Advanced Compiler System for Education

Politecnico di Milano - DEI
Formal Languages and Compiler Group
Andrea Di Biagio and Giovanni Agosta

December 23, 2008

Contents

Chapter 1

Acse

ACSE (Advanced Compiler System for Education) is a simple compiler developed for educational purpose as a tool for the course "Formal languages and compiler". ACSE is able to translate a source code written in LanCE 1.3 (Language for Compilers Education) into an assembly for the MACE architecture (see the MACE documentation in Chapter 3). Figure 1.1 shows the components mentioned above and their interaction.

The main goal of this documentation is to show the compilation process by introducing every single step made by the compiler. Also, in the following sections there is a brief discussion of the grammar for a source code written in LanCE that can be given as input to the ACSE compiler. Finally, a brief introduction is given for each module and library used by ACSE.

1.1 Compilation process

In figure 1.2 the entire compilation process is shown. Every phase of the compilation chain takes as input the output of the previous phase. ACSE takes as input a source file written in LanCE language (see 1.3).

The compilation process is depicted in figure 1.2. The source file is analyzed and 'tokenized' (i.e., subdivided into tokens) by a lexer in the first phase of the compilation process. The string of tokens is then processed by the parser in order to check and analyze the overall structure of the source program.

ACSE uses a parser automatically generated with Bison. Bison is a general-purpose parser generator that converts an annotated context-free grammar into an LALR(1) deterministic bottom-up parser for the given grammar.

At first the SDT (Syntax-Direct Translator) analyzes the correctness of a tokenized input provided by the lexer; then it executes some semantic

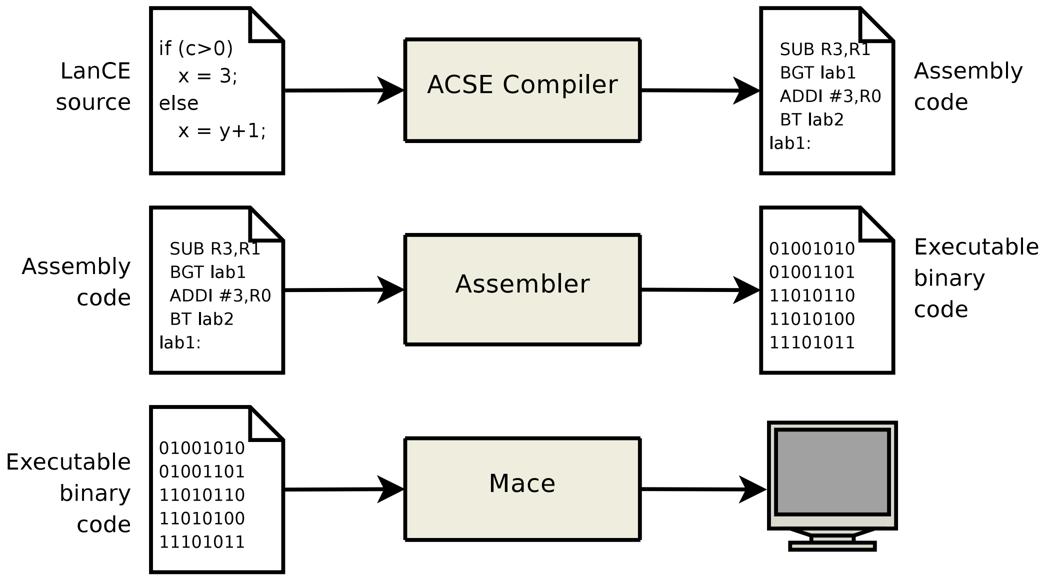

Figure 1.1: Overview of the ACSE tools

actions (if any) in correspondence of each recognized grammatical rule. The current parser implementation provides support for both error tracking and notification of simple warning messages.

During the parsing process, the tokenized input is transformed into specific assembly statements for the target machine (`MACE`). However at this point of the compilation process, the assembly produced as output uses an unbounded number of registers. Since the target machine has a limited set of general-purpose registers, liveness analysis and register allocation steps are then performed.

Finally the assembly code is written as output in the last phase of the

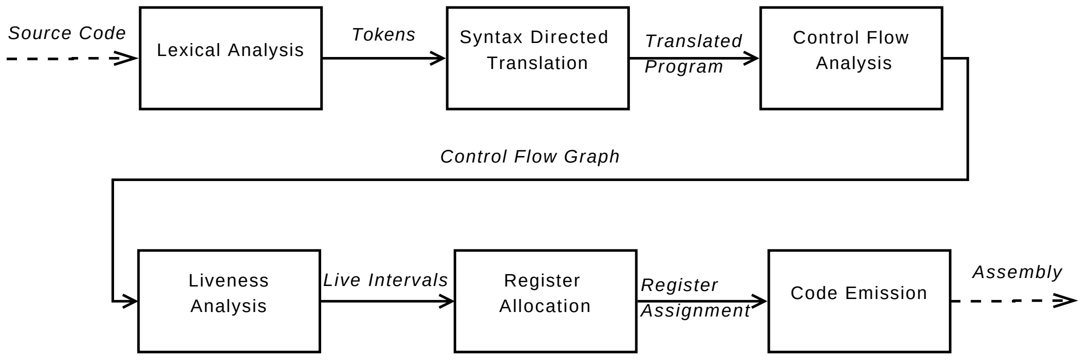

Figure 1.2: Compilation process

compilation process.

1.2 Tokens for `LanCE`

As previously said, a lexer component scans the input source code in search of specific patterns of strings. These patterns are defined via regular expressions and are used in order to identify lexical tokens (or lexemes). Regular expressions are coded in the form of `FLEX` rules (see the `FLEX` documentation).

`ACSE` delegates the lexical analysis to a 'lexer' automatically generated by `FLEX`.

The following is a list of bindings between tokens and regular expressions.

Token	Regular Expression	Token	Regular Expression
DIGIT	[0-9]	ID	[a-zA-Z_][a-zA-Z0-9_]*
LBRACE	{	RBRACE	}
LSQUARE	[	RSQUARE	]
LPAR	(	RPAR	)
SEMI	;	COLON	:
PLUS	+	MINUS	-
MUL_OP	*	DIV_OP	/
MOD_OP	%	AND_OP	&
OR_OP	|	NOT_OP	!
ASSIGN	=	LT	<
GT	>	SHL_OP	<<
SHR_OP	>>	EQ	==
NOTEQ	!=	LTEQ	<=
GTEQ	>=	ANDAND	&&
OROR	||	COMMA	,
DO	do	ELSE	else
FOR	for	IF	if
TYPE	int	WHILE	while
RETURN	return	READ	read
WRITE	write		

1.3 A grammar for `LanCE`

In this section a grammar for the source code is provided.

program :	*var_declarations statements*
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*
	| *do_while_statement* SEMI
	| *while_statement*
	| *return_statement* SEMI
read_write_statement :	*read_statement*
	| *write_statement*
assign_statement :	IDENTIFIER LSQUARE *exp* RSQUARE ASSIGN *exp*
	| IDENTIFIER ASSIGN *exp*
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 NUMBER
       | NOT_OP IDENTIFIER
       | 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
```

Using the parser that Bison generated from this grammar, the syntax analyzer is able to check that a token string is syntactically correct.

1.4 Example of Source Code

An example of a program — compliant with the grammar shown in the previous section — that computes the factorial of an integer number and prints the result to standard output is shown in table 1.1.

If the number given as input is negative, the program writes '-1' to standard output and then exits.

1.5 Semantic actions

The parser tests the correctness of each statement found in the source program and eventually performs semantic actions. In a Bison grammar, a

