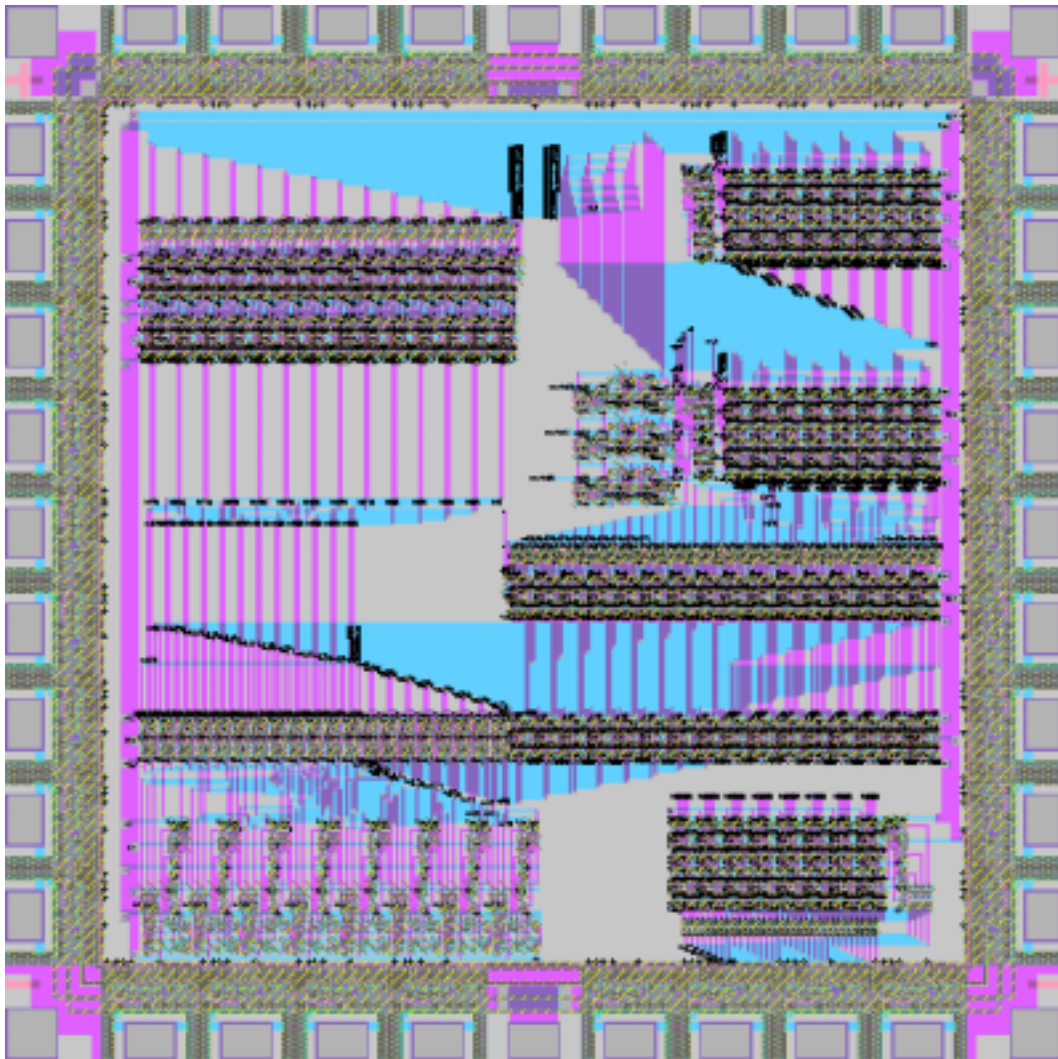


DES Encryption Chip

Braden Pellett
May May Wang
Steve Yan



E158 | VLSI | Professor Harris
April 11, 2001

Final Project: DES Encryption Chip

Braden Pellett

May May Wang

Steve Yan

E158 | VLSI | Professor Harris

April 11, 2001

I. Introduction

The Data Encryption Standard (DES) is a cryptographic algorithm. The government instituted DES as a standard encryption method in July 1977 and last reaffirmed it in 1993. When DES was considered secure, it was used regularly in government use for transmission of sensitive binary-encoded data between two endpoints on a non-secure medium.

DES relies on a key system to both encrypt and decrypt information. The original algorithm uses a 64-bit key that consists of a 56-bit randomly generated binary number along with 8 bits for parity checking. The values in this key are used to convert a 64-bit block of input data into a 64-bit block of encrypted output data following a specific series of permutations. The encrypted data is deciphered by applying the encryption key and reverse of the DES encryption algorithm.

This project uses the same DES encryption process using a 32-bit key to encode a 32-bit block of input data via a VLSI chip package, more specifically a 2.2 x 2.2 mm 40-pin MOSIS “TinyChip” fabricated using a 1.5 μm process. The layout fits in a $2200\lambda \times 2200\lambda$ area. The logic on the chip is driven synchronously via a two-phase clock input.

1.1. Algorithm Overview

A brief overview of the implemented DES encryption algorithm is as follows.

Standard DES encryption uses a 64-bit key to encrypt a 64-bit data block. Due to space constraints on the Mosis TinyChip, this project implements only 32-bit encryption. Furthermore, DES encryption depends on the strength of the bit swizzle blocks described below. Since this project was a practice in VLSI chip design rather than the mathematical details of encryption, randomly chosen swizzle blocks were used. Therefore the strength of encryption that the chip produces is not guaranteed.

The DES encryption algorithm can be divided into two computational paths: a path that handles the actual data encryption, which we will call the “datapath”, and a path that takes as input an initial key and calculates the series of keys to be used in encrypting the data. Standard DES uses 16 keys for encryption; this variant of DES uses only 8.

Datapath

The design encrypts data using the following method:

The 32 bits of the input block to be enciphered are first subjected to the following permutation, called the initial permutation *IP*:

IP							
29	25	21	17	13	9	5	1
31	27	23	19	15	11	7	3
28	24	20	16	12	8	4	0
30	26	22	18	2	6	10	14

That is the permuted input has bit 29 of the input as its first bit, bit 50 as its second bit, and with 14 as its last bit. The permuted input block is then the input to a key-dependent computation described below. The output of that computation is then subjected to the following permutation which is the inverse of the initial permutation:

IP ⁻¹			
8	24	3	16
9	25	2	17
10	26	1	18
11	27	0	19
12	28	4	20
13	29	5	21
14	30	6	22
15	31	8	23

The permuted input is further scrambled using 8 iterations of modulo arithmetic computations defined in terms of a cipher function f which operates on two blocks, one of 16 bits and one of 24 bits, and produces a block of 16 bits. Let the 32 bits of the input block to an iteration consist of a 16-bit block L followed by a 16-bit block R . Let the 32-bit input block be LR .

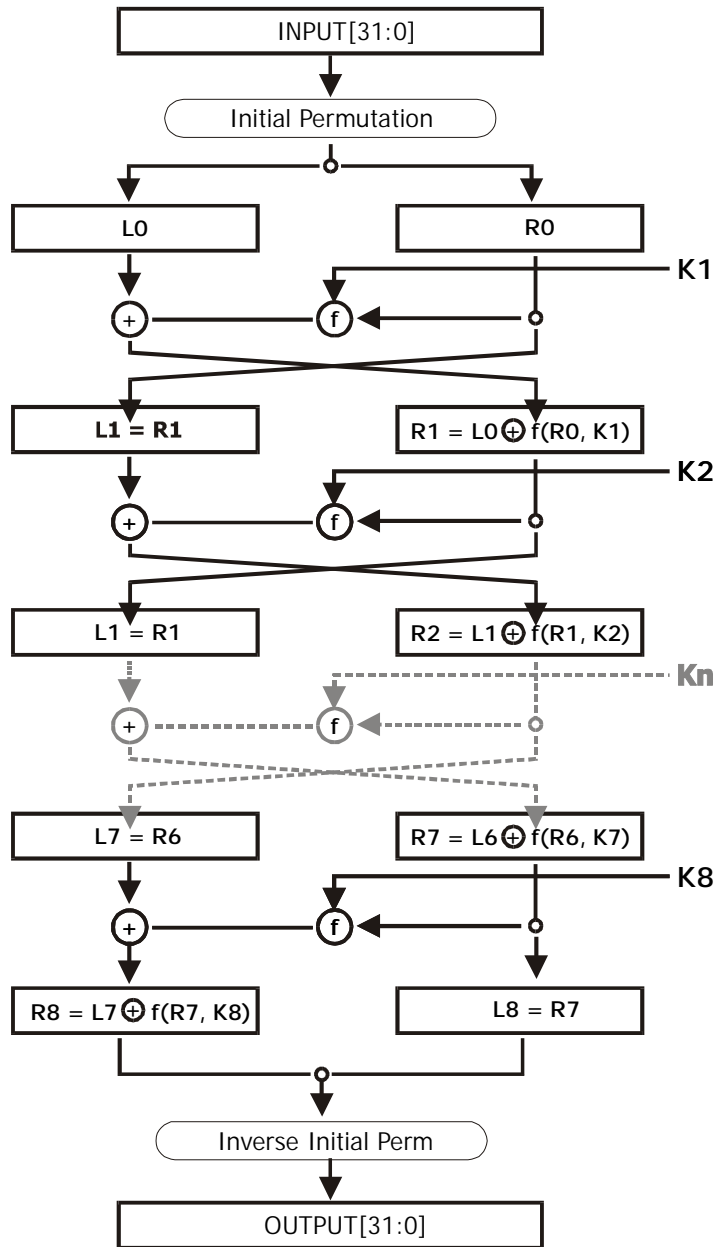
Let K be a block of 24 bits chosen from the 32-bit key, where K is selected using a key schedule that takes as input an integer n in the range from 1 to 8 and the 32-bit KEY block. Then the output $L'R'$ of an iteration with input LR is defined by:

$$L' = R$$

$$R' = L \oplus f(R, K)$$

where \oplus denotes bit-by-bit addition modulo 2. If $L'R'$ is the output of the 8th iteration then $R'L'$ is what we call the preoutput block. Since these bit swizzle blocks are permanent, the bits of the 32-bit bus are hardwired to match the permutations.

The following is a block diagram of the enciphering computation:



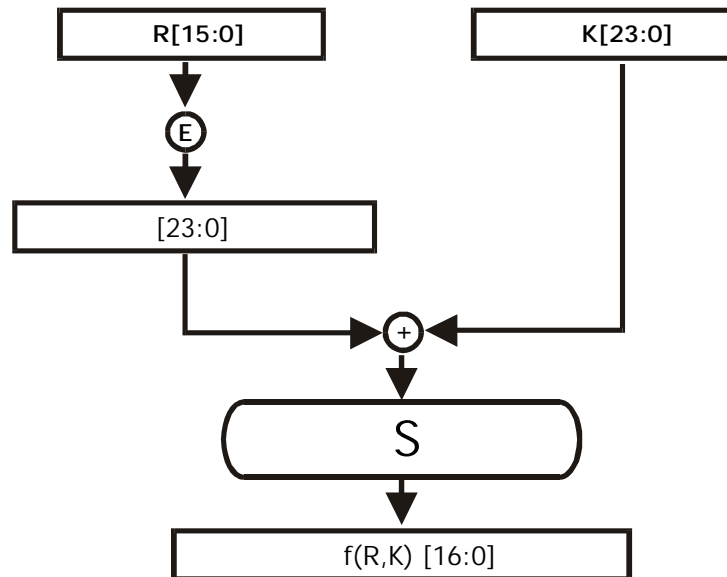
Enciphering computation

Let the 32 bits of the input block to an iteration consist of a 16 bit block L followed by a 32 bit block R . Therefore the input block is LR . Let K be a block of 24 bits chosen from the 32-bit key designed by KEY . Let K_n designate a function which takes as input the 32-bit key and outputs the n^{th} calculated key.

Let E denote a function which takes a block of 16 bits as input and yields a block of 24 bits as output. Let E be such that the 24-bits of its output are obtained by selecting the bits in its inputs in order according to the following table:

E		
1	2	3
3	4	5
5	6	7
7	8	9
9	10	11
11	12	13
13	14	15
15	16	1

The exclusive OR of the output of E and the value of K are piped through a function block S which maps the result to a 16-bit value (see diagram below.)



$f(R,K)$ cipher function

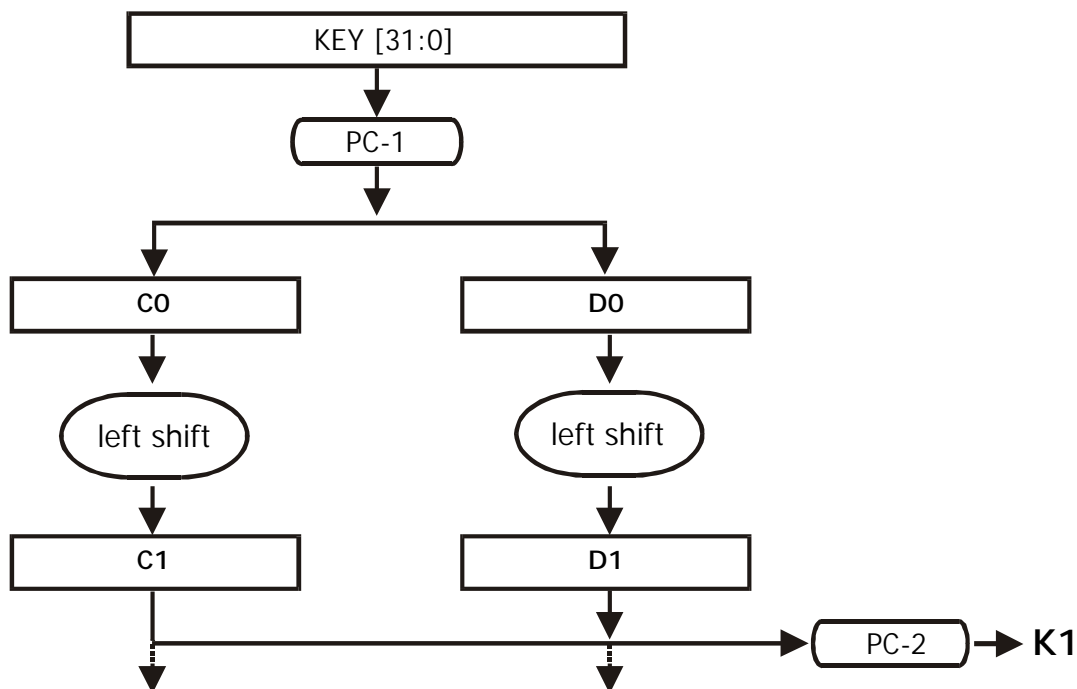
Keypath

The key schedule is calculated using a series of left shifts preceded and followed by two permuted choice blocks, designated PC-1 and PC-2. This key schedule feeds into the datapath by the following values for PC-1 and PC-2 and left shift permutations:

PC-1			
24	16	8	0
25	17	9	1
26	18	10	2
27	19		
30	22	14	6
29	21	13	5
28	20	12	4
11	3		

PC-2		
6	11	13
2	4	5
1	12	3
7	9	0
25	14	19
23	21	27
16	22	20

The following computation is performed in the keypath:



Key schedule calculation.

1.2. Hardware Implementation

1. I/O

Since the MOSIS package only allows for 40 pins, 6 of which are power and ground lines, data is segmented into 8-bit bytes and piped into 8×4 arrays of latches that hold the data and key bits. These registers allow the entire message and key to be entered into the process before continuing to the next step. Furthermore, each 32-bit memory array is addressed using two address bits controlled externally by the user. Therefore input and output can be entered and extracted from the chip asynchronously.

2. Initial Permutation

Both 32-bit outputs from the key and data register must undergo an initial permutation, or “bit-swizzling”. Bit-swizzling means that blocks of wires are swapped, or “swizzled”, to permute the data of a 32-bit word as it transitions between logic blocks in the chip. Not all of the bit wires need to be swizzled and retained, or vice versa, some of the bit wires can be repeated. For example, in the case of the 32-bit key, only 28 bits are permuted and sent into the barrel shifter, while the permuted message retains all 32 bits.

3. Initial Components for the 8-iteration Calculation

DES relies on an 8-iteration calculation that relies on the sum of two modulo arithmetic calculations. Both summations can be performed by XOR gates.

In the initial iteration, the first XOR sums up a 24-bit K derived from the key, and a permuted 24-bit E derived from the message. In order to determine K , the 28-bit permuted key is split into two 14-bit sections C , D and each section is left-shifted. This is implemented by two barrel shifters composed of a series of flip-flops that don't lose their values until after they transfer their value to the next register in line. Each of these barrel shifters includes two transmission gates to select between data from the previous flip-flop and initialization data from the IP permutation block.

4. Calculation of $f(R,K)$

The result of the first XOR (the sum of K and E) is a 24-bit solution or eight sections of 3-bit solutions. Each of the eight 3-bit solutions behave as address bits to eight S function blocks that act as ROM, holding 2 bits of data at each location. Each S-block is implemented as two muxed smaller ROM blocks that use pull-down transistors to instantiate their values. The eight sections of 2-bit retrieved values or 16-bit solution is then summed up with L_0 using the second XOR. The result is a 16-bit solution that becomes known as R_1 .

5. 8-iteration calculation

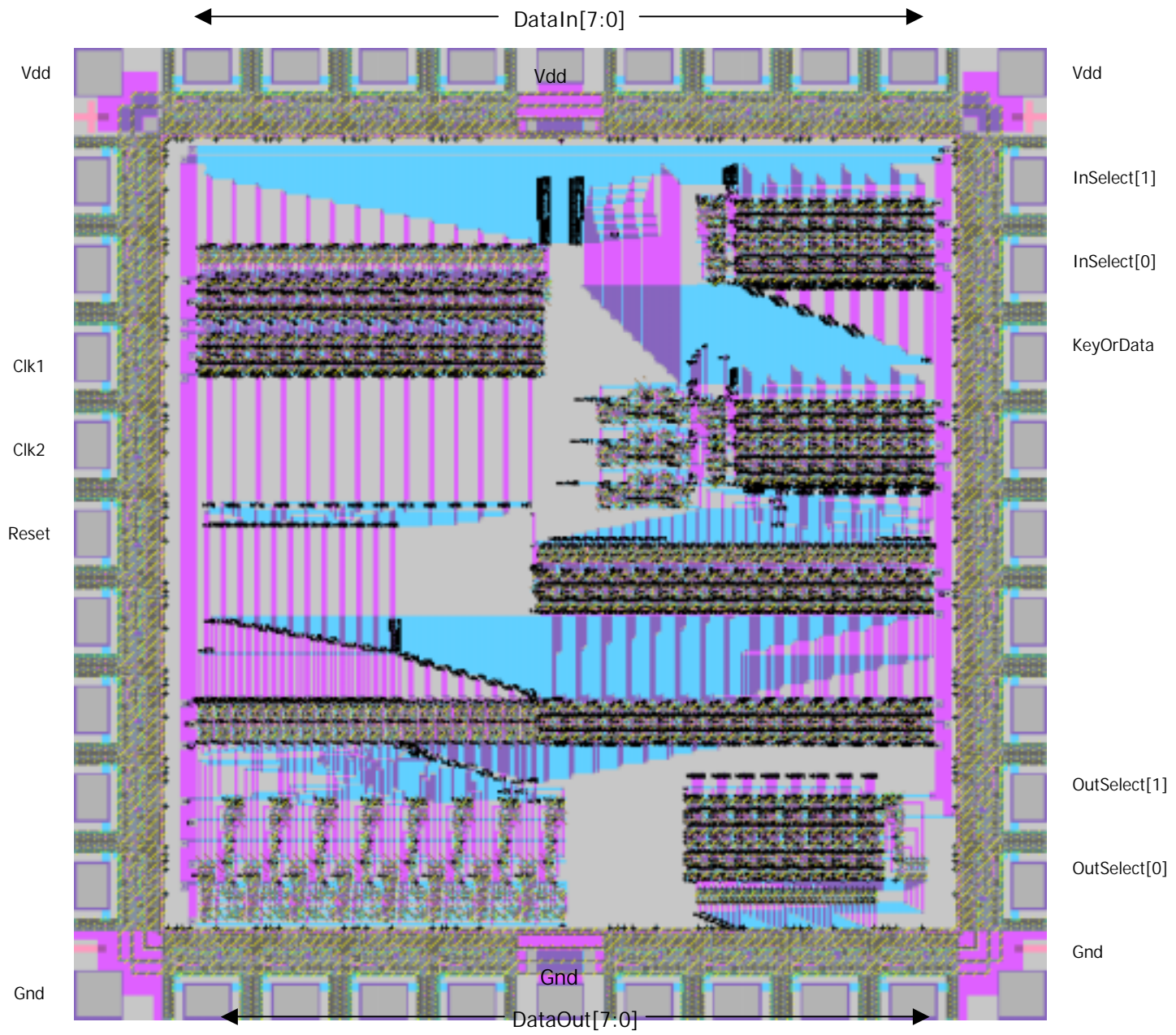
The calculation in the chip is controlled by a synchronous counter (composed of an incrementer and a register). Each iteration depends on the results of the previous iteration. Close to the end of the first iteration, values for R_1 , R_0 , and L_0 , were obtained. In the second iteration, the value of R_1 becomes the new value for R_0 . The value of R_0 becomes the new value of L_1 , and the value of L_1 will become the value for L_0 . Then the first XOR repeats its calculation for the second iteration. At the end of the eighth iteration, the value of R_1 and L_1 enter a final permutation and exit through output registers. To switch the values of R_0 , L_0 , R_1 , and L_1 without losing data, four arrays of registers are used in combination with two-

phase clocking. When the correct phase of the clock goes high, the array of registers release their value to the next array.

1.3. I/O Pinout

The following is the pinout of the chip implemented design.

