SOFTWARE TOOLS FOR MICROPROCESSOR BASED SYSTEMS

C. Halatsis

ABSTRACT
After a short review of the hardware and/or software tools for the development of single-chip, fixed instruction set microprocessor-based systems we focus on the software tools for designing systems based on microprogrammed bit-sliced microprocessors. Emphasis is placed on meta-microassemblers and simulation facilities at the register-transfer-level and architecture level. We review available meta-microassemblers giving their most important features, advantages and disadvantages. We also make extensions to higher-level microprogramming languages and associated systems specifically developed for bit-slices. In the area of simulation facilities we first discuss the simulation objectives and the criteria for choosing the right simulation language. We concentrate on simulation facilities already used in bit-slices projects and discuss the gained experience. We conclude by describing the way the Signetics meta-microassembler and the ISPS simulation tool have been employed in the design of a fast microprogrammed machine, called MICE, made out of ECL bit-slices.

I. INTRODUCTION
The past few years have seen dramatic developments in the area of microprocessors. The fast advancing LSI semiconductor technology has been used to both produce microprocessors with increased performance, as is the case with the new generation of super-micros, and to produce microprocessors with increased functionality, such as the single-chip microcomputers which combine on the same chip most of the functional parts (CPU, RAM, ROM, I/O) of a computer.

Bipolar bit-slices and supermicros are the two high performance products of the µP industry used already for the construction of mainframes and minicomputers. Bit-slices derive their high performance from the bipolar technology itself and their ability to be strung together and controlled via microprogramming to form a complete high performance computer system. The supermicros derive their high performance from their sophisticated architecture rather than the use of faster logic, although their clock rates are generally higher than those of their 8-bit and 16-bit predecessors. The number and bit length of their registers, the power and regularity of their instruction sets, and the flexibility of their addressing modes enable the new supermicros when properly programmed to qualify more like mainframe CPUs than most existing minicomputer CPUs. This increased performance of bit slices and supermicros has enabled the use of microprocessors in one of

\textsuperscript{a)} Digital Systems Laboratory, Computer Center, NRC"DEMOCRITOS", Athens, Greece
the two philosophically diverse application areas of microprocessors, namely
the computer industry. The other and more vast area is the "user-packaged
microprocessors". The term denotes a user defined µP-based product composed
of microprocessor chips along with memory, I/O chips and other support cir-
cuits. There is an enormous diversity in user-packaged microprocessor appli-
cations ranging from replacing hardware sequential "random" logic to very
complex real-time applications.

The following three aspects provide a unified approach to the develop-
ment of a user defined µP-based system.

(a) hardware and software development aspects are synergistic.
(b) range of functions need to be implemented, and
(c) ordered sequence of activities involved in the development of a total
product.

That hardware and software development aspects for a bit-slice based
system are synergistic is undoubtful. To carry-out a good design with bit-
slices requires the mastery of an enormous amount of logical and timing
detail of the slices as well as choice of the functions and bit patterns of
the microprogram. This is especially true when one wants to come up with a
host-machine architecture that emulates efficiently a given target machine.
The same is also true for single-chip µP-based systems. Writing software
for such systems involves much more than just "writing" the machine instruc-
tions to perform some functions. The programmer should be a competent de-
signer especially when his system is to replace some hardware sequential
logic. Only by the grateful integration of the hardware and software aspects
one will benefit from the increased performance/functionality of present
µP's.

The appearance of new µP product does not usually coincide with cor-
responding supporting software. This implies that the user usually has to
develop himself all software for his µP-based system for functions that
range in level from handling interrupts up to providing a friendly user
interface.

The sequence of steps involved in the development of a µP-based pro-
duct are as follows
dead fa1 1) (see Fig.1).

(a) System requirements. The purpose of this stage is to produce a system
requirements document that addresses the issues of why the system is
needed and how it will fulfill these needs. A context analysis questions
the need for such a system and why certain technical and economical con-
straints are to be used. In effect this stage establishes the performance
that must be achieved, the functions that are to be performed, and the
design constraints (technical, operational, and economical).

(b) System design. The first step here is to select the proper microprocessor
and next to see which of the functions specified in the system require-
ments are to allocated to the hardware, firmware, and software parts of
the system, under the established design constraints. This leads to firm hardware, firmware, and software requirements.

(c) **Hardware, firmware, and software specification.** Based on the hardware, firmware, and software requirements the hardware, firmware, and software aspects of the system are established using well established techniques such as stepwise refinement, top-down design, hierarchical decomposition and modularity. Detailed logic diagrams, and microprogram and program flow-charts are the outcome of this step.

(d) **Hardware, firmware, and software implementation.** At this stage the detailed logic diagrams are turned into wiring lists, chip placement, PCB boards. The program and microprogram flowcharts are transformed into code modules using suitable programming languages which then are translated into the machine languages, and the microlanguage of the uP.

(e) **Debugging of hardware, firmware, and software separately.** This is the first step of a multi-step bottom-up test procedure which is completed at the next stage of the design process. Hardware debugging involves the removal of short-circuits, open-circuits, blown-up IC’s, wrong wirings, etc. Firmware and software debugging involves removing the programming language errors and then removing logic errors.

(f) **Integration and validation.** This stage involves the integration of the hardware, firmware, and software into one system and testing it to ensure it functions as described in the requirements specification.

(g) **Maintenance.** This is the final often overlooked step. The first kind of maintenance is correction of system faults. This may be small if the previous steps have been followed carefully. The second kind of maintenance is updating due to changes in requirements, introduction of new functions, changes in the processing or data environment and possibly need for performance enhancements.

Successful, efficient and timely implementation of each of the above design steps necessitates the use of appropriate tools. The proliferation of uP’s have led to a bewildering array of development tools that facilitate the application of these devices. The purpose of these lectures is to present and review critically the available software tools. We shall concentrate on tools for the design phases of implementation and testing for both single-chip fixed instruction set and bit-sliced uP’s. We shall elaborate on the tools for bit-slices and we shall go in detail on a meta-microassembler and simulator facility we used in the implementation and testing of a fast micro-programmed machine called MICE, made out of ECL bit-slices.

**II. DEVELOPMENT TOOLS**

The rapid proliferation of microprocessors has spawned a bewildering host of tools that facilitate the use of these devices. These tools range from pure hardware ones such as oscilloscopes and logic state analysers
upward through hardware/software micro-based development systems to pure software tools that run on minis and time-sharing systems. Figure 2 gives the spectrum of the available development tools for microprocessors. In practice a mixture of these tools are needed for each specific project.

1. **Oscilloscope**. It is the most primitive tool for debugging hardware. It gives the user pulse characteristic capability and is useful for debugging a specific function rather than isolating a problem from the beginning because it can not stimulate the system.

2. **Logic analyser**. It is an improvement over the oscilloscope. Its basic ability is to monitor and store in memory different sets of inputs and display them later in binary or other suitable form on a CRT. As the oscilloscope, the logic analyser is a passive observational tool and does for state domain information essentially what the oscilloscope does for time domain information. Using various display formats (timing diagram, state diagram, state map) the logic analyser helps the designer in performing both logic state analysis and logic timing analysis. The former requires synchronous sampling of the digital information and so it uses the clock of the system under test. The latter needs fast, asynchronous sampling using its own clock which may be 4 to 20 times faster than the clock of the system under test.

