CA4003 - Compiler Construction

Code Generation
to MIPS

David Sinclair

Code Generation

The generation of machine code depends heavily on:

- the *intermediate representation*;and
- the target processor.

We are going to focus on the generation of *symbolic machine code* for the MIPS processor, but these concepts can be generalised to other processors.

Symbolic machine code is a text-based representation that is translated into *binary code* by an *assembler* and *linker*.

Symbolic machine code is:

- more readable than binary code;
- use label as destinations of jumps; and
- allows the use of constants as operands.

MIPS Revision

The `.data` directive defines the following section as data.

The `.text` directive defines the following section as instructions

The `.space n` directive reserves `n` bytes of memory. The directive `.word w1, w2, ..., wn` reserves and initialises `n` words.

The `.asciiz` directives defines a null terminated string, e.g. `.asciiz "Hello"`. Similarly, the `.ascii` directive defines a string without the null terminator.

The `.align n` directive causes the next data or code to be located at an address divisible by 2^n.

MIPS Revision (2)

```
int x;                        .data
char y, z;                    x:      .space 4
                              y:      .space 1
                              z:      .space 1
foo ( )
{                             .align 2
  ...
}                             .text
                              foo:
                                      code for foo
int a[10];
                              .data
                              a:      .space 40

bar ( )                       .text
{                             bar:
  ...                                 code for bar
}
```

MIPS Revision (3)

A constant *n* can be loaded into a register *r* using the load immediate instruction, e.g.

```
li      r, n
```

A scalar global variable can be loaded directly by name, e.g. to load an `int` into register $5

```
lw      $5, x
```

An element of a global array can be accessed by calculating the offset from the array's base address, e.g. to load A[i]

```
lw      r0, i          # if i was global
la      r1, A
sll     r0, r0, 2      # scale by 4
add     r1, r1, r0
lw      r2, (r1)
```

MIPS Revision (4)

The MIPS instructions to load an 8-bit character at address addr into a 32-bit register reg, use the `lb reg, addr` instruction. Similarly, storing a 32-bit register reg into an 8-bit char at address addr, use `sb addr, reg`.

In MIPS, a constant's size is restricted to 16-bits. If a larger constant is needed, the `lui` instruction is used to load a 16-bit constant into the upper half of a 32-bit register. Then another 16-bit constant can be ored with the register to form a 32-bit constant.

Intermediate Code Patterns

Each *intermediate code* instruction, in our case each 3-address code, is mapped to one or more MIPS instructions.

To take advantage of complex machine instructions, we use *patterns* that map a *sequence of intermediate code instructions* to one or more machine instructions. We try to map the longer patterns first.

Variables marks the *last* in an *intermediate code pattern* **must** be the last use of that variable in the *intermediate code*.

In the following, t, r_d, r_s and r_t can match arbitrary variables and k can match any constant. These are just a **subset** of possible patterns. It is important that at least one pattern can match any *intermediate code* instruction.

Arithmetic Operations

$r_d := r_s + r_t$	`add` r_d, r_s, r_t
$r_d := r_t$	`add` $r_d, \$R0, r_t$
$r_d := r_s + k$	`addi` r_d, r_s, k
$r_d := k$	`addi` $r_d, \$R0, k$
$r_d := r_s - r_t$	`sub` r_d, r_s, r_t
$r_d := r_s - k$	`addi` $r_d, r_s, -k$
$r_d := r_s * r_t$	`mul` r_s, r_t
$r_d := r_s * k$	`mul` r_s, k
$r_d := r_s / r_t$	`div` r_s, r_t
$r_d := r_s / k$	`div` r_s, k

The results of `mul` and `div` are store in 2 special registers `$hi` and `$lo`. For `mul` `$hi` has the most significant word and `$lo` has the least significant word. For `div`, `$lo` has the quotient and `$hi` has the remainder.

Jumps

`goto` *label*	`j` *label*
`label` *lbl*	*lbl*:
`if` $r_s = r_t$ `then` $label_t$ `else` $label_f$ `label` $label_f$	`beq` r_s, r_t, $label_t$ $label_f$:
`if` $r_s = r_t$ `then` $label_t$ `else` $label_f$ `label` $label_t$	`bne` r_s, r_t, $label_f$ $label_t$:
`if` $r_s = r_t$ `then` $label_t$ `else` $label_f$	`beq` r_s, r_t, $label_t$ `j` $label_f$
`if` $r_s < r_t$ `then` $label_t$ `else` $label_f$ `label` $label_f$	`slt` r_d, r_s, r_t `bne` r_d, \$R0, $label_t$ $label_f$:
`if` $r_s < r_t$ `then` $label_t$ `else` $label_f$ `label` $label_t$	`slt` r_d, r_s, r_t `beq` r_d, \$R0, $label_f$ $label_t$:
`if` $r_s < r_t$ `then` $label_t$ `else` $label_f$	`slt` r_d, r_s, r_t `bne` r_d, \$R0, $label_t$ `j` $label_f$

Jumps (2)

`if` r_s *relop* r_t `then` *lbl*	`b`*cc* r_s, r_t, *lbl*

where the condition code *cc* is given by:

op	cc	op	cc	op	cc
<=	le	<	lt	!=	ne
==	eq	>	gt	>=	ge

Additional specialist patterns can be developed for other operators than $=$ and $<$.

Arrays

To read the i^{th} element of a global array x into reg_3, where each element is 2^n bytes wide, we can use:

```
load the value of i into reg_1
load the starting address of x into reg_2
sll   reg_1, reg_2, n      #scale for width unless n = 0
add   reg_2, reg_1, 0(reg_2)
lw    reg_3, 0(reg_2)
```

If the element is only 1 byte wide, use `lb` instead of `lw`.

To store a value at index i of global array x, use the `sw`, or `sb`, instruction instead of `lw`, or `lb`.

Arrays - Example

Let x be a global array of integers, y an array of characters at offset 40 from `$sp` and i and j be global integers. Then the source language statement `y[i] = x[j]`, could translate to the 3-address code sequence `tmp= x[j]; y[i] = tmp`, where `tmp` is a compiler generated temporary that has spilled onto the stack at offset 24.

This then generates the following machine code

```
lw   $t0, j
la   $t1, x           # load starting address of x
sll  $t0, $t0, 2
add  $t1, $t0, $t1    # $t1 is addr of x[j]
lw   $t2, ($t1)
sw   $t2, 24($sp)     # tmp = x[j]

lw   24($sp), $t0
lw   $t1, i
la   $t2, 40($sp)
add  $t2, $t1, $t2    # $t2 is addr of y[i]
sb   $t0, ($t2)       # y[i] = tmp
```

Procedure Calls

We discuss earlier how the *callee* sets up the stack frame on entry to a procedure. Here we give the code generated for the *caller*.

`param x` (x is an int/char)	*load x into* reg_1
	`sw` reg_1`,-4($sp)`
	`la $sp,-4($sp)`

`param x` (x is an array)	*load address of x into* reg_1
	`sw` reg_1`,-4($sp)`
	`la $sp,-4($sp)`

`call p, n`	`jal p`
	`la $sp,`k`($sp)`
	where $k = 4 * n$

Building Code Sequences

There can be more than one matching pattern. There are 2 algorithmic approaches:

- **Greedy** Find the first longest pattern at the start of the code, translate it and continue with the remaining code.
- **Dynamic** Assign a cost to each machine instruction and find the match that minimises the total cost using dynamic programming.

Also the order in which the patterns appear is important! If we are adopting the *greedy algorithm*, first longest matching pattern, then we need to arrange the patterns so that when two patterns overlap, the longest pattern is listed first.

Example

Generate symbolic machine code for the following 3-address code sequence

$a = a + b^{last}$
$d = c + 8$
$M[d^{last}] = a$
if $a = c$ **then** lbl_1 **else** lbl_2
label lbl_2

using the following subset of patterns and the *Greedy Matching Algorithm*.

$t = r_s + k$ $r_t = M[t^{last}]$	`lw` r_t, `k(`r_s`)`
$r_t = M[r_s]$	`lw` r_t, `0(`r_s`)`
$r_t = M[k]$	`lw` r_t, `0(`r_s`)`
$t = r_s + k$ $M[t^{last}] = r_r$	`sw` $r_t, k(r_s)$

Example (2)

$M[r_s] = r_t$	`sw` r_t, `0(`r_s`)`
$M[k] = r_t$	`sw` r_t, `k($0)`
$r_d = r_s + r_t$	`add` r_d, r_s, r_t
$r_d = r_s$	`add` r_d, `$0`, r_t
$r_d = r_s + k$	`addi` r_d, r_s, `k`
$r_d = k$	`addi` r_d, `$0`, `k`
goto *label*	`j` *label*
if $r_s = r_t$ **then** lbl_t **else** lbl_f **label** lbl_f	`beq` r_s, r_t, lbl_t lbl_f:
if $r_s = r_t$ **then** lbl_t **else** lbl_f **label** lbl_t	`bne` r_s, r_t, lbl_f lbl_t:
if $r_s = r_t$ **then** lbl_t **else** lbl_f	`beq` r_s, r_t, lbl_t `j` lbl_f
if $r_s < r_t$ **then** lbl_t **else** lbl_f **label** lbl_f	`slt` r_d, r_s, r_t `bne` r_d, `$0`, lbl_t lbl_f:

Example (3)

if $r_s < r_t$ **then** lbl_t **else** lbl_f **label** lbl_t	`slt` r_d, r_s, r_t `beq` r_d, \$0, lbl_f lbl_t:
if $r_s < r_t$ **then** lbl_t **else** lbl_f	`slt` r_d, r_s, r_t `bne` r_d, \$0, lbl_t `j` lbl_f
label *label*	*label* :

Only one pattern matches the start (prefix) of our example and generates an `add` instruction. The prefix of the remaining 3-address code lines is matched by 2 patterns. We choose the first longest pattern. The remainder of the code is matched by another pattern.

```
        add   a, a, b
        sw    a, 8(c)
        beq   a, c, lbl1
lbl2:
```