



# N1 Manual

Dirk Heisswolf

June 15, 2025

---

## Revision History

Date	Change
March 21, 2019	Initial release
March 22, 2019	Changed encoding of ALU operands (see <a href="#">Table 2-1</a> and <a href="#">Table 4-1</a> )
April 17, 2019	Fixed some copy & paste errors
April 26, 2019	Fixes and a description of the branch condition in <a href="#">Section 2.4 “Conditional Branches”</a>
May 6, 2019	Added input <code>pbus_rty_i</code> Corrected and specified reference point for relative addresses
May 7, 2019	Changed definition of reference point for relative addresses
October 14 2019	Added <a href="#">Section 3 “Extensions”</a> Updated <a href="#">Section 7.1 “Integratation Parameters”</a>
November 11, 2022	Fixed typos Started chapter <a href="#">Section 8 “Architecture Description”</a>
January 4, 2024	Modified “Status and Control Instructions” in <a href="#">Figure 2-1</a> and <a href="#">Section 2.9 “Function Register Access”</a>
February 5, 2024	Extended sections <a href="#">Section 2.7 “Stack Instructions”</a> and <a href="#">Section 3.1 “ROT Extension”</a> , including a description of the stack underflow detection
February 12, 2024	Swapped order of operands in shift instructions (see <a href="#">Table 2-1</a> )
February 29, 2024	Added <a href="#">Section 2.9.3 “Throw Code Register (TC)”</a> , <a href="#">Section 3.2 “Interrupt Extension”</a> , <a href="#">Section 3.4 “KEY/EMIT Extension”</a> , and <a href="#">Section 4-2 “Non-standard Forth words”</a> Updated multiplication operators in <a href="#">Table 2-1</a>
April 24, 2025	Added <a href="#">Section 8.2 “Common Internal Interfaces”</a> and <a href="#">Section 8.2.1 “Stack Interface”</a> Removed “Stack Bus” in <a href="#">Section 7.2 “Interfaces”</a>
May 2, 2025	Added <a href="#">Section 8.2.2 “Memory Interface”</a>
May 14, 2025	Updated <a href="#">Section 8.2.2 “Memory Interface”</a>
May 15, 2025	Updated <a href="#">Section 8.1.1 “Naming Convention of Interface Signals”</a> Added <a href="#">Section 8.2.3 “Register Interface”</a> Updated <a href="#">Section 2.9 “Function Register Access”</a> mapping
June 6, 2025	Added <a href="#">Section 3.3 “Program Counter Extension”</a> Updated <a href="#">Section 2.4 “Conditional Branches”</a>
June 15, 2025	Reintroduced <a href="#">Section 2.9.3 “Throw Code Register (TC)”</a> Redefined reset and exception behavior in <a href="#">Section 6.1 “Reset”</a> , <a href="#">Section 6.2 “Exceptions”</a> , <a href="#">Table 6-1</a> Updated <a href="#">Section 3.2 “Interrupt Extension”</a>

## Contents

<b>1 Overview</b>	<b>5</b>
<b>2 Instruction Set</b>	<b>7</b>
2.1 Return from a Call (;)	8
2.2 Jump Instructions	8
2.3 Call Instructions	8
2.4 Conditional Branches	8
2.5 Literals	8
2.6 ALU Instructions	8
2.7 Stack Instructions	10
2.8 Memory Access Instructions	14
2.9 Function Register Access	14
<b>3 Extensions</b>	<b>17</b>
3.1 ROT Extension	17
3.2 Interrupt Extension	19
3.3 Program Counter Extension	19
3.4 KEY/EMIT Extension	19
<b>4 Forth Words</b>	<b>22</b>
4.1 ANS Forth Words	22
4.2 Non-Standard Forth Words	24
<b>5 Stacks</b>	<b>26</b>
5.1 Parameter Stack	26
5.2 Return Stack Stack	27
<b>6 Reset, Exceptions, and Interrupts</b>	<b>28</b>
6.1 Reset	28
6.2 Exceptions	28
6.3 Interrupts	29
<b>7 Integration Guide</b>	<b>30</b>
7.1 Integratation Parameters	30
7.2 Interfaces	31
7.3 Target Specific Design Files	33
<b>8 Architecture Description</b>	<b>34</b>
8.1 Design Principles	34
8.2 Common Internal Interfaces	34
8.3 Instruction Execution Cycle	37
8.4 Design Components	39
<b>9 Verification Status</b>	<b>42</b>
<b>10 Tool Summary</b>	<b>43</b>
<b>11 Glossary</b>	<b>44</b>
<b>12 References</b>	<b>47</b>

## List of Figures

2-1	Instruction encoding . . . . .	7
2-2	Transition encoding of stack instructions . . . . .	10
2-3	Transitions to and from the intermediate stack . . . . .	11
3-1	Stack transitions of the ROT extension . . . . .	17
3-2	Program Counter Register . . . . .	19
3-3	KEY? Register . . . . .	19
3-4	EMIT? Register . . . . .	20
3-5	KEY/EMIT Register . . . . .	20
5-1	Stack Architecture . . . . .	26
8-1	Plain Linear Execution . . . . .	38
8-2	Execution of an Extended Instruction . . . . .	38
8-3	Execution of a Memory Access Instruction . . . . .	39
8-4	Execution of a Change of Flow Instruction . . . . .	39
8-5	Program flow interrupted by an exception - TBD . . . . .	39
8-6	Block Diagram . . . . .	40

## List of Tables

2-1	ALU operations . . . . .	9
2-2	Common stack operations . . . . .	11
2-2	Common stack operations . . . . .	12
2-2	Common stack operations . . . . .	13
2-3	Rules of Stack Underflow Detection . . . . .	14
2-4	Function registers . . . . .	14
2-5	Parameter Stack Depth Register Bit Description . . . . .	15
2-6	Return Stack Depth Register Bit Description . . . . .	15
2-6	Return Stack Depth Register Bit Description . . . . .	16
2-7	Throw Code Register Bit Description . . . . .	16
3-1	Improved stack operations . . . . .	18
3-2	Rules of Stack Underflow Detection . . . . .	18
3-3	Exception and Interrupt Mask Register Bit Description . . . . .	20
3-4	Exception and Interrupt Mask Register Bit Description . . . . .	20
3-5	Exception and Interrupt Mask Register Bit Description . . . . .	21
4-1	ANS Forth words . . . . .	22
4-1	ANS Forth words . . . . .	23
4-1	ANS Forth words . . . . .	24
4-2	Non-standard Forth words . . . . .	25
6-1	Throw codes . . . . .	29
10-1	Tool Summary . . . . .	43

## 1 Overview

The N1 is a 16-bit stack machine, targeted for low-end FPGA applications. Its instruction set and architecture are designed for efficient execution of [Forth](#) code.

Here is a summary of the N1's characteristics:

### Memory connection:

- 16-bit [Von-Neumann-Architecture](#)
- Separate address space for [stack](#) content
- [Wishbone](#) interfaces to main and stack memory
- Up to 128KB (main) memory space
- Memory addressable in 16-bit entities only

### Stacks:

- Two hardware [stacks](#) ([parameter](#) and [return stack](#))
- Each [stack](#) consists of three segments:

#### [Upper stack:](#)

- Shift registers with selectable shift direction for each individual cell
- Fixed size
  - \* Upper [parameter stack](#): 4 [cells](#)
  - \* Upper [return stack](#): 1 [cell](#)

#### [Intermediate stack:](#)

- Buffer with lazy data transfers to and from the lower stack
- Configurable size

#### [Lower stack:](#)

- [RAM](#) space shared by both [stacks](#)
- [Stacks](#) grow towards each other
- Up to 128KB in size

### Instruction set:

- Fixed instruction size of 16-bit
- [Jumps](#) and [calls](#)
  - [Indirect addressing](#)
  - [Direct addressing](#) within a 32KB window
  - Two bus cycle execution time
  - Return from [calls](#) performed concurrently with last instruction
- [Conditional branches](#)
  - [Direct relative addressing](#) within a 16KB range
  - Two bus cycles of execution time if branch is taken, one cycle if not
- [Literals](#)
  - [Immediate](#) encoding of literals between -2048 and 2047
  - Literals out of this range require one additional instruction
- [Arithmetic and logic operations](#)
  - Single cycle [ALU](#) operations include:

- \* Sum and Difference
  - \* Comparisons
  - \* Signed and unsigned products
  - \* Bitwise logic operations
  - \* Multi-bit shifts
- Optional **immediate** encoding of one operand, using 5-bit encoding
- **Stack** operations
  - All 1024 stack transitions of the **upper stack** encodable
- Memory I/O
  - **Indirect addressing**
  - **Direct addressing** within a 511B window
  - Two bus cycle execution time if branch is taken, one cycle if not

**Exceptions:**

- Exception handler invoked by five error conditions:
  - **Parameter stack** overflow
  - **Parameter stack** underflow
  - **Return stack** overflow
  - **Return stack** underflow
  - Access violations in the (main) address space

**Interrupts:**

- Optional interrupt support through external interrupt controller
- Automatic interrupt acknowledge (flag clearing) supported

## 2 Instruction Set

The intent of the N1's instruction set is to map most of the essential Forth words to single cycle instructions. [Figure 2-1](#) illustrates the instruction format.

