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.

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.

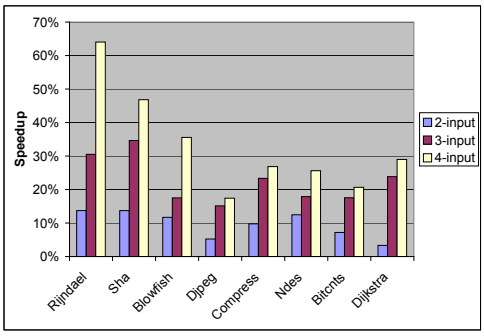

Figure 1: Impact of limited input operands on performance speedup of custom instructions.

Commercial embedded processors 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, 14] 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*, 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

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

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, 14, 15]. 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. [12] 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 works [4, 5, 12] 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. [12] 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 [12] 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

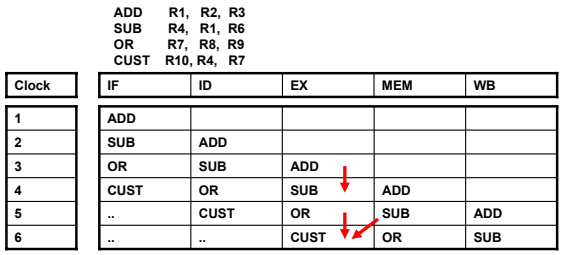

Figure 2: Illustration of data forwarding for a sequence of instructions.

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 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 highlighted. 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

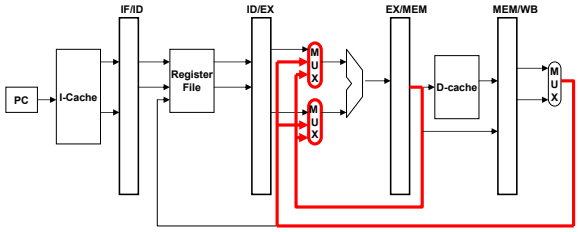

Figure 3: Data forwarding in a pipeline.

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 [11] 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).

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

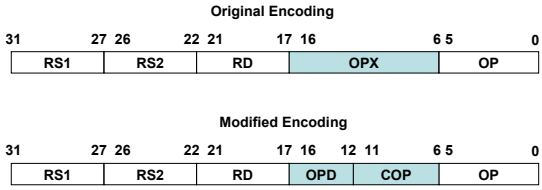

Figure 4: Encoding format of custom instructions.

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 number of possibilities that need to be encoded is 24, i.e., we require 5 bits to encode the information. The modified encoding in Figure 4 shows the new instruction format with the operand information. 5 bits from the `OPX` field are used to encode the operand information (`OPD` field). The remaining 6 bits (`COP` field) can be used to specify the custom function to be performed. Thus there can be 64 distinct custom instructions each having up to four input operands. Note that decoding the operand information is done in parallel to the instruction decoding in the `ID` stage of the pipeline and hence would not affect the cycle time.

3.3 Predictable Forwarding

The key to exploiting forwarding for custom instructions is to determine at compile time (i.e., statically) whether an operand can be obtained from the forwarding path. In Figure 3 there are two forwarding paths. Let us assume a sequence of instructions $\langle I_1, I_2, \ldots I_n \rangle$. An operand of instruction I_i is available from the forwarding path only if instruction I_{i-1} or I_{i-2} produces the operand. For example in Figure

