Implementation of algorithms into hardware

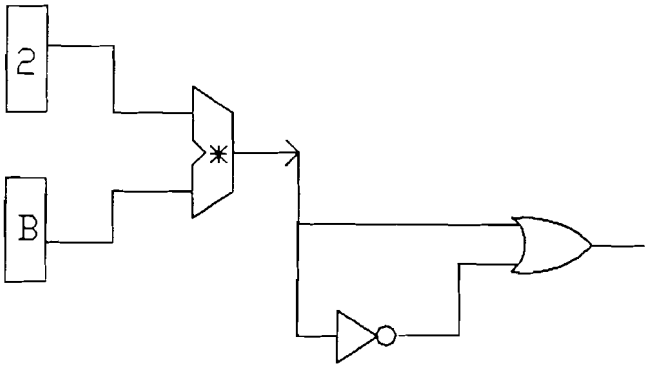
Klemann, O.M.R.

Award date:
1988

Eindhoven University of Technology
Department of Electrical Engineering
Digital Systems Group (EB)

IMPLEMENTATION OF ALGORITHMS INTO HARDWARE

by O.M.R. Klemann



Master thesis report by: O.M.R. Klemann
Coach: Prof. ir. M.P.J. Stevens

Eindhoven, The Netherlands
July 1988

ACKNOWLEDGEMENTS

I would like to thank:

- my parents for everything

- prof. ir. M.P.J. Stevens. He has not only been an excellent coach but a very good personal adviser as well. I hope the nice contact will remain after leaving the EUT.

- all members of the Digital Systems Group for their help and friendship

- Dominique for her support and love

Title page: hardware implementation of W. Shakespeare's

" (2B) OR NOT(2B) "

SUMMARY

Since designing of VLSI devices has become more and more complex while the time to design ICs has to be reduced, powerful design tools have become very important.
The Digital Systems Group of the EUT is working on a new approach to design digital systems by implementing algorithms directly into silicon.
Starting at the top level using a high level programming language the system has to generate a complete digital system design automatically.

This report describes the lower part of such a system.
The high level programming language has to be compiled to an Register Transfer Language (RTL).
Then the algorithm written in that RTL can be implemented in hardware, using the method described in this report.

Procedures will be mapped onto finite state machines.
These finite state machines consist of a control part and an information processing part.

CONTENTS

1.INTRODUCTION

In general, the problem faced by digital designers is to create a hardware implementation of a digital system which exhibits a specified behavior, and is "optimal" with respect to some set of design goals.
Often, the design goals compete (cost and performance, for example).

Representations of a digital system have been defined at several different levels of abstraction to describe a design at different stages of the design process.

The level of abstraction on which a designer works is dependent on his contribution to the design process. System architects rarely deal with transistors and logic gates, and logic designers may not have access to a whole system level description. However, in the context of the whole design process, each designer must be able to understand the effect that design decisions at his level of the design process have on the product as a whole.

The evaluation of a design which may be represented at several different levels of abstraction requires a multilevel representation and analysis aids which can effectively use it.

Several multilevel analysis aids have been developed that require more than one level of abstraction as input. One of these aids, multilevel simulation is being used by the Digital Systems Group of the EUT. In general, simulation at a low level of abstraction is accurate but slow and expensive, while simulation at a higher level of abstraction is less expensive but delivers less detailed information.

To avoid simulation at all levels between behavior and hardware description an other methodology has to be used: "correctness by construction". In this report such a methodology is introduced.
A way to come to automatic synthesis hardware, starting with an algorithm is described.

2.THE DIGITAL DESIGN PROCESS

2.1.History

Until a few years ago, the normal method of designing a digital system was not a strict way.
Starting with a specification, the problem was divided into functional blocks. Then one would try to find a realization of these blocks by "gluing" standard pieces of hardware together.
A so called "bottom-up" approach. It was almost impossible to find out what the consequences of changes in one part of the design would be in other parts.
Only small digital systems can be made this way, since the only way of testing the design is to build it.

2.2.Top down

Considering the high complexity of modern systems new design strategies have become necessary.
We need a structural design method using a hierarchy of design representations organized by levels of abstraction.
This section describes a traject that is used by the Digital Systems Group of EUT. From the top level down, this hierarchy typically includes the following levels of abstraction:

1. Specification
2. Behavioral level
3. Functional level
4. Structural level
5. Physical level

Specification
Starting to design a digital system, we need the desires of the customer. In general this will be a description in a common language.
At this place tasks and performance of the system have to be known.
This informal description must be transformed into a formal description in order to be able to make decisions and to be able to test if the system will meet the specified constraints.

Behavioral Level
At this level the behavior of a digital system is described without specifying its structure or implementation. More about this level can be found in Chapter 3.

Functional level
At the functional level the architecture is defined using abstract components (functional blocks) and their interconnections. A control sequence is often included to specify how the abstract components are invoked.

Structural level
The functional blocks are translated into logic circuits, functional cycles, microprogramming, finite state machines and boolean expressions.

Physical Level
At this level the digital system implementation is described in terms of physical devices, their placement and their interconnections.

Designs-in-progress move between these levels of representation through three basic design activities:

Synthesis: This activity is the creative act of generating a new lower level of representation of information based on a higher level (more abstract) specification

Optimization: This activity involves adding detail within a level or altering the representations within a level to effect a better design. Transforming a behavioral description to be more parallel could be an optimization to increase speed.

Analysis: This is the activity of evaluating a design representation to indicate where optimization or better synthesis could be performed to better meet specified design goals. This evaluation often includes functional correctness, cost, performance, and testability measurements.
Performance analysis often will be done by system simulation.

2.3.A new approach

The method described above is a systematic way to come to a good digital system design. Considering this systematic approach, thoughts arose to automate the traject. Starting with an algorithm there might be ways to come to automatic generation of hardware. In the literature one can often find the word "silicon-compilation" to describe this process.

Since the behavioral description does not implicate an implementation, there must be an intermediate level do describe functional blocks with their communication and synchronization. This level combines the functional level an structural level of the top down method.
An intermediate language has been chosen to describe the data transformations and control operations at that level. This language is called a register transfer language (RTL).

Since the RTL is closely related to a hardware description, this language can be used to generate a hardware design.

At the RTL level, optimization can be realized by clustering variables, operators and interconnection. This will be described in Chapter 8. At this level statements can be executed in parallel, which also has to be considered when translating the functional blocks into RTL.
Figure 2.3.1 shows the total traject of the new approach.

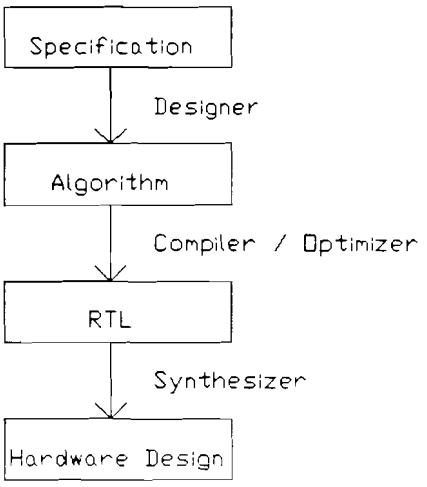


Fig. 2.3.1. General diagram of automatic system design.

3.BEHAVIORAL LEVEL

There are many ways to describe the behavior of a system.
The Digital Systems Group of the EUT has chosen for a Pascal-like hardware description language 'HHDL'.
Using a software package of Silvar-Lisco 'SL2000' a system described in 'HHDL' can be simulated at this level.

At this level one has to describe:
- which actions can be distinguished e.g. subprocesses, procedures, functions, and data exchange
- which actions can be executed in parallel
- synchronization and relative timing of subprocesses, procedures and functions
- shareability of data communication lines

The problem to be implemented is split up into functional blocks.
The program obtained will look like this:

```
Program(input, output)

    declaration of global variables

    Procedure1(header variables)
    Procedure2(header variables)
        ..
    ProcedureJ(header variables)

(co)begin
      procedure1(global variables)
      procedure2(global variables)
        ..
      procedureJ(global variables)
(co)end
```

In general the algorithmic description of a procedure looks like this:

```
    Procedure1(h1,..,hm)
    var h1,..,hm      /* header variables */
    (co)begin
      /* statements, flow-control, procedure calls,
         synchronization.
      /* communication up by using global variables
      /* communication down by using header variables
    (co)end
```

Figure 3.1 shows a schematic diagram of a program

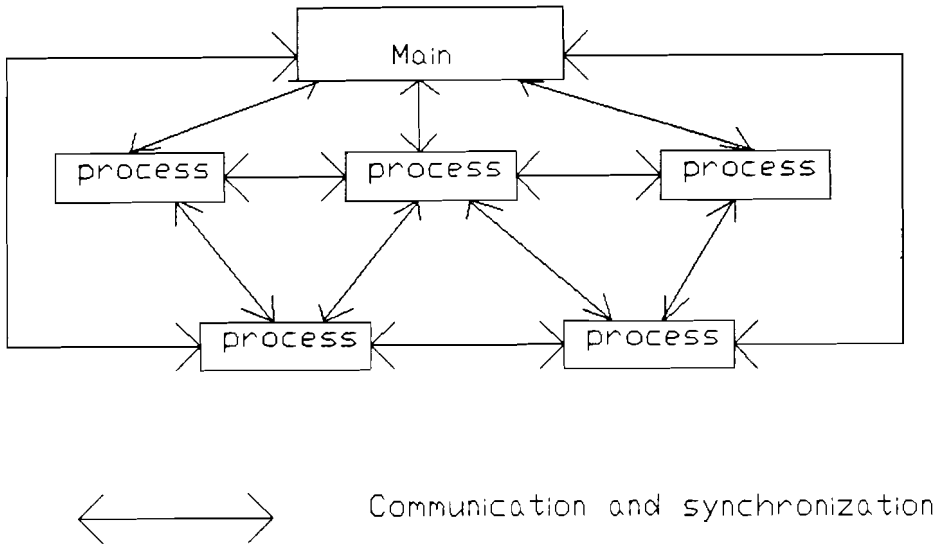


Fig.3.1. Algorithm composed of functional modules

It is built up by modules(procedures). Modules have entry and exit points. Through these points procedures communicate and synchronize. In the program this is done by global and header variables. These variables will lead to buses between the modules. Communication and synchronization among modules is beyond the goal of this report. For further information, see [23].

Splitting up a program into functional blocks does not implicate that the implementation in hardware will use the same functional blocks.

4.REGISTER TRANSFER LEVEL

The behavioral language has such a high level that the system cannot directly be implemented from this language . We need a language that describes the functions in such a way that they can be implemented directly in hardware. Several languages of this type have been proposed in the literature (AHPL, ISPS, KARL).[6, 7, 12, 22]

4.1.Purposes

The language can be used for the following purposes:
- Functional specification of the system, without concern for the implementation.
- Description of a specific implementation. In this case the description corresponds to the implementation as close as possible.
- Aid in the process of designing a system. Different realizations can be tested on costs, speed, and size.

4.2.Requirements

At this level data transfers and transformation (register transfers) are described. Also the control sequence is included.

For this reason the following elements in such a language are necessary:

- Data objects (data elements or structures of data elements) with a set of allowable values.
- Operators, to perform actions like AND , OR, ADD, and SUBTRACT.
- Functions, having objects as arguments. For example sqrt(x) is a function to calculate the square root of x.
- Assignments, which assign the results of a operation to an object.
- Structuring constructs, which allow the construction of algorithms as a composition of other algorithms.
- Special separators to enable parallel execution of statements.
- Comments, for documentation.

A function can be implemented by using standard operators or by using dedicated hardware (for example a high speed multiplier). When using composed operators it is necessary to know sources and destination and the time needed for the operation. The control unit has to take care of the right timing and control signals. Sometimes it is possible to perform a function in-register. An example of a function that can be performed in-register is a decremental register. In stead of using an ALU for decrementing the value in a register, a special register can be used which has a special input to decrease the value directly by one.

4.3.Language

In this report a C-like language has been used as the register transfer language.
A new element has been introduced:

--> 2

means go to state 2.
It is a control function, which must be executed by the control unit.

5.FROM BEHAVIOR TO RTL

We now want to project the behavioral description onto the register transfer language. This could be done by a compiler. Since it was not the goal of this project to design such a compiler, only the specifications are given below.

Functions of the compiler:
- compile the behavior description to RTL
- use standard elements (modules) of a library (registers, operators etc.)
- trade off costs and speed; find an optimum between number of operators and time to execute an algorithm
- supply a control part representation (the micro-operation sequence)
- supply timing information

For the methodology developed the system is divided into a data subsystem and a control subsystem, which is closely related to the RT language nature itself.
So the RTL synthesis has two tightly coupled facets: the design of the data paths, and the design of the finite state machine which controls these data paths. Fig. 5.0 shows a diagram of a system described this way.

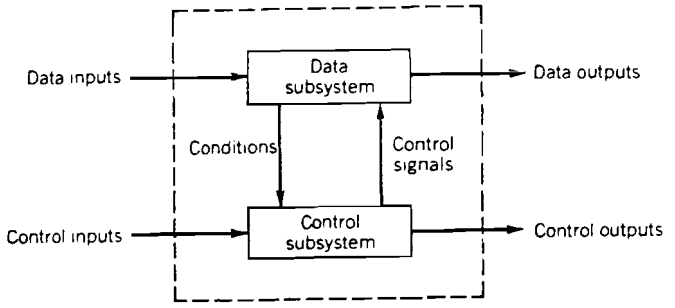


Fig. 5.0. Structure of a system with data and control subsystems

The compiler translates a design from the behavioral description to the functional structure (RT) level. Procedures in the behavior language will be mapped on basic blocks (see Chapter 8). Designs entering the logic synthesis and module selection step are represented at a behavioral level which may contain operations that are not directly realizable by devices in a module set. In cases a match between members of the module set and operations in a design cannot be found, some methodology must be applied to reduce the difference until the available modules can be used to implement the description.

5.1.Data unit

The data unit contains registers, operators and interconnection. The elements are part of a library, so the problem to be solved

by the compiler is a mapping problem. During compilation an optimum solution has to be found in that sense that the number of registers, operators, and interconnection units has to be minimized. A trade off between speed and cost has to be made. Parallel solutions will be fast but take a lot of chip area since they will cost more elements. Using less elements will decrease the speed of the system. So speed and costs are input vectors to the compiler. More about optimization can be found in Chapter 8.

5.2.Control unit

The control information for a design is provided in a form separate from the data path.
For representing the control flow, a precedence graph could be used [1].
Labeling methods and calculation of the critical path can also be found in [1]. During compilation such a description could be used to find the maximum number of operations that can be performed at any time.
The control flow (e.g. a micro-operation sequence) has to be made by hand, since we are not able to generate it automatically. The micro-operation sequence is a list of operations with sources, destinations, and interconnection links.
The problem to be solved is to map control specifications into hardware implementations.
The data paths of the system under design must have been fully defined, device primitive operation times must be known, and the sequence of control signals necessary for each primitive operation must be accessible to the synthesis routines. The required operation of the machine must be specified as a linear, ordered list of micro-operations which affect either the control flow or the data path. The synthesis routines must determine micro-instruction format, control store word width and length, clustering of micro-operations into microinstructions, and controller hardware parameters.

Automatic synthesis of microprogrammable control hardware is different from microcode compilation because there are more degrees of freedom available to the synthesis program. Microcode assembly. optimization, and compilation assume knowledge of existing control hardware, with word widths, branching strategies and formats predefined. Conventional optimization routines perform either word optimization or bit optimization but do not take into account the influence of one on the other. Synthesis must consider both optimizations at once.
Timing and clocking are the most difficult part of this synthesis.
More about the control unit can be found in Chapter 7.

6.DATA UNIT

The data unit performs the information processing tasks. The data unit cannot perform any computation by itself; it has to be controlled by a controller. The controller determines the sequence in which operations must be performed. The controller sends the appropriate control signals and transfer pulses to the data unit. The next step of the controller may depend on the current status of the information being processed. See Fig. 6.0.

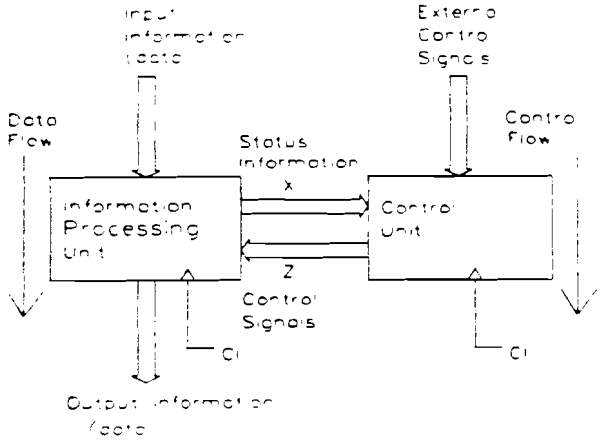


Figure 6.0 Basic model of a digital system

The basic tasks of the data unit are:

1. Storage of data
2. Information transfer
3. Operations
4. Decisions (status)

6.1.Storage

The storage components provide storage for data and condition values, using registers build up by elements that can store bits (flip-flops). The basic storage component is a register, with a LOAD and a RESET operation. The LOAD signal is used to load the input vector into the register, synchronized with the clock. This corresponds to the assignment instruction of the RT language. For initialization (reset) purposes a RESET operation can be implemented . An output enable line can be added to the register when buses are used. The register contents are only available at the output when the READ signal is present; otherwise the outputs are undefined (three state).
An n-bit register with LOAD and RESET operations is shown in Figure 6.1.1.
Because of the synchronous mode of operation, when loading a new input vector, the old contents of the register is also

available for that clock period. That is, the old contents of a register is also available at the time it is being loaded [2, 22].

