energy



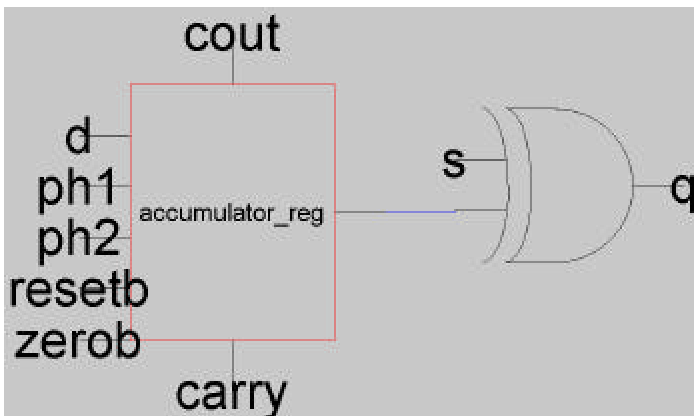
Zoom of right portion



Zoom of left portion (datapath_energy)

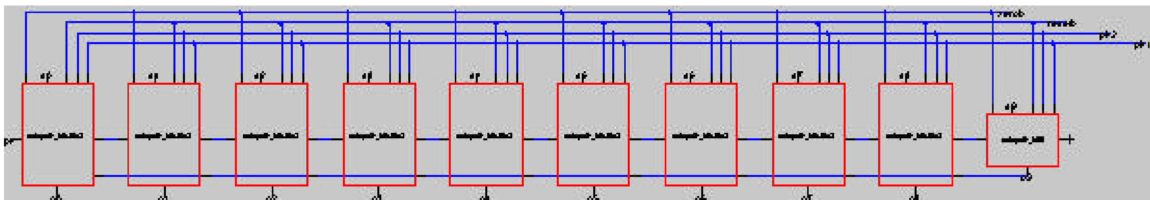


<u>datapath_coherent_1bit</u>

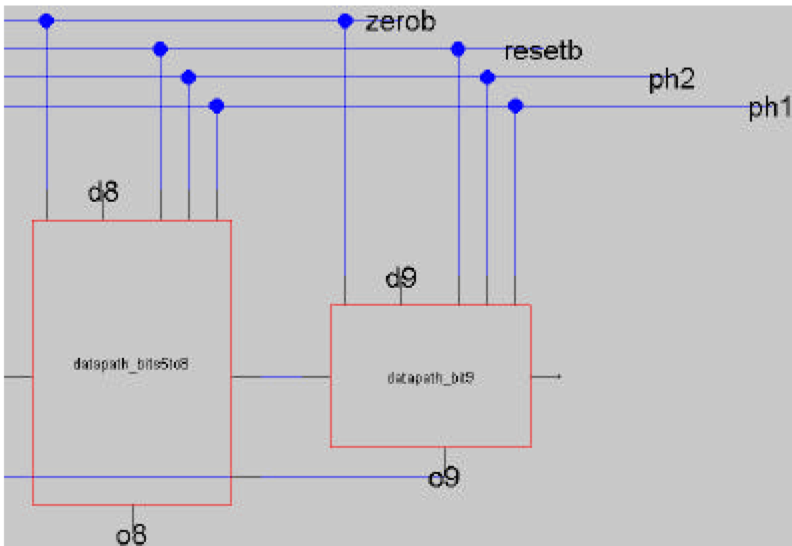


<u>datapath_coherent</u>

Connects facets for 10-bit coherent accumulator

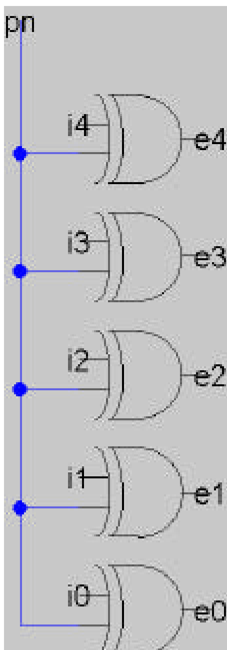


Zoom of right portion (datapath_coherent)



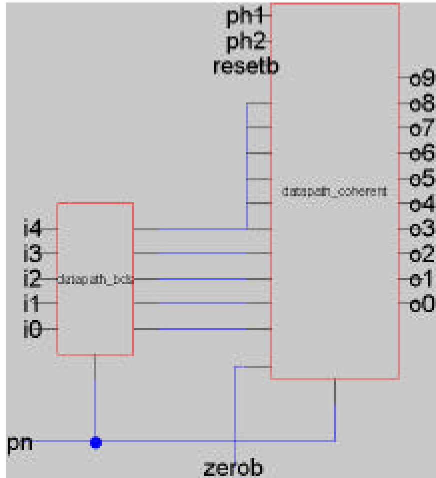
<u>datapath_bds</u>

Correlator of PN code with each I or Q bit



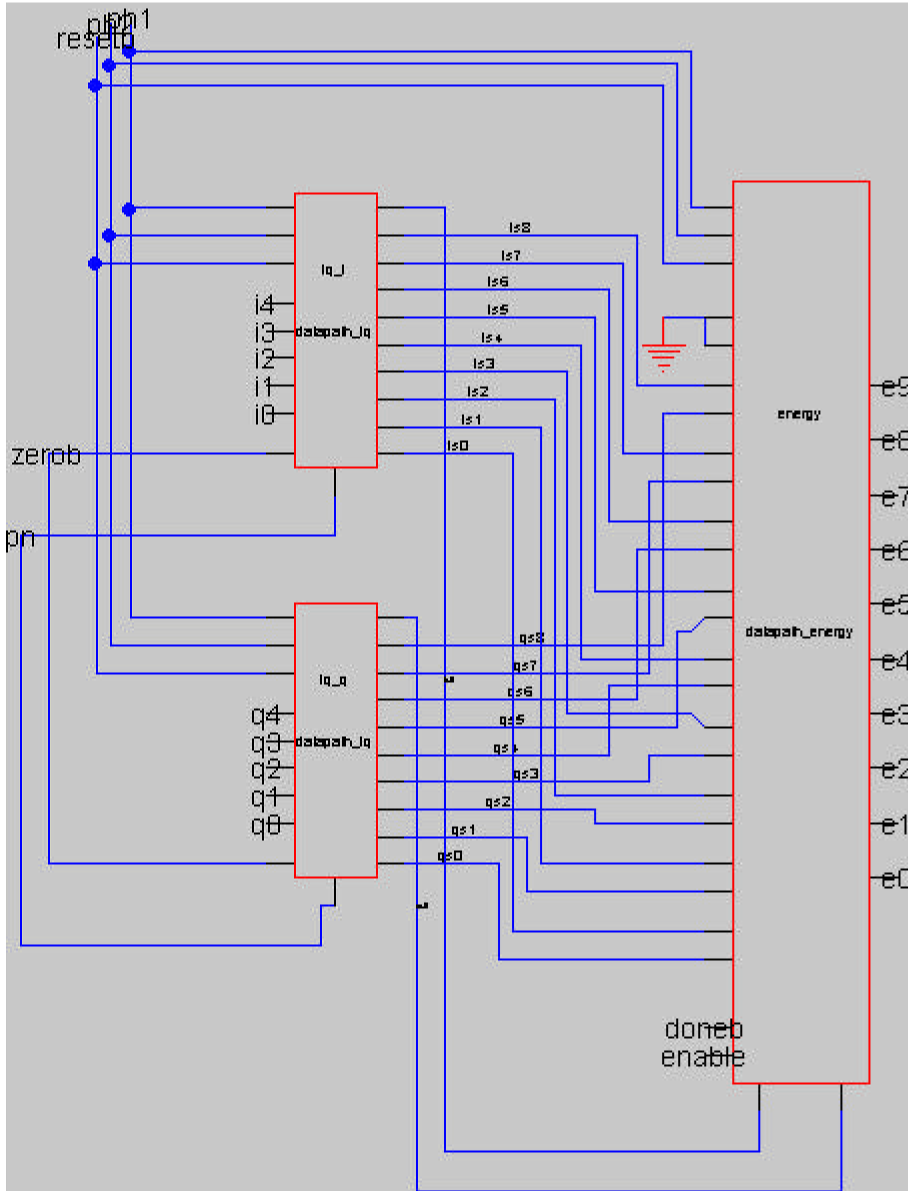
datapath_iq

Connects bds with coherent accumulator – the top 5 bits of the accumulator are the same as the top bit of the incoming I or Q signal to be consistent with 2's complement



Datapath

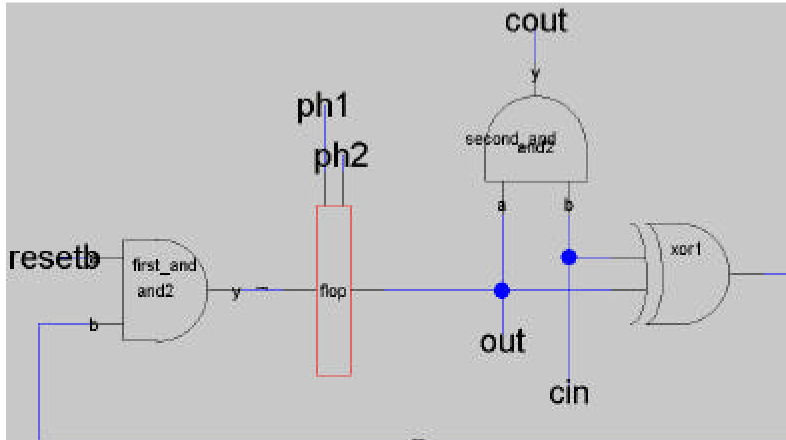
Connects bds with coherent accumulator with noncoherent accumulator



