Instruction selection (1)

At some point during the compilation process, the intermediate representation of the code will be translated into the instruction set of the target machine

The translation is built by selecting instructions from the target architecture to implement the operations encoded in the intermediate representation

Translating involves several choices, and may be either a simple process or a more complex one

Instruction selection (2)

Complexity of the translation

The translation may follow a simple instruction by instruction strategy, translating each individual intermediate representation instruction in isolation

Improving the generated code will then be up to a subsequent optimisation phase

The translation process may be more sophisticated and try to generate better code directly

In this case, the final optimisation phase may be simpler

In any event, for every intermediate representation instruction, there must be a translation into target machine instructions

Instruction selection (3)

Choice of instructions

The target instruction set is usually rich enough to provide several ways of encoding the operations that must be performed

For each one of those operations, a sequence of target instructions that implements it must be chosen

The choice is made based on the costs of functionally equivalent alternatives

NOTE:

MIPS will be used as the target instruction set in these slides

Instruction selection (4)

Instruction sequence cost

Size

The cost may be the length of the sequence, i.e., the number of instructions in the sequence

It this case, the goal is to minimise the code size

Speed

The cost may be the execution time of the sequence, which may not be directly related to its length

It this case, the goal is to obtain fast code

Power

The cost may be related to the power used by the sequence

It this case, the goal is to minimise the power consumption during execution, which is important for embedded devices

Introduction to the MIPS architecture (1)

MIPS is a RISC architecture

MIPS32 is the 32 bits MIPS version, and everything in these slides pertains to this version, unless otherwise noted

MIPS has 32 general purpose (32 bits, integer) registers and 32 single precision (32 bits) floating point registers (which double as 16 double precision (64 bits) floating point registers)

Every MIPS instruction is 32 bits long

Addresses are 32 bits long

Introduction to the MIPS architecture (2)

Instructions operate on values which are either in registers or contained within the instruction itself (in the immediate field), and their result goes into a register

The exceptions to this rule include

- The instructions that load a value from memory
- The instructions that store a value into memory
- The jump and conditional branch instructions, which alter the flow of control

Jump and conditional branch instructions have one delay slot (the instruction following the jump or the conditional branch is always executed)

- Delay slots will be mostly ignored in what regards instruction selection

Introduction to the MIPS architecture (3)

Typical MIPS instructions

Instruction	Operation
`add` r_1, r_2, r_3	$r_1 \leftarrow r_2 + r_3$
`addi` r_1, r_2, n	$r_1 \leftarrow r_2 + n$
	Immediate instruction, n is a signed 16 bit integer
`addiu` r_1, r_2, n	$r_1 \leftarrow r_2 + n$
	Immediate instruction, n is a signed 16 bit integer
`ori` r_1, r_2, n	$r_1 \leftarrow r_2 \mid n$ (bitwise OR)
	Immediate instruction, n is an unsigned 16 bit integer
`j` l	$\mathrm{PC} \leftarrow l$
`jal` l	`$ra` $\leftarrow \mathrm{PC} + 4$; $\mathrm{PC} \leftarrow l$
	Calls function l, after saving the return address in `$ra`
`jr` r_1	$\mathrm{PC} \leftarrow r_1$
	Used to return control from a function
`beq` r_1, r_2, l	If $r_1 = r_2$ then $\mathrm{PC} \leftarrow l$, else $\mathrm{PC} \leftarrow \mathrm{PC} + 4$
`bne` r_1, r_2, l	If $r_1 \neq r_2$ then $\mathrm{PC} \leftarrow l$, else $\mathrm{PC} \leftarrow \mathrm{PC} + 4$

Introduction to the MIPS architecture (4)

Typical MIPS instructions (cont.)

Instruction	Operation
`slt` r_1, r_2, r_3	$r_1 \leftarrow r_2 < r_3 \ ?\ 1 : 0$
`lui` r_1, n	$r_1 \leftarrow n \times 2^{16}$
	n is an unsigned 16 bit integer
`lw` $r_1, \mathit{offset}(r_2)$	$r_1 \leftarrow \mathrm{MEM}[r_2 + \mathit{offset}]$
	Reads a (32 bit) word from memory
	offset is a signed 16 bit integer
`sw` $r_1, \mathit{offset}(r_2)$	$\mathrm{MEM}[r_2 + \mathit{offset}] \leftarrow r_1$
	Writes a (32 bit) word to memory

Remarks

r_1, r_2 and r_3 stand for any general purpose registers

PC is the processor's program counter

Immediate instructions embed a 16 bit integer value which may be either signed or unsigned, depending on the instruction

Introduction to the MIPS architecture (5)

