



# N1 Manual

Dirk Heisswolf

January 23, 2019

---

## Revision History

Date	Change
January 23, 2019	Pre-release

## Contents

<b>1</b>	<b>Glossary</b>	<b>5</b>
<b>2</b>	<b>Overview</b>	<b>7</b>
<b>3</b>	<b>Instruction Set</b>	<b>8</b>
3.1	Return from a Call (;) . . . . .	9
3.2	Jump Instructions . . . . .	9
3.3	Call Instructions . . . . .	9
3.4	Conditional Branches . . . . .	9
3.5	Literals . . . . .	9
3.6	ALU Instructions . . . . .	9
3.7	Stack Instructions . . . . .	10
3.8	Memory Access Instructions . . . . .	14
3.9	Control Instructions . . . . .	14
<b>4</b>	<b>ANS Forth Words</b>	<b>15</b>
<b>5</b>	<b>Stacks</b>	<b>16</b>
5.1	Parameter Stack . . . . .	16
5.2	Return Stack Stack . . . . .	17

**List of Figures**

3-1	Instruction encoding . . . . .	8
3-2	Transition encoding of stack instructions . . . . .	11
5-1	Stack Architecture . . . . .	16

**List of Tables**

3-1	ALU operations . . . . .	10
3-2	Common stack operations . . . . .	11
3-2	Common stack operations . . . . .	12
3-2	Common stack operations . . . . .	13
3-2	Common stack operations . . . . .	14
3-3	Control instructions . . . . .	14
4-1	ALU operations . . . . .	15

# 1 Glossary

**;**  
End of a [word](#) definition in [Forth](#).

**ALU**  
Arithmetic Logic Unit.

**byte**  
An 8-bit data entity.

**call**  
A change of the program flow, where a return address is kept on the [return stack](#).

**cell**  
A data entity within a [stack](#).

**conditional branch**  
A change of the program flow without return option, only if a certain (non-zero) argument value is given.

**Forth**  
Forth is a extensible stack-based programming language.

**intermediate stack**  
The section of the stack, which serves as a buffer between the [lower stack](#) and the [upper stack](#). See [Section 5 “Stacks”](#).

**IST**  
A bit field in the stack instruction which controls data movement on the intermediate [parameter stack](#) or [return stack](#). The mnemonic stands for “**I**ntermediate **S**tack **T**ransition”.

**jump**  
A change of the program flow without return option.

**literal**  
A fixed numerical value within the program code.

**lower stack**  
The section of the stack which stored in RAM. See [Section 5 “Stacks”](#).

**opcode**  
Encoding of a machine instruction. Short for “operation code”.

**parameter stack**  
A [LIFO](#) storage mainly for keeping call parameters and return values.

**RAM**

Random access memory.

**return stack**

A [LIFO](#) storage mainly for maintaining return addresses of [calls](#).

**stack**

A [LIFO](#) storage.

**upper stack**

The section of the stack, which contains the [TOS](#). It supports reordering of its storage [cell](#). See [Section 5](#) “[Stacks](#)”.

**UST**

A bit field in the stack instruction which controls data movement between two neighboring [cells](#) in the upper [parameter stack](#) or [return stack](#). The mnemonic stands for “**U**pper **S**tack **T**ransition”.

**word**

The term word is used in two different contexts throughout this document. It refers to either a 16-bit data entity or a callable code sequence in [Forth](#) terminology.

## 2 Overview

The N1 is a small stack machine, inspired by the J1 Forth CPU[1]. Just like its paragon, the N1 is a 16-bit processor which implements basic [Forth words](#) directly in hardware. However the N1 parts from the J1's simplistic design approach in two ways:

- The N1 supports a larger code space of up to 32KB. Therefore it has its own instruction set (see [Section 3 “Instruction Set”](#)).
- The N1 implements its parameter and return stacks as shallow register stacks, which overflow into RAM. The overall depth of each stack is determined by the available RAM. (see [Section 5 “Stacks”](#)).



### 3 Instruction Set

The intent of the N1's instruction set is to map most of the essential Forth words to single cycle instructions. [Figure 3-1](#) illustrates the basic structure of the instruction encoding.

