Energy Efficient Special Instruction Support in an Embedded Processor with Compact ISA

Dongrui She
Eindhoven University of Technology, The Netherlands
d.she@tue.nl

Yifan He
Eindhoven University of Technology, The Netherlands
y.he@tue.nl

Henk Corporaal
Eindhoven University of Technology, The Netherlands
h.corporaal@tue.nl

ABSTRACT

The use of special instructions that execute complex operation patterns is a common approach in application specific processor design to improve performance and efficiency. However, in an embedded generic processor with compact instruction set architecture (ISA), such instructions may lead to large overhead as: *i*) more bits are needed to encode the extra opcodes and operands, resulting in wider instructions; *ii*) more register file (RF) ports are required to provide the extra operands to the function units. Such overhead may increase energy consumption considerably.

In this paper, we propose to support flexible operation pair patterns in a processor with a compact 24-bit RISC-like ISA using: *i*) a partially reconfigurable decoder that exploits the locality of patterns to reduce the requirement for opcode space; *ii*) a software controlled bypass network to reduce the requirement for operand encoding and RF ports. We also propose an energy-aware compiler backend design for the proposed architecture that performs pattern selection and bypass-aware scheduling to generate energy efficient codes. Though proposed design imposes extra constraints on the operation patterns, the experimental results show that the average dynamic instruction count is reduced by over 25%, which is only about 2% less than the architecture without such constraints. Due to the low overhead, the total energy of the proposed architecture reduces by an average of 15.8% compared to the RISC baseline, while the one without constraints achieves almost no energy improvement.

Categories and Subject Descriptors

B.1.4 [**Control Structures and Microprogramming**]: Microprogram Design Aids; C.1 [**Processor Architectures**]; D.3.4 [**Programming Languages**]: Processors

General Terms

Algorithms, Design

Keywords

Reconfigurable architecture, special instruction, low power, code generation

CASES'12, October 7-12, 2012, Tampere, Finland.

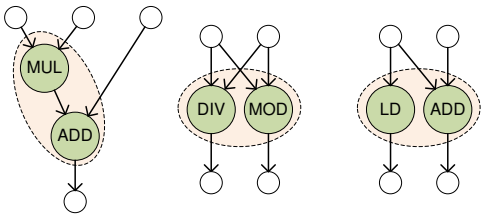

Figure 1: Special operation patterns

1. INTRODUCTION

Embedded systems, especially the ones in mobile devices like smart phones, are becoming more and more important in everyday life. The rapid development in embedded processors enables such devices to run high performance applications like wireless communication and high definition video codecs. However, power efficiency is becoming the bottleneck in high performance embedded system design, especially for those ones that run on limited power sources like batteries. Moreover, high power dissipation makes the chip's thermal design more difficult.

Many applications contain frequently executed operation patterns in the data-flow graphs (DFGs), like the ones shown in Fig. 1. In this work, *special instructions* are defined as instructions that execute such patterns. When properly utilized, special instructions are able to dramatically reduce the number of instructions and communication in the datapath, which have great impact on both performance and energy consumption. In application specific instruction set processor (ASIP) design, it is common to synthesis instruction sets that support such patterns in the targeted applications to achieve better performance and energy efficiency [15, 18, 23]. In this paper, we tackle the problem of integrating flexible special instruction support in an embedded generic processor with a compact instruction set architecture (ISA). Most previous works focused on improving the performance [18, 19]. Apart from performance improvement, the main focus of this work is on energy efficiency of the processor for different types of applications. In most mainstream processor architectures, only few number of such patterns are supported, as supporting arbitrary operation patterns in a generic processor incurs large overhead. From an energy efficiency point of view, the overhead is mainly caused by:

- More bits in the instruction to encode opcodes for all possible patterns and extra operands in the special instructions. It results in wider instruction, and the instruction fetch consumes more energy, even when a normal instruction is fetched. In a compact RISC ISA like the ARM Thumb [5], this problem is more serious as the number of bits in the instruction is very limited.

- More ports in the register file (RF) to provide sufficient data bandwidth for the special function units. A RF with more ports is much less energy efficient. Also, even the normal instructions need to pay the extra cost. Methods like register file clustering [14] or FU internal registers [23] are able to partially solve the problem. But such methods usually lack flexibility and often lead to very complex code generation.

To achieve high energy efficiency, the support for special instructions needs to have low overhead, while still being able to support applications from different domains. In this paper, we propose a schema for integrating special instruction unit (SFU) into a RISC-like embedded processor with 24-bit instruction width. The SFU supports flexible *operation pair* patterns. To integrate the SFU into the RISC datapath with minimum overhead, we use:

- A partially reconfigurable decoder that allows low overhead reconfiguration for each kernel to use its specific patterns. As a result, no extra bits are needed for the special instruction opcode.
- A bypass network in the datapath that is exposed to software, thereby reducing the requirement for both operand encoding and register file ports.

The use of a reconfigurable decoder and an explicit bypass network imposes some constraints on the special instructions the processor can execute, e.g., at least one of the operands of a three-input special instruction has to come form the bypass network. A compiler backend is designed to generate energy efficient code for the proposed architecture. The compiler selects patterns and performs bypass aware scheduling to utilize the SFU and explicit bypass network. A bypass aware DFG transformation is introduced to improve both bypassing and special instruction generation. Experimental results show that for a set of benchmarks form different domains, the proposed architecture achieves an average of 25% reduction in dynamic instruction count, which is only 2% worse than the architecture without constraints on the special instructions. As for energy consumption, the proposed architecture achieves an average reduction of 15.8%, while the unconstrained architecture only reduces 1%. The key contributions of this paper are:

- We propose an architecture that supports flexible operation pairs in a processor with a compact 24-bit RISC-like ISA. The proposed architecture has a partially reconfigurable decoder and a software-controlled bypass network, allowing the processor to support operation pairs without increasing the instruction width or number of register file ports. The proposed architecture is implemented in synthesizable Verilog RTL.
- A compiler backend is designed for the proposed architecture. It is capable of utilizing the SFU and the explicit bypass network to generate energy efficient target code. A complete compiler for the target architecture is implemented based on the LLVM framework.
- Comprehensive and detailed experimental results demonstrate that the proposed architecture and compiler are able to improve the energy efficiency significantly.

The remainder of this paper proceeds as follows: Section 2 describes the DFG patterns we consider in this work and the design of the SFU that executes such patterns. The proposed integration of SFU into the processor datapath with explicit bypass is depicted in Section 3. Section 4 introduces the compiler backend design for the proposed architecture. Detailed and comprehensive results that demonstrate the effectiveness of the proposed design are given in Section 5. Section 6 discusses related work. Finally, Section 7 concludes our findings and discusses future work.

