Synthesis of Application Specific Instructions for Embedded DSP Software

Hoon Choi, In-Cheol Park, Seung Ho Hwang and Chong-Min Kyung
Department of Electrical Engineering,
Korea Advanced Institute of Science and Technology, Taejon, Korea
hchoi@snoopy.kaist.ac.kr, icpark@ee.kaist.ac.kr

Abstract
Application specific instructions play an important role in reducing the required code size and increasing performance. This paper describes a new approach to generate application specific instructions for DSP applications. The proposed approach is based on a modified subset-sum problem, and can support multi-cycle complex instructions as well as single cycle instructions, while the previous state-of-the-art approaches can generate only the single-cycle instructions or can just select instructions from the fixed super-set of possible instructions. In addition, the proposed approach can also be applicable to the case that instructions are predefined. The experimental results on real applications show that the proposed approach is effective in making the instructions meet the given constraints without attaching special hardware accelerators.

1 Introduction
Due to the advance of VLSI technology, a lot of ASIC’s (Application Specific Integrated Circuit) are being used in various systems. Compared to the general purpose processors, an ASIC can satisfy various constraints such as performance, area, and power by finding an optimal architecture for an application. However, as the complexity of applications increases, more flexibility is required to accommodate design errors and specification changes which may happen at the later design stages. Since ASIC is specially designed for one behavior, it is difficult to adopt any changes at the later design stage. In contrast, a programmable processor can be easily adapted to different applications by changing only the programs. This is the reason why ASIP’s (Application Specific Instruction set Processor) are widely accepted in numerous systems.

Generally an ASIP has a programmable architecture which is tuned to a few number of applications. Choosing an optimal instruction-set for the specific applications under the constraints such as chip area and power consumption is crucial in maximizing the performance of the ASIP. This leads to several researches to make the tools which analyze the given applications and determine the optimal instruction-set to maximize performance.

There have been many works related to the ASIP synthesis [1-12], which can be categorized into four classes. First, the design of area efficient hardware blocks of an ASIP was handled in [1-4]: An evolution programming approach for the area efficient design was presented in [1][2], and in [3][4] the grouping problem that control-data flow graphs are bundled into m groups is considered to synthesize area efficient multi-function accelerators. These are used only to design hardware blocks and have almost no consideration on the relation between the synthesized hardware and the corresponding instruction. Second, the matching of a code sequence into a predefined instruction-set was handled in [5-9]. A tree based approach employing dynamic programming technique was proposed in [5]. In [6][7], the instruction selection phase and the register allocation phase are merged into a single tree covering phase, and an optimal scheduling algorithm to minimize the number of memory spills was proposed. Integer Linear Programming (ILP) based approaches considering instruction-level parallelism have been proposed in [8][9]. However, these approaches did not take into account the generation of new instructions optimal for the given applications. Third, how to select instructions and implement them were handled in [10][11]. Instructions are selected from the fixed super-set of all the possible instructions based on the intermediate language of GNU compiler. Hence, it cannot generate new instructions for a specific application, but just select them from the super-set. Lastly, the instruction generation problem was treated in [12] by formulating the problem as a modified scheduling problem of µ-operations (MOP’s). In the approach, each MOP is represented as a node to be scheduled, and a simulated annealing scheme was applied for solving the scheduling problem. This work is important in that it tried to generate application specific complex instructions. Complex instructions are more powerful than simple instructions because complex instructions can use the full power of execution engines supported in the processor, resulting in higher performance by exploiting more paral-
elism in MOP’s. However, in general purpose DSP processors, complex instructions have not been widely used because their complex instructions are too general to be used efficiently and powerfully for a specific application. In contrast, complex instructions of an ASIP can be efficient if the complex instructions are tuned for the applications that the ASIP aims at.

Although application specific complex instructions are usually executed in multi-cycle, only the single-cycle complex instructions are considered in [12]. The multi-cycle instruction, however, has two noticeable advantages over the single-cycle instruction. First, it can reduce the program memory size, which is crucial in embedded systems. Second, it can reduce the number of required code-fetches, thus can speed up the execution especially if the code is in the external memory that is much slower than the ASIP. In addition, the less memory accesses lead to the reduction of power consumption since fetching codes from memory consumes large power [13].

In this paper, we propose a new approach based on the subset-sum problem [15] to generate optimal instruction-set including multi-cycle complex instructions as well as single-cycle complex instructions. The proposed method can be also used to match the code sequence to the predefined instructions.

2 Backgrounds

In this section, we briefly address the target micro-architecture of the ASIP to be synthesized and the overview of our ASIP synthesis system called Partita.

