Preliminary Niki V2 Instruction Set

A project of Stefan Nikolaj - 04.11.2023

To abstract Niki from a specific programming language, there is a need for a platform-agnostic instruction set which can execute all of Niki's commands. The commands implement a limited instruction set as detailed by Prof. Werner. The first part of this document will list the instructions and the second part will explain them further, as well as their rationale.

The Niki processor shall have the following registers which the instructions will read and write:

- SP - stack pointer
- PC - program counter
- BC - ball counter
- A - primary binary expression register, also used for conditional jumps
- B - secondary binary expression register
- ER - error register, contains error codes if the robot is incapable of executing certain movement commands, set to '0' if there are no errors
- DIR - direction which Niki is currently facing
- COL - column index in the grid in which Niki is located
- ROW - row index in the grid in which Niki is located

The assembly code mnemonics are only meant to be produced by an interpreter/compiler or human who knows what they're doing. This document serves as a reference for implementers.

Table of contents:

Table 1: All instructions and their binary encodings

Instruction type and name(s)	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Instruction cycle length	Can affect ER
Binary expression (<AND/OR/NOTA/NOTB>)	0	0	0	0	0	x	a	a	1	No
Set Register (SR <A/B> <RESET/SET> (or <0/1>))	0	0	0	0	1	x	!A / B	!RESET / SET	1	No
Movement (MOV/LEFT/TAKE/DROP)	0	0	0	1	x	x	d	d	1	Yes
Stack (PUSH/RET)	0	0	1	x	x	x	x	!PUSH / RET	1	No
Load Sensor (LD <A/B>, <sensor>)	0	1	x	!A / B	b	b	b	b	1	No
Jump (JMP/CJMP)	1	c	c	c	c	c	c	c	2 (+1)	No

Legend:
‘0’ - Bit is cleared
‘1’ - Bit is set
‘x’ - Don’t care
‘!’ - Active low register/instruction, condition only applies when the bit is ‘0’
“<>” - Denote instruction operand names
‘a’ - Series of bits corresponding to which binary expression to use (see Table 2)
“!A / B” - The instruction will load the result in register A if the respective bit is ‘0’, and register B if the respective bit is ‘1’
“!RESET / SET” - The instruction will set the selected register to ‘0’ if the respective bit is ‘0’, and set the selected register to ‘0’ if the

respective bit is ‘1’
‘d’ - Series of bits corresponding to which movement-related command will be executed (see Table 3)
“!PUSH / RET” - The instruction will be executed as PUSH if the respective bit is ‘0’, and RET if the respective bit is ‘1’
‘b’ - Series of bits corresponding to which sensor to load (see Table 4)
‘c’ - Series of bits corresponding to the address of the jump

Table 2: Binary expressions

Binary Expression	Bit 1	Bit 0
A := A AND B	0	0
A := A OR B	0	1
A := NOT A	1	0
B := NOT B	1	1

Table 3: Movement commands

Name	German Version	Bit 1	Bit 0	Value in ER if error condition met
Move (MOV)	vor	0	0	0b01
Turn left (LEFT)	drehe_links	0	1	N/A
Take ball (TAKE)	nimm_auf	0	0	0b10
Drop ball (DROP)	gib_ab	1	1	0b11

Table 4: Sensors

Sensor	German Version	Bit 3	Bit 2	Bit 1	Bit 0
North	nordwaerts	0	0	0	0
West	westwaerts	0	0	0	1
South	suedwaerts	0	0	1	0
East	ostwaerts	0	0	1	1
Front free	vorne_frei	0	1	0	0
Left free	links_frei	0	1	0	1
Right free	rechts_frei	0	1	1	0
Ball on place	platz_belegt	0	1	1	1
Has balls	hat_Vorrat	1	0	0	0

Instruction descriptions

Binary Expression - BINE <exp>

Instruction name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Binary expression	0	0	0	0	0	x	a	a

Where the bits denoted by ‘a’ represent the following expressions:

Binary Expression	Expression Name	Bit 1	Bit 0
A := A AND B	AND	0	0
A := A OR B	OR	0	1
A := NOT A	NOTA	1	0
B := NOT B	NOTB	1	1

Most expressions pertain to register A because it is the register used in conditional jumps. Given that both register A and register B can be negated, combined with the fact that they can be ANDed and ORed together allows for the binary expressions NAND and NOR to be computed as well.

This instruction is meant to be used before if-statements, while-statements and conditional jumps.

