FPIX1 Digital Architecture and Operation: Design and Simulations

Jim Hoff
PPD/ETT/ES
June 25, 1998

Abstract

FPIX1 is a test chip for the C0/BTeV collaboration. It represents a first step towards the final pixel readout architecture necessary for this experiment. It does not use multiple tokens, FIFOs or multiple end-of-column readout busses to increase speed. Its purpose is to determine exactly how fast the chip can process information internally and therefore allow accurate extrapolation of the ultimate possible speed. The FPIX1 Digital Architecture can be divided into three mutually dependent pieces. These are the Pixel Cell, the End-of-column Logic and the Chip Logic. The Pixel Cells control individual pixel detectors. There will be almost 3000 Pixel Cells on the FPIX1 chip. They receive commands from and transmit information to the End-of-column Logic. Separate End-of-column Logic cells control each column. There will be 17 End-of-column Logic cells on the FPIX1 chip. They each control 160 Pixel cells, receive data from those cells and pass that data on to the Chip Logic. The Chip Logic receives information from the outside world, processes it, and passes that information on to the End-of-column Logic cells. It also receives data from the End-of-column Logic and passes it on to the outside world. There will be only one Chip Logic cell on the FPIX1 chip. The performance of the FPIX1 chip critically depends on the exchange of information between these three processing pieces. This paper is concerned with that information exchange and the algorithms used in the chip.

Introduction

FPIX1 is a single step towards the final FPIX chip which will satisfy, as much as is possible, the requirements of the C0/BTeV collaboration. As such, it is expected there will be changes and additions to FPIX1. These changes will be determined after FPIX1 has been tested. The task set out for FPIX1 is as follows: a 1x8 array of FPIX1 chips will simultaneously acquire data from approximately 23000 ATLAS-style pixel detectors during a beam test.

A basic tenant of the FPIX1 design is that it must be as simple as possible. This fundamental assumption serves two purposes. First, the simpler a system is, the more easily it is tested and the less there is that can possibly go wrong. Second, the simplest system provides a data point from which the performance of the final system with all of its additional complexities can be reliably extrapolated. This basic tenant of FPIX1 has eliminated grouping, FIFOs, multiple tokens and multiple readout buses from the design. All of these complexities can (and probably will) be added in future designs, and they will only increase the readout speed.

The rest of this paper will concern the operation of the Pixel Cell, the End-of-column Logic, and the Chip Logic and the inter-operation of these pieces.

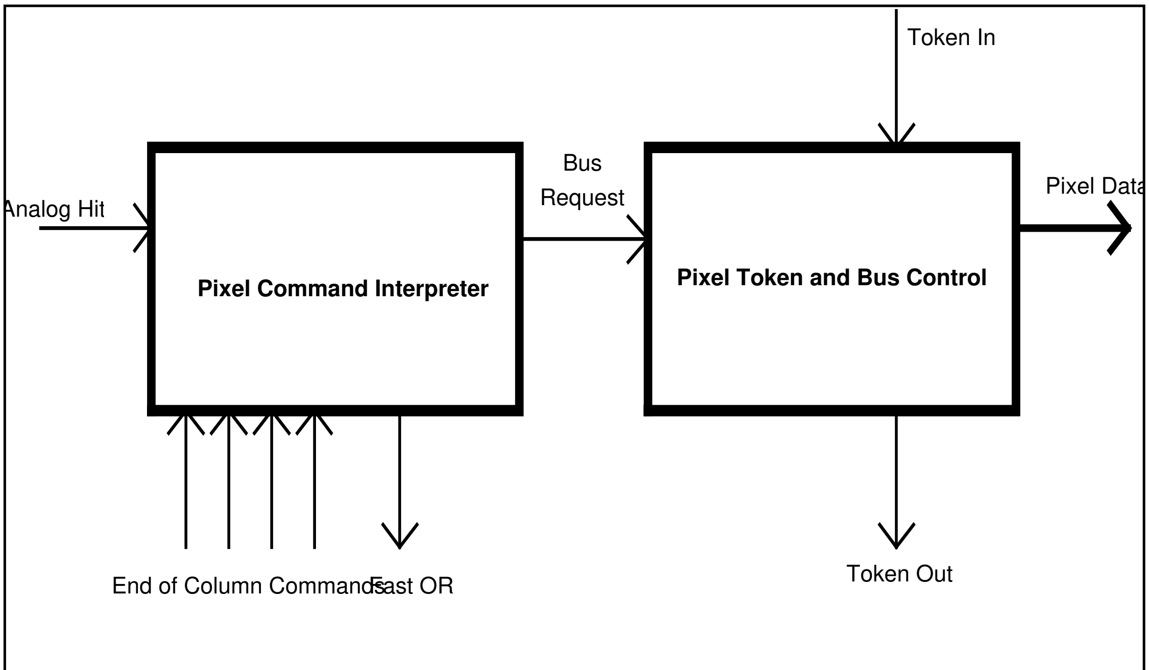


Figure 1: Pixel Cell: Black Box View

Logic Cells

The Pixel Cell

The above picture shows a black-box view of the Pixel cell, and, in particular, its two major components: the Command Interpreter and the Token and Bus Control. The Command Interpreter accepts an analog hit from the pixel detector and depending on the status of the End-of-column Commands, processes the hit. If appropriate, the End-of-column Logic is alerted to the presence of a new hit via the Fast OR signal. When commanded by the End-of-column Logic, the Command Interpreter requests the bus via the Token and Bus Control Logic. The End-of-column Logic will provide a token to the column of Pixel Cells as a means to regulate bus access. This token will propagate through the Pixel Cells with no information and stop where the Bus is requested. When a bus is requested and a token is received, the pixel's data is loaded onto the buses and driven to the End-of-column Logic. Finally, if commanded by the End-of-column Logic, a pixel will reset its contents.

A Detailed Description of the Commands

First, each column is a separate entity, so commands from one End-of-column Logic cell go only to its own column. Second, the End-of-column Logic commands all Pixel Cells simultaneously. This second point is particularly important because it means that the End-of-column Logic and the Pixel Cells must coordinate command and interpretation to guarantee that only the appropriate pixels obey the appropriate commands. Finally, there are four End-of-column Command Sets each corresponding to one of the four End-of-column Registers. These registers store interesting BC0 numbers and via their Command Set instruct their associated pixels.

