Faculty of Science

Machine Code Generation

Cosmin E. Oancea
cosmin.oancea@diku.dk

Department of Computer Science (DIKU)
University of Copenhagen

December 2014 Compiler Lecture Notes

Structure of a Compiler

Program text
↓
Lexical analysis
↓
Symbol sequence
↓
Syntax analysis
↓
Syntax tree
↓
Type Checking
↓
Syntax tree
↓
Intermediate code generation
⟶
Intermediate code
↑
Machine code generation
↑
Symbolic machine code
↑
Register allocation
↑
Ditto with named registers
↑
Assembly and linking
↑
Binary machine code

1. Quick Look at MIPS
2. Intermediate *vs* Machine Code
3. Exploiting Complex Instructions
4. Machine-Code Generation in FASTO

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*.

Remember 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)

```
        .data
val:    .word 10, -14, 30
str:    .asciiz "Hello!"
_heap_: .space 100000
        .text
        .global main
        la $28, _heap_
        jal main
        ...
_stop_:
        ori  $2, $0, 10
        syscall
main:
        la   $8, val     # ?
        lw   $9, 4($8)   # ?
        addi $9, $9, 4   # ?
        sw   $9, 8($8)   #...
        j    _stop_      #jr $31
```

Remember 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)

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

```
_stop_:
        ori  $2, $0, 10
        syscall
main:
        la   $8, val     # ?
        lw   $9, 4($8)   # ?
        addi $9, $9, 4   # ?
        sw   $9, 8($8)   #...
        j    _stop_      #jr $31
```

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

1 Quick Look at MIPS

2 Intermediate *vs* Machine Code

3 Exploiting Complex Instructions

4 Machine-Code Generation in Fasto

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 two issues are solved in the next two lessons.

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 $\neq$ 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

```
fun compileExp e vtable place =
  case e of
    Constant (IntVal n, pos) =>
      if n < 32768 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

```
fun compileExp e vtable place =
  case e of
    Constant (IntVal n, pos) =>
      if n < 32768 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?

```
fun compileExp e vtable place =
  case e of
    Constant (IntVal n, pos) =>
      if n < 0 then
        compileExp (Negate (Constant (IntVal (~n), pos), pos)) vtable place
      else if n < 32768 then
        [ Mips.LI (place, makeConst n) ]
      else
        [ Mips.LUI (place, makeConst (n div 65536))
        , Mips.ORI (place, place, makeConst (n mod 65536)) ]
```

1. Quick Look at MIPS
2. Intermediate *vs* Machine Code
3. Exploiting Complex Instructions
4. Machine-Code Generation in FASTO

Exploiting Complex Instructions

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

- load/store instruction also involve address calculation,
- arithmetic instructions that scales 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:

```
t2 := t1 + 116
t3 := M[ t2 ]
```

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

```
lw r3, 116(r1)
```

MIPS Example

The two intermediate-code instructions:

```
t2 := t1 + 116
t3 := M[ t2 ]
```

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

```
lw r3, 116(r1)
```

IFF t_2 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 *liveness analysis*; we write:

```
t2 := t1 + 116
t3 := M[ t2^last ]
```

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

$t := r_s + k$ $r_t := M[t^{last}]$	`lw` $r_t,\ k(r_s)$

t, r_s and r_t can match arbitrary IL variables, k can match any constant (big constants have already been eliminated).

Patterns for MIPS (part 1)

$t := r_s + k$, $r_t := M[t^{last}]$	lw	$r_t,\ k(r_s)$
$r_t := M[r_s]$	lw	$r_t,\ 0(r_s)$
$r_t := M[k]$	lw	$r_t,\ k(\mathtt{R0})$
$t := r_s + k$, $M[t^{last}] := r_t$	sw	$r_t,\ k(r_s)$
$M[r_s] := r_t$	sw	$r_t,\ 0(r_s)$
$M[k] := r_t$	sw	$r_t,\ k(\mathtt{R0})$
$r_d := r_s + r_t$	add	$r_d,\ r_s,\ r_t$
$r_d := r_t$	add	$r_d,\ \mathtt{R0},\ r_t$
$r_d := r_s + k$	addi	$r_d,\ r_s,\ k$
$r_d := k$	addi	$r_d,\ \mathtt{R0},\ k$
GOTO *label*	j	*label*

Must cover all possible sequences of intermediate-code instructions.