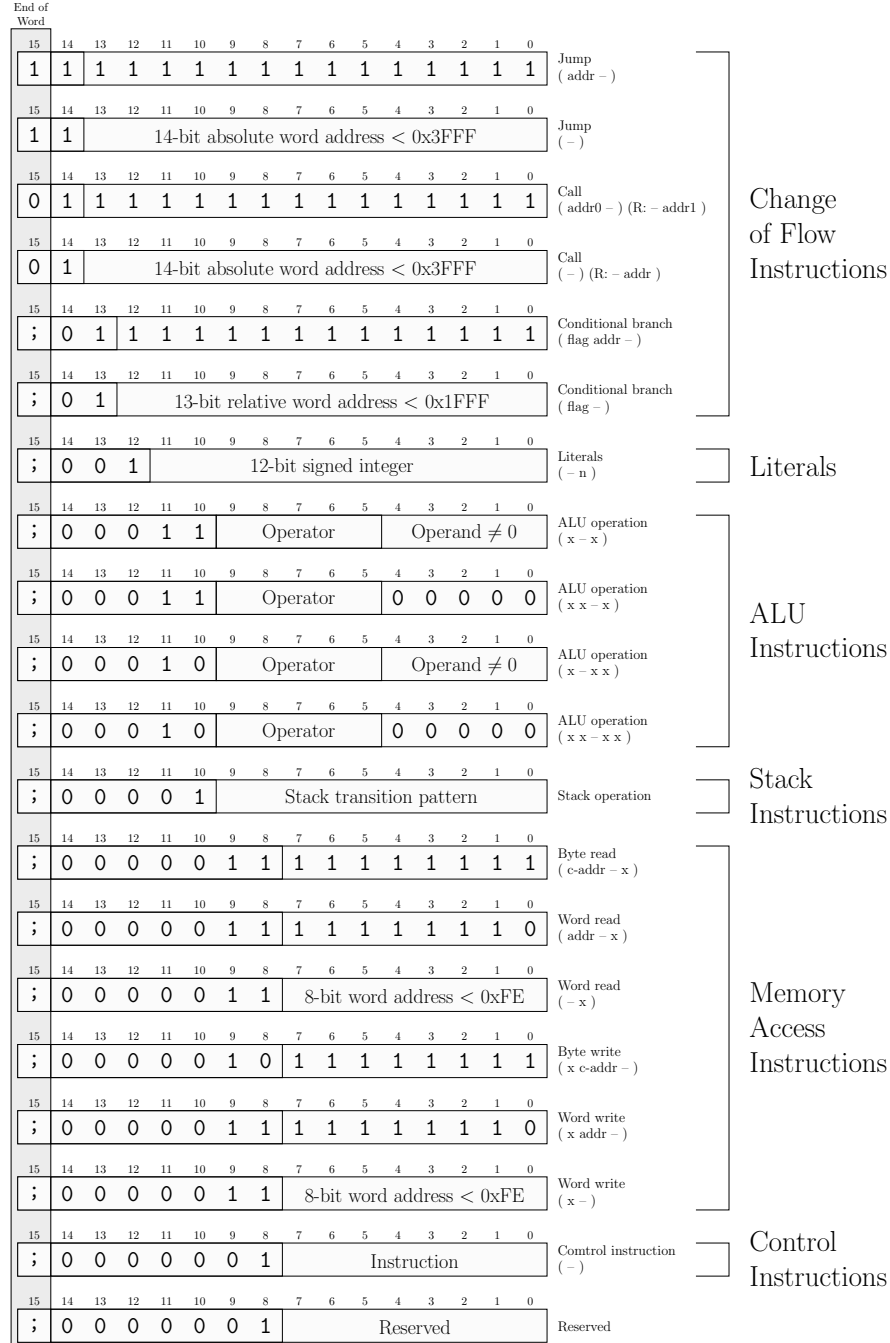


Figure 3-1: Instruction encoding

### 3.1 Return from a Call (;)

Rather than providing a dedicated instruction to end the execution of word in Forth and to return the program flow to its caller, the N1 allows to perform this operation in parallel to the execution of any of its instructions. Each [opcode](#) contains a bit (bit 15) to indicate, that the current instruction in the last operation in the current word. If this bit is set, the program flow will resume at the calling word as soon as the operation is performed.

