MetaCore: An Application-Specific Programmable DSP Development System

Jin-Hyuk Yang, Byoung-Woon Kim, Sang-Joon Nam, Young-Su Kwon, Dae-Hyun Lee, Jong-Yeol Lee, Chan-Soo Hwang, Yong-Hoon Lee, *Member, IEEE*, Seung-Ho Hwang, *Member, IEEE*, In-Cheol Park, *Member, IEEE*, and Chong-Min Kyung, *Senior Member, IEEE*

***Abstract*—This paper describes the MetaCore system which is an application-specific instruction-set processor (ASIP) development system targeted for digital signal processor (DSP) applications. The goal of the MetaCore system is to offer an efficient design methodology meeting specifications given as a combination of performance, cost, and design turnaround time. The MetaCore system consists of two major design stages: design exploration and design generation. In the design exploration stage, MetaCore system accepts a set of benchmark programs and structural/behavioral specifications for the target processor and estimates the hardware cost and performance for each hardware configuration being explored. Once a hardware configuration and instruction set are chosen, the system helps generate the target processor design in the form of hardware description language (HDL) along with the application program development tools such as C compiler, assembler, and instruction set simulator. The effectiveness of the MetaCore system was verified with a successful design of MDSP-II, a programmable DSP processor targeted for mobile communication.**

***Index Terms*—Application-specific instruction-set processor (ASIP), design efficiency, design turnaround time, digital signal processor, processor specification, retargetable compilation.**

I. Introduction

DIGITAL signal processors (DSP's) are used in an extremely diverse range of applications, from speech, audio, and telecommunication on the low-frequency side of the spectrum to image, video, and radar processing on the high-frequency side [1]. However, no single DSP can meet the requirements in terms of throughput, area, and power consumption of all applications. The requirement for the competitive performance for a specific application has generated a need for the so-called application-specific instruction-set processor (ASIP) [2], [3].

Two things are very important for a successful ASIP design, especially for the consumer-electronics market. The first objective of the ASIP design is the design quality addressed in terms of speed, power, chip area, and flexibility. In the ASIP design, the most important issue is the definition of architecture and instruction set, which are closely affected by each other. In other words, instruction set designed without the knowledge of sufficient details of the underlying architecture generally incurs excessive execution time and hardware cost. The matching between architecture and the instruction set can only be achieved through a diverse exploration of design space based on the understanding of the interaction between architecture (hardware) and algorithm of the given application (software).

The second objective of the ASIP design is the short turnaround time, which depends on the efficiency of transforming the given processor specification to lower level design, e.g., chip layout. The traditional design approach of describing design entity using hardware description language (HDL) or a schematic and verifying the design using simulation needs a significant number of iterations between higher level specification and lower level implementation.

In this paper, we present a systematic approach called MetaCore to develop an ASIP for DSP applications. MetaCore system consists of two major design stages: design exploration and design generation. MetaCore system, given a set of benchmark programs and structural/behavioral specifications of target ASIP as its inputs, produces information on the design such as performance/cost parameters for the given processor configuration, which guides the designer through the subsequent steps of decisions or choices among alternatives. Another salient feature of the MetaCore system is that it generates the ASIP design in the form of HDL along with a set of application program development tools such as C compiler, assembler, and instruction set simulator.

The rest of this paper is organized as follows. Section II reviews related works on ASIP for DSP applications. Section III describes the design philosophy and framework of the proposed MetaCore system. Section IV presents the ASIP design flow based on the MetaCore design environment. Section V describes a practical ASIP design example based on the MetaCore design flow, followed by concluding remarks.

II. Related Works

In this section, we briefly review the existing ASIP development systems. Instruction set optimization, which is one of the most important tasks in the ASIP design process, was once viewed as the selection of appropriate instructions from the predefined super set. An early system in this direction includes the EPICS [4] and PEAS-I [3], [5], where the given reconfigurable architecture is tuned to each specific application by changing some architectural parameters such as bit width of hardware functional blocks, register file size, memory size, etc. PEAS-I generates profiling information from the given set of application programs. Based on the profiles, the user selects useful instructions from a predefined super set. Selecting instructions from a

Manuscript received August 20, 1998; revised May 16, 1999. This work was supported by LG Semicon, Inc.

The authors are with the Department of Electrical Engineering, Korea Advanced Institute of Science and Technology (KAIST), Taejon, Korea.

Publisher Item Identifier S 1063-8210(00)00772-1.

super set cannot always fully satisfy the demand of diverse applications. In these systems, however, the predefined super set is fixed, i.e., not extensible by the user.

On the other hand, Holmer [6] and Huang [7] have focused on generating the instruction set. They adopted the pipelined control model and parameterized datapath. The parameters for the datapath include the number of read/write ports of register/memory, number of functional units, and the number of cycles for memory operation. For a given set of parameters, the instruction set that best utilizes the hardware resources is synthesized such that minimal cycle count for the given benchmarks is achieved. However, the underlying architecture they assumed is based on the register-based load/store machine, which is not suitable for DSP applications having extensive memory interactions. Another problem, as stated by the authors of [7], is that they cannot deal with the large-size real-world applications.

Recently, various ASIP development systems that permit user-defined application-specific instructions to be equipped with the target processor have been introduced including FlexWare [8], Chess [9], ILP-based approach [10], Move [11] Aviv [12], and Xtensa [13]. Each of these systems consists of its own processor description language and a retargetable code generator. The processor description language is used to describe the target processor's instruction set and the hardware structure, while the major task of the retargetable code generators is generating the optimized machine code for the target processor. There is a large variety in the processor description languages of the systems: for example, list of instructions only [8], behavioral descriptions only [10], and mixed behavioral/structural descriptions [9], [12], [13]. Some of these are insufficient to precisely describe the hardware operations of the target processor. On the other hand, the retargetable code generation procedures in the systems are all similar, i.e., the retargetable code generators convert the input application program into a dataflow graph, and cover the dataflow graph with patterns that correspond to target instructions. One of the weaknesses of these systems is that they hardly provide feedback to the target processor design, i.e., the user cannot know the efficiency of the designed target processor. FlexWare includes an instruction set simulator, Insulin [8], which provides a cycle-based VHDL simulation. However, the simulator is only used to debug the application program. On the other hand, Chess [9] and Xtensa [13] provide only the usage count (UC) of each hardware block and instruction in the cadidate instruction set. Another problem is that the systems in [9], [10], and [12] do not provide any link to hardware synthesis systems, so that the user has to do substantial work to get the target processor, which includes HDL coding and simulation.

MetaCore system proposed in this paper adopts a DSP-oriented architecture and provides various design facilities such as processor description languages and retargetable system software which cover the whole ASIP design stages from design exploration to design generation. The processor description languages provide a precise description of the behavior of each instruction and hardware block of the target processor. During the design exploration stage, MetaCore system provides feedback as to the efficiency of current design based on the analysis of hardware cost and dynamic behavior of application programs. Furthermore, the system gives hints for the user to develop application-specific instructions. MetaCore system also generates synthesizable HDL code for the given processor specification as a standard link to modern hardware synthesis system.

III. Design Philosophy and Framework

Major issues in the design of MetaCore system are:

1) how to obtain programmable yet efficient microarchitecture for given DSP applications;
2) how to extract features (such as instruction set and requirement for special functional blocks) for improving the performance/cost ratio of the target processor;
3) how to automatically generate the target DSP processor equipped with the extracted features.

In order to solve these problems, we have developed MetaCore system which consists of a reconfigurable architecture, processor specification language, and a number of tools supporting ASIP design (the tools will be described in following section).

The reconfigurable architecture is designed to meet the general characteristics of various DSP applications. The reconfigurability offers a high degree of performance/cost optimization as required for each specific application by supporting the MetaCore user to modify the instruction set and the architecture of the target processor. In the MetaCore system, the modifications of the instruction set and architecture, either addition of some application-specific instructions with or without the incorporation of some macro blocks supporting them, or some deletions, if necessary, are guided by such information as UC of each instruction, gate count, and frequent sequence of contiguous instructions as provided through the built-in analysis of benchmark programs.

