

Code Size Reduction in Heterogeneous-Connectivity-Based DSPs Using Instruction Set Extensions

Partha Biswas, *Student Member, IEEE*, and Nikil D. Dutt, *Senior Member, IEEE*

Abstract—Existing trend of processors shows a progress toward customizable and reconfigurable architectures. In this paper, we study the benefit of combining the architectural design of a VLIW DSP and the concepts of modern customizable processors like ASIPs (Application Specific Instruction Set Processors) for code size reduction. VLIW DSP architectures exhibit heterogeneous connections between functional units and register files for speeding up special tasks. Such architectural characteristics can be effectively exploited through the use of complex instruction set extensions (ISEs). Although VLIWs are increasingly being used for DSP applications to achieve very high performance, such architectures are known to suffer from increased code size. This paper also addresses how to generate and use ISEs that can result in significant code size reduction in VLIW DSPs without degrading performance. Unfortunately, contemporary techniques for generation of ISEs when applied before resource-binding fail to generate legal ISEs for VLIW architectures with heterogeneous connectivity between the functional units and register files. We propose a Heuristic-based approach to generate ISEs for a generalized heterogeneous-connectivity-based VLIW DSP architecture. We achieve an average code size reduction of 25 percent on the MiBench suite with no penalty in performance by applying our ISE generation algorithms on the TI TMS320C6xx, a representative VLIW DSP. We also show that the overhead of the required architectural assists for our approach is minimal: The TMS320C6xx pipeline meets the required timing with only a limited overhead in area.

Index Terms—Coprocessors, ASIP, VLIW, DSP, instruction set extensions, code size reduction.

1 INTRODUCTION

THE embedded application domain is the fastest growing market segment in the microprocessor industry. An application-specific instruction-set processor (ASIP) is particularly suited for embedded applications that have common characteristics such as embedded control or digital-signal processing. These applications demand good performance, low power, and reduced code size. Typically, the datapath of an ASIP is optimized for an application class by addition of special functional units for frequently used operations and elimination of infrequently used units. Unlike Application-Specific Integrated Circuits (ASICs), ASIPs retain the benefit of flexibility through their programmability.

A digital signal processor (DSP) is a well-known class of ASIPs designed to perform common operations—typically math-intensive operations on digital encodings of analog signals. In order to boost performance, many modern DSPs employ VLIW-style architectures that can execute more instructions in parallel. A VLIW DSP, by virtue of having a regular instruction set, presents a compiler-friendly processor model at the cost of larger code size. However, when these processors are embedded on a chip together with instruction memory, the code size has to be limited. We propose a novel algorithm to augment the instruction set of

a VLIW DSP with complex instructions that can significantly reduce the code size. We also introduce architectural assists to provide execution support for the newly added complex instructions. Our algorithm, together with the architectural assists, guarantees that the performance is preserved even with the reduced code size.

We use the term, **Heterogeneous-Connectivity-based DSP** or **HCDSP** for a VLIW DSP architecture with functional units having restricted accesses to register files. The TI TMS320C6xx [1] processor is an example of an HCDSP that issues eight instructions per cycle to eight functional units partitioned into two clusters. Conceptually, the TMS320C6xx architecture contains two register files with any functional unit in each cluster able to access one or both register files based on the connectivity.

There has been a large body of work on generating ISEs for special purpose DSPs. However, in the presence of heterogeneous connectivity between the register files and the functional units, contemporary techniques used for generating ISEs may fail to exploit and generate legal extensions to the instruction set. In this paper, we present a Heuristic-based algorithm that generates new complex instructions for extending the instruction set of HCDSPs with the goal of code size reduction. On a representative architecture (the TI TMS320C6xx [1]), we demonstrate the ability of our algorithm to achieve significant code size reduction (up to 37 percent) over the base instruction set architecture with no loss in performance. In order to seamlessly execute the instruction set extensions, we enhanced the architecture with some architectural assists. We synthesized a prototype design of the assists using

• The authors are with the Center for Embedded Computer Systems, Donald Bren School of Information and Computer Sciences, 444 CS, University of California, Irvine, CA 92697. E-mail: {partha, dutt}@cecs.uci.edu.

Manuscript received 15 Apr. 2004; revised 29 Nov. 2004; accepted 05 Apr. 2005; published online 16 Aug. 2005.

For information on obtaining reprints of this article, please send e-mail to:tc@computer.org, and reference IEEECS Log Number TCSI-0129-0404.

Synopsys Design Compiler and showed that the additional hardware changes do not violate the timing constraints and incur limited area overhead. Thus, we show how to enhance the existing instruction set architecture and the pipeline of TMS320C6xx based on its HCDSP model in order to achieve code size reduction.

The rest of the paper is organized as follows: In Section 2, we discuss related research work. Section 3 presents the motivation for our work. Section 4 demonstrates TMS320C6xx as a HCDSP and discusses our model of an HCDSP architecture. In Section 5, we present our Heuristic-based algorithm to minimize the code size. Section 6 outlines the entire framework and shows the efficacy of our approach on the MiBench suite with discussions on hardware realization of the architectural assists. Finally, Section 7 concludes the paper.

2 RELATED WORK

We discuss related research work in two domains: code size reduction in VLIW DSP processors and application-specific instruction set synthesis.

The hardware solutions ([3], [4], [5], [6], [7]) developed for minimizing code size in VLIW processors mainly focus on changing the instruction formats to incorporate new templates which can lead to compressed code. A typical approach stores instructions in a compressed form in both memory and instruction cache. The instructions are expanded each time they are fetched from cache. The code size reduction thus comes at a cost of increased complexity in the control path. The software solution [8] to the same problem is integrated in a compiler which trades-off code size for performance while scheduling code for the VLIW processor. Our approach is complementary to these previous approaches since we reduce code size by generating ISEs with no penalty in performance.

Several research efforts have studied instruction set synthesis. One of the important steps used in automatic generation of ISEs is **Clustering** of atomic instructions (i.e., instructions that cannot be further subdivided). This clustering step can take two flavors: clustering dependent instructions and clustering parallel instructions. In the context of pipelined RISC processors, a complex instruction obtained by clustering instructions connected through a dependency chain in the data flow graph results in increased performance by exercising forwarding paths between the execution units. This is exemplified by a recent work [11] that generated ISEs for a pipelined RISC processor under bitwidth constraints and demonstrated a significant increase in performance over the native instruction set. The other kind of clustering (that groups independent instructions) was used in architectures containing parallel execution units with the goal of minimizing code size in DSP processors ([12], [13]). In an earlier work [9], application-specific design and integration of an instruction set, a microarchitecture, and an instruction set mapping is proposed for performance. This approach also considers the synthesized instruction to be composed of one or more parallel microoperations. Another related earlier work [10] considers dynamic execution frequency of code segments and synthesizes a complex instruction out of a

cycle's worth of microoperations with the dual goal of code size reduction and performance enhancement. This work assumes that the new data path associated with the new instruction set is designed by hand or using a high-level synthesis system. However, we not only synthesize new ISEs to reduce code size but also study the architectural assists needed to support those ISEs.

The ISEs were also used as specialized instructions in coprocessors, which are extensions to the main processor instruction set. The main goal in ([14], [15], [12], [16], [17], [18], [19]) was to maximize performance of the system in the presence of coprocessor(s) supporting specialized ISEs. Both kinds of clustering were employed in generating ISEs with a bound on the number of read/write ports in the register file.

