CERN-Data Handling Division
DD/79/7
October 1979

CERN LIBRARIES, GENEVA

CM-P00059819

ARCHITECTURAL CONSIDERATIONS FOR A MICROPROGRAMMABLE
EMULATING ENGINE USING BIT-SLICES

C. Halatsis*)

A. van Dam⁺)

J. Joosten

M. Letheren

(To be submitted to the 7th International Symposium on Computer
Architecture, La Baule, France. May 6-8, 1980)

*) permanent address, NRC Democritos, Athens

⁺) permanent address, Brown University, Providence

ARCHITECTURAL CONSIDERATIONS FOR A MICROPROGRAMMABLE
EMULATING ENGINE USING BIT-SLICES

*) +)

C. Halatsis, A. van Dam, J. Joosten, M. Letheren
CERN, Geneva, Switzerland.

*) permanent address, NRC Democritos, Athens, Greece

+) permanent address, Brown University, Providence, USA

ABSTRACT

This paper describes architectural considerations which led to the design of a fast programmable processor made from ECL bit-slices. The processor will be used as an on-line data filtering engine for high energy physics experiments. Unlike prior designs of such engines, the processor supports both user (horizontal) microcode and emulation of the PDP-11 fixed point instruction set (without memory management and multiple interrupt levels). In addition to an overview of the techniques used to achieve an execution speed of roughly three times that of the PDP-11/70 CPU, strengths and weaknesses of bit-slices are discussed, as are the use of a Signetics meta assembler and the ISPS Architecture simulation systems (described in the companion paper "Simulation of a Horizontal Bit-Slice Processor using the ISPS Architecture Facility" - van Dam et al *).

1. Introduction

The purpose of this paper is to present the design of a MICroprogrammable filtering "Engine" (MICE) built from ECL bit-slices. The engine will run high energy physics data-reduction algorithms at substantially higher speeds than can be achieved with standard minis, yet preserve the latter's full programmability.

* under preparation

1.1 Motivation for on-line data reduction in high energy physics

Research in high energy physics is directed towards investigation of ever rarer phenomena at ever higher energies. This has been made possible by improvements in particle accelerator technology, the development of new forms of electronic particle detectors and the introduction of computers for data acquisition and control in experiments.

As a result of the pursuit of rare particle interactions the typical experiment now collects vast quantities of numeric data, often more than 90% of which is useless "background", i.e. non-interesting particle interactions. Traditionally the experimental data is recorded on magnetic tape and the "good" events (particle interactions) are filtered from the often overwhelming background by resource consuming off-line data analysis on the largest available number crunching facilities.

In order to get a feeling for the problem, consider a typical experiment which generates as many as 10^6 to 10^7 consecutive particle interactions per second. A fast hardwired first level "trigger" reduces this rate to about 10^3 per sec by looking for relatively simple time and space correlations characteristic of the type of interaction desired, typically with a few nanoseconds time resolution. When the trigger logic signals the detection of an event, of the order of 10^3 16-bit data words should be read out from the detectors into the Data Acquisition Computer (DAC). A typical time for a block transfer of 1000 words from the detectors to the Data Acquisition Computer's memory, including software overhead, is 4 msec. During the read-out the trigger is disabled ("dead") and all potential triggers occurring during this period are therefore lost. Figure 1 shows, for particle interactions occurring randomly in time, the fraction of potential triggers which can be collected by the DAC as a function of the "dead-time" imposed by the read-out of the event. For our typical experiment only 20% of the potentially triggerable interactions can be recorded; hence the physicist is unable to take full advantage of the expensive resources at his disposal. In addition perhaps less than 1% of the events actually collected are of the "good" type, and several hundred hours of CDC7600 equivalent CPU time will be required to filter them out.

Clearly, if we introduce a second level trigger which can reject background events in a time much shorter than the event read-out time and thereby reduce deadtime for rejected events (the majority) we would gain in two ways:

- 1) the expensive accelerator and detector resources would be more efficiently used because fewer good trigger events are lost due to dead-time for read-out of unwanted events.
- 2) The amount of expensive off-line number crunching would be reduced because of greatly improved fractions of good events in the recorded sample.

Assume that a 2nd level trigger with a decision time of $100\ \mu\text{s}$ reduced the rate of block transfer requests to the data acquisition computer by a factor 10. The average dead time per event would then be $100\ \mu\text{s} + 0.1 \times 4\text{ms} = 0.5\text{ms}$. In this case we would be able to record 67% of the available good events at the previous rate of 10×3 while reducing by a factor 2 the volume of data to be analysed off-line. Therefore we have an approximate $67:20 \times 2:1 \sim 7:1$ "improvement" ratio.

