**NOTICE WARNING CONCERNING COPYRIGHT RESTRICTIONS:**
The copyright law of the United States (title 17, U.S. Code) governs the making of photocopies or other reproductions of copyrighted material. Any copying of this document without permission of its author may be prohibited by law.

CMU-CS-79-123

# Instruction Set Processor Specifications (ISPS): The Notation and its Applications

Mario R. Barbacci
Department of Computer Science
Carnegie-Mellon University
17 May 1979

Copyright -C- 1979 Mario R. Barbacci

The development of ISPS is part of the research on the Symbolic Manipulation of Computer Descriptions effort at CMU and is sponsored by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 3597, monitored by the Air Force Avionics Laboratory Under Contract F33615-78-C-1551.

The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the US Government.

UNIVERSITY LIBRARIES
[illegible]
[illegible]

# Table of Contents

# 1. Introduction

The ISPS computer description language is an evolutionary step towards the formalization of the digital design process at the higher or behavioral levels. ISPS is the second implementation of ISP as a computer language[1] and has been used as a design tool which covers a wider area of application that any other hardware description language. Thus, besides simulation and synthesis of hardware, software generation, program verification, and architecture evaluation and control are amongst the current applications based on ISPS. The range of current and contemplated application areas are proof of the usefulness of the notation and its extension mechanisms.

This paper is divided into two parts. The first part describes the notation, its intended use, and the extension mechanisms which allow multiple applications or areas of research to co-exist and share machine descriptions. The second part briefly describes some of the current applications for ISPS; only enough detail is presented to illustrate the highly diverse set of problem areas that depend on a formal machine description.

The appendix presents some language features not often found in programming languages (or machine description languages, for that matter). Topically, the appendix belongs after the first part however, it is not critical to the understanding of the of the applications for ISPS and has been postponed to allow a smooth transition between the description of the notation and the description of its uses.

[1]In this paper we shall make no distinction between ISP, the original notation [Bell, 1971], and its implementations as computer languages, ISPL [Barbacci, 1976a] and ISPS [Barbacci, 1977]. All examples used in this paper are based on ISPS.

# 2. The Notation

## 2.1 The ISPS Paradigm

High level programming languages reduce the complexity of the programming task and increase reliability and readability through the use of abstractions.

By using abstractions, a program can be structured as a hierarchy of functions and data structures. The proper use of these mechanisms allows a program to be developped in a step-wise process. Advocates of "top-down" versus "bottom-up" techniques argue about the relative merits of their approach. The arguments however, have more to do with the structure or order of the program design process than with the nature of the process itself.

The same concerns that motivate the use of high level programming languages appear in the digital system design process and many of the structured programming techniques can be put to use in what has been traditionally the domain of hardware engineers. However, although the methods can be borrowed, the differences in the problem domain suggest the use of tools and notations specially designed for digital systems.

The design philosophy of ISPS was guided by two principles, flexibility and simplicity. Specifically, it was desired to design a computer description language that would be appropriate for diverse applications: automated design, simulation (for both software development and hardware debugging), and automatic generation of machine relative software (in particular, compiler-compilers), [Barbacci, 1974b]. Thus, although ISPS can be viewed as a programming language, the aim of the notation is to describe computers and other digital systems, not necessarily general computational algorithms.

The ISPS paradigm is shown in Figure 2-1. The main characteristic is the use of a formally defined intermediate representation for the parse trees. This intermediate format (called Global Data Base or GDB, for short) can be easily used by a multitude of application programs, written in any language, and running on any machine.

The definition of what constitutes a 'correct' ISPS description depends to some extent on the nature of the application programs using the machine description. An assembler generator (e.g. [Wick, 1975]) might, for instance, require the specification of the instruction mnemonics but it might not have any use for the specification of the memory technology. The situation is reversed when a design automation system (e.g. [Siewiorek, 1976]) uses the same parse tree. A compiler-compiler (e.g. [Leverett, 1979]) system might be interested in the 'cost' of each instruction in order to generate optimal code.

To allow the co-existence of multiple applications, ISPS provides an extension facility for

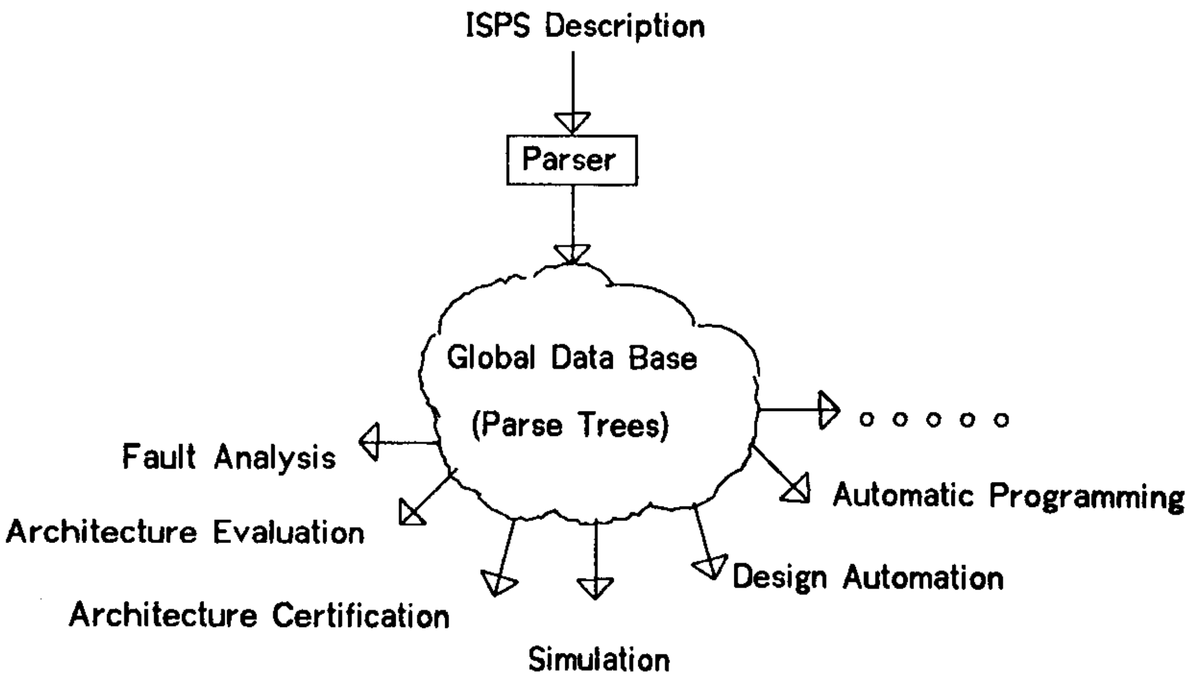


Figure 2-1: ISPS Paradigm

the specification of application dependent information. This information is attached to the parse trees and can be easily retrieved by the application programs. Because of the open ended nature of the application dependent information, the parser can only perform syntactic analysis of the extensions. Relatively little can be done at parse time with regard to the semantic analysis and the bulk of the semantic analysis of the extensions thus lies in the domain of the application areas[1].

The format of the parse trees used by the application programs is defined in the reference manual ([Barbacci, 1977]). For the purposes of this paper we are only concerned with the language constructs used to specify the application dependent information.

[1]As experience with the language grows, the semantic knowledge built into the parser will be augmented to incorporate those aspects that are common to all applications or which can result in contradictory assumptions by the users of the machine description.

## 2.2 The ISPS Notation

ISPS describes the interface (i.e. external structure) and the behavior of hardware units (called entities in the language). The interface describes the number and types of carriers used to store and transmit information between the units. The behavioral aspects of the unit are described by procedures which specify the sequence of control and data operations in the machine.

A complete separation between the specification of the structure and behavior of a digital system is not an easily realizable or even desirable goal. Structure and behavior go hand in hand and its is a measure of the power of any design language to be able to enhance one aspect over the other. Thus, ISPS favors the behavioral aspects over the structural or implementation aspects by hiding (abstracting) information. The structural information is never completely eliminated; thus, the preocupation of ISPS with register lengths, data path widths, connections of registers and functional units, etc; other details such as part numbers, component speed, layouts, physical location, integrated circuit technology, etc. are however, successfully opaqued and need never be specified in an ISPS description[1].

In the simplest case, a unit is simply a storage carrier (a register or a memory), completely specified by its bit and word dimensions:

```
Accumulator<0:15>                                  a 16-bit register
R[0:7]<15:0>                                       a register array
Mp[0:#177777]<0:7>                                 a 16K byte memory
```

In the general case, a unit may consists of an interface and a procedure which describes its behavior. The procedural part may contain not only data and control operations, but also the declaration of local units of arbitrary complexity. Local units are not accessible to external units, allowing the encapsulation of portions of the design in a well structured

[1]The fact that they can be easily specified by the designer does not alter the argument. The language allows the specification of these and other details for the benefit of some but not necessarily all application programs. Thus, the number of bits in a register is of concern to all applications since it impinges on the behavior of the system. The fact that it is build in MOS instead of Bipolar is important only to say, a synthesis program, it is not important at all to a program verifier.

manner[2].

```
Mark1 :=                                      The Mark-1 Computer, [Lavington, 1975]
      Begin
      ** Primary.Memory **                               Primary Memory Section
      M[0:8191]<31:0>,                                    8K words, 32 bits/word

      ** Central.Processor **                            Central Processor Section
      Processor :=
            Begin
            ** Processor.State **                        Processor State Section
            PI\Present.Instruction<15:0>,                  Instruction Register
                  F\Function<0:2> := PI<15:13>,                          Opcode
                  S<0:12> := PI<12:0>,                          Operand Address
            CR\Control.Register<12:0>,                          Program Counter
            Acc\Accumulator<31:0>,

            ** Instruction.Cycle **          Instruction Interpretation Section
            I.Cycle :=
                  Begin
                  PI = M[CR]<15:0> next                       Instruction Fetch
                  Decode F =>                              Instruction Decoding
                        Begin
                  0\JMP   := CR = M[S],                           Jump Indirect
                  1\JRP   := CR = CR + M[S],                      Jump Relative
                  2\LDN   := Acc = - M[S],                      Load Complement
                  3\STO   := M[S] = Acc,                                  Store
                  4:5\SUB := Acc = Acc - M[S],                         Subtract
                  6\CMP   := If Acc Lss 0 =>                   Conditional Skip
                                   CR = CR + 1,
                  7\STP   := Stop(),                                       Halt
                        End next
                  CR = CR + 1 next                  Increment Program Counter
                  Restart I.Cycle                     Repeat Instruction Cycle
                  End
            End
      End
```

Figure 2-2: The MARK-1 Computer

[2]The same rules of scope introduced by Algol-60 are used in ISPS.

UNIVERSITY LIBRARIES
[illegible]

Figure 2-2 shows the complete description of the Manchester University Mark-1 Computer [Lavington, 1975]. The description consists of two sections depicting the primary memory and the central processor. The latter is further divided into the processor state declarations and the instruction interpretation cycle. The entire behavior of the computer is described by a single procedure (ICYCLE) which fetches, decodes, and executes the instructions. For additional examples of ISPS descriptions the reader should consult [Bell, 1978].

