fischertechnik NANO Processor Project

Stefan Falk (steffalk)

Table of Contents

The fischertechnik NANO processor project

Overview

The fischertechnik NANO processor is a planned effort to construct a minimalistic computer out of standard fischertechnik components, relying on mechanics, electro-mechanics and simple digital electronics only. No computer interface (like the fischertechnik Robo interface) shall be used. Although a language, *NanoBit,* together with a compiler running on a PC, is planned, the produced machine language code will be just some black bars on printed paper. The NANO processor shall be able to execute the so compiled program directly by reading and advancing that paper code without any connection to a real computer or other microprocessor-based device.

To achieve this goal, tradeoffs will have to be made. NANO is intended to be able to execute a really minimalistic instruction set on words of 1 bit length only. As moving mechanical parts in a sequence might already be too complex to achieve directly, the machine language will be really RISC-like and contain separate instructions for moving the different single parts of the machine. Thus, even the simplest operation such as advancing a stack pointer or the like will probably consist of *several* NANO machine language statements. The machine language will be rather esoteric. Programming NANO at the machine language level might be best compared to directly writing *microcode* for a real processor.

To help planning the hardware and constructing a compiler, an assembler language gets defined containing useful, but still very simple commands. These commands will already demand an implementation that combines several of really executable primitive operations for even simple instructions as addressing the next bit in some memory. The assembly language planned is described later in this document.

To simplify programming, a higher-level language, *NanoBit,* is planned to be part of the project, including a compiler for that language, which will produce NANO machine language statements in the form of printed paper. The idea is that the NANO processor has a means of reading black bars from paper tape and directly executing them in its hardware. The NanoBit compiler will eventually even be able to perform some simple optimizations at the assembler or machine language level. For example, an instruction of moving some part of the machine in one direction, followed by an instruction of moving it in the other direction, might be superfluous. This might lead to compiled code which has a performance (if we may use this ambitious term at all) which might be hardly reached by manually creating the machine code for a given problem. Great benefits are expected from jump chain optimizations, as jumps will be relatively long-running operations, as the paper containing the program code must be moved to the jump destination.

Of course, performance will not be comparable to even the simplest ever existing real processor, as we rely on moving mechanical parts. The power of the NANO processor will be really diminutive. To give an idea of what *eventually* could be achieved, the author would be *very* happy if only the once famous game *NIM* could ever be programmed for and executed by NANO. At the time of this writing, it is, however, completely unclear if the project can ever advance to even only come near to that point.

The next steps must be:

- Find a reliable solution for storing bits and reading them without altering them.
- Implement a stack and operations for manipulating it, such as loading a bit from memory onto the stack, storing the top stack bit into memory, increasing and decreasing the stack pointer (ideally, with runtime error checking for over- and underflow).
- Find a way to represent memory addresses for the load and store operations. One idea is the implementation of one single memory pointer to a “current” memory location. The memory might consist of an array of bit memory elements and a means to move a “current” pointer mechanically via a motor and position sensors. Together with some increase/decrease pointer instructions, we could be able to implement a stack.
- Find a way to implement I/O in the form of waiting for user input and setting some form of output. Perhaps input will be just manually setting some memory locations, and output will be just displaying the content of some memory locations mechanically or using a lamp. However, if input shall be accepted by a *running* program, some form of waiting for the input to be completed will probably have to been implemented.
- Implementing the NAND operation as the only planned computation which the NANO processor can execute directly. This includes reading the top two values of the stack, passing them to a fischertechnik AND/NAND electronic module or some electro-mechanic part, popping the two operands from the stack and storing the resulting value on the stack. This might very well be divided up into several machine language instructions.
- Define the machine language at the lowest level. This includes finding an efficient way how some bit pattern on printed paper can be decoded, and what microcode-like part-instructions can be defined and implemented in the machine.
- Finding a way to implement jump instructions to addresses or some forms of labels. While the latter might be easier to implement, as an address will consist of more than one bit, a means to jump to an arbitrary program address might enable the use of more complex algorithms in programs and thus be a worthy goal.

