N1 Manual

Dirk Heisswolf

March 13, 2019

Revision History

Date	Change
March 13, 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 19
- 5.1 Parameter Stack . . . 19
- 5.2 Return Stack Stack . . . 20

6 Reset, Exceptions, and Interrupts 21
- 6.1 Reset . . . 21
- 6.2 Exceptions . . . 21
- 6.3 Interrupts . . . 22

7 Integration Guide 23
- 7.1 Integratation Parameters . . . 23
- 7.2 Interfaces . . . 23
- 7.3 Target Specific Design Files . . . 25

8 Verification Status 26

9 Tool Summary 27

List of Figures

3-1 Instruction encoding 8
3-2 Transition encoding of stack instructions 11
5-1 Stack Architecture 19

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 Concurrent Control Instructions 14
3-4 Non-concurrent control instruction encoding 14
3-4 Non-concurrent control instruction encoding 15
4-1 ANS Forth words 16
4-1 ANS Forth words 17
4-1 ANS Forth words 18
6-1 Throw codes 22
9-1 Tool Summary 27

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 (see Section 3.3 “Call Instructions“).

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 (see Section 3.4 “Conditional Branches“).

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**intermediate **S**tack **T**ransition”.

jump

A change of the program flow without return option (see Section 3.2 “Jump Instructions“).

LIFO

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

literal

A fixed numerical value within the program code (see Section 3.5 “Literals“).

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.

throw code

A unique identifier for each type of exception.

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 contols 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 16-bit stack machine, targeted for low-end FPGA applications. It's instruction set and architecture is designed for efficient execution of Forth code.
Here is a summary of the N1's characteristics:

- Hardware support for parameter and return stack
  - Cells are 16 bit wide
  - Over and underflows monitored in hardware
  - Upper cells of both stacks stored in registers
    - Number of register based cells configurable via integration parameters
    - Shift direction of topmost cells individually controllable, supporting common stack operations without additional data paths in hardware
    - Data movement to and from RAM with hysteresis behavior to reduce bus accesses
  - Lower cells stored in RAM
    - Dedicated address space for stacks, up to 128KB in size
    - RAM space for stacks configurable via integration parameter
    - Stacks allocated to grow towards eachother
    - Dedicated whshbone interface for stack accesses

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 instructuion encoding.

15 (End of Word)	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	Operation	Group
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	Jump (addr –)	Change of Flow Instructions
1	1	14-bit absolute address < 0x3FFF														Jump (–)	
0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	Call (addr0 –) (R: – addr1)	
0	1	14-bit absolute address < 0x3FFF														Call (–) (R: – addr)	
;	0	1	13-bit signed relative address													Conditional branch (flag –)	
;	0	0	1	12-bit signed integer												Literals (– n)	Literals
;	0	0	0	1	1	Operator					Operand $\neq 0$					ALU operation (x – x)	ALU Instructions
;	0	0	0	1	1	Operator					0	0	0	0	0	ALU operation (x x – x)	
;	0	0	0	1	0	Operator					Operand $\neq 0$					ALU operation (x – x x)	
;	0	0	0	1	0	Operator					0	0	0	0	0	ALU operation (x x – x x)	
;	0	0	0	0	1	Stack transition pattern										Stack operation	Stack Instructions
;	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	Memory read (addr – x)	Memory Access Instructions
;	0	0	0	0	0	1	1	8-bit address < 0xFE								Memory read (– x)	
;	0	0	0	0	0	1	0	1	1	1	1	1	1	1	1	Memory write (x addr –)	
;	0	0	0	0	0	1	0	8-bit address < 0xFE								Memory write (x –)	
;	0	0	0	0	0	0	1	Set of instructions								Control instructions (–)	Control Instructions
;	0	0	0	0	0	0	0	Instruction								Control instruction	

