Design and implementation of reorder buffer in superscalar pipeline processor

Chenyang Mao *

Silc Business School, Shanghai University, Shanghai, China

* Corresponding Author Email: 1797281318@shu.edu.com

Abstract. With the increasing demand for high performance and low power consumption in System-on-Chip (SoC) scenarios such as mobile terminals and servers, superscalar processors have become the core computing unit of high-performance SoCs by virtue of their multi-instruction parallel capabilities. However, the out-of-order execution mechanism of superscalar processors leads to contradictions between "out-of-order completion" and "in-order commit" of instructions, as well as difficulties in state recovery when branch prediction fails or exceptions occur. To solve these problems, this paper designs and implements a Reorder Buffer (ROB) module for 4-way issue superscalar pipelines. The ROB module adopts a parameterized design (data width 64bit, depth 32 entries) to realize functions such as instruction temporary storage, in-order commit, data forwarding, branch flush, and exception handling. Functional verification is carried out, covering 9 core scenarios including basic instruction issue, execution result writeback, branch flush, and post-flush instruction re-issue. Based on the 7nm FinFET process library for logic synthesis, the results show that the module achieves complete sequential convergence (slack 0.003ns) with a maximum operating frequency of 1607.72MHz, a total area of 10760.880μm², and a total dynamic power consumption of 4.6127mW. This ROB module can stably support the out-of-order execution and in-order commit of 4 instructions per cycle, providing a reliable core structure for high-performance 4-way superscalar processors.

Keywords: Superscalar Processor, Reorder Buffer (ROB), Performance optimization.

1. Introduction

In modern electronic systems, System-on-Chip (SoC) has become the mainstream paradigm of integrated circuit design - it integrates multiple functional modules such as processors, memories, and interfaces onto a single chip to meet the comprehensive demands of "high performance, low power consumption, and small size" in scenarios such as mobile terminals, servers, and Internet of Things devices. The processor, as the computing core of SoC, directly determines the processing capability of the entire chip. Superscalar processors break through the performance bottleneck of traditional single-instruction-stream processors by "simultaneously issuing and executing multiple instructions", and are the core architecture commonly adopted in current high-performance SoCs.

However, the out-of-order execution mechanism of superscalar processors presents several challenges. Out-of-order execution allows instructions to be executed in advance when their operands are ready, fully exploiting the instruction-level parallelism in programs. However, the "completion order" of instructions can be out of sync with the "original program order" (instructions issued earlier may be delayed due to unmet dependencies, while those issued later may complete earlier). This asynchrony is fundamentally at odds with the in-order commit requirement for program semantic correctness - if the results of out-of-order completed instructions are directly written back to registers or memory, it will disrupt the dependencies of subsequent instructions on the results of previous instructions, leading to program errors. Additionally, when branch prediction errors or instruction exceptions occur, quickly rolling back the erroneous instruction stream and restoring the processor to a correct state is also a key issue for ensuring system stability.

The Reorder Buffer (ROB) is precisely the core structure born to address these contradictions. It provides temporary storage and status tracking space for each issued instruction: all instructions enter the ROB for temporary storage after being issued, recording their execution status (such as whether they are completed and whether their results are valid); only when an instruction meets the conditions

that the preceding instructions in the program sequence have been committed, it has completed execution, and there are no exceptions, will it be committed from the ROB (writing the results back to the register file or memory); if a branch prediction error or exception occurs, the ROB can quickly mark all instructions on the erroneous path as invalid and trigger the acquisition of a new correct instruction stream, achieving the consistency recovery of the processor state.

In the complex application scenarios of SoC, the design efficiency of the ROB directly affects the performance ceiling and stability of the superscalar processor: for mobile SoCs, the low power consumption and high throughput design of the ROB determine the device's battery life and response speed; for server SoCs, the ROB's scheduling capability for large-scale parallel instructions directly affects the computing throughput; for SoCs with high real-time requirements (such as autonomous driving chips), the error recovery speed of the ROB determines the system's reliability. Therefore, in-depth research on the design and implementation of the ROB has crucial engineering value and theoretical significance for improving the overall performance of SoC.

The first part aims to introduce the research background of this paper, the second part will introduce some background knowledge and research methods used in the study, the third part will introduce the design details of specific modules, the fourth part will present the test code and logic synthesis results to demonstrate the correctness of the module functions, and finally summarize the research of this paper and outline the prospects for the future.

