

# Finite State Machines, Programmable Logic Devices, and the Crux of Growing Complexity

Niklaus Wirth, 20. 9. 2008

## 1. Introduction

This essay covers many facets. We start with a brief explanation of the concept of finite state machine (FSM) and its implementation with programmed logic devices (PLD). Then we present a widely used, although outdated sample of the species, the GAL22V10 (1993), which represents the FSM in almost pure fashion. We investigate how circuits, that is, configurations for the PLD are specified. In turn we define a small formalism, a language specific for this purpose. Thereby we show how small languages may easily be defined and implemented for special tasks like the one at hand.

Its implementation, a sort of compiler, translates the specification into a configuration and finally into a bit stream to be loaded (programmed) into the chip. We show that this task is, albeit not trivial, quite straight-forward, resulting in a compiler not exceeding a few pages of source program. It functions as a single-pass compiler, that is, without intermediate data structure or intermediate code, as is commonly deemed indispensable.

Then we consider a successor of the GAL chip, a PLD of "the next generation", making use of the wider possibilities offered by the progress of semiconductor technology at the time. We encounter the phenomenon that a more powerful technology lures designers into adding features and facilities that had to be omitted when circumstances were tighter. In the drive to design more powerful and more flexible devices, unexpected complexity creeps in. In this case, it may be unobtrusive, as long as one considers the design itself. It becomes apparent if one tries to use the device, mainly if one wishes to make full use of the gates and registers of the PLD, and wants to achieve an optimal design.

The modern trend is to hide these difficulties and to let the compiler take care of them. The enormous power of modern workstations allows to build compilers of tremendous complexity which try many configurations and placements of gates. They make use of highly complex search algorithms based on heuristics and backtracking. It is an activity where computers are much superior to human designers.

Here we show how one or two innocent looking design decisions of the PLD architects have profound implications on the configuring compiler, increasing its complexity dramatically, sometimes by orders of magnitude. One may guess that a better team work between hardware architects and software engineers might have curtailed this complexification considerably. Even if computing power is available in abundance, it is still a good principle to avoid avoidable complexity, or, in other words, to make it as simple as possible. After all, the ability to find simpler solutions is the hallmark of a competent engineer.

## 2. The Concept of Finite State Machine

A FSM is a conceptual device that at any time assumes one of  $n$  states. It moves from state to state in discreet steps. The state after a transition is determined by (1) the state  $S_i$  and (2) the value of input  $x_i$  before the transition. Thus, the machine is defined by a state transition function  $f$

$$S_{i+1} = f(S_i, x_i) \text{ for } i = 0, 1, \dots$$

and by an initial state  $S_0$ . Typically, a second function  $g$  is associated with the machine which defines an output  $y$ .

$$y_i = g(S_i, x_i)$$

The significance of this concept is that it is fundamental to all computers. All computers are basically FSMs, albeit with so many possible states that the FSM concept is no longer helpful in defining their organization. However, their core, and many components, are identifiable as FSMs.

### 3. The Architecture of the GAL22V10

The earliest versions of integrated circuits (chips) contained one or several gates or registers. Soon followed standard, frequently needed components such as multiplexors, decoders, adders, sets of registers. The FSM would have been an obvious next choice, had it not been for the fact that all these chips contained a fixed digital circuit. The FSM, however, stood for a class of circuits with individual state and output functions.

The realization of a general FSM became possible through the innovative concept of the *Programmable Logic Device* (PLD). The GAL22V10 chip to be introduced in detail here was one of the earliest representatives of the class of PLDs (1990). The core idea rests on the fact that any logic function ( $z$ ), independent of the number of arguments, can be transformed into a single sum of terms ( $y_i$ ) of the arguments ( $x_i$ ) and their negations ( $x'_i$ ). This is called the *normal form* of the function. Expressions are sequences of terms connected by or-operators.

$$Z = y_0 + y_1 + \dots + y_m$$

Terms are sequences of variables (or their negation) connected by and-operators.

$$y_i = x_0 * x_1 * \dots * x_n$$

In a PLD, the variables occurring in every term can be freely selected or omitted by *programming*. The selection is defined by values stored in the *configuration memory*. Loading this memory is called *programming* the PLD. The state of the memory thus defines the logic function(s), and it is not changed during the use of the circuit. This memory must be considered as a matrix *FM* of bits, where each bit determines, whether a variable is present or absent in a term.  $FM[i, j]$  means "variable  $i$  is a factor in term  $j$ ". (Typically, even  $i$  denote the variable numbered  $i/2$ , and odd  $i$  denote the negation of variable with number  $i/2$ ). The structure of a PLD is shown in Fig. 1. The structure of each horizontal line (term) is shown in Fig. 2.

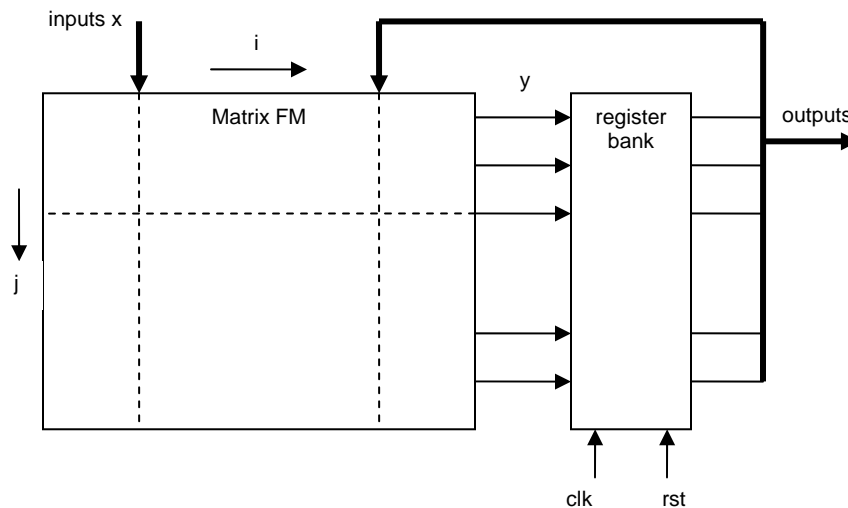


Fig. 1. Schematic of a PLD

Notably, the values of the registers (the state) are also available as arguments of the transition function  $f$  represented by the matrix FM. The GAL chip features 10 registers (and outputs) and 12 inputs. (In fact, registers can be disconnected from their associated output pins. In this case the pin can be used as an additional input pin. Hence, the GAL may be used to implement functions with up to 21 arguments).

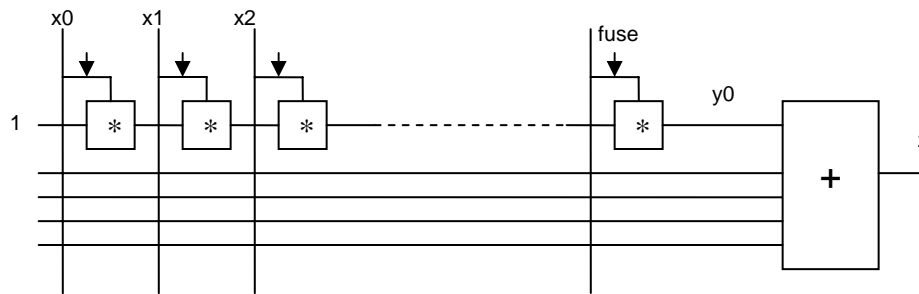


Fig. 2. Circuit representing an expression

Incidentally, we call the matrix  $FM$  for *fuse map*. In early versions, the matrix was programmable only once by burning connections that were looked at like fuses. Modern PLDs are programmable many times.

