Advanced Compiler System for Education

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

December 3, 2007

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).

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 is briefly discussed 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 1.1 is shown the entire compilation process. 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. (see1.3).

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 or parser for a given grammar.

At first the SDT analyzes the correctness of a tokenized input provided by the lexer; then it executes some semantic 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 owns a limited set of general-purpose registers, the liveness analysis and a register allocation steps are then performed.

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

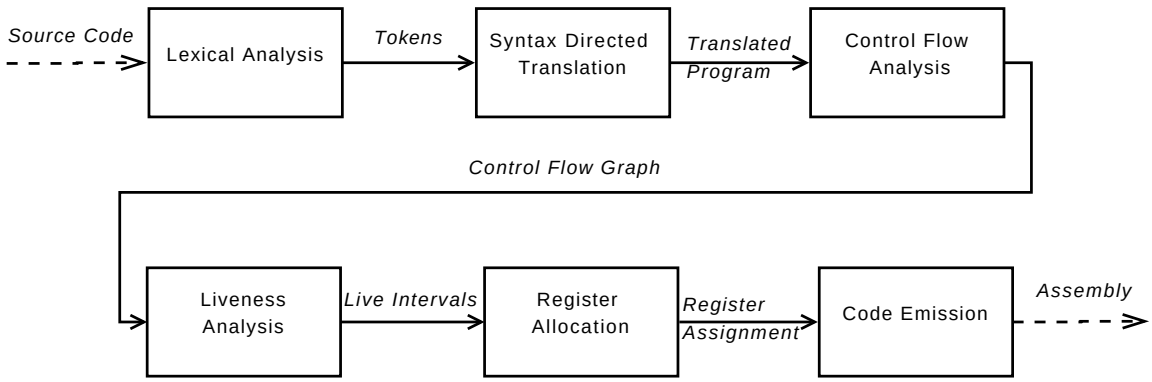

Figure 1.1: Compilation process

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 is provided a grammar for the source code.

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
statements :	*statements statement*
	| *statement*
statement :	IDENTIFIER LSQUARE *exp* RSQUARE ASSIGN *exp* SEMI
	| IDENTIFIER ASSIGN *exp* SEMI
	| *if_statements*
	| *if_statements* ELSE *code_block*
	| *while_statements*
	| *do_while_statements* SEMI
	| SEMI
	| RETURN SEMI
	| READ LPAR IDENTIFIER RPAR SEMI
	| WRITE LPAR *exp* RPAR SEMI
code_block :	*statement*
	| LBRACE *statements* RBRACE
while_statements :	WHILE LPAR *exp* RPAR *code_block*
do_while_statements :	DO *code_block* WHILE LPAR *exp* RPAR
if_statements :	IF LPAR *exp* RPAR *code_block*

```
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
```

Taking the parser generated by Bison starting 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 out to standard output the result is shown in table 1.1.

If the number given as input is negative the program writes to standard output '-1' before exiting.

1.5 Semantic actions

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

```
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

a 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. Then, each assembly instruction is stored inside a data structure called `t_program_infos` 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 element:

- An instance of a symbol table
- An instance of label manager

- A list of program variables
- A list of instruction and assembler directives.

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

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 a file called `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).

For example: an assembly ADD instruction is a ternary instruction that performs a binary add between two values. An instruction that performs a sum between `'R1'` and `'R2'`, and stores the result in `'R3'` can be declared as follows:`ADD R3 R1 R2`

the keyword "`ADD`" is used as an operation identifier. The number of instruction parameters depends only on the instruction type. For example: ternary instructions accept only register identifiers as parameters.

In the previous example the parameters `'R3' 'R1' 'R2'` were 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, a label 'L1' is associated to an assembly ADD instruction.

A 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 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 assigned 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 data type; `100` as initial value; the string "`var`" as 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 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. An entry of the symbol table 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 */
		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 */
		WRITE R4 0	/* write R4 to standard output */
return; }		HALT	/* stop the program execution */
fact = 1;	L2:	ADDI R2 R0 #1	
while(value > 0) {	L3:	SUBI R5 R1 #0	
		SGT R5 0	/* set R3 on 'less than zero' */
		BEQ L4	/* 'branch on equal' to label L4 */
fact = value * fact;		MUL R6 R1 R2	/* binary mult. operation */
		ADDI R2 R6 #0	
value = value - 1; }		SUBI R7 R1 #1	
		ADDI R1 R7 #0	
		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 table for a symbol
- 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 with Bison giving as input the file `Acse.y`.

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

- `reserveLabelID`
- `fixLabelID`.

`reserveLabelID` is used when the user code requires the creation of a new label. `fixLabelID` 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 for 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/store instructions from/to array elements.

An example is the function `loadArrayElement` that takes as input:

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

This function returns as output a register location identifier that holds 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 also 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:

- perform_binary_comparison;
- perform_bin_numeric_op.

The function `perform_bin_numeric_op` is used to generate instructions for binary numeric operations (for example: the expression "a + 5"). `perform_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 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 as input a variable/symbol identifier, the `get_symbol_location` look up the symbol table in search of the register location where the symbol is stored. If the requested variable/symbol was never loaded from memory to a register (i.e. the symbol holds an invalid register location info), this function asks for a free register where to assign 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 example describes 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 is an expression and which 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 represent 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 which values are stored into unknown register locations. At 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.

Finally, we can use the function `gen_andb_instruction` declared in `axe_gencode.h` in order to generate an assembly `ANDB` instruction. A `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 register identifiers as parameters for the `ANDB` instruction;
- A field that is used to specify the addressing mode for both 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 destination register and the second source register are directly addressed. The value `CG_INDIRECT_ALL` is used when both destination register and source register are indirectly addressed. The value `CG_INDIRECT_DEST` is used when the destination register is indirectly addressed. The value `CG_INDIRECT_SOURCE` is used when the second source register is indirectly addressed.

If we want to store result of the ANDB instruction in a new register (i.e. a register that is unused), at first we must call the function `getNewRegister` declared in `axe_engine.h`. Once called, the function `getNewRegister` returns as output a new register identifier. At last, 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 `perform_bin_numeric_op` is used to perform all the operations discussed so far in this section.

The function `perform_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 (for example: the macro `ANDB` defined in `axe_constants.h` should be used as identifier for the previous example);

Valid binary operation identifiers are:

- `ADD`
- `ANDB`
- `ORB`
- `SUB`
- `MUL`
- `DIV`

The function `perform_binary_comparison` can be used to implement the semantic action for the rule `exp` : | *exp* LT *exp*

The function `perform_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 (for example: the macro `_LT_` defined in `axe_constants.h` should be used in this last example);

Valid condition codes 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”).

Both the function perform_bin_numeric_op and the function perform_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 in the following manner:

```
do_while_statements  : DO
                     {
                        $1 = reserveLabel(program);

                        fixLabel(program, $1);
                     }
                     code_block WHILE LPAR exp RPAR
                     {
                        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 a 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 using the non-terminal code_block); otherwise the control get out from the loop.

We can use the function reserveLabel 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 instances of t_axe_label (a structure declared in axe_stuct.h which only field is an integer value).

The function fixLabel declared in axe_engine.h takes as input an instance of t_program_infos and an instance of t_axe_label (the label that

must be assigned to the following assembly statement). We can use the `fixLabel` to assign a label to the first instruction of the loop body.

Note that both the function `reserveLabel` and the function `fixLabel` are defined as wrappers respectively for the `reserveLabelID` and `fixLabelID` (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 describes 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. The assembler takes an assembly as input (i.e. symbolic assembler - 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 at first are interpreted and then 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 does an assembler program works (which translation steps are performed and in which 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 be 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).

An assembly instruction can be directly mapped in an equivalent machine code instruction. However, an assembly contains also `assembler directives` and symbols (i.e. labels) that can't be directly translated into machine code. Usually an assembler directive is used (for example) to notify to the assembler the beginning of a block of data; Labels can be assigned to specific memory locations. 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 (performing a syntactic analysis on the input file and notifying all the encountered errors to the standard error).

As post-condition, only valid assembly files will be translated in "well formed" object file.

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

- The assembler initialize its internal data structures.
- An assembly given as input is parsed: every instruction/directive is validated.
- An object file is written 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 at first are validated (i.e. the consistency of each instruction is checked) and then translated in an intermediate form. Assembler directives are always interpreted and never translated in machine instructions. Assembler directives typically are used to manipulate the content of the data segment once loaded the object file in memory.

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

2.2 Assembly format

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

An instruction specifies an operation type and a list of operands. Operand types can be:

- register identifiers.
- immediate values
- address values

Registers can be directly or indirectly addressed only in ternary instructions. Also in ternary instructions only the destination register and the second source register can be indirectly addressed.

Supported operand types are:

- register identifiers.
- immediate values
- address values

The current section will use the following notation for register identifiers and immediate values

- `Rn` Register 'n'.
- `Rdest` Destination Register.
- `Rsource1` First source operand.
- `Rsource2` Second source operand.
- `(Rn)` Indirect register 'n'.
- `#imm` Immediate value. `imm` is an integer value.

Assembler directives use their own semantic and format. Current implementation uses a very little subset of the `GNU assembler directives`.

2.2.1 Ternary Instructions

Rdest and RSource2 can be directly or indirectly addressed. Here we will give a brief description of every instruction. A complete description of every instruction can be found in the MACE documentation.

ADD	*Add binary*
Syntax:	ADD Rdest RSource1 RSource2
Examples	Semantics
ADD R2 R1 R3	R2 ← R1 + R3
ADD R2 R1 (R3)	R2 ← R1 + [R3]
ADD (R2) R1 (R3)	[R2] ← R1 + [R3]

SUB	*Subtract binary*
Syntax:	SUB Rdest RSource1 RSource2
Examples	Semantics
SUB R2 R1 R3	R2 ← R1 - R3
SUB R2 R1 (R3)	R2 ← R1 - [R3]
SUB (R2) R1 (R3)	[R2] ← R1 - [R3]

ANDL	*AND logical*
Syntax:	ANDL Rdest RSource1 RSource2
Examples	Semantics
ANDL R2 R1 R3	R2 ← R1 && R3
ANDL R2 R1 (R3)	R2 ← R1 && [R3]
ANDL (R2) R1 (R3)	[R2] ← R1 && [R3]

ORL	*OR logical*
Syntax:	ORL Rdest RSource1 RSource2
Examples	Semantics
ORL R2 R1 R3	R2 ← R1 ‖ R3
ORL R2 R1 (R3)	R2 ← R1 ‖ [R3]
ORL (R2) R1 (R3)	[R2] ← R1 ‖ [R3]

