Formal Languages and Compilers

ACSE: control statements

Alessandro Barenghi & Michele Scandale

January 8, 2020

ACSE: Control statements

This lecture is about **control statements**: they allow to have different execution traces at runtime on different inputs.

```
control_statement : if_statement
                  | while_statement
                  | do_while_statement SEMI
                  | return_statement SEMI
                  ;
```

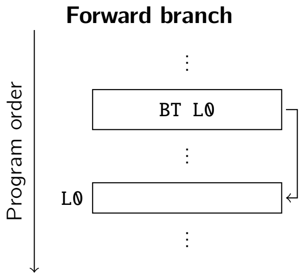
In ACSE only a few basic control statements are implemented: complex ones are just specializations and/or extensions.

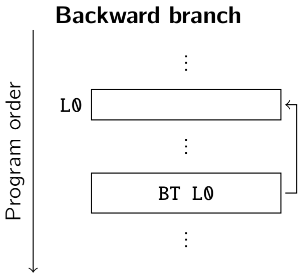
Important: control statements are implemented with branch instructions.

ACSE: Branches

Branch instructions are used to select which instruction must be executed after them. We will consider only *PC-relative* branches: the destination is specified as an offset from the PC:

forward	A positive offset indicates a forward jump
backward	A negative offset indicates a backward jump





`ACSE`: Branches

Remember that `ACSE` is a **syntax directed transducer**: code is generated while parsing.

Forward branches:

- when we generate them we only know that we must jump, but the destination instruction has not been emitted yet
- used in conditionally executed code blocks (e.g. `if` constructs, loop exits)

Backward branches:

- when we generate them we know where to jump, the destination has already been emitted
- typically used in loops (the backedge)

`ACSE`: Branches and Labels

Managing directly branch targets by hand is challenging. To simplify this task a common solution is to introduce **symbolic addresses** or **labels**.

To manage labels there are three useful APIs (see `axe_engine.h`):

- `newLabel` declares a new label in the compiler context
- `assignLabel` binds a label to the current position in the code (*logical address* of the next instruction to be emitted)
- `assignNewLabel`, does both of them, in order

The translation from *logical* to *physical* addresses is performed by the ACSE backend.

ACSE: Branches and Labels

Forward jump

- reserve a label *lbl* when a jump is needed
- jump to *lbl*
- fix *lbl* when the corresponding statement is reached

Backward jump

- reserve and fix label when the jump target is emitted
- emit jump to *lbl* when the jump statement must be generated

Fall-through path Conditional branches have two destinations:

- the label operand, when the branch is taken
- the next instruction if the branch is not taken (the *fall-through* path)

ACSE: `while` statement

Let's analyze the code generation for the **`while`** statement.

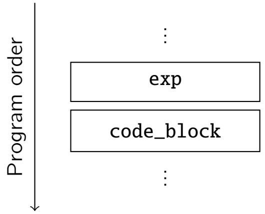
```
while_statement : WHILE LPAR exp RPAR code_block
                ;
```

ACSE: while statement

Let's analyze the code generation for the **while** statement.

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

Generated code chunks

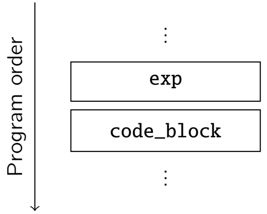


ACSE: `while` statement

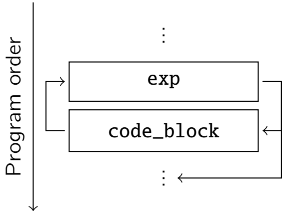
Let's analyze the code generation for the **`while`** statement.

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

Generated code chunks



Expected control flow



ACSE: `while` statement

Once **`exp`** has been evaluated we can:

- enter the loop
- skip the loop

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

- two paths: taken and not taken

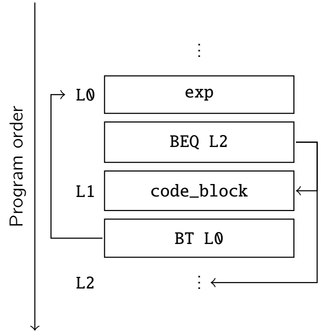
At the end of the **`code_block`** we need to re-evaluate the loop condition:

- unconditional branch to **`exp`** evaluation

All we need is emitting jumps!