#### 4. A Language for Specifying FSMs

We now turn our attention to the problem of how to specify a specific FSM. As mentioned already, the FSM is determined by its transition function (and possibly an output function) and an initial state. Our task then is to define a language, a formalism, allowing to express logical functions. We will restrict this to express them in their normal form. Taking into account the possibilities offered by the GAL chip, we allow outputs to include a register, and to possibly be negated. The simple language is specified in terms of EBNF as follows:

```

program =      "MODULE" identifier ";"
              ["IN" declaration ";"] ["OUT" declaration ";"]
              "BEGIN" ["RST" term ";"] assignment {";" assignment} "END" identifier.

declaration =  identifier {";" identifier}.

assignment =   identifier ":@" ["~"] ["REG"] expression | "0" | "1".

expression =   term {"+" term}.

term =         factor {"*" factor}.

factor =       identifier ["' '"].

```

Declarations allow variables to be given arbitrary names, and to give them a type, either input or output. The compiler can thus check, whether an output is defined (associated with an expression), and whether an input is inadvertently assigned a value. There remains the problem of associating variables with pins in the case of using the GAL chip. We here solved this problem in a very simple way by assigning the numbers 0, 1, ..., 9 to outputs, and 11, 12, ..., 21 to inputs in the sequence of their declaration. (Note that variable 10 is used as clock signal, and should normally not be used as an input, although the GAL permits this). Some examples of assignments are:

```

x := 1
x := a*b' + a'*b
x := REG a
x := ~REG a*b' + a'*b

```

Note that these assignments are not statements executed in sequence, but are static equations. A dynamic behaviour stems only from registers: At every tick of the (implicit) clock the value of their argument is copied to their output (and the input presumably assumes a new value). We assume that all registers are driven by the same clock. Such circuits are called *synchronous*. The following are short examples of an 8-bit binary and a 2-digit decimal counter.

```

MODULE Counter;
  IN ci;
  OUT q0, q1, q2, q3, q4, q5, q6, q7, co;
BEGIN

```

```

q0 := REG(q0*ci' + q0*ci);
q1 := REG(q1*q0' + q1*ci' + q1*q0*ci);
q2 := REG(q2*q1' + q2*q0' + q2*ci' + q2*q1*q0*ci);
q3 := REG(q3*q2' + q3*q1' + q3*q0' + q3*ci' + q3*q2*q1*q0*ci);
q4 := REG(q4*q3' + q4*q2' + q4*q1' + q4*q0' + q4*ci' + q4*q3*q2*q1*q0*ci);
q5 := REG(q5*q4' + q5*q3' + q5*q2' + q5*q1' + q5*q0' + q5*ci'
+ q5*q4*q3*q2*q1*q0*ci);
q6 := REG(q6*q5' + q6*q4' + q6*q3' + q6*q2' + q6*q1' + q6*q0' + q6*ci'
+ q6*q5*q4*q3*q2*q1*q0*ci);
q7 := REG(q7*q6' + q7*q5' + q7*q4' + q7*q3' + q7*q2' + q7*q1' + q7*q0' + q7*ci'
+ q7*q6*q5*q4*q3*q2*q1*q0*ci);
co := q7*q6*q5*q4*q3*q2*q1*q0*ci
END Counter.

MODULE DecCounter;
  IN ci: BIT;
  OUT q0, q1, q2, q3, q4, q5, q6, q7, co, ch: BIT;
BEGIN
  q0 := REG(q0*ci' + q0*ci);
  q1 := REG(q1*q0' + q1*ci' + q3*q1*q0*ci);
  q2 := REG(q2*q1' + q2*q0' + q2*ci' + q2*q1*q0*ci);
  q3 := REG(q3*q0' + q3*ci' + q3*q2*q1*q0*ci);
  ch := q3*q0*ci;
  q4 := REG(q4*ch' + q4*ch);
  q5 := REG(q5*q4' + q5*ch' + q7*q5*q4*ch);
  q6 := REG(q6*q5' + q6*q4' + q6*ch' + q6*q5*q4*ch);
  q7 := REG(q7*q4' + q7*ch' + q7*q6*q5*q4*ch);
  co := q7*q4*ch
END DecCounter.

```

The initial state of both counters is zero, i.e. all registers are reset to 0. Our third example is a 4-bit shifter, rotating a 1-bit. Evidently we do not wish to start with all registers having value 0. But the language (nor the GAL) offers any alternative! The following program shows how to solve this situation.

```

MODULE Shifter;
  OUT s0, s1, s2, s3;
BEGIN s0 := ~REG s1';
  s1 := REG s2; s2 := REG s3; s3 := REG s0
END Shifter.

```

We emphasise that we restrict this small language to the essential. Intentionally, we omit all kinds of facilities that might be useful and convenient in practice, that are even conventional. There are no features for abbreviations, macros, for replication and parametrization. They would merely distract from the issue at hand, the specification of simple FSMs and their compilation into configuration streams for PLDs.

## 5. A Compiler for the FSM-Language

A simple specification raises the expectation for a simple implementation. Indeed, the entire compiler is described in a source text of less than 10 pages. The most noteworthy property is that the result, the configuration expressed in terms of the fuse map, is computed in a single pass over the source text.

The compiler follows the principles outlined in [1]. The text is scanned by a simple recursive descent parser, supported by a scanner identifying symbols of the syntax. Processing declarations results in a list of the declared variables, each entry being of type *Variable*. Each variable is given a type and a number. *inv* and *reg* specify whether or not the signal, and with it the PLD cell, contains a register and/or an inverter.

```

TYPE Variable = POINTER TO RECORD
  typ, num: INTEGER;
  inv, reg, def: BOOLEAN;
  next: Variable;
  name: ARRAY 32 OF CHAR
END ;

```

A recursive descent parser consists of a set of procedures, one corresponding to every non-terminal syntax symbol. They are augmented by statements corresponding to the semantics of the symbol, in this case by statements affecting the fuse map. It is declared as a matrix of Boolean variables, each denoting a connection (an and-gate). The value TRUE means "connection is open" (or "factor not present"). The entire matrix is initialized with TRUE. The fuse matrix for the GAL consists of 44 columns (there are 22 signals), and each column has 132 terms. It is to be noted that not all of the 10 sums (outputs) have the same number of terms (horizontal lines). The index of the first term of sum (cell)  $k$  is  $row[k]$ . The index of the column corresponding to signal  $i$  is  $col[i]$ . The mapping arrays  $row$  and  $col$  are constant.

```
CONST M = 22; (*nof signals*)
      N = 132; (*nof product terms in and-matrix*)
      R = 10; (*nof registers and of output signals*)

TYPE Column = ARRAY N OF SHORTINT; (*0 = "connection closed"*)

VAR sym: INTEGER; (*last symbol read by scanner*)
    idlist, guard, clk: Variable;
    row: ARRAY R+1 OF INTEGER; (*map from signal number to row index*)
    col: ARRAY M OF INTEGER; (*map from signal number to column index*)

    S0, S1: ARRAY R OF SHORTINT; (* arch rows *)
    FM: ARRAY 2*M OF Column; (*fuse map*)
```