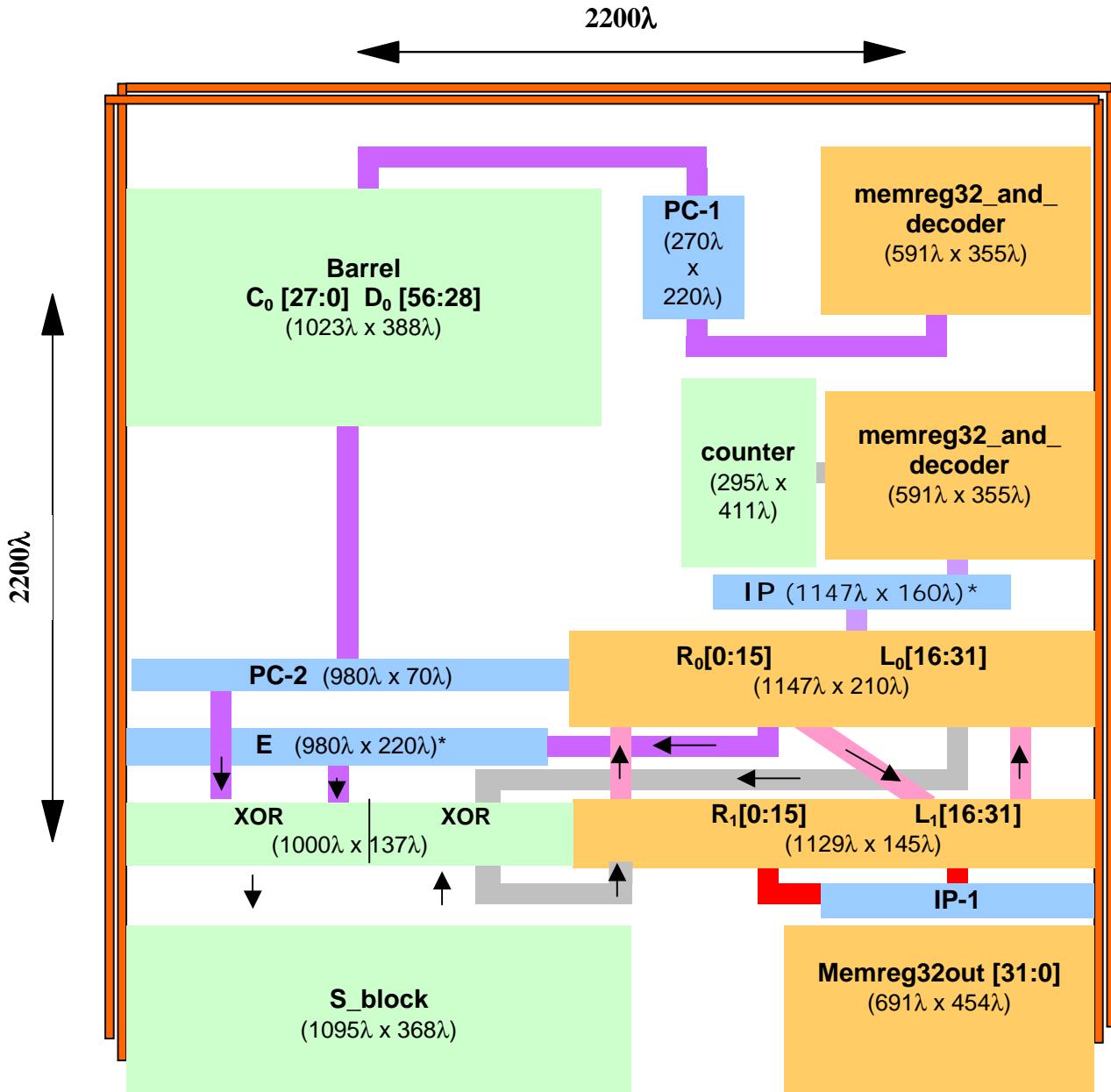
Input		# pins
NAME	Description	
V _{DD}	Power (3)	23 pins
Gnd	Ground (3)	
Clk ₁	Clock, phase 1 (1)	
Clk ₂	Clock, phase 2 (1)	
Reset	Global reset (1)	
KeyOrData	Select either key or data input (1)	
OutSelect[1:0]	Output word selection (2)	
InSelect[1:0]	Input word selection (2)	
DataInReady	Data ready to be read in (1)	
DataIn[7:0]	Unencrypted data word in (8)	
Output		
NAME	Description	
DataOutReady	Goes HI when encrypted data is ready (1)	9 pins
DataOut[7:0]	Encrypted data out (8)	
Total		32 pins






FLOOR PLAN

2.1. Introduction

This project implemented a 32-bit version of the DES encryption algorithm onto a $2200\lambda^2$ layout area. The floor plan demonstrates how the chip components fit within the given area.



	Register Block	* not drawn to scale [E, IP]
	Bit-Swizzling Block	
	Logic Block	

2.2. Layout

As seen above in the diagram, the floor plan consists of three main types of blocks: a register block, bit swizzle block, and a logic block. Register and logic blocks consist of the leaf cells discussed in the next section.

[Note that bit swizzle blocks are simply blocks of wires that are swapped, or “swizzled”, to permute the data of a 32-bit word as it transitions between logic blocks in the chip. Their area estimate was calculated by finding the area that the wires required to change directions from the output bus to the changed inputs designated for those bits.]

Below is a listing of the facet-by-facet area.

Leaf Cell Name	Layout Size (W x H)	Area (λ^2)	Notes
memreg32_and_decoder	591 x 355	209805	Accepts and holds 8-bit segments of the input until the full 32-bits are received.
L ₀ R ₀	1147 x 210	240870	Holds data until correct clock phase, then delivers values to another array of registers
L ₁ R ₁	1129 x 145	163705	
Memreg32out	691 x 454	313714	Accepts the full 32-bits of the output, and allows it to leave the chip in 8-bit segments.
Barrel	1023 x 388	396924	Left-shifts the two halves of the permuted key with each iteration.
Counter	296.5 x 118	34987	Calculates the number of iterations that the encryption process has undergone.
S_block [7:0]	1095 x 368	402960	Takes in a 3-bit address and delivers a 2-bit output.
IP and IP ⁻¹ (swizzle 32 bits)	1147 x 160	183520	Bit-swizzles the input so that a permuted output is sent.
PC-1 (swizzle 32 bits)	270 x 220	59400	
PC-2 (swizzle 32 bits)	980 x 70	68600	
E (swizzle 16 bits)	980 x 220	215600	

FACET PERFORMANCES

3.1. Leaf Cells

Leaf Cell Name	Layout Size (W x H)	Design Time (hrs)	DRC	ERC	NCC
xor2	46 x 60	11	Pass	Pass	
std_latch_mirror_mux	132 x 209.5	6	Pass	Pass	Pass
std_latch_mirror	80 x 240	3	Pass	Pass	Pass
std_latch_mirror_2	80 x 120	3	Pass	Pass	Pass
register	150.5 x 103	5	Pass	Pass	Pass
counter	296.5 x 118	9	Pass	Pass	Pass
s_block [7:0]	131 x 110.5	16	Pass	Pass	Pass
decoder24_en	96.5 x 265	5	Pass	Pass	Pass

3.2. Higher Level Cells

Leaf Cell Name	Layout Size (W x H)	Design Time (hrs)	DRC	ERC	NCC
mem[31:0]	591 x 355	5	Pass	Pass	Pass
L ₀ R ₀ [31:0]	1147 x 210	3	Pass	Pass	Pass
L ₁ R ₁ [31:0]	1129 x 145	3	Pass	Pass	Pass
Memreg32out [31:0]	691 x 454	3	Pass	Pass	Pass
C ₀ D ₀ (barrel shifters)	1023 x 388	6	Pass	Pass	Pass
IP and IP ⁻¹ (swizzle 32 bits)	1147 x 160	1	Pass	Pass	Pass
PC-1 (swizzle 32 bits)	270 x 220	1	Pass	Pass	Pass
PC-2 (swizzle 32 bits)	980 x 70	1	Pass	Pass	Pass
E (swizzle 16 bits)	980 x 220	1	Pass	Pass	Pass
Keypath	2185 x 2210	10	Pass	Pass	Fail
Datapath	1579 x 1755	10	Pass	Pass	Fail

3.3. Simulation Results

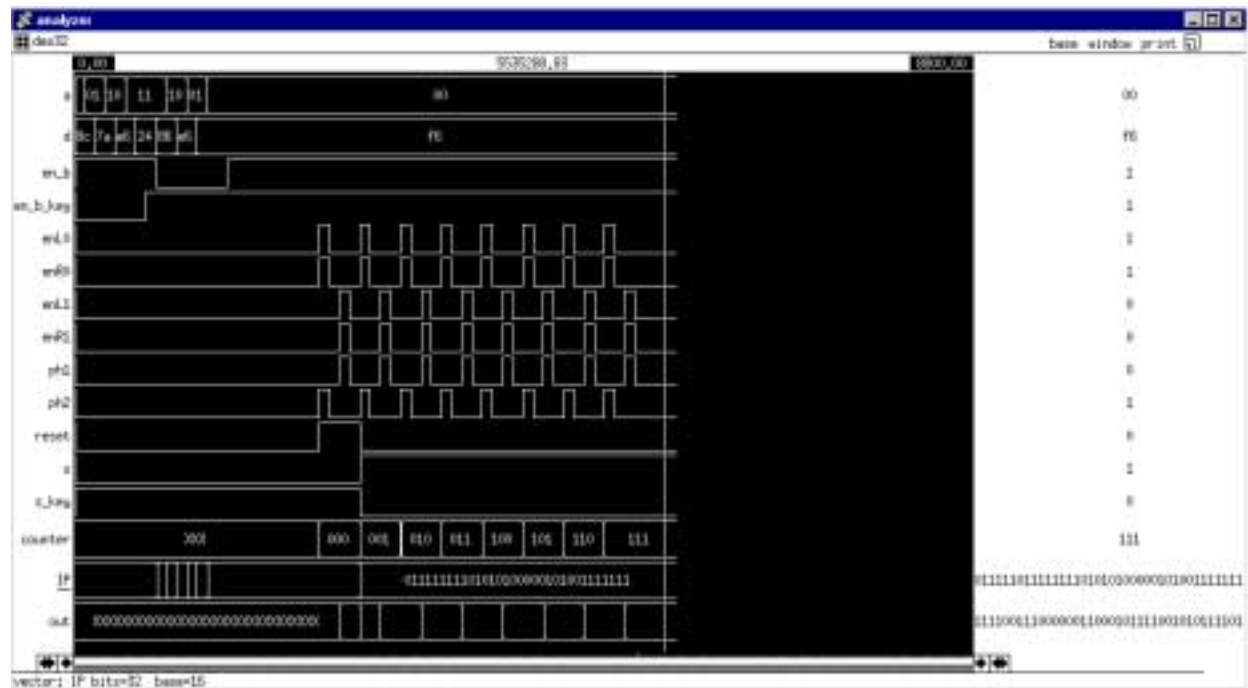
Our design works in schematic form as verified by our encryption test program. However, the top-level layout facet does not simulate correctly, nor does it NCC check correctly with the accompanying schematic. However, each of the layout cells one layer below the top level do simulate correctly and NCC check. These working layout blocks and a working top-level schematic lead us to believe that our design is sound.

Below are simulation waveforms of the top-level schematic and each of the main sub-blocks contained in the top-level layout.

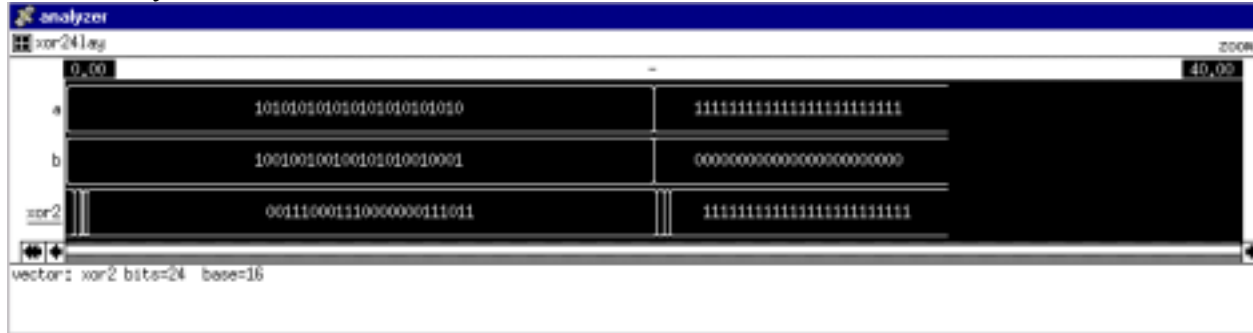
1. Top Level schematic simulation. The output vector out was verified by the included encryption verification program.

Vectors:

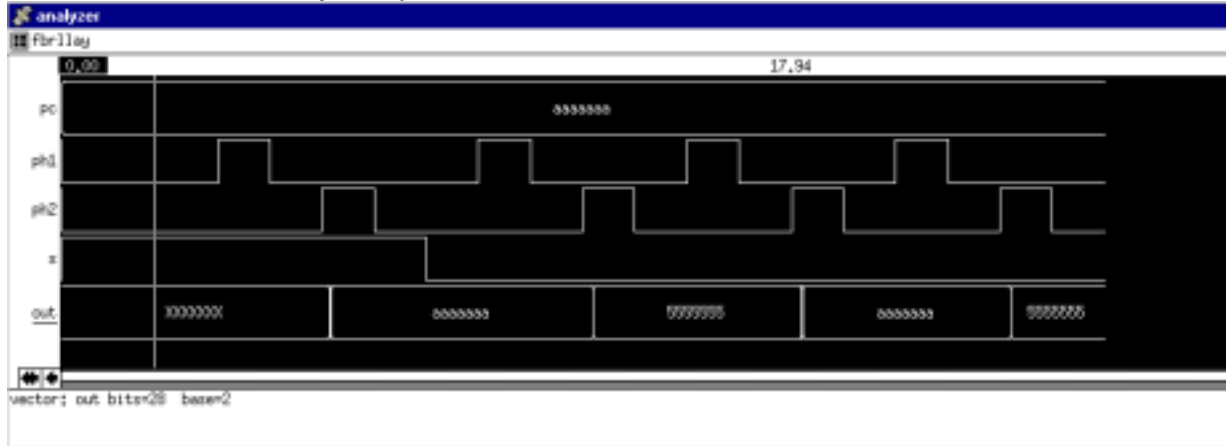
a[1:0]	address selection for 32-bit memory registers
d[7:0]	8-bit input bus
enL0, enR0, enL1, enR1	Clock signals for latches in facet LoRo{lay}
ph1, ph2	Two phase clocks
reset	Counter reset
s	Selects between previous value and output of IP permutation block in facet compute_mem{lay}
s_key	Selects between output of PC-1 permutation block previous flip-flop in facet fullbarrel{lay}
counter[2:0]	3-bit counter
IP[31:0]	Output value of IP permutation block
out[31:0]	Output value of facet compute_mem{lay} from subfacet L1R1{lay}



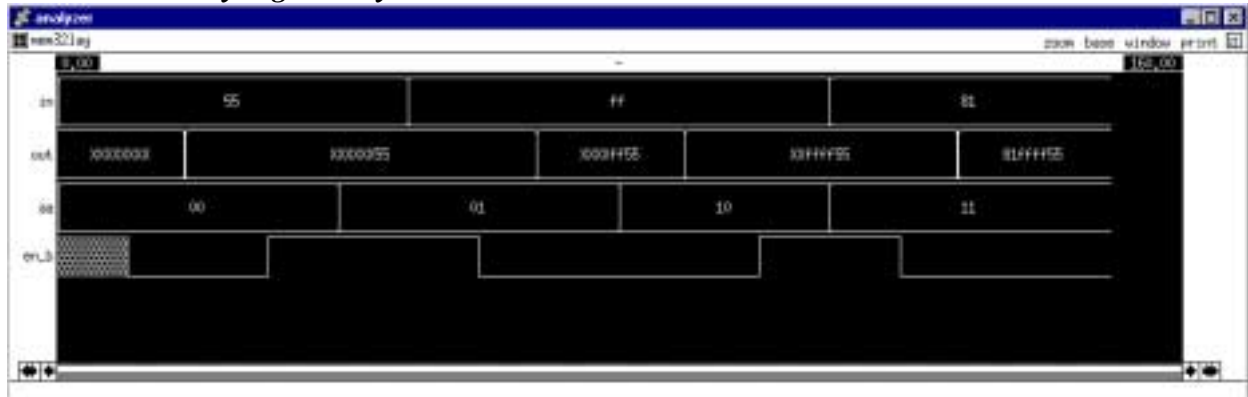
2. XOR Layout simulation



3. Full 32-bit barrel shifter layout simulation



4. 32-bit memory register layout simulation



POSTFABRICATION TEST PLAN

4.1. Procedure

Should this design be sent to fabrication, note that a ring oscillator, an *odd*-numbered circular ring of inverters, should be included in the design. The ring oscillator should be included in order to verify that electrical behavior is present on the chip. Testing of the chip can be performed using a functional chip tester that reads IRSIM .cmd files. The .cmd file can be used to assert values on the input pins and verify that the correct output is received on the output pins. A tester of the chip will not have to take on a “big bang” approach. The design allows for all latches to be made transparent as a simple first step. This will allow one of the input pins to be asserted and, with all latches transparent, should immediately generate a valid output signal. Appendix B shows the contents of a .cmd that that successfully simulates the schematic of the design.

In addition to the IRSIM file, in the process of designing our chip we wrote a simple 32-bit DES encryption program to verify the validity of our testing results. This program is included in Appendix A at the end of this document.

```
des32.cmd (IRSIM command file)
```

```
stepsize 100
```

```
|  
| Tester for the big kahuna, part I  
|
```

```
|  
| Select which of the 4 parts of the 32 bits we are  
| inputting.  
|  
vector a a1 a0
```

```
|  
| The 8-bit input bus  
|  
vector d d[{7:0}]
```

```
|  
| Allows memory register to receive data  
|  
en_b
```

```
|  
| Allows key register to receive data  
|  
en_b_key
```

```
|  
| enable the previous state latch  
|  
enL0, enR0
```

```
|  
| enable the current state latch  
|  
enL1, enR1
```

```

|
| The clock phases
| ph1, ph2
|
|
| Reset the chip
| reset
|
|
| Tell the previous state latch to select the
| new input (0) or the next input (1).
| s
|
|
| Tell the barrel shifter to select the new key
| (0) or to keep on shiftin' (1).
| s_key
|
|
| The counter [output]
vector counter counter[{2:0}]
|
|
| The output of IP
vector IP IP[{31:0}]
|
|
| The output of L1R1, right before IP^-1
vector out out[{31:0}]
vector finout finout[{7:0}]
|
|
| The output of the PC1
vector pc1 pc[{27:0}]
|vector barrel barrelnode18[{27:0}]
vector key_out key_outnode18[{23:0}]
vector xor_out xor_outnode18[{23:0}]
vector sOut sOut[{15:0}]
|
|*****
ana a d en_b en_b_key enL0 enR0 enL1 enR1 ph1 ph2 reset s s_key counter IP out finout
ana a1_out a0_out output_en
ana pc1
|ana barrel
ana key_out
ana xor_out
ana sOut

```



```
l a0_out a1_out
h output_en

l reset s ph1 ph2
h s_key
l enL0 enR0 enL1 enR1

h en_b en_b_key
```

```
|*****
| INPUT KEY
| (ASCII Value = "1^g$")
|
```

```
l en_b_key

set a 00
set d 10001100
s
set a 01
s
set d 01111010
s
set a 10
s
set d 11100110
s
set a 11
s
set d 00100100
s
```

```
h en_b_key
```

```
s
```

```
|
| END INPUT KEY
|*****
```

```
|*****
| INPUT DATA
| (ASCII value = "ooga")
|
```

```
l en_b
```

```
set a 11
set d 10000110
s
set a 10
s
set d 11100110
s
set a 01
s
set d 11110110
s
set a 00
s
set d 11110110
s
```

```
h en_b
```

```
|  
| END INPUT DATA  
| *****
```

```
s  
s  
s  
s  
s  
s  
s  
s  
s
```

```
clock ph1 0 0 1 0  
clock enL1 0 0 1 0  
clock enR1 0 0 1 0
```

```
clock ph2 1 0 0 0  
clock enL0 1 0 0 0  
clock enR0 1 0 0 0
```

```
h reset
```

```
c
```

```
l s_key
```

```
h s
```

```
l reset
```

```
c
```

```
c
```

```
c
```

```
c
```

