RCG: Retargetable Code Generation Methodology for Embedded Processors

Yang-Hsin Fan, Trong-Yen Lee, Tsung-Hsun Yang

Journal of Convergence Information Technology, Volume 6, Number 2. February 2011

Abstract

Embedded systems tightly serve the needs of consumer electronics, home automation and portable devices. During the period of developing embedded systems, code generation is always seen a design challenge because every embedded processor has its own specific characteristic. In this paper, we propose a retargetable code generation (RCG) methodology which can generate retargetable C program for embedded processors. In addition, the generated code is further verified for time constraint with the global time, local time and kernel time. Finally, two design examples namely normal traffic light controller (NTLC) and four phases stepping motor control (FPSMC) are used to illustrate the feasibility in the embedded platform.

Keywords: Retargetable Code Generation, Embedded Software, Processor Synthesis

1. Introduction

Microcontrollers widespread apply to develop embedded system and manufactured by many companies such as Atmel, Philips, Texas Instruments, etc. In recent years, the requirements of embedded systems are rapidly increasing in such application as automobiles, avionics and portable devices. When develop various embedded system applications, every processor may be selected as a component of system architecture. The diverse characteristics such as interrupt modes, timer activities and peripheral of processors result that embedded software has to re-design when the target processor has been changed. Therefore, obviously, the re-design causes that design cost and time consuming of embedded software must increase greatly accompanying with the change.

Embedded software such as C language and assembly language are frequently used to develop embedded system while the processor is decided. Considering program’s flexibility, efficiency and productivity, C language is better empirically than assembly language. That is to say, designers could rapidly develop embedded software as C language that is independent from processors. Such software is known as American national standard institute (ANSI) C code which has limited functionalities because it is unable to control interrupt modes, timer activities and peripheral. Consequently, code for executing interrupt modes, timer activities and input/output peripheral in various embedded processors has to spend more time and cost on modifying diversity code for the listed controlled interfaces before the software downloads to specific embedded processors such as 8051/52 or peripheral interface controller (PIC). Owing to ANSI C code is unable to control processors corresponding to interrupt modes, timer activities and input/output peripheral. Code generation technique for interrupt modes, timer activities and input/output peripheral rapidly develop embedded systems becoming a significant issue. Importantly, it can not only save the cost of design and time but also fast generate the retargetable code and accurate code with efficient and reliable approach.

To solve code generation for interrupt modes, timer activities and input/output peripheral of embedded system, we propose a retargetable code generation (RCG) methodology for embedded processors which is capable for automatic generating appropriate embedded C code. Moreover, RCG is designed for evaluating functional correction and meeting time constraint. The RCG provides contributions to code generation issue are sum up as follows. First, we present retargetable code generation tree (RCGT) as interface to translate the ANSI C into embedded C code for retargetable processor. Second, we develop a retargetable code generation algorithm (RCGA) for generating
embedded software which accompanies interrupt, time or input/output functionalities. Third, we evaluate the time constraints of producing code by global time, local time and kernel time. Finally, the effectiveness of RCG is exhibited by assessing normal traffic light controller (NTLC) and four phases stepping motor control (FPSMC) design examples on two embedded platforms.

2. Preliminaries

Due to that the code generation for embedded system is a crucial issue, several techniques [1]-[14] have been proposed by many researchers. In 2000, for instance, Leupers [1] successfully solves the optimization of embedded processor by using assembly language. However, it is incapable of solving retargetable code generation. Regarding synthesis tools for hardware-software codesign (HSC), Parkinson et al. [2] develop a tool to translate sections of C code into behavioral VHDL. Similarly, Rettberg et al. [3] propose a useful flowchart based on the state flow models with multiple inputs and a single output for code generation. Additionally, in 2002, Chung et al. [4] not only create a framework for the optimization of embedded software but also present algorithms and a tool flow to reduce the computational effort of programs. To reduce the matching time, Chen et al. [5] use a simple tree pattern matching algorithm in the code generator to reduce it to 69% effectively. In 2002, Leupers [6] provides a survey of methods and techniques which dedicates efficient code generation for embedded processors. Surprisingly, Lee et al. [7] introduce the interrupt time petri nets (ITPN) model and the interrupt-based quasi-dynamic scheduling (IQDS) algorithm to model embedded systems with interrupt property and find task schedules with system constraints, respectively. However, this work only focuses on 8051 processor code generation.