1.2 Rationale for a microprogrammable emulation engine

The key idea of the on-line filtering concept is to give the greatest throughput of "good" events to the Data Acquisition Computer by applying progressively more restrictive but more time-consuming event selection procedures (Ref. 1). Typical is the two level filter, where the first level is provided by the simple, fast hardwired "primary" trigger. The second level is provided by slower hardwired or programmable logic performing more sophisticated filtering algorithms on the already reduced event stream.

The ideal on-line filter would be both extremely fast for manipulation of low precision (16-bit) integer data and flexible to allow easy adaptation to the changing requirements of a physics experiment.

In the study of physical phenomena, the important parameter is the number of events collected; in principle the rate at which they are collected is immaterial. However, the faster it is, the less

time experimental resources are tied up, and therefore the less expensive the experiment. On the other hand, it is in the nature of high energy experimentation that trigger algorithms evolve constantly for ever greater selectivity during the many months of data collection and experimentation, so that flexibility and ease of change are often of greater importance than speed. If speed criteria dominate, the filter would be hard-wired (Ref. 2) whilst if flexibility and ease of implementation are more important, software on standard minis (Ref. 3) or specially designed processors can be used (Ref. 4).

The MICE is meant to provide a good compromise between the speed of hardwired logic and the flexibility of (mini) software. Unlike prior designs of microprogrammable filters (Ref. 4) our engine has been designed explicitly both to interface easily to an experiment and be an efficient host for microprogrammed emulation of a standard mini. Thus the user can write low level micro-code for time critical algorithms and program non time-critical tasks at a conveniently high level, making use of standard software development tools with which he is probably already familiar. We chose to emulate the PDP-11 architecture because of the symmetry of the instruction set, its memory mapped I/O and its wide acceptance within the high energy physics community. Our PDP-11 fixed point instruction set emulator runs at about 3 times the speed of Digital Equipment Corporation's PDP-11/70 CPU operating out of cache memory with 100% cache hits.

We found that the speed ratio between hardwired algorithms and software on a PDP-11/70 is typically 50:1. MICE will allow the following speed/flexibility tradeoffs: when running on the high level but relatively slow emulated PDP-11, algorithms will run about 3 times faster than on a PDP-11/70. When coded in low level but faster and still flexible microcode they should run 5 to 10 faster than on a PDP-11/70, 5 to 10 times slower than in hardware, and when running on special purpose hardwired units controlled by MICE, even closer to that of strictly hardwired processors.

MICE supports only those architectural features which are necessary for on-line filtering - hence there is no floating point hardware, no memory management and only a simple one level interrupt facility. These simplifications reduced the complexity of the host design, and allowed us to concentrate design effort on an efficient overlap scheme for target instruction fetch, decode and execute, and

some other special features for fast execution of both target code and micro code.

2. Integration of the filtering engine into the experiment

The engine is not intended to replace the data acquisition and control computer of the experiment, but it should prefilter the data stream from the experiment. The engine is therefore controlled by the Data Acquisition Computer (DAC). There are many ways in which the data acquisition system can be structured in order to use an engine (or multiple engines) to prefilter the data stream. However all such systems require the fastest possible rate of data transfer between the experiment and the filter engine. Figure 2 shows a typical system organisation using a single engine operating on a single data stream. The engine performs the filter algorithm on a subset of the event data made available to it via a fast readout system (Ref.5) connected via a DMA controller to the engine's 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 out the complete event from the detector.

Clearly the filter engine will increase the throughput of "good" events only if the time it requires to read out and process the subset of data is shorter than the time required to read out the complete event via the slow readout. Other configurations are possible where multiple engines work in parallel on multiple data streams in order to increase throughput (Ref.1).

