Register-Like Storage Block Used as Histograms, Cluster Buffers and Hough Transform Accumulators for HEP Trigger Systems

Jinyuan Wu

Abstract—In high energy physics experiment trigger systems, block memories are utilized for various purposes, especially in binned searching algorithms. In these algorithms, the storages are demanded to perform like a large set of registers. The writing and reading operation must be performed in single clock cycle and once an event is processed, the memory must be globally reset. These demands can be fulfilled with registers but the cost of using registers for large memory is unaffordable. Another common requirement is the boundary coverage feature during reading process. When a memory bin is addressed, the stored contents in the addressed bin and its neighboring bin must be output simultaneously. In this paper, a register-like block storage design scheme fulfilling these requirements is described. The implementation and test results are presented. Practical applications of the register-like block storage as cluster buffers, histograms and Hough transform accumulators are discussed.

Index Terms—Trigger System, FPGA Applications, Hough Transform

I. INTRODUCTION

IN modern high energy physics experiments trigger systems, data are usually reorganized in binned memories, or cluster buffers. Consider a multilayer detector as illustrated in Fig. 1, hit data from each detector layer arrive in random order and are to be stored in a binned memory for future reading out.

![Fig. 1. An application of block memory bins for high energy physics experiment trigger system: BIN_NB is an integer representing the bin number which is usually calculated from the coordinate of the detector elements. Each column in memory block above represents a long memory word that is addressed by the BIN_NB. The memory word is divided into several bit fields (one hit per field) so that data from multiple hits with the same BIN_NB can be stored.](image)

In this example, charged particle tracks pass through several detector layers. The data from the particle hits are to be clustered together so that tracks can be reconstructed for further trigger algorithm. We consider a sub-process of using a track segment candidate based on the hits from the first two detector layers to find the hit data on the third layer.

In the first step of this sub-process, detector raw data are stored in the memory bins indexed by the bin number BIN_NB. A bin in this example represents a range of coordinate on the third detector surface and in many cases, a bin may include several detector elements such as silicon strips, scintillating fibers or straw tubes. Therefore, in an event, it is possible to have multiple hits within a bin. At the design stage, a maximum number of hits allowed in a bin is usually chosen. An overflow exception handling scheme is also defined which will be discussed in later sections. Once all hits in an event are written into the memory, they are to be read out according to the location being pointed by the track segment.

A crucial requirement is the boundary coverage during the readout process. The track segment points to a location on the surface of the detector and an integer is derived from the location coordinate as the bin number (BIN_NB) which is used to address the memory. It is very common that the track segment points to a location near a boundary of a bin and the actual hit data may be stored either in the pointed bin or in the neighboring bin. It is usually necessary to bring hit data from both bins out for finer track fitting processes.

The trigger firmware processes data in event-based fashion, and an “event” in collider experiments is usually a beam crossing. The process takes three phases: (1) storing data or “booking”, (2) reading data and (3) refreshing the memory. The input hit data are first fed in one word per clock cycle and are stored into the memory bins. After filling up the memory with the data from an event, the trigger algorithm will search the data based on the index of the bins and read them out. Note that the searching process may not address all bins containing the hit data and it may also address empty bins. After reading process, a single clock cycle refreshing command is issued by the users and all memory bins will be effectively cleared to prepare for the next event. It is well known that regular block memories do

The author is with Fermi National Accelerator Laboratory, Batavia, IL 60510 USA (phone: 630-840-8911; fax: 630-840-2950; e-mail: jywu168@fnal.gov).


This manuscript has been authored by Fermi Research Alliance, LLC under Contract No. DE-AC02-07CH11359 with the U.S. Department of Energy, Office of Science, Office of High Energy Physics.
not support global reset. To fulfill this requirement, an event ID tagging scheme is used. The single clock operation and global refreshing schemes developed in our previous work [1-3] are combined into a unified scheme in this work.