## 2.3 Application Dependent Information

*Premature binding is the root of all evil*

ISPS provides mechanisms by which an application program can receive private or semi-private information about a computer description. The type of information that might be needed by an application program is unbound and not, necessarily, part of the basic semantics of the notation understood by the parser.

Application dependent information is specified via qualifiers or attribute-value pairs of the form:

```
{attribute:value,value,...;attribute:value,value,...}
```

Attributes are user (i.e. application dependent) specified identifiers; Values are either identifiers, constants, or text strings. Qualifiers can be attached to any declaration, operation, or operand in a description. They are checked for syntactic correctness by the ISPS parser, but no attempt is made to ascertain their semantic correctness[1].

While writing the ISPS description, related units can be grouped under a common heading (called a section), as shown in Figure 2.3. By dividing a description into sections, application programs can search parse trees for specific groups of declarations (say, all floating point operations, processor state registers, external interface data and control signals, etc.). Sections do not define new scopes of declarations (i.e. they do not define Algol-like blocks). A section is therefore an organizational device, not an abstraction. A section header (an identifier enclosed in **s) can have qualifiers attached to it to permit the specification of attributes that are common to all units declared within the section.

Delaying some of the semantic checks until application time permits a graceful evolution of a design. New applications can be developed and experiments with machine descriptions can

[1]Unless of course, the attribute is already known to the parser.

```
unit1
        ** section 1 **
        unit1.1
        unit1.2
        .....
        ** section 2 **
        unit1.3
        unit1.4
        .....
unit2
        ** section 3**
        unit2.1
        unit2.2
        .....
        ** section 4 **
        unit2.3
        unit2.4
        .....
.....
```

Figure 2-3: Partition of a Description into Sections

be carried out without having to modify the ISPS parser, the global data base, other application programs, or any other existing description.

To illustrate a simple use of the extension mechanism, the following section shows the type of information that could be conveyed to a synthesis program.

## 2.4 Example: Component Specification

Data and control operations in ISPS suggest the existence of corresponding functional units and data paths but not to the extent that they constrain the implementation. Thus, the register transfer operators "<=", and "=" indicate the existence of a logical path between registers of the machine but they do not specify an actual data path. The implementor (human or synthesis program) is free to realize all transfers with a single, shared bus, with a set of buses, or even with dedicated buses for each register transfer. By the same token, arithmetic and logical operators suggest the existence of functional units. The actual number, type, and interconnection is left open. An implementor is free to select these units in a variety of ways, depending on the cost, speed, size, reliability, etc. required of the target system.

The lack of restrictions in the implementation of an ISPS description might not be a desirable feature and one might wish to constraint the implementor. For instance, declarations of registers and functional units can be qualified with the component part to be used:

```
R[0:3]<0:3> {Module: SN74170}                    register file declaration
X = Y SLO {Module:SN54199} 3                         logical shift left
```

In some cases the attribute name is sufficient to convey the information and a simplified mechanism is provided:

```
M[0:255]<0:7> {ROM}                              Attribute as Qualifier
ROM M[0:255]<0:7>!                                 Attribute as Keyword
```

In the second example, ROM is used as if it were a keyword specifying the 'type' of the declaration. ROM is not a real keyword in the language and the syntax allows the specification of an arbitrary number of these pseudo-keywords prefixing a declaration. They are all collected by the parser and stored in the global data base format as if they had been specified inside { and }. Thus, both declarations of the Read Only Memory (M) are equivalent.

The ability to specify the behavior of an entire family of implementations allows a manufacturer to apply different trade-offs and produce machines aimed at different segments of the market while preserving the basic compatibility between all the members of the family, the ability of sharing software.

The ability to partially specify the structure and components of a system permits the manufacturer to place some bounds on the cost and performance of the implementations. In general, the more restricted the description, the less variability among members of the family. These however, are design decisions which ISPS leaves entirely in the hands of the architect of the system.

Since the behavioral aspects do not change, other by-products of the description (e.g. compilers, program verifiers, architecture evaluations, etc) can be obtained before, after, or even during the specification of the implementation information.

# 3. The Applications

## 3.1 Evaluation and Certification of Instruction Set Processors

Selecting a computing architecture is not a well established science. The problem is particularly severe when one tries to choose an architecture meant to serve a broad range of users whose present and future requirements are poorly understood. Recent work within the Department of Defense has resulted in the development of a methodology to specify, evaluate, and select candidate computer architectures for tactical applications [Burr, 1977; Wald, 1977; Fuller, 1978; Dietz, 1978].

### 3.1.1 The CFA project

The concept of writing benchmarks or test programs is not a new idea in the field of computer performance evaluation. The main difference in the approach used in the CFA project was the departure from the traditional measured gathered from typical computer performance studies, namely the execution speed of a test program. A computer architecture does not specify the instruction execution times and the following alternative measures were developed:

1. S -- Number of bytes used to represent a test program.
2. M -- Number of bytes transferred between primary memory and the processor during the execution of a test program.
3. R -- Number of bytes transferred among (selected) internal registers of the processor during the execution of the test program.

The S measure is an indication of the amount of memory needed to represent the programs running on a computer. For the same technology, two architectures that require different amounts of memory would have different costs. A small S measure is a good feature of a computer architecture.

The M and R measures characterize the bandwidth of the data paths between the central processor and the primary memory, and between the internal registers of the central processor, respectively. For the same technology, a higher M measure implies a larger volume of information that has to be transferred, thus implying a slower instruction rate or a costlier implementation. Similarly, a higher R measure implies a slower instruction rate or costlier implementation. Notice that techniques such as memory interleaving, cache memories, and pipelines could be used to improve the performance of a system. However, these are implementation details and are not part of the architecture. That is, two architectures with different M (or R) measures could be implemented using these and other speed-up mechanisms. The relative ranking given by their M (R) measures would not change.

### 3.1.2 Data Collection

The experiments were conducted on the ISPS simulation facility depicted in Figure 3-1 ([Barbacci, 1976b]). The process starts with the creation of an ISPS description, as shown at the top of the figure. The ISPS parse tree is processed by a program (GDBRTM) which generates code for an artificial machine. This machine is dubbed the Register Transfer Machine (RTM) and its order code was selected to suit the syntax and semantics of ISPS (e.g. there is one RTM operation for each ISPS data or control operation). The ISPS simulator is simply a software implementation of the RTM machine.

Throughout the execution of the RTM code, the simulator keeps count of ALL activities. These can be categorized into three classes:

1. Counting the bits read from each register or memory declared in the ISPS description.
2. Counting the bits written into each register or memory declared in the ISPS description.
3. Counting the number of times each procedure or labelled statement declared in the ISPS description has been encountered (i.e. executed).

The architecture evaluation is based on the computation of the S, M, and R for a collection of benchmark or test programs. These programs are assembled into a simulator command file which loads the simulated memory and machine registers. Under control of the user, the program thus loaded can be started and run to completion, at which point the activity counters can be dumped into computer files for post-processing, as depicted in figure 3-2.

### 3.1.3 Architecture Certification

The ISPS parser, the RTM code generator, and the simulator, together with a a number of auxiliary programs (e.g. cross-assemblers, cross-compilers, data reduction routines, etc) have been used in several architecture evaluation projects, patterned after the CFA selection project. The same tools have also been used to verify the correctness of the machine descriptions by running the manufacturer's machine diagnostics on the simulated machine [Barbacci, 1978a].

A current research project is attempting to generate the machine diagnostics from the ISPS description [Lai, 1979]. The difference between these two uses of the ISPS notation should

