RAPID PROTOTYPING FOR SIMULATION DEBUGGING ENVIRONMENT: AN ENHANCED DEVELOPING METHOD FOR EMBEDDED COMPUTER SOFTWARE

Zhongmin YU and Yoshinao AOKI

Faculty of Engineering
Hokkaido University
Sapporo 060
Japan

ABSTRACT

Embedded computers(EC) have been used widely in the world, however, embedded computer software is often difficult to develop for the lack of suitable debugging environment. With the traditional in-circuit debugging method, it is difficult to analyze the reason of error; also it is impossible to debug a software before the implementation of associated hardware; moreover, it costs both money and time to create the hardware, so that it can hardly keep up with the ever-changing of a great number of ECs. To tackle the problem, we have made research on the prototyping method for simulation debugging environment(SDE) of embedded computer software. Our purpose is to establish an environment to construct SDE and use the software SDE to substitute the traditional in-circuit debugging environment. By the method, users can construct a SDE at a higher speed and lower cost, the development efficiency for embedded computer software will also be increased greatly.

1. INTRODUCTION

Unlike the general computer software, an embedded computer software usually runs in real-time control environment, to debug it, developers must utilize extra special system. The in-circuit debugging environment is the most generally used so far, where add-on hardware must be prepared to support the execution of the embedded computer software. This kind of environment has some shortcomings: first, because it includes both the debugged software and the associated hardware, to analyze whether an error is hardware related or software related becomes difficult; second, users are unable to debug their software before the completion of associated hardware; third, one type of in-circuit hardware can be applicable for only one type of EC. As a great numbers of ECs are used and changed frequently each year, therefore the traditional method can hardly keep up with the ever-changing of the ECs.

Many simulation systems have been created for embedded application system[1]-[4], but they are mainly used for hardware circuit design instead of software debugging. These systems normally aim at special purpose and are often too large, complicated and professional to be established, utilized and maintained.

In this paper, we propose a SDE model at first, then we introduce the prototyping method for SDE. Our objective is to establish SDE quickly and use this software-made SDE to substitute the hardware debugging environment.

2. SOFTWARE SIMULATION DEBUGGING

2.1 SDE model

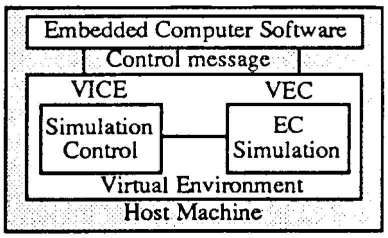


fig.1 Conceptual model for the virtual environment

As shown in fig.1, SDE supports a virtual environment in the host machine, this virtual environment includes a virtual EC(VEC) and a virtual ICE(VICE). In the virtual environment, SDE can provide sufficient functions to support debugging activities in an ordinary computer where no in-circuit hardware required. Users can run the program, watch the state change of the EC or trace the output of an I/O signal. Users may also use copies of SDE to debug different modules or programs at the same time. In addition, 0/1 sequence file can also be used as input signal for execution of a program. All the above functions will enable users to test their application software program for correctness in functionality and timing. Just as in a true hardware debugging environment, even if there is no in-circuit hardware supported, users can also debug their software. This will shorten debugging period and reduce faults before run-time application of the software and finally lead to get a higher qualified computer software.

2.2 Architecture

Generally, we divide SDE into three layers, different layers communicate with each other through inter-layer primitives.

The bottom level is the simulation kernel, its role is to synchronize the simulation activities and execute them on the VEC. Whenever a primitive is called or an instruction is executed, the according primitives of the kernel will work to drive the VEC and change VEC's status.

The middle layer is the simulation shell, it connects the top layer and the bottom layer by supporting functions for simulation debugging control. such as visualization aids, tracing supporting or the transformation of a high level function into several kernel primitives, also many debugging commands are provided in this layer.

The top layer is the GUI part, this layer mainly has to do with the interface between the SDE and user. Several windows are provided for visualization of EC's status.

3. SDE PROTOTYPING ENVIRONMENT

3.1 Objectives

As a result of the diversity and complexity of ECs, we would not obtain a desired SDE as fast as possible if the traditional manual programming method were used, for this reason, we have created a prototyping environment called SPACE(SDE Prototype Automatic Construction Environment), SPACE aims at the following objectives:

1) Rapid prototyping for debugging environment of an embedded software instead of an embedded hardware;

2) Software simulation method will be used to establish a "soft" debugging environment in a heterogeneous machine for replacing the in-circuit hardware debugging environment.