General purpose registers

MIPS general purpose registers are numbered $0 through $31

There is a widely followed convention on the use of MIPS registers

Name	Number	Role
$0 or $zero	0	Always contains the value 0
$at	1	Reserved for use by the assembler
$v0 and $v1	2–3	Function return value(s)
$a0–$a3	4–7	Function or procedure arguments
$t0–$t9	8–15, 24–25	Caller-saved temporary values
$s0–$s7 *	16–23	Callee-saved temporary values
$k0 and $k1	26-27	Reserved for the operating system
$gp *	28	Global pointer (base address of the global variables memory area)
$sp *	29	Stack pointer
$fp *	30	Frame pointer
$ra *	31	Return address

* Callee-saved registers

Introduction to the MIPS architecture (6)

Caller-saved registers

The contents of caller-saved registers are not guaranteed to be preserved across a function call, according to the convention (any function or procedure is free to use those registers as it pleases)

If a function or procedure uses a caller-saved register and needs the value it contains after calling some function or procedure, then it, the caller, must save its contents before performing the call (in a memory location within its activation record, for example)

Introduction to the MIPS architecture (7)

Callee-saved registers

At the end of a function or procedure, the contents of callee-saved registers must be what they were when the function or procedure started executing (from the point of view of the caller, their contents are preserved across function calls)

If a function or procedure uses any of these registers, then it, the callee, must save its contents before modifying them (in a memory location within its activation record, for example) and restore them before finishing executing

Introduction to the MIPS architecture (8)

Special general purpose registers

The only thing special about `$0` is that its value cannot be changed

`$ra` (a.k.a. `$31`) is set by the MIPS function calling instructions, such as `jal`

Function calling convention

One MIPS function calling convention includes passing the first 4 argument (32 bit) words in registers `$a0` to `$a3`

If the return value of a function fits in 32 bits, it is returned in register `$v0`

If it is 64 bits long, the least significant half of the result is returned in register `$v1`

Introduction to the MIPS architecture (9)

Floating point registers

MIPS floating point registers are numbered `$f0` through `$f31`

When using floating point registers for double precision floating point numbers, only the even numbered registers may be used: `$f0`, `$f2`, `$f4`, ..., `$f30` (`$f0` represents the pair `$f0`-`$f1`)

TACL reals are double precision, so only even numbered registers will be used

Name	Role
`$f0`–`$f2`	Function return value(s)
`$f4`–`$f10`	Caller-saved registers
`$f12`–`$f14`	Function or procedure arguments
`$f16`–`$f18`	Caller-saved registers
`$f20`–`$f30`	Callee-saved registers

Introduction to the MIPS architecture (10)

MIPS64 is the 64 bit version of the MIPS architecture

All 32 general purpose and 32 floating point registers are 64 bits wide

Addresses are 64 bits long

Instructions are still 32 bits long

Every MIPS32 instruction works the same when run in a MIPS64 implementation

A MIPS32 system may have a 64 bit FPU (i.e., a Floating Point Unit where all 32 registers are 64 bits wide)

Simple translations (1)

This section presents translations of some IR instructions into MIPS instructions

It also introduces some of the problems faced when deciding how to implement the IR instructions in the target instruction set

No optimisations whatsoever are considered

Simple translations (2)

i_add

The IR instruction

$$t_3 \leftarrow \text{i\_add } t_3, t_2$$

may be directly implemented by the MIPS instructions

$$\texttt{add } t_3, t_3, t_2$$

and

$$\texttt{addu } t_3, t_3, t_2$$

The only difference between these instructions is that the first one raises an exception if the result of the operation overflows

Other instruction sequences could achieve the same result, but they all would include one of the instructions above and nothing would be gained by using them

Simple translations (3)

i_add (cont.)

The first choice to make is whether arithmetic operations in TACL may raise overflow exceptions or not

The answer is no (*à la* C) and hereafter the instructions used will be those that do not

So, the default translation of an IR instruction of the form

$$t_i \leftarrow \text{i\_add } t_j, t_k$$

can be

$$\texttt{addu } t_i, t_j, t_k$$

Simple translations (4)

i_sub

Similarly to the i_add instruction, the i_sub instruction

$$t_1 \leftarrow \text{i\_sub } t_1, t_4$$

may be translated to

$$\texttt{subu } t_1, t_1, t_4$$

Simple translations (5)

i_value

The i_value instruction puts an integer value into a temporary location

This may be achieved with an immediate MIPS instruction, such as

```
addi     addiu     ori
```

For instance,

$$t_4 \leftarrow \text{i\_value } 1$$

may be translated to one of

```
addi  t4, $0, 1
addiu t4, $0, 1
ori   t4, $0, 1
```

(When adding to 0, no overflow is possible)

But...

Simple translations (6)

i_value (cont.)

