A Subset FORTRAN Compiler for a Modified Harvard Architecture

J. R. Parker

Abstract

The architecture of the target computer clearly has a great impact on the design of a compiler's code generator. In particular, the near complete separation of the code and data memories of the TMS32010 microprocessor creates some interesting problems.

I. Introduction

FORTRAN is a relatively old programming language. Its age shows up in every aspect of its design; the use of line boundarys as statement terminators, the lack of any dynamic storage allocation, the lack of procedure nesting, and the prevailence of labels and GOTO statements, and many other characteristics have caused this language to fall into disuse in many applications areas. Still, many programs originally written in Fortran are still in use, and in some areas, particularly engineering and heavily numerical disciplines, it it still very popular.

A great deal of signal processing software has been written in Fortran, which is one motivation for interest in a TMS32010 Fortran compiler. Indeed, Fortran has become the language by which workers in signal and image processing can communicate their methods. When the TMS32010 signal processor and its successors became commercially available, it seemed inevitable that a Fortran language compiler would sooner or later be needed for them.

II. The Target Processor

A block diagram of the TMS32010 appears in figure 1. Being a Harvard type architecture, the data memory and the code memory are physically separate and logically distinct. Access to operands in data memory is very fast: on the order of 100 nSec. Access to instructions in program memory is also very fast ( ), but to access data residing in program memory can take 1.2 microseconds or longer. As a result, operands should always be stored in data memory. There are two problems with this. First, data memory is quite small, only 144 16 bit words, so it will often be the case that not all data will fit. Second, there is no fast mechanism for loading constants of arbitrary size into the data memory. There is a load constant instruction, but this is only effective for unsigned numbers less than 256. The alternative is a slow transfer from program memory.

Other features of the TMS32010 architecture also interfere with normal code generation. The program memory is not very large, consisting of at most 4096 words (12 bit address). A cross compiler would be difficult enough to implement, but a native compiler with reasonable language features would be nearly impossible. Also, the hardware stack has only four levels; since return addresses are stored here, this is a serious restriction. The TMS32010 is an accumulator based processor with very few registers, and these all have a special purpose. This creates a problem with storage for temporary variables, which must be allocated in data memory. There are only two addressing modes that are at all general: **direct** from data memory, and **indirect** through an *auxilliary register* to data memory. There are a few instructions with unique modes,

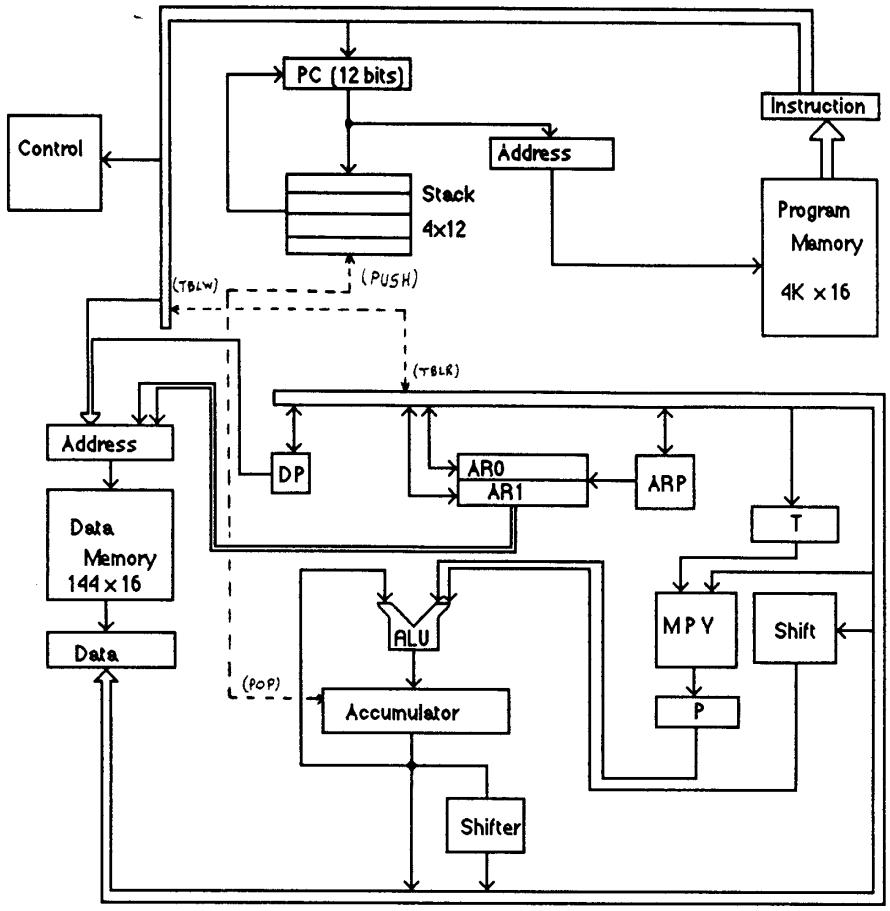

Figure 1 - The TMS32010 Processor

such as the load constant instruction described above, but these are limited in application.

