FERMILAB-PUB-06-063-E

The Application of Tiny Triplet Finder (TTF) in BTeV Pixel Trigger

Jinyuan Wu, M. Wang, E. Gottschalk and Z. Shi

***Abstract*— We describe a track segment recognition scheme called the Tiny Triplet Finder (TTF) that involves grouping of three hits satisfying a constraint such as forming of a straight line. The TTF performs this O(n^3) function in O(n) time, where n is number of hits in each detector plane. The word "tiny" reflects the fact that the FPGA resource usage is small. The number of logic elements needed for the TTF is O(Nlog(N)), where N is the number of bins in the coordinate considered, which for large N, is significantly smaller than O(N^2) needed for typical implementations of similar functions. The TTF is also suitable for software implementations as well as many other pattern recognition problems.**

***Index Terms*—Trigger, Pattern Recognition, Tiny Triplet Finder, TTF, FPGA Firmware**

I. Introduction

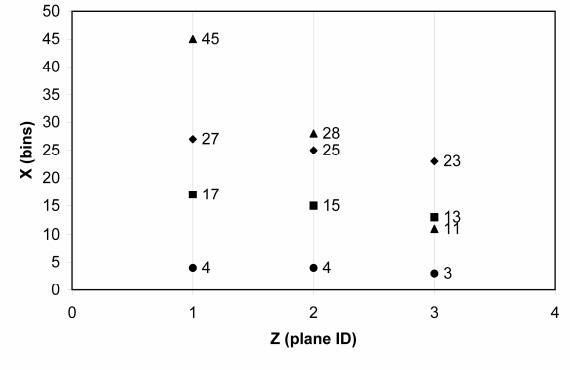
TRACK segment finding is an essential process in many trigger systems for high-energy physics experiments. For example in the Fermilab BTeV[1] trigger system[2][3][4], we need to identify track segments from the coordinates of pixel detector hits from three adjacent detector planes forming a straight-line segment in the non-bend view. For a given track segment, the following relationship holds:

$$u_A + u_C = 2u_B$$

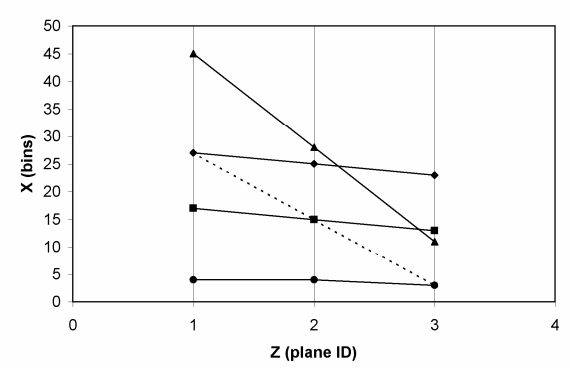
Where u_A, u_B and u_C are the hit coordinates on planes A, B and C in the non-bend view. Such segments consisting of three hits are referred to as "triplets" [2] (See Fig. 1).

In the BTeV detector, the interaction points are distributed along the beam axis in a wide range. There are two free parameters for a track segment even in the non-bend projection. A possible track segment is not formed until at least three hits are aligned.

Therefore the triplet finding process in BTeV is different from another type of straight track segment finding in other high energy physics experiments when the interaction point is known. There is only one free parameter for a track segment in the non-bend projection in this case. A possible track segment is formed after two hits and the known interaction point are aligned. The process is significantly simpler than the BTeV triplet finding process.



(a)



(b)

Fig. 1. Triplet finding: The interaction points are distributed along the beam (Z) axis. Equally spaced pixel detectors are placed in Z=constant planes. The ZX is the non-bend view. The dashed line represents a fake triplet.

Straightforward software implementation of the triplet finding function would require $O(n^3)$ execution time, where n is number of hits per plane, in order to examine all possible combinations of three hits using three layers of nested loop. In hardware implementations, the execution time can be reduced to $O(n)$, the time required to fetch the data. This execution time reduction is accomplished by "unrolling" two layers of loops, which consumes significant amount of silicon resources in the device. The number of logic elements needed for typical triplet finding implementations is $O(N^2)$ where N is the number of bins that each plane is divided into.