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 operationis 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 ber 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 cab be either an absolute address placed on the the top of the Parameter stack or relative 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 reult 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:

$$uimm = \text{opcode}[4{:}0]$$

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

$$oimm = \text{opcode}[4{:}0] - 16$$

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	Absolute value	$oimm$ + ABS(x1)	x2 + ABS(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 ≠ $uimm$?	x1 ≠ x2?
01011	Not-equals comparison	$oimm$ ≠ x1?	x2 ≠ x1?
01100	Unsigned minimum	UMIN(x1, $uimm$)	UMIN(x1, x2)
01101	Signed maximum	MAX($oimm$, x1)	MAX(x2, x1)
01110	Unsigned maximum	UMAX(x1, $uimm$)	UMAX(x1, x2)
01111	Signed minimum	MIN($oimm$, x1)	MIN(x2, x1)
10000	Unsigned product	x1 ∗ $uimm$	x1 ∗ x2
10001	Unsigned product	x1 ∗ $simm$	x1 ∗ x2
10010	Signed product	x1 ∗ $uimm$	x1 ∗ x2
10011	Signed product	x1 ∗ $simm$	x1 ∗ x2
10100	Logic AND	x1 ∧ $simm$	x1 ∧ x2
10101	Logic XOR	x1 ⊕ $simm$	x1 ⊕ x2
10110	Logic OR	x1 ∨ $uimm$	x1 ∨ x2
10111	Reserved		
11000	Logic right shift	x1 ≫ $uimm$	x2 ≫ x1
11001	Logic left shift	x1 ≪ $uimm$	x2 ≪ x1
11010	Arithmetic right shift	x1 ≫ $uimm$	x2 ≫ x1
11011	Reserved		
11100	Set upper bits of a literal value	$simm$, x1[11:0]	$simm$, x2[11:0]
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.

