코드감소와 성능향상을 위한 이질 레지스터 분할 및 명령어 구조 설계

論 文
48A-12-13

Code Size Reduction and Execution Performance Improvement with Instruction Set Architecture Design based on Non-homogeneous Register Partition

權 寧 濬* · 李 赫 宰**
(Young-Jun Kwon · Hyuk-Jae Lee)

Abstract - Embedded processors often accommodate two instruction sets, a standard instruction set and a compressed instruction set. With the compressed instruction set, code size can be reduced while instruction count (and consequently execution time) can be increased. To achieve code size reduction without significant increase of execution time, this paper proposes a new compressed instruction set architecture, called TOE (Two Operations Execution). The proposed instruction set format includes the parallel bit that indicates an instruction can be executed simultaneously with the next instruction. To add the parallel bit, TOE instruction format reduces the destination register field. The reduction of the register field limits the number of registers that are accessible by an instruction. To overcome the limited accessibility of registers, TOE adapts non-homogeneous register partition in which registers are divided into multiple subsets, each of which are accessed by different groups of instructions. With non-homogeneous registers, each instruction can access only a limited number of registers, but an entire program can access all available registers. With efficient non-homogeneous register allocator, all registers can be used in a balanced manner. As a result, the increase of code size due to register spills is negligible. Experimental results show that more than 30% of TOE instructions can be executed in parallel without significant increase of code size when compared to existing Thumb instruction set.

Key Words : Computer architecture, instruction set architecture (ISA), instruction-level parallelism, embedded system, code size reduction, non-homogeneous register partition

1. Introduction

Embedded systems often require compact code size. Otherwise, the chip area occupied by software can increase and consequently demanding more manufacturing cost. For code size reduction, commercial products, such as ARM/Thumb [10] and MIPS16 [11] processors accommodate two different instruction sets, a standard 32-bit instruction set and a compressed 16-bit instruction set. The 32-bit instruction set is a complete instruction set while 16-bit instruction is derived from the 32-bit instruction set by selecting frequently-used instructions and converting them in 16-bit instruction format. By using 16-bit instructions as many as possible, a programmer can substantially reduce the overall program size. For typical examples, the compressed code may require around 70% of the space of the original code, while using 40% more instructions [5].

* 正 會 員 : MIPS Technologies Inc., Member of Technical Staff
** 正 會 員 : Intel Corporation, Senior Engineer
接受日字 : 1999年 10月 19日
最終完了 : 1999年 11月 11日

The increase of the number of instructions can increase execution time. To avoid a significant increase of the execution time, this paper proposes a new 16-bit compressed instruction set, called TOE (Two Operations Execution). To exploit instruction level parallelism [6], TOE instruction format includes one bit that indicates the current instruction can be executed simultaneously with the next instruction. Since parallel execution is decided and explicitly specified by a compiler, hardware scheduling is not required, unlike superscalar machines [6], thus keeping hardware complexity relatively low.

The addition of the parallel bit requires the reduction of other fields, such as an opcode field or register selection fields. The reduction of an opcode field might not be desirable because it decreases the number of 16-bit instructions. If register selection fields is reduced, the number of accessible registers for a given instruction is reduced. For example, if the register selection field is reduced to two bits (from three bits in the original Thumb or MIPS16), then only four registers are accessible regardless of the number of physically available registers. This limited accessibility can reduce the number of instructions that can be convertible to

16-bit instruction format.

This limitation of register accessibility can be relieved by non-homogeneous registers [1,7] in which registers are divided into multiple subsets, and then the access of registers in each group are limited depending on the type of instructions. At compile time, register allocation [9] needs to balance the use of different groups of registers. Also, it is desirable to allocate registers before instruction scheduling so that the scheduler can identify TOE instructions. This can be achieved by the register allocation technique based on register-reuse chains [4, 3].

This paper is organized as follows. Section 2 presents the TOE instruction set architecture. Section 3 explains the register allocation for non-homogeneous partition. Section 4 shows the evaluation results and Section 4 presents the conclusions.