There are four types of commands sent from the End-of-column Logic to the Pixel Cells:

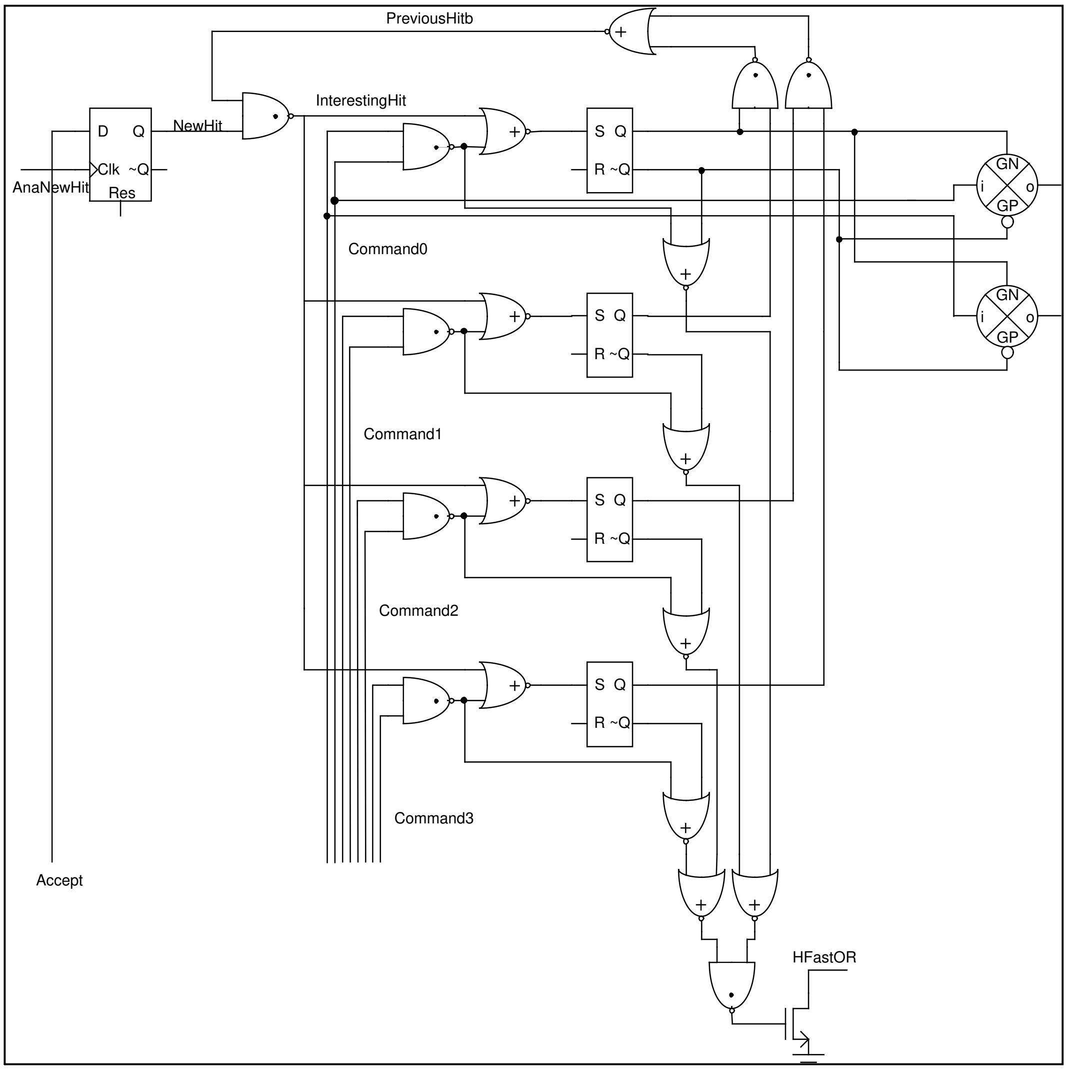
1. **Idle (00)** - If a particular pixel is Empty, then the Idle Command is ignored. If the pixel is Full (i.e. has stored a hit), then the Idle Command instructs the pixel to wait.

2. **Reset (01)** – If a particular pixel is Empty, then the Reset Command is ignored. If the pixel is Full, then the Reset Command instructs the pixel to clear its hit registers and return to the Empty state.

3. **Output (10)** – If a particular pixel is Empty, then the Output Command is ignored. If the pixel is Full, then the Output Command instructs the pixel to request the bus. When

the pixel has the bus, then its data is driven to the End-of-column Logic, and the pixel resets itself to the Empty state.

4. **Write (11)** – If a particular pixel is Empty, the Write Command tells the Pixel Cell that incoming hits from the pixel detector should reference themselves to the End-of-column register that is giving this Write Command. When a hit arrives, the Pixel Cell then activates the Fast OR to alert the End-of-column register that this BC0 is interesting. It also changes its internal state to Full, and shuts off all commands except those coming from the register currently sending this Write Command. If this Pixel Cell is already Full, then the Write Command is ignored.



A Detailed Description of a Hit from the Pixel Cell's Perspective

The above figure details the majority of the Pixel Cell logic. Command0, Command1, Command2, and Command3 as well as Accept are generated by the End-of-column Logic and driven up the column. First, the rising edge of a new hit from the Pixel Cell's discriminator will make "NewHit" equal to "Accept". If the End-of-column Logic has activated Accept (i.e. made it a Logical 1), then NewHit is activated by the firing of the discriminator. If Accept is set to a Logical 0, then nothing from the analog section of the Pixel Cell can affect the digital section.

