electronics

Article

Specially-Designed Out-of-Order Processor Architecture for Microcontrollers

Yunhao Hu [1,2], Jie Chen [1], Kaiben Zhu [1], Qijun Xing [1], Wei Liu [1,2,*], Junfeng Shen [3,*] and Ge Gao [4,*]

[1] School of Physics and Technology, Wuhan University, Wuhan 430072, China
[2] Hubei Luojia Laboratory, Wuhan 430072, China
[3] College of Biology and Agricultural Resources, Huanggang Normal University, Huanggang 438000, China
[4] School of Computer Science, Wuhan University, Wuhan 430072, China
[*] Correspondence: wliu@whu.edu.cn (W.L.); jfshen@hgnu.edu.cn (J.S.); gaoge@whu.edu.cn (G.G.)

Abstract: In very large-scale integration circuit (VLSI) systems, microcontrollers are often implanted to manage the whole system to complete the given computing tasks. They play an essential part as regulators, which should allocate resources steadily and issue instructions promptly to drive functional units. However, most of the recent research focuses on the operation at the software level or the scheduling at the SoC level, ignoring the impact of the microarchitecture and the features of controlled sub-modules. This paper analyzes the requirements of microcontrollers in the VLSI system with various constraints and conditions that should be considered in the hardware implementation of such microarchitecture. Furthermore, this paper takes an open-source design using RISC-V ISA as the prototype to implement hardware microarchitecture. This design integrates the techniques of out-of-order processing, which are usually used on superscalar processors. As a result, the design quadruples the number of pipelined instructions, greatly alleviating the stalling of the instruction stream with a maximum extra look up table utilization of 18.37% in FPGA implementation.

Keywords: microcontroller; out-of-order; microarchitecture; RISC-V; hardware implementation

Citation: Hu, Y.; Chen, J.; Zhu, K.; Xing, Q.; Liu, W.; Shen, J.; Gao, G. Specially-Designed Out-of-Order Processor Architecture for Microcontrollers. *Electronics* **2022**, *11*, 2989. https://doi.org/10.3390/electronics11192989

Academic Editor: Juan M. Corchado

Received: 22 August 2022
Accepted: 15 September 2022
Published: 21 September 2022

1. Introduction

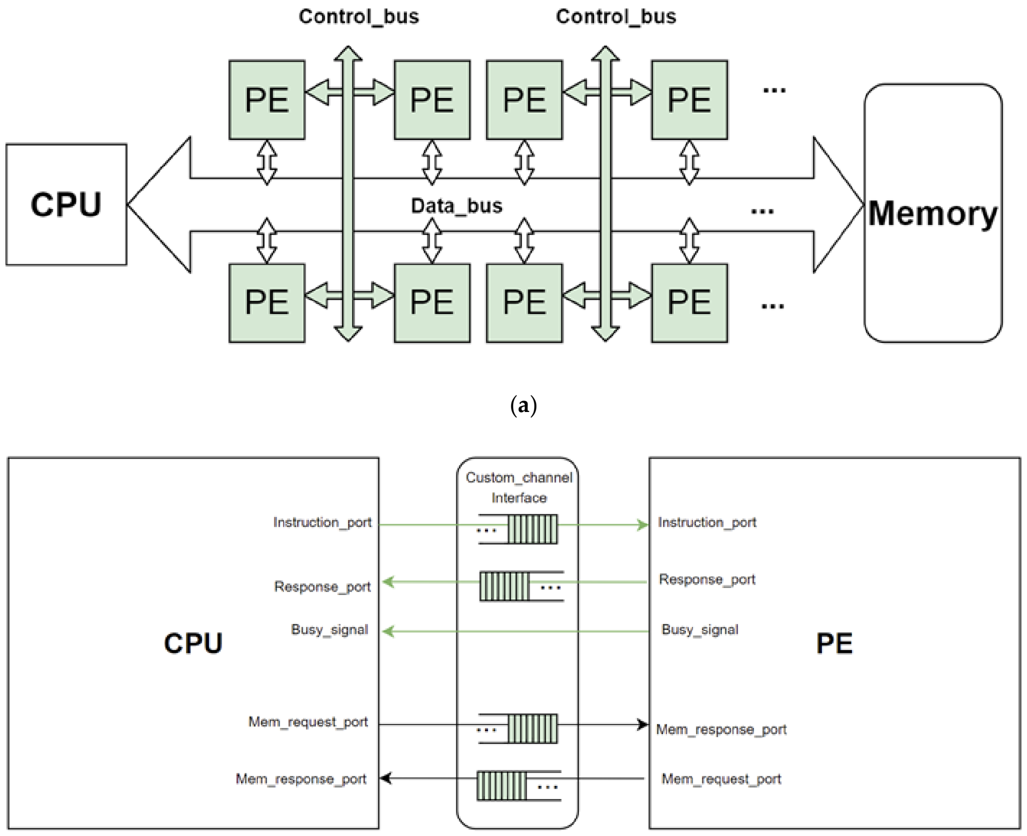
Generally, the microcontroller performs its role independently. It is embedded to make serial communication or some simple calculating tasks, and meets the processing requirements as well. However, as the requirements of machine learning tasks gradually increase, the microcontroller itself cannot perform the tasks of large-scale parallel computing. For example, the microcontroller plays a role in managing multiple coprocessors to better complete the computing tasks in the AI processor [1–3]. The current research is mostly limited to the scheduling at the software level and the SoC level [4–6], lacking specific analysis of the micro-architecture inside the controller. The study discusses the design of such a controller microarchitecture, starting with how the microcontroller can better perform in a VLSI system.

The function of the microcontroller in a VLSI system can be mainly divided into two parts: control and computing. As for the function of control, a microcontroller is required to be able to schedule other processing units to work correctly, such as configuring the registers of sub-modules, scheduling the data in memory, accepting interrupt requests and processing interrupt transactions. In addition, the microcontroller inevitably needs to perform some computing tasks in some cases. For example, in an AI processor, most of the computations are performed by sub-modules in high-speed parallel processing. When some operators adopt unusual data formats, or the computing mode is so complex, it is difficult for the sub-modules to handle.

There are two main methods to control, which are shown in Figure 1a,b:

- Sub-modules work through register configuration: reading and writing registers make up the major means of sub-module controlling. For example, NVDLA, an open-source deep learning accelerator from NVIDIA Corporation, works in this way [7]. The controller configures registers in a certain order, and then the sub-module starts up after the operation. This method is simple, but has high hardware portability since each sub-module is linked directly through a standard bus.

- Issuing instructions through the custom instruction interface: in modern microarchitecture, there will be a custom instruction interface for specific requirements. Processor designers formulate custom instructions according to the computing characteristics. In a general controlling flow, after a custom instruction is recognized in the processor dispatch unit, it will be transmitted to the sub-module through the custom interface directly. In return, the sub-module sends results back. In this method, the coupling between the processor and the sub-module is tighter. Therefore, it is necessary to consider the computing characteristics of the processor and the sub-modules, as well as their interaction. The custom instruction interfaces are the key to realizing such control, among which the most mature and popular is the ROCC interface in the Rocket architecture [8].



(**a**)

(**b**)

Figure 1. (**a**) The CPU controls the sub-module by reading and writing registers through the data bus and the control bus. (**b**) The CPU controls the sub-module by transmitting information through the custom instruction interface.

RISC-V is an open-source instruction set, which was proposed by the *University of California, Berkeley* in 2010 [9,10]. The original purpose of its design was to achieve Turing completeness with the fewest instructions and to implement unique operations like floating-point and vector computations in a modular and extensible way. RISC-V absorbs the development experience of x86 [11], ARM [12] and other advanced architectures [13]. To prevent the instruction set from becoming more complicated because of commercialization, modular architecture is used in RISC-V innovatively. Therefore, RISC-V can apply a unified architecture to modularize different parts of its instruction set to meet different applications.

This feature is particularly suitable for microcontroller designs in VLSI systems, so the following research will be based on RISC-V.

2. Contributions

This paper makes the following contributions:

- Detailed analysis of the microarchitecture requirements of microcontrollers in VLSI systems and the difficulties that microarchitecture hardware implementation needs to address.
- Based on the analysis of an open-source industrial-grade microcontroller design, this paper discusses the shortcomings, problems and improvement strategies of the existing microarchitecture.
- This paper proposes a hardware design of the microarchitecture for a functional and low power consumption microcontroller, which is implemented on the FPGA platform.

3. Challenges and Motivation

The problem that the microcontroller needs to solve to operate efficiently in a VLSI system focuses on the coordination of multi-cycle instructions and the correct response to abnormal conditions.

3.1. Analysis of Out-of-Order Processing

At present, the in-order design is mainly used in the microarchitecture design for microcontrollers, i.e., instructions are sequentially entered into the processor in the order of the assembly instructions compiled by the software. The microarchitecture of the in-order design brings lower power, less hardware complexity and higher operating frequency. However, the microarchitecture of in-order execution cannot adapt to the operation of a large number of multi-cycle instructions. Since the cycles required for the multi-cycle instruction's execution to obtain the result are unknown, subsequent instructions must be stalled due to in-order constraints, and can only be executed after the previous multi-cycle instruction has finished execution.