EORL	*Exclusive OR logical*
Syntax:	EORL Rdest RSource1 RSource2
Examples	Semantics
EORL R2 R1 R3	R2 ← R1 $\oplus$ R3
EORL R2 R1 (R3)	R2 ← R1 $\oplus$ [R3]
EORL (R2) R1 (R3)	[R2] ← R1 $\oplus$ [R3]

ANDB	*AND bit by bit*
Syntax:	ANDB [Rdest] [RSource1] [RSource2]
Examples	Semantics
ANDB R2 R1 R3	R2 ← R1 & R3
ANDB R2 R1 (R3)	R2 ← R1 & [R3]
ANDB (R2) R1 (R3)	[R2] ← R1 & [R3]

ORB	*OR bit by bit*
Syntax:	ORB Rdest RSource1 RSource2
Examples	Semantics
ORB R2 R1 R3	R2 ← R1 | R3
ORB R2 R1 (R3)	R2 ← R1 | [R3]
ORB (R2) R1 (R3)	[R2] ← R1 | [R3]

EORB	*Exclusive OR bit by bit*
Syntax:	EORB Rdest RSource1 RSource2
Examples	Semantics
EORB R2 R1 R3	R2 ← R1 $\oplus$ R3
EORB R2 R1 (R3)	R2 ← R1 $\oplus$ [R3]
EORB (R2) R1 (R3)	[R2] ← R1 $\oplus$ [R3]

MUL	*MUL binary*
Syntax:	MUL Rdest RSource1 RSource2
Examples	Semantics
MUL R2 R1 R3	R2 ← R1 * R3
MUL R2 R1 (R3)	R2 ← R1 * [R3]
MUL (R2) R1 (R3)	[R2] ← R1 * [R3]

DIV	*DIV binary*
Syntax:	DIV Rdest RSource1 RSource2
Examples	Semantics
DIV R2 R1 R3	R2 ← R1 / R3
DIV R2 R1 (R3)	R2 ← R1 / [R3]
DIV (R2) R1 (R3)	[R2] ← R1 / [R3]

SHR	*Binary Shift to Right*
Syntax:	SHR Rdest RSource1 RSource2
Examples	Semantics
SHR R2 R1 R3	R2 $\leftarrow$ R1 $\gg$ R3
SHR R2 R1 (R3)	R2 $\leftarrow$ R1 $\gg$ [R3]
SHR (R2) R1 (R3)	[R2] $\leftarrow$ R1 $\gg$ [R3]

SHL	*Binary Shift to Left*
Syntax:	SHL Rdest RSource1 RSource2
Examples	Semantics
SHL R2 R1 R3	R2 $\leftarrow$ R1 $\ll$ R3
SHL R2 R1 (R3)	R2 $\leftarrow$ R1 $\ll$ [R3]
SHL (R2) R1 (R3)	[R2] $\leftarrow$ R1 $\ll$ [R3]

ROTL	*Rotate binary*
Syntax:	ROTL Rdest RSource1 RSource2
Semantics:	Rdest $\leftarrow$ RSource1 RSource2
Description:	Actually the `ROTL` instruction is not supported by the current architecture. `RSource1` value is rotated to left by `RSource2` positions.

ROTR	*Rotate binary*
Syntax:	ROTR Rdest RSource1 RSource2
Semantics:	Rdest $\leftarrow$ RSource1 RSource2
Description:	Actually the `ROTR` instruction is not supported by the current architecture. `RSource1` value is rotated to right by `RSource2` positions.

NEG	*Negate*
Syntax:	NEG Rdest RSource1 RSource2
Examples	Semantics
NEG R2 R1 R3	R2 ← - R3
NEG R2 R1 (R3)	R2 ← - [R3]
NEG (R2) R1 (R3)	[R2] ← - [R3]
Note:	RSource1 is unused.

SPCL	*Special opcode*
Syntax:	SPCL Rdest RSource1 RSource2
Semantics:	Undefined at the moment.

2.2.2 Binary Instructions

ADDI	*Add with Immediate operand*
Syntax:	ADDI Rdest RSource1 #Immediate
Example	Semantics
ADDI R2 R1 #VAL	R2 ← R1 + VAL

SUBI	*Subtract with Immediate operand*
Syntax:	SUBI Rdest RSource1 #Immediate
Example	Semantics
SUBI R2 R1 #VAL	R2 ← R1 - VAL

ANDLI	*AND with Immediate operand*
Syntax:	ANDLI Rdest RSource1 #Immediate
Example	Semantics
ANDLI R2 R1 #VAL	R2 ← R1 && VAL

ORLI	*OR with Immediate operand*
Syntax:	ORLI Rdest RSource1 #Immediate
Example	Semantics
ORLI R2 R1 #VAL	R2 ← R1 || VAL

EORLI	*Exclusive OR with Immediate operand*
Syntax:	EORLI Rdest RSource1 #Immediate
Example	Semantics
EORLI R2 R1 #VAL	R2 ← R1 ⊕ VAL

ANDBI	*AND bit by bit with Immediate operand*
Syntax:	ANDBI Rdest RSource1 #Immediate
Example	Semantics
ANDBI R2 R1 #VAL	R2 ← R1 & VAL

ORBI	*OR bit by bit with Immediate operand*
Syntax:	ORBI Rdest RSource1 #Immediate
Example	Semantics
ORBI R2 R1 #VAL	R2 ← R1 | VAL

EORBI	*Exclusive OR bit by bit with immediate operand*
Syntax:	EORBI Rdest RSource1 #Immediate
Example	Semantics
EORBI R2 R1 #VAL	R2 ← R1 ⊕ VAL

