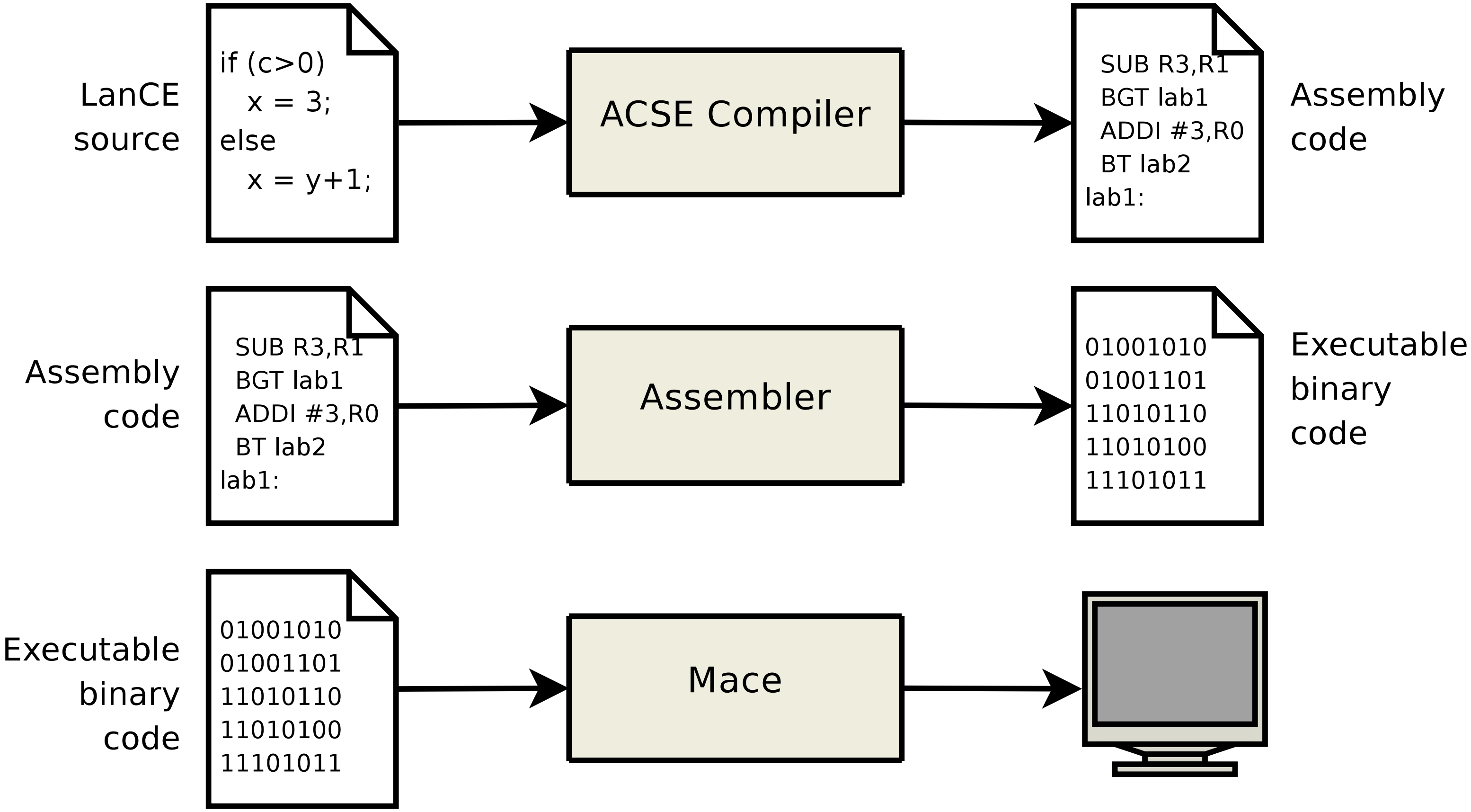
ACSE: Advanced Compiler System for Education