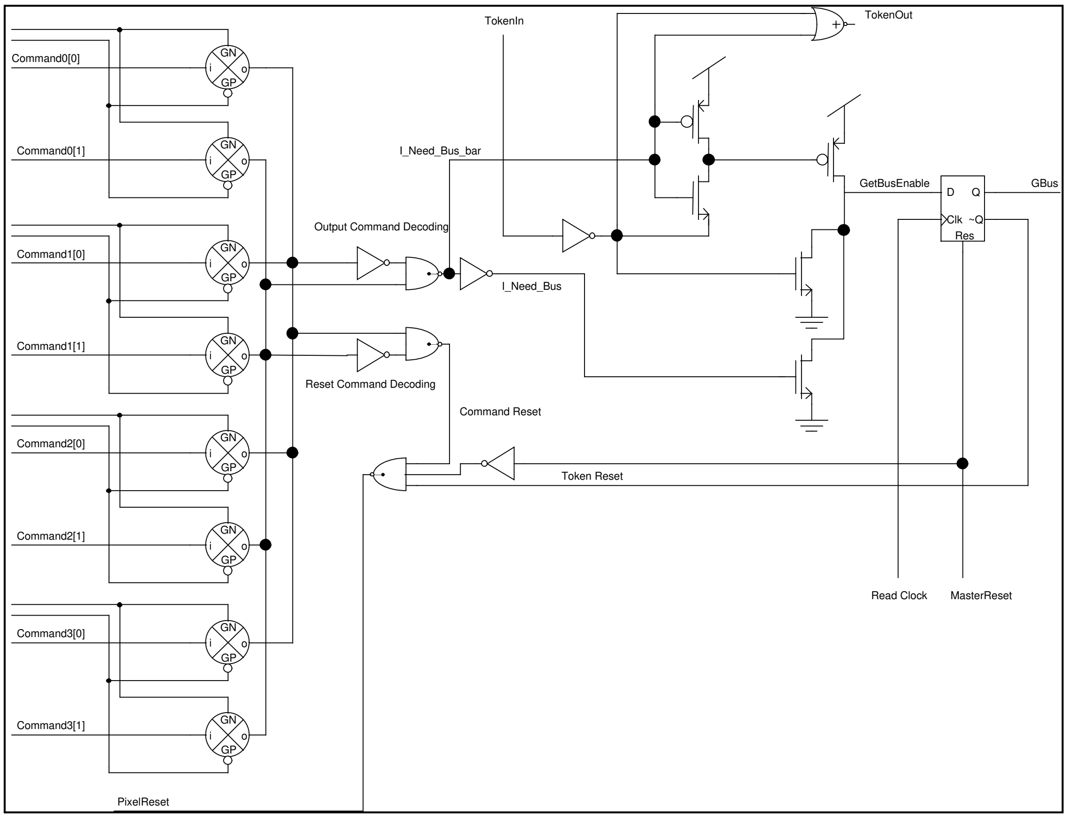
PreviousHitb is set to a Logical 1 if and only if the Pixel Cell is in the Empty State. If the Pixel Cell is Full, then PreviousHitb is a Logical 0 and the signal "InterestingHit" will remain inactive (Logical 1) regardless of activity in the analog section. It is by this mechanism that a Full Pixel Cell ignores subsequent Write Commands. In other words, if the Pixel Cell is Full, then even if another End-of-column register presents a Write Command to the Pixel Cell and there is another hit on the pixel detector causing the discriminator to fire, the fact that PreviousHitb is a Logical 0 will prevent this new hit from disturbing the state of the Pixel Cell.

If there is no previous hit (i.e. the Pixel Cell is Empty) and the End-of-column Logic indicates that the Pixels should accept new hits (i.e. Accept is set to Logical 1) and one of the Command Sets is presenting a Write Command to the Pixel Cell and if the discriminator fires, then InterestingHit will become active and the SR flip flop associated with the End-of-column register presenting the Write Command will activate. This will cause PreviousHitb to become a Logical 0, preventing future hits from disturbing this logical state. At the same time, the nmos transistor that pulls the HFastOR line low will be activated. The Fast OR (a.k.a. the HFastOR) is a distributed pseudo-nmos NOR gate in which a single pmos transistor (located in the End-of-column Logic) pulls the Fast OR line high unless any nmos transistor (one located in each of the Pixel Cells) pulls the line low. Note that the HFastOR signal is not activated until the record of the hit is already stored in the Pixel Cell, and that signal will remain active until the End-of-column register removes the Write Command. This guarantees that the End-of-column Logic has the time to acknowledge that a hit occurred. However, it also means that if the End-of-column Logic mistakenly re-asserts the Write Command to a Full Pixel Cell, the HFastOR will automatically fire. The Pixel Cells only know if they have been hit. They do not understand time in any way. It is impossible to insert in the Pixel Cell the extra logic necessary for the Pixel Cell to be aware that it was hit in a particular BC0 crossing unless the BC0 clock is driven up the column to all pixels. This would take up space and it would be an extra noise source to the analog front end. Therefore, the End-of-column Logic must be smart enough not to re-assert the Write Command until it has asserted a Reset Command or an Output Command.

The figure shows only two CMOS transmission gates on the right. In fact, there are two for every Command Set. As the figure indicates, when a hit is stored in the Pixel Cell only two of the eight transmission gates are activated. These are the two associated with the End-of-column register providing the Write Command when the hit occurred. It is by this mechanism that the Pixel Cell dynamically associates itself with only one End-of-column register. After the association has been made, the Pixel Cell ignores the commands of all other registers except the one with which it is associated.

Finally, the Pixel Cell is dumb. If more than one End-of-column register is asserting the Write Command, then the Pixel will associate itself with more than one End-of-column register. The consequences of this are unpredictable, and the Pixel Cell does not have the space to provide the logic necessary to make sure such problems do not occur. Therefore, it is up to the End-of-column registers to make sure that only one Write Command is issued at a time.

A Detailed Description of Outputting from the Pixel Cell's Perspective



The above figure shows the eight CMOS transmission gates on the left, only two of which will be active when the Pixel is Full and none of which will be active when the Pixel is Empty. When active these transmission gates will pass the commands of the associated End-of-column register on to the rest of the Pixel Cell. When the Output Command is given, two things happen. First, in a manner identical to the HFastOR, the Pixel Cell activates an RFastOR to indicate to the End-of-column Logic that there is data to be output.

Second, the Output Command Decoder will activate the Token and Bus Control Logic via the I_Need_Bus signal. When this signal is active, the TokenOut is forced to a Logical 0, and when the TokenIn is active (i.e. when the Column Token has arrived), then the GetBusEnable signal is activated. At the next rising edge of the Read Clock, the Pixel will have control of the bus, and its address will be driven for one Read Clock cycle. The act of getting the bus also forces the Pixel Cell to reset itself via the Token Reset signal. This will immediately revert the Pixel Cell to the Empty state making the I_Need_Bus signal inactive. This will release the Pixel Cell's hold on the RFastOR signal and it will release the Token to fall to the next active pixel (i.e. TokenOut will become a Logical 1). Note that the Token dropping occurs while the Pixel Cell's data is being driven. At the next rising edge of the Read Clock, this Pixel Cell will release the bus because, due to the reset, it thinks it no longer needs the bus. This mechanism of pipelined token passing and self resetting is what enables FPIX1 to output new data at each rising edge of the Read Clock.

If the Pixel Cell had no data, then the Output Command would have been ignored, and the TokenIn signal would have immediately caused TokenOut to be driven to a Logical 1 (i.e. the Column Token would have passed right through to the next active Pixel Cell).

A Detailed Description of Resetting from the Pixel Cell's Perspective

As seen in the above figure, there are three ways to reset the Pixel Cell. The first happens when the End-of-Column Logic issues a Reset Command. This is called a Command Reset. The second happens when the Pixel Cell's Data is output. This is called a Token Reset. The third happens when a Data or Master Reset is activated.

In all three cases, Resetting causes the following:

1. All of the SR flip-flops in the Pixel Cell are reset. These are the flip-flops which associate the Pixel Cell with one of the four End-of-column registers

2. The input D-flip-flop is reset. (This is the flip-flop ensures that only the rising edge of a New Hit activates the Pixel Cell logic. It also enables the End-of-column Logic to force the pixels in the column to ignore new inputs via the Accept signal.)

