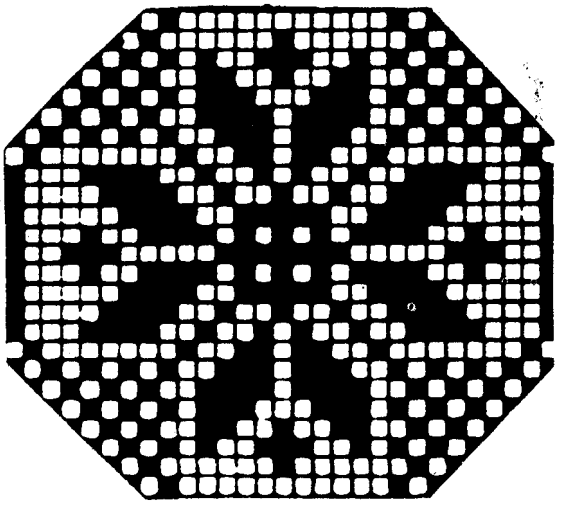


PMS: A Notation To Describe Computer Structures

by Mario Barbacci, C. Gordon Bell
and Daniel Siewiorek

INTRODUCTION

The PMS notation was developed to describe the physical structure of computer systems in terms of a small number of elementary components for textbook (Bell and Newell, 1971).

There are seven basic component types, each distinguished by the kinds of operations (function) it performs:

Memory, M. A component that holds or stores information (i.e., i-units) over time. Its operations are reading i-units out of the memory, and writing i-units into the memory. Each memory that holds more than a single i-unit has associated with it an *addressing system* by means of which particular i-units can be designated or selected. A memory can also be considered as a switch to a number of sub-memories. The i-units are not changed in any way by being stored in a memory.

Link, L. A component that transfers information (i.e., i-units) from one place to another in a computer system. It has fixed terminals. The operation is that of transmitting an i-unit (or a sequence of them) from the component at one terminal to the component at the other. Again, except for the change in spatial position, there is no change of any sort in the i-units.

Control, K. A component that evokes the operations of other components in the system. All other components are taken to consist of a set of discrete operations, each of which – when evoked – accomplishes some discrete transformation of state. With the exception of a processor,

P, all other components are essentially passive and require some other active agent (a K) to set them into small episodes of activity.

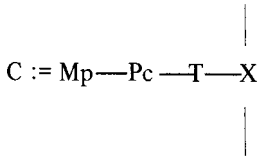
Switch, S. A component that constructs a link between other components. Each switch has associated with it a set of possible links, and its operations consist of setting some of these links and breaking others.

Transducer, T. A component that changes the i-unit used to encode a given meaning (i.e., a given referent). The change may involve the medium used to encode the basic bits (e.g., voltage levels to magnetic flux, or voltage levels to holes in a paper card) or it may involve the structure of the i-unit (e.g., bit-serial to bit-parallel). Note that T's are meaning preserving, but not necessarily information preserving (in number of bits), since the encodings of the (invariant) meaning need not be equally optimal.

Data-operation, D. A component that produces i-units with new meanings. It is this component that accomplishes all the data operations, e.g., arithmetic, logic, shifting, etc.

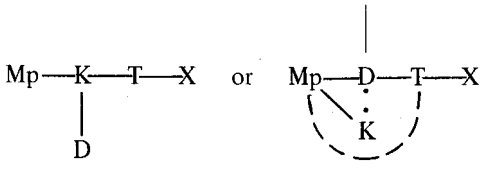
Processor, P. A component that is capable of interpreting a program in order to execute a sequence of operations. It consists of a set of operations of the types already mentioned – M, L, K, S, T and D – plus the control necessary to obtain instructions from a memory and interpret them as operations to be carried out.