Patterns for MIPS (part 2)

Pattern	MIPS code
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 $label_f$
LABEL $label$	$label$:

Compiling Code Sequences: Example

$a := a + b^{last}$
$d := c + 8$
$M[d^{last}] := a$
IF $a = c$ THEN $label_1$ ELSE $label_2$
LABEL $label_2$

Compiling Code Sequences

Example:

IL code	Machine code
$a := a + b^{last}$	add a, a, b
$d := c + 8$	sw a, $8(c)$
$M[d^{last}] := a$	
IF $a = c$ THEN $label_1$ ELSE $label_2$	beq a, c, $label_1$
LABEL $label_2$	$label_2$:

Two approaches:

Greedy Alg: Find the first/longest pattern matching a prefix of the 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:

$r_t := r_s$	`mov` $r_t,\ r_s$
$r_t := r_t + r_s$	`add` r_t, r_s
$r_d := r_s + r_t$	`move` $r_d,\ r_s$ `add` r_d, r_t

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.

We talk more about optimizations in the next two lectures!

1 Quick Look at MIPS

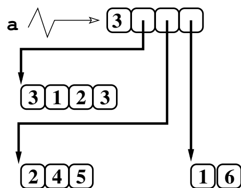
2 Intermediate *vs* Machine Code

3 Exploiting Complex Instructions

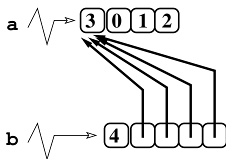
4 Machine-Code Generation in Fasto

Fasto Arrays

```
a = { {1, 2, 3},
      {4, 5}, {6} }
```

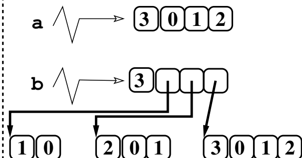



```
let a = iota(3) in
let b = replicate(4, a)
```




```
fun [int] mkArr(int a)=
            iota(a+1)

let a=iota(3) in
let b=map(mkArr,a) in..
```



Let us translate `let a2 = map(f, a1)`, where `a1,a2 : [int]` and R_{a1} holds `a1`, R_{a2} holds `a2`, R_{HP} is the heap pointer.

Example: Translation of `let a2 = map(f, a1)`

R_{a1} holds a1, R_{a2} holds a2, R_{HP} is the heap pointer, `a1,a2 : [int]`

```
len = length(a1)
a2  = malloc(len*4)
i = 0
while(i < len) {
    tmp   = f(a1[i]);
    a2[i] = tmp;
}
```

```
lw    R_len , 0(R_a1)
move  R_a2  , R_HP
sll   R_tmp, R_len, 2
addi  R_tmp, R_tmp, 4
add   R_HP, R_HP, R_tmp
sw    R_len , 0(R_a2)
addi  R_it1 , R_a1, 4
addi  R_it2 , R_a2, 4
move  R_i   , $0
```

```
loop_beg:
        sub    R_tmp, R_i, R_len
        bgez   R_tmp, loop_end
        lw     R_tmp, 0(R_it1)
        addi   R_it1, R_it1, 4
        R_tmp = CALL f(R_tmp)
        sw     R_tmp, 0(R_it2)
        addi   R_it2, R_it2, 4
        addi   R_i, R_i, 1
        j      loop_beg
loop_end:
```

Compiler.sml:

- dynalloc generates code to allocate an array,
- ApplyRegs gen code to call a function on a list of registers args.
- applyFunArg is called for SOACS (`map`,`reduce`,`filter`,`scan`) and needs to be extended for unnamed functions (λs).

Mips Representation is in File Mips.sml

Several differences:

- Mips.MOVE($$r_d$, r_s) $\equiv$ r_d := r_s;
- Mips.LW(r, r_m, K) $\equiv$ lw r, K(r_m) $\equiv$ r := M[r_m+K];
- Mips.SW(r, r_m, K) $\equiv$ sw r, K(r_m) $\equiv$ M[r_m+K] :=r;
- Mips.BEQ(r,"0",some_label) jumps to some label if register r holds value 0, because register $0 is hardwired to value 0.
- Mips.BEQ(r,"1",some_label) is probably a bug: compares the value of register r with whatever happens to be hold in register $1 (not with value 1)!