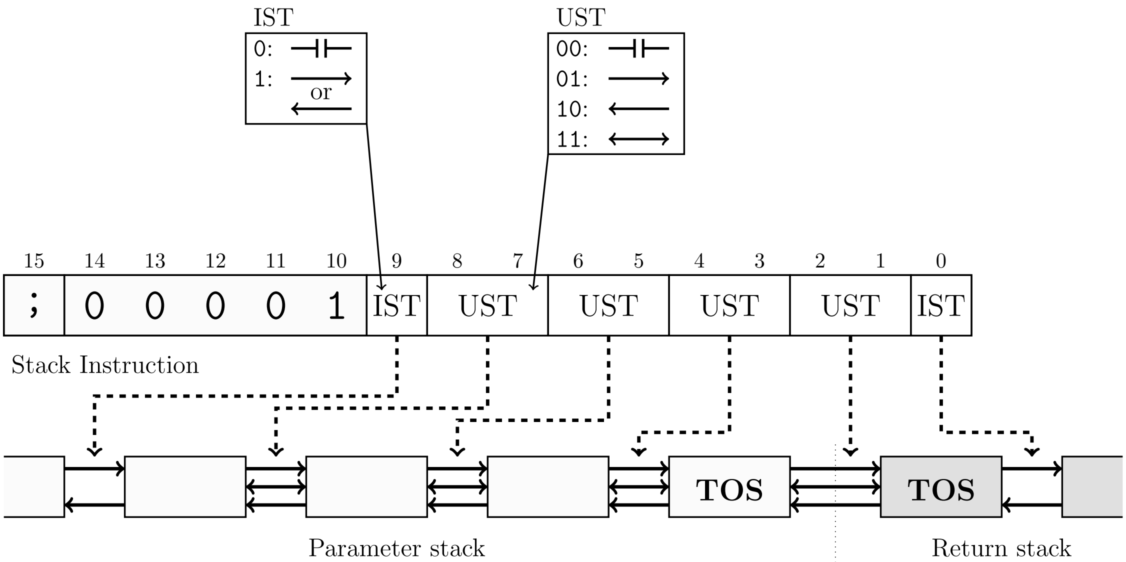


Figure 3-2: Transition encoding of stack instructions

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

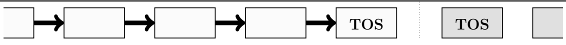
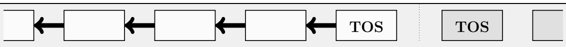
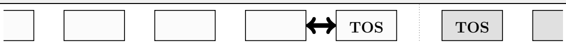
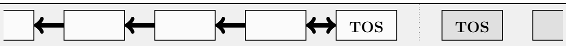
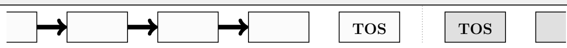
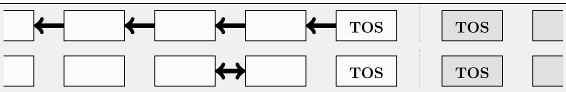
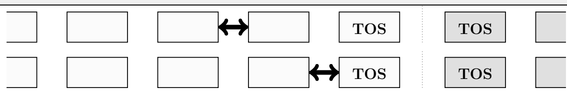
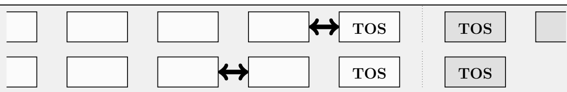
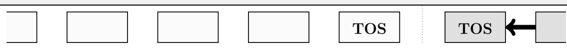
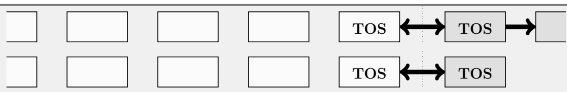
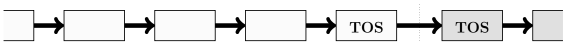
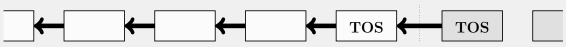
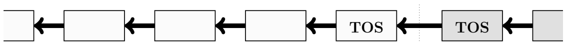
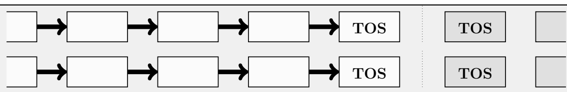
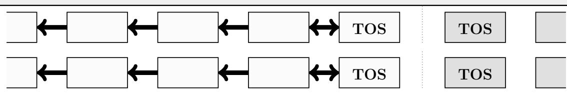
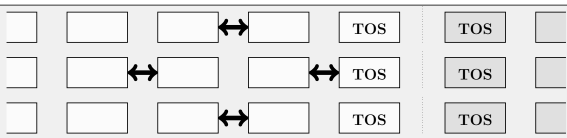
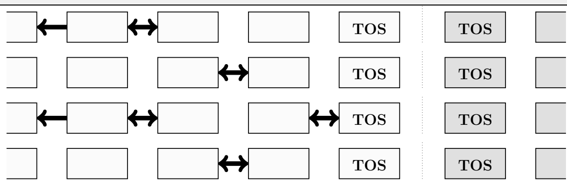
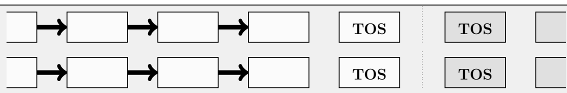
It is possible to put the UST fields into a combination which would trigger a data transfer of two source cells to a single desination cell. In these cases, the resulting data in the desination cell is undefined.

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.