Procedure *Assignment* identifies the variable  $v$  and then evaluates the expression of the defining value. The expression may be prefixed by a *not* or a *reg* symbol, denoting inversion and the presence of a register respectively. These properties are recorded in the record of variable  $v$ .

```
PROCEDURE assignment;
  VAR v: Variable; n: INTEGER;
BEGIN
  IF sym = PLDS.ident THEN
    v := this(); n := v.num;
    IF n >= R THEN PLDS.Mark("this an input"); n := 0 END ;
    IF sym = PLDS.eql THEN PLDS.Get(sym) ELSE PLDS.Mark(":= expected") END ;
    IF v.def THEN PLDS.Mark("multiple assignment") ELSE v.def := TRUE END ;
    IF sym = PLDS.int THEN
      IF PLDS.val = 0 THEN S0[n] := 0 ELSIF PLDS.val # 1 THEN PLDS.Mark("bad value") END ;
      PLDS.Get(sym)
    ELSE
      IF sym = PLDS.not THEN PLDS.Get(sym) ELSE S0[n] := 1 END ;
      IF sym = PLDS.reg THEN PLDS.Get(sym); S1[n] := 0 END ;
      expression(row[n] + 1, row[n+1])
    END
  END
END
END assignment;
```

We note the two parameters in the call of *expression*. The first denotes the index of the first matrix row (term) associated with the sum assigned to variable  $v$ , the second the index of its last row. Only rows in this range may be affected by the call of *expression*, which calls procedure *term* for each term in the expression.

```
PROCEDURE expression(n, limit: INTEGER);
BEGIN term(n); INC(n);
  WHILE sym = PLDS.or DO
    PLDS.Get(sym); term(n);
    IF n < limit THEN INC(n) ELSE PLDS.Mark("too many terms") END
  END
END
END expression;
```

Procedure *term* in turn calls on *factor* for each factor in the term.

```
PROCEDURE term(j: INTEGER);
BEGIN factor(n);
  WHILE sym = PLDS.and DO PLDS.Get(sym); factor(j) END
END term;
```

A factor is a variable in true or inverted form. Procedure *factor* finally establishes the appropriate connections in the fuse map. The column index is given by the variable's number  $v.num$ , incremented by 1 in case of inversion, and the row index is given by the parameter  $n$ .

```

PROCEDURE factor(j: INTEGER);
  VAR v: Variable; inv: INTEGER;
BEGIN v := this();
  IF sym = PLDS.inv THEN PLDS.Get(sym); inv := 1 ELSE inv := 0 END ;
  IF v # NIL THEN FM[col[v.num]+inv, j] := 0 END
END factor;

```

### Programming the GAL

After the fuse map has been computed, it is loaded into the GAL's configuration memory, which is an EEPROM. This is done using a standard protocol (JTAG) with 4 wires. The GAL chip constitutes a simple FSM for loading the data, the so-called *bit-stream*, serially over the line. The 4 signals of the JTAG protocol, mapped onto the PC's parallel port bits, are:

TDI	out D0	data input
TMS	out D1	mode
TCK	out D2	clock
TDO	in D5	data output (used for verifying the loaded configuration)

The GAL's loader FSM has three states: *idle*, *shift*, *execute*, as shown in Fig. 3. State transitions occur if the clocked mode signal is 1. 5-bit instruction codes are shifted into the device in the shift state, and they control which operation is executed in the subsequent execute mode.

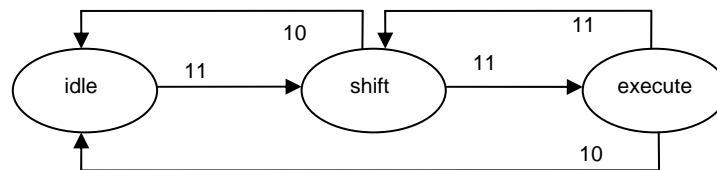


Fig. 3. State diagram of GAL's loader FSM

The relevant commands are: *Shift data*, *erase*, *program*, *verify*, and *shift architecture*. The first shifts 132 data bits plus 6 column address bits into the data register, from where they are moved into the addressed fuse map column by the *program* instruction. The shift architecture command shifts 20 bits – S0, S1 for each cell – into the device's architecture register. The resulting loader program is reasonably straight forward.

### Remarks

The first remark concerns the global signal for resetting all registers. According to the GAL specifications, it is generated by a single term. In our language it is represented by an optional statement *RST term* in the heading of the program.

There are very few facilities of the GAL which we have omitted. One of them is the global preset signal. Another is the possibility to use tri-state gates for the outputs controlled by signals generated in the fuse map. In our scheme, all signals are either inputs or outputs at all times.

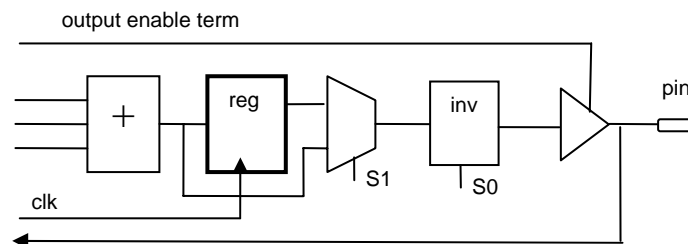


Fig. 4. GAL-cell with register, inverter, and tri-state output

## 6. The Architecture of the MACH-4 64-32 PLD

As progress in semiconductor technology continued, feature sizes shrank and an increasing number of transistors could be placed on a chip. Hence bigger and bigger PLDs became available. We choose AMD's MACH-4 version (1996) as an example for further investigations.

"Bigger" obviously means more signals and more registers. In this case, their number is 64. To be more specific, there are 32 cells with a pin, each containing two or-gates and two registers, but only one of them feeding into a pin. From each cell, 3 signals flow back to the fuse matrix, namely one from each register, and one from the pin (which can be configured as an input). Hence there are  $3 \times 32 = 96$  signals. In fact there are 2 additional input-only pins, and hence a total of  $N = 98$  signals.

Unfortunately, the fuse map grows with the square of the number of signals. A remedy to this dire fact had to be found. It consisted in subdividing the PLD into several partitions, in our case into 4. They are called *blocks* and consist of their individual fuse matrices and register sets. However, they are connected by the inputs to the fuse matrices, which are fed from the cell outputs not directly, but via a so-called *programmable interconnect matrix* (PIM). This is shown in Fig. 5. Note that all inputs (except 2) come from pins connected to cells whose tri-state gates are turned off.

On first sight, nothing has been gained. Now the PIM rather than the FM has  $n^2$  connections! Experience shows, however, that very rarely all, or even a majority of signals is used in any partition. The function of the PIM is to select a subset of all signals as inputs to the individual FMs, which now have only 33 (counting the inverted signals 66) inputs instead of 98. But how does this selection occur? Evidently it cannot be fixed once and for all. The solution lies in making the PIM (actually one PIM for each block) programmable in the sense of leaving open a choice of a few (say 9 out of 98) signals for each of the 33 inputs of the associated FM. The selection of the 9 out of the 98, however, is fixed by programming.

