

# POWER EFFICIENT MODIFIED FIFO BASED HASH JOIN ARCHITECTURE

Mr. M Vamsi Krishna Allu<sup>1</sup> J. JONEY BETTILLOW<sup>2</sup> K. MOUNIKA<sup>3</sup> K.V.S.N. SUDHEER<sup>4</sup> L. MEGHANA<sup>5</sup>  
K.SANTHI KUMARI<sup>6</sup>

Assistant Professor Sir C.R Reddy college of Engineering [1]  
UG Scholars, Sir C R Reddy College of Engineering [2,3,4,5,6]

**Abstract-** The main objective of this concept is to design a hash join operator with at-most efficiency. This project presents a non-collision parallel static random-access memory (SRAM)-based hash join architecture. This architecture utilizes multiple hash functions and content addressable memories (CAMs) to eliminate hash collision, thereby ensuring a worst constant memory access for each phase in the hash join algorithm and consequently improving the hash join throughput. These Hash joins are useful in the implementation of a relational database management system, sorting, aggregation. In the era of Internet of Things (IOT) and Big Data, fast query processing is a crucial requirement of the modern DBMS. The performance of the central processing unit (CPU) is not growing sufficiently quickly to handle the rapidly increasing amount of data, leading to demands for new processing methods to speed up database systems. A non-collision parallel hash join strategy is proposed. Proposed strategy addresses the hash collision and provides insert and query operations of the hash join algorithm with a worst-case constant time. A parallel hash join architecture comprising multiple channels and a CAM is constructed. This architecture distributes the tuples in different hash channels and therefore, there is no need for duplicate storages. clock gating architecture to limit the switching activity of the address decoder which improves the power efficiency of the proposed FIFO. Element structure is adapted to evaluate the clock cycle to the present ring counter block and to release the clock pulse to the next ring counter block. FIFO Memory accessing does not need write operations so data lines and read/write lines to the SRAM memory architecture is omitted.

**Index Terms**— Database operation, hash join, hardware acceleration, FPGA, parallel pipeline.

## INTRODUCTION

A hash join is a database join operation used to combine two sets of data based on a common attribute or join condition. In a hash join, both input datasets are first partitioned based on the join key, and then each partition is hashed into a hash table.

Once the hash tables are constructed, the join operation is performed by matching the hash values of the join keys between the two tables.

Hash joins are often faster than other join algorithms, such as nested loop joins or merge joins, especially when dealing with large datasets, as they typically have a time complexity of  $O(n)$ , where  $n$  is the number of rows in the input datasets.

They are commonly used in database systems to optimize join operations.

Hash join is a widely used technique in database query processing for joining two large datasets efficiently. It works by partitioning the rows of the two input datasets into buckets based on a hash function applied to the join key, then matching rows with the same hash value across the datasets. This approach minimizes the need for sorting and allows for parallel processing, making it particularly effective for large-scale data operations.

### 1.1 HASH JOIN ARCHITECTURE:

Hash join architecture is a method used in database management systems to join tables efficiently. It involves hashing the join columns from both tables and then comparing the hashed values to find matching rows. This method is particularly effective for large datasets as it minimizes the need for sorting and allows for faster retrieval of data. Hash join architecture typically involves two phases: **build phase** and **probe phase**. In the build phase, a hash table is created by hashing the smaller of the two tables. In the probe phase, the larger table is hashed and compared with the hash table created in the build phase to identify matching rows

1. **Input Data:** Two input datasets that need to be joined based on a common attribute (join key).
2. **Hash Function:** A hash function is applied to the join key of each row in both input datasets to generate a hash value. This hash value determines the partition to which the row belongs.
3. **Partitioning:** Rows from each input dataset are partitioned into buckets based on their hash values. This step ensures that rows with the same hash value (and potentially the same join key) are grouped together.
4. **Hash Tables:** Hash tables are built for each partition or bucket of the smaller dataset. These hash tables store the join key and corresponding values from the smaller dataset.
5. **Probe Phase:** For each row in the larger dataset, the hash function is applied to its join key to determine the corresponding bucket. The hash table of that bucket is then probed to find matching rows from the smaller dataset.
6. **Join Operation:** Matching rows found during the probe phase are combined to form the joined result. This process continues until all rows from the larger dataset have been processed.
7. **Output:** The joined result is produced as the output of the hash join operation

### 1.2 FIFO:

FIFO stands for "First In, First Out." It's a method for organizing and manipulating a data buffer, where the oldest (first) entry, or "head" of the queue, is processed first, and the newest (last) entry, or "tail" of the queue, is processed last. The FIFO structure.

In terms of memory, a FIFO structure is often implemented using a queue data structure. It's commonly used in computing and electronic circuits to manage data flow between different parts of a system. For example, in computer science, FIFO can be used in scheduling algorithms, caching systems, and network routing. In hardware, FIFOs are used to buffer data between components that operate at different speeds or have different data processing rates.

In essence, a FIFO memory operates like a line at a supermarket checkout: the first person to join the line is the first one to be served, and as more people join, they line up behind the others in the order they arrived. Similarly, in a FIFO memory, data items are stored and retrieved in the order they were added.

### 1.3 CONTENT ADDRESSABLE MEMORY (CAM):

Using the hash join architecture, it utilizes multiple hash functions and content addressable memories (CAMs) to eliminate hash collision, thereby ensuring a worst constant memory access for each phase in the hash join algorithm and consequently improving the hash join throughput. Useful in the implementation of a relational database management system sorting, aggregation. Further, this project is modified using clock gating technique to reduce power consumption of FIFO. Content Addressable Memory (CAM) is a specialized type of computer memory that allows data to be accessed based on its content rather than its storage address. It enables fast search and retrieval operations by comparing the input data with the stored contents of the memory cells, making it useful for applications like network routing tables and associative caching.

## LITERATURE SURVEY

1. [12] C. Kim, T. Kaldewey, V. W. Lee, E. Sedlar, A. D. Nguyen, N. Satish, J. Chhugani, A. Di Blas, and P. Dubey, "Sort vs. hash revisited: Fast join implementation on modern multi-core cpus," *Proc. VLDB Endow.*, 2009.

Join is an important database operation. As computer architectures evolve, the best join algorithm may change hand. This paper re-examines two popular join algorithms – hash join and sort-merge join – to determine if the latest computer architecture trends shift the tide that has favored hash join for many years. For a fair comparison, we implemented the most optimized parallel version of both algorithms on the latest Intel Core i7 platform. Both implementations scale well with the number of cores in the system and take advantages of latest processor features for performance. Our hash-based implementation achieves more than 100M tuples per second which is 17X faster than the best reported performance on CPUs and 8X faster than that reported for GPUs. Moreover, the performance of our hash join implementation is consistent over a wide range of input data sizes from 64K to 128M tuples and is not affected by data skew. We compare this implementation to our highly optimized sort-based implementation that achieves 47M to 80M tuples per second. We developed analytical models to study how both algorithms would scale with upcoming processor architecture trends. Our analysis projects that current architectural trends of wider SIMD, more cores, and smaller memory bandwidth per core imply better scalability potential for sort-merge join. Consequently, sort-merge join is likely to outperform hash join on upcoming chip multiprocessors. In summary, we offer multicore implementations of hash join and sort-merge join which consistently outperform all previously reported results. We further conclude that the tide that favours the hash join algorithm has not changed yet, but the change is just around the corner.

**2. [13] S. Schuh, X. Chen, and J. Dittrich, “An experimental comparison of thirteen relational equi-joins in main memory,” in SIGMOD, 2016.**

Relational equi-joins are at the heart of almost every query plan. They have been studied, improved, and re-examined on a regular basis since the existence of the database community. In the past four years several new join algorithms have been proposed and findings. This makes it surprisingly hard to answer a very simple question: what is the fastest join algorithm in 2015? In this paper we will try to develop an answer. We start with an end-to-end black box comparison of the most important methods. Afterwards, we inspect the internals of these algorithms in a white box comparison. We derive improved variants of state of-the-art join algorithms by applying optimizations like software write combine buffers, various hash table implementations, as well as NUMA-awareness in terms of data placement and scheduling. We also inspect various radix partitioning strategies. Eventually, we are in the position to perform a comprehensive comparison of thirteen different join algorithms. We factor in scaling effects in terms of size of the input datasets, the number of threads, different page sizes, and data distributions. Furthermore, we analyze the impact of various joins on an (unchanged) TPC-H query. Finally, we conclude with a list of major lessons learned from our study and a guideline for practitioners implementing massive main-memory joins. As is the case with almost all algorithms in databases, we will learn that there is no single best join algorithm. Each algorithm has its strength and weaknesses and shines in different areas of the parameter space.