2. OPERATION PATTERNS AND SPECIAL FUNCTION UNIT

Each basic block of a program can be represented by a data-flow graph (DFG) $G(V, E_d, E_f)$, where:

- V is a set of nodes. Each node in V represents either an actual operation or a live-in variable (register file or immediate). In this work, we assume that the operations in V can be directly mapped to a function unit (FU) in a typical RISC processor.
- E_d is a set of directed edges. An edge $e = (u, v)$ represents that node v consumes the output of u, i.e., there is true data dependency between u and v.
- E_f is a set of directed edges. An edge $e = (u, v)$ represents that there is false/output dependency between node v and u.

For a basic block, the DFG is a directed acyclic graph (DAG). A *special operation pattern* is defined as a subgraph of a DFG that contains more than one basic operation. Fig. 1 shows some examples of these patterns. Compared to a combination of basic operations that performs the same computation, executing a special operation pattern using a special instruction has a few advantages:

- Fewer instructions are needed to execute the operations, resulting in less control overhead.
- The communication between operations can be done within the FU, which is usually much more efficient.

For a certain application, some special operation patterns appear frequently. In application specific instruction-set processor (ASIP) design, a common approach for improving performance as well as energy efficiency is to synthesize special function units that support these patterns [15, 18, 23, 26]. Different from the work of ASIP design, the goal of this work is to support special operation patterns in a RISC-like generic processor, without introducing heavy modification to existing architecture and code generation framework. Instead of trying to support arbitrary operation patterns, we focus on a specific type of operation pattern, namely, operations pairs. The definition of the operation pair pattern, as well as motivation of choosing such patterns are given in Section 2.1. The design of a special function unit (SFU) that provides flexible support for these patterns is depicted in Section 2.2. In Section 2.3, we analyze a set of kernels based on the patterns supported by the proposed SFU.

2.1 Operation Pair Patterns

In this work, we want to integrate the support for special operation patterns without major modification to the original RISC architecture. A RISC processor typically has two read ports and one write port (2R1W). Though there are some other possible sources for input operands, like immediate field and bypass network, the number of source operands cannot grow dramatically without heavy modification to the instruction format. The same holds for the destination operand. In addition, the number of arbitrary operation patterns in different applications is huge. The FU that supports all these patterns is likely to be very complex and inefficient. So in this work, we focus on *operation pair* patterns, i.e., patterns with two operations a and b that meet the following criteria:

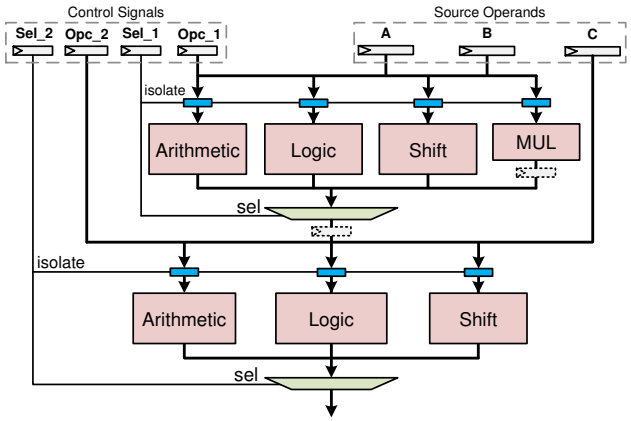

Figure 2: Special function unit

Kernel	Description	Domain
FIR	5-tap finite impulse response filter	Image
Histogram	256-bin histograming	Image
YUV2RGB	YUV to RGB color space conversion	Image
IDCT	2D 8x8 Inverse cosine transformation	Image /Coding
MatVec	Matrix vector multiplication	General
CRC	Cyclic redundancy check code calculation	Network/Storage
DES	The Data Encryption Standard algorithm	Security

Table 1: Kernel description

- There is true dependency between a and b: $(a, b) \in E_d$.
- There are at most three input operands. More formally, for a set of edges P that contains all edges to a or b in E_d except (a, b), we have $|P| \leq 3$;
- At most only one of a and b has consumer outside the pattern, i.e., at least one of the following holds:
  - The result of a is only consumed by b: $\{(a, c)|(a, c) \in E_d, c \neq b\} = \emptyset$. If this constraint is met, only b may have consumers outside the pair pattern.
  - b has no consumer: $\{(b, c)|(b, c) \in E_d\} = \emptyset$. If this constraint is met, only a may have consumers outside the pair pattern.
- There is no path from a to b in G other than (a, b). So combining them does not create cycles in G.

Integrating such patterns in a RISC processor is relatively easy: we only need to supply one more source operands than for a normal operation.

2.2 Special Function Unit Design

The design of our special function unit is shown in Fig. 2. The SFU supports two levels of basic operations. To avoid introducing large area and timing overhead, only one multiplier is included in the SFU, which is put in the first level. The design of the SFU allows almost arbitrary combinations of operation pairs that satisfy the constraints in Section 2.1. When fully decoded, an 18-bit control signal is needed for the SFU to execute one special operation. To improve the energy efficiency of the SFU, *operand isolation* is used to isolate each sub-function-unit. So a unit only toggles when it actually needs to perform computation, thereby reducing unintended circuit activities.

As shown in Fig. 2, the SFU can be pipelined, or partially pipelined, which allows architectures with SFUs to reach high frequency if necessary.

2.3 Application Analysis

We analyzed seven kernels listed in Table 1, which come from various application domains. In total, 35 distinct pair

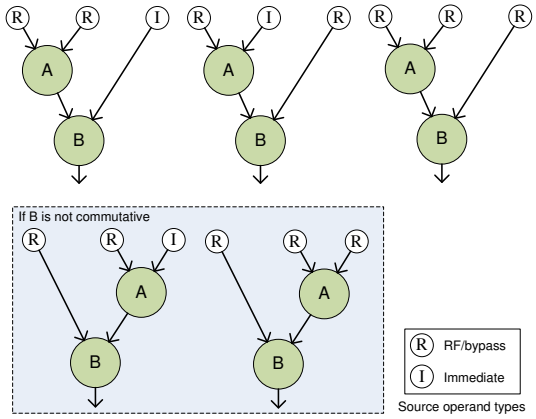

Figure 3: Cases for the same pattern that need different coding

Total	Hist	FIR	IDCT	Y2R	MatVec	CRC	DES
35	2	3	9	9	11	9	12

Table 2: Kernel pattern statistics

patterns that can be supported by the SFU are needed for these kernels. The number of patterns can grow much larger if more applications from different domains are included. In addition, to generate a valid special instruction from an operation pattern, more information needs to be encoded, e.g., if there is immediate and whether the immediate is for the first or the second operation. Fig. 3 shows an example of a three input operation pair that requires different control coding. When all these factors are considered, the total number of different special instruction patterns can easily grow to way over one hundred.