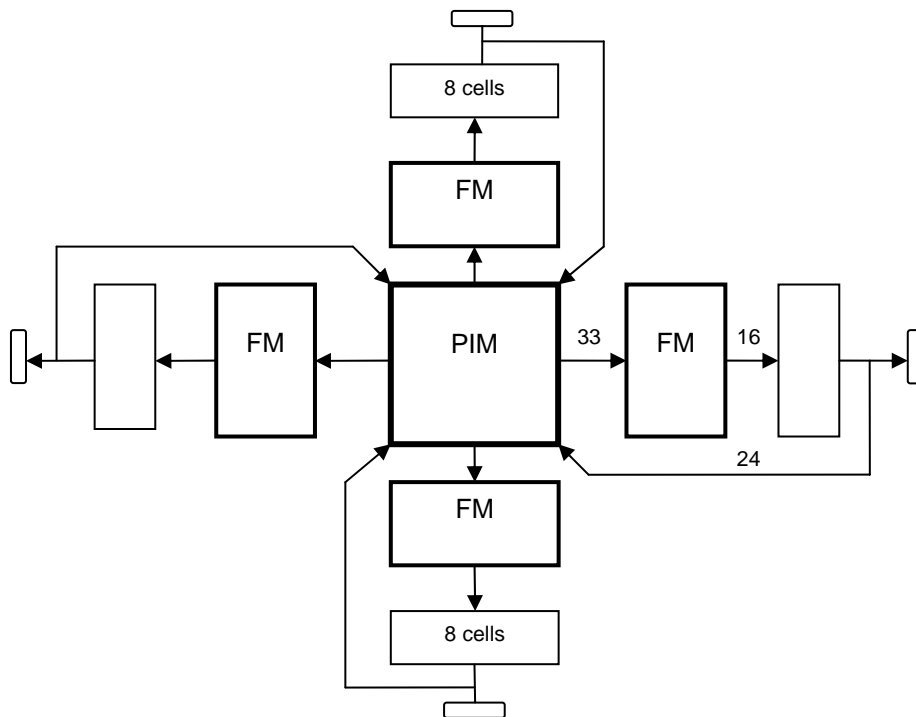


Fig. 5. The 4 partitions connected by the PIM

“Bigger” also means more complex transition functions, and this implies wider and- and or-gates. Again, there is a quadratic factor involved. A solution to keep the number of terms and factors within manageable bounds was found in a flexible instead of rigid connection of the or-gates (sums) to the cells (registers and pins). The output of an or-gate can now be selected to feed into one of 4 neighbouring cells. With every cluster having 5 terms, the number of terms for a cell can be as high as 20. This fan-out concept is shown in Fig. 6.

Which are the consequences of these new concepts? The consequences for the circuit designer are that (1) cells must be allocated to allow sufficiently many terms to be steered to it, and (2) cells must be allocated in blocks, such that arguments of the expressions belonging to blocks are evenly distributed. In other words, each block should have as few inputs as possible. These are tough requirements needing careful analysis and a search for a good solution. Of course problems become critical only if one wishes to make optimal use of the available resources. The difficulties give rise to calls for a compiler which performs these tasks automatically.

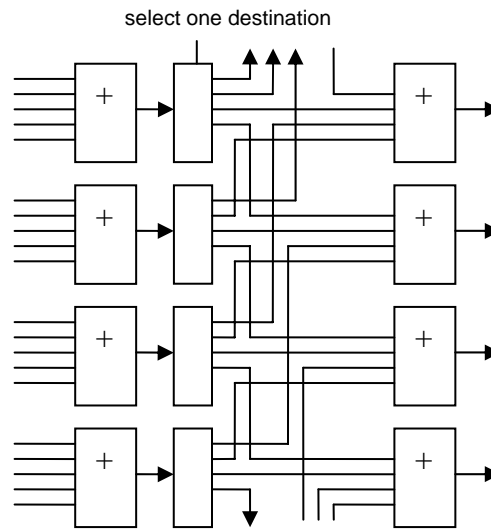


Fig. 6. Steering terms to cells

In addition to these features needed to keep the growing size manageable, one feature of the MACH-4 chip is provided to save gates in certain designs: Within each product term, it is possible to connect one of the 5 terms with an exclusive rather than a regular or-operator, i.e. to configure

$$y = x_0 - (x_1 + x_2 + x_3 + x_4)$$

instead of

$$y = x_0 + x_1 + x_2 + x_3 + x_4$$

We extend our language accordingly to

$$\text{expression} = \text{term} [ \text{"-"} \text{term} ] \{ \text{"+"} \text{term} \}.$$

The convenience of this feature is apparent in the following program representing a binary counter. It is indeed considerably simpler than the one shown above for the GAL.

```

MODULE Counter;
  IN ci: BIT;
  OUT q0, q1, q2, q3, q4, q5, q6, q7, co: BIT;
BEGIN
  q0 := REG(q0 - ci);
  q1 := REG(q1 - q0*ci);
  q2 := REG(q2 - q1*q0*ci);
  q3 := REG(q3 - q2*q1*q0*ci);
  q4 := REG(q4 - q3*q2*q1*q0*ci);
  q5 := REG(q5 - q4*q3*q2*q1*q0*ci);

```



```

q6 := REG(q6 - q5*q4*q3*q2*q1*q0*ci);
q7 := REG(q7 - q6*q5*q4*q3*q2*q1*q0*ci);
co := q7*q6*q5*q4*q3*q2*q1*q0*ci
END Counter.

```

The MACH4 offers a feature not present in the GAL, namely to initialize a register either to 0 or to 1. We reflect this by an optional apostrophe after a variable's identifier in its declaration. The apostrophe implies that the initial value is 1. As a consequence, the trick used in the example of the shifter using inversions ( $s0 := \sim\text{REG } s1'$ ) is no longer necessary.

```

MODULE Shifter;
  OUT s0', s1, s2, s3;
BEGIN s0 := REG s1; s1 := REG s2; s2 := REG s3; s3 := REG s0
END Shifter.

```

## 7. A Compiler for the MACH-4 PLD

Even more profound are the consequences of the architectural innovations for the design of a compiler. We quickly realize that the computation of the fuse matrices requires knowledge about the distribution of input signals by the PIM. Computation of the PIM in turn requires knowledge about which inputs occur in the respective blocks. A single-pass compiler will obviously not do. Compilation has to proceed through the following steps:

1. Parse the source text, build a list of variables and a graph representing the circuit.
2. Assign variables (signals) to blocks.
3. Identify the arguments (inputs) of each block.
4. Compute the 4 PIMs.
5. Compute the 4 FMs.
6. Load the data, configure the PLD.

The six tasks are evidently a lot more complicated than the entire compilation for the GAL. We will now have a look at the individual steps. Note that steps 3 – 5 can be performed independently for each block.

### 7.1. The Parser

The parsing process follows the same principle as that for the GAL: recursive descent. The difference lies in the generation of a data structure conveniently representing the program, instead of establishing the fuse map directly. The elements of this structure are of type *Signal*, of which the named variable is an extension:

```

TYPE Signal = POINTER TO RECORD
  op: INTEGER; x, y: Signal
END ;

Variable = POINTER TO RECORD (Signal)
  typ, blk, num: INTEGER;
  neg, reg: BOOLEAN;
  next: Variable;
  name: ARRAY 32 OF CHAR
END ;

```

The parsing procedures *term* and *expression* show how this structure is built up. From *expression* is also apparent, how care is taken of the new feature of the xor-operator.

```

PROCEDURE term(): Signal;
  VAR s, t: Signal;
BEGIN s := factor();
  WHILE sym = PLDS.and DO
    PLDS.Get(sym); NEW(t); t.op := 2; t.x := s; t.y := factor(); s := t
  END ;
  RETURN s
END term;

PROCEDURE expression(): Signal;
  VAR s, t, u: Signal; n: INTEGER;

```

```

BEGIN s := term();
  IF sym = PLDS.xor THEN
    PLDS.Get(sym); NEW(u); u.op := 4; u.y := s; s := term(); n := 2
  ELSE n := 1; u := NIL
  END ;
  WHILE sym = PLDS.or DO
    PLDS.Get(sym); NEW(t); t.op := 3; t.x := s; t.y := term(); INC(n); s := t
  END ;
  IF u = NIL THEN u := s ELSE u.x := s END ;
  RETURN u
END expression;

```

Note that the information concerning the cell, i.e. whether or not a register or an inverter are present, is retained in the variable descriptor rather than in the data structure. Neither of them shows up in the structure, which solely represents the expressions to be implemented in the fuse matrix.