3. **Microprocessor analysers**. They are similar to logic analysers except that they can display digital information in assembly-language mnemonics (disassembly) rather than ones and zeros. Because of this, they are suited for monitoring the address and data busses and effectively for viewing the program flow in mnemonic form. Usually plug-in units personalize the analyser to each particular microprocessor.

4. **PROM programmers**. They are used to write programs into programmable ROMS. Program debugging with PROM's by using a PROM programmer is an iterative process of manually keying into the PROM's, checking them in the system, correcting bugs in the listing and reprogramming. This loop goes on until the user obtaining an operational code. When a PROM programmer is a peripheral device to a development system operates under the control of utility software that reads the object code of a program, writes it into PROM and verifies the PROM contents. In general this tool should be used in the final stages of product development and not as a primary debug aid.

5. **Programmer panels**. This tool is usually a display with switches that resembles a computer front panel. It represents a crude way to debug a μP-based system.

6. **Prototyping kits**. These are complete build-it-yourself microcomputer systems, valuable aids in learning about microprocessors rather than used as a development aid. They are considered to be the ancestors of the micro-based development systems. They are usually equipped with a
Fig. 1. Design steps.

Fig. 2. Spectrum of µP Development tools.
minimal monitor on ROM that allows the user to load a small program in
machine code and run it.

7. Microcomputer development systems (MDS's) \cite{1} An MDS is a data-
processing unit used mainly for software development of μP-based systems.
Most of them also support the integration of the software and the hard-
ware. Others stop short of the integration step, but support the other
steps of the software development cycle, namely design, coding and de-
bugging on the MDS itself. MDS's have common capabilities of processors
memory, console, mass storage, in-circuit emulation and system software.
They differ in the type of console, mass storage device, in-circuit emu-
lation architecture, high-level language support and universality (whether
or not they can support different microprocessors).

7.1. Development systems for bit-slices\cite{3}.
The main purposes of these development tools are to

- provide the control store for the microcode
- provide instrumentation for debugging and tracing the execution of
  the microprogram
- accept the user system built out of Bit-slices.

These development systems range from stand-alone systems (as it is
the case of the system 29 of Fig. 3(a) used for the 2900 series bit-slices),
through systems that look as peripherals to a standard MDS (as it is the
case of the MACE 29/800 system of Fig. 3(b) used for the 2900 and 10800
series bit-slices), to systems that are MDS-plug-in emulator cards (as it
is the case of the ICE-30 emulator for the 3000 series bit-slices).

One interesting feature in one of these systems (that of Fig. 3(b)) is
the capability to configure the control store after the user needs (2kw x
112 bits, 4kw x 48 bits, 6kw x 32 bits, and 8kw x 16 bits).

7.2. Debugging aids on an MDS \cite{3,4}. These aids facilitate error detection in
program testing. Programs loaded into memory are executed under the super-
vision of the debug system. Single-instruction stepping, program tracing,
and breakpoints are the fundamental features of a debug system. Additional
commands handle memory display, program execution, data storage in memory,
and set values in the processor registers. Recent advances in this area are:
(a) symbolic debugging. This enables the designer to refer to memory loca-
tion by name rather than absolute numerical address. This capability is
especially handy for dealing with relocatable programs. (b) multiple break-
points or a sequence of breakpoints by ANDing or ORing breakpoint condi-
tions.

The above debugging aids are useful, as said, at the software test
phase. Additional debugging aids are provided at the integration phase by
the in-circuit emulators immediately below.
Fig. 3. Development systems for bit-slices.
(a) Stand alone system. (b) Peripheral to an MDS.
8. **In-circuit emulator**. This is one of the most stimulating tools for use primarily in the integration phase of the design. It may be thought of as a combination of a debugger and a logic analyser. It comes as a component of an MDS. It helps the designer to monitor, control and modify in an interactive and dynamic way, the interplay between the hardware and the software of the μP-based system under development. In particular, it extends into the prototype system interactive debugging and logic analysis facilities. The in-circuit emulator cable and associated buffers give the MDS access to the target (prototype) system itself. In most cases an in-circuit emulator can emulate the target CPU, work as logic analyser and simulate ROM, RAM, and I/O.

As a CPU emulator it allows the user to do the following interactively:

- Examine and modify all CPU registers and RAM location of the prototype.
- Execute the target program in single-step, multistep or continuously.
- Start and stop program execution at will.
- Input and output data to prototype I/O ports.
- Specify conditional breaks to permit flexible control over hardware/software debugging.

As a logic analyser, the in-circuit emulator monitors and stores address, data, and control-signal activity at the CPU. Its trace memory records this information synchronously. The memory may be examined by the user later on enabling the designer to look back at the last N program steps executed.

As a ROM/RAM simulator, the in-circuit emulator allows you to substitute the memory of the host (MDS) system into the prototype. A memory-mapping capability enables the user to test target memory in small increments by executing some code in the memory of the target processor and the rest in emulator’s memory.

**Mini and large computer systems**. They support the cross-development approach mainly for software and firmware development of a μP-based product. Typical cross-tools are cross-assemblers, cross-compilers of high level languages, meta-assemblers and meta-microassemblers, and simulators. These cross-tools combined with other tools already available on minis or large computer systems, such as text editors, file-management systems, operating system, utilities and libraries form a set of design tools for microprocessor software (and firmware) development usually far more efficient than the corresponding set of tools resident on an MDS. The reason is four fold. First is the increased computing power of the host machine (mini or large system) and its peripherals on which the cross-tools run compared to the MDS’s μP. Second reason is that the cross-language translators are usually more powerful than their corresponding
MDS-resident translators. Third reason is the increased sophistication of the remaining software support (editors, operating system etc) compared to the MDS-resident software. Because of the latter, these systems are often used to produce and store project documentation in addition to the software/firmware development. The main disadvantage of these systems is the lack of the real-time debug facilities offered by the ICE of an MDS. Other disadvantage is the costly transfer of the developed software/firmware to the prototype system. This usually is done either by burning PROMs or by paper-tape. As a consequence turn-around times for program fixes are large. A final disadvantage is that the cross-development may prove very costly if the user is not aware of his costs. This is especially true for cross-development using a time-sharing service. However, careful use of time-sharing can avoid large initial capital outlays, while supporting a number of programmers on the same project.

With the exception of one manufacturer of bit-slices who offers some resident tools, firmware development for the rest relies solely on cross-tools.

Basic concepts and characteristics of cross-tools are given in the appropriate sections below.

**Integrated µP-development systems.** These combine the real-time debug facilities of MDS's and the efficient environment for µP software/firmware development of mini and large computer systems, eliminating thus their respective shortcomings. Typically an integrated µP development facility consists of a mini or large computer system with cross-development tools connected to a variety of MDS's and user prototype systems. The system can support concurrently many users and a variety of microprocessors. Software/firmware is mainly developed on the mini or large computer interactively via a terminal using the cross-development tools. Then the object code is down-loaded to the appropriate MDS for real-time debugging. Finally the debugged software/firmware is down-loaded to the prototype system for final testing.

An interesting integrated system is the TRIAD system developed at the MIT Digital System Laboratory. Fig.4 shows the structure of TRIAD. The basic cell in this system is a "triad" that allows close communication among the mini-based development system, the MDS, and the prototype. The system meets the following crucial characteristics that lead to a unified system approach to the design, debugging and software (firmware)/hardware development of microprocessor-based systems:

(a) The system is suited to both engineers and programmers.