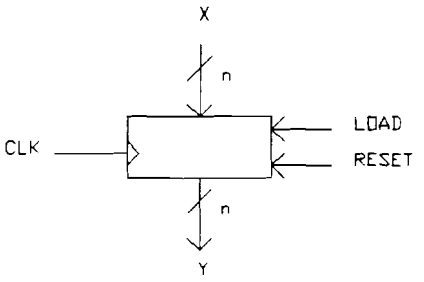


Fig. 6.1.1. Register.

The LOAD operation could be described as
'if LOAD then Y := X'.
The RESET operation could be described as
'if RESET then Y := 0'.

The storage function can be implemented by using individual (separate) registers or arrays of registers or a combination of both. The access to an individual register requires separate datapaths and control signals. The more concurrency in a system the more individual registers are necessary. On the other hand they need more interconnections. If a large number of registers is required, an array of registers is used to reduce the number of datapaths. In an array of registers, see for example Figure 6.1.2, some datapaths are shared so that only one operation involving the shared datapath can be performed at a time, imposing a limitation on the concurrency achievable by the system.
Figure 6.1.3 shows an array of registers organized as a Random-Access Memory (RAM). By supplying an address at the A input a register can be selected. Then a READ or a WRITE operation can be performed.

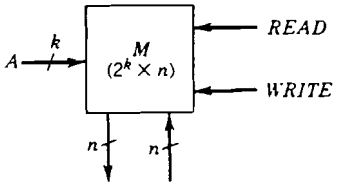


Fig. 6.1.3. Random-Access Memory

6.2.Information transfer

Two types of transfers are possible:
- transfer of information from outside the system to a register or vice versa.
- transfer of information from register to register.

Information transfer is initiated by the receipt of a control vector and accomplished a transfer pulse (clock).
Figure 6.2.1 shows an example of a transfer network. Control vector Z determines which transfer from the inputs to the registers will occur. The values of the input and control signals must be stable before the transfer pulse occurs. Control and input signals may change between transfer pulses.

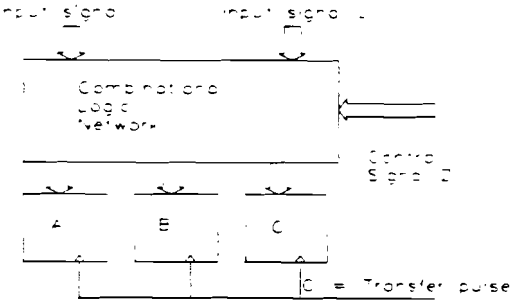


Fig. 6.2.1 Example of a transfer network.

The contents of registers not referenced remain unchanged. The external behavior of the data unit of the example is defined by the specification table (also in Fig. 6.2.1). The specification table can be used to write the logic equations at the logic level. The width of a datapath is the number of bits that can be transmitted simultaneously. Transmission of data on datapaths can be parallel (all bits simultaneously) or serial (one bit at a time). Parallel transmissions are fast but require wider datapaths. Uni- and bi-directional datapaths are possible. Bidirectional datapaths reduce the number of connections but the transmissions are limited to one at the time. A shared datapath, or bus, provides for transmission between several sources and destinations. At one time only one source can be active but more than one destinations are possible. A multiplexer is used to connect several sources to one destination (Figure 6.2.3.).

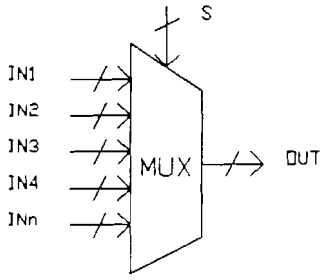


Fig. 6.2.3. A multiplexer.

A demultiplexer connects one source to one of several possible destinations. Its block diagram is shown in Figure 6.2.4.

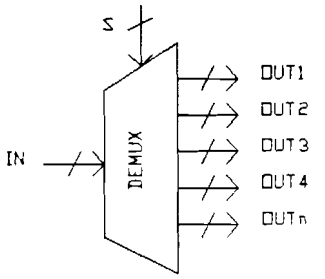


Figure 6.2.4. A demultiplexer.

6.3.Operations

An operation (function) at the implementation level has bit-vectors as operands and bit-vectors as results. An operator is implemented by a combinational network. Standard operations correspond to standard modules like AND, OR etc. More complex operators (multipliers, dividers etc) are implemented by a sequential network: in this case the operator includes the corresponding control sequence (semicentralized control).
Such a complex operator can be seen as a subsystem, see Figure 6.3.1.
The operator P accepts the inputs when initiated by the START signal, performs the operation selected by the control signals (C_0, C_1), and generates the signal DONE when the output bit-vector Y is computed.

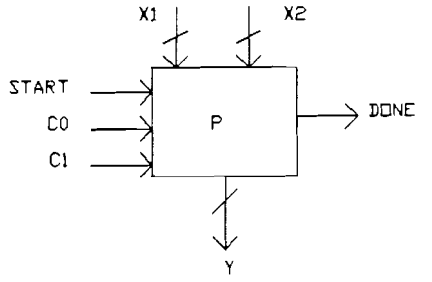


Fig. 6.3.1 Operator as a subsystem.

The operator is specified by the function it performs. The specification can be given by function tables, switching expressions, etc.

Figure 6.3.2 shows an example. By supplying a control vector Z the data unit will perform the action specified in the specification table.

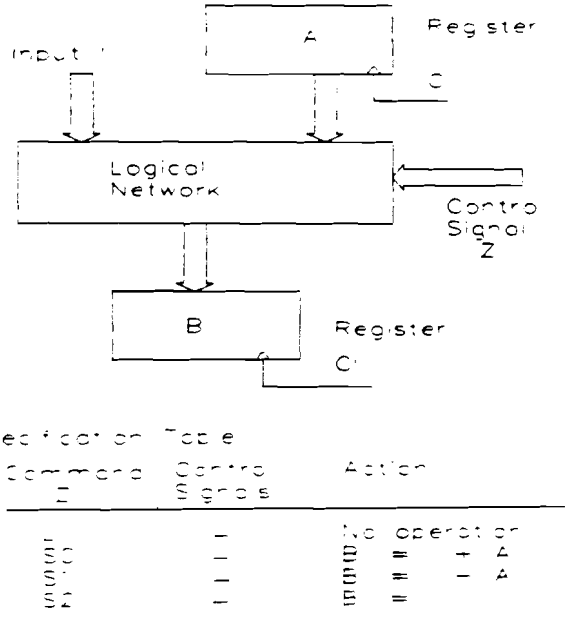


Fig.6.3.2. Example of an operational network

It is also possible to feed back the output of a register to the input of the operational network. In this case the register used as feedback must be of the master-slave type or an edge triggered flip-flop to avoid races. In examples of Chapter 8 we will see that this construction reduces the number of registers.

6.4.Decisions

In our general model of a digital system (Fig. 6.0) there is also an signal from the data unit to the control unit. This is a status

signal. After having executed an operation the data unit can set some signal lines. The control unit can supply the next state vector by using these status signals. Since the next state now depends on the status signals of the data unit, decisions can be made, see Figure 6.4. In Chapter 7, this is worked out in detail to implement 'if then else', 'while', and 'for' instructions. Figure 6.4.2 shows the corresponding specification table and status signal table. The status X is independent of the control signal, so in fact the combinational network can be split up into a part producing the status signal X, and a part processing the data controlled by the vector Z.

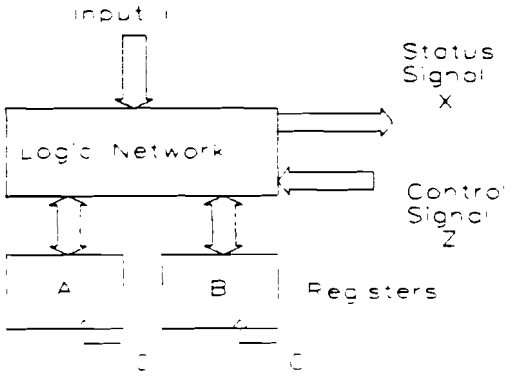


Fig. 6.4.1 Structure of an operational network with a status signal X

Specification Table

Command	Control Signals	Action
-	-	No operation
S0	-	A = A-B, B = A
S1	-	A = A, B = A+B
S2	-	A = B, B = B

Status Signal Table

Condition	Status Signal x
A = B	x2
A < B	x1
A > B	x3

Fig. 6.4.2 Specification and signal table

6.5.Modular and bit-slice organization

The data subsystem can be partitioned in two basic ways:
- a modular organization
- a bit-slice organization

In a modular organization each module implements a function. For example, separate modules could implement data storage and arithmetic operations. Each of these modules can be implemented as a network of components.
In a bit-slice organization a module implements all functions for one bit in a datapath. Several of such modules are required for the complete processing part.

7.CONTROL UNIT

There are many ways to implement the control unit. In the next paragraphs some approaches are described.
To be able to design a control unit there are two requirements. The first requirement is a description of all the devices used in the data path and how the device must be controlled for each operation.
The second requirement is a description of the control sequence. A control sequence is a representation of the proper order of events which should occur in the digital system. This sequence may be represented in a RTL language, a state table or it may be expressed in detail by specifying the actual bit patterns used to control the devices used in the data path.

A two-phase clock is assumed. Phase one invokes controller state changes; phase two invokes data path operations.

The Digital Systems Group of the EUT has software tools to minimize the number of states of a controller. It also has tools to minimize the combinatorial part of the controller.

A general procedure for designing a control unit, given a register-transfer algorithm and a data path, consists of the following steps:
- assign a state to each line of statements of the RTL algorithm;
- construct the state diagram according to the precedences specified in the RTL algorithm.
- for each state determine the outputs (control signals) that are active in the state, depending possibly on conditions and external inputs.
- determine the inputs (conditions and external inputs) causing the state transition.
- create the state table
- use tools to minimize the state table
- construct the control unit.

There are two types of implementations of a finite state machine:

1. Mealy-type
2. Moore-type

7.1. Mealy-type implementation

Using a Mealy-type implementation, the output is a function of the present state and the inputs. This means that if the inputs are not stable, the output is also not stable. This could be a disadvantage. An advantage of this type of implementation is that usually less states are necessary than with a Moore-type implementation to implement the same function.
Figure 7.1. shows a general diagram of a Mealy-type implementation of a control unit.

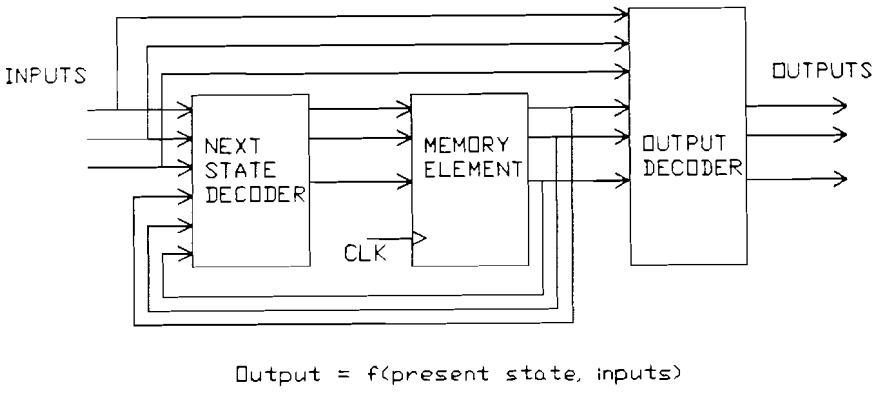


Fig. 7.1. Mealy-type implementation of a control unit.

7.2. Moore-type implementation

Typical for a Moore-type implementation of a finite state machine is the fact that the output only depends on the present state, see Figure 7.2.

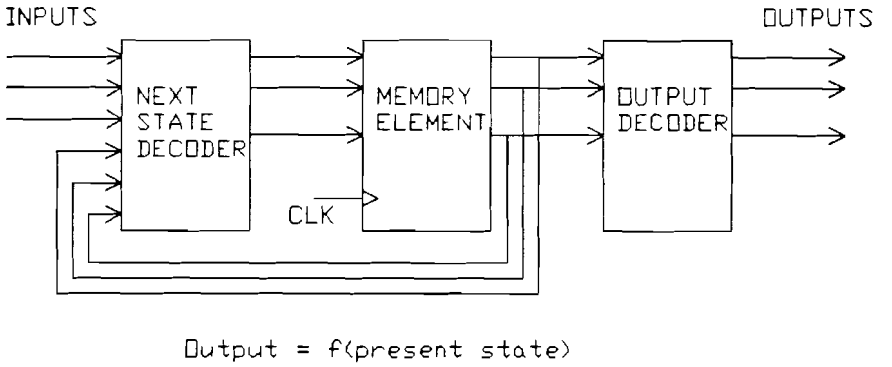


Fig. 7.2. Moore-type implementation of a control unit.

In the examples in this report Moore-type implementations of control units have been used.

7.3. Microprogramming

By using an instruction controlled controller the controller becomes very flexible.

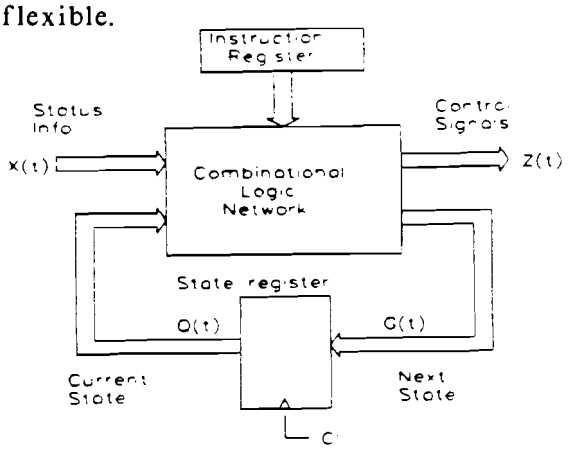


Fig. 7.3 Instruction controlled controller.

Figure 7.3 shows a basic form of such an instruction controlled controller. The instruction sequence is called a "micro program". By loading an instruction in the instruction in the instruction register the controller will perform a function. In cooperation with the status signals supplied by the data unit, the next instruction can be loaded. This approach can lead to a full micro-controller, which basically has the same structure.
This kind of controller has not been used in the examples in this report, since the primary goal of the research has been to implement algorithms with a minimum amount of hardware. This kind of controller will lead to an overcapacity of the hardware.

7.4. Pipelining

If data is delivered in a stream form and the system has to perform the same computation on every element of the stream, there are methods to increase the speed of the system. If we use the implementations discussed until now, each instance of the computation would begin after the previous one has finished.
A method to increase the speed is called "pipelining".
The data units performing a step of a computation all work in parallel. If the time to compute in a step is called 'T', then after the first computation has been done, after every time 'T' a result is delivered.
In more sophisticated systems there could also be a piece of controller in each stage. This could be used when the processing time per stage is not equal for all stages. Then we need communication between the control units (Figure 7.4) [2].

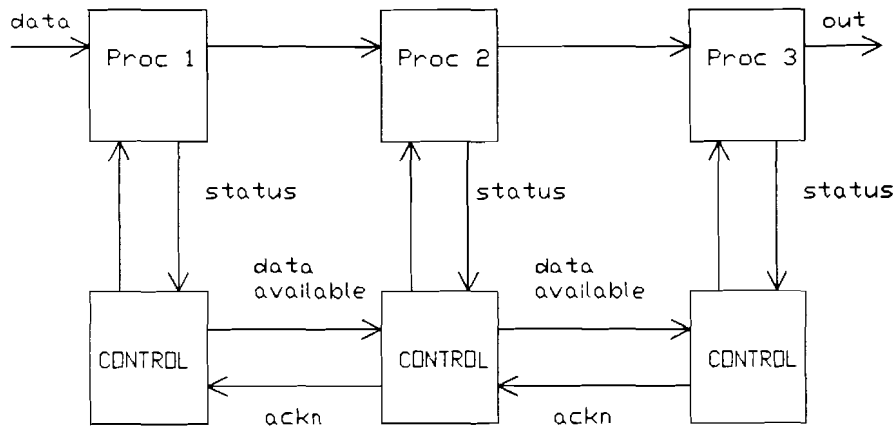


Figure 7.4 Pipelined system with decentralized control.

7.5. Conditional instruction modules

Until now we have seen implementations with a centralized controller. As said before it is also possible to put a part of the controller in the data path.A general diagram of a controller could be like Figure 7.5

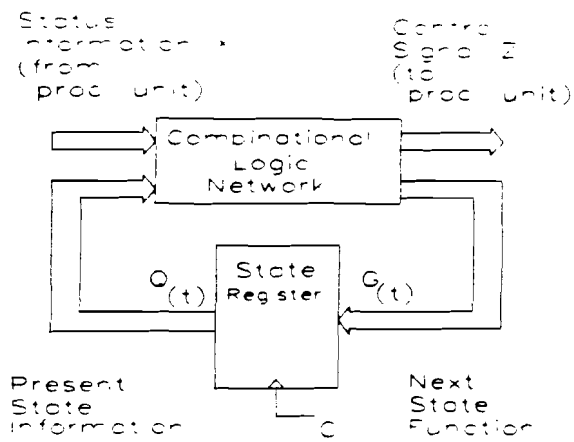


Fig. 7.5 General diagram of a control unit with conditional instructions

Now, let's take a closer look at conditional instructions like:

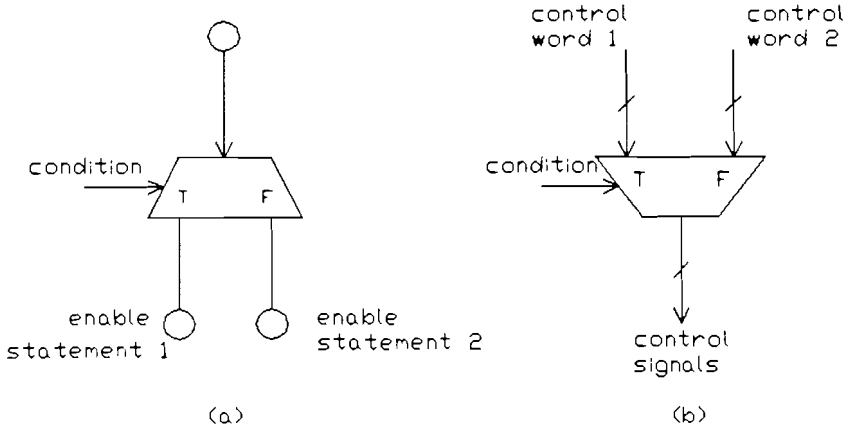
while {condition} do {statement}
if {condition} then {statement1} else {statement2}
for {condition} do {statement}
repeat {statement} until {condition}

Conditions and statements can be calculated and executed by the processing part. But branching has to be done by the control unit since the control signals depend on the conditions.