## 7.2. Assignment of variable numbers

During compilation, every variable is identified by its number rather than its name. The numbers range from 0 to 97. The processing of its declaration associates a name with a number. As mentioned before, the concepts of blocks and of product term steering strongly determine the number association, if an optimal solution for the circuit is desired. As the rules for optimal assignment (sometimes called *placement*) are rather complex, one would wish this placement to be performed automatically by the compiler.

It is this drive for optimality that makes compilation utterly complicated. We forego this and adopt the simple solution to assign numbers in the order of the declarations. We acknowledge the fact that this may lead to situations where some signals might not be routable through the interconnect matrix. Output variables obtain numbers 0, 2, 4, ..., 62. Signals of "embedded" registers receive numbers 1, 3, 5, ..., 63. Input variables obtain numbers 64, 65, ..., 96, 97.

However, there is also a good reason to let the user assign numbers: The user must know to which pin a signal is attached. Often, a designer wishes to connect a signal to a fixed pin due to board design reasons. In order to provide this possibility, we let signal numbers be specified in declarations. If they are omitted, the compiler assigns sequentially.

## 7.3. Computing the interconnect matrix

Each of the four fuse matrices features  $M = 33$  signals as inputs. They must be fed through the block's interconnect matrix, which provides 9 choices (out of 98) for each of the 33 inputs. Evidently, for each block, we must first determine, which signals occur as inputs to the block's fuse matrix. (Note that this was the primary reason for a single-pass compiler being impossible). This search simply requires a scan of the variable list. The result is

```
input[0], input[1], ... , input[nofins-1]
```

Then follows, for each input, a search for a matrix column to which it is possible to feed that input. The 9 possible choices are given by a constant  $M \times 9$  matrix PIM. For fuse matrix column  $i$ , the available choices are  $PIM[i, 0], \dots, PIM[i, 8]$ . Hence, we must find, for input  $x$ , a column  $i$  and an index  $j$  such that  $PIM[i, j] = x$ . If found, the index  $j$  is entered into the fuse matrix FM by setting  $FM[i, k+j]$  to zero, where  $k$  is an appropriate offset. The interconnect data are embedded in the fuse matrix.

The challenge is to find indices  $i$  and  $j$  for each input  $x$ . This requires a search based on backtracking. It must proceed and recede in a fashion such that all possibilities had been explored before a failure is reported.

```

PROCEDURE connectable(s, i: INTEGER): BOOLEAN;
  VAR j: INTEGER;
BEGIN j := 0;
  WHILE (j < 9) & (PIM[i, j] # s) DO INC(j) END ;

```

```

    RETURN j < 9
END connectable;

PROCEDURE ComputePIM(blk, n: INTEGER; VAR res: BOOLEAN);
  VAR i, j, k, h, s, s1: INTEGER;
      done, fail: BOOLEAN;
      S: ARRAY M OF INTEGER; (*of 0.. 8*)
      selected: ARRAY M OF BOOLEAN;
BEGIN k := 0; fail := FALSE;
  FOR i := 0 TO M-1 DO col[i] := 0; selected[i] := FALSE END ;
  WHILE (k < n) & ~fail DO
    s := input[k]; i := col[s]; done := FALSE;
    WHILE (i < M) & ~done DO
      IF ~selected[i] THEN j := 0;
        WHILE (j < 9) & (PIM[i, j] # s) DO INC(j) END ;
        IF j < 9 THEN (*connection found*)
          selected[i] := TRUE; S[i] := j; col[s] := i; done := TRUE;
        END
      END ;
      INC(i)
    END ;
    IF done THEN INC(k)
    ELSIF k = 0 THEN fail := TRUE
    ELSE (*backtrack*)
      REPEAT DEC(k); s1 := input[k]; i := col[s1]; selected[i] := FALSE; col[s1] := 0;
      UNTIL (k = 0) OR connectable(s, i);
      WHILE (i >= 0) & selected[i] OR ~connectable(s1, i) DO DEC(i) END ;
      col[s1] := i+1; (*continue search*)
    END
  END ;
  IF k = n THEN
    res := TRUE; (*enter result S in FM*)
    IF blk IN {1, 2} THEN j := 189 ELSE j := 0 END ;
    IF blk IN {2, 3} THEN h := 1 ELSE h := 0 END ;
    FOR k := 0 TO n-1 DO
      s := input[k]; i := col[s]; FM[2*i + h, S[i]+j] := 0
    END
  ELSE res := FALSE
  END
END ComputePIM;

```

#### 7.4. Computing the Fuse Matrix

The next task is to compute the fuse matrix for each block from the data structure representing the expressions. Procedure *ComputeFM* first computes the row index of the first product term for the block, and then initializes the in and out multiplexors of all cells in the block. Then it disables all tri-state gates, i.e. sets all pins to input, and steers all product terms straight to their associated cells. After these initializations it traverses the list of variables, configuring the cells and processing the associates data structure by calling procedure *Or*.

```

PROCEDURE ComputeFM(k: INTEGER); (*for block k*)
  VAR v: Variable; i, j, j0: INTEGER;
BEGIN j0 := (k*21 DIV 2 + 1) * 9;
  FOR i := 0 TO 15 DO
    allocPT[i] := FALSE; j := first(i) + j0 + 4; (*S10, S11, S12 = 000 in and out mux*)
    FM[76, j] := 0; FM[77, j] := 0; FM[78, j] := 0
  END ;
  (*set all PT feeds to straight*)
  FOR i := 0 TO 15 DO j := first(i) + j0 + 4; FM[68, j] := 1; FM[69, j] := 0 END ;
  v := org;
  WHILE v # guard DO
    IF v.blk = k THEN (*variable in this block*)
      IF v.typ >= 2 THEN (*output or hidden variable*)
        IF v.x # NIL THEN ConfigCell(v, j0);
          IF v.x = one THEN allocPT[v.num MOD 16] := TRUE ELSE Or(v.x, v.num MOD 16, j0) END
        ELSE PLDS.Mark("undefined")
        END
      ELSIF v.typ = 1 THEN (*input variable*)
        END
    END
  END

```

```

    END ;
    v := v.next
  END ;
  IF rst # NIL THEN And(rst, j0 + 88) ELSE Zero(j0 + 88) END ;
  Zero(j0 + 89); (*preset row*)
END ComputeFM;

