POLITECNICO
MILANO 1863

Daniele Cattaneo
Andrea di Biagio
Giovanni Agosta

Advanced Compiler System for Education

Toolchain manual

Version 1.2.3
December 13, 2021

Contents

Introduction

ACSE (*Advanced Compiler System for Education*) is a simple toolchain developed for educational purposes for the course “Formal Languages and Compilers”. The toolchain aims to be a representative example of a complete computing system – albeit simplified – in order to illustrate what happens behind the scenes when a program is compiled and then executed.

The ACSE package is comprised of three tools, whose relationship is shown in fig. 1.

The first tool is the ACSE compiler, which accepts a program in a simplified C-like language called LANCE (*Language for Compiler Education*), and produces a compiled program in assembly language for a fictional architecture named MACE (*Machine for Advanced Compiler Education*).

The second tool is an assembler, which is able to take the assembly language code produced by ACSE and transform it to a binary executable *object file*.

The last tool is a simulator, named MACE just like the fictional architecture it implements. This simulator is able to read the object files produced by the assembler and interpret them, thus executing the program originally compiled by ACSE.

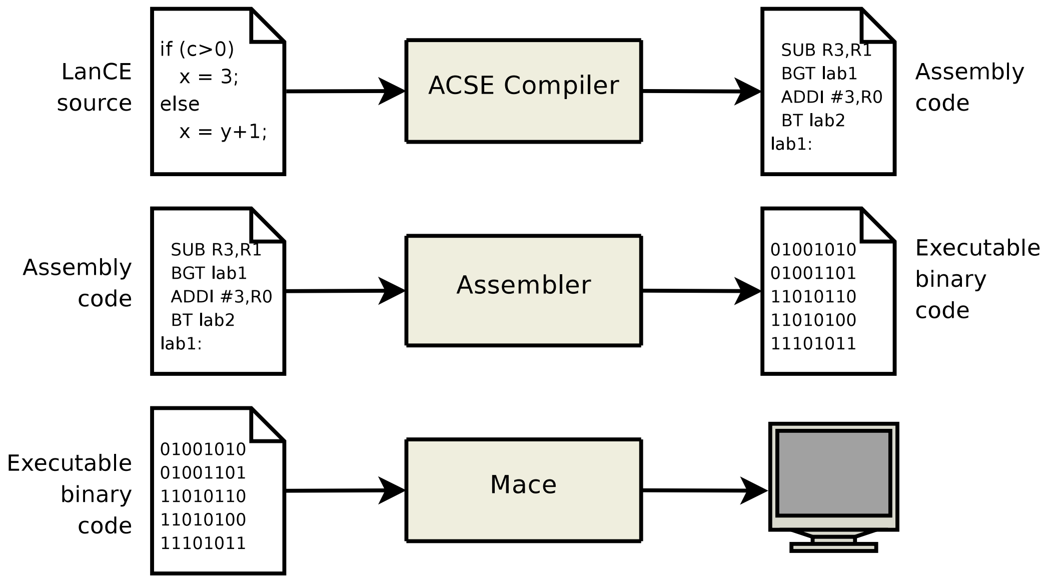


Figure 1.: Overview of the ACSE tools

This manual aims to describe these three tools in the ACSE package, together with the specifications of the MACE architecture and the LANCE language, in order to allow the interested student to deepen their knowledge of such tools and – by extension – of the process underlying the compilation and execution of a program in a modern computer architecture.

Organization of this manual

This manual is organized in three chapters and two appendixes.

First, in chapter 1 we describe the MACE architecture and its simulator, showing the layout of the *instruction set architecture* (ISA) and the execution model of the processor. Since the operation of the ACSE compiler and the assembler is strongly dependent on the MACE architecture, knowledge of MACE is a prerequisite for being able to understand the rest of the material.

Then, in chapter 2 we talk more in detail about the ACSE compiler: how it works in general, its internal architecture and *application programming interfaces*) (APIs), and how it is possible to add new constructs to the LANCE language.

Finally, in chapter 3 we briefly discuss the operation of the assembler.

The appendixes are devoted to more detailed information about the grammar of LANCE and the detailed specification of every instruction available in the MACE architecture.

Prerequisites

This manual specifically deals with the ACSE compiler toolchain, and does not aim to cover in sufficient amount of detail any other technologies which ACSE as a whole depends on.

Specifically, the reader is assumed to have at least a basic knowledge of the following topics: syntax and operation of *bison* and *flex*, theory of formal languages, theory of computer architectures.

In case a deeper understanding of these topics is desired, we encourage the following readings:

CHARLES DONNELLY, RICHARD STALLMAN. *Bison: The Yacc-compatible Parser Generator.* Free Software Foundation, 2021. `https://www.gnu.org/software/bison/manual/`

VERN PAXSON, WILL ESTES, JOHN MILLAWAY. *Lexical Analysis With Flex.* `https://westes.github.io/flex/manual/`

STEFANO CRESPI REGHIZZI, LUCA BREVEGLIERI, ANGELO MORZENTI. *Formal languages and compilation.* Springer, 2013.

JOHN L. HENNESSY, DAVID A. PATTERSON. *Computer architecture: a quantitative approach.* Elsevier, 2011.

While ACSE is certainly a representative example of a compiler, it does not completely illustrate the complexity that has evolved in the field of programming languages and compilers in the past 40 years. For example, we do not illustrate topics such as *Abstract Syntax Trees* (ASTs), intermediate representations, optimizations, linkers and libraries, operating systems interfaces, *Application Binary Interfaces* (ABIs), *Read-Eval-Print Loop* (REPL) user interaction models, *Just-In-Time* (JIT) compilation, just to name a few.

Therefore ACSE can only be but a starting point in the process of exploring compiler technologies. However, its simplicity is meant to be an encouragement to not be scared by the depth of the topic.

1. MACE

MACE (Machine for Advanced Compiler Education) is both a CPU architecture design, and a program that emulates that CPU architecture and executes object files (containing both machine code and data) produced by an assembler.

In this chapter we briefly introduce the internal architecture of the MACE machine and the execution steps performed by the simulator at run-time. The object file format specification and the symbolic assembler are discussed in the assembler documentation (chapter 3).

1.1. Architecture

Figure 1.1 shows the architectural design of the machine emulated by MACE.

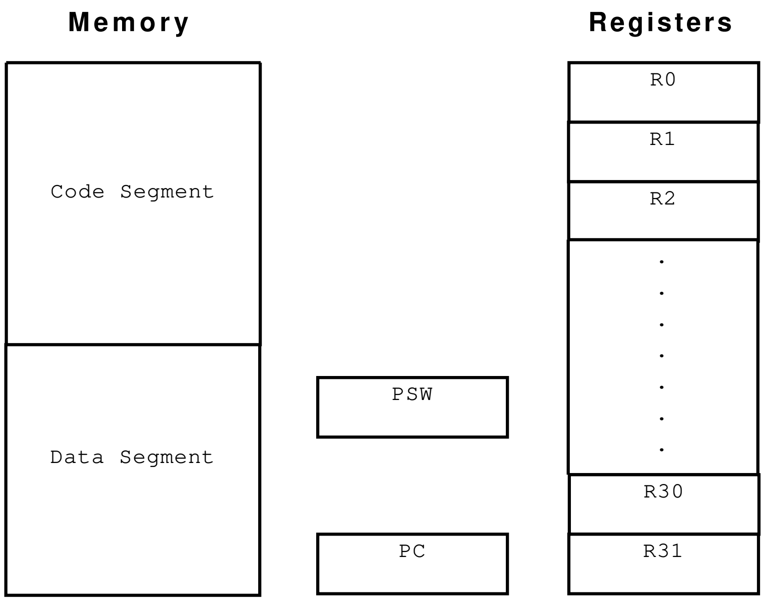


Figure 1.1.: Architectural Design of MACE

The MACE architecture has a word size of 32-bit, and 32-bit of addressing space for memory pointers. All operations – both in memory and in registers – operate on word-sized units. There is no provision for operating on smaller or larger data types (8-bit, 16-bit, 64-bit...). Unless otherwise specified, all words are assumed to represent *signed* integers in two's complement representation.

The architecture is composed by:

- One constant zero register (*R0*). This register is always set to 0, and even if an instruction tries to assign a value to *R0*, all the following instructions will always see a value of zero inside it.
- 31 general-purpose 32-bit registers (*R1* to *R31*).
- one 32-bit program counter register (*PC*);
- one 32-bit status register (*PSW*);
- a bank of 4096 32-bit words (16 KiB) of random access memory, addressed at word granularity.

1.1.1. Registers

Registers *R1* to *R31* are general purpose registers, and can be used for any purpose the programmer wishes. By convention, when a program uses the *JSR* and *RET* instructions, the *R31* register is used as the stack pointer. Note that ACSE does not use these instructions.

The *R0*, *PC* and *PSW* registers have a special purpose. *R0* simply allows easy access to the zero constant in ternary instructions.

The *PC* register contains the address of the instruction currently executing. The processor automatically increments its content after the execution of every instruction that is not a branch. On the contrary, branch instructions insert a new value in the *PC*.

The Status Register (*PSW*) is a 32-bit register used to handle conditional branches. Only its four least-significant bits are currently meaningful, all other bits are reserved for future usage. The majority of instructions that modify a register affect the content of the *PSW*, changing the four least-significant bits depending on the result generated by the instruction itself. Each of the four significant bits in the *PSW* is given a name, as shown in fig. 1.2.

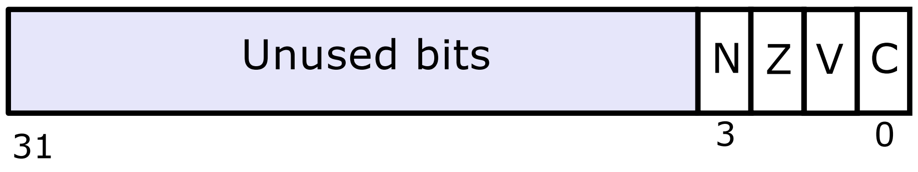


Figure 1.2.: Status Register

The bit ‘N’ (Negative) is set if the most significant bit of the result of an instruction (typically an arithmetic operation) is set to 1; otherwise it is cleared.

The bit ‘Z’ (Zero) is set if the result of an instruction (typically an arithmetic operation) is equal to zero; otherwise it is cleared.

The bit ‘V’ (Overflow) is set only if an arithmetic overflow occurs, implying that the result cannot be represented in the destination operand size.

The bit ‘C’ (Carry) is set if a carry out of the most significant bit of the result occurs for an addition, or if a borrow occurs in a subtraction.

1.1.2. Memory

The memory is organized as a sequence of 32-bit words. Only full 32-bit words can be addressed, and addresses contained in registers and instructions represent the number of words from the beginning of the memory. For example, address 1 represents the second 32-bit word in memory, i.e., from the fifth to the eighth byte. This is different from common CPUs, which address memory at byte boundaries.

MACE is a *Von Neumann* type architecture; in other words, instructions and data share the same addressing space – and therefore the same memory area. By convention, instructions (the *code segment*) take the area of memory that begins at address zero, and data (the *data segment*) is stored immediately after instructions end.

1.1.3. Instruction set

MACE is a RISC-like machine, and as such, instructions all have the same size of exactly one word (i.e., 32 bits). Instructions are divided in 4 categories, with 4 different types of encoding. Figure 1.3 illustrates the four categories and their general scheme of encoding.

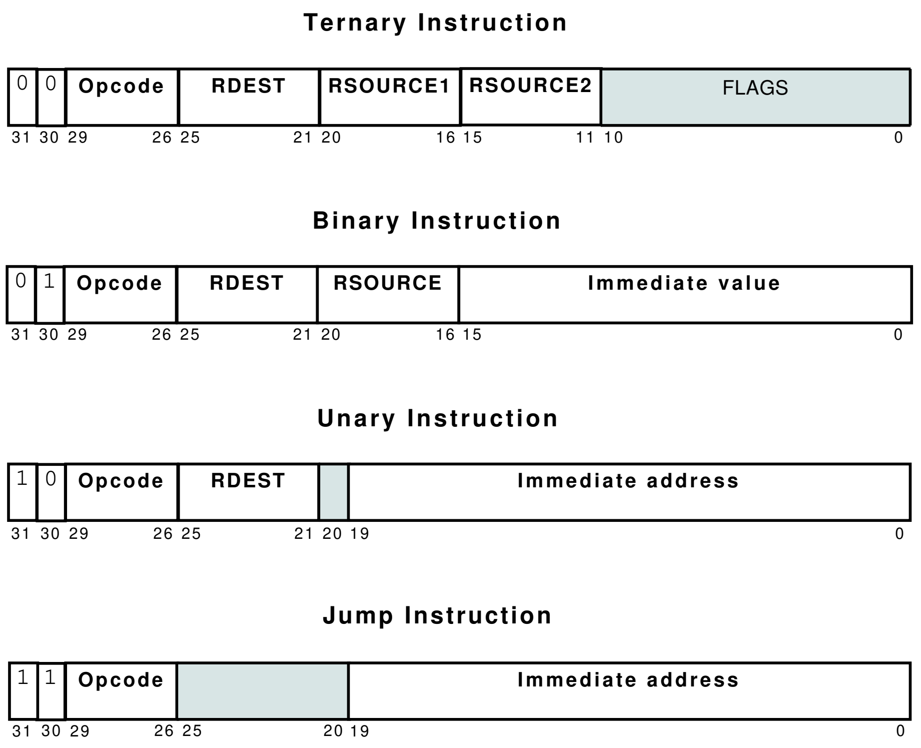


Figure 1.3.: Instruction Formats

The instruction format is specified by the two most significant bits of every instruction (bits 30 and 31). For example, as we can see in Figure 1.3 ternary instructions always have those bits set to '0'.

The encoding of each instruction contains fields to specify the function performed by the instruction (i.e. the *opcode* field) and the location of every operand.

Apart from its machine-readable binary encoding, an instruction also has a human-readable *symbolic form*, which is used in assembly language source code. The symbolic form of every instruction follows a common pattern:

⟨*name*⟩ [⟨*operand 1*⟩ [⟨*operand 2*⟩ [⟨*operand 3*⟩]]]

The *name* field represents both the instruction type (ternary, binary, unary, jump) and the opcode, and is chosen to shortly describe the operation performed by the instruction, such as ‘ADD’ for addition, ‘SUB’ for subtraction, *et cetera*. Another name for this field is *mnemonic*.

The subsequent operands can be either register references (such as *R1*, *R10...*), addresses (denoted by simple numbers) or immediates (numbers preceded by the # symbol), depending on the instruction format and the specific instruction. Not all instructions take all three operands: this is denoted by the square brackets in the pattern shown above.

A complete reference of all instructions available in MACE can be found in appendix B.

Instruction operands Most instructions take two (*ternary instructions*) or one (*binary instructions*) register operands, process them, and store the result in the destination register. Other instructions only have a single register operand that acts as both the destination and the source (*unary instructions*). *Jump instructions* do not have register operands.

Every register operand is encoded as a 5-bit value that represents the register identifier. For example, register *R3* will be encoded by the binary number corresponding to the number *3*: ‘00011’.

Binary instructions have a third *immediate operand*, that is, a **constant** value contained within the encoding of the instruction itself. Immediate operands are 16-bit values, and are implicitly sign-extended to 32-bit before use. In assembly language source code, immediate operands always appear prefixed by the ‘#’ symbol.

Unary instructions and jump instructions have an additional 20-bit *address operand*. In unary instructions, this operand is typically used as an absolute pointer to a memory location, or it goes unused entirely. In jump instructions, this operand is a signed integer offset to the address of the target instruction of the jump.

Direct and indirect addressing Registers can be addressed either *directly* or *indirectly*. *Direct addressing* means that the instruction operates on the value contained in the given register, while *indirect addressing* means that the instruction operates on the value contained at the memory address specified by the given register. In the symbolic assembly language syntax, indirect addressing is specified by enclosing the register name in parenthesis.

For example, the assembly instruction

```
ADD R3 R1 R2
```

represents the operation of summing the values contained in registers *R1* and *R2*, storing the result into register `R3`.

Conversely, the instruction

```
ADD (R3) R1 R2
```

still sums the values contained in registers *R1* and *R2*, but the result is stored into the *memory cell* whose address is contained in register *R3*. For example if register *R3* contains the value 100, the result of the sum will be stored into the 32-bit memory cell at address 100. Register *R3* itself is not modified in any way.

Unary and binary instructions always use *direct* addressing. Ternary instructions can use both direct and indirect addressing, but only for the destination and the second source register; the first source register is always directly addressed even for ternary instructions.

Flag bits In ternary instructions the information about direct or indirect addressing, alongside other modifiers, is encoded inside the *flag bits*: the 11 least-significant bits of an instruction word. The format of the flag bits is shown in Figure 1.4.

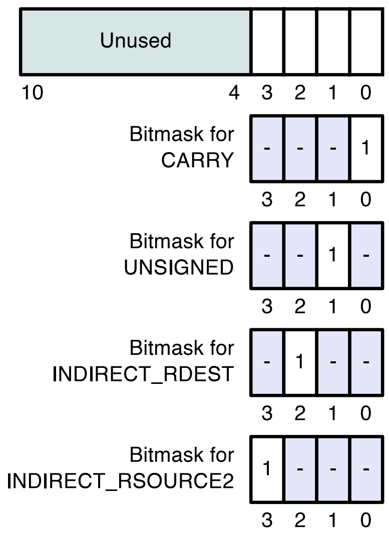


Figure 1.4.: Flag bits for ternary instructions

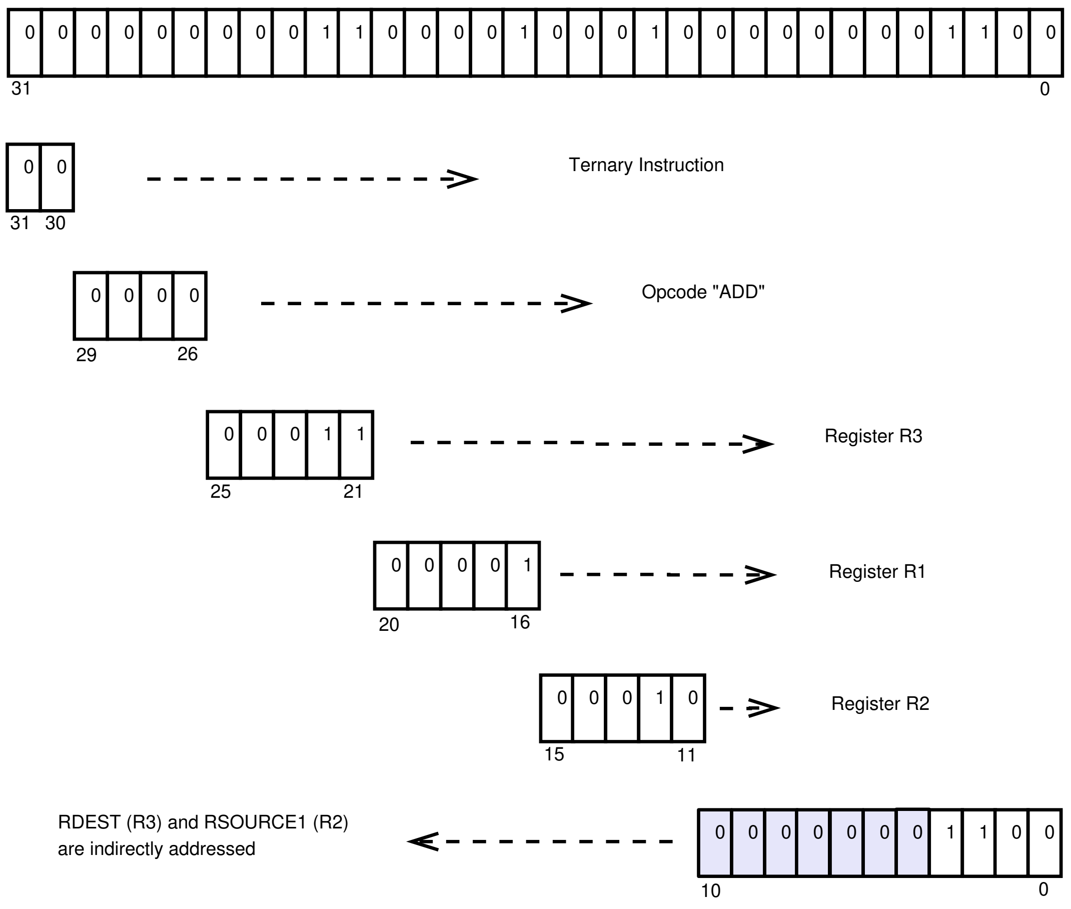
The bit ‘INDIRECT_RDEST’ is set to 1 if the ‘DEST’ register is indirectly addressed.