2. Preliminary

Before introducing the basic knowledge, it is necessary to conduct research by reviewing the literature on ROB and superscalar pipelines. To address the problem of ROB blocking in superscalar processors caused by long instruction execution delays and consecutive decoding, Xi'an University of Posts and Telecommunications proposed an instruction out-of-order submission mechanism. This mechanism designs a configurable multi-buffer instruction submission structure, which realizes the classified retirement of memory operation instructions and ALU instructions [1]. National University of Defense Technology employed a performance analysis method based on counters to analyze the processor, and discovered that the instruction renaming pauses were relatively frequent due to the insufficient number of renaming registers. This issue was particularly severe in the CFP2000 program. Based on this, it was proposed to increase the number of renaming registers to enhance the processor's performance, and a Verilog code with configurable renaming register count was implemented [2]. JSS Research Foundation proposed a 16GHz ultra-large-scale pipelined inner product operation unit, which is suitable for both signed and unsigned number operations. This design adopts a five-stage pipelined architecture and consists of four parallel working 8×8 multipliers. The ultra-large-scale pipelined architecture can concurrently complete four 8×8 product operations within three clock cycles [3]. National University of Singapore proposed a multi-threaded superscalar processor (MCMS), which is an extended version of the traditional superscalar architecture, aiming to support multi-threading technology. At the same time, the research also proposed a hardware implementation scheme for the multi-threaded structure. Based on the trajectory-driven simulation results, it was shown that after adopting MCMS, the instruction parallelism was indeed significantly improved. Under the condition of comparable hardware resources, the MCMS processor with four hardware contexts can achieve an acceleration effect of up to 2.5 times [4]. Chip Multi-Processor (CMP) is designed to enhance system performance by improving instruction-level and thread-level parallelism. However, according to previous research results, CMP outperforms superscalar processors only in floating-point operation applications. Therefore, National Changhua University of Education proposed a new type of microprocessor that supports two execution modes, allowing users to manually select the appropriate mode to run applications based on the characteristics of the workload [5]. Lahore University of Management Sciences evaluated the feasibility of using a general-purpose superscalar architecture as the core computing engine for high-performance digital signal processing algorithms. Unlike traditional dedicated signal processing engines, this study adopted a superscalar

processor architecture specifically designed for SRC. A modeling tool based on iterative SimpleScalar was developed to analyze various parameters of the superscalar processor. By comprehensively considering power consumption and performance indicators, an efficient solution was ultimately designed [6]. Simulating the trace-driven behavior of out-of-order superscalar processors is no easy task. When the trace only includes some of the executed instructions for simplification, the combination of this dynamic characteristic with the static characteristic of the trace may lead to significant deviations in the results. Therefore, University of Pittsburgh proposed and comprehensively evaluated the Pairwise Dependency Cache Miss Model (PDCM) - a framework for achieving fast and accurate trace-driven simulation [7]. Zhejiang University addressed the issue of slow pipeline retirement caused by the long delay of instructions occupying the top of the reordering cache for an extended period in superscalar processors, proposing an efficient retirement mechanism that quickly retires instructions without exception risks and writes the operation results out of order. This solution separates the result cache from the reordering cache, with the result cache serving as the cache for writing back the operation results, and the reordering cache responsible for the sequential retirement of instructions and the maintenance of precise exceptions [8]. Beijing Microelectronics Technology Institute conducted an analysis of the relationship between the pipeline performance in superscalar processors and the factors related to branch prediction, as well as a comparison with the performance of scalar processors. Based on reasonable assumptions, they obtained the analytical function of the relationship between them. From different perspectives, they deeply analyzed the impact of branch prediction accuracy, the proportion of branch instructions, the moment of branch prediction errors, the size of program instructions, the pipeline depth, and the width of instruction parallel emission on pipeline performance. This provides a reference for the research on modern RISV processors and the study of branch prediction technology [9].

The core mechanism of modern superscalar processors to achieve multi-instruction parallelism lies in the ability to issue multiple instructions within a single clock cycle and support their out-of-order execution. In contrast, traditional pipeline technology is constrained by an in-order issue mechanism - instructions must strictly follow the original program sequence. Once a stall occurs in the pipeline (for example, due to data or control dependencies between two adjacent instructions), all subsequent instructions are blocked and cannot proceed. This limitation is particularly prominent in processors with multiple functional units: when some units are idle due to waiting for previous instructions, the parallel potential of other functional units is wasted, resulting in low overall execution efficiency.