2. Thumb and TOE Instruction Set Architecture

ARM7T processor can execute two different instruction sets, 32-bit ARM instruction set and 16-bit Thumb instruction set [5]. The Thumb instruction set is not a complete instruction set but includes frequently-used ARM instructions. A programmer can reduce code size by converting a 32-bit ARM instruction to the corresponding 16-bit Thumb instruction whenever possible. However, it is not always possible to convert an ARM instruction to a Thumb instruction because Thumb is not a complete instruction set. In addition, the reduction of register selection fields also prevents some ARM instructions from being convertible to a Thumb instruction. Since the register selection field is reduced to 3 bits in a Thumb instruction format, each Thumb instruction can access at most eight registers (e.g. from R0 to R7). Thus, any ARM instruction that accesses beyond the scope of Thumb register selection field (e.g., R11) cannot be convertible to a Thumb instruction. Therefore, the limitation in the number of addressable registers also significantly reduces the number of 16-bit convertible instructions.

The limited accessibility can be improved by non-homogeneous registers architecture which divides registers into multiple subsets, and each subset is associated with a group of instructions which can access the registers in the subset. Thus, different subsets of registers can be accessed by different groups of instructions. Therefore, although each instruction can access only a limited number of registers, an entire program can access all registers. In addition, with a well-optimized register allocator, all registers can be used in a balanced manner.

Non-homogeneous registers allow to reduce the register selection field without affecting the accessibility of registers for an entire program. The reduction of register selection field can allow the increase of other fields, such as opcode field and immediate operand field. With the increase of the opcode field, more instructions can be included in 16-bit format, and consequently more 32-bit instructions can be convertible to 16-bit instructions. The increase of immediate operand field also increases the number of 16-bit convertible instructions.

In this paper, the saved bit due to the reduction of register selection field is used as the parallel bit as well as additional opcode field and/or immediate operand field. The parallel bit indicates that the current instruction can be executed simultaneously with the next instruction. With the parallel bit, this paper enhances the Thumb instruction set and proposes TOE (Two Operations Execution) instructions. A TOE processor can execute either 32-bit instructions or 16-bit instructions. The 32-bit instruction set is exactly the same as an ARM instruction set, while the 16-bit instruction set is modified from the Thumb instruction set in order to assist parallel execution.

As shown in Fig. 1 (a), TOE supports the instruction format with three register fields, where two serve as the operands and the other serves as the destination register. This format applies to the TOE instructions for addition (ADD) and subtraction (SUB). The opcode field of the instruction is seven bits in length. Two 3-bit operand fields can access any of the eight general-purpose registers, respectively. The 2-bit destination register field can select one register out of four registers and the remaining one bit is the parallel bit that signifies whether the TOE instruction can be executed in parallel with the next TOE instruction. As shown in Fig. 1 (b), TOE also has the capacity to support the instruction format that uses two register fields and one immediate value field. Like the three-register format, there exist a 7-bit opcode field, a parallel bit, and a 2-bit destination register field. Since only one 3-bit source register field exists, the remaining three bits of instruction represent an immediate value. As another example shown in Fig. 1 (c), for the load/store instructions (LDR/STR), the instruction format again reduces the destination register field to 2 bits, and the parallel bit is added. In summary, for most TOE instructions, one bit is reduced in the destination register field in the Thumb instruction format, and the reduced bit is used as the parallel bit.

The benefit of TOE instruction set can be explained in a case when the TOE-aware core is connected to high-speed 32-bit read-only memory (ROM). When two 16-bit TOE instructions are fetched from external memory in one fetch cycle, the two instructions can be executed together if specified as executable in parallel, and without making one of them wait in the prefetch

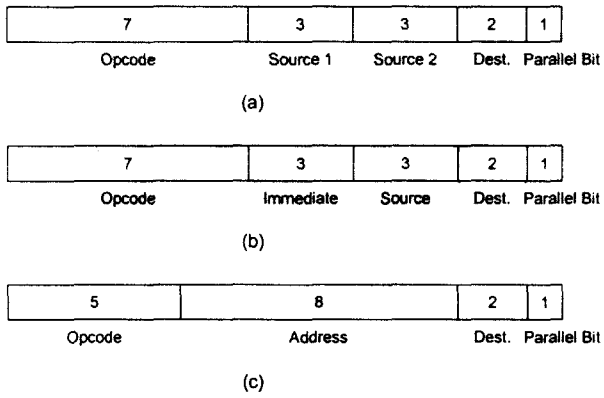

Fig. 1 Examples of TOE instruction design: (a) instruction format for three registers, (b) instruction format for two registers and one immediate, and (c) instruction format for load/store operations.

buffer, in contrast to the Thumb-aware core. Without any change in memory bandwidth, TOE might increase the performance of the system. When connected to slow, low-cost 32-bit ROM, the performance gain by TOE might be overshadowed by the slow memory fetches.

