Exploiting Forwarding to Improve Data Bandwidth of Instruction-Set Extensions

Ramkumar Jayaseelan , Haibin Liu , Tulika Mitra
School of Computing, National University of Singapore
{ramkumar,liuhb,tulika}@comp.nus.edu.sg

ABSTRACT

Application-specific instruction-set extensions (custom instructions) help embedded processors achieve higher performance. Most custom instructions offering significant performance benefit require multiple input operands. Unfortunately, RISC-style embedded processors are designed to support at most two input operands per instruction. This data bandwidth problem is due to the limited number of read ports in the register file per instruction as well as the fixed-length instruction encoding. We propose to overcome this restriction by exploiting the data forwarding feature present in processor pipelines. With minimal modifications to the pipeline and the instruction encoding along with cooperation from the compiler, we can supply up to two additional input operands per custom instruction. Experimental results indicate that our approach achieves 87–100% of the ideal performance limit for standard benchmark programs. Additionally, our scheme saves 25% energy on an average by avoiding unnecessary accesses to the register file.

Categories and Subject Descriptors

C.3 [**Special-purpose and application-based systems**]: Real-time and embedded systems

General Terms

Performance, Design

Keywords

Instruction-set Extensions, Data Forwarding

1. INTRODUCTION

Application-specific instruction-set extensions, also called custom instructions, extend the instruction-set architecture of a base processor [6, 7, 9]. Processors that allow such extensibility have become popular as they strike the right balance between challenging performance requirement and short time-to-market constraints of embedded systems design. Custom instructions encapsulate the frequently occurring computation patterns in an application. They are implemented as custom functional units (CFU) in the datapath of an existing processor core. CFUs improve performance through parallelization and chaining of operations. Thus custom instructions help simple embedded processors achieve considerable performance and energy efficiency.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
DAC 2006, July 24–28, 2006, San Francisco, California, USA.
Copyright 2006 ACM 1-59593-381-6/06/0007 ...$5.00.

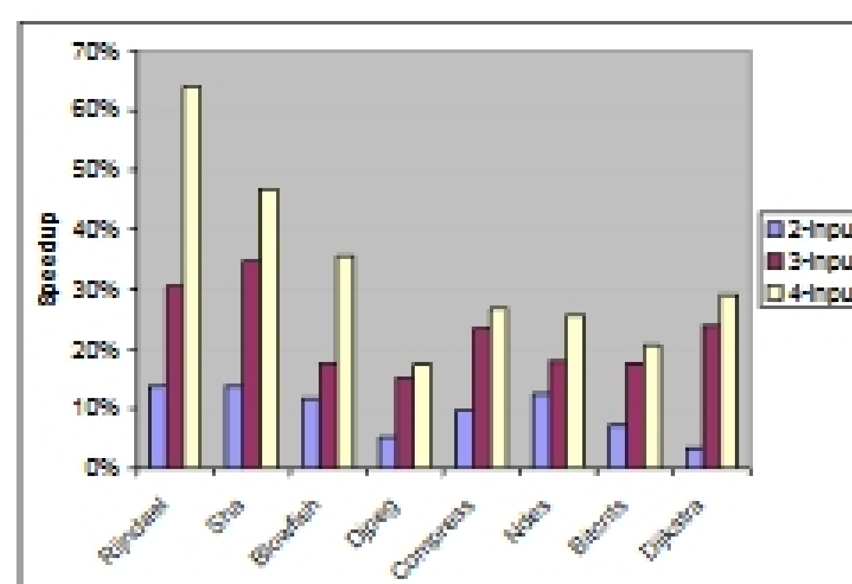

Figure 1: Impact of limited input operands on performance speedup of custom instructions.

Commercial embedded processor supporting instruction-set extensibility, such as Altera Nios-II [6] and Tensilica Xtensa [7], are all RISC-style cores with simple instructions and fixed-length instruction encoding formats. However, custom instructions typically encapsulate quite complex computations. This results in a fundamental mismatch between the base processor core and the new extensions both in terms of ISA definition and micro-architecture. Simple RISC-style instructions use at most two register input operands and one register output operand. As a result, at the micro-architectural level, the base processor core supports two register read ports per instruction. Unfortunately, multiple studies [3, 15] have shown that custom instructions generally require more than two input operands to achieve any significant performance gain. Figure 1 plots the speedup due to custom instructions with at most 2, 3, and 4 input operands, respectively[1]. Clearly, performance drops significantly as we restrict the number of input operands per custom instruction.