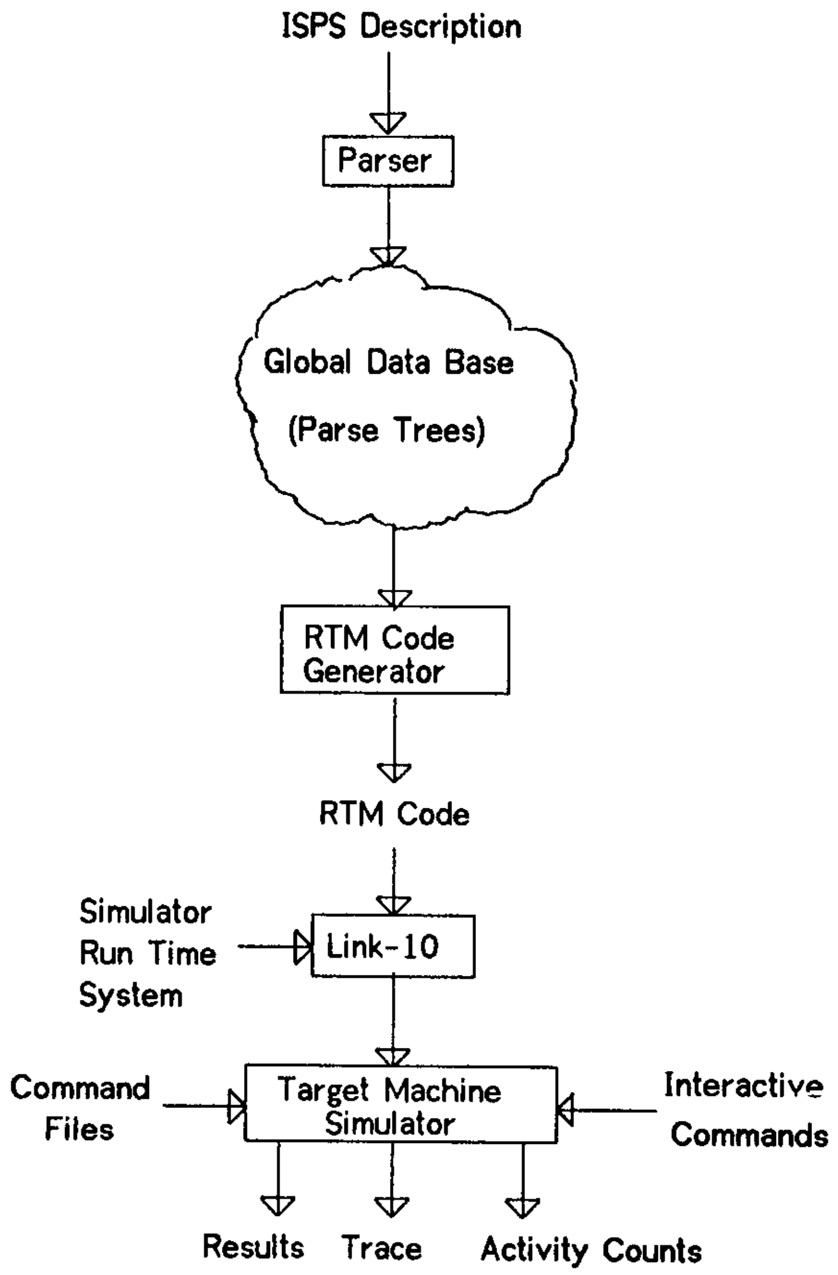


Figure 3-1: ISPS Simulation

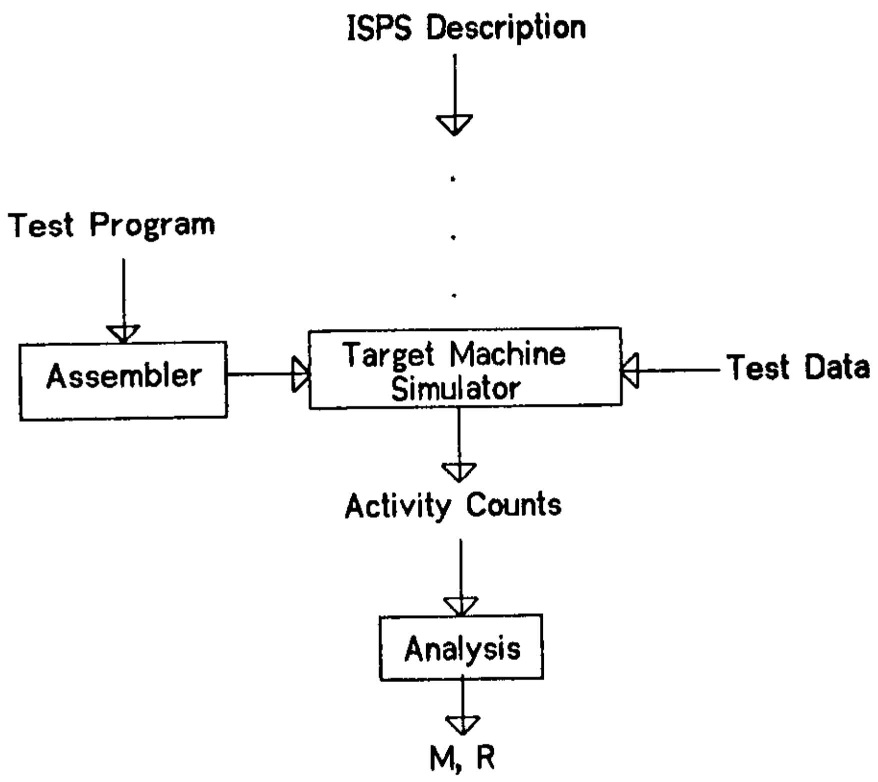


Figure 3-2: Architecture Evaluation

be explained. Verifying the correctness of a machine description by treating it as one more implementation of the machine, capable of correctly executing the manufacturer provided diagnostics, increases our level of confidence in the evaluation results. Once a description has been so certified, it can then be used as a procurement document, available to second-source manufacturers [Barbacci, 1979]. The second application complements and expands this use of ISPS. Any implementation of a computer, whether based on an existing architecture or a completely new architecture, must be certified as correct. By performing a symbolic execution of the ISPS description ([Oakley, 1979]), a list of all architecture features that must be present in any implementation can be compiled and used as a check list to generate the certification programs[1], as shown in Figure 3-3.

[1]It should be mentioned that although Oakley's work had the generation of diagnostics as the initial goal, it was soon evident that the range of its application was much broader. For instance, we are currently interfacing the output of the symbolic executor with the input required by the Production Quality Compiler Compiler (PQCC) project [Leverett, 1979]. This application is described later in this paper.

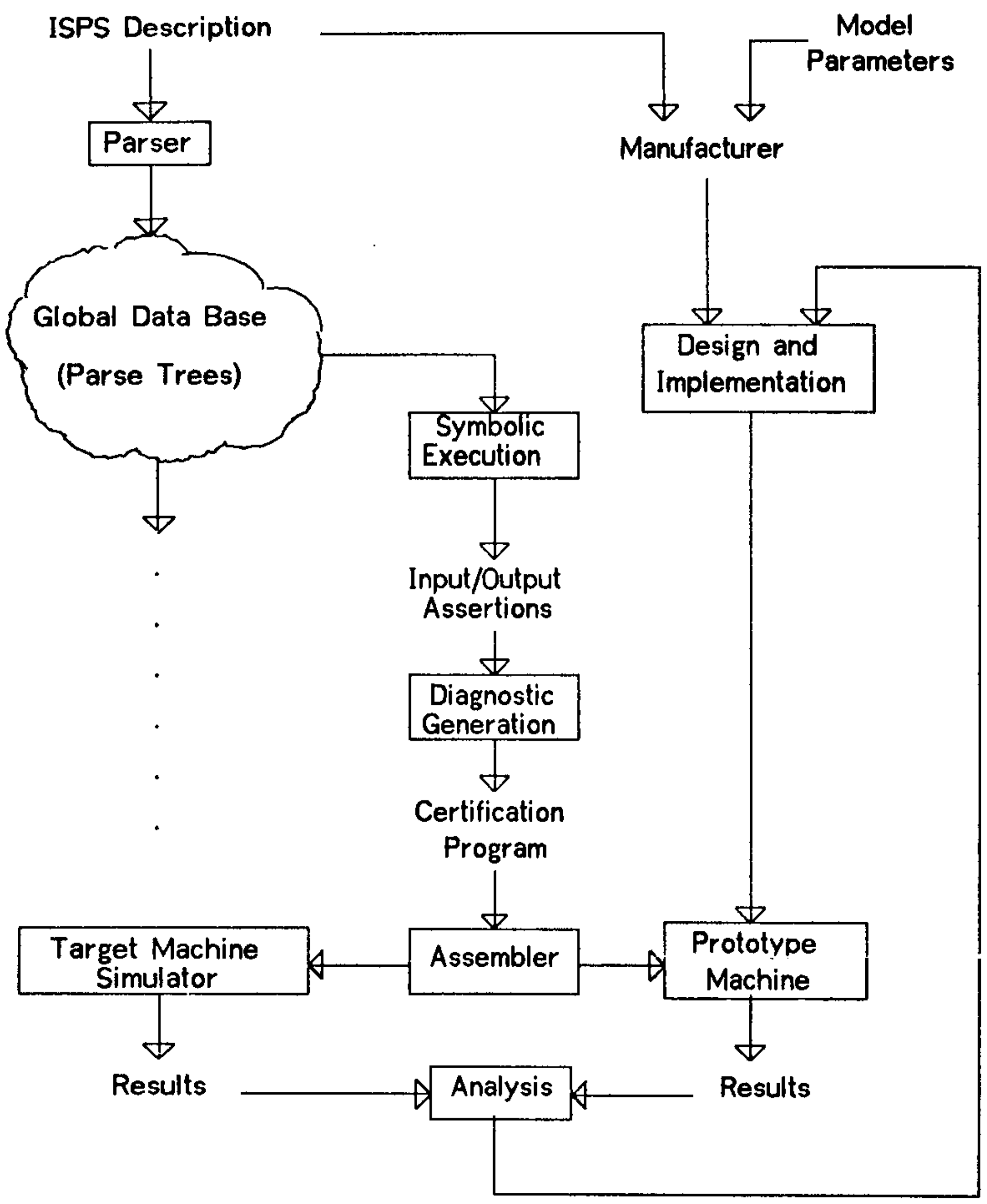


Figure 3-3: Certification of Architecture Implementations

This approach provides better coverage than the standard machine diagnostics currently in use. Even if model dependencies must be allowed, the automatic generation of the check-list provides a good mechanism to document and highlight departures from the architecture specification.

## 3.2 Technology Relative Synthesis

The use of Large Scale Integration (LSI) has made complete processors and memories available in a single silicon chip. Two aspects of this technological revolution are worth discussing: 1) primitive components continue to increase in complexity, and 2) the rate of introduction of new components continues to increase.

The increased power of the primitive components has reduced the need to descend into the lower levels of design. There exist standard families of components that allow designers to remain entirely within the register transfer level. While this simplifies the design task, the rapid evolution of components on the other hand, requires an acceleration in the design process if a new technology is to realize its potentiality. This can only be achieved through automatic means.

Conventional design automation systems tend to be characterized by a fixed, built-in technology and a straightforward or canonical implementation philosophy. That is, the designer's specification is translated into hardware specifications in a manner very similar to a macro-expansion in an assembly language program.

The CMU RT-CAD System [Siewiorek, 1976; Parker, 1979], Figure 3-4, attempts to eliminate both constraints. The system accepts the description of the components as one of its inputs, thus speeding up the incorporation of new technologies into the design process. Since the system operates on an abstract or symbolic description of the modules, a non-existing module set can be fed to the system for experimentation and evaluation of the potential advantages and disadvantages of a proposed module or set of modules. The second input to the system is an ISPS description of the behavior of the target system. This specification is first translated into a graph representation of behavior. By using a set of graph transformation algorithms, this initial graph can be transformed into alternative graphs, all of which represent the same behavior.

These alternative designs constitute points in the design space, [Barbacci, 1974]. Given two points and a set of user goals and constraints, the design system can automatically accept or reject an alternative design. A new design is acceptable if it is closer to the desired goal than the design alternative from which it was derived, as depicted in Figure 3-5.