Hardware

The NANO processor architecture

The processor consists of the following parts:

- A device capable of reading the printed black bars on paper as the only way to feed the processor with instructions. This device must also be capable of executing some form of jump instruction, that is, move the paper to some defined point in the code without executing other code.
- A memory consisting of an array of identical mechanical 1-bit memory elements. Only one bit is addressable at a time. This is achieved by a combined manipulator/reader part which can set the "current" bit to 0 or 1, read the current bit into some device (such as a relay or flip-flop), and which can be moved to the next or previous bit (this would constitute a form of pointer to a bit). Attempts to decrease the pointer below the first bit should be ignored by the hardware, as this will be needed to initialize the pointer to 0 by simply executing as many "decrease pointer" instructions as there are bits in the memory. Attempts to increase the pointer above the highest bit could be detected as an error, causing the processor to halt.
- A simple "ALU", to be described later.
- Some form of input/output – this is completely undefined up to now.
- A clock driving the processor.

The logic unit

The logic unit is used as a temporary memory and for computation of the NAND operation on two bits. It consists of the following parts:

- A two-bit shifting register made of two fischertechnik J-K-flip-flop units.
- A NAND unit whose inputs are connected to the outputs of the two shift register flip-flops. This unit steadily computes the NAND of the bits in the shifting register.
- The first flip-flop of the shifting register is also used as the (only) accumulator. The accumulator can be set to either the current value of the current bit of the memory device, or to the output of the NAND gate. The current value of the accumulator can either be fed to the shifting register (thereby shifting the former first bit into the second flip-flop, whose previous content gets lost), or it can be stored in the current bit of the memory device.

The NANO processor assembly language

Note: This is a *very* early draft. It is intended just to think about the first ideas about how the processor could really work.

Assembly language for the NANO processor will very probably not be executed directly. Some assembly language commands might very well need further reduction to an even lower-level notation, roughly comparable to microcode for existing microprocessors. As the NANO processor will not really be able to run microcode from assembly language instructions, the target object code produced by an assembler or compiler will indeed *be* that "microcode", which will be even more primitive than the assembly language statement proposed in this section.

The following commands might be implemented:

Opcode	Meaning
M+	Increase the memory pointer by 1 position. Overflow should cause the machine to halt.
M-	Decrease the memory pointer by 1 position. Underflow will be silently ignored.
M>A	Load the bit of the current address of the memory device into the accumulator.
A>M	Store the current value of the accumulator to the current address of the memory device.
A>S	Feed the current value of the accumulator into the shifting register.
N>A	Set the accumulator to the current value of the NAND gate.
0>A	Set the accumulator to 0 (FALSE).
1>A	Set the accumulator to 1 (TRUE). One of the *0>A* or *1>A* operations could be replaced by the sequence to invert the accumulator. For example *1>A* would be the same as the sequence *0>A, A>S, A>S, N>A*.
A>I	Set the *Ignore* register to the current value of the accumulator: If and only if the current value of the accumulator is TRUE, set the processor to ignore all instructions until a label is found. This is to enable branches: The instructions following the *A=0?* instruction could be a jump instruction to some address or label, followed by a label serving as the point where execution shall resume when the accumulator's current value was FALSE. Also, this instruction would be directly usable as an *IF A = 1 THEN StatementSequence END* construct as long as the *StatementSequence* does not contain labels or *Jump* instructions. The compiler would have to detect this situation and could simply append a label after the instruction sequence, causing the *Ignore* register to automatically be reset.
Halt	Halt the processor clock, that is, turn off the motor which advances the paper ROM. This may be used at the end of programs as well as for waiting for input. Inputting values would include manually setting memory bits and pressing a key to start the processor clock again.
Jump	Jump to some location. How the address or label should be provided (and found) is undefined yet, however.

Ideas for the implementation of the assembly language

As realizing the project with the existing fischertechnik equipment of the author is desired, the paramount of the processor design must be saving hardware components (fischertechnik parts). Therefore, the design has to be as simplistic and minimalistic as ever possible. This is true for the assembly language, but to an even greater extent for the implementation of the actual hardware.

What must the processor do to actually execute an instruction? At last, it must just turn on some electric signal: The pulse needed to set the accumulator register, a pulse for turning on a motor, or the like. The point is for the processor to know *which* signal must be turned on, and for *how long*. In the end, the instructions read from the paper code must drive the processor to turn some lines on, until some condition is met.