In terms of the verification of time constraints, Fu et al. [8] propose a formal coverification approach based on linear hybrid automata and an algorithm result to the linear hybrid automata framework. In 2002, Hsiung et al. [9] present a time-extended quasi-static scheduling algorithm (TEQSS) to synthesize real-time embedded software code through a set of time complex-choice petri nets. TEQSS mainly focuses on meeting memory and time constraints but it does not address retargetable code generation for embedded systems. Following this, in 2004, Lee and Hsiung [10] propose an embedded software synthesis and prototyping (ESSP) methodology to solve the software synthesis, software verification, code generation, and system emulation.

In the event that automatic code generation is concerned, Manohar and Bhatia [11] indicate a tool for automatic code generation in designing user interfaces on character terminals and a user interface tool for designer to solve problems on complex library call. However, the tool lacks of facilitation which leads to its non-support mouse events. Charot and Messé [12] suggest a flexible code generation framework for the design of application specific programmable processors and use library modules to achieve flexible compilation passes. The framework consists of two levels, one is re-targeting code generation that define compilation flow, and another allows user to select and link modules from the library for building a compiler. Unfortunately, their library has not been completed by the time of publishing the paper.

On the other hand, some researchers dedicate embedded software from perspective of the system. Since 2003, systematic embedded software based on system C redefinition and overloading has been proposed. Hence, in [13], Herrera et al. use the same system C code in system-level specification for the embedded software generation. Moreover, writing the corresponding library is necessary regarding of independent RTOS issue. In 2005, Kwon et al. [14] propose embedded software generation from system level specification to multi-tasking embedded systems. Both virtual OS wrapper and APIs are applied as layer structure in generated software scheme. In 2010, Aoshima and Kanasugi [15] adopt redundant binary number in genetic algorithm for processor that can find the optimized solution and decrease error rates. In [16], Dossis describes an approach to automatically produce non-standard, custom and directly implementable hardware description code for intelligent web service. In [17], Dossis proposes intelligent, knowledge-based system (IKBS) that can automatically design framework for verified and correct hardware accelerators. It contributes not only provably-
correct, high-level hardware synthesis method but also a unified prototype tool-chain based on compiler-compiler and IKBS techniques.

3. Retargetable code generation

Most of the embedded code highly depends on system architecture that consists of specific processor, memory allocation and input/output interfaces, etc. Such feature causes time consuming on re-design while target processor is changed. Consequently, we propose a retargetable code generation (RCG) methodology for embedded processors to solve the mentioned problem that can produce satisfying user-given functional requirement code. The details of RCG are introduced as follows. First, we demonstrate the design flow of RCG in the beginning. Next, on the basis of binary tree, the presented translation method with retargetable code generation tree (RCGT) aims to translate ANSI C code into retargetable embedded code. Third, we describe a retargetable code generation algorithm (RCGA) which targeted at generating retargetable code for various processors. Then, an evaluation discussion of generated embedded code determination by global time, local time and kernel time is displayed. In the final part, the development of a graphical user interface (GUI) improves the code generation interface is consulted. Table 1 shows the comparison of RCG to other techniques that dedicate to code generation issue.

Table 1. Comparison RCG to other techniques for code generation

<table>
<thead>
<tr>
<th>Methods</th>
<th>[1][3][6][12][13][14]</th>
<th>[2][11]</th>
<th>[4][5][7]</th>
<th>[8][9]</th>
<th>[10]</th>
<th>RCG (Our)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Code generation</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Algorithm</td>
<td>–</td>
<td>–</td>
<td>Yes</td>
<td>–</td>
<td>–</td>
<td>Yes</td>
</tr>
<tr>
<td>Evaluation</td>
<td></td>
<td>–</td>
<td>–</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Graphical user interface</td>
<td></td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>Yes</td>
</tr>
<tr>
<td>Retargetable code generation</td>
<td>–</td>
<td>Yes</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>Yes</td>
</tr>
</tbody>
</table>

3.1. RCG design flow

The design flow of RCG is shown in Figure 1 which consists of two design parts which are code generation and code evaluation. Step (1) to step (6) demonstrate code generation method which aims at producing retargetable embedded C code. Then we evaluate the generated code with system functionalities and time constraint from step (7) to step (9). The detail steps are described as follows:

(1). System specification: the system designer provides system specification such as the types of processor, memory sizes, I/O interfaces, interrupt modes and priorities, etc.
(2). Initialization: the designer chooses the type of target processor and setting initial condition in an embedded system. Target processor may be 8051 or PIC. The initial process includes enable/disable interrupt timer/counter control (TCON) register or timer mode control (TMOD) register, etc.
(3). Parsing: parsing source code for syntax, variables, keywords, operator and operand.
(4). Set parameters: we design a GUI for setting interrupt, timer and I/O interface to generate appropriate code for retargetable embedded processors.
(5). Build up retargetable code generation tree (RCGT): after parsing the syntax of source code, we translate the source code into RCGT for code generation. The translation method is discussed in Section 3.2.
(6). Code generation: we translate the source C code into the embedded C code by code generation algorithm which is discussed in Section 3.3. The generated code consists of original ANSI C code and the interrupts modes, timer activities and I/O interfaces with specific processor.
(7). Evaluate the generated code: the generated code with interrupts modes, timer activities and input/output interfaces will be evaluated. Also, three kinds of time variable, local time, global time and kernel time, will be evaluated at the same time. The details of evaluation are shown in Section 3.4.
(8). Code simulation: code generation algorithm produces the code in RCG which will be tested by Keil C compiler [18] or Hi-Tech C compiler [19]. They refer to 8051-based or PIC-based embedded systems, respectively.

(9). Verification: we use two different kinds of emulation platform, 8051 and PIC16F of WINICE [20], to verify the two generated code.

![Figure 1. RCG design flow](image)

3.2. Translation method

We propose a translation method through RCGT to transfer source C code into embedded C code. The RCGT is based on binary tree which can be used to represent hierarchical and structured data. The time complexity of RCGT is $O(\log N)$ where $N$ is the number of nodes in tree. As below, definition 1 defines RCGT which is illustrated in Figure 2.

**Definition 1.** RCGT consists of root $R$, left subtree $P$ and right subtree $S$. Left subtree $P$ is a set of characteristic of processor. Right subtree $S$ is a set of ANSI C code. $P$ consists of interrupt $B$ and input/output port $G$. $B$ is comprised of timer interrupts $T_i$ and trigger interrupts $T_r$. A set of external interrupts and serial port interrupt forms $T_r$. $G$ consists of Input $I_p$ and output $O_p$ which represent various input and output interfaces. $S$ includes a set of functions $F$ and variant tokens $T$ which comprises a set of variable $V$ and keyword $K$ in ANSI C.

- $R$ node: a root node without any parent nodes in a tree, such as node $R$ in Figure 2.
- $P$ node: a process node which is used for specific kinds of processor such as 8051, PIC or others.
- $S$ node: a subtree node consists of a set of statements that independent of processor.
- $B$ node: a node represents various interrupt for processor which depends of specific processors.
- $G$ node: a node consists of various input/output ports for processor.
- $F$ node: a node represents a set of functions such as delay, interrupt or others.
- $T$ node: a node consists of variables, keyword and other such as for, variant variables and comment, respectively.
RCG: Retargetable Code Generation Methodology for Embedded Processors
Yang-Hsin Fan, Trong-Yen Lee, Tsung-Hsun Yang
Journal of Convergence Information Technology, Volume 6, Number 2. February 2011

- $T$: node: a node consists of various timers interrupt for processor.
- $I$: node: a node consists of various external interrupt and serial port interrupt for processor.
- $O$: node: a set of various input ports.
- $V$: node: a set of variables.
- $K$: node: a set of keywords.
- $E$: node: a set of external interrupts.
- $S$: node: a set of serial port interrupts.
- Level: the total of levels from the level of root node to the level of leaf node level. For example, R node is at Level 1, and K node is at Level 4 in Figure 2.

Figure 2. An example of RCGT

Table 2 defines the structure of Figure 2 among nodes. From the top-down viewpoint, root node comprises processor type nodes and subtree nodes. On one hand, processor_type is either 8051 or PIC depending on target system. On the other hand, subtree independent of target system that consists of a set of functions and tokens. $P$ node consists of various interrupts and input/output port. For example, the type of interrupts in 8051 is divided into timer, external and serial port interrupt. The timer interrupt can be triggered by TF0 and TF1. The external and serial port interrupt can be triggered by IE0, IE1, low_level or negative edge in 8051 processor. Regarding of input/output port represents via $G$ that has four available input/output ports, P0, P1, P2 and P3, in 8051 processor. The token denotes keyword, variable and other. Reserve words such as for, while, if in ANSI C are defined as keyword token. Non-reserve words such as letter, digit, '_' or combining_char are classified as variables token. Any character from 'a' to 'z' or 'A' to 'Z' is defined as letter. The numeral, '0' to '9' are symbolized as digit. The combining_char generally consists of arbitrary letter, digit and '_'. Besides keyword and variable tokens, we define head_file if standard I/O header file or other files is included.

