# SPECIFICATION OF DIGITAL SYSTEMS

G. A. Blaauw

Rapporteur : Dr. S. K. Shrivastava

## Introduction

The design of a digital system starts with the specification of the architecture of the system and continues with its implementation and its subsequent realisation. The talk concentrates upon the first two design areas, as is shown in figure 1. For reference the subjects taught in our department are given in figure 2.

## Design areas

A few definitions are in order, so as to avoid any confusion (see figure 3). By architecture I mean 'appearance to the user' - it is the functional specification of the system (its behavioural appearance). By implementation I mean 'internal logical organisation which performs the functions specified by the architecture' and by realisation I mean 'the physical components in which the logical organisation is embodied'. The main characteristics of these areas in terms of product, language requirements, purpose, and quality are shown in figure 4. For example, from the figure we see that the purpose of architecture is to provide a function. Once that function is established, the purpose of implementation is to give a proper cost-performance and the purpose of realisation is to build and maintain the appropriate logical organisation.

As an example, when we design a multiprocessing system, we must ask the question 'Do we mean the implementation, the architecture or both?' In case we talk about the architecture we want a system with the outer appearnace of multiple processes going on simultaneously. The system need not be implemented that way - quite often there is only one processor available. When looking at multiprocessing from an implementation point of view, our purpose is reliability or performance. If may be necessary to adhere to the outer appearance of a single process (see figure 5).

## Design languages

The many types of design descriptions that are employed are summarised in figure 6. The written description can not be missed - an informal description is essential. It is ambiguous, however- what is more, any attempts to make it unambiguous may make the description unreadable - like an insurance policy! Timing diagrams, flow diagrams etc. are useful as an illustration. They supplement the written description, but since they are not complete they do not remove the ambiguities. What is necessary is to supplement the written description with a rigorous description in an irredundant formal language. It is preferable to use both methods side by side so that possible misconceptions in the written text and the illustrations can be resolved by the formal language

specifications, whereas the latter in turn can be explained by text.

Many formal languages are available for the description of hardware design; a few are shown in figure 7. Two of these languages have been used to some extent in actual machine design. Bell and Newel's PMS/ISP has been used for the PDP11 series of computers and Iverson's APL has been used for the description of the IBM system/360.

Desiderata

A hardware design language should be (i) sufficiently high level, (ii) conversational, (iii) general purpose, and (iv) structured (see figure 8).

(i) High Level : The language should be able to express easily, and directly the desired function. Also, it should avoid suggesting an implementation (for example, by introducing unnecessary concepts). It should avoid involved or clever language constructs and it should be understandable to the users (that is, readable, pronouncable and not cryptic).

(ii) Conversational : A language that is available conversationally aids the description by the elimination of errors through syntax checks. Also, the description is executable and thus demonstrates the architecture. It is of course not possible to verify the architecture - there is nothing to compare it with. What is possible is to check the implementation against the architecture. By selectively substituting implementation for architecture one can perform efficient simulation experiments.

(iii) General purpose : When the language is general purpose the same language can be used for all levels of design (architecture, implementation, realisation). Similarly, design tools necessary for testing, collecting performance statistics, editors, tracers etc. can also be built using the same language. Thus a general purpose language represents a desirable economy of thought for the designer.

(iv) Structured : A language that exhibits good structure permits the designer to show the structure of his design. It allows the designer to develop his specifications in a top-down way so that design details can be developed and expressed gradually.

The Digital Technique sub-department of the Twente University of Technology uses APL for the design of architecture and implementation (properly supplemented by a written description and illustrations). I have found that it adequately meets the above criteria.

Specification of the Implementation

The implementation is derived from the formal specification of the architecture. A first step may be to restate the architecture. For example, the architecture of a multiplication is given in figure 9(a) and is restated in figure 9(b). In figure 9(a), the two's complement interpretations of a signed integer, multiplier and multiplicand are multiplied to get a

product. The architecture specifies that the product is represented by the number of digits which is the sum of the lengths of multiplier and the multiplicand. Figure 9(b) on the other hand shows an equivalent multiplier architecture with extended operands. By extending the operands the representations can be treated as unsigned binary numbers, which simplifies the implementation.