The bit ‘INDIRECT_RSOURCE2’ is set to 1 if the ‘RSOURCE2’ register is indirectly addressed.

If the bit ‘CARRY’ of an instruction is set to 1, some instructions will have special behavior if the ‘C’ flag of the PSW is set to 1. For *ADD*, *MUL*, *SHL*, and *SHR* instructions, the result of the operation is incremented by 1. For *SUB*, *DIV*, *NEG* instructions, the result of the operation is decremented by 1. This makes it possible to implement arithmetic on values larger than a word.

If the bit ‘UNSIGNED’ is set to 1, MACE treats the values stored in RSOURCE1 and RSOURCE2 as unsigned integers. This bit only affects the *MUL*, *DIV*, and *SHR* instructions; all other instructions do not change their behavior because of the properties of two’s complement representation.

Encoding example To conclude, the encoding of an example instruction is shown in fig. 1.5.



Decoded Instruction : ADD (R3) R1 (R2)

Figure 1.5.: An example of an encoded ADD instruction

1.2. How MACE works

The MACE execution model is simple. First, the internal state of the machine is initialized by a bootstrap procedure that works in the following manner:

1. Test if the object file given as input exists and is readable.
2. The content of every machine register is set to zero. Thus, `PC` will point to the first instruction in the code segment.
3. A block of memory of size 16 KiB is reserved which will contain both the code and data segments.
4. The machine code and data segments are loaded from the object file into memory.

Once the machine has completed the bootstrap procedure, the program is ready to be executed. The execution process works as follows:

- Repeat:
    1. Fetch the next instruction according to the value of `PC`

2. Decode the fetched instruction
3. Execute the instruction
4. Update (if necessary) the content of the register file
5. Update the value of the program counter (`PC`)
6. Update the value of the status register (`PSW`)

- Until a `HALT` instruction is encountered.

2. ACSE

ACSE is the simple compiler included in the ACSE toolchain package. ACSE is able to translate source code written in LANCE into assembly code for the MACE architecture (see the MACE documentation in Chapter 1).

In this chapter we will show the compilation process by introducing every step involved, describing in detail the data structures and processes that come into play. Finally, we give a brief introduction to the programming interface that allows the extension of ACSE to additional language constructs.

2.1. Introduction to ACSE

Figure 2.1 shows the pipeline of the compilation process implemented by ACSE. Every phase of the compilation chain takes as input the output of the previous phase. The input of ACSE itself is a source code file written in the LANCE language (see 2.1.1).

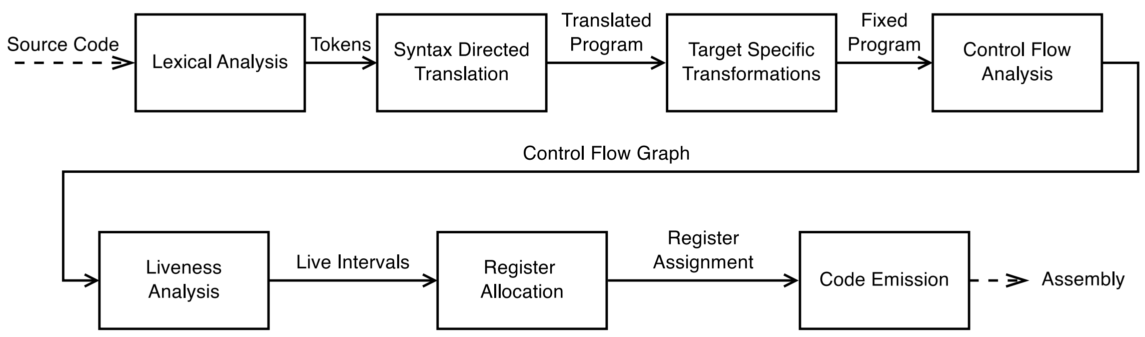


Figure 2.1.: Compilation process

In the first phase of the compilation process, the source file is analyzed and *tokenized* (i.e., subdivided into tokens) by the lexer. More specifically, the lexer component scans the input source code in search of specific string patterns. These patterns are described by regular expressions and each one identifies a certain lexical token (or lexeme).

The string of tokens is then processed by the parser in order to check and analyze the overall structure of the source program.

The lexer and parser used by ACSE are automatically generated by the *flex* and *bison* tools respectively. *Flex* and *bison* are general-purpose lexer and parser generators. *Flex* converts a map between regular expressions to token identifiers into a state-machine-based tokenizer. *Bison* is a general-purpose parser generator that converts an annotated context-free grammar into an LALR(1) deterministic bottom-up parser that recognizes it.

During the parsing process, the parser executes some *semantic actions* (if any) in correspondence of each recognized grammatical rule. Each semantic action directly transforms the tokenized input into equivalent assembly language instructions for the target machine (MACE). This process is called *Syntax-Directed Translation* (SDT). The current parser implementation provides support for both error tracking and notification of simple warning messages.

The lexical analysis, parsing and SDT passes form the *frontend* of ACSE. The *intermediate assembly code* produced by the frontend uses an unbounded number of registers.

Since the target machine has a limited set of registers, and a limit on the size of immediate parameters, additional passes are then performed. First, instruction with immediate arguments larger than 16-bit are replaced with equivalent instructions that satisfy this constraint. Second, liveness analysis and register allocation steps are performed. In the last phase of the compilation process, the assembly code is written in the output. These last steps form the *backend* of ACSE.

2.1.1. The LANCE language

LANCE is a small language with a syntax similar to C. It features scalar variables, one-dimensional arrays, expressions, statements for performing assignments and reading or writing from standard input, and basic control structures such as *if*, *while* and *do-while*.

The formal definition of the LANCE language is found in appendix A, in the form of *flex* rules for the lexer and a grammar in Backus-Naur form suitable for *bison*.

An example LANCE program that computes the factorial of an integer number and prints the result to standard output is shown in listing 2.1. If the number given as input is negative, the program writes '-1' to standard output and then exits.

2.2. Fundamental programming interfaces

In this section we describe the most basic programming interfaces implemented in ACSE which support the compilation task. We start by describing the most important data structure used in the compiler, the *linked list*, proceeding with an overview of how *bison* semantic actions work.

After these foundational topics, we go deeper in the design of ACSE, describing the data structure used to hold the current compilation state, and how to implement the generation of instructions and labels. Finally, we describe how ACSE handles expressions, variables and arrays.

```
int value, fact;         /* variable declarations */

read(value);             /* read from standard input the
                          * value of 'value' */
if (value < 0) {         /* invalid input */
   write(-1);
   return;
}

fact = 1;                /* initialize 'fact' */
while (value > 0) {      /* compute the factorial of value */
   fact = value * fact;
   value = value - 1;
}

write(fact);             /* write the result to standard output */
```

Listing 2.1.: An example of LANCE source code

2.2.1. The linked list data structure

To simplify the understandability of the code of ACSE, the only complex data structure used in it is the *double-linked list*. A common implementation of this data structure is found in the *collections.h* and *collections.c* files.

A double-linked list is a kind of *linked list* where each item of the list has two *links* or *pointers*: one to the *previous* element and one to the *next* element. Each element is a dynamically allocated instance of the structure *t_list*. The first element of the list has the *previous* pointer set to *NULL*; in the same way, the *next* pointer of the last element of the list is set to *NULL* as well. An empty list is represented by a single *NULL* pointer of type *t_list**. The definition of *t_list* is shown in listing 2.2.

```
typedef struct t_list {
   void          *data;
   struct t_list *next;
   struct t_list *prev;
} t_list;
```

Listing 2.2.: The definition of the *t_list* type.

Each element in the list has a *data* pointer, available for storing the data associated to that list element. The user of the list library is responsible for the memory management of such data pointers. Often, in API functions related to lists, the word *element* is used to refer to the *data* pointer. Instead, the word *link* is used to refer to an instance of *t_list*.

The *data* field can also be used to contain an integer number, by using the *INTDATA* macro to cast the integer into a *void**. Since there is no dynamic allocation involved in the operation of *INTDATA*, no memory management is required in this case.

Shorthand macros are available to fetch the next, previous and data pointers in a list element: *LNEXT*, *LPREV* and *LDATA*. If the data pointer is in fact an integer casted

```
typedef struct t_data_in_list {
   /* ... */
} t_data_in_list;

/* Create the list */
t_list *list = NULL;
/* Allocate an element */
t_data_in_list *element = malloc(sizeof(t_data_in_list));
/* Add the element at the beginning of the list */
list = addElement(list, element, 0);

/* ... */

/* Free all the elements in the list */
for (t_list *i = list; i; i = LNEXT(i)) {
   free(LDATA(i));
}
/* Free the t_list instances. After this call, the list pointer
 * is now invalid and must not be dereferenced anymore. */
freeList(list);
```

Listing 2.3.: Inserting elements in a list, and deallocating a list.

to a pointer by *INTDATA*, use *LINTDATA* instead of *LDATA* to retrieve the integer directly.

Creating a list Creating a list is simply a matter of declaring a pointer to a *t_list* initialized to *NULL*. This pointer is a valid empty list.

Adding elements The main primitive for adding and removing elements to a list is the *addElement()* function. This function takes, in order, a pointer to a list, the data element to insert, and the position where to insert the new element. The function returns a new pointer to the head of the list, which must replace the original pointer passed to the function. This pattern also holds for all other functions that modify a list.

It is possible to insert items at the end of the list by passing -1 as the position argument of *addElement()*.

Other utility functions are available for inserting elements in specific locations, such as *addSorted()*, *addLast()*, *addFirst()*, *addBefore()*, *addAfter()*.

Removing elements The two primitives for removing an elements from a list are *removeElement()* and *removeElementLink()*. *removeElementLink()* requires a pointer to a specific *t_list* node to remove, while *removeElement()* searches a specific data element in the list (through pointer comparison) to identify the node to be removed.

The utility function *removeFirst()* that removes the first element of the list is also available.

Finding and fetching elements The last set of functions available for working with lists allows to find elements or read them from the list without having to traverse it manually. The simplest of such functions are *findElement()* and *getPosition()*. *findElement()*

finds the list element with the specified data pointer. *getPosition()* returns the index of a certain element in the list.

Deallocating a list Since the list programming interface uses dynamic allocations, once a list is not needed anymore it must be deallocated. A list can be deallocated by using the *freeList()* function, but this function does not free the objects pointed to by the *data* pointer. Therefore, first you must remember to iterate through the list and free all the data elements manually.

A simplified example of how ACSE lists can be used is shown in listing 2.3. We refer to the header documentation of ACSE for more details about other list-related functions and advanced features of the list package.

2.2.2. Semantic actions

In a *bison* grammar, each grammar rule can have an associated action, called a *semantic action*, consisting of a C code block. After the parser tests the correctness of a statement found in the source program by applying a grammar rules, it executes the corresponding semantic action. More specifically, a semantic action is executed every time the non-terminal associated to it is recognized and reduced.

Bison also provides the possibility of defining *mid-rule* semantic actions. Such semantic actions are associated with only a part of a grammar derivation, and they execute once that part is recognized and reduced. Internally, mid-rule semantic actions are implemented by splitting the grammar rule into multiple sub-rules, one for each mid-rule action. An example is shown in listing 2.4.

```
do_while_statement:
   DO
   { /* action 1 */ }


   code_block WHILE LPAR exp RPAR
   { /* action 2 */ }
;
```

(a) *Bison* source with mid-rule actions.

```
$@1:
   %empty
   { /* action 1 */ }
;
do_while_statement:
   DO $@1 code_block WHILE LPAR exp RPAR
   { /* action 2 */ }
;
```

(b) Equivalent code without mid-rule actions.

Listing 2.4.: Underlying implementation of mid-rule semantic actions in *bison*.

The SDT pass of ACSE is implemented through the *bison* semantic actions. Each statement or construct is translated into one or more equivalent assembly instructions by following the procedures codified in each corresponding semantic action. All assembly instructions are stored inside a data structure called *t_program_infos*, which is defined in *axe_engine.h*.

2.2.3. The program data structure

An instance of *t_program_infos* contains all useful information about the program being compiled. Specifically, before starting with the lexical analysis and the syntactical analysis, the compiler initializes a global instance of *t_program_infos* called *program*. In order to generate the intermediate representation of the compiled program, the SDT pass will operate on that specific global instance.

The definition of *t_program_infos* is shown in listing 2.5, and it comprises various elements which describe the program as a whole.

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

Listing 2.5.: The definition of the *t_program_infos* type.

The most important element is the *instructions* list, which contains the list of all the instructions that form the code of the compiled program. In fact, the semantic actions that translate the program into various assembly instructions boil down simply to a programmed sequence of insertions inside this list. When the assembly code is generated, this list is traversed, and each instruction is printed to the output file.

The *variables* list, instead, is a list of all the variables declared in the program. For scalar variables (i.e., variables that are not arrays), the register associated with the variable is stored in the *symbol table* (*sy_table*). The *data* list contains a list of assembler directives used in the code emitter to produce the static allocations of the variables in the variable list.

Finally, the *label manager* (*lmanager*) stores the list of all labels placed in the program. Labels allow to create constant pointers to a memory location, in order to be able to refer to that location in the instructions of the compile program.

Other elements in the structure are auxiliary program-local variables used by various utility functions. Specifically, *instrInsPtrStack* is used by the target-specific transformation stage to allow the insertion of sequences of instructions in the middle of the *instructions* list. Finally, *current_register* is a field that contains the next register identifier that has not yet been used in the generated code of the program.

2.2.4. Generating instructions

The primary API for generating new instructions and placing them in the program is defined in *axe_gencode.h*. Each function in *axe_gencode.h* generates a different instruction, and the arguments of the function are used to generate the arguments of the instruction. Then, the instruction is automatically appended to the end of the list of instructions, and the function returns a pointer to the *t_axe_instruction* structure inserted in the list (see section 2.5.1).

Register identifiers When an instruction to generate takes a register parameter, the programmer passes a *register identifier* to the instruction generation function. A register identifier is simply an integer number that acts as a reference to that register.

In the assembly code, the register identifier is the number which appears following the *R* character. For example, in the instruction

```
ADD R3 R1 R2
```

the register identifier associated to register *R3* is simply the number *3*.

During the operation of the frontend stages (therefore also during the operation of the SDT) the number of registers is assumed to be infinite; the limit of 32 registers present in the MACE architecture is enforced at a later stage. Therefore, the SDT code does not need to worry about how many registers are used.

When the SDT generates new instructions, it often needs to ensure that the result of an operation is stored in a register that is not being used for some other purpose.

To simplify keeping track of registers, ACSE provides the *getNewRegister()* function. This function returns a different register identifier every time it is called.

Important! Register identifiers **do not have any relation with the value the register will assume once the program is executed**. They represent a **reference** to the register, but the register itself *does not exist* at *compile time* (i.e., during the compilation process). It only appears as soon as the program is loaded into the MACE simulator (or another compatible implementation of the architecture) and executed (i.e. at *runtime*).

It should be obvious that **it is not possible to access the value of something that does not exist yet**. The fact that register identifiers have the type *int* should not trick you into believing that that value is the content of the register.

There are four patterns of code generation functions, corresponding to the four types of instruction encoding.

Ternary instruction generators Functions that generate ternary instructions have the following form:

```
t_axe_instruction *gen_nnn_instruction(t_program_infos *program,
   int r_dest, int r_source1, int r_source2, int flags);
```

nnn is the lowercase name of the instruction to generate, for example ‘add’ for generating *ADD* instructions.

program is a pointer to the *t_program_infos* structure where to add the instruction – typically the global instance, itself named *program*.

r_dest, *r_source_1*, and *r_source_2* are the register identifiers of the operands of the instruction.

Finally, *flags* is an integer bitmask that specifies which registers are directly or indirectly addressed. This field can assume three values: *CG_DIRECT_ALL*, *CG_INDIRECT_ALL*, *CG_INDIRECT_DEST*, and *CG_INDIRECT_SOURCE*.

The value *CG_DIRECT_ALL* is used when both the destination register and the second source register are directly addressed.

The value *CG_INDIRECT_ALL* is used when both the destination register and the second source register are indirectly addressed.

The value *CG_INDIRECT_DEST* is used when only the destination register is indirectly addressed.

The value *CG_INDIRECT_SOURCE* is used when only the second source register is indirectly addressed. The addressing mode of the first source register is always direct.

Binary instruction generators Functions that generate binary instructions have the following form:

```
t_axe_instruction *gen_nnn_instruction(t_program_infos *program,
   int r_dest, int r_source1, int immediate);
```

Just like for ternary instructions, *nnn* is the lowercase name of the instruction to generate, *program* is a pointer to the *t_program_infos* structure, *r_dest* and *r_source_1* are the register identifiers of the operands.

immediate is the immediate argument of the instruction, which is not restricted to 16-bits here.

Unary instruction generators Functions that generate unary instructions can have several forms, depending on the meaning of address operand.

Address is a pointer (*LOAD*, *STORE*, *MOVA*):

```
t_axe_instruction *gen_nnn_instruction(t_program_infos *program,
   int r_dest, t_axe_label *label, int address);
```

Address is unused (*READ*, *WRITE*, *Scc*):

```
t_axe_instruction *gen_nnn_instruction(t_program_infos *program,
   int r_dest);
```

Just like for other instructions, *nnn* is the lowercase name of the instruction to generate, *program* is a pointer to the *t_program_infos* structure, *r_dest* is the identifier of the register operand.

Generation functions of instructions that also take an address parameter (*LOAD*, *STORE*, *MOVA*) have two additional arguments that are used for specifying that parameter. You can either specify a label, in the form of a pointer to a *t_axe_label* structure, or an absolute address by using the *address* parameter.

The *address* parameter is ignored unless the *label* parameter is set to *NULL*.

2.2.5. Generating labels

ACSE provides a facility for creating labels pointing to specific assembly instructions. After an instruction has been labeled, when generating branches, its label can be used to specify that the control flow will be transferred to that instruction when the branch is taken.

In an assembly code listing, such as the output of the compiler, labels look like the example below:

```
L1: ADD R3 R1 R2
```

In this example, the label 'L1' points to the address of that specific ADD instruction. At runtime, a branch to label 'L1' will transfer control to that ADD instruction, and execution will continue from there.

There are three primary functions used to create labels and make them point to an instruction, all defined in *axe_engine.h*: *newLabel()*, *assignLabel()* and *assignNewLabel()*. In the compiler, each label is represented with an instance of the structure *t_axe_label*.

The *newLabel()* function creates and returns a pointer reference to a new label, without making it actually point to any specific instruction. Basically, this function tells to the rest of the compiler that in the future a label will be generated, but this is not the moment to generate it yet: some other piece of code has to be produced first.

The *assignLabel()* function takes as input a pointer to *t_program_infos*, and a pointer to a *t_axe_label* previously created by *newLabel()*. It assigns the given label to the next assembly instruction that will be generated, therefore generating the label itself.