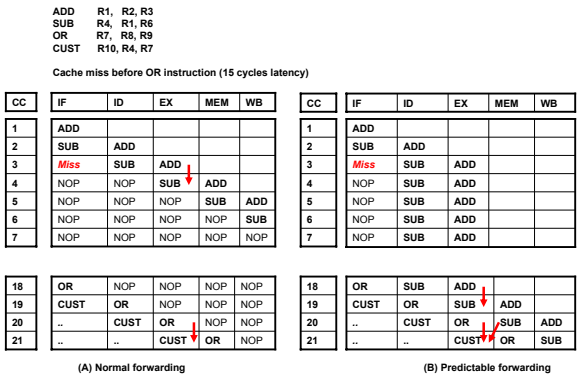

Figure 5: Pipeline behavior for I-cache miss

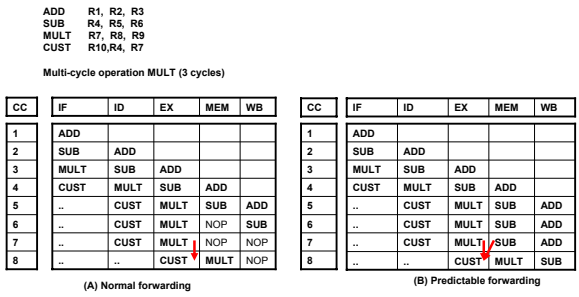

Figure 6: Pipeline for multi-cycle operation.

2, the operands of the custom instruction CUST are available from the forwarding paths as they are produced by the two immediate predecessors (SUB and OR instructions).

However this property does not hold when there are multi-cycle operations and also in the event of instruction cache misses. This is because these events introduce bubbles in the pipeline and hence affect forwarding between dependent instructions. Figure 5(A) illustrates the problem in the event of an instruction cache miss. The OR instruction misses in the instruction cache and hence the custom instruction CUST cannot obtain the result of the SUB instruction (register R4) from the forwarding path. This is not a problem for conventional pipeline because normal instructions will simply read the result from the register file (it was relying on data forwarding only when the result has not yet been written to the register). Unfortunately, the custom instruction *has to* read the data from the forwarding path, i.e., it does not have the fall back option of reading from the register file. Similarly, multi-cycle operations can affect the forwarding path in the pipeline. Figure 6(A) shows the pipeline behavior when a multi-cycle instruction MULT is executed in the pipeline.

Data cache misses and branch mispredictions occur in the MEM stage and hence only the instruction in the WB stage will not be able to forward the result. As we do not assume any forwarding from the WB stage anyway (it is taken care of by split register read/write as discussed in Section 3.1), branch misprediction and data cache misses do not affect our forwarding path.

We suggest a simple change in the pipeline control logic to guarantee forwarding between instruction I_{i-1}/I_{i-2} and instruction I_i in the event of instruction cache misses and multi-cycle operations. Events such as cache misses create bubbles in the pipeline draining out instructions in later stages of the pipeline. This affects forwarding. This can be clearly seen by comparing Figure 2 with Figure 5(A). In Figure 2, the SUB instruction is in the pipeline when the custom instruction enters the EX stage. However in Figure 5(A), due to the instruction cache miss the SUB instruction leaves the pipeline before the custom instruction enters the EX stage. Therefore the value cannot be forwarded; instead it should be read from the register file in the ID stage.

The key insight is to stall the instructions in the later stages of the pipeline (after the event) rather than allowing them to progress. That is, instead of introducing NOPs into the pipeline (as shown in Figure 5(A)), we retain the contents of the later stages for the duration of the I-cache miss. This way when the normal flow resumes, the pipeline looks like as if the event did not happen at all. This scenario is shown in Figure 5(B). Similarly, for multi-cycle operations the effect on the pipeline execution is shown in Figure 6(B). Note that stalling the pipeline stages as opposed to introducing bubbles (NOPs) does not

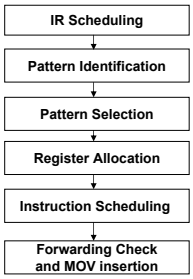

Figure 7: Compilation toolchain

cause any additional performance degradation in terms of instruction throughput.

To achieve this, we simply need stall signal for each pipeline latch. When the stall signal is set, the latch holds its current value. A stall unit is responsible for stalling the pipeline during cache misses and multi-cycle operations. To ensure forwarding, the stall signals for latches in the later stages of the pipeline must be set (`ID/EX`, `EX/MEM` and `MEM/WB`) for the duration of the cache miss. In a similar fashion, the stall signals for the `EX/MEM` and `MEM/WB` latch must be set for the duration of the multi-cycle operation.

