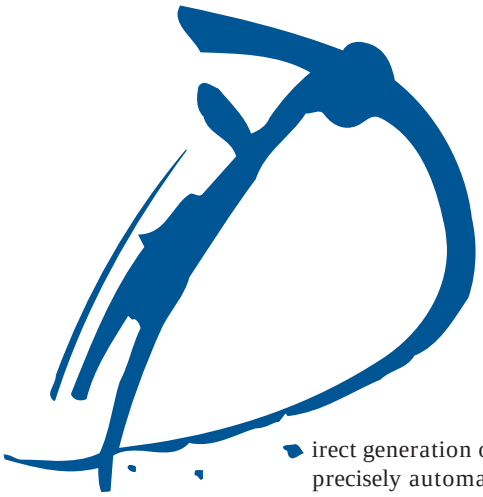
Cybersquare

Niklaus Wirth
Swiss Federal Institute of Technology,
Zurich

Hardware Compilation: Translating Programs into Circuits



Instead of separate programming languages and hardware description languages, a single language could conceivably permit us to compile parts of a program into instruction sequences for a conventional processor and other parts into circuits for programmable gate arrays.

Direct generation of hardware from a program—more precisely automatically translating a program specified in a programming language into a digital circuit—is an idea of long-standing interest. Thus far, the concept has appeared to be an uneconomical method of largely academic, but hardly practical, interest. It has therefore not been pursued with vigor and consequently has remained an idealist's dream.

Recent advances in semiconductor technology have kindled new interest in the topic, as automatic hardware translation has become possible with abundant transistor resources. In particular, the advent of programmable components—devices representing circuits that are easily and rapidly reconfigurable—has brought the idea closer to practical realization.

DISTINCT ENTITIES

Computer engineers and scientists have traditionally considered hardware and software as distinct entities with little in common in their design process. In the logic design phase, perhaps the most significant difference is that we predominantly regard programs as ordered sets of statements to be interpreted sequentially, one after the other. In contrast, we view circuits—again to simplify the matter—as sets of subcircuits operating concurrently (in parallel), with the same activities recurring in each clock cycle forever. Programs run, circuits are static.

Another reason for considering hardware and software design as different is that, in the latter case, compilation ends the design process (if we ignore debugging). In the former, compilation merely produces an abstract circuit, which hardware designers map to a physical medium. This mapping is a difficult, costly, and tedious process that must account for the physical properties of the selected parts, propagation delays, fanout, and other details.

COMMONALITIES

With the increasing use of hardware description languages, however, it has become evident that the two subjects have several traits in common. Hardware description languages let circuit specifications assume textual forms like programs, replacing traditional circuit diagrams with text. This development is analogous to the replacement of flowcharts with program text several decades ago, when increasingly complex programs began requiring flowcharts that spread over many pages.

0018-9162/98/$10.00 © 1998 IEEE

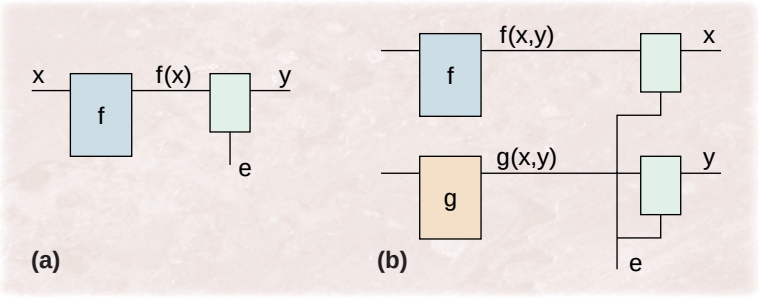


Figure 1. Circuits that implement (a) single assignment and (b) parallel composition. In parallel composition, the registers have the same enable signal.

Yet, I emphasize that common hardware description languages (HDL) describe circuits as static entities, whereas programming languages implicitly postulate a process, an activity. Hence, if our goal is to derive a circuit from a programming-language text, this derivation must include the automatic generation of a sequencing mechanism that corresponds to the program's control structure. In addition to rules for translating addition, multiplication, and other operators into circuits, we need recipes for translating if, `while`, `repeat`, and other control structures. This stands in contrast to HDLs, where the sequencing circuitry must be explicitly formulated in the source text.

