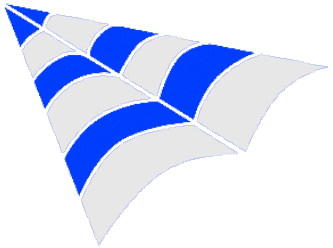
ALaRI
Software Compilers



CODE GENERATION FOR MIPS INSTRUCTION SET USING THE ACSE COMPILER

Tutor: Giovanni Agosta

Project Authors:
Luis Gabriel Murillo
Ricardo Andres Velasquez

A.A. 2008

Contents

List of Figures

List of Tables

1 Introduction

The Advanced Compiler System for Education (ACSE) is a compiler designed to translate source code written in LanCE language into an assembler code for the MACE architecture. Even though the ACSE is a simple compiler, it provides all the elements to perfectly understand how a compiler works, since it is able to perform actions such as create a control flow graph, execute a liveness analysis and make a register allocation.

Due to the fact that the ACSE is hardly restricted to generate code for the MACE architecture, this project aims to create a modified core for the ACSE compiler, which will allow to generate working MIPS assembler code, suitable to execute in the SPIM simulator. Modifications mainly in the Back-end have been made to reach the goal, but some other modifications in the Front-end were used to extend the initial grammar supported by the parser, and allow the insertion of the Switch structure and the Break and Continue statements. This report briefly describes first some hints of the MIPS Architecture, then the changes made to the ACSE compiler's back-end and front-end, and finally the results obtained with the new compiler.

2 MIPS Architecture

The architecture of the MIPS computers is simple and regular, which makes it easy to learn and understand. The processor contains 32 general-purpose registers and a well-designed instruction set that make it a propitious target for generating code in a compiler. The implemented version of the instruction set in this project is MIPS R2000/R3000, supported by SPIM v6.5 simulator. In this section we present some particular hints needed to implement the MIPS code generation inside the ACSE compiler.

2.1 CPU Registers

The MIPS central processing unit contains 32 general purpose registers that are numbered from 0 to 31. Each register is designated by $n. Register $0 always contains the hardwired value 0. MIPS has established a set of conventions as to how registers should be used. These suggestions are guidelines, which are not enforced by the hardware. However a program that violates them will not work properly with other software. Table 1 lists the registers and describes their intended use. Registers $at (1), $k0 (26), and $k1 (27) are reserved for use by the assembler and operating system. Registers $a0 to $a3 (4 to 7) are used to pass the first four arguments to routines (remaining arguments are passed on the stack). Registers $v0 and $v1 (2, 3) are used to return values from functions. Registers $t0 to $t9 (8 to 15, 24, 25) are caller-saved registers used for temporary quantities that do not need to be preserved across calls. Registers $s0 to $s7 (16 to 23) are callee- saved registers that hold long-lived values that should be preserved across calls. Register $sp (29) is the stack pointer, which points to the last location in use on the stack. Register $fp (30) is the frame pointer. Register $ra (31) is written with the return address for a call by the jal instruction. Register $gp (28) is a global pointer that points into the middle of a 64K block of memory in the heap that holds constants and global variables. The objects in this heap can be quickly accessed with a single load or store instruction.

If we compare the previous registers with the registers in MACE architecture, is noticeable that, in order to produce a good assembler for MIPS, a change in the way the registers are allocated by ACSE is needed, since in MACE's assembler you can use any register between R1 and R31 to perform the normal operations, while to build the MIPS assembler is necessary to use only the registers $zero, $t0 to $t9 and $s0 to $s7. The registers $v0 to $v2 are used in the Read and Write operations, as we will show later.

Register Name	Number	Usage
$zero	0	Constant 0
$at	1	Reserved
$v0	2	Return from a function
$v1	3	Return from a function
$a0	4	Argument 1
$a1	5	Argument 2
$a2	6	Argument 3
$a3	7	Argument 4
$t0	8	Temporary
$t1	9	Temporary
$t2	10	Temporary
$t3	11	Temporary
$t4	12	Temporary
$t5	13	Temporary
$t6	14	Temporary
$t7	15	Temporary
$s0	16	Saved Temporary
$s1	17	Saved Temporary
$s2	18	Saved Temporary
$s3	19	Saved Temporary
$s4	20	Saved Temporary
$s5	21	Saved Temporary
$s6	22	Saved Temporary
$s7	23	Saved Temporary
$t8	24	Temporary
$t9	25	Temporary
$k0	26	Reserved for OS Kernel
$k1	27	Reserved for OS Kernel
$gp	28	Global area pointer
$sp	29	Stack pointer
$fp	30	Frame pointer
$ra	31	Return Address

Table 1: CPU Registers in MIPS Architecture

2.2 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 (integer) c and the contents of register rx as the address. The addressing modes provided by the virtual machine for load and store instructions are shown in Table 2.

