Code generation for a RISC Machine

Petr Kroha

1. Introduction

An important part of the whole development in computers comes from the idea of a new computer architecture based on processors with a simplified instruction set.To date, two prominent approaches to designing CPUs and their instruction sets have emerged - reduced instruction set computer (RISC) and complex instruction set computer (CISC) designs.The primary difference between these strategies is whether to use a small number of fast-executing primitive instructions or to use more complicated instructions that make writing programs easier.

From the first projects at IBM (IBM 801) and at Berkeley and Stanford [1] we can follow the development of the RISC idea to the current RISC machines. From the software point of view we can see the difference in number of machine instruction (CISC about 200, RISC about 30).Because of the simplicity of a hardware construction we can achieve the same performance much cheeper or we can realize the chip on GaAs technology and increase the speed of computing considerably.

In the following paragraphs we shall briefly show the structure of our RISC machine, general principles and interesting programming techniques used for engineering of a RISC compiler and using them for modifying of the UNIX C compiler for purposes of our RISC machine.

2. Structure of the used RISC machine

The detailed description of our RISC machine is given in [5].We shall present here only information needed for understanding of the following parts of our contribution.From the same reason we shall not describe the possibilities of parallelity and reconfigurability.

The most instructions of our RP-RISC machine (Reconfigurable Parallel RISC machine) have three operands. Every instruction is 32 bits long.The most significant 8 bits are used for the operation code, but only 5 bits really serve as code of operation, whereas the other 3 bits contain the information about the number of clock cycles needed for the instruction execution. The greatest number of clock periods is needed for CALL-instruction (6 cycles).Due to the fast execution of this instruction was used the windowing on a register field.The usually used stack concept for the CALL and RETURN instructions doesn't satisfy the condition of the LOAD-STORE access as the only access to the memory.Using procedures involves two groups of time-consuming operations: saving or restoring registers on each call and return, and passing parameters and results to and from the procedure.In the window concept each procedure call allocates a new "window" which is a subset of the large register field.This windows are overlapping for nested procedure calling(see Fig.1).

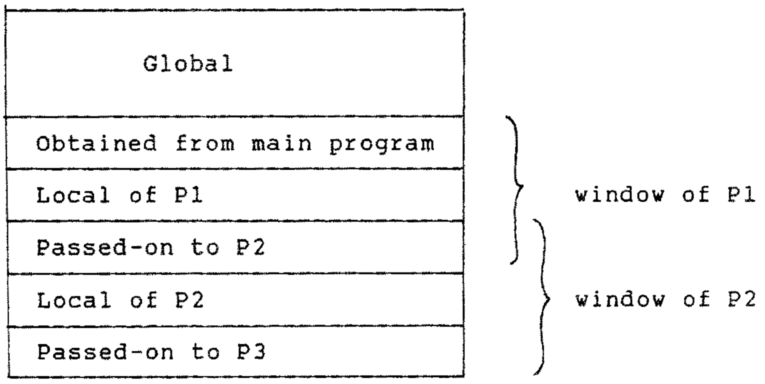

Fig. 1 Register windowing

The CALL and RETURN instructions are then only moving the window pointer on the field of registers.To the obvious programming techniques in RISC programs we can count the placing of the mostly used constants (e.g. zero, one) into the global registers.From the 32 global registers in our RP-RISC machine there are 12 registers used for saving constants and for some optimization purposes and 20 registers are used for purposes of evaluation of expressions as usual.The user can access in every moment only 56 registers - 32 global and 24 registers of the window.The total amount of registers is 288.The window pointer is incremented by CALL instruction and decremented by RET instruction. Each window has three parts:

- 8 registers for parameters obtained from previous subroutine
- 8 registers for local variables
- 8 registers for passed-on parameters.

Stack in memory and stack pointer are used automatically by hardware means only if subroutine nesting is too deep.

The relevant instructions are shown in the following table.