Out-of-order execution technology breaks through this limitation: it allows instructions to deviate from the original issue sequence, and as long as the operands of a certain instruction are ready (i.e., not dependent on the results of other unfinished instructions), it can be issued out-of-order to the corresponding functional unit for execution. This dynamic scheduling approach minimizes stalls caused by instruction dependencies, fully utilizes the multi-functional unit resources of the processor, and significantly improves the overall efficiency of instruction execution.

The core of a superscalar processor is the exploitation of Instruction-Level Parallelism (ILP), which includes an instruction fetch unit, a decode unit, an issue unit, multiple functional units (such as ALUs, multipliers, Load/Store units, etc.), and a register file which reads the operands from the physical register file (PRF) based on the instruction selected by the arbitration circuit. The instruction fetch unit retrieves multiple instructions from the instruction cache (for example, a four-issue processor fetches four instructions each time), and after decoding, the issue unit analyzes the readiness of the instruction operands and the occupancy of the functional units: if the operands of an instruction are ready and the functional unit is idle, it will be issued out-of-order to the functional unit for execution (for example, independent ADD and SUB instructions can be issued simultaneously). However, out-of-order execution only addresses the parallelism in the execution stage; the write-back and commit of instructions must strictly follow the program sequence - instructions that appear earlier in the program must have their results written back first, otherwise subsequent instructions that depend on these results will read incorrect data. This requirement of "out-of-order execution, in-order

commit" is precisely the core scenario where the ROB plays a crucial role. Figure 1 displays a typical superscalar processor microarchitecture consisting of nine components: Fetch, Decode, Register Renaming, Dispatch, Issue, Register File, Execute, Write-back, ROB and Commit.

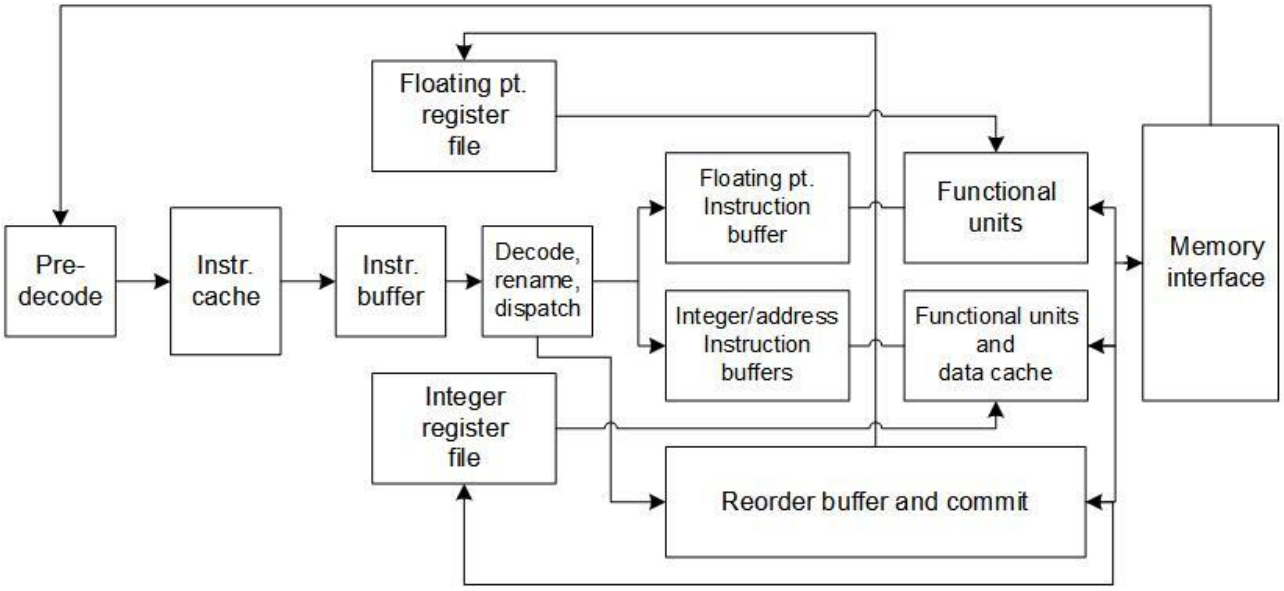

Fig 1. Typical superscalar processor microarchitecture

The data and control dependencies that occur during out-of-order execution are essentially due to multiple instructions sharing register resources to store different variable values, leading to dependencies between instructions due to resource competition. If register renaming technology is used to allocate independent registers for different variables, the originally dependent instructions can be executed in parallel, avoiding performance loss. The Tomasulo algorithm is a dynamic scheduling algorithm based on this idea, which achieves flexible instruction scheduling through register renaming and has been widely used in superscalar pipelines.

