- [6] T. A. C. M. Claasen and W. F. G. Mecklenbrauker, "Comparison of the convergence of two algorithms for adaptive FIR digital filters," *IEEE Trans. Circuits Syst.*, vol. CAS-28, pp. 510-518, June 1981.
- [7] F. R. Magee and J. G. Proakis, "Adaptive maximum likelihood sequence estimation for digital signaling in the presence of intersymbol interference," *IEEE Trans. Inform. Theory*, vol. IT-19, pp. 120-124, Jan. 1973.

An Approach to Programmable Signal Processor Assemblers and Simulators

TERESA H.-Y. MENG AND DAVID G. MESSERSCHMITT

Abstract—A method of programming an assembler and simulator for a programmable digital signal processor (PDSP) is proposed. This method is general and simple to execute, and makes use of the UNIX® utilities *awk*, *sed*, and *grep* to translate a source program, consisting of assembly instructions and C statements, to either machine code for the assembler or a C program for the simulator. The approach also provides the advantages that little programming effort is required to generate a new assembler and simulator for another processor instruction set. It also allows a PDSP to be simulated in the context of a larger system containing other PDSP's, other hardware, and an external environment. A disadvantage is that the approach is applicable only to the UNIX operating system.

I. INTRODUCTION

There are now available several single-chip programmable digital signal processors (PDSP's) [1]–[6]. The programming of a new assembler and simulator for a new PDSP instruction set is generally considered to be a major undertaking. We describe here an approach that avoids extensive programming effort, and allows an assembler and simulator to be generated in typically less than a day's effort. In addition, the approach allows the PDSP to be simulated in the context of a simulation of the system in which it is embedded, including the simulation of multiple PDSP's, associated hardware, and the external environment.

We avoid most of the programming by combining existing UNIX® operating system [7] utilities. UNIX utilities translate a source program for the target PDSP to machine code for the assembler and generate a C program for the simulator. Simulation is accomplished by compiling and executing the simulator C program that was generated. The main advantage of the approach is its universality and the ease with which new assemblers and simulators for different PDSP's can be generated from an existing assembler and simulator. Also, debugging C statements (not executed by the PDSP's) can be added to the assembly source file, and the debugging of assembly code is aided by the symbolic debugging facilities of UNIX.

The TI TMS32010 PDSP was used as an example. Starting with the TMS32010 assembler and simulator, it took only about 5 h to generate a simulator for the Fujitsu MB8764 and 10 h for the TMS32020. In Section II we describe the basic approach, and Section III gives conclusions. A basic familiarity with UNIX utilities and the C language is assumed.

Paper approved by the Editor for Transmission Systems of the IEEE Communications Society. Manuscript received September 30, 1985; revised April 15, 1986. This work was supported by the Semiconductor Research Corporation.

The authors are with the Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, CA 94720.

IEEE Log Number 8611280.

II. GENERAL STRUCTURE AND METHODOLOGY

Our approach is to consider the source file as text and avoid extensive programming by using UNIX-provided text editors to modify the contents. The commands borrowed from UNIX are:

- grep** A family of commands that search the input files for lines matching a pattern and copy those lines to the standard output.
- sed** A noninteractive stream text editor.
- awk** A pattern scanning and processing language which searches a set of files for patterns performs specified actions upon lines or fields of lines which contain instances of those patterns.

A. Implementation of the Assembler

An assembler generates machine code of a target PDSP corresponding to a given assembly language program. Our assembler is a mnemonic translator with no facilities for macros, link loading, etc. In our case, the source program as a whole, including assembly instructions and optional added C statements (to be included in the simulation), is regarded as lines of character strings. The **grep** command extracts every line that is recognized as a particular assembly mnemonic (this is repeated for each mnemonic), **sed** translates the mnemonic to an appropriate machine code, and **awk** generates the hexadecimal representation to be stored in the program ROM. Indirect addressing, labels, constant definitions, expressions, and dynamic memory allocations are simple ways to represent addresses of the data RAM in an instruction, as provided by assembly directives, and are realized by some special **awk** commands. Imbedded C statements (for the simulation) are ignored by the assembler.