2.1 Interfaces

Figure 3 is a block diagram of the engine and its interfaces. The engine is interfaced to the DAC via the CAMAC-interface module, which uses the sub-address and function codes of a single CAMAC station (Ref.6). Some of the CAMAC-interface commands available to the DAC are:

- 1) Loading and dumping of programs and data to/from writeable control store (WCS) or target memory (TM).
- 2) Commands which control the engine's clock - e.g. hold the clock, suspend the CPU, run the clock n cycles and then hold it (for hardware debugging).
- 3) Commands which emulate the PDP-11 console - i.e. halt, load program counter, start, continue.
- 4) A command which causes diagnostic micro-code in MICE to dump the internal registers of the CPU into a register in the CAMAC-interface at the end of every micro-cycle, from where they can be transferred to the DAC.
- 5) Commands which handle interrupts to or from the engine, e.g. load MICE interrupt register, read LAM* register, load LAM mask register, test LAMs, etc.

The DAC loads or dumps WCS or TM by block transfers over the "external" bus (E-bus). The E-bus is also used by the DMA-interface for fast block transfers of data from the experiment to the TM.

The CPU normally accesses TM via the fast ECL "processor" bus (P-bus). The P-bus is interfaced to the U-bus - a slower asynchronous TTL bus which is a subset of the DEC "UNIBUS". The U-bus, used for input/output to slower devices, is a logical extension of the P-bus and operates by holding off the clock until the U-bus cycle is complete. Hence all devices on the P and U-buses can be accessed in a single micro-cycle irrespective of their speed. For design simplicity, neither the P-bus nor the U-bus support vectored interrupts or bus mastership arbitration. Hence the U-bus can only support simple, non-DMA UNIBUS devices, such as the teletype or papertape reader, where interrupts are replaced by software status tests.

3. Overview of Design Objectives and Tradeoffs

The key design objectives were :

- 1) Simplicity of design and short implementation time.
- 2) High execution speed
- 3) Simplicity for the programmer
- 4) Low cost.

*CAMAC interrupt - see Ref.6).

Design and implementation time was considered the most critical factor, even more so than raw execution speed. Low design and manufacturing cost to allow replication was given lowest priority. Thus simplicity of design was emphasised, leading us to select bit slices as the highest level building blocks. Though slower than discrete ECL logic, we felt the use of the full functionality of the Motorola 10800 ECL bit slice family would give us a modular, flexible and relatively simple design at still acceptable speed. Prior experience with this family (e.g. Ref. 7) for design of simple processors demonstrated the functional adequacy and tractability of the M10800 family.

Simplicity and speed dominating cost led us to impose no artificial constraints on micro-instruction word width - in our classically horizontal host we do not "pattern" or "program" a micro-instruction field for multiple uses as a function of mode bits, nor do we generally encode fields that control the bit slices. Only the ALU slice is steered by external decoding logic (ROM). This direct control also makes the design more understandable for the microprogrammer in that the host is very regular, with private groups of fields controlling each slice/functional unit. As explained in section 5, however, the microprogrammer need not be concerned with a 49 field, 112 bit micro-instruction.

Simplicity for the eventual programmer also led us to design the host to make it easily accessible from the target level but also to keep the need for having to microprogram to a minimum. For example, we implement a "jump to micro-address" instruction, and also a REPEAT n instruction which repeats the next target instruction n times. Typically the repeated target instruction will be a 2 operand arithmetic instruction with at least one operand in auto-increment mode. Repeating it provides, at the target level, a simple fast vector instruction and eliminates the overhead of loop control (i.e. decrement counter, test for zero, and branch back). For example a read/modify/write arithmetic vector operation on target memory can be performed in three micro-cycles per operand (micro-cycle time is 105 ns). Note that the host supports repeated execution of micro-instructions and micro-subroutines, which allows vector operations to be implemented at the micro-code level at even faster rates - e.g. search of a target memory vector in one micro-cycle per element.

Finally, we realized from the start that a significant investment in software could probably help decrease the total hardware design and implementation time and in any case would especially be useful for subsequent use of the system. Thus we traded off hardware design time for finding and adapting to our purposes a good, general purpose microassembler, and for writing a complete simulator of the host and its interfaces. For the latter purpose we used Carnegie Mellon University's ISPS, a register transfer machine simulator (see section 5). Probably half the total manhours on the project were in fact spent on customising and using the microassembler and ISPS tools; we are confident the investment was more than worth it.

4. Implementation Strategies and Host Architecture

The goals of simplicity and short implementation time made it mandatory to use where possible standard techniques for achieving speed rather than developing innovative but complex solutions. This resulted in a fairly standard use of all the slices (see Fig. 4). We have the ALU (MC10800), the Register File (MC10806), the Micro-sequence Controller (MC10801) and the Target Memory-Interface (MC10803). A four phase clock with a cycle time of 105 ns (and in some cases 90 ns) is provided by the Programmable Clock circuit (MC10802). Additional logic is used to enhance flexibility and/or speed - see section 5.1.