In this article, we describe a new algorithm that performs the triplet finding function, which we refer to as the Tiny Triplet Finder (TTF) [5]. We also describe sample hardware

Manuscript received June 2, 2005, revised March 13, 2006. This work was supported in part Operated by Universities Research Association Inc. under Contract No. DE-AC02-76CH03000 with the United States Department of Energy.

J. Wu, M. Wang, E. Gottschalk and Z. Shi are with Fermi National Accelerator Laboratory, Batavia, IL 60510 USA (phone: 630-840-8911; fax: 630-840-2950; e-mail: jywu168@ fnal.gov).

implementations of the TTF using low cost FPGA devices. Logic element usage in TTF implementation is $O(N\log(N))$ which is significantly smaller than $O(N^2)$ when N is large.

After the triplets are found, the triplets in one part of the detector are matched with triplets found in other part of the detector. The process of matching two data items require $O(n^2)$ execution time in software and it can be reduced to $O(n)$ in hardware by unrolling one layer of the loops, with logic element usage of $O(N)$. The "Hash Sorter" [6] or other schemes can be used for this type of applications.

II. Principle

Consider three equally spaced detector planes in the non-bend view as shown in Fig. 2. We first divide the two outer detector planes, Plane_A and Plane_C, into N bins (N = 64 in this example), choosing a bin as unit of the coordinate in the non-bend view and rounding to an integer. Since the detector plane usually contains far more pixels than 64 in the projection being considered, the binning is actually merging several pixels together. For example, to merge 1024 pixels into 64 bins, 16 pixels are merged together. To convert the pixel number into the bin number in this case, simply shift the pixel number by 4 bits.

In general, there exist N^2 possible track combinations, or "roads" in this configuration. A "road" is defined here as a line segment passing through one of the N bins in each of Plane_A and Plane_C. Directly implementing all possible combinations using either logic elements or content addressable memories (CAMs) would require a large amount of silicon resources.

In the Tiny Triplet Finder, two register arrays, BitReg_A and BitReg_C, are used to record the hits in the detector planes. When a hit coordinate from a detector plane is input, one of the 64 bits in the register array corresponding to its position is set. After all hits in Plane_A and Plane_C are recorded, the algorithm cycles through each hit from Plane_B. In the special case when the hit is at the mid-point of Plane_B (see the top of Fig. 2), there are 64 possible track combinations or roads. Each possibility is checked through bit-wise coincident logic between the bit patterns recorded in BitReg_A and BitReg_C. If a pair of corresponding bits in BitReg_A and BitReg_C are both set, e.g., (0, 63), (1, 62) or (2, 61), etc., the bit-wise logic will output the pattern of the matching pair(s) corresponding to a possible track segment passing through the hits in Plane_A, Plane_C and Plane_B. The bit-wise coincident logic is primarily a bit-wise AND of the patterns in the two registers. In a real implementation, a bit-wise OR with the neighboring bits in one pattern may first be performed to cover boundaries. An example of the bit-wise coincident logic is shown in the bottom of Fig. 2.

For hits that are not at the mid-point of Plane_B, the bit-wise coincident logic is identical, except that the positions of the bit patterns representing the hits on Plane_A and Plane_C are shifted relative to each other by an amount determined from the coordinate of the hit on Plane_B (see the second and third configurations of Fig. 2). The constraint for the triplet can be rewritten in the following form:

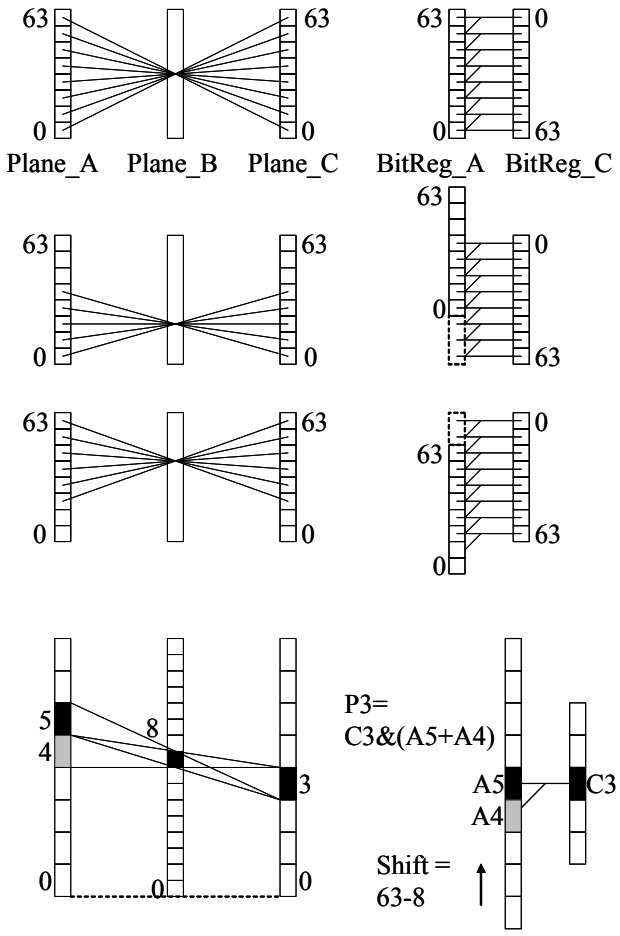


Fig. 2. Principle of Tiny Triplet Finder: Plane_A and Plane_C are divided into 64 bins, respectively. For any hit in Plane_B, there are up to 64 possible "roads" (left). The correlation of the roads is implemented as bit-wise coincident logic (bit-wise AND-OR) between two bit maps: BitReg_A and BitReg_B (right). Note that the two bit maps have reversed bit order. The two bit maps are shifted relative to each other for different Plane_B hits.

$$u_A = -u_C + 2u_B$$

It can be seen that the relative shift between the bit patterns is $2u_B$. In a practical implementation, the unit of the Plane_B hit coordinate is chosen so that $2u_B$ is an integer from 0 to $2N$-1 (To merge 1024 pixels to 128 bins in our example, simply shift the pixel number by 3 bits.). It can also be seen that the orders of the two bit patterns relative to each other should be reversed due to the negative sign between u_A and u_C.

Additionally, hits from different tracks as well as noise hits can also satisfy the coincident logic to produce fake tracks. The simplest way to deal with the fake tracks is to encode and output all of them and perform arbitration at a later stage. The users may also use a priority encoder to choose a particular track depending on the physics requirement of the experiment.

In the Tiny Triplet Finder, only N (64 in this example) combinations are implemented in the bit-wise coincident logic instead of N^2 (64 X 64 = 4096) combinations. Taking advantage of symmetry, all possible combinations can be achieved by shifting the bit patterns.

In the time domain, the total execution time is taken up by the following processes:

1. Setting the bit patterns BitReg_A and BitReg_C.
2. Looping over hits in Plane_B, shifting the bit pattern in BitReg_A, performing the bit-wise coincident and decoding matching pair(s) found.

The processes take approximately $2n$ clock cycles to execute making them essentially $O(n)$, although small non-linear contributions exist when more than one pair is found by the bit-wise coincident logic.

Generally speaking, the probability of creating fake triplets that cause the non-linear contribution increases as n increases, and decreases as N increases. When number of bins N is sufficiently large and number of hits n is small, the non-linear contribution should be small. This is true for all triplet finding schemes and the results studied for other schemes remain valid here also.

Ignoring the small non-linear contributions, we see that the TTF unrolls two layers of loops so that an $O(n^3)$ process can now be executed in $O(n)$ time.

The acceleration is accomplished through the use of the bit-wise coincident logic that simultaneously finds all matching hits on Plane_A and Plane_C for each hit on Plane_B in a single operation, making the process time proportional to the number of hits n on Plane_B.

III. FPGA Implementations of the Tiny Triplet Finder

The block diagram of the Tiny Triplet Finder implemented in an FPGA device is shown in Fig. 3.

A. Bit Array Filling

As the hit data from Plane_A and Plane_C are fetched from input FIFO's, a bit corresponding to each hit is set in the BitReg blocks. The resulting hit patterns are presented at the output ports on buses AQ and CQ. As mentioned earlier, the bit order of CQ is reversed. Meanwhile, the full hit data are stored into memory buffers called "Hash Sorters" [6] for fast retrieval later. For simplicity, one may think of the Hash Sorters as memory areas that are each divided into 64 bins. When a hit sets a bit in the BitReg register array, the full hit data are written into the corresponding bin in the Hash Sorter.