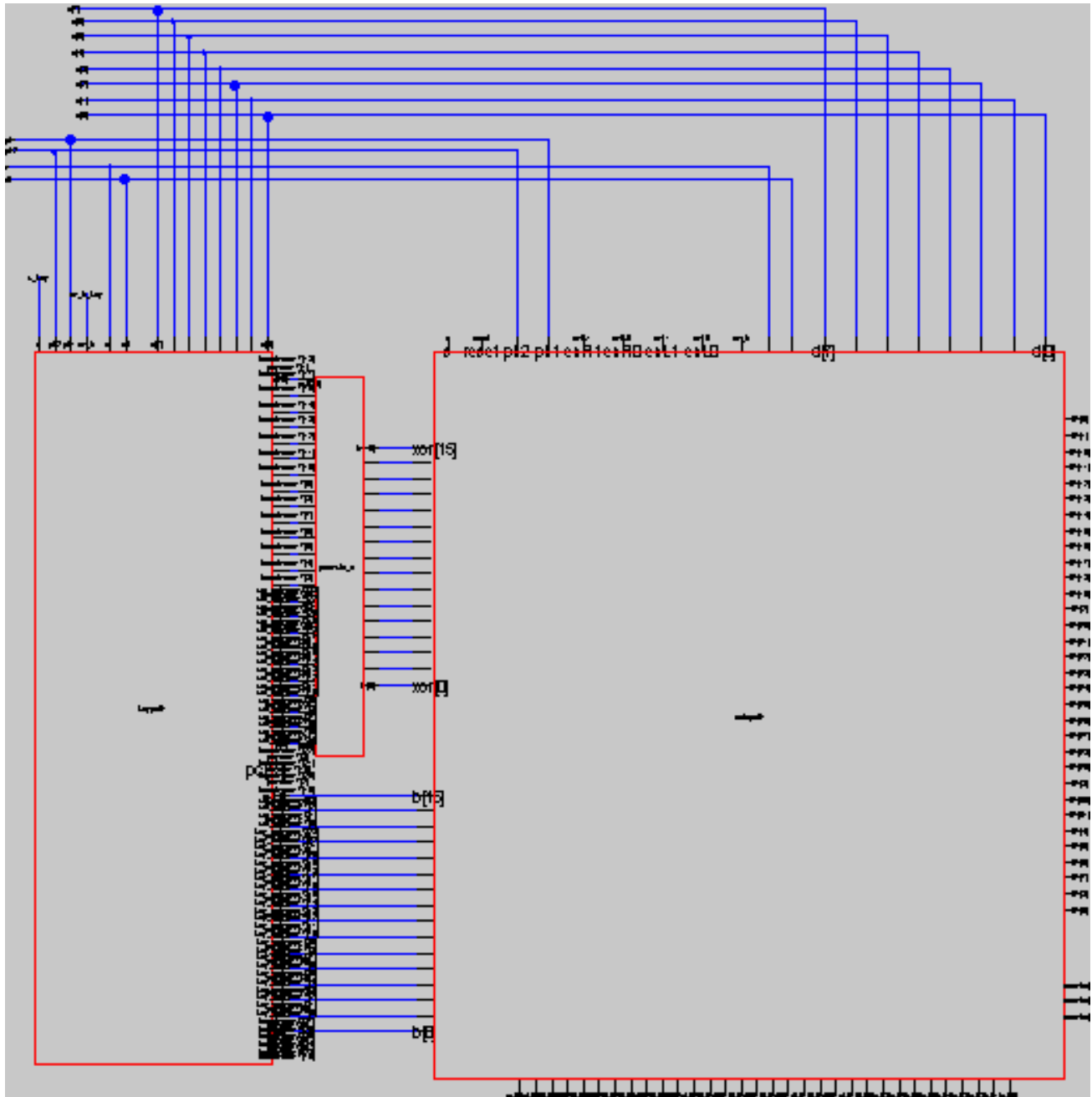
```
c
```

```
c
```

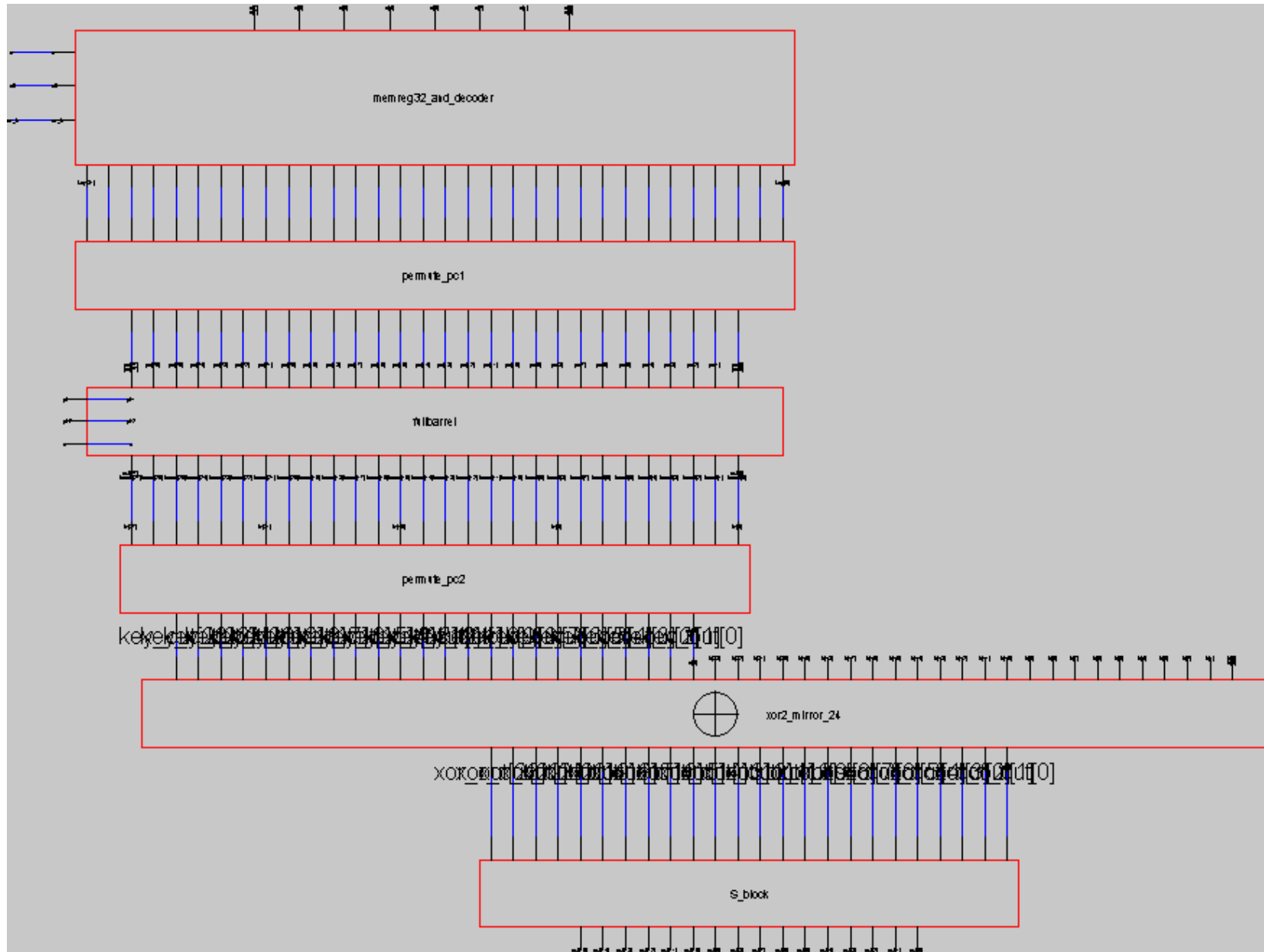
```
c
```

SCHEMATICS

5.1. DES Chip Schematic (des32)

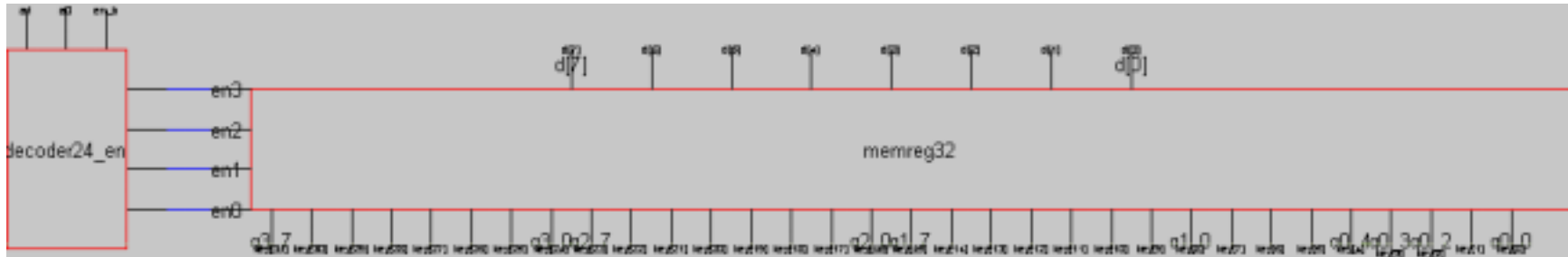


Keypath



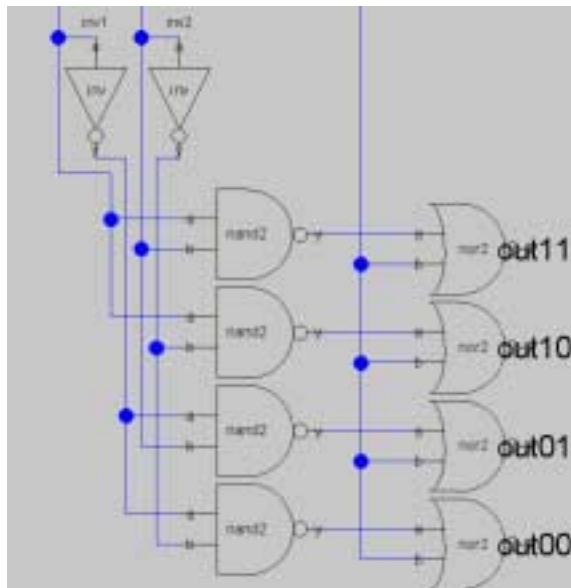
memreg32_and_decoder

uses decoder24_en and memreg32

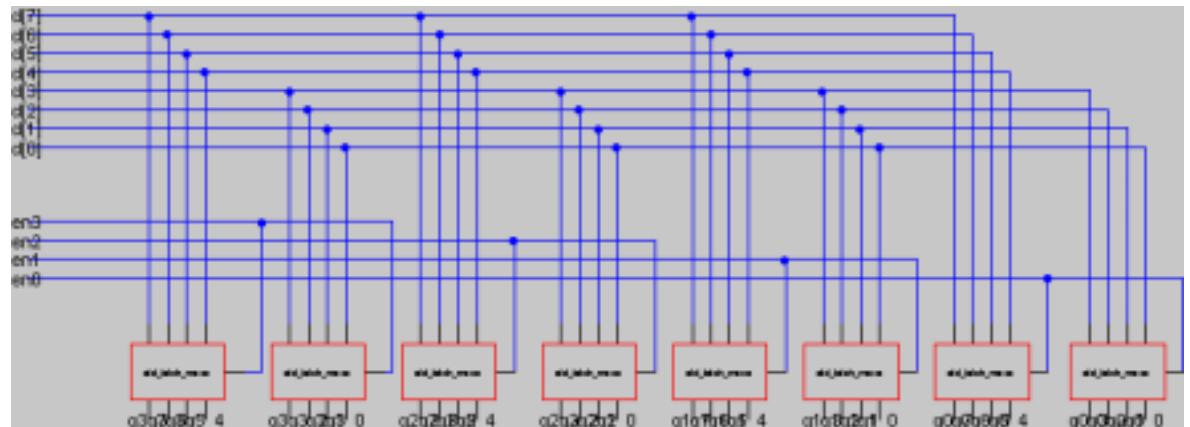


Requires one register for each of the 32 information bits to be held and a decoder to distinguish between the segments. The decoder is 2-bit binary to 4-bit mutually exclusive lines. When one of the four decoder outputs goes high, eight latches become transparent, allowing the value to be carried by the registers. Each of the four decoder outputs correspond to one of the (4) 8-bit segments of the 32-bit key.

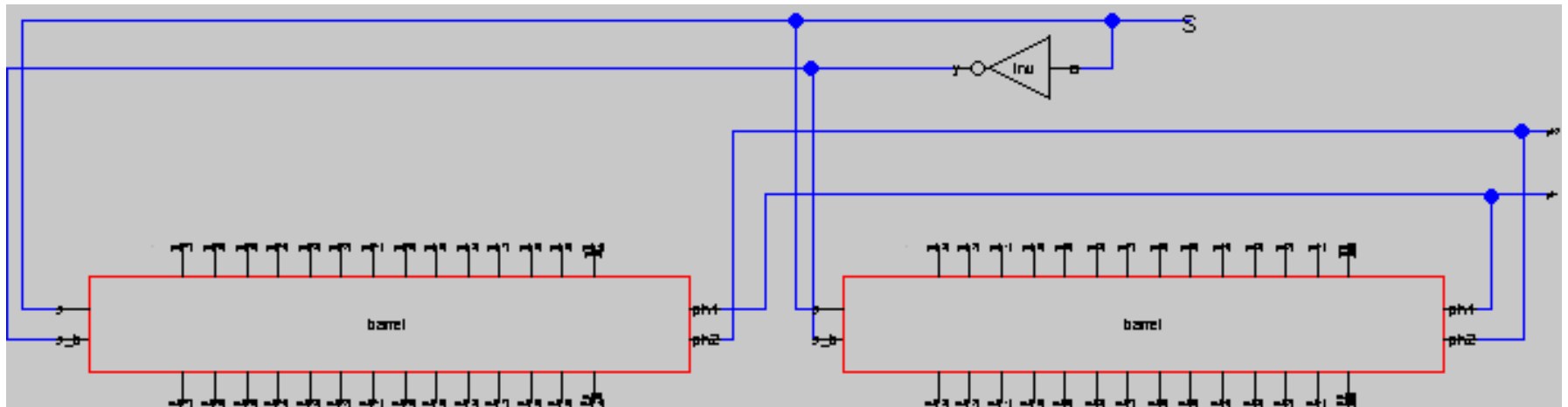
decoder24_en



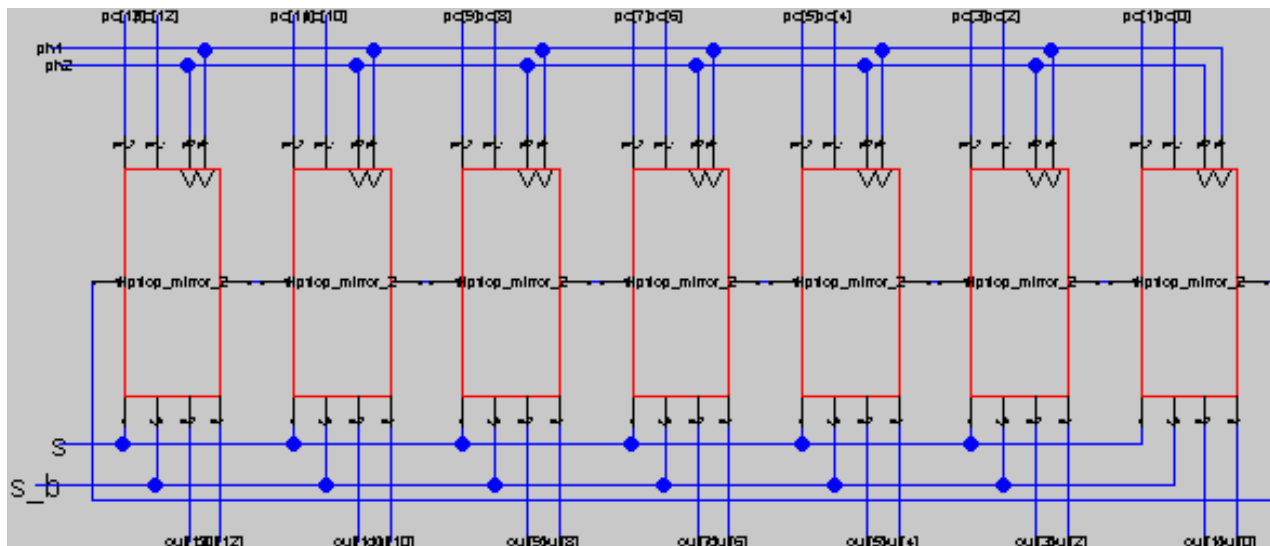
memreg32 - uses std_latch_mirror



fullbarrel
uses flipflop_mirror_2

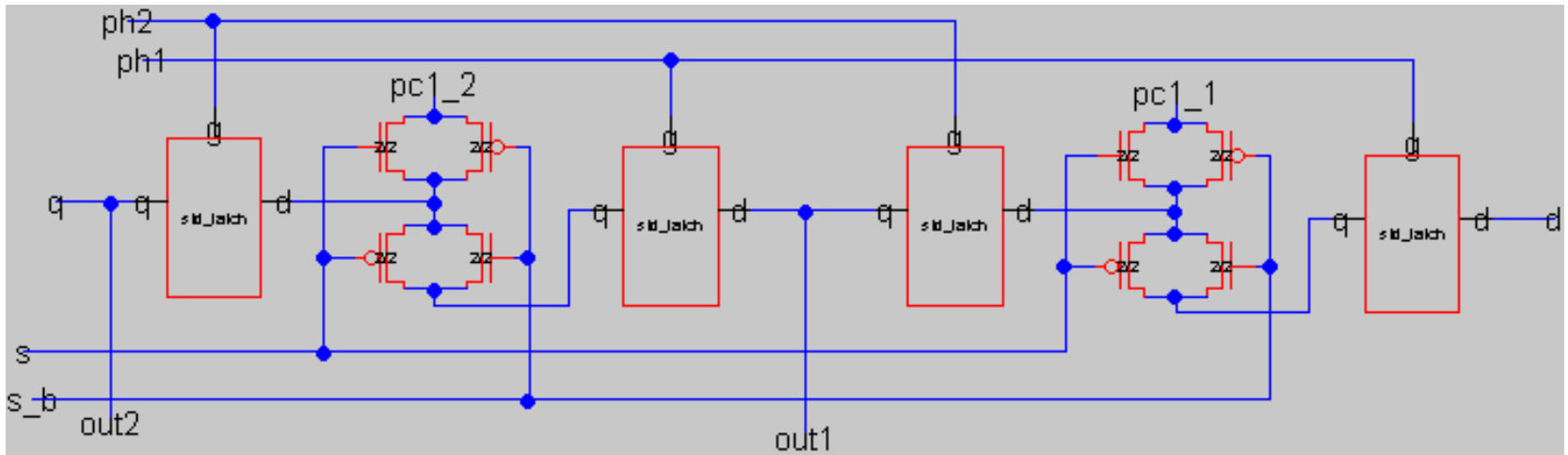


barrel – uses flipflop_mirror_2

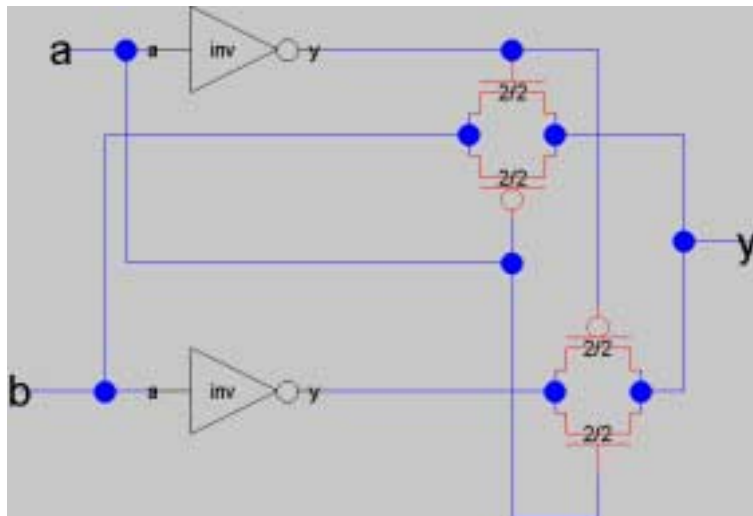


The fullbarrel divides the 28-bit permuted key and sends each half into a barrel shifter, which left shifts the 14-bit section with every two phases of the clock.

flipflop_mirror_2 - uses std_latch



xor2

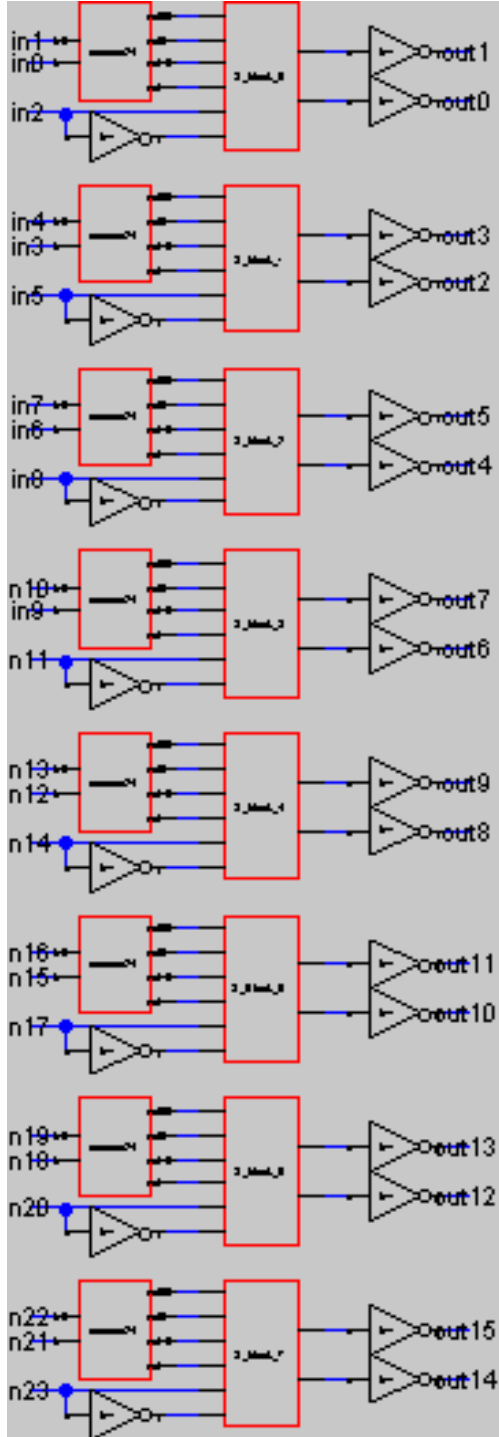


xor2_mirror_24



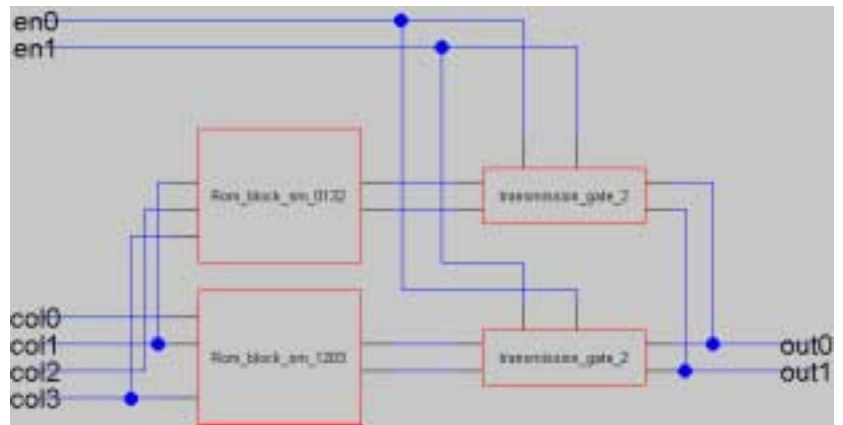
The XOR gate serves as the main computation unit for the encryption chip. 32 XORs are used to compute bitwise modulo 2 addition on two 32-bit words. The XOR gate layout is $46\lambda \times 60\lambda$, with a 16×2 array not exceeding $736\lambda \times 120\lambda$. Due to the multiple connections of each XOR gate, ready accessibility requires the full XOR array to not exceed a total width of 1100λ (1/2 of the available chip length).

S_block



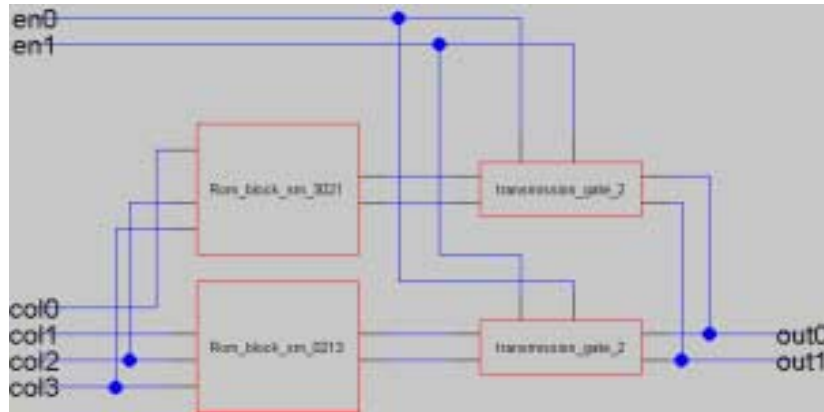
(s_block_0)

- uses transmission_gate_2,
ROM_block_0132, ROM_block_1203



S_blocks are ROM modules with a 2-bit output with a set of 4 lines and a set of 2 lines, where selecting one line from each set will select one of the 2-bit outputs. This represents a table of width 4 and height 2 filled with 2-bit outputs. There are eight S-block designs corresponding to the eight different functions.