The transition from such a (restated) architecture to an implementation is usually not self evident. For example the use of positional representation of most arithmetic processes originally required considerable imagination and many centuries of development, although by now it is familiar. The basic implementation algorithm, such as a ripple adder, which bridges the gap between architecture and implementation is called the initial implementation algorithm.

The architecture, the initial implementation, as well as all subsequent design algorithms can formally be stated in APL. Therefore the algorithms can be simulated and verified against each other, giving a controlled design process.

The initial algorithm uses operators that represent available components, such as AND and OR circuits, or system components, such as adders, counters, and registers of which the implementation is known. The initial algorithm can be modified to obtain a first rough optimum of cost and performance (for example a 'restoring division' can be modified to 'non-restoring division').

Once an optimal initial algorithm is obtained it can be separated in two parts: datapath and its control (see figure 10). The datapath contains gates which direct the flow of information through the path. The control specifies the setting of these gates.

## Specification of the architecture

### Quality.

A first subject concerned with the specification of an architecture is its evaluation. A good architecture should be consistent, that is with a partial knowledge of the system the remainder of the system can be predicted. The use of a consistent architectural specification leads us not to link what is independent, not to introduce what is immaterial, and not to restrict what is inherent, which are the concepts of orthogonality, propriety, and generality.

The stack description of the Burroughs B5500 illustrates how superfluous implementation details can enter the architectural specification. The B5500 stack is described as a 48-bit A register, a 48-bit B register, and a portion of core. Validity of information in the A and B registers is indicated by the control flip-flops AROF and BROF respectively. A 15-bit S register (stack pointer) addresses the top word in the core portion of the stack. Figure 11 shows that four configurations are possible, which functionally are all equivalent.

Every time some word of information is pushed on or popped

from the stack, one of these four possible configurations must be specified. The B5500 documentation manual, however, does not specify explicitly the resulting stack configuration. Presumably in the interest of brevity all changes of the stack are specified by the phrase: 'the description of the operations assumes the necessary stack adjustments'. This already suggests that it is unimportant to know which configuration at a particular moment is actual. Indeed, at a functional level it is of no interest to know of the existence of A or B registers: all we need to know is that there is a conceptual STACK, from which information can be popped or on which information can be pushed, irrespective whether part of this STACK resides in registers, and other parts in core.

As another example, figure 12 shows an inconsistency in the addressing of INTEL 8080. The 8080 addresses its bits from right to left. The byte addressing is also from right to left in memory. In registers, however, it is from left to right.

Steps

The architectural design process may be viewed as a top down process, as depicted in figure 13. We start with a list of user primitives - elementary functions, which the user needs to express himself directly and completely. At a lower level sets of architectural alternatives are proposed. They are evaluated by expressing the primitives in terms of these alternatives. Also the initial implementations for these alternatives are developed. Thus the ability to use and to build the architecture is tested. As an example consider the simple case of integer addition. The starting point for the architecture is some statement like: "we have two operands on which an operation is performed to give a result". The first step at the architectural level is to realise that we do not work with numbers but their representation (see figure 14). One starts with bit patterns which are interpreted as numbers. To store the result, one 'goes down' to its representation. There are exceptions however - as shown by the dotted line - when the result is no longer in the representable domain. What is needed is a domain function to bring the result back into the representable domain. The design of such a function is a crucial aspect of the addition architecture.

Starting with the mathematical concepts of addition, we can decompose the architecture in (i) interpretation, (ii) representation, (iii) operation, (iv) domain function, and (v) signalling (for indicating whether the result is +, -, 0, has an overflow, or a carry).