Machine instruction for the RP-RISC machine

Instruction		Function
ADD	R1,R2,R3	R1 + R2 --> R3
ADC	R1,R2,R3	R1 + R2 + C --> R3
SUB	R1,R2,R3	R1 - R2 --> R3
SBC	R1,R2,R3	R1 - R2 - C --> R3
AND	R1,R2,R3	R1 and R2 --> R3
OR	R1,R2,R3	R1 or R2 --> R3
XOR	R1,R2,R3	R1 xor R2 --> R3
SRL	R	shift right logical with carry
ICR	R,n,const	insert constant into nth cell of register R
ICA	addr. in page	insert 16-bit constant into the

		address register
ICP	number of page	insert 8-bit constant into the prefix register
LMR	[direct address]	load contents of given address into the register RP (one of the global registers).If no direct address is given, addressing is indirect,i.e.by means of address register
SMR	[direct address]	store contents of the register RP into the given address (as in LMR)
CJZ CJC CJN	[direct address]	conditional jump in accordance with flags ZERO,CARRY,NEGATIVE (if set), addressing identical with LMR, SMR
CALL		subroutine call
	[direct address]	addressing like LMR,SMR
RET		return from subroutine

Tab. 1 Instruction set of RP-RISC machine (the relevant subset)

3. Special programming techniques used in a RISC compiler

A RISC machine provides primitive hardware functions rather than bundling functionality into complex instruction, allowing the compiler to optimize below the level of other architectures.The compiler can generate optimal code for a simple machine more easily since the architecture provides fewer alternatives in performing a given function.This allows the compiler to focus its attention on other aspects of optimization, including global optimization and an efficient run-time environment.It must, of course, support a reduced instruction set by providing for function not present in the hardware.Each compiler phase usually performs optimizations.After local optimization will be done the peep-hole optimizations and architecture-dependent pipeline scheduling.

Procedure call, entry and return must be designed as fast and simple as possible, because there are often called the procedures and the functions of the run-time support standing for the complex instructions of CISC.It was discovered that 95% of the calls pass fewer than four parameters [3].

Using of pipelining improves performance by 20% on average [4], but brings problems because the next instruction will be started before the current instruction has completed its activity.We speak about data conflicts and branch conflicts and solve this by load delay and branch delay methods.A load delay occurs when one instruction loads a value into a register and the next instruction uses that register.It is necessary to change the instruction order so that the instruction following the load does not depend on the load.This permits one instruction to access memory in parallel while executing another instruction.An other effect in pipeline reorganization is called a delayed branch.When a branch instruction is executing the logical order and the physical order of instructions are different.The logically next instruction should be prepared but the physically next instruction is.The basic solution is to insert so many NOP-instruction how many are needed for elimination of the branch conflict.

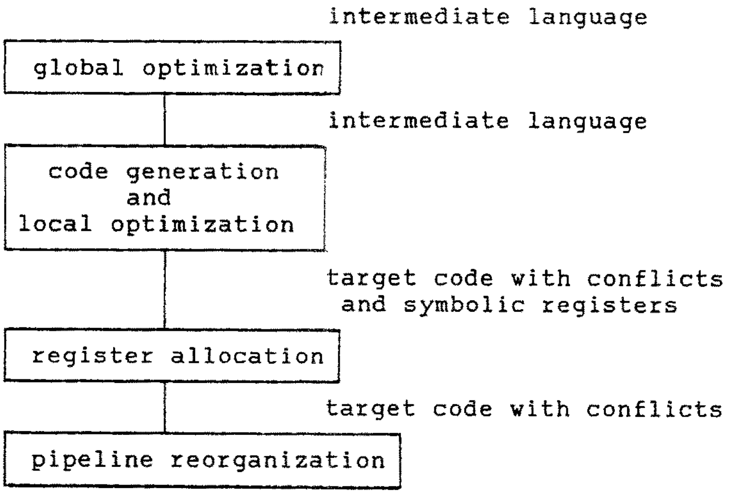
intermediate language
global optimization
intermediate language
code generation and local optimization
target code with conflicts and symbolic registers
register allocation
target code with conflicts
pipeline reorganization

Fig.2 Scheme of a back-end of a RISC compiler

The first phase of the code generation is the global optimization and substitution of small procedures by sequences of instructions.In the following process a naive code will be generated which uses unlimited set of registers and doesn't worry about data and branch conflicts caused by pipelining.During this process are complex instructions of the intermediate language substituted by sequences of the RISC primitive instructions and a local optimization is provided.Register allocation will be organized by help of a method for coloring graphs [2].The last phase of the code generator deletes data and branch conflicts.Some algorithms are given in [4].