Entry Point to CodeGen.sml

- stack grows from high to low mem, heap is statically alloc,
- user defined funs & CT strings, helper funs, errors, etc.

```
fun compile funs =
 let val () = stringTable := []
 val funsCode = List.concat
          (List.map compileFun funs)
 (* magic init of prg's CT strings *)
 val (stringinit, stringdata) = ... in
 [  Mips.TEXT "0x00400000",
    Mips.GLOBL "main"       ]
 (*init heap & string pointers*)
@ (Mips.LA (HP, "_heap_")::stringinit)
@ [ Mips.JAL ("main",[]),(* run pgm *)
    Mips.LABEL "_stop_",(*end pgm&err*)
    Mips.LI ("2","10"),(*syscall exit*)
    Mips.SYSCALL ]

@ funsCode   (* code for functions *)
(* chr : int -> char *)
@ [Mips.LABEL "chr",(*trunc to 8 bits*)
   Mips.ANDI("2", "2", makeConst 255),
   Mips.JR (RA,[]), ...] ...
```

```
@ [Mips.LABEL "getint",
   Mips.LI ("2","5"),
   Mips.SYSCALL,(*read_int syscall*)
   Mips.JR (RA,[]) ]
@ [Mips.LABEL "_IllegalArrErr_",
   Mips.LA ("4","_IllegalArr_"),
   Mips.LI ("2","4"),
   Mips.SYSCALL, (*print str*)
   Mips.MOVE("4","5"),Mips.LI("2","1")
  ,Mips.SYSCALL, (*print line*)
   Mips.J "_stop_"]
@ [ (* String CT & Err Msg *)
   Mips.DATA "", Mips.ALIGN "2",
   Mips.LABEL "_IllegalArr_",
   Mips.ASCIIZ "Error: Illegal " ]
 @ (* program's string literals *)
   List.concat stringdata
  (* Heap (array allocation) *)
 @ [Mips.ALIGN "2",Mips.LABEL "_heap_"
   ,Mips.SPACE "100000" ]
```

CodeGen.sml: Compiling a Plus Expression

- compiling a function eventually leads to compiling its body (exp),
- which is implemented in compileExp and receives as args:
  - e — the to-be-compiled expression
  - vtable — binding Fasto-variable names to symbolic-register names
  - place — the symbolic register that will hold the result of input expression e
- and results in (a list of) Mips code.

```
fun compileExp e vtable place =
    case e of ...
    | Plus (e1, e2, pos) =>
        (* two new uniquely name symbolic registers *)
        let val t1 = newName "plus_L"
            val t2 = newName "plus_R"

            (* compiles e1 and e2, the results *)
            (* will be in registers t1 and t2  *)
            val code1 = compileExp e1 vtable t1
            val code2 = compileExp e2 vtable t2

            (* adds t1 and t2 and result of e is in place *)
        in code1 @ code2 @ [Mips.ADD (place,t1,t2)] end
```

CodeGen.sml: Compiling an IF Expression

- Proper jumping code requires implementing both `compileCond` and `compileExp` for short-circuited operators `&&` and `||`.
- Should you decide to implement only at `compileExp` level then an extra Jump is used when used as an `if` condition.
  (The provided implem of == suffers the same drawback.)

```
fun compileExp e vtable place = case e of ...
    | If (e1, e2, e3, pos) =>
        let val thenLabel = newName "then"
            val elseLabel = newName "else"
            val endLabel = newName "endif"
            val code1 = compileCond e1 vtable thenLabel elseLabel
            val code2 = compileExp e2 vtable place
            val code3 = compileExp e3 vtable place
        in code1 @ [Mips.LABEL thenLabel] @ code2  @
           [ Mips.J endLabel, Mips.LABEL elseLabel ] @
           code3 @ [Mips.LABEL endLabel]
        end ...
and compileCond c vtable tlab flab =
    let val t1 = newName "cond"
        val code1 = compileExp c vtable t1
    in  code1 @ [Mips.BNE (t1, "0", tlab), Mips.J flab] end
```