Since our representation range is limited, we must be able to extend it - hence we must satisfy the extended addition user primitive. We take the representation of multiple fields, interpret them as numbers, add them and represent them again. The procedure for this primitive must use the normal instructions like add. Thus the add instruction should signal a carry out and the carry must be added in a subsequent 'add with carry' operation. If we have signalling in single precision we should have the same signalling in extended precision. As far as I know, no machine does this properly (that is, tells whether the extended result is +, - or 0). The solution is to add the carry and set the signals such

that their previous value is taken into account.

Questions

Professor McKeeman questioned the suitability of APL as a specification language on the grounds that is not very readable. Professor Blaauw conceded that there is the initial temptation to 'write everything in a line'. Every description technique however has to be learned so that its tools are used properly. Professor Pyle asked whether there is any significant difference between a design/specification language and a programming language. The speaker replied that in essence there is little difference between the two; his remarks, however, apply specifically to the design of digital systems.

With reference to figure 9(a) and (b), Professor Dijkstra asked the speaker whether he has proved the equivalence between the two specifications of the same design. The speaker replied that while he has not proved the equivalence, as a designer, he has sufficient confidence in their equivalence. (The speaker was in error here; he has proved it on pages 74 and 75 of the book mentioned below.) Lastly, Professor Randell asked whether the speaker's recent book (Digital System Implementation, Prentice Hall, 1976) matches the courses given at his institution. Professor Blaauw replied that his book is used as a text book for his course on the implementation of digital systems on the system component level.

# SPECIFICATION OF DIGITAL SYSTEMS

INTRODUCTION

WORK AT TWENTE TECHNICAL UNIVERSITY

DESIGN AREAS

ARCHITECTURE, IMPLEMENTATION,
REALIZATION

DESIGN LANGUAGES

DESIDERATA

SPECIFICATION OF THE IMPLEMENTATION

STEPS

VERIFICATION

SPECIFICATION OF THE ARCHITECTURE

QUALITY

STEPS

Figure 1

# SUB-DEPARTMENT OF DIGITAL TECHNIQUE
## TWENTE UNIVERSITY OF TECHNOLOGY

| | | |
|---|---|---|
| **ARCHITECTURE** | | |
| 5 | COMPUTERS | BLAAUW<br>RAATGERINK<br>GEERDINK |
| 5 | INTERFACES | VISSERS |
| (5) | NETWORKS | V.D. DOLDER |
| **IMPLEMENTATION** | | |
| 3 | COMBINATORIAL AND SEQUENTIAL CIRCUITS | BONNEMA<br>V.D. KNAAP |
| 4 | SYSTEM COMPONENTS | BLAAUW<br>V.D. DOLDER |
| **REALISATION** | | |
| | PLACEMENT AND ROUTING | AL<br>V.D. KNAAP |
| 5 | REALISATION WITH MICROPROCESSORS | WILMINK |

Legend: The number indicates the year in the curriculum in which the lecture is given. The name of the lecturer is underlined. Other names indicate staff engaged in research in this area.

( ) in preparation.

Figure 2

# SOME DEFINITIONS

ARCHITECTURE

POSITIVE : APPEARANCE TO USER

NEGATIVE : INNER STRUCTURE NOT KNOWN

IMPLEMENTATION

LOGICAL STRUCTURE

WHICH PERFORMS ARCHITECTURE

REALISATION

PHYSICAL STRUCTURE

WHICH EMBODIES IMPLEMENTATION

Figure 3

| | ARCHITECTURE | IMPLEMENTATION | REALISATION |
|---|---|---|---|
| PRODUCT | PRINCIPLES OF OPERATION | LOGICAL DESIGN | RELEASE TO MANUFACTURING |
| .ANGUAGE | WRITTEN ENGLISH<br>FORMAL DESCRIPTION | BLOCK DIAGRAM<br>LOGICAL EXPRESSIONS | LISTS AND DIAGRAMS |
| 'URPOSE | FUNCTION | COST/PERFORMANCE | BUILD AND MAINTAIN |
| UALITY | CONSISTENCY | WIDE SCOPE | RELIABILITY |

