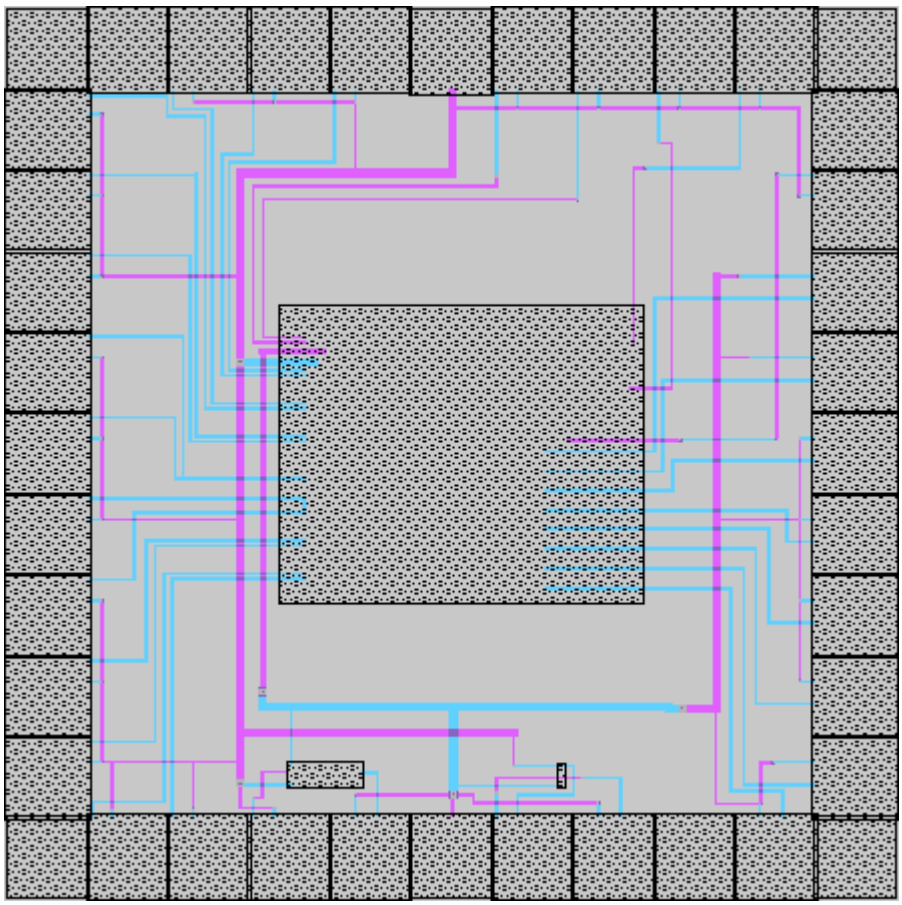
E158 VLSI Final Project: The Carry Look-Ahead Adder

Jason Yelinek
Jeff Miller

April 11, 2001

E158 Intro to VLSI
Prof. Harris



Jason Yelinek

Jeff Miller

April 10, 2001

1. FUNCTIONAL OVERVIEW

This chip is a 32-bit adder that uses carry look-ahead logic to speed up execution. It functions just the same as any other adder: it takes in two 32-bit numbers and returns their sum. Since we are restricted to 40 pins, the chip takes the input in chunks of 8 bits (per number) over four clock cycles and it returns the result over the same four clock cycles. This means that we have 16 pins dedicated to input (8 for each number) and 8 pins dedicated to output.

Ex. So if we wanted to do
0000 0010 1001 1000 0101 1010 0101 0111
+ 0011 0010 10010000 0101 1010 0100 0011

1st cycle :
C0 = 0
A = 01010111
B = 01000011
Y = 10011010
c8 = 0.

2nd cycle
C0 = 0
A = 01011010
B = 01011010
Y = 10110100
c8 = 0

3rd cycle
C0 = 0
A = 10011000
B = 10010000
Y =0011000
c8 = 1

4th cycle
C0 = 1
A = 00000010
B = 00110010
Y = 00110101
c8 = 0

In order to control the different states needed to add the numbers in 8 bit chunks, we used a finite state machine to keep track of the current state. Essentially, this is a counter that counts to 4, and when it reaches 4 it restarts. When in the first state (i.e. the first cycle) it sets the carry in to 0 since we shouldn't have a carry before we start adding. In states 2, 3, and 4, it takes the carry out from the previous cycle and sends it along as the carry in to the next cycle. The FSM also allows for a restart button if need be.

The main reason to use a carry look-ahead adder (CLA) as opposed to a ripple carry adder is increased speed. Rather than wait through all the carries in the ripple chain

to propagate through, the CLA uses a couple basic equations to precompute the carry out for every bit. The idea is that for the n^{th} bit we will give a carry to the $n+1^{th}$ bit based on the equation:

$$C_{i+1} = A_i B_i + (A_i \oplus B_i) C_i \quad (1)$$

This is fundamentally composed of two ideas. We will carry regardless of the carry in if both of our inputs are 1. We will carry if the carry in was 1 and at least one of our inputs is high. The former is called the propogate term, and the latter the generate term. So using equation (1) and assuming that we know C_0 we can calculate C_1, C_2, C_3, and C_4:

$$C_1 = A_0 B_0 + (A_0 \oplus B_0) C_0$$

$$C_2 = A_1 B_1 + (A_1 \oplus B_1)(A_0 B_0 + (A_0 \oplus B_0) C_0)$$

$$C_3 = A_2 B_2 + (A_2 \oplus B_2)(A_1 B_1 + (A_1 \oplus B_1)(A_0 B_0 + (A_0 \oplus B_0) C_0))$$

$$C_4 = A_3 B_3 + (A_3 \oplus B_3)(A_2 B_2 + (A_2 \oplus B_2)(A_1 B_1 + (A_1 \oplus B_1)(A_0 B_0 + (A_0 \oplus B_0) C_0)))$$

Simplifying with a substitution for the two aforementioned terms we get:

$$P_i = A_i B_i$$
$$G_i = A_i \oplus B_i$$
$$C_1 = P_0 + G_0 C_0$$
$$C_2 = P_1 + G_1 P_0 + G_1 G_0 C_0$$
$$C_3 = P_2 + G_2 P_1 + G_2 G_1 P_0 + G_2 G_1 G_0 C_0$$
$$C_4 = P_3 + G_3 P_2 + G_3 G_2 P_1 + G_3 G_2 G_1 P_0 + G_3 G_2 G_1 G_0 C_0$$

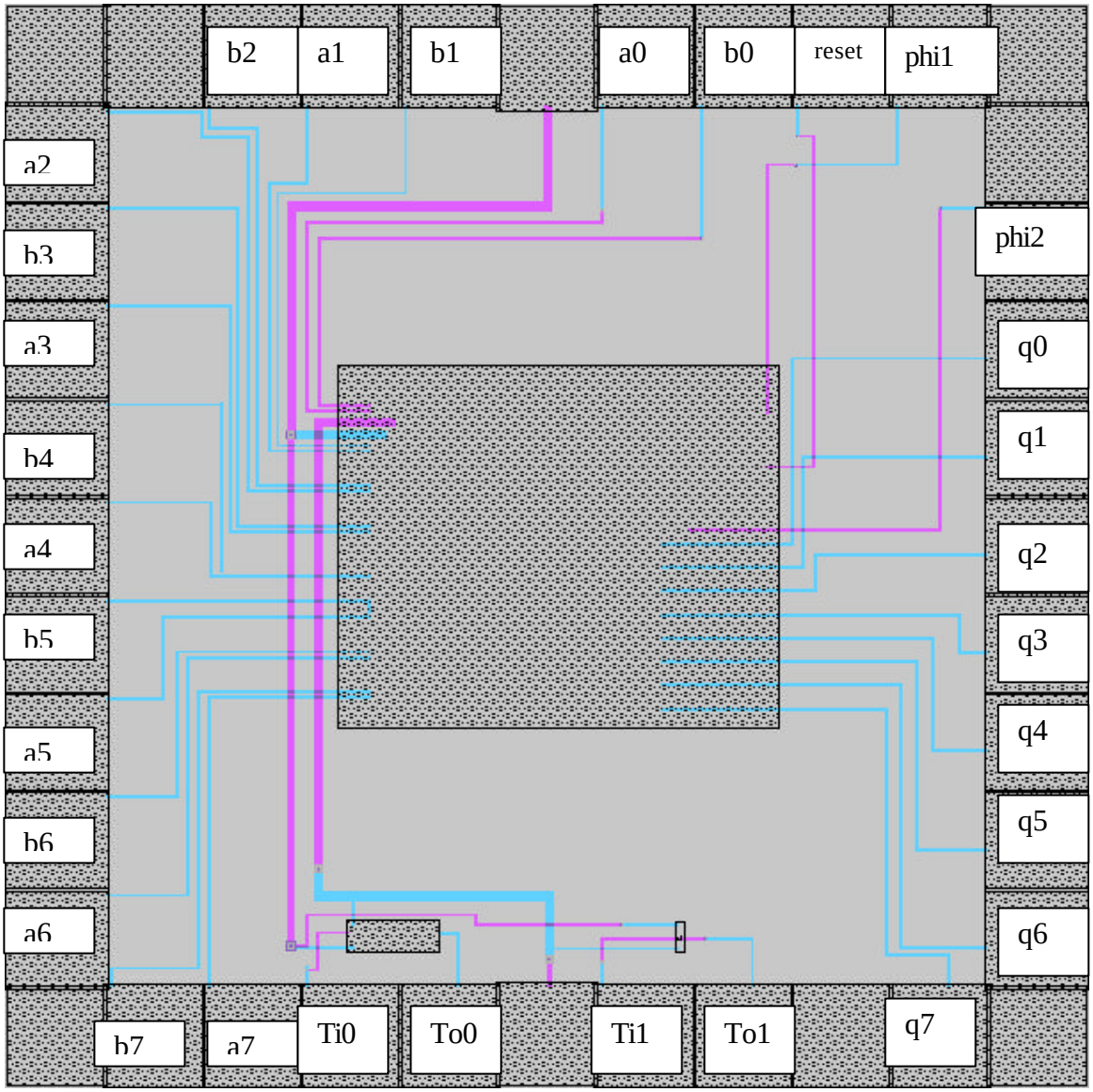
The P_i and G_i terms can be calculated in parallel because we have the inputs available when we take in the 8 bits of a and b. Thus we extracted them out into a preprocessing

unit that computed all 8 P's and all 8 G's. This allowed us to compute each carry bit independently of the others for the most part.

The carry look-ahead logic gets quite large after 4 bits, so in order to keep the chip at a reasonable size, we did only 4 bits of carry look-ahead at a time. To finish off 8 bits in the cycle, we took the carry out of the first four bits and tied it into another carry look-ahead logic block to compute the second 4 bits.

The chip then delays the outputs through a set of 8 latches and ultimately off the chip on 8 output pins.

2. CHIP PINOUT



Inputs:

a[7:0] – the first number to be added

b[7:0] – the second number to be added

reset – should be set for one cycle at startup to make sure the FSM is in the correct state

phi1, phi2 – two phase clocks. The clocks must never be high at the same time.