In the microarchitecture design of superscalar processors, the out-of-order technique is often used to solve this problem [14,15]. In the absence of data hazard, multi-cycle instructions will not stall the instruction stream, so subsequent instructions need not wait for the early execution. However, the design of this kind of microarchitecture for out-of-order processing is often too expensive to be directly applied to the microcontroller aiming at lower power and less complexity. Moreover, many of the related technologies of out-of-order processing are general concepts, not only applicable to superscalar processors.

We draw lessons from these ideas, through some unique design techniques to achieve efficient work of in-order processors. In the current research, A. Hilton [16] applies the technique of out-of-order processing to the in-order processor, and the design of "Flea-Flicker" two-pass pipelining [17] combines the out-of-order processing technologies into the in-order processor, which effectively avoids the stalling during the operation of the instructions. The advanced performance can be achieved by integrating the ideas of out-of-order processing into the microarchitecture of in-order processing with appropriate software scheduling.

According to McFarlin [18]'s research, the ability of software scheduling is still limited. Some scheduling must be realized through hardware out-of-order design. Because memory-access latency, branch and other multi-cycle instruction execution information are not transparent to the software, they can only be known during the actual execution of the instruction stream. Therefore, the research aims to further integrate the idea of out-of-order processing into the microarchitecture design of microcontrollers, and find the balance between out-of-order processing design and low-power design.

The workflow of the microcontroller includes fetching instructions and then operating the registers. Additionally, it controls the external sub-modules through memory-access instructions or custom instructions. For the RISC-V instruction set, only two memory-access instructions (load and store) can interact externally, i.e., the microcontroller based on the RISC-V instruction set mainly controls the sub-modules through these two memory-access instructions [9]. Therefore, in the design of an out-of-order processor, additional attention should be paid to an efficient stream of the memory-access instructions.

3.2. Analysis of Data Hazards

Although out-of-order processing can improve the operation of the instruction stream in the processor, it brings the problem of data hazards and requires additional hardware logic to deal with. The data hazards can be divided into three types:

- WAW (Write After Write): If the destination register index that the subsequent instruction needs to write back is the same as that of the previous one, then in the process of out-of-order execution, the subsequent instruction may write back before the previous one. The result of the previous instruction will overwrite the written back result of the subsequent instruction.
- WAR (Write After Read): If the destination register index that the subsequent instruction needs to write back is the same as that which the previous instruction needs to read, then in the process of out-of-order execution, the subsequent instruction may write back before the previous one reads the register, at which point the previous instruction reads the wrong data of source operands.
- RAW (Read After Write): If the source register index that the subsequent instruction needs to read is the same as that which the previous instruction needs to write back, then in the process of out-of-order execution, the subsequent instruction may read the register before the previous one writes back, at which point the subsequent instruction reads the wrong data of source operands.

3.3. Analysis of Static and Dynamic Hardware Scheduling in Abnormal Cases

The working environment for microcontrollers in VLSI system is diverse and complex. To keep the processor in a stable working state, there must be some additional scheduling designs in the hardware. There are two main types of scheduling at the hardware level: static scheduling and dynamic scheduling. For instance, the static scheduling strategy for branch instructions tends to be forward branches [19]. Since most instructions are forward branches, static scheduling can meet most of the prediction requirements. To further improve the prediction efficiency, a dynamic prediction strategy must be used at the cost of complicated logic and extra power consumption [20]. However, in the microarchitecture design of the microcontroller, its power consumption and complexity are often limited. To sum up, static prediction designs are usually adopted in the scheduling strategy. But in some cases, some dynamic scheduling designs can achieve very pleasant results without major side effects.

In the hardware design of microcontrollers, a suitable scheduling strategy can coordinate most of the operating states. But the scheduling strategy cannot resolve all the possible exceptions. To ensure the correctness of the processor, it is necessary to have a relevant hardware design to deal with abnormal situations.

4. Architecture and Analysis

The discussion will be based on the *Nuclei Technology Corporation*'s Hummingbird e203, which is an open-source, low-power and industrial-grade microcontroller [21]. Detailed processor core microarchitecture is shown in Figure 2. It reflects the microarchitecture design considerations of existing industrial-grade microcontrollers, while exposing many problems that can be optimized.

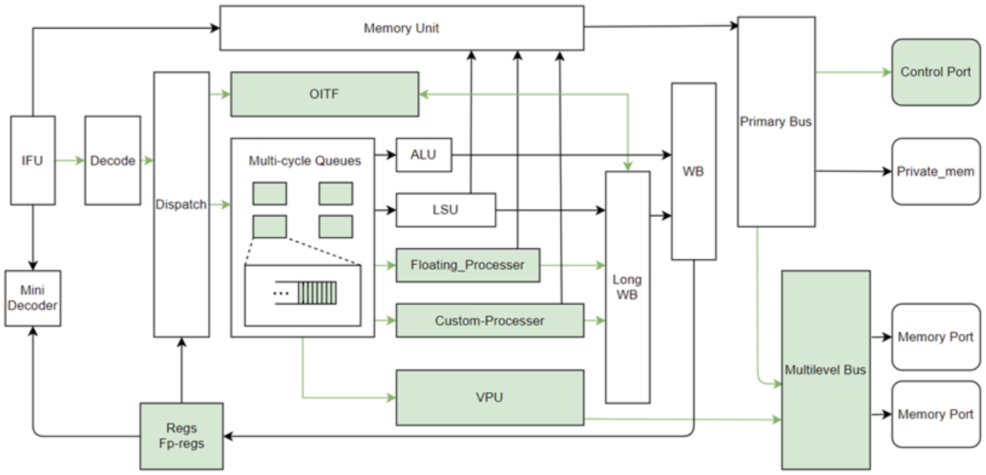


Figure 2. The overall hardware architecture diagram. Additional designs are marked in green color.

4.1. Pipeline of Original Microarchitecture

It can be said that the Hummingbird e203 adopts a two-stage pipeline structure. The front-end unit (instruction-fetch) fetches instructions in order, and puts the fetched instructions into the instruction registers. Every cycle, the back-end will dispatch the instructions from the instruction registers to different execution units in order.

The instructions are divided into two types:

- Instructions executed in one cycle. Such instructions are executed, committed and written back in one cycle on the second stage of the pipeline.
- Instructions executed in multiple cycles. When multi-cycle instructions are dispatched to the execution units and have not yet been written back, they are typically known as Outstanding Instructions.

Since the e203 processor writes back in order, there is a module called Outstanding Instruction Track FIFO (OITF). Every time a multi-cycle instruction is dispatched, an entry is registered in the OITF module, and the entry information is sent to the multi-cycle instruction controller at the same time. In the multi-cycle instructions write-back stage, the controller's entry information will be compared with the one in the OITF. If matched, it will be deregistered and written back, to ensure dispatch and write back in order.

In conclusion, the e203 processor adopts a microarchitecture of in-order issuing, out-of-order execution and in-order write-back. By the way, we add the floating-point processing unit and the vector processing unit to the original CPU design to meet some special needs.

4.2. Order of the Instructions Matters

The order of multi-cycle instruction (MI) and single-cycle instruction (SI) has a large impact on the data hazards. Two cases according to different orders are discussed:

- MI/SI after SI. Since the previous instruction completes all operations in one cycle, the subsequent instruction does not produce data hazards with the previous one.
- MI/SI after MI. Since the previous instruction requires multiple cycles to be executed, when the subsequent instruction operates on the register file, likely the previous one has not been written back. This will result in a data hazard. We call these MAM and SAM for short.

4.3. Stalling in Pipeline

The original processor design adopted a simple stalling approach to deal with the data hazards. Stalling in pipeline possibly occurs in both the instruction dispatch unit and instruction writeback unit:

1. When the instruction dispatch unit dispatches a multi-cycle instruction, it compares the source operand register index and result register index of the instruction with

every entry in the OITF. If the data hazard gets checked, the dispatch stage will be stalled.

2. When the instruction write-back unit writes back a multi-cycle instruction, it cannot be deregistered if the information in the multi-cycle instruction controller is different from the entry information of the OITF, thus, stalling the write-back stage.

The disadvantage of such a design is that it will stall the pipeline and prevent subsequent instructions operation frequently when operating numbers of multi-cycle instructions, e.g., when the controller configures the registers of the sub-module by the store instructions, and, due to stalling, the store instructions can only be issued one by one (as shown in Figure 3).

