

Nanoassembler specification

Filip Höfer

3rd April 2004

Chapter 1

Nanoassembler

Nanoassembler (or Nanoprocessor Assembler, NA) is a low level programming language for nanoprocessors. This is the specification of the language. We will call any text that conforms to this specification a *nanoprogram*. A line of a nanoprogram will be called a *source line*. Tabs and spaces will be called *white characters*.

NA is a case-insensitive context-free language.

1.1 Basic Elements

NA distinguishes these basic elements:

- directive,
- instruction,
- label,
- empty.

A single basic element is allowed to spread over several source lines. This is achieved by a backslash (\) at the end of source line that continues at the next source line. A single source line may contain at most one element. These elements may be placed in any order unless otherwise stated. An example is shown in Figure 1.1.

An empty element consists of zero or more white characters; the only purpose of this element is format of the nanoprogram. Other elements are described in the following sections.

Figure 1.1: Basic elements

```
#directive
label1:
        instruction operand1,\
        operand2,\
        operand3
label2:
```

Figure 1.2: Comments

```
;this is an empty line
instruction ; this instruction has no operands
```

1.1.1 Comments

Any element may be appended by a comment. A comment begins with a semicolon (;). An example is shown in Figure 1.2.

1.1.2 Identifiers

The first character of a valid identifier is alphabetical or an underscore. Other characters are alphabetical, numerical or underscore. The minimal length of an identifier is one, the maximal length is not limited. Note that NA is case-insensitive.

Identifiers must be unique unless otherwise stated.

1.1.3 Strings

String is a sequence of printable characters except for double quote (") that is delimited by double quotation marks, i.e., "string", "also a string despite of characters ./@:#".

A string is treated as a single case-sensitive entity.

1.1.4 Numbers

NA supports decimal and hexadecimal numbers. Decimal numbers are written in a normal way, hexadecimal numbers begin with 0x; i.e., 30 equals 0x1E.

1.1.5 Arithmetic Expressions

The following operators are supported (ordered in accord to descending precedence):

()	parentheses
+ -	unary plus, unary minus, bitwise not
* /	multiplication, integer division
+ -	addition, subtraction
<< >>	bitwise shift left, bitwise shift right
&	bitwise and
^	bitwise xor
	bitwise or

Allowable operands of these operators are numbers and identifiers. An example of a valid arithmetic expression is (id1-id2) * 3. The result of an arithmetic expression is a number in binary complement code.

1.2 Directives

A directive begins with a sharp (#). Supported directives are enlisted in alphabetical order.

1.2.1 #define

This directive provides three kinds of definitions.

Macro Definitions

Syntax:

```
#define macro1_name right side
```

or

```
#define macro2_name
```

The name of a macro must be a valid identifier (see Section 1.1.2).

In the first case, the name of the macro is followed by a sequence of white characters and the right side. The right side can be any sequence of printable characters that starts with a non-white character and contains even (or zero) number of double quotes—it means that there is no unterminated string. Any occurrence of the left side of the macro in the nanoprogram is automatically expanded into the right side provided that the name of the macro is delimited by one of these separators: white character, comma (,), colon (:), semicolon (;), sharp(#), backslash (\), any brace ({, }, [,], (,)) or arithmetic character (+, -, *, /, >, <, =, ^, |, &) and it is not inside a string.

In the second case, everything is the same except for the fact that the right side of the macro is empty. These macros are useful for conditional compilation.

The definition of a macro must anticipate its use.

Embedded User C Code

Syntax:

```
#define {embedded user C code}
```

The user code may be any valid C code except for functions. It means that it can be:

- C directive,
- definition of C variable(s),
- C command(s).

Note that definitions of variables should forego commands. The purpose of this code is to define a custom interpreter. The definitions of C variables and C directives allow the user to specify the entities that can be used within instruction definitions. The commands may be used for any required initialization of the interpreter, i.e., processing of parameters.

The embedded user C code may contain these automatic variables:

<code>ip</code>	instruction pointer (<code>unsigned</code>)
<code>code1</code>	code of interpreted instruction (<code>unsigned long long</code>)
<code>force_finish</code>	assign 1 to force end of interpretation (<code>int</code>)
<code>code_to_source</code>	conversion table: ip to source line (<code>unsigned *</code>)
<code>current_line</code>	line in assembler source (<code>unsigned</code>)
<code>current_file</code>	assembler source file (<code>char *</code>)
<code>current_include_line</code>	top level file line from which the file is included (<code>unsigned</code>)
<code>argc</code>	interpreter parameters count (<code>int</code>)
<code>argv</code>	interpreter parameters (<code>char**</code>)
<code>p_stack</code>	pointer to call stack (<code>STACK *</code>)

Definitions of the following macros affect debugger commands `print` and `printout`:

```
#define { #define PRINT_MEM pointer }
#define { #define PRINT_TYPE type }
#define { #define PRINT_MIN minimum address }
#define { #define PRINT_MAX maximum address }
#define { #define PRINTIN_MEM pointer }
#define { #define PRINTIN_TYPE type }
#define { #define PRINTIN_MIN minimum address }
#define { #define PRINTIN_MAX maximum address }
#define { #define PRINTOUT_MEM pointer }
#define { #define PRINTOUT_TYPE type }
#define { #define PRINTOUT_MIN minimum address }
#define { #define PRINTOUT_MAX maximum address }
```

Commands `print`, `printin` and `printout` display contents of linear memories. The first line defines the pointer to the memory. The second line defines the type of values that are stored in the memory. The third and fourth lines define minimum and maximum addresses. The other eight lines are analogous, but they concern `printin` and `printout` commands and thereby the access to input memory and output memory.