Outputs

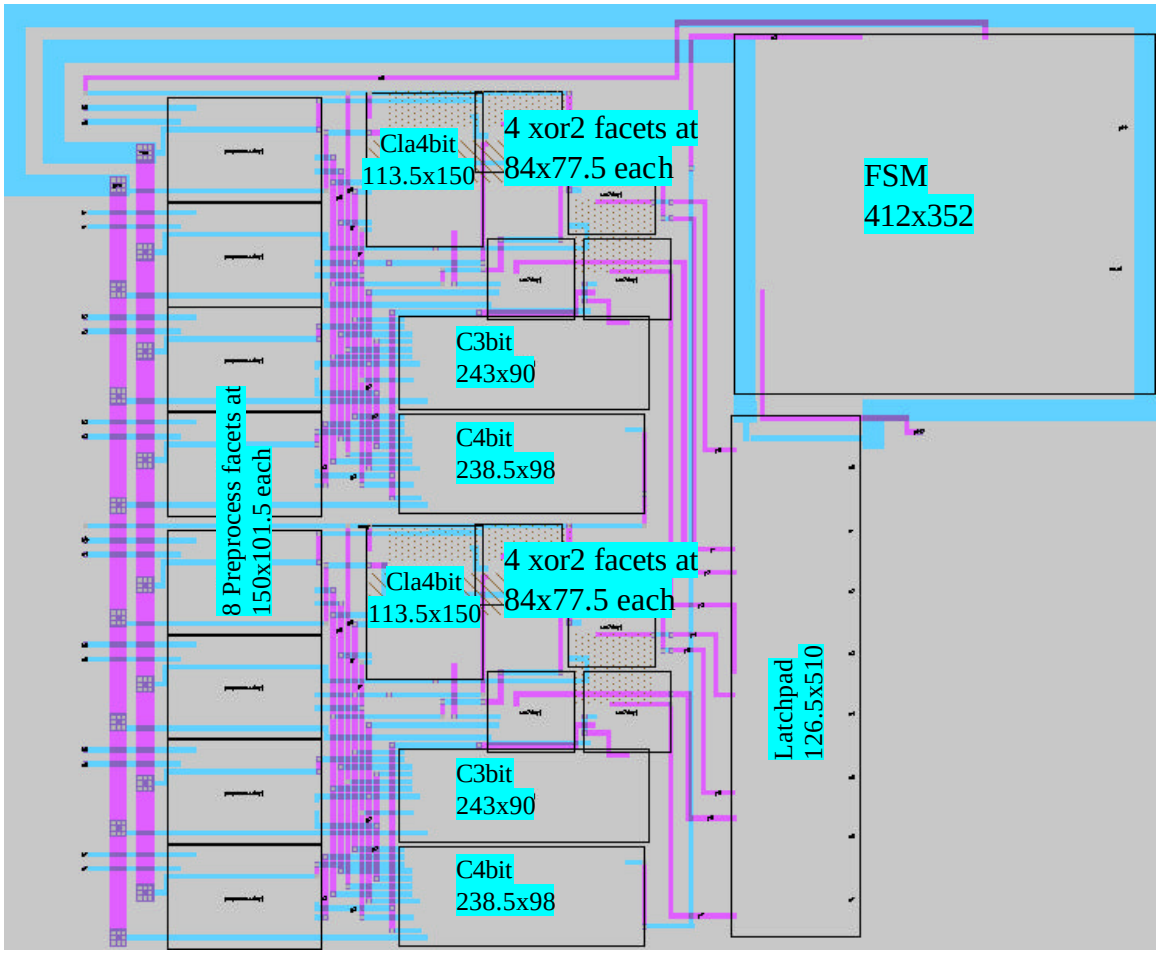
q[7:0] – the sum of the inputs

Test Structures

Ti0, To0 – the input and output of a nand ring oscillator minus the final inverter

Ti1, To1 – the input and output of a simple inverter

3. CHIP FLOORPLAN



4. AREA AND DESIGN TIME DATA

Cell	Dimensions	Area	Transistors	N-Type	P-Type	Area / Transistor	Design Time (hrs)
aoi145{lay}	136.5x79	10784	14	7	7	770	4
aoi32{lay}	52.5x89	4673	10	5	5	467	2
c3bit{lay}	243x90	21870	28	14	14	781	2
c4bit{lay}	238.5x98	23373	28	14	14	835	5
cla4bit{lay}	113.5x150	17025	26	13	13	655	2
fsm{lay}	412x352	145024	114	57	57	1272	6
fullpath4{lay}	577x416	240032	210	105	105	1143	7
fullpath8{lay}	577x839	484103	420	210	210	1153	3
latchpad{lay}	126.5x510	64515	96	48	48	672	3
preprocess{lay}	150x101.5	15225	20	10	10	761	3
toplevel{lay}	**1125x922.5**	**1037813**	**630**	**315**	**315**	**1647**	3
xor2{lay}	84x77.5	6510	12	6	6	543	2
					total design time:		42

5. SIMULATION RESULTS

We used IRSIM to test our chip using the following instructions. We exported the IRSIM deck from Electric by opening our toplevel{lay} facet, going to "Simulation Interface"->"Write IRSIM Deck." Then, we uploaded that file to the Unix server. We wrote a special C++ utility to generate IRSIM batch files, and used it to create 5 batch files. Each was run on our IRSIM deck, and the output was logged. Then, we used a second C++ utility to check the results by parsing the output log and comparing it to expected values. We used this method to check 36,880 test vectors. We had a 100% success rate, so our chip works.

The test vectors were chosen as follows:

Test 1: 9472 test vectors, 100% success

Every value of 'a' from 0 to FF in increments of 7 added to every value of 'b' from 0 to FF.

Test 2: 6656 test vectors, 100% success

Every value of 'a' from 0 to FF00 in increments of A00 added to every value of 'b' from 0 to FF00 in increments of 100.

Test 3: 13312 test vectors, 100% success

Every value of 'a' from 0 to FF0000 in increments of 50000 added to every value of 'b' from 0 to FF0000 in increments of 10000.

Test 4: 7424 test vectors, 100% success

Every value of ‘a’ from 0 to FF000000 in increments of 9000000 added to every value of ‘b’ from 0 to FF000000 in increments of 1000000.

Test 5: 16 test vectors, 100% success

Hand selected group of tests:

Input ‘a’	Input ‘b’	Output ‘y’
0xFFFFFFFF	0xFFFFFFFF	0xFFFFFFFE
0x00000000	0xFFFFFFFF	0xFFFFFFFF
0x0000FFFF	0xFFFFFFFF	0x0000FFFE
0xFFFFFFFF	0x00000001	0x00000000
0xFFFFFFFE	0x00000001	0xFFFFFFFF
0xFF0000FF	0xFFFFFFFF	0xFF0000FE
0xFF00FFFF	0xFFFFFFFF	0xFF00FFFE
0x00000000	0x00000000	0x00000000
0x00000001	0x000000FF	0x00000100
0x000000AF	0xFFFFFFFF	0x000000AE
0x0000FFFF	0x00001111	0x00011110
0xF0000000	0x10000000	0x00000000
0x0000e000	0xFFFF2FFF	0x00000FFF
0xaaaaaaaa	0xaaaaaaaa	0x55555554
0xafde6382	0x00025678	0xAFE0B9FA
0xFFFFFFFF	0x11111111	0x11111110

We feel that this set of vectors cover all of the reasonable situations in which our chip could fail. Since this is a four-cycle processor, it is important to make sure that the 8 bit adder component works correctly for all reasonable 8 bit inputs. We tested about half of all the possible combinations of inputs ‘a’ and ‘b’, and they cover the entire range and weren’t related in any special way. Therefore, we are certain that the adder works. The only other part of the project is the finite state machine that controls the 4-cycle 32-bit add. It has four states, and the carry can be 0 or 1, so there are 8 possible cases. We tested this circuit more than 1000 times for each of the 8 possible cases with a 100%

success rate. Therefore, we are sure the finite state machine works. Since the finite state machine and the adder were thoroughly tested by our selection of test vectors, we can be sure that we would have found a bug if it existed.

6. VERIFICATION RESULTS

Cell Name	Function	Complexity	DRC	ERC	NCC
AOI145	OR(AND5,AND4,1)	4	Pass	Pass	Pass
AOI32	OR(AND3,AND2)	3	Pass	Pass	Pass
c3bit	carry lookahead for bit 3	2	Pass	Pass	Pass
c4bit	carry lookahead for bit 4	3	Pass	Pass	Pass
cla4bit	2 bit carry lookahead	3	Pass	Pass	Pass
FSM	4 state FSM for 32 bit addition	5	Pass	Pass	Pass
fullpath4	4 bit CLA	4	Pass	Pass	Pass
fullpath8	8 bit CLA	2	Pass	Pass	Pass
latchpad	8 latches	3	Pass	Pass	Pass
preprocess	xor2 and and2	3	Pass	Pass	Pass
toplevel	**pinouts, buses, full layout**	**3**	**Pass**	**Pass**	**Pass**
xor2	xor2	2	Pass	Pass	Pass

7. POSTFABRICATION TEST PLAN

First, test the pads to make sure the fab didn't short power and ground or make a faulty padframe. Test the resistance between power and ground to make sure there isn't a short. Next, verify that the test structures work properly. We included a nand ring oscillator minus the output inverter, and a simple inverter. Here we want to make sure we get the correct results out of both. An inverter will invert the input. Our nand ring oscillator will give an inverter input when it starts low. Then on a rising edge it will begin to oscillate, and it only stops after a falling edge. It will start oscillating again on another rising edge and so on.

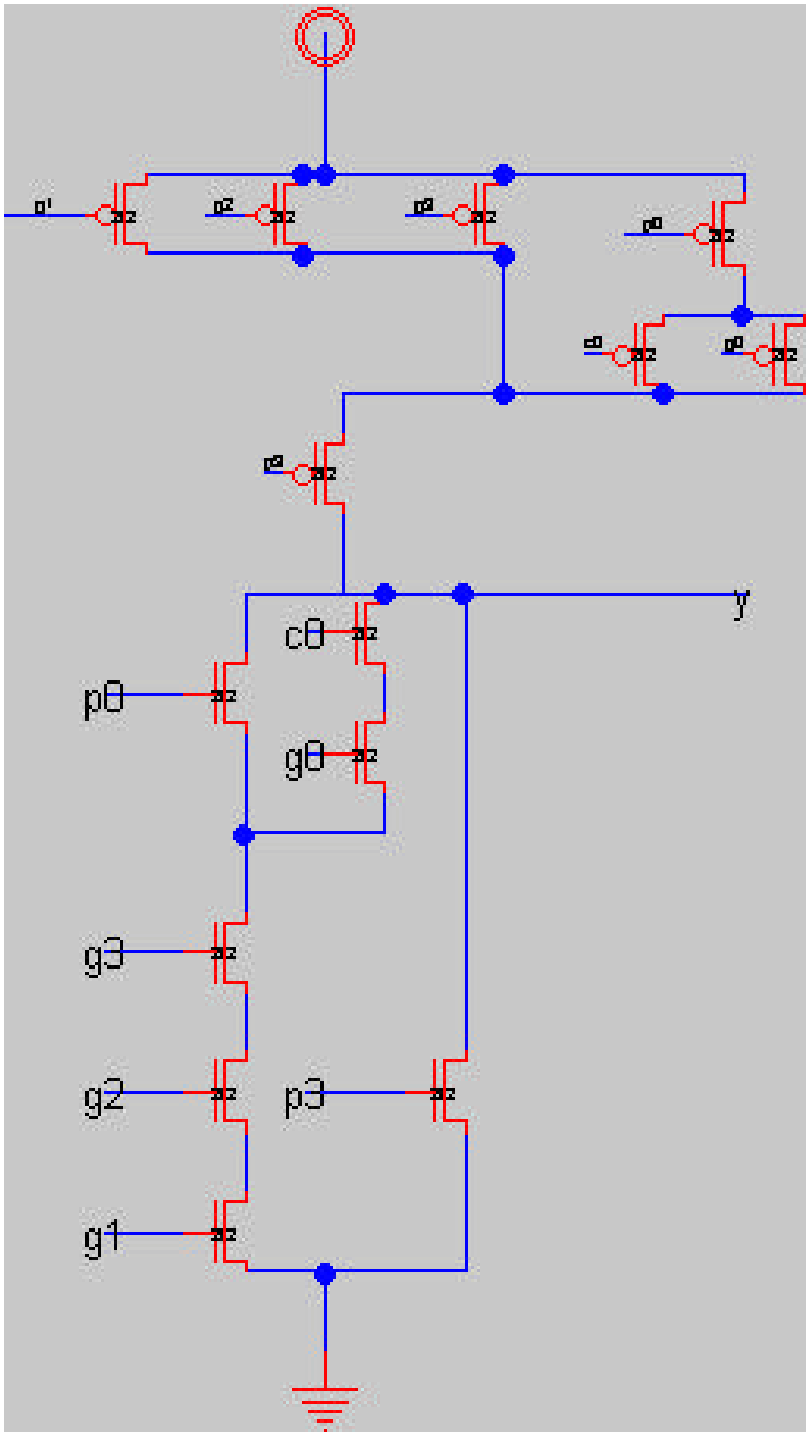
If the test structures work, then the chip came out of the fab at least partially functional. However, to make sure there aren't errors on the chip in places other than the test structures, use the test vectors from Test 5 (see section 5) to verify that the chip works as expected. If the testing process is not automated, then it is practical to test the sixteen vectors plus a few from each of the other test groups.

