SOM2IF15 – Compilation

MIPS Assembly Code Generation

Frédéric Loulergue

2022

Outline

Result of the Previous Phase

Translation of IR

- The result is basically: `List<Pair<Frame, List<Command>>>`
- For each LPL function we have:
  - A `Frame` that is a translation of the function's header
  - A `List<ir.com.Command>` that is a translation of the function's body
- `ir.com.Command` is quite close to MIPS assembly instructions
- but some IR commands contain **expressions**
- no expressions in MIPS assembly code

Outline

Reminder: Intermediate Representation

Expressions

$$
\begin{array}{rcll}
\langle expr \rangle & ::= & number & \text{literal number} \\
& | & \texttt{reg}_i & \text{temporary register} \\
& | & \texttt{reg}_i[\langle expr \rangle] & \text{memory read} \\
& | & unop\ \langle expr \rangle & \text{unary operation function application} \\
& | & \langle expr \rangle\ binop\ \langle expr \rangle & \text{binary operation application}
\end{array}
$$

where

- $i \in \mathbb{Z}$
- *unop* contains `-`, `!`
- *binop* contains all arithmetic and Boolean binary operations but NOT `++` and `--`

In Java

- Sub-classes of `ir.expr.Expressions`
- A visitor: `ir.expr.Visitor`

MIPS Code Generator

Principles

- Generation of `List<String>` (to simplify)
- Use of the utility methods in `mips.Asm`

Expressions

- Like `pico`: compilation as in a stack machine
- For each (sub-)expression:
  - compilation to produce a value
  - push this value to the stack
- Unlike `pico`:
  - more binary operations
  - variable cannot be allocated statically

Size of Values & Stack Operations

Asm.java

```
static int sizeOf(ir.Type type) {
    if (type == Type.BYTE) return 1;
    return 4;
}
static List<String> push(String register) {
    List<String> asmCode = MakeList.one(command("sub $sp, 4"));
    asmCode.add(command("sw " + register + ", 4($sp)"));
    return asmCode;
}

static List<String> pop(String register) {
    List<String> asmCode =
  MakeList.one(command("lw " + register + ", 4($sp)"));
    asmCode.add(command("add $sp, 4"));
    return asmCode;
}
```

```
@Override
public List<String> visit(Byte exp) {
    List<String> asmCode =
  MakeList.one(Asm.command("li $t0, " + exp.getValue()));
    asmCode.addAll(Asm.push("$t0"));
    return asmCode;
}

@Override
public List<String> visit(Int exp) {
    List<String> asmCode =
  MakeList.one(Asm.command("li $t0, " + exp.getValue()));
    asmCode.addAll(Asm.push("$t0"));
    return asmCode;
}
```

Binary Expressions

Expression.java

```
public List<String> visit(Binary exp) {
    List<String> left = exp.getLeft().accept(this);
    List<String> right = exp.getRight().accept(this);
    String op = null;
    switch (exp.getOp()) {
        case ADD:
            op = "addu";
            break;
            // ...
    List<String> asmCode = left;
    asmCode.addAll(right);
    asmCode.addAll(Asm.pop("$t2"));
    asmCode.addAll(Asm.pop("$t1"));
    asmCode.add(Asm.command(op + " $t0, $t1, $t2"));
    asmCode.addAll(Asm.push("$t0"));
    return asmCode;
}
```

Reminder: Array Memory Layout

- The value of an array is its address
- Assuming the size of each cell is sz and there are n cells:
  - 4 bytes at the very beginning to store the number of cells (n)
  - $sz \times n$ bytes to store the cell values
- For a string *literal* like `"hello"`:
  - the number of cells: number of characters + 1
  - memory layout:
    - 4 bytes to store the size: `int` value 6
    - 6×1 bytes to store the characters `'h'`, `'e'`, `'l'`, `'l'`,
    - `'o'`, `'\0'`

Temporary Registers

Naive Approach

- Let's store them on the stack
- A temporary register = an offset wrt to the current frame

Reminder

- the stack grows in decreasing addresses
- `$fp` beginning of the current frame
- `$sp` end of the current frame
- we use the stack to store intermediate results:
  `$sp` changes all the time
- `$fp` changes only during a procedure call

$\Longrightarrow$ A register associated with an offset wrt `$fp`

Example: LPL Code and IT Translation

```
int select
  (bool theFirst,
   int first,
   int second)
{
  if (theFirst)
    return first
  else
    return second
}
```

```
==== FRAME: L0:
  FRAME INFO:
    {entryPoint=L0: , exitPoint=L1:
     parameters=[reg0, reg1, reg2],
     result=Optional[reg3],
     locals=[], size=0}
  CODE: [CJump (reg0, L2: , L3: ),
         L2: ,
             reg3 := reg1,
             goto L1: ,
         L3: ,
             reg3 := reg2,
             goto L1: ]
==== END FRAME
```