B. Looping B Hits and Shifting Bit Pattern

After all hits from Plane_A and Plane_C have been written into the Hash Sorters, the hits from Plane_B are fetched from the input FIFO. The coordinate of each Plane_B hit is used to determine the relative shift distance between the two bit patterns AQ and CQ. The shifter shifts the bit pattern AQ by this amount and presents the shifted pattern at port A2Q. The full hit data from Plane_B are also stored for later retrieval in a buffer which can either be a hash sorter or a regular output FIFO.

The shifter is implemented in a two-stage pipeline to increase operation frequency. Although the shifter requires a relatively large number, i.e., $O(N\log(N))$ of logic elements compared to other blocks in this design, it is still much smaller than typical implementations where $O(N^2)$ logic elements are needed.

C. Bit-wise Coincident Logic

The bit pattern CQ and the shifted pattern of AQ, A2Q, are

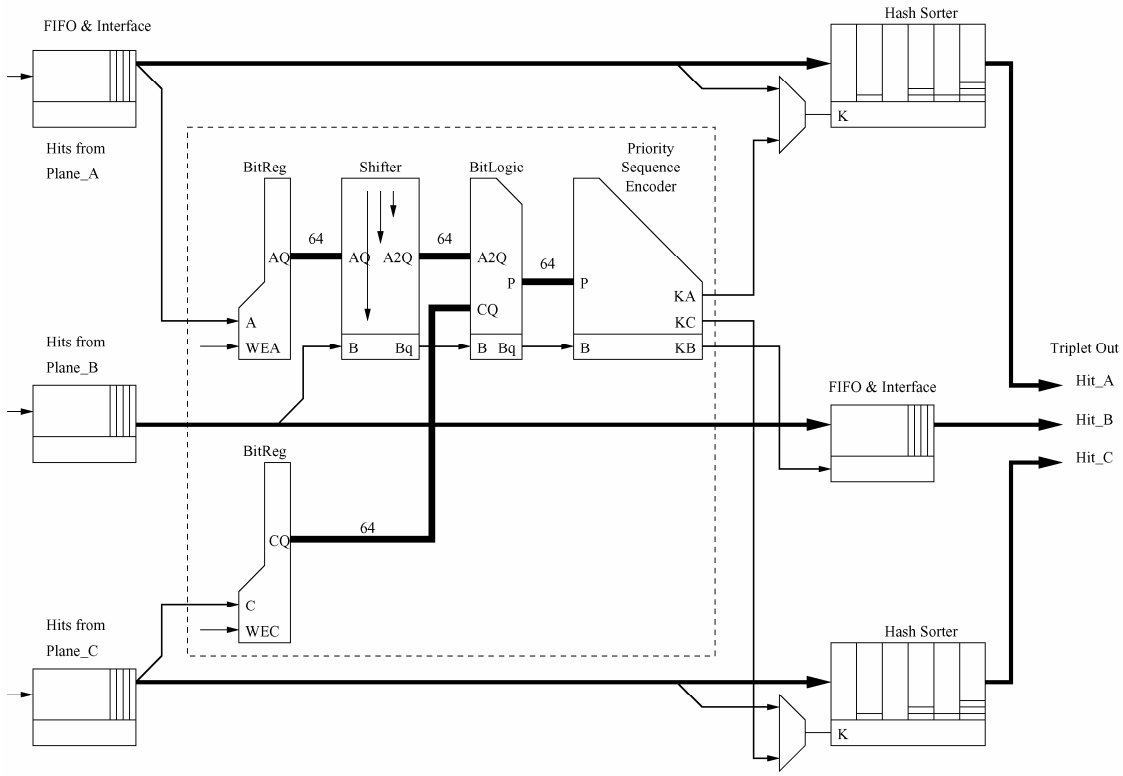


Fig. 3. Block diagram of triplet finding processor: The Tiny Triplet Finder is shown in the dashed box.

sent to the "BitLogic" block in which the bit-wise coincident logic is performed. The coincident logic is essentially a bit-wise AND. The OR logic among the neighboring bits in A2Q is included to cover the boundaries. See the bottom of Fig. 2 for an example.

Any non-zero bit in the resulting bit pattern P indicates a found triplet. The location of this bit represents the coordinate of the Plane_C hit belonging to the triplet. The coordinate of the Plane_A hit can be derived from this location and the distance of shift.