3) Simulating EC on the basis of high level functional behavior but not of the low circuit level.

4) A simple and convenient toolkit instead of a large and complicated system will be created.

3.2 Construction and prototyping steps

SPACE is an integrated environment, as will be introduced in section 4,5 and 6, it mainly contains three parts: 1) The specification part to specify the main features of an EC's CPU; 2) The generation part to translate the specification into C code; 3) The reusable library to provide function for construction of kernel, shell and GUI part of SDE. SPACE provides users with a convenient environment for specifing and prototyping, it utilizes abstract specification, automatic code generation and reusability techniques in the development process.

As shown in fig.2, to create a SDE for an EC, the first step is to analyze the EC on which the embedded computer software will execute, following this step, the user must use the special editor to write the specification for the EC, the specification will be translated into an executable form of specification in C code; if some specialities are difficult to specify, the user must prepare some handmade simulation functions which will be linked with the generated C code later; Following the above steps, the main parts of the simulation kernel will be established, but it still need refining and optimizing. An executable prototype of SDE can be obtained by linking the generated code and reusable function with the refined code; When the users accepted the prototype demonstrated behaviors, they will do further works to make the initial SDE system become a usable one. If the prototype is not so perfect, the user will go back to the former steps for further reversion.

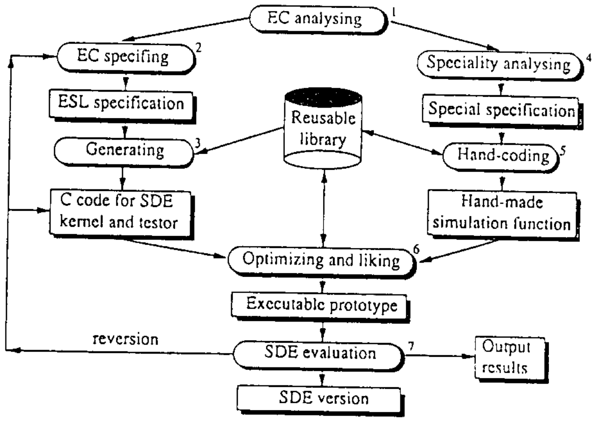


fig.2 Prototyping flow in SPACE

4. DESCRIPTION OF EC

4.1 A small specification language

We have developed a small language called ESL (Embedded Computer Specification Language) for specification of EC. ESL mainly supports for specification of high level functional behavior[5][6], it does not provide description for detail hardware circuit. The behavioral specification method aims at description only the functionalities of an EC. This method views the functional unit as a black-box and cares for mainly the input and output activities. The advantage is that, behavioral specification is generally shorter than lower level circuit specification, thus the specification is easy to write, read and change. In addition, the behavioral specification makes the simulation specification of EC easier, less error and allows a considerably shorter design cycle for SDE.

According to the object-oriented design method, in ESL, we classify an EC into several classes, each class has many objects corresponding to the entities in the EC. A class normally includes attribute part and operation part. The value of all attributes at one time represents the status of EC in run time. We view the attributes as variables and view their operations as functions calculating and changing the value of them. We have made an abstraction for the ordinary types of ECs. This abstraction includes the definition of attributes and the operation rules for each class, therefore, once the operations and attributes of an object are given, SPACE will get enough information to generate the simulation functions for the object. Based on the discussion, we designed the ESL to make it fit for specifing the type and attributes of each object to provide SPACE with necessary information for prototyping. An example of ESL file "MC05.ESL" will be given in fig.3.