The processing unit will supply a status after having calculated the condition. This status signal is used to calculate the next state.

7.5.1.If..then..else

This statement can easily be implemented by using a multiplexer. There are two possibilities to use such a multiplexer. The control word can be switched, or a flow token (bit) can be switched. Figure 7.8.1.a shows token flow model; Figure 7.8.1.b shows the control word switching method. Normally this function is integrated in the state machine itself. The next state depends on the status signal of the processing unit. For this reason only the token flow switching will be used in examples to implement an 'if..then..else' statement alternatively.



if {condition} then {statement1} else {statement2}

Fig. 7.8.1 Implementation of if() then () else(). (a) token flow switching. (b) control word switching.

7.5.2.CASE

Now that we have a implementation for an 'if..then..else' statement, the implementation of a 'case' statement is easy. Figure 7.8.2. shows a diagram. The token flow model and the control word switching model are given. Again only the token flow switching will be used in examples.

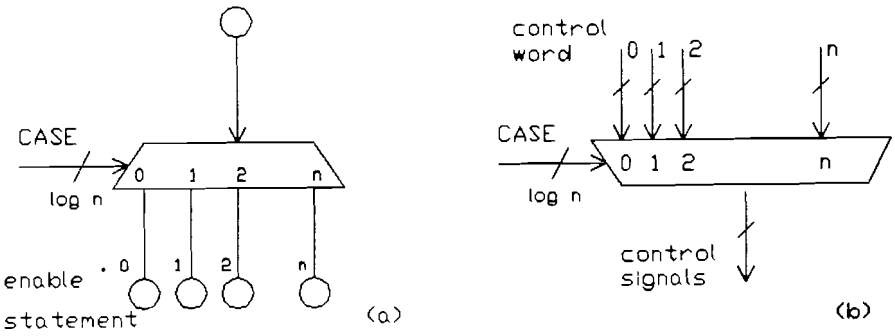


Fig.7.8.2 Implementation of a 'CASE' statement. (a) token flow switching. (b) control word switching.

7.5.3.FOR loops

Incremental 'for' loops are of the same type.
A choice can be made to put the counter into the processing unit or the control unit.
To reduce the communication overhead, the loop counter will be placed in the control unit.A general diagram could be like Figure 7.8.3.a
The module can be pre-loaded with the begin and end values of the counter. The value of the counter (i) is available on the outside.

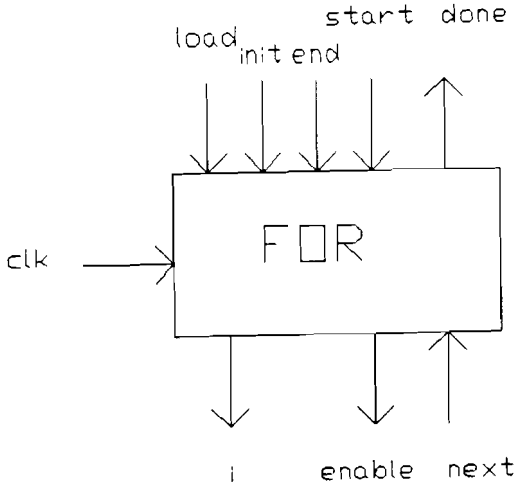


Fig. 7.8.3.a For {init condition} to {end condition} do Module

Figure 7.8.3.b shows the inside of the module. There are to steps to be executed in a 'for' module. The module is started by making the 'enable for' signal high.
The first step is to compare the counter (i) and the end condition register. These where pre-loaded with start values. If the condition is false, then the enable process signal becomes high.
After a next signal from the enabled process the flip-flop will toggle and step 2 is executed: the counter is decremented.

If the counter has the same value as the end condition register the enable process signal becomes low and the done signal becomes high.

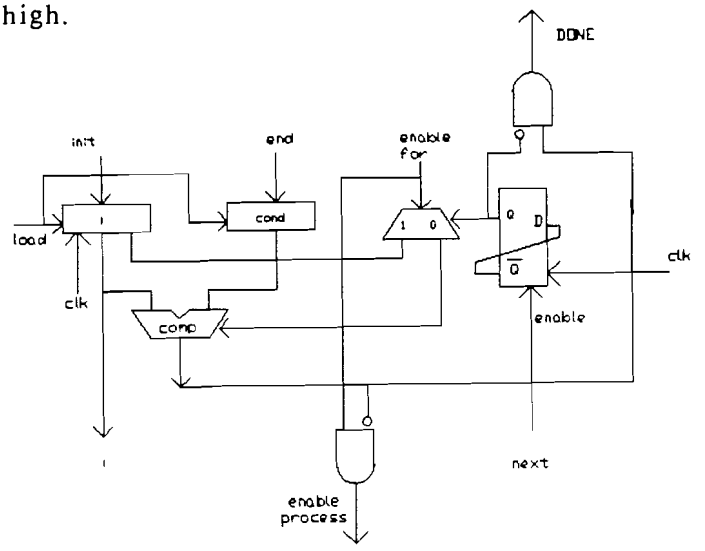


Fig.7.8.3.b For {init condition} to {end condition} do implementation.

Figure 7.8.3.c shows an implementation of the comparator

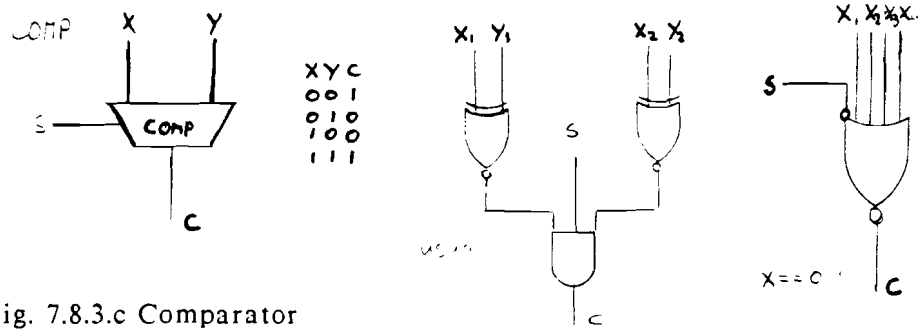


Fig. 7.8.3.c Comparator

7.5.4.WHILE

If we want to implement a 'while' statement, we need status information of the data unit. For example a comparator as shown in fig. 7.8.2.c can be used to calculate conditions. Using the token flow method we can again use the multiplexer to continue or stop the while loop depending on the condition.
Of course the enable process signal can be expanded to a series of enable signals to implement more statements inside the while loop.

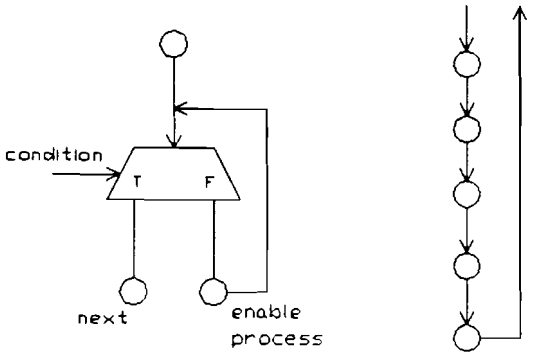
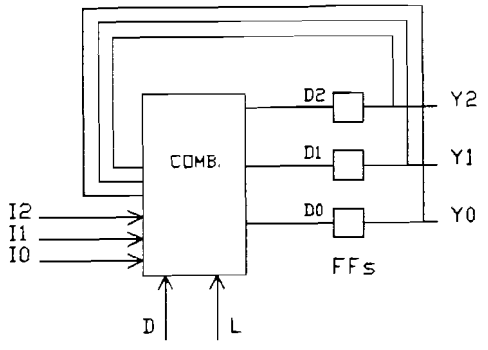


Fig. 7.8.4. (a) WHILE implementation (b) expansion of the enable signal

7.6.Decremental register

In the last section we have used a decremental register. An implementation of such a register could be like Fig. 7.9.0. This is a Moore-type implementation.
When using the bit-slice method, we obtain modules to implement a decremental register of any size.
In both cases the L input is used to load an initial value; the D input is used to enable decrementing the register.
To come to an nit-slice implementation we first make a general subtractor.



LOAD = $L\overline{D}$
Decrement = $\overline{L}D$
NOP = $\overline{L}\,\overline{D}$

$$D_2 = (\overline{D}Y_2 + Y_2Y_0 + Y_2Y_1 + \overline{Y}_2\overline{Y}_1\overline{Y}_0D)\,\overline{L} + I_2L\overline{D}$$

$$D1 = (Y_1Y_0 + \overline{D}Y_1 + D\overline{Y}_1\overline{Y}_0)\,\overline{L} + I_1LD$$

$$D_0 = \overline{Y}_0\overline{L}D + I_0L\overline{D}$$

Fig. 7.9.0 Decremental register (Moore type)

7.6.1.General subtractor

Using the bit-slicing method we need a unit that can subtract two bits using a borrow-in (BI) and generating a result (R) and a borrow-out (BO), see Figure 7.9.1.a.

Fig. 7.9.1.a Symbol of an subtractor-bit

In Figure 7.9.1.b the state table is given.

X	Y	BI	R	BO
0	0	0	0	0
0	0	1	0	1
0	1	0	1	1
0	1	1	0	1
1	0	0	1	0
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

$$R = \overline{BI}\,(\overline{X}Y + X\overline{Y}) + XYBI$$

$$BO = YBI + \overline{X}Y + \overline{X}BI$$

Fig. 7.9.1.b State table subtractor

The next figure shows the implementation of one element of the subtractor.

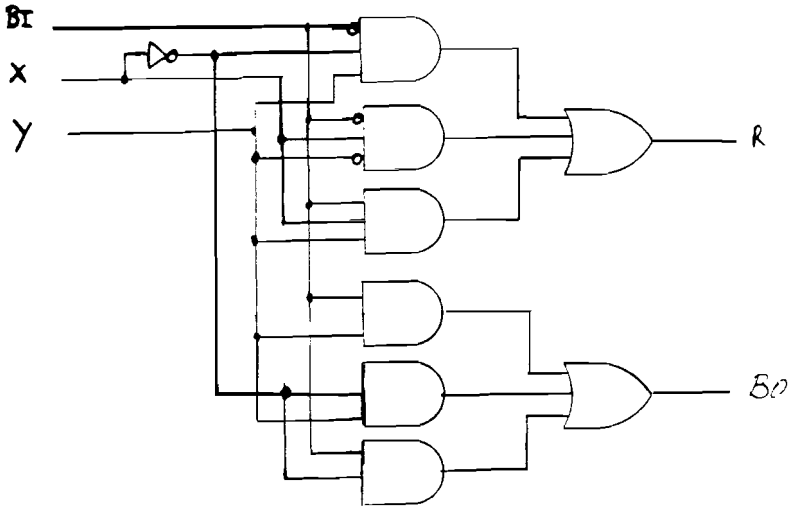


Fig. 7.9.1.c Implementation of a subtractor-bit element

7.6.2.Decremental register (bit-slice)

Now that we have designed a subtractor element it is easy to design a decremental register. Using these elements the value of Y is 1 for the first element and 0 for all others, see Figure 7.9.2.a.

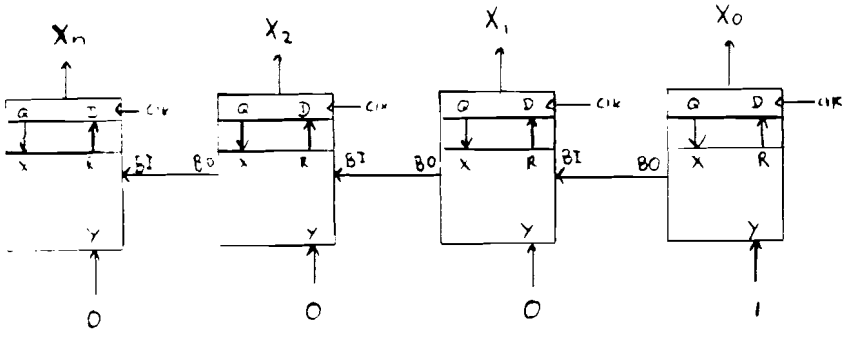


Fig. 7.9.2.a Decremental register, bit-slice method.

The subtractor element of the last section can be used to implement the decremental register. Substitution of the Y values gives the following equations:

Element X_0: $R = \overline{X}$
$BO = \overline{X}$
$BI = 0$
$Y = 1$

Element X_n: $R = \overline{BI}X_n$

$BO = \overline{X}_nBI$

LOAD = $L\overline{D}$
Decrement = $\overline{L}D$
NOP = $\overline{L}\overline{D}$

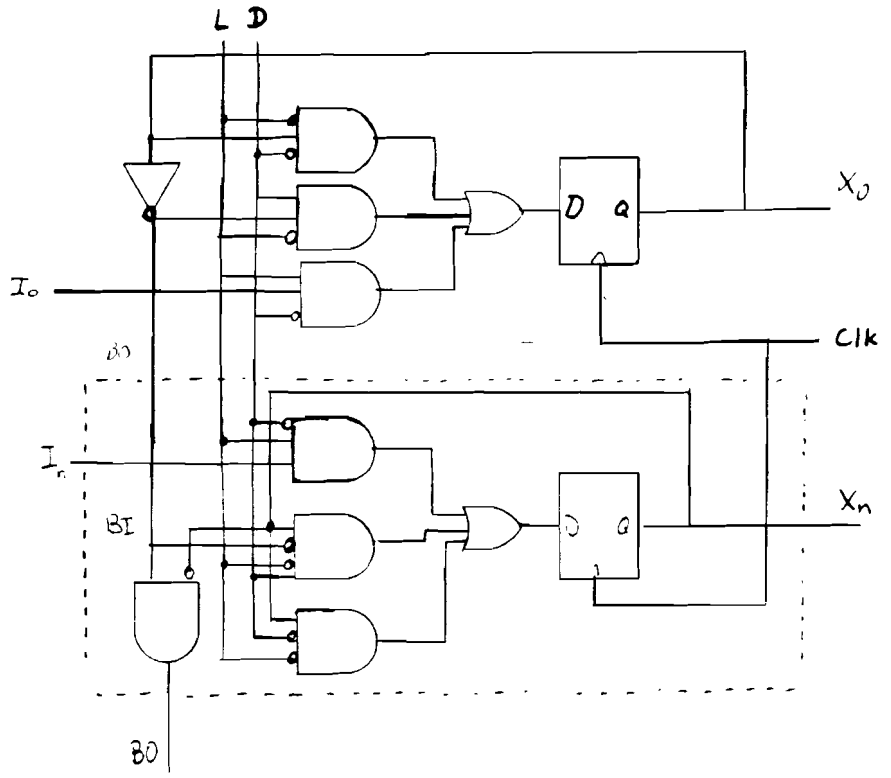


Fig. 7.9.2.b Implementation of the decremental register, bit-slice method.

8.DATA PATH SYNTHESIS

The input to the synthesis system is a code sequence, written in the RTL language.
This sequence contains variables and operators. We now want to reduce the number of registers, operators and their interconnection, and then generate the hardware design by a mapping procedure.

A basic block is a linear sequence of operation codes having one entry point (the first operation executed) and one exit point (the last operation executed). A code sequence generally contains a number of basic blocks linked by conditional or concurrent control constructs. For example see Table 1; this is a code sequence which consists of a single basic block.
Code statements appearing on the same line are executed in parallel by a single control step.
This code sequence, adopted from [8], will be used as an example illustrating the synthesis procedure.

TABLE 1 Code sequence

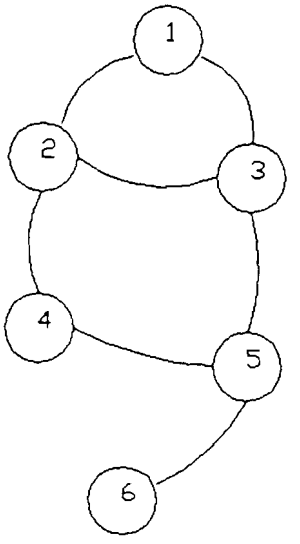
```
1. R2  = R1 + R2 ;     R12 = R1
2. R5  = R3 - R4 ;     R7  = R3 * R6 ;   R13 = R3
3. R8  = R3 + R5 ;     R9  = R1 + R7 ;   R11 = R10 / R5
4. R14 = R11 AND R8 ;  R15 = R12 OR R9
5. R1  = R14
```

The resources for data paths are registers, data operators, and interconnection units. Using the code sequence as input, data paths synthesis involves three tasks: assigning a variable to a suitable number of registers, data operators, and interconnection units. Two variables can be assigned to the same physical resource if and only if there is no conflict in the use of the two resources. Of course this rule can be expanded to N variables. (N > 2)
N variables can be assigned to the same physical resource if and only if each pair of these N variables does not have usage conflict. Let each variable be represented as a node in an undirected graph. The relationship of shareability can be represented by the connectivity of the two nodes. Mapping of registers, data operators, and interconnection units can all be formulated into the clique-partitioning problem whose mathematical definition is presented now.