(b) It allows easy access to the entire spectrum of development tools, including higher-level problem-oriented languages, automatically configurable hardware structures, and down-loading mechanisms.

(c) The software/firmware development tools are closely linked to the
Fig. 4. The TRIAD Integrated µP development system.
hardware under development allowing thus rapid tailoring of the development system to the application under consideration.

(d) The system has a high throughput and quick turnaround to modifications by retaining a history of the development effort.

(e) The system is modular both in size and flexibility, making very easy future extensions.

Moreover, the system is intended to serve in the future the system requirements and design phases. Ultimate goal is to provide the capability to cast the user’s problem in some appropriate formalism and transform it into an appropriate configuration of microprocessors and a set of software/firmware modules to run on this configuration.

In the following sections we review software tools for single chip fixed instruction set μP’s and bit-slices.

III. SOFTWARE TOOLS FOR SINGLE-CHIP FIXED INSTRUCTION SET MICROPROCESSORS

There are five main categories of software tools all for μP software development:

. editors
  . language translators
    . assemblers
    . compilers
    . interpreters
  . loaders and linkers
  . Simulators
  . Debuggers

In addition to these there is also the system software of the host machine on which these tools run. The system software includes:

. operating system
. file management
. Utilities

Depending on whether or not the host machine is the same or not as the microprocessor for which software is developed some of the above software tools are classified as resident tools (MDS) and cross tools (minis and large computers).

1.- Text editors. These are programs that allow the programmer to enter, modify, and store in a file his source program. Usually the programmer enters his program in assembly or high-level language through a keyboard or paper tape into the editor’s local buffer where it is then edited using appropriate commands and transferred finally to a file ready for translation. Using appropriate commands the programmer can insert, delete, copy or replace a single character, a string of characters, a whole line, a whole page or even a whole file of source text (Fig.5). Obviously not all editors for μP source program
development provide all these facilities. In fact the first resident editors were very primitive usually tailored to the requirements of the µP assembly language only.

Editors are often classified as line editors and page editors depending on whether the "editing window" is a single line of text or a set of lines called page. The latter requires a CRT terminal with local buffer. Commands such as move or find are used to position the editor pointer to the editing window.

An editor usually has several levels of operation mode. Typical is the inefficient case of two level implementation, namely command level and text level. A good and friendly text editor depends not only on the versatility of the text editing commands but also on the easy and safe way of moving among the various levels of editing modes.

No doubt the text editors on large time sharing computers are far more sophisticated and friendly that those provided on an MDS. This is due not only to the simple fact that the former have been around longer but also to the fact that the supporting system software (operating system and file management) is more powerful.

2.-Language translators. Assemblers, high level language compilers, and interpreters are the three categories of language translators which may come in the form of resident or cross tools.

Assemblers and compilers translate source programs into µP machine code for later execution. An interpreter, on the other hand, does not generate machine code. Instead it actually executes some code each time an executable set of source statements is encountered. An interpreter is superior
to an assembler or a compiler in that the executing program can be interrupted, changed, and resumed. Each source statement is interpreted anew each time it is to be executed.

Among the factors that should influence the selection of the uP to be incorporated into the user product is the availability of language translators in connection with the characteristics of the system to be developed, and the system on which to run the translator.

Start-up costs is one of the factors to be taken into account. These costs chiefly consist of the cost of a development system plus the programming costs. Fig.6 shows the way the start-up costs vary with respect to the level of the programming language to be used.

Another factor is programming efficiency. Typical measures are memory-use efficiency and execution time. Fig. 7a gives memory-use efficiency in terms of the program size and the level of the programming language. For assembly language efficiency depends on programmer experience. For a high-level language efficiency is very low for small programs and approaches for large programs the efficiency of assembly language used by an experienced programmer. It is, however, higher compared to assembly by a novice programmer. Efficiency above 100% can be achieved with a high-level interpreted language.

A third factor is programming productivity/cost. Fig.6 shows also the way programming costs vary with respect to program size. Generally the programming productivity is much higher for HLL's and hence the programming cost much lower.

There is a crossover point between the total costs of a system programmed respectively in assembly language and in a HLL. This is due to the fact that the reduced memory cost of a system programmed in assembly language outweighs the reduced HLL programming cost above certain number of systems to be delivered. This is shown in Fig. 7b.

Reference 17 reviews in a critical way the microprocessor assemblers of six popular single-chip microprocessors. It also provides a list of the most important features to look for which support good program design. These features are:

. Source, Object, and Listing Formats which are Easy to Use, Read, and Understand
. Ability to Define and Manipulate Meaningful Symbols
. Ability to Specify Data Constants in the most Meaningful Form
. Ability to Specify an Arithmetic or Logical Expression, Evaluate it, and Use it as Operand in an instruction or directive
. Provision of alphabetical listing of symbols with their values
. Provision of alphabetically sorted Cross-Reference listing of all symbols along with the statement defining the symbol and the statements referring to this symbol

19)  
20)
Fig. 6. Start-up costs and programming productivity/cost

Fig. 7a. Programming memory-use efficiency

Fig. 7b. Crossover Point
- 255 -

. Good Error Diagnostics
. Parameterized Macro Facility
. Conditional Assembly Facility
. Relocatable Object Code
. Provision of Linkage Editing Capability

Microprocessors have caused a proliferation of programming languages. At least twenty different high-level microprocessor language are in use today. These are thoroughly reviewed in Ref.21. They are classified according to their distance from assembly language into the following five categories:

(1) structured assemblers like PLz-ASM
(2) high-level machine dependent languages like Smal, BSAL, Mistral, PL/65
(3) high-level uP-oriented languages like MPL, PL/M, PLz-SYS
(4) high-level Systems Languages like Basic, Pascal, RTL/2, C
(5) high-level application oriented languages like APL, FORTRAN, COBOL.

It falls outside the scope of these lectures detailed reference to these languages. The interested reader may refer to Ref.21 for a critical review of the basic characteristics of these languages (program structure, data types, operators, expressions, control structures, procedure calls and parameter, memory allocation, input/output, and compile-time operations). The only point we wish to raise is the spectrum of translation options of these languages and the tradeoffs among these options with respect to program execution, preparation and portability. Translation option range from full translation, through translation to assembler and intermediate language to full interpretation. Translation to Abstract Machine Code (AMC) and then interpretation, is a good compromise for non-time critical applications. Already a chip set has been announced which can directly interpret PCODE the AMC of Pascal.

3.- Loaders and Linkers. A loader transfers the object program from an external medium, like paper tape or a diskette file, into the microcomputer RAM. Other function of a loader may be (a) the conversion of a relocatable code into loadable code and (b) the establishment of linkages between object modules with reference to each other.

Linkage editing. The relocation function implies that the language translator produces relocatable code. The linkage editing also implies a language translator capable to communicate linkage information. Both these functions, contrary to the main function of loading, may be cross-operations.

4.- Simulators. They are cross tools which allow the user to test (execute) his object program on a host machine. The usual simulators provided by the uP-manufacturers simulate them at the machine instruction level, leaving usually out input/output operations, interrupts and other time critical operations. For this reason they can not completely replace program testing on the microcomputer itself (for example on an MDS using a resident debugger). The Simulators also provide diagnostic information, normally not provided
by a debugger, plus some statistics.