To make the core and relevant software modification process simple and flexible, the MetaCore system is provided with structural/behavioral specification language, with which one can describe the target data path and the instruction set architecture (ISA). The structural language is called MetaCore structural language (MSL), and the behavioral language is called MetaCore behavioral language (MBL).

A. Microarchitecture and Instruction Set

DSP applications are generally characterized as computationally intensive with a large data set, loop-dominant control flow behavior, and accumulation-based operations [1]. To support these characteristics, it is necessary that the microarchitecture should support effective data communication between memory system and execution units, low-overhead loop control, and accumulator-based instruction set architecture. MetaCore system provides predefined microarchitecture supporting the general characteristics of various DSP applications.

Fig. 1 shows the structure of predefined microarchitecture provided by the MetaCore system. The heart of the microarchitecture consists of three execution units operating in parallel, i.e., program control unit (PCU), data arithmetic and logical unit (DALU), and address generation unit (AGU). The PCU is a pipelined instruction decoder with branching (direct/indirect

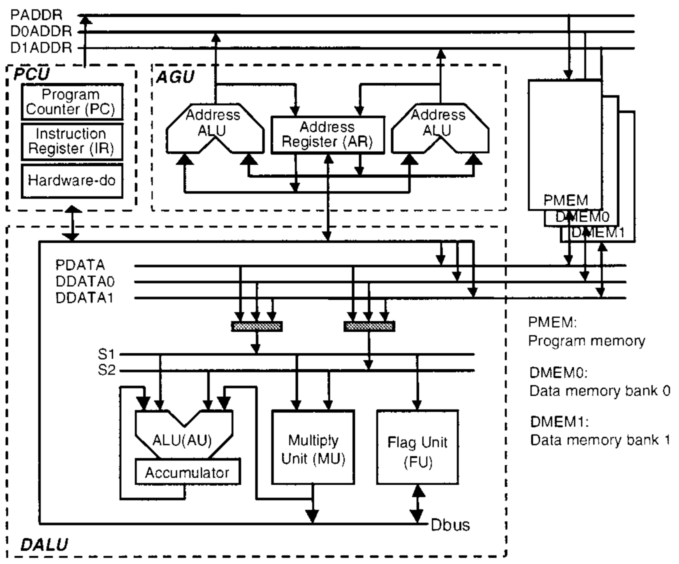


Fig. 1. The microarchitecture of MetaCore consisting of PCU, DALU, and AGU.

jump, call/return) and interrupt handling (goto/return from interrupt). The PCU also supports hardware-do loop [1] capability that reduces the cycle overhead for loop control. The DALU, which consists of the Multiply-ACcumulate (MAC) unit and various basic hardware units, have been designed to optimize the time-critical inner-loop functions of DSP algorithms. The DALU also contains the flag unit (FU) including flag register that is updated when arithmetic/logic instructions are executed. The flag register consists of four flag fields such as carry (c), overflow (v), zero (z), and negative (n). The AGU is responsible for effective indirect addressing of data operands in memory. The AGU contains dedicated interconnections among address registers (AR's) and increment/decrement capabilities for efficient traversal of the special memory structures. The addressing modes supported by the AGU are linear, modular, and bit-reverse mode [1], [14].

The parallel processing capability, e.g., performing the execution of arithmetic/logic operation, fetching operands from memory and updating the operand address pointer in parallel, is heavily exploited for increasing the data throughput. For example, Fig. 2 shows the MAC instruction which multiplies two operands pointed to by address registers AR1 and AR2 and accumulates the result in a0. During multiplication and accumulation, the address pointers AR1 and AR2 are incremented by one in parallel.

The design style of the predefined microarchitecture is parameterized and pipelined. The architectural parameters include register file size, bus width, address space of each memory, and bit width of functional blocks. The pipeline consists of five stages for instruction fetch, instruction decode, operand read, execute, and result write. The first two stages, instruction fetch and instruction decode, have the same behavior for all instructions, while the behaviors of the remaining three stages, operand read, execute, and result write, are dependent on the semantics of the instructions.

MetaCore system has a predefined instruction set which consists of instructions mostly used in the implementation of

```
MAC   a0  AR1++  AR2++                                   [description]
a0 = a0 + (mem[AR1] * mem[AR2]);  AR1=AR1+1, AR2=AR2+1  [ operation ]
```

Fig. 2. The description and actual operations of the arithmetic instruction MAC. MAC instruction performs four arithmetic operations (including multiplication, accumulation, and two address calculations) and two operand fetches from data memories DMEM0 and DMEM1.

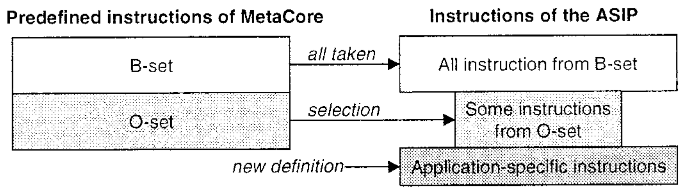

Predefined instructions of MetaCore
Instructions of the ASIP
B-set
O-set
all taken
selection
new definition
All instruction from B-set
Some instructions from O-set
Application-specific instructions

Fig. 3. Typical organization of instruction set of the ASIP designed based on MetaCore.

various DSP applications (see Fig. 3). The predefined instructions are classified into two classes, i.e., basic class (B-set) and optional class (O-set). B-set includes 12 arithmetic/logic instructions, six data move instructions, and nine control transfer instructions. O-set includes 13 arithmetic/logic instructions. The B-set instructions are essential and are not omissible from any ASIP designed under the MetaCore environment, while the O-set instructions can be eliminated to satisfy given design constraints.

The user can construct the target instruction set by selecting useful instructions from the predefined instruction set. The instruction set customization helps the user to design a cost-effective DSP processor. After selecting the useful instruction from the predefined instruction set, further design refinement can be achieved by modifying datapath structure and specifying application-specific instructions. The instruction set of the ASIP as generated via MetaCore consists of three groups, i.e., all the instructions in the B-set of the predefined instruction set, some instructions selected from O-set, and user-defined application-specific instructions.

B. Specification of Target Processor

The processor (target ASIP) specification in the MetaCore system is described using the structural specification language MSL and behavioral specification language MBL. MSL is used to specify the data path structure of the target microarchitecture, while MBL is used to specify the architectural parameters and the behavior of instructions for the target ASIP.

The MSL description consists of declarations of hardware resources such as busses, latches, multiplxer, functional units, and interconnections among the hardware resources. Fig. 4 shows portions of MSL and MBL descriptions for predefined architecture and instruction set. The MSL description Fig. 4(a) represents the hardware structure of DALU part of the predefined architecture shown in Fig. 1. The hardware resource is declared in the resource clause, and the interconnections among them are described in *dflow* clause. The MBL description Fig. 4(b) represents hardware parameters for target processor and each instruction's low-level mechanisms such as bit operation and timing information. The *hardware* clause describes the architectural parameters for each resource. In this example, the width of each bus and execution unit is 32 b, and the accumulator consists of four different 32-b registers. During each instruction cycle, the behavior of instruction fetch and instruction decode stages, which are identical for all instructions, are not included in the specification of ISA. Only the behavior of the remaining pipeline stages, i.e., operand read, execute, and result write, of each instruction is specified in the *def_inst* clause and *operand* clause as follows.