When testing the chip, the clock must be carefully managed. If phi1 and phi2 are high at the same time, even briefly, the chip will not function. They must be separated by a short time to allow the latches to settle before their inputs are changed. Before any input is given, set the reset bit to high for one cycle. In the next cycle, give the 8 lowest bits of each input on the appropriate pins (least significant bits on a0 and b0 up to most significant a7 and b7). The inputs should be constant for the entire clock cycle, and should be changed during the break between phi2 and phi1. On the next cycle, read and record the output vector (q7-q0), and send the next 8 bits of each operand to the chip. Repeat this until 32 bits of input have been sent, and 32 bits of result have been received.

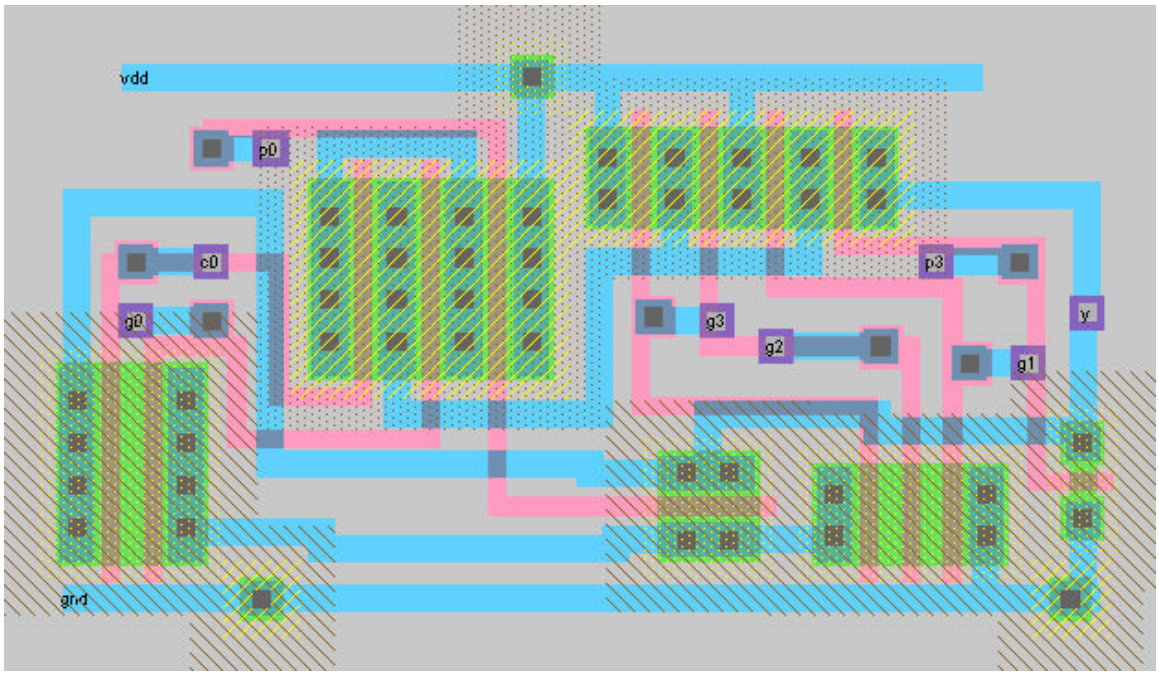
When the chip has finished receiving each number, it is immediately ready for the next 32 bit number, so the chip only needs to be reset once.