The Hough transform scheme is a popular track segment seeding method [4-12], in which an array of accumulators is utilized to book a 2D “histogram” for each coming detector hit. Using logic elements in FPGA to implement the Hough transform accumulators would consume large amount of silicon resources. A very natural approach is to use block memories (instead of logic elements) inside FPGA to implement the accumulators to reduce resource consumptions [11-12]. However, after processing an event, how to refresh the block memories promptly to prepare for the next event is still to be studied which will be discussed in this paper. Although the Hough transform accumulators do not look like clustering buffers described above, intrinsically they share many common behavior features. Using register-like storage block to implement Hough transform accumulators will help to reduce silicon area usage significantly.

In this paper, the full structure of the register-like storage block is first discussed in Section II, followed by implementation and test results in Section III. In Section IV, we will describe several applications of the register-like storage block including cluster buffers, histograms and an actual implementation [13] of a Hough transform accumulator array for track segment seeding applications.

II. STRUCTURE OF THE REGISTER-LIKE STORAGE BLOCK

The detector data are fed into the trigger system and in the processing stages of the firmware, the data are usually fetched one hit per clock cycle. The memory block bins are to be updated as the data fetched in every clock cycle. While writing a data into a memory within one clock is not difficult, it is a challenge to update the memory location within a single clock cycle. To update a memory bin, the contents of the bin must first be read out and the new hit data is added or concatenated into the original data word to form a new data word which is then written back into the memory bin. The updating process takes several clock cycles to complete which requires a dual port memory with a reading port and a writing port and a suitably designed pipeline so that the data can be processed one hit per clock cycle. Note that once a hit to be filled into a bin is fed into the pipeline, another hit to be filled into the same bin can come as early as the next cycle. In this case, the first data has not been written back into the memory bin before the reading cycle of the second update process. This is similar as the read-after-write (RAW) hazard in contemporary microprocessor design. To solve this hazard, a data forwarding scheme [1-3] is utilized.

To fulfill the requirements of single clock updating, reading and refreshing operations, the functional block is organized as a set of pipelines. The block diagram of register-like storage block is shown in Fig. 2. It can be unfolded based on its operating logic for clarity as shown in Fig. 3.

Fig. 2. Block diagram of the register-like storage block

Fig. 3. Logic structure of the register-like storage block
During the process of data booking, data to be stored (DATA), the operating command (CMD) and the bin index (BIN_NB) arrive to the input ports of the pipeline at the same clock cycle. The bin index is selected to feed into port AA to read out the contents stored in the memory bin. After two clock cycles, the contents appear at port QA to be checked in the EVID Check Unit. Several bits in the memory words are assigned as event ID field, if the current location was written during the booking process in the current event, this field will store the current event ID. Otherwise, it will store the event ID from old events. The event ID is checked with the current event ID which is maintained in Refresh Counter Control & Pipeline block. If the stored contents are from the current event, it will be sent to the Data Update Unit for further process. If not, the stored contents will be ignored and all output bits can be optionally set to zeros.

In the Data Update Unit, the new input data (after appropriate pipeline delay) and the old contents in the memory bin are used to build a new data word. The updating algorithm can be chosen by the users depending on the application. For example, the old contents can be moved upward to higher bits and the new data will be inserted to the lower bits of the data word, which will allow the memory bin to store multiple hits. For another popular application, histogram booking, the new input data is always 1 (so it is unnecessary to implement the Input Data Pipeline), the old content is simply added by 1.

The output of the Data Update Unit is sent to the second memory port DB to write back to the memory bin. At the same time, the result is also sent to the Data Forwarding Unit. The Forward Control block checks input data bin number to determine whether the Data Update Unit should use the data read out from the memory or there is a newer data in the Data Forwarding Unit. The Data Forwarding Unit always contains the most current values of the memory bins that are just updated.

After data booking process, the data reading out process uses the same pipeline and the read address is delayed and sent to port AB and the contents appear at QB, which are checked by another EVID Check Unit.

The reason of delaying the read out process is to wait for the memory bins to be fully updated. In this arrangement, the last writing command can be immediately followed by the first reading command without losing a clock cycle, which is crucial for high luminosity trigger systems.

