Machine Code Generation

Cosmin E. Oancea
cosmin.oancea@diku.dk
Modified by Marco Valtorta (UofSC) for CSCE 531 Spring 2020

Department of Computer Science (DIKU)
University of Copenhagen

February 2018 IPS Lecture Slides
Structure of a Compiler

Program text

↓

Lexical analysis

↓

Symbol sequence

↓

Syntax analysis

↓

Syntax tree

↓

Type Checking

↓

Syntax tree

↓

Intermediate code generation

Binary machine code

↑

Assembly and linking

↑

Ditto with named registers

↑

Register allocation

↑

Symbolic machine code

↑

Machine code generation

↑

Intermediate code
1 Quick Look at MIPS

2 Intermediate vs Machine Code

3 Exploiting Complex Instructions
Intended Learning Objectives

Students should be able to:

- translate simple, imperative programs (supporting while loops and sequential (short-circuiting) boolean operators) to a low-level three-address intermediate representation (IR) code.

- apply (by hand) the pattern-based machine code generation technique in order to translate simple programs written in three-address IR code to machine code (MIPS).
Symbolic Machine Language

A text-based representation of binary code:

- more readable than machine code,
- uses labels as destinations of jumps,
- allows constants as operands,
- translated to binary code by *assembler* and *linker*.
Fast Introduction to MIPS

.data: the upcoming section is considered data,
.text: the upcoming section consists of instructions,
.global: the label following it is accessible from outside,
.asciiz "Hello": string with null terminator,
.space n: reserves n bytes of memory space,
.word w1, .., wn: reserves n words.

Mips Code Example: $ra = $31, $sp = $29, $hp = $28 (heap pointer)

<table>
<thead>
<tr>
<th>.data</th>
<th><em>stop</em>:</th>
</tr>
</thead>
<tbody>
<tr>
<td>val: .word 10, -14, 30</td>
<td>ori $2, $0, 10</td>
</tr>
<tr>
<td>str: .asciiz &quot;Hello!&quot;</td>
<td>syscall</td>
</tr>
<tr>
<td><em>heap</em>: .space 100000</td>
<td>main:</td>
</tr>
<tr>
<td>.text</td>
<td>la $8, val # ?</td>
</tr>
<tr>
<td>.global main</td>
<td>lw $9, 4($8) # ?</td>
</tr>
<tr>
<td>la $28, <em>heap</em></td>
<td>addi $9, $9, 4 # ?</td>
</tr>
<tr>
<td>jal main</td>
<td>sw $9, 8($8) #...</td>
</tr>
<tr>
<td>...</td>
<td>j <em>stop</em> #jr $31</td>
</tr>
</tbody>
</table>
# Fast Introduction to MIPS

**Mips Code Example: $ra = $31, $sp = $29, $hp = $28 (heap pointer)**

```assembly
.data
val: .word 10, -14, 30  main:
str: .asciiz "Hello!"
_heap_: .space 100000
.text
.global main
la $28, _heap_
jal main
...

_stop_: ori $2, $0, 10
syscall  # syscall 10
        # means exit
la $8, val  # loads start address
          # of array val in reg $8
lw $9, 4($8)  # loads the second element
          # of val in register $9
addi $9, $9, 4  # adds ct 4 to the second
               # element of val
sw $9, 8($8)  # and stores it in the
             # third element of val
j _stop_  # or jr $31 (to reg $31)
        # jumps to label _stop_
```

The third element of val, i.e., 30, is set to $-14 + 4 = -10$. 

---

7 / 25
1 Quick Look at MIPS

2 Intermediate vs Machine Code

3 Exploiting Complex Instructions
Intermediate and Machine Code Differences

- machine code has a limited number of registers,
- usually there is no equivalent to CALL, i.e., need to implement it,
- conditional jumps usually have only one destination,
- comparisons may be separated from the jumps,
- typically RISC instructions allow only small-constant operands.

The first issue will be solved by means of register allocation (Ch.8 [M]).

The second issue is solved in Ch.9 [M].
Two-Way Conditional Jumps

IF $c$ THEN $l_t$ ELSE $l_f$ can be translated to:

branch_if_cond $l_t$
jump $l_f$

If $l_t$ or $l_f$ follow right after IF-THEN-ELSE, we can eliminate one jump:

IF $c$ THEN $l_t$ ELSE $l_f$

$l_t$: 
... 

$l_f$: 

can be translated to:

branch_if_not_cond $l_f$
Comparisons

In many architectures the comparisons are separated from the jumps: first evaluate the comparison, and place the result in a register that can be later read by a jump instruction.

- In **MIPS** both = and ≠ operators can jump (beq and bne), but < (slt) stores the result in a general register.

- **ARM** and **X86**’s arithmetic instructions set a flag to signal that the result is 0 or negative, or overflow, or carry, etc.