In order to meet the design goals we tried to :

- 1) keep the 40ns target memory as busy as possible prefetching instructions and operands.
- 2) to do as much pipelining as possible without needing the complexity of residual control (having the state of the machine during one micro-instruction depend on prior instruction states.
- 3) to have all micro-instructions take the same time by computing the address of the next micro-instruction and fetching it always during the current cycle, even for conditional branches.

We tried to balance the architecture so that each functional unit (and memory) could be kept busy nearly every micro-instruction and that most paths/computations would take roughly the same amount of time. The cost for this logical and physical simplicity is a somewhat (25%) longer cycle for the simplest arithmetic instruction, which is considered acceptable.

There is no arbitration logic for resolving memory access conflicts between the P-bus and the E-bus (Fig. 3); instead the E-bus may access WCS or TM by suspending the CPU, i.e. holding the CPU clock and switching the memory ports to the E-bus. This solution allowed us to keep memory accesses from the P-bus (the majority in our applications) very fast (~40 ns). It was the same design goal which led us to limit the electrical load on the P-bus by connecting I/O devices to the U-bus.

To achieve balance and homogeneity we tried to reduce data and control flow to common patterns. For example, the way in which Source and Destination operands are fetched is the same and independent of the opcode of the target instruction. Effectively, execution microcode deals with prepared (prefetched) operands from a "standard place", without knowledge of (i.e. dependency on) the addressing modes used. To this end we prefetch operands into the Register File into which ALU results are also stored (and possibly written out to Target Memory on the next cycle). A target register-to-register operation takes place in one micro cycle (Read/Modify/Write of the Register File); because of the decoupling of operand fetching and target instruction execution the same micro-instruction can also be used to execute a target memory to register instruction.

4.1 The Micro-instruction Pipeline Scheme

The micro-instruction pipeline scheme is shown in Figure 5. During the current micro-cycle we always calculate the address NA_{i+1} of the next micro-instruction (in the Micro Sequencer or Fork logic as described below) and fetch it into the micro-instruction pipeline register ($\mu Fi+1$, STR PR) ready for execution in the next cycle. The Micro Sequencer is therefore strobed at the midpoint of the cycle (STR MC). We chose this scheme because of its simplicity and the resulting ease of writing and understanding the microcode.

To be able to calculate the next micro-address and fetch the corresponding micro-instruction in one cycle, it was necessary to add external table lookup logic (i.e., mapping ROMs) in order to (conditionally) "fork" on :

- 1) the Source and Destination operand addressing modes,
- 2) the target instruction op code,
- 3) interrupts and internal hardware status signals.

During the micro cycle in which the fork 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. The fork operation is enabled during the last micro cycle of every target instruction execution in order to decode the prefetched next target instruction. The resulting fork is either to a source operand fetch sequence, or to a destination operand fetch sequence, or to the target instruction execution sequence. Internal status signals can effectively interrupt the normal microprogram sequence after any micro-cycle via hardware which forces a fork independently of the the microprogram control (examples are bus timeout, odd address errors, CAMAC-interface control lines for activating diagnostic microcode, PDP-11 console emulating control signals such as HALT, START and CONTINUE).

4.2 Target Instruction Pipeline Scheme

The target memory pipeline scheme prefetches next instructions, operands or data for operand address calculation according to one of two standard schemes shown below. The scheme used depends on whether a result is written back to Target Memory or not (Ij is the instruction fetched from Target Memory location j). See Figure 6.

Scheme A - no Target Memory Writeback (1 cycle)

cycle ti:

involving:

Execute Ij, no TM write.	Register File and ALU.
Decode Ij+1 and fetch micro-instruction to execute Ij+1.	Decoding ROM, fork logic, WCS, and micro-sequencer.
Fetch Ij+2 (or Operand of Ij+1).	Memory Interface, Target Memory
Prepare Fetch address Ij+3.	Memory Interface (effective address calculation).

Scheme B - Target Memory Writeback (2 cycles)

cycle t_i :

Execute I_j

Fetch I_{j+2}

cycle t_{i+1} :

Write result of I_j in TM.

Decode I_{j+1} and fetch micro-instruction to execute I_{j+1} .

Prepare Fetch Address I_{j+3} .