```
int value, fact;        /* variables 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 */
```

Table 1.1: An example of source code

grammar rule can have an action made up of C statements. Each time the parser recognizes a match for that rule, the action is executed.

If the statement is syntactically correct, a semantic action (a Bison rule) is executed every time a non-terminal, associated with a semantic action, is recognized and reduced.

The main goal of the ACSE SDT is to gather all the useful information about each statement of the source program (the source code given as input). Each instruction is typically translated into one or more assembly instructions. All assembly instructions are stored inside a data structure called `t_program_infos`, which is defined in `axe_engine.h`.

1.6 Program information

An instance of `t_program_infos` contains all the useful information about a program being compiled.

A `t_program_infos` is a composition of various elements:

- An instance of a symbol table
- An instance of a label manager
- A list of program variables

- A list of instructions and assembler directives.

Before starting with the lexical analysis and the syntactical analysis, the compiler initializes an instance of `t_program_infos` called `program`.

The source program is translated (by executing various semantic actions) into various assembly instructions and assembler directives that are orderly stored inside the `program` instance.

Data structures for assembly instructions and assembler directives are defined in the file `axe_struct.h`.

An assembly instruction is described by:

- An operation identifier (for example, 'SUB');
- A set of instruction parameters which depend on the instruction type;
- A user comment (optional);
- A label identifier (optional).

Valid instruction parameters are:

- Register identifiers;
- Immediate values (signed integer values);
- Addresses (for example, label identifiers).

Example: an assembly ADD instruction is a ternary instruction that performs a binary sum of two values. An instruction that performs the sum of registers `R1` and `R2`, and stores the result in register `R3` can be declared as follows: `ADD R3 R1 R2`.

The keyword `ADD` is used as an operation identifier. The number and type of instruction parameters depends only on the instruction type. For example, the ternary instruction ADD accepts only register identifiers as parameters.

In the previous example, the parameters `R3`, `R1` and `R2` are all register identifiers. A register identifier is an alias for a machine general-purpose register.

A user comment can be associated to an assembly instruction for debugging purpose. A label can be associated to an assembly instruction as shown in the example below:

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

In this example, the label 'L1' is associated to the assembly ADD instruction.

An assembler directive is defined as follows:

- Directive type identifier (for example, `.WORD`);
- Value associated with the directive;
- A label identifier (optional).

Here is an example of an assembler directive .WORD: `.WORD 0`

More information about assembly instructions and assembler directives can be found in the Assembler documentation.

1.7 Source Program 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.

Structure `t_axe_variable` defines:

- A data type (for example, INTEGER);
- An Array Size (defined only if the variable is an array);
- An initial value;
- A variable identifier;

Here is an example of how it's possible to declare in a Source Program a variable called `var` as an integer with an initial value of '100':

`int var = 100;`

An instance of `t_axe_variable` for the program variable "var" would be defined in the following manner: `INTEGER` as the data type; `100` as the initial value; the string `"var"` as the variable identifier.

All the instances `t_axe_variable` are used at the end of the compilation process in order to produce `.WORD` and/or `.SPACE` assembler directives (see the MACE documentation in Chapter 3).

In our example, the `t_axe_variable` defined for the program variable `var` will produce a `.WORD 100` data directive.

1.8 Symbol Table

A symbol table is a data structure used at translation time to keep track of each program variable encountered in the source program. Each entry of the symbol table is associated to a single program variable, and it is defined in the following manner:

Source Program		Assembly	Comments
int value, fact;		.DATA	variables declarations
	L0:	.WORD 0	initialize 4 bytes of data to 0
	L1:	.WORD 0	initialize 4 bytes of data to 0
read(value);		.TEXT	start of a block of code
		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 × 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

Table 1.2: Intermediate Assembly representation

- ID - A variable identifier (see section 1.7)
- Type - The data type of the variable ‘ID’ (see section 1.7)
- A Register identifier.

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

1.9 Example

After parsing, the program in table 1.1 produces the intermediate assembly shown in table 1.2.

The content of the symbol table is shown in table 1.3.

Variable Identifier	Type	Register Location
value	INTEGER	R1
fact	INTEGER	R2

Table 1.3: Symbol Table

1.10 API documentation

The main purpose of this section is to give an overview of the essential programming interfaces of ACSE. Since the ACSE design defines several functions over a number of header files, the user is also referred to the commented source code.

The file `symbol_table.h` declares the functions needed to manipulate the content of a symbol table:

- look up a symbol in a symbol table
- define and insert a new symbol into a symbol table
- set the register location information of a symbol
- retrieve the register location information associated with a symbol.

The `symbol_table.h` includes a file called `sy_table_constants.h` which defines a set of macros used for error tracking.

The file `axe_struct.h` defines many data structures used by the parser. The parser is automatically generated by running Bison with the file `Acse.y` as input.

The `Label Manager` is defined in `axe_labels.h`. A Label Manger offers a set of functions to work with labels:

- `newLabelID`
- `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.

The library `axe_gencode.h` defines a set of functions that can be directly used to generate assembly instructions and insert them into an instance of `t_program_infos`.

For example, the function `gen_add_instruction` is used to create an ADD instruction.

The file `axe_engine.h` defines the `t_program_infos` data structure and other useful functions used (for example) to:

- initialize a new instance of `t_program_infos`
- add an assembly instruction to a `t_program_infos` instance
- create a variable (i.e., an instance of `t_axe_variable`) and assign it to an instance of `t_program_infos`
- request a free register location (see the function `getNewRegister`)
- write an assembly file as output of the compilation process (see the function `writeAssembly`).

The file `axe_array.h` provides a set of functions used to generate load and store instructions from/to array elements. An example is the function `loadArrayElement`, which takes as input:

- an array variable identifier
- an array subscript identifying an array element.

This function generates instructions that load the content of the given element of the given array in a register, and it returns the location identifier for the register that will hold the value of the specified array element.

According to the Bison grammar defined in `Acse.y`, an expression is defined by the non-terminal `exp` (see the section 1.3). The file `axe_structs.h` defines an expression as an instance of `t_axe_expression`. An instance of `t_axe_expression` contains two fields:

- A value (a register identifier or an immediate value)
- An expression type.

The expression type is used to determine if the value of the expression is stored into a register or is an immediate value. The file `axe_expressions.h` defines two functions used to generate instructions for expressions:

- handle_binary_comparison;

- handle_bin_numeric_op.

The function `handle_bin_numeric_op` is used to generate instructions for binary numeric operations (for example, the expression "a + 5"), while `handle_binary_comparison` is used to generate instructions that perform a comparison between two values. These functions take as input two expressions (the two operands of the binary comparison/operation) and return an instance of `t_axe_expression` (containing the reference to the register holding the result of the binary comparison/operation).

Finally, the file `axe_utils.h` defines the function `get_symbol_location` that works as a wrapper for the functions defined in `symbol_table.h`. Given a variable/symbol identifier as input, `get_symbol_location` goes through the symbol table in search of the register location where the symbol is stored. If the requested variable/symbol has never been loaded from memory to a register before (i.e., the symbol contains an invalid register location info), this function searches for a free register where to store the variable/symbol. At last, the register location (where the variable/symbol is stored) is returned as output to the caller.

1.11 Examples of Bison semantic actions

In this section we discuss three examples of bison semantic actions. The first and the second examples describe how to manipulate expressions. The third example describes how to define semantic actions for a `do-while` statement.

1.11.1 Expressions

Section 1.10 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.

Here we will discuss the semantic rules associated with the following Bison rules:

`exp` :
    | *exp* AND_OP *exp*
    | *exp* LT *exp*

According to the FLEX source file `Acse.lex`, an AND_OP token represents a 'binary and' operator (`&`). An example of 'binary and' operation is the following: `a & b` (where `a` and `b` are variables declared in the source program).

Suppose in this example that both a and b are program variables whose values are stored into unknown register locations. First we have to query the symbol table in order to retrieve the register locations associated with both a and b. This can be done by calling the function get_symbol_location declared in axe_utils.h for each of the two program variables.

Then we can use the function gen_andb_instruction, which is declared in axe_gencode.h, in order to generate an assembly ANDB instruction. A call to gen_andb_instruction requires five parameters:

- An instance of t_program_infos that contains all the information about the program being compiled;
- A destination register identifier;
- Two source register identifiers as parameters for the ANDB instruction;
- A field that is used to specify the addressing mode for both the destination register and the second source register. This field can assume one of the following values (defined in axe_constants.h):
  - CG_DIRECT_ALL
  - CG_INDIRECT_ALL
  - CG_INDIRECT_DEST
  - 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 address mode of the first source register is always direct.

If we want to store the result of the ANDB instruction in a new register (i.e., a register that is unused), we must first call the function getNewRegister declared in axe_engine.h, which returns a new register identifier as output. Then, we are able to call the gen_andb_instruction.

However, according to the Bison grammar, the result must be stored in an instance of t_axe_expression. Thus, in our example we have to create an instance of t_axe_expression by calling the function create_expression defined in axe_struct.h.

Actually, the function handle_bin_numeric_op is used to perform all the operations discussed so far in this section.

The function `handle_bin_numeric_op` takes as input the following parameters:

- An instance of `t_program_infos` that contains all the information about the program being compiled;
- Two instances of `t_axe_expression` (one for each of the two operands);
- An operation identifier. Valid binary operation identifiers (defined in `axe_constants.h`) are:
  - `ADD`
  - `ANDB`
  - `ORB`
  - `SUB`
  - `SHL`
  - `SHR`
  - `MUL`
  - `DIV`

  For the previous example, the macro `ANDB` should be used as identifier.

The function `handle_binary_comparison` can be used to implement the semantic action for the rule
`exp` : | *exp* LT *exp*
The function `handle_binary_comparison` takes as input the following parameters:

- An instance of `t_program_infos` that contains all the information about the program being compiled;
- Two instances of `t_axe_expression` (one for each of the two operands);
- A condition code. Valid condition codes (defined in `axe_constants.h`) are:
  - `_LT_` (i.e., “less than zero”);
  - `_GT_` (i.e., “greater than zero”);
  - `_EQ_` (i.e., “equal to zero”);
  - `_NOTEQ_` (i.e., “not equal to zero”);

– `_LTEQ_` (i.e., "less than or equal to zero");
– `_GTEQ_` (i.e., "greater than or equal to zero").

For the latest example, the macro `_LT_` should be used.

Both functions `handle_bin_numeric_op` and `handle_binary_comparison` return as output an instance of `t_axe_expression`.

1.11.2 do-while statement

A Bison semantic action for a *do-while* statement can be formalized as follows:

```
do_while_statements  : 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);
                     };
```

In order to implement a *do-while* statement, 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. An expression is used to formalize the loop termination condition. In the given example, if the outcome of `exp` 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.

We can use the function `newLabel` declared in `axe_engine.h` to ask the Label Manager (associated with a specific instance of `t_program_infos`) for a new label identifier. A label is an instance of `t_axe_label` (a structure declared in `axe_stuct.h` whose only field is an integer value).

The function `assignLabel` declared in `axe_engine.h` takes as input an instance of `t_program_infos` and an instance of `t_axe_label`. It assigns the given label to the next assembly statement that will be generated. We can use `assignLabel` to assign a label to the first instruction of the loop body by calling the function before `code_block` is analyzed by the parser.

Note that both the function `newLabel` and the function `assignLabel` are defined as wrappers respectively for the `newLabelID` and `assignLabelID` (declared in `axe_labels.h`).

Finally, we use a `gen_bne_instruction` to generate a conditional branch instruction to the first instruction of the loop body.

Chapter 2

Assembler

This manual describe how an assembler should translate an assembly produced by an `AXE` compiler into a valid executable (an object file) for a `MACE` architecture. If you want to learn more about the `MACE` internal architecture, see the `MACE` documentation (Chapter 3). The assembler takes an assembly as input (i.e., a symbolic assembly program — the output of a compilation process) containing both instructions and assembler directives. Each assembly instruction is directly translated into a specific machine code statement. Every statement is encoded according to the "binary format rules" discussed in the `MACE` documentation. Assembler directives are interpreted and discarded without producing any machine code instructions. The output of the translation process is an object file containing both machine code instructions and data information.

The first section describes how an assembler program works (the translation steps performed and their order). The second section introduces the assembly language and the assembler directives supported by the current implementation of the assembler. The last section describes the internal structure of an object file and its binary format.

2.1 How the Assembler works

According to the theory, an assembler is a program that translates an assembly into an object file for a specific architecture. The binary format of an object file typically depends on both the underlying architecture and the Operative 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 purpose).

In general, an assembly instruction is directly mapped in an equivalent machine code instruction. However, an assembly contains also *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 in 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. As post-condition, only valid assembly files are translated in "well formed" object files.

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

- The assembler initializes its internal data structures.
- It parses the assembly given as input and validates every instruction/directive.
- It writes an object file as output 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. These information will be used then in the last phase in order to produce a valid object file.

The second phase consists in 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.

2.2 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 as follows:

```
.DATA
Data directive no. 1
Data directive no. 2
...
Data directive no. n
.TEXT
First instruction
Second instruction
...
Last instruction
```

The assembler directive `.DATA` marks the beginning of a data section; within that section space for data and variables can be allocated (see Section 2.2.5). The directive `.TEXT` marks the beginning of the code section, which contains assembly instructions. This must be always the last section in an assembly file.

This is the format of a directive or instruction line:

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

Both label and comment are optional (for this reason, they are surrounded by square brackets). A label can be referenced as an operand of some instructions (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;
- address values (typically labels).

Unary and binary instructions use always *direct* addressing, while ternary instructions can use both direct and indirect addressing (with some limitations; see below). For the difference between the two addressing modes, see Section 3.3.

Assembler directives have their own semantics and format. The current implementation uses a very small subset of the *GNU assembler directives.*

The next pages give a brief description of every instruction and directive. For instructions, a complete description can be found in the MACE documentation (Section 3.4).

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

R*n* Register *n*.

(R*n*) Indirect addressing of register n; `[Rn]` is used in the description of the operation

Rdest Destination Register.

Rsource1 First source operand.

Rsource2 Second source operand.

#imm Immediate value. `imm` is a signed 16-bit integer value.

L*n* A label

2.2.1 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	*Add binary*
Syntax:	ADD Rdest Rsource1 Rsource2
Examples	Semantics
ADD R3 R1 R2	R3 ← R1 + R2
ADD R3 R1 (R2)	R3 ← R1 + [R2]
ADD (R3) R1 (R2)	[R3] ← R1 + [R2]

SUB	*Subtract binary*
Syntax:	SUB Rdest Rsource1 Rsource2
Examples	Semantics
SUB R3 R1 R2	R3 ← R1 − R2
SUB R3 R1 (R2)	R3 ← R1 − [R2]
SUB (R3) R1 (R2)	[R3] ← R1 − [R2]

ANDL	*AND logical*
Syntax:	ANDL Rdest Rsource1 Rsource2
Examples	Semantics
ANDL R3 R1 R2	R3 ← R1 && R2
ANDL R3 R1 (R2)	R3 ← R1 && [R2]
ANDL (R3) R1 (R2)	[R3] ← R1 && [R2]

ORL	*OR logical*
Syntax:	ORL Rdest Rsource1 Rsource2
Examples	Semantics
ORL R3 R1 R2	R3 $\leftarrow$ R1 $\|\|$ R2
ORL R3 R1 (R2)	R3 $\leftarrow$ R1 $\|\|$ [R2]
ORL (R3) R1 (R2)	[R3] $\leftarrow$ R1 $\|\|$ [R2]

EORL	*Exclusive OR logical*
Syntax:	EORL Rdest Rsource1 Rsource2
Examples	Semantics
EORL R3 R1 R2	R3 $\leftarrow$ R1 $\bigoplus$ R2
EORL R3 R1 (R2)	R3 $\leftarrow$ R1 $\bigoplus$ [R2]
EORL (R3) R1 (R2)	[R3] $\leftarrow$ R1 $\bigoplus$ [R2]

ANDB	*AND bit by bit*
Syntax:	ANDB Rdest Rsource1 Rsource2
Examples	Semantics
ANDB R3 R1 R2	R3 $\leftarrow$ R1 & R2
ANDB R3 R1 (R2)	R3 $\leftarrow$ R1 & [R2]
ANDB (R3) R1 (R2)	[R3] $\leftarrow$ R1 & [R2]

ORB	*OR bit by bit*
Syntax:	ORB Rdest Rsource1 Rsource2
Examples	Semantics
ORB R3 R1 R2	R3 $\leftarrow$ R1 $\|$ R2
ORB R3 R1 (R2)	R3 $\leftarrow$ R1 $\|$ [R2]
ORB (R3) R1 (R2)	[R3] $\leftarrow$ R1 $\|$ [R2]

EORB	*Exclusive OR bit by bit*
Syntax:	EORB Rdest Rsource1 Rsource2
Examples	Semantics
EORB R3 R1 R2	R3 $\leftarrow$ R1 $\bigoplus$ R2
EORB R3 R1 (R2)	R3 $\leftarrow$ R1 $\bigoplus$ [R2]
EORB (R3) R1 (R2)	[R3] $\leftarrow$ R1 $\bigoplus$ [R2]

MUL	*Multiply binary*
Syntax:	MUL Rdest Rsource1 Rsource2
Examples	Semantics
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	*Divide binary*
Syntax:	DIV Rdest Rsource1 Rsource2
Examples	Semantics
DIV R3 R1 R2	R3 ← R1 / R2
DIV R3 R1 (R2)	R3 ← R1 / [R2]
DIV (R3) R1 (R2)	[R3] ← R1 / [R2]

SHR	*Binary shift to the right*
Syntax:	SHR Rdest Rsource1 Rsource2
Examples	Semantics
SHR R3 R1 R2	R3 ← R1 $\gg$ R2
SHR R3 R1 (R2)	R3 ← R1 $\gg$ [R2]
SHR (R3) R1 (R2)	[R3] ← R1 $\gg$ [R2]

SHL	*Binary shift to the left*
Syntax:	SHL Rdest Rsource1 Rsource2
Examples	Semantics
SHL R3 R1 R2	R3 ← R1 $\ll$ R2
SHL R3 R1 (R2)	R3 ← R1 $\ll$ [R2]
SHL (R3) R1 (R2)	[R3] ← R1 $\ll$ [R2]

ROTL	*Rotate binary to the left*
Syntax:	ROTL Rdest Rsource1 Rsource2
Semantics:	Rdest ← Rsource1 Rsource2
Description:	`Rsource1` value is rotated to the left by `Rsource2` positions.
Note:	Actually, the `ROTL` instruction is not supported by the current architecture.

ROTR	*Rotate binary to the right*
Syntax:	ROTR Rdest Rsource1 Rsource2
Semantics:	Rdest ← Rsource1 Rsource2
Description:	`Rsource1` value is rotated to the right by `Rsource2` positions.
Note:	Actually, the `ROTR` instruction is not supported by the current architecture.

NEG	*Negate*
Syntax:	NEG Rdest Rsource1 Rsource2
Examples	Semantics
NEG R3 R1 R2	R3 $\leftarrow$ - R2
NEG R3 R1 (R2)	R3 $\leftarrow$ - [R2]
NEG (R3) R1 (R2)	[R3] $\leftarrow$ - [R2]
Note:	Rsource1 is unused.

SPCL	*Special opcode*
Syntax:	SPCL Rdest Rsource1 Rsource2
Semantics:	Undefined at the moment.

2.2.2 Binary Instructions

Binary instructions are instructions with two register operands, although they generally have three operands. As before, the first operand is the *destination.* All register operands of binary instructions are always *directly* addressed.

ADDI	*Add with immediate operand*
Syntax:	ADDI Rdest Rsource #Immediate
Example	Semantics
ADDI R2 R1 #1	R2 $\leftarrow$ R1 + 1

SUBI	*Subtract with immediate operand*
Syntax:	SUBI Rdest Rsource #Immediate
Example	Semantics
SUBI R2 R1 #1	R2 $\leftarrow$ R1 $-$ 1

ANDLI	*AND with immediate operand*
Syntax:	ANDLI Rdest Rsource #Immediate
Example	Semantics
ANDLI R2 R1 #1	R2 $\leftarrow$ R1 && 1

ORLI	*OR with immediate operand*
Syntax:	ORLI Rdest Rsource #Immediate
Example	Semantics
ORLI R2 R1 #1	R2 $\leftarrow$ R1 $\|\|$ 1

EORLI	*Exclusive OR with immediate operand*
Syntax:	EORLI Rdest Rsource #Immediate
Example	Semantics
EORLI R2 R1 #1	R2 $\leftarrow$ R1 $\oplus$ 1

ANDBI	*AND bit by bit with immediate operand*
Syntax:	ANDBI Rdest Rsource #Immediate
Example	Semantics
ANDBI R2 R1 #345	R2 $\leftarrow$ R1 & 345

ORBI	*OR bit by bit with immediate operand*
Syntax:	ORBI Rdest Rsource #Immediate
Example	Semantics
ORBI R2 R1 #345	R2 $\leftarrow$ R1 $\mid$ 345

EORBI	*Exclusive OR bit by bit with immediate operand*
Syntax:	EORBI Rdest Rsource #Immediate
Example	Semantics
EORBI R2 R1 #345	R2 $\leftarrow$ R1 $\oplus$ 345

MULI	*Multiply binary with immediate operand*
Syntax:	MULI Rdest Rsource #Immediate
Example	Semantics
MULI R2 R1 #345	R2 $\leftarrow$ R1 $\times$ 345

DIVI	*Divide binary with immediate operand*
Syntax:	DIVI Rdest Rsource #Immediate
Example	Semantics
DIVI R2 R1 #345	R2 $\leftarrow$ R1 / 345

SHRI	*Binary Shift to the right with immediate operand*
Syntax:	SHRI Rdest Rsource #Immediate
Example	Semantics
SHRI R2 R1 #3	R2 $\leftarrow$ R1 $\gg$ 3

SHLI	*Binary Shift to the left with immediate operand*
Syntax:	SHLI Rdest Rsource #Immediate
Example	Semantics
SHLI R2 R1 #3	R2 $\leftarrow$ R1 $\ll$ 3

ROTLI	*Rotate binary to the left with immediate operand*
Syntax:	ROTLI Rdest Rsource #Immediate
Example	Semantics
ROTLI R2 R1 #3	R2 $\leftarrow$ R1 rotated to the left by 3 positions
Note:	Actually, the `ROTLI` instruction is not supported by the current architecture.

ROTRI	*Rotate binary to the right with immediate operand*
Syntax:	ROTRI Rdest Rsource #Immediate
