Performance evaluation of Cascade ALU architecture for asynchronous super-scalar processors

Motokazu OZAWA[†]
ozawa@hal.rcast.u-tokyo.ac.jp

Masashi IMAI[†]
miyabi@hal.rcast.u-tokyo.ac.jp

Yoichiro UENO[‡]
ueno@hpcl.c.dendai.ac.jp

Hiroshi NAKAMURA[†]
nakamura@hal.rcast.u-tokyo.ac.jp

Takashi NANYA[†]
nanya@hal.rcast.u-tokyo.ac.jp

[†]Research Center for Advanced Science and Technology, The University of Tokyo, Japan
[‡]Dept. of Information and Communication Engineering, Tokyo Denki University, Japan

Abstract

Current out-of-order architectures have the critical path in the memory structure. Since the memory access delay mainly consists of wire delays, the feature size reduction will make little contribution on the critical path reduction. Therefore, the performance of the out-of-order architecture will not improve in spite of an expected advance in future technologies. To solve this problem, we present a novel architecture, called the Cascade ALU architecture, in which the critical path lies in the ALU. Since the ALU latency mainly consists of gate delays, the cycle time can be reduced with feature size reduction. In the Cascade ALU architecture, the instruction execution latency varies depending on executed instructions. Thus, an asynchronous implementation is suitable for the Cascade ALU. Since asynchronous handshake overhead may be too large to enhance the processor performance with the Cascade ALU. We show a method for hiding the handshake overhead, based on the fine-grain pipelining. Finally, we show the evaluation results that demonstrate the Cascade ALU architecture can achieve a good performance scalability in the ALU latency reduction.

1 Introduction

The performance of a synchronous processor is represented as the product of IPC (Instructions Per Cycle) and clock frequency. In the current processors, a super-scalar architecture with out-of-order instruction execution is widely used to achieve high IPC. Although these architectures need large logic circuits such as dynamic instruction scheduling, not only the IPC but also the clock frequency have been significantly improved so far. The reason is that the cycle time is reduced by feature size reduction even if the large logic circuits are contained.

However, it is going to be widely recognized that wire delays, instead of gate delays, are moving into dominance in VLSI design with future semiconductor technologies [3]. Therefore, the memory access delay is hard to be reduced in spite of feature size reduction because its delay mainly consists of wire delays. On the other hand, it is reasonable to assume that the ALU delay which mainly consists of gate delays still continues to be reduced as the feature size reduces.

In an out-of-order architecture, the critical path is determined by the instruction window access time [8]. Since the instruction window consists of memories, the cycle time will not be reduced by a feature size reduction. As a result, advanced semiconductor technologies do not necessarily imply any performance enhancement [1].

To solve this problem, we present a new architecture which we call the Cascade ALU architecture. The Cascade ALU architecture is a super-scalar architecture with in-order instruction issue. In the Cascade ALU, the dependencies among instructions are solved by the cascading connection of ALUs. Since the instructions contain many dependencies, the critical path is determined by the ALU latency. This implies that its cycle time can be further reduced with an advance in semiconductor technologies.

In the Cascade ALU architecture, the instruction execution latency varies depending on the data dependencies. This makes an asynchronous system more suitable to implement the Cascade ALU than a synchronous system. However, since the cycle time of asynchronous processors tends to become large because of the handshake overhead [6, 11], the overhead reduction is essential for the Cascade ALU to achieve high performance. Thus, we show a method for hiding asynchronous handshake overhead, based on the fine-

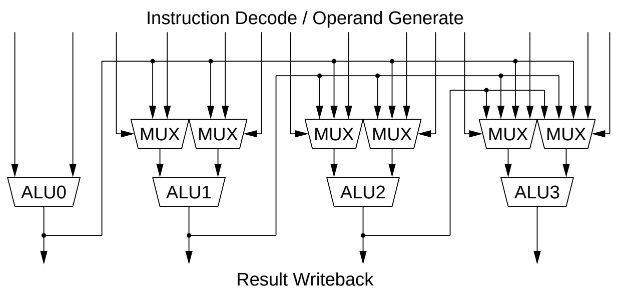


Figure 1. Structure of basic Cascade ALU

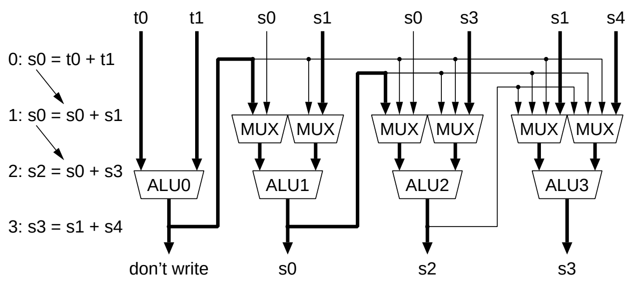


Figure 2. Example of instruction execution

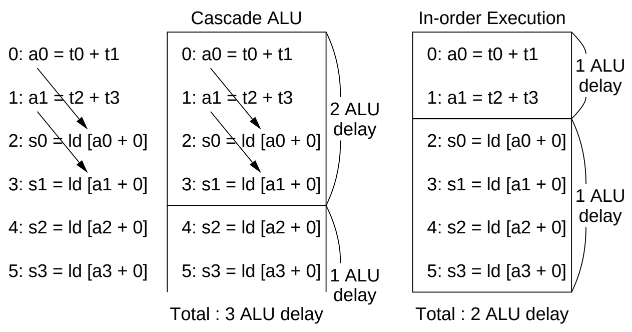


Figure 3. Performance degradation case

grain pipelining.

Finally, we present some evaluation results for the Cascade ALU architecture. The results show that the Cascade ALU architecture achieves a good performance scalability in the ALU latency reduction with little area penalty.

2 Cascade ALU architecture

2.1 Basic Cascade ALU

Figure 1 shows the structure of a basic Cascade ALU which consists of four ALUs. The output of each ALU is connected to the inputs of the succeeding ALUs. The multiplexors associated with each ALU select appropriate operands either from the decode stage or the outputs of the preceding ALUs. The ALUs that execute instructions are allocated in an in-order way, starting from the leftmost ALU. The number of instructions that are issued at a time is equal to the number of ALUs included in the basic Cascade ALU.

