FERMILAB-TM-2110

BTEV/C0

FPIX CORE ARCHITECTURE AND THE PREFPIX2 CHIP

ARCHITECTURE AND SIMULATION

JIM HOFF AND ABDER MEKKAOUI

TABLE OF CONTENTS

FPIX CORE ARCHITECTURE AND THE PREFPIX2 CHIP

ARCHITECTURE AND SIMULATIONS

1 INTRODUCTION

preFPIX2 is a developmental step in the evolution of the final BTeV pixel architecture. It is a smaller version of a fully functional FPIX Core. It is a necessary step between FPIX1 and FPIX2 mostly for monetary reasons. Both FPIX1 and FPIX2 must be bump bonded to 18x160 arrays of ATLAS pixel detectors. Therefore, since each pixel is 50 µm by 400 µm, each FPIX chip cannot possibly be smaller than 7.2 mm by 8 mm. Since such a large chip is expensive, the collaborators are only being conservative by producing smaller versions of full FPIX chips when testing different ideas.

Most importantly, preFPIX2 continues a progression towards smaller and smaller device geometries. FPIX0 was developed using Hewlett-Packard's 0.8µm CMOS process. FPIX1 was developed using Hewlett-Packard's 0.5µm CMOS process. FPIX2 will be developed in IBM's 0.25 µm process or TSMC's 0.25 µm process or in both processes. The major objectives of the development of preFPIX2 are to test our ability to successfully develop deep submicron IC chips and to test the capabilities of both the TSMC and IBM processes. The goal of reducing the process geometry is to take advantage of the higher and higher radiation tolerances they provide [1]. To further the goal of high radiation tolerance, FPIX2 will be developed using radiation tolerant design techniques, in particular, enclosed transistors [2]. preFPIX2 is the first functional chip developed at Fermilab to use such techniques.

Finally, preFPIX2 has been developed to test a number of algorithmic modifications to the original read-out control developed in FPIX1. The FPIX1 readout architecture, also called the Command Driven Architecture, has been highly successful in all tests. However, it was decided that it could be improved substantially and simplified dramatically without changing it fundamentally.

The purpose of this paper is to describe in detail this new version of the Command Driven Architecture and to describe a concept that is somewhat new to FPIX – the division of labor between the Core and the Periphery.

2 A SHORT NOTE ON TERMINOLOGY

It can be somewhat confusing talking about Pixels because so many words mean the same thing and because, very often, the same word is used to mean many things. The following is a list of words and their meanings as they will be used in this paper.

1. **Pixel Detector** - The actual semiconductor device through which a high-energy particle will pass and leave an electrical trace.

2. **Pixel Cell** - The VLSI circuit that deals with the electrical signals of the Pixel Detector and, among other things, converts those signals into usable digital data (i.e. converts hit data to recorded data).

3. **Pixel Cell Array** - The array of Pixel Cells viewed as a single entity

4. **Pixel Detector Array** - The array of Pixel Detectors viewed as a single entity

5. **Hit Data** - The electrical signal of a Pixel Detector or the electrical signals Pixel Detector Array. Hit data can be considered the current state of the Pixel Detector Array. Hit data changes with each beam cross over.

6. **Recorded Data** - The result of the conversion of Hit Data into usable digital data that is in part stored in the Pixel Cell Array and in part stored in the End-of-column Logic. Recorded data remains stored until it is read out.

7. **Output Data** - Also called Core Output Data. It is the data stream output from the Core.

Most of the confusion occurs with the overuse of the word "data". This is why it is important to be explicit with the three types of data in the FPIX Core. Hit data is produced by the pixel detector array. It is an ever-changing snap shop of the interaction chamber. Hit data is converted into recorded data by the pixel cell array. Recorded data is held indefinitely. It is converted to output data by the combined efforts of the Pixel Cell, End-of-column Logic and the Core Logic. As recorded data is converted into output data it is erased from the pixel cell array. The pixel detector array *supplies* or *has* hit data. The pixel cell array *has* recorded data. The Core *streams* output data. If the FPIX Core has done its job properly, the original hit data can be reconstructed from the output data. In fact, that is the ultimate purpose of the FPIX Core.

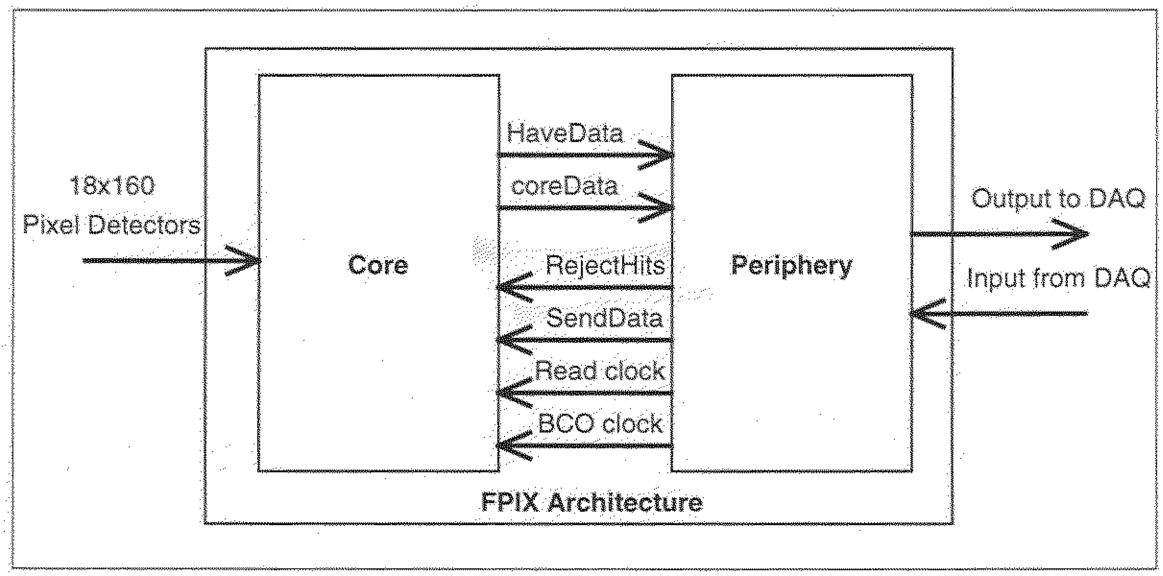


Figure 1: Core and Periphery Cells

Starting with preFPIX2, the FPIX architecture should be viewed as two black boxes: the Core and the Periphery (See Figure 1). The Core is connected to the pixel detector array and to the Periphery. The Periphery is connected to the Core and to the DAQ system. The job of the Core is to accept hit data from the pixel detector array, convert it into the Core output data stream and present that stream to the Periphery in a consistent fashion. The job of the Periphery is to take that predictable output data stream and convert it into a form acceptable by whatever DAQ system is connected to the chip. The purpose of this division of labor is to allow Core and Periphery architectures to develop with some independence. It ensures that changes to the DAQ system do not influence the way pixel detector arrays are handled and it ensures that changes in the way pixel detector arrays are handled do not influence the way the DAQ system gathers data. This allows the problem of pixel readout to be optimized in isolation from the myriad problems of output data organization. Some users might want triggered operation whereas others might want non-triggered operation. Some might want time-ordered output whereas others might not care. Some might want to row- or column-ordered output, etc. All ordering and/or triggering as well as any requirements of chip-to-DAQ communication should be handled in the Periphery. The Core would remain untouched by these requirements.

To accomplish this division of labor, a consistent interface between the Core and Periphery has been defined (See Figure 1). The Periphery provides the Core with the Beam Cross Over Clock (BCO clock) and the Read Clock which establish the timing for hit data and output data, respectively. In order to reduce sources of error, the design of the Core should be such that these two clocks are not *required* to be related in either frequency or synchronicity. The Periphery also provides the Core with two signals, SendData and RejectHits. RejectHits controls the conversion of hit data into recorded data that is performed by the pixel cell array. If it is active, the conversion is suspended and new hit data is ignored. If it is inactive, the conversion is enabled, and new hit data is recorded. SendData controls the conversion of recorded data into output put data that is performed by the End-of-column Logic and the Core Logic. If it is active the conversion is enabled and recorded data is output from the Core. If it in inactive, the conversion is suspended and recorded data is held in the pixel cell array. The two controls are independent as shown in Table 1.

Table 1: Meaning of Send Data and Reject Hits

State	Meaning	Action
00	Send Data inactive Reject Hits inactive	No recorded data is sent. Core will accept new hit data. After an infinite amount of time, the entire pixel cell array will be full of recorded data.
01	Send Data inactive Reject Hits active	No recorded data is sent. No new hit data is accepted. The state of the Core will not change one or both of these signals is changed.
10	Send Data active Reject Hits inactive	Recorded data is sent. Core will accept new hit data. Normal operational mode.
11	Send Data active Reject Hits active	Recorded data is sent. Core will not accept new hit data. After a short time, the Core will be empty of recorded data.

The Core provides the Periphery with the HaveData signal. When HaveData is active, the Core is streaming output data. When inactive, the Core is NOT streaming output data. It is possible for HaveData to be inactive in spite of the fact that there is recorded data somewhere in the pixel cell array. This would depend on timing. HaveData is an indicator that the Core is streaming output data. It is not an indicator of the presence of recorded data in the pixel cell array or of the presence of hit data in the pixel detector array. coreData is the output data of the Core. In the interest of speed, no compromises were made on the width of the output data word. Each word contains the full BCO number (time stamp) of the event, the full column number, the full row address and the full magnitude of the hit. Any concatenation or separation of output data words should be handled in the Periphery.

The amount of output data manipulation and organization that occurs in the Periphery can vary over a very wide spectrum. It has, in fact, no logical limit. Its only practical limitation is that, typically, pixel projects like to limit the amount of chip area not dedicated to the pixel cell array (the detection area). The Periphery could contain FIFOs, serializers, Content-Addressable Memory, etc. It could even be empty provided that the DAQ system can be structured to provide the signals necessary to run the Core. This last fact is demonstrated by preFPIX2 itself. preFPIX2 is an FPIX Core with a small (18x32) pixel cell array and no Periphery.

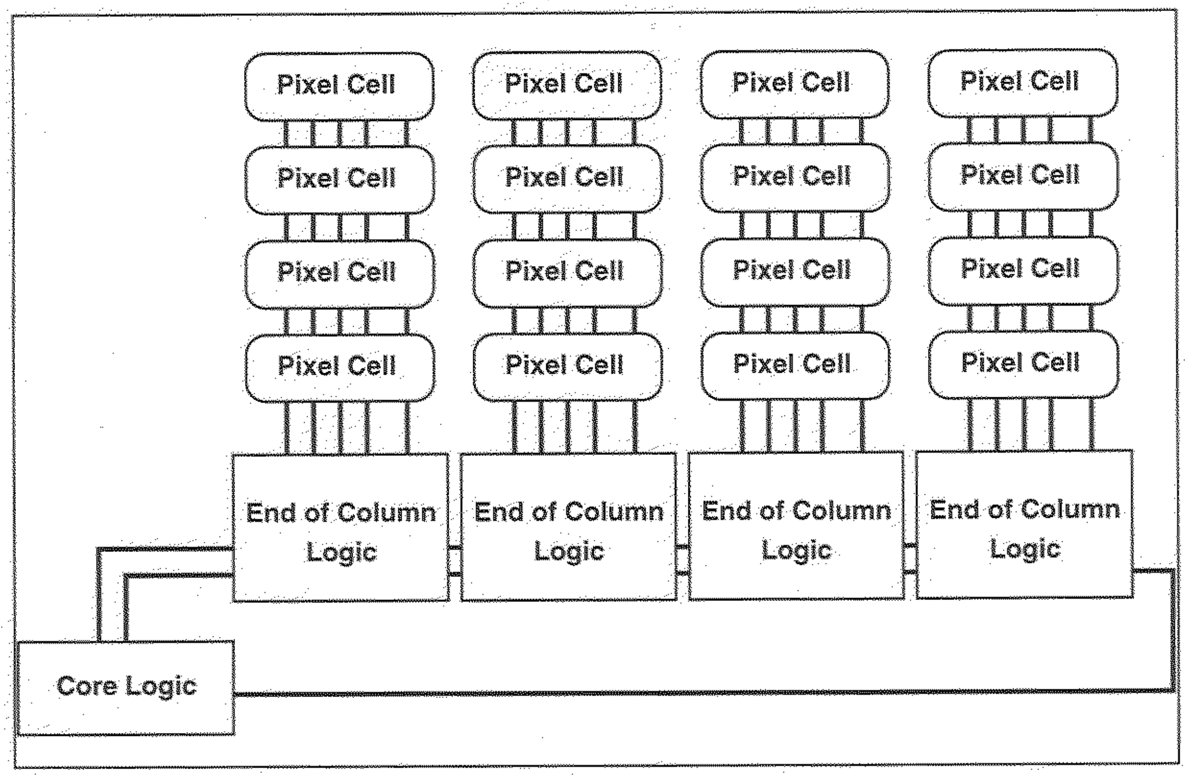


Figure 2:Orgainization of a 4 x 4 Core

4 CORE ORGANIZATION

The organization of the FPIX Core is very similar to the organization of FPIX1 (See Figure 2). The array of pixel cells controls the array of pixel detectors (one cell to one detector). The pixel cells must be, of course, the same physical size as the pixel detectors. The cells are "aware" of hit data coming from the detectors. They are controlled by commands and tokens coming from the End-of-column Logic cells. Most importantly, the pixel cells are not "aware" of time at all. They live in a virtually asynchronous world. Each cell can receive hit data from its pixel detector. If commanded to do so, it records (stores) that hit data. Finally, if commanded to do so, the cell will either output or reset its recorded data.

Pixel cells are organized into columns, each of which is controlled by an End-of-column Logic. The End-of-column Logic is "aware" of the presence of recorded data somewhere in the column. It also controls the commands and tokens sent to the pixel cells in its column, though it is not aware which pixels are obeying which commands. The End-of-column Logic is aware of time through the BCO clock and Read clock as well as the BCO number provided by the Core Logic. Most importantly, the End-of-column Logic knows whether it is Talking or Silent. If it is Silent, it knows if it has Nothing to Say, if it has Something to Say (the next time it gets the token) or if it has Finished Talking.

Finally, the Core Logic controls access to the output data bus by controlling the so-called Horizontal Token. Similar to the End-of-column Logic, the Core Logic knows whether it is Talking or Silent. At startup, the Core Logic is reset to the Silent state. When there is data to be output from any End-of-column Logic, the Core Logic makes a transition to the Talking state and initiates the Horizontal Token drop. When the Horizontal Token makes it out of the other side of the columns, the Core Logic makes the transition back to the Silent state. The Core Logic provides the BCO number to all columns and it contains some diagnostic logic as well.

It is very important to realize that nothing in the FPIX Core is aware of a chip token. The chip token was used in FPIX1 to indicate to an FPIX1 chip that it had the right to output data onto the external data bus. The FPIX Core *always* thinks it has the right to output data provided that SendData is active. Any chip-to-chip arbitration for an external bus would be handled by the Periphery.

5 THE PIXEL CELL

5.1 SIGNALS

The Kill and Inject Logic Signals

1	KI_RESET	Input. Resets the kill and inject logic so that the cell is not killed and not injected
2	KILL_CLK KILL_CLKB	Input. Advances/clocks the kill shift register
3	KILL_SHIN	Input. Shift input for the kill shift register
4	KILL_SHOUT	Output. Shift output for the kill shift register
5	INJ_CLK INJ_CLKB	Input. Advances/clocks the inject shift register
6	INJ_SHIN	Input. Shift input for the inject shift register
7	INJ_SHOUT	Output. Shift output for the inject shift register

Command Signals

8	COMA<1:0>	Input. Command Pair
9	COMB<1:0>	Input. Command Pair
10	COMC<1:0>	Input. Command Pair
11	COMD<1:0>	Input. Command Pair

Token and Control Signal

12	ACCEPT	Input. Controls the Hit Data to Recorded Data conversion. If ACCEPT=1, the pixel cell accepts new hits. If ACCEPT=0 the pixel cell ignores new hits
13	COLTOKENIN	Input. The Column Token Input. If COLTOKENIN=1 and the pixel needs the token, then on the next rising edge of the COLREADCLK, the pixel will output its data. If COLTOKENIN=0, then the pixel must wait.
14	RFASTNOR	Output. Will be pulled low when the pixel needs to output data. The signal is always in response to an Output Command from the End-of-column Logic
15	HFASTNOR	Output. Will be pulled low when the pixel cell has received a hit (has converted hit data into recorded data). The signal is always in response to hit data from a detector cell while ACCEPT=1 and the

		End-of-column Logic is driving a Listen Command.
16	COLTOKENOUT	Output. If the pixel cell does not need to output its data, COLTOKENOUT=COLTOKENIN. If the pixel does need to output data, then COLTOKENOUT=0 until the pixel cell gets control of the bus.
17	COLREADCLK COLREADCLKB	Input. Clock strobe for output data
18	DATARESET	Input. Driven from the main Reset signal pad. When active, any recorded data is wiped from the pixel.
19	READRESET	Input. Equal to the main Reset signal pad OR-ed with the End-of-column Logic entering the Silent state. Guaranteed reset of pixels that have been outputting.
	Data Signals	
20	PIXDATA<7:0>	Output. Row address.
21	PIXDATA<10:8>	Output. Pixel Cell ADC output
	Bias Signals.	
20	VTH<7:0>	Input. Threshold control input (includes main threshold and ADC thresholds).
21	VREF	Input. Reference voltage for second stage amplifier (threshold voltages are relative to VREF)
22	VFB2	Input. Second stage feedback bias voltage (can be connected to VREF)
23	VBBP	Input. Input current. Bias current for the front end preamp.
24	VBBP2	Input. Bias current for the second stage. Can be connected to VBBP provided that the current is doubled.
25	VDIFFB	Input. Bias current for the leakage compensation amplifier.
26	VFF	Input. Bias current controlling the feedback of the preamplifier
27	VBBNL	Input. Bias current for the preamplifier

5.2 PIXEL CELL ANALOG FRONT END

To be written by Abder Mekkaoui

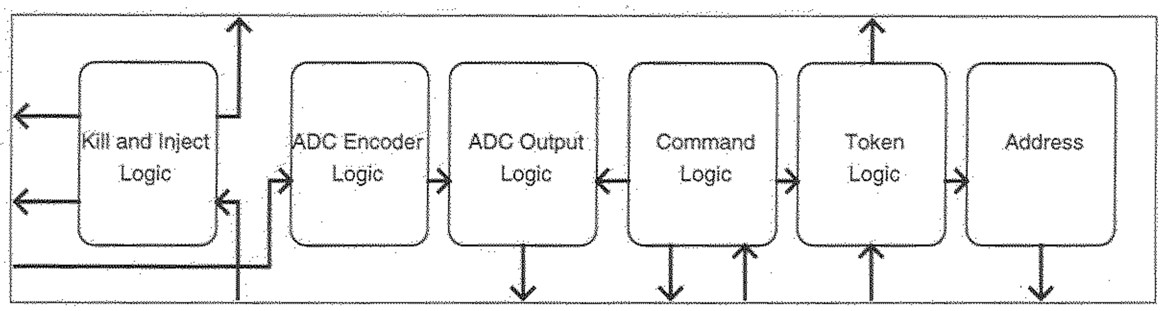


Figure 3: General organization of the Digital Control of the Pixel Cell

5.3 PIXEL CELL DIGITAL CONTROL

The true complexity of the pixel cell lies in the fact that it is rigidly constrained to the size of the pixel detector it controls. From an algorithmic perspective, it is actually quite simple. Figure 4 shows the organization of the Digital Control section of the pixel cell. In the drawing, the analog front end is to the far left and outside of the graphic. The Kill and Inject Logic is one segment of a pair of shift registers that wind throughout the pixel cell array. The kill signal closes off the input of the analog front end. The inject signal enables a diagnostic input of the analog front end. The ADC Encoder Logic converts the seven-bit thermometer code output of the front end's analog-to-digital converter into a three-bit binary code. The ADC Output Logic drives the three ADC bits down the column. The Command Logic interprets the Commands and control signals coming from the End-of-column Logic to the pixel. It controls the conversion of hit data into recorded data. It also controls the resetting of the pixel cell following a read-out or a chip-wide reset. The Token Logic controls the access to the column bus by grabbing or passing a column token. Finally, the Address is the unique positional data of the pixel cell that is driven to the End-of-column Logic whenever the pixel cell outputs its recorded data.

5.3.1 KILL AND INJECT LOGIC

It is absolutely essential to both the testing and the operation of the FPIX chips that each pixel cell be individually injectable and killable. To be injectable means that a test pulse can inject a charge into the pixel cell. This allows each individual pixel to be tested directly and controllably. To be killable means that a pixel cell can be forced to ignore hit data from its pixel detector *and* from test pulses. A killed pixel cell masks hit data from the digital sections of the pixel and, consequently, prevents the hit data from being recorded. This allows noisy pixels to be shut off by the user. This is different from the RejectHits signal that the Periphery provides the Core. First, killing a pixel affects only that pixel whereas activating the RejectHits signal affects the entire pixel cell array. Second, killing a pixel cell is done in the analog front end of the pixel cell whereas activating the RejectHits signal instructs the digital back ends of all pixel cells in the pixel cell array to ignore incoming hits.

The logic for the Kill and Inject signals is simplicity itself. From the perspective of a single pixel cell, the kill state (1=killed, 0=not killed) is stored in one flip-flop and the inject state (1=injectable, 0=not injectable) is stored on a second flip-flop. The outputs of each flip-flop are passed to the analog front end of the pixel cell where they perform the kill and inject functions. There are independent clock signals for each of the two flip-flops, but they share a common reset signal. This is shown in Figure 4 on the left-hand side.