The three major steps in assembly are described in more detail in the following three subsections.

1) *Separation of Assembly Instructions from C Statements*: The assembler first removes imbedded C statements. Our convention is to add an identification `"/**/"` at the start of each line containing a C statement, and these lines are removed by the **sed** command `"/*/*.*\*/d"`.

2) *Label Translations*: Branch instructions require that labels be translated into exact program locations. This can be accomplished by parsing the source program twice. In the first pass, each assembly instruction is assigned a program address (by an **awk** program) which is placed at the beginning of each line. This same **awk** program generates a **sed** command file, which contains statements like

`s/label/address/.`

On the second pass this command file is used to replace the branch destination in the source file with the corresponding address. Subroutine calls are treated in the same way to obtain the addresses of subroutines. More complicated branching expressions such as argument passing in subroutines or destination addresses relative to the instruction location are easily implemented by an **awk** command.

3) *Machine Code Generation*: In the final step, each assembly instruction goes through a pipelined sequence of commands such as

```
egrep 'instr1|instr2|..'
input_file|sed -f sed.instr|awk -f awk.instr
```

where *instr1*, *instr2*, etc., are the assembly mnemonics that have the same instruction field format. Because the number of different instruction formats is usually quite small (seven for the TMS32010), the assembling process needs only a few lines of command sequences, as described above.

The **egrep** command passes only the lines in the input file

which have one of the character strings specified in the first argument. Labels, specified by lower case characters, cannot be confused with mnemonics. The **sed** command file **sed.instr** translates each *instr* to a sequence of bits. Finally, the function **awk.instr** combines the opcode bits and operands to generate hexadecimal machine code. These commands split the original source input file into pieces, each containing similar groupings of instructions. Since each assembly line has been given an address at the beginning of the line, the instructions are reordered by concatenating these files together and using the UNIX command **sort**.

B. Implementation of the Simulator

The purpose of a simulator is to simulate and debug the assembly program by reproducing its functionality on the VAX. The simulator consists of two phases.

1) Translation phase: Translate assembly mnemonics to C functions line by line.

2) Simulation phase: Compile the resulting C program while linking to a collection of such functions defined in supporting files, and then run the resulting object program.

Translation is performed similarly to the assembler. **Awk** programs are used to convert assembly mnemonics into C functions and, at the same time, generate auxiliary code and files whenever necessary.

For the simulation phase, all the opcode functions are provided in a header file **instr.h** and a file of C functions **support.c**. These two files are unique to a particular processor and instruction set, and are the primary files which must be rewritten to generate a simulator for a new PDSP. These files are straightforward to generate: given the instruction description, we simulate the functionality of that instruction by defining a C macro statement. Our experience is that for present generation PDSP's, generating a new assembler and simulator takes less than one day, although it is difficult to extrapolate this experience to new generations which generally have much more complicated instruction sets.

The general approach to the design of the simulation C program which is code generated from the assembly program is as follows.

1) Each register and data memory location is assigned an integer variable to provide a simple way to trace assembly instructions.

2) Each instruction is executed as a single function call. Groups of similar instructions share a single function call with arguments generated through the use of the C macro preprocessor.

3) The simulator has been interfaced with BLOSIM [8] to achieve simulations of multiprocessor systems and simulations of a PDSP chip in combination with other hardware.

With these features in mind, the following subsections describe the phases of the code generation and the structure of the resulting C program.

1) *Translation Phase*: The translation phase is different from the assembler translation only in that the output is a C program instead of machine code. The same sequence of commands suffices: separate C statements from assembly code, define constants, label each instruction, assign function names to instructions, and finally reorder the instructions using the labels. These steps are described in the following subsections.

a) *Separation of Assembly and C Statements*: First, C statements are separated from assembly instructions; the former are simply inserted in the final simulation program. Numerical labels are added to each line of the source file so that, after C statements are extracted from the source file, they can later be reinserted in the processed assembly program to become the simulator. A simple **awk** program and the command **sort** are used.

b) *Generation of the Flow-of-Control Program*: A

single C main program is generated to represent the entire assembly program for simulation. The sequence of assembly instructions is preserved in the simulator C program to ease the task of debugging. Therefore, an assembly language subroutine (lines of mnemonics expressing the functions of this subroutine) is represented not as a C subprogram, but rather as lines of C statements in the single main program. We simulate a direct branch or subroutine call with a **goto** statement to a corresponding label placed in the C program.