However, if we look into each individual kernel, we can see that the number of patterns used in one kernel is much smaller than the number of total patterns. Table 2 shows the statistics of pattern matches in the seven representative kernels from different domains. The statistics show that it is possible to exploit the *temporal locality of patterns* to reduce the number of patterns a processor needs to support during the execution of an application or a kernel. Findings in [17, 26] also lead to similar conclusion. This observation can be used to guide the design of efficient special instruction support in processors, which is discussed in Section 3.

3. INTEGRATING SFU INTO PROCESSORS WITH COMPACT ISA

In this work, a 4-stage RISC processor with a 24-bit instruction set architecture (ISA) is used as the baseline. The key features of the baseline architecture are described in Table 3. Most instructions are three-address instructions: two source operands and one destination are encoded. Fig. 4 depicts the datapath of the baseline processor.

In the baseline architecture, the major limiting factors of integrating the SFU introduced in Section 2.2 are:

Instruction width	24 bits
Pipeline stages	4
Register file	32b×32, 2R1W
Opcode	6 bits
Immediate	8 bits

Table 3: Key features of the baseline ISA

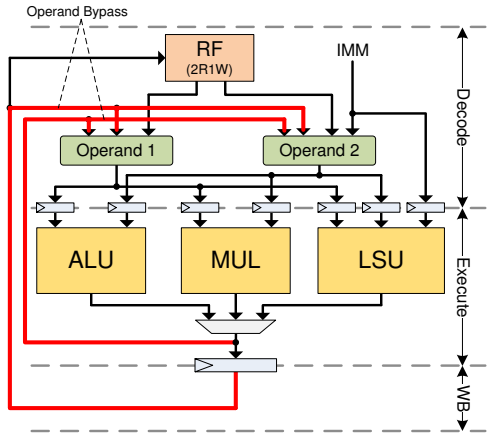

Figure 4: A typical RISC datapath

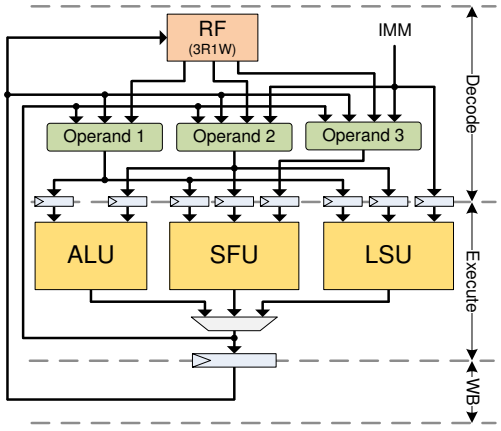

Figure 5: SFU datapath without constraints

- After adding the basic integer and control operations, only less than 16 opcodes are left in the opcode space.
- At most 3 bits can be used for encoding the extra operand in three-input instructions, which are not enough for a register index.
- The 2R1W RF cannot provide enough operand bandwidth for the SFU.

A straightforward solution to these problems is to increased instruction width and number of RF ports. To accommodate the extra opcodes and register index, at least additional 7 bits are needed (5 bits for the third register operand, 2 bits for extra opcodes). As a result, the width of the instruction memory increases to 32 bits. In addition, the RF needs to have three read ports (3R1W) in order to provide sufficient bandwidth for the SFU. The resulting datapath is shown in Fig. 5. To avoid high area overhead, the multiplier is absorbed into the SFU. Based on the estimation of CACTI [6], the energy consumption of each access to the instruction memory is increased by 10% to 30% depending on the size and configuration. And based on the implementation result, the energy consumption of the RF is also increased by 12% due to the extra read port. Since both instruction memory and register file are among the most frequently used components in a processor, an architecture with such large overhead is unlikely to be energy efficient.

To improve the energy efficiency, such overhead have to be mitigated. In this work, we propose an energy efficient support for the SFU by using: *i*) a partially reconfigurable

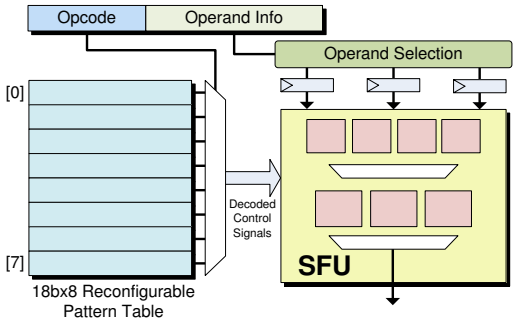

Figure 6: Partially reconfigurable SFU decoder

decoder that exploit the locality of the operation patterns to reduce the opcode encoding requirement; *ii*) a software-controlled bypass network that exploit the processor pipeline to reduce the operand encoding and RF port requirement. Section 3.1 and Section 3.2 describe the details of the partially reconfigurable decoder and the software-controlled bypass network, respectively. Section 3.3 depicts how the SFU is integrated into the baseline processor.

3.1 Partially Reconfigurable Decoder for SFU

As discussed in Section 2.3, a key observation is that although a large number of patterns are needed to cover the operation patterns in different applications, only a small number of such patterns are active in one kernel, i.e., in most kernels, the operation patterns have good locality. To utilize such locality, this work introduces a partially reconfigurable decoder for the SFU.

Fig. 6 depicts the structure of the reconfigurable decoder for the SFU. The center of the decoder is a look-up-table with eight entries, called *pattern table*. Each entry in the pattern table stores an 18-bit control signal required by a special instruction. Since the table only has eight entries, the free opcodes in the opcode space can be used to address it. When a special instruction is fetched, the decoder reads a pattern table entry and uses it to control the SFU; when a normal instruction is fetched, the decoder proceeds as a normal RISC decoder, and the pattern table is clock gated to eliminate unnecessary accesses.

The pattern table is visible to the software. So when different operation patterns are needed, the software can reconfigure the SFU decoder by writing extra control signal needed by these operations into the pattern table. By enabling the reconfiguration of the pattern table, the processor is able to use all the operation patterns supported by the SFU. And since in most cases the operation patterns have good locality, the overhead of reconfiguration is very low.

3.2 Explicit Bypass

In a typical pipelined datapath of a processor, like the one in Fig. 4, there is a bypass/forwarding network, whose primary function is to avoid pipeline stalls caused by data dependencies. A side effect of such a network is that many operands can be read from the pipeline registers instead of the RF. There are two types of RF access elimination:

- *Bypassing*: the result of an operation can be read from the pipeline register before it is written back to RF;
- *Dead writeback elimination*: if all uses of a variable are bypassed, its writeback is no longer necessary.