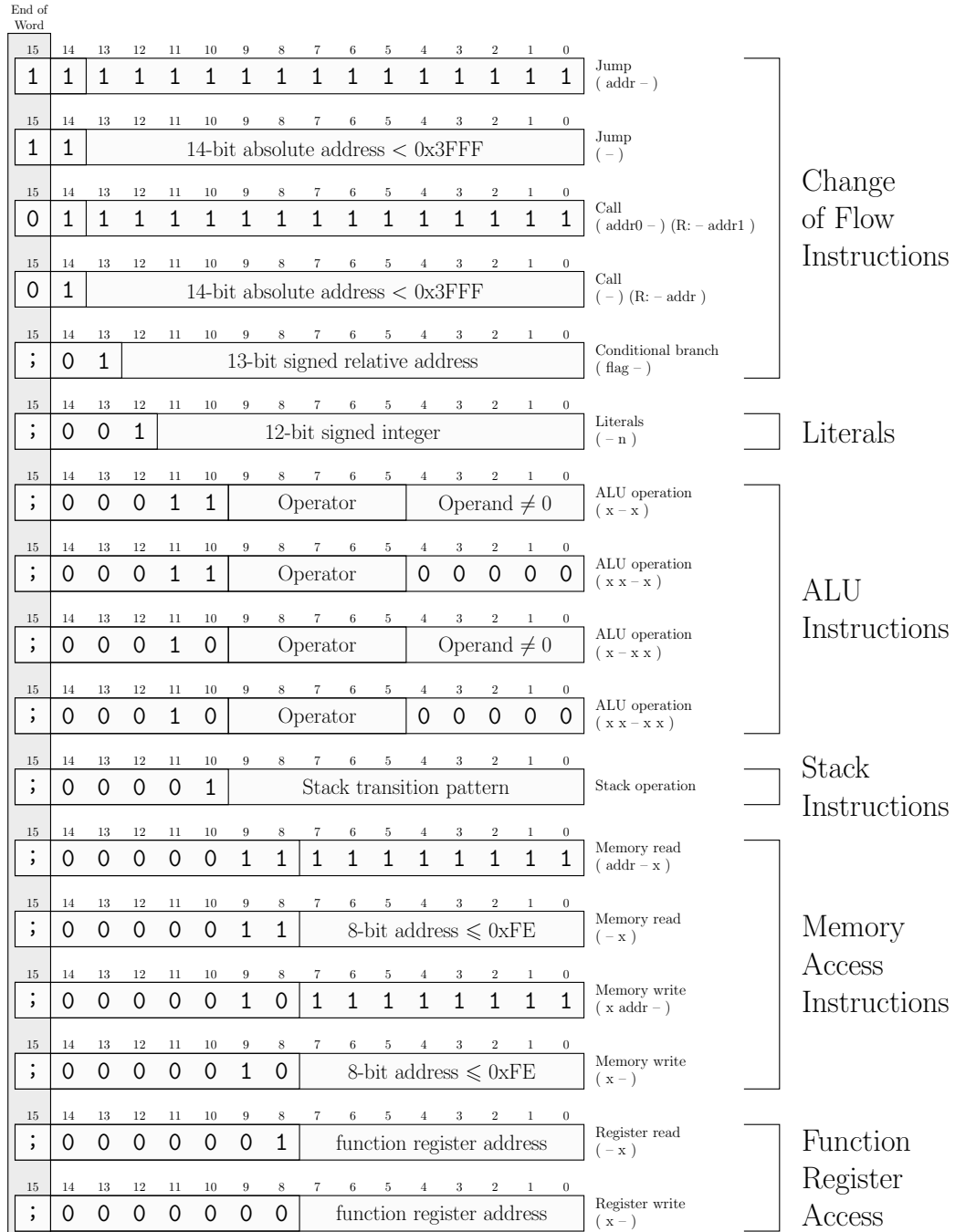


Figure 2-1: Instruction encoding



## 2.1 Return from a Call (;)

Rather than providing a dedicated instruction to end the execution of word in Forth and to return the caller's program flow, 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 is the last operation of 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 2-1](#), bit 15 is also distinction between the encoding of [jump](#) and of [call](#) instructions. Considering that the last [call](#) in a word definition can be optimized to a [jump](#), bit 15 can be regarded as the termination bit for [call](#) instructions as well. For a Forth compiler, this means that the semi-colon (;) always translates to setting bit 15 of the last instruction.

## 2.2 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 either be placed on the top of the [parameter stack](#) or encoded into the opcode of the instruction (only for destination addresses < 0x3FFF).

## 2.3 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 either be placed on the top of the [Parameter stack](#) or encoded into the opcode of the instruction (only for destination addresses < 0x3FFF).

## 2.4 Conditional Branches

[Conditional branches](#) invoke a change of program flow depending on the argument at the [top](#) of the [parameter stack](#). If it is zero, then the branch is taken. The branch destination is a [relative address](#), encoded into the opcode of the instruction in the range of  $\pm 8\text{KB}$ . A relative address of value zero points to the instruction following the [conditional branches](#). A "return from call" (;) is only executed if the branch is not taken.

## 2.5 Literals

Signed integer [literals](#) of 12-bit length can be pushed onto the [parameter stack](#) within a single instruction. For larger integers a supplemental [ALU](#) instruction is required. (see encoding 11100 in [Table 2-1](#))

## 2.6 ALU Instructions

[ALU](#) instructions perform an operation on two [cell](#) values, resulting in a new double [cell](#) value. The result can either be 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 either be 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. The [immediate](#) value

is interpreted as either 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 2-1 lists the supported ALU operations.

Table 2-1: ALU operations

Encoding	Operation	( x1 - d )	( x1 x2 - d )
00000	Sum	$uimm + x1$	$x1 + x2$
00001	Absolute value	$oimm + \text{ABS}(x1)$	$x1 + \text{ABS}(x2)$
00010	Difference	$x1 - uimm$	$x2 - x1$
00011	Difference	$oimm - x1$	$x1 - x2$
00100	Unsigned minimum value	$\text{UMIN}(uimm, x1)$	$\text{UMIN}(x1, x2)$
00101	Signed maximum value	$\text{MAX}(oimm, x1)$	$\text{MAX}(x1, x2)$
00110	Unsigned maximum value	$\text{UMAX}(uimm, x1)$	$\text{UMAX}(x1, x2)$
00111	Signed minimum value	$\text{MIN}(oimm, x1)$	$\text{MIN}(x1, x2)$
01000	Equals comparison	$uimm = x1?$	$x1 = x2?$
01001	Equals comparison	$oimm = x1?$	$x1 = x2?$
01010	Not-equals comparison	$uimm \neq x1?$	$x1 \neq x2?$
01011	Not-equals comparison	$oimm \neq x1?$	$x1 \neq x2?$
01100	Unsigned greater-than comparison	$uimm > x1?$	$x1 > x2?$
01101	Signed lower-than comparison	$oimm < x1?$	$x1 < x2?$
01110	Unsigned lower-than comparison	$uimm < x1?$	$x1 < x2?$
01111	Signed greater-than	$oimm > x1?$	$x1 > x2?$
10000	Unsigned product	$uimm * x1$	$x1 * x2$
10001	Signed product	$simm * x1$	$x1 * x2$
10010	Reserved		
10011	Reserved		
10100	Logic AND	$simm \wedge x1$	$x1 \wedge x2$
10101	Logic XOR	$simm \oplus x1$	$x1 \oplus x2$
10110	Logic OR	$uimm \vee x1$	$x1 \vee x2$
10111	Reserved		
11000	Logic right shift	$x1 \gg uimm$	$x1 \gg x2$
11001	Logic left shift	$x1 \ll uimm$	$x1 \ll x2$
11010	Arithmetic right shift	$x1 \gg uimm$	$x1 \gg x2$
11011	Reserved		
11100	Set upper bits of a literal value	$simm, x1[11:0]$	$simm, x2[11:0]$
11101	Reserved		
11110	Reserved		
11111	Reserved		

## 2.7 Stack Instructions

The N1's stack instruction aims at efficiently implementing common stack operations of the [Forth](#) language, while only implementing the essential data paths, which are needed for plain push and pull operations.