```

Output cells are configured by assigning values to the appropriate bits in the so-called architecture rows of the fuse matrix. This includes the inclusion/exclusion of the inverter, the register, and also enabling the exclusive-or capability of the first product term, the choice of register as a D-type, and the enabling of the tri-state output.

```

PROCEDURE ConfigCell(v: Variable; j0: INTEGER);
  VAR j, n: INTEGER;
BEGIN (* architecture fuses, function and default:
  S0: xor, 1
  S1: inverter, 1
  S3, S2: product term steering, 01
  S5,, S4: clock selection, 11
  S6: reset/prest, 1
  S8, S7: cell/reg typ, 11 (no reg)
  S9: cell type, synchronous, 1 *)
  j := first(v.num MOD 16) + j0 + 4; (*arch row*)
  IF ~v.neg THEN (*S1*) FM[67, j] := 0 END ;
  IF ~v.rst & ~v.neg OR v.rst & v.neg THEN (*S6 = 0, preset*) FM[72, j] := 0 END ;
  IF v.reg THEN
    (*S7, S8: D-Reg*) FM[73, j] := 0; FM[74, j] := 0;
    (*S4, S5: clk 0*) FM[70, j] := 1; FM[71, j] := 0
  END ;
  IF v.x.op = 4 THEN (*S0*) FM[66, j] := 0 (*XOR*) END ;
  IF v.typ = 1 THEN (*OE off*) FM[0, j-5] := 0; FM[1, j-5] := 0 END
END ConfigCell;

```

The processing of the data structures representing the expressions is analogous to the case of the GAL, except that the calls of procedures *Or*, *And*, and *Opd* are not within the parser, but are guided by traversing the structure. They correspond to *expression*, *term*, and *factor* respectively. Procedure *Or* is more complicated than its corresponding *expression*, because in the MACH PLD terms may be drawn from up to four product term groups by steering the groups (see explanation above). There are 5 terms per group. Hence, when the limit is reached, further terms must be drawn from neighboring groups. This is achieved by setting further FM-bits in the cell's architecture row (see procedure *SetPTfeed*). In order to draw term groups to a given cell, it is necessary to ascertain that the group is not used by another cell. Therefore, the use of a group is recorded in the global array *allocPT*. This represents a rather simple-minded scheme. A better one would require a solution requiring several passes over the data structure of the circuit to be compiled. The frequency of critical situations may be reduced by spreading variable allocation as much as possible. We here refrain from introducing further sophistication.

```

PROCEDURE Opd(s: Signal; j: INTEGER);
  VAR n, i: INTEGER;
BEGIN
  IF s.op = 1 THEN (*inv*) n := 1; s := s.y ELSE n := 0 END ;
  i := s(Variable).num;
  FM[col[i]*2+n, j] := 0
END Opd;

PROCEDURE And(s: Signal; j: INTEGER);
BEGIN
  WHILE s.op = 2 DO Opd(s.y, j); s := s.x END ;
  Opd(s, j)
END And;

PROCEDURE Or(s: Signal; k, j0: INTEGER);
  VAR j, j1, lim: INTEGER;
BEGIN j1 := first(k) + j0; j := j1; allocPT[k] := TRUE; lim := j+5;
  WHILE s.op >= 3 DO
    IF j = lim THEN (*PT cluster full*)
      IF (k < 15) & ~allocPT[k+1] THEN
        IF ODD(k) THEN j := j1+6 ELSE j := j1+5 END ;

```

```

        allocPT[k+1] := TRUE; SetPTfeed(j, -1, 1, 1)
    ELSIF (k > 0) & ~allocPT[k-1] THEN
        IF ODD(k) THEN j := j1-5 ELSE j := j1-6 END ;
        allocPT[k-1] := TRUE; SetPTfeed(j, 1, 1, 0)
    ELSIF (k < 14) & ~allocPT[k+2] THEN
        j := j1 + 11; allocPT[k+2] := TRUE; SetPTfeed(j, -2, 0, 0)
    ELSE PLDS.Mark("too many terms")
    END ;
    lim := j+5
END ;
And(s,y, j); INC(j); s := s.x
END ;
And(s, j); INC(j);
WHILE j < lim DO Zero(j); INC(j) END
END Or;

```

The auxiliary procedure *Zero(j)* makes product term *j* a 0 by selecting a variable and its inverse (in fact, all columns could have been selected with the same effect). Function *first(n)* computes the row index of the first product term of variable with number *n*.

```

PROCEDURE Zero(j: INTEGER);
    VAR i: INTEGER;
    BEGIN FM[0, j] := 0; FM[1, j] := 0
    END Zero;

PROCEDURE first(n: INTEGER): INTEGER; (*index of first PT of cell n*)
    BEGIN RETURN n * 11 DIV 2 + 1
    END first;

PROCEDURE SetPTfeed(j, s1, s0: BOOLEAN);
    BEGIN FM[68, j+4] := s0; FM[69, j+4] := s1; (*S2, S3*)
    END SetPTfeed;

```

## 7.5. Loading the Fuse Map

The fuse map is loaded over the same serial interface as that of the GAL: A clock (TCK), a mode (TMS) and data signals (TDI, TDO). The finite state machine in the PLD is considerably more elaborate than that of the GAL. Also, there exist several shift registers, and the state of the FSM determines, which one is loaded (or read): There are the instruction register (6 bits), the row register (80 bits), the column register (378 bits), and the ID-code register (32 bits). Only a single bit of the row register is set to 1, and it determines which of the columns of the fuse map is loaded from or read into the column register upon issuing a program or a verify command. A further complication arises from the requirement to enter a *programming mode* (through a specific command), before any other command can be executed. This is a measure to prevent accidental reprogramming.

## 7.6. Remarks

The MACH chip offers several additional facilities which have not been considered here, and whose implementation would complicate the compiler even further. Evidently they have been included in order to satisfy wishes of customers and to widen the usefulness of the PLD. They are here listed, together with the chosen default in parentheses.

1. *Clocks*. There are two input-only pins, and they are typically used for the clock. The clock can be selected individually for each register.
2. *Block clocks*. The clocks supplied to the registers can be selected for each block individually. (Default such that signal I1 is the clock for all blocks).
3. *Asynchronous mode*. Every cell can be either in synchronous or asynchronous mode. In the latter, the clock signal can be taken from a product term. (All registers synchronous).
4. *Inputs can be latched* on an individual basis. The enables for the latches are also selectable. (Inputs are not latched).

5. *Output switch matrices.* Cell outputs can be steered to a number of the pins in the block according to one of 8 schemes. (Every register outputs to the pin of its own cell).
6. *Inputs* to the fuse matrix can be selected by a 4 to 3 multiplexer for every cell.
7. *Reset and preset* signals for the registers can be generated by a product term for each block. (Reset selectable by RST assignment, Preset ignored).
8. The *register mode* can be selected for every cell among D-type, T-type, latch, or without register. (All registers D-type).
9. The register hold time can be selected globally.
10. Each block can be powered down if not used.

## 8. An experimental circuit board

A circuit board has been built for the purpose of testing the presented software. It contains a GAL22V10 (Lattice Logic) and a MACH4 – 32/64 (AMD) chip. In addition, it features a continuous clock signal of 10 Hz and drivers for the LEDs. Apart from sharing the clock, the two PLDs are independent (s. Fig. 7).

The 10 outputs of the GAL (numbered 0 – 9) and the 32 outputs of the MACH4 (numbered 0 – 31) all drive LEDs via 74ALS241 drivers.

Input 11 of the GAL and I0 (numbered 32) of the MACH4 are connected to the TDI signal, and they can be used to control the PLD.

Input 10 of the GAL and I1 of the MACH4 (numbered 33) are connected to a clock source. This source can be selected to either come through a fifth wire from the PC, or from the local clock.

The board is connectable to a PC via parallel port. The following 4 signals are used for loading the bit stream. The bit numbers  $D_i$  refer to the PC's Port Register:

PC	PLD	
D0 out	TDI and 11/96	data input
D1 out	TMS	mode (of FSM)
D2 out	TCK	clock for loading
D3 out	input 10/33	clock for running
D5 in	TDO	data output

The local clock is generated by a simple RC-oscillator with a 555 chip.

