Graphical block-diagram based programming environment for a DSP silicon compiler

L.-G. Jeng
D.-Y. Bai
L.-G. Chen

Indexing terms: Digital signal processing, X-window, Graphic user interface, Circuit design, Silicon compiler

Abstract: The paper presents a graphical block-diagram based programming tool, which is a new software development system for digital signal processing (DSP). This system provides a block-diagram editor; therefore, the DSP system can be easily built by connecting functional blocks as described in the signal-flow graph. A block library is set up, which includes basic signal processing functions and some frequently used high level signal processing functions like FIR, IIR, FFT, etc. A friendly graphic user interface (GUI) based on the 'X-Window' system is developed to support each design step. Users can then improve their design easily according to the GUI's response. To guarantee the correctness of the circuits designed, the debugging and verifying capabilities are embedded in our system. The full system is combined with a structural silicon compiler, which we developed previously. Several designs will be illustrated as examples. It has been shown that this tool will be suitable for DSP system design by using a silicon compiler.

1 Introduction

Several silicon compilers, including Cathedral [1], System Architect's Workbench [2], and Hyper [3], have been proposed to design DSP systems. These systems, and other general-purpose silicon compilers, always provide behavioral description languages, such as Silage [4] and ISPS [5], for use in the description of electronic circuits. Unfortunately, for large modern circuit design, the complexity of behavioral description is increasing and has encumbered documentation in design. Therefore, it is still not easy for DSP system designers to get their own DSP chips. Obviously, a much more convenient design environment for designers is needed. As graphics displays cost less and thus become more easily available, we believe it is time to take advantage of the ability of the human brain to process pictures; this is more efficient than verbal description [6]. In this situation, a graphical block-diagram based editor is then developed to help circuit designers and to document designed circuits well. In the last decade, some graphic programming tools have

been developed [7-10] in many fields. Recently, certain commercial DSP software, including Comdisco, EDC, NI!Power, Gabriel etc., are available for such techniques as gate array, standard cell, FPGA or programming DSP chips. We will ascertain the features from these tools and add some specific features of DSP circuit to create our system for DSP/ASIC chip design.

Since the goal is to use a block-diagram as input description of DSP circuits, we should develop some tools to deal with the block-diagram operation for a DSP circuit. These tools include the block-diagram editor, block-diagram syntax checker and graphic compiler. In designing these tools the significant problem encountered is how to operate these block-diagrams efficiently and friendly on the screen. Therefore, we devise a special data structure effective in editing, debugging and compiling.

The graphical entry will be translated into C-like behavioral description language and/or 'design specification behavioral description language' (DSL), which was developed by our group. The system flow chart and data flow chart of our graphic designed environment are shown in Fig. 1a and b. Finally, the synthesised data is transferred to TSP format, the input description of Genesil, before the chip layout can be obtained.

2 Block-diagram entry

2.1 Block-diagram editor

At the highest level, applications are always described as block-diagrams. A block can have arbitrary granularity and may call from a standard library which includes DSP primitive elements. Some primitive elements need parameters, such as bit numbers, coefficients and order numbers, to specify their characteristics.

Most DSP circuits can be completed by the existing function unit. The following function units are included in our system now. The first class consists in input/output operators to represent I/O port and includes four operators: input operator, output operator, constant operator, and symbol I/O operator. The second class lies in data-flow operators and includes five operators: adder, subtracter, multiplier, divider, and delay operator. The third class exists in macro operators and includes seven

The authors would like to thank the Honorary Editor, Dr. S.S. Lawson, and the reviewers for their valuable comments and suggestions which have improved the quality of this paper. This work is supported in part by National Science Council under Grant NSC82-0404-E-002-218.

© IEE, 1993

Paper 9513G (E5, E10), first received 23rd July 1991 and in final revised form 5th January 1993

The authors are with the Department of Electrical Engineering, National Taiwan University, Taipei, Taiwan 10764, Republic of China

operators: FIR, IIR, correlation, FFT, absolute value, maximum value, minimum value. The last class is the interconnection operator used to connect function units in the block diagram.

To reduce circuit complexity and/or to increase modularity, the system supports hierarchical design. Designers can split their design into several modules. For example, a 2-order lattice structure filter can be decomposed into two basic lattice blocks. We design the basic lattice block first, as shown in Fig. 2a and b; then complete a 2-order

lattice structure filter by cascading two basic lattice blocks, as shown in Fig. 2c and d.

The three terms which generally form a DSP circuit are defined as follows. A primitive function unit is the most primitive element which represents a basic DSP operation. A symbol is a function unit at the lower hierarchy which is composed of primitive function units. A circuit is comprised of symbols and function units.