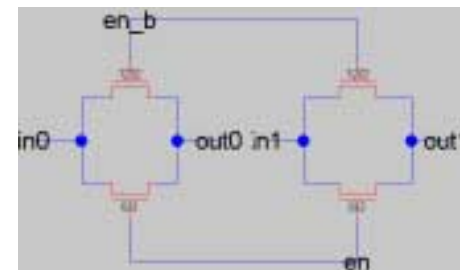
- (s_block_1) - uses transmission_gate_2, ROM_block_1023, ROM_block_2130



Same Design, (different ROM blocks)

- (s_block_2) - uses transmission_gate_2, ROM_block_3021, ROM_block_0213
- (s_block_3) - uses transmission_gate_2, ROM_block_3102, ROM_block_2013
- (s_block_4) - uses transmission_gate_2, ROM_block_1302, ROM_block_2031
- (s_block_5) - uses transmission_gate_2, ROM_block_1320, ROM_block_0231
- (s_block_6) - uses transmission_gate_2, ROM_block_0312, ROM_block_3120
- (s_block_7) - uses transmission_gate_2, ROM_block_2310, ROM_block_3201

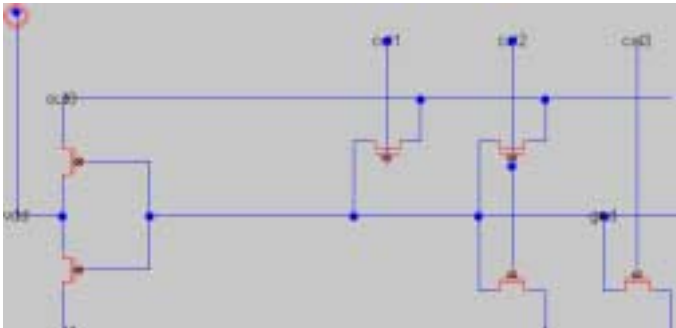
transmission_gate_2)



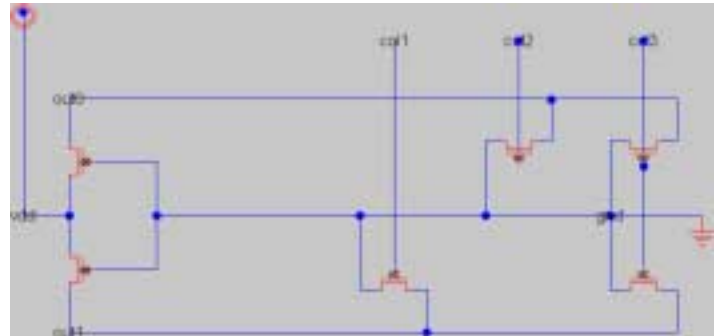
ROM_blocks

The ROM is implemented as two-mux'ed smaller ROM blocks with 4 inputs and a 2-bit output. Each smaller ROM block uses a pull-down transistor to instantiate a value.

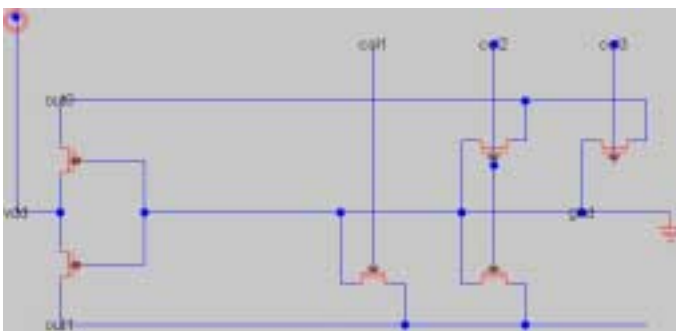
Rom_block_sm_0132



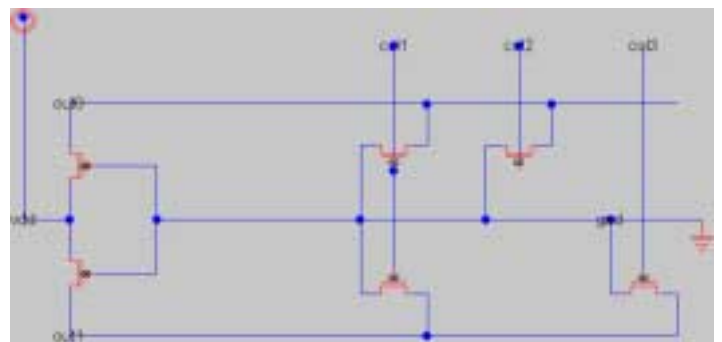
Rom_block_sm_0213



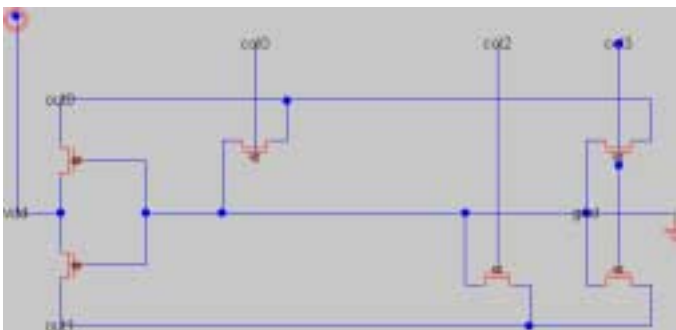
Rom_block_sm_0231



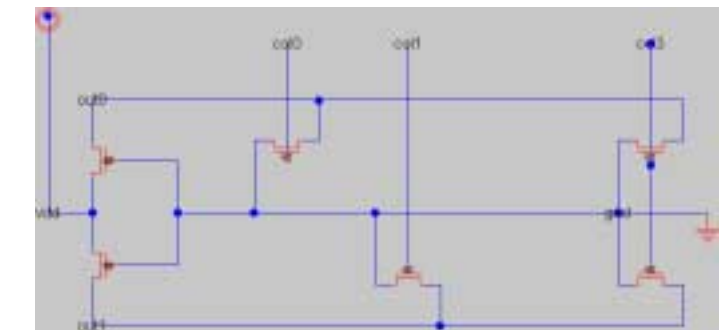
Rom_block_sm_0312



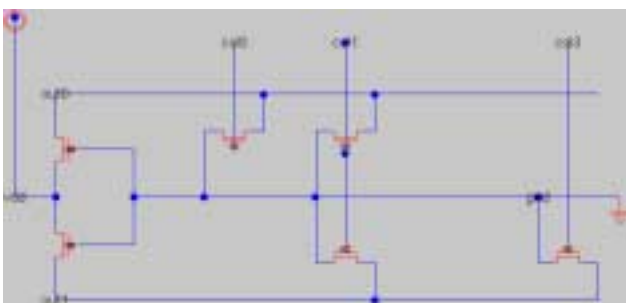
Rom_block_sm_1023



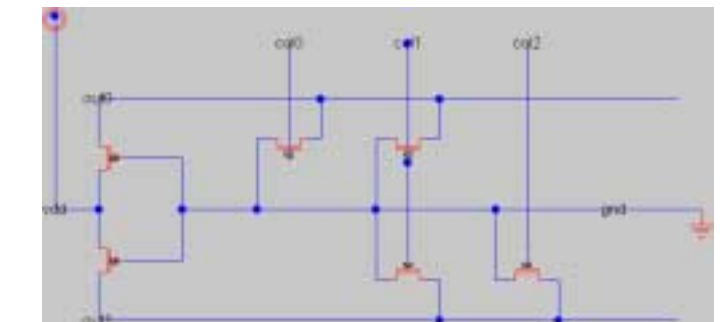
Rom_block_sm_1203



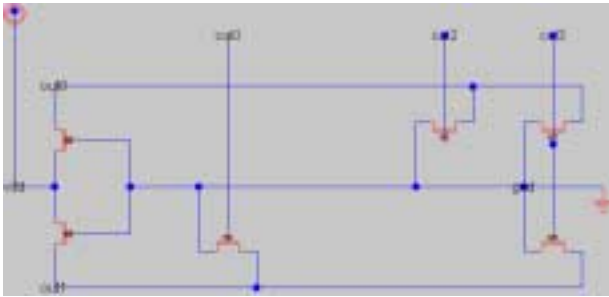
Rom_block_sm_1302



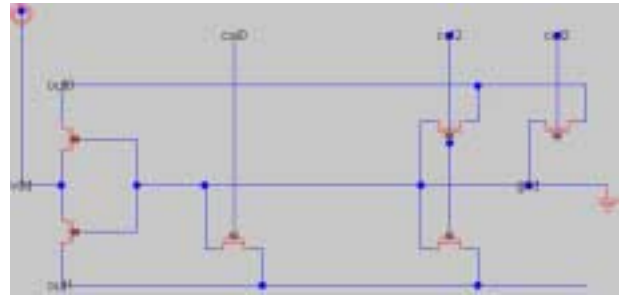
Rom_block_sm_1320



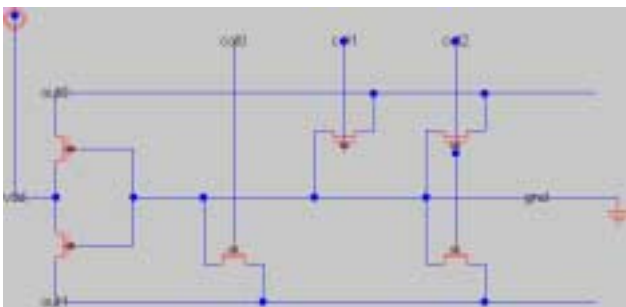
Rom_block_sm_2013



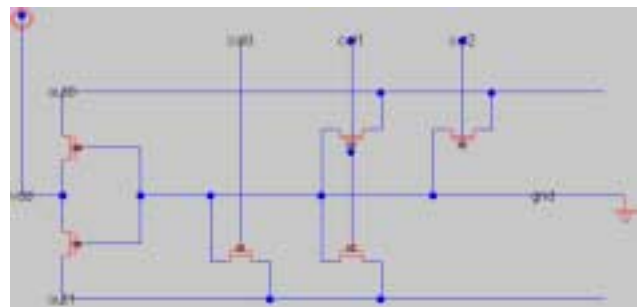
Rom_block_sm_2031



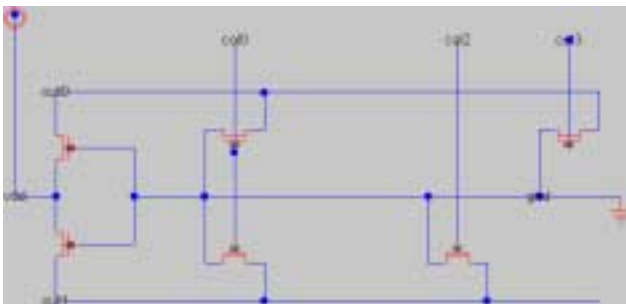
Rom_block_sm_2130



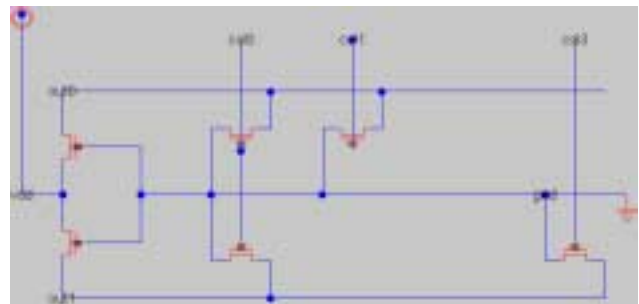
Rom_block_sm_2310



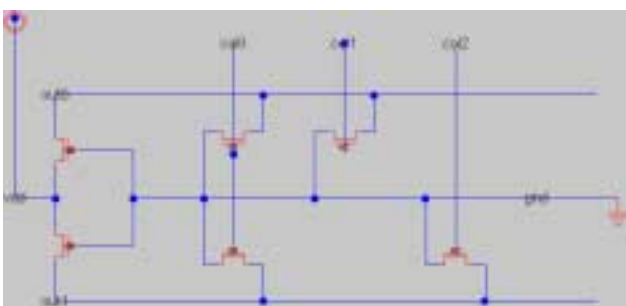
Rom_block_sm_3021



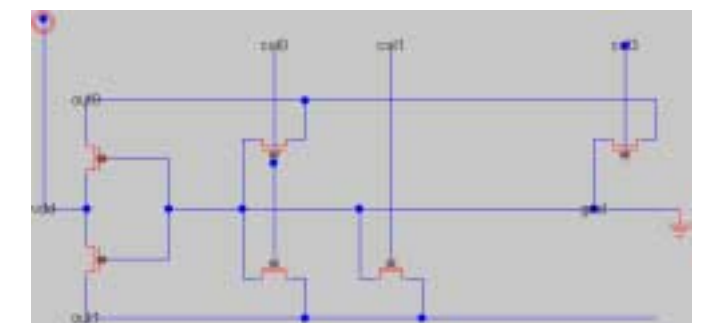
Rom_block_sm_3102



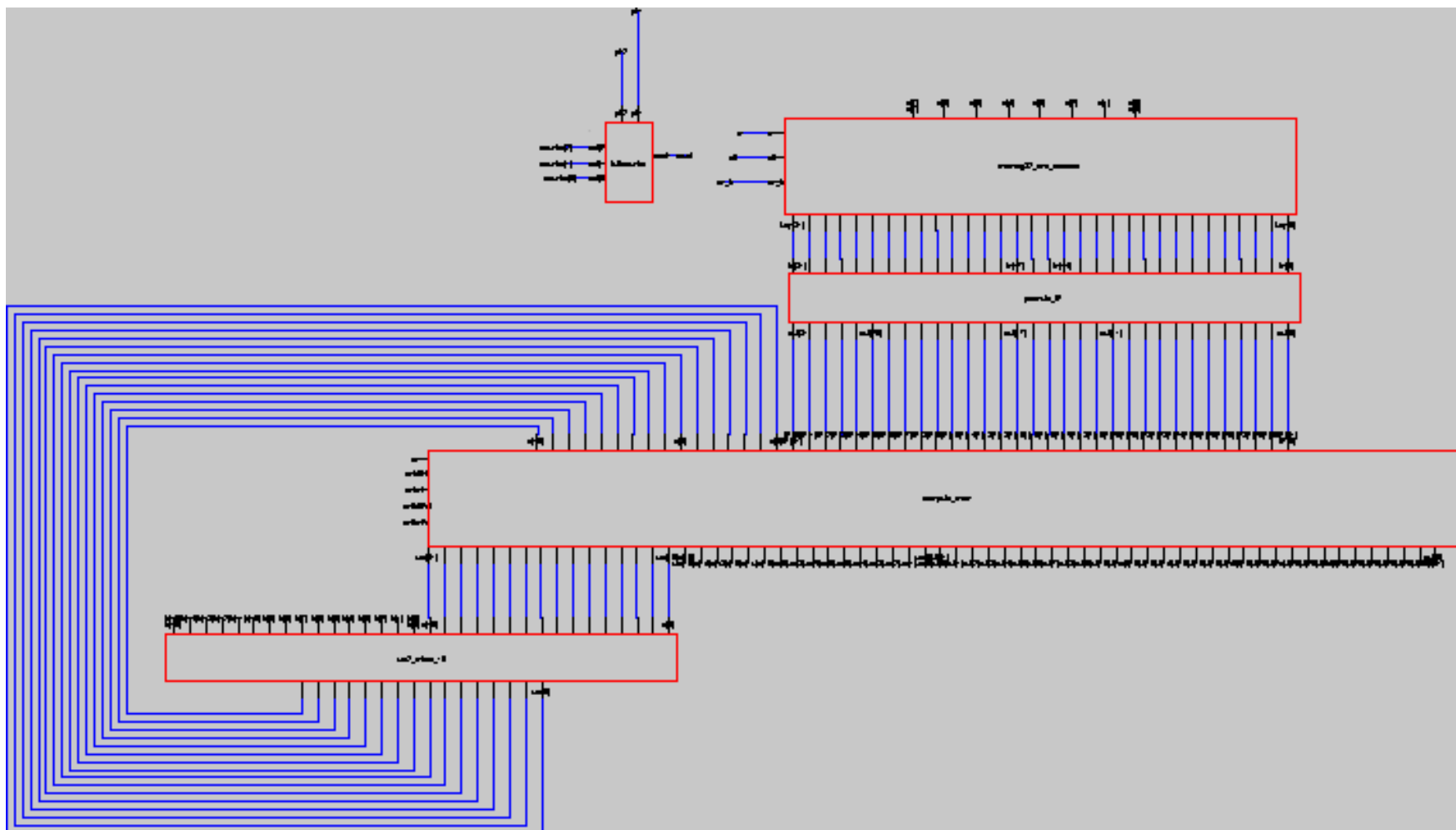
Rom_block_sm_3120



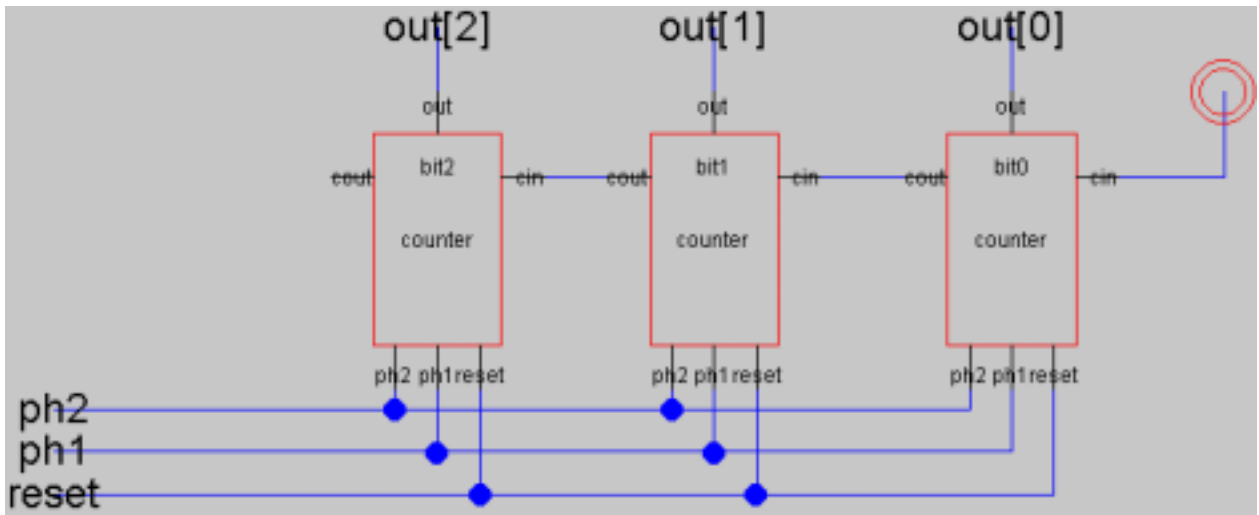
Rom_block_sm_3201



datapath

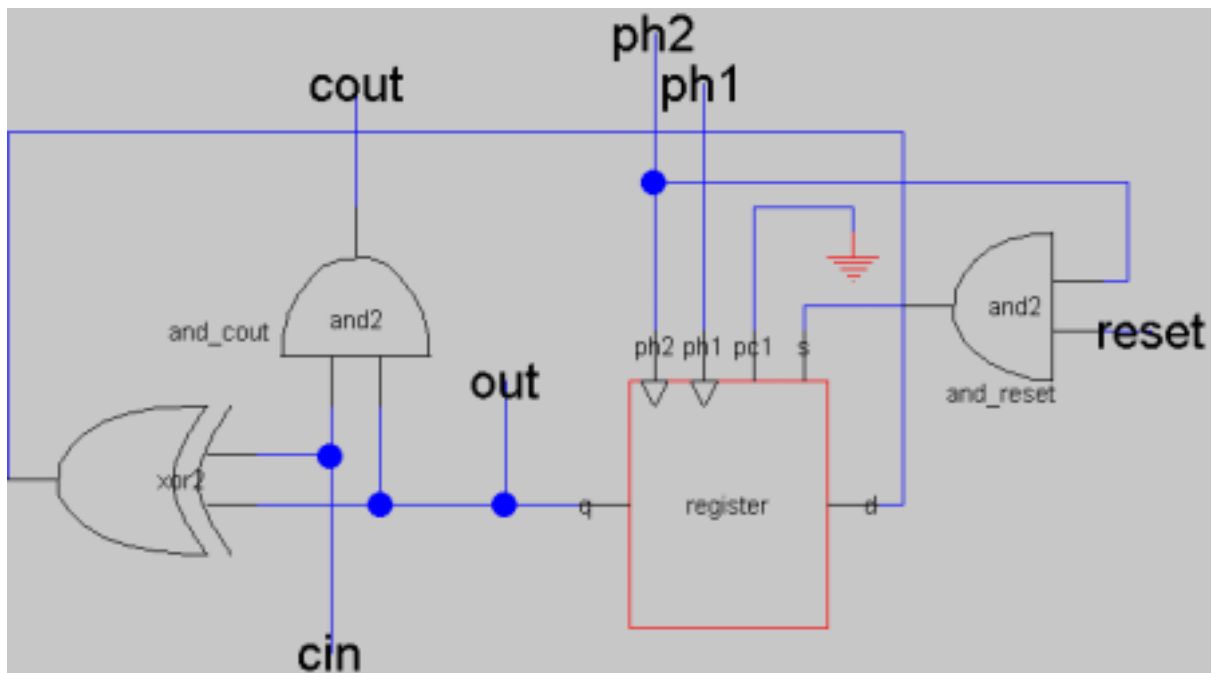


fullcounter - uses counter

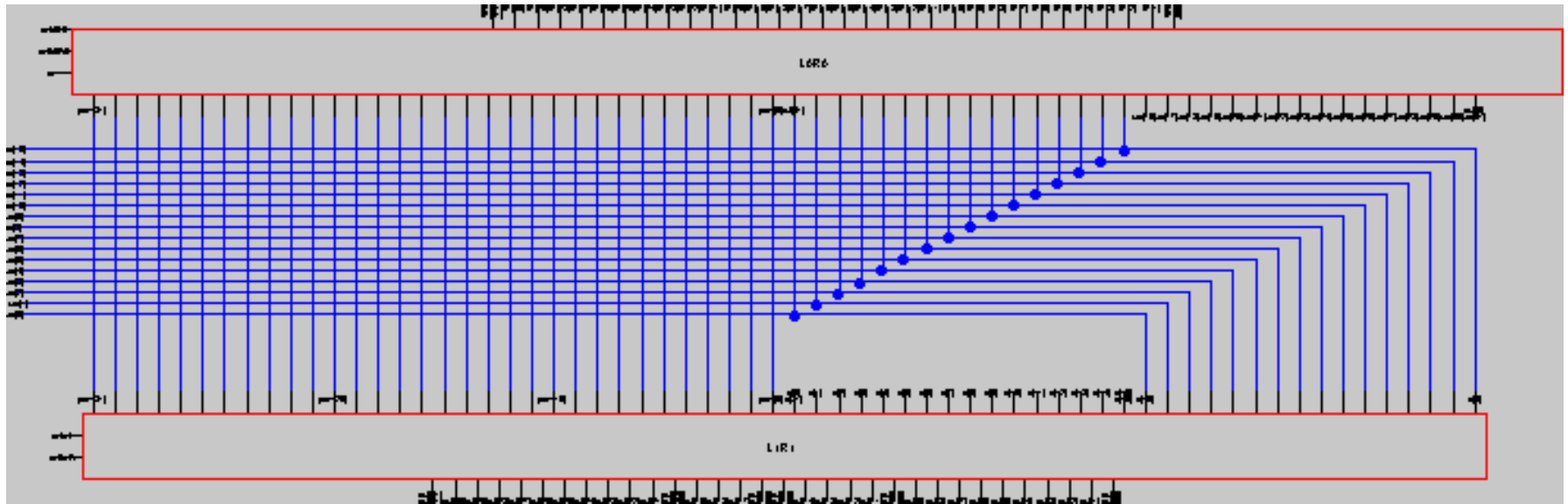


A synchronous counter consisting of an incrementer (XOR and AND) and a register. A 3-bit counter is used to count the number of calculation iterations during encryption.

