CERN-Data Handling Division
DD/80/30
A. van Dam
M. Barbacci
C. Halatsis
J. Joosten
M. Letheren
December 1980

Simulation of a Horizontal Bit Sliced Processor Using the ISPS Architecture Simulation Facility

(To be published in a special issue of IEEE Transactions on Computers: Micro-programming Tools and Techniques)

DD-ml

Simulation of a Horizontal Bit Sliced Processor Using the ISPS Architecture Simulation Facility

A. van Dam
Brown University
Providence, RI

M. Barbacci
Carnegie-Mellon University
Pittsburgh, PA

C. Halatsis
NRC Democritos
Athens, Greece

J. Joosten
CERN
Geneva, Switzerland

M. Letheren
CERN
Geneva, Switzerland

14 May 1980
1 December 1980

Table of Contents

1. Introduction

The MICroprogrammed filter Engine (MICE), [Halatsis, 1980] is a fast, microprogrammable processor built with ECL bit slices (Motorola ECL 10800 series) intended primarily to be used as an on-line data filtering engine for high energy physics experiments. The processor supports both user microcode and emulation of a subset of the DEC PDP-11 architecture (no floating point, memory management or multiple interrupt levels).

MICE was designed and implemented at CERN, Geneva, Switzerland during the latter portion of 1978 and the first half of 1979. The design parameters, intended application, and architecture of MICE are described in [Halatsis, 1980]. In this paper we concern ourselves with the use of a hardware description language used to model and simulate the hardware during its development. We treat the problem of describing a pipelined, horizontal (112 bits wide) host machine, implemented using bit slices with considerable potential for parallelism.

2. Simulation as a Design Tool

As indicated in [Halatsis, 1980], there were six basic reasons for using simulation as a design tool:

1. To 'verify' the correctness of the host design, by tracing the execution of micro-instructions.

2. To be able to write and debug micro-code (e.g. the target emulator) before the hardware was ready, and indeed even afterwords; debugging on a time-sharing terminal with good debugging and diagnostic features is easier than doing it on a minimally available piece of hardware.

3. To check the hardware against a non-varying standard 'definition' so as to be able to 'certify' it.

4. To allow a quick assessment of the impact of changes (fixes, enhancements, etc.)

5. To have the machine description provide 'living', i.e. dynamic, interactive project documentation which is easier to understand than static, passive diagrams, code, or text, and is always up-to-date.

6. To allow measurement, evaluation, and identification of bottlenecks.

Rather than implementing a simulator in a general purpose language, the designers adopted an existing, well debugged, and documented architecture description/simulation facility based on the ISPS notation [Barbacci, 1977, 1978]

The notation and its use in a variety of applications are described elsewhere [Barbacci, 1981]. The objective of this paper is to demonstrate the advantages of high level software tools and the thought processes that permitted the successful development of a MICE model in a very short period of time.

In particular, we want to illustrate the novel use of an instruction set description language in a project involving a horizontal micro-programmable processor.

This paper is not intended to be a tutorial or a survey on hardware description languages. A large number of these languages have been proposed and some have even been implemented. However, many tend to be used in fixed areas of application, or specialize in a particular level of design, or are built around a specific model of timing and synchronization, or are even specialized for a particular component technology. This variability restricts most comparisons to a rather small set of dimensions. Nevertheless, research on hardware description languages is very active, and readers interested in tutorials or comparisons of the most popular languages should consult [CHDL, 1974; CHDL, 1975; CHDL, 1979].

3. An Overview of MICE

MICE is meant to provide a good compromise between the speed of hardwired logic and the flexibility of software in the real time environment provided by high energy physics experiments. A typical experiment generates large volumes of data (10^6 to 10^7 particle interactions/second) which are analyzed in real time by fast, hardwired logic, to look for some simple properties of the data which characterize interactions of a desired type. When an event of the right nature is detected, data is collected (in the order of 10^3 words of 16 bit each) and transmitted to a Data Acquisition Computer (DAC). The detector is disabled during the data collection and transmission phase. The percentage of 'interesting' events which are lost while the detector is disabled can be quite high (80%) thus preventing the full utilization of a very expensive facility. In addition, only a minor fraction (perhaps less than 10%) of the events collected provide 'good' results, and only after hundreds of hours of CPU time on large computers (CDC-7600 class) have been used to filter them out.

MICE is intended to provide a flexible second level detection mechanism by performing more complex algorithms than are used in the very fast hardwired primary detector. It can reject 'uninteresting' events selected by the primary detection mechanism in a shorter time than would be required for a full read-out to the DAC. An added advantage is that the data collected contains a higher percentage of 'good' events, thus reducing the amount of off-line data processing required to filter out a given number of good events.

Figure 1, depicts a typical system configuration. MICE performs a filter algorithm on a subset of the event data made available to it by a fast read out system connected (via DMA) to its target memory. If an event is to be rejected, the engine issues the REJECT signal which clears the system and awaits the next trigger interrupt. If an event is to be accepted the engine interrupts the DAC which then reads the complete event from the detectors.