D. Priority Sequence Encoder

The locations of the non-zero bits are encoded in the "Priority Sequence Encoder" block which can accommodate situations with more than one triplet. Normally there is only one non-zero bit in pattern P and the encoder outputs the location of the bit. If there are two or more non-zero bits, the encoder changes a signal (EN in Fig. 4) to halt earlier pipeline stages, allowing the locations of all the non-zero bits to be reported sequentially.

This block is also implemented as a pipeline. Although it takes 6 clock cycles to encode the non-zero bit(s) in P, the block accepts one P pattern during each clock cycle, as long as the pipeline is not halted.

IV. Test Designs and Silicon Resource Usage

We have test compiled the Tiny Triplet Finder with N=64 and N=128 bins in an Altera Cyclone device (EP1C4) [7].

Results of the full simulation of the Tiny Triplet Finder are shown in Fig. 4. The simulation uses hit coordinates given in Fig. 1(a) as an example. The coordinates for Plane_B are multiplied by 2 to obtain the shift distance B. All four real triplets in this example are found in addition to a fake triplet which also satisfies the triplet condition KA + KC = KB. The fake triplet is represented by the dashed line in Fig. 1(b) with hits at X = 27, 15 and 3, which corresponds to KA = 27, KB = 30 and KC = 3 in Fig. 4.

The outputs of the Priority Sequence Encoder, KA, KB, and KC, are the bin numbers where the original hit data are stored in the Hash Sorters (or FIFO for Plane_B hits). These numbers are used as addresses to read out the hit data in the corresponding bins to send to later stages for further processing.

If there is more than one hit stored in a bin, the Hash Sorter will output all the hits in the bin so that later stages can make better choices. In this case, the pipeline in earlier stages will be halted, allowing multiple hits to be read out.

Another interesting point shown in this example is that we have found a triplet (KA=5, KB=8, KC=3) corresponding to the input (A=4, B=8, C=3). One of the input coordinates is off by 1 bin due to a boundary effect and/or a round-off error. Our bit-wise coincident logic covers this kind of difference. To trace back the original hits in the Plane_A at bin 4, the hash sorter will check both bin KA and KA-1, i.e., both bin 5 and bin 4 in this example.

The compilation results are shown in Table I for all functional blocks shown within the dashed box in Fig. 3. Clearly, the Tiny Triplet Finder can easily be accommodated in currently available middle-sized FPGAs.

The resource usages for two other typical implementations are also shown for comparison. The first implementation uses Content Addressable Memories (CAM) which can be implemented fairly efficiently with Altera Embedded System Blocks (ESBs) [8]. For this case, the silicon usage for 64 X 64 =4096 roads has been calculated without considering boundary effects and other supporting logic.

