An Aggressive Implementation Method of Branch Instruction Prefetch

Yufeng Sun1,*, Yidong Yuan1, Weili Li1, Dongyan Zhao1, Liang Liu1, Rui Tian2

1Beijing Smartchip Microelectronics Technology CO., LTD, Beijing, China
2Metrology Center, STATE GRID Ningxia Marketing Service Center, Ningxia, China

*Corresponding author’s email: sunyufeng@sgitg.sgcc.com.cn

Abstract. Due to the demand in the field of industrial control, we developed a CPU core based on ARMv8m architecture [1], named TS400, and its performance is comparable to Arm Cortex-M33 [2]. The TS400 has four stages of pipelining, thus having a higher clock frequency than Cortex-M33 which has three stages. With the increase of pipeline stages, extra idle clock beats will be introduced when the pipeline is flushed, which will increase the clock cycles per instruction (CPI) value and decrease the score of CoreMark. Accurate instruction fetch and branch prediction can effectively reduce the impact of refreshing pipeline, at the cost of extra logic resources. In TS400, an aggressive branch instruction prefetch method is designed. Compared with branch prediction technology, this method does not need complex branch prediction logic and is suitable for the design of embedded CPU. The aggressive branch instruction prefetch method includes: 1) reducing the time cost of conditional branch target fetching to the minimum by taking the branch first and then confirming the branch taking result; 2) optimizing the bus control signal timing of the prefetch instruction to make the target address prefetch respond in time. The aggressive branch prefetch method reduces the impact of pipeline stall caused by the execution of conditional branch instructions as much as possible, thus achieving better running performance than Cortex-M33 at the same clock frequency, while the clock frequency performance is superior to Cortex-M33.

Keywords-embedded CPU; branch instruction; prefetch

1. Introduction

In 2011, arm introduced armv8 architecture, which was developed based on ARM32-bit architecture. Cortex-M series processors were widely used in various embedded application fields, such as industrial control system, intelligent measurement of power systems, automobile control systems, etc., because they can take into account performance, power consumption, code density and price. Cortex-M series processors once accounted for 50% of ARM processor shipments. Due to the limited application scenarios of Cortex-M series processors, their operating frequency is basically within 200MHz, and their microarchitecture features short pipeline stages and no branch prediction unit. Cortex-M33 processor is an important member of ARMv8 architecture processor family, which has the characteristics of high performance, low logic gate number, high configurability and low power consumption. The TS400 core design takes into account the application scenarios that require higher
performance, in which Cortex-M33 can't meet the performance requirements. In the process of design, we pursue higher performance and introduce four-stage pipeline structure. The features of Cortex-M33, Cortex-M4 [3] and TS400 are shown in Table 1.

TS400 can achieve slightly better score than M33 in CoreMark running, which is largely due to adopting an aggressive prefetch method in the execution of branch instructions. Compared with the general implementation of instruction prefetch, this method makes the fetching process of branch more coupled with the CPU pipeline at a lower cost of logic gate resources, reduces the CPU pipeline stall cycle and improves the CPU pipelining efficiency. In the following chapter III, the general structure of the fetch module and the influence of branch instruction on pipeline stall will be introduced. In chapter IV, we will introduce the general implementation method of branch instruction prefetch. In chapter V, the implementation method of aggressive branch instruction prefetch implemented in TS400 will be introduced. In chapter VI, the CPU running performance results based on different branch instruction fetching methods are given. The last chapter VII is the conclusion.

|                | M33 | M4   | TS400 |
|----------------|-----|------|-------|
| Pipeline stages| 3   | 3    | 4     |
| Instruction issue | partly dual issue | single issue | partly dual issue |
| Dynamic branch prediction | no | no | no |
| CoreMark@/Mhz     | 4.02 | 3.42 | 4.04 |

2. Instruction Fetching and Performance Loss of Branch Implementation

The instruction fetching unit (IFU) is responsible for feeding the instruction pipeline with the most likely stream of instructions, and balancing the speed difference between the pipeline running speed and the bus memory access speed [4,5]. The structure of the instruction fetching module is shown in Figure 1.

Instruction buffer is used to buffer the instructions read through AHB bus. It alleviates the mismatch between pipeline running speed and instruction fetching delay, and improves the parallelism of instruction fetching and instruction execution. However, when the branch instruction is executed, if the branch is taken, the instruction loaded in the instruction buffer needs to be discarded, and the target address instruction needs to be acquired through the instruction bus. In the meantime the pipeline needs to keep waiting, resulting in performance loss. For unconditional branch, the loss of reframing is inevitable. For conditional branch, when the condition fails, the instruction continues to be executed
sequentially, and the decoding stage can still obtain the instructions from the instruction buffer, thus avoiding the performance loss of re-fetch. When the condition passes, it is necessary to clear the instruction buffer and re-fetch. However, because the condition judged by the branch instruction may be related to the result of the instruction executed before it, the cycle of waiting for the condition result will be additionally introduced. TS400 takes this into account, and initiates the access to the target address of the branch instruction when the condition is not yet confirmed, and hides the cycle of waiting for the condition confirmation in the memory access cycle of re-fetching, thus reducing the performance loss of re-fetching.