In MICE a user can write low level micro-code for time-critical algorithms and, using standard software development tools, he can program non-critical tasks at a higher level, using the fixed-point PDP-11 instruction set. Used this way, MICE provides a range of performances between 3 and 10 times that of a PDP-11/70. When special functional units are added to the CPU and controlled by micro-code, its performance can approach that of special hardwired controllers (approximately 50 times the performance of a PDP-11/70).

In the MICE project, design and implementation time were more critical than raw speed. As a consequence, standard bit slices rather than discrete ECL logic were chosen. The goal of simplicity of microprogramming lead to an environment in which the user was somewhat isolated (via a large set of micro-code assembler macros) from the actual encoding of bit patterns in the micro-code word (49 fields, 112 bits/word, with almost no field encoding). This technique had important consequences for the modelling technique that was finally adopted.

Figure 2 is a block diagram of the engine. The Target Memory (TM) contains PDP-11 instructions and data. The Memory Interface (MI) contains registers and an ALU used to compute effective addresses. The Register File (RF) contains both scratchpad registers and the PDP-11 general purpose registers, with the exception of R[7] (the program counter) which is kept in the MI module. In one micro-cycle, the contents of two registers containing prefetched PDP-11 target operands maybe sent to the ALU, operated on, and the result stored back in the RF. The Micro-sequence Controller (MC) controls the flow of the micro-program contained in the Writable Control Storage (WCS). Additional logic, external to the bit slices is used to enhance flexibility and speed by providing extra registers and data paths for time-critical operations.

Micro-instructions are pipelined in standard fashion by fetching the next micro-instruction while the current micro-instruction is being executed. Machine instructions for the PDP-11 target machine are also pipelined with the support of 3 registers (T, IR, and A in the figure) and the ALU inside the MI module. When fully utilized, the host can execute a target machine instruction while the next target instruction is being decoded, a third target instruction (or operand) is being fetched, and yet a fourth target instruction's address is being computed (in MI). In order for this scheme to operate correctly with the PDP-11 instruction set, some non-critical restrictions on flow of control are necessary. For example, one can not 'manufacture' a PDP-11 instruction and then execute it as the next instruction.

4. 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.

In the simplest case, a unit is simply a carrier (a bus, a register, a memory, etc.), completely specified by its bit and word dimensions, as shown in Figure 3.

The examples in the figure are taken directly from the specification of the Writable Control Store (WCS) in MICE. The control memory consists of 1K words, each 128 bits wide (only 112 bits are currently used). Micro-instructions are loaded into the Pipeline Register (PR) for decoding and execution.

Different fields of a micro-instruction control different data paths and functional units in the machine. For convenience of description and debugging, these fields tend to be grouped according to their function. Fields must be declared by specifying a name and a structure (e.g. ALUMX1IOF<9:0>), together with the corresponding portion of the carrier (e.g. PR<111:102>) of which they are a part. Notice that bit 'names' used in the left and right hand side of a field definition are completely independent of each other. Thus, bit '9' of ALUMX1IOF is mapped onto bit '111' of PR, bit '8' onto bit '110', etc. Fields can be mapped over other fields, as in the definition of ALUFF<5:0>. For 1-bit fields (or carriers), there is no need to specify a bit name inside the '<' and '>' brackets as in the declaration of DGI0OBF.

Hardware behavior can be modelled by procedures containing data and control operations. Figure 4 displays an abridged copy of the behavioral description of the Micro-sequencer Control (MC) unit. Procedure declarations are similar to those of a high level programming language, with a procedure name followed by a (possibly empty) list of parameters and the body enclosed in a BEGIN...END block.

The MC procedure computes micro-instruction addresses depending on fields of the current micro-instruction, external branch conditions, and other signals. The ICF field of a micro-instruction is used as an 'operation code' which is decoded and its value used to select one of a number of alternative register transfer sequences. This selection mechanism is implemented in ISPS with the DECODE operation. If the value of ICF is 0, the micro-instruction address register, CR0 is pushed onto a small stack, internal to the MC unit and a new value (computed by procedure NA) is loaded. If the value of ICF is 15 (Hexadecimal F), a return address is popped from the stack into CR0 while another register (CR1) is incremented by the value computed by procedure CIN1. To further draw the analogy between the role of the ICF field and the operation code in a vertical machine, each of the alternatives is labelled with both the value of the ICF field (0,1,...,"F) and an instruction mnemonic, as in **'0\jsr := ...'**, where 0 is the operation code and JSR stands for Jump to SubRoutine. The '\' operator is used to introduce aliasses for constants (as in the example) or identifiers, as in **'pr\pipeline.register'**.

The two operations illustrated in the figure (JSR and RTN) differ in one important aspect. Although both consist of two steps, the steps are separated by different delimiters: 'NEXT' is used to indicate a

sequential operation while ';' is used to indicate a concurrent operation. Concurrency is ISPS in defined as 'process' concurrency and no assumptions are made about the synchronization of the operations. Thus, conflicting use of source and destination carriers, as in 'A = 5;B = A' can yield unpredictable results (no assumptions of the form *compute all expressions into temporary variables before performing the transfers* can be made.)

