

Exercise session 1

LC-3 Architecture, and Lexical Analysis

1.1 The LC-3 architecture

We give you the “LC-3 cheat sheet”. The objective is to refresh memories about the LC-3 assembly you already saw in LIF6. Your teaching assistant will make some demo (of Pennsim simulator) during this session. Companion sheet: A.

EXERCISE #1 ► TD

On paper, write (in LC-3 assembly language) a program which initializes the $R0$ register to 1 and increments it until it becomes equal to 10.

EXERCISE #2 ► Hand disassembling

In Figure 1.1 we depicted a toy example with its corresponding assembly code. Disassemble the two first instructions in the table.

Fill the first two row of the table, read the rest of the solution, and answer the following questions:

- Which instruction is used to load data from memory?
- Could we do it another way?
- How is the pointer jumping done to create the loop?
- What happens to the labels in the disassemble program?

Address	Content	Binary	Instructions	pseudo-code
x3000	x5020			
x3001	x1221			
x3002	xE404	1110 010 0 0000 0100	LEA R2, Offset9=4	$R_2 \leftarrow x3007$ (label end)
x3003	x6681	010 011 010 00 0001	LDR R3, R2, 1	$R_3 \leftarrow mem[R_2 + 1]$ (label of data \rightarrow x3008)
loop:x3004	x1262	0001 001 001 1 00010	ADD R1, R1, 2	$R_1 \leftarrow R_1 + 2$
x3005	x16FF	0001 011 011 1 11111	ADD R3, R3, -1	$R_3 \leftarrow R_3 - 1$
x3006	x03FD	0000 001 1 1111 1101	BRp Offset9=-3	if $R_3 > 0$ goto loop
end:x3007	xF025	1111 0000 0010 0101	TRAP x25	HALT
data:x3008	x0006	data	-	

Figure 1.1: A binary/hexadecimal program (tp1-52.asm)

EXERCISE #3 ► C to LC-3- Skip if you are late

Translate into LC-3 code the following C-code:

```
x=5;
if (x>12) y=70; else y=x+12
```

1.2 Lexical Analysis

A bit of ANTLR4 syntax is given as companion material.



EXERCISE #4 ► Regular expressions for lexing

Use the ANTLR4 syntax to define ANTLR4 macros to define:

1. Identifiers : any sequence of letters, digits and `_` that do not begin by a digit nor `_`.
2. Floats like `-3.96` (the sign is optional, but the dot is not).
3. Scientific notation like `-1.6E - 12`.

EXERCISE #5 ► Romans numbers

Write an ANTLR4 lexical file that reads and interprets Roman numerals : `IV` → 4 ... You can use the fact that the lexical analysis always takes the rule to match the longest subchain.

ba64a1e on 28 Mar
 beardlybread documentation typos
 2 contributors 

284 lines (215 sLoc) 10.5 KB Raw Blame History

Lexer Rules

A lexer grammar is composed of lexer rules, optionally broken into multiple modes. Lexical modes allow us to split a single lexer grammar into multiple sublexers. The lexer can only return tokens matched by rules from the current mode.

Lexer rules specify token definitions and more or less follow the syntax of parser rules except that lexer rules cannot have arguments, return values, or local variables. Lexer rule names must begin with an uppercase letter, which distinguishes them from parser rule names:

```
/** Optional document comment */
TokenName : alternative1 | ... | alternativeN ;
```

You can also define rules that are not tokens but rather aid in the recognition of tokens. These fragment rules do not result in tokens visible to the parser:

```
fragment
HelperTokenRule : alternative1 | ... | alternativeN ;
```

For example, DIGIT is a pretty common fragment rule:

```
INT : DIGIT+ ; // references the DIGIT helper rule
fragment DIGIT : [0-9] ; // not a token by itself
```

Lexical Modes

Modes allow you to group lexical rules by context, such as inside and outside of XML tags. It's like having multiple sublexers, one for context. The lexer can only return tokens matched by entering a rule in the current mode. Lexers start out in the so-called default mode. All rules are considered to be within the default mode unless you specify a mode command. Modes are not allowed within combined grammars, just lexer grammars. (See grammar XMLLexer from Tokenizing XML.)

```
rules in default mode
...
mode MODE1;
rules in MODE1
...
mode MODEN;
rules in MODEN
...
```