**3. [14] D. A. Alcantara, V. Volkov, S. Sengupta, M. Mitzenmacher, J. Owens, and N. Amenta, “Building an efficient hash table on the gpu,” in GEMS, 2011.**

This chapter describes a straightforward algorithm for parallel hash table construction on the graphical processing unit (GPU). It constructs the table in global memory and use atomic operations to detect and resolve collisions. Construction and retrieval performance are limited almost entirely by the time required for these uncoalesced memory accesses, which are linear in the total number of accesses; so the design goal is to minimize the average number of accesses per insertion or lookup. In fact, it guarantees a constant worst-case bound on the number of accesses per lookup. Further, one alternative to using a hash table is to store the data in a sorted array and access it via binary search. Sorted arrays can be built very quickly using radix sort because the memory access pattern of radix sort is very localized, allowing the GPU to coalesce many memory accesses and reduce their cost significantly. However, binary search, which incurs as many as  $\lg (. N)$  probes in the worst case, is much less efficient than hash table lookup. GPU hash tables are useful for interactive graphics applications, where they are used to store sparse spatial data-usually 3D models that are voxelized on a uniform grid. Rather than store the entire voxel grid, which is mostly empty, a hash table is built to hold just the occupied voxels.

#### **4. [4] Classification Framework for the Parallel Hash Join with a Performance Analysis on the GPU by KA Wozniak, 2017.**

The hash join operator is one of the most important relational operators in database applications and a prominent research topic in the domain of parallel processing. However, up to date, no consistent algorithm design guidelines for high-performance implementations on parallel platforms have been derived from the available experimental results. In this work we define a taxonomy of the parallel hash join operator landscape and categorize state of the art research accordingly. Moreover, we implement and benchmark three taxonomy types: A sequential implementation on the CPU, a hybrid CPU-GPU implementation as well as a fully parallel version on the GPU. The results show that (1) the hybrid CPUGPU type outperforms the other two, showcasing the benefits of a good fit between algorithm type and hardware platform choice, (2) the poor end-to-end performance of the GPU-only type highlights the impact of GPU specific synchronization and contention issues that appear with an unfit design choice, (3) parallelization improves runtime by a factor of 2.2X in the end-to-end algorithm, a factor of 83X in the join phase and shows good scaling behaviour with increasing number of threads. This proves that the GPU is a valuable co-processor option for computation offloading in database applications. We anticipate this classification framework to be a starting-point for design decisions for parallel big data hash join operators on other heterogeneous systems.

#### **5. [15] H. Pirk, S. Manegold, and M. Kersten, “Accelerating foreign-key joins using asymmetric memory channels,” in ADMS, 2011.**

Indexed Foreign-Key Joins expose a very asymmetric access pattern: the Foreign-Key Index is sequentially scanned whilst the Primary-Key table is target of many quasi-random lookups which is the dominant cost factor. To reduce the costs of the random lookups the fact-table can be (re-) partitioned at runtime to increase access locality on the dimension table, and thus limit the random memory access to inside the CPU’s cache. However, this is very hard to optimize and the performance impact on recent architectures is limited because the partitioning costs consume most of the achievable join improvement [3]. GPGPUs on the other hand have an architecture that is well suited for this operation: a relatively slow connection to the large system memory and a very fast connection to the smaller internal device memory.

## **EXISTING METHOD**

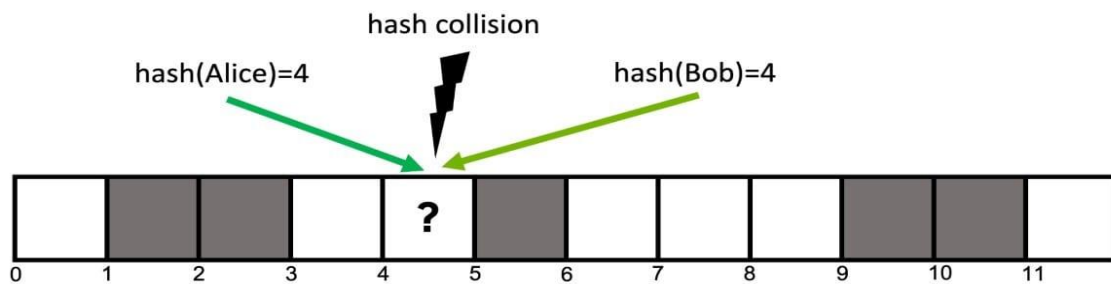
### **3.1 OVERVIEW OF HASH JOIN TECHNIQUES:**

Hash join (like merge join) can only be used if there is at least one equality clause in the join predicate. This is usually not an issue because joins are typically used to reassemble relationships, expressed with an equality predicate between a primary key and a foreign key. Let’s call the set of columns in the equality predicate the “hash key,” because these are the columns that contribute to the hash function. Additional predicates are possible, and are evaluated as “residual predicate” separately from the comparison of hash values. Note that the hash key can be an expression, as long as it can be computed exclusively from column in a single row.

### 3.2 HASH COLLISION:

In computer science, a hash collision is a random match in hash values that occurs when a hashing algorithm produces the same hash value for two distinct pieces of data. Hashing algorithms are often used to prevent third parties from intercepting digital messages.

In fact, hashing algorithms provide the extra layer of protection necessary to secure the transmission of a message to its recipient.



**Fig: 3.1**

Represents how the hash collision occurs

In computer science, hashing is a common practice used for a variety of purpose including cryptography, data indexing, and data compressing. Both hashing and cryptography protect data by transforming it into a secure format. However, while cryptography protect data by transforming it into a secure format. However, while cryptography uses a process called encryption, hashing uses a mathematical formula called as hash function to truncate one value into another. The hash collision occurrence is shown in the Fig 3.1.

### 3.3 PREVENTION OF HASH COLLISION USING LINEAR PROBING TECHNIQUE:

Collision is occurred, when same index keys are placed on a hash table map on the same location. So, to avoid these collisions we are using the linear probing technique to resolve this. Linear probing is a strategy for resolving collisions. In this the new key is placed in the closest following empty cell. Here the elements are stored wherever the hash function maps into a hash table, if that cell is filled then the next consecutive location is searched to store that value. Here generally we use arrays. The Table-3.1 represents that how the linear probing is done to avoid the hash collisions.

**Table-3.1:** The collisions are prevented by using linear probing technique

SL.NO	KEY	HASH	ARRAY INDEX	AFTER LINEAR PROBING, ARRAY INDEX
1	1	$1\%20 = 1$	1	1
2	2	$2\%20 = 2$	2	2
3	42	$42\%20 = 2$	2	3
4	4	$4\%20 = 4$	4	4
5	12	$12\%20 = 12$	12	12
6	14	$14\%20 = 14$	14	14
7	17	$17\%20 = 17$	17	17
8	13	$13\%20 = 13$	13	13
9	37	$37\%20 = 17$	17	18

The formula for the linear probing is  $= \text{key} \% \text{list\_size}$ . By using the division module, we will perform the linear probing. In the above formula the term “key” is the value of which we need to take it on the numerator and then the list\_size is that it represents size of the list and it is placed on the denominator. At first the division operation is performed, then after performing we take the remainder as the array index. If that array index of which we get from the remainder is same as the value in which it is present in the above list, then it is called hash collision. Then by using the linear probing technique we will increase the index value of +1 for each step until we get the new array index which is not present in the table. By using this process, we are going to avoid the collisions. It’s possible to cascade CAM up to eight levels without incurring a performance penalty but beyond this the cost/power return becomes unfeasible.

### 3.4 HASH JOIN OPERATORS:

Hash join operates in two phases:

- Build phase
- Probe phase

In the former, the first table is scanned, and the hash function is used to populate a hash table with the tuples. In the latter, the second table is scanned, and the hash table is probed to find matching results. In this paper, the first and second tables are assumed to have N and M rows, respectively ( $N < M$ ). The tuples are a part of the integrated records, stream into the join architecture with a {rid, key} format.

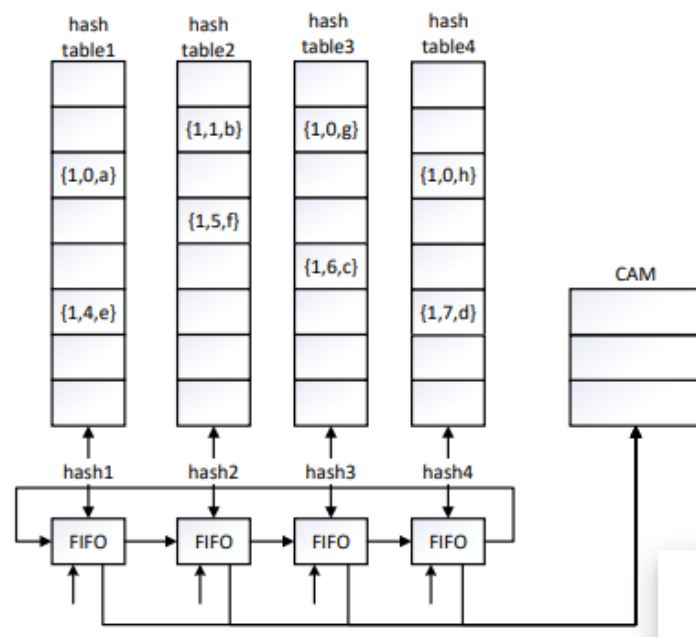


The rid is used to distinctly identify a row, and the key is the join attribute. The join results {rid1, rid2, key} are produced when the second table streams in.