In the light of the common traits I've mentioned, program variables have circuit counterparts in the form of clocked registers. The counterparts of expressions are combinational circuits (a circuit that contains neither feedback loops nor registers) of gates. The fact that programs operate mostly on numbers, whereas circuits work with binary signals, is of no further significance. We know how to represent numbers in terms of arrays of binary signals (bits) and how to implement arithmetic operations by combinational circuits.

Ian Page demonstrated that direct compilation of hardware is actually a fairly straightforward process, at least if we ignore aspects of economy (circuit simplification).[1]

I follow in his footsteps and formulate this essay in the form of a tutorial. As a result, we recognize some principal limitation of the translation process, beyond which the process may still be applicable, although unrealistic. Examining the translation process lets us more clearly perceive what is better left to software and also points out an area where hardware implementation is beneficial, even necessary: parallelism.

Hopefully, we end up with a better understanding of the several important aspects (such as structuring and modularization) that hardware and software design share, which may well be expressed in a common notation.

TRANSLATING PROGRAMS INTO CIRCUITS

Step by step, I will introduce programming concepts, each expressed by a basic construct.

Variable declarations

Typically, variables are introduced by a declaration. We use the notation

```
VAR x, y, z: Type
```

In the corresponding circuit, registers represent the variables and hold their values. The output carries the name of the variable represented by the register, and the enable signal determines when a register is to accept a new input. In this work I assume that all registers are clocked by the same clock signal—that is, all the circuits we derive are synchronous. Since clock signals are thus assumed implicit, they will not appear in the following circuit diagrams.

Assignment statements

An assignment is expressed by the statement

```
y := f(x)
```

where y is a variable, x is a set of variables, and f is an expression in x. The assignment statement corresponds to the circuit shown in Figure 1a. Function f results in a combinational circuit. Enable signal e is active when the assignment is to occur. Given time t when the register(s) have received a value, combinational circuit f yields the new value f(x) after time span pd, the *propagation delay* of f. Thus, the next clock tick must not occur before time $t + pd$. In other words, the propagation delay determines the clock frequency to be less than $1 / pd$. The synchronous-circuit concept dictates that designers choose a frequency according to the circuit f with the largest propagation delay of all function circuits.

The simplest types of variables have only two possible values, say, 0 and 1. They are said to be of type Bit (or Boolean). However, we may just as well consider composite types—variables consisting of several bits. A type Byte, for example, might consist of eight bits, and the same holds for a type Integer. The translation of arithmetic functions (addition, subtraction, comparison, and multiplication) into corresponding combinational circuits is well understood. Although the resulting circuits are of considerable complexity, they are still purely combinational circuits.

Parallel composition

Parallel composition of statements S0 and S1 is denoted by

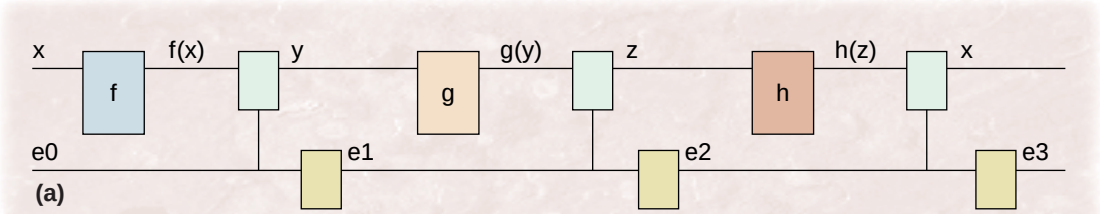
```
S0, S1
```

which means that the two component statements execute concurrently. In parallel composition, the affected registers characteristically use the same enable signal. Translation into the circuit shown in Figure 1b is straightforward.