Giovanni Agosta and Andrea Di Biagio

Summary

- The ACSE compiler
- The Assembler
- The MACE architecture

Overview

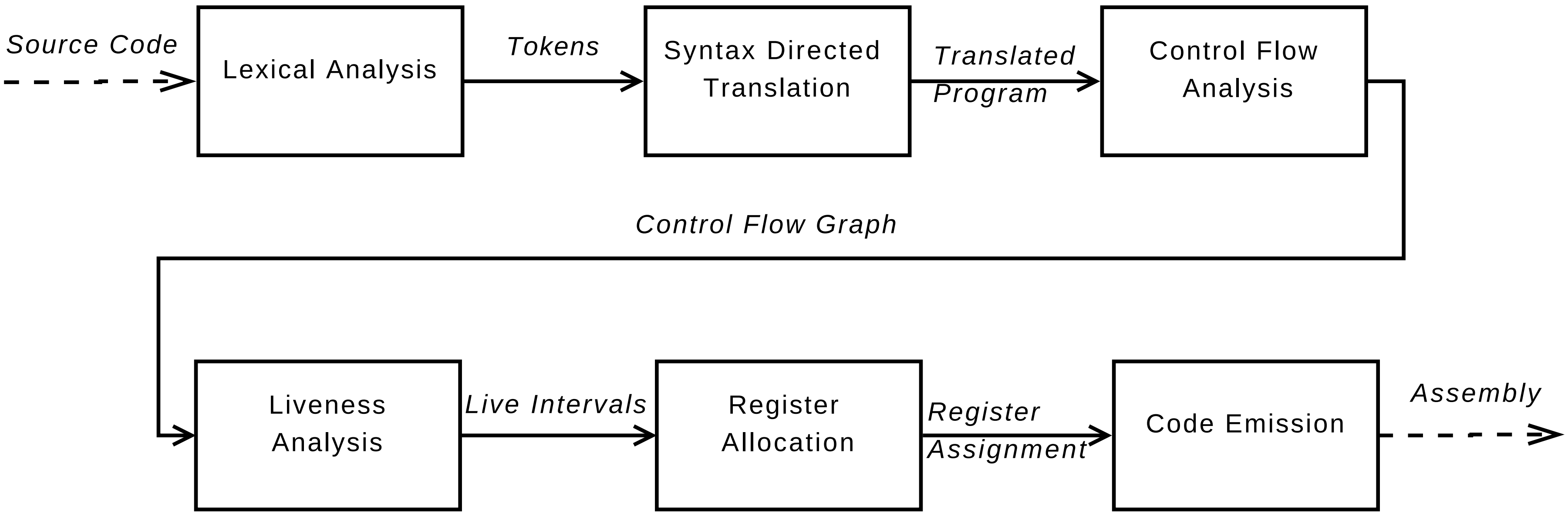

LanCE
source
if (c>0)
x = 3;
else
x = y+1;
ACSE Compiler
SUB R3,R1
BGT lab1
ADDI #3,R0
BT lab2
lab1:
Assembly
code
Assembly
code
SUB R3,R1
BGT lab1
ADDI #3,R0
BT lab2
lab1:
Assembler
01001010
01001101
11010110
11010100
11101011
Executable
binary
code
Executable
binary
code
01001010
01001101
11010110
11010100
11101011
Mace

ACSE

Overview

- The Advanced Compiler System for Education
- Compilation process
- The Language for Compiler Education
- Modules and libraries

Compilation process



Compilation process: Front-End

- Source file tokenized by the *scanner* (or lexer)
  - The scanner is generated using `Flex`
- String of tokens processed by the *parser*
  - ACSE parser: LALR(1), generated with `Bison`
- Syntax Directed Translation
  - Check the tokenized input for syntactic correctness
  - Execute semantic actions for each recognized grammar rule

Compilation process: Back-End

- Transform tokenized input to assembly statements for the target machine
  - The assembly produced uses an unlimited number of registers!
- Liveness Analysis and Register Allocation steps
  - The assembly produced now uses a limited number of registers!
- The assembly code is now written out
- The Assembler must now be invoked to produce an object file

Source Language: LanCE

- The Language for Compiler Education (LanCE)
- Simple C-based syntax:
  - Standard set of arithmetic/logic operations
  - Reduced control flow (while/do-while, if)
  - A single type: integer

LanCE: Tokens

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	IDENTIFIER	{ID}
NUMBER	{DIGIT}+		

`LanCE`: Grammar (1)

`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

LanCE: Grammar (2)

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

LanCE: Grammar (3)

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*

LanCE: Example

```
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 stdout */
```

ACSE SDT: Overview

- Goal: gather all useful information about each statement of the source program
- Each instruction is typically translated into one or more assembly instructions
- Global compiler environment `t_program_infos`:
  - Used to store the collected information, including the assembly instructions
  - Defined in **`axe_engine.h`**.
- Data structures for instructions and directives are defined in **`axe_struct.h`**

ACSE SDT: Program Information

`t_program_infos` contains:

- An instance of a symbol table
- An instance of label manager
- A list of program variables
- A list of instruction and assembler directives.

Assembly instructions

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).

Assembly instructions: Example

- Assembly ADD instruction: ternary instruction
  `ADD R3 R1 R2`
  - Sum `‘R1’` and `‘R2’`
  - Stores the result in `‘R3’`
  - The keyword “`ADD`” identifies the operation
- Number of parameters depends on instruction type
  - Ternary instructions accept only registers identifiers
  - Register identifier: alias for a machine general-purpose register
- Additional information:
  - User comments: for debugging purposes
  - Labels: `L1:   ADD R3 R1 R2`

Assembly directives

A Assembler directive is defined as follows:

- Directive type identifier (e.g.: `.WORD`)
- Value associated with the directive
- Optional label identifier

Example of assembler directive: `.WORD 0`

Axe Variables

- “Variable” item: `t_axe_variable`
- Symbol Table: within `t_program_infos`

`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;

Axe Variable example

- Source program: variable “`var`” as integer with initial value ‘100’
  - `int var = 100;`
- `t_axe_variable` for “var”:
  - `INTEGER` as data type;
  - `100` as initial value;
  - the string “`var`” as variable identifier.
- `t_axe_variable` instance used to produce `.WORD` and/or `.SPACE` assembler directives
  - The `t_axe_variable` defined for “var” produces: `.WORD 100`

Symbol Table

- *Symbol table*: a data structure used at translation time to keep track of source program variables.
- The Symbol Table contains, for each variable
  - ID: variable identifier
  - Type: data type of the variable
  - A Register identifier.
- The register identifier refers to the register location where the variable is currently stored.