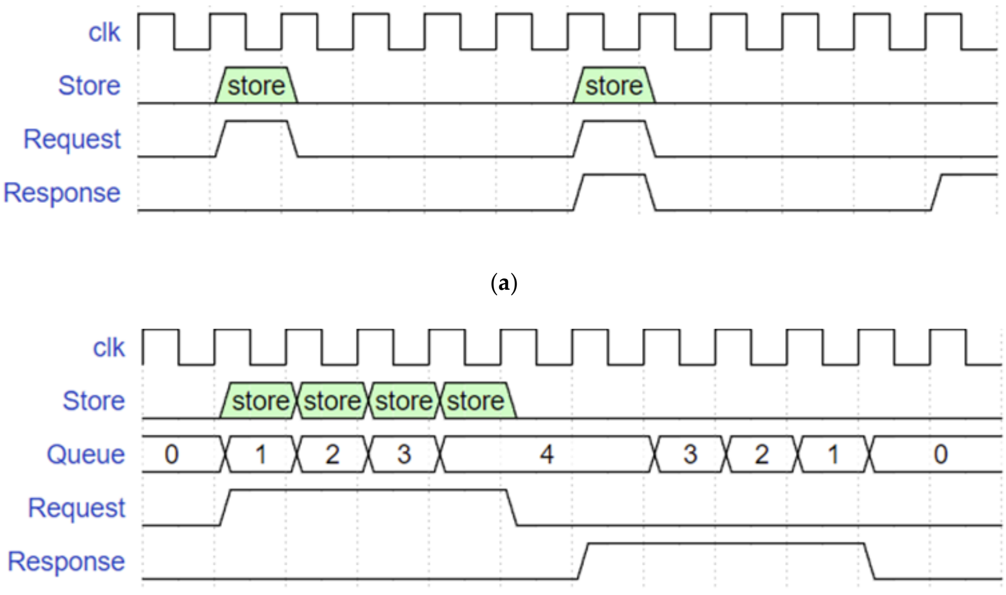


Figure 3. (**a**) Assuming that the store instruction takes five cycles to respond, they can only be executed one by one because of stalling. (**b**) Added store-queue, multiple store instructions can be executed consecutively without stalling.

4.4. Differences Caused by Different Multi-Cycle Instruction Dispatch Modes

There are many kinds of multi-cycle instructions, e.g., load & store instructions, multiply and divide instructions, vector instructions and floating-point instructions.

The dispatching methods of multi-cycle instructions may be completely different, and they can be divided into two types:

1. Stalled Multi-cycle Instruction Unit: the execution unit can no longer accept instructions when operating one instruction. The next instruction can only be accepted after the current operation is completed, e.g., a divider that continuously loops and calculates the result by using a serial technique.
2. Pipelined Multi-cycle Instruction Unit: the execution unit has a pipeline structure. It can execute multiple instructions at the same time, expanding the throughput of instructions by the technique of the pipeline. Therefore, multi-cycle instructions can be continuously dispatched to the execution unit in the pipeline style unless the pipeline is full or stalling.

According to the different dispatching methods of execution units, targeted design is required to achieve better parallel operation of multi-cycle instructions.

5. Implementation

To verify the above analysis and discussion, we propose a specific hardware design in terms of multi-cycle instruction scheduling, which enables microcontrollers to achieve higher performance in the control and computing tasks in the VLSI system. Next, we'll discuss the details of the hardware implementation.

5.1. Designs for Instruction Flow

Aiming at the hardware design requirements of microcontrollers in VLSI system, this paper analyzes the existing industrial microcontroller design in Section 4. Due to the stalling problem, the existing microcontroller designs fail to perform control and computation tasks smoothly and efficiently when handling multiple multi-cycle instructions and controlling a large number of sub-modules. The microarchitecture is designed to avoid the stalling of the instruction issue queue, the stalling of the instruction write-back register and the latency of the memory-access instructions.

The stalling problem in the instruction issue queue is mainly due to the data hazards of RAW and WAW, i.e., while the register data that the previous instruction needs to write has not been written yet, the subsequent instruction that has related data requirements needs to be stalled. Since the operand has been fetched from the register file before issuing due to the microarchitecture of the processor, which is a two-stage pipeline, the data hazard of WAR can be ignored. Such stalling is necessary to prevent incorrect processing results.

However, if there is no data hazard in the instruction queue after the stalled instruction at this time, it can bypass the stalled instruction and perform processing operations in advance. A waiting queue is designed to alleviate the stalling of issuing instructions. As shown in Figure 4a, if the issue queue is stalled, the stalled instructions will be rerouted to the waiting queue first. When the problem of the data hazard is solved, waiting queue instructions can be fired and executed normally. The hardware design proposes a clever solution without the side effect of high consumption. Abella Ferrer [22] researches and summarizes similar hardware designs for optimizing instruction issue queues.

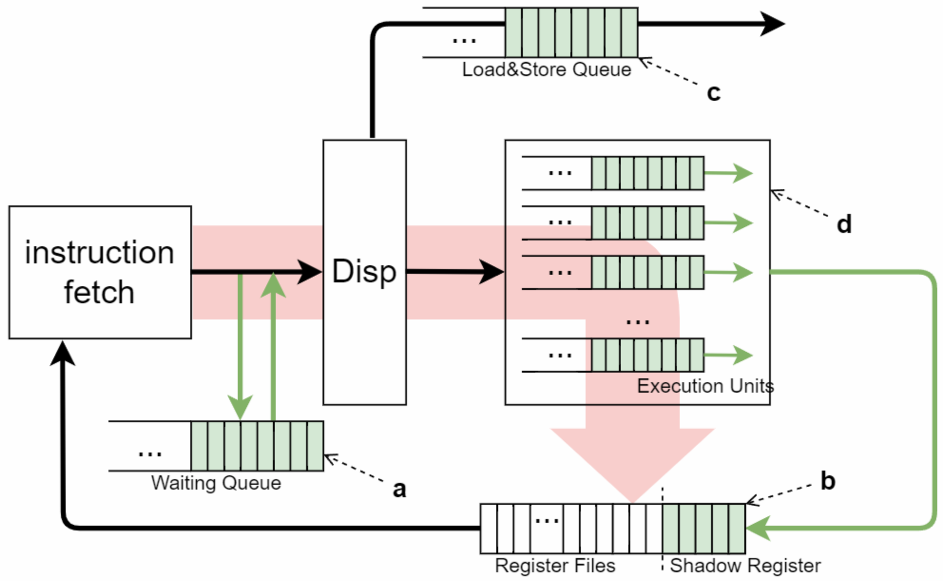


Figure 4. Schematic diagram of instruction flow. The green color marks the additionally added path for instructions, while the red color marks the main path. (**a**) Waiting queue. (**b**) Shadow register. (**c**) Load-Store queue. (**d**) Execution queue.

In addition, we increase the lanes of instruction flow by adding physical registers, which further solves the risk of the WAW data hazard. In the RISC-V instruction set specification, the processor has only 32 standard registers [9], which are called logic registers. Due to the limitation of registers' amount, the WAW data hazard occurs when two instructions write to the same register successively. In essence, this data hazard is a completely avoidable false data hazard, because there is no data exchange between the two. The technique of adding physical registers is applied to deal with it. Such technique is very common in the microarchitecture design of superscalar processors [23,24], but the consumption is too large for microcontrollers targeting lower power areas.

Therefore, we make a small trade-off. According to the analysis in Section 4, we only try to solve the SAM problem, while the MAM problem can be left to the waiting queue to handle. OITF has to judge every multi-cycle instruction to solve the MAM problem, which takes N $*$ (N-1) times (as shown in Figure 5a), while the SAM problem takes only N times (as shown in Figure 5b). Leaving the MAM problem in the waiting queue can minimize the logic complexity from O(N^2) to O(N).

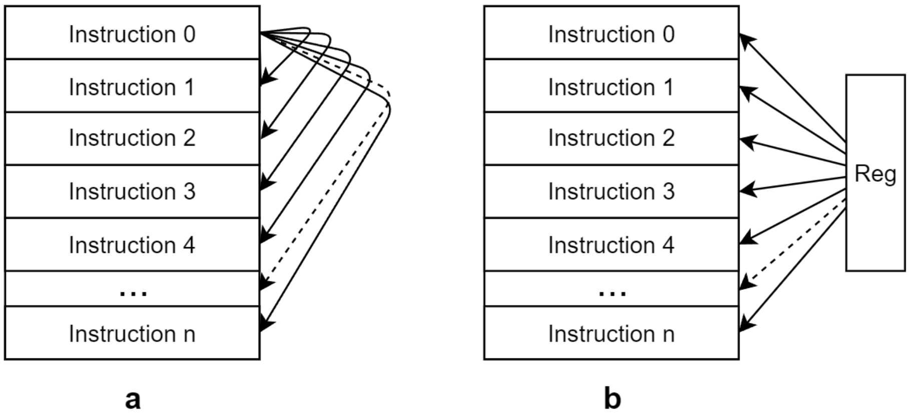


Figure 5. (**a**) Each multi-cycle instruction in an entry of depth N is compared with each other. (**b**) The instruction outside the entry only needs to be compared N times.

In Figure 4b, in addition to the original 32 physical registers and logical registers corresponding one by one, we call the newly added registers shadow register. Once a WAW occurs, the subsequent instructions are written back to the shadow register first. Data is updated from the shadow register to the logical register when the data hazard situation is resolved. If the shadow register is in the full state, the data hazard caused by the new instructions will be passed forward to the issue waiting queue for solution. Moreover, if the issue waiting queue is also in the full state, the pipeline will be stalled until the idle state occurs and then the pipeline will continue.

