Design and Optimization of a Divider in Pipelined Processors Based on Quotient-Try

ZHANG Yu[1,2], LI Chunqiang[1,2], HU Wei[1,2], SHEN Huan[1,2], LIU Jing[1,2]
College of Computer Science and Technology, Wuhan University of Science and Technology, Wuhan, Hubei, 430065, China
Hubei Province Key Laboratory of Intelligent Information Processing and Real-time Industrial System, Wuhan, Hubei, 430065, China
zy_123f@foxmail.com, wustlcq@foxmail.com, huwei@wust.edu.cn, ShenHuan201620@outlook.com, luijing_cs@wust.edu.cn

***Abstract*—Pipeline is an effective approach to improve the performance of the processors, and the bottleneck of the pipelined devices is its longest execution path. In classical five-stage pipeline, division instruction takes up more clock cycles than the other general instructions because of its complex operations. In this paper, we proposed a design of optimized divider based on quotient-try. When DIV instruction is being executed, divisor and dividend are transmitted to the divider, while quotient and remainder are determined by using the highest bit of dividend minus divisor. This divider has been tested on simulation environment. The experiment results showed that this divider could reduce the complexity of circuit design and accelerate the speed of operating compared with the divider which was based on reciprocal-multiply.**

Keywords—FPGA, Quotient-try, Pipeline, Divider

I. INTRODUCTION

With the rapid progress of semi-conductor technology, the processors can provide powerful computing ability. When more and more transistors are integrated onto the single core, the performance of the processors has increased greatly in recent years [1-2]. At the same time, different approaches are proposed to improve the performance from the architectural views [3-6]. Instructions are the basic support for the processors. Add, subtract, multiply and divide operation is commonly used in the numerical calculation, in which division is the most complex one and the most difficult one to implement. Therefore, there are existing proposed approaches including SRT, GoldSchmidt and Newton-Raphson algorithms. A good divider design, can effectively improve the performance of the assembly line. In this paper, an optimization method of divider is proposed based on quotient-try in the pipelines. And it is implemented as a binary division by using Verilog HDL The 32 bits division requires at least 32 clock cycles to get the result.

system resources. Because of this, the actual throughput of the pipeline is:

$$T_p = \frac{n}{\sum_{i=1}^{m} \quad +}$$

$$T_p = \frac{n}{\sum_{i=1}^{m} \Delta t_k + (n-1)\max(\Delta t_1, \Delta t_2, \ldots, \Delta t_k)} \quad (2)$$

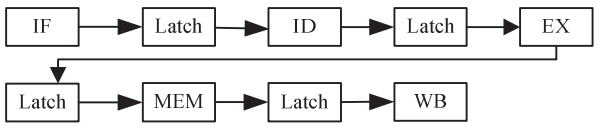
This paper is organized as follows. We describe pipeline technology in brief in Section 2, and describe our design in Section 3. In Section 4, we depict a case study and the analysis. And in Section 5, we draw the conclusions and give the future work.

II. PIPELINE TECHNOLOGY OVERVIEW

The pipeline technology, which can improve the parallelism of the internal time of the processors, is to decompose a complex task into serval sub processing stages, each of which runs in parallel with the others. Typical processing of instructions can be divided into five stages: fetch, decode, execute, access memory and write back, as shown in Figure 1. A latch is required between two stages, which is used to ensure that the execution result of one stage is able to be passed to next stage. The main measures to measure the pipeline include throughput, speedup and efficiency. Throughput refers to the number of tasks or the number of outputs per unit time. The calculation formula is as follows:

$$T_p = \frac{n}{T_k} \quad (1)$$

Fig. 1. Instruction Pipeline



In (1), n is the number of tasks, T_k is the time to deal with n tasks. When the processing time of each stage of the pipeline is not equal, there is a "bottleneck" stage in this pipeline. The reason is when the longest stage is under processing, the other stages will be in the "stall" phenomenon, which will waste the

In (2), Δt_i is the consumed time of the i-th stage. The pipeline has m stages in total. The first part in the denominator is the time required for pipeline to complete the first part of the tasks. The second part is the time to complete the rest of n-1 tasks. Thus, the throughput of pipeline is limited to the stage which has longest processing time. This stage becomes the

978-1-5090-6161-7/17/$31.00 ©2017 IEEE

"bottleneck" of the whole pipeline. Optimizing the "bottleneck" parts can effectively improve throughput of pipeline. Divide instruction is such a "bottleneck" in the pipeline.