8. SCHEMATICS AND LAYOUT

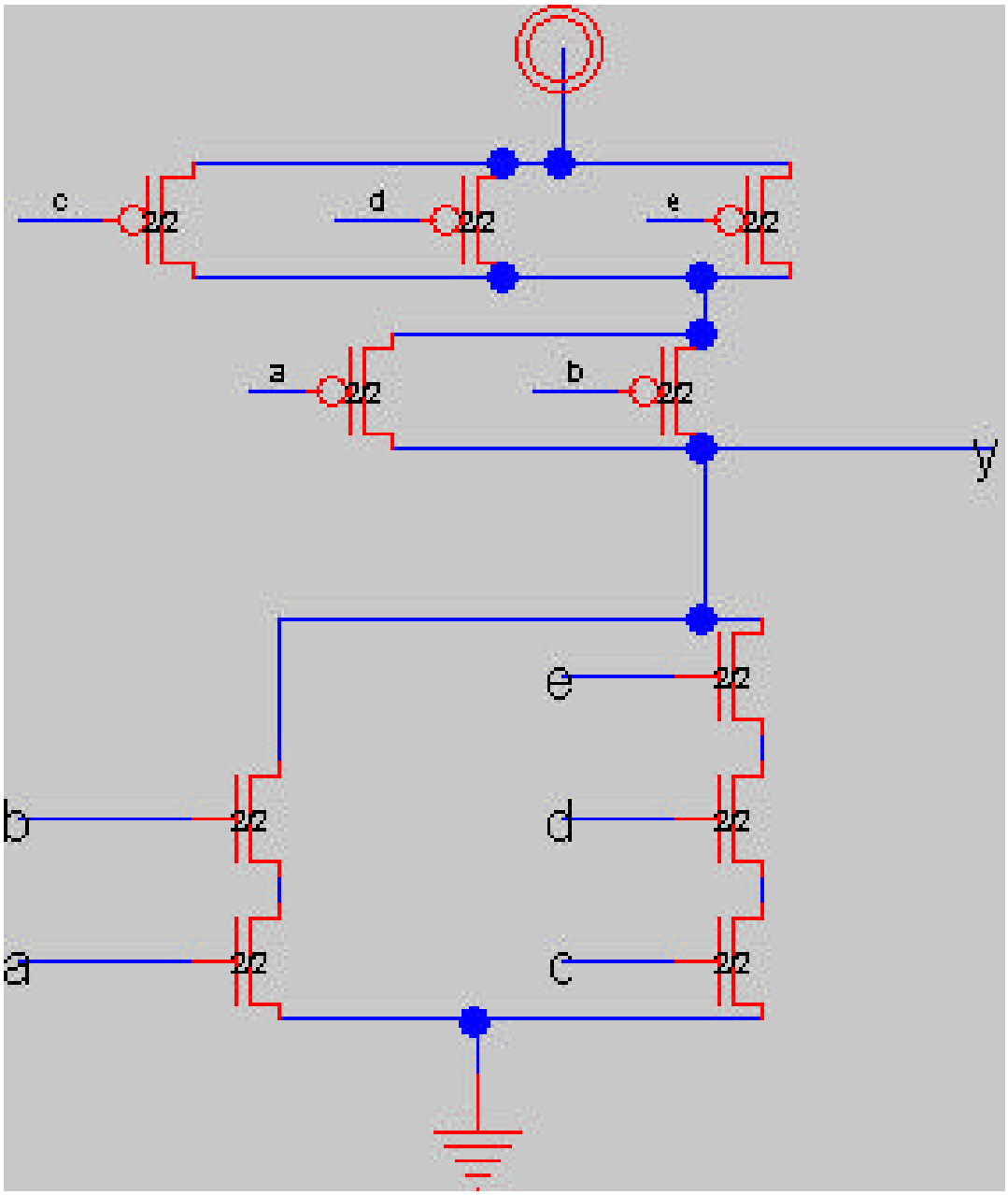
aoi145{sch}



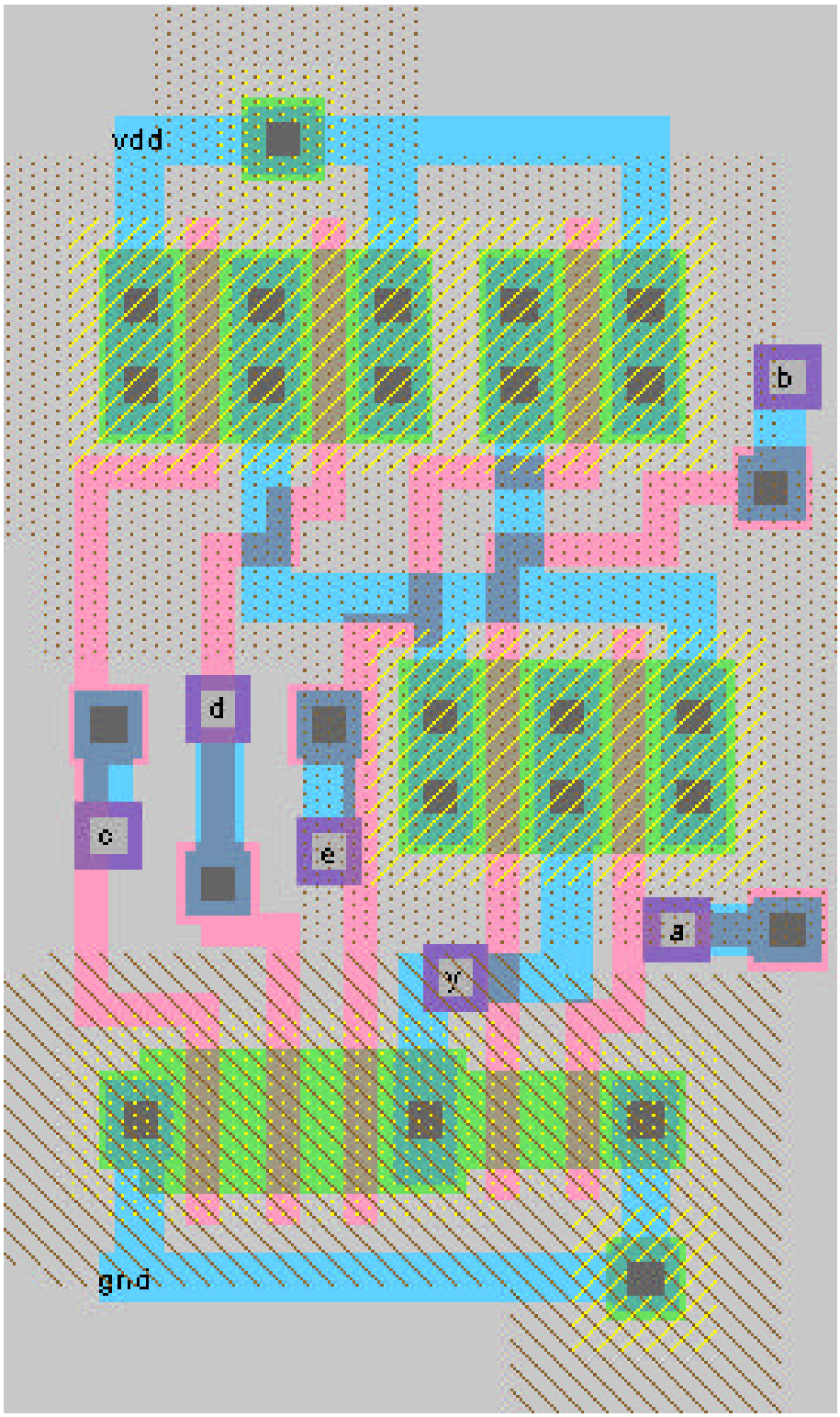
aoi145{lay}



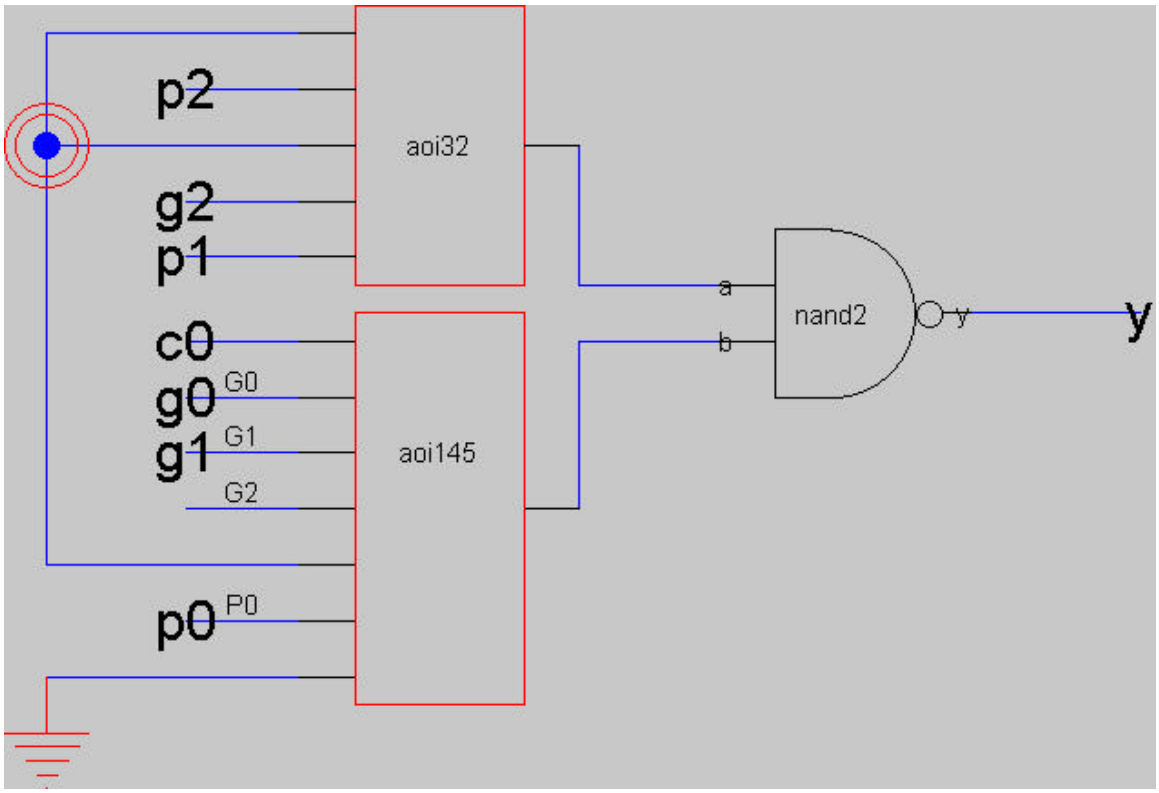
aoi32{sch}



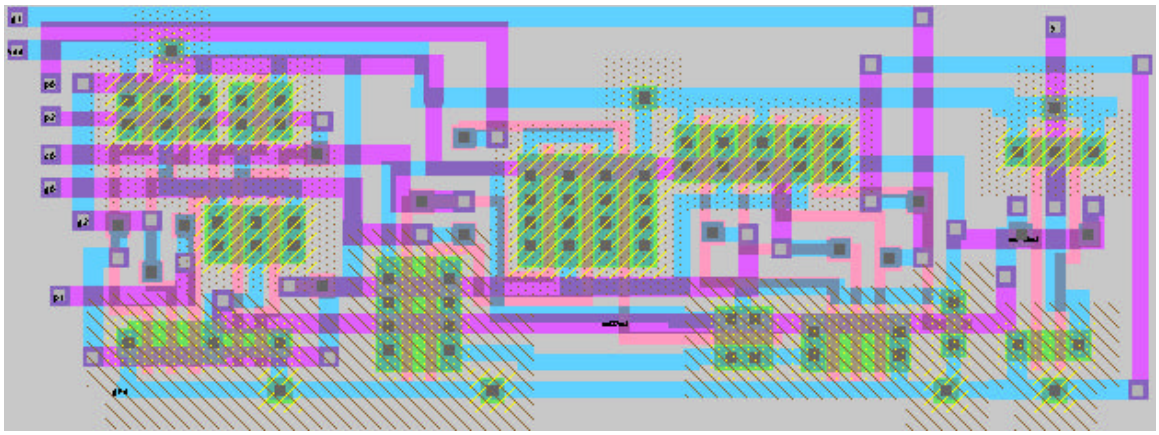
aoi32{lay}



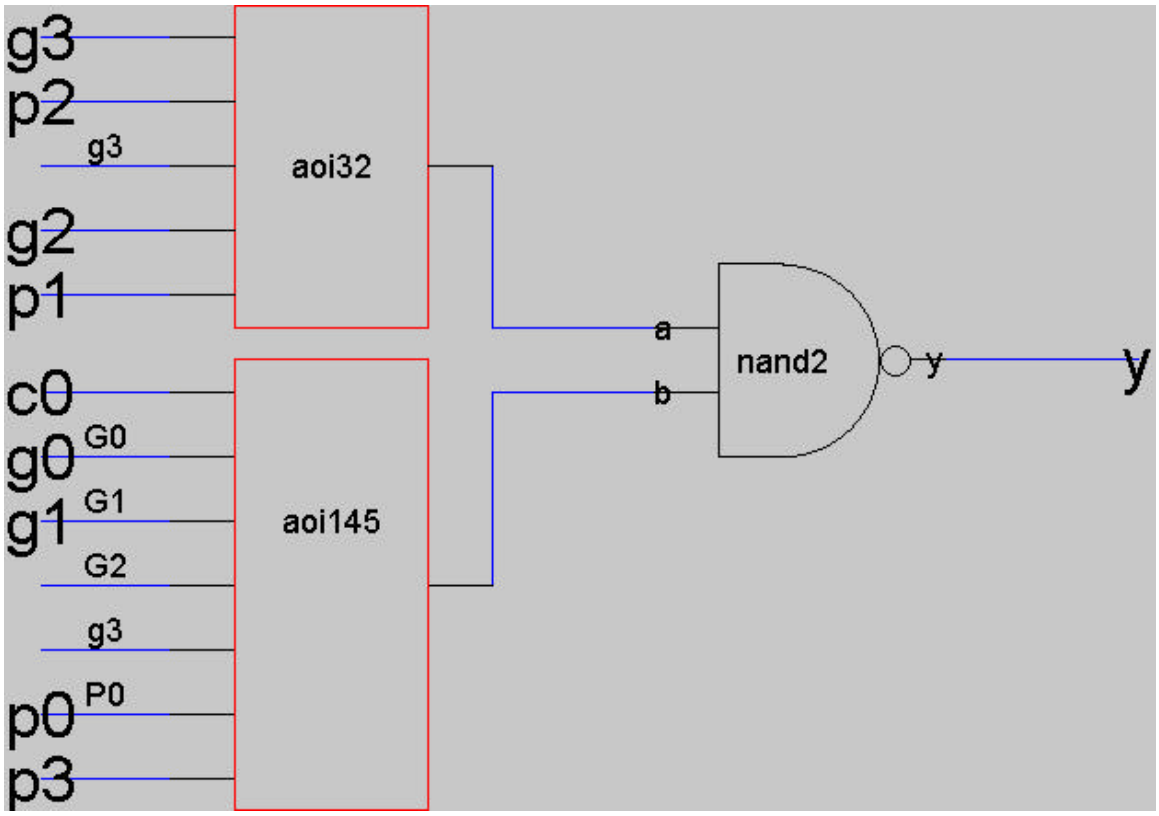
c3bit{sch}



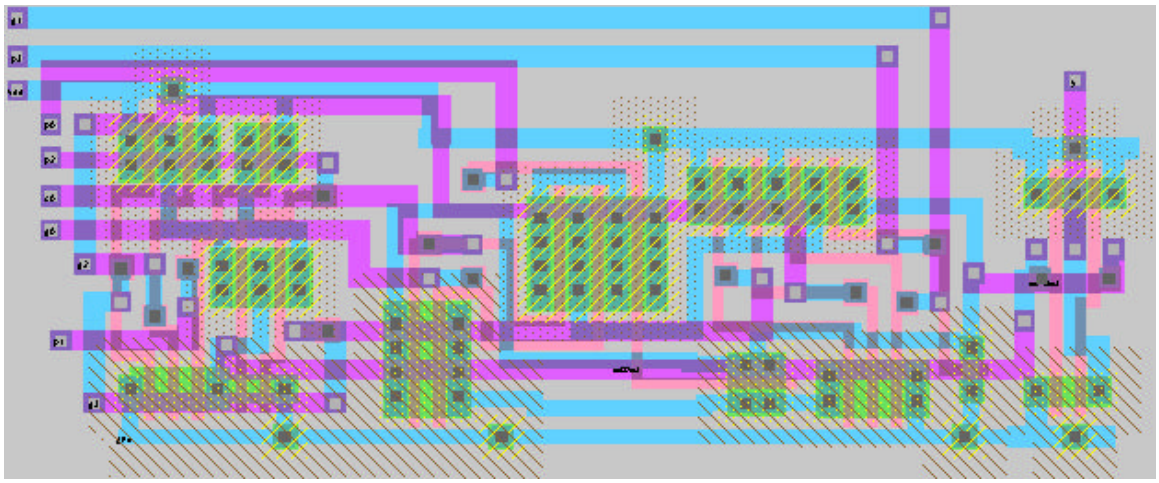
