IJHSR
ijhighschoolresearch.org
RESEARCH ARTICLE

A High-Concurrency Design Based on FPGA for Aerospace Testing Systems

Yuqian Han

Jinling High School Hexi Campus, Nanjing, 210019, China; hanyuqian_2024@qq.com

ABSTRACT: Aerospace testing systems are crucial for the successful launch of spacecraft, responsible for conducting comprehensive health checks. As system devices grow increasingly complex, modern aerospace testing systems face high-concurrency challenges with over 2,000 channels. To address this, this paper proposes an FPGA (Field-Programmable Gate Array) architecture to replace the CPU (Central Processing Unit) for implementing the core task module—the real-time decision-making subsystem. First, a parallel logic architecture based on an FPGA is proposed. Leveraging experimental data, an adaptive mapping function between FPGA resources and the number of logic instances is fitted to address the matching issue between the quantity of parallel logic units and FPGA resources. Second, a hash mapping algorithm based on MurmurHash3 is designed to achieve the mapping between data channels and parallel logic units, thereby mitigating hash polarization and ensuring the balanced distribution of input signals across logic channels. Finally, a register rule writing algorithm based on modulus division is developed to guarantee the balanced allocation of decision rules in distributed registers. Additionally, a fast retrieval algorithm based on binary search is designed to enable efficient and rapid access to distributed registers. Test results show that the FPGA-based real-time decision-making system can automatically adapt to the FPGA resource structure, achieving a reduction in processing latency by more than 20 times and an increase in throughput by over 4 times compared to CPU systems, significantly enhancing the real-time processing capability of aerospace testing systems.

KEYWORDS: Aerospace Testing System, FPGA, Algorithms, Parallel Processing, Distributed Cache.

Introduction

The aerospace testing system[1,2] is the ground-based core system in aerospace engineering, responsible for the integrated testing and launch control of carrier rockets and spacecraft. It spans the entire lifecycle of spacecraft, from design and manufacturing to launch, ensuring the safety and reliability of space missions through high-precision measurement and telemetry technologies. With advancements in modern aerospace technology, testing systems face significant high-concurrency challenges during mission execution. Modern aerospace missions require comprehensive real-time health sensing and monitoring of various aspects, including structural integrity, vibration, shock, thrust, combustion, leakage, power supply, equipment performance, and atmospheric conditions. As spacecraft systems become increasingly complex, real-time parallel processing of over 2,000 objects is required. The data generation rate from test objects can reach millions of sampling points per second, and the identification and anomaly detection of these signals must be completed within milliseconds. This poses significant performance challenges to the testing system.

The traditional testing system architecture, centered around the CPU, struggles to meet the requirements of high concurrency. The CPU is constructed based on the Single Instruction Stream Single Data Stream (SISD) paradigm,[3,4] with limited Instruction-Level Parallelism (ILP), which severely hampers its ability to handle multiple tasks in parallel.[5,6] Additionally, the "memory wall" phenomenon is particularly prominent in CPUs,[7,8] as their shared bus architecture results in memory access delays accounting for over 60% of the total processing time, significantly reducing the overall system efficiency. CPUs also exhibit notable limitations in real-time scheduling. Under a polling scheduling mechanism, the response time for 2000 channels can extend to several seconds, failing to meet the needs of testing subsystems for high-speed real-time data processing. To address these issues, we employ an FPGA to replace the CPU for real-time control and decision-making functions. Leveraging its exceptional parallel processing capabilities, an FPGA can execute multiple independent operations simultaneously, effectively meeting high concurrency demands. For instance, in spacecraft telemetry, tracking, and command (TT&C), an FPGA can simultaneously process orbital data for hundreds of targets, achieving a Query Per Second (QPS) in the millions, whereas traditional CPU architectures, constrained by core count and cache size, typically achieve a QPS only in the tens of thousands to hundreds of thousands.

This paper presents the design and implementation of a real-time decision-making function for a testing system based on an FPGA, which meets the requirements for real-time detection of a large volume of concurrent data. The main contributions are as follows:

1. Designed and implemented an FPGA-based real-time decision logic system.

2. Design and completion of a mapping algorithm for hardware resources and the quantity of parallel logic units.

3. Designed a mapping algorithm for signal channels and logic instances, employing the MurmurHash3 algorithm to ef-

DOI: 10.36838/IJHSR813.43

fectively resolve hash polarization issues and enhance parallel performance.