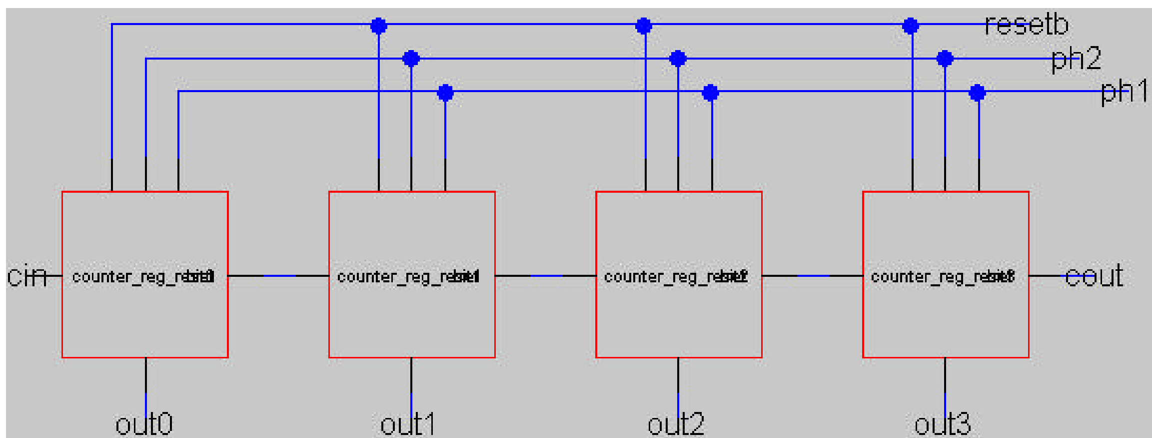
Coherent/Non Coherent Dwell Counter Cells