Figure 4

MULTIPROCESSING

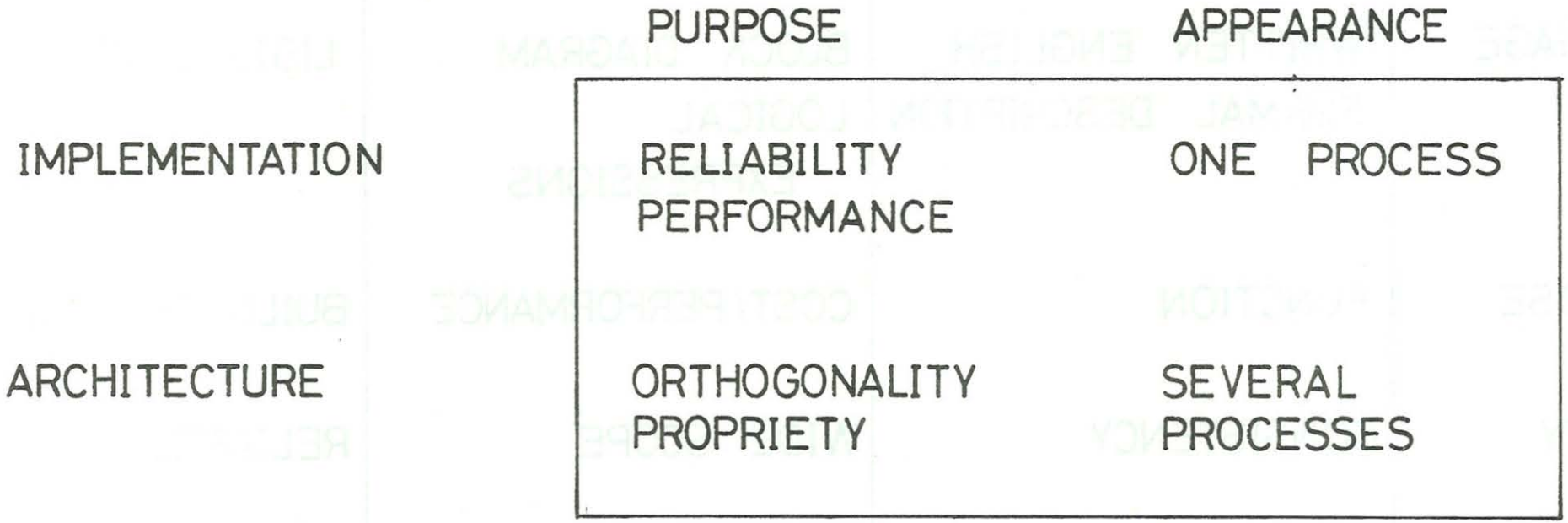



Figure 5

| TYPES OF DESCRIPTION | COMMENTS |
|---|---|
| WRITTEN | NEEDED |
| TIMING DIAGRAMS | ILLUSTRATION |
| STATE DIAGRAMS | |
| BLOCK DIAGRAMS | |
| FLOW DIAGRAMS | |
| NASSI - SCHNEIDERMAN CHARTS | TOOL |
| HIPO'S | |
| DECISION TABLES | |
| FORMAL STATEMENTS | AUTHORATIVE |

Figure 6 : Description of design

# A FEW HARDWARE DESIGN LANGUAGES

| | |
|---|---|
| 1952 | RTL, REED |
| 1962 | APL, IVERSON |
| 1964 | RTL, SCHORR<br>LOTIS, SCHLAEPPI |
| 1965 | CDL, CHU |
| 1966 | CASSANORE, MERMET |
| 1967 | DDL, DULEY & DIETHEYER |
| 1968 | AHPL, HILL & PETERSON |
| 1969 | ALERT, FRIEDMAN |
| 1970 | PMS, BELL & NEWELL<br>ISP |
| 1973 | AHPL, HILL & PETERSON |