4. Features of the UNIX C compiler

The version of the UNIX C compiler [6] we have got in the source form doesn't use a machine-independent intermediate language, but it is not important for our purposes.Each intermediate file is represented as a sequence of binary numbers without any explicit delimiters.It consists of a sequence of logical lines, each headed by an operator, and possibly containing various operands (numbers or strings).Expressions are transmitted in a reverse-Polish notation; as they are being read, a tree is built which is isomorphic to the tree constructed in the last phase.

The reader maintains a stack; each leaf of the expression tree (name, constant) is pushed on the stack;each unary operator replaces the top of the stack by a node whose operand is the old top-of -stack; each binary operator replaces the top pair on the stack with a single entry. When the expression is complete there is exactly one item on the stack.Following each expression is a special operator which passes the unique previous expression to the optimizer and to the code generator.

The optimizer tries to modify and simplify the tree so its value may be computed more efficiently and conveniently by the code generator.Each interior node will be marked with an estimation of the number of registers required to evaluate it.This register count is needed to guide the code generation algorithm.The common subexpression are not discovered and used for optimization.The basic algorithms the depth-first scan of the tree,but the code generation is in principle independent of any particular machine.It depends largely on a set of tables.It could seem to be easy to modify this tables for producing code for other machines, but:

- structure of these tables is machine-dependent
- it is non-trivial to prepare such tables.

The basic code generation procedure works with a pointer to a tree representing an expression, with the name of a code generation table, and returns the number of the register in which the value should be placed.There are four code generation tables:

- REGTAB actually produces code which causes the value represented by the expression tree to be placed in a register.
- CCTAB is used in the case, when we need not the value of the expression, but instead the value of the condition code resulting from evaluation of the expression.This table is used to evaluate the expression after IF.
- SPTAB is used when the value of an expression is to be pushed on the stack(e.g. an actual argument).
- EFFTAB is used when an expression is to be evaluated for its side effects, not its value.This occurs mostly for expressions which are statements, which have no value.It doesn't need to leave the value in a register.

All of the tables besides REGTAB handle a relatively few special cases.If one of these tables does not contain an entry applicable to the given expression tree, the table REGTAB will be used and perhaps a redundant code will be generated.Parts of the items in the REGTAB can be macros which are expanded during the generation.

The processing of register allocation during the code generation is based on Sethi-Ullman algorithm.

The main code generation tables consist of entries each containing an operator number and a pointer to a subtable for the corresponding operator.A subtable consists of a sequence of entries, each with a key describing certain properties of the operands of the operator involved;associated with the key is a code string.Once the subtable corresponding to the operator is found, the subtable is searched linearly until a key is found such that the properties demanded by the key are compatible with the operands of the tree node.A successful match returns the code string; an unsuccessful search returns a failure indication.

5. Modification for the RISC target code

Because the global optimization is running on the program in the intermediate language it wasn't changed.This means the first we have to change is the target code to be generated.The main part of it is generated by help of the table REGTAB.The table CCTAB isn't relevant for our purposes because we have the operands evaluated in registers for testing anyway (the LOAD-STORE access to the memory).The SPTABLE isn't relevant too because we have no instructions working with stack.The table EFFTAB we needn't for the same reason as the table CCTAB, because we always obtain a value in a register.

The REGTAB table is organized hierarchically.The first level is built on the operator of the tree node representing the operation to be translated (has about 50 entries).The second level is built on the types of operands in the compile stack (total about 120 entries).Because of many possibilities of addressing in the PDP 11 instruction set there are many cases to distinguish.For the PDP 11 code will be the operator of indirection translated not immediately.Its operand in the compile stack will be only signed and the operation of indirection will be generated in the most cases by help of possibilities of the addressing modes of the PDP 11.In operands in the REGTAB for the PDP 11 is distinguished between operand *X and *(X+c) because of addressing modes (R) and X(R).For the RP-RISC machine we have only one way, how to translate the indirect addressing.It simplifies the tables because the number of cases is therefor considerable smaller (about 4 times).