4 Compilation Toolchain

As mentioned before, our technique requires cooperation from the compiler. We need to determine at compile time whether a specific operand can be forwarded and encode the custom instruction accordingly. In addition, the compiler can schedule the instructions appropriately so as to maximize the opportunity of forwarding. We now describe how these concerns are addressed in the compilation toolchain for custom instruction selection and exploitation.

The relevant portion of the compilation toolchain is shown in Figure 7. Pattern identification is performed at the intermediate representation (IR) level just prior to register allocation and after the scheduling of the intermediate instructions. We use the pattern identification scheme discussed in [15] that involves construction of the data dependency graphs for each basic block followed by identification of all possible patterns that satisfy the given constraints. In our case, we impose a constraint that the patterns should have at most 4 input operands and one output operand. This is followed by the selection of a subset of patterns to be implement as custom instructions. We now describe the pattern selection phase in detail.

4.1 Pattern Selection

We use a heuristic pattern selection method. Given the set of identified patterns, we first club together identical subgraphs using the algorithm presented in [10]. All the identical subgraphs map to a single custom instruction and are called the instances of a pattern. Associated with each pattern instance, we have an execution count (obtained through profiling) and the speedup. A greedy heuristic method is employed for pattern selection [14]. It attempts to cover each original instruction in the code with zero or one custom instructions using a priority function given by

$$Priority_{i.j} = speedup_{i.j} \times frequency_{i.j}$$

where $Priority_{i.j}$, $speedup_{i.j}$, and $frequency_{i.j}$ are the priority, performance speedup and execution frequency of the j^{th} instance of pattern i. The pattern instances are chosen starting with the highest priority one.

In our forwarding-based approach, the performance speedup of a pattern instance depends on how many of its input operands can be forwarded. Suppose we have a 4-input custom instruction. Two of its operands can be obtained from the forwarding path and two are read from the register file. Then we can easily encode that custom instruction. However, if we cannot obtain any operand from the forwarding path, then we need to add additional `MOV` instructions in the code. Let us suppose the custom instruction needs three input operands `R2, R3, R4`. `R2, R3` can be read from the register file. For `R4`, we insert a redundant instruction `MOV R4, R4` just before the custom instruction. This ensures that the operand `R4` can be obtained from the forwarding path. The latency of a `MOV` instruction is one clock cycle. Accordingly, we update the performance speedup of all the custom instruction instances. Notice that this is a conservative estimate; a `MOV` instruction might not be required after the instruction scheduling discussed in the next subsection.

4.2 Instruction Scheduling

The speedup of a custom instruction depends heavily on the final instruction scheduling (after register allocation) due to the forwarding constraint. Given a basic block with custom instructions, we have formulated the problem of finding the optimal schedule with forwarding as an integer linear programming (ILP) problem. Note that we are considering an in-order pipeline and we do not modify the register allocation.

Let B be a basic block consisting of both normal instructions and custom instructions. If a custom instruction C is dependent on another instruction I for one of its operands, then the distance between C

and I determines whether the operand can be obtained from the forwarding paths. If more than two input operands of custom instruction C are not available from the forwarding paths, then we need to introduce redundant `MOV` instructions as discussed in the previous subsection. These additional `MOV` instructions increase the execution time and hence we would like to eliminate as many `MOV` instructions as possible. The optimal schedule is an ordering of the instructions in B that results in the minimal number of `MOV` instructions.