5.2. Design of the Multi-Cycle Instruction Queues

Control and part of the computation function are the indispensable functions of the microcontroller. Since the processor is based on the RISC-V ISA, this paper takes full advantage of its modular features, and implements the configurable characteristics of the hardware design. For example, in the AI processor, part of the algorithm cannot be optimized by the compiler (delivered to the domain-specific accelerator to compute), so the microcontroller needs to add support for floating-point operation instructions or vector operation instructions to meet this part of the demand.

As discussed in Section 4, different multi-cycle instructions have different dispatch modes, so the separate queues are designed according to the characteristics of each multi-cycle instruction. In Figure 4c, we have specially designed a memory-access queue for the instructions to meet a large number of memory-access requirements. In Figure 4d, we can effectively adapt to different dispatch modes of multi-cycle instructions by configuring the depth of instruction queues. For example, we can configure the depth of the memory-access queue according to the number of sub-modules to be controlled, or configure a specific instruction queue depth based on the dispatch mode of the multi-cycle instruction execution unit. These specially-designed instruction queues enhance the flexibility and extensibility of the microcontroller.

5.3. Design of System Architecture

Additional modules and complex buses make the simple system architecture unable to meet the new requirements. Therefore, we also design the system architecture and the access interface with ingenuity (as shown in Figure 6). In a VLSI system, due to the large number of modules to be scheduled, the bus is often very complex, and then the biggest problem is that it will bring additional memory access latency.

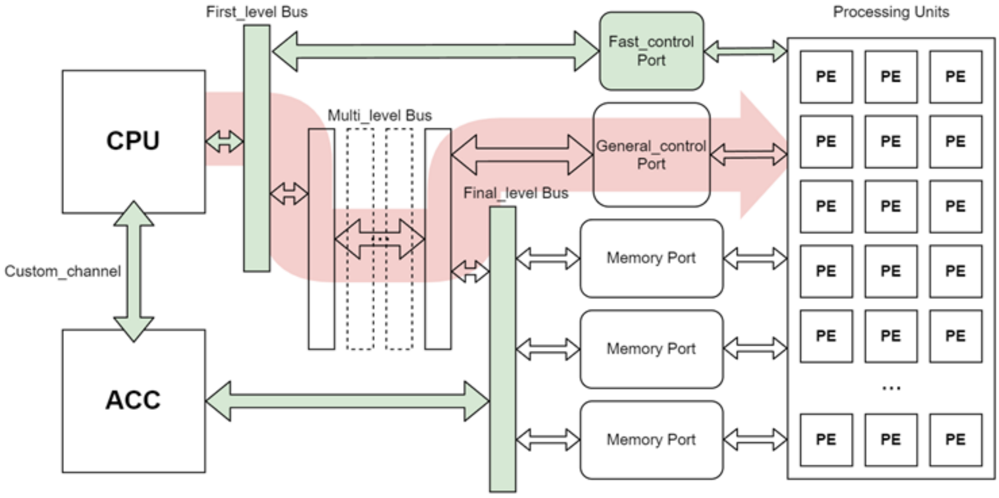


Figure 6. Schematic diagram of system architecture.

The design is mainly reflected in the following two parts:

1. The fast access interface that directly bypasses the bus is additionally developed. This fast access interface is used for some sub-modules requiring high responsiveness control, while the general access interface needs to step across the multi-level bus.
2. Separate access interface for vector processor. Due to the huge memory demand of the vector processor, a large number of data streams may affect the execution of the basic memory instructions. Therefore, as shown in Figure 6, we decouple the access interface of the vector processor from that of the microcontroller, so the vector computing data streams are separated from the control streams.

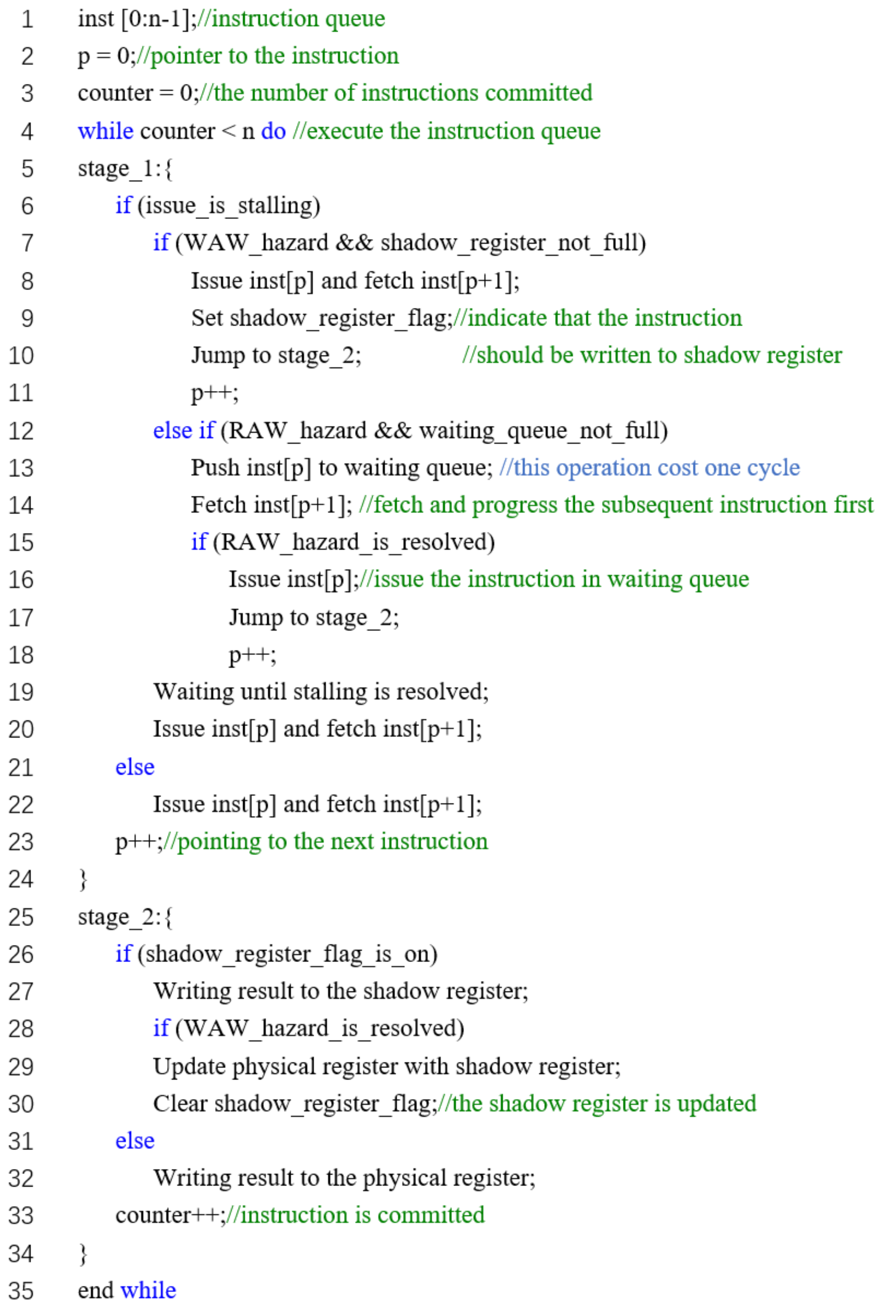
6. Results

6.1. Function Analysis

The impact on the operation of pipeline is shown in the pseudocode in Figure 7. Without the waiting queue and the shadow registers, issuing instructions and operating the registers may result in stalling. Causes for the stalling could include the stalling correlation being a WAW hazard or RAW hazard or a full multi-cycle instruction queue (discussed in Section 5.2). The stalling will prevent the subsequent instructions from issuing or being fetched, delaying the entire pipeline for several, or even dozens, of clock cycles.

To address the issue of stalling, the waiting queue and the shadow registers are implemented in the design. They can avoid pipeline stalling in most scenarios, costing not more than one clock cycle for each instruction.

For example, when the WAW hazard occurs, the data will be written into the shadow registers rather than the physical registers. Though the previous instructions have not returned the result, the pipeline can move on. The data stored in the shadow registers will be written into the physical registers after the WAW hazard is resolved. The shadow registers eliminate the WAW data dependencies directly to avoid stalling. Therefore, it does not take any extra clock cycle. The shortcoming is that it can only handle the stalling caused by the WAW hazard.



```
1   inst [0:n-1];//instruction queue
2   p = 0;//pointer to the instruction
3   counter = 0;//the number of instructions committed
4   while counter < n do //execute the instruction queue
5   stage_1:{
6       if (issue_is_stalling)
7           if (WAW_hazard && shadow_register_not_full)
8               Issue inst[p] and fetch inst[p+1];
9               Set shadow_register_flag;//indicate that the instruction
10              Jump to stage_2;            //should be written to shadow register
11              p++;
12          else if (RAW_hazard && waiting_queue_not_full)
13              Push inst[p] to waiting queue; //this operation cost one cycle
14              Fetch inst[p+1]; //fetch and progress the subsequent instruction first
15              if (RAW_hazard_is_resolved)
16                  Issue inst[p];//issue the instruction in waiting queue
17                  Jump to stage_2;
18                  p++;
19          Waiting until stalling is resolved;
20          Issue inst[p] and fetch inst[p+1];
21      else
22          Issue inst[p] and fetch inst[p+1];
23      p++;//pointing to the next instruction
24  }
25  stage_2:{
26      if (shadow_register_flag_is_on)
27          Writing result to the shadow register;
28          if (WAW_hazard_is_resolved)
29          Update physical register with shadow register;
30          Clear shadow_register_flag;//the shadow register is updated
31      else
32          Writing result to the physical register;
33      counter++;//instruction is committed
34  }
35  end while
```

Figure 7. Pseudocode of the operation of pipeline.

