



N1 Manual

Dirk Heisswolf

March 5, 2019

Revision History

Date	Change
March 5, 2019	Pre-release

Contents

1	Glossary	5
2	Overview	7
3	Instruction Set	8
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	11
3.8	Memory Access Instructions	14
3.9	Control Instructions	14
4	ANS Forth Words	16
5	Stacks	18
5.1	Parameter Stack	18
5.2	Return Stack Stack	19
6	Reset, Interrupts, and Exceptions	20
7	Integration Guide	21
8	Architecture	22
9	Verification Status	23
10	Tool Summary	24

List of Figures

3-1	Instruction encoding	8
3-2	Transition encoding of stack instructions	11
5-1	Stack Architecture	18

List of Tables

3-1	ALU operations	10
3-2	Common stack operations	12
3-2	Common stack operations	13
3-2	Common stack operations	14
3-3	Simple Control Instructions	14
3-4	Nonconcurrent control instruction encoding	14
3-4	Nonconcurrent control instruction encoding	15
4-1	ANS Forth words	16
4-1	ANS Forth words	17

1 Glossary

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

ALU
Arithmetic Logic Unit.

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 contols 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.

LIFO
A memory which is accessible in last in - first out order.

literal
A fixed numerical value within the program code.

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

LSB
The least significant bit.

MSB
The most significant bit.

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.

TOS

The top [cell](#) of a [stack](#).

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 refers to a callable code sequence in [Forth](#) terminology.

2 Overview

The N1 is a small stack machine, inspired by the J1 Forth CPU[2]. 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.

