Compilers
Code Generation

LEIC

FEUP-FCUP

2022

This lecture

Basic Blocks

Symbolic Machine Code

Intermediate code vs. machine code

Instruction Patterns

Machine code generation

JVM

This lecture

Basic Blocks

Symbolic Machine Code

Intermediate code vs. machine code

Instruction Patterns

Machine code generation

JVM

Basic Blocks

Basic blocks: maximal sequences of consecutive three address instructions such that:

- The flow of control can only enter the basic block through the first instruction in the block. That is, there are no jumps into the middle of the block
- Control leaves the block without halting or branching except at the last instruction in the block

Partition of three-address code into basic blocks

Leader: first instruction in a basic block

- The first three address instruction in the intermediate code is a leader
- Any instruction that is the target of a jump (conditional or unconditional) is a leader
- Any instruction that immediately follows a jump (conditional or unconditional) is a leader

For each leader its basic block consists of itself and all instructions up to but not including the next leader or the end of the intermediate program.

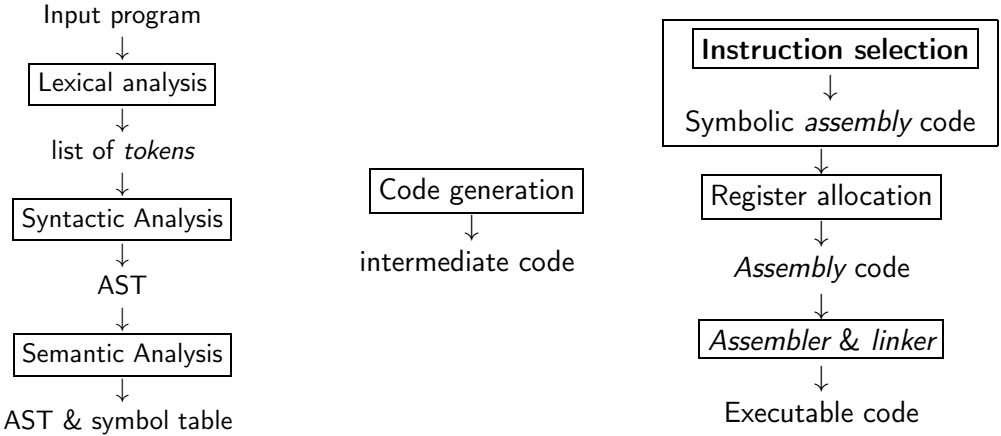
Example

```
1)  i := 1
2)  j := 1
3)  t1 := 10*i
4)  t2 := t1+j
5)  t3 := 8*12
6)  t4 := t3-88
7)  M[t4] := 0
8)  j := j+1
9)  COND j <= 10 GOTO (3)
10) i := i+1
11) COND j <= 10 GOTO (2)
12) i := 1
13) t5 := i-1
14) t6 := 88*t5
15) M[t6] := 1
16) i := i+1
17) COND i <= 10 GOTO (13)
```

Leaders: 1, 3, 2, 13, 10, 12, (18)

Basic Blocks: leader 1 -> [1], leader 2 -> [2], leader 3 -> [3-9], leader 10 -> [10,11], leader 12 -> [12], leader 13 -> [13-17]

The compiler



This lecture

Basic Blocks

Symbolic Machine Code

Intermediate code vs. machine code

Instruction Patterns

Machine code generation

JVM

Symbolic machine code (“assembly”)

- Text representation of machine code
- More readable than binary code
- Symbolic labels for addresses and constants
- Compiled to executable machine code by the *assembler* and the *linker*

Example: MIPS architecture

- RISC architecture
- 32 integer registers \$0 – \$31 de 32-bits
- Set of instructions *load/store*
- Fixed size instructions (32-bits)
- Operations between 3 registers or registers and constants (*immediate*)

Registers

$0	zero ($zero)
$1	*assembler* temporary register
$2–$3	return values ($v0–$v1)
$4–$7	arguments ($a0-$a3)
$26–$27	reserved for the *kernel*
$28	global pointer ($gp)
$29	stack pointer ($sp)
$30	frame pointer ($fp)
$31	return address ($ra)
$8–$25	temporary registers $t0–$t7, $s0–$s7 e $t9–$t10

Example

```
            .data               # data
val:        .word 10, -14, 30   # array of constants
str:        .asciiz "Hello!"    # string ended in zero
_heap_:     .space 10000        # reserve 10k bytes
            .text               # instructions
            .global main        # main
            la $gp, _heap_      # inicialize reg $28 = _heap_
            jal main            # jump and link (reg $31)
_stop_:
            li $2,10            # reg $2 = 10
            syscall             # syscall 10: exit
main:
            la $8, val          # reg $8 = val
            lw $9, 4($8)        # reg $9 = val[1]
            addi $9, $9, 4      # reg $9 = reg $9 + 4
            sw $9, 8($8)        # val[2] = reg $9
            jr $ra              # jump register $31$ (return address)
```

This lecture

Basic Blocks

Symbolic Machine Code

Intermediate code vs. machine code

Instruction Patterns

Machine code generation

JVM

Diferences between the intermediate code and machine code

1. Machine code has a finite number of registers
2. There is no function CALL instruction, i.e. it has to be implemented using the stack and/or registers for memory allocation
3. Conditional jumps have goto to only one label
4. It may be necessary to do the conditional expression separated from the conditional jump
5. RISC architectures only allow small constants

Points 1 and 2 will be presented next weeks (register allocation and run-time memory management).