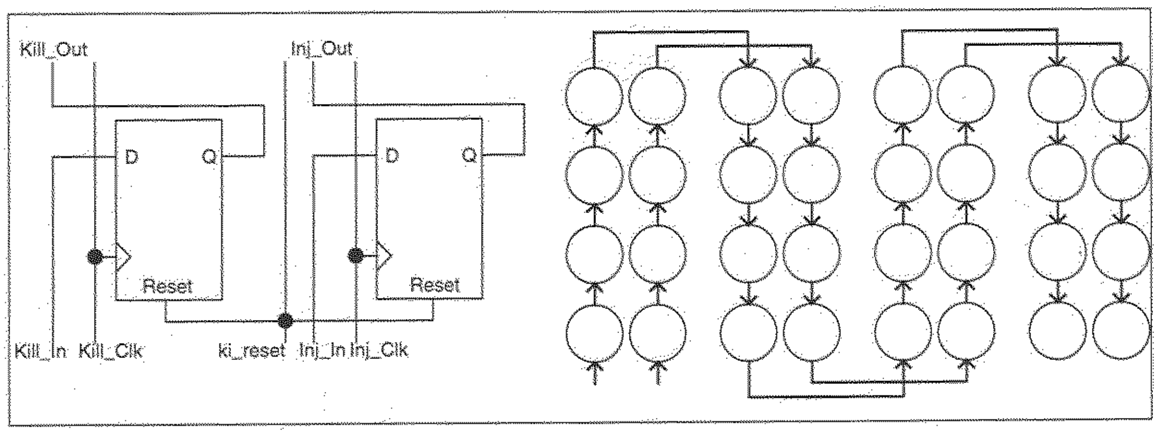


Figure 4: Kill and Inject Logic in preFPIX2(within one pixel cell (left) and in a 4x4 pixel cell array (right))

From the perspective of the entire pixel cell array, the Kill and Inject logic resembles two parallel shift registers. This is shown in Figure 4 on the right hand side. The outputs of each flip-flop in a particular pixel cell are the inputs to the kill and inject flip-flops of the next pixel cell. The separate clocks allow the two scan paths to be operated independently. This is an improvement over FPIX1. The common reset is also an improvement over FPIX1, which required the user to scan in the kill and inject states, even if the user required no kills and no injects. In preFPIX2 and beyond, a programming reset will reset all pixel cells to "not killed" and "not injected".

The kill and inject logic has no effect on the digital back-end of the pixel cell.

5.3.2 ADC ENCODER LOGIC

The ADC Encoder accepts as its inputs the 7-bit wide thermometer-code that is output by the flash ADC located in the analog front end of the pixel cell. The ADC Encoder generates a 3-bit binary output from the input as shown in Table 2

Table 2: ADC Thermometer Code Output vs. Input

Input							Output		
T_6	T_5	T_4	T_3	T_2	T_1	T_0	B_2	B_1	B_0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	1	0	0	1
0	0	0	0	0	1	1	0	1	0
0	0	0	0	1	1	1	0	1	1
0	0	0	1	1	1	1	1	0	0
0	0	1	1	1	1	1	1	0	1
0	1	1	1	1	1	1	1	1	0
1	1	1	1	1	1	1	1	1	1

There are a number of ways to implement this transformation. The minimal method requires 82 transistors. It is a nand-nand CMOS network the outputs of which come from 4-input nand gates. It is an irregular design, which would have required a large amount of routing and, consequently, a large amount of space. For these reasons, this method was rejected.

The chosen method for the transformation is pass-transistor logic, which, while requiring 100 transistors to implement, is extremely regular in its layout and can be fabricated in an area 55µm by 50µm. This design style is essentially an array of CMOS 2-to-1 multiplexors. Each of the seven input bits controls three multiplexors, one per output bit. Depending on the state of each input bit, the multiplexors choose to either pass the output of the previous multiplexor or drive a new output. Algebraically, this is shown as:

$$P_0 = 0$$

$$P_{n+1} = T_n \cdot B_n + \overline{T_n} \cdot P_n \qquad \text{Equation 1}$$

$$Out = P_7$$

where Tn is each of the seven thermometer-code input bits, Bn is the binary number to drive if Tn=1, and Pn is the binary output of the previous multiplexor to be passed if Tn=0. To achieve a thermometer code to binary transformation, the various Bn for the three output bits are defined as shown in Table 3

Table 3

	Bit 2	Bit 1	Bit 0
B_0	0	0	1
B_1	0	1	0
B_2	0	1	1
B_3	1	0	0
B_4	1	0	1
B_5	1	1	0
B_6	1	1	1

The schematic that implements this transformation is shown in Figure 5. The layout is shown in Figure 6. This implementation method is slower than the full CMOS version, but speed is not an important limitation for the ADC Encoder. The End-of-column Logic guarantees that there will be a minimum of one beam crossing period (132ns) between the arrival of a hit and the request for output. The pass-transistor logic implementation is easily capable of settling to its final value in that length of time. Figure 7 clearly shows that not only does the ADC Encoder accurately make the transformation, but it also does it in less than 1ns.

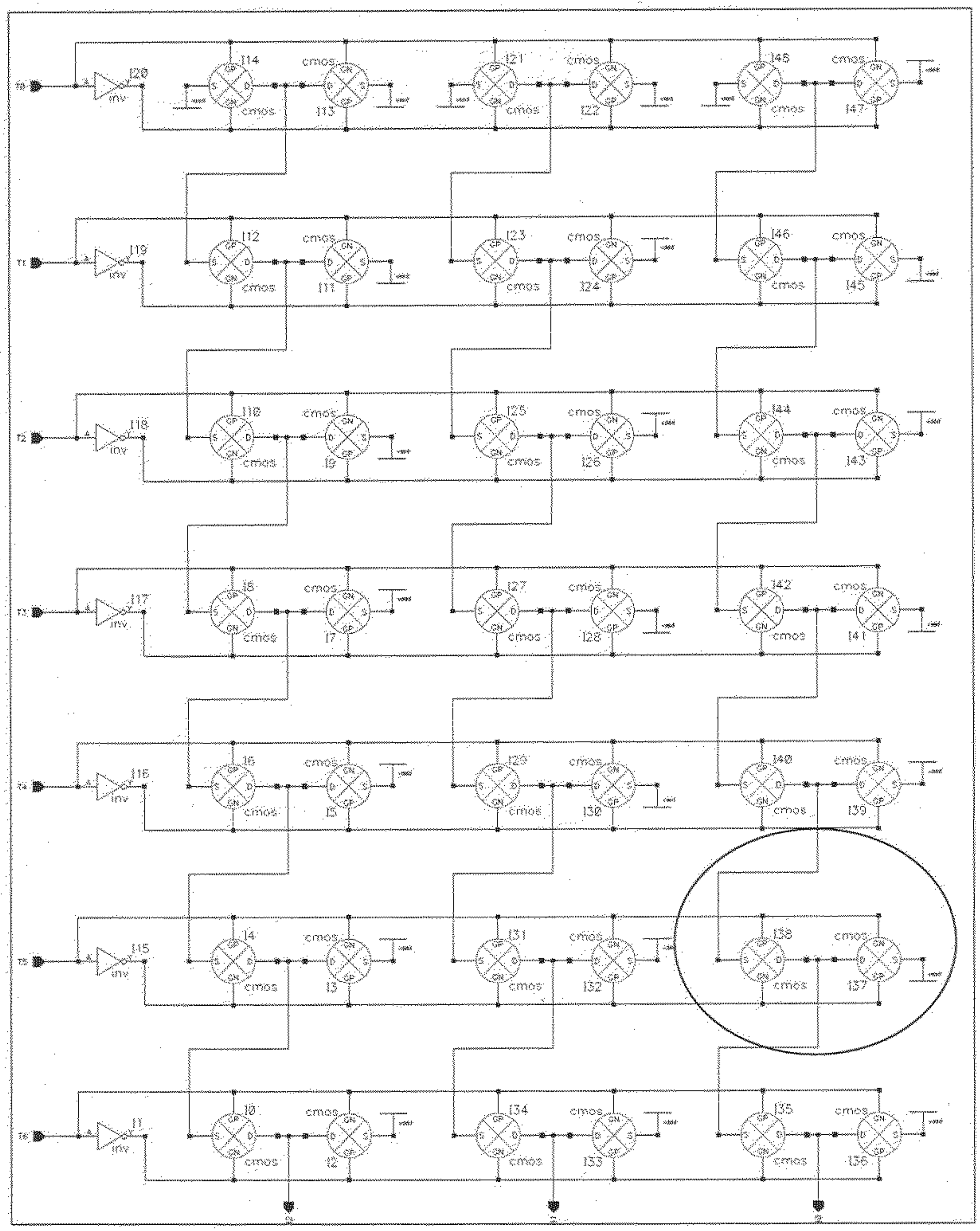


Figure 5: The schematic for the Pixel ADC Encoder. The oval surrounds a single 4-transistor CMOS multiplexor. The encoder is obviously an array of these devices. Their left inputs are the outputs of the previous multiplexors, and their right inputsare either tied to power or ground, reflecting the Bn numbers shown in Table 3.

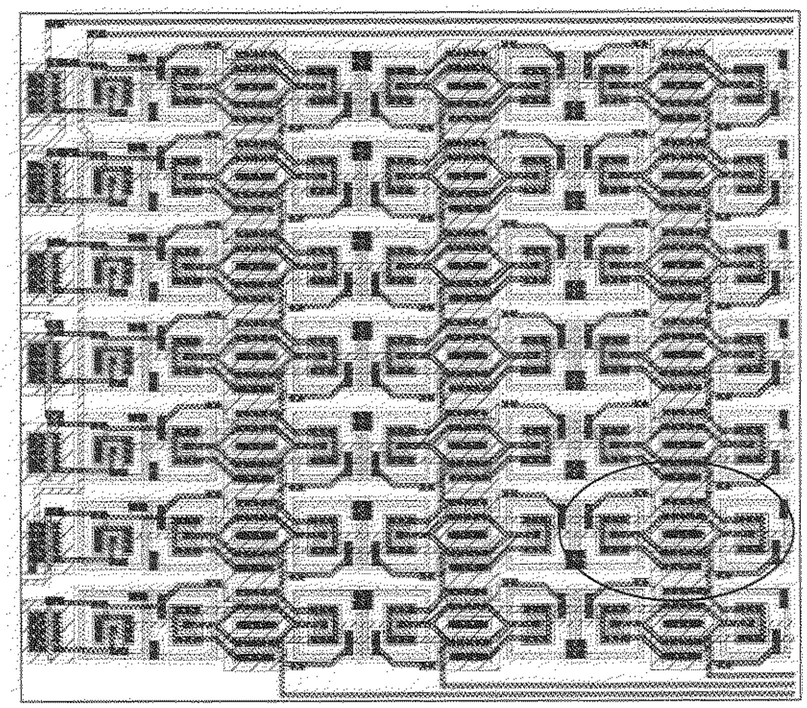


Figure 6: The Pixel ADC Encoder Layout. The oval surrounds a single 4-transistor CMOS multiplexor.

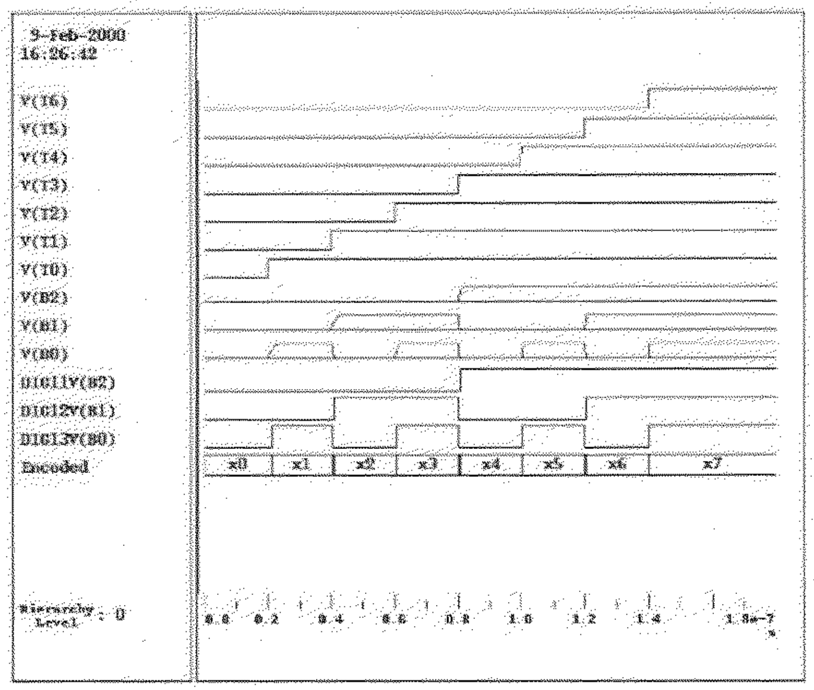


Figure 7: ELDO simulation of the ADC Encode. The upper seven signals [V(T6)-V(T0)] are the incoming thermometer code which changes every 20ns. The next three [V(B2)-V(B0)] are the encoded outputs. The final signal [Encoded] shows the hexidecimal output of the encoder.

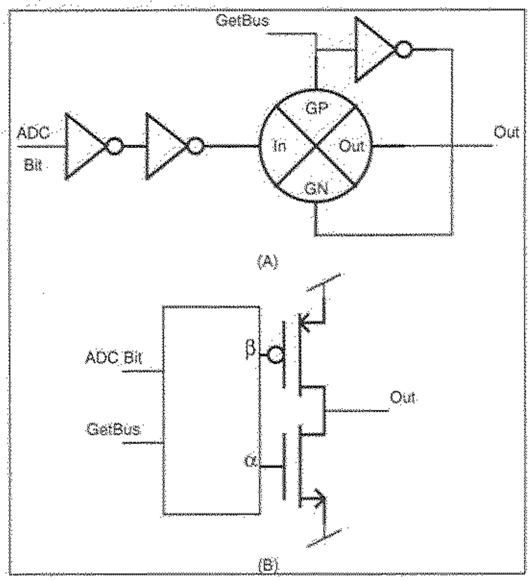


Figure 8: (A) The old (FPIX1) way of driving ADC outputs (B) The new (FPIX2) way of driving ADC outputs

5.3.3 ADC OUTPUT LOGIC

In previous versions of FPIX, there was no encoder. Instead, the outputs of three comparator latches were driven directly by CMOS drivers. CMOS switches controlled by the pixel GetBus signal either connected the drivers to or tri-stated the drivers from the three column-wide output lines. This is shown in Figure 8(A).

There are several improvements that can be made to this approach. First, the CMOS drivers require two inverting stages to drive the original signal. Second, the CMOS switches must be made very large to minimize the resistance on the output lines. These large CMOS switches also serve to load the GetBus signal, which controls the release of both pixel address data and pixel ADC data.

In preFPIX2, these two functions (driving and tri-stating) are combined into two transistors, one pull-up and one pull-down. At any given time, these two transistors either will both be off or only one will be on. If they are both off, the ADC Output is tri-stated. If one or the other is on, the output line will be pulled high (if the encoded ADC bit is a one) or low (if the encoded ADC bit is a zero). In this case, the speed of the GetBus signal is improved because it is only loaded by a minimum size CMOS gate. This is shown in Figure 8(B).

The logic inside the box in Figure 8(B) is very straightforward. If GetBus is a zero, then α=0 and β=1. If GetBus is a one and ADC Bit is a zero, then α=1 and β=1. If GetBus is a one and ADC Bit is a one, then α=0 and β=0. This yields the following equations for α and β:

$$\alpha = \left(ADCbit + \overline{GetBus}\right)' \qquad \text{Equation 2}$$
$$\beta = \left(ADCbit \bullet GetBus\right)'$$

These equations can be easily implemented with eight minimum sized transistors. All three ADC Output drivers with their support logic can be laid out in an area 19μm by 50μm. this is shown in

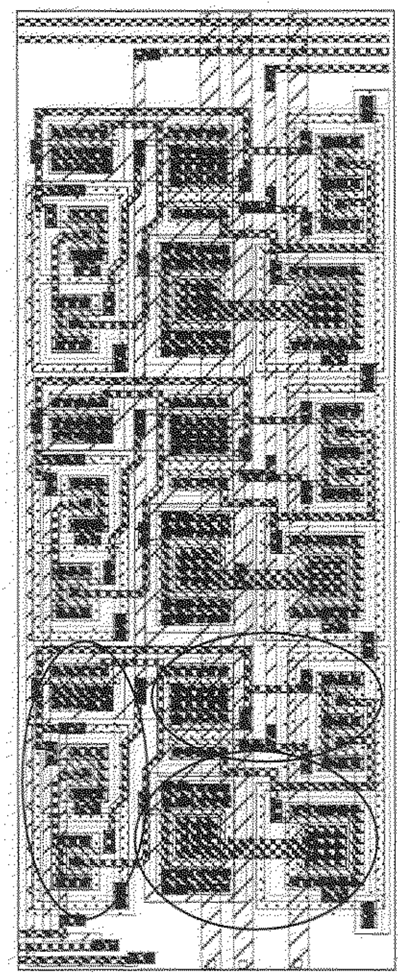


Figure 9: The ADC Outpu DriverLayout. The three ovals cover (clockwise from the lower right) the two drive transistors, the NOR gate for generating α and the NAND gate for generating β. These three circuits are repeated three times, once for each ADC bit.

5.3.4 COMMAND LOGIC

In the interest of keeping this section self-contained, some of this information may be a repetition of information given elsewhere. First, the Command Logic of a pixel cell is either Full or Empty, meaning it is either holding recorded data from a previous hit or it is not. Second, the Command Logic in each pixel cell accepts as input four pairs of Command Lines from the End-of-column Logic. Each pair of Command Lines is used to transmit the following commands:

1.	**Idle**	**(00)**	– do nothing
2.	**Listen**	**(11)**	– listen for incoming hits
3.	**Output**	**(10)**	– output data (address and ADC) when the pixel cell gets the bus
4,	**Reset**	**(01)**	– reset the Command Logic of a pixel cell back to Empty and reset the ADC data

Other Command Logic input signals are:

1.	AnaNewHit	– from the analog front end of the pixel cell; the actual "new hit" signal
2.	Accept[1]	– from the End-of-column Logic; If high (One), accept new data; If low (Zero), reject new data.
3.	GetBus (gbus)	– from the Token Control Logic; indicates whether or not the pixel cell controls the output bus

The Command Logic output signals are:

1.	NeedToken	– a signal to the Token Control Logic that pixel cell needs the output bus; only happens when the Command Logic is Full and has been ordered to Output.
2.	HFastOR	– a signal to the End-of-column Logic that the pixel cell has received new hit data and converted it to recorded data; only happens when an End-of-column register is issuing a Listen
3.	RFastOR	– a signal to the End-of-column Logic that the pixel needs to the output bus; only happens when the Command Logic is Full and has been ordered to Output.

5.3.4.1 Algorithm

1) While Empty, the Command Logic observes all four command lines with equal priority and ignores all commands except the Listen Command.

2) When AnaNewHit arrives from the analog front end:

[1] The Accept signal is derived from the RejectHits input to the Core.

a) If the Accept signal is low (Zero), the new hit data is ignored and discarded. Nothing happens.

b) If the Accept signal is high (One) and the pixel cell is Full, the new hit data is ignored and discarded. Nothing happens.

c) If the Accept signal is high (One) and the pixel cell is Empty:

   i) If no Command Lines are ordering the pixel to "Listen", the pixel cell waits until one of the Commands Lines orders it to "Listen". The pixel cell will wait indefinitely for this to happen. It is the responsibility of the End-of-column Logic to make sure the pixel cell does not wait forever.

   ii) If a pair of Command Lines is ordering the pixel to Listen:

      (1) the pixel cell will latch the hit data, thereby completing the transformation of hit data to recorded data

      (2) The pixel cell will focus its attention onto the pair of Command Lines that had issued the Listen Command when the hit data arrived. Until the pixel cell is read-out or reset by commands from that particular pair of Command Lines, the pixel cell will ignore commands from all other Command Lines. This is called associating the pixel with an End-of-column Register.

      (3) The pixel cell will consider itself Full until it is read-out or reset.

      (4) The pixel cell will alert the End-of-column Logic of the presence of a hit by pulling low the HFastOR signal. It will hold this signal low until its associated End-of-column Register acknowledges the hit by withdrawing the Listen Command.

3) While Full (not Empty):

   a) Commands issued on unassociated Command Lines are ignored.

   b) If the associated Command Lines issue an Idle Command, the pixel cell does nothing, and the recorded data remains recorded

   c) If the associated Command Lines issue a Listen Command, the pixel cell will pull the HFastOR signal low until the Listen Command is withdrawn. Recorded data remains unchanged. (This situation should never happen under ordinary operation. This condition was added to the algorithm to increase the Single Event Effect tolerance of FPIX2.)

   d) If the associated Command Lines issue a Reset Command

      i) The pixel cell will reset itself

      ii) The Full pixel cell will become Empty.

      iii) The recorded data will be erased

iv) A reset will be issued by the Command Logic to the analog front end of the pixel cell for the purpose of resetting the ADC latches.

e) If the associated Command Lines issue an Output Command

i) The pixel cell acknowledges the command by pulling the RFastOR low. The pixel cell will remain in this state for as long as the Output Command is being issued to the Full pixel cell.

ii) The Command Logic signals the Token Control Logic via the NeedToken signal that it needs the output bus. The pixel cell will remain in this state for as long as the Output Command is being issued to the Full pixel cell.

iii) Eventually, the Token Control Logic will signal the Command Logic that the pixel cell has control of the output bus (via the GetBus signal).

