Flex , Bison and the **ACSE** compiler suite

Marcello M. Bersani

LFC – Politecnico di Milano

LANCE in a nutshell

- Data type: int, array
- Basic I/O: read(), write()
- Arithmetic expressions
- If-then-else
- While
- Do-while

ACSE data structure

```
typedef struct t_program_infos
{
  t_list *variables;
  t_list *instructions;
  t_list *data;
  t_axe_label_manager *lmanager;
  t_symbol_table *sy_table;
  int current_register;
} t_program_infos;
```

- Axe_engine.h

DO-WHILE

```
do
   code_block
while (exp)
```

```
Lk:   cb_i1
      cb_i2
      ...
      cb_ih
      exp_i1
      ...
      exp_in
      load/andb
      bne Lk
```

```
do_while_statement  : DO
   {
      $1 = newLabel(program);
      assignLabel(program, $1);
   }
   code_block WHILE LPAR exp RPAR
   {
      if ($6.expression_type == IMMEDIATE)
            gen_load_immediate(program, $6.value);
      else
            gen_andb_instruction(program,$6.value,$6.value, $6.value, CG_DIRECT_ALL);

      gen_bne_instruction (program, $1, 0);
   }
;
```

DO-WHILE

- Axe_utils.c

```
int gen_load_immediate(t_program_infos *program, int immediate)
{
  int imm_register;

  imm_register = getNewRegister(program);
  gen_addi_instruction(program, imm_register, REG_0, immediate);

  return imm_register;
}
```

Do-while data structure

```
%union {
   …
   t_axe_label *label;
}

%token <label> DO
```

ACSE grammar overview

```
program  : var_declarations statements                    {...};

var_declarations : var_declarations var_declaration   { /* does nothing */ }
                 | /* empty */                          { /* does nothing */ }
;

var_declaration   : TYPE declaration_list SEMI                          {...};

declaration_list [t_list]  : declaration_list COMMA declaration      {...}
                           | declaration                                        {...}
;

declaration [t_axe_declaration] : IDENTIFIER ASSIGN NUMBER                  {...}
                                | IDENTIFIER LSQUARE NUMBER RSQUARE  {...};
                                | IDENTIFIER                                {...}
;
```

ACSE grammar overview

```
code_block  : statement                          { /* does nothing */ }
            | LBRACE statements RBRACE           { /* does nothing */ }
;

statements  : statements statement               { /* does nothing */ }
            | statement                          { /* does nothing */ }
;

statement   : assign_statement SEMI              { /* does nothing */ }
            | control_statement                  { /* does nothing */ }
            | read_write_statement SEMI          { /* does nothing */ }
            | SEMI                               { gen_nop_instruction(program);}
;
```

ACSE grammar overview

```
control_statement : if_statement                { /* does nothing */ }
                  | do_while_statement SEMI     { /* does nothing */ }
                  | while_statement             { /* does nothing */ }
                  | return_statement SEMI       { /* does nothing */ }
;

read_write_statement : read_statement           { /* does nothing */ }
                     | write_statement          { /* does nothing */ }
;

assign_statement : IDENTIFIER LSQUARE exp RSQUARE ASSIGN exp {...}
                 | IDENTIFIER ASSIGN exp                     {...}
```

ACSE grammar overview

```
if_statement  : if_stmt                          {...}
              | if_stmt ELSE                     {...}
                code_block                       {...}
;

if_stmt [t_axe_label] :  IF [t_axe_label]        {...}
                         LPAR exp RPAR           {...}
                         code_block              {...}
;
```

ACSE grammar overview

```
while_statement  : WHILE [t_while_statement]      {...}
                         LPAR exp RPAR            {...}
                         code_block               {...}
;

do_while_statement  : DO  [t_axe_label]            {...}
                          code_block WHILE LPAR exp RPAR  {...}
;
```

ACSE grammar overview

exp [**t_axe_expression**] : NUMBER {...}

| IDENTIFIER {...}

| IDENTIFIER LSQUARE exp RSQUARE {...}

| NOT_OP NUMBER {...}

| NOT_OP IDENTIFIER {...}

| exp AND_OP exp {...}

...

| LPAR exp RPAR { $$ = $2; }

| MINUS exp {...}

Expressions

- Each expression is associated with a unique register

```
typedef struct t_axe_expression
{
   int value;        /*immediate or register identifier*/
   int expression_type;       /* only integer */
} t_axe_expression;
```

Defined in axe_expression.h

Expressions

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

- These rules do not produce MACE code!
  - Symbolical representation

Expressions - arrays

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

- **loadArrayElement** returns the register where the selected element is stores
- axe_array.c manages LOAD/STORE operations of element of array

Expr data structure

```
%union {
   …
   char *svalue;
   t_axe_expressions *expr;
}

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

Expressions - arrays

```
int loadArrayElement(t_program_infos *program, char *ID,
    t_axe_expression index){

  int load_register;
  int address;
  address = loadArrayAddress(program, ID, index);
  load_register = getNewRegister(program);
  gen_add_instruction(program, load_register, REG_0, address,
    CG_INDIRECT_SOURCE);
  return load_register;
}
```

Expressions - arrays

```
void storeArrayElement(t_program_infos *program, char *ID,
    t_axe_expression index, t_axe_expression data){
  int address;
  address =  loadArrayAddress(program, ID, index);

  if (data.expression_type == REGISTER) {
     gen_add_instruction(…, address, REG_0, data.value, CG_INDIRECT_DEST);
  }  else  {
    int imm_register;
    imm_register = gen_load_immediate(program, data.value);
    gen_add_instruction(…,address,REG_0,imm_register,CG_INDIRECT_DEST);
  }
}
```

Expressions - arrays

```
int loadArrayAddress(t_program_infos *program, char *ID, t_axe_expression index){
  int mova_register;
  t_axe_label *label;
…
  label = getLabelFromVariableID(program, ID);
…
  mova_register = getNewRegister(program);
  gen_mova_instruction(program, mova_register, label, 0);
  if (index.expression_type == IMMEDIATE){
    if (index.value != 0){
      gen_addi_instruction (program, mova_register, mova_register, index.value);
    }
  } else {
 …
    gen_add_instruction(program, mova_register, mova_register, index.value, CG_DIRECT_ALL);
  }
  return mova_register;
}
```

Expressions

```
exp: …| NOT_OP NUMBER    {  if ($2 == 0)
                              $$ = create_expression (1, IMMEDIATE);
                            else
                              $$ = create_expression (0, IMMEDIATE);
      }
      | NOT_OP IDENTIFIER  {
                              int identifier_location;
                              int output_register;

                              identifier_location =  get_symbol_location(program, $2, 0);
                             output_register = getNewRegister(program);
                             gen_notl_instruction (program, output_register, identifier_location);
                              $$ = create_expression (output_register, REGISTER);
                             free($2);
       }
```

Expressions

```
exp: ...| MINUS exp
        {
            if ($2.expression_type == IMMEDIATE){
                $$ = $2;
                $$.value = - ($$.value);
          } else {
              t_axe_expression exp_r0;

              exp_r0.value = REG_0;
              exp_r0.expression_type = REGISTER;
              $$ = handle_bin_numeric_op (program, exp_r0, $2,SUB);
           }
        }
```

Expressions

- Wrappers in axe_expression.c

t_axe_expression **handle_bin_numeric_op** (t_program_infos *program, t_axe_expression exp1, t_axe_expression exp2, int binop);

t_axe_expression **handle_binary_comparison** (t_program_infos *program, t_axe_expression exp1, t_axe_expression exp2, int condition);

Expressions