Components of the seven types can be connected to make *stored program digital computers,* abbreviated by C. For instance, the classical configuration for a computer is:



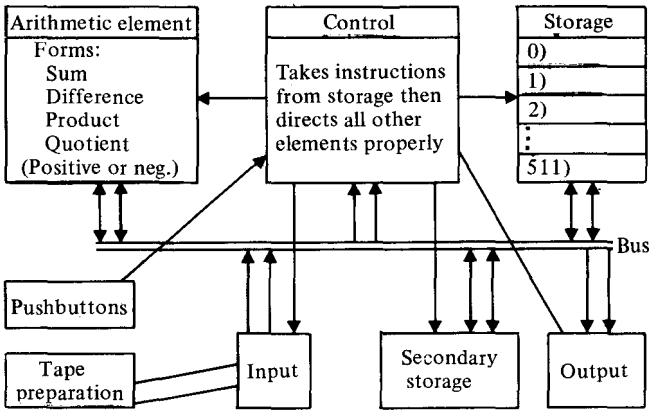
Here Pc indicates a central processor and Mp a primary memory, namely, one which is directly accessible from a P and holds the program for it. T (input/output device) is a transducer connected to the external environment, represented by X. (The colon-equals (:=) indicates that C is the name of what follows to the right.)

The classic diagram had four components, since it decomposed the Pc into a control and an arithmetic unit:

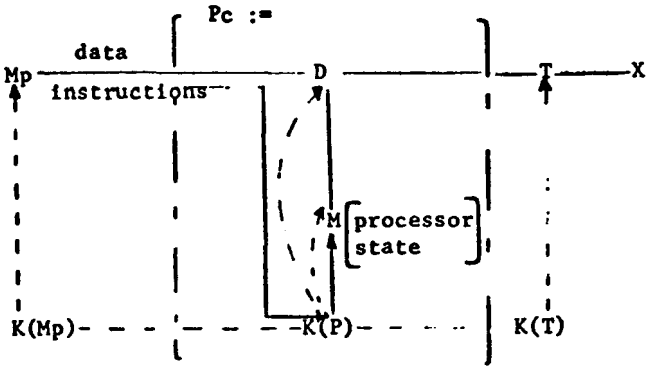


where the heavy information carrying lines are for instructions and their data, and the dotted lines signify control. Diagrams such as these correspond roughly to the conventional, simplified block diagrams of computers. The following one from the MIT Whirlwind computer is one of the earliest. Later we will diagram Whirlwind in PMS notation.

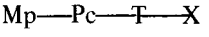
Often logic operations were lumped with control, instead of with data operations – but this no longer seems to be an appropriate way to functionally decompose the system.



Now we associate local control of each component with the appropriate component to get:



where the heavy lines carry the information in which we are interested, and the dotted lines carry information about when to evoke operations on the respective components. The heavy information carrying lines between K and Mp are instructions. Now, suppressing the K's, then lumping the processor state memory, the data operators, and the control of the data, operators and processor state memory to form a central processor, we again get:

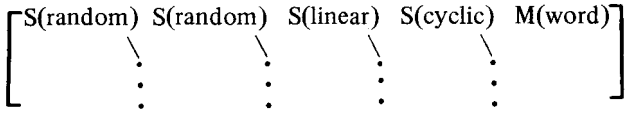


Computer systems can be described in PMS at varying levels of detail. For instance, we did not write in the links (L's) as separate components. These would be of interest only if the delays in transmission were significant to the discussion at hand. Similarly, often the encoding of information is unimportant; then there is no reason to show the T's. The same statement holds for K's. Sometimes one wants to show the locus of control, say when there is one control for many components, as in a tape controller. But often this is not of interest.