Sequential composition

Traditionally, sequential composition of statements S0 and S1 is denoted by

```
S0;S1
```



Cycle	e0	e1	e2	e3	x	y	z
0	1	0	0	0	x0	Unspecified	Unspecified
1	0	1	0	0	x0	f(x0)	Unspecified
2	0	0	1	0	x0	f(x0)	g(f(x0))
3	0	0	0	1	h9g(f(x0)))	f(x0)	g(f(x0))

(b)

Figure 2. Circuit implementation (a) and signal values (b) for sequential composition.

Sequencing operator ";" signifies that S1 is to be executed after completion of S0, which necessitates a sequencing mechanism. The example of three assignments

```
y := f(x); z := g(y); x := h(z)
```

translates into the circuit shown in Figure 2a, with enable signals e0, e1, and e2 corresponding to the statements.

In the circuit of Figure 2a, the upper line contains the combinational circuits and registers corresponding to the assignments. The lower line contains the sequencing machinery assuring the proper sequential execution, and each statement is associated with an individual enable signal e. This signal determines when the assignment is to occur, that is, when the register holding the respective variable is to be enabled. The sequencing part is a "one-hot" state machine, which means that at any time exactly one enable signal is active (hot). Figure 2b shows the signal values before and after each clock cycle.

The preceding example is a special case in the sense that each variable is assigned only a single value: *y* is assigned f(*x*) in cycle zero, *z* is assigned g(*y*) in cycle one, and *x* is assigned h(*z*) in cycle two. The generalized case—in which different signals are fed to the same variable, although at different times—is reflected in the following short example:

```
x := f(x); x := g(x)
```

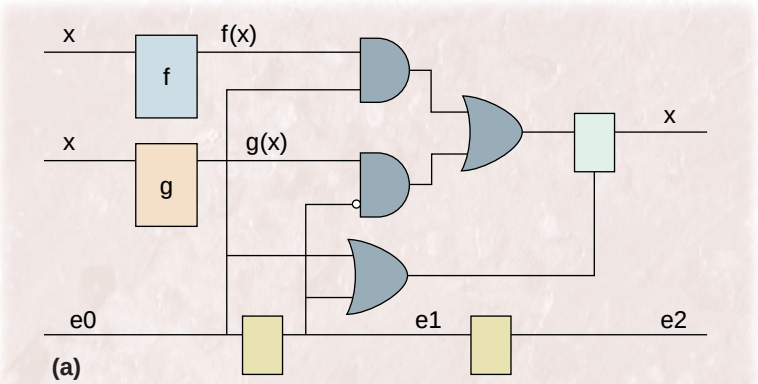
This example translates into the circuit shown in Figure 3. Since *x* receives two values, namely f(*x*) in cycle zero and g(*x*) in cycle one, a multiplexer needs to precede the register holding *x*.

Conditional composition

Conditional composition is expressed by the statement

```
IF b THEN S END
```

where *b* is a Boolean variable, and S is a statement. The circuit shown in Figure 4a is derived from this



Cycle	e0	e1	e2	x
0	1	0	0	x0
1	0	1	0	f(x0)
2	0	0	1	g(f(x0))

(b)

Figure 3. Circuit implementation (a) and table of signal values (b) for two assignments to the same variable.

state when S is *y* := f(*x*). The only difference between this circuit and that of Figure 1 is the derivation of the associated sequencing machinery that produces the enable signal for *y*.

We can now clearly see that the sequencing machinery, and it alone, directly reflects the control statements, whereas the circuit's data parts reflect assignments and expressions. We can easily generalize the conditional statement to the following form:

```
IF b0 THEN S0 ELSE S1 END
```

In this statement's corresponding circuit in Figure 4b, we only show the sequencing part with the enable signals corresponding to the various statements. At any time, at most one of the statements S is active after being triggered by e0.

