NOP – A Simple Experimental Processor for Parallel Deployment

Oskar Schirmer
Göttingen, 2016-12-20

Abstract

The design of a parallel computing system using several thousands or even up to a million processors asks for processing units that are simple and thus small in space, to make as many processing units as possible fit on a single die.

The design presented herewith is far from being optimised, it is not meant to compete with industry performance devices. Its main purpose is to allow for a prototypical implementation of a dynamic software system as a proof of concept.
## Contents

1 Overview 4

2 Registers and Execution Model 6

3 Communication Switch 7

4 Instruction Opcodes 8
   4.1 Immediate Constant 8
   4.2 No Operation 8
   4.3 Add 9
   4.4 Subtract 9
   4.5 Multiply 9
   4.6 Unsigned Divide 9
   4.7 Signed Divide 10
   4.8 Bitwise And 10
   4.9 Bitwise Or 10
   4.10 Bitwise Exclusive Or 10
   4.11 Pop 11
   4.12 Duplicate 11
   4.13 Exchange 11
   4.14 Load From Stack 11
   4.15 Swap Bit Fields 12
   4.16 Load, Decrement, Push, Store 12
   4.17 Logarithm 12
   4.18 Shift And Rotate Left 13
   4.19 Shift And Rotate Right 13
   4.20 Sign Extend 14
   4.21 Zero 14
   4.22 Unconditional Jump 14
   4.23 False Jump 15
   4.24 Load Constant 15
   4.25 Load Word 15
   4.26 Store Word 15
   4.27 Count Bits 16
   4.28 Stop 16
   4.29 Break 16
   4.30 Start New Thread 16
   4.31 Call Function 17
   4.32 Return From Function 17
4.33 Store To Stack ........................................... 17
4.34 Load, Push, Increment, Store ................................ 18
4.35 Get Port Destination ...................................... 18
4.36 Set Port Destination ...................................... 18
4.37 Output Word ............................................. 18
4.38 Output End Token ........................................ 19
4.39 Output Pause Token ..................................... 19
4.40 Input Word ............................................... 19
4.41 Check Input Port ......................................... 20
4.42 Clear Event List .......................................... 20
4.43 Set Output Event ......................................... 20
4.44 Set Input Event .......................................... 20
4.45 Set Input End Event ..................................... 21
4.46 Wait ...................................................... 21
4.47 Current Time ............................................ 21
4.48 Wait With Timeout ...................................... 22
4.49 Increment Stack Pointer ................................ 22
4.50 Compare Unsigned ....................................... 22
4.51 Compare Signed .......................................... 23
4.52 Combined Number ....................................... 23
4.53 Calculate Global Port Number ............................. 23
4.54 Calculate Stack Pointer .................................. 24
4.55 Available Threads ....................................... 24
4.56 Cycles Per Threads ...................................... 24
4.57 Total Cycles .............................................. 24

5 Implementation ............................................. 25

6 Discussion .................................................. 26
1 Overview

The Null Operand Parallel processor is designed to build a parallel computing system out of large numbers of such instances. Its main purpose is to allow for a prototypical implementation of a dynamic software system as a proof of concept.

While such processors have been designed, existing approaches either allow for substantial simplification ([2009dm]), or do lack flexible communicating means or sufficient resources ([2011ga]) needed when it comes to implementing a dynamic software system, i.e. an operating system with user interaction and dynamic process creation.

The Null Operand Parallel processor is composed of a number of processing units, each with its own local fast memory, and a communication switch to allow exchanging messages between the processing units, and between processors that are connected via an external link (see figure 1).

---

Each processing unit provides multiple register sets to implement a number of independant hardware threads. Those threads that are active are scheduled in a simple round robin manner. Threads may block – waiting for data availability – and stop – either controlled or upon fault.

All addressing for a single thread is done relative to base pointers, one for code and constant data, and one for variable data. This way all operations are base register relative, and so any thread is fully relocatable even at runtime.

The single thread is designed to perform sequential instructions one at a time. Instructions are encoded in eight bit each, with no operands encoded. Instead, operations are performed on a single thread local stack\textsuperscript{1}.

\textsuperscript{1}which is a well known concept, see e.g. [1963bc], see also [2007ae]
All addressing is done wordwise, i.e. there are no byte addressable items at all, and consequently, there are no alignment issues. The only register to address subunits of words is the instruction pointer, as there are always four instructions in a word.

For each thread, the processing unit implements a number of data channel ports that are directly connected to the communication switch. The communication switch provides external links, so it is possible to connect processors and thus build a large computation network\(^2\).