### 3.4.1 BUILD PHASE:

During the build phase, the FPGA fetches the first table's tuples in parallel. The FIFO in each channel is used to cache the incoming tuples from the local storage or other channels. The key of the tuple is hashed in each channel and the location of the tuple is saved in the corresponding address on the hash table calculated by each channel's hash function.

Tuples are stored in the hash tables in the {status, rid, key} format, and the status (1 bit) indicates whether the row of the hash table is occupied. Conflicting hash values in each channel are shifted to the next channel; this shift process is terminated until this tuple is inserted to one hash table successfully or reaches the shift threshold (the channel number); then, it is moved to the CAM with a {rid, key} format.



**Fig.3.2:** Example of a four-hash channel with a CAM hash join architecture.

The key in the hash tables and CAM will be compared to the key of the second table during the probe phase. As shown in Fig.3.4, there are multiple hash channels, and each channel has dedicated resources allowing distribution in the first table in parallel during the build phase. In each hash channel, non-conflicting tuples require two clock cycles to update the hash table (i.e. to determine whether the corresponding address is available and to write data).

If a hash collision has occurred, conflicting tuples are pushed to the FIFO for further processing. Thus, the memory access time of each hash channel remains constant during the build phase.



It stores eight tuples (a–h), every row of the hash table has three members: {status, rid, key}. The status indicates whether the row is occupied.

### 3.4.2 PROBE PHASE:

During the probe phase, the tuples in the second table are streamed in and compared with those of the first table for matching results. When a match is found, the rows are joined together and formatted into a {rid1, rid2, key} style.

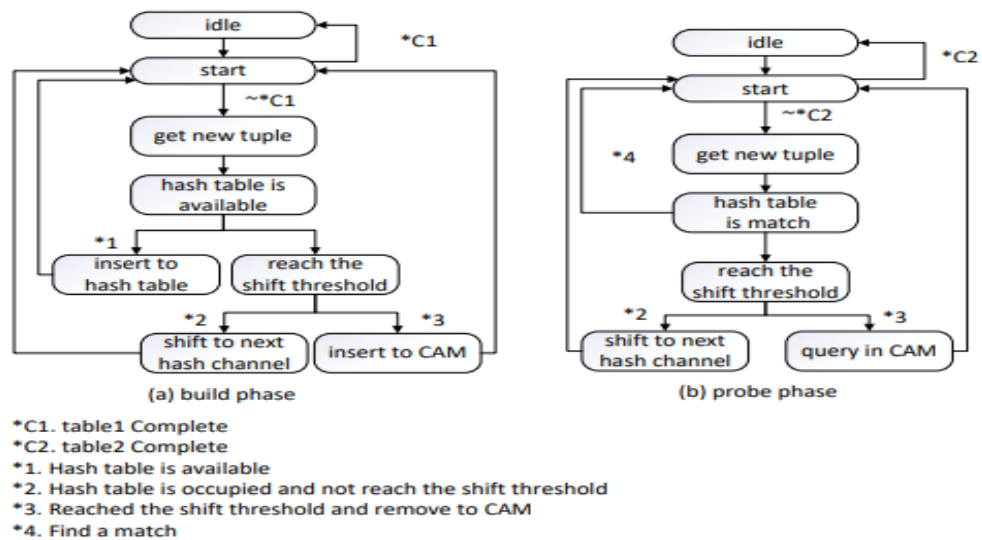
Tuples from the second table move through the hash channel in the pipeline, and multiple hash channels operate in parallel until they are either invalidated or joined. Similar to the build phase, keys from the second table worked through the multiple hash channels in parallel. In each hash channel, the tuple is routed to the corresponding address calculated by each channel's hash function to find a match.

This is performed on each channel concurrently in a pipeline. If a match is found in one channel, the join result is generated for further processing; otherwise, this tuple shifts to the next channel for the next step lookup. If a tuple cannot find a match in all channels, it would be sent to the CAM for query operation.

Mismatch in all channels and CAM results in the tuple being discarded. In this paper, we focus on the “N-to-1” join relationship, such that once the key of the second table is matched, the search process is terminated. Certain components during the probe phase of each channel require a constant number of cycles to complete and therefore have no stall of the pipeline. Exact table matching operations on CAMs only use one clock cycle. Thus, the memory access time during the probe phase is also constant.

### 3.5 DATA SHIFT STRATEGY:

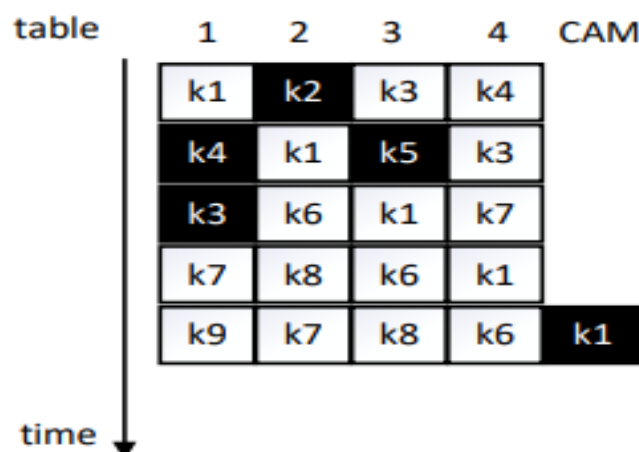
The main idea behind our design is to provide multiple hash channels to distribute the first table with a small CAM, thus ensuring that the tuples that cannot be inserted into all the hash tables can be moved to the CAM, generating a non-collision hash join scheme. In Fig 3.5, an example of the timeline of memory access operations for the data shift non-collision scheme is illustrated.



**Fig.3.3:** Finite-state machines for the build phase and probe phase.

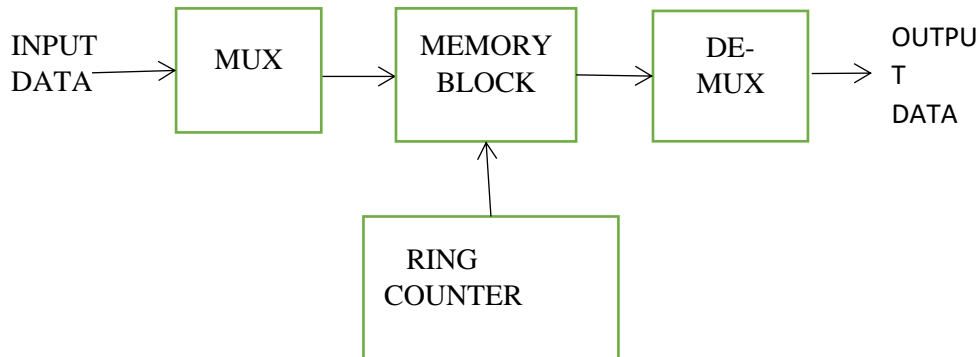
It can be seen that during the first time slot, items (K1, K2, K3, K4) stream and the item K2 is inserted (or probed) simultaneously; in the second time slot, the remaining items (K1, K3, K4) are shifted, while a new item (K5) substitutes the inserted (or probed) item, and items (K4, K5) are inserted (or probed) successfully; therefore, in the third time slot can process two new items (K6, K7); in the fourth time slot, no item is successfully inserted (or probed); in the fifth time slot, item (K1) is moved to the CAM because the shift threshold (the hash channel number) has been reached.

And the worst-case access time occurs. This strategy operates similarly both in the build phase and probe phase, and is developed by finite-state machines (FSMs), as shown in Fig.3.3.



**Fig.3.4:** Data shift strategy for build phase and probe phase.

### 3.6 FIFO ARCHITECTURE:

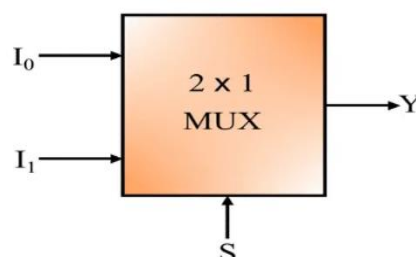


**Fig 3.5:** Existing block of Memory organisation

In the existing method, the input data is taken as binary numbers and that data is sent to the mux. The mux is also called as the selection input. It selects and gives one output data. Then after that output data is given to the memory block of which it is used to store the data of which the multiplexer is sent. After that ring counter is a type of counter composed of a circular shift register. A ring counter connects the output of the last shift register to the first shift register input and circulates a single one (or zero) bit around the ring. For example, in a 4-register one-hot counter, with initial register values of 1000, the repeating pattern is: 1000, 0100, 0010, 0001, 1000. Note that one of the registers must be pre-loaded with a 1 (or 0) in order to operate properly. This output data of the ring counter is given to the memory block, This memory block is used to store the data which comes from the ring counter. After that this total data is given to the demultiplexer of which it will take the single input and gives more number of outputs. Based on type of mux we will take the demux. If mux is 4:1 then we take the demux as 1:4. This total operation is shown in the below Fig 3.5.