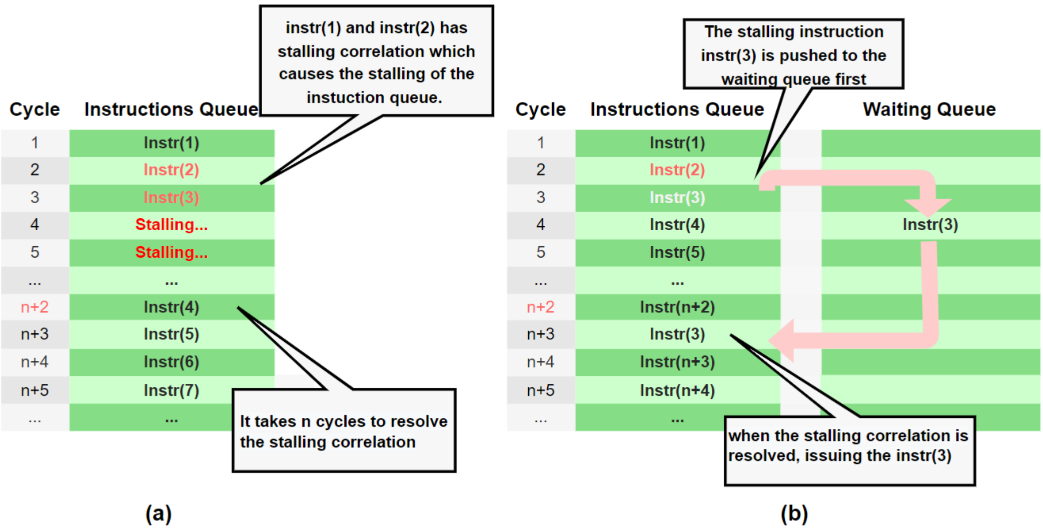
Unlike the shadow registers, the waiting queue handles the stalling by changing the order of the issuing instructions. When a stalling correlation between instructions occurs, the current instruction will be stored in the waiting queue, waiting for the stalling to be resolved. In this process, it takes one cycle to store the instruction in the waiting queue and takes another cycle to fetch it (as shown in the pseudocode on line 14). Thus, it takes one more clock cycle to operate each waiting instruction in the pipeline.

By contrast, the waiting queue can work in any stalling situation, while the shadow registers can only avoid the stalling caused by WAW. The waiting queue takes one cycle to address one stalling issue, while the shadow registers do not cost extra cycles. The contrast of both designs shows that WAW hazards should be solved by shadow registers. Only if the shadow registers are full should the waiting queue work. The shadow registers play the roles of assistant and pioneer. This strategy reduces the waste of the clock cycle and prevents the waiting queue from being too busy to handle other stalling.

6.2. Performance

The performance of the hardware microarchitecture is greatly affected by the actual operating environment, e.g., different implementations of the algorithm, compiler optimization and actual memory access latency all make a difference. To evaluate the performance of the hardware optimization, the theoretical and practical aspects are analyzed and evaluated.

The case shown in Figure 8 assumes that the stalling correlation occurs between instr(3) and instr(2). Instr(2) needs N cycles to return the result. In the common case, the pipeline will stall for N cycles. Under the action of the waiting queue, the order of instructions is changed. The subsequent instructions will be issued continuously. Instr(3) enters the waiting queue and is issued and executed again after instr(2) returns the result. In this case, the pipeline stalls for only one cycle, saving N-1 clock cycles.



Cycle	Instructions Queue
1	Instr(1)
2	Instr(2)
3	Instr(3)
4	Stalling...
5	Stalling...
...	...
n+2	Instr(4)
n+3	Instr(5)
n+4	Instr(6)
n+5	Instr(7)
...	...

instr(1) and instr(2) has stalling correlation which causes the stalling of the instuction queue.

It takes n cycles to resolve the stalling correlation

(a)

Cycle	Instructions Queue	Waiting Queue
1	Instr(1)	
2	Instr(2)	
3	Instr(3)	
4	Instr(4)	Instr(3)
5	Instr(5)	
...	...	
n+2	Instr(n+2)	
n+3	Instr(3)	
n+4	Instr(n+3)	
n+5	Instr(n+4)	
...	...	

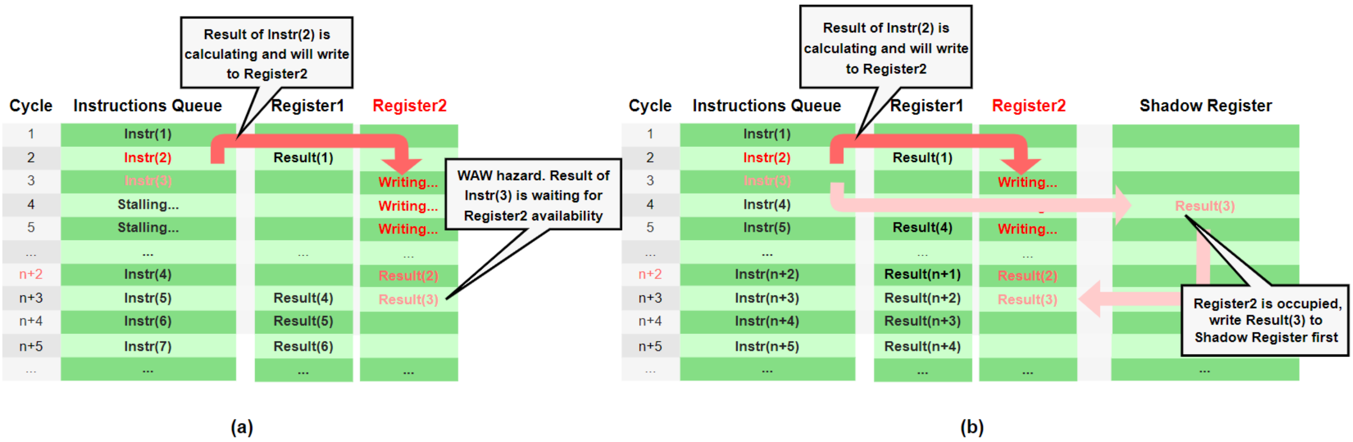
The stalling instruction instr(3) is pushed to the waiting queue first

when the stalling correlation is resolved, issuing the instr(3)

(b)

Figure 8. Theoretical analysis of the waiting queue. (**a**) Without waiting queue. (**b**) Optimized with waiting queue.

As shown in Figure 9, the WAW data hazard occurs between instr(3) and instr(2). The execution results of instr(2) and instr(3) both need to be written into register2, while result(2) needs N cycles to return. In the common case, the pipeline will stall for N cycles. Due to the shadow registers eliminating the WAW data dependencies, result(3) enters the shadow registers and will be written into register2 after the result(2) is written. In this case, the pipeline does not stall, saving N clock cycles.



Cycle	Instructions Queue	Register1	Register2
1	Instr(1)		
2	Instr(2)	Result(1)	
3	Instr(3)		Writing...
4	Stalling...		Writing...
5	Stalling...		Writing...
...	...	...	...
n+2	Instr(4)		Result(2)
n+3	Instr(5)	Result(4)	Result(3)
n+4	Instr(6)	Result(5)	
n+5	Instr(7)	Result(6)	
...	...	...	...

Result of Instr(2) is calculating and will write to Register2

WAW hazard. Result of Instr(3) is waiting for Register2 availability

(a)

Cycle	Instructions Queue	Register1	Register2	Shadow Register
1	Instr(1)			
2	Instr(2)	Result(1)		
3	Instr(3)		Writing...	
4	Instr(4)			Result(3)
5	Instr(5)	Result(4)	Writing...	
...	...	...	...	
n+2	Instr(n+2)	Result(n+1)	Result(2)	
n+3	Instr(n+3)	Result(n+2)	Result(3)	
n+4	Instr(n+4)	Result(n+3)		
n+5	Instr(n+5)	Result(n+4)		
...	...	...	...	...

Result of Instr(2) is calculating and will write to Register2

Register2 is occupied, write Result(3) to Shadow Register first

(b)

Figure 9. Theoretical analysis of the shadow register. (**a**) Without shadow register. (**b**) Optimized with shadow register.

To evaluate the theoretical maximum speedup, the processing time under different conditions is listed in Table 1. In a total of M instructions, we use m to denote the number of hazards and n to denote the average stalling cycles.

