Design Automation for Embedded Systems, 4 , 329–351 (1999)

FACE: Fine-tuned Architecture Codesign Environment for ASIP Development

I-HORNG JENG paul@cc.ee.ntu.edu.tw
Dept. of Electrical Engineering, National Taiwan University, Taipei 106, Taiwan, R.O.C.

FEIPEI LAI paul@cc.ee.ntu.edu.tw
Dept. of Information Engineering, National Taiwan University, Taipei 106, Taiwan, R.O.C.

YUH-DAR TSENG ydtseng@via.com.tw
VIA Technologies, INC., Taipei 106, Taiwan, R.O.C.

Abstract. High-performance, reliable, and robust products with a short development schedule are general design aims. FACE was developed to achieve these goals, including the organization of a design flow, a frequency-driven information analyzer, compiler techniques (code generator and instruction optimization), and a hierarchical object design library. This paper explores the design space of a retargetable compiler and a reconfigurable hardware, which combine both software and hardware reprogrammability. The environment, FACE, we have developed allows us to quickly move the functions between software and hardware in a state of flux. Finally, it generates the application specific integrated processor (ASIP) and a compiler for the new ASIP architecture. The case study is considered which demonstrates the efficiency in ASIP design of FACE.

Keywords: Code Generator, Retargetable Compiler, Hardware Library

1. Introduction

Historically, the evolution of hardware-design entries in terms of paradigm shifts: First there was polygons (physical layout), then netlist (gate-level), then HDL code (behavioral synthesis). In the same matter, the developmental process of software is also from low-level to high-level. It goes without saying that these developments have brought about a revolution in computer. This track explicitly points out that the developments of more complex structures urgently need a higher-level design hierarchy. However, the developments of software not only seldom regard the impact on hardware architecture but also often neglect many potential factors, and vice versa. As a result, the desired consequence is not obtained. Hence, it is quite clear that performance is strongly interdependent on hardware and software because a computer design would have many aspects to be considered, including instruction set architecture, code optimization, hardware organization and technology, and physical implementation.

Firstly, J. Sato et al. [1] propose an integrated design framework for application-specific instruction set processors. The top-down framework customizes an instruction set from a super set, decides the hardware architecture derived from the GNU CC's [2] abstract

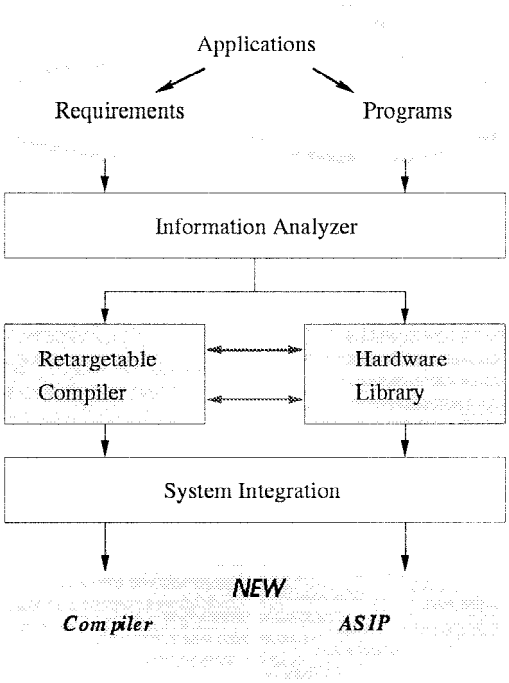

Figure 1. Overview of FACE.

machine model. Based on the similar design environment, I.-J. Huang et al. [3] use bottom-up method and synthesize the instruction sets directly in order to find new instructions.

These two work focus on smaller application benchmarks, only according to their characteristics, to select their instructions. Both are lack of large benchmark's behavior information because larger ones (e.g. SPEC) call many system functions that is in object-code format and can't be loaded into the simulator. Moreover, their simulators are not compiled by mature CAD tools and seem not accurate in hardware cost (e.g. area, power) estimation.

The above have given us a direct stimulus to construct a high-level design environment, hardware/software codesign, called FACE (Fine-tuned Architecture Codesign Environment). Besides I.-J. Huang et al.'s annealing method, we create new instructions according to larger benchmark's characteristics and peephole optimizations. And through a three-step decision-making process, we make balancing tradeoffs for customized ASIP design. The brief overview of FACE is shown in Figure 1. A system specification in FACE consists of the following four major parts:

1. Information analyzer: The analyzer extracts various kinds of information from application programs and requirements, especially with respect to the high-frequency fractions. These data are valuable for the partition and optimization of hardware and software.

2. Retargetable compiler: We use a machine independent front-end, a code generator and simple machine description to facilitate users constructing a desired compiler. Moreover, this compiler can generate better code according to the characteristics of applications.

3. Hardware library: We use object-oriented concept for codesigning and implementing systems. This paradigm increases design extendibility and decreases the cost and complexity of maintenance.

4. System integration: The design follows the steps of our design procedure to integrate the hardware and software in the same development process. FACE finally outputs the ASIP architecture and the corresponding compiler.

The remainder of this paper is organized as follows: In section 2, we introduce the framework of FACE and describe the flow of development processes step by step. It begins by transferring requirement specification to objective function and has a global view of the entire codesign life cycle. The next two sections discuss what concepts and methods we use to implement the retargetable compiler and hardware library, respectively. In section 5, we describe the interface between hardware and software, tuning box. It combines the compiler and hardware library to guide designers in making decisions. Then, we illustrate a multimedia example of using FACE in an ASIP design and show our experimental results. Last of all, we make some conclusions and future work for FACE.

2. FACE Design Flow

System development process is a very sluggish and reticular activity. In order to quickly design a complex and well-done architecture, a systematic development method must be adopted. So the design procedure is developed to incorporate into the FACE environment.

2.1. Design Methodology

Before dealing with the design procedure and detailed framework of FACE, the combination of top-down and bottom-up approaches is first investigated. Our design methodology employs both top-down design and bottom-up design to control the complexity and cost of hardware maintenance, respectively. As in a project, the top-down work first progresses according to the analysis and requirements. We decompose a system into functional blocks, instead of keeping on breaking this process down. The benefits of starting with top-down design are:

- System is directly modeled for the application, so quality and performance tradeoffs can be made at the earlier stage.
- System is more resilient to modify and extend.
- Large designs are easier to manage.