Transmission of data to a remote port on a different processor may require using the switches of one or more intermediate processors, whenever there is no direct connection between the originating and the target processor. To handle arbitrary – and especially transitional – channels, the communication switch implements a generic, table driven routing algorithm.

For peripheral data transmission, the communication switch provides a set of bidirectional interfaces to transfer data to and from peripheral units without no specific transmission protocol.

Numbers for the current test implementation:

|                                |     |
|--------------------------------|-----|
| processing units per processor | 4   |
| threads per processing units   | 8   |
| word size in bits              | 32  |
| words per local memory         | 16384|
| channel ports per thread       | 32  |
| external link channels         | 4   |
| peripheral transmission interfaces | 8   |

The Null Operand Parallel Processor does not provide:

– shared memory
– cache memory
– interrupts
– program flow exceptions
– virtual memory addressing
– privileged modes
– dedicated specialised peripheral units

\(^2\)Note, that neither the switch concept is new – see e.g. [2009dm] – nor is the external link concept – see e.g. [2008es]
2 Registers and Execution Model

Execution per thread is based on six registers, all 14 bit wide, with the exception of the instruction pointer:

| Register | Description | Details |
|----------|-------------|---------|
| ip       | instruction pointer | 16 bit wide, the lowest 2 bits referencing the opcode within a word, least significant opcode first |
| sp       | stack pointer   | growing downwards |
| ld<sub>n</sub> | data limits | lower and upper bound for memory data access |
| lc<sub>n</sub> | code limits | lower and upper bound for memory code and constants access |

Another two registers are derived from the limits:

| Register | Description | Details |
|----------|-------------|---------|
| cp       | constants pointer | equals lc<sub>0</sub> + 64 |
| dp       | data pointer | equals ld<sub>0</sub> + 64 |

Whenever the instruction pointer is out of range of the code limits, or the stack pointer is out of range of the data limits, the thread is stopped as faulty (see figure 2).

At start time of a thread, its stack pointer is initialised to ld<sub>1</sub>. When a thread is stopped – be it faulty or not – an exception message is sent to a destination port, stored into the exc register at thread start time.
3 Communication Switch

To send a message through a channel, first the destination channel port number has to be assigned to the local channel port. Any subsequent message send through this channel is send to that destination, until a new destination is assigned. A channel global port number is 32 bit wide:

| processor id resp. routing command | unit# | thread# | port# |
|-----------------------------------|-------|---------|-------|
| bit 31 .. 10                      | bit 9 .. 8 | bit 7 .. 5 | bit 4 .. 0 |

For special values of the upper 22 bits of the global port number, special local routing is performed:

| routing command value | routing action                                      |
|-----------------------|-----------------------------------------------------|
| 0                     | connect to local unit                               |
| 1                     | connect to peripheral line                          |
| 2                     | connect to router configuration block               |
| 4 .. 7                | connect to external link 0 .. 3                     |
| processor id ≥ 8      | connect according to routing table                  |

A channel transmission path in use is blocked from its local to the destination end as long as the message is sent. To end a message, the sender must send a terminatory END token. When sending a message is paused and shall be continued later, the sender is free to send in between a PAUSE token, which – in contrast to the END token – is not delivered to the receiving destination. By sending a PAUSE token, the channel transmission path is freed until sending the message is continued.
4 Instruction Opcodes

All instructions are encoded as eight bit opcodes, and do not encode operands explicitly but work on stack data.

Boolean values are words, 0 is false, all other values are true. Instructions, that generate a boolean value, will always push -1 for true.

In the following detailed description of the single instructions, m denotes the local memory. Other single character variables denote temporary values.

--sp is the stack pointer predecremented before use, sp++ is the stack pointer postincremented after use.

The port function calculates a global port number from the processors id, the processing units number, the given thread number, and the local channel port number:

\[
\text{port}(t, p)_{31..10} \leftarrow \text{id}_{\text{processor}} \\
\text{port}(t, p)_{9..8} \leftarrow \text{number}_{\text{unit}} \\
\text{port}(t, p)_{7..5} \leftarrow t \\
\text{port}(t, p)_{4..0} \leftarrow p
\]

4.1 Immediate Constant

| Opcodes: 0x00..0x7F, 0xC0..0xFF |

All opcodes, except those in the range from 0x80 to 0xBF, do load their own constant value – sign extended – onto the stack:

\[
a_{31..8} \leftarrow \text{opcode}_7 \\
a_{7..0} \leftarrow \text{opcode} \\
m(--\text{sp}) \leftarrow a \\
\text{ip} \leftarrow \text{ip} + 1
\]

4.2 No Operation

| Name: NOP | Opcode: 0x80 |

No operation:

\[
\text{ip} \leftarrow \text{ip} + 1
\]

