An Efficient Instruction Fetch Architecture for a RISC-V Soft Processor on an FPGA

Hiromu Miyazaki
School of Computing,
Tokyo Institute of Technology
Tokyo, Japan
miyazaki@arch.cs.titech.ac.jp

Junya Miura
School of Computing,
Tokyo Institute of Technology
Tokyo, Japan
miura@arch.cs.titech.ac.jp

Kenji Kise
School of Computing,
Tokyo Institute of Technology
Tokyo, Japan
kise@c.titech.ac.jp

ABSTRACT

To reduce the code size of application programs for RISC-V soft processors on an FPGA, it is desirable for the processor to support the RISC-V compressed instruction extension. In this paper, we implement an efficient instruction fetch unit. We clarify the problem of instruction fetching in pipelining processors that support the extension. To solve the problem of instruction fetching, we propose two instruction fetch units using a decompressed cache and a compressed cache, respectively. We implement the proposed fetch units and evaluate their performance, hardware resources, and operating frequency. Through the evaluation, we show that the proposed unit with a compressed cache is the best and achieves 21.8% better fetch performance using reasonable hardware resources than the baseline architecture.

CCS CONCEPTS

• **Computer systems organization → Reconfigurable computing**; *Reduced instruction set computing*.

KEYWORDS

Soft Processor, RISC-V, FPGA, Instruction Fetch, Instruction Cache, Compressed Instruction

ACM Reference Format:
Hiromu Miyazaki, Junya Miura, and Kenji Kise. 2019. An Efficient Instruction Fetch Architecture for a RISC-V Soft Processor on an FPGA. In *The 10th International Symposium on Highly-Efficient Accelerators and Reconfigurable Technologies (HEART 2019), June 6–7, 2019, Nagasaki, Japan.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3337801.3337803

1 INTRODUCTION

The cost-effective soft processors for FPGAs like MicroBlaze [9] and Nios II [3] are widely used in various fields. On the other hand, RISC-V [1] is becoming popular as a RISC based open instruction set architecture (ISA) which has been developed at the University of California, Barkley, and the demand for cost-effective soft processors based on RISC-V will become higher. To reduce the code size of application programs, RISC-V defines 2-byte instructions as the compressed instruction extension [8] in addition to the standard 4-byte instructions, where each 2-byte instruction can be decompressed into the equivalent 4-byte instruction.

HEART 2019, June 6–7, 2019, Nagasaki, Japan

ACM ISBN 978-1-4503-7255-8/19/06.
https://doi.org/10.1145/3337801.3337803

We aim to develop a cost-effective scalar soft processor of RISC-V that supports the compressed instruction extension. As the first step, we implement a cost-effective instruction fetch unit. If an instruction fetch unit cannot supply enough instructions for a processor backend, the processor performance may be degraded significantly. Therefore, the performance of an instruction fetch unit is important. In a scalar processor of our target, it is optimal to fetch one instruction per each clock cycle.

In an instruction fetch unit, the following two operations are performed: to fetch an instruction specified by the program counter (PC), and to update the PC with the proper value. Because the access to the main memory is slow, an instruction cache is used to reduce the access latency. If an access to the instruction cache hits, an instruction can be fetched within a few cycles. This improves the efficiency of instruction fetch.

In this paper, we describe efficient instruction fetch architectures for FPGAs that support the compressed instruction extension of RISC-V focusing on the organization of an instruction cache.

We construct the baseline architecture of an instruction fetch unit for a RISC-V soft processor supporting the compressed instructions.

The main modules of this fetch unit are a conventional instruction cache [6], a gshare branch predictor [5], and branch target buffers (BTBs) [6]. This unit has the following two BTBs for storing branch target addresses: the first BTB (BTB1) is for conditional branches that may change the control flows depending on execution results, and the other BTB (BTB2) is for unconditional branches that always change control flows. This unit also has PC, IR (instruction register), some adders, a multiplexer, and other combinational circuits. The PC and IR are registers updating at the positive clock edge. This unit has some registers updating at the negative clock edge to map memories of some modules to BRAM of our target Xilinx FPGAs.