(1) The GetBus signal will reset the Command Logic (see 3.d.i through 3.d.iii above).

(2) The pixel cell will release the RFastOR signal.

(3) The next read clock cycle, a signal will be released by the Command Logic to reset the analog front end of the pixel cell.

The true difficulty in the above algorithm is that all of this functionality had to be implemented in an area 50μm by 75μm. The design is best understood if it is broken down into five subsections: the Front Command Cells, the Hit Conditioners, the Reset Logic, the Passed Command Cells, and the FastOR Logic.

5.3.4.2 *The Front Command Cells*

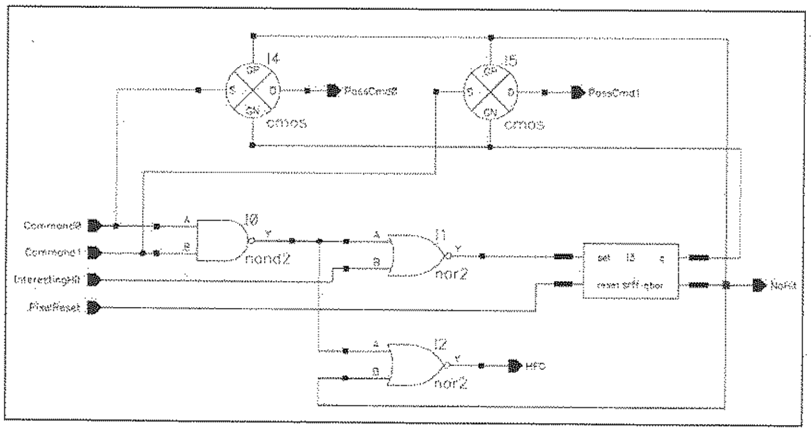


Figure 10: A Schematic of the Front Command Cell

There are four Front Command Cells in each pixel cell. One Front Command Cell connects to each pair of Command Lines. These four separate Front Command Cells allow an Empty pixel cell to observe all four End-of-column Registers with equal priority as required in the above Algorithm, Part 1.

Each Front Command Cell contains the logic to decode the Listen Command (and only the Listen Command). In fact, the code for the Listen Command (11) was chosen to simplify the Front Command Cell because with this code, a single nand gate is all that is necessary to decode the command.

All Front Command Cells are connected to the InterestingHit signal, which is a conditioned version of the hit signal output by the analog front end of the pixel cell. More will be said about this signal later. For now, it is enough to know that it activates when there is new hit data for this pixel cell. It is active low.

Each Front Command Cell contains a single SR flip-flop. This flip-flop is actually where recorded data is stored. If a pair of Command Lines connected to a Front Command Cell is issuing the Listen Command and InterestingHit goes active, then the SR flip-flop in that Front Command Cell will be set and the pixel cell is now Full. The flip-flop will remain set until the pixel cell is reset or until the pixel cell is read out. The act of setting one of these flip-flops is actually the transformation of hit data to recorded data. The complementary outputs of this SR flip-flop are called Hit and NoHit. They have two purposes:

1. Within each Front Command Cell, the Hit and NoHit signals are used to open or close two CMOS switches (see Figure 10). The inputs to these switches are the two Command Line inputs to the Front Command Cell. The outputs of these two switches are the two PassCmd signals. If a Front Command Cell has recorded a hit in its SR flip-flop, then the two Command Lines are passed to the PassCmd signals. If a Front Command Cell does not have a recorded hit in its SR flip-flop, then the two Command Lines are blocked from the PassCmd signals. This is how the Command Logic associates itself with only one End-of-column register. When the pixel cell is Empty, each of the four Front Command Cells are "looking" at their respective End-of-column register. The Front Command Cells can only recognize the Listen Command. Any End-of-column register issuing Idle, Output or Reset is ignored. At any given time, only one of the End-of-column registers will be issuing a Listen command. When a hit arrives, the SR flip-flop in the Front Command Cell connected to that pair of Command Lines will be set, and those Command Lines will be passed to the rest of the pixel cell where logic exists to decode other commands. Other Command Lines will be blocked from the rest of the pixel cell and their commands ignored. As shall be shown later, InterestingHit will be prevented from going active as long as the pixel cell is Full.

2. External to the Front Command Cells, all of the NoHit signals are ORed together to produce a signal that is high (One) when the pixel cell is Empty and low (Zero) when any Front Command Cell SR flip-flops are set. This signal is called PreviousHitb.

5.3.4.3 *The Hit Conditioners*

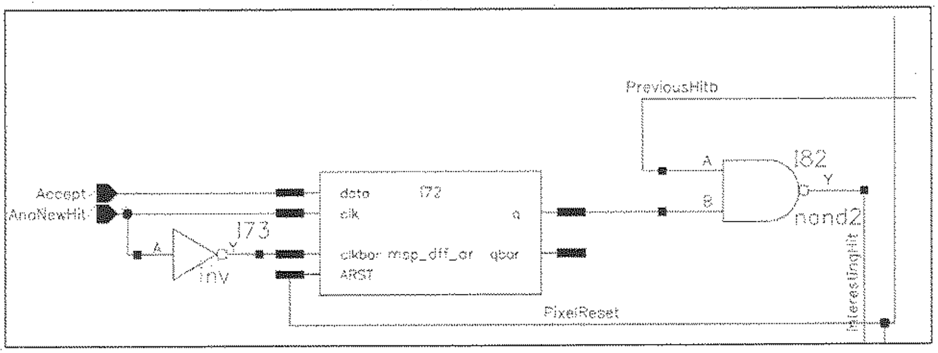


Figure 11: The Command Logic Hit Conditioner. "msp_dff_ar" is an asynchronously resettable positive edged, D flip-flop.

The job of the Hit Conditioner is to block unwanted hits while passing the desired ones as quickly as possible. The Hit Conditioner is shown in Figure 11.

The Accept signal is a One if the End-of-column Logic wishes the pixel cells in its column to accept new hits. It is a Zero otherwise. AnaNewHit is the discriminator output of the analog front end of the pixel cell, which goes high when there is a new hit. The positive edge of AnaNewHit clocks an edge-triggered D flip-flop (msp_dff_ar). If Accept is a One, then the output of the flip-flop will be a One indicating a new hit. If Accept is a Zero, then the output of the flip-flop will be (remain) a Zero, and the Command Logic will never know that a hit occurred. It is possible to condition AnaNewHit with a simple AND gate that combines Accept and AnaNewHit. However, the output of that AND gate would depend on the duration of the AnaNewHit signal, which itself is dependent on bias settings and radiation damage. Hits that occurred while Accept was a Zero might appear as hits that suddenly occurred as soon as the Accept signal was restored to a One. The flip-flop prevents that from happening. Only hits whose rising edge arrives when Accept is a One will generate hit data in the Command Logic.

The output of the flip-flop is further conditioned by the PreviousHitb signal which is a Zero when the pixel cell is full and a One when it is Empty. InterestingHit, the output of the Hit Conditioner, will go to Zero only for a hit that arrives when Accept is a One *and* the pixel cell is Empty.

5.3.4.4 *The Reset Logic*

There are several resets that affect the pixel cell. The first is the Reset Command, which is an order by the End-of-column Logic to reset. This would be a column-wide reset for any pixel cells obeying that particular pair of Command Lines. The second is a Master Reset, which is a chip-wide order from a user to erase the data in the FPIX and start over. The third is an indication from the Token Control Logic that the pixel has the bus. When this signal is received, the pixel is outputting its data, and the recorded data can be erased in preparation for the next hit.

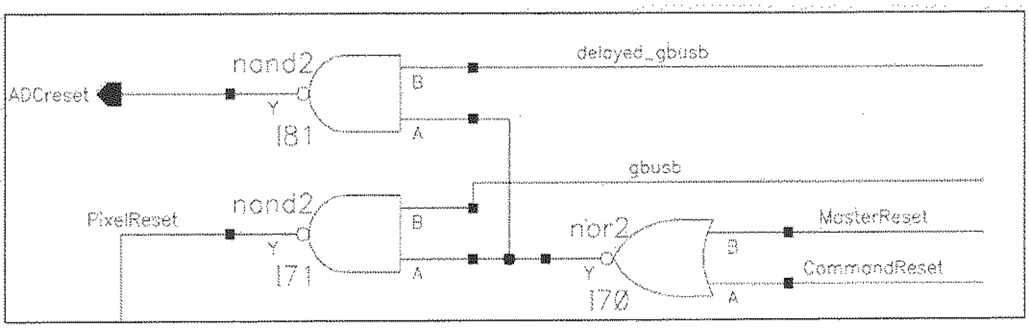


Figure 12: The pixel reset logic

Note that in Figure 12 there are two outputs from the reset logic. One is PixelReset, which resets the Front Command Cells and the Hit Conditioner. The other is the ADC reset, which resets the flip-flops in the analog front end. The only difference between the two is delay. In the Figure, gbusb is the indication from the Token Control Logic that the pixel cell controls the bus. "delayed_gbusb" is identical to gbusb except that it is delayed by one Read Clock period. This delay is essential because the ADC data is stored on latches and it cannot be reset until after the read-out is done.

5.3.4.5 The Passed Command Logic

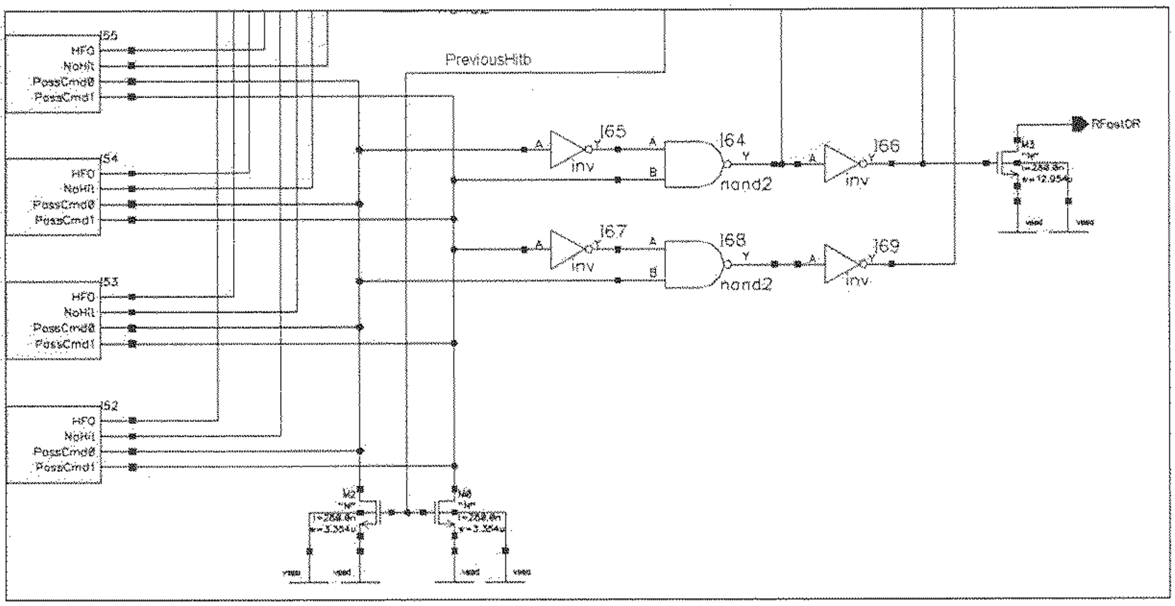


Figure 13: The Passed Command Logic showing the outputs of the four Front Command Cells (far left), the pull-down transistors (bottom), the command decode logic (center right) and the RFastOR transistor (far right).

In Section 5.3.4.1, the Algorithm for the Command Logic specifically mentions that an Empty pixel cell must view all four Command Lines with equal priority and must respond only to a Listen Command. A Full pixel cell, on the other hand, must obey only the commands issued to it by its associated End-of-column Register. The four Front Command Cells accomplish these two all-important tasks. First, each Front Command Cell is connected to a different pair of Command Lines and each Front Command Cell contains its own Listen Command decoder so that all four pair of Command Lines are observed equally. Second, an association between a pixel cell and a particular End-of-column Register is established when one of the Front Command Cells is receiving the Listen Command from its Command Lines as a hit occurs[2]. Under this specific set of circumstances, a flip-flop internal to that particular Front Command Cell is set, and CMOS switches are opened in the Front Command Cell, allowing only its Command Lines to be passed onward.

These associated Command Lines are passed to the Passed Command Logic shown in Figure 13. This is little more than two two-bit decoders, one that looks for 01 (Reset) and the other that looks for 10 (Output). If the Reset Command is issued, then the Passed Command Logic activates the CommandReset signal to the Reset Control Logic. If the Output Command is issued, then the Passed Command Logic activates the NeedToken signal to the Token Control Logic and pulls down the RFastOR. If the Listen Command or Idle Command are issued, they have no effect on the Passed Command Logic.

When the pixel cell is Empty, no Command Lines are passed to the Passed Command Logic. This ensures that the pixel cell ignores all commands but Listen while it is empty. However, floating inputs to the Passed Command Logic could cause spurious errors by dynamically storing erroneous commands. This would also be a serious source of Single Event Effect errors if left uncorrected. Therefore, the transistors shown on the bottom of Figure 13 are used to force the inputs to the Passed Command Logic to the Idle Command (00).

5.3.4.6 The FastOR Logic

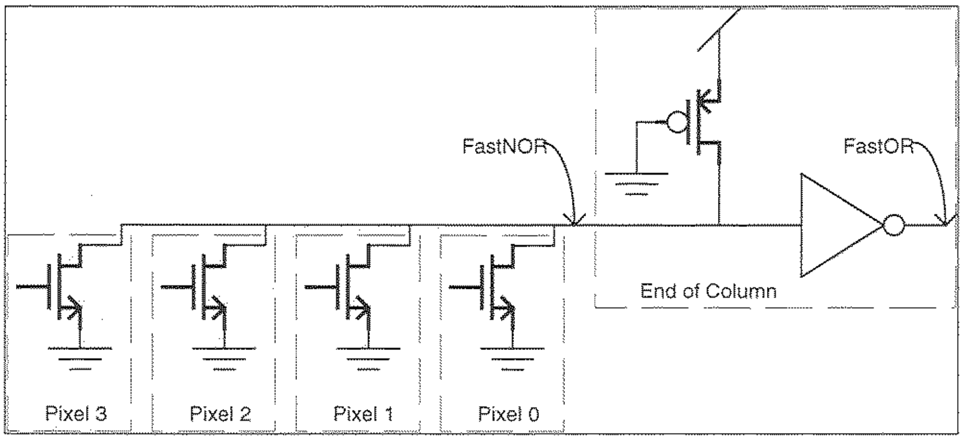


Figure 14: A schematic of the FastOR Logic

Figure 14 shows a simple schematic of a FastOR system. It is essentially a distributed pseudo-NMOS NOR gate with an inverter. Each of the four "pixel cells" in the "column" contains a single pull-down transistor. The "End-of-column Logic" contains a pull-up transistor and an inverter. Whenever a gate of one of the pull-down transistors goes high, the FastNOR line

[2] It is the job of the End-of-column Logic to make sure only one End-of-column Register is issuing the Listen Command at any given time.

is lowered, and the FastOR line is driven high. If all of the gates of the pull-down transistors are low, then the pull-up transistor raises the FastNOR line and the FastOR line is driven low.

The key performance issues are the width-to-length ratio of the pull-up pfet, the width-to-length ratio of the pull-down nfets, and the geometry of the FastNOR line itself. In FPIX1, there was some difficulty with the FastNOR lines due to their irregular shape. This shape led to a higher-than-anticipated resistance on the RFastOR and HFastOR lines. Consequently, hit data occurring in the upper pixel cells (more than 90 – 100 pixel cells away from the End-of-column Logic) was not recognized by the End-of-column Logic because it could not pull the FastNOR lines below the trip point of the inverter (See Figure 14). Therefore, in preFPIX2 and beyond, the geometry of the FastNOR lines is rigidly defined to be rectangular. The only variables in the geometry of the FastNOR lines are their length and their width. The pixel width and the number of pixels per column, of course, define the length. It is, therefore, is out of the hands of the chip designer. Simulation and practicality determine FastNOR line width.

Larger nfet width (See Figure 14) increases the drive capability of each pull-down transistor. However, it also increases the capacitance of the FastNOR line, which slows signal propagation. Larger pfet width decreases the rise time of the FastNOR line, but it also increases the fall time and the minimum voltage attainable by the pull-down transistors. If this voltage approaches the trip point of the inverter, it threatens the ability of a pixel cell to make the FastOR signal.

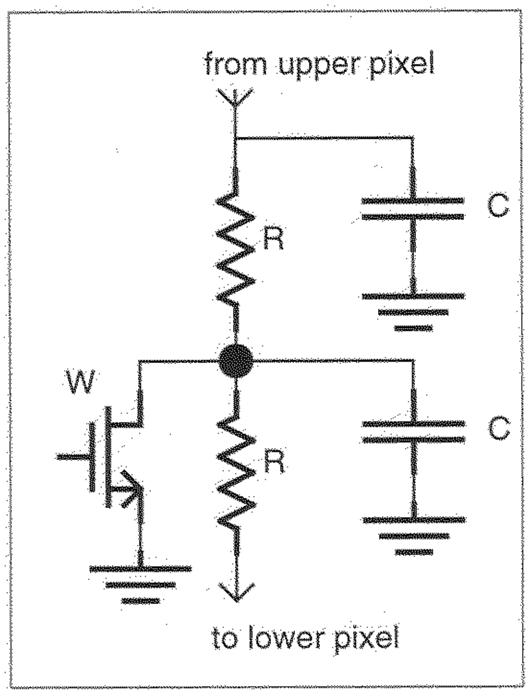


Figure 15: Pixel Cell Model used in simulations of the FastOR circuitr

In order to simulate the relative effects of nfet width, pfet width and line width, numerous simulations were performed. In each simulation, each pixel cell was modeled as shown in Figure 15. It was decided before layout began that all signals propagating up or down a column would be carried by metal2 (second level metal). Since the pixel width is defined to be 50μm, the resistance R in the figure represents the resistance of 25μm of metal2. Similarly, the capacitance C represents the capacitance of 25μm of metal2 covered above by metal3 and below by metal1 and running parallel to 25μm of metal2 on either side. In other words, the maximum capacitance that could be experienced by a metal2 run. It was also assumed that the width W would be constrained to those widths obtainable by enclosed geometry transistors. For a 160 pixel column, the conclusions of these simulations are shown in Table 4.

Table 4: FastOR optimal geometries

Nfet width	12.95μm
Pfet width	5.0μm
Line width	2.0μm

Simulations of the above geometries under sigma and supply variation are shown in Figure 16

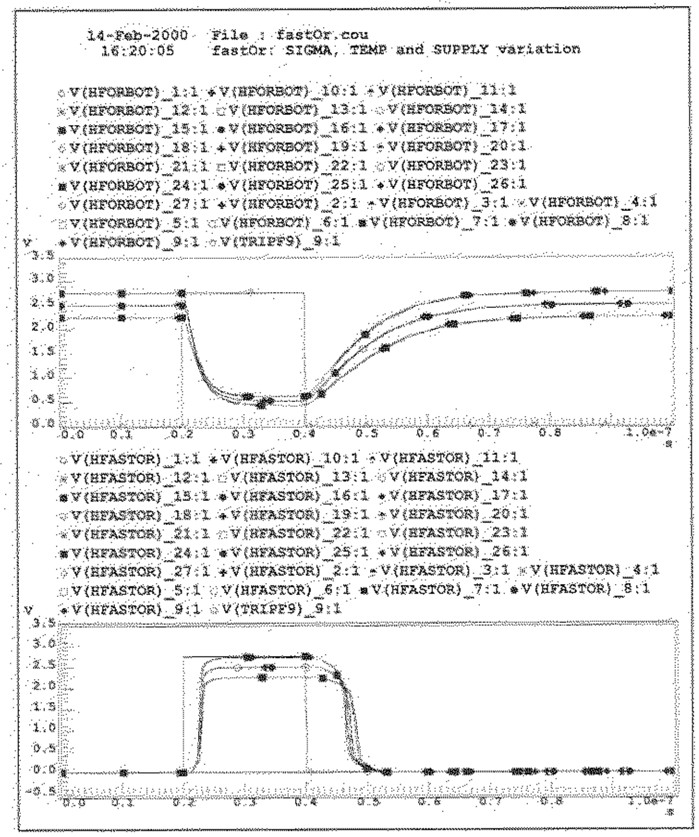


Figure 16: Simulations of the FastOR

The square waves in the figure show the topmost pixel cell activating its nfet pull-down. The upper set of waves are the signal on the FastNOR line right next to the End-of-column Logic. The lower set of waves are the FastOR output.

The propagation delay of the rising edge of the FastOR signal is 2.5 to 3ns. The propagation delay of the falling edge is 5 to 10ns. The minimum voltage of the FastNOR line is 0.4 to 0.6

volts. Of considerable importance is also how the FastOr behaves over a ±20% variation in line width. This corresponds to a ±20% variation in both line resistance and capacitance.