#### 3.6.1. MULTIPLEXER:

Input buffer is a multiplexer. The operation of the mux is used to select one of many input signals and forward it to a single output line. There is another input line that is the selection line of which it is used to give the data more efficiently for our given input data. These selection lines are depends on our number of inputs we had given to the multiplexer. Ex: We had taken multiplexer of 2:1, of which it has only 2 input lines and 1 output line of which it has only one selection line. Mainly these multiplexers are used in the data communication of which multiple data streams into single channel. The Fig 3.6 represents the multiplexer and the Table 3.7 shows the truth table of the multiplexer.



Multiplexers can be implemented using logic gates such as AND gates, OR gates, and NOT gates, or using electronic switches such as transmission gates or CMOS switches.

**Fig 3.6: 2:1 MUX**

**Table 3.2: Truth Table for 2:1 mux**

S	I0	I1	Y
1	A	B	A
0	A	B	B

### 3.6.2. MEMORY BLOCK:

(RAM) Random-access memory (RAM) is a form of computer data storage. Today, it takes the form of integrated circuits that allow stored data to be accessed in any order (that is, at random). "Random" refers to the idea that any piece of data can be returned in a constant time, regardless of its physical location and whether it is related to the previous piece of data.

The word "RAM" is often associated with volatile types of memory (such as DRAM memory modules), where the information is lost after the power is switched off. Many other types of memory are RAM as well, including most types of ROM and a type of flash memory called NOR-Flash.

Scan design has been the backbone of design for testability (DFT) in industry for about three decades because scan-based design can successfully obtain controllability and observability for flip-flops. Serial Scan design has dominated the test architecture because it is convenient to build.

In RAS, flip-flops work as addressable memory elements in the test mode which is a similar fashion as random access memory (RAM). This approach reduces the time of setting and observing the flip-flop states but requires a large overhead both in gates and test pins. Despite of these drawbacks, the RAS was paid attention by many researchers in these years.

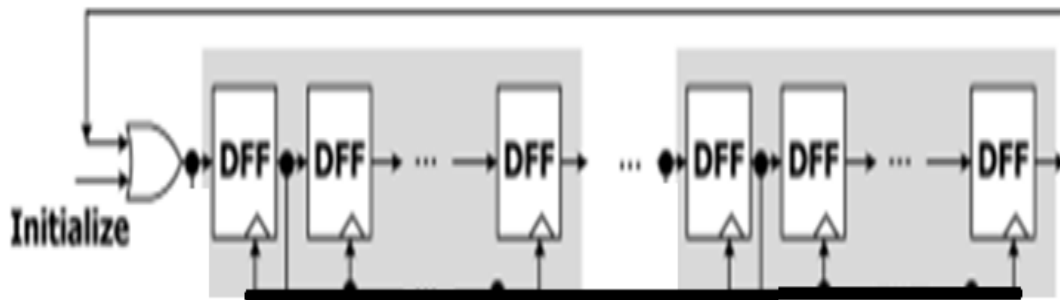
### 3.6.3 RING COUNTER:

A ring counter is a type of counter composed of a circular shift register. The output of the last shift register is fed to the input of the first register. There are two types of ring counters: A ring counter connects the output of the last shift register to the first shift register input and circulates a single one (or zero) bit around the ring.

For example, in a 4-register one-hot counter, with initial register values of 1000, the repeating pattern is: 1000, 0100, 0010, 0001, 1000. Note that one of the registers must be pre-loaded with a 1 (or 0) in order to operate properly.

A twisted ring counter also called Johnson counter connects the complement of the output of the last shift register to its input and circulates a stream of ones followed by zeros around the ring. For example, in a 4-register

counter, with initial register values of 0000, the repeating pattern is: 0000, 1000, 1100, 1110, 1111, 0111, 0011, 0001. If the output of a shift register is fed back to the input.



**Fig 3.7:** Operation of Ring counter

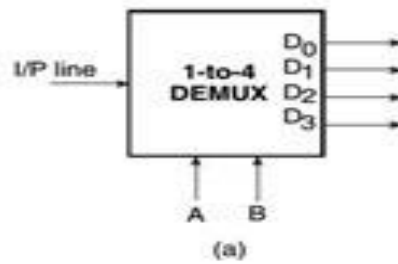
The data pattern contained within the shift register will recirculate as long as clock pulses are applied. For example, the data pattern will repeat every four clock pulses in the fig 3.7. However, we must load a data pattern. All 0's or all 1's doesn't count. 4-bit ring counter, which means it consists of 4 D flip-flops connected in a ring configuration.

Initially, if all flip-flops are reset, all outputs are 0. When the clock pulse arrives, the value at the input of FF1 gets transferred to its output (Q1). The value at Q1 gets transferred to Q2 on the next clock pulse. Similarly, the value at Q2 gets transferred to Q3, and Q3 to Q4. Finally, the value at Q4 gets transferred back to Q1, completing the loop. This process continues with each clock pulse, causing the logic level to circulate around the ring. Each flip-flop in the ring holds a different bit of the counter value. So, as the clock pulses, the counter effectively counts in binary from 0001 to 1111 (or in decimal from 1 to 15), with each clock pulse representing one count increment. Ring counters are often used in applications where a cyclic sequence is needed, such as generating control signals for sequential circuits or creating timing signals in digital systems. Fig 3.7 represents the operation of ring counter.

### 3.6.4 DEMUX:

It is normally called as the “demultiplexer”. The operation of the demux is opposite to the multiplexer, by taking a single input and gives many numbers of output lines based on control signals or selection signals. The selection of the output line is controlled by a set of binary control inputs.

The number of control lines depends on the number of output lines. There are some of common sizes include 1:2, 1:4, 1:8, 1:16, etc. where the first number represents the number of output lines and the second number represents the number of control lines. It is also to distribute a single incoming data stream to multiple output channels. It is used for data routing and distribution.



I/P	Select		O/P			
	A	B	D <sub>0</sub>	D <sub>1</sub>	D <sub>2</sub>	D <sub>3</sub>
1	0	0	1	0	0	0
1	0	1	0	1	0	0
1	1	0	0	0	1	0
1	1	1	0	0	0	1

(b)

**Fig 3.9:** Representation of demux diagram and its truth table

Demultiplexers operate based on the principle of Boolean logic. The control inputs determine which output line receives the input signal. For example, in a 1:4 demultiplexer, there are two control inputs (typically labelled as A and B), and the combination of these inputs selects one of the four output lines to receive the input signal.

Each demultiplexer has a truth table that defines the output for every combination of input signals and control inputs. Demultiplexers can be implemented using logic gates such as AND gates, OR gates, and NOT gates, or using electronic switches such as transmission gates or CMOS switches. Demultiplexers introduce some delay in the signal path due to the time taken to select and route the input signal to the selected output line. The Fig 3.10 represents demux diagram and its truth table.

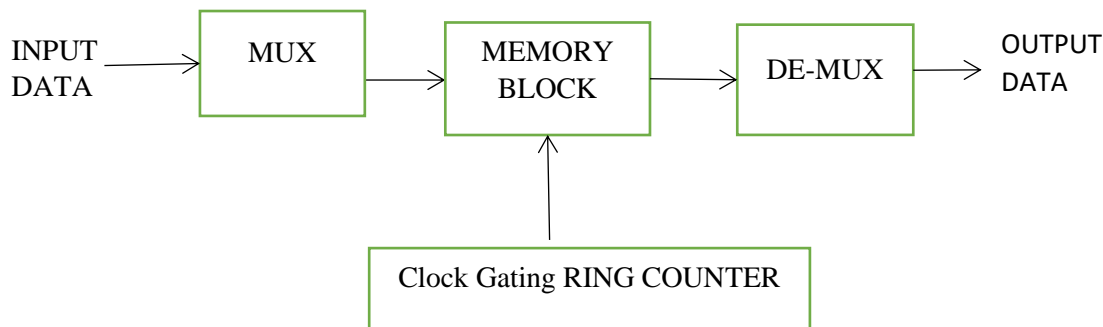
## 4. PROPOSED ARCHITECTURE

### 4.1 MEMORY ORGANISATION OF PROPOSED ARCHITECTURE:

The memory organisation of the FIFO architecture in existing method can be implemented by using normal RING COUNTER. During this process the power consumption is more due to the clock supply. By controlling this issue, we can insert the modified RING COUNTER is also called as “CLOCK GATING RING COUNTER”.

## 4.2 MODIFIED FIFO ARCHITECTURE:

Modified FIFO architecture can be organized with the replaced ring counter that is connected to the memory block. In this architecture the total block cell is divided into two blocks, by insertion of two SR flipflops in each memory block and the clock supply also divided into two clock pulses which is shown in Fig 4.1. However, this can generate non-collision hash join architecture with efficient power supply.



**Fig.4.1:** Proposed block of Memory organisation

### 4.2.1. CLOCK GATING RING COUNTER:

In Fig.4.1, the modifications are held in the ring counter block and the working of the clock gating and the ring counter that can be explained in the below process.

A clock gating ring counter is a sequential logic circuit used in digital design. It combines the concepts of clock gating and ring counter.

**Clock Gating:** In digital circuits, clock gating is a technique used to reduce power consumption by disabling the clock signal to certain parts of a circuit when they are not in use. This is achieved by using a gating signal to enable or disable the clock signal.