At the lowest level, the processor must be able to do the following:

- Set some polarity register for a motor:
  - A motor will control the current memory pointer and must be able to move forward and backward.
  - A motor might be used to set a memory location to a desired value (TRUE or FALSE), realized perhaps by some turning wiping device flipping some mechanics to the left or right.
  - For some instructions, only one polarity might be valid (for example to set a digital input in an electronic unit). It must be ensured by the processor or at least by the assembler or compiler that no possibly damaging polarity will be fed to electronic units.
- Select the line to turn on (and also remember which line is selected for later actually turning it on). The following lines could be turned on:
  - The motor for moving the memory address.
  - The motor for setting a memory bit content.
  - The motor for advancing the paper containing the program code.
  - Perhaps some output lights.
- Turn on the selected line in the selected polarity. This could trigger a monoflop issuing a short pulse to the relay currently selected. The so started motor could release a stop switch then, causing it to turn further until the stop switch is pressed again. Meanwhile, the monoflop would have turned off in order to not cause the motor to keep being turned on.

Perhaps polarity and line selection can be combined in a single device which can have enough distinct states as low-level signals are needed to implement the instruction set.

Also, the conditional (if) operation and the jump instruction must be implemented.

The jump instruction cannot be divided arbitrarily deep into lower-level instructions because, as the paper advances, no “next” low-level-instruction can be read. To some extent, the jump instruction must really be implemented by the hardware.

The conditional instruction could be implemented by having a “ignore all instructions” register which can be set by the *A>I* opcode if and only if the current accumulator value is FALSE, and be reset any time a label opcode is detected by the hardware.

Microcode details

The actual code that NANO will execute directly should consist of the following bits, realized each by a separate light barrier on the paper ROM:

- *Direction:* This bit determines the polarity of the signal to fire. If set (represented by a black bar), this black bar must appear slightly earlier than the trigger bit so that the polarity is set correctly when the trigger signal gets fired.
- *Select:* This bit pulses the signal in the direction determined by the *Direction* bit to the signal selection motor, which will increment or decrement the signal selector.
- *Trigger:* This bit pulses the signal in the direction determined by the *Direction* bit to the part of the CPU which is currently addressed by the signal selector.

The signal distributor must be capable of distributing the trigger signal to the following parts of the CPU:

- The memory pointer (incrementing or decrementing it).
- The actor setting the current memory bit (its direction is determined by the current value of the accumulator, so that this signal transfers the content of the accumulator to the current memory bit).
- The CP input for the second flip-flop of the ALU's shift register (setting the second flip-flop of the shift register to the contents of the first one, the accumulator).
- The CP input for the first flip-flop, which constitutes both the first shift register flip-flop and the accumulator.

Hardware implementation

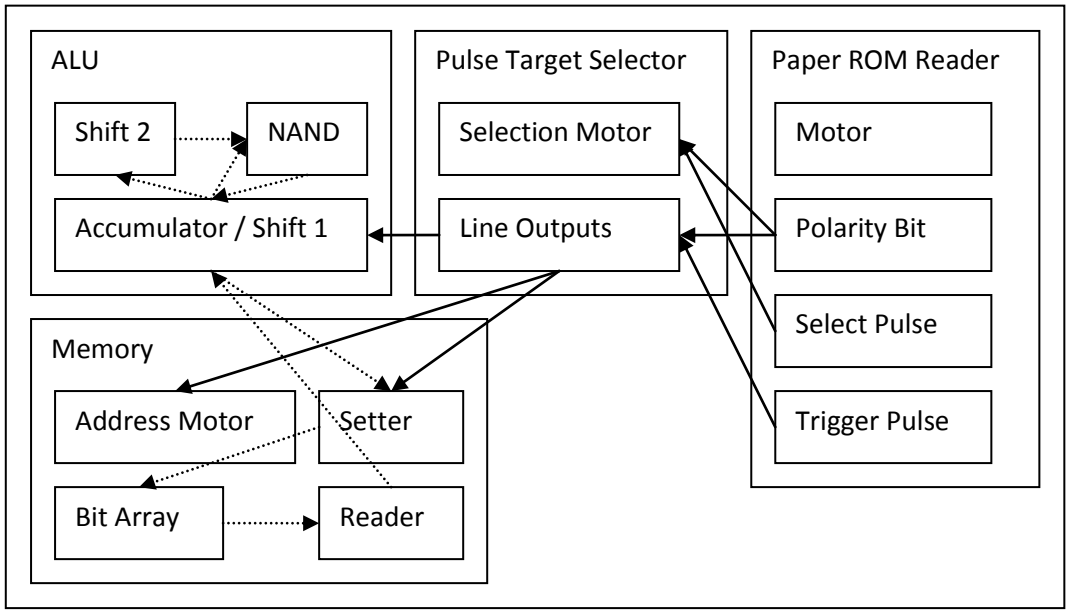
As running a program means processing low-level machine instructions, which in turn means giving pulses on several lines, here is a list of the actors which must be triggered:

- The motor forwarding the paper ROM.
- The motor moving the memory pointer.
- The motor or magnet altering the current bit in memory
- The motor managing the switches to distribute the command trigger to the different command recipients.