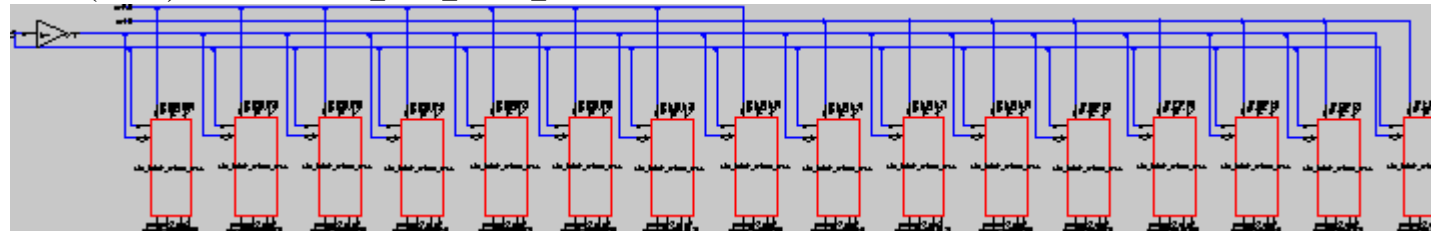
counter - uses register



compute_mem – major components are LOR0 and L1R1

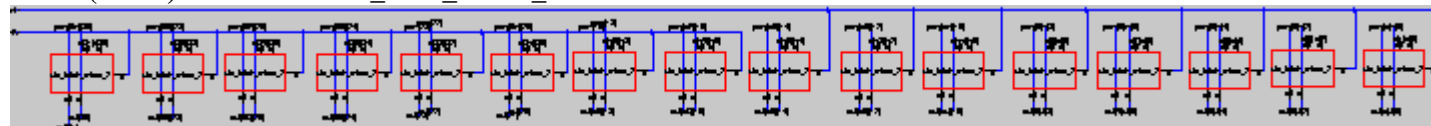


LOR0 (LOR0) - uses std_latch_mirror_mux



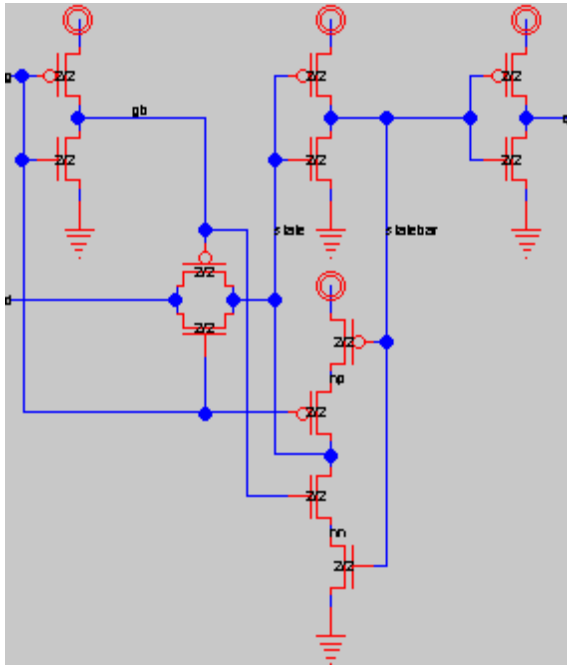
registers retain the value of LOR0 and L1R1 until the clock phase enables the switching of values.

LOR0 (LOR0) - uses std_latch_mirror_2



Various latch designs used

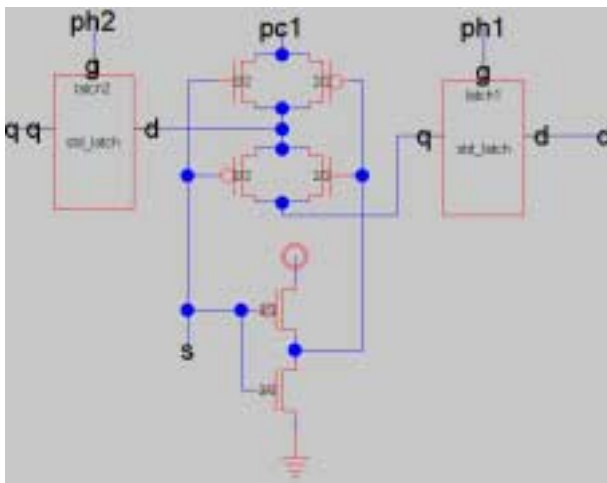
std_latch



Arrays of latch cells serve as registers in the chip to hold input, key, and output data. Furthermore, a set of intermediate registers is used to store data as it proceeds through the 8-iteration calculation cycle. It is necessary that the process be able to read from all the bits in a given register at once, since the computation requires calculating all bits at once for a given data block.

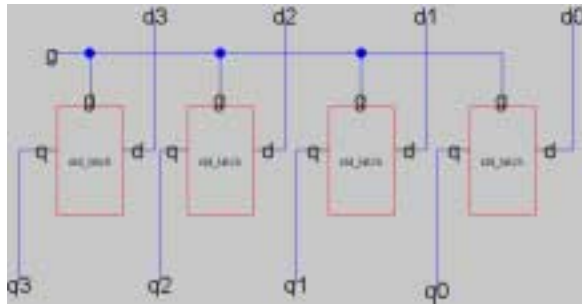
Excluding wiring, latch cells take up the most space in this chip design. Since the latches used were always grouped in large arrays to handle batches of 8 to 32 bits of data, the density of their arrangement was optimized. Three distinct latch designs were created.

register - uses std_latch



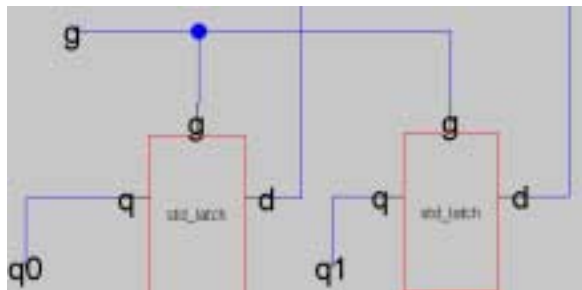
A regular flip-flop (two latches) with two transmission gates. The flip-flops form two 16-bit barrel shifters used to left shift the encryption key data. The flip-flops must be able to select between initial key data and the neighboring flip-flop.

std_latch_mirror - uses std_latch



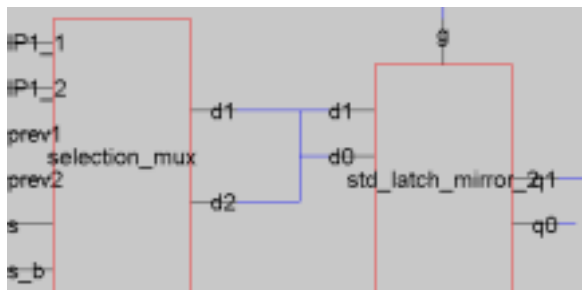
A mirrored, 1x2 array of latches. These latches need only be able to store data. These latches form a larger 16x2 array of latches that are used to store data during encryption.

std_latch_mirror_2 - uses std_latch



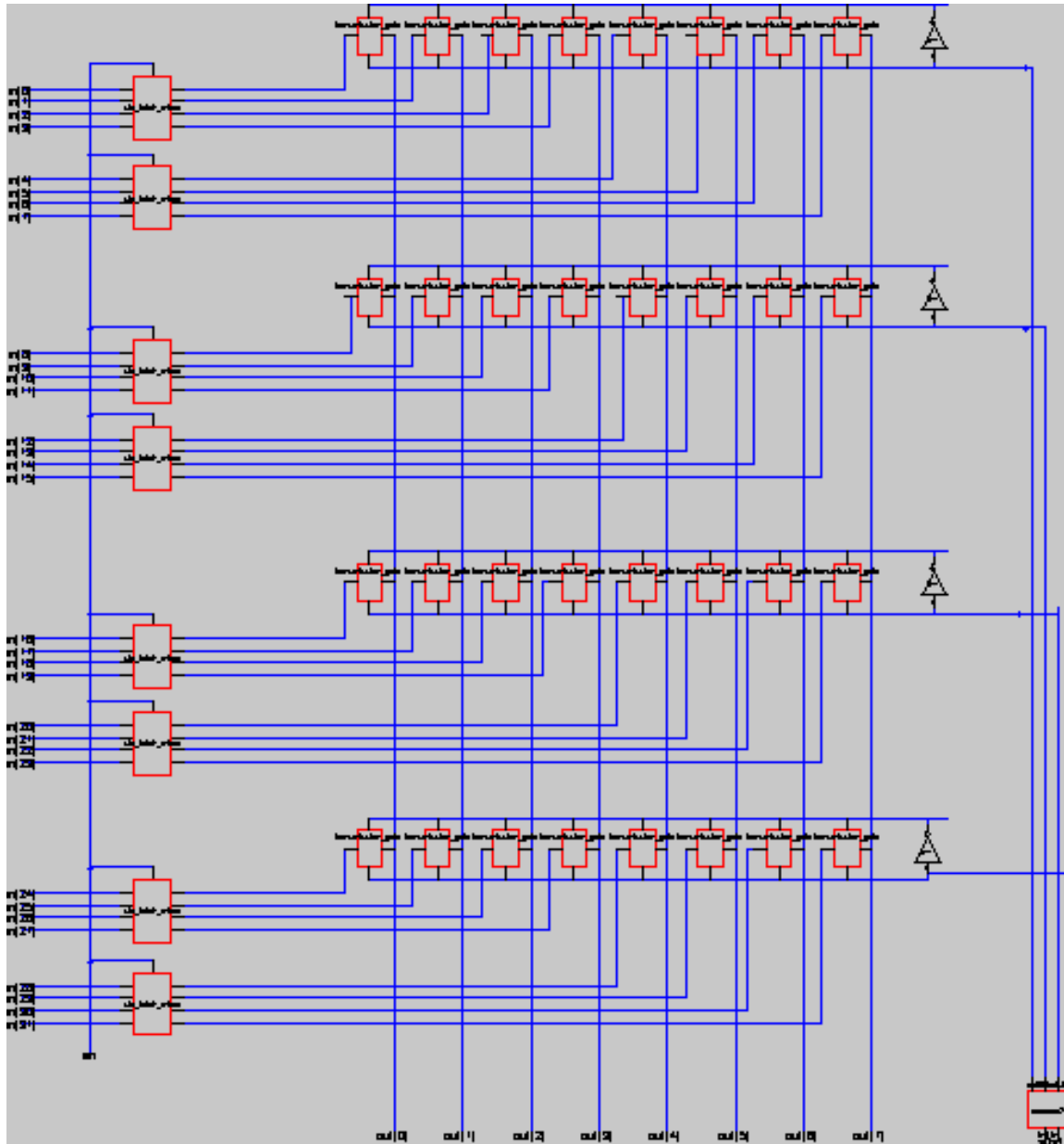
A mirrored, 1x4 array of latches. These latches form a larger 8x4 array of latches that are used to store the 28-bit encryption key and 32-bit word for encryption.

std_latch_mirror_mux - uses std_latch

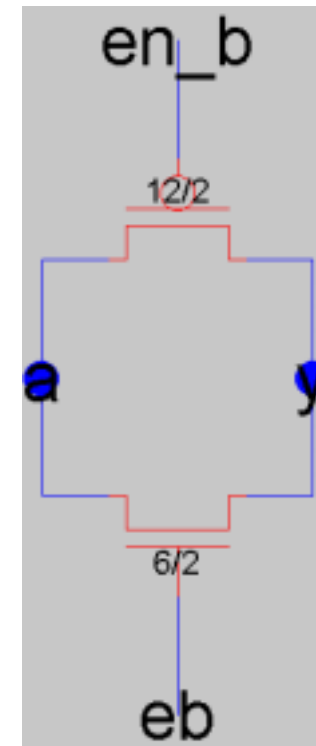


A mirrored, 1x2 array of latches. Each latch is fitted with two transmission gates at the top to select between two possible inputs to the latch. This leaf cell is used to form the 16x2 array of latches that are used to store data during encryption. These latches must have the ability to choose between initial input data or data computed during computation.

memreg32out - uses transmission_gate, std_latch_mirror



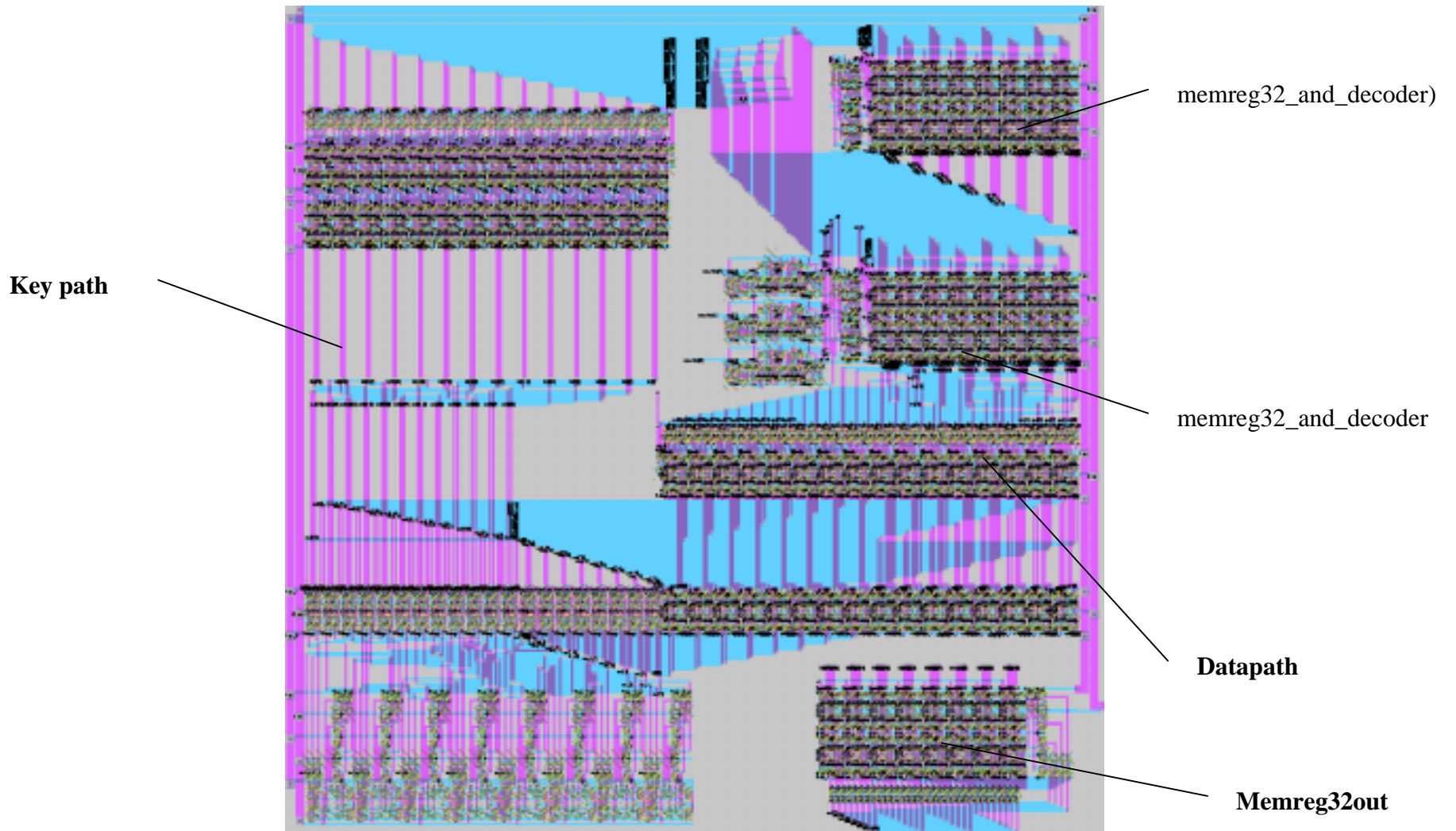
Transmission_gate



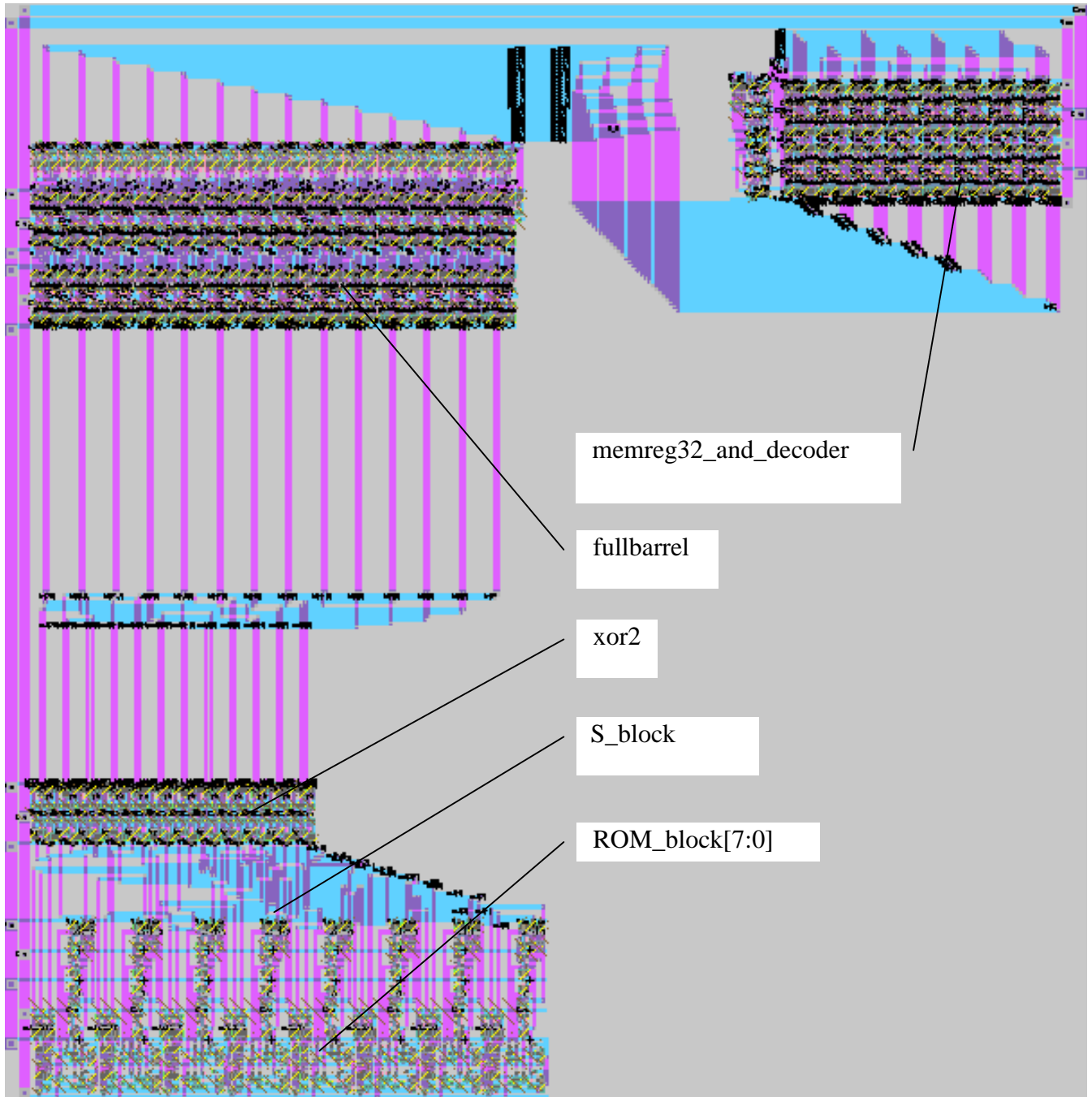
Similar to the input register, memreg32out takes in 32 bits of the output and partitions it into 8-bit segments.