Users design their DSP circuits in which the use of some subcircuits repeats itself frequently. These subcircuits can be designed as a user's defined function unit and saved in the library for further use. The process in the creation of the user's defined function unit is described as follows: first, users construct the subcircuit with the existing primitive function units and define a new symbol for this subcircuit; then, they store this new symbol in the function unit library. As given in Fig. 2b, the basic lattice block can be defined as the symbol 'lattice', and users can save the lattice in the library for further use.

a

b

Fig. 1 a System flow chart; b Data flow chart

a

b

c

d

Fig. 2 a Lattice section; b Symbol block-diagram of a; c lattice filter; d circuit block-diagram of c

2.2 Block-diagram syntax checker

Some errors may arise during block-diagram editing. They stem from mistakes in block-diagram drawing or function designing. Typical errors are floating lines and floating function units. The latter occurs when the input/output fails to make contact with any object. Syntax checking and verification are designed for this case. The errors will be detected and highlighted through the use of the syntax checker. Meanwhile, the circuit's correctness can be verified when the values in each function unit's input and output are traced.

2.3 Graphic compiler

The task of graph compilation is to translate the block-diagram into behavioral description language. Each block represents a DSP task and is corresponding to a code paragraph of the behavioral description. In our system, where are two behavioral description languages supported: C-like and design specification language (DSL). The differences between these two behavioral description languages are listed in Table 1. Users can

Table 1: The features of two hardware description languages

C-like	1. Higher level language 2. Easy to port and compatible 3. Module capability
DSL	1. Lower level language 2. Able to handle the circuit's architecture 3. More control on data operation and storage unit

select either language to meet their requirement. The graphic compilation process is depicted as follows:

- (i) Write down a code paragraph for every block except the I/P and O/P blocks.
- (ii) Assign constant value to the constant block.
- (iii) Assign the input signal to the I/P block.
- (iv) In case the first block has no signal assigned to them, start from the first block to write down the statements:
 - (a) Assign the input statement from other blocks.
 - (b) Assign the output statement to other blocks.

Then next block.

- (v) Write down the O/P paragraph for all O/P blocks.

2.4 Graphic interface

A graphic interface supports users in their design at each step in their production of circuits. This is because the graphic interface can provide a natural supplement for other means of communication. The 'translation distance' required for graphic comprehension is often small [11], and therefore the circuits more powerful and useful.

Utilising the graphic user interface, one can move, delete, load and save his drawing in the course of block-diagram editing. When debugging is under way, one selects the nets of function units, then traces the value in these nets to ensure that the circuits are correct. Having gone through these steps, the graphic compiler then translates the block-diagram to behavioral description languages. Finally, the high level synthesiser will deal with these behavioral description languages and produce the input format TSP for the Genesil CAD tool. All these steps are in contact with a graphic user interface. The user can watch the results from GUI's feedback at each step, while typical synthesis programs make their own decisions without the benefit of the user's expertise.

3 Design technique

3.1 Data structures

There are three classes of data structure in our system. Each data structure is designed for a special purpose. The first class is the editing step, at which objects are represented in the block-diagrams, and include function units as well as wires. The second class is the syntax checking step, at which the relations between function units are organised. Two types of data structure are formed: nets and function units. The third class involves graphic compilation, at this stage the only knowledge to be understood is the relations among the function units, while both the positions of the units and the data structure of the net are no longer in need. Therefore, the data structure will be condensed in the course of graphic compilation to generate the C-like or DSL codes.

3.2 Block-diagram editor

The editor basically provides drawing, modifying, loading, and saving operations. The icons of blocks are all displayed on the screen in a flat manner, as shown in Fig. 3. The icons include function units and the editor's commands. In case of the user's designed function blocks, the editor provides the open symbol and open sheet commands to define the user's designed function block. Fig. 4a shows a new function block of the symbol, which is named as a lattice. The symbol is opened in Fig. 4b. Then we need to open a design sheet to draw the circuit of this symbol on the open sheet shown in Figs. 4c and 4d. In this way, with sufficient function blocks provided, the system can handle most of the DSP algorithms based on the synchronous data flow computation model such as the DSP algorithms through multirate or feedback recursion.

Hierarchical design is permitted in our system as described in Section 2.1. In C-like language, a symbol represents a subprogram in a hierarchical block diagram. For example, a lattice block is a subprogram in a 2-order lattice filter circuit. In DSL language, DSL can deal with the architecture design such as the buffers and the buses, but DSL does not provide the feature of separation of the main program and subprograms. Thus, we should flat the whole block diagram into one hierarchical level, then treat the block diagram as one DSL program.