Note that during the reading process, output contents from memory port QA is not used for Data Update Unit as in the booking process. Therefore, the A port of the memory is free to be used to read out contents in any bin. While port AB is addressed with BIN_NB (after appropriately delayed in the pipeline), we generate a new integer BIN_NB + 1 or -1 to address AA. This way, the outputs from QA and QB contain raw data from two neighboring bins allowing suitable boundary coverage.

After the reading process, a single clock cycle REFRESH command is issued which overwrites a rotationally selected memory bin (after several cycles of pipeline delay) and changes the EVID for the new event (or beam crossing in some applications). The booking command for the next event can be issued immediately following the refresh cycle. The book, read and refresh processes for an event can be connected end-to-end together without any missing clock cycles.

It should be pointed out that during the booking and the reading processes, the logic position of the memory port A in the pipeline is different.

During the booking process, the port A is used to read out the old contents in a bin stored in the memory. Therefore, it is addressed with the BIN_NB immediately at the early stage in the pipeline. For reading process, however, the port A is used to read out data in the neighboring bin. In this case, it is addressed with a delayed version of BIN_NB +1 or -1 and its logic position in the pipeline becomes the same as the port B.

III. IMPLEMENTATION AND TEST RESULTS

The register-like storage block is designed, implemented, and tested in an Altera Cyclone V FPGA device (5CEBA4F23C7N). The actual top design interface block is shown in Fig. 4. The block uses 466 logic elements (ALM) which is about 3% of the target device.

In this demo design, the input hit data is chosen to be 8-bit for simplicity. The block RAM are divided into 256 memory bins with 36 bits each that can store up to 3 hits and a 12-bit event ID. The block is driven by a 250 MHz clock which is generated with a phase lock loop (PLL) inside the FPGA device. In every clock cycle, an 18-bit word (KDB[17..0]) containing a 2-bit command, an 8-bit data and an 8-bit bin number is fed into the block. The 2-bit command word instructs the block to operate one of the four processes: NOOP, Book, Read and Refresh.

In book operation, the input data is filled into the memory indexed by the bin number while in read operation, the data word is primarily ignored and only the bin number is used. In our design, bit 0 of the data word is used as the “half-bin” bit of the bin number which is used to indicate if the actual index is closer to higher or lower boundary of the bin. When this bit is 1, the higher bin is output from the QA port and when it is 0, the lower bin is output. (Note that QB port always output contents of the addressed bin).

Some operation examples of the register-like storage block are presented in Fig. 5. In the table, the second column lists the output words from the FPGA module showing the test results. Each line represents a 250 MHz clock cycle. In each clock cycle, the functional block may perform NOOP, Book, Read or Refresh operation. In the event shown, we have written data A3, B3 into bin number 03, A4, B4 into 04, A5, B5, C5 into 05 and so on during the Book processes. Note that in this example, multiple data words may be written into a memory bin and the
two words to be written into the same bin can be as close as in two adjacent clock cycles. With appropriately designed data forwarding unit, the functional block will update the addressed memory bin correctly regardless the number of clock cycles between the book operations.

Note that during the book operation, the memory outputs are ignored by the later stage. In actual implementation, the block RAM is set to output "NEW DATA" (i.e., showing newly written data) for writing operation and therefore, the data field MemDataA (part of QCKA port) actually shows non-relevant numbers are read from corresponding bins, which are ignored by the later stage. In actual implementation, the block RAM is set to output "NEW DATA" (i.e., showing newly written data) for writing operation and therefore, the data field MemDataA (part of QCKA port) actually shows non-relevant values such as A3 and B3 at line 17.

The Read command can be issued immediately after the last Book command. When a memory bin is read, the contents read out consist both the data written and the event ID (EVID) when the bin was updated. The stored EVID is compared with the current event ID and only if they are identical, as shown in line 21 to 27, the data bits stored in the bin is considered valid. If the stored EVID is different from the current event ID, the data in the bin will be treated as invalid and the bin is considered empty. For example, at line 34, some non-zero numbers are read from corresponding bins, which are ignored since they are left over from old events.