LAYOUT

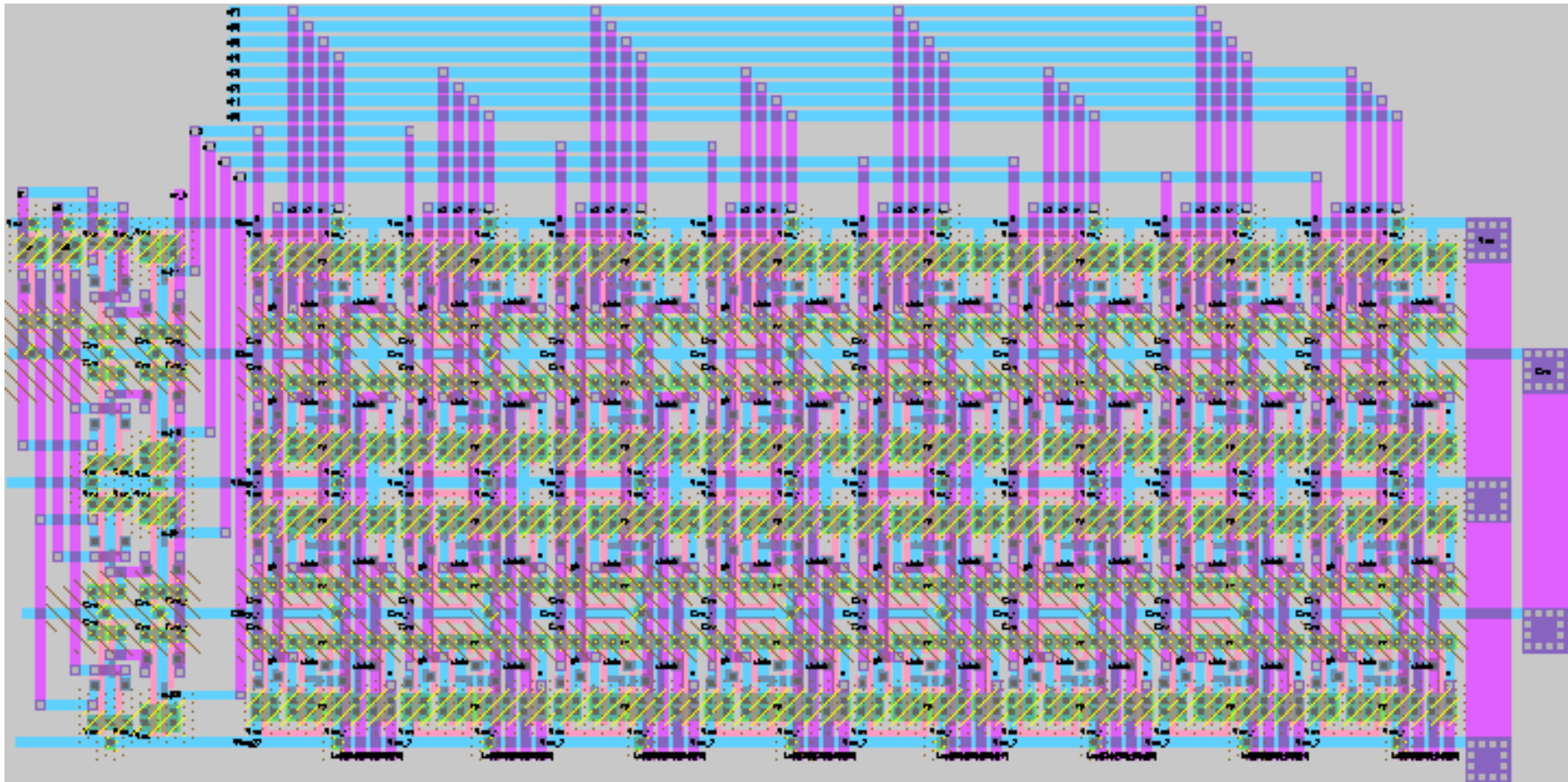
6.1. DES Chip Layout (des 32)



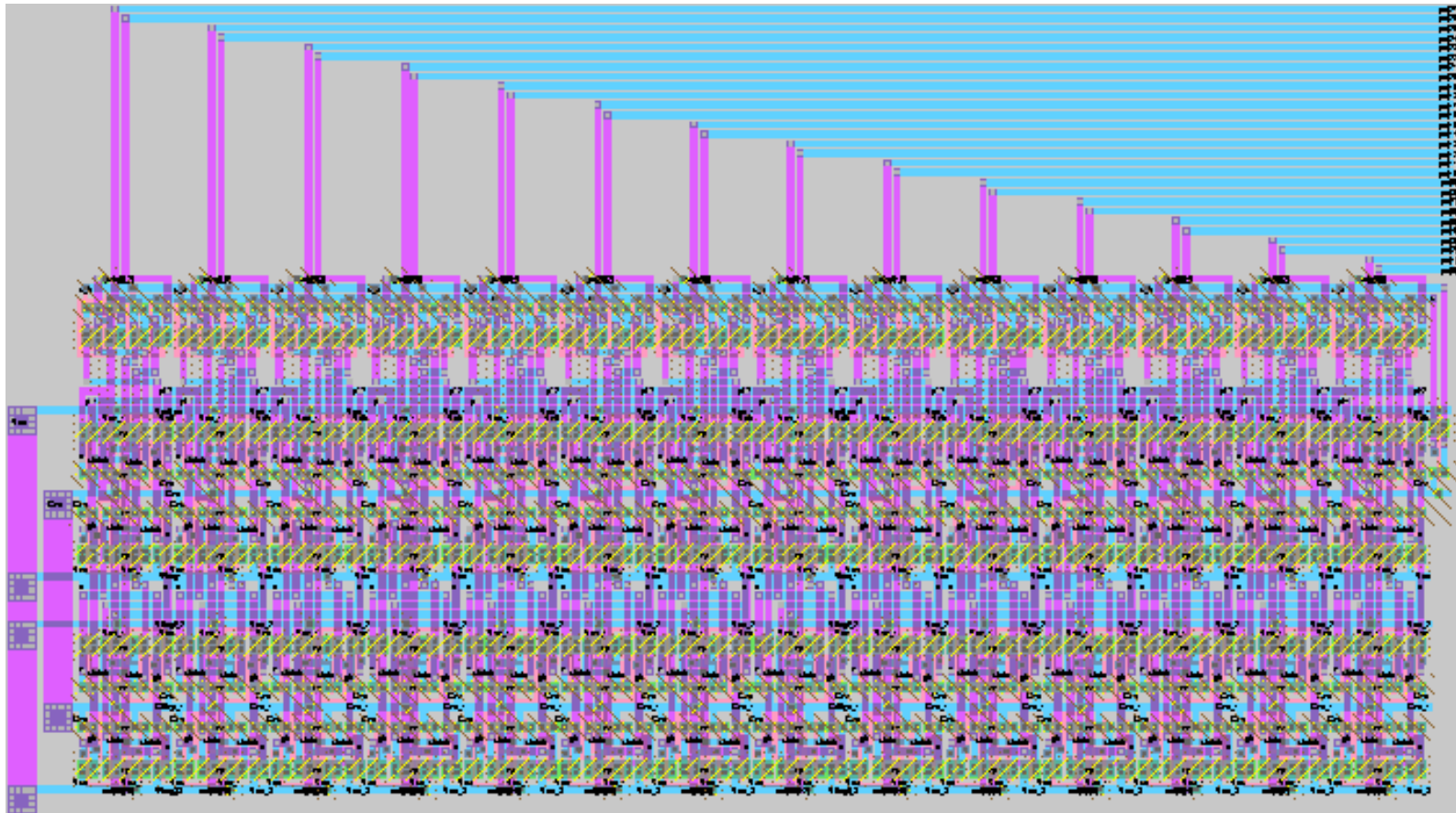
keypath



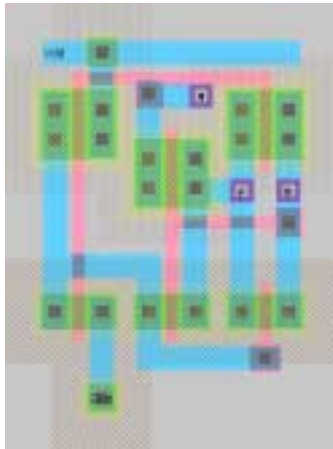
memreg32_and_decoder



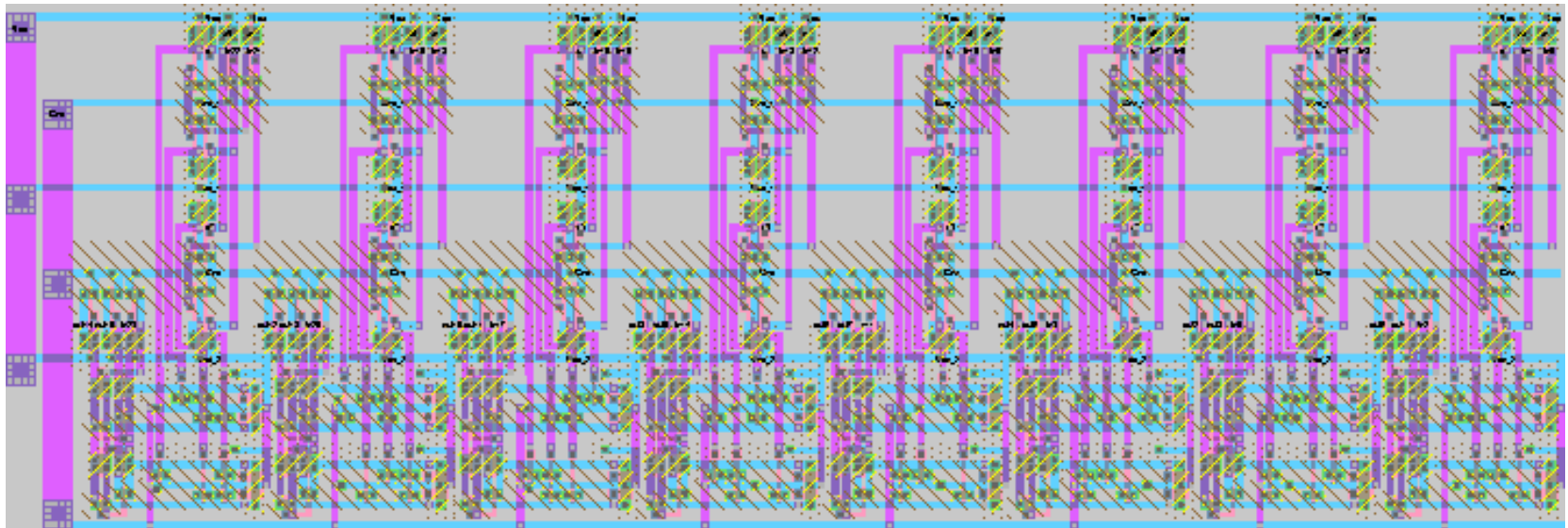
fullbarrel



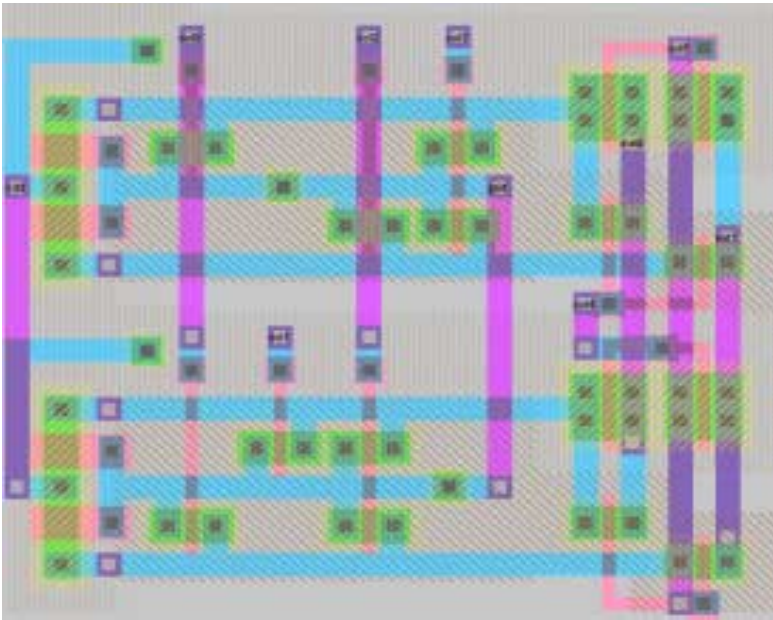
xor2



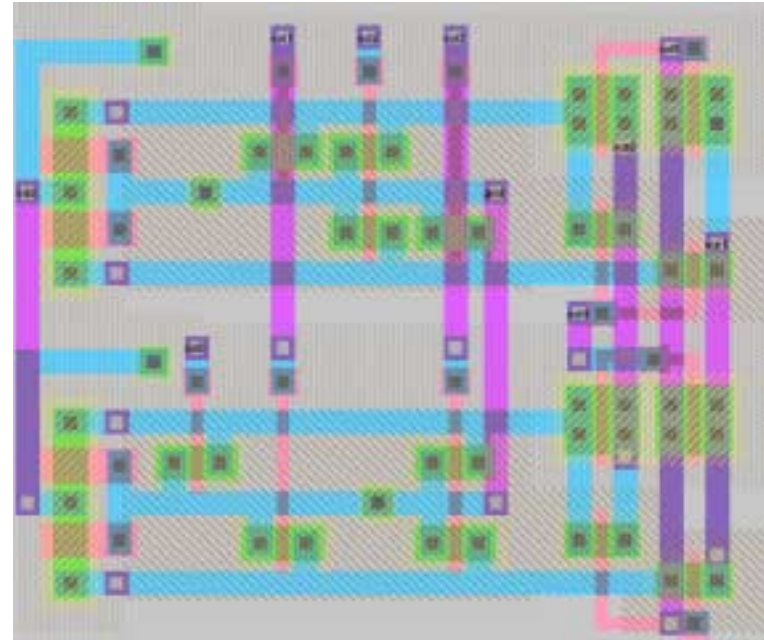
S_block



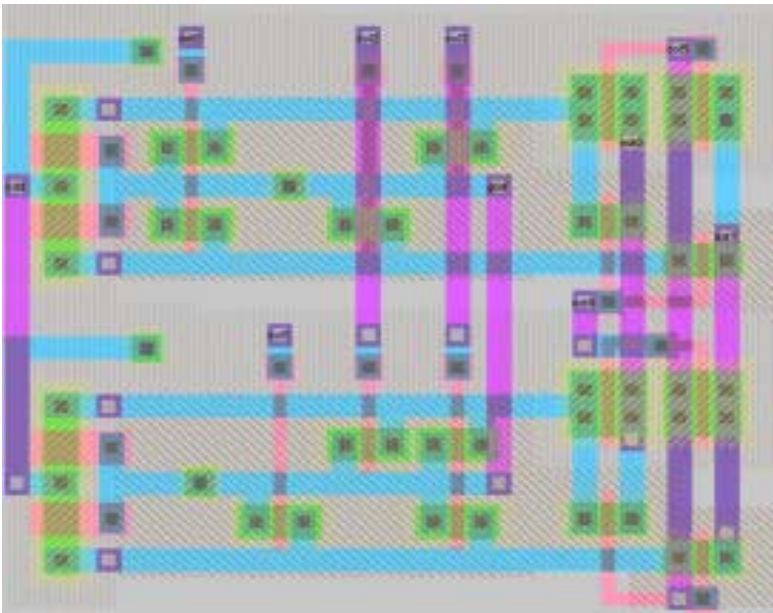
(s_block_0)



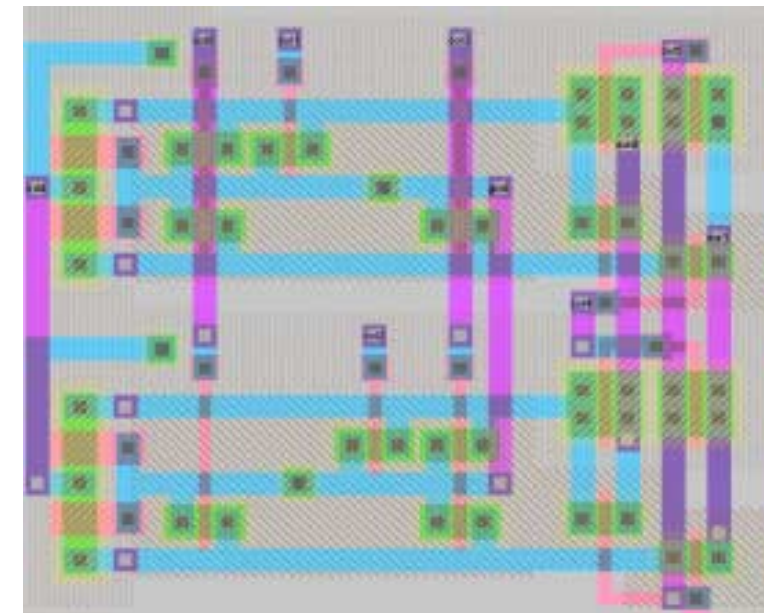
(s_block_1)



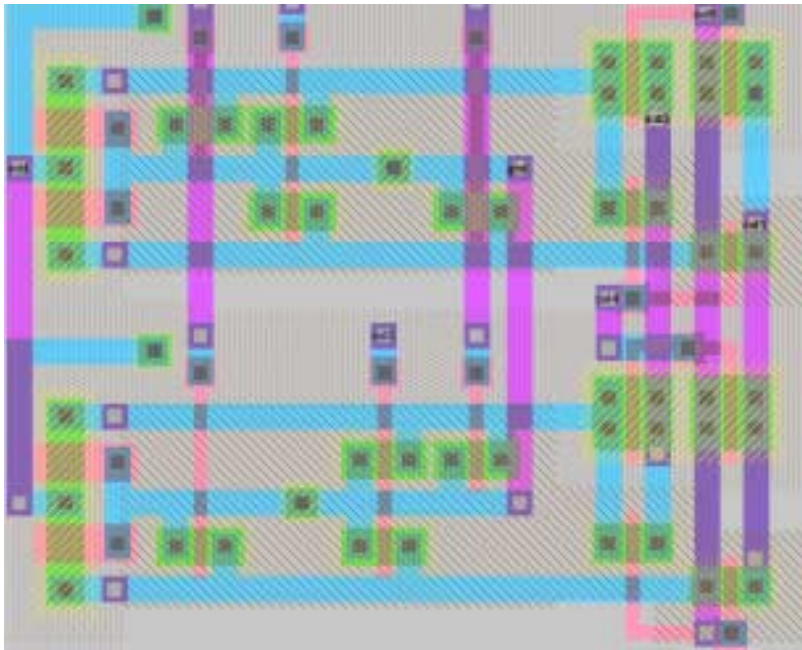
(s_block_2)



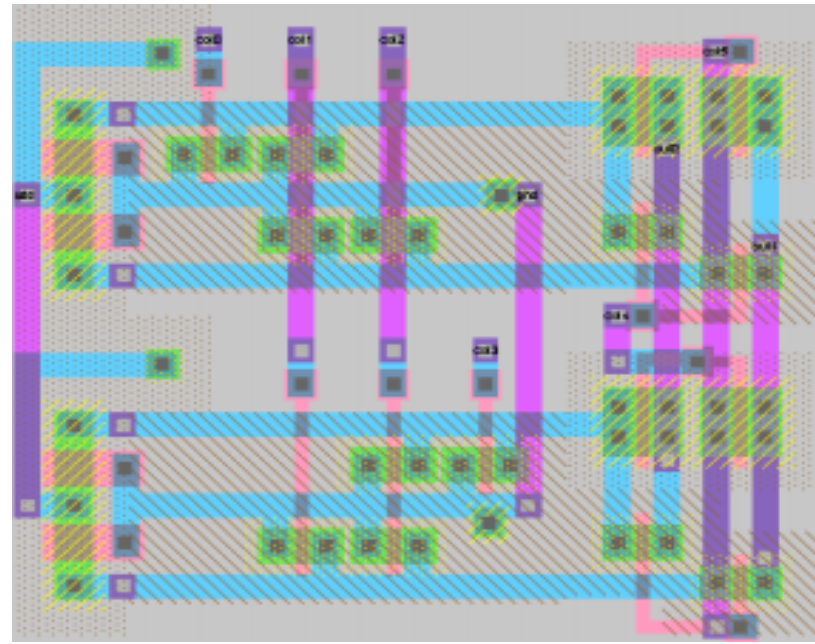
(s_block_3)



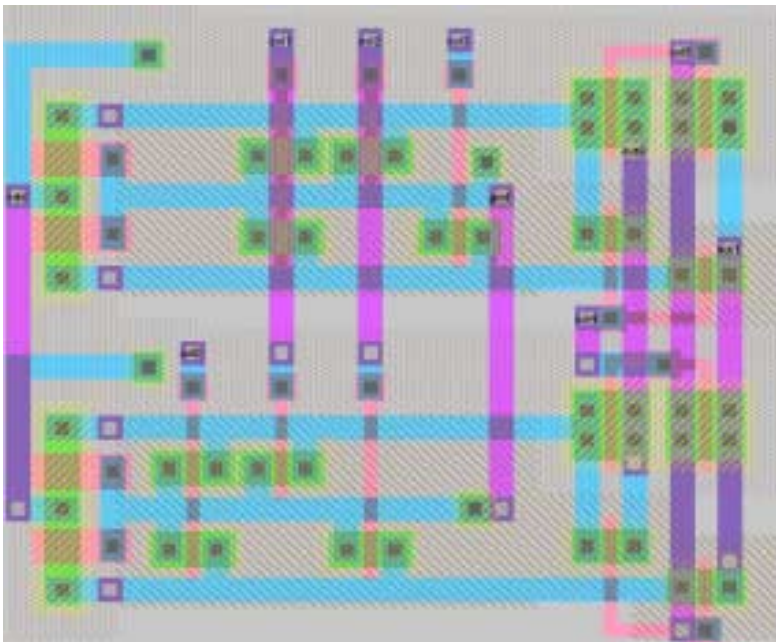
(s_block_4)



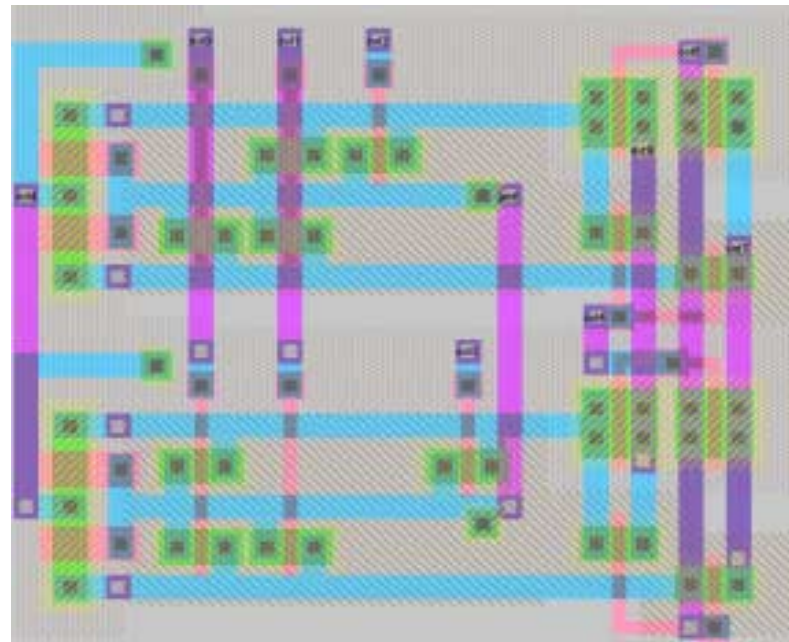
(s_block_5)



(s_block_6)

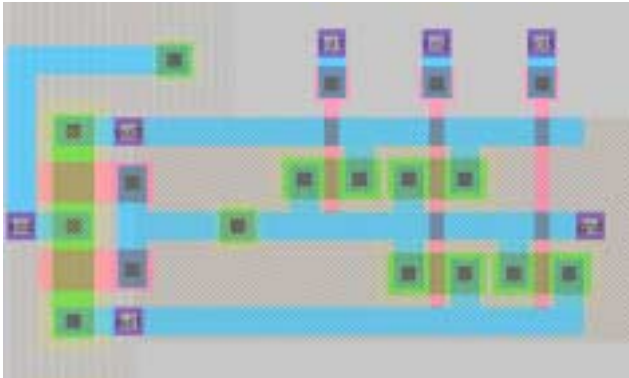


(s_block_7)