Table 2. Definition nodes of RCGT

```plaintext
root ::= { processor_type, subtree }
processor_type ::= { target_8051 | target_PIC }
subtree ::= { function, token }
target_processor ::= {Interrupt, Gate}
Interrupt ::= { Timer, Trigger }
Gate ::= { input, output }
functions ::= { delay, function, other }
token ::= { keyword, variable, other }
Trigger ::= { External, Serial port }
keyword ::= { for | while | if | else | case | switch | do }
variable ::= { letter | digit | '_' | combining_char }
letter ::= { 'a' ~ 'z' | 'A' ~ 'Z' }
digit ::= { '0' ~ '9' }
combining_char ::= { 'a' ~ 'z' | 'A' ~ 'Z' & '0' ~ '9' & '_' }
other ::= { comment, head_file }
comment ::= { '//' | '/*' & '*/' }
head_file ::= { 'include' }
```
RCGT is an effective interface that can be used to translate ANSI C code into retargetable embedded code. When ANSI C code has loaded, it will be constructed in the right subtree of RCGT where labels S in Figure 2. Hierarchical structure of RCGT results in traversing each node become more easily than non-hierarchical structure. That is, the right subtree can be restored easily to ANSI C code by recursive traverse tree technique. Similarly, the left subtree of RCGT can be constructed when target processor is determined. Once RCGT has been constructed, we generate target embedded code by code generation algorithm that is introduced in next section. Figure 3 illustrates a subtree of $W = X \times (Y + Z)$ statement in the node $V$ of level 4 and a delay function code how arranges in the node $T_i$.

The advantages of RCGT include hierarchical structure, better time complexity as $O(\log N)$ and the ability of generating retargetable code. Hierarchical structure forms an excellent platform that is helpful for generating retargetable code. Regarding to the ability of RCGT, processors with interrupt, timer or input/output control can be easily constructed to left subtree then retargetable code generation algorithm can use it to produce embedded code.

Figure 3. RCGT for a statement and delay function

3.3. Retargetable code generation algorithm (RCGA)

Hundreds of thousands embedded applications require various embedded code to drive interrupt, timer control or peripheral output. When developing embedded application, the embedded code is various because target processor may be 8051-based, PIC-based or others. In most embedded systems, processors are always designed for handling interrupt, timer and peripheral input/output. Here, we propose a retargetable code generation algorithm (RCGA) to automatically produce embedded C code with interrupt, timer or input/output interface. Moreover, we develop a graphical user interface (GUI) for setting various parameters. The algorithm of retargetable code generation is shown in Table 3. The steps of RCGA are described as follows:

Step (1), open the source file and assign a name to output file.
Step (2), set the initial condition of embedded system such as interrupt, timer enable/disable, and input/output initial value, etc.
Step (3), set the required parameters of embedded processor such as interrupt, timer or input/output port.
Step (4), we parse the syntax of source C code, and split the token into keyword, variable, comment, and head file.
Step (5), after getting the token from source ANSI C code, the next step is to find statements related to interrupt, timer and peripheral input/output.
Step (6), construct statements to RCGT such as Figure 3.
Step (7), generate embedded code with interrupt, timer and input/output ports according to RCGT.

Table 3. Retargetable code generation algorithm (RCGA)