The *assignNewLabel()* function creates a label and makes it point to the next instruction immediately. It is equivalent to first calling *newLabel()* and then immediately calling *assignLabel()*.

Using two functions for allocating and placing labels is required because it is often needed to refer to a label before the label actually appears in the output of the compiler. An example in which these label generation functions are needed is shown in listing 2.6.

```
gen_sub_instruction(program, 3, 1, 2, CG_DIRECT_ALL);
t_axe_label *l1 = newLabel(program);
gen_bmi_instruction(program, l1, 0);
gen_add_instruction(program, 4, 4, 3, CG_DIRECT_ALL);
assignLabel(program, l1);
gen_addi_instruction(program, 4, 0, 1);
```

(a) Compiler code. Register identifiers are inline for illustrative purposes only.

```
     SUB R3 R1 R2

     BMI L1
     ADD R4 R4 R3

L1:  ADDI R4 R0 #1
```

(b) Generated code.

Listing 2.6.: Generating forward branches with labels.

2.2.6. Handling expressions

According to the *bison* grammar defined in *Acse.y*, an expression is defined by the non-terminal *exp* (see appendix A.2). To implement the nesting structure of expressions, the *exp* rule is recursive.

In its non recursive expansions, an expression can be defined either by the value of a variable, an array element, or a constant integer. Constant integer values correspond to the token 'INTEGER'.

Of course, an expression is constant when its value is computed by applying operators exclusively to constants. A common optimization in expression handling consists in computing the value of constant expressions at compile time rather than at runtime. This optimization is called *constant folding*.

As soon as a variable access appears in a sub-expression, the expression itself is no longer constant and it cannot be computed at compile time in its entirety: a sequence of instructions must be generated.

ACSE provides a framework for implementing constant folding of expressions, defined in the files *axe_structs.h* and *axe_expressions.h*.

In the SDT, an expression or a sub-expression is represented by a value of type *t_axe_expression*. *t_axe_expression* is a structure containing two fields:

expression_type	*REGISTER* if the value of the expression will not be constant, *IMMEDIATE* if it is constant.
value	If *expression_type* is *REGISTER*, a register identifier which will contain the value of the expression at **runtime**. Otherwise, this field contains the constant value of the expression.

The file *axe_expressions.h* defines two functions used to apply operators to expressions, generating code *only when strictly required*: *handle_bin_numeric_op()* and *handle_binary_comparison()*.

```
extern t_axe_expression handle_bin_numeric_op(
      t_program_infos *program,
      t_axe_expression exp1, t_axe_expression exp2,
      int binop);
extern t_axe_expression handle_binary_comparison(
      t_program_infos *program,
      t_axe_expression exp1, t_axe_expression exp2,
      int condition);
```

handle_bin_numeric_op() is used to handle numeric operators such as addition, subtraction, multiplication. *handle_binary_comparison()* is used to handle comparisons between two values. These functions take as input two expressions (the two operands of the binary comparison/operation) and return a new value of type *t_axe_expression* representing the new expression value.

Both functions take as inputs two expression values, *exp1* and *exp2*, which represent the operands to the operation being performed. Whether the function generates code or not, it depends on the *expression_type* of the two input expressions, as shown in table 2.1.

exp1 type	*exp2* type	Generates code?	Output exp. type
IMMEDIATE	IMMEDIATE	No	IMMEDIATE
IMMEDIATE	REGISTER	Yes	REGISTER
REGISTER	IMMEDIATE	Yes	REGISTER
REGISTER	REGISTER	Yes	REGISTER

Table 2.1.: Table of cases in which *handle_bin_numeric_op()* and *handle_binary_comparison()* can and cannot generate code depending on the input expressions.

The *binop* or *condition* arguments to these functions specify the operation which must be executed (in case of constant folding) or generated. An exhaustive list of possible operations is shown in table 2.2.

binop	Operation	*binop*	Operation
ADD	`+`	EORL	Logical XOR
ANDB	`&`	SUB	`-`
ANDL	`&&`	MUL	`*`
ORB	`|`	SHL	`<<`
ORL	`||`	SHR	`>>`
EORB	`^`	DIV	`/`

(a) *handle_bin_numeric_op()*

condition	Operation
_LT_	`<`
_GT_	`>`
_EQ_	`==`
NOTEQ	`!=`
LTEQ	`<=`
GTEQ	`>=`

(b) *handle_binary_comparison()*

Table 2.2.: Table of operation codes handled by *handle_bin_numeric_op()* and *handle_binary_comparison()*.

2.2.7. Handling variables and arrays

The LANCE language supports both scalar variables and one-dimensional arrays. Only a single data type is supported, signed 32-bit integers. While arrays and scalars are declared in similar ways, their implementation in the compiler is very different.

In ACSE, every scalar variable is associated to a register which contains the value of the variable. Assignments to a variable are implemented by generating an assignment to the variable's associated register. No memory accesses are required, even though variables are also given a memory location where their value is saved periodically (see section 2.6.4). The existence of these memory location is completely transparent to the SDT, but makes it possible to extend LANCE to support pointer operations.

Arrays are implemented as a contiguous fixed-size memory region, statically allocated in the data segment. The size of the memory region is the same as the length of the array. Variable-length and dynamically allocated arrays are not supported.

Creating variables The primary interface used in the SDT to create new variables is the *createVariable()* function, declared in *axe_engine.h*.

```
/* add a variable to the program */
extern void createVariable(t_program_infos *program, char *ID,
      int type, int isArray, int arraySize, int init_val);
```

The *ID* argument specifies the name of the variable which will be created. This name will also be used to uniquely identify the variable in other related APIs. The *isArray* argument must be set to '0' when creating a scalar variable, and to '1' when creating an array. If *isArray* is '1', then the *arraySize* argument specifies the number of elements in the array, otherwise it is unused. Note that zero-size arrays are illegal. If *isArray* is '0', then the *init_val* argument specifies the initial value of the scalar variable. In other words, this is the value assumed by the variable before its first assignment. For arrays, this parameter is unused.

The *type* argument, finally, specifies the type of the variable. However, since only integer types are supported, this argument will always be equal to *INTEGER_TYPE*, the identifier of the *int* type (see *axe_constants.h*).

For example, to create the following variable:

```
int var = 100;
```

the required invocation of *createVariable()* is:

```
createVariable(program, "var", INTEGER_TYPE, 0, 0, 100);
```

Retrieving properties of a variable You can retrieve the properties of a variable with the *getVariable()* function.

```
/* get a previously allocated variable */
extern t_axe_variable *getVariable(t_program_infos *program,
      char *ID);
```

Given a variable name *ID*, the *getVariable()* function returns the pointer to a *t_axe_variable* structure that contains the properties of the variable. The definition of this structure is shown in listing 2.7.

```
typedef struct t_axe_variable
{
   int type;                /* Always INTEGER_TYPE */
   int isArray;             /* Not zero if the variable is an array */
   int arraySize;           /* If 'isArray', the size of the array */
   int init_val;            /* If '!isArray', initial value of the var. */
   char *ID;                /* The name of the variable */
   t_axe_label *labelID;    /* A label that refers to the location
                             * of the variable inside the data segment */
} t_axe_variable;
```

Listing 2.7.: Definition of *t_axe_variable*.

For example, you can use the *t_axe_variable* structure to determine if a variable is an array or not, the size of arrays, and the initial value of scalars.

Retrieving the associated register of scalar variables Given a variable identifier as input, the *get_symbol_location()* function returns the register location where a variable is stored.

Once the register identifier has been retrieved, it can be used to generate code that reads or writes the variable simply by reading from the register or assigning a value to it.

Generating accesses to array elements The file *axe_array.h* provides a set of functions used to generate load and store instructions from/to array elements. The most important functions to do so are *loadArrayElement()* and *storeArrayElement()*.

```
extern int loadArrayElement(
      t_program_infos *program, char *ID, t_axe_expression index);
extern void storeArrayElement(t_program_infos *program, char *ID,
      t_axe_expression index, t_axe_expression data);
```

The *loadArrayElement()* function takes as input an array variable name, and an array subscript identifying an array element – represented as an expression member for convenience. It generates instructions that load the content of the given element of the array in a register, and it returns the identifier of the register that will hold the value of the specified array element at runtime.

On the contrary, *storeArrayElement* generates instructions that store a given value (again represented as an expression) in a specific element of an array. The value to store is specified by the *data* argument.

Important! For scalar variables, given a variable identifier, *get_symbol_location()* will always return the same register identifier, because that register identifier **is** in fact "the" variable. On the contrary, for *arrays*, the register returned by a call to *loadArrayElement()* is **not** "the" array element. In fact, array elements are not stored in registers, but in memory locations. As a result, **for a given identifier and array index, subsequent calls to *loadArrayElement()* will return different, newly allocated, registers.**

This is why **the assignment of a value to an array element cannot be generated as an assignment to the register returned by *loadArrayElement()***, you must use *storeArrayElement()*.

2.3. The Syntactic Directed Translator

The syntactic directed translator (SDT) of ACSE is automatically generated, alongside the parser, by running *bison* with the file *Acse.y* as input. It recognizes grammatical structures from a stream of tokens, and produces the equivalent code in MACE assembly language. You can see how each statement of an example program directly corresponds to the generated assembly code in table 2.3.

In this section we discuss the semantic values passed from the *flex*-generated lexer to the parser, and then we briefly show two examples of *bison* semantic actions. The first example describes how a typical expression semantic action works. The second example describes the semantic action implementing the `do-while` statement.

Source Program		Assembly	Comments
		`.DATA`	variables declarations
`int` `value,`	`L0:`	`.WORD 0`	initialize 4 bytes of data to 0
`fact;`	`L1:`	`.WORD 0`	initialize 4 bytes of data to 0
		`.TEXT`	start of a block of code
`read``(value);`		`READ R1 0`	read from standard input
`if` `(value < 0) {`		`SUBI R3 R1 #0`	sub immediate: R1 - 0
		`SLT R3 0`	set R3 on less than zero
		`BEQ L2`	'branch on equal' to label L2
`write``(-1);`		`ADDI R4 R0 #-1`	add immediate: R4 = -1
		`WRITE R4 0`	write R4 to standard output
`return``;`		`HALT`	stop the program execution
`} fact = 1;`	`L2:`	`ADDI R2 R0 #1`	set R2 to 1
`while``(value > 0) {`	`L3:`	`SUBI R5 R1 #0`	compare R1 with 0
		`SGT R5 0`	set R5 on 'greater than zero'
		`BEQ L4`	'branch on equal' to label L4
`fact = value * fact;`		`MUL R6 R1 R2`	mult.: R6 = R1 $\times$ R2
		`ADDI R2 R6 #0`	R2 = R6
`value = value - 1;`		`SUBI R7 R1 #1`	R7 = R1 - 1
		`ADDI R1 R7 #0`	R7 = R1
`}`		`BT L3`	'branch true' to label 'L3'
`write``(fact);`	`L4:`	`WRITE R2 0`	write R2 to standard output
		`HALT`	stop the program execution

(a) Source code (left) and generated MACE assembly code corresponding to each statement (right)

Variable Identifier	Type	Register	Label
value	INTEGER	R1	L0
fact	INTEGER	R2	L1

(b) Associated symbol table

Table 2.3.: Comparison between the statements in an example program and the output of the ACSE compiler.

2.3.1. Union declaration, operator precedences and associativity

Like any other *bison* parser, the ACSE SDT defines the list of tokens, the precedence of the tokens that appear in rules where a shift-reduce conflict may occur, and the union declaration that defines the possible types a semantic value can have.

The union declaration The union declaration used in ACSE is shown in listing 2.8. It features both types of general utility, and types specifically defined for specific syntactic features.

```
%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;
}
```

Listing 2.8.: The union declaration used by the ACSE SDT.

The *intval* and *svalue* elements are used for basic utility types: *int* and C string (*char* *) respectively.

The *list*, *expr* and *label* elements are ACSE-specific, and they respectively store a reference to a *t_list* (see section 2.2.1), an expression value (see section 2.2.6), and a reference to a label (see section 2.2.5).

Finally, the *decl* and *while_stmt* elements are specific to the implementations of the semantic actions for variable declarations and while/do-while statements, and are not used anywhere else.

Operator precedences and associativity ACSE defines the precedence and associativity of several tokens that appear in an expression, in order to establish their precedence. The piece of code defining these properties is shown in listing 2.9.

The precedence of the COMMA and ASSIGN tokens is required by the syntax of the *declaration_list* non-terminal. The rest of the tokens are defined to reflect the same operator precedence and associativity rules used in the C programming language.

A special mention is required for the NOT_OP token which is defined as right-associative even though the syntax does not allow it to be associated. Therefore, in this particular case, the *%right* associativity specification could be replaced by *%nonassoc* or *%precedence* without changing the syntax or the semantics of the language. More information on how to specify precedence and associativity will be discussed in section 2.4.5.

```
%left COMMA
%left ASSIGN
%left OROR
%left ANDAND
%left OR_OP
%left AND_OP
%left EQ NOTEQ
%left LT GT LTEQ GTEQ
%left SHL_OP SHR_OP
%left MINUS PLUS
%left MUL_OP DIV_OP
%right NOT_OP
```

Listing 2.9.: Operator precedence section in the ACSE SDT source code.

2.3.2. The ACSE lexer

The ACSE lexer is generated automatically by *flex* from the source code file in the *Acse.lex* file. The generated lexer handles the tokenization process of LANCE source code files, providing a function named *yylex()* which returns the identifier of the next token. Additionally, *yylex()* also modifies a global variable called *yylval*, which represents the semantic value of the token. The type of *yylval* is the union type declared by using the *%union* declaration shown in section 2.3.1.

While the lexer leaves the semantic value of most tokens uninitialized, the IDENTIFIER and NUMBER tokens are assigned a string value (*yylval.svalue*) and an integer value (*yylval.intval*) respectively, which match the substring corresponding to the token.

For example, an IDENTIFIER token which appears as the text “abcdef” in the source code, will have a string value of “abcdef”. A NUMBER token which appears as the text “1234” in the source code, will have an integer value of 1234.

Important! The string value (of type *char* *) of an IDENTIFIER token is *dynamically allocated* in the lexer, therefore it needs to be deallocated with the *free()* function once it is no longer referenced.

Typically, string values are no longer referenced at the end of the semantic action where they are used. However, like in any C program, if the pointer to the string is also copied to some other data structure with a longer lifetime, freeing the string too early will cause undefined behavior when the dangling pointer is accessed (*use-after-free* situation).

2.3.3. Example: expressions

Section 2.2.6 introduced what an expression is and which of the functions exposed by *axe_expressions.h* can be used in order to generate code for expressions.

In this section we will discuss the semantic actions associated with the following *bison* rule:

exp : *exp* AND_OP *exp*

According to the *flex* source file *Acse.lex*, the AND_OP token represents the LANCE 'bitwise and' operator (`&`). An example of a 'bitwise and' operation is the following: `a & b` (where `a` and `b` are variables declared in the source program). An example implementation of the semantic action is shown in listing 2.10.

```
exp: exp AND_OP exp
   {
      if ($1.expression_type == IMMEDIATE &&
            $3.expression_type == IMMEDIATE) {
         $$ = create_expression($1.value & $3.value, IMMEDIATE);
      } else {
         int r_dest, r_src1, r_src2;

         r_dest = getNewRegister(program);
         if ($1.expression_type == IMMEDIATE)
            r_src1 = gen_load_immediate(program, $1.value);
         else
            r_src1 = $1.value;
         if ($3.expression_type == IMMEDIATE)
            r_src2 = gen_load_immediate(program, $3.value);
         else
            r_src2 = $3.value;

         gen_andb_instruction(program,
               r_dest, r_src1, r_src2, CG_DIRECT_ALL);
         $$ = create_expression(r_dest, REGISTER);
      }
   };
```

Listing 2.10.: A semantic action that implements the AND_OP expression operator.

The two operators of the AND_OP operator correspond to the two *exp* non-terminals on the right and on the left of the token. In the semantic actions of the SDT, the semantic value of these two non-terminals in the grammar is available through the `$1` and `$3` symbols. Their type is *t_axe_expression*, as for all *exp* non-terminals. Therefore, the value of the two subexpressions could be either stored in a register, or it could be a constant (*IMMEDIATE*).

When the value of both members is a constant, the result is also a constant. To handle this case, we explicitly check the expression type of `$1` and `$3`, and in the case they are both equal to *IMMEDIATE* we perform the computation directly during the compilation process (at *compile-time*).

If at least one member is not constant, we must generate the *ANDB* instruction which, at runtime, will compute the value of the expression and store it into a new register (i.e., a register that has not been used elsewhere in the program). In order to retrieve the identifier of this register, we must first call the function *getNewRegister()* declared in *axe_engine.h*, which returns a new register identifier.

Since the *ANDB* instruction has two register operands, and no immediate operands, if one of the two members is a constant value we must also generate the code that temporarily stores that constant into a new register. This can be easily done with the *gen_load_immediate()* function.

At this point we can use the function *gen_andb_instruction()*, which is declared in *axe_gencode.h*, in order to generate an assembly *ANDB* instruction.

Finally, according to the *bison* semantic value of the *exp* non-terminal, the result must be stored in a *t_axe_expression* value. Thus, in our example we have to create a *t_axe_expression* by calling the function *create_expression()* defined in *axe_struct.h*.

In the actual implementation, the function *handle_bin_numeric_op()* is used to perform all the operations discussed so far, as shown in listing 2.11. For the 'bitwise and' operator, the operation identifier is *ANDB*.

```
exp: exp AND_OP exp
   {
      $$ = handle_bin_numeric_op(program, $1, $3, ANDB);
   };
```

Listing 2.11.: The semantic action associated to *AND_OP* expression operators. It is equivalent to the semantic action shown in listing 2.10.

2.3.4. Example: *do-while* statements

The *bison* semantic action for the *do-while* statement is shown in listing 2.12.

```
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);
                    };
```

Listing 2.12.: The semantic action associated to a *do-while* statement.

In order to implement a *do-while* statement, first we have to assign a label to the first instruction in the loop body. That label will be used as the target for a conditional jump instruction. This is done by the call to *assignLabel()* in the first semantic action.

After the label is generated, the body of the loop is parsed and compiled by the rules of the *code_block* non-terminal. At the end of the code block, the grammar expects the expression which represents the loop termination condition. If the value of the expression is different from zero (i.e., the loop condition is verified), the control should jump back to the first instruction of the loop body (defined by the non-terminal *code_block*); otherwise the control gets out from the loop.

The code which computes the expression is generated by the rules of the *exp* non-terminal. In case the value of the expression is a constant, it is *materialized* – in other words, copied into a register. Otherwise, to ensure the zero flag of the *PSW* is updated, an *ANDB* instruction is generated. Finally, we use *gen_bne_instruction()* to generate a conditional branch instruction which jumps to the first instruction of the loop body when the condition is false.

Note: There is actually an alternative way to handle the case of constant expressions. Simply, in case of 'IMMEDIATE' expressions, we generate a different branch altogether depending on the value. If the value is non-zero, a *BT* instruction is generated. Otherwise, if the value is zero, no branch at all is generated.

2.4. Extending the Syntactic Directed Translator

The SDT is the component of ACSE that must be modified in order to add support for additional language constructs of the LANCE language. In this section we illustrate several common tasks and the different programming patterns required to implement new constructs.

In general, when implementing new constructs, the programmer must do the following tasks:

1. Establish which new tokens shall be added to the lexer (*Acse.lex*)
2. Establish and implement the grammatical rules required to describe the syntax for the parser (*Acse.y*)
3. Understand how the semantics of the new construct can be implemented in terms of lower-level basic constructs and of assembly language instructions
4. Implement the code that generates the lower-level assembly instructions in the semantic actions in *Acse.y*.