In this paper, we present a novel scheme that exploits the forwarding logic in processor pipeline to overcome the data bandwidth limit per custom instruction. *Data forwarding*,

[1]Details of the experimental setup are given in Section 5

also known as *register bypassing*, is a standard architectural method to supply data to a functional unit from internal pipeline buffers rather than from programmer-visible registers. In conventional processors, forwarding is used to resolve data hazard between two in-flight instructions. We observe that, in many cases, at least some of the input operands of a custom instruction are available from the data forwarding logic. Thus, we leverage on the data forwarding logic to provide additional inputs to the custom instructions.

The key to exploiting such a scheme is of course a compile time check to determine if a specific operand can indeed be obtained from the forwarding logic. However, in the presence of statically unpredictable events, such as cache miss for a custom instruction, this cannot be guaranteed at compile time. As the custom instruction gets delayed, the instruction supplying the operand may complete execution and leave the pipeline. Therefore, the operand is no longer available from the forwarding logic. To circumvent this problem, we propose minimal changes in the pipeline control hardware to guarantee the availability of the operand from the forwarding logic under such scenarios. At the same time, we ensure that the changes do not have any negative impact on the instruction throughput of the pipeline.

Finally, we need to address the related problem of instruction encoding to support additional operands. Assuming an instruction format similar to the Altera Nios-II processor [6], we show that minimal modification to the encoding scheme can support up to 64 custom instructions each having up to 4 input operands.

2. RELATED WORK

Significant research effort has been invested in issues related to instruction-set extensions for the past few years. Most of this effort has concentrated on the so called "design space exploration" problem to choose an appropriate set of custom instructions for an application [1, 3, 15, 16]. The first step of this exploration process identifies a large set of candidate patterns from the program's dataflow graph and their frequencies via profiling. Given this library of patterns, the second step selects a subset to maximize the performance under constraints on the number of allowed custom instructions and/or the area budget.

Limited data bandwidth is one of the key problems in the implementation of custom instructions. This problem arises because custom instructions normally require more than two input operands whereas the register file provides only two read ports per instruction. Increasing the number of read ports to the register file is not an attractive option as the area and power consumption grow cubically with the number of ports.

The Nios-II processor [6] solves this problem by allowing the custom functional unit (CFU) to read/update either the architectural register file or an internal register file. However, additional cycles are wasted to move the operands between the architectural register file and the internal register file through explicit `MOV` instructions. Similarly, the MicroBlaze processor [9] from Xilinx provides dedicated Fast Simplex Link (FSL) channels to move operands to the CFU. It provides `put` and `get` instructions to transfer operands between the architectural register file and the CFU through FSL channels.

Cong et al. [4, 5] eliminate these explicit transfer of operands with the help of a shadow register file associated with the CFU. Shadow register file is similar to the internal register file of Nios-II in that they both provide input operands to the CFUs. However, the major difference is that the shadow registers are updated by normal instructions during the write back stage. An additional bit in the instruction encoding decides whether the instruction should write to the shadow register file in addition to the architectural register.

Pozzi et al. [13] suggest an orthogonal approach to relax the register file port constraints. Their technique exploits the fact that for a CFU with pipelined datapath, all the operands may not be required in the first clock cycle. Therefore, the register accesses by the CFU datapath can be distributed over multiple cycles. This approach will have limited performance benefit if most custom instructions with multiple operands require single-cycle datapath.

Though the previous work [4, 5, 13] improve the data bandwidth, they do not address the related problems of encoding multiple operands in a fixed-length instruction format and data hazards.

- Fixed-length and fixed-position encoding employed in RISC processors do not provide enough space to encode the additional operands of the custom instructions. Previous works do not discuss the issue of encoding operands for custom instructions. For example, the work by Pozzi et al. [13] still requires a register identifier corresponding to each input operand of a custom instruction. Similarly, the work based on shadow register files requires either the shadow register identifier or an architectural register identifier for each input operand of a custom instruction.