<table>
<thead>
<tr>
<th>Procedure RCGA(P, F) /* the whole function of the RCGA include such elements as below: */</th>
</tr>
</thead>
<tbody>
<tr>
<td>P = {A_i</td>
</tr>
<tr>
<td>F is the file Stream</td>
</tr>
<tr>
<td>T_i is the finite set of token in F</td>
</tr>
<tr>
<td>R_i is the subtree constructed by fetching the keyword)</td>
</tr>
<tr>
<td>Begin</td>
</tr>
<tr>
<td>Open_file(F); (1)</td>
</tr>
<tr>
<td>Initialization(F); (2)</td>
</tr>
<tr>
<td>Parameter_Setting(F); (3)</td>
</tr>
<tr>
<td>//Set interrupt vector, timer, I/O port type, etc.</td>
</tr>
<tr>
<td>While(do not end of file)</td>
</tr>
<tr>
<td>GetToken(F, T_i); (4)</td>
</tr>
<tr>
<td>//get the variable into T_i from the file F</td>
</tr>
<tr>
<td>Parsing (T_i); (5)</td>
</tr>
<tr>
<td>//map the variable into I/O port</td>
</tr>
<tr>
<td>Transfer_code_to_RCGT(T_i, R_i); (6)</td>
</tr>
<tr>
<td>//construct the statements to the RCGT</td>
</tr>
<tr>
<td>End // end of While</td>
</tr>
<tr>
<td>Retargetable_Code_Generation(R_i); (7)</td>
</tr>
<tr>
<td>End //end of Begin</td>
</tr>
<tr>
<td>End //end of Procedure</td>
</tr>
</tbody>
</table>

3.4. Evaluation

The time constraint is another important factor which needs to take into account corresponding to produce embedded software. Generally, embedded systems are designed for satisfying hard time constraint which defines a time for executing all tasks. For instance, if the system time constraint is limited to 100ns in an embedded system, it means the total execution time must be under 100ns in order to avoid malfunction in the rest of process. Hence, evaluating embedded code for meeting time constraint is essential with respect to code generation. Equation (1) defines time constraints which consist of global time, local time and kernel time. The details of each time constraints are discussed subsequently.

\[ k + \sum_{i=0}^{n} L_i \leq S, \text{ where } i = 0, 1, 2, \ldots, n. \]  \hspace{1cm} (1)

(1) Global time: A system time, S, represents maximum execution time that jobs must be finished with given time even though interrupt or timer is triggered. For example, if S and interrupt subroutine denotes individually 500ms, 70ms, it indicates that only two interrupts are allowed to insert into the job in case of system execute for 350ms. If the third interrupt intends to trigger system from any process, the system will reject it unless the system finishes the task of whole system. In order to help designer rapidly evaluate the global time, designers can set the maximum system execution time to check the violation of execution time by proposed GUI tool.

(2) Local time: we provide the functionality to set the maximum execution time of each subroutine or interrupt function within proposed GUI. Such time is called local time variable Li. After code generation, local time variable will be verified whether the execution time is longer than global time S or not. We set the maximum execution time as local time while two or more subroutine or interrupt function is executed simultaneously.

(3) Kernel time: we denote the time of main program as kernel time K.

(4) Functional simulation and verification: we verify the time constraints of the system by global time, local time and kernel time according to Equation (1). For functional testing, we will simulate the
embedded C code in the compiler environment and download the execution code into emulation board to verify the function of designing system.

4. Design examples

In this section, we use two embedded system design examples, normal traffic light controller (NTLC) and four phases stepping motor control (FPSMC), to illustrate the feasibility of our proposed RCG. It can generate retargetable embedded C code for NTLC and FPSMC with interrupt and timer functionality. Two compilers namely Keil C compiler and PIC C compiler are used to produce executable code which can emulate and test embedded software functionality in 8051 and PIC platform. Table 4 displays the specification of emulation board. In this experiment, we evaluate the generated code by timer, interrupt and input/output port functionality.

The first design example is a NTLC which applies an intersection of roads $R_a$ and $R_b$. Normally, traffic heavy and emergency states are three states in NTLC system. Normal state means $R_a$ and $R_b$ has the same running time of traffic light in this system. Traffic heavy state is used for longer running time of traffic light in green light in $R_a$ than $R_b$ because $R_a$ traffic is busier than $R_b$. The last, the emergency state is especially suitable for traffic accident. According to above specifications, timer 0, interrupt 0 and 1 in RCG will be used in order to meet functionality for the three states respectively. As a result, system designer firstly edit an ANSI C code as main program which is shown in Figure 4. Next, designer uses our graphical user interface based on RCGA to set parameter such as initialization, interrupt, timer mode and input/output value for system constraints. Figure 5 illustrates the relationship between RCGT and partial generated code for 8051-based embedded system. It also demonstrates the RCGT that is rotated to left for 90 degree. Figure 6 shows the partial generated code of PIC-based embedded system.