To the best of our knowledge, no work has yet been done to generate complex instructions for HCDSP processors. A complex instruction in the context of a VLIW processor is an instruction occupying a VLIW instruction slot. If two parallel instructions are combined to form a complex instruction, the number of operands in the generated instruction is the sum of the number of operands in the constituent instructions. In that case, the bitwidth allocated for each instruction slot may not be sufficient to accommodate the new instruction. For example, in TMS320C6xx, a 32-bit instruction format cannot accommodate a six-operand instruction that combines two parallel three-operand instructions. It is important to note here that streaming SIMD extensions on Intel *Itanium*TM [2] are able to pack more data inside one instruction by exploiting subword parallelism. However, in the context of a VLIW processor, we define a complex instruction (that can occupy one of the VLIW slots) as a composition of base instructions connected with each other by a read-after-write dependency chain. Consequently, a complex instruction here by definition generates only one output. A VLIW instruction in our model of TMS320C6xx consists of eight 32-bit instructions, each of which can be a base instruction or a complex instruction.

Due to its heterogeneous connectivity between the register files and the functional units, the HCDSP presents a more generalized model of a VLIW DSP than a simple VLIW architecture. Our goals are to minimize the generated code size as well as to minimize the number of new ISEs generated. The second goal ensures that there is maximum reuse of the ISEs.

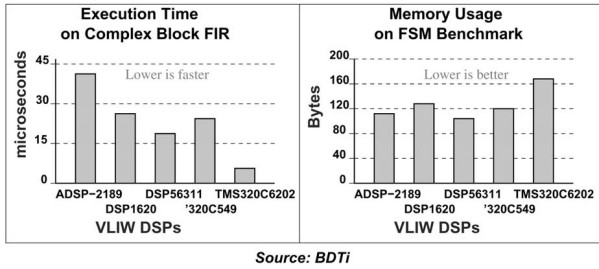
3 MOTIVATION

We now discuss the different factors that motivated our work.

3.1 Poor Code Density in TMS320C6xx

The TMS320C6xx was tuned for DSP applications by matching them with right kind of functional unit. In 0.25um technology, it was very practical to have several multipliers, adders, and other functional units on the chip. The resulting architecture was an HCDSP architecture.

Fig. 1 shows the execution time and memory usage in a representative benchmark for different VLIW DSPs. Out of these DSPs, TMS320C6xx is the only one with an HCDSP



(a) (b)

Fig. 1. VLIW DSPs: (a) Execution time on complex block FIR. (b) Memory usage on FSM benchmark.

architecture and is the fastest. However, it also has the highest program memory bandwidth requirements. If we can extend the instruction set of the TMS320C6xx with useful complex instructions, the memory requirement can be decreased substantially to make the architecture comparable with other VLIW DSPs in terms of memory usage without sacrificing performance.

3.2 Legal Instruction Set Extensions

We present a typical scenario of an HCDSP architecture in Fig. 2. Instructions *add.A1*, *add.A2*, *mul.M1*, and *mul.M2* can only be executed by functional units A1, A2, M1, and M2, respectively. The instruction *add.A2* cannot read registers written by *mul.M1* because M1 can only write into the register file X and A2 can read source operands only from the register file Y. Similarly, *mul.M2* writes into registers which cannot be read by *add.A1* based on the connections of functional units A1 and M2 to register files X and Y. The only legal combinations allowed are: *mul.M1;add.A1* and *mul.M2;add.A2*.

Fig. 3 shows the possible legal and illegal combinations of instructions that are allowed based on the connectivity of functional units to specific register files. Clustering instructions just on the basis of data-flow in an application results in instruction combinations: *mul.M2;add.A1* and *mul.M1;add.A2* that does not have any meaning in the underlying architecture. Therefore, connectivity information is an important parameter in synthesizing valid ISEs for HCDSPs.

3.3 Opportunities for Improvement in TMS320C6xx

In a VLIW DSP architecture, the instructions after dispatch may finish execution at varied time intervals. During the

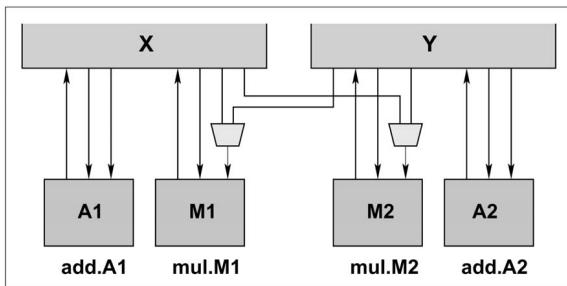


Fig. 2. An example of HCDSP architecture.

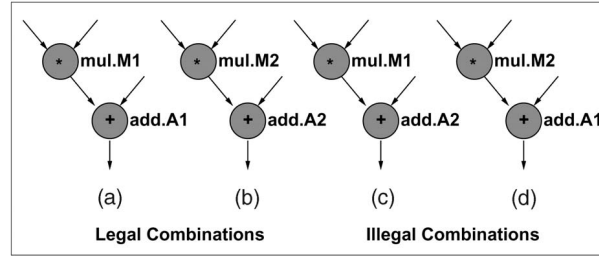


Fig. 3. Possible combinations for a MAC instruction.

execution of an instruction, a functional unit reads the operands from register file(s), performs execution, and writes the result into a register file. We call the difference between the finish time and the dispatch time of an instruction, the **execution latency** of the instruction. After waiting for a time equal to the execution latency of an instruction, the dependent instruction can start execution.

In TMS320C6xx, the execution latencies of multiply and add instructions are two cycles and one cycle, respectively. Applying these values to the sample architecture shown in Fig. 2, we show the pipeline with a *mul.M1* instruction followed by a dependent *add.A1* instruction in Fig. 4a. The instruction *add.A1* begins execution after waiting two cycles for *mul.M1* to finish execution.

Fig. 4b shows the pipeline for *mul.M1;add.A1*, a complex instruction formed by combining *mul.M1* and *add.A1*. It is assumed that the other instruction on which *add.A1* is dependent has been scheduled before *mul.M1*. If the code has been scheduled, there will not be any resource conflict for dispatching *add.A1* after *mul.M1* has finished execution. It is important to note that the execution of *mul.M1;add.A1* terminates exactly at the same point where *mul.M1* followed by *add.A1* finishes. This clearly shows that using *mul.M1;add.A1* compacts two instructions into one without affecting the performance. Therefore, we conclude here that clustering instructions into a complex instruction in a VLIW architecture does not affect performance. The mechanism to break the compact complex instruction into base instructions and separating the dispatch distance between the

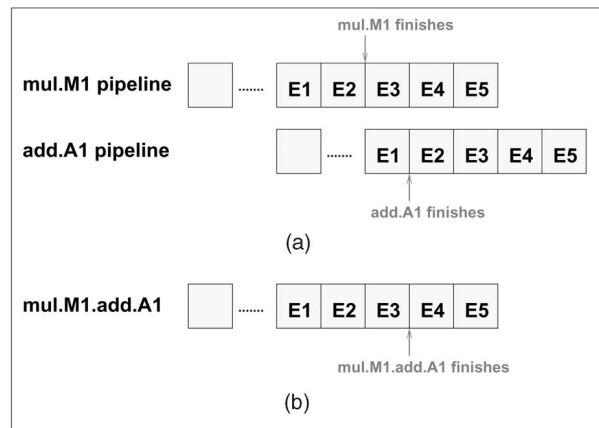


Fig. 4. Instruction pipeline for (a) *mul.M1* instruction followed by *add.A1* instruction. (b) Complex instruction *mul.M1;add.A1*.

