NAC (N-address code) programming language reference

Date: 2011-07-01
Author: Nikolaos Kavvadias
Email: nikolaos.kavvadias@gmail.com
Revision: 0.0.4 (2011-07-01), 0.0.3 (2010-11-29), 0.0.2 (2010-10-11), 0.0.1 (2010-09-05)
Web site: http://www.nkavvadias.com

Contents

1 Introduction

NAC (N-Address Code) is the name of a simplistic imperative programming language with light semantics devised by Nikolaos Kavvadias. Its main use is as an executable/interpretable intermediate representation for compilation frameworks (compilers, high-level synthesis tools, etc).

NAC statements are either labels, n-address instructions or procedure calls.

A label is formatted as follows:

- label:

An n-address instruction is actually the specification of a mapping from a set of n ordered inputs to a set of m ordered outputs. An n-address instruction (or else termed as an {m, n}-NAC) is formatted as follows:

- `outp1, ..., outpm <= operation inp1, ..., inpn;`

where

- *operation* is a mnemonic referring to an IR-level instruction
- *outp1*, ..., *outpm* are the m outputs of the instruction
- *inp1*, ..., *inpn* are the n inputs of the instruction

Similarly, a procedure call, which is a non-atomic operation is formatted as follows, in order to distinguished from an atomic operation:

- `(outp1, ..., outpm) <= procedure-name (inp1, ..., inpn);`

where

- *procedure-name* is the name of called procedure.

For a procedure without input and output arguments, the following notation is used to distinguish it from an atomic operation with no arguments:

- `() <= procedure-name ();`

NAC is a typed language. This means all declared objects (global variables, constants, local variables, input and output procedure arguments) have a type specification. Data types in NAC are classified in the following categories:

- UNSIGNED_INTEGER denoted as U<num>: [Uu][0-9]+
- SIGNED_INTEGER denoted as S<num>: [Ss][0-9]+
- Fixed-point numbers are denoted as Q<ipart>.F<ipart>[S|U]: Q[0-9]+.[0-9]+[S|U], with ipart being the integer part and fpart the fractional part of the number. SIGNED_FIXED_POINT uses the *S* suffix, whereas UNSIGNED_FIXED_POINT uses the *U* suffix, correspondingly
- FLOATING_POINT denoted as F<spart>.<epart>.<mpart>: F[0|1].[0-9]+.[0-9]+, with spart being the sign, epart the exponent and mpart the mantissa of the number
- RATIONAL (no consistent format yet)
- CONTINUED_FRACTION (no consistent format yet)

As of 2010-11-29, there is initial support for the SIGNED_FIXED_POINT and UNSIGNED_FIXED_POINT data types. Support for UNSIGNED_INTEGER and SIGNED_INTEGER data types is considered mature.

In NAC parlance, the following keywords are used:

`globalvar` a global scalar or single-dimensional array variable. An array variable is permitted to have an optional numerical initialization. A scalar variable is assumed to be initialized to zero.

`localvar` a local scalar or single-dimensional array variable. This variable is only visible within the procedure. Again, an array variable is permitted to have an optional numerical initialization. A scalar variable is assumed to be initialized to zero.

`in` an input argument to the given procedure.

`out` an output argument to the given procedure.

Please note that the use of `constant` (declaration of a globally-visible constant value) has been discontinued and will not be supported in the future.

2 NAC instructions

The NAC programming language is extensible, meaning that the grammar accepts user-specific instruction mnemonics.

A common set of NAC instructions is listed below, along with the corresponding format and description.

No-operation: `nop`

```
nop;
```

Performs no action at all.

Move operand: `mov`

```
dst1 <= mov src1;
```

Copy the contents of operand src1 to dst1.

Load constant: `ldc`

```
dst1 <= ldc cnst1;
```

Copy the value of cnst1 to operand dst1.

Unconditional jump: `jmpun`

```
S_dst1 <= jmpun;
```

Jump to label `S_dst1`.

Conditional jump: `jmpeq`, jmpne“, `jmplt`, jmple“, `jmpgt`, jmpge“

```
S_TRGT, S_TRGF <= jmpzz src1, src2;
```

where:

- `zz` can be one of the following:
  - eq: jump if equal
  - ne: jump if not equal
  - lt: jump if less than
  - le: jump if less than or equal
  - gt: jump if greater than
  - ge: jump if greater than or equal
- src1, src2 are the instruction source operands
- S_TRGT, S_TRGF, are the target addresses for a true and false condition, respectively

Binary logical instructions: `and`, `ior`, `xor`, `nand`, `nor`, `xnor`

```
dst1 <= <mnemonic> src1, src2;
```

where:

- mnemonic can be one of the following:
  - and: Logical AND
  - ior: Logical inclusive-OR
  - xor: Logical exclusive-OR
  - nand: Logical NAND
  - nor: Logical NOR
  - xnor: Logical XNOR

- src1, src2 are the source operands
- dst1 is the destination operand

Unary logical instruction: `not`

```
dst1 <= not src1;
```

Copy the 1's-complement of operand src1 to dst1.