c3bit{lay}



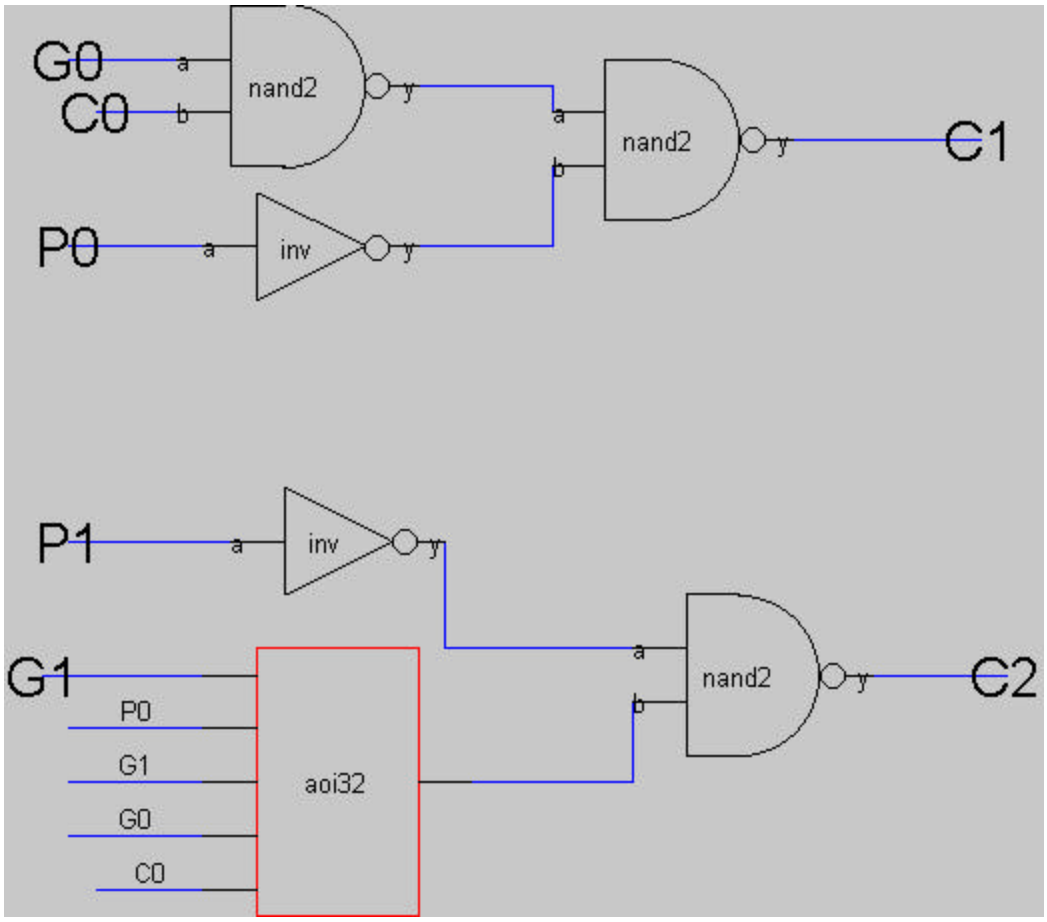
c4bit{sch}


g3
p2
g3
g2
p1
aoi32
c0
g0
G0
g1
G1
G2
g3
p0
P0
p3
aoi145
a
b
nand2
y
y

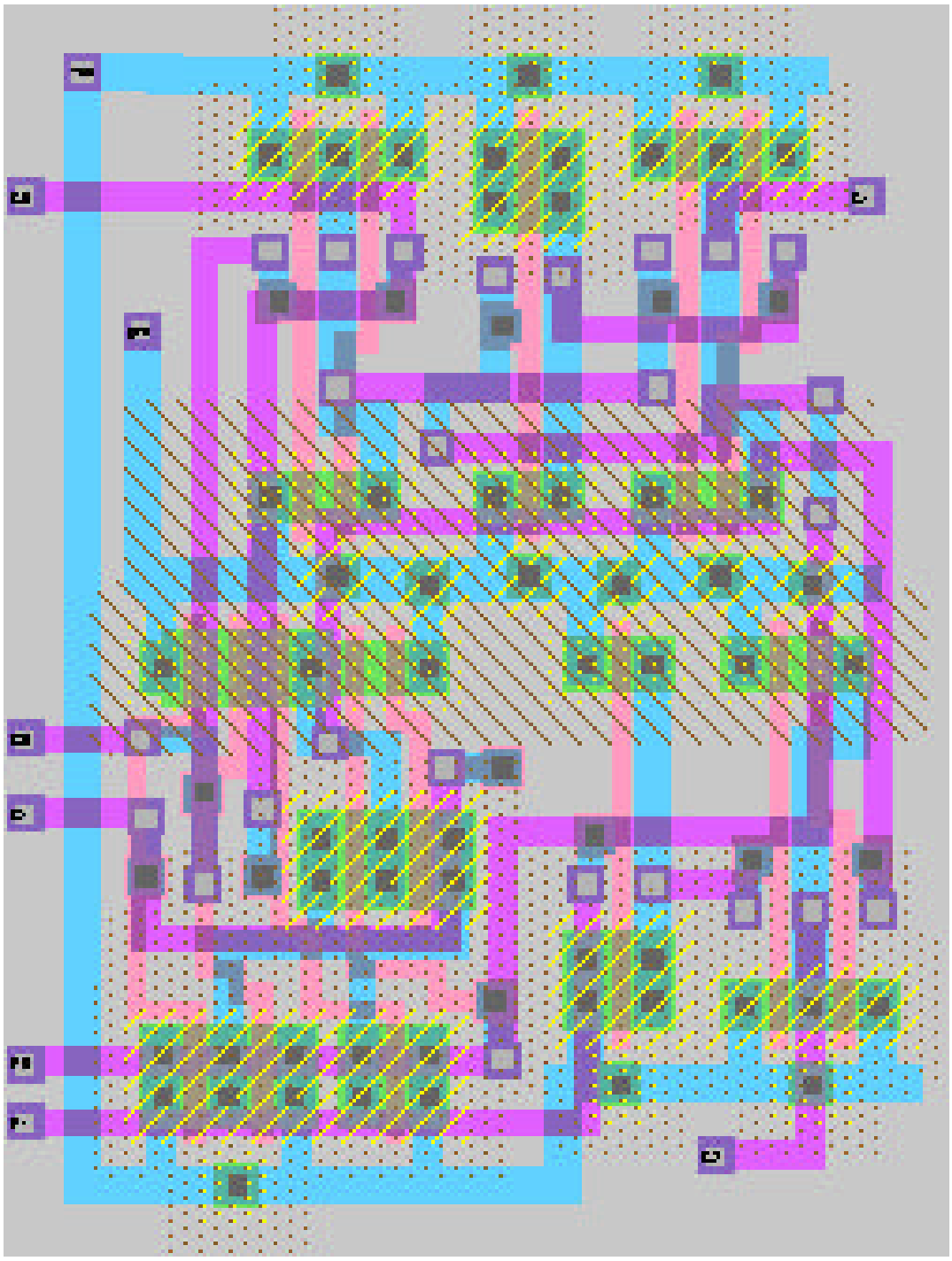
c4bit{lay}



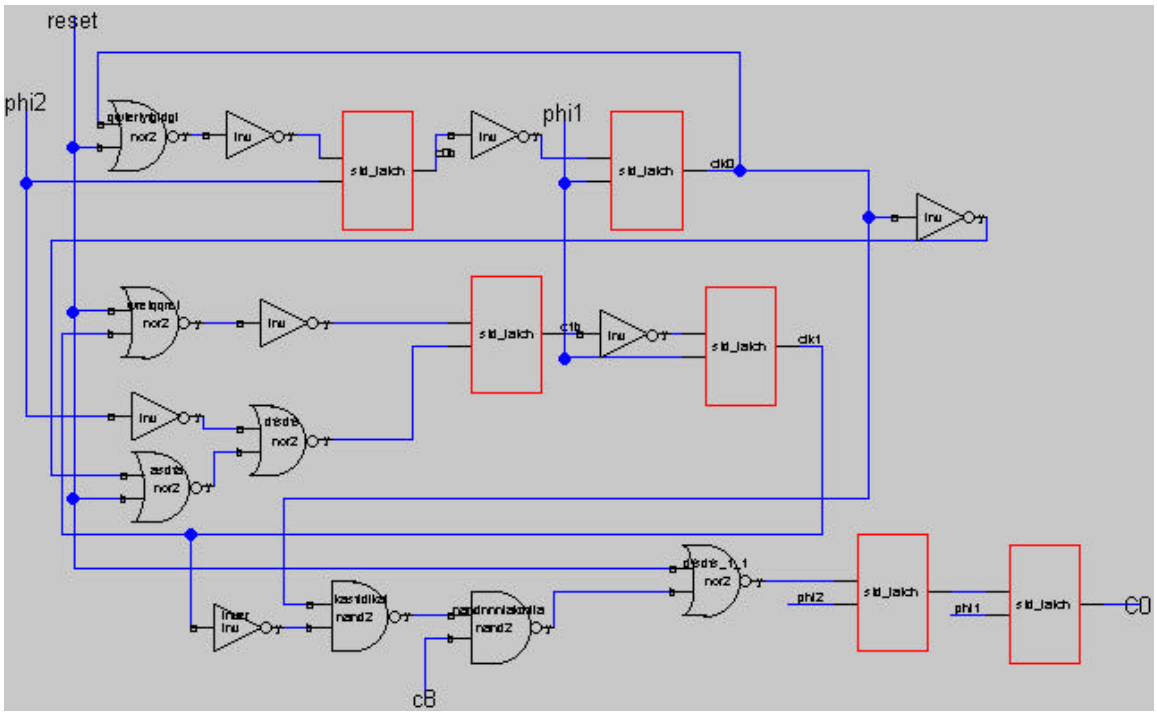
cla4bit{sch}


G0
C0
a
b
nand2
y
a
b
nand2
y
C1
P0
a
inv
y
P1
a
inv
y
G1
P0
G1
G0
C0
aoi32
a
b
nand2
y
C2