To fetch an instruction specified by PC, an instruction cache is accessed. Then the cache hit or miss is determined. If the cache hits, a valid instruction is fetched immediately. Otherwise, if the cache misses, the fetch unit accesses the main memory and copies a proper cache line to the cache, then it reaccesses the cache and hits this time.

The other operation of a fetch unit is to update PC with the proper address among five candidates as follows. The first candidate is obtained by adding 4 to the PC, which is used when a fetched instruction is 4-byte and not predicted to be a taken branch. The second candidate is obtained by adding 2 to the PC, which is used when a fetched instruction is 2-byte and not predicted to be a taken branch. The third candidate is a branch target address from BTB1, which is used when a fetched instruction is a conditional branch and is predicted to be taken. The fourth candidate is the branch target address from BTB2, which is used when a fetched instruction is an unconditional branch. The fifth candidate is the correct execution

address from the later pipeline stages, which is used when the computed PC in the fetch unit is wrong or an interrupt occurs. This is the highest priority in the five candidates.

Next, we describe the organization and behavior of the conventional cache in the baseline fetch unit. An example instruction sequence of RISC-V program that has 4-byte instructions and 2-byte instructions shown below is used.

```
0x0000 c.addi a1, 0x1    # 2-byte insn A
0x0002 c.addi a2, 0x2    # 2-byte insn B
0x0004 c.add  a1, a2     # 2-byte insn C
0x0006 add    a3, a1, a2 # 4-byte insn D
0x000A c.add  a2, a1     # 2-byte insn E
0x000C sub    a4, a1, a2 # 4-byte insn F
0x0010 c.add  a4, a3     # 2-byte insn G
```

We label each instruction as A, B, C, D, E, F, and G in order. An add instruction labeled as D and a sub instruction labeled as F are 4-byte instructions. Other instructions beginning with 'c.' are 2-byte compressed ones.

Because the minimum instruction length is 2-byte, we regard 2-byte as 1-word in this paper unless noted otherwise. An instruction may be one word or two words, so this cache outputs two words to fetch one instruction at least. If the cache misses, a stall occurs due to access to the main memory.

Figure 1(a) shows two cache lines of a conventional cache stored the example instruction sequence. We assume that the line size is 8-byte. Byte addresses are assigned to each word in cache lines. We assume that 4-byte instruction D consists of the lower 2-byte D1 and the upper 2-byte D2. Similarly, instruction F consists of F1 and F2. In this figure, 4-byte instructions are emphasized using gray color.

We describe the problem of the conventional cache focusing on the 4-byte instruction D. D1 and D2 for this instruction are stored across two cache lines. This occurs because 2-byte compressed instructions and 4-byte instructions are mixedly placed. Therefore, this unit has to access two cache lines when fetching instruction D. Even if both lines are stored in the instruction cache, it may read 2-byte D1 from Line 1 in the first cycle and may read 2-byte D2 from Line 2 in the next cycle. In this way, a 4-byte instruction across two cache lines is fetched by accessing in two cycles. This additional cycle of cache access cause performance degradation of instruction fetching.

2 PROPOSED ARCHITECTURES

2.1 Fetch Unit using Decompressed Cache

To solve the problem discussed in the previous section, we propose an instruction fetch unit using a novel instruction cache named *decompressed cache*. The drawback using a conventional cache is to access two cache lines for fetching one instruction as shown in Figure 1(a). This occurs because 2-byte and 4-byte instructions are mixedly mapped to cache lines. Therefore, if the instructions stored in a cache are all 4-byte, instructions can be aligned in cache lines and an instruction never crosses two cache lines.

The key idea of a decompressed cache is that all 2-byte compressed instructions are decompressed to 4-byte equivalent instructions and then stored in the cache.