Counter_reg_reset

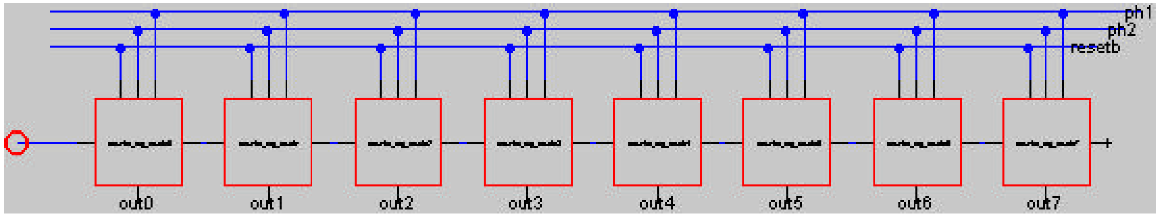
1-bit resettable counter



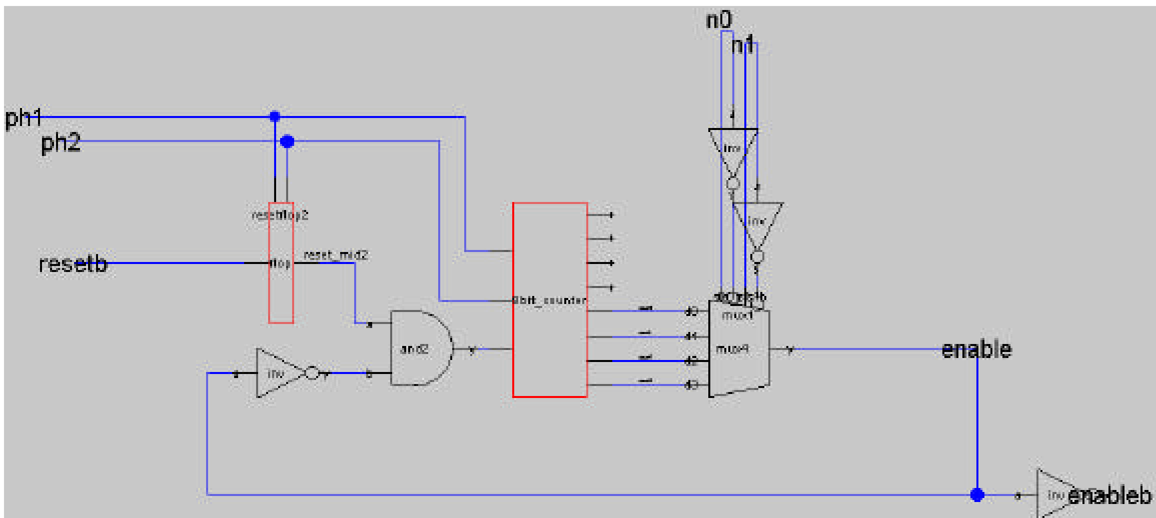
4-bit counter



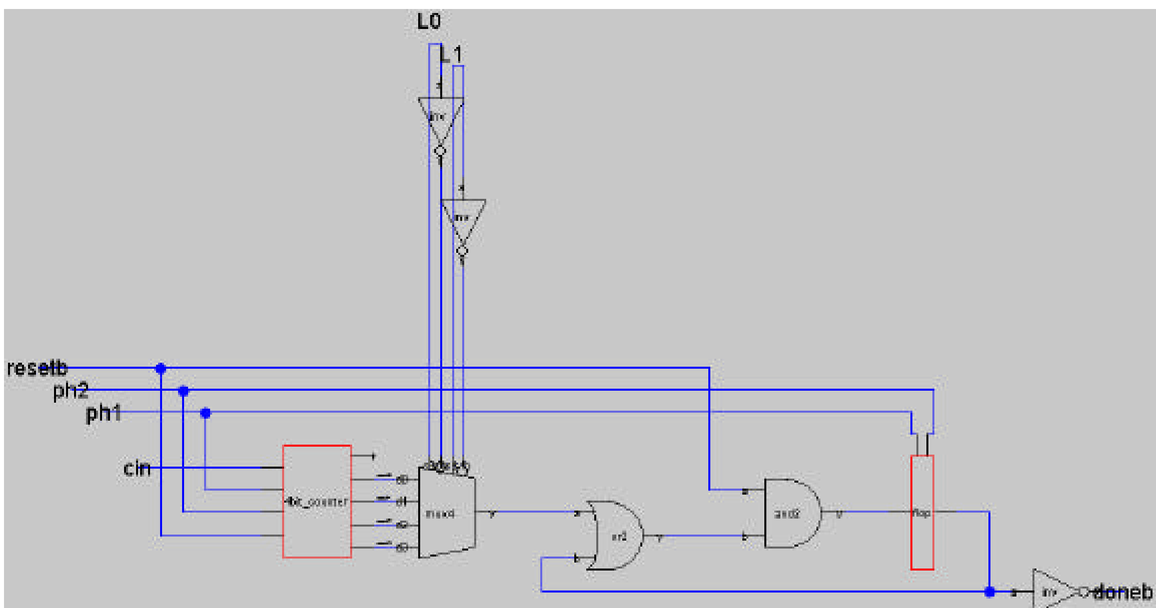
8-bit counter



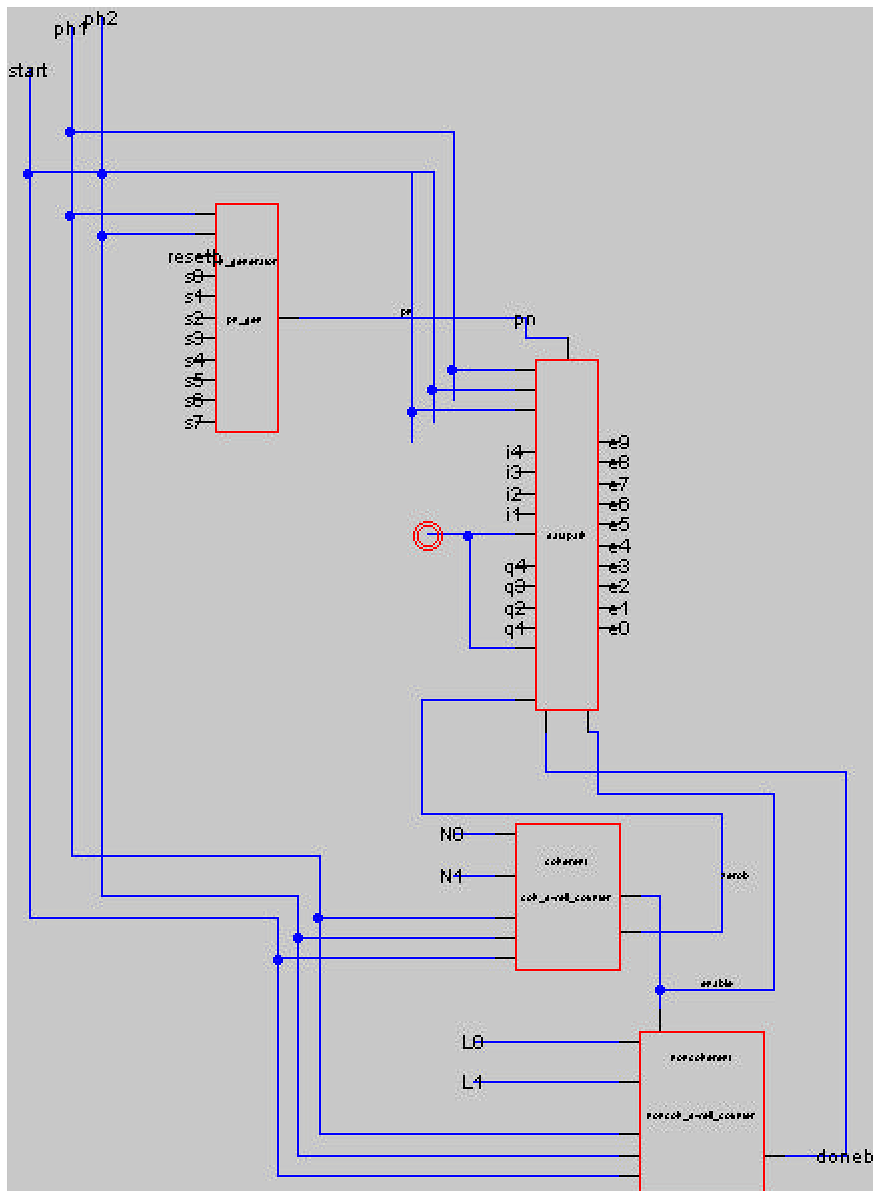
coh_dwell_counter



noncoh_dwell_counter



Searcher

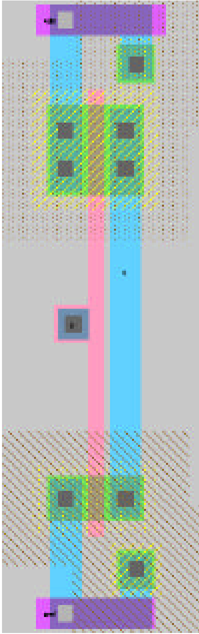


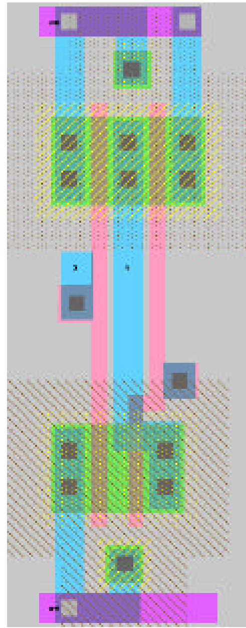
LAYOUT

