(9)

ISP: A LANGUAGE TO DESCRIBE INSTRUCTION SETS AND OTHER REGISTER TRANSFER SYSTEMS

M. Barbacci, C.G. Bell, and A. Newell
Department of Computer Science
Carnegie-Mellon University

ABSTRACT

This paper describes the evolution of a notation, ISP (Instruction Set Processor), which was originally developed for defining the instruction set, data-types and operations and the interpreter of a computer, giving essentially the same information as in a programming manual. ISP has been used in a book (Bell and Newell, 1971), in programming manuals, and papers to describe many computers. As part of the evolution of the language, much consideration has been given to the readability and simplicity of the notation as a descriptive tool, as well as to some other properties such as extensibility and fidelity, required by the notation as a design tool. ISP has also been extended (evolved) to handle Register Transfer (RT) systems for description, simulation and design purposes, including a flow chart form used in the Register Transfer Module System (Bell, Grason and Newell, in press). For RT design it has been necessary to incorporate additional facilities to describe the switching circuits (i.e. combinational and sequential components).

This research was supported by the Advanced Research Projects Agency of the Department of Defense, under contract no. F44620-70-C-007 and monitored by Air Force Office of Scientific Research. We would like to thank Richard Johnsson for his assistance in editing this paper.

INTRODUCTION

Five major levels of a hierarchy in digital systems can be recognized, for which we are interested in formally defining using notations and/or computer recognized languages. They are:

PMS level (System level).- The top level of description, evaluates the gross properties of the computer system. Its elements are processors, memories, switches, peripheral units etc, and the parameters are costs, memory capacities, information flow rates, power etc.

Programming level.- The basic components are the machine instructions, operations and the interpretation cycle (which are defined at the RT level). The behavior of the processor is determined by the nature and sequence of its operations. This sequence is given by a set of bits in primary memory (a program) and a set of interpretation rules. Thus, if we specify the nature of the operations and the rules of interpretation, the actual behavior of the processor depends solely on the initial conditions and the particular program.

Register Transfer level (Functional level).- Data flow and control operate in discrete steps. A combination of switching circuits is used to form registers, register-transfer and other data operations. The elements (registers) are combined (transformed) according to some rule and then stored (transfered) in another register. The rules of transformation can be almost anything, from simple transfers to complex logical and arithmetic expressions.

Switching circuit level (Sequential and **Combinational sublevels).-** The system structure is given by a collection of gates and flip-flops, and the behavior by a set of boolean equations. Timing is carried out at a finer degree than at the preceding level, a time unit being usually in the order of a gate delay.

Circuit level.- Gates are described as some interconnection of diodes, transistors, resistors, etc. according to electrical circuits laws. Most of the discrete properties of the previous two levels are lost, and timing is carried out at a finer degree, where transient behavior is usually a very important consideration.

The **PMS** and Circuit levels, by their continuous treatment of time, belong to a different category of design than the other intermediate levels, and will not be dealt with here.

For each of our levels of interest, there is a certain need to be formal, for communications purposes, and to have a representation that is convenient to use in the design process, so that concepts can be stated easily, partially completed designs can be studied, and analysis can take place. Finally, there should be a convenient, unambiguous mapping of the notation into physical objects. The mapping is not part of the notation, but of the particular description, thus allowing many possible mappings.

The ISP notation was developed to do a descriptive task: To be able to formalize the information normally given in basic machine manuals and if possible to supplement and eventually replace what are known as "programming reference manuals"; for instance, the DEC PDP-11/45 Processor Handbook (DEC, 1971), carries a description of the processor in ISP. Hence the essential requirements were of readability, completeness, flexibility and brevity. In evolving the language, the intent was to present a notation which could represent almost any design at the Programming, RT and Switching levels, in a structural and formal way, so that it could be easily converted to hardware or to a microprogrammed interpreter. Other requirements were that the notation should also be a design tool, not merely a descriptive tool. Doing this we expect to produce programs which can analyze designs, simulate them, and in some cases carry out limited design activities (e.g. ISP description of the data and control parts of a particular RT scheme).