Branch instructions appear frequently in programs, and a large number of studies show that the proportion of branches in the whole program accounts for about one-third [6,7,8]. Elaborate design of the execution of branch instructions can significantly improve the performance of CPU cores.

3. Prefetch Method of Branch Instruction

A typical sequential pipeline conditional branch execution is shown in Figure 2.

![Figure 2. Sequential Conditional Branch Execution](image)

The process is as follows: 1) initiate conditional branch in pipeline execution stage, 2) stall pipeline when conditions are not ready, 3) judge conditions when conditions are prepared, 4) if the branch is taken, send branch request and branch address to fetch stage, otherwise the program counter is counted sequentially, 5) send the branch address to instruction bus, 6) wait for instruction data to be valid, 7) register the instruction data and send it to decode stage[9,10]. In this way, the process of updating the condition code and accessing the bus of the fetch stage is executed in sequence. Assuming that the updating of the condition code requires m cycles, the bus access requires n cycles, and the bus data needs to be registered to the decode stage, the number of cycles that need to be stalled in the pipeline is m+n. For example, if the instruction before conditional branch is a division instruction, the division calculation takes 4 cycles, and the branch target address instruction fetching takes 2 cycles, then six bubbles need to be inserted in the pipeline.

Prefetch of branch instructions [3,11] refers to processing at the decoding stage before the branch instruction flows to the execution stage. For unconditional branch, directly send the branch request and branch address to the fetch stage. For conditional branch, if there is no correlation between the condition and the execution of the previous instruction, whether to send branch request to the fetch stage is decided according to the condition code value. If there is correlation between the condition to be judged and the previous instruction, it is necessary to stall the pipeline and wait for the condition code to be refreshed. For the scenarios of unconditional branch and conditional branch having no correlation with the previously executed instruction, the prefetch of the branch target will reduce the instruction fetching time by one cycle, thus reducing the pipeline stall by one cycle, as shown in Figure 3. For the scenario with conditional branch whose condition is related to the previously executed instruction, the execution of the branch instruction is the same as the sequential pipeline conditional branch.
4. Branch Prefetch of TS400

Aggressive branch prefetch method for branch instruction execution is adopted in TS400. The implementation flow is shown in Figure 4.

The difference between the aggressive branch prefetch method and the general branch prefetch method is that when the decode stage does not obtain the exact result of the branch condition to be judged, it still initiates a branch to the instruction fetching stage from decode stage, and gives a branch confirmation signal when the condition code result is obtained from execution stage, thus reducing pipeline stall. If the branch is taken, it discards the branch target instruction, and continues to fetch the instructions from the instruction buffer in sequence and send them to the decode stage. Thus, the process of determining the condition code and fetching the branch target is parallel. Take the previous example as an example, the pipeline only needs to stall 4 clock cycles in the case a division instruction is followed by a conditional branch having correlation with it. From the view of the decode stage, there is no difference between sequential fetching and branching, which greatly improves the instruction operation efficiency.

![Figure 4. Branch Target Prefetch In Ts400](image-url)
In order to improve the clock frequency, we had considered registering the address sent to the bus, but when the branch is taken, the next address after the branch target needs an extra clock cycle to be valid, which leads to a stall of one more clock cycle in the fetching process, so we gave up registering the address and made effort to make address logic as clean as possible.

![Program Counter Logic In Instruction Fetching Stage](image)

**Figure 5.** Program Counter Logic In Instruction Fetching Stage

TS400 has further optimized the bus access control at the fetch level to shorten the response time of re-fetch, as shown in Figure 6. In the figure, in the shaded clock cycle, the access address of addr1+4 is being sent to the instruction bus. At this time, it should be possible to cancel the access and make new access to branch target address: new_addr. However, since force_ready is 0, the fetch logic does not draw a conclusion that the branch address can be received at this time, so it continues to access addr1+4 address, and then accesses new_addr. Compared with stopping access to addr1+4 immediately, this method wastes 2 cycles as shown in Figure 6. The wasted time is determined according to the bus response signal. If the response signal does not arrive in time, more clock cycles will be wasted.

The above bus access logic is optimized in TS400. When force_valid is 1, no matter force_ready is 1 or not, the branch address is directly initiated. The improved access logic is shown in Figure 7. This improvement eliminates the useless access of the instruction bus, thus is much more efficient.

