Microprocessors and Microsystems 51 (2017) 8–17

Contents lists available at ScienceDirect

Microprocessors and Microsystems

journal homepage: www.elsevier.com/locate/micpro

Floating accumulator architecture

Yuan-Shin Hwang[a,*], Wei-Che Hsu[b]

[a] Department of Computer Science and Information Engineering, National Taiwan University of Science and Technology, Taipei 106, Taiwan
[b] MediaTek Inc., Science-Based Industrial Park, Hsinchu 300, Taiwan

ARTICLE INFO

Article history:
Received 4 May 2016
Revised 16 March 2017
Accepted 16 April 2017
Available online 17 April 2017

Keywords:
Instruction set architecture
Accumulator architecture
Instruction format
Compiler

ABSTRACT

Although technology advancement can pack more and more physical registers in processors, the numbers of architectural registers defined by the instruction set architectures (ISAs) remain relatively small on most modern processors. Exposing more architectural registers to compilers and programmers can improve the effectiveness of compiler optimization and the quality of code. However, increasing the number of architectural registers by simply adding extra bits to the register fields of instructions will expand the code size. Therefore, a better way of exposing more ISA registers without significantly expanding the code size is needed. This paper presents a new ISA called *Floating Accumulator Architecture* (*FAA*) that can expand the number of ISA registers without increasing the instruction length. Unlike the accumulator architecture whose accumulator is a fixed, special register, FAA dynamically chooses a register from the general-purpose register file as the accumulator. In other words, the accumulator in FAA is an alias to some register in the register file at any instruction, and the alias relation can be dynamically updated by FAA at any program points. Since the accumulator implicitly stores the result, the destination register field can be omitted from FAA instructions, resulting in a saving of 3 to 5 bits for each instruction. This new free instruction bit space can be utilized in two possible ways: doubling the number of ISA registers of modern 32-bit RISC processors or maintaining the number of ISA registers for 16-bit instructions on embedded processors. This paper presents the result of utilizing the free bit space to double the number of ISA registers from 16 to 32 on ARM processors, and experimental results show that performance can be improved by 7.6% on average for MediaBench benchmarks.

1. Introduction

Technology advancement has made it easy for modern processors to pack large physical register files. As more physical registers available to store variables close to the pipeline, performance can be improved. However, the numbers of architectural registers defined by the instruction set architectures (ISAs) remain relatively small on most modern processors, ranging from 8 registers on IA-32 to 32 registers on most RISC processors. Exposing more architectural registers to compilers and programmers can improve the effectiveness of compiler optimization and the quality of code. Unfortunately, increasing the number of architectural registers by simply adding extra bits to the register fields of instructions will expand the code size, as adding one bit in the register field typically causes the instruction size to grow by 2 or more bits due to multiple register fields in the instruction. As a result, modern processors generally contain higher number of physical registers than those exposed in the ISA.

Instruction encoding space is more restricted on embedded processors, as many of them provide “reduced bit-width” instruction sets which encode the most commonly used instructions using fewer bits [4,8,22]. Even if the hardware can support more registers, the number of architectural registers defined by ISA is much smaller due to the encoding issue. A very good example is the ARM processor with a 32-bit instruction set and a 16-bit instruction called the Thumb [19]. Due to the smaller encoding space, most THUMB instructions can only access 8 registers although all 16 registers are physically present and can contain values. Similarly, the MIPS16 embedded processor also supports such dual instruction set feature, and its instructions can access only 8 registers out of 32 general-purpose registers seen by MIPS32 instructions [20]. The 16-bit format is shown to have significant cost-performance advantages over the 32-bit format under typical memory system performance constraints [8,22]. However, the compromises made in designing the Thumb or MIPS16 instruction set leads to significantly increased instruction counts [14].

Restricting the number of architectural registers typically incurs performance penalty since compilers and programmers can only utilize the exposed architectural registers. There have been several

* Corresponding author.
E-mail address: shin@csie.ntust.edu.tw (Y.-S. Hwang).

http://dx.doi.org/10.1016/j.micpro.2017.04.007
0141-9331/

approaches proposed to increase the number architectural registers without expanding the code size significantly. For instance, register windows have been designed to provide more registers than allowed in the encoding [18,21]. Differential encoding is a new register encoding scheme that allows more registers to be addressed in the operand field of instructions than the direct encoding currently being used [22]. Hardware managed register allocation schemes have even be developed to allocate more physical registers at runtime [23]. Some new instructions have been introduced into the instruction sets that can make use of all registers in all instructions by changing the visible subset of registers at any program point [14]. Another possible approach is to find some underutilized fields in instructions to represent more registers. It might be beneficial to trade conditional execution for more registers on ARM processors, as every ARM instruction carries a 4-bit condition field in that specifies the predicate of conditional execution but ratios of conditionalized instructions are generally very low [5].