```
exp: ...
| exp DIV_OP exp     {
     $$ = handle_bin_numeric_op(program, $1, $3, DIV);
   }
| exp LT exp      {
     $$ = handle_binary_comparison(program, $1, $3,
    _LT_);
| ...
```

Expressions

```
t_axe_expression handle_binary_comparison (t_program_infos *program , t_axe_expression exp1, t_axe_expression exp2, int condition){
…
if (  (exp2.expression_type == IMMEDIATE) && (exp1.expression_type == IMMEDIATE) ){
     return handle_bin_comparison_Imm(exp1.value, exp2.value, condition);
 }

output_register = getNewRegister(program);

if (exp2.expression_type == IMMEDIATE) {
   gen_subi_instruction (program, output_register, exp1.value, exp2.value);
 }
 else if (exp1.expression_type == IMMEDIATE){
     gen_subi_instruction (program, output_register, exp2.value, exp1.value);
     gen_neg_instruction (program, output_register, output_register, CG_DIRECT_ALL);
 }
 else{
    gen_sub_instruction (program, output_register, exp1.value, exp2.value, CG_DIRECT_ALL);
 }

switch(condition) {
     case _LT_ : gen_slt_instruction (program, output_register); break;
     case _GT_ : gen_sgt_instruction (program, output_register); break;
     …
  }
 return create_expression (output_register, REGISTER);
}
```

Assignments

```
assign_statement :
    IDENTIFIER LSQUARE exp RSQUARE ASSIGN exp
    {
        storeArrayElement(program, $1, $3, $6);
        free($1);
    }
    | IDENTIFIER ASSIGN exp
    {
        int location; t_axe_instruction *instr;
        location = get_symbol_location(program, $1, 0);
        if ($3.expression_type == IMMEDIATE)
            gen_addi_instruction(program, location, REG_0, $3.value);
        else
            gen_add_instruction(…, location, REG_0, $3.value, CG_DIRECT_ALL);
        free($1);
    };
```

Read/Write

```
read_statement : READ LPAR IDENTIFIER RPAR  {
    int location;

    location = get_symbol_location(program, $3, 0);
    gen_read_instruction (program, location);
    free($3);
}
```

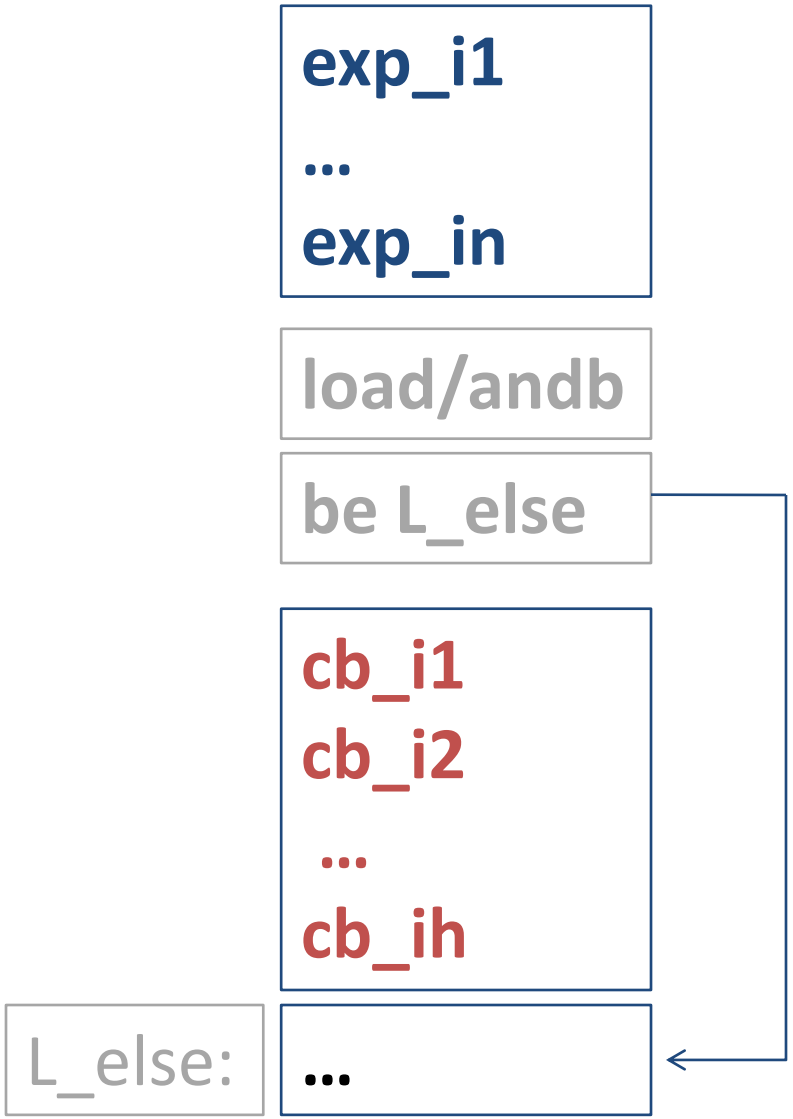
Read/Write

```
write_statement : WRITE LPAR exp RPAR  {
   int location;

   if ($3.expression_type == IMMEDIATE)
     location = gen_load_immediate(program, $3.value);
   else
      location = $3.value;
    gen_write_instruction (program, location);
}
```

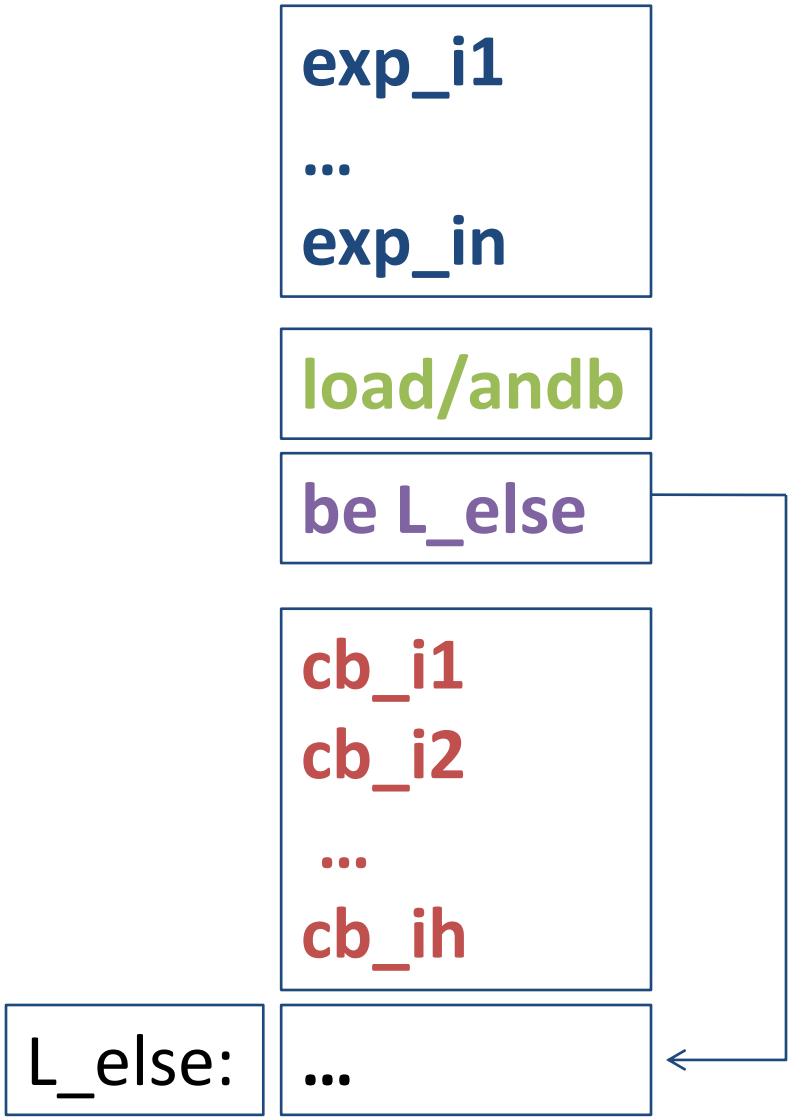
If-then

If (**exp**)
    code_block



If-then

If (**exp**)
    code_block



If-then

If (**exp**)
code_block