Table 1. Theoretical cycle consumption under different designs. Assuming that the number of instructions is M, the number of WAW hazard is m1, the WAW extra cycle cost is n1, the number of RAW is m2 and the RAW extra cycle cost is n2.

Hardware Design	Stalling Cycle	Total Cycles
Original	m1 ∗ n1 + m2 ∗ n2	M + m1 ∗ n1 + m2 ∗ n2
Shadow Register	m2 ∗ n2	M + m2 ∗ n2
Waiting Queue	m1 + m2	M + m1 + m2
Shadow Register + Waiting Queue	m2	M + m2

The theoretical speedup rate of the shadow registers is:

$$\text{SpeedUpRate} = \left(\frac{\text{M} + \text{m1}{*}\text{n1} + \text{m2}{*}\text{n2}}{\text{M} + \text{m2}{*}\text{n2}} - 1\right) \times 100\% \tag{1}$$

Compared to the original design, the stalling cycles of WAW are saved. Because m1 represents the number of WAW hazards, and n1 represents the average number of clock cycles consumed per WAW hazard, this means that the more cycles WAW hazards consume, the higher the speedup rate provided by the shadow registers in the system.

The theoretical speedup rate of the waiting queue is:

$$\text{SpeedUpRate} = \left(\frac{\text{M} + \text{m1}{*}\text{n1} + \text{m2}{*}\text{n2}}{\text{M} + \text{m1} + \text{m2}} - 1\right) \times 100\% \tag{2}$$

With the operation of waiting queue, the clock cycles consumed by a single hazard (n1 and n2) have almost no effect on the pipeline. The result depends on the total number of hazards. Therefore, in the case of few hazards (m1 and m2) with a large number of WAW cycle cost, the shadow registers perform better than the waiting queue.

The most theoretical speedup rate of the coordination between the waiting queue and shadow registers is:

$$\text{SpeedUpRate} = \left(\frac{\text{M} + \text{m1}{*}\text{n1} + \text{m2}{*}\text{n2}}{\text{M} + \text{m2}} - 1\right) \times 100\% \tag{3}$$

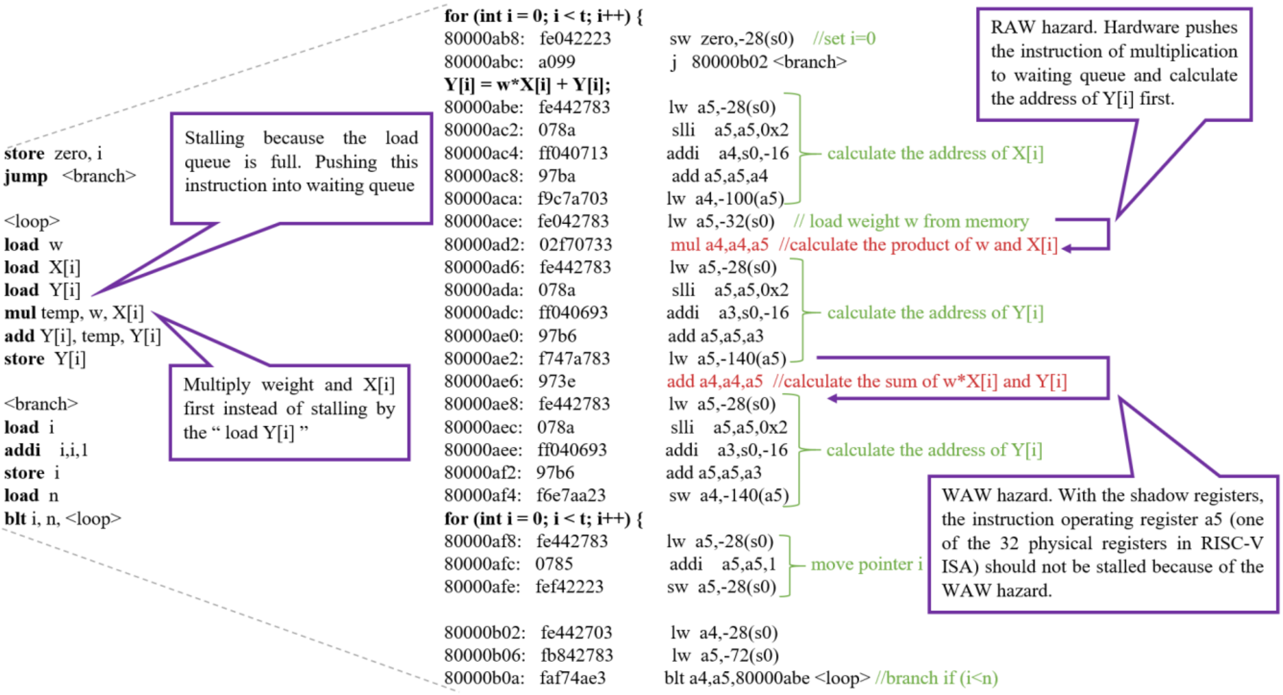
The above two designs work together to maximize optimization. As a result, the stalling cycles are reduced to m2, and the total cycles are reduced to only N + m2. Since in most cases m2 is much smaller than m1f ∗ n1 + m2 ∗ n2, the theoretical speedup rate can be (m1 ∗ n1 + n2)/N and the final actual speedup rate can be close to 20%, which is quite impressive for microcontrollers.

What is also worth noticing is that when m1 = 1, n1 = n and m2 = 0 (there is only 1 WAW hazard), the theoretical speedup rate is:

$$\text{SpeedUpRate} = \left(\frac{\text{M} + \text{n}}{M} - 1\right) \times 100\% = \frac{\text{n}}{M} \times 100\% \tag{4}$$

For instance, if N = 100, n = 10 (pipeline will stall for n cycles because of the hazard) and the speedup rate will be 10%.

The computation of SAXPY (Scalar Alpha X Plus Y) is one of the most important and frequent AI computing operators. From a practical point of view, the processing of SAXPY computation is analyzed to evaluate the actual performance of the hardware optimization. The assembly codes are shown in Figure 10.



```
store zero, i
jump  <branch>

<loop>
load  w
load  X[i]
load  Y[i]
mul temp, w, X[i]
add Y[i], temp, Y[i]
store  Y[i]

<branch>
load  i
addi   i,i,1
store  i
load  n
blt i, n, <loop>
```

Stalling because the load queue is full. Pushing this instruction into waiting queue

Multiply weight and X[i] first instead of stalling by the “ load Y[i] ”

```
for (int i = 0; i < t; i++) {
80000ab8:  fe042223        sw  zero,-28(s0)   //set i=0
80000abc:  a099            j  80000b02 <branch>
Y[i] = w*X[i] + Y[i];
80000abe:  fe442783        lw  a5,-28(s0)
80000ac2:  078a            slli   a5,a5,0x2
80000ac4:  ff040713        addi   a4,s0,-16      calculate the address of X[i]
80000ac8:  97ba            add a5,a5,a4
80000aca:  f9c7a703        lw  a4,-100(a5)
80000ace:  fe042783        lw  a5,-32(s0)   // load weight w from memory
80000ad2:  02f70733        mul a4,a4,a5  //calculate the product of w and X[i]
80000ad6:  fe442783        lw  a5,-28(s0)
80000ada:  078a            slli   a5,a5,0x2
80000adc:  ff040693        addi   a3,s0,-16      calculate the address of Y[i]
80000ae0:  97b6            add a5,a5,a3
80000ae2:  f747a783        lw  a5,-140(a5)
80000ae6:  973e            add a4,a4,a5  //calculate the sum of w*X[i] and Y[i]
80000ae8:  fe442783        lw  a5,-28(s0)
80000aec:  078a            slli   a5,a5,0x2
80000aee:  ff040693        addi   a3,s0,-16      calculate the address of Y[i]
80000af2:  97b6            add a5,a5,a3
80000af4:  f6e7aa23        sw  a4,-140(a5)
for (int i = 0; i < t; i++) {
80000af8:  fe442783        lw  a5,-28(s0)
80000afc:  0785            addi   a5,a5,1       move pointer i
80000afe:  fef42223        sw  a5,-28(s0)

80000b02:  fe442703        lw  a4,-28(s0)
80000b06:  fb842783        lw  a5,-72(s0)
80000b0a:  faf74ae3        blt a4,a5,80000abe <loop> //branch if (i<n)
```

RAW hazard. Hardware pushes the instruction of multiplication to waiting queue and calculate the address of Y[i] first.

WAW hazard. With the shadow registers, the instruction operating register a5 (one of the 32 physical registers in RISC-V ISA) should not be stalled because of the WAW hazard.

Figure 10. Assembly codes of SAXPY computation. Left, the handwritten assembly codes. Right, the assembly codes compiled by GCC compiler with optimized level of −O3 (the most).

In the handwritten assembly codes of the implementation of SAXPY, three instructions consecutively accessing memory causes stalling because the load queue is full. With the waiting queue, the stalling instruction is pushed to the waiting queue to issue later instead of stalling the operation of the multiply instruction, which has no stalling correlation. With the memory access latency of 4 cycles (that can operate two load instructions simultaneously, as discussed in Section 4.3), multiplication of 5 cycles and the addition of 1 cycle, the cycles consumed in the computation loop are optimized from 18 cycles to 16 cycles, which is 12.5% speedup.