Symbol Table: Example

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	multiplication: 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

Symbol Table: Example

Content of the Symbol Table for the example:

Variable Identifier	Type	Register Location
value	INTEGER	R1
fact	INTEGER	R2

ACSE API: Symbol Table

- **`symbol_table.h`**: functions to manipulate the symbol table
  - look up a symbol
  - define and insert a new symbol
  - set the register location information of a symbol
  - retrieve the register location associated with a symbol
- **`axe_struct.h`**: data structures used by the parser (defined in **`Acse.y`**)

ACSE API: Label Manager

- Label Manager (**`axe_labels.h`**): functions to work with labels
  - `newLabelID`: user code requires the creation of a new label
  - `assignLabelID`: assign a given label to an instruction

ACSE API: Code Generation

- `axe_gencode.h`: functions to generate assembly instructions
- E.g.: `gen_add_instruction` is used to create an ADD
- `axe_array.h`: functions to generate load/store instructions from/to array elements
- E.g.: `loadArrayElement` that takes as input:
  - Input: an array variable identifier
  - Input: an array subscript identifying an array element.
  - Output: a register location identifier that holds the value of the specified array element

ACSE API: Miscellanea

`axe_engine.h` defines the `t_program_infos` data structure, plus functions to

- initialize an instance of `t_program_infos`
- add an assembly instruction to a `t_program_infos`
- create a variable and assign it to a `t_program_infos`
- request for a free register location (`getNewRegister`)
- write an assembly file as output (`writeAssembly`)

ACSE API: Expressions

- `axe_structs.h`: expression type `t_axe_expression`, with the following fields:
  - A value: a register identifier or an immediate value
  - An expression type: either “register” or “immediate value”
- Functions used to generate instructions for expressions:
  - `handle_binary_comparison`: comparison expressions
  - `handle_bin_numeric_op`: arithmetic expressions
  - Both take two expressions as input and return as expression
- `axe_utils.h`
  - `get_symbol_location`: wrapper for the functions in `symbol_table.h`
  - Look up the register for a variable, if not present get a new one

Semantic Actions Examples

Three examples of bison semantic actions:

- Arithmetic expression
- Comparison expression
- `do-while` statement

Arithmetics Expressions (1)

- LanCE Grammar rule:

  exp :

  | *exp* AND_OP *exp*

- Source code: “`a & b`”

- Semantic action: `$$ = handle_bin_numeric_op (program, $1, $3, ANDB);`

Arithmetics Expressions (2)

- Query the symbol table to retrieve the register locations associated with both a and b.
  - `get_symbol_location`
- Generate an assembly `ANDB` instruction
  - `gen_andb_instruction`
- `handle_bin_numeric_op` can be used to perform all these operations

Arithmetics Expressions (3)

`gen_andb_instruction` requires five parameters:

- Pointer to `t_program_infos`;
- A destination register identifier;
- Two register identifiers as parameters for the `ANDB` instruction;
- The addressing mode:
  - `CG_DIRECT_ALL`
  - `CG_INDIRECT_ALL`
  - `CG_INDIRECT_DEST`
  - `CG_INDIRECT_SOURCE`

Arithmetics Expressions (4)

- handle_bin_numeric_op takes the following parameters:
  - Pointer to t_program_infos
  - Two t_axe_expression, one for each operand
  - An operation identifier (e.g.: ANDB)
- Valid binary operation identifiers:
  - ADD
  - ANDB
  - ORB
  - SUB
  - MUL
  - DIV

Comparison Expressions (1)

- LanCE Grammar rule:

  exp :

  $|$ *exp* LT *exp*

- Source code: “`a < b`”
- Semantic action: `$$ = handle_binary_comparison (program, $1, $3, _LT_);`

Comparison Expressions (2)

- `handle_binary_comparison` is used to implement the semantic action
- `handle_binary_comparison` takes the following parameters:
  - Pointer to `t_program_infos`
  - Two `t_axe_expression`, one for each operand
  - A condition code (e.g.: `_LT_`)
- Valid condition codes:
  - `_LT_`
  - `_GT_`
  - `_EQ_`
  - `_NOTEQ_`
  - `_LTEQ_`
  - `_GTEQ_`

Do-while Statement (1)

```
do_while_statements  : DO
          {
            $1 = newLabel(program);
            assignLabel(program, $1);
          }
          code_block WHILE LPAR exp RPAR
          {
            gen_bne_instruction (program, $1, 0);
          };
```

Do-while Statement (2)

- Assign a label to the loop body first instruction via `newLabel` and `assignLabel`
- Check if `exp` is different from zero
  - If so, the control jumps back to the first instruction of the loop body (`code_block`)
  - Otherwise the control get out from the loop.
- Use `gen_bne_instruction` to generate a conditional branch instruction

Assembler

Assembler workflow

- Initialize internal data structures
- Check input code for syntactic errors
- Translate labels to memory addresses
- Translate assembly instruction to machine code
- Write out the object file

Assembly format

- Format of an assembly source file:

  `.DATA`
  Data directive no. 1
  . . .
  Data directive no. n

  `.TEXT`
  First instruction
  . . .
  Last instruction

- Format of a line:
  [*Label* `:`] *Instruction or directive* [`/*` *comment* `*/`]

Assembly format (2)

- An instruction specifies an operation type and a list of operands.
- Operand types:
  - register identifiers
  - immediate values
  - address values
- In ternary instructions the destination register and the second source register can be indirectly addressed.