Although out-of-order execution can improve efficiency, it can cause two types of conflicts: data conflicts and control conflicts. Data conflicts can be resolved through the dynamic scheduling mechanism of the reservation station - the reservation station tracks the readiness status of the operands of instructions and only schedules the execution of instructions when the required data is ready; control conflicts mainly rely on branch prediction technology to handle, but branch prediction is bound to have failures. If only out-of-order execution of instructions is allowed without guaranteeing the ordered write-back of results, once the prediction fails, the processor state cannot be correctly restored. Therefore, on the basis of the Tomasulo algorithm, a speculative execution mechanism - ROB - is introduced at the hardware level.

One of the major functions of the ROB is to provide source operands to the Arithmetic Logic Unit (ALU). The ROB is essentially an extension of the register renaming technology, providing additional temporary storage resources, similar to the role of the reservation station in extending the register file in the Tomasulo algorithm. The core function of the ROB is to temporarily store the execution results of instructions during the gap between the completion of instruction execution and the formal submission (writing back to the architectural registers). During this process, the ROB can directly provide the required source operands for subsequent instructions, similar to the operand supply mechanism of the reservation station in the Tomasulo algorithm.

Unlike the Tomasulo algorithm, where instructions update the register file immediately upon completion and subsequent instructions directly read the results from the register file, in the ROB mechanism, data registers are not updated until instructions are committed. Instead, the ROB temporarily provides operands. This mechanism is similar to a bypass path: when the result of an instruction is temporarily stored in the ROB, it can be directly forwarded to the execution unit waiting for that data, without going through the register file. This design not only ensures the flexibility of

out-of-order execution but also guarantees the recoverability of the state in case of branch prediction failure through an ordered commit mechanism. Figure 2 illustrates the provision of the source operand.

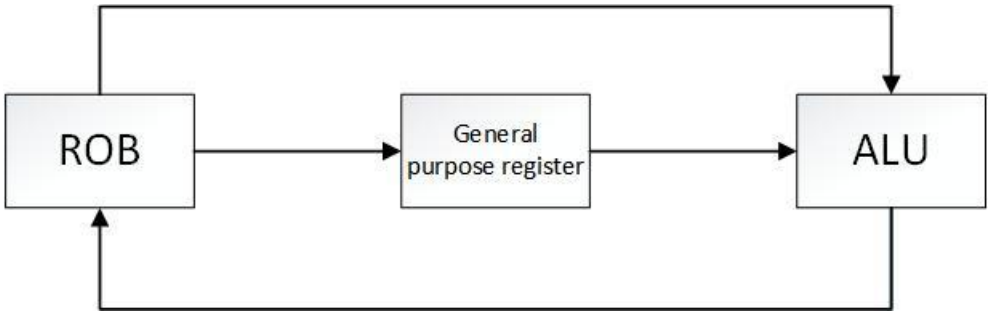

Fig 2. Provision of the source operand

Another major function of ROB is to ensure the ordered submission of instructions and the recovery of execution in case of abnormal behavior [10]. During the submission stage, the results must be submitted to the designated storage units in the original order of the program at the beginning. After instruction decoding and before out-of-order issue, each instruction has already applied for an empty entry in the reordering buffer in sequence and recorded the key information of the instruction (type, destination physical register, PC, operands, etc.) and marked it as "allocated but not executed". When the instruction execution is completed, the reordering buffer can ensure that the final results are submitted in order. Once an interruption or abnormal behavior occurs, all remaining execution results that have not been submitted will be discarded, and the system will enter the interruption handling or restart execution from the correct instruction.