8.1.Clique partitioning

Let G be a graph consisting of a finite number of nodes and a set of undirected edges connecting pairs of nodes. A nonempty collection C of nodes of G formes a complete graph if each node in C is connected to every other node of C. A complete graph C is said to be a clique with respect to G if C is not contained in any other complete graph contained in G. The clique partitioning problem is to partition the nodes in G into a number of disjoint

clusters such that each node appears in one and only one cluster. Furthermore, each of these clusters itself forms a complete graph (clique). See fig. 8.1. This figure shows an undirected graph. The nodes are variables which must be combined. There are more than one solutions to this problem. The following clusters can be formed: (1,2,3) , (4,5) , and (6). An other solution could be : (1,3) , (2,4) , and (5,6).



The search for cliques in a graph has been proved to be NP-complete.[3,4]
In Appendix A algorithms can be found to find cliques in a undirected graph.

8.2.Allocation of registers

As indicated, it is generally beneficial to assign more than one variable to the same physical registers. This section describes how to minimize the number of registers.
Given a set of variables of a code sequence, the problem of storage allocation is to combine those variables which can share a register. Two variables can be combined if they have disjoint lifetimes. A variable is live between the time of its definition and last use. A variable is dead between the time of its last use and the next definition. If the live periods of two variables are not overlapped, they have disjoint lifetimes.
In reality this constraint can be relaxed. Two variables X and Y can be combined if their lifetimes are overlapped in such a way that one of them is used as a source and the other is used as a destination or vice versa in the same statement. In addition, the variable which is used as the source is dead in the next time interval, i.e., the use is a "last use". Lifetime analysis is an essential process for constructing the table which identifies the constraints of storage sharing. Automatic lifetime analysis has not been solved in this report. More about solutions can be found in literature about compiler design. In the examples described in this report, lifetime analysis has been done by hand.
Pure data transfers are special cases.

8.2.1.A procedure for combining variables.

If there are n variables and each pair of variables are proved to be combinable (there are n(n-1)/2 different pairs), then these n variables can be assigned to the same physical register. Let the nodes of a graph be the variables and each pair of nodes which can be combined be joined by an edge. Then a graph which contains the lifetime relationship among all the variables can be constructed. In the method used this information is transformed to a table. Since the goal is to assign these variables to the minimum number of physical registers, this is translated into the clique-partitioning problem.
The combination of each pair of variables which are related by pure data transfers would cause these operations to be eliminated. This improvement reduces the number of control functions. If a horizontal list in the code sequence is occupied by pure data transfers, the control step can be deleted resulting in a faster implementation. To take this property into account, during clique partitioning with the cp-algorithm version 2, one can indicate which variables belong to pure data transfers. The program will give these variables a higher priority to be combined.
Once the variables have been compacted, the code sequence is updated. The names of variables which are grouped together are assigned to the same name. Then operations of moving the content of a variable to itself are deleted.

8.2.2.Construction of the compatible table

The live/dead status of all variables are represented by a lifetime list. The compatible table is the table which contains a "1" if variables are combinable and a "0" if not. To construct a compatible table the lifetime list and the code sequence are traced and inspected. Variables are compared two by two (the columns are compared). Unless the conditions given above are satisfied, the crosspoints combining variables which are live in the same time interval are marked "0". Combinable variables are marked "1". Those pairs which associate with pure data transfers in some time intervals are marked "2".

8.2.3.Grouping registers into scratch pad memories

Having assigned all the variables to suitable physical locations, the next step is to investigate the possibility of grouping several registers into sets of scratch pad memories. Those variables which have disjoint access time can be grouped together. This allocation problem has been worked out in detail by M. Balakrishnan et al. [5].

8.2.4.Example

Let the code sequence of Table 1 be given. Assume that the program is itself a loop. Having executed the statements in the last

line, the control flow is passed back to the statements in the first line. Applying the lifetime analysis algorithm to the example, this must be done by hand, the status of these variables in each time interval is indicated in Table 2.

TABLE 2 Lifetime trace

Time	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10	R11	R12	R13	R14	R15
Entry	1	1	0	1	0	1	0	0	0	1	0	0	0	0	0
1	1	1	1	1	0	1	0	0	0	1	0	1	0	0	0
2	1	0	1	1	1	1	1	0	0	1	0	1	0	0	0
3	1	0	1	1	1	1	1	1	1	1	1	1	0	0	0
4	0	0	0	1	0	1	0	1	1	1	1	1	0	1	1
5	1	1	0	1	0	1	0	0	0	1	0	0	0	1	1
Exit	1	1	0	1	0	1	0	0	0	1	0	0	0	0	0

Having derived the life/dead history of each variable, the compatible table can be constructed. This table called "CONNECT". Two variables are compared for every point in time (so the columns are compared). If two variables do not have usage conflict, their crosspoint will be marked by "1" in the table "CONNECT". If not so, the crosspoint will be marked by "0". The nodes R2 and R3 are used as a source and destination in the statement. In addition, the source variable is dead in the next time interval. Therefore, the crosspoint (2,3) is marked "1". Repeatedly applying the procedure to the entire lifetime, the resulting compatible table is given in Table 3.

TABLE 3 Compatible variable pairs

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	1								1				1	2	
2		1						1	1		1		1		2
3			1										2	1	1
4				1									1		
5					1			1			1		1	1	1
6						1							1		
7							1		1				1	1	1
8								1					1	1	
9									1				1		1
10										1			1		
11											1		1	1	
12												1	1		1
13													1	1	1
14														1	
15															1

Among these compatible variable-pairs, those pairs associated with pure data transfers in some time intervals are marked with "2" (by hand). Applying the cp-algorithm, the variables are partitioned into eight clusters. They are:

clique = 2 7 9 15
clique = 3 8 13
clique = 1 14
clique = 5 11
clique = 4
clique = 6
clique = 10
clique = 12

The variables in each of these clusters can be assigned to the same physical location. The code sequence in Table 1 can be redefined into the form in Table 4.
There are fifteen variables in the original code sequence. They have been compacted into eight. Furthermore, the last step has been eliminated. The program can now be realized in four steps.

TABLE 4. Improved Code sequence

1. R3 = R1 + R2 ;	R12 = R1	
2. R5 = R3 - R4 ;	R2 = R3 * R6	
3. R3 = R3 + R5 ;	R2 = R1 + R2 ;	R5 = R10 / R5
4. R1 = R5 AND R3 ;	R2 = R12 OR R2	

8.3.Allocation of data operators

If an operation is used more than once grouping those operators to one reduces the hardware needed to implement the sequence. This can be done if the operators are not used at the same time.
It is also possible to combine operators performing different operations into an ALU.

8.3.1.Grouped operator categories

An important issue for the allocation of data operators is choosing an appropriate set of operators to group together. In order to minimize the number of multiplexers and demultiplexers, not only the operations must be compared but the relation between their sources and destinations as well. By inspecting the relationship between two operations, there are sixteen cases. These sixteen cases can be classified into eight categories [8]. They are listed below.

C8 The operations and the three pairs of variables are all the same
C7 The operations are different but the three pairs of variables are the same
C6 The operations and two pairs of variables are the same. The third pair of variables is different
C5 Two pairs of variables are the same. The operations and one pair of variables are different.
C4 The operations and one pair of the variables are the same. The other two pairs of variables are different

C3 One pair of the variables is the same. The operations and the other two pairs of variables are different
C2 THe operations are the same. All three pairs of variables are different.
C1 The operations and all three pairs of variables are different

The algorithm for allocating data operators can be described as follows:
1. Trace through the code sequence and mark the crosspoints of combinable operators with the category number and non-combinable operators with "0" (operators used simultaneously).
2. Collect crosspoints of category 8. Use the cp-algorithm to reduce table G and table C8.
3. Having reduced the table C8, table G together with the subtables of categories 7, 6, 5, 4, 3, 2, and 1 are reduced one by one.

8.3.2.Example

Let each operation in Table 4 be assigned to a specific name. An assignment is given in Table 5. Using the code sequence and operator assignment, the compatible table G is shown in Table 6. The integer in the matrix identifies the category of the pair.

TABLE 5 Operator identifiers

$+_1$	$-_1$	$*_1$	$+_2$	$+_3$	$/_1$	AND_1	OR_1
1	2	3	4	5	6	7	8

The crosspoints of category 6 in G is retrieved to form the subtable C6. C6 only consists of one pair; it is (1,5). The original table G and the subtable C6 are reduced. The variables 1 and 5 are combined. In the reduced table the categories of the pairs (1,3) and (1,8) are updated to 3 and 5 respectively. The reduction procedure is continued until the tables becomes empty. The data operators are finally grouped into three clusters. They are: (1,3,5,8) , (2,4,7), and (6).

TABLE 6. Compatible pairs categories

G

OP	1	2	3	4	5	6	7	8
1	1	1	4	6	1	1	3	
2			3	1	1	3	1	
2			3	3	1	3	3	
4						3	1	
5						1	5	
6						3	1	
7								
8								

C6

OP	1	2	3	4	5	6	7	8
1						6		
2								
2								
4								
5								
6								
7								
8								

clique = 1 5

G

OP	1	2	3	4	5	6	7	8
1	1	3	4			1	1	5
2			3	1	1	3	1	
2			3	3	1	3	3	
4						3	1	
5						1	5	
6						3	1	
7								
8								

C5

OP	1	2	3	4	5	6	7	8
1								5
2								
2								
4								
5								5
6								
7								
8								

clique = 1 8
clique = 5 8

This procedure has to be continued until category C1 has been partitioned.

Total result:
clique = 1 3 5 8 (forms ALU1)
clique = 2 4 7 (forms ALU2)
clique = 6 (forms ALU3)

8.4.Allocation of interconnection units

The procedure to reduce the amount of interconnection units involves two steps:
- alignment of operands
- allocation of interconnection units.

An operation may be either commutative or noncommutative. For commutative operations the position of the two operands can be flipped, in order to save multiplexers. Alignment of operands decreases the number of interconnection units.
Interconnection variables which are never used simultaneously can be grouped together to form buses. The goal is to group the interconnection variables into the minimum number of clusters. The problem is again formulated into the clique-partitioning problem.

Trying to reduce the number of multiplexers and demultiplexers, during optimization interconnection variables with the same source and variables with the same destination have a higher priority to be combined then other interconnection variables.
There is no profit in combining two interconnection variables which originate from different sources and connect to different sinks, since this will introduce more multiplexers. On the other hand, it is generally beneficial to group those interconnection variables which originate from the same source or connect to the same sink to share a common bus. The details of the formulation are given below.
A lifetime trace is constructed. Then the code sequence is traced through. The columns of variables are compared two by two. If in a time interval two interconnection variables are used simultaneously the crosspoint formed by these two variables is set to "0", else it is set to "1". Those interconnection variables which are associated with the same source, even when they are used concurrently, can still share a common interconnection. Therefore, when the compatible table is constructed, these crosspoints can be set to "1". Crosspoints combining variables with the same source or connect to the same sink, are marked with "2". Then the cp-algorithm is applied.
If more than one bus is connected to a single input port, a multiplexer in front of the input port must be inserted.

8.4.1.Example

The example is based on the code sequence in Table 5 and the ALUs allocated in section 8.3. Inspecting the operands of the operations associated with ALU2, it is found that the positions of the two operands of the statement "R1 = R5 AND R3" can be flipped to the form of "R1 = R3 AND R5". Using the indices in Table 7 the compatible table (G) in Table 8 is constructed.

Table 7. Indices of interconnection variables

source	destination	index
R1	R12	1
R1	ALU1.In1	2
R2	ALU1.In2	3
R3	ALU1.In1	4
R3	ALU2.In1	5
R4	ALU2.In2	6
R5	ALU2.In2	7
R5	ALU3.In2	8
R6	ALU1.In2	9
R10	ALU3.In1	10
R12	ALU1.In1	11
ALU1.Out	R2	12
ALU1.Out	R3	13
ALU2.Out	R1	14
ALU2.Out	R3	15
ALU2.Out	R5	16
ALU3.Out	R10	17

Table 8.Compatible interconnection pairs

INT	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
1		2		1	1	1	1	1	1	1	1	1		1	1	1	1
2				2		1			1		1			1		1	
3				1		1			2							1	
4					2		1	1		1	2		1	1	1		1
5													1				
6							2	1		1	1		1	1	1	1	1
7								2	1				1			1	
8								1		1			1	1		1	
9									1	1			1	1	1		1
10										1			1	1		1	
11													1		1	1	1
12												1					
13														1	2	1	1
14															2	2	1
15																1	
16																	1
17																	

In Table 8 those crosspoints of compatible interconnection variables with the same source or the same sink are marked with a "2". In version 2 of the cp-program this will give them a higher priority. So nodes 14 and 16 are combined. We obtain the following clusters:

clique = 13 14 15 16 (forms MUX1)
clique = 1 2 4 11 (forms MUX2)
clique = 6 7 8 (forms MUX3)
clique = 3 9 (forms MUX4)
clique = 5
clique = 10
clique = 12
clique = 17

Now the complete processing unit can be drawn.
Figure 6.3.1 shows the completed allocation of the data part.

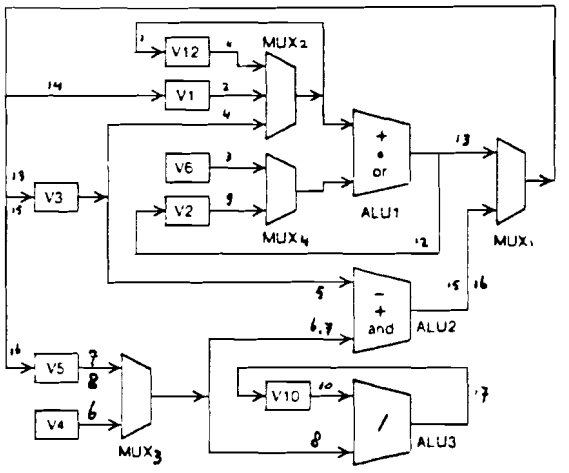


Fig. 8.4.1. Data processing unit of the example

8.5.Implementation

As indicated in the first section, the input to the synthesis is a value trace.
For the operator allocation only the cp-algorithm can be used: composing the special subtables C8..C1 has to be done by hand.
For the interconnection allocation alignment has to be done by hand. Also creating the matrix containing the compatible pairs (crosspoints) has to be done by hand. Then the cp-algorithm can be applied.
Sometimes a lot of clusters (cliques) are generated. It is difficult to supply rules for picking out the minimum covering set. The problem can be compared to the covering problems of maximum compatibles in state minimization strategies. This could be a point of research.

At the Carnegie Mellon University a similar system called "Emerald" has been developed. More of the traject has been implemented, but for the clique-partitioning problem an other algorithm has been used, which is a lot slower than the algorithm used in this report, see [8].
In the next examples, when finding more clusters then desired, a set is picked out applying two rules:
- if an node only can be found in one clique, always pick this clique.
- try to find clusters that cover other (smaller) clusters.

8.6.Example. Implementation of the abc formula

To demonstrate the possibilities of the synthesis method the abc formula is implemented in hardware.

Formula:

$$x1 = (-b + \sqrt{b2-4ac})/2a$$
$$x2 = (-b - \sqrt{b2-4ac})/2a$$

In RTL there are many ways to describe this formula. We give two possible methods. The first method tries to reduce the number of ALUs, the second method tries to perform as many concurrent statements as possible.
Let us describe the algorithm in RTL without taking care of the number of registers. Every time we think we need a new register we use it, without trying to be too smart to save registers, since the reduction of the number of registers is the task of the cp-algorithm.

Code sequence 1:

```
1. E = 4 * A
2. F = E * C
3. G = B
4. H = B * G
5. D = H - F
6. I = SQRT(D)
7. K = 2 * A
8. L = I - B
9. M = -I
10. N = M - B
11. O = L / K
12. P = N /K
```

Code sequence 2:

```
1. E = 4 * C ; G = B
2. F = E * C ; H = B * G
3. D = H - F
4. I = SQRT(D) ; K = 2 * A
5. L = I - B ; M = -I
6. N = M - B ; O = L / K
7. P = N / K
```

We now use the cp-algorithm to reduce the number of registers in both sequences.
The trace table and the other tables produced during the session can be found in Appendix B.
After reduction with the cp-program we find:

Reduced code sequence 1:

```
1. E = C * 4
2. E = C * E
3. C = B
4. C = C * B
5. C = C - E
6. C = SQRT(C)
7. A = 2 * A
8. E = C - B
9. C = -C
10. C = C - B
11. B = E / A
12. C = C / A
```

Reduced code sequence 2:

```
1. E = 4 * A ; F = B
2. E = E * C ; C = B * F
3. C = C - E
4. A = SQRT(C) ; C = 2 * A
5. F = A - B ; A  = -A
6. B = A - B ; A = F / C
7. B = B / C
```

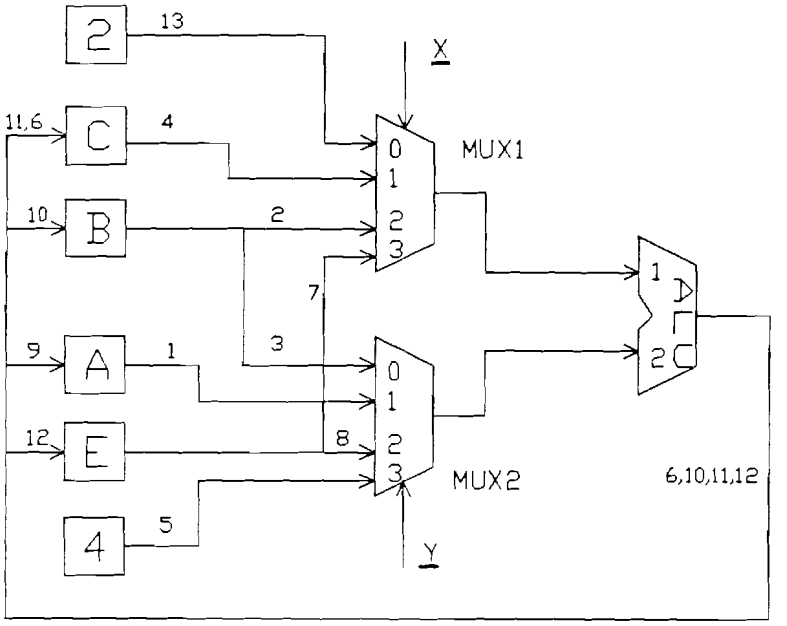


Fig. 8.6.1.a Processing unit of code sequence 1.