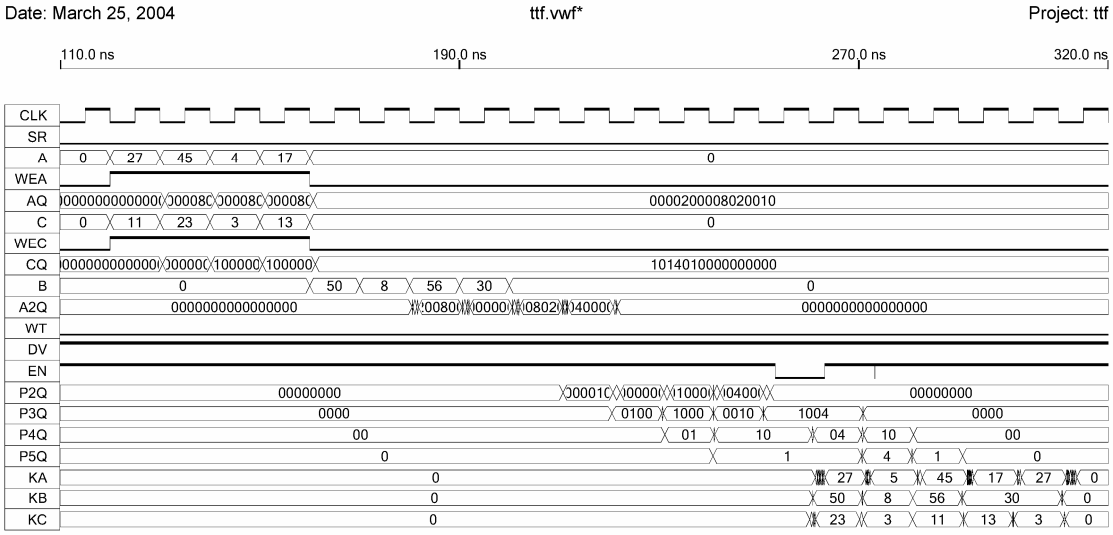


Fig. 4. Simulation results of the Tiny Triplet Finder: Signals CLK and SR are the 100MHz system clock and system reset. Signals A, WEA, AQ, C, WEC, CQ, B, A2Q, KA, KB and KC are as shown in Fig. 3. The signals AQ, CQ and A2Q are 64-bit bit patterns in hexadecimal representation. WT (wait) and DV (data valid) are input signals not discussed in this paper. The signals EN and P2Q to P5Q are internal ones in the Priority Sequence Encoder. As a consequence of finding more than one possible triplet, signal EN becomes low to stop the pipeline inside and before the Priority Sequence Encoder for a clock cycle to allow the extra triplet being output.

TABLE I
SILICON USAGE OF TRIPLET FINDER IMPLEMENTATIONS

Devices: Price: (05/2005)	**EP1C4** $32	**EP2A40** $1560-$3300	
	Logic Cells (4000)	Logic Cells (30,855)	Embedded System Blocks (160)
TTF (64 bits)	944 (23%)	944 (3%)	-
TTF (128 bits)	1681 (42%)	1681 (5%)	-
CAM using ESB (64 bits)	Not fit		128 (80%)
Hough Trans. (64 bits)	Not fit	16384 (53%)	
Device: Price: (05/2005)	**xc2v1000** $213		
	Logic Cells (10240)		
TTF (64 bits, using RAM16x1)	456 (4%)		

Another implementation uses the Hough transform scheme [9]. The number shown includes only the 2-D histogram, assuming each bin can be implemented with 4 logic cells. Decoder and other supporting logic are not included.

Since these two other implementations do not fit in the EP1C4 device, they were accommodated with a 7 times larger EP2A40 APEX II device [8].

Furthermore, as the bin number increases from 64 to 128, the logic cell usage of the Tiny Triplet Finder increases only by about a factor of 2 while for the other two implementations an increase by a factor of 4 is anticipated.

In Table I, we also listed the resource usage of a 64-bit TTF implemented using distributed RAM devices available in the FPGA devices from Xilinx [10]. The details are discussed in the next section.

V. IMPLEMENTATION USING DISTRIBUTED RAMS

In this section, a more efficient implementation using distributed RAMs is described.

In today's main stream FPGA devices, lookup tables are used to implement combinational logic. The lookup tables are small RAMs, usually 16 locations by 1 bit in size. In some device families like Xilinx Virtex-II [10], the users are allowed to write and read the RAM which provides possibilities to implement functions more compactly. As an example, the implementation of the bit registers and shifters in the TTF using distributed RAMs is described in this section.

In Fig. 5, the bit maps for a 64-bit TTF are shown. Two register arrays corresponding to Plane_A (left) and Plane_C (right) are formed using 64 distributed 16x1-bit RAMs for each array.

While filling hits for Plane_A, each RAM is given a rotational address, i.e., the address of a RAM differs from the address of the next RAM by 1. For any hit in Plane_A, up to 16 RAMs, 7 above, 8 below the hit position (marked with an arrow for each hit in Fig. 5), are enabled to be written. The bit being filled in the RAM at the hit location is bit 7, while in the RAMs above and below the hit location are bits 6-0 and bits 8-15, respectively. The net effect is that each hit sets up to 16 bits in the array as shown in the left bit map. Although up to 16 bits are written for each hit, only one clock cycle, (not 16) is needed for the writing process.