The process of generating lower-level assembly language instructions corresponding to the higher-level semantics of a language construct is called *lowering*.

While the first and the second step tend to be trivial or otherwise fairly easy, understanding and implementing the correct lowering tends to be non-obvious and/or troublesome because of the nature of operation of an SDT. In this chapter we will focus on discussing the implementation of the lowering step, both in terms of specific issues related to syntactic directed translation, and the implementation of the most common control structures.

Note: In this section, we will often compare the MACE assembly code we want ACSE to generate with a pseudocode representation of its semantics, and the compiler semantic action code that generates it.

In the pseudocode representations, all variables that appear correspond to temporary registers. In the MACE assembly code, the notation `Rvar` indicates the register corresponding to the *var* variable appearing in the pseudocode. In the semantic actions, the `r_var` integer variable contains the register identifier corresponding to the *var* variable in the pseudocode.

2.4.1. Generating conditional branches

A very common pattern required by many constructs is the conditional branch. This is required any time a piece of code needs to be executed only in certain situations.

Usually, in high-level languages such as C, the condition is *positive*; in other words, the block of code is executed when the condition is true. In assembly language, this translates to a branch that *skips over* the block of code when the condition is *false*: in other words the condition must be *negative*.

Let us consider the case in which the loop condition is in this form:

```
if a ⟨operator⟩ b then
    code block
```

There are two ways to compute a binary condition for branching. One method consists in using the conditional tests built in the conditional branch itself. This requires the condition to be inverted manually. Alternatively, you can use the *handle_binary_comparison* function to generate code which will place the comparison result – zero or one – into a register. At that point, the conditional branch will only need to check if the comparison result is zero, and the condition does not need to be inverted.

Note: This discussion only applies if at least one of the members of the comparison is not a constant.

A comparison where both members are constant can be checked at compile time. At that point, no generation of branches is necessary: if the condition is false, the block of code is simply not generated, or skipped over with an unconditional branch if it is not possible to avoid generating it. For example, one case in which it is not usually possible or practical to avoid generating a block of code is when another semantic action is responsible for it.

Condition computation in the branch instruction In case we choose to implement the condition computation with the branch instruction itself, the code which must be generated follows a common pattern shown in listing 2.13.

In the listing, *r_a* and *r_b* are integer variables containing the identifiers of the registers holding the value of the two members of the comparison. Additionally, the fictional

```
gen_sub_instruction(program,                          SUB R0 Ra Rb
      REG_0, r_a, r_b, CG_DIRECT_ALL);
t_axe_label *l_skip = newLabel(program);
gen_bcc_instruction(program, l_skip, 0);              Bcc Lskip

/* code that generates 'code block' */                /* code block */

assignLabel(program, l_skip);                 Lskip:
```

(a) Compiler code. (b) Generated code.

Listing 2.13.: Pattern for generating a conditional branch which jumps over a block of code.

Instruction	Branch condition	Do-not-branch condition
BNE	$\neq$	$=$
BEQ	$=$	$\neq$
BGE	$\geq$	$<$
BLT	$<$	$\geq$
BGT	$>$	$\leq$
BLE	$\leq$	$>$

Table 2.4.: Branch instructions to use for all supported comparison operators.

Bcc instruction must be replaced by the branch instruction corresponding to the correct condition, according to table 2.4.

The *SUB* instruction is used to set the MACE processor's *PSW* register to the correct state for implementing the condition codes. In fact, the logical formulas used to compute the condition – shown in appendix B.1 – assume that the last instruction executed is specifically a *SUB* instruction. Specifying *R0* as the destination register causes the subtraction's result to be discarded. This is intended, because the actual result of the subtraction is not needed for the branch instruction. Only the *PSW* modification is needed, and it will happen regardless of whether the result of the subtraction is discarded or not.

If the second operand (the *right-hand side*) to the comparison is a constant, the *SUB* instruction can be replaced with *SUBI*. If the *left-hand side* operand is a constant, in order to replace *SUB* with *SUBI* it is necessary to swap the two operands as well, taking care to adjust the comparison operator.

When the comparison is specifically equality or inequality with zero ($a = 0$ or $a \neq 0$), instead of generating a *SUB* instruction you can generate the following instruction:

```
ANDB Ra Ra Ra
```

This instruction updates the Z flag in the *PSW* register according to the value in without modifying any other register. To branch when a is equal to zero, use the *BEQ* instruction, to branch when a is not zero, use the *BNE* instruction.

Condition computation with expressions The expression-handling machinery in *handle_binary_comparison()* can also be used to compute loop conditions.

The two arguments to the comparison must first be wrapped in two *t_axe_expression* structures, then they are passed to *handle_binary_comparison()* alongside with the constant corresponding to the desired operation (see table 2.2). *handle_binary_comparison()* will automatically generate the appropriate instruction for computing the comparison and placing the result in a register, updating the flags.

Note: When using *handle_binary_comparison()* for this purpose, be careful to always ensure that it is impossible for both members of the comparison to be constants (*IMMEDIATE* expressions), because in this situation no code is generated to update the *PSW* register.

After calling *handle_binary_comparison()*, if you need a branch taken when the condition is false, generate a *BEQ* instruction. Otherwise, if the branch must be taken when the condition is true, generate a *BNE* instruction.

Example Let us consider the case in which we need to generate the following conditional branch:

```
if a < 10 then
    code block
```

When exploiting built-in conditional tests available in branch instructions, the correct semantics are generated by the following compiler code:

```
gen_subi_instruction(program, REG_0, r_a, 10);
t_axe_label *l_skip = newLabel(program);
gen_bge_instruction(program, l_skip, 0);

/* code that generates 'code block' */

assignLabel(program, l_skip);
```

When using the expression-handling machinery, the same condition can be generated in this way:

```
handle_binary_comparison(program,
      create_expression(r_a, REGISTER),
      create_expression(10, IMMEDIATE),
      _LT_);
t_axe_label *l_skip = newLabel(program);
gen_beq_instruction(program, l_skip, 0);

/* code that generates 'code block' */

assignLabel(program, l_skip);
```

2.4.2. Generating loops

In a normal conditional branch, the direction of the branch is the same as the normal execution flow. Instead, when we have a branch whose direction is *backwards*, we do not have a conditional branch but a *loop*. A branch instruction is still present, but now it determines whether the loop must stop or not.

In many high-level languages, loops typically appear in two control structures named *while* and *do-while*, whose main difference is that in *while* loops the condition is at the beginning of the loop, while in *do-while* loops the condition is at the end. *For* loops can be implemented simply by adapting them to *while* loops.

```
while a ⟨operator⟩ b do
    [loop body]
```

```
do
    [loop body]
while a ⟨operator⟩ b
```

In assembly language, *while* loops consist of a check of the loop condition, and a branch that skips the entire loop if the condition is false. Then, the loop body's code follows, and finally an unconditional branch brings the execution flow back to the loop condition check. Instead, *do-while* loops start with the loop body itself. The check for the loop condition is at the end, and if the condition is true a conditional branch jumps back to the beginning of the loop.

Example of while loop Let us assume we want to generate the code that implements a loop that executes its body a certain number of times, as determined by the contents of a register. A loop counter register will contain the current number of iterations completed.

```
i ← 0
while i < n then
    [loop body]
    i ← i + 1
```

The compiler code generating this loop, alongside with the generated code itself, is shown in listing 2.14.

```
int r_i = gen_load_immediate(program, 0);
t_axe_label *l_next = assignNewLabel(program);
t_axe_label *l_break = newLabel(program);
gen_sub_instruction(program,
      REG_0, r_i, r_n, CG_DIRECT_ALL);
gen_bge_instruction(program, l_break, 0);

/* code that generates 'loop body' */

gen_bt_instruction(program, l_next, 0);
assignLabel(program, l_break);
```

(a) Compiler code.

```
        ADDI Ri R0 #0
Lnext:

        SUB R0 Ri Rn
        BGE Lbreak

        /* loop body */

        BT Lnext
Lbreak:
```

(b) Generated code.

Listing 2.14.: Generation of a simple *while* loop.

We use the *gen_load_immediate()* function to initialize a newly allocated register to zero, which will serve as the “counter variable” (in technical terms, the *induction variable*) of the loop. Using the techniques shown in section 2.4.1, we generate a check of whether the loop condition is false, which in that case jumps out of the loop. Finally, after the body of the loop, we generate a jump to the loop condition check again to start the next loop iteration.

Notice that this implementation properly handles the case in which the loop iteration count n is zero or negative.

Example of a do-while loop Another way to implement a loop that executes for a certain number of iterations is to make the loop counter decrease until it reaches zero. The induction variable contains how many loop iterations are left. Let us see an implementation of this kind of loop using a *do-while* construct.

```
i ← n
do
    loop body
    i ← i - 1
while i > 0
```

The code that generates this loop is shown in listing 2.15.

```
int r_i = getNewRegister(program);
gen_add_instruction(program,
      r_i, r_n, REG_0, CG_DIRECT_ALL);
t_axe_label *l_next = assignNewLabel(program);

/* code that generates 'loop body' */

gen_subi_instruction(program, r_i, r_i, 1);
gen_subi_instruction(program, REG_0, r_i, 0);
gen_bgt_instruction(program, l_next, 0);
```

(a) Compiler code.

```
       ADD Ri, Rn, R0
Lnext:

       /* loop body */

       SUBI Ri Ri #1
       SUBI R0 Ri #0
       BGT Lnext
```

(b) Generated code.

Listing 2.15.: Generation of a simple *do-while* loop.

First, we retrieve a register identifier to use for the induction variable, then we generate an *ADD* instruction which copies the number of iterations to perform to that register. Then, the body of the loop is generated, and after that, the check of the loop condition.

A feature common to all *do-while* loops is that the first iteration is always performed at least once. As a result, this loop will execute exactly once when n is zero. Therefore, *while* loops are more general, and in doubt they should be preferred.

Finally, notice that the code generated in the example can be optimized. Specifically, the last *SUBI* instruction can be removed from the generated code. This optimization reuses the modifications to *PSW* performed by the first *SUBI* instruction to also compute the loop condition. This greater ability for optimization is the main advantage of *do-while* loop: less code must be generated and executed for the loop to function.

Loop unrolling All the examples we have seen generate the code which implements a loop, which will execute at runtime. The compiler itself does not perform any loop. However, if the number of iterations the loop must perform is known at compile time, instead of generating a loop, the same code can be generated multiple times instead. This technique is called *loop unrolling* and it may achieve a small performance advantage for very small iteration counts, because it removes the code that checks for the loop condition.

Loop unrolling is **strongly discouraged** because a proper implementation must have a maximum limit on the number of unrolled iterations. Above that limit, a standard run-time loop should be generated instead, to avoid excessive code size. An example of a simple implementation of loop unrolling *without* this limit check is shown in listing 2.16.

```
for (int i=0; i<5; i++) {
   /* Code that generates 'loop body'.
    * An example loop body follows. */
   gen_add_instruction(program,
         r_a, r_a, r_b, CG_DIRECT_ALL);
}
```

(a) Compiler code.

```
ADD Ra Ra Rb
ADD Ra Ra Rb
ADD Ra Ra Rb
ADD Ra Ra Rb
ADD Ra Ra Rb
```

(b) Generated code.

Listing 2.16.: Generation of a simple unrolled loop.

2.4.3. Sharing state between semantic actions

In a *bison* parser, each semantic action is a separate independent C language scope. As a result, local variables defined in a semantic action cannot be accessed in any other semantic action, even if it belongs to the same grammatical rule. However, it is often necessary to share variable between semantic actions, especially for constructs where the same grammar rule contains multiple actions.

Global variables When the syntax does not allow nesting, or when it is not required to support nesting, the simplest approach for sharing variables between semantic actions is simply to introduce one or more global variables. This is not sufficient for nestable syntax because the semantic actions of inner constructs would overwrite the global variable.

You can see an example of this approach in listing 2.17a, where it is used to implement the *while* statement. Notice how, when the statement is nested, the *while_data* variable would be overwritten during the parsing of the code_block non-terminal, and the last semantic value would access the labels associated with the incorrect statement.

For handling nestable grammar constructs, different approaches are therefore needed.

Semantic value abuse One approach for sharing variables which is safe when rules can be nested, is to assign a semantic value to a token in the rule which would not have one otherwise. This allows all semantic actions in the same rule to access the same variable simply by accessing the semantic value of that token.

The main limitation of this method is that it does not work when the variable must also be shared across multiple rules, and not just in the same rule. Additionally, only the tokens that appear *before* the first semantic action requiring access to the variable are eligible.

This technique is shown in listing 2.17b.

Global stacks Another approach to sharing variables across semantic actions and rules is using a global stack.

The global stack is simply a global linked list (see section 2.2.1) whose first element represents the top of the stack. The stack contains pointers to the variables used by each grammar rule in reverse nesting order, such that the top of the stack will always contain the variable used by the rule currently being parsed. At the beginning of the first semantic action in the rule, a new variable is allocated and pushed onto the top of the stack; the last rule will need to free the pointer, and then remove it from the stack. Other semantic actions can access the variable by accessing the first element of the global list.

This last choice, the most complicated one but the most general, is shown in listing 2.17c.

2.4.4. List syntax and recursive rules

In a typical programming language, the majority of constructs are fixed-form, but others are variable and can contain multiple repeating structures. The most common example is expressions, which are a recursive tree of operators represented in linear form: an operator can be repeated multiple times, applied to different subexpressions.

Another case is represented by lists, not in the sense of a data structure, but as repeated instances of a certain grammatical structure. In the C language, an example of a list appears when specifying the arguments to a function. Another example, which is also found in the basic version of the LANCE language, is the declaration list, which allows multiple variables to be declared on the same line.

The only way to represent this kind of syntax is to use *recursive grammar rules*, that is, rules which are include themselves in their definition. Let us look at an example from ACSE itself: the *bison* syntax for the grammar rules implementing definition lists.

```
declaration_list : declaration_list COMMA declaration
                 | declaration;
     declaration : IDENTIFIER ASSIGN NUMBER
                 | IDENTIFIER LSQUARE NUMBER RSQUARE
                 | IDENTIFIER;
```

axe_struct.h

```
typedef struct t_while_stmt {
   t_axe_label *l_cond;
   t_axe_label *l_end;
} t_while_stmt;
```

Acse.y

```
t_while_stmt while_data;


while_statement:
  WHILE
  {



    while_data.l_cond = 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);
    while_data.l_end = newLabel(program);
    gen_beq_instruction(program, $1.l_end, 0);
  }
  code_block
  {

    gen_bt_instruction(program,
      while_data.l_cond, 0);
    assignLabel(program, while_data.l_end);


  };
```

(a) Implementation with a global variable

axe_struct.h

```
typedef struct t_while_stmt {
   t_axe_label *l_cond;
   t_axe_label *l_end;
} t_while_stmt;
```

Acse.y

```
%union {
   t_while_stmt while_stmt;
}
%token <while_stmt> WHILE

while_statement:
  WHILE
  {



    $1.l_cond = 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.l_end = newLabel(program);
    gen_beq_instruction(program, $1.l_end, 0);
  }
  code_block
  {

    gen_bt_instruction(program, $1.l_cond, 0);

    assignLabel(program, $1.l_end);


  };
```

(b) Implementation with a shared semantic value

axe_struct.h

```
typedef struct t_while_stmt {
   t_axe_label *l_cond;
   t_axe_label *l_end;
} t_while_stmt;
```

Acse.y

```
t_list *while_stk = NULL;


while_statement:
  WHILE
  {
    t_while_stmt *self =
      malloc(sizeof(t_while_stmt));
    while_stk = addFirst(while_stk, self);
    self->l_cond = assignNewLabel(program);
  }
  LPAR exp RPAR
  {
    t_while_stmt *self = LDATA(while_stk);
    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);
    self->l_end = newLabel(program);
    gen_beq_instruction(program, self->l_end, 0);
  }
  code_block
  {
    t_while_stmt *self = LDATA(while_stk);
    gen_bt_instruction(program, self->l_cond, 0);

    assignLabel(program, self->l_end);
    while_stk = removeFirst(while_stk);
    free(self);
  };
```

(c) Implementation with a global stack

Listing 2.17.: Various implementation of the *while* statement, using different programming patterns to share variables across semantic actions.

Like recursive procedures or functions, recursive grammar rules have both a base case and a recursive case, in the form of two different expansions of the rule. In our example, *declaration_list* can be expanded either as

> *declaration_list* : *declaration_list* COMMA *declaration*

which constitutes the *recursive* case, or as

> *declaration_list* : *declaration*

which is the *base* case.

Left and right recursion In the recursive case just shown, the recursion happens right at the beginning of the rule: this is called *left recursion.* On the contrary, *right* recursion occurs where the recursion happens at the end of the rule:

```
declaration_list : declaration COMMA declaration_list
                 | declaration;
```

Left recursion and right recursion are equivalent, however some classes of parsers can only handle grammars with one or the other kind of recursion. For instance, LL(1) parsers cannot handle left recursion.

The LALR(1) parser included in *bison* can handle both left and right recursion. However, left recursion is preferred, because right recursion defers the reduction of all the recursive rules at the last possible opportunity, and therefore requires additional stack space in proportion to the number of elements in the list. Instead, with left recursion all recursive rules are reduced as soon as possible, and do not take additional space on the stack.

Empty lists In the example shown previously, the base case of *declaration_list* expands to *declaration*, thereby preventing a situation in which the entire *declaration_list* non-terminal expands to no token at all. In other words, having a non-empty base case in a recursive grammar prevents empty lists. Instead, if empty lists should be allowed in the syntax, the recursion base case shall simply match the empty string.

In the *bison* syntax for grammars, empty cases are simply left empty, there is no need to introduce an explicit token for empty strings:

```
declaration_list : declaration_list COMMA declaration
                 | ;
```

In alternative, you can use the empty string special token `%empty`.

```
declaration_list : declaration_list COMMA declaration
                 | %empty ;
```

```
decl_list : decl_list COMMA decl              /* Reduction 1 */
            { /* ''list recursive'' action */ }
          | decl                              /* Reduction 2 */
            { /* ''list base'' action */ }
          ;
     decl : DECL                              /* Reduction 3 */
            { /* ''declaration'' action */ }
          ;
```

Input, lookahead	Action	LALR Stack	Action Executed
DECL, DECL, DECL	Shift DECL	DECL	–
, DECL, DECL	Reduce (3)	*decl*	⟨declaration⟩
, DECL, DECL	Reduce (2)	*decl_list*	⟨list base⟩
, DECL, DECL	Shift COMMA	*decl_list* COMMA	–
DECL, DECL	Shift DECL	*decl_list* COMMA DECL	–
, DECL	Reduce (3)	*decl_list* COMMA *decl*	⟨declaration⟩
, DECL	Reduce (1)	*decl_list*	⟨list recursive⟩
, DECL	Shift COMMA	*decl_list* COMMA	–
DECL	Shift DECL	*decl_list* COMMA DECL	–
	Reduce (3)	*decl_list* COMMA *decl*	⟨declaration⟩
	Reduce (1)	*decl_list*	⟨list recursive⟩

(a) Left recursion

```
decl_list : decl COMMA decl_list              /* Reduction 1 */
            { /* ''list recursive'' action */ }
          | decl                              /* Reduction 2 */
            { /* ''list base'' action */ }
          ;
     decl : DECL                              /* Reduction 3 */
            { /* ''declaration'' action */ }
          ;
```

Input, lookahead	Action	LALR Stack	Action Executed
DECL, DECL, DECL	Shift DECL	DECL	–
, DECL, DECL	Reduce (3)	*decl*	⟨declaration⟩
, DECL, DECL	Shift COMMA	*decl* COMMA	–
DECL, DECL	Shift DECL	*decl* COMMA DECL	–
, DECL	Reduce (3)	*decl* COMMA *decl*	⟨declaration⟩
, DECL	Shift COMMA	*decl* COMMA *decl* COMMA	–
DECL	Shift DECL	*decl* COMMA *decl* COMMA DECL	–
	Reduce (3)	*decl* COMMA *decl* COMMA *decl*	⟨declaration⟩
	Reduce (2)	*decl* COMMA *decl* COMMA *decl_list*	⟨list base⟩
	Reduce (1)	*decl* COMMA *decl_list*	⟨list recursive⟩
	Reduce (1)	*decl_list*	⟨list recursive⟩

(b) Right recursion

Figure 2.2.: Comparison of the parsing process performed by a *bison*-generated LALR parser for left-recursive and right-recursive grammars.