TABLE 1
Instruction Formats in TMS320C67x

Units	Instruction Types	Bit Position				
		6	5	4	3	2
L	all	-	-	1	1	0
M	all	0	0	0	0	0
D	most	1	0	0	0	0
D	LD/ST w/ 15-bit off	-	-	-	1	1
D	LD/ST baseR + off	-	-	-	0	1
S	most	-	1	0	0	0
S	ADDK (16 bit cst)	1	0	1	0	0
S	Field ops(imm forms)	-	0	0	1	0
S	MVK, MVKH	-	1	0	1	0
S	Bcond, disp	0	0	1	0	0

two instructions by the execution latency of the first instruction has been dealt with in Section 6.5.

When adding new compact instructions to the existing instruction set, it is also important to consider the available bitwidth to support new instruction formats. Table 1 presents the existing instruction formats in TMS320C67x instruction set.

Each instruction in TMS320C6xx is 32-bit wide, in which five format-select bits (bit positions 6, 5, 4, 3, and 2) are used for selecting one of the 10 existing formats. Each instruction format is associated with one of the four units, namely, L-unit, M-unit, D-unit, or S-unit. There are three types of formats for instructions executing in D-unit, while there are five types of formats for instructions in S-unit. All the instructions executed in L-unit and M-unit have uniform instruction formats. Entries marked “-” indicate that those bit positions are partly used by the opcode field. Using the five format-select bits, it is possible to implement 32 different formats. Therefore, the bitwidth is sufficient to accommodate 22 additional formats. By accommodating the complex instructions in the unused format space, the decoder does not have to be entirely redesigned.

4 HCDSP ARCHITECTURE

With the background presented in Section 1 and Section 3, we present the TMS320C6xx architecture as an example of HCDSP in Section 4.1. Section 4.2 presents a generalized model of the heterogeneous connectivity in an HCDSP architecture.

4.1 TMS320C6xx: An HCDSP

In this paper, we use the TMS320C6xx architecture (shown in Fig. 5) as an exemplar for illustrations and experiments. TMS320C6xx is an 8-issue VLIW machine with two register files A and B, each having 16 32-bit registers. The register file connectivity in the architecture is shown below.

The eight functional units (L1, L2, S1, S2, M1, M2, D1, and D2) in the datapaths can be divided into two groups of four each. A functional unit X1 in one group has an almost identical corresponding unit X2 in the other, where X is L, S, M, or D. Each unit is capable of running one instruction out of an 8-instruction VLIW packet. An instruction run in unit X1 is of the form *mnemonic.X1* and that in unit X2 has the form *mnemonic.X2*.

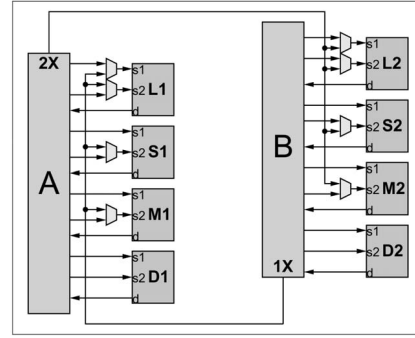


Fig. 5. Heterogeneous connectivity in TI TMS320C6xx architecture.

There are three functional units available for executing eight different instructions. Each VLIW execution packet to be fed to the functional units is decided statically by a compiler targeting TMS320C6xx. A packet consisting of instructions to be executed in parallel should be free of resource conflicts.

4.2 HCDSP Architecture Model

We propose a generalized model of the heterogeneous connectivity in an HCDSP architecture (shown in Fig. 6). The architecture has a set of R register files $\mathcal{X} = \{X_1, X_2, \dots, X_R\}$ and a set of F functional units $\mathcal{Y} = \{Y_1, Y_2, \dots, Y_F\}$. The heterogeneity is in the connection between the register files and the functional units.

The relation between a functional unit $Y_i (1 \leq i \leq F)$ and a register file $X_j (1 \leq j \leq R)$ can be represented as follows:

$$Y_i \vdash X_j \leftarrow \{P_1, \dots, P_m\} \text{ op } \{Q_1, \dots, Q_n\},$$

where \vdash implies Y_i “writes into” X_j , *op* defines the operation to be performed by the instruction, the first source operand is a register in $\{P_1, \dots, P_m\} \subset \mathcal{X}$, the second source operand is a register in $\{Q_1, \dots, Q_n\} \subset \mathcal{X}$, and $1 \leq m, n \leq R$. A base instruction run in unit Y_i is of the form *mnemonic.Y_i*.

We define three functions that are used in deriving the connectivity constraints:

- **Writes(Y_i):** returns the register file written by Y_i based on connectivity.
- **WrittenBy(X_j):** returns a set of functional units that can write into the register file X_j .
- **Binds(Y_i):** returns a set of operations bound to Y_i .

In order to legally combine two instructions, the following connectivity constraint must be obeyed:

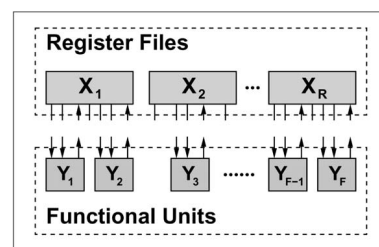


Fig. 6. Heterogeneous connectivity model.

An instruction $i1.X$ can be combined with another instruction $i2.Y$ dependent on the former through a source “ s_i ,” where $i = 1$ or 2 and $X, Y \in \mathcal{Y}$, iff there exist paths from output “ d ” of X to a register file and from the same register file to input “ s_j ” of Y , where $j = 1$ or 2 . (The output port “ d ” and the input ports “ s_1 ” and “ s_2 ” are shown in Fig. 6.) We represent the resultant complex instruction as $i1.X;i2.Y$. The complex instruction employs a bypass latch instead of a register to store the intermediate result.

Using this formulation of the HCDSP architecture model, we show the derivation of connectivity constraints in Section 5.1.

5 OUR APPROACH

Before presenting our ISE generation algorithm (in Section 5.4), we first discuss the derivation of connectivity constraints from the generalized HCDSP model in Section 5.1. Then, we enumerate further important considerations for generating no-penalty complex instructions in Section 5.2 and Section 5.3.

5.1 Derivation of Connectivity Constraint

Our goal is to compact 3-operand instructions of the form I1: $x = a \text{ op1 } b$ and I2: $y = x \text{ op2 } c$ into a 4-operand instruction, $y = (a \text{ op1 } b) \text{ op2 } c$, where x and y are register operands; a , b , and c can be register or immediate operands; and op1 and op2 are operators. The operations op1 and op2 to be executed by instructions I1 and I2, respectively, are bound to functional units Y_1 and $Y_2 \in \mathcal{Y}$. For the model presented in Fig. 6, the connectivity constraint for operation in I2 is as follows:

$$Y_2 \vdash \text{Writes}(Y_2) \leftarrow \{P_1, \dots, P_m\} \text{ op2 } \{Q_1, \dots, Q_n\}.$$

The set of functional units that can write into operands $\{P_1, \dots, P_m\}$ is given by:

$$Y_P = \bigcup_{i=1}^m \text{WrittenBy}(P_i).$$

Similarly, the set of functional units that can write into operands $\{Q_1, \dots, Q_n\}$ is given by:

$$Y_Q = \bigcup_{i=1}^n \text{WrittenBy}(Q_i).$$

It follows that the legal data flow across functional units for execution of I2 in Y_2 can be expressed as:

$$Y_2 \leftarrow (Y_P, Y_Q).$$

In other words, the instruction I2 executing in functional unit Y_2 can legally get the first and the second source operands from the outputs of any of the functional units in Y_P and Y_Q , respectively. Thus, instruction I2 can be coupled as a second instruction only with any instruction in $\text{Binds}(Y_P)$ and $\text{Binds}(Y_Q)$, supplying first and second sources, respectively, for I2. This expresses the connectivity constraint for instruction I2 under the given HCDSP model. The TMS320C6xx architecture fits into the HCDSP model with $\mathcal{Y} = \{L1, L2, S1, S2, M1, M2, D1, D2\}$ and $\mathcal{X} = \{A, B\}$.