Example	Semantics
ROTRI R2 R1 #3	R2 $\leftarrow$ R1 rotated to the right by 3 positions
Note:	Actually, the `ROTRI` instruction is not supported by the current architecture.

NOTL	*Logical complement*
Syntax:	NOT Rdest Rsource #Immediate
Example:	Semantics
NOTL R2 R1 #0	R2 $\leftarrow$! R1
Note:	The third operand is ignored.

NOTB	*Binary complement*
Syntax:	NOTB Rdest Rsource #Immediate
Example	Semantics
NOTB R2 R1 #0	R2 $\leftarrow \sim$ R1
Note:	The third operand is ignored.

2.2.3 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. Again, the first operand (where present) is the *destination*, and all register operands are always *directly* addressed.

NOP	*No Operation*
Syntax:	NOP

MOVA	*Move Address to Register Location*
Syntax:	MOVA RDest Address
Example	Semantics
MOVA R2 L1	R2 ← L1 (where L1 is a label)

JSR	*Jump To Subroutine*
Syntax:	JSR RDest Address
Note:	Not implemented yet.

RET	*Return from Subroutine*
Syntax:	RET
Note:	Not implemented yet. Currently, the assembler generates a HALT instruction instead of a RET.

LOAD	*Fill a register with a value read from memory*
Syntax:	LOAD RDest Address
Example	Semantics
LOAD R2 L1	R2 ← [L1] (where L1 is a Label)

STORE	*Spill a value*
Syntax:	STORE RSource Address
Example	Semantics
STORE R2 L1	[L1] ← R2 (where L1 is a Label)

HALT	*Halt the machine processor*
Syntax:	RET

Scc	*Set according to condition ‘cc’*
Syntax:	Scc Rdest Address
Semantics (pseudo-code):	IF cc == 1 THEN Rdest ← 1; ELSE Rdest ← 0.
Note:	`Address` parameter is unused. For more information see the `MACE` documentation
Possible values for ‘cc’:	
EQ	set on equal;
GE	set on greater than or equal;
GT	set on greater than;
LE	set on less than or equal;
LT	set on less than;
NE	set on not equal;
Description:	The specified condition code ‘cc’ is tested. If the condition is true, ‘Rdest’ is set to one; Otherwise ‘Rdest’ is set to zero.
Example:	
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*
Syntax:	READ RSource Address
Example	Semantics
READ R2 0	Read a 32-bit signed integer value from standard input, and store it into ‘R2’.
Note:	`Address` parameter is unused.

WRITE	*Write an integer value to standard output*
Syntax:	WRITE RSource Address
Example	Semantics
WRITE R2 0	Write the 32-bit signed integer value stored in R2 to standard output.
Note:	`Address` parameter is unused.

2.2.4 Jump Instructions

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

Bcc	*Branch on condition cc*
Syntax:	Bcc Label
Semantics (pseudocode):	IF cc == 1 THEN jump to label `Label`.
Note:	For more information, see the `MACE` documentation.
Possible values for 'cc':	
EQ	Branch on equal;
GE	Branch on greater than or equal;
T	Branch always: the branch is always 'taken';
F	This condition is never verified, thus, branches like this are never 'taken';
HI	Branch on higher than
LS	Branch on lower than or same;
GT	Branch on greater than;
LE	Branch on less than or equal;
LT	Branch on less than;
NE	Branch on not equal;
CC	Branch on carry clear;
CS	Branch on carry set;
VC	Branch on overflow clear;
VS	Branch on overflow set;
PL	Branch on plus (i.e. positive);
MI	Branch on minus (i.e. negative);
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.
Examples:	
BEQ L1	Branch to L1 on "equal to zero"
BT L3	Always branch to L3
BLT L2	Branch to L2 on "less than zero"

2.2.5 Assembler Directives

The current implementation provides only a very limited set of assembler directives. All assembler directives have names that begin with a period ('.') and every directive has its own semantic associated with.

Here is a list of all the supported directives:

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

.text *Marks the beginning of a block of instructions*

.word	*Reserve and set a memory word (32-bit) in the data segment*
Syntax:	`.word` VAL
Semantics:	Reserve a 32-bit memory location inside the data segment and set the starting value of the location to `VAL`
Examples:	
`.word 5`	reserve a word location and set its content to 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 reserve (without initialize) a given number of bytes into the data segment*
Syntax:	`.space` VAL
Semantics:	Reserve `VAL` (contiguous) bytes inside the data segment. If VAL is not a multiple of 4, it rounded up to the next multiple of 4, so as data memory addresses are always aligned to 32-bit boundaries. The reserved memory is filled with zeros.
Examples:	
`.space 32`	32 contiguous bytes reserved
`L1 .space 5`	8 (and not just 5) contiguous bytes reserved; the memory can be referred to with the label `L1`

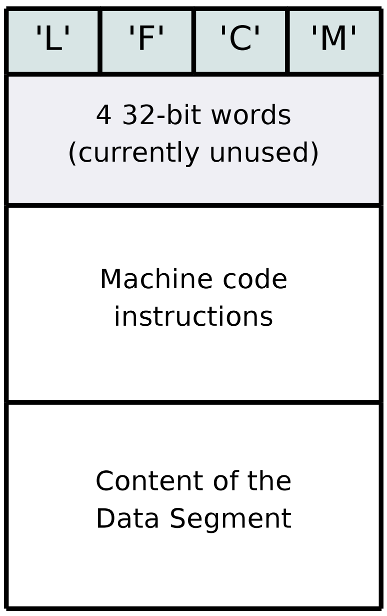
'L'
'F'
'C'
'M'
4 32-bit words (currently unused)
Machine code instructions
Content of the Data Segment

Figure 2.1: Object file format

2.3 Object file format

An object file is returned as result of the assembling process. It contains both machine code instructions and the information about the data segment (its size and initial content).

Figure 2.1 shows the format of an object file. An object file is composed by an header of 20 bytes 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.

Chapter 3

MACE

`MACE` (Machine for Advanced Compiler Education) is a program that emulates the execution of an object file (containing both machine code and data) produced by an assembler.

Both the object file format specification and the symbolic assembler are discussed in the assembler documentation (Chapter 2). This document describes the overall internal machine architecture and the supported instruction set.

The first sections briefly introduce the internal architecture of the MACE machine and the execution steps performed by the machine at run-time. The last two sections describe the complete instruction set and the encoding format of every instruction.

3.1 Architecture

Figure 3.1 shows the architectural design of the machine emulated by MACE.

The architecture is composed by:

- 32 general-purpose 32-bit registers;
- one 32-bit program counter (`PC`);
- one 32-bit status register (`PSW`);
- a 32-bit memory, word-addressed.

3.1.1 Data Registers

The current architecture provides 32 general-purpose registers (`R0`–`R31`). These registers are typically used for word (32 bits) operations. Register `R0` is al-

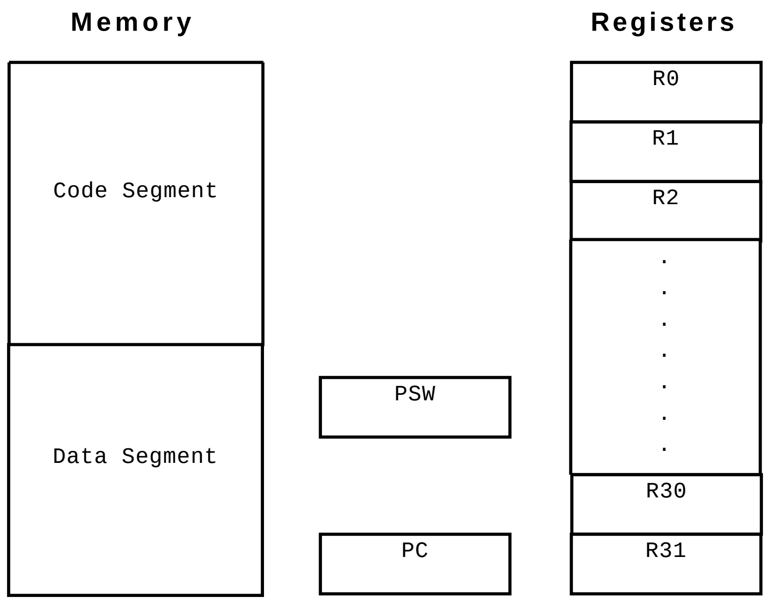

Figure 3.1: Architectural Design of MACE

ways set to 0 (i.e., even if an instruction tries to assign a value to R0, all the following instructions will always see a value of zero inside the R0 location).

3.1.2 Program Counter

The PC contains the address of the instruction currently executing. After the execution of an instruction, the processor automatically increments its content, or, for branch instructions, inserts a new value in the PC.

3.1.3 Status Register

The Status Register (PSW) is a 32-bit register, but only its four less-significant bits are available in user mode. Many integer instructions affect the content of the PSW. Program instructions also use certain combinations of these bits to control program and system flow. The four less-significant bits represent conditions about the result generated by previous operations. In the instruction set definitions, the PSW is illustrated as in figure 3.2.

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.