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

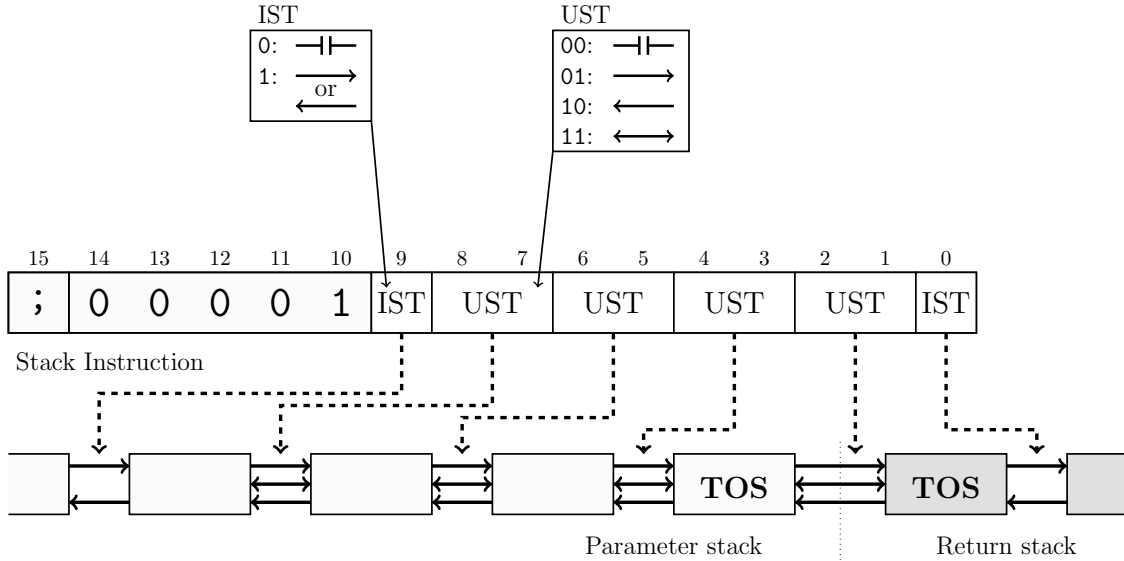


Figure 2-2: Transition encoding of stack instructions

The stack instruction contains four **UST** fields which control the data movement within the upper four [cells](#) of the [parameter stack](#) and the top [cell](#) 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 destination [cell](#). These combinations are reserved for instruction set extensions (see [Section 3 "Extensions"](#)). If no related instruction set extension is implemented, the outcome of these stack operations is undefined. In practice, the resulting data in the destination [cell](#) is then a logic OR of all sources.

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](#) (see [Figure 2-3](#)).

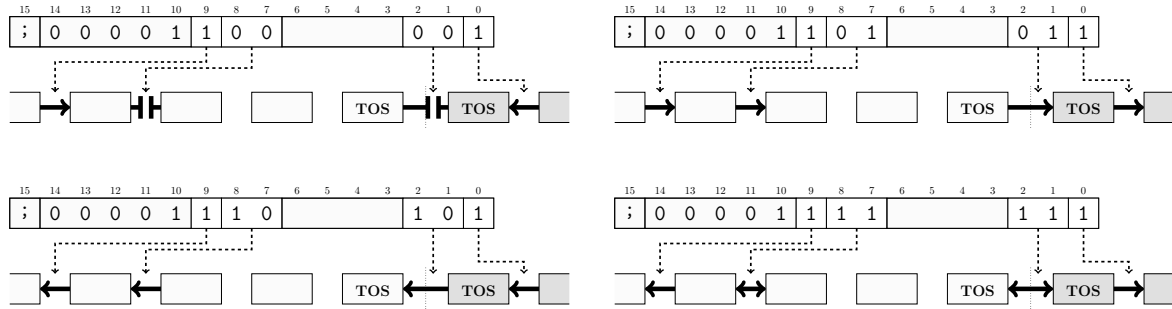


Figure 2-3: Transitions to and from the intermediate stack

### 2.7.1 Common Stack Operations

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

Table 2-2: 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 - )		0x0401
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

...continued

Table 2-2: Common stack operations

Word	Description	Transitions	Opcode
2DUP	( x1 x2 – x1 x2 x1 x2 )		0x0758
			0x0758
2SWAP	( x1 x2 x3 x4 – x3 x4 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
2TUCK	( x1 x2 x3 x4 – x3 x4 x1 x2 x3 x4 )		0x046B
			0x0587
			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

...continued

Table 2-2: Common stack operations

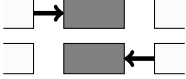
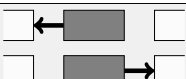
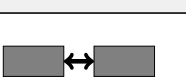
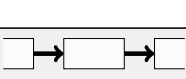
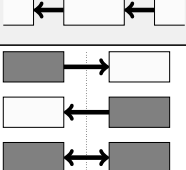
Word	Description	Transitions	Opcode
-2ROT	( x1 x2 x3 x4 x5 x6 - x5 x6 x1 x2 x3 x4 )		0x0460 0x0598 0x06AB 0x0598 0x06AB 0x0598 0x0755 0x0418 0x0755
2RDROP	( R: x1 x2 - )		0x0401 0x0401
2RDUP	( R: x1 x2 - x1 x2 x1 x2 )		0x0755 0x0757 0x06AB 0x06AB
2>R	( x1 x2 - ) ( R: - x1 x2 )		0x06AB 0x06AB
2R@	( - x1 x2 ) ( R: x1 x2 - x1 x2 )		0x0755 0x0757
2R>	( - x1 x2 ) ( R: x1 x2 - )		0x0755 0x0755

### 2.7.2 Stack Underflow Detection

The required number of arguments for a stack instruction is determined by the transition pattern (UST and IST fields). The rules listed in Table 3-2 are applied.

subsubsectionCommon Stack Operations Table 2-2 shows how common *stack* operations in *Forth* are mapped N1 instructions.

Table 2-3: Rules of Stack Underflow Detection

Rule	Transitions	Description
“DROP” Rule		A cell that will be dropped (overwritten, without passing on it’s content) from either direction must hold content, otherwise a stack underflow exception will be triggered.
“DUP” Rule		A cell that will be duplicated in either direction must hold content, otherwise a stack underflow exception will be triggered.
“SWAP” Rule		Two neighboring cells that will be swapped must both hold content, otherwise a stack underflow exception will be triggered.
“Shift” Rule		Cells that will be shifted in either direction, will not be checked for content
“Cross” Rule		Any cell that is shifted, swapped or duplicated across the <a href="#">parameter stack/return stack</a> boundary must hold content, otherwise a stack underflow exception will be triggered.

## 2.8 Memory Access Instructions

Memory access instruction perform read or write accesses to the system’s 64K word (128KB) address space. Data is solely accessed in 16-bit entities. Accesses to a 255 word (510B) window in the main address space, can be done through an immediate addressing. This offers faster access to frequently used system variables.

## 2.9 Function Register Access

The N1 processor provides a set [function registers](#), which provide access to a number of processor features (see [Table 2-4](#)). These [function registers](#) are read and written through special [opcodes](#).

Table 2-4: [Function registers](#)

Address	Mnemonic	Name
0x00	<a href="#">PSD</a>	<a href="#">Parameter Stack Depth Register</a>
0x01	<a href="#">RSD</a>	<a href="#">Return Stack Depth Register</a>
0x02	<a href="#">TC</a>	<a href="#">Throw Code Register</a>
0x03	<a href="#">PC</a>	<a href="#">Optional Program Counter Register</a>
0x04	<a href="#">KEY?</a>	<a href="#">Optional KEY? Register</a>
0x05	<a href="#">EMIT?</a>	<a href="#">Optional EMIT? Register</a>
0x06	<a href="#">KEY</a>	<a href="#">Optional KEY/EMIT Register</a>

### 2.9.1 Parameter Stack Depth Register (PSD)

Offset: 0x00

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Read	PSD															
Write	PSDCLR															

The current depth of the [parameter stack](#) is captured in the parameter stack depth register.

Writing 0x0000 to this register will clear the [parameter stack](#).

Table 2-5: Parameter Stack Depth Register Bit Description

Bit	Postion	Description
PSD	15..0	<b>Parameter Stack Depth</b> <b>Read:</b> Current parameter stack depth
PSDCLR	15..0	<b>Clear Parameter Stack</b> <b>Write:</b> Writing the value 0x0000 will clear the <a href="#">parameter stack</a> . All other write accesses have no effect.

### 2.9.2 Return Stack Depth Register (RSD)

Offset: 0x01

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Read	RSP															
Write	RSPCLR															

The current depth of the [return stack](#) is captured in the parameter stack depth register.

Writing 0x0000 to this register will clear the [return stack](#).

Table 2-6: Return Stack Depth Register Bit Description

Bit	Postion	Description
RSD	15..0	<b>Return Stack Depth</b> <b>Read:</b> Current return stack depth.

...continued



Table 2-6: Return Stack Depth Register Bit Description

Bit	Postion	Description
RSDCLR	15..0	<b>Return Stack Depth</b> <b>Write:</b> Writing the value 0x0000 will clear the <a href="#">return stack</a> . All other write accesses have no effect.

### 2.9.3 Throw Code Register (TC)

Offset: 0x02

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Read	TC															
Write	THROW															

The Throw Code Register (TC) captures the [throw code](#) of the last exception (see [Table 6-1](#)). [Throw code](#) values 0x0000 and 0x0001 indicate, that there is no pending interrupt or exception (see [Table 6-1](#)). Writing any value other than 0x0000 and 0x0001 to this register will trigger an exception (see [Section 6.2 “Exceptions”](#)). With the help of the [Interrupt Extension](#), exceptions can also be triggered by external hardware events.