The assembly codes produced by the compiler are enlarged because, to address the data for computing in arrays (X[i], Y[i]), the calculation of the address is added. In the optimization of the compiler, it reorders the order of the multiply instruction and memory access instruction, i.e., placing the multiplication before the memory load for Y[i]. As discussed in Section 3.1, the compiler plays a role in reordering the instructions, but without accurate information of hardware processing, the optimization is limited. As shown in Figure 10, the stalling is unavoidable because the computation of multiplication and addition must wait until the data is accessed from the memory. In addition, to optimize the use of registers (the number of physical registers is limited in RISC-V ISA), the compiler usually reuses the same register, i.e., register a5 in this piece of assembly codes, which causes a WAW data hazard. After optimizing the hardware microarchitecture, the cycles assumed in the computation loop are optimized from 40 cycles to 30 cycles, which is 33.3% speedup.

6.3. Hardware Utilization

The hardware design has been implemented in the *XILINX* XC7A200T-2FBG484I FPGA platform [25]. The results are shown in Table 2 below. The hardware resource consumption mainly lies in OITF (introduced in Section 4.1), recording the execution of multi-cycle instructions and catching data hazards. OITF is, essentially, a FIFO-structure module. The depth of OITF means the number of the entries it owns to record the multi-cycle instructions, which indicates the number of multi-cycle instructions that can be executed simultaneously.

Table 2. Extending the depth of OITF to 8, the table contains the experimental data in applying optimized designs to the original design. For example, when extending the instruction stream lanes to 12 with full functions, the consumption of look up table (LUT) is 4209, flip-flop (FF) is 2204 and multiplexers (MUX) is 332.

	Utilization			**Functions**			
	LUT	**FF**	**MUX**	**Lanes**	**Max-Pipelined Instructions**	**Write-Back Optimized**	**Issuing Optimized**
Original	3832	1832	289	2	2	×	×
Extended_OITF	3972	2076	328	2	8	×	×
Extended_OITF + Waiting Queue	4106	2134	332	3	8	√	×
Extended_OITF + Shadow Register	3997	2124	328	3	8	×	√
Extended_OITF + Shadow Register + Waiting Queue	4131	2162	332	4	8	√	√
Extended_OITF + Shadow Register + Waiting Queue + More Lanes	4209	2204	332	12	8	√	√

Additional data lanes have been added to optimize stalling by out-of-order processing of the instruction stream. However, to prevent data hazard caused by out-of-order processing, increased hardware logic complexity to judge the multi-cycle instructions recorded in OITF is essential. As shown in Figure 5b, each additional judgment will increase N logical constraints.

Extra supports for different multi-cycle instructions mean more pipelined instructions and more lanes for data streams. In addition, the demand of maximum pipelined instructions registered in OITF will naturally increase. Therefore, predictably, the probability of stalling inflates steeply as more and more operations are embedded for complex applications. The direct consequence is lower efficiency.

Two designs work to keep subsequent instructions flowing continuously: a waiting queue handling both WAW and RAW stalling problems, and shadow registers assisting to relieve the WAW stalling further. The former one takes all possible situations into consideration so that it has outstanding performance in alleviating stalling, while the latter one consumes only on average half of the LUT utilization compared to the former one (as shown in Figure 11), as a supplement for WAW.

Simultaneously, both a shadow register and waiting queue could develop one additional data lane for instructions with only 134 and 25 extra LUT consumption. If the two designs are both equipped as such, there will be four lanes for instructions (the result could be linear superposed) with only extra 159 LUT, 86 FF and 4 MUX consumption when the depth of OITF is eight.

According to the experiment, adding two extra concurrent operations in the execution unit (Figure 4d) will triple the number of data lanes. As a result, the data lanes increase by six times, and the max-pipelined instructions increase fourfold, while write-back and issuing stalling problems are both substantially optimized at the extra cost of 377 (+9.83%) LUT, 372(+20.31%) FF and 43(+14.88%) MUX.

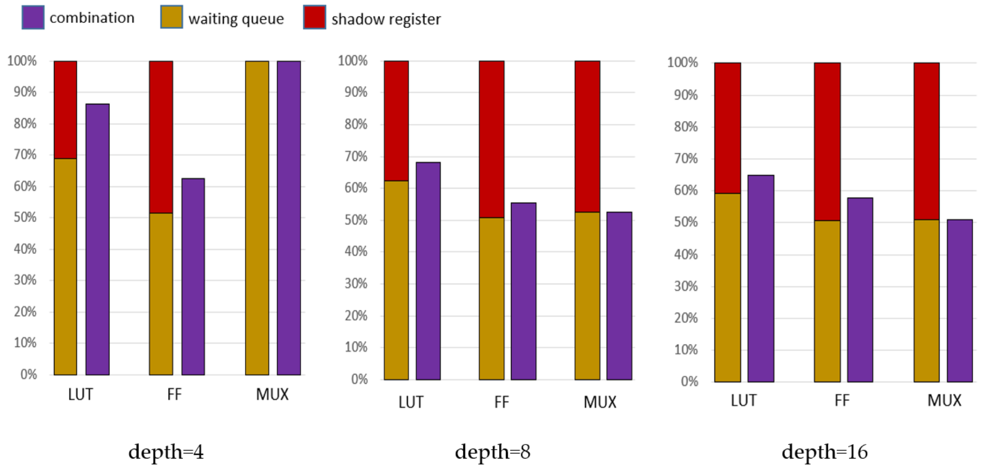


Figure 11. Comparison of hardware resource utilization of the optimizing designs and the combined design with different depths of OITF. Because of reused resources, the combined design consumes less than the individual optimizing designs of the two.

The hardware implementation results show the consumption caused by additional data lanes in issuing stage and write-back stage. The designs quadruple the number of pipelined instructions and increase the number of lanes by six times, greatly alleviating the stalling of the instruction stream without huge consumption of hardware resources.

As shown in Figure 12, the maximum number of pipelined multi-cycle instructions is the most significant cost to the hardware design. For example, if the depth of OITF is 2, the maximum extra LUT utilization is 4.02%, FF utilization is 3.82% and MUX utilization is 1.38%, while the depth of OITF is 16, the maximum extra LUT utilization is 18.37%, FF utilization is 44.65% and MUX utilization is 41.87% (with full function and extending lanes to 12). Therefore, this indicator has to be considered carefully in the hardware design tradeoff. In general scenarios, different multi-cycle instructions will not be executed in large numbers at the same time. To reduce hardware consumption, a maximum number of pipelined multi-cycle instructions does not need to satisfy the state that all multi-cycle instruction queues are fully loaded.

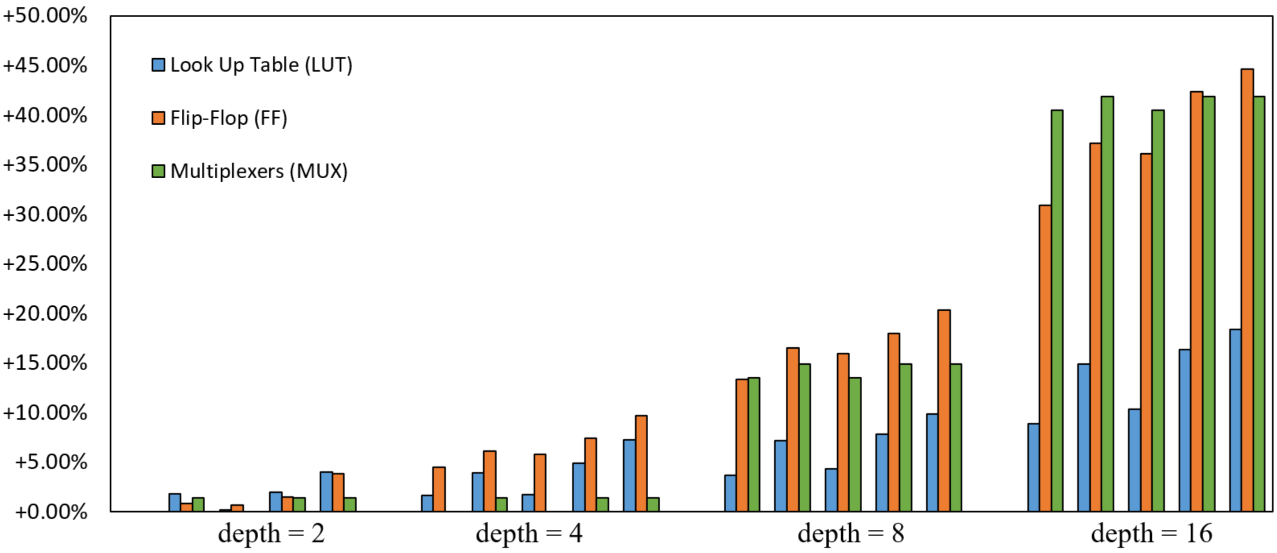


Figure 12. The percentage of increased utilization with different depths of OITF.