The last three columns contain output data from memory port A for boundary coverage. In this example, the memory bin at BIN_NB -1 is addressed in each read operation. Fig. 5 shows some examples of the operations of the register-like storage block.

Fig. 5. Some examples of the operations of the register-like storage block
For convenience of visualization, the bin number and data values written into the memory bins are used as X and Y coordinates in the plot. Each dot represents a stored data ranging 0 to 255 and the data points belonging to an event are connected.

Should we select all events booked into the register-like storage block, the plot would have a very chaotic looking. For clarity, we have chosen the current events, some recent events and some very old events (more than 256 events ago) in the plot.

After finish booking all data points, data in all 256 memory bins are read out and plotted as shown in Fig. 7.

All data points written during the current event are seen and the memory bins containing data from current event can be easily identified. These data points identified as belonging to the current event are plotted in the top portion of the plot for clarity (A constant 256 is automatically added to the date point in the spread sheet formular when a memory bin is identified containing data points of the current event).

Also, note that all data points written during the old events do not exist since they are erased in the refresh processes. Some points from the recent events may also be erased but many of them are not yet erased in the refreshing process, which are plotted in the lower portion of the plot.

The current event is identified by the event ID (EvtID) tag stored in the memory bin. Should the number of bits of the event ID is infinity, one would not need the Refresh command. When the event ID field contains finite bits, a Refresh command is used to prevent rolling over of the event ID field.

To study the refreshing process, several thousand events are processed in the register-like storage block and the lower 9 bits of event ID (EVID = EvtID & 0x1ff) in all 256 bins are plotted as shown in Fig. 8.

In this example, current event ID is 450 and the highest dots in the plot represent data being written in the current event.

In every event, the refresh operation overwrites 1 bin rotationally selected from 256 bins and these bins form a diagonal line. Therefore, the oldest event ID that exists in the entire memory is at most 256 events earlier than the current event. Some bins may contain newer event ID if they are used in the past 256 events which are dots above the diagonal line. If we store 9 or more bits as event ID, the counter rollover will not cause mistakes during the event ID comparison. Similarly, at least 10, 11 or 12 bits should be used if the block RAM has 512, 1024 or 2048 bins.

IV. NOTES ON SEVERAL PRACTICAL APPLICATIONS

The register-like storage block is a building block that can be utilized for various functions in high energy physics trigger or offline event selection systems. In this section, we discuss several examples of practical applications.

A. The Cluster Buffers

The firmware described in previous sections is essentially a cluster buffer. Data in an event are first written into bins in the cluster buffer and read out later. Depending on distribution of
the hits, more than one hit may be stored in a bin, but not all bins are written or read in an event.

![Fig. 9. The data update operation for the cluster buffer](image1)

One possible design of the data update unit in the cluster buffer is a data pusher as shown in Fig. 9. When a data word arrives and to be stored in the a memory bin addressed by the bin number (BIN_NB), the old data in the bin is first read out as indicated by 1 above. Next 2, the current data is saved into the lower bits while the old data are pushed to higher bits. The updated data field is then saved back to the memory block as shown in 3 above. In our design, we reserved memory space allowing up to three hits to be saved in a bin, but obviously the users may assign different capacities in their own applications.

During readout process, all data words in the memory bin are output to the later stage. If the event ID field of the memory bin is different from the current event ID, the bin is treated as an empty bin although the bin may contain data words from old events.

An issue that the designers must consider is the exception handling. In other words, in some unlikely cases, if too many hits are to be filled into a bin causing it to full, how the overflow should be handled. A simplest approach is to do nothing, and the newest data will be kept while the oldest data are pushed out of the storage. It is also possible to implement additional logic so that oldest data are kept while newest data are dropped. If it is necessary, the users are allowed to design more sophisticated exception handling logic.

The contents being stored in the cluster buffer can be actual hit data or indices to the hit data buffer in many cases.