The target architecture of the ASIP is a µ-program controlled pipelined architecture. Like the most DSP processors, it has a separate address generation unit (AGU), and can access two data-memories (XDM and YDM) simultaneously to fetch the operands. The µ-code in the µ-ROM is composed of eight fields: two fields for two simultaneous data-memory accesses, one field for arithmetic and shift operations, and one field for multiply and register data transfer operations. Hence, an arithmetic operation (or a shift operation) and a multiply operation (or a register move operation) can be executed in parallel. The operation in each field of the µ-control word is called MOP (µ-operation).

The ASIP supports three classes of instructions: P, C and S classes. First, P-class contains instructions that are not only primitive but also essential in all applications, i.e., simple arithmetic instructions and control instructions such as branch and call. The P-class instructions are always supported in all the generated ASIP’s. It is executed by the execution kernel. Second, C-class is composed of the instructions that are more complex than P-class instructions. Though it is also executed in the execution kernel controlled by the µ-program, it is more powerful than the P-instruction due to following two aspects. First, C-instruction can control all the units in the kernel at the same time, while P-instruction can use a limited number of units because of the instruction encoding problem. In other words, C-instruction can fully use the parallelism supported in the kernel. Second, C-instruction can reduce the code-memory size and the number of code-fetches because a C-instruction is comparable to several P-instructions. It is very important in the embedded system (the main target of the ASIP) having small internal code-memory. Therefore we can claim that generating appropriate C-instruction set is a very important task for the synthesis of the ASIP. The last instruction class is the S-class which is a set of instructions that are assisted by special hardware units called S-HW’s.

Now we address the overview of our ASIP synthesis system, Partita. The inputs to Partita are the application program written in C, typical input data for the application, and the user given constraint such as maximum execution time allowed. The input application is transformed into a MOP list. In the MOP list, almost all the concurrency in the source program are preserved. We sample-run the MOP list with the given typical input data to obtain the running frequency profile of each MOP. We first match the MOP list to the P-instructions to estimate the execution time if only the P-instructions are employed. If it is acceptable, we actually match the MOP list to the P-instructions. However, if not satisfactory, we start to generate C-instructions from the MOP list. If the generated C-instructions enable the code sequence to meet the timing constraint, we map the rest of the MOP list, not covered by the generated C-instructions, to the P-instructions. However, if the generated C-instructions are still not sufficient, we try to generate S-instructions. If the generated S-instructions fail to meet the timing constraint, we conclude that synthesizing the ASIP that meets the timing constraint is impossible. Otherwise, the rest of the MOP list is mapped to the P-instructions.

After finding the code sequence meeting the timing constraint, hardware module generation phase starts. S-HW’s and other necessary hardware modules are synthesized with considering the newly generated C-instructions and S-instructions. In this paper, we focus on the P-code matching and the C-instruction generation phases.

3 Generation of C-class Instructions

We first describe the generation of single-cycle C-instructions (SCC-instructions), and then extend it for the generation of multi-cycle C-instructions(MCC-instructions). Notice that this kind of separation between SCC-instruction and MCC-instruction is just for the convenience of explanation. We actually generate both of them simultaneously in a single framework.

3.1 Generation of Single-Cycle C-instructions

This subsection explains how SCC-instructions are generated from a MOP list. The difference between the estimated execution time with using only the P-instructions ($T_p$)
and the user given constraint \( (T_d) \) is represented as \( T_{d, \text{req}} \). If we generate a SCC-instruction, \( SC_i \), by merging several MOP’s, there is generally a speed gain \( g_i \). The problem of generating SCC-instructions can be formally stated as follows:

**Problem 1:** Given a MOP list, generate a SCC-instruction set such that 1) the total gain should be no less than \( T_{d, \text{req}} \), 2) the generated instructions should be used as many as possible in the application, and 3) the number of generated instructions should be as small as possible.

The rationale behind the requirements is to generate a small set of SCC-instructions which can be frequently used in the application. This prevents a code sequence whose pattern is rarely used in the application from being generated as a SCC-instruction. Such a rarely used code sequence can become a SCC-instruction if and only if it is indispensible to meeting the timing constraint. The number of generated C-instructions should be as small as possible, since each new C-instruction requires additional space in the \( \mu \)-ROM and makes the instruction decoder complex.

We can solve this problem optimally by formulating it as the subset-sum problem [15].

**Subset-sum problem:** Given \( S \) and \( t \), where \( S \) is a set \( \{x_1, x_2, ..., x_n\} \) of positive integers and \( t \) is a positive integer, find a subset of \( S \) whose sum is as large as possible but not larger than \( t \).

Although the time complexity of the subset-sum problem is exponential to obtain an optimal solution, there is a good approximation algorithm whose time complexity is polynomial.