4.3 Add

| Name: ADD | Opcode: 0x81 |
Pop two words, push the sum:
\[
a \leftarrow m(sp++) \\
b \leftarrow m(sp++) \\
m(--sp) \leftarrow b + a \\
ip \leftarrow ip + 1
\]

4.4 Subtract

| Name: SUB | Opcode: 0x82 |
|----------|--------------|

Pop a subtrahend, pop a minuend, subtract the subtrahend from the minuend, push difference:
\[
a \leftarrow m(sp++) \\
b \leftarrow m(sp++) \\
m(--sp) \leftarrow b - a \\
ip \leftarrow ip + 1
\]

4.5 Multiply

| Name: MUL | Opcode: 0x83 |
|----------|--------------|

Pop two words, push the product:
\[
a \leftarrow m(sp++) \\
b \leftarrow m(sp++) \\
m(--sp) \leftarrow b \times a \\
ip \leftarrow ip + 1
\]

4.6 Unsigned Divide

| Name: UDIV | Opcode: 0x84 |
|-----------|--------------|

Pop a divisor, pop a dividend, unsigned divide dividend by divisor, push quotient and remainder. On division by zero stop thread:
\[
a \leftarrow m(sp++) \\
if a = 0 then fault \\
b \leftarrow m(sp++) \\
q \leftarrow b / a \\
m(--sp) \leftarrow q \\
m(--sp) \leftarrow b - q \times a
\]

9
\text{ip} \leftarrow \text{ip} + 1

4.7 Signed Divide

| Name: SDIV               | Opcode: 0x85 |
|-------------------------|-------------|

Pop a divisor, pop a dividend, signed divide dividend by divisor according to the Euclidean division (see e.g. [2001dl]), push quotient and remainder. On division by zero stop thread:
\begin{align*}
a & \leftarrow \text{two's complement } \mathsf{m} \mathsf{(sp++)} \\
\text{if } a = 0 \text{ then fault} \\
b & \leftarrow \text{two's complement } \mathsf{m} \mathsf{(sp++)} \\
q & \leftarrow b / a \\
\mathsf{m} \mathsf{(--sp)} & \leftarrow q \\
\mathsf{m} \mathsf{(--sp)} & \leftarrow b - q \times a \\
\text{ip} & \leftarrow \text{ip} + 1
\end{align*}

4.8 Bitwise And

| Name: AND               | Opcode: 0x86 |
|-------------------------|-------------|

Pop two words, push the conjunction:
\begin{align*}
a & \leftarrow \mathsf{m} \mathsf{(sp++)} \\
b & \leftarrow \mathsf{m} \mathsf{(sp++)} \\
\mathsf{m} \mathsf{(--sp)} & \leftarrow b \text{ and } a \\
\text{ip} & \leftarrow \text{ip} + 1
\end{align*}

4.9 Bitwise Or

| Name: OR                | Opcode: 0x87 |
|-------------------------|-------------|

Pop two words, push the disjunction:
\begin{align*}
a & \leftarrow \mathsf{m} \mathsf{(sp++)} \\
b & \leftarrow \mathsf{m} \mathsf{(sp++)} \\
\mathsf{m} \mathsf{(--sp)} & \leftarrow b \text{ or } a \\
\text{ip} & \leftarrow \text{ip} + 1
\end{align*}

4.10 Bitwise Exclusive Or
Name: XOR  
Opcode: 0x88

pop two words, push the exclusion:
\[ a \leftarrow m(sp++) \]
\[ b \leftarrow m(sp++) \]
\[ m(--sp) \leftarrow b \text{ xor } a \]
\[ ip \leftarrow ip + 1 \]

4.11 Pop

Name: POP  
Opcode: 0x89

Pop one word and discard it:
\[ sp \leftarrow sp + 1 \]
\[ ip \leftarrow ip + 1 \]

4.12 Duplicate

Name: DUP  
Opcode: 0x8A

Pop one word and push it twice:
\[ a \leftarrow m(sp) \]
\[ m(--sp) \leftarrow a \]
\[ ip \leftarrow ip + 1 \]

4.13 Exchange

Name: EXCH  
Opcode: 0x8B

Pop one word, pop another, then push the one, push the other:
\[ a \leftarrow m(sp++) \]
\[ b \leftarrow m(sp++) \]
\[ m(--sp) \leftarrow a \]
\[ m(--sp) \leftarrow b \]
\[ ip \leftarrow ip + 1 \]