### B. Histograms

Histograms can be utilized, for example, for online bin-by-bin calibration of time-to-digital convertors (TDC) [14] or other digitizers. A histogram can be viewed as a set of counters, one counter per bin, which increases by 1 when the a entry is to be accumulated into a bin during booking phase. The data updating operation of an online histogram unit is shown in Fig. 10.

![Fig. 10. The data update operation for online histograms unit](image2)

Of course, one may use logic elements in FPGA to implement an array of counter as online histogram but that will consume large amount of silicon resource. The register-like storage block can be turned into an online histogram unit with a minor modification of the data update unit. Simply put an adder in the data update unit and add the input number by 1 will fulfill the logic function for histogram.

In many applications, an exception of overflow in each bin must be handled appropriately in the histograms. Usually, let the adder stop adding the input number by 1 when the input is bigger than certain value would be sufficient.

Implementing multiple dimension histograms is straightforward. For example, a 2048 bin 1D histogram can be used as a 32-column x 64-row 2D histogram by simply assigning address lines. The lower 5 address lines are assigned to the first dimension which are used to index the 32 columns. The higher 6 address lines are assigned to the second dimension to index the 64 rows.

In some situations, the online histogram may have less strict demands than the register-like storage block we discussed. For example, the incoming data may not come every clock cycle but rather, it may take at least a few clock cycles for the front-end circuit to produce one. In some applications the histograms may not need to be reset within a single clock cycle but rather, the users have enough time to readout the entire content of the memory and write zero into all locations. In this case, the histogram circuit can be simplified by eliminating some functional block in the register-like storage block.

### C. The Hough Transform Accumulator Array

The Hough transform is a useful tool for track segment seeding in multi-layer detectors. In the Hough transform, a point (a hit on a detector layer) in physical coordinate space is transformed into a curve (or straight line in many cases) in track parameter space. Hits on different detector layers generated by the same track map to different curves on the parameter space, and the curves intersect to a spot representing the same set of track parameters.

In each detector/track configuration, usually there are several possible choices of Hough transform parameters and to certain extent they are mathematically equivalent. However, an appropriate set of parameters will help to optimize performance in practical implementations and an example is shown in Fig. 11.

![Fig. 11. An example of parameter selections in Hough transform](image3)

In a typical barrel detector in a solenoidal magnetic field,
track curvature or inverse transverse momentum \((qA/P_t)\) and track initial angle \((\phi_t)\) are usually chosen as the Hough transform parameters. The hits in different detector layers are mapped into straight lines in the Hough transform space as shown in top-right diagram in Fig. 11. The inclined angles of these straight lines in actual implementation are determined by the bin width of the parameters and they are confined within 45 to 90 degrees. In this situation, the intersections of these straight lines are blurred that causes track segment seed performances less optimal.

If we replace the parameter \((\phi_t)\) with the coordinate of hit on layer 2 \((\phi)\), the intersection in the Hough transform space will be a lot sharper as shown in lower-right diagram in Fig. 11. This selection will help to reduce fake track segment rate when an event contains large number of hits. Similarly, the parameter choice of \((\phi_s)\) given in Reference [11-12] helps to improve the segment seeding performance.

In hardware implementation, an array of accumulators is needed and the accumulators in the cells along the curve will increase by 1 for an incoming hit. Ironically, the accumulators can be implemented with FPGA logic elements, but it would consume large silicon area and increase the system cost. The register-like storage block is a suitable alternative scheme featuring significant lower resource usage. We will discuss an example taken from our recent work [13] as shown in Fig. 12.

The Hough transform block is a component in an exercise of implementation of a 3D track segment seeding engine. The detector model has the same diameters and lengths as the CMS Outer tracker except the tilting of the silicon sensor modules is not included for simplicity. The Hough transform block processes hits in r-z view with parameters \(z_0\) which is the z coordinate of the track starting point along the beam line and \(z_{375}\) which is approximately the z coordinate of the layer 2 hit. The choice of parameter \(z_{375}\) ensures that a sharp intersection of straight lines corresponding to layer 1, 2 and 3 hits in the Hough transform space, as discussed above. Note that the straight line corresponding to a layer 2 hit is a 90-degree vertical line in the Hough transform space.