4. Devised a register parallel access algorithm and a fast lookup algorithm, enabling high-speed reading of registers.

5. Based on actual sample data, designed and conducted system experiments, demonstrating a 25-fold reduction in processing delay and a 4.5-fold increase in throughput compared to CPUs, significantly improving processing performance.

Methods

Motivation:

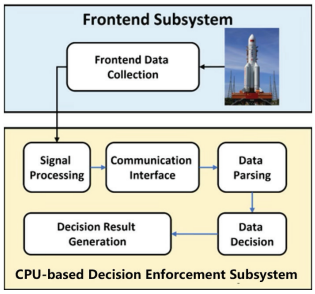

Figure 1: CPU-based Processing Architecture.

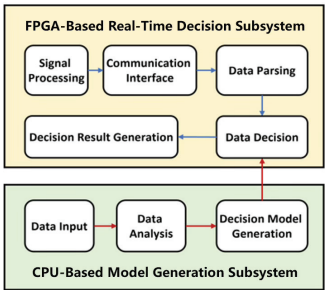

Figure 2: FPGA-based Processing Architecture.

Figure 1 illustrates the architecture of a traditional CPU-based spacecraft testing system, depicting the process by which the CPU processes signals and accomplishes real-time decision-making. This system primarily consists of a CPU core processing unit, a data acquisition subsystem, signal processing and interface circuits, a communication management module, a real-time decision-making module, a display and recording subsystem, as well as auxiliary circuits such as power supply and clock circuits. Among these components, the CPU core processing unit serves as the "brain" of the entire testing system, responsible for executing real-time decision-making functions. These functions involve real-time processing of received commands, such as fault diagnosis and state estimation. Based on the processing results, the system generates decision-making information in real time. However, due to the limitations of the CPU architecture, the CPU core processing unit struggles to meet high-concurrency, low-latency processing requirements, thereby becoming a bottleneck in the system.

Figure 2 depicts a system architecture where an FPGA is employed to replace the CPU for executing real-time decision-making functions, illustrating the process by which the FPGA accomplishes real-time decision-making. The CPU generates decision rules offline and transmits them to the FPGA, which then performs real-time decision-making. Leveraging parallel logic, the FPGA can simultaneously conduct computations on dozens to hundreds of logic units, meeting high-concurrency and low-latency requirements through parallel processing. The generation of the decision model is handled by the CPU, which then distributes the model to the FPGA's cache, serving as the basis for real-time decision-making. This paper primarily focuses on the design and implementation of an FPGA-based real-time decision-making system. For relevant design content regarding model generation, please refer to the paper on lightweight model design.

FPGA Decision-Making Logic System Design:

The fundamental idea behind the design of the decision-making logic is centered around distributed logical decision-making units, with connections to message input, memory access, and result output. To achieve high system performance, a multi-channel input method is employed for message input, while a distributed Register is utilized for memory access. System performance is enhanced through maximized parallelization.[9]

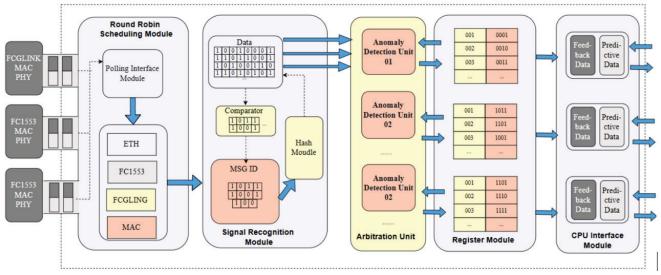

Figure 3: Architecture of the Decision-Making Logic System.

Figure 3 illustrates the architecture of an FPGA-based decision-making logic system, including five modules: the message scheduling module, signal recognition module, decision-making logic module, decision rule register module, and CPU interface module. The message scheduling module consists of three FIFO (First-In-First-Out) queues, serving as channels for the input of data signals. Data is extracted from these three FIFO queues using a polling method to ensure fair processing of all queues. After passing through the FIFO queues, the data is submitted to the signal identification module for information parsing and extraction. Subsequently, based on the monitoring object to which the signal belongs, a hash algorithm is employed to map the message to a specific decision-making logic unit within the decision-making logic module. It is important to note that due to varying data volumes generated by different monitoring objects, conventional hash mapping can lead to polarization effects. Therefore, data analysis and algorithm adjustments are required to mitigate the hash polarization effect.