For the formulation, we need to define some terms. A MOP in the given MOP list is represented as \( m_i \). The \( m_i \) may or may not have dependencies to other \( m_j \)'s. A compatible MOP group, \( C_i \), is the set of \( m_j \)'s that can be performed in the same cycle and can be specified in a single \( \mu \)-code line, i.e., \( m_j \)'s have no dependencies to one another and can be packed together in a single \( \mu \)-code line. In other words, a compatible MOP group is a candidate of SCC-instruction. As an example, given a MOP list \( \{m_1, m_2, m_3, m_4, m_5, m_6, m_7, m_8\} \), assume that \( m_1, m_2 \) and \( m_3 \) can be performed in a single-cycle, and so do \( m_6 \) and \( m_7 \). The possible five \( C_i \)'s are shown in Fig. 1.

![Fig. 1: Possible \( C_i \)'s](image)

Note that we consider all the possible \( C_i \)'s for \( m_1, m_2 \) and \( m_3 \) (i.e., not only \( C_1 \) but also \( C_{1,2} \)). Since a certain \( m_i \) may be included in more than one \( C_i \)'s, there is a constraint in selecting \( C_i \)’s that the selected \( C_i \)'s have no common \( m_j \)'s. For example, we have to select only one among \( C_{1,2} \). We call such a constraint \( C_i \) selection constraint.

We assign gain \( g_i \) to each \( C_i \) according to the speed gain (considering \( C_i \)'s running frequency profile) obtained by the generation of a SCC-instruction corresponding to the \( C_i \). Now the Problem 1 can be restated as follows:

**Problem 2:** Given a MOP list, select \( C_i \)'s satisfying \( C_i \) selection constraint, such that the following three ordered requirements are met: 1) the sum of \( g_i \)'s of selected \( C_i \)'s should be no less than \( T_{d, \text{req}} \), 2) the generated instructions should be used as many as possible in the application, and 3) the number of different SCC-instructions corresponding to the selected \( C_i \)'s should be as small as possible.

The third requirement can not be replaced by “the number of selected \( C_i \)'s should be as small as possible”. That is due to the fact that the number of selected \( C_i \)'s is not always equal to that of the different SCC-instructions to be generated because some \( C_i \)'s can be implemented by the same C-instruction. For example, given a MOP list \( \{m_1, m_2, m_3, m_4, m_5, m_6, m_7, m_8\} \), assume that \( T_{d, \text{req}} \) is 3 and the possible \( C_i \)'s are \( C_1=\{m_1, m_3\} \), \( C_2=\{m_1, m_7\} \), \( C_3=\{m_2, m_3\} \), \( C_4=\{m_1, m_2, m_3\} \), \( C_5=\{m_6, m_7\} \) and \( C_6=\{m_5, m_8\} \). The corresponding \( g_i \)'s are computed as 1, 1, 1, 2, 1 and 1, respectively. Let us also assume that \( C_2, C_3 \) and \( C_5 \) can be supported by the same C-instruction (henceforth such \( C_i \)'s are called c-isomorphic. The exact meaning and the way to decide the c-isomorphism will be covered later). If we try to minimize the number of selected \( C_i \)'s, the solution is to select \( C_4 \) and \( C_6 \). In this case, the number of different SCC-instructions is equal to that of the selected \( C_i \)'s. However, if we select \( C_2, C_3 \) and \( C_6 \), the number of different SCC-instructions is one (not three because they are mapped into the same C-instruction), while that of the selected \( C_i \)'s is three.

Now consider the second requirement. We may try to meet the second requirement (i.e., try to find frequently used C-instructions) by representing all the \( C_i \)'s that are c-isomorphic as a new single \( C_i \), whose gain is set to the sum of all the \( C_i \)'s gain, and then selecting \( C_i \)'s based on the gain. Though it may be a good method to meet the second requirement, it is based on the assumption that if the single \( C_i \) is selected, all the c-isomorphic \( C_i \)'s are selected and implemented by the same C-instruction. Hence, the scheme does not allow the case that only some of the c-isomorphic \( C_i \)'s are selected and the others are not selected. As an illustration, consider the above example again. Let us assume that the c-isomorphic \( C_2, C_3 \) and \( C_5 \) are represented as \( C_2 \). Then, the gain of \( C_1, C_2, C_3 \) and \( C_4 \) are 1, 3, 1 and 2, respectively (note \( C_2 \)'s gain is the sum of the gains of \( C_2, C_3 \) and \( C_5 \)). If we use the above scheme, \( C_2 \) that has the largest gain is selected, and the c-isomorphic \( C_i \)'s (i.e., \( C_2, C_3 \) and \( C_5 \)) are automatically selected. Hence, in that scheme the total gain is limited to 3 (note that only one among \( C_1-C_2 \) can be selected). So if the given \( T_{d, \text{req}} \) were 4, we could not find a solution. However, the solution for \( T_{d, \text{req}} = 4 \) actually exists and it is to select \( C_2, C_3 \) and \( C_5 \) (two C-instructions with gain 4). Notice that among the three c-isomorphic \( C_i \)'s only two are selected in the solution. This example claims that we should take into account the possibility that not all...
the c-isomorphic C_i's are mapped into the same C-instruction; some of them may be 1) included in a more larger C-instruction, 2) divided and parts of them may be used for different C-instructions, 3) remain as m_i's and lately mapped into P-instructions, etc.