Repetitive composition

We traditionally express repetitive constructs as `repeat` and `while` statements. Since only the sequencing machinery reflects a program statement's control structure, I omit showing the associated data assignment circuitry. Instead, I represent each state-

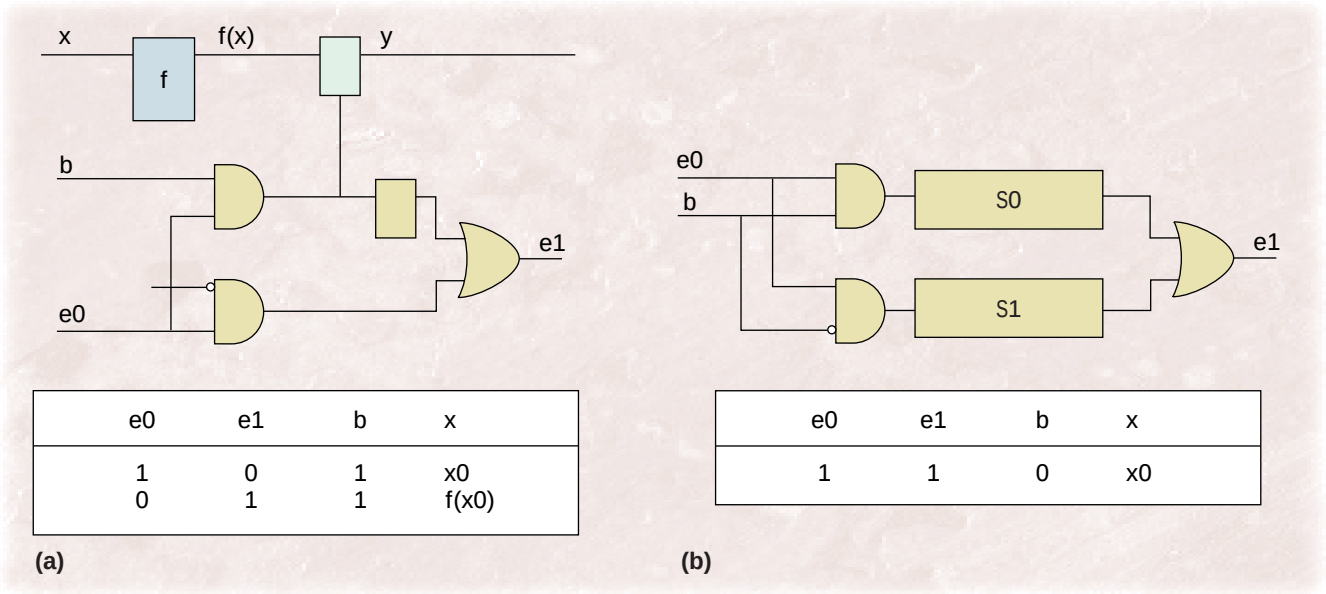


Figure 4. Conditional composition: (a) Circuit for if-then and its signal values; and (b) circuit for if-then-else and its signal values. (Only the sequencing part of this circuit is shown.)

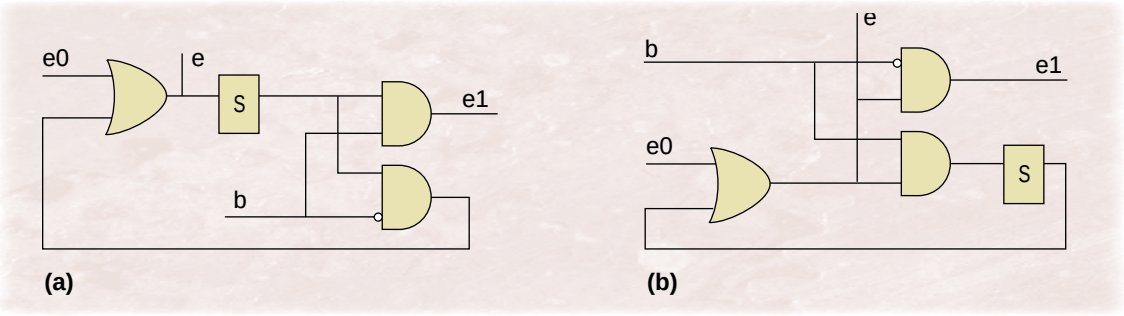


Figure 5. Circuits that implement repetitive composition: (a) repeat and (b) while.

ment by its associated enable signal alone. The examples concern two statements

```
REPEAT S UNTIL b
WHILE b DO S END
```

With *b* again standing for a Boolean variable, these statements translate into the circuits shown in Figure 5.