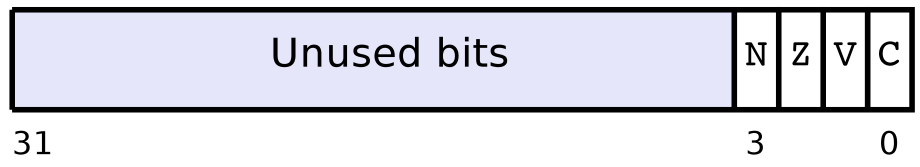

Figure 3.2: Status Register

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.

3.1.4 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 the memory, i.e., from the fifth to the eighth byte. MACE addressing is different from common CPUs, which address memory at byte boundaries.

3.2 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:

- Test if the object file given as input exists and is readable.
- The content of every machine register is set to zero. Thus, `PC` will point to the first instruction in the code segment.
- A block of memory of size 2 KB is reserved and will contain both the code and data segments.
- The machine code is loaded from the object file into the code segment.
- Data is loaded from the object file into the data segment, right after the code segment.

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

- Repeat:
  - Fetch the next instruction according to the value of `PC`
  - Decode the fetched instruction
  - Execute the instruction
  - Update (if necessary) the content of the register file
  - Update the value of the program counter (`PC`)
  - Update the value of the status register (`PSW`)
- Until a `HALT` instruction is encountered.

3.3 Addressing Capabilities

Most instructions take one or two source operands, process them, and store the result in the destination location. Some instructions have only one destination operand. External microprocessor references to memory are either program references that refer to program space or data references that refer to data space. Program space is the section of memory that contains the program instructions and any immediate data operands residing in the instruction stream. Data space is the section of memory that contains the program data. Data items in the instruction stream can be accessed with the program counter relative addressing modes.

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. For example, the instruction `ADD R3 R1 R2` means that the values contained in registers `R1` and `R2` are summed, and the result is stored into register `R3`; `ADD (R3) R1 R2` means that the values contained in registers `R1` and `R2` are summed, and the result is stored into the memory cell whose address is contained in register `R3`; in this latest case, register `R3` is not modified in any way.

3.3.1 Instruction Format

Instructions consist of exactly one word (i.e., 32 bits). Figure 3.3 illustrates the general composition of every type of instruction.

An instruction specifies the function to be performed with an operation code (i.e. `opcode`) and defines the location of every operand.

Possible operands are:

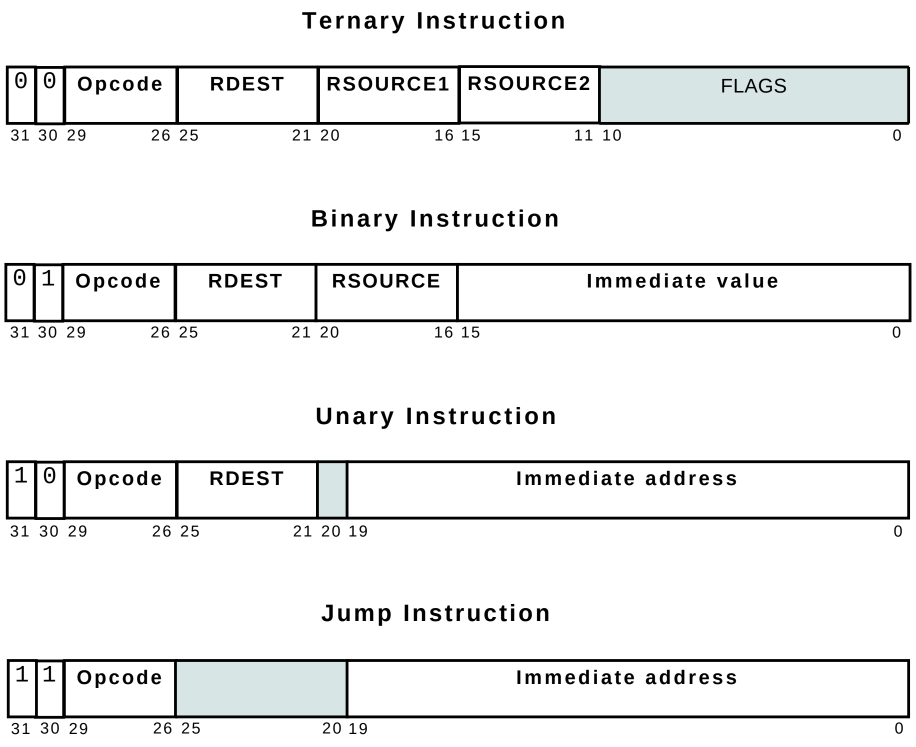

Figure 3.3: Instruction Formats

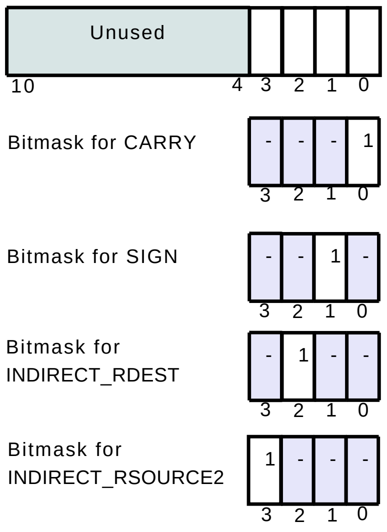

Figure 3.4: Flag bits for ternary instructions

- register identifiers, directly or indirectly addressed;
- immediate values, i.e, a number specified within the instruction;
- address values (typically labels).

The two most significant bits of every instruction (bits 31 and 30) are always set to a value that depends on the instruction format. For example, as we can see in Figure 3.3 ternary instructions always have those bits set to '0'.

Every register identifier is defined as a 5-bit value that represents the general register number. For example, register R3 will be encoded in the following binary format: '00011'.

Unary and binary instructions use always *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 addressing even for ternary instructions.

In ternary instructions this information is encoded inside the *flag bits* (the 11 less-significant bits of an instruction word). The format of the flag bits is shown in Figure 3.4.

If the bit 'CARRY' of an instruction is set to 1, and the bit 'C' in the PSW is also set to 1, the result of the binary operation between RSOURCE1 and RSOURCE2 is incremented by 1 for ADD and MUL, and it is decremented by for SUB and DIV. This makes possible to implement multi-precision arithmetic.

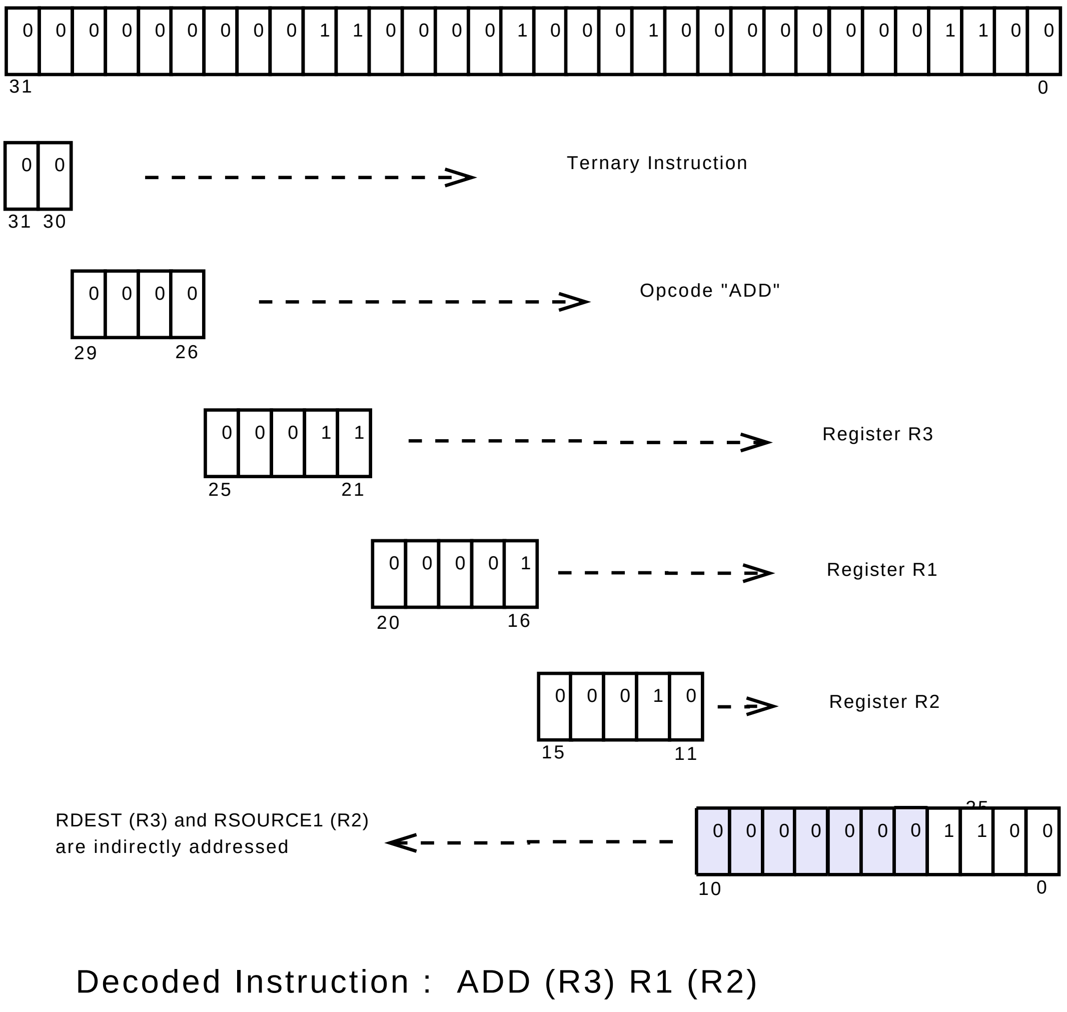

Figure 3.5: An example of an encoded ADD instruction

If the bit ‘SIGN’ is set to 1, MACE treats the values stored in RSOURCE1 and RSOURCE2 as signed integers.

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

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

Figure 3.5 shows an example of coding for the instruction `ADD (R3) R1 (R2)`.

3.4 Instruction Set

In this section, instructions are listed by mnemonics. 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 effect of an instruction on the PSW is specified by the following codes:

U 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.

The following notation is used when we refers to registers and immediate values:

R*n* A register location; it may be either directly or indirectly addressed, depending on the instruction.

imm An immediate value, a signed 16-bit integer.

3.4.1 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	*Add binary*
Syntax:	ADD Rdest Rsource1 Rsource2
Semantics:	Rdest ← Rsource1 + Rsource2
Binary Opcode:	‘0000’
Description:	Add the source operand ‘Rsource1’ to ‘Rsource2’ and store the result into the destination location ‘Rdest’.

Condition codes:	`N Z V C`
	`* * * *`

SUB	*Subtract binary*
Syntax:	SUB Rdest Rsource1 Rsource2
Semantics:	Rdest ← Rsource1 − Rsource2
Binary Opcode:	‘0001’
Description:	Subtract the source operand ‘Rsource1’ from ‘Rsource2’ and store the result into the destination location ‘Rdest’.

Condition codes:	`N Z V C`
	`* * * *`

ANDL	*AND logical*
Syntax	ANDL Rdest Rsource1 Rsource2
Semantics	Rdest ← Rsource1 && Rsource2
Binary Opcode:	‘0010’
Description:	Perform an AND between the source operands ‘Rsource1’ and ‘Rsource2’ and store the result into the destination location ‘Rdest’.

Condition codes:	`N Z V C`
	`* * 0 0`

ORL	*OR logical*
Syntax	ORL Rdest Rsource1 Rsource2
Semantics:	Rdest ← Rsource1 | | Rsource2
Binary Opcode:	‘0011’
Description:	Perform an OR between the source ‘Rsource1’ and ‘Rsource2’ and store the result into the destination location ‘Rdest’.

Condition codes:	`N Z V C`
	`* * 0 0`

EORL	*Exclusive OR logical*
Syntax	EORL Rdest Rsource1 Rsource2
Semantics:	Rdest $\leftarrow$ Rsource1 $\oplus$ Rsource2
Binary Opcode:	‘0100’
Description:	EOR (exclusive or) the source operand ‘Rsource1’ with ‘Rsource2’ and store the result into the destination location ‘Rdest’.

Condition codes:	`N Z V C`
	`* * 0 0`

ANDB	*AND bit by bit*
Syntax	ANDB Rdest Rsource1 Rsource2
Semantics	Rdest $\leftarrow$ Rsource1 & Rsource2
Binary Opcode:	‘0101’
Description:	ANDB the source operands ‘Rsource1’ and ‘Rsource2’ and store the result into the destination location ‘Rdest’.

Condition codes:	`N Z V C`
	`* * 0 0`

ORB	*OR bit by bit*
Syntax	ORB Rdest Rsource1 Rsource2
Semantics:	Rdest $\leftarrow$ Rsource1 $\mid$ Rsource2
Binary Opcode:	‘0110’
Description:	ORB the source operands ‘Rsource1’ and ‘Rsource2’, and store the result into the destination location ‘Rdest’.

Condition codes:	`N Z V C`
	`* * 0 0`

EORB	*Exclusive OR bit by bit*
Syntax	EORB Rdest Rsource1 Rsource2
Semantics:	Rdest $\leftarrow$ Rsource1 $\oplus$ Rsource2
Binary Opcode:	‘0111’