ALU :	*	/	SQRT	+	-	▬	transparent
CODE:	1	2	3	4	5	6	0

T	LOAD A B C E	MUX1 X_2X_1	MUX2 Y_2Y_1	ALU $Z_3Z_2Z_1$	Istruction
1	0 0 0 1	0 1	1 1	0 0 1	E = C * 4
2	0 0 0 1	0 1	1 0	0 0 1	E = C * E
3	0 0 1 0	1 0	X X	0 0 0	C = B
4	0 0 1 0	0 1	0 0	0 0 1	C = C * B
5	0 0 1 0	0 1	1 0	1 0 1	C = C - E
6	0 0 1 0	0 1	X X	0 1 1	C = SQRT(C)
7	1 0 0 0	0 0	0 1	0 0 1	A = 2 * A
8	0 0 0 1	0 1	0 0	1 0 1	E = C - B
9	0 0 1 0	0 1	X X	1 1 0	C = ▬C
10	0 0 1 0	0 1	0 0	1 0 1	C = C - B
11	0 1 0 0	1 1	0 1	0 1 0	B = E / A
12	0 0 1 0	0 1	0 1	0 1 0	C = C / A

Fig. 8.6.1.b Control table of code sequence 1.

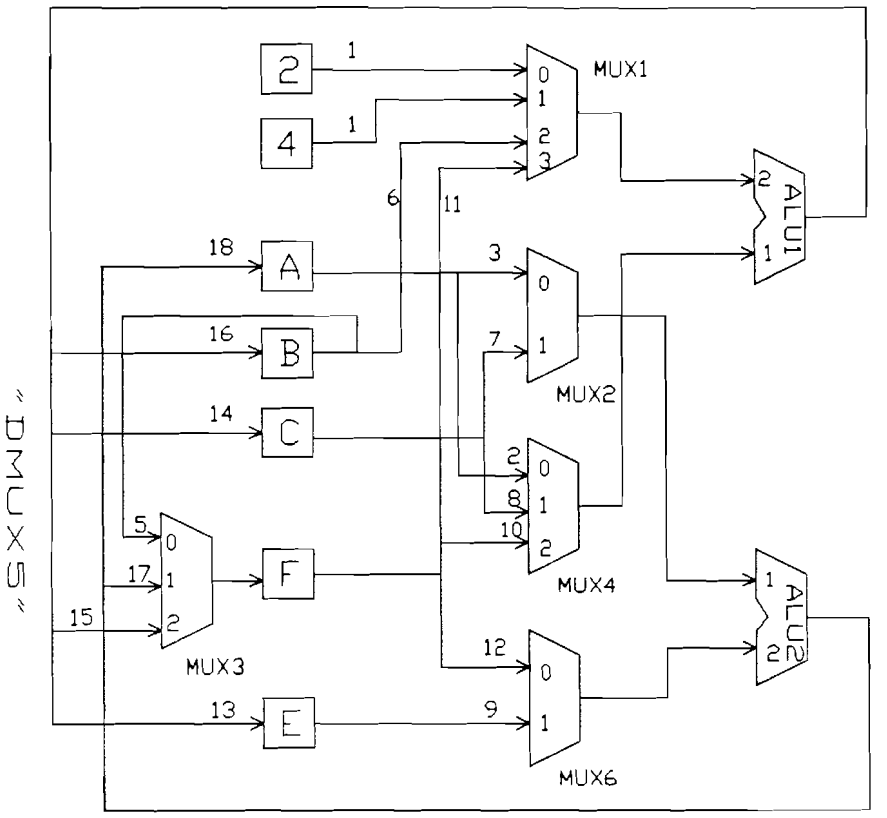


Fig. 8.6.2 Processing unit of code sequence 2.

This is an example of a typical space-time tradeoff in the implementation of digital systems: the faster implementation requires more components and, therefore, occupies a larger space.

8.7.Conclusion

The suggested method for minimization of registers, operators and interconnection units out of a code sequence has been proven to deliver nice results. The whole traject could be automated. In this report only the clique-partitioning problem has been translated to an algorithm in the C-language (see Appendix A). The cp-algorithm can be used to find clusters containing maximum compatible registers, operators or interconnection units very fast. Sometimes it is hard to find the minimum covering set.

9.IMPLEMENTATION OF HIGH LEVEL ALGORITHMS

9.1.Methods

Until now we have seen implementations of algorithms described in RTL.
Now we want to implement algorithms described in a higher level programming language.
To break down an algorithm into blocks that can be implemented by structures from a library we can use two methods.
The first method uses a 'normal' compiler. The code obtained is a kind of assembly, but with typical RTL instructions.
The second method searches for conditional blocks, in order to implement each block as a state machine.

9.1.1.METHOD 1

1. input and output variables obtain their own register.
2. the program must be compiled. Conditional statements, like 'for','while', 'if then else', and 'case' must all be translated to RTL branch instructions.
3. Optimize registers, ALUs, datapaths, and states.

9.1.2.METHOD 2

1. input and output variables obtain their own register.
2. search for the following conditional statements:
   - for
   - if
   - while
   - case
3. These statements form special blocks which must be implemented as a unit. Some of them can be combined. To do so we must examine the variables inside the blocks. Two blocks containing the same variables cannot be implemented as parallel blocks. They must be evaluated sequentially. The advantage is that the hardware for implementing the first block can also be used by the second since the conditional statements consume quiet a lot of chip area.
   Two blocks that do not contain the same variables can be processed in parallel.
   As usual velocity and chip area can be exchanged.

4. When the units have been chosen, further optimization of the blocks can take place.
5. Optimize registers, ALUs, datapaths, and states.

If we want to use method 2, we need special hardware implementable blocks to perform the conditional statements. In chapter 4 we have already seen a general description of such a statement implementation.

9.2.Examples

We now give some simple algorithms and their implementation in hardware. The results of the cp-algorithm used for minimization can be found in Appendix B.

9.2.1.Example 1. A receiver for serial reception of packages with variable length.

We want to design a receiver for packages of 8, 7, 6, or 5 bits.
The receiver must be set in the right mode in advance.
The packages are transmitted in a serial way, LSB first.
After reception, the package must be available in parallel form. If the package is smaller than 8 bits, the rest of the receiving register is filled with 0's.

The following algorithm describes the function:

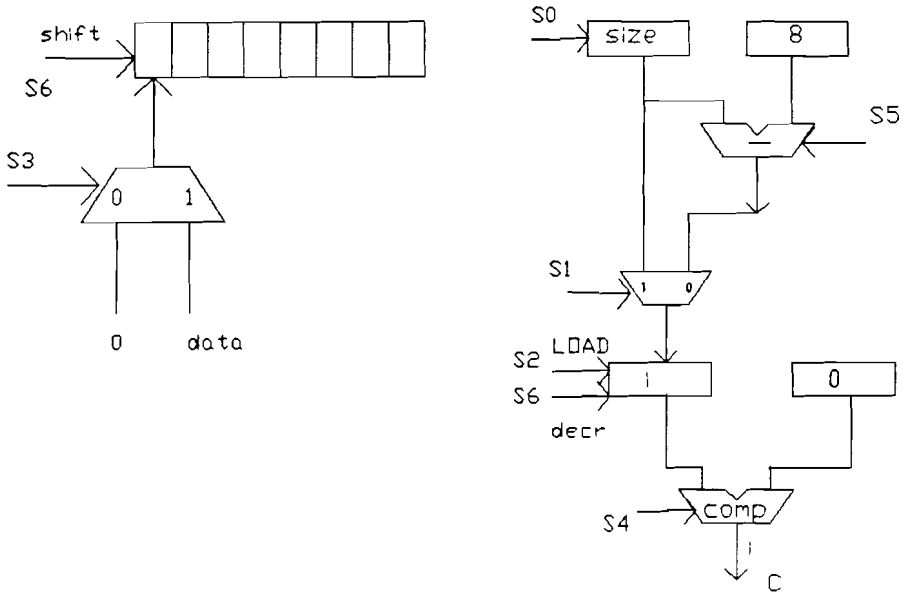
■ Algorithm 1

```
int size;          /* size of a package */
int reg[8];        /* register for serial in; parallel out*/
int i;

input size;

for (i= size; i > 0; i--)
    {
      reg >> 1;          /* shift  */
      reg[7] = data;
     };
for (i= 8 - size; i > 0; i --)
     {
       reg >> 1;         /*  shift    */
       reg[7] = 0;
     };
```

When we use method 1 we obtain the following RTL:

```
1. size = input
2. i = size
3. shift(reg); reg[7] = data; decr(i);
4. if (i == 0) continue else -> 3
5. i = 8 - size
6. shift(reg); reg[7] = 0; decr(i);
7. if (i == 0) continue else -> 6
```

In Appendix B more about the results obtained during the design process can be found. Figure 9.2.1.1 shows the final result.
Remark: Filling the register with zeros this way could lead to loss of channel capacity, unless special clocking is used.



Q^v	C		$S_6S_5S_4$ S_3S_2 S_1S_0	$Y_2Y_1Y_0$	C	
	0	1			0	1
0	1	1	0 0	0 0 0	0 0 1	0 0 1
1	2	2	0 0 1	0 0 1	0 1 1	0 1 1
2	3	3	0 1 1 0	0 1 1	0 1 0	0 1 0
3	4	4	1 1 0 0	0 1 0	1 0 0	1 0 0
4	3	5	0 1 0 0	1 0 0	0 1 0	1 0 1
5	6	6	0 1 1 0 0	1 0 1	1 1 1	1 1 1
6	7	7	1 0 0 0	1 1 1	1 1 0	1 1 0
7	6	2 Q^{v+1}	0 1 0 0	1 1 0	1 1 1	0 1 1
				Present	Next	

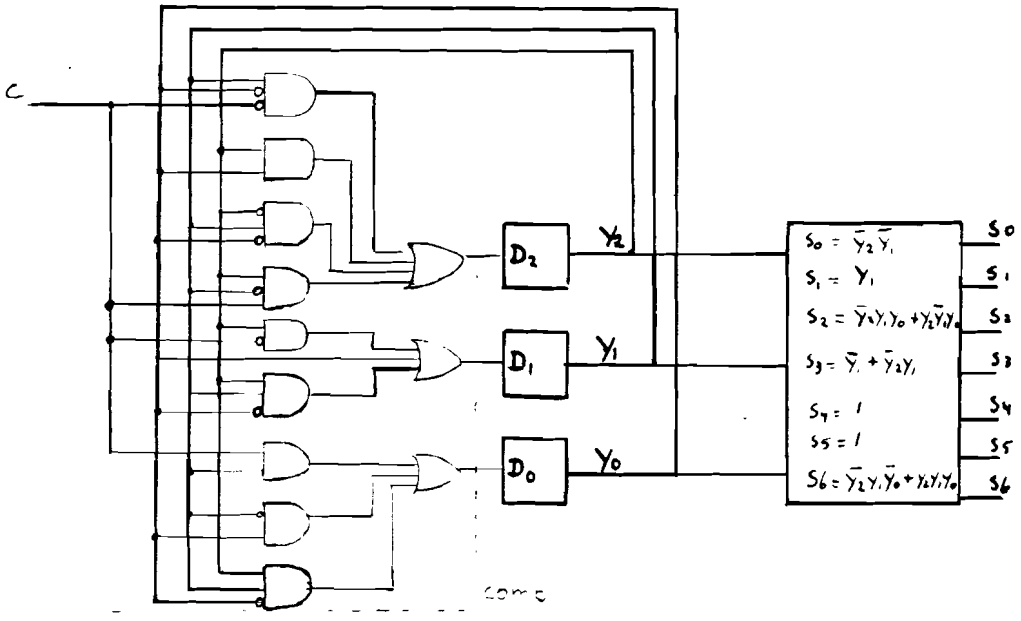


Fig. 9.2.1.1. Implementation of Alg.1 using method 1 (a) data unit (b) state table (c) controller

Examining the algorithm one finds that the for loops cannot be evaluated parallel, but that the structure is similar. This means that the same piece of hardware can be used for these loops.
We also need a register to store the size and a register to store the received bits.

Using method 2 we obtain the following implementation:
First we translate the program to RTL:

1. size = input
2. i = size
3. enable for1
4. i = 8 - size
5. enable for2

Then we generate the hardware:

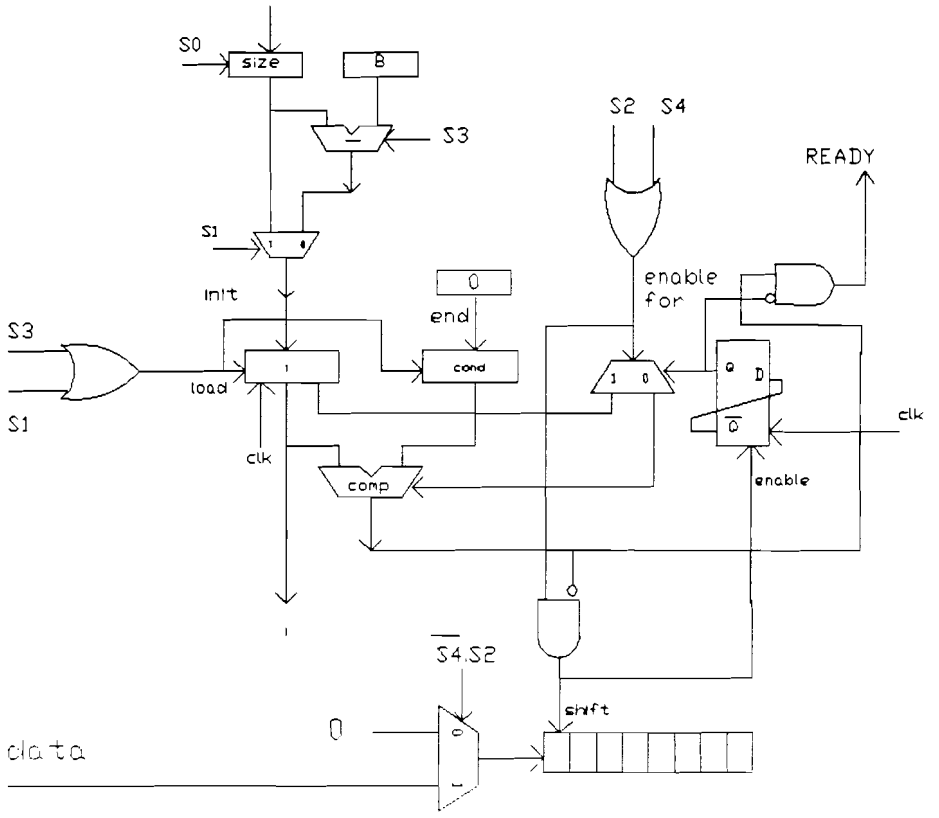


Fig. 9.2.1.2.(a) Implementation of Alg.1 using method 2 data unit

Q^v	Ready		Control
	1	0	
0	1	1	S_0
1	2	2	S_1
2	3	2	S_2
3	4	4	S_3
4	1	4 Q^{v+1}	S_4

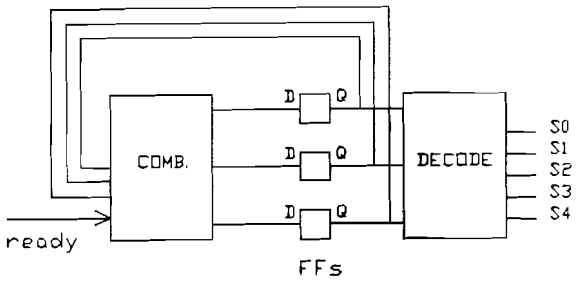


Fig. 9.2.1.2.(b) Implementation of Alg.1 using method 2
state table and controller

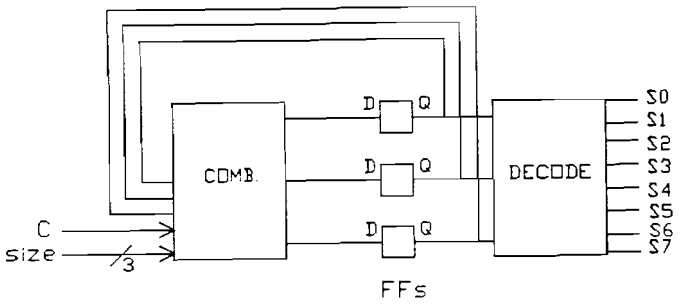
Of course this is not the only algorithm to describe the controller. The algorithm determines a great part of what the hardware will look like.