4.14 Load From Stack
Pop an index, add it to the stack pointer, push word it points to:

\[
\begin{align*}
a & \leftarrow m(sp++) \\
b & \leftarrow m(sp + a) \\
m(--sp) & \leftarrow b \\
ip & \leftarrow ip + 1
\end{align*}
\]

### 4.15 Swap Bit Fields

Pop a 5 bit mask, pop a word, for each set bit \(i\) in the mask reverse all 32 bits in a word, exchanging each bit with the bit at offset \(2^i\). Use mask 24 for endianness swap, use mask 31 for full bitwise reversal:

\[
\begin{align*}
a & \leftarrow m(sp++) \\
b & \leftarrow m(sp++) \\
\text{for } i \text{ in } 0..4 \text{ do} \\
\quad \text{if } a_i = 1 \text{ then} \\
\quad \quad \text{for } k \text{ in } 0..31 \text{ do} \\
\quad \quad \quad b'_{k} & \leftarrow b_{(k \text{ xor } (2^i))} \\
\quad \quad b & \leftarrow b' \\
ip & \leftarrow ip + 1
\end{align*}
\]

### 4.16 Load, Decrement, Push, Store

Pop an index, add it to the data pointer, load the word it points to, decrement it, push it, store it back:

\[
\begin{align*}
a & \leftarrow m(sp++) \\
b & \leftarrow m(a + dp) \\
b & \leftarrow b - 1 \\
m(--sp) & \leftarrow b \\
m(a + dp) & \leftarrow b \\
ip & \leftarrow ip + 1
\end{align*}
\]

### 4.17 Logarithm
Determine position of highest bit set, -1 for zero word:
\[ a \leftarrow m(sp++) \]
\[ \text{if } a = 0 \text{ then } b \leftarrow -1 \]
\[ \text{else } \]
\[ b \leftarrow i \mid a_i = 1 \land \forall j > i : a_j = 0 \]
\[ ip \leftarrow ip + 1 \]

### 4.18 Shift And Rotate Left

Pop a bit count, pop a rotator word and a shifter word, shift the shifter by count bits to left, shifting in the bits from the rotator, which is rotated to the left by count bits, push the shifter result:
\[ a \leftarrow m(sp++) \]
\[ b \leftarrow m(sp++) \]
\[ c \leftarrow m(sp++) \]
\[ a' \leftarrow a_{4..0} \]
\[ \text{if } a < 32 \text{ then } \]
\[ c'_{31..a'} \leftarrow c_{(31-a')}..0 \]
\[ \text{else } \]
\[ c'_{31..a'} \leftarrow b_{(31-a')}..0 \]
\[ c'_{(a'-1)..<0} \leftarrow b_{31..(32-a')} \]
\[ m(--sp) \leftarrow c' \]
\[ ip \leftarrow ip + 1 \]

### 4.19 Shift And Rotate Right

Pop a bit count n, pop a rotator word and a shifter word, shift the shifter by n bits to right, shifting in the bits from the rotator, which is rotated to the right by n bits, push the shifter result:
\[ a \leftarrow m(sp++) \]
\[ b \leftarrow m(sp++) \]
\[ c \leftarrow m(sp++) \]
\[
a' \leftarrow a_{4.0}
\]

if \( a < 32 \) then
\[
c'_{(31-a')}..0 \leftarrow c_{31..a'}
\]
else
\[
c'_{(31-a')}..0 \leftarrow b_{31..a'}
\]
\[
c'_{31..(32-a')} \leftarrow b_{(a'-1)..0}
\]
\[
m(--sp) \leftarrow c'
\]
ip \leftarrow ip + 1

### 4.20 Sign Extend

| Name: SIGN | Opcode: 0x92 |
|------------|--------------|

Pop a word, push true if it is negative, false otherwise:
\[
a \leftarrow m(sp++)
\]
\[
a_{31..1} \leftarrow a_{0}
\]
\[
m(--sp) \leftarrow a
\]
ip \leftarrow ip + 1

### 4.21 Zero

| Name: ZERO | Opcode: 0x93 |
|------------|--------------|

pop a word, push true if it is zero, false otherwise:
\[
a \leftarrow m(sp++)
\]
if \( a = 0 \) then
\[
b \leftarrow -1
\]
else
\[
b \leftarrow 0
\]
m(--sp) \leftarrow b
ip \leftarrow ip + 1