The decision-making process of the decision-making logic unit is as follows:

1. The CPU pre-generates decision rules through offline training and writes the trained decision rules into a temporary register.

2. The decision-making logic retrieves the decision rule corresponding to the current data signal from the temporary register via a read enable signal.

3. The decision-making logic compares the current signal data with the decision rule to obtain a decision result

The read efficiency of register rules is a critical factor affecting system performance. We employ a parallel design and memory access optimization algorithm to enhance register access efficiency. Additionally, since the data in temporary registers is pre-written, there are no write operations during real-time decision-making. Therefore, the read process of the decision logic on the registers does not require locking and will not contaminate the register data.

In summary, based on the above FPGA system design, we need to address three key issues:

1. Channel mapping algorithm to determine the correlation between the number of logic units and the available system resources.

2. Hash polarization algorithm to ensure uniform mapping of data traffic from input channels to the logic unit array.

3. Register fast hit and parallel access algorithms to achieve high-performance Register reads.

Mathematical Relationship for Number of Logic Units and FPGA Resources:

Different FPGA models vary in their available hardware resources. The number of decision-making logic units in an FPGA is constrained by its available hardware resources. FPGA hardware comprises various resources such as Look-Up Tables (LUTs), Flip-Flops (FFs), and Block RAMs (BRAMs), and it is essential to identify the key resource constraints that impact the decision-making logic process. This paper employs an experimental approach to determine the influence factors of different resources. Specifically, simulations and syntheses were conducted at a frequency of 250 MHz using Verilog on a Xilinx Virtex-7 XT series FPGA chip with the Vivado 2023.1 software.

Table 1: Proportion of Hardware Resource Consumption by Judgment Units.

Resource	LUT	FF	BRAM
Utilization	10.247	26121	690
Utilization%	3.38%	4.03%	66.99%

Table 1 shows the consumption of LUT, FF, and BRAM resources by the FPGA decision-making logic. It can be observed that the logic algorithm requires a very small proportion of LUT and FF resources, accounting for no more than 10% of the chip's total resources. The primary resource demand comes from BRAM, which accounts for nearly 70%.

We conducted further tests on the resource utilization rate of BRAM.

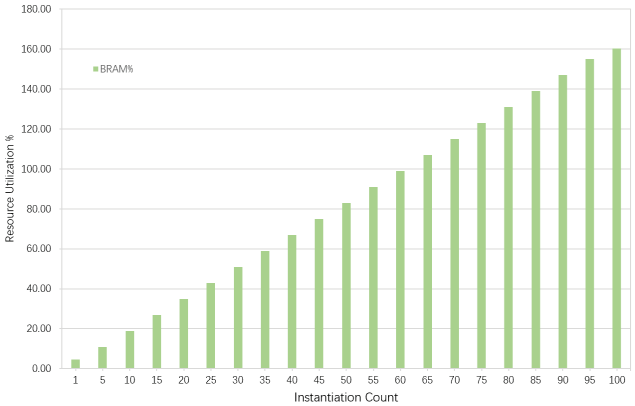

Figure 4: The Relationship Between Unit Instances and Performance Under the Same BRAM.

Figure 4 illustrates the number of logical instances generated by BRAM under different resource utilization rates. The vertical axis represents the BRAM resource utilization rate, while the horizontal axis denotes the number of decision-making units. Through function fitting, it is found that there is a linear relationship between the number of decision model instances and the BRAM resource utilization rate.

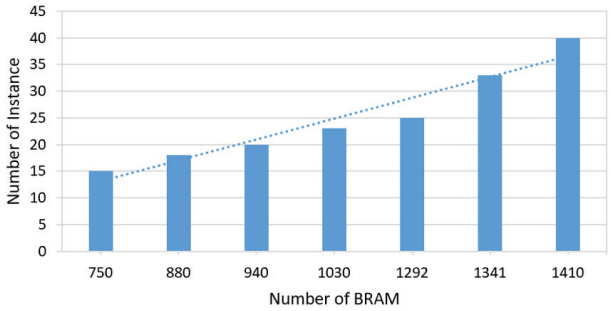

Figure 5: The Relationship Between Total BRAM Resources and the Number of Logic Units.