MULI	*MUL binary with Immediate operand*
Syntax:	MULI Rdest RSource1 #Immediate
Example	Semantics
MULI R2 R1 #VAL	R2 ← R1 * VAL

DIVI	*DIV binary with Immediate operand*
Syntax:	DIVI Rdest RSource1 #Immediate
Example	Semantics
DIVI R2 R1 #VAL	R2 ← R1 / VAL

SHRI	*Binary Shift to Right*
Syntax:	SHRI Rdest RSource1 #Immediate
Example	Semantics
SHRI R2 R1 #VAL	R2 $\leftarrow$ R1 $\gg$ VAL

SHLI	*Binary Shift to Left*
Syntax:	SHLI Rdest RSource1 #Immediate
Example	Semantics
SHLI R2 R1 #VAL	R2 $\leftarrow$ R1 $\ll$ VAL

ROTLI	*Rotate binary*
Syntax:	ROTLI Rdest RSource1 #Immediate
Example	Semantics
ROTLI R2 R1 #VAL	R2 $\leftarrow$ R1 rotated to left of VAL positions

ROTRI	*Rotate binary*
Syntax:	ROTRI [Rdest] [RSource1] #[Immediate]
Example	Semantics
ROTRI R2 R1 #VAL	R2 $\leftarrow$ R1 rotated to right of VAL positions

NOTL	*Logical complement*
Syntax:	NOT Rdest RSource1 #Immediate
Example:	Semantics
NOTL R2 R1 #VAL	R2 $\leftarrow$! R1
Note:	RSource1 is unused.

NOTB	*Binary complement*
Syntax:	NOTB Rdest RSource1 #Immediate
Example	Semantics
NOTB R2 R1 #VAL	R2 $\leftarrow \sim$ R1

2.2.3 Unary Instructions

NOP	*No Operation*
Syntax:	NOP

MOVA	*Move Address to Register Location*
Syntax:	MOVA RDest Address
Example	Semantics
MOVA R2 L1	R2 $\leftarrow$ L1 (where L1 is a Label)

JSR	*Jump To Subroutine*
Syntax:	JSR RDest Address

Note : not implemented yet.

RET
Syntax:

Note : not implemented yet. Actually an HALT instruction is translated instead of a RET.

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

STORE	*Spill a value*
Syntax:	STORE RSource Address
Example	Semantics
STORE R2 L1	L1 $\leftarrow$ 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 from standard input an integer value*
Syntax:	READ RSource Address
Example	Semantics
READ R2 0	Read from input an 32-bit signed integer value, and store the value to 'R2'.
Note:	`Address` parameter is unused.

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

2.2.4 Jump Instructions

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, branch 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;
BPL	Branch on plus (i.e. positive);
BMI	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 with the 32-bit integer value '5'
`.word 0`	reserve a word location and set its content with the 32-bit integer value '0'

.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.
Examples:	
`.space 8`	5 contiguous bytes reserved
`.space 32`	32 contiguous bytes reserved

2.3 Object file format

An object file is returned as result of the assembler process. It contains both machine code instructions and data information (that will be stored inside the data segment).

In figure 2.1 is shown the prototype of an object file

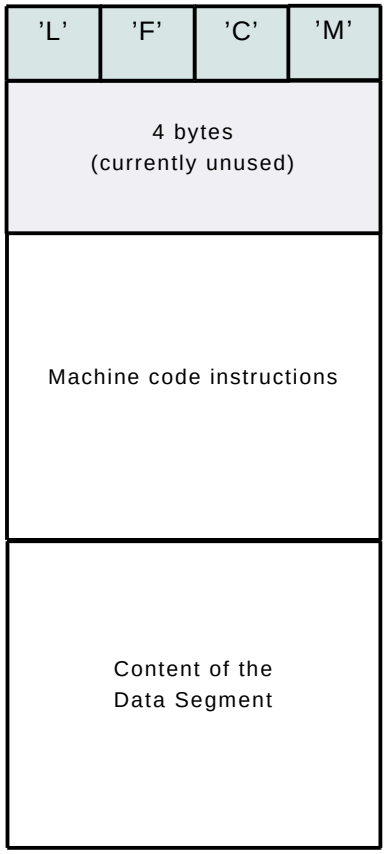

Figure 2.1: Object file format

An object file is composed by an header of 20 bytes followed by the content of the instruction segment and the content of the data segment. The first 4 bytes must always contain the ASCII binary representation for the 'L' 'F' 'C' 'M' characters. The following 16 bytes are actually unused: they are reserved for future uses.

Chapter 3

MACE

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

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

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

3.1 How MACE works

`MACE` execution model is simple. At 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 2Kb is reserved and will contain both code and data segment
- Machine code is loaded from the object file into the code segment
- Data is loaded from the object file into the data segment

Once 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 program counter (`PC`)
  - update the value of the status register (`PSW`)
- Until an `HALT` instruction is encountered.