However, in conventional processor architectures, such bypassing network is invisible to software, which makes it difficult to eliminate RF accesses: *i*) bypassing requires RF

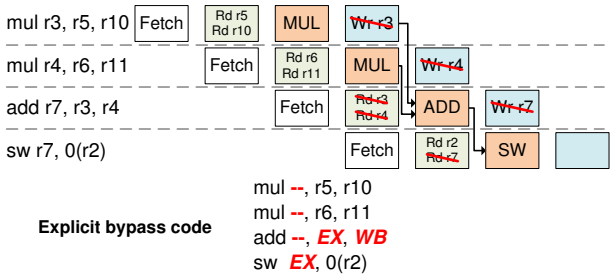

Figure 7: Reduce RF accesses via explicit bypass

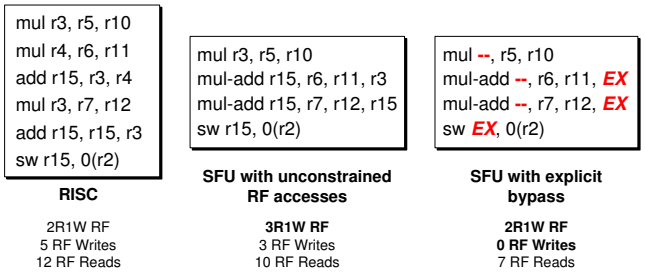

Figure 8: Special instruction example

indexes to be checked before decode stage, which may increase the critical path of fetch stage or results in an extra pipeline stage; *ii*) dead writeback elimination is impossible unless liveness information is explicitly encoded in instructions. In this work, we propose to use a bypass network that is controlled by software, i.e., the bypassing information is statically encoded in the instructions. Fig. 7 shows an example of reducing RF accesses via explicit bypassing. Apart form reducing the total number of register accesses, explicit bypassing helps integrating the SFU without increasing instruction width and RF ports in two ways:

- Encoding a bypass source uses fewer bits than an RF index, as the number of bypass sources are much fewer than the number of registers in RF (4 vs. 32).
- Fewer RF ports are required when some operands are bypassed.

By imposing the constraint that at least one of the source operands in a three-input special instruction has to come from the bypass network, the special instructions can be encoded in the 24-bit instruction, and there is no need to increase the number of RF ports. Fig. 8 shows an example of special instructions. In processor with unconstrained SFU, the number of instructions is reduced from 6 to 4, at the cost of increasing the number of read ports of the RF from 2 to 3. In a processor with explicit bypassing, the same code size improvement can be achieved even when there is a constraint that at least one of the operands comes from the bypass network. And with such a constraint, the requirements for the instruction bits and RF port are reduced.

3.3 Integrating SFU into Processor Datapath

We propose an architecture that is able to support all the operation pair patterns of the SFU described in Section 2.2, by employing the partially reconfigurable decoder and explicit bypass network introduced in previous subsections. Fig. 9 shows the datapath of the proposed processor architecture. Note that because there are input registers for each FU, the result of one operation is stable at the output port of the FU until the next operation that uses the same FU comes. So it is possible to use the output of each FU as

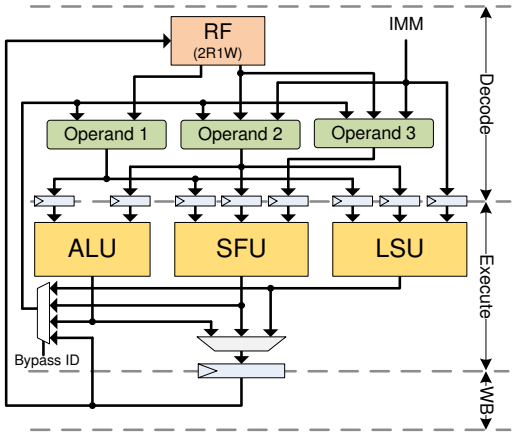

Figure 9: SFU datapath with constraints

a separate bypass source, which increases the possibility of bypassing. Compared to the one with direct SFU support (Fig. 5), the proposed architecture imposes extra constraints on the special instructions it can execute:

- For a three-input special instruction, at least one of the source operands has to come from the bypass network.
- At most eight special instruction patterns are active at the same time. To support different patterns, the program needs to reconfigure the pattern table.

With these constraints, the proposed architecture is much more energy efficient: instruction width remains 24 bits instead of 32 bits and the RF is 2R1W instead of 3R1W. To use explicit bypass without changing the normal instruction format, part of the RF address space is used for the bypass source. As a result, the number of registers in the RF reduces from 32 to 28. The effect of a smaller RF is mitigated by the explicit bypassing, as it eliminates the necessity of allocating registers for short-live variables in many cases.

The introduction of a pattern table and an explicit bypass results in extra context when exceptions happen. The pattern table can be handled in a similar fashion as general purpose registers. For explicit bypass, it is required that the processor saves the complete state for the execute and writeback stages of the pipeline. This can be done using a scan-chain that automatically saves/restores the registers when exceptions happen. Since the number of registers is small, the overhead in area and response time is small.

4. CODE GENERATION FOR SPECIAL INSTRUCTIONS

The compiler in this work is implemented based on the open-source LLVM framework [3]. Fig. 10 shows the backend compilation flow for the proposed architecture. The input of the backend is a low-level intermediate representation (IR), which is basically RISC assembly with virtual registers, embedded with control-flow and data-flow information. Since the proposed architecture only has small modification to the original RISC architecture, most part of the compiler can simply reuse the same passes as a compiler for RISC architecture. The main difference is that the backend needs to be aware of the explicit bypass network and has to utilize the special function unit (SFU). The selection of pair patterns for generating special instructions is described in Section 4.1. Section 4.2 discusses the changes in instruction scheduler and register allocator due to explicit bypass. A transparent DFG transform that improves bypassing and

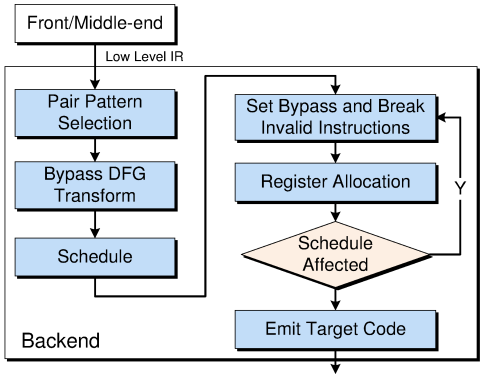

Figure 10: Compiler backend flow

helps the selected patterns to meet the architectural constraints, is described in Section 4.3.

4.1 Pair Pattern Selection

To use the SFU, the compiler needs to choose pairs of DFG nodes that can be used to generate special instructions, i.e., the pair pattern selection. The first step of pair pattern selection is to find all the node pairs whose patterns are supported by the SFU in the data-flow graph (DFG) under the constraints described in Section 2.1. A set containing all these node pairs is obtained by a scan through all nodes in the DFG, M. Each pair in M is called a *match*. The matches that are obviously not going to meet the operand bypass constraint, e.g., the ones with three non-constant live-in variables, are excluded from M.