Figure 5 illustrates the direct relationship between BRAM resources and generated decision-making logic units. Typically, we need to reserve some space on the FPGA for additional functionalities

As the design scale increases, the number of logic units grows, and the corresponding requirements for cache, data width, and buffer resources also increase. From experimental observations, they exhibit an approximately linear relationship. To establish this linear relationship, we derive a linear model and perform fitting using the least squares method:

Let y be the number of BRAMs and t be the number of logic units. The objective is to derive the model:

$$y = kt + b$$

The goal of the least squares method is to minimize the sum of squared errors:

$$J = \sum_{i=1}^{n} (y_i - kt_i - b)^2$$

By taking partial derivatives of the system of equations and setting them to zero:

$$\partial S/\partial K = -2 \sum_i t_i (y_i - kt_i - b) = 0$$

$$\partial S/\partial b = -2 \sum_i (y_i - kt_i - b) = 0$$

By solving the equations, the linear function is derived as:

$$UnitNum = \frac{1}{50} BramNum + 4$$

Therefore, for a specific Field-Programmable Gate Array (FPGA) model, the number of decision logic instances is approximately one-fiftieth of the total BRAM resources.

Hash Mapping Algorithm for Parallel Unit and Input Channel:

The system needs to simultaneously monitor signals from more than 2,000 different objects, while the number of parallel decision-making units in the FPGA is far less than the number of signal objects. Therefore, a hash algorithm[10,11,12] is employed to achieve a uniform mapping from the test objects to the decision-making unit array.

$$UnitNo = OnjNo \ \% \ UnitNum$$

By using the aforementioned formula, test objects can be evenly distributed across the decision-making unit array. For instance, with 2,000 objects and 40 decision-making units, each unit will be allocated 50 objects.

DOI: 10.36838/IJHSR813.43

The hash algorithm achieves an even distribution of test objects, but due to varying signal periods generated by each object and differing amounts of data carried by each signal, there are significant disparities in the signal data volumes produced by different objects.

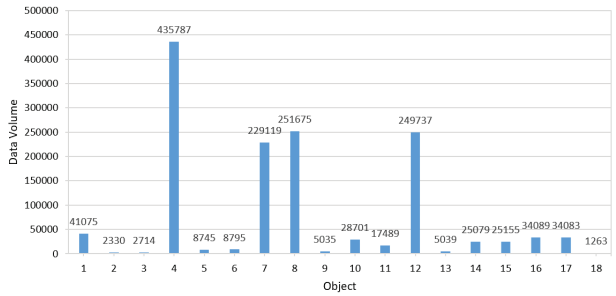

Figure 6: Data Volume of Objects.

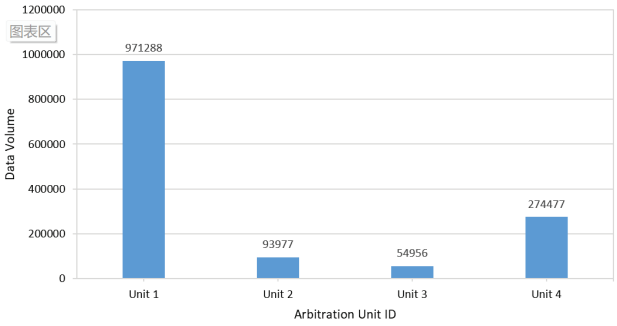

Figure 7: Data Volume of Decision-Making Units.

Figure 6 reveals significant disparities in the data volumes of different detection objects, resulting in a severe imbalance in the data traffic handled by the decision-making units. Figure 7 illustrates the data volumes allocated to four decision-making units using the hash algorithm based on the data in Figure 6, showing a typical polarization phenomenon. Therefore, it is necessary to optimize the hash algorithm to achieve a balanced distribution of data volumes.

Currently, mainstream polarization-optimized hashing algorithms include CityHash,[13] FNV,[14] SipHash,[15] MurmurHash3,[16] CRC32[17] (32-bit Cyclic Redundancy Check), etc. Among them, CityHash is optimized for big data hashing and offers good performance, but its hash distribution uniformity is not ideal. FNV is suitable for simple hash tables but has a relatively high collision rate. When processing input data with strong regularity, the hash distribution may not be uniform enough, leading to hash polarization issues. SipHash is suitable for HashMaps. Although it performs well in terms of hash distribution uniformity, its design focus is more on security rather than performance. MurmurHash3 excels in hash distribution uniformity and can effectively address hash polarization problems. Its design takes into account both performance and distribution uniformity, making it suitable for scenarios requiring efficient and uniform hash computations. CRC32 is primarily used for error detection; it is extremely fast but prone to collisions. Given that object mapping is a static deployment process rather than a runtime operation, and the primary consideration is the requirement for distribution uniformity, the MurmurHash3 algorithm is thus selected.