The immediate field of the MIPS instructions is only 16 bits long

If the binary representation of the integer value is longer than 16 bits, two instructions will be needed to implement the operation

For example

$$t_4 \leftarrow \text{i\_value } 6543210$$

would be translated to the sequence

```
lui t4, 99
ori t4, t4, 55146
```

since $6543210 = 63d76a_{16} = 99 \times 2^{16} + 55146$

The add instructions could not be used here, because they sign-extend the value in the immediate field

Simple translations (7)

i_mul

Computing the product of two integers in MIPS requires two instructions

The MIPS multiplication instructions leave the result of the operation in the two special registers hi (the 32 most significant bits of the result) and lo (the 32 least significant bits)

Since TACL works with 32 bit integers, the IR instruction

$$t_6 \leftarrow \text{i\_mul } t_4, t_5$$

may be implemented by the two instruction sequence

$$\texttt{mult } t_4, t_5$$
$$\texttt{mflo } t_6$$

(Notes: signed and unsigned multiplications are different; overflow is not detected; at some point, a '`mul` t_i, t_j, t_k' instruction has been added)

Simple translations (8)

i_div, mod

The quotient and the remainder of the division of two signed integers are computed by the same MIPS instruction

The quotient is left in the lo register and the remainder in hi

The IR instructions

$$t_8 \leftarrow \text{i\_div } t_6, t_7 \qquad \text{and} \qquad t_8 \leftarrow \text{mod } t_6, t_7$$

may be implemented, respectively, by the two instruction sequences

`div` t_6, t_7 `mflo` t_8	and	`div` t_6, t_7 `mfhi` t_8

(Notes: MIPS does not detect division by 0; recently, separate `div` and `mod` instructions with 3 register arguments were added to the architecture)

Simple translations (9)

i_lt

MIPS has the following instruction

$$\texttt{slt}\ t_i, t_j, t_k$$

which sets t_i to 1 if the value in t_j is less than the one in t_k, and to 0 otherwise

Having the above instruction, the IR instruction

$$t_8 \leftarrow \text{i\_lt}\ t_6, t_7$$

may be translated as

$$\texttt{slt}\ t_8, t_6, t_7$$

Simple translations (10)

jump

An IR jump l_i instruction can be translated to the MIPS jump instruction

$$\texttt{j}\ l_i$$

cjump

MIPS conditional branch instructions only have one label

If the branch is not taken, the execution falls through to the next instruction and

$$\text{cjump}\ t_5, l_1, l_2$$

may be implemented by any of the two two-instruction sequences

`bne` t_5, `$0`, l_1 `j` l_2	and	`beq` t_5, `$0`, l_2 `j` l_1

Simple translations (11)

i_gload, i_gstore

At this stage, it is too soon to decide the location of global variables in memory; which is a task better left to the assembler

Every integer or boolean global variable will be defined in the data section of the assembly file and the assembler will determine its address

Example (20)

A global declaration like `var int n = 26;` will lead to the assembly declaration

```
      .data
n:    .word 26
```

Simple translations (12)

i_gload, i_gstore (cont.)

An IR instruction like

$$t_1 \leftarrow \text{i\_gload } \texttt{@}n$$

may be translated to the pseudo-instruction

$$\texttt{lw } t_1, n$$

which the assembler will translate into the appropriate MIPS instructions

The same is valid for i_gstore instructions, where `sw` will replace `lw`

Simple translations (13)

itor

Converting an integer to a floating point number is a two step operation

The first step consists in moving the integer value into a floating point register

In the second step, the integer value is converted

$$fp_1 \leftarrow \text{itor } t_6$$

is implemented by the sequence

```
mtc1     t6, fp1
cvt.d.w  fp1, fp1
```

Code generation for functions (1)

The code for a function or procedure must include code which implements the calling convention for functions and manages the program's stack

This code is distributed among the caller function and the callee

The caller must, at least, place the arguments of the called function somewhere from where the latter will be able to fetch them, transfer the execution to the called function code and set the return address

Once the callee returns, the caller may fetch the result from some predefined location

It may also be up to the caller to finish restoring the stack to the state it was in before the call

Code generation for functions (2)

The code for a function or procedure will include a prologue and an epilogue

Prologue

Creates the activation record for the function or procedure, or completes its creation, in case the caller has already done part of the job

Epilogue

Puts the function result somewhere where the caller will be able to access it

Does the callee's part in what concerns the elimination of its activation record

Transfers execution to the function or procedure return address

Code generation for functions (3)

Calling a function or procedure involves executing the following code

Instructions	Execution context
Pre-call sequence	Caller
Call	Caller $\rightarrow$ callee
Prologue Function or procedure body Epilogue	Callee
Function or procedure return	Callee $\rightarrow$ caller
Post-call sequence	Caller

Simple translations — Extended example (1)

Consider the following definition and its intermediate representation on the next slide

TACL code

```
fun int factorial(int n)
[
  var int r = 1;

  if (n > 0)
    r = n * factorial(n - 1);

  ^ r
]
```