Table 2-7: Throw Code Register Bit Description

Bit	Postion	Description
TC	15..0	<b>Throw Code</b> <b>Read:</b> <a href="#">throw code</a> of the last exception.
THROW	15..0	<b>Throw Trigger</b> <b>Write:</b> Triggers an exception with the written <a href="#">throw code</a> .

### 3 Extensions

The instruction set of the N1 processor (see [Section 2 “Instruction Set”](#)) reserves a number of undefined [opcodes](#) for functional extensions. These extensions imply a trade-off between hardware complexity and functional improvements. They can be selected individually for each system integrating the N1 processor (see [Section 7 “Integration Guide”](#)).

#### 3.1 ROT Extension

The [ROT extension](#) adds two data paths to the [upper stack](#), allowing direct data transfers between the top and the third element of the [parameter stack](#). These new stack transitions are performed by the regular stack instructions (see [Section 2.7 “Stack Instructions”](#)), using some of the reserved stack transition patterns. [Figure 3-1](#) illustrates the usage of the [ROT extension](#).

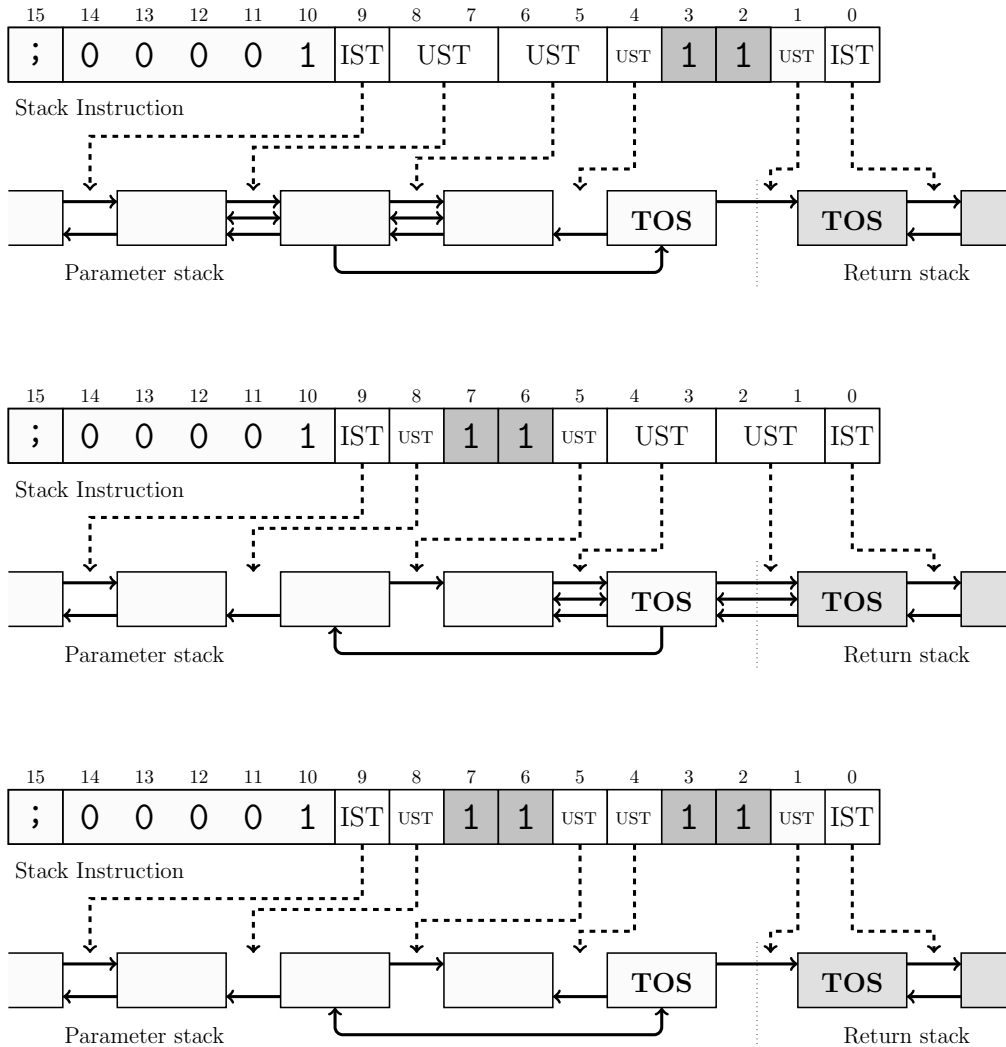


Figure 3-1: Stack transitions of the [ROT extension](#)

### 3.1.1 Accelerated Stack Operations

The [ROT extension](#) improves the execution time and code density of the three common stack operations TUCK, ROT, and -ROT (see [Table 3-1](#)). This means that all common single-cell [parameter stack](#) operations shown in [Table 2-2](#) can be executed in one cycle if the [ROT extension](#) is enabled.

Table 3-1: Improved stack operations

Word	Description	Transitions	Opcode
TUCK	( x1 x2 - x2 x1 x2 )		0x07C0
ROT	( x1 x2 x3 - x2 x3 x1 )		0x041C
-ROT	( x1 x2 x3 - x3 x1 x2 )		0x04E0

N1 processors with [ROT extension](#) are backward compatible to the ones without. All stack operations can still be executed as listed in [Table 2-2](#), even if the [ROT extension](#) is enabled.

### 3.1.2 Stack Underflow Detection

The [ROT extension](#) introduces three new stack underflow detection rules. These rules are listed in [Table 3-2](#).

Table 3-2: Rules of Stack Underflow Detection

Rule	Transitions	Description
“TUCK” Rule		If the downward <a href="#">ROT extension</a> path is used and the target cell is shifted further downward, then the <a href="#">parameter stack</a> must hold at least <b>two</b> values prior to the stack operation.
“ROT” Rule		For all other stack operations, which use any of the <a href="#">ROT extension</a> paths and for which the “TUCK” rule does not apply, the <a href="#">parameter stack</a> must hold at least <b>three</b> values prior to the stack operation.

### 3.2 Interrupt Extension

Interrupt are exceptions which are triggered by external hardware events. They are support by N1 processor with optional Interrupt Extension. If available, hardware, external to the processor, is able to trigger exceptions as described in [Section 6.2 “Exceptions”](#). Interrupts are only enabled if the Throw Code Register (TC) holds the value 0x0000. By setting the Throw Code Register (TC) to 0x0001, interrupts can be blocked, without triggering a new exception.

### 3.3 Program Counter Extension

The Program Counter Extension maps the PC into the [function register](#) space. The Program Counter Register (PC) (see [Figure 3-2](#)) always reads the PC of the current instruction. Write accesses are not supported. With the help of this extension, the N1 processor is able to perform relative jumps.

Offset: 0x03

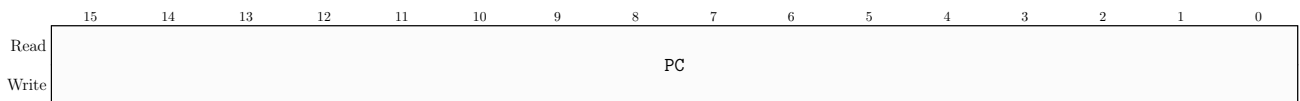


Figure 3-2: Program Counter Register

### 3.4 KEY/EMIT Extension

The KEY/EMIT extension adds support for an I/O device to the N1 core. [Forth](#) words KEY, KEY?, EMIT, and EMIT? dre directly mapped to a set of [function register](#).

#### 3.4.1 KEY? Register

The KEY? register (see [Figure 3-3](#)) indicates whether there is data to be received from the input device.

Offset: 0x04

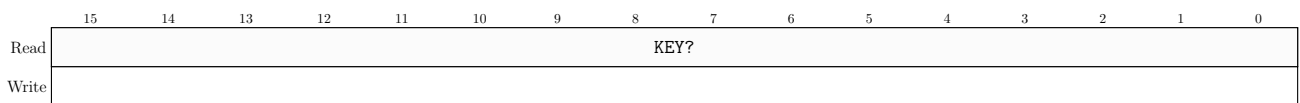


Figure 3-3: KEY? Register

The KEY? field in this read only register resembles the functionality of the KEY? word (see [Table 3-4](#).

Table 3-3: Exception and Interrupt Mask Register Bit Description

Bit	Postion	Description
KEY?	15..0	<b>Input Data Available Query</b> <b>Read:</b> <b>true:</b> Input data is available <b>false:</b> Input data is not available

### 3.4.2 EMIT? Register

The EMIT? (see [Figure 3-4](#)) register indicates whether the I/O device is ready to transmit data.

Offset: 0x05

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Read	EMIT?															
Write																

Figure 3-4: EMIT? Register

The EMIT? field in this read only register resembles the functionality of the EMIT? word (see [Table ??](#)).