The above simulators are totally inadequate to be used in a multi-
microprocessor environment. In this case what is needed is a simulator at
the chip-level. Simulation at the chip-level requires that both the data
operations of the processor and the interface signal changes that result
from these operations be simulated. Such a chip-level simulator is also
useful to study the propagation of faults through the microprocessor and the
processing of indeterminate inputs. Fig.8 presents the block diagram of
such a simulator which has been integrated in a general logic simulator\cite{22}. The simulation of the signal line changes has been achieved by employing a
form of high-level microprogramming. Signal control words are stored in a
fixed control table.

Problems related to the level of simulation are discussed in more de-
tail in the next section.

IV. SOFTWARE TOOLS FOR BIT-SLICES

While the development of a single-chip fixed instruction set µP-based
system involves mainly software development, the development of a system
based on bit-slices involves equally well hardware and firmware development.

Bit slices are ordinarily used to built a host machine with an archi-
tecture tailored to the specific needs. This host machine may be programmed
by the end user at the following levels:

(a) host level only using microprogramming (firmware)
(b) target level only by providing first, once and for all, a microprogrammed
emulator of a target machine (usually a standard mini) and then using
the available software development tools of the target machine
(c) host and target level using both firmware and software development tools.

In this section we examine software tools that aid the hardware and
firmware development of a bit-slices based system. Microprogramming languages
with their associated translators and simulators are two main classes of
these tools.

1.- Microprogramming languages. In this area the microassembler continues to
be the most widely accepted tool for writing microcode. An ordinary micro-
assembler provides mnemonic names for each microoperation; writing a micro-
program consists of combining these mnemonics into microinstructions. The
microassembler, however, does not assist the microprogrammer to exploit the
parallelism available in a horizontal host machine. The microassembler pro-
vides also the usual facilities of an assembler, namely, use of labels,
starting address in the control store, cross-reference tables, load maps, etc.

Microassemblers for bit-slices have, in addition to the above, two im-
portant features. One is that it provides for the definition of the host
machine microinstruction. The second is a macro facility that allows the
microprogrammer to combine microoperations into higher level operations. These two features classify the microassemblers for bit-slices as meta-microassemblers to emphasize the ability to be customized to a particular host architecture, exactly the same way as meta-assemblers are customized to a particular target machine.

All available meta-microassemblers belong to the class of the adaptive meta-assemblers. Microprogramming using a meta-microassembler proceeds in three distinct phases (see Fig.9):

(a) **Definition phase.** In this phase the user using the statements of the meta-language of the meta-microassembler defines

- microinstruction width(s) and formats
- fields, sub-fields, and default values
- mnemonics
- symbolic microassembly language opcodes

The meta-microassembler acquires these definitions, parses them if necessary, and builds an internal representation of the host machine. Effectively it adapts itself to recognize the user microassembly language.

(b) **Assembly phase.** This phase is similar to the normal assembly phase of an ordinary assembler. It accepts the source microprogram statements written in the previously defined users microassembly language and produces object microcode using the internal representation of the host machine. Usually more than one source microprogram statements are used
to write a single microinstruction in the user's microassembly language. These statements are converted into bit patterns of the fields involved which are then assembled to a single microinstruction word.

(c) **Formatting phase.** At this phase the object microcode is suitably formatted ready to be loaded into the control store of the host, or to be used for burning PROMs, or to be passed to a simulator for execution/verification/debugging. The formatting usually involves reordering of the various fields of the microinstruction according to the physical layout of them. This feature implies that at the definition phase the microinstruction fields can be ordered in any logical way the microprogrammer desires.

Reference 23 reviews critically most of the available micro-assemblers for bit-slices. Fig.10 presents a comparative summary of their features.

Desirable features for bit-slice microassemblers are:

- easy to learn definition meta-language
- ability to define any format and width
- default values and default microoperations
- ability to support the definition of a flexible, simple and of readable syntax user microassembly language
<table>
<thead>
<tr>
<th>GENERAL</th>
<th>AMDASM</th>
<th>CROMS</th>
<th>DAPL</th>
<th>MICRO-AID</th>
<th>RAPID</th>
<th>RAVASM</th>
<th>SIGNETICS</th>
<th>MACE</th>
</tr>
</thead>
<tbody>
<tr>
<td>AVAILABILITY</td>
<td>CSC</td>
<td>NCSS</td>
<td>RCC</td>
<td>NCSS</td>
<td>NCSS</td>
<td>NCSS</td>
<td>NCSS</td>
<td>EXORciser</td>
</tr>
<tr>
<td>SOURCE LANGUAGE</td>
<td>SYSTEM 29</td>
<td>PL/I</td>
<td>FORTRAN</td>
<td>NOVA</td>
<td>PL/I</td>
<td>FORTRAN</td>
<td>TYMERSHARE</td>
<td>6800 µP</td>
</tr>
<tr>
<td>GENERAL PURPOSE</td>
<td>FORTRAN</td>
<td>FORTRAN</td>
<td>PL/I</td>
<td>PL/I</td>
<td>FORTRAN</td>
<td>FORTRAN</td>
<td>FORTRAN</td>
<td>6800 µP</td>
</tr>
<tr>
<td>MAIN BIT-SLICE SUPPORT</td>
<td>YES</td>
<td>NO</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td></td>
<td>2900</td>
<td>3000</td>
<td>2900</td>
<td>6700</td>
<td>(ROM)</td>
<td>2900</td>
<td>2900</td>
<td>2900</td>
</tr>
</tbody>
</table>

| DEFINITION FEATURES | | |
|---------------------|-------------|-------------|-------------|-------------|-------------|-------------|-------------|
| VARIABLE MICROINSTRUCTION | YES | YES | YES | YES | NO | YES | YES |
| WIDTH (MAXIMUM) | (128) | (64) | (256) | (96) | (128) | (96) | (128) |
| DEFAULT FIELD VALUES | YES | YES | YES | YES | YES | YES | YES |
| DON'T CARE FIELDS | YES | NO | YES | NO | NO | NO | N.A. |
| SYMBOLIC COMPOSITION | YES | YES | YES | NO | YES | YES | YES |
| SUB-FIELD DEFINITION | YES | NO | NO | NO | YES | YES | NO |
| SYNTAX | COMPLEX | SIMPLE | LEVELS OF | SIMPLE | MM/KWAD | UTILIZED | HIGH-LEVEL | COMPLEX |
| | RESTRICTIVE | SOPHISTICATION | | ERROR-PRONE | | FLAVOR | | |

| ASSEMBLY FEATURES | | |
|-------------------|-------------|-------------|-------------|-------------|-------------|-------------|-------------|
| MACRO CAPABILITY | NO | YES | YES | NO | NO | NO | YES | YES |
| MULTIPLE FIELDS PER LINE (SEPARATOR) | YES (%) | YES | YES | NO | YES | YES | YES | YES |
| FIELD OVERLAP DETECTION | YES | YES | YES | YES | YES | YES | YES | YES |
| OBJECT CODE LISTING | PARALLEL | INTERLIST | INTERLIST | INTERLIST | NONE | PARALLEL | INTERLIST | PARALLEL |
| SYMBOL CROSS REFERENCE | INTERLIST | | | | | | | |

| POST PROCESSING | | |
|------------------|-------------|-------------|-------------|-------------|-------------|-------------|-------------|
| PROM PROGRAMMING UTILITY | AMPROM | XMAP | NONE | (INTEGRATED) | STMT | RAYROM | MICRO FORMAT | NONE |
| VARIABLE FORMAT | YES | NO | - | YES | YES | YES | YES | - |
| INTERACTIVE | YES | NO | - | YES | NO | YES | NO | - |
| OUTPUT FORMAT | BNF/HX | BNF/HX | - | BNF/H | S/H/S | BNF/HX | BNF/HX | |
| MAPPING FROM DATA | YES | NO | NO | NO | NO | NO | (BUILT-IN) | |
| MICROINSTRUCTION FIELD | YES | NO | NO | NO | NO | NO | YES | NO |
| REDORGANIZATION | | | | | | | | |

Fig. 10. Features of available bit-slice microassemblers.
. interspersing of comments with microstatements
. microprogram origin directive
. directive for formatting of the listing
. error diagnostics
. symbol tables
. cross-reference table of symbols
. no dependence on specific family of bit-slices, but optional support of popular bit-slices

High-level microprogramming languages are not generally suitable for bit-sliced horizontal machines. The compiler of a high-level microprogramming language should be able to detect microoperations that can be done in parallel and also to recognize timing conflicts. This implies that a microprogram written in a high-level language should be rearranged in an optimal way. Microprogram optimization of this sort has been shown to be NP-hard and an intractable problem (see Ref. 24).