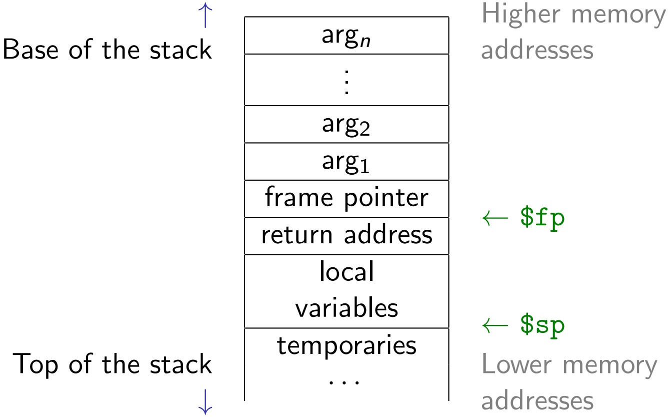
Simple translations — Extended example (2)

Intermediate representation

```
        t0 <- i_value 1
        @r <- i_lstore t0
        t1 <- i_aload @n
        t2 <- i_value 0
        t3 <- i_lt t2, t1
        cjump t3, l0, l1
l0:     t4 <- i_aload @n
        t5 <- i_aload @n
        t6 <- i_value 1
        t7 <- i_sub t5, t6
        t8 <- i_call @factorial, [t7]
        t9 <- i_mul t4, t8
        @r <- i_lstore t9
l1:     t10 <- i_lload @r
        i_return t10
```

Simple translations — Extended example (3)

Contents of the activation records (a.k.a. stack frames) used in this example



Arguments are passed to the function in the stack frame

The function result is returned in register `$v0` (or `$f0`)

The `$sp` register contains the address of the top of the stack

Simple translations — Extended example (4)

In this function calling convention, the prologue

- Saves the caller's frame pointer
- Initialises the callee's frame pointer
- Saves the callee's return address (only needed if the function or procedure calls some function or procedure)
- Allocates room in the stack for local variables and temporary locations used for spilling registers

As for the epilogue, it

- Copies the function result to `$v0` (only inside a function)
- Restores the callee's return address
- Pops the callee's activation record
- Restores the caller's frame pointer
- Terminates the execution of the function or procedure, by jumping to the return address

Simple translations — Extended example (5)

The pre-call instruction sequence pushes the call actual arguments onto the stack, from last to first (which corresponds to start assembling the callee's activation record)

Since the callee pops its activation record from the stack, the post-call instruction sequence will only contain instructions to fetch the value returned by a function

Simple translations — Extended example (6)

MIPS instructions, still with IR temporaries

```
factorial:
        sw    $fp, -4($sp)
        addiu $fp, $sp, -4
        sw    $ra, -4($fp)
        addiu $sp, $fp, -8
        ori   t0, $0, 1
        sw    t0, -8($fp)
        lw    t1, 4($fp)
        ori   t2, $0, 0
        slt   t3, t2, t1
        beq   t3, $0, l1
        j     l0
l0:     lw    t4, 4($fp)
        lw    t5, 4($fp)
        ori   t6, $0, 1
        subu  t7, t5, t6
        addiu $sp, $sp, -4
        sw    t7, 0($sp)
        jal   factorial
        or    t8, $0, $v0
        mult  t4, t8
        mflo  t9
        sw    t9, -8($fp)
l1:     lw    t10, -8($fp)
        or    $v0, $0, t10
        lw    $ra, -4($fp)
        addiu $sp, $fp, 8
        lw    $fp, 0($fp)
        jr    $ra
```

Simple translations — Extended example (7)

MIPS code, with registers (and some comments)

```
factorial:
        sw    $fp, -4($sp)       # save caller frame pointer
        addiu $fp, $sp, -4       # set callee frame pointer
        sw    $ra, -4($fp)       # save return address
        addiu $sp, $fp, -8       # allocate space for locals
                                 # (plus the return address)
        ori   $t0, $0, 1
        sw    $t0, -8($fp)       # r = 1
        lw    $t0, 4($fp)
        ori   $t1, $0, 0
        slt   $t0, $t1, $t0      # n > 0?
        beq   $t0, $0, l1
        j     l0
```

(continues. . .)

Simple translations — Extended example (8)

MIPS code, with registers (cont.)

```
l0:  lw    $t0, 4($fp)       # n
     lw    $t1, 4($fp)
     ori   $t2, $0, 1
     subu  $t1, $t1, $t2     # n - 1

     addiu $sp, $sp, -4      # save $t0 value
     sw    $t0, 0($sp)

     addiu $sp, $sp, -4      # push argument
     sw    $t1, 0($sp)
     jal   factorial         # call function
     or    $t1, $0, $v0      # get call return value
```

(continues. . .)

