Control in ACSE

Ettore Speziale

Politecnico di Milano

Contents

Contents

Control Statements

Control statements allows to customize the execution trace at run-time:

- if
- while
- for
- ...

They are implemented through *jumps*:

- special instructions
- allow to select the next instruction to execute at run-time

Contents

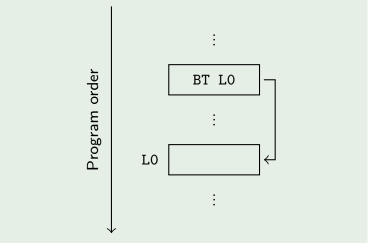
Where to Jump? I

ACSE is a syntax-directed translator:

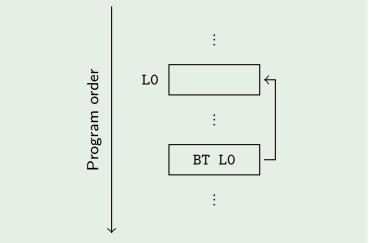
- instructions emission constrained by source code ordering

But jumps are specials:

Forward jump



Backward jump



Where to Jump? II

Forward jumps:

- conditionals, loop exits
- when we generate the jump we only known that we must jump (jump target not yet emitted)

Backward jumps:

- found in loops
- when we generate the jump we known where to jump (we have already emitted the code where we want to jump)

Where to Jump? III

To address jump translation:

- physical address vs logical location

Labels represent logical locations.

Addresses

Consider a `while` statement containing 4 instructions:

physical address 4 instructions after loop head

logical address the statement following the loop

Labels

The `axe_engine.h` contains APIs for label management:

Label management APIs

Function	Meaning
`newLabel`	create a label
`assignLabel`	bind a label to a logical address [1]
`assignNewLabel`	combined operation

Binding to physical addresses performed by ACSE.

[1] Fixing.

Exploiting Labels

Two scenarios:

Forward jump

1. create a label *lbl* when a jump is needed
2. jump to *lbl*
3. fix *lbl* when the corresponding statement is reached

Backward jump

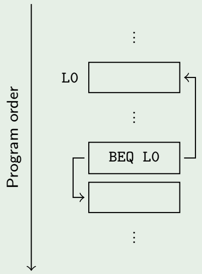
1. create and fix label at jump target
2. emit jump to *lbl* when the jump statement must be generated

Fall-through Path

Usually branches have two outgoing edges:

jump	points to the label associated with the jump instruction
fall-through	points to the next statement

Branch paths



Contents

A Real Control Structure I

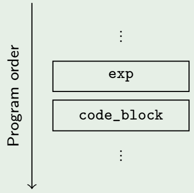
Consider the `while` statement:

`while` grammar rule

```
while_statement:
  WHILE LPAR exp RPAR code_block;
```

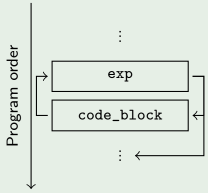
What we have:

Naked `while`



What we would have:

Theoretical flow



A Real Control Structure II

Once `exp` has been evaluated we can:

- exit the loop
- enter the loop

We need a *conditional jump* to handle such case:

- two paths: taken and not taken

At the end of the `code_block` we need to re-evaluate the loop condition:

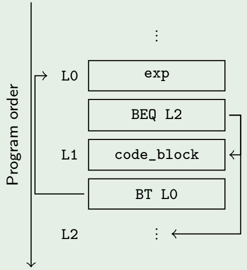
- unconditional branch to `exp` evaluation

All what we need is emitting jumps!

While Layout I

By reserving spaces for jumps the code layout is:

Code segment layout

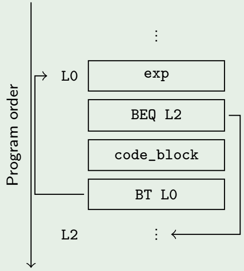


While Layout II

Edges targets three instructions:

- the fall-through edge is implicit
- can be eliminated
- we need only two labels

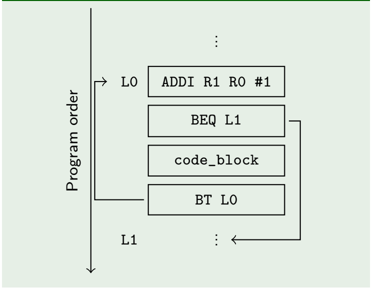
Removing useless labels



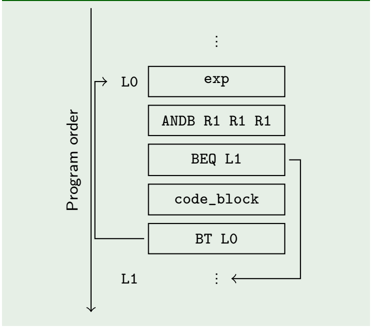
While Layout III

Since the BEQ jumps predicates over the zero bit, we must enforce its evaluation:

exp is an immediate



exp is an intermediate



While Sources

The `exp` type is known at compile-time:

- `while_statement` customization performed at compile-time

On sources (`Acse.y`):

- lookup the `while_statement` rule
- the `WHILE` token is *typed*

We need:

- an action in the middle to generate the loop exit jump
- an action to generate the backward jump and marking the statement end label

Contents

Handling Constructs

All programming languages are built around few simple constructs:

- many constructs just “syntactic sugar”

Better to type rules related to complex constructs:

- keep code clean!

Try starting with a scheme:

- to get an overview
- some minds work better with pictures

Do not redo work:

- read the ACSE headers
- some code already present (e.g. `collections.h`)

Contents

Bibliography

A. Di Biagio and G. Agosta.
Advanced Compiler System for Education.
http://compilergroup.elet.polimi.it, 2008.

Formal Languages and Compilers Group.
Software Compilers.
http://compilergroup.elet.polimi.it, 2010.