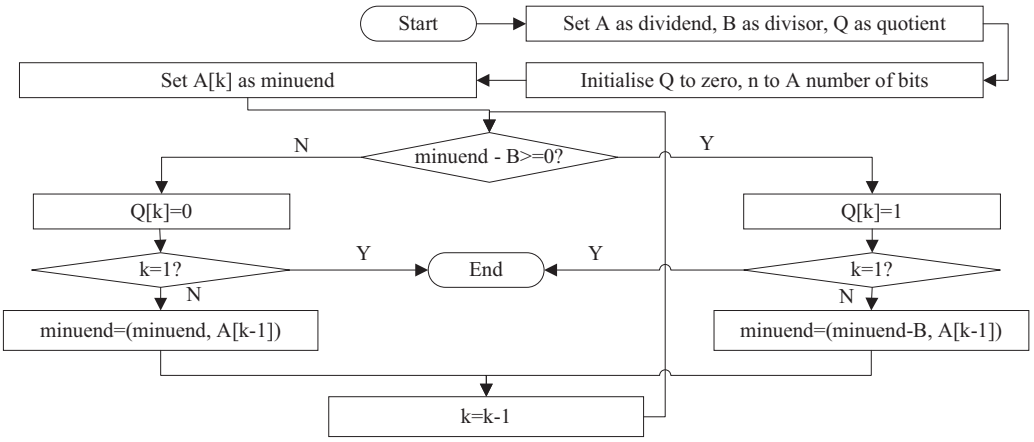
III. Quotient-Try Based Optimization

The basic idea of quotient-try is to use the highest of dividend minus the divisor, the quotient is determined 0 or 1 depending on the result of the subtraction operation, and then the quotient and remainder are got.

A. Description of Quotient-Try Based Division

Suppose that the dividend is A, the divisor is B, the quotient is Q, the remainder is R, and the bits of dividend is [n:1]. And then the processing of the division is showed as the follows in Figure 2.

Fig. 2. Flow Chart of Quotient-Try



Step 1. The highest A[n] is taken as the minuend from the dividend, and divisor B is taken as subtrahend, and then subtracts subtrahend from minuend. If the difference is less than 0, the highest of quotient is 0, or it will be 1.

Step 2. If the quotient in first step is 0, then the second highest bit is taken from the dividend and combined with the last minuend to be the new minuend A[n:n-1]. And the previous step is repeated. If the quotient in latest step is 1, the new minuend is A[n]-B:A[n-1], which is combined with the second highest from the dividend and the difference in latest step. The set n is now taken as n-1.

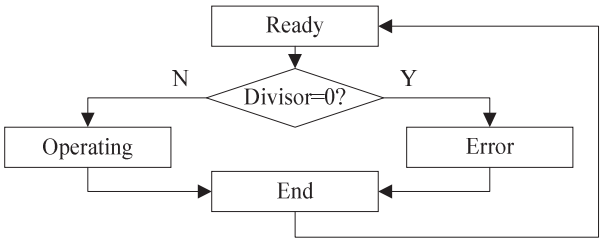
Step 3. The new dividend is used to minus the divisor B. If the difference is less than 0, the quotient Q[n] is 0, or it will be 1.

Step 4. Repeat the above steps until n is equal to 1.