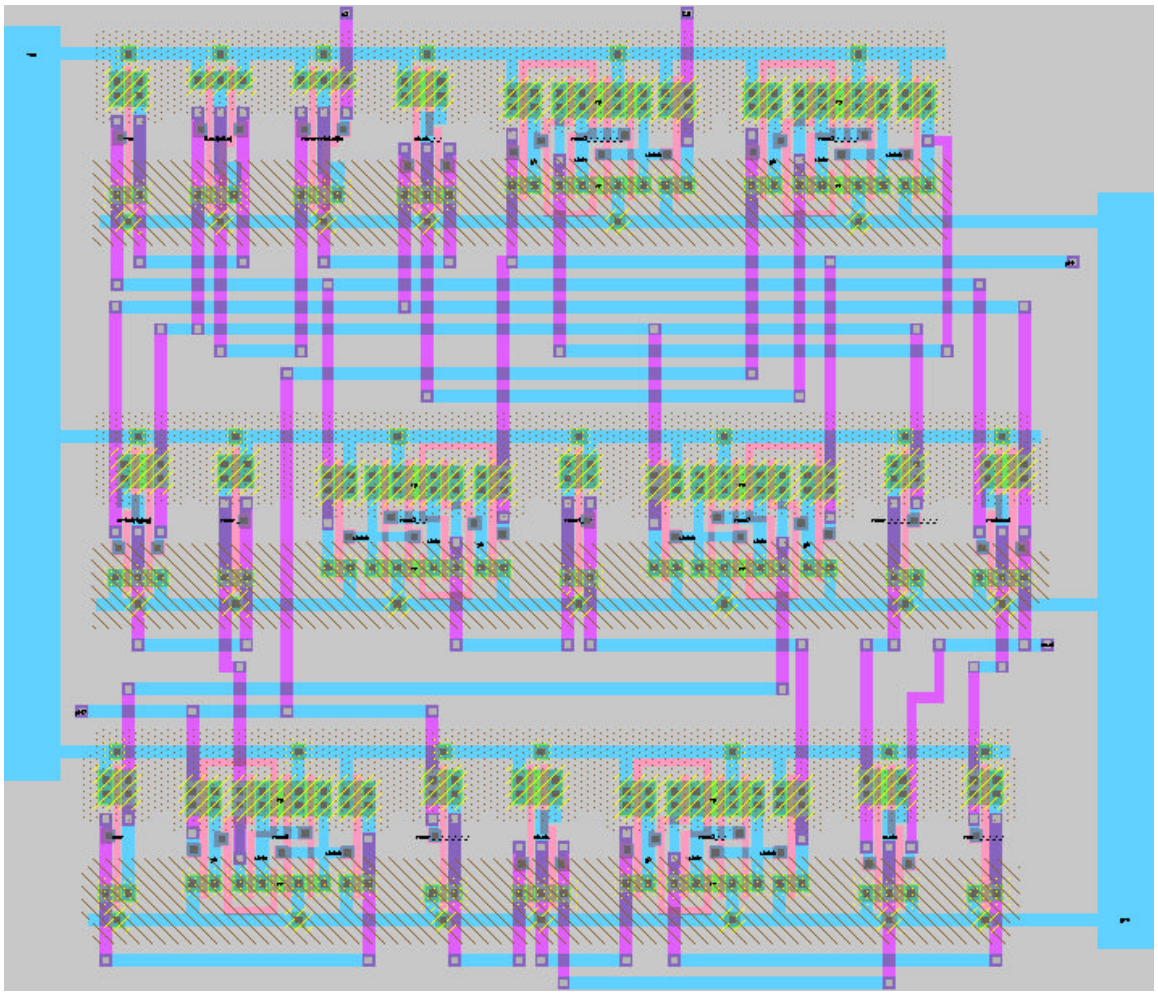
cla4bit{lay}



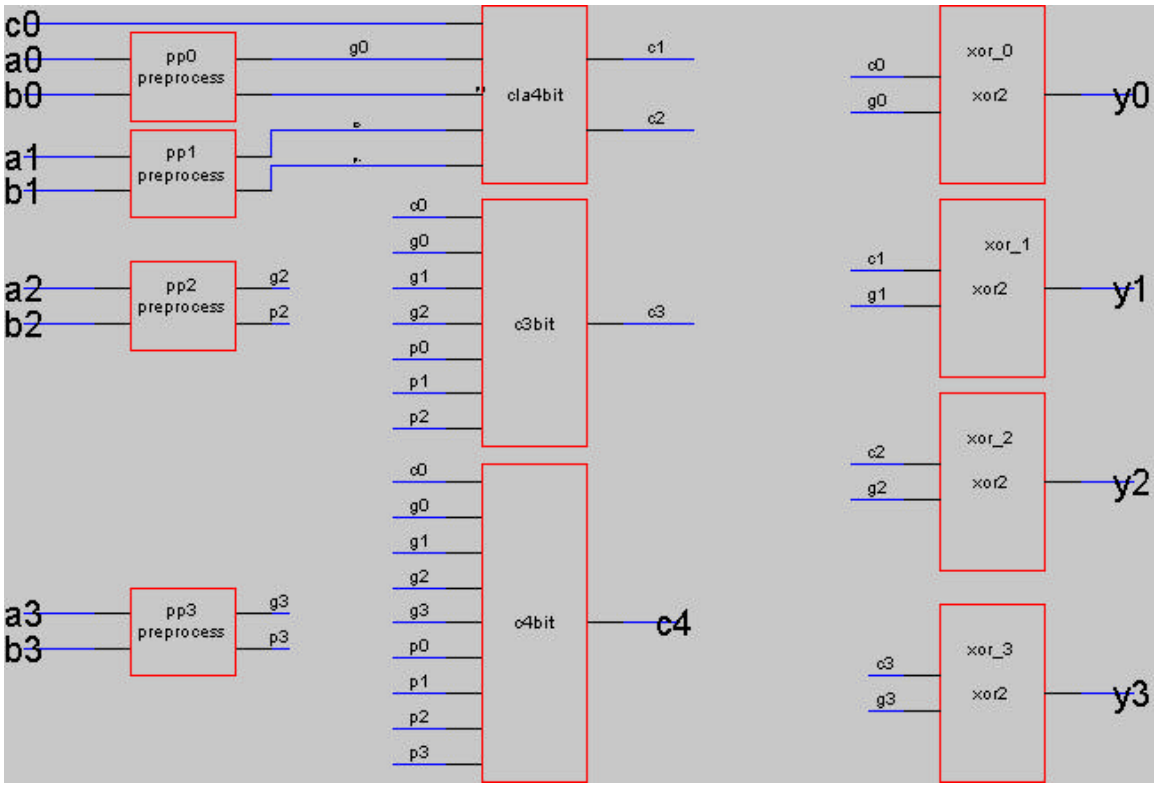
fsm{sch}


reset
phi2
phi1
nor2
inv
std_jakch
clk0
clk1
phi2
phi1
nand2
c8
c0

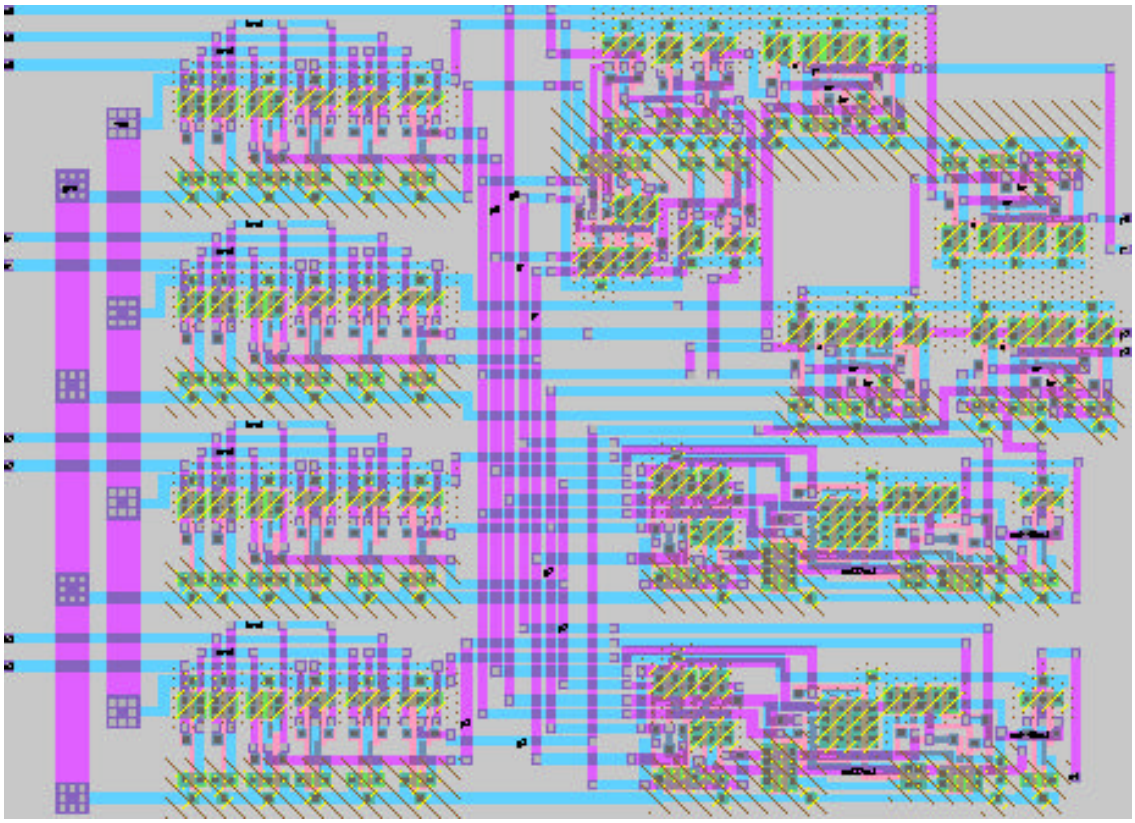
fsm{lay}



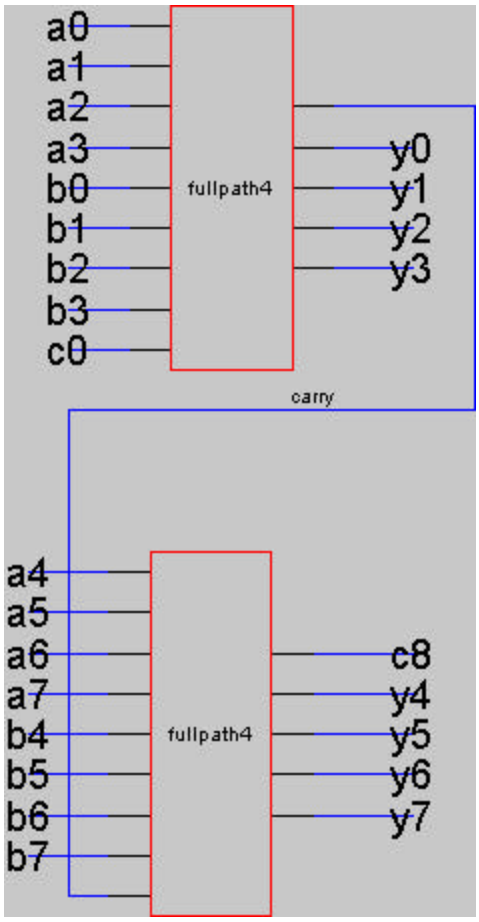
fullpath4{sch}