Assembly format: Notation

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

Instructions and Directives

- Assembler directives are a subset of the *GNU assembler directives*
- Instructions come in four formats:
    - Ternary instructions
    - Binary instructions
    - Unary instructions
    - Jump instructions

Ternary Instructions

- 3 (2 sources + 1 destination) register operands
- Direct or indirect addressing for second source and destination operands; direct addressing for first source operand

Arithmetic: ADD, SUB, MUL, DIV, NEG

Logical: ANDL, ORL, EORL

Bitwise: ANDB, ORB, EORB

Shift: SHR, SHL

Rotate: ROTR, ROTL

Ternary Instructions: Arithmetic

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]

Arithmetic Instructions: ADD, SUB, MUL, DIV

Ternary Instructions: NEG

NEG	*Negate*
Syntax:	NEG Rdest Rsource1 Rsource2
Examples	Semantics
NEG R3 R1 R2	R3 ← - R2
NEG R3 R1 (R2)	R3 ← - [R2]
NEG (R3) R1 (R2)	[R3] ← - [R2]

- Rsource1 is ignored.

Ternary Instructions: Logical & Bitwise

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]

Logical: ANDL, ORL, EORL

Bitwise: ANDB, ORB, EORB

Ternary Instructions: Shift & Rotate

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

Shift: SHR, SHL

Rotate: ROTR, ROTL

- ROTR and ROTL not currently implemented in MACE.

Binary Instructions

- 2 (1 source + 1 destination) register operands + 1 16-bit immediate operand
- Only direct addressing for register operands

Arithmetic: ADDI, SUBI, MULI, DIVI

Logical: ANDLI, ORLI, EORLI, NOTL

Bitwise: ANDBI, ORBI, EORBI, NOTB

Shift: SHRI, SHLI

Rotate: ROTRI, ROTLI

Binary Instructions: Arithmetic

ADDI	*Add with Immediate operand*
Syntax:	ADDI Rdest Rsource1 #Immediate
Example	Semantics
ADDI R2 R1 #1	R2 ← R1 + 1

Arithmetic Instructions: ADDI, SUBI, MULI, DIVI

Binary Instruction: Logical & Bitwise

ANDLI	*AND with immediate operand*
Syntax:	ANDLI Rdest Rsource1 #Immediate
Example	Semantics
ANDLI R2 R1 #1	R2 ← R1 && 1

Logical: ANDLI, ORLI, EORLI

Bitwise: ANDBI, ORBI, EORBI

Binary Instructions: NOTL and NOTB

NOTL	*Logical complement*
Syntax:	NOT Rdest Rsource1 #Immediate
Example:	Semantics
NOTL R2 R1 #0	R2 ← ! R1

NOTB	*Binary complement*
Syntax:	NOTB Rdest Rsource1 #Immediate
Example	Semantics
NOTB R2 R1 #0	R2 ← $\sim$ R1

- The third operand is ignored.

Binary Instructions Shift & Rotate

SHRI	*Binary Shift to Right*
Syntax:	SHRI Rdest Rsource1 #Immediate
Example	Semantics
SHRI R2 R1 #3	R2 $\leftarrow$ R1 $\gg$ 3

Shift SHRI, SHLI

Rotate ROTRI, ROTLI

- ROTRI and ROTLI not currently implemented in MACE.

Unary Instructions

- 1 (destination) register operand
  + 1 20-bit address operand
- Only direct addressing for the register operand

Special: NOP, HALT

Load/Store: MOVA, LOAD, STORE

Conditional: Scc

I/O: READ, WRITE

Unary Instructions: NOP, HALT, and MOVA

NOP	*No operation*
Syntax:	NOP

HALT	*Halt the machine processor*
Syntax:	HALT

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

Unary Instructions: LOAD and STORE

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

STORE	*Save a register into memory*
Syntax:	STORE Rsource Address
Example	Semantics
STORE R2 L1	[L1] ← R2 (where L1 is a Label)

Unary Instructions: Scc

Scc	*Set according to condition ‘cc’*
Syntax:	Scc Rdest Address
Semantics:	IF cc == 1 THEN Rdest ← 1; ELSE Rdest ← 0.
Example: SGT R2 0	Set the value of R2 to 1 if the condition GT is verified, to 0 otherwise.
Note:	`Address` parameter is unused.

Possible values for ‘cc’:

EQ	equal;	**LE**	less than or equal;
GE	greater than or equal;	**LT**	less than;
GT	greater than;	**NE**	not equal;

Binary Instructions: READ and WRITE

READ	*Read an integer from standard input*
Syntax:	READ Rsource Address
Example	Semantics
READ R2 0	Read from standard input a 32-bit signed integer value, and store the value into R2.

WRITE	*Write an integer to standard output*
Syntax:	WRITE Rsource Address
Example	Semantics
WRITE R2 0	Write to standard output the 32-bit signed integer value stored in R2.

- Address parameter is ignored.

Jump Instructions

Bcc	*Branch on condition cc*
Syntax:	Bcc Label
Semantics:	IF cc == 1 THEN jump to label `Label`.
Description:	‘cc’ is tested: if the condition is true, the PC is set to the label address.
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”

Condition Codes (1)

EQ	Branch on equal;
GE	Branch on greater than or equal;
T	Branch always;
F	Never branch;
HI	Branch on higher than
LS	Branch on lower than or same;
GT	Branch on greater than;
LE	Branch on less than or equal;

Condition Codes (2)

LT	Branch on less than;
NE	Branch on not equal;
CC	Branch on carry cleared;
CS	Branch on carry set;
VC	Branch on overflow cleared;
VS	Branch on overflow set;
BPL	Branch on plus (i.e. positive);
BMI	Branch on minus (i.e. negative);

Assembler Directives

.data Beginning of a block of data directives

.text Beginning of a block of instructions

.word Reserve and set a memory word (32-bit) in the data segment

.space Reserve a given number of bytes into the data segment

Assembler Directives: *.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, set its content to 5
`.word 0`	reserve a word, set its content to 0

Assembler Directives: ***.space***

Reserve (but do not initialize) a given number of bytes into the data segment

Syntax:	`.space` VAL
Semantics:	Reserve `VAL` (contiguous) bytes inside the data segment.