We now present the way to solve the Problem 2 using a modified subset-sum problem. The Problem 2 can be re-formulated as follows:

Problem 3: Given S and T_d where S is the set of the gain g_i of the corresponding C_i, {g_1, g_2, ..., g_n}, find a subset of S whose sum is no less than T_d with satisfying following two requirements: 1) the generated instructions corresponding to the selected g_i's should be used as many as possible in the application, and 2) the number of different SCC-instructions corresponding to the selected g_i's should be as small as possible.

We can see that Problem 3 is an extension of the subset-sum problem. So, the Problem 3 can be solved by the extended subset-sum problem solver [15]. Fig. 2 shows the pseudo-code of the proposed SCC-instruction generation algorithm using the subset-sum problem.

SCC generation
1. G_d ← dependency graph(MOP list)
2. C ← generate all the possible C_i's(G_d)
3. G_i ← conflict graph(C)
4. G_c ← Sub-set-sum solver for SCC(C, S, G_i)
5. G_m ← change code(MOP list, SCC's)

Subset-sum solver for SCC
1. n ← |S|
2. RG ← ∑ S /* Remaining Gain */
3. L_i ← (0, 0, Ø, 0) /* (TG, XG, CS, ISS) */
4. for i ← 1 to n
5. L_i ← L_i ∪ (Li-1, i)
6. RG ← RG - Si/* RG - S(i): i'th element of S */
7. remove from Li every element whose ISS > MAX_ISS or TG + RG < T_d
8. for every element in Li, whose TG is larger than T_d
9. if XG > XG' or (XG == XG' and ISS < ISS') then
10. TG' ← TG, XG' ← XG, CS' ← CS, ISS' ← ISS
11. return (TG', XG', CS', ISS')

Fig. 2: SCC-instruction generation algorithm

For the given MOP list, dependency graph (G_d) among MOP's is first built. The data-dependency, output-dependency, and data-anti-dependency among MOP’s are checked and represented in the G_d. Each node in the graph represents a MOP, and each edge represents a dependency between two nodes. Based on G_d, all the possible C_i's are generated and their gains are computed (Notice that we do not consider the c-isomorphic C_i's in the gain computation phase. They are considered in the subset-sum problem solver). Then conflict graph (G_c) to represent the C_i selection constraint is built, where each node represents each C_i and an edge between two nodes represents that they have some common elements, hence only one of them can be selected as a solution. The subset-sum problem solver extended for the generation of SCC-instruction is employed to find the optimal solution for the given C_i's, S, and G_c. Then the MOP list is modified based on the SCC-instructions found by the subset-sum problem solver.

The details of the extended subset-sum problem solver is as follows. L_i is a list of all the possible solution candidates. Each element of L_i (i.e., each solution candidate) is in the form of (TG, XG, CS, ISS) which represents the total gain, extra gain, the list of selected C_i's, and the size of corresponding C-instruction set (with considering c-isomorphic C_i's). Operation L_i ⊕ i denotes a new list derived from L_i, for each element in the L_i, the gain of C_i (i.e., g_i) is added to the TG, 1 is added to XG if CS already contains some C_i's that are c-isomorphic with C_i, C_i is added to CS, and ISS is updated to the number of different C-instructions in CS. Before deriving new list by ⊕ operation, the C_i selection constraint should be checked between the CS of every element in L_i and the C_i to be added. In step 7, elements that have no possibility to become a solution are eliminated from the list; the elements whose C-instruction-set size (i.e., the number of different C-instructions) is larger than the instruction space allocated for C-instructions and those whose gain, TG, has no possibility to meet the T_d are eliminated. In step 8-10, the best solution, i.e., the elements of L_i which are most frequently used in the code with the minimum C-instruction-set size, is searched among the elements satisfying T_d. Notice that XG, extra gain, gives a favor to the C-instructions which are used frequently in the code. The (TG’, XG’, CS’, ISS’) keeps the best solution found.

As an illustration, the change of L_i is shown in Fig. 3 step by step for the example mentioned before; among six C_i's with gains 1, 1, 1, 2, 1 and 1, respectively, C_2, C_3 and C_6 are c-isomorphic. We can see in L_6 that (3, 2, [C_2, C_5, C_6], 1) is found as the solution for T_d = 3, and (4, 1, [C_6, C_5, C_6], 2) for T_d = 4, which are the optimal solutions as explained before. So one SCC-instruction is generated for T_d = 3 and two SCC-instructions (one for C_6, and the other for C_3 and C_5) are generated for T_d = 4.

