A Code Generator to translate Three-Address intermediate code to MIPS assembly code

Pandaba Pradhan
Department of Computer Science & Engineering
Institute of Technical Education & Research, SOA University, Odisha, India

Abstract- Compiler is a translator, which translates the source code of a program to the corresponding object code of the same program. So, if the compiler can translate a source program to it's machine code easily and quickly, then the performance of the system will be high. Hence we can increase the performance of a system by designing a compiler, which can produce the target code easily. Here the consideration will be on how to translate the three-address codes generated by the compiler into conventional machine code. Many different machine instruction sets exist; one for each computer architecture. Examples include the Intel x86 architecture, the SPARC, the Alpha, the Power PC, and the MIPS. In this paper the target architecture used is MIPS architecture. The MIPS architecture is similar to RISC. It is a load-store architecture that is arithmetic instructions operate only on registers. It has 32 general-purpose registers of 32 bits each. Since, the MIPS access the memory in load and store operations, so it's instruction execution time is less. Therefore, it's performance is high. Three address codes are the intermediate representations of the source language. These are independent of the target system. It is easy to understand and write them. In this paper a code generator is described, which is used to translate the three address codes to MIPS assembly codes.

Keywords – Translator, Compiler, Source code, Machine code, Performance, MIPS, Code Generator

1. INTRODUCTION

A Compiler [1, 2, 15] is a translator from one language, the input or source language, to another language, the output or target language. The target language is an assembly language or the machine language for a computer processor. The compiler requires a two-step process to run a program; Step-1: Execute the compiler (and possibly an assembler) to translate the source program into a machine language program.Step-2: Execute the resulting machine language program, supplying appropriate inputs. Modern compilers contain two (large) parts, each of which is often subdivided. These two parts are the front end and the back end. The front end analyzes the source program, determines its constituent parts, and constructs an intermediate representation of the program. Typically the front end is independent of the target language. The back end synthesizes the target program [10, 11] from the intermediate representation produced by the front end. Typically the back end is independent of the source language.

The primary objective of the code generator is to convert atoms or syntax trees to instructions. The internal form is translated by the code generator [3-5] into object code. Typically, the object code is a program for a virtual machine. The virtual machine consists of three segments: a data segment, a code segment and an expression stack. The data segment contains the values associated with the variables. Each variable is assigned to a location, which holds the associated value. Thus, part of the activity of code generation is to associate an address with each variable. The code segment consists of a sequence of operations. The expression stack contains the expression to be evaluated.

The rest of the paper is organized as follows. MIPS Architecture [16] and its instructions are explained in section II. Three-Address codes and their types are explained in section III. Proposed code generation algorithm is explained in section IV. Experimental results are presented in section V. Concluding remarks are given in section VI.

II. MIPS ARCHITECTURE

MIPS [17] stand for the Microprocessor without Interlocked Pipeline Stages. It has a design, based on the RISC microprocessor architecture. It was developed by MIPS Computer Systems Inc.It was the revolutionary findings of John L.Hennessy, to overcome the problems in CISC architecture. MIPS processors are used widely in embedded systems.

A. MIPS R2000 PROCESSOR----

A MIPS processor consists of an integer-processing unit (the CPU) and a collection of coprocessors that perform ancillary tasks or operate on other types of data such as floating-point numbers (Shown in Figure-1). SPIM simulates two coprocessors. Coprocessor 0 handles exceptions and interrupts. Coprocessor 1 is the floating-point unit. SPIM simulates most aspects of this unit.

B. Addressing Modes---

MIPS is a load-store architecture, which means that only load and store instructions access memory. Computation instructions operate only on values in registers. The bare machine provides only one memory-addressing mode: c(rx), which uses the sum of the immediate c and register rx as the address. The virtual machine provides the following addressing modes for load and store instructions shown in the Table-1:

Table: 1

Format	Address computation
(register)	contents of register
imm	immediate
imm (register)	immediate + contents of register
label	address of label
label ± imm	address of label + or – immediate
label ± imm (register)	address of label + or – (immediate + contents of register)