The power of expression of the RP-RISC code is illustrated in the following table.

Instruction set of PDP 11			Instruction set of RP-RISC	
CLR	R	(clear)	ADD	RO,RO,R
COM	R	(complement)	SUB	RO,R,R
INC	R	(increment)	ADD	R1,R,R
DEC	R	(decrement)	ADD	R-1,R,R
NEG	R	(negate)	XOR	R,R-1,R
ASR	R	(arith. shift right)	SRA	R
ASL	R	(arith. shift left)	ADD	R,R,R
MOV	Ra,Rb	(Rb <- Ra)	ADD	RO,Ra,Rb
ADD	Ra,Rb	(Rb <- Ra + Rb)	ADD	Ra,Rb,Rb
SUB	Ra,Rb	(Rb <- Rb - Ra)	SUB	Rb,Ra,Rb
BIS	Ra,Rb	(Rb <- Ra or Rb)	OR	Ra,Rb,Rb
BIC	Ra,Rb	(Rb <- Ra and Rb)	SUB	RO,Ra,RP
			AND	RP,Rb,Rb
BIT	Ra,Rb	(Ra and Rb)	AND	Ra,Rb,RP
TST	R	(R ? O)	SUB	R,RO,RP
CMP	Ra,Rb	(Ra ? Rb)	SUB	Ra,Rb,RP
JMP	address	(jump)	ADD	RO,RO,RP
			CJZ	address
JSR	PC,address	(jump to subr.)	CALL	address
RTS	PC	(ret. from subr.)	RET	

Tab. 2 Comparison of some instruction of PDP 11 and RP-RISC.

To have a scale for comparing the branching of tables we will not take in account the operations in the float and double arithmetic. Than we become the following table.

Operator	Number of distinguished cases of operands	
	PDP 11	RP-RISC
JUMP	2	1
CALL	3	1
LOAD	6	2
ASSIGN	7	1
PLUS	12	3

Tab. 3 Number of cases of operands to be distinguished

The structure of generated code we can see on the following example compared with the code for PDP 11.
First of all we shall show a function without a local field.

Example 1:

In C language:

```
   getnum (ap)
     char *ap;
     {
     register char *p;
     register int  n,c;
       p=ap;
       n=0;
       while ((c=*p++) >= '0' && c<= '9')
             n = n*10 + c - '0';
       if ( *--p !=0 )
             return(o);
       return(n);
     }
```

In code for PDP 11:

```
_getnum:
        jsr     r5,csv
        mov     4(r5),r4
        clr     r3
.3:
        movb    (r4)+,r2
        cmp     r2,#60
        blt     .2
        cmp     r2,#71
        bgt     .2
        mov     r3,r0
        asl     r0
        asl     r0
        add     r3,r0
        asl     r0
        add     r2,r0
        sub     #60,r0
        mov     r0,r3
        br      .3
.2:
        tstb    -(r4)
        beq     .4
        clr     r0
        br      .1
.4:
        mov     r3,r0
.1:
        jmp     cret
```

In code for RP-RISC:

```
        ICR     R43,0,48
        ICR     R44,0,57
        ADD     R32,R0,R40          ... p=ap
        ADD     R0,R0,R41           ... n=0
L3:
        ADD     R40,R0,RA
        LMR
        ADD     RP,R0,R42           ... c=*p++
        SUB     R42,R43,RP
        CJN     L2
        SUB     R44,R42,RP
        CJN     L2
        ADD     R41,R0,R20
        ADD     R20,R20,R20
        ADD     R20,R20,R20
        ADD     R41,R20,R20
        ADD     R20,R20,R20         ... 10 * n -> R20
        ADD     R42,R20,R20
        SUB     R20,R43,R20
        ADD     R20,R0,R41
        SUB     R0,R0,R0
        CJZ     L3
L2:
        SUB     R40,R1,R40
        ADD     R40,R0,RA
        LMR
        SUB     RP,R0,RP
        CJZ     L4
        ADD     R0,R0,R40
        SUB     R0,R0,R0
        CJZ     L1
L4:
        ADD     R41,R0,R40
L1:
        RET
```

Example 2:

Using of a local field presumes to have a run-time support function for location of the field in memory and storing its address in the local variable with the same order as the local field has.