ALSO: HARGOL, APDL, HDL, CONLAN

Figure 7

# DESIDERATA FOR DESIGN LANGUAGES

HIGH LEVEL
CONVERSATIONAL
GENERAL
STRUCTURED

Figure 8

```
∇ ARCHMPY;PRODUCT
[1]   PRODUCT←(TWOC VL)+(TWOC MR)×TWOC MD
[2]   PD←((ρMR,MD)ρ2)⊤PRODUCT
∇

∇ N←TWOC R
[1]   N←2⊥(¯1↓R),R
∇
```

PROGRAM 3-1 MULTIPLIER ARCHITECTURE

Figure 9(a)

```
∇ ARCHMPYX;VLX;MRX;MDX;PRODUCT
[1]   ⍝ OPERAND EXTENSION
[2]   VLX←(ρMR) EXTEND VL
[3]   MRX←(ρMD) EXTEND MR
[4]   MDX←(ρMR) EXTEND MD
[5]   ⍝ MULTIPLICATION
[6]   PRODUCT←(2⊥VLX)+(2⊥MRX)×2⊥MDX
[7]   PD←((ρMRX)ρ2)⊤PRODUCT
∇

∇ RX←N EXTEND R
[1]   RX←R [Nρ0],R
∇
```

PROGRAM 3-3 MULTIPLIER ARCHITECTURE WITH EXTENDED OPERANDS

Figure 9(b)