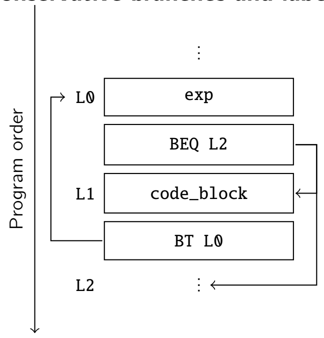
ACSE: `while` statement

Conservative branches and labels



ACSE: while statement

Conservative branches and labels

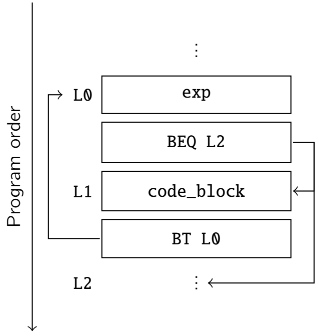


The control flow graph can be simplified:

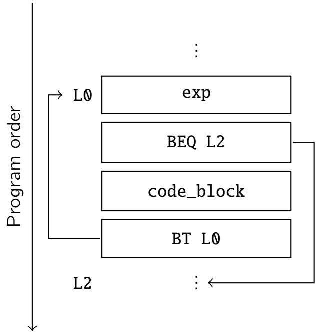
- the fall-through edge is implicit
- we can remove both its edge and its label

ACSE: `while` statement

Conservative branches and labels



Minimal branches and labels



The control flow graph can be simplified:

- the fall-through edge is implicit
- we can remove both its edge and its label

ACSE: `while` statement

Conditional branches read the PSW in order to verify the value of their condition. We must ensure that the PSW contains the correct value when the jump is executed.

Issues arise when the condition of a branch is the value of an *expression*:

- `t_axe_expression` handle both immediates an registers
- remember that costant folding may yield an immediate expression as a result

The *immediate* case is generally handled specializing the code:

- We know the expression value at compile time
- We decide at compile time whether to generate or not an unconditional branch

By contrast, the *register* case usually is handled as follow:

```
t_axe_expresion my_exp;

if (my_exp.expression_type == REGISTER)
  gen_andb_instruction(program, my_exp.value, my_exp.value,
                       my_exp.value, CG_DIRECT_ALL);
```

ACSE: while statement

```
while_statement : WHILE {
                    $1 = create_while_statement();
                    $1.label_condition = assignNewLabel(program);
                  }
                  LPAR exp RPAR {
                    if ($4.expression_type == IMMEDIATE)
                      gen_load_immediate(program, $4.value);
                    else
                      gen_andb_instruction(program, $4.value, $4.value,
                                                    $4.value, CG_DIRECT_ALL);

                    $1.label_end = newLabel(program);
                    gen_beq_instruction (program, $1.label_end, 0);
                  }
                  code_block {
                    gen_bt_instruction(program, $1.label_condition, 0);
                    assignLabel(program, $1.label_end);
                  }
                ;
```

Note: `WHILE` token has a semantic value in order to store informations accessible from the semantic actions while avoiding global variables.
Remember that a semantic value is accesible only if its grammar symbol is on the parsing stack!

Handling Constructs

All programming languages are built around a few simple constructs:

- those not already present in ACSE can be found in the exam solutions

Typing nonterminals related to complex constructs helps:

- keeps the code clean, puts the variables in the correct scope

Try starting with a sketch of the code structure:

- provides a quick overview
- a graphical representation always helps

Do not rewrite available code:

- understand the facilities in the ACSE headers
- list handling already available (e.g. `collections.h`)

Class task I

Let's implement a **forall** loop:

```
int i, x;
read(x);

forall (i = 0 to 10 + x)
  write(i);

forall (i = 10 + x downto 0)
  write(i);
```

The *forall* loop defines an induction variable that evolves in the loop from the initial value *I* with unitary (positive or negative) steps until it reaches the limit value *L*.
Note: the statement must allow nesting!
Note: you can assume or not that the number of iterations of the loop (tripcount) is statically computed at the beginning of the loop.

Class task II

Let's extend the previous exercise adding the **break** and **continue** statements w.r.t. forall loops.

```
int i, x;
read(x);

forall (i = 0 to 10 + x) {
  write(i);
  if (i > 5) continue;

  forall (j = 10 + x downto 0)
    if (i + j / 3 > 2) break;
    write(j);
}
```

Class task II

Let's extend the previous exercise adding the **break** and **continue** statements w.r.t. forall loops.

```
int i, x;
read(x);

forall (i = 0 to 10 + x) {
  write(i);
  if (i > 5) continue;

  forall (j = 10 + x downto 0)
    if (i + j / 3 > 2) break;
    write(j);
}
```

Hint: we may need an external stack to keep information about forall-loop labels in order to know the branch destination when generating the code of **break** and **continue** statements.

Class task III

Let's implement a C-style **for** loop:

```
int i, j;

for (i = 0, j = 100; i < 10 && j > 85; i = i +1, j = j - 2) {
  write(i + j);
  write(i - j);
}
```

You can assume that:

- the first clause is used to initialize variables
- the second clause is a condition expression that must hold to execute the loop body
- the last clause is used to update variables

Hint: a *for*-loop is equivalent to a while loop defined as follows:

```
int i, j;

i = 0, j = 100;
while (i < 10 && j > 85) {
  write(i + j);
  write(i - j);
  i = i +1;
  j = j - 2;
}
```