```
if_stmt: IF{
          $1 = newLabel(program);
        }
        LPAR exp RPAR
        {
          if ($4.expression_type == IMMEDIATE)
             gen_load_immediate(program, $4.value);
          else
             gen_andb_instruction(..., $4.value, $4.value, $4.value, ...);
             gen_beq_instruction (program, $1, 0);
         }
         code_block { $$ = $1; }
```

exp_i1
…
exp_in

load/andb

be L_else

cb_i1
cb_i2
…
cb_ih

L_else:

…

If-then

```
if (exp)
    code_block
```

```
if_stmt: IF{
            $1 = newLabel(program);
         }
         LPAR exp RPAR
         {
           if ($
              gen_
           el
              gen
              gen_
         }
         code_block { $$ = $1; }
```

No runtime code is emitted by this action

exp_i1
…
exp_in

load/andb

value, $4.value, $4.value, …);

program, $1, 0);

be L_else

cb_i1
cb_i2
…
cb_ih

L_else:

…

If-then

```
if_stmt: IF{
          $1 = newLabel(program);
        }
        LPAR exp RPAR
        {
          if ($4.expression_type == IMMEDIATE)
             gen_load_immediate(program, $4.value);
          else
             gen_andb_instruction(..., $4.value, $4.value, $4.value, ...);
             gen_beq_instruction (program, $1, 0);
         }
         code_block { $$ = $1; }
```

```
if (exp)
    code_block
```

```
exp_i1
...
exp_in
```

```
load/andb
```

```
be L_else
```

```
cb_i1
cb_i2
...
cb_ih
```

```
L_else:
```

```
...
```

If-then

```
if (exp)
    code_block
```

```
if_stmt: IF{
            $1 = newLabel(program);
         }
         LPAR exp RPAR
         {
           if ($4.expression_type == IMMEDIATE)
              gen_load_immediate(program, $4.value);
           else
              gen_andb_instruction(..., $4.value, $4.value, $4.value, ...);
              gen_beq_instruction (program, $1, 0);
          }
          code_block { $$ = $1; }
```

```
exp_i1
...
exp_in
```

```
load/andb
```

```
be L_else
```

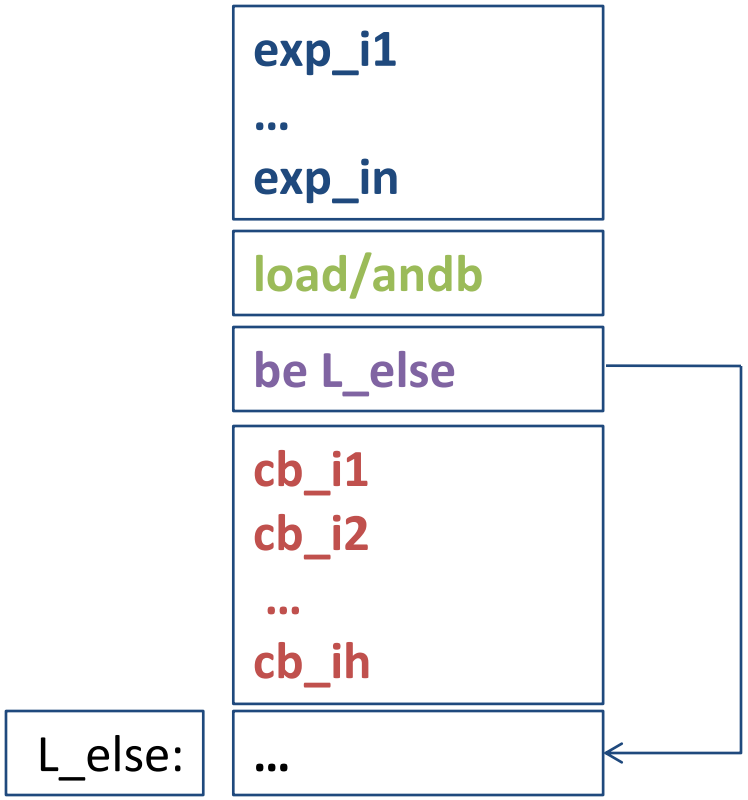
```
cb_i1
cb_i2
 ...
cb_ih
```