MurmurHash3 effectively disrupts the regularity of input data through a confusion step, uniformly distributing monitoring object identifiers with potential correlations into the hash space. Based on MurmurHash3, our algorithm is as follows:

1. Set the hash seed as seed_i, and define constants C1, C2, C3, and C4.

2. Load the data block k1 and initialize the hash value h1 with the seed. Then, perform the following operations on k1 and h1:

$$k1 = k1 * C1 \quad (1)$$

$$k1 = rotl32(k1, 15) \quad (2)$$

$$k1 = k1 * C2 \quad (3)$$

$$h1 = rotl32(h1 \hat{} k1, 13) * 5 + 32'he6546b64 \quad (4)$$

$$h1 = h1 \hat{} 32'd4 \quad (5)$$

$$h1 = (h1 \hat{} (h1 \gg 16)) * C3 \quad (6)$$

$$h1 = (h1 \hat{} (h1 \gg 13)) * C4 \quad (7)$$

$$h1 = h1 \hat{} (h1 \gg 16) \quad (8)$$

The key steps of the algorithm include:

Multiplication Confusion: Multiply the current data block by a pre-selected constant to produce nonlinear mixing.

Rotation Diffusion: Use cyclic shift operations to diffuse high-order bits into low-order positions.

XOR Merge: Perform XOR operations on intermediate results to further enhance the avalanche effect.

Final Mixing: Introduce data length information and perform final multiplication, shift, and XOR operations to ensure all input bits influence the final hash value.

The algorithm is highly hardware-friendly, as its multiplication, shift, and XOR operations are efficiently supported by FPGAs. The constant selection is flexible, allowing adjustments based on actual resource and performance requirements. In the FPGA implementation, the process is structured as a multi-stage pipeline, with each stage processing one data block and sequentially performing the confusion, rotation, and merge operations. The key function rotl32(x, r) (32-bit cyclic left shift by r bits) directly utilizes FPGA routing resources, avoiding bit loss in software shifts. Through pipelining, the module can complete the hash computation for one object identifier per clock cycle. Figure 8 shows that this algorithm effectively disperses high-traffic objects, which were originally concentrated in a few units. The optimization significantly improves the load distribution balance among the units, eliminating performance bottlenecks caused by a few overloaded units. This provides a crucial balanced load foundation for real-time processing of 2,000 concurrent signal streams.

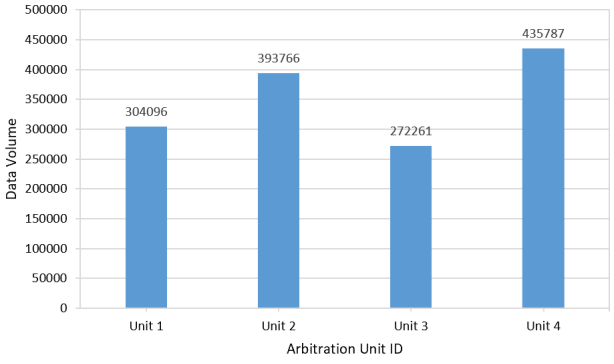

Figure 8: Balanced results using the MurmurHash3 algorithm.

Register Access Algorithm Design:

To enhance system efficiency and prevent register access from becoming a bottleneck, the design of register access needs to minimize the number of lookups and maximize access speed as much as possible. To this end, this paper optimizes register access efficiency from two aspects: optimizing lookup algorithms and improving parallel access.

High-Speed Cache Matching Algorithm:

Table 2: Segment of the Test Sample Set.

Timestamp	data3	data6	data7
42402781	28275	3171	3766
42402781	28275	3171	3766
42402786	28385	3171	3766
42402786	28385	3171	3766
42402791	28496	3171	3766
42402791	28496	3171	3766
42402796	28606	3171	3767
42402796	28606	3171	3767
42402801	28717	3172	3767
42402801	28717	3172	3767
42402805	28828	3172	3767

Table 2 displays a portion of the data from this sample set. In this set, each object varies in terms of data volume, decision logic, and signal type. The following presents a performance test conducted based on these samples.