For giving the ACSE capability to generate MIPS assembler, was necessary to modify all the structure of the axe_gencode.c file, aiming to re-organize the instruction generation. One of the facts that motivated such change was the differences between the addressing modes in MACE and the addressing modes in MIPS, because for example in MACE is possible to access the memory in an indirect way using the ternary instructions (three operands), while in MIPS is only possible with the load and store instructions which have not ternary structure.

Format	Address Computation
register	Contents of Register
imm	Immediate
imm(register)	Contents of Register + Immediate
symbol	Address of Symbol
symbol +/- imm	Address of Symbol + or - Immediate
symbol +/- imm(register)	Address of Symbol + or - (Cont. Register + Imm)

Table 2: Addressing Modes in MIPS

2.3 Instructions

The MIPS R2000/R3000 assembler has a big number of instructions, grouped in the categories shown below. Although not all the different kinds of instructions are needed to translate the LanCE language, almost all the instructions between the IS were implemented inside the ACSE core. Only the Floating Point Instruction weren't implemented.

- Arithmetical and Logical Instructions
- Constant-Manipulating Instructions
- Branch Instructions
- Jump Instructions

- Trap Instructions
- Load Instructions
- Store Instructions
- Data Movement Instructions
- Floating Point Instructions
- Exception and Interrupt Instructions

Each group of instructions has different formats, going from unary instructions to ternary instructions, like in ACSE original assembler, but the format of the MIPS doesn't match this assembler. For example, big changes to the branch instructions should be made.

2.4 System Calls

The System Calls are a small set of services provided by the SPIM simulator. Although this services are not included inside the original MIPS instruction set, they are very useful when the assembler code is going to be tested, because using them we can read an input or write an output to the console. Implementing the syscalls is possible to translate the READ and WRITE instructions provided inside the LanCE grammar. To request a service using syscall, the service code should be written to the $v0 register, and the arguments to the $a0 to $a3 registers, thus replacing the way in which the READ and WRITE instructions were previously generated for MACE. The available services in the SPIM simulator are used to read and write different types of data, such as integers, floats, doubles and strings, but according to the requirements of LanCE language just the services for integer manipulation were needed. Table 3 shows the syscalls implemented inside the ACSE, to accomplish the LanCE requirements. In Figure 1 we show a code example of how a syscall is implemented.

```
li $v0, 1       # system call code for print_int
li $a0, 5       # integer to print
syscall         # print it
```

Figure 1: Syscalls used in ACSE to generate MIPS assembler

3 Back-End Modifications

Due to the fact that the instructions in MIPS assembler significantly differ from those in the MACE assembler, was necessary to make a lot of changes in the components of the back-end, such as axe_gencode, axe_cflow_graph, axe_transform, axe_expressions, axe_engine, axe_array and axe_constants.

Basically, the work involved to remove incompatible instructions, change the name to some other instructions, add some new useful instructions, and modify the addressing modes in others. All this in order to obtain a right assembly code for MIPS architecture. Addressing modes changes in some instructions (mainly branch instructions) have implied several changes also in the front end.

The most significative changes were made inside the gencode file, where we almost reconstructed again the entire module. The initial funtions to build unary, binary, ternary and jump instructios were replaced according to the needs of the MIPS instructions. Such instructions were clasified into the following groups, and one function per group was built in the mentioned file.

- ternary instructions
- binary immediate instructions
- binary instructions
- unary immediate instructions
- unary instructions
- load/store instructions
- ternary branch instructions
- binary JUMP instructions
- unary jump instructions
- unconditionally jump

Service	System call code (in $v0)	Argument	Result
print_int	1	integer in $a0	
read_int	5		integer in $v0

Table 3: Syscalls implemented in ACSE

- special instructions

The subgroups represent the whole set of instructions for MIPS R2000/R3000 (excuding FP instructions), and in the following subsections we explain in detail all the MIPS instructions that our back-end is able to generate, and the technique we implemented to generate it taking as beginning the ACSE core.