```
L_else:
```

```
...
```

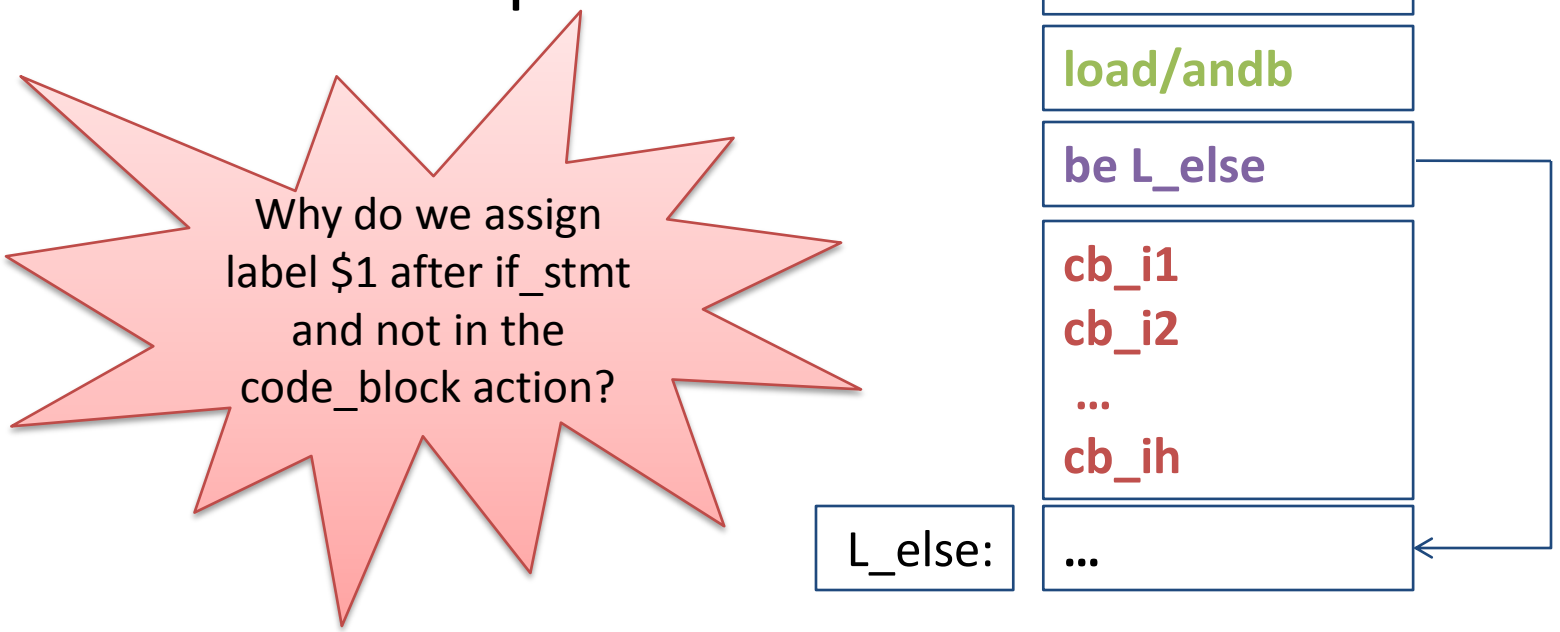
If-then

```
if_statement: if_stmt {
                    assignLabel(program, $1);
              }
            | ...
```



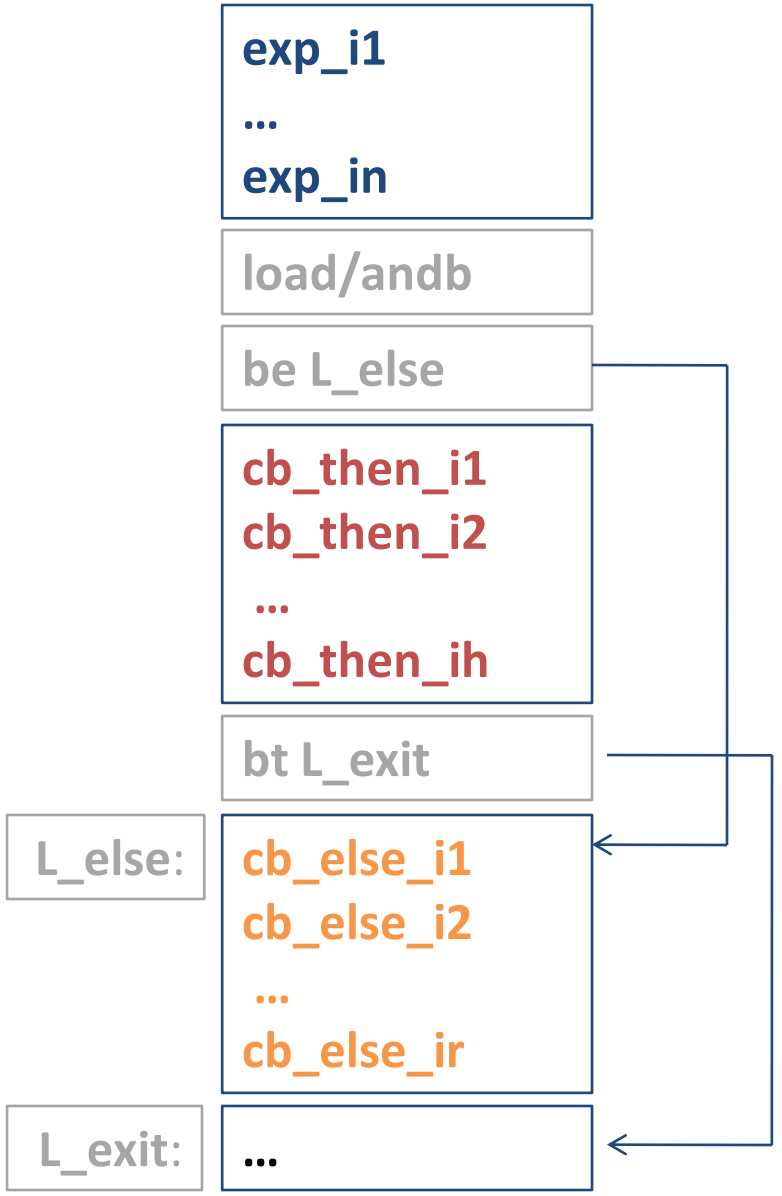
If-then

```
if_statement: if_stmt {
                    assignLabel(program, $1);
              }
            | ...
```



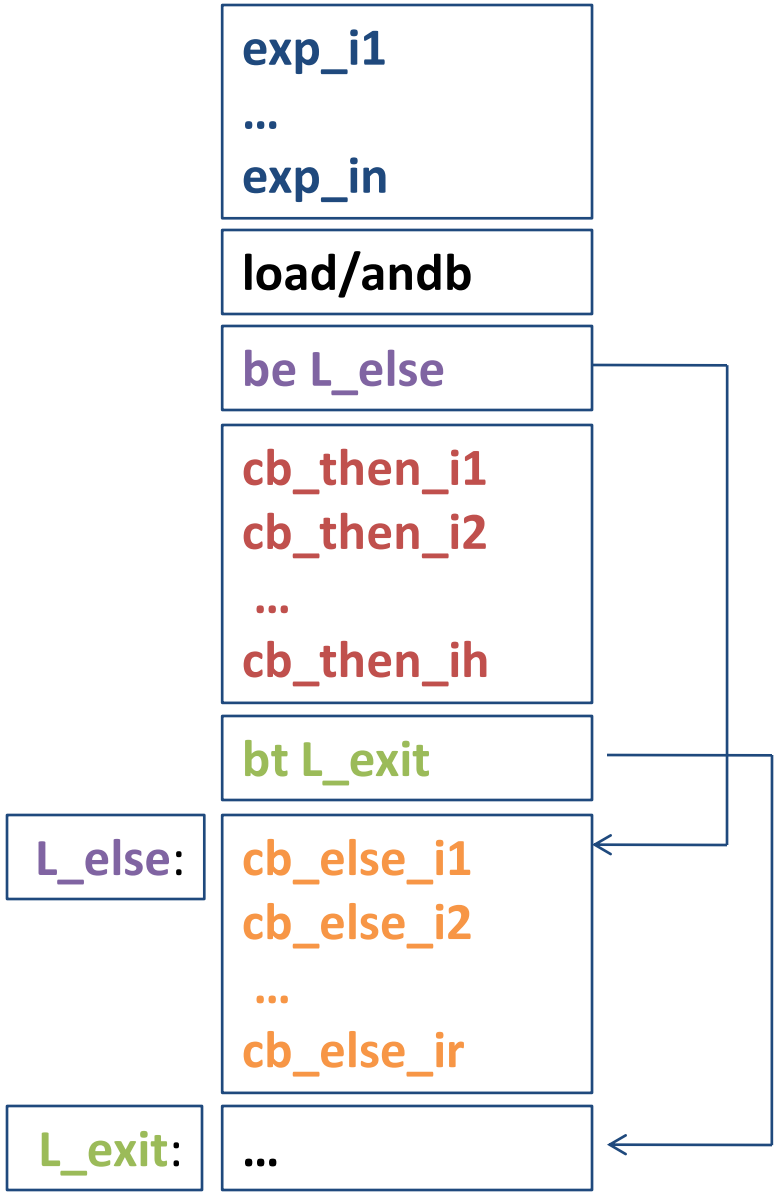
If-then-else