3. The Pixel Cell is forced into the Empty state and PreviousHitb is set to Logic 1. This effectively opens the door for new inputs.

4. The Pixel Cell, which prior to the reset had been ignoring all but one register, is now free to look at all registers again.

It should be reiterated that the Pixel Cell is dumb. It does not care why or how it got the commands it got. It simply responds to them. Therefore, if the Pixel was in an Output mode (i.e. if the End-of-column Logic was driving an Output Command up the column) and then the command was switched to a Reset Command, the Pixel Cell would reset itself in about 1 ns, even if the Token had already arrived. Therefore, the End-of-column Logic must take care not to allow spurious commands to be driven up the columns.

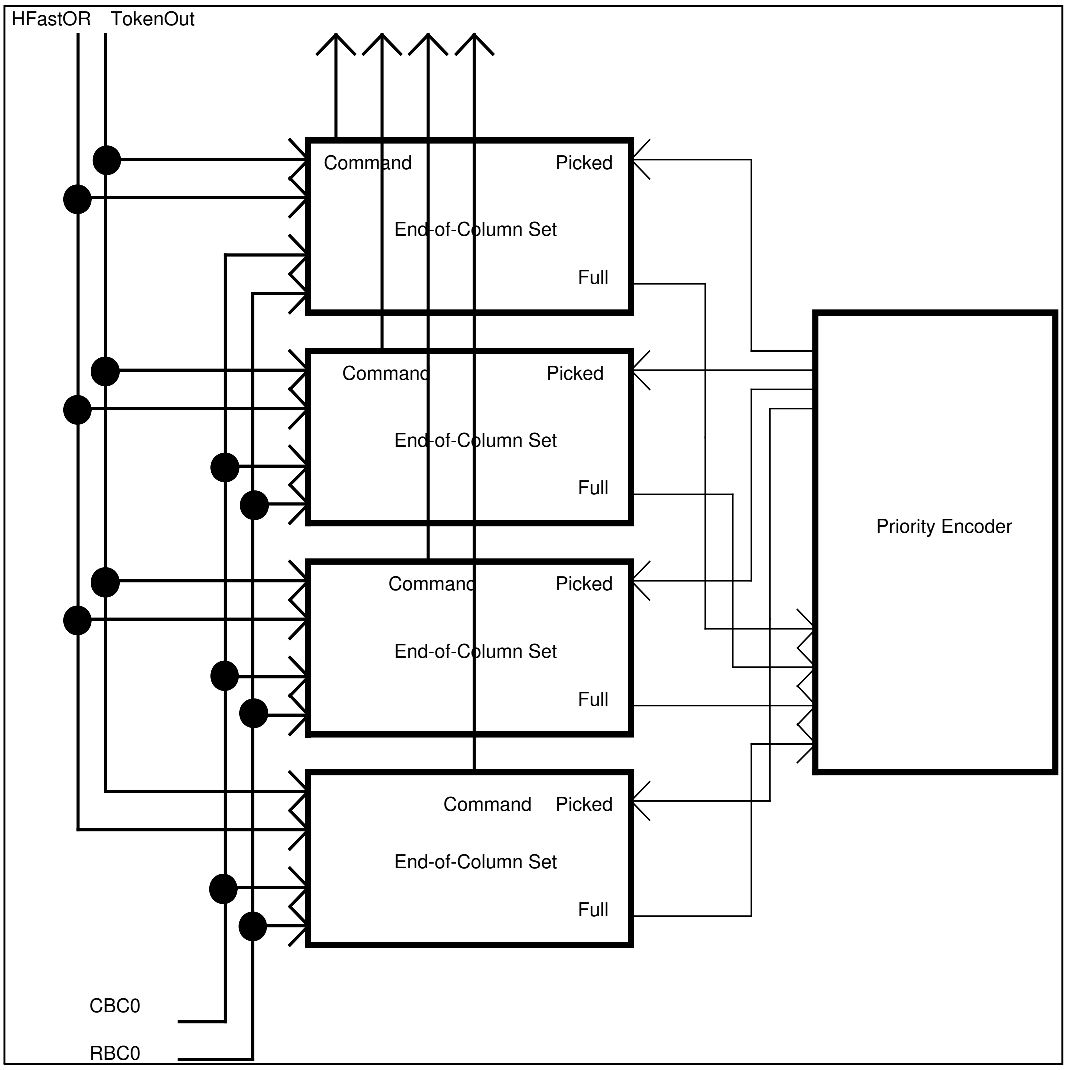
Interface Limitations as dictated by the Pixel Cell

To summarize what was stated previously, several limitations are placed on signals from the End-of-column Logic by the Pixel Cell. They are:

1. One and only one End-of-column register may issue a Write Command at any given time. The consequence of not obeying this rule is that a Pixel Cell could simultaneously associate itself to two or more End-of-column registers if more than one Write Command were given.