In the general case, a unit consists of an interface (carrier) 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 manner[1]. This is illustrated by the PUSH and POP procedures used in the description of MC (Figure 5).

5. The Structure of MICE

As described in [Halatsis, 1980], there were some problems with the functionality of the bit slices which resulted in the addition of external SSI and MSI components. These were required to provide data paths which were not present in the standard slices, and also to provide functions that were too expensive (i.e. slow) to obtain using the functions already available in the slices.

For example, to calculate the next micro-address and then fetch the corresponding micro-instruction in one cycle, it was necessary to add external table look-up logic (i.e. mapping ROMs) in order to (conditionally) branch on:

1. the source and destination operand addressing modes,
2. the target instruction operation code,
3. interrupts and internal hardware status signals.

During the micro-cycle in which the branch occurs, the mapped micro-address is also loaded into the sequencer's micro-address register (CR0) so that it is available for the next micro-address calculation in the following cycle.

Similarly, an external target memory address register was added to allow bypassing of the memory interface's MAR and its associated strobing to give us a single cycle address-and-fetch for the more common or simple PDP-11 addressing modes. Also, a number of external gates were added to provide symmetry of data flow not available from the data paths of the slices. The net effect was an increase in the complexity of the host architecture. In addition, at least in principle, unpredictable micro-code can be written and it was a policy decision to discourage this practice by providing the

---

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

users with high level macros to implement a more structured, virtual architecture, yet without sacrificing speed by hiding architectural features of the horizontal host machine.

The system designers could then take advantage of the features of the host machine by programming at the individual field level, while normal users could program a higher level virtual host machine which still takes advantage of the machines's inherent parallelism[2].For example, register to register arithmetic and logical operations involving the register file, the ALU, and external multiplexers and gates controlled by a dozen or so fields, were encapsulated in simple macros specifying source and destination operands and an operation code.

6. Modelling

One of the most important decisions to be made whenever simulation tools are to be used is the selection of a level of modelling. Several such levels were conceptually feasible and in this section we describe the thought process followed before we decided on a particular style of description and simulation.

At one extreme, the modelling of the system as a PDP-11 instruction set processor was clearly inadequate since the objective was not to test the target instruction set but the host machine design. A PDP-11 instruction set description would have hidden all of the important features and potential problems in the host machine.

At the other extreme, the combination of standard slices, together with the avoidance of pathological micro-instructions eliminated the need for expensive, detailed simulation at the gate level.

Modelling the higher level virtual host seen by most users suffered from the same problems that led to rejecting the PDP-11 instruction set level as a viable alternative. This level hides most of the timing and concurrency details that characterize the host machine. Thus, while it is acceptable and even desirable for a user to be unaware of these details, the designers needed a level of description closer to the actual hardware.

Modelling the host at the component level, describing and simulating the bit slices and the extra logic and data paths was closer to what was needed. However, even at this level, much unnecessary detail and expense could be incurred. The level of description finally adopted could be characterized as 'virtual slice' simulation. Thus, while we described and simulated the operation of the individual

---

[2]Notice that the parallelism which must be dealt with at the host level is not usually found at the target level. Conventional architecture simulation in general and ISP simulation in particular have not had to deal with the problems of untangling parallelism and races since the input/output operations are usually handled separately.

slices, not all details were included, only those portions of the slices that are used in the host. Thus, for instance, the BCD arithmetic capabilities of the slices are not used in the host and are not included in the description.

Using the 'virtual slice' approach does not, of course eliminate all the potential sources of difficulty in the description and simulation of the host machine. We will address two of these problems in the remainder of the paper.

6.1. Order of Firing of Activities and Granularity of Time

Since each of the virtual slices is generally active during one or more phases of the five phase clock cycle, we were faced with a fundamental problem of how to simulate these many parallel activities, and at what level of time granularity. In conventional instruction cycles for a vertical micro-programmed host or a target machine, there are essentially only a few independent activities at most during each clock cycle and one simply simulates the individual register transfers and functional units on a clock cycle basis. Our problem appeared more complex, in that there are half a dozen functional units operating in parallel and potentially communicating with each other by sharing buses at various times during a single micro-cycle. Furthermore there are many latches and registers inside each chip, which are read and written at different times (not necessarily on a clock edge) and a very large combination of functions to be handled, given our micro instruction with fourty-nine fields.

We first attempted to reduce this combinatorial complexity by finding a single canonical order for firing the slices and simulating both inter- and intra-slice source-destination register transfers. This attempt failed because different combinations of individual functions often require different timing sequences, and even if we would be satisfied with a number of different sequences, there were too many to be individually analyzed. Furthermore, there appeared to be many register transfer cycles of the type A->B->...->A, which are legal in the hardware because of built in propagation delays and strobes to control the synchronization. They cannot be implemented in ISPS by just writting the transfer directly because order dependent conflicts between old and new values of carriers would occur during simulation.