The attractive features of the processor, especially for signal processing, are its speed and simplicity. A sixteen bit by sixteen bit multiply, giving a thirty-two bit result, takes 200 nanoseconds. This speed is possible in part by the use of instruction pipelining, which in turn is made possible by the separation of program and data memories. While assembler programs for specific purposes can be made to run very quickly, the code generated by a compiler will not take full advantage of the architecture and will be much slower as a result. The programmer makes a speed/programming time tradeoff on many systems, but it is more pronounced on this special purpose computer.

It is possible to 'extend' the data memory by connecting a fast memory to the I/O ports and reading from the memory using IN and OUT instructions. This uses extra hardware that can be constructed in many incompatable ways. Because of this, the F320 compiler was designed for use with the basic processor with no extensions; however, because of the way that I/O is implemented, the option of extending the processor remains.

III. The Compiler

The compiler, named F320, was written in PASCAL, and implements a subset of the FORTRAN 66 language. There are also a few extensions that apply specifically to the target processor, and are mostly implemented as builtin functions. Restrictions of note are:

1. Separate compilation is not implemented. Subroutines, functions, block data, and the main program must reside on the same source file. This violates a basic tenet of FORTRAN, but should not be a very serious problem in practice; given the restricted program memory size, F320 programs will of necessity be small. Moreover, this allows the checking of the number of arguments passed to a routine against the number declared.
2. The number of dimensions allowed on array variables is two or less. Again, the small memory creates the need for this restriction. Not only do multidimensional arrays grow in size very quickly, the code needed to computed the address grows in size also. Very few FORTRAN programs use three or more dimensions in any case.
3. For reasons discussed later, variables residing in COMMON storage cannot be passed by reference as arguments.
4. Some features not involving the generation of code were omitted. Principal amcng these are **equivalence** statements and **statement functions.** Since the major interest of the study relates to the generation of code, these features were thought to be unimportant.
5. The only data types implemented so far are INTEGER and LOGICAL, and arrays of these. Floating point is being examined very carefully, and if included will be at the programmer's option. The inclusion of REAL variables in a program will result in the appropriate library functions being included in the object file. Fixed point may be a more reasonable alternative given the memory size of the machine.
6. All input and output involves unformatted integers. FORMAT statements are often interpreted at runtime, requireing both memory and time. The TMS32010 performs 16 bit output in 400 nanoseconds, which seems like a better idea. I/O is performed directly to and from a selected I/O port on the chip to simplify interconnection.

Should a production compiler be implemented, the omissions above can all be corrected.

IV. Code Generation

The compiler generates assembler code, which in turn requires two more passes to assemble, making a total of three passes. The optimizer is an optional fourth pass. Most of the hard decisions in the code generator were the result of the tradeoff between the small memory size and the fast speed of the processor.

1. Local variables and arguments were allocated space in data memory. This includes variables whose declarations are allowed to default. Common variables reside in program memory, resulting in slower access times and more extensive code generation for variable access. This gives the programmer an explicit means of controlling the space-time tradeoff for a particular application. The sample TMS32010 code below shows the difference in accessing local and common variables.

PROBLEM: Load the accumulator with the value of variable A.

Declaration:	Declaration:
INTEGER A	INTEGER A COMMON /X/ A
Simple Local: Get value	Simple common: Get value
LAC A / Load acc with / contents of / location A	CALL LDAC / Load acc with address dec A+X / of A in block X. TBLR * / Move from prog. Mem. / into data memory. LAC * / Load value into Acc.

The LDAC procedure is used to load a constant of arbitrary size into the accumulator.

The LDAC procedure is a library routine, which will only be loaded if it is used. In fact, the same is true of all library procedures, such as builtins and the divide and exponentiation routines.

2. Temporary variables used during the evaluation of an expression are allocated on a kind of stack, implemented by using one of the two auxilliary registers in indirect mode. In this mode, the operand resides at the address indicated by the register, which may be incremented or decremented after the access. The '*' character in assembly programs indicates this mode of access through whichever auxilliary register is indicated by a 1 bit register called the *auxilliary register pointer*, or ARP. Hence a 'push', or store of a temporary, would be coded as:

```
SACL *-
```

which stores the current accumulator value through the current AR into data memory, then decrements the AR that was used. In this scheme, the AR used is initialized to 143, the highest address in data memory, and grow down towards zero. Popping a temporary value is performed by:

```
MAR *+          / Increment AR value
LAR *       / Load Accumulator through AR.
```

Note that the increment must be done first, so in general this takes two instructions, although the optimizer can eliminate many of these.