The following data flows will be needed and thus must be triggered by the hardware:

From	To	Purpose
Paper ROM polarity bit	Signal selector motor	Select what the next trigger instruction will do
Paper ROM polarity bit	Address selection motor	Increase/decrease the memory pointer
Paper ROM polarity bit	ALU accumulator	Set the accumulator to a constant value given in code
Accumulator	ALU shift register 2	Feed the second ALU flip-flop with a value
Accumulator	Memory setter	Store the current value of the accumulator into the current memory location
Memory reader	ALU accumulator	Read the current memory location and store its value in the accumulator
ALU NAND output	ALU accumulator	Compute the NAND of the two ALU flip-flops and store the result in the accumulator

This is a draft sketch of the hardware components of the NANO processor. Dotted lines denote data flow, normal lines denote command flow.



Currently, the realization of jump instructions is still under investigation.

The NanoBit language

Introduction

NanoBit is the high-level language planned for the fischertechnik NANO processor project. Its syntax borrows heavily from Niklaus Wirth's famous languages *Pascal* and *Modula-2*. Notable features and, above all, restrictions with regard to those languages are:

- The language is designed to be easily implemented with a single pass source code analysis phase, and with the possibility to extend the language easily later. No look-ahead on symbols is needed for compiling; only one symbol at a time is sufficient.
- Casing of keywords or variable names is irrelevant, as in Pascal, to simplify programming.
- There are two forms of source code files, namely *programs* and (library) *modules*. Only programs compile to executable code (in the form of printed paper), whereas modules serve to build a library of useful procedures which will be compiled in line with programs or modules imported in a program. Although the compiler may be invoked on a module, no code would probably be produced, but the module will be checked for the rules of the language. If a module is imported in a program or several modules which the program imports directly or indirectly, each module used in the entire program will be compiled only once, producing the executable code for it.
- The *IMPORTS* clause allows for include files with logical rather than physical names. While NanoBit is not planned to support separate compilation, the syntax is like in Modula-2, that is, you specify module names, not file names or paths. The implementation will probably choose a scheme about how a file name will be constructed from a given module name.
- For simplicity of the compiler, names of objects in imported modules might be considered global. That is, for example, no two procedures or global constants or variables in the complete program might be allowed to have the same name. The language might not offer qualifying ambiguous identifiers with the name of the module in which they are defined. However, the intention is to allow local constants and variables at the procedure level at least.
- The only data type defined so far is *BOOLEAN,* as the NANO CPU will be a 1-bit-processor. However, you must declare every variable, constant or other object having a data type. This is for clarity and to allow for future expansion of the language to eventually incorporate an *INTEGER* or *CARDINAL* (unsigned integer) data type (perhaps even with an instance-bound bit length) for which the compiler would produce the correct NANO-CPU bit-wise instructions.
- Procedures may not be nested (in contrast to Pascal and Modula-2), but passing parameters by value is supported. A later version might support return values as well. Passing parameters by reference will not be possible because there is no room to pass a memory address onto the stack.
- Recursive calls cannot be made possible because it would be necessary to store return addresses somewhere. Currently, there is no idea how this could be implemented with the available fischertechnik parts of the author.
- The *IF* statement is properly bracketed, as in Modula-2. There is no dangling-else ambiguity as in Pascal or C-like languages.
- The only unary operator defined so far is *NOT,* the only binary operators defined so far are *AND, OR, =* and *#* (meaning the test for inequality, as in Modula-2). As NAND is planned to be the only one operator that the NANO CPU will be able to execute directly, the compiler will

have to compile every expression occurring in the program into sequences of NAND and some form of load and store operations.

- Operators of arbitrary deep priorities will be supported. Again, this is to be able to complement the language with additional operators at a later time.

NanoBit syntax in EBNF

CompilationUnit ::= Program | Module.

Identifier ::= Letter {Letter | Digit}.

Program ::= "PROGRAM" Identifier ";" [Imports] Declarations "BEGIN" StatementSequence "END" ".".