B. Optimization Approach

This paper divides the divider into four stages according to the processing steps of divider including ready stage, operating stage, end stage and error stage. Each stage is described as the follows. In ready stage, the divider can enter the division operation. In operating stage, the divider is executing. In end stage, the divider runs over and outputs the results. In error stage, an error occurs. For example, the divisor is 0. When the divider in the initialization, it is at the ready stage, reading operands. If the divisor is 0, it means that an error occurred. The divider should enter error stage after it sets the quotient and remainder to 0 and returns results. If the divisor is not 0, the divider should enter operating stage and begin the computing until the division is completed. And it enters end stage and returns the final result. The transferring from one stage to another is shown in Figure 3.

Fig. 3. Transformation between Each Stage



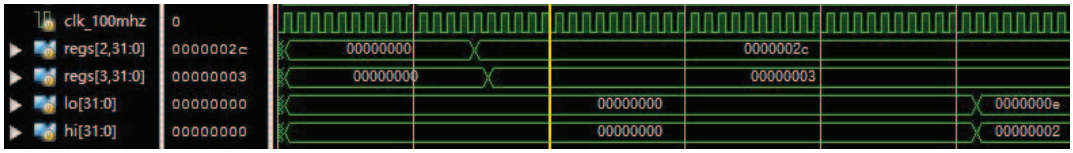
The divider receives operands at ready stage, and then enters operating stage or error stage according to the value of the divisor. After entering operating stage, a counter CNT should be initialized to 0, and the division should be decided as signed or unsigned. If it's signed division, and the dividend or divisor is a negative number, the negative one should be transformed to its complement. In the operating stage, if CNT is less than 32, it means that the division operation is not over, the quotient should be determined 0 or 1 depending on the difference at this time, and add 1 to CNT. If CNT is equal to 32, it means that the division operation is completed. If the division is signed and the dividend has different sign to the division, the results should be

transformed to complements. And then it enters the end stage. In end stage, the results are output, and the divider enters into ready stage again. However, if there is an error during the execution, the divider enters in the error stage.

IV. Implementation and Analysis

In this section, the optimized divider is implemented as a case. There are two numbers, a dividend A which value is 101100b and a divisor B which value is 11b. The emulation results of this divider is shown in Figure 4, in which $2 register holds the dividend 101100b and $3 register holds the divisor 11b. After 32 clock cycles, the quotient 00111b is got and stored in LO register and the remainder 10b is stored in HI register.

Fig. 4. Results of Emulation



The following is the detail. 1) The highest bit 1b of A is taken and minus B, and the result is 11b. 2) The difference is less than 0, and then the divider sets one bit of the quotient to 0, takes out the next bit of A, and combines with the last 1b. The new minuend, which is 10b, is used to minus 11b, which results the value of B. 3) The difference is less than 0, and then the divider sets one bit of the quotient to 0, takes out the next bit of A, and combines with the last 10b. The new minuend, which is 101b, is used to minus 11b, which results the value of B. 4) The difference is equal to 10b, and then the divider sets one bit of the quotient to 1, takes out the next bit of A, and combines with the difference 10b. The new minuend, which is 101b, is used to minus 11b, which results the value of B. 5) The difference is equal to 10b, and then the divider sets one bit of the quotient to 1, takes out the next bit of A, and combines with the difference 10b. The new minuend, which is 100b, is used to minus 11b, which results the value of B. 6) The difference is equal to 1b, and then the divider sets one bit of the quotient to 1, takes out the next bit of A, and combines with the difference 1b. The new minuend, which is 10b, is used to minus 11b, which results the value of B. 7) The difference is less than 0, and then the divider sets one bit of the quotient to 1. At this point, all bits of A have been taken out. The division is finished, and the divider gets the quotient 001110b and the remainder 10b.

The reason why the divider is difficult to implement is that division shall get each bit of quotient in turn and is not the same as multiplication, which can get part of results in parallel. The divider based on reciprocal-multiplication needs to evaluate the reciprocal, which increases the complexity of the design, and is only applicable to floating point division operation. This paper proposed a new method based on quotient-try for the divider implementation, which occupies little logic resource. As it shown in Figure 4, in 100MHz system clock, a 32-bit division requires only 640ns to be completed.