Table 3-4: Exception and Interrupt Mask Register Bit Description

Bit	Postion	Description
EMIT?	15..0	<b>Output Device Ready Query</b> <b>Read:</b> <b>true:</b> Output device is ready to transmit data <b>false:</b> Output device is not ready to transmit data

### 3.4.3 KEY/EMIT Register

The KEY/EMIT register (see [Figure 3-5](#)) is used to receive data from and to transmit data to the I/O device. Accesses to this register are blocking.

Offset: 0x06

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Read	KEY															
Write	EMIT															

Figure 3-5: KEY/EMIT Register

The KEY field resembles the KEY word and the EMIT field resembles the EMIT word (see [Table 3-5](#)),

Table 3-5: Exception and Interrupt Mask Register Bit Description

Bit	Postion	Description
KEY	15..0	<b>Input Data</b> <b>Read:</b> Input data, removed from input device when read.
EMIT	15..0	<b>Output Data</b> <b>Read:</b> Output data

## 4 Forth Words

The following sections show recommended implementations of [Forth](#) words.

### 4.1 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 directly mapped to N1 instructions.

Table 4-1: ANS Forth words

Word	Stack	Description	Opcode
!	( x addr - )	Store x at addr	0x02FF
*	( n1 u1 n2 u2 - n3 u3 )	Multiply n1 u1 by n2 u2	0x0E00
+	( n1 u1 n2 u2 - n3 u3 )	Add n1 u1 to n2 u2	0x0C00
+	( n1 u1 a-adr - )	Add n1 u1 to the cell at addr	0x0403 0x03FF 0x0C00 0x0755 0x02FF
-	( n1 u1 n2 u2 - n3 u3 )	Subtract n2 u2 from n1 u1	0x0C40
0<	( n - flag )	Test if n is negative	0x0DF0
0<>	( x - flag )	Test if x is not zero	0x0D70
0>	( n - flag )	Test if n is greater than zero	0x0DB0
0=	( x - flag )	Test if x is not zero	0x0D30
1+	( n1 u1 - n2 u2 )	Increment n1 u1	0x0C01
1-	( n1 u1 - n2 u2 )	Decrement n1 u1	0x0C1F
2!	( x1 x2 addr - )	Store x2 at addr and x1 at addr+1	0x0750 0x0460 0x02FF 0x0C01 0x02FF
2*	( x1 - x2 )	Shift x1 one bit towards the <a href="#">MSB</a>	0x0F41
2/	( x1 - x2 )	Shift x1 one bit towards the <a href="#">LSB</a> , while the <a href="#">MSB</a> remains unchanged	0x0F40
2@	( addr - x1 x2 )	Fetch x2 from addr and x1 at addr+1	0x0750 0x0C01 0x03FF 0x0418 0x03FF
2DROP	( x1 x2 - )	Drop cell pair x1 x2	0x06A8 0x06A8
2DUP	( x1 x2 - x1 x2 x1 x2 )	Duplicate cell pair x1 x2	0x0758 0x0758
2OVER	( x1 x2 x3 x4 - x1 x2 x1 x2 x3 x4 x1 x2 )	Copy cell pair x1 x2 to the <a href="#">TOS</a>	0x0750 0x0460 0x0789 0x0460

...continued

Table 4-1: ANS Forth words

Word	Stack	Description	Opcode
2>R	( x1 x2 - ) (R: - x1 x2 )	Shift cell pair x1 x2 to the <a href="#">return stack</a>	0x06AB 0x06AB
2R>	( - x1 x2 ) (R: x1 x2 - )	Shift cell pair x1 x2 to the <a href="#">parameter stack</a>	0x0755 0x0755
2R@	( - x1 x2 ) (R: x1 x2 - x1 x2 )	Copy cell pair x1 x2 to the <a href="#">parameter stack</a>	0x0755 0x0757
2ROT	( 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	0x0DA0
<>	( 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	0x0DE0
>R	( x - ) (R: - x )	Shift x on to the <a href="#">return stack</a>	0x06AB
?DUP	( x - 0 x x )	Duplicate x if it is not zero	0x0750 0x0D30 0x2001 0x06A8
@	( addr - x )	Fetch x from addr	0x03FF
ABS	( n - u )	Absolute value 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 <a href="#">parameter stack</a> without +n	0x0100
DROP	( x - )	Drop x from the /glsp	0x06A8
DUP	( x - x x )	Duplicate x	0x0750
EKEY	( - x )	Receive one input event u <sup>1</sup>	0x0107
EKEY?	( - flag )	Return true if input event is available <sup>4</sup>	0x0104
EMIT	( x - )	Send x to the output device <sup>4</sup>	0x0005
EMIT?	( - flag )	Return true if the output device is ready for data <sup>4</sup>	0x0105
EXECUTE	( i*x xt - j*x )	Execute xt	0x7FFF
FALSE	( - false )	FALSE flag	0x1000

...continued

<sup>1</sup>Requires the [EKEY/EMIT extension](#)



Table 4-1: ANS Forth words

Word	Stack	Description	Opcode
I	( - n u ) ( R: n u - n u )	Copy the innermost loop index n u onto the <a href="#">parameter stack</a>	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 <a href="#">parameter stack</a>	0x0755 0x0407
LSHIFT	( x1 u - x2 )	Shift x1 u bits towards the <a href="#">MSB</a>	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	0x0CA0
MIN	( n1 n2 - n3 )	n3 is the lesser of n1 and n2	0x0CE0
NEGATE	( n1 - n2 )	n2 is the two's complement of n1	0x0C70
NIP	( x1 x2 - x2 )	Drop x1	0x06A0
OR	( x1 x2 - x3 )	Bitwise logic OR of x1 and x2	0x0EC0
OVER	( x1 x2 - x1 x2 x1 )	Copy x1 to the <a href="#">TOS</a>	0x0758
R>	( - x ) (R: x - )	Shift x to the <a href="#">parameter stack</a>	0x0755
R@	( - x ) (R: x - x )	Copy x to the <a href="#">parameter stack</a>	0x0754
RSHIFT	( x1 u - x2 )	Shift x1 u bits towards the <a href="#">LSB</a>	0x0F00
ROT	( x1 x2 x3 - x2 x3 x1 )	Rotate the three topmost cells	0x0460 0x0418 or 0x041C <sup>2</sup>
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 or 0x07C0 <sup>2</sup>
U<	( u1 u2 - flag )	Test if u1 is lower than u2	0x0DC0
U>	( u1 u2 - flag )	Test if u1 is greater than u2	0x0D80
UM*	( u1 u2 - d )	Multiply u1 by u2	0x0A00
XOR	( x1 x2 - x3 )	Bitwise logic XOR of x1 and x2	0x0EA0

## 4.2 Non-Standard Forth Words

To provide access to features which are unique to the N1 processor, the word definitions listed in [Table 4-2](#) are recommended.

<sup>2</sup>Requires the [ROT extension](#)

Table 4-2: Non-standard Forth words

Word	Stack	Description	Opcode
CLRPS	( i*x- )	Clears the <a href="#">parameter stack</a>	0x1000 0x0000
CLRRS	( i*x- )	Clears the <a href="#">return stack</a>	0x1000 0x0001
IDIS	( - )	Disable interrupts <sup>3</sup>	0x1000 0x0003
IEN	( - )	Enable interrupts <sup>3</sup>	0x1FFF 0x0003
IEN?	( - flag )	Return true if interrupts are enabled <sup>3</sup>	0x0103
PEEK	( - x )	Read from the input device, while preserving the input event <sup>4</sup>	0x0106
RDEPTH	( - +n )	+n is the number of cells on the <a href="#">return stack</a>	0x0101

<sup>3</sup>Requires the [Interrupt extension](#)<sup>4</sup>Requires the [EKEY/EMIT extension](#)

## 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 segments: the [upper stack](#), the [intermediate stack](#), and the [lower stack](#).

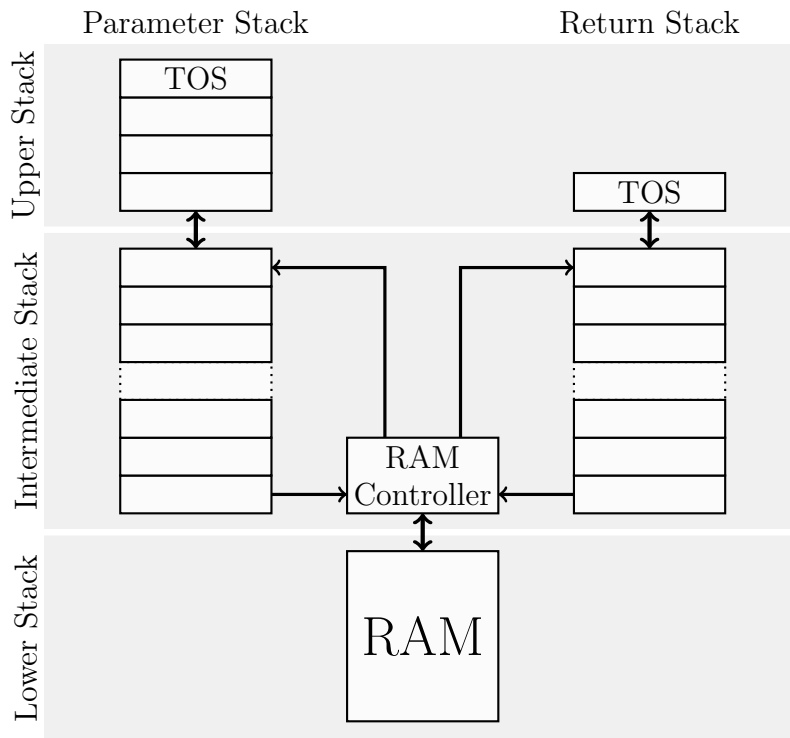


Figure 5-1: Stack Architecture

### 5.1 Parameter Stack

The [upper parameter stack](#) holds the four topmost data entries. Its purpose is to perform stack and [ALU](#) operations (see [Section 2.7 “Stack Instructions”](#) and [Section 2.6 “ALU Instructions”](#)). When the capacity of the [upper stack](#) is exceeded, older data entries are shifted 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 operations 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 [intermediate stack](#)’s 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 by 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 [return 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 [parameter stack](#) and the [return stack](#) become cleared and the [Throw Code register \(TC\)](#) is set to 0x0000 (see [Table 6-1](#)). After every 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