3.2 Architecture

In figure 3.1 is shown the architectural design of the machine simulated by MACE

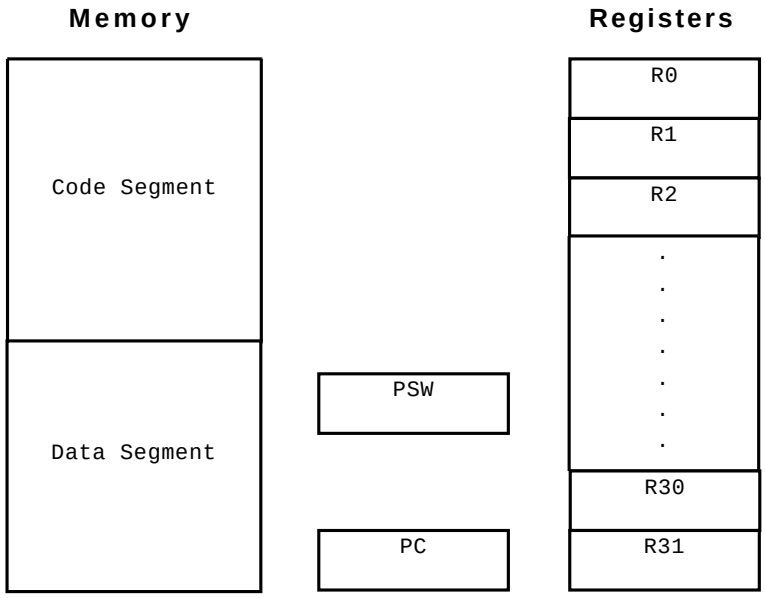

Figure 3.1: Architectural Design

The architecture is composed by:

- 32 General-Purpose 32-Bit registers.
- 32-Bit Program Counter (`PC`)
- 32-Bit Status Register (`PSW`)

3.2.1 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 always 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).

3.2.2 Program Counter

The `PC` contains the address of the instruction currently executing. During instruction execution, the processor automatically increments the content or places a new value in the `PC`.

3.2.3 Status Register

The Status Register (`PSW`) is a 32-Bit register, but only his four lower bits are available in the 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 first four bits represent a condition of the result generated by an operation. In the instruction set definitions, the `PSW` is illustrated as in figure 3.2.

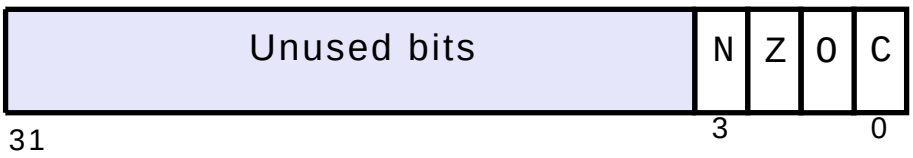

Figure 3.2: Status Register

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 operand size.

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

3.3 Addressing Capabilities

Most operations take a source operand and destination operand, compute them, and store the result in the destination location. Single-operand operations take a destination operand,compute it, and store the result in the destination location. 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.

3.3.1 Instruction Format

Instructions consist of exactly one word (i.e. 32-bit). 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:

- register identifiers.
- immediate values
- address values

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

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

Registers can be either directly or indirectly addressed only in ternary instructions. However, only the destination register `RDEST` and the second source register `RSOURCE2` can be indirectly addressed.

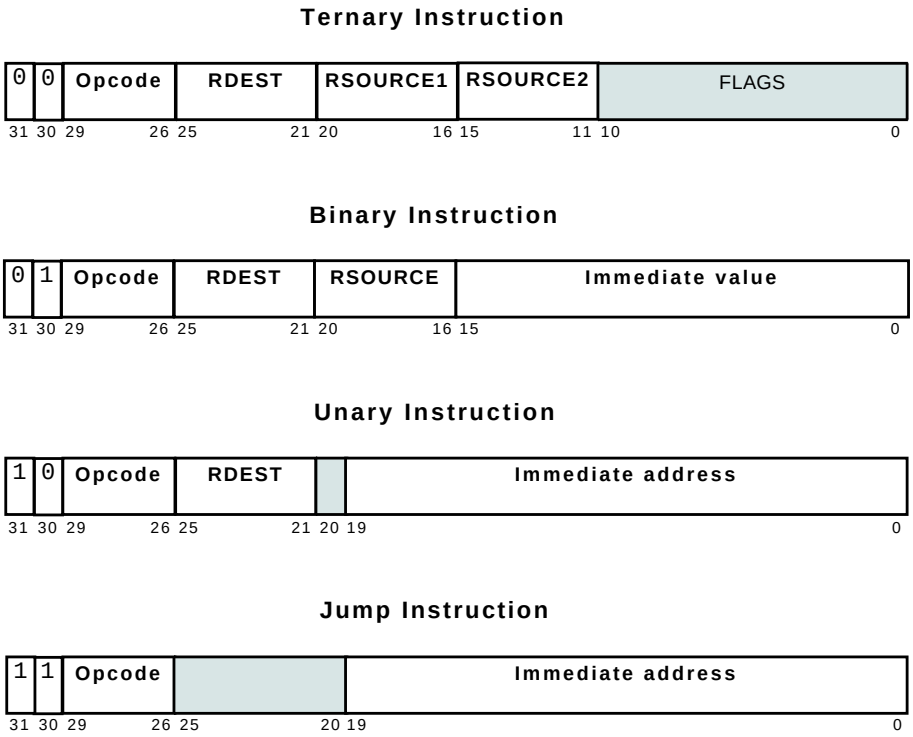

Figure 3.3: Instruction Formats

In ternary instructions these information are encoded inside the `flag bits`(the 11 less significant bits of the instruction word). Flag bits format is shown in figure 3.4.