For a basic block B, let $I_1 \dots I_N$ and $C_1 \dots C_M$ be the normal instructions and custom instructions, respectively. Let $input(C_i)$ be the number of *unique* input operands for custom instruction C_i. For example, a custom instruction `CUST R1, R2, R3, R2, R4` has 3 unique input operands corresponding to registers `R2`, `R3`, `R4` (the output operand is `R1`). Let $MC_i^1 \dots MC_i^{input(C_i)}$ represent the `MOV` instructions associated with the unique input operands of custom instruction C_i. Finally, let $D_i^1 \dots D_i^{input(C_i)}$ represent the instructions that produce the corresponding input operands of C_i. That is, C_i is dependent on $D_i^1 \dots D_i^{input(C_i)}$.

The total number of instructions in basic block B can be bounded by

$$S_{max} = N + M + \sum_{i=1}^{M} input(C_i) \tag{1}$$

A map function maps each instruction to a unique identifier between 1 and S_{max}.

Scheduling Constraints We introduce binary variables $X_{i,j}$ ($1 \le i \le S_{max}, 1 \le j \le S_{max}$) to represent the instruction ordering where

$$X_{i,j} = \begin{cases} 1 & \text{if instruction with identifier } i \text{ occupies the } j^{th} \text{ position} \\ 0 & \text{otherwise} \end{cases}$$

Notice that the instruction schedule considers all instructions, i.e., normal instructions, custom instructions as well as `MOV` instructions. Each normal/custom instruction can occupy exactly one position.

$$\sum_{j=1}^{S_{max}} X_{map(I_i),j} = 1 \;\; \forall i \;\; (1 \le i \le N) \tag{2}$$

$$\sum_{j=1}^{S_{max}} X_{map(C_i),j} = 1 \;\; \forall i \;\; (1 \le i \le M) \tag{3}$$

However, the `MOV` instructions may or may not be scheduled depending on the forwarding; therefore

$$\sum_{j=1}^{S_{max}} X_{map(MC_i^k),j} \le 1 \quad \forall i,k \;\; (1 \le i \le M, \;\; 1 \le k \le input(C_i)) \tag{4}$$

Moreover, each position can have at most one instruction.

$$\sum_{i=1}^{S_{max}} X_{i,j} \leq 1 \ \ \forall j \ \ (1 \leq j \leq S_{max}) \tag{5}$$

Before we introduce additional constraints, let us define two new functions: $position(i)$ denoting the position of the instruction with identifier i and $distance(i', i)$ denoting the distance between instructions with identifiers i' and i.

$$position(i) = \sum_{j=1}^{S_{max}} j \times X_{i,j}$$

$$distance(i', i) = position(i) - position(i')$$

Dependency Constraints: A Read-after-Write dependency occurs between i' and i (RAW(i',i)) if instruction i reads a result written by instruction i'. As a result of this data dependency, instruction i must be scheduled after instruction i'. This is expressed by the following constraint

$$position(i) > position(i') \quad \text{for all RAW}(i',i) \tag{6}$$

A Write-after-Write(WAW) dependency occurs between two instructions i' and i (WAW(i',i)) if i and i' write to the same register and i occurs after i' in program order. Similarly, a Write-after-Read(WAR) dependency occurs between instructions i' and i (WAR(i',i)) if i writes into a register that i' reads and i occurs after i' in program order. As our instruction scheduling is performed after register allocation as shown in Figure 7, WAW and WAR dependencies need to be taken into account. These are expressed by the following constraints

$$position(i) > position(i') \quad \text{for all WAW}(i',i), \text{WAR}(i',i) \tag{7}$$

We do not consider the `MOV` instructions for WAW and WAR dependencies. `MOV` instructions do not alter the register values and hence cannot create WAW and WAR dependencies for other instructions. RAW dependencies between a `MOV` instruction and other instructions are handled later in the formulation.