Selective composition

The CASE statement expresses the selection of one statement out of several:

```
CASE k OF
0 : S0 | 1 : S1 | 2 : S2 | ... | n : Sn
END
```

The core of this statement's associated sequencing circuitry, shown in Figure 6, is a decoder.

PRELIMINARY DISCUSSION

At this point, it is clear that we can transform arbitrary programs consisting of assignments, conditional, and repeated statements into circuits according to fixed rules, and can therefore do so automatically. However, we must also anticipate and fear that the resulting circuit's complexity may quickly surpass reasonable bounds.

After all, every expression occurring in a program results in an individual combinational circuit, every add symbol yields an adder, and every multiply symbol turns into a combinational multiplier with potentially hundreds or thousands of gates. Thus, for the present, the scheme I've just presented is hardly practical except for toy examples.

Cynics will remark, however, that soon the major problem will no longer be the economical use of components, but rather to find good use of the hundreds of millions of transistors on a chip.[2] Despite such warnings, I propose to examine ways to reduce the projected hardware explosion.

The best solution is to share subcircuits among various parts of a program, in a way analogous to software subroutines. We must therefore find a way to translate subroutines and subroutine calls, but emphasize that the driving motivation is to share circuits and not to reduce program text.

Therefore, we must resist simply developing a facility for declaring subroutines and textually substituting their calls by their bodies—that is, handling them

like macros. I do not deny the usefulness of such a facility—as is, for example, provided in the language Lola, where calls (that is, circuit instantiations) may even be parameterized.[3] But such expansions of the circuit for each call contributes to the transistor explosion rather than helping to avoid it.

SUBROUTINES

The circuits obtained in the examples thus far are basically state machines. If we let every subroutine translate into such a state machine, a subroutine call then corresponds to

- the suspension of the calling machine,
- the activation of the called machine, and
- upon completion, a resumption of the caller's suspended activity.

A first measure to implement subroutines is to provide every register in the sequencing part with a common enable signal, which allows us to suspend and resume a given machine. The second measure is to provide a stack (first-in, last-out store) containing the identifications of suspended machines, typically numbers from 0 to *n*. I propose the following implementation, which is to be regarded as a fixed base part of all generated circuits and is analogous to a runtime subroutine package in software. The extended circuit is a push-down machine.

The stack of machine numbers (analogous to return addresses) consists of a (static) RAM of *m* words, each of *n* bits, and an up/down counter generating the memory addresses. Each of the *n* bits in a word corresponds to a circuit representing a subroutine. Only one of these bits has the value one at a given time, thus identifying the caller to be reactivated. Hence, the (latched) read-out directly specifies the enable-signal values of the *n* circuits. The stack is operated by

- the *push* signal, which increments the counter and then writes the applied input to memory; and
- the *pop* signal, which decrements the counter.

The push signal is activated whenever a state register representing a call statement becomes active. The pop signal is activated when control reaches the last state of a circuit representing a subroutine.

An obvious enhancement would be to add an encoder and decoder at the memory's input and output. Adding encoders and decoders enlarges the number of possible subcircuits without unduly expanding memory width.

This scheme for representing subroutines so far excludes recursive procedures. The reason for this is that every subcircuit must have at most one suspension point—that is, it can have been activated only once at most. To cover recursive procedures, a scheme must permit several suspension points. This implies that the stack entries must not only identify suspended subcircuits, but also the suspension point of the associated call. We could do this in a way similar to that for storing circuit identification, either as a bit set or an encoded value. The respective state-reenable signals must then be properly qualified, further complicating the circuit. I shall not pursue this topic further.