**Ring Counter:** A ring counter is a type of counter in which the output of each flip-flop is connected to the input of the next flip-flop in a circular arrangement. The counter advances by one count with each clock cycle, with only one flip-flop being in the active state at any given time.

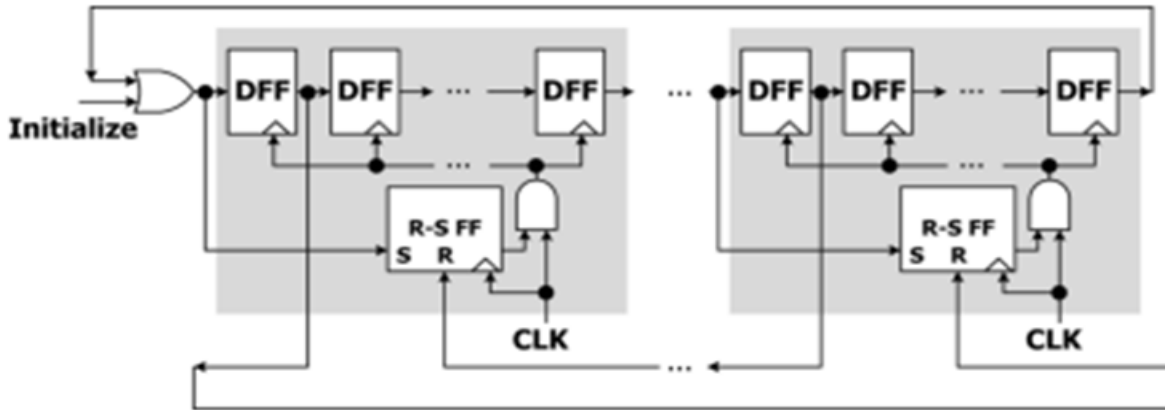
Combining these two concepts, a clock gating ring counter uses clock gating to enable or disable the clock signal to individual stages of a ring counter. This can be useful for applications where power consumption needs to be minimized or where clock signals need to be dynamically controlled based on certain conditions.

For example, in a system where certain operations are only performed periodically, the clock gating ring counter can disable the clock signal to those parts of the system during idle periods, thus saving power. When those operations are needed, the clock gating can be enabled to allow the clock signal to propagate through the ring counter and perform the necessary tasks.



#### 4.3. NON-COLLISON PARALLEL HASH JOIN STRATEGY WITH CLOCK GATING FIFO:

Non-collision parallel hash join strategy with Clock gating FIFO is implemented in given Fig.4.2 that divided the ring counter of D flipflops block into two equal parts and the AND gate is inserted in between the SR flipflop and the D flip flops in the equal manner.



**Fig.4.2: Modified Ring Counter with SR Flip-Flops**

The output of the last D flip flop is given the input to OR gate and the gate output is given input to the first SR flipflop and the output 1 of d-ff and the input of the half block of first d-ff is given the inputs to the second SR flipflop.

The clock pulse is supply to the both SR flipflops and AND gate, when the half of the ring counter is in active state and other half is in disable state because of the ring counter is allow only single 1, if the logic 1 is applied to the first half, then the clock supply is given to the particular block only remaining block is off state.

The above block diagram Fig.4.2 shows the Power controlled Ring counter. First, total block is divided into two blocks. Each block is having one SR FLIPFLOP controller.

**Table 4.1:** SR Flip Flop Truth Table

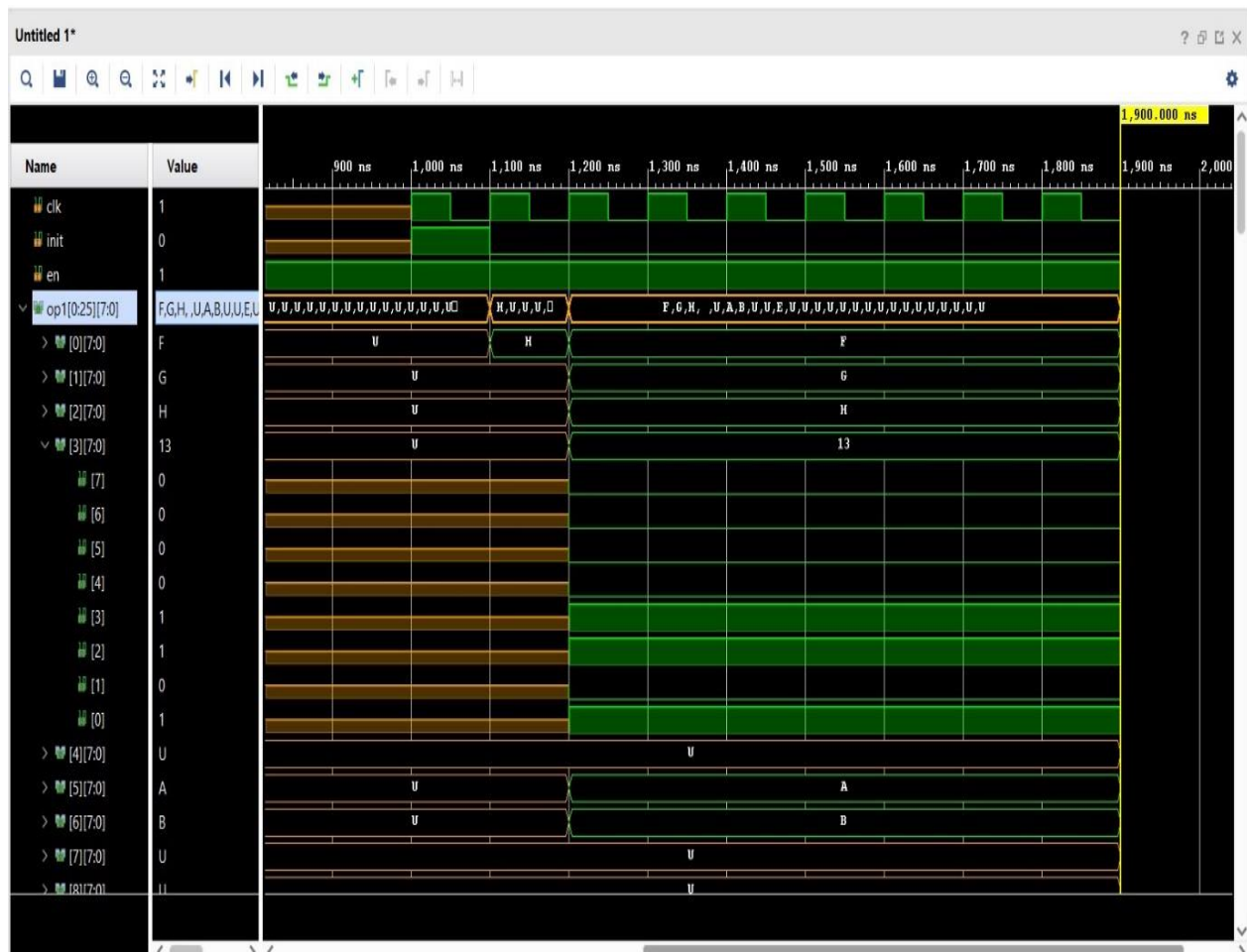
S	R	Q(t-1)	Q
0	0	0	0
0	0	1	1
0	1	1	0
0	1	1	0
1	0	1	1
1	0	1	1
1	1	0	X
1	1	1	X

It is a Flip Flop with two inputs, one is S and other is R. S here stands for Set and R here stands for Reset. Set basically indicates set the flip flop which means output 1 and reset indicates resetting the flip flop which means output 0. Here clock pulse is supplied to operate this flip flop, hence it is clocked flip flop shown in Table 4.3. During this process the entire power is reduced compare to the existing method and the non-collision hash join architecture is implemented.