L_0 = [(0, 0, Ø, 0)]
L_1 = [(0, 0, Ø, 0), (0, 0, C_1, 1)]
L_2 = [(0, 0, Ø, 0), (0, 0, C_1, 1), (0, 0, C_2, 1), (0, 0, C_3, 1), (1, 0, C_4, 1)]
L_3 = [(0, 0, Ø, 0), (0, 0, C_1, 1), (0, 0, C_2, 1), (0, 0, C_3, 1), (0, 0, C_4, 1), (0, 0, C_5, 1), (0, 0, C_6, 1)]
L_4 = [(0, 0, Ø, 0), (0, 0, C_1, 1), (0, 0, C_2, 1), (0, 0, C_3, 1), (0, 0, C_4, 1), (0, 0, C_5, 1), (0, 0, C_6, 1)]
L_5 = [(0, 0, Ø, 0), (0, 0, C_1, 1), (0, 0, C_2, 1), (0, 0, C_3, 1), (0, 0, C_4, 1), (0, 0, C_5, 1), (0, 0, C_6, 1)]
L_6 = [(0, 0, Ø, 0), (0, 0, C_1, 1), (0, 0, C_2, 1), (0, 0, C_3, 1), (0, 0, C_4, 1), (0, 0, C_5, 1), (0, 0, C_6, 1)]
L_7 = [(0, 0, Ø, 0), (0, 0, C_1, 1), (0, 0, C_2, 1), (0, 0, C_3, 1), (0, 0, C_4, 1), (0, 0, C_5, 1), (0, 0, C_6, 1)]

Fig. 3: Applying the proposed algorithm to an example

3.2 Generation of Multi-Cycle C-instructions

We extend the proposed method for the generation of MCC-instructions. The problem of generating MCC-instruction is almost the same as that of SCC-instruction generation.

Problem 4: Given a MOP list, generate MCC-instruction-set satisfying the following three ordered requirements: 1) the total gain should be no less than T_d, 2)
the generated MCC-instructions should be used as many times as possible in the application, and 3) the number of generated MCC-instructions should be as small as possible.

The most important difference between the SCC-instruction generation and the MCC-instruction generation is in the generation of \( C_i \)'s. Only the \( m_i \)'s that can be executed together in a cycle are included in a \( C_i \) in the SCC-instruction generation. However, in the MCC-instruction generation, a \( C_i \) can include all the MOP's that can be executed in sequel as well as in parallel. This enlarges the solution space for the MCC-instruction, and as a result more powerful instructions can be found compared to the SCC-instructions. However, the enlarged solution space causes the explosion of possible \( C_i \)'s for a large sized code, and thus the algorithm in Fig. 2 can become inefficient. To overcome such a situation, we limit the maximum length of \( C_i \) to a certain value based on the following rationale. A long \( C_i \) (i.e., \( C_i \) includes a large number of MOP's) has little chance to be selected as a MCC-instruction because the extra gain (\( \text{XG} \) and C-instruction set size \( \text{ISS} \)) factors favor MCC-instructions applicable multiple times in the code. Hence we can prune such long \( C_i \)'s from the solution space at the expense of little degradation of the solution quality. We use encoding constraints to limit the length of \( C_i \). In a multi-cycle \( C_i \), there are several operands that have to be encoded in the instruction format. Generally, the number of such operands increases as the length of \( C_i \) increases. Thus if we limit the number of operands allowed to be specified in the instruction format, the long \( C_i \) has more chance to violate the constraints.

If we fail to find a solution under the length limit \( k \), we increase the limit, and then retry to find a solution. Since the solution space being searched increases as the limit increases, the chance to find a solution increases also. However, we cannot increase the limit beyond some bound because it may significantly degrade the programmability. If the bound is reached, we move to make S-class instructions that will be run in special hardwares.

### 3.3 C-Isomorphism

In this part, we present the definition and a way to consider the c-isomorphism in generating C-instructions. Given a MOP list, identical \( C_i \)'s are those having the same operation sequence and the same operands. Clearly, identical \( C_i \)'s can be implemented by the same C-instruction. Since such identical \( C_i \)'s rarely exist in real codes, we use c-isomorphic \( C_i \)'s instead of the identical \( C_i \)'s in generating C-instructions. Two isomorphic \( C_i \)'s are c-isomorphic if a single C-instruction can specify both of them. Because of the limit on the operand encodings allowed in the instruction format, all of isomorphic \( C_i \)'s are not c-isomorphic.