As shown in [Figure 3-1](#), bit 15 is also used to distinguish jump and call. Considering that the last call in a word definition can be optimized to a jump to the first instruction of the called word, bit 15 can be regarded as termination bit for these instructions as well.

For a Forth compiler, this means that the semi-colon (;) always translates to setting bit 15 of the last instruction.

### 3.2 Jump Instructions

[Jump](#) instructions transfer the program flow to any [word](#) location within the supported 64KB program space. [Jump](#) instructions consume an absolute destination address, which can be either placed on the top of the [Parameter stack](#) or encoded into the opcode of the instruction (only for destination addresses < 0x3FFF).

### 3.3 Call Instructions

[Call](#) instructions temporarily transfer the program flow to any [word](#) location within the supported 64KB program space, while pushing a return address onto the return stack. [Call](#) instructions consume an absolute destination address, which can be either placed on the top of the [Parameter stack](#) or encoded into the opcode of the instruction (only for destination addresses < 0x3FFF).

### 3.4 Conditional Branches

[Conditional branches](#) invoke a change of program flow depending on an argument on the [parameter stack](#). The branch destination can be either an absolute address placed on the top of the [Parameter stack](#) or relative address, encoded into the opcode of the instruction (only for destination addresses < 0x1FFF).

### 3.5 Literals

Signed integer [literals](#) of 12-bit length can be pushed onto the [parameter stack](#) within a single instruction. For larger integers a supplemental TBD [call](#) is required.

### 3.6 ALU Instructions

[ALU](#) instructions perform an operation on two [cell](#) values, resulting in a new double [cell](#) value. The result can be either placed entirely onto the [parameter stack](#), or truncated, discarding the most significant [cell](#). The first operand is always taken from the [Parameter stack](#). The second operand can be either taken from the [Parameter stack](#) or encoded into the opcode of the instruction. In the latter case, the interpretation of the embedded 5-bit value depends on the operation. It is either regarded as an unsigned (*uimm*) or a sign extended value (*simm*). [Table 3-1](#) lists the supported [ALU](#) operations.

Table 3-1: ALU operations

Encoding	Operation	( x1 - d )	( x1 x2 - d )
00000	Sum	$x1 + uimm$	$x1 + x2$
00001	Sum	$oimm + x1$	$x2 + x1$
00010	Difference	$x1 - uimm$	$x1 - x2$
00011	Difference	$oimm - x1$	$x2 - x1$
00100	Unsigned lower-than comparison	$x1 < uimm?$	$x1 < x2?$
00101	Signed greater-than comparison	$oimm < x1?$	$x2 < x1?$
00110	Unsigned greater-than comparison	$x1 > uimm?$	$x1 > x2?$
00111	Signed lower-than comparison	$oimm > x1?$	$x2 > x1?$
01000	Equals comparison	$x1 = uimm?$	$x1 = x2?$
01001	Equals comparison	$oimm = x1?$	$x2 = x1?$
01010	Not-equals comparison	$x1 \neq uimm?$	$x1 \neq x2?$
01011	Not-equals comparison	$oimm \neq x1?$	$x2 \neq x1?$
01100	Unsigned product	$x1 * uimm$	$x1 * x2$
01101	Unsigned product	$x1 * simm$	$x1 * x2$
01110	Signed product	$x1 * uimm$	$x1 * x2$
01111	Signed product	$x1 * simm$	$x1 * x2$
10000	Logic AND	$x1 \wedge simm$	$x1 \wedge x2$
10001	Logic OR	$x1 \vee uimm$	$x1 \vee x2$
10010	Logic XOR	$x1 \oplus simm$	$x1 \oplus x2$
10011	Reserved		
10100	Logic right shift	$x1 \gg uimm$	$x2 \gg x1$
10101	Logic left shift	$x1 \ll uimm$	$x2 \ll x1$
10110	Arithmetic right shift	$x1 \gg uimm$	$x2 \gg x1$
10111	Reserved		
11000	Set upper bits of an immediate value	$simm, x1[11:0]$	$simm, x2[11:0]$
11001	Reserved		
11010	Reserved		
11011	Reserved		
11100	Current interrupt vector	vector address	
11100	Current error code	throw code	
11100	Parameter stack status	IPS:UPS	
11100	Return stack status	IPS:UPS	

### 3.7 Stack Instructions

The N1's stack instruction aims at efficiently implementing the essential stack operations in [Forth](#) only using the data pathes which needed for the stack's push and pull operations.