Flow-of-control statements where the destination address is known only at runtime represent a special problem. Examples of such statements are the instructions **RET** and **CALA** in the TMS320 and indirect branches. **RET**, which causes a return at the end of a subroutine, is usually realized with a stack that stores the return address at its top. Since all the registers, including the program counter and the stack pointer, have been declared as integer variables, the problem of simulating this instruction can be treated by software in the same way as by hardware. However, the return address must be indicated, since there is no explicit address associated with each instruction in the simulator. **CALA** uses the current accumulator value as the subroutine address, and this is known only at runtime. Indirect branches on register values have no specified destination in the instruction, so programmers need to specify the allowable branch destinations by putting labels in the source file. (This also helps in debugging the program at run time).

The way we handle these special flow of control instructions is best illustrated by the following fragment of an automatically generated C simulation program:

```
acc_branch:
switch(DESTIN) {
  case 0:
    goto reset;
    break;
  case 2:
    goto return1;
    break;
  case 45:
    goto nopfunc;
    break;
  default:
    return(101);
    break;
}
reset:
/* Load accumulator with constant 45. */
LDA(LACK(45));
/* Call the subroutine whose address */
/* is the accumulator content. */
CALA(); goto acc_branch;
return1:
.
.
.
nopfunc:
LDA(LACK(0));
RET(); goto acc_branch;
```

This program fragment may be confusing at first, but the idea is simple. Since branch destinations in the TMS320 program are represented by numerical values (for example, the contents of the accumulator), the branching is performed by the switch statement added by the **awk** program which records the program address of each assembly instruction. Control is transferred based on the value of numerical variable **DESTIN**, the program address of the next instruction. **DESTIN** is assigned a value in the function associated with that particular instruction. The number of cases in the switch statement is the sum of twice the number of subroutines (one

for **CALA** and one for **RET**) and the number of labels specified by the user for indirect branching. The first instruction (at address 0x000, label **reset**) loads the number 45 into the accumulator so that **CALA** can call subroutine **nopfunc** by setting **DESTIN** equal to the content of the accumulator. The statement **goto acc_branch**, added by the translator, directs the flow of the program to the **switch** statement, which makes a second jump to the desired subroutine block beginning at address 0x045 (label **nopfunc**). At the end of the subroutine, **RET** is realized by, again, the translator-added **goto acc_branch** statement. At this time **DESTIN** is set to 2, the value at the top of the stack, which is the address of the next instruction. The label **return1** is also added by the translator to provide a destination for the **goto** statement. Thus, **RET** and **CALA** function properly while the basic structure of the assembly program is retained in the C program.

c) *Translation of Assembly Mnemonics to C Functions:* Various **awk** functions translate different instructions to corresponding C functions. All the instructions calling for actions on the same kind of registers are given a common function name. The nature of the call and an example are given in Section II-B-2.

d) *Interfaces:* The simulator can interface with I/O devices in two ways: using I/O files or using the standard BLOSIM interface to other simulation routines. If the simulator runs as an independent program, the port number is given as a parameter to define the I/O files in input and output instructions. On the other hand, simulations of multiple PDSP's and simulations of a PDSP in combination with other types of hardware are achieved by defining the PDSP as one block in a BLOSIM simulation. The BLOSIM interface code, such as initialization code, parameter definitions, and state variable declarations for each register and memory location, is added automatically. This interface begins execution if input data are ready in an input buffer, and causes a context switch if no input data are available. After a context switch, the simulator begins execution at the stopped instruction if new data are provided.