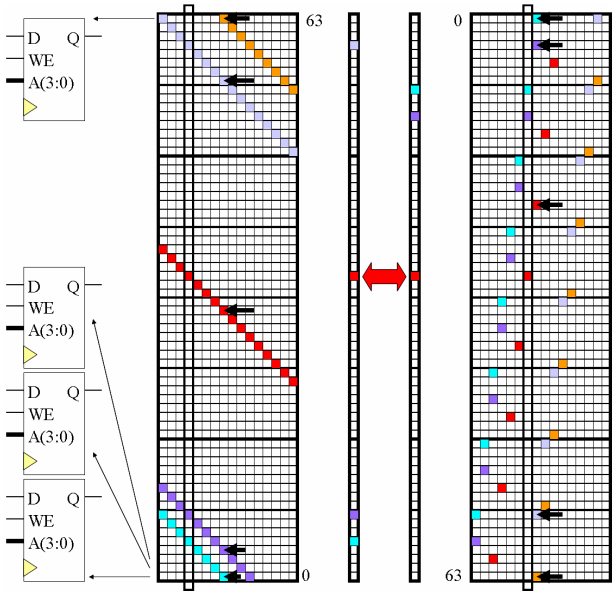


Fig. 5. Tiny Triplet Finder implemented with distributed RAMs: Each row of the bit map represents a 16x1-bit RAM with addresses from 0 to 15 mapped from left to right. The position of each plane hit is marked with an arrow. The vertical bar in each RAM map indicates how the bank of 64 RAMs is read out for a given read address.

While reading the register array for Plane_A, all the RAMs are given the same address (indicated with a vertical bar in Fig. 5), so the hit pattern appears at the outputs of the array. Changing the address shifts the hit pattern. In the register array for Plane_A, relative shift from +7 units to -8 units can be achieved. Again, the reading of the shifted pattern takes a single clock cycle (not 7 or 8).

To cover the entire range of the shift, the register array for Plane_C is also implemented similarly. The bit being filled in the RAM at the hit location is bit 7. In the RAMs 8m (where m = 1 to 7) bins above the hit location, the bits (7+m) are filled, and similarly in the RAMs 8n bins below the hit location, the bits (7-n) are filled. For each Plane_C hit, a jumping patterns as show in the right bit map in Fig. 5 is written.

In the reading phase, the RAMs of the register array for Plane_C are all given the same address. The hit pattern appears at the outputs of the array. If the address changes by 1, the hit pattern jumps upward or downward by 8.

Combining the two arrays allows all of the relative shifts between them to be achieved. In fact, the addresses during read for the two arrays are derived from the coordinate of the Plane_B hit. Plane_B is divided into 128 bins and a hit on Plane_B is represented by a 7-bit integer B(6:0). The reading addresses RA(3:0) for Plane_A and RC(3:0) for Plane_C are constructed as: RA(3:0)={B(6),B(2:0)} and RC(3:0)=B(6:3)-B(6). Some examples are tabulated in Table II.

It can be seen that when B<=63, possible values for RA are 0 to 7, or left half of the Plane_A bit map, and when B>63, possible values for RA are 8 to 15, or the right half. In other words, B(6) is used as a "page selection bit" for RA. The

reader may also note that given 4+4 address lines in two bit maps, in order to generate relative shifts from +63 to -64, only half of the Plane_A bit map would be needed. The reason why redundant bits are booked and B(6) is used as a "page selection bit" for RA(3:0) is for better plane edge coverage for the bit-wise coincident logics. As shown in Fig. 2, when the Plane_B hit is below the mid-point, or B<=63, the bin 0 edges of Plane_A and C must be contained in the coincident logics. When B>=63, the other edges, the bin 63 edges of the Plane_A and C must be contained. It can be shown that without redundant page in bit map for Plane_A, which is free in this case, about 8 bits additional RAMs and bit-wise coincident logics beyond 64 bits would be required to cover the track roads being shifted out of the plane edges. The otherwise unused bits in RAMs for Plane_A are used to contain the logics within 64 bits (see Fig. 5 and the last column of Table II).

TABLE II
GENERATION OF READING ADDRESSES OF THE RAM BANKS

	RA(3:0) = {B(6), B(2:0)}		RC(3:0) = B(6:3) – B(6)		Total Shift	Covered Plane Edges
B(6:0)	RA	A shift	RC	C shift		
0	0	+7	0	-56	+63	The bin 0 edges of plane A and C.
1	1	+6	0	-56	+62	
7	7	+0	0	-56	+56	
8	0	+7	1	-48	+55	
9	1	+6	1	-48	+54	
62	6	+1	7	0	+1	
63	7	0	7	0	0	both
64	8	-1	7	0	-1	The bin 63 edges of plane A and C.
65	9	-2	7	0	-2	
70	14	-7	7	0	-7	
71	15	-8	7	0	-8	
72	8	-1	8	+8	-9	
126	14	-7	14	+56	-63	
127	15	-8	14	+56	-64	

In the A shift and C shift columns, the + sign represents an upward shift of the corresponding hit pattern in Fig. 5. The total shift in the table = (A shift) - (C shift).