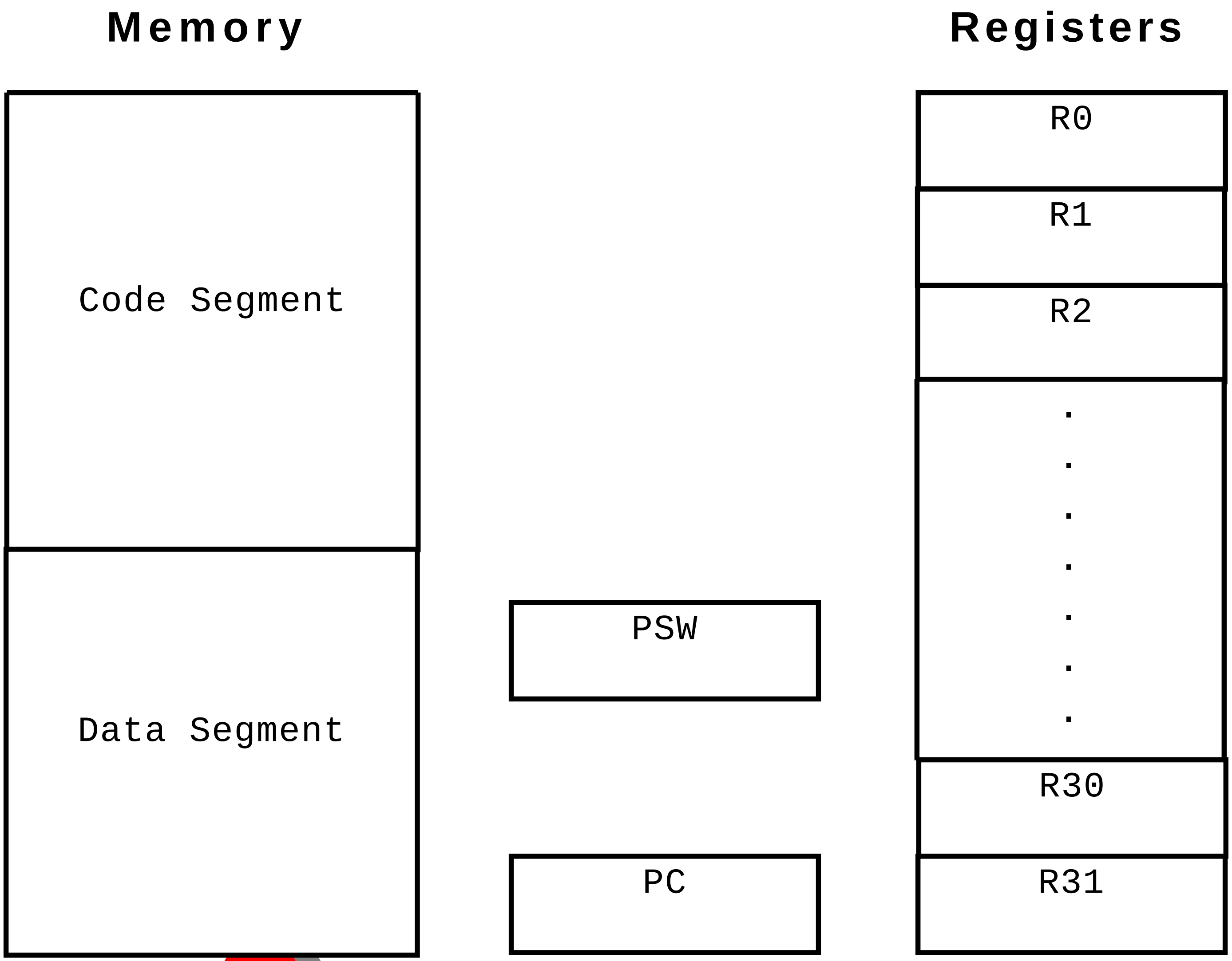
Examples:

`.space 8`	8 contiguous bytes reserved
`.space 32`	32 contiguous bytes reserved

- Memory is kept aligned to 4-byte boundaries.

MACE

MACE Architecture

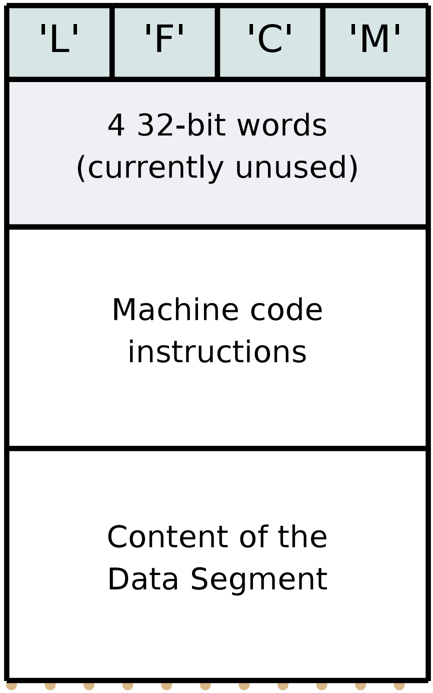


MACE Components

- 32 general-purpose 32-bit registers: `R0`–`R31`
  - Note that `R0` is wired to 0!
- One 32-bit Program Counter (`PC`)
- One 32-Bit Status Register (`PSW`)
- A 32-bit memory, word-addressed.