Performance Evaluation:

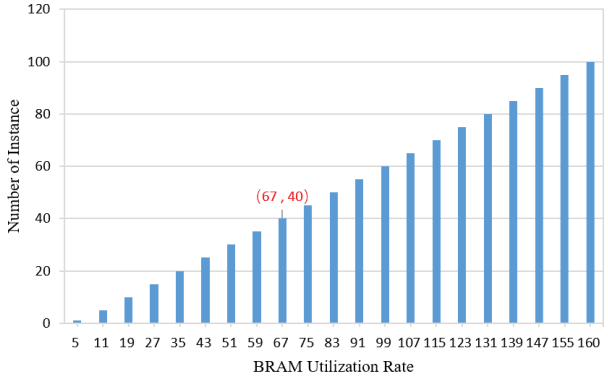

Figure 12: Linear relationship.

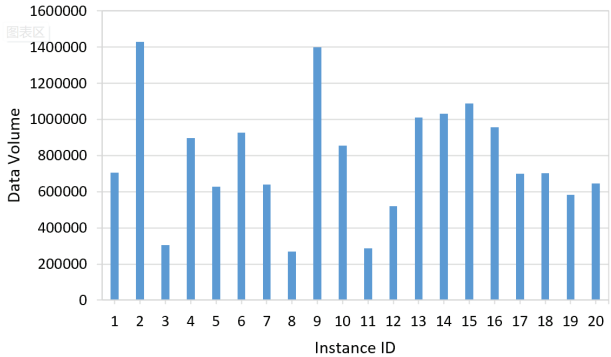

Figure 13: Traffic Distribution of Unit Array.

Figure 12 shows the experimental results regarding the BRAM resource utilization rate versus the number of decision-making units. As the BRAM utilization rate increases, the number of instantiated decision-making modules rises linearly, which aligns with the pattern identified in this paper. To facilitate further functional expansion of the modules in subsequent stages, we have chosen to instantiate 40 relevant modules, reserving 30% of the space for expanded functionalities.

Figure 13 depicts the traffic distribution of the first 20 decision logic units, with a total data volume of 16,526,469. By employing the MurmurHash3 algorithm, the average deviation can be reduced to 30.1%, demonstrating a relatively good load-balancing effect.

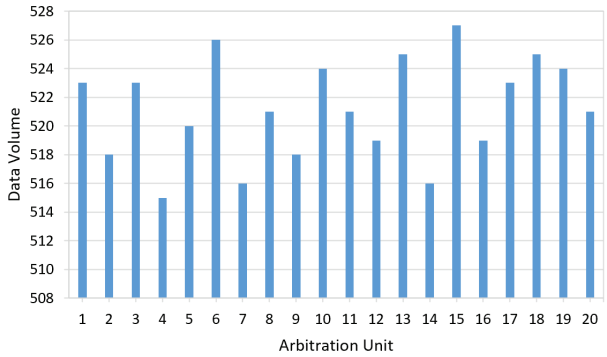

Figure 14: Segment of Register Data Distribution.

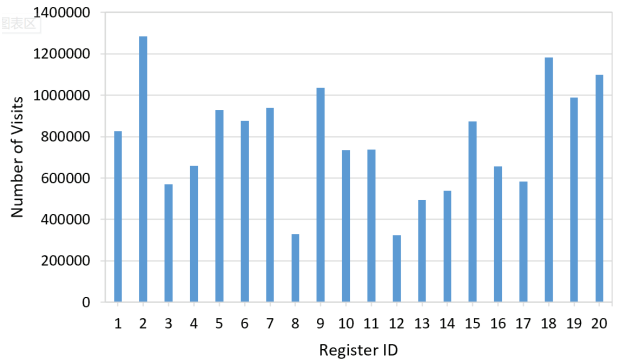

Figure 15: Distribution of Register I/O Traffic.

Figure 14 illustrates the number of rules allocated to the first 20 modules within the register module. The register parallelization algorithm results in a maximum deviation of only 1.44% and an average deviation of merely 0.5%, indicating that the algorithm employed for accessing and determining logical rules in the register module achieves a relatively good load-balancing effect.

Figure 15 shows the number of read operations sent from the decision-making module to the register module. The data indicates an average deviation of 28.11%, with a relatively even I/O distribution.