Since each instruction stored in a decompressed cache is always 4-byte, we regard 4-byte as 1-word in section 2.1. Although an original 4-byte instruction or a 4-byte instruction decompressed from a compressed one is stored in each word, a minimum length

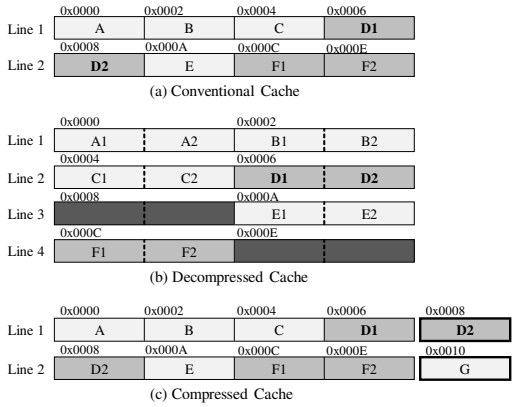

Figure 1: The behavior of storing the example sequence.

of the original instructions is 2-byte. Therefore, a start address of each word in a cache line should be aligned by 2-byte.

Figure 1(b) shows the behavior of storing the example instruction sequence on a decompressed cache whose block size is 8-byte. Each word in a cache line is shown by solid lines, and a dashed line indicates a boundary of 2-byte which is defined as 1-word in other sections. A start address of each word in Line 1 is aligned by 2-byte as 0x0000 and 0x0002, and the other lines are aligned by 2-byte as well. 2-byte instruction A is decompressed to a 4-byte instruction which consists of A1 and A2, and then stored in the left field of Line 1. Similarly, other 2-byte instructions are decompressed to 4-byte ones and stored. 4-byte instructions of D whose address of 0x0006 and F whose address of 0x000C shown in gray color are stored to their proper fields.

Although the instruction D is stored across two cache lines in a conventional cache, we can see that any instruction is stored in one cache line in a decompressed cache solving the drawback of a conventional cache.

The words emphasized using black color in this figure are defined as *blank words* which are never accessed and an arbitrary value can be stored. Because the example instruction sequence has no instructions whose address are 0x0008 or 0x000E, the words accessed by these addresses are blank words.

The drawback of this decompressed cache is as follow. Because this cache stores 4-byte instruction for 2-byte compressed instruction, the efficiency of cache utilization is reduced compared to a conventional cache. Moreover, when the cache block size is larger than a word, this cache has some blank words. It wastes the valid words in the cache and reduces its hit rate.

The first proposal is obtained by replacing an instruction cache in the baseline fetch unit with the decompressed cache.

2.2 Fetch Unit using Compressed Cache

We propose another instruction fetch unit using a novel instruction cache named *compressed cache*. The drawback of a conventional cache to access two cache lines for fetching 4-byte data occurs when the instruction fetch unit accesses from the last word of a cache line as shown in Figure 1(a) fetching instruction D.

The key idea to solve the drawback is adding one word to the end of each cache line. The slot to be stored the additional word is

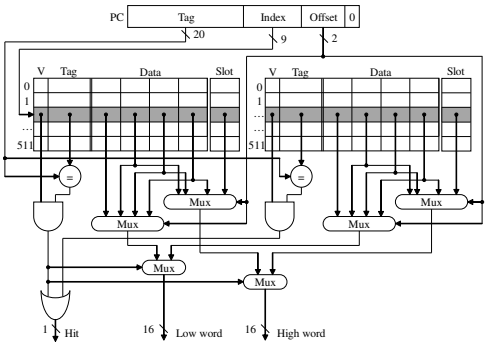

Figure 2: The block diagram of a compressed cache.

called a *complement word slot*. The next word of the last word of a cache line is always stored in a complement word slot added to the same cache line. In a compressed cache, access from a complement word slot is prohibited. By using this slot, 4-byte data is always output accessing just one cache line when an instruction cache hits, and the drawback will be solved.