- Data hazards occur in a pipeline when the dependent instruction reads the register before the source instruction writes into it. These are resolved by employing data forwarding as discussed in Section 1. For a multiple-operand custom instruction, data hazards can occur on any of the input operands. It is not clear how data hazards are handled for the additional operands in case of multi-cycle register reads [13] or shadow registers [5].

Our work addresses both of these important issues. In addition, our method avoids unnecessary register accesses and thereby saves energy (see Section 5).

3. PROPOSED ARCHITECTURE

Our proposed architecture exploits data forwarding logic in the processor pipeline to supply additional operands per custom instruction. In addition, we require minimal modification of the instruction encoding to specify the additional operands per custom instruction. In this section, we describe these modifications in the processor pipeline and the instruction encoding. We assume a RISC-style in-order pipeline that is prevalent in embedded processor architectures with extensibility feature. For illustration purposes, we use a simple MIPS-like 5-stage pipeline. However, our technique can be easily applied to other in-order pipelines. We begin with a brief review of the data forwarding logic as it is central to our discussion.

3.1 Data Forwarding

We will illustrate our technique through a simple, 5-stage, MIPS-style pipeline shown in Figure 3. The five pipeline

```
ADD   R1,  R2, R3
SUB   R4,  R1, R8
OR    R7,  R8, R9
CUST  R10, R4, R7
```

Clock	IF	ID	EX	MEM	WB
1	ADD				
2	SUB	ADD			
3	OR	SUB	ADD		
4	CUST	OR	SUB	ADD	
5	..	CUST	OR	SUB	ADD
6	..	..	CUST	OR	SUB

Figure 2: Illustration of data forwarding for a sequence of instructions.

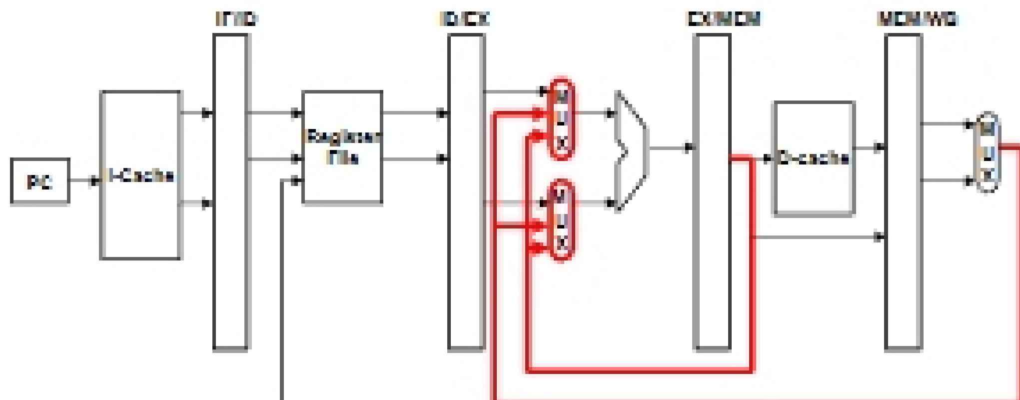

Figure 3: Data forwarding in a pipeline.

stages are: instruction fetch (`IF`), instruction decode/register read (`ID`), execute (`EX`), memory access (`MEM`) and write-back (`WB`). *Data forwarding* or *register bypassing* is a common technique used to reduce the impact of data hazards in pipelines. Consider the execution of the sequence of instructions shown in Figure 2 in a MIPS pipeline (the first register identifier of each instruction specifies the destination operand and the other two specify the source operands). There is a dependency between the `ADD` instruction and the `SUB` instruction through register `R1`. The `ADD` instruction writes the result into the register file in clock cycle 5. However, the `SUB` instruction reads the register file in clock cycle 3 and hence would read a wrong value. This is known as data hazard in the pipeline. To prevent data hazard, we can stall the pipeline for two clock cycles till the `ADD` instruction writes register `R1`. This would result in significant performance degradation. A more efficient method is to forward the result of the `ADD` instruction to the input of the functional unit before it has been written to the register file. This is based on the observation that the `SUB` instruction requires the input only in clock cycle 4 and the `ADD` instruction produces the result at the end of clock cycle 3. Thus, forwarding avoids pipeline stalls due to data hazards.