The problem of finding c-isomorphic \( C_i \)'s is somewhat similar to that of finding (topologically) isomorphic \( C_i \)'s which can be handled by the graph matching algorithm and/or string matching algorithm [14]. However, there is an important difference between those two. For c-isomorphism, we have to consider the encoding constraints on each \( C_i \) because each \( C_i \) corresponds to a C-instruction.

As an illustration, consider Fig. 4 showing four \( C_i \)'s.

**Fig. 4: Instruction c-isomorphism**

We can see that \( C_1 \) and \( C_2 \) are identical, hence they can be implemented by the same C-instruction. For \( C_1 \) and \( C_3 \), we can see that they are not identical but isomorphic in topological point of view; \( R1 \) in \( C_1 \) is replaced by \( R5 \) in \( C_3 \). To make \( C_1 \) and \( C_3 \) share the same C-instruction, we have to provide a piece of information to the instruction regarding which register should be used as the first operand (destination) of the ADD. In other words, we have to encode the information in the instruction. If such an encoding is allowed, \( C_1 \) and \( C_3 \) can be mapped into the same C-instruction, but if it is not, we cannot merge them. Similarly, though \( C_1 \) and \( C_4 \) are isomorphic, we need two operand encodings in order to unite them as a single C-instruction; one for \( R1 \) in \( C_1 \) and \( R5 \) in \( C_4 \), and the other for \( R2 \) in \( C_3 \) and \( R6 \) in \( C_4 \). This simple example in Fig. 4 illustrates that whether two isomorphic \( C_i \)'s can be merged into a single C-instruction (i.e., c-isomorphic) is dependent on the amount of allowed additional encodings.

Here we want to address the use of temporary registers to reduce the required encoding information. Let us assume that the ASIP has three temporary registers accessible in the µ-codes. We use these temporary registers to reduce the required encoding information by replacing the general registers of \( C_i \)'s with them. As an illustration, assume that \( R1 \) in \( C_1 \) and \( R5 \) in \( C_4 \) of Fig. 4 are not used any more after the MOV instruction, i.e., they are not live variables after the MOV instruction. Then we can replace them with an temporary register. As a result, \( C_1 \) and \( C_3 \) become identical with no encoding information at all. We have to analyze the variable’s life time to find such a register.

The encoding constraint has a significant effect on the resulting code. In the case that we increase the allowed number of encodings, we may find C-instructions used more frequently in the code. However, due to the additional encoding information, the required size of the instruction format (i.e., the number of bits) of each C-instruction increases. The size of µ-ROM may decrease due to the C-instructions that cover many c-isomorphic \( C_i \)'s. On the contrary, if we decrease the allowed encodings, the size of an instruction format decreases and the µ-ROM size may increase.

### 3.4 Considering Other Basic Blocks

Hitherto, we have addressed the C-instruction generation considering a basic block alone. In this part, we present a
way to simultaneously consider other basic blocks in the generation of C-instructions. This is very important because it enables us to find more frequently used C-instructions in the application program from a global point of view.

The method is straight forward: We scan all the basic blocks with generating and gathering all the possible C_i’s and the corresponding conflict graphs. Since each basic block is executed in sequence, C_i’s in the different basic blocks can be simultaneously selected. Hence no additional edges are necessary between the different conflict graphs. We run the subset-sum problem solver with gathered C_i’s. The subset-sum problem solver then finds the optimal C-instruction-set for the entire application program, not for a single basic block.

For large sized applications, the method may become inefficient due to the large number of C_i’s gathered from all the basic blocks. In this case, we first group the basic blocks such that each group has basic blocks having similar C_i’s, and then we apply the above method for each group.

### 4 Generation of P-class Instructions

P-instruction is a predefined one-cycle instruction that can perform a limited set of MOP’s in a cycle. Compared to the SCC-instruction, the only difference is the possible set of MOP’s that can be performed in parallel: For P-instructions, a predefined and limited set is allowed due to the instruction encoding constraint, while for the SCC-instructions a more large sized set is allowed. So we can think the P-instruction as a special subset of the SCC-instruction. Thus if we limit the possible C_i’s only to those including MOP’s that are allowed to be executed in parallel in a single P-instruction, we can use the algorithm in Fig. 2 for the P-instruction matching.

### 5 Experimental Results

The proposed method has been implemented in C language on the SPARC-20 workstation with 128 Mbytes main memory. We tested the proposed method on 23 benchmark programs; the DSPStone benchmarks and some well-known DSP applications. For the given benchmark program, we first transform it into the MOP list, and then mapped it into the P-instructions. Then we employed the proposed C-instruction generation algorithm to reduce the execution cycle by 10% compared to the P-instruction code (P-code).