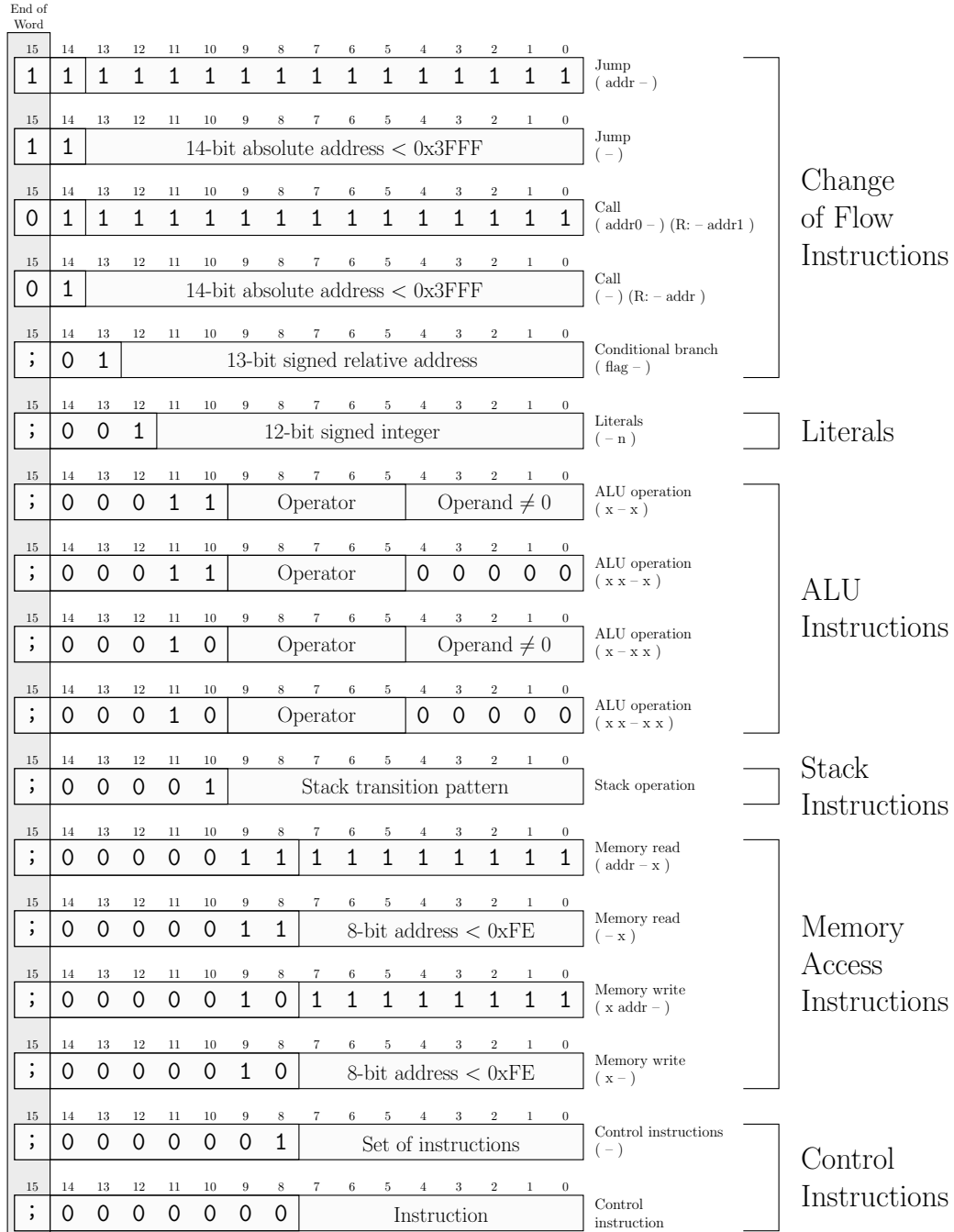


Figure 3-1: Instruction encoding

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 address location within the supported 128KB 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 address location within the supported 128KB 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*), a sign extended (*simm*), or an offsetted (*oimm*) integer value:

$$\begin{aligned}
uimm &= \text{opcode}[4:0] \\
simm &= \begin{cases} \text{opcode}[4:0], & \text{if } \text{opcode}[4:0] < 16 \\ \text{opcode}[4:0] - 32, & \text{if } \text{opcode}[4:0] \geq 16 \end{cases} \\
oimm &= \text{opcode}[4:0] - 16
\end{aligned}$$

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 XOR	$x1 \oplus simm$	$x1 \oplus x2$
10010	Logic OR	$x1 \vee uimm$	$x1 \vee x2$
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 a literal value	$simm, x1[11:0]$	$simm, x2[11:0]$
11001	Reserved		
11010	Reserved		
11011	Reserved		
11100	Reserved		
11101	Reserved		
11110	Reserved		
11111	Reserved		

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 patter is shown in **Figure 3-2**.

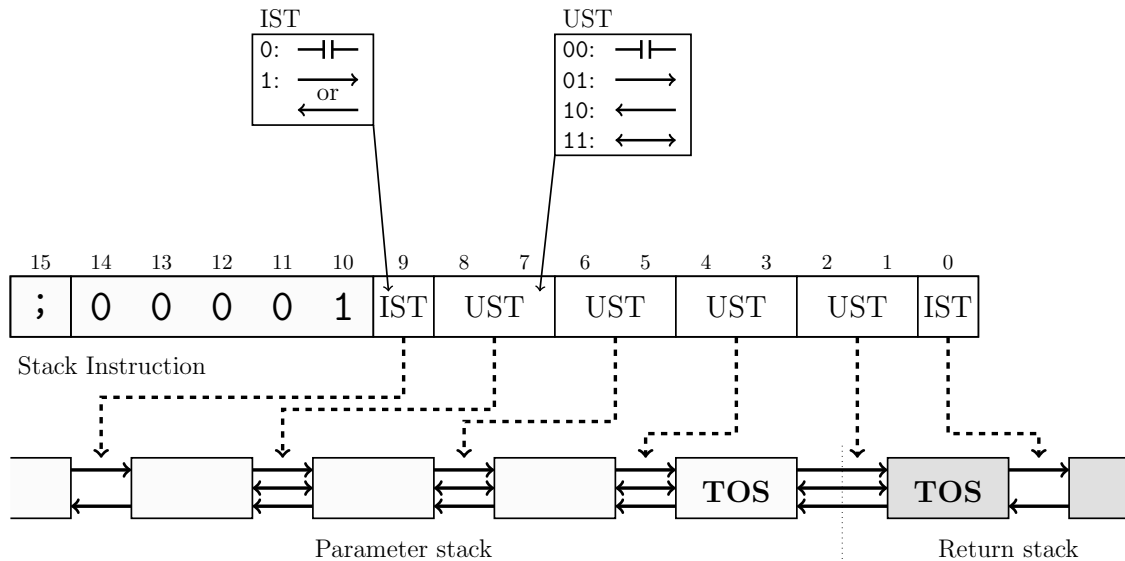


Figure 3-2: Transition encoding of stack instructions

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:

- 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 desination **cell**. In these cases, the resulting data in the desination **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
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



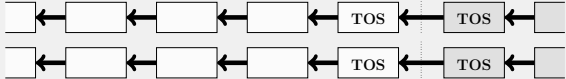
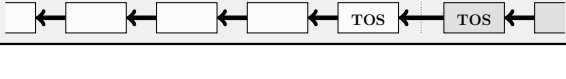
...continued

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)		0x06AB 0x06AB

...continued

Table 3-2: Common stack operations

Word	Description	Transitions	Opcode
2R@	$(-x1\ x2)$ $(R: x1\ x2 - x1\ x2)$		0x0755
			0x0757
2R>	$(-x1\ x2)$ $(R: x1\ x2 -)$		0x0755
			0x0755

3.8 Memory Access Instructions

Memory access instructions perform read or write accesses to the system's 128KB address space. Data is solely accessed in 16-bit entities. Accesses to a 510B 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 two types of control instructions to manipulate the internal state of the CPU. The first type are simple control instructions. These don't consume input from the stacks nor do they produce a return value. These instructions can be executed concurrently and combined into a single CPU instruction. [Table 3-3](#) shows the set of simple control instructions and their encoding.

Table 3-3: Simple Control Instructions

Encoding	Action
0b0000001xxxxxx1	Enable interrupts
0b0000001xxxxxx10	Disable interrupts
0b0000001xxxxxx1xx	Enable exceptions
0b0000001xxxxxx10xx	Disable exceptions
0b0000001xxx1xxxx	Reset parameter stack
0b0000001xx1xxxxx	Reset return stack

The second type of control instructions, trigger an internal sequence of actions and consume multiple clock cycles of execution time. [Table 3-4](#) lists the encoding of these complex control instructions.

Table 3-4: Nonconcurrent control instruction encoding

Encoding	Instruction
0x00FF	Fetch parameter stack pointer (= number of cells) (- +n)

...continued

Table 3-4: Nonconcurrent control instruction encoding

Encoding	Instruction
0x00FE	Store parameter stack pointer (+n -)
0xb00FD	Fetch return stack pointer (= number of cells) (- +n)
0x00FC	Store return stack pointer (+n -)
0x00FB	Execute pending ISR, even if interrupt requests are disabled (-)
0x00FA	Copy the current throw code on to the parameter stack . n = 0, if no exception has occurred. (- n)

4 ANS Forth Words

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

Table 4-1: ANS Forth words

Word	Stack	Description	Opcode
!	(x a-addr -)	Store a cell	0x0000
*	(n1 u1 n2 u2 - n3 u3)	Multiply two cells	0x0000
+	(n1 u1 n2 u2 - n3 u3)	Add two cells	0x0000
-	(n1 u1 n2 u2 - n3 u3)	Subtract a cell from another	0x0000
0<	(n - flag)	Check if cell is negative	0x0000
0<>	(x - flag)	Check if cell is not zero	0x0000
0>	(n - flag)	Check if cell is greater than zero	0x0000
0=	(x - flag)	Check if cell is zero	0x0000
1+	(n1 u1 - n2 u2)	Increment cell	0x0000
1-	(n1 u1 - n2 u2)	Decrement cell	0x0000
2*	(x1 - x2)	Shift cell one bit towards the MSB	0x0000
2/	(x1 - x2)	Shift cell one bit towards the LSB	0x0000
;	(-) (R: nest-sys -)	Return to the calling word	0x0000
<	(n1 n2 - flag)	Compare two signed numbers	0x0000
<>	(x1 x2 - flag)	Compare two cells for inequality	0x0000
=	(x1 x2 - flag)	Compare two cells for equality	0x0000
>	(n1 n2 - flag)	Compare two signed numbers	0x0000
>R	(x -) (R: - x)	Move a cell from the parameter stack to the return stack	0x0000
@	(a-addr - x)	Fetch a cell	0x0000
AND	(x1 x2 - x3)	Bitwise logic AND	0x0000
BL	(- char)	Space character	0x0000
CELL+	(a-addr1 - a-addr2)	Increment address	0x0000
CHAR+	(a-addr1 - a-addr2)	Increment address	0x0000
DEPTH	(- +n)	Number of cells on the parameter stack	0x0000
DROP	(x -)	Remove a cell from the parameter stack	0x0000
DUP	(x -)	Duplicate a cell on the parameter stack	0x0000
EXIT	(-) (R: nest-sys -)	Return to the calling word	0x0000
FALSE	(- false)	Return FALSE value	0x0000
INVERT	(x1 - x2)	Bitwise invert	0x0000
LSHIFT	(x1 u - x2)	Shift cell multiple bits towards the MSB	0x0000
M*	(n1 n2 - d)	Signed multiplication	0x0000
NEGATE	(n1 - n2)	Two's complement	0x0000
NIP	(x1 x2 - x2)	Drop the cell below the TOS	0x0000
OR	(x1 x2 - x3)	Bitwise logic OR	0x0000
OVER	(x1 x2 - x1 x2 x1)	Copy second parameter stack entry to the TOS	0x0000
R>	(- x) (R: x -)	Move a cell from the return stack to the parameter stack	0x0000
R@	(- x) (R: x - x)	Copy a cell from the return stack to the parameter stack	0x0000
RSHIFT	(x1 u - x2)	Shift cell multiple bits towards the LSB	0x0000
S>D	(n - d)	Sign-extend cell	0x0000
SWAP	(x1 x2 - x2 x1)	Swap two cells on the parameter stack	0x0000
TRUE	(- true)	Return TRUE value	0x0000