The actual firmware implementation based on the register-like storage block is shown in Fig. 13.

In the diagram above, two sets of Hough transform space are implemented each handling hits from layer 1 and layer 3 of the detector. Each horizontal row in the diagram represents a physical register-like storage block which contains 256 bins to cover the \(z_{375}\) dimension. Each register-like storage block represents a \(z_0\) bin and in our case, we have 10 \(z_0\) bins total for each Hough transform space.

For each hit in layer 1 or 3, an integer coordinate \(nZ_1\) or \(nZ_3\) is generated. For each \(nZ_1\) or \(nZ_3\), all RAM blocks for different \(nZ_0\) are written at different addresses \((nZ_{375})\), which corresponding to a straight line in the parameter plane. The addresses are pre-calculated reflecting the slopes of the straight lines and are stored in a set of ROM lookup tables. The scheme of using lookup tables will accommodate any shapes of the curves beyond straight lines to maintain flexibility in case the Hough transform engine is to be used for more complicate detector geometries as in the examples described in References [6-10].

The contents of the cells in the Hough transform space are bit patterns as shown in Fig. 14.

The payload in each cell is a 128-bit word which represents a bit pattern spanning in \(\phi\) dimension. For any hit in layer 1 or layer 3, the \(z\) coordinate is used to address the RAM location and the \(\phi\) coordinate is used to set a particular bit in the 128-bit words for all the RAM blocks. Note that this bit setting is an update operation meaning that while the selected bit is set, the other bits remain unchanged.

During the readout phase, hits in layer 2 are cycled through the Hough transform block. The \(z\) coordinate of the layer 2 hits is converted into addresses \((nZ_2)\) of the RAM blocks. The conversion can be done in the ROM lookup tables also for maximum flexibility, but in our application, \(nZ_2\) is nearly the same as \(nZ_{375}\) and therefore the RAM blocks are addressed.
directly by nZ2. As pointed out earlier, a layer 2 hit represents a vertical line in the Hough transform space.

For each layer 2 hit, two sets of 128-bit patterns are output from the two Hough transform spaces. Each set contains 20 patterns: 2 patterns for each physical register-like storage block and 10 blocks representing 10 z0 bins. The bit patterns with the same nZ0 (plus boundary overlap) from the two Hough transform spaces are bough together to feed the Tiny Triplet Finders. The Tiny Triplet Finders will search for the coincidence in the Φ dimension which will be discussed in other papers. Note that the Hough transform is not used to complete the full segment seeding process, but rather, it is merely a feeder to another stage (Tiny Triplet Finder) so that constraints of track segments in 3 dimensions can be utilized for better fake segment rejection ratio.

V. CONCLUSIONS

A register-like storage block is designed, implemented and tested. The design has been further verified in a practical application of an exercise implementation of a track segment seeding engine. Single clock writing, reading and refreshing performance allows wide applications in various high energy physics trigger systems. The firmware scheme allows designers to insert more pipeline stages into complex combination logics. In our design, both the EVID Check Unit and the Data Update Unit are pipeline stages, respectively. This way, the operating speed of the block RAM resource in the FPGA device used. This scheme does not rely on special features of the FPGA memory operating mode such as read-before-write or byte enables, etc. and therefore porting the firmware between FPGA families becomes possible and simple.

The cluster buffers, histograms and Hough transform accumulators are several examples of the applications of the register-like storage blocks. The register-like storage blocks are intended to be building blocks in online trigger systems and offline event pre-processing accelerators. Features of the register-like storage blocks, especially single clock cycle updating and single clock cycle refreshing allow users to implement event processing functions with high event rate. Eliminating the bottle-neck of memory resetting time between two events inside FPGA, for example, helps to reduce number of copies of processing functional blocks that otherwise must be massively duplicated in the system.

REFERENCES