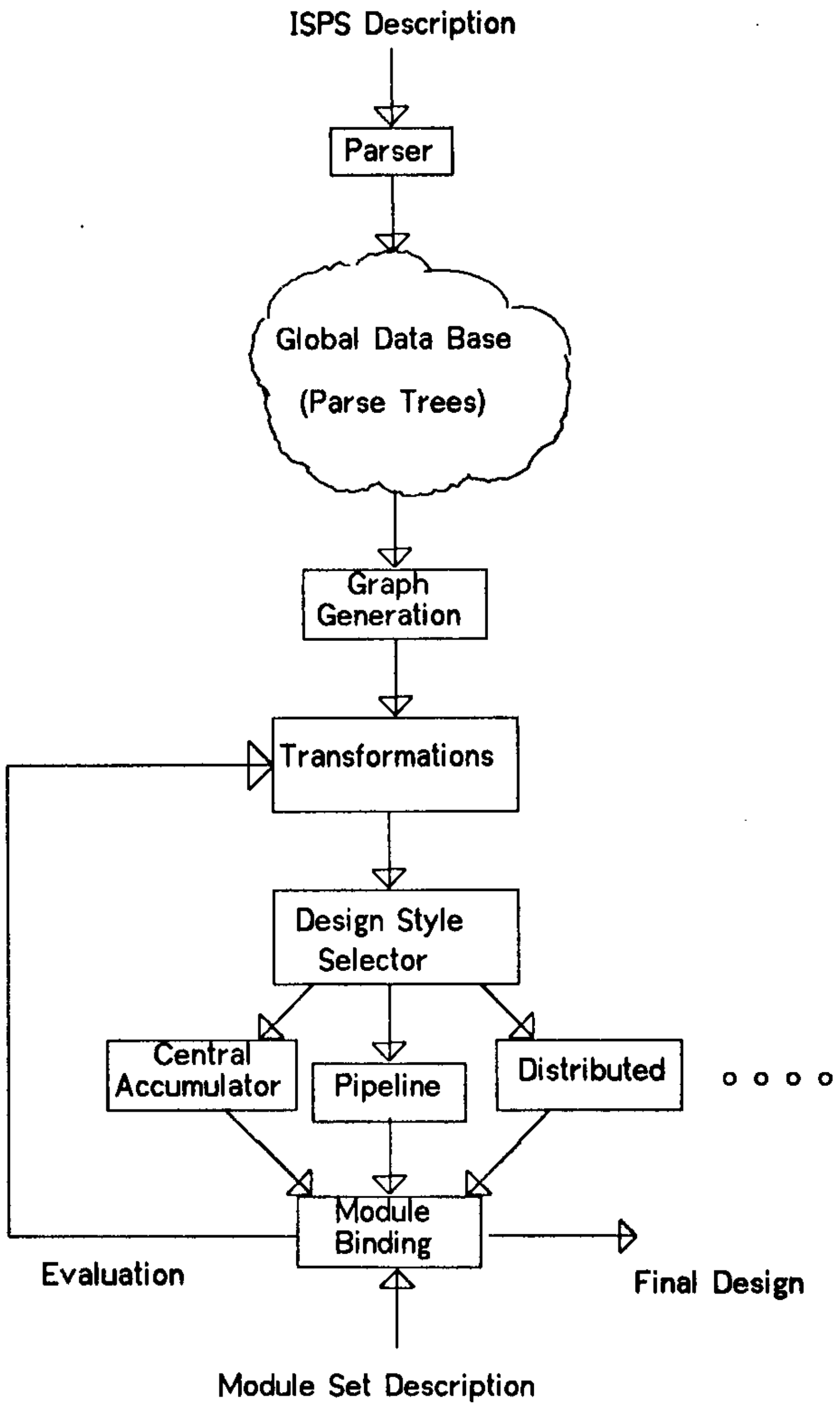


Figure 3-4: The CMU RT-CAD System

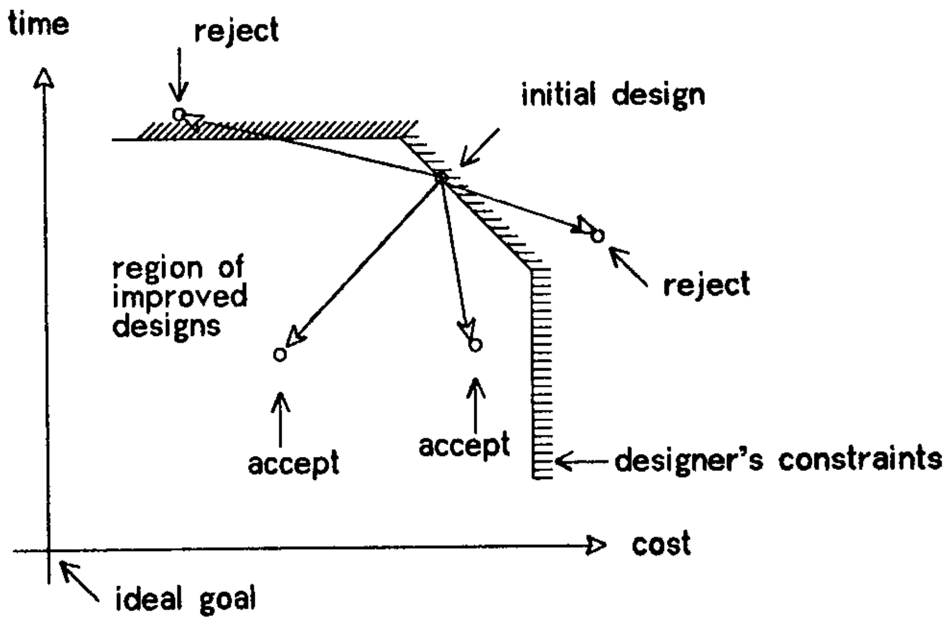


Figure 3-5: Cost-Speed Design Space

The evaluation of alternative solutions is heavily dependent on the particular technology used to implement the target system. Moreover, within a given technology, there are different ways of connecting components to achieve a certain target design. Organizational aspects are captured via a design style [Thomas, 1977]. A design style is a set of rules that guide the interconnection of the abstract components used in the behavioral graph representation.

Once a design style has been selected, a design style allocator (there is one for each design style: pipelined, central accumulator, distributed arithmetic, etc) is invoked to generate the path graph for the system [Hafer, 1978]. This is a layout of the registers, functional units, data paths, and their interconnections. It is still an abstract representation since no physical components have been selected and bound to the abstract components used in the path graph. The selection of the actual components is the task of the module binding phase [Leive, 1977].

The module binding phase uses the information gathered in the previous stages as well as information stored in the module data base to select the physical components and the order in which they are bound to the abstract components specified in the path graph. The output of the module binding phase can then be evaluated and the result reported back to the previous stages, thus closing the design loop.

Once a design is selected as the best of the alternatives that have been considered, the module binding phase generates the necessary information that is needed by the physical design system. This is the place where traditional design automation systems start i.e. with a logic diagram in which components and interconnections are completely specified. As described in [Parker, 1979], our system interfaces with the Sandia Laboratory's Design Automation System [Preas, 1977]. The complete system can then take an ISPS specification as input and generate the LSI masks.

The key parameter that permits the realization of such an exploration of alternative designs is the fact that the initial, behavioral description does not specify the actual implementation. By not specifying the actual components, the system is free to try alternative realizations, evaluate them, and select the best one according to the designer goals.

## 3.3 Software Generation

The proliferation of instruction sets brought about by the rapidly increasing rate of introduction of new computers demands an acceleration of the software generation process. Not only operating systems and utility programs must be developed but also compilers and assemblers to help the users take advantage of the newer, faster, and cheaper computers in the market.

Some of the items in the above list have been studied for a number of years and it is becoming a realistic goal to attempt to automate their development. In particular, assemblers and compilers are well understood and can be automated to some degree[Wick, 1975; Cattell, 1978; Leverett, 1979].

### 3.3.1 Assembler Generators

John Wick [Wick, 1975] has demonstrated the feasibility of using a formal description of an instruction set as one of the inputs to an automatic assembler generator. The system is capable of handling a wide variety of computers with substantially different architectures. The two man contributions of his work were the formalization of the assembly process and the technology for building assemblers in a manner independent of any particular computer. The second main contribution, and the one of more relevance to this paper was the formalization of a method for describing the target computer to the generator. Wick's notation (ISP') is an implementation of Bell and Newell's ISP, in which through the judicious

UNIVERSITY LIBRARIES
CARNEGIE-MELLON UNIVERSITY
PITTSBURGH, PENNSYLVANIA 15213

use of keywords and writing style guidelines, the characteristics of the machine that are relevant to the assembly process (e.g. operation codes and mnemonics, instruction formats, data types, etc) can be highlighted for the benefit of the assembler generator. Although Wick did not use ISPS, his approach bears a striking resemblance to the ISPS paradigm: application dependent information is added on to a formal machine description in such a way that different aspects of the machine can be selectively extracted, without affecting other applications or descriptions.

### 3.3.2 Compiler-Compilers

A more ambitious project, dubbed the Production Quality Compiler Compiler, or PQCC for short [Leverett, 1979], is an investigation of the code generation process. The practical goal of the project being the construction of truly automatic compiler-writing systems which produce compilers that are competitive with hand generated compilers in every respect. These are called Production Quality Compilers or PQC's for short.

To achieve this goal, the PQCC system must operate from descriptions of both the source language and the target computer, Figure 3-6. The PQCC project builds on work in two areas: code optimization, and compiler-writing systems:

1. "With some notable exceptions, previous work in compiler development tools has focused on the parsing and lexical analysis phases of compilation. Thus "compiler-compiler" has come to be almost synonymous with "parser generator". We would like to extend the function of compiler-compilers, to include the production of optimizers and code generators. We believe that compiler-compilers will become far more popular as commercial software development tools when this is done.

2. "A great many code optimization techniques are known and have appeared in the literature. Nevertheless, construction of optimizing compilers is still nearly a black art. As a result such compilers tend to be expensive both to build and to use. They also tend to be unreliable, in that the object code produced after optimization may be either worse than unoptimized code or altogether wrong. We would like to organize, even to formalize, the huge "bag of tricks" associated with code optimization. Ultimately optimization should be done as routinely, cheaply, and reliably as parsing" [Leverett, 1979]

Implied in the above goals is the reason for the failure of previous attempts to build useful compiler-compilers: In order to generate good code, the target machine instruction set is built into the system, thus preventing any reasonable degree of transportability. In those situations were transportability has been a required feature the solution has been to generate pseudo-machine code. For each new instruction set the pseudo-machine instructions are then translated via macros written in the target machine language [Feldman, 1966]. While

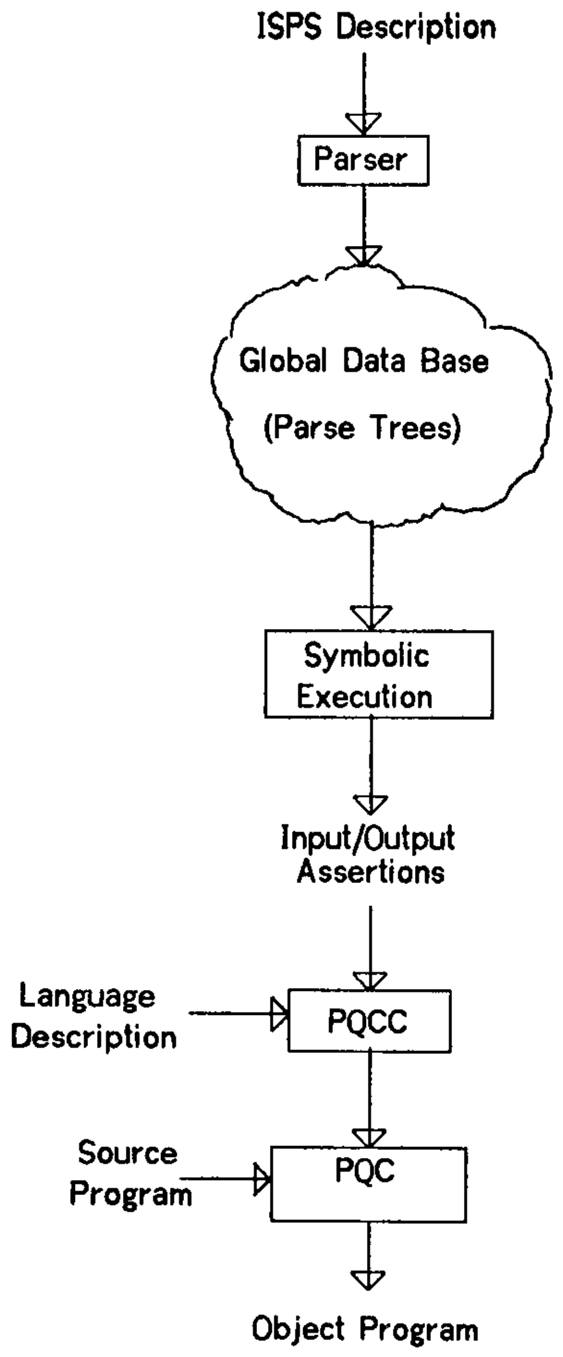


Figure 3-6: The Production Quality Compiler-Compiler

runable programs are produced by this technique, they are poor in terms of size and run time efficiency. There are several reasons behind this problem: built-in preconceptions about existing instructions, the introduction of an extra level of abstraction that must be hand translated, and the lack of consideration for specific machine features that can do certain things more efficiently than others.