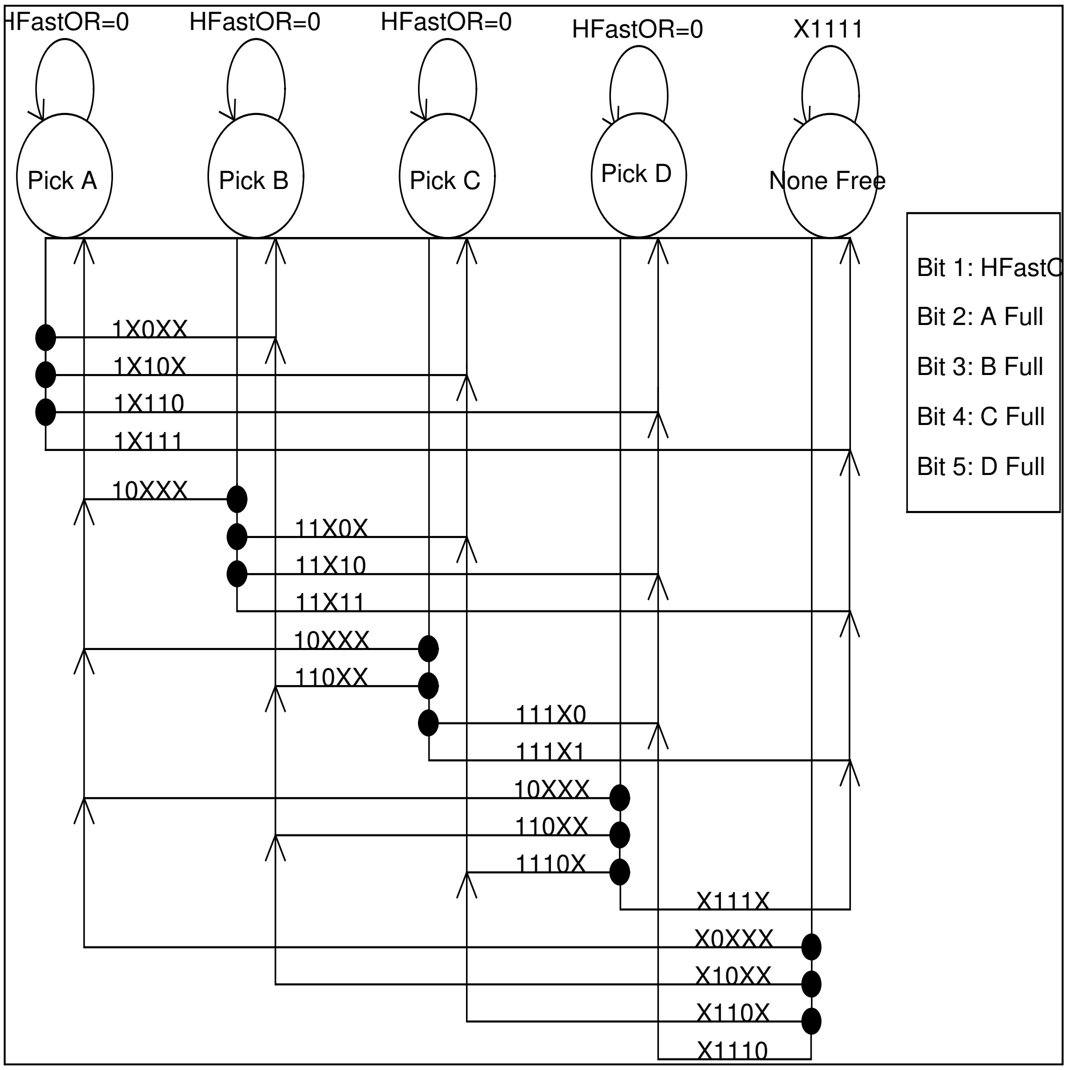
2. Once the End-of-column register acknowledges a Fast OR signal (i.e. the arrival of a new hit) by removing the Write Command, it cannot re-issue a Write Command until it has issued a Reset Command or an Output Command or until the outside world has activated the Data Reset. The consequence of not obeying this rule is that spurious Fast OR signals would be sent to the End-of-column registers if Write Commands were issued to Pixel Cells in the Full state.

3. Care must be taken with the Reset Command. It is the fastest command, taking only 1ns to reset a Pixel Cell. Therefore, spurious signals on the command lines must be avoided.



The End-of-column Logic

The above figure shows the basic block diagram of the End-of-column Logic. It consists of one Priority Encoder and four End-of-column Sets. The End-of-column Sets themselves consist of one End-of-column register for holding a BC0 number, two digital comparators for checking the held BC0 number against the Current BC0 number (CBC0) and the Requested BC0 number (RBC0), and one End-of-column State Machine for generating the



appropriate commands. In addition, there is one End-of-column Mask Shift register for storing the CBC0 mask. This will be explained below.

The Priority Encoder

The Priority Encoder is a very straightforward state machine and it is shown in the above figure. First, it changes state with the BC0 clock. Second, under typical operation (i.e. if at least one register is free) the Priority Encoder is not allowed to change state unless there is a hit (i.e. if HFastOR is a Logical 1). Third, if there is a Hit, the present state cannot be the next state because, by definition, the present register will be occupied. Finally, given that the above rules are obeyed, the priority encoding is as follows:

1. If register A is free, register A is the next pick.

2. If register A is full and register B is free, register B is the next pick.

3. If registers A and B are full and register C is free, register C is the next pick.

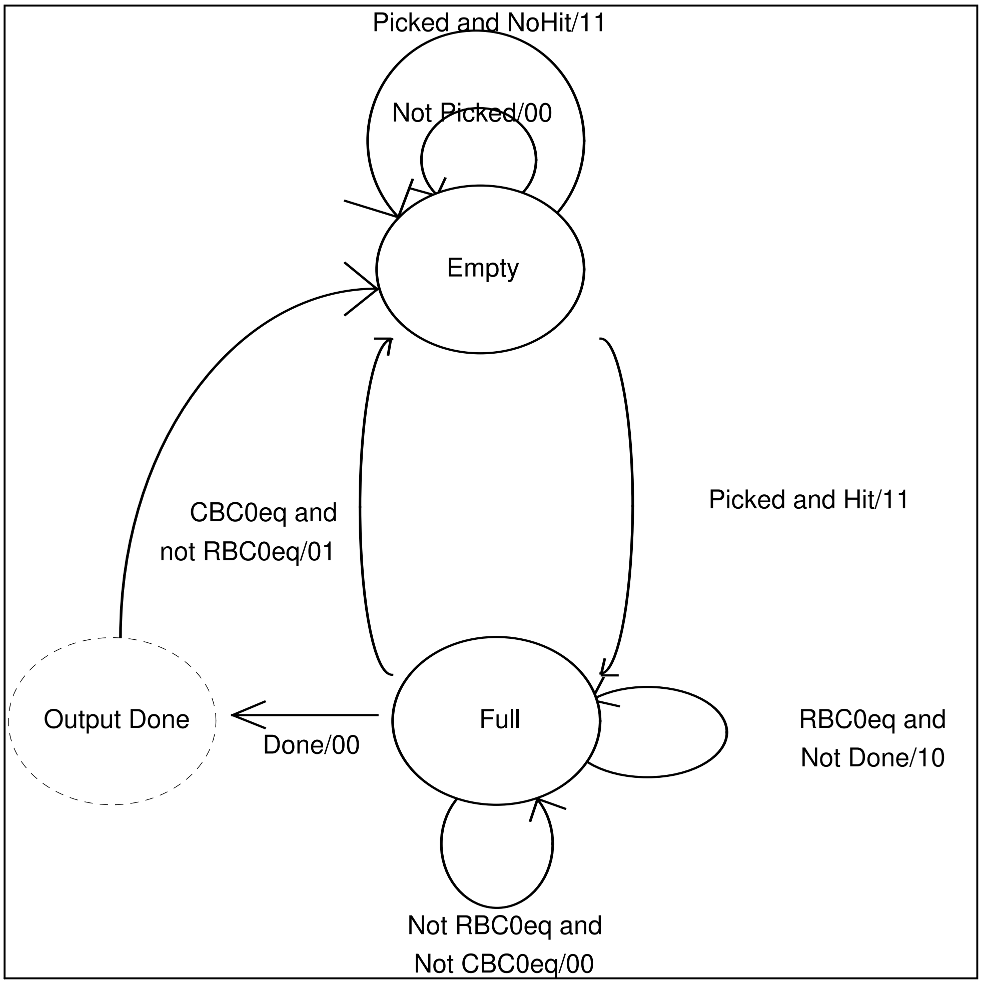
4. If registers A, B, and C are full and register D is free, register D is the next pick.