3. Compiler Support for TOE instruction set and Non-homogeneous Register Partition

The architecture based on non-homogeneous register partition [1,7] can lead to an unbalanced use of registers such that one register subset is heavily used while other subsets have unused registers. For balancing the use of different subsets of registers, an efficient register allocator needs to be implemented. For the design of an efficient non-homogeneous register allocator, it is necessary to design an efficient homogeneous register allocator first, and then extend it for non-homogeneous registers.

Register allocation [9] is an important compile-time optimization that maps variables and temporaries into either registers or memory. The goal of register allocation is to map as many variables and temporaries into registers as possible. One of the most widely used register allocation techniques is based on the graph coloring algorithm [9], and the limitation of this algorithm comes from the fact that instructions are scheduled before register allocation. When register allocation and instruction scheduling are performed as separate tasks, the optimal solution for one of the tasks can adversely affect the solution to the other task. Thus, if instruction scheduling is performed before register allocation, additional constraints can be introduced to register allocation, resulting in a non-optimal register allocation.

Recently, a new register allocation technique based on register-reuse chains was proposed [3, 4]. In this technique, registers are allocated prior to scheduling and, consequently, this allows greater freedom in register allocation optimization than the one based on graph coloring. Register allocation in [3, 4] is a procedure to decompose a dependence graph into register-reuse chains. Each register reuse chain is a linked list of nodes where each node is an assignment that represents the value of the variable that is stored in the register. All values in the same register-reuse chain are assigned to the same register. For more detailed discussions about register allocation based on register-reuse chains, refer to [3, 4, 8, 12].

Non-homogeneous register allocation can be implemented by extending homogeneous register allocation. Recall that non-homogeneous register architecture assigns a specific group of registers to each instruction. Therefore, additional constraints need to be integrated into homogeneous register allocation to guarantee that each instruction accesses a register in the proper group. The algorithm consists of two main functions, a driver function and a workhorse function. The driver function takes data dependency graphs or directed acyclic graphs (DAGs) as input. It visits each node in the DAGs in breadth first search (BFS) order. If a node in the DAGs is not visited yet, a new chain is created. Then, starting from the node, the workhorse function is called to recursively search nodes that use the same register set as the head node. The found nodes, when they are not visited yet, are attached to the chain and marked as visited. This search-and-attach process is performed until no more nodes can be attached to the current chain. The driver function stops when all nodes in DAGs are visited. For more discussions about register-reuse chain merging and merging criterion to reduce additional dependencies, refer to [12].

Fig. 2 shows the flow of register allocation and instruction scheduling for the TOE-aware processors. Note that variable register allocation is performed before instruction scheduling to identify the instructions that can be mapped to TOE instructions. Only then, the identified TOE instructions can be scheduled to be executed in

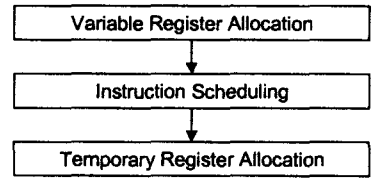

Fig. 2 The flow of register allocation and instruction scheduling for the TOE architecture.

parallel. After instruction scheduling, temporary values are assigned to registers. For a given instruction, the dedicated destination registers are given high priority to be selected.

4. Experimental Results

This section evaluates the efficiency of the TOE architecture. The first result shows the percentage of total instructions that can be mapped into TOE instructions. The second evaluation shows the number of TOE instructions that can be executed with one another TOE instruction in parallel.

Fig. 3 shows the percentage of instructions that are convertible to 16-bit instructions. For 10 benchmark programs, the numbers of instructions convertible to TOE are compared with Thumb. On average, the percentage of instructions convertible to Thumb is 59.5%, while that of TOE is 71.9%. This shows TOE leads to a significant decrease of code size when compared with Thumb. Note that, except fib (Fibonacci numbers generation), the number of instructions convertible to TOE instructions is higher than that of Thumb. The reason is that the current register partition has R6-R9 as temporary registers. If a program requires many registers and frequently uses R8 and R9 after consuming lower number registers, then the instructions using R8 and R9 are not convertible to Thumb. In contrast, some instructions, such as LDR and STR, using R8 and R9 as target destination registers, can still be counted as TOE instruction. Thus, depending on program characteristics, there can be more TOE instructions than Thumb instructions, resulting in higher encoding efficiency.