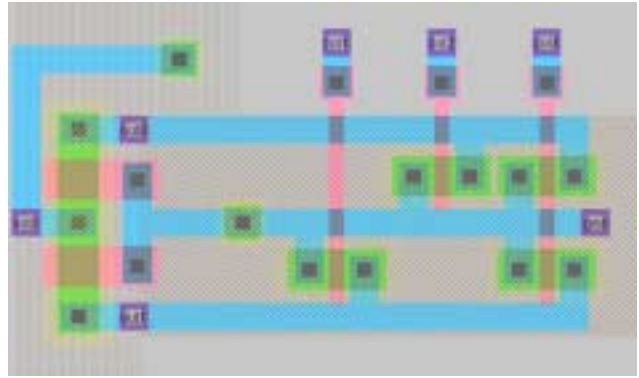


ROM_blocks

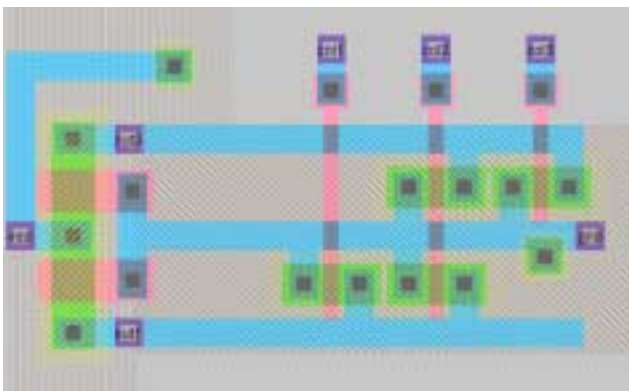
Rom_block_sm_0132



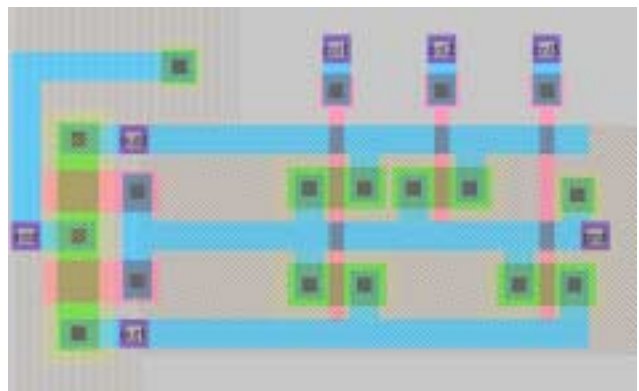
Rom_block_sm_0213



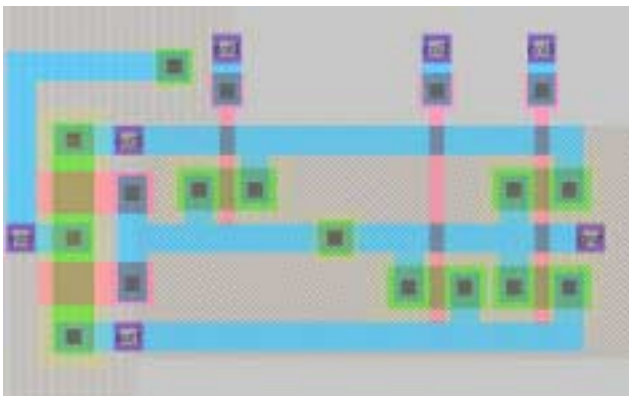
Rom_block_sm_0231



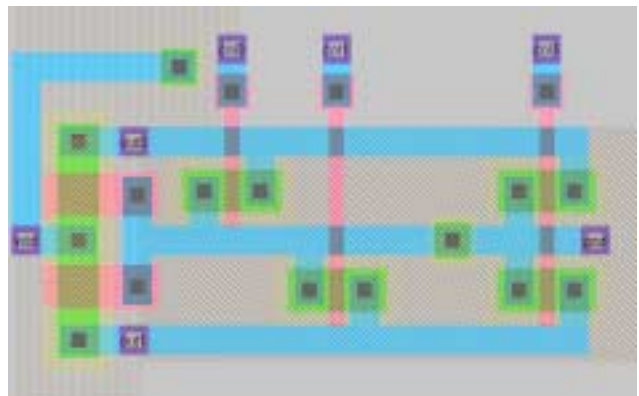
Rom_block_sm_0312



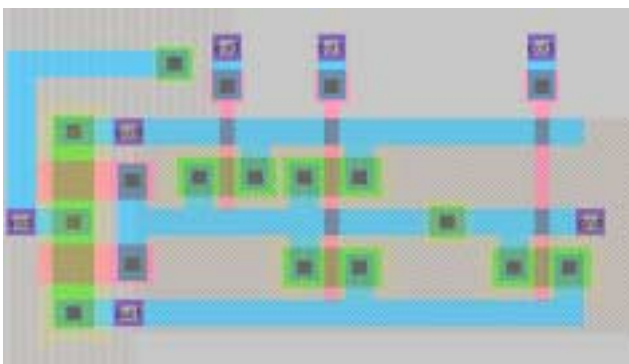
Rom_block_sm_1023



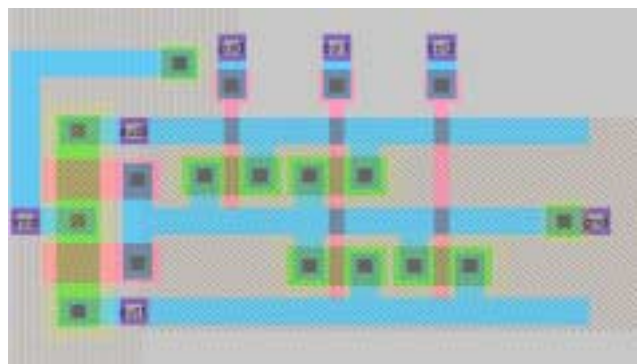
Rom_block_sm_1203



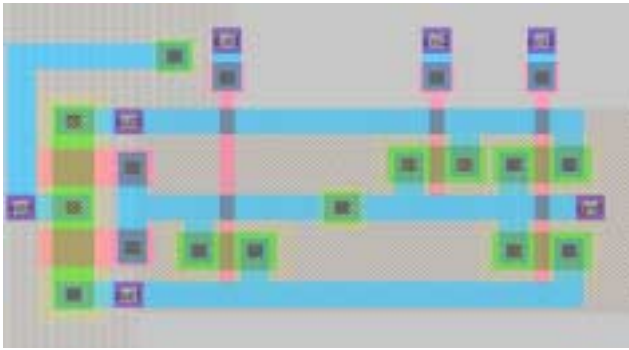
Rom_block_sm_1302



Rom_block_sm_1320



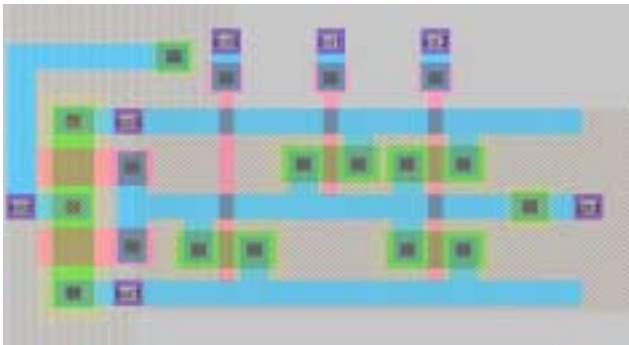
Rom_block_sm_2013



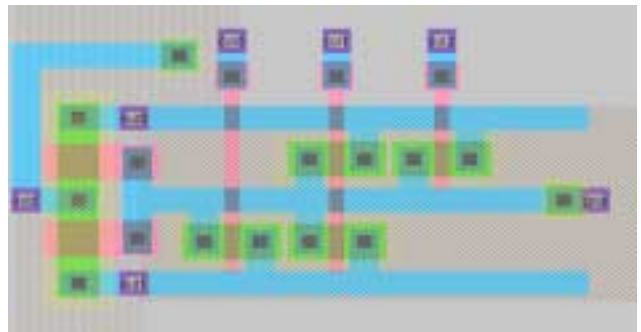
Rom_block_sm_2031



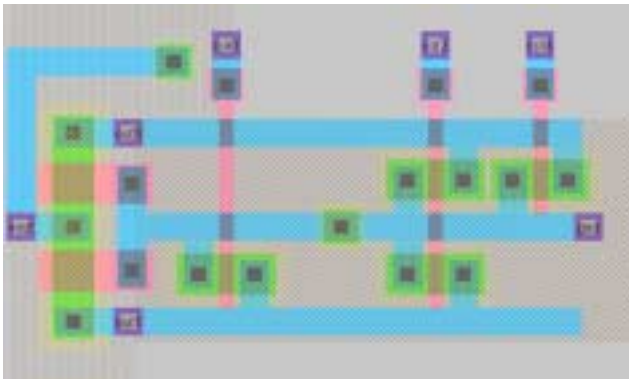
Rom_block_sm_2130



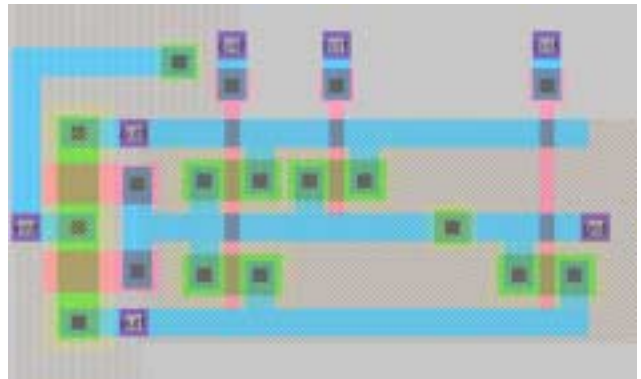
Rom_block_sm_2310



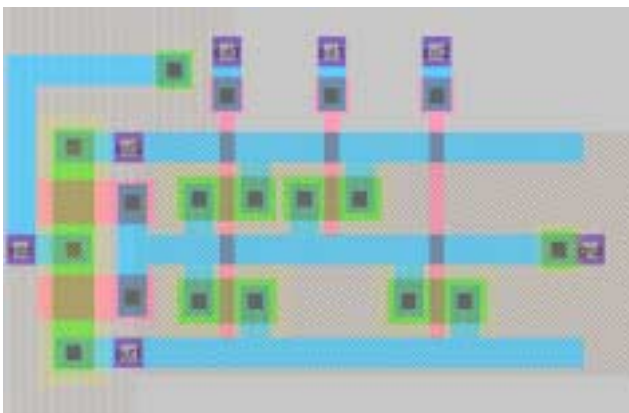
Rom_block_sm_3021



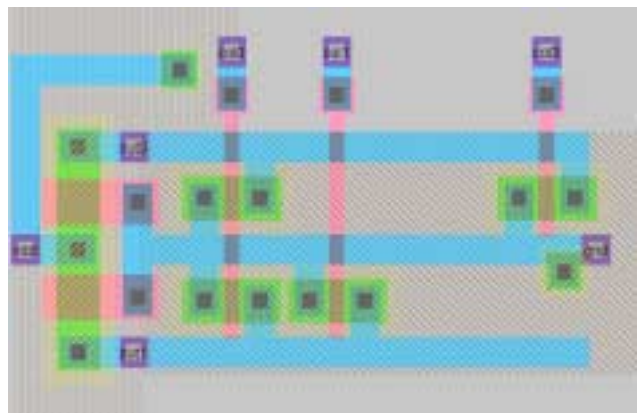
Rom_block_sm_3102



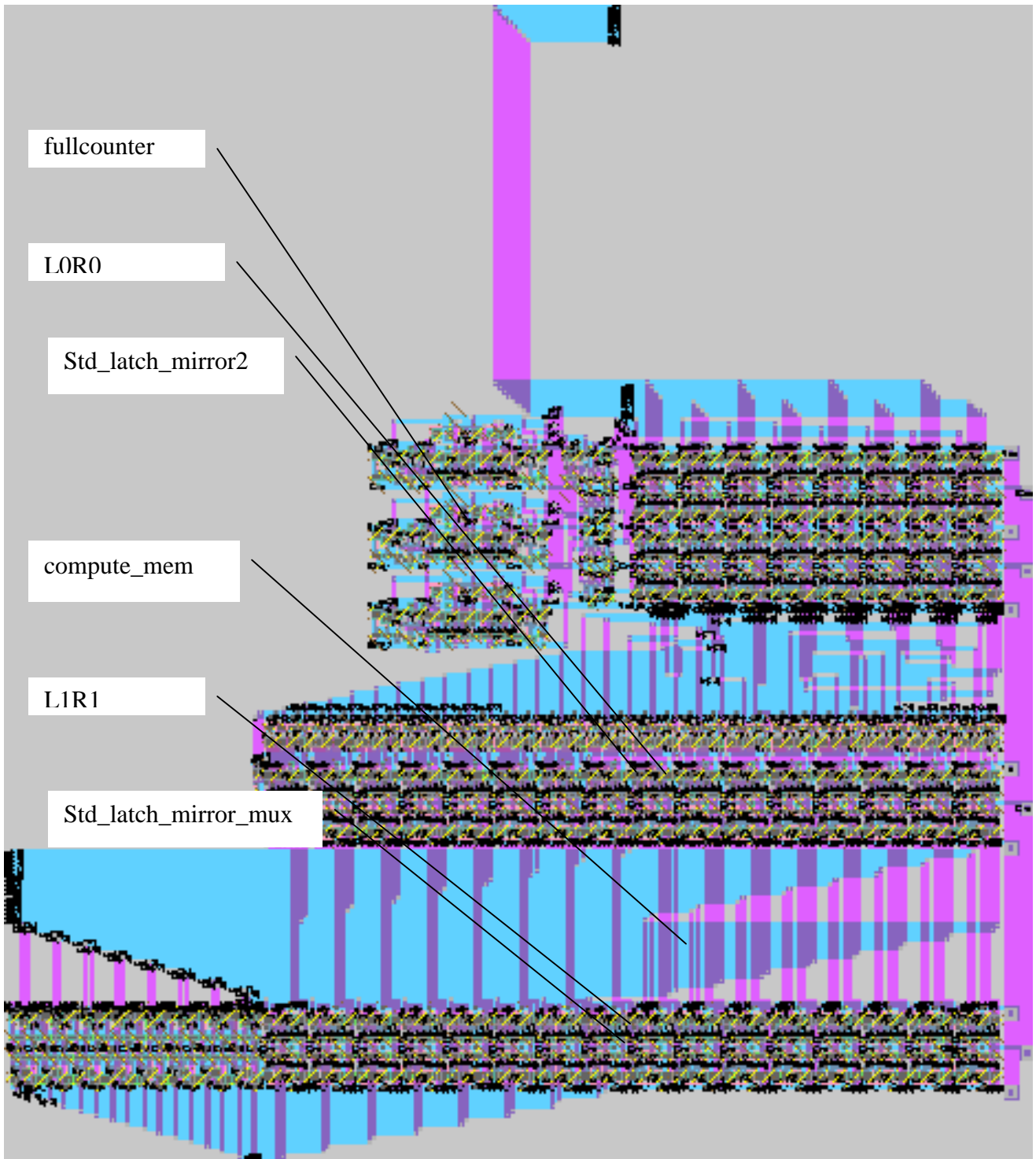
Rom_block_sm_3120



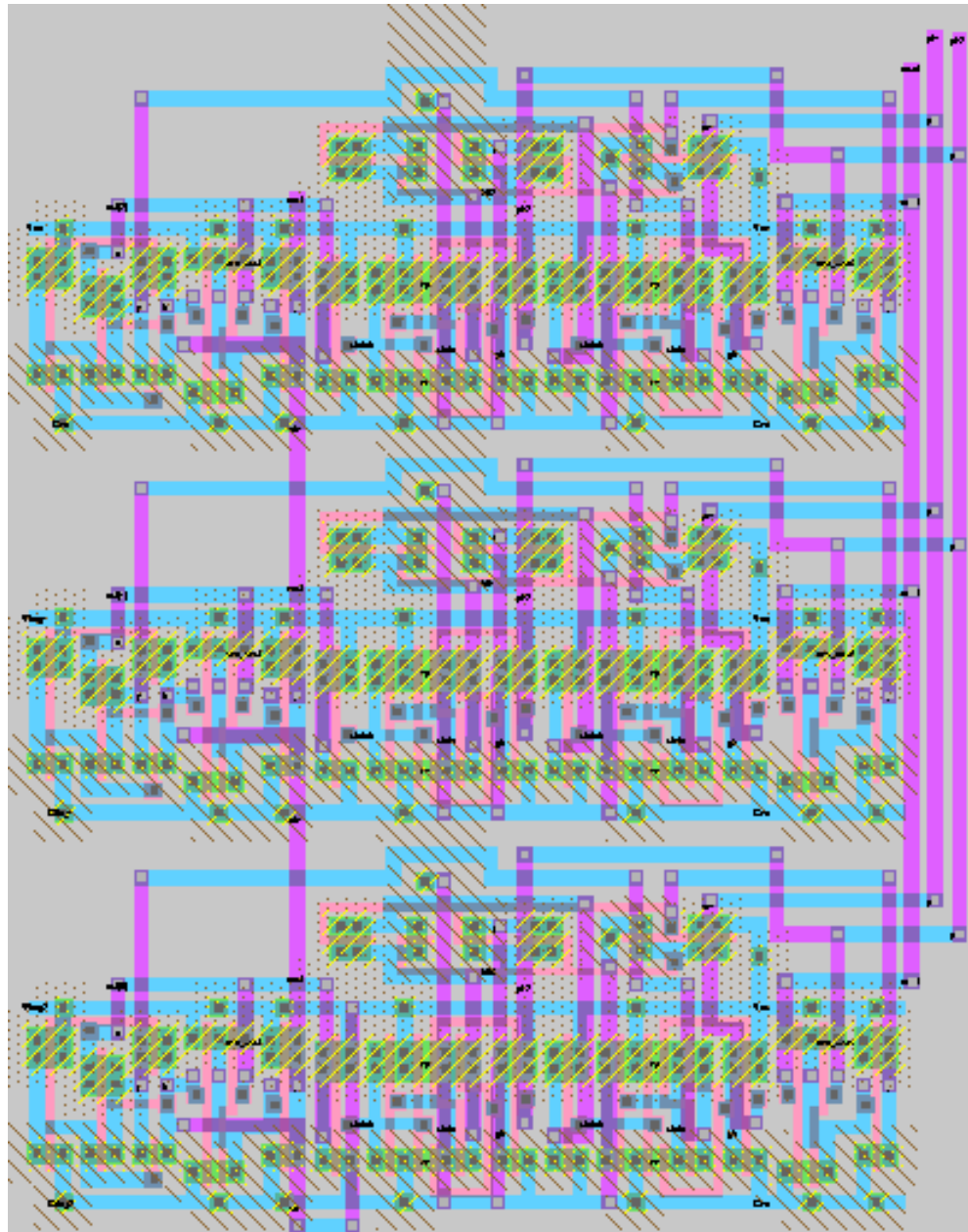
Rom_block_sm_3201



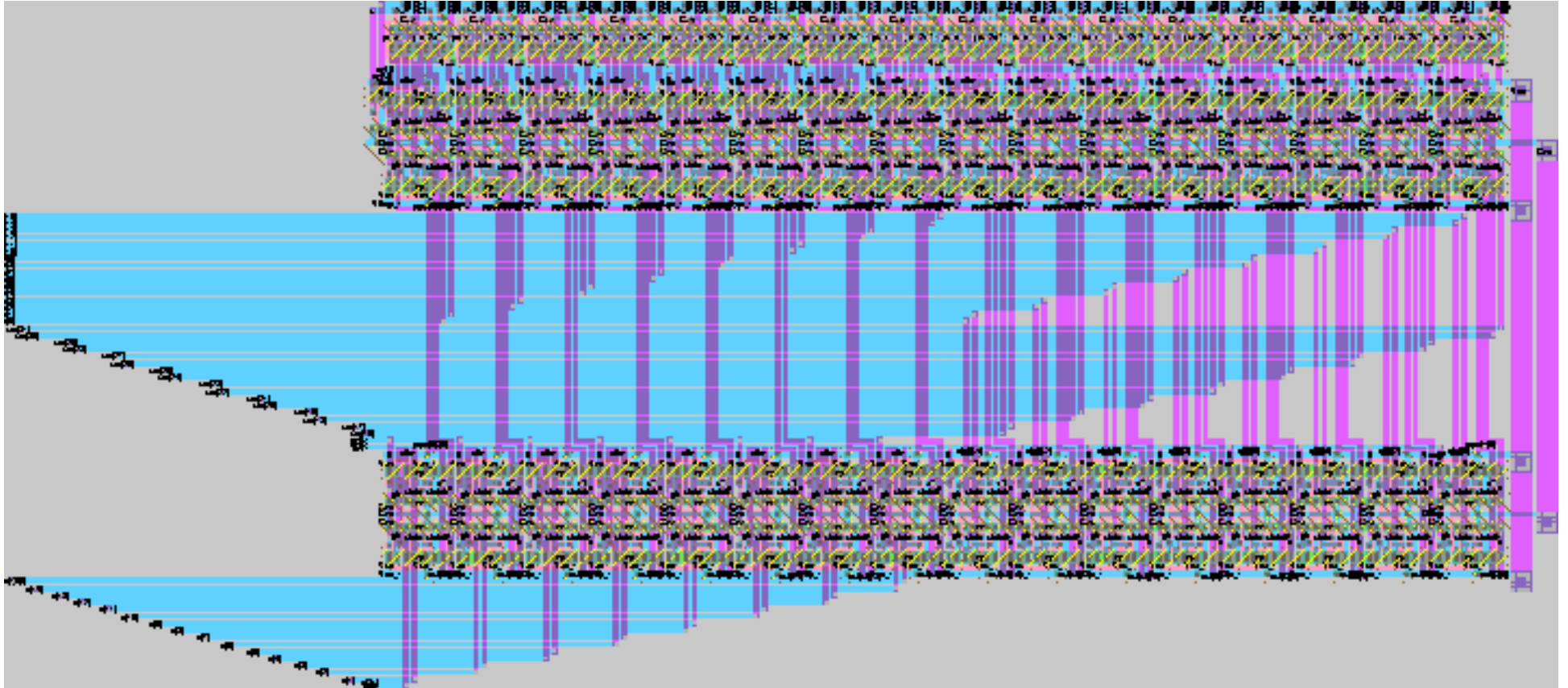
datapath



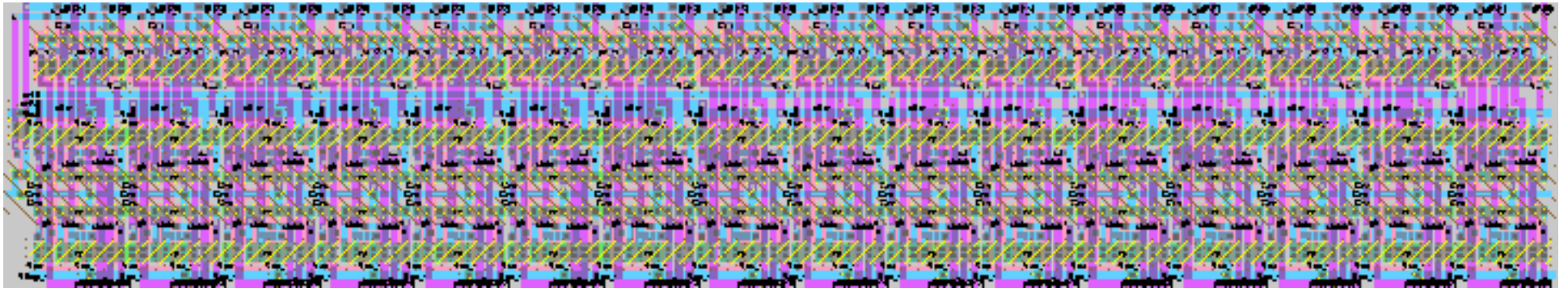
fullcounter



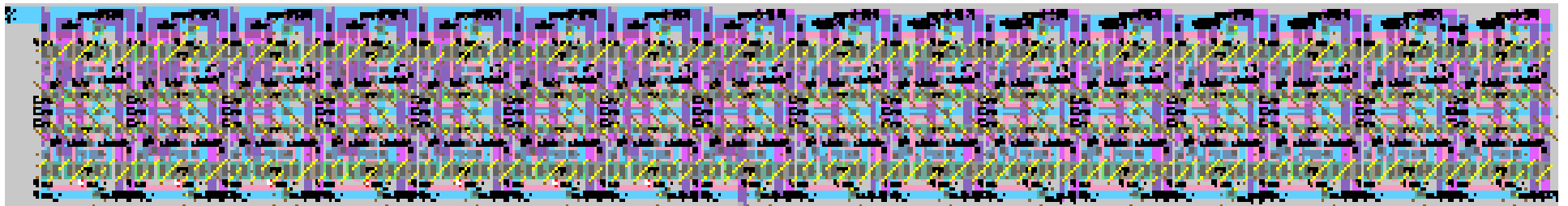
compute_mem



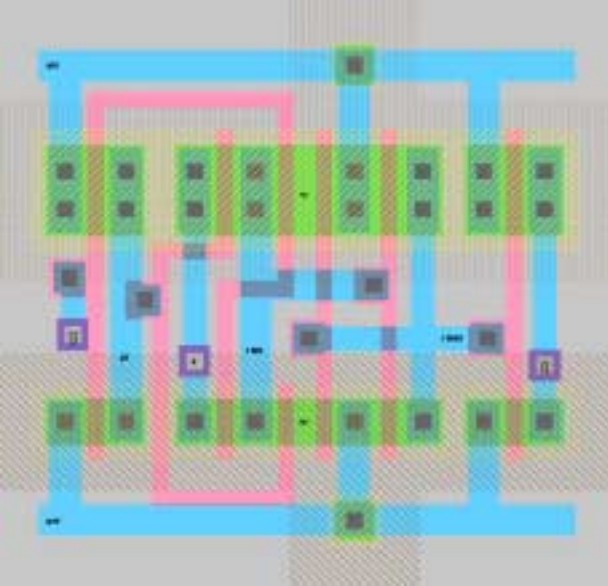
L0R0



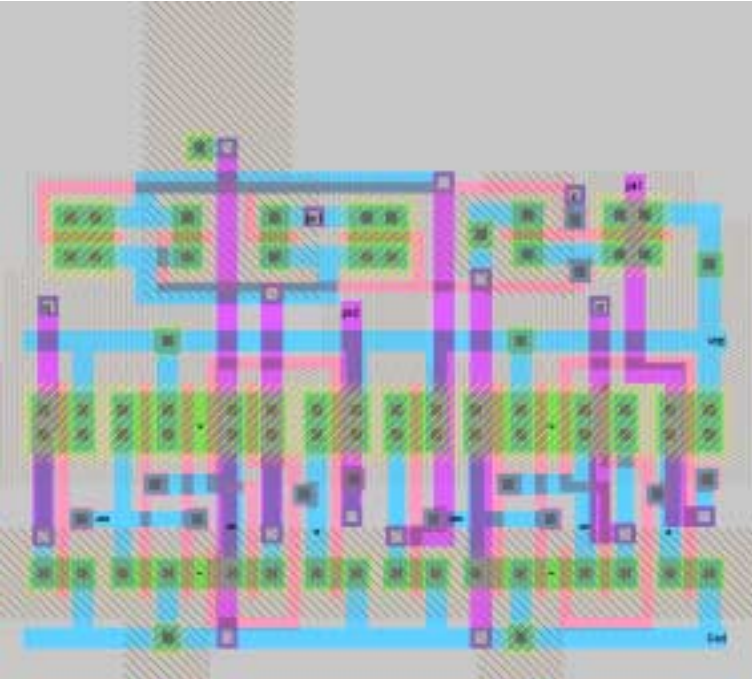
L1R1



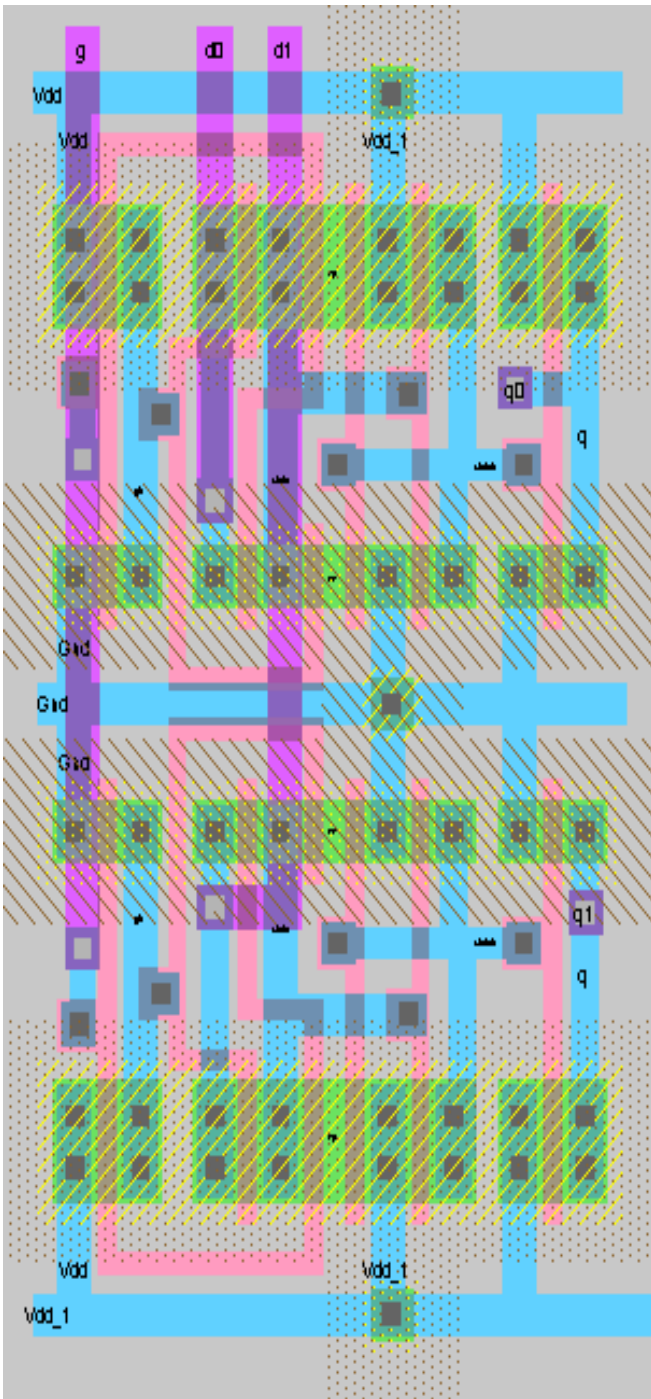
std_latch (latch modified in all latch designs above)



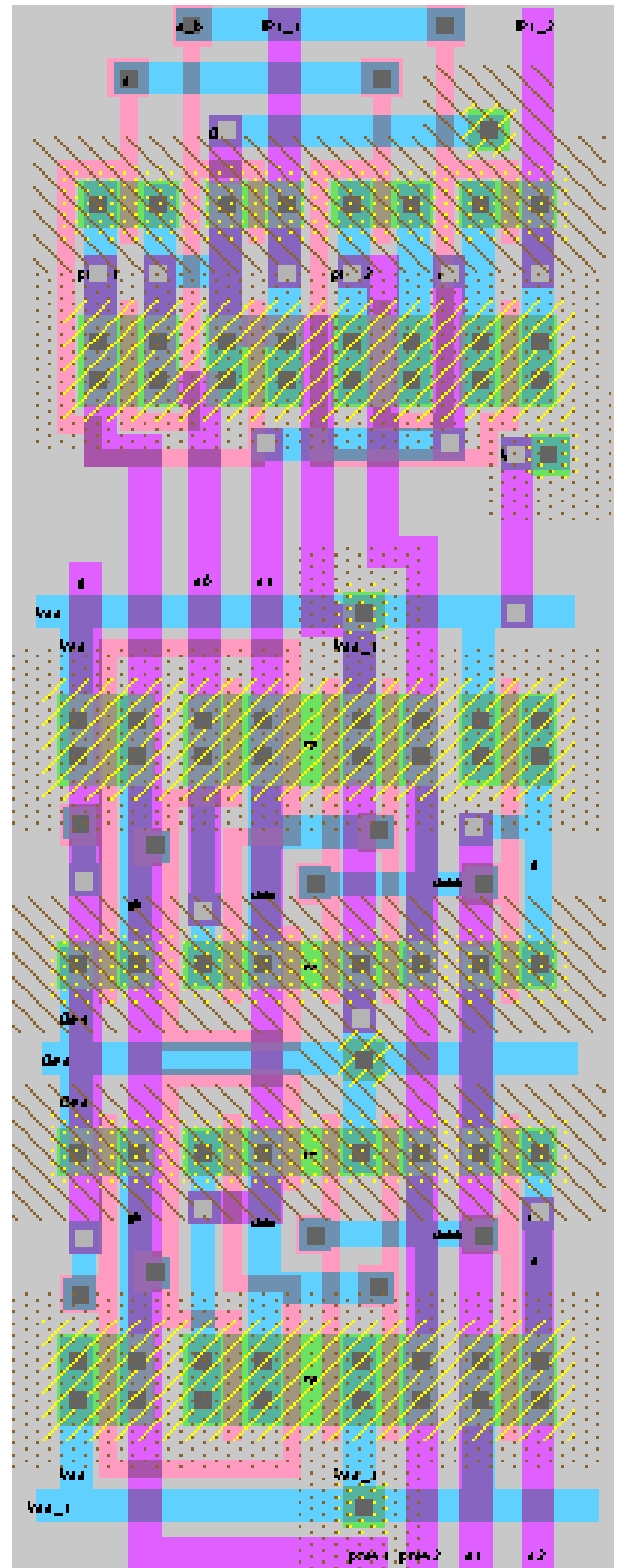
register



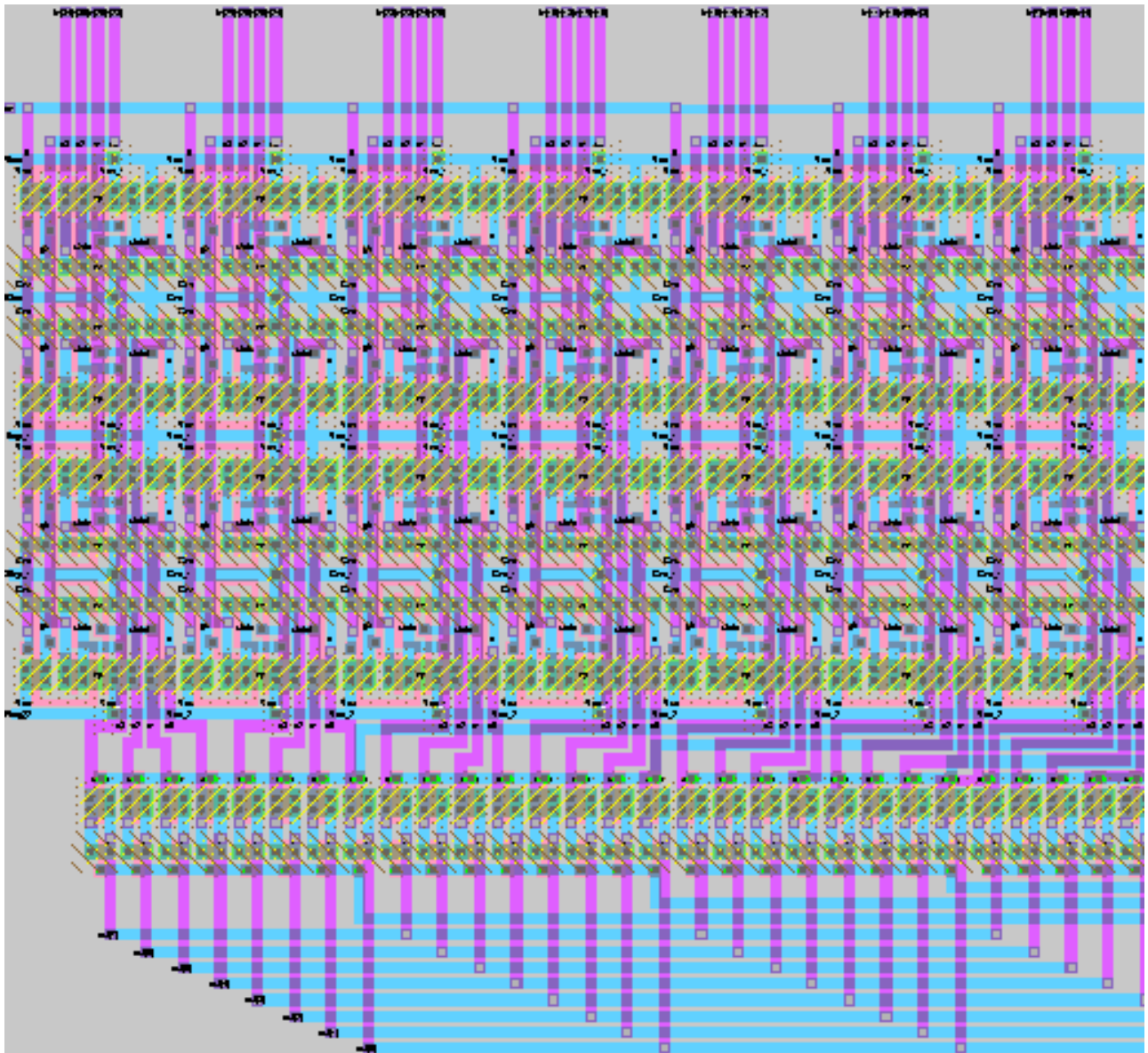
std_latch_mirror2 (latch associated with L0R0)



std_latch_mirror_mux (latch associated with L1R1)



Memreg32out



7.1 References

“Data Encryption Standard: Federal Information Processing Standards Publication.” Gaithersburg, MD: Computer Systems Laboratory, National Institute of Standards and Technology, 1993.

Appendix A | 32-bit DES Encryption verification program

```
/*
 * Input file specs:
 *
 * IP[0:31]
 * IPInv[0:31]
 * PC-1[0:27]
 * PC-2[0:23]
 * E[0:23]
 * P[0:15]
 * S_1[0:1][0:3]
 * S_2[0:1][0:3]
 * ...
 * S_8[0:1][0:3]
 *
 */

#include <iostream>
#include <fstream>

/*
 * Parameters
 */
#define CMP_WID      32
#define READ_WRDS   4
#define ITERS       8
#define CD_WID      28
#define S_IN_WID    24
#define S_OUT_WID   16

#define S_NUM        8 // number of tables
#define S_TAB_H      2 // number of rows
#define S_TAB_W       4 // number of columns
#define S_TAB_OUT_W  2 // bits of output

void permute(bool* input, bool** output, istream &p, int length);
void barrelShiftsLeft(bool* input, int length, int times);
void bitwiseXOR_ptr(bool* input1, bool** input2, bool* output, int length);
void makeSBlocks(bool*** &s, istream &paramFile);
int fileRead(bool* word, istream &file, int bytes);
char boolToChar(bool** word_ptr);

int main(int argc, char** argv)
{
    ifstream paramFile(argv[2]);
    ifstream keyFile(argv[3]);

    bool* key      = new bool[CMP_WID];
    bool* CD       = new bool[CD_WID];
    bool* input    = new bool[CMP_WID];
    bool* cmpPrev  = new bool[CMP_WID];
    bool* cmpCurr  = new bool[CMP_WID];
    bool* sIn      = new bool[S_IN_WID];
    bool* sOut     = new bool[S_OUT_WID];

    bool** IP      = new bool*[CMP_WID];
    bool** IPInv   = new bool*[CMP_WID];
    bool** PC1     = new bool*[CD_WID];
```

```

bool** PC2 = new bool*[S_IN_WID];
bool** E = new bool*[S_IN_WID];
bool** P = new bool*[S_OUT_WID];

bool** K;

bool**** s;

/*****
 * Create permutation blocks
 */

/*
 * Make IP permutation block
 */
permute(input, IP, paramFile, CMP_WID);

/*
 * Make IPInv permutation block
 */
permute(cmpPrev, IPInv, paramFile, CMP_WID);

/*
 * Make PC-1 permutation block
 */
permute(key, PC1, paramFile, CD_WID);

/*
 * Make PC-2 permutation block
 */
permute(CD, PC2, paramFile, S_IN_WID);

/*
 * Make E permutation block
 */
permute(cmpPrev, E, paramFile, S_IN_WID);

/*
 * Make P permutation block
 */
permute(sOut, P, paramFile, S_OUT_WID);

/*
 * Make S-function blocks
 */
makesBlocks(s, paramFile);

/*
 * Read in key and init CD
 */
fileRead(key, keyFile, READ_WRDS);
for (int i=0; i<CD_WID; i++)
    CD[i] = *(PC1[i]);

cerr << "The key is:\n";
for (int i=CMP_WID-1; i>=0; i--)
    cerr << key[i];
cerr << endl << endl;

// cerr << "After PC1, it is:\n";
// for (int i=27; i>=0; i--)
//     cerr << *(PC1[i]);
// cerr << endl << endl;

/*
 * Generate the keys

```

```

*/
K = new bool*[ITERS];
for (int i=0; i<ITERS; i++)
{
    K[i] = new bool[S_IN_WID];

    for (int j=0; j<S_IN_WID; j++)
        K[i][j] = *(PC2[j]);

    // cerr << "Key[" << i << "] is:\n";
    // for (int j=23; j>=0; j--)