Description:	EORB (exclusive or) the source operands ‘Rsource1’ and ‘Rsource2’, and store the result into the destination location ‘Rdest’.

Condition codes:	`N Z V C`
	`* * 0 0`

MUL	*Multiply binary*
Syntax	MUL Rdest Rsource1 Rsource2
Semantics:	Rdest $\leftarrow$ Rsource1 $\times$ Rsource2
Binary Opcode:	‘1000’
Description:	Multiply the 32-bit ‘Rsource1’ operand by the 32-bit ‘Rsource2’ operand and store the result into the destination ‘Rdest’. Both the sources and destination are 32-bit word values.

Condition codes:	`N Z V C`
	`* * * *`

DIV	*DIV binary*
Syntax	DIV Rdest Rsource1 Rsource2
Semantics:	Rdest $\leftarrow$ Rsource1 / Rsource2
Binary Opcode:	‘1001’
Description:	Divide the 32-bit ‘Rsource1’ operand by the 32-bit ‘Rsource2’ operand and store the result into the destination ‘Rdest’. Both the sources and destination are 32-bit word values.

Condition codes:	`N Z V C`
	`* * * *`

SHR	*Binary Shift to the Right*
Syntax	SHR Rdest Rsource1 Rsource2
Semantics:	Rdest $\leftarrow$ Rsource1 $\gg$ Rsource2
Binary Opcode:	‘1011’
Description:	SHR Performs a binary ‘shift to right’ on the 32-bit ‘Rsource1’ operand. The number of bits shifted is contained in the 32-bit ‘Rsource2’ operand. The result of the shift operation is stored into the destination register ‘Rdest’.

Condition codes:	`N Z V C`
	`* * 0 *`

SHL	*Binary Shift to the Left*
Syntax	SHL Rdest Rsource1 Rsource2
Semantics:	Rdest $\leftarrow$ Rsource1 $\ll$ Rsource2
Binary Opcode:	‘1010’
Description:	SHL Performs a binary ‘shift to left’ on the 32-bit ‘Rsource1’ operand. The number of bits shifted is contained into the 32-bit ‘Rsource2’ operand. The result of the shift operation is stored into the destination register ‘Rdest’.

Condition codes:	`N Z V C`
	`* * 0 *`

ROTL	*Rotate binary to the left*
Syntax	ROTL Rdest Rsource1 Rsource2
Semantics:	Rdest $\leftarrow$ Rsource1 rotated by <Rsource2> to Left
Binary 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.
Note:	Actually, the ROTL instruction is not supported by the current architecture; it is executed as a NOP operation

Condition codes:	`N Z V C`
	`* * 0 *`

ROTR	*Rotate binary to the right*
Syntax	ROTR Rdest Rsource1 Rsource2
Semantics:	Rdest $\leftarrow$ Rsource1 rotated by <Rsource2> to Right
Binary 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.
Note:	Actually, the ROTR instruction is not supported by the current architecture; it is executed as a NOP operation.

Condition codes:	`N Z V C`
	`* * 0 *`

NEG	*Negate*
Syntax	NEG Rdest Rsource1 Rsource2
Semantics:	Rdest $\leftarrow$ 0 - Rsource2 Rsource1 is unused.
Binary Opcode:	‘1110’
Description:	Negate the value of ‘Rsource2’ and store the result into ‘RDest’.

Condition codes:	`N Z V C`
	`* * * *`

SPCL	*Special opcode*
Syntax	SPCL Rdest Rsource1 Rsource2
Semantics:	Undefined at the moment.
Binary Opcode:	‘1111’
Description:	Will be used for future developments

Condition codes:	`N Z V C`
	`U U U U`

3.4.2 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*
Syntax:	ADDI Rdest Rsource1 #Immediate
Semantics:	Rdest $\leftarrow$ Rsource1 + #Immediate
Binary Opcode:	‘0000’
Description:	Add the source operand ‘Rsource1’ to the ‘immediate’ value and store the result into the destination location ‘Rdest’.

Condition codes:	`N Z V C`
	`* * * *`

SUBI	*Subtract with Immediate operand*
Syntax:	SUBI Rdest Rsource1 #Immediate
Semantics:	Rdest ← Rsource1 − #Immediate
Binary Opcode:	‘0001’
Description:	Subtract the source operand ‘Rsource1’ from ‘immediate’ value and store the result into the destination location ‘Rdest’.

Condition codes:	`N Z V C`
	`* * * *`

ANDLI	*AND with Immediate operand*
Syntax	ANDLI Rdest Rsource1 #Immediate
Semantics	Rdest ← Rsource1 && #Immediate
Binary Opcode:	‘0010’
Description:	Performs a logical AND between the source operand ‘Rsource1’ and the ‘Immediate’ operand and store the result into the destination location ‘Rdest’.

Condition codes:	`N Z V C`
	`* * 0 0`

ORLI	*OR with Immediate operand*
Syntax	ORLI Rdest Rsource1 #Immediate
Semantics:	Rdest ← Rsource1 | | #Immediate
Binary Opcode:	‘0011’
Description:	Performs an OR between the source operand ‘Rsource1’ and the ‘Immediate’ operand and store the result into the destination location ‘Rdest’.

Condition codes:	`N Z V C`
	`* * 0 0`

EORLI	*Exclusive OR with Immediate operand*
Syntax	EORLI Rdest Rsource1 #Immediate
Semantics:	Rdest $\leftarrow$ Rsource1 $\oplus$ #Immediate
Binary Opcode:	'0100'
Description:	EOR (exclusive or) the source operand 'Rsource1' with 'Immediate' and store the result into the destination location 'Rdest'.

Condition codes:	`N Z V C`
	`* * 0 0`

ANDBI	*AND bit by bit with Immediate operand*
Syntax	ANDBI Rdest Rsource1 #Immediate
Semantics	Rdest $\leftarrow$ Rsource1 && #Immediate
Binary Opcode:	'0101'
Description:	ANDBI the source operand 'Rsource1' to the 'Immediate' operand and store the result into the destination location 'Rdest'.

Condition codes:	`N Z V C`
	`* * 0 0`

ORBI	*OR bit by bit with Immediate operand*
Syntax	ORBI Rdest Rsource1 #Immediate
Semantics:	Rdest $\leftarrow$ Rsource1 $\mid$ #Immediate
Binary Opcode:	'0110'
Description:	ORBI the source operand 'Rsource1' to the 'Immediate' operand, and store the result into the destination location 'Rdest'.

Condition codes:	`N Z V C`
	`* * 0 0`

EORBI	*Exclusive OR bit by bit with immediate operand*
Syntax	EORBI Rdest Rsource1 #Immediate
Semantics:	Rdest ← Rsource1 ⊕ #Immediate
Binary Opcode:	‘0111’
Description:	EORBI (exclusive or) the source operand ‘Rsource1’ with the ‘Immediate’ operand and store the result into the destination location ‘Rdest’.

Condition codes:	`N Z V C`
	`* * 0 0`

MULI	*Multiply binary with Immediate operand*
Syntax	MULI Rdest Rsource1 #Immediate
Semantics:	Rdest ← Rsource1 * #Immediate
Binary Opcode:	‘1000’
Description:	Multiply the 32-bit ‘Rsource1’ operand by the ‘Immediate’ operand and store the result into the destination ‘Rdest’. Both the source and destination registers are 32-bit, but the immediate operand value is always 16-bit.

Condition codes:	`N Z V C`
	`* * * *`

DIVI	*Divide binary with Immediate operand*
Syntax	DIVI Rdest Rsource1 #Immediate
Semantics:	Rdest ← Rsource1 / #Immediate
Binary Opcode:	‘1001’
Description:	Divide the 32-bit ‘Rsource1’ operand by ‘Immediate’ operand and store the result into the destination ‘Rdest’. Both the sources 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`
	`* * * *`

SHRI	*Binary Shift to Right*
Syntax	SHRI Rdest Rsource1 #Immediate
Semantics:	Rdest $\leftarrow$ Rsource1 $\gg$ #Immediate
Binary Opcode:	'1011'
Description:	SHR Performs a binary 'shift to right' on the 32-bit 'Rsource1' operand. The number of bits shifted is given by the value of the Immediate operand 'Immediate'. The result of the shift operation is stored into the destination register 'Rdest'.

Condition codes:	`N Z V C`
	`* * * *`

SHLI	*Binary Shift to Left*
Syntax	SHLI Rdest Rsource1 #Immediate
Semantics:	Rdest $\leftarrow$ Rsource1 $\ll$ #Immediate
Binary Opcode:	'1010'
Description:	SHL Performs a binary 'shift to left' on the 32-bit 'Rsource1' operand. The number of bits shifted is given by the value of the Immediate operand 'Immediate'. The result of the shift operation is stored into the destination register 'Rdest'.

Condition codes:	`N Z V C`
	`* * * *`

ROTLI	*Rotate binary*
Syntax	ROTLI Rdest Rsource1 #Immediate
