Hardware compilation
the translation of programs into circuits

Eidgenössische
Technische Hochschule
Zürich

Departement Informatik
Institut für
Computersysteme

Niklaus Wirth

Hardware Compilation

The Translation of Programs into Circuits

January 1998

286

Hardware Compilation: The Translation of Programs into Circuits

N. Wirth, 6.12.97

Abstract

We explain how programs specified in a sequential programming language can be translated automatically into a digital circuit. The possibility to specify which parts of a program are to be compiled into instruction sequences for a conventional processor, and which ones are to be translated into customized circuits has gained relevance with the advent of large programmable gate arrays (FPGA). They open the door to introduce massive, fine–grained parallelism. In order to demonstrate the feasibility of this approach, we present a tiny programming language featuring the basic programming and circuit design facilities.

1. Introduction

The direct generation of hardware from a program – more precisely: the automatic translation of a program specified in a programming language into a digital circuit – has been a topic of interest for a long time. So far it appeared to be a rather uneconomical method of largely academic but hardly practical interest. It has therefore not been pursued with vigour and consequently remained an idealist's dream. It is only through the recent advances of semiconductor technology that new interest in the topic has re–emerged, as it has become possible to be rather wasteful with abundantly available 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 by a large measure.

Hardware and software have traditionally been considered as distinct entities with little in common in their design process. Focussing on the logical design phase, perhaps the most significant difference is that programs are predominantly regarded as ordered sets of statements to be interpreted *sequentially,* one after the other, whereas circuits – again simplifying the matter – are viewed as sets of subcircuits operating *concurrently* ("in parallel"), the same activities reoccurring in each clock cycle forever. Programs "run", circuits are static.

Another reason for considering hard– and software design as different is that in the latter case compilation ends the design process (if one is willing to ignore "debugging"), whereas in the former compilation merely produces an abstract circuit. This is to be followed by the difficult, costly, and tedious phase of mapping the abstract circuit onto a physical medium, taking into account the physical properties of the selected parts, propagation delays, fanout, and other details.

With the progressing introduction of hardware description languages, however, it has become evident that the two subjects have several traits in common. Hardware description languages let specifications of circuits assume textual form like programs, replacing the traditional circuit diagrams by texts. This developent may be regarded as the analogy of the replacement of flowcharts by program texts several decades ago. Its main promoter has been the rapidly growing complexity of programs, ehibiting the limitations of diagrams spreading over many pages.

In the light of textual specifications, variables in programs have their counterparts in circuits in the form of clocked registers. The counterparts of expressions are combinational circuits of gates. The fact that programs mostly operate on numbers, whereas circuits feature binary signals, is of no further significance. It is well understood how to represent numbers in terms of arrays of binary signals (bits), and how to implement arithmetic operations by combinational circuits.

In a recent paper, I. Page [1] has demonstrated that direct compilation of hardware is actually fairly straight–forward, at least if one is willing to ignore aspects of economy (circuit simplification). We 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 it may still be applicable, but not be realistic. Thereby we perceive more clearly what should better be left to software. We also recognize an area where implementation by hardware is beneficial – even necessary: parallelism. Hopefully we end up in obtaining a better understanding of the fact that hardware and software design share several important aspects, such as structuring and modularization, that may well be expressed in a common notation.

Subsequently we will proceed step by step, in each step introducing another programming concept expressed by a programming language construct.

2. Variable Declarations

All variables are introduced by their declaration. We use the notation

VAR x, y, z: Type

In the corresponding circuit, the variables are represented by registers holding 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. We postulate that all registers be clocked by the same clock signal, that is, all derived circuits will be synchronous circuits. Therefore, clock signals will subsequently not be drawn and assumed to be implicit.

3. 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 is translated into the circuit according to Fig. 1, with the function f resulting in a combinational circuit (i.e. a circuit neither containing feedback loops nor registers). *e* stands for enable; this signal is active when the assignment is to occur.

Given a time t when the register(s) have received a value, the combinational circuit f yields the new value f(x) after a time span *pd* called the propagation delay of f. This requires the next clock tick to occur not before time *t+pd.* In other words, the propagation delay determines the clock frequency f < 1/pd. The concept of the synchronous circuit dictates, that the frequency is chosen according to the circuit f with the largest propagation delay among all function circuits.