Order of evaluation and semantic actions In general, *bison*-generated parsers execute semantic actions at the end of a rule when the rule is reduced. As a result, the order of execution of semantic actions embedded in a recursive grammar depends on the order in which the parser decides to reduce each recursive rule. This order is determined by whether we have left recursion or right recursion, and is illustrated by fig. 2.2.

It is evident that left recursion has a further advantage over right recursion. With left recursion, the semantic action associated to the base case of recursion is executed when the first element of the list is found. The semantic actions of the recursive case are executed every time another element is found and parsed, in **left-to-right** order. In other words, the order of execution reflects the natural order in which we expect to read the list.

On the contrary, with right recursion the actions of the recursive rule are executed as if the input was read in backwards order, right-to-left. The base case action executes first but will be associated with the parsing of the last element of the list rather than the first, and then the recursive case action executes for the second-last item, the third-last item, and so on, in backwards order. This discrepancy in the order of parsing only extends to the recursively defined non-terminal. The list elements themselves, if they are associated with a separate grammar rule, are still parsed and reduced in the order they appear.

Note: In summary, always use left recursion!

2.4.5. Operator precedence and associativity

While strictly left-recursive and right-recursive grammar rules are inherently non-ambiguous, other kinds of recursion might be. The classic example is infix-notation[1] expressions: without establishing priority rules, expressions without parenthesis are ambiguous.

When using an LR-class parser, the ambiguities in infix expressions become *shift-reduce conflicts* which happen when a second operator is encountered in succession. If the parser chooses to *shift*, the parsing process continues, and the operator on the *left* is prioritized. Otherwise, if the parser *reduces*, the operator on the *right* is prioritized instead.

Precedence and *associativity* are two different but similar mechanisms to control how these shift-reduce conflicts are resolved. *Precedence* applies in the case of ambiguity between different operators, while *associativity* applies when the ambiguity is between multiple instances of the same operator, or between operators with the same precedence.

Precedence is simply an ordering of the operators, which establishes which operators shall be prioritized with respect to others. On the other hand, associativity is a property of each

[1]*Infix* notation is the usual way to write expressions, with the operator between the two operands. Other notations are *prefix* (operator before both operands) and *postfix* (operator after both operands), both of which are inherently not ambiguous. Prefix and postfix notation are also known as *polish* and *reverse-polish* notation respectively. In particular, reverse-polish notation is used in some scientific calculators.

single operator, and there are only three possible associativity classes: *left-associative*, *non-associative*, and *right-associative*. When an operator is *left-associative*, between two instances of an operator the one on the left is prioritized; a *right-associative* operator instead will prioritize the instance on the right. In *Non-associative* operators, instead, associating the operator is illegal and produces a syntax error. Figure 2.3 shows a summary of how precedence and associativity influence the parsing process of a sample expression.

Precedence			**Associativity**	
$a+b\times c$			$a+b+c$	
$+ > \times$	$\times > +$	Left	None	Right
$(a+b)\times c$	$a+(b\times c)$	$(a+b)+c$	*Syntax error*	$a+(b+c)$

Figure 2.3.: Example of how precedence and associativity influence how an expression is evaluated

Internal operation of the operator precedence parser Internally to a *bison*-generated LALR(1) parser, the implementation of operator precedences is simple. When a shift-reduce conflict occurs and the lookahead token has been assigned a given precedence and/or associativity, the parser decides what to do depending on the candidate rule to be reduced. In practice, the parser compares the last token in that rule with the lookahead.

If the token in the rule is the same one as the lookahead token, or if it is a different token with the same precedence, we must check associativity. If the associativity of the first token is *left*, the parser *reduces*. If the associativity of the first token is *right*, the parser *shifts*. If the first token is *non-associative*, a syntax error is generated. Note that *bison* does not allow declaring two tokens with the same precedence but with different associativity.

If the token in the rule is instead a different token than the lookahead, the precedences of these two tokens are compared. If the first token has *higher* precedence than the lookahead token, the parser *reduces*. On the contrary, the parser *shifts*.

During the construction of the pilot automaton[2], *bison* is always able to recognize all cases in which we have shift-reduce conflicts not resolved by appropriate precedence rules. Therefore, it's impossible for the parser to be in a situation where the candidate rule to be reduced does not have a token which decides its precedence.

Specifying precedence In *bison*, precedence must be explicitly declared in the source code of the parser: for ACSE, the *Acse.y* file. The order in which each operator's precedence is declared establishes the precedence itself: operators declared first have **lower** precedence than the operators declared later. All the operators contained in the same precedence declaration are assigned the same precedence.

There are four kinds of operator precedence declarations, and each of them also specifies the associativity of the operator:

[2]The *pilot automaton* is a state machine which drives the parsing process, and is derived from the language's grammar.

%left	Declares one or more left-associative operators.
%right	Declares one or more right-associative operators.
%precedence	Declares precedence of one or more operators without applying an associativity. If the grammar of the language allows the operator to be associated, *bison* produces an error.
%nonassoc	Declares precedence of one or more operators and disallows them to be associated. If the parsed input contains the same operator twice in a row, parsing stops and a syntax error is raised.

Important! Always remember to consider whether to specify precedence and/or associativity when adding a rule to the *exp* non-terminal!

2.4.6. Other kinds of ambiguities

Apart from conflicts derived from infix operators, grammars can contain other kinds of ambiguities generating a shift/reduce conflict. A typical example occurs in the *if/else* construct as defined in C and also in LANCE. When multiple *if* statements are nested together without explicitly delimiting their scope with braces, it is ambiguous to which *if* each *else* part belongs to. This is shown in fig. 2.4.

```
if (...)                 if (...)
   if (...)                 if (...)
      statement;               statement;
   else                  else
      statement;            statement;
```

Figure 2.4.: Different valid interpretations of a *dangling else*. The last *else* part may belong to either of the two *if* statements.

An alternative way provided by *bison* to solve shift/reduce conflicts apart from the precedence mechanism discussed in section 2.4.5 is the *%expect* declaration.

The *%expect* declaration can be found in *Acse.y*, and has the following form:

`%expect` ⟨*number*⟩

where in the base implementation of ACSE the number parameter is set to 1. This declaration tells *bison* that it is acceptable to have unresolved shift/reduce conflicts, if the number of those conflicts is exactly the one specified by the *%expect* declaration. In that case, all shift/reduce conflicts will be resolved arbitrarily by *bison*, typically by preferring *shift* over *reduce*.

In ACSE, the *%expect* declaration is necessary because of the presence of the dangling else ambiguity. When other similar ambiguities are introduced, the number of conflicts to be expected needs to be increased.

Acse.lex

```
"execute" { return EXECUTE; }
"when"    { return WHEN; }
```

axe-struct.h

```
typedef struct exec_when_t {
   t_axe_label *l_block;
   t_axe_label *l_exp;
   t_axe_label *l_exit;
} exec_when_t;
```

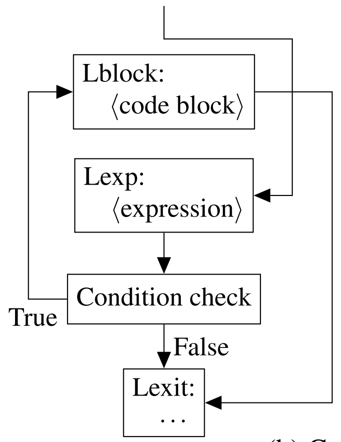
Acse.y

```
%union {
   /* ... */
   exec_when_t exec_when_data;
}

/* ... */

%token <exec_when_data> EXECUTE
%token WHEN
```

Acse.y

```
exec_when:
   EXECUTE
   {  /* semantic action 1 */
      $1.l_exp = newLabel(program);
      gen_bt_instruction(program, $1.l_exp, 0);
      $1.l_block = assignNewLabel(program);
   }
   code_block
   {  /* semantic action 2 */
      $1.l_exit = newLabel(program);
      gen_bt_instruction(program, $1.l_exit, 0);
      assignLabel(program, $1.l_exp);
   }
   WHEN LPAR exp RPAR
   {  /* semantic action 3 */
      if ($7.expression_type == IMMEDIATE) {
         if ($7.value != 0)
            gen_bt_instruction(program, $1.l_block, 0);
      } else {
         gen_bne_instruction(program, $1.l_block, 0);
      }
      assignLabel(program, $1.l_exit);
   };
```

(a) Required modifications to ACSE.



```
    BT Lexp
Lblock:
    code block
    BT Lexit
Lexp:
    expression

    BNE Lblock


Lexit:
    ...
```

$\left.\begin{array}{l}\texttt{BT Lexp}\\\texttt{Lblock:}\end{array}\right\}$ Generated by semantic action 1

} Generated by *code_block*

} Generated by semantic action 2

} Generated by *exp*

} Generated by semantic action 3 (expression is not constant)

(b) Control-flow-graph and generated code

Figure 2.5.: Illustration of how to extend ACSE to support the *execute-when* control statement.

2.4.7. Execution order and generation order

An important limitation of syntactic-driven translation is that the order in which code is generated reflects the order in which each syntactic element is parsed. Since in most programming languages the order in which code appears in the syntax also reflects the order in which code is executed – barring control structures – this tends to not be a problem.

Sometimes, however, it is necessary to force an execution order which is different than the order in which code is generated. This can be done by judicious use of unconditional branches and shared variables across semantic actions. Let us illustrate this process with an example.

The execute-when statement Let us consider a fictional control statement named *execute-when*, which is not different from a standard *if* then except that the block of code and the condition are inverted.

```
execute
   code block
when ( expression )
```

We define tokens EXECUTE and WHEN for the “execute” and “when” words respectively. The grammar of the statement to be added to *Acse.y* is the following:

```
exec_when: EXECUTE code_block WHEN LPAR exp RPAR;
```

Both the *code_block* and *exp* non-terminals in the grammar rule generate code, corresponding to the translation of the conditional block and the conditional expression. In the output assembly listing the code corresponding to the block will appear earlier than the code of the expression.

If no additional branches are inserted, at run-time the block is executed first, and then the expression value is computed (except if it is a constant, of course). But in order to execute the block only when the expression value is non-zero, we have to compute the expression first. To achieve this behavior, we must generate additional branches, which unconditionally skip over the block and go directly to the expression computation code.

The complete set of modifications to ACSE required for implementing the control structure correctly is shown in fig. 2.5, together with the control-flow-graph and the corresponding assembly code that the compiler generates.

Note that special code is generated in case the expression is constant, and we rely on the fact that the *PSW* register is already up-to-date after the execution of the code generated by the semantic actions of the *exp* non-terminal.

2.4.8. Common mistakes

Extending ACSE is not a particularly complex process, but it requires through knowledge of MACE assembly language, *bison* syntax and semantics, and of the principles of operation of sintactic-driven-translation. When this knowledge is lacking, however, embarrassing and obvious in-hindsight mistakes are often made.

This section outlines the most common of these mistakes using as examples *actual attempts* at extending ACSE, and briefly explains the consequence of each mistake.

Note: Most of these mistakes are caused by a confusion about what happens at compile-time and what happens at run-time. In other words, to the inexperienced programmer it might appear that somehow every C language structure that appears in the semantic actions of *Acse.y* gets translated to code in the output of the compiler. This is not the case: there is no magical device that converts C code that appears in semantic actions into assembly language inside ACSE, and if it did exist we would not be here talking about compilers because the magical device would be the compiler.

Instead, what truly happens is that the functions implemented by ACSE create data structures akin to instructions, and insert them into the output. These functions operate just like *printf()*: they do not do any computation themselves. They simply print out specific assembly language instructions. These functions have been discussed earlier in section 2.2.4.

Using register IDs as the value they represent Perhaps the most common mistake made by inexperienced programmers in extending ACSE is to assume that integer variables holding a register identifier can be used to access the value that the register will have at runtime.

However, these variables only contain an arbitrary integer, which is the "name" of the register. This integer tells **which** register we are referring to, but holds no information about **what** will be contained in the register. After all, unless we execute the program, we can't predict the value that each MACE register will contain.

Listing 2.18 shows a typical example of this mistake. The incorrect code has been adapted from the solution of a classroom assignment, written by a student at Politecnico di Milano on January 19, 2021.

```c
char *a1 = /* ... */;
char *a2 = /* ... */;
t_axe_expression i = /* ... */;

if (load_array_element(program, a1, i) == load_array_element(program, a2, i)) {
    /* ... */
} else if (load_array_element(program, a1, i) < load_array_element(program, a2, i)) {
    /* ... */
} else if (load_array_element(program, a1, i) > load_array_element(program, a2, i)) {
    /* ... */
}
```

Listing 2.18.: **Incorrect** code. The register identifiers are compared, rather than the contents of the registers themselves.

The student forgot that *load_array_element()* returns a register identifier, and thought that the return value was the contents of the array proper. Except that it is impossible to know the contents of an array at compile time! Without exaggeration, there is not a single line of code in the listing that is not wrong in some way. In listing 2.19, we show the correct code which actually achieves what the student was meaning to do.

```
char *array1 = /* ... */;
t_axe_expression i = /* ... */;

int r_a = load_array_element(program, array1, i);
int r_b = load_array_element(program, array2, i);
gen_sub_instruction(program, REG_0, r_a, r_b, CG_DIRECT_ALL);
t_axe_label *l_notequal = newLabel(program);
t_axe_label *l_greater = newLabel(program);
t_axe_label *l_done = newLabel(program);
gen_bne_instruction(program, l_notequal, 0);
/* ... */
gen_bt_instruction(program, l_done, 0);
assignLabel(program, l_notequal);
gen_bgt_instruction(program, l_greater, 0);
* ... */
gen_bt_instruction(program, l_done, 0);
assignLabel(program, l_greater);
/* ... */
assignLabel(program, l_done);
```

Listing 2.19.: **Correct** code. The comparison is performed by generating the appropriate instructions.

Important! In general, an easy rule to remember is that it does not make sense – and therefore is incorrect – to do any kind of mathematics or comparison on register identifiers.

Expressions: not handling non-constant expressions Another common mistake is handling expression incorrectly. There are disparate ways to do so.

The most common mistake is forgetting to handle either the IMMEDIATE or the REGISTER expression type. Since the meaning of the *value* field of the *t_axe_expression* struct varies depending on the type of the expression, it is easy to generate incorrect code when not explicitly checking the type of expression being handled.

One example appears in listing 2.20, adapted from a student's (incorrect) solution to an assigment given on July 12, 2021.

```
some_statement:
   TOKEN1 IDENTIFIER exp
   {
      int r_loc = get_symbol_location(program, $2, 0);
      gen_move_immediate(program, r_loc, $3.value);
      /* ... */
   };
```

Listing 2.20.: **Incorrect** code. The case in which the expression is not constant is not handled.

The student wanted to assign a value to a scalar variable, but they forgot that the value of the expression identified by *$3* could be both a register identifier or a constant value. Explicitly checking for both cases fixes the issue, as shown in listing 2.21 This is just one example, but other similar cases are almost countless.

```
some_statement:
   TOKEN1 IDENTIFIER exp
   {
      /* ... */
      int r_loc = get_symbol_location(program, $2, 0);
      if ($3.expression_type == IMMEDIATE)
         gen_move_immediate(program, r_loc, $3.value);
      else
         gen_addi_instruction(program, r_loc, $3.value, 0);
      /* ... */
   };
```

Listing 2.21.: **Correct** code. We explicitly check if the expression's type is IMMEDIATE or not.

Expressions: generating branches incorrectly A particularly baffling variant of the mistake described above manifests when the value returned by *handle_binary_comparison()* is used directly in a conditional statement such as *if*. A snippet of code adapted from yet another student's work made in February 2021 has been reproduced in listing 2.22.

```
t_axe_expression e1 = create_expression(/*...*/, IMMEDIATE);
t_axe_expression e2 = create_expression(/*...*/, REGISTER);
if (handle_binary_comparison(program, e1, e2, _LT_)) {
  /* ... */
}
```

Listing 2.22.: **Incorrect** code. It makes no sense to put the result of *handle_binary_-comparison* in a condition.

This does not even qualify as valid C code, as the condition in an *if* statement must be a scalar value which can be casted to an integer (*int*), and of course a structure of type *t_axe_expression* will not do. The correct code is shown in listing 2.23.

```
t_axe_expression e1 = create_expression(/*...*/, IMMEDIATE);
t_axe_expression e2 = create_expression(/*...*/, REGISTER);
handle_binary_comparison(program, e1, e2, _LT_);
t_axe_label *l_skip = newLabel(program);
gen_beq_instruction(program, l_skip, 0);
/* ... */
assignLabel(l_skip);
```

Listing 2.23.: **Correct** code. The result of the comparison is only known at run-time, so we have to generate the appropriate branches.

Expressions: assuming that code is always generated Another common mistake in handling expressions is forgetting that, in certain situations, expression operations do not generate code.

One might be tempted to use *handle_bin_numeric_op()* and *handle_binary_comparison()* to generate code for performing mathematical computations inside a loop, for example to increment a value. But to do so, we must remember two things: first, these functions do not generate any code when both the input expressions are constant (IMMEDIATE), and in case they do indeed generate code, they will always reserve a new register to store the results. When these two points are forgotten, incorrect code such as what is shown in listing 2.24 may result.

```
t_axe_expression e_1 = create_expression(0, IMMEDIATE);
t_axe_label *l_loop = assignNewLabel(program);
/* ... */
e_1 = handle_bin_numeric_op(program, e_1, create_expression(1, IMMEDIATE), ADD);
/* ... */
gen_bt_instruction(program, l_loop, 0);
```

Listing 2.24.: **Incorrect** code. No assignment is made, and no code for implementing the addition is generated.

The code in this example was meant to generate the code to increment a variable by one every time a loop iteration is performed. However, since both expressions passed to *handle_bin_numeric_op* are of IMMEDIATE type, no code is generated, and *e_1* is simply replaced by the constant expression '1' **at compile-time**. In listing 2.25 the problem is fixed by reserving a new register initialized to zero to use as a counter variable. Notice how we also don't re-assign the resulting expression to the source expression, as that does not generate the code for the assignment and would not result in a continuously incrementing variable. We must explicitly generate an *ADDI* instruction to perform the assignment.

```
int r_counter = gen_load_immediate(program, 0);
t_axe_expression e_zero = create_expression(r_counter, REGISTER);
t_axe_label *l_loop = assignNewLabel(program);
/* ... */
t_axe_expression e_temp = handle_bin_numeric_op(program,
      r_counter, create_expression(1, IMMEDIATE), ADD);
gen_addi_instruction(program, r_counter, e_temp.value, 0);
/* ... */
gen_bt_instruction(program, l_loop, 0);
```

Listing 2.25.: **Correct** code. We explicitly allocate a register for the counter, in order to ensure that *handle_bin_numeric_op* generates code. It is safe to assume the type of *e_temp* is 'REGISTER' because one operand to the expression is always of type 'REGISTER'.