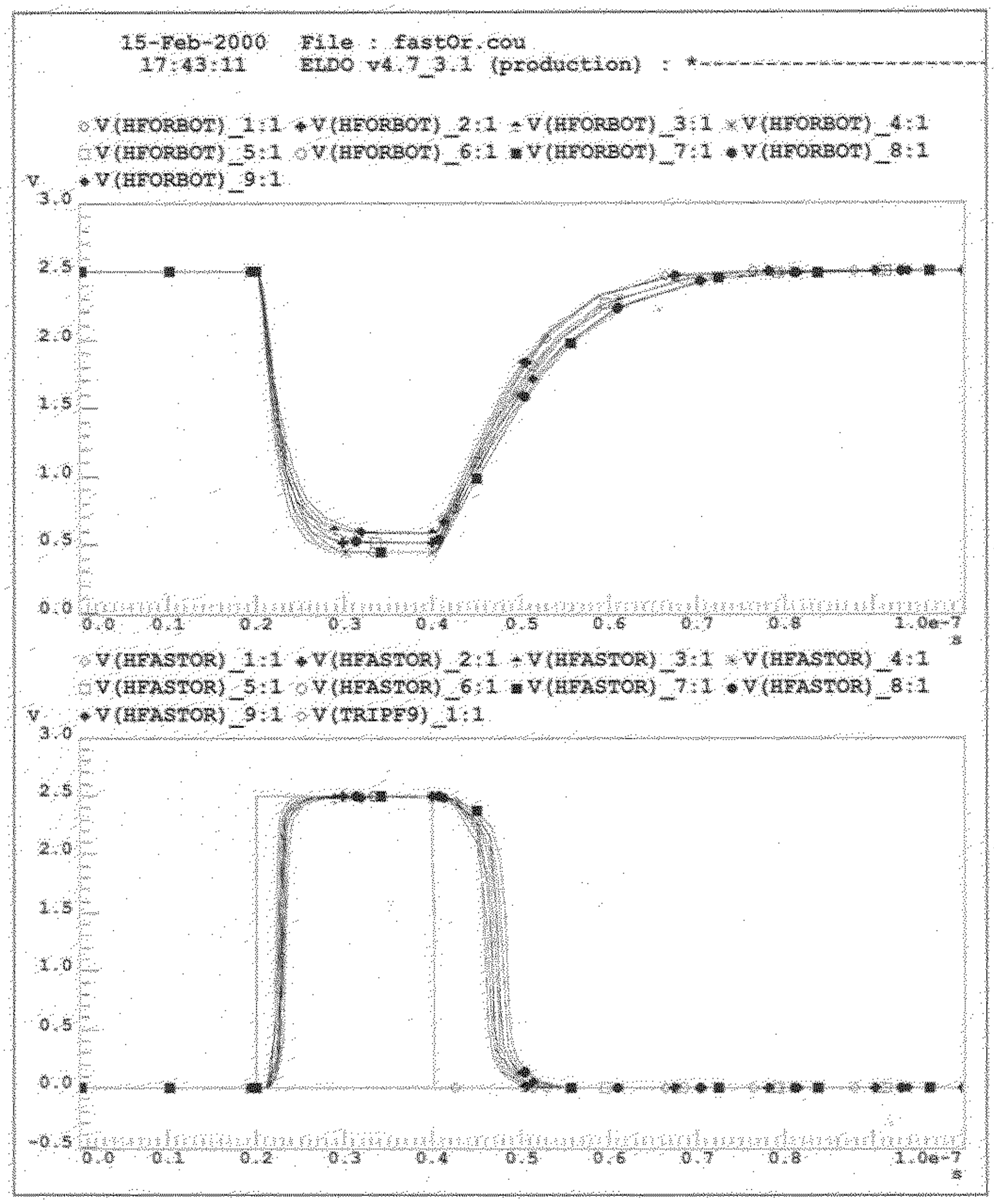


Figure 17: FastOr simulations as a function of line resistance and capacitance

Figure 17 shows that the rising edge propagation delay is 2.2ns to 3.6ns. The falling edge propagation delay is between 5.8ns and 8.4ns. The FastNOR minimum voltage is between 0.4V and 0.6V. All delays are well within the timing requirements of FPIX2. **Note:** All geometries and timings will have to be recalculated for any change in pixel width or in the number of pixels per column.

5.3.5 TOKEN CONTROL LOGIC

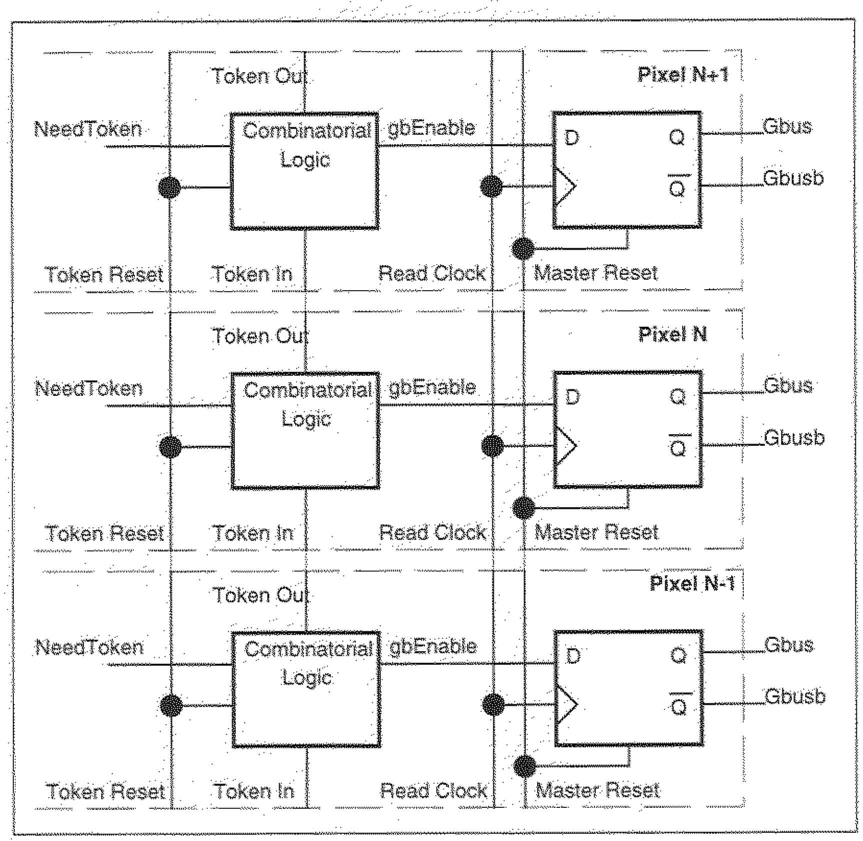


Figure 18: The Interactions of several Token Control Logic Cells

Frequently, several pixels in a single column are hit simultaneously. When this happens all of those hit pixels will associate themselves with the same End-of-column register. Therefore, when that End-of-column register issues the Output Command, all of those pixels will try to get control of the output bus. A column token imposes a sequential order onto the readout of those pixels so they do not all output their data simultaneously. To reiterate, the readout sequence is as follows:

1) A pixel has recorded data.

2) The pixel receives the Output Command from its associated End-of-column register

3) The pixel receives the column token

4) When the previous three conditions are met, at the next rising edge of the Read Clock, the pixel cell will:

   a) Output its data (address and ADC);

   b) Release the column token to the next pixel that needs the bus; and

   c) Reset itself.

To accomplish the above sequence, each pixel needs Token Control Logic interacting as shown in Figure 18. This logic has three main goals: 1) to acquire the token when it becomes available if the pixel cell needs it; 2) to pass the token as rapidly as possible if the pixel cell does not need it; and 3) to clear all token information as rapidly as possible to make the column ready for the next readout sequence. Goal number 3 was a limiting factor for previous versions of FPIX which did not have the Token Reset shown in Figure 18. In these older versions, the End-of-column Logic would assert the column token at the beginning of a readout sequence, and then withdraw it when the readout was finished. The asserted token would propagate up the column, being grabbed by pixel cells that needed it and being passed by pixel cells that did not. At the conclusion of the readout sequence, the "withdrawal" of the token would have to propagate up the column through all pixel cells sequentially. Obviously, if a second readout sequence was issued too quickly, then it would be possible for more than one pixel cell to think it had control of the bus at the same time. The "withdrawal" of the token might not have had enough time to propagate all the way up the column, and pixels higher in the column might think that they still had the token.

The Token Control Logic has two parts, a purely combinatorial section and an asynchronously resettable, positive edge-triggered d-flip-flop that ensures that data output will be synchronous to the Read Clock. The combinatorial section, labeled as such in Figure 18, has two inputs, NeedToken from the Command Logic section of the pixel cell and TokenIn from the previous pixel cell. It also has two outputs, gbEnable which is the input to the Token Control flip-flop and TokenOut which becomes the TokenIn of the next pixel cell. Within the Token Control Logic itself, for historical reasons, NeedToken is referred to as "hit" and the inverse of NeedToken as "hitb". Details of the combinatorial section are shown in Figure 19.

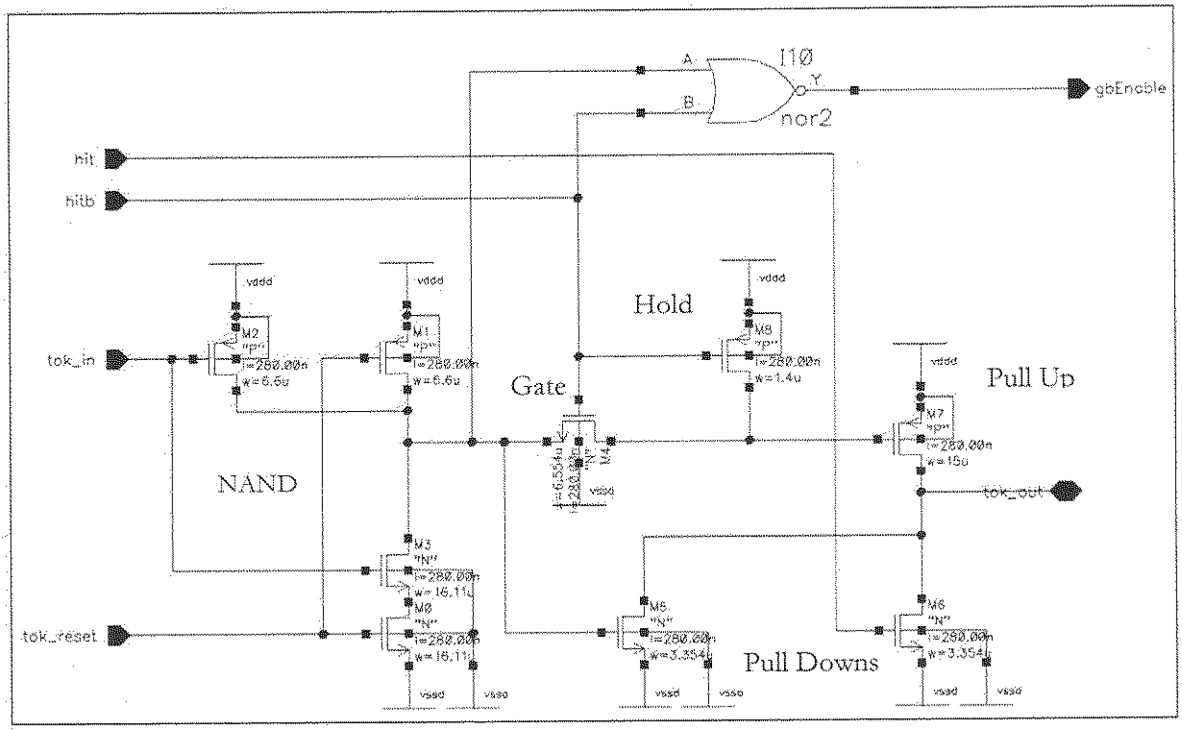


Figure 19: The combinatorial portion of the Token Control Logic

From Figure 18, the Read Clock is distributed to all pixel cells in a column simultaneously. In order to fulfil the required readout sequence, gbEnable should only be activated when the pixel cell has the token and the pixel cell needs the token. By DeMorgan's Law,

$$NeedToken \bullet TokenIn = (\overline{NeedToken} + \overline{TokenIn})' \qquad \text{Equation 3}$$

This explains the simple NOR gate driving the gbEnable signal in Figure 19. The TokenOut is not quite as simple.

1) If NeedToken (hit) is active (high), then TokenOut must be a zero. In other words, the token cannot be passed until this pixel cell has gained control of the bus and has been reset to Empty. Therefore, NeedToken (hit) is connected to the gate of an nfet transistor that pulls TokenOut down to zero when NeedToken is a one.

2) Regardless of the state of NeedToken, if the token has not yet arrived to a pixel cell (TokenIn is zero) then TokenOut must be zero. TokenIn and Token_res are the two inputs to the NAND in Figure 19. The output of that NAND is connected to the gate of an nfet transistor that is capable of pulling TokenOut down to zero. If Token_res is a innactive (Token_res=1), then the output of the NAND gate is equal to the inverse of TokenIn. Therefore, if the token has not yet arrived (TokenIn is a zero) then the output of the NAND gate is a one, and TokenOut is a zero.

3) When Token_Res is activated (Token_Res=0), the output of the NAND is automatically driven to a one. Therefore, TokenOut will be pulled to a zero by the pull-down transistor connected to the output of the NAND.

4) When the pixel cell does not need the token (NeedToken=0) then hitb is a one. When hitb is a one, the "Gate" transistor connects the output of the NAND directly to the TokenOut pull-up transistor. Therefore, as soon as the TokenIn arrives, TokenOut will be pulled to a one. This is how fast passing of the token is accomplished.

5) When the pixel cell needs the token (NeedToken=1), then hitb is a zero, and the "Gate" transistor disconnects the output of the NAND from the TokenOut pull-up transistor. Instead, the relatively weak "Hold" transistor keeps the gate of the TokenOut pull-up transistor high, cutting-off the pull-up transistor and leaving the state of the TokenOut in the hands of the pull-down transistors.

6) Finally, when a pixel cell needs the token (NeedToken=1), TokenOut is essentially forced to a zero. When the token does arrive, gbEnable is activated, and on the next rising edge of the Read Clock, the pixel cell will get the bus. When the pixel cell gets the bus, it will reset itself to Empty via the Reset Logic. When the pixel cell is Empty, by definition, NeedToken becomes a zero. Therefore, when a pixel cell gets control of the bus, TokenOut suddenly becomes free to follow the state of TokenIn, and it goes high.

Like the FastOR Logic, numerous simulations were performed to optimize the circuit. In fact, each transistor in Figure 19 was individually optimized for the IBM process. This makes it less than perfectly optimized for the TSMC process. If the collaboration elects to go with the TSCM process or with a 256 pixel column, this optimization will have to be performed again. Table 5 shows the final transistor sizes.

Table 5: Optimized Token Passing Widths

NAND pfet	Width=6.6μm
	Length=0.8μm
NAND nfet	Width=16.15μm
	Length=0.8um
TokenOut pull-down nfets	Width=3.35μm
	Length=0.8μm
TokenOut pull-up pfet	Width=15μm
	Lendth=0.8μm
Gate nfet	Width=6.55μm
	Lendth=0.8μm
Hold pfet	Width=1.4μm
	Length=0.8μm
Token_Res trace width	1.0μm
Token_In/Token_Out trace width	0.56μm
Read Clock trace width	1.0μm

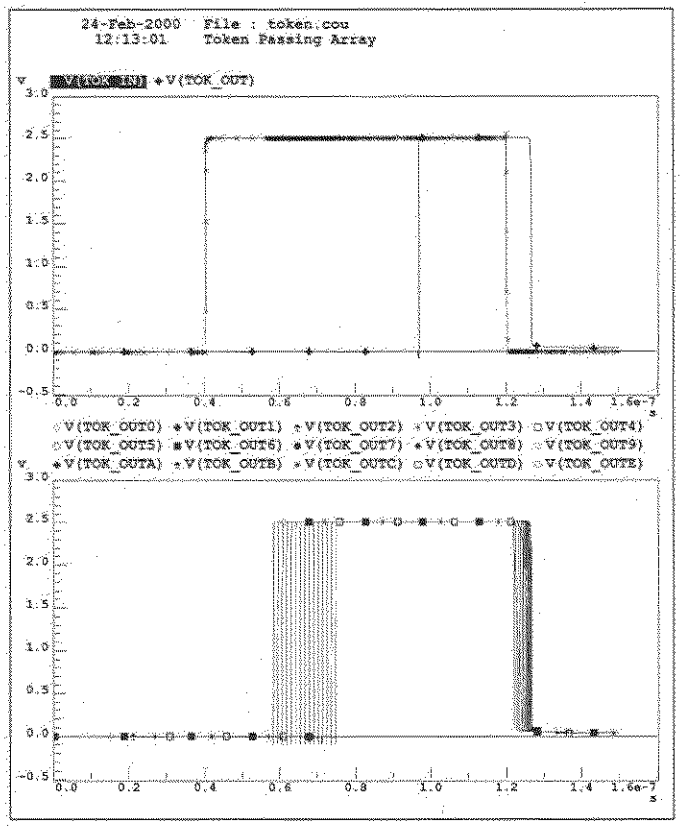


Figure 20: Token Passing Simulations

Figure 20 shows a simulation of an entire column of Token Control Logic Cells interacting as they will in the FPIX chips. Since it is too confusing to show all 160 TokenOut signals at the same time, the lower graph in Figure 20 shows every tenth TokenOut. The upper graph shows the token input to the column, and the final token output from the column. The simulation is a worst case scenario in which pixel0, the pixel cell closest to the End-of-column Logic, and pixel159, the pixel cell furthest from the End-of-column Logic, are both hit simultaneously, and no pixel cells in between them are hit. Under this specific condition – admittedly very rare, the token must be passed through 158 pixel cells between the start of one Read Clock cycle and the start of the next. In the lower graph, the series of rising edges indicate the token skip frequency – the frequency at which the Token Control Logic will pass the token through pixel cells that do not need the Token. At the rising edge of the Read Clock, pixel0 will grab the bus and begin to pass the token. The time between the rising edge of TOK_OUT0 and the rising edge of TOK_OUTE is the time required for the token to skip through 140 pixel cells. This time is 16.5ns, which corresponds to 118ps per pixel cells or a pixel skip frequency of 8.48 GHz. As a side note, if there were 256 pixels in a column, the entire column could be traversed in 30.2ns. With a 160-pixel column, the entire column could be traversed in 18.8ns. For the 256-pixel column, the pixel skip frequency would be a limiting factor for Read Clock frequencies greater than 32.3 MHz. For the 160-pixel column, the pixel skip frequency would be a limiting factor for Read Clock frequencies greater than 53 MHz.

The reset time is simply the difference between the falling edges of the column Token input and the column Token output shown in the upper graph of Figure 20. This time is 6.48ns, which is considerably shorter than any Read Clock frequencies being considered. In fact, it only becomes a limiting factor for Read Clock frequencies in excess of 155 MHz.

5.3.6 ADDRESS LOGIC

The address logic is very straightforward in the pixel cell. Each pixel cell in the column has its own unique combination of nmos and pmos transistors that make up its physical address. All address transistors, whether pull-up (pfet) or pull-down (nfet) are the same physical size. This means that the fall times will be faster than the rise times. However, the physical size and regularity of the address transistors are more important than the relative speeds of the rise or fall times. As long as an address settles to its final value within the read cycle, that is all that matters.

The final design specifications are shown below.

Table 6: Optimized Token Passing Widths

Transistor Width	9.8μm
Trace Width	1μm

5.4 A DETAILED DESCRIPTION OF A HIT FROM THE PIXEL CELL'S PERSPECTIVE

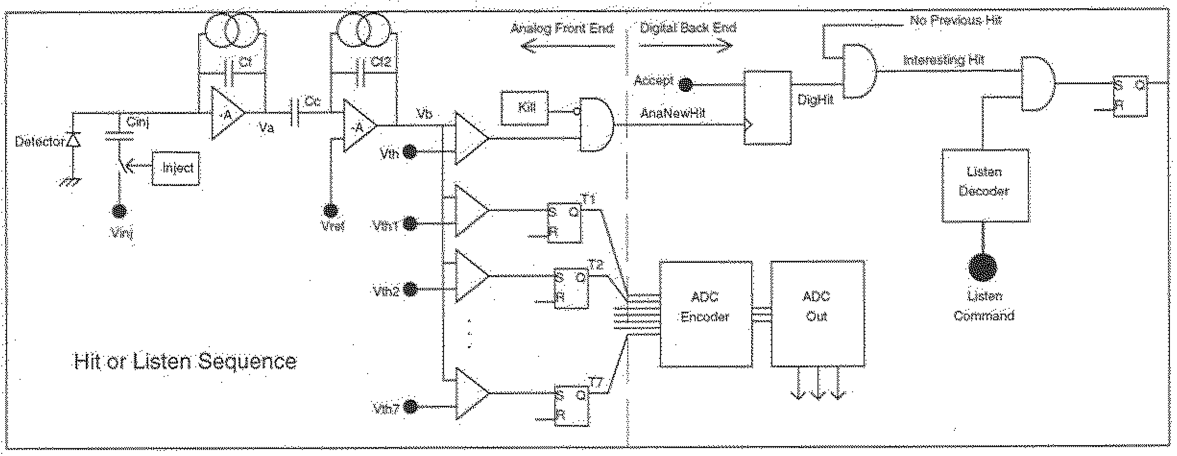


Figure 21: A simple schematic of the Hit Sequence

When a particle passes through the Detector, a pulse of electrons is injected into a preamplifier. The integral of that pulse of electrons is, to first order, the amount of charge deposited on the detector by the particle. This charge pulse is amplified by

$$Va = -\frac{Q}{C_f}$$

This voltage is amplified with respect to Vref (set external to the pixel cell).

$$Vb = -\frac{C_c}{C_{f2}}(Va - Vref) + Vref$$

A second means of generating hit data is to program the Inject logic to inject a charge into the preamplifier on a falling edge of Vinj. The effect on Vb will be the same as if a real hit was seen by the Detector.