Overall, the ROB is essentially a first-in-first-out (FIFO) buffer structure, but each entry has the ability to track complex states. The workflow is divided into four stages: when an instruction is enqueued, the ROB allocates an idle entry for the issued instruction, records the key information of the instruction (type, destination physical register, PC, operands, etc.) and marks it as "allocated but not executed"; when the execution result is written back, after the functional unit completes the execution, the result is written back to the result field of the corresponding ROB entry and marked as "execution completed" (branch/load-store instructions also need to record jump, memory address, etc. information); when submitting as needed, the ROB head pointer points to the "earliest entered and pending submission" entry, continuously checks the head entry, if "execution completed and no exception", then write back the result, release the entry and move the pointer backward, if not completed or with an exception, then wait/trigger subsequent processing (since submission is in order, even if subsequent instructions are completed in advance, they must wait for the submission of the preceding instructions); when recovering from errors, if there is a branch prediction error or instruction exception, the ROB receives a refresh signal, marks subsequent entries starting from the erroneous instruction entry as "invalid", stops submission and updates the fetch PC, and re-obtains the instruction stream to quickly roll back the state.

3. ROB Module Design

This experiment is based on the 4-way ROB module for instruction reordering. It verifies the correctness of its core functions in the out-of-order execution processor, including: verifying the sequentiality of the entire process of instruction emission, execution, and submission (out-of-order execution, sequential submission); verifying the processing capability of the data forwarding logic for register dependencies; verifying the ROB refresh function when branch prediction errors occur; verifying the reporting and submission logic for exception handling (such as division by zero exception); verifying the instruction scheduling capability under the ROB full-load boundary condition; familiarizing with the interface interaction between the ROB module and other units of the processor (emission unit, execution unit, branch prediction unit, submission unit). The ROB module, as the core of the processor's out-of-order execution, interacts externally through standardized

interfaces with the emission unit, execution unit, branch prediction unit, and submission unit, and internally implements functions such as instruction storage, arbitration, control, and forwarding through sub-modules. The overall structure is as follows. Figure 3 shows the overall structure of ROB module.

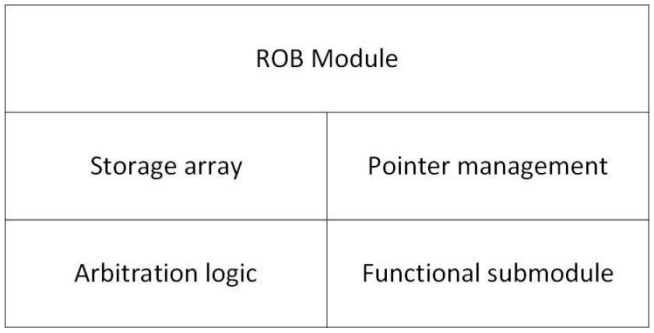

Fig 3. Overall Structure

The ROB is the core module of the superscalar pipeline and needs to interact with multiple modules such as the emission unit, execution unit, branch prediction unit, etc. Its ports are classified by function into seven categories: control signals, instruction emission interface, execution result write-back interface, branch refresh interface, data forwarding interface, submission result interface, and debugging interface. The module adopts parameterized design to enhance flexibility, and the core parameters include: data width DATA_WIDTH = 64 (suitable for 64-bit integer/floating-point operations and address space), ROB depth ROB_DEPTH = 32 (can buffer 32 instructions simultaneously, balancing pipeline delay and buffering capacity), pointer width PTR_WIDTH = 5 (derived from 32 depth, meeting the requirements of loop counting), register address width REG_ADDR_WID = 5 (supports 32 physical register addressing, suitable for register renaming mechanism), PC width PC_WIDTH = 64 (supports 64-bit virtual address space, conforming to modern processor design).