FPGAs adopt a hardware-based parallel pipeline design, where the entire process from instruction entry into the FPGA to execution completion is accomplished in parallel and in real time by hardware. In contrast, CPUs employ a time-slice-based serial instruction scheduling mechanism, which requires the operating system to perform a large number of polling

DOI: 10.36838/IJHSR813.43

scheduling operations. The shared bus mechanism adopted by CPUs also greatly reduces the overall efficiency of the system, especially in high-concurrency environments where the inefficiency becomes more prominent.

Table 3: System Latency.

	Latency in clock cycles per (unit/instance)
FPGA	7
CPU	180+

Table 3 presents the comparative results of system latency. The data analysis is as follows:

1. The message scheduling module comprises protocol polling, message reception, and message transmission, totaling 4 clock cycles.

2. The signal identification module includes message reception, CRC verification, header separation, type retrieval, type comparison, and hash processing. Among these, hash processing is the most time-consuming, taking a total of 5 clock cycles from message reception to state machine processing and then transmission.

3. In the decision logic unit, signal reception, retrieval of temporary register data, signal comparison, and signal transmission together take 7 clock cycles.

4. The CPU interface module involves receiving abnormal signals, transmitting abnormal signals upward, and sending the latest prediction signals to the temporary register module, totaling 2 clock cycles.

During FPGA system simulation, the total latency from data input to data output is 7 clock cycles. With a primary frequency of 250 MHz, the system latency time is 35ns. The latency time for the CPU to process a single message is significantly longer than that of the FPGA. This is because the CPU needs to fetch and execute instructions one by one to complete its tasks, typically requiring multiple instructions for a single operation, such as loading operations from memory, executing arithmetic instructions, and storing results back into memory. In contrast, the FPGA can utilize pipeline design to allow data to enter the processing unit every clock cycle, producing results sequentially after 7 clock cycles, thereby demonstrating significantly superior processing efficiency compared to the CPU.

Table 4: Comparison of System Throughput.

	Throughput (Gbps)
FPGA	16
CPU	3.6

Table 4 presents the throughput comparison results between the FPGA and CPU. In this context, throughput refers to the maximum amount of message data that the anomaly processing module can handle per unit time. To achieve optimal throughput performance, the system instantiates the anomaly processing module to the maximum extent possible to eliminate queuing time. It can process 64-bit data per clock cycle, resulting in a final throughput of up to $64 * 250 * 10^6 = 16$ Gbps.

Unlike FPGAs, which can be customized with a large number of parallel processing units for specific tasks, CPUs process a single message by executing a series of instructions. Taking a modern high-performance CPU with a clock frequency of 3.0 GHz as an example, it can execute 1.5 instructions per clock cycle, yielding an instruction throughput rate of $3.0 * 10^9 * 1.5 = 4.5 * 10^9$. The data throughput rate is calculated as $(4.5 * 10^9 / N * 8)$ bps, where N represents the number of instructions required per 1B of data. In this system operation, processing 8B of data requires no fewer than 100 instructions, meaning N is not less than 10. Therefore, the maximum CPU throughput is 3.6 Gbps.

The above experiments demonstrate that the FPGA system effectively leverages the advantages of parallel processing. It can adaptively generate logic instances of decision-making units based on the available FPGA resources, resulting in relatively balanced loads among decision-making units and balanced I/O among cache units. Compared to the CPU, the FPGA system reduces processing latency by a factor of 25 and increases throughput by approximately 4.5 times.

Conclusion

This paper proposes a high-performance decision-making functional design based on an FPGA for spacecraft testing systems. This design identifies a linear relationship between available resources and the number of logic units, addressing the issue of adaptive logic resource generation. A hash algorithm based on MurmurHash3 is devised to achieve balanced distribution of semaphores across parallel logic units. Additionally, cache mapping and rapid lookup algorithms are designed to enable high-speed access to parallel caches. Experimental validation demonstrates the effectiveness of this system design in handling high-concurrency requirements.

Future work will further explore online learning and collaborative mechanisms between FPGAs and CPUs. We can implement online learning from two aspects: interface design and task strategy. A synchronous message interface is adopted between the CPU and FPGA to standardize their operation timing. Two synchronization processes are defined:

1. Data synchronization from the FPGA to the CPU, including synchronization start and synchronization end messages.

2. Rule writing from the CPU to the FPGA, including write start and write end messages.

When receiving a data synchronization task, the CPU allocates an independent cache and creates a dedicated session context to ensure reliable synchronization. When receiving a rule-writing task, the FPGA suspends decision-making and instructs the front-end signal acquisition unit to pause transmission. The system resumes operation only after writing is completed, ensuring decisions are based on the latest rules and avoiding misjudgment. These strategies effectively improve the robustness and intelligence of the system.