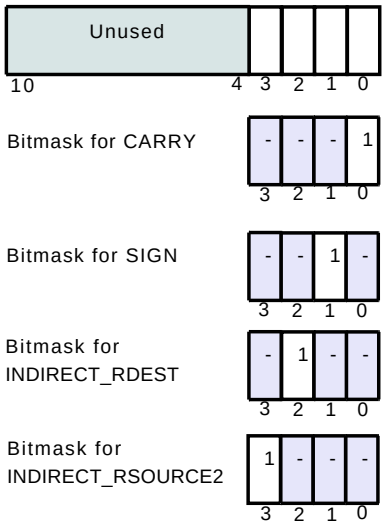

Figure 3.4: Function bits

If the bit 'CARRY' is set to 1, the result of the binary operation between RSOURCE1 and RSOURCE2 is incremented by 1.

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.

In figure 3.5 is shown an example for a `''ADD (R3) R1 (R2)''` instruction.

3.4 Instruction Set

Instructions are listed by mnemonic. 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 may take.

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)

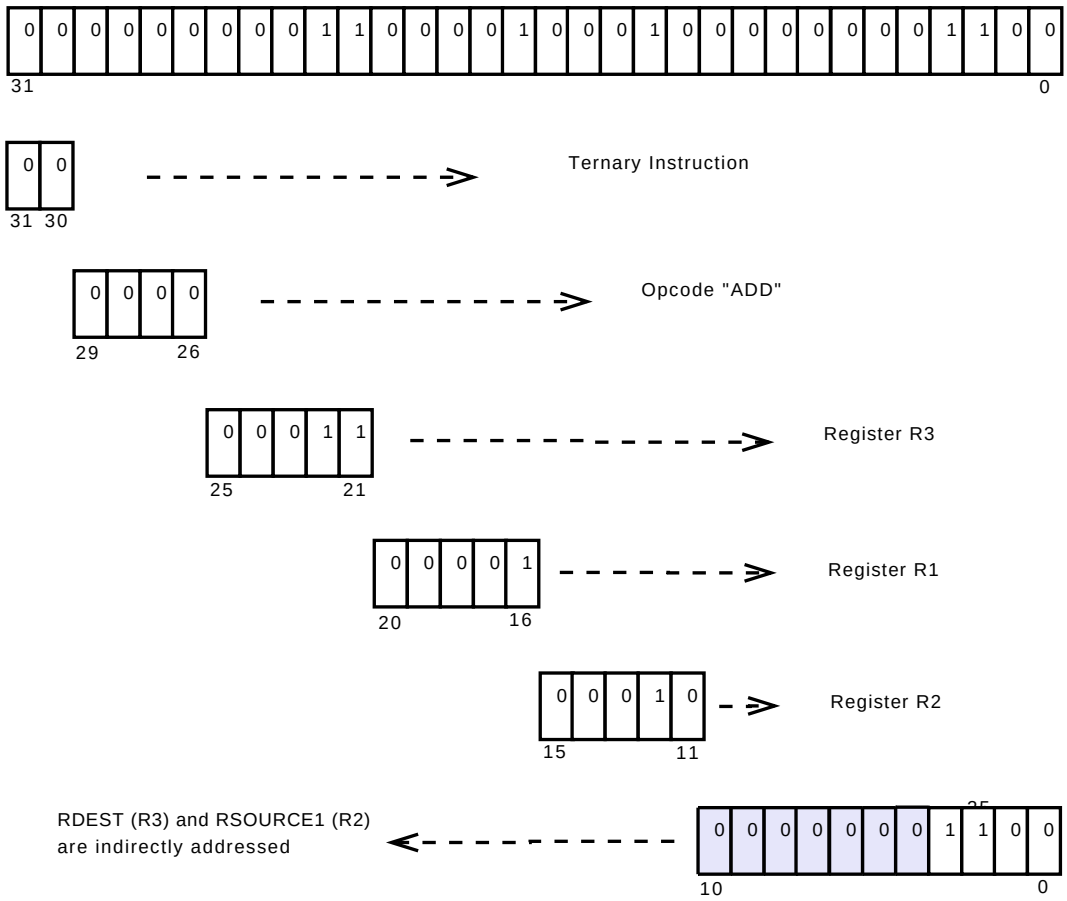

Figure 3.5: An example of encoded ADD instruction

- `-` The bit remains unchanged by the execution of the instruction
- `*` The bit is set or cleared according to the outcome of the instruction.

the legal source and destination addressing modes are specified by their assembly language syntax. The following notation is used when we refers to registers and immediate values:

- `Rn` A Register location.
- `imm` Immediate value which may be 16 bit or 32 bits, depending on the instruction.

3.4.1 Ternary Instructions

ADD	*Add binary*
Syntax:	ADD [Rdest] [RSource1] [RSource2]
Semantics:	[Rdest] ← [RSource1] + [RSource2]
Binary Opcode:	'0000'
Description:	Add the source operand 'Rsource1' to 'Source2' and store the result in the destination location 'Rdest'.
Condition codes:	`N Z V C` `* * * *`

SUB	*Subtract binary*
Syntax:	SUB [Rdest] [RS
Semantics:	[Rdest] ← [RSou
Binary Opcode:	'0001'
Description:	Subtract the 'Rsource1' to the result in th 'Rdest'.
Condition codes:	`N Z V C` `* * * *`