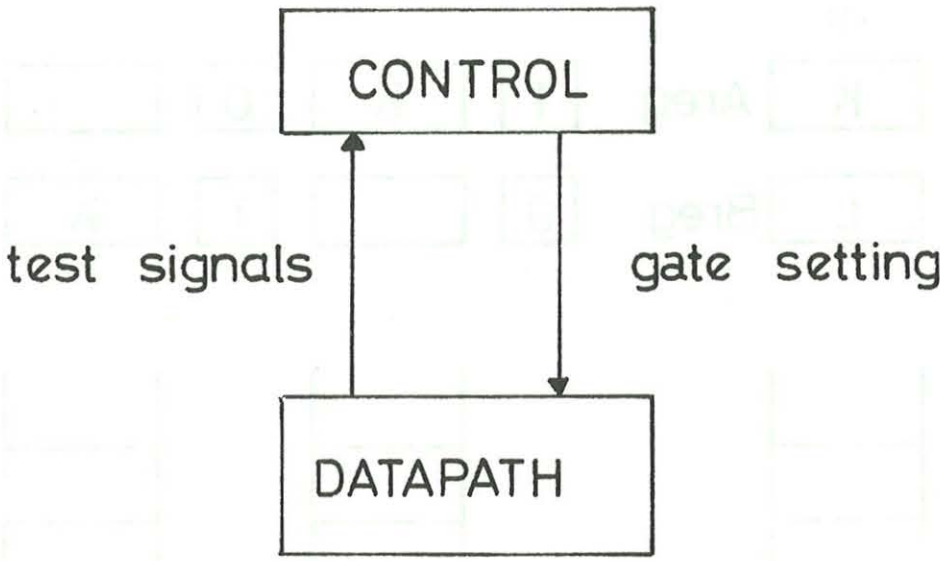



Figure 10 : Relation between datapath and control

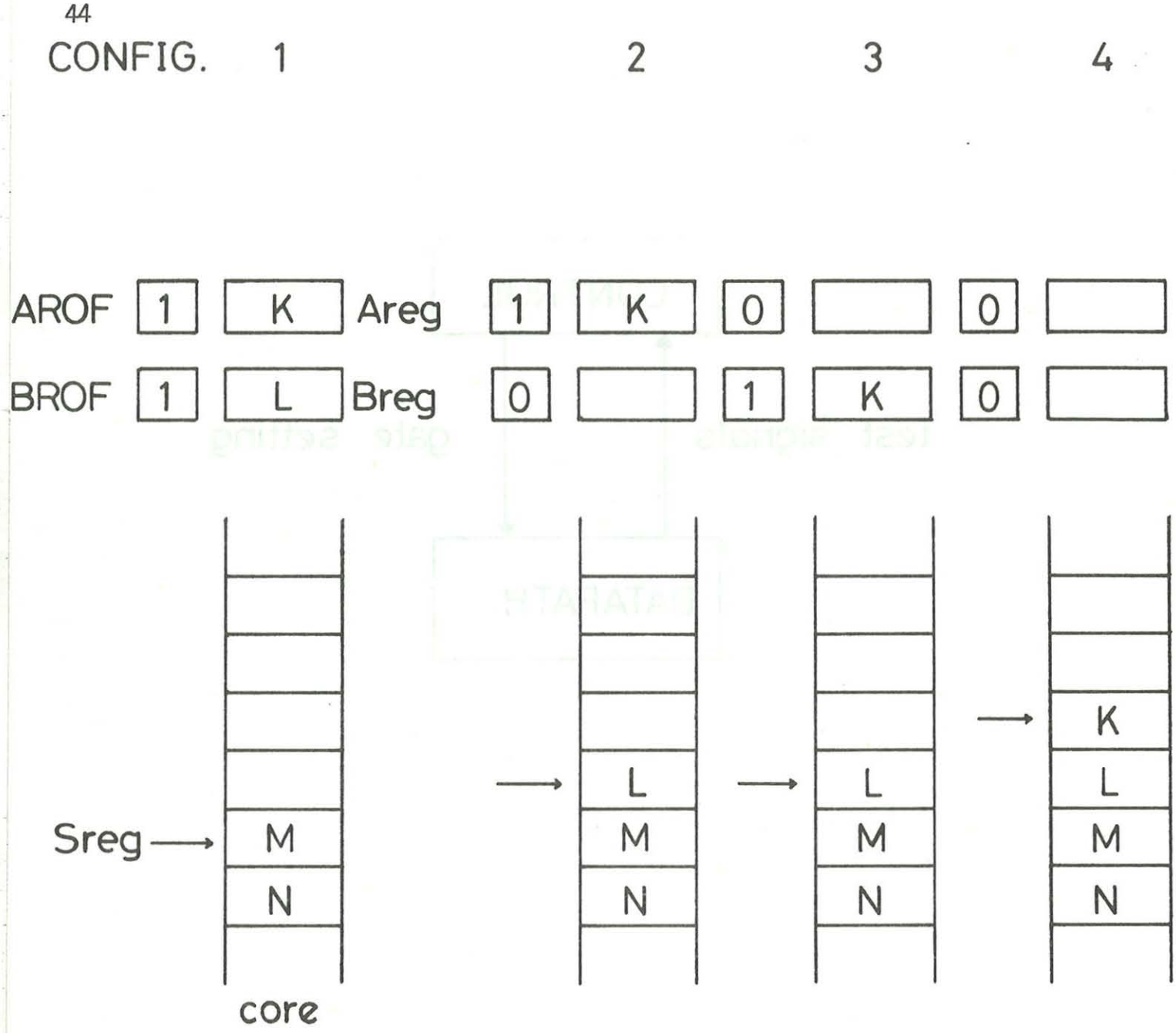
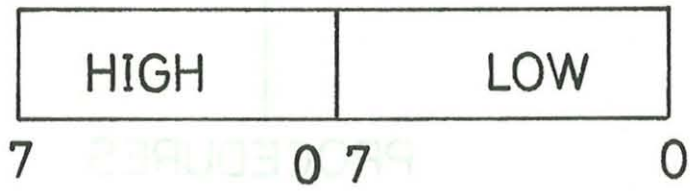
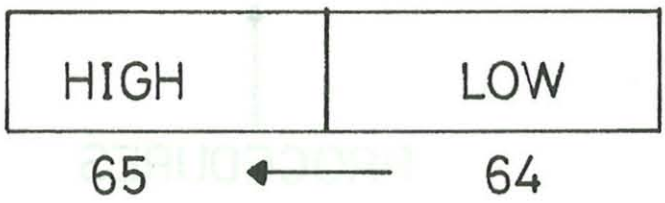