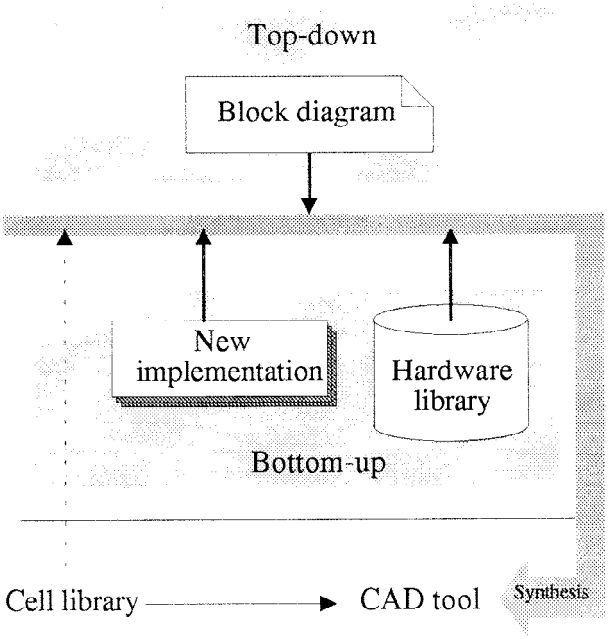

Figure 2. The design methodology in FACE.

- Different parts of the design can work at the same time.
- Strong connection among blocks.

After a desired functional view is complete, these blocks are made up from our hardware library, we then connect them to entire system. If there is no desired function in the library, we refine it at the lower levels of abstraction, then build for later use. The major benefit of bottom-up design is that the reuse of blocks reduces development effort, such as re-implementation, testing, and maintenance time. To sum up, the design methodology in which we construct the system at a very high level of abstraction using top-down approach, then each block is modeled by hardware library using bottom-up approach. The design methodology is presented in Figure 2. By the complementary relationship of these two approaches, all the advantages are both utilized and the effectiveness of design is further enhanced.

2.2. Framework of FACE

In FACE, VHDL [4] is used to describe the hardware architecture and the software description is represented in C. Our intention is to take these two parts together, simplify them for the developer. The advantage of this integration is that it is easy to move functions

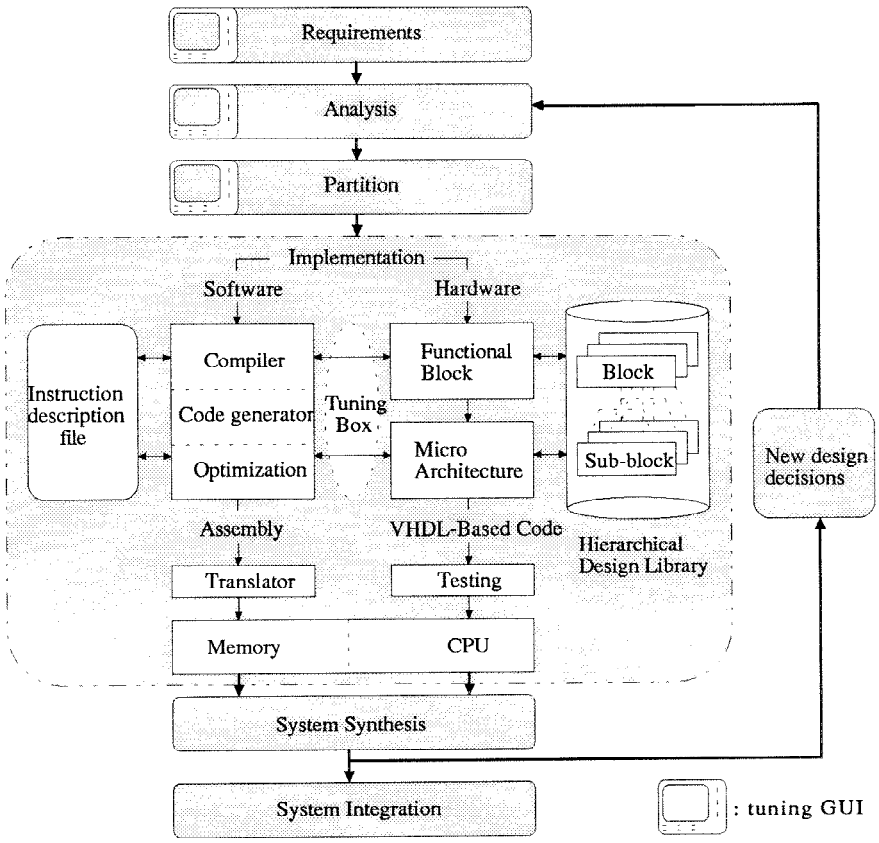

Figure 3. Detailed view of FACE.

between hardware and software. The overall structure of FACE is shown in Figure 3. We use Ackermann example program to illustrate the seven steps of the environment:

```
acker(n,m) { if(n==0) return m+1;
             else if(m==0) return acker(n-1,1);
             else return acker(n-1, acker(n,m-1)); }
```

1. Requirements: The requirements step transfers application attributes, from known information given by user, to general rules. User can provide larger benchmark's behavior information here such as multimedia characteristics [5]. Then it determines design goals and objective functions. Finally, gathers all resources necessary for the project. For example, Ackermann is a recursive function growing fast with small values. So, it

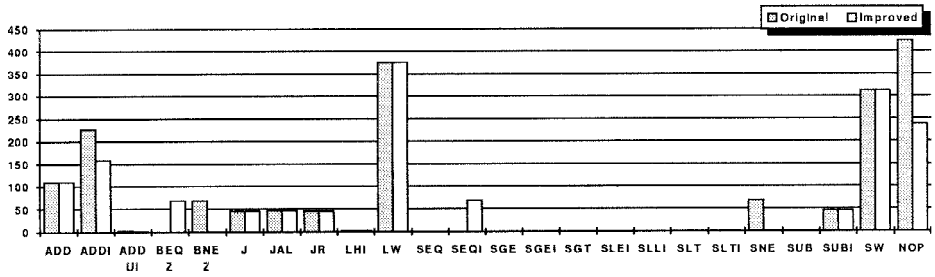

Figure 4. The instruction execution counts and distribution for Ackermann.

needs fast procedure call and enough stack space. Its objective function may be

$$OBJ = MIN(t_i) + MAX(A_s)$$

t_i: speed of procedure-call i

A_s: area of stack s

2. Analysis: The analysis step analyzes the behavior of application programs, and measures the instruction usage distributions and instruction patterns for peephole optimization. Ackermann's instruction counts and distribution statistics is illustrated in Figure 4. If not satisfied, users can add their new instruction templates into the existing instruction description file and thus tuning the instruction distribution results.

3. Partition: Based on the information gathered in steps 1, 2, and replacement rules, we evaluate an optimal instruction set and make out a draft of the computer structure. For example, the rules of load/store instruction in Ackermann are denoted in linear algebra:

$$Load^{+} = (1,0) \cdot Add \oplus (0,1) \cdot Load$$
$$Store^{+} = (1,0) \cdot Add \oplus (0,1) \cdot Store$$

+: index mode of

$\oplus$: linear combination

Originally, there exist lots rules of instruction combinations in experience. Then through friendly GUI (Graphic User Interface), users can combine their replacement rules to make a better and customized partition.