Components are themselves decomposable into other components. Memories are composed of a switch (the addressing switch) and a number of submemories. Thus a memory is recursively defined as either a memory or a switch to other memories. The decomposition stops with the unit-memory, which is one that stores only a single i-unit, hence requires no addressing. Likewise, a switch is

often composed of a cascade of 1-way to n-way switches. For example, the switch that addresses a word on a multiple-headed disk might look like:

M.disk :=



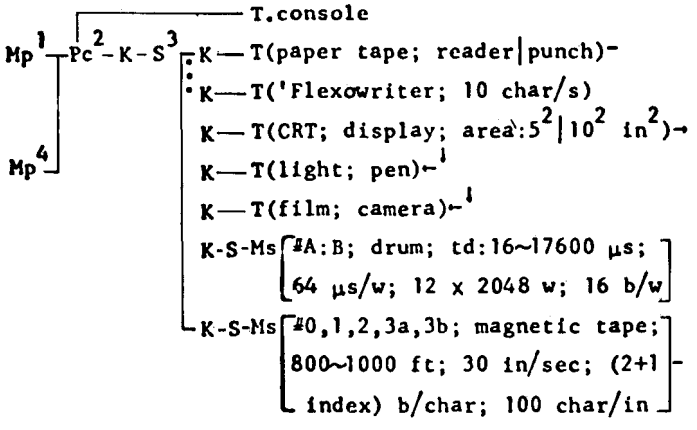
The first S(random) selects a specific Ms.disk drive unit; the second S (random) is a switch with random addressing that selects the head (the platter and side); S(linear) is a switch with linear accessing that selects the track; and S(cyclic) is a switch with cyclic addressing that finally selects the M(word) along the circular recurring track. Note that the switches are realized by differing technologies and thus have varying performance.

Various notational conventions designate specifications for a component, e.g., Mp for a functional classification, and S(cyclic) for a type of switch access function in the case of rotating memory devices like drums. There are many other additional specifications one wants to give – a single general way of providing additional specifications is used so that if X is a component, we can write:

$$X(a_1:v_1; a_2:v_2; \ldots .)$$

to indicate that X is further specified by attribute a_1 having value v_1, attribute a_2 having value v_2, etc. Each parameter (as we call the pair a:v) is well defined independently of what other parameters are given; hence there is no significance to the order in which they are written, or the number which have to be written.

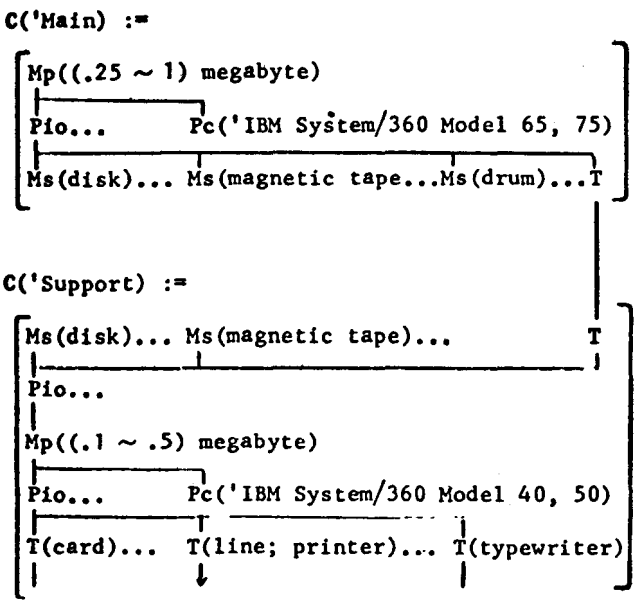
The following PMS diagrams describe actual computer systems at varying degrees of detail. The Whirlwind computer is represented as:



[1]M (toggle switch; 8 μs/w; 32 w; 16 b/w)
[2]Pc(50 kop/s; 16 b/w; 1 instruction/w; 1 address/instruction; M. processor state (3 w); technology; vacuum tube; 1948 1966)
[3]S(fixed; from: Pc: to: 8 K; concurrency: 1)
[4]Mp(# 0:1 core; 8 μs/w; 1024w; 16 b/w; taccess: 2 μs)

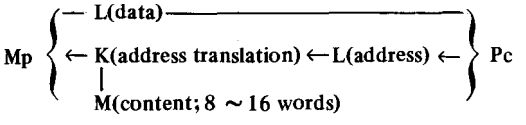
Note that most of the important attribute: value characteristics about the machine are given. In addition, it might be noted that the machine has only limited processor/input-output concurrency due to the switching structure.