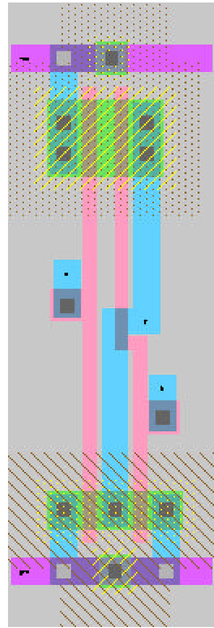
Basic cells

inv/nand2/or2

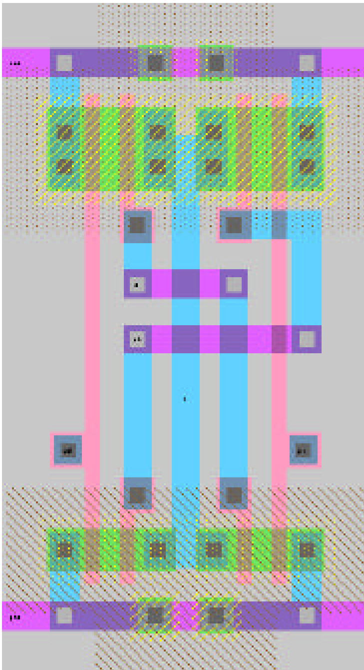
inv nand2 or2



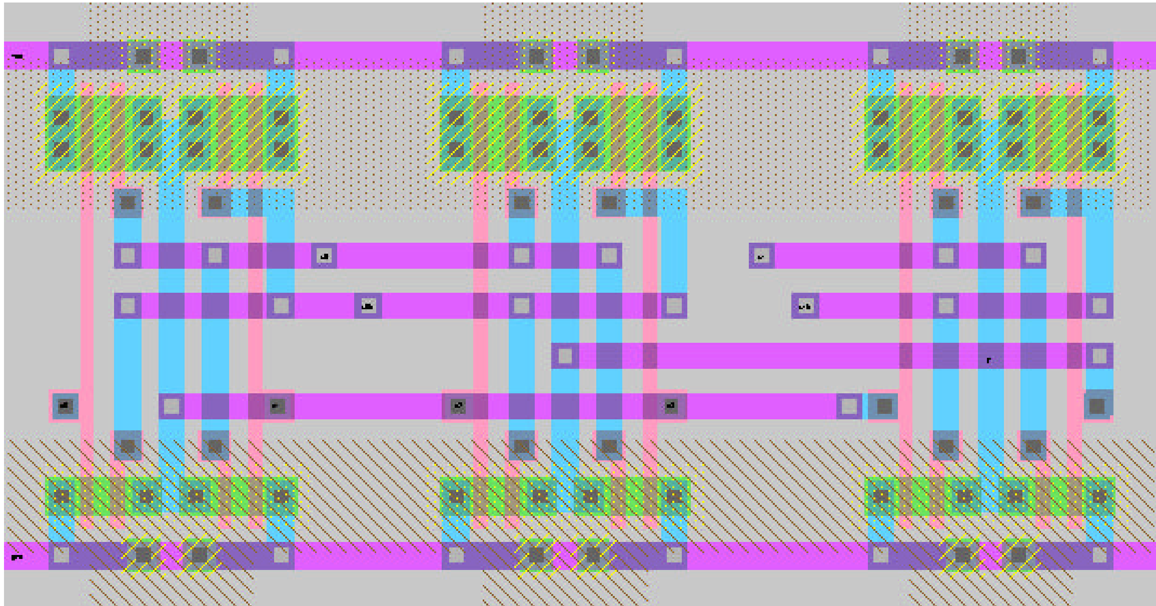




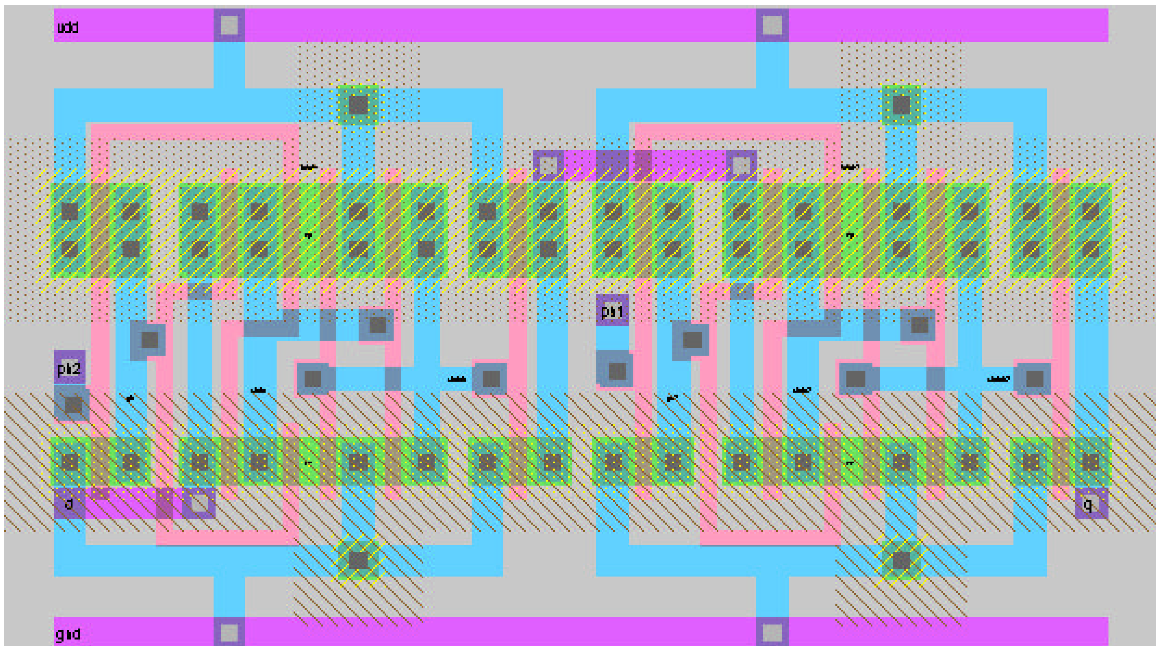
mux2



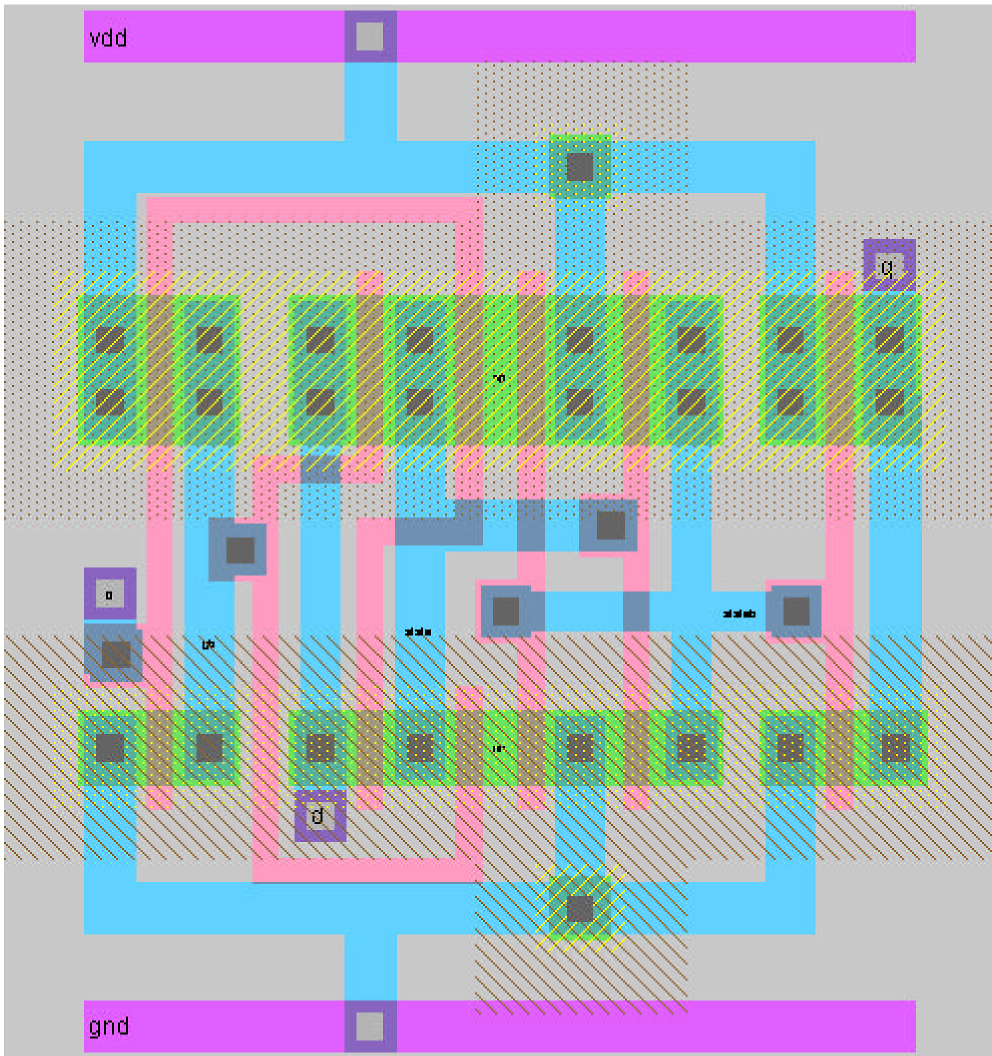
mux 4



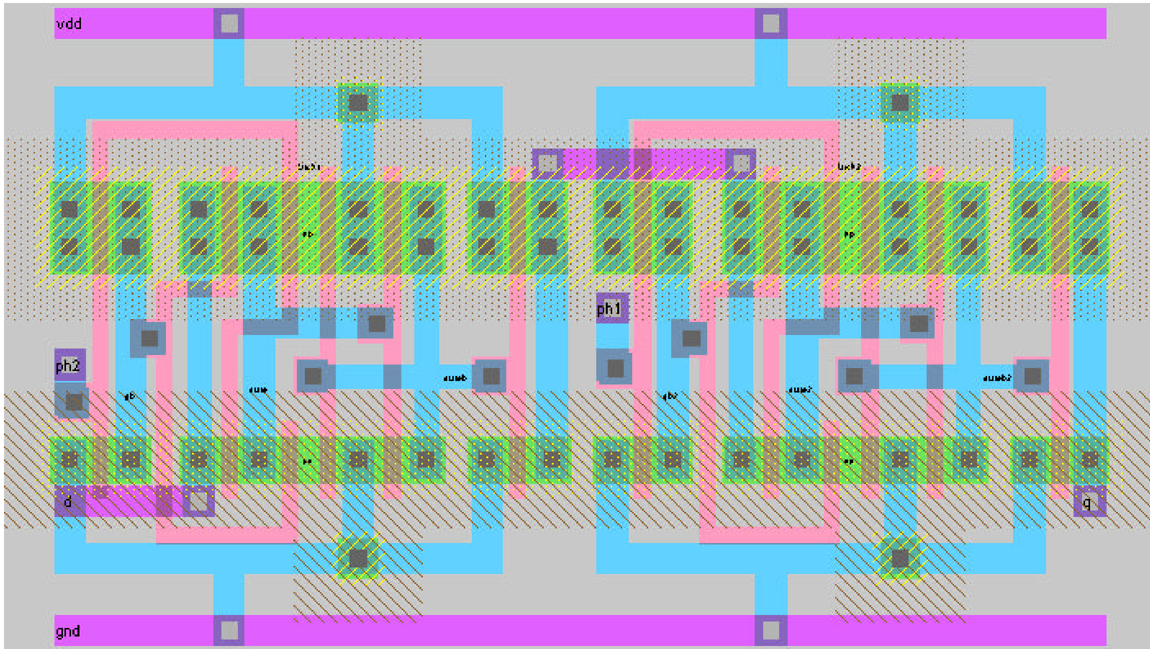
fulladder



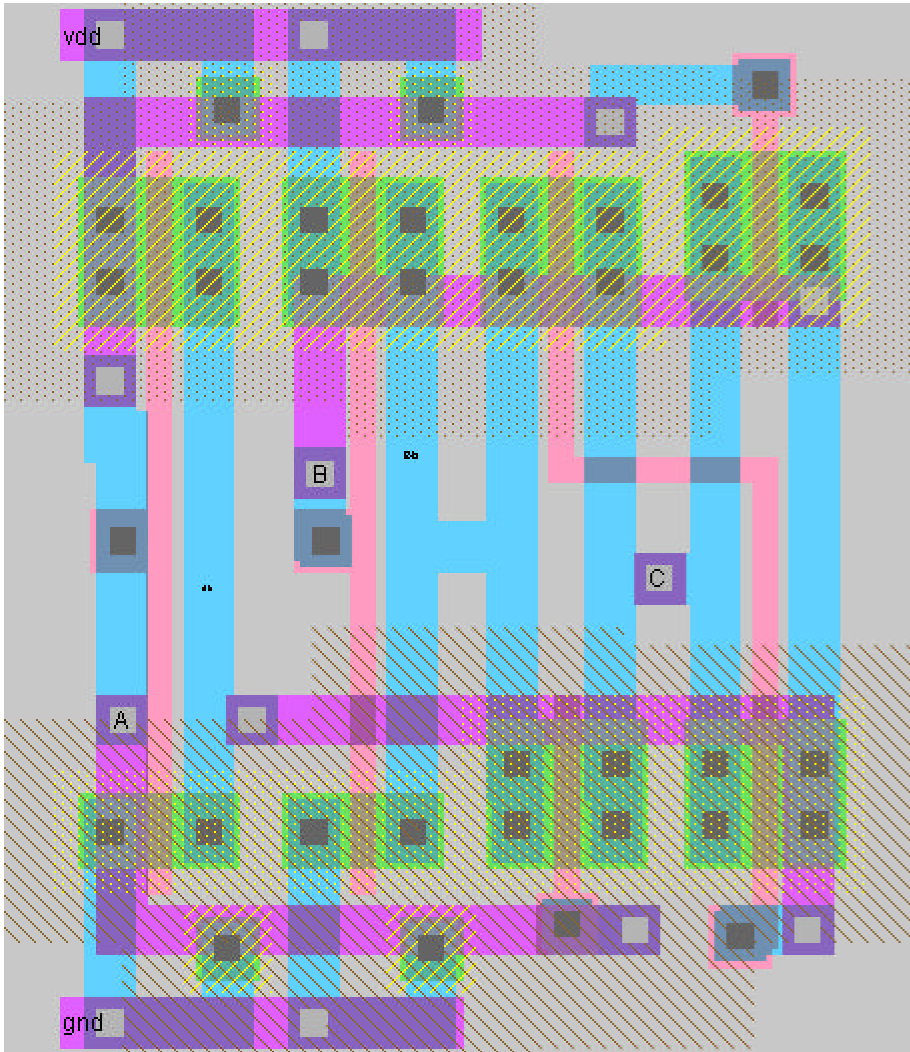
latch



flop

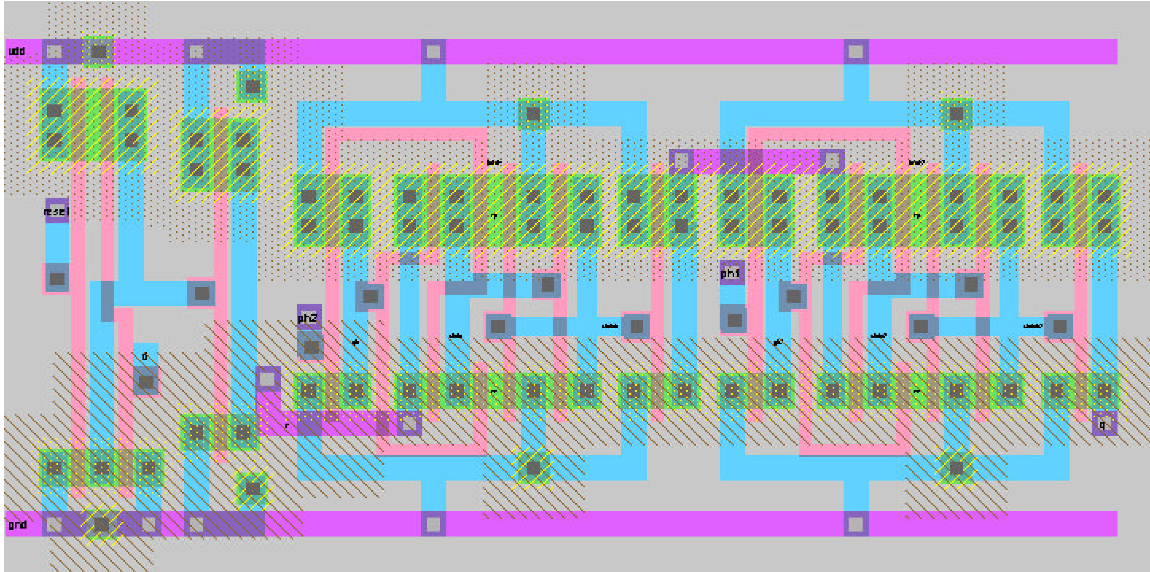