4. Implementation: First, we find the desired functions in hardware library and concurrently modify the instruction description file. Then we develop compiler optimization techniques. If there is no off-the-shelf component for those functions, we implement and store them in library. In addition, we develop a tuning box to trigger the instruction replacement options then do an actual replacement job. The instruction descriptions of index-mode load/store in Ackermann are described as

```
Load: (set (r SI 0) (mem: SI (plus: SI (r SI 1) (r SI 2))))
Store: (set (mem: SI (plus: SI (r SI 0) (r SI 1))) (r SI 2))
```

SI: Single Integer mode

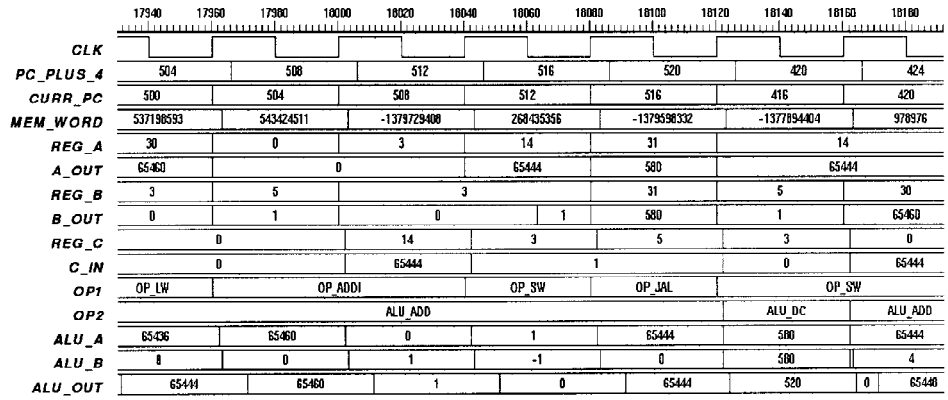

Figure 5. The Waveform simulation snapshot for Ackermann.

After a three-step decision-making process (steps 1, 2, and 3), we make balancing trade-offs here for customized hardware/software design and guide users to select. However, this time-consuming step is consumed mainly on testing not synthesizing. Thus, we develop objects sharing software and hardware codesign information to reduce errors and automate the hardware-module editing as much as possible.

5. System synthesis: To verify the correctness of whole implementations, we execute some benchmarks with the hardware synthesized by the SYNOPSYS tool. The simulation result snapshot for Ackermann is shown in Figure 5.

6. New design decisions: If the design goals are not matched exactly, we go back to step 2. Some changes, such as adding new instructions or decreasing register number or tuning the addressing space, will dramatically affect the execution frequencies of instructions. It is therefore necessary to repeat step 2 after revision either in instruction set or in hardware architecture. For example, FACE finds and suggests user to substitute new instruction for adjacent compare-jump instruction patterns:

```
Compare: (set (cc0) (compare (r SI 0) (r SI 1)))
Jump: (set (pc) (if_then_else (ne (cc0) (const_int 1)) (label_ref 2) (pc)))
New_instruction: (set (pc)
                      ((cc0) (compare (r SI 0) (r SI 1)))
                      (if_then_else (ne (cc0) (const_int 1)) (label_ref
2) (pc)) )
```

   The design procedure for us is incremental type of design methodology. The next cycle will be initiated under the following two issues:

   - Enhance the current version.
   - Port to new applications.

7. System integration: Finally, FACE generates a compiler and the ASIP architecture.

As described above, generating our compiler back-ends automatically needs two components: machine description file and instruction description file. Specially, machine description file contains some information like register layout (name, number, fixed, call-used, etc.), stack layout (offset, pointer, etc.), and storage layout, which is similar with GNU CC's md.h and can be decided in advance. Then, the instruction description file is done by user or by default. Next, automatically selecting modifications to instruction sets is processed during pattern matching. We use Replacement1 to ReplacementN options to do the real replacement job but firstly tuning box's rule engine to make a selection decision. Besides the tuning box of step 4, we provide a manual tuning GUI for each decision-making step (1, 2, and 3) manually as shown in Figure 3 to make a better and customized optimization.

3. Compiler

In the traditional system design, compiler is not developed until the hardware has been built. This causes software a great deal of constraints, so we concurrently develop compiler and hardware. Our compiler implementation is based on ARDEN [6] [7] which we have developed. In this paper, we briefly describe the components and process flow of the compiler, details of actual algorithms for the whole compiler procedure can be found in [6] [7].

3.1. Prototype of Compiler

Figure 6 shows the flow of compiler generation in FACE. There are four critical components in this compiler:

1. Front-end processor: The front-end processor is adapted from GNU CC [2]. The task of this component is translating C language into intermediate representation language (RTL), a parenthesized expression form.

2. Code generator: A code generator recognizes the architecture description file and outputs the object code. This architecture description file combines instruction description file described by user and machine description file contains constant definitions of register, storage, and stack layout. Specially, the machine description file shares the same information with hardware library. This is the key feature to make the compiler retargetable. Furthermore, the instruction selection is accomplished by searching through instruction description based on a pattern match routine [6] [7]. The syntax of instruction description is as follows:

```
%define_insn
    @Macro expression@
        { Template }
    @Replacement 1@
```

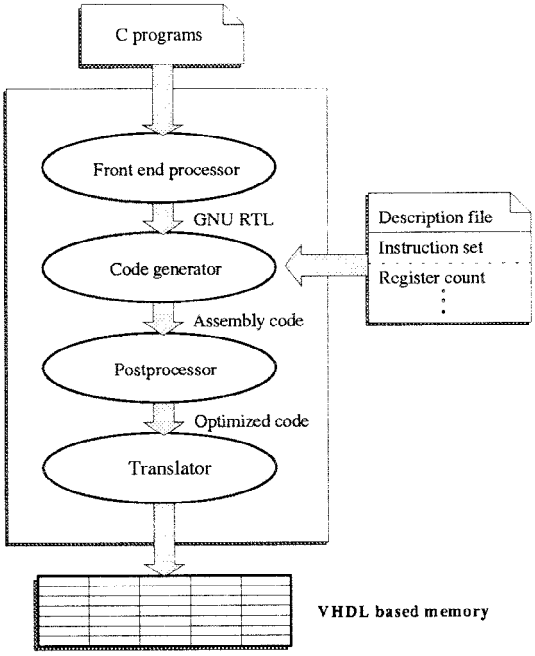

Figure 6. The prototype of compiler in FACE.

```
    { Action 1 }
@Replacement 2@
    { Action 2 }
        •
        •
@Replacement N@
    { Action N }
    @Statistical Data@
%
```

The macro expression defines the functions that will be expanded in other entries, thus shrinking the redundancy. The replacement option is used to satisfy the design goal as accurately as possible and triggered by tuning box's rule engine to make a suitable decision. Actions are the output object code for this rule. If one of the replacement rules is selected, the corresponding action will be output. The statistical data is for partitioning hardware and software.