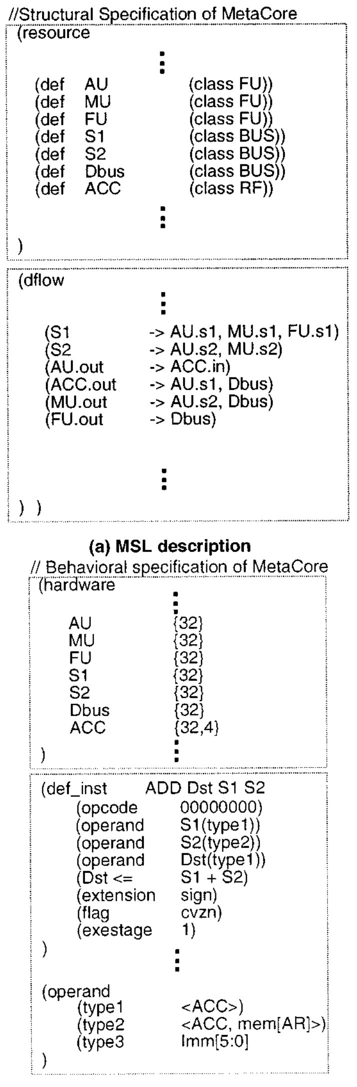
- The def_inst clause represents the mnemonic and the instruction behavior in detail: opcode, arithmetic function, and flag fields that are affected by the execution of instruction and the number of execution stages in the pipeline needed to execute the instruction.
- The operand clause defines the type of operands.

The example shows that ADD instruction, whose opcode is 8-b 00 000 000, adds S1 and Dst and writes the result to Dst. The flag fields affected by ADD instruction are carry (c), overflow (v), zero (z), and negative (n). The number of execution stages for ADD instruction is one. The MSL and MBL descriptions for the predefined architecture and instruction set are provided to the user for the initial design. Modification of the design, including addition/deletion of some features to/from the initial design, can be performed by changing the descriptions. The deletion of optional instruction is straightforward. In the case of adding user-defined instructions, there are two restrictions in describing the target processor for the sake of simplicity.

1) Control instructions such as branch, jump, and return are not supported as user-defined instructions.
2) All user-defined instructions have at most two source operands and one destination operand.

Even though the restrictions limit the user's freedom to define application-specific instructions, they are not significant. The control flow of DSP application is rather simple, and the predefined instruction set already includes various control instructions including zero-overhead looping (hardware do-loop), many kinds of branching, etc. The second restriction comes from the fixed-sized instruction word. When an operation needs more operands, a proper sequence of instructions for the operation can be used.

If the user wants to add new arithmetic/logic operations, he can simply do so by including the description for the corresponding functional blocks in the MSL description and newly defining the instructions in the MBL description. The modification can be fully supported by the system tools such as instruction set simulator (ISS), assembler, compiler, performance analyzer, and HDL code generator (see Section IV).



```
//Structural Specification of MetaCore
(resource
            :
  (def  AU        (class FU))
  (def  MU        (class FU))
  (def  FU        (class FU))
  (def  S1        (class BUS))
  (def  S2        (class BUS))
  (def  Dbus      (class BUS))
  (def  ACC       (class RF))
            :
)
(dflow
            :
   (S1         -> AU.s1, MU.s1, FU.s1)
   (S2         -> AU.s2, MU.s2)
   (AU.out     -> ACC.in)
   (ACC.out    -> AU.s1, Dbus)
   (MU.out     -> AU.s2, Dbus)
   (FU.out     -> Dbus)
            :
) )
```

(a) MSL description

```
// Behavioral specification of MetaCore
(hardware
            :
     AU          {32}
     MU          {32}
     FU          {32}
     S1          {32}
     S2          {32}
     Dbus        {32}
     ACC         {32,4}
)           :
(def_inst      ADD Dst S1 S2
     (opcode       00000000)
     (operand      S1(type1))
     (operand      S2(type2))
     (operand      Dst(type1))
     (Dst <=       S1 + S2)
     (extension    sign)
     (flag         cvzn)
     (exestage     1)
)            :
(operand
     (type1        <ACC>)
     (type2        <ACC, mem[AR]>)
     (type3        Imm[5:0]
)
```

(b) MBL description

Fig. 4. The processor specification of predefined architecture and instruction set using MSL and MBL.

IV. ASIP Design Flow Under MetaCore Environment

Fig. 5 shows the ASIP design flow under the MetaCore environment. The inputs to MetaCore system are benchmark programs and processor specification described using MSL and MBL. At first, MetaCore system parses the processor specification and generates control files for MetaCore system softwares. The control files consist of architectural information used for each system software.

The design exploration in the initial design stage directly affects the quality of the final design. In the MetaCore system, such tools as MCC (C compiler), MASM (assembler), MISS (instruction set simulator), and MPA (performance analyzer) help the user explore the instruction set space. After a successful design exploration, the design generation corresponds to the design transformation from the higher level specification to the lower level implementation, which is handled by SMART, an HDL code generator for the MetaCore architecture.

The design exploration and design generation in the MetaCore system can be done in the very abstract level, while the definition of low-level details and constraints are described in the system library. The system library consists of descriptions of macro blocks in the form of parameterized HDL and physical details such as cost, speed, and power.

A. Design Exploration

The main task of the design exploration stage is to evaluate the target design through estimating lower level implementation characteristics based on the instruction-level analysis of benchmark programs. The user can, based on the estimations of speed,

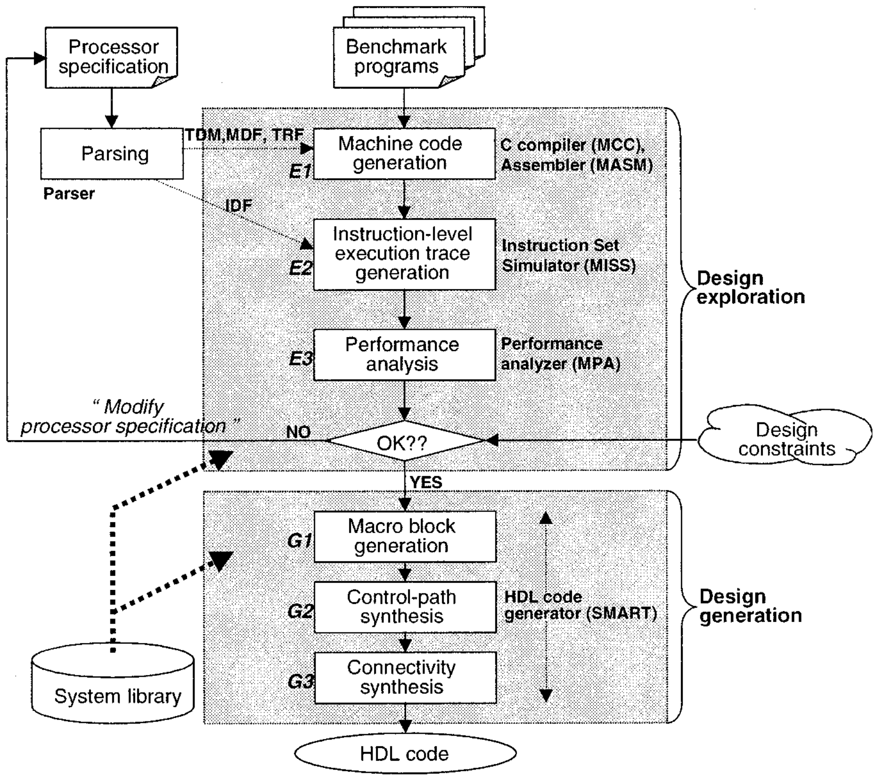


Fig. 5. The ASIP design flow under the MetaCore environment consisting of design exploration stage (E1, E2, and E3) and design generation stage (G1, G2, and G3, which are performed by an HDL code generator, SMART). TDM, MDF, and TRF are control files for controlling the translation process of C-compiler (MCC) and assembler (MASM) as generated by parsing the processor specification input.

