Report

A computer system for model helicopter flight control technical memo Nr. 6: the Oberon compiler for the strong-ARM processor

Author(s):
Wirth, Niklaus

Publication Date:
1999

Permanent Link:
https://doi.org/10.3929/ethz-a-006653165 →

Eidgenössische
Technische Hochschule
Zürich

Departement Informatik
Institut für
Computersysteme

Niklaus Wirth

A Computer System for Model Helicopter Flight Control

Technical Memo Nr. 6: The Oberon Compiler for the Strong-ARM Processor

March 1999

314

A Computer System for Model Helicopter Flight Control

Technical Memo Nr. 6

The Oberon Compiler for the Strong-ARM Processor

N. Wirth 7. 3. 99

Abstract

We describe a compiler, and in particular its generated code, translating from Oberon to binary code for the Strong–ARM RISC processor. It is a very compact single–pass system (< 50K bytes). Compiled modules are linked and down–loaded into the embedded processor. This memo is intended to provide programmers information about generated code patterns in order to be able to estimate the complexity of their programs.

0. Introduction

In late 1997 we decided to express the control system for the model helicopter in terms of the language Oberon. At the same time, the decision was made to use DEC's StrongARM processor DS1035 as the core of the control system. An immediate consequence of these decisions was the necessity of an Oberon compiler for that processor, and, since none was available, to build one. An important objective was to demonstrate the language's suitability for expressing programs for real–time applications, requiring efficiency and predictable performance, and encouraging structured and modular design.

Although compiler technology is a reasonably mature subject, most people hesitate to build their own compilers, even if considerable manpower is available, which was not the case here. A major reason for this hesitation is that compilers are considered to be inherently complex programs. At the latest since the advent of RISC computers, there is also good reason for this belief. The credo of the RISC movement was that computer architectures should not be designed to please the compiler builders or the high–level language designers, but that instead utmost code execution efficiency was the only valid goal. To bridge the widening "semantic gap" between language and computer was the very objective of compiler technology. Computer and compiler were to be seen as a married couple, and a highly complex, "optimizing" compiler was to be, if not an unpleasant burden, a natural necessity. And indeed, processors have become so fast that the worst compilation times are tolerated because hardly noticeable, and compiler sizes in the order of megabytes are fatalistically accepted.

But the consequence that hardly anyone will consider building a compiler exists and is deplorable. The path closest to compiler building is to construct a parser for one's special language using a toolbox yielding a translator generating C code. This option is based on the belief that by tapping the vast C software one gains all the benefits of highly optimizing compilers and sophisticated debugging tools without investing much effort. It turns out to be a sobering experience.

In this project, we chose a different approach, namely that of building our own compiler from scratch. This was reasonable in view of the author's previous experience in compiler design, but still subject to controversy concerning the way to achieve "optimized code". A very sophisticated, highly optimizing code generating strategy was obviously out of the question. A first version of the compiler was to be available within a few months, and the workforce consisted of one part–time person, the author. Instead, I decided to build the compiler along the lines of a simple, single–pass compiler with on–the–fly code generation as elaborated in [1]. Having a scanner and a parser already available saved a few days of labour. The crucial part, without doubt, was the code generator.

Therefore, we will report on the conventional parts of the compiler only scantily, and rather discuss in some more details our approach to achieving the efficiency of generated code that an engineer building a real–time control system demands. It turns out, that he must be able to depend on predictable performance, a requirement more fundamental than mere efficiency. Modern processors perform poorly

in this respect; their instruction pipelines and data caches speed up average performance considerably, but provide worst cases that let this gain turn out to be of marginal value. Sophisticated compilers, rearranging and perturbing code in obscure ways, add to this misery. A compiler producing code that can be derived from the source program according to few, obvious rules is a much better solution. Given the fact that modern RISC architectures depend strongly on sophisticated code generation algorithms, we feared that we might have to sacrifice considerably with respect to run–time efficiency.

However, the performance of most systems depends critically only on the performance of a small fraction of the entire program, typically the inner loops contained in procedures at the outer fringes of the procedure calling hierarchy. With this in mind, we proceeded to add a few facilities to the language – a freedom only the compiler builder possesses – facilities with the aim to let the programmer direct the compiler at producing effient code without sophisticated source analysis. For example, the programmer may specify certain simple variables to be allocated in registers. Such facilities, simple to implement in the compiler, require some deeper understanding from the programmer. It has been critizised that it was exactly the duty of the compiler to free the programmer from such implementation–specific considerations. As this may be, our experience showed that competent programmers do have this understanding anyway and appreciate being able to express it. The much smaller size and complexity of the compiler, and its predictability, are infinitely more valuable. Indeed, one may observe that many compilers have made it much too easy to concoct optimized lousy code.

1. The Structure of the Compiler

The compiler is structured conventionally into modules according to the following tasks: Scanning, parsing, and code generation. In addition, a base module is provided containing the definitions of data types used throughout the compiler, in particular the types *Object*, denoting elements of the "symbol table", and *Type*, describing data types. This module also contains the routines for building up and inspecting the symbol table and, moreover, generating and loading symbol files.

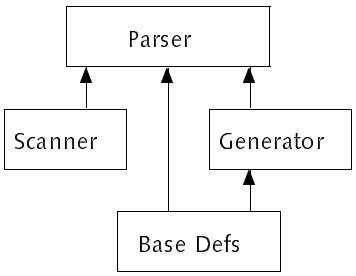

Fig. 1. The compiler's module structure

The structure is shown in Fig. 1. It ignores a single import of all modules from the Scanner, namely the import of a single procedure to mark the position of detected errors in the source file. This was necessary, because the scanner "owns" the source file rider, i.e. no reading position is available outside module Scanner.

Module	Data "owned"	Source lines	Code bytes	Data bytes
Scanner	Source file, rider, table of keywords	247	3904	1188
Parser	(main module)	710	10188	52
Generator	Type *Item*, Code array, register allocation	1036	13712	9988
Base	Types *Object* and *Type*, symbol table	357	4292	100
Total		2350	32096	11508

Care has been taken to divide the compiler into the two main modules (parser, generator) according to the following criterion: Parts that are dependent on the source language but independent of the target architecture were to be placed in module OSAP (parser), whereas those that depend on the target would

belong to module OSAG (generator). As a result, the interface of the generator has become fairly large (60 procedures).

2. The Scanner (OSAS)

The task of the scanner is the recognition of language symbols (tokens) in the source text. They are identifiers, integers, real numbers, strings, and other symbols. The latter are either special characters, character pairs, or keywords:

*	/	DIV	MOD	&	+	–	OR
=	#	<	<=	>	>=	↑	↑–
.	,	!	:	)	]	OF	THEN
THEN	DO	TO	BY	(	[	~	:=
FALSE	TRUE	NIL	;	END	ELSE	ELSIF	UNTIL
IF	WHILE	REPEAT	FOR	RETURN	ARRAY	RECORD	POINTER
RIDER	CONST	TYPE	VAR	PROCEDURE	BEGIN	IMPORT	MODULE

RIDER, ↑–, and ! are symbols added due to language extensions. On the other hand, the following are symbols not used in the Oberon–SA subset. They are nevertheless recognized by the scanner in order to ensure compatibility with full Oberon implementations.

IN	IS	..	|	}	{	CASE	LOOP
WITH	EXIT						

The keywords are listed in a hash table. Hash table and function are such that most keywords are recognized in the first try, and only a few require two tries. The source text is passed as a parameter to the scanner by calling OSAS.Open. Then the source file is attached to the corresponding rider which keeps track of the current reading position. Procedure OSAS.Mark is used to issue error diagnostics which refer to the current source position.