Object file format

- Header: “LFCM” plus 16 reserved bytes
- Machine code
- Whole data segment, already initialized



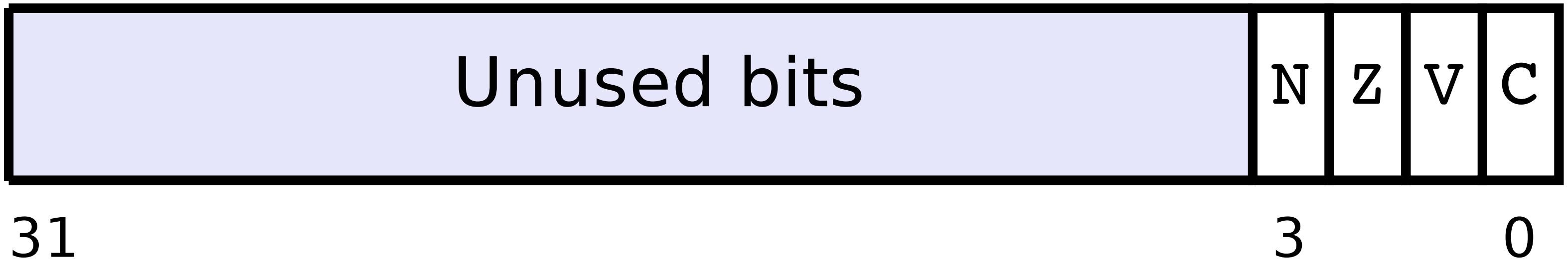
MACE Bootstrap

- Test if the object file exists and is readable
- Set all machine registers (including PC) to zero
- Reserve a 2 KB block of memory (for code and data segments)
- Load machine code from object file to code segment
- Load data from object file to data segment

MACE Execution Loop

- Repeat:
  - Fetch the next instruction according to `PC`
  - Decode the fetched instruction
  - Execute the instruction
  - Update the content of the register file
  - Update the value of `PC`
  - Update the value of `PSW`
- Until a `HALT` instruction is encountered.

MACE Processor Status Word



Unused bits

N Z V C

31 3 0

- N (Negative): set if the most significant bit of the result of an instruction is 1; cleared otherwise.
- Z (Zero): set if the result of an instruction is equal to zero; cleared otherwise.
- V (Overflow): set only if an arithmetic overflow occurs (the result cannot be represented in the destination size).
- C (Carry): set if a carry out occurs for an addition, or a borrow occurs in a subtraction.

MACE Addressing Modes

- Direct addressing: the content of register. E.g., `R1`.
- Indirect addressing: the content of the memory addressed by a register. E.g., `(R2)`.

MACE Addressing Modes

- Direct addressing: the content of register. E.g., `R1`.
- Indirect addressing: the content of the memory addressed by a register. E.g., `(R2)`.
- Example: `ADD R3 R1 (R2)`

Before ADD:

Registers		Memory	
R0:	0	0:	100
R1:	1	1:	110
R2:	2	2:	120
R3:	3	3:	130

After ADD:

Registers		Memory	
R0:	0	0:	100
R1:	1	1:	110
R2:	2	2:	120
R3:	121	3:	130

MACE Instruction Types

Ternary: 3 (2 sources + 1 destination) register operands

Binary: 2 (1 source + 1 destination) register operands
+ 1 16-bit immediate operand

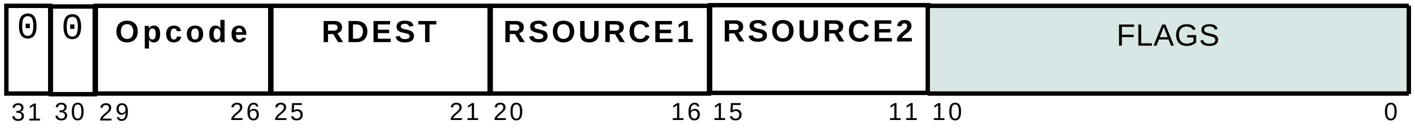
Unary: 1 (destination) register operand
+ 1 20-bit address operand

Jump: 1 20-bit address operand

- Direct or indirect addressing for second source and destination operands in ternary instructions
- Other register operands always directly addressed
- Some operands are not used

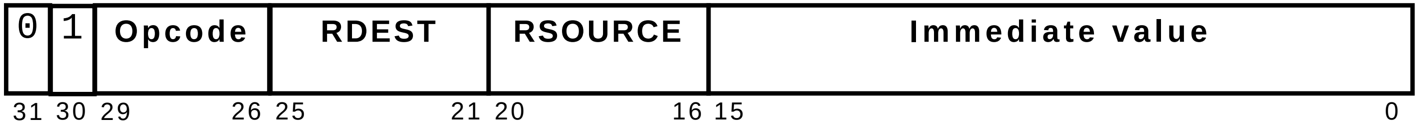
MACE Instruction Format

Ternary Instruction



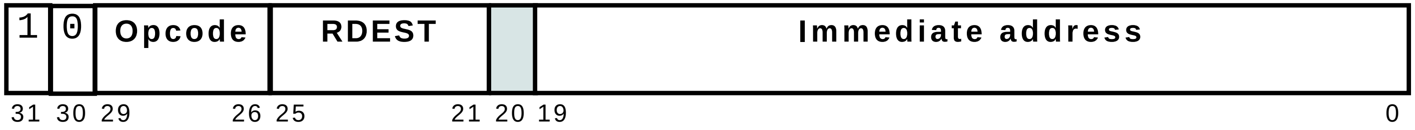
0	0	Opcode	RDEST	RSOURCE1	RSOURCE2	FLAGS
31	30	29 26	25 21	20 16	15 11	10 0