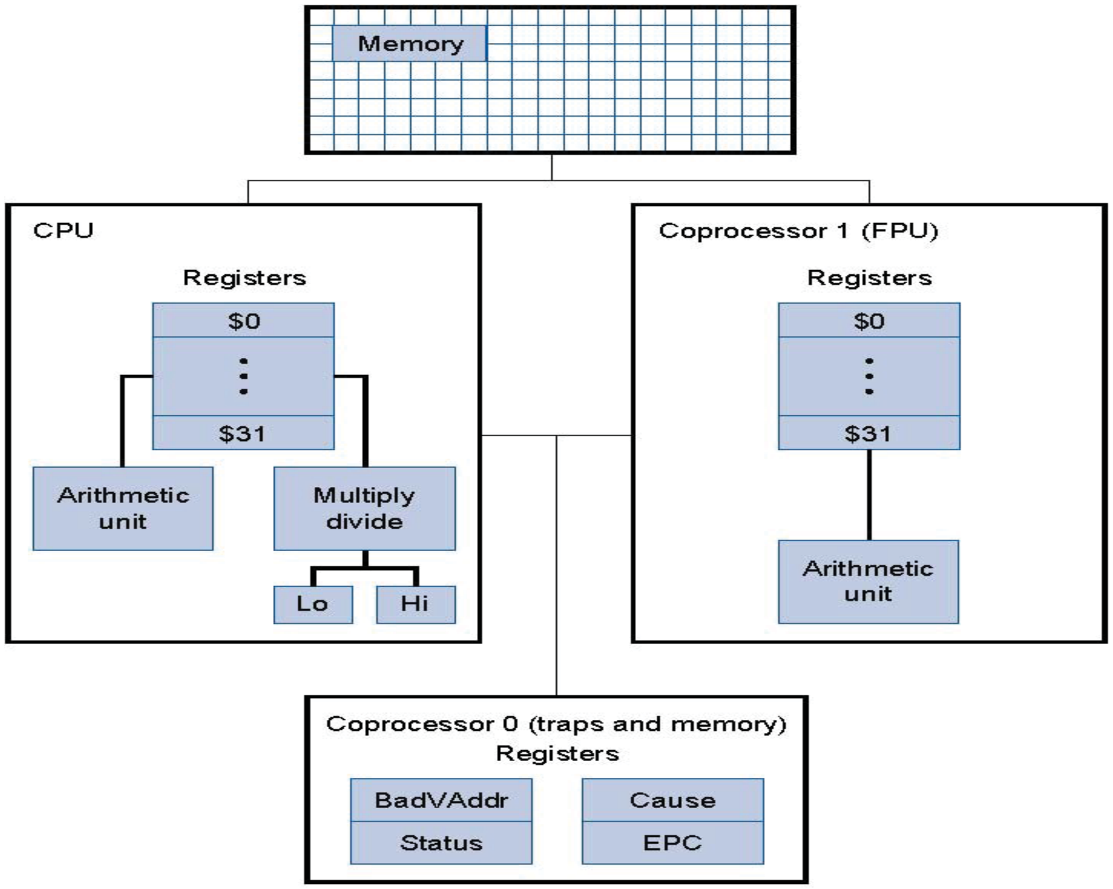

Figure-1 : MIPS R2000 CPU and FPU.

C. Assembler Syntax---

Comments in assembler files begin with a sharp sign (#). Everything from the sharp sign to the end of the line is ignored. Identifiers are a sequence of alphanumeric characters, underbars (_), and dots (.) that do not begin with a number. Instruction opcodes are reserved words that cannot be used as identifiers. Labels are declared by putting them at the beginning of a line followed by a colon, for example:

.align n	Align the next datum on a 2n byte boundary. For example, .align 2 aligns the next value on a word boundary. .align 0 turns off automatic alignment of .half, .word, .float, and .double directives until the next .data or .kdata directive.
.ascii str	Store the string str in memory, but do not null-terminate it.
.asciiz str	Store the string str in memory and null-terminate it.

D. MIPS Instruction Set---

MIPS instruction set contains all instructions, their meanings, syntax, semantics, and bit encodings. The syntax given for each instruction refers to the assembly language syntax supported by the MIPS assembler. Hyphens in the encoding indicate "don't care" bits, which are not considered when an instruction is being decoded. General-purpose registers (GPRs) are indicated with a dollar sign ($). The words SWORD and UWORD refer to 32-bit signed and 32-bit unsigned data types, respectively.

III. THREE-ADDRESS CODES

Three-address code (TAC or 3AC) [6] is a form of representing intermediate codes. It is used by the compilers to generate the target code for any system.