The next step is to choose a subset S of M for generating special instructions. Obviously each DFG node should only be used by one pattern in S, as duplicating DFG nodes only results in extra energy consumption in the pair patterns. A match interference graph $G_I(V, E)$ can be built:

- V is a set of nodes representing all possible matches.
- E is a set of undirected edges. $(u, v) \in E$ means that match u and match v share a common DFG node. So u and v cannot be selected simultaneously.

An example of match interference graph is given in Fig. 11: on the left is a DFG with four possible matches; on the right is the match interference graph of the four possible matches. S should be an *independent set* of G_I, i.e., nodes in S are pair-wise non-adjacent in G_I. The objective here is to find as many pairs as possible, which is essentially to get the *maximum independent set* (MIS) of G_I, i.e., the independent set with maximum cardinality.

Though MIS is NP-complete in general, the minimum degree heuristics performs very well for sparse and bounded degree graph [2]. In the DFG pair pattern selection, many nodes in the match interference graph have the same degree, which results in many ties in minimum degree selection Since in the proposed architecture, only limited number of operation patterns are supported without reconfiguration, the pattern frequency is used to break the ties. The algorithm used for pattern selection is depicted in Algorithm 1. The algorithm yields $\{1, 3\}$ for the example in Fig. 11, which is the MIS of the interference graph.

4.2 Instruction Scheduling and Register Allocation

A list scheduler is used to perform basic block level scheduling. In the proposed architecture, the total number of physical registers are reduced as part of the RF address space is

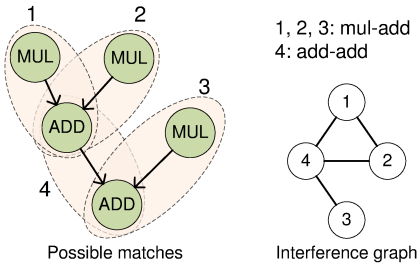

Figure 11: Operation patterns matches

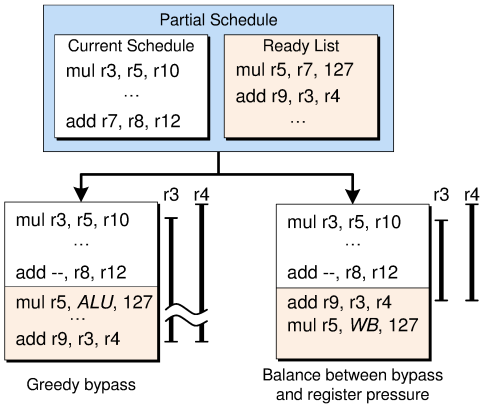

Figure 12: Bypass and register pressure trade-offs

used by bypass sources. So although explicit bypass eliminates the need for many temporary registers, it is still very important for the compiler to make sure that register pressure stays low. When the list scheduler greedily chooses the node with maximum number of bypass, the register pressure may go up. Fig. 12 shows an example of how a greedy bypass scheduler may increase the register pressure. In this work we use a scheduling algorithm which is similar to the integrated prepass scheduling (IPS) [1]. Depending on the register pressure of the current partial schedule, The scheduler switches between between two policies: *i*) choose the node that maximizes bypassing, or *ii*) choose the node that minimizes register pressure. The details of the scheduling algorithm is given in Algorithm 2. The register pressure threshold can be chosen based on the estimation of available registers for the basic block. The register allocation is done with a graph-coloring algorithm. The register allocation is almost the same as the one used for a RISC processor, except that small constant values (ones that can fit in the

Algorithm 1: Pattern Selection

Input : Match interference graph of the basic block $M(V, E)$ and pattern frequency F_p
Output : Set of pattern matches S in which nodes do not interfere with each other

$S \leftarrow \emptyset$
while $M \neq \emptyset$ **do**
  $D \leftarrow \{d | d \in V, \nexists u \in V : degree(u) < degree(d)\}$
  if $|D| = 1$ **then**
    $n \leftarrow D[0]$
  else
    $Q \leftarrow \{q | q \in D, \nexists u \in D : F_p(Pat(u)) > F_p(Pat(q))\}$
    `// Pick the first one if Q has more than one node`
    $n \leftarrow Q[0]$
  end
  $S \leftarrow S \cup \{n\}$
  Remove n from M, along with all its edges and neighbors
end

instruction immediate field) in special instructions are not always re-materialized to immediate filed when it results in an instruction with two immediate values, which is invalid.

After the scheduling, a scan through all instructions is preformed to check for invalid special instructions, i.e., the instructions that do not meet the constraints given in Section 3.3. If a special instruction is found to be invalid, the checker decomposes it into normal instructions. Due to the nature of explicit bypassing, this transformation does not increase register usage. Then the compiler collects pattern informations and decides where to insert the reconfiguration codes. In this work, there are two possible scenarios:

- If the number of patterns used in a function is less than or equal to the pattern table size, all patterns are loaded at the entry block of the function.
- If the number of patterns used in a function exceeds the pattern table capacity, the compiler tries to perform reconfiguration before entering each intensive loop. The loop information can be obtained through static estimation or profiling.

When both ways fail to accommodate all used patterns, the compiler selects the most frequently used patterns. And a special instruction whose pattern is not in the pattern table is decomposed to two normal instructions. When there is a function call, the pattern table becomes part of the context, and needs to be saved like the general purpose registers. When compiler optimization is enabled, the frequently called simple functions usually get in-lined. So we expected the reconfiguration overhead to be negligible in most cases.

As shown in Fig. 10, whenever a code transformation changes the schedule, the bypass status of each instruction needs to be updated, so the same check needs to be performed. The process terminates: in the worst case, the loop stops when all special instructions are decomposed to normal instructions. In practice only one or two iterations are sufficient in most cases.

Algorithm 2: Basic Block Scheduling

```
   Input  : DFG G(V, E_d, E_f) and register pressure threshold t_r
   Output : The schedule of the DFG T : V ↦ ℕ
1  // Set number of cycles based on conservative estimation
2  R ← ∅                                              // Ready set
3  L ← ∅                                          // Live variable set
4  S ← ∅                                // Set of scheduled operations
5  c ← 0
6  while |S| ≠ |V| do
7      if |L| < t_r then
8          o ← find_node_with_max_bypass (R, T)
9      else
10         o ← find_node_reduces_max_register_pressure (R,
           T, L, G)
11     end
12     for s ∈ {u|u ∈ V, (o, u) ∈ E_d ∪ E_f} do
13         if s is enabled by o then
14             R ← R ∪ {s}
15         end
16     end
17     for p ∈ {u|u ∈ V, (u, o) ∈ E_d} do
18         if o is last use of p then
19             L ← L \ {p}
20         end
21     end
22     if o has value output then
23         L ← L ∪ {o}
24     end
25     S ← S ∪ {o}
26     T[o] ← c
27     R ← R \ {o}
28     c ← c + 1
29 end
```