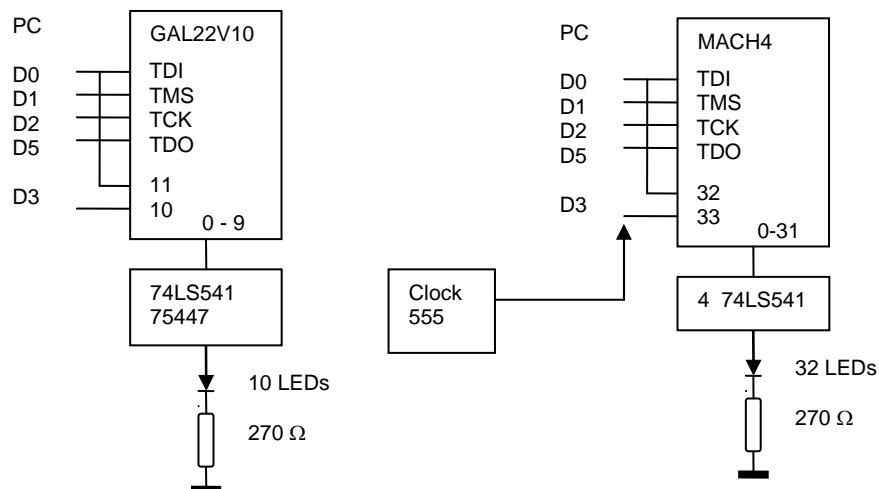
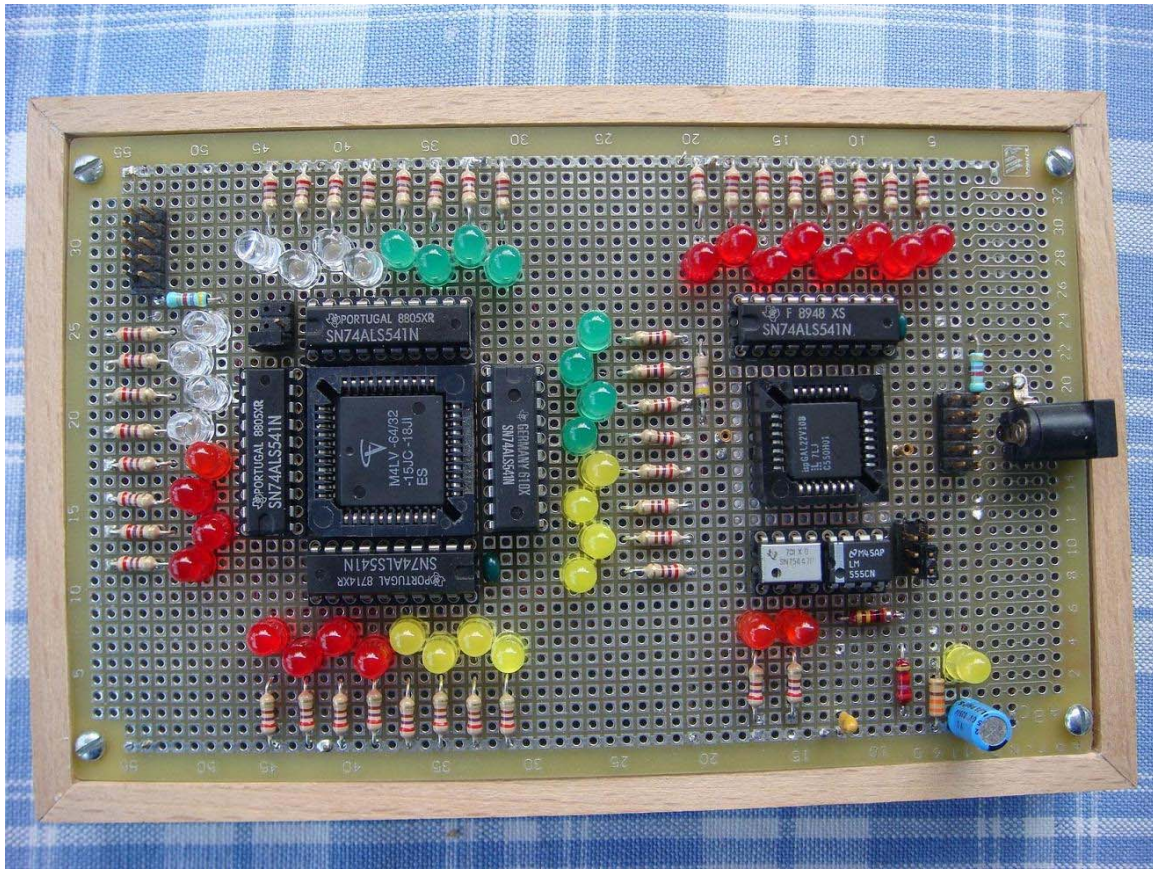


Fig. 7. Circuit board with GAL22V10 and MACH4 – 64/32 PLDs



### 8.1. Sample specifications for the GAL 22V10

```

MODULE M0;
  IN r;
  OUT q0, q1, q2, q3, q4, q5, q6, q7, q8, q9;
BEGIN
  q0 := REG q0';
  q1 := REG q1*q0' + q1'*q0;
  q2 := REG q2*q1' + q2*q0' + q2*q1*q0;
  q3 := REG q3*q2' + q3*q1' + q3*q0' + q3*q2*q1*q0
END M0.

```

```

MODULE M1;
  IN d;
  OUT q0, q1, q2, q3, q4, q5, q6, q7;
BEGIN
  q0 := ~REG d'*q7' + d*q1';
  q1 := REG d'*q0 + d*q2;
  q2 := REG d'*q1 + d*q3;
  q3 := REG d'*q2 + d*q4;
  q4 := REG d'*q3 + d*q5;
  q5 := REG d'*q4 + d*q6;
  q6 := REG d'*q5 + d*q7;
  q7 := REG d'*q6 + d*q0;
END M1.

```

```

MODULE M2;
  OUT q0, q1, q2, q3, q4, q5, q6, q7, d;
BEGIN
  d := REG d'*q2 + d*q1';
  q0 := ~REG d'*q3' + d*q1';
  q1 := REG d'*q0 + d*q2;
  q2 := REG d'*q1 + d*q3;
  q3 := REG d'*q2 + d*q0;

```

```

q4 := REG d*q7 + d*q5;
q5 := REG d*q4 + d*q6;
q6 := REG d*q5 + d*q7;
q7 := ~REG d*q6 + d*q4';
END M2.