The “select” signals for the multiplexors and the “enable” signals for the result write-back are appropriately generated so as to satisfy the data dependency that exists among the instructions issued simultaneously.

If a RAW (Read After Write) dependence exists among the instructions issued simultaneously, the multiplexor connected to the inputs of the ALU that executes the depending “Read instruction” selects the result of the last “Write instruction”. The execution of “Read instruction” starts only after the last “Write instruction” has completed. The ALU executing “Read instruction” is reserved during this period.

If a WAW (Write After Write) dependence exists, only the result of the last “Write instruction” is written into registers without waiting the preceding “Write instruction”. This behavior is correct only if the following “Read instruction” which has the RAW dependence uses the result of the last “Write instruction”. The register renaming is not required for resolving this dependence.

After all, the execution of instructions with RAW dependence must wait for the completion of the preceding “Write instruction” execution, while the instructions with no RAW dependence are executed immediately. Consequently, the Cascade ALU realizes out-of-order instruction executions within the instructions issued at a time.

Figure 2 shows an example of instruction execution. In this example, result of instruction `0` is not written because of WAW on `s0`. Then, the input MUX for instruction `1` selects instruction `0`’s result because of RAW on `s0` (instruction `0`’s destination). Similarly, the input MUX for instruction `2` selects instruction `1`’s result because of RAW on `s0` (instruction `1`’s destination). Since the instruction `3` does not have a RAW or WAW on any other instructions, its execution precedes instructions `1` and `2`. Suppose each ALU operation requires one time unit, then these four instructions need three time units for execution as shown below.

Time 1	`0:s0 = t0 + t1`	`3:s3 = s1 + s4`
Time 2	`1:s0 = s0 + s1`	
Time 3	`2:s2 = s0 + s3`	

2.2 Pipelined Cascade ALU

In some cases, the performance of the basic Cascade ALU mentioned in section 2.1 may become lower than an in-order instruction execution. Figure 3 shows an example of the performance degradation. In the Cascade ALU,

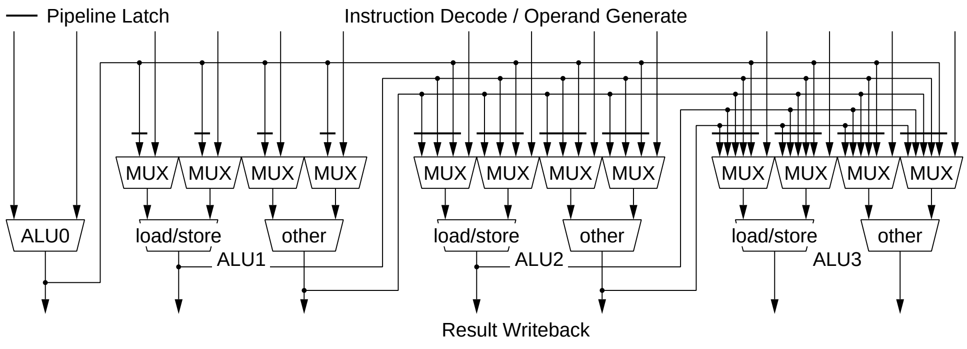


Figure 4. Structure of pipelined Cascade ALU

the instruction execution latency becomes large because of the ALU cascading. Therefore, the Cascade ALU requires 3 ALU delays while the in-order instruction execution requires only 2 ALU delays as shown in Figure 3.

Then, we introduce a pipelining technique into the basic Cascade ALU in order to reduce the cycle time. However, without increasing the number of available ALUs, the performance would not be improved well because of ALU conflicts. In the basic Cascade ALU, when the execution of an instruction is delayed because of RAW dependence, an ALU conflict may occur between the current instruction and the following instruction.

Instead of increasing the number of ALUs which requires a large amount of logic gates, we introduce a structure where each ALU operation (add, sub, logic, load / store etc.) is implemented as a separated block. Since the divided operations can be executed in parallel, we can avoid a resource conflict with little increase in the amount of logic gates in the pipelining of basic Cascade ALU.

In this structure, unless the current instructions try to use the same separated block as the previous instructions, the execution of the instructions is not postponed. Thus, decreasing the opportunity of ALU conflicts is important to achieve high performance. Generally, aggressive ALU separation will reduce the possibility of ALU conflicts while it causes an increase in the amount of wires and multiplexors.

A good compromise is the separation of load / store operations from other operations because these operations have a long execution latency. Therefore, we adopt the structure as shown in Figure 4, where ALU1, ALU2 and ALU3 are divided into the load / store operation and other operations. In this structure, ALU0 is not divided since any execution at the ALU0 is never postponed.

Figure 5 shows an execution of the pipelined Cascade ALU. In this figure, the executed instructions are the same as shown in Figure 3 and all the operation latencies are assumed to be one time unit. Since no resource conflicts between instructions `0,1,2,3` and `4,5,6,7` exist, instructions `4,5,6,7` can be issued at Time 1. As a result, the performance degradation is avoided in this case.

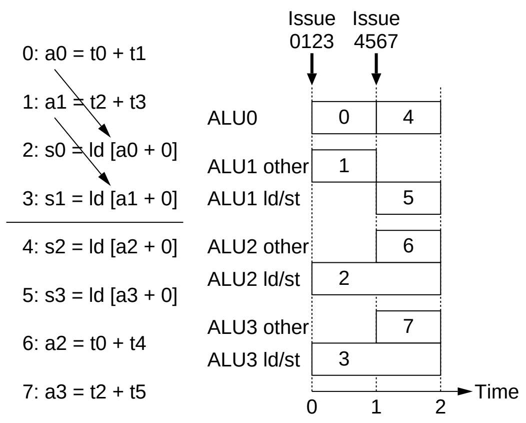


Figure 5. Instruction execution with pipelined Cascade ALU

We will call the “pipelined Cascade ALU” simply as the “Cascade ALU” in the succeeding sections.

2.3 Instruction issue method

In the ordinary out-of-order architecture, the instruction window dynamically assigns instructions to ALUs. However, the instruction window limits the overall cycle time as described in section 1.

Then, in the Cascade ALU, we choose the in-order instruction issue to eliminate the instruction window. In the Cascade ALU, the number of ALUs may become larger than the out-of-order architecture because instructions are assigned statically to ALUs. However, it does not always lead to the increase of total area because the instruction window is eliminated.