At a somewhat higher level, PMS is useful for describing the structure and the interaction of various larger components in a computer network. The IBM two computer, ASP system can be represented as:



Here, we have not described several of the important characteristics such as the link bandwidth, and various characteristics of the interconnected computer although we could have.

Similarly, lower level features can be shown as in the mapping structure between a processor and its memory:



Again, a diagram might also include the information rate and width of the links, and the access time of the memory.

At a much lower level of detail, the PMS names adequately and clearly describe the structures of registers and switching circuits. Most combinational circuits correspond to data-operations D's or switches S's, and combinational circuit design consists of making more complex D's and S's. Sequential circuits take small amounts of memory M, and proceed to build controls K's. At a higher level more complex sequentially controlled P's are formed from D's and K's. Finally, the special name of P's and C's are used for particular structures.

Reference

[1] Bell, C. G. and A. Newell, *Computer Structures: Readings and Examples*, McGraw-Hill, 1971.

ISP: A Notation To Describe A Computer's Instruction Sets

by Mario Barbacci, C. Gordon Bell,
and Daniel Siewiorek

The ISP (for Instruction Set Processor) notation was developed for a text [Bell&Newell, 1971] to precisely describe the programming level of a computer in terms of its Memory, Instruction Format, Data Types, Data Operations, Interpreting a Specific Instruction Set.

The behavior of a 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 (usually in the central processor). Thus, if we specify the nature of the operations and the rules of interpretation, the actual behavior of the processor depends on the initial conditions and the particular program.

Computers are usually described in terms of the following relatively fixed format:

Memory.– Physical components which hold information encoded in data.

Primary–memory.–Contains program and its data.

Processor–state.–Registers accessible to the program – i.e. general registers and program location counter.

Console–state.– Lights and switches enabling communication with the processor.

Input/Output–state.– Controller registers accessible to the program.

Data–Types.– Described in terms of registers which could carry information.

Data–Operations.– Defining operations that can be carried out in terms of data–types.

Instruction–Format.– Specific instances of data–types.

Interpreter.– The mechanism of the processor which fetches, decodes, and executes the instructions.

Instruction–Set.– Definition of the particular instructions that the processor executes.

