Compiling Techniques

Lecture 12: Code generation (EaC Chapter 7)

Christophe Dubach / Lu Li

23 October 2018

Table of contents

Boolean and Relational Values

How to represent (x<10 && y>3)?

It depends on the target machine

Several approaches:

- Numerical representation
- Positional Encoding (*e.g.* MIPS assembly)
- Conditional Move and Predication

Correct choice depends on both context and ISA (Instruction Set Architecture)

Numerical Representation

- Assign values to true and false, usually 1 and 0
- Use comparison operator from the ISA to get a value from a relational expression

Example

x < y	cmp_LT rx, ry → r1

if (x < y)
  stmt1
else
  stmt2
	cmp_LT rx, ry → r1
	    cbr r1 → L1
	    stmt2
	    br → Le
	L1: stmt1
	Le:

Positional Encoding

What if the ISA does not provide comparison operators that returns a value?

- Must use conditional branch to interpret the result of a comparison
- Necessitates branches in the evaluation
- This is the case for MIPS assembly (and Java ByteCode for instance)

Example: x<y

$$
\begin{array}{llll}
 & \text{br\_LT} & \text{rx, ry} & \rightarrow L_T \\
 & \text{loadI} & 0 & \rightarrow \text{r1} \\
 & \text{br} & & \rightarrow L_E \\
L_T: & \text{loadI} & 1 & \rightarrow \text{r1} \\
L_E: & \ldots & &
\end{array}
$$

If the result is used to control an operation, then positional encoding is not that bad.

Example

```
if (x < y)
  a = c + d;
else
  a = e + f;
```

Corresponding assembly code

Boolean comparison

```
      cmp_LT  rx, ry -> r1
      cbr     r1     -> L_T
      add     re, rf -> ra
      br             -> L_E
L_T : add     rc, rd -> ra
L_E : ...
```

Positional encoding

```
      br_LT   rx, ry -> L_T
      add     re, rf -> ra
      br             -> L_E
L_T : add     rc, rd -> ra
L_E : ...
```

Conditional Move and Predication

Conditional move and predication can simplify this code.

Example

```
if (x < y)
  a = c + d;
else
  a = e + f;
```

Corresponding assembly code

Conditional Move	Predicated Execution
`cmp_LT rx, ry → r1`	`cmp_LT rx, ry → r1`
`add rc, rd → r2`	`(r1)? add rc, rd → ra`
`add re, rf → r3`	`(!r1)? add re, rf → ra`
`cmov r1, r2, r3 → ra`	

Last word on boolean and relational values: consider the following code x = (a<b) & (c<d)

Corresponding assembly code

Positional encoding	**Boolean Comparison**
br_LT ra , rb → L_1	
br → L_2	
L_1: br_LT rc , rd → L_3	cmp_LT ra , rb → r1
L_2: loadI 0 → rx	cmp_LT rc , rd → r2
br → Le	and r1 , r2 → rx
L_3: loadI 1 → rx	
L_e: . . .	

Here the boolean comparison produces much better code.

Best choice depends on two things

- Context
- Hardware

Control-Flow

- If-then-else
- Loops (for, while, ...)
- Switch/case statements

If-then-else

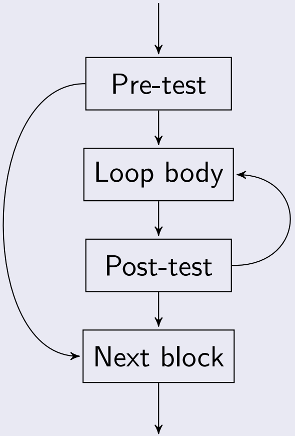
Follow the model for evaluating relational and boolean with branches.

Branching versus predication (*e.g.* IA-64, ARM ISA) trade-off:

- Frequency of execution:
  uneven distribution, try to speedup common case
- Amount of code in each case:
  unequal amounts means predication might waste issue slots
- Nested control flow:
  any nested branches complicates the predicates and makes branching attractive

Loops

Basic pattern



- evaluate condition before the loop (if needed)
- evaluate condition after the loop
- branch back to the top (if needed)

while, for and do while loops all fit this basic model.

Example: for loop

```
for (i=1; i<100; i++) {
  body
}
next stmt
```

Corresponding assembly

```
    loadI 1      → r1
    loadI 100    → r2
    br_GE  r1,r2 → L2
L1: body
    addI r1,1    → r1
    br_LT r1,r2  → L1
L2: next stmt
```

Exercise

Write the assembly code for the following while loop:

```
while (x >= y) {
  body
}
next stmt
```

Most modern programming languages include a break statements

- Exits from the innermost control-flow statement
  - Out of the innermost loop
  - Out of a case statement
- Solution:
  - use an unconditional branch to the next statement following the control-flow construct (loop or case statement).
  - skip or continue statement branch to the next iteration (start of the loop)

Case Statement (switch)

Case statement

```
switch (c) {
  case 'a': stmt1;
  case 'b': stmt2; break;
  case 'c': stmt3;
}
```

1. Evaluate the controlling expression
2. Branch to the selected case
3. Execute the code for that case
4. Branch to the statement after the case

Part 2 is key.

Strategies:

- Linear search (nested if-then-else)
- Build a table of case expressions and use binary search on it
- Directly compute an address (requires dense case set)

Exercise

Knowing that the character 'a' corresponds to the decimal value 97 (ASCII table), write the assembly code for the example below using linear search.

```
char c;
...
switch (c) {
  case 'a': stmt1;
  case 'b': stmt2; break;
  case 'c': stmt3; break;
  case 'd': stmt4;
}
stmt5;
```

Exercise : can you do it without any conditional jumps?