Today we will talk about points 3–5.

Conditional jumps with two labels

`COND` *c* *labelt* *labelf* is compiled to:

```
branch_if_c    labelt
jump           labelf
```

Conditional jumps with two labels

`COND` *c* *labelt* *labelf* is compiled to:

```
        branch_if_c    labelt
        jump           labelf
```

Optimization: If the next instruction is *labelt* ou *labelf* we may delete a jump:

```
  COND c labelt labelf
  LABEL labelt
  ...
```

is compiled to:

```
         branch_if_not_c labelf
labelt: ....
```

Conditional jumps with two labels

`COND` *c* *labelt* *labelf* is compiled to:

```
        branch_if_c    labelt
        jump           labelf
```

Optimization: If the next instruction is *labelt* ou *labelf* we may delete a jump:

```
  COND c labelt labelf
  LABEL labelt
  ...
```

is compiled to:

```
         branch_if_not_c labelf
labelt: ....
```

(We keep *labelt* because it may be used by other instructions.)

Conditional expressions

In many architectures comparisons must be done before the jump instruction

Examples:

- In MIPS comparisons $=$ and $\neq$ are direct (`beq` and `bne`) but $<$, $>$ etc. must calculate the condition and put the result in a register (`slt`, `sgt`, etc.)
- In ARM and X86 arithmetic expressions use *flags* (zero, negative, *overflow*, *carry*,...)

Conditional expressions (cont.)

MIPS:

```
beq $t1, $t2, ltrue    # if $t1==$t2 goto ltrue
...                    # othewise
```

MIPS:

```
slt $1, $t1, $t2        # compare $t1 < $t2 and store the result in $1
bne $1, $zero, ltrue    # if $1 is different from zero goto ltrue
...                     # otherwise$
```

(Use $1 as a temporary register.)

Constants

Machine code enable constants of a limited size:

- MIPS: constants of 16 bits; for bigger constants use `lui` to store 16 bits in the high halt part of a register
- ARM has 8 bit constants which may be stored in different positions in a register

Code generation must split some instructions if necessary.

Example: the assignment

```
t0 := 0x87fe000
```

corresponds to 2 instructions in MIPS (using $1 as a temporary register):

```
lui $1,0x87            # $1 = 0x870000
ori $t0, $1, 0xfe00    # $t0 = 0x87fe00
```

Pseudo-insttructions

- Assembly support *pseudo-instructions*.
  - `blt`, `bgt`, `ble`, `bge` (*branch*) with conditions $<$, $>$, $\leq$, $\geq$
  - `li` (*load immediate*) with constants $>$ 16 bits
  - `move` between registers
- Are not implemented in *hardware*
- Compiled to machine code by the assembler::

pseudo-instruction	translation
`move $t1, $t2`	`addu $t1, $zero, $t2`
`li $t0, 0x87fe00`	`lui $1,0x87` `ori $t0, $1, 0xfe00`
`blt $t0, $t1, label`	`slt $1, $t0, $t1` `bne $1, $zero, label`

This lecture

Basic Blocks

Symbolic Machine Code

Intermediate code vs. machine code

Instruction Patterns

Machine code generation

JVM

Intermediate Code Patterns

- Compile intermediate code patterns directly to assembly
- Compile patterns as big as possible
- If this is not possible, use patterns for individual instructions

Example (1)

COND $r_s = r_t$ $label_t$ $label_f$, LABEL $label_f$		beq	r_s, r_t, $label_t$
	$label_f$:		
COND $r_s = r_t$ $label_t$ $label_f$, LABEL $label_t$		bne	r_s, r_t, $label_f$
	$label_t$:		
COND $r_s = r_t$ $label_t$ $label_f$		beq	r_s, r_t, $label_t$
		j	$label_f$
COND $r_s < r_t$ $label_t$ $label_f$, LABEL $label_f$		blt*	r_s, r_t, $label_t$
	$label_f$:		
COND $r_s < r_t$ $label_t$ $label_f$, LABEL $label_t$		bge*	r_s, r_t, $label_f$
	$label_t$:		
COND $r_s < r_t$ $label_t$ $label_f$		blt*	r_s, r_t, $label_t$
		j	$label_f$
LABEL $label$	$label$:		
JUMP $label$		j	$label$

(* *pseudo*-instructions.)