The solution we implemented was not to try to approximate the actual (micro) timing of each of the slices, but rather to implement only the crudest level of timing, that of strobing the registers internal to each slice as well as the external ones at the right phase of the five phase clock. This implies that the state of the simulated machine (i.e. the status of both clocked and unclocked components) changes only at discrete time intervals, the five phases of the clock (see Figure 6). These timings are based on worst case propagation delays and therefore the simulation does not mimic the hardware faithfully, a point that should not concern the microprogrammer. The manner in which changes in sources are propagated to their destinations is described below. Conflicts are resolved by introducing explicit

master and slave copies of each of those registers which can act both as source and destination during a microcycle. Old values are copied from the slave copies while new values are strobed into the master copies. At the end of the microcycle all master values are transferred to their slaves. In this way, the intrinsic parallelism of the machine is simulated at a level which is both manageable and gives a realistic picture of what the microprogrammer can expect to see in each register at each clock edge.

Note that the firing order problem would appear even if we used ISPS to model the parallel activity with parallel processes, say one process for each 'virtual slice', or even one process per major activity such as register file read and register file write. This is because such processes would have to be synchronized using the ISPS 'DELAY(time)' primitive so that a *consuming* process gets an up-to-date value from its *producing* process, via the interface carrier. This again implies determining mutually consistent amounts of delay, which is equivalent to defining a canonical order of firing.

We have however used the parallel processor facility of ISPS to model independent activities going on in our machine, e.g. MICE CPU, and the two DMA processes (see Figure 7). In addition a number of parallel processes can be activated in order to simulate CAMAC interface commands that control MICE in real time (e.g. setting and clearing of control signals such as *diagnose*, *suspend*, *halt-clock*, etc.)

6.2. Propagation of Changes

A problem closely related to that of deciding firing order was that of deciding how to propagate changes. The straightforward method would appear to be to propagate a change from its source to its destination(s), as soon as the change takes place. For example, if we model a bus as a simple carrier, driven by multiple sources, the description would contain several statements of the form:

source.1 = expression NEXT bus = source.1

This is clearly incorrect since the value present in the bus would then reflect the value of the last source driving the bus. A more correct description of the behavior of the bus sources is given by statements of the form:

source.n = expression NEXT bus = source.1 op source.2 op ...

Where 'op' is either 'AND' or 'OR'[3]. A shorter and more readable description can be achieved by defining the bus as a procedure of the form:

bus()<..> := Begin bus = source.1 op source.2 op source.3 op ... End

---

[3] In general, when multiple sources can drive the bus lines, an implied logical operation takes place. Typically, the sources are OR-ed or AND-ed together and this must be clearly displayed.

Now, whenever any of the sources changes, the procedure must be invoked to load its carrier with the correct value:

source.n = expression NEXT bus()

Whenever the current value in the bus is to be used as input to some component, the bus carrier can be used in an expression. For instance:

x = y + bus

In this style of description, activation of procedures describing combinational logic occur as a consequence of a change in some input carrier. There are two problems, however: 1) there is an implied retention property being attached to the bus carrier, and 2) there is a spreading of information (invocation of procedures) throughout the description because each of the entities for which the bus is a source must also be invoked in turn and so on, to propagate the original change.

An alternative mechanism is to transfer the activation of the bus procedure from the site where an input carrier changes, to the site where the bus is to be read. In this way, when an entity is accessed, we then ask which sources might have created it as their destination, and consequently what its up to date value should be. In other words, we don't propagate a change as it happens, but only as it is needed to supply the latest value to a carrier affected by the change. Thus, source changes can be described as before:

source.n = expression !notice, no bus activation

Whenever the bus is to be used as input to some component, its value must be 'computed' explicitly:

x = y + bus()

In this style, no retention properties are implied (the value is computed whenever needed, using the latest values of its sources), the description is shorter, and moreover, the knowledge about the identity of the bus has been restricted to those places that use the bus as an input. In effect, this method handles propagations by going backwards from destination to source whenever needed rather than spreading from a source to all its destinations everytime the source changes. While at first this recomputation every time the bus is needed seems very ineficient, it does not happen more than a few times for each slice per microcycle and it is the best way to guarantee that the bus contains "fresh" results, without timing conflicts. Furthermore, we avoid needlessly propagating changes to entities which are not in turn used as sources.

The same considerations were applied to other components without retention properties (e.g. output of multiplexers). All components without retention properties were modelled as procedures to be invoked whenever their output lines were to be used in an expression. Thus one views such logic

not as passive passthrough of its inputs but as an active entity, whose procedure computes a function of its inputs. To illustrate this principle, the example in Figure 8 describes one of the multiplexors driving the OBus (MX5), Figure 2.

The example indicates that the rule implies a regression from the desired output values back towards its sources, through potentially many levels of logic (e.g., OB, MX5, CONSTANT, CONSTANT.ROM). The regression stops whenever a source with retention properties is encountered (e.g. a register). It is assumed that a source without retention properties (e.g. OB) has reached steady state at the moment that its carrier is to be used.