Figure 2 shows the block diagram of a compressed cache of 2-way set associative having 512 sets. Each cache entry consists of a valid bit, a tag, a 4-word block, and a complement word slot. This cache outputs two words shown as a high word and a low word. If the fetched instruction is a 2-byte instruction, the low word is a valid instruction.

Figure 1(c) shows two cache lines storing the example instruction sequence. Each cache entry has an 8-byte cache line and a complement word. D2 and G are stored in each complement word slot. In a conventional cache, instruction D is stored across two cache lines. In the compressed cache, D1 and D2 are stored in the same cache entry of Line 1. Therefore, 4-byte instruction D can be fetched from Line 1 in one cycle. The main drawback of this compressed cache is the hardware resource increase because one additional word for a complement word slot is added to each cache line.

The second proposal is obtained by replacing an instruction cache in the baseline fetch unit with the compressed cache.

3 EVALUATION

We evaluate the proposed fetch units in terms of instruction fetch efficiency, operating frequency, and hardware resources to show that the proposal achieves better performance than the baseline. To evaluate the instruction fetch performance, we implement each proposed unit in Verilog HDL. The organization of the implemented instruction fetch unit includes an instruction cache, a gshare branch predictor, and BTBs.

We model a conventional scalar processor of 5-stage pipelining which has instruction fetch (IF), instruction decode (ID), execution (EX), memory access (MEM), and write-back (WB) stages. A branch instruction including both unconditional and conditional is executed in EX stage. If a branch prediction is wrong, it updates PC with the correct address and flushes wrong instructions in IF/ID and ID/EX pipeline registers.

All instruction caches adopt a typical 2-way set associative cache whose line size is 8-byte. The compressed cache has 2-byte complement word slot for each cache line. We set 8 cycles of a cache miss penalty considering the average penalty of 37 cycles obtained through the preliminary evaluation using a Nexys4 DDR board running a processor at 100MHz and DDR2 memory at 300MHz and assuming the use of an L2 cache. The number of sets for all cache configurations is 512, therefore the cache capacity is about 8KB[1]. A gshare branch predictor has a 16-bit branch history register and a 16KB pattern history table which is an array of 65,536 2-bit counters. Each BTB has a structure like a direct-mapped cache with 1,024 entries where each entry has 1-bit valid bit, 21-bit tag, and 31-bit[2] target address.

Table 1: The synthesize and implementation results.

	F [MHz]	Registers	LUTs	BRAMs
Decomp	86.2	220 (0.53%)	214 (1.0%)	11 (22%)
Comp	86.2	220 (0.53%)	223 (1.1%)	12 (24%)

The performance evaluation is done by trace-driven simulation in Verilog HDL. It uses the instruction traces obtained by executing some benchmark programs using Spike RISC-V ISA simulator [2]. We verified the correctness of our simulator by comparing the sequence of PC and IR values for retired instructions obtained by the implemented fetch unit and the correct sequence obtained by Spike simulator.

We use 4 benchmark programs including 099.go which is an internationally ranked go-playing program, 129.compress which is an in-memory version of the common UNIX utility, 130.li which is Xlisp interpreter, and 147.vortex which is an object-oriented database from SPEC CINT95 [7]. Although CINT95 has eight programs, we use four programs which can be compiled with a little code modification using a RISC-V cross compiler, gcc version 7.2.0 using -O2 optimization flag. For each benchmark program, the first 70 million instructions are skipped to exclude its initialization phase and the following 20 million instructions are used for evaluations.

The performance metric is the fetched valid instructions per cycle (FIPC) which is obtained by dividing the number of valid fetched instructions by the number of elapsed cycles. The ideal fetch unit fetching one instruction every clock cycle has FIPC = 1.