Binary Instruction



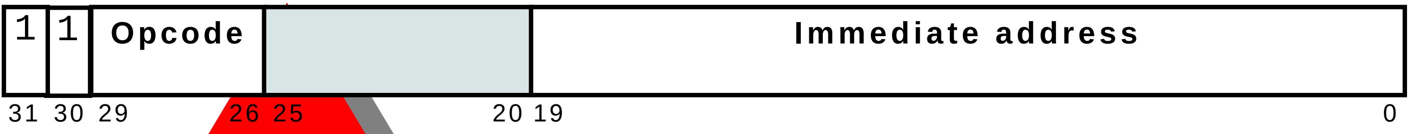
0	1	Opcode	RDEST	RSOURCE	Immediate value
31	30	29 26	25 21	20 16	15 0

Unary Instruction



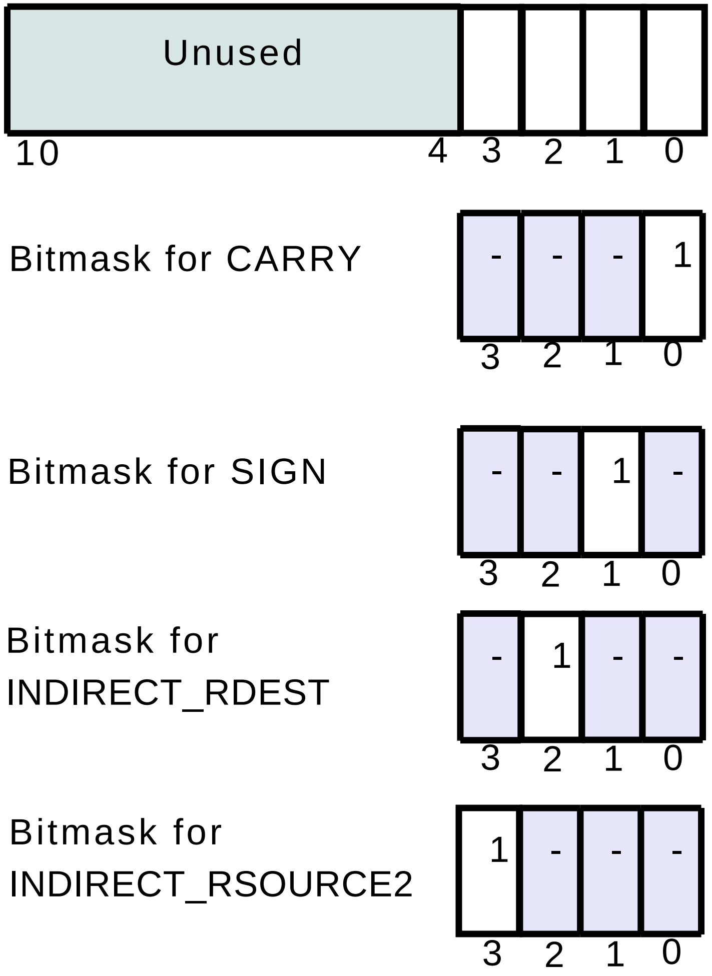
1	0	Opcode	RDEST		Immediate address
31	30	29 26	25 21	20	19 0

Jump Instruction



1	1	Opcode		Immediate address
31	30	29 26	25 20	19 0

MACE Instruction Format (2)