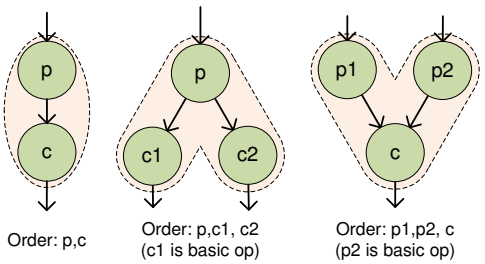

Figure 13: DFG clustering for explicit bypass

4.3 Bypass-Aware Graph Clustering

Whether an operand is bypassed or not depends on the schedule distance between the producer operation and the consumer. In the proposed architecture, bypassing not only affects the number of RF accesses, but also decides whether a special instruction is valid. To better utilize the explicit bypass, and more importantly, to reduce the number of special instructions invalidated by the scheduling, we introduce a bypass aware DFG transformation before the scheduling. The basic idea is to cluster a set of nodes if the nodes can be scheduled in such a way that:

- All the intermediate results are bypassed.
- Register pressure does not increase.
- Combining these nodes does not result in a cyclic dependency.

However, to find arbitrary subgraphs that meet these constraints is difficult as the number of possible subgraphs grows exponentially with the number of nodes. In this work, we introduce fixed patterns that meet the constraints and can be scheduled easily. We choose the patterns shown in Fig. 13, as they are easy to match and common in DFGs from different applications. The order in of each pattern in Fig. 13 represents the internal order of the nodes in the cluster, which satisfies the afore-mentioned constraints. The clustering can be done iteratively until no more transform is possible.

The transformation described in this subsection is transparent to the scheduler. After the graph clustering, the scheduler can schedule the resulting graph as if it is a normal DFG. After the scheduling, a valid schedule of the original DFG can be produced by expanding each clustered node to the internal list of DFG nodes.

5. EVALUATION AND ANALYSIS

Table 4 presents the architectures used in the experiments. The proposed architecture, i.e., with partially reconfigurable decoder, explicit bypass network, and constrained special instruction patterns (see Section 3.3), is called *SFU-24*. And the architecture that integrates SFU without the constraints in *SFU-24* is called *SFU-32*. The datapaths of the baseline, SFU-32 and SFU-24 are shown in Fig. 4, Fig. 5 and Fig. 9, respectively. All three cores are implemented in Verilog RTL and synthesized with TSMC 90nm low power library at 1.2V and typical case. Clock gating is used to minimize dynamic power consumption. The core energy consumption is estimated with the backend information and real toggle rate generated by post-synthesis simulation. The area and energy consumption of the memory are estimated with CACTI [6], using 90nm low operating power technology. Table 5 shows the energy model of the memory used in the experiments.

5.1 Area and Frequency

The implementation results of the three architectures are shown in Table 6. The increase in the core area is under-

Architecture	Baseline (Base)	Unconstrained SFU (SFU-32)	Proposed (SFU-24)
Instruction Width	24 bits	32 bits	24 bits
Instruction Memory	12kB 24-bit	16kB 32-bit	12kB 24-bit
	4k words		
Data Memory	16kB 32-bit		
Register File	32b×32 2R1W	32b×32 3R1W	32b×28 2R1W
SFU Patterns	0	128	8

Table 4: Configuration of different architectures

Memory	16kB 32-bit	12kB 24-bit
Energy per access (pJ)	15.38	11.62

Table 5: Memory energy consumption

standable and expected, as the SFU, as well as its decoding part, are much more complex compared to simple FUs in RISC. The core area of SFU-32 is slightly larger than SFU-24 as it needs to support more patterns in the decoder. The difference in memory area between SFU-32 and SFU-24 is significant. This is caused by the instruction memory since SFU-32 uses 32-bit instructions, while SFU-24 uses 24-bit instructions. In all, the SFU-32 pays a very high price in terms of area. In contrast, the proposed SFU-24 realizes the special instruction support with a relatively small overhead. In particular, it does not increase the memory area, which is the dominant part in many modern processors.

The reduced maximum frequency of SFU-32 and SFU-24 is mainly caused by the un-pipelined SFU, which has two levels of sub-function-units. It can be mitigated by introducing a pipeline stage in the SFU, though the trade-offs are out of the scope of this work. In this work, we use the un-pipelined SFU in both SFU-32 and SFU-24.

5.2 Energy Consumption

Table 1 lists the benchmarks used in the experiments. These kernels are from various application domains. The code for the proposed SFU-24 is generated by the compiler described in Section 4. For SFU-32, the code generation process is almost the same as SFU-24, except that all the constraints on operand bypassing and opcode space are removed, and no reconfiguration code is generated. All benchmark programs are compiled with maximum optimization enabled (-O3). Table 7 shows the absolute results of the baseline processor. The memory energy in the table includes both instruction memory and data memory. The energy consumption of each kernel is calculated by multiplying the number of cycles with the average energy (i.e., core + memory) per cycle. In the remainder of this sub section, we normalize all results to the baseline.

Architecture	Base	SFU-32	SFU-24
Normalized Core Area	1	1.309	1.268
Normalized Memory Area	1	1.154	1
Maximum Frequency	450MHz	325MHz	325MHz

Table 6: Implementation result comparison

Kernel	Simulated Cycles	Average Core Energy per Cycle	Average Memory Energy per Cycle
Histogram	21547	11.05pJ	16.10pJ
FIR	40973	18.41pJ	16.24pJ
IDCT	2303	17.93pJ	14.56pJ
YUV2RGB	43032	17.88pJ	13.82pJ
MatVec	3729	13.27pJ	14.00pJ
CRC	162017	12.73pJ	11.82pJ
DES	857130	14.89pJ	14.64pJ

Table 7: Results of the baseline architecture

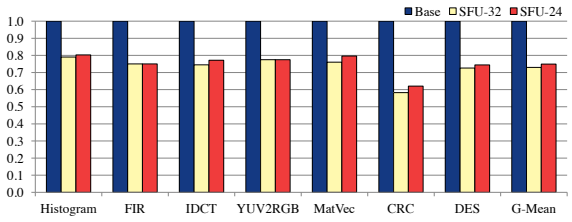

Figure 14: Dynamic instruction count (overhead included)

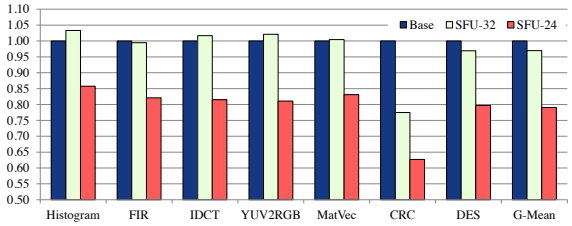

Figure 15: Normalized memory energy consumption

Fig. 14 shows the normalized dynamic instruction count of the three different cores. Including the overhead of reconfiguration, SFU-24 achieves a reduction of 25%, which is only 2% worse than SFU-32. When the instruction width is factored in, as shown in Fig. 15, the memory energy consumption of SFU-24 is much less than SFU-32. Though the number of fetches is reduced dramatically, SFU-32 only achieves 3.5% average memory energy reduction due to increased instruction width. In 4 out of 7 benchmarks the energy consumption actually goes up. In contrast, the proposed SFU-24 is able to directly convert the reduction in instruction count into memory energy saving. An average of 21% saving is observed.