We evaluate hardware resources and frequency of proposed fetch units which are implemented on xc7a35tcsg324-1 Artix-7 of low-power and xc7vx485tffg1761-2 Virtex-7 of high-performance by using Xilinx Vivado 2017.4[3]. We select the highest frequency which satisfies the clock constraint changing by 0.2 ns steps.

The fetch units with a conventional cache, a decompressed cache, and a compressed cache are named *Baseline*, *Decomp*, and *Comp*, respectively.

Table 1 shows the synthesis and implementation results of two proposed units targeting Artix-7 FPGA. The maximum operating frequency, the number of used registers, the number of used LUTs as logic, and the number of used BRAMs are denoted as F, Registers, LUTs, and BRAMs, respectively. The numbers in parentheses indicate the utilization rate of the entire FPGA. The maximum operating frequency of both Decomp and Comp is the same as 86.2MHz because their critical paths are almost the same. These critical paths

[1] The capacities are different due to the use of a complement word slot, and so on. The total size of a compressed cache is 12.625KB.
[2] Although PC is 32-bit, 31-bit is enough for a target address because the LSB is always zero.
[3] For synthesizing Flow_PerfOptimaized_high strategy is used. For implementation Flow_RunPostRoutePhysOpt strategy is used.

Table 2: The performance evaluation results.

		099.go	129.co	130.li	147.vo
Compressed insn rate		0.515	0.298	0.701	0.725
Gshare hit rate		0.808	0.920	0.957	0.974
Baseline	FIPC	0.542	0.783	0.767	0.589
Decomp	FIPC	0.416	0.975	0.859	0.429
	Cache hit rate	0.825	0.999	0.983	0.834
Comp	FIPC	0.630	0.979	0.959	0.694
	Cache hit rate	0.929	1.000	1.000	0.947

are from registers which are updated at the negative clock edge in a gshare branch predictor, BTBs, and a cache through a multiplexer to PC which is updated at the positive clock edge. The hardware resource usages are almost the same, and their registers and LUT usages are less than 1.1% of entire FPGA. Comp occupies 12 BRAMs which are one more BRAMs of Decomp because it requires a complement word slot for each cache line. Among these 12 BRAMs, 4 are used for a cache, 4 are used for a gshare branch predictor, 2 are for BTB1, and 2 are for BTB2. In an evaluation using Virtex-7 FPGA, the maximum operating frequency of Decomp is 135MHz, and that of Comp is 139MHz. The hardware resource usages are almost the same as one implemented on Artix-7.

Table 2 shows the evaluation results of the baseline and two proposals by Verilog simulations. The benchmark name is written in the first six characters. The first row shows the rates of compressed instructions. They vary from 29.8% of 129.compress to 72.7% of 147.vortex. The average of four benchmarks is 56.0%, and it is shown that the compressed instructions are effectively used to reduce their code size.

The hit rates of a gshare branch predictor for each fetch unit were the same value in three significant digits. Therefore, the value is shown in one row. As shown in the table, The gshare hit rates range from 80.8% to 97.4%, and their average is 91.5%. This result matches the known hit ratio of gshare [5]. We see the hit rates of the instruction caches. The result shows the hit rate of Comp is better than the one of Decomp in all benchmarks. On average, the hit rate of Comp is 96.9% while one of Decomp is 91.0%. This hit rate reduction of Decomp is due to the inefficient use of cache lines on a decompressed cache as shown in black in Figure 1(b). Moreover, a complement word slot helps to increase the line size. So, the hit rate of Comp is high as 96.9% on average.

The FIPC of Baseline is the estimated value from the FIPC of Comp because the implementation of Baseline has some options when two cache lines have to be accessed for fetching one instruction. We estimate the FIPC of Baseline by assuming that two clock cycles are required to access two cache lines for an instruction as instruction D in Figure 1(a). This evaluation result shows that Comp achieves the best performance for all benchmark programs among three fetch units. The FIPC of Comp is 0.816 while the FIPC of both Decomp and Baseline is 0.670 on average. The proposal of Comp achieves 21.8% better fetch performance than Decomp and Baseline.