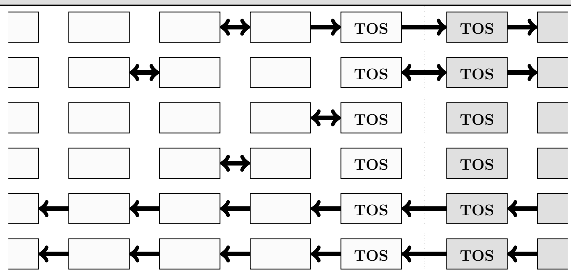
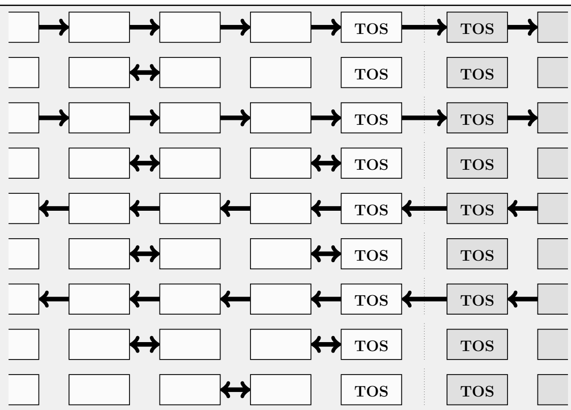
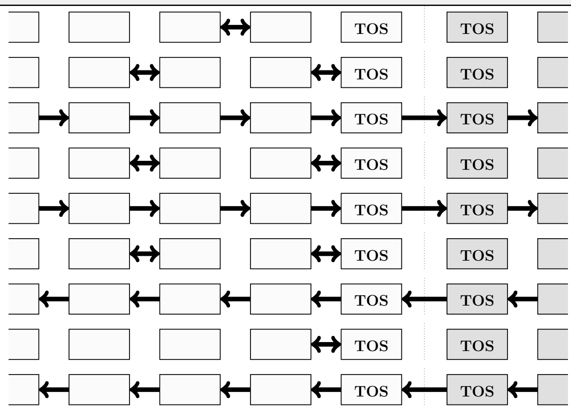
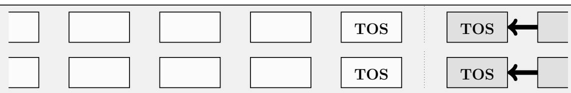
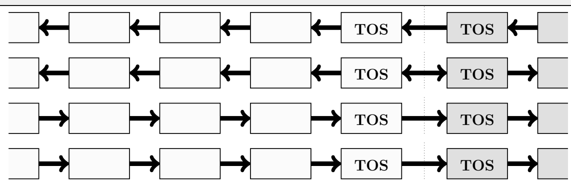
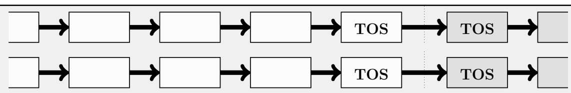
Table 3-2 shows how stack operations in Forth are mapped N1 instructions.

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)		0x0407 0x0406
>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

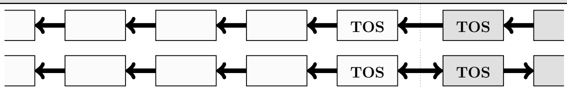
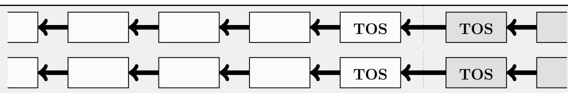
...continued

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 0x0418 0x0755
2RDROP	(R: x1 x2 –)		0x0401 0x0401
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 instruction perform read or write acesses 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 thir encoding.

Table 3-3: Concurrent Control Instructions

Encoding	Action
`0b0000001xxxxxxxx1`	Enable interrupts
`0b0000001xxxxxx10`	Disable interrupts
`0b0000001xxxxx1xx`	Enable exceptions
`0b0000001xxxx10xx`	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: Non-concurrent control instruction encoding

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

...continued

Table 3-4: Non-concurrent 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 −)

4 ANS Forth Words

The N1 processor aims at executing Forth code in an efficient way. Table 4-1 provides a list of standard ANS Forth[1] words which can be easily mapped to N1 instructions.