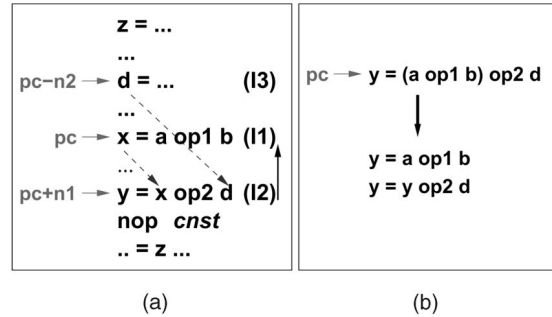


Fig. 7. Instruction sequence: (a) Candidate pair {I1,I2}. (b) Complex instruction generated from {I1,I2}.

5.2 Bit-Width Constraint

One of the most important considerations for instruction set synthesis is the bit-width available for representation of complex instruction formats. In TMS320C6xx, each operand field consumes 5 bits for accessing 32 registers (16 registers in either of the two register files). Therefore, with a 5-bit wide format-select field and an opcode field varying in length depending on the instruction, it is feasible to include only up to four operands in the instruction format. This essentially means that our algorithm should look for opportunities to combine two dependent instructions, each having at the most three operands combined together to form a complex instruction having four operands.

5.3 Latency Constraints

While the connectivity constraints help prune illegal combinations, latency constraints are important for preserving the performance of the VLIW DSP architecture. Fig. 7 shows a sequence of base instructions that are subject to the following dependency and latency considerations for valid combinations:

- *Primary Constraint:* Within a basic block, an instruction I1 at pc and another dependent instruction I2 at $(pc + n1)$ can be combined to form a complex instruction if I2 does not have a second source operand or the second source is dependent on an instruction, I3 at $(pc - n2)$, where $n1, n2 > 0$. (Refer to Fig. 7a) We call this a dependency constraint and the pair {I1,I2} satisfying dependency constraint a candidate pair. An arrow in Fig. 7a shows that the instruction I2 goes up to combine with the instruction I1 when selected in a cluster. All the subsequent considerations are latency constraints.
- If $(\text{latency}(I3) \leq (n2 + \text{latency}(I1)))$, only then it is legal to combine I1 and I2 so that I2 gets the result of I3 well in time (Fig. 7a). Note here that if {I1,I2} is executed as a complex instruction at pc , the second operand of I2 will be fetched with a delay. This is important for preserving the performance.
- Fig. 7b shows the execution semantics of the complex instruction obtained by combining instructions I1 and I2 shown in Fig. 7a). If the result x produced by I1 is used in an instruction at $(pc + n3)$ such that $n3 > 0$, then I1 cannot be combined with I2 because x does not exist in the complex instruction.

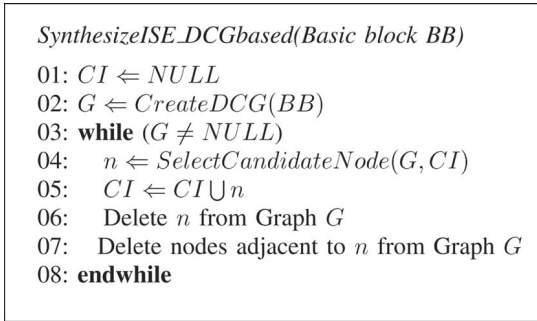


Fig. 8. DCG-based ISE generation algorithm.

In other words, in this execution model, the intermediate output of a complex instruction is never available to subsequent instructions.

5.4 ISE Generation Algorithm

We first define a Restricted Data Dependence Graph (RDDG) as $G^{RDDG}(N, E)$, where each node $\in N$ is an instruction and an edge $\in E$ exists between two nodes only if the pair of nodes satisfies the latency constraints. The RDDG is effectively a restricted subset of the Data Dependence Graph (DDG). Each data dependence edge between two instructions $I1$ and $I2$ in RDDG is potentially a complex instruction combining $I1$ and $I2$ through one of the sources of $I2$. Two data dependencies conflict when using one dependency as a complex instruction invalidates the possibility of the other becoming a complex instruction.

We now introduce the notion of Dependence Conflict Graph (DCG), which is the heart of our algorithm. A DCG is a graph $G^{DCG}(N^d, E^c)$, where N^d is a set of nodes representing data dependencies and E^c is a set of edges connecting two nodes with conflicting dependencies. Fig. 12a illustrates a DCG generated from an RDDG shown in Fig. 9a.

A **greedy** solution to the problem would consider sequentially each unmarked instruction I in a basic block and test the possibility of legally and profitably combining I with each instruction $I1 \in DUChain[I]$ ($DUChain$ and $UDChain$ refer to Def-Use chain and Use-Def chain, respectively) into a complex instruction. The legality is ensured by testing the connectivity and dependency constraints (as discussed in Section 5.1 and Section 5.3). If the test evaluates to true, the greedy approach would immediately add the pair $\{I, I1\}$ to a set representing generated complex instructions and mark the constituent base instructions. The consideration of only unmarked instructions for combination allows a valid replacement by the generated complex instruction. For example, consider the RDDG shown in Fig. 9a. When an edge is chosen, the corresponding nodes combine to form a complex instruction. In our discussion, we don't show the satisfaction of the connectivity constraint explicitly, but it acts as a decisive step in choosing the final instruction combination. The greedy algorithm can first choose edge 1 or 2 as both are legal in terms of the dependency constraint. It randomly picks edge 2, i.e., instruction $\{I1, I4\}$, and marks the corresponding nodes. The next instruction in sequence is $I2$, whose combination with instruction $I5$ (through edge 6)

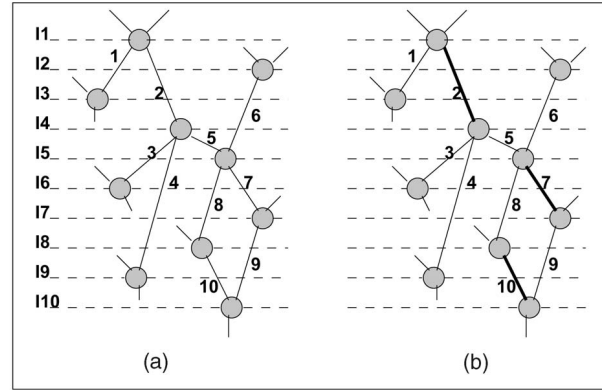


Fig. 9. Running greedy algorithm. (a) Restricted data dependence graph (RDDG). (b) Greedy solution.

violates the dependency constraint. The subsequent instruction $I3$ cannot be combined with $I1$ because $I1$ has already been marked. The next legal choice is edge 7, i.e., instruction $\{I5, I7\}$. Proceeding in this way, the greedy algorithm ends up choosing edges 2 ($\{I1, I4\}$), 7 ($\{I5, I7\}$), and 10 ($\{I8, I10\}$), as shown in Fig. 9b through bold edges. Note that the order in which the greedy algorithm examines the instructions is determined by the sequence of instructions in the basic block, which may not be optimal. For instance, edge 1 is preferable to edge 2 for achieving maximum code size reduction because choosing edge 2 discards the possibilities of choosing edges 1, 3, 4, and 5, while choosing edge 1 discards only edge 2. So, we propose a better Heuristic-based ISE Generation approach which employs the DCG representation to circumvent the problem inherent in the greedy approach. We call this approach a **DCG-based** approach.