xor2

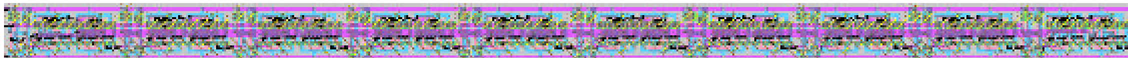


PN Generator Cells

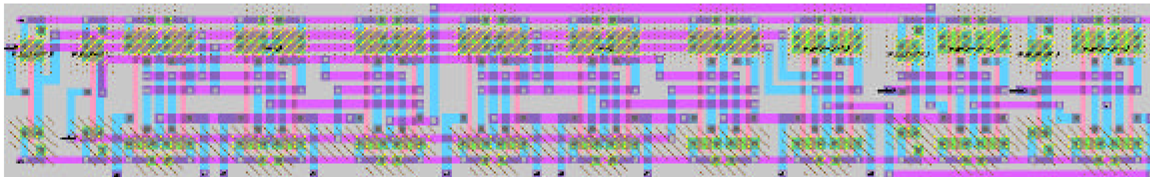
pn_gen_or_flop



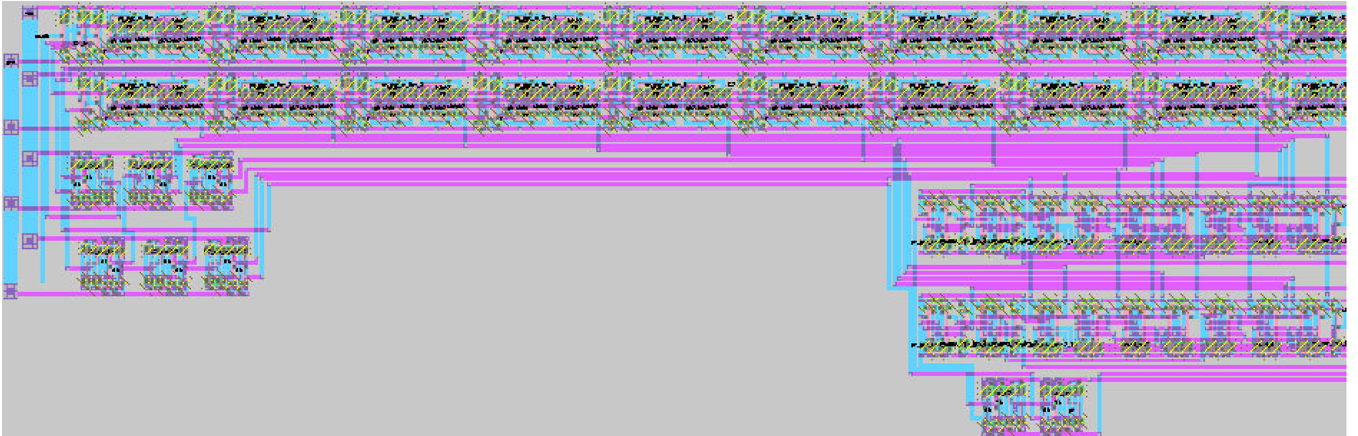
pn_gen_10_lfsr



pn_gen_mux

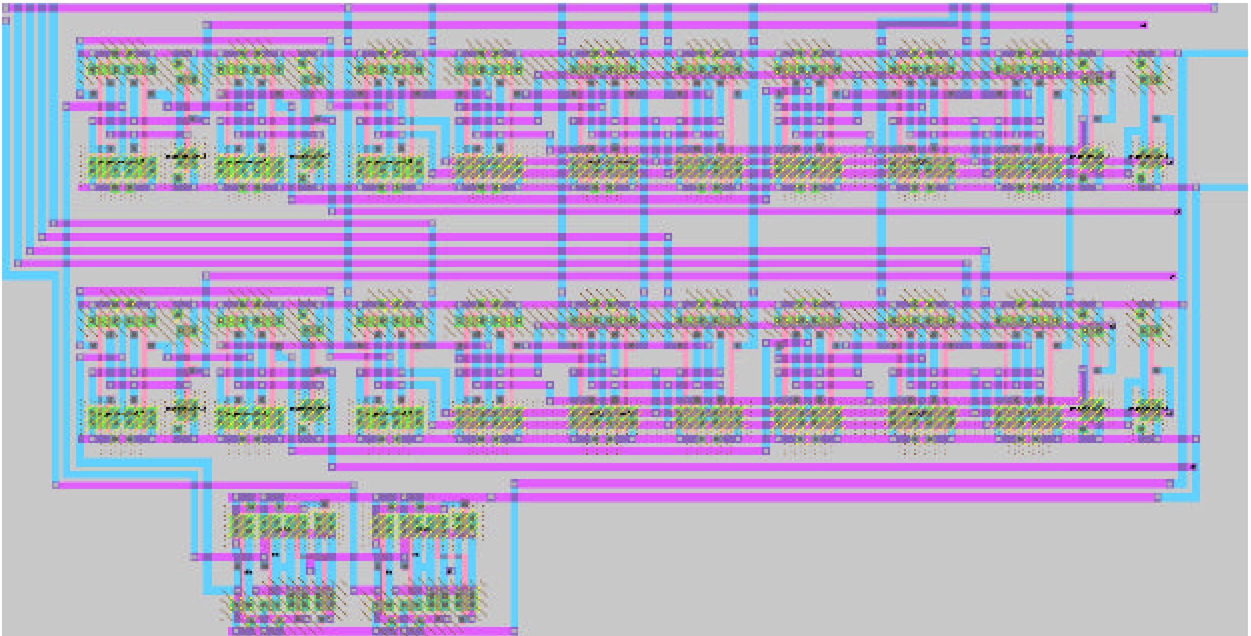


pn_gen



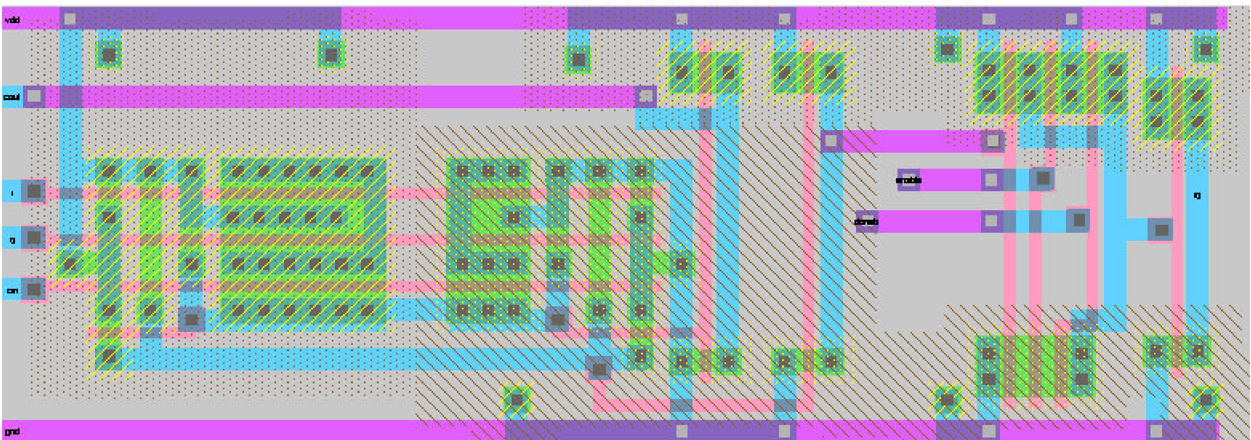
Zoom of bottom left portion of pn_gen (6 XOR's)

Zoom of bottom right portion of pn_gen (2 10-bit MUXES and 2 XOR's)