This degree of overlapping is supported by the 3 instruction-pipeline registers T,IR,A and the use of the Memory Interface ALU for effective address calculations in parallel with the main ALU operation. Most normal PDP-11 one or two operand instructions take only a single execution micro cycle, after any required source and destination fetch cycles. Thus the target memory is used nearly every cycle for reading or writing. Only if the T register is already full or if an extra cycle is needed to compute the next target fetch address (auto-decrement, certain cases of successful branches) is the target memory not active. (Note that this simple pipelining scheme makes certain operations which we consider "unstructured" in our environment either illegal (e.g., operating on PC) or legal but erroneously executed (calculating an operand in instruction I_j for storing as part of I_{j+1} or I_{j+2}).

4.3 Special features.

As explained in section 3 we introduced new target instructions in order to facilitate the task of speeding up the time-critical parts of programs written in a higher level language (e.g. assembler or PL11 (Ref. 8).

Thus we have introduced the target instruction "jump to micro-code" (JMC) which provides a general mechanism to migrate functions from assembly language code to user microcode.

Another special feature offers the possibility of vectorising single instruction loops from the target level by preceeding the instruction to be vectorised with the new instruction "repeat next target instruction" (RPT). The RPT instruction sets the MICE into target instruction repeat mode, where the incrementing of the target PC is inhibited and the repeat counter is decremented at each instruction decoding fork until it becomes zero, where upon the MICE leaves the target instruction repeat mode. Repeated execution of an instruction using auto-increment/decrement addressing modes to step through arrays in memory removes the overhead of loop control, e.g:

```
MOV COUNT,R2
AGAIN:  ADD R0,(R1)+
        SOB R2,AGAIN
```

can be replaced by:

```
RPT COUNT
ADD R0,(R1)+
```

The first requires 6 micro-cycles per element of the array, whilst the second requires only 3 micro-cycles per element. Addressing modes which alter the PC are illegal in repeat mode and are trapped in the instruction decoding logic.

The Micro-sequencer provides powerful facilities for repeating a single micro-instruction or micro-subroutine. These features can be used to efficiently implement micro-coded loops. The single micro-instruction loop can be conditionally exited by wire-ANDing the appropriate status signals to bits in the next micro-instruction address to force a conditional multi-way jump. This standard technique is available in the MICE hardware.

As an example of the potential speeds achievable by microprogramming in MICE, consider the following assembly language program which scans an array in memory for an element smaller than the value in R1:

```
MOV COUNT,R2
LOOP:  CMP (R0)+,R1
      BLT OUT
      SOB R2,LOOP
      .
      .
      .
OUT:   NOP
      .
      .
```

This is executed on the PDP-11/70 in a time of 1.65 microsecs per loop pass. On MICE the same algorithm has been executed in a single micro-instruction loop using the repeat micro-instruction mode, the micro-branch address modification technique and the target memory instruction pipeline to overlap:

- 1) The conditional micro-branch on compare result of element j
- 2) The compare of element j+1 (in ALU)
- 3) The fetching of element j+2 (into T register)
- 4) The autoincrement of R0 (in Memory Interface ALU)

Thus the search is performed at a speed of 105 ns per array element, a factor 16 times faster than in assembler on the PDP-11/70.

Another special feature, which proved extremely useful during debugging of the hardware was the built-in diagnostic micro-code. When the "diagnose" request signal is set by the DAC the MICE microprogram is interrupted. The "diagnose" micro-interrupt service micro-code saves the interrupted value of CR0 for its return address and then proceeds to dump all internal registers to a standard register in the CAMAC-interface, from where they are retrieved by the DAC. The MICE clock is held off until the DAC collects the data from the CAMAC-interface register. On return from the "diagnose" service micro-sequence the next micro-instruction of the interrupted micro-program is executed before the diagnose request causes the next micro-interrupt. In this way we were able to step through the microprogram from the DAC console, getting a dump of the MICE registers after every micro-cycle. This feature required minimal extra hardware and could equally well be used to gather statistics on algorithms running in the MICE (at both target and host machine levels).

It is also foreseen to connect special purpose units to the extended buffered internal CPU buses. These hardwired units would replace time consuming micro-code required for certain functions, e.g. multiplication, internal product, etc.