The figure 4 fully depicts the hardware implementation logic of the ROB module in the code. Each module and signal flow can be found corresponding implementation in the code. After the instruction emission interface at the top of the module receives signals such as issue_vld (issue valid), issue_inst_vld[3:0] (4-bit instruction valid), etc., the write_port_arb combinational logic in the code will first count the number of valid instructions (valid_cnt), and then determine the ROB space based on rob_used_cnt (the number of used entries calculated by the read-write pointers). If the remaining space is sufficient, it will fully receive (write_port_arb = issue_inst_vld), otherwise, it will partially receive according to the priority, and finally, through the sequential logic, the instruction information (issue_inst_pc, issue_inst_rd_phy, etc.) will be written into the corresponding entry of the rob_array, while setting valid = 1, ready = 0, and updating write_ptr. After the write-back interface receives exec_vld[3:0] (4-bit result valid), exec_rob_ptr (target entry pointer), and other signals, the exec_port_arb combinational logic will resolve the write-back conflicts (such as checking if it points to the same ROB entry) according to the port priority ($0 \rightarrow 1 \rightarrow 2 \rightarrow 3$). After the arbitration passes, the sequential logic will update the result (execution result), branch_taken (branch direction), except (exception flag), and other fields of the corresponding entry in rob_array, and set ready = 1. The data forwarding module in the combinational logic first parses exec_rs1_phy/exec_rs2_phy (the source physical registers required by the execution unit), and then traverses the rob_array. If it finds an entry with "valid = 1 and ready = 1" and the rd_phy matches, it will forward its result to rob_exec_data1/2 and set rob_exec_data1/2_vld valid. This exactly corresponds to the nested for loop traversal logic in the code. The pointer management module calculates rob_used_cnt (write_ptr >= read_ptr is write_ptr - read_ptr + 1, otherwise it is ROB_DEPTH - read_ptr + write_ptr + 1) through the combinational logic, and then generates rob_full (full) and rob_almost_full (almost full) states, and simultaneously pre-calculates next_write_ptr and read_ptr_next for subsequent update use. "Commit control module" in the sequential logic first checks up to 4 entries starting from read_ptr through a for loop, counts the "valid = 1 and ready = 1" commit_cnt, and then outputs the information of these

entries (rd_phy, result, pc, etc.) in sequence to the commit_rd_phy, commit_result and other commit interfaces, while clearing the valid of the already committed entries and updating read_ptr = (read_ptr + commit_cnt) % ROB_DEPTH. When the bp_flush_vld signal is active, the branch refresh module traverses the rob_array through sequential logic to clear the valid status of the entries after bp_flush_ptr, and resets write_ptr = bp_flush_ptr + 1 and read_ptr = 0. This is consistent with the logic of the if (bp_flush_vld) branch in the code. The branch feedback module checks the entries pointed to by read_ptr in the sequential logic. If they are "valid = 1, ready = 1, and is_branch = 1", it outputs rob_branch_pc (branch instruction PC), rob_branch_taken (actual jump direction), and rob_branch_update_vld (feedback valid), which is used to update the branch predictor. The debug interface directly extracts the internal states of rob_array's valid/ready bits, write_ptr, read_ptr, etc., corresponding to the dbg_rob_valid/dbg_rob_ready generated by the generate block and the directly assigned dbg_write_ptr/dbg_read_ptr in the code. The entire module is driven by the clock and asynchronously resets rob_array and all pointers when reset, forming a complete "instruction emission → storage → result write-back → data forwarding → sequential submission → branch feedback/refresh" loop, which is one-to-one corresponding to the code implementation. Figure 4 in the following illustrates the ROB module architecture.