Unused				
10 … 4	3	2	1	0

Bitmask for CARRY

3	2	1	0
-	-	-	1

Bitmask for SIGN

3	2	1	0
-	-	1	-

Bitmask for INDIRECT_RDEST

3	2	1	0
-	1	-	-

Bitmask for INDIRECT_RSOURCE2

3	2	1	0
1	-	-	-

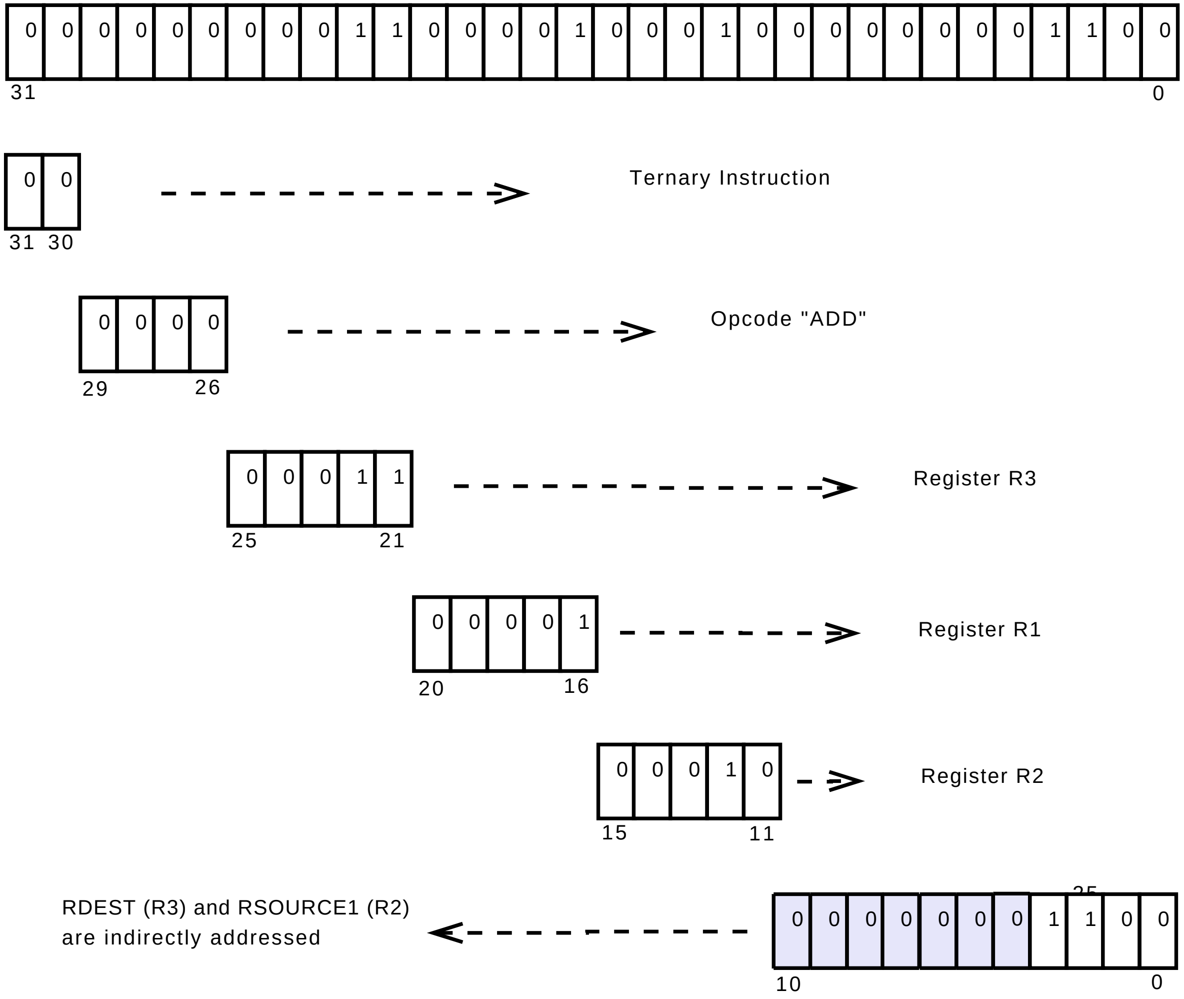
If CARRY is set, the result of the operation is incremented or decrement by the carry bit of the PSW

If SIGN is set, RSOURCE1 and RSOURCE2 are treated as signed integers

INDIRECT_RDEST is set if RDEST is indirectly addressed

INDIRECT_RSOURCE2 is set if ‘RSOURCE2’ is indirectly addressed

Example: ADD



0 0 0 0 0 0 0 0 0 1 1 0 0 0 0 1 0 0 0 1 0 0 0 0 0 0 0 0 1 1 0 0

31 0

0 0

31 30

Ternary Instruction

0 0 0 0

29 26

Opcode "ADD"

0 0 0 1 1

25 21

Register R3

0 0 0 0 1

20 16

Register R1

0 0 0 1 0

15 11

Register R2

0 0 0 0 0 0 0 1 1 0 0

10 0

RDEST (R3) and RSOURCE1 (R2)
are indirectly addressed

Decoded Instruction : ADD (R3) R1 (R2)

Instruction Set: Notation

Possible effects on `PSW` bits:

U The state of the bit is undefined

- The bit remains unchanged

***** The bit is set or cleared depending on the result

0 The bit is always cleared

Operands:

R*n* A register location, either directly or indirectly addressed.

imm An immediate value, a signed 16 bit integer.

Instruction Set: ADD

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
	* * * *

Instruction Set: ADDI

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 * * * *

Instruction Set: MOVA

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`
	`- - - -`

Instruction Set: LOAD

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`
	`- - - -`

Instruction Set: STORE

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`
	`- - - -`

Instruction Set: Scc (1)

Scc	*Set according to condition ‘cc’*
Syntax	Scc [Rdest] [Address]
Semantics	IF cc == 1 THEN [Rdest] ← 1; ELSE [Rdest] ← 0.
Note:	Address is unused.

’cc’ is checked against PSW register.

E.g.: ‘SEQ Rx’ stores 1 into Rx if bit ‘N’ is set in PSW; otherwise Rx is set to zero.

Instruction Set: Scc (2)

Binary Op-codes:

SEQ	‘0111’
SGE	‘1000’
SGT	‘1001’
SLE	‘1010’
SLT	‘1011’
SNE	‘1100’

Condition codes:

N	Z	V	C
0	*	0	0

Instruction Set: READ

READ	*Read from standard input an integer value*
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’

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

Instruction Set: Write

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

Instruction Set: Jump

Bcc	*Branch on condition cc*
Syntax	Bcc [Label]

Semantics	IF cc == 1 THEN
	PC ← PC + **Displacement**;

- *Displacement* is the distance between *PC* and the label address
- ‘cc’ is checked against the `PSW` register
- E.g.: ‘BEQ Label’ performs a branch to ‘label’ address if bit N is set in `PSW`. Otherwise Rx is set to zero.

Instruction Set: Branch Conditions

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