In C language:

```
int a[100];
move ()
{
  int i,b[50],j;
    j=5;
    for ( i=0;i<50;i++)
        b[i]=a[i*2]+1;
    i=0;
    }
```

For PDP 11 we obtain:

```
_move:
       jsr    r5,csv
       sub    #146,sp
       mov    #5,-156(r5)
       clr    -10(r5)
       br     .4
.5:
       mov    -10(r5),r4
       asl    r4
       asl    r4
       add    #_a,r4
       mov    (r4),r4
       inc    r4
       mov    -10(r5),r0
       asl    r0
       add    r5,r0
       mov    r4,-154(r0)
.3:
       inc    -10(r5)
.4:
       cmp    -10(r5),#62
       blt    .5
.2:
       clr    -10(r5)
.1:
       jmp    cret
```

For RP-RISC we obtain:

```
      ICR     R10,0,50
      ADD     R10,R0,RP
      CALL    Loc-field
      ADD     RP,R0,R41
      ICR     R42,0,5            ....j=5
      ADD     R0,R0,R40          ....i=0
      SUB     R0,R0,R0               jmp L1
      CJZ     L1
L2:   ADD     R40,R40,R43        ....i * 2 -> R43
      ICA     _A
      ADD     RA,R43,RA          ....A[i*2]
      LMR                        ....A[i*2] -> RP
      ADD     R1,RP,RP           ....A[i*2] + 1 -> RP
      ADD     R41,R0,RA
      ADD     R40,RA,RA          ....B[i]
      SMR                        ....RP -> B[i]
      ADD     R1,R40,R40         ....i=i+1
L1:   SUB     R40,R10,RP         ....i < 50
      CJN     L2
```

We can see that we obtained:
19 instructions a 4 byte for RP-RISC ... 76 byte
18 instruction of variable length for PDP 11 ... 60 byte.
If we make a summa of the time needed for execution of instructions in one sequence we obtain:
-for RP-RISC: 20 microsec. (about 100 cycles for 5 MHz clock)
-for PDP 11 : 71.94 microsec.
The tables are not the only one place, where the machine dependent code occurs.For example for a conditional branch there is in the UNIX C compiler a procedure:

```
   branch (label,aop,c)
   {
    register op;
     if (op=aop) PRINS(op,c,branchtab);
           else  printf("jbr");
     printf('\tL%d\n',label);
   }
```

The dependent code occurs on about 70 places.The number of registers used for evaluation is in the UNIX C compiler defined as 3, we extended to 20 (only through changing a constant).

6. Conclusions

The algorithms of the UNIX C compiler was used to prove some properties of a Small C compiler for a RISC machine.Because of lack of the most addressing modes many tables and many procedures were becoming much more simple.The problem of data and branch conflicts we hadn't to solve because our machine doesn't use pipelining.This is due to the motivation of the machine in control of parallel processes, where the greatest part of the processor activity can be spent by waiting for an asynchronous event.That's why it didn't seem to be important to use pipelining.On examples we showed that the increase of number of instructions of the generated code when compiling a program isn't so considerable as will be usually proposed.

References

[1] Patterson,D.A.,Sequin,C.: A VLSI RISC. Computer,15,No 9,1982.
[2] Chaitin,G.J.: Register Allocation and spilling via graph coloring.Proceedings of the SIGPLAN'82 Symposium of Compiler Construction, SIGPLAN Not. 17, 1982.
[3] Chow,F.: Engineering RISC compiler system. Compcon Spring 1983.
[4] Gross,T.R.: Code optimization technique for pipelined architecture. Compcon Spring 1983.
[5] Blazek.Z.,Kroha,P.: Design of a reconfigurable parallel RISC machine. EUROMICRO'87,Portsmouth. reprinted in: Microprocessing and microprogramming 21(1987), North-Holland.
[6] Ritchie,D.M.: A Tour through the UNIX C Compiler. Bell Laboratories,documentation of UNIX.

[7] Blazek,Z.:A reconfigurable processor with RISC features. Thesis,Czech Technical University, (to appear on Czech)
[8] PDP 11 Processor Handbook.Digital Equipment Corporation,1978.