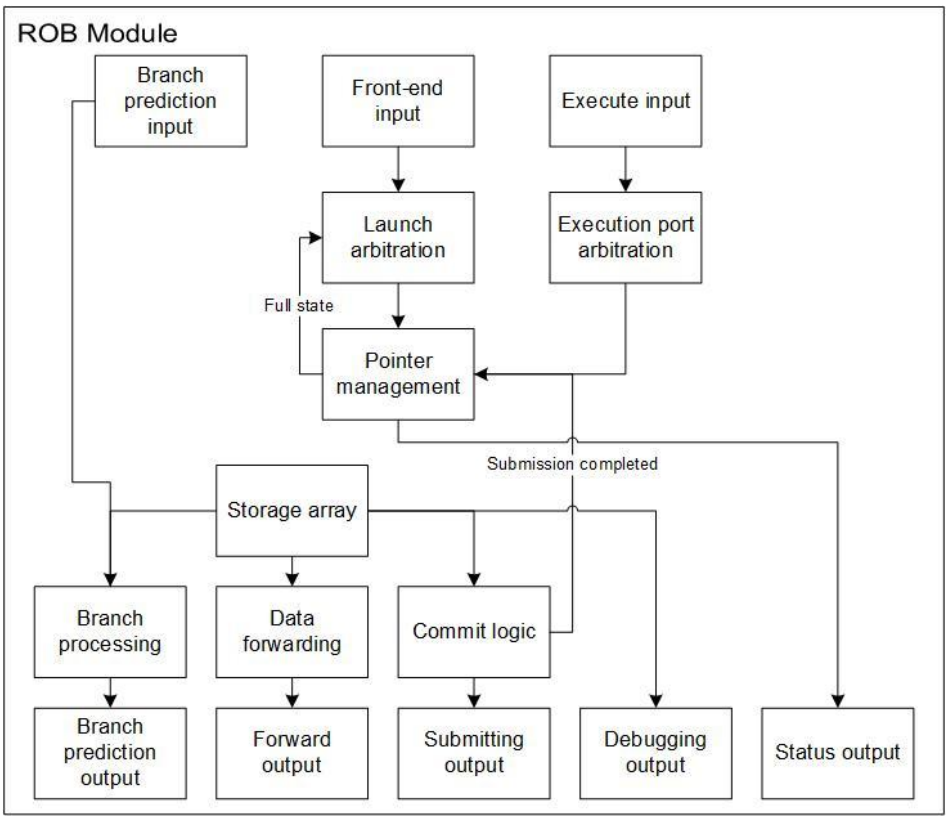

Fig 4. ROB Module architecture

4. Results and Analysis

The testbench code is written to verify the core functions of the ROB module in a 4-way superscalar processor. It conducts comprehensive tests on the ROB's out-of-order reception and sequential submission mechanisms by simulating scenarios such as instruction emission, execution result write-back, and branch refreshing. The specific implementation is as follows: Firstly, initialize signals and set up the environment, define parameters consistent with the ROB module (such as data width and depth), declare all interface signals, generate a 100MHz clock (period 10ns) as the module's synchronization reference, generate a low-level valid asynchronous reset signal to support initialization and mid-test reset, and instantiate the ROB module (DUT) to complete the connection of all signals; Secondly, design the core test scenarios, including Scenario 1 (basic instruction

emission, emitting 4 ordinary instructions (non-branching, non-memory access) at once, verifying the ROB's entry allocation and pointer update functions), Scenario 2 (partial instruction emission, emitting 2 instructions (only the lower 2 bits are valid), verifying the ROB's handling capability for non-consecutive instructions), Scenario 3 (writing back execution result to entry 1, writing the execution result to entry 0 and 2, verifying the correct writing of the result to the corresponding entry's result field and setting the ready flag), Scenario 4 (writing back execution result to entry 2, writing the results to entries 1, 3, and 4, verifying the arbitration logic for multi-port write-back (correct update when there is no conflict)), Scenario 5 (branch instruction emission, emitting 1 branch instruction, verifying the ROB's marking of special instruction types (is_branch field)), Scenario 6 (branch result processing, writing back the execution result of the branch instruction (including the jump direction branch_taken), verifying the storage of branch information), Scenario 7 (branch refreshing, triggering a branch prediction error (bp_flush_vld = 1), verifying the ROB's clearing of incorrect path instructions (entries after bp_flush_ptr have valid = 0) and updating the pointer), Scenario 8 (instruction emission after refreshing, emitting new instructions after branch refreshing, verifying the ROB's resumption of receiving instructions from the correct path), Scenario 9 (result processing after refreshing, writing back the results of the refreshed new instructions, verifying the normal execution flow after refreshing), and asynchronous reset test (triggering reset during normal operation to verify the ROB's function of clearing all entries and resetting the pointer). Table 1 shows the synthesis results based on 7nm FinFET process.

Table 1. The synthesis results based on 7nm FinFET process

Name	Parameter values
Timing margin	0.003ns
Core operating frequency	1607.72MHz
Combinational logic area	7596.696μm²
Sequential logic area	3164.184μm²
Total area	10760.88μm²
Cell internal power	3.6895mW
Net switching power	923.1589μW
Total dynamic power	4.6127mW
All ports number	2519