5. If none of the registers are free, then NoneFree is activated, and no registers are picked.

It is assumed that with no free registers, no End-of-column Sets will be presenting a Write Command to the column. Consequently, no Hits will be recognized by the End-of-column Logic since the HFastOR signal is activated by the simultaneous presence of a new hit and a Write Command. Therefore, when no registers are free, the End-of-column Priority Encoder ignores the HFastOR line and changes state as soon as there is a free register.

The End of Column State Machine

The End-of-column State Machine is the only Mealy State machine in the entire chip. The others (Priority Encoder, Column Readout, Chip Readout, CBC0 and RBC0 counters) are all Moore State machines. The difference is that the outputs of Moore state machines depend solely on the present state of the machine whereas the outputs of Mealy state machines depend both on the present state of the machine and on the present states of the asynchronous inputs to the machine. This choice was dictated by the fact that new hits must be tagged according to the current BC0 number. This means that they are recorded



based on the BC0 clock. However, they are output based on the Read Out clock. These two clocks cannot be assumed to be the same frequency or even to be synchronous. Therefore, a Moore State machine would have had to change state on two different clock edges, which is clearly impossible. The Mealy State machine, on the other hand, can be made to change state on only one clock edge with provisions made for dealing with the seemingly asynchronous signals arriving in time with the second clock edge. Finally, provisions also need to be made so that the column can understand that it is done reading out, but the rest of the chip is not yet done.

The above state machine accomplishes these purposes. First, transitions from Empty to Full and from Full to Empty occur on BC0 clock edges. Transitions from Full to Output Done can occur at any time. Transitions from Output Done to Empty occur on BC0 clock edges. Output Done, therefore, acts as a pseudo-Full state. The state machine thinks that it is Full, and therefore the Priority Encoder cannot pick the register for the storing of new hits. However, in the Output Done pseudo-state, the End-of-column Logic will relinquish control of the buses thereby allowing the requested BC0 counter to be incremented. This allows multiple BC0 numbers to be requested within the same BC0 cycle.

Note that if a register is empty and not picked by the Priority Encoder, the command output by the End-of-column State Machine is Idle. If, however, the register is picked, the command becomes Write, and this is maintained until either the Priority Encoder picks another register or until the next positive BC0 clock edge after a Hit is recorded. Recall that the Pixel Cell does not know anything about time and it does not care who is sending it the Write Command. It is the responsibility of the Pixel Cell, when hit, to maintain the

HFastOR signal until the Write Command is removed. It is the responsibility of the Priority Encoder to make sure that the picked register does not change until a Hit is registered. Finally, it is the responsibility of the End-of-column State Machine to make sure that the Write Command is not removed until the end of the BC0 cycle. It is by this mechanism that all Pixel Cells hit within the same BC0 clock period are associated with the same End-of-column Register.

During output, it is the responsibility of the End-of-column Logic to supply the column with the Read command and the token. When the token drops out of the bottom of the column, then the output is done and the State Machine changes from Full to Output Done. This state must be maintained until the next rising edge of the BC0 clock when the state will change to Empty.

The End-of-column Register, Comparators and Mask Register

The End-of-column Register is a set of six asynchronously resettable, positive edged D flip-flops. Their reset line is attached to the Data Reset signal. They latch the current BC0 number at the rising edge of the Full signal from the End-of-column state machine.

The End-of-column Comparators are just a set of exclusive or gates that ultimately yield an equal or not-equal signal. There are two comparators per End-of-column Set. One is for comparison of the BC0 number stored in the End-of-column Register with the Requested BC0 number. The result of this comparator is the RBC0eq signal that triggers an Output Command from the End-of-column State Machine.

The second comparator is for the comparison of the stored BC0 number with the current BC0 number. The result of this comparator is the CBC0eq signal that triggers a Reset Command from the End-of-column State Machine. Also associated with this comparator is the End-of-column Mask Shift Register. This is a six bit wide set of asynchronously resettable positive-edge triggered D flip-flops arranged serially as part of a scan path. A Logical 1 in any bit of the Mask Shift Register causes all of the current BC0 comparators to ignore that bit in their comparisons. By this mechanism, it is possible to arbitrarily set the reset time lag. For instance, if only the last two bits of the Mask Shift Register are set to a Logical 0 and all the other bits are set to a Logical 1, then the End-of-column State Machine will reset itself four BC0 clock cycles after it receives a hit. The Mask Shift Register is reset by the Program Reset signal, and it resets itself to all Logical 0 signals.

The RBC0eq signals from each End-of-column Set are combined to form the ColHasData (Column Has Data) signal. In turn, the ColHasData signals from each End-of-column Logic cell are combined to form the ChipHasData signal. This signal is used by the Chip Logic to determine if the chip must grab the external bus to transmit data.

A Detailed Description of a Hit from the End-of-column Logic's Perspective

If there are no free registers, then NoneFree is active, and the Priority Encoder is not pointing to any register. Moreover, the End-of-column Logic will set the Accept signal to a Logical 0. This will prevent any hits from being accepted by any Pixel Cells in the column.

If there is at least one free register, the Priority Encoder will be pointing to one and only one End-of-column Set. This End-of-column Set is now said to be Empty:Next, and the

State Machine in this Set will output a Write Command. As time passes, the current BC0 counter will advance in the Chip Logic, but as long as there is no hit, the state of the Priority Encoder and the Empty:Next End-of-column Set will not change.

When there is a hit somewhere in the column, the HFastOR will be activated. By this time, the hit Pixel Cell has already associated itself with the Empty:Next End-of-column Register, but, nevertheless, it will continue to assert the HFastOR until the Empty:Next End-of-column State Machine removes the Write Command. If other Pixel Cells in the column become active between this point in time to the next rising edge of the BC0 clock, those Pixel Cells will also associate themselves with the Empty:Next End-of-column Register. These newly hit Pixel Cells will also try to assert the HFastOR signal, but since it is already active, this will have no effect on the End-of-column Logic. Finally, the Priority Encoder, which also observes the state of the HFastOR signal, will be aware that a hit has occurred, and the next state of the Priority Encoder will be set up.