The opcode of the stack instruction contains a 10-bit wide field to specify a transition pattern of the upper [cells](#) of the [parameter stack](#) and the [return stack](#). The structure transition patten is shown in [Figure 3-2](#).

The stack instruction contains four [UST](#) fields which control the data transfer within the upper four [cells](#) of the [parameter stack](#) and the top of the [return stack](#). Each [UST](#) field determines the direction of data transfer between two neighboring stack [cells](#). Four options are selectable:

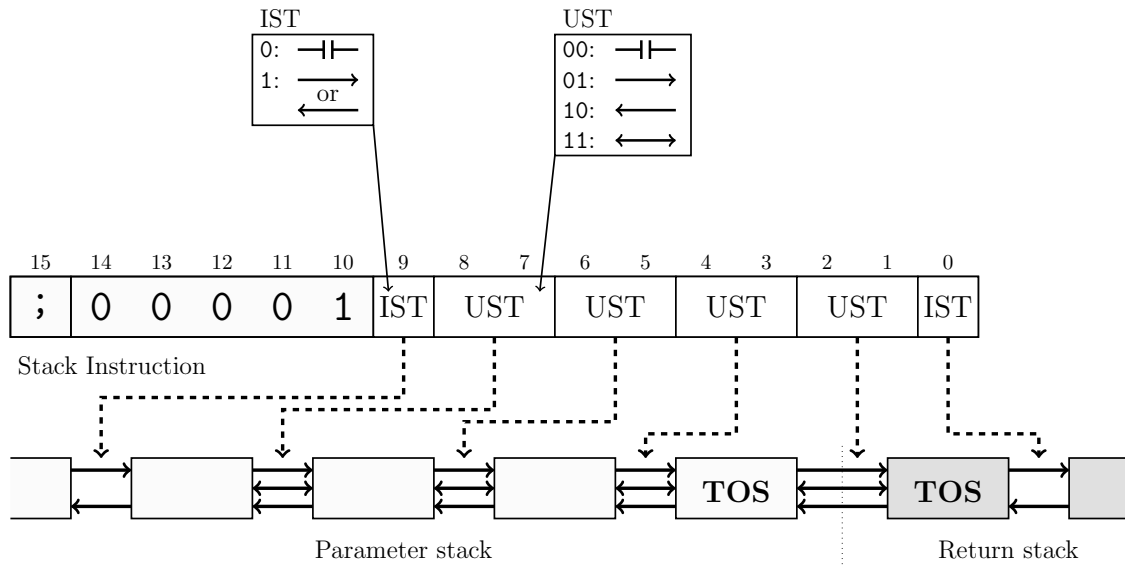


Figure 3-2: Transition encoding of stack instructions

- No data transfer
- Data transfer upwards (or towards the [return stack](#))
- Data transfer downwards (or towards the [parameter stack](#))
- Data exchange between two stack [cells](#)

It is possible to put the [UST](#) fields into a combination which would trigger a data transfer of two source [cells](#) to a single destination [cell](#). In these cases, the resulting data in the destination [cell](#) is undefined.

The two remaining [IST](#) fields in the stack instruction control the data movement of the [lower stacks](#). Two options are selectable:

- No data transfer
- Data shift throughout the entire [intermediate stack](#). The direction is determined by the data movement of the lowest cell of the [upper stack](#).











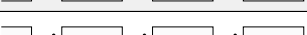
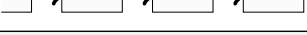

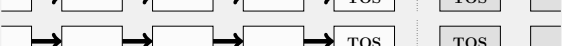






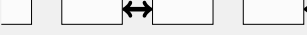

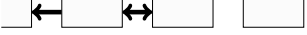

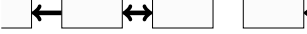

[Table 3-2](#) shows how [stack](#) operations in [Forth](#) are mapped N1 instructions.