3. The Parser (OSAP)

The syntax of the language Oberon is arranged such that programs can be parsed by a straightforward, top–down parsing scheme with a lookahead of one symbol. We use the implementation technique of recursive descent. In principle, every nonterminal class is represented by a parsing procedure. Since the parser module is supposed to process all aspects that are independent of the target architecture, type checking is also performed by this module. Code generation, however, is delegated to the generator module through appropriate procedure calls.

The parsing of declarations results in the construction of the data structure traditionally called "symbol table", where records are kept of all declared entities, organized according to their scope of visibility. For each procedure declaration entered, a new scope is opened. Its local declarations form a linear list (field *next*) of records of type *Object.*

```
TYPE Object = POINTER TO RECORD
        class, lev, expo: INTEGER;
        next, anc: Object;
        type: Type;
        name: Name;
        val: LONGINT
    END ;
```

```
TYPE Type = POINTER TO RECORD
        form, ref, nofpar, opt: INTEGER;
        dsc, typobj: Object;
        base: Type;
        size: LONGINT
    END ;
```

Record field *class* designates the class of object represented. The various values are:

1	Var	Variable (or value parameter)
2	Par	VAR parameter
3	Const	Constant
4	Fld	Record field

5	Typ	Data type
6	SProc	Standard, inline procedure
7	Mod	Module
8	Reg	Register variables (or value parameter in leaf procedure)
9	RegI	VAR parameter in leaf procedure

Field *lev* denotes the level of nesting of the declared object. 0 stands for global level. *expo* specifies whether or not the object is exported (from the global scope). *name* is the identifier (ascii string) denoting the object, and *type* is a pointer to its type descriptor. *val* stands for the object's value in the case of a constant. Procedures are represented as constants of a procedure type.

In records of type *Type* the field *form* denotes the structure of the respective type. The various values are:

1	Bool	BOOLEAN	
2	Char	CHAR	
3	Int	INTEGER	
4	Real	REAL	
5	Pointer	pointer type	base = type of dereferenced object
6	Rider	rider type	base = array element type
7	Nil		
8	No type	(proper procedure)	
10	Proc	procedure type	base = result type, dsc = parameter list
11	String		
12	Array	array type	base = element type
13	Record	record type	dsc = field list

For all forms, the field *size* indicates the number of bytes taken by an object of the respective type. The field *ref* serves to hold a flag during the process of generating a symbol file (OSAB.Export). The field *opt* is reserved for procedure types to indicate special cases such as leaf procedures or interrupt handlers.

The compiler's parsing procedures carry a parameter of type *OSAG.Item.* They not only determine whether a syntactic construct had been properly recognized, but also deliver a description of the construct in the form of an *Item* record. It contains among other details used for code generation the data type of the construct. The parser uses this information to perform the required type consistency checks. As an example consider

```
PROCEDURE term(VAR x: OSAG.Item);   (*x := x op y*)
    VAR y: OSAG.Item; op: INTEGER;
BEGIN factor(x);
    WHILE (sym >= OSAS.times) & (sym <= OSAS.and) DO
        op := sym; OSAS.Get(sym);
        IF op = OSAS.and THEN CheckBool(x); OSAG.And1(x); factor(y); CheckBool(y); OSAG.And2(x, y)
        ELSIF x.type.form = OSAB.Real THEN factor(y); CheckReal(y); OSAG.RealOp(op, x, y)
        ELSE CheckInt(x); factor(y); CheckInt(y); OSAG.MulOp(op, x, y)
        END
    END
END term;
```

The important point is that *Items* are always variables local to parsing procedures. They are never allocated dynamically on the heap, and therefore need never be collected as "garbage". This contributes significantly to a compiler's efficiency, and it is a natural consequence of the recursive–descent parsing strategy.

4. The Strong-ARM Architecture

Before explaining certain code sequences corresponding to specific language constructs, we outline the architecture of the Strong-ARM processor (Advanced Risc Machines) and some of its peculiarities. At first sight, it represents the classic RISC configuration with a set of 16 registers and an arithmetic–logic unit (ALU). Instructions specify a destination register for the result, a first source operand from a register, and a second operand either from a register or as a constant. A unique and very useful feature is that the

second operand can be shifted before being applied to the specified operation.

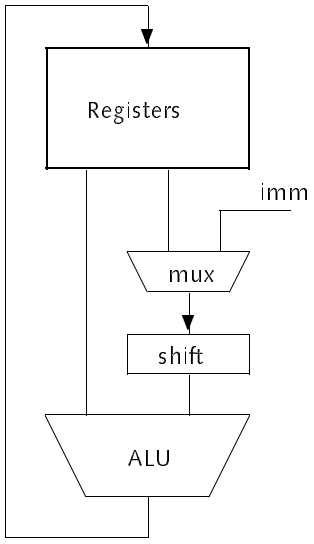

Fig. 1. Data paths of the RISC core

Thus, the basic instructions have the following forms and formats. The operations include addition, subtraction, the basic logical operations, and straight copying.

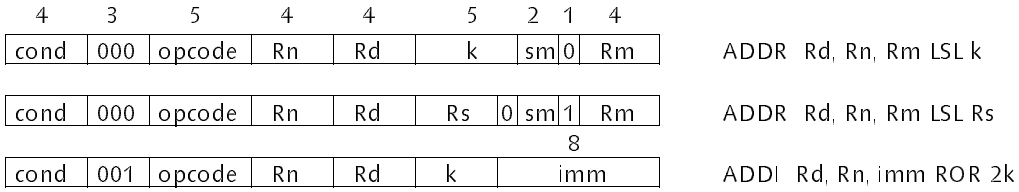

Fig. 2. Register instruction formats

The mode of shifting (sm) can be specified as logical left, logical right, arithmetic right, and rotate right. One of the quirks of the ARM is that constants are specified as an 8–bit field only. This field, however, can be shifted to any place within the 32–bit word. Nevertheless, this is an awkward restriction, and it is particularly aggravating, because every instruction wastes 4 bits for specifying under which condition the instruction is to be executed. This facility is usually available for (conditional) branch instructions only.

A similarly awkward restriction governs the load and store instructions. The address is computed either as the sum of a register and an offset, or of two registers. The offset is a 12–bit field only; 16 would have been much better!

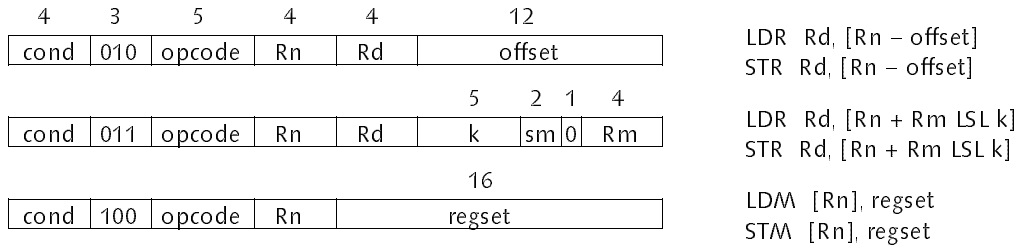

Fig. 3. Load/store instruction formats

Branch instructions, on the other hand, feature a large, PC–relative word offset of 24 bits. Register R15 is the program counter PC. For procedure calls the Branch and Link instruction is used which deposits the current PC in register R14. Upon return from the procedure, that value is moved into R15 causing a return jump.

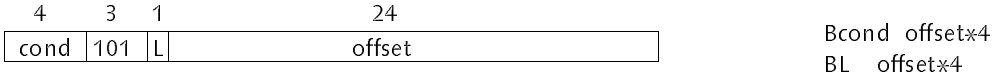

Fig. 4. Branch instruction format

In our implementation, register R13 is reserved as stack pointer SP, and R12 as local frame pointer FP. As a result, only registers R0 – R11 are available for intermediate results during expression evaluation.