Figure 6 shows the structure of an instruction execution

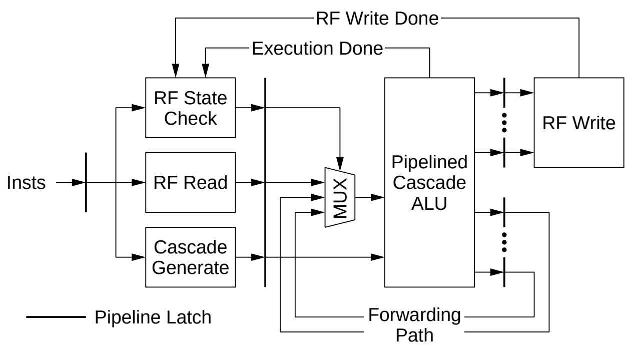


Figure 6. Instruction execution block

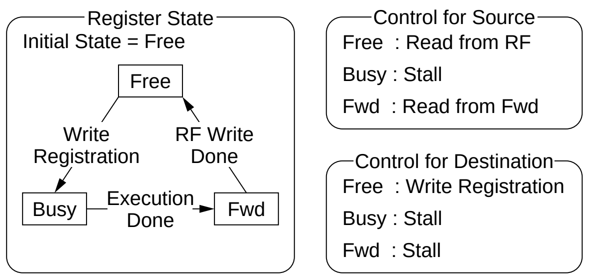


Figure 7. Register state and issue control

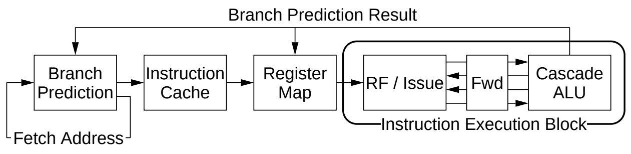


Figure 8. Instruction pipeline structure

block for the Cascade ALU. In this structure, the "RF State Check" holds states of registers to control the instruction issue. The following three states are required to control.

Free : The register is not the destination of any instructions under execution. The latest value are ready in RF.

Busy : The register is reserved as the destination of instructions under execution.

Fwd : The register is not the destination of any instructions under execution. However, the latest value exists in the forwarding register.

When the fetched instructions enter the execution block, the "RF State Check" checks the states for such registers that are used for the source and destination operands. Then, the issue are controlled depending on each state as shown shown in Figure 7. To avoid deadlock, the destination operands are not checked until all the states for the source operands become "Free" or "Fwd".

Since the execution completion timing varies depending on instructions, the "Execution Done" signal and the "RF Write Done" signal are controlled independently for each ALU and register. Thus, the issue timing for each instructions are determined independently. However, to simplify the issue process, all the simultaneously fetched instructions are issued at a time when all the instructions are available for issue.

2.4 Instruction pipeline

Figure 8 shows the instruction pipeline structure for Cascade ALU. This structure contains two blocks to support the speculative execution.

The "Branch Prediction" is the same mechanism as the conventional microprocessors.

The "Register Map" is used for the recovery from prediction mistake. The recovery method is based on MIPS R10000 [12]. In this method, the register mapping from ISA to physical register is always changed when the fetched instruction crosses the conditional branch. The register mapping is discarded when the correspond branch prediction is missed.

Thus, the processor structure with the Cascade ALU is the same as the current out-of-order processors except for the instruction execution block.

2.5 Qualitative analysis on performance

The difference between the Cascade ALU and the out-of-order architecture exists only in the instruction execution block. Therefore, we consider the performance with focusing on the instruction execution.

The ALU latency which consists of the gate delays can be reduced by the feature size reduction. On the other hand, the instruction window or the branch prediction requires memory accesses. Thus, the latency of these blocks can not be reduced easily by the feature size reduction. Moreover, pipelining these blocks does not helpful for performance improvement because the behavior of these blocks depends on the just previous output from themselves,

After all, the increasing difference of the latencies between instruction issue and ALU is essential when the feature size is reduced. Therefore, we consider the performance in case of the ALU latency reduction.

Figure 9 shows an comparison of the instruction execution between the out-of-order architecture and the Cascade ALU. In this figure, it is assumed that the out-of-order architecture and the Cascade ALU have the same number of ALUs, four instructions are fetched and issued every cycle,

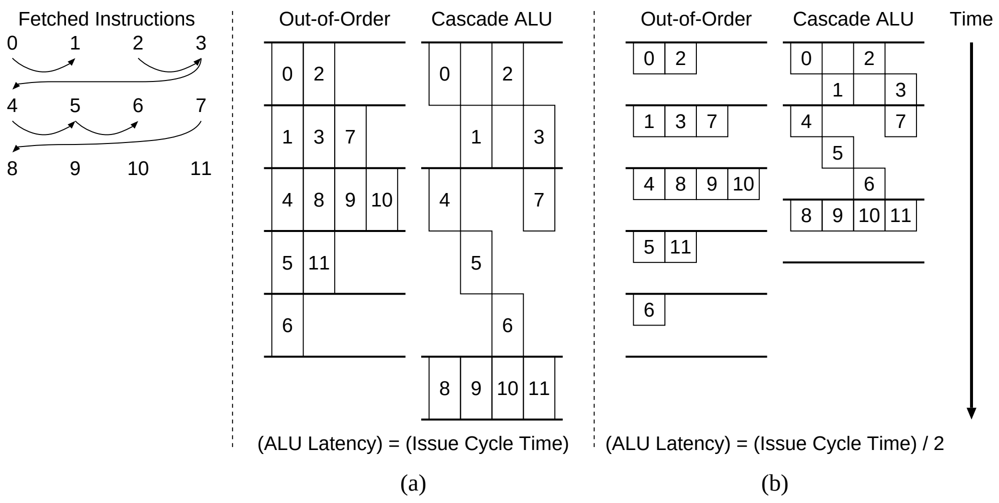


Figure 9. Comparison between out-of-order architecture and Cascade ALU

and the out-of-order architecture has an infinite number of instruction window entries. In “Fetched Instructions”, the arrows indicate RAW dependencies.

When the ALU latency is equal to the instruction issue cycle time, the execution with the Cascade ALU needs more time than the out-of-order architecture as shown in Figure 9 (a). The reason is that the Cascade ALU exploits less ILP than the out-of-order architecture because of the in-order instruction issue.