Fig. 8 (*SynthesizeISE_DCGbased* procedure) outlines the DCG-based approach, which has two main objectives: to maximize the reduction in code-size without affecting the performance and to minimize the number of new complex instructions generated. Finding the Maximum Independent Set (MIS) for G^{DCG} can yield the solution to the first objective—getting maximum code size reduction without hampering the performance. Unfortunately, MIS is known to be NP-complete [20]. So, any heuristic employed for getting a maximal solution should pay attention to the second objective, i.e., minimizing the number of new instructions generated. Our DCG-based algorithm, shown as Fig. 8, strives to meet both the objectives. The algorithm uses *SatisfyConnDepCons* procedure to determine if two instructions i and j can be legally combined based on the dependency and connectivity constraints (as explained in Section 4). It also uses *SatLatConstraints* procedure for evaluating the latency constraints which ensures a profitable composition (as discussed in Section 5.3).

The algorithm, embedded in *SynthesizeISE_DCGbased* procedure, starts by creating the DCG from the DDG using *CreateDCG* procedure (Fig. 10). By checking whether the latency constraints are satisfied, this procedure effectively generates the DCG from the RDDG. The algorithm then calls *SelectCandidateNode* procedure (Fig. 11), which selects a candidate node representing a profitable complex instruction to be added to a growing list called CI . The

```

CreateDCG(Basicblock BB)
01:  $N \leftarrow E \leftarrow NULL$ 
02: foreach  $I \in Instructions[BB]$ 
03:   foreach  $J \in DUChain[I]$ 
04:     if ( $SatLatConstraints(I, J, dep[I, J])$ )
05:       /*  $\{I, J\}$  represents a complex instr */
06:        $ciNode \leftarrow CreateNode(G, \{I, J\})$ 
07:        $Ins1[ciNode] \leftarrow I; Ins2[ciNode] \leftarrow J$ 
08:        $sn[ciNode] \leftarrow SrcNum(dep[I, J], J)$ 
09:        $N \leftarrow N \cup ciNode$ 
10:     endif
11:   endfor
12: endfor
13: foreach  $n \in N$ 
14:    $I_1 \leftarrow Ins1[n]; I_2 \leftarrow Ins2[n]$ 
15:   foreach  $I \in UDChain[I_1] \cup UDChain[I_2]$ 
16:     if ( $Node[\{I, I_1\}] \in N$ )
17:        $e \leftarrow CreateEdge(G, n, Node[\{I, I_1\}])$ 
18:     elseif ( $Node[\{I, I_2\}] \in N$ )
19:        $e \leftarrow CreateEdge(G, n, Node[\{I, I_2\}])$ 
20:     endif
21:    $E \leftarrow E \cup e$ 
22: endfor
23: foreach  $I \in DUChain[I_1] \cup DUChain[I_2]$ 
24:   if ( $Node[\{I_1, I\}] \in N$ )
25:      $e \leftarrow CreateEdge(G, n, Node[\{I_1, I\}])$ 
26:   elseif ( $Node[\{I_2, I\}] \in N$ )
27:      $e \leftarrow CreateEdge(G, n, Node[\{I_2, I\}])$ 
28:   endif
29:    $E \leftarrow E \cup e$ 
30: endfor
31: endfor
32: return  $G$ 

```

Fig. 10. Creation of DCG.

SelectCandidateNode procedure selects a node in DCG with minimum degree in order to maximize the chances of combination to form complex instruction. For instance, in Fig. 12a, selecting node 2 (with degree = 4) as a complex instruction invalidates the possibilities of nodes 3, 4, and 5 to be considered as complex instructions, while choosing node 1 (having degree = 2) jeopardizes the consideration of only node 2. The *SelectCandidateNode* also checks for the dependency and connectivity constraints to ensure a legal combination. When there are more than one nodes having the minimum degree, the heuristic breaks the tie by selecting the node having the complex instruction that has already been encountered before and increments its frequency. This guarantees maximum reuse of the selected complex instruction. The selected node and all its adjacent nodes are then deleted from the DCG. The process of selection and deletion is continued until the graph becomes empty. The list CI finally contains the nodes representing newly generated complex instructions with frequencies of their occurrence stored in *Frequency*. For illustration, consider the DCG in Fig. 12a derived from Fig. 9a. The DCG-based algorithm first chooses node 1 having the minimum degree in the DCG and adds to an empty list, CI. The node 1, its adjacent node 2, and their corresponding

```

SelectCandidateNode(Graph G, Set CI)
01:  $N \leftarrow$  Set of nodes with minimum degree
02: foreach  $n \in N$ 
03:   if ( $\exists n_1 \in CI$ )
04:     s.t.  $CmplxInstr[n_1] = CmplxInstr[n]$ 
05:      $Frequency[CmplxInstr[n]]++$ 
06:   return  $n$ 
07:   else
08:     if ( $SatisfyConnDepCons$ 
09:       ( $Ins1[n], Ins2[n], sn[n]$ ))
10:       return  $n$ 
11:     else
12:       return  $NULL$ 
13:     endif
14:   endif
15: endfor

```

Fig. 11. Selection of profitable candidate.

edges are then removed from the DCG. Out of the minimum-degree (= 2) nodes, 3, 4, and 10, that do not violate the dependency constraint, node 3 is chosen based on *Frequency* and added to CI. Nodes 4 and 5 are then removed from the graph. Continuing in this way, subsequently, node 10 and then node 7 are added to CI. The nodes in CI map to edges 1, 3, 7, and 10 in the RDDG shown in Fig. 12b as bold edges. Thus, the complex instructions generated are: 1 ($\{I1, I3\}$), 3 ($\{I4, I6\}$), 7 ($\{I5, I7\}$), and 10 ($\{I8, I10\}$).

The running time of *SynthesizeLSE_DCGbased()* is $O(n^2)$, where n is the number of instructions. The algorithm is integrated with other complex phases of the compiler such as Instruction Selection and Scheduling. Therefore, the time taken to perform generation of new instructions is dominated by the time taken by the other, more complex phases.

6 EXPERIMENTAL RESULTS

We integrated our algorithm in the EXPRESSION [22] framework used for generating a retargetable compiler and simulator. We extracted from the EXPRESSION model of TI TMS320C6xx, the instruction latencies used for deriving latency constraints, and the connectivity model used for

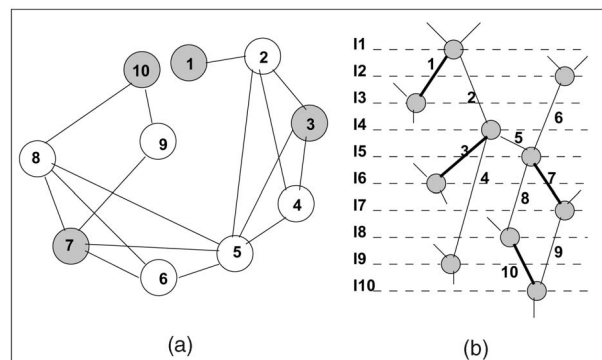


Fig. 12. Running DCG-based algorithm. (a) Dependence conflict graph. (b) DCG-based solution.