The results are shown in Table 1. For all the benchmarks, we used the following constraints: maximum encoding = 6, maximum length of a C-instruction = 3, maximum number of different C-instructions = 3. The P-code cycle and P-code BB show the execution cycle and the number of basic blocks in the P-code, respectively. The objective gain for each benchmark is shown in T_d. The C-code cycle shows the execution cycle of the new generated code including C-instructions, and the Gain Σ shows the corresponding total gain, i.e., the amount of reduced execution cycle. They show that the proposed method can find the solutions meeting the T_d for all the benchmarks. The total gain is composed of two components: the gain due to the reduced code-fetches, denoted as Gain fetch, and the gain due to the parallel execution of MOP’s, denoted as Gain par. The number of generated C-instructions (C-instruction num) and the number of their occurrences in the code (C-instruction occur) verify that the proposed method generates the C-instructions which are used frequently in the code, for example, two C-instructions that are used eleven times are found for the benchmark fir2dim. The last column (CPU (s)) shows the CPU time in second for the generation of the C-instructions. For the last two benchmark programs, we divided them into two groups because of their size as explained in the section 3, and then each group was handled separately.

To see the effect of different constraints on the generated C-instructions, we changed the constraints and repeated above experiments. The used constraints are: maximum encoding = 6, maximum length of a C-instruction = 3, and maximum number of different C-instructions = 2. Notice that the maximum encoding constraint and the maximum length constraint are relaxed. Table 2 shows the results that are different from those of Table 1. Specifically we can see that C-instructions composed of three MOP’s (3mop) are generated as well as those composed of two MOP’s (2mop).

Some of the C-instructions generated in the above experiments are shown in Fig. 5. Fig. 5-(a), (b) and (c) show the C-instructions generated because they are commonly used at the beginning of the functions (subroutines). Notice that the execution cycle of (c) is two, not three, because MOV and LDI can be executed in one cycle. The code in (d) is a frequently used one at the end of a loop (e.g., for, while, etc.). The (e) and (f) are the C-instructions that are equal to the shift by immediate value and the push immediate value which are commonly used but are not supported in the P-instruction set.

![Fig. 5: Some of the generated C-instructions](image)

### 6 Conclusions

In this paper, we presented a new approach to generate an optimal instruction set from the given DSP applications. We transformed the instruction generation problem to the extended subset-sum problem, and used the subset-sum problem solver to synthesize application specific instructions. Along with many things to be considered such as c-isomorphism and encoding constraints, we showed the way to apply the proposed framework for the generation of single-cycle C-instructions and multi-cycle C-instructions. In addition, we described how to apply the proposed method...
in order to match the given application into the predefined instructions. The experimental results indicate that the proposed approach is effective in reducing the code size as well as increasing the performance.