DEC PDP–8 ISP Description

```
Primary Memory, Mp
Memory \ Mp [0:7777↓8] ⟨0:11⟩        extended memory not
                                     included;

Processor State. Interrupt handling is not included
Accumulator\AC ⟨0:11⟩
Link.bit\L                           Link bit, extension to
                                     the Accumulator for
                                     overflow and carry;
Program.counter\PC ⟨0:11⟩
Run                                  1 when Pc is inter-
                                     preting instructions or
                                     "running";

Console State
Data.Switches ⟨0:11⟩                 data entered via
                                     console;
Input/Output State
IO.pulse.1                           pulses to I/O devices;
IO.pulse.2
IO.pulse.4

Instruction Format
Instruction.register\IR ⟨0:11⟩

 Operation.code\OP ⟨0:2⟩ :=
  IR ⟨0:2⟩
 Indirect.address.bit\IB :=
  IR ⟨3⟩                             0, direct; 1, indirect
                                     memory reference;
 Page0.bit\PO := IR ⟨4⟩              0 selects page 0; 1
                                     selects this page;
 Page.address ⟨0:6⟩ :=
  IR ⟨5:11⟩

 IO.select⟨0:5⟩ := IR ⟨3:8⟩          selects an input/output
                                     device
 IO.P1.bit := IR ⟨11⟩                these 3 bits control the
                                     selective
 IO.P2.bit := IR ⟨10⟩                generation of pulses to
                                     I/O devices;
 IO.P4.bit := IR ⟨9⟩

 This.page ⟨0.4⟩ := PC ⟨0:4⟩         current page number

Instruction Interpretation Process
Interpreter := (Run⇒                 Pc operates while Run
                                     bit is set to 1
  IR← Mp[PC] ; PC← PC+1; next        instruction fetch
  Execute.instruction; next          instruction execution
  Interpreter)                       interpretation cycle
                                     loop
Effective Address Calculation Process
Direct.address\DA ⟨0:11⟩ := (
  (PO ⇒DA := 0□Page.address); (¬PO ⇒DA := This.
page□Page.address) )

Effective.address\EA ⟨0:11⟩ := (
(¬IB⇒EA := DA);                      direct memory refer-
                                     ence
( IB⇒                                indirect memory refer-
                                     ence
  (DA ≥10↓8 ∧ DA≤17↓8⇒
   Mp[DA] ← Mp[DA]+1); next          auto indexing
  EA := Mp[DA] ) );                  defines the effective
                                     address
Instruction Set and Instruction Execution Process
Execute.instruction := (
(OP = 0⇒
  AC← AC ∧ Mp[EA]);                  logical and
```

```
(OP = 1⇒L□AC← L□AC +
  Mp[EA]);                          two's complement add
(OP = 2⇒Mp[EA] ← Mp[EA]
  + 1; next                         index and skip if 0
    (Mp[EA] =0 ⇒ PC ← PC + 1));
(OP = 3⇒Mp[EA] ← AC; AC
  ← 0);                             deposit and clear AC
(OP =4⇒Mp[EA] ←PC;
  next PC ← EA + 1);                jump to subroutine
(OP = 5⇒PC← EA);                    jump
(OP = 6⇒(IO.P1.bit ⇒                microprogrammed to
  IO.pulse.1 ← 1); next             generate 3 pulses to an
    (IO.P2.bit⇒IO.pulse.2           I/O device addressed by
  ← 1); next                        IO.select
    (IO.P4.bit⇒IO.pulse.4←
  1) );
(OP = 7⇒operate instruction. Does not include the EAE
option.
    (¬IR ⟨3⟩ ⇒                      operate group 1
    (IR ⟨4;6⟩ = 1⇒AC← ¬AC);         complement AC
    (IR ⟨4;6⟩ = 2⇒AC← 0);           clear AC
    (IR ⟨4;6⟩ = 3⇒AC ←
      7777↓8);                      set AC to ones
    (IR ⟨5;7⟩ = 1⇒L← ¬L);           complement link bit
    (IR ⟨5;7⟩ = 2⇒L ← 0);           clear L
    (IR ⟨5;7⟩ = 3⇒L ←1); next       set L to 1
    (IR ⟨8:10⟩ = 2⇒L□ AC
      ←L□ AC*2 {rotate} );          rotate left
    (IR ⟨8:10⟩ = 3⇒L□AC
      ←L □ AC*4 {rotate} );         rotate twice left
    (IR ⟨8:10⟩ = 4⇒L □ AC
      ←L □ AC÷ 2 {rotate} );        rotate right
    (IR ⟨8:10⟩ = 5⇒L□ AC
      ←L□ AC÷4 {rotate} ); next     rotate twice right
    (IR ⟨11⟩ ⇒L □ AC←
      L□ AC+1))                     increment AC, end of
                                      group 1

    ( IR ⟨3⟩ ∧ ¬IR ⟨11⟩ ⇒           group 2
    Skip.condition := (IR ⟨5⟩ ∧
      AC ⟨0⟩ ) ∨(IR ⟨6⟩∧AC=0)
      ∧(IR ⟨7⟩ ∧L) (IR ⟨4⟩ ⇒
        AC ← 0);                    clear AC
    (Skip.condition ⊕ IR ⟨8⟩ ⇒PC
      ←PC + 1); next                skip
    (IR ⟨9⟩ ⇒AC← AC ∨ Data.
      switches); next               "read" console switches
    (IR ⟨10⟩⇒ Run← 0) );            halt, end of group 2
)                                     end op operate
)                                     instruction end of
                                      instruction execution
                                      process
```

Memory

Memory components or information carriers are hierarchically organized information structures, in which each level consists of a number of subcarriers, all identically organized. This decomposition eventually yields elementary carriers that can not be further decomposed (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.