In the PQCC project, machine independence is achieved by generating *machine relative* PQC's, that is, compilers whose code generation algorithms take advantage of the specific target machine language, the machine being specified as a set of input/output assertions about the format, cost, and side effects of each instruction.

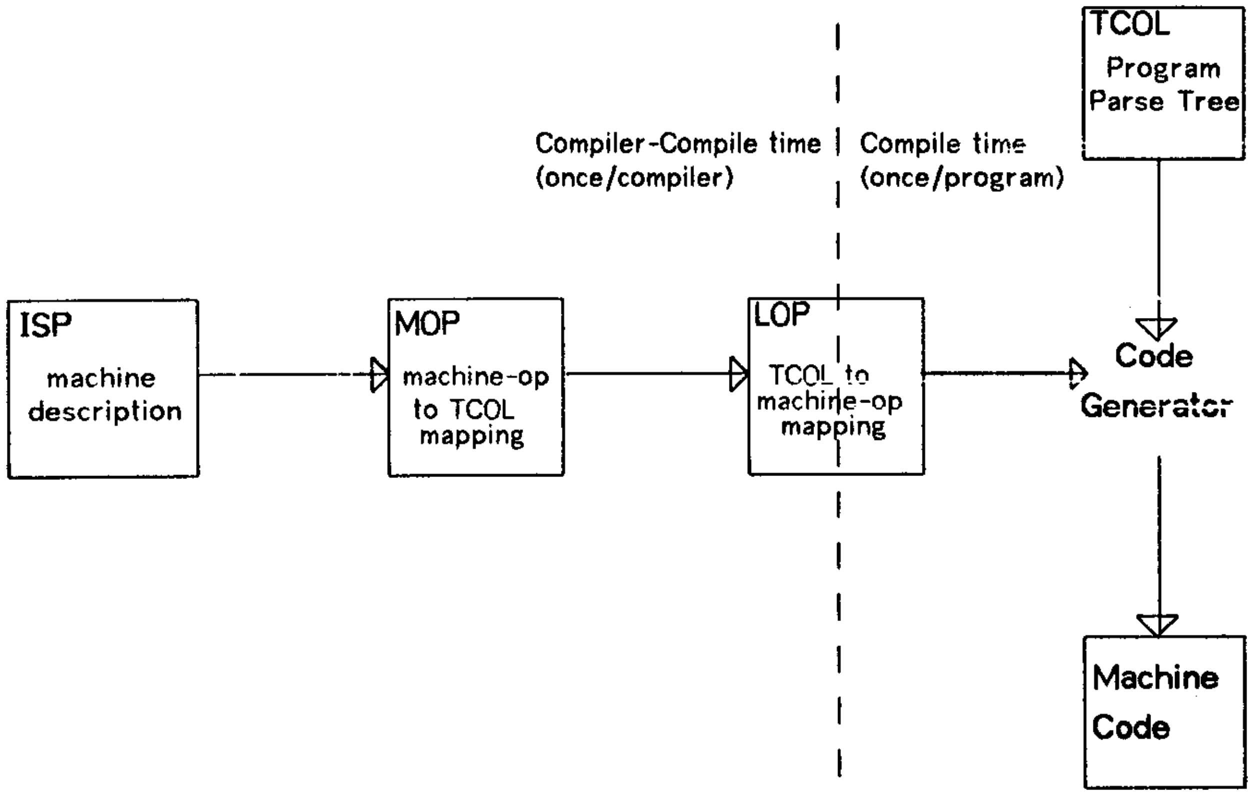

TCOL
Program Parse Tree
Compiler-Compile time (once/compiler)
Compile time (once/program)
ISP
machine description
MOP
machine-op to TCOL mapping
LOP
TCOL to machine-op mapping
Code Generator
Machine Code

Figure 3-7: Code Generator Generation (Figure 1 in [Cattell, 1978])

Code generation in a PQC is a two step procedure, as shown in Figure 3-7, [Cattell, 1978]. During compiler-compile time, the assertions about the machine instructions (Machine OPerations or MOPs for short) are generated. A language dependent subset of these is selected and used to drive the actual PQC code generation phase.

The code generation phase of a PQC, uses the assertions as a library of templates. Each template consists of a subtree pattern and a code sequence. Briefly, if a subtree of the program (parse) tree matches the pattern, the instructions in the code sequence are emitted, their operands being determined by the value of the pattern "parameters". Thus, each instruction is represented not by an algorithm to simulate it, but by a group of pre- and post-conditions (in terms of machine state i.e. the contents of various locations in the machine) of its execution.

### 3.3.3 Symbolic Execution

An assertion description expresses the semantics of an instruction more directly than a procedural description can. This has been shown not only in code generation but also in program verification [Crocker, 1977]. A procedural description on the other hand, is more desirable in other applications (not the least important of which being human readability). Thus, to provide a link between procedural and assertion descriptions is to provide the link between those applications which for whatever reason prefer one flavor of description over the other.

The feasibility of this link has been shown by John Oakley [Oakley, 1979] through the symbolic execution of ISPS descriptions.

"Conceptually, Symbolic Execution is just like normal program execution except that inputs to the program are symbols (representing unknown, but fixed values) rather than numbers. Expressions involving such symbols are called *symbolic values*. Variables on the left-hand side, called output variables, are set to symbolic values rather than calculated numeric quantities. Thes symbolic output really represent output assertions...

"When an ISPS conditional statement, an IF or DECODE, is executed symbolically, the predicate usually involves symbolic values. Since the truth value of the predicate will be determined by the numeric value eventually given to the symbolic inputs, it is not generally possible to determine which branch out of a conditional statement will be executed during symbolic execution. Consequently, each branch must be executed, one at a time, using backtracking to return to unexecuted branches. Associated with each branch is the symbolic path condition, taken from the IF or DECODE predicate, that must be true for this particular branch to be taken. This symbolic path condition is called an *input assertion* since it represents a constraint on the possible values that the input symbols can have if this branch is to be taken." [Oakley, 1979].

By taking the instruction cycle procedure as the root node, the tree of all possible execution paths throughout the procedural machine description can be traversed. Each execution path defines a functional instruction, completely described in terms of its input and output assertions[1]. These are the inputs required by the PQCC.

## 3.4 Reliability and Fault Tolerance

At the time of this writing (May, 1979) an entire new area of applications for ISPS descriptions is being developed at CMU. The simulation of digital circuits under faults has been used with great success in the development of fault tolerant digital systems [Avizienis, 1978]. We are currently trying to extend these techniques to a higher level of design, where the structural information is not available.

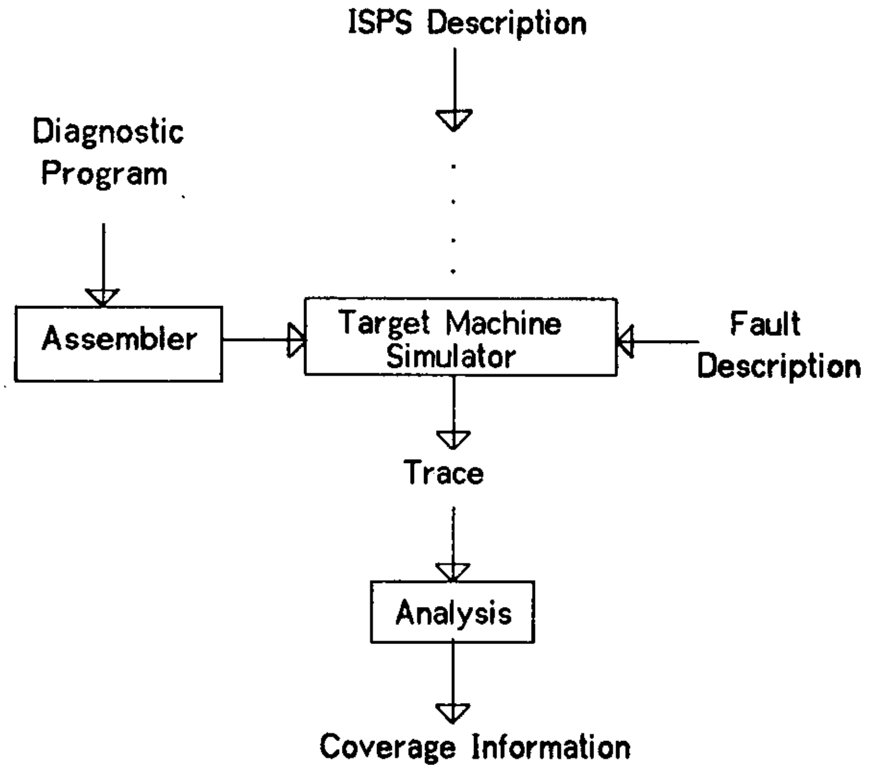


Figure 3-8: Functional Fault Simulation

[1]This is a very rough sketch of the process followed in Oakley's system. In particular, the distinction between a functional instruction and what most programmers understand as a machine instruction is a critical aspect of his work.

The ISPS simulator is being extended to allow the specification of a variety of data and control faults, of a permanent or transient nature, Figure 3-8. Thus, a user will be able to specify stuck-at faults and shorts in the registers, data paths, functional units, and control elements of the machine[1].

The usefulness of simulating and studying faults at a level removed from the physical machine becomes apparent when one considers the potential uses of the results. By injecting faults at the ISP level, we are in fact simulating the effect of a fault across <u>all</u> implementations.

To see how this is possible, consider the effect of the following fault description: "The sign bit of Register 5 is always 0". If we translate this fault (stuck-at-0) into its corresponding physical realization, we have to take into consideration many details that are peculiar to one implementation: board and pin position, electrical properties (what does 0 mean?), gate delays, etc. All of these attributes vary from implementation to implementation and little can be carried over to other models. At the ISP level the fault is clearly defined and its effect (e.g. make all data coming from register 5 positive) can be readily traced. By running test programs, the effect of faults in the instruction set processor can be observed at the programming level. If the test programs have diagnostic capabilities, the identification of the faulty function can be used to narrow down the identification of the physical fault. While this might not be detailed enough to allow the isolation of the faulty gate, will be detailed enough to isolate the group of components that implement the sign bit of register 5. Given the ever decreasing costs of hardware, this type of diagnostic ability might be enough to allow the replacement of the faulty board by a new one and this might be the extend of the repair. For certain applications, where repairs are not possible (e.g. aerospace computers), simulation of the fault tolerance characteristics of the programming level might be used to design systems in which the critical components (i.e. those implementing critical functions) are replicated. It also permits the design of software that adapts to the presence of faults (e.g. by replication or relocation, by changing the register allocation scheme, by using alternative algorithms, etc.).