- Like assembly language:
  Can have symbolic labels, representing statement's address
  Labels are translated to indices as in an assembler
  Support for flow of control (conditionals, jumps)
- Sequence of statements of form
  x := y op z

where x, y, z are names, constants, or compiler-generated
temporaries (t1, t2, etc.) and op is any operator (arithmetic: +, * , logical and, or)
If w := x + y * z
Then the three address codes are:
t1 := y * z
t2 := x + t1
w := t2

A. Types of Three-Address Codes------

The TAC's are of different types such as:
• Dyadic-op assignment: x := y op z (where op = +, and)
• Copy: x := y
• Monadic-op assignment: x := op z (op = -, not)
• Unconditional jump: goto L
• Conditional jump: if x relop y goto L (op = <, =)
• Parameter-passing procedure calls:

```
param x1
param x2
...
param xn
call p, n
```

IV. Proposed Code Generation Algorithm

The most prominent aspect of building a back end of a compiler is the construction of a code generator [7-9]. Here the code generator is able to map a sequence of three address codes to a program of the target language. The generated code is written to one or more files. These files can be compiled with the aid of a compiler for the target language.

The first challenge in code generation is instruction selection [6]. Instruction selection is highly target machine dependent. It involves choosing a particular instruction sequence to implement a three-address code. Even for a simple three-address code there may exist a choice of possible implementations. Besides instruction selection, there are challenges for register allocation and code scheduling. Register allocation aims to use registers effectively by minimizing register spilling (storing a value held in a register and reloading the register with something else). Even a few unnecessary loads and stores can significantly reduce the speed of an instruction sequence. Assembly code files should end with the suffix '.asm' or '.s'. The SPIM simulator reads in assembly files, so there is no need to translate to machine code.

A. *Code Generation Algorithm*-----

The code generation algorithm has the following segments:

- Data and Text Segments

  A set of data declarations must be preceded by the line ".data"

  A section of code (assembly instructions) must be preceded by ".text"

- Identifiers and Labels

  A global identifier id in the source program will translate to the same identifier id in the assembly code generated. Local variables will not map to identifiers, but will be accessed via displacements off the frame pointer. In order to print string constants, it will be necessary to first declare the string, then access that string using a label generated for it. A label is simply an identifier.

B. ***Register Allocation*----**

An essential component of any code generator is its register allocator [12-14]. Machine registers must be assigned to program variables and expressions. Since registers are limited in number, they must be reclaimed (reused) throughout a program. A register allocator may be a simple "on the fly" algorithm that assigns and reclaims registers as code is generated. The procedure for register allocation is:

Register Allocation(On The Fly Algorithm)

```
  Goal is to assign each variable to one of k registers.
In general, it is an NP-complete problem. So we use a greedy heuristic:
 •Build interference graph G
     –Use liveness analysis
          –G(x,y)=true if x & y are live at same point.
 •Simplify the graph G
   –If x has degree < k, push x and simplify G-{x}
   –If no such x, then we need to spill some node.
•Once graph is empty, start popping nodes and assigning
        them registers.
     –Always have a free register since sub-graph G-{x} can't
        have >= k interfering nodes.
```

V. Experiment and Result

The SPIM simulator[17] is used to simulate the below program.

1./*program to translate three-address code to MIPS code*/

```
#include<stdio.h>
#include<string.h>
#include<conio.h>
```

```
#include<ctype.h>

void main()
{
char a[20];
int i,j=0;
FILE *fp;
if(( fp = fopen("z.dat", "r")) != 0)
 {
 while(fscanf(fp,"%s", a) != EOF)
 {
 i=0;
 if( strlen(a)==6)
 {
 i=i+3;
 if(islower(a[i]))
  printf("lw $t%d, (%c)\n", j++,a[i]);
 else
{
        for(i=3;i < strlen(a);i++)
                {

                if(isdigit(a[i]))
                {
                x= a[i] - '0';
                k=k*10 +x;
                }
                }
        printf("li $t%d, %d\n", j,k);
}


i=i+2;
if(islower( a[i]))
        printf("lw $t%d, (%c)\n", j++,a[i]);
 else
        {
        for(i=3;i < strlen(a);i++)
                {

                if(isdigit(a[i]))
                {
                x= a[i] - '0';
                k=k*10 +x;
                }
                }
        printf("li $t%d, %d\n", j,k);
}


        i=i-1;
        if(a[i] == '+')
        printf("add $t%d, $t%d, $t%d\n", j,j-1,j-2);
        else if( a[i] == '-')
        printf("sub $t%d, $t%d, $t%d\n", j,j-2,j-1);
        else if( a[i] == '*')
 printf("mul $t%d, $t%d, $t%d\n", j,j-2,j-1);
        else if( a[i] == '/')
        printf("div $t%d, $t%d, $t%d\n", j,j-2,j-1);

        }
        else if(strlen(a)==4) {
        i=i+3;
        if( islower(a[i]))
{
        printf("lw $t%d, %c\n",j,a[i]);

        printf("copy %c, $t%d\n", a[i-3],j);
}
        else
         printf("li $t%d, %c\n",j,a[i]);
                                }
                              j=j+1;
                                }
                }
  }
```