In this work, we clarified the problem of instruction fetching which supports the RISC-V compressed instructions. Then, to solve the problem, we proposed two fetch architectures focusing on the organizations of an instruction cache. Not focusing on an instruction cache, an instruction fetch unit using *instruction buffer* has been proposed [4] as an efficient instruction fetch unit for ARM's Thumb instruction. An instruction buffer is a FIFO memory that temporarily stores instructions between instruction fetch stage and decode stage of the pipeline.

Although the proposal using compressed cache tries to fetch 4-byte data every cycle, the 2-byte of a high word is abandoned if the fetched instruction is a 2-byte one. This drawback can be mitigated when the instruction buffer is used. Regardless of 2-byte or 4-byte, the fetched data from the conventional cache will be enqueued to the instruction buffer. Then, 2-byte or 4-byte data for one instruction is dequeued from the buffer if the head instruction is 2-byte or 4-byte, respectively.

This scheme using an instruction buffer can mitigate the problem we mentioned earlier. On the other hand, this scheme requires more complex hardware of an instruction buffer than IR register of our proposal. Note that this scheme has a lot of options to be considered, and the implementation and evaluation of this scheme is out of this research. The performance evaluation of this scheme and comparison to our proposals is one of our future works.

4 CONCLUSION

We proposed two instruction fetch architectures of RISC-V soft processors that support the compressed instructions. The first one is the instruction fetch architecture with a decompressed cache. This cache always stores the 4-byte standard instructions after a 2-byte instruction is decompressed to the equivalent 4-byte one. The second one is the instruction fetch architecture with a compressed cache. This cache uses the complement word slots to solve the problem that an instruction may across two cache lines.

We evaluated these proposals and the baseline architecture in terms of instruction fetch efficiency, operating frequency, and hardware resources. Through the evaluation, we showed that the proposed unit with a compressed cache using reasonable hardware resources achieved the best performance, and this unit can fetch 0.816 instructions per cycle on average. This unit achieved 21.8% better fetch performance than the baseline unit.

ACKNOWLEDGMENTS

This work is supported by VLSI Design and Education Center (VDEC), the University of Tokyo with the collaboration with Synopsys Corporation. This work is supported by JSPS KAKENHI Grant Number JP16H02794.

REFERENCES

[1] RISC-V Foundation. 2019. RISC-V Foundation | Instruction Set Architecture (ISA). https://riscv.org/.
[2] RISC-V Foundation. 2019. Spike, a RISC-V ISA Simulator. https://github.com/riscv/riscv-isa-sim.
[3] Intel. 2018. *Nios II Processor Reference Guide.*
[4] Arvind Krishnaswamy and Rajiv Gupta. 2003. Enhancing the Performance of 16-bit Code Using Augmenting Instructions. In *Proceedings of the 2003 ACM SIGPLAN Conference on Language, Compiler, and Tool for Embedded Systems (LCTES '03).* ACM, 254–264. https://doi.org/10.1145/780732.780767
[5] Scott McFarling. 1993. *Combining branch predictors.* Technical Report. Technical Report TN-36, Digital Western Research Laboratory.
[6] David A. Patterson and John L. Hennessy. 2017. *Computer Organization and Design RISC-V Edition: The Hardware Software Interface.* Morgan Kaufmann.
[7] Standard Performance Evaluation Corporation. 1995. SPEC CINT95 Benchmarks. https://www.spec.org/cpu95/CINT95/.
[8] Andrew Waterman and Krste Asanović. 2017. *The RISC-V Instruction Set Manual Volume I: User-Level ISA, Document Version 2.2.* RISC-V Foundation. https://content.riscv.org/wp-content/uploads/2017/05/riscv-spec-v2.2.pdf
[9] Xilinx. 2018. *MicroBlaze Processor Reference Guide* (v2018.2 ed.).