Fig. 4 shows the numbers of 16-bit convertible instructions with a different register partition. In this partition, lower-numbered registers are used as temporary registers. The percentage of instructions convertible to Thumb is 69.6% and that of TOE is 70.8%. Because lower-numbered registers are used as temporary registers, the compression ratio of Thumb is better than the results shown in Fig. 3. Note that the actual code size efficiency can be somewhat lower than these numbers, because extra mode conversion instructions, such as BX, are required to switch back and forth between ARM and Thumb/TOE modes.

Fig. 5 and 6 show the percentage of total TOE instructions that can be executed in parallel. For the register partition for the results of Fig. 3, 33.4% of TOE instructions can be executed in parallel, on average(see Fig. 5). For the register partition for the results of Fig. 4, 32.1% of TOE instructions can be executed inparallel (see Fig. 6). In the compiler used in this paper, the scope of optimization is limited by a basic block [9], while the

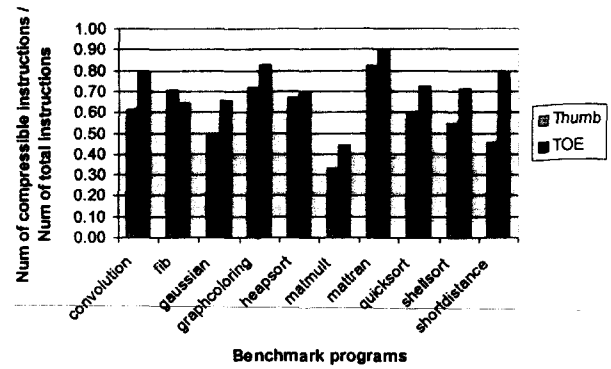

Fig. 3 Percentage of total instructions compressible into Thumb and TOE format.

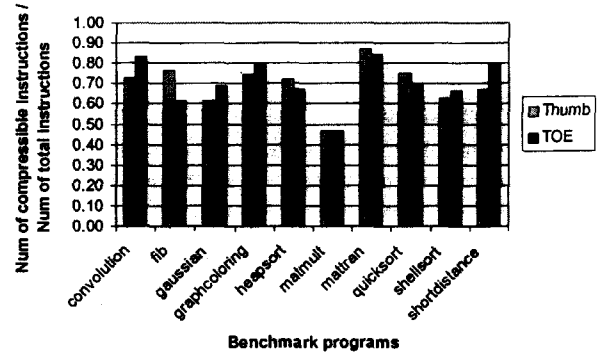

Fig. 4 Percentage of total instructions compressible into Thumb and TOE format.

scope of code generation is limited by an extended basic block [9]. For correct code generation, instructions cannot be rescheduled across the boundary of an extended basic block. Although the term extended is used in [9], the extended basic block is, in fact, not a superset of a basic block. Instead, it is often the case that a single basic block contains the boundary of extended basic blocks. Thus, the scope of the instruction scheduler is limited by both basic blocks and extended basic blocks. Consequently, the parallelism extracted by the compiler is limited. The limited scope accounts for relatively small number of parallel-executable TOE instructions. If a more efficient parallel scheduler is available, a larger number of TOE instructions can be executed in parallel, thus achieving additional speedup.

5. Conclusions

This paper proposes a new instruction set architecture, called TOE (Two Operations Execution), for embedded system. The TOE instruction format includes the parallel bit that indicates the instruction can be executed

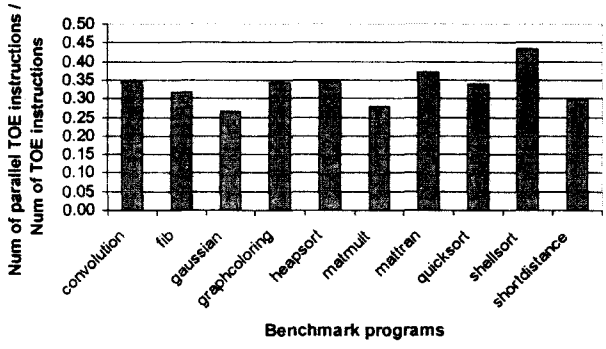

Fig. 5 Percentage of total TOE instructions executable in parallel.