CodeGen.sml: Compiling an Equality Expression

- compiles the first subexpression in t1
- compiles the second subexpression in t2
- sets place to 0
- if t1 $\neq$ t2 then jump over the next instr place := 1, i.e.,
- if t1 and t2 differ then place remains set to 0.

```
fun compileExp e vtable place =
    case e of ...
    | Equal (e1, e2, pos) =>
        let val t1 = newName "eq_L"
            val t2 = newName "eq_R"
            val code1 = compileExp e1 vtable t1
            val code2 = compileExp e2 vtable t2
            val falseLabel = newName "false"
        in  code1 @ code2 @
            [ Mips.LI (place,"0")
            , Mips.BNE (t1,t2,falseLabel)
            , Mips.LI (place,"1")
            , Mips.LABEL falseLabel ]
        end
```

Preparation for exam

1. Explain briefly the motivation for having an intermediate language between m source languages and n hardware.

Preparation for exam

1 Explain briefly the motivation for having an intermediate language between *m* source languages and *n* hardware.

2 Translate the following statement to intermediate code and then to MIPS code, assuming initially `vtable=[a->v, b->w]`. Logical-and operator `&&`, should be translated with jumping (short-circuited) code ... (see Assignment 3, Exercise 2.)

```
while (b != 0) && (a/b != 0) {
    if b < a then { a := a - b }
             else { b := b - a }
}
```

Preparation for exam

1. Explain briefly the motivation for having an intermediate language between *m* source languages and *n* hardware.

2. Translate the following statement to intermediate code and then to MIPS code, assuming initially `vtable=[a->v, b->w]`. Logical-and operator &&, should be translated with jumping (short-circuited) code ... (see Assignment 3, Exercise 2.)

```
while (b != 0) && (a/b != 0) {
    if b < a then { a := a - b }
              else { b := b - a }
}
```

3. implement Mips code generator in Fasto for multiplication or for logical negation (not) or for logical or ...

I will not insist on (intermediate) code generation because you did enough of it for the project...

Preparation for exam

1. The greedy algorithm for compiling code sequences always result in optimal (most efficient) code.

Preparation for exam

1. The greedy algorithm for compiling code sequences always result in optimal (most efficient) code.
2. implement Mips code generator in Fasto for multiplication or for logical negation (not) or for logical or ...
3. Long question (8pts): show simple C pseudocode for filter/scan/zipWith, then translate it to Mips using the array layout used in Fasto (see Assignment 3, Exercise 4)

I will not insist on Mips code generation because you did enough of it for the project...

Preparation for exam

4 Using the patterns below translate the code on the left to Mips:

$t := r_s + k$, $M[t^{last}] := r_t$	`sw` $r_t,\ k(r_s)$
$M[r_s] := r_t$	`sw` $r_t,\ 0(r_s)$
$r_d := r_s + r_t$	`add` $r_d,\ r_s,\ r_t$
$r_d := r_s + k$	`addi` $r_d,\ r_s,\ k$
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 $label_f$	`beq` $r_s,\ r_t,\ label_t$ `j` $label_f$

$a := a + b^{last}$
$d := c + 8$
$M[d^{last}] := a$
IF $a = c$ THEN $label_1$ ELSE $label_2$
LABEL $label_2$

This example appears in this lecture notes.

Preparation for exam

4 Using the patterns below translate the code on the left to Mips:

$t := r_s + k$, $M[t^{last}] := r_t$		`sw`	$r_t,\ k(r_s)$
$M[r_s] := r_t$		`sw`	$r_t,\ 0(r_s)$
$r_d := r_s + r_t$		`add`	$r_d,\ r_s,\ r_t$
$r_d := r_s + k$		`addi`	$r_d,\ r_s,\ k$
IF $r_s = r_t$ THEN $label_t$ ELSE $label_f$, LABEL $label_f$	 $label_f$:	`beq`	$r_s,\ r_t,\ label_t$
IF $r_s = r_t$ THEN $label_t$ ELSE $label_f$		`beq` `j`	$r_s,\ r_t,\ label_t$ $label_f$

$a := a + b^{last}$		`add`	$a,\ a,\ b$
$d := c + 8$		`sw`	$a,\ 8(c)$
$M[d^{last}] := a$			
IF $a = c$ THEN $label_1$ ELSE $label_2$		`beq`	$a,\ c,\ label_1$
LABEL $label_2$	$label_2$:		