The simplest types of variables have only 2 possible values, say 0 and 1. They are said to be of type BIT (or Boolean). However, we may just as well also consider composite types, that is, variables consisting of several bits. A type Byte, for example, might consist of 8 bits, and the same holds for a type Integer. The translation of arithmetic functions (addition, subtraction, comparison, multiplication) into corresponding combinational circuits is well understood. Although they be of considerable complexity, they are still purely combinational circuits.

4. Parallel Composition

We denote parallel composition of two statements S0 and S1 by

S0, S1

The meaning is that the two component statements are executed concurrently. The characteristic of parallel composition is that the affected registers feature the same enable signal. Translation into a circuit is straight–forward, as shown in Fig 1 (right side) for the example

x := f(x, y), y := g(x, y)

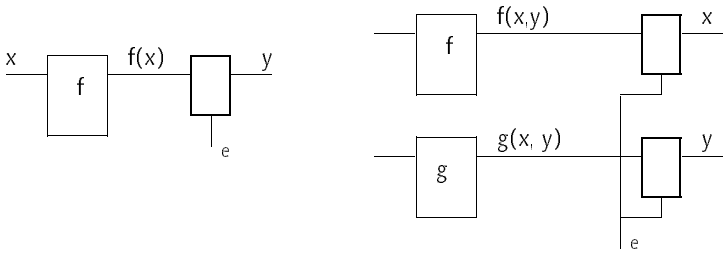

Fig. 1. Single assignment, and parallel composition

The characteristic of parallel composition is that the registers feature the same enable signal.

5. Sequential Composition

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

S0; S1

The sequencing operator ";" signifies that S1 is to be executed *after* completion of S0. This necessitates a sequencing mechanism. The example of the three assignments

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

is translated into the circuit shown if Fig. 2, with enable signals e corrsponding to the statements.

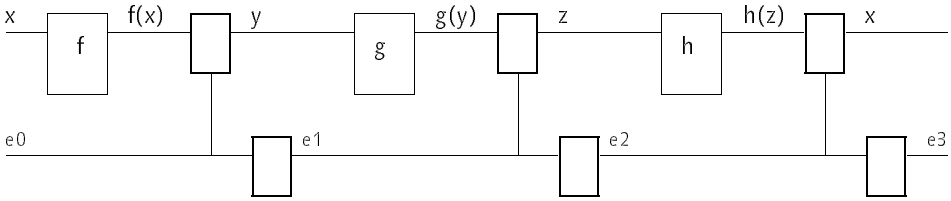

Fig. 2. Sequential composition

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 of the three assignments. With each statement is associated an individual enable signal e. It determines when the assignment is to occur, that is, when the register holding the rspective variable is to be enabled. The sequencing part is a "one–hot" state machine. The following table illustrates the signal values before and after each clock cycle.

e0	e1	e2	e3	x	y	z
1	0	0	0	x0	?	?
0	1	0	0	x0	f(x0)	?
0	0	1	0	x0	f(x0)	g(f(x0))
0	0	0	1	h(g(f(x0)))	f(x0)	g(f(x0))

The preceding example is a special case in the sense that each variable is assigned only a single value, namely y is assigned f(x) in cycle 0, z is assigned g(y) in cycle 1, and x is assigned h(z) in cycle 2. The generalized case is reflected in the following short example:

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

which is transformed into the circuit shown in Fig. 3. Since x receives two values, namely f(x) in cycle 0 and g(x) in cycle 1, the register holding x needs to be preceded by a multiplexer.

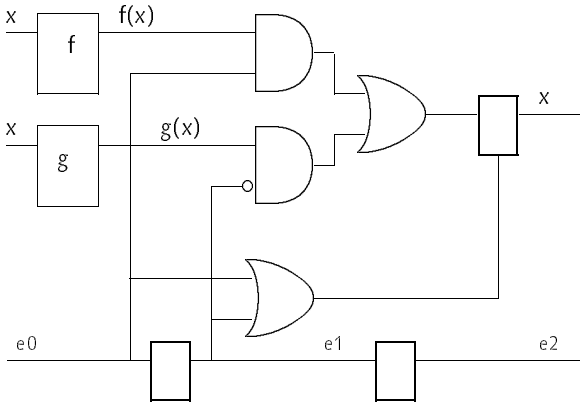

Fig. 3. Two assignments to the same variable

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

6. Conditional Composition

This is expressed by the statement

IF b THEN S END

where b is a Boolean variable and S a statement. The circuit shown in Fig. 4 on the left side is derived from it with S standing for y := f(x). The only difference to Fig. 1 is the derivation of the associated sequencing machinery yielding the enable signal for y.