4.2 The editor

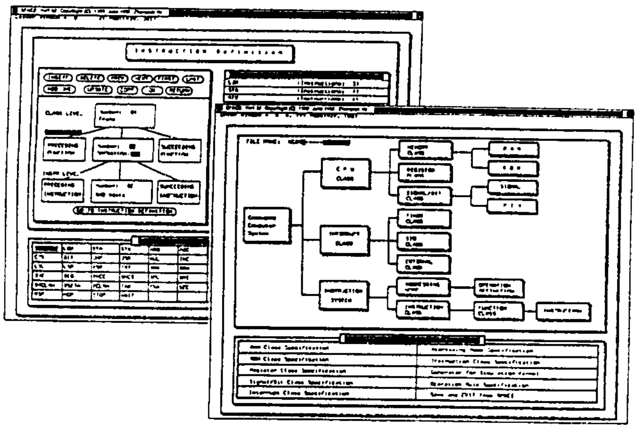


fig.3 Example windows of the ESL editor

To help user write an errorless specification, we have created a special editor for ESL. In the editor, each class of an EC is provided with one kind of definition window called "template". The definition of the attributes for an object is done in the associated template. The main role of a template is to check the input data and direct the user in his work, whenever a wrong data that is beyond the limitation of the template is inputted, the template would not confirm it and some warning or guiding messages will be given. In addition, the objects defined in one template would be cited in the definition of other classes of objects. In the end, the editor will give out a specification file with extended name of ".ESL". Fig.3 shows two eidtor's windows in which the upper window is the main frame for classes definition, and the back window is used for instruction classes definition.

5. GENERATOR

The generator uses ".ESL" file as input data, it scans the file two times. In the first time, the data frameworks for attributes of objects are formed; In the second time, C programs for construction and testing of simulation kernel are generated. YACC[7] is used for parts of the generation work. As also shown in fig.4, the following C files are the output of the generator:

- Definition head files for classes
- Data structure definition for attributes of classes
- Simulation function for operation of classes
- Instruction set index table
- Instruction simulation functions
- General simulation functions
- Program for prototype tester
- Makefile
- Warning and error message

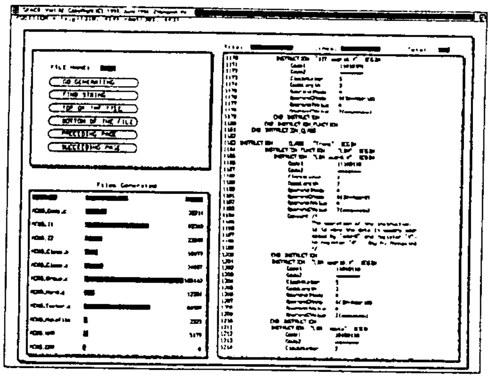


fig.4 The generator window

As to the particular operation that is impossible to specify, SPACE generates a function frame with sufficient comment and guiding information. User will refine those functions through adding some handmade code. After combining the handmade programs with the generated code under the help of the reusable library, a prototype of SDE will been established.

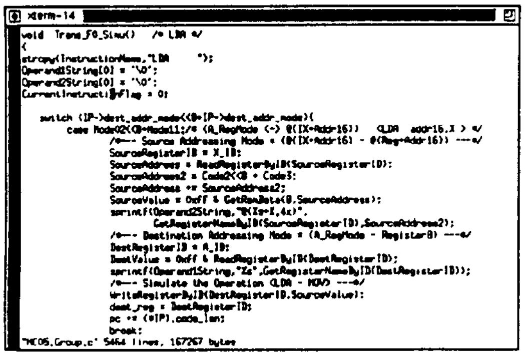
Program for prototype tester is also an output of the generator. In our experimental system, the tester includes mainly test of basic function of simulation kernel such as register transferring, instruction simulation and so on. SPACE will only select one typical state for the test work. Undoubtedly, the tester is a helpful tool for users, because it can give user a group of test-run results from which the user can easily evaluate the generated prototype. If some errors are found, the users can change their ESL specification or C file and then repeat the previous process.

6. REUSE

Prototyping of SDE in SPACE also emphasizes reuse at all levels, this means that, some routines developed previously can be utilized later. Users are able to combine compatible sets of reusable parts with each other easily at various levels and in different ways. We have developed a reusable library, the library can be accessed both in step of translating an ESL specification into C code and in step of writing a handmade program. The library includes functions supporting for simulation kernel, shell and GUI part respectively. In the library, the reuse ratio for GUI part is higher than the other two parts. Because of the specialities of EC, the kernel part has the lowest reuse ratio, therefore, part of the simulation kernel will be generated by generator. In addition, the library can be enriched by adding some extra simulation function if a reusable component can not be found. We find that the reusable library can considerably reduce the re-writing of tedious programs.