```

## 8.2. Sample specifications for the MACH4 – 64/32

```

MODULE M3;
  IN rst (32); OUT q0(0), q1, q2, q3;
BEGIN RST rst';
  q0 := REG q0';
  q1 := REG q1 - q0;
  q2 := REG q2 - q1*q0;
  q3 := REG q3 - q2*q1*q0;
END M3.

MODULE M4;
  IN rst (32);
  OUT q0' (0), q1, q2, q3;
BEGIN RST rst';
  q0 := REG q3;
  q1 := REG q0;
  q2 := REG q1;
  q3 := REG q2;
END M4.

MODULE M5;
  OUT q0'(0), q1, q2, q3, q4, q5, q6, q7', q8'; q9, q10, q11, q12, q13, q14, q15',
  q16', q17, q18, q19, q20, q21, q22, q23', q24', q25, q26, q27, q28, q29, q30, q31';
  VAR d (0);
BEGIN d := REG d*q2 + d*q1';
  q0 := REG d*q3 + d*q1; q1 := REG d*q0 + d*q2; q2 := REG d*q1 + d*q3; q3 := REG d*q2 + d*q0;
  q4 := REG d*q7 + d*q5; q5 := REG d*q4 + d*q6; q6 := REG d*q5 + d*q7; q7 := REG d*q6 + d*q4;
  q8 := REG d*q11 + d*q1; q9 := REG d*q8 + d*q10; q10 := REG d*q9 + d*q11; q11 := REG d*q10 + d*q8;
  q12 := REG d*q15 + d*q13; q13 := REG d*q12 + d*q14; q14 := REG d*q13 + d*q15; q15 := REG d*q14 + d*q12;
  q16 := REG d*q19 + d*q17; q17 := REG d*q16 + d*q18; q18 := REG d*q17 + d*q19; q19 := REG d*q18 + d*q16;
  q20 := REG d*q23 + d*q21; q21 := REG d*q20 + d*q22; q22 := REG d*q21 + d*q23; q23 := REG d*q22 + d*q20;
  q24 := REG d*q27 + d*q25; q25 := REG d*q24 + d*q26; q26 := REG d*q25 + d*q27; q27 := REG d*q26 + d*q24;
  q28 := REG d*q31 + d*q29; q29 := REG d*q28 + d*q30; q30 := REG d*q29 + d*q31; q31 := REG d*q30 + d*q28;
END M5.

MODULE M6;
  IN r(32);
  OUT q0(0), q1, q2, q3, q4, q5, q6, q7, q8, q9, q10, q11, q12, q13, q14, q15,
  q16, q17, q18, q19, q20, q21, q22, q23, q24, q25, q26, q27, q28, q29, q30, q31;
  VAR d (0);
BEGIN RST r'; d := REG d*q1 + d*q2;
  q0 := REG d*q1; q1 := REG d*q2 + d*q0; q2 := REG d*q3 + d*q1; q3 := REG d' + d*q2;
  q7 := REG d*q6; q6 := REG d*q5 + d*q7; q5 := REG d*q4 + d*q6; q4 := REG d' + d*q5;
  q8 := REG d*q9; q9 := REG d*q10 + d*q8; q10 := REG d*q11 + d*q9; q11 := REG d' + d*q10;
  q15 := REG d*q14; q14 := REG d*q13 + d*q15; q13 := REG d*q12 + d*q14; q12 := REG d' + d*q13;
  q16 := REG d*q17; q17 := REG d*q18 + d*q16; q18 := REG d*q19 + d*q17; q19 := REG d' + d*q18;
  q23 := REG d*q22; q22 := REG d*q21 + d*q23; q21 := REG d*q20 + d*q22; q20 := REG d' + d*q21;
  q24 := REG d*q25; q25 := REG d*q26 + d*q24; q26 := REG d*q27 + d*q25; q27 := REG d' + d*q26;
  q31 := REG d*q30; q30 := REG d*q29 + d*q31; q29 := REG d*q28 + d*q30; q28 := REG d' + d*q29;
END M6.

```

## 9. Conclusions

From the foregoing presentation it is amply evident that unobtrusive additions to the basic PLD (as represented by the GAL chip) leading to the larger (MACH4) chip have complexified the implementation of the compiler most dramatically. This occurred although some special features of the MACH have been ignored, and although no perfection was sought. This example is by no means untypical for many situations in hardware design and its consequences on software.

When this point is raised, the typical and defensive answer is that the added features were demanded and that they increased the power of the design. The more complicated software was



the unavoidable price, but was of no consequences, because it could easily be compensated by modern, faster computers which effectively hide the added problems.

As that may be so, it is nevertheless interesting to wonder whether alternatives had been considered. In most cases, there would have been alternatives. Whether they would have been better in some respect, is another question. Let us consider our example at hand.

We first realize upon closer reflection that the MACH architecture essentially represents not one FSM, but rather four of them. They are, however, interconnected. The interconnect matrix, whose structure remains undisclosed, is necessarily sparse, and it is cleverly organized so that its sparseness remains vague. It is the handling of this interconnect problem which complicates the compiler in particular.

Might it not have been wiser to honestly reflect this fact in the design of our small language, not to hide the actual structure in the name of helping the designer? For example, this could be achieved by openly presenting the four partitions as submodules, together with explicit lists of those input signals (variables) that are imported from the other modules. This would considerably simplify the compiler (render the need for a 2-pass scheme superfluous), and also make the user aware of the actual structure of the chip's design.

A deeper reaching alternative would have been at the PLD-architecture level itself. The designers might have chosen to omit an interconnect matrix altogether, and in stead designate a subset of a block's outputs to be available in the other blocks. This solution might not only have simplified the compiler, but the PLD itself, possibly allowing to increase the number of cells.

As it stands now, the crept-in complexity cannot be reverted, because future chips will have to be compatible. Therefore we simply add the problem to the pile called legacy software.

In discussing this situation, we must be aware of a trend that has become pervasive. It is the trend to relieve customers from problems and difficulties whatever the cost. Customers often would not understand the difficulties, and often do not much care about the cost. They are quite willing to pay for any difficulty taken off their shoulders.

This is by no means a novel phenomenon. Simplifying the design process was also the driving force behind the development of assemblers, languages and compilers half a century ago. Compilers had always been complex programs. Early versions were complicated and messy, because they lacked solid principles, and early programmers were rather skeptical about using compilers and afraid to trust them. The situation improved when the concept of formal syntax was introduced and the problem of syntactic analysis was mastered. But code generation remained rather a craft, mostly because of baroque and irregular computer structures and instruction sets. This situation improved with a growing awareness against home-made complexity. The advent of simpler, regular computer architectures (RISC) alleviated the problem further. The translation of language statements into code sequences became more logical and systematic.

However, the urge to "optimize" (a more truthful verb would be "to improve") code increasingly let translation slide back to a sophisticated and obscure craft. The burden of obtaining efficient and dense code was shifted from hardware to software. It became increasingly accepted that there can be no limit to the sophistication of compilation. This led to the current situation, where nobody ever dares to analyze the generated code. It would in most cases be difficult to understand it.

The same development is now in progress in the field of hardware design. Transistors – the ultimate element – are available in huge quantities. Hence, automated design tools are a sheer necessity. This situation is aggravated by the fact that the translation from logic equations to circuit layouts is much more difficult than that from language statements to code sequences. The reason is that the result is a two-dimensional layout taking into account physical phenomena and limitations. The current tendency is, moreover, to translate not from equations, but directly from algorithmic statements into a circuit, thereby having to bridge an even wider, a huge gap. This width becomes manifest as soon as a design is erroneous. The results show up in testing, only after compilation. Tracing them back across the gap is often very difficult, and sometimes

impossible without understanding the very translation process (which the tool's promoters had proclaimed to be superfluous).

The width of this gap worsens the dilemma between blindly using convenient tools without having a full understanding on the one hand, and avoiding the use of such tools and foregoing modern devices and methods on the other hand. Whatever, the *grand challenge* is in the hands of teachers.

With faster computers, complex software is normal and unavoidable, and in mastering complex situations computers are much more reliable and efficient than humans. Therefore, sophisticated search algorithms for translation become normality, and designers will increasingly rely on automated tools, be it in hardware or software development. Understanding the whole has simply become impossible.

Even if this may be so, and even if we learn to trust these complex tools, no effort should be left out to eliminate avoidable complexiy.