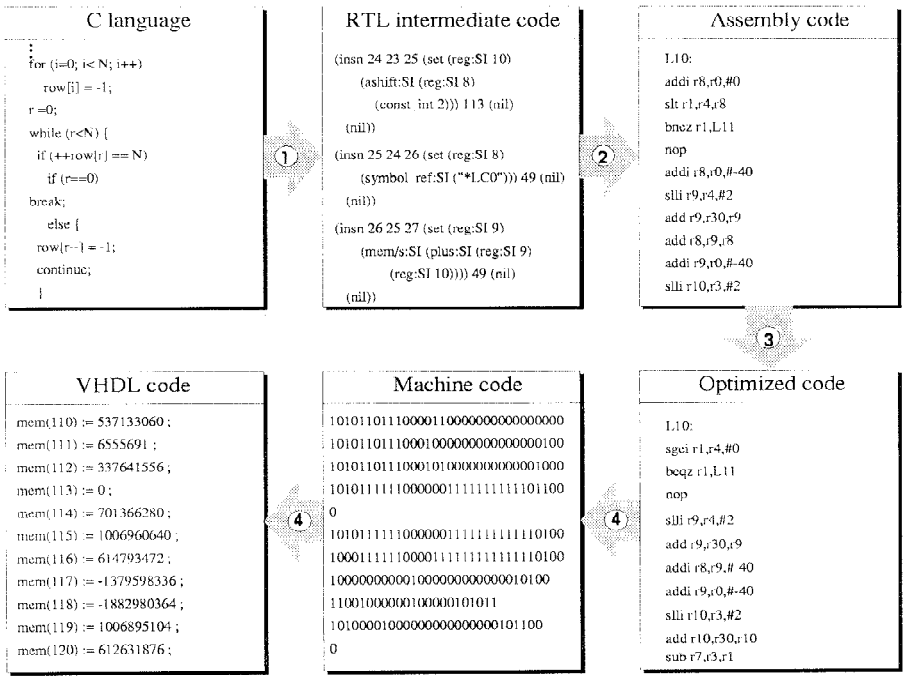

Figure 7. The six stages in our compiler.

3. Postprocessor: In order to improve performance of a new architecture, we use a popular technique, peephole optimization [7], for retargetable compiler. In addition, we also develop other approaches [8] to increasing performance, such as instruction scheduling and register allocation. The results of these techniques will be described in case study.

4. Translator: The translator translates assembly code to the target machine code, then generates the memory map of VHDL behavioral description. It combines functions of assembler and loader. That is, not only it translates assembly codes into embedded machine codes, but further translates them into VHDL memory codes (as shown in Figure 7). Memory codes mean machine codes that have been loaded into VHDL behavioral memory variables and ready to execute. The memory code can be input to the ASIP for simulation execution.

We believe that plenty of instruction templates built in GNU CC can cover most of ASIP. Even if we don't have one-to-one mapping between every RTL template and assembly code (under 2%), we can still include them by our peephole optimizer [7].

As depicted in Figure 6, the integrating four parts have been implemented in a prototype C compiler generator. First, the front-end processor translates C programs into RTL. Then the code generator maps the RTL into assembly code according to the patterns in machine

description file. After generating assembly code, some optimization schemes are performed to improve execution speed. Last, the optimized code is translated into VHDL code.

To see an example of a quick use of the compiler, a simple program in Figure 7 is sequentially translated into different intermediate languages from C to VHDL, in other words, the input of the compiler is C programs and the output is VHDL based memory code. Note that in Figure 7, the numbers, 1, 2, 3, 4, in the circles represent the four parts in the compiler, a front-end processor, a code generator, a postprocessor, a translator, respectively.

3.2. Partition

In FACE codesign environment, we concurrently develop compiler and hardware. Architecture description file shares information in-between to make partition. Codesign is concerned with providing support, which is generally automated, for the identification of hardware and software components [9]. The goal is to optimize an implementation in terms of factor such as speed, cost and etc., and to satisfy the design constraints.

More commonly, the specification of hardware system uses an implementation language such as VHDL [10] and migrates part of the functionality from this into software source code, where as in others software parts are translated into a hardware description language. In our VHDL behavioral descriptions, which are based on identification of DLX in advance.

There are three general ways to decide how to partition. They are with regard to FACE's analysis, partition, and tuning respectively.

- Performance experiments on prototype system (e.g. Srivastava and Brodersen 1992).

- The analysis of system cost factors and the optimization of these through mathematical techniques (e.g. Kumar et al. 1993).

- The analysis of system cost factors and the optimization of these through guided user selection (e.g. D'Ambrosio and Hu 1994).

It will normally be the case that, if the time to complete operation is tightly constrained, a hardware solution will be the preferred choice. Alternatively, if the availability of hardware that can be configured for special actions is limited, then a software solution might be the best choice. However, modern signal processors, coupled with the use of parallelism, can achieve very high performance rates with software algorithms and, similarly, hardware synthesized from high-level specifications can perform very complex functions. Thus, we exploit linear algebra concept to solve the partition decision tradeoffs accurately and efficiently. Stated very simply, there exist many basic set of instruction that can extend to any instruction set. Taking this view, we can use linear transformation technique to compute the balancing tradeoff point and make a best partition.

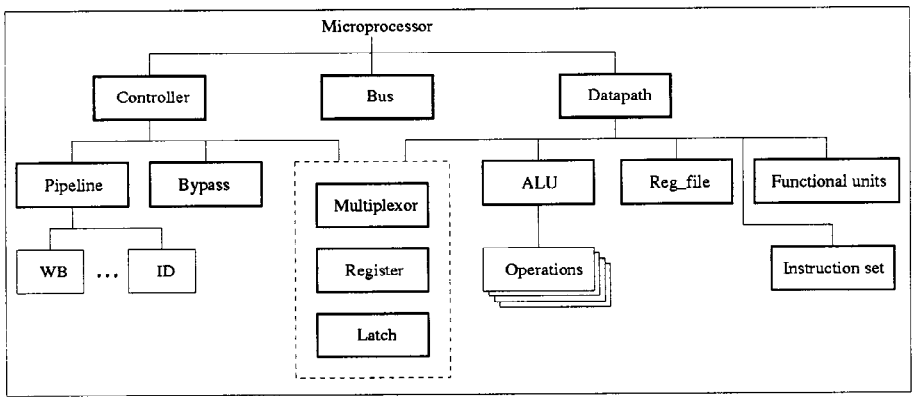

Figure 8. Hierarchical hardware design library.