Relationship to other work.- There has been considerable related work for our three levels of interest (below the PMS level); in particular, APDL (Darringer, 1969) can be considered a predecessor of ISP. At the Programming level, the monumental description of the IBM/360 in APL (Falkoff, Iverson and Sussenguth, 1964) is the most impressive. In Europe, several versions of Algol, including Algol 68, have evolved and been used to describe several computers. There have been programs written in Algol, APL, Fortran and various machine Assembly languages to simulate a given computer. As a by-product, these descriptions define the machine in a fundamentally RT language.

At the RT level, there have been numerous languages, although the first informal use in a text appears to be that of Bartee, Reed and Lebow (1962). Schorr (1962) describes an early language for this purpose. Chu wrote about CDL in 1964. He has written two books about this (Chu, 1970, 1972) and a simulation program exists. Again, when a particular RT structure such as a stored program computer or a microprogrammed computer is designed, it is essentially trivial to convert its flowchart definition into a conventional RT-like programming language. Thus there is a significant lack of need to produce a definitive RT language.

At the switching circuit level, simulators have been used for some time, both because they are needed (it is too costly to build LSI chips and experiment with them) and because the alternative of writing a realistic simulation in a conventional programming language is difficult. Most universities have their own logic design simulators for clocked sequential, delay-less circuits. For this later type, however, a standard programming language is adequate.

LANGUAGE REQUIREMENTS

What properties are desired in a language for writing behavioral and structural descriptions of a digital system?

RT languages are similar to most programming languages since they both carry out register assignments. The parallel nature of hardware suggests that it could be useful to at least have a special notation, even though programming languages such as FORTRAN can be used for this purpose.

We can divide the set of properties in two classes: one of them consists of the requirements for a scientific notation used in a design process; and the other has to deal directly with the objects we are devising, namely digital systems, particularly computers.

GENERAL PROPERTIES

A) **Readability.**- The notation is going to be used as a conveyor of information, not only between man and machines but among humans, who do not all have the same experience or involvement in the design. A description in this notation should be precise, conscise and elegant (considerations of typography, character sets, formats, the way operations such as array accesses are described, etc.), it should be usable as the ultimate source of information about the object. Information should be extracted from the context rather than by syntax which clouds the description (e.g. register declarations, keywords etc.).

B) **Familiarity.**- Primitive concepts in the language should be named and used in a way consistent with general practice.

C) **Generality.**-The notation should describe the elements occurring in the universe of interest, and at several levels of detail (in a hierarchical way), by supressing repetitive or unnecessary detail.

D) **Simplicity.**- There should be few primitive concepts, and they should be used consistently throughout the description, avoiding special cases of more general concepts, or things that are of relative importance or that imply a specific implementation.

E) **Unambiguity.**- The interpretation of a description should be unique.

SPECIFIC PROPERTIES

A) **Extensibility.**- The language should be able to extend gracefully by defining constructs in terms of elements already in the language. This also gives the capability to describe machines in a hierarchical fashion.

B) **Fidelity.**- The organization of the description should reflect the organization of the machine, making the intentions of the designers transparent to the users, humans or simulation/production automation processes. Thus, at

the lowest level it must be able to describe physical elements (e.g. gates, flip-flops, delay lines...).

C) **Timing and Concurrency.**- Machines are essentially parallel and this implies that concurrency should be the rule rather than the exception.

D) **Syntactically simple** (writable).- The notation is a tool for designers and descriptions should be written by them and not (necessarily) by programmers.

E) **Hardware independence.**- The notation should be relatively independent of any hardware technology, machine organization, timing mode, design procedure or simulation/ production techniques. This is somewhat in conflict with the fidelity property.

F) **Separability.**- The notation should be able to express the dichotomy between data and control. It should express the structure and behavior of the data flow, which implies the behavior of the control part. Also separability should permit the function of a primitive (e.g. an AND gate) to be described in an independent fashion.