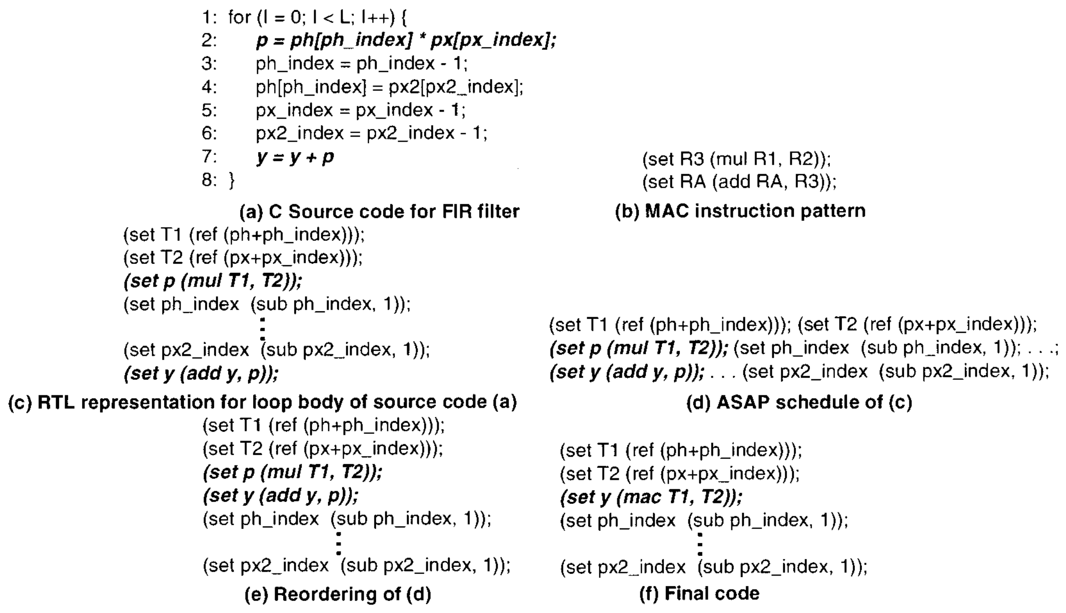
power, and area, optimize the target design by tuning the data path and/or instruction set and newly defining application-specific instructions with the corresponding macro blocks if necessary. As shown in Fig. 5, the related design steps for the design exploration are from steps E1 to E3. The operations in each design step and its associated tools are described below.

1) Machine Code Generation (Step E1): MCC and MASM, which translate the application programs in C and assembly code into object code, respectively, have retargetability to support the instruction set variation. The retargetability of MCC is based upon the compiler–compiler method introduced by GCC (GNU C-Compiler) [15]. The whole compilation process is divided into two steps. In the first step, which is a machine-independent process, MCC translates the source program into register-transfer level (RTL) representation. In the second step, MCC translates the RTL representation into the assembly code. The control files, i.e., machine description file (MDF) and target description macros (TDM's) generated by parser, control the translation process. MDF defines the mapping rule between patterns of RTL representation and instructions, while TDM consists of such parameters as stack area, and number of registers. The two major jobs in the second step are code selection and register allocation [15], [16]. In the code selection, nodes of the dataflow graph represented using the RTL are covered with patterns that correspond to instructions. The pattern for each instruction is described in MDF.

It was shown that the problem of generating optimal coverings, i.e., pattern matching, is NP-complete [22]. Furthermore, the result of pattern matching is highly dependent on the coding style of the source program. In an FIR filter example from DSPstone [20], shown in Fig. 6(a), the operation 2 "$p = ph[ph\_index] * px[px\_index]$" and the operation 7 "$y = y + p$" together correspond to an MAC instruction. In the original RTL representation in Fig. 6(c), the multiplication and the addition are separated from each other. Hence, MCC cannot directly match the operations to the MAC instruction pattern in Fig. 6(b). As a result, multiplication and addition are individually mapped to MUL and ADD instructions rather than an MAC instruction.

To reduce the effect of coding style and enhance the quality of the code selection, node reordering is used. In Fig. 6(a), the operation 7 is not dependent on any operation between operation 2 and operation 7. So, the operation 7 can be moved next to the operation 2 such that the two operations can be mapped to an MAC instruction.

To reorder the original RTL description, MCC performs as soon as possible (ASAP) scheduling [23] for all operations in the RTL. From the ASAP-scheduled code, MCC can efficiently find the target operation to be reordered since the data dependency between operations can be easily recognized and the search depth is limited to the depth of the target instruction pattern graph. Using the ASAP-scheduled code in Fig. 6(d), an MAC pattern can be found by searching two consecutive control steps for a multiplication followed by an addition because the depth of the MAC instruction pattern graph is two. In the ASAP-scheduled code "(*set p* (*mul T1, T2*))" and "(*set y* (*add y, p*))" are scheduled in consecutive control steps and



(a) C Source code for FIR filter

```
for (l = 0; l < L; l++) {
    p = ph[ph_index] * px[px_index];
    ph_index = ph_index - 1;
    ph[ph_index] = px2[px2_index];
    px_index = px_index - 1;
    px2_index = px2_index - 1;
    y = y + p
}
```

(b) MAC instruction pattern

```
(set R3 (mul R1, R2));
(set RA (add RA, R3));
```

(c) RTL representation for loop body of source code (a)

```
(set T1 (ref (ph+ph_index)));
(set T2 (ref (px+px_index)));
(set p (mul T1, T2));
(set ph_index (sub ph_index, 1));
        :
(set px2_index (sub px2_index, 1));
(set y (add y, p));
```

(d) ASAP schedule of (c)

```
(set T1 (ref (ph+ph_index))); (set T2 (ref (px+px_index)));
(set p (mul T1, T2)); (set ph_index (sub ph_index, 1)); . . .;
(set y (add y, p)); . . . (set px2_index (sub px2_index, 1));
```

(e) Reordering of (d)

```
(set T1 (ref (ph+ph_index)));
(set T2 (ref (px+px_index)));
(set p (mul T1, T2));
(set y (add y, p));
(set ph_index (sub ph_index, 1));
        :
(set px2_index (sub px2_index, 1));
```

(f) Final code

```
(set T1 (ref (ph+ph_index)));
(set T2 (ref (px+px_index)));
(set y (mac T1, T2));
(set ph_index (sub ph_index, 1));
        :
(set px2_index (sub px2_index, 1));
```

Fig. 6. The procedure of the code selection performed by MCC. To enhance the quality of the code selection, a code reordering technique is used.

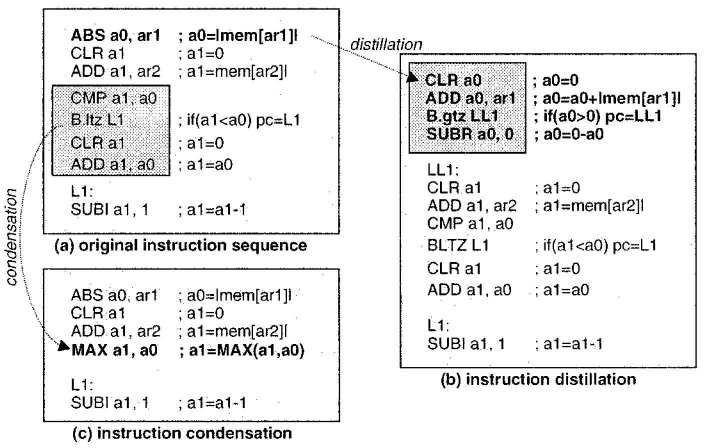
can be mapped to an MAC instruction. Then MCC reorders the RTL description such that two operations are consecutively located, as shown in Fig. 6(e). Finally, MCC performs the pattern matching between the reordered RTL and instruction patterns. The final code containing an MAC operation is shown in Fig. 6(f).

After the code selection, register allocation is performed. In the process of translating C into RTL, MCC assumes that there is infinite number of registers, i.e., the number of identical registers used in RTL representation is much larger than the number of registers of the target ASIP. Hence, it is necessary to assign the identical registers in RTL representation to real registers of the target ASIP. In the process of register allocation, if some registers in RTL representation cannot be assigned to real registers because of the shortage of registers in the target ASIP, they are saved to the stack area described in the TDM, which is accessed using the address pointer. MCC performs register allocation by using the algorithm used in GCC [15].

After generating the assembly code of the application program, MASM translates the assembly code into a binary file which can be simulated on MISS. The retargeting of MASM is controlled by translation rule file (TRF) which defines the mapping rule between assembly mnemonic and encoding bit pattern for each instruction.