Table 4-1: ANS Forth words

<table>
<tr><th>Word</th><th>Stack</th><th>Description</th><th>Opcode</th></tr>
<tr><td>!</td><td>(x addr –)</td><td>Store x at addr</td><td>0x02FF</td></tr>
<tr><td>*</td><td>(n1|u1 n2|u2 – n3|u3)</td><td>Multiply n1|u1 by n2|u2</td><td>0x0E00</td></tr>
<tr><td>+</td><td>(n1|u1 n2|u2 – n3|u3)</td><td>Add n1|u1 to n2|u2</td><td>0x0C00</td></tr>
<tr><td>+!</td><td>(n1|u1 a-adr –)</td><td>Add n1|u1 to the cell at addr</td><td>0x0403
0x03FF
0x0C00
0x0755
0x02FF</td></tr>
<tr><td>-</td><td>(n1|u1 n2|u2 – n3|u3)</td><td>Subtract n2|u2 from n1|u1</td><td>0x0C40</td></tr>
<tr><td>0<</td><td>(n – flag)</td><td>Test if n is negative</td><td>0x0CF0</td></tr>
<tr><td>0<></td><td>(x – flag)</td><td>Test if x is not zero</td><td>0x0D70</td></tr>
<tr><td>0></td><td>(n – flag)</td><td>Test if n is greater than zero</td><td>0x0CA0</td></tr>
<tr><td>0=</td><td>(x – flag)</td><td>Test if x is not zero</td><td>0x0D70</td></tr>
<tr><td>1+</td><td>(n1|u1 – n2|u2)</td><td>Increment n1|u1</td><td>0x0C01</td></tr>
<tr><td>1-</td><td>(n1|u1 – n2|u2)</td><td>Decrement n1|u1</td><td>0x0C1F</td></tr>
<tr><td>2!</td><td>(x1 x2 addr –)</td><td>Store x2 at addr and x1 at addr+1</td><td>0x0750
0x0460
0x02FF
0x0C01
0x02FF</td></tr>
<tr><td>2*</td><td>(x1 – x2)</td><td>Shift x1 one bit towards the MSB</td><td>0x0F41</td></tr>
<tr><td>2/</td><td>(x1 – x2)</td><td>Shift x1 one bit towards the LSB, while the MSB remains unchanged</td><td>0x0F41</td></tr>
<tr><td>2@</td><td>(addr – x1 x2)</td><td>Fetch x2 from addr and x1 at addr+1</td><td>0x0750
0x0C01
0x03FF
0x0418
0x03FF</td></tr>
<tr><td>2DROP</td><td>(x1 x2 –)</td><td>Drop cell pair x1 x2</td><td>0x06A8
0x06A8</td></tr>
<tr><td>2DUP</td><td>(x1 x2 – x1 x2 x1 x2)</td><td>Drop cell pair x1 x2</td><td>0x0758
0x0758</td></tr>
<tr><td>2OVER</td><td>(x1 x2 x3 x4 – x1 x2 x1 x2 x3 x4 x1 x2)</td><td>Copy cell pair x1 x2 to the TOS</td><td>0x0750
0x0460
0x0789
0x0460</td></tr>
<tr><td>2>R</td><td>(x1 x2 –) (R: – x1 x2)</td><td>Shift cell pair x1 x2 to the return stack</td><td>0x06AB
0x06AB</td></tr>
<tr><td>2R></td><td>(– x1 x2) (R: x1 x2 –)</td><td>Shift cell pair x1 x2 to the parameter stack</td><td>0x0755
0x0755</td></tr>
</table>

...continued

Table 4-1: ANS Forth words

Word	Stack	Description	Opcode
2R@	(– x1 x2) (R: x1 x2 – x1 x2)	Copy cell pair x1 x2 to the parameter stack	0x0755 0x0757
2OVER	(x1 x2 x3 x4 x4 x5 x6 – x3 x4 x5 x6 x1 x2)	Rotate three cell pairs	0x06AB 0x0580 0x06AB 0x0598 0x0755 0x0598 0x0755 0x0598 0x0460
2SWAP	(x1 x2 x3 x4 – x3 x4 x1 x2)	Swap two cell pairs	0x0460 0x0598 0x0460
;	(–) (R: addr –)	Return to the calling word	0x8400
<	(n1 n2 – flag)	test if n1 is lower than n2	0x0CE0
<>	(x1 x2 – flag)	test if x1 is different than x2	0x0D40
=	(x1 x2 – flag)	test if x1 equals x2	0x0D00
>	(n1 n2 – flag)	test if n1 is greater than n2	0x0CA0
>R	(x –) (R: – x)	Shift x on to the return stack	0x06AB
?DUP	(x – 0|x x)	Duplicate x if it is not zero	0x0750 0x2001 0x06A8
@	(addr – x)	Fetch x from addr	0x03FF
ABS	(n – u)	Absolute vale of n	0x0C30
AND	(x1 x2 – x3)	Bitwise logic AND of x1 and x2	0x0E80
BL	(– char)	Space character	0x1020
CELL+	(addr1 – -addr2)	Increment addr1	0x0C01
DEPTH	(– +n)	+n is the number of cells on the parameter stack without +n	0x00FF
DROP	(x –)	Drop x from the /glsps	0x06A8
DUP	(x – x x)	Duplicate x	0x0750
FALSE	(*i**x xt – *j**x)	Execute xt	0x7FFF
FALSE	(– false)	FALSE flag	0x1000
I	(– n|u) (R: n|u – n|u)	Copy the innermost loop index (n|u) onto the parameter stack	0x0754
INVERT	(x1 – x2)	Bitwise inverse of x1	0x0EBF
J	(– n|u) (R: x n|u – x n|u)	Copy the next-outer loop index (n|u) onto the parameter stack	0x0755 0x0407
LSHIFT	(x1 u – x2)	Shift x1 u bits towards the MSB	0x0F20
M*	(n1 n2 – d)	Multiply n1 by n2	0x0A40
M+	(n1 n2 – d)	Add n1 to n2	0x0800
MAX	(n1 n2 – n3)	n3 is the greater of n1 and n2	0x0DB0
MIN	(n1 n2 – n3)	n3 is the lesser of n1 and n2	0x0DF0
NEGATE	(n1 – n2)	n2 is the two's complement of n1	0x0C70
NIP	(x1 x2 – x2)	Drop x1	0x06A0