4. Hierarchical Object Design Library

A large system can be divided into functional blocks with several levels of details from design specifications (Figure 8). Thus, this demands a hierarchical design approach to speeding the partitioning and reducing development effort through reuse. P. Darche et al. [11] has proposed a very similar data structure but totally different behavioral mechanism from us. They are dynamic run-time entity and ours is static compile-time. In FACE, we have been developing a hardware design library to efficiently produce a system prototype with object oriented concept.

4.1. Object-Based Design

Object concept is not a new method for designing software. Over the past two decades, a number of excellent ideas and issues have been proposed in this field. It provides a useful platform for fast prototyping. But the basic concepts are not applied in the hardware development until recently [12].

FACE's hardware library is constructed in objects. For instance, datapath can be composed of the ALU, register file, special functional unit, some registers and latches. After the decomposition, a system is partitioned into classes and objects from these classes in hierarchies, then the different modules can be designed concurrently. It is therefore easy to add new functions in the architecture. This composition depends on what functions of a model the designers want, such as shown in Figure 9. It makes the system easy to understand, reuse, verify and extend.

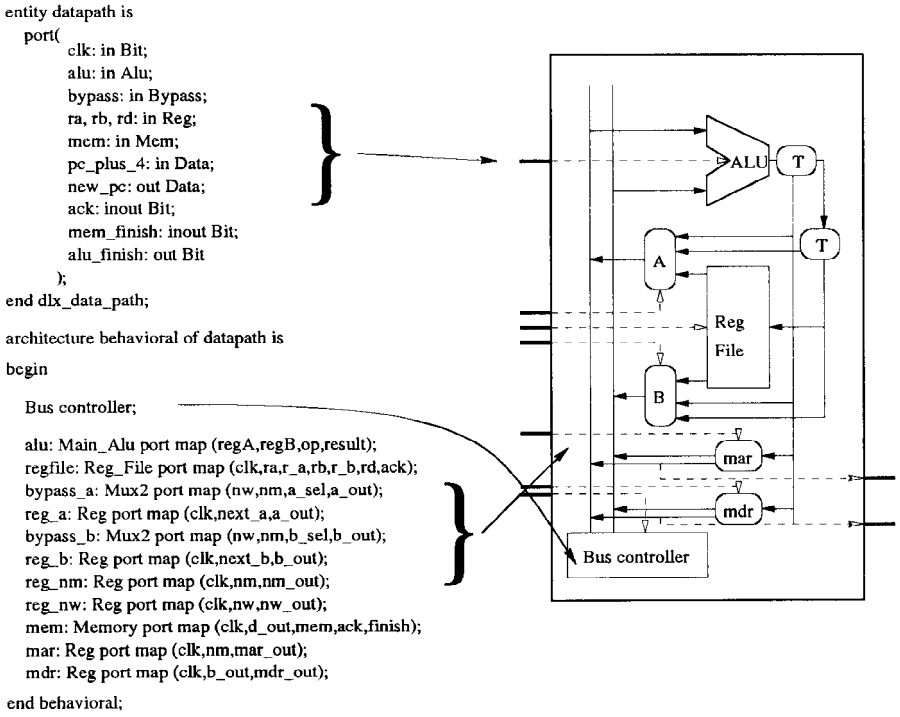

Figure 9. The composition of datapath.

4.2. Object-Oriented Design

Most of data-processing instructions can be in the unit of *Byte*, *Subword* or *Word*. Not only ALU and instruction set are to be designed simultaneously, but also the control units are under consideration concurrently. Because instruction can bind to *Byte*, *Subword* and *Word* type dynamically, which satisfies the polymorphism characteristics.

Additionally, *Subword* and *Word* data type can inherit the attributes and behaviors of *Byte* class. Then just extend the width and tune a little bit statements to inherit it. Not only the homogeneous instructions are inheritable, heterogeneous ones apply the characteristics. For example, jump instruction belonging to control unit may inherit Add instruction class because program counter may be increased by the ALU unit.

5. Tuning

The most stubborn and sophisticated phase throughout design processes is making tradeoffs. Even tuning small portions of a system needs momentous efforts on the exploration of

hardware and software tradeoffs. And a number of alternatives need to be evaluated for a function. To simplify this process, we establish a tuning box to store replacement rules between hardware/software as a knowledge base.

5.1. Tuning Box

A complete system consists of numerous functions to be performed. It is hard to make suitable decisions under several constraints. A function performs in hardware, it means the execution speed, cost, and area will be rising. On the contrary, if a function performs in software, it seems that most factors will be going down. To reach the highest performance under predefined conditions, we make tradeoffs in two levels: global functional view, and local processors. The primary principle of making higher level decisions, including creating instruction set and partitioning hardware, is based on three-step decision-making mechanism: requiring, analyzing, and partitioning. The corresponding representation methods are objective functions [13], peephole optimization patterns [7], and replacement rules, respectively. They are, however, not enough to approach a required ASIP architecture, there are still many shortcomings to address somewhere in the whole processor. So, two issues are considered in tuning the local processor:

- Reduce the hardware overhead: If a function is not in the timing critical path, it is replaced by a sequence of instructions.

- Optimize the execution speed: To meet timing requirements, reasonable amounts of extra hardware are employed.

Figure 10 shows the closely interactive relationships among the tuning box, hardware and software, and also presents the items which move between hardware and software.

5.2. Glue Logic Tuning

We choose DLX [14] as our base architecture. Figure 11 shows the modified DLX processor diagram. The paths for control are in dotted lines and the paths for data transfer are in solid lines.

Glue logic means the connection circuits among all the major hardware modules. The simulation results show that pipeline hazards slowed down the speed. So, we add a glue logic, the bypassing unit, to improve the performance (refer to Figure 11). Besides, we recompiled the programs using the new version of compiler, and reanalyzed programs to tune the local processor and re-measure frequencies of instruction patterns for the peephole optimization [7]. To eliminate the pipeline stall, we developed new instruction scheduling schemes [8] for resolving the issue.