3.3 Syntax checker

The reason for going through the syntax checking process is that all the errors arising from the block diagram's drawing and the function's designing will be detected at this step. The data structure at the time of syntax checking is designed for this purpose. The drawing errors can be classified into two types. First, a net that contacts without a function unit's output connector called the floating net can be detected from the 'pointer of output function unit' field of corresponding net data structure. Second, I/O connectors of a function unit that contacts without a net can be deleted from the 'pointer of input net' and the 'pointer of output net' field of the respected function unit. The algorithm of detecting drawing errors is listed as follows:

Test function unit

For all function units

- (1) if their pointer of the input connector is NULL → wrong
- (2) else if one of their output connectors is NULL → wrong
- (3) else → correct

Next function unit

Test floating nets

For all nets

(1) if their out field is NULL → wrong

(2) else → correct

Next net

We provide another capability to detect the function's design error. Within a DSP block-diagram, every net contains responding signal values. We can trace the value of a net to prove the correctness of a designed block-diagram. In this manner, we should find the relation between a primitive function unit input and output nets. For example, the output net's value of one adder will be the sum of the input net's value.

We can treat a 'symbol' as a circuit, and have the syntax checker pass input values down to a lower hierarchy, evaluate these values, and then return the values up to a higher hierarchy. The algorithm is described as follows:

While not all destination nets' value is calculated

Start from the first net

do

if all preceding nets' values are calculated

calculate this net's value based on function unit's type and assign value to this net

next net

until last net

4 Example

Here, we demonstrate how to use this process in the design of a real circuit. The example circuit is a TV signal translator which transfers the *RGB* signals to luminance and chrominance. Three intermediate results will be shown. The first result is the block diagram of the signal translator created by the user. The second result is one of the two formats of behavioral description languages that our system supports. From the program listed in Figs. 5b and c, we can find that DSL language is a lower level language than C-like. In DSL language, key words like ROM, VAR, and CONSTANT appear in the program. A guidance for the user is that an experienced designer should use DSL to get a better result, while an inexperienced designer would better use C-like to bypass the flatting process. The result of the layout shown in Fig. 5d is produced by Genesil CAD tool.

5 Conclusions

This paper presents a graphical block-diagram based programming tool for DSP design using a silicon compiler. This system consists of the block-diagram editor, graphic compiler, and user interface. With a block-diagram editor, the hardware description of DSP application can be carried out easily. For a complex circuit,

Fig. 3 The screendump of the block diagram editor

hierarchy design is permitted also. After editing a block diagram, the syntax checker helps the user verify the design. If the designed block diagram is correct, the graphic compiler then translates the block diagram into behavioral description language and passes through a

high level synthesiser to Genesil. At each step, a responding data structure is designed for easy operation.

The whole system is based on X-Window [10] and therefore preserves the well-known features of compatibility and machine independence. Several design exam-

a

b

c

d

Fig. 4 a Open symbol; b after open symbol; c open sheet; d after open sheet

a

```
#include <stdio.h>
#include "exp3_c_file.fold.h"
main(argc,argv)
int argc;
char *argv[];
{
    int system_c,system_g;
    datain = fopen(argv[1],"r");
    dataout = fopen(argv[2],"w");
    while ((system_c=getc(datain))!=79)
    {
        ungetc(system_c,datain);
        fscanf(datain,"%f",&in3);
        fscanf(datain,"%f",&in2);
        fscanf(datain,"%f",&in1);
        mul3_out=in1*0.113000;
        mul2_out=0.586000*in1;
        mul1_out=in1*0.300000;
        mul6_out=in2*0.512000;
        mul5_out=in2*0.000000;
        mul4_out=in2*0.000000;
        mul9_out=in3*0.000000;
        mul8_out=in3*0.000000;
        mul7_out=in3*0.512000;
        add1_out=mul1_out+mul4_out;
        add3_out=mul2_out+mul5_out;
        add4_out=add3_out+mul8_out;
        add5_out=mul3_out+mul6_out;
        add2_out=add1_out+mul7_out;
        add6_out=add5_out+mul9_out;
        fprintf(dataout,"%f ",add2_out);
        fprintf(dataout,"%f ",add4_out);
        fprintf(dataout,"%f ",add6_out);
    }
    fclose(datain);
    fclose(dataout);
}
```