```
If (exp)
    code_block
else
    code_block
```



If-then-else

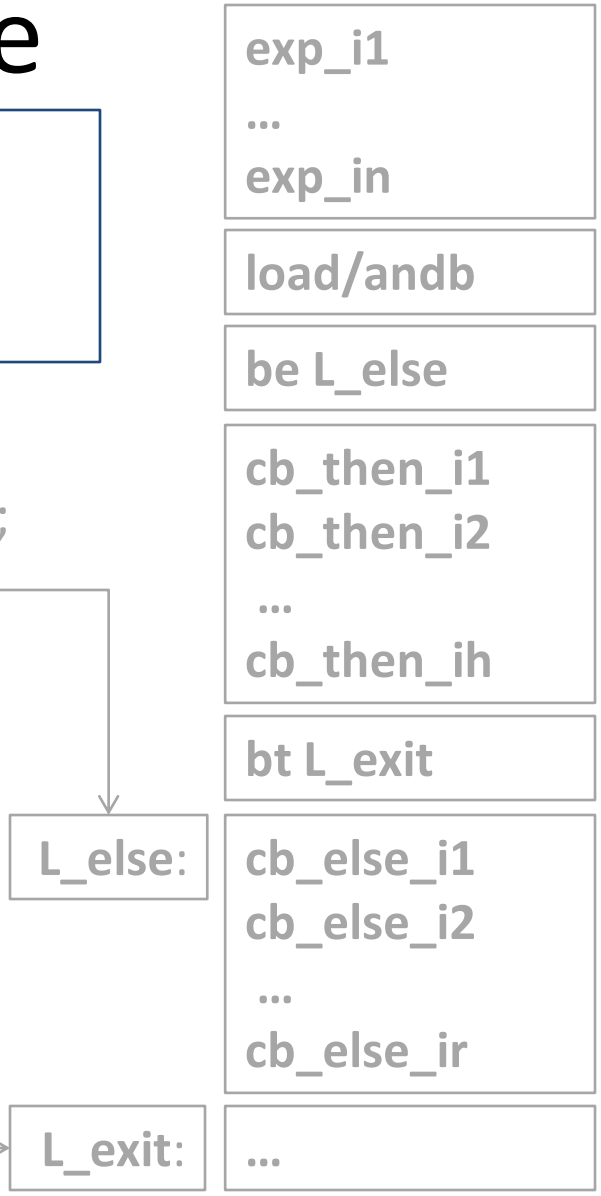
```
If (exp)
    code_block
else
    code_block
```



If-then-else

```
if_statement   : …
               | if_stmt ELSE
               {
                 $2 = newLabel(program);
                 gen_bt_instruction (program, $2, 0);
                 assignLabel(program, $1);
               }
               code_block
               {
                 assignLabel(program, $2);
               }
;
```

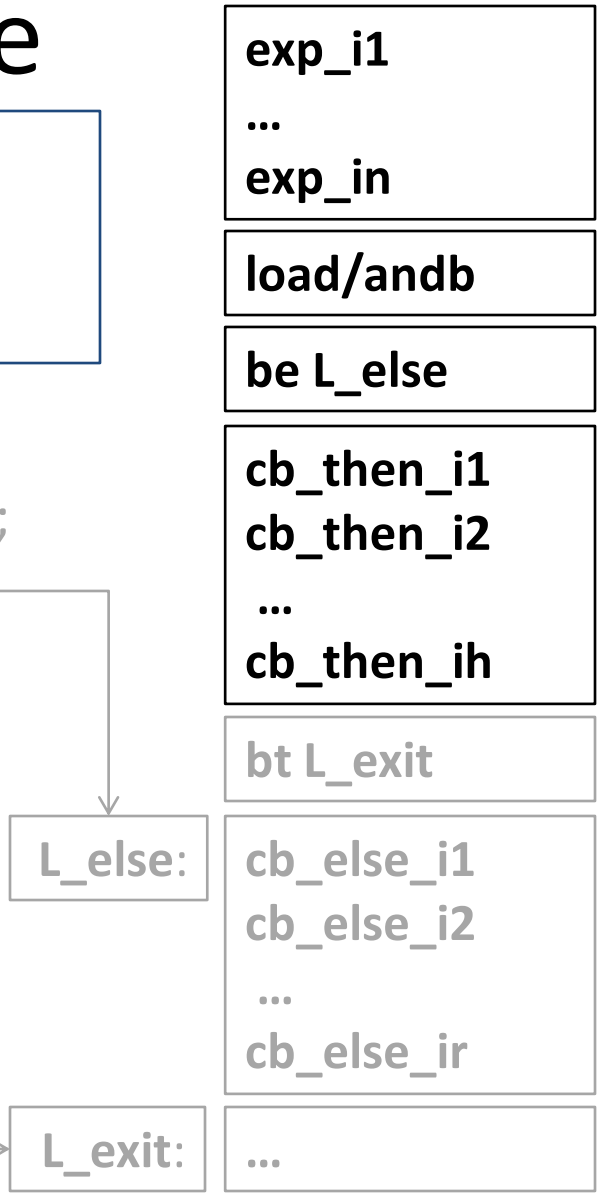
```
If (exp)
   code_block
else
   code_block
```



If-then-else

```
if_statement   : …
          | if_stmt ELSE
          {
            $2 = newLabel(program);
            gen_bt_instruction (program, $2, 0);
            assignLabel(program, $1);
          }
          code_block
          {
            assignLabel(program, $2);
          }
;
```

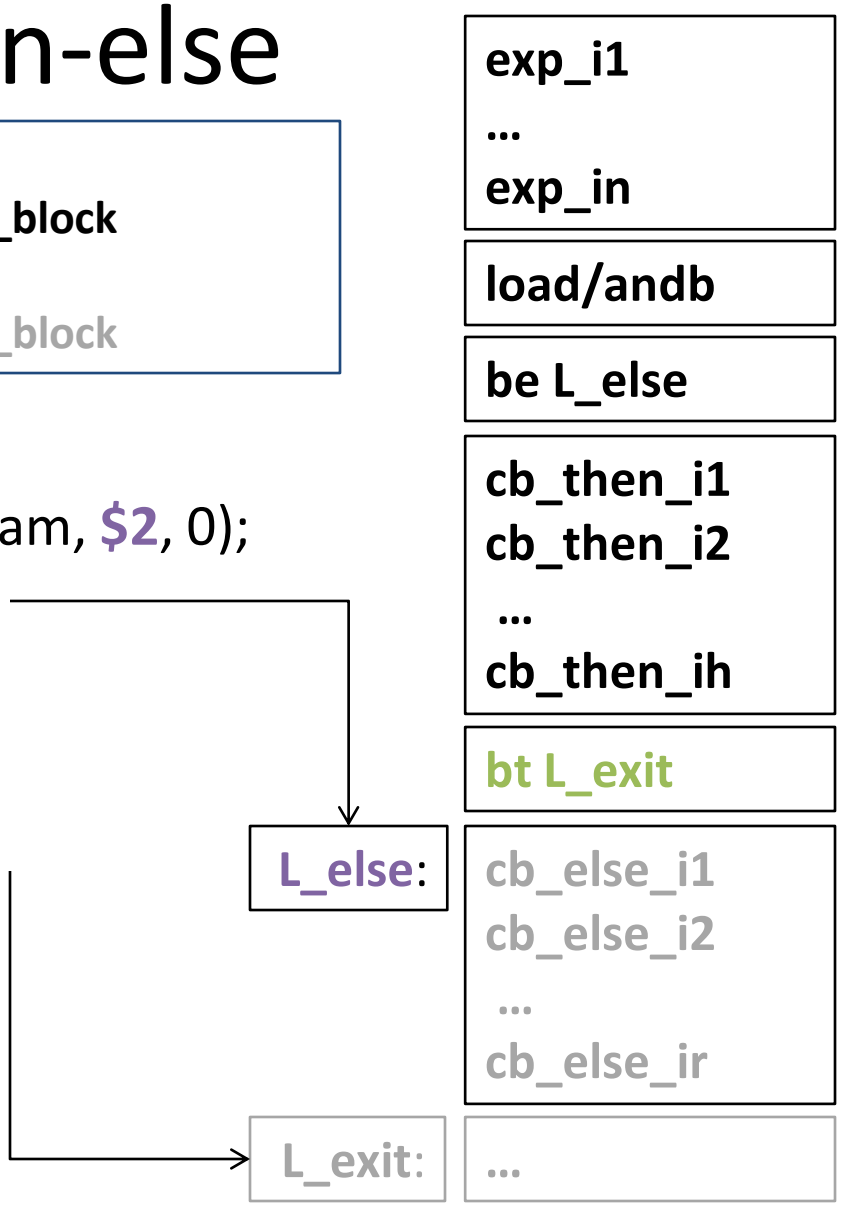
```
If (exp)
   code_block
else
   code_block
```



If-then-else