[1]It should be clarified that we are referring to the *abstract* data paths, functional units, etc. since the actual physical characteristics are not specified in the ISPS description.

## 3.5 Conclusions

In this paper we present some applications in the area of automatic design of both hardware and software in which a computer description language serves as the vehicle for specifying the behavior of the target system. The goal is to achieve a unified environment for research on multiple application areas. We visualize an environment which will support multiple, concurrent users, investigating different aspects of the problem domain. The environment should permit the implementation of application programs in different programming languages which manipulate machine descriptions written in different computer description languages. So far, application programs to process GDB trees have been written in BLISS, LISP, SAIL, and SIMULA-67. The only machine description language in use is ISPS but there are no inherent limitations in the GDB format that would prevent translating other machine description languages into the same data base use by the application programs. Interestingly enough, these application programs can include translators from GDB trees into other machine description languages. This would permit us to use and expand on previous work based on other machine description languages.

## 4. References

[Amdahl, 1964] G.M. Amdahl, G.A. Blaauw, and F.P. Brooks, "Architecture of the IBM System/360", *IBM Journal of Research and Development*, April 2, 1964, pp. 87-101.

[Avizienis, 1978] A. Avizienis, "Fault Tolerance: The Survival Attribute of Digital Systems", Proceedings of the IEEE, Vol. 66, Number 10, October 1978, pp. 1109-1125.

[Barbacci, 1974a] M.R. Barbacci, "Automated Exploration of the Design Space for Register Transfer Systems", PhD Thesis, Department of Computer Science, Carnegie-Mellon University, 1974.

[Barbacci, 1974b] M.R. Barbacci and D.P. Siewiorek, "Some Aspects of the Symbolic Manipulation of Computer Descriptions", Technical Report, Department of Computer Science, Carnegie-Mellon University, 1974.

[Barbacci, 1976a] M.R. Barbacci, "The ISPL Compiler and Simulator User's Manual", Technical Report, Computer Science Department, Carnegie-Mellon University, 1976.

[Barbacci, 1976b] M.R. Barbacci, and D.P. Siewiorek, "Evaluation of the CFA Test Programs via Formal Computer Descriptions", IEEE Computer Society, COMPUTER, Vol. 10, No. 10, October 1977.

[Barbacci, 1977] M.R. Barbacci, G.E. Barnes, R.C. Cattell, and D.P. Siewiorek, "The ISPS Computer Description Language", Technical Report, Department of Computer Science, Carnegie-Mellon University, 1977.

[Barbacci, 1978] M.R. Barbacci and R.A. Parker, "Using Emulation to Verify Formal Machine Descriptions", COMPUTER, Vol. 11, No. 5, May 1978.

[Barbacci, 1979] M.R. Barbacci, W.D. Dietz, and L.J. Szewerenko, "Specification, Evaluation, and Validation of Computer Architectures Using Instruction Set Processor Descriptions", *ACM/IEEE 1979 International Symposium on Computer Hardware Description Languages and Their Applications*. October 1979, Palo Alto, California.

[Bell, 1971] C.G. Bell and A. Newell, Computer Structures: Readings and Examples, McGraw-Hill Book Company, New York, 1971.

[Bell, 1978] C.G. Bell, J.C. Mudge, and J.E. McNamara, Computer Engineering: A DEC View of Hardware Systems Design, Digital Press, 1978.

[Burr, 1977] W.E. Burr, A.H. Coleman, and W.R. Smith, "Overview of the Military Computer Family Architecture Selection", *Proceedings of the AFIPS National Computer Conference*, Volume 46, 1977.

[Cattell, 1978] R.G.G. Cattell, "Formalization and Automatic Derivation of Code Generators", PhD Thesis, Department of Computer Science, Carnegie-Mellon University, 1978.

[Crocker, 1977] S.D. Crocker, "State Deltas: A Formalism for Representing Segments of Computation", PhD Thesis, University of Southern California, Information

Sciences Institute, ISI/RR-77-61, 1977.

[Dietz, 1978] W. Dietz and L. Szewerenko, "Phase III - Comparative Evaluation of Candidate Computer Architectures", Technical Report, Department of Computer Science, Carnegie-Mellon University, 1978.

[Fuller, 1978] S.H. Fuller, G. Mathew, and L. Szewerenko, "Phase II - Comparative Evaluation of the MCF Computer Architectures", Technical Report, Department of Computer Science, Carnegie-Mellon University, 1978.

[Feldman, 1966] J. Feldman, "A Formal Semantics for Computer Languages and its Application in a Compiler-Compiler", CACM Vol. 9, No. 1, January 1966, pp. 3-9.

[Hafer, 1978] L. Hafer and A.C. Parker, "Register Transfer Level Automatic Digital Design: The Allocation Process", *Proceedings of the 15th Design Automation Conference*, Las Vegas, 1978.

[Lai, 1979] L. Lai, "Functional Testing of Digital Systems", PhD Thesis Proposal, Department of Computer Science, Carnegie-Mellon University, April 1979.

[Lavington, 1975] S.H. Lavington, A History of Manchester Computers, National Computing Centre Publications, Manchester, England, 1975.

[Leive, 1977] G.W. Leive, "The Binding of Modules to Abstract Digital Hardware Descriptions", PhD Thesis Proposal, Department of Electrical Engineering, Carnegie-Mellon University, 1977.

[Leverett, 1979] B.W. Leverett, *et al.*, "An Overview of the Production Quality Compiler-Compiler Project", Technical Report CMU-CS-79-105, Department of Computer Science, Carnegie-Mellon University, 1979.

[Oakley, 1979] J. Oakley, "The Symbolic Execution of Formal Machine Descriptions", PhD Thesis, Department of Computer Science, Carnegie-Mellon University, 1979.

[Parker, 1979] A.C. Parker *et al.*, "The CMU Design System: An Example of Automated Data Path Design", Proc. of the Sixteenth Design Automation Conference, San Diego, California, June 1979.

[Preas, 1977] B.T. Preas and C.W. Gwyn, "Architecture for Contemporary Computer Aids to Generate IC Mask Layouts", *Eleventh Annual Asilomar Conference on Circuits, Systems, and Computers*, November 1977.

[Siewiorek, 1976] D.P. Siewiorek and M.R. Barbacci, "The CMU RT-CAD System -- An Innovative Approach to Computer Aided Design", *Proceedings of the AFIPS National Computer Conference*, Volume 45, 1976.

[Thomas, 1977] D.E. Thomas, "The Design and Analysis of an Automated Design Style Selector", PhD Thesis, Department of Electrical Engineering, Carnegie-Mellon University, 1977.

[Wald, 1977] B. Wald and A. Salisbury (guest editors), "Military Computer Architectures: A Look at the Alternatives", COMPUTER, Vol. 10, No. 10, October 1977.

# I. Appendix: Invocation and Dynamic Control of Units

## 4.1 Invocation of Units

To illustrate how units of varying complexity can be described in the language, we will describe an arithmetic unit, its interface, and its use, Figure 4-1.

```
ALU(A<0:15>,B<0:15>,F\Function<3:0>)<0:15> :=
        BEGIN
        DECODE F =>
                BEGIN
                0\Add   := ALU = A + B,
                1\Sub   := Alu = A - B,
                2\Not.A := Alu = Not A,
                3\Not.B := Alu = Not B,
                4\Or    := Alu = A Or B,
                ..........
                14\Zero := Alu = 0,
                15\Ones := Alu = "FFFF
                END
END,
```

Figure 4-1: The Behavior of an Arithmetic Unit

The interface to the arithmetic unit consists of four carriers. Three of these carriers (A, B, and F) are used to receive and retain the input operands and the function code. The fourth carrier has, by convention, the same name as the functional unit (ALU)[1]. Pictorially, the arithmetic unit described above could be conceived to have the structure depicted in Figure 4-2.

The arithmetic unit can be used to specify the behavior of other entities. Thus, assume that somewhere else in the description we encounter a line of the form:

```
.... next X = Alu(Y,Z,5) next ....
```

Further assume that X, Y, and Z have been declared elsewhere (assume they are registers). The example indicates that the input carriers of the arithmetic unit are loaded with the

[1]For simplicity we are appealing to the intuition of the reader and we use terms such as "input carrier" or "output carrier". The language allows the specification of multiple input, output, or even bidirectional carriers, as we shall see in later examples.

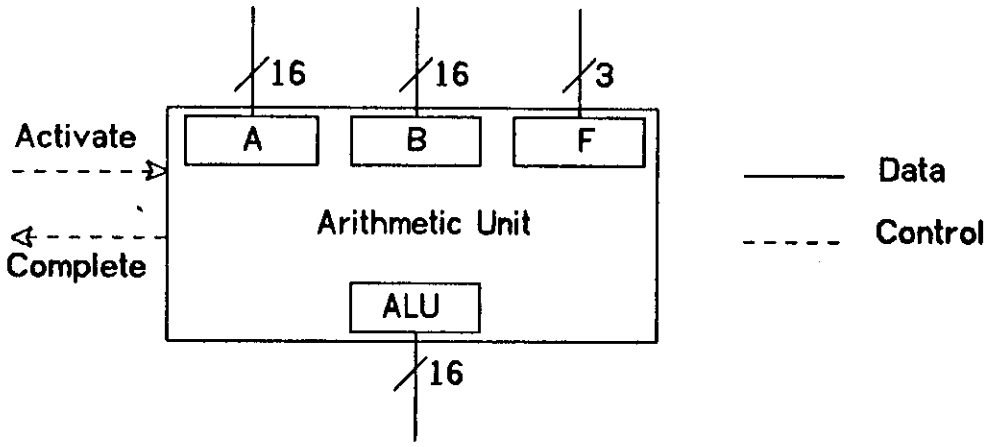


Figure 4-2: Arithmetic Unit Interface

contents of Y, Z, and the constant 5. The unit is then invoked and when its operation has been completed, the result appears in the output carrier (ALU<0:15>). The transfer operation can then proceed. Pictorially, the operation is depicted in Figure 4-3(a).

The output carrier from the arithmetic unit can be used directly, as in:

```
Zcc = Alu Eql 0
```

A condition code flag (Zcc) is set based on the result of the last operation performed by the arithmetic unit. The absence of an invocation (syntactically represented by a possibly empty, list of carrier names or expressions enclosed in parenthesis) indicates that this use of the arithmetic unit does not activate the behavioral part.