In a separate section below we elaborate on one microassembler, the SIGNETICS\(^{24}\) and show how it was applied in a particular case.

2.- Simulators for bit-slices based uP systems. There are basically five design levels of a computer system and hence five levels at which one can simulate it, namely

(a) system's architectural level
(b) functional/behavioral level
(c) logic design level
(d) physical design level
(e) circuit level

Each level of simulation describes the system in a different way, it has different objectives, and it uses different simulation tools and simulation languages.

(a) At the system's architectural level one describes the configuration of a system in terms of processors, memories, channels, peripherals etc. Questions that need to be answered by the simulation are related to the system's performance, cost, reliability, maintainability etc. Examples of specific problems studied at this levels are: job flow through the system, disk I/O queueing, influence of memory interleaving and cache memory on system's performance. Tools are generally PMS\(^{26}\) — Processor-Memory-Switch-languages. Specifically, one may use either a general purpose language, (like FORTRAN, PL/I or ALGOL), or a general purpose simulation language (like GPSS, SIMSCRIPT or SIMULA) or a specialized simulator.

(b) At the functional/behavioral level one describes the system as it is seen from the user's viewpoint. The main objective here is the functional correctness of the system. Tools are generally ISP\(^{26}\) — Instruction Set Processor—type languages and simulators. Depending who is the user this level breaks down to several sublevels. Thus for a bit-slices based micro-
programmable machine we distinguish the following two levels

- target machine level
- host machine level

At the target level one simulates the architecture of the target machine as it is seen by a user who programs the machine at that level (usually this is the assembly (machine) language level).

At the host machine level one simulates the host machine architecture as it is seen by the microprogrammer. This is the level of simulation for which the designer of such a machine, or a sophisticated user, is interested in. There are several good reasons for which simulation at the host machine level is very desirable if not a must. These reasons are:

- to "verify" the correctness of the host design, by tracing the execution of microinstructions.
- to be able to write and debug microcode (and target code for new machine) before hardware is ready; and after it is to debug code without needing it, on any time-sharing terminal.
- to check hardware against a non-varying "definition" so as to be able to "certify" it.
- to allow quick assessment of the impact of changes (fixes, enhancements, etc.).
- to provide "living" (dynamic, interactive) documentation which is easier to understand, than static, passive diagrams.
- to allow measurement, evaluation, and identification of bottlenecks.

A number of simulation systems have been used or developed for simulating a bit-sliced system at the host level. One such system uses the GASP-PL/I language, a combined discrete/continuous simulation language, based on PL/I. The system has been used to simulate a PDP-8/A emulator system made out of Intel 3000 series bit-slices.

A second system reported in Ref. 28 uses the SIMULA language. The simulator is in fact part of an integrated "metamicroassembler-simulator" system called MICGEN. The simulator can virtually support any bit-slices (it has been used already for the 8x02 and Am 2901 slices). The simulator employs a three-level hierarchical structure. Level 0 contains the simulator monitor and standard procedures. Level 1 contains the description of the various bit-slices (in the form of process declarations of SIMULA) out of which the system is to be composed. These descriptions are either taken from a library or are supplied (written) by the user. The third level (Level 3) is written by the user and describes the host architecture by generating and linking together processes (bit-slice descriptions) of Level 2.

A third system uses the ISPS simulator and language, which is based on the ISP notation. ISPS has been used to simulate a fast emulating ma-
machine, MICE\textsuperscript{30} made out of ECL M10800 series bit-slices. In section VI we present the main features of the ISPS and show how we used it to simulate MICE.

(c) At the logic design level we distinguish the Register Transfer sublevel and the logic gate sublevel. The main objective at both sublevels is functional correctness too. However, the functional correctness is certified at lower levels of detail compared to the ISP level.

At the Register Transfer (RT) level the system is a collection of registers, operators (like AND, OR, NOT, EQV, XOR, +, -, 2's complement, shift operators etc.), and data paths. Besides functional correctness, crude timing analysis and detailed performance prediction are conducted. Suitable tools are RT description languages and RT simulators\textsuperscript{31,32}.

At the logic gate level the system is a collection of logic primitives such as gates, flip-flops, multiplexers etc., and their interconnections. Besides functional correctness, detailed timing analysis, detection etc. are among the problems answered by the simulation at this level. The available tools are logic simulators which are very dependent on the technology.

(d) At the physical design level the system is described in terms of racks, motherboards, RC boards, etc., into which it is partitioned. The problems attacked at this level concern the partitioning of the system, the board layout, the layout of IC's etc. The description of the system at this level serves as the interface to the manufacturing environment, as well as the system's documentation. Tools at this level range from PC layout and routing to automated drafting systems and design rule verifiers.

(e) At the circuit level the system is described as a collection of voltages, current sources, resistors etc. Among the problems considered at this level, are waveform analysis, thermal analysis, and temperature sensitivity. Tools at this level are circuit analysis program which are also very dependent on the technology.

V. SOFTWARE TOOLS USED IN THE DESIGN OF MICE

As said in the introduction the design of a bit-slices based system requires the mastery of enormous amount of knowledge on hardware, firmware and software, combined with the use of proper design tools. The resulting system is a microprogrammed, if not a user microprogrammable machine with a unique host architecture. This implies that the designer has to develop his own micro-assembly language in which he will write his microprograms. The most proper tools towards this is a meta-microassembler because it can be easily and quickly customized to the host machine architecture by declaring the format of the microinstruction (fields and subfields). Moreover by using a meta-microassembler with powerful macrofacilities it is possible with some but rewarding effort, to define higher level micro-operations, for multiple field assignment, with high-level language flavour.
A second, entirely optional tool, but of major impact, is a simulator of the host machine for the reasons we gave in section IV.2. Since the design uses high-level building blocks—the bit-slices—there is no need to resort to simulation at a level lower the host machine level, provided that the required timing and fan-in/fan-out rules are carefully observed.

Instead of developing a simulator for a particular design using a general purpose language it is much preferable to use a suitable architecture description ISP language with well debugged features and powerful simulation run-time commands that allow the user to trace, set breakpoints, preset and interrogate register and memory values in an interactive way.