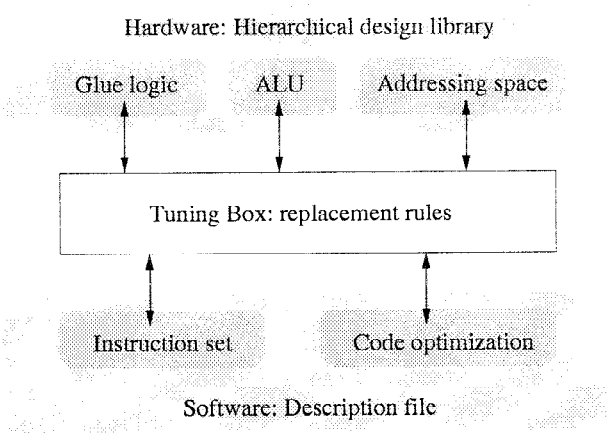

Figure 10. The bridge between hardware and software.

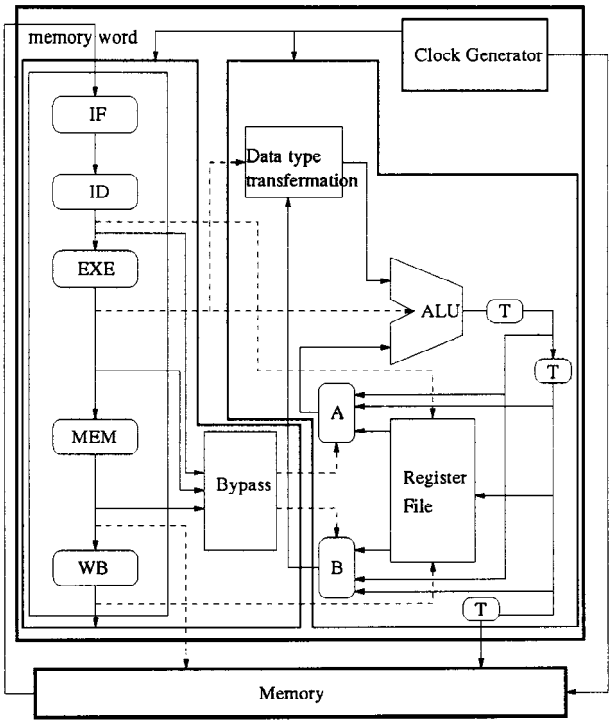

Figure 11. The block diagram for the modified DLX.

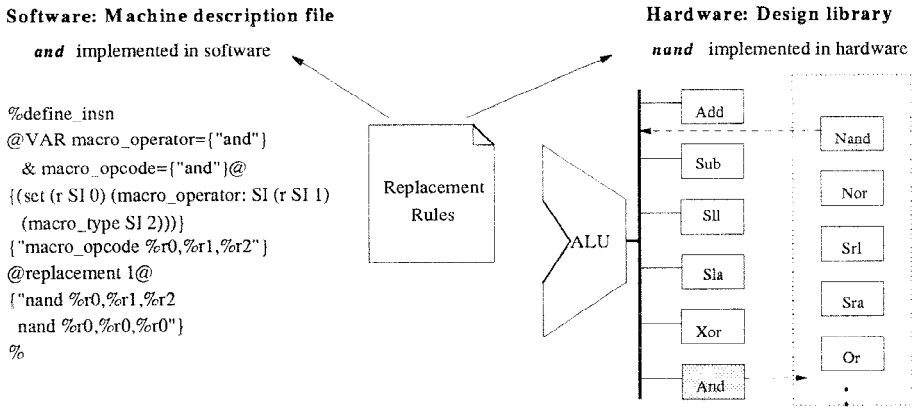

Figure 12. The replacement between hardware/software for logical operators, *and*, *nand*.

5.3. Instruction Set Tuning

The instruction set architecture plays an important role [9] in computer architecture, it can be treated as the communicating interface between hardware and software. So the primary part in tuning box is instruction set replacement rules.

For example, the logical operators contain *xor*, *and*, *nand*, *or*, *nor*, and so on. These operations are all implemented both in hardware and software, that is, if one operation maps a mnemonic code in the instruction set, there is a corresponding hardware component which is stored in design library to perform it, or this operation will be performed by the combination of other operations which modify the machine description file, as illustrated in Figure 12. This device allows us to choose the most appropriate processor platform with a minimal effort.

5.4. Addressing Space Tuning

One of the multimedia applications, file compression, contains a high proportional "High Load" instruction. This special-purpose data moving operation, called LHI in DLX or SETHI in SPARC, co-works with general move operation to accomplish a long-datum load. It appears almost all in loading "label address" to register, because the address needs a full 32-bit word to represent. For example,

```
lhi r4,(LC0>>16)&0xffff
addui r4,r4,(LC0&0xffff)
```

Figure 13 shows the pie chart of execution cycles of a typical compression program SPECint92's Compress.c. If we reduce the two-instruction combination to one, then we

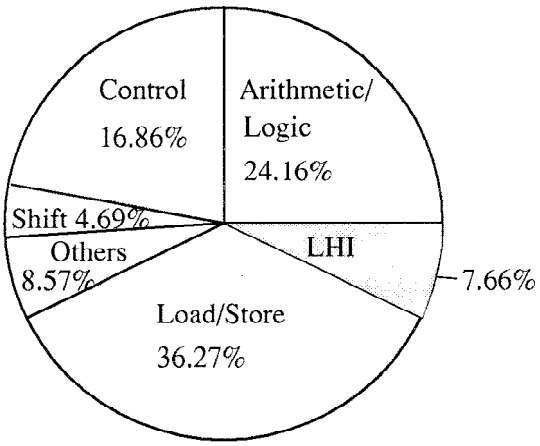

Figure 13. The pie chart of Compress's execution cycle time.

can save nearly 8% execution time. All we have to do is to sacrifice half addressing-space for it. That is, we force the leftmost significant bit of instruction to 1 and call it LHI. As a result, the addressing-space will decrease to 2^{31} but be able to load the address one time. Of course, if the opcode originally has 14 bit to represent, now it loses one bit. These are tradeoffs between speed and space.

6. Case Study

6.1. A Graph Example

Let's focus on a typical multimedia application: line-drawing algorithm. Bresenham [15] developed a more efficient one to determine pixel positions using only integer arithmetic. The program codes are very suitable to execute in subword versions of arithmetic operations such as subword add and shift-add [5].

