Compiler Lab

Submit by 8th Feb 2010

Second Year Computing Laboratory
Department of Computing, Imperial College
Work as a group of 1-3 people
For cs2 and jmc2 students taking the compilers course

Summary

Construct a working compiler which can translate a small subset of C to mips assembler code. The source of the scanner and parser for the language is provided. You will write a code generator in Java that walks over the abstract syntax tree produced by the parser and produces assembly code for a MIPS risc machine.

Program structure

The provided main program (written in java)

1. Calls the scanner and parser which parses the program, checks all variables have been declared before use and constructs an abstract syntax tree (AST).
2. If parser does not find any syntax errors the code generator is called with the following information:
   - the AST (which does not contain declaration statements).
   - An opened java `PrintStream code`. The `print` or `println` methods of `code` can be used write to the stream assembler to the assembler file.

Provided files

Compiler.java	The main file of the compiler.
Pico.g	The grammar of the language pico, which is translated to provide the scanner and parser.
Gen.g	The source of the tree parser that you will modify to generate assembler code.
makefile	To compile the files and build the compiler.
compile	A shell script to run your compiler.
runtests	A shell script to run your compiler on the provided test files.

Input language pico

Pico is a small subset of C defined by:

1. A program must contain only one function definition, `main`.
2. Variables can only be declared as int.
3. Structs, unions, pointers,enum and arrays cannot be declared.
4. The only control statements are **if**, **while** and **break**.
5. Three functions are accepted `main`, `printf` and `scanf`. `printf` can only write ints (as well as the string in the first argument). `scanf` only reads ints.
6. The only preprocessor statement (starting with '#') that is recognised is '#include', and it is ignored. This allows you to include `stdio.h` in a program and compile it using a another C compiler.

The Grammar

The ebnf grammar below is generated from the file `Pico.g`, Terminals (keywords or special symbols are uppercase, the names of the rules are lowercase.

Grammar of Pico

[1]	program	→	(decl)* main_def
[2]	decl	→	INT init (COMMA init)* SEMICOLON
[3]	main_def	→	INT MAIN LPAREN RPAREN block
[4]	block	→	LBRACKET (decl)+ (statement)* RBRACKET | LBRACKET (statement)* RBRACKET
[5]	statement	→	SEMICOLON | block | assignment | if_stmt | while_loop | break_stmt | call
[6]	assignment	→	IDENT ASSIGN expr SEMICOLON
[7]	if_stmt	→	IF LPAREN expr RPAREN statement (ELSE statement | EMPTY)
[8]	while_loop	→	WHILE LPAREN expr RPAREN statement
[9]	break_stmt	→	BREAK SEMICOLON
[10]	call	→	printf | scanf
[11]	printf	→	PRINTF LPAREN STRING (COMMA expr)* RPAREN SEMICOLON
[12]	scanf	→	SCANF LPAREN STRING (COMMA address)+ RPAREN SEMICOLON
[13]	init	→	IDENT ASSIGN expr | IDENT
[14]	expr	→	and_expr (OR and_expr)*
[15]	address	→	AMP IDENT
[16]	and_expr	→	bit_or (AND bit_or)*
[17]	bit_or	→	bit_xor (BITOR bit_xor)*
[18]	bit_xor	→	bit_and (BITXOR bit_and)*
[19]	bit_and	→	equal_expr (AMP equal_expr)*
[20]	equal_expr	→	comp_expr ((EQUAL | NOT_EQUAL) comp_expr)*
[21]	comp_expr	→	shift_expr ((LESS | GREATER | GREATER_EQUAL | LESS_EQUAL) shift_expr)*
[22]	shift_expr	→	add_expr ((SHIFT_LEFT | SHIFT_RIGHT) add_expr)*
[23]	add_expr	→	mul_expr ((PLUS | MINUS) mul_expr)*
[24]	mul_expr	→	sign_expr ((TIMES | DIVIDE | MOD) sign_expr)*
[25]	sign_expr	→	NOT primary_expr | MINUS primary_expr | BITNOT primary_expr | primary_expr
[26]	primary_expr	→	IDENT | NUMBER | LPAREN expr RPAREN

Target architecture

MIPS is a RISC : a load-store architecture with a lot of registers (32) and a regular instruction set (instructions are all one word long and there are a small number of address formats). Only load and store instructions access data memory, other instructions use a register as the destination and either two registers or one register and a constant as the source, for example to increase a variable in memory:

```
        .data
```

```
count:
      .word 41          # data word initialized to 41
      .text
start:
       lw $t0,count     # load word from count (memory) to register t0
       addi $t0,$t0,1   # add the immediate number 1 to register t0
       sw  $t0,count    # store the register t0 in word count
```

Summary of the MIPS simulator and assembler

Patterson and Hennessy's book 'Computer Organization and Design' describes the MIPS architecture in detail. The complete simulator is described in the online documentation for spim [http://www.doc.ic.ac.uk/lab/secondyear/spim].