- **PowerPC** and **Itanium** have separate boolean registers.
Constants

Typically, machine instructions restrict constants’ size to be smaller than one machine word:

- **MIPS32** uses 16 bit constants. For larger constants, `lui` is used to load a 16-bit constant into the upper half of a 32-bit register.

- **ARM** allows 8-bit constants, which can be positioned at any (even-bit) position of a 32-bit word.

Code generator checks if the constant value fits the restricted size:

  - if it fits: it generates one machine instruction (constant operand);
  - otherwise: use an instruction that uses a register (instead of a ct) generate a sequence of instructions that load the constant value in that register.

Sometimes, the same is true for the jump label.
Demonstrating Constants

let rec compileExp e vtable place =
    match e with
    | Constant (IntVal n, pos) ->
        if n < 0 then ...
        else if n < 65536 then
            [ Mips.LI (place, makeConst n) ]
        else
            [ Mips.LUI (place, makeConst (n div 65536))
              ; Mips.ORI (place, place, makeConst (n mod 65536)) ]

What happens with negative constants?
Demonstrating Constants

let rec compileExp e vtable place =
  match e with
  | Constant (IntVal n, pos) ->
    if n < 0 then ...
  else if n < 65536 then
    [ Mips.LI (place, makeConst n) ]
  else
    [ Mips.LUI (place, makeConst (n div 65536))
      ; Mips.ORI (place, place, makeConst (n mod 65536)) ]

What happens with negative constants?

let rec compileExp e vtable place =
  match e with
  Constant (IntVal n, pos) =>
    if n < 0 then
      compileExp (Negate (Constant (IntVal (-n), pos), pos)) vtable place
    else if n < 65536 then
      [ Mips.LI (place, makeConst n) ]
    else
      [ Mips.LUI (place, makeConst (n div 65536))
        ; Mips.ORI (place, place, makeConst (n mod 65536)) ]
Quick Look at MIPS

Intermediate vs Machine Code

Exploiting Complex Instructions
Exploiting Complex Instructions

Many architectures expose complex instructions that combine several operations (into one), e.g.,

- load/store instructions also involve address calculation,
- arithmetic instructions that scale one argument (by shifting),
- saving/restoring multiple registers to/from memory storage,
- conditional instructions (other besides jump).

In some cases: several IL instructions → one machine instruction. In other cases: one IL instruction → several machine instructions, e.g., conditional jumps.
MIPS Example

The two intermediate-code instructions:

\[
\begin{align*}
    t2 & := t1 + 116 \\
    t3 & := M[t2]
\end{align*}
\]

can be combined into one MIPS instruction (?)

\[
\text{lw } r3, 116(r1)
\]
MIPS Example

The two intermediate-code instructions:

\[
\begin{align*}
t2 & := t1 + 116 \\
t3 & := M[t2]
\end{align*}
\]

... can be combined into one MIPS instruction (?)

\[
lw r3, 116(r1)
\]

IFF \( t2 \) is not used anymore! Assume that we mark/know whenever a variable is used for the last time in the intermediate code.

This marking is accomplished by means of \textit{liveness analysis} (Ch.8 [M]); we write:

\[
\begin{align*}
t2 & := t1 + 116 \\
t3 & := M[t2^{\text{last}}]
\end{align*}
\]
Intermediate-Code Patterns

- Need to map each IL instruct to one or many machine instructs.
- Take advantage of complex-machine instructions via patterns:
  - map a sequence of IL instructs to one or many machine instructs,
  - try to match first the longer pattern, i.e., the most profitable one.
- Variables marked with last in the IL pattern must be matched with variables that are used for the last time in the IL code.
- The converse is not necessary, i.e., if a variable is not marked with last in the pattern, then it still may be matched by a variable used for the last time in IL.

\[
\begin{align*}
t &:= r_s + k \\
r_t &:= M[t^{last}] \\
lw & r_t, k(r_s)
\end{align*}
\]

$t$, $r_s$ and $r_t$ can match arbitrary IL variables, $k$ can match any (small) constant.
Patterns for MIPS (part 1)

<table>
<thead>
<tr>
<th></th>
<th>lw</th>
<th>rt, k(rs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>( t := r_s + k, )</td>
<td>lw ( r_t, k(r_s) )</td>
<td></td>
</tr>
<tr>
<td>( r_t := M[t^{last}] )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( r_t := M[r_s] )</td>
<td>lw ( r_t, 0(r_s) )</td>
<td></td>
</tr>
<tr>
<td>( r_t := M[k] )</td>
<td>lw ( r_t, k(R0) )</td>
<td></td>
</tr>
<tr>
<td>( t := r_s + k, )</td>
<td>sw ( r_t, k(r_s) )</td>
<td></td>
</tr>
<tr>
<td>( M[t^{last}] := r_t )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( M[r_s] := r_t )</td>
<td>sw ( r_t, 0(r_s) )</td>
<td></td>
</tr>
<tr>
<td>( M[k] := r_t )</td>
<td>sw ( r_t, k(R0) )</td>
<td></td>
</tr>
<tr>
<td>( rd := rs + rt )</td>
<td>add ( rd, rs, rt )</td>
<td></td>
</tr>
<tr>
<td>( rd := rt )</td>
<td>add ( rd, R0, rt )</td>
<td></td>
</tr>
<tr>
<td>( rd := rs + k )</td>
<td>addi ( rd, rs, k )</td>
<td></td>
</tr>
<tr>
<td>( rd := k )</td>
<td>addi ( rd, R0, k )</td>
<td></td>
</tr>
<tr>
<td>GOTO label</td>
<td>j ( label )</td>
<td></td>
</tr>
</tbody>
</table>