3.1 ternary instructions

OPCODE RDEST, RSOURCE1, RESOURCE2

ADD	Addition with overflow
ADDU	Addition without overflow
AND	and bitwise
DIV	Divide with overflow (pseudoinstruction)
DIVU	Divide without overflow immediate (pseudoinstruction)
MUL	Multiply without overflow
MULO	Multiply with overflow (pseudoinstruction)
MULOU	Unsigned multiply with overflow (pseudoinstruction)
NOR	nor bitwise
OR	or bitwise
REM	Remainder (pseudoinstruction)
REMU	Unsigned remainder (pseudoinstruction)
SLLV	Shift left logical variable
SRAV	Shift right arithmetic variable
SRLV	Shift right logical variable
ROL	Rotate Left (pseudoinstruction)
ROR	Rotate Right (pseudoinstruction)
SUB	Subtract with overflow
SUBU	Subtract without overflow
XOR	Exclusive or
SLT	Set less than
SLTU	Set less than unsigned
SEQ	Set equal (pseudoinstruction)
SGE	Set greater than equal (pseudoinstruction)
SGEU	Set greater than equal unsigned (pseudoinstruction)
SGT	Set greater than (pseudoinstruction)
SGTU	Set greater than unsigned (pseudoinstruction)
SLE	Set less than equal (pseudoinstruction)
SLEU	Set less than equal unsigned (pseudoinstruction)
SNE	Set not equal (pseudoinstruction)
MOVN	Move conditional not zero
MOVZ	Move conditional zero

3.2 binary immediate instructions

OPCODE RDEST, RSOURCE1, IMMEDIATE

ADDI	Addition immediate with overflow
ADDIU	Addition immediate without overflow
ANDI	and bitwise immediate
ORI	or bitwise immediate
SLL	Shift left logical
SRA	Shift rigth arithmetic
SRL	Shift rigth logical
XORI	xor immediate
SLTI	Set less than immediate
SLTIU	Set less than unsigned immediate

3.3 binary instructions

OPCODE RDEST, RSOURCE1

ABS	Absolute Value
CLO	Count leading ones
CLZ	Count leading zeros
DIV	Divide with overflow (quotient in register lo and the remainder in register hi)
DIVU	Divide without overflow immediate (quotient in register lo and the remainder in register h
MULT	Signed Multiply (low-order word in lo and high-order word in hi)
MULTU	Unsigned Multiply (low-order word in lo and high-order word in hi)
MADD	Multiply add (64 bit result in the concatenated register lo and hi)
MADDU	Unsigned multiply add (64 bit result in the concatenated register lo and hi)
MSUB	Multiply subtract (64 bit result in the concatenated register lo and hi)
NEG	Negate value with overflow (pseudoinstruction)
NEGU	Negate value without overflow (pseudoinstruction)
NOT	bitwise negation (pseudoinstruction)
JALR	Jump and link register
TEQ	Trap if equal
TNE	Trap if not equal
TGE	Trap if greater equal
TGEU	Unsigned Trap if greater equal
TLT	Trap if less than
TLTU	Unsigned Trap if less than
MOVE	Move

3.4 unary immediate instructions

OPCODE RDEST, IMMEDIATE

LUI	Load upper immediate
LI	Load immediate (pseudoinstruction)
TEQI	Trap if equal immediate
TNEI	Trap if not equal immediate
TGEI	Trap if greater equal immediate
TGEIU	Unsigned trap if greater equal immediate
TLTI	Trap if less than immediate
TLTIU	Unsigned Trap if less than immediate

3.5 unary instructions

OPCODE RDEST

JR	Jump register
MFHI	Move from hi
MFLO	Move from lo
MTHI	Move to hi
MTLO	Move to lo

3.6 binary jump instructions

OPCODE RSOURCE1, RESOURCE2, LABEL

BEQ	Branch on equal
BNE	Branch on not equal
BGE	Branch on greater than equal (pseudoinstruction)
BGEU	Branch on greater than equal unsigned (pseudoinstruction)
BGT	Branch on greater than (pseudoinstruction)
BGTU	Branch on greater than unsigned (pseudoinstruction)
BLE	Branch on less than equal (pseudoinstruction)
BLEU	Branch on less than equal unsigned (pseudoinstruction)
BLT	Branch on less than (pseudoinstruction)
BLTU	Branch on less than unsigned (pseudoinstruction)
BNEZ	Branch on not equal zero (pseudoinstruction)

3.7 unary jump instructions

OPCODE RSOURCE1, LABEL

BGEZ	Branch on greater than equal zero
BGEZAL	Branch on greater than equal zero and link
BGTZ	Branch on greater than zero
BLEZ	Branch on less than equal zero
BLTZAL	Branch on less than and link
BLTZ	Branch on less than zero
BEQZ	Branch on equal zero (pseudoinstruction)

3.8 unconditionally jump

OPCODE

B	Unconditionally Branch
J	Unconditionally
JAL	Jump and link

3.9 load/store instructions

LOAD/STORE instruction are the only way to access memory.

