Introducing ACSE

Alessandro Barenghi   Ettore Speziale   Michele Tartara

Politecnico di Milano

Contents

Contents

Advanced Compiler System for Education

It is our simple compiler front-end:

- accepts a C-like language
- generates a RISC-like intermediate code

Usually, the lab test requires:

- to add tokens to the accepted language
- to accept new statements
- to translate new statements into intermediate code

Getting ACSE

- available on course site [2]
- download, unzip, make and play
- full manual available [1]

Quick Start I

Build a simple hello world:

hello.src

```
write(72);   // H
write(101);  // e
write(108);  // l
write(108);  // l
write(111);  // o
write(33);   // !
```

Compile and run

```
$ acse hello.src
$ asm output.asm
$ mace output.o
72
101
108
108
111
33
```

Quick Start II

Three tools:

- compiler to assembly (`acse`)
- assembler to machine code (`asm`)
- interpreter (`mace`)

In this course we modify the first:

- last two allow to try your programs

A dump of intermediate representation are `.cfg` files [1]:

- easy to see your edits here

[1] Produced by `acse`.

Sources

The ACSE sources are contained into the `acse` directory:

- well commented
- easy to understand

All data structures accessible through the `program` global:

- a huge number of helper functions allows to perform common operations (e.g. getting a new temporary register) without using the low level interface
- related helpers grouped in the same module
- module headers heavily documented

Contents

Which tongue does ACSE speak?

ACSE:

1. reads LANCE
2. produces an intermediate assembly
3. emits MACE assembly

Languages are very simple:

- should be easy to understand

For a complete reference see the manual [1].

LANguage for Compiler Education

A very small subset of C99:

- standard set of arithmetic/logic/relational operators
- reduced control flow statements (`while`, `do while`, `if`)
- a scalar type (`int`)
- unidimensional arrays of integers

Very limited support to I/O:

reading `read(var)` stores into `var` an integer read from `stdin`

writing `write(var)` write `var` to `stdout`

Intermediate Representation

LANCE code is first translated into a RISC-like language:

- few essential computing instructions (e.g. `ADD`)
- memory instructions (e.g. `LOAD`)
- jumps (e.g. `BEQ`)
- special I/O instructions (e.g. `READ`)

Two addressing modes:

direct data inside the register

indirect data at memory location pointed by register

Data storage:

- unbounded registers
- unbounded memory

How to Read the Manual I

Instructions come into four flavors:

Instructions classification

Type	Operands	Example
Ternary [2]	1 destination and 2 source registers	`ADD R3 R1 R2`
Binary	1 destination and 1 source register, 1 immediate operand	`ADDI R3 R1 #4`
Unary	1 destination register, 1 address operand	`LOAD R1 L0`
Jump	1 address operand	`BEQ L0`

How to Read the Manual II

Operands:

Operands Syntax

Type	Syntax	Notes
Directed addressing with register	`Rn`	The n-th register
Undirected addressing with register	`(Rn)`	Data whose address is store into the n-th register
Address	`Ln`	The address identifier by the n-th label [3]
Immediate	`#n`	The scalar integer constant *n*

[2] Destination and second source indirectly addressable.

[3] More on this later.

Register Notes I

There are two special registers:

zero `R0` contains the 0 constant, cannot be written

status implicitly read/written by some instructions, not directly accessible

The status register contains four bits [4]:

- negative
- zero
- overflow
- carry

Register Notes II

Special registers are essential:

Constant loading

```
ADDI R3 R0 #5
```

Since `R0` always contains 0, `R3` is filled with 5

Branch is taken only when the zero bit in the status register isn't set:

- zero bit implicity set by `SUB` when its result is 0

Conditional jumping

```
SUBI R3 R1 1
BNE L0
```

[4] Heavily exploited by jumps.