Binary arithmetic instructions: `add`, `sub`

```
dst1 <= mnemonic src1, src2;
```

where:

- mnemonic can be one of the following:
  - add: 2's-complement addition
  - sub: 2's-complement subtraction
- src1, src2 are the source operands
- dst1 is the destination operand

Unary arithmetic instructions: `neg`

```
dst1 <= neg src1;
```

Copies the negated version of src1 to dst1.

Quaternary multiplexing instruction: `mux`

```
dst1 <= muxzz src1, src2, src3, src4;
```

where:

- zz can be one of the following:
  - eq: jump if equal
  - ne: jump if not equal
  - lt: jump if less than
  - le: jump if less than or equal
  - gt: jump if greater than
  - ge: jump if greater than or equal
- src1, src2, are the source operands compared: `if (src1 zz src2)`
- src3 is the copy operand when the comparison evaluates to TRUE
- src4 is the copy operand when the comparison evaluates to FALSE
- dst1 is the destination operand
- NOTE: A `muxzz` is equivalent to the following C code:

```
if (src1 zz src2) { // zz is either "==", "!=", "<", "<=", ">", or ">="
  dst1 = src3;
} else {
  dst1 = src4;
}
```

Set on comparison instruction: `set`

```
dst1 <= setzz src1, src2;
```

where:

- zz can be one of the following:
  - eq: jump if equal
  - ne: jump if not equal
  - lt: jump if less than
  - le: jump if less than or equal
  - gt: jump if greater than
  - ge: jump if greater than or equal
- src1, src2, are the source operands compared: `src1 zz src2`
- src3 is the copy operand when the comparison evaluates to TRUE
- src4 is the copy operand when the comparison evaluates to FALSE
- dst1 is the destination operand (gets a value either 0 or 1).
- NOTE: A `setzz` is equivalent to the following C code:

```
‘‘dst1 = (src1 zz src2); // zz is either "==", "!=", "<", "<=", ">", or ">="
```

Complex unary arithmetic instructions: `abs`

```
dst1 <= abs src1;
```

Copies the absolute value of src1 to dst1.

Complex binary arithmetic instructions: `max`, `min`

```
dst1 <= mnemonic src1, src2;
```

where:

- mnemonic can be one of the following:
  - max: Assign the maximum of src1 and src2 to dst1
  - min: Assign the minimum of src1 and src2 to dst1
- src1, src2 are the source operands
- dst1 is the destination operand

Shift instructions: `shl`, `shr`

```
dst1 <= mnemonic src1, src2;
```

where:

- mnemonic can be one of the following:
  - shl: Logical left shift of src1 by the amount stored in src2, with the result copied to dst1
  - shr: Either logical or arithmetic (depending on the operand data types) shift of src1 by the amount stored in src2, with the result copied to dst1
- src1, src2 are the source operands
- dst1 is the destination operand

Rotate instructions: `rotl`, `rotr`

```
dst1 <= mnemonic src1, src2;
```

where:

- mnemonic can be one of the following:

  - rotl: Left rotation of the value of src1 by the amount stored in src2, with the result copied to dst1
  - rotr: Right rotation of the value of src1 by the amount stored in src2, with the result copied to dst1
- src1, src2 are the source operands
- dst1 is the destination operand

Multiplication instructions: `mul`

```
dst1 <= mul src1, src2;
```

Multiplies the contents of src1 and src2 and copies the (possibly truncated) result to dst1.

Combined division-modulus instructions: `divrem`

```
dst1, dst2 <= divrem src1, src2;
```

Divides the contents of src1 and src2 and copies the quotient to dst1 and the remainder to dst2.