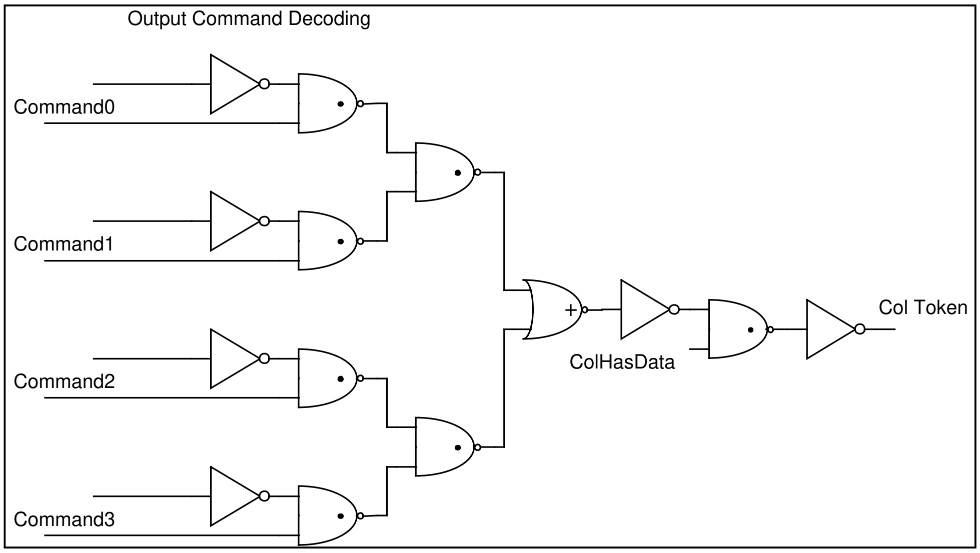
At the next rising edge of the BC0 clock, the Empty:Next End-of-column Set will become Full:Idle and it will remain in this state for at least one BC0 clock cycle. In this state, the End-of-column State Machine will change its command from Write to Idle. This will have two effects. First, it will cause the End-of-column Register to latch the current BC0 number. Second, it will cause the hit Pixels to stop asserting the HFastOR signal.

Also at this next rising edge of the BC0 clock, the Priority Encoder will point to another End-of-column Set, making it the new Empty:Next Set.

A Detailed Description of Outputting from the End-of-column Logic's Perspective

There is a second End-of-column State Machine called the Column Output Machine. It has four states. When the End-of-column Logic is not outputting data, it is said to be in the Silent State. When it is outputting data, it is said to be Talking.

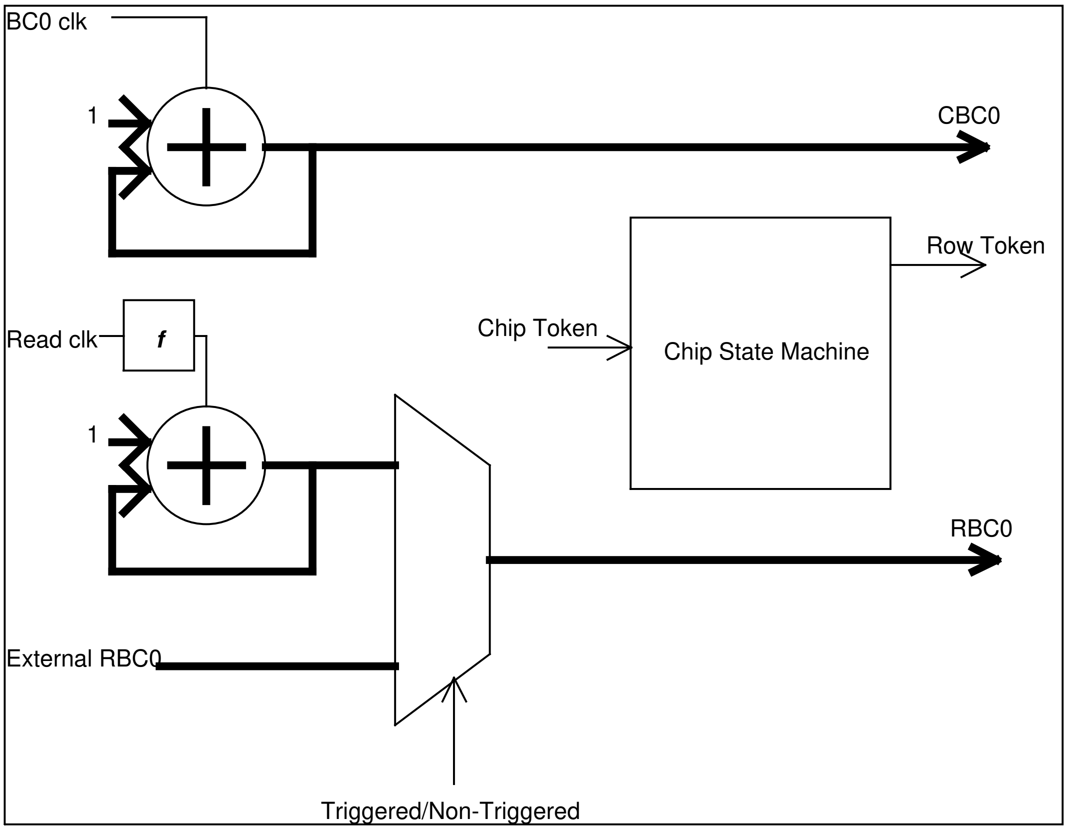
As the Requested BC0 number changes, in either Triggered or Non-triggered mode, it is continually being compared to all of the BC0 numbers stored in all of the End-of-column Sets in all of the columns simultaneously. If there is a match and that register is Full, then the End-of-column Set immediately switches to Full:Output. The Output Command is sent up the column to all pixels associated with this register. Meanwhile, the ColHasData signal is activated by the End-of-column Logic cell, and this, in turn, activates the ChipHasData signal.



The above figure shows how the Output command signals are decoded in the End-of-column Logic just as they are in the Pixel Cells. This decoded signal is combined with the ColHasData signal and then driven up the column as the Column Token. This somewhat elaborate scheme for generating the token ensures that the Column Token cannot arrive at a Pixel Cell before that Pixel Cell realizes that it needs the Column Token.

At this point, the End-of-column Logic waits for the arrival of the Row Token. This Row Token is created by the Chip Logic when it sees the ChipHasData signal. When the Row Token arrives, the Column Output Machine switches from Silent to Talking, and the Read Clock is permitted to pass up the column to the Pixel Cells. Recall from the Pixel Cell section that the pipelined Column Token passing logic will pass the Column Token to the first Pixel Cell with data, but that that cell will not take the bus until the next rising edge of the Read Clock. The action of the Column Output Machine prevents simultaneous access of the data bus by multiple columns.