### 4.22 Unconditional Jump

| Name: UJP | Opcode: 0x94 |
|-----------|--------------|

Pop an offset, add it to the current instruction pointer:
\[
a \leftarrow m(sp++)
\]
ip \leftarrow ip + a
4.23 False Jump

| Name: FJP         | Opcode: 0x95 |
|------------------|-------------|

Pop an offset and a condition, if condition is false, add the offset to the current instruction pointer:

\[ a \leftarrow m(sp++) \]
\[ b \leftarrow m(sp++) \]
\[ \text{if } b = 0 \text{ then} \]
\[ \quad \text{ip} \leftarrow \text{ip} + a \]
\[ \text{else} \]
\[ \quad \text{ip} \leftarrow \text{ip} + 1 \]

4.24 Load Constant

| Name: LDC         | Opcode: 0x96 |
|------------------|-------------|

Pop an index, add it to the constant pointer, push word it points to. However, if that pointer is out of range of the constant limits, then subtract the constant pool size (lc_1 - lc_0) and add the data pointer instead:

\[ a \leftarrow m(sp++) \]
\[ \text{if } (a + cp) \geq lc_1 \text{ then} \]
\[ \quad b \leftarrow a - (lc_1 - lc_0) + dp \]
\[ \text{else} \]
\[ \quad b \leftarrow a + cp \]
\[ m(--sp) \leftarrow m(b) \]
\[ \text{ip} \leftarrow \text{ip} + 1 \]

4.25 Load Word

| Name: LD          | Opcode: 0x97 |
|------------------|-------------|

Pop an index, add it to the data pointer, push word it points to:

\[ a \leftarrow m(sp++) \]
\[ m(--sp) \leftarrow m(a + dp) \]
\[ \text{ip} \leftarrow \text{ip} + 1 \]

4.26 Store Word
Pop an index, add it to the data pointer, pop a word and store it:
\[ a \leftarrow m(sp++) \]
\[ m(a + dp) \leftarrow m(sp++) \]
\[ ip \leftarrow ip + 1 \]

### 4.27 Count Bits

Pop a word, count the number of bits set:
\[ a \leftarrow m(sp++) \]
\[ b \leftarrow \sum_{i=0..31} (a_i) \]
\[ ip \leftarrow ip + 1 \]

### 4.28 Stop

Stop the thread:
stop

### 4.29 Break

Do nothing, but stall for debugging:
\[ ip \leftarrow ip + 1 \]

### 4.30 Start New Thread

Start a new thread with given limits, ip, and exception destination port, set sp to ld1, dp to ld0+64, and cp to lc0+64, push port id of new threads control port:
if all threads in use then fault
exc ← m(sp++)
ld₁’ ← m(sp++)
ld₀’ ← m(sp++)
b ← m(sp++)
lc₁’ ← m(sp++)
lc₀’ ← m(sp++)
ip’ ← (b + lc₀’) * 4
sp’ ← ld₁’
a ← port(thread’, 0)
m(--sp) ← a
ip ← ip + 1

4.31 Call Function

| Name: CALL | Opcode: 0x9D |
|------------|--------------|

Pop an offset, add it to the current instruction pointer, save return address to the stack:
a ← m(sp++)
m(--sp) ← ip + 1 - lc₀
ip ← ip + a

4.32 Return From Function

| Name: JUMP | Opcode: 0x9E |
|------------|--------------|

Restore the instruction pointer from the stack:
a ← m(sp++)
ip ← a + lc₀

4.33 Store To Stack

| Name: STX | Opcode: 0x9F |
|-----------|--------------|

Pop an index, pop a word, add the index to the stack pointer, store word:
a ← m(sp++)
b ← m(sp++)
$m(sp + a) \leftarrow b$
$ip \leftarrow ip + 1$

### 4.34 Load, Push, Increment, Store

| Name: LDINC | Opcode: 0xA0 |
|-------------|--------------|

Pop an index, add it to the data pointer, load the word it points to, push it, increment it, store it back:

$a \leftarrow m(sp++)$
$b \leftarrow m(a + dp)$
$m(--sp) \leftarrow b$
$b \leftarrow b + 1$
$m(a + dp) \leftarrow b$
$ip \leftarrow ip + 1$

### 4.35 Get Port Destination

| Name: GETPORT | Opcode: 0xA1 |
|---------------|--------------|

Pop local port id, push local ports destination:

$a \leftarrow m(sp++)$
$b \leftarrow dest_a$
$m(--sp) \leftarrow b$
$ip \leftarrow ip + 1$

### 4.36 Set Port Destination

| Name: SETPORT | Opcode: 0xA2 |
|---------------|--------------|

Pop local port id and remote port id, set local ports destination:

$a \leftarrow m(sp++)$
$b \leftarrow m(sp++)$
$dest_a \leftarrow b$
$ip \leftarrow ip + 1$