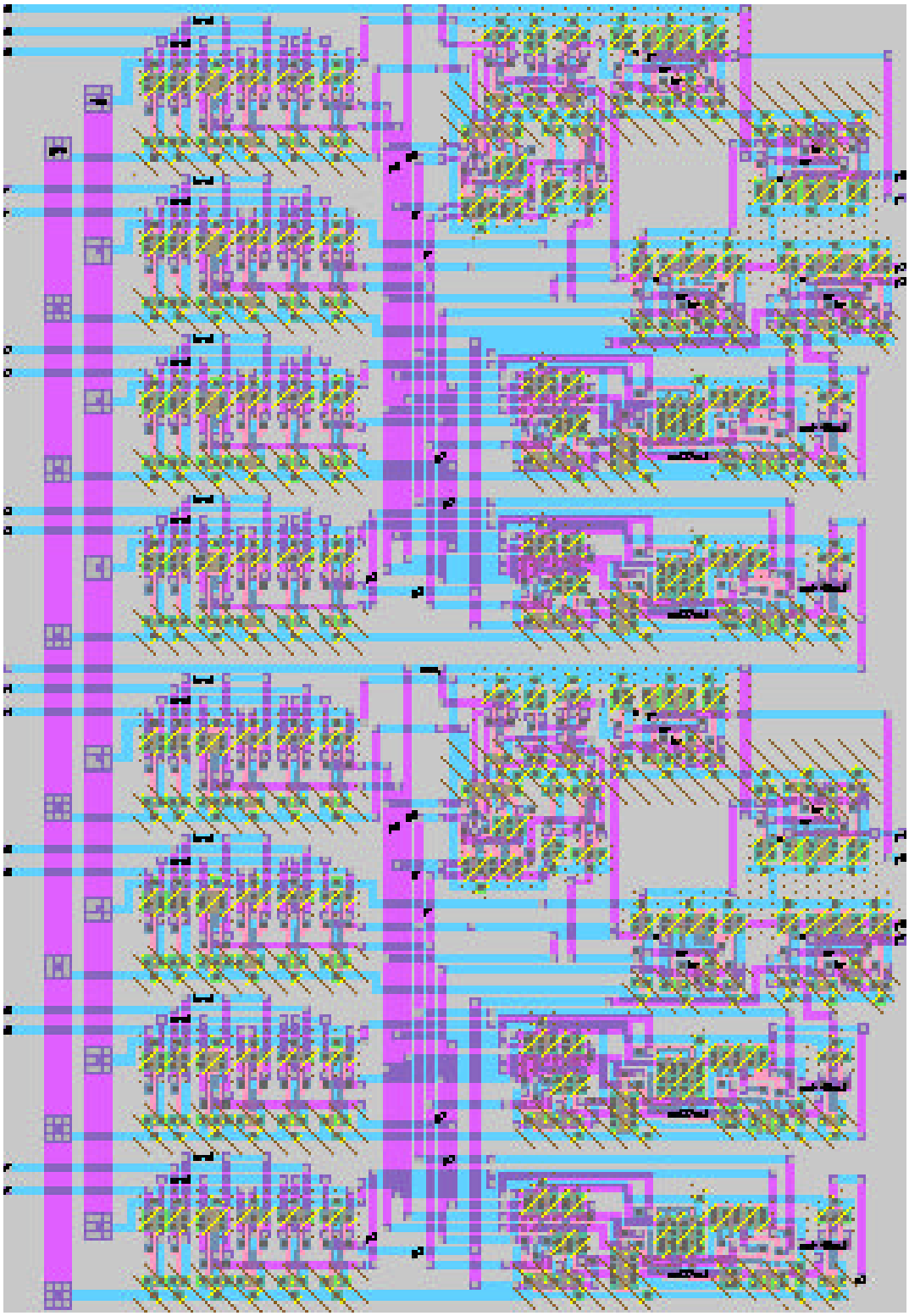
fullpath4{lay}



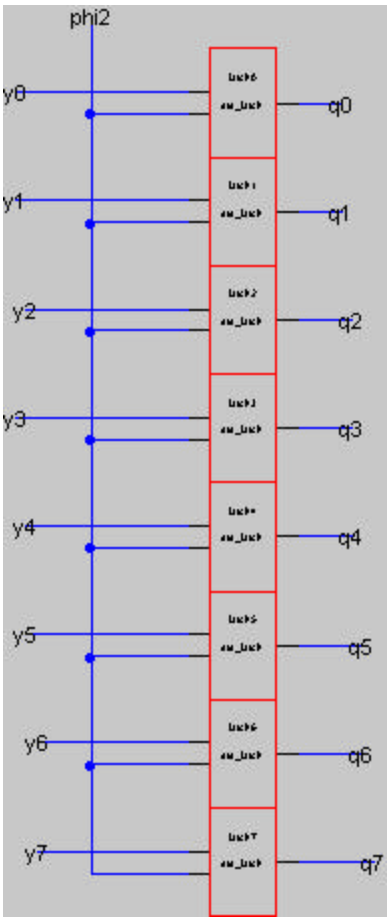
fullpath8{sch}



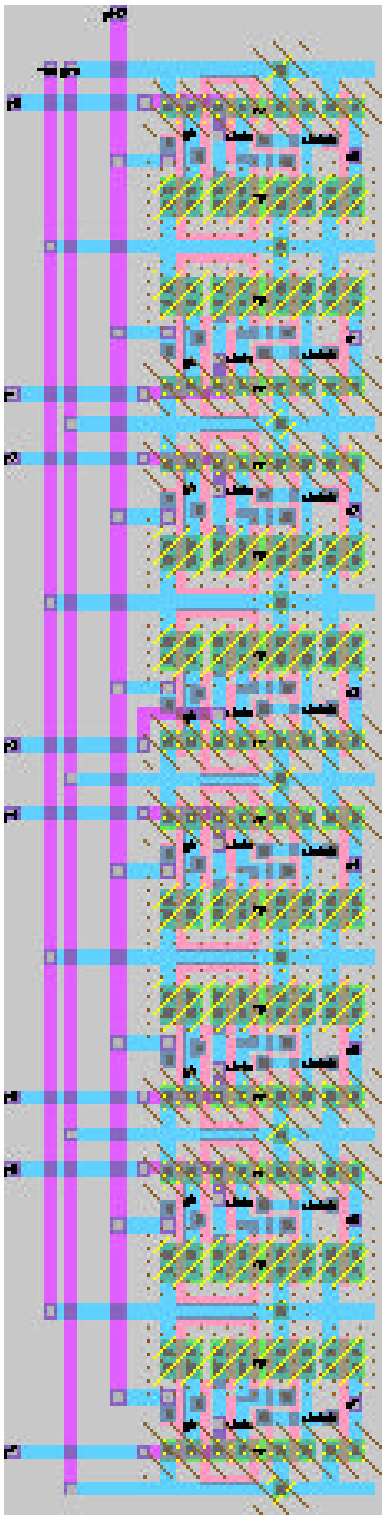
fullpath8{lay}



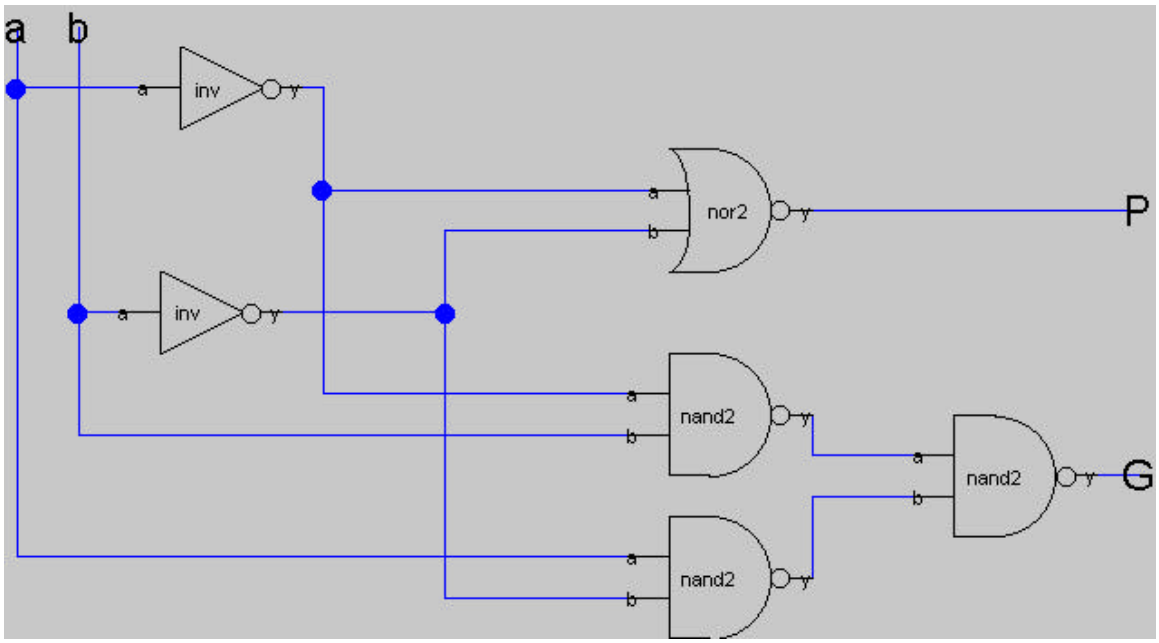
latchpad{sch}



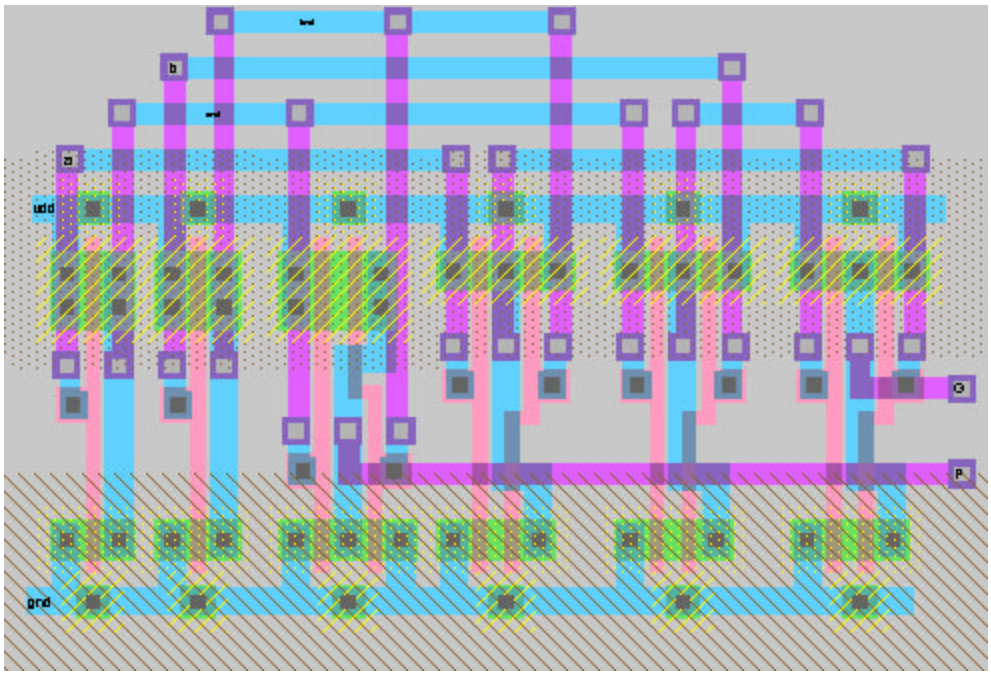