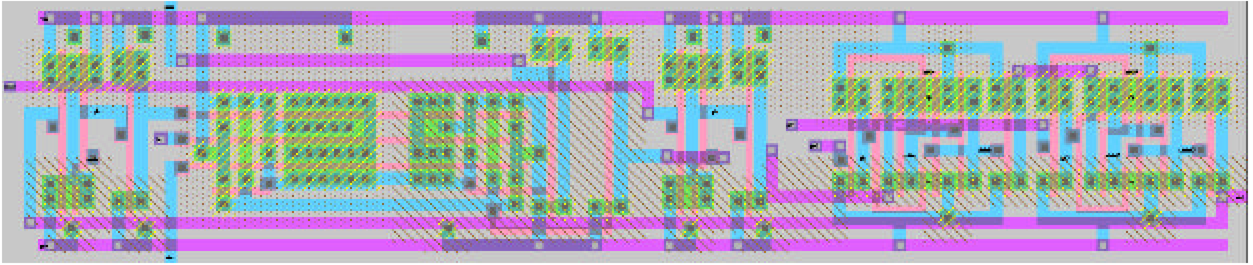


Datapath Cells

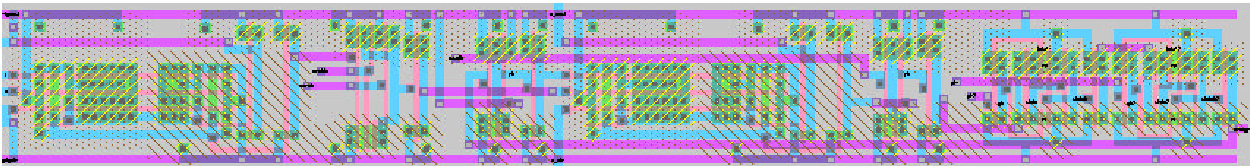
<u>coh_adder_1bit</u>



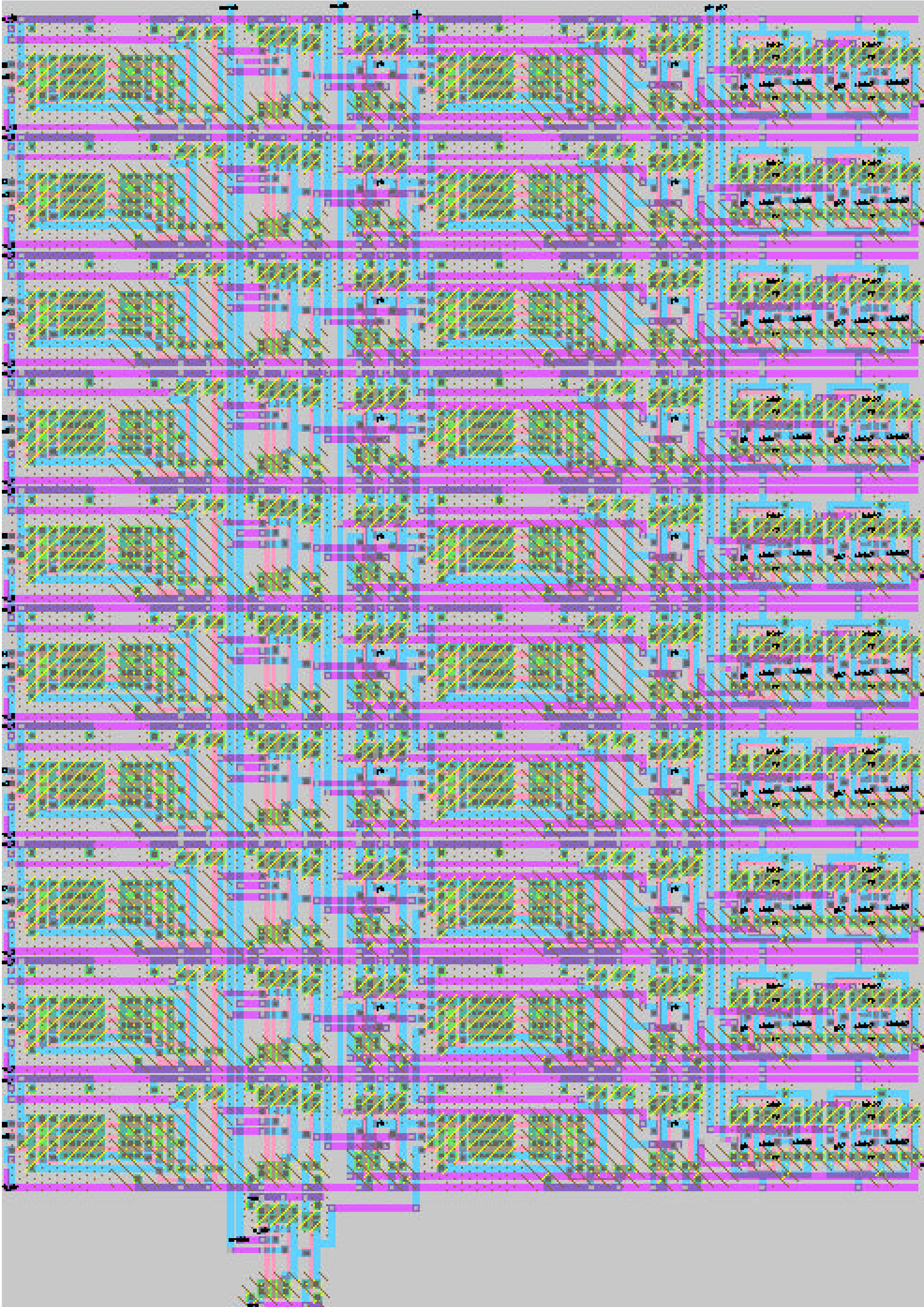
accumulator_reg



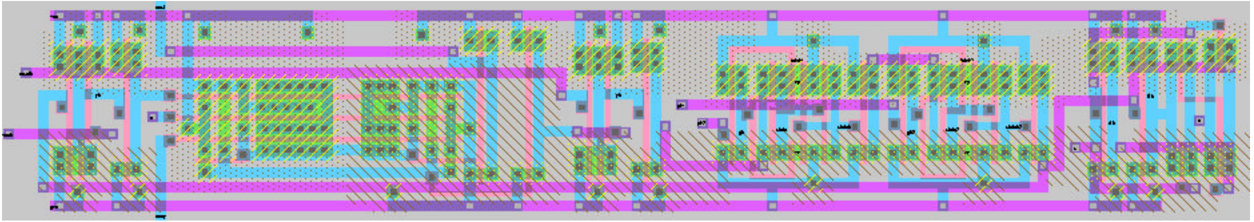
datapath_energy_1bit



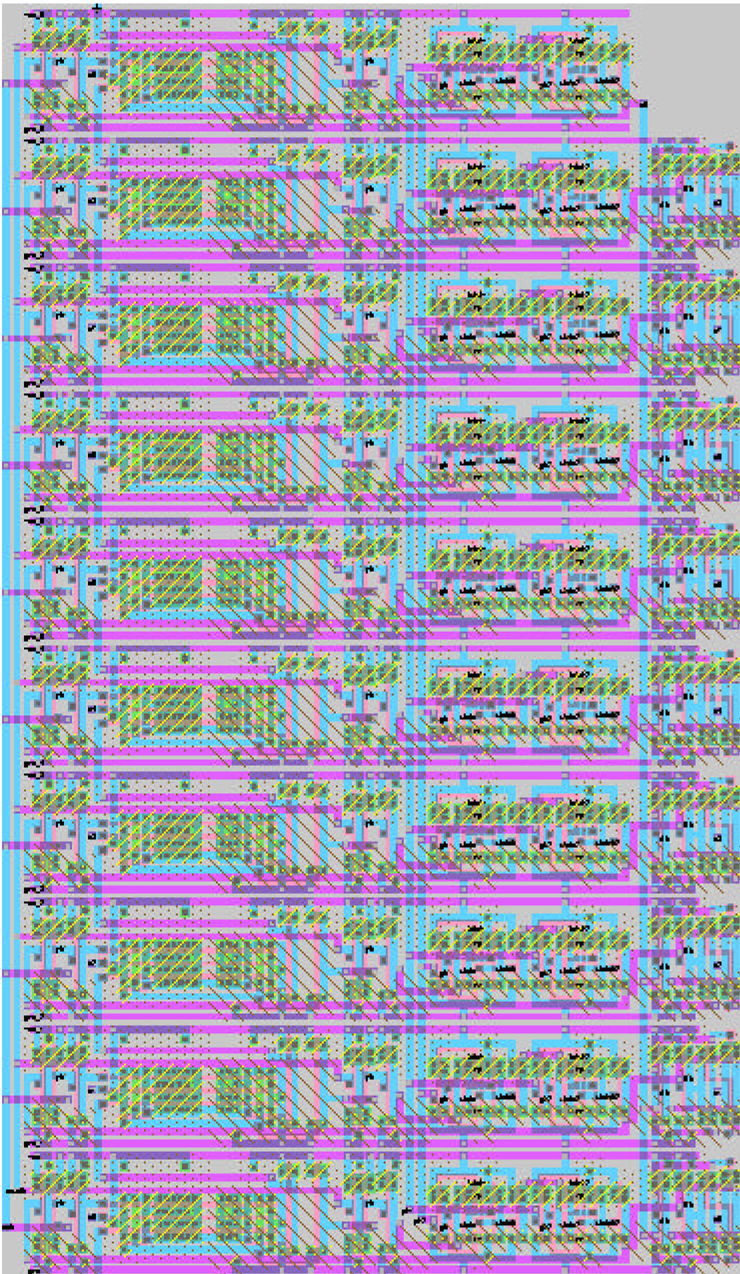
Datapath_energy



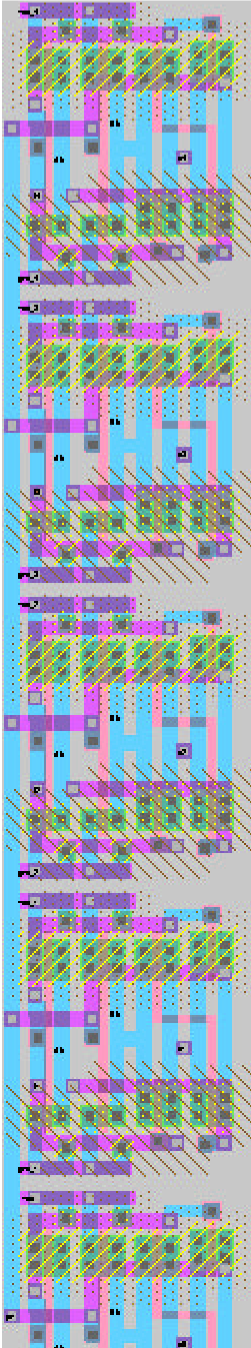
datapath_coherent_1bit



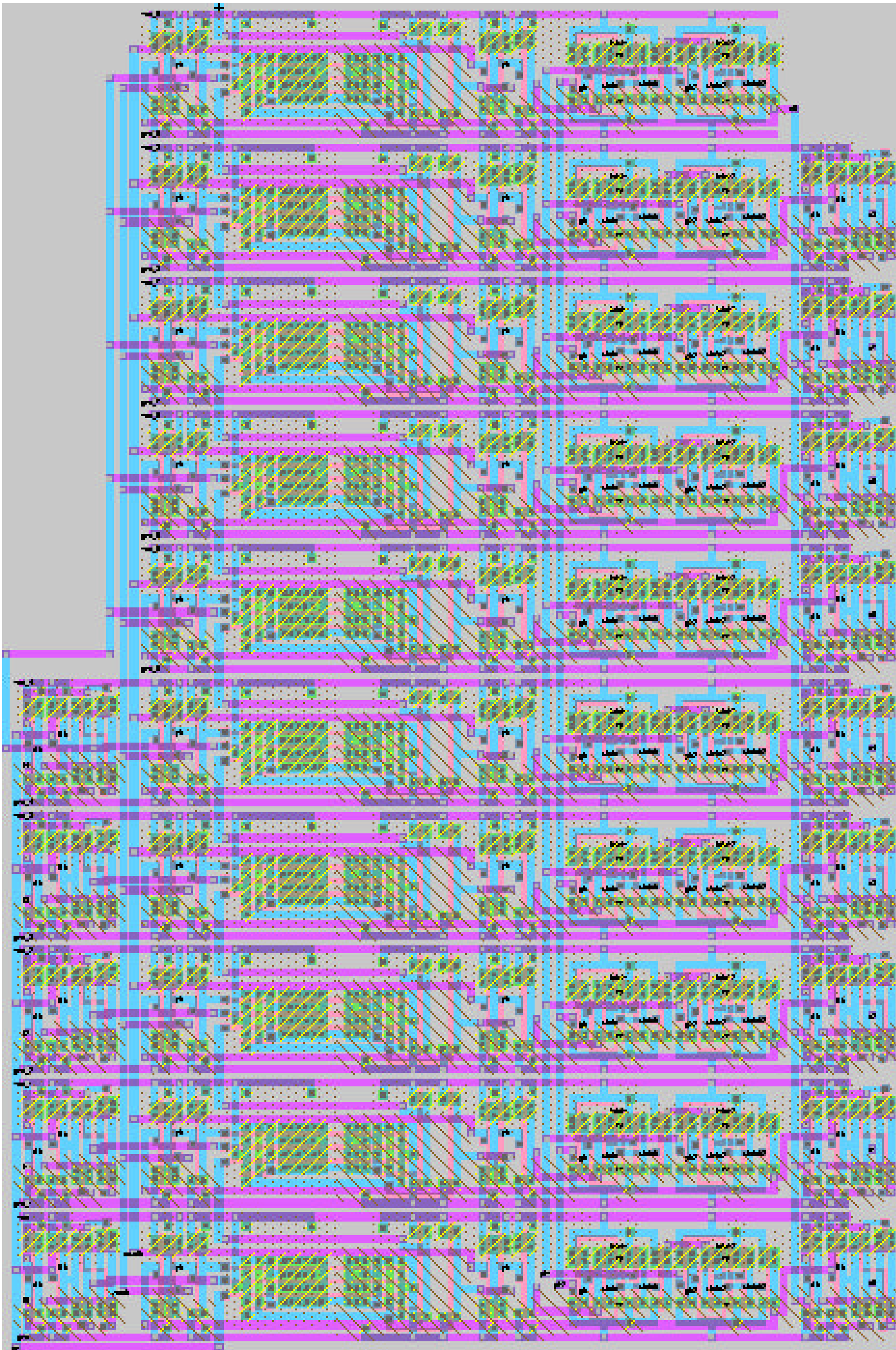
datapath_coherent



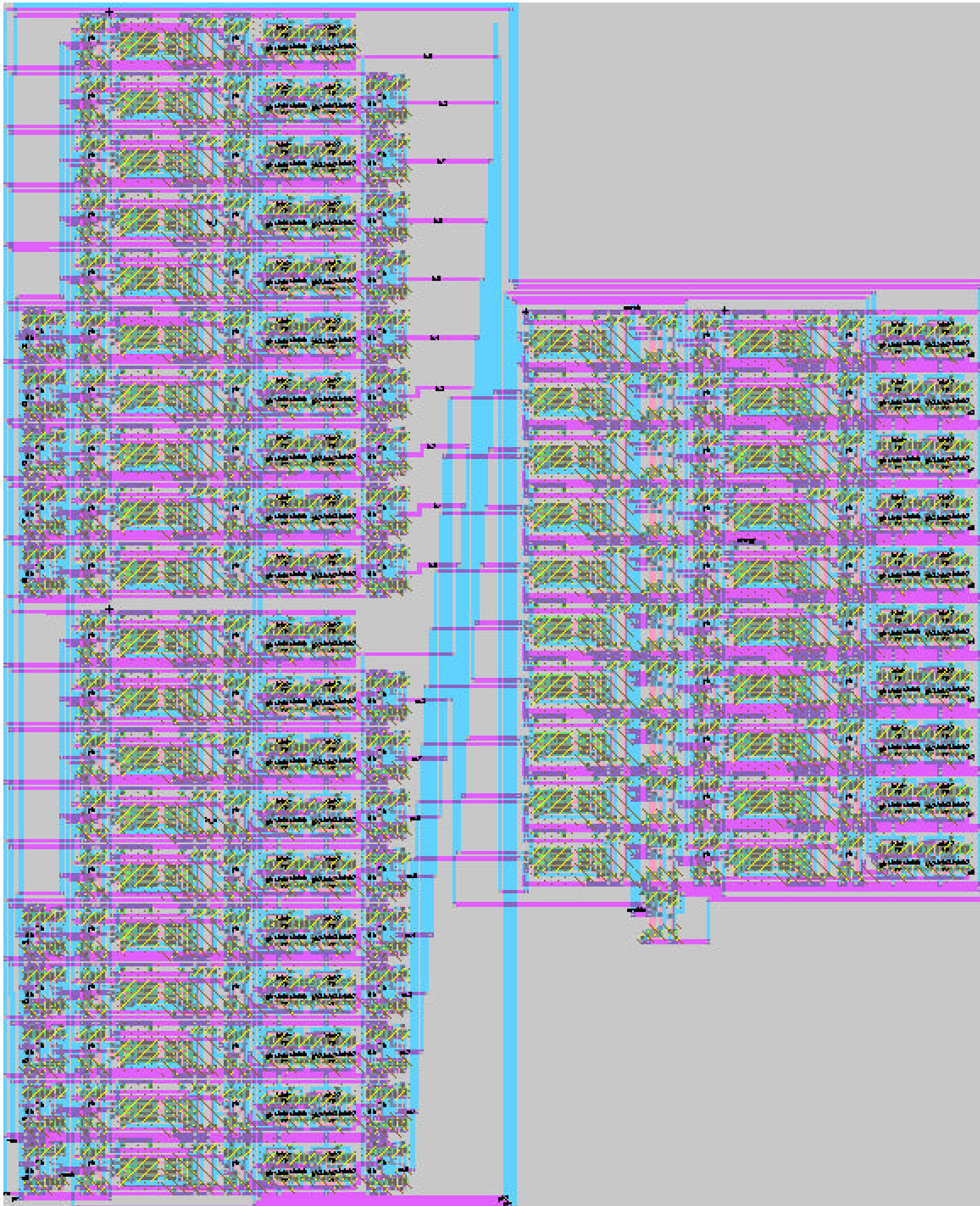
datapath_bds



datapath_iq

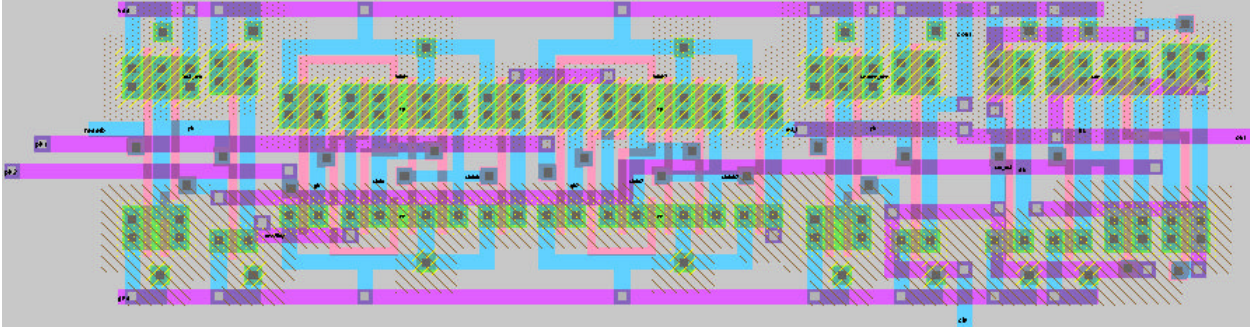


Datapath



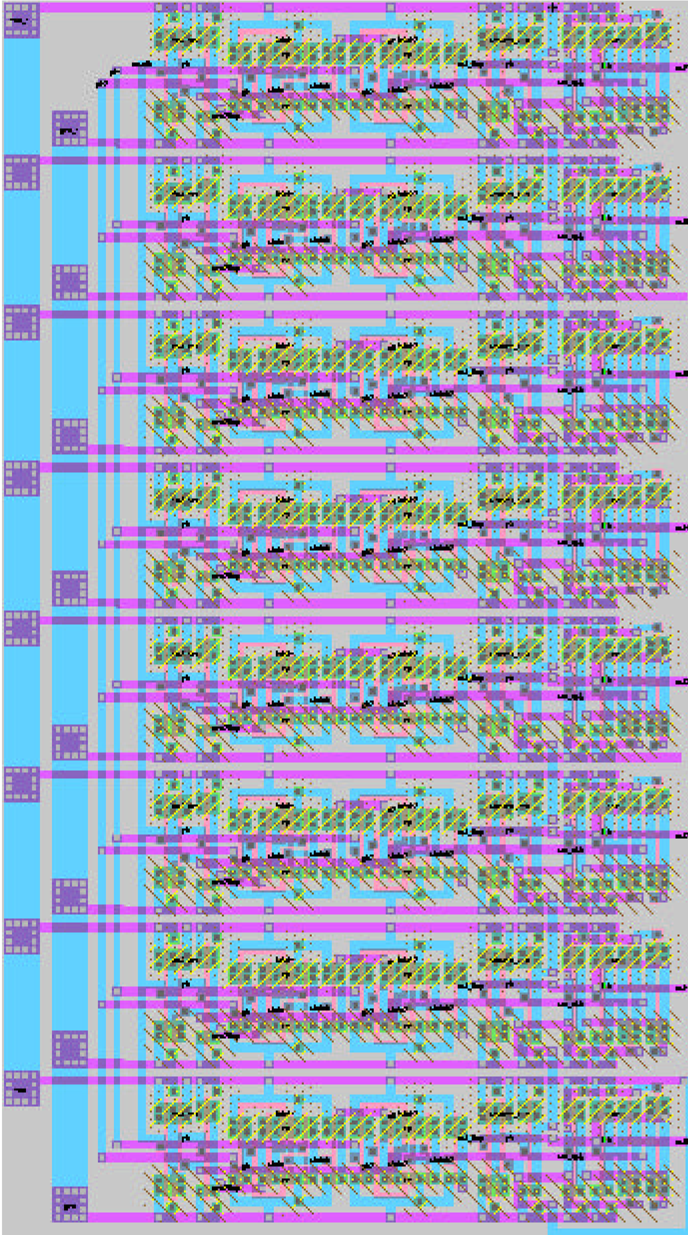
Coherent/Noncoherent Facets

Counter_reg_reset



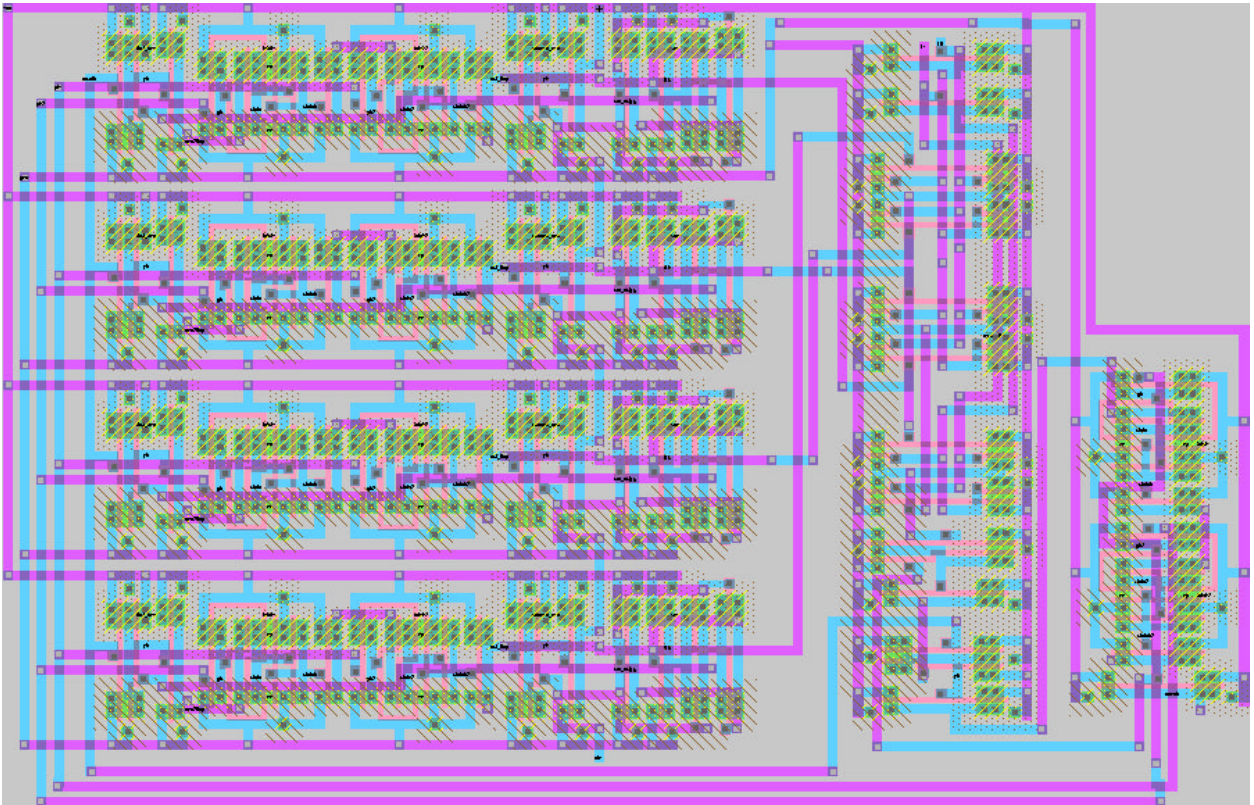
4-bit counter

8-bit counter



coh_dwell_counter

noncoh_dwell_counter



APPENDIX A : vlsi_searcher.m SOURCE CODE

```
function energy = vlsi_searcher(satid, N, L, offset)

conv_vlsi

pndata = full_signal_delay(satid, 1, 0);
pndata = pndata - 1; %% Converting to 1s and 0s
pndata = pndata ./ -2;

pndata(1:15)
pndata = shift(pndata, 1023 - offset);
pndata(1:15)

iqdata = fopen('iqdata_vlsi.in', 'r');

nonc_count = 0;
coh_count = 0;
energy = 0;

while (nonc_count < 2^L)
   coh_count = 0;
   coh_i = 0;
   coh_q = 0;
   while (coh_count < 2^(N+4)+1)
      iqhex = fgetl(iqdata);
      iqbin = hex_bin(iqhex);
      i = [iqbin(3) iqbin(4) iqbin(5) iqbin(6) iqbin(7)];
      q = [iqbin(8) iqbin(9) iqbin(10) iqbin(11) iqbin(12)];

      %%% BDS REG
      if (pndata(1) == 0)
         ieff = i;
         qeff = q;
      else
         ieff = (i .* -1) + 1; %% Invert
         qeff = (q .* -1) + 1;
      end

      %% I/Q Accumulate
      coh_i = coh_i + bin_dbl_t(ieff) + pndata(1); % Accumulate and use PN as
carry in