The Assembler syntax

Comments start with a hash "#", identifiers are alphanumeric including "_" and ".", and cannot start with a digit. Labels must start at the beginning of the line and end in a colon ":" and not have same name as an instruction. Arguments for instructions and directives are separated by commas.

Registers

Table 1. MIPS User Registers

Description	Assembler Syntax
always 0	$zero
call arguments (first 4 arguments)	$a0-$a3
call result is returned in	$v0,$v1
temp registers (caller-save)	$t0-$t9
temp registers (callee-save)	$s0-$s7
stack pointer	$sp
frame pointer (use only for procedure call)	$fp
return address (use only for procedure call)	$ra

Table 2. MIPS Address Modes

Description	Assembler Syntax
register contents	$t0
immediate	number
immediate + register constant	name(register)
address of symbol	symbol (i.e label)
address of symbol plus immediate	symbol + or - immediate
address of symbol plus immediate plus register	symbol + or - immediate(register)

The assembled program is split into two segments, `text' containing the executable code and `data' containing variables and constants, the two directives `.text` and `.data` control which segment the assembler puts each command in. The .data and .text directives can be used as many times as needed, the assembler joins the sections in to one data segment and one executable segement.

Table 3. Assembler directives

Directive	Operand	function

.asciiz	string	reserve memory and initialize it with the string and the ending zero char
.data		the items following are stored in the data segment
.globl	label	declared the label as global
.space	n	reserve an area of memory, n bytes long in the current segment
.text		the items following are stored in the text (program) segment
.word	n1,n2,..	reserve words of memory initialised to n1,n2..

Instruction Set

Except for 6 two operand instructions all instructions have three operands, the destination (a register) and two sources, the first source is a register, the second can be a register or constant. When the second source is a number the assembler generates a different (immediate) instruction, in the example above 'addi' could be written as 'add'. The subset of mips instructions below are those you are likely to use in the compiler,

The mips assembler has two modes, in the default mode some of the source instructions (and instructions using the more complicated adress mode) are translated into more than one machine instruction when written into memory, so spim shows the executable memory as slightly different to the code loaded. In bare mode the assembler source must contain only basic instructions and address modes that machine can store in one word.

Table 4. Arithmetic and Bit Instructions

Instruction	Operands	function
add	dest,src1,src2	dest = src1 + src2
sub	dest,src1,src2	dest = src1 - src2
mul	dest,src1,src2	dest = src1 * src2
div	dest,src1,src2	dest = src1 / src2
rem	dest,src1,src2	dest = src1 remained src2
neg	dest,src1	dest = - src1
and	dest,src1,src2	dest = src1 bitwise and src2
or	dest,src1,src2	dest = src1 bitwise or src2
not	dest,src1	dest = bitwise not src
xor	dest,src1,src2	dest = src1 bitwise xor src2
sll	dest,src1,src2	dest = src1 shift left by src2
srl	dest,src1,src2	dest = src1 shift right by src2

Table 5. Branches

Instruction	operands	function
b	label (16 bit offset)	unconditional jump
beq	src1,src2,label	branch if src1 = src2
bne	src1,src2,label	branch if src1 /= src2
bgt	src1,src2,label	branch if src1 > src2
bge	src1,src2,label	branch if src1 >= src2
blt	src1,src2,label	branch if src1 < src2
ble	src1,src2,label	branch if src1 <= src2
j	label (26 bit address)	jump to label

jal	label or Reg	jumps to label saving the return address in $ra
syscall	-	calls the operating system $v0 = 1 : print $a0 as an integer $v0 = 4 : print the string addressed by $a0 $v0 = 5 : read an integer into $v0 $v0 = 10 exit (stop)

All of the conditional branches have an extra form when the second operand is 0, for example

```
 beq $a0,0,label
```

can be written as

```
 beqz $a0,label
```

Table 6. Load, store and misc

Instruction	operands	function
li	dest,immed	dest = immediate value (16 bits)
lui	dest,immed	dest = immediate value left shift 16
la	dest,src	dest = address of src
lw	dest,src	dest := data from src address (32 bit word)
sw	src,dest	store src 32 bit value in memory at dest
move	dest,src	dest := src register move.