Regardless of how the charge was injected, Vb is simultaneously compared to eight different voltages by eight different comparators. Seven of these Comparators comprise a flash analog-to-digital converter which produces a 7-bit thermometer code. These signals are latched by Set-Reset flip-flops which will hold the thermometer code indefinitely. Note that this use of S-R flip-flops forms a natural peak detector. The seven outputs of the S-R flip-flops, T1-T7, are input to the ADC Encoder which converts them into a 3-bit binary number. This number is passed to the ADC Output Drivers. Note also that the ADC Encoder and the ADC Output Drivers are purely combinatorial. The memory of the hit magnitude is in the seven S-R flip-flops that latch the thermometer code.

The eighth Comparator is the main hit comparator. If Vb exceeds Vth, then a hit has occurred. If the user has programmed this pixel cell to be killed, then the hit data will be blocked by an AND gate and the digital back end will never know a hit occurred. Consequently, the End-of-column Logic and the rest of the chip will never know a hit has occurred. If the user has not programmed this pixel cell to be killed, then the hit data is passed to the Digital Back End of the pixel cell as the signal AnaNewHit.

A rising edge on AnaNewHit will pass the value of the Accept signal onto the rest of the back end as the signal DigHit. Only a rising edge on AnaNewHit has any effect. Therefore, the pixel cell cannot be adversely effected by pulse duration in the Analog Front End. If Accept is a one, DigHit becomes a one. If Accept is a zero, DigHit remains a zero, and the rest of the cell will never know a hit occurred.

DigHit is ANDed with a signal that is a one if there are no previous hits contained in the pixel cell and a zero if there are previous hits stored in the pixel cell. If there are no previous hits, then this new hit is an interesting hit, and InterestingHit becomes a one. If there were previous hits, then this new hit is not interesting, and InterestingHit will remain a zero and the rest of the pixel cell will never know a hit occurred.

If an End-of-column Command Register is issuing a Listen Command, then a Listen Decoder will output a one, and the hit will be recorded onto an S-R flip-flop. The Previous Hit signal will now indicate that there is a recorded hit in the pixel cell, blocking any subsequent hits from affecting the pixel cell.

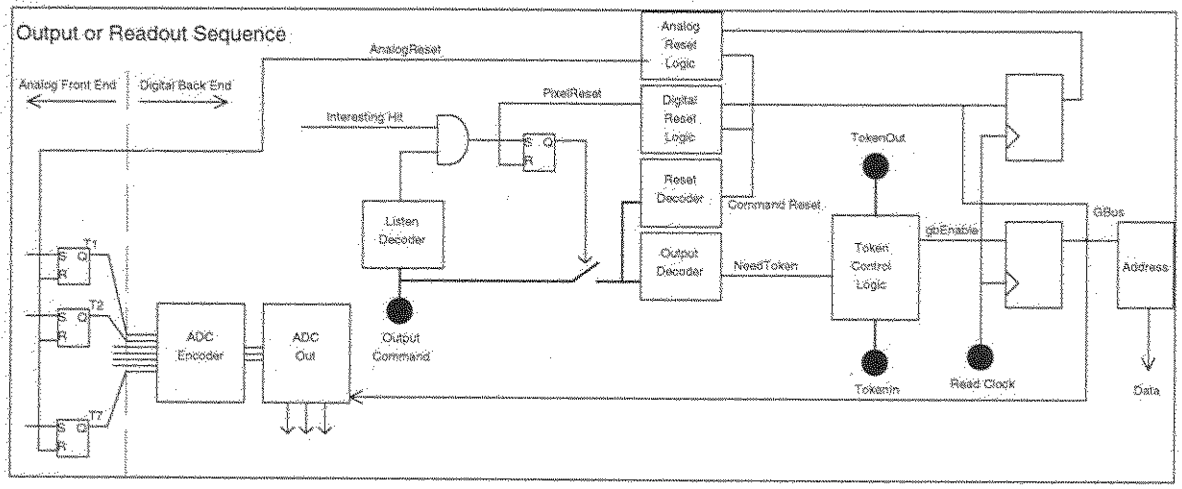


Figure 22: A simple schematic of the Readout Sequence

After the hit has been recorded, the output S-R flip-flop that stored the hit closes a switch connecting the commands of the End-of-column register with the Reset and Output Decoders in the pixel cell. When the Output Command is issued, NeedToken is activated and the pixel cell waits for the column token to arrive. Note that the pixel cell could not respond to the Output Command unless it had already received a hit because in the absence of a hit, the Output Decoder is disconnected from all Command Lines.

When the Token arrives, the pixel cell waits for the next rising edge of the Read Clock before it activates Gbus. This causes the Digital Reset Logic to reset the S-R flip flop that had stored the hit, making the pixel cell Empty again. This releases the Token to the next hit pixel. Gbus also causes the address data to be driven onto the address bus and the ADC data to be driven by the ADC Output drivers.

At the next rising edge of the Read Clock, Gbus is latched by a second flip-flop, the output of which activates the Analog Reset Logic that resets the Thermometer code in the Analog Front End.

If a Reset Command had been issued instead of the Output Command, the Reset Decoder would have activated the CommandReset line which would have simultaneously reset both the thermometer code in the Analog Front End and the S-R flip-flop that recorded the hit in the Digital Back end.

6 THE END-OF-COLUMN LOGIC

6.1 SIGNALS

Outputs from an End-of-column Logic to the pixel cells in its column.

1	COMA<1:0>	Output. Command State Machine A command pair
2	COMB<1:0>	Output. Command State Machine B command pair
3	COMC<1:0>	Output. Command State Machine C command pair
4	COMD<1:0>	Output. Command State Machine D command pair
5	ACCEPT	Output. Controls the Hit Data to Recorded Data conversion for all pixels in the column. If ACCEPT=1, the End-of-column Logic is ordering pixel cells to accept new hits. If ACCEPT=0 the End-of-column Logic is ordering pixel cell to ignore new hits
6	COLTOKENIN	Output. The Column Token for arbitrating bus access
7	COLREADCLK COLREADCLKB	Output. The Read Clock released by the End-of-column Logic to the pixel cells in the column. It is only released when the End-of-column is "Talking" otherwise, it is held at zero.
8	DATARESET	Output. The Master Reset relayed from the chip input pads, through the End-of-column Logic and to all pixel cells. Will cause a reset of all pixel cell digital back ends and analog front ends.
9	READRESET	Output. A Reset specifically for the two flip-flops in each pixel cell's Token Control Logic. This Reset is activated when the End-of-column Logic has "Nothing to Say"

Outputs from an End-of-column Logic to the Core Logic.

10	HTOKOUT	Output. Horizontal Token Output for column-to-column bus arbitration
11	HAVEDATA	Output. Indication of the presence of data to be output from the column
12	COLDATA<7:0>	Output. Row address.
13	COLDATA<12:8>	Output. Column address.
14	COLDATA<20:13>	Output. BCO Number
15	COLDATA<23:21>	Output. ADC Magnitude

Inputs to an End-of-column Logic from the Pixel Cell.

16	HFASTOR	Input. Hit Fast OR indicator of the presence of a hit somewhere in the column
17	RFASTOR	Input. Read Fast OR indicator of the presence of data to be output somewhere in the column. When this signal goes from active to inactive, the End-of-column Logic knows that the column is done Outputting data
18	PIXDATA<7:0>	Input. Row address.
19	PIXDATA<10:8>	Input. ADC Magnitude.
20	COLTOKENOUT	Input. The Token Out of the highest pixel cell's Token Control Logic. When this activates, the token has passed through all of the pixels. This signal is used as a diagnostic. If COLTOKENOUT activates and the RFASTOR still has not gone inactive, then something is wrong.
21	COLDATA<23:21>	Output. ADC Magnitude

Inputs to an End-of-column Logic from the Core Logic.

22	BCO<7:0>	Input. The Beam Cross-over Number; indicator of time.
23	HTOKIN	Input. Horizontal Token Input for column-to-column bus arbitration
24	BCOCLK_IN BCOCLKB_IN	Input. Beam Cross-over clock.
25	READCLK_IN READCLKB_IN	Input. Read Clock
26	CHIPSENDDATA	Input. When active (1) a "Talking" End-of-column Logic can continue to change its data at every rising edge of the Read Clk. When inactive (0) a "Talking" End-of-column Logic must latch the data being sent and not change it until CHIPSENDDATA goes active.
27	MASTERREJECT	Input. When high (1), the ACCEPT signal output to the pixel cells must be low (0). When low (0), the ACCEPT signal output to the pixel cells will be high (1) unless all four End-of-column registers are full. MasterReject is a system-wide throttle.
28	DATARESET_IN	Input. The Master Reset directly from the chip pads
29	CORESILENT	Input. When high (1), the Core is not outputting data. When low (0), the Core is outputting data.

6.2 OVERVIEW

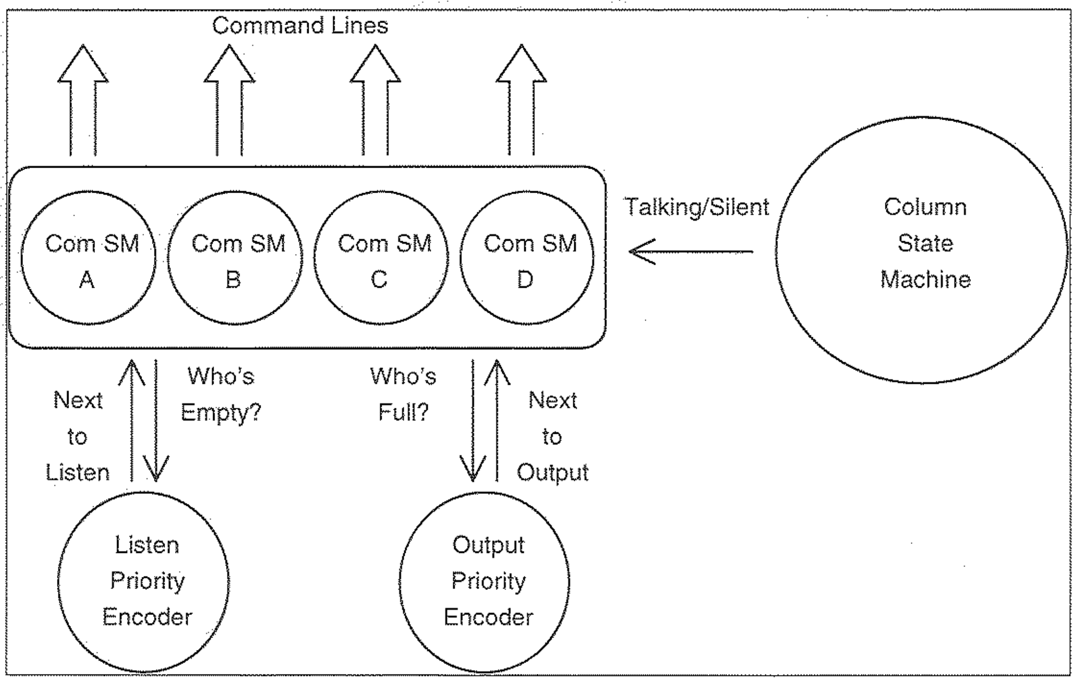


Figure 23: A Schematic Overview of the End-of-column Logic

The majority of the intelligence in the FPIX Core is located in the End-of-column Logic. It needs to understand what commands to issue to the pixel cells and when to issue those commands. It must also understand time with respect to the BCO clock, the Read Clock and the BCO number.

Each End-of-column cell consists of

1. four Command State Machines which issue commands to pixel cells via the Command Lines,

2. four Registers which are paired one-for-one with Command State Machines and which store BCO numbers when necessary,

3. one Column State Machine which controls the End-of-column Logic in general,

4. one Hit Priority Encoder which determines which State Machine/Register pair is the next to "listen" for hits

5. one Output Priority Encoder which determines which State Machine/Register pair is the next to output data., and

6. necessary support logic.

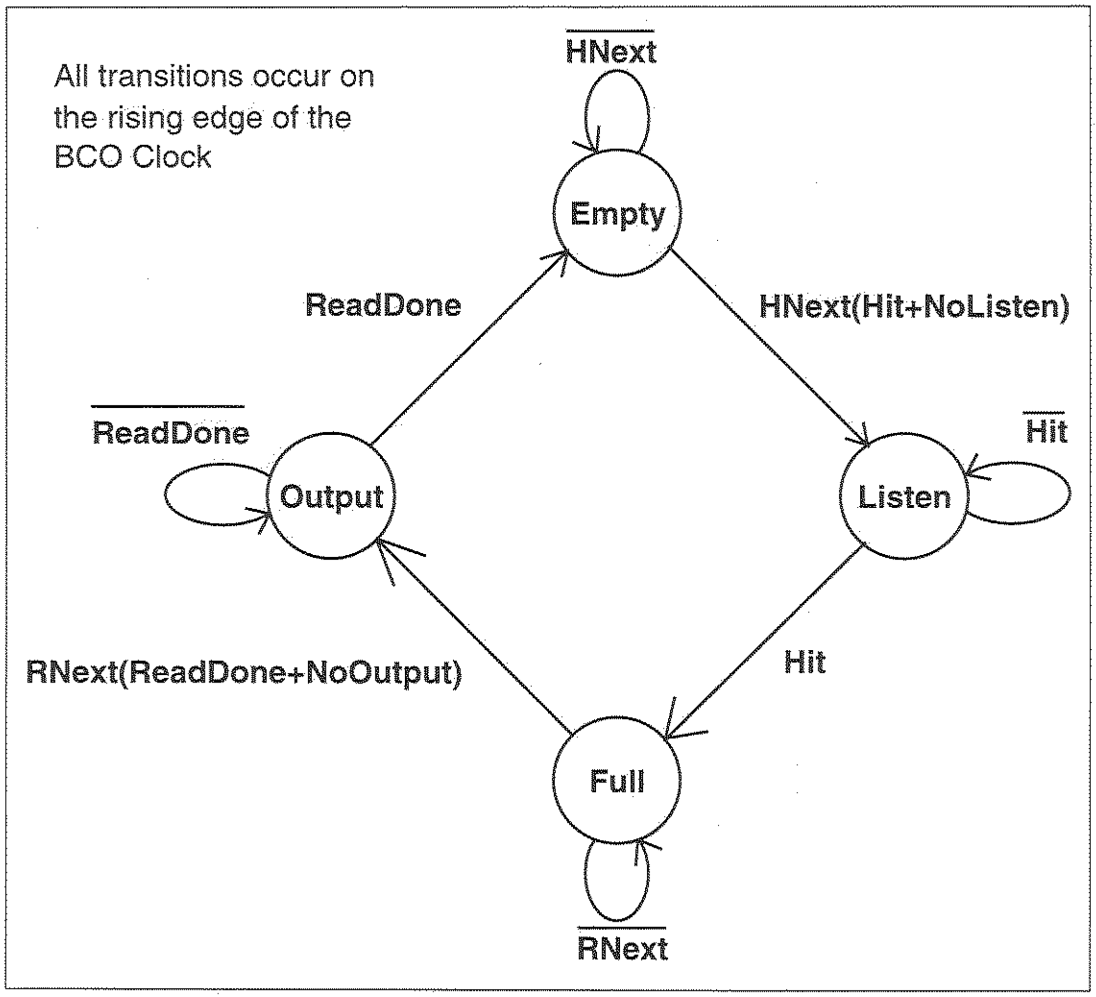
All of these subcircuits are considerably different from their counterparts in FPIX1. Figure 23 shows the interactions between the major components of the End-of-column Logic.

6.3 THE COMMAND STATE MACHINES

The four Command State Machines operate independently of one another and their primary purpose is to generate the four Command Line pairs. Each is a Mealy state machine that changes with each rising edge of the BCO clock and that has four states:

1. Empty - No hit has been received and not listening for hits
2. Listen - No hit has been received, but listening for one
3. Full - A hit has been received, but not outputting yet
4. Output - A hit has been received and the data is being output

The states flow as shown in Figure 24.



All transitions occur on the rising edge of the BCO Clock
$\overline{\text{HNext}}$
Empty
ReadDone
HNext(Hit+NoListen)
$\overline{\text{ReadDone}}$
Output
Listen
$\overline{\text{Hit}}$
RNext(ReadDone+NoOutput)
Hit
Full
$\overline{\text{RNext}}$

Figure 24: The Command State Machine state diagram

At any given time, only one Command SM can be allowed to be in the Listen state. If there is a hit somewhere in the column, the pixel cells that are hit are going to try to associate themselves with whichever Command SM is issuing the Listen Command. If more than one Command SM is issuing a Listen Command, then any hit pixel cells are going to try to obey more than one Command SM. The results could be unpredictable.

Moreover, there *must* be a Command SM in the Listen state unless all four Command SMs are full. Recall from the description of the pixel cells that they only pull down the HFastOR line if they get a hit when a Command Line is issuing the Listen Command. If no Command SMs are in the Listen state, then no one will be issuing a Listen Command and no pixel cells will pull down the HFastOR line. The result is a hung chip.

This problem is solved by the Hit Priority Encoder and the HNext and NoListen signals that it generates. There is one NoListen signal for all four Command SMs. When it is active (1), then no Command SMs are in the Listen state. There is a unique HNext signal for each Command SM and it is the job of the Hit Priority Encoder to make sure that a maximum of one of these signals is active at any time. When HNext is active for a particular Command SM, then that Command SM will be the next SM to make the transition to the Listen state. The function of these signals and the state machines is best described by example.

1. If Command SMa is in the Empty state and has an active HNext signal and no Command SMs are in the Listen state, then NoListen will be active. On the next rising edge of the BCO clock, Command SMa will make the transition to the Listen State.

2. If Command SMa is in the Listen state and Command SMb is in the Empty state and has an active HNext signal, NoListen will be inactive. At the next rising edge of the BCO clock *after* there has been a hit somewhere in the column, Command SMa will make the transition to the Full state and Command SMb will make the transition to the Listen state.

Similarly, at any given time there can be only one Command SM in the Output state. If more than one SM were allowed into the Output state at a given time, then information from more than one time slice would be output at the same time.

As in the case with the Listen state, these problems associated with the Output state are solved by the Output Priority Encoder and the Rnext and NoOutput signals that it generates. There is one NoOutput signal for all four Command SMs. When it is active (1), then no Command SMs are in the Output state. There is a unique RNext signal for each Command SM and it is the job of the Output Priority Encoder to make sure that a maximum of one of these signals is active at any time. When RNext is active for a particular Command SM, then that Command SM will be the next SM to make the transition to the Output state. The function of these signals and the state machine is again best described by example.

3. If Command SMa is in the Full state and has an active RNext signal and no Command SMs are in the Output state, then NoOutput will be active. On the next rising edge of the BCO clock, Command SMa will make the transition to the Output State.

4. If Command SMa is in the Output state and Command SMb is in the Full state and has an active RNext signal, NoOutput will be inactive. At the next rising edge of the BCO clock *after* the column is done reading out, Command SMa will make the final transition back to the Empty state and Command SMb will make the transition to the Output state.

In the above four examples, two signals are left unexplained. A Command SM receives information about hits via the HFastOR circuitry, which will be described later. Since hit arrival is virtually synchronous with the BCO clock, no further conditioning of the HFastOR signal is necessary. It might be logical to assume that the Command SM receives information about the conclusion of the read cycle directly from the RFastOR circuitry. However, since read out is performed synchronous with the Read Clock and not the BCO clock, an additional conditioning step is necessary to ensure stability. This circuitry is shown in Figure 25. The first part of this conditioning is performed by the Column State Machine, which operates at the Read Clock frequency and which accepts the output of the RFastOR circuitry and generates the colSilent signal (column silent). The edge triggered d-flip-flop in the figure ensures that only the rising edge of colSilent affects Done and NotDone. This guarantees that only the completion of the present read out will activate the Done signal. The input to the edge-triggered d-flip-flop is the Output state signal. This guarantees that Done will only be activated in the SM in the Output state. An S-R flip-flop actually creates the Done and NotDone signals. It is reset to NotDone whenever the Command SM is in either the Empty state or the Listen state. The Empty state also resets the edge-triggered d-flip-flop. The S-R flip-flop is set and Done is activated only at the rising edge of the colSilent signal if the Command SM is presently NotDone. This two-step process guarantees signal stability in spite of the fact that the Command SMs operate on the BCO clock and readout occurs on a different clock.