    //     cerr << K[i][j];
    // cerr << endl;

    barrelShiftsLeft(CD, (CD_WID/2), 1);
    barrelShiftsLeft(&(CD[CD_WID/2]), (CD_WID/2), 1);
}

/*
* Read in input word
*/
fileRead(input, cin, READ_WRDS);
for (int i=0; i<CMP_WID; i++)
    cmpPrev[i] = *(IP[i]);

cerr << "Data word is:\n";
for (int j=31; j>=0; j--)
    cerr << input[j];
cerr << endl;

// cerr << "Data word after IP:\n";
// for (int j=31; j>=0; j--)
//     cerr << cmpPrev[j];
// cerr << endl;

for (int iteration=0; iteration<ITERS; iteration++)
{

    // cerr << "**** ITERATION " << iteration << " ****\n";

    // cerr << "Input into E --> \n ";
    // for (int j=15; j>=0; j--)
    //     cerr << cmpPrev[j];
    // cerr << endl;

    bitwiseXOR_ptr(K[argv[1][0]=='e'?iteration:(ITERS-iteration-1)],
        E, sIn, S_IN_WID);

    // cerr << "Input into sblock --> \n ";
    // for (int j=23; j>=0; j--)
    //     cerr << sIn[j];
    // cerr << endl;

/*
* Pass through S-function block
* (Note: Things are a bit reversed with the S-lookup

```

```

    * because we hooked up the S-lookup table in hardware
    * in reverse.)
    */
for (int i=0; i<S_NUM; i++)
    for (int j=0; j<S_TAB_OUT_W; j++)
        sOut[i*S_TAB_OUT_W+j] = s[i]
            [ sIn[2+i*3] ]
            [ sIn[1+i*3] + (sIn[0+i*3] * 2) ]
            [S_TAB_OUT_W-j-1];

// cerr << "Output of sblock --> \n ";
// for (int j=15; j>=0; j--)
//     cerr << sOut[j];
//     cerr << endl;

/*
 * Get left half of current computation
 */
for (int i=0; i<(CMP_WID/2); i++)
    cmpCurr[i+(CMP_WID/2)] = cmpPrev[i];

/*
 * Get right half of current computation
 */
bitwiseXOR_ptr(&(cmpPrev[CMP_WID/2]), P, cmpCurr, CMP_WID/2);

/*
 * Copy current computation to previous computation
 */
for (int i=0; i<CMP_WID; i++)
    cmpPrev[i] = cmpCurr[i];
}

cerr << "Word output right before IP^-1 -->\n ";
for (int j=31; j>=0; j--)
    cerr << cmpCurr[j];
cerr << endl;

for (int i=0; i<(CMP_WID/2); i++)
{
    cmpPrev[i] = cmpCurr[i+(CMP_WID/2)];
    cmpPrev[i+(CMP_WID/2)] = cmpCurr[i];
}

cerr << "Word output (binary) -->\n ";
for (int j=31; j>=0; j--)
    cerr << *(IPInv[j]);
cerr << endl;

/*
 * Convert back from binary to base-256 (i.e. ASCII)
 */
for (int i=0; i<READ_WRDS; i++)
    cout << boolToChar(&(IPInv[i*8]));

/*
for (int i=0; i<CMP_WID; i++)
    cout << *(IPInv[i]);
cout << endl;
*/
}

```

```

void permute(bool* input, bool** output, istream &p, int length)
{
    int index;

    for (int i=0; i<length; i++)
    {
        p >> index;
        output[i]=&(input[index-1]);
    }
}

/*
void barrelShiftsLeft(bool* input, int length, int times)
{
    bool* temp = new bool[length];

    for (int i=0; i<times; i++)
        temp[i] = input[i];
    for (int i=times; i<length; i++)
        input[i-times] = input[i];
    for (int i=0; i<times; i++)
        input[length-times + i] = temp[i];
}
*/

void barrelShiftsLeft(bool* input, int length, int times)
{
    bool* temp = new bool[times];

    for (int i=0; i<times; i++)
        temp[i] = input[i+length-times];
    for (int i=length-1; i>=times; i--)
        input[i] = input[i-times];
    for (int i=0; i<times; i++)
        input[i] = temp[i];
}

void bitwiseXOR_ptr(bool* input1, bool** input2, bool* output, int length)
{
    for (int i=0; i<length; i++)
        // output[i] = ((input1[i]) & *(input2[i]));
        output[i] = ((input1[i]) | *(input2[i])) & !((input1[i]) & *(input2[i]));
}

void makeSBlocks(bool**** &s, istream &paramFile)
{
    int out;

    s = new bool***[S_NUM];

    for (int i=0; i<S_NUM; i++)
    {
        s[i] = new bool**[S_TAB_H];
        for (int j=0; j<S_TAB_H; j++)
        {
            s[i][j] = new bool*[S_TAB_W];
            for (int k=0; k<S_TAB_W; k++)
            {
                s[i][j][k] = new bool[S_TAB_OUT_W];
                paramFile >> out;
                for (int l=(S_TAB_OUT_W-1); l>=0; l--)
                {
                    s[i][j][k][l] = out%2;
                    out /= 2;
                }
            }
        }
    }
}

```

```

    }
  }
}

```

```

int fileRead(bool* word, istream &file, int bytes)
{
  char in;

  for (int i=0; i<bytes; i++)
  {
    if (!file.eof())
      in = file.get();
    else
      in = '\0';

    cerr << in;

    for (int j=7; j>=0; j--)
    {
      word[i*8+j] = (in%2);
      in /= 2;
    }

    cerr << endl;

    return file.eof();
  }
}

```

```

char boolToChar(bool** word_ptr)
{
  char c=0;
  int bit = 1;

  for (int i=7; i>=0; i--)
  {
    c += *(word_ptr[i]) * bit;
    bit *= 2;
  }

  return c;
}

```