```
if_statement   : …
          | if_stmt ELSE
          {
            $2 = newLabel(program);
            gen_bt_instruction (program, $2, 0);
            assignLabel(program, $1);
          }
          code_block
          {
            assignLabel(program, $2);
          }
;
```

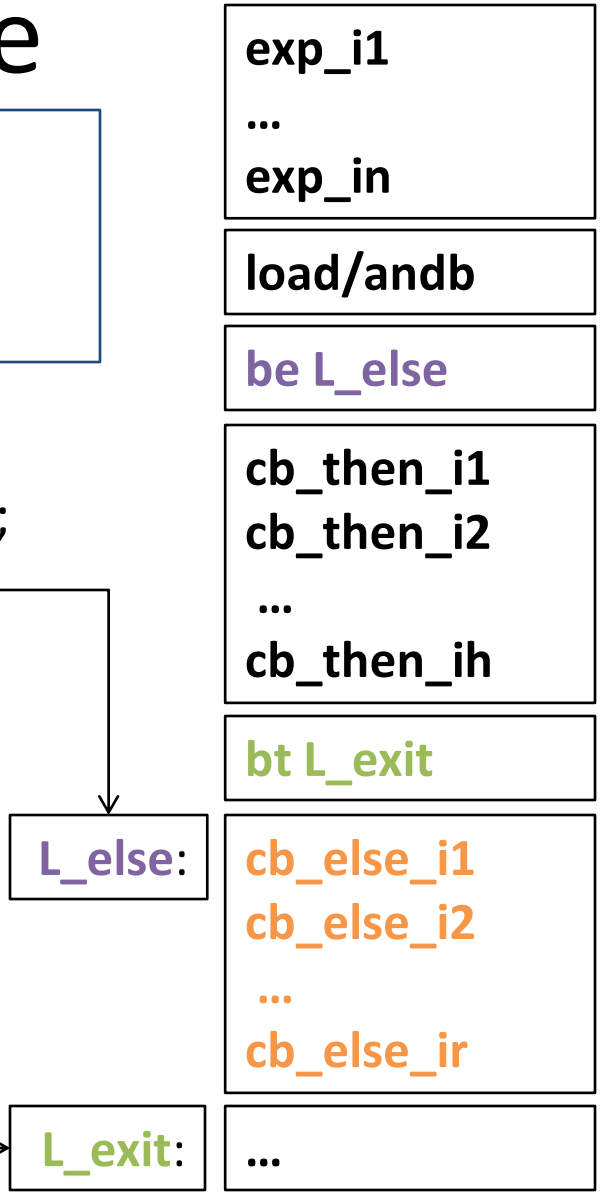
```
If (exp)
   code_block
else
   code_block
```



If-then-else

```
if_statement   : …
          | if_stmt ELSE
          {
            $2 = newLabel(program);
            gen_bt_instruction (program, $2, 0);
            assignLabel(program, $1);
          }
          code_block
          {
            assignLabel(program, $2);
          }
;
```

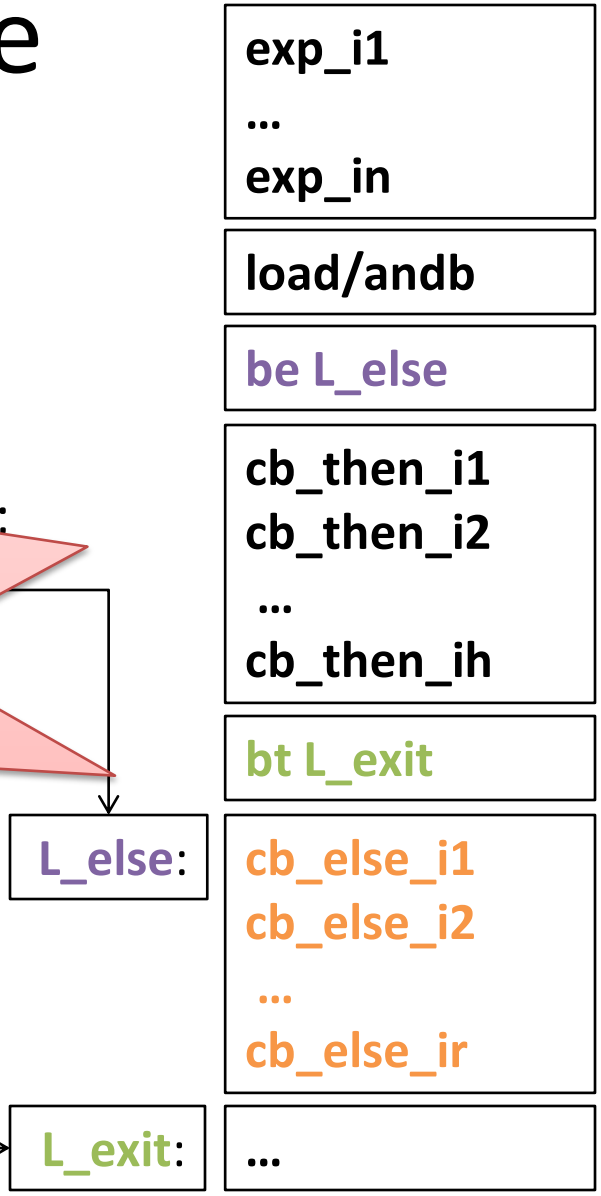
```
If (exp)
   code_block
else
   code_block
```



If-then-else

```
if_statement   : …
               | if_stmt ELSE
                 {
                   assi                    2);
                 }
;
```

Label L_else is not always attached to the first instruction after code_block (then)



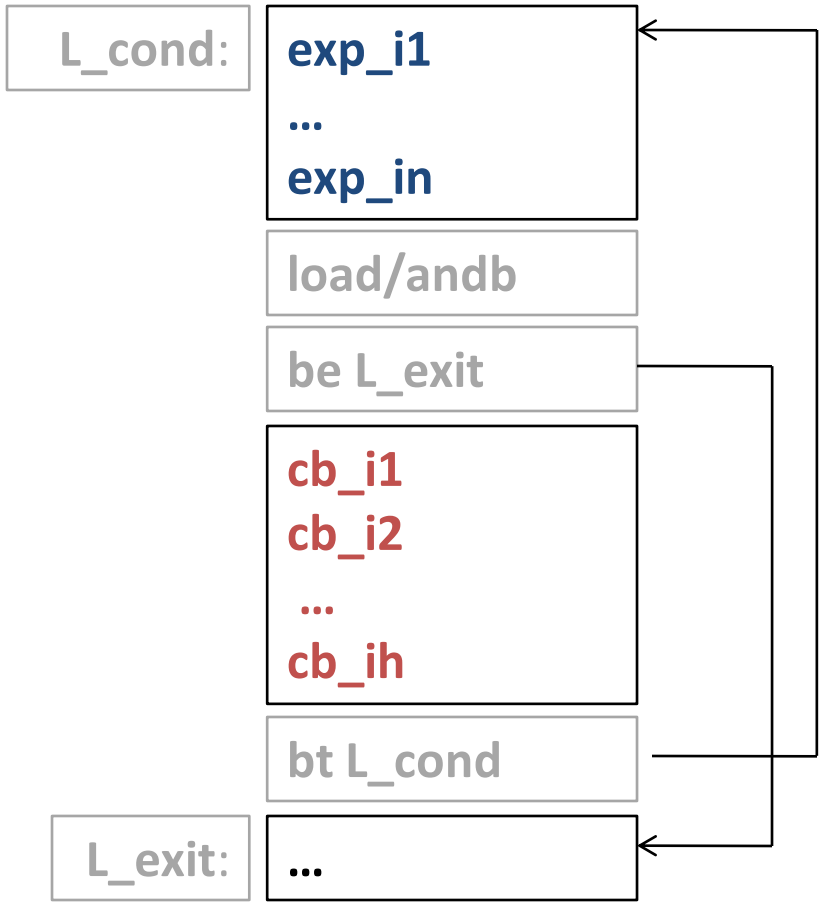
If-then-else data structure