### 4.37 Output Word
Name: OUT  Opcode: 0xA3

Pop local port id and a data word, send data into port; may block:
\[ a \leftarrow m(sp++) \]
\[ b \leftarrow m(sp++) \]
\[ out_a \leftarrow b \]
\[ ip \leftarrow ip + 1 \]

4.38 Output End Token

Name: OUTEND  Opcode: 0xA4

Pop local port id, send an END token into port; may block:
\[ a \leftarrow m(sp++) \]
\[ out_a \leftarrow END \]
\[ ip \leftarrow ip + 1 \]

4.39 Output Pause Token

Name: OUTPAUSE  Opcode: 0xA5

Pop local port id, send a PAUSE token into port; may block:
\[ a \leftarrow m(sp++) \]
\[ out_a \leftarrow PAUSE \]
\[ ip \leftarrow ip + 1 \]

4.40 Input Word

Name: IN  Opcode: 0xA6

Pop local port id, receive a data word from port, push it; may block:
\[ a \leftarrow m(sp++) \]
\[ b \leftarrow in_a \]
if \( b = END \) then fault
\[ m(--sp) \leftarrow b \]
\[ ip \leftarrow ip + 1 \]
4.41 Check Input Port

Name: INMORE  Opcode: 0xA7

Pop local port id, check whether end token was received, discard end token and push zero if so, push non-zero otherwise; may block:

\[
a \leftarrow m(sp++)
\]
\[
\text{if } \text{in}_a = \text{END} \text{ then}
\]
\[
b \leftarrow \text{in}_a
\]
\[
b \leftarrow 0
\]
\[
\text{else}
\]
\[
b \leftarrow -1
\]
\[
m(\text{-}sp) \leftarrow b
\]
\[
ip \leftarrow ip + 1
\]

4.42 Clear Event List

Name: EVCLEAR  Opcode: 0xA8

Clear the event list for current thread:

\[
ev_{s,s} \leftarrow \epsilon
\]
\[
ip \leftarrow ip + 1
\]

4.43 Set Output Event

Name: EVOUT  Opcode: 0xA9

Pop local port id, add event handle for output availability, pop an offset, add it to the current instruction pointer, use as event vector:

\[
a \leftarrow m(sp++)
\]
\[
b \leftarrow m(sp++)
\]
\[
ev_{a,out} \leftarrow b + ip
\]
\[
ip \leftarrow ip + 1
\]

4.44 Set Input Event

Name: EVIN  Opcode: 0xAA


Pop local port id, add event handle for input availability, pop an offset, add it to the current instruction pointer, use as event vector:

\[a \leftarrow \text{m}(\text{sp}++), b \leftarrow \text{m}(\text{sp}++)\]
\[\text{ev}_{a,\text{in}} \leftarrow b + \text{ip}, \text{ev}_{a,\text{end}} \leftarrow b + \text{ip}\]
\[\text{ip} \leftarrow \text{ip} + 1\]

### 4.45 Set Input End Event

| Name: EVEND | Opcode: 0xAB |

Pop local port id, add event handle for input end token availability, pop an offset, add it to the current instruction pointer, use as event vector:

\[a \leftarrow \text{m}(\text{sp}++), b \leftarrow \text{m}(\text{sp}++)\]
\[\text{ev}_{a,\text{end}} \leftarrow b + \text{ip}\]
\[\text{ip} \leftarrow \text{ip} + 1\]

### 4.46 Wait

| Name: WAIT | Opcode: 0xAC |

Wait for any of the configured events, as soon as any one occurs, load the corresponding event vector as new instruction pointer; may block:

\[
\text{if } \text{out}_i = \epsilon \text{ and } \text{ev}_{a,\text{out}} \text{ then} \\
\quad \text{ip} \leftarrow \text{ev}_{a,\text{out}}
\]

\[
\text{or if } \text{in}_i = \text{END} \text{ and } \text{ev}_{a,\text{end}} \text{ then} \\
\quad \text{ip} \leftarrow \text{ev}_{a,\text{end}}
\]

\[
\text{else if } \text{in}_i \neq \epsilon \text{ and } \text{ev}_{a,\text{in}} \text{ then} \\
\quad \text{ip} \leftarrow \text{ev}_{a,\text{in}}
\]

\[
\text{else} \\
\quad \text{wait}
\]

### 4.47 Current Time

| Name: NOW | Opcode: 0xAD |

Push the current time counter:
m(--sp) ← time
ip ← ip + 1

4.48 Wait With Timeout

| Name: WAITTMO | Opcode: 0xAE |
|---------------|--------------|

Pop a time, wait for any of the configured events. As soon as any one occurs, load the corresponding event vector as new instruction pointer. When no event occurs until the specified time is reached, continue instruction execution without branching; may block:

\[
a ← m(sp++)
\]

if time - a ≥ 0 then
  ip ← ip + 1
or if out, = ε and eva,out then
  ip ← eva,out
or if in, = END and eva,end then
  ip ← eva,end
else if in, ≠ ε and eva,in then
  ip ← eva,in
else
  wait