Exceptions are a mechanism to handle error conditions. They are thrown either by hardware or software. There are five error conditions, which are detected by the N1 hardware:

#### [Parameter stack overflow](#)

A [parameter stack](#) overflow occurs when the capacity of the lower stack's RAM is exceeded

#### [Return stack underflow](#)

A [parameter stack](#) underflow occurs when an instruction requires more arguments than available on the [stack](#) and when a stack instruction would result 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

#### [Return stack underflow](#)

A [return stack](#) underflow occurs when an instruction requires more arguments than available on the [return stack](#).

#### [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

Any other error condition must be detected by software and thrown by writing to the [Throw Code register \(TC\)](#). Whenever an exception is thrown, the N1 processor keeps the associated [throw code](#) (see [Table 6-1](#) in the [Throw Code register \(TC\)](#)). In case of a [parameter stack](#) overflow or a [return stack](#) overflow, both stacks will be cleared. Otherwise, the [return stack](#) and the lower content of the [parameter stack](#) will be kept, allowing the [exception](#) to be caught by software. The N1 processor will then continue with a call to address 0x0000.

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 (no exception)
0x0001 (1)	No exception
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

### 6.3 Interrupts

interrupts are an optional feature (see [Section 3.2 “Interrupt Extension”](#))

## 7 Integration Guide

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

### 7.1 Integratation Parameters

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

#### SP.WIDTH

Stack pointer width.

This parameter determines the address width of the [lower stack](#). Values in the range of 5 to 16 are valid. The default value is 12.

#### IPS.DEPTH

Depth of the intermediate [parameter stack](#).

This parameter determines the number of [cells](#) in the [intermediate stack](#) of the [parameter stack](#). Any value larger than 2 is valid. The default value is 8. The purpose of the [intermediate stack](#) is to conceal fluctuations in stack usage to the [lower stack](#). The optimal value should be derived from the application use case.

#### IPS.BYPASS

Bypass the intermediate [parameter stack](#).

This parameter provides the option of bypassing the intermediate [parameter stack](#). If set to a non-zero value, the [lower stack](#) will be directly connected to the [upper stack](#). The default value is 0.

#### IRS.DEPTH

Depth of the intermediate [return stack](#).

This parameter determines the number of [cells](#) in the [intermediate stack](#) of the [return stack](#). Any value larger than 2 is valid. The default value is 8. The purpose of the [intermediate stack](#) is to conceal fluctuations in stack usage to the [lower stack](#). The optimal value should be derived from the application use case.

#### IRS.BYPASS

Bypass the intermediate [return stack](#).

This parameter provides the option of bypassing the intermediate [return stack](#). If set to a non-zero value, the [lower stack](#) will be directly connected to the [upper stack](#). The default value is 0.

#### PBUS.AADR.OFFSET

Offset for direct [jump](#) or [call](#) addressing.

This parameter determines the location of the 32KB window for [jumps](#) and [calls](#) with [direct addressing](#). The default value is 0x0000.

#### PBUS.MADR.OFFSET

Offset for direct data accesses.

This parameter determines the location of the 511B window for memory I/O with [direct addressing](#). This window should cover commonly used [Forth](#) variables. The default value is 0xFFFF.

**EXT\_ROT**

Enable the [ROT extension](#).

Recovering from an exception requires some free [stack](#) space. This parameter enables the [ROT extension](#) if set to a non-zero value. It is disabled by default.

**7.2 Interfaces**

The N1 processor provides four interfaces which must be connected at system level. A fifth one (see [Section 7.2.4 “Probe Signals”](#)) is only to be used for verification and debug purposes.

**7.2.1 Clock and Resets**

This interface provides clocks and resets for all sequential logic in the N1 design.

**clk\_i**

Single clock input.

This clock is used for all interfaces as well as all internal sequential logic.

**async\_rst\_i**

Asynchronous reset input.

This active high reset input may assert asynchronously, but must deassert synchronously. This signal is not required if a synchronous reset (**sync\_rst\_i**) is implemented. If unused, this input must be tied to 0.

**sync\_rst\_i**

Synchronous reset input.

This active high reset input must assert and deassert synchronously. This signal is not required if an asynchronous reset (**async\_rst\_i**) is implemented. If unused, this input must be tied to 0.

**7.2.2 Program Bus**

This interface connects the N1 to the main memory. All signals comply to the [Wishbone](#) protocol [2].

**pbus\_cyc\_o**

Cycle indicator output.

This output signal corresponds to signal **CYC\_0** of the Wishbone specification [2].

**pbus\_stb\_o**

Strobe output.

This output signal corresponds to signal **STB\_0** of the Wishbone specification [2].

**pbus\_we\_o**

Write enable output.

This output signal corresponds to signal **WE\_0** of the Wishbone specification [2].

**pbus\_adr\_o**

Address bus.

These output signals correspond to bus **ADR\_0** of the Wishbone specification [2].

**pbus\_dat\_o**

Write data bus.

These output signals correspond to bus **DAT\_0** of the Wishbone specification [2].



**pbus\_tga\_cof\_jmp\_o**

Change of flow indicator.

This output signal corresponds to bus **TGA\_O** of the Wishbone specification [2].

It indicates, that the current bus access was caused by a **jump** instruction. This information may be used to trace the program flow.

**pbus\_tga\_cof\_cal\_o**

Change of flow indicator.

This output signal corresponds to bus **TGA\_O** of the Wishbone specification [2].

It indicates, that the current bus access was caused by either a **call** instruction or an interrupt service request. This information may be used to trace the program flow.

**pbus\_tga\_cof\_bra\_o**

Change of flow indicator.

This output signal corresponds to bus **TGA\_O** of the Wishbone specification [2].

It indicates, that the current bus access was caused by a **conditional branch** instruction. This information may be used to trace the program flow.

**pbus\_tga\_cof\_eow\_o**

Change of flow indicator.

This output signal corresponds to bus **TGA\_O** of the Wishbone specification [2].

It indicates, that the current bus access was caused by a return from a **call**. This information may be used to trace the program flow.

**pbus\_ack\_i**

Acknowledge input.

This input signal corresponds to signal **ACK\_I** of the Wishbone specification [2].

If unused, this input must be tied to 1.

**pbus\_err\_i**

Error indicator input.

This input signal corresponds to signal **ERR\_I** of the Wishbone specification [2].

It informs the N1 processor, that the current address exceeds the valid range of the connected memory system. If unused, this input must be tied to 0.

**pbus\_rty\_i**

Retry input.

This input signal corresponds to signal **RTY\_I** of the Wishbone specification [2].

It terminates the bus cycle, indicating that the target is not ready to accept or provide new data. In contrast to asserting **pbus\_stall\_i** or delaying **pbus\_ack\_i**, the bus cycle termination via **pbus\_rty\_i** does not block the processor from handling interrupts. If unused, this input must be tied to 0.

**pbus\_stall\_i**

Pipeline stall input.

This input signal corresponds to signal **STALL\_I** of the Wishbone specification [2]. If unused, this input must be tied to 0.

**pbus\_dat\_i**

Read data bus.

These input signals correspond to bus **DAT\_I** of the Wishbone specification [2].

### 7.2.3 Interrupt Interface

This interface connects an optional interrupt controller to the N1 processor.

#### `irq_ack_o`

Interrupt acknowledge.

This output signal asserts for one clock cycle, whenever the current interrupt is serviced. It may be used for automatic flag clearing.

#### `irq_req_i`

Interrupt request.

Any non-zero value driven to this bus interface is interpreted as interrupt request. The value determines the start address of the interrupt service routine that is to be executed by the N1 processor. This bus must be tied to 0x0000 if no interrupt controller is connected.

### 7.2.4 Probe Signals

This interface propagates all internal states of the N1 processor to the outside. It is solely intended for verification and debug purposes and should be left unconnected for system integration. The signals in this interface are specific to the internal implementation of the N1 processor and may change with every revision.