V. Conclusions and Future Work

Processors are the most important components of the computers. Instructions provide the basic computing support for the processors. The efficiency of the basic instructions plays a key role in the performance enhancement. A divider, which can complete 32-bit integer division, is proposed in this paper based on quotient-try, which is easier to implement and has fast processing speed. However, the float point division has not been taken into consideration. And in addition, when the dividend is not very big, the divider will waste some cycles from the quotient-try. These issues should be optimized in the future.

References

[1] Hammond L, Nayfeh B A, Olukotun K. A Single-Chip Multiprocessor[J]. Computer, 1997, 30(9):79-85.

[2] Cescirio W, Baghdadi A, Gauthier L, et al. Component-based design approach for multicore SoCs[J]. 2002:789-794.

[3] Park Y, Seo S, Park H, et al. SIMD defragmenter: efficient ILP realization on data-parallel architectures[J]. Acm Sigarch Computer Architecture News, 2012, 40(1):363-374.

[4] Hormati A H, Choi Y, Woh M, et al. MacroSS: macro-SIMDization of streaming applications[J]. Acm Sigplan Notices, 2010, 38(1):285-296.

[5] Xu Q, Zhang Y, Chakrabarty K. SOC Test Architecture Optimization for Signal Integrity Faults on Core-External Interconnects[C]. IEEE Design Automation Conference. IEEE, 2007:676-681.

[6] Pereira, Monica Magalh&#, de Ara&#, jo, S&#, et al. Using traditional loop unrolling to fit application on a new hybrid reconfigurable architecture[C]. ACM Symposium on Applied Computing.

[7] Kumar D, Singh K P. Design of High performance MIPS-32 Pipeline Processor[C]. International Conference on Recent Trends of Computer Technology in Academia. 2012.

[8] Reaz M B I, Islam M S, Sulaiman M S. A single clock cycle MIPS RISC processor design using VHDL[C]. IEEE International Conference on Semiconductor Electronics, 2002. Proceedings. ICSE. 2003:199-203.

[9] Balpande R S, Keote R S. Design of FPGA Based Instruction Fetch & Decode Module of 32-bit RISC (MIPS) Processor[C]. Communication Systems and Network Technologies (CSNT), 2011 International Conference on. IEEE, 2011:409-413.
[10] Aneesh R, Jiju K. Design of FPGA based 8-bit RISC controller IP core using VHDL[C]. India Conference (INDICON), 2012 Annual IEEE. IEEE, 2012:427 - 432.
[11] Bahaidarah, M, Al-Obaisi, H, Al-Sharif, T, et al. A novel technique for run-time loading for MIPS soft-core processor[C]. Electronics, Communications and Photonics Conference. IEEE, 2013:1-4.
[12] Ghosal M, Deshmukh A Y. Design of RISC Based MIPS Architecture with VLSI Approach[M]. Wireless Networks and Computational Intelligence. Springer Berlin Heidelberg, 2012:456-466.
[13] Pradhan M. Simulation and Verification of Self Test 16Bit Processor[J]. International Journal of Computer Applications, 2011, 20(1):42-45.
[14] Patterson D A, Hennessy J L. Computer Organization and Design, Fifth Edition: The Hardware/Software Interface[M]. Morgan Kaufmann Publishers Inc., 2013.
[15] Gimarc C E, Milutinovic V M. RISC principles, architecture, and design[C]. High-Level Language Computer Architecture. 1989:1-64.
[16] Harris D M, Harris S L. - Digital Design and Computer Architecture[M]. Digital design and computer architecture =. Chian Machine Press,, 2014:11-11.
[17] De J R R, Ordaz-Moreno A, Vite-Frias J A, et al. 8-bit CISC Microprocessor Core for Teaching Applications in the Digital Systems Laboratory[C]. IEEE International Conference on Reconfigurable Computing and Fpga's, 2006. Reconfig. IEEE, 2006:1-5.