The latter is a direct consequence of the level of description. In a gate level simulation, events are continuously repeated until signals reach steady state. In a register transfer level simulation, signals are computed once unless explicitly described otherwise. Although the latter is clearly more efficient, errors in the design could be easily overlooked. To detect some of these errors, two features of ISPS were used. The first one was the use of a predeclared carrier in the language. This carrier is dubbed 'UNDEFINED' and can be used as any user declared carrier. When it is used as a source in a register transfer statement, the destination carrier is marked as having an undefined or illegal value which is readily detected by the simulator. In the MICE description we assume that carriers (in particular, combinational logic carriers such as buses and multiplexers) reach steady state at their right time, determined by the actual physical timings we expected. They are UNDEFINED before this time. Any attempt to access one of these carriers before the right time is therefore detected by the simulator and the proper diagnostic message is issued.

Figure 9 illustrates the use of UNDEFINED. The RD.CM carrier is assumed to be undefined during the first three phases of the clock (clk = 1, 2, 3). It reaches steady state during phase 4, at which point one of several potential sources is loaded into the carrier. The procedure depicts the behavior of the CAMAC read interface. An attempt to transfer (i.e. read) a value from the CAMAC interface into one of the CPU carriers before clock phase 4, is trapped by the simulator as an error (this is of course, a worst case assumption since different sources become available at different times).

The second feature of the language that was used in the description was the detection of 'recursive' calls on an already executing procedure. This allows the detection of potential race conditions. Specifically, the enabling of some chains of data paths could lead to a situation in which a bus is being driven by a signal from a slice which is in turn driven directly from another bus, which in turn is receiving a signal from the first bus. This loop takes the form of a series of nested calls on the procedures describing the behavior of the combinational data paths along the loop. Eventually, when the loop is closed, an attempt is made to call the procedure that started the chain and this is again detected by the simulator.

An example of this situation could arise when for instance, the register file (RF) is to be loaded from the input bus (IBUS) through the A6 port (Figure 2). To compute the value in IBUS, all its inputs are evaluated and AND-ed together. One of these inputs comes from the memory interface (MI) slice, port I3. This port could be driven, through the slice internal data paths by the O3 port, which in turn requires the evaluation of the output bus (OBUS) signal. This signal is the conjunction of several possible sources, and one of these (G3) could be driven in turn by the input bus, whose value we were trying to compute in the first place!

7. Conclusions

The MICE description models the CPU, buses, CAMAC interface, and DMA interface in 5000 lines of ISPS (including extensive comments). The CPU description was written in 1 man-month, the remainder in another 2 man-months by MICE designers without prior experience with ISPS.

When the description is compiled and linked with the simulator run time system, the program occupies 400 Kbytes of memory, 110 Kbytes of which is the simulator run time system (this portion does not depend on the particular ISPS description being simulated, while the description specific portion grows with the complexity of the description and the size of the memories and register files).

Simulation of a single micro-cycle requires 400 milliseconds of PDP-10 (KL-1080) processor time. The load on the machine is, however, slight, on the order of a few CPU minutes per hour, since man-machine interactions (commands, responses, and execution traces) dominate the total elapsed time.

In a system of the complexity of the MICE host, there are many ways to get in trouble due to conflicting use of data paths. We were not interested in being able to simulate ALL microcode sequences one could write, only 'friendly' micro-code, trapping anything that looked suspicious as an error.

The responsibility for detecting errors has been divided between the micro-assembler and the simulator. The former catches errors resulting from simultaneously enabling potentially conflicting data transfers. The latter catches errors resulting from accessing data before it has settled (still UNDEFINED) or from accessing the wrong data (recursive calls).

The actual implementation encountered no logical errors thanks to the thorough simulation; the only problems were ECL technology related. As the hardware is extended, the simulator is in constant use to verify the correctness of enhancements and alterations. Also the DEC PDP-11 diagnostics are run after each change to give an additional check. It is fair to say that the project would not have been completed as rapidly and as cleanly without our powerful software tools.

8. References

[Halatsis, 1980] C. Halatsis, A. van Dam, J. Joosten, and M. Letheren, "Architectural Considerations for a Micro-programmable Emulating Engine Using Bit Slices", *Proceedings of the 7th Annual Symposium on Computer Architecture*, IEEE-CS and ACM, La Baule, France, May 1980.

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

[Barbacci, 1978] M. Barbacci and A. Nagle, "An ISPS Simulator", Technical Report, Department of Computer Science, Carnegie-Mellon University, 1978.

[Barbacci, 1981] M. Barbacci, "Instruction Set Processor Specifications (ISPS): The Notation and its Applications", IEEE-CS *Transactions on Computers*, Vol. C-30, No. 1, January 1981.

[CHDL, 1974] *Proceedings of the 2nd International Symposium on Computer Hardware Description Languages*, Darmstadt, ACM German Chapter Lectures W-1974.