ANDL	*AND logical*
Syntax	ANDL [Rdest] [RSource1] [RSource2]
Semantics	[Rdest] ← [RSource1].[RSource2]
Binary Opcode:	'0010'
Description:	Performs an AND between the source operand 'Rsource1' and the 'Source2' operand and store the result in the destination location 'Rdest'.
Condition codes:	`N Z V C` `* * 0 0`

ORL	*OR logical*
Syntax	ORL [Rdest] [RS
Semantics:	[Rdest] ← [RSou
Binary Opcode:	'0011'
Description:	Performs an OR operand 'Rsourc operand and st destination loca
Condition codes:	`N Z V C` `* * 0 0`

EORL	*Exclusive OR logical*
Syntax	EORL [Rdest] [RSource1] [RSource2]
Semantics:	[Rdest] ← [RSource1] ⊕ [RSource2]
Binary Opcode:	'0100'
Description:	EOR (exclusive or) the source operand 'Rsource1' with 'Rsource2' and store the result in the destination location 'Rdest'.
Condition codes:	`N Z V C` `* * 0 0`

ANDB	*AND bit by bit*
Syntax	ANDB [Rdest] [
Semantics	[Rdest] ← [RSou
Binary Opcode:	'0101'
Description:	ANDB the sour to the 'Rsource the result in th 'Rdest'.
Condition codes:	`N Z V C` `* * 0 0`

ORB	*OR bit by bit*
Syntax	ORB [Rdest] [RSource1] [RSource2]
Semantics:	[Rdest] ← [RSource1] | [RSource2]
Binary Opcode:	'0110'
Description:	ORB the source operand 'Rsource1' to the 'Rsource2' operand, and store the result in the destination location 'Rdest'.
Condition codes:	`N Z V C` `* * 0 0`

EORB	*Exclusive OR bi*
Syntax	EORB [Rdest] [
Semantics:	[Rdest] ← [RSou
Binary Opcode:	'0111'
Description:	EORB (exclusi operand 'Rsc 'Rsource2' oper result in the 'Rdest'.
Condition codes:	`N Z V C` `* * 0 0`

MUL	*MUL binary*
Syntax	MUL [Rdest] [RSource1] [RSource2]
Semantics:	[Rdest] ← [RSource1] * [RSource2]
Binary Opcode:	'1000'
Description:	Multiply the 32-bit 'RSource1' operand by the 32-bit 'RSource2' operand and store the result in 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] [RS
Semantics:	[Rdest] ← [RSou
Binary Opcode:	'1001'
Description:	Divide the 32-bi by the 32-bit 'R store the resul 'Rdest'. Both t nation are 32-bi
Condition codes:	N Z V C * * * *

SHR	*Binary Shift to Right*
Syntax	SHR [Rdest] [RSource1] [RSource2]
Semantics:	[Rdest] ← [RSource1] ≫ [RSource2]
Binary Opcode:	'1011'
Description:	SHR Performs a binary 'shift to right' on the 32-bit 'RSource1' operand. The number of bits shifted is stored into the 32-bit 'RSource2' operand. The result of the shift operation is stored in the destination register 'Rdest'.
Condition codes:	N Z V C * * * *

SHL	*Binary Shift to*
Syntax	SHL [Rdest] [RS
Semantics:	[Rdest] ← [RSou
Binary Opcode:	'1010'
Description:	SHL Performs a on the 32-bit 'RS number of bits the 32-bit 'RSo result of the shi in the destinatio
Condition codes:	N Z V C * * * *

ROTL	*Rotate binary*
Syntax	ROTL [Rdest] [RSource1] [RSource2]
Semantics:	[Rdest] ← [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. Actually the ROTL instruction is not supported by the current architecture; it is translated as a NOP operation
Condition codes:	`N Z V C` `* * 0 *`

ROTR	*Rotate binary*
Syntax	ROTR [Rdest] [
Semantics:	[Rdest] ← [RS <[RSource2]> t
Binary Opcode:	'1101'
Description:	Rotate the bits right. A rotate the sense that one end is shifte That is, no bit a rotate. Actual tion is not supp architecture; it operation
Condition codes:	`N Z V C` `* * 0 *`

NEG	*Negate*
Syntax	NEG [Rdest] [RSource1] [RSource2]
Semantics:	[Rdest] ← 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] [F
Semantics:	Undefined at th
Binary Opcode:	'1111'
Description:	Will be used for
Condition codes:	`N Z V C` `* * * *`

3.4.2 Binary Instructions

ADDI	*Add with Immediate operand*
Syntax:	ADDI [Rdest] [RSource1] #[Immediate]
Semantics:	[Rdest] ← [RSource1] + #[Immediate]
Binary Opcode:	'0000'
Description:	Add the source operand 'Rsource1' to the 'immediate' value and store the result in the destination location 'Rdest'.
Condition codes:	N Z V C * * * *

SUBI	*Subtract with Im*
Syntax:	SUBI [Rdest] [ate]
Semantics:	[Rdest] ← [RSou
Binary Opcode:	'0001'
Description:	Subtract the 'Rsource1' to t and store the res 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 an AND between the source operand 'Rsource1' and the 'Immediate' operand and store the result in the destination location 'Rdest'.
Condition codes:	N Z V C * * 0 0

ORLI	*OR with Immed*
Syntax	ORLI [Rdest] [ate]
Semantics:	[Rdest] ← [RSou
Binary Opcode:	'0011'
Description:	Performs an OR operand 'Rsourc ate' operand and destination loca
Condition codes:	N Z V C * * 0 0