Simple translations — Extended example (9)

MIPS code, with registers (cont.)

```
      lw    $t0, 0($sp)          # restore $t0 value
      addiu $sp, $sp, 4

      mult  $t0, $t1
      mflo  $t0                  # n * factorial(n - 1)
      sw    $t0, -8($fp)         # r = ...
l1:   lw    $t0, -8($fp)
      or    $v0, $0, $t0         # set return value

      lw    $ra, -4($fp)         # restore return address
      addiu $sp, $fp, 8          # restore caller stack pointer
      lw    $fp, 0($fp)          # restore caller frame pointer
      jr    $ra                  # return from function
```

Simple translations — Extended example (10)

Simple register allocation

If the IR is generated following a postfix tree walk over the AST, with no attempts at optimisation, most temporaries have very short live ranges

Furthermore

1. Temporaries are created once and used once*
2. For every two temporaries, either their live ranges are disjoint or one is contained within the other
3. No live range crosses a basic block boundary*

These characteristics make it possible to allocate registers for temporaries in one pass through the IR, provided enough registers are available

* More about this later...

Simple translations — Extended example (11)

Algorithm for simple register allocation

1. Let $\{r_0, r_1, \ldots, r_{K-1}\}$ be the registers available and let $n = 0$ be the number of registers currently in use
2. For every IR instruction i
   a. If i uses any temporary, decrement n by the number of temporaries it uses
   b. If i is a call instruction and $n > 0$, emit code to save registers $r_0, \ldots, r_{n-1}$ before the call
   c. If i creates a new value, assign the temporary to register r_n and increment n by 1
   d. Emit the code for i, replacing the temporaries by the registers they were assigned to
   e. Emit code to restore the values of the registers saved in step b.

Simple translations — Extended example (12)

Notes about the algorithm

This algorithm is similar to the Sethi-Ullman algorithm which, besides, tries to minimise the number of registers used (For this, two passes over the IR are needed)

If $n \geq K$ before incrementing it in Step 2.c., then a register must be spilled

+ The best register to spill is the one with the lowest number that is not currently spilled, which is the one whose use is farther away
+ When the temporary whose register has been spilled is finally used, it must be loaded from memory into its assigned register (which should be free at the time)

Simple translations — Extended example (13)

Preserving caller-saved registers

When `factorial` is called recursively, there is one register holding a value that will be needed after the call

That register is `$t0` (identified in step 2.b. of the algorithm), which is a caller-saved register

Saving the value of `$t0` is effected by the instructions below, which store it within the stack

```
addiu $sp, $sp, -4      # save $t0 value
sw    $t0, 0($sp)
```

After the call, the register value must be restored, which is done by the instructions

```
lw    $t0, 0($sp)       # restore $t0 value
addiu $sp, $sp, 4
```

Simple translations — Extended example (14)

When live ranges cross basic blocks boundaries

The IR on the right corresponds to the following line of code, from the `fibonacci (v2)` function

```
c = n == 0 || n == 1;
```

In that IR, the t_2 temporary is live across 3 different basic blocks, and is created and used more than once

$$
\begin{array}{rl}
 & t_0 \leftarrow \text{i\_aload } @n \\
 & t_1 \leftarrow \text{i\_value } 0 \\
 & t_2 \leftarrow \text{i\_eq } t_0, t_1 \\
 & \text{cjump } t_2, l_0, l_1 \\
l_1 : & t_3 \leftarrow \text{i\_aload } @n \\
 & t_4 \leftarrow \text{i\_value } 1 \\
 & t_5 \leftarrow \text{i\_eq } t_3, t_4 \\
 & t_2 \leftarrow \text{i\_copy } t_5 \\
l_0 : & @c \leftarrow \text{i\_lstore } t_2
\end{array}
$$

To be correct, the generated code must use the same register for every occurrence of t_2

To guarantee that it happens, the register to which t_2 is assigned should not be freed until the last line where t_2 appears

To know which is that line requires two passes over the IR

The Sethi-Ullman algorithm (1)

The Sethi-Ullman algorithm is an algorithm for register allocation for expressions

It starts by computing the minimum number of registers needed to evaluate an expression

- Depicted in procedure label

Guided by that information, it then generates code for evaluating the expression, handling the spilling of registers, if an expression needs more than the K registers available

- Depicted in procedure Sethi-Ullman

The Sethi-Ullman algorithm (2)

Procedure label(IR e)

1. If e is an i_value or i_*load instruction
   - e.needs ← 1
2. Otherwise, if e is an i_inv or not instruction
   - 2.1 Let e_1 be e's operand
   - 2.2 label(e_1)
   - 2.3 e.needs ← e_1.needs
3. Otherwise, e is a two operand arithmetic or comparison instruction
   - 3.1 Let e_1 and e_2 be e's left and right operands, respectively
   - 3.2 label(e_1)
   - 3.3 label(e_2)
   - 3.4 If e_1.needs and e_2.needs are equal
     - e.needs ← e_1.needs + 1