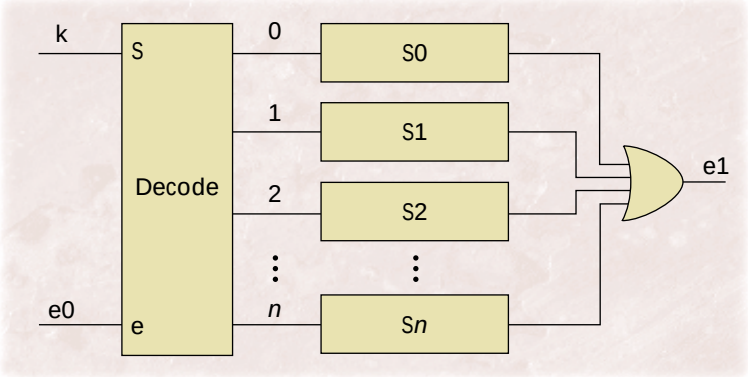


Figure 6. Circuit that implements selective composition.

A SMALL LANGUAGE

I've formulated a small, concrete programming language that integrates the preceding programming concepts. This language's syntax and semantics are essentially those of Pascal and its successor, Oberon.[4] In Figure 7, I present the syntax in extended Backus-Naur notation, with curly braces denoting repetition and brackets denoting optionality. In this language, the construct {s: *x*,*y*} denotes selection according to a Boolean value: if ~s then *x* else *y*. In the following notation

- "~" denotes a negation (not),
- "&" is a conjunction (and), and
- "#" is not equal.

```
ident = letter {letter | digit}.
integer, digit{digit}.
factor = ident | integer | "TRUE" | "FALSE" | "~" factor | "ODD" factor |
   "(" expression ")" | "{" expression ":" expression "," expression "}".
term = factor {("*" | "/" | "&") factor}.
SimpleExpression = ["+" | "–" | OR) term}.
expression = SimpleExpression {("=" | "#" | "<" | ">=" | "<=" | ">")
   Simple Expression}
assignment = ident ":=" expression {"," ident ":=" expression}.
IfStatement = "IF" expression "THEN" StatementSequence ["ELSE"
   StatementSequence] "END".
WhileStatement = "WHILE" expression "DO" StatementSequence
   "END".
Statement = [assignment | IfStatement | WhileStatement].
StatementSequence = Statement {";" Statement}.
type = "BOOLEAN" | "INTEGER".
IdentList = ident {"," ident} ":" type.
declarations = ["CONST" {IdentList ";"}] "VAR" {IdentList ";"}.
module = "MODULE" ident ";" declaration
   "BEGIN" StatementSequence "END" ident ".".
```

Figure 7. Small programming language integrating hardware and software design.

The form $x := a, y := b$ denotes concurrent assignment (sometimes written as $x,y := a,b$). Constants are not given values in the program text, but are assumed to be specified at runtime, acting as inputs.

The small sample programs in Table 1 illustrate the capabilities and limitations of the language I've presented. They also serve as test cases for studying the feasibility of the outlined translation rules.

I wrote the experimental compiler for this language, which consists of two modules, in Oberon.[4] The compiler generates a data structure—essentially a binary tree—which represents the circuit's gates and registers. The format of this data structure is defined by the Lola system, which features various tools for mapping the abstract circuit onto programmable-logic devices and field-programmable gate arrays. The main compiler module, which contains a scanner, parser, and data structure generator, is only 500 lines long and compiles into 5,200 bytes of executable code. The second module implements the generator routines for integer arithmetic. It consists of 250 lines of program code and compiles into 3,500 bytes. Both these modules are available on the Web at http://www.lola.ethz.ch and http://www.oberon.ethz.ch.

Table 1. Sample programs and resulting number of gates (including inverters) and registers in automatic circuit translation.

Program module	Data circuits: No. of registers	Data circuits: No. of gates	Sequencer: No. of registers	Sequencer: No. of gates
MODULE First; CONST a,b : BOOLEAN; VAR x,y,z : BOOLEAN; BEGIN x := a & b; y := ~a OR b; z := a # b END First.	3	7	4	1
MODULE Second; CONST a,b : INTEGER; VAR x,y,z : INTEGER; BEGIN x := a + b, y := a - b, z := a * b END Second.	24	282	2	1
MODULE MinMax; CONST a,b : INTEGER; VAR min,max : INTEGER; BEGIN IF a<b THEN min := a, max := b ELSE min := b, max := a END END MinMax.	16	125	3	5
MODULE Log; CONST a,b : INTEGER; VAR x,y : INTEGER; BEGIN x := 0; y := a; WHILE y # 0 DO x := x + 1, y := y / 2 END END Log.	16	105	4	5
MODULE Multiply; CONST a,b : INTEGER VAR x,y,z,n : INTEGER BEGIN n := 8; x := a; y := b; z := 0; WHILE n # 0 DO IF ODD x THEN z := z + y END; x := x / 2, y := y * 2, n := n - 1 END END Multiply.	32	231	4	9