Set Register - SR <A/B> <RESET/SET> (or <0/1>)

Instruction name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Set Register (SR <A/B> <RESET/SET> (or <0/1>))	0	0	0	0	1	x	!A / B	!RESET / SET

This instruction sets or resets registers A and B according to the following table:

Result	Bit 1	Bit 0
Register A is set to 0	0	0
Register A is set to 1	0	1
Register B is set to 0	1	0
Register B is set to 1	1	1

This instruction is meant to be used when the system is reset as well as before jumps. Since one jump instruction exists for both conditional and unconditional jumps, the compiler/interpreter shall automatically set register A to ‘1’ so as to fulfil the jump criterion (is A equal to ‘1’).

Movement - MOV / LEFT / TAKE / DROP

Instruction name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Movement (MOV/LEFT/TAKE/ DROP)	0	0	0	1	x	x	d	d

These instructions can affect Niki in the physical world and include the four basic motion commands that Niki can execute. The instruction executed is detailed in the following table:

Name	German Version	Bit 1	Bit 0	Value in ER if error condition met
Move (MOV)	vor	0	0	0b01
Turn left (LEFT)	drehe_links	0	1	N/A
Take ball (TAKE)	nimm_auf	0	0	0b10
Drop ball (DROP)	gib_ab	1	1	0b11

These instructions can affect the error register ER. They are meant to work both on simulated versions of Niki, as well as ones in the real world. Their implementation can vary. In simulated versions of Niki, the interpreter shall change the DIR, COL and ROW registers based on the executed instruction. In hardware versions, the DIR, COL and ROW registers may be ignored or omitted in favor of real sensors.

The instruction Take ball increases the ball counter BC by 1 if a ball exists in that position. If there is no ball in that position, the respective value from the table above will be loaded into ER.

The instruction Drop ball decreases the ball counter BC by 1 if BC is greater than 0. If BC is 0, the respective value from the table above will be loaded into ER.

Stack - PUSH / RET

Instruction name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Stack (PUSH/RET)	0	0	1	x	x	x	x	!PUSH / RET

If Bit 0 is set to ‘0’, the CPU will execute a PUSH instruction, which will push the current value of the program counter PC on the stack and increase the stack pointer SP by 1. This instruction is meant to be used before jumps to functions.

If Bit 0 is set to ‘1’, the CPU will execute a RET (return) instruction, which will load the value on the stack indicated by the stack pointer SP in the program counter PC and decrease the stack pointer by 1. This instruction is meant to be used before returning from functions.

Load Sensor - LD <A/B>, <sensor>

Instruction name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Load Sensor (LD <A/B>, <sensor>)	0	1	x	!A / B	b	b	b	b

Loads the value of the sensor indicated in the lowermost 4 bits (Bit 3 - Bit 0) into either A (if Bit 4 is ‘0’) or B (if Bit 4 is ‘1’). The sensor is chosen according to the following table:

Sensor	German Version	Bit 3	Bit 2	Bit 1	Bit 0
North	nordwaerts	0	0	0	0
West	westwaerts	0	0	0	1
South	suedwaerts	0	0	1	0
East	ostwaerts	0	0	1	1
Front free	vorne_frei	0	1	0	0
Left free	links_frei	0	1	0	1
Right free	rechts_frei	0	1	1	0
Ball on place	platz_belegt	0	1	1	1
Has balls	hat_Vorrat	1	0	0	0

This instruction is meant to be used before conditional statements.

The implementation of the sensors can vary. The North, West, South and East sensors should copy the value of the DIR register.

Jump - JMP / CJMP

Instruction name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Jump (JMP/CJMP)	1	c	c	c	c	c	c	c

Jumps to a location in memory by loading the value of the instruction in the program counter PC. This is the only 2-byte instruction, with the initial 7 bits Bit 6 - Bit 0 form the top 7 bits of the program counter PC, while during the next load cycle another 8 bits are loaded which form the bottom 8 bits of the program counter PC. Thus only a maximum of $2^{15} = 32768$ memory locations can exist. If the instruction was only 1-byte, only $2^{7} = 128$ memory locations could exist, which would make complex programs impossible.

The CPU shall execute the same code for both JMP and CJMP, where it will check whether the condition A = 1 has been met in order to execute the jump, and it shall be the responsibility of the compiler to include a SET A instruction before each JMP. The result of this is that each JMP instruction shall take 3 cycles to execute due to the necessity of the SET A instruction, and it shall be 3 bytes long. The CJMP instruction shall take 2 cycles to execute and it shall be 2 bytes long.