Hint: use the JR MIPS instruction which jumps directly to an address stored in a register.

Static versus Dynamic

- Static allocation: storage can be allocated directly by the compiler by simply looking at the program at compile-time. This implies that the compiler can infer storage size information.
- Dynamic allocation: storage needs to be allocated at run-time due to unknown size or function calls.

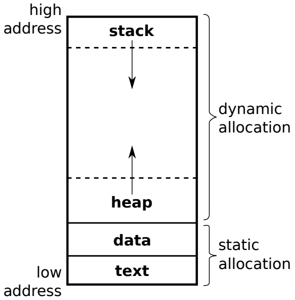
Heap, Stack, Static storage

Static storage:

- Text: contains the instructions
- Data: contains statically allocated data (*e.g.* global variables, string literals, global arrays of fixed size)

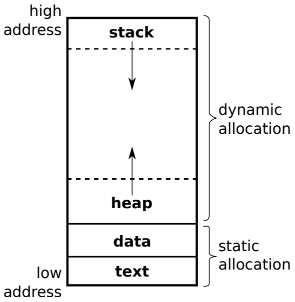
Dynamic Storage:

- Stack: used for function calls, used for local variables (if known size), register spilling (register allocation)
- Heap: used for dynamic allocation (*e.g.* malloc)



Example

```
char c;                     data
int arr[4];                 data
void foo() {
  int arr2[3];              stack
  int* ptr =                heap
    (int*) malloc(sizeof(int)*2);
  ...
  {
    int b;                  stack
    ...
    bar("hello");           data
  }
  ...
}
```



Primitive types and Arrays

Typically

- int and pointer types (*e.g.* char∗, int ∗, void∗) are 32 bits (4 byte).
- char is 1 byte

However, it depends on the data alignment of the architecture. For instance, char typically occupies 4 bytes on the stack (if the data alignment is 4 bytes).

Structure types

In a C structure, all values are aligned to the data alignment of the architecture (unless packed directive is used).

```
struct myStruct_t {
  char c;
  int x;
};
struct myStruct_t ms;
...
```

In this example, it is as if the value c uses 4 bytes of data.

```
.data
ms_myStruct_t_c:   .space   4
ms_myStruct_t_x:   .space   4

.text
...
```

Stack variable allocation

The compiler needs to keep track of where variables are allocated on the stack.

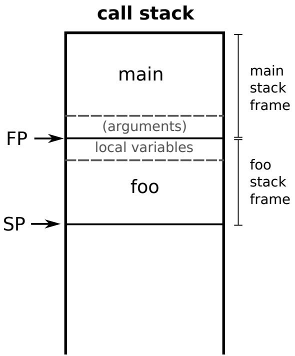
- Problem: stack pointer can move.
- Solution: use another pointer, the frame pointer

Frame pointer

- The frame pointer must be initialised to the value of the stack pointer, just when entering the function (in the prologue).
- Access to variables allocated on the stack can then be determined as a fixed offset from the frame pointer.

```
int foo() {
    ...
}
void main() {
    ...
    foo(a,b)
    ...
}
```

- The frame pointer (FP) always points to the beginning of the local variables of the current function, just after the arguments (if any).
- The stack pointer (SP) always points at the bottom of the stack, where memory is free (the stack grows downwards).



Function calls

```
int bar(int a) {
   return 3+a;
}
void foo() {
  ...
  bar(4)
  ...
}
```

- foo is the caller
- bar is the callee

What happens during a function call?

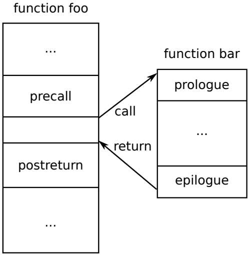
- The caller needs to pass the arguments to the callee
- The callee needs to pass the return value to the caller

But also:

- The values stored in temporaries registers needs to be saved somehow.
- Need to remember where we came from so that we can return to the call site.

General convention:

- precall: pass the arguments via dedicated registers or stack
- postreturn: read the return value from dedicated register or stack
- prologue: initialised the frame pointer and save all the temporary registers onto the stack
- epilogue: restore all the temporary registers from the stack



Other convention possible but may lead to larger code size.

Example

```
int bar(int a) {
   return 3+a;
}

bar:

addi $sp, $sp, -4    # decrement stack pointer by 4
sw   $t0, 0($sp)     # save $t0 onto the stack

li   $t0,3           # load 3 into $t0
add  $t0,$a0,$t0     # add t0 and first argument

add  $v0, $zero,$t0 # copy the result in return register
lw   $t0, 0($sp)     # restore original $t0 from stack
addi $sp, $sp, 4     # increment stack pointer by 4

jr   $ra             # jumps to return address
```

Example

```
void foo() {
   ...
   bar(4)
   ...
}

foo:
...

li    $t0, 4              # store 4 into $t0
add   $a0, $zero, $t0 # copy value into argument register

jal   bar                 # jump and link (ra=PC+8)

add   $t0, $zero, $v0 # copy returned value to $t0
...
```

Final words

What if need to pass more than 4 arguments (mips only has 4 "argument" registers by convention):

- Use the stack, by pushing the arguments in the precall
- Read the argument from the stack using the frame pointer

What if callee makes a call to another function?

- Need to save the return address of caller and frame pointer on the stack and restore after the call (should be part of precall/postreturn).

Next lecture

Instruction selection

- Peephole Matching
- Tree-pattern matching