HARDWARE REQUIREMENTS

The three intermediate hardware levels: Programming, Register Transfer and Switching can all be described using a single language. In fact, conventional programming languages have been used. The issue is, however, how much they must be changed to reflect parallelism, timing and the structure of the object being represented.

The RT and Switching circuit levels can be considered by treating them as two different design problems. Actually, the RT level is a generalization of the Switching level, the structural elements are arrays of identical subsystems belonging to that level, i.e., registers are made of flip-flops and gates, driven by clocks (or clock-like signals). The behavior is described by transformations (functions) and transfers between registers. The key element that sets this level appart is the appearance of control (the ability to perform these transformations and transfers in a selective way) as an explicit entity. Later, a (conceptually) graceful way of treating both levels in a continuous fashion will be given. The reason for this approach is that, although the RT level is a perfectly valid level of design, it has not been fully defined and understood (it is not a problem of youthfulness, the level was recognized as such in the early 50's). As a consequence we do not have a proven and accepted (complete?) set of primitive elements. Also, there is no accepted design form/style, and usually informal flow-chart and data path diagrams are used here. The result is that a designer, working at the RT level must ocasionally descend one level (to the gate level) to describe precisely a particular piece of hardware. Contrast this with the gate level of design, with the traditionally accepted set of primitives (AND, OR, FLIP-FLOP etc.) where it is indeed rare for a designer to have to describe a gate in terms of diodes or transistors.

DATA OPERATORS, DATA TYPES and CARRIERS

The Switching and RT levels will be related by providing a common notation and a few primitive entities: The first element is the **data-operation** which produces bit patterns with new meaning. They do the processing by transforming information. **Data-operations** work on **data-types**, composed of a value (meaning) and a representation (encoding). Associated with each **data-type** there is a **carrier**, used in storing and transmitting the **data-types**.

DATA-OPERATIONS.- Create information (instances of **data-types**) with new meaning, in which process it may destroy some existing information. The **data-operation** takes its input (the **data-type carriers**) operates on the data and presents the output (the resulting **data-type carriers**). The complexity of a **data-operation** ranges from a transfer path between **carriers**, a combinational network, to more complex arithmetic expressions, including sequences of simpler **data-operations**, e.g.:

NAME	SYMBOL	EXAMPLE
concatenation		A B
extraction	<...>	A<1>
transfer	←	A ← ...
add	+	N + M
and	∧	A ∧ B

DATA-TYPES.- Encode a value into an information medium. This encoding ranges in complexity from a single bit to a highly complex entity (a floating point number, with one or two signs, exponent and mantissa fields, posibly using different arithmetic representations such as excess-n, sign-magnitude, ones or two's-complement etc.), e.g., BIT, BYTE (usually 8 bits), CHARACTER (6, 7 or 8 bits), DIGIT, FLOATING POINT, COMPLEX, WORD.

CARRIERS.- Are hierarchically organized information structures, in which each level consists of a number of **sub-carriers** all identically organized. This decomposition eventually yields elementary **carriers** that can not be decomposed further (e.g. a bit **carrier**). Almost all information in computers is organized in these terms, for instance, a memory consists of a number of words, each of a number of characters, each of a number of bits, e.g., BIT, BYTE, HALF WORD, WORD, DOUBLE WORD.

The number of subcarriers at any level of decomposition is given by bracketed lists of names (if specific names are associated with the sub-carriers) or constants (positive or negative numbers), much like array declarations in FORTRAN or ALGOL. The **range** operator (":") is used to denote abbreviated list of elements, e.g., A<S;P;15:0> is a 18 bit register.

There are two types of **carriers** in ISP, the difference being the relative latency of the value associated with the **carrier**:

1) The first type corresponds to memory elements (flip-flops, registers, memories) and are declared by giving the name of the **carrier** and a description of the structure, e.g.:

M[0:255]<0:11> A memory (carrier) with 256 words (subcomponents), each of 12 bits.

ACC<S;0:11> A 13 bit register, the first one is named "S", for purely mnemonic reasons. The name is "local" to the register ACC and stands for itself (it is not to be evaluated like an expression).