7. IMPLEMENTATION

We have made an experimental SPACE system on X11 environment of SUN workstation, fig.3,4 and fig.5 give some examples of the system. The present system main deals with 4,8 and 16 bits' microcomputers in which the instruction is limited to have two operands. As experiments, we have used SPACE to create SDE prototypes for 4 and 8 bits' microcomputers as NEC-*upd75xx*, Motorola MC68HC05 and NEC-*upd78k*. Recently, we are working toward the perfecting of SDE for 16 bits' microcomputer HITACHI H8/532.



```
void  Trans_F0_Simu()   /* LDA */
{
strcpy(InstructionName,"LDA      ");
Operand1String[0] = '\0';
Operand2String[0] = '\0';
CurrentInstructionFlag = 0;

   switch (IP->dest_addr_mode<<8+IP->dest_addr_mode){
        case Mode02<<8+Mode11;/* (A_RegMode <-> @(IX+Addr16))   <LDA  addr16,X > */
                /*--- Source Addressing Mode = (@(IX+Addr16) - @(Reg+Addr16)) ---*/
                SourceRegisterID = X_ID;
                SourceAddress = ReadRegisterByID(SourceRegisterID);
                SourceAddress2 = Code2<<8 + Code3;
                SourceAddress += SourceAddress2;
                SourceValue = 0xff & GetRomData(0,SourceAddress);
                sprintf(Operand2String,"@(%s+%.4x)",
                        GetRegisterNameByID(SourceRegisterID),SourceAddress2);
                /*--- Destination Addressing Mode = (A_RegMode - Register8) ---*/
                DestRegisterID = A_ID;
                DestValue = 0xff & ReadRegisterByID(DestRegisterID);
                sprintf(Operand1String,"%s",GetRegisterNameByID(DestRegisterID));
                /*--- Simulate the Operation (LDA - MOV) ---*/
                WriteRegisterByID(DestRegisterID,SourceValue);
                dest_reg = DestRegisterID;
                pc += (*IP).code_len;
                break;
"HC05.Group.c" 5464 lines, 167267 bytes
```

fig.5 An example of generated C code for simulation of instruction "LDA addr8,X" specified in fig.4

By qualitative estimation, about more than half of the work can be saved in comparing with the traditional method. For example, to create SDE for MC68HC05, we have used about 28K bytes of data for specifing the main parts of CPU (registers, signals, bits, memory, addressing modes, instruction set and so on) at register transformation level, then we obtain nearly 400K bytes of C code from the generator. In addition, by using of reusable library, the manual works have been reduced greatly.

8. CONCLUSIONS

SPACE is a prototyping environment for simulation debugging system of embedded computer software. It provides users not only with editor to specify the CPU's main features of an embedded computer, but also with generator to translate the specification into C code. By manual refining and the using of the reusable library, it is easy to construct a SDE whenever a new kind of embedded computer is needed. Of course there are also some problems to be resolved, we will make efforts toward the problems, a number of future enhancement is planed.

REFERENCES

[1] R.A. Saleh, I. Inoue and S. Ido, "Enhanced Circuit Simulation: Expectations, Problems, Implementation and Integration," Trans. IEICE, Vol. J74-A, No.8, 1991

[2] J. Ootani, D. Konno and T. Adachi, "A Proposal on Cooperative Development Environment for Next-Generation Circuit Simulator," Trans. IEICE, Vol. J74-A, No.8, 1991

[3] R. Hartley, et al., "A Rapid-Prototyping Environment for Digital-Signal Processors," IEEE Design and Test of Computer, June, 1991

[4] Luqi, "Computer-Aided Prototyping for A Command and Control System Using CAPS," IEEE Software, Jan., 1992

[5] R. Camposano, "From Behavior to Structure: High-Level Synthesis," IEEE Design and Test of Computers, Oct., 1990

[6] M. Ohmura, H. Yasuura and K. Tamaru, "Behavioral Verification of CPUs Using Functional Information Extraction," Trans. IEICE, Vol.J76-A,No.9, 1993

[7] S.C. Johnson, "Yacc - Yet Another Compiler-Compiler," Comp. Sci. Tech. Rep. No.23, Bell Laboratories, July 1975