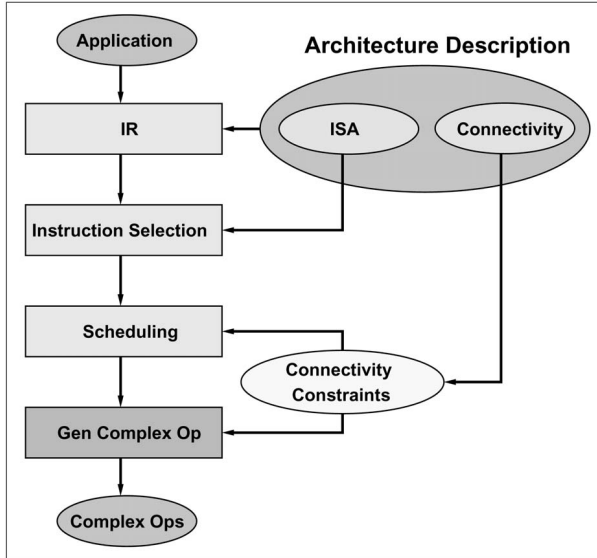


Fig. 13. Our methodology.

deriving the connectivity constraints. For our purpose, we modified the retargetable compiler to explore opportunities for compacting instructions.

6.1 Methodology

Fig. 13 shows the flow of our framework. The target architecture is specified in terms of datapath connectivity and instruction set architecture (ISA) from which the instruction latencies and the connectivity model are derived. An input application is converted into an Intermediate Representation (IR) suitable for compiler optimizations. The IR is in Static Single Assignment (SSA) form so that there are only Read-After-Write (RAW) dependencies. The instruction selection phase transforms each generic instruction into all possible target instructions. For example, an integer multiplication operation is mapped to MPY.M which encompasses two target instructions, MPY.M1 and MPY.M2. The instruction scheduler schedules the target instructions and also binds them to possible functional units based on the available resources. Note here that even after the code is scheduled, the resource binding is not yet complete. For example, a scheduled instruction MPY.M is bound to both units M1 and M2, necessitating a connectivity-based analysis to be done later in *Gen Complex Op* phase to prune out illegal combinations. The scheduler ensures that the instructions in a candidate pair (as defined in Section 5.2) are separated by the latency distance dictated by the first instruction in the pair. This guarantees that the resource(s) used by complex instruction representing the candidate pair does not conflict with subsequent instructions. The *Gen Complex Op* phase (Fig. 13) takes the scheduled code as input and generates new complex instructions, which are legal and profitable combinations of the base instructions.

6.2 Code Size Reductions

We conducted our experiments on MiBench [23] benchmarks from the University of Michigan. The results are collectively presented in Table 2 and Fig. 14. The leftmost

TABLE 2
Experimental Results on MiBench Benchmarks

Application Programs	# Base Instructions	Greedy Approach		DCG-based Approach	
		# Inst	% CS Redn	# Inst	% CS Redn
FFT	736	179	24	181	24
crc_32	150	35	23	36	24
adpcm	417	75	17	75	17
gsm	9855	2763	28	2806	28
pgp	40886	7750	18	8107	19
rijndael	3674	1378	37	1385	37
blowfish	3015	827	27	897	29
sha	410	109	26	110	26
mad	13230	3274	24	3369	25
gif2tiff	36983	8473	22	8814	23
jpeg	32827	8602	26	8756	25
susan	7232	2065	28	2217	30
qsort	247	49	19	49	19
bitcount	612	166	27	169	27
basicmath	994	171	17	172	17
sphinx	56097	12720	22	13170	23
rsynth	6318	1378	21	1408	22
stringsearch	997	274	27	282	28
dijkstra	292	75	25	76	26

column in the table shows the benchmarks (*Application Programs*) in the order of different areas in Embedded applications: Telecommunications, Security, Consumer, Automotive and Industrial Control, Office Automation, and Network applications. The next column shows the number of base instructions in memory (*# Base Instructions*). The subsequent columns together present the efficacy of using ISEs (generated by greedy or DCG-based approach). The metrics used are the number of instances when a complex instruction replaces two base instructions (*# Instances*) and the percentage code size reduction (*percent CS Reduction*). The percentage code size reduction is calculated as follows:

$$\%Impr = (\#Inst/\#BaseIns) \cdot 100.$$

On an average, the DCG-based algorithm achieves 25 percent reduction in code size, 1 percent more than that obtained by the greedy algorithm. Fig. 14 shows that the DCG-based algorithm also results in fewer new complex instructions than the greedy algorithm. Thus, using the DCG-based algorithm, the base instruction set needs

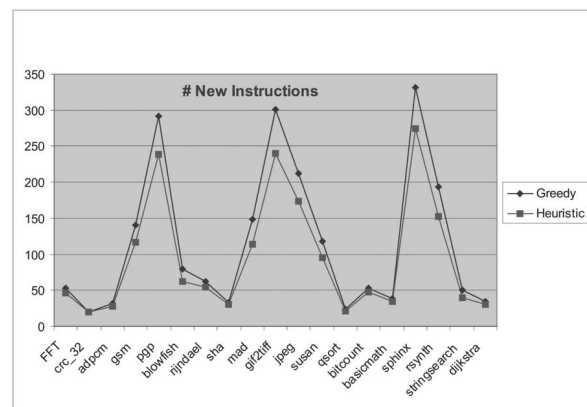


Fig. 14. Comparison of the number of new complex instructions generated.

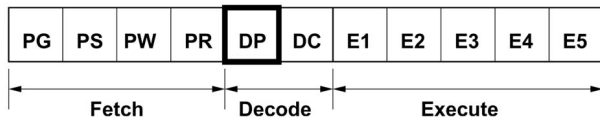


Fig. 15. Pipeline stages in TI TMS320C6xx.

augmentation with fewer new instructions and, at the same time, the augmented instruction set achieves more code size reduction than that obtained using the greedy algorithm.

The ISE generation algorithm is restricted to finding complex instructions within a basic block. Therefore, the chances of combination are higher in applications having larger basic blocks, as is demonstrated in *rijndael* benchmark. The largest benchmark is *sphinx* having 56,097 instructions from the base instruction set. For this application, the DCG-based algorithm generates 274 new instructions, which, when used as extensions to the instruction set, yields 23 percent reduction in code size. The total number of new instructions generated for all the benchmarks was 523. Note that we can easily incorporate the newly generated complex instructions into the existing instruction set since there is sufficient unused encoding space for adding 22 more instruction formats, each accommodating up to 32 instructions totaling 704. Some of the examples of the generated complex instructions are *MPYDP.M1;ADDDP.L2*, *LDDW.D1;ADDSP.L1*, *AND.S2;ADD.S1*, etc.

6.3 Impact of Our Work

With an average 25 percent improvement in code size as obtained by the heuristic-based algorithm, the memory usage of TMS320C6xx (illustrated by the tall bar for the TMS320C6xx in Fig. 1b) becomes comparable with the other VLIW DSPs (i.e., the same level as the other DSPs in Fig. 1b).