Lexer Rule Elements

Lexer rules allow two constructs that are unavailable to parser rules: the `..` range operator and the character set notation enclosed in square brackets, `[characters]`. Don't confuse character sets with arguments to parser rules. `[characters]` only means character set in a lexer. Here's a summary of all lexer rule elements:

Syntax	Description
T	Match token T at the current input position. Tokens always begin with a capital letter.
'literal'	Match that character or sequence of characters. E.g., 'while' or '='.
[char set]	Match one of the characters specified in the character set. Interpret x-y as set of characters between range x and y, inclusively. The following escaped characters are interpreted as single special characters: \n, \r, \b, \t, and \f. To get], \, or - you must escape them with \. You can also use Unicode character specifications: \uXXXX. Here are a few examples: <pre>WS : [\n\u000D] -> skip ; // same as [\n\r] ID : [a-zA-Z] [a-zA-Z0-9]* ; // match usual identifier spec DASHBRACK : [\-\\]+ ; // match - or] one or more times</pre>
'x'..'y'	Match any single character between range x and y, inclusively. E.g., 'a'..'z'. 'a'..'z' is identical to [a-z].
T	Invoke lexer rule T; recursion is allowed in general, but not left recursion. T can be a regular token or fragment rule. <pre>ID : LETTER (LETTER '0'..'9')* ; fragment LETTER : [a-zA-Z\u0000-\u00FF_] ;</pre>
.	The dot is a single-character wildcard that matches any single character. Example: <pre>ESC : '\\\ ' . ; // match any escaped \x character</pre>
{«action»}	Lexer actions can appear anywhere as of 4.2, not just at the end of the outermost alternative. The lexer executes the actions at the appropriate input position, according to the placement of the action within the rule. To execute a single action for a role that has multiple alternatives, you can enclose the alts in parentheses and put the action afterwards: <pre>END : ('endif' 'end') {System.out.println("found an end");} ;</pre> The action conforms to the syntax of the target language. ANTLR copies the action's contents into the generated code verbatim; there is no translation of expressions like \$x.y as there is in parser actions. Only actions within the outermost token rule are executed. In other words, if STRING calls ESC_CHAR and ESC_CHAR has an action, that action is not executed when the lexer starts matching in STRING.
{«p»}?	Evaluate semantic predicate «p». If «p» evaluates to false at runtime, the surrounding rule becomes "invisible" (nonviable). Expression «p» conforms to the target language syntax. While semantic predicates can appear anywhere within a lexer rule, it is most efficient to have them at the end of the rule. The one caveat is that semantic predicates must precede lexer actions. See Predicates in Lexer Rules.
~X	Match any single character not in the set described by x. Set x can be a single character literal, a range, or a subrule set like ~(x y 'z') or ~[xyz]. Here is a rule that uses ~ to match any character other than characters using ~(\r\n)*: <pre>COMMENT : '#' ~[\r\n]* '\r'? '\n' -> skip ;</pre>

Just as with parser rules, lexer rules allow subrules in parentheses and EBNF operators: `?`, `*`, `+`. The COMMENT rule illustrates the `*` and `?` operators. A common use of `+` is `[0-9]+` to match integers. Lexer subrules can also use the nongreedy `?` suffix on those EBNF operators.

Appendix A

LC3

A.1 Installing Pennsim and getting started

To install and use PennSim, read the following documentation :

<http://castle.eiu.edu/~mathcs/mat3670/index/Webview/pennsim-guide.html>

A.2 The LC3 architecture

Memory, Registers The LC-3 memory is shared into words of 16 bits, with address of size 16 bits (from $(0000)_H$ to $(FFFF)_H$).

The LC-3 has 8 main registers : R0, ..., R7. R6 is reserved for the execution stack handling, R7 for the routine return address. They are also specific 16 bits registers: PC (*Program Counter*), IR (*Instruction Register*), PSR (*Program Status Register*).

The PSR has 3 bits N,Z and P that indicate if the last value written in one of the R0 to R7 registers (viewed as a 16bits 2-complement integer) is strictly negative (N), nul (Z) or strictly positive(P).