However, when the ALU latency is reduced to a half of the instruction issue cycle time, the execution with the Cascade ALU needs less time than the out-of-order architecture as shown in Figure 9 (b). The reason is that the Cascade ALU can execute two depending instructions within one instruction issue cycle. Thus, the total execution time is reduced by ALU latency reduction. Meanwhile, in the out-of-order architecture, the execution of two depending instructions always needs two instruction issue cycle. Therefore, the total execution time is not reduced when only the ALU latency is reduced.

Consequently, the performance of the Cascade ALU is dominated by the ratio of ALU latency to the instruction issue cycle time. The ALU latency is likely to be reduced by advanced process technologies, but the instruction issue cycle time is not reduced easily. Thus, the Cascade ALU is expected to have a good performance scalability with future process technologies.

Interconnections for cascading ALUs and forwarding path still require long wires in the Cascade ALU. However, it is presented in [8] that these delays are not critical if the number of ALUs is smaller than four. Of course, these delays should be considered to take a good scalability with the number of ALUs.

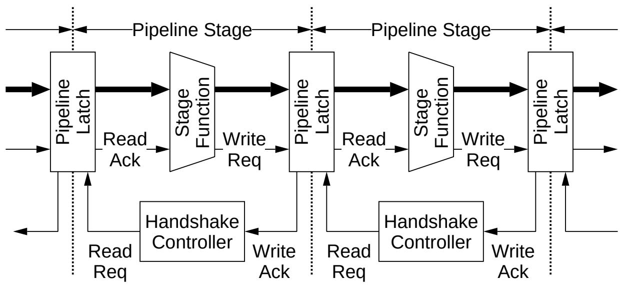


Figure 10. Asynchronous pipeline structure

3 Asynchronous implementation

In the Cascade ALU, the number of cascaded ALUs depends on the dependencies among the instructions. Moreover, the latency of each ALU varies with the operation type and the input value. Thus, the execution latency of the Cascade ALU dynamically changes from cycle to cycle.

Consequently, the asynchronous implementation is suitable for the Cascade ALU. However, in the asynchronous implementation, the handshake overhead causes the performance degradation. Therefore, we present a technique to reduce this overhead in this section.

3.1 Overhead of asynchronous pipeline

In an asynchronous pipeline structure, data transfer between stages is usually done by a handshake protocol as shown in Figure 10. If the operation delay varies from stage to stage, the read / write operation from or to a pipeline latch may be blocked.

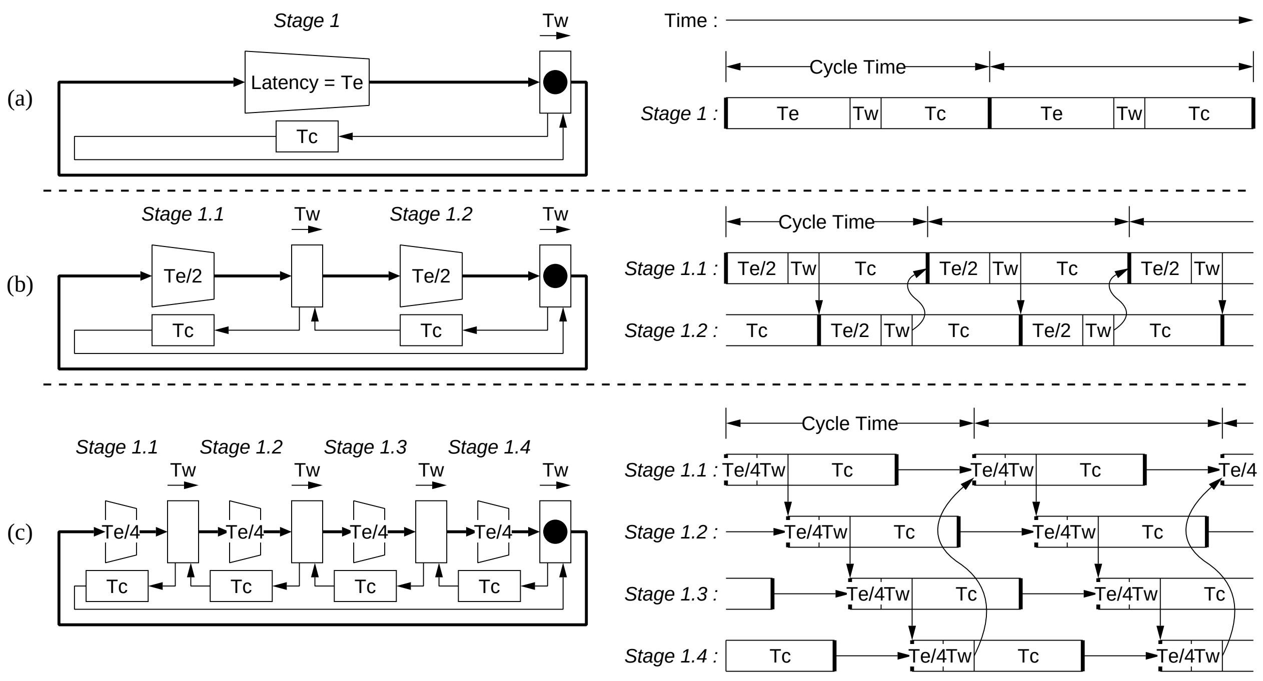


Figure 11. Effect of fine grain pipeline

In the asynchronous pipeline structure, each stage repeats the following sequence.

Exec : Execute stage operation (delay is Te)

Write : Write operation result (delay is Tw)

Ctrl : Reset stage and request next data (delay is Tc)

Therefore, the cycle time of the asynchronous pipeline can be estimated as “Te + Tw + Tc”. Since the cycle time of a synchronous pipeline is “Te + Tw”, the “Tc” is considered as the overhead due to the asynchronous handshaking. For example, the “Tc” delay accounts for about 50 % of the total cycle time in the asynchronous processor TITAC-2 [6, 11].

Recently, several fine-grain pipelining techniques [5, 9, 10] are introduced. In these techniques, a single stage function can be constructed by one or two logic cells at an extreme. If the pipeline is organized in a straight-line, these pipelining can effectively reduce the cycle time.

However, a processor pipeline is usually organized as a set of ring structures such as the result forwarding or the branch prediction. Since the ring structure throughput is restricted by the ring length and the number of data tokens at the initial time, the fine-grain pipelining does not necessarily imply a throughput enhancement.