It can be seen that the functions of the BitReg and the Shifter blocks in Fig. 3 are integrated in the two RAM arrays. By eliminating the Shifter block, the silicon resource usage is further reduced as shown in Table I.

As mentioned earlier that the maximum possible range of relative shift between the two patterns should be 256 given 4+4 address lines of the two arrays, so it is possible to build a 128-bit TTF without using a shifter. (Of course, as mentioned above, about 16 bits additional RAMs and coincident logics will be needed to cover the plane boundaries.) For bigger TTF, a shifter will be needed, but the shifting functions provided by the RAM arrays will reduce the size of the shifter significantly. For example, for a 1024-bit TTF, the shift range is 2048 that requires an 11-stage shifter, one stage for each bit of the shift index. With RAM arrays, 8 bits in the shift index can be eliminated leaving only 3 to be implemented in the shifter.

A disadvantage for the implementation discussed in this section is longer reset time. Since the 16x1-bit RAM does not support fast reset, 16 clock cycles are needed to clear the RAM arrays to prepare for a new event, while the register-based implementation takes only one clock cycle.

VI. CONCLUSION

We have described an FPGA implementation of the Tiny Triplet Finder. Since the Tiny Triplet Finder algorithm uses no special logic operations other than shift and bit-wise AND/OR, it is also suitable for software implementation. In most CPU or DSP processors, the TTF algorithm is expected to allow execution time to be reduced from $O(n^3)$ to $O(n)$.

Although we used the simplest configuration, i.e., straight tracks in three equally spaced detector planes as an example in this document, the TTF algorithm can be extended to configurations with non-equally spaced planes and more than three planes. We have discussions on these cases in Reference [5]. In another document we are currently composing, several applications with curved tracks are studied.

In addition to track segment finding, the TTF algorithm may also be used in hit recognition problems in wire chambers, time of flight counters, and GEM/MICROMEGAS detectors. These applications will be discussed in separate documents.

REFERENCES

[1] Kulyavtsev et al., BTeV proposal, Fermilab, May 2000, available: { http://www-btev.fnal.gov/DocDB/0000/000066/002/index.html}

[2] E.E. Gottschalk, BTeV detached vertex trigger, Nucl. Instrum. Meth. A 473 (2001) 167.

[3] M. Wang et al., "A Commodity Solution Based High Data Rate Asynchronous Trigger System for Hadron Collider Experiments", Proceedings of the IEEE Real Time Conference 2005, Stockholm, Sweden, June 2005, to be published.

[4] M. Votava et al., "BTeV Trigger/DAQ Innovations", Proceedings of the IEEE Real Time Conference 2005, Stockholm, Sweden, June 2005, to be published.

[5] J. Wu, et al., "Tiny Triplet Finder (TTF) – A track segment recognition scheme and its FPGA implementation developed in the BTeV level 1 trigger system", Proceedings of the 10th workshop on electronics for LHC and future experiments, p68, (2004) FERMILAB-CONF-04-270-E available: {http://www.slac.stanford.edu/spires/find/hep/www?r=fermilab-conf-04-270-e}

[6] J. Wu, M. Wang, E. Gottschalk, G. Cancelo and V. Pavlicek [for BTeV collaboration], "Hash sorter: Firmware implementation and an application for the Fermilab BTeV level 1 trigger system," FERMILAB-CONF-03-357-E available: {http://www.slac.stanford.edu/spires/find/hep/www?r=fermilab-conf-03-357-e} Presented at IEEE 2003 Nuclear Science Symposium (NSS) and Medical Imaging Conference (MIC), Portland, Oregon, 19-24 Oct 2003}

[7] Altera Corporation, "Cyclone FPGA Family Data Sheet", (2003) available via: {http://www.altera.com/}

[8] Altera Corporation, "APEX II Programmable Logic Device Family Data Sheet", (2002) available via: {http://www.altera.com/}

[9] R. Fruhwirth et al., "Data Analysis Techniques for High-Energy Physics", 2nd ed., Cambridge, 2000

[10] Xilinx Inc., "Virtex-II Platform FPGAs: Complete Data Sheet", (2005) available: { http://direct.xilinx.com/bvdocs/publications/ds031.pdf}