The performance of a VLIW DSP is essentially attributed to the Instruction Scheduler of the compiler. The opportunities to find legal combinations of instructions are affected by the work done by the scheduler: If fewer VLIW slots are utilized by the Instruction Scheduler, then there are more opportunities for combination. This enables us to do a trade-off between the performance achievable by optimally scheduled instructions and the code-size reduction obtainable by exploiting the opportunities of using complex instructions. A complex instruction utilizes one register less than the number of registers used by the constituent base instructions. Consequently, there is an overall reduction in register pressure for every combination of base instructions. Therefore, it is likely that a spilled code can be freed of spilling just by efficiently combining instructions. As a result, the performance cannot degrade but, at the best, can increase.

One might argue that adding complex instructions to a regular instruction set of a VLIW machine can lead to an increase in compiler complexity. In that context, it is important to note that the same algorithm for generating complex instructions can be used to generate code for a VLIW machine having these complex instructions. This algorithm can be added as a back-end to a compiler targeting an HCDSP architecture like TMS320C6xx having a regular instruction set.

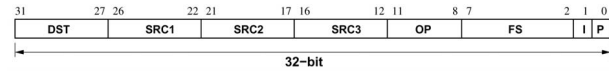


Fig. 16. Instruction format for ISE.

6.4 Modified Pipeline

Our approach requires a small modification to the original pipeline of the TI TMS320C6xx due to the introduction of complex instructions into the instruction set. The TI TMS320C6xx pipeline consists of fetch, decode, and execute stages without any pipeline interlocks.

As shown in Fig. 15, the fetch stage (composed of Program Address Generate (PG), Program address Send (PS), Program Access Ready Wait (PW), and Program Fetch Packet Receive (PR) phases) takes four cycles; the decode stage (containing Instruction Dispatch (DP) and Instruction Decode (DC) phases) takes two cycles; and the execute stage (containing E1 to E5 phases) takes one to five cycles depending on the type of instruction. During the DP phase of decode, the instructions in an execute packet are assigned to appropriate functional units for execution. In the DC phase, the source registers, destination registers, and associated paths are decoded for execution in the functional units.

We propose modification in the DP phase to include the following tasks:

- The simple instructions in the execute packet are assigned to appropriate functional units in the next clock cycle.
- The complex instructions are split into the constituent base instructions by referring to a look-up table.
- The first base instruction is assigned to the appropriate functional unit in the next clock cycle. Once the execution of the first instruction has finished, the second base instruction is assigned to the appropriate functional unit in the subsequent clock cycle.

These small changes to the pipeline nevertheless introduce an overhead (potentially for delay and area); we discuss the impact of the overhead and show that this impact is minimal in the following section.

6.5 Architectural Assists

Fig. 16 shows a possible instruction format for the TMS320C6xx ISE.

The operands are 5-bit wide for accessing a total of 32 registers in register-files A and B. There are 64 combinations of units possible for an ISE containing two base instructions each accessing one of the eight functional units. The 6-bit wide format-select (FS) field essentially accounts for these 64 combinations and each format can accommodate up to 16 different instructions through a 4-bit opcode (OP) field. The I-bit determines whether the instruction is an ISE or a simple instruction. Abiding by the original instruction format, the P-bit determines whether the instruction (complex or simple) executes in parallel with the next instruction in the fetch-packet. Depending on the P-bit, one to eight instructions from the fetch-packet can be dispatched to the functional units in parallel.

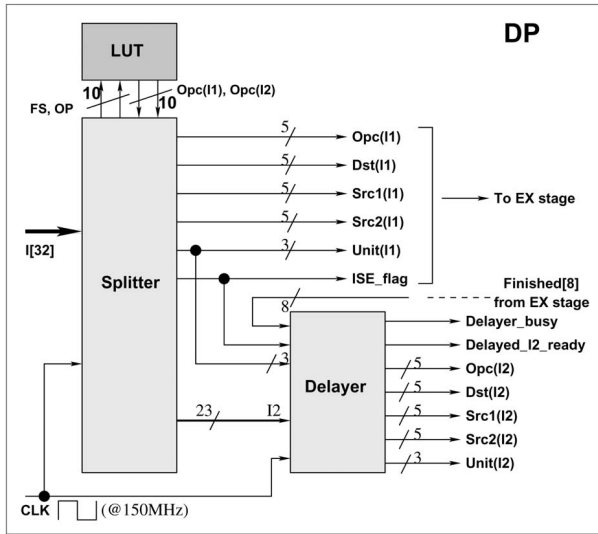


Fig. 17. DP-phase in the decode stage of the pipeline in Fig. 15.

The only modification to the pipeline occurs in the **DP** phase of the decode stage as shown in Fig. 15. Fig. 17 shows the schematic of a hardware implementation for the architectural assists used in the modified **DP** phase. Each instruction I in the execute packet is assigned to a **Splitter** unit. The **Splitter** checks the I -bit to distinguish between simple and complex instructions. If the instruction is simple, it assigns the instruction (as $I1$) to the respective functional unit at the rising edge of the clock, CLK . If the instruction is complex (ISE), it splits I into $I1$ and $I2$ based on a look-up table, **LUT**. In order to map an ISE, I into constituent base instructions $I1$ and $I2$, the **Splitter** sends the FS and OP fields of the instruction to **LUT**. The **LUT** contains the mappings of upto 1,024 complex instructions into their base opcodes. The units corresponding to the base opcodes are also encoded in FS . The first base opcode along with the other fields extracted from I (together designated as $I1$) are assigned to the appropriate functional unit at the positive edge of the clock. A signal ISE_flag is used to notify the corresponding EX unit about the complex instruction.

Each EX unit is identified by a unit number ranging between 1 and 8. The EX unit, upon completion of execution sets a bit corresponding to its unit number in the $finished[8]$ signal. The second base instruction ($I2$) was sent to a **Delayer** unit to wait for the execution of $I1$ to finish. If $Finished[Unit(I1)] = 1$, the **Delayer** releases the second base instruction $I2$ to the appropriate functional unit at the next positive edge of the clock. The signals $Delayer_busy$ and $Delayed_I2_ready$ clearly indicate the state of the **Delayer**.

In order to evaluate the overhead of these architectural assists, we synthesized the logic in Fig. 17 using Synopsys Design Compiler for a 1.0 μm LSI_{10K} library. Our design of the DP phase successfully met the timing constraints with a 150 MHz clock (CLK) and the total cell area reported was only 874 gates. Note, however, that the technology we used for synthesis was an older 1.0 μm technology, whereas the TMS320C62x used a 0.18 μm technology. In spite of using an older, more conservative technology, we still met the delay constraints. Of course, we expect that the delay

overhead and the cell area will improve for the 0.18 μm technology. Therefore, we established that the hardware overhead imposed by the architectural assists in our technique does not affect performance and incurs only a small overhead in area.