The Signetics meta-microassembler and the Carnegie Mellon University's ISPS simulator are the two tools that have been successfully and efficiently employed in the design of a bit-slices based system, the MICE. The machine is made cut of the Motorola 10800 series ECL bit-slices. MICE is a fast user microprogrammable processor that emulates the PDP-11 fixed point instruction set (without memory management and multiple levels of interrupts). The machine is used for running high energy physics data-reduction algorithms at speed of roughly three times that of the PDP-11/70 CPU when programmed at the target machine (PDP-11 emulation) level. Fig.11 gives an overview of the computing resources at CERN used for the development of MICE. The microassembler runs on the IBM machine and the ISPS on the PDP-10. The designer using terminal C1 connects either on the IBM or the PDP-10 via the INDEX line-switch exchange. Initially the designer connects to IBM to customize first the Signetics meta-microassembler (microinstruction format and macros definition), and then to write his microprograms in the microprogram language he just defined. The microprograms are then translated into binary by the microassembler and formatted in a way that can be accepted by ISPS simulator of MICE via the CERNET and OMNET networks. They are also formatted in a way that they can be loaded on the Control Store of MICE via the test computer PDP-11/40 for real time testing. The designer can then connect to the PDP-10 to start a simulation session to debug both the hardware and the sent over firmware (microprogram). The user can also via the C2 terminal control the operation of MICE. A number of debugging routines, run on PDP-11/40, facilitate the integration of hardware, firmware, and software on MICE, in real time.

In addition to the above tools a wire-up program run on the CDC machine is used to produce the paper tape that will drive the wire-up machine for wiring the MICE boards.

In the following we review the main features of the Signetics meta-microassembler and the ISPS system and show the way they have been used in the design of MICE.

1.- The Signetics meta-microassembler

It is a general meta tool for building assemblers. It is written in
Fig. 11. Resources used for MICE development

FORTRAN and consists of two parts: the microassembler and the microformatter. Its main features are:

Definition features
- variable microinstruction width (32-4095 bits)
- default field values
- symbolic composition
- syntax with high level flavour
- IF...THEN...ELSE...FI statement (for conditional field assignment)

Assembly features
- nested macro capability - MICROS
- multiple fields, microps, per line (space is used as separator)
- detection of field overlap and multiple field assignment
- object code listing interleaved with source code and error messages
- symbol cross-reference tables

Post processing/Formatting features
- micro formatter for PROM programming
- variable format
- output object code in binary or hex
- built-in mapping PROM data generation

Macro (MICROPS) library provided
- for 8x02, 3000, and 2900 series bit-slices

Lacking features
- message generation (by the programmer)
- correlation between field values (for detecting incompatibilities)

The macro capability (called MICROP) of a meta-microassembler is different to that of ordinary assemblers. In the latter case a macro is used to replace a number of instructions (in the vertical sense), whereas in the former case a MICROP is used to replace a number of microoperations (and hence a number of field assignments) within the same micro-instruction (in the horizontal sense).

Fig.12 lists the meta-language (statements), of the meta-microassembler used in the Definition Section to define the microinstruction format and the user microprogramming language (MICROPS).

The INSTRUCTION statement defines the width in bits of the micro-instruction. The FIELD statement defines the name, the width, and the default value of each field in the microinstruction. The FORMAT statement allows the definition of subfields within fields.

The MICROP statement in the form given in Fig.12 allows one or more field value assignments to be associated with a symbolic name.

The INTRINSIC statement specifies the name of a definition file which the user wants to be automatically called from the library and embedded in the Definition Section.

Fig.13 is part of the microinstruction format definition of MICE. Note that comments are freely interspersed within inverted commas. Observe that the microinstruction width is defined to be 150 bits though 128 only do actually exist. The extra 22 bits have been added in order to overcome the lacking features of the meta-microassembler given earlier. Thus an ERRORF field of 10 bits is used to output an error code whenever an error condition is detected by the definition body, other that those conditions which are automatically detected by the meta-microassembler itself. This way we overcome the first lacking feature. The SEMAPHORF field of 12 bits is used to signal indirectly incompatibilities between values in different fields that are set to these values by different MICROPS. Two incompatible MICROPS try to set the same semaphore bit something that is detected automatically by the microassembler at assembly time (see assembly features).

Fig.14 gives the statements used at the Program Section. It is seen that a microinstruction is a collection of field value assignments
Fig. 12. Meta-language for the Definition Section.

Fig. 13. Microinstruction format definition of MICE
(Fieldname=value) and names of MICROPs (microname arguments). As we shall see soon below the Signetics meta-microassembler allows the definition of MICROPs with arguments.

Fig.15 lists statements that are used either at the Definition Section or the Program Section.

The true power of the Signetics meta-microassembler is due to the extended features of it that are listed in Fig.16. Of these, the IF statement allows conditional field value assignment and testing of the validity of arguments. The extended MICROP form allows recursive argument definition in a MICROP a feature that makes the system a powerful meta tool with high level flavour.

Using the above features a powerful set of MICROPs may be defined. This set constitutes the user microprogram language. Fig.17 presents some typical instances of MICROPs. Fig.18 gives as an example the definition body of the ADD MICROP, in order the reader to get a feeling of the complexity of the definition. Note that the IF statements are used to test the validity of the arguments.

As a final example Fig.19 gives the microprogram (in symbolic form and microassembly) that implements the PDP-11 target instruction

```
ADD X(R_s) (R_d)
```

The reader is assumed familiar with the PDP-11 and he should consult also the block diagram of MICE given in Ref. 30. The microprogram consists of four parts:
1st part: Source operand fetch (2 microinstructions)
2nd part: Destination operand fetch (1 microinstruction)
3rd part: Execute (ADD) (1 microinstruction)
4th part: Result write back (1 microinstruction)

Note that the 4th part is overlapped with the decoding of the next target instruction.

2.- The ISPS System and its use in MICE

ISPS is an implementation of the ISP notation introduced initially by Bell and Newell (see Ref.26) as a formalism to describe the programming level in a hierarchy of digital systems descriptions. Although ISPS is oriented towards the description of Instruction Set Processors, it contains a fair number of constructs and extention to ISP which can be used to describe a large class of Register Transfer Systems (digital computers are a subset of the latter, namely, those systems that fetch, decode, and execute instructions). ISPS covers a wider area of application than any other hardware description language. Thus, besides simulation and synthesis of hardware, other applications based on ISPS are: software generation (assembler genera-
\begin{verbatim}
.PRG PROGRAM Name WIDTH numberofbits LENGTH numberofwords;
     Fieldname = value 
  .Label : { Micropname arguments } ...;
  .ORG address ;
  .END ;
\end{verbatim}

Fig. 14. Statements for the Program Section

\begin{verbatim}
  .Label : ... EQU value ;
  .Label : ... SET value ;
  OFF
  .LIST { SOURCE }
     { SOURCE OBJECT }
  .OBJECT { ON }
     { OFF }
  .SPACE value ;
  .EJECT ;
  .TITLE 'TEXT';
\end{verbatim}

Fig. 15. Statements for either Definition or Program Section

1) Expressions : operand operator operand
    operators : SHR, SHL/,+-, /EQ, NE, GT, GE, LT, LE, NOT, AND, OR, XOR, .
2) The DCL (DECLARE) statement
    absolutolocation : label:... DCL value, width ;
3) The IF statement
    IF booleanvalue THEN operand ELSE operand FI ;
4) Extended MICROS
    MICROP Micropname ASSIGN Operand... DEFAULT Operand... ;

   . Micropname = Arg
   . Micropname = (Arg,...,Arg)
   . Micropname = (Arg,...,Arg)Arg,...,Arg