Carriers are defined in ISP by a name and description of their structure, where the number of subcarriers at each level of decomposition is given by bracketed lists of names (if specific names are associated with the subcarrier) or constants, much like array declarations in Algol, e.g.:

AC ⟨0:11⟩ *AC is the name of a carrier, a register 12 bits wide, named from 0 to 11 (from left to right). The ":" or range operator is used to denote an abbreviated list of elements.*

For descriptive purposes there is an abbreviation or alias operator "\", which is used as a delimiter for a list of names, all of which are thus made equivalent, e.g.:

Accumulator \ AC ⟨0:11⟩ *is a valid definition of the carrier, but now it can be referred to as either "Accumulator" or "AC" indistinctly.*

Memory \Mp[0:7777↓8] ⟨0:11⟩ *square brackets are used to specify those dimensions where the accessing is done through some "addressing" (switching) scheme. The memory consists of 4096 words, each of 12 bits, named, from left to right: 0, 1, . . ., 11. Constants are, by default, decimal numbers, unless otherwise specified by the ↓(base) operator.*

Elements are specified by "names" (numbers do not indicate relative position), therefore, it is legal to describe a 7 bit register as:

R ⟨A;15;B;13;11;9:10⟩

The only concession to the use of numbers as both names and position indicators is by using the range (":") operator, whereby, the abbreviated lists consist of the bounds and all integers in between, with the implication that these consecutive numbers also name consecutive (from left to right) elements.

Carriers do not necessarily have bits as their 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 definition, a base ("↓") operator and a "size" (i.e. the size of the alphabet) operand, e.g.:

A ⟨0:3⟩ ↓16 *is a register of 4 elements; each one can assume as value a hexadecimal digit.*

TR ⟨0:7⟩↓3 *a ternary register, 8 characters long, the characters are named TR ⟨0⟩ ...TR ⟨7⟩.*

Data operators

Data operators produce bit patterns with new meaning, they do the real processing by transforming information. Data operators work on data types (which are composed of a value or meaning and a representation or encoding of information). Associated with the data types we have carriers, the physical components used in storing and transmitting the data types.

Data operations create information (instances of data types) with new meaning, in which process they may destroy some existing information. The data operators take their inputs (data type carriers), operate on the data and present the result as output (the resulting data type carrier). Data operators are essentially intercarrier communication networks, whose complexity varies from a simple transfer path to combinational networks to more complex transformations involving sequences of simpler operations.

Data operators in ISP include the following classes:

–Concatenation (□)
–Boolean (¬, ∨, ∧, ⊕, ≡)
–Arithmetic (+, −, ∗ ÷)
–Relational (=, ≠, <, ≤, >, ≥)
–Transfer (←)

Operation sequences

In ISP, concurrency of actions is the rule rather than the exception, and it is reflected in the used of the ";" as a delimiter for lists of concurrent actions. Sequencing is expressed by using the term "next" as a delimiter for lists of sequential actions. Complex concurrent and sequential activities can be described in terms of simpler activities using "next", ";", "(", and ")" in a recursive way, e.g.:

IR ← Mp[PC] *single action*

IR ← Mp[PC] ; PC ← PC + 1 *concurrent actions*

IR ← Mp[PC] ; PC ← PC + 1; next Processor.state ← 1 *action sequence of two steps in parallel followed by a third step*

(IR ← Mp[PC] ; PC ← PC + 1; next Processor.state ← 1);(AC ← 0; MQ ← AC) *concurrent action sequences*

(OP = 2 ⇒ Mp[EA] ← Mp[EA] + 1; next (Mp[EA] = 0 ⇒ PC ← PC + 1)) *conditional action sequences can be defined in terms of conditional action sequences. Parenthesis are used to indicate the scope of the conditional activities.*

Instruction expressions

Instructions are described by *instruction expressions* (conditional actions) of the form:

condition ⇒ action-sequence

where the condition (a Boolean expression which is either true or false) describes when the instruction will be evoked, and the action sequence describes what transformations take place between what memories.