2) The second type of carrier (memoryless) is defined to be the output of combinational circuits. They are declared using the **define** operator (":="), its action being that every time the carrier (the left hand side) is used, the defining expression (the right hand side) is evaluated. We can think of them as wires, the **data-types** being signals propagating along them, e.g.:

C := A $\wedge$ B an AND gate with inputs A and B and output (wire) C, or C := AND (A,B). Constant signals can be defined likewise: D := 1, as can more complex (memoryless) carriers:

X<0:11> := Y<0:11> $\wedge$ $\neg$(W<5:11> W<0:4> $\vee$ Z<0:11>) a network with inputs W, Y and Z and output X (12 bits).

Carriers do not necessarily have bits as the most elementary components; in fact a **carrier** can be denoted as a structure of elements each of which can assume values out of some arbitrary alphabet (the alphabet for bits being "0" and "1"). This is denoted by appending to the carrier declaration a **base** operator ("$\downarrow$") and a "size" operand, e.g., A<0:3> $\downarrow$ 16, A is a register of 4 elements, each one can assume as value a hexadecimal digit (they are not 16 bits !).

DELAY AS A DATA OPERATION

One of the major limitations in practical logical circuits is the presence of "inertia" in the response of the circuit, a time during which the occurrence of a transition between values of a signal is held. Delays are also introduced in links due to the finite signal propagation speed. In many applications it is important to merge the occurrence of events (signals) arriving through different (delaywise that is) paths, between specific time limits.

These different instances of delays can be described in a uniform way by treating them explicitly rather than by burying the delay in the internal constitution of a gate or introducing "busy-waits" in delay units or links. An element, then, can be described by two components, the operative one being an ideal version of the element and an inertial component which describes and accounts for the delay of the real element. This has the advantage that the level of detail or realism in the description is left up to the user. Idealized elements can be used or the delay can be treated at a more global level. For instance, the delay of an arithmetic unit which performs an addition can be described without mentioning individual gate delays. Another advantage in this treatment is that there is a consistent and formal way to express certain peculiarities of the elements such as response times, rising and falling signals, inertial responses (in which the switching signal that triggers the event is required to be stable for some minimum amount of time), etc.

The way delays are treated will differ from most simulation techniques in use. In most cases, the simulation of delays results in the "scheduling" of an event at some future time (the event being the ideal version of the operation). This violates a conceptual understanding of the time lag, in that what is actually wanted is the occurrence of the event at the present time, based upon past history. The current scheme has the advantage that now ideal and real (practical) elements can be mixed in a description without doing violence to the notation. A special function with two parameters, **PREVIOUS**, is introduced: PREVIOUS(t,I) will deliver as result the value of the variable (signal) "I" , "t" units of time ago, e.g.:

C := A ∧ B "ideal" (delayless) AND gate.

C := **PREVIOUS**(td,A) ∧ **PREVIOUS**(td,B) AND gate with constant delay of td units.

X ← **PREVIOUS**(t,Y)

OUT := (IN ∧ **PREVIOUS**(tr,IN)) ∨ (¬IN ∧ **PREVIOUS**(tf,IN)) an identity gate (amplifier?) with different response times to rising and falling signals, the signal being "IN".

CONTROL-OPERATORS INDUCED BY THE RT LEVEL

A **control-operation** is a circuit that evokes operations in other components. There are two kinds: **evoke** and **evoke-next**. The first one includes the testing of conditions under which the evoking occurs. These conditions can vary in complexity from a single bit test, to a boolean operation, to arithmetic and relational expressions.

In the **evoke** type, the control operation has the following format: **condition** => **action-sequence**, where the **condition** is a boolean expression to be tested and the **action-sequence** describes which operations take place. There are two important features in the action-sequence. The first is that each action in the sequence may itself be conditional (i.e. evocations can be nested). The second is that some actions are sequentially dependent on each other, because the result of one is used as an input to the other; on other ocassions, a set of actions are independent, and can occur in parallel (this is the normal situation, and therefore the default assumption).