2) Instruction-Level Execution Trace Generation (Step E2): MetaCore Instruction Set Simulator (MISS) simulates the target ASIP by interpreting the effects of instructions in the object code and generates an instruction-level execution trace of the benchmark program. MISS can also be reconfigured to support the variation in the architecture and instruction set based on the information in the instruction definition file (IDF) which consists of encoding bit pattern and the sequence of function names required to simulate the instruction. The details of MISS can be found in [17].

3) Performance Analysis (Step E3): Based on the analysis of an instruction-level execution trace of the benchmark program,



(a) original instruction sequence

```
ABS a0, ar1     ; a0=|mem[ar1]|
CLR a1          ; a1=0
ADD a1, ar2     ; a1=mem[ar2]|
CMP a1, a0
B.ltz L1        ; if(a1<a0) pc=L1
CLR a1          ; a1=0
ADD a1, a0      ; a1=a0
L1:
SUBI a1, 1      ; a1=a1-1
```

(b) instruction distillation

```
CLR a0          ; a0=0
ADD a0, ar1     ; a0=a0+|mem[ar1]|
B.gtz LL1       ; if(a0>0) pc=LL1
SUBR a0, 0      ; a0=0-a0
LL1:
CLR a1          ; a1=0
ADD a1, ar2     ; a1=mem[ar2]|
CMP a1, a0
BLTZ L1         ; if(a1<a0) pc=L1
CLR a1          ; a1=0
ADD a1, a0      ; a1=a0
L1:
SUBI a1, 1      ; a1=a1-1
```

(c) instruction condensation

```
ABS a0, ar1     ; a0=|mem[ar1]|
CLR a1          ; a1=0
ADD a1, ar2     ; a1=mem[ar2]|
MAX a1, a0      ; a1=MAX(a1,a0)
L1:
SUBI a1, 1      ; a1=a1-1
```

Fig. 7. Examples of instruction *distillation* and instruction *condensation*.

MPA generates such statistics as the execution time and total hardware cost. If the current processor specification satisfies the design constraints given as required performance, cost, or performance/cost ratio, the user proceeds to design generation step. Otherwise, the user has to modify the processor specification.

MPA also helps the user optimize the target processor design toward the given application by providing such information as the UC of each instruction and the frequent sequence of contiguous instructions which can be redefined as a new application-specific instruction.

B. Instruction-Level Design Refinement

Two methods of instruction-level design optimization in the MetaCore system are instruction *distillation* and *condensation*. Distillation corresponds to eliminating infrequent instructions, while condensation corresponds to replacing a frequent sequence of contiguous instructions by a newly defined application-specific instruction. Fig. 7 shows an example of distillation and condensation. In this example, the original instruction sequence (a) is transformed into a new sequence (b) by distillation, i.e., substitution of the instruction ABS by a four-instruction sequence, which takes more clock cycles while requiring less hardware cost. On the other hand, frequent instruction sequence shown as the shadowed box in (a) can be replaced by a new instruction MAX by condensation as shown in (c), which takes fewer clock cycles, but requires additional hardware cost.

The two instruction-level optimization methods can be performed with the help of MPA. The major tasks of MPA for distillation are to dispel the O-set instructions which fail to satisfy the instruction selection criterion (see Section V) and to estimate its effect on the performance and cost. Although the effect of distillations can be calculated after the modification of processor descriptions and design evaluation (through steps E1–E3 in Fig. 5), the procedure will take substantial simulation time. Hence, to reduce the simulation time, MPA immediately performs distillations for the selected O-set instructions and evaluates the effect. The distillation procedure is straightforward, i.e., substituting the selected O-set instructions with the corresponding B-set instruction sequences.

The objective of MPA for condensation is denoting the frequent sequences of contiguous instructions. The sequence of instructions can be redefined as a new instruction which can reduce total execution time. To find out frequent sequences of contiguous instructions, MPA uses a structure known as a weighted control flow graph (WCFG), which is a directed graph with instructions as nodes. Each node is weighted with the number of execution of the corresponding instruction. Edges in the graph represent the control transfers between instructions. The WCFG is constructed after instruction-level simulation using MISS.

At first, MPA invokes an empty table, called instruction sequence table (IST), which consists of a number of entries each containing a subgraph and its occurrence. From the start node of the WCFG, MPA performs graph matching between the subgraph of current search window and previously founded subgraphs listed in IST. The size of search window is determined by the user constraint such as the maximum length of sequence of instructions and the maximum number of operands. If the current subgraph matches one of the previously founded subgraphs in IST, MPA updates the number of occurrences of the subgraph. Otherwise, the current subgraph is newly registered in IST. The matching between subgraphs is tested in a topological point of view. For example, the subgraph S0 in Fig. 8 is identical to S1. For S0 and S2, they are not identical, but topologically equivalent, i.e., the instruction sequence and the data dependency between instructions are the same for S0 and S2. We can see that S0 and S3 are different since the data dependencies in two code sequences are different. If the current subgraph is identical or topologically equivalent to one of subgraphs in IST, the two subgraphs are considered to be matched since both of two subgraphs can be defined as the same instruction.

When the search window reaches the end node and all nodes in the WCFG were tested, MPA reports the subgraphs whose occurrence exceeds the threshold as given by the user. After that,

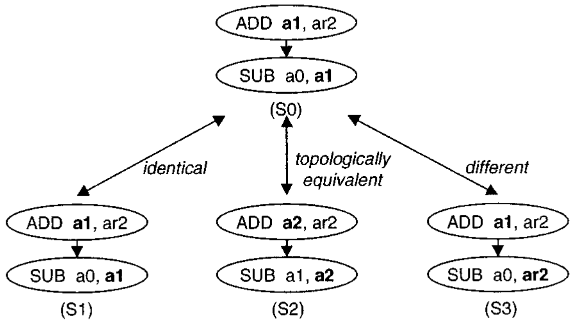


Fig. 8. Matching between subgraphs in a topological point of view. Two subgraphs S0 and S2 are not identical, but topologically equivalent since the instruction sequence and the data dependency between instructions are the same between S0 and S2.

based on the subgraphs, definition and design of new instructions can be done according the user's intuition.

C. Design Generation

In the design generation stage, a synthesis tool called SMART is used to translate the given processor specification into the corresponding HDL code. The HDL code generation process as performed by SMART is divided into three steps, as shown in Fig. 9.

1) Macro Block Generation (Step G1): In this step, SMART generates necessary functional blocks that support instructions described in the behavioral specification. SMART accesses the parameterized macro block in the MetaCore system library and assigns the parameters for each macro block such as the size of register/memory and bit width of each functional block to the value specified in hardware configuration in MBL specification.

2) Control Path Synthesis (Step G2): In control path synthesis, a data-stationary control model [18] is used for the pipeline design. In this model, the opcode is transmitted through each stage of pipeline in synchrony with data. At each stage, the opcode together with a possible flag bit from the data path is decoded to generate the control signals necessary to drive the datapath. The decoders for each pipeline stage are derived from the relationship between instructions and control input of functional blocks.

The basic operations of controller such as program counter (PC) updating, branching, and interrupt handling of the MetaCore are predefined using HDL code. Hence, the control path synthesis in SMART only needs to generate the decoder logic for each pipeline stage.

3) Connectivity Synthesis (Step G3): Finally, SMART connects all control inputs and data input/output busses of each functional block to appropriate control outputs of decoder and system busses.

V. Design Example

In this section, we show the effectiveness of the MetaCore system and the proposed design methodology using an ASIP design targeted for global system for mobile communication (GSM) application [19].