Recursive Argument Definition : Arg = Arg(Arg,...,Arg)

\begin{verbatim}
   Fieldname = value
  .Operand = { IF ... THEN ... ELSE ... FI }
            Micropname
\end{verbatim}

Fig. 16. Extensions of the Signetics meta-microassembler
READ,WRITE
LIR(B),LIR(E) ! read,write to/from the target memory
! load instruction register at the beginning (B),
! at the end (E) of the micro-cycle
LIR,GTR
FORK
! load,gate T register
dest
EXEC
! enable forking
! for destination operand fetch
RF(REG, R4) DEST ! for execute sequence
! read and/or write from Register file
BIB ! P-bus to I-bus
IDR ! I-bus to Data Register
BDR ! P-bus to Data Register
PNT3(0C,2)PB,MAR ! P=2 to PC and MAR
LXADR(OB) ! load XADR from C-bus
EXADR ! enable XADR to address the target memory
LATCH IB ! latch I-bus in latch L of ALU
ADD(OP1, OP2)IB,ACC! add OP1 and OP2 and put the result in I-bus and Accumulator.
SPS(SAPL) ! set program status CC's for add operations
CONDITION(BEQ) ! test for equal to zero condition
INC ! increment micro program counter by 1
JMP L1 ! jump to the micro address L1

Fig.17. Typical MICROPS developed for MICE

MICROP ADD( #OP1 = L, #OP2 = A) #DEST1 = IB, #DEST2 = 0, #DEST3 = 0 ASSIGN
IF ( ( #OP1 EQ L) AND ( #OP2 EQ A) ) OR ( ( #OP1 EQ A) AND ( #OP2 EQ L) )
THEN ALU1F = FAADD MALU1D ( #DEST1, #DEST2, #DEST3 ) ELSE
IF ( ( #OP1 EQ ACC) AND ( #OP2 EQ L) ) OR ( ( #OP1 EQ L) AND ( #OP2 EQ ACC) )
THEN IF ( #DEST1 NE ACC) AND ( #DEST2 NE ACC) AND ( #DEST3 NE ACC)
 THEN ALU1F = FAADD MALU1D ( #DEST1, #DEST2, #DEST3 ) ELSE
 ALU1F = FACCP TO MALU1D ( #DEST1, #DEST2, #DEST3 ) 1 FI ELSE
 IF ( ( #OP1 EQ L) AND ( #OP2 EQ C) ) OR ( ( #OP1 EQ C) AND ( #OP2 EQ L) )
 THEN IF ( #DEST1 NE ACC) AND ( #DEST2 NE ACC) AND ( #DEST3 NE ACC)
 THEN ALU1F = FAADD MALU1D ( #DEST1, #DEST2, #DEST3 ) ELSE
 ALU1F = FACCP MALU1D ( #DEST1, #DEST2, #DEST3 ) 1 FI ELSE
 ERRORF = ERMAAD FI FI FI DEFAULT SPS(SAPL);

Fig.18. Definition of the ADD MICROPS

tors, compiler-compilers and symbolic execution), program verification,
arithmetic evaluation and certification, design automation, and Reliability
and fault analysis.

Fig.20 shows the steps involved in the simulation of a microprogrammed
machine at the host machine level. First the ISPS description of the host
machine is parsed and put in an intermediate representation called Global
Data Base. This intermediate format of the host machine is processed by a
program (GDBRTM) which produces code for an ideal Register Transfer Machine.
When this code is combined with ISPS simulator (which is simply a software
implementation of RTM), the Simulator of our host machine is produced. The
simulator is run using some test microprogram and data to obtain results,
tracing information and counts of activities (like counting the number of
TARGET PDP-11 INSTRUCTION : ADD X(Rs) (Rd)

1. T → IB
   Rs → OB
   X+Rs → MAR
   READ
   P-bus → T

2. READ
   P-bus → IB
   IB → RSS
   PC+2 → PC,MAR
   FORK

3. Rd → OB
   OB → XADR
   READ(XADR)
   P-bus → IB
   IB → RDD
   FORK

4. RSS → OB
   RDD → IB
   IB → L
   ADD,SPS
   IB → RDD,DR
   LIR(E),LTR
   READ
   FORK

5. LA
   DR → P-bus
   WRITE
   PC+2 → PC,MAR
   FORK

1. GTR RF(RS,R)OB ADD3(OB,IB)MAR TRDNO READ LTR JMP $+1;
2. READ BIB RF(RSS, W) BIB PNT3(PC, 2)PC, MAR CONDITION(1) BRC FORK DEST;
3. LONGCYCLE RF(RD, R)OB LXADR(OB) EXADR READ BIB RF(RDD, W)IB CONDITION (1) BRC FORK EXEC;
4. RF(RSS, R)OB RF(RDD, RW)IB LATCH IB ADD(A, L)IB IDR SPS(SAPL) LIR(E)
   READ LTR JMP CTYPEB;
5. CTYPEB : LA EXADR RDB WRITE PNT3(PC+2)PC, MAR CONDITION(1) BRC FORK;

Fig.19. Micro-sequence for the ADD X(Rs)(Rd) PDP-11 instruction
Fig. 20. Host machine Simulation using ISPS.
times a register is read or written, or counting the number of times a unit has been activated. The simulator proceeds interactively using appropriate commands of the ISPS simulator.

2.1. The ISPS notation

ISPS describes the interface (i.e., external structure) and the behaviour 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 behaviour aspects of the unit are described by procedures which specify the sequence of transformations on the information in the carriers.

In the simplest case an entity is simply a carrier (a bus, a register, a memory, etc.), completely specified by its bit and word dimensions, like

\texttt{ir\textbackslash instruction.register <15:0>} ! instruction register
\texttt{tm\textbackslash target.memory [4095:0]<15:0>} ! 4k 16-bit words of target memory
\texttt{wcs[1023:0]<127:0>} ! 1k 128-bit words of Writable Control Store

In another case an entity may have only behavior like the following abridged copy of the behavioral description of the Micro-sequencer Control (MC) unit of MICE:

\texttt{mc( ):=} ! micro sequencer
\begin{verbatim}
BEGIN
 DECODE icf \Rightarrow ! microinstruction control field
 BEGIN
 0\textbackslash jsr := ! jump to subroutine
  BEGIN
   push(\texttt{cr0}) \texttt{NEXT} ! save return address
   \texttt{cr0 = na( )}
  END,

  "F\textbackslash rtn := ! return from subroutine
  BEGIN
   \texttt{cr0 = pop( )}; ! pop return address
   \texttt{cr1 = cr1 + cin1( )}
  END

 END
\end{verbatim}

The \texttt{mc( )} procedure computes the address of the next microinstruction depending upon on fields of the current microinstruction, external branch conditions, etc. The \texttt{icf} field of a microinstruction is used as an "operation code" which decoded and its value is used to select one of a number of alternative register transfer sequences. This selection mechanism is implemented in ISPS with the \texttt{DECODE} operation. If the value of \texttt{icf} is 0, the micro-instruction address register, \texttt{cr0}, is pushed into a small stack, in-
ternal to the mc unit and a new next address is computed by the na( ) procedure and loaded into cr0. If the value of icf is 16 (hex F), the return address is popped from the stack into cr0 while another register, cr1, is incremented by the value computed by procedure cin1. The \"\" operator is used to introduce aliases for constants (as in the example) or for identifiers, as in the previous examples. The two operations above (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 in ISPS is defined as "process" concurrency and no assumptions are made about synchronization of operations. Thus, conflicting use of source and destination carriers, as in "A = 5 ; B = A" can yield unpredictable results.