5. Storage Layout, Variable Addressing, and Procedure Calls

Every module occupies a storage block for its global variables, and following it a block for the code. After the code follows the area for constants. The stack or workspace contains the procedure frames containing the parameters and local variables (see Fig. 5).

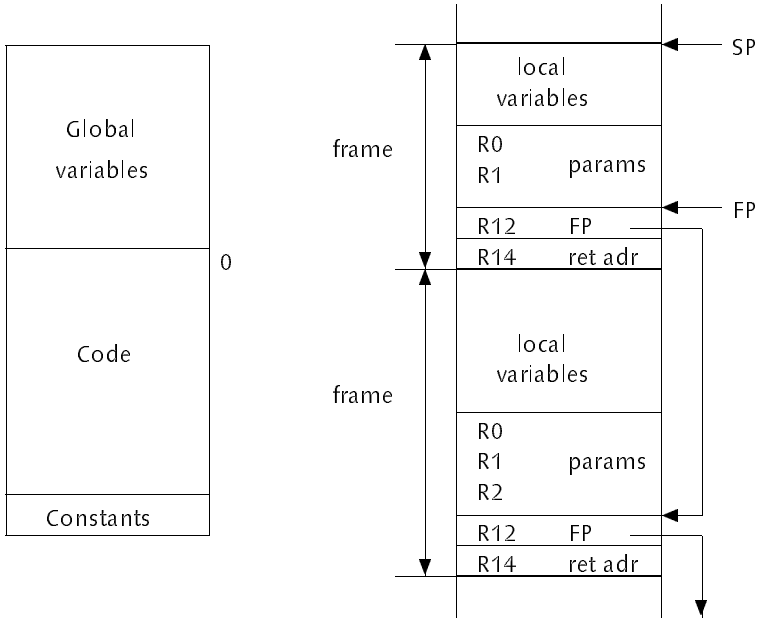

Fig. 5. Module block and stack frames

Before the call of a procedure, its actual parameters are evaluated and stored in the work registers in ascending order. A branch and link instruction transfers control to the procedure and stores the return address in R14. The entry code consists of 3 instructions. They moye the parameters, FP, and the return address onto the stack, and they adjust FP and SP:

```
STM   SP, [R0, R1, ... , R12, R14]   push onto stack parameters, FP, and return address
ADDI  FP, SP, m                      FP := SP + m,  m = size of param space
SUBI  SP, SP, n                      SP := SP + n,  n = size of local variable space (omitted if n = 0)
```

The exit code restores the stack to the state before the call:

```
MOVR  SP, FP                         SP := FP
LDM   SP, [R12, R15]                 pop from stack FP and return address (to PC)
```

Evidently entry and exit code sequences are quite short. This is mainly due to the convenient multiple load and store instructions which can be programmed as push and pop operations.

Variables are addressed relative to a base address. In the case of local variables the base is the FP register (R12), and the offset is negative. Global variables use PC–relative addresses, i.e. the base address is the PC (R15). This makes changing addresses at load–time unnecessary for module–internal references. We allow access only to variables that are either global or strictly local. This makes it unnecessary to introduce a so–called static link for procedure calls, which would introduce additional overhead. Nevertheless, procedures can be nested.

Oberon–SA features so–called *leaf procedures.* Their entry code differs in so far as the parameters are *not* pushed onto the stack. In the special case of *fast leaf procedures*, neither FP nor the return address are stored. In the extreme case of a fast leaf procedure without local variables, the entry code vanishes, and the exit code reduces to a single instruction copying R14 into R15. Hence the term fast is well justified.

In Oberon–SA, scalar variables can be declared to be allocated in registers. If local to a procedure marked as leaf, they are allocated adjacent to the parameters, otherwise in descending order starting with R11 as shown in Fig. 6.

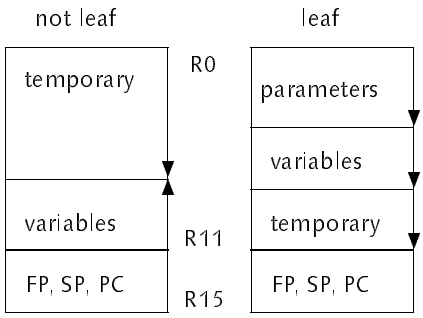

Fig. 6. Register allocation

6. Constant and Address Generation

Constants are generated by instructions with an "immediate" field. In the case of the Strong–ARM processor, the actual constant field is only 8 bits wide. It is augmented by a 4–bit rotate count which specifies the position in the 32–bit word where the 8–bit constant is to be placed. The MVNI instruction can be used to position an 8–bit constant before inversion of the entire 32–bit word.

For values requiring more than 8 bits, there are two possible solutions: Either compose the value through a sequence of OR–instructions, or place the constant in some memory location and access it through a regular load instruction. We chose the latter alternative. The short 12–bit offsets in load instructions practically force a placement of the constants between procedures. Therefore, the compiler collects constants in a table and appends them to the code at the end of each procedure. This may lead to further problems if procedure code is longer than 1K words.

The offset field of load and store instructions is 12 bits wide, restricting offsets to 4K bytes or, more significantly, to 1K instructions. Particularly in the case of global variable access, where the offset is relative to the instruction itself, this value is too small in most cases. The obvious solution is to add an 8–bit value shifted by 12 bit positions using an additional ADDI instruction.

Apart from loads and stores, addresses must be generated as actual parameters corresponding to formal VAR parameters. In this case, the same severe 8–bit limitation applies as in the case of constant generation, with the result that offset values must quite frequently be placed in separate memory locations. The short constant and offeset fields in the ARM–instructions appear as an unpleasant, genuine design flaw. This critizism is supported by the presence of a 4–bit condition field in every instruction that is hardly ever used (with the exception of conditional branches, of course), and could have been used for lengthening the offset and immediate fields.

```
PROCEDURE P(x, y, z: INTEGER; VAR w: INTEGER);
BEGIN P(5, 53248, –3, z)    (*13 * 2↑12 = 53248*)
END P;

  0 E92D 500F   STM   [0, 1, 2, 3, 12, 14]   entry code, save x, y, z, w, FP, LINK on stack
  1 E28D C010   ADDI  R12, R13, 16           FP := SP + 16
  2 E3A0 0005   MOVI  R0, 5                  R0 := 5
  3 E3A0 1A0D   MOVI  R1, 13 ROR 20          R1 := 53248
  4 E3E0 2002   MVNI  R2, 2                  R2 := –3
  5 E24C 3F02   SUBI  R3, R12, 2 ROR 30      R3 := FP – 8
```

```
6 EBFF FFF8    BL      P               P
7 E1A0 D00C    MOVR    R13, R12        exit code, SP := FP
8 E8BD 9000    LDM     [12, 15]        restore FP, PC
```

7. Code Generation for INTEGER Expressions

The parameter of type *Item* of parsing procedures not only specifies the type of the construct parsed (see above), but also its *mode.* That is, it specifies whether for example a term is located in a register, in store as a local or global variable, whether it is directly or indirectly addressed, and so on.

```
TYPE Item = RECORD
              mode: INTEGER;
              type: OSAB.Type;
              a, b, r: LONGINT   (*private fields*)
            END ;
```

An *Item* is essentially like an *Object,* but whereas an Object denotes a named entity and belongs to the symbol table, the Item is a temporary entity occurring during expression evaluation. Items are allocated as local variables in the stack, whereas Objects are allocated in the heap. The attribute *mode* corresponds to the attribute *class* of objects. The set of Item modes is primarily determined by the set of addressing modes featured by the processor. An *Object* is transformed into an *Item* by procedure *OSAG.MakeItem.* A copy of its attributes is made rather than passing a reference. This is advisable, because Items, i.e. their type and mode, may change, for example due to detection of type inconsistencies, whereas Objects must never be altered.