...continued

Table 4-1: ANS Forth words

Word	Stack	Description	Opcode
`OR`	(x1 x2 – x3)	Bitwise logic OR of x1 and x2	0x0EC0
`OVER`	(x1 x2 – x1 x2 x1)	Copy x1 to the TOS	0x0758
`R>`	(– x) (R: x –)	Shift x to the parameter stack	0x0755
`R@`	(– x) (R: x – x)	Copy x to the parameter stack	0x0754
`RSHIFT`	(x1 u – x2)	Shift x1 u bits towards the LSB	0x0F00
`ROT`	(x1 x2 x3 – x2 x3 x1)	Rotate the three topmost cells	0x0460 0x0418
`S>D`	(n – d)	Sign-extend n	0x0A41
`SWAP`	(x1 x2 – x2 x1)	Swap x1 and x2	0x0418
`TRUE`	(– true)	TRUE flag	0x1FFF
`TUCK`	(x1 x2 – x2 x1 x2)	Copy x1 below x2	0x0750 0x0460
`U<`	(u1 u2 – flag)	test if u1 is lower than u2	0x0CC0
`U>`	(u1 u2 – flag)	test if u1 is greater than u2	0x0C80
`UM*`	(u1 u2 – d)	Multiply u1 by u2	0x0A00
`XOR`	(x1 x2 – x3)	Bitwise logic XOR of x1 and x2	0x0EA0

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.

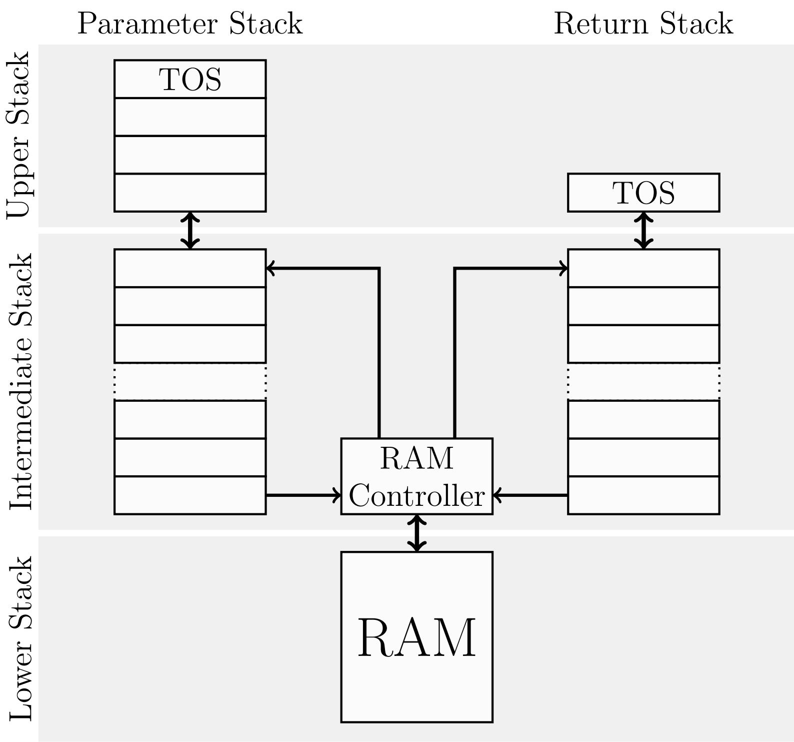


Figure 5-1: Stack Architecture

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, Exceptions, and Interrupts

There are three hardware mechanisms in the N1 processor, which can stop the ongoing program flow in order to react to an urgent hardware condition: Reset, Exceptions and Interrupts.