Division instructions: `div`, `rem`

```
dst1 <= mnemonic src1, src2;
```

where:

- mnemonic can be one of the following:
  - div: Divides the contents of src1 and src2 and copies the quotient to dst1
  - rem: Divides the contents of src1 and src2 and copies the remainder to dst1
- src1, src2 are the source operands
- dst1 is the destination operand

Data type/bitwidth conversion instructions: `zxt`, `sxt`, `trunc`

```
dst1 <= mnemonic src1;
```

where:

- mnemonic can be one of the following:
  - zxt: Zero-extends src1 to the (larger) bitwidth of dst1
  - sxt: Sign-extends src1 to the (larger) bitwidth of dst1
  - trunc: Truncates src1 to the (smaller) bitwidth of dst1
- src1 is the source operand
- dst1 is the destination operand

Bit manipulation instructions: `bitins`, `bitext`

```
dst1 <= mnemonic src1, src2, src3;
```

where:

- mnemonic can be one of the following:
  - bitins: Insert a bitvector denoted by the downto range [src2..src3] of src1 to dst1
  - bitext: Extract a bitvector denoted by the downto range [src2..src3] from src1 and assign it to dst1
- src1 is the source operand

- src2 are two source operands (constant or variables) that denote the downto range. The runtime numerical value of src2 must be larger or equal to src3, and within the range of dst1
- dst1 is the destination operand

These instructions define bitfield insertion and extraction primitives. They can also be defined for fixed-point operands given additional constraints.

Load variable from array: `load`

```
dst1 <= load src1, src2;
```

Loads the contents of array src1 from the absolute address src2 to the variable dst1.

Store variable to array: `store`

```
dst1 <= store src1, src2;
```

Stores the value of variable src1 to address src2 of array dst1.

2.1 Fixed-point operators

This is a proposed list of extension operators for use with UNSIGNED_FIXED_POINT and SIGNED_FIXED_POINT variables.

Conversion from integer to fixed-point format: `i2ufx`, `i2sfx`

```
dst1 <= i2zfx src1;
```

where:

- `z` can be one of the following:
  - u: conversion to the ufixed (UNSIGNED_FIXED_POINT) format
  - s: conversion to the sfixed (SIGNED_FIXED_POINT) format
- src1 is the source operand
- dst1 is the destination operand

Converts an integer to a fixed-point number without loss of precision.

Conversion from fixed-point to integer format: `ufx2i`, `sfx2i`

```
dst1 <= zfx2i src1;
```

where:

- `z` can be one of the following:
  - u: conversion to the ufixed (UNSIGNED_FIXED_POINT) format
  - s: conversion to the sfixed (SIGNED_FIXED_POINT) format
- src1 is the source operand
- dst1 is the destination operand

Converts a fixed-point number to an integer. In case of a non-zero fractional part of the fixed-number, truncation occurs. The type of the integer result (UNSIGNED_INTEGER or SIGNED_INTEGER) must be compatible to the type of the fixed-point input argument to assure a proper conversion.

Resize instruction: `resize`

```
dst1 <= resize src1, src2, src3;
```

where:

- src1 is the source fixed-point operand
- src2, src3 are numerical values (integers) that denote the new size (high-to-low range) of the resulting fixed-point operand
- dst1 is the destination fixed-point operand

Fixed-point rounding instructions: `ceil`, `fix`, `floor`, `round`, `nearest`, `convergent`

```
dst1 <= mnemonic src1;
```

where:

- src1 is the source operand
- dst1 is the destination operand

These operations are used to performing rounding of fixed-point operands with different criteria. They emulate the behavior of corresponding MATLAB intrinsic functions. Rounding behavior is summarized as follows:

- ceil: round towards plus infinity.
- fix: round towards zero.
- floor: round towards minus infinity.
- round: round to nearest; ties to greatest absolute value.
- nearest: round to nearest; ties to plus infinity.
- convergent: round to nearest; ties to closest even.

3 Macroinsructions

For simplifying programming in the NAC language, a set of macroinstructions are available:
A) Automatic replacement of incomplete conditional jumps: The pattern

```
S_TRUE <= jmpxx opnd1, opnd2;
```

is replaced by:

```
S_TRUE, S_FALSE <= jmpxx opnd1, opnd2;
S_FALSE:
```

Label `S_FALSE` is generated only if it doesn't already exist.
B) Addition of "forgotten" unconditional jumps. The pattern:

```
      no-jump-instruction;
LABEL:
```

is replaced by:

```
      no-jump-instruction;
      LABEL <= jmpun;
LABEL:
```

4 Structure of a NAC program

A NAC program can be specified in a single source file that can contain global variable definitions and their initializations, constant declarations, and a list of procedures. Each procedure is comprised of the following: - the procedure name - a list of ordered input arguments - a list of ordered output arguments - a list of localvar declarations - a list of statements (the main NAC subprogram)