3. Subroutine calls use a normal CALL instruction, which places the return address on the hardware stack. Since there are only four levels, the subroutine first pops this address from the stack and moves it into program memory. This permits the nesting of calls more than three deep; the instructions that move data to and from program memory (TBLR and TBLW) also use the top of the stack. The code for a call to a subroutine X appears below:

```
CALL X                  SUBROUTINE X

 produces:        produces:

call X_             r_X:   dec 0      / Place for return address
                    X_:    pop        / Get Return address
                           sacl *     / Store in data mem.
                           call LDAC  / Load program memory address
                           dec r_X    / into accumulator
                           tblw *     / Store return address in
                                      / program memory.
```

The **return** statement generates code to reverse this process, placing the return address

back on the stack.

4. Arguments were a problem. The restricted memory and stack size permits few reasonable solutions, and of course FORTRAN passes all arguments by reference. Since all of the compilation units reside on the same file, the solution that was chosen involves writing the addresses of the actual arguments passed into the actual locations allocated for the formal arguments. If these do not exist by the time the subroutine is called, they are allocated, and will be not be reallocated when the actual **subroutine** statement is seen. This could be modified for separate compilation, but is much simpler without. The value stored in the argument address is the address of the variable or expression passed, in data memory. For this reason *common variable cannot be passed by reference;* they do not reside in data memory, and the subroutine receiving the argument will have no way of knowing in which memory the address is appropriate. Solutions were tried involving negative addresses for common variables and allowing common to exist only in addresses that do not overlap with data memory (IE address $>$ 144). The complexity of the code needed to properly determine the location of the argument at runtime and retrieve its value was too great considering the rewards. In any event, common variables are shared among all subroutines and functions, and the *value* of a common variable can be passed as an expression. Other solutions are to allocate common variables in data memory, or to implement a call by value result. The former solution is unacceptable, the latter is under investigation.

5. Division and exponentiation are implemented as assembly code procedures, using a normal call-return sequence instead of the slower scheme above. The instruction set includes a multiply, but not a divide, instruction. Sample code for two simple expressions is:

```
Expression: I*2+1                Expression: K/I*3

lac 0      /Load I               lac 2      / Load K, addr 2
sacl *-    / Store temp          sacl *-    / Store value
lack 2     / Load 2              lac 0      / Load I, addr 0
mar *+     / Pop I val           sacl *     / Store value
lt *       / into multiplier     call div_  / Perform division.
sacl *     / Store 2             sacl *-    / Save result
mpy *      / Multiply            lack 3     / Load constant 3
pac        / result to Acc       mar *+     / Pop value from stack
sacl *-    / push result         lt *       / Div result into multiplier
lack 1     / Load 1              sacl *     / constant 3 to stack
mar *+     / Pop result          mpy *      / Multiply
add *      / And add 1.          pac        / Result to accumulator

TIME: 2.4 Microseconds           TIME:   19.6 Microseconds

Optimized level 1:

lac 0                            lac 2
sacl *                           sacl *-
lack 2                           lac 0
lt *                             sacl *
sacl *                           call div_
mpy *                            sacl *
pac                              lack 3
sacl *                           lt *
lack 1                           sacl *
add *                            mpy *
                                 pac

TIME: 2.0 Microseconds           TIME: 19.4 Microseconds

Optimized level 2:

lt 0       / I into multiplier   lac 2
mpyk 2     / Mult. by 2                    sacl *-
lack 1     / 1 into acc.                   lac 0
apac       / Add result                    sacl *
```

```
                                call div_
                                lt *
                                mpyk 3
                                pac
```

Time: 0.8 Microseconds TIME: 18.8 Microseconds

The above code shows a number of things. First, optimization can reduce execution time by a factor of three in some cases. Second, the lack of a division instructions costs dearly, and division should therefore be used sparingly if possible. Third, the use of data memory as a 'stack' for temporaries is shown, as is the fact that the stack need never get very deep.

VI. Conclusions

The choice of a Non-Von Neumann processor as the target of a compiler, even a simple one, can result in the need to make painful compromises. Clearly, other choices could have been made at various points in the implementation of the F320 compiler, and some of these are still being explored. For example, there is obvously much work to be done in the optimization of the code that would be very profitable, and this is being done. Still, much of the advantagee of a special purpose processor can be lost in a high level language code generator - much more than for a more general purpose device.

Work is continuing with an implementation of floating point and fixed point numbers, with the optimization pass, and with the construction of very efficient formatted I/O routines.

References

[1] Texas Instruments Ltd., *TMS32010 User's Guide,* 1983.

[2] Baldwin, R.D., *Thr Backward-Directed GOTO in FORTRAN,* SIGPLAN Notices Vol. 19 No. 8, Aug 1984.

[3] VanTuyl, R.R., *On Evolution Of FORTRAN,* SIGPLAN Notices Vol 19 No. 11, Nov. 1984

[4] Aho, A.V., Ullman, J.D., *Principles of Compiler Design,* Addison-Wesley Publishing, Reading Mass., 1977.