      coh_q = coh_q + bin_dbl_t(qeff) + pndata(1); % Accumulate and use PN as
carry in

      coh_count = coh_count + 1;
      pndata = shift(pndata, 1022); % gets next PN bit
   end

   % INV / MUX
   coh_i_bin = dbl_bin(coh_i, 10, 10, 0); % Conv. to BIN 2's compl.
   coh_q_bin = dbl_bin(coh_q, 10, 10, 0);

   if (coh_i_bin(1) == 0)
      c2n_i = coh_i_bin;
   else
```

```
      c2n_i = (coh_i_bin .* -1) + 1;
   end

   if (coh_q_bin(1) == 0)
      c2n_q = coh_q_bin;
   else
      c2n_q = (coh_q_bin .* -1) + 1;
   end

   c2n_i_dec = bin_dbl(c2n_i);
   c2n_q_dec = bin_dbl(c2n_q);

   %% CLINIC
   %non_dec = ((c2n_i_dec + coh_i_bin(1))^2) + ((c2n_q_dec +
coh_q_bin(1))^2);
   %energy = energy + non_dec;

   %% VLSI
   non_dec =  c2n_i_dec + c2n_q_dec + coh_i_bin(1); %% Uses bit 9 of I as
carry in
   energy = energy + non_dec + coh_q_bin(1); %% Uses bit 9 of Q as carry in

   nonc_count = nonc_count + 1;
end

fclose(iqdata);

foo = vlsi(satid, N, L, offset)

y = energy;
```

APPENDIX B : vlsi.m SOURCE CODE

```
function w = vlsi(satid, N, L, offset)

satid_bin = conv_satid(satid);

warning('Creating files for VLSI simulation from iqdata_vlsi.in ...');

fid1 = fopen('iqdata_vlsi.in', 'r');
fid2 = fopen('searcher.cmd', 'w');

fprintf(fid2, 'clock ph2 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0\n');
fprintf(fid2, 'clock ph1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1
1 1 1 1 1 1 1 0 0 0 0\n');
fprintf(fid2, '\n');

fprintf(fid2, 'vector i i4 i3 i2 i1\n');
fprintf(fid2, 'vector q q4 q3 q2 q1\n');
fprintf(fid2, 'vector satid s7 s6 s5 s4 s3 s2 s1 s0\n');
fprintf(fid2, 'vector energy e9 e8 e7 e6 e5 e4 e3 e2 e1 e0\n');
fprintf(fid2, 'vector N N1 N0\n');
fprintf(fid2, 'vector L L1 L0\n\n');
fprintf(fid2, '\n');