Fig. 10 shows the block diagram of GSM system. In the design of ASIP targeted for GSM, we use the GSM base-band

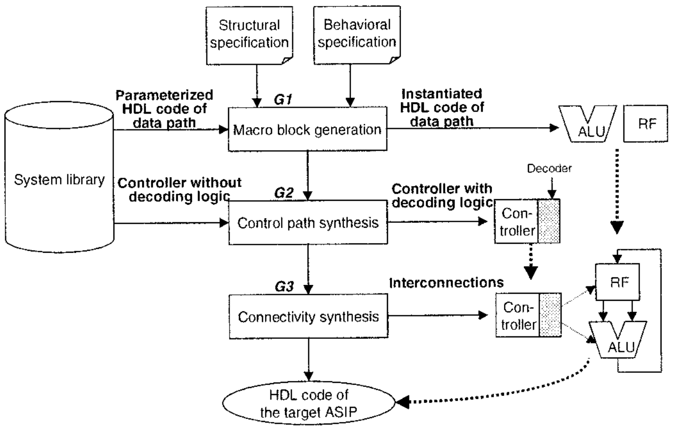


Fig. 9. HDL code generation performed by SMART consisting of three steps, i.e., macro block generation, control path synthesis, and connectivity synthesis.

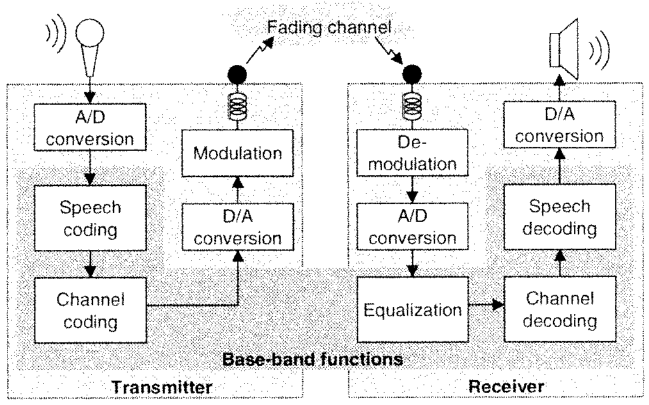


Fig. 10. Block diagram of GSM system.

functions, i.e., speech coding, channel coding, equalization, channel decoding, and speech decoding, which are programmed in C language as benchmark programs. The initial design is started with the specifications of predefined instruction set and the base architecture shown in Fig. 1. MCC and MASM are used to translate the given benchmark programs into machine codes, and MISS is used to simulate the programs and generate various instruction-level execution trace using 1-s speech data as input. Then, MPA analyzes the instruction trace and produces UC of the each predefined instruction, as shown in Table I.

Based on the instruction UC and the associated hardware cost, we developed a decision criterion which is applied to each instruction to decide whether to include it or not, i.e., the following inequality, which is an extension from [6], must be met to accept the instruction within the instruction set:

$$F(\Delta C, \Delta G) \triangleq (100/P) * \Delta C/C + \Delta G/G_m < 0$$

where C denotes the number of cycles for executing benchmark program, while G_m denotes the total gate count divided by the number of instructions, i.e., average number of gates needed to implement each of the predefined instruction. ΔC denotes cycle count gain due to the inclusion of the instruction (ΔC is negative when the cycle count is reduced), and ΔG represents additional gate count needed to implement the instruction. The above inequality criterion denotes that the instruction is accepted if it reduces at least $P\%$ (P is set to one in our experiment) of the total clock cycles while the additional hardware cost due to the inclusion of the instruction stays within the average number of logic gates per instruction. If multiple instructions shared the same functional unit such that the inequality cannot be applied to each instruction independently, we grouped them and tested the acceptance of the group using the inequality. The result of acceptance or rejection of each optional instruction according to the above rule is shown in Table II.

In Table II, UC denotes the number of occurrences of each instruction for each benchmark program. Cycles saved (CS) represents the number of clock cycles saved by each execution of the instruction. For example, instruction BCR (bit clear) in the third row of Table II, which clears arbitrary bit position of destination operand, can be substituted by the sequence of load high-immediate value (LHI), load immediate value (LDI), and AND (logical AND) instructions. Hence, the CS of BCR is two, because three-cycle instruction sequence is condensed into one instruction (cycle). Number of cycles (C) denotes the total number of execution cycles before the inclusion of the instruction. According to the instruction acceptance criterion, only four optional instructions, IDIV, MAC, MPY, and RND are accepted, where $F(\Delta C, \Delta G)$ is negative.

Further design refinement was performed based on the information on frequent sequences of contiguous instructions extracted by MPA. We have devised five application-specific instructions to replace the corresponding sequence of contiguous instructions. The results are listed in Table III. (The instructions listed in Table III also satisfy the same decision criterion used to decide the acceptance of predefined instructions.) For example, instruction SDIS is in the last row of Table III, which calculates the square distance of two complex numbers, which is used in Viterbi equalization. SDIS condenses ten-cycle instruction sequence into two cycles. Hence, the cycles saved is eight. The multiplication of UC and cycles saved represents the overall benefit due to the inclusion of the instruction.

To generate the target processor and system software equipped with the instructions in Table III, we modified the processor descriptions as described in Section III-B and added the information on a newly designed functional unit to the system library. Additionally, we also directly inserted the patterns for the instructions to MDF (see Section IV-A) such that C compiler can translate the corresponding sequences of primitive instructions to the instructions in Table III.

Fig. 11 shows performance and cost of various ASIP designs EM0, EM1, and EM2 as generated by MetaCore. EM0 is the initial design containing all predefined instructions of the MetaCore system. EM1 is the design obtained by instruction distillation method based on the accept/reject formula, $F(\Delta C, \Delta G)$, while EM2 is the design containing all application-specific instructions listed in Table III. EM1 has about 20% advantage over EM0 in terms of performance/cost ratio, while EM2 has 79% and 48% advantage over EM0 and EM1, respectively.

EM2 was selected as the final design for GSM application. Using SMART, HDL code was generated for EM2 from the MSL and MBL descriptions (the size of MSL and

TABLE I
USAGE COUNT OF EACH PREDEFINED INSTRUCTION FOR EACH FUNCTIONAL BLOCK OF THE GSM APPLICATION PROGRAM: OPTIONAL INSTRUCTIONS GROUPED AS A CLASS USE THE SAME FUNCTIONAL BLOCK

Instructions		Functional blocks in GSM program					Whole GSM program
		speech encoder	speech decoder	equal-izer	channle encoder	channel decoder	
B-set	.	2.7M	0.3M	38.1M	58.6K	4.1M	45.3M
O-set	ABS	21K	1.6K	0	0	0	22.6K
	BF0, BF1	0.5K	0	3.2K	1.7K	1.3K	6.7K
	BCR, BST, BTS	4.2K	0.8K	0	0	0	5K
	EXT	0	0	0.5K	0	0	0.5K
	IDIV	0	0	63.6K	21K	18K	102.6K
	MAC, MPY	1M	1.5M	4M	0	0	6.5M
	MRG	0	0	0	0	0	0
	RND	78.6K	136K	0	0	0	214.6K
Total usage count		3.8M	1.9M	42.2M	81.3K	4.2M	52.1M

TABLE II
RESULT OF ACCEPT/REJECT FOR EACH OF THE PREDEFINED INSTRUCTIONS IN THE OPTIONAL CLASS ACCORDING TO THE DECISION RULE: IN THIS EXPERIMENT, G_m IS $18\,064/40 \simeq 452$, WHEREAS 18064 IS THE TOTAL GATE COUNT NECESSARY TO IMPLEMENT ALL THE PREDEFINED INSTRUCTIONS AND 40 IS THE NUMBER OF INSTRUCTIONS OF THE PREDEFINED INSTRUCTION SET. P WAS SET TO ONE IN THIS EXPERIMENT