Confusion between run-time and compile-time assignments A different mistake entirely consists in confusing assignments in C with the generation of code that assigns a value to a register. This simple but frequent fallacy involves the belief that somehow the assignment of one expression into another actually generates the code for moving a value from one register to another. This usually happens in response to a need to assign a different value to a subexpression depending on the result of a branch. A very typical case of this mistake is shown in listing 2.26, again a listing adapted from the work of a student dating from September 2, 2012.

```
t_axe_label *l_else = newLabel(program);
t_axe_label *l_break = newLabel(program);
gen_bne_instruction(program, l_else, 0);
$$ = create_expression(0, IMMEDIATE);
gen_bt_instruction(program, l_break, 0);
assignLabel(program, l_else);
$$ = create_expression(r_register, REGISTER);
assignLabel(program, l_break);
```

Listing 2.26.: **Incorrect** code. Only the second assignment to *$$* matters, because the *gen_bcc_instruction()* functions do not skip the code in the compiler, and because assigning to *$$* does not generate any code.

Grasping the mental mechanism that underlies this mistake has eluded generations of teachers. The only cure is to learn what generates code and what does not, and avoiding superficial reasoning based on how the code looks. To achieve this, we encourage reading this manual from top to bottom. Anyway, the listing shown above can be fixed by assigning to *$$* only once, and reserving a register specifically for the value being returned.

```
t_axe_label *l_else = newLabel(program);
t_axe_label *l_break = newLabel(program);
int r_dest = getNewRegister(program);
$$ = create_expression(r_dest, IMMEDIATE);
gen_bne_instruction(program, l_else, 0);
gen_addi_instruction(program, r_dest, REG_0, 1);
gen_bt_instruction(program, l_break, 0);
assignLabel(program, l_else);
gen_addi_instruction(program, r_dest, r_register, 0);
assignLabel(program, l_break);
```

Listing 2.27.: **Correct** code. Only one assignment to *$$* is performed, which wraps a single register in an expression. A different value is assigned to the register at run-time, depending on which branch is taken.

Important! We did this reasoning with the specific example of an assignment to *$$*, but it is valid with any other C statement “enclosed” within branches. Remember: **all branch-generating functions (*gen_b*cc*_instruction()*) do not affect the execution flow of the compiler!**

Forgetting the special behavior of $$ in mid-rules One last fairly common mistake, albeit a more technical one, is to forget that *$$* assigns the semantic value corresponding to the current non-terminal being expanded only in the end-rule action.

```
non_terminal:
   TOKEN1 TOKEN2
   {
      /* This does not assign a semantic value to non_terminal,
       * but to this semantic action. */
      $$ = /* ... semantic value ... */;
   }
   TOKEN3 TOKEN4
   {
      /* At this point, $3 is equal to the value assigned to $$ in the rule above. */
   };
```

Listing 2.28.: Example of how *$$* does not assign a value to the non-terminal being expanded in mid-rule actions.

In mid-rule actions, *$$* is bound to the semantic action of the *mid-rule semantic action itself*. This value can be casted to the appropriate data type – as defined in the *union* declaration, see section 2.3.1 – using the following syntax:

`%<`*type*`>%` and `%<`*type*`>`*n*

Anyway, to assign the semantic value of subexpressions or other non-terminals, always remember to do the assignment to *$$* in the end-of-rule semantic action.

2.5. Internals of the ACSE frontend

In this section we will briefly discuss the internal data structures found in the frontend of ACSE.

2.5.1. Internal data representation of instructions

The primitive APIs for generating new instruction structures is the *alloc_instruction()* function. The instructions generated in this way are not automatically added in a program, nor they are directly usable. The various fields of the returned instruction must be filled in manually by the user.

Once the instruction structure has been fully prepared, its pointer must be passed to the *add_instruction()* function defined in *axe_engine.h* in order to add it to the program at the end of the instruction list.

Data structures for assembly instructions and assembler directives are defined in the file *axe_struct.h*. All data structure definitions related to instructions are shown in listing 2.29.

An assembly instruction is described by an operation identifier (*opcode*, for example 'SUB', corresponding to an instruction name as shown in appendix B), a set of instruction parameters which depend on the instruction type, an optional user comment (*user_comment*) and a label identifier (*labelID*). The user comment is automatically generated to reflect the line of code in the input that produced the instruction. The label identifier, instead, is used to place a label pointing to that particular instruction.

```
typedef struct t_axe_label {
   int labelID;            /* label identifier */
   char *name;             /* optional name of the label */
} t_axe_label;

typedef struct t_axe_register {
   int ID;                 /* an identifier of the register */
   int indirect;           /* 1 for indirect addressing of the reg. */
} t_axe_register;

typedef struct t_axe_address {
   int addr;               /* a Program Counter */
   t_axe_label *labelID;   /* a label identifier */
   int type;               /* one of ADDRESS_TYPE or LABEL_TYPE */
} t_axe_address;

typedef struct t_axe_instruction {
   int opcode;             /* instruction opcode (for example: ADD) */
   t_axe_register *reg_1;  /* destination register */
   t_axe_register *reg_2;  /* first source register */
   t_axe_register *reg_3;  /* second source register */
   int immediate;          /* immediate value */
   t_axe_address *address; /* an address operand */
   char *user_comment;     /* comment for the instruction */
   t_axe_label *labelID;   /* label associated with the instruction */
} t_axe_instruction;
```

Listing 2.29.: Definition of *t_axe_instruction* and related types.

The operation identifiers are actually C macro defines, which can be found in the file *axe_constants.h*.

Depending on the instruction, valid parameters can be register identifiers (*reg_1*, *reg_2*, *reg_3*), immediate signed integer values (*immediate*), or an address (*address*). Ternary instruction shall have only the three register identifiers. Binary instruction shall have two registers (*reg_1* and *reg_2*) and an immediate argument. Unary instructions shall have one register (*reg_1*) and the address argument. Jump instructions only have the address argument.

Unused address or register arguments are set to *NULL*. If an instruction does not have an immediate argument, the *immediate* field is ignored.

Used address or register arguments are represented by a dynamically allocated pointer to the appropriate data structure: *t_axe_register* or *t_axe_address*.

Registers This register identifier appears in the *ID* field of *t_axe_register*.

The *t_axe_register* also has the capability of representing indirectly addressed registers. These are the registers that, in an assembly listing, appear enclosed by parenthesis, such as register *R2* in the following example:

```
ADD R3 R1 (R2)
```

These indirectly addressed registers are represented by setting the *indirect* field to the value *1*. Otherwise, this field must be set to zero.

2.5.2. The Label Manager

The *Label Manager* is defined in *axe_labels.h*, and it consists of a list of all the labels defined by the program. The interface of the Label Manger consists of a set of low-level functions to work with labels, the most important of which are *newLabelID()* and *assignLabelID()*.

newLabelID() is used when the user code requires the creation of a new label. *assignLabelID()* is used to assign a given label to an instruction.

Note that both the function *newLabel()* and the function *assignLabel()* are defined as wrappers for the *newLabelID()* and *assignLabelID()* functions respectively.

2.5.3. Internal data representation of variables

Every time a program variable declaration is found in the source code, an instance of *t_axe_variable* is created and added to the list of variables of a *t_program_infos* instance by calling the *createVariable()* function as discussed in section 2.2.7.

In the SDT, *createVariable()* is not used directly; instead, a *t_list* of declarations (*t_axe_declaration*) is built, and then it is passed to the *set_new_variables()* function, which calls *createVariable()* for each declaration in the list.

At variable creation, an assembler directive is also generated – as we will discuss later in section 2.5.4 – to store the variable's value in memory. `.WORD` directives are used for scalar variables, while `.SPACE` directives are used for arrays (see the assembler documentation in chapter 3).

All the instances of *t_axe_variable* are stored in the list in the current *program* instance. Additionally, the variable is added to a separate data structure, called *symbol table*. Each entry of the symbol table is associated to a single program variable, and it is defined as shown in listing 2.30.

```
typedef struct {
   char *ID;            /* Variable identifier */
   int type;            /* Type associated with the variable */
   int reg_location;    /* A register identifier */
} t_symbol;
```

Listing 2.30.: Definition of *t_symbol*.

The register identifier refers to the register identifier (a machine general-purpose register) where the variable is currently stored.

The file *symbol_table.h* declares the functions needed to manipulate the content of a symbol table: define and insert a new symbol into a symbol table (*putSym()*), set the register location information of a symbol (*setLocation()*), and retrieve the register location information associated with a symbol (*getLocation()*). Finally, *symbol_table.h* includes a file called *sy_table_constants.h* which defines a set of macros used for error tracking.

2.5.4. Internal data representation of directives

In ACSE, the list of assembler directives to generate in the output is kept in the *data* list found in *t_program_infos*. This list contains instances of the *t_axe_data* structure, whose definition is shown in listing 2.31.

```
typedef struct t_axe_data {
   int directiveType;       /* the type of the current directive
                             * (for example: DIR_WORD) */
   int value;               /* the value associated with the directive */
   t_axe_label *labelID;    /* label associated with the current data */
} t_axe_data;
```

Listing 2.31.: Definition of *t_axe_data*.

2.6. Internals of the ACSE backend

In this section, we will very briefly discuss what happens inside ACSE after the SDT completes, and before the assembly language code is generated. This stage is called the *backend*, and involves some additional transformations of the code to make it suitable for execution by the MACE machine.

The backend is organized in multiple stages. The *target-dependent transformations* stage performs some initial normalization of the code, currently only the removal of all immediate arguments larger than 16 bit. The *control-flow analysis* stage computes a control-flow-graph from the code, and the *liveness analysis* determines how long each register is used. This information is then used to *insert load/store instructions* around variable accesses, and for performing *register allocation*. Finally, the list of instructions is reconstructed from the control-flow-graph and it is printed – together with the assembler directives – to the compiler's output file.

2.6.1. Target-dependent transformations

The entry point for all target-dependent transformations is defined in the *axe_target_transform.h* file, and it consists of the *doTargetSpecificTransformations()* function.

This function is a wrapper for the *fixLargeImmediates()* function defined in the file *axe_-target_transform.c*. This function scans the list of instructions found in the *program* instance, and when it finds an instruction with an immediate argument, it checks if that immediate argument is in the range $[-32768, 32767]$. This interval corresponds to the signed numbers representable in 16 bits.

If an immediate argument is found which does not fit in that range, additional instructions are generated to "split the immediate in pieces" and load it into a register. For example, the number 50 000 000 is reconstructed by generating code which computes the following expression:

$$763 \times 2^{16} - 3968$$

Then, the original instruction is modified to the ternary format, and the third argument is changed to the register containing the constant.

2.6.2. Control Flow Analysis

The transformations performed by the previous pass simply consisted of replacing single instructions with a sequence of multiple instructions, with no global impact on the rest of the program. All local transformations that operate on brief sequences of instructions are called *peephole transformations*. Other transformations instead require a deeper knowledge of the structure of the program. To collect the data required for performing them, the next step for ACSE is to compute the *control-flow-graph* of the program.

In very simple terms, a control-flow-graph (CFG) is none other than a form of flow-chart. The blocks in a CFG are called *basic blocks*, and they simply consist of straight-line lists of instructions. The execution flow is represented by the edges of the graph.

The interface of the CFG generation pass in ACSE is found in the *axe_cflow_graph.h* file, which also defines a function called *createFlowGraph()* which builds a CFG from a list of instructions. The list of instructions is split in different basic blocks after each branch or *HALT* instruction in the list, and before each label. Then, the edges of the graph are inserted, by looking at all the branch instructions in the program (see the *updateFlowGraph()* function in *axe_cflow_graph.c*).

2.6.3. Liveness Analysis

After the control flow analysis, the function *performLivenessAnalysis()* is called to perform the *liveness analysis*. This analysis reads the control flow graph, and uses it to determine the *liveness interval* of each register. The liveness interval of a register is none other than the range of instructions where the register is actually needed in the program. The interval starts at the first instruction in the program which assigns a value to a register, and it ends at the point where no further instructions in the program will use the register.

Liveness analysis is a fixed-point algorithm: it scans the control flow graph multiple times until we reach a point where no modification is done to the liveness intervals. Each pass of the algorithm follows the list of basic blocks in the program in *reverse* order, propagating the list of registers currently alive between each instruction using the following induction rules:

- If an instruction "reads" (or, more properly, *uses*) a register, it means that earlier some other instruction assigned a value to that register, and thus it must be "alive" before that instruction.
- If an instruction "assigns" (or, more properly, *defines*) a register, it means that no instruction before it is using the value being assigned here. Therefore, we consider the variable not "alive" before this instruction.

This algorithm is explained in more detail in *Compilers: Principles, Techniques and Tools*, by A. V. AHO, M. S. LAM, R. SETHI and J. D. ULLMANN.

2.6.4. Load/Store insertion

As we described in section 2.2.7, every scalar variable is associated to a register which contains its value, and assignments to variables are simply assignments to that register. However, variables are also meant to be accessed through the memory location dedicated to them. The most important case in which this is needed in base ACSE is for loading the initial value of variables, which is stored in the *data* segment of the executable. Another case in which access to the memory location of variables is needed is in case ACSE is extended for pointer operations.

The *load/store insertion* pass is a compilation step unique to ACSE which detects accesses to the register associated to each variable, and inserts appropriate *LOAD* and *STORE* instructions to keep the contents of the variable's memory location in synchronization with the variable's associated register. This operation is performed by the *insertLoadAndStoreInstr()* function defined in *axe_transform.h*.

The *LOAD* and *STORE* instructions are only inserted before the first use of each variable, and after the last definition of each variable in each basic block. As a result, the memory cell of the variable is only updated at each basic-block boundary.

2.6.5. Register allocation

After load/store insertion, the last transformation pass executed by ACSE in the code is *register allocation*. Up until now, the code of the program being compiled assumes that an infinite amount of *virtual* registers are available, but the MACE machine only has 32 *physical* registers (31 without counting the special-purpose *R0*). The purpose of register allocation is to analyze the program and transform it such that only 31 registers at most are used at a given time.

The main challenge of register allocation consists in the fact that *virtual* registers that are being used at the same moment – in other words, registers whose live intervals overlap – must be allocated to *different physical registers*. If we represent each virtual register with a node in a graph, and if we connect registers with overlapping live intervals with an edge, register allocation simply becomes the task of coloring the nodes in the graph in such a way that adjacent nodes have different colors. Each color represents a physical register. This *graph coloring* problem is one of the classical problems in computer science, and the bad news is that minimizing the number of colors is an NP-complete task.

The algorithm used by ACSE to perform register allocation is called *linear-scan*, and it is described in detail in M. POLETTO and V. SARKAR, *Linear Scan Register Allocation*, ACM Transaction on Programming Languages and Systems, Vol. 21, No. 5. It is not an exact algorithm, but it provides a good approximation and it is fast. As a result, linear scan is currently being used even in production-quality compilers such as GCC.

Like its name suggests, this algorithm scans the list of live intervals previously generated by the liveness analysis in order of starting point. Each new liveness interval is associated to a new register from a pool of currently unused registers, and at the end of each interval its associated register is put back in the unused register pool.

When the register pool is empty but we still need more registers, the interval is *spilled* instead. Spilled intervals are stored not in a register, but in a memory location specifically allocated. Three registers are reserved as locations where to temporarily store the values from spilled virtual registers, making the effective number of registers available to the register allocator 28.

2.6.6. Code generation

The final compilation step in ACSE is the *code generation*. In this step, the list of directives and the list of instructions of the program is read, and the assembly language output corresponding to each instruction or directive is generated. This step is performed by the *writeAssembly()* function defined in *axe_target_asm_print.h*.

3. Assembler

In this chapter we describe how the assembler translates the output produced by the ACSE compiler into a valid executable (an object file) for the MACE architecture.

The assembler takes an assembly language file as input (i.e., a symbolic assembly program – the output of a compilation process) containing both instructions and assembler directives. Each assembly statement is directly translated into a specific machine code instruction, according to the "binary format rules" discussed in the MACE documentation. On the contrary, assembler directives do not get translated in any machine code instructions, but they affect the way the assembly listing is interpreted, or they generate static allocations. The output of the translation process is an object file, containing both machine code instructions and data allocation and initialization information.

In the following sections, we will describe how the assembler program works (the translation steps performed and their order); then we introduce the assembly language and the assembler directives supported by the current implementation of the assembler. Finally, the last section describes the internal structure of an object file and its binary format.

3.1. How the assembler works

An assembler is defined as a program that translates an assembly file into an object file for a specific architecture. The binary format of an object file typically depends on both the underlying architecture and the operating system. Thus, the structure of an object file may vary and typically contains a lot of information other than machine code and data information (for example, a symbol table that is used by a linker for code relocation purposes).

In general, each assembly instruction is directly mapped to an equivalent machine code instruction. However, assembly code may also contain *assembler directives* and symbols (i.e., labels) that cannot be directly translated into machine code. Usually an assembler directive is used (for example) to indicate the beginning of a block of data to the assembler. Labels can be assigned to specific memory locations; the assembler has the job of translating all the labels to valid memory addresses.

Also, an assembler verifies the correctness of the assembly code given as input: it performs a syntactic analysis on the input file and prints all the encountered errors to the standard error stream. As post-condition, only valid assembly files will be translated to "well formed" object files.

We can formalize the behavior of an assembler in the following three macro-phases:

1. Data structure initialization.
2. Parsing of the input assembly code, validation of every instruction/directive.
3. Generation of the output object file, using all the information gathered during the parsing process.

In the first phase, the assembler initializes various internal data structures for future use/-modification (during the parsing process). Those data structures will be filled with information about data directives and symbolic instructions. This information will be used in the last phase in order to produce a valid object file.

The second phase consists of a parsing process where instructions are first validated (i.e., the syntax and consistency of each instruction is checked) and then translated to an intermediate form. Assembler directives are always interpreted and never translated into machine instructions. Assembler directives typically are used to manipulate the content of the data segment.

In the last phase all the information gathered during the parsing process are finally used to make an object file that will be written and returned as the output of the whole program.

3.2. Assembly format

In this section we will introduce the format of an assembly file and the syntax of each supported instruction and data directive.

The format of an assembly source file is shown in listing 3.1.

```
.DATA
    Data directive 1
    Data directive 2
    ...
    Last data directive
.TEXT
    Instruction 1
    Instruction 2
    ...
    Last instruction
```

Listing 3.1.: Overall format of a valid assembly language file.

The assembler directive `.DATA` marks the beginning of a data section; within that section space for data and variables can be allocated (see section 3.2.1). The directive `.TEXT` marks the beginning of the code section, which contains assembly instructions. This must always be the last section in an assembly file.

Comments follow the C block syntax, and can appear anywhere in the assembly listing.

This is the format of a directive or instruction line:

[*Label*:] *Instruction or directive* [`/*` *comment* `*/`]

Both label and comment are optional, and this is indicated by surrounding them with square brackets. A label can be referenced as an operand of an instruction (e.g., jumps), while comments are ignored by the assembler.

An instruction specifies an operation type and a list of operands. Operands can be register identifiers, directly or indirectly addressed; immediate values, i.e, a number specified within the instruction; or address values (typically labels).

Unary and binary instructions always use direct addressing, while ternary instructions can use both direct and indirect addressing with some limitations. For a more through discussion of addressing modes in MACE, see section 1.1.3. The list of every instruction can be found in appendix B.

3.2.1. Assembler Directives

Assembler directives have their own semantics and format, separate from instructions. The current implementation provides only a very limited set of assembler directives, a subset of the *GNU assembler directives*. All assembler directives have names that begin with a period ('.') and every directive has its own semantic associated with.

A list of all the supported directives follows.

.data Marks the beginning of a block of data directives.

.text Marks the beginning of a block of instructions.

.word Reserves and initializes a memory word (32-bits) in the data segment.

Syntax: .word ⟨*value*⟩

Semantics: Reserves a 32-bit memory location inside the data segment and set the starting value of the location to ⟨*value*⟩.