A set of parallel actions is given by a list of actions, using the ";" as separator. e.g.: A ← B ; B ← A will exchange the contents of registers A and B.

The sequential case is denoted by the use of the **evoke-next** type of control. When sequencing is required, the term **NEXT** is used as a delimiter. e.g.: A ← B ; **NEXT** B ← A transfers the contents of register B to A and then transfers it back to B (equivalent to: A ← B).

PRIMITIVES INDUCED BY THE PROGRAMMING LEVEL

(The example at the end of the paper may be useful to complement the following definitions).

Instructions are special cases of **data-types**, with a fixed part of the format being the operation-code and the rest being interpreted depending on the instruction. The effect of each instruction is described by an **instruction-expression**, similar to the **evoke** control operation: **condition** => **action-sequence**, where the condition is usually of the form: op-code = number, and the action-sequence describes what transformations of data take place between what memories. These take the transfer form, i.e.: **memory-expression** ← **data-expression**. The **data-expression** describes the transformation of information (if any) and the information pattern that is to be placed in the memory described by the memory expression.

Memory-expressions specify the contents of a memory (a **carrier**) according to some "address". An address is a **data type**, which is usually an integer, although, sometimes, there are special data-operations that work only on addresses (i.e. address-expressions) to compute addresses.

The carrier structure is given in the declaration by bracketed lists of dimensions. Square brackets ("[" and "]") specify those dimensions where the accessing is done through some "addressing" scheme (switching). Angle brackets ("<" and ">") are required in the declaration and are used to describe the carrier's substructure.

The carrier is accessed through its identifier followed by a description of the structure of the "result". This structure is similar to that used in the declaration insofar as number of dimensions is concerned, although angle brackets are not required when it is understood (i.e. from the declaration) what is the number of bits involved. The difference appears, in the format and number of elements inside the bracketed lists. These list elements or **selectors** can be not only names or constants but also expressions or bound pairs of expressions (address-expressions, using the **range** operator). Furthermore, bound pairs used as selectors do not have any relationship with bound pairs used in the declaration; they imply an abbreviated list of element names and can in fact cross bound pairs defined in the declaration, can reverse the ordering of names (implying a reversing of the relative ordering of the elements) and may even be used when no bound pairs were used in the declaration.

Interpreters are control entities that select and execute (interpret) instructions, taking the system through sequences of steps which change the state of the system, sometimes permanently, setting the initial conditions for the next instruction. Interpreters are described by **action-sequences** that include **control** and **data-operations**.

Parenthesis are used to group actions allowing nesting of action sequences; parenthesized action sequences become actions and can therefore be executed (initiated would be a better term) concurrently by writting them in

lists using the ";" as delimiter.

WAIT expressions are used to delay or hold the execution of an action sequence, pressumably while some other actions are taking place in parallel, e.g. :

(A←A+1; NEXT B←A); (WAIT(5); NEXT C←B+C)

THE LANGUAGE, ISP

Among the goals for the notation is the ability to describe any register, data-type or operation desired by the designer of a particular computer, and it has a rich set of extension facilities, including alias declarations, text replacement, operator and process declarations, etc. Registers, operators, macros etc, are not required to be declared using explicitly reserved words. These do have to be declared of course, but the list of declarations is simply a list. Local declarations can be used to provide more flexibility; these declarations are given before the action sequence that describes the operator or process.

There is complete flexibility in the choice of names; in fact, the period (".") is included as a name character, allowing phrase-like names, e.g., Long.name. Moreover, names can be built up of arbitrary characters by enclosing any string of characters in single quotes. As a further consideration, in naming **carrier** elements no restriction is put on the ordering of elements in the structure, and naming and ordering of elements in the substructure can be used interchangeably. The **range** operator (":") denotes a range or abreviated list of values.

There is a facility to declare registers which hold information in any base, not necessarily binary; similarly, in writing most numbers, base 10 is used, although in some cases a change in number base may be desired. This is indicated, in both cases, by the **base** operator ("↓") followed by the desired base.