When the last Pixel is read, the RFastOR returns to a Logical 0, and several things happen. First, the End-of-column State Machine switches to the Output Done State. Second, the Column Output Machine passes the Row Token to the next column with hit Pixel Cells in a manner similar to the pipelined passing of the Column Token. Third, ColHasData becomes inactive. This last event facilitates the release of the ChipHasData signal that, in turn, allows multiple requested BC0 numbers to be reviewed in each BC0 cycle.

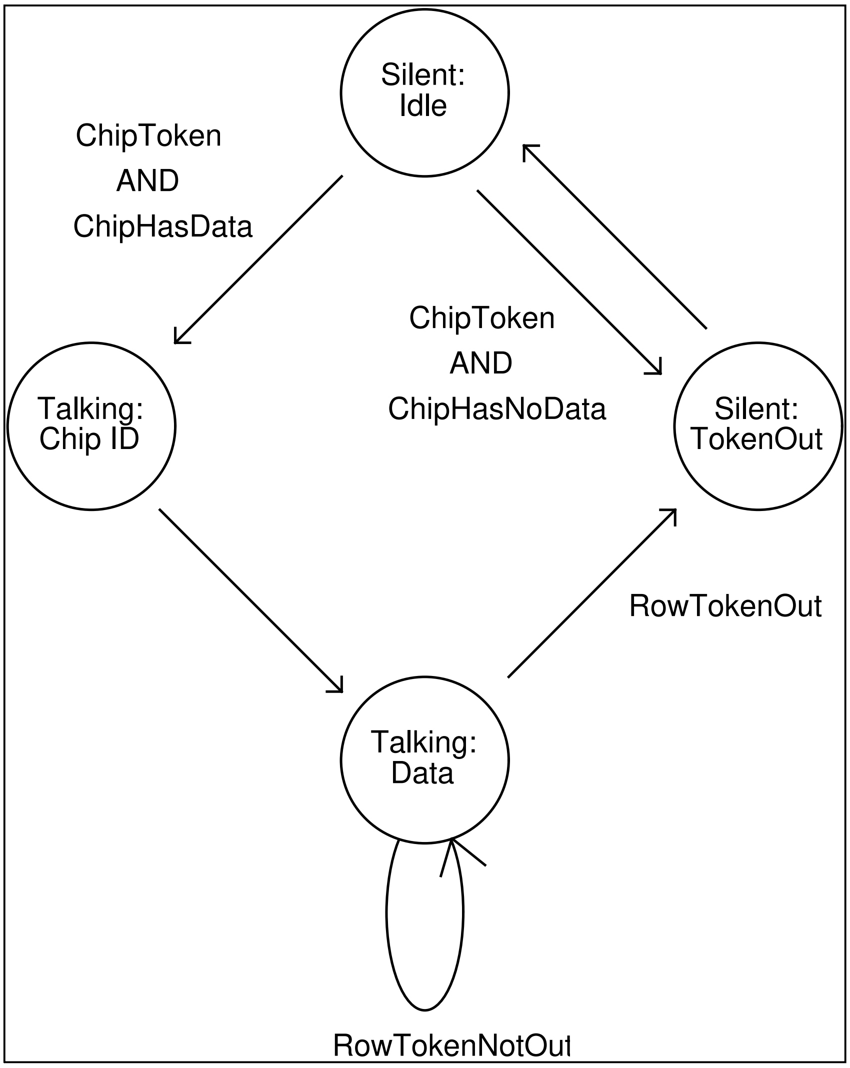


The Chip Logic

The above figure shows the basic layout of the Chip Logic. It consists of two counters, a multiplexer and a state machine. The first of the counters is the current BC0 counter that is nothing more than a simple counter. It changes state at the rising edge of the BC0 clock, and at the Data Reset signal, it resets to zero.

The second counter is more complicated. First, on the Data Reset signal, it resets to 2. It changes state on the rising edge of the Read Clock. However, it only changes state under certain conditions. First, it does not change state if the ChipHasData signal is active. Second, it must remain a certain distance from the current BC0 number. Therefore, every time it changes state, it adds its value to a user defined RBC0 Lag Number. If the Requested BC0 number plus the RBC0 Lag number equals the current BC0 number, then the RBC0 number is not incremented. Of course, if the chip is set to Triggered mode, then the RBC0 number is generated externally as shown in the figure.

The last element of the Chip Logic is the Chip State Machine. Its state diagram is shown



below. In essence, it is the same as the Column Output Machine with a few minor modifications. First, the Silent:TokenOut State was added because the Chip Token must always be passed even if there is no data available. Second, the Talking state was divided into Talking:ChipID and Talking:Data. This is to accommodate the requirement that it output its Chip ID and the BC0 number before it begins to output data. This is ultimately to help the outside world route the data as necessary.

Verilog Simulations

The following Verilog simulations are all performed on a 4x4 matrix of Pixel Cells. Each column has its own End-of-column Logic, and the matrix itself has its own Chip Logic. All circuits have been modeled to the transistor level, including the Fast OR circuits (HFastOR, RFastOR and ChipHasData). Delays have been specified for all primitives such as nand, nor, and not. Where required, such as in to Fast ORs and the token passing logic, they have been explicitly specified. Therefore, the timing results as determined by Verilog are reasonably accurate.

The Chip ID is preset to 02. It can be found in the first two hexadecimal bits in the first word output by the matrix. The second two hexadecimal bits in the first word are the BC0 number of the event. The first two hexadecimal bits of the remaining data words are the column ID. The four column IDs used were 01, 02, 04, and 08. The second two hexadecimal bits in the remaining data words are the row IDs of the pixels. The four Row IDs used are 33, f0, 55, and 0f.

The first four simulations show single pixel hits. The first is a hit to the first column, first row. The second is a hit to the second column, second row. The third is a hit to the third column, third row. Finally, the fourth is a hit to the fourth column, fourth row.

The second set of simulations show multiple hits in one column. In fact, each simulation shows one full column being filled.

The third set of simulations show multiple columnar hits. The first shows one hit pixel per column. The second shows all four pixels hit in all four columns.

The fourth set of simulations shows multi-time slice hits. The first shows two different pixels hit in two successive time slices.

The second simulation was a pleasant surprise. It shows all sixteen pixels hit in two time slices. The first time slice is successfully output. All seventeen data words make it out. However, the second time slice outputs only ten words – all of the first column, all of the second column and the first two Pixel Cells of the third column. At first, it was believed that there was some sort of error until it was realized that the second hit occurred during the readout of the first time slice. Therefore, the as the simulation indicates, the last two Pixel Cells of the third column and the entire fourth column were full during the hit. Consequently, the pervious hit protection of the Pixel Cells prevented the new hits from overwriting the old.

The third simulation is a repeat of the second simulation with the second sixteen hits arriving somewhat later. As the figure indicates, all seventeen data words make it out for both time slices.