latchpad{lay}



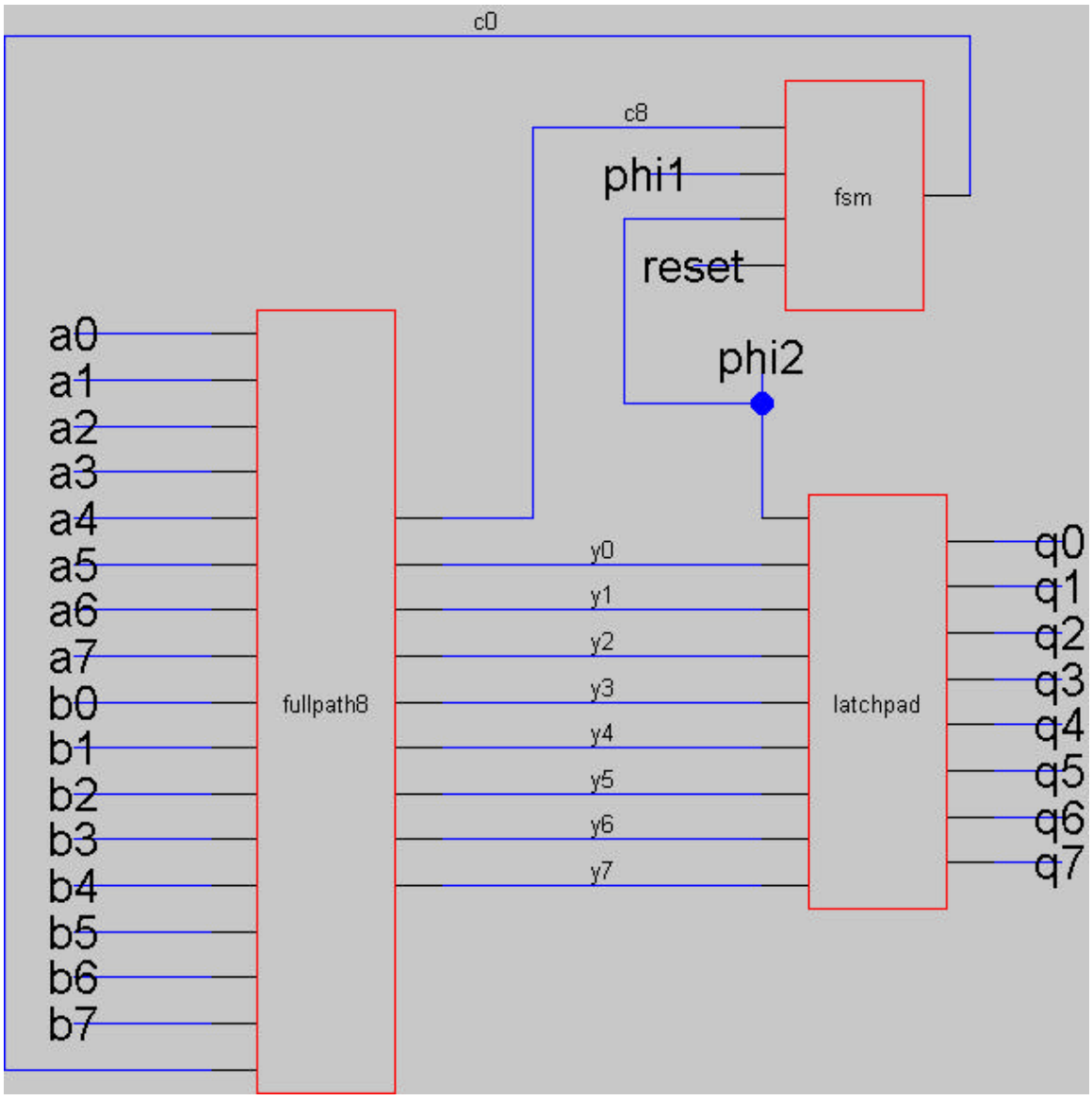
preprocess{sch}



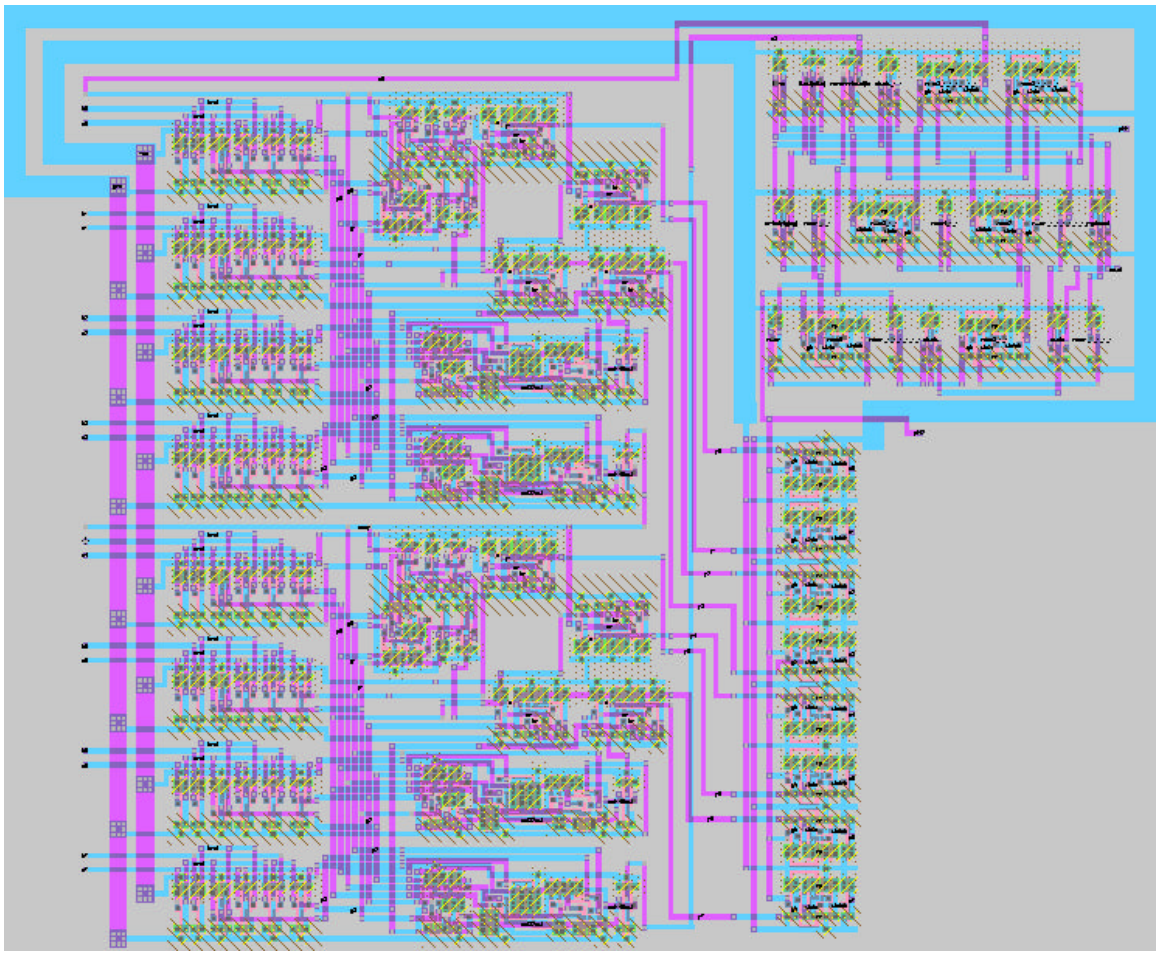
preprocess{lay}



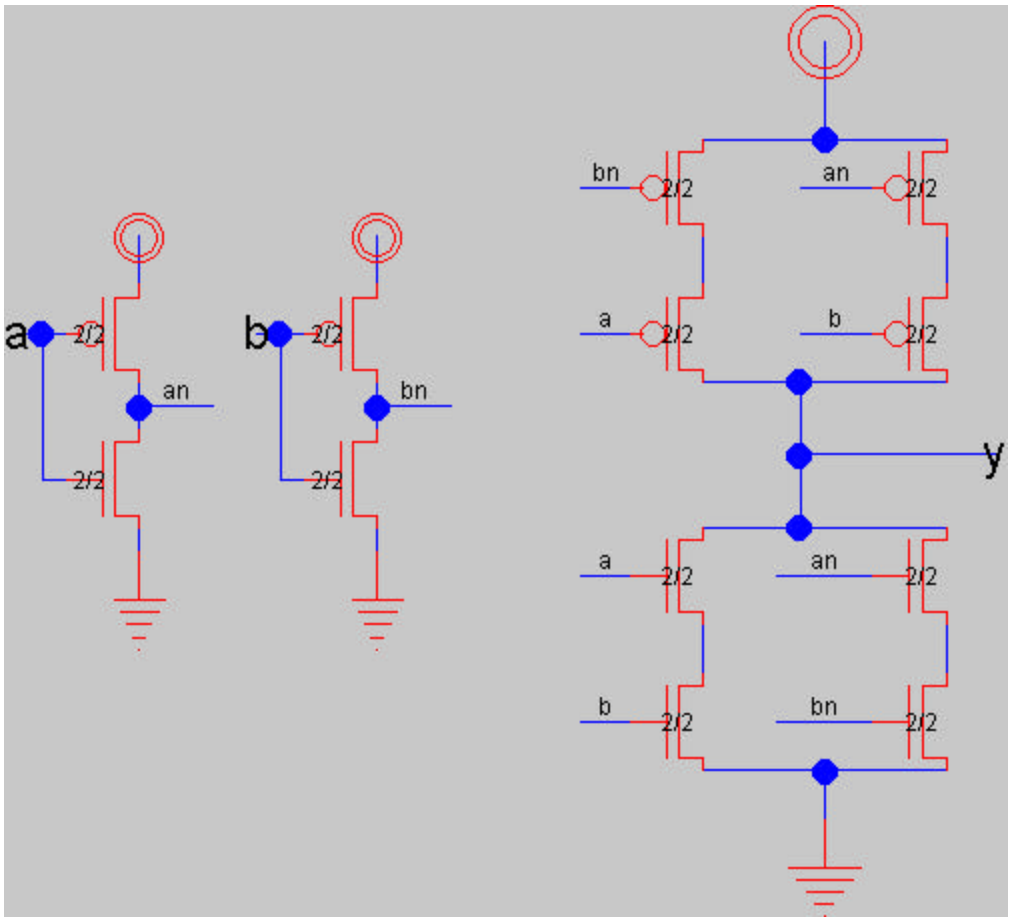
toplevel{sch}



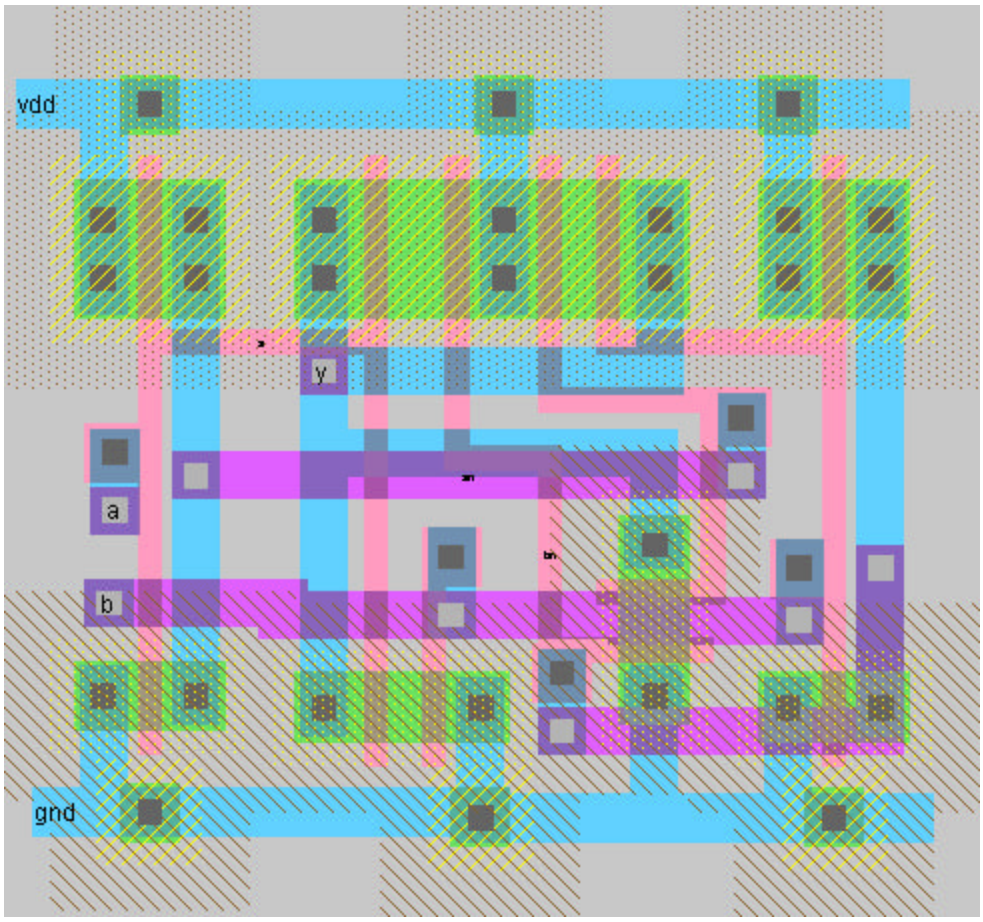
toplevel{lay}



xor2{sch}



xor2{lay}



top{lay}