By default, the input carriers to the arithmetic unit (A, B, and F) can retain the values transmited during the invocation. These values are retained throughout the activation of the unit and are not affected by any changes in the values contained in the initial sources (Y and Z). Some arithmetic units are build this way, and the example clearly depicts the behavior. This is not however, the only way to build the unit. For instance, one could conceive a situation in which for cost or speed considerations, the arithmetic unit is not to have its own storage. Instead, it must operate directly on the original carriers used at the invocation site. This is easily described in ISPS as:

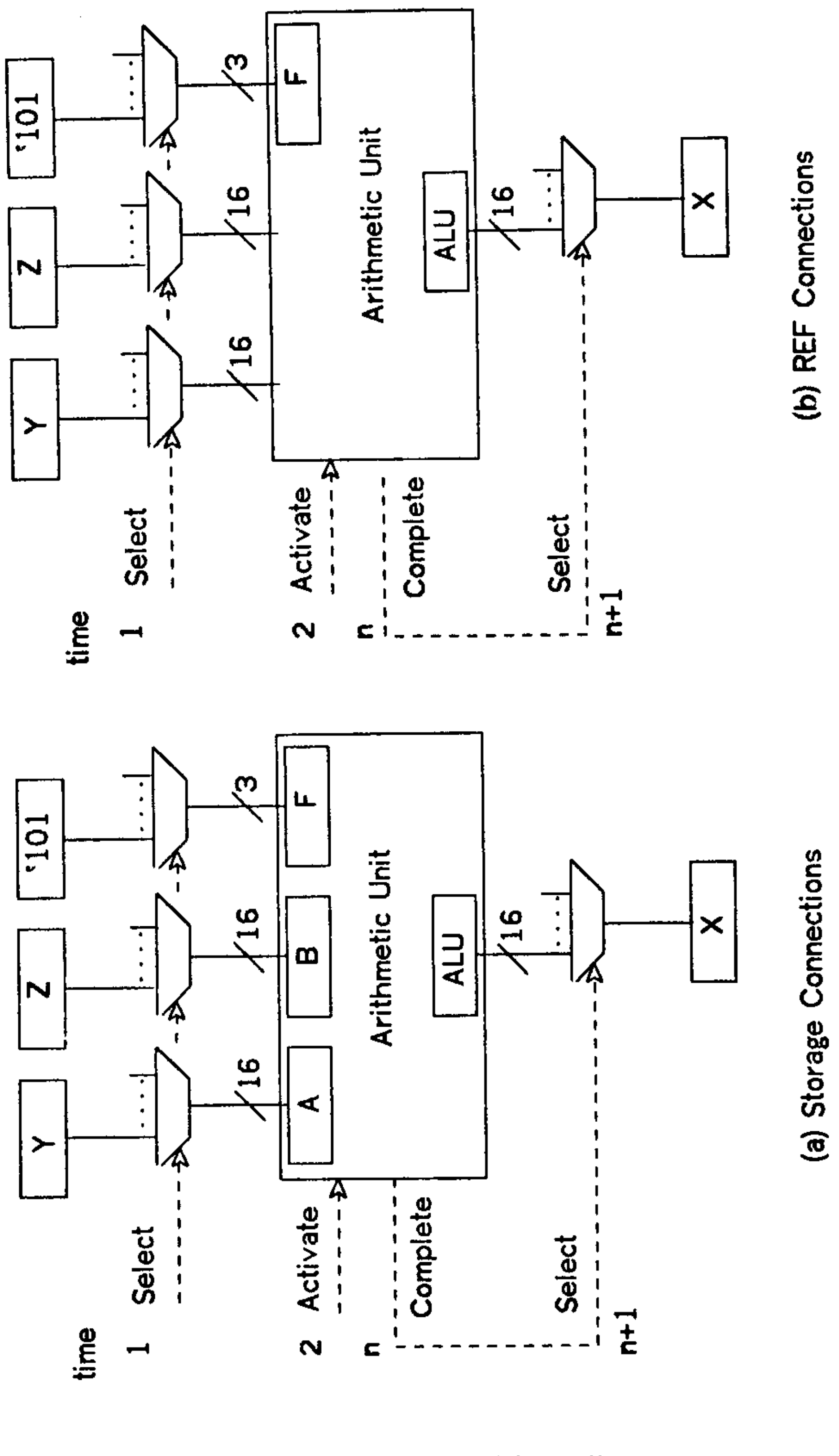


Figure 4-3: Arithmetic Unit Invocation

UNIVERSITY LIBRARIES
CARNEGIE-MELLON UNIVERSITY
PITTSBURGH, PENNSYLVANIA 15213

```
ALU(REF A<0:15>, REF B<0:15>, F\Function<3:0>)<0:15> :=
        BEGIN .......... END,
```

When the keyword REF (short for reference) is used to prefix the name of a carrier, the carrier is said not to have any retention properties and is simply a synonym for the actual carrier used in an invocation. When invoked, the arithmetic unit operates directly on Y and Z (F is still latched), Figure 4-3(b). If Y and Z are being used concurrently somewhere else, their value may change during the activation of the arithmetic unit and the result of the invocation might be unpredictable (*caveat emptor*).

The notation allows the specification of more complicated patterns of behavior. In the previous examples, the caller of the arithmetic unit does not receive the result until the unit has completed its operation. In some systems, speed is a critical factor and perhaps waiting for an operation to complete is not desirable. Let's assume that we have the situation where the arithmetic unit is too slow for the caller to wait for the completion of the operation. Instead, we want to be able to start the unit and proceed doing some other operations until some time later when the result will be available. This is easily described by modifying the description of the unit as follows:

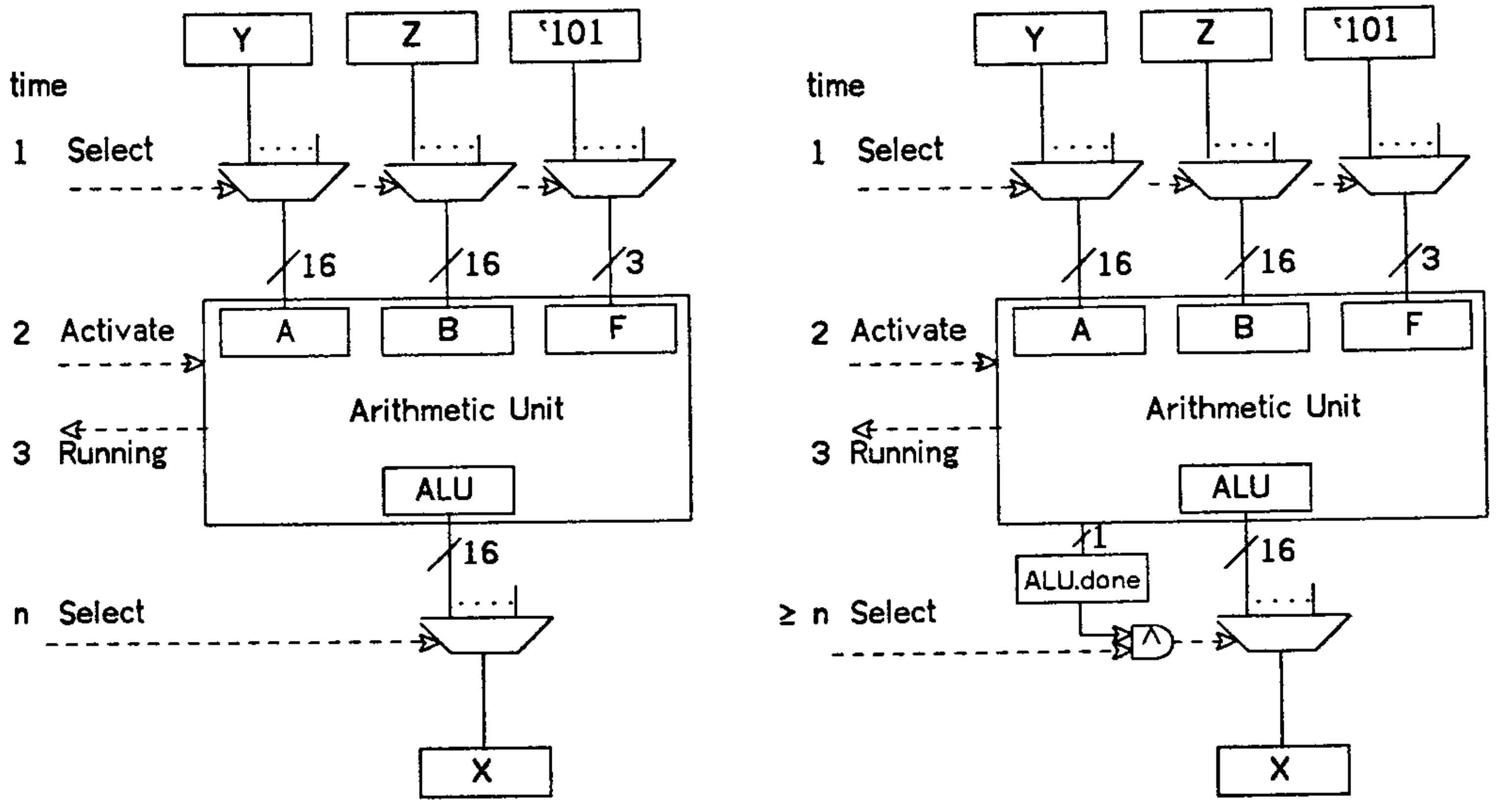
```
PROCESS ALU(A<0:15>, B<0:15>, F\Function<3:0>)<0:15> :=
        BEGIN ......... END
```

The keyword PROCESS indicates that the unit can operate independently of the caller, and moreover, that the caller does not have to wait for the completion of the activation, Figure 4-4(a)..

The new unit can now be used as in:

```
... next ALU(Y,Z,5) next .... next X = ALU; Zcc = ALU EQL 0
```

When a PROCESS-like unit is invoked, it is possible to execute other operations while the unit completes its activation, as depicted in Figure 4-4(a). Notice that it is not necessarily true that the unit has finished by the time its carrier is used (X = ALU). If the designer has been careful to allow for sufficient time to elapse before using the output of the unit, no problems arise. In situations when not enough work is available to overlap the operation of the unit, some other mechanism must be used to synchronize the operations of the system.



(a) Process Activation

(b) Process Activation with Explicit Completion Signal

Figure 4-4: Arithmetic Unit as a Process

Let's examine one method (by no means the only one): Assume the existence of a flag which is set whenever the arithmetic unit completes its operation, Figure 4-4(b). The invocation of the unit and the use of the result can now be described as follows:

```
... ALU(Y,Z,5) next
.......  next
Wait(alu.done) next
X = ALU; Zcc = ALU Eql 0 next
...
```

and the arithmetic unit is, of course, modified as follows:

```
PROCESS ALU(A<0:15>, B<0:15>, F<3:0>)<0:15> :=
        BEGIN
        ALU.done = 0 next
        DECODE F =>
                BEGIN
                ......
                END next
        ALU.done = 1
        END
```

When concurrent activations of units (whether they have the PROCESS attribute or not) can appear, it might not be advisable to start a new activation before a prior one is completed. To protect the activities of a unit, the keyword CRITICAL can be used as a prefix to the declaration of the unit, as in:

```
        CRITICAL ALU(A<0:15>, B<0:15>, F<3:0>)<0:15> :=
                BEGIN ...... END,
```

A CRITICAL entity is protected by an arbitration mechanism which delays concurrent activations until a prior activation is completed.