Fig. 17 shows the normalized core energy consumption. Comparing to the baseline processor, the proposed SFU-24 reaches a maximal core energy reduction by 21.5% in the FIR case, and by 11.2% on average. The main contributions of energy reduction are from: 1) reduced RF access energy; 2) reduced datapath and control path overhead due to merged operations.

On the other hand, the SFU-32 increases the average core energy by 0.3%. And it performs very bad in two cases: FIR and IDCT, in which the core energy increases by over 8%. The explicit bypass network is an important contributing factor in this huge difference. As show in Fig. 16, the number of accesses to the RF in SFU-24 is significantly reduced. In addition, the RF in SFU-24 has less ports than the one in SFU-32. As a result, the core of SFU-24 consumes much less energy compared to SFU-32, for which in both FIR and IDCT, a degradation of over 5% is observed.

Fig. 18 shows the normalized total energy consumption. The proposed SFU-24 reduces both the memory and core energy, and it achieves an average saving of 15.8%. It reaches a maximal of 33.1% energy saving in CRC. While the total energy saving of SFU-32 is only 1.1%.

These results show that although the use of SFU is able to significantly reduce the dynamic instruction count, directly putting the SFU into a generic processor without any constraint does not result in an energy efficient architecture. The proposed architecture with a partially reconfigurable decoder and an explicit bypass network is able to reach a balance between the energy efficiency and the flexibility of

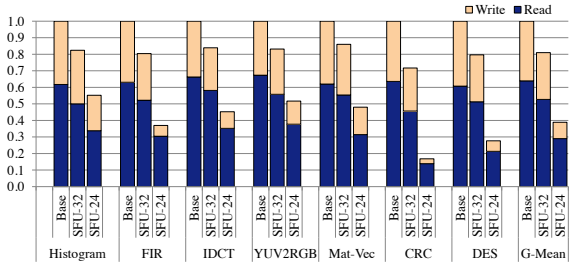

Figure 16: Normalized number of RF accesses

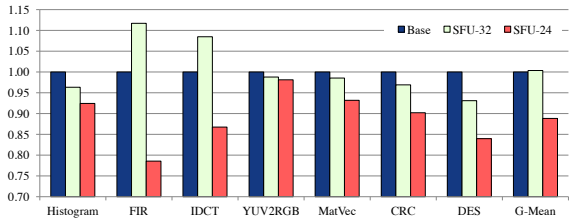

Figure 17: Normalized core energy consumption

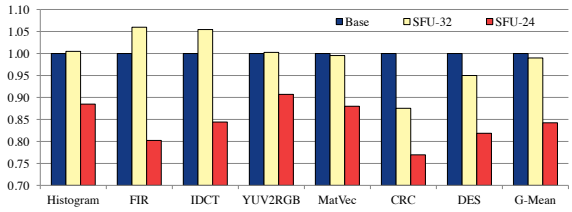

Figure 18: Normalized total energy consumption

the SFU, and it results in an architecture with high energy efficiency and good performance.

6. RELATED WORK

The use of complex operation patterns, called instruction set extension (ISE), is common in instruction set synthesis for ASIP design [15, 18, 23]. There are also studies trying to integrate such ISE in general purpose architectures [19, 20, 21], most of which focus on improving the performance.

The data bandwidth from the register file to the FUs is an important constraint in ISE design [13]. Leupers et al. introduced special register file called *internal registers* (IR) for the special instruction units [23]. The IR is an effective way of implementing application specific special instruction, but it lacks flexibility and complicates the code generation as the registers and FUs are no longer orthogonal, i.e., an FU cannot accesses arbitrary registers. Karuri et al. proposed RF clustering in single issue processor to mitigate the register file port pressure in ISE in ASIP design [14]. While reducing port pressure, the RF clustering, which is similar to what is used in clustered VLIW architectures, also makes the code generation much more complex. Pozzi and Ienne exploited the fact that pipelined SFUs do not need all operands in the same cycle to distribute register file accesses across multiple cycles [4]. This cannot be applied to the SFUs that are similar to the one used in this work. Utilizing the bypass network has been proven to be an efficient way to increase operand bandwidth and reduce register file energy in different types of architectures [7, 10, 12, 24]. Jayaseelan et al. proposed explicit forwarding to reduce register file port pressure and operand encoding cost for application specific ISE in a RISC-like datapath, which resembles the idea of explicit bypass in this work [21]. However the power model used in [21] only considers the consumption of the register file, which is over-simplified for a complete processor design. The overall energy efficiency of the proposed architecture is not clear. Cong et al. proposed shadow registers to solve the operand bandwidth issue for supporting special instructions in a configurable processor [11]. The shadow registers are similar to explicit pipeline registers, but have more flexibility. To avoid dramatical increase of control bits, the shadow registers are hash-mapped, which may be less efficient in terms of energy. In this paper, we explored the trade-offs in utilizing bypass network for energy efficient ISE in a generic processor with compact ISA and presented detailed and realistic results. The proposed solution achieved high energy efficiency while maintaining the generality of the baseline.

In ASIP designs, dynamic instruction set configuration is often used to optimize the resource usage. The rotating instruction set processing platform (RISPP) uses a run-time reconfigurable instruction set to enable the reuse of resources for special instructions in ASIP [16]. Huynh et al. proposed dynamic instruction set configuration for a flexible reconfigurable custom instruction unit, addressing the trade-offs between area, performance and reconfiguration cost [9]. Some recent works proposed integration of special instructions for a relative wide range of applications. Clark et al. proposed integration of a configurable compute accelerator (CCA) into a general-purpose processor [19, 20]. The architecture of CCA is relatively complex and it requires up to 4 inputs and 2 outputs, as its main objective is improving performance. The control part of CCA is designed to be transparent such that the code can be executed with or without CCA. Woh et al. proposed AnySP, a wide SIMD signal processor targeting wireless and multimedia applications [17]. In AnySP the idea of operation pairs is similar to the SFU design in this work, and the operand problem is partially solved by introducing an extra small RF. In PEPSC, an architecture designed for efficient scientific computing, Dasika et al. proposed a FPU that is capable of executing up to five back-to-back operation [8]. In this work we exploited the locality of the special operation patterns in designing a partially reconfigurable decoder to achieve energy efficient integration of SFU into a RISC processor with compact ISA, which allowed the proposed architecture to improve energy efficiency substantially in different domains.