...continued

Table 4-1: ANS Forth words

Word	Stack	Description	Opcode
U<	(u1 u2 – flag)	Compare two unsigned numbers	0x0000
U>	(u1 u2 – flag)	Compare two unsigned numbers	0x0000
UM*	(u1 u2 – d)	Unsigned multiplication	0x0000
XOR	(x1 x2 – x3)	Bitwise exclusive-OR	0x0000

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](#).

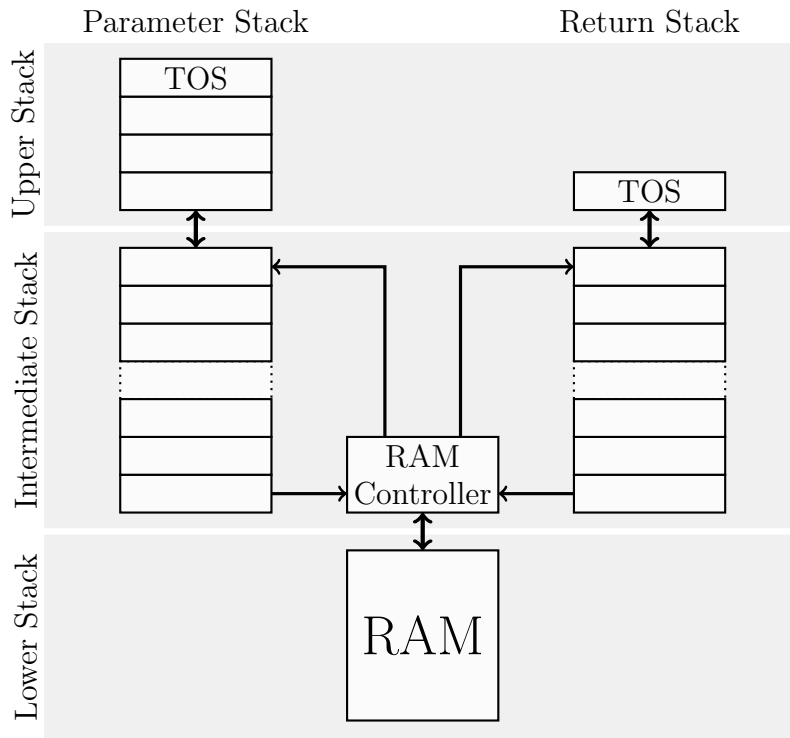


Figure 5-1: Stack Architecture

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 onle 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 a memory controller that is shared between the [parameter stack](#) and the [return stack](#). Within the [RAM](#) both stacks will grow towards each other. Moving cell content from one stack to the other ([>R](#) or [R>](#)) will never lead to a stack overflow.

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 identical to the ones of the [parameter stack](#).

6 Reset, Interrupts, and Exceptions

TBD

7 Integration Guide

TBD

8 Architecture

TBD

9 Verification Status

TBD

10 Tool Summary

TBD

References

- [1] American National Standard for Information Systems, 1994.
- [2] Menlo Park James Bowman, Willow Garage. J1: a small Forth CPU Core for FPGAs. <http://www.excamera.com/files/j1.pdf>, 2010.