- Algorithm 2

```
int size;
size = input;


for (i=8; i>0;i--)
  {
   reg[0] = reg[1]; reg[1] = reg[2]; reg[2] = reg[3];
   reg[3] = reg[4];
  switch (size) {
  case 5 : reg[4] = data; reg[5] = reg[6]; reg[6] = reg[7];
   reg[7] = 0;
  case 6 : reg[4] = reg[5]; reg[5] = data; reg[6] = reg[7];
     reg[7] = 0;
  case 7 : reg[4] = reg[5]; reg[5] = reg[6]; reg[6] = data;
     reg[7] = 0;
  case 8 : reg[4] = reg[5]; reg[5] = reg[6]; reg[6] = reg[7];
   reg[7]=data;
  };
};
```

Using method 1 we obtain this RTL:

```
1. size = input; i= 8;
2. reg[0] = reg[1]; reg[1] = reg[2]; reg[2] = reg[3]; reg[3] = reg[4];
    switch(size) {
case 5 : reg[4] = data; reg[5] = reg[6]; reg[6] = reg[7]; reg[7] = 0;
case 6 : reg[4] = reg[5]; reg[5] = data; reg[6] = reg[7]; reg[7] = 0;
case 7 : reg[4] = reg[5]; reg[5] = reg[6]; reg[6] = data; reg[7] = 0;
case 8 : reg[4] = reg[5]; reg[5] = reg[6]; reg[6] = reg[7];reg[7]
=data;
  }; decr(i);
3. if c continue else -> 2;
```



Q^v	C		Size				Control
	0	1					
0	1	1	x	x	x	x	
1	2	2	x	x	x	x	S_0
2	3	3	0	1	0	1	S_{34} S_1
2	3	3	0	1	1	0	S_{33} S_1
2	3	3	0	1	1	1	S_{32} S_1
2	3	3	1	0	0	0	S_{31} S_1
3	2	0	x	x	x	x	S_7

Fig. 9.2.1.3.(a) Alg.2; control unit and state table.

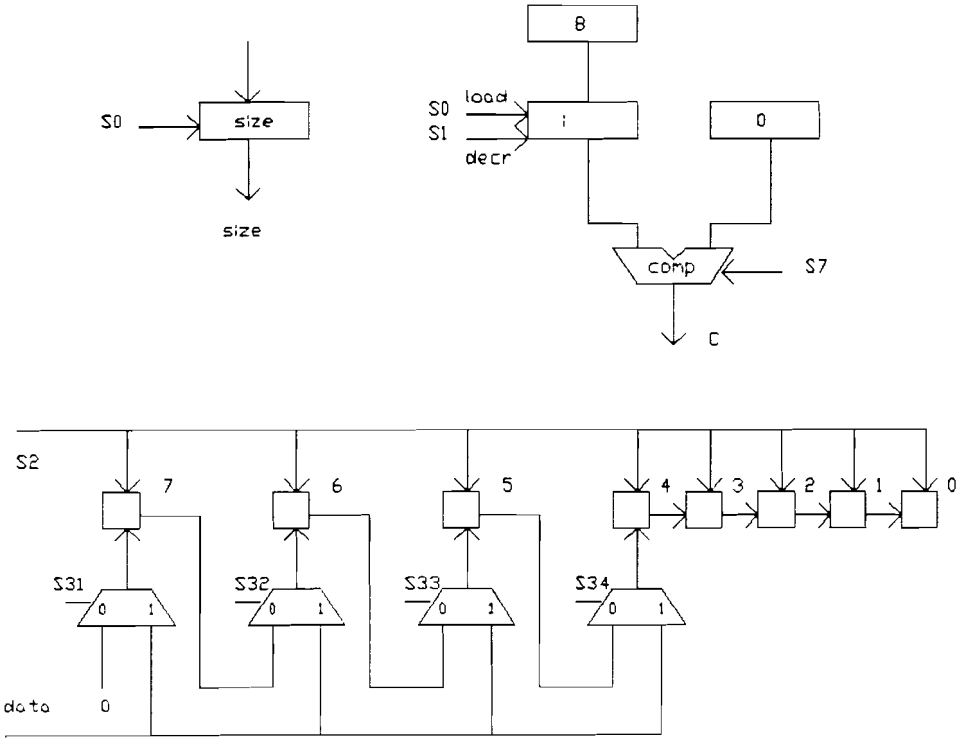


Fig. 9.2.1.3.(b) Implementation of Alg.2 using method 1; data unit.

9.2.2.Example 2. Video controller

The next example is a video controller. This controller must be able to load a video ram memory and then supply the bit patterns of the lines to be displayed on screen. We could describe the display process by the following algorithm:

Algorithm video

```
#define N 35          /* number of characters */
char video[80][25];   /* RAM */
int  disp[8][N]       /* ROM, contains bit patterns characters */

for(x=0; x<=24; x++)
 for(z=0; z<=7; z++)
  for(y=0; y<=80; y++)
   out = disp[video[x][y]][z];  /* points to bit in ROM */
```

Using method 1 the RTL can be compiled out of the algorithm.

RTL:

```
e: x==25        /* conditions */
d: z==8
c: y==81

1. x = 0;
2. z= 0;
3. y = 0;
4. out = disp[video[x][y]][z]; incr(y);
5. if c incr(z) else -> 4;
6. if d incr(x) else -> 3;
7. if e continue else -> 2;
```

Q^v	Q^{v+1} C 0	C 1	D 0	D 1	E 0	E 1	Control C 0	C 1	D 0	D 1	E 0	E 1
0	1	1	1	1	1	1						
1	2	2	2	2	2	2			S_0			
2	3	3	3	3	3	3			S_4			
3	4	4	4	4	4	4			S_2			
4	5	5	5	5	5	5			S_7	S_3		
5	4	6	-	-	-	-		S_5				
6	-	-	3	7	-	-				S_1		
7	-	-	-	-	2	0						

Fig.9.2.2.1.a State table Alg. video method 1.

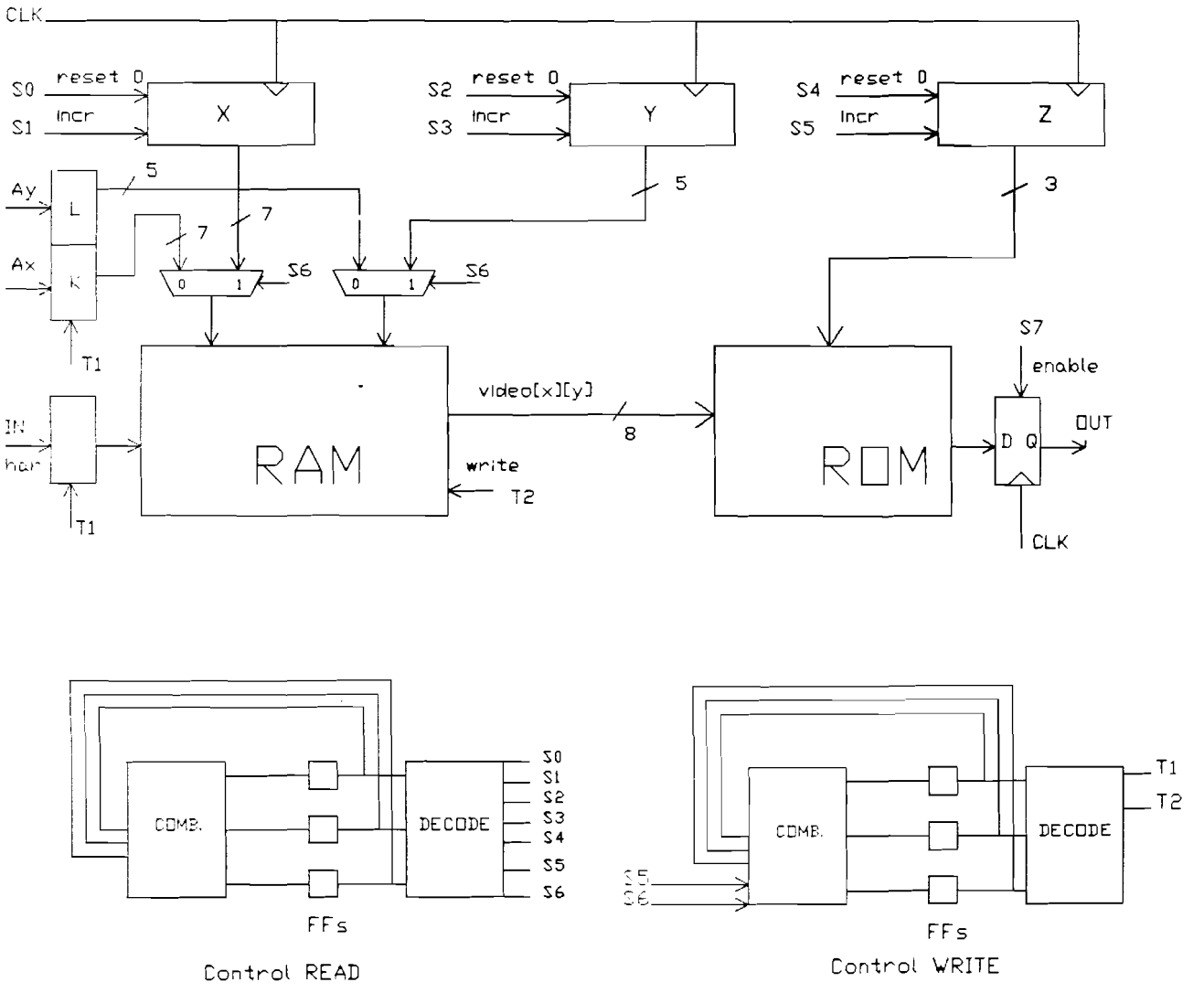


Fig.9.2.2.1.b Implementation of Alg. video using method 1.
Here incremental registers are used

Using method 2 we obtain the following RTL after compilation:

RTL:
Controller 1
1 enable for2
2. if e continue else -> 1

For2
1 enable for3
2. if d continue else -> 1

For3
1. out = disp[video[x][y]][z]; incr(y);
2. if c continue else -> 1

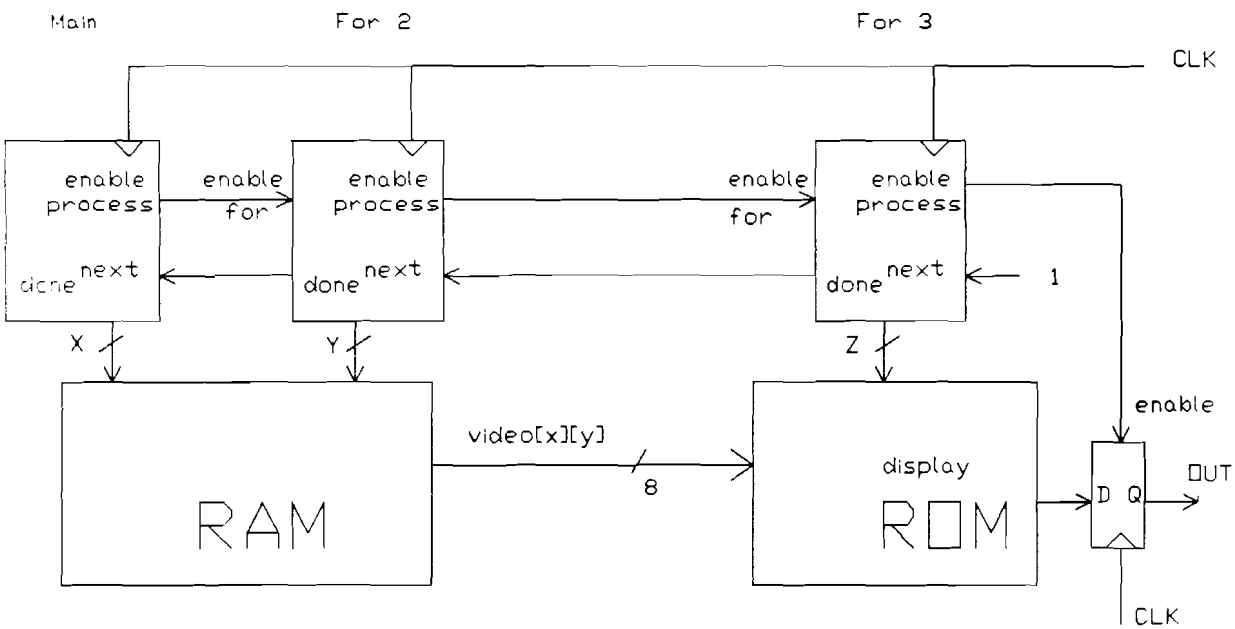


Fig.9.2.2.2. Implementation of Alg. video using method 2.

The two examples show that when using method 2, a need for synchronization arises.

9.3.Parallel processing

Until now we have seen systems to be implemented by one data path and one control unit. But we have seen in chapter 3 that it is possible to have more than one process. How to implement systems with processes running in parallel ?
Figure 9.3.1 shows methods. Each process is implemented as a subsystem consisting of a data path and a control unit. Communication and synchronization has to be done by the control units. One possibility is to have one main controller controlling the others; an other way is to have a control bus with a protocol for communication among the controllers.
The data paths are also connected by a bus. In this way data can be transported from one subsystem to another. "Expensive" units like multipliers and dividers can be shared by processes (if it is possible to arrange that two or more processes do not need such a unit at the same time).

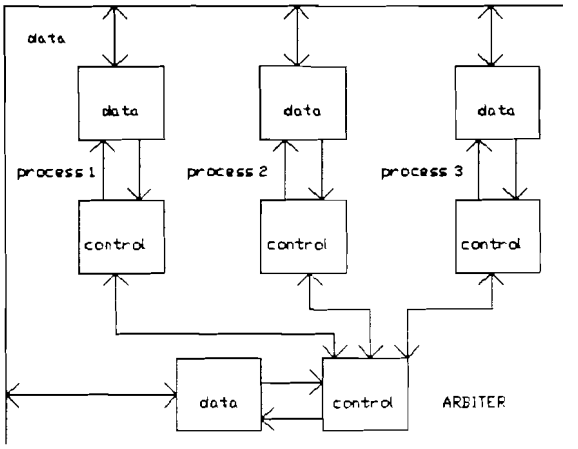


Fig. 9.3.1. (a) Implementation of parallel processes with arbiter

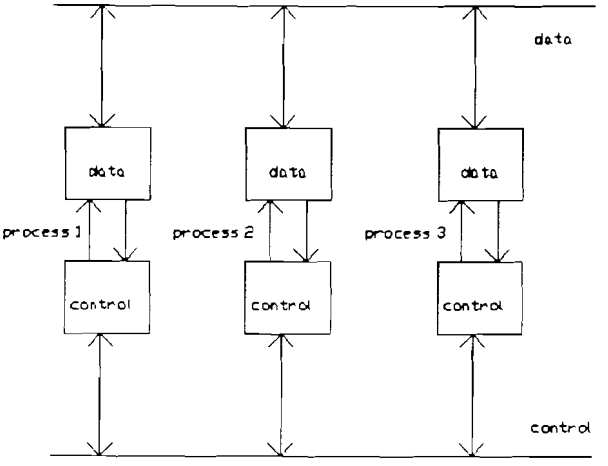


Fig. 9.3.1. (b) Implementation of parallel processes with control bus

It is beyond the goal of this report to work this out further.

9.4.Communication and synchronization

Consider an algorithm with multiple procedures or multiple procedure calls.
Since every procedure will be implemented as a finite state machine some kind of communication has to take place since the whole system consists of linked state machines.
Suppose we use vectors to start and stop processes. The processes can be protected by semaphores.
We now can distinguish two sorts of implementations:

1. alternately performing state machines
2. parallel performing state machines

9.4.1.Alternately performing state machines
The arbiter implements the main program.
State machine B implements a special procedure.
When running the main program the special procedure has to be started. State machine A will send a start vector to state machine B. State machine B needs the operands and will open the input ports. After having executed the procedure state machine B will send a "done vector". Now state machine A will open the input ports for reading the result.
Using this method means that state machine A will not run when state machine B is running and vice versa.
Advantage is that synchronization is easy to implement. Disadvantage is the waste of time.

9.4.2.Parallel performing state machines

See Figure 9.4.2. State machine A represents the main program.
State machine B implements a special procedure.
When running the main program the special procedure has to be started. State machine A will send a start vector to state machine B. State machine B needs the operands and will open the input ports. After having executed the procedure state machine B will send a "done vector". Now state machine A will open the input ports for reading the result.

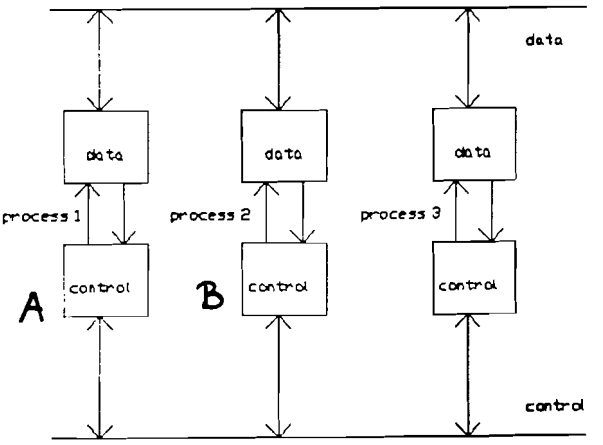


Fig.9.4.2. parallel performing state machines

Using this method means that state machine A will run when state machine B is running and vice versa. Only at specific times when A needs the results of B or vice versa the state machines will wait until the "done vector" from the other has been received. So sometimes the performance will be the same as by using method 1. On the other hand state machine A now can start multiple procedures. So a same problem implemented with this method could be solved much faster.
Disadvantage of this method is that synchronization is difficult to implement and the danger for deadlock. Advantage is the fast operation because processes can run in parallel.

9.4.3.Communication with the environment

Since a digital system has its own system clock, special technics must be used to communicate with other systems. There are many methods to synchronize two systems, but we only give one at this place. It is the handshaking protocol. Figure 9.4.3 shows the signals and timing. More about this protocol can be found in the literature.