Since all operations in a computer result in modifications of bits in memories, each action in a sequence takes the following form:

memory–expression ← data-expression

the data-expression, patterned after standard mathematical notation, describes the transformation of information (if any) and the information pattern that is to be placed in the memory described by the memory-expression, e.g.:

(OP = 2 ⇒ AC ← AC ∧ Mp[EA]) *If the contents of carrier OP is equal to 2 then the action is performed.*

Modifying data operations

Expressions can be followed by a modifier, providing more information about the meaning and interpretation of the operands and operators. A modifier consists of a data type name or an operation mode enclosed in curly brackets "{" and "}",e.g.:

L□AC ← L□AC * 2 {shift}
L□AC ← L□AC * 2 {rotate}
A ← B + C {1's complement}
A ← B + C {2's complement}
A ← ((B + C {1's complement}) * 2 {shift})

The instruction format

The instruction register, because of its important function, has (usually) a more complicated structure than most other internal registers (not physically, but by the meaning assigned to its components). It is always divided in fields, with proper names that provide information to the programmer about their function during the interpretation cycle. Thus, we have *operation codes, addresses* (with modifiers: *modes, bases, indexes), device selectors* and *commands* for i/o instructions, *micro-commands* for micro-operation instruction, etc. This factorization of the register bits is not unique to the instruction register, for instance we may refer to the sign of the accumulator register by its own name, or to fields in the *processor state register* (a register containing a selected subset of the processor status information).

These subfields are declared in terms of the main register, but are used as if they were independent registers, with their own structure and naming conventions, e.g.:

Instruction.Register \ IR ⟨0:11⟩ *the instruction register is declared as part of the processor state*

Operation.code \ OP⟨0:2⟩ := IR ⟨0:2⟩ *the operation code field consists of the first three bits of IR*

Indirect.address.bit \ IB := IR ⟨3⟩ *the fourth bit of IR specifies the addressing mode*

Page0.bit \ PO := IR⟨4⟩ *the fifth bit of IR selects the page in memory*

Page.address ⟨0:6⟩ := IR ⟨5:11⟩ *the last seven bits of IR define the page address*

IO.select ⟨0:5⟩ := IR ⟨3:8⟩ *the device selection field. Subfields can overlap.*

The interpretation cycle

During the execution of the program, some set of bits (an *instruction*) is read from Mp to a memory within Pc, called the *instruction register.* This set of bits then determines the immediately following sequence of operations. After this sequence has occurred, the next instruction to be executed is determined and obtained, and the entire cycle repeats itself. This *interpretation cycle* is performed by a part of the processor called the *interpreter.* The effect of each instruction can be expressed entirely in terms of the information stored in memories at the end of the cycle (plus any changes made to the outside world).

During execution, operations may have their own internal states, as sequential circuits, which are not represented as bits in memory. But at the end of the cycle, whatever effect is to be carried on to a later time has been staticized in bits of some memory.

This modularization of the description allows the designer to divide the processor in conceptually independent units (the actual hardware may or may not be structured in that way).

Instruction.Interpreter := (Run ⇒ Fetch; next Instruction.Execution; next Instruction.Interpreter
) *this sequence activates the* Fetch *and* Instruction.Execution *processes and loops i.e. the Interpretation cycle*
Fetch := (IR ← Mp[PC] ; PC ← PC + 1) *the instruction fetch process*
Instruction.Execution := (
(OP = 0 ⇒ . . . .);
. . . . . .
(OP = 7 ⇒ . . . .)
); *defines each instruction in terms of the operation code to which it responds.*