Table 3-2: Common stack operations

Word	Description	Transitions	Opcode
DROP	( x - )		0x06A8
DUP	( x - x x )		0x0750
SWAP	( x1 x2 - x2 x1 )		0x0418
OVER	( x1 x2 - x1 x2 x1 )		0x0758

...continued

Table 3-2: Common stack operations

Word	Description	Transitions	Opcode
NIP	( x1 x2 - x2 )		0x06A0
TUCK	( x1 x2 - x2 x1 x2 )		0x0750
			0x0460
ROT	( x1 x2 x3 - x2 x3 x1 )		0x0460
			0x0418
-ROT	( x1 x2 x3 - x3 x1 x2 )		0x0418
			0x0460
RDROP	( R: x - )		0x0001
RDUP	( R: x - x x )		0x0007
			0x0006
>R	( x - ) ( R: - x )		0x06AB
R@	( - x ) ( R: x - x )		0x0754
R>	( - x ) ( R: x - )		0x0755
2DROP	( x1 x2 - )		0x06A8
			0x06A8
2DUP	( x1 x2 - x1 x2 x1 x2 )		0x0758
			0x0758
2SWAP	( x1 x2 x3 x4 - x4 x3 x1 x2 )		0x0460
			0x0598
			0x0460
2OVER	( x1 x2 x3 x4 - x1 x2 x3 x4 x1 x2 )		0x0780
			0x0460
			0x0798
			0x0460
2NIP	( x1 x2 x3 x4 - x3 x4 )		0x06A0
			0x06A0

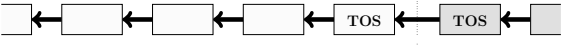



...continued

Table 3-2: Common stack operations

Word	Description	Transitions	Opcode
2TUCK	( x1 x2 x3 x4 – x3 x4 x1 x2 x3 x4 )		0x046B 0x0487 0x0418 0x0460 0x0755 0x0755
2ROT	( x1 x2 x3 x4 x5 x6 – x3 x4 x5 x6 x1 x2 )		0x06AB 0x0580 0x06AB 0x0598 0x0755 0x0598 0x0755 0x0598 0x0460
-2ROT	( x1 x2 x3 x4 x5 x6 – x5 x6 x1 x2 x3 x4 )		0x0460 0x0598 0x06AB 0x0598 0x06AB 0x0598 0x0755 0x0018 0x0755
2RDROP	( R: x1 x2 – )		0x0001 0x0001
2RDUP	( R: x1 x2 – x1 x1 x1 x2 )		0x0755 0x0757 0x06AB 0x06AB
2>R	( x1 x2 – ) ( R: – x1 x2 )		0x0000 0x0000

...continued

Table 3-2: Common stack operations

Word	Description	Transitions	Opcode
2R@	$(-x1\ x2)$ $(R: x1\ x2 - x1\ x2)$		0x0000
			0x0000
2R>	$(-x1\ x2)$ $(R: x1\ x2 -)$		0x0000
			0x0000

### 3.8 Memory Access Instructions

Memory access instructions perform read or write accesses to the system's 64-Kbyte address space. Data can be accessed in [word](#) or [byte](#) entities. Misaligned [word](#) accesses are not supported. [Word](#) accesses to a 510-Kbyte subset of the address space can be done through an immediate addressing. This will offer faster access to frequently used system variables.

### 3.9 Control Instructions

The N1 implements a set of instructions to control some of its internal components. None of these instructions consume input arguments from the [parameter stack](#), nor do they produce a return value. The encoding of these instructions is shown in [Table 3-3](#). Multiple control instructions can be combined to one.

Table 3-3: Control instructions

Encoding	Instruction
xxxxxx11	Enable interrupts
xxxxxx10	Disable interrupts
xxxxx1xx	Reset <a href="#">parameter stack</a>
xxxx1xxx	Reset <a href="#">return stack</a>

## 4 ANS Forth Words

Table 4-1 provides a list of standard ANS Forth words (see [?]) which directly map to hardware instructions of the N1 processor.