As a result, the reduction of the “Ctrl” delay, as well as “Write” delay, is essential to achieve high performance in asynchronous processors as explained in the next subsection.

3.2 Fine-grain pipeline for hiding overhead

Figure 11 shows the effect of fine-grain pipelining to a ring-structured data-path. Figure 11 (a) shows an original ring structure which consists of one stage and one data token. Figure 11 (b) shows a ring divided into two fine-grain stages, and Figure 11 (c) shows the case where the original stage is divided into four fine-grain stages.

In Figure 11 (b) and Figure 11 (c), for each pair of two adjacent stages, the succeeding “Exec” (Te) and the preceding “Ctrl” (Tc) run concurrently, so that the “Control” delay (Tc) is hidden. As a result, the fine-grain pipelining can reduce the cycle time of the ring structure.

To achieve a performance enhancement in the fine-grain pipeline, the “Write” delay (Tw) must be small enough compared to “Exec” delay (Te) and “Ctrl” delay (Tc).

For example, the cycle time for the case in Figure 11 (b) is smaller than the case in Figure 11 (a), but the cycle time for the case in Figure 11 (c) is larger than the case in Figure 11 (b). This is because the “Write” delay (Tw) is not small enough to enjoy the effect of the fine-grain pipelining.

To eliminate the “Write” delay overhead, we can use the DDL (Differential Domino Logic) circuit (Figure 12) as the basic logic components. Since the stage function and the pipeline latch can be integrated in the DDL circuit, the “Write” delay can be considered as nearly zero. Therefore, the stage can be divided without any increase in “Write” delay.

Table 1. Cycle time of fine-grain pipelined ring (*ns*)

Tc	Number of physical stages						
	4	5	6	7	8	9	10
$2 \times 0.5 + 0.679$	5.683	**4.650**	**4.650**	**4.650**	**4.650**	**4.650**	**4.650**
$2 \times 1.0 + 0.679$	7.683	6.016	5.469	5.004	**4.650**	**4.650**	**4.650**
$2 \times 1.5 + 0.679$	9.683	7.682	6.914	6.004	5.835	5.539	5.180
$2 \times 2.0 + 0.679$	11.683	9.348	8.414	7.388	7.169	6.680	6.430

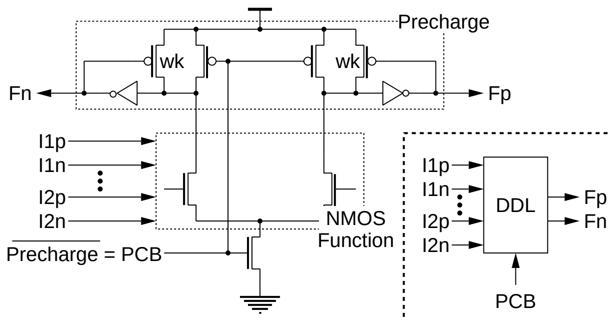


Figure 12. DDL circuit

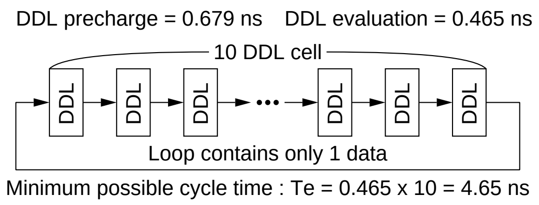

DDL precharge = 0.679 ns DDL evaluation = 0.465 ns
10 DDL cell
DDL
Loop contains only 1 data
Minimum possible cycle time : Te = 0.465 x 10 = 4.65 ns

Figure 13. Evaluated ring structure

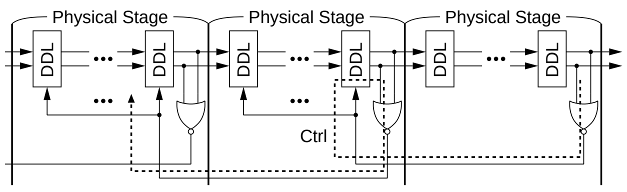


Figure 14. Structure of fine-grain pipeline

To confirm the effectiveness, we apply the fine-grain pipelining to a simple ring and show a simulation result. Figure 13 shows the simulated ring which consists of ten DDL cells. The DDL is designed with 0.25 μm process which is provided by NEC. Then, the delay is derived with 0.24 *mm* wire and 3 fan-out. It can be seen from Figure 13 that the minimum possible cycle time is 4.65 *ns*.

Figure 14 shows the fine-grain pipeline structure which is used here. This structure is known as a PS0 pipeline [10]. We divide the ring shown in Figure 13 into a number of physical stages. For example, if we add one NOR gate for every two DDLs, the number of total physical stages amounts to five (10/2). The dotted arrow in Figure 14 shows “Ctrl” path. The “Ctrl” delay (Tc) consists of “(Precharge) $+ 2 \times$ (NOR)”.

Table 1 shows the cycle time achieved for the cases where the NOR delay is 0.5, 1.0, 1.5, 2.0 *ns* and the number of physical stages is 4 through 10. Since the cycle time does not increase with the number of physical stages in this result, we see that the “Write” delay overhead is zero. If Tc is smaller than $2 \times 1.0 + 0.679$ *ns*, the cycle time can be reduced to the minimum possible cycle time (4.65 *ns*). This means that the overhead due to Tc is completely hidden. However, if Tc is larger than $2 \times 1.5 + 0.679$ *ns*, the cycle time can not be reduced as far as 4.65 *ns*. This means that the overhead due to Tc is not completely hidden, even if the extreme case (one DDL for one stage) of fine-grain pipelining is applied.

To summarize, the fine-grain pipelining can effectively hide the overhead of the asynchronous ring structure. However, the design effort for reducing “Ctrl” delay is still important since the effectiveness of the pipelining is limited.

4 Evaluation

To evaluate the Cascade ALU architecture, we have designed it at RTL[1]. Then, the Cascade ALU has been laid out with NEC 0.25 μm 4 layer metal CMOS process. The target process includes originally designed DDL cells and asynchronous cells such as C elements. The DDL cells implement any logic functions with up to three inputs.

4.1 Evaluation condition for Cascade ALU

Figure 15 shows the designed pipeline structure. The specification for each block or stage is as follows.