Selection and scheduling for special instructions is one of the most important parts in code generation for ASIP and many reconfigurable architectures. Kastner et al. proposed an algorithm for generating special instructions in a system with reconfigurable fabrics [22]. Guo et al. proposed a graph covering algorithm for code generation of Montium reconfigurable processor [25]. Park et al. presented a greedy algorithm for increasing the bypassing in a RISC processor [24]. In [21], integer linear programming (ILP) is used to perform bypass aware scheduling in a processor with application specific ISE. The proposed algorithm inserts register copying instructions to meet the special instruction constraints.

In this work, we proposed a novel architecture that uses special instructions to improve the energy efficiency of a generic processor with a compact ISA. Two major issues: *i*) opcode and operand encoding; *ii*) operand bandwidth to SFU are solved by using a partially reconfigurable decoder and explicit bypass network.

7. CONCLUSIONS AND FUTURE WORK

Integrating a special function unit (SFU) that executes complex operations into a generic processor for energy efficiency is not easy, as special instructions may incur large overhead, especially when the ISA is a compact one. This

paper introduced an architecture for integrating SFU that supports flexible operation pair patterns in a generic processor with a compact ISA. A partially reconfigurable decoder and a software-controlled explicit bypass network are used to: *i*) encode extra opcodes and operands in the limited instruction coding space; *ii*) supply sufficient data to the special instructions without increasing the number of register file ports. We presented a compiler backend design for the proposed architecture. The compiler is able to utilize the SFU and the explicit bypass network to generate energy efficient code. Results including benchmarks from different domains demonstrate that the proposed architecture and compiler are effective: average dynamic instruction count is reduced by over 25%. The total processor energy consumption is reduced by 15.8%.

Further trade-offs between performance, energy and area are possible when extra pipeline stages in the SFU are introduced. Future work also includes supporting more complex patterns, and exploring the trade-offs between the complexity of the SFU and the energy efficiency of the processor.

8. ACKNOWLEDGMENTS

This work is supported by the Dutch Technology Foundation STW, project NEST 10346, and the Ministry of Economic Affairs of the Netherlands, project EVA PID07121.

9. REFERENCES

[1] J. R. Goodman and W. Hsu. Code scheduling and register allocation in large basic blocks. In *Proceedings of the 2nd International Conference on Supercomputing*, pages 442–452, 1988.

[2] M. Halldórsson and J. Radhakrishnan. Greed is good: approximating independent sets in sparse and bounded-degree graphs. In *Proceedings of the 26th Symposium on Theory of Computing*, pages 439–448, 1994.

[3] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization*, pages 75–86, 2004.

[4] L. Pozzi and P. Ienne. Exploiting pipelining to relax register-file port constraints of instruction-set extensions. In *Proceedings of the 2005 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, pages 2–10, 2005.

[5] ARM Ltd. ARM Thumb Instruction Set. http://www.arm.com/.

[6] CACTI. cacti 5.3, rev 174. http://quid.hpl.hp.com:9081/cacti/.

[7] D. She et al. Scheduling for register file energy minimization in explicit datapath architectures. In *Design, Automation Test in Europe Conference Exhibition 2012*, 2012.

[8] G. Dasika et al. PEPSC: A power-efficient processor for scientific computing. In *Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques*, pages 101–110, 2011.

[9] H. P. Huynh et al. An efficient framework for dynamic reconfiguration of instruction-set customization. In *Proceedings of the 2007 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, pages 135–144, 2007.

[10] J. Balfour et al. An energy-efficient processor architecture for embedded systems. *Computer Architecture Letters*, 7(1):29–32, 2007.

[11] J. Cong et al. Architecture and compilation for data bandwidth improvement in configurable embedded processors. In *Proceedings of the 2005 International Conference on Computer-Aided Design*, pages 263–270, 2005.

[12] J.Balfour et al. Operand registers and explicit operand forwarding. *Computer Architecture Letters*, 8(2):60–63, 2009.

[13] K. Atasu et al. Automatic application-specific instruction-set extensions under microarchitectural constraints. In *Proceedings of the 40th Design Automation Conference*, pages 256–261, 2003.

[14] K. Karuri et al. Increasing data-bandwidth to instruction-set extensions through register clustering. In *Proceedings of the 2007 International Conference on Computer-Aided Design*, pages 166–171, 2007.

[15] K. Karuri et al. A generic design flow for application specific processor customization through instruction-set extensions. In *Proceedings of the 9th International Workshop on Embedded Computer Systems*, pages 204–214, 2009.

[16] L. Bauer et al. Rispp: Rotating instruction set processing platform. In *Proceedings of the 44th Design Automation Conference*, pages 791 –796, 2007.

[17] M. Woh et al. AnySP: anytime anywhere anyway signal processing. In *Proceedings of the 36th International Symposium on Computer Architecture*, pages 128–139, 2009.

[18] N. Clark et al. Processor acceleration through automated instruction set customization. In *Proceedings of the 36th International Symposium on Microarchitecture*, pages 129–140, 2003.

[19] N. Clark et al. Application-specific processing on a general-purpose core via transparent instruction set customization. In *Proceedings of the 37th International Symposium on Microarchitecture*, pages 30–40, 2004.

[20] N. Clark et al. An architecture framework for transparent instruction set customization in embedded processors. In *Proceedings of the 32nd International Symposium on Computer Architecture*, pages 272–283, 2005.

[21] R. Jayaseelan et al. Exploiting forwarding to improve data bandwidth of instruction-set extensions. In *Proceedings of the 43rd Design Automation Conference*, pages 43–48, 2006.

[22] R. Kastner et al. Instruction generation for hybrid reconfigurable systems. *ACM Trans. on Design Automation of Electronic Systems*, 7(4):605–627, 2002.

[23] R. Leupers et al. A design flow for configurable embedded processors based on optimized instruction set extension synthesis. In *Design, Automation and Test in Europe, 2006*, pages 581–586, 2006.

[24] S. Park et al. Bypass aware instruction scheduling for register file power reduction. In *Proceedings of the 2006 Conference on Language, Compilers, and Tool Support for Embedded Systems*, pages 173–181, 2006.

[25] Y.Guo et al. A graph covering algorithm for a coarse grain reconfigurable system. In *Proceedings of the 2003 Conference on Language, Compiler, and Tool for Embedded Systems*, pages 199–208, 2003.

[26] P. Yu and T. Mitra. Characterizing embedded applications for instruction-set extensible processors. In *Proceedings of the 41st Design Automation Conference*, pages 723–728, 2004.