EORLI	*Exclusive OR with Immediate operand*
Syntax	EORLI [Rdest] [RSource1] #[Immediate]
Semantics:	[Rdest] ← [RSource1] ⊕ #[Immediate]
Binary Opcode:	'0100'
Description:	EOR (exclusive or) the source operand 'Rsource1' with 'Immediate' and store the result in the destination location 'Rdest'.
Condition codes:	N Z V C * * 0 0

ANDBI	*AND bit by operand*
Syntax	ANDBI [Rdest] ate]
Semantics	[Rdest] ← [RSou
Binary Opcode:	'0101'
Description:	ANDBI the sour to the 'Immedia the result in th 'Rdest'.
Condition codes:	N Z V C * * 0 0

ORBI	*OR bit by bit with Immediate operand*
Syntax	ORBI [Rdest] [RSource1] #[Immediate]
Semantics:	[Rdest] ← [RSource1] | #[Immediate]
Binary Opcode:	'0110'
Description:	ORBI the source operand 'Rsource1' to the 'Immediate' operand, and store the result in the destination location 'Rdest'.
Condition codes:	N Z V C * * 0 0

EORBI	*Exclusive OR bit operand*
Syntax	EORBI [Rdest] ate]
Semantics:	[Rdest] ← [RSou
Binary Opcode:	'0111'
Description:	EORBI (exclus operand 'Rsourc ate' operand and destination loca
Condition codes:	N Z V C * * 0 0

MULI	*MUL 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 in 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` `* * * *`

DIVI	*DIV binary with*
Syntax	DIVI [Rdest] [ate]
Semantics:	[Rdest] ← [RSou
Binary Opcode:	'1001'
Description:	Divide the 32-bi by 'Immediate' result in the dest the sources and word values exc immediate value long.
Condition codes:	`N Z V C` `* * * *`

SHRI	*Binary Shift to Right*
Syntax	SHRI [Rdest] [RSource1] #[Immediate]
Semantics:	[Rdest] ← [RSource1] ≫ #[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 in the destination register 'Rdest'.
Condition codes:	`N Z V C` `* * * *`

SHLI	*Binary Shift to*
Syntax	SHLI [Rdest] [ate]
Semantics:	[Rdest] ← [RSo ate]
Binary Opcode:	'1010'
Description:	SHL Performs a on the 32-bit 'RS number of bits s value of the Im mediate'. The re ation is stored i ister '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 Actually the ROTLI instruction is not supported by the current architecture; it is translated as a NOP operation
Condition codes:	`N Z V C` `* * 0 *`

ROTRI	*Rotate binary*
Syntax	ROTRI [Rdest] [ate]
Semantics:	[Rdest] ← [RS <#[Immediate]>
Binary Opcode:	'1101'
Description:	Rotate the bits right. A rotate o the sense that t one end is shifte That is, no bit i a rotate Actuall tion is not supp architecture; it i 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 complem*
Syntax	NOTB [Rdest] [ate]
Semantics:	[Rdest] ← ∼[RS Immediate is un
Binary Opcode:	'1110'
Description:	Perform a binar the value of 'RS stored into 'RDe
Condition codes:	`N Z V C` `* * 0 0`

3.4.3 Unary Instructions

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 when we works on address (pointers) or arrays.
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. Actually this opcode is not supported.
Condition codes:	`N Z V C` `* * * *`

RET	*Return from Subroutine*
Description:	RET is not supported by the current architecture. Actually a RET instruction is translated as a HALT instruction.
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] ← *[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] ← [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] ← 1; ELSE [Rdest] ← 0.
Note:	Condition 'cc' is computed starting from the value of the `PSW` register. For example: 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

GE

GT

LE

LT

NE

Description:

Binary Op-codes:	
SEQ	'0111'
SGE	'1000'
SGT	'1001'
SLE	'1010'
SLT	'1011'
SNE	'1100'
Condition codes:	`N Z V C` `0 * 0 0`

WRITE	*Write to standard output an integer value*
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 to standard output a 32-bit value. 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` `- - - -`

READ	*Read from star value*
Syntax	READ [RSourc
Semantics:	Read from inpu teger value, ar 'RSource'. Add
Binary Opcode:	'1101'
Description:	This instructio from standard Actually the RI plemented with declaration of ' header file of th
Condition codes:	`N Z V C` `* * * *`

3.4.4 Jump Instructions

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: instruction 'BEQ Label' perform a branch to the instruction labeled 'label' 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 (PSW). For example: C refers to the carry bit of the PSW register.

Possible values for 'cc':

EQ	set on equal; cc ← Z
GE	set on greater than or equal; cc ← N.V + $\overline{N}.\overline{V}$
T	branch always. The branch is always 'taken';
F	This condition is never verified. Thus, branch like this are never 'taken';
HI :	Branch on higher than cc ← $\overline{C}.\overline{Z}$
LS	Branch on lower than or same; cc ← C.Z
GT	set on greater than; cc ← N.V.$\overline{Z}$ + $\overline{N}.\overline{V}.\overline{Z}$

LE set on less than or equal;
cc $\leftarrow$ Z + N.$\overline{V}$ + $\overline{N}$.V

LT set on less than;
cc $\leftarrow$ $\overline{N}$.V + N.$\overline{V}$

NE set on not equal;
cc $\leftarrow$ $\overline{Z}$

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

BPL Branch on plus (i.e. positive);
cc $\leftarrow$ $\overline{N}$

BMI 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
- - - -
```