4.4 An example of CPU operation

As a simple example of target instruction pipelining and emulation (Fig.7), assume that during t_i we decode $I_{j+1} = \text{ADD } X(R0), (R1)$; this is an add with source mode 6 (Register Indexed), destination mode 1 (Register deferred). Three discrete phases will be required: Source operand fetch, Destination operand fetch, and Execution. At the end of t_i , the decoding of I_{j+1} will have enabled the Source operand fetch output of the decoding ROM which resulted in the fetching of the first micro-instruction in the mode 6 source operand fetch micro-sequence (two micro instructions). Simultaneously we prefetched the word at $j+2$, which is the index X in this case, and incremented the Program Counter (Prepare Fetch I_{j+3}). During t_{i+1} this first micro-instruction causes the effective address to be calculated from index X and source register $R0$ in the Memory Interface unit. At the same time the word at $j+3$ is prefetched (either destination operand or next instruction) to "pay back" the word previously fetched and "borrowed" as (source) operand rather than next instruction.

In t_{i+2} the second Source fetch micro-instruction causes the operand to be fetched into a standard register in the Register File (RFs), and PC is again incremented to prepare the next fetch. Since this is the last source operand micro instruction, it also enables the destination fetch micro-sequence output from the decoding ROM so that simultaneously the first (and only) Destination micro-instruction is accessed. At t_{i+3} it fetches the Destination operand (into Rfd) using $R1$ as an address (see section 5.1), and enables the execution sequence output of the decoding ROM to cause the ADD micro-instruction to be fetched. This is executed in t_{i+4} , in parallel with the fetch of I_{j+4} , and at t_{i+5} the result is written back in TM while PC is incremented for the next fetch, and I_{j+3} is decoded by enabling the decoding ROM. Thus five micro-cycles of 105 ns are required to complete the instruction. Note that, when all memory reads are from cache, the PDP-11/70 requires 2.1 micro-seconds to execute the same instruction (4 times slower than our machine).

5. Findings

Most of our observations probably will not surprise those experienced in design with bit slices, but we have not seen them clearly stated in the trade- and formal literature.

5.1 Slices are not always sufficient for their intended purpose

The 10800 family of slices is a rich family, but the individual slices have both too much (i.e., underutilized) functionality and too little orthogonality/regularity/symmetry. We used less than half of the functions, data paths, and registers of the sequencer, ALU and memory interface slices. At the same time, they lack symmetry - the ability to transfer source X to destination Y does not imply the existence of the inverse transfer although it makes sense and was needed. This forced us to use external data paths (via multiplexors and gates) and registers to supply the symmetry. While we could not fully use the available functionality, we paid for it nonetheless in terms of chip complexity (and therefore speed).

It certainly would help if the recent trend to design symmetric, orthogonal CPU's (e.g. DEC's VAX) would also appear in the design of future slices; where technology limitations make that impossible, better documentation on how and why the subset was chosen and how to work with it would be useful.

Another reason we found it necessary to add external SSI and MSI "glue" was to bypass slice features when they would take too much time. Two main examples of this both concern the bypassing of internal address registers, one normally used to address the control store (CRO in the micro-sequencer), the other normally used to address the target memory (MAR in the Memory Interface). As mentioned above, CRO is bypassed with the decoding ROM for target instruction decoding (forking) because testing target address mode bits with the sequencer and then loading CRO would take too many cycles. Similarly, MAR is bypassed, for example, for the common operand address modes register deferred, auto-increment, and auto-increment deferred. To use MAR we would have to fetch the target operand address from a register in the Register File and then strobe it in MAR at the end of the cycle. Instead we load the external address register with the operand address from the Register

File 30ns into the cycle, immediately address the Target Memory, and bring the operand through the Memory Interface to the Register File to store it at the end of the same cycle. During this same cycle auto-increment of the target address may take place in the ALU.

Because of the external logic there are many ways to accomplish a given function - multiple independent data paths, registers, and even ALU's. The resultant complexity and potential for writing dangerous (unpredictable) microcode must be masked from the ordinary microprogrammer by supplying him with high level macros which present him with a more structured, more vertical (encoded) architecture - see below.