[CHDL, 1975] *Proceedings of the 3rd International Symposium on Computer Hardware Description Languages and their Applications*, New York, September 1975.

[CHDL, 1979] *Proceedings of the 4th International Symposium on Computer Hardware Description Languages*, Palo Alto, California, October 1979.

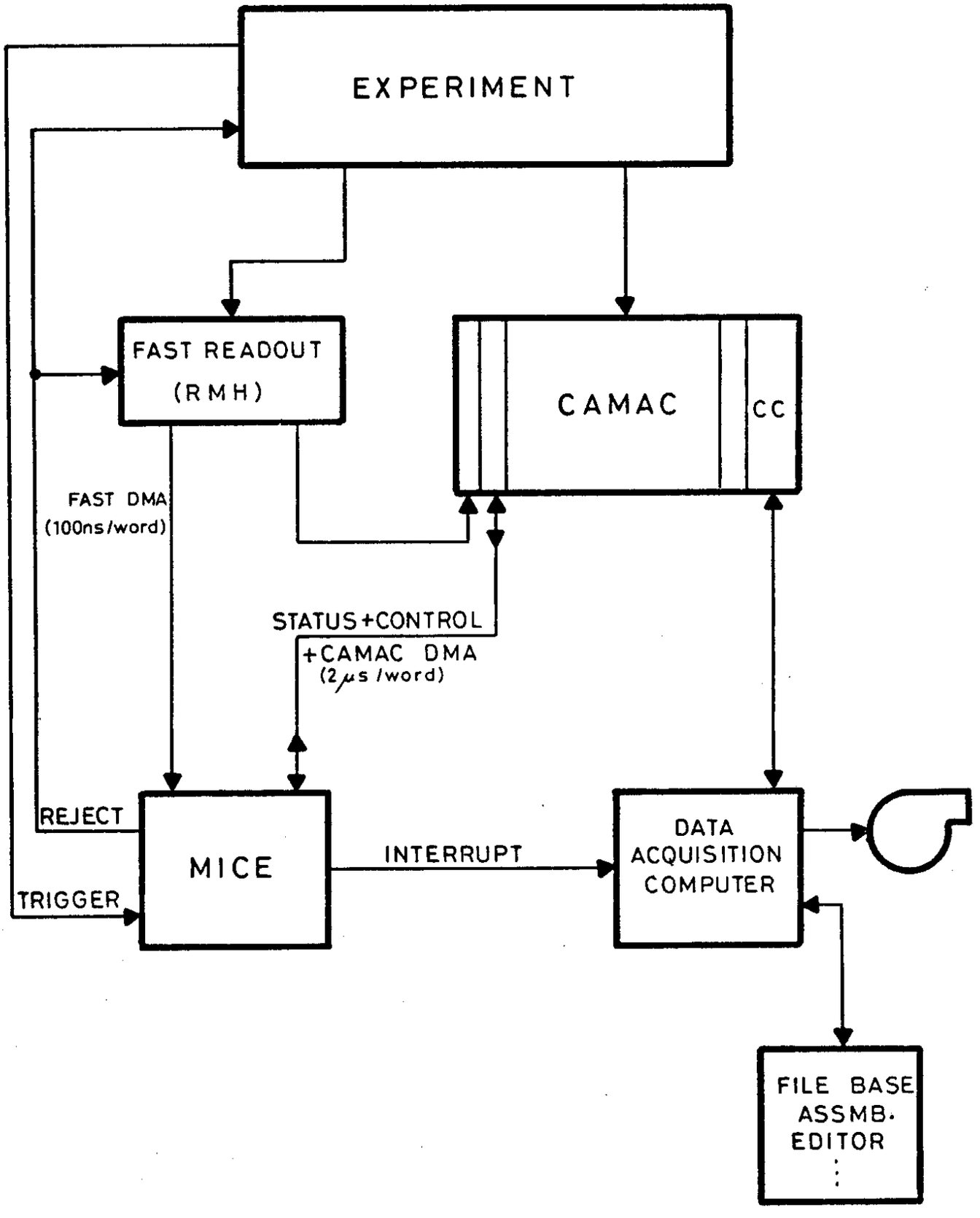

Figure 1: System Configuration [Figure 2 in Halatsis, 1980]