Example: The Activation Record

- All the temporary registers will be stored on the stack
- In the example: 4 temporary registers
- To simplify: on the stack all values will have a machine word size
- On the heap, values will have their actual size

MIPS Code Generation for Expressions

IR Expressions to MIPS

- Constants & binary expressions: provided

Register Read

- `offset` in the mapping regAlloc: `Map<Register, Integer>`
- this offset is wrt to `$fp`
- the default size is a machine word
- reading: `lw $t0, offset($fp)`
- the result should be pushed on the stack

MIPS Code Generation: Memory Read: $\texttt{reg}_i[\langle expr \rangle]$

- the address of the array in $\texttt{reg}_i$: basically a register read, no need to push it on the stack, let's assume it is in `$t0`
- the LPL offset is $\langle expr \rangle$: recursively compile it.
  The generated code will push the integer value on the stack.
- pop the index: let's assume it is in `$t1`
- the LPL offset is not the MIPS offset
- if i is the LPL offset, sz the size of cells, MIPS offset: $4 + sz \times n$
  You need to generate the code to compute this value.
  where sz is actually a constant you can obtain from:
  `sizeOf` the *type* of the memory read
  Let's assume your code stores this MIPS index in `$t2`
- depending on the size: `lb` or `lw` (see `load`)
- `load $t3, $t2($t0)` is not possible: the offset should be a literal
- instead: add `$t0` to `$t2` and use `load $t3, ($t2)`
- don't forget to push `$t3` on the stack!

Outline

Reminder: Intermediate Representation

Instructions

$\langle com \rangle$	::=	*label*:	a label
	|	$\texttt{reg}_i := \langle expr \rangle$	write to a register
	|	$\texttt{reg}_i + \langle expr \rangle := \langle expr \rangle$	write in memory
	|	$\texttt{jump}\ (\langle expr \rangle)\ label,\ label$	conditional jump
	|	$\texttt{goto}\ label$	unconditional jump
	|	$\texttt{call}\ frame\ \langle expr \rangle, \ldots, \langle expr \rangle$	function/system call
	|	$\texttt{reg}_i := frame\ \langle expr \rangle, \ldots, \langle expr \rangle$	function/system call

MIPS Code Generation

- Label: direction generation
- `goto` *label*: `j label`
- Write to register:
  - compilation of the expression: the execution of the generated code puts the value on the stack
  - register: get its *offset* from the mapping register → offset
  - final code: $code_{\text{expression}} ++ pop(\texttt{\$t0}) ++ [\texttt{save \$t0}, offset(\texttt{\$fp})]$ where `save` is:
    - `sw` (if the type is `INT` or `ADDRESS`)
    - `sb` (if the type is `BYTE`)
- Write to memory (array): similar to write to register but:
  - We need to get the address of the array (it's in a register)
  - We need to compute the offset for the given cell
  - The LPL index is in the first expression: the generated code puts it on the stack (i)
  - The ASM index is: $4 + sz \times i$ where sz is the size of a cell

MIPS Code Generation: Function and Procedure Calls

- First step: parameter passing (only call by value)
  First four parameters in $a0 to $a3
  (for now no function with more than 4 parameters)
- Second step: call the function/procedure
- Other aspects of functions/procedures: in the Prologue and Epilogue when generating code for a frame and its associated fragment

Outline

Frames and Fragments

- A fragment is a list of `Command`
- Concatenation of the `List<String>` obtained for each `Command`
- Frames: represent functions
- Prologue:
  - code to save callee-saved MIPS registers
  - code to update `$fp` and `$sp`
  - code to associate MIPS register parameters and IR registers
- body of the function (from the fragment)
- Epilogue:
  - code to restore callee-saved MIPS registers
  - if function: copy the result in `$v0`
  - jump back to the caller

Outline

Code Generation for Programs

- MIPS assembly programs require a `.text`, and possibly `.data`, directive
- MIPS assembly programs require a `main` label
- an explicit call to the exit system call is necessary
- Code generation for each frame