Acknowledgments

I am deeply grateful to my high school for its support in experimental design, laboratory setup, testing, and other aspects of my research.

References

1. S. Y. Zhou, "Research on an automatic testing technology of incremental angular encoder based on LabVIEW," in Annual Meeting of the China Aviation Industry Technical Equipment Engineering Association, Qingdao, China, 2021, pp. 147–151.
2. JM Pires, M Pantoquilho, N Viana. "Real-Time Decision Support System for Space Missions Control. [C]", Information and Knowledge Engineering, 2004, pp.152-160.
3. Flynn, M. J. (1972). "Some Computer Organizations and Their Effectiveness". IEEE Transactions on Computers, C-21(9), pp. 948-960.
4. M. Horowitz *et al.*, "The Energy Wall and the ILP Wall and their implications for the future of multiprocessors," IEEE International Symposium on Computer Architecture, San Diego, CA, USA, 2007, pp. 1-1
5. Lam, M. S., & Wilson, R. P. (1992). "Limits of control flow on parallelism". In Proceedings of the 19th Annual International Symposium on Computer Architecture (ISCA), pp. 46–57
6. Wall, D. W. (1991). "Limits of instruction-level parallelism". In Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), pp. 176–188
7. W. A. Wulf and S. A. McKee, "Hitting the memory wall: Implications of the obvious," ACM SIGARCH Computer Architecture News, Vol. 23, No.1, 1995, pp. 20–24
8. Saulsbury, A., Pong, F., & Nowatzyk, A. (1996). "Missing the Memory Wall: The Case for Processor/Memory Integration". In Proceedings of the 23rd Annual International Symposium on Computer Architecture (ISCA '96), pp. 90-101
9. Wenhao Lin; Xinshi Zang; Zewen Li; Evangeline F. Y. Young. "An Open-Source High-Concurrency and High-Performance Parallel Router for UltraScale FPGAs", IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. 45, No.2, 2026, pp.659-672
10. Jia Xu, Han Pu, Dong Wang, "Sparse Convolution FPGA Accelerator Based on Multi-Bank Hash Selection", Micromachines, Vol. 16, No.1, 2025, p.22
11. M Mitzenmacher. "The Power of Two Choices in Randomized Load Balancing[J]", IEEE Transactions on Parallel and Distributed Systems, 2001, 12(10): pp. 1094-1104.
12. Oliver A, Sebastian H, Gerhard P F, Benjamin S, Thomas K, Tomas K, Wolfgang L, *et al.* "HASHI: an Application Specific Instruction Set Extension for Hashing. [C]", Very Large Data Bases Conference, 2014: 25-33.
13. Pike, G., & Alakuijala, J. (2011). Google. CityHash. Available at: https://googleblog.com/2011/04/introducing-cityhash.html, (accessed: 2025)
14. Weiqing Y, Chengxuan J, Xian L, Zhiwei B, *et al.* "A FNV Based IPv6 Address Autoconfiguration Scheme for Power IoT Sensory Device[J]", 2022 2nd International Conference on Networking Systems of AI (INSAI), 2022: 216-221.
15. Jean-Philippe Aumasson, Daniel J. Bernstein. "SipHash: A Fast Short-Input PRF.[J]", Lecture Notes in Computer Science, 2012: 489-508.
16. Guy L. Steele, Sebastiano Vigna. "LXM: Better Splittable Pseudorandom Number Generators (and Almost As Fast)[J]", Proceedings of the ACM on Programming Languages, 2021, 5
17. Mathys Walma. "Pipelined Feed-Forward Cyclic Redundancy Check (CRC) Calculation[J]", Computing Research Repository, 2006
18. Lin, Anthony. "Binary Search Algorithm.", WikiJournal of Science, Vol. 2, No.1, WikiMedia Foundation, 2019, pp. 1–13

Author

Yuqian Han is currently a 10-grade student at Jinling High School, Hexi Branch in Nanjing, China. He has a strong interest in electronic engineering and computer science, In his spare time, he enjoys participating in scientific experiments to solve practical problems.

DOI: 10.36838/IJHSR813.43