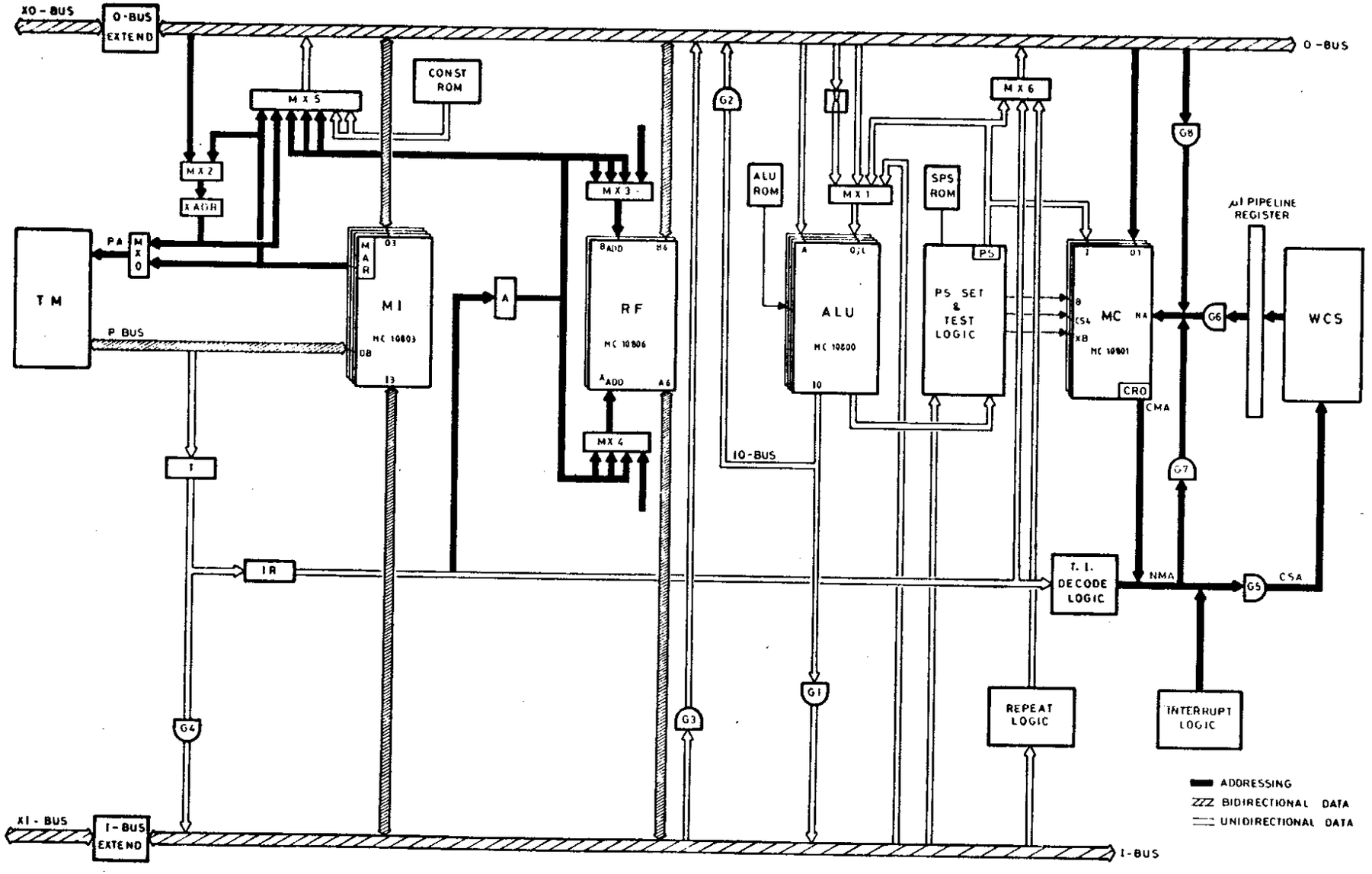

Figure 2: MICE CPU [Figure 4 in Hellatsis, 1980]

```
wcs[1023:0]<127:0>,        ! 1K words of writable control store, each 128 bits

pr\pipeline.register<127:0>,                          !Micro-instruction register

                                                       ! Micro-instruction fields
sparef<15:0>     := pr<127:112>,                                    !not used
alumx1iof<9:0>   := pr<111:102>,                                 ! ALU/MX1 control
        aluff<5:0>       := alumx1iof<9:4>,                        ! ALU function
        smx1f<1:0>       := alumx1iof<3:2>,                        ! MX1 function
        dgi0ibf<>        := alumx1iof<1>,                            ! G1 control
        dgi0obf<>        := alumx1iof<0>,                            ! G2 control
. . . . . .
rfcf<17:0>       := pr<17:0>,                                  ! RF/MX3/MX4 control
        raf<>            := rfcf<17>,                             ! Output to IBus
        waf<>            := rfcf<16>,                            ! Input from IBus
        rbf<>            := rfcf<15>,                             ! Output to OBus
        wbf<>            := rfcf<14>,                            ! Input from OBus
        sarfaf<1:0>      := rfcf<13:12>,                  ! MX4 (address) control
        sarfbf<1:0>      := rfcf<11:10>,                  ! MX3 (address) control
        arfaf<4:0>       := rfcf<9:5>,                      ! Direct input to MX4
        arfbf<4:0>       := rfcf<4:0>,                      ! Direct input to MX3
```

Figure 3: ISPS Declaration of Micro-Instruction Format

```
mc() :=
Begin
Decode icf =>                                          ! Micro Instruction Control
        Begin
        0\jsr  := Begin push(cr0) Next cr0 = na() End,
                  . . . . . .
        "F\rtn:= Begin cr0 = pop() ; cr1 = cr1 + cin1() End,
        End
End
```