Table 4. The specification of emulation board

<table>
<thead>
<tr>
<th>CPU</th>
<th>8051/52</th>
<th>16F877</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>System clock</td>
<td>16MHz</td>
</tr>
<tr>
<td></td>
<td>Program memory</td>
<td>64KBytes</td>
</tr>
<tr>
<td></td>
<td>Data memory</td>
<td>64KBytes RAM</td>
</tr>
<tr>
<td></td>
<td></td>
<td>256Bytes EEPROM</td>
</tr>
</tbody>
</table>

```c
#include <stdio.h>
#include <stdlib.h>
void delay(int);
main()
{
    int i, j=2, temp1, temp2;
    char table[3]={'0x01','0x02','0x04'};
    while (1)
    {
        for(i=0;i<=2;i++ )
        {
            temp1=table[i];
            temp2=table[2-i];
            delay (10000);
        }
        j--;
    }
    void delay(int count)
    {
        int i,j;
        for(i=0;i<count;i++)
            delay(j=1500;j++)
        } //select which side can be allowed
```

Figure 4. ANSI C code for NTLC
Four phases stepping motor control (FPSMC) is the second design example in our experiment. Industry application in FPSMC is usually used to control position system such as printer and disk. Two main functions in FPSMC are motor direction and running time. Motor direction includes counterclockwise and clockwise. According to FPSMC functionality, both interrupt and timer modes will be used to control motor direction and running time, respectively. For example, either counterclockwise or clockwise is controlled by interrupt mode. Figure 7(a) shows ANSI C code for FPSMC. Figure 7(b) and Figure 7(c) display partial embedded C of FPSMC for 8051-based and PIC-based embedded systems which are automatically generated through RCG.

Two steps for compilation and emulation in this experiment are used for verifying the feasibility of RCG to apply in embedded systems. Firstly, Keil C and Hi-Tech C compile the output embedded C code from RCG to generate execution code for 8051-based processors and PIC-based processors, respectively. Secondly, if the results are correct, the next step for emulation will be verified by WINICE board of 8051 and PIC-16/17 embedded system.
The experiment results show that RCG can generate embedded C codes then check the correction through software compiler verification or emulation board such as 8051 and PIC of WINICE. In terms of external circuit examples, we not only download the generation code of RCG into retargetable embedded systems but also verify circuit functionality. Besides, both code size and execution time are also compared between ANSI C and embedded C code. As shown in the second to fourth column of Table 5, code size of embedded system are obviously larger than ANSI C due to the reason that interrupt, timer or serial control code are added by RCG. Moreover, we calculate execution time as execution period times system clock that either PIC is 0.5×10⁻⁶ or 8051 is 0.2×10⁻⁶ second. Table 6 indicates that embedded C is slower than ANSI C code which is caused by interrupt, timer or serial control. Also, they are produced by RCG.

### Table 5. Code size for two design examples

<table>
<thead>
<tr>
<th>Examples</th>
<th>Source C code</th>
<th>Embedded C code for 8051</th>
<th>Embedded C code for PIC</th>
</tr>
</thead>
<tbody>
<tr>
<td>NTLTC</td>
<td>618 Bytes</td>
<td>3714 Bytes</td>
<td>3928 Bytes</td>
</tr>
<tr>
<td>FPSMC</td>
<td>346 Bytes</td>
<td>2860 Bytes</td>
<td>3108 Bytes</td>
</tr>
</tbody>
</table>

### Table 6. Execution time for two design examples

<table>
<thead>
<tr>
<th>Examples</th>
<th>Source C code</th>
<th>Embedded C code for 8051</th>
<th>Embedded C code for PIC</th>
</tr>
</thead>
<tbody>
<tr>
<td>NTLTC</td>
<td>1321×10⁻⁶</td>
<td>1708×10⁻⁶</td>
<td>1899×10⁻⁶</td>
</tr>
<tr>
<td>FPSMC</td>
<td>2124×10⁻⁶</td>
<td>2848×10⁻⁶</td>
<td>2969×10⁻⁶</td>
</tr>
</tbody>
</table>

### 5. Conclusion

We proposed a retargetable code generation methodology, namely RCG, to solve embedded code generation for specific processors. To achieve RCG, both retargetable code generation tree and
algorithm have presented. By RCG, embedded code with interrupt modes, timer activities and peripheral input/output can be produced for target processor. For evaluating generated code, not only function correction but also time constraints such as the global time, local time and kernel time variables are demonstrated. Finally, two embedded design examples results show that the RCG successfully generate code for retargetable processors.

6. References