## 7.3 Target Specific Design Files

All adder and multiplier logic of the N1 design is located in a single [Verilog](#) module called `N1_dsp`. A synthesizable implementation of this module, can be found in the file `rtl/verilog/N1_dsp.synth.v`. If desired, this file can be replaced by one containing a alternative implementation of the `N1_dsp` module. An example is given in in the file `rtl/verilog/N1_dsp.iCE40UP5K.v`. It contains a custom implementation for Lattice iCE40 FPGAs, utilizing four hard instantiated `SB_MAC16` macro cells.

## 8 Architecture Description

The following sections provide some descriptions of the internal N1 design.

### 8.1 Design Principles

The RTL implementation of the N1 follows a number of design principles which are captured in following sections.

#### 8.1.1 Naming Convention of Interface Signals

For all signals, which do not implement a common standard (e.g. [Wishbone](#)), the following signal naming rules are used throughout the design:

- Signals which are grouped into an interface are prefixed with a meaningful interface name.  
Example: ups\_push
- All other point-to-point connections, contain a mnemonic of the sending and the receiving block in its prefix. The format of the prefix is:  
*<sender mnemonic>2<receiver mnemonic>...*  
Example: fc2ir\_capture
- Control signals which represent a request, end with a verb in imperative form.  
Example: fc2ir\_expend
- Status signals representing a busy indicator, have the postfix *...\_bsy*  
Example: prs2fc\_bsy
- If a signal is connected to the interface of a module, a further postfix is added to indicate the signal direction:
  - Input signals: *...\_i*
  - Output signals: *...\_o*  
Example: prs2fc\_bsy\_o
- Names of signals which are only used within one design block are kept short and don't follow a particular naming convention.

#### 8.1.2 Handshaking

A high signal level of a control signal is interpreted as a request by the receiving design block. The request is expected to be immediately accepted by the receiver and processed in the next clock cycle, unless the receiver provides a busy indicator (*...\_bsy*). In this case the request is only accepted if the busy indicator was deasserted in the cycle, in which the request is made.

### 8.2 Common Internal Interfaces

The subblocks in the N1 design use common interfaces for common functionality. These interfaces follow the [naming conventions](#) and [handshaking concept](#) described in [Section 8.1 “Design Principles”](#).

### 8.2.1 Stack Interface

All stacks are controlled using the following interface:

**<stack name>\_clear\_o/\_i** (controller → stack)

Request to clear the stack.

**<stack name>\_clear\_bsy\_i/\_o** (controller ← stack)

Busy indicator.

The stack will be cleared if <stack name>\_clear\_i is asserted while

<stack name>\_clear\_bsy\_o is deasserted.

**<stack name>\_push\_o/\_i** (controller → stack)

Request to push a data word onto the stack.

**<stack name>\_push\_data\_o/\_i[15:0]** (controller → stack)

Data word to be pushed onto the stack.

The data word must be supplied in the same clock cycle as the request.

**<stack name>\_push\_bsy\_i/\_o** (controller ← stack)

Busy indicator.

<stack name>\_push\_data\_i will be pushed onto the stack if <stack name>\_push\_i is asserted while <stack name>\_push\_bsy\_o and <stack name>\_full\_o are deasserted.

**<stack name>\_full\_i/\_o** (controller ← stack)

Overflow indicator.

<stack name>\_full\_o is asserted when the stack is full and a new push request would cause an overflow.

**<stack name>\_pull\_o/\_i** (controller → stack)

Request to pull a data word from the stack.

**<stack name>\_pull\_data\_i/\_o[15:0]** (controller ← stack)

Data word to be pulled from the stack. If the stack is not empty

(<stack name>\_empty\_o deasserted) and ready for a pull operation

(<stack name>\_pull\_bsy\_o deasserted), then <stack name>\_pull\_data\_o always shows the data at the top of the stack.

**<stack name>\_pull\_bsy\_i/\_o** (controller ← stack)

Busy indicator.

The data at the top of the stack will be removed if <stack name>\_pull\_i is asserted while <stack name>\_pull\_bsy\_o and <stack name>\_empty\_o are deasserted.

**<stack name>\_empty\_i/\_o** (controller ← stack)

Underflow indicator.

<stack name>\_empty\_o is asserted when the stack is empty and a new pull request would cause an underflow.

The stack interface is also used for FIFOs.

### 8.2.2 Memory Interface

Memories are connected through the following interface:

**<memory name>\_addr\_o/\_i[n-1:0]** (controller → memory)

Memory address.

**<memory name>\_access\_o/\_i** (controller → memory)

Access request.

**<memory name>\_rwb\_o/\_i** (controller → memory)

Data direction selector (high for read, low for write).

**<memory name>\_access\_bsy\_i/\_o** (controller ← memory)

Busy indicator.

A request is valid if **<memory name>\_access\_i** is asserted while

**<memory name>\_access\_bsy\_o** is deasserted.

**<memory name>\_wdata\_o/\_i[15:0]** (controller → memory)

Write data.

Write data must be driven in the same clock cycle as the request.

**<memory name>\_rdata\_i/\_o[15:0]** (controller ← memory)

Read data.

Read data must be captured one clock cycle after a valid request has been captured, unless a delay is indicated.

**<memory name>\_rdata\_del\_o/\_i[15:0]** (controller → memory)

Read data delay indicator.

If asserted, this signal will postpone expected read data by one clock cycle..

### 8.2.3 Register Interface

Registers are accessed through the following interface:

**<register block name>\_addr\_o/\_i[n-1:0]** (controller → register block)

Register address.

**<register block name>\_set\_o/\_i** (controller → register block)

Register write request.

Register address and write data must be supplied in the same clock cycle as the write request.

**<register block name>\_set\_data\_o/\_i[n-1:0]** (controller → register block)

Register write data.

**<register block name>\_set\_bsy\_i/\_o** (controller ← register block)

Register write busy indicator.

**<register block name>\_get\_o/\_i** (controller → register block)

Register read request.

The register address must be supplied in the same clock cycle as the write request. Unless the busy indicator is asserted, read data is available in the same clock cycle as the request.

**<register block name>\_get\_data\_o/\_i[n-1:0]** (controller → register block)

Register read data.

`<register block name>_get_bsy_i/_o` (controller  $\leftarrow$  register block)  
Register read busy indicator.

#### 8.2.4 Instruction Boundaries

The [instruction register](#) always contains the instruction which is currently in execution. Before the execution of an instruction can be concluded and the next one can begin, the following conditions must be fulfilled:

- The program bus must be available - TBD
- The parameter and the return stack must be available - TBD

### 8.3 Instruction Execution Cycle

The execution cycle of the N1 processor characterized by the following design components:

#### Program Counter

A 16-bit register, which contains the memory location of the next instruction to be executed. It is implemented within the [DSP Block](#).

#### Address Bus

The address output of the [Program Bus](#) (`pbus_adr_o`).

#### Read Data Bus

The read data input of the [Program Bus](#) (`pbus_dat_i`).

#### Instruction Register

A 16-bit register holding the opcode of the instruction, which is currently executed (see [Section 8.4.2 “Instruction Register \(ir\)”](#)).

#### Instruction Stash Register

A 16-bit register to temporarily store an upcoming opcode. (see [Section 8.4.2 “Instruction Register \(ir\)”](#)).

The following sections show the timing relation of these design components in different execution scenarios.

#### 8.3.1 Plain Linear Execution

Most of the N1 instructions are executed in a single clock cycle. [Figure 8-1](#) shows the typical linear execution flow of single cycle instructions.

The opcode stored in the instruction register determines which instruction is currently being executed. The program counter points to the address of the next instruction. The address bus is unregistered and always runs one clock cycle ahead of the program counter. The resulting data on the read data bus is captured by the instruction register in the next clock cycle.

#### 8.3.2 Execution of Extended Instructions

In some cases the execution of an instruction can span multiple cycles (i.e. non-concurrent control instructions or any instruction waiting for a blocked stack access). [Figure 8-2](#) illustrates the timing in these scenarios.

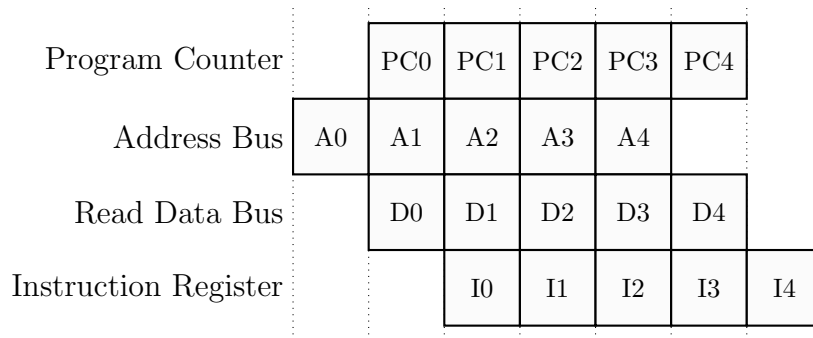


Figure 8-1: Plain Linear Execution

Whenever an opcode needs to be captured from the read data bus, but the instruction register is blocked by an instruction spanning multiple cycles, the incoming opcode needs to be temporarily stashed away in a separate register. When the execution of the ongoing instruction is finished, the stashed opcode is moved into the instruction register.