6.1 Reset

A reset puts the entire sequential logic of the N1 into a defined initial state. The return stack becomes completely cleared and the parameter stack is re-initialized to hold exactly one cell, containing the reset indicator `0x0000` (see Table 6-1). After reset, program execution will begin at address `0x0000`. Any context of the previous program flow is lost. Resets are generated by the system's hardware and occur at least once during power-up.

6.2 Exceptions

Exceptiona are triggered by error conditions and allow the software to restore the functionality of the system. There are five error conditions, which can be detected by the N1 hardware:

Parameter stack overflow
: A parameter stack overflow occurs when the capacity of the lower stack's RAM is exceeded (excluding a little margin, which is required for the error handling).

Return stack stack underflow
: A parameter stack underflow occurs then the stack when an instruction requires more arguments than available and when a stack instruction would resuly in non-continuous filling of the stack.

Return stack overflow
: A return stack overflow occurs when the capacity of the lower stack's RAM is exceeded (excluding a little margin, which is required for the error handling).

Return stack overflow
: A return stack underflow occurs then the stack when an instruction requires more arguments than available.

Address out of range
: This error condition indicates a memory access to a restricted address. This can either be caused by an instruction fetch or a data access

In any of these cases, the N1 processor will push a throw code onto the parameter stack (see Table 6-1), which is specific to the error condition and proceed with code execution at address `0x0000`. The return stack and the lower content of the parameter stack remain untouched. The context of the previous program execution is not reserved. To avoid reoccurance of error conditions during the execution of the handler routine, excepions are temorarily disabled after detection. Exceptions must then be reenabled by a control instruction (see Table 3-3), when the error is resolved. The throw codes listed in Table 6-1 comply with the exception word set of the ANS Forth standard[1].

Table 6-1: Throw codes

Throw Code	Condition
0x0000 (0)	Reset
0xFFFD (-3)	Parameter stack overflow
0xFFFC (-4)	Parameter stack underflow
0xFFFB (-5)	Parameter stack overflow
0xFFFA (-6)	Parameter stack underflow
0xFFF7 (-9)	Invalid memory address

The five hardware exceptions can be easily complemented by user defined software exceptions. A software exception can be thrown by pushing a unique throw code onto the parameter stack and performing a jump to address 0x0000. Hardware and software exceptions can then be handled by a common exception handler routine.

6.3 Interrupts

Interrupts are service requests, which are generated by the peripheral hardware. They cause a temporary interruption of the ongoing program flow. When an iterrupt occurs, the program counter is saved to the return stack and a interrupt service routine is executed. The location of the interrupt service routine is determined by the system's interrupt controller hardware. Further interrupts are automatically disabled during the execution of the interrupt service routine and must be manually reenabled by a control instruction (see Table 3-3) before resuming the prior program flow.

7 Integration Guide

This section explains the interfaces and configurations of the N1 processor for system integration.

7.1 Integratation Parameters

The N1 processor supports six integration parameters to configure the design for application specific needs:

`SP_WIDTH`
: Stack pointer width.

`IPS_DEPTH`
: Depth of the intermediate parameter stack.

`IRS_DEPTH`
: Depth of the intermediate return stack.

`PBUS_AADR_OFFSET`
: Offset for direct jump or call addressing.

`PBUS_MADR_OFFSET`
: Offset for direct data accesses.

`PS_RS_DIST`
: Safety distance between the parameter stack and the return stack.

7.2 Interfaces

7.2.1 Clock and Resets

`clk_i`
: Common clock input for all Wishbone interfaces.
This clock input corresponds to signal `CLK_I` of the Wishbone specification [2].

`async_rst_i`
: Optional asynchronous reset input for all sequential logic.
This reset signal may assert asynchronously, but must deassert synchronously.
If no asynchrounous reset is implemented, this input must be tied to zero.

`sync_rst_i`
: Synchronous reset input.
For WbXBC components, this synchronous reset is not required, if an asynchronous reset is provided. If no synchrounous reset is implemented, this input must be tied to zero. This reset input corresponds to signal `RST_I` of the Wishbone specification [2].