Must cover all possible sequences of intermediate-code instructions.
# Patterns for MIPS (part 2)

<table>
<thead>
<tr>
<th>Condition</th>
<th>MIPS Instruction</th>
<th>Label</th>
</tr>
</thead>
<tbody>
<tr>
<td>$r_s = r_t$</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>beq r_s, r_t, label_t</code></td>
<td><code>label_t:</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>bne r_s, r_t, label_f</code></td>
<td><code>label_f:</code></td>
</tr>
<tr>
<td>$r_s &lt; r_t$</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>slt r_d, r_s, r_t</code></td>
<td><code>label_f:</code></td>
</tr>
<tr>
<td></td>
<td><code>bne r_d, R0, label_t</code></td>
<td><code>label_t:</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>slt r_d, r_s, r_t</code></td>
<td><code>label_t:</code></td>
</tr>
<tr>
<td></td>
<td><code>beq r_d, R0, label_f</code></td>
<td><code>label_f:</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>slt r_d, r_s, r_t</code></td>
<td><code>label_f:</code></td>
</tr>
<tr>
<td></td>
<td><code>bne r_d, R0, label_t</code></td>
<td><code>label_t:</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>beq r_s, r_t, label_t</code></td>
<td><code>label_t:</code></td>
</tr>
<tr>
<td></td>
<td><code>bne r_s, r_t, label_f</code></td>
<td><code>label_f:</code></td>
</tr>
</tbody>
</table>

LABEL `label` | `label:`
Compiling Code Sequences: Example

\[
a := a + b^{\text{last}}
\]
\[
d := c + 8
\]
\[
M[d^{\text{last}}] := a
\]
\[
\text{IF } a = c \text{ THEN } label_1 \text{ ELSE } label_2
\]
\[
\text{LABEL } label_2
\]
Compiling Code Sequences

Example:

\[
\begin{align*}
    a & := a + b^{last} \\
    d & := c + 8 \\
    M[d^{last}] & := a \\
    \text{IF } a = c \text{ THEN } label_1 \text{ ELSE } label_2 \\
    \text{LABEL } label_2
\end{align*}
\begin{align*}
    \text{add } & a, a, b \\
    \text{sw } & a, 8(c) \\
    \text{beq } & a, c, label_1 \\
    \text{label}_2 & : 
\end{align*}
\]

Two approaches:

**Greedy Alg:** Find the first/longest pattern matching a prefix of the \text{IL} code + translate it. Repeat on the rest of the code.

**Dynamic Prg:** Assign to each machine instruction a cost and find the matching that minimize the global / total cost.
Two-Address Instructions

Some processors, e.g., X86, store the instruction’s result in one of the operand registers. Handled by placing one argument in the result register and then carrying out the operation:

\[
\begin{align*}
    r_t &:= r_s & \text{mov} & r_t, r_s \\
    r_t &:= r_t + r_s & \text{add} & r_t, r_s \\
    r_d &:= r_s + r_t & \text{move} & r_d, r_s \\
    & & \text{add} & r_d, r_t
\end{align*}
\]

Register allocation can remove the extra move.
Optimizations

Can be performed at different levels:

**Abstract Syntax Tree:** high-level optimization: specialization, inlining, map-reduce, etc.

**Intermediate Code:** machine-independent optimizations, such as redundancy elimination, or index-out-of-bounds checks.

**Machine Code:** machine-specific, low-level optimizations such as instruction scheduling and pre-fetching.

Optimizations at the intermediate-code level can be shared between different languages and architectures.
Code Hoisting

Code hoisting means moving common code in the body of a loop outside the loop, so that it executed only once, rather than every time the body is executed. This may require unrolling while loops once, to prevent executing code even when the condition of the loop is false; such execution could lead to run-time errors.

Original loop:

```plaintext
while (j < k) {
    sum = sum + a[i][j];
    j++;
}
```

After unrolling once:

```plaintext
if (j < k) {
    sum = sum + a[i][j];
    j++;
    while (j < k) {
        sum = sum + a[i][j];
        j++;
    }
}
```
Some Other Types of Optimization

Common Subexpression Elimination  Simple methods for common subexpression elimination work on *basic blocks*, i.e., straight-line code without jumps or labels. More on this in Chapter 10.

Constant Propagation

Index-Check Elimination