7. Conclusions

To adapt the requirements of AI computing, more and more sub-modules and extra instruction set extensions in microcontrollers are calling for improvement of the microarchitecture design. Out-of-order microarchitecture is worth considering to satisfy the demands.

Out-of-order processing techniques significantly boost the efficiency of the instruction stream's execution. However, side-effects include data hazards and the heavy consumption of logical resources. Therefore, an out-of-order microarchitecture for low-power implementation with slight consumption is significant and valuable.

According to the analysis of existing industrial-grade microcontrollers, the stalling problems during processing are exposed. An out-of-order processing microarchitecture is proposed to solve these problems without heavy consumption. The experimental results based on a RISC-V prototype to evaluate the consumption of hardware resources and the performance improvement in pipelines prove the feasibility.

This design quadruples the number of pipelined instructions and increases the data lanes by six times while greatly alleviating the stalling of the instruction stream, which makes it possible for microcontrollers to handle complex AI computing tasks as superscalar processors do.

8. Future Work

In the VLSI system, the primary mission of the microcontroller is to cooperate and coordinate the work of the sub-modules, and its performance highly depends on the characteristics of the sub-modules. Therefore, software tests like SPEC2017 for individual CPUs are not convincing because they only measure the performance of the processor individually and ignore the impact of the operation of sub-modules on the processor. Due to various situations, a complete test will be complicated, e.g., whether the compiler considers the processor's microarchitecture information when optimizing the code, whether the processing characteristics of sub-modules and processing units are met during the test, etc. Further research is needed to explore a dependable benchmark considering the impact of the AI algorithm, compiler optimization and the compute pattern of the different sub-modules.

Author Contributions: Conceptualization, Y.H. and W.L.; methodology, Y.H.; software, Y.H., J.C. and K.Z.; validation, Y.H., J.C. and K.Z.; formal analysis, Y.H.; investigation, Y.H.; resources, W.L.; data curation, Y.H., J.C., Q.X. and K.Z.; writing—original draft preparation, Y.H.; writing—review and editing, Y.H., J.C., Q.X., K.Z. and W.L.; visualization, Y.H.; supervision, Y.H. and W.L.; project administration, W.L.; funding acquisition, W.L., J.S. and G.G. All authors have read and agreed to the published version of the manuscript.

Funding: This work was supported by the Special Fund of Hubei Luojia Laboratory (Grant No. 220100025), the Key Project of Hubei Province (Grant No. 2021BAA179), the Key Projects of Science and Technology to Help the Economy 2020 (Grant No. SQ2020YFF0426493) and the Frontier Projects of Applied Foundation in Wuhan (2019010701011386).

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Vasiljevic, J.; Bajic, L.; Capalija, D.; Sokorac, S.; Ignjatovic, D.; Bajic, L.; Trajkovic, M.; Hamer, I.; Matosevic, I.; Cejkov, A. Compute substrate for Software 2.0. *IEEE Micro* **2021**, *41*, 50–55. [CrossRef]
2. Fleischer, B.; Shukla, S.; Ziegler, M.; Silberman, J.; Oh, J.; Srinivasan, V.; Choi, J.; Mueller, S.; Agrawal, A.; Babinsky, T. A scalable multi-TeraOPS deep learning processor core for AI trainina and inference. In Proceedings of the 2018 IEEE Symposium on VLSI Circuits, Honolulu, HI, USA, 18–22 June 2018; pp. 35–36.
3. Fowers, J.; Ovtcharov, K.; Papamichael, M.; Massengill, T.; Liu, M.; Lo, D.; Alkalay, S.; Haselman, M.; Adams, L.; Ghandi, M. A configurable cloud-scale DNN processor for real-time AI. In Proceedings of the 2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA), Los Angeles, CA, USA, 1–6 June 2018; pp. 1–14.
4. Saha, S.S.; Sandha, S.S.; Srivastava, M. Machine Learning for Microcontroller-Class Hardware—A Review. *arXiv* **2022**, arXiv:2205.14550.

5. Parai, M.K.; Das, B.; Das, G. An overview of microcontroller unit: From proper selection to specific application. *Int. J. Soft Comput. Eng. IJSCE* **2013**, *2*, 228–231.
6. Babiuch, M.; Foltýnek, P.; Smutný, P. Using the ESP32 microcontroller for data processing. In Proceedings of the 2019 20th International Carpathian Control Conference (ICCC), Kraków, Poland, 26–29 May 2019; pp. 1–6.
7. Corporation, NVIDIA. NVDLA Open Source Hardware, Version 1.0. Available online: https://github.com/nvdla/hw (accessed on 8 July 2022).
8. Asanovic, K.; Avizienis, R.; Bachrach, J.; Beamer, S.; Biancolin, D.; Celio, C.; Cook, H.; Dabbelt, D.; Hauser, J.; Izraelevitz, A. *The Rocket Chip Generator*; Technical Report UCB/EECS-2016-17; EECS Department, University of California: Berkeley, CA, USA, 2016; p. 4.
9. Waterman, A.; Asanović, K. The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 2019121. Available online: https://riscv.org/wp-content/uploads/2019/12/riscv-spec-20191213.pdf (accessed on 8 July 2022).
10. Asanović, K.; Patterson, D.A. *Instruction Sets Should Be Free: The Case for risc-v*; Technical Report UCB/EECS-2014-146; EECS Department, University of California: Berkeley, CA, USA, 2014.
11. Blem, E.; Menon, J.; Sankaralingam, K. Power struggles: Revisiting the RISC vs. CISC debate on contemporary ARM and x86 architectures. In Proceedings of the 2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA), Washington, DC, USA, 23–27 February 2013; pp. 1–12.
12. Blem, E.; Menon, J.; Sankaralingam, K. A Detailed Analysis of Contemporary Arm and x86 Architectures. UW-Madison Technical Report. 2013. Available online: https://caxapa.ru/thumbs/788118/10.1.1.364.1145.pdf (accessed on 8 July 2022).
13. Liu, S.; Du, Z.; Tao, J.; Han, D.; Luo, T.; Xie, Y.; Chen, Y.; Chen, T. Cambricon: An instruction set architecture for neural networks. In Proceedings of the 2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA), Seoul, Korea, 18–22 June 2016; pp. 393–405.
14. Celio, C.P. *A Highly Productive Implementation of an Out-of-Order Processor Generator*; University of California: Berkeley, CA, USA, 2017.
15. Palacharla, S.; Jouppi, N.P.; Smith, J.E. Complexity-effective superscalar processors. In Proceedings of the 24th Annual International Symposium on Computer Architecture, Denver, CO, USA, 2–4 June 1997; pp. 206–218.
16. Hilton, A.; Nagarakatte, S.; Roth, A. iCFP: Tolerating all-level cache misses in in-order processors. In Proceedings of the 2009 IEEE 15th International Symposium on High Performance Computer Architecture, Raleigh, NC, USA, 14–18 February 2009; pp. 431–442.
17. Barnes, R.D.; Sias, J.W.; Nystrom, E.M.; Patel, S.J.; Navarro, J.; Hwu, W.-m.W. Beating in-order stalls with "flea-flicker" two-pass pipelining. *IEEE Trans. Comput.* **2005**, *55*, 18–33. [CrossRef]
18. McFarlin, D.S.; Tucker, C.; Zilles, C. Discerning the dominant out-of-order performance advantage: Is it speculation or dynamism? *ACM SIGARCH Comput. Archit. News* **2013**, *41*, 241–252. [CrossRef]
19. Kulkarni, K.N.; Mekala, V.R. A Review of Branch Prediction Schemes and a Study of Branch Predictors in Modern Microprocessors. 2016. Available online: https://www.researchgate.net/profile/Venkata-Mekala/publication/266891966_A_Review_of_Branch_Prediction_Schemes_and_a_Study_of_Branch_Predictors_in_Modern_Microprocessors/links/545ac9ed0cf2c46f6643898c/A-Review-of-Branch-Prediction-Schemes-and-a-Study-of-Branch-Predictors-in-Modern-Microprocessors.pdf (accessed on 8 July 2022).
20. Mittal, S. A survey of techniques for dynamic branch prediction. *Concurr. Comput. Pract. Exp.* **2019**, *31*, e4666. [CrossRef]
21. Technology, N.S. Hummingbirdv2 E203 Core and SoC. Available online: https://github.com/riscv-mcu/e203_hbirdv2 (accessed on 8 July 2022).
22. Abella Ferrer, J.; Canal Corretger, R.; González Colás, A.M. Power-and complexity-aware issue queue designs. *IEEE Micro* **2003**, *23*, 50–58. [CrossRef]
23. Mittal, S. A survey of techniques for designing and managing CPU register file. *Concurr. Comput. Pract. Exp.* **2017**, *29*, e3906. [CrossRef]
24. Yeager, K.C. The MIPS R10000 superscalar microprocessor. *IEEE Micro* **1996**, *16*, 28–41. [CrossRef]
25. Inc, X. 7 Series FPGAs Configuration (UG470 v1.13.1). Available online: https://docs.xilinx.com/v/u/en-US/ug470_7Series_Config (accessed on 8 July 2022).