As shown in the table above, through the sequential synthesis of the industry process under the 7nm FinFET process technology, this ROB module performs exceptionally well in terms of timing, area, and power consumption. In terms of timing, the slack of the core clock is 0.003ns, achieving sequential convergence without violations, and the maximum operating frequency can reach 1607.72MHz, which can fully support the parallel processing requirement of 4-way superscalar pipelines with 4 instructions per cycle, providing sufficient timing margin for the "emission-execution-submit" full process; In terms of area, the total area is 10760.880μm². Thanks to the high transistor density of the 7nm technology, the 32-depth entry storage (including multi-dimensional information) and multi-port control logic are compactly integrated without excessive area expansion, adapting to the chip layout requirements of high-performance processors; In terms of power consumption, the total dynamic power is only 4.6127mW, with 80% of the power consumption coming from the core logic unit (activity of the active core logic unit), and 20% from the interconnect switch power consumption. This reflects the low power consumption characteristics of 7nm technology, which effectively controls power consumption during high-frequency operation at 1.6GHz, helping the processor balance high performance and energy efficiency. Overall, the integrated performance of this ROB module can fully meet the design requirements of high-performance 4-way superscalar processors.

5. Conclusion

In conclusion, the ROB module processed by 7nm technology demonstrates excellent performance balance: in terms of timing, the maximum operating frequency of 1607.72 MHz and the timing margin (0.003ns) ensure the high throughput requirements of the 4-way superscalar pipeline, enabling stable support for out-of-order execution and sequential submission of 4 instructions per cycle; in terms of area and power consumption, the compact integration (10760.880μm²) and low dynamic power consumption (4.6127mW) fully utilize the high density and low leakage characteristics of the 7nm process, laying the foundation for the energy efficiency optimization of high-performance processors. Overall, this module already possesses the core capability to support mid-to-high-end 4-way superscalar processors, and its design logic (such as multi-port arbitration, branch refreshing mechanism) can be directly reused in similar architectures.

In terms of future prospects, further optimization can be achieved in three aspects: Firstly, in terms of process upgrade, if migrated to 3nm and more advanced processes, with higher transistor density and lower latency, the operating frequency can be increased to above 2 GHz, and the unit area power consumption can be further reduced, adapting to the requirements of next-generation ultra-high-frequency processors; Secondly, in terms of function expansion, the ROB depth (such as 64 entries) can be expanded to support deeper pipelines, and the adaptation to vector instructions and complex memory operations can be enhanced, improving the support for scenarios such as AI computing and big data processing; Thirdly, in terms of energy efficiency optimization, dynamic voltage and frequency scaling (DVFS) and low power modes can be introduced, combined with dynamic closure of idle entry circuits based on instruction activity prediction, to ensure performance while further reducing standby power consumption. Additionally, through collaborative design optimization with the branch prediction unit and register renaming module, cross-module delays can be reduced, improving the operational efficiency of the entire processor core, providing more flexible hardware support for emerging scenarios such as heterogeneous computing and edge computing.

References

[1] LI Zhao, LIU Youyao, JIAO Jiye, et al. Superscalar processor out-of-order submit mechanism research and design. Computer engineering, 2021, 47 (4): 180-186.

[2] LI Wenzhe. Design and optimization of Register Renaming Mechanism for out-of-order superscalar Processor. National University of Defense Technology, 2015.

[3] SRavindra P. Rajput, M.N. Shanmukha Swamy, Superscalar pipelined inner product computation unit for signed unsigned number, Perspectives in Science, 2016, 8: 606-610.

[4] K.S. Loh, W.F. Wong, Multiple context multithreaded superscalar processor architecture, Journal of Systems Architecture, 2000, 46(3): 243-258.

[5] Chao-Chin Wu, Embedding a superscalar processor onto a chip multiprocessor, Microprocessors and Microsystems, 2004, 28(4): 147-156.

[6] Faheem Sheikh, Shahid Masud, Rehan Ahmed, Superscalar architecture design for high performance DSP operations, Microprocessors and Microsystems, 2009, 33(2): 154-160.

[7] Kiyeon Lee, Sangyeun Cho, Accurately modeling superscalar processor performance with reduced trace, Journal of Parallel and Distributed Computing, 2013, 73(4): 509-521.

[8] Li Xiaoming, Yang Jun, Meng Jianyi. A Retirement Scheme for ROB to Achieve Fast Instruction Completion. Computer Engineering and Applications, 2015, 51(24): 40-44.

[9] Zhang Shiyuan, Yu Lixin. Analysis of the Impact of Branch Prediction on the Performance of Superscalar Pipelines. Microelectronics & Computer, 2015, 32(08): 167-171 + 176.

[10] Zhang He. The study of the reorder buffer of the superscalar processor. Information aspect, 2009, 28 (16): 16-18.