Infix notation was chosen for all register transfers rather than functional, prefix or postfix, simply because it is the most used in both engineering and programming cultures. When the designer defines a new operator, he is allowed to define this operation in infix mode, although functional notation can be used.

Registers can also be partial registers (defining them as subregisters of an already declared register) or a concatenation of registers. The **concatenation** operator (" ") is like any other infix operator, but it is natural for it to appear on both sides of a transfer operation.

The use of reserved words has been avoided. Practically the only reserved keywords are **NEXT**, **WAIT** and **PREVIOUS**. A large character set, including upper and lower case, special symbols etc, is used, but the notation can be used with smaller character sets (e.g. ASCII), and the transliterations of the special characters are simple and intuitive, for instance, ":=" is a primitive character in ISP and the transliteration is a ":" followed by a "=".

The policy for the use of comments requires a duplicate set of italized characters. Comments can appear anywhere in the description, and can be of any length or contain any character. For publication purposes, as is the case with ALGOL, this is easily done, but it is no doubt too much to expect from a computer implementation.

REFERENCES

Bartee, T. C. "Digital Computer Fundamentals" McGraw-Hill, Inc., 1960

Bartee, T. C., Lebow, I. L. and Reed, I. S. "Theory and Design of Digital Machines" McGraw-Hill, Inc., 1962

Bell, C. G. and Newell, A. "Computer Structures: Readings and Examples" McGraw-Hill, Inc., 1971

Bell, C. G., Grason, J. and Newell, A. "Register Transfer Design: Computers and Digital Systems using PDP-16" Digital Press, in press.

Chu, Y. "Introduction to Computer Organization" Prentice-Hall, Inc., 1970

Chu, Y. "Computer Organization and Microprogramming" Prentice-Hall, Inc., 1972

Darringer, J. A. "The Description, Simulation and Automatic Implementation of Digital Computer Processors", PhD Thesis, EE Department, Carnegie-Mellon University, Pittsburgh Pa, May 1969.

Digital Equipment Corp, PDP-11/45 Processor Handbook, 1971.

Falkoff, A. D., Iverson, K. E. and Sussenguth, E. H. "A Formal Description of System/360" IBM Systems Journal, Vol. 3, No. 3, 1964

Schorr, H. "A Register Transfer Language to describe Digital Systems" TR-30, EE Department, Princeton University, Sept. 1962

EXAMPLE

ISP Description of a Pedagogical Computer, (Chu, 1970).

```
Register Declarations
A\Accumulator<0:17>;
D\Program.Counter<0:11>;
R\Instruction.Register<0:17>;
F<0:5> := R<0:5>;
C\Address<0:11> := R<6:17>;
G\Go;
Primary Memory
M[0:4096]<0:17>;
Console Switches
Power.on;
Start.on;
Stop.on;
Interpreter
Console.activity :=  (Power.on => G←0; NEXT
          (Stop.on=>G←0);(Start.on=>G←1;NEXT Interpreter); NEXT
          Console.activity );
Interpreter:=(R←M[C];D←D+1;NEXT Execute.Instruction);
Instruction Set
Execute.Instruction := (
          Add := (F = 0 => A ← A + M[C]);
          Sub := (F = 1 => A ← A - M[C]);
          Jop := (F = 2 ∧ A<0> => D ← Address);
          Sto := ( F = 3 => M[C] ← A);
          Jmp := (F = 4 => D ← Address);
          Shr := (F = 5 => A ← Shift.right A);
          Cil := (F = 6 => A ← A<1:17>     A<0>);
          Cla := (F = 7 => A ← 0; NEXT A ← M[C]);
          Stp := (F = 8 => G ← 0; NEXT Hold);
                 (F = 9 => );
                 (F = 10 => Hold);
                 (F ≥ 11 => );
          NEXT Share);
Share := ((G => Interpret); (¬ G => Hold) );
Hold := ((¬ G => C ← 0; D ← 0); (G => Share) )
```