Note that unless the assembler is in bare mode the values for the li (and lui instructions) can be longer than 16 bits.

Memory and Register allocation

As the subset of C to be compiled does not support user functions the location of a variable will be known at compile time and variables should be stored in the data segment rather than on the stack. The compiler requires that variable declarations must be placed before statements in a block. Variables in inner blocks hide variables with the same name, for example here the variable `i` refers to different locations at different steps of the program:

```
int i;

int main() {
   int j = 0;
    i =1;
   if ( j ==0) {
     int i =j;
    printf("i is %d\n",i);
}
   printf("i is %d\n",i);
 }
```

The parser translates identifiers to unique name depending on the scope they are in.

Register allocation

The compiler must make best use of registers to avoid the slower load and store instructions when possible. You can use both caller-save and callee-save temp registers as scratch registers since you are not compiling user procedures. When the program does input or output using the syscall instructions the registers $a0 and $v0 are used, otherwise the argument and result registers can be used as well. Providing the generated program uses an exit system call the `$sp` and `$ra` can also be used as general registers.

To get best performance out of a MIPS machine the compiler must use registers to hold variable values when possible, and use the memory when required, for example if it not possible to know until the program is run which branch of an 'if' is taken.

It is difficult to test the register allocator by as part of the compiler, work through the milestone to produce a tested register allocator

Generating Code

The parser produces an abstract syntax tree representing the program as a java tree. To generate the code the program must traverse the AST and write out the appropriate code for each node. Antlr provides support for traversing and checking trees using a *tree parser* described below.

Input/output

The calls to `printf` and `scanf` in the program will be converted by the parser into sequences of simpler calls in the tree for the functions `printd` (print an integer), `prints` (print a string) and `getd` (read an integer) which have one argument and which you should implement using the appropriate system call in the assembler.

Order of writing the compiler.

1. After testing your register allocator, add code to the the tree grammar to print integers - translating calls to `printd`
2. Add code to translate arithmetic expressions. Look at the provided test files for suitable test files, and use spim to see if the result of your compiler can be assembler and produces the expected result.
3. Extend your compiler to generate code for assignment and other statements. When you have added a statement test it by running your code on spim.

General points

The generated code must make an exit system call so it stops at the end of the `main` function.

In C (and java) the second argument of `&&` (and) and `||` (or) is not evaluated if the value of the first argument makes the result known For example when

```
if ( 1==2 && 1/0 > 0)
{ printf("wrong\n"); }
```

is compiled and run it should not stop a zero divide exception or print anything.

If the code generator evaluates constant expressions like the example above, it should not crash with a zero divide, and also not tell the user that the code is incorrect, since the `1/0` might not be executed.

Using a tree parser

If the tree was built with a different class for each type of tree node, you should use the *Visitor* pattern to define and perform operations on the tree. In this exercise **antlr** (version 3) tool will a create a tree parser from the provided tree grammar. The tree parser will walk from the root of the AST. When a rule matches the current node of the tree code (actions) can be called at the node, either before or after the branches of the node are traversed. Actions can receive arguments and pass back results.

For example in this part of the tree grammar:

```
expr :
     binaryexpr
   | unaryexpr ;

unaryexpr :
     (UMINUS expr)
     (BITNOT expr)
   | IDENT
   | NUMBER ;

  binaryexpr   :
       ^(PLUS  expr expr )
     | ^(MINUS expr expr )
     ..
 cond  :
    ^(IF condition statement  )
    ^(IELSE condition condition)
     ..
```

the grammar says the root of a tree representing a binary expression contains a 'PLUS' and two branches both expressions. Tokens have uppercase names and other non-terminal are lowercase. The example as shown will walk down an expression tree and report any unexpected nodes as errors.