References

<table>
<thead>
<tr>
<th>benchmarks</th>
<th>cycle</th>
<th>BB</th>
<th>Td</th>
<th>cycle</th>
<th>Σ</th>
<th>fetch</th>
<th>par</th>
<th>num</th>
<th>occur</th>
<th>time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>biquad_4</td>
<td>99</td>
<td>10</td>
<td>10</td>
<td>88</td>
<td>11</td>
<td>8</td>
<td>3</td>
<td>2</td>
<td>8</td>
<td>136.2</td>
</tr>
<tr>
<td>cmplx_mul</td>
<td>68</td>
<td>4</td>
<td>7</td>
<td>60</td>
<td>8</td>
<td>8</td>
<td>0</td>
<td>2</td>
<td>8</td>
<td>91.5</td>
</tr>
<tr>
<td>cmplx_up</td>
<td>52</td>
<td>4</td>
<td>5</td>
<td>47</td>
<td>5</td>
<td>5</td>
<td>0</td>
<td>2</td>
<td>5</td>
<td>68.3</td>
</tr>
<tr>
<td>convolution</td>
<td>48</td>
<td>7</td>
<td>5</td>
<td>43</td>
<td>5</td>
<td>5</td>
<td>0</td>
<td>2</td>
<td>5</td>
<td>27.9</td>
</tr>
<tr>
<td>dot_product</td>
<td>44</td>
<td>6</td>
<td>4</td>
<td>40</td>
<td>4</td>
<td>4</td>
<td>0</td>
<td>2</td>
<td>4</td>
<td>16.5</td>
</tr>
<tr>
<td>f1r</td>
<td>67</td>
<td>8</td>
<td>7</td>
<td>59</td>
<td>8</td>
<td>8</td>
<td>0</td>
<td>3</td>
<td>8</td>
<td>184.4</td>
</tr>
<tr>
<td>f1r2dim</td>
<td>150</td>
<td>30</td>
<td>15</td>
<td>134</td>
<td>16</td>
<td>11</td>
<td>5</td>
<td>2</td>
<td>11</td>
<td>170.3</td>
</tr>
<tr>
<td>for01</td>
<td>51</td>
<td>6</td>
<td>5</td>
<td>45</td>
<td>6</td>
<td>6</td>
<td>0</td>
<td>3</td>
<td>6</td>
<td>62.0</td>
</tr>
<tr>
<td>fract11</td>
<td>46</td>
<td>5</td>
<td>4</td>
<td>42</td>
<td>4</td>
<td>4</td>
<td>0</td>
<td>2</td>
<td>4</td>
<td>34.6</td>
</tr>
<tr>
<td>fract8</td>
<td>82</td>
<td>9</td>
<td>8</td>
<td>73</td>
<td>9</td>
<td>9</td>
<td>0</td>
<td>2</td>
<td>9</td>
<td>26.4</td>
</tr>
<tr>
<td>index</td>
<td>44</td>
<td>5</td>
<td>4</td>
<td>40</td>
<td>4</td>
<td>4</td>
<td>0</td>
<td>2</td>
<td>4</td>
<td>18.2</td>
</tr>
<tr>
<td>int</td>
<td>63</td>
<td>6</td>
<td>6</td>
<td>57</td>
<td>6</td>
<td>6</td>
<td>0</td>
<td>2</td>
<td>6</td>
<td>198.5</td>
</tr>
<tr>
<td>notchfilter01</td>
<td>62</td>
<td>5</td>
<td>6</td>
<td>56</td>
<td>6</td>
<td>4</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>85.3</td>
</tr>
<tr>
<td>notchfilter02</td>
<td>62</td>
<td>5</td>
<td>6</td>
<td>55</td>
<td>7</td>
<td>5</td>
<td>2</td>
<td>2</td>
<td>5</td>
<td>86.6</td>
</tr>
<tr>
<td>lms</td>
<td>125</td>
<td>10</td>
<td>13</td>
<td>112</td>
<td>13</td>
<td>10</td>
<td>3</td>
<td>3</td>
<td>10</td>
<td>124.6</td>
</tr>
<tr>
<td>madd03</td>
<td>44</td>
<td>5</td>
<td>4</td>
<td>40</td>
<td>4</td>
<td>4</td>
<td>0</td>
<td>2</td>
<td>4</td>
<td>18.3</td>
</tr>
<tr>
<td>mat1x3</td>
<td>25</td>
<td>5</td>
<td>2</td>
<td>23</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>0.1</td>
</tr>
<tr>
<td>nested_for02</td>
<td>63</td>
<td>10</td>
<td>6</td>
<td>56</td>
<td>7</td>
<td>5</td>
<td>2</td>
<td>2</td>
<td>5</td>
<td>62.2</td>
</tr>
<tr>
<td>parallel02</td>
<td>81</td>
<td>8</td>
<td>8</td>
<td>72</td>
<td>9</td>
<td>9</td>
<td>0</td>
<td>2</td>
<td>9</td>
<td>212.1</td>
</tr>
<tr>
<td>temp01int</td>
<td>74</td>
<td>6</td>
<td>7</td>
<td>66</td>
<td>8</td>
<td>6</td>
<td>2</td>
<td>3</td>
<td>6</td>
<td>261.2</td>
</tr>
<tr>
<td>temp02int</td>
<td>65</td>
<td>5</td>
<td>6</td>
<td>58</td>
<td>7</td>
<td>5</td>
<td>2</td>
<td>2</td>
<td>5</td>
<td>104.2</td>
</tr>
<tr>
<td>adpcm</td>
<td>195</td>
<td>54</td>
<td>20</td>
<td>171</td>
<td>24</td>
<td>24</td>
<td>0</td>
<td>4</td>
<td>24</td>
<td>205.5</td>
</tr>
<tr>
<td>idct</td>
<td>406</td>
<td>31</td>
<td>16</td>
<td>390</td>
<td>16</td>
<td>15</td>
<td>1</td>
<td>4</td>
<td>15</td>
<td>127.4</td>
</tr>
</tbody>
</table>

**Table 1**: Experimental results

<table>
<thead>
<tr>
<th>benchmarks</th>
<th>P-code cycle</th>
<th>C-code cycle</th>
<th>Σ</th>
<th>fetch</th>
<th>par</th>
<th>2mop</th>
<th>3mop</th>
<th>occur</th>
<th>time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>f1r</td>
<td>67</td>
<td>7</td>
<td>59</td>
<td>7</td>
<td>7</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>for01</td>
<td>51</td>
<td>6</td>
<td>45</td>
<td>6</td>
<td>6</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>temp01int</td>
<td>74</td>
<td>7</td>
<td>66</td>
<td>8</td>
<td>6</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>4</td>
</tr>
</tbody>
</table>

**Table 2**: Experimental results