LA	Load Address
LB	Load Byte
LBU	Load unsigned byte
LH	Load Halfword
LHU	Load unsigned halfword
LW	Load word
LWC1	Load word coprocessor 1
LWL	Load word left
LWR	Load word right
LD	Load doubleword (pseudoinstruction)
ULH	Unaligned load halfword (pseudoinstruction)
ULHU	Unaligned load halfword unsigned (pseudoinstruction)
ULW	Unaligned load word (pseudoinstruction)
SB	Store Byte
SH	Store Halfword
SW	Store word
SWC1	Store word coprocessor 1
SDC1	Store double coprocessor 1
SWL	Store word left
SWR	Store word right
SD	Store doubleword
USH	Unaligned Store halfword (pseudoinstruction)
USW	Unaligned Store word (pseudoinstruction)

3.10 special instructions

NOP
SYSCALL

4 Front-End Modifications

Other modifications for the ACSE compiler included the implementation of the *Switch* structure and the *Break* and *Continue* statements. Implementing support for such statements helps to increase the functionality of the ACSE compiler. The changes for achieve this operation were made mainly in the front-end of the compiler, and especificaly to the Lexical Analyzer and the Parser. Inside the lexical analyzer the support for the new tokens were added, while in the parser the actions and the new grammatical rules were defined. The explanation of the operation and the limitations of the new structures are discussed in the following subsections.

4.1 Switch

In order to implement the switch structure three grammar rules were added to the *acse.y* file. These rules act to detect the structure of the *switch*, the structure of the *set of cases + default*, and the structure of each *individual case*. A summary of the grammar rules is shown in the Figure 2.

```
switch_statements  : SWITCH LPAR exp RPAR LBRACE  {/*ACTIONS*/}
                     case_statements {/*ACTIONS*/}
                     RBRACE{/*ACTIONS*/} ;

case_statements :  case_statement case_statements { /* does nothing */ }
                |  DEFAULT COLON statements BREAK SEMI{ /* does nothing */ }
                | /* empty */ { /* does nothing */ };

case_statement : CASE exp {/*ACTIONS*/}
                 COLON statements BREAK SEMI {/*ACTIONS*/};
```

Figure 2: Grammar Rules for Switch

From the grammar rules is easy to notice that the structure can switch expressions such as *a+b*, *a-b*, *a*, like the real C-like switch. Is also noticeable that the *default* statemente should be put after the last *case*, as this was motivated for the way we implemented the operation of the structure. Basically the structure works in this way: When the *switch* clause and the condition are recognized, the end label of the switch and the register holding the condition are stored, and the *case* clause begins to be recognized. When the case and the value of the case are recognized, a comparison between the condition and the value is performed, followed by a branch instruction which will allow to perform or to skip the execution of the statements between the case. If the statements are executed, an unconditional jump to the end label

of the switch structure is performed. The same metodology is implemented for all the cases, and finally if any condition was satisfied the statements under the *default* clause are performed. The big challenge when implementing the switch was to find a way to store the end label and the condition to be evaluated before entering each case. The solution was found using a stack (t_list) from the class *collections.c* to store them. This implementation allows also to create nested switches completely funtionals, since with the stack is easy to use always the right labels and conditions. The switch was proved compiling single switches and nested switches, and the results were satisfactory under the MACE architecture. The only constraint to use the MACE architecture is to comment the insertion of a NOP instruction (seems that the Assembler compiler doesn't support NOP instructions) in the end label of the switch, and to use instead a harmless instruction (we used instead a ADD R0 R0 R0). Some compiled examples can be found inside the folder tests, contained in the front-end folder of the project.

4.2 Continue and Break

The Continue and the Break statements were implemented inside all the loops (Do while, while and for), creating the needed unconditional branch instructions. The grammar rules for Break and Continue were added to a copy of code_block, called code_block_bc, and then this rule was used inside the loops statements and the if statement. In order to make those instructions to work in the correct way we implemented again the stack t_list to store the ID of the label where to go next if the parser found a continue or a break. To separate the Break and the Continue we used two different lists for each one. The IDs are added to the top of the list when a condition label(continue list, step label in case of a for) or a end label (break list) is created inside the loops. Then if a break or a continue is found, the jump instruction is created with the first element of the list. After each loop, the first element of the list is deleted. Some compiled examples can be found inside the folder tests, contained in the front-end folder of the project.