Example

```
void main(){
  print("Hello World!\n")
}
```

Example I

```
        .text
main:
        jal L0
        li $v0, 10
        syscall
L0:
        move $t0, $fp
        move $t1, $ra
        move $fp, $sp
        addi $sp, $sp, -12
        sw $t0, 8($sp)
        sw $t1, 4($sp)
        li $t0, 1
        sub $sp, 4
        sw $t0, 4($sp)
        li $t0, 14
        sub $sp, 4
        sw $t0, 4($sp)
        lw $a1, 4($sp)
        add $sp, 4
        lw $a0, 4($sp)
        add $sp, 4
        jal entryNew
        sw $v0, 0($fp)
        li $t0, 72
        sub $sp, 4
        sw $t0, 4($sp)
        li $t0, 0
        sub $sp, 4
        sw $t0, 4($sp)
        lw $t2, 4($sp)
        add $sp, 4
        lw $t1, 0($fp)
        li $t3, 1
        mul $t2, $t2, $t3
        add $t1, $t1, $t2
        li $t3, 4
        add $t1, $t1, $t3
        lw $t2, 4($sp)
        add $sp, 4
        sb $t2, ($t1)
        li $t0, 101
        sub $sp, 4
        sw $t0, 4($sp)
        li $t0, 1
        sub $sp, 4
        sw $t0, 4($sp)
        lw $t2, 4($sp)
        add $sp, 4
        lw $t1, 0($fp)
        li $t3, 1
        mul $t2, $t2, $t3
        add $t1, $t1, $t2
        li $t3, 4
        add $t1, $t1, $t3
        lw $t2, 4($sp)
        add $sp, 4
        sb $t2, ($t1)
        li $t0, 108
        sub $sp, 4
```

Example II

```
sw $t0, 4($sp)
li $t0, 2
sub $sp, 4
sw $t0, 4($sp)
lw $t2, 4($sp)
add $sp, 4
lw $t1, 0($fp)
li $t3, 1
mul $t2, $t2, $t3
add $t1, $t1, $t2
li $t3, 4
add $t1, $t1, $t3
lw $t2, 4($sp)
add $sp, 4
sb $t2, ($t1)
li $t0, 108
sub $sp, 4
sw $t0, 4($sp)
li $t0, 3
sub $sp, 4
```

```
sw $t0, 4($sp)
lw $t2, 4($sp)
add $sp, 4
lw $t1, 0($fp)
li $t3, 1
mul $t2, $t2, $t3
add $t1, $t1, $t2
li $t3, 4
add $t1, $t1, $t3
lw $t2, 4($sp)
add $sp, 4
sb $t2, ($t1)
li $t0, 111
sub $sp, 4
sw $t0, 4($sp)
li $t0, 4
sub $sp, 4
sw $t0, 4($sp)
lw $t2, 4($sp)
add $sp, 4
```

```
lw $t1, 0($fp)
li $t3, 1
mul $t2, $t2, $t3
add $t1, $t1, $t2
li $t3, 4
add $t1, $t1, $t3
lw $t2, 4($sp)
add $sp, 4
sb $t2, ($t1)
li $t0, 32
sub $sp, 4
sw $t0, 4($sp)
li $t0, 5
sub $sp, 4
sw $t0, 4($sp)
lw $t2, 4($sp)
add $sp, 4
lw $t1, 0($fp)
li $t3, 1
mul $t2, $t2, $t3
```

Example III

```
add $t1, $t1, $t2
li $t3, 4
add $t1, $t1, $t3
lw $t2, 4($sp)
add $sp, 4
sb $t2, ($t1)
li $t0, 87
sub $sp, 4
sw $t0, 4($sp)
li $t0, 6
sub $sp, 4
sw $t0, 4($sp)
lw $t2, 4($sp)
add $sp, 4
lw $t1, 0($fp)
li $t3, 1
mul $t2, $t2, $t3
add $t1, $t1, $t2
li $t3, 4
add $t1, $t1, $t3
```

```
lw $t2, 4($sp)
add $sp, 4
sb $t2, ($t1)
li $t0, 111
sub $sp, 4
sw $t0, 4($sp)
li $t0, 7
sub $sp, 4
sw $t0, 4($sp)
lw $t2, 4($sp)
add $sp, 4
lw $t1, 0($fp)
li $t3, 1
mul $t2, $t2, $t3
add $t1, $t1, $t2
li $t3, 4
add $t1, $t1, $t3
lw $t2, 4($sp)
add $sp, 4
sb $t2, ($t1)
```

```
li $t0, 114
sub $sp, 4
sw $t0, 4($sp)
li $t0, 8
sub $sp, 4
sw $t0, 4($sp)
lw $t2, 4($sp)
add $sp, 4
lw $t1, 0($fp)
li $t3, 1
mul $t2, $t2, $t3
add $t1, $t1, $t2
li $t3, 4
add $t1, $t1, $t3
lw $t2, 4($sp)
add $sp, 4
sb $t2, ($t1)
li $t0, 108
sub $sp, 4
sw $t0, 4($sp)
```