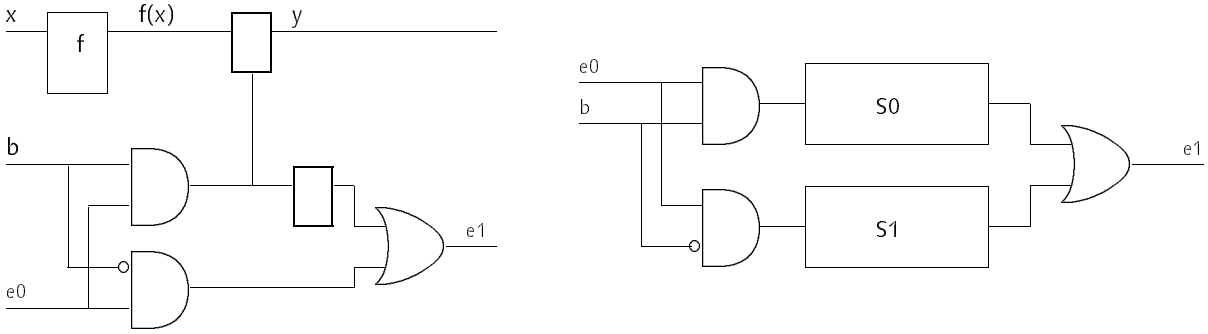

Fig. 4. Conditional composition

e0	e1	b	x
1	0	1	x0
0	1	1	f(x0)

e0	e1	b	x
1	1	0	x0

It now emerges clearly that the sequencing machinery, and it alone, directly reflects the control statements, whereas the data parts of circuits reflect assignments and expressions.

The conditional statement is easily generalized to its following form:

IF b0 THEN S0 ELSE S1 END

In its corrsponding circuit on the right side of Fig. 4 we show the seqencing part only 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.

7. Repetitive Composition

We consider repetitive constructs traditionally expressed as repeat and while statements. Since the control structure of a program statement is refelected by the sequencing machinery only, we may omit showing associated data assignments, and instead let each statement be represented by its associated enable signal only.

We consider the statements

REPEAT S UNTIL b and WHILE b DO S END

b again standing for a Boolean variable, they translate into the circuits shown in Fig. 5, where e stands for the enable signal of statement S.

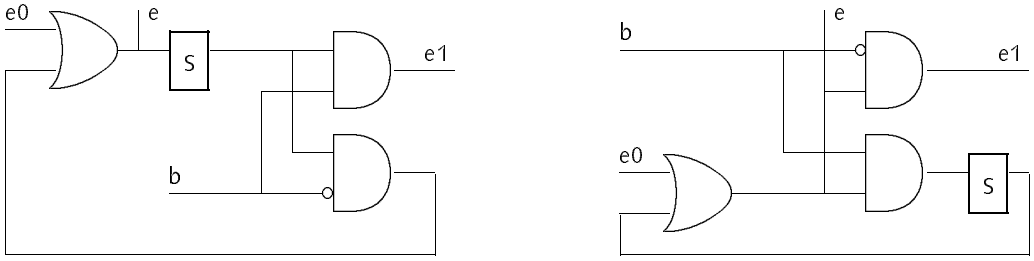

Fig. 5. Repetitive compositions

8. Selective Composition

The selection of one statement out of several is expressed by the case statement, whose corresponding sequencing circuitry containing a decoder is shown in Fig. 6.

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

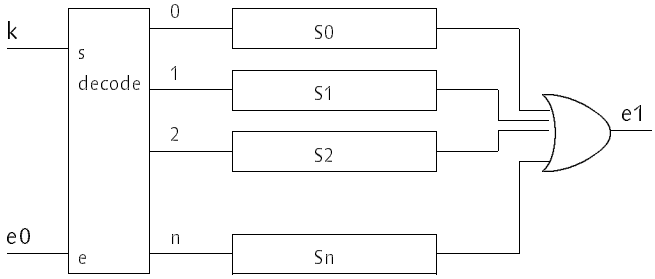

Fig. 6. Selective composition

9. Preliminary Discussion

At this point we realize that arbitrary programs consisting of assignments, conditional and repeated statements can be transformed into circuits according to fixed rules, and therefore automatically. However, it is also to be feared that the complexity of the resulting circuits 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, every multiply symbol turns into a combinational multiplier, potentially with hundreds or thousands of gates. We agree that for the present time this scheme 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 [4].