Instruction Definitions

Syntax:

```
#define name  $\$id_1, \dots, \$id_n$  :opcode  $\$id_{i_1}[l_{i_1}][opcode_{i_1}] \dots \$id_{i_m}[l_{i_n}][opcode_{i_n}]$  [{\
instruction semantics (embedded user C code)}]
```

Please note that unproportional brackets (`[,]`) are used only to denote optional parts while proportional brackets (`(,)`) are the mandatory part of the syntax.

The above notation defines an instruction with mnemonic *name* (it must be a valid identifier). This instruction has *n* operands. Each operand has an id that must be a valid identifier, but it need not be unique in the nanoprogram. It is used only to match left-side ids to right-side ids inside definition of a single instruction; the sides are split by the semicolon (`:`). The left side describes the syntax of the new instruction. This generic instruction has the following syntax:

```
name operand1, ..., operandn
```

Figure 1.3: Example of instruction definition and machine code

Definition:

```
instruction1 $a, $b, $c : 101011 $a[2] $b[8] 00 $x[2] $c[4]
```

Machine code:

```
101011aabbbbbbbb00xxccc
```

The right side of the definition describes the machine code. It begins with an opcode (operation code). It is a string of ones and zeros, i.e., 0100 or "0100"; quotes are optional in this case. Sequence of operands eventually interleaved with the rest of opcode follows. The number (or generally arithmetic expression with numbers and macros) in brackets l_{i_j} denotes the length of the operand id_{i_j} in bits. If the id_{i_j} matches to a id_k on the left side, the operand number k is put on the position of id_{i_j} every time the instruction is compiled. Otherwise, the part of the machine code occupied by id_{i_j} is undefined. The sum of lengths of operands and opcode (all its parts) should be equal to defined instruction length (see Section 1.2.7). Example of instruction definition and its machine code is shown in Figure 1.3.

The semantics of instructions follows the same rules as embedded user C code (see previous topic in this Section) except for the fact that user defined variables are local. Most importantly, the same automatic variables and predefined functions may be utilized. The semantics is expected to contain definitions of variables and commands only.

1.2.2 #else

This is a directive for conditional compilation; **#else** may be utilized only in conjunction with **#ifdef** or **#ifndef** (see below).

1.2.3 #endif

This is a directive for conditional compilation; **#endif** may be utilized only in conjunction with **#ifdef** or **#ifndef**. The correct syntax as well as semantics are described in the specification of these directives (see below).

1.2.4 #ifdef

This is a directive for conditional compilation. Syntax:

```
#ifdef identifier  
...  
#endif
```

or

```

#ifdef identifier
...
#else
...
#endif

```

The part of nanoprogram between `#ifdef` and `#endif` (or `#else` respectively) is processed if and only if *identifier* has been previously defined. This identifier may be a macro or a label. The else branch is processed in the opposite case.

1.2.5 #ifndef

This is a directive for conditional compilation. Syntax is the following:

```

#ifndef identifier
...
#endif

```

or

```

#ifndef identifier
...
#else
...
#endif

```

The part of nanoprogram between `#ifndef` and `#endif` (or `#else` respectively) is processed if and only if *identifier* has been previously defined neither as a macro nor as a label. The else branch is processed in the opposite case.

1.2.6 #include

Syntax:

```
#include string
```

The *string* is a filename; it may contain an absolute or a relative path. The referred file must contain a nanoprogram consisting of directives only. Recursive including is possible.

The included directives have the same meaning as those that are directly in the nanoprogram. Instruction definitions are recommended to be included from an external file.

1.2.7 #instrlen

Syntax:

```
#instrlen arithmetic_expression
```

This instruction specifies the length of a single instruction in bits. It is a mandatory attribute of the instruction set and it must be defined prior to defining instructions. Macros and numbers are the allowable operands of the arithmetic expression.

1.2.8 #name

Syntax:

#name *string*

This macro specifies the name of the instruction set (or the nanoprocessor). Its use is optional.

1.2.9 #step

Syntax:

#step

If *ip* is the instruction pointer of the previous instruction, this directive inserts *ip modulo stepval* NOPs. NOP (no operation) instruction and *stepval* (see directive **#stepval** below) must be defined.

1.2.10 #stepval

Syntax:

#stepval *number*

This directive defines *stepval* value. This value is utilized by **#step** directive (see above).

1.2.11 #undef

Syntax:

#undef *identifier*

This directive allows to undefine a previously defined macro or instruction.

1.3 Instructions

Only those instructions that are defined in the nanoprogram (or in included files) may be used. The correspondence between instruction definition and instruction syntax is specified in Section 1.2.1. The syntax of a generic instruction is the following:

name *operand*₁, ..., *operand*_{*n*}

Each operand is an arithmetic expression which may contain labels and macros.

1.4 Labels

Syntax:

label:

A *label* should be a unique identifier. If a label is utilized as an instruction operand, it is expanded to pointer to the instruction that follows the label.