Example (2)

$r_d := r_s + r_t$	`add` 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_d, r_s, r_t
$r_d := r_t$	`move` r_d, r_t
$r_d := k$	`li` r_d, k
$r_t := M[r_s]$	`lw` r_t, $0(r_s)$
$M[r_s] := r_t$	`sw` r_t, $0(r_s)$

It is necessary to deal with all possibilities of intermediate code (the other arithmetic expressions)

Example

Intermediate code

```
LABEL max
COND t1<t2 label1 label2
LABEL label1
t3 := t2
JUMP label3
LABEL label2
t3 := t1
LABEL label3
```

MIPS code

Example

Intermediate code

```
LABEL max
COND t1<t2 label1 label2
LABEL label1
t3 := t2
JUMP label3
LABEL label2
t3 := t1
LABEL label3
```

MIPS code

```
max:
```

Example

Intermediate code

```
LABEL max
COND t1<t2 label1 label2
LABEL label1
t3 := t2
JUMP label3
LABEL label2
t3 := t1
LABEL label3
```

MIPS code

```
max:
    bge $t1, $t2, label2
label1:
```

Example

Intermediate code

```
LABEL max
COND t1<t2 label1 label2
LABEL label1
t3 := t2
JUMP label3
LABEL label2
t3 := t1
LABEL label3
```

MIPS code

```
max:
    bge $t1, $t2, label2
label1:
    move $t3, $t2
```

Example

Intermediate code

```
LABEL max
COND t1<t2 label1 label2
LABEL label1
t3 := t2
JUMP label3
LABEL label2
t3 := t1
LABEL label3
```

MIPS code

```
max:
    bge $t1, $t2, label2
label1:
    move $t3, $t2
    j label3
```

Example

Intermediate code

```
LABEL max
COND t1<t2 label1 label2
LABEL label1
t3 := t2
JUMP label3
LABEL label2
t3 := t1
LABEL label3
```

MIPS code

```
max:
    bge $t1, $t2, label2
label1:
    move $t3, $t2
    j label3
label2:
```

Example

Intermediate code

```
LABEL max
COND t1<t2 label1 label2
LABEL label1
t3 := t2
JUMP label3
LABEL label2
t3 := t1
LABEL label3
```

MIPS code

```
max:
    bge $t1, $t2, label2
label1:
    move $t3, $t2
    j label3
label2:
    move $t3, $t1
```

Example

Intermediate code

```
LABEL max
COND t1<t2 label1 label2
LABEL label1
t3 := t2
JUMP label3
LABEL label2
t3 := t1
LABEL label3
```

MIPS code

```
max:
    bge $t1, $t2, label2
label1:
    move $t3, $t2
    j label3
label2:
    move $t3, $t1
label3:
```

Example

Intermediate code

```
LABEL max
COND t1<t2 label1 label2
LABEL label1
t3 := t2
JUMP label3
LABEL label2
t3 := t1
LABEL label3
```

MIPS code

```
max:
    bge $t1, $t2, label2
label1:
    move $t3, $t2
    j label3
label2:
    move $t3, $t1
label3:
```

Compilation of sequences of instructions

- Greedy algorithm:
  1. translate the bigger patterns which matches the intermediate code
- True compilers!: assign costs to each machine instruction; choose the match which minimizes the cost using a dynamic programming algorithm
- The first approach is ok for RISC architectures; the second approach should be used for CISC architectures

Array indexes

- To calculate addresses one must do the product by the element size (ex: 4 bytes for 32-bits integers)
- Detail: If the size is a power of 2 instead of multiplication use *left shift* (more efficient)
- Example: we may compile `t1 := t1 * 8` to:

  ```
  sll $t1, $t1, 3
  ```

  (because $8 = 2^3$)
- In general: compile

  $$r_d := r_s * 2^k$$

  to:

  $$\texttt{sll } r_d,\ r_s,\ k$$

Example

(Assignment in C)

```
int x[N], i;
...
x[i] = 5;
```

(Assume $[\mathtt{x} \mapsto t_0, \mathtt{i} \mapsto t_1]$.)

Intermediate code:

```
t2 := t1
t2 := t2*4
t2 := t2 + t0
t4 := 5
M[t2] := t4
```

Assembly code (MIPS):

```
move $t2, $t1
sll $t2, $t2, 2
add $t2, $t2, t0
li $t4, 5
sw $t4, 0($t2)$
```

Complex instructions

Many architectures combine several operations into one machine instruction:

- operations *load/store* which do address calculations;
- arithmetic instructions which combine *add* and *shifts*;
- instructions to transfer several registers to/from memory

Thus we may compile several intermediate code instructions $\rightarrow$ one machine instruction

Example

Compile:

$$t2 := t1 + 8$$
$$t3 := M[t2]$$

to:

```
lw $t3, 8($t1)
```

Example

Compile:

$$t2 := t1 + 8$$
$$t3 := M[t2]$$

to:

```
lw $t3, 8($t1)
```

But: only if `t2` is not used later!