Examples:

`.word 5`	reserve a word location and set its content with the 32-bit integer value '5'
`L1 .word 0`	reserve a word location and set its content to the 32-bit integer value '0'; the location can be referred to with the label `L1`

.space This directive reserves a given number of words into the data segment.

Syntax: .space ⟨*value*⟩

Semantics: Reserve ⟨*value*⟩ (contiguous) 32-bit words inside the data segment. The reserved memory is filled with zeros.

Examples:

`.space 32`	32 contiguous words reserved
`L1 .space 5`	5 contiguous words reserved; the memory can be referred to with the label `L1`

3.3. Object file format

As a result of the assembling process, an object file is produced. It contains both machine code instructions and the initial content of the data segment. Figure 3.1 shows the format of an object file.

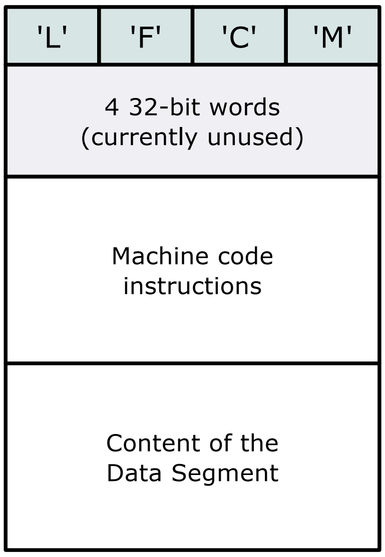

'L'
'F'
'C'
'M'
4 32-bit words (currently unused)
Machine code instructions
Content of the Data Segment

Figure 3.1.: Object file format

An object file is composed by a 20-byte header followed by the instruction segment and the data segment. The first 4 bytes of the header must always contain the ASCII binary representation of the ‘L’ ‘F’ ‘C’ ‘M’ characters; the following 16 bytes are currently unused.

The instruction and data segments are loaded into memory by MACE exactly as they are in the file, so no relocation information is stored in the object file. All the symbolic addresses (i.e., labels) are resolved by the assembler before emitting the object file.

A. LANCE Language Specification

This appendix contains the detailed specification of the syntax of the LANCE language.

A.1. Token Definitions

The following table shows the regular expressions used for matching and identifying every token required by the LANCE grammar.

Token	Regular Expression
DIGIT	[0-9]
ID	[a-zA-Z_][a-zA-Z0-9_]*
LBRACE	“{”
RBRACE	“}”
LSQUARE	“[”
RSQUARE	“]”
LPAR	“(”
RPAR	“)”
SEMI	“;”
PLUS	“+”
MINUS	“-”
MUL_OP	“*”
DIV_OP	“/”
AND_OP	“&”
OR_OP	“|”
NOT_OP	“!”
ASSIGN	“=”
LT	“<”
GT	“>”
SHL_OP	“<<”
SHR_OP	“>>”
EQ	“==”
NOTEQ	“!=”
LTEQ	“<=”
GTEQ	“>=”
ANDAND	“&&”
OROR	“||”
COMMA	“,”
DO	“do”

Token	Regular Expression
ELSE	“else”
IF	“if”
TYPE	“int”
WHILE	“while”
RETURN	“return”
READ	“read”
WRITE	“write”

A.2. Grammar Definition

The Backus-Naur form of the LANCE language grammar follows.

In the grammar, terminal symbols (or tokens, see appendix A.1) are shown in regular type, and non-terminal symbols are shown in *italic* type. The axiom of the grammar is the *program* non-terminal, and the empty string is represented by the character ε. Finally, $\perp$ represents the end of the string.

Non-terminal	Expansion
program :	*var_declarations statements* $\perp$
var_declarations :	*var_declarations var_declaration*
|	ε
var_declaration :	TYPE *declaration_list* SEMI
declaration_list :	*declaration_list* COMMA *declaration*
|	*declaration*
declaration :	IDENTIFIER ASSIGN NUMBER
|	IDENTIFIER LSQUARE NUMBER RSQUARE
|	IDENTIFIER
code_block :	*statement*
|	LBRACE *statements* RBRACE
statements :	*statements statement*
|	*statement*
statement :	*assign_statement* SEMI
|	*control_statement*
|	*read_write_statement* SEMI
|	SEMI
control_statement :	*if_statement*
|	*while_statement*
|	*do_while_statement* SEMI
|	*return_statement* SEMI
read_write_statement :	*read_statement*
|	*write_statement*
assign_statement :	IDENTIFIER LSQUARE *exp* RSQUARE ASSIGN *exp*
|	IDENTIFIER ASSIGN *exp*

Non-terminal	Expansion
if_statement :	*if_stmt*
|	*if_stmt* ELSE *code_block*
if_stmt :	IF LPAR *exp* RPAR *code_block*
while_statement :	WHILE LPAR *exp* RPAR *code_block*
do_while_statement :	DO *code_block* WHILE LPAR *exp* RPAR
return_statement :	RETURN
read_statement :	READ LPAR IDENTIFIER RPAR
write_statement :	WRITE LPAR *exp* RPAR
exp :	NUMBER
|	IDENTIFIER
|	IDENTIFIER LSQUARE *exp* RSQUARE
|	NOT_OP *exp*
|	*exp* AND_OP *exp*
|	*exp* OR_OP *exp*
|	*exp* PLUS *exp*
|	*exp* MINUS *exp*
|	*exp* MUL_OP *exp*
|	*exp* DIV_OP *exp*
|	*exp* LT *exp*
|	*exp* GT *exp*
|	*exp* EQ *exp*
|	*exp* NOTEQ *exp*
|	*exp* LTEQ *exp*
|	*exp* GTEQ *exp*
|	*exp* SHL_OP *exp*
|	*exp* SHR_OP *exp*
|	*exp* ANDAND *exp*
|	*exp* OROR *exp*
|	LPAR *exp* RPAR
|	MINUS *exp*

B. MACE Instruction Set Architecture

In this appendix we list all the instructions supported by the MACE architecture and its assembler.

The information provided about each instruction is: its assembler syntax, its description in words, the effect its execution has on the condition codes (i.e., the effect on the value stored inside the *PSW* register), and the addressing modes it accepts.

The following notation is used for the operands in the descriptions and the examples:

R*n*	Register *n*.
(R*n*)	Indirect addressing of register *n*. Square brackets ([R*n*]) are used in the description of the operation instead.
R⟨*dest*⟩	Destination Register.
R⟨*source1*⟩	First source operand (ternary instructions only).
R⟨*source2*⟩	Second source operand (ternary instructions only).
R⟨*source*⟩	Single source operand (binary instructions only).
#⟨*imm*⟩	Immediate value. *imm* is a signed 16-bit integer value.
L*n*	A label.

The effect of an instruction on the *PSW* is specified by the following codes:

?	The state of the bit is undefined (i.e., its value cannot be predicted).
-	The bit remains unchanged by the execution of the instruction.
*	The bit is set or cleared depending on the outcome of the instruction.
0	The bit is always cleared.

We use the following symbols to denote logical operations in the symbolic descriptions. When dealing with logical operations, any non-zero value is interpreted as a value of *true*, while zero values are interpreted as *false*. The result of such operations is always either zero or one. Bitwise operations, instead, operate on each bit of the 32-bit operands individually (in other words, bit by bit).

$\neg$	Logical NOT

$\wedge$	Logical AND
$\vee$	Logical OR
$\oplus$	Logican XOR
$\sim$	Bitwise NOT (one's complement)
&	Bitwise AND
$\vert$	Bitwise OR
ˆ	Bitwise XOR
>>	Shift right (i.e. '11100101' >> 1 = '01110010')
<<	Shift left (i.e. '11100101' << 1 = '11001010')
$\circlearrowleft$	Rotate left (i.e. '11100101' $\circlearrowleft$ 1 = '11001011')
$\circlearrowright$	Rotate right (i.e. '11100101' $\circlearrowright$ 1 = '11110010')

B.1. Conditional tests

Some instructions – such as jump instructions – compute a *conditional test* on the value of the PSW, and operate differently depending on the result of the test. Each test has an abbreviation that appears as 'cc' in the instruction set reference. The complete list of tests appears in table B.1.

Code	Meaning	Logical test
T	True	1
F	False	0
HI	Higher than	$\neg C \wedge \neg Z$
LS	Lesser than	$C \vee Z$
CC	Carry Clear	$\neg C$
CS	Carry Set	C
NE	Not Equal	$\neg Z$
EQ	Equal	Z
VC	Overflow Clear	$\neg V$
VS	Overflow Set	V
PL	Plus	$\neg N$
MI	Minus	N
GE	Greater or Equal	$(N \wedge V) \vee (\neg N \wedge \neg V)$
LT	Less Than	$(\neg N \wedge V) \vee (N \wedge \neg V)$
GT	Greater Than	$(N \wedge V \wedge \neg Z) \vee (\neg N \wedge \neg V \wedge \neg Z)$
LE	Less or Equal	$Z \vee (N \wedge \neg V) \vee (\neg N \wedge V)$

Table B.1.: Table of conditional tests

B.2. Ternary Instructions

Ternary instructions are instructions with three register operands. The first operand is the *destination*, while the other two are *sources*. The first source register (i.e., the second operand of the instruction) is always *directly* addressed, while both the destination and the second source registers can be either directly or indirectly addressed.

ADD **Addition**

Operation	$R\langle dest\rangle \leftarrow R\langle source1\rangle + R\langle source2\rangle$
Assembler Syntax	ADD R⟨*dest*⟩ R⟨*source1*⟩ R⟨*source2*⟩
Opcode	‘0000’
Description	Add the source operand R⟨*source1*⟩ to R⟨*source2*⟩ and store the result into the destination location R⟨*dest*⟩.

Condition Codes	`N Z V C`
	`* * * *`

Examples	ADD R3 R1 R2	$R3 \leftarrow R1 + R2$
	ADD R3 R1 (R2)	$R3 \leftarrow R1 + [R2]$
	ADD (R3) R1 (R2)	$[R3] \leftarrow R1 + [R2]$

SUB **Subtraction**

Operation	$R\langle dest\rangle \leftarrow R\langle source1\rangle - R\langle source2\rangle$
Assembler Syntax	SUB R⟨*dest*⟩ R⟨*source1*⟩ R⟨*source2*⟩
Opcode	‘0001’
Description	Subtract the source operand R⟨*source1*⟩ from R⟨*source2*⟩ and store the result into the destination location R⟨*dest*⟩.

Condition Codes	`N Z V C`
	`* * * *`

Examples	SUB R3 R1 R2	$R3 \leftarrow R1 - R2$
	SUB R3 R1 (R2)	$R3 \leftarrow R1 - [R2]$
	SUB (R3) R1 (R2)	$[R3] \leftarrow R1 - [R2]$

ANDL Logical AND

Operation	$R\langle dest\rangle \leftarrow R\langle source1\rangle \wedge R\langle source2\rangle$
Assembler Syntax	ANDL R⟨*dest*⟩ R⟨*source1*⟩ R⟨*source2*⟩
Opcode	‘0010’
Description	Perform an AND between the source operands R⟨*source1*⟩ and R⟨*source2*⟩ and store the result into the destination location R⟨*dest*⟩.

Condition Codes

```
N Z V C
* * 0 0
```

Examples

ANDL R3 R1 R2	$R3 \leftarrow R1 \wedge R2$
ANDL R3 R1 (R2)	$R3 \leftarrow R1 \wedge [R2]$
ANDL (R3) R1 (R2)	$[R3] \leftarrow R1 \wedge [R2]$

ORL Logical OR

Operation	$R\langle dest\rangle \leftarrow R\langle source1\rangle \vee R\langle source2\rangle$
Assembler Syntax	ORL R⟨*dest*⟩ R⟨*source1*⟩ R⟨*source2*⟩
Opcode	‘0011’
Description	Perform an OR between the source operands R⟨*source1*⟩ and R⟨*source2*⟩ and store the result into the destination location R⟨*dest*⟩.

Condition Codes

```
N Z V C
* * 0 0
```

Examples

ORL R3 R1 R2	$R3 \leftarrow R1 \vee R2$
ORL R3 R1 (R2)	$R3 \leftarrow R1 \vee [R2]$
ORL (R3) R1 (R2)	$[R3] \leftarrow R1 \vee [R2]$

EORL Logical Exclusive OR (XOR)

Operation	$R\langle dest\rangle \leftarrow R\langle source1\rangle \oplus R\langle source2\rangle$
Assembler Syntax	EORL R⟨*dest*⟩ R⟨*source1*⟩ R⟨*source2*⟩
Opcode	‘0100’
Description	EOR (exclusive or) the source operand R⟨*source1*⟩ with R⟨*source2*⟩ and store the result into the destination location R⟨*dest*⟩.

Condition Codes

```
N Z V C
* * 0 0
```

Examples

EORL R3 R1 R2	$R3 \leftarrow R1 \oplus R2$
EORL R3 R1 (R2)	$R3 \leftarrow R1 \oplus [R2]$
EORL (R3) R1 (R2)	$[R3] \leftarrow R1 \oplus [R2]$

ANDB Bitwise AND

Operation	R⟨*dest*⟩ ← R⟨*source1*⟩ & R⟨*source2*⟩	
Assembler Syntax	ANDB R⟨*dest*⟩ R⟨*source1*⟩ R⟨*source2*⟩	
Opcode	‘0101’	
Description	ANDB the source operands R⟨*source1*⟩ and R⟨*source2*⟩ and store the result into the destination location R⟨*dest*⟩.	

Condition Codes	`N Z V C`
	`* * 0 0`	

Examples	ANDB R3 R1 R2	R3 ← R1 & R2
	ANDB R3 R1 (R2)	R3 ← R1 & [R2]
	ANDB (R3) R1 (R2)	[R3] ← R1 & [R2]

ORB Bitwise OR

Operation	R⟨*dest*⟩ ← R⟨*source1*⟩ | R⟨*source2*⟩	
Assembler Syntax	ORB R⟨*dest*⟩ R⟨*source1*⟩ R⟨*source2*⟩	
Opcode	‘0110’	
Description	ORB the source operands R⟨*source1*⟩ and R⟨*source2*⟩ operand, and store the result into the destination location R⟨*dest*⟩.	

Condition Codes	`N Z V C`
	`* * 0 0`	

Examples	ORB R3 R1 R2	R3 ← R1 | R2
	ORB R3 R1 (R2)	R3 ← R1 | [R2]
	ORB (R3) R1 (R2)	[R3] ← R1 | [R2]

EORB Bitwise exclusive OR (XOR)

Operation	R⟨*dest*⟩ ← R⟨*source1*⟩ ˆ R⟨*source2*⟩	
Assembler Syntax	EORB R⟨*dest*⟩ R⟨*source1*⟩ R⟨*source2*⟩	
Opcode	‘0111’	
Description	EORB (exclusive or) the source operands R⟨*source1*⟩ and R⟨*source2*⟩, and store the result into the destination location R⟨*dest*⟩.	

Condition Codes	`N Z V C`
	`* * 0 0`	

Examples	EORB R3 R1 R2	R3 ← R1 ˆ R2
	EORB R3 R1 (R2)	R3 ← R1 ˆ [R2]
	EORB (R3) R1 (R2)	[R3] ← R1 ˆ [R2]

MUL Multiplication

Operation	$R\langle dest\rangle \leftarrow R\langle source1\rangle \times R\langle source2\rangle$
Assembler Syntax	MUL $R\langle dest\rangle$ $R\langle source1\rangle$ $R\langle source2\rangle$
Opcode	‘1000’
Description	Multiply the 32-bit $R\langle source1\rangle$ operand by the 32-bit $R\langle source2\rangle$ operand and store the result into the destination $R\langle dest\rangle$. Both the sources and destination are 32-bit word values.

Condition Codes	`N Z V C`
	`* * * 0`

Examples	MUL R3 R1 R2 — $R3 \leftarrow R1 \times R2$ MUL R3 R1 (R2) — $R3 \leftarrow R1 \times [R2]$ MUL (R3) R1 (R2) — $[R3] \leftarrow R1 \times [R2]$

DIV Division

Operation	$R\langle dest\rangle \leftarrow R\langle source1\rangle \div R\langle source2\rangle$
Assembler Syntax	DIV $R\langle dest\rangle$ $R\langle source1\rangle$ $R\langle source2\rangle$
Opcode	‘1001’
Description	Divide the 32-bit $R\langle source1\rangle$ operand by the 32-bit $R\langle source2\rangle$ operand and store the result into the destination $R\langle dest\rangle$. Both the sources and destination are 32-bit word values.

Condition Codes	`N Z V C`
	`* * * 0`

Examples	DIV R3 R1 R2 — $R3 \leftarrow R1 \div R2$ DIV R3 R1 (R2) — $R3 \leftarrow R1 \div [R2]$ DIV (R3) R1 (R2) — $[R3] \leftarrow R1 \div [R2]$

SHL Binary shift to the left

Operation	R⟨*dest*⟩ ← R⟨*source1*⟩ << R⟨*source2*⟩
Assembler Syntax	SHL R⟨*dest*⟩ R⟨*source1*⟩ R⟨*source2*⟩
Opcode	‘1010’
Description	Performs a binary ‘shift to left’ on the 32-bit R⟨*source1*⟩ operand. The number of bits shifted is contained into the 32-bit R⟨*source2*⟩ operand. The result of the shift operation is stored in the destination register R⟨*dest*⟩. The carry flag is set to the last bit shifted out.

Condition Codes	`N Z V C`
	 `* * 0 *`

Examples		
	SHL R3 R1 R2	R3 ← R1 << R2
	SHL R3 R1 (R2)	R3 ← R1 << [R2]
	SHL (R3) R1 (R2)	[R3] ← R1 << [R2]

SHR Binary shift to the right

Operation	R⟨*dest*⟩ ← R⟨*source1*⟩ >> R⟨*source2*⟩
Assembler Syntax	SHR R⟨*dest*⟩ R⟨*source1*⟩ R⟨*source2*⟩
Opcode	‘1011’
Description	Performs a binary ‘shift to right’ on the 32-bit R⟨*source1*⟩ operand. The number of bits shifted is contained in the 32-bit R⟨*source2*⟩ operand. The result of the shift operation is stored into the destination register R⟨*dest*⟩. The carry flag is set to the last bit shifted out.

Condition Codes	`N Z V C`
	 `* * 0 *`

Examples		
	SHR R3 R1 R2	R3 ← R1 >> R2
	SHR R3 R1 (R2)	R3 ← R1 >> [R2]
	SHR (R3) R1 (R2)	[R3] ← R1 >> [R2]

ROTL Binary rotation to the left

Operation	R⟨*dest*⟩ $\leftarrow$ R⟨*source1*⟩ $\circlearrowleft$ R⟨*source2*⟩
Assembler Syntax	ROTL R⟨*dest*⟩ R⟨*source1*⟩ R⟨*source2*⟩
Opcode	‘1100’
Description	Rotate the bits of the operand to the left. A rotate operation is circular in the sense that the bit shifted out at one end is shifted into the other end. That is, no bit is lost or destroyed by a rotate. After the operation, carry flag is set to the last bit rotated in.

Condition Codes	`N Z V C`
	`* * 0 *`

Examples		
	ROTL R3 R1 R2	R3 $\leftarrow$ R1 $\circlearrowleft$ R2
	ROTL R3 R1 (R2)	R3 $\leftarrow$ R1 $\circlearrowleft$ [R2]
	ROTL (R3) R1 (R2)	[R3] $\leftarrow$ R1 $\circlearrowleft$ [R2]

ROTR Binary rotation to the right

Operation	R⟨*dest*⟩ $\leftarrow$ R⟨*source1*⟩ $\circlearrowright$ R⟨*source2*⟩
Assembler Syntax	ROTR R⟨*dest*⟩ R⟨*source1*⟩ R⟨*source2*⟩
Opcode	‘1101’
Description	Rotate the bits of the operand to the right. A rotate operation is circular in the sense that the bit shifted out at one end is shifted into the other end. That is, no bit is lost or destroyed by a rotate. After the operation, carry flag is set to the last bit rotated in.