Addressing Modes by Example

This should be known, anyway . . . :

Direct addressing: `ADD R3 R1 R2`

Before

Register file		Memory	
*R*1	1	0x*d*	0
*R*2	2	0x*e*	0
*R*3	0	0x*f*	0

↓

After

Register file		Memory	
*R*1	1	0x*d*	0
*R*2	2	0x*e*	0
*R*3	3	0x*f*	0

Indirect addressing: `ADD R3 R1 (R2)`

Before

Register file		Memory	
*R*1	1	0x*d*	2
*R*2	0x*d*	0x*e*	0
*R*3	0	0x*f*	0

↓

After

Register file		Memory	
*R*1	1	0x*d*	2
*R*2	0x*d*	0x*e*	0
*R*3	3	0x*f*	0

Contents

Reading I

To parse we need:

scanner see `Acse.lex`

parser see `Acse.y`

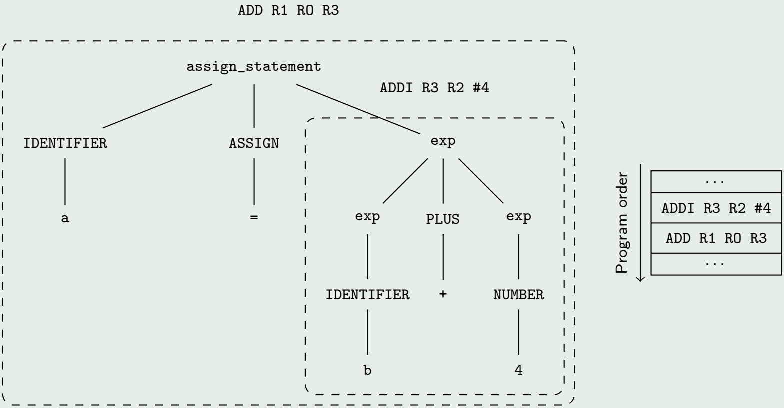
ACSE is a *syntax directed translator*:

- translation is performed while parsing LANCE files
- once an instruction is emitted, you cannot go back

Reading II

A simple example:

Translating a = b + 4



Variables I

A LANCE variable is matched by the `IDENTIFIER` token:

- custom typed to a `char*`, the name of the variable

Type declaration with bison

```
%union {
  ...
  char* svalue;
  t_axe_expression expr;
  ...
}
```

Variables II

Semantic values are initialized by the scanner:

Saving identifier names

```
{ID} {
        yylval.svalue = strdup(yytext);
        return IDENTIFIER;
    }
```

Variables III

Bindings declared inside `Acse.y`:

Rules binding

```
...
%token <svalue> IDENTIFIER
...
%type <expr> exp
...
```

- the same for other constructs (e.g. numbers)
- non-terminals can be typed too (e.g. `exp`)

More Info about Variables

Internal representation of variables:

ACSE variable representation

```
typedef struct t_axe_variable {
   ...
   int isArray;
   int arraySize;
   ...
   char* ID;
   ...
} t_axe_variable;
```

To get here, use `getVariable` [5].

[5] In `axe_engine.h`.

Scalars I

Scalar variables management:

symbol table low level interface, almost useless for this course

helpers into `axe_utils.h` many high level functions

Thumb rule:

- each scalar variable is stored in a register

Scalars II

Let's try to print a scalar [6]:

Writing an integer

```
int a;
write(a);
```

Intermediate

```
WRITE R1 0
HALT
```

Scalars III

How does ACSE translate the code?

Touched ACSE code - Write rule [7]

```
write_statement:
  WRITE LPAR exp RPAR {
    ...
    location = $3.value;
    gen_write_instruction(program,
                          location);
  }
```

Scalars IV

Touched ACSE code - Expression rule

```
exp: NUMBER { ... }
   | IDENTIFIER {
     int location;
     location = get_symbol_location(
                    program, $1, 0);
     $$ = create_expression(location,
                         REGISTER);
     free($$);
   }
...
```

[6]Implicitly initialized to 0.

[7]Simplified view.

Arrays I

Internal representation: base plus offset:

- no need to known technical details
- `axe_array.h` contains helpers for common operations

Arrays II

Now, try printing an array element:

Array output

```
int a[10];
write(a[1]);
```

Intermediate

```
MOVA R1 L0
ADDI R1 R1 #1
ADD R2 R0 (R1)
WRITE R2 0
HALT
```

Arrays III

And inside ACSE?

Touched ACSE code - Expression rule [8]

```
exp: NUMBER { ... }
...
    | IDENTIFIER LSQUARE exp RSQUARE {
      int reg;
      reg = loadArrayElement(program,
                             $1, $3);
      $$ = create_expression(reg,
                             REGISTER);
      free($$);
    }
...
```

[8] Obviously, write rule still touched.

Contents

Bibliography

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

Formal Languages and Compilers Group.
Formal Languages and Compilers – CorsiOnline.
http://corsi.metid.polimi.it, 2010.