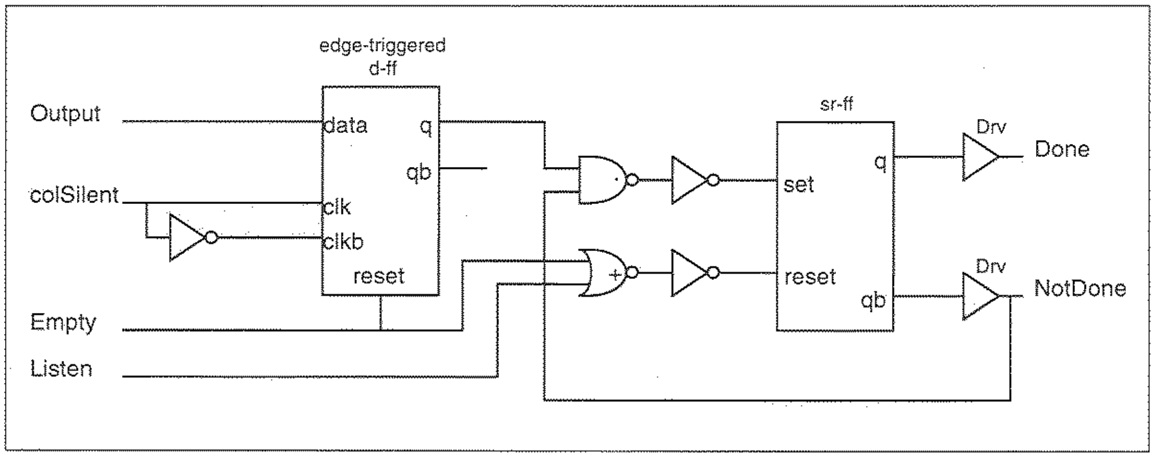


Figure 25: Conditioning Circuitry for Read Done

Finally, the actual commands sent up the Command Lines are generated from the Empty, Listen, Full and Output states and the Done and NotDone signals.

$$Com0 = Listen + Output \cdot Done$$
$$Com1 = Listen + Output \cdot NotDone$$
Equation 3

When in the Listen state, Com0 and Com1 will both be high, and 11 is the Listen Command. When in either the Empty or the Full state, Com0 and Com1 will both be low, and 00 is the Idle Command. When in the Output state before the readout is done, Com0 will be a 1 and Com1 will be a zero, and 10 is the Output Command. Finally, when in the Output state *after* the readout is done, Com0 will be a 0 and Com1 will be a 1, and 01 is the Reset Command. This last feature was added to ensure that if there was any communication difficulty with the column token, then at least the column could be made to function *up to* the point of the communication difficulty. If any pixel cells remained unread after the End-of-column Logic thought it was done, then those pixel cells would be reset and they would not interfere with further operation of the column.

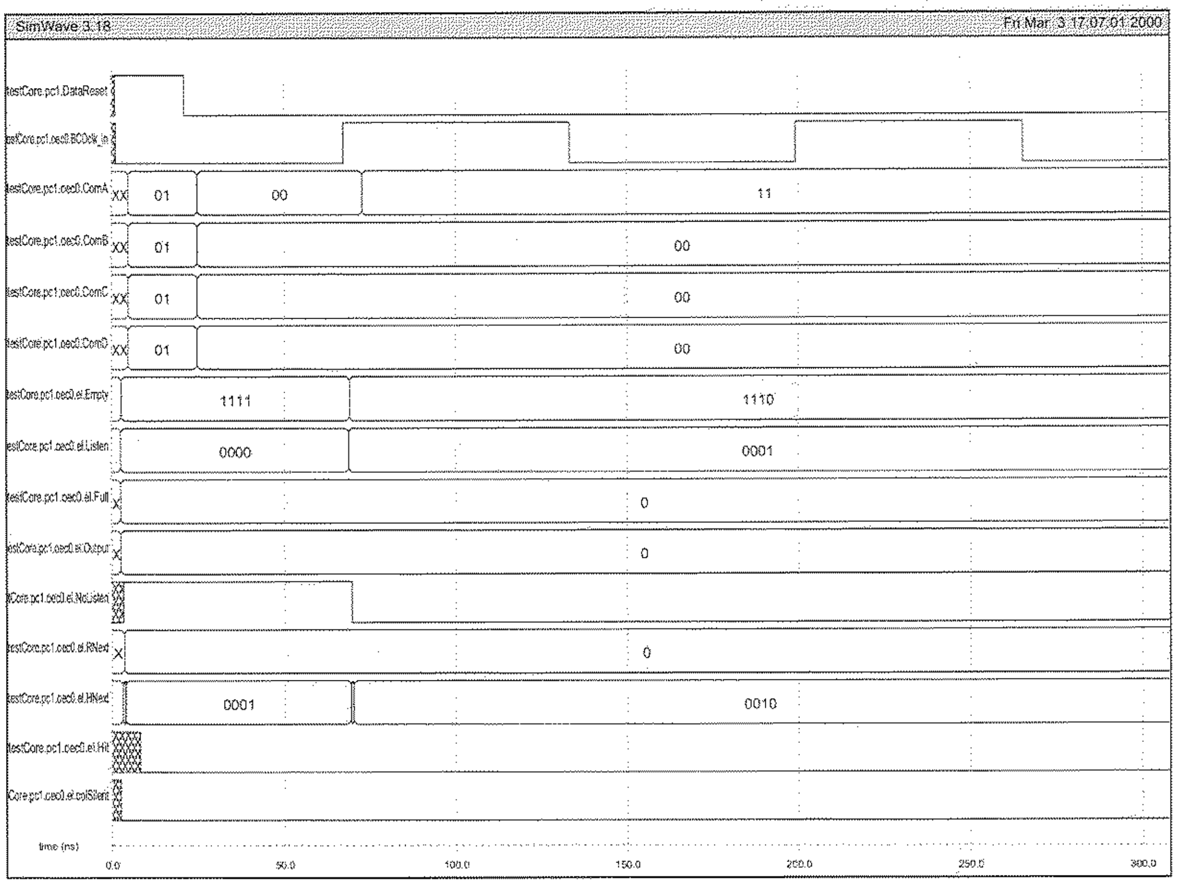


Figure 26: End-of-column Logic at Reset. The signals are, from top to bottom, Master Reset, the BCO clock, the four pairs of Command Lines, Empty state indicators, Listen state indicators, Full state indicators, Output state indicators, the NoListen signal, the RNext signal, the HNext signal, the Hit indicator, and the column Silent signal. The state indicators decode the state of each of the four Command State Machines and indicate (with a 1) when a state machine is in a certain state.

Figure 26 shows an End-of-column Logic cell during a Master reset, and, in particular, shows those signals important to the Command SMs. First, during the actual reset (in the first 20 ns of the graph), all four Command Lines are driven to the Reset Command (01). This is accomplished not in the state machines themselves, but rather in "override" logic attached to the drivers of the Command Lines. In this fashion, regardless of the state of the Command SM, the Reset Command will be driven up the column during a Master Reset. Note that after the completion of the Master Reset, all four Command Lines are showing the Idle Command (00); the Empty state indicator is showing that all four Command SMs are in the Empty state (1111); the Listen state indicator is showing that no Command SM is in the Listen state (0000); as expected, the NoListen signal is active (1) because no Command SM is in the Listen state; and, finally, the HNext signal indicates that Command SMa has been selected as the next state machine to move to the Listen state (0001). At the next rising edge of the BCO clock, Command Line A changes to the Listen Command (11); the Empty state indicator changes to 1110, indicating that Command SMa is no longer in the Empty state; the Listen state indicator changes to 0001 indicating that Command SMa is now in the Listen state; NoListen goes inactive; and HNext indicates that Command SMb has been selected as the next state machine to move to the Listen state. Finally, note that at the next rising edge of the BCO clock, everything remains the same because nothing has happened that would precipitate a state change in any of the Command SMs.

Figure 27 shows a single Command SM during a hit cycle. All of the signals in the figure are connected directly to that Command SM. At first, the SM is in the Listen state as indicated by the Listen state indicator and by the presence of the Listen Command (11) at the Command Line pair. At approximately 800ns, there is a hit. At the next rising edge of the BCO clock, the SM makes the transition to the Full state. The Command Line changes to the Idle Command (00). The Output Priority Encoder selects the SM to be the next to output by activating the RNext signal at approximately 850ns. At the next rising edge of the BCO clock (almost 1000ns), the SM makes the transition to the Output state where it remains until the rising edge of the BCO clock after the arrival of the colSilent signal at approximately 1200ns. During the Output state, the Command Line pair issues first the Output Command (10), and then the Reset Command (01). Finally, at approximately 1250ns, the SM makes the transition back to the Empty state. Note that the Listen Priority Encoder selects this SM (via the HNext signal) to be the next SM to make the transition to the Listen state. This is coincidental.

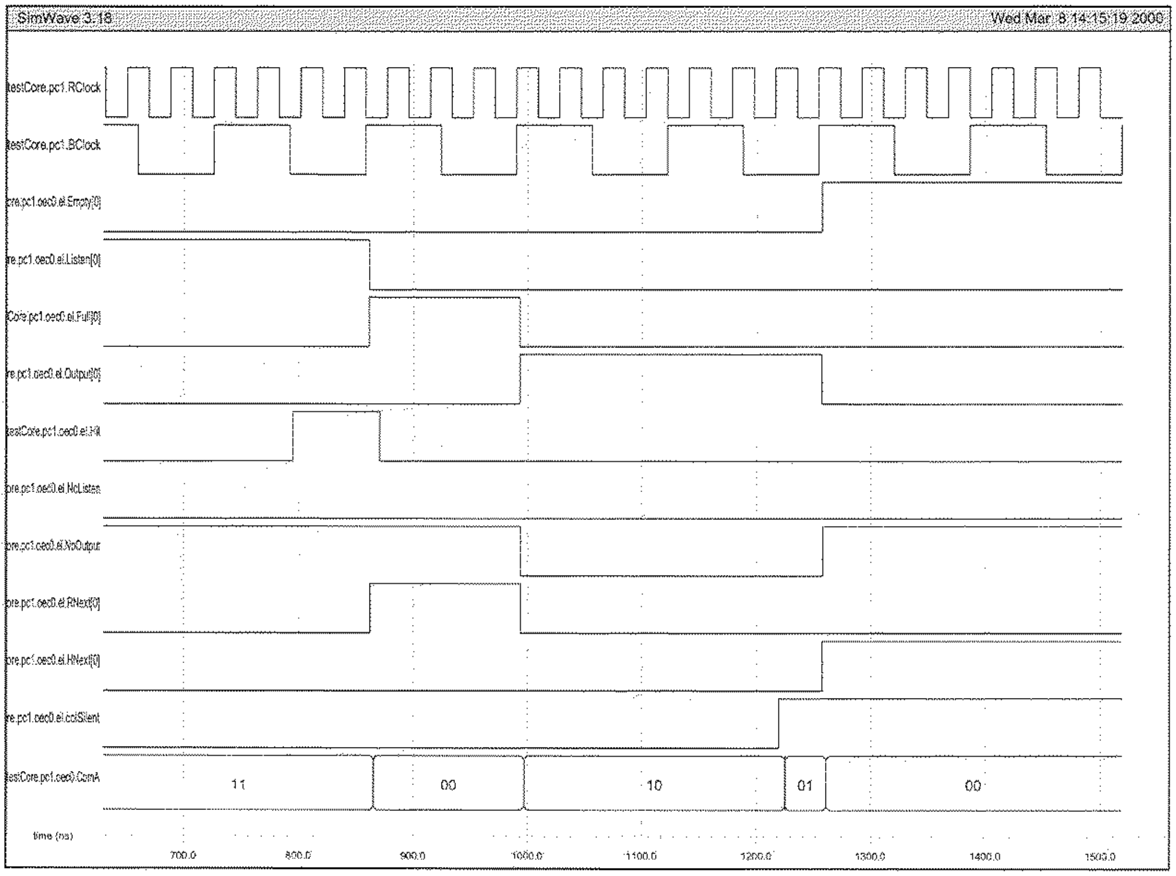


Figure 27: One Command SM in the End-of-column Logic during a hit cycle. From top to bottom the signals are the Read Clock, the BCO clock, the Empty state indicator, the Listen state indicator, the Full state indicator, the Output state indicator, the Hit signal, the NoListen signal, the NoOutput signal, the RNext signal, the HNext signal, the column silent signal, and the Command Line pair.

6.4 THE COLUMN STATE MACHINE

Each End-of-column Logic cell has one Column State Machine that, in short, controls the read out of the column. It changes its state, if necessary, only on the rising edge of the Read Clock and it has four states:

1.	Nothing to Say	-	No pixels need to be read out at this time
2.	Something to Say	-	Pixels are waiting to be read out
3.	Talking	-	Pixels are being read out
4.	Silent	-	The column is done reading out this event

The states flow as shown in Figure 28.

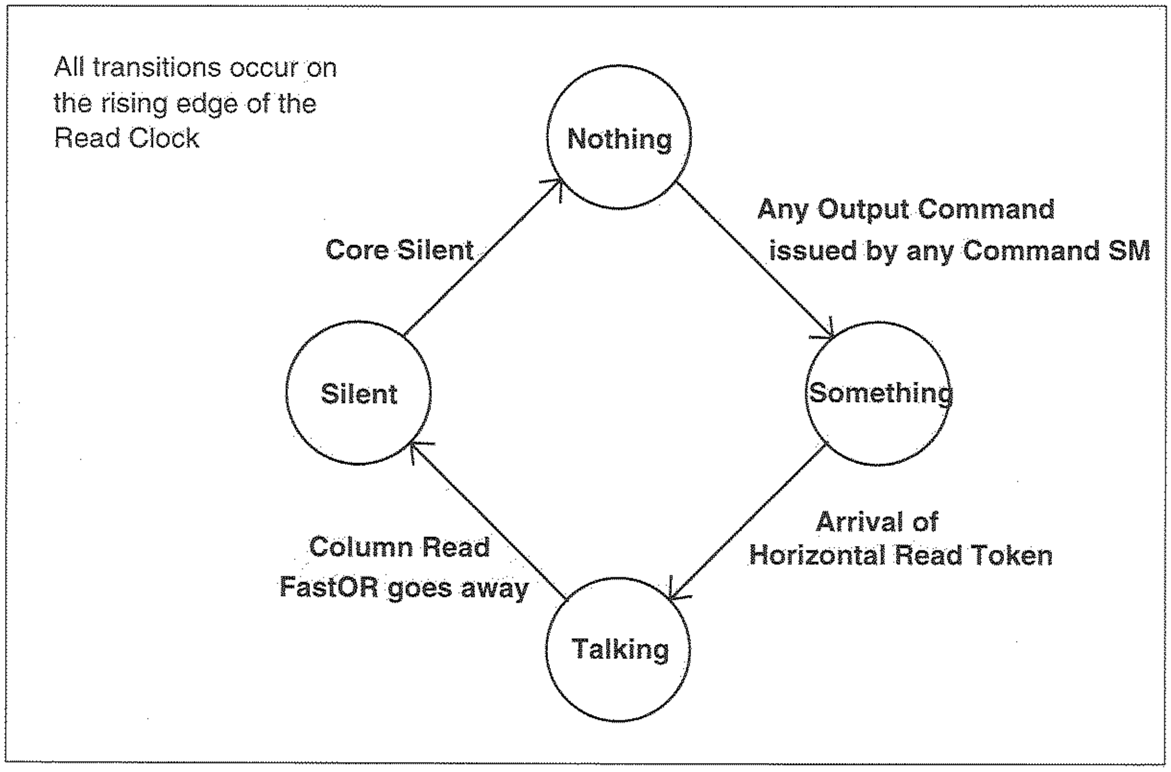

All transitions occur on the rising edge of the Read Clock
Nothing
Any Output Command issued by any Command SM
Something
Arrival of Horizontal Read Token
Talking
Column Read FastOR goes away
Silent
Core Silent

Figure 28: The Column State Machine State Diagram

6.4.1 NOTHING TO SAY

Upon Reset, the Column State Machine is forced into the "Nothing to Say" state. This is only logical since, after a Reset, the column cannot possibly need to output data. The SM will remain in the "Nothing to Say" state until any one of the four Command State Machines issues an Output Command (10). At the first rising edge of the Read Clock after receiving an Output Command, the Column SM will make the transition to the "Something to Say" state.

Waiting until there is an Output Command has some important implications. First, this implies that the some Command SM had been in the Listen State; some pixel had received hit data; the Command SM moved into the Full State and finally into the Output State. Second, the same circuitry that decodes an Output Command in each pixel cell is also used to decode the Output Commands for the Column SM. Furthermore, the Column SM decodes the actual command lines that are sent up the column. Therefore, by the time the Column SM is aware that some Command SM is issuing the Output Command, all pixels associated to that Command SM are also aware that it is issuing the Output Command. This eliminates almost all timing-sensitive problems associated with read out. For example, the Output Command must arrive before the Column Token. If the Column Token arrives first, then the pixel cell will not yet realize that it needs to grab the token and it will therefore pass it on. This can lead to more than one pixel driving the bus at the same time.

During the "Nothing to Say" state, the Read Clock is blocked from the column via the Clock Control Logic which will be explained hereafter.

6.4.2 SOMETHING TO SAY

After the Column SM has received an Output Command and has made the transition to the "Something to Say" state, clearly there are pixels in the column that need to be read out. However, the Core is not ready for this column to read out yet. The Core needs to arbitrate among all of the columns that have "Something to Say". Like the column which arbitrates via the Column Token, the Core arbitrates via the Horizontal Token (Htok). The first rising edge of the Read Clock after the Column State Machine receives the Htok, it makes the transition into the Talking state.

During the "Something to Say" state, the Read Clock is still blocked from the column via the Clock Control Logic. CoreHasData, a diagnostic signal that indicates when there is data to output, is activated during this phase. CoreHasData is a distributed OR similar to the HFastOR and it can be activated by any End-of-column Logic. Finally, and most importantly, the Column Token is sent up the column in this phase. Again, this guarantees that the Output Command has been sent to the pixel *before* the token gets there. Secondly, this makes sure that in each column the first pixel with output data has the token before the Read Clocks are released into the column.

6.4.3 TALKING

After the Column SM has received the Horizontal Token and makes the transition to the Talking state, it is free to output its data. It will remain in this state until the RFastOR circuitry indicates that the last pixel is outputting its data. The first rising edge of the Read Clock after the rising edge of the RFastOR line, the Column SM will make the transformation to the Silent state.

During the Talking state, the Read Clock is finally transmitted up the column via the Clock Control Logic. CoreHasData, the diagnostic signal that indicates when there is data to output, is still active during this phase. Tri-state buffers are enabled in this state connecting the column bus to the Core output bus.

6.4.4 SILENT

After a Talking Column SM has seen the rising edge of the RFastOR and makes the transition to the Silent state, it no longer has data to output, but other columns in the Core may still be outputting. Therefore, the Column SM remains in the Silent state until it receives the coreSilent signal indicating that the whole Core is done outputting. This helps prevent a hot column from grabbing the Core Output bus again and again while other columns are trying to output. When the coreSilent signal is received, the Column SM will make the transition back to the "Nothing to Say" state after the next rising edge of the Read Clock.

During the Silent state, the Read Clock is still transmitted up the column via the Clock Control Logic. This ensures that the last pixel cell to output receives enough clock edges to reset both the digital back end and the ADC latches. The column token is reset in this phase to prepare for the next read out cycle. The colSilent signal is issued to the Command SM so they can make their necessary transition between their Output states and their Empty states. Finally, the tri-state buffers connecting the column bus to the Core output bus are disconnected.

6.4.5 SIMULATIONS

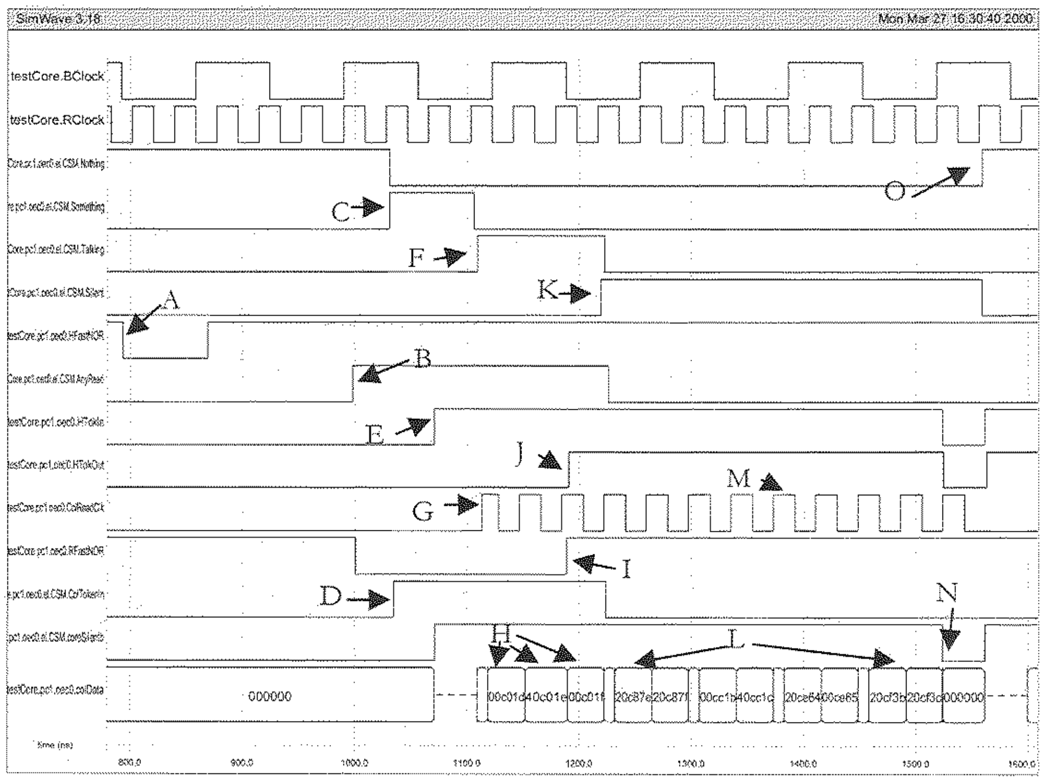


Figure 29: Verilog simulation of a single Column State Machine.