Thomas M. Conte et al. [5] propose some features that generally characterize multimedia applications:

- Small native data types (8 or 16 bits)
- Large data set sizes
- Computationally intensive features, but with highly predictable branches
- Large amounts of inherent data parallelism

We illustrate these attributes one by one through Bresenham's line-drawing example (refer to Figure 14). Firstly, the total twelve parameters are all in 16-bit **int** type and satisfy the

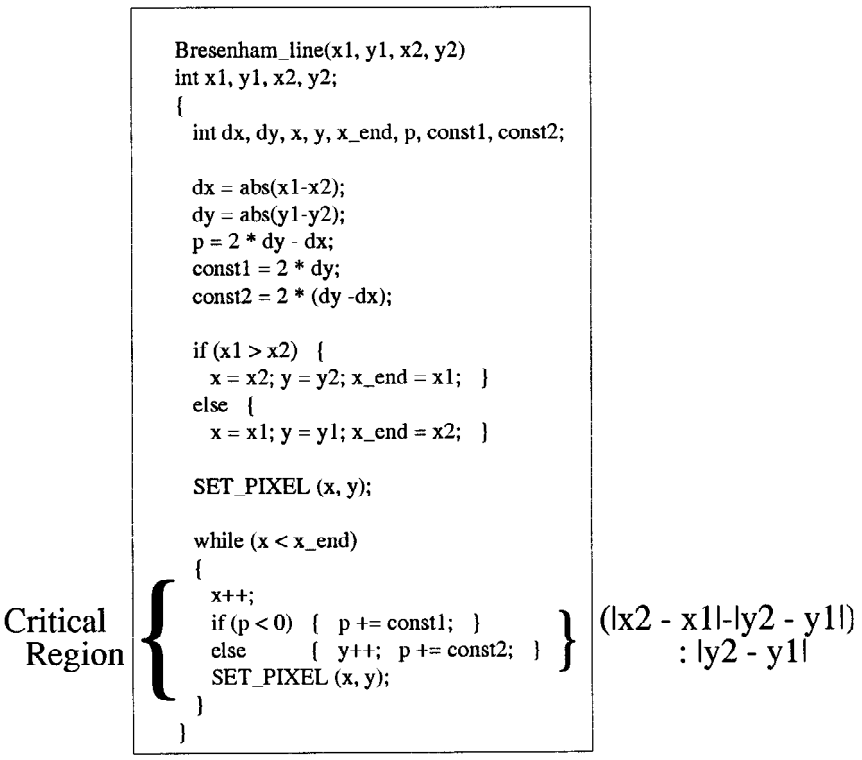

Figure 14. Bresenham's line-drawing algorithm in C.

first two attributes. Worthy to be mentioned, the Huffman code used for file compression [16] seems more suitable to show this because of its 26-character-code tree. Next, in the only **while**-loop, the critical region, we even can predict the branching probability (|x2 - x1| - |y2 - y1|) : |y2 - y1| accurately. Thus, we concentrate on the **else**-block optimization:

- The C source codes

```
y++;
p+=const2;
```

- The corresponding DLX assembly codes

```
lw      r3,68(r30)      ;; load y word
add     r3,r3,#1        ;; y++
sw      -68(r30),r3     ;; store y word
lw      r3,-84(r30)     ;; load p word
lw      r4,-100(r30)    ;; load const2 word
```

```
add     r3,r3,r4          ;; p+=const2
sw      -84(r30),r3       ;; store p word
```

But |P| and |const2| are far smaller than 2^{15}(32,768) if |x|, |y| are not determined in resolution as high as 2^{14}(16,384). This shows the above fourth attribute.

- The code reduction from inherent data parallelism

```
ld      r3,-68(r30),-84(r30)      ;; load y, p subword
ld      r4,#1,-100(r30)           ;; load 1,const2 subword
add     r3,r3,r4                  ;; y++, p+=const2
st      r3,-68(r30),-84(r30)      ;; store y,p subword
```

Assume that the **while**-block in Figure 14 loops N times, the transformation result is optimized in 2N times. Exploiting Intel's MMX [17] methodology, we combine these new multimedia instructions into floating-point module so modify DLX into MMX version.

Similar situation appears in other application. For instance, calculation of the coefficients for the Newton formula [18], whose critical region is a double-loop matrix processing. The region iterates 444 times and contains twelve additions that could be reduced to six. Thus it totally can reduce 2664 cycles.

6.2. Experimental Results

The benchmark set we choose is multimedia orientation, except Ackermann.

- Huffman [16]: Construction and computation of the Huffman code. The input sequence is "A SIMPLE STRING TO BE ENCODED USING A MINIMAL NUMBER OF BITS."

- Newton [18]: Calculation of the coefficients for the Newton formula. The input is $f(x) = (1 + x^2)^{-1}$.

- Bresenham_line and _circle [15]: Two improved quick drawing algorithms. The parameters are all under 100.

The result shows that Newton and Ackermann is great optimized by PO (Peephole Optimization) option. As for RR (Replacement Rules) option, Newton and Bresenham_line benefit much. Table 1 and Figure 15 indicate the amount of reduced dynamic instruction execution from information analyzer and instruction optimization. The values denote the total number of instructions executed. The performance improvement is shown at Table 2 and Figure 16 for our ASIP codesign environment from system specification to product. The speedup is calculated based on the execution cycles of the cosynthesis software (benchmarks) and hardware architecture (ASIP) on CAD tool, SYNOPSYS.

Table 1. Percentage of reduced number of executed instructions.

Program	*Original*	*with PO*	*Ratio$_{PO}$(%)*	*with PO+RR*	*Ratio$_{PO+RR}$(%)*
Huffman	200292	192056	95.9	187028	93.4
Newton	33768	12705	37.6	9834	29.1
Ackermann	42818	20754	48.4	20177	47.1
Bresenham_line	54083	53052	98.1	51598	95.4
Bresenham_circle	239132	237275	99.2	232137	97.1

PO: Peephole Optimization; RR: Replacement Rules

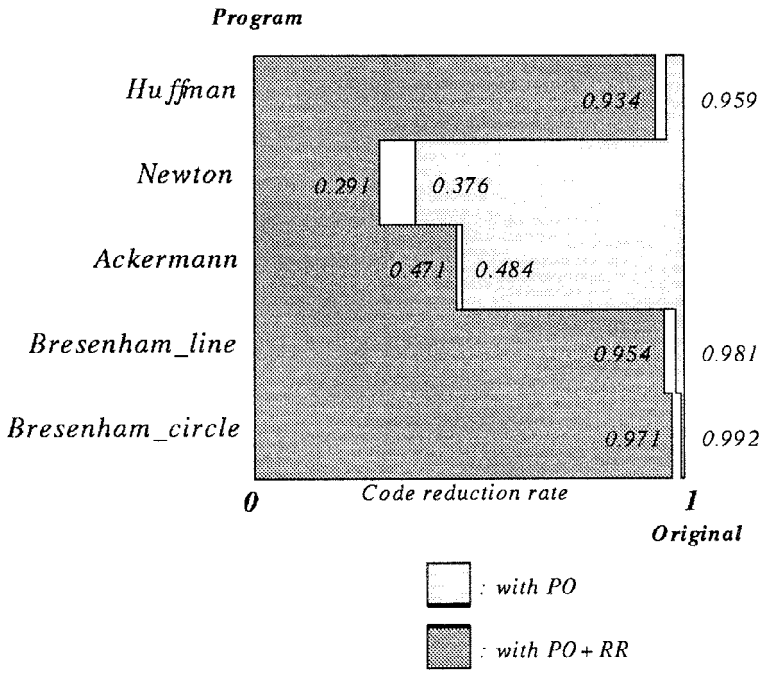

Figure 15. Percentage diagram of reduced number of executed instructions.