The typical NAC program is structured as follows:

```
<Global variable declarations>

procedure <name-1> (
  <comma-separated input arguments>,
  <comma-separated output arguments>
)
{
  <Local variable declarations>
  <NAC labels, instructions and procedure calls>
}
...
procedure <name-n> (
  <comma-separated input arguments>,
  <comma-separated output arguments>
)
{
  <Local variable declarations>
  <NAC labels, instructions and procedure calls>
}
```

5 Data type inference rules

Since version 0.0.3 the declarations of `constant` items has been removed, and for this reason constant items are recognized by scanning through the NAC program prior any actual further manipulations (e.g. code generation). A small set of simple rules are used for data type inference of constant values:

1. When a constant appears in an "ldc" or "store" operation, it obtains the type of the result operand.
2. When a constant appears in any other operation, then it obtains the type of the first input operand. This assumes that the constant appears only as the second, third or fourth input operand for this operation.

6 NAC grammar

Here follows the BNF-style grammar specification for the NAC programming language.

6.1 YACC/bison grammar

This grammar uses the notation of the YACC/Bison parser generators.

```
%token T_LPAREN T_RPAREN T_LBRACE T_RBRACE T_LBRACKET T_RBRACKET
%token T_COMMA T_COLON T_SEMI T_ASSIGN T_EQUAL
%token T_PROCEDURE T_LOCALVAR T_GLOBALVAR T_CONSTANT T_IN T_OUT
%token T_ID

%start nac_top
```

```
%%

nac_top : procedure_list
  | globalvar_def procedure_list
  ;

globalvar_def : globalvar_prefix id_list T_SEMI
  | globalvar_def globalvar_prefix id_list T_SEMI
  ;

globalvar_prefix : T_GLOBALVAR type_spec
  ;

procedure_def : proce-
dure_prefix T_LPAREN arg_list T_RPAREN T_LBRACE stmt_list T_RBRACE
  | procedure_prefix T_LPAREN arg_list T_RPAREN T_LBRACE local-
var_list stmt_list T_RBRACE
  ;

procedure_list : procedure_def
  | procedure_list procedure_def
  ;

procedure_prefix : T_PROCEDURE id
  ;

localvar_list : localvar_prefix id_list T_SEMI
  | localvar_list localvar_prefix id_list T_SEMI
  ;

localvar_prefix : T_LOCALVAR type_spec
  ;

stmt_list : /* empty */
  | stmt_list
  stmt
  ;

stmt : nac
  | pcall
  | label
  ;

nac :  opnd_out_list assign_op id opnd_in_list T_SEMI
  | opnd_out_list assign_op id T_SEMI
  | id opnd_in_list T_SEMI
  | id T_SEMI
  ;

pcall :  T_LPAREN opnd_out_list T_RPAREN as-
sign_op id T_LPAREN opnd_in_list T_RPAREN T_SEMI
  | T_LPAREN opnd_out_list T_RPAREN assign_op id T_SEMI
```

```
  | id T_LPAREN opnd_in_list T_RPAREN T_SEMI
  | T_LPAREN T_RPAREN assign_op id T_LPAREN T_RPAREN T_SEMI
  ;

assign_op : T_ASSIGN
  ;

label :  id T_COLON
  ;

opnd_out_list : id_list
  ;

opnd_in_list : id_list
  ;

arg_list : /* empty */
  | arg_in
  | arg_out
  | arg_list T_COMMA arg_in
  | arg_list T_COMMA arg_out
  ;

arg_in : T_IN type_spec id
  ;

arg_out : T_OUT type_spec id
  ;

id_list : id
  | id_list T_COMMA id
  ;

id : T_ID
  ;

type_spec : T_ID
  ;
```

6.2 EBNF grammar

This grammar follows the EBNF notation as used by N. Wirth.

```
nac_top = {gvar_def} {proc_def}.
gvar_def = "globalvar" anum decl_item_list ";".
proc_def = "proce-
dure" [anum] "(" [arg_list] ")" "{" [{lvar_decl}] [{stmt}] "}".
stmt = nac | pcall | id ":".
nac = [id_list "<="] anum [id_list] ";".
pcall = ["(" id_list ")" "<="] anum ["(" id_list ")"] ";".
id_list = id {"," id}.
decl_item_list = decl_item {"," decl_item}.
decl_item = (anum | uninitarr | initarr).
```

```
arg_list = arg_decl {"," arg_decl}.
arg_decl = ("in" | "out") anum (anum | uninitarr).
lvar_decl = "localvar" anum decl_item_list ";".
initarr = anum "[" id "]" "=" "{" numer {"," numer} "}".
uninitarr = anum "[" [id] "]".
anum = (letter | "_") {letter | digit}.
id = anum | ["-"] numeric.
numeric = (integer | fxpnum).
fxpnum = [integer] "." integer.
integer = digit {digit}.
```