Forwarding Constraints We express the condition under which a specific `MOV` instruction must be scheduled. A custom instruction *may* require a specific `MOV` instruction if the corresponding input operand cannot be forwarded. Let MC_i^k be the k^{th} `MOV` instruction corresponding to custom instruction C_i. Now the `MOV` instruction MC_i^k can be eliminated if the instruction D_i^k producing the corresponding operand is at most 2 instructions away from C_i. Let F_i^k be a binary variable denoting if `MOV` instruction

MC_i^k can be eliminated (i.e., the operand can be forwarded). The dependency constraints ensure that $distance(D_i^k, C_i) > 0$. Therefore,

$$F_i^k = \begin{cases} 1 & \text{if } distance(D_i^k, C_i) \leq 2 \\ 0 & \text{otherwise} \end{cases} \tag{8}$$

This constraint can be linearized as follows

$$distance(D_i^k, C_i) + \infty F_i^k > 2$$
$$distance(D_i^k, C_i) - \infty(1 - F_i^k) \leq 2$$

Let S_i^k be a binary variable denoting if a `MOV` instruction is scheduled. Clearly,

$$S_i^k \leq 1 - F_i^k \tag{9}$$

$$S_i^k = \sum_{j=1}^{S_{max}} X_{map(MC_i^k),j} \tag{10}$$

If more than two input operands of custom instruction C are not available from the forwarding paths, then we need to introduce redundant `MOV` instructions. Therefore,

$$\sum_{k=1}^{input(C_i)} S_i^k = \max\left(0,\ input(C_i) - 2 - \sum_{k=1}^{input(C_i)} F_i^k\right) \tag{11}$$

This can be linearized as

$$\sum_{k=1}^{input(C_i)} S_i^k \geq 0 \tag{12}$$

$$\sum_{k=1}^{input(C_i)} S_i^k \geq input(C_i) - 2 - \sum_{k=1}^{input(C_i)} F_i^k \tag{13}$$

Scheduling Constraints for `MOV` instructions A `MOV` instruction must be scheduled before the corresponding custom instruction. This is ensured by the constraint

$$distance(MC_i^k, C_i) \geq 1 \tag{14}$$

Note that the above constraint is trivially satisfied for the `MOV` instructions that need not be scheduled. Moreover, a `MOV` instruction must be scheduled at most 2 instructions away from the the corresponding custom instruction.

$$distance(MC_i^k, C_i) \leq 2 \ \text{ if } \ S_i^k = 1 \tag{15}$$

This can be linearized as

$$distance(MC_i^k, C_i) - \infty(1 - S_i^k) \leq 2$$

Finally, we should ensure that a `MOV` instruction is scheduled after the instruction that produces the corresponding input operand. That is,

$$distance(D_i^k, MC_i^k) \geq 1 \quad \text{if} \quad S_i^k = 1 \tag{16}$$

This can be linearized as

$$distance(D_i^k, MC_i^k) + \infty(1 - S_i^k) \geq 1$$

Objective Function Our objective is to assign values to the $X_{i,j}$ variables such that the total number of scheduled `MOV` instructions is minimized. That is,

$$\text{minimize} \quad \sum_{i=1}^{M} \sum_{k=1}^{input(C_i)} S_i^k \tag{17}$$

Thus we have formulated the problem of finding the optimal schedule as a integer linear programming problem. Note that the ILP formulation determines the optimal ordering of instructions within a basic block assuming that operands across basic blocks cannot be forwarded. In general the source operands for a custom instruction can come from preceding basic blocks and satisfy forwarding constraint. Hence in the compiler flow shown in Figure 7, we perform another check for the forwarding constraints after instruction scheduling.

5 Experimental Evaluation

In this section we discuss the experimental evaluation of our proposed architecture.

5.1 Setup

Table 1 shows the characteristics of the benchmark programs selected mostly from MiBench [8]. We use SimpleScalar tool set [2] for the experiments. The programs are compiled using gcc 2.7.2.3 with -O3 optimization.

Given an application, we first exhaustively enumerate all possible patterns and their instances [15]. We impose a constraint of maximum 4 input operands and 1 output operand for any pattern. Table 1 shows the number of patterns and pattern instances generated for each benchmark. The execution

Benchmark	Source	Patterns	Instances
Rijndael	MiBench	17	1790
Sha	MiBench	11	33
Blowfish	MiBench	13	197
Djpeg	MiBench	34	133
Compress	GothenBurg	11	26
Ndes	FSU	13	39
Bitcnts	MiBench	11	28
Dijkstra	MiBench	4	5

Table 1: Characteristics of benchmark programs.