Figure 11 : Possible configurations of words in a B5500 stack

DIGIT ADDRESSING:




BYTE ADDRESSING:
IN MEMORY




IN REGISTERS

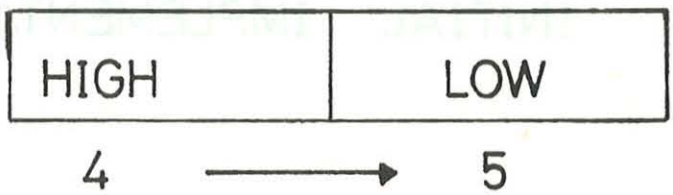



Figure 12 : Intel 8080 Addressing Direction

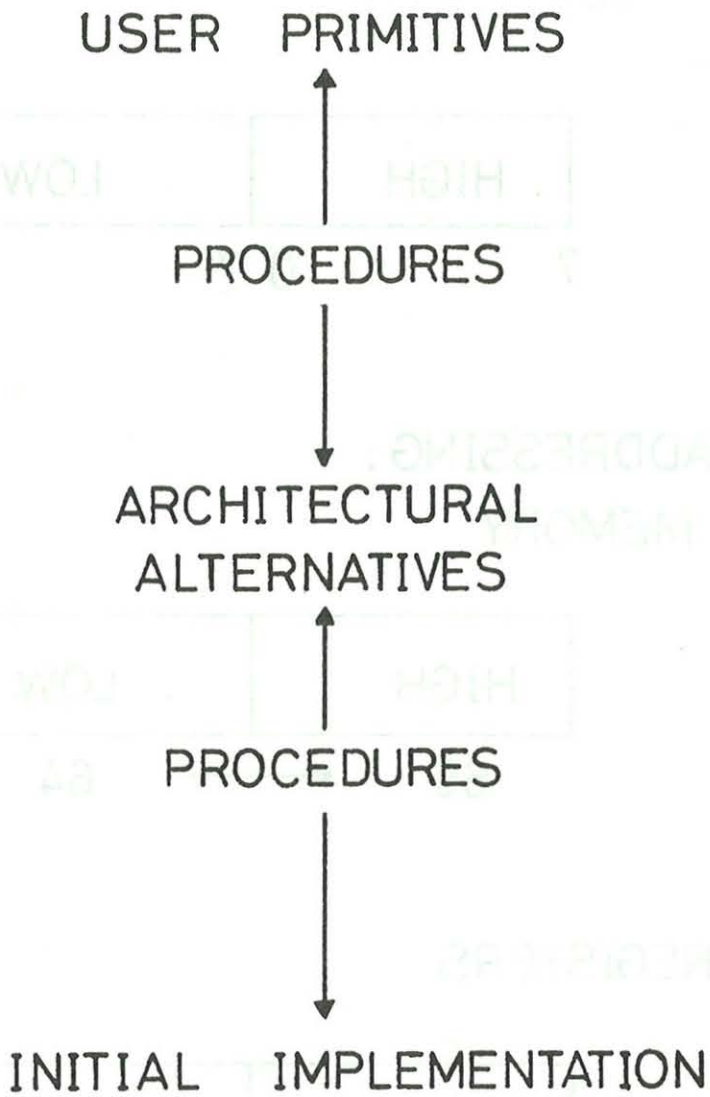



Figure 13 : Design Process

# EXAMPLE: ARITHMETIC

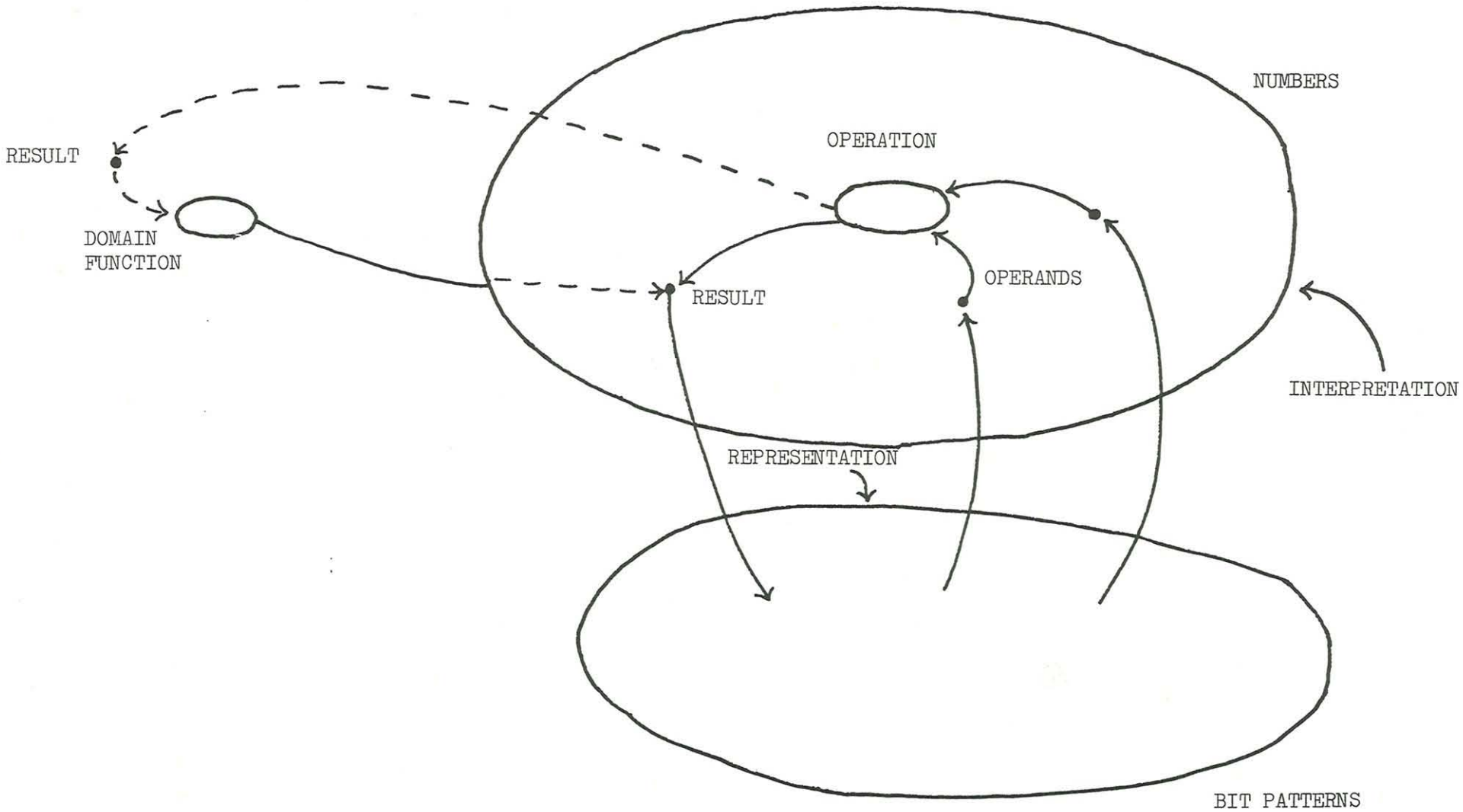



Figure 14