It is now evident that subroutines, and more so procedures, introduce a significant degree of complexity into a circuit. It is indeed highly questionable whether it is worthwhile considering their implementation in hardware. It comes as no surprise that state machines that implement assignments, and conditional and repetitive statements—but not subroutines—play such a dominant role in sequential circuit design.

In automatically generating circuits, our principal concern is to make optimal use of the implemented facilities. Optimal use is achieved if most of the subcircuits yield values that contribute to the process most of the time. The simplest way to achieve this goal is to introduce as few sequential steps as possible. Making optimal use of facilities is also an underlying principle in instruction set architectures, in which each instruction performs (almost) the same steps. These steps are typically the subcycles of an instruction interpretation (instruction fetch, address computation, and instruction execution).

Hardware acts as what the software field knows as an interpretive system. The strength of hardware lies in the potential for subcircuits to operate concurrently. Although this is also a topic in software design, we must be aware that genuine concurrency is only possible if we have concurrently operating circuits to support the software concept. Thus much of the work on parallel computing in software actually ends up implementing only quasiconcurrency—pretended concurrent execution—conveniently hiding the underlying sequentiality.

This leads me to contend that any scheme of direct hardware compilation may well omit the concept of subroutines, but must include the facility to specify concurrent, parallel statements. Such a hardware programming language may indeed be the best way to let programmers specify parallel statements, which we call fine-grained parallelism.

Coarse-grained concurrency may well be left to conventional programming languages, where parallel processes interact infrequently, and where such processes are generated and deleted at arbitrary but

distant intervals. Such a claim is amply supported by the fact that introducing fine-grained parallelism has been left to compilers (an "under-the-hood" implementation). This is because compilers may be tuned to particular architectures, and may therefore take advantage of their target computer's specific characteristics and resources.

In this light, the consideration of a common language for hardware and software specification has a certain merit. It may also reveal the inherent difference in the designers' goals. As Chuck Thacker expressed succinctly, "Programming [software] is about finding the best sequential algorithm to solve a problem, and implementing it efficiently. The hardware designer, on the other hand, tries to bring as much parallelism to bear on the problem as possible, in order to improve performance." In other words, a good circuit is one where most gates contribute to the result in every clock cycle. For hardware designs, exploiting parallelism is not an optional luxury but a necessity.

Hardware compilation has gained interest in practice primarily because of the recent advent of large-scale programmable devices. These devices can be configured on the fly, and hence be used to directly represent circuits generated through a hardware compiler. It is therefore quite conceivable that parts of a program could be compiled into instruction sequences for a conventional processor and other parts could be compiled into circuits to be loaded onto programmable gate arrays. Although specified in the same language, the engineer will observe different criteria for good design in the two areas. ❖

References

1. I. Page, "Constructing Hardware-Software Systems from a Single Description," *J. VLSI Signal Processing*, Dec. 1996, pp. 87-107.
2. Y.N. Patt et al., "One Billion Transistors, One Uniprocessor, One Chip," *Computer*, Sept. 1997, pp. 51-57.
3. N. Wirth, *Digital Circuit Design*, Springer-Verlag, New York, 1995.
4. M. Reiser and N. Wirth, *Programming in Oberon: Steps beyond Pascal and Modula*, Addison-Wesley, ACM Press, 1992.

Niklaus Wirth is a professor at the Swiss Federal Institute of Technology (ETH), Zurich, and is best known for developing Pascal. He received the IEEE Computer Pioneer award in 1988 and the ACM Turing award in 1984. Contact him at wirth@inf.ethz.ch.