The different ways of specifying the arithmetic unit used in these examples suggests the parsimonious nature of the notation. A careful attempt has been made not to overburden the language with a multiplicity of syntactic constructs; similar concepts are expressed in similar syntax. Thus, regardless of the specific nature of the arithmetic unit, the invocation mechanism is the same [ALU(...)]. The unit declaration provides the rest of the information via its qualifiers (PROCESS, CRITICAL, REF, etc.)

The same characteristics appear in the specification of the control operations, specially in the RETURN-like operators described in the following section.

## 4.2 Dynamic Control

In programming languages, errors and exceptional conditions detected by a called procedure are handled by the caller. Typically, the procedure will set some global variable to a termination code which must then be examined by the caller. In a step wise refinement process, new procedures and (possibly) new error conditions can be introduced. It is also the case that one might want to rewrite a procedure, replacing a previous version. In either case, it is not a good practice to have to retrofit the unmodified code to accommodate for new error conditions.

ISPS, in addition to the standard programming-like return mechanisms, allows the called procedure to determine where control is to return, bypassing the caller. This is not achieved through a goto (there is none in the language), instead, ISPS provides a generalized return mechanism that allows the callee to specify the termination of multiple levels of invocation, up to a level where the error condition can be handled.

Three control operations, LEAVE, RESTART, and RESUME allow the writer of an ISPS description to model the modes of termination of an entity activation. Of the three, only the first one bears resemblance to a programming-like return operation.

The simplest use of LEAVE is to indicate a return or completion of an activation, Figure 4-5(a). The format of this operation is:

```
    LEAVE unitname
```

The identifier following the operator must be the name of a unit, not necessarily a lexicographically enclosing one: The only condition that must be met is that the operation must occur inside the activation of the unit named in the operation[1], Figure 4-5(b). The LEAVE operation terminates the activation as it had been completed. In addition, any activations initiated during the execution of the activation to be terminated and not yet completed are also terminated.

The example in Figure 4-6 shows the use of LEAVE in a dynamic context.

[1] inside means both static and dynamic nesting. The former is defined by the lexicographical nesting of the entities. The latter is defined by the nesting of entity activations.

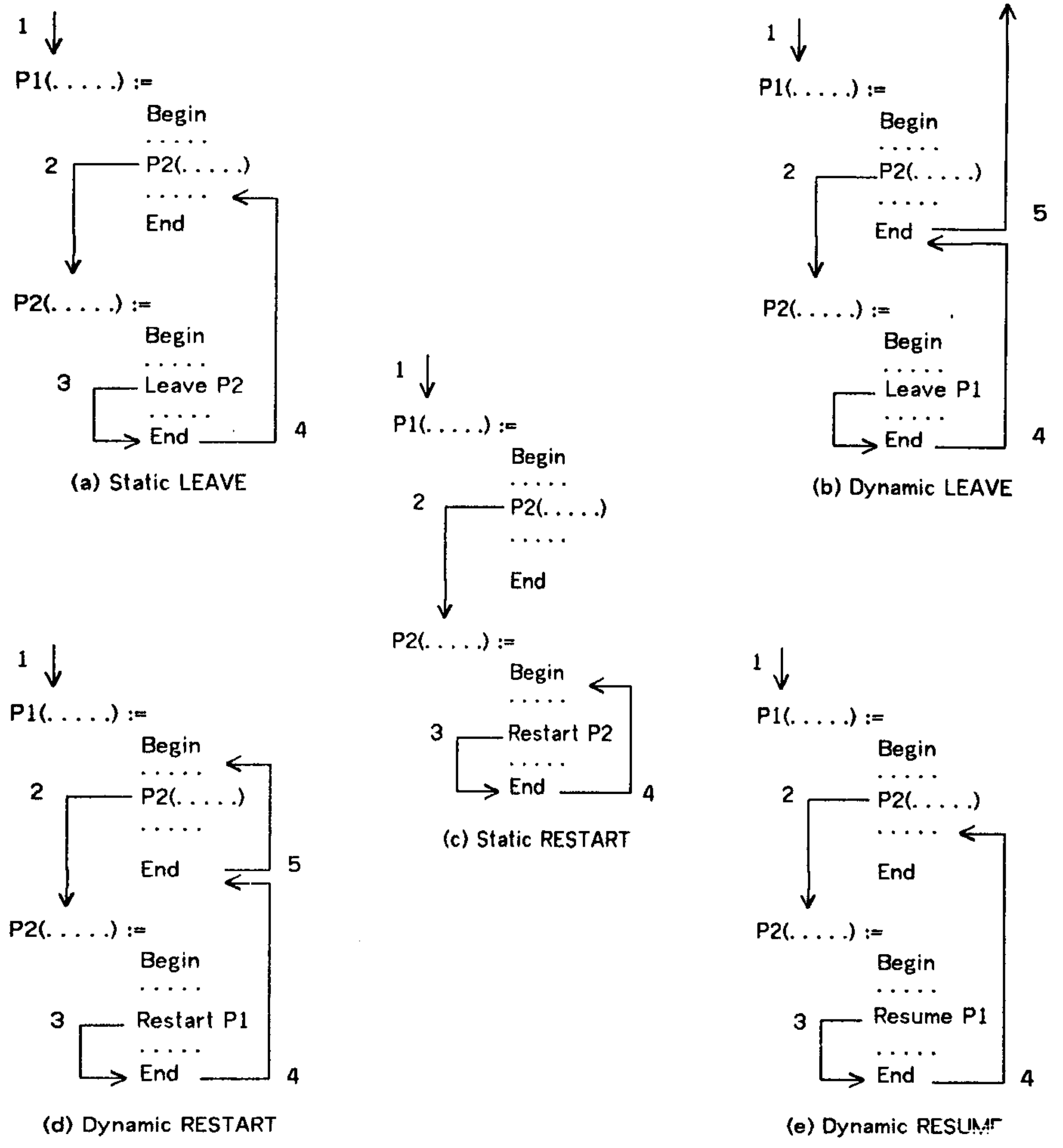


Figure 4-5: Static and Dynamic Control

```
Interpreter :=
        Begin
        .... next
        Icycle() next                                  Invoke Instruction Interpretation
        If  Error  Eql  1  =>  Begin  ....  End  next              Error Handler
        .....
        End,

Icycle :=
        Begin
        PC = PC + 2 next                                  Increment Program Counter
        IR = Rword(PC) next                                       Instruction Fetch
        Decode IR<0:3> =>                                     Instruction Decoding
                Begin
                .....
                ACC = ACC + Rword(IR<4:15>)                              Data Fetch
                .....
                End,

Rword(Addr<0:11>)<0:15> :=
        Begin
        If  Addr  Gtr  Upper.Bound  =>                       Address Boundary Check
                (Error = 1 next Leave Icycle) next                            Abort
        Rword = MP[Addr]                                                Memory Read
        End,
```

Figure 4-6: Example of Dynamic Leave Operation

In the example, INTERPRETER activates ICYCLE which fetches, decodes, and executes the instructions. In doing so, ICYCLE activates RWORD which is used to access the memory (MP) of the machine. RWORD checks that the memory address is in bounds before performing the access operation. If a boundary error is detected, a flag (ERROR) is set and the rest of the operation of ICYCLE is aborted (RWORD returns directly to INTERPRETER, at the point where it activated ICYCLE). Notice that we could have let ICYCLE handle the error by terminating RWORD with 'LEAVE RWORD'. However, this would have meant that the ICYCLE procedure had to check the error flag (ERROR) after every call to RWORD. Depending on the size or complexity of the description, this might be undesirable.

The other two ISPS control operators, RESUME and RESTART have a format similar to that of the LEAVE operator:

```
RESUME unitname
RESTART unitname
```

The RESUME operator specifies the name of a unit to which control is to return. The RESTART operator terminates and re-activates the unit named in the operation. Figures 4-5(c), 4-5(d), and 4-5(e) illustrate the general case of these operators. It is a matter of style and of course, of the problem on hand, to decide which mechanism is more descriptive or appropriate.

The conditions for aborting multiple levels of activation might not be errors or unpredicted situations at all, as the previous example implies; they might very well be part of the normal operation of the machine. The example in Figure 4-7 illustrates this situation via the RESTART operator.

INTERPRET depicts the instruction interpretation cycle of the DEC PDP-8. The normal sequence of operations is to fetch an instruction from memory, increment the program counter (PC), and decode/execute the instruction. After each instruction has been completed, the processor tests for the presence of pending interrupts. If interrupts are enabled and pending, the processor saves the program counter in location 0 and starts executing instructions at location 1 (the interrupt handling routine).

INPUT.OUTPUT is invoked to perform input and output operations and also to control the interrupt mechanism. Two I/O operations are of particular interest. IOF is used to turn off the INTERRUPT.ENABLED flag, thus preventing the processor from trapping on future interrupt requests. This operation would typically be the first instruction of the interrupt handling routine. ION enables interrupts. It typically occurs at the end of the interrupt handling routine. However, its effect must be delayed for one instruction, to allow the processor to execute one more instruction (the return from the interrupt handler). If interrupts were to be allowed immediately after the ION operation, any pending interrupts would cause the processor to save the program counter (which is pointing to the instruction following the ION, the return instruction) into location 0, thus destroying its initial value (the user's program counter). The delay is achieved by aborting the rest of the instruction cycle (the test for pending interrupts) and RESTARTing the cycle from the beginning. This will allow the return instruction to be executed. Pending interrupts can then be trapped in the user's context, and can be serviced in the normal manner.

```
interpret :=                                     PDP-8 Instruction Interpretation Cycle
        Begin
        Repeat Begin
                IR = M[PC] Next                                      Instruction Fetch
                PC = PC + 1 Next                            Increment Program Counter
                Decode op =>                                       Decode and Execute
                        Begin
                        ....                                     opcodes 0 through 5
                        #6\iot := input.output(),
                        ....                                                opcode 7
                        End Next
                If interrupt.enabled And interrupt.request =>             interrupts?
                        Begin
                        M[0] = PC Next                          Save Program Counter
                        PC = 1                                    Branch to location 1
                        End
                End
        End,

input.output :=
        Begin
        Decode IR<3:11> =>
                Begin
                #001\ion :=
                        Begin                                       Turn Interrupt ON
                        interrupt.enabled = 1 Next                  Enable Interrupts
                        Restart interpret                      Delay Interrupt Checks
                        End,
                #002\iof :=
                        Begin                                      Turn Interrupt OFF
                        interrupt.enabled = 0
                        End,
                ....                                             other i/o operations
                End
        End,
```

Figure 4-7: PDP-8 Interrupt Control

UNIVERSITY LIBRARIES
CARNEGIE-MELLON UNIVERSITY
PITTSBURGH, PENNSYLVANIA 15213