4.49 Increment Stack Pointer

| Name: POPN | Opcode: 0xAF |
|------------|--------------|

Pop a summand, add it to the stack pointer:

\[
a ← m(sp++)
\]

sp ← sp + a
ip ← ip + 1

4.50 Compare Unsigned

| Name: ULESS | Opcode: 0xB0 |
|-------------|--------------|

Pop a subtrahend, pop a minuend, subtract the subtrahend from the minuend, push true if negative, false otherwise, all unsigned:
a ← m(sp++)
b ← m(sp++)
if b < a then
c ← -1
else
c ← 0
m(--sp) ← c
ip ← ip + 1

4.51  Compare Signed

| Name      | Opcode: 0xB1 |
|-----------|--------------|

Pop a subtrahend, pop a minuend, subtract the subtrahend from the minuend, push true if negative, false otherwise, all signed:

a ← two's complement m(sp++)
b ← two's complement m(sp++)
if b < a then
c ← -1
else
c ← 0
m(--sp) ← c
ip ← ip + 1

4.52  Combined Number

| Name      | Opcode: 0xB2 |
|-----------|--------------|

Pop a word, pop another word, multiply the latter by 192, push the sum:

a ← m(sp++)
b ← m(sp++)
m(--sp) ← b * 192 + a
ip ← ip + 1

4.53  Calculate Global Port Number

| Name      | Opcode: 0xB3 |
|-----------|--------------|

Pop a word, calculate the global port id, push it:
a ← m(sp++)
m(--sp) ← port(thread, a)
ip ← ip + 1

### 4.54 Calculate Stack Pointer

| Name: LDAX         | Opcode: 0xB4 |
|-------------------|--------------|

Pop an index, add it to the stack pointer, push the data pool index for the word it points to:

\[
a ← m(sp++)
a' ← a + sp - dp
m(--sp) ← a'
ip ← ip + 1
\]

### 4.55 Available Threads

| Name: THREADS     | Opcode: 0xB5 |
|-------------------|--------------|

Determine the number of threads available to start:

\[
m(--sp) ← \sum_{i=0..7}(-started_i)
ip ← ip + 1
\]

### 4.56 Cycles Per Threads

| Name: THRCYC      | Opcode: 0xB6 |
|-------------------|--------------|

Push the instruction cycle counter of the thread:

\[
m(--sp) ← cycles_i
ip ← ip + 1
\]

### 4.57 Total Cycles

| Name: CYCLES      | Opcode: 0xB7 |
|-------------------|--------------|

Push the total instruction cycle counter of all threads of the processing unit:

\[
m(--sp) ← \sum_{i=0..7}(cycles_i)
ip ← ip + 1
\]
5 Implementation

There is no hardware implementation of the Null Operand Parallel processor, but a software simulation. As this simulator is to be taken as a proof of concept only, optimisation efforts have been restricted to the bare minimum. It may be given a number of command line parameters to take influence on its detailed behaviour. Next to a number of possible options, it is given a socket number so it can handle a number of sockets, one for each of the four external links. Furthermore, it may be given up to four socket numbers of other simulator instances which it shall connect to:

\texttt{nopsim [options] first-own-socket [connect-to-socket ...]}

For a first compiler to support code generation, see [2016os]. However, there is no reason to not implement further compilers for other languages.

Options to the simulator are:

| Option | Description |
|--------|-------------|
| \texttt{-i} --init file | initial code to read, first five words skipped |
| \texttt{-f} --file file | open file on peripheral link, starting at third, i.e. at internal peripheral link #2. link #0 is always connected to standard input, link #1 is always connected to standard output |
| \texttt{-t} --trace=mask | full trace, optionally restricted to threads: For the 32 bit mask, each single bit represents a thread, starting at processing unit #0, thread #0, processing unit #0, thread #1, and so forth, up to processing unit #3, thread #7. For each bit set, for the corresponding thread a full trace is sent to stderr. The mask may be omitted to ask for trace of all threads |
| \texttt{-x} --extern=mask | external links trace, optionally for selected links. Bit 0 to 3 represent the external links, bit 4 to 11 represent peripheral lines, bit 12 represents the router configuration block |
| \texttt{-l} --intern=mask | internal links trace, optionally restricted to threads. The mask to be used as with \texttt{--trace} |
| \texttt{-d} --debug | enable break instruction and debug mode |
| \texttt{-h} --help | show help and exit |

Initially, the simulator will open sockets first-own-socket to first-own-socket+3. For the first \(n\) among these, with \(n\) the number of connect-to-socket parameters, it will attempt to connect to the latter sockets. For the remaining \(4 - n\)
sockets, i.e. \texttt{first-own-socket+n} to \texttt{first-own-socket+3}, it will wait for another simulator to connect to it. Only when all four sockets are connected, the simulator will start executing opcodes.