Var denotes the normal addressing of variables in memory, with attribute *r* specifying the base address (in register FP for local, PC for global variables), and *a* the offset. *Par* denotes indirect access (as needed for parameters) with *r* and *a* used as in the *Var* case, and with *b* denoting the second offset. *Const* denotes a constant with *a* being its value, *Reg* a variable or intermediate result in register *r*, and *RegI* a variable's address in register *r*. *CC* specifies that the operand is represented by the condition code (procesor status flags), *Var1* that an operand's address is the sum of the two values held in registers *r* and *a*. And finally *RegQ* indicates that the denoted variable is indirectly accessed via a rider in register *r* with stride *a*.

1	Var	Variable (or value parameter)	a = offset		r = base reg
2	Par	VAR parameter	a = offset0	b = offset1	r = base reg
3	Const	Constant	a = value		
4	Fld	Record field	a = offset		
6	SProc	Standard, inline procedure	a = proc. no.		
8	Reg	Register variables			r = reg no
9	RegI	VAR parameter in leaf procedure			r = adr reg no
12	Var1	Variable	a = reg no		r = reg no
13	CC	Condition code	a = F–chain	b = T–chain	r = cond code
14	RegQ	Rider mode	a = stride		r = reg no

The result of an evaluation is either assigned to a variable, or it is passed as a parameter to a procedure. In both cases, the value must be placed in a register. The respective *Item* must have *mode = Reg.* Procedure *load* causes values to be transferred into registers. Registers are used like a stack to hold intermediate results. Consider the following assignments and the resulting code:

```
PROCEDURE P(x, y: INTEGER; VAR z: INTEGER);
    VAR w: INTEGER;
BEGIN w := x – 9; z := (x + y) * (x – y)
END P;
```

```
3 E51C 000C    LDR     R0, [FP–12]     mode of x is Var
4 E250 0009    SUBI    R0, R0, 9       mode of "9" is Const
5 E50C 0010    STR     R0, [FP–16]     mode of w is Var
```

```
 6 E51C 000C    LDR     R0, [FP–12]          x
 7 E51C 1008    LDR     R1, [FP–8]           y
 8 E090 0001    ADDR    R0, R0, R1           mode of x+y is Reg
 9 E51C 100C    LDR     R1, [FP–12]          x
10 E51C 2008    LDR     R2, [FP–8]           y
11 E051 1002    SUBR    R1, R1, R2           x–y
12 E010 0091    MUL     R0, R0, R1           (x + y) * (x – y)
13 E51C 1004    LDR     R1, [FP–4]           mode of z is Par
14 E581 0000    STR     R0, [R1+0]           z
```

If parameters and local variables are held in registers, the code is considerably shorter:

```
PROCEDURE* P(x, y: INTEGER; VAR z: INTEGER);
    VAR w: INTEGER*;
BEGIN w := x – 9; z := (x + y) * (x – y)
END P;
```

```
 2 E250 3009    SUBI    R3, R0, 9            w := x–9
 3 E090 4001    ADDR    R4, R0, R1           R4 := x+y
 4 E050 5001    SUBR    R5, R0, R1           R5 := x–y
 5 E014 0495    MUL     R4, R4, R5           R4 := R4*R5
 6 E582 4000    STR     R4, [R2+0]           z := R4
```

8. Local Code Improvements

The employed technique of code generation on–the–fly, in conjunction with delayed code emission in the case of constant operands, allows for some code improvements without additional complication. Apart from the direct evaluation of expressions of constants (constant folding). Obvious cases are multiplication or division by a power of 2, where a single shift replaces a multiply or a divide instruction. For the Strong–ARM this is particularly essential, because there is no divide instruction at all. The facility of a combined shift and add instruction goes one step further: It allows the use of a shift in all cases where the multiplier is of the form 2^l or 2^{l+1}. In order to keep the code listing in the following example reasonably short, variables are allocated in registers:

```
PROCEDURE* P(x, y: INTEGER);
    VAR z: INTEGER*;
BEGIN z := x * 8; z := 3*x;
    z := x DIV 4; z := x MOD 256; z := x MOD 1000000H
END P;
```

```
 2 E1B0 2180    MOVR    R2, R0 LSL 3         z := 8*x
 3 E090 2080    ADDR    R2, R0, R0 LSL 1     z := x + 2*x
 4 E1B0 2140    MOVR    R2, R0 ASR 2         z := x DIV 4
 5 E200 20FF    ANDI    R2, R0, 0FFH         z := x MOD 100H
 6 E3D0 24FF    BICI    R2, R0, 0FFH ROR 8   z := x MOD 1000000H
```

The modulus operation is implemented by a AND instruction, if the modulus is 2^n with n <= 8. It mask out all but the last n bits. If 24 <= n < 32, a BIC instruction is used with an appropriate shift.

The described optimizations can be handled well within the concept of context–free compilation. This means that the code for any parsed sentential construct depends only on its syntactic constituents. We have breached this concept in the case of assignments (and in the function ODD). Without this breach, an assignment to a register variable, say *z := x+y*, would first evaluate the expression, placing the sum into some work register. The parsing procedure for assignment would then have to move this result into the register representing z. In order to avoid this redundant move instruction, the last instruction previously issued is inspected, and its destination field is changed accordingly. This implies a code fixup of the sort systematically performed in peephole optimization.

9. Code Generation for REAL Expressions

The DS1035 processor does not feature a floating–point unit. Therefore, operations on numbers of type REAL must be programmed as procedures using integer arithmetic. First, in the expectation of acquiring a floating–point unit later when available, an emulator package was designed. The compiler generated ARM floating–point instructions, which the processor would recognize as undefined coprocessor instructions causing a trap. When it turned out that an FP–unit would not be forthcoming, we decided to replace the trap interpreter by a regular module containing a set of procedures representing the basic floating–point operations:

```
PROCEDURE* Add(x, y: INTEGER);
PROCEDURE* Multiply(x, y: INTEGER);
PROCEDURE* Divide(x, y: INTEGER);
PROCEDURE* Floor(x: INTEGER);
PROCEDURE* Float(x: INTEGER);
```

These procedures are implemented as fast leaf procedures. It turned out as a big surprise that this change boosted floating–point performance by the large factor of 3. The primary reason is that the saving and restoring of the entire bank of registers can be omitted. The reasonably complex decomposition of instructions into their component fields during trap handling is avoided too.