Figure 4: Procedure MC Describes the Behavior
of the Micro-Sequencer Control Unit

```
push(x<15:0>) :=                          ! The PUSH procedure accepts
        BEGIN                                     !a 16 bit carrier
        cr7 = cr6  NEXT
        cr6 = cr5  NEXT
        cr5 = cr4  NEXT
        cr4 = x
        END,

pop()<15:0>  :=                         ! The POP procedure pops the top of
        BEGIN                             ! the stack and returns the value
        pop = cr4  NEXT
        cr4 = cr5  NEXT
        cr5 = cr6  NEXT
        cr6 = cr7  NEXT
        cr7 = "0000
        END,
```

Figure 5: The Units PUSH and POP Illustrate
a Carrier/Procedure Combination

```
MAIN ucycle() {PROCESS} : =
    BEGIN
    init() NEXT                                          ! System initialisation
    pr = wcs[csa()] NEXT                                 ! Load pipeline reg. on
                                                         ! phase1 of first cycle
    REPEAT
        BEGIN
        clk = 1 NEXT
        t1()   NEXT                                      ! phase 1 procedures
        clk = 2 NEXT
        t2()   NEXT                                      ! phase 2 procedures
        clk = 3 NEXT
        t3()   NEXT                                      ! phase 3 procedures
        clk = 4 NEXT
        t4()   NEXT                                      ! phase 4 procedures
        WAIT ( NOT hold() ) NEXT                           !Wait while hold on
        clk = 5 NEXT
        t5()   NEXT                                      ! phase 5 procedures
        IF init.semaphore => ( RESTART ucycle )
        END
    END,
```

Figure 6: Clock Phases

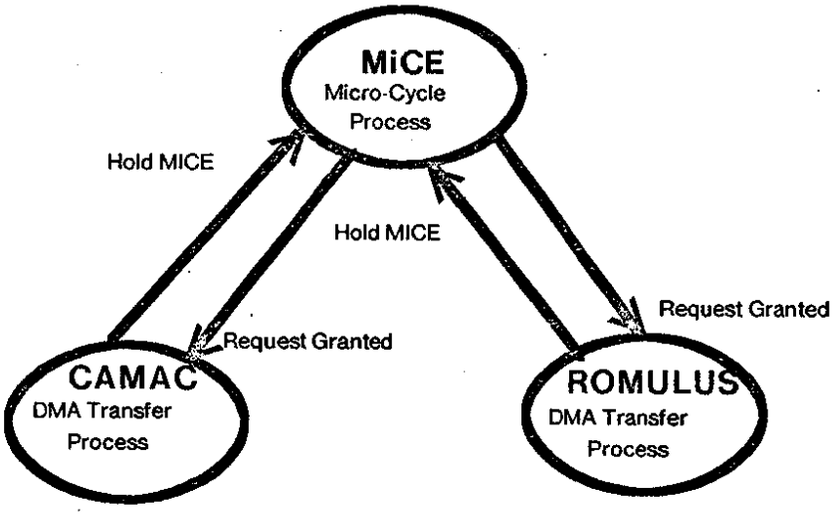

Figure 7: System Parallel Processes

```
ob()<15:0> :=
        BEGIN
        ob =    x.ob() AND mx5() AND mi.o3() AND rf.b6() AND
                g2() AND mx6() AND g3() AND mc.o1()
        END,
. . . . .
mx5()<15:0> :=
        BEGIN
        DECODE smx5f =>                          ! field of micro-instruction
                BEGIN
                6 := mx5 = constant(),           ! zero extend 8-bit constant
                0 := mx5 <= constant(),          ! sign extend 8-bit constant
                3 := mx5 <= a<7:0>@'0,            ! sign extend branch offset
                2 := mx5 = a<5:0>@'0,               ! zero extend, SOB offset
                1 := mx5 = a<3:0>,                                ! status bits
                5 := mx5 = xadr,
                4 := mx5 = mar,
                7 := mx5 = "FFFF            ! Hex FFFF to put 1's on the bus
                END
        END,
. . . . .
constant()<7:0> :=
        BEGIN
        constant = constant.rom[acstf]
        END,
. . . . .
constant.rom[15:0]<7:0>,
```

Figure 8: Modelling a Multiplexer

```
rd.cm()<15:0> :=
BEGIN
DECODE (clk LSS 4)  =>
        BEGIN
        0\false := DECODE pa()<4:1> =>
                        BEGIN
                        1:= rd.cm = dma.ba,          ! Read DMA base address
                        2:= rd.cm = dma.wc,            ! Read DMA word count
                        3:= rd.cm = ubus.wreg,               ! Read PC value
                        6:= rd.cm = "00 @ mice.int.st(),   ! Interrupt status
                        OTHERWISE:= rd.cm = "FFFF
                        END,
        1\true := rd.cm = UNDEFINED()
        END
END,
```

Figure 9: CAMAC Read Interface