%%% Visualization

fprintf(fid2, 'ana resetb start ph2 i q enable energy doneb\n\n');

%%% Set Sat ID HERE
fprintf(fid2, 'set satid ');
for count = 1 : 8
   fprintf(fid2, '%i', satid_bin(count));
end
fprintf(fid2, '\n');

N_bin = dbl_bin(N, 2, 2, 0);
L_bin = dbl_bin(L, 2, 2, 0);

fprintf(fid2, 'set N ');
for count = 1 : 2
   fprintf(fid2, '%i', N_bin(count));
end
fprintf(fid2, '\n');

fprintf(fid2, 'set L ');
for count = 1 : 2
   fprintf(fid2, '%i', L_bin(count));
end
fprintf(fid2, '\n\n');


%%% Reset searcher
fprintf(fid2, 'l resetb start\n');
fprintf(fid2, 'c\n');
fprintf(fid2, 'c\n');
```

```
fprintf(fid2, 'c\n');
fprintf(fid2, 'c\n\n');

fprintf(fid2, 'h resetb\n');
for count = 1 : offset
   fprintf(fid2, 'c\n');
end

fprintf(fid2, 'h start\n\n');

while (feof(fid1) == 0)
   hex_10bit = fgetl(fid1);
   bin_10bit = hex_bin(hex_10bit);
   q_5bit = [bin_10bit(8) bin_10bit(9) bin_10bit(10) bin_10bit(11)
bin_10bit(12)];
   i_5bit = [bin_10bit(3) bin_10bit(4) bin_10bit(5) bin_10bit(6)
bin_10bit(7)];

   q_4bit = [bin_10bit(8) bin_10bit(9) bin_10bit(10) bin_10bit(11)];
   i_4bit = [bin_10bit(3) bin_10bit(4) bin_10bit(5) bin_10bit(6)];

   fprintf(fid2, 'set i ');
   for count = 1 : 4
      fprintf(fid2, '%i', i_4bit(count));
   end
   fprintf(fid2, '\n');
   fprintf(fid2, 'set q ');
   for count = 1 : 4
      fprintf(fid2, '%i', q_4bit(count));
   end
   fprintf(fid2, '\n');
   fprintf(fid2, 'c\n\n\n\n\n');

end

fprintf(fid2, 'c\n');
fprintf(fid2, 'c\n');
fprintf(fid2, 'c\n');
fprintf(fid2, 'c\n');
fprintf(fid2, 'c\n');
fprintf(fid2, 'c\n');
fprintf(fid2, 'c\n');
fprintf(fid2, 'c\n');
fprintf(fid2, 'c\n');
fprintf(fid2, 'c\n');
fprintf(fid2, 'c\n');

fclose(fid1);
fclose(fid2);

w = 0;
```