7 Examples

7.1 2D Euclidean distance approximation (eda.nac)

`eda.nac` is the N-address code (NAC) implementation for a 2D Euclidean distance approximation algorithm given by the equation: `eda = MAX(0.875*x+0.5*y, x) where x = MAX(|a|,|b|), y = MIN(|a|,|b|)`.

```
procedure eda (in s16 in1, in s16 in2, out u16 out1)
{
  localvar u16 x, y, t1, t2, t3, t4, t5, t6, t7;
  localvar s16 a, b;

S_1:
  a <= mov in1;
  b <= mov in2;
  t1 <= abs a;
  t2 <= abs b;
  x <= max t1, t2;
  y <= min t1, t2;
  t3 <= shr x, 3;
  t4 <= shr y, 1;
  t5 <= sub x, t3;
  t6 <= add t4, t5;
  t7 <= max t6, x;
  out1 <= mov t7;
}
```

7.2 Iterative algorithm for the Fibonacci sequence (fibo.nac)

`fibo.nac` is the N-address code (NAC) implementation for the iterative version of Fibonacci series computation.

```
procedure fibo(in u31 n, out u31 outp)
{
  localvar u31 res, x;
  localvar u31 f0, f1, f, k;

  LL0:
    x <= mov n;
    f0 <= ldc 0;
```

```
    f1 <= ldc 1;
    res <= mov f0;
    S_EXIT, LL1 <= jmple x, 0;

  LL1:
    res <= mov f1;
    S_EXIT, LL2 <= jmpeq x, 1;

  LL2:
    k <= ldc 2;
    LL3 <= jmpun;

  LL3:
    f <= add f1, f0;
    f0 <= mov f1;
    f1 <= mov f;
    res <= mov f;
    k <= add k, 1;
    LL3, S_EXIT <= jmple k, x;

  S_EXIT:
    outp <= mov res;
}
```

7.3 Sum of array elements (arraysum.nac)

The following computes the sum of the elements of array arr[10], that is the sum of the first ten primes.

```
globalvar s32 arr[10]={2,3,5,7,11,13,17,19,23,27};

procedure main (in s32 in1, out s32 out1)
{
  localvar s32 D_1963;
  localvar s32 i;
  localvar s32 sum;

L0001:
  sum <= ldc 0;
  i <= ldc 0;
  D_1221 <= jmpun;
D_1220:
  i0 <= mov i;
  D_1963 <= load arr, i;
  sum <= add sum, D_1963;
  i <= add i, 1;
  D_1221 <= jmpun;
D_1221:
  D_1220, D_1222 <= jmplt i, in1;
D_1222:
  out1 <= mov sum;
}
```

8 Suggested coding style - Limitations

Here follows a list of suggestions for easier programming and code generation in NAC.

1. No support for recursion. Actually, can be expressed syntactically, but no attempts have been made to establish interpretation semantics.
2. At each time, a single translation unit (one NAC-file) can be provided as input.
3. Non-root procedures cannot have "streaming" outputs (outputs producing a sequence of values over time).
4. Streaming inputs are syntactically possible but have not yet been thoroughly tested.
5. Global variables are arrays. Scalar globals can be emulated as arrays of size 1.
6. It is probable that a `record` type will be added in order to support high-level programming features, such as ANSI C structs, in a future revision of NAC.
7. Use labels prefixed by `S_` such as: `S_1`, `S_2`, `S_EXIT`. This is not mandatory, just preferred coding style.

And some notes clarifying some issues for potential hardware implementations.

1. An array can be implemented either as a distributed LUT RAM (asynchronous read) or as an embedded memory (synchronous read).
2. The initialization of local array variables of a callee function can only take effect in a potential hardware implementation when applied by an addressing-store NAC operation sequence. This means that, initialization at declaration site, should not be used for localvar arrays for a non-root procedure.
3. Array input and output arguments of procedures donnot map to embedded memories (block RAMs). This also applies for globalvar and localvar arrays that are passed to/from procedures.
4. Global variables should be accessed (read or written) only from within the top-level procedure (root procedure of the program call tree). This is not a permanent limitation since it is possible to establish access from any procedure to global variables given that a proper interconnect is available. Such interconnect could be a multiplexer-based bus. Each procedure would be assigned a unique ID in order to control the corresponding multiplexers (for input data, output data and address ports) interfacing to the global storage. This applies much more easily to single-threaded implementations of NAC programs.