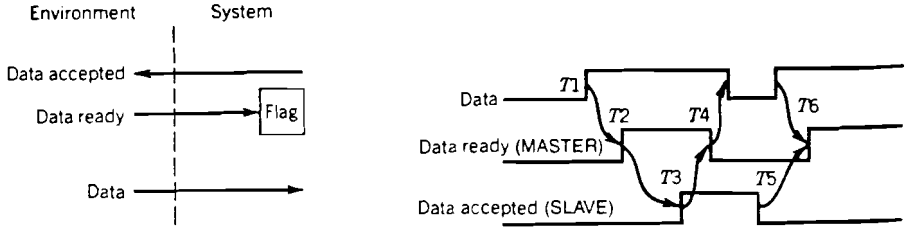


Fig. 9.4.3 Handshaking protocol

See also the master thesis report of E.J. van Dort, Digital Systems Group, TUE. In this report more about parallel processing and timing can be found.

10.CONCLUSIONS

The top-down approach for designing digital systems as used by the Digital Systems Group of the EUT has been the basis for investigating possibilities for automatic implementation of algorithms into hardware. This approach relies on seeing a digital system as a control part and a processing part.

Some modules have been proposed to implement statements of algorithms written in a register transfer language. Also modules for implementing higher level statements like 'if..then..else' and 'for-loops' have been introduced.
An algorithm for clique-partitioning has been written to help to minimize registers, operators, and interconnection in the processing unit. This algorithm has shown nice results in examples. Only the covering problem is a point of further study.

It has been shown that an algorithm written in a RTL, can directly be implemented in hardware.

Timing and clocking is still a problem.

So the main object for further research will be a compiler, to transform a high-level description into an RT description.

LITERATURE

[1] Davio, M.
Digital systems with algorithm implementation.
Wiley-Interscience, 1983. - XVII, 505 p.
Electro-bibl. : LLP 83 DAV

[2] Ercegovac, Milos Draguin
Digital systems and hardware/firmware algorithms.
Chichester : Wiley, 1985. - XIX, 838 p. - ISBN 0-471-8-8393-X
Electro-bibl. : LLP 85 ERC

[3] Even, S.
Graph algorithms.
Computer Science Press, 1979. - IX, 249 p.
(Computer software engineering series)
Electro-bibl. : CHK 79 EVE

[4] Christofides, N.
Graph theory : an algorithmic approach.
Academic Press, 1975 (Computer science and applied mathematics)
Rekencentrum bibl. : CHK 75 CHR

[5] Balakrishnan, M. et al.
Allocation of multiport memories in data path synthesis
IEEE Trans. CAD, Vol. CAD-7(1988), No. 4, p536-540

[6] Hill, F.J. and Peterson, G.R.
Digital systems: hardware organization and design
John Wiley & Sons, New York, 1973.

[7] Hafer, L.J. and Parker, A.C.
Automated synthesis of digital hardware.
IEEE Trans. Computers, Vol. C-31(1982), No. 2, p.93-109.

[8] Chia-Jeng Tseng and Siewiorek, D.P.
Automated synthesis of data paths in digital systems.
IEEE Trans. CAD, Vol. CAD-5(1986), No. 3, p.379-395.

[9] Nagle, A.W. et al.
Synthesis of hardware for the control of digital systems.
IEEE Trans. CAD, Vol. CAD-1(1982), No. 4, p.201-212.

[10] Tricky, H.
Flamel: a high-level hardware compiler
IEEE Trans. CAD, Vol. CAD-6(1987), No. 2, p.259-269.

[11] Bron, C. and Kerbosch, J.
Algorithm 457: Finding all cliques of an undirected graph.
Comm. ACM, Vol. 16(1973), No. 9, p.575-577.

[12] Hafer, L.J. and Parker, A.C.
A formal method for the specification, analysis, and design of register transfer level digital logic.
IEEE Trans. CAD, Vol. CAD-2(1983), No. 1, p.4-17.

[13] Thomas, D.E. and Nestor, J.A.
Defining and implementing a multilevel design representation with simulation applications.
IEEE Trans. CAD, Vol. CAD-2(1983), No. 3, p.135-144.

[14] Shin, K.G. and Ramanathan, P.
Clock synchronization of a large multiprocessor system in the presence of malicious faults.
IEEE Trans. Computers, Vol. C-36(1987), No. 1, p.2-12.

[15] Kanakia, H.R. and Tobagi, F.A.
On distributed computations with limited resources.
IEEE Trans. computers, Vol. C-36(1987), No. 5, p.517-528

[16] Jahanian, F. and Mok, A.K.-L.
A graph-theoretic approach for timing analysis and its implementation.
IEEE Trans. computers, Vol. C-36(1987), No.8, p.961-975.

[17] Granski, M. et al.
The effect of operation scheduling on the performance of a data flow computer.
IEEE Trans. computers, Vol. C-36(1987), No. 9, p.1019-1029.

[18] Zurawski, J.H.P. and Gosling, J.B.
Design of a high speed square root multiply and divide unit.
IEEE Trans. computers, Vol. C-36(1987), No. 1, p.13-23.

[19] Hill, F.J. et al.
Hardware compilation from an RTL to a storage logic array target.
IEEE Trans. CAD, Vol. CAD-3(1984), No. 3, p.208-217.

[20] Hirayama, M.
A silicon compiler system based on asynchronous architecture.
IEEE Trans. CAD, Vol. CAD-6(1987), No. 3, p.297-304.

[21] Nakamura, Y.
An integrated logic design environment based on behavioral description.
IEEE Trans. CAD, Vol. CAD-6(1987), No. 3, p.322-336.

[22] Hartenstein, R.W.
Fundamentals of structured hardware design; a design language approach at register transfer level.
North-Holland, Amsterdam, 1977.

[23] Dort, E.J. van,
master thesis report:Description and implementation of digital systems.

APPENDIX A

Clique partitioning algorithm

A maximal complete subgraph (clique) is a complete subgraph that is not contained in any other complete subgraph.
In this backtracking algorithm a branch-and-bound technique has been used to cut off branches that cannot lead to a clique. It has been based on the procedure "extend" of algorithm 457 [11].

Version 1.0

The input is expected to be a file which must be called 'cliq.in'.
The format is:

```
[number of variables] [number of time slots]
[0/1] [0/1] etc  /* the number of columns must be equal to the number
*/
[0/1] [0/1] etc   /* of variables. column 1 for variable 1
[0/1] [0/1] etc  /* the number of rows must be equal to the number of
*/
[0/1] [0/1] etc  /* of time slots, row 1 = time 1. */
-1
```

a '1' in the matrix is used to indicate that the variable is 'live' at
this point of time.
a '0' in the matrix is used to indicate that the variable is 'dead' at
this point of time.

Example:

```
4 5                          /* 4 variables; 5 time slots */
1 0 0 1                      /* var 1 and var 4 are live  */
1 1 0 1                      /* vars 1,2, and 4 are live  */
1 1 1 0                      /* vars 1,2, and 3 are live  */
1 1 1 0                      /*                                    */
0 1 1 0                      /* vars 2 and 3 are live              */

-1                           /* end of file */
```

```
#define N 20
#define CE 20
#include <stdio.h>
FILE *cliqin, *cliq;
int c;
int  connect[N+1][N+1];
int compsub[N+1],all[N+1];
int tab[N+1][N+1];
int t;

extend(old,ne,ce)                    /* extracts cliques from graph */
     int old[];
     int ne,ce;
     {
       int new[CE+1];        /* local variables. no need to be */
       int nod,fixp;         /* declared recursively           */
       int newne,newce,i,j,count,pos,p,s,sel,minnod;
       int loc;
       minnod=ce;
       nod = 0;
     for (i=1;(i<= ce)&& (minnod !=0);i++)
       {                           /* determine each counter value and look */
         p=old[i]; count=0;  /* for minimum                                    */
         for (j=ne+1;(j <=ce)&& (count < minnod) ;j++)
```

```
        if (connect[p][old[j]]==0) /* count disconnections           */
          {count++; pos=j;};       /* save position of potential candidate
*/
       if (count < minnod)         /* test new minimum */
          {fixp=p; minnod=count;
             if (i<=ne) s=pos;
             else {s=i; nod=1;}
          };               /* if fixed point initially chosen from */
      };                          /* candidates then number of disconnections*/
                                   /* will be preincreased by one            */
      for (nod=minnod + nod;nod>=1;nod--) /* backtrack cycle */
      {
       p=old[s]; old[s]=old[ne+1];   /* interchange   */
       sel=old[ne+1]=p;
       newne=0;                                /* fill new set 'not' */
       for (i=1;i<=ne;i++)
        if (connect[sel][old[i]]==1)
         {++newne; new[newne]=old[i];};
       newce=newne;                           /* fill new set 'candidates' */
       for (i=ne+2; i<=ce; i++)
          if (connect[sel][old[i]]==1)
           { newce++;new[newce]=old[i];};
       c++; compsub[c]=sel;     /* add to compsub */
       if (newce==0)
        {
          fprintf(cliq,"\n clique=");
          for (loc=1;loc<=c;loc++)
           fprintf(cliq,"\t%d", compsub[loc]);
        }
        else
            if (newne < newce)
             extend(new,newne,newce);
       c--;                                   /* remove from compsub    */
       ne++;                    /* add to 'not' */
       if (nod>1)
        {s=ne;   /* select a candidate disconnected to the fixed point
*/
           s++; /* look for candidate */
            while (connect[fixp][old[s]]==1)    s++;
       }          /* end selection     */
     }       /* end backtrack cycle    */
   }         /* end extend               */



subgraph(m)
 int m;
 {
  for (c=1;c<=m;c++)
      all[c]=c;
  c=0;
  extend(all,0,m);
 }
```

```
output(m)                          /* writes array connect   */
int m;
{
int l,n;
fprintf(cliq,"\n\n CONNECT \n\n");
for (l=1;l<=m;l++)
     {fprintf(cliq,"\n");
      for(n=1;n<=m;n++)
        fprintf(cliq,"%2d",connect[l][n]);
     }
}

intrace(m,t)                       /* reads the value trace table */
int m;
{
 int k,l;
 for(k=0;k<=t;k++)
    {
     for (l=1;l<=m;l++)
       fscanf(cliqin,"%d",&tab[k][l]);
    };
fclose(cliqin);
fprintf(cliq,"\n TRACETABLE \n\n");   /* print tracetable */
for (k=0;k<=t;k++)
 {
  for (l=1;l<=m;l++)
    fprintf(cliq,"%2d",tab[k][l]);
  fprintf(cliq,"\n");
 };
}

comptrace(m,t)          /* checks which variables can be combined */
int m,t;
{
 int comp;
 int i,j,k;
 for (i=1; i<m ;i++)
  {
   connect[i][i] = 1;
   for (j=i+1;j<=m;j++)
      { comp=1;
        k=0;
        while ((k<=t) && (comp==1))
            {
             if (tab[k][i] && tab[k][j])
               if (tab[k+1][i] && tab[k+1][j])
                  comp=0;
             k++;
            };
        connect[i][j]=connect[j][i]=comp;
      };
  };
  connect[m][m]=1;
 }
```

```
main()
{
int m,t;
cliqin = fopen("cliq.in","r");
fscanf(cliqin, "%d%d",&m,&t);      /* number of variables, timeslots */
cliq = fopen("cliq.lst","w");
t+=1;
intrace(m,t);
comptrace(m,t);
output(m);
subgraph();
fclose(cliq);
}
```

Version 2.0

This version tries to minimize the number of cliques produced.
The input is expected to be a file which must be called 'cliq.in'.
The format is:

```
[number of variables] [number of time slots]
[0/1] [0/1] etc   /* the number of columns must be equal to the number
*/
[0/1] [0/1] etc    /* of variables. column 1 for variable 1
[0/1] [0/1] etc   /* the number of rows must be equal to the number of
*/
[0/1] [0/1] etc   /* of time slots, row 1 = time 1. */
[special variable] [special variable] etc.
-1
```

a '1' in the matrix is used to indicate that the variable is 'live' at
this point of time.
a '0' in the matrix is used to indicate that the variable is 'dead' at
this point of time.

a special variable is the number of a variable which must be combined
to another special variable for optimum's sake. Usually these numbers
are obtained from the G1 graph.

Example:

```
4 5                    /* 4 variables; 5 time slots */
1 0 0 1                /* var 1 and var 4 are live  */
1 1 0 1                /* vars 1,2, and 4 are live  */
1 1 1 0                /* vars 1,2, and 3 are live  */
1 1 1 0                /*                           */
0 1 1 0                /* vars 2 and 3 are live     */
2 3                    /* if possible var 2 and var 3 must be combined */
-1                     /* end of file */
```

```
#define N 20
#define CE 20
#include <stdio.h>
FILE *cliqin, *cliq;

int c;
int  connect[N+1][N+1];
int  conmain[N+1][N+1];
int compsub[N+1],all[N+1];
int tab[N+1][N+1];
int t;
int spec2[N+1];

extend(old,ne,ce)              /* extracts cliques from graph */
    int old[];
    int ne,ce;
    {
     int new[CE+1];             /* local variables. no need to be */
     int nod,fixp;              /* declared recursively           */
     int newne,newce,i,j,count,pos,p,s,sel,minnod;
     int loc;
     minnod=ce;
     nod = 0;
    for (i=1;(i<= ce)&& (minnod !=0);i++)
     {                          /* determine each counter value and look */
      p=old[i]; count=0;   /* for minimum                                  */
      for (j=ne+1;(j <=ce)&& (count < minnod) ;j++)
       if (connect[p][old[j]]==0) /* count disconnections                 */
        {count++; pos=j;};        /* save position of potential candidate
*/
      if (count < minnod)         /* test new minimum */
        {fixp=p; minnod=count;
           if (i<=ne) s=pos;
           else {s=i; nod=1;}
        };                 /* if fixed point initially chosen from */
     };                    /* candidates then number of disconnections*/
                           /* will be preincreased by one          */
     for (nod=minnod + nod;nod>=1;nod--) /* backtrack cycle */
     {
      p=old[s]; old[s]=old[ne+1];   /* interchange  */
      sel=old[ne+1]=p;
      newne=0;                          /* fill new set 'not' */
      for (i=1;i<=ne;i++)
       if (connect[sel][old[i]]==1)
        {++newne; new[newne]=old[i];};
      newce=newne;                     /* fill new set 'candidates' */
      for (i=ne+2; i<=ce; i++)
     if (connect[sel][old[i]]==1)
      { newce++;new[newce]=old[i];};
      c++; compsub[c]=sel;            /* add to compsub    */
```

```
        if (newce==0)
         {
        fprintf(cliq,"\n clique=");
        for (loc=1;loc<=c;loc++)
         {if (c !=1)                          /* NEW in this version  */
             spec2[compsub[loc]] = 1;          /* record used cliques */
          fprintf(cliq,"\t%d", compsub[loc]);
         };
         }
        else
       if (newne < newce)
        extend(new,newne,newce);
        c--;                                   /* remove from compsub    */
        ne++;                    /* add to 'not' */
        if (nod>1)
         {s=ne;  /* select a candidate disconnected to the fixed point
*/
            s++; /* look for candidate */
             while (connect[fixp][old[s]]==1)    s++;
        }          /* end selection     */
     }       /* end backtrack cycle    */
   }         /* end extend               */

subgraph(m)
 int m;
 {
  for (c=1;c<=m;c++)    /* fill all */
      all[c]=c;
  c=0;                          /* call extend */
  extend(all,0,m);
 }

output(m)                       /* writes array connect  */
int m;
{
int l,n;
fprintf(cliq,"\n\n CONNECT \n\n");
for (l=1;l<=m;l++)
   {fprintf(cliq,"\n");
    for(n=1;n<=m;n++)
        fprintf(cliq,"%2d",connect[l][n]);
   };
}
```

```
intrace(m,t)                    /* reads the value trace table */
int m;
{
 int k,l;
 for(k=0;k<=t;k++)
    { for (l=1;l<=m;l++)
       fscanf(cliqin,"%d",&tab[k][l]);
    };
 fprintf(cliq,"\n TRACETABLE \n\n");   /* print tracetable */
 for (k=0;k<=t;k++)
  { for (l=1;l<=m;l++)
     fprintf(cliq,"%2d",tab[k][l]);
    fprintf(cliq,"\n");
  };
}
comptrace(m,t)        /* checks which variables can be combined */
int m,t;
{ nt comp;
 int i,j,k;
 for (i=1; i<m ;i++)
  {
   conmain[i][i] = 1;
   for (j=i+1;j<=m;j++)
      { comp=1;
        k=0;
        while ((k<=t) && (comp==1))
       {
          switch(tab[k][i])                  /* conditions for compatibility
*/
            {
              case 0   : comp = 1; break;
              case 1   : switch(tab[k][j])
                         {
                          case 0 : comp = 1; break;
                          case 1 : comp = 0; break;
                          case 2 : if ((k+1<=t) && (tab[k+1][i]==0))
                                     comp = 1;
                                       else {comp = 0;printf("YES2");};
                                       break;
                         }; break;
              case 2   : switch(tab[k][j])
                         { case 0 : comp = 1; break;
                          case 1 : if ((k+1<=t) && (tab[k+1][j]==0))
                                     comp = 1;
                                   else comp = 0; break;
                          case 2 : comp = 0;break;
                         }; break;
            };  k++;
        };                                   /* end while */
       conmain[i][j]=conmain[j][i]=comp;
       };                                    /* end for j  */
  };                                         /* end for i */
   conmain[m][m]=1;
}
```

```
main()
{
int m,t;
int k,l;
int g,spec[N+1];

  cliqin = fopen("cliq.in","r");
  fscanf(cliqin, "%d%d",&m,&t);      /* number of variables, timeslots */

  cliq = fopen("cliq.lst","w");
  t+=1;

intrace(m,t);
comptrace(m,t);
  for(k=1;k<=m;k++)
    for(l=1;l<=m;l++)
      connect[k][l]=conmain[k][l];
output(m);
subgraph(m);                    /* calculate all cliques */

  fscanf(cliqin,"%d",&g);
  while (g != -1)
  {
    printf("\t %d",g);
    spec[g]=1;
    fscanf(cliqin,"%d",&g);
  };
    for(k=1;k<=m;k++)
      {
       for (l=1;l<=m;l++)           /* remove all edges not connected */
        if (spec[l] != 1)           /* to the special nodes     */
          connect[k][l] = connect[l][k] = 0;
       connect[k][k] = 1;
      };
output(m);
for(k=1;k<=m;k++)
  spec2[k] = 0;                     /* reset spec2          */
subgraph(m);                       /* calculate reduced cliques */

for (k=1;k<=m;k++)
  printf(" \t %d%",spec2[k]);
for (k=1;k<=m;k++)
  for (l=1;l<=m;l++)
     connect[k][l]=conmain[k][l];
for (k=1;k<=m;k++)
  {for (l=1;l<=m;l++)
     if (spec2[l] == 1)
       connect[k][l] = connect[l][k] = 0;
  };
output(m);
subgraph(m);                    /* cliques after reduction */
fclose(cliq);
fclose(cliqin);
}
```