Semantics:	Rdest ← Rsource1 rotated by <#Immediate> to Left
Binary 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.
Note:	Actually, the ROTLI instruction is not supported by the current architecture; it is executed as a NOP operation.

Condition codes:	`N Z V C`
	`* * 0 *`

ROTRI	*Rotate binary*
Syntax	ROTRI Rdest Rsource1 #Immediate
Semantics:	Rdest ← Rsource1 rotated by <#Immediate> to Right
Binary 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.
Note:	Actually, the ROTRI instruction is not supported by the current architecture; it is executed as a NOP operation.

Condition codes:	`N Z V C`
	`* * 0 *`

NOTL	*Logical complement*
Syntax	NOT Rdest Rsource1 #Immediate
Semantics:	Rdest ← ! Rsource1 Immediate is unused.
Binary Opcode:	'1110'
Description:	Perform a logical NOT operation on the value of 'Rsource1'. The result is stored into 'RDest'.

Condition codes:	`N Z V C`
	`* * 0 0`

NOTB	*Binary complement*
Syntax	NOTB Rdest Rsource1 #Immediate
Semantics:	Rdest ← ∼Rsource1 Immediate is unused.
Binary Opcode:	'1110'
Description:	Perform a binary NOT operation on the value of 'Rsource1'. The result is stored into 'RDest'.

Condition codes:	`N Z V C`
	`* * 0 0`

3.4.3 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*
Syntax	NOP
Binary 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 Location*
Syntax	MOVA Rdest Address
Semantics:	Rdest ← Address
Binary Opcode:	'0001'
Description:	Move the value of Address into 'RDest'. Address is a 20-bit value
Usage:	MOVA instructions are typically used to work on addresses (pointers) or arrays; in the assembly, a label is used instead of a numeric address.

Condition codes:	`N Z V C`
	`- - - -`

JSR	*Jump To Subroutine*
Syntax	JSR Rdest Address
Semantics:	Jump to the subroutine at address Address. Store the return value of the subroutine inside the register 'RDest'.
Binary Opcode:	'0010'
Description:	This instruction implements a jump to subroutine.
Note:	Currently, this opcode is not supported.

Condition codes:	`N Z V C`
	`- - - -`

RET	*Return from Subroutine*
Description:	RET is not supported by the current architecture.
Note:	Currently, the RET instruction is not supported.
Binary Opcode:	'0011'

Condition codes:	`N Z V C`
	`- - - -`

LOAD	*Fill a register with a value read from memory*
Syntax	LOAD Rdest Address
Semantics:	Rdest $\leftarrow$ *Address
Binary Opcode:	‘0100’
Description:	Load the value previously stored at ‘Address’ memory location inside the register ‘Rdest’

Condition codes:	`N Z V C`
	`- - - -`

STORE	*Spill a value*
Syntax	STORE Rsource Address
Semantics:	*Address $\leftarrow$ Rsource
Binary Opcode:	‘0101’
Description:	Store the value of ‘Rsource’ to the ‘Address’ memory location

Condition codes:	`N Z V C`
	`- - - -`

HALT	*Halt the machine processor*
Binary Opcode:	‘0110’

Condition codes:	`N Z V C`
	`- - - -`

Scc	*Set according to condition ‘cc’*
Syntax	Scc Rdest Address
Semantics (pseudo-code):	IF cc == 1 THEN Rdest $\leftarrow$ 1; ELSE Rdest $\leftarrow$ 0.
Note:	Condition ‘cc’ is computed starting from the value of the `PSW` register. For example, the instruction ‘SEQ Rx’ stores 1 into Rx if the bit ‘N’ of the status register `PSW` is set, otherwise Rx is set to zero. `C,N,V,Z` refer to the bits of the status register. For example: `C` refers to the carry bit of the `PSW` register. Address is unused.
Possible values for ‘cc’:	
EQ	set on equal; cc $\leftarrow$ Z
GE	set on greater than or equal; cc $\leftarrow \mathrm{N} \cdot \mathrm{V} + \overline{\mathrm{N}} \cdot \overline{\mathrm{V}}$
GT	set on greater than; cc $\leftarrow \mathrm{N} \cdot \mathrm{V} \cdot \overline{\mathrm{Z}} + \overline{\mathrm{N}} \cdot \overline{\mathrm{V}} \cdot \overline{\mathrm{Z}}$
LE	set on less than or equal; cc $\leftarrow \mathrm{Z} + \mathrm{N} \cdot \overline{\mathrm{V}} + \overline{\mathrm{N}} \cdot \mathrm{V}$
LT	set on less than; cc $\leftarrow \overline{\mathrm{N}} \cdot \mathrm{V} + \mathrm{N} \cdot \overline{\mathrm{V}}$
NE	set on not equal; cc $\leftarrow \overline{\mathrm{Z}}$
Description:	The specified condition code ‘cc’ is tested. If the condition is true, ‘Rdest’ is set to one; Otherwise ‘Rdest’ is set to zero.
Binary Op-codes:	
SEQ	‘0111’
SGE	‘1000’
SGT	‘1001’
SLE	‘1010’
SLT	‘1011’
SNE	‘1100’

Condition codes:	`N Z V C`
	`0 * 0 0`

READ	*Read an integer value from standard input*
Syntax	READ Rsource Address
Semantics:	Read from input an 32-bit signed integer value, and store the value to ‘Rsource’. Address is unused.
Binary Opcode:	‘1101’
Description:	This instruction loads a 32-bit value from standard input into a register. Actually the READ instruction is implemented with a `scanf`. [See the declaration of ‘scanf’ in the `stdio.h` header file of the C standard library].

Condition codes:	`N Z V C`
	`* * * *`

WRITE	*Write an integer value to standard output*
Syntax	WRITE Rsource Address
Semantics:	Write to standard output a 32-bit signed integer value stored into Rsource. Address is unused.
Binary Opcode:	‘1110’
Description:	This instruction writes a 32-bit value to standard output. Actually the WRITE instruction is implemented with a `printf`. [See the declaration of ‘printf’ in the `stdio.h` header file of the C standard library].

Condition codes:	`N Z V C`
	`- - - -`

3.4.4 Jump Instructions

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

Bcc	*Branch on condition cc*
Syntax	Bcc Label
Semantics (pseudo-code):	IF cc == 1 THEN `PC` ← `PC` + **Displacement**; *Displacement* is the distance between the current ***PC*** and the address associated with the given ‘Label’
Note:	Condition ‘cc’ is analyzed by taking into consideration the value of the `PSW` register. For example, the instruction ‘`BEQ Label`’ performs a branch to the instruction labeled ‘label’ if the bit ‘N’ of the status register `PSW` is set. `C,N,V,Z` refer to the bits of the status register (`PSW`). For example, `C` refers to the carry bit of the `PSW` register.
Possible values for ‘cc’:	

EQ	Branch on equal;
	cc ← Z

GE	Branch on greater than or equal; cc ← $N \cdot V + \overline{N} \cdot \overline{V}$
T	Branch always: the branch is always ‘taken’; cc ← 1
F	This condition is never verified, thus, branches like this are never ‘taken’; cc ← 0
HI :	Branch on higher than cc ← $\overline{C} \cdot \overline{Z}$
LS	Branch on lower than or same; cc ← $C \cdot Z$
GT	Branch on greater than; cc ← $N \cdot V \cdot \overline{Z} + \overline{N} \cdot \overline{V} \cdot \overline{Z}$
LE	Branch on less than or equal; cc ← $Z + N \cdot \overline{V} + \overline{N} \cdot V$
LT	Branch on less than; cc ← $\overline{N} \cdot V + N \cdot \overline{V}$
NE	Branch on not equal; cc ← $\overline{Z}$

CC	Branch on carry clear; $cc \leftarrow \overline{C}$
CS	Branch on carry set; $cc \leftarrow C$
VC	Branch on overflow clear; $cc \leftarrow \overline{V}$
VS	Branch on overflow set; $cc \leftarrow V$
PL	Branch on plus (i.e. positive); $cc \leftarrow \overline{N}$
MI	Branch on minus (i.e. negative); $cc \leftarrow N$

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.

Binary Opcodes:

BT	‘0000’
BF	‘0001’
BHI	‘0010’
BLS	‘0011’
BCC	‘0100’
BCS	‘0101’
BNE	‘0110’
BEQ	‘0111’
BVC	‘1000’
BVS	‘1001’
BPL	‘1010’
BMI	‘1011’
BGE	‘1100’
BLT	‘1101’
BGT	‘1110’
BLE	‘1111’

Condition codes:

N	Z	V	C
-	-	-	-