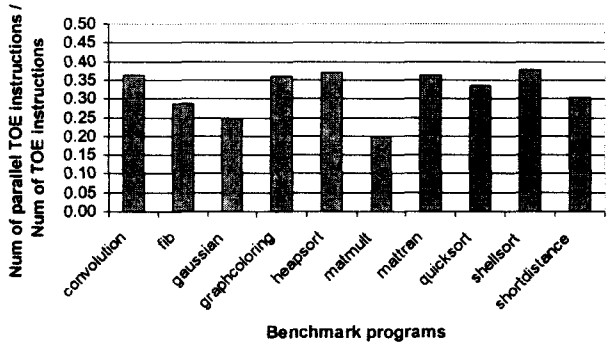

Fig. 6 Percentage of total TOE instructions executable in parallel.

simultaneously with the next instruction. More than 30% of instructions can be executed in parallel while code size is almost the same as Thumb. With more efficient instruction scheduler, the number of parallel TOE instructions can further increase. Depending on the register set partition, the number of 16-bit convertible instructions change significantly. Therefore, an efficient way of finding an optimal register partitioning needs to be developed.

참 고 문 헌

[1] G. Araujo and S. Malik, Optimal Code Generation for Embedded Memory Non-homogeneous Register Architectures, *Proc. 1995 Int'l Symp. on Systems Synthesis*, 1995, pp. 36-41.

[2] ARM, An Introduction to Thumb, ARM, 1995. http://www.arm.com

[3] D. Berson, R. Gupta, and M. Soffa, *URSA: A Unified Resource Allocator for Registers and Functional Units in VLIW Architecture*, Technical Report 92-21, Univ. of Pittsburgh, Comp. Sci. Dept., Nov. 1992.

[4] D. Berson, R. Gupta, and M. Soffa, Resource Spackling: A Framework for Integrating Register Allocation in Local and Global Schedules, *Proc. IFIP WG 10.3 Working Conference on Parallel Architectures and Compilation Techniques*, 1994.

[5] S. Furber, *ARM System Architecture*, Addison Wesley, New York, NY, 1996.

[6] J. Hennessy and D. Patterson, *Computer Architecture A Quantitative Approach*, 2nd ed., Morgan Kaufmann, San Francisco, CA, 1996.

[7] S. Liao, *Code Generation and Optimization for Embedded Digital Signal Processors*, Ph.D. Thesis, MIT Dept. of EECS, Jan. 22, 1996.

[8] X. Ma, *Non-homogeneous Register Allocation for Embedded Processors*, M.S. Thesis, Louisiana Tech University, 1998.

[9] S. Muchnick, *Advanced Compiler Design Implementation*, Morgan Kaufmann, San Francisco, CA, 1997.

[10] S. Segars, K. Clarke, and L. Goudge, Embedded Control Problems, Thumb, and the ARM7TDMI, *IEEE Micro*, vol. 15, no. 5, October 1995, pp. 22-30.

[11] D. Sweetman, *See MIPS Run*, Morgan Kaufmann, San Francisco, CA, 1999.

[12] Y. Zhang, *A Systematic Integration of Register Allocation and Instruction Scheduling*, Ph.D. Dissertation, Louisiana Tech University, 1999.

저 자 소 개

권 영 준(權 寧 濬)

1965년 6월 17일 생. 1988년 서울대 전기공학과 졸업. 1996년 Texas A&M 대학 전산학과 졸업(공박). 1997년~현재 MIPS Technologies Inc. Member of Technical Staff.
E-mail : kwon@mips.com

이 혁 재(李 赫 宰)

1965년 2월 8일 생. 1987년 서울대 전자공학과 졸업. 1996년 Purdue 대학교 전기 및 컴퓨터공학과 졸업(공박). 1996년-1998년 Louisiana 공과대학교 전산학과 조교수. 1998년~현재 Intel Senior Engineer.
E-mail : jasonl@ichips.intel.com