## RESULTS AND DISCUSSIONS

### 6.1. EXISTING METHOD:

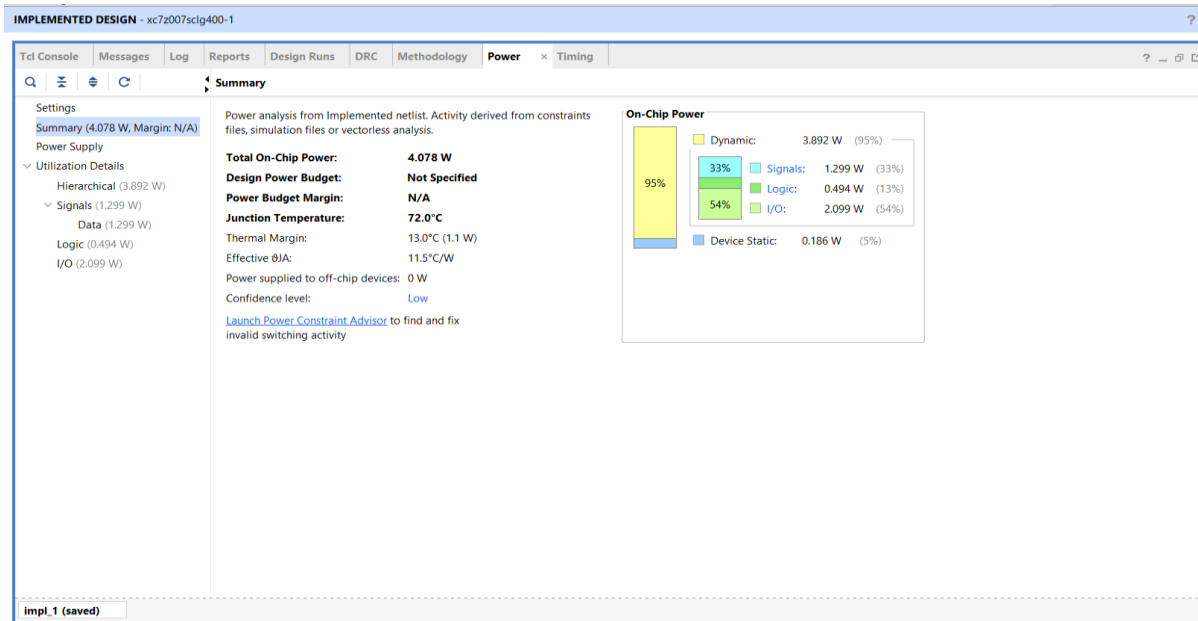
The simulation results of existing method is implemented by using Xilinx Vivado software tool. In each method power synthesis is generated and the total on-chip power and the dynamic and static powers are designed in the implementation. The power delayed products are shown in the table 5.1 and four memory locations.



**Fig.5.1:** Simulation results of existing architecture.

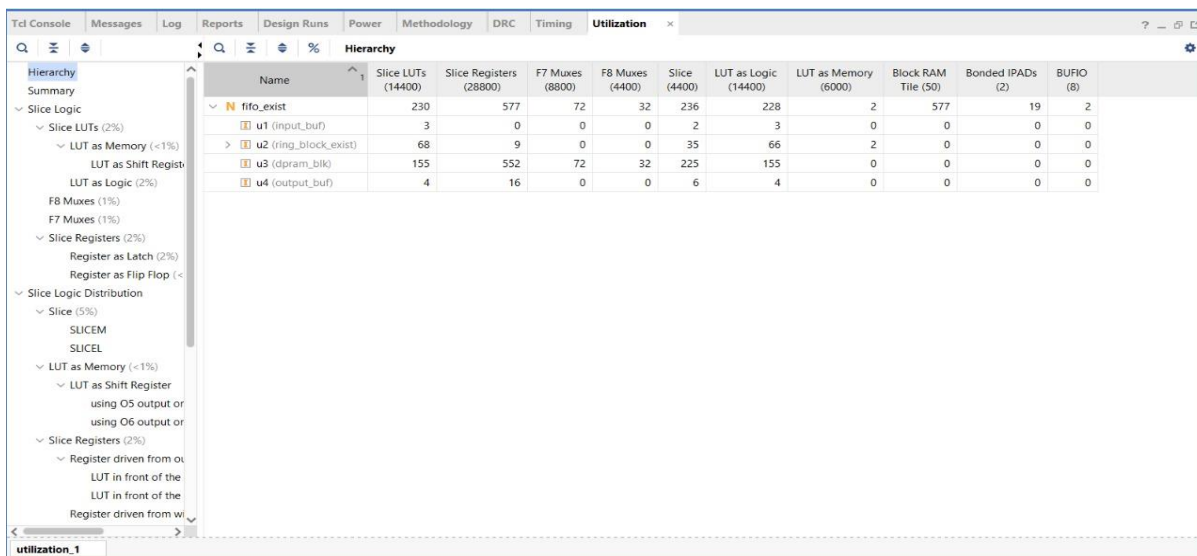
The hash join architecture can be implemented by using the FIFO architecture, in this memory organisation RING COUNTER is used. It consists a set of flip-flops connected in a circular manner. In this simulation any two hash

tables (hash1, hash2, hash3, hash4) can be joined and the combined memory locations are generated in the output as shown in (fig.5.1).



**Fig 5.2:** Power generated in exiting method

The total On-Chip power that can be optimised in this design is **4.078 W** as shown in Fig 5.2. The dynamic power is 3.892 W and the static power is 0.186 W are simulated in run synthesis which are given in Table 5.1.



Name	Slice LUTs (14400)	Slice Registers (28800)	F7 Muxes (8800)	F8 Muxes (4400)	Slice (4400)	LUT as Logic (14400)	LUT as Memory (6000)	Block RAM Tile (50)	Bonded IPADs (2)	BUFIO (8)
<b>N f1fo_exist</b>	230	577	72	32	236	228	2	577	19	2
<b>u1 (input_buf)</b>	3	0	0	0	2	3	0	0	0	0
<b>u2 (ring_block_exist)</b>	68	9	0	0	35	66	2	0	0	0
<b>u3 (dpram_blk)</b>	155	552	72	32	225	155	0	0	0	0
<b>u4 (output_buf)</b>	4	16	0	0	6	4	0	0	0	0

## 6.2 PROPOSED METHOD:

The simulation results of the proposed method is implemented by using Xilinx Vivado software tool. In each method power synthesis is generated and the total on-chip power and the dynamic and static powers are designed in the implementation. The power delayed products are shown in the Table 6.1 and four memory locations.

Unconstrained Paths - NONE - NONE - Setup											
Name	Slack	Levels	Routes	High Fanout	From	To	Total Delay	Logic Delay	Net Delay	Requirement	Source Clock
Path 1	∞	3	4	17	enable	data_output[2]	5.377	3.778	1.599	∞	input port clk
Path 2	∞	3	4	17	enable	data_output[3]	5.377	3.778	1.599	∞	input port clk
Path 3	∞	3	4	17	enable	data_output[5]	5.377	3.778	1.599	∞	input port clk
Path 4	∞	3	4	17	enable	data_output[7]	5.377	3.778	1.599	∞	input port clk
Path 5	∞	3	4	17	enable	data_output[0]	5.351	3.752	1.599	∞	input port clk
Path 6	∞	3	4	17	enable	data_output[1]	5.351	3.752	1.599	∞	input port clk
Path 7	∞	3	4	17	enable	data_output[4]	5.351	3.752	1.599	∞	input port clk
Path 8	∞	3	4	17	enable	data_output[6]	5.351	3.752	1.599	∞	input port clk
Path 9	∞	6	6	2	u3/dpram_reg[54][0]/G	u4/dmux_out_11_reg[0]/D	3.804	1.629	2.175	∞	
Path 10	∞	6	6	2	u3/dpram_reg[54][1]/G	u4/dmux_out_11_reg[1]/D	3.804	1.629	2.175	∞	