2) *Simulation Phase:* Execution of the simulation C program requires two supporting files **instr.h** and **support.c**, which depend strongly on the PDSP being simulated.

After being translated, each instruction is represented by

```
function(instr(arguments));
```

where *function* is a C function that simulates the instruction and *instr(arguments)* gives the arguments to that function. The arguments are generated by macros in **instr.h**, which defines the actions taken by each instruction in the form

```
#define instr(arguments) actions
```

In turn, these actions are actually function arguments which are passed to *function* defined in **support.c**, and specialize that function to the particular instruction. This scheme allows us to group similar instructions together as a single function in **support.c**, and then specialize that function to individual instructions by macros in header file **instr.h**. For example, all ALU instructions can be grouped together, while all data move instructions form another category.

Following is an example of using three macros in file **instr.h** and function **ACC()** in file **support.c** to implement "add to accumulator with shift," "subtract from accumulator with shift," and "load accumulator with shift." Each instruction is simulated by the C function **ACC(instr(argu1, argu2))**.

In file "instr.h":

```
#define ADD(a, s) acc,(ram[a + dp*128] < < s)
#define SUB(a, s) acc, - 1*(ram[a + dp*128] < < s)
#define LAC(a, s) 0,(ram[a + dp*128] < < s)
```

In file "support.c":

```
/* ACC:
 * Simulate the TMS320 accumulator instructions.*/
ACC(acum, val)
int acum, val;
{
    acc = acum + val;
    /* Overflow test is omitted for simplicity.*/
}
```

The C functions in **support.c** are the real implementation of the simulator, since all the other code added in translation is simply associated with flow of control and not the actual execution of instructions. This supporting file is easy to program given the description of the instruction set, since all the peculiarities of the assembly language have been processed to a universal format before simulation. As for **instr.h**, we simulate instructions by copying the instruction description from the processor manual in a straightforward fashion.

III. CONCLUSIONS

A disadvantage of this approach is that the assembler and the code generation portion of the simulation are slow, although the simulation is quite fast. As a concrete example, a TMS32010 assembly program implementing a 34-order low-pass FIR filter, which occupied 66 ROM locations (including coefficients), was written. Assembly took 5.5 s and translation to a simulator C program took 5.2 s on a VAX 750. The actual simulation time for 8000 samples was 160 s. It is unlikely that a simulator which is much faster could be designed, but the assembly and simulator code generation phases could be speeded up considerably by replacing the time-critical UNIX commands with specialized C programs.

The approach described here is most useful and appropriate for quickly generating an assembler and simulator for a new custom or application-specific PDSP instruction set, where performance is less important than design effort. It is less appropriate for widely used PDSP catalog parts because of its performance and nonportability outside UNIX. This technique illustrates the power and utility of the UNIX operating system philosophy in simplifying complex programming tasks.

The ability to simulate multiple PDSP's or PDSP's as a part of a larger system is another important capability, and one that is largely independent of the particular simulator implementation approach.

REFERENCES

- [1] *TMS32010 User's Guide*, Texas Instruments, Inc., 1983.
- [2] R. C. Chapman, "Digital signal processor," *Bell Syst. Tech. J.*, vol. 60, Sept. 1981.
- [3] Fujitsu Labs, Ltd., "A high performance LSI digital signal processor for communication," in *Proc. IEEE Int. Conf. Commun.*, June 1983.
- [4] "Digital signal processor, data sheet for the uPD7720 signal processor interface (SPI)," NEC Electronics U.S.A., Inc.
- [5] OKI Elec. Ind. Co., Ltd., "A digital signal processor for data modem," in *Proc. IEEE Int. Conf. Commun.*, June 1983.
- [6] *2920 Analog Signal Processor Design Handbook*, Intel Corp., Aug. 1980.
- [7] D. M. Ritchie, "The evolution of the UNIX time-sharing system," *Bell Syst. Tech. J.*, vol. 63, Oct. 1984, part 2.
- [8] D. G. Messerschmitt, "A tool for structured functional simulation," *IEEE J. Select. Areas Commun.*, vol. SAC-2, Jan. 1984.