5. Performance

CoreMark [12] was developed by Shay Gla-On of EEMBC in 2009, aiming to replace the outdated Dhrystone as the industry standard. CoreMark is developed based on C language, covering list processing, matrix operation, state machine, cyclic redundancy check(CRC) and other functions. The test program can be freely downloaded and transplanted to the platform to be tested, and it is one of the most common benchmark programs for evaluating the performance of embedded CPU.

Keil is used as the development tool, the compiler is arm-clang6.12, and the optimization option is -Omax. The test object is TS400 processor core. For conditional branch, it adopts sequential pipeline conditional branch, branch instruction prefetch and aggressive branch instruction prefetch respectively.

The results of CoreMark running scores are shown in Table 2.

|                  | clock | addr+1 | addr1+4 | new_addr |
|------------------|-------|--------|---------|----------|
| core_mtx_i_haddr |       |        |         |          |
| core_mtx_i_htrans| 2     | 3      | 2       |          |
| hready/haddracc  |       |        |         |          |
| mtx_core_i_hrdata|       |        |         |          |
| force_valid      |       |        |         |          |
| force_addr       |       |        | new_addr|          |
| force_ready      |       |        |         |          |
Figure 6. Instruction Fetching Before Optimization.

Figure 7. Instruction Fetching After Optimization.

It can be seen from the results of CoreMark score that the performance of the aggressive branch instruction prefetch implementation method in the design process of TS400 is improved by about 9.8% compared with the sequential pipeline implementation method and 4.4% compared with the general branch instruction prefetch method. Aggressive branch instruction prefetch implementation method significantly improves the CPU performance.

Table 2. CoreMark Scores of Ts400 Adopting Different Branch Instruction Implementation Method

| Branching method | Sequential pipeline | General prefetch | Aggressive prefetch |
|------------------|---------------------|------------------|---------------------|
| CoreMark@/Mhz    | 3.68                | 3.87             | 4.04                |

6. Conclusion
In this paper, aiming at the pipeline stall problem introduced in the process of conditional branch instruction execution, an aggressive implementation method of branch instruction prefetch is proposed. It can be seen from the cores of CoreMark that, this method reduces the performance loss caused by branch instruction execution, and achieves the maximum parallelization of instruction execution and instruction fetching under resource-constrained design constraints. Compared with Arm Cortex-M33, the CPU core designed in this paper has smaller CPI and higher clock frequency, which is suitable for industrial control applications with higher performance requirements.

Acknowledgment
This research was supported by the science and technology project of State Grid Corporation of China, Research on Key Technology of Low Power Consumption of Embedded CPU Core (No. 546816190004).

References
[1] Arm v8-M Architecture Reference Manual, DDI0553A.m ID02102019, 2020
[2] Arm Cortex-M33 Processor Datasheet, https://developer.arm.com/ip-products/processors/cortex-m/cortex-m33
[3] Arm Cortex-M4 Processor Datasheet, https://developer.arm.com/ip-products/processors/cortex-m/cortex-m4
[4] John L. Hennessy, David A. Patterson, “Computer Architecture: Aquantitative Approach”, ISBN 978-7-115-297765-5
[5] Balaram Sinharoy, R. Kalla, W. J. Starke, H. Q. Le, R. Cargnoni, J. A. Van Norstr, B. J. Ronchetti, J. Stuecheli, J. Leenstra, G. L. Guthrie, D. Q. Nguyen, B. Blaner, C. F. Marino, E. Retter and P. Williams 2011 IBM POWER7 multicore server processor *IBM Journal of Research and Development* **55** (3) p 1-1

[6] Douglas W. Clark and Henry M. Levy 1982 Measurement and Analysis of Instruction Use in the VAX-I 1/780 *The 9th Annual Symposium on Computer Architecture* **10** (3) pp 9-17

[7] Cheryl A. Wiecek 1982 A Case Study of VAX-II Instruction. Set Usage for Compiler Execution *Proceedings of the Symposium on Architectural Support for Programming Languages and Operating Systems* pp 177-184

[8] L. J. Shustek 1978 Analysis and Performance of Computer Instruction Sets *Stanford Linear Accelerator Center*

[9] Killian, E. 1991 MIPS R4000 technical overview–64 bits/100 MHz or bust *Hot Chips III Symposium Record*, Stanford University, Palo Alto, Calif., 1.6–1.19.

[10] Jean-Loup Baer 2010 Microprocessor Architecture: From Simple Pipelines to Chip Multiprocessors *Cambridge University Press*

[11] Lam, M. S., and R. P. Wilson 1992 Limits of control flow on parallelism Proc. 19th Annual Int’l. Symposium on Computer Architecture (ISCA) *Gold Coast Australia* pp 46-57

[12] https://www.eembc.org/coremark/.