7. Conclusions and Future Work

A Fine-tuned Architecture Codesign Environment (FACE) has been developed. FACE allows us to implement an ASIP and make critical decisions early in the design processes.

We presented the environment which employs the design procedure to integrate a compiler

Table 2. Speedups for our design environment.

Program	*Original*	*with PO*	*Speedup$_{PO}$*	*with PO+RR*	*Speedup$_{PO+RR}$*
Huffman	314794	302410	1.04	296890	1.06
Newton	56154	20127	2.79	17256	3.25
Ackermann	68338	32934	2.08	32357	2.11
Bresenham_line	82179	79812	1.03	78317	1.05
Bresenham_circle	361318	356866	1.01	352532	1.03

PO: Peephole Optimization; RR: Replacement Rules

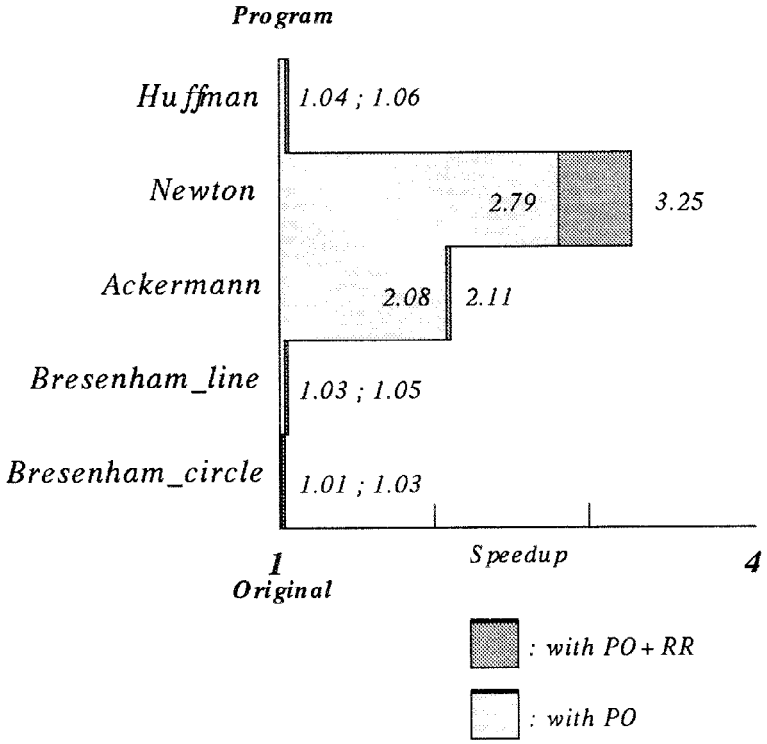

Figure 16. Speedups diagram for our design environment.

generator and a hardware design library to reduce the complexity. The accomplished features and benefits of the study is described in the following:

- Higher-level specification and lower-level simulation
  We adopt a mature CAD tool, SYNOPSYS, to be the experimental platform. And

exploit lots of application-oriented attributes to be objective function base. These two highest and lowest end-points make the whole environment more efficient and more reliable.

- Three-step decision-making mechanism
  Requiring, analyzing, and partitioning is our proposed strategy. Requirement stage takes a macro-view of special-purpose application behavior. Analysis stage then goes through the program to analyze the instruction patterns in a micro-view. Finally, by applying replacement rules, partition stage can make a best tradeoff.

- Object-oriented tuning box
  The mechanism does an actual replacement job and provides a user-interface to guide designers in making decisions. By the object-oriented instruction description model, we can make the migration between hardware/software quick and change the instruction set easily.

The empirical results show that applying both our environment and design procedure on system level design can generate better results with respect to the proposed metric. The future work should be improving the whole codesign environment, involving operating system and CAD tools, for larger benchmarks.

References

1. J. Sato et al. PEAS-I: A hardware/software codesign system for ASIP development. *IEICE Trans. Fundamentals of Electronics, Communications and Computer Sciences* E77-A(3): 483–491, Mar. 1994.
2. R. M. Stallman. Using and porting GNU CC. *Free Software Foundation*, Version 2.7, 1996.
3. I. J. Huang et al. Synthesis of application specific instruction sets. *IEEE Trans. Computer-Aided Design* 14(6): 663–675, June 1995.
4. W. Stephen. *VHDL Analysis and Modeling of Digital Systems*. McGRAW-HILL, 1993.
5. T. M. Conte et al. Challenges to combining general-purpose and multimedia processors. *IEEE Computer* 33–37, Dec. 1997.
6. F. Lai, S. L. Hwang, and T. S. Chen. ARDEN - ARchitecture Development ENvironment. *IEEE TENCON'93*, Oct. 1993.
7. T. S. Chen, F. Lai et al. Peephole optimizer in retargetable compilers. *IEICE Trans. Information and Systems* E79-D(9): 1248–1256, Sept. 1996.
8. F. Lai and Y. K. Chao. The complementary relationship of interprocedure register allocation and inlining. *International Conference on Computer Languages*, Toulouse, France, pp. 253–264, May 1994.
9. D. Morris et al. *Object Oriented Computer Systems Engineering*. Springer-Verlag, 1996.
10. S. Kumar et al., A framework for hardware/software codesign. University of Virginia, Technical Report No. 920525.0, 1992.
11. P. Darche et al. ActNet: the actor model applied to mobile robitic environments. *Object-Based Parallel and Distributed Computation, selected papers of OBPDC'95, LNCS N. DG 1107*, Springer-Verlag, 273–289, 1996.
12. S. Kumar et al. Object-oriented techniques in hardware design. *IEEE Computer* 64–70, June 1994.
13. N. N. Binh et al. A hardware/software partitioning algorithm for pipelined instruction set processor. *Proc. of Euro-DAC'95* 176–181, 1995.
14. D. A. Patterson and J. L. Hennessy. *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann Publishers, 1996.
15. D. Hearn and M. P. Baker. *Computer Graphics*. Prentice-Hall, 1986.
16. R. Sedgewick. *Algorithms in C*. Addison-Wesley, 1990.

17. D. Bistry et al. *The Complete Guide to MMX Technology*. McGraw-Hill, 1997.
18. S. D. Conte and Carl de Boor. *Elementary Numerical Analysis: An Algorithmic Approach*. McGraw-Hill, 1980.