```
%union {
   …
   t_axe_label *label;
}

%token <label> IF
%token <label> ELSE
```

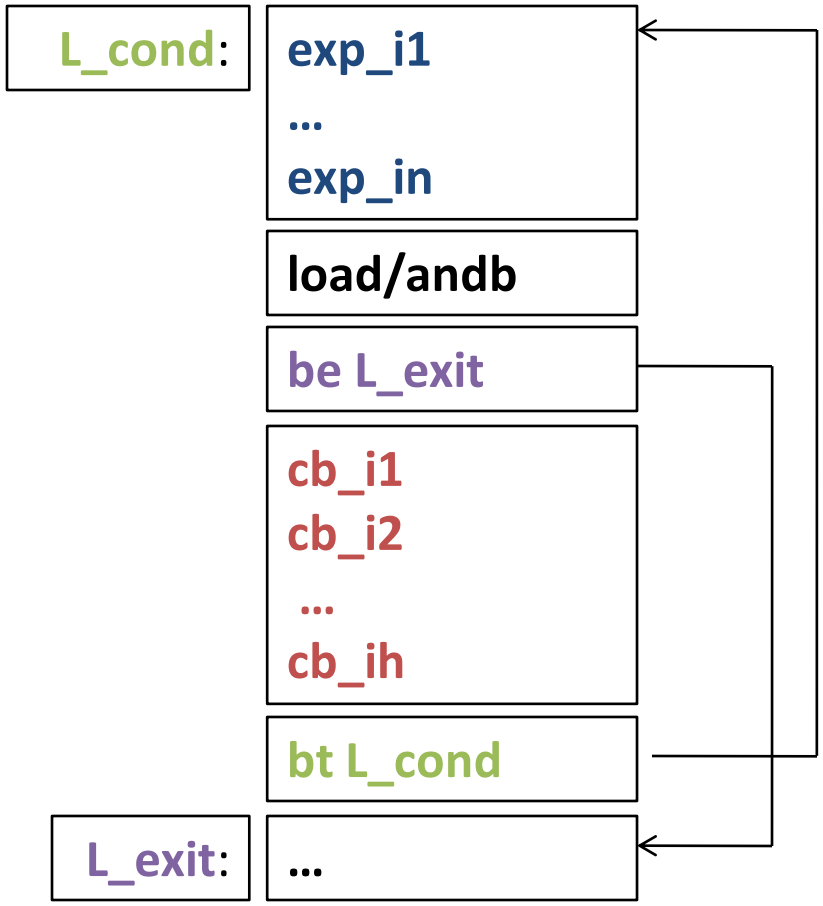
While

```
while (exp)
    code_block
```



While

```
while (exp)
    code_block
```

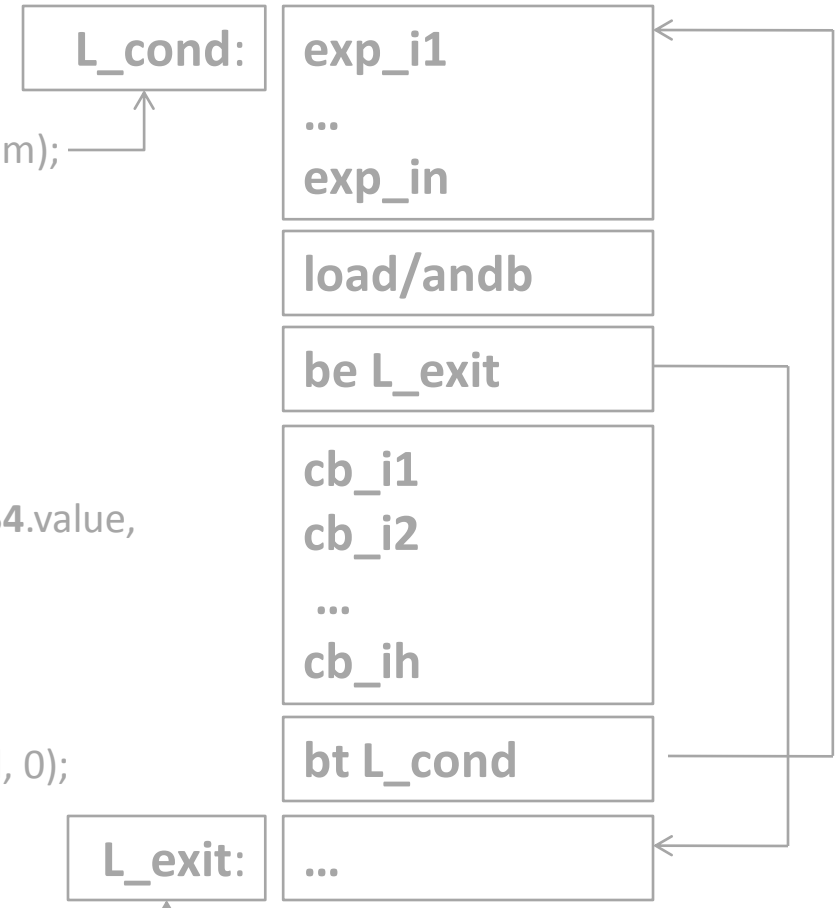


While

while (**exp**)
code_block