b

Fig. 5 a Block diagram of example; b C-like behavioral description language;

```

MODEL exp3 {
  INPUT_BUF[8,0] ip[3];
  OUTPUT_BUF[8,0] op[3];
  ROM[8,0] c[9]={3,5,1,0,0,5,5,0,0};
  CONSTANT[8,0] common_c[12]={0,1,2,3,4,5,6,7,8,16,32,64};
  VAR[8,0] zero,one,two,three,four,five,six,seven,eight;
  VAR[8,0] mul3_in1,mul3_out;
  VAR[8,0] mul2_in2,mul2_out;
  VAR[8,0] mul1_in1,mul1_out;
  VAR[8,0] mul6_in1,mul6_out;
  VAR[8,0] mul5_in1,mul5_out;
  VAR[8,0] mul4_in1,mul4_out;
  VAR[8,0] mul9_in1,mul9_out;
  VAR[8,0] mul8_in1,mul8_out;
  VAR[8,0] mul7_in1,mul7_out;
  VAR[8,0] add1_out;
  VAR[8,0] add3_out;
  VAR[8,0] add4_out;
  VAR[8,0] add5_out;
  VAR[8,0] add2_out;
  VAR[8,0] add6_out;
  VAR[8,0] tmp1,tmp2;
  PROCESSOR procl;
  init() {
    zero=common_c[0];
    one=common_c[1];
    two=common_c[2];
    three=common_c[3];
    four=common_c[4];
    five=common_c[5];
    six=common_c[6];
    seven=common_c[7];
    eight=common_c[8];
  }
}

main() {
  mul9_in1=ip[two];
  mul8_in1=ip[two];
  mul7_in1=ip[two];
  mul6_in1=ip[one];
  mul5_in1=ip[one];
  mul4_in1=ip[one];
  mul3_in1=ip[zero];
  mul2_in2=ip[zero];
  mul1_in1=ip[zero];
  tmp2=c[two];
  mul3_out=mul3_in1*tmp2;
  tmp1=c[one];
  mul2_out=tmp1*mul2_in2;
  tmp2=c[zero];
  mul1_out=mul1_in1*tmp2;
  tmp2=c[five];
  mul6_out=mul6_in1*tmp2;
  tmp2=c[four];
  mul5_out=mul5_in1*tmp2;
  tmp2=c[three];
  mul4_out=mul4_in1*tmp2;
  tmp2=c[eight];
  mul9_out=mul9_in1*tmp2;
  tmp2=c[seven];
  mul8_out=mul8_in1*tmp2;
  tmp2=c[six];
  mul7_out=mul7_in1*tmp2;
  add1_out=mul1_out+mul4_out;
  add3_out=mul2_out+mul5_out;
  add4_out=add3_out+mul8_out;
  add5_out=mul3_out+mul6_out;
  add2_out=add1_out+mul7_out;
  add6_out=add5_out+mul9_out;
  op[one]=add4_out;
  op[two]=add6_out;
  op[zero]=add2_out;
}

#main
#PROCESSOR
#MODEL

```

Fig. 5 c DSL behavioral description language; d layout

ples have been illustrated to show that the system does really provide a friendly design environment for DSP system designers. More power simulation tools and verification capability should be added in the future. And, more useful macro blocks should be developed for application in the specific domain.

6 References

- DE MAN, H., RABAEY, J., SIX, P., and CLAESEN, L.: 'Cathedral-II: a silicon compiler for digital signal processing', *IEEE Des. Test Comput.*, Dec. 1986, 3, (6), pp. 13-25
- POTKONJAK, M., and RABAEY, J.: 'A scheduling and resource allocation algorithm for hierarchical signal flow graphs'. Proceedings of 26th DAC, ACM-IEEE, 1989, pp. 7-12
- CHU, C.M., POTKONJAK, M., THALER, M., and RABAEY, J.: 'HYPER: an interactive synthesis environment for high performance real time application'. Proceedings of ICCD-89, IEEE, 1989, pp. 432-435
- HIFINGER, P.N.: 'A high-level language and silicon compiler for digital signal processing'. Proceedings of CICC-85, IEEE, 1985, pp. 213-216
- BARBACCI, M.R.: 'Instruction set processor specifications (ISPS): the notation and its applications', *IEEE Trans. Comput.*, Jan. 1981, C-30, (1), pp. 24-40
- GINERT, E.P., and TANIMOTO, S.L.: 'Pict: an interactive graphical programming environment', *Computer*, Nov. 1984, 17, (11), pp. 7-25
- LEE, E.A., HO, W., GOEI, E., BIER, J.C., and BHATTACHARYYA, S.: 'A design environment for DSP', *IEEE Trans. Acoust., Speech, Signal Process.*, Nov. 1989, ASSP-37, (2), pp. 1751-1762
- KRAMLICH, D.: 'Program visualization: graphics support for software development'. Proceedings of 20th DAC, ACM-IEEE, 1983, pp. 235-238
- BABB, R.G.: 'Parallel processing with large grain data flow technique', *Computer, IEEE*, July, 1984, 17, (7), pp. 55-61
- JONES, O.: 'Introduction to the X-Window system' (Prentice-Hall, Reading, 1989)
- SMITH, D.C.: 'Pygmalion: a creative programming environment', (PhD dissertation, Department of Computer Science, Stanford University (Tech. report STAN-CS-75-499, 1975)