Module ::= "MODULE" Identifier ";" [Imports] Declarations "END" ".".

Imports ::= "IMPORTS" Identifier {"," Identifier} ";".

Declarations ::= {ConstantList | VariableList | ProcedureDeclaration}.

ConstantList ::= "CONST" ConstantDeclaration {"," ConstantDeclaration} ";".

ConstantDeclaration ::= Identifier ":" DataType "=" Literal.

Literal ::= BooleanLiteral.

BooleanLiteral ::= "FALSE" | "TRUE".

VariableList ::= "VAR" VariableDeclaration {"," VariableDeclaration} ";".

VariableDeclaration ::= Identifier ":" DataType.

ProcedureDeclaration ::= "PROCEDURE" Identifier [FormalParameterList] ";" {ConstantList | VariableList} "BEGIN" StatementSequence "END" ";".

FormalParameterList ::= "(" FormalParameter {";" FormalParameter} ")".

FormalParameter ::= Identifier ":" DataType.

StatementSequence ::= [Statement] {";" [Statement]}.

Statement ::= Assignment | IfStatement | WhileStatement | RepeatStatement | ProcedureCall.

Assignment ::= Identifier ":=" Expression.

IfStatement ::= "IF" Expression "THEN" StatementSequence {"ELSEIF" Expression "THEN" StatementSequence} ["ELSE" StatementSequence] "END".

WhileStatement ::= "WHILE" Expression "DO" StatementSequence "END".

RepeatStatement ::= "REPEAT" StatementSequence "UNTIL" Expression.

ProcedureCall ::= Identifier [ActualParameterList].

ActualParameterList ::= "(" Expression {"," Expression} ")".

Expression ::= ExpressionWithPriority0.

ExpressionWithPriorityN ::= (ExpressionOfPriorityGreaterN {BinaryOperatorOfPriorityN ExpressionOfPriorityGreaterN}) | Factor.

BinaryOperatorOfPriority0 ::= "=" | "#";

BinaryOperatorOfPriority1 ::= "OR";

BinaryOperatorOfPriority2 ::= "AND";

Factor ::= [UnaryOperator] Element.

UnaryOperator ::= "NOT".

Element ::= ConstantIdentifier | VariableIdentifier | Literal | "(" Expression ")".

NanoBit examples

The following examples show the use of the NanoBit language and give an idea about what kind of problems might be tackled, if the project will ever succeed. As I/O is completely undefined at the time of this writing, the programs presented here so far just serve as syntax examples and will have to be complemented later.

Program 1: A one bit full adder

In this example, some variables get manipulated. As it is undefined up to now how I/O will be implemented, the operations presented here are just in-memory.

Adder1.nbp

```
program Adder1;

      procedure Add(input1: boolean, input2: boolean);

            var result, carry: boolean;

      begin
            result := (input1 and not input2) or (input2 and not input1);
            carry := input1 and input2
      end;

      var input1: boolean;

begin
      input1 := false;
      Add(input1, false);
      Add(input1, true);
      input1 := true;
      Add(input1, false);
      Add(input1, true)
end.
```

Program 2: Use of modules

This program performs the same tasks as program 1. However, the procedure used within is located in a separate module.

Mathematics.nbm

```
module Mathematics;

	procedure Add(v1: boolean, v2: boolean);

		var result, carry: boolean;

	begin
		result := (v1 and not v2) or (v2 and not v1);
		carry := v1 and v2
	end;

end.
```

Adder2.nbp

```
program Adder2;

	imports Mathematics;

	var input1: boolean;

begin
	input1 := false;
	Add(input1, false);
	Add(input1, true);
	input1 := true;
	Add(input1, false);
	Add(input1, true)
end.
```

History of this document

Version 0.1 of 2008-10-16: First draft.

Version 0.2 of 2008-10-16: Made clear that NanoBit won't support recursive procedure calls. Supplemented by-value procedure parameters. Corrected the examples provided (the variables in the module example would be declared after their first use).

Version 0.3 of 2008-10-20: Worked on the assembly language.

Version 0.4 of 2008-10-20: Fixed some typos.

Version 0.5 of 2008-11-11: Fixed year 2008 instead of 2009 in this version history.

Version 0.6 of 2008-11-16: Work in progress on the machine language.

Version 0.7 of 2008-12-05: Included a draft sketch for the hardware.