To generate code for an expression using the tree parser you need to add actions to the tree grammar. An action is a section of code in the language the parser is implemented, which is executed as the parser reaches the associated node in the tree and which can get information from the parser. In antlr variables and methods for use in actions can be included the grammar in the methods section, look at the top of the provided files `Pico.g` and `Gen.g`.

```
// result: the register in which  the value will be put
 expr[Register use] returns [ Register result ] :                        ❶
     binaryexpr[use] {result = $binaryexpr.result; }
   | unaryexpr[use]  {result = $unaryexpr.result; } ;

  unaryexpr [Register use] returns  :
   ..
   | id=IDENT  { result = getFreeRegister();
     code.println("lw\t"+result.toString()+","+$id.getText());   }       ❷
      ..

 binaryexpr [ Register use ]  returns  [Register result  ]              ❸
    @init{   Register left=null,right=null; }                            ❹
    @after { freeRegister($right); }
    :

       ^(o1=PLUS  left=expr right=expr )
            { result=arith($o1,$left,$right); }                          ❺
          ..

  cond[int trueLabel,int falseLabel]  :                                  ❻
```

```
^(IF condition[trueLabel,falseLabel]
        {defineLabel(trueLabel);}
     statement  )
        {defineLabel(falseLabel);}
```

❶ Register is a class defined in the sample solution and when an expression tree has been traversed a register is returned. The variables in the list of declarations following following the *returns* keyword will contain values when the expr rule ends.

❷ `id` refers to the tree node of an identifier, the action (in brackets) is a (list of) java statements. The method `println` writes the "lw" instruction to load the value from the memory) the variable `id` an AST, use using the AST `getText()` method to get the name of the variable as a String into a register.

❸ Arguments can be passed to the actions of a grammar rule. Here it is used to specify an existing register to be used for the result.

❹ An *@init* section can contain declarations and code which is executed before a choice in the rule is chosen. An *@after* section is called after the rule is completed.

❺ After walking down the two expression branches the tree parser will return the names of the registers to hold the values. and the method `arith` will write the assembly instruction and return the register name of the result.

❻ Two label numbers are passed in as arguments to conditional statements and two actions are used to define the labels before and after the body of the statement.

When **antlr** translates the grammar to a java program implementing the parser, the actions are placed in the java source. Actions are not translated as methods, but you can call methods in an action. If the java in an action has a syntax or type error it will usually reported only when the code generated for the grammar is compiled and error messages will refer to lines in the generated java. Antlr adds comments to the generated java with the file and line number where the code come from, for example

```
// pico/Gen.g:441:4: ^(EQUAL ...)
```

Documentation for antlr version 3 is provided at http://www.antlr.org/

Running the compiler and the generated code

Compiling and running the compiler

The provided `makefile` will generate java files from each of the grammar files and then compile the generated java files.

The provided shell script `compile` runs the compiler with one source file name as an argument. The compiler will look for a source file first in the current directory and if it is not found looks in the directory `/vol/lab/secondyear/compiletests`. The assembler produced will be written in the current directory with a suffix `.mips`. The binary of the sample solution can be run by typing **labcompile**.

Running the assembler

The mips assembler file can be run using the **spim** command :

```
spim  helloworld.mips
```

assembles the file and runs it. If you type **mips** without a file name commands are read from the terminal, type **help** for a list of commands.

xspim provides an X windows interface for **spim** so registers and memory can be inspected and changed. A Playstation 2 (which has a mips cpu) with linux can be used instead.

Questions and Further Information

The page http://www.doc.ic.ac.uk/lab/secondyear/compiler/faq.html will contain answers to questions.

Testing

A number of test files are available in `/vol/lab/secondyear/compilertests`, the file names indicate what compiler feature it tests. If the Compiler does not find the source file given to it as an argument it will look in the test directory.

The compiler should be tested as it is developed and if the compiler supports input and output early on it will be easier to see if a test is running correctly.

Submission and Assessment

Milestone

Submit the source of your register and register allocator classes and your extensions to `RegExer.java` after a week.

Submission

At the end of the exercise use **zip** to create `gen.zip` containing your modified `Gen.g` and the source of the classes you have written to be called by the tree parser. Submit `gen.zip` to cate.

Assessment

Correctness

The compiler, when given a correct program, will generate code that can be assembled and run giving the expected result. If the compiler runs out of resources such as registers it must stop with a clear error message.

Efficiency

The compiler makes the best use of the machine features to represent the program.

The marks are allocated as follows:

Mark distribution	
Completion of the milestone on time	20%
Correct compilation	40%
Efficient use of MIPS instructions in generated code	20%
Design, layout and readability	20%