The drawback is that the arguments must, according to the general parameter convention, be placed in registers R0 and R1, with the result being returned in register R0. Therefore, either expressions must be restricted and never have more than a single intermediate result, or intermediate results must be saved (pushed on the stack) and restored (popped from the stack). It was felt that the restriction would be too much of an inconvenience, but also that the push and pop operations would introduce intolerable overhead hidden from the programmer. The solution from this dilemma is a compromise: We relax the restriction and allow a single intermediate result in the hope of covering most cases occurring in practice. Push and pop operations are avoided by introducing replicas of the *Add* and *Multiply* procedures in the run–time support module that expect parameters to be in R1 and R2 instead of R0 and R1, with the intermediate result in R0 remaining unaffected.

```
PROCEDURE F(x, y, z: REAL): REAL;
BEGIN z := x – (y + (z + 1.0)); RETURN (x + y) * (x + z)
END F;

  2 E51C 0004   LDR    R0, [FP–4]          z
  3 E3A0 15FE   MOVI   R1, 3F800000H       1.0
  4 EB01 0000   BL                         Add
  5 E51C 1008   LDR    R1, [FP–8]          y
  6 EB01 0004   BL                         Add
  7 E51C 100C   LDR    R1, [FP–12]         x
  8 E220 0102   XORI   R0, R0, 2 ROR 2     invert sign
  9 EB01 0006   BL                         Add
 10 E50C 0004   STR    R0, [FP–4]          z

 11 E51C 000C   LDR    R0, [FP–12]         x
 12 E51C 1008   LDR    R1, [FP–8]          y
 13 EB01 0009   BL                         Add
 14 E51C 100C   LDR    R1, [FP–12]         x
 15 E51C 2004   LDR    R2, [FP–4]          z
 16 EB02 000D   BL                         Add1
 17 EB03 0010   BL                         Multiply
```

Comparisons, sign inversion, and computation of the absolute value are simple enough to generate inline code rather than calling an imported procedure. The following example shows the resulting code:

```
PROCEDURE F(x, y, z: REAL);
BEGIN z := –x; z := ABS(y);
   IF x < y THEN z := 1.0 END ;
```

```
   IF x = 2.0 THEN z := 1.0 END
END F;

   2 E51C 000C   LDR   R0, [FP–12]          x
   3 E220 0102   XORI  R0, R0, 2 ROR 2      invert sign
   4 E50C 0004   STR   R0, [FP–4]           z
   5 E51C 0008   LDR   R0, [FP–8]           y
   6 E3C0 0102   BICI  R0, R0, 2 ROR 2      clear sign
   7 E50C 0004   STR   R0, [FP–4]           z

   8 E51C 000C   LDR   R0, [FP–12]          x
   9 E51C 1008   LDR   R1, [FP–8]           y
  10 E010 2001   ANDR  R2, R0, R1
  11 41E0 0000   MVNR  R0, R0               complement R0, if R0 and R1 negative
  12 41E0 1001   MVNR  R1, R1               complement R1, if R0 and R1 negative
  13 E150 0001   CMPR  R0, R1
  14 AA00 0001   BGE   1                    branch to 17, if x >= y
  15 E3A0 05FE   MOVI  R0, 3F800000H        1.0
  16 E50C 0004   STR   R0, [FP–4]           z

  17 E51C 000C   LDR   R0, [FP–12]          x
  18 E350 0101   CMPI  R0, 1 ROR 4          2.0
  19 1A00 0001   BNE   1                    branch to 22, if x # 2.0
  20 E3A0 05FE   MOVI  R0, 3F800000H        1.0
  21 E50C 0004   STR   R0, [FP–4]           z
```

10. Code Generation for BOOLEAN Expressions, IF–, and WHILE statements

Boolean (logical) expressions are implemented using conditional jumps. Consider first the simple case of the comparison x < y. It is translated into a single compare instruction which places the result into the three flag bits Z, N, and C of the processor status register, from which all relations (=, <, > etc.) can be derived. As a consequence, we represent the Boolean result of a comparison by the relation used in the comparison, that is, by Item mode CC with field r specifying the relation.

IF– and WHILE statements contain Boolean expressions. Their compilation schemata are the following:

```
IF x < y THEN S END

      CMP   x, y
      BGE   L     (branch if not less)
      code(S)
L:    ...

WHILE x > y DO S END

L0:   CMP   x, y
      BLE   L1    (branch if not greater)
code(S)
      BR    L0
L1:   ...
```

The corresponding parser routine is:

```
IF sym = OSAS.if THEN
    OSAS.Get(sym); expression(x); CheckBool(x); OSAG.CFJump(x);  (*conditional forward jump*)
    Check(OSAS.then, "no THEN");
    StatSequence; L0 := 0;
    WHILE sym = OSAS.elsif DO
        OSAS.Get(sym); OSAG.FJump(L0); OSAG.Fixup(x); expression(x); CheckBool(x);
        OSAG.CFJump(x); Check(OSAS.then, "no THEN"); StatSequence
    END ;
    IF sym = OSAS.else THEN
        OSAS.Get(sym); OSAG.FJump(L0); OSAG.Fixup(x); StatSequence
    ELSE OSAG.Fixup(x)
    END ;
    OSAG.FixLink(L0); Check(OSAS.end, "no END")
ELSIF sym = OSAS.while THEN
    OSAS.Get(sym); L0 := OSAG.pc; expression(x); CheckBool(x); OSAG.CFJump(x);
    Check(OSAS.do, "no DO"); StatSequence; OSAG.BJump(L0);  (*back jump to L0*)
    OSAG.Fixup(x); Check(OSAS.end, "no END")
...
```

The problem of forward jumps in a single pass compiler is easily solved by retaining the location of the branch instruction whose address has to be supplied at a later time in a local variable L0. In the case of a conditional statement with several elsif clauses, many forward branches will occur that lead to the same location, and whose address will therefore have to be fixed up at the same time. This case is solved by building a list of those instructions; the links are stored in the instructions themselves in the place of their eventual addresses. Procedure *FixLink* traverses this list, replacing the links by the branch offsets.

In the case of a Boolean expression with AND and OR operators, a conditional jump is taken as soon as the result is known. In Fig. 7, BF denotes a branch to be taken if the preceding expression yields FALSE, and BT a branch if TRUE.

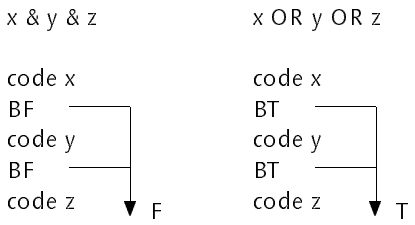

Fig. 7. Implementation of AND and OR with conditional branches

The Item descriptors of Boolean terms and expressions therefore do not only specify a condition code, but also (the origins of) chains of branch instructions that have to be completed by inserting branch destination addresses. This is shown by a short example, where again the variables are allocated to registers in shorten the code listings.

```
PROCEDURE* P(x, y, z, w: INTEGER);
    VAR a: INTEGER*;
BEGIN
    IF x < y THEN a := 1 END ;
    IF (x < y) & (z < w) & (a < 0) THEN a := 1 END ;
    IF (x < y) OR (z < w) OR (a < 0) THEN a := 2 END
END P;

  2 E150 5001   CMPR  R0, R1        R0 = x, R1 = y
  3 AA00 0000   BGE   0             branch to 5, if not less
  4 E3A0 4001   MOVI  R4, 1         R4 = a

  5 E150 5001   CMPR  R0, R1        R0 = x, R1 = y
  6 AA00 0004   BGE   4             branch to 12, if not less
  7 E152 5003   CMPR  R2, R3        R2 = z, R3 = w
  8 AA00 0002   BGE   2             branch to 12, if not less
  9 E354 5000   CMPI  R4, 0         R4 = a
 10 AA00 0000   BGE   0             branch to 12, if not less
 11 E3A0 4001   MOVI  R4, 1

 12 E150 5001   CMPR  R0, R1
 13 BA00 0003   BLT   3             branch to 18, if less
 14 E152 5003   CMPR  R2, R3
 15 BA00 0001   BLT   1             branch to 18, if less
 16 E354 5000   CMPI  R4, 0
 17 AA00 0000   BGE   0             branch to 19, if not less
 18 E3A0 4002   MOVI  R4, 2
```

11. Indexed Variables

Access to elements of an array, expressed by indexed variables, requires the computation of an address at run–time. Consider the array declaration

a: ARRAY n_2, n_1, n_0 OF INTEGER

and the indexed variable a[i, j, k]. Its address is then

base(a) + i*s_2 + j*s_1 + k*s_0

s_0 = size(INTEGER) $s_1 = n_0 * s_0$ $s_2 = n_1 * s_1$ $s_3 = n_2 * s_2$ = size(a)