Example IV

```
li $t0, 9
sub $sp, 4
sw $t0, 4($sp)
lw $t2, 4($sp)
add $sp, 4
lw $t1, 0($fp)
li $t3, 1
mul $t2, $t2, $t3
add $t1, $t1, $t2
li $t3, 4
add $t1, $t1, $t3
lw $t2, 4($sp)
add $sp, 4
sb $t2, ($t1)
li $t0, 100
sub $sp, 4
sw $t0, 4($sp)
li $t0, 10
sub $sp, 4
sw $t0, 4($sp)
```

```
lw $t2, 4($sp)
add $sp, 4
lw $t1, 0($fp)
li $t3, 1
mul $t2, $t2, $t3
add $t1, $t1, $t2
li $t3, 4
add $t1, $t1, $t3
lw $t2, 4($sp)
add $sp, 4
sb $t2, ($t1)
li $t0, 33
sub $sp, 4
sw $t0, 4($sp)
li $t0, 11
sub $sp, 4
sw $t0, 4($sp)
lw $t2, 4($sp)
add $sp, 4
lw $t1, 0($fp)
```

```
li $t3, 1
mul $t2, $t2, $t3
add $t1, $t1, $t2
li $t3, 4
add $t1, $t1, $t3
lw $t2, 4($sp)
add $sp, 4
sb $t2, ($t1)
li $t0, 10
sub $sp, 4
sw $t0, 4($sp)
li $t0, 12
sub $sp, 4
sw $t0, 4($sp)
lw $t2, 4($sp)
add $sp, 4
lw $t1, 0($fp)
li $t3, 1
mul $t2, $t2, $t3
add $t1, $t1, $t2
```

Example V

```
li $t3, 4
add $t1, $t1, $t3
lw $t2, 4($sp)
add $sp, 4
sb $t2, ($t1)
li $t0, 10
sub $sp, 4
sw $t0, 4($sp)
li $t0, 13
sub $sp, 4
sw $t0, 4($sp)
lw $t2, 4($sp)
add $sp, 4
lw $t1, 0($fp)
li $t3, 1
mul $t2, $t2, $t3
add $t1, $t1, $t2
li $t3, 4
add $t1, $t1, $t3
lw $t2, 4($sp)
```

```
L1:
```

```
add $sp, 4
sb $t2, ($t1)
lw $t0, 0($fp)
sub $sp, 4
sw $t0, 4($sp)
lw $a0, 4($sp)
add $sp, 4
jal entryPrintString
move $sp, $fp
lw $ra, -8($fp)
lw $fp, -4($fp)
j $ra
```

```
entryPrintInt:
li $v0, 1
syscall
exitPrintInt:
j $ra
```

```
entryPrintChar:
li $v0, 11
syscall
exitPrintChar:
j $ra

entryPrintString:
li $t0, 4
add $a0, $a0, $t0
li $v0, 4
syscall
exitPrintString:
j $ra

entryLength:
lw $v0, ($a0)
exitLength:
j $ra
```

Example VI

```
entryNew:
        mul $t0, $a0, $a1
        li $t1, 4
        add $t0, $t0, $t1
        move $a0, $t0
        li $v0, 9
        syscall
        sw $a1, ($v0)
exitNew:
        j $ra
```

Outline

Code generation is very naive

- A bit more efficient : use as many processor registers as possible to translate IR registers, and use only the stack when there isn't any processor registers available.
- More efficient : a static analysis to determine which temporary registers are used at the same time (*liveness* analysis), and then a graph coloring algorithm to associate only alive temporary registers to processor registers

Instruction selection

- The IR expresses one operation at each tree node
- In real processors, several of these can sometimes be combined : selecting the most efficient sequence of IR instructions to translate to assembly code is a classical optimization

Other topics

- Automatic memory management : garbage collection
- Other paradigms: object, functional, logic
- Other optimization techniques
- More static analyses (for optimization or correctness)
- …

Related classes

S3 – Analyses statique et dynamique

Améliorer la qualité des logiciels par :

- analyse statique : principes et mise en œuvre pratique avec des outils industriels tels que Frama-C, Astree) ;
- tests (analyse dynamique) : principes, techniques et pratiques de mise en œuvre ;
- et la collaboration des analyses statiques et dynamiques.

S3 – Initiation à la recherche

- Séminaire (6h) de l'équipe Langage Modèles et Vérification (LMV) autour du test (incluant relation test et preuve)
- Lecture d'articles scientifiques (langages & compilation) et projet recherche associé

Related classes

S4 – Programmation haute performance

- Comment tirer partie des performances processeurs et accélérateurs actuels ?
- Applications au calcul scientifique et au traitement d'images