ISA : MIPS-I (integer only). Data width is 32-bit.

[1]The RTL source is available from `http://www.hal.rcast.u-tokyo.ac.jp/~ozawa/cgi/cvsweb.cgi`

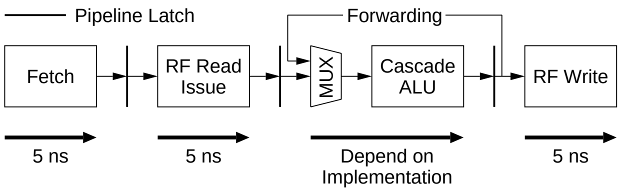


Figure 15. Evaluated pipeline structure

Fetch : 4 instructions are fetched every cycle. The multiply, divide and branch instructions can be fetched only 1 instruction per cycle. The instruction cache and branch prediction are assumed to hit always.

Issue : Instructions are issued as described in section 2.3. If the fetched instructions contain a branch, the issue waits until the preceding branch is resolved.

Cascade ALU : 4 ALUs are available. Each ALU is divided into the load / store and other operations. The structure is shown in Figure 4.

Register File : 8-read/4-write ports are provided. However, since the Cascade ALU requires 7 write ports in the worst case, Register File has arbiters to avoid conflicts between load / store and other operations. When a conflict occurs at the write port, the conflicting operation stalls until the write port is released.

Forwarding Path : 7 paths are available. Consequently, the forwarding path never conflicts.

Data Cache : 1 port is available. The data cache are assumed to hit always.

The designed pipeline structure does not contain the "Register Map" because the branch prediction always hits.

Figure 15 shows the latency assumptions for each stage. The latency of the branch prediction and the instruction / data cache with typical configurations is about 5 *ns* in the process we use. Thus, the latency for the "Data Cache" and all the stages except the "Cascade ALU" is assumed to be 5 *ns*. This latency can not be reduced easily with the feature size reduction because these blocks require memory accesses. Thus, this latency (5 *ns*) is assumed to be constant although the ALU latency is altered in the evaluation.

As described in section 2.5, the performance is affected by the ratio of ALU latency to the instruction issue cycle time. So we made three assumptions on operation latency as shown in Table 2. Note that `actual` denotes the average latency of a synthesized circuit with the process we use. In the `actual` latency, we assumed that all the cells have 0.24 *mm* wire and 3-fan-out.

Table 2. Operation latency assumptions (*ns*)

	`sync`	`moderate`	`future`	`actual`
`add`	5	3	1	2.65
`sub`	5	3	1	2.90
`shift`	5	3	1	2.44
`logic`	5	1	1	0.80
`cond`	5	3	1	1.96
`mul`	20	20	10	6.46
`div`	20	20	10	14.20

Table 3. Evaluated benchmarks

Name	Description	Number of instruction	Frequency of branch
`dhry`	dhrystone 2.1	609	20.20 %
`des` [7]	encrypt 8 bytes	823	0.49 %
`idct` [4]	8×8 IDCT	3022	1.65 %
`adpcm` [4]	decode 64 data	2503	28.05 %
`rs`	(64,56) RS coding	2706	11.68 %

`sync` **:** This latency is aligned to a multiple of the issue cycle time (5 *ns*).

`moderate` **:** This latency reflects `actual`. The difference between `moderate` and `actual` in `mul` and `div` has little effect on the performance because the benchmarks have few `mul` and `div` operations.

`future` **:** This is intended to reflect the latency which is reduced by scaling with the minimum feature size in future technologies.

Table 2 does not contain the load / store operation latency. The load / store operation consists of "Address Calc" (`add`) and "Data Cache Access". Therefore, the load / store latency is given by the sum of `add` and "Data Cache Access".

In the performance evaluation, we assume that all the asynchronous handshake overhead is hidden by using the fine-grain pipelining technique.

4.2 Performance evaluation method

Table 3 shows the evaluated benchmarks. All benchmarks are compiled by SGI MIPSPro C Compiler version 7.3.1.1m with `-non_shared -mips1 -O2`.

The performance evaluation is performed by measuring the execution time of benchmarks. In the Cascade ALU, the execution time is derived from the logic simulation results. Since the minimum possible cycle time of the Cascade ALU is still limited by the instruction issue cycle time, we choose the IPC calculated from Eq.1 as the performance index.

$$\text{IPC} = \frac{(\text{Issue Cycle Time}) \times (\text{Instruction Count})}{(\text{Execution Time})} \qquad (1)$$

To compare the Cascade ALU with the out-of-order architecture, we select MIPS R10000 [12] as the out-of-order architecture. As for MIPS R10000, the performance is evaluated under two assumptions.

`R10000 real`: All the parameters are the same as MIPS R10000. Thus, the instruction / data cache miss and the branch prediction miss may occur.

`R10000 ideal`: The instruction / data cache and the branch prediction always hit. Moreover, branch instructions can be fetched without limitation. Other parameters are the same as MIPS R10000.

In `R10000 real`, "Execution Time" in Eq.1 is derived from the execution results on the real machine (SGI O2 R10000 175MHz). On the other hand, in `R10000 ideal`, that value is derived from the cycle level simulation results.

In MIPS R10000, the cycle time is determined by the instruction window [12]. In this case, unless the instruction window latency is reduced, the cycle time is not reduced. Then, the performance can not be improved even though the ALU latency is reduced. Therefore, we do not change the ALU latency for MIPS R10000 evaluations.

In the Cascade ALU, we assume a perfect instruction / data cache and perfect branch prediction. On the other hand, only one branch instruction can be fetched per cycle. These assumptions are slightly different from those for MIPS R10000. Thus, in the performance comparison, we should consider that the performance is affected by these differences.

Firstly, we consider the difference on the instruction / data cache. In all the benchmarks, the data set is very small. Thus, the cache miss has little effect on the performance.

Secondly, we consider the difference on the branch prediction and instruction fetch. In `des` and `idct`, a few branch instructions are contained. Thus, the effect is negligible in these benchmarks. However, in `dhry`, `adpcm` and `rs`, a certain number of branch instructions is contained. Therefore, there would be effect on performance in these benchmarks.