5.2 Hardware Design means Software Tools

Software tools and development are a (the ?) major portion of a bit slice hardware design. Because the host must be microprogrammed, one obviously needs a powerful, convenient micro-assembler. We also used a simulation system to check out the host design. A micro-assembler is commonly customized to the host architecture by declaring the layout of the micro-instruction (fields, subfields, default values of those subfields) to a meta assembler, and then using its macro facilities to declare "instructions" which set the fields belonging to each of the functional units in the host architecture. We found the Signetics Meta Assembler written in FORTRAN entirely usable, but spent several man/months writing a large set of (nested) macros (called microps by Signetics) to control assignments to groups of fields at a level well above that of controlling 49 independent fields in our horizontal instruction. These microps do extensive error checking (using IF THEN ELSE parameter checking) to make sure field values are within range, and are mutually consistent within the definition of the microprogrammer's host, which is essentially a higher level, suitably restricted version of the actual hardware. Considerable effort went into error diagnostics, an area in which the Signetics assembler was somewhat weak; for example it provides no easy way to correlate field values in separate microps for the same micro instruction. More general macro facilities (e.g. parameter type checking) would also have been useful. A typical set of microps, to route two operands (RS and RD) from the register-file (RF) to the ALU, add them and restore the result in RD, (see Fig.4 for the data paths) is as follows:

RF(RS,R) OB RF(RD,RW) IB LATCH IB ADD(A,L) IB ;
a construction which hides the details of gate, multiplexor, and function code settings for the two slices and their glue, as well as the time-multiplexed use of the I-bus, and prevents inconsistent/conflicting use of the buses.

The simulation of the host was the other major software project and although entirely optional, of major impact. We decided to simulate the host for a variety of reasons, which are becoming increasingly accepted in the hardware design community:

- 1) To "verify the correctness of the host design, by tracing its execution of micro-instructions.
- 2) To be able to write and debug microcode (e.g. the target emulator) before the hardware was ready, and indeed, even afterwards; debugging on any 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 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 (e.g., FORTRAN, PASCAL, SIMULA) we picked Carnegie Mellon University's ISPS (Ref. 9) because it is specifically designed for architecture simulation, and has many useful, well debugged features and tools. It consists of a description language in which data transfers and data operations are specified, a simulator for the resultant register transfer machine, and an interactive command language for tracing, setting breakpoints, loading and interrogating register and memory values, etc. Multiple concurrent processes are supported to mirror hardware parallelism, and timing and measurement facilities are also available. Our use of ISPS is described in a companion paper (Van Dam et al.); once we learned to use ISPS properly, the CPU description and simulation was completed in three man-weeks, the interface and I/O logic in another man-month.

Regretably, ISPS was not installed on our PDP-10 until after most of the logical design was completed; if it had been available earlier, much tedious hand simulation of the microcode and the architecture could have been avoided. As is, our learning investment in ISPS has been more than justified in terms of finding many (mostly small) design errors before the construction of the hardware and allowing easy testing of additions to the basic system. It currently runs exclusively on the PDP-10, unfortunately, but makes only modest demands on the computer since the simulation is dominated by man/machine interaction at the console.

6. Conclusion

Working with bit slices involves sophisticated, integrated hardware, firmware, and software design, using high level, complex hardware building blocks. It cannot be done the same way designers work with fixed instruction set micro-processors, which requires primarily ordinary assembly language programming knowledge and far less hardware experience. The design task is accomplished faster and easier than working with lower level (MSI) building blocks and yields a fast flexible result. A significant investment in software tools for a proper design methodology made it possible to design, build and test our filtering engine in about one year, using a host architecture simulator on a PDP-10, a micro-assembler/PROM loader on a 370/168, a wire-wrap program on a CDC 7600, a PDP 11/40 to drive the hardware, and two networks to tie all these software ingredients together and make them easily accessible from a single terminal.

7. Acknowledgements

The authors are indebted to J.Antonioz-Blanc, S.Brobecker and A.van Praag for their assistance at various stages of the project. We are grateful to Dr. M.Barbacci, of Carnegie Mellon University, for the installation of the ISPS system and invaluable advice and assistance in the writing of the ISPS description of the MICE. It is a pleasure to thank Dr. P.Zanella and C.Verkerk for their continued support and encouragement.

8. References

1. C.Verkerk, "On-line Filtering", Proc. 1978 CERN School of Computing, CERN 78-13, pp. 65-87.
2. See for example:
M.Hansroul, C.Verkerk and P.Zanella, "Hardware Processors for pattern recognition tasks in wire chamber data", Int.Conf. on Instrumentation for High Energy Physics, Frascati, 8-12 May, 1973, pp. 497-499.
A.Fucci, M.F.Letheren, F.H.Sumner, C.Verkerk and W.G.Vree, "Design of Hardware Processors for use in High Energy Physics", CERN DD/74/32. Also published in Proc. of 1974 