Instructions :

Syntax	Action	NZP
NOT DR,SR	DR <- not SR	*
ADD DR,SR1,SR2	DR <- SR1 + SR2	*
ADD DR,SR1,Imm5	DR <- SR1 + SEXT(Imm5)	*
AND DR,SR1,SR2	DR <- SR1 and SR2	*
AND DR,SR1,Imm5	DR <- SR1 and SEXT(Imm5)	*
LEA DR,label	DR <- PC + SEXT(PCOffset9)	*
LD DR,label	DR <- mem[PC + SEXT(PCOffset9)]	*
ST SR,label	mem[PC + SEXT(PCOffset9)] <- SR	
LDR DR,BaseR,Offset6	DR <- mem[BaseR + SEXT(Offset6)]	*
STR SR,BaseR,Offset6	mem[BaseR + SEXT(Offset6)] <- SR	
BR[n][z][p] label	Si (cond) PC <- PC + SEXT(PCOffset9)	
NOP	No Operation	
RET	PC <- R7	
JSR label	R7 <- PC; PC <- PC + SEXT(PCOffset11)	

Assembly directives

.ORIG add	Specifies the address where to put the instruction that follows
.END	Terminates a block of instructions
.FILL val	Reserves a 16-bits word and store the given value at this address
.BLKW nb	Reserves nb (consecutive) blocks of 16 bits at this address
;	Comments

Predefined interruptions TRAP gives a way to implement system calls, each of them is identified by a 8-bit constant. This is handled by the OS of the LC-3. The following macros indicate how to call them:

instruction	macro	description
TRAP x00	HALT	ends a program (give back decisions to OS)
TRAP x20	GETC	reads from the keyboard an ASCII char, and puts its value into R0
TRAP x21	OUT	writes on the screen the ASCII char of R0
TRAP x22	PUTS	writes on screen the string whose address of first character is stored in R0
TRAP x23	IN	reads from keyboard an ASCII char, outputs on screen and stores its value in R0

Constants : The integer constants encoded in hexadecimal are prefixed by **x**, in decimal by an optional **#** ; they can appear as parameters of the LC-3 instructions (immediate operands, be careful with the sizes) and directives like **.ORIG**, **.FILL** et **.BLKW**.

Coding tricks

- Initialisation to zero of a given register: **AND Ri ,Ri ,#0**
- Initialisation to a constant n (representable on 5 bits in complement to 2):
AND Ri ,Ri ,#0
ADD Ri ,Ri ,n
- Computation of the (integer) opposite $R_i \leftarrow (-R_j)$ (1+ complement to 2):
NOT Ri ,Rj
ADD Ri ,Ri ,#1
- Multiplication $R_i \leftarrow 2R_j$: **ADD Ri ,Rj ,Rj**
- Copy $R_i \leftarrow R_j$: **ADD Ri ,Rj ,#0**

A.3 LC3 simplified instruction set

Here is a recap of instructions and their encoding:

syntaxe	action	NZP	codage																
			opcode				arguments												
			F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0	
NOT DR,SR	DR ← not SR	*	1	0	0	1	DR		SR				1 1 1 1 1 1						
ADD DR,SR1,SR2	DR ← SR1 + SR2	*	0	0	0	1	DR		SR1				0	0 0		SR2			
ADD DR,SR1,Imm5	DR ← SR1 + SEXT(Imm5)	*	0	0	0	1	DR		SR1				1	Imm5					
AND DR,SR1,SR2	DR ← SR1 and SR2	*	0	1	0	1	DR		SR1				0	0 0		SR2			
AND DR,SR1,Imm5	DR ← SR1 and SEXT(Imm5)	*	0	1	0	1	DR		SR1				1	Imm5					
LEA DR,label	DR ← PC + SEXT(PCOffset9)	*	1	1	1	0	DR		PCOffset9										
LD DR,label	DR ← mem[PC + SEXT(PCOffset9)]	*	0	0	1	0	DR		PCOffset9										
ST SR,label	mem[PC + SEXT(PCOffset9)] ← SR		0	0	1	1	SR		PCOffset9										
LDR DR,BaseR,Offset6	DR ← mem[BaseR + SEXT(Offset6)]	*	0	1	1	0	DR		BaseR				Offset6						
STR SR,BaseR,Offset6	mem[BaseR + SEXT(Offset6)] ← SR		0	1	1	1	SR		BaseR				Offset6						
BR[n][z][p] label	Si (cond) PC ← PC + SEXT(PCOffset9)		0 0 0 0				n	z	p	PCOffset9									
NOP	No Operation		0 0 0 0				0	0	0	0 0 0 0 0 0 0 0									
RET (JMP R7)	PC ← R7		1 1 0 0				0 0 0			1 1 1				0 0 0 0 0 0					
JSR label	R7 ← PC; PC ← PC + SEXT(PCOffset11)		0 1 0 0				1	PCOffset11											