For fair comparison, the performance of `R10000 real` should be considered better than the results shown here. The reason is that the performance is degraded by the branch prediction miss. On the other hand, the performance of `R10000 ideal` should be considered worse than the results shown here because the instructions can be fetched without any limitation in this model. However, the performance would be improved only a little even if all the fetch limitation are eliminated. Thus, even if we use the `R10000 ideal` performance directly, the fairness of the comparison can be kept.

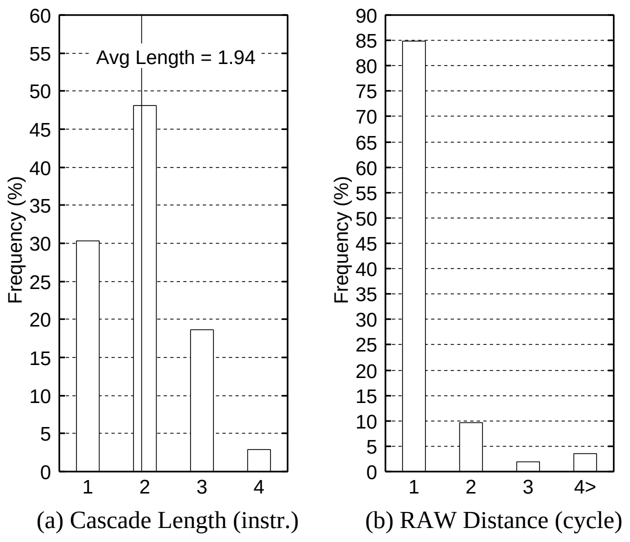


Figure 16. Benchmark characteristics

4.3 Performance evaluation result

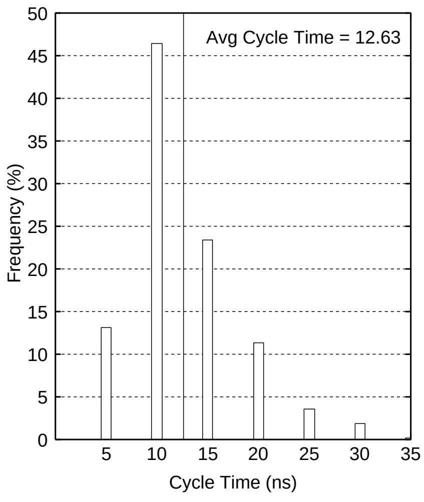
Figure 16 shows the benchmark characteristics which affects the performance of the Cascade ALU.

Figure 16 (a) shows the frequency of the ALU cascade length. The ALU cascade length is calculated assuming all the operation have the same latency. Since the average cascade length is around 2, the average instruction execution latency of the Cascade ALU becomes twice as large as each operation latency.

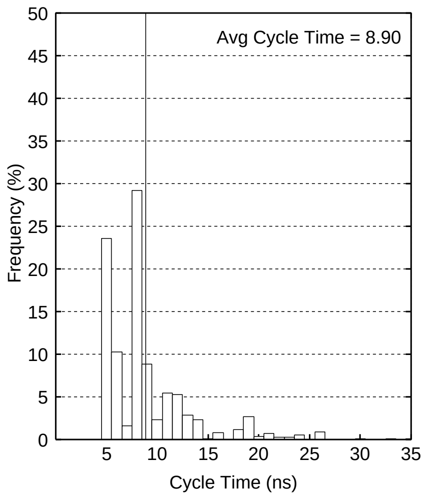
Figure 16 (b) shows the distance of the RAW dependencies between the instruction issue cycles. In this figure, almost 85 % of all the issues have RAW dependence on the immediately preceding cycle. In this case, many operands are supplied through the forwarding path. Thus, when the execution latency is larger than the instruction issue cycle time (5 *ns*), the cycle time is determined by the execution latency.

Figure 17 shows the evaluated IPC for the MIPS R10000 and the Cascade ALU. From the comparison among `sync`, `moderate` and `future`, we find out that the IPC for the Cascade ALU becomes higher as the ALU latency becomes smaller. This indicates that the performance of the Cascade ALU is certainly dominated by the ALU latency.

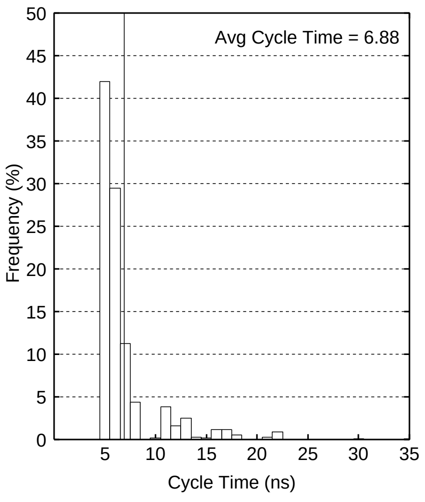
The IPC for `sync` is lower than that for MIPS R10000. In `sync`, the operation latency is almost the same as the instruction issue cycle time. In this case, as described in section 2.5, the Cascade ALU is slower than the out-of-order architecture because of the ILP disadvantage. On the other



(a) `sync`



(b) `moderate`



(c) `future`

Figure 18. Cycle time histogram for Cascade ALU

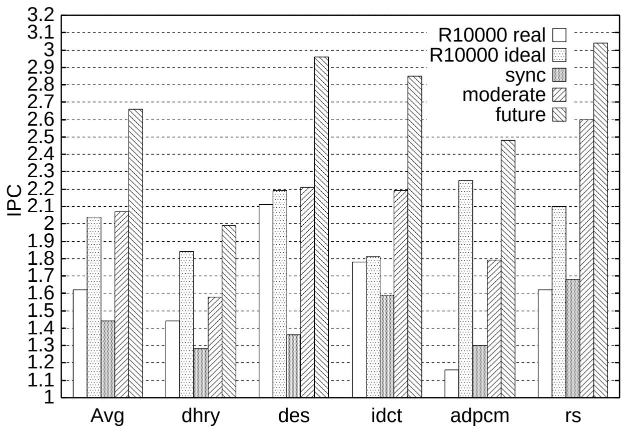


Figure 17. IPC results

hand, the IPC for `moderate` is close to that for `R10000 ideal`. This indicates that the ILP disadvantage in the Cascade ALU is compensated by the ALU latency reduction.