Benchmark	Ideal	Forwarding	MOV
Rijndael	64.03%	63.85%	45.44%
Sha	46.82%	40.77%	20.94%
Blowfish	35.56%	35.56%	24.38%
Djpeg	17.42%	17.36%	15.43%
Compress	26.85%	26.85%	21.77%
Ndes	25.62%	22.37%	15.55%
Bitcnts	20.67%	20.67%	18.58%
Dijkstra	28.99%	28.99%	19.12%

Table 2: Speedup under different architectures.

frequencies of the pattern instances are obtained through profiling. The hardware latencies and area of custom instructions (patterns) are obtained using Synopsys synthesis tool. Finally, the number of execution cycles of a custom instruction is computed by normalizing its latency (rounded up to an integer) against that of a multiply accumulate (MAC) operation, which we assume takes exactly one cycle. We do not include floating-point operations, memory accesses, and branches in custom instructions as they introduce non-deterministic behavior. The set of patterns identified is provided as input to the selection phase which outputs the set of custom instructions selected.

The speedup of an application using custom instructions is defined as follows

$$Speedup = (\frac{Cycle_{orig}}{Cycle_{ex}} - 1) * 100$$

where $Cycle_{orig}$ is the number of cycles when the benchmark executes without custom instructions and $Cycle_{ex}$ is the number of cycles when custom instructions are added. For the speedup calculations we assume a single issue in-order processor with 100% data cache hit rate.

5.2 Results

We compare the speedup of each benchmark for three different architectures. The first architecture is "ideal" as it has support for four read ports in the register file and enough space to encode these operands in the instruction format. This architecture is able to provide the highest speedup with custom instructions. The second architecture is based on our idea of exploiting data "forwarding". In this case, we may need additional `MOV` instructions when more than two operands must be read from the register file. The final architecture "MOV" is based on Nios-II where custom `MOV` instructions are used to transfer data from the architectural register file to internal register files (see Section 2).

Table 2 shows the speedup obtained for the three different architectures. The normalized speedup for"forwarding" and "MOV" with respect to the "ideal" case is shown in Figure 8. The performance of forwarding is very close to the ideal performance limit (96% on an average). This is because for the majority of the selected patterns, at least two operands can be obtained through forwarding; thus `MOV` instructions are inserted rarely. This can be seen from Table 3, about 90-100% of selected pattern instances satisfy the forwarding requirements. The case where custom move instructions are inserted ("MOV") achieves only 70% of the performance limit. Thus our technique which uses forwarding to supply additional operands can overcome the limitations in number of register ports and instruction encoding without affecting performance.

In addition, our technique reduces the energy consumption in the register file. As data forwarding is predictable in our approach (refer Section 3.3), register file reads can be avoided for forwarded operands. Table 4 presents the register file energy consumption for the three different architectures. The first column is the ideal case where there are four read ports in the register file. The second column is the case where there are two read ports and custom `MOV` instructions are inserted. The third column is our forwarding-based approach that avoids redundant register file accesses. The energy values presented here are obtained using CACTI 3.2 [13] for 130 nm technology. It is clear from Table 4 that increasing the number of ports of the register file is not an attractive option as it almost doubles the energy consumption. By comparing the second and third column, it can be seen that forwarding results in significant savings in the energy consumption of the register file (25% on an average).