Consider the example

```
PROCEDURE P;                              adr              size
    VAR i, j, x, y: INTEGER;              –4, –8, –12, –16 4
        a: ARRAY 8 OF INTEGER;            –48              32
        b: ARRAY 3, 5 OF INTEGER;         –108             60
BEGIN x := a[4]; x := a[i]; y := b[i, j]
END P;

  3 E51C 0020   LDR   R0, [FP–32]         a[4]
  4 E50C 000C   STR   R0, [FP–12]         x
  5 E51C 0004   LDR   R0, [FP–4]          i
  6 E08C 0100   ADDR  R0, FP, R0 LSL 2    base + 4*i
  7 E510 0030   LDR   R0, [R0–48]
  8 E50C 000C   STR   R0, [FP–12]         x

  9 E51C 0004   LDR   R0, [FP–4]          i
 10 E3A0 1014   MOVI  R1, 20              5*4
 11 E020 C091   MUL   R0, FP, R0, R1      base + i*20
 12 E51C 1008   LDR   R1, [FP–8]          j
 13 E080 0101   ADDR  R0, R0, R1 LSL 2    + j*4
 14 E510 006C   LDR   R0, [R0–108]
 15 E50C 0010   STR   R0, [FP–16]         y
```

If the length of a subarray is a power of 2, the multiply/add instruction is replaced by the faster shift/add:

```
PROCEDURE P;                              adr          size
    VAR i, j, x, y: INTEGER;              –4, –8, ...  4
        b: ARRAY 4, 4 OF INTEGER;         –80          64
BEGIN y := b[i, j]
END P;

  3 E51C 0004   LDR   R0, [FP–4]          i
  4 E08C 0200   ADDR  R0, FP, R0 LSL 4    base + 16*i
  5 E51C 1008   LDR   R1, [fp–8]          j
  6 E080 0101   ADDR  R0, R0, R1 LSL 2    + j*4
  7 E510 0050   LDR   R0, [R0–80]
  8 E50C 0010   STR   R0, [FP–16]         y
```

These examples demonstrate the use of the combinations of multiply and add, and shift and add operations in a single instruction, leading to rather short instruction sequences for address computations.

If the arrays are formal VAR–parameters, indexing is combined with indirect addressing. This is shown by the following example:

```
TYPE Vector = ARRAY 8 OF INTEGER;
     Matrix = ARRAY 4, 4 OF INTEGER;

PROCEDURE P(VAR a: Vector; VAR b: Matrix);
    VAR i, j, x, y: INTEGER;
BEGIN x := a[0]; x := a[i]; y := b[i, j]
END P;

  3 E51C 0008   LDR   R0, [FP–8]          adr(a)
  4 E590 0000   LDR   R0, [R0]            a[0]
  5 E50C 0014   STR   R0, [FP–20]         x
```

```
 6 E51C 0008   LDR    R0, [FP–8]          adr(a)
 7 E51C 100C   LDR    R1, [FP–12]         i
 8 E790 0101   LDRR   R0, [R0 + R1 LSL 2] adr(a) + 4*i
 9 E50C 0014   STR    R0, [FP–20]         x

10 E51C 0004   LDR    R0, [FP–4]          adr(b)
11 E51C 100C   LDR    R1, [FP–12]         i
12 E51C 2010   LDR    R2, [FP–16]         j
13 E080 0102   ADDR   R0, R0, R2 LSL 2    adr(b) + 4*j
14 E790 0201   LDRR   R0, [R0, R1 LSL 4]  + 16*i
15 E50C 0018   STR    R0, [FP–24]         y
```

12. Riders

One of the few extensions of Oberon–SA is the rider feature. It is used to traverse an array sequentially element by element. In the following example, the inner product of the arrays a and b is computed:

```
PROCEDURE InnerProd(VAR a, b: ARRAY OF REAL; n: INTEGER): REAL;
    VAR s: REAL; ra, rb, limit: RIDER REAL;
BEGIN s := 0.0; SET(ra, a, 0); SET(rb, b, 0); SET(limit, a, n);
    WHILE ra < limit DO s := ra↑ * rb↑ + s END ;
    RETURN s
END InnerProd;
```

Riders are a particularly attractive feature on processors that offer an addressing mode with implied pre– or post increment or decrement. The Strong-ARM is such a processor, allowing to implement queues and stacks very efficiently. The rider represents a pointer (address) to be held in a register. Access via rider implies indirect access via that register with automatic increment of the rider address. The code for the preceding function procedure is shown below:

```
 3 E3A0 0000   MOVI   R0, 0
 4 E50C 0010   STR    R0, [FP–16]         s := 0
 5 E51C B00C   LDR    R11, [FP–12]        SET(ra, a, 0)
 6 E51C A008   LDR    R10, [FP–8]         SET(rb, b, 0)
 7 E51C 900C   LDR    R9, [FP–12]
 8 E51C 0004   LDR    R0, [FP–4]          n
 9 E089 9100   ADDR   R9, R9, R0 LSL 2    SET(limit, a, n)

10 E15B 0009   CMPR   R11, R9             ra < limit
11 AA00 0006   BGE    6                   branch to 19
12 E49B 0004   LDR    R0, [R11], 4        ra↑, post inc adr
13 E49A 1004   LDR    R1, [R10], 4        rb↑, post inc adr
14 EB02 0000   BL     Product
15 E51C 1010   LDR    R1, [FP–16]         s
16 EB01 000E   BL     Sum
17 E50C 0010   STR    R0, [FP–16]         s
18 EAFF FFF6   B      –10                 branch to 10
19 E51C 0010   LDR    R0, [FP–16]         s
```

13. FOR statements

Instead of using a WHILE or a REPEAT statement, an iteration can be expressed by a FOR statement, if an integer value is progressing over a range of values. The following example computes the sum 1 + 2 + ... + n:

```
PROCEDURE H(n: INTEGER): INTEGER;
    VAR i, sum: INTEGER;
BEGIN sum := 0;
    FOR i := 1 TO n DO sum := sum + i END ;
    RETURN sum
END H;
```

```
 3 E3A0 0000   MOVI   R0, 0
 4 E50C 000C   STR    R0, [FP–12]        sum := 0
 5 E3A0 0001   MOVI   R0, 1

 6 E51C 1004   LDR    R1, [FP–4]         n
 7 E150 0001   CMPR   R0, R1
 8 CA00 0007   BGT    7                  branch to 17
 9 E50C 0008   STR    R0, [FP–8]         i
10 E51C 000C   LDR    R0, [FP–12]        sum
11 E51C 1008   LDR    R1, [FP–8]         i
12 E090 0001   ADDR   R0, R0, R1
13 E50C 000C   STR    R0, [FP–12]        sum
14 E51C 0008   LDR    R0, [FP–8]         i
15 E280 0001   ADDI   R0, R0, 1          +1
16 EAFF FFF4   B      –12                branch to 6
17 E51C 000C   LDR    R0, [FP–12]        RETURN sum
```

This example also shows the substantial savings gained, if H is marked as a leaf procedure with its variables and parameters allocated in registers:

```
PROCEDURE* H(n: INTEGER): INTEGER;
    VAR i, sum: INTEGER*;
BEGIN sum := 0;
    FOR i := 1 TO n DO sum := sum + i END ;
    RETURN sum
END H;
```

```
 2 E3A0 2000   MOVI   R2, 0              sum := 0
 3 E3A0 1001   MOVI   R1, 1              i := 1

 4 E151 1000   CMPR   R1, R0             i < n
 5 CA00 0002   BGT    2                  branch to 9
 6 E092 2001   ADDR   R2, R2, R1         sum := sum + i
 7 E281 1001   ADDI   R1, R1, 1          i := i+1
 8 EAFF FFFA   B      –6                 branch to 4
 9 E1A0 0002   MOVR   R0, R2             RETURN sum
```