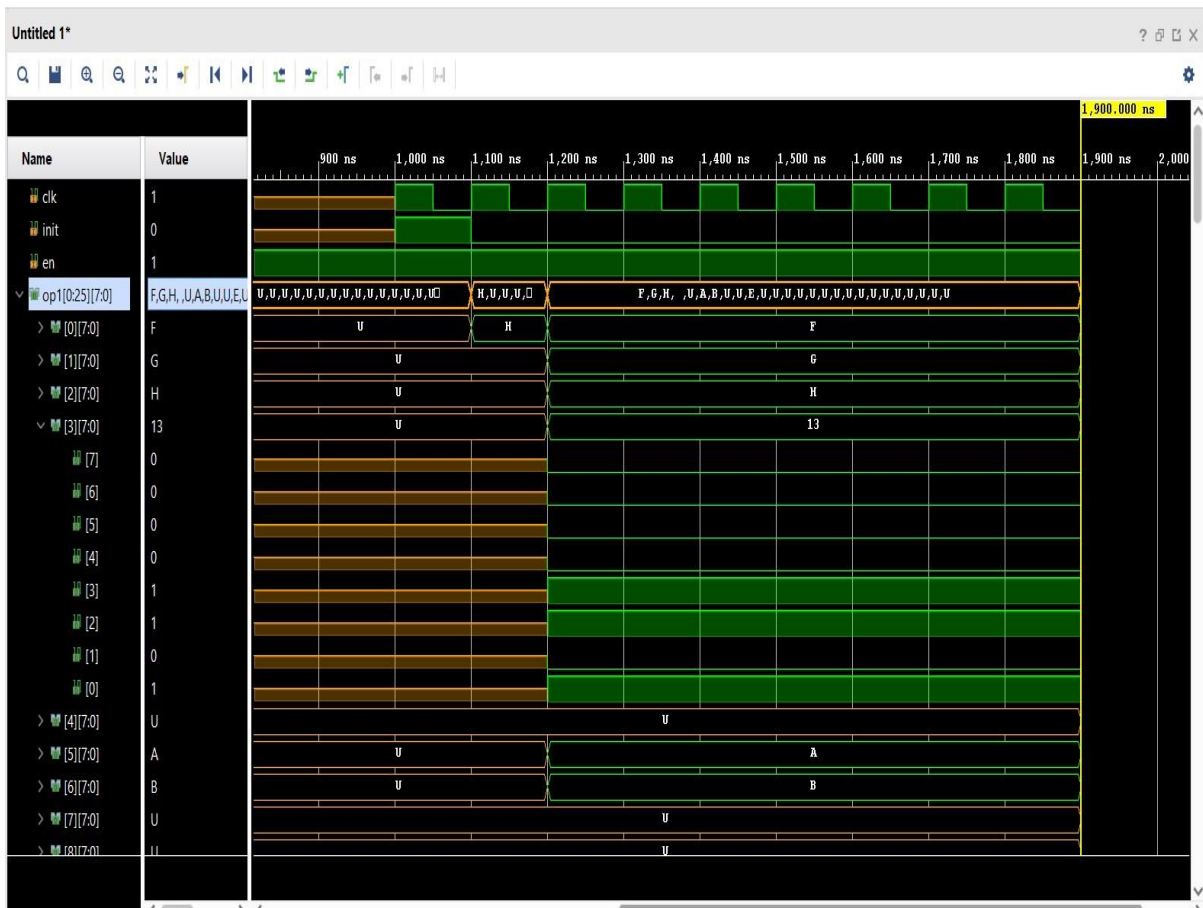
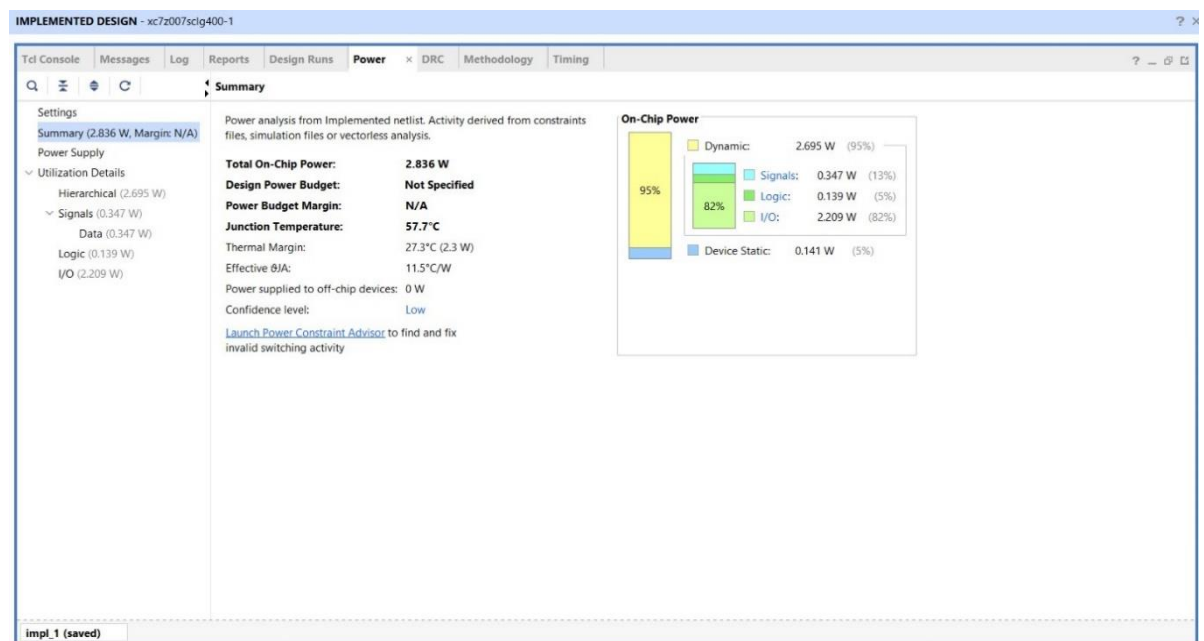


Fig.5.5: Simulation results of proposed architecture.

The above simulation results are same as in the existing method but in this design ring counter that can be modified.

The modified “RING COUNTER using Clock Gating” technique is used and the power consumption can be reduced by this clock gating technique.

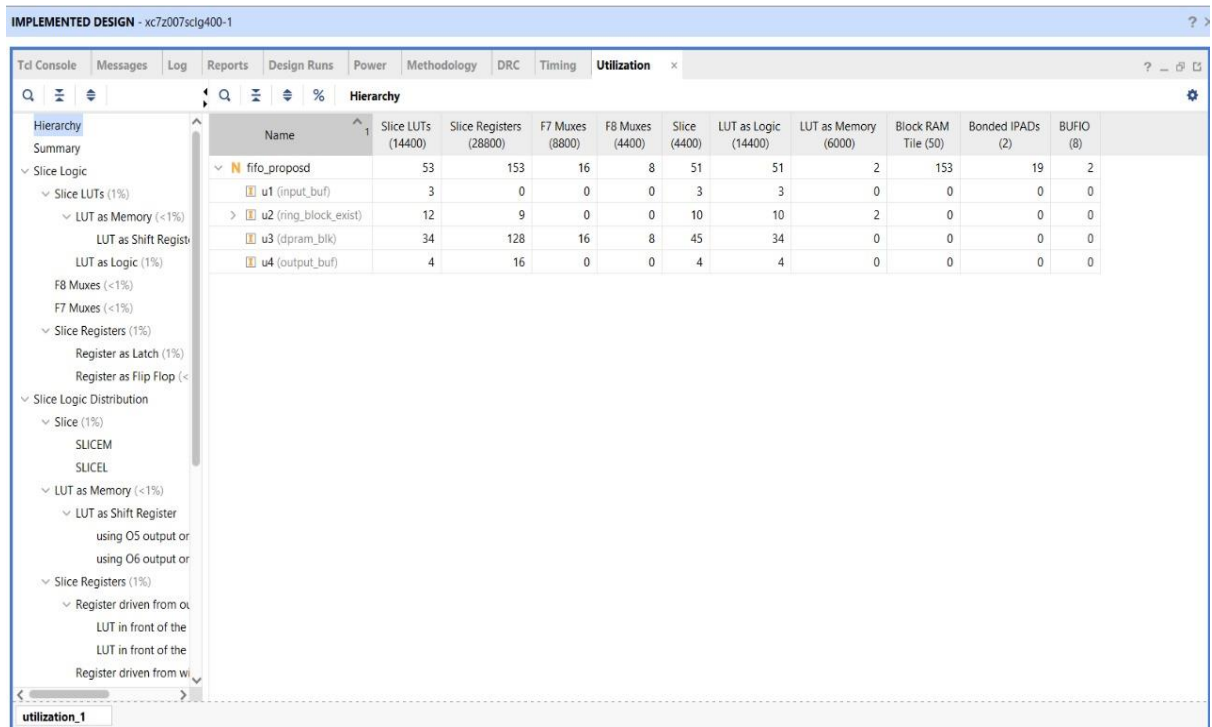
The total on chip power that can be reduced as compare to existing method is 2.836W as shown in Fig 5.5. The dynamic power is 0.141 W and the static power is 2.713 W are simulated in run synthesis which are given in Table 5.1.



**Fig 5.6:** Total on chip power in proposed method

The area and the time delay in proposed method are generated as 508 gates are used

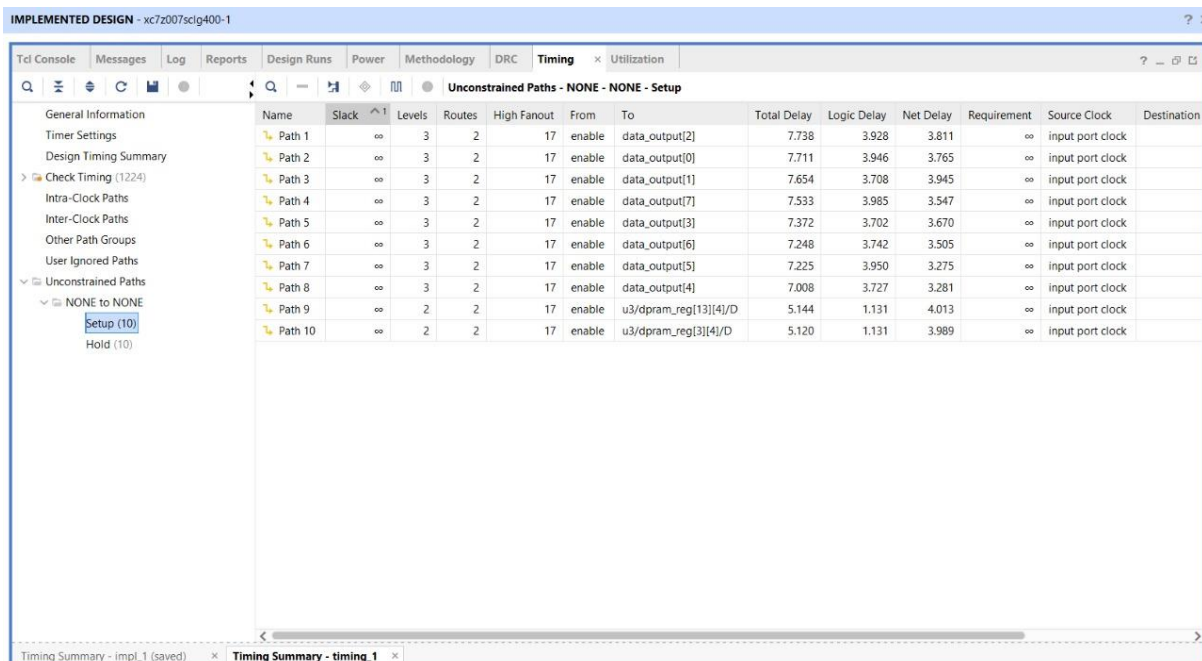
And the time is taken as 7.738ns as shown in Fig.5.7 and Fig 5.8.