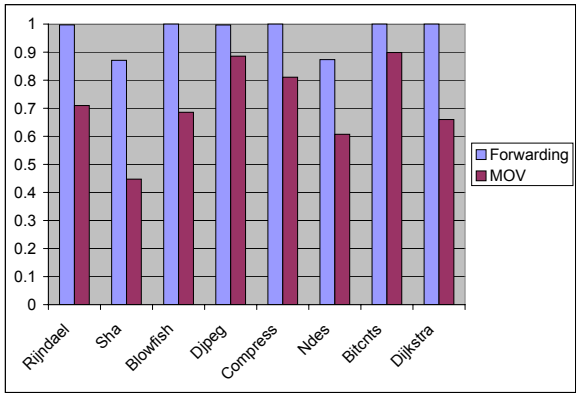

Figure 8: Comparison of the three architectures.

Benchmark	**Instances**	**Forwarded Instances**	**Percentage**
Rijndael	1790	1786	99.77
Sha	33	30	90.90
Blowfish	197	197	100
Djpeg	133	131	100
Compress	26	26	100
Ndes	39	35	89.74
Bitcnts	28	28	100
Dijkstra	5	5	100

Table 3: Percentage of custom instructions that can take advantage of forwarding.

Benchmark	Energy(μJ)		
	Ideal	**MOV**	**Forwarding**
Rijndael	31,462	16,393	12,373
Sha	15,180	7,909	6,587
Blowfish	30,653	15,972	12,951
Djpeg	3,721	1,939	1,411
Compress	4	2	2
Ndes	46	24	17
Bitcnts	41,689	21,722	15,915
Dijkstra	49,078	25,573	16,811

Table 4: Register file energy consumption under different architectures

6 Conclusion and Future Work

In this paper we have shown how data forwarding can be exploited to implement multiple-input single-output (MISO) custom instructions on a processor with limited number of register ports. Our technique overcomes the restrictions imposed by limited register ports and instruction encoding achieving almost ideal speedup. In the future, we plan to address restrictions on the number of output operands.

7 Acknowledgments

This work was partially supported by NUS research grant R252-000-171-112 and A*STAR Project 022/106/0043.

References

[1] K. Atasu, L. Pozzi, and P. Ienne. Automatic application-specific instruction-set extensions under microarchitectural constraints. In *DAC*, 2003.

[2] T. Austin, E. Larson, and D. Ernst. SimpleScalar: An infrastructure for computer system modeling. *IEEE Computer*, 35(2), 2002.

[3] N. Clark, H. Zhong, and S. Mahlke. Processor acceleration through automated instruction set customization. In *MICRO*, 2003.

[4] J. Cong et al. Instruction set extension with shadow registers for configurable processors. In *FPGA*, 2005.

[5] J. Cong, G. Han, and Z. Zhang. Architecture and compilation for data bandwidth improvement in configurable embedded processors. In *ICCAD*, 2005.

[6] Altera Corp. Nios processor reference handbook.

[7] R. E. Gonzalez. Xtensa: A configurable and extensible processor. *IEEE Micro*, 20(2), 2000.

[8] M. R. Guthausch et al. Mibench: A free, commercially representative embedded benchmark suite. In *IEEE 4th Annual Workshop on Workload Characterization*, 2001.

[9] Xilinx Inc. Microblaze soft processor core.

[10] R. Kastner et al. Instruction generation for hybrid reconfigurable systems. *ACM Transaction on Design Automation of Electronic Systems*, 7(2), 2002.

[11] D. Paterson and J. Hennessey. *Computer Organization and Design: The Hardware/Software Interface*. Morgan Kaufmann, 3rd edition, 2004.

[12] L. Pozzi and P. Ienne. Exploiting pipelining to relax register-file port constraints of instruction-set extensions. In *CASES*, 2005.

[13] P. Shivakumar and N. P. Jouppi. CACTI 3.0: An integrated cache timing, power and area model. Technical Report 2001/2, Compaq Computer Corporation, 2001.

[14] P. Yu and T. Mitra. Characterizing embedded applications for instruction-set extensible processors. In *DAC*, 2004.

[15] P. Yu and T. Mitra. Scalable custom instructions identification for instruction-set extensible processors. In *CASES*, 2004.