Figure 18 shows the histogram of the Cascade ALU cycle time. In all the histograms, the frequency of 5 *ns* cycle time is less than 50 %. This indicates that the cycle time of the Cascade ALU is not limited by the instruction issue cycle time (5 *ns*) even if the ALU latency is reduced to 1 *ns* in `future`. In `future`, the frequency of over 5 *ns* is about 55 %. Thus, the cycle time for about 55 % of all the instruction executions can be reduced by further reduction of ALU latency.

Table 4. Integer execution block area of MIPS R10000 (mm^2)

Function Block	0.35 μm	0.25 μm
Integer Data Path	7.91 (64 bit)	1.98 (32 bit)
Address Queue	8.38	4.19
Integer Queue	6.52	3.26
Total	22.81	9.43

These results indicate that the Cascade ALU has a good performance scalability on the ALU latency reduction which will be brought by the advance of future semiconductor technologies. Moreover, in `moderate`, the performance is almost the same as the current out-of-order processor. Consequently, the Cascade ALU outperforms current out-of-order processors with near future semiconductor technologies.

4.4 Area evaluation

As described in section 4.3, the performance of the Cascade ALU is almost the same as MIPS R10000. Next, we compare the Cascade ALU area with the integer execution block area of MIPS R10000 [2]. MIPS R10000 uses 0.35 μm technology and 64-bit ALU. Thus, the area of MIPS R10000 should be converted to 0.25 μm technology and 32-bit ALU for fair comparison. Table 4 shows the converted area of the integer execution block.

Figure 19 shows the layout of the Cascade ALU. In this

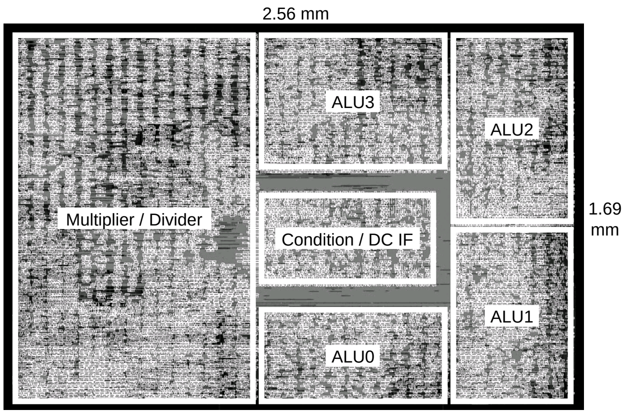


Figure 19. Layout of Cascade ALU

figure, the Cascade ALU occupies 4.33 mm^2. This area is larger than “Integer Data Path” of MIPS R10000 because the Cascade ALU has more ALUs. However, “Address Queue” and “Integer Queue” are not necessary in the Cascade ALU because of the in-order instruction issue. Thus, the total area of the instruction execution block on the Cascade ALU becomes smaller than MIPS R10000. This result shows that the Cascade ALU imposes little area penalty compared with the current out-of-order architecture.

5 Conclusion

A novel Cascade ALU architecture is proposed in this paper. An asynchronous system is suitable to implement the Cascade ALU architecture because of the execution latency fluctuation. For the asynchronous implementation, it is essential to hide asynchronous handshake overhead e.g. by using the fine-grain pipelining technique.

The performance evaluation result shows that the Cascade ALU has a good performance scalability on the ALU latency reduction. The layout result shows that the Cascade ALU imposes little area penalty compared with the current out-of-order architecture which achieves almost the same performance.

These results imply that the proposed Cascade ALU architecture can outperform the out-of-order architecture with an expected progress in future semiconductor technologies.

Acknowledgment

This work was supported in part by the Semiconductor Technology Academic Research Center (STARC) in Japan.

References

[1] V. Agarwal, M. S. Hrishikesh, S. W. Keckler, and D. Burger. Clock rate versus IPC: The end of the road for conventional microarchitectures. In *Proc. International Symposium on Computer Architecture*, pages 248–259, June 2000.

[2] A. Ahi, Y. chin Chen, R. Conrad, R. Martin, R. Ramchandani, M. Seddighnezhad, G. Shippen, H. men Su, H. Sucar, N. Vasseghi, W. V. Jr., and K. Yeager. R10000 superscalar microprocessor. In *Proc. HOT Chips VII*, Aug. 1995.

[3] International technology roadmap for semiconductors (1999 edition). `http://public.itrs.net/`.

[4] C. Lee, M. Potkonjak, and W. H. Mangione-Smith. MediaBench: a tool for evaluating and synthesizing multimedia and communicatons systems. In *Proc. International Symposium on Microarchitecture*, pages 330–335, Dec. 1997.

[5] A. J. Martin, A. Lines, R. Manohar, M. Nystroem, P. Penzes, R. Southworth, and U. Cummings. The design of an asynchronous MIPS R3000 microprocessor. In *Advanced Research in VLSI*, pages 164–181, Sept. 1997.

[6] T. Nanya, A. Takamura, M. Kuwako, M. Imai, M. Ozawa, M. Ozcan, R. Morizawa, and H. Nakamura. Scalable-delay-insensitive design: A high-performance approach to dependable asynchronous systems. In *Proc. International Symp. on Future of Intellectual Integrated Electronics*, pages 531–540, Mar. 1999.

[7] The Openssl Project. `http://www.openssl.org/`.

[8] S. Palacharla, N. P. Jouppi, and J. E. Smith. Complexity-effective superscalar processors. In *Proc. International Symposium on Computer Architecture*, pages 206–218, June 1997.

[9] S. Schuster, W. Reohr, P. Cook, D. Heidel, M. Immediato, and K. Jenkins. Asynchronous interlocked pipelined CMOS circuits operating at 3.3 – 4.5 GHz. In *International Solid State Circuits Conference*, Feb. 2000.

[10] M. Singh and S. M. Nowick. High-throughput asynchronous pipelines for fine-grain dynamic datapaths. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 198–209, Apr. 2000.

[11] A. Takamura, M. Kuwako, M. Imai, T. Fujii, M. Ozawa, I. Fukasaku, Y. Ueno, and T. Nanya. TITAC-2: An asynchronous 32-bit microprocessor based on scalable-delay-insensitive model. In *Proc. International Conf. Computer Design*, pages 288–294, Oct. 1997.

[12] K. C. Yeager. The MIPS R10000 superscalar microprocessor. *IEEE Micro*, 16(4):28–40, Aug. 1996.