Condition Codes	`N Z V C`
	`* * 0 *`

Examples		
	ROTR R3 R1 R2	R3 $\leftarrow$ R1 $\circlearrowright$ R2
	ROTR R3 R1 (R2)	R3 $\leftarrow$ R1 $\circlearrowright$ [R2]
	ROTR (R3) R1 (R2)	[R3] $\leftarrow$ R1 $\circlearrowright$ [R2]

NEG Arithmetic negation (two's complement)

Operation	R⟨*dest*⟩ $\leftarrow$ 0 − R⟨*source2*⟩
Assembler Syntax	NEG R⟨*dest*⟩ R⟨*source1*⟩ R⟨*source2*⟩
Opcode	'1110'
Description	Negate the value of R⟨*source2*⟩ and store the result into R⟨*dest*⟩.

Condition Codes	`N Z V C`
	`* * * *`

Note	R⟨*source1*⟩ is unused.
Examples	NEG R3 R1 R2 — R3 $\leftarrow$ 0 − R2 NEG R3 R1 (R2) — R3 $\leftarrow$ 0 − [R2] NEG (R3) R1 (R2) — [R3] $\leftarrow$ 0 − [R2]

SPCL Special opcode

Operation	Currently undefined.
Assembler Syntax	SPCL R⟨*dest*⟩ R⟨*source1*⟩ R⟨*source2*⟩
Opcode	'1111'
Description	Reserved for future expansion.

Condition Codes	`N Z V C`
	`? ? ? ?`

B.3. Binary Instructions

Binary instructions are instructions with two register operands, although they generally have three operands. All register operands of binary instructions are always *directly* addressed. Most binary instructions have an immediate operand, which is a signed 16-bit integer.

ADDI Add with immediate operand

Operation	$R\langle dest\rangle \leftarrow R\langle source\rangle + \langle imm\rangle$
Assembler Syntax	ADDI $R\langle dest\rangle$ $R\langle source\rangle$ $\#\langle imm\rangle$
Opcode	‘0000’
Description	Add the source operand $R\langle source\rangle$ to the immediate value and store the result into the destination location $R\langle dest\rangle$.

Condition Codes	`N Z V C`
	`* * * *`

Examples	ADDI R2 R1 #1 R2 ← R1 + 1

SUBI Subtract with immediate operand

Operation	$R\langle dest\rangle \leftarrow R\langle source\rangle - \langle imm\rangle$
Assembler Syntax	SUBI $R\langle dest\rangle$ $R\langle source\rangle$ $\#\langle imm\rangle$
Opcode	‘0001’
Description	Subtract the source operand $R\langle source\rangle$ from immediate value and store the result into the destination location $R\langle dest\rangle$.

Condition Codes	`N Z V C`
	`* * * *`

Examples	SUBI R2 R1 #1 R2 ← R1 - 1

ANDLI Logical AND with immediate operand

Operation	$R\langle dest\rangle \leftarrow R\langle source\rangle \wedge \langle imm\rangle$
Assembler Syntax	ANDLI $R\langle dest\rangle$ $R\langle source\rangle$ $\#\langle imm\rangle$
Opcode	‘0010’
Description	Performs a logical AND between the source operand $R\langle source\rangle$ and the immediate operand and store the result into the destination location $R\langle dest\rangle$.

Condition Codes	`N Z V C`
	`* * 0 0`

Examples	ANDLI R2 R1 #1 R2 ← R1 $\wedge$ 1

ORLI Logical OR with immediate operand

Operation	$R\langle dest\rangle \leftarrow R\langle source\rangle \vee \langle imm\rangle$
Assembler Syntax	ORLI $R\langle dest\rangle$ $R\langle source\rangle$ #$\langle imm\rangle$
Opcode	‘0011’
Description	Performs an OR between the source operand $R\langle source\rangle$ and the immediate operand and store the result into the destination location $R\langle dest\rangle$.

Condition Codes	`N Z V C`
	`* * 0 0`

Examples	ORLI R2 R1 #1 $R2 \leftarrow R1 \vee 1$

EORLI Logical Exclusive OR with immediate operand

Operation	$R\langle dest\rangle \leftarrow R\langle source\rangle \oplus \langle imm\rangle$
Assembler Syntax	EORLI $R\langle dest\rangle$ $R\langle source\rangle$ #$\langle imm\rangle$
Opcode	‘0100’
Description	EOR (exclusive or) the source operand $R\langle source\rangle$ with immediate and store the result into the destination location $R\langle dest\rangle$.

Condition Codes	`N Z V C`
	`* * 0 0`

Examples	EORLI R2 R1 #1 $R2 \leftarrow R1 \oplus 1$

ANDBI Bitwise AND with immediate operand

Operation	$R\langle dest\rangle \leftarrow R\langle source\rangle \,\&\, \langle imm\rangle$
Assembler Syntax	ANDBI $R\langle dest\rangle$ $R\langle source\rangle$ #$\langle imm\rangle$
Opcode	‘0101’
Description	ANDBI the source operand $R\langle source\rangle$ to the immediate operand and store the result into the destination location $R\langle dest\rangle$.

Condition Codes	`N Z V C`
	`* * 0 0`

Examples	ANDBI R2 R1 #345 $R2 \leftarrow R1 \,\&\, 345$

ORBI Bitwise OR with immediate operand

Operation	$R\langle dest\rangle \leftarrow R\langle source\rangle \mid \langle imm\rangle$
Assembler Syntax	ORBI R$\langle dest\rangle$ R$\langle source\rangle$ #$\langle imm\rangle$
Opcode	‘0110’
Description	ORBI the source operand R$\langle source\rangle$ to the immediate operand, and store the result into the destination location R$\langle dest\rangle$.

Condition Codes	`N Z V C`
	`* * 0 0`

Examples	ORBI R2 R1 #345 $R2 \leftarrow R1 \mid 345$

EORBI Bitwise exclusive OR with immediate operand

Operation	$R\langle dest\rangle \leftarrow R\langle source\rangle$ ^ $\langle imm\rangle$
Assembler Syntax	EORBI R$\langle dest\rangle$ R$\langle source\rangle$ #$\langle imm\rangle$
Opcode	‘0111’
Description	EORBI (exclusive or) the source operand R$\langle source\rangle$ with the immediate operand and store the result into the destination location R$\langle dest\rangle$.

Condition Codes	`N Z V C`
	`* * 0 0`

Examples	EORBI R2 R1 #345 $R2 \leftarrow R1$ ^ 345

MULI Multiply with immediate operand

Operation	$R\langle dest\rangle \leftarrow R\langle source\rangle \times \langle imm\rangle$
Assembler Syntax	MULI R$\langle dest\rangle$ R$\langle source\rangle$ #$\langle imm\rangle$
Opcode	‘1000’
Description	Multiply the 32-bit R$\langle source\rangle$ operand by the immediate operand and store the result into the destination R$\langle dest\rangle$. Both the source and destination registers are 32-bit, but the immediate operand value is always 16-bit.

Condition Codes	`N Z V C`
	`* * * 0`

Examples	MULI R2 R1 #345 $R2 \leftarrow R1 \times 345$

DIVI Divide with immediate operand

Operation	R⟨*dest*⟩ ← R⟨*source*⟩ ÷ ⟨*imm*⟩
Assembler Syntax	DIVI R⟨*dest*⟩ R⟨*source*⟩ #⟨*imm*⟩
Opcode	'1001'
Description	Divide the 32-bit R⟨*source*⟩ operand by immediate operand and store the result into the destination R⟨*dest*⟩. Both the source and destination are 32-bit word values exception made for the immediate value which is always 16-bit long.

Condition Codes	`N Z V C`
	`* * * 0`

Examples	DIVI R2 R1 #345 — R2 ← R1 ÷ 345

SHLI Binary shift to the left with immediate operand

Operation	R⟨*dest*⟩ ← R⟨*source*⟩ << ⟨*imm*⟩
Assembler Syntax	SHLI R⟨*dest*⟩ R⟨*source*⟩ #⟨*imm*⟩
Opcode	'1010'
Description	SHL Performs a binary 'shift to left' on the 32-bit R⟨*source*⟩ operand. The number of bits shifted is given by the value of the immediate operand. The result of the shift operation is stored into the destination register R⟨*dest*⟩. The carry flag is set to the last bit shifted out.

Condition Codes	`N Z V C`
	`* * * *`

Examples	SHLI R2 R1 #3 — R2 ← R1 << 3

SHRI Binary shift to the right with immediate operand

Operation	R⟨*dest*⟩ ← R⟨*source*⟩ >> ⟨*imm*⟩
Assembler Syntax	SHRI R⟨*dest*⟩ R⟨*source*⟩ #⟨*imm*⟩
Opcode	'1011'
Description	SHR Performs a binary 'shift to right' on the 32-bit R⟨*source*⟩ operand. The number of bits shifted is given by the value of the immediate operand. The result of the shift operation is stored into the destination register R⟨*dest*⟩. The carry flag is set to the last bit shifted out.

Condition Codes	`N Z V C`
	`* * * *`

Examples	SHRI R2 R1 #3 — R2 ← R1 >> 3

ROTLI Binary rotation to the left with immediate operand

Operation	R⟨*dest*⟩ $\leftarrow$ R⟨*source*⟩ $\circlearrowleft$ ⟨*imm*⟩
Assembler Syntax	ROTLI R⟨*dest*⟩ R⟨*source*⟩ #⟨*imm*⟩
Opcode	‘1100’
Description	Rotate the bits of the operand to the left. A rotate operation is circular in the sense that the bit shifted out at one end is shifted into the other end. That is, no bit is lost or destroyed by a rotate. After the operation, the carry flag is set to the last bit rotated in.

Condition Codes	`N Z V C`
	`* * 0 *`

Examples	ROTLI R2 R1 #3 R2 $\leftarrow$ R1 $\circlearrowleft$ 3

ROTRI Binary rotation to the right with immediate operand

Operation	R⟨*dest*⟩ $\leftarrow$ R⟨*source*⟩ $\circlearrowright$ ⟨*imm*⟩
Assembler Syntax	ROTRI R⟨*dest*⟩ R⟨*source*⟩ #⟨*imm*⟩
Opcode	‘1101’
Description	Rotate the bits of the operand to the right. A rotate operation is circular in the sense that the bit shifted out at one end is shifted into the other end. That is, no bit is lost or destroyed by a rotate. After the operation, the carry flag is set to the last bit rotated in.

Condition Codes	`N Z V C`
	`* * 0 *`

Examples	ROTRI R2 R1 #3 R2 $\leftarrow$ R1 $\circlearrowright$ 3

NOTL Logical negation

Operation	R⟨*dest*⟩ $\leftarrow \neg$ R⟨*source*⟩
Assembler Syntax	NOT R⟨*dest*⟩ R⟨*source*⟩ #⟨*imm*⟩
Opcode	‘1110’
Description	Perform a logical NOT operation on the value of R⟨*source*⟩. The result is stored into R⟨*dest*⟩.

Condition Codes	`N Z V C`
	`* * 0 0`

Note	The third operand is ignored.
Examples	NOTL R2 R1 #0 R2 $\leftarrow \neg$ R1

NOTB Bitwise negation (one's complement)

Operation	R⟨*dest*⟩ $\leftarrow \sim$ R⟨*source*⟩
Assembler Syntax	NOTB R⟨*dest*⟩ R⟨*source*⟩ #⟨*imm*⟩
Opcode	'1111'
Description	Perform a binary NOT operation on the value of R⟨*source*⟩. The result is stored into R⟨*dest*⟩.

Condition Codes	`N Z V C`
	`* * 0 0`

Note	The third operand is ignored.
Examples	NOTB R2 R1 #0 R2 $\leftarrow \sim$ R1

B.4. Unary Instructions

Unary instructions are instructions with one *register* operand, although some of them have two operands. Given their small number, also instructions with no operands have been put in this category. All register operands are always *directly* addressed.

NOP No operation

Operation	—
Assembler Syntax	NOP
Opcode	‘0000’
Description	No operation performed. This instruction has no effect on the machine internal state

Condition Codes	`N Z V C`
	`- - - -`

MOVA Move address to register

Operation	$R\langle dest\rangle \leftarrow \langle address\rangle$	
Assembler Syntax	MOVA $R\langle dest\rangle$ $\langle address\|label\rangle$	
Opcode	‘0001’	
Description	Move the specified 20-bit absolute address into $R\langle dest\rangle$.	

Condition Codes	`N Z V C`
	`- - - -`	

Examples	MOVA R2 L1	R2 $\leftarrow$ L1 (where L1 is a label)

JSR Jump to subroutine

Operation	$R\langle dest\rangle \leftarrow R\langle dest\rangle - 1;$ $[R\langle dest\rangle] \leftarrow PC;$ $PC \leftarrow \langle address\rangle$	
Assembler Syntax	JSR $R\langle dest\rangle$ $\langle address\|label\rangle$	
Opcode	‘0010’	
Description	Jumps to the subroutine at the specified address, pushing the return address to the stack pointed to by $R\langle dest\rangle$.	

Condition Codes	`N Z V C`
	`- - - -`	

Examples	JSR R31 L1	Jumps to the subroutine starting at label L1, using R31 as the stack pointer.

RET Return from subroutine

Operation	PC $\leftarrow$ [R$\langle dest\rangle$]; R$\langle dest\rangle \leftarrow$ R$\langle dest\rangle + 1$
Assembler Syntax	RET R$\langle dest\rangle$ $\langle address\|label\rangle$
Opcode	'0011'
Description	Returns from a subroutine, popping the return address from the stack pointed to by R$\langle dest\rangle$.

Condition Codes	`N Z V C`
	`- - - -`

Note	The address parameter is unused.
Examples	RET R31 0 — Returns from a subroutine using R31 as the stack pointer.

LOAD Load a register value from memory

Operation	R$\langle dest\rangle \leftarrow$ [$\langle address\rangle$]
Assembler Syntax	LOAD R$\langle dest\rangle$ $\langle address\|label\rangle$
Opcode	'0100'
Description	Load the value previously stored at the specified memory location inside the register R$\langle dest\rangle$

Condition Codes	`N Z V C`
	`- - - -`

Examples	LOAD R2 L1 — R2 $\leftarrow$ [L1] (where L1 is a Label)

STORE Store a register value to memory

Operation	[$\langle address\rangle$] $\leftarrow$ R$\langle dest\rangle$
Assembler Syntax	STORE R$\langle dest\rangle$ $\langle address\|label\rangle$
Opcode	'0101'
Description	Store the value of R$\langle dest\rangle$ to the specified memory location.

Condition Codes	`N Z V C`
	`- - - -`

Note	In this particular instruction, R$\langle dest\rangle$ takes the role of a source register instead of a destination register.
Examples	STORE R2 L1 — [L1] $\leftarrow$ R2 (where L1 is a Label)

HALT Halt the machine

Operation	—
Assembler Syntax	HALT
Opcode	‘0110’
Description	Halts the machine processor, or terminates the simulation.
Condition Codes	N Z V C - - - -

Scc Set according to condition ‘cc’

Operation

```
IF cc = 1 THEN
    R⟨dest⟩ ← 1;
ELSE
    R⟨dest⟩ ← 0.
```

Assembler Syntax Scc R⟨*dest*⟩ ⟨*address*|*label*⟩

Opcodes

SEQ	‘0111’	Set on equal
SGE	‘1000’	Set on greater than or equal
SGT	‘1001’	Set on greater than
SLE	‘1010’	Set on less than or equal
SLT	‘1011’	Set on less than
SNE	‘1100’	Set on not equal

Description The specified conditional ‘cc’ is tested. If the condition is true, R⟨*dest*⟩ is set to one; Otherwise R⟨*dest*⟩ is set to zero. Details on the evaluation of ‘cc’ are found in appendix B.1.

Condition Codes

```
N Z V C
0 * 0 0
```

Note Address parameter is unused.

Examples

SGT R2 0	set the value of R2 to 1 if the condition GT is verified; 0 otherwise.

READ Read an integer value from standard input

Operation	R$\langle$*dest*$\rangle \leftarrow$ [User Input]
Assembler Syntax	READ R$\langle$*dest*$\rangle$ $\langle$*address*$\vert$*label*$\rangle$
Opcode	‘1101’
Description	This instruction loads a signed 32-bit value from standard input into a register. In the MACE simulator the READ instruction is implemented with the *scanf* C library function.

Condition Codes	`N Z V C`
	`– – – –`

Note	Address parameter is unused.
Examples	READ R2 0 — Read a 32-bit signed integer value from standard input, and store it into ‘R2’.

WRITE Write an integer value to standard output

Operation	[User Output] $\leftarrow$ R$\langle$*dest*$\rangle$
Assembler Syntax	WRITE R$\langle$*dest*$\rangle$ $\langle$*address*$\vert$*label*$\rangle$
Opcode	‘1110’
Description	This instruction writes a 32-bit signed value to standard output. In the MACE simulator the WRITE instruction is implemented with the *printf* C library function.

Condition Codes	`N Z V C`
	`– – – –`

Note	Address parameter is unused. In this particular instruction, R$\langle$*dest*$\rangle$ takes the role of a source register instead of a destination register.
Examples	WRITE R2 0 — Write the 32-bit signed integer value stored in R2 to standard output.

XPSW Exchange values between PSW and register

Operation	R$\langle$*dest*$\rangle$ $\leftrightarrows$ PSW
Assembler Syntax	XPSW R$\langle$*dest*$\rangle$ $\langle$*address*|*label*$\rangle$
Opcode	‘1111’
Description	The previous value of the PSW register is stored in R$\langle$*dest*$\rangle$, and the previous value of R$\langle$*dest*$\rangle$ is stored in the PSW.

Condition Codes	`N Z V C`
	`* * * *`

Note	Address parameter is unused. This instruction allows to set PSW to values that are otherwise impossible, but it does not allow setting the unused bits of the PSW to any value other than zero.

B.5. Jump Instructions

Jump instructions permit the execution of instructions in non-sequential order.

Bcc Branch on condition cc

Operation IF cc = 1 THEN
PC ← PC + ⟨*address*⟩ − 1.
(The decrement by 1 is required because, after each instruction, the PC is automatically incremented by 1.)

Assembler Syntax Bcc ⟨*address*|*label*⟩

Opcode

BT	‘0000’	Branch always
BF	‘0001’	Branch never
BHI	‘0010’	Branch on unsigned higher than
BLS	‘0011’	Branch on unsigned lower than or same
BCC	‘0100’	Branch on carry clear
BCS	‘0101’	Branch on carry set
BNE	‘0110’	Branch on not equal
BEQ	‘0111’	Branch on equal
BVC	‘1000’	Branch on overflow clear
BVS	‘1001’	Branch on overflow set
BPL	‘1010’	Branch on plus (i.e. positive)
BMI	‘1011’	Branch on minus (i.e. negative)
BGE	‘1100’	Branch on greater than or equal
BLT	‘1101’	Branch on less than
BGT	‘1110’	Branch on greater than
BLE	‘1111’	Branch on less than or equal

Description The specified condition code ‘cc’ is tested. If the condition is true, the program counter will be modified in order to point to a specific labeled instruction. Details on the evaluation of ‘cc’ are found in appendix B.1.

Condition Codes

N	Z	V	C
–	–	–	–

Note The address field of the instruction encoding is interpreted as a signed displacement between the current PC and the new PC. In the assembler, explicitly specified addresses are encoded without modifications. When a label is specified, the distance between the branch instruction and the label is computed automatically.

Examples

BEQ L1	Branch to L1 on “equal to zero”
BT 0	Infinite loop
BT 1	Jump to the next instruction (same as NOP)
BT 5	Jump over the next 4 instructions
BT -1	Jump to the previous instruction