Instructions in the O(optional)-set	Usage count (UC)	Cycles saved (CS)	Total cycles saved ($\Delta C = UC * CS$)	Number of cycles (C)	$(100/P) * \Delta C/C$ ($P = 1$)	Incremen-tal gate count (ΔG)	$\Delta G/G_m$	$F(\Delta C, \Delta G)$	Accept/ Reject
ABS	22.6K	3	-67.8K	52.3M	-0.129	144	0.252	0.123	reject
BF0, BF1	6.7K	10	-67K	52.3M	-0.128	808	1.788	1.660	reject
BCR, BST, BTS	5K	2	-10K	52.2M	-0.019	64	0.141	0.122	reject
CLIP	0K	5	0K	52.2M	0	180	0.398	0.398	reject
EXT	0.5K	8	-4K	52.2M	-0.008	140	0.309	0.301	reject
IDIV	102.6K	12	-1.2M	53.4M	-2.3	114	0.252	-2.048	**ACCEPT**
MAC, MPY	6.5M	192	-1248M	1300M	-95.994	2161	4.780	-91.212	**ACCEPT**
MRG	0K	7	0K	52.2M	0	340	0.752	0.752	reject
RND	214.6K	5	-1.1M	53.3M	-2.06	120	0.265	-1.795	**ACCEPT**

TABLE III
FIVE USER-DEFINED APPLICATION-SPECIFIC INSTRUCTIONS FOR GSM APPLICATION. CMPY AND SDIS ARE TWO-CYCLE INSTRUCTIONS

Instruction	Description	Incremental gate count	Usage count (UC)	Cycles saved (CS)
CMPY	Multiply-accumulates on complex numbers	2096	771.6K	4
CVENC	Convolutional encoding	306	169.8K	23
HDIS	Calculate hamming distance between two numbers.	87	349.6K	17
ACS	Add-Compare-Select	101	350.1K	3
SDIS	Calculate square distance between two numbers	1808	939.3K	8

MBL descriptions for EM2 are 74 and 1079 lines, respectively). Peripherals such as serial/parallel input/output and timer were attached to EM2, and the result chip is called MDSP-II [21]. The lower level implementations, logic synthesis, and physical layout were done using the HDL code and verified with benchmark programs. Fig. 12 shows a photomicrograph of MDSP-II which consists of DSP core obtained as EM2, 12-Kbyte program memory, 8-Kbyte data memory, and peripherals. The special functional block called mobile communication acceleration unit (MCAU) supports the application-specific instructions listed in Table III. The dimension of the MDSP-II chip is 9.7 mm $\times$ 9.8 mm using the 0.6-μm CMOS trilevel metal (TLM) process technology. The peak performance is 55 MHz with the 5-V supply, with the average power consumption of 12 mW/MIPS.

VI. CONCLUSION

In this paper, we have presented a DSP-oriented ASIP development system called MetaCore that can generate efficient ASIP using benchmark-driven design methodology. The MetaCore system helps the user during the whole process of DSP ASIP design from design exploration to design generation. In the design exploration phase, users can obtain the optimized instruction set by deleting infrequently used instructions from the predefined instruction set of MetaCore and/or newly defining application-specific instructions with the help of such tools as assembler (MASM), C compiler (MCC), instruction set simulator (MISS), and performance analyzer (MPA) as provided by the MetaCore system. In the design generation phase, users can simply obtain the HDL code of the target ASIP equipped

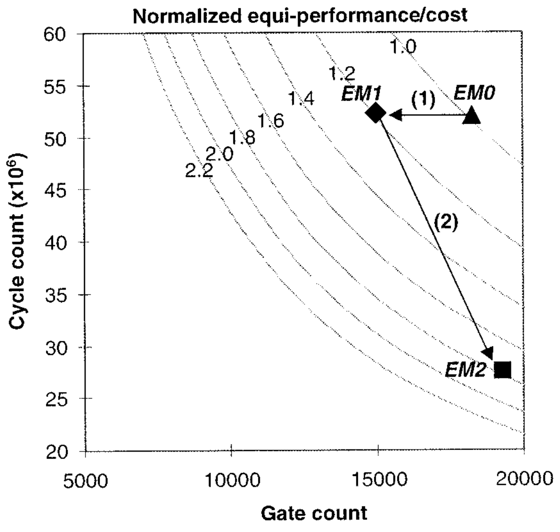


Fig. 11. Performance (as measured by the inverse of cycle count) and cost (gate count) of various ASIP designs EM0, EM1, and EM2 as generated by the MetaCore design framework. Cycle counts of EM0, EM1, and EM2 are 52.2, 52.2, and 27.3 M, respectively, while gate counts of EM0, EM1, and EM2 are 18.1, 15.0, and 19.4 K, respectively. The move (1) from EM0 to EM1 is to reduce the gate count by excluding some relatively less demanding instructions, while the move (2) from EM1 to EM2 is to significantly improve the performance with the adoption of some application-specific hardware blocks at the marginal increase of the gate count.

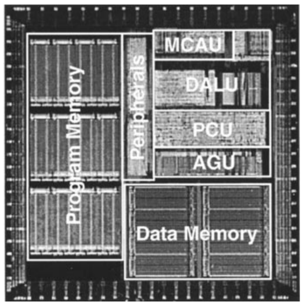


Fig. 12. Photomicrograph of MDSP-II.

with the user-defined application-specific instructions with the help of the HDL code generator, SMART. In the whole design process, the interaction between the user and the system software is provided by the structural/behavioral processor specification languages (MSL/MBL), which have been originally developed for the purpose of flexibility and simplicity in describing the target ASIP. The effectiveness of MetaCore system has been shown using real-world design examples. In the MetaCore system, the instruction set is optimized by the selection among the predefined instruction set and newly defining application-specific instructions based on the decision on information such as the execution time, gate count, and UC of each instruction. A programmable DSP processor called MDSP-II targeted for GSM application has been successfully demonstrated using the proposed MetaCore system.

References

[1] E. A. Lee, "Programmable DSPs: A brief overview," *IEEE Micro*, vol. 10, pp. 14–16, Oct. 1990.

[2] P. G. Paulin, C. Liem, T. C. May, and S. Sutarwala, "DSP design tool requirements for embedded systems," *IEEE J. VLSI Signal Processing*, vol. 9, pp. 23–47, Jan. 1995.

[3] J. Sato, A. Y. Alomary, and M. Imai, "PEAS-I: A hardware/software codesign system for ASIP development," *IEICE Trans. Fundamentals*, vol. E77-A, no. 3, pp. 483–491, Mar. 1994.

[4] R. Woudsma *et al.*, "EPICS, a flexible approach to embedded DSP cores," in *Proc. Int. Conf. Signal Processing Applications and Technology (ICSPAT)*, Oct. 1994.

[5] J. Sato, M. Imai, and N. Hikichi, "An integrated design environment for application specific integrated processor," in *Proc. Int. Conf. Computer Design*, 1991.

[6] B. Holmer and A. M. Despain, "Viewing instruction set design as an optimization problem," in *Proc. 24th Int. Conf. Microarchitecture*, 1991.

[7] I. Huang and A. M. Despain, "Synthesis of application specific instuction sets," *IEEE Trans. Computer-Aided Design*, vol. 14, pp. 663–675, June 1995.

[8] P. G. Paulin *et al.*, "FlexWare: A flexible firmware development envionment for embedded systems," in *Code Generation for Embedded Processors*. Norwell, MA: Kluwer Academic, 1995, pp. 67–84.

[9] D. Lanneer *et al.*, "Chess: Retargetable code generation for embedded DSP processors," in *Code Generation for Embedded Processors*. Norwell, MA: Kluwer Academic, 1995, pp. 85–102.

[10] T. Wilson *et al.*, "An ILP-based approach to code generation," in *Code Generation for Embedded Processors*. Norwell, MA: Kluwer Academic, 1995, pp. 103–118.

[11] C. Liem, *Retargetable Compilers for Embedded Core Processors*. Norwell, MA: Kluwer Academic, 1997.

[12] S. Hanono and S. Devadas, "Instruction selection, resource allocation, and scheduling in the AVIV retargetable code generator," in *Proc. 35th Design Automation Conf.*, 1998.

[13] *Application Specific Microprocessor Solutions: Data Sheet for Xtensa V1*, Tensilica, Inc., 1998.

[14] C. Liem, P. G. Paulin, and A. Jerraya, "Address calculation for retargetable compilation and exploration of instruction set architecture," in *Proc. 33rd Design Automation Conf.*, 1996.