■ Example ABC formula sequence 1

Step1.Numbering the variables:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
A	B	C	D	E	F	G	H	I	K	L	M	N	O	P

TRACETABLE

```
1 1 1 0 0 0 0 0 0 0 0 0 0 0 0
1 1 1 0 1 0 0 0 0 0 0 0 0 0 0
1 1 1 0 1 1 0 0 0 0 0 0 0 0 0
1 1 0 0 0 1 1 0 0 0 0 0 0 0 0
1 1 0 0 0 1 1 1 0 0 0 0 0 0 0
1 1 0 1 0 1 0 1 0 0 0 0 0 0 0
1 1 0 1 0 0 0 0 1 0 0 0 0 0 0
1 1 0 0 0 0 0 0 1 1 0 0 0 0 0
0 1 0 0 0 0 0 0 0 1 1 1 0 0 0
0 1 0 0 0 0 0 0 0 1 1 1 1 0 0
0 1 0 0 0 0 0 0 0 0 1 1 1 1 0 0
0 0 0 0 0 0 0 0 0 0 1 1 0 1 1 0
0 0 0 0 0 0 0 0 0 0 1 0 0 1 1 1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1
```

CONNECT

```
1 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1
0 1 0 0 0 0 0 0 0 0 0 0 0 1 1 1
0 0 1 1 0 1 1 1 1 1 1 1 1 1 1 1
0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1
0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1
0 0 1 1 1 1 0 0 1 1 1 1 1 1 1 1
0 0 1 1 1 0 1 1 1 1 1 1 1 1 1 1
0 0 1 1 1 0 1 1 1 1 1 1 1 1 1 1
0 0 1 1 1 1 1 1 1 1 0 0 1 1 1 1
1 0 1 1 1 1 1 1 1 0 1 0 0 0 0 1
1 0 1 1 1 1 1 1 1 0 0 1 0 0 1 1
1 0 1 1 1 1 1 1 1 1 0 0 1 1 1 1
1 1 1 1 1 1 1 1 1 1 0 0 1 1 0 1
1 1 1 1 1 1 1 1 1 1 0 1 1 0 1 0
1 1 1 1 1 1 1 1 1 1 1 1 1 0 1
```

clique=	15	4	3	7	8	9	12	13
clique=	15	4	3	7	8	10		
clique=	15	4	3	7	8	11		
clique=	15	4	3	6	9	12	13	
clique=	15	4	3	6	10			
clique=	15	4	3	6	11			
clique=	15	4	5	7	8	9	12	13
clique=	15	4	5	7	8	10		
clique=	15	4	5	7	8	11		
clique=	15	4	5	6	9	12	13	
clique=	15	4	5	6	10			
clique=	15	4	5	6	11			
clique=	15	2	13					
clique=	15	1	12	13				
clique=	15	1	11					
clique=	15	1	10					
clique=	14	4	3	7	8	9	12	
clique=	14	4	3	7	8	11		
clique=	14	4	3	6	9	12		
clique=	14	4	3	6	11			
clique=	14	4	5	7	8	9	12	
clique=	14	4	5	7	8	11		
clique=	14	4	5	6	9	12		
clique=	14	4	5	6	11			
clique=	14	2						
clique=	14	1	11					
clique=	14	1	12					

9	10	11	12	13	14	15
I	K	L	M	N	O	P
	x				x	x
			x	x		
x	x		x			

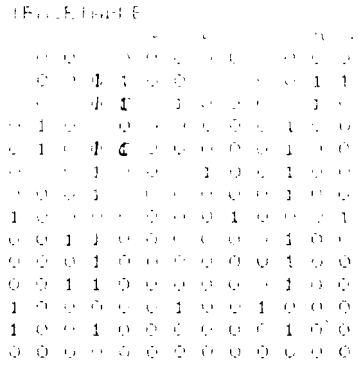
ors
y one operator

connection

tination Label

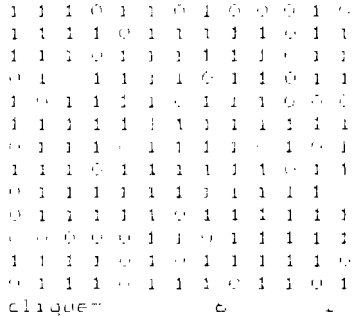
	1
ALU.IN2	2
ALU.IN1	3
ALU.IN1	6
B	5
ALU.IN2	7
ALU.IN1	8
ALU.IN2	9
	10
JT A	11
UT B	12
UT C	13
OUT E	
OUT ALU.IN1	

abc can 1



interconnection

CONNECT



clique=
clique=
clique=
clique=
clique=
clique=
clique=
clique=
clique=
clique=
clique=
clique=
clique=
clique=
clique=

Example ABC formula sequence 2

Step1.Numbering the variables:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
A	B	C	D	E	F	G	H	I	K	L	M	N	O	P

TRACETABLE

```
1 1 1 0 0 0 0 0 0 0 0 0 0 0 0
1 1 1 0 1 0 1 0 0 0 0 0 0 0 0
1 1 1 0 1 1 1 1 0 0 0 0 0 0 0
1 1 0 1 0 1 0 1 0 0 0 0 0 0 0
1 1 0 1 0 0 0 0 1 1 0 0 0 0 0
0 1 0 0 0 0 0 0 1 1 1 1 0 0 0
0 1 0 0 0 0 0 0 0 1 1 1 1 1 0
0 0 0 0 0 0 0 0 0 1 0 0 1 1 1
0 0 0 0 0 0 0 0 0 0 0 0 0 1 1
```

CONNECT

```
1 0 0 0 0 0 0 0 1 1 1 1 1 1 1
0 1 0 0 0 0 0 0 0 0 0 0 1 1 1
0 0 1 1 0 1 0 1 1 1 1 1 1 1 1
0 0 1 1 1 1 1 1 1 1 1 1 1 1 1
0 0 0 1 1 1 0 1 1 1 1 1 1 1 1
0 0 1 1 1 1 1 0 1 1 1 1 1 1 1
0 0 0 1 0 1 1 1 1 1 1 1 1 1 1
0 0 1 1 1 0 1 1 1 1 1 1 1 1 1
1 0 1 1 1 1 1 1 1 0 1 1 1 1 1
1 0 1 1 1 1 1 1 1 0 1 0 0 0 1
1 0 1 1 1 1 1 1 1 0 1 0 1 1 1
1 0 1 1 1 1 1 1 1 0 0 1 1 1 1
1 1 1 1 1 1 1 1 1 0 1 1 1 0 1
1 1 1 1 1 1 1 1 1 0 1 1 0 1 0
1 1 1 1 1 1 1 1 1 1 1 1 1 0 1
```

```
clique=    15    13    9    4    6    11    7
clique=    15    13    9    4    6    11    3
clique=    15    13    9    4    6    11    5
clique=    15    13    9    4    6    12    3
clique=    15    13    9    4    6    12    5
clique=    15    13    9    4    6    12    7
clique=    15    13    9    4    8    11    5
clique=    15    13    9    4    8    11    3
clique=    15    13    9    4    8    11    7
clique=    15    13    9    4    8    12    3
clique=    15    13    9    4    8    12    7
clique=    15    13    9    4    8    12    5
clique=    15    13    9    1    11
clique=    15    13    9    1    12
clique=    15    13    2
clique=    15    10    4    6    5
clique=    15    10    4    6    7
clique=    15    10    4    6    3
clique=    15    10    4    8    7
clique=    15    10    4    8    5
clique=    15    10    4    8    3
clique=    15    10    1
clique=    14    9    4    6    11    7
clique=    14    9    4    6    11    3
clique=    14    9    4    6    11    5
clique=    14    9    4    6    12    3
clique=    14    9    4    6    12    5
clique=    14    9    4    6    12    7
clique=    14    9    4    8    11    5
clique=    14    9    4    8    11    3
clique=    14    9    4    8    11    7
clique=    14    9    4    8    12    3
clique=    14    9    4    8    12    7
clique=    14    9    4    8    12    5
clique=    14    9    1    11
clique=    14    9    1    12
clique=    14    2
```

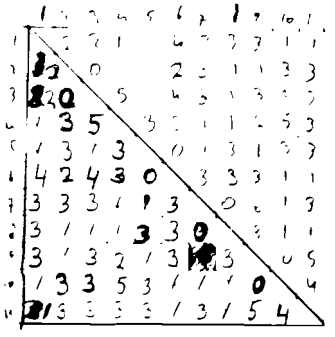
Combine:
2,13,15
1,9,12,14
3,4,8,10
6,7,11
5

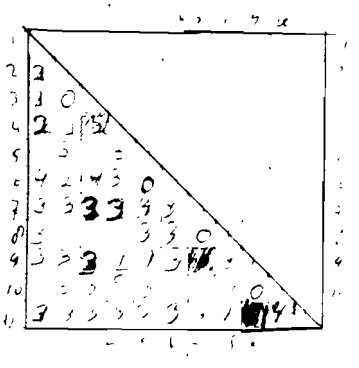
Reduce:

	1 A	2 B	3 C	4 D	5 E	6 F	7 G	8 H	9 I	10 K	11 L	12 M	13 N	14 O	15 P
A	x								x			x		x	
B		x											x		x
C			x	x				x		x					
E					x										
F						x	x				x				

Step2.Operators

*1	*2	*3	-1	SQRT	*4	-2	-1	-3	/1	/2
1	2	3	4	5	6	7	8	9	10	11





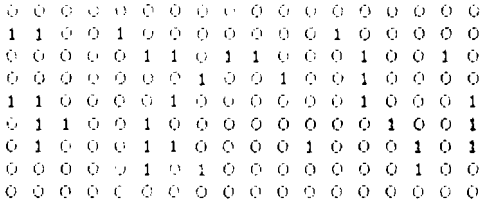
ALU1 = (7,9,3,4,11,1,6)
ALU2 = (5,8,10,2)

Step3.Interconnection

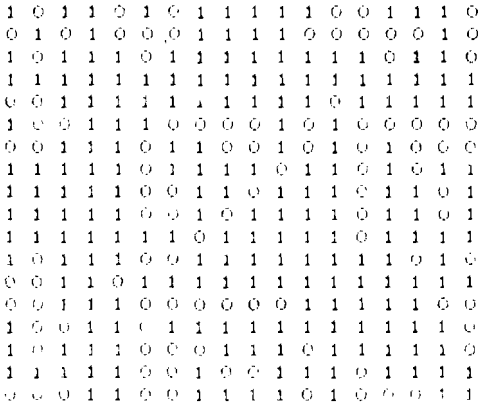
Source Destination Label

2	ALU1.IN2	4
4	ALU1.IN2	1
A	ALU1.IN1	2
A	ALU2.IN1	3
B	F	5
B	ALU1.IN2	6
C	ALU2.IN1	7
C	ALU1.IN1	8
E	ALU2.IN2	9
F	ALU1.IN1	10
F	ALU1.IN2	11
F	ALU2.IN1	12
ALU1.OUT	E	13
ALU1.OUT	C	14
ALU1.OUT	F	15
ALU1.OUT	B	16
ALU2.OUT	F	17
ALU2.OUT	A	18

TRACETABLE



CONNECT



clique= 4 11 5 15 12 10
clique= 4 11 5 15 12 9
clique= 4 11 5 15 12 17
clique= 4 11 5 15 16 9
clique= 4 11 5 15 16 10
clique= 4 11 5 15 16 17
clique= 4 11 5 16 10
clique= 4 11 5 18 17
clique= 4 11 5 18 9
clique= 4 11 1 3 12 10
clique= 4 11 1 3 12 9
clique= 4 11 1 3 12 17
clique= 4 11 1 3 16 9
clique= 4 11 1 3 16 10
clique= 4 11 1 3 16 17
clique= 4 11 1 6 ①
clique= 4 11 1 15 12 10
clique= 4 11 1 15 12 9
clique= 4 11 1 15 12 17
clique= 4 11 1 15 16 9
clique= 4 11 1 15 16 10
clique= 4 11 1 15 16 17
clique= 4 11 2 9
clique= 4 11 2 10
clique= 4 11 2 17
clique= 4 11 13 3 12 10
clique= 4 11 13 3 12 9
clique= 4 11 13 3 12 17
clique= 4 11 13 3 7
clique= 4 11 13 3 16 9
clique= 4 11 13 3 16 10
clique= 4 11 13 3 16 17
clique= 4 11 13 6
clique= 4 11 13 15 12 9
clique= 4 11 13 15 12 10
clique= 4 11 13 15 12 17
clique= 4 11 13 15 7
clique= 4 11 13 15 16 10
clique= 4 11 13 15 16 9
clique= 4 11 13 15 16 17
clique= 4 11 13 18 9
clique= 4 11 13 18 10
clique= 4 11 13 18 17
clique= 4 8 5 3 12 9
clique= 4 8 5 3 12 10
clique= 4 8 5 3 12 17
clique= 4 8 5 3 7 ②
clique= 4 8 5 15 12 10
clique= 4 8 5 15 12 9
clique= 4 8 5 15 12 17
clique= 4 8 5 15 7
clique= 4 8 5 18 10
clique= 4 8 5 18 17 ③
clique= 4 8 5 18 9
clique= 4 8 1 12 3 10
clique= 4 8 1 12 3 9
clique= 4 8 1 12 3 17
clique= 4 8 1 12 15 9
clique= 4 8 1 12 15 10
clique= 4 8 1 12 15 17
clique= 4 8 2 9
clique= 4 8 2 10 ④
clique= 4 8 2 17
clique= 4 8 13 12 3 9
clique= 4 8 13 12 3 10
clique= 4 8 13 12 3 17
clique= 4 8 13 12 15 10
clique= 4 8 13 12 15 9
clique= 4 8 13 12 15 17
clique= 4 8 13 7 3
clique= 4 8 13 7 15
clique= 4 8 13 18 9
clique= 4 8 13 18 17
clique= 4 8 13 18 10
clique= 4 14 5 3 12
clique= 4 14 5 3 16
clique= 4 14 5 15 12
clique= 4 14 5 15 16
clique= 4 14 13 12 3
clique= 4 14 13 12 15
clique= 4 14 13 16 15 ⑤
clique= 4 14 13 16 9

```
advice:
same destination:
1,6,11      ALU1.IN2
5,15,17     F
2,8,10      ALU1.IN1
3,7         ALU2.IN1
9,12        ALU2.IN2


combine:
1,6,11       MUX1
3,7          MUX2
5,17,18      MUX3
2,8,10       MUX4
13,14,15,16  MUX5
9,12         MUX6
```

■ Example algorithm 1 method 1.

Step1.Numbering the variables:
1. size
2. i
3. reg7

C, 0, and 'data' are single signals
0000 and 1000 (8) are constants and must always be implemented as a single register.

Step2.Trace table

	1	2	3
1	1	0	0
2	1	1	0
3	1	1	1
4	1	1	1
5	1	1	1
6	0	1	1
7	0	1	1

clique = 1
clique = 2
clique = 3

Step3.Operators

- comp1 comp2
1 2 3

	1	2	3
1			
2		8	
3			

clique = 2,3

/* one comparator can be used */

Step4.Interconnection

Source	Destination	Label	1	2	3	4	5	6	7	8	9
size	i	1	1				1			2	
data	reg7	2	2		2						
0	REG7	3	3								
8	ALU1.IN1	4	4								
size	ALU1.IN2	5	5								
i	ALU2.IN1	6	6								
0000	ALU2.IN2	7	7								
ALU1.OUT	i	8	8								
ALU2.OUT	c	9	9								

clique = 1,5 /* same source */
clique = 4

clique = 1,6 /* same destination */
clique = 2,3

Example algorithm 2 method 1.

Step1.Variables
'size' and 'i' are not compatible, they are both used at the same times. reg[0], reg[1], etc. are all one bit long. They are also used at the same times (parallel load), so they cannot be combined.

Step2. Operators
There is only one operator

Step3. Interconnection

Source	Destination	Label	1	2	3	4	5	6	7	8	9	10	11	12	13
8	i	1	1												
reg[1]	reg[0]	2	2												
reg[2]	reg[1]	3	3												
reg[3]	reg[2]	4	4												
reg[4]	reg[3]	5	5												
data	reg[4]	6	6									2	1	1	1
reg[6]	reg[5]	7	7										2		
reg[7]	reg[6]	8	8											2	
0	reg[7]	9	9												2
reg[5]	reg[4]	10	10												
data	reg[5]	11	11											1	1
data	reg[6]	12	12												1
data	reg[7]	13	13												
i	comp1.IN1	14													
0	comp1.IN2	15													
comp1.OUT	C	16													

```
clique = 6,11,12,13 /* same source */

clique = 6,10 /* same destination */
clique = 7,11
clique = 8,12
clique = 9,13
```