Figure 3 shows the pipeline with the data forwarding logic darkened. Forwarding paths are provided from the `EX` stage (the latch `EX/MEM`) and the `MEM` stage (the latch `MEM/WB`) to the functional units. Multiplexers are placed before the functional unit to select the operand either from the register file or from the forwarding paths. Note that there is no forwarding path from the output of the `WB` stage. The hazards in this stage are handled by ensuring that the register writes happen in the first half of a clock cycle and the register reads happen in the second half of a clock cycle. Interested readers can refer to [12] for further details.

We observe that in most cases, the operands of custom instructions are available from the forwarding paths. In Figure 2, the custom instruction `CUST` reads both its input operands from the forwarding path. Hence, the forwarding path can be used as a proxy to cover up for the lack of number of read ports in the register file. The two latches (`EX/MEM` and `MEM/WB`) can provide up to two additional input operands for a custom instruction (the other two come from the register file). Note that in a conventional pipeline, an instruction reads from the register file in the `ID` stage even if it later uses the data from the forwarding logic. In contrast, we do not allow a custom instruction to read from the register file if the corresponding operand will be supplied from the forwarding path. The challenge now is to identify at compile time which operands will be available from the forwarding logic, encoding that information in the instruction, and ensuring that the operand is available even in the presence of unpredictable events (e.g., instruction cache miss).

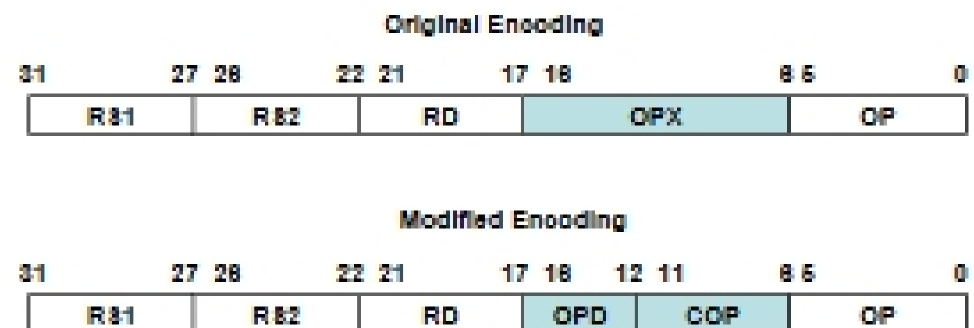

Figure 4: Encoding format of custom instructions.

3.2 Instruction Encoding

We now describe the instruction encoding in the presence of custom instructions that exploit forwarding logic to obtain up to two additional input operands. The basic idea behind our encoding is *not* to affect the decoding of normal instructions. We also try to minimize the number of bits required to encode the operand information. We illustrate our encoding with the instruction format of Nios-II processor. However, the general idea is applicable to any RISC-style instruction format.

The original encoding in Figure 4 is the format for custom instructions in Nios-II. It consists of a 6-bit opcode field `OP`, which is fixed at 0x32 for all custom instructions. The 11-bit opcode extension field `OPX` is used to distinguish different custom instructions. 3-bits from the `OPX` field is used in Nios-II to indicate whether each source/destination register refers to the architectural or the internal register file (see Section 2). The rest of the 15-bits are used to specify the two source and one destination operands.

As we do not want to affect the encoding of normal instructions, all the information about the operands of the custom instructions are encoded as part of the 11-bit opcode extension field `OPX`. Each operand of a CFU can come either from the two register ports or from one of the two forwarding paths. However, the number of input operands of a custom instruction need not be encoded as the datapath of the CFU can ignore the extra inputs. For example, a 3-input custom operation would ignore the fourth operand.

Among the four input operands, at most two operands are specified using the forwarding path. There are $C_2^4 = 6$ possibilities for the choice of these two operands among the four input operands. In addition, for each of the operands from the forwarding path, we need to specify whether it comes from the `EX/MEM` latch or the `MEM/WB` latch. There are a total of four possibilities in this case and hence the total