Inspite of such warnings, we propose to look at ways to reduce the projected hardware explosion. The best solution is to *share subcircuits* among various parts. This, of course, is equivalent to the idea of the subroutine in software. We must therefore find a way to translate subroutines and subroutine calls. We emphasize that the driving motivation is the sharing of circuits, and not the reduction of program text.

Therefore, a facility for declaring subroutines and textually substituting their calls by their bodies, that is, handling them like macros, must be rejected. This is not to deny the usefulness of such a feature, as it is for example provided in the language Lola, where the calls may be parametrized [2, 3]. But with its expansions of the circuit for each call it contributes to the circuit explosion rather than being a facility to avoid it.

10. Subroutines

The circuits obtained so far are basically state machines. 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 its completion, a resumption of the suspended activity of the caller.

A first measure to implement subroutines is to provide every register in the sequencing part with a common enable signal. This allows to suspend and resume a given machine. The second measure is to provide a stack (a first–in last–out store) of identifications of suspended machines, typically numbers from 0 to some limit n. We propose the following implementation, which is to be regarded as a fixed base part of all circuits generated, analogous to a run–time 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 RAM–addresses. Each of the n bits in a word corresponds to one of the circuits representing a subroutine. One of them has the value 1, identifying the circuit to be reactivated. Hence, the (latched) read–out directly specifies the values of the enable signals of the n circuits. This is shown in Fig. 7. The stack is operated by the *push* signal, incrementing the counter and thereafter writing the applied input to the RAM, and the *pop* signal, decrementing the counter.

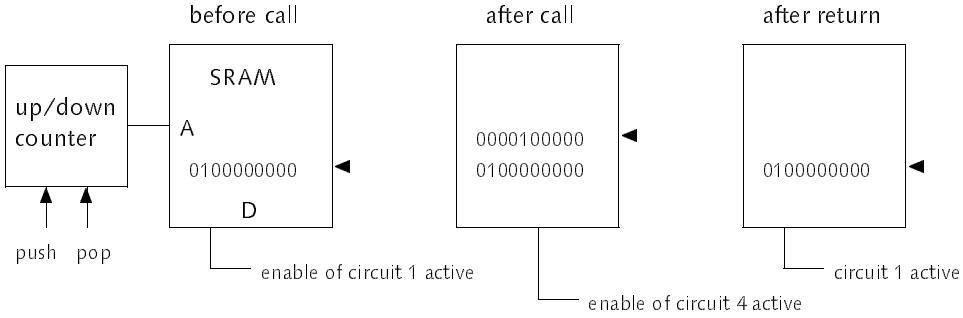

Fig. 7. Subroutine mechanism

The *push* signal is activated whenever a state register belonging to a call statement becomes active. Since a subroutine may contain several calls itself, these register outputs must be ORed. A better solution would be a bus. The *pop* signal is activated when the last state of a circuit representing a subroutine is reached. An obvious enhancement with the purpose to enlarge the number of possible subcircuits without undue expansion of the RAM's width is to place an encoder at the input and a decoder at the output of the RAM. A width of n bits then allows for 2^n subroutines.

The scheme presented so far excludes recursive procedures. The reason is that every subcircuit must have at most one suspension point, that is, it can have been activated at most once. If recursive procedures are to be included, a scheme allowing for several suspension points must be found. This implies that the stack entries must not only contain an identification of the suspended subcircuit, but also of the suspension point of the call in question. This can be achieved in a manner similar to storing the circuit identification, either as a bit set or an encoded value. The respective state re–enable signals must then be properly qualified, further complicating the circuit. We shall not pursue this topic any further.

11. A Small Language

To conclude, we formulate a small, concrete programming language that integrates the programming concepts listed before. Its syntax and semantics are essentially those of Pascal and its successor Oberon. The syntax is presented in Extended BNF [2] with curly braces denoting repetition and brackets denoting optionality.