In the general case, an entity consists both of an interface (carriers) 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. The following ISPS description refers to an arithmetic unit that consists of four carriers. Three of these carriers (a,b, and f) are used to receive and retain the input operands and the function code. The fourth carrier has, by convention, the same name as the unit (alu).

\texttt{alu(a<15:0>,b<15:0>,f(function<3:0>)<15:0>:=}
\texttt{BEGIN}
\texttt{DECODE f =>}
\texttt{BEGIN}
\texttt{0\add := alu = a + b,}
\texttt{1\sub := alu = a - b,}
\texttt{3\not.a := alu = NOT a,}
\texttt{4\not.b := alu = NOT b,}
\texttt{...}
\texttt{14\zero := alu = 0,}
\texttt{15\ones := alu = "FFFF}
\texttt{END}
\texttt{END,}

An entity may be used to specify the behavior of other entities. For example the statement

\texttt{... NEXT x = alu(y,z,0) NEXT ...}

invokes the arithmetic unit above to perform the operations \( x = y + z \). Initially the input carriers of the alu are loaded with the contents of the registers y and z and the constant 0. The alu is then invoked and when the add operation has been completed, the result appears in the output carrier \texttt{alu<15:0>}. Finally the output carrier is transfered to the register x.
The output carrier of an entity may also be accessed without activating it. For example the statement

\[ zcc = alu \text{ EQ} 0 \]

uses the output of the alu unit above without activating it, to set accordingly the zero bit of the condition code register.

In the above examples of the alu entity the input carriers a, b and \( f \) retain their values, loaded at the beginning, throughout the invocation of the alu. If, however, the alu had been defined using the \text{REF} keyword to prefix the names of the carriers \( a \) and \( b \),

\[
\text{alu(REF a<15:0>,REF b<15:0>,f\backslash function<3:0>)<15:0> :=}
\]

\[
\text{BEGIN.........................END,}
\]

then the carriers \( a \) and \( b \) do not have any retention properties. Instead when the alu unit is invoked then it operates directly on the registers \( y \) and \( z \); the registers are connected to the alu throughout its activation.

When an entity is defined as a \text{PROCESS}, like

\[
\text{PROCESS alu(a<15:0>,b<15:0>,f\backslash function<3:0>)<15:0> :=}
\]

\[
\text{BEGIN.........................END}
\]

then it can operate independently of the calling entity, and, moreover, the caller does not have to wait for the completion of the activation.

\text{ISPS, in addition to the standard programming-like return mechanisms, allows the called procedure to determine where control is to return, bypassing the caller. Three control operations, LEAVE, RESTART, and RESUME allow the writer of an ISPS description to model the modes of termination of an entity activation.}

Besides the above, ISPS provides register transfer operators \((<=,=)\), ordinary operators \((\text{AND, OR, NOT, EQV, XOR, } +, -, \text{ MINUS})\), shift operators, field concatenation \((\oplus)\), relational operators \((\text{EQL, NEQ, LSS, LEQ, GEQ, CTR, TST})\), and conditional statements \((\text{IF conditional expression} \Rightarrow \text{action})\).

2.2. The ISPS Simulator

The ISPS simulator supports tracing \((\text{TRACE})\), break pointing \((\text{BREAK, ABREAK, WBREAK})\), setting and interrogation of values \((\text{SETVALUE, VALUE})\), starting, and stopping of simulation \((\text{START, CONTINUE, EXIT, INITIATE, KILL})\), input and output of values to/from a terminal or a file (automatically or via a command, command files may be invoked), and gathering and accessing statistics \((\text{CRT, SETCRT, OUTCRT})\).

The reader may refer to Refs. 34 and 35 for a detail exposition of the ISPS language and simulator.

2.3. Simulation of MICE using the ISPS

ISPS has been used to simulate MICE at the host machine level. Though there are various-levels of description of a host machine, the one adopted
can be characterized as "virtual slice" simulation, because while it describes and simulates the operation of the individual bit-slices, out of which MICE is composed, not all the details were included, but only those portions of the slices that are used in the host. 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 a host machine of the complexity of MICE. Two of the most important problems are:
(a) order of firing of activities and granularity of time
(b) propagation of changes.

(a) Since each of the virtual slices is generally active during one or more phases of the five phase clock cycle, a fundamental problem is of how to simulate these many parallel activities, and at what level of time granularity. In conventional instruction cycles for a vertical microprogrammed 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. In the case of MICE the problem appears 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 ship, 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 a micro-instruction with forty-nine fields, many of them encoded.

One method to reduce this combinational complexity is to find a single canonical order for firing the slices and simulating both inter and intra-slice source-destination register transfers. This attempt has failed because different combinations of individual functions often require different timing sequences, which are too many to be individually analyzed. Furthermore, there appears 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 adopted 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 Fig.21). These timing are based on worst case propagation delays and therefore the simulation does not mimic the hardware
MAIN ucycle () {PROCESS} :=
BEGIN
  init() NEXT  ! System initialisation
  pr = wcs [csa()] NEXT  ! Load pipeline reg.on
  : phase 1 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,

Fig.21. Clock Phases

Fig.22. System Parallel Processes
faithfully, a point that should not concern the microprogrammer. Register transfer cycles are resolved by introducing explicit master and slave copies of each of these registers which can act both as sources and destinations 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.

Note that the firing order problem would appear even if one uses 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. However, the parallel processor facility of ISPS has been used to model independent activities going on in our machine, e.g. MICE CPU, and the two DMA processes (see Fig.22). 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, bolt-clock, etc.).

(b) A problem closely related to that of deciding firing order is 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". 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

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 occurs 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 destinations, 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:

\[ x = y + \text{bus()} \]

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 inefficient, 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, one avoids needlessly propagating changes to entities which are not in turn used as sources.

The same considerations are 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 following example describes one of the multiplexors driving the OBus (MX5) of MICE.

\[
\text{mx5()} <15:0> :=
\]

BEGIN
\[
\text{DECODE smx5f =>}    \]

! 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,

The example indicates that the rule implies a regression from the desired output values back towards its sources, through potentially many levels of logic. The regression stops whenever a source with retention properties is encountered (e.g. a register). It is assumed that a source without retention properties 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 are used. The first one is 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 and are treated as latches afterwards. Any attempt to access one of these carriers before the right time is therefore detected and the proper diagnostic message is issued.

The second feature of the language that was used in the description is 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 the other 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.
VI. SUMMARY

We have given an overview of the tools for development of microprocessor based systems. We have concentrated on development tools specifically for bit-slices based systems. We have elaborated on two categories of tools, meta-microassemblers and simulators for the host machine level. Going through two particular tools, the Signetics meta-microassemblers and the ISPS simulation system, we have shown how they have been used in the design process of a fast ECL bit-slices based machine. We have discussed the problems the designer faces when he tries to use these tools.

The main point which these lectures have tried to emphasize is that working with bit-slices involves sophisticated, integrated hardware, firmware, and software design that can be successfully accomplished only by using proper design tools.

* * *

REFERENCES

16) B.E.Gladstone, Comparing Microcomputer Development System Capabilities,
19) P.Hughes, Factoring in software costs avoids red ink in microprocessor projects, Electronics, p.126, (June 22, 1978).