The compiler knows this information using static analysis, in this case variable *liveness* analysis (next weeks you will learn this).

This lecture

Basic Blocks

Symbolic Machine Code

Intermediate code vs. machine code

Instruction Patterns

Machine code generation

JVM

Machine code genereation

- In the intermediate code patterns used we are assuming that temporary variables correspond to registers
- To assign temporary variables of the intermediate code code to registers the compiler do (register allocation)
- Naive alternative:
  - re-use temporaries during intermediate code generation
  - assign each temporary to a different register
  - Problem: it does not work for long expressions nor CISC architectures...
  - ...but it usually works for RISC architectures (because they have many registers available)

Re-use temporary variables

- Compilation to intermediate code generates new temporary variables
- But we know that some of them are not going to be needed later
- Thus: they can be re-used for the next expressions

$transExp\ (expr,\ table,\ dest)\ =$ case $expr$ of	
$\vdots$	
e_1 **binop** e_2	$t_1 = newTemp()$ $t_2 = newTemp()$ $code_1 = transExp(e_1, table, t_1)$ $code_2 = transExp(e_2, table, t_2)$ $popTemp(2)$ (t_1, t_2 are not used later) return $code_1 ++ code_2 ++ [dest := t_1$ **binop** $t_2]$

Re-use temporary variables (cont.)

Function popTemp just decrements the global variable counter:

```
int temp_count = 0;  // global counter

int newTemp() {
    return temp_count ++;
}

void popTemp(int k) {
   temp_count -= k;
}
```

The number of temporaries used in a function (or method) is the maximal depth of their arithmetic expressions (independently of the number of instructions).

Example

```
int f (int x, int y) {
  return 3*x + 2*y;
}
```

```
f(a0, a1) [
t3 := 3
t4 := a0
t1 := t3 * t4
t3 := 2
t4 := a1
t2 := t3 * t4
t0 := t1 + t2
RETURN t0
]
```

- When compiling to assembly code (MIPS) you may now use registers \$t0–\$t4 for the temporary variables $t1 \ldots t4$
- Next lecture: you will see how to compile function parameters and the return value

This lecture

Basic Blocks

Symbolic Machine Code

Intermediate code vs. machine code

Instruction Patterns

Machine code generation

JVM

Compilation of Three Address Code to the JVM

JVM:

- JVM is strongly typed: we must deal with the fact that the JVM instructions occasionally require explicit type information
- JVM is a stack machine: push operands on the stack, do their computation and then pop their result off the stack

Compilation of Three Address Code to the JVM (cont.)

Three address code instruction (local variables):

```
x = y + z
```

Suppose that x is the local variable 1, y local variable 2 and z local variable 3. The JVM equivalent is:

```
iload_2
iload_3
iadd
istore_1
```

$iload_x$ loads the int value from local variable x onto the stack. iadd adds the two elements on the top of stack and pushes the result.

Compilation of Three Address Code to the JVM (cont.)

Three address code instruction (class foo variables):

```
x = y + z
```

JVM equivalent:

```
aload_0
getfield foo/y I
aload_0
getfield foo/z I
iadd
putfield foo/x
```

aload_0 pushes the self object

Compilation of Three Address Code to the JVM (cont.)

Three address code instruction:

```
x = y
```

JVM equivalent:

```
iload_2
istore_1
```

x is local variable 1 and y local variable 2

Compilation of Three Address Code to the JVM (cont.)

Three address code instruction:

```
x = -y
```

JVM equivalent:

```
iload_2
ineg
istore_1
```

Compilation of Three Address Code to the JVM (cont.)

Three address code instruction:

```
if x < y then goto L
```

JVM equivalent:

```
iload_1
iload_2
isub
iflt L
```

x local variable 1 and y local variable 2

Compilation of Three Address Code to the JVM (cont.)

Three address code instruction:

```
if x then goto L
```

JVM equivalent:

```
iload_1
ifne L
```

Compilation of Three Address Code to the JVM (cont.)

Three address code instruction:

```
param x
```

JVM equivalent:

```
iload_1
```

(x is local variable 1)

Compilation of Three Address Code to the JVM (cont.)

Three address code instruction:

```
call p,n
```

JVM equivalent:

```
invokevirtual #p
```

p is a symbolic reference to method p

Compilation of Three Address Code to the JVM (cont.)

Three address code instruction:

```
return x
```

JVM equivalent:

```
iload_x
ireturn
```

(_x is the number of local variable x)

a good starting point for compilation to the JVM:
https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-3.htmljvms-3.7