INPUT:

```
p:=3
a:=b+c
x:=y
```

OUTPUT:

```
li $t0, 3

lw $t1, (b)
lw  $t2,  (c )
add  $t3, $t2, $t1

lw  $t0, (y)
move x,$t0
```

VI.CONCLUSION

In this paper, the MIPS assembly codes are generated from three-address codes. First, we worked with LEX and YACC tools to generate the three address codes of a given source program (Both C & C++). Then we have designed a code generator to translate the three address codes to MIPS codes. While generating the MIPS assembly codes, we have taken only arithmetic instructions to make the design of the code generator simple. So, the code generator generates MIPS assembly codes from three-address codes.

Finally, The SPIM simulator runs the assembly programs of MIPS directly. Hence, it avoids the translation of assembly code to machine code. The so designed code generator can be enhanced by adding all other instructions such as branch instructions.

REFERENCES

[1] AHO, A. V, “Translator writing systems: Where do they now stand?”, IEEE Cornput. 23,8 (1980),9-14.

[2] Andrew W. Appel, " Modern Compiler Implementation in C," Cambridge University Press, ISBN-0-521-60765-5.

[3] John R. Levine, Tony Mason, Doug Brown, " Lex and Yacc.O’Reilly.

[4] MAHADEVAN GANAPATHI, CHARLES N. FISCHER, " Affix Grammar Driven Code Generation ", In ACM Transactions on Programming Languages and Systems, Vol. 7, No. 4, October 1985, Pages 560-599.

[5] AHO, A. V.,GANAPATHI," Efficient tree pattern matching: An aid to code generation ", In Conference Record of the 12th Annual ACM Symposium on Principles of Programming Languages(New Orleans, La., Jan. 1985), ACM, New York .

[6] Saumya Debray,Notes on Translating Three-Address Code to MIPS Assembly Code.

[7] Frank Budinsky, Marilyn Finnie, Patsy Yu, Automatic Code Generation from Design Patterns. T.J. Watson Research Center.

[8] Susan L. Graham, Table-driven code generation. Computer, 17(8), August 1980.

[9] CHAITIN, G. J, Register allocation and spilling via graph coloring In Proceedings of the SZGPLANBZ Symposium on Compier Construction (Boston, Mass., June 23-25, 1982). ACM SZGPLAN Not. 17,6 (June 1982).

[10] ALFRED V. AHO, MAHADEVAN GANAPATHI and STEVEN W. K. TJIANG. ACM Transactions on Programming Languages and Systems. Vol. 11, No. 4, October 1989, Pages 491-516.

[11] Frederick Smith, Dan Grossman, Greg Morrisett, Luke Hornof, and Trevor Jim. Compiling for Runtime Code Generation.

[12] K. Pettis and R. C. Hansen, ’’Profile Guided Code Positioning” , Proceedings of the ACM SIGPLAN ’90 Conference on Programming Language Design and Implementation, pp.16-27, June 1990.

[13] D. W. Wall, ’’Global Register Allocation at Link Time”, Proceedings of the SIGPLAN 1986 Symposium on Compiler Construction, June 1986.

[14] D. W. Wall, ’’Register Window vs. Register Allocation” , Proceedings of the SIGPLAN ’88 Conference on Programming Language Design and Implementation,June 1988.

[15] A.V. Aho, R. Sethi and J.D. Ullman, Compilers: Principles, Techniques, and Tools, Addison-Wesley Publishing Company.

[16] MIPS Assembly Language Programmer’s Guide.

[17] James Larus , “SPIM: A MIPS32 Simulator”.