As a special case, a stand-alone system may be simulated by omitting all parameters.

Each of the processing units will run code from a simulated boot ROM at address 0x3fc0, which contains the following instructions intended to read some initial code from its first internal port (see option \texttt{--init}):

\begin{verbatim}
0 IN      read initial instruction word position
-64 INMORE initial memory reference (first word)
10 FJP    when done, skip to POP below
DUP       make use of memory reference
0 IN       read a word
EXCH ST    store it to memory
1 ADD      increment memory reference
-12 UJP    loop for next word
POP        discard memory reference
4 MUL JUMP transfer control to newly loaded code
\end{verbatim}

\section{Discussion}

The main purpose of the \textit{Null Operand Parallel} processor is to allow for a prototypical implementation of a dynamic software system as a proof of concept, and its design is restricted to features that are needed to serve this purpose.

For instruction encoding a single operation stack oriented approach without registers – occasionally referred to as \textit{bytecode} – has been chosen, to achieve compact encoding, and to allow for a most simple code generation, reducing effort in compiler implementation. Clearly there are disadvantages, e.g. code execution is slower than with a multiple register encoded instruction set, because extra instructions are needed to fetch and store operands. Other stack oriented designs have been proposed, that implement multiple operations per instruction, with the overall number of instructions reduced as to allow for more compact implementation in hardware (e.g. [2010jb]).

Program flow exceptions have been replaced by exception messages, but these need to be reduced to a minimum, as a process should care for as much exceptional states as possible on its own. Exceptional states may be divided into two classes, one class are exceptional states that arise due to wrong encoding, e.g. illegal instructions, resource inavailabilities, stack overflow and
other range check faults. These exceptions need external handling, so it is inevitable to send a message to some responsible process. The second class are exceptional states that arise due to unexpected input data, e.g. division by zero, and sudden reception of an end token where a message was not expected to end. These exceptions should all be handled by the process itself. For sudden reception of an end token, the instruction encoding could be chosen to force evaluation of the token type received by the process, e.g. by extending the input instruction to return an extra boolean to indicate the type of the token received. For division by zero, a similar approach could be taken, e.g. by extending the division instruction to return an extra boolean to indicate successful operation, or by simply redefining the division operation to result in a defined value whenever the denominator is zero. The latter still requires the process to do some extra check on the denominator first, but this is not worse than the traditional exception based handling.

Directly connected to exception handling is the question of how to respond to processes dying prematurely, resulting in ports and thus channels to be no longer available. For channels used for outgoing messages, any pending message is finalised with an end token, but for channels used for incoming messages, the process sending a message would need to be notified about the fact that the destination has vanished.

Support for floating point calculation has not been implemented, as it is not needed for basic operating system design.

True general purpose I/O lines are not foreseen, because an implementation in hardware is not planned with this first design. There are other designs that prove smooth integrability into a channel based design is possible (e.g. [2009dm]).
Literature

[1963bc] Burroughs Corporation: “The Operational Characteristics of the Processor for the Burroughs B5000”, 1963, Detroit, Michigan

[2001Da] Daan Leijen: “Division and Modulus for Computer Scientists”, University of Utrecht, Dept. of Computer Science, December 3, 2001

[2007Le] Charles Eric LaForest: “Second-Generation Stack Computer Architecture”, April 2007, University of Waterloo, Canada

[2008Es] ECSS Secretariat: “SpaceWire – Links, nodes, routers and networks”, ECSS-E-ST-50-12C, 31 July 2008, ESA-ESTEC.

[2009De] David May: “The XMOS XS1 Architecture”, Version 1.0, XMOS Ltd., 2009/10/19

[2010Jo] James Bowman: “J1: a small Forth CPU Core for FPGAs”, Willow Garage, Menlo Park, CA, 2010

[2011Ga] GreenArrays, Inc.: “F18A Technology Reference – Product Data Book”, http://www.greenarraychips.com/home/documents/greg/DB001-110412-F18A.pdf, 12 April 2011

[2016Go] Oskar Schirmer: “GuStL – An Experimental Guarded States Language”, Göttingen, 2016