[15] R. M. Stallman, *Using and Porting GNU CC for Version 2.6*: Free Software Foundation Inc., Sept. 1996.

[16] A. V. Aho, M. R. Sethi, and J. D. Ullman, *Compilers—Principles, Techniques, and Tools*. Reading, MA: Addison-Wesley, 1986.

[17] J. Y. Lee *et al.*, "Architecture selection of a flexible DSP core using reconfiguration system software," in *Proc. ISCAS*, May 1998.

[18] P. M. Kogge, *The Architecture of Pipelined Computers*. New York: McGraw-Hill, 1981.

[19] L. Hanzo and J. Stefanov, *Mobile Radio Communications*. New York: IEEE Press, 1992.

[20] V. Zivojnovic, J. Martinez, C. Schlager, and H. Meyr, "DSPstone: A DSP-oriented benchmarking methodology," in *Proc. Int. Conf. Signal Processing Applications & Technology (ICSPAT)*, Oct. 1994.

[21] B. W. Kim, J. H. Yang, and C. M. Kyung, "MDSP-II: A 16-bit DSP with mobile communication accelerator," *IEEE J. Solid-State Circuits*, vol. 34, no. 3, 1999.

[22] A. V. Aho and S. C. Johnson, "Optimal code generation for expression trees," *J. ACM*, vol. 23, no. 3, July 1976.

[23] G. D. Micheli, *Synthesis and Optimization of Digital Circuits*. New York: McGraw-Hill, 1994.

Jin-Hyuk Yang received the B.S. degree in electronic engineering from Kyungpook National University, Korea, in 1992 and the M.S. degree in electrical engineering from the Korea Advanced Institute of Science and Technology (KAIST), Taejon, Korea, in 1994. He is currently working toward the Ph.D. degree in electrical engineering at KAIST.

His current research interests include VLSI architecture for DSP and general-purpose microprocessor and design methodology for VLSI.

Byoung-Woon Kim was born in Masan, Korea, in 1971. He received the B.S. degree in electronics engineering from Kyungpook National University, Taegu, Korea, in 1994 and the M.S. degree in electrical engineering from the Korea Advanced Institute of Science and Technology (KAIST), Taejon, Korea, in 1996. He is currently working toward the Ph.D. degree in electrical engineering at KAIST.

His research interests include hardware–software codesign for application-specific DSP and low-power DSP processor design.

Sang-Joon Nam received the B.S. and M.S. degrees in electrical engineering from the Korea Advanced Institute of Science and Technology (KAIST), Taejon, Korea, in 1993 and 1995, respectively. He is currently working toward the Ph.D. degree in electrical engineering at KAIST.

His current research interests include multiple-issue microprocessor design, multimedia VLSI design, and verification methodology.

Young-Su Kwon was born in Bong-Hwa, Korea, on January 19, 1976. He received the B.S. degree in electrical engineering from the Korea Advanced Institute of Science and Technology (KAIST), Taejon, Korea, in 1997. He is currently working toward the M.S. degree in electrical engineering at KAIST.

His current research interests include DSP core development and 3-D graphics hardware design.

Dae-Hyun Lee received the B.S. degree in electrical engineering from Seoul National University, Seoul, Korea, in 1993 and the M.S. degree in electrical engineering from the Korea Advanced Institute of Science and Technology (KAIST), Taejon, Korea, in 1995. He is currently working toward the Ph.D. degree in electrical engineering from KAIST.

His research interests include hardware–software codesign and low-power system-on-chip design.

Jong-Yeol Lee was born in Taejon on January 24, 1970. He received the B.S. and M.S. degrees in electrical engineering from the Korea Advanced Institute of Science and Technology (KAIST), Taejon, Korea, in 1993 and 1996, respectively. He is currently working toward the Ph.D. degree in electrical engineering at KAIST.

His research interests include CAD, high-level synthesis, and code generation for various processors.

Chan-Soo Hwang was born in Suwon, Korea, on July 4, 1975. He received the B.S. degree in electrical engineering in 1997 from the Korea Advanced Institute of Science and Technology (KAIST), Taejon, Korea. He is currently working toward the M.S. degree at KAIST.

His research interests include equalization in wired, wireless, and storage channels and multirate signal processing and their implementations.

Yong-Hoon Lee (M'84) was born in Seoul, Korea, on July 12, 1955. He received the B.S. and M.S. degrees in electrical engineering from Seoul National University, Seoul, in 1978 and 1980, respectively, and the Ph.D. degree in systems engineering from the University of Pennsylvania, Philadelphia, in 1984.

From 1984 to 1988, he was an Assistant Professor with the Department of Electrical and Computer Engineering, State University of New York at Buffalo. Since 1989, he has been with the Department of Electrical Engineering, Korea Advanced Institute of Science and Technology (KAIST), Taejon, Korea, where he is currently a Professor. His research activities are in the area of one- and two-dimensional digital signal processing, VLSI signal processing, and digital communication systems.

Seung-Ho Hwang (M'89) received the B.S. degree in electronics engineering from Seoul National University, Seoul, Korea, in 1979, the M.S. degree in electrical engineering from the Korea Advanced Institute of Science and Technology (KAIST), Taejon, Korea, in 1981, and the Ph.D. degree in electrical engineering and computer science from the University of California, Berkeley.

He was at the University of California, Berkeley, as a Postgraduate Researcher and at Schlumberger Technologies, Inc. as a Senior Software Engineer from June 1989 to September 1990. He joined KAIST in September 1990 as an Associate Professor in the Department of Electrical Engineering, and during his sabbatical year he was at Silicon Image, Inc. His current reseaech interests include CAD algorithms for high-level synthesis, power estimation, hardware/software codesign, and VLSI design.

In-Cheol Park (M'92) received the B.S. degree in electrical engineering from Seoul National University, Seoul, Korea, in 1986 and the M.S. and Ph.D. degrees in electrical engineering from the Korea Advanced Institude of Science and Technology (KAIST), Taejon, Korea, in 1988 and 1992, respectively.

From May 1995 to May 1996, he was at the IBM T. J. Watson Research Center, Yorktown, NY, as a Postdoctoral Member of the Technical Staff in the area of circuit design. He joined the Department of Electrical Engineering at KAIST in June 1996 as an Assistant Professor. His current research interest includes CAD algorithms for high-level synthesis and VLSI architectures for general-purpose microprocessors.

Chong-Min Kyung (M'81–SM'99) received the B.S. degree in electronic engineering from Seoul National University, Seoul, Korea, in 1975 and the M.S. and Ph.D. degrees in electrical engineering from the Korea Advanced Institude of Science and Technology (KAIST), Taejon, Korea, in 1977 and 1981, respectively.

After graduation from KAIST, he was at AT&T Bell Laboratories, Murray Hill, NJ, from April 1981 to January 1983 in the area of semiconductor device and process simulation. In February 1983, he joined the Department of Electrical Engineering at KAIST, where he is now Professor. His current research interests include microprocessor/DSP architecture, chip design, and verification methodology. He is Director of the IDEC (Integrated Circuit Design Education Center), which was established to promote the VLSI design education in Korean universities through CAD environment setup and chip fabrication services and providing various educational materials and media related to integrated circuits and systems design. He is also Director of CHiPS (Center for High-Performance Integrated Systems) at KAIST. During 1993–1994, he served as the Asian Representative in the ICCAD (International Conference on Computer-Aided Design) Executive Committee. He also served as Vice Chairman of the 1999 COOLChips II held in Kyoto, Japan, and as Cochair of the program committee of ASP-DAC (Asia and South Pacific Design Automation Conference) 2000. He received the Most Excellent Design Award and Special Feature Award in the University Design Contest at the ASP-DAC 1997 and 1998, respectively. He received the Best Paper Award at the 36th DAC (Design Automation Conference), New Orleans, LA, the 10th ICSPAT (International Conference on Signal Processing Applications and Technology), Orlando, FL, in September 1999, and the 1999 ICCD (International Conference on Computer Design), Austin, TX.