Table 4-1: ALU operations

Word	Stack	Description	Opcode
!	( x a-addr – )	Store cell	0000
*	( n1 u1 n2 u2 – n3 u3 )	Multiply two cells	0000
+	( n1 u1 n2 u2 – n3 u3 )	Add two cells	0000
-	( n1 u1 n2 u2 – n3 u3 )	Subtract a cell from another.	0000



## 5 Stacks

The N1 operates with two stacks: the [parameter stack](#) to perform data transactions and the [return stack](#) to manage the program flow. As illustrated in [Figure 5-1](#), each of these stacks consists of three hardware components: the [upper stack](#), the [intermediate stack](#), and the [lower stack](#).

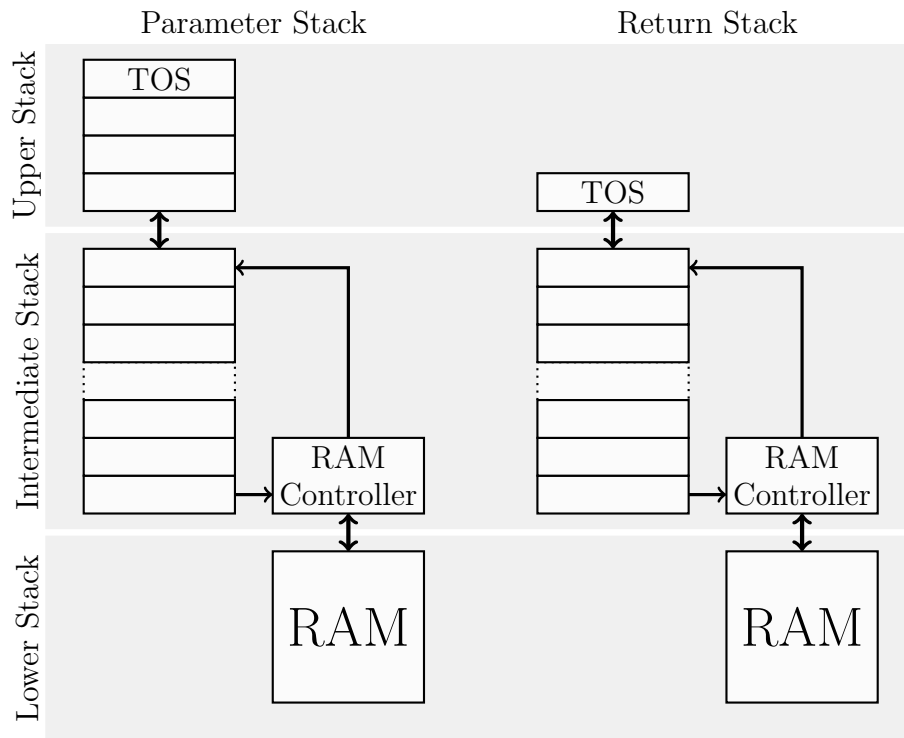


Figure 5-1: Stack Architecture

### 5.1 Parameter Stack

The [upper stack](#) of the [parameter stack](#) contains is four [cells](#) deep and contains the most recent data entries. It's purpose is to perform stack and ALU operations (see [Section 3.7 “Stack Instructions”](#) and [Section 3.6 “ALU Instructions”](#)). When the capacity of the [upper stack](#) is exceeded, older data entries are transferred to the [intermediate stack](#).

The [intermediate stack](#) serves as a buffer between the [upper stack](#) and the [lower stack](#) which resides in [RAM](#). The purpose of the [intermediate stack](#) is to minimize [RAM](#) traffic to and from the [lower stack](#). Push operation to the [intermediate stack](#) are only propagated to the [lower stack](#), when the buffer capacity is exceeded. Pull operations are only propagated, when the [intermediate stack](#) is empty. [Stack](#) fluctuations within the buffer capacity are not visible to the [lower stack](#).

The [lower stack](#) is a region of the [RAM](#), which is managed by the memory controller of the [intermediate stack](#).

## 5.2 Return Stack Stack

The [upper stack](#) of the [parameter stack](#) has the capacity of one [cell](#). The [intermediate stack](#) and [lower stack](#) are similar to the ones of the [parameter stack](#).

## References

- [1] Menlo Park James Bowman, Willow Garage. J1: a small forth cpu core for fpgas.  
<http://www.excamera.com/files/j1.pdf>, 2010.