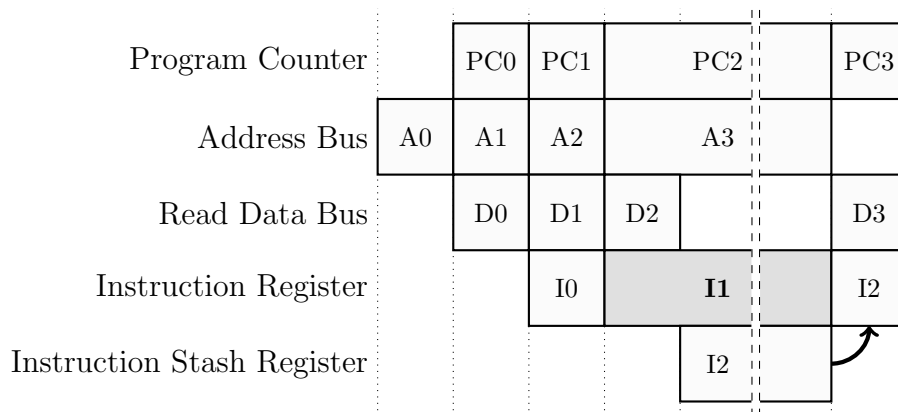


Figure 8-2: Execution of an Extended Instruction

### 8.3.3 Execution of Memory Access Instructions

A special case of multi-cycle instructions are [memory access instructions](#). These instructions perform their memory accesses on the program bus. [Figure 8-3](#) illustrates how opcode fetches and data accesses are interleaved.

### 8.3.4 Change of Flow Instructions

TBD

### 8.3.5 Exceptions and Interrupts

TBD

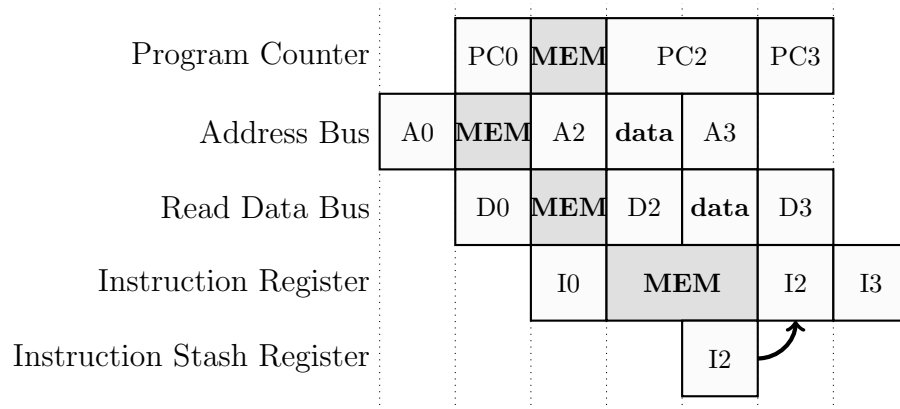


Figure 8-3: Execution of a Memory Access Instruction

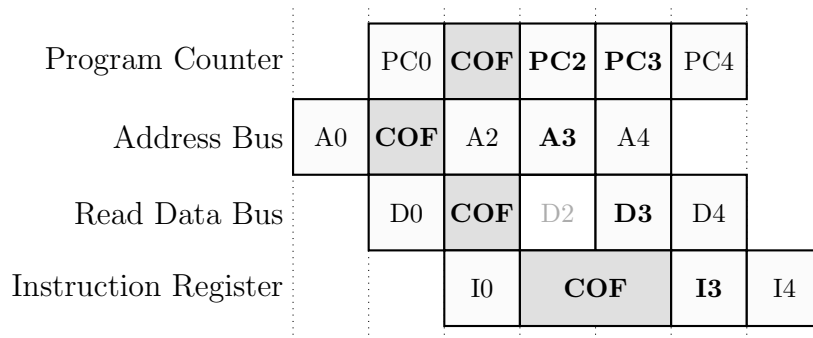


Figure 8-4: Execution of a Change of Flow Instruction

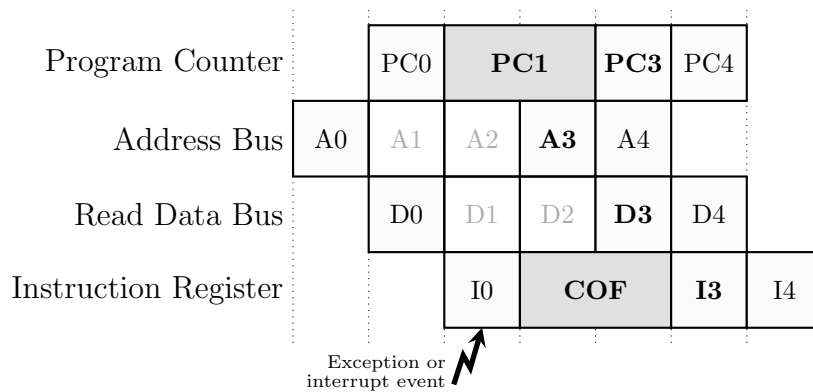


Figure 8-5: Program flow interrupted by an exception - TBD

## 8.4 Design Components

The N1 architecture is divided in 11 subblocks as shown in [Figure 8-6](#).



TBD

Figure 8-6: Block Diagram

#### 8.4.1 Flow Control Block (fc)

The flow control block is implemented in the Verilog module `N1_fc` (`N1_fc.v`). It manages the [instruction cycles](#) of the N1 core. It handles the control and response signals of the [program bus's Wishbone](#) interface and it communicates with the other N1 components by sending requests and receiving status information. No actual data passes through the `N1_fc` module. The interfaces to the N1 components, which are under the control of the flow control block, are explained in the following sections.

##### 8.4.1.1 Control and Status Interface to the [Instruction Register](#)

The flow control block is able to request has the following request signals to the instruction register:

###### `fc2ir_capture`

Capture the [program bus's](#) read data (`pbus_dat_i`) in the instruction register at the next clock edge.

###### `fc2ir_stash`

Capture the [program bus's](#) read data (`pbus_dat_i`) in the stash register at the next clock edge.

###### `fc2ir_expend`

The read data input of the [Program Bus](#) (`pbus_dat_i`)

###### `fc2ir_expend`

Copy the stash register's content into the instruction register at the next clock cycle.

The following status signals are coming from the instruction register:

#### 8.4.2 Instruction Register (ir)

#### 8.4.3 Arithmetic Logic Unit (alu)

TBD

#### 8.4.4 DSP Block (dsp)

TBD

#### 8.4.5 Exception Handler (excpt)

TBD

#### 8.4.6 Upper Stack (us)

TBD

**8.4.7 Intermediate Parameter Stack (ips)**

TBD

**8.4.8 Intermediate Return Stack (irs)**

TBD

**8.4.9 Lower Stack (ls)**

TBD

## 9 Verification Status

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

## 10 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 10-1](#) summarizes the tools, used for this project.

Table 10-1: Tool Summary

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

## 11 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 2.3 “Call Instructions”](#)).

### CATCH extension

An optional extension to support CATCH functionality, described in [Section ?? “??”](#)

### 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 2.4 “Conditional Branches”](#)).

### direct addressing

Addressmode, where the address is encoded into the [opcode](#) of an instruction

### EKEY/EMIT extension

An optional extension to support EKEY and EMIT functionality, described in [Section 3.4 “KEY/EMIT Extension”](#)

### exception

An error condition that will disrupt the program flow.

### Forth

Forth is an extensible stack-based programming language.

### function register

A processor internal register, that provides access to a hardware feature.

### immediate data

A data value, which is encoded into the [opcode](#) of an instruction

### indirect addressing

Address mode, where the address is stored on the [parameter stack](#).

### intermediate stack

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

**Interrupt extension**

An optional extension to support interrupts, described in [Section 3.2 “Interrupt Extension”](#)

**IST**

A bit field in the stack instruction which controls data movement on the intermediate [parameter stack](#) or [return stack](#). The mnemonic stands for “Intermediate Stack Transition”.

**jump**

A change of the program flow without return option (see [Section 2.2 “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 2.5 “Literals”](#)).

**lower stack**

The section of the stack which is 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.

**relative addressing**

Addressmode, where the address is given relative to the current position in the execution flow

**return stack**

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

**ROT extension**

An optional extension of the N1 instruction set, described in [Section 3.1 “ROT Extension”](#)

**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 that 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”.

**Verilog**

The hardware description language used for the N1 implementation.

**Von-Neumann-Architecture**

A computer architecture where instruction fetches and data I/O occur over the same memory interface.

**Wishbone**

An open bus protocol. see [\[2\]](#)

**word**

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

## 12 References

- [1] American National Standard for Information Systems, 1994.
- [2] Wishbone b4. [http://cdn.opencores.org/downloads/wbspec\\_b4.pdf](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>.