Figure 29 shows a single Column SM as it progresses through an entire read out cycle. At approximately 800ns (A), the Column SM is in the "Nothing to Say" state and a hit occurs somewhere in the column. This has no effect on the state of the Column SM. Also, though the Read Clock is free running, it does not reach the pixels in the column. At approximately 1000ns (B), the AnyRead signal activates indicating that some Command SM is issuing the Output Command. At the first rising edge of the Read clock after the arrival of the AnyRead signal, the Column SM makes the transition to the "Something to Say" state (C). In this state, the column Token is issued (D). After the arrival of the Horizontal Token (E), the Column SM makes the transition to the Talking state (F). Once in the Talking state, the Read Clock is released to the column (G), and for each rising edge of the Read Clock, one pixel of information is driven to the Core Output bus (H) until the rising edge of the RFastNOR (I) which indicates that this column is outputting its last pixel. At this point the Horizontal Token is released (J) and on the next rising edge of the Read Clock, the Column SM makes the transition to the Silent state (K). While in the Silent state, other columns continue to drive the Core Output bus (L) and the Read Clock remains active in the Silent column (M). Finally, the coreSilent signal activates (N) (shown here inverted) and at the next rising edge of the Read Clock, the Column SM makes the transition back to the "Nothing to Say" state (O).

6.4.6 HORIZONTAL TOKEN PASSING LOGIC

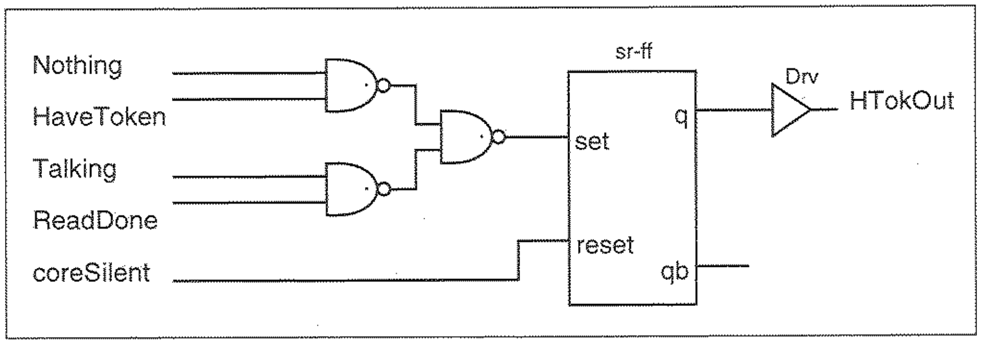


Figure 30: Horizontal Token Passing Logic

The Horizontal Token Passing Logic in the Column SM is complicated enough to warrant further explanation. It is shown in Figure 30. The token itself is originally generated by the Core when the Core makes its own transition from coreSilent to coreTalking. It would be possible to have a completely combinatorial horizontal token pass. However, it is critical to the read out speed of the Core that the horizontal token resets uniformly across the Core. Moreover, since the horizontal token pass is so critical to proper operation of the Core and in the interest of single event upset (SEU) tolerance, it is important that one SEU in the Core Logic cannot hang the chip by destroying the horizontal token. Therefore, the horizontal token passes through a series of SR flip-flops, one per column. These are all reset at once when coreSilent activates.

If a column has "Nothing to Say" and it has the token, it immediately sets the SR flip-flop and passes the token. If it has "Something to Say", then when it has the token, HtokOut will not be set. Instead, the Column SM will make the transition to the Talking state and then, when the RFastOR logic activates the ReadDone signal, the Horizontal Token Passing Logic will set the SR flip-flop and pass the token. Note, ReadDone activates *while* the End-of-column Logic is outputting the last pixel. Therefore, the next column with "Something to Say" will get the Horizontal Token and be ready to output by the next rising edge of the Read Clock. This is demonstrated in Figure 29 (L).

Another important note regards the Column SM "having the token". The signal "HaveToken" in Figure 30 is not simply the HtokIn signal in Figure 29. Instead,

$$HaveToken = HTokIn \bullet \overline{HTokOut} \qquad \text{Equation 4}$$

In other words, a column has the token when it has received the token from the previous column but has not yet passed the token to the next column. If a column has passed the token on, then it does not have the token any more.

The Horizontal Token Passing Logic actually poses a minor problem for the FPIX Core. It is a limiting factor in the readout speed. By simulation, if there is a hit in the 17th column, it takes more than one Read Clock cycle for the horizontal token to reach the 17th column. This is because the skip frequency of the Horizontal Token Passing Logic is only 22.2 MHz.

Additional simulations have revealed that the majority of the problem is in the driver shown in Figure 30. If that driver is eliminated and replaced with a properly sized inverter of qb, then the skip frequency is increased to 34.4 MHz.

6.4.7 CLOCK CONTROL LOGIC

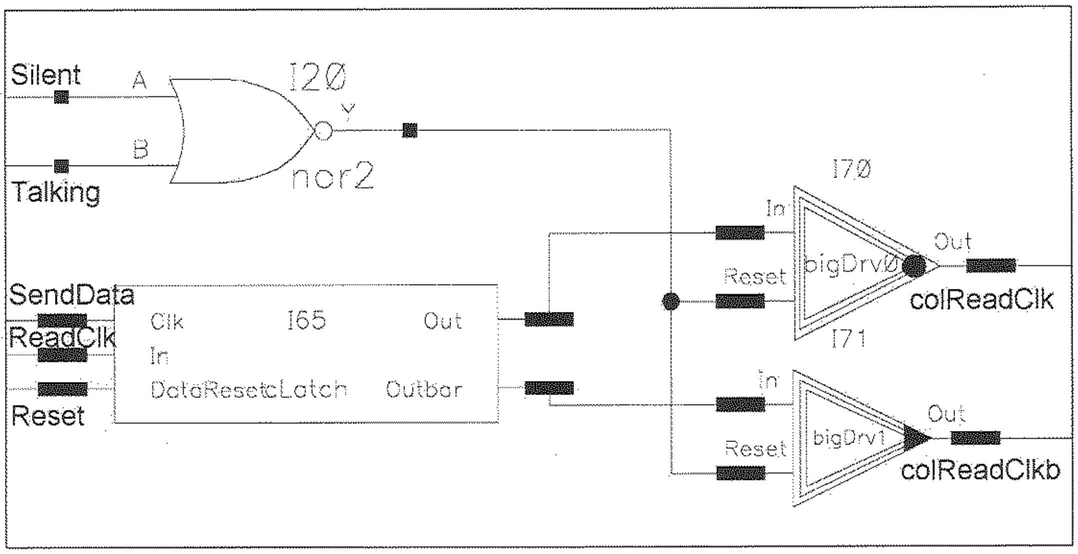


Figure 31: The clock control logic

The Clock Control Logic needs to do a number of things.

1. When the Column SM is in either the "Nothing to Say" or "Something to Say" states, the Read Clock must be blocked from the column. Moreover, the column Read Clock must be a zero during these states.

2. When the Column SM is in either the Talking or Silent states, the Read Clock must be passed to the column.

3. When ChipSendData is inactive (low), the Clock Control Logic needs to hold the clock at its present state regardless of whether that state is a one or a zero.

4. The Clock Control Logic must be resettable.

5. It must be capable of driving the entire column in a timely fashion.

These functions are accomplished in several steps as shown in Figure 31. The two drivers, bigDrv1 and bigDrv0, have enough strength to drive the column's line capacitance. Moreover, when either Talking or Silent are active, bigDrv0 drives the colReadClk to a zero and bigDrv1 drivers the colReadClkb to a one.

The circuit cLatch converts ReadClk to a differential CMOS signal when SendData is active. When SendData is inactive, cLatch holds the last state of the ReadClk. Finally, at Reset Out and Outbar are set to one and zero, respectively.

6.5 THE END-OF-COLUMN REGISTERS

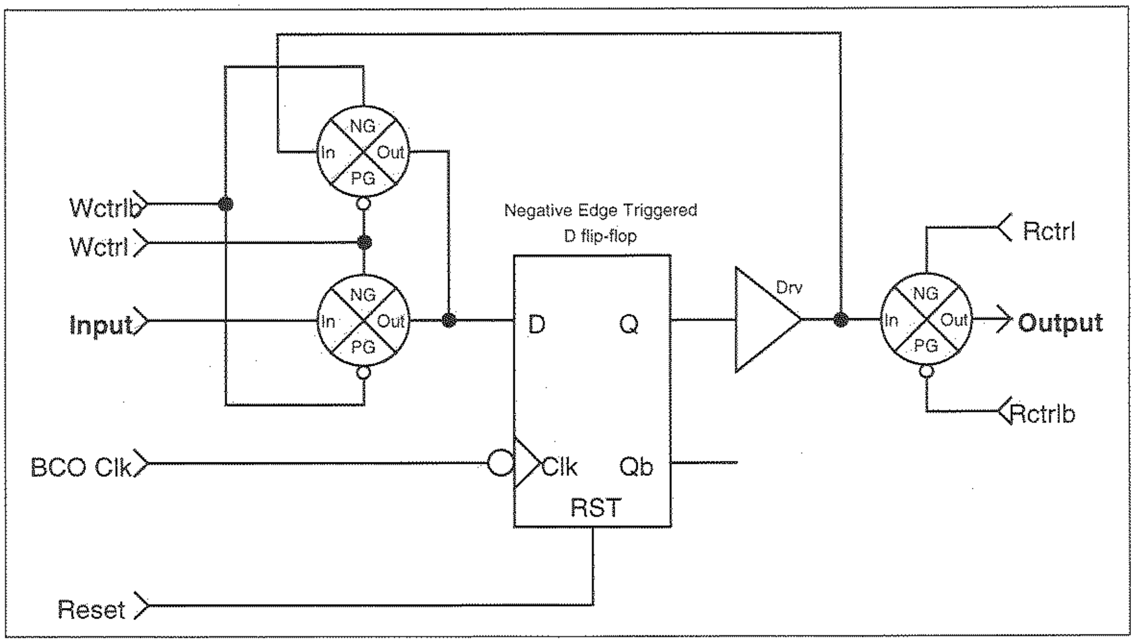


Figure 32: A single bit of an End-of-column Register

There is one eight bit End-of-column Register for each End-of-column Command State Machine. When a particular Command SM is in the Listen state and receives a hit, the job of the Register is to record the current BCO number which acts as a timestamp for the event. When the Command SM is in the Output state, then the Register must output the recorded BCO number.

The Command SM makes the transition from the Listen state to the Full state at the same time that the BCO number is changing from "n" to "n+1". (They both change state on the rising edge of the BCO Clock.) To prevent any race condition from developing, each bit of the End-of-column register is designed as shown in Figure 32. The heart of each bit of the End-of-column is a *negative* edge-triggered d flip-flop. The input to each flip-flop is a CMOS 2-to-1 multiplexor. When the write control (Wctrl) is active, then on the negative edge of every BCO clock, the Register will be updated to the present value of the BCO number. When the write control is inactive, then on the negative edge of every BCO clock, the Register will be refreshed to its present value. The Wctrl signal of each End-of-column Register is equivalent to the Listen signal of the corresponding End-of-column Command SM.

The CMOS switch at the output of Figure 32 allows all four End-of-column Registers to be placed on the same bus. The Rctrl signal of each End-of-column Register is equivalent to the Output signal of the corresponding End-of-column Command SM.

A simulation of an End-of-column Register is shown in Figure 33. "DataIn" in the figure is the BCO number. It changes on every rising edge of the BCO clock. "StoredBCO" is the number being held by the End-of-column Register. Note that it changes to the current value of

"DataIn" on every negative edge of the BCO clock as long as the Command SM is in the Listen state. "DataOut" is the output of the End-of-column Register. It is tristated unless the Command SM is in the Output state.

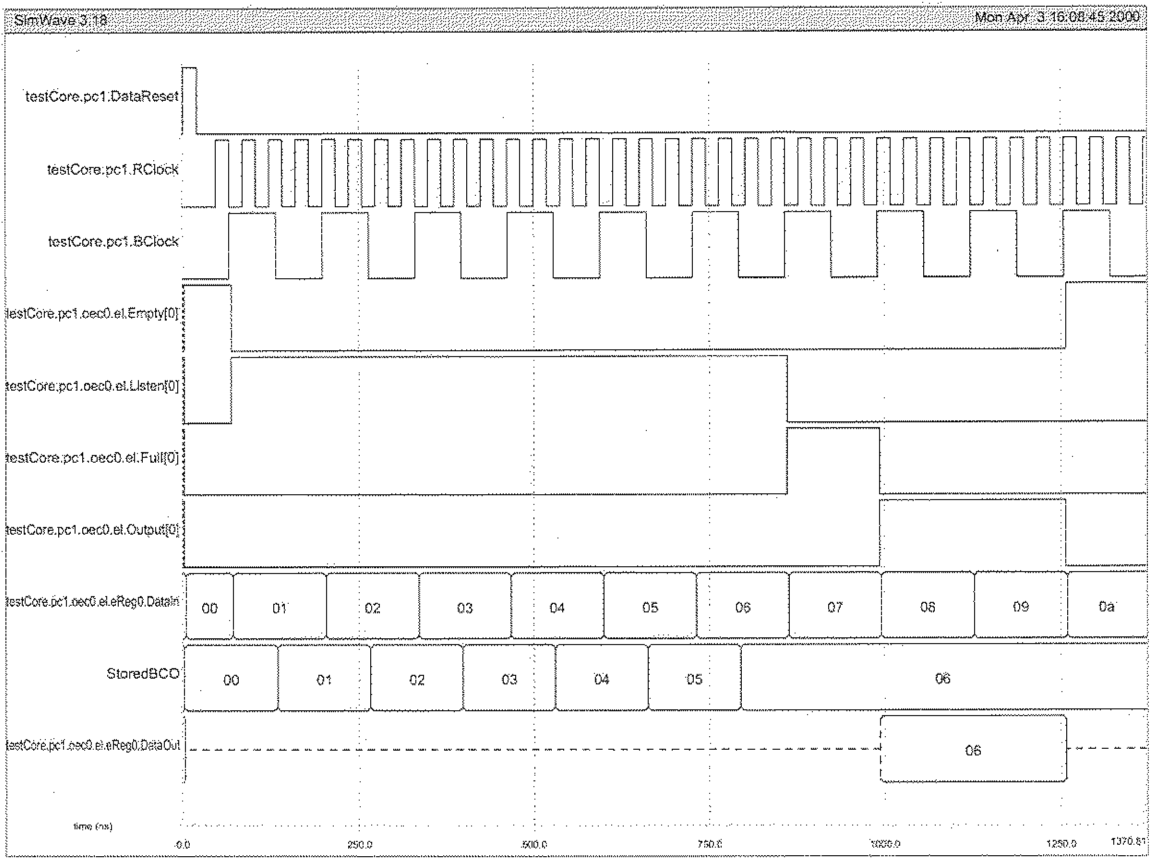


Figure 33: Simulation of a single End-of-column Register through a hit cycle

6.6 FAST OR LOGIC

There are two types of FastOR logic used in the End-of-column Logic. The first and simplest is the HFastOR logic shown in Figure 34. In this circuit, a weak PMOS transistor serves as the pull-up for the distributed NOR gate throughout the column. This NOR is inverted to an OR and then run through a non-clocked d flip-flop. The output, Hit, is a one when the HFastOR is low (pulled down).

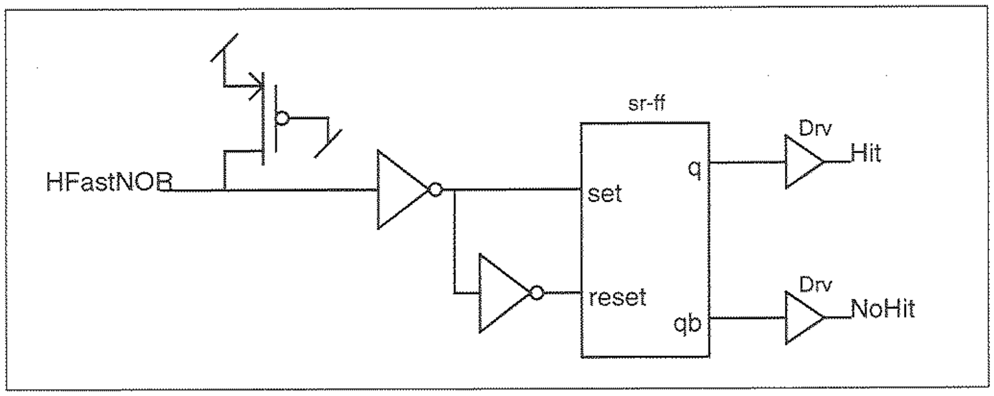


Figure 34: HFastOR Logic

The RFastOR logic is slightly more complicated because it is actually looking for the rising edge of the RFastNOR signal. On that rising edge, the column is outputting the last pixel that needs to be output. To accomplish this, the RFastNOR signal is brought directly into the clock input of a positive edge-triggered d flip-flop. On the rising edge of RFastNOR, the flip-flop changes ReadDone to active. ReadDone remains active until it is reset, which is accomplished by either the column going silent or the master reset activating.

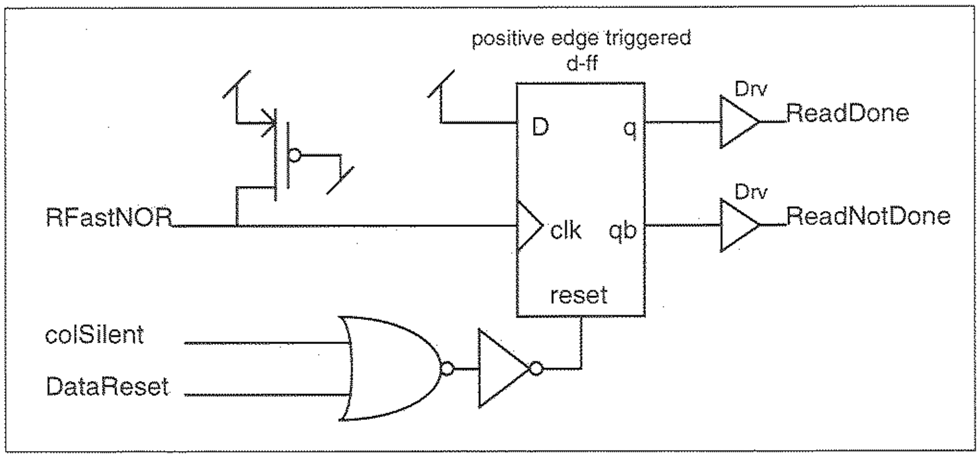


Figure 35: RFastOR Logic

6.7 LISTEN PRIORITY ENCODER

As indicated earlier, there were extensive modifications to Command State Machine in the development of the FPIX Core. Principle among them was the change in the state structure from the simple "Empty" and "Full" used in FPIX1 to the more complete "Empty", "Listen", "Full", and "Output" used in the FPIX Core. Using the latter allows the complicated, state-machine-based Priority Encoder used in FPIX1 to be replaced with a simple combinatorial logic block in the FPIX Core. The logic simply makes State Machine 0 the next to listen if it is currently in the Empty state. If SM0 is not Empty, then SM1 is next if it is Empty. If both SM0 and SM1 are not Empty, then SM2 is next if it is Empty. If SM0, SM1 and SM2 are all not Empty, then SM3 is next if it is not Empty. This logic is shown in Figure 36.

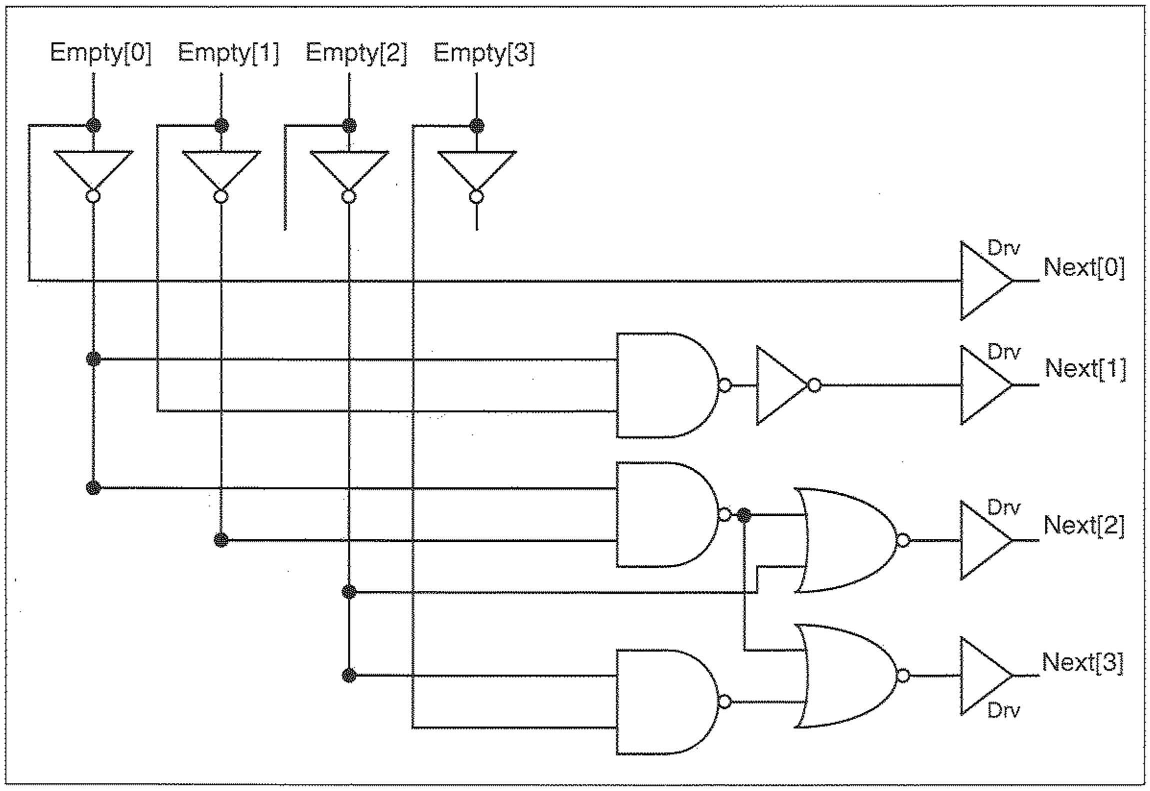


Figure 36: Listen Priority Encoder

Also, the Listen Priority Encoder is responsible for determining if no Command State Machines are in the Listen state and if not Command State Machines are free. "NoneFree" is active (high) if no Command SM is in the Listen state *and* no Command SM is in the Empty state. If NoneFree is active, then the Accept signal to the pixel column is brought low (inactive) so that no new hit data will be converted into recorded data until an End-of-column register is free. This prevents hits from appearing in the wrong BCO number on hot chips.