14. Device Access and Interrupt Handlers

An essential property of a language for control applications is its ability to express direct access to devices. In Oberon–SA it is represented by three procedures (which in standard Oberon must be imported from the pseudo–module SYSTEM):

```
GET(adr, var)     fetch value at address adr, assign it to var
PUT(adr, val)     assign value val at address adr
BIT(adr, n)       bit n at address adr, a Boolean function
```

Typically, *adr* specifies the address of a "memory mapped" location belonging to a device interface. As an example, the following two (fast leaf) procedures receive and send a byte to a serial line interface at address UART = 3000020H:

```
PROCEDURE** receive(): INTEGER;
    VAR x: INTEGER*;
BEGIN
    REPEAT UNTIL BIT(UART+4, 0);  (*ready to receive*)
    GET(UART, x); RETURN x MOD 100H
END receive;
```

```
 4 E59F 1018   LDR    R1, [PC+24]        R1 := 3000024H
 5 E591 2000   LDR    R2, [R1]
 6 E1B0 20E2   MOVR   R2, R2 ROR 1       bit 0 to sign
 7 5AFF FFFB   BPL    –5                 branch to 4
 8 E59F 100C   LDR    R1, [PC+12]        R1 := 3000020H
```

```
 9 E591 0000    LDR    R0, [R1]            get
10 E200 00FF    ANDI   R0, R0, 0FFH        MOD 100H
11 E1A0 F00E    MOVR   PC := R14           return
12 0300 0024
13 0300 0020
```

```
PROCEDURE** send(x: INTEGER);
BEGIN
    REPEAT UNTIL ~BIT(UART+4, 1);  (*ready to send*)
    PUT(UART, x)
END send;
```

```
14 E59F 1014    LDR    R1, [PC+20]         R1 := 3000024H
15 E591 2000    LDR    R2, [R1]
16 E1B0 2162    MOVR   R2, R2 ROR 2        bit 1 to sign
17 4AFF FFFB    BMI    -5                  branch to 14
18 E59F 1008    LDR    R1, [PC+8]          R1 := 3000020H
19 E581 0000    STR    R0, [R1]            put x
20 E1A0 F00E    MOVR   PC := R14           return
21 0300 0024
22 0300 0020
```

An interrupt handler is a parameterless procedure that is called when the processor receives an interrupt signal (or performs a trap). Its entry address is stored in a fixed location as specified by the processor. In Oberon–SA, a procedure is specified to be an interrupt handler by a mark in its heading. The mark must specifiy a return offset as specified by the processor for every interrupt source. In the following example, we assume the presence of a global variable x which is incremented by each interrupt:

```
PROCEDURE [4] IRQ;
BEGIN x := x+1
END IRQ;
```

```
 0 E92D 5FFF    STM    PC, [R0 .. R12, R14]   push registers onto stack
 1 E28D C030    ADDI   FP, SP, 48             FP := SP+48
 2 E51F 0014    LDR    R0, [PC-20]            R0 := x
 3 E290 0001    ADDI   R0, R0, 1              +1
 4 E50F 001C    STR    R0, [PC-28]            x := R0
 5 E24C D030    SUBI   SP, FP, 48             SP := FP-48
 6 E8BD 5FFF    LDM    SP, [R0 .. R12, R14]   pop registers from stack
 7 E25E F004    SUBS   PC, R14, 4             return, restore PC and PSR
```

A special case is the fast interrupt request, indicated by the mark value 20. In the fast interrupt mode, entered after receiving such a request, the processor features a new set of registers R8 – R14. The compiler automatically allocates local variables in registers R8, R9, R10, and R11. This makes it unnecessary to save and restore registers, and lets the interrupt latency drop to nearly zero.

```
PROCEDURE [20] FIQ;
BEGIN x := x+1
END FIQ;
```

```
 8 E51F 802C    LDR    R8, [PC-44]            x,  no registers saved
 9 E298 8001    ADDI   R8, R8, 1              +1
10 E50F 8034    STR    R8, [PC-52]            x
11 E25E F004    SUBS   PC, R14, 4             return, restore PC and PSR
```

Access to special processor registers is made possible by a set of additional predeclared procedures which generate inline code, typically a single instruction. These procedures are:

```
PROCEDURE LDPSR(s, x)       load processor status register   s = code, x = value
PROCEDURE STPSR(s, v)       store processor status register  s = code, v = variable
PROCEDURE LDCFG(n, x)       load configuration               n = register no.
PROCEDURE STCFG(n, v)       store configuration
```

PROCEDURE FLUSH	flush caches and write buffer

15. Implementation of Floating–Point Operations

As mentioned in Section 8, due to lack of floating–point instructons, arithmetic operations on data of type REAL are performed by a set of procedures contained in module SPL (Standard Program Library). It goes without saying that in this case utmost efficiency is mandatory. SPL in fact gave rise to the Oberon–SA facility of fast leaf procedures, and is intended to remain reserved for this particular application. In conjunction with a very small number of special inline procedures, it made it possible to satisfy our ambition to program this module in Oberon instead of taking resort to assembler code. Subsequently we wish to exhibit the module SPL and thereby show that indeed very efficient and compact implementation of floating–point is possible entirely within Oberon–SA. The following 6 procedures compile into 248 instructions only. All parameters and variables are held in registers.

$$x = (-1)^s * 2^{e-127} * 1.m$$

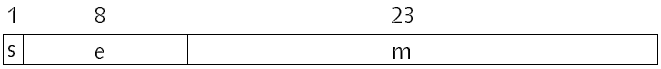

Fig. 8. Floating–point format

The mentioned special inline procedures are:

ADDC(x, y, z)	x := y + z + carry	x, y, z must be register variables
MULD(x, y, z)	<x, x'> := y * z	x + Rn, x' = R[n+1]
XOR(x, y)	x XOR y	used for sign bits
NULL(x)	x = 0	with sign bit cleared

```
MODULE SPL;  (*NW 25.11.98 / 28.2.99, Flt. Pt. for SA*)
   CONST B = 127; C = 800000H; E = 100H; S = 80000000H;

    PROCEDURE** Sum*(x, y: INTEGER);  (*1*)
        VAR xe, ye, s: INTEGER*;
    BEGIN
        IF NULL(x) THEN x := y
        ELSIF ~NULL(y) THEN
            xe := x DIV C MOD E - B;  (*extract exponent xe and mantissa x*)
            IF x >= 0 THEN x := (x MOD C + C)*2 ELSE x := -(x MOD C + C)*2 END ;
            ye := y DIV C MOD E - B;  (*extract exponent ye and mantissa y*)
            IF y >= 0 THEN y := (y MOD C + C)*2 ELSE y := -(y MOD C + C)*2 END ;
            IF xe < ye THEN
                ye := ye - xe; xe := xe + ye; (*denormalize x*)
                IF ye <= 25 THEN x := ASR(x, ye) ELSE x := 0 END
            ELSIF ye < xe THEN
                ye := xe - ye;  (*denormalize y*)
                IF ye <= 25 THEN y := ASR(y, ye) ELSE y := 0 END
            END ;
            s := x + y; x := ABS(s);
            IF x # 0 THEN  (*normalize x*)
                IF x >= 4*C THEN x := (x+2) DIV 4; INC(xe)
                ELSIF x >= 2*C THEN x := (x+1) DIV 2
                ELSE xe := xe-1;
                   WHILE x < C DO x := 2*x; xe := xe-1 END
                END ;
                IF xe < -B THEN x := 0  (*underflow*)
                ELSIF xe > B THEN x := 7FFFFFFFH  (*overflow*)
                ELSE x := (xe + B)*C - C + x
                END ;
                IF s < 0 THEN INC(x, S) END
            END
        END
    END Sum;
```

```
PROCEDURE** Product*(x, y: INTEGER);  (*3*)
    VAR xe, ye, zh, s: INTEGER*;
BEGIN
    IF NULL(y) THEN x := 0
    ELSIF ~NULL(x) THEN
        s := XOR(x, y);
        xe := x DIV C MOD E - B; x := (x MOD C + C) * 20H;
        ye := y DIV C MOD E - B; y := (y MOD C + C) * 20H;
        xe := xe + ye; MULD(ye, x, y);  (*ye is lower part of z*)
        IF zh >= 4*C THEN x := (zh+2) DIV 4; xe := xe+1 ELSE x := (zh+1) DIV 2 END ;
        IF xe > B THEN x := 7FFFFFFFH  (*overflow*)
        ELSIF xe < -B THEN x := 0  (*underflow*)
        ELSE x := (xe + B)*C - C + x
        END ;
        IF s < 0 THEN INC(x, S) END
    END
END Product;

PROCEDURE** Quotient*(x, y: INTEGER);  (*5*)
    VAR xe, ye, q, s: INTEGER*;
BEGIN
    IF ~NULL(x) THEN s := XOR(x, y);
        IF NULL(y) THEN x := 7FFFFFFFH
        ELSE
            xe := x DIV C MOD E - B; x := x MOD C + C;
            ye := y DIV C MOD E - B; y := y MOD C + C;
            xe := xe - ye;
            IF x < y THEN x := x*2; xe := xe - 1 END ;
            (*ye is counter*) ye := 25; q := 0;
            REPEAT q := 2*q;
               IF x >= y THEN x := x - y; INC(q) END ;
               x := 2*x; ye := ye - 1
            UNTIL ye = 0;
            q := (q+1) DIV 2;  (*round*)
            IF xe > B THEN x := 7FFFFFFFH  (*overflow*)
            ELSIF xe < -B THEN x := 0  (*underflow*)
            ELSE x := (xe + B)*C - C + q
            END
        END ;
        IF s < 0 THEN INC(x, S) END
    END
END Quotient;

PROCEDURE** Floor*(x: INTEGER);  (*7*)
    VAR e, n: INTEGER*;
BEGIN
    IF ~NULL(x) THEN
        e := x DIV C MOD E - (B+23); n := x MOD C + C;
        IF x < 0 THEN n := -n END ;
        IF e < -31 THEN x := 0
        ELSIF e < 0 THEN x := ASR(n, -e)  (*denormalize*)
        ELSE x := LSL(n, e)
        END
    END
END Floor;

PROCEDURE** Float*(x: INTEGER);  (*8*)
    VAR xe, s: INTEGER*;
BEGIN s := x;
    IF x # 0 THEN
        x := ABS(x); xe := 23;
        IF x >= 2*C THEN (*normalize*)
            REPEAT x := x DIV 2; INC(xe) UNTIL x < 2*C
        ELSIF x < C THEN
            REPEAT x := 2*x; DEC(xe) UNTIL x >= C
        END ;
        x := (xe + B)*C - C + x;
        IF s < 0 THEN INC(x, S) END
    END
```

```
    END Float;
END SPL.
```

16. Object– and Symbol–Files

Object files consist of the sequence of compiled instructions preceded by a header. Their syntax is given by:

```
CodeFile = modname key:4
           {name key:4 fix:2} 0
           {name entry:4} 0
           nofent:2 {entry:2}
           len:2 {codeword:4}.
```

The header starts with the module name (ASCII string terminated by a zero byte) and associated key (4 bytes). They are followed by a number of name/key/fix triples of the imported modules. This list is terminated by a zero byte. Then follows the sequence of command names and their entry addresses, also terminated by a zero byte. The last part of the header is the list of entry points (of exported procedures).

The linker/loader, after allocating memory and loading the instruction sequence, computes the addresses of calls to imported procedures. For each imported module, there exists a list of instructions referring to procedures of that module. The links of this list are in the address fields of the instructions themselves, and the address of the first instruction of the list is given by the *fix* value in the object file's list of imports. The actual offset is obtained from the number of the procedure to be called, used as index to the table of entry points of the respective module. To this value we add the difference between the origins of the present and the called module.

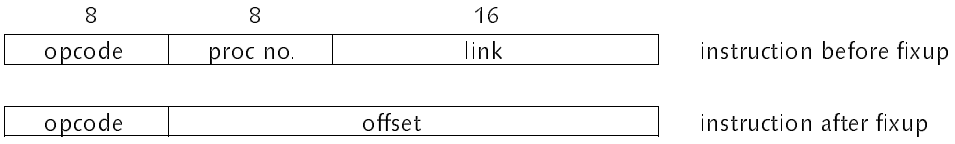

Fig. 9. Calls to external procedures

Because only procedures (and types), but not variables can be imported, this is the only fixup operation to be performed by the linking loader. The decision to exclude variable import was influenced by the consideration that variable export is contrary to the concept of abstract data types, but not less by the unavoidable inefficiency of external access caused by the short offset fields of load/and store instructions.

Symbol files are condensed and liearized forms of the compiler's symbol table. They contain the descriptions of the externally visible properties of all exported constants (including procedures) and data types. Their syntax is given by:

```
SymFile =   len:4 key:4 modname {object} 0.
object =    (CONST form value | TYP type) name.
param =     (VAR | PAR) chk type.
type =      ref [POINTER type
                | ARRAY type size
                | RECORD size {FLD type offset name} 0
                | PROC type {param} 0
                | TYP key modname typname].
```

The symbol file is generated by scanning the symbol table of the global scope. Only objects marked as exported are considered. An object's type is output before any other attribute of the object. Type descriptors are marked and numbered as soon as the descriptor is output to the symbol file. When the same type is referenced again, only its reference number is output, avoiding the writing of copies of the same data. Note that procedures are considered to be constants of a procedure type, the latter being characterized by the list of parameters and (type of) the result of the procedure. When reading a symbol file, the compiler collects the read types in an array. Reference numbers then act as indexes to this array.

Reflecting the recursiveness of the syntax, both input and output procedures are recursive too.

17. Conclusions

We argue that it is worthwhile to construct a compiler from scratch, if one strives for good, but possibly not optimal code. In designing an Oberon compiler for the ARM architecture, we have demonstrated that with the relatively little effort of 3 – 4 man months it is possible to construct a small and highly efficient compiler generating good code. The compiler program consists of 2350 lines of Oberon source text, or 32K bytes of code and 12K bytes of static data.

This degree of economy is achieved through straight–forward schemes of parsing and code generation, resulting in a single pass process. An important aid is the introduction of compiling hints in the source language, such as marking procedures as leaf procedures which leave parameters and local variables in registers. This information could be gathered by a more sophisticated compiler by a pre–pass over the source text. Another facility, with the purpose of accelerating sequential array access, is the rider.

An important advantage of designing one's own compiler is that specific, perhaps machine–dependent facilities can easily be incorporated. Such facilities are required if, for example, specific resources must be accessible, such as processor status register, memory management registers, or device interfaces. This is typically the case for embedded applications, which also require a predictable performance for real–time sensing and control.

Reference

1. N. Wirth. Compiler Construction. Addison–Wesley, 1995.