7.2.2 Program Bus

`pbus_cyc_o`
: Cycle indicator output.
This output signal corresponds to signal `CYC_O` of the Wishbone specification [2].

`pbus_stb_o`
: Strobe output.
This output signal corresponds to signal `STB_O` of the Wishbone specification [2].

`pbus_we_o`
: Write enable output.
This output signal corresponds to signal `WE_O` of the Wishbone specification [2].

`pbus_adr_o`
: Address bus.
These output signals correspond to bus `ADR_O` of the Wishbone specification [2].

`pbus_dat_o`
: Write data bus.
These output signals correspond to bus `DAT_O` of the Wishbone specification [2].

`pbus_tga_o`
: Address bus tags.
These output signals correspond to bus `TGA_O` of the Wishbone specification [2].

`pbus_ack_i`
: Acknowlede input.
This input signal corresponds to signal `ACK_I` of the Wishbone specification [2].

`pbus_err_i`
: Error indicator input.
This input signal corresponds to signal `ERR_I` of the Wishbone specification [2].

`pbus_stall_i`
: Pipeline stall input.
This input signal corresponds to signal `STALL_I` of the Wishbone specification [2].

`pbus_dat_i`
: Read data bus.
These input signals correspond to bus `DAT_I` of the Wishbone specification [2].

7.2.3 Stack Bus

`sbus_cyc_o`
: Cycle indicator output.
This output signal corresponds to signal `CYC_O` of the Wishbone specification [2].

`sbus_stb_o`
: Strobe output.
This output signal corresponds to signal `STB_O` of the Wishbone specification [2].

`sbus_we_o`
: Write enable output.
This output signal corresponds to signal `WE_O` of the Wishbone specification [2].

`sbus_adr_o`
: Address bus.
These output signals correspond to bus `ADR_O` of the Wishbone specification [2].

`sbus_dat_o`
: Write data bus.
These output signals correspond to bus `DAT_O` of the Wishbone specification [2].

`sbus_tga_o`
: Address bus tags.
These output signals correspond to bus `TGA_O` of the Wishbone specification [2].

`sbus_ack_i`
: Acknowlede input.
This input signal corresponds to signal `ACK_I` of the Wishbone specification [2].

`sbus_stall_i`
: Pipeline stall input.
This input signal corresponds to signal `STALL_I` of the Wishbone specification [2].

`sbus_dat_i`
: Read data bus.
These input signals correspond to bus `DAT_I` of the Wishbone specification [2].

7.2.4 Interrupt Interface

`irq_ack_o`
: Interrupt acknowledge.
TBD

`irq_req_i`
: Interrupt request.
TBD

7.2.5 Probe Signals

7.3 Target Specific Design Files

8 Verification Status

The implementation of the N1 design is currently still ongoing. Verification has not yet begun.

9 Tool Summary

One of the main goals of the N1 project is to use a design and verification flow, based on open source EDA tools. Table 9-1 summarizes the tools, used for this project.

Table 9-1: Tool Summary

Tool	Version	Usage
Verrilator[4]	3.874	Linting
Icarus Verilog[6]	0.9.7	Linting
Yosys[8]	0.7+627	Linting, Formal Verification
SymbiYosys[7]	Sep. 12, 2018	Formal Verification
GTKWave[3]	3.3.95	Waveform Viewer
Verilog-Perl[5]	3.418-1	Gerneration of design data for GTKWave[3]

References

[1] American National Standard for Information Systems, 1994.

[2] Wishbone b4. http://cdn.opencores.org/downloads/wbspec_b4.pdf, 2010.

[3] BSI. Gtkwave. http://gtkwave.sourceforge.net.

[4] Wilson Snyder. Verilator. http://www.veripool.org/verilator.

[5] Wilson Snyder. Verilog-perl. http://www.veripool.org/verilog-perl.

[6] Stephen Williams. Icarus verilog. http://iverilog.icarus.com.

[7] Clifford Wolf. Symbiyosys. https://github.com/cliffordwolf/SymbiYosys.

[8] Clifford Wolf. Yosys open synthesis suite. http://www.clifford.at/yosys.