6.8 OUTPUT PRIORITY ENCODER

One of the inefficiencies in FPIX1 involves how it selects which Command SM to output next. External to the End-of-column Logic, the a requested BCO counter incremented through possible BCO numbers and a Command SM started to output when their was a match between the requested BCO and its stored BCO number.

In the FPIX Core, there are no requested BCO numbers, so each End-of-column Logic must have the ability to determine which Command SM goes next. A straightforward priority encoder such as the one used in the Listen Priority Encoder will not work. Old data could get "trapped" in Command SM3 and never get out because Command SM0 continually receives the right to output. Instead, what is used is a circular priority encoder in which, at any given time, the

Command SM with the lowest priority is the SM that is currently in the Output state. If no SM is in the Output state, then a "seed" SM establishes the priority.

The algorithm has two parts. The first describes how to pass or withhold the right to output.

1) If I am **not** in the Full state and I receive the right to output from my neighbor to my left, I pass the right to output to my neighbor on my right.

2) If I am **not** in the Full state and I am the seed register, I pass the right to output to my neighbor on my right.

3) If I am in the Output state, I pass the right to output to my neighbor on my right.

4) If I **am** in the Full state and I receive the right to output from my neighbor to my left, I withhold the right to output from my neighbor to my right

5) If I **am** in the Full state and I am the seed register, I withhold the right to output from my neighbor to my right.

The second part of the algorithm describes which register gets to advance to the Output state next.

1) If I **am** in the Full state and **not** in the Output state and I am the seed register, then I am the next to Output

2) If I **am** in the Full state and **not** in the Output state and I receive the right to output from my neighbor to my left, I am the next to Output.

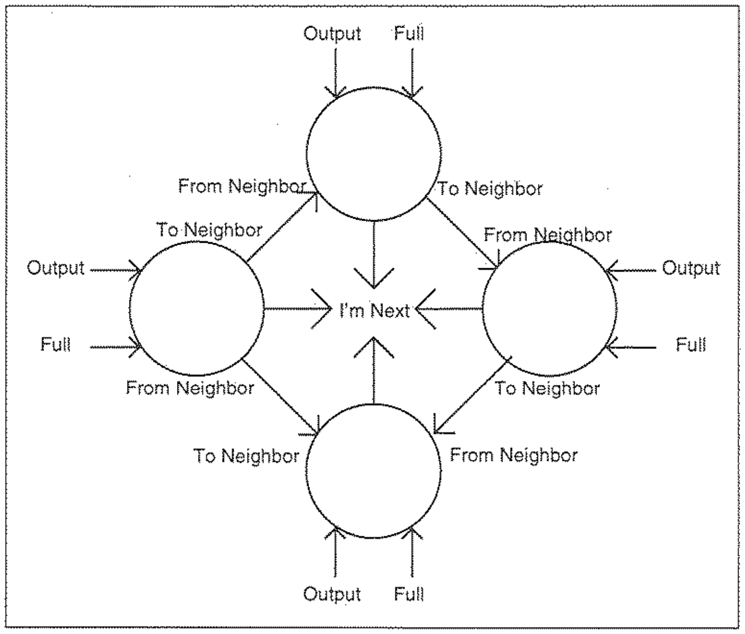


Figure 37: A diagram of the Output Priority Encoder logic

The above algorithm can be accomplished with purely combinatorial logic. Four identical circuits are arranged in a circle. The seed input of three of them are grounded and the last is tied to a signal which is high if not Command SMs are in the Output state.

6.9 A DETAILED DESCRIPTION OF A HIT FROM THE END-OF-COLUMN LOGIC'S PERSPECTIVE

Upon reset, the four Command State Machines are forced to their Empty state and the Column State Machine is forced to its Nothing-to-say state. The column Token and the column Read Clock are both forced to a zero. The RFastOR Logic is reset, indicating that any reads are not done.

The fact that all four Command State Machines are Empty, NoListen is activated indicating that nobody is in the Listen State. Moreover, since all of the State Machines are Empty, NoneFree is inactive, indicating that there are free State Machines. Therefore, at the next rising edge of the BCO clock, one state machine will be moved to the Listen state. The Listen Priority Encoder selects SM0 for this honor since it is Empty. Recall that an Empty SM0 supersedes all other State Machines. Furthermore, since NoneFree is inactive the Accept signal driven up the column is a one (assuming the user is not activating the MasterReject).

In this state, with no Command State Machines Full or Outputting, the Output Priority Encoder is indicating that no one will be the next to Output and that everyone has the right to output. However, nobody is paying attention to the Output Priority Encoder at this point.

The End-of-column Logic can remain in this state indefinitely, with one Command SM outputting a Listen Command, the other three outputting the Idle Command, and the Column SM indicating Nothing-to-say. The Listen Priority Encoder indicates that SM1 will be the next to Listen, but there has been, as yet, no cause to advance the state machines so the Listen Priority Encoder will just continue to indicate that SM1 will be the next to Listen. The Output Priority Encoder is still outputting nonsense, but at this point, no one cares. At every falling edge of the BCO clock, the BCO number is latched by SM0's End-of-column Register.

Eventually, the HFastOR logic will indicate that a hit has been received. This will cause several things to happen.

1) SM0 will make the transition from the Listen State to the Full State.

2) SM0 will stop outputting the Listen Command and start outputting the Idle Command.

3) SM1 will make the transition from the Empty State to the Listen state because it was marked as the next to listen by the Listen Priority Encoder.

4) SM1 will start outputting the Listen Command

5) The Listen Priority Encoder will mark SM2 as the next to listen.

6) Now that something is in the Full state (SM0), the Output Priority Encoder will indicate that SM0 is marked as the next to output.

7) SM0's End-of-column Register will hold the last BCO number it latched.

The system will not remain in this state for long. Since no one is in the Output State, NoOutput is active. With an active NoOutput signal and an Output Priority Encoder which indicates that someone (SM0) is marked as the next to output, then on the next rising edge of the BCO clock, SM0 will make the transition from the Full State to the Output state.

For the sake of simplicity, we can assume that no other hits have occurred. However, this is not a requirement of the system. Had their been another hit, SM1 would have moved into the Full state, SM2 would have moved to the Listen state, etc.

The transition to the Output state drives the Output Command up the column. This alerts the Column State Machine that someone is in need of outputting. At the next rising edge of the Read Clock, the column state machine will make the transition from the Nothing-to-say state to the Something-to-say state. This has two effects.

1) The Column Token will be driven up the column to the first pixel that requires it.

2) The Core Logic will be alerted that the Core "HasData".

The system can remain in this state indefinitely with Command SM0 in the Output state and the Column SM in the Something-to-say state. What we are waiting for is the horizontal Token to indicate that this End-of-column Logic can grab the Core bus. When this happens, at the next rising edge of the Read Clock, the Column SM will make the transition from the Something-to-say state to the Talking state. This will release the column Read Clocks into the column and data will begin to pour through the End-of-column to the Core bus. It will also release the BCO number stored in SM0's End-of-column Register to the Core bus.

At the rising edge of the RFastOR, the ReadDone signal is activated. This passes the Horizontal token to the next needy column, and on the next rising edge of the Read Clock, the Column SM will make the transition to the Silent state. This activates the Done signal in Command SM0.

While Done is active and until the next rising edge of the BCO clock, Command SM0 will output the Reset Command just in case there is some miscommunication between the End-of-column Logic and the pixels. At the next rising edge of the BCO clock, Command SM0 will make the transition back to the Empty state. This will reset the Done signal, completing SM0's hit cycle.

The Column SM will remain in the Silent state until it receives the signal from the Core indicating that the Core has gone silent, i.e. that no other columns are talking. When this happens, the Column SM makes its final transition back to the Nothing-to-say state completing its hit cycle. This last transition also resets any flip-flops in the address section of the column that may still be active. Again, this is just a precautionary measure in case there is some miscommunication between the End-of-column Logic and the pixels.

7 CORE LOGIC

7.1 SIGNALS

Outputs from the Core Logic to the End-of-column Logic cells.

1	CORETALKING	Output. Signals that the Core is Talking
2	CORESILENT	Output. Signals that the Core is Silent
3	BCO<7:0>	Output. The current BCO number
4	COMD<1:0>	Output. Command State Machine D command pair

Outputs from the Core Logic to Periphery.

5	COREHASDATA	Output. Signals that the Core has data it needs to output
6	COREHIT	Output. Signals that some pixel the Core has been hit
7	COREERROR	Output. Signals that the column Token in some column has exited the top of the column before Read Done activates.

Inputs to the Core Logic

8	BCOCLK_IN BCOCLKB_IN	Input. Beam Cross-over clock.
9	READCLK_IN READCLKB_IN	Input. Read Clock
10	DATARESET	Input. Reset

7.2 INTRODUCTION

The increased intelligence of the End-of-column Logic and the decision to design the FPIX Core as a purely non-triggered system has greatly simplified the Core design. It consists of a simple counter, a very stupid state machine and a trio or diagnostic output circuits.

7.3 CORE COUNTER

One of the three principle components of the Core Logic is the Core Counter that is nothing more than a resettable counter that changes state on the rising edge of the BCO clock. The counter has seven bits wide. The reset is asynchronous.

7.4 CORE STATE MACHINE

The second major component of the Core Logic is the Core State Machine. Its sole purpose is to determine when the Core is Talking and when it is Silent. It operates synchronous with the rising edge of the Read Clock.

There is no chip token as far as the FPIX Core is concerned. Therefore, as soon as the Core has data to send, the Core SM changes state to Talking on the next rising edge of the Read Clock.

The Talking signal becomes the horizontal token passed among the columns during the Output sequence. The Core SM remains in the Talking state until the horizontal token comes out of the last column in the Core. At the next rising edge of the Read Clock after the horizontal token comes out of the last column, the Core SM switches back to the Silent state. Its that simple.

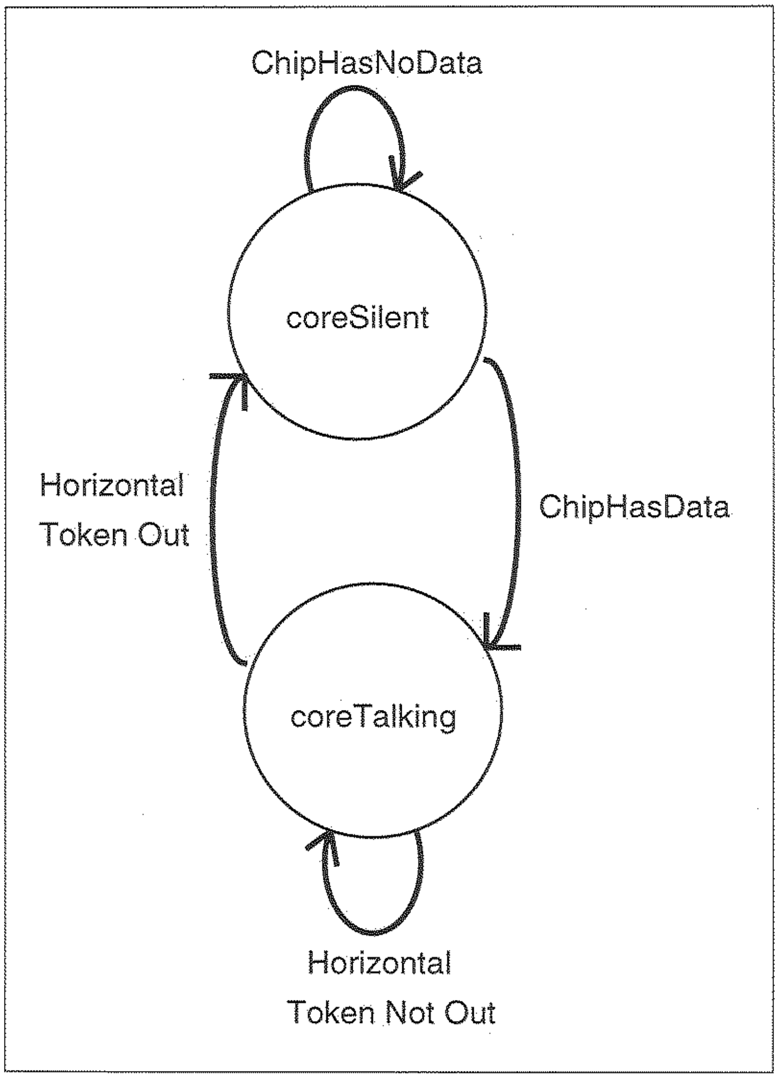


Figure 38: Core State Diagram

7.5 DIAGNOSTIC SIGNALS

The Core Logic also supports three diagnostic circuits identical to the HFastOR logic in each column. However, instead of operating within a single column, they operate across all of the columns. Within each End-of-column Logic, there are three pull-down transistors. One is gated by the hit signal output from that column's HFastOR Logic. This transistor will be used to generate a signal indicative of the presence of a hit anywhere in the Core.

A second pull-down transistor is gated by the logical ORing of the Column State Machine's Something-to-say and Talking signals. This transistor will be used to generate the ChipHasData signal used by the Core State Machine to make transitions between Silent and Talking.

The third pull-down transistor is gated by the logical ANDing of Column Token Out with ReadNotDone. If the column Token comes out of the top of the column before Read Done activates, then there has been some kind of error. This pull-down transistor is used to indicate the presence of such an error.

This is illustrated for three columns in Figure 39.

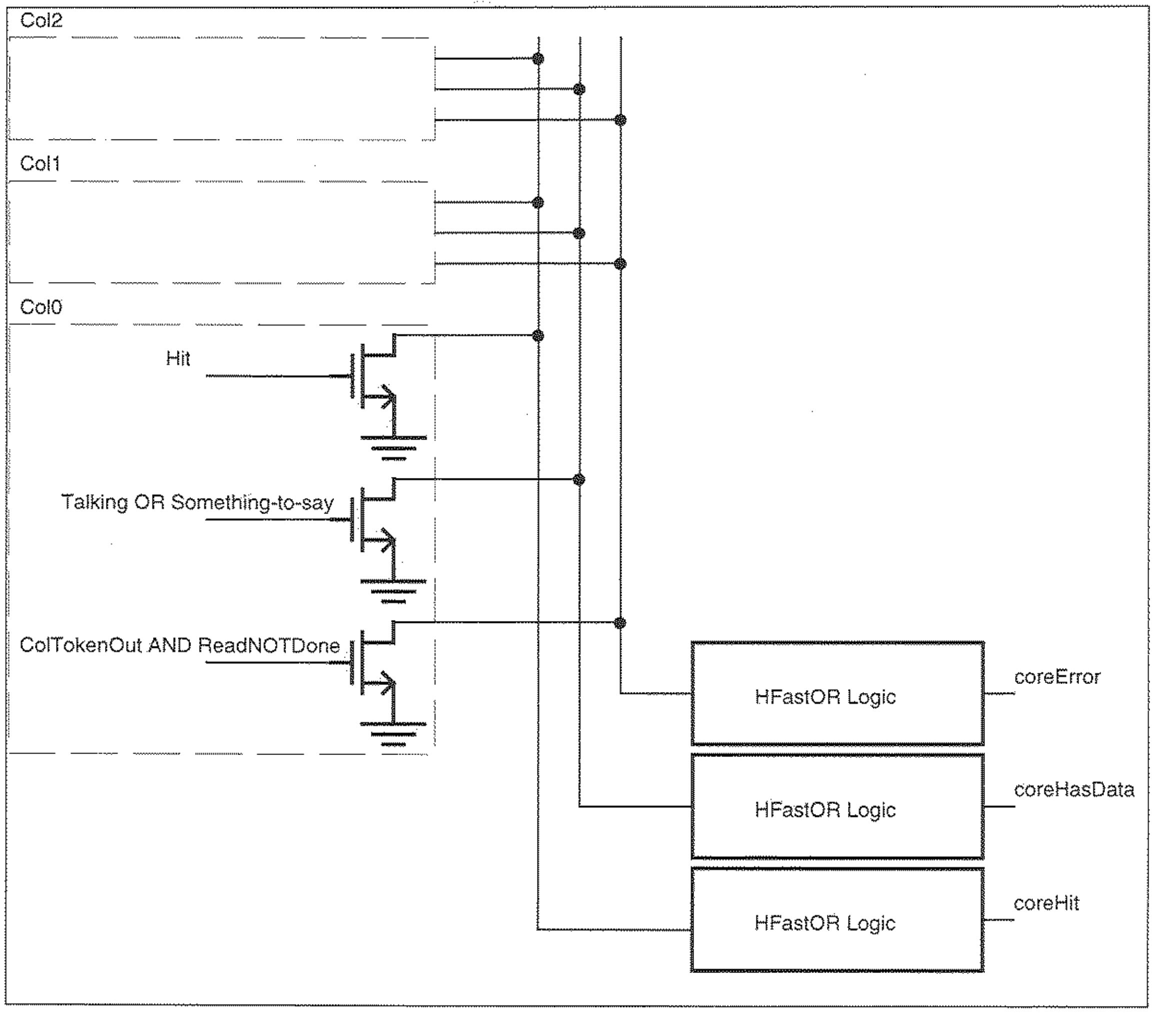


Figure 39: Core Diagnostic signals

8 VERILOG SIMULATION

The FPIX Core design was subjected to extensive Verilog simulation both before and after layout. First, realistic propagation delay values were determined by SPICE simulation for standard blocks such as inverters, 2- and 3-input Nand gates, 2- and 3-input Nor gates, etc. Larger components such as flip-flops were created from the building blocks. Other components such as large drivers were simulated in SPICE driving the maximum conceivable load. The delay required to drive such a load was back annotated to the Verilog model. Finally, special nodes such as HFastOR lines were individually simulated under realistic conditions (i.e. full sized transistors and full 160-pixel columns). The delays on such lines were back annotated to the Verilog model as well.

The Verilog model of the FPIX Core itself is completely structural in nature. No behavioral modeling was used. The reason for this is simple: since such pains were taken to accurately back annotate structural block delays to the Verilog model, it made no sense to short-cut those delays by modeling circuits behaviorally. Furthermore, Cadence provides a path whereby schematics can be extracted directly from structural Verilog code. Using this path assured the designers that layout-versus-schematic comparison was, in effect, a layout-versus-Verilog comparison, and, therefore, we could be comfortably certain that the final chip would behave as it was simulated.

The Verilog code for the simulation could be broken down into three parts- the Detector, the FPIX Core, and the DAQ. The stimuli for the Verilog simulation was derived from the results of Monte Carlo analysis of the BTeV interaction chamber. Each set of stimuli represents 5000 time slices of operation or approximately 0.7 milliseconds. Three primary sets of stimuli were used, one at half the expected luminosity, one at full luminosity, and one at double the expected luminosity. Each hit includes the row and column number of the hit pixel and the magnitude of the hit expressed as a 5-bit number.

At the start of each simulation, all 5000 time slices are read into a memory array, and then at each rising edge of the BCO clock, 18x160 "pixels" in the Detector are loaded with the hits for that times slice. Each "pixel" in the detector is actually a tiny delay element that connects to one of the pixel cells in the FPIX Core. Based on the magnitude of the hit, the arrival of the hit data is delayed from the pixel cell in the FPIX Core. For a very large hit, the delay is very small – approximately 40ns. For a very small hit, the delay can be greater than 100 ns. These delays were determined by tests of earlier FPIX preamplifiers, and they can be adjusted to allow for studies of time walk. In addition to the hit data, the most significant 3 bits of the 5-bit magnitude are held by the "analog section" of the pixel cell to be used as the ADC values. Finally, each pixel in the detector "dies" for a period of time after it has been hit. If the hit was small in magnitude, the pixel dies for only 50 ns. If the hit was very large, the pixel dies for as much as 2 μs. This corresponds to the expected behavior of the preamplifier.

The DAQ is a very simple system latches the output of the FPIX Core on the falling edge of every Read Clock whenever coreTalking is active. It stores all of these values and then compares them with the original data input to the simulation. It then give an indication of the number of matches, misses (output data not found in the stimuli), scratches (garbage data), and missed originals (stimuli not found in the output data). This is shown in the table below:

Table 7: Simulation Results

Luminosity	Hits	Scratches	Matches	Misses	Missed Originals	Accuracy
0.5	1342	0	1341	0	1	99.9%
1.0	2751	0	2748	1	4	99.8%
2.0	11643	0	11537	31	106	99.1%

A detailed analysis of the data revealed that the majority of the missed originals correspond to one of two things. One, a second hit occurs on the same "pixel" while the "pixel" is dead. Two, a second hit occurs on the same pixel while the digital section is waiting to output its data. As a consequence, these hits, which are real to the Monte Carlo analysis, are never seen by the DAQ system since FPIX "ignores" them. Hence, they are "missing originals" – i.e. hits present in the stimuli that do not make it to the DAQ system.

A large percentage of the Misses can be attributed to time walk on small magnitude hits.

In any case, the accuracy of the FPIX Core and its ability to reconstruct its inputs faithfully is extremely encouraging. Even at twice the luminosity, we should not be limited in any way by the FPIX Core.