utilization\_1

utilization_1											
Hierarchy											
Name	Slice LUTs (14400)	Slice Registers (28800)	F7 Muxes (8800)	F8 Muxes (4400)	Slice (4400)	LUT as Logic (14400)	LUT as Memory (6000)	Block RAM Tile (50)	Bonded IPADs (2)	BUFIO (8)	
fifo_proposed	53	153	16	8	51	51	2	153	19	2	
u1 (input_buf)	3	0	0	0	3	3	0	0	0	0	
u2 (ring_block_exist)	12	9	0	0	10	10	2	0	0	0	
u3 (dpram_blk)	34	128	16	8	45	34	0	0	0	0	
u4 (output_buf)	4	16	0	0	4	4	0	0	0	0	

Fig 5.7: Area in Proposed Method



Timing Summary - impl\_1 (saved) x Timing Summary - timing\_1 x

Unconstrained Paths - NONE - NONE - Setup												
Name	Slack	Levels	Routes	High Fanout	From	To	Total Delay	Logic Delay	Net Delay	Requirement	Source Clock	Destination
Path 1	∞	3	2	17	enable	data_output[2]	7.738	3.928	3.811	∞	input port clock	
Path 2	∞	3	2	17	enable	data_output[0]	7.711	3.946	3.765	∞	input port clock	
Path 3	∞	3	2	17	enable	data_output[1]	7.654	3.708	3.945	∞	input port clock	
Path 4	∞	3	2	17	enable	data_output[7]	7.533	3.985	3.547	∞	input port clock	
Path 5	∞	3	2	17	enable	data_output[3]	7.372	3.702	3.670	∞	input port clock	
Path 6	∞	3	2	17	enable	data_output[6]	7.248	3.742	3.505	∞	input port clock	
Path 7	∞	3	2	17	enable	data_output[5]	7.225	3.950	3.275	∞	input port clock	
Path 8	∞	3	2	17	enable	data_output[4]	7.008	3.727	3.281	∞	input port clock	
Path 9	∞	2	2	17	enable	u3/dpram_reg[13][4]/D	5.144	1.131	4.013	∞	input port clock	
Path 10	∞	2	2	17	enable	u3/dpram_reg[13][4]/D	5.120	1.131	3.989	∞	input port clock	

Fig 5.8: Time Delay in Proposed Method

### 6.3 SYNTHESIS RESULTS:

It is observed from the synthesis results, it seems the proposed solution significantly reduces the total on-chip power consumption from 4.078 W to 2.836 W. The area is same in existed and proposed methods. The total no. of gates required are 508 gates in both methods, total time delay in existing is 5.377ns and the total time delay in proposed is 7.738ns.

**Table 5.1:** Comparison of Area, Power, Time delay.

	AREA	DELAY	POWER
<b>EXISTING</b>	Total no. of gates required: 508 gates	Total delay: 5.377ns	Total on chip power: 4.078W
<b>PROPOSED</b>	Total no. of gates required: 508 gates	Total delay: 7.738ns	Total on chip power: 2.836W

### CONCLUSION:

In this project, multiple hash channels are provided to distribute the same table, tuples are processed in each channel in a pipeline, and each hash channel operates in parallel. A small CAM is used to resolve the hash collision by storing the tuples that cannot be inserted to all the channels. A data shift strategy is used in the build and probe phases to reduce the stalling of FPGA. Our design achieves a  $O(1)$  memory access time of the hash table for each phase, a constant time of the CAM insert operation in the build phase, and one clock cycle of the CAM search operation in the probe phase to improve the hash join throughput and ensure a deterministic worst case query time. Extended modified memory accessing scheme yields parameter optimisation when compare with existing method. The power has reduced by 1.242 watts in proposed architecture. So, non-Collision-based hash join architecture with power optimized architecture is concluded.



**FUTURE SCOPE:**

The other enhancement of this project is that the analysis of the critical parameters of the system characteristics to the performance gives insights to the architectural improvements for future hardware. Memory bandwidth for FPGA will grow by taking advantage of the improved frequency of FPGA and number of memory channels as well as the utilization of High Bandwidth Memory (HBM). The size of the on-chip RAMs can be even bigger according to its development over the last decade. For example, Xilinx Ultra Scale+ devices can have total 62.5MB on-chip RAMs. The Open CL SDK for FPGA can be much stronger providing better timing results and resource utilization.

**REFERENCES**

- [1] B. He, K. Yang, R. Fang, M. Lu, N. Govindaraju, Q. Luo, and P. Sander, "Relational joins on graphics processors," in SIGMOD, 2008.
- [2] J. He, M. Lu, and B. He, "Revisiting co-processing for hash joins on the coupled cpu-gpu architecture," Proc. VLDB Endow., 2013.
- [3] R. Rui, H. Li, and Y. C. Tu, "Join algorithms on gpus: A revisit after seven years," in ICBD, 2015.
- [4] R. Rui and Y.-C. Tu, "Fast equi-join algorithms on gpus: Design and implementation," in SSDBM, 2017.
- [5] M. Yabuta, A. Nguyen, S. Kato, M. Edahiro, and H. Kawashima, "Relational joins on gpus: A closer look," TPDS, 2017.
- [6] T. Kaldewey, G. Lohman, R. Mueller, and P. Volk, "Gpu join processing revisited," in DaMoN, 2012.
- [7] H. Wu, G. Damos, S. Cadambi, and S. Yalamanchili, "Kernel weaver: Automatically fusing database primitives for efficient gpu computation," in MICROArch, 2012. [8] J. He, S. Zhang, and B. He, "In-cache query co-processing on coupled cpu-gpu architectures," Proc. VLDB Endow., 2014. [9] P. Sioulas, P. Chrysogelos, M. Karpathiotakis, R. Appuswamy, and A. Ailamaki, "Hardware-conscious Hash-Joins on GPUs," ICDE, 2019.
- [10] S. Blanas, Y. Li, and J. M. Patel, "Design and evaluation of main memory hash join algorithms for multi-core cpus," in SIGMOD, 2011.
- [11] J. Teubner, G. Alonso, C. Balkesen, and M. T. Ozsu, "Main-memory hash joins on multi-core cpus: Tuning to the underlying hardware," in ICDE, 2013.

- [12] C. Kim, T. Kaldewey, V. W. Lee, E. Sedlar, A. D. Nguyen, N. Satish, J. Chhugani, A. Di Blas, and P. Dubey, "Sort vs. hash revisited: Fast join implementation on modern multi-core cpus," Proc. VLDB Endow., 2009.
- [13] S. Schuh, X. Chen, and J. Dittrich, "An experimental comparison of thirteen relational equi-joins in main memory," in SIGMOD, 2016.
- [14] D. A. Alcantara, V. Volkov, S. Sengupta, M. Mitzenmacher, J. Owens, and N. Amenta, "Building an efficient hash table on the gpu," in GEMS, 2011.
- [15] H. Pirk, S. Manegold, and M. Kersten, "Accelerating foreign-key joins using asymmetric memory channels," in ADMS, 2011.
- [16] C. Gregg and K. Hazelwood, "Where is the data? why you cannot debate cpu vs. gpu performance without the answer," in ISPASS, 2011.
- [17] Y. Yuan, R. Lee, and X. Zhang, "The yin and yang of processing data warehousing queries on gpu devices," Proc. VLDB Endow., 2013.
- [18] M. Heimel, M. Saecker, H. Pirk, S. Manegold, and V. Markl, "Hardwareoblivious parallelism for in-memory column-stores," Proc. VLDB Endow., 2013.
- [19] J. Krueger, et. al. "Fast Updates on Read Optimized Databases Using MultiCore CPUs," Proceedings of the VLDB Endowment, Vol. 5, No. 1, August 2012.
- [20] B. Low, B. Ooi, and C. Wong, "Exploration on Scalability of Database Bulk Insertion with Multi-threading" Int. J. New Computer Architectures and Their Applications, 2011.
- [21] R. Johnson, V. Raman, R. Sidle, and G Swart, "Rowwise Parallel Predicate Evaluation" Proc. Int. Conf on VLDB'08.
- [22] R. Mueller, J. Teubner, and G. Alonso, "Glacier: a query-tohardware compiler", In ACM SIGMOD.
- [23] J. Teubner and R. Mueller, "How soccer players would do stream joins," SIGMOD '11.
- [24] M. Sadoghi et al, "Multi-Query Stream Processing on FPGAs", IEEE Int. Conference on Data Engineering (ICDE), 2012.
- [25] B. Sukhwani, et. al. "Database Analytics Acceleration using FPGAs," Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques (PACT), 2012.