This paper presents a new ISA called *floating accumulator architecture* (*FAA*) that can expand the number of ISA registers without widening the instructions. Similar to the traditional accumulator architecture, FAA reduces the instruction width by making the accumulator as the default destination of instructions. However, unlike the accumulator architecture whose accumulator is a fixed, special register, FAA dynamically chooses a register from the general-purpose register file as the accumulator. In other words, the accumulator in FAA can be viewed simply as an alias to some general-purpose register at any instruction, and the alias relation can be dynamically updated by FAA at any program points. Since the accumulator implicitly stores the result, the destination register field can be omitted from FAA instructions, resulting in a saving of 3 to 5 bits for each instruction. There could be two possible ways to utilize the free bit space: quadrupling the number of ISA registers of modern 32-bit RISC processors and maintaining the number of ISA registers for 16-bit instructions on embedded processors. This paper presents an LLVM [15] implementation that utilizes the free bit space to double the number of ISA registers on ARM processors, and experimental results shows that performance can be improved by 7.6% on average for MediaBench benchmarks when the number of ISA registers is extended from 16 to 32.

The main results of this paper are as follows:

- FAA can generally shorten instructions by 3 to 5 bits when comparing to the general-purpose register (GPR) architectures with the same register file size.
- FAA can double the number of ISA registers of modern 32-bit RISC processors.
- FAA can avoid reducing the number of ISA registers for 16-bit instructions on embedded processors.
- The microarchitecture for FAA is very similar to that of the GPR architecture, as the accumulator in FAA can be viewed simply as an alias to some general-purpose register at any instruction. Therefore, it is very easy to implement FAA based on the GPR microarchitecture.

The rest of this paper is organized as follows. Section 2 briefly recaps some well-known instruction set architectures. Section 3 introduces the new ISA and presents the microarchitecture that supports FAA. Section 4 presents an applications of FAA to double ARM registers, and Section 5 shows the experimental results. Section 6 surveys the related work, and Section 7 concludes this paper.

2. Instruction set architectures (ISAs)

ISAs can be generally classified into the following three types

- Stack architecture
- Accumulator architecture, and
- General-purpose register architecture

based on the type of internal storage in the processor [7,10]. A stack architecture uses a operand stack to execute the instruction, and the operands are implicitly on the top of the stack, as shown in Fig. 1(a). An accumulator architecture has a special register called the accumulator, which implicitly stores one operand and the result, while the other operand comes from the memory, as shown in Fig. 1(b). Both the stack and accumulator architectures have good code density, since implicit operands do not occupy any bit fields in instructions. Although most early computers used stack or accumulator architectures, almost all the new architectures nowadays use the general-purpose register (GPR) architecture as registers are faster than memory and can be used effectively and efficiently. Since there are multiple registers to hold the result and operands, both the destination and operand registers must be explicitly addressed by register fields of instructions in the GPR architecture, as depicted in Fig. 1(c). Consequently, the GRP architecture has the worst code size among the three, as the length of instructions is generally the greatest.

3. Floating accumulator architecture (FAA)

The key idea of the floating accumulator architecture (FAA) is to assign one of the general-purpose registers as the accumulator, which will in turn be the implicit destination register of an instruction. For any ALU operation, e.g. Ri = Rj op Rk, Ri must be designated as the accumulator (i.e. A ≡ Ri) and hence the operation will be in fact represented by the instruction A = Rj op Rk. Since the accumulator is implicitly addressed, it does not need to occupy any register field in the instruction, as illustrated in Fig. 2. Specifically, FAA can use the two-address instruction format to express instructions that are commonly represented by the three-address instruction format in GPR architecture, and hence FAA can generally save the space of a register field (usually 3 to 5 bits) in every instruction than the GPR architecture. As a result, FAA can have better code density than the GRP architecture, while still enjoying the advantages of accessing multiple general-purpose registers.

In contrast to the accumulator architecture, the accumulator of FAA is not a fix, special-purpose register. Any general-purpose register in the register file can be dynamically designated as the accumulator. If the value of the current accumulator can not be overwritten, the value can be easily retained in the register by assigning another register in the register file as the new accumulator. This feature avoids the drawback of the accumulator architecture that has to frequently copy data in and out of the accumulator.

The general form of the instructions in FAA is

A = Rj op Rk

which performs the specified operations on the values in registers Rj and Rk and then stores the result in the current accumulator A. When a new accumulator is needed, it can be assigned by the following accumulator assignment statement:

Rj = Rj op Rk

This statement performs an operations on the values of registers Rj and Rk, and then stores the result in Rj. In addition, it assigns Rj as the accumulator A. The previous accumulator can now be referenced through the notion B or its original register number.

Fig. 3 is a sequence of code from SPEC benchmark 164.gzip, along with the compiled Alpha assembly code and its equivalent register transfer notation [13]. It is one of the more frequently executed parts of the program. The corresponding FAA code that is displayed in Fig. 3(d) shows that FAA can express the code with almost the same number of instructions as Alpha. Only the first