```
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);
                  };
```

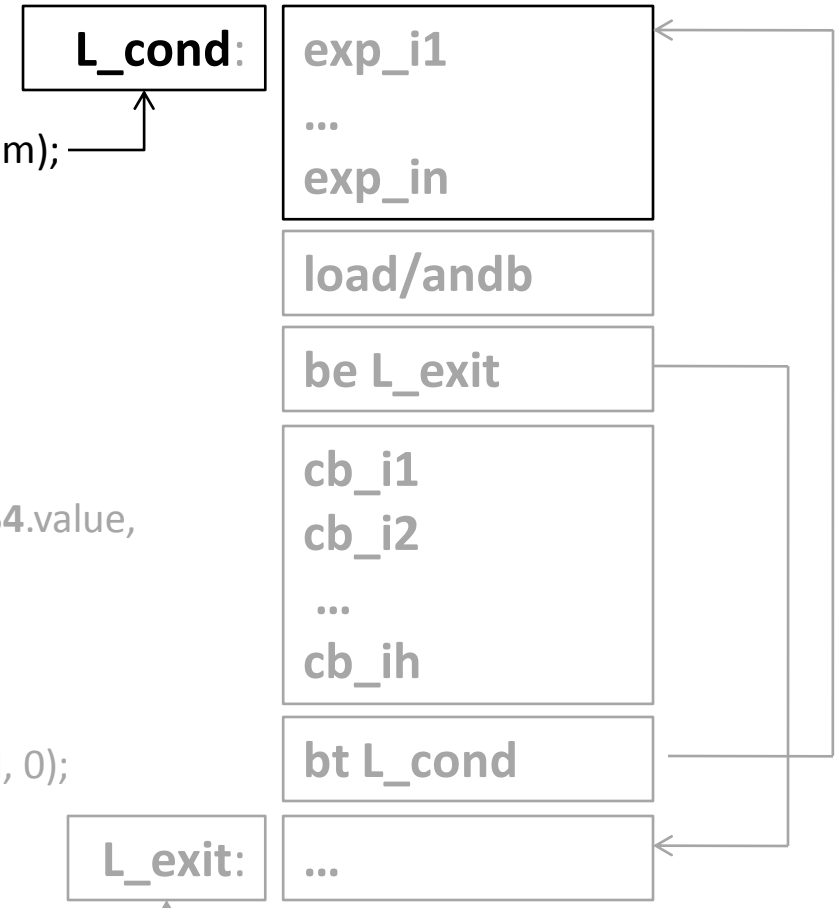


While

```
while (exp)
   code_block
```

```
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);
              };
```

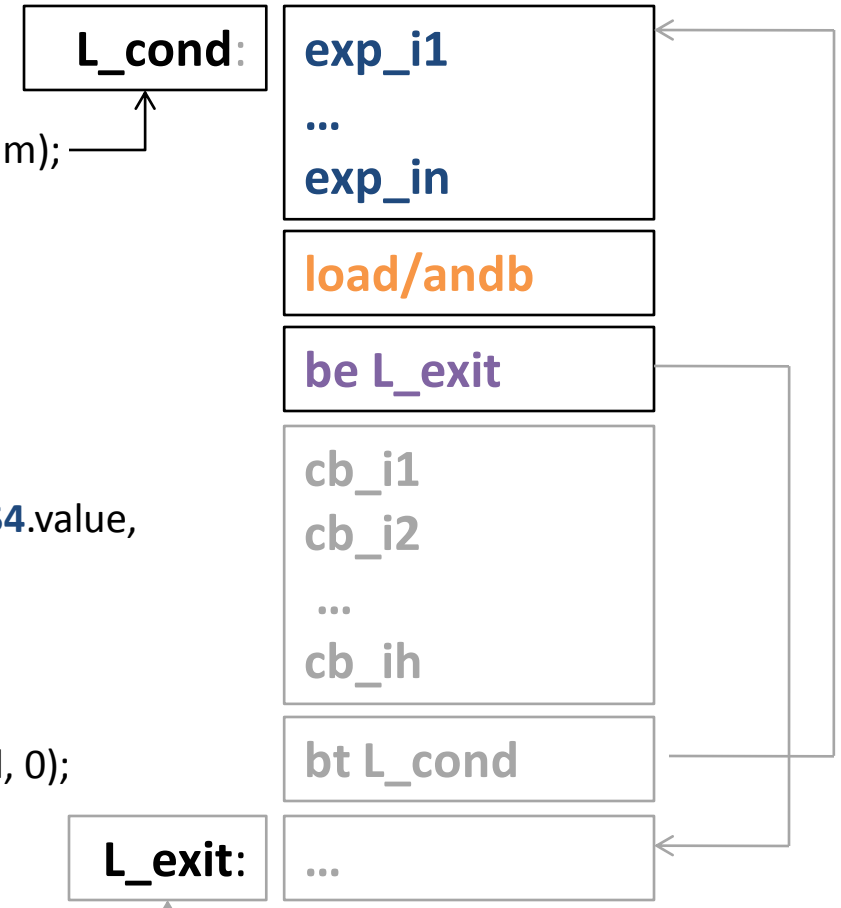


While

```
while (exp)
    code_block
```

```
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);
                  };
```

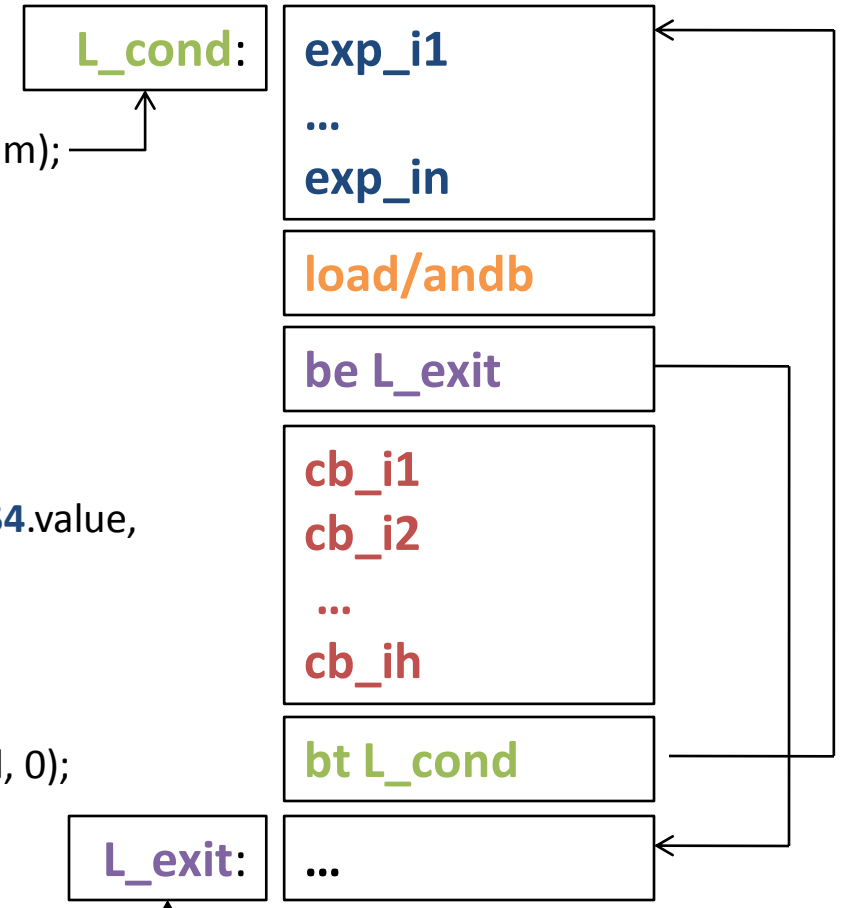


While

```
while (exp)
   code_block
```

```
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);
                  }
```



While data structure

```
%union {
  …
   t_while_statement while_stmt;
}

%token <while_stmt> WHILE
```

While data structure

```
typedef struct t_while_statement{
    t_axe_label *label_condition;
    t_axe_label *label_end;
} t_while_statement;
```

ACSE %union

```
%union {
   int intval;
   char *svalue;
   t_axe_expression expr;
   t_axe_declaration *decl;
   t_list *list;
   t_axe_label *label;
   t_while_statement while_stmt;
}
```

Non-terminals

%token <label> DO
%token <while_stmt> WHILE
%token <label> IF
%token <label> ELSE
%token <intval> TYPE
%token <svalue> IDENTIFIER
%token <intval> NUMBER

%type <expr> exp
%type <decl> declaration
%type <list> declaration_list
%type <label> if_stmt