The construct {s: x, y} denotes selection according to a Boolean valuse: if ~s then x else y. "~" denotes negation (not), "&" conjunction (and), and "#" denotes "not equal". 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 by specified at run time, acting as inputs.

```
ident = letter {letter | digit}.
integer = digit {digit}.
factor = ident | integer | "TRUE" | "FALSE" | "~" factor | "ODD" factor |
          "(" expression ")" | "{" expression ":" expression "," expression "}".
term = factor {("*" | "/" | "&") factor}.
SimpleExpression = ["+"|"–"] term {("+"| "–" | "OR") term}.
expression = SimpleExpression [("=" | "#" | "<" | ">=" | "<=" | ">") SimpleExpression].
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 ";" declarations
          "BEGIN" StatementSequence "END" ident "." .
```

The following small sample programs illustrate the capabilities and limitations of the language. They also serve as test cases allowing to study the feasibility of the outlined translation rules.

```
MODULE First;                                          22 gates
    CONST a, b: BOOLEAN;                               7 registers
    VAR x, y, z: BOOLEAN;
BEGIN x := a & b; y := ~a OR b; z := a # b
END First.

MODULE Second;                                         290 gates
    CONST a, b: INTEGER;                               26 registers
    VAR x, y, z: INTEGER;
BEGIN x := a+b, y := a–b, z := a*b
END Second.

MODULE MinMax;                                         140 gates
    CONST a, b: INTEGER;                               19 registers
    VAR min, max: INTEGER;
BEGIN
    IF a < b THEN min := a, max := b ELSE min := b, max := a END
END MinMax.

MODULE Log;                                            122 gates
    CONST a: INTEGER;                                  20 registers
    VAR x, y: INTEGER;
BEGIN x := 0; y := a;
    WHILE y # 0 DO x := x+1, y := y/2 END
END Log.

MODULE Multiply;                                       256 gates
    CONST a, b: INTEGER;                               36 registers
    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.
```

The experimental compiler for this language was written in *Oberon* and consists of two modules. It generates a data structure, essentially a binary tree, representing the gates and registers of the circuit. This structure is the basis of the *Lola System* [4], which contains various tools for mapping the abstract circuit onto PLDs and FPGAs. The main compiler module, containing scanner, parser, and generator of the data structure, is 500 lines long and compiles into 5200 bytes of code. The second module implements the generator routines for integer arithmetic. It consists of 250 lines of program text, and compiles into 3500 bytes of code.

12. Conclusions

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 worth while considering their implementation in hardware. It comes as no surprise that state machines, implemeting assignments, conditional and repetitive statements, but not subroutines, play such a dominant role in squential circuit design. The principal concern in this area is to make optimal use of the implemented facilities. This is achieved if most of the subcircuits yield values contributing to the process most of the time. The simplest way to achieve this goal is to introduce as few sequential steps as possible. It is the underlying principle in computer architectures, where (almost) the same steps are performed for each instruction. The steps are typically the subcycles of an instruction interpretation. Hardware acts as what is known in software as an interpretive system.

The strength of hardware is the possibility of *concurrent* operation of subcircuits. This is also a topic in software design. But we must be aware of the fact that concurrency is only possible by having concurrently operating circuits to support this concept. Much work on parallel computing in software actually ends up in implementing quasi–concurrency, that is, in pretending concurrent execution, and, in fact, in hiding the underlying sequentiality. This leads us to contend that any scheme of direct hardware compilation may well omit the concept of subroutines, but *must* include the facility of specifying concurrent, parallel statements.

Such a hardware programming language may indeed be the best way to let the programmer specify parallel statements. We call this *fine–grained* parallelism. Coarse–grained concurrency may well be left to conventional programming languages, where parallel processes interact infrequently, and where they are generated and deleted at arbitrary but distant intervals. This claim is amply supported by the fact that fine–grained parallelism has mostly been left to be introduced by compilers, under the hood, so to say. The reason for this is that compilers may be tuned to particular architectures, and they can therefore make use of their target computer's specific characteristics.

In this light, the consideration of a common language for hard– and software specification has a certain merit. It may reveal the inherent difference in the designers' goals. As C. Thacker expressed it 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. Exploiting parallelism is not an optional luxury, but a necessity.

We end by repeating that hardware compilation has gained interest in practice primarily because of the recent advent of large–scale programmable devices. These 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 are compiled into instruction sequences for a conventional processor, and other parts 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. *Journal of VLSI Signal Processing 12*, 87–107 (1996).
2. N. Wirth. *Digital Circuit Design.* Springer–Verlag, Heidelberg, 1995.
3. www.Lola.ethz.ch
4. Y. N. Patt, et al. One Billion Transistors, One Uniprocessor, One Chip. Computer, 30, 9 (Sept. 1997) 51–57.