7 SUMMARY AND FUTURE DIRECTIONS

We presented a scheme for reducing code size in a Heterogeneous-Connectivity-based DSP (HCDSP) architecture by enhancing its Instruction Set Architecture (ISA) and incorporating new architectural assists in its pipeline. We also proposed a Heuristic-based algorithm to identify Instruction Set Extensions (ISEs) to be added to the ISA. We applied our scheme to a well-known HCDSP (TI TMS320C6xx) by modeling its architecture in a retargetable framework. The connectivity and latency constraints were derived from the architecture description. Based on these constraints and the dependency constraints derived from the application, our framework was able to generate profitable complex instructions as extensions to the existing instruction set. The generated ISEs were able to achieve an average code size reduction of 25 percent without losing any performance. We also showed that the overhead incurred by the architectural assists is minimal: We synthesized a prototype model of the assists using Synopsys Design Compiler for 150 MHz TMS320C62x and verified that the design still met the timing requirement. Our future work will study the implication on power and performance through the generation of ISEs targeting different embedded domains.

ACKNOWLEDGMENTS

This work was supported in part by grants from the US National Science Foundation (CCR 0203813 and CCR 0205712). The authors would like to thank Jong-eun Lee and other members of the ACES laboratory (<http://www.ics.uci.edu/~aces>) for their input.

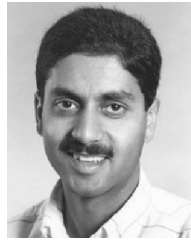
REFERENCES

- [1] TI TMS320C6xx User Manual, <http://www.ti.com>, 2005.
- [2] Intel Itanium Processor, <http://developer.intel.com/design/itanium/manuals.htm>, 2005.
- [3] S. Hanono and S. Devadas, "Instruction Selection, Resource Allocation and Scheduling in the AVIV Retargetable Code Generator," *Proc. Design Automation Conf.*, pp. 510-515, 1998.
- [4] T.M. Conte, S. Banerjia, S.Y. Larin, K.N. Menezes, and S.W. Sathaye, "Instruction Fetch Mechanisms for VLIW Architectures with Compressed Encodings," *Proc. 29th Int'l Symp. Microarchitecture*, pp. 201-211, 1996.
- [5] S. Aditya, S.A. Mahlke, and B.R. Rau, "Code Size Minimization and Retargetable Assembly for EPIC and VLIW Instruction Formats," Technical Report PL-2000-141, HP Labs, 2000.
- [6] M. Kozuch and A. Wolfe, "Compression of Embedded System Programs," *Proc. Int'l Conf. Computer Design*, pp. 270-277, 1994.
- [7] S.Y. Liao, S. Devadas, and K. Keutzer, "Code Density Optimization for Embedded DSP Processors using Data Compression Techniques," *IEEE Trans. Computer Aided Design*, vol. 17, no. 7, pp. 601-608, 1998.
- [8] H. Zhou and T.M. Conte, "Code Size Efficiency in Global Scheduling for ILP Processors," *Proc Sixth Ann. Workshop Interaction between Compilers and Computer Architectures*, 2002.

- [9] I.-J. Huang and A.M. Despain, "Generating Instruction Sets and Microarchitectures from Applications," *Proc. Int'l Conf. Computer-Aided Design*, 1994.
- [10] B.K. Holmer, "A Tool for Processor Instruction Set Design," *Proc. Conf. European Design Automation*, 1994.
- [11] J.-E. Lee, K. Choi, and N. Dutt, "Efficient Instruction Encoding for Automatic Instruction Set Design of Configurable ASIPs," *Proc. Int'l Conf. Computer Aided Design*, 2002.
- [12] H. Choi, J.-S. Kim, C.-W. Yoon, I.-C. Park, S.H. Hwang, and C.-M. Kyung, "Synthesis of Application Specific Instructions for Embedded DSP Software," *IEEE Trans. Computers*, 1999.
- [13] R. Leupers and P. Marwedel, "Instruction Selection for Embedded DSPs with Complex Instructions," *Proc. European Design Automation Conf.*, 1996.
- [14] F. Onion, A. Nicolau, and N. Dutt, "Compiler Feedback in ASIP Design," *Proc. Conf. Design, Automation, and Test in Europe*, 1995.
- [15] M. Arnold and H. Corporaal, "Instruction Set Synthesis Using Operation Pattern Detection," *Proc. Fifth Ann. Conf. Advanced School for Computing and Imaging*, 1999.
- [16] F. Sun, S. Ravi, A. Raghunathan, and N.K. Jha, "Synthesis of Custom Processors Based on Extensible Platforms," *Proc. Int'l Conf. Computer Aided Design*, 2002.
- [17] D. Goodwin and D. Petkov, "Automatic Generation of Application Specific Processors," *Proc. Int'l Conf. Compilers, Architectures, and Synthesis for Embedded Systems*, 2003.
- [18] K. Atasu, L. Pozzi, and P. Ienne, "Automatic Application-Specific Instruction-Set Extensions under Microarchitectural Constraints," *Proc. Design Automation Conf.*, 2003.
- [19] P. Biswas, S. Banerjee, N. Dutt, L. Pozzi, and P. Ienne, "ISEGEN: Generation of High-Quality Instruction Set Extensions by Iterative Improvement," *Proc. Design, Automation and Test in Europe*, 2005.
- [20] R.M. Karp, "Reducibility Among Combinatorial Problems," *Complexity of Computer Computations*, Plenum Press, 1972.
- [21] S.S. Muchnick, *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.
- [22] A. Halambi, P. Grun, V. Ganesh, A. Khare, N. Dutt, and A. Nicolau, "EXPRESSION: A Language for Architecture Exploration through Compiler/Simulator Retargetability," *Proc. Conf. Design, Automation and Test in Europe*, 1999.
- [23] M.R. Guthaus, J.S. Ringenberg, D. Ernst, T.M. Austin, T. Mudge, and R.B. Brown, "MiBench: A Free Commercially Representative Embedded Benchmark Suite," *Proc. IEEE Sixth Ann. Workshop Workload Characterization*, <http://www.eecs.umich.edu/jringenb/mibench/>, Dec. 2001.



Partha Biswas received the BTech degree in computer science and engineering from the Indian Institute of Technology, Kharagpur, in 1998 and the MS degree in information and computer science from the University of California, Irvine, in 2002, where he is currently pursuing the PhD degree. He has served in the positions of software engineer and member of technical staff at Cadence Design Systems, Noida, India, from 1998 to 2000. His primary interests include architectures and compilers for embedded systems and application-specific processor design. He is a student member of the IEEE.



Nikil D. Dutt (SM) received the PhD degree in computer science from the University of Illinois at Urbana-Champaign in 1989 and is currently a professor of CS and EECS at the University of California, Irvine. He is affiliated with the following centers at UCI: CECS, CPCC, and CAL-IT2. His research interests are in embedded systems design automation, computer architecture, optimizing compilers, system specification techniques, and distributed embedded systems. He received best paper awards at CHDL 89, CHDL 91, VLSIDesign 2003, and CODES+ISSS 2003. Professor Dutt currently serves as editor-in-chief of *ACM Transactions on Design Automation of Electronic Systems (TODAES)* and as associate editor of *ACM Transactions on Embedded Computer Systems (TECS)*. He was an ACM SIGDA Distinguished Lecturer during 2001-2002, and is currently an IEEE Computer Society Distinguished Visitor for 2003-2005. He has served on the steering, organizing, and program committees of several premier CAD and embedded system design conferences and workshops, including ASPDAC, CASES, CODES+ISSS, DATE, ICCAD, ISLPED, and LCTES. He is a senior member of the IEEE, serves on the advisory boards of ACM SIGBED and ACM SIGDA, and is vice chair of IFIP WG 10.5.

► For more information on this or any other computing topic, please visit our Digital Library at www.computer.org/publications/dlib.