     Otherwise
     - e.needs ← max(e_1.needs, e_2.needs)

The Sethi-Ullman algorithm (3)

Procedure Sethi-Ullman(IR e)

1. If e is a two operand instruction
   1.1 Let e_1 and e_2 be e's left and right operands, respectively
   1.2 If e_1.needs $\geq K$ and e_2.needs $\geq K$
      1.2.1 Sethi-Ullman(e_2)
      1.2.2 $n \leftarrow n - 1$
      1.2.3 Spill e_2.register
      1.2.4 Sethi-Ullman(e_1)
      1.2.5 e_2.register $\leftarrow r_{n+1}$
      1.2.6 Restore e_2.register value
   1.3 Otherwise, if e_1.needs $\geq$ e_2.needs
      1.3.1 Sethi-Ullman(e_1)
      1.3.2 Sethi-Ullman(e_2)
      1.3.3 $n \leftarrow n - 1$
   1.4 Otherwise, e_1.needs $<$ e_2.needs
      1.4.1 Sethi-Ullman(e_2)
      1.4.2 Sethi-Ullman(e_1)
      1.4.3 $n \leftarrow n - 1$

(continues. . .)

The Sethi-Ullman algorithm (4)

Procedure Sethi-Ullman(IR e) (cont.)

2. Otherwise, if e is an i_inv or not instruction
   2.1 Let e_1 be e's operand
   2.2 Sethi-Ullman(e_1)
3. Otherwise, e is an i_value or i_*load instruction
   - $n \leftarrow n + 1$
4. e.register $\leftarrow r_n$
5. Emit code for e's top level operation

n is a global variable which contains the number of the last register in use, and whose initial value is 0

(Note that if registers go from r_0 to r_{K-1}, n's initial value must be -1)

Example IR

Example (21)

The IR code on the right will be used to motivate some of the examples that follow

$$
\begin{array}{rl}
 & t_1 \leftarrow \text{i\_aload } @a \\
 & t_2 \leftarrow \text{i\_aload } @b \\
 & t_3 \leftarrow \text{i\_value } 0 \\
l_1: & t_3 \leftarrow \text{i\_add } t_3, t_2 \\
 & t_4 \leftarrow \text{i\_value } 1 \\
 & t_1 \leftarrow \text{i\_sub } t_1, t_4 \\
 & t_5 \leftarrow \text{i\_value } 0 \\
 & t_5 \leftarrow \text{i\_lt } t_5, t_1 \\
 & \text{cjump } t_5, l_1, l_2 \\
l_2: & \text{i\_return } t_3
\end{array}
$$

Not so simple translations (1)

In the code from the example, the following instruction sequence appears

$$t_4 \leftarrow \text{i\_value } 1$$
$$t_1 \leftarrow \text{i\_sub } t_1, t_4$$

Applying the simple instruction selection scheme, it would be translated to

```
ori   t4, $0, 1
subu  t1, t1, t4
```

But this could be more efficiently implemented by the MIPS instruction

```
addiu t1, t1, -1
```

Not so simple translations (2)

Another case is the sequence on the left below, for which a possible naive translation is shown on the right

IR	MIPS code
$t_5 \leftarrow$ i_value 0	`ori` t_5, `$0`, 0
$t_5 \leftarrow$ i_lt t_5, t_1	`slt` $t_5,\ t_5,\ t_1$
cjump t_5, l_1, l_2	`bne` t_5, `$0`, l_1
	`j` l_2

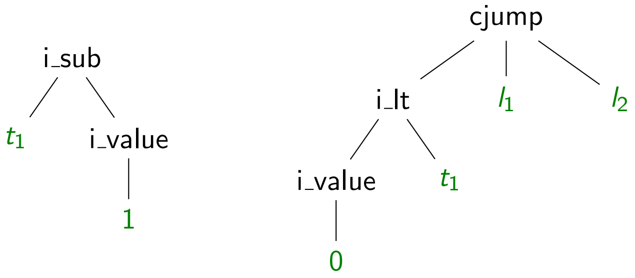
However, both sequences below would be better implementations of that IR code (both in terms of number of instructions and of execution time)

`slt` t_5, `$0`, t_1	`bgtz` t_1, l_1
`bne` t_5, `$0`, l_1	`j` l_2
`j` l_2	

Not so simple translations (3)

To be able to achieve translations like those presented in the previous slides, the compiler must track the dependencies between the IR instructions

For that, it is convenient to regard the IR instructions as trees



Not so simple translations (4)

IR instructions are used as the basis for building patterns, known as tiles

To each tile corresponds a sequence of target machine instructions, with an associated cost

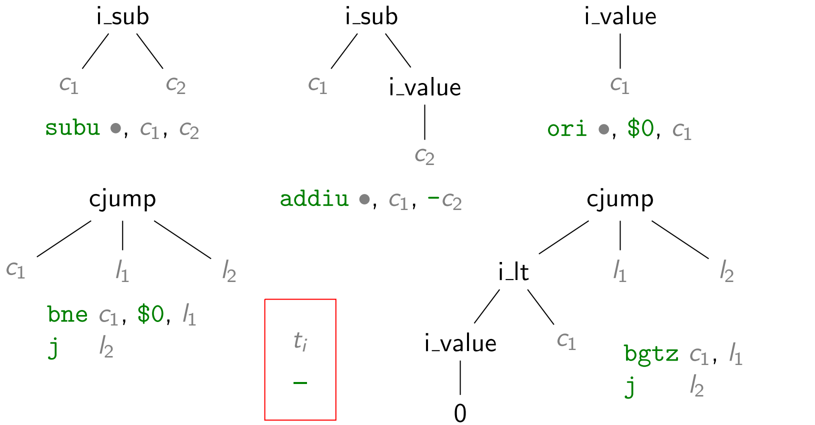
Instruction selection may be regarded as finding a tiling that covers the IR tree(s)

The more sophisticated algorithms try to minimise the cost of tiles used in the tiling

Not so simple translations (5)

Some tiles and their MIPS code

The • stands for the temporary assigned to the root of the tile



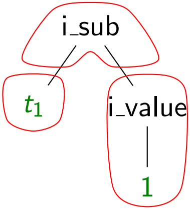
The highlighted tile corresponds to the case where the value is already in a (temporary) register and nothing needs to be done to put it there

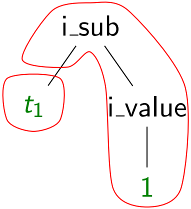
Not so simple translations (6)

Below are shown two tilings for the IR fragment

$$t_4 \leftarrow \text{i\_value } 1$$
$$t_1 \leftarrow \text{i\_sub } t_1, t_4$$

along with the corresponding MIPS implementation





`ori` t_4, \$0, 1
`subu` t_1, t_1, t_4

`addiu` $t_1, t_1,$ -1

Maximal munch (1)

A greedy instruction selection algorithm

1. Matches the largest possible tile at the root of the IR tree
2. Calls itself recursively for the roots of the subtrees not covered by the tile used in the previous step

Maximal munch finds an optimal tiling for the tree

A tiling is optimal when no two adjacent tiles can be combined to obtain a tiling with a lower cost

An optimum tiling has the lowest cost possible

Maximal munch (2)

Implementation

Implementing Maximal munch is just performing a case analysis on the IR

The following slides exemplify how to implement the translation of

$$t_4 \leftarrow \text{i\_value } 1$$
$$t_1 \leftarrow \text{i\_sub } t_1, t_4$$

to

$$\texttt{addiu } t_1, t_1, \texttt{-}1$$

Maximal munch (3)

TACL implementation

```
proc munch_expression(IR e)
[
  match (e)
  [
    (i_sub e1 e2) -> [
      munch_expression(e1);
      if (e2 == (i_value n) && n <= 32768)    # 32768 = 2^15
        emit("addiu", e.temp, e1.temp, -n);
      else [ munch_expression(e2);
             emit("subu", e.temp, e1.temp, e2.temp); ]
    ]
    (i_value n) ->
      if (n < 65536)                          # 65536 = 2^16
        emit("ori", e.temp, "$0", n);
      else [ emit("lui", e.temp, n / 65536);
             emit("ori", e.temp, e.temp, n % 65536); ]
    ...
  ]
]
```

Maximal munch (4)

Remarks

In the MIPS `addiu` r_1, r_2, m instruction, m is a signed 16 bit integer, *i.e.*, $-2^{15} \leq m < 2^{15}$

So, `addiu` $t_j, t_k, -n$ may only be used to implement

$$t_i \leftarrow \text{i\_value } n$$
$$t_j \leftarrow \text{i\_sub } t_k, t_i$$

when $0 \leq n < 2^{15}$ (i_value's argument is nonnegative)

In the MIPS `ori` r_1, r_2, m instruction, m is an unsigned 16 bit integer, *i.e.*, $0 \leq m < 2^{16}$

So, `ori` t_i, \$0, n may only be used to implement

$$t_i \leftarrow \text{i\_value } n$$

when $0 \leq n < 2^{16}$

Maximal munch (5)

TACL implementation (version 2)

If the temporaries have not been assigned, munch_expression must create one and return it

```
fun temp munch_expression(IR e)
[
  var temp t0; ...

  match (e)
  [
    (i_sub e1 e2) -> [
      t0 = new_int_temp();
      t1 = munch_expression(e1);

      if (e2 == (i_value n) && n <= 32768)
        emit("addiu", t0, t1, -n);
      else [
        t2 = munch_expression(e2);
        emit("subu", t0, t1, t2);
      ]
    ]
```

(continues on the next slide. . .)

Maximal munch (6)

TACL implementation (version 2, cont.)

(. . . continued)

```
      (i_value n) -> [
        t0 = new_int_temp();

        if (n < 65536)
          emit("ori", t0, "$0", n);
        else [
          emit("lui", t0, n / 65536);
          emit("ori", t0, t0, n % 65536);
        ]
      ...
    ]

    ^ t0
  ]
```

Maximal munch (7)

Maximal munch is easier to implement in a language like Prolog

```
munch_expression(i_sub(T, E1, i_value(_, N)), T) :-
  N <= 2 ** 15,
  !,
  munch_expression(E1, T1),
  emit(addiu, T, T1, -N).

munch_expression(i_sub(T, E1, E2), T) :-
  munch_expression(E1, T1),
  munch_expression(E2, T2),
  emit(subu, T, T1, T2).

munch_expression(i_value(T, N), T) :-
  (N < 2 ** 16 ->
    emit(ori, T, '$0', N)
  ; Hi16 is N >> 16,              % most significant 16 bits
    Lo16 is N /\ 0xffff,          % least significant 16 bits
    emit(lui, T, Hi16),
    emit(ori, T, T, Lo16)).

...
```

Dynamic programming for instruction selection (1)

Dynamic programming

- A technique employed to solve optimisation problems
- Builds an optimal solution using optimal solutions for subproblems
- Works bottom-up

In instruction selection, dynamic programming is used to obtain optimum tilings

Dynamic programming for instruction selection (2)

Implementation

Instruction selection through dynamic programming may be implemented in the following way

1. Compute the optimum instruction sequence for every child of the current IR node recursively, along with its cost
2. For every tile that matches the root of the current IR tree, compute the cost of using that tile as

$$\text{tile cost} + \sum \text{cost of the subtrees not covered by the tile}$$

3. Select a tile that corresponds to the least cost computed in the previous step
4. (See the next slide)

Dynamic programming for instruction selection (3)

Implementation (cont.)

3. (See the previous slide)
4. Build the translation by combining
   - the translation for the tile selected in step 3., and
   - the translations of the subtrees obtained in step 2. when the selected tile was being considered
5. Set the cost of the current node's optimum translation to the least value obtained in step 2.

Remarks

There may be more than one optimum translations

When there are several tiles for which the cost computed in step 2. is the least cost, regardless of which of those tiles is selected in step 3., the resulting translation is always optimum

Tree grammars (1)

The structure of a tree may be described through a tree grammar

In the following excerpt, R stands for register and N for an integer

$$
\begin{array}{lcl}
R & \rightarrow & (\text{i\_sub } R \text{ (i\_value } N)) \\
R & \rightarrow & (\text{i\_sub } R\ R) \\
R & \rightarrow & (\text{i\_value } N) \\
R & \rightarrow & t_i
\end{array}
$$

The above rules partially specify IR trees for expressions, whose values are stored in a (temporary) register

A complete grammar would include rules for declarations (needed if the code generator does not have access to the symbol table), statements (like cjump and call), and labels

Tree grammars (2)

Code emission is performed via the semantic actions associated with the rules

$R \;\rightarrow\;$ (i_sub R_1 R_2)

```
[[
  var temp r = new_int_temp();
  emit("subu", r, R1, R2);
  $$ = r;
]]
```

Tree grammars for instruction selection are highly ambiguous, as the fragment of the previous slides shows

The quality of the code obtained depends on the order of the rules and on the default behaviour of the generated parsers, which determines the layout of the grammar

Bottom-up rewrite systems

A bottom-up rewrite system (BURS) combines tree grammars and dynamic programming to specify a code generator (BURSs generate code-generators)

Code generators specifications include tree grammar rules with associated costs and semantic actions

Depending on the system, costs may be fixed or variable

Costs guide BURS-based code generators in finding optimum tilings of the IR

Semantic actions handle the generation of code for the selected rules

Peephole optimisation

Peephole optimisation is code optimisation technique

The technique consists in applying a sliding window, with 3 to 5 instructions, to the code

The instructions in the window are analysed and when it is possible to combine some of them to obtain a better instruction sequence, the change is made

The window may be physical (it only allows to examine consecutive instructions in program order) or logical (it allows examining consecutive instructions in execution order)

To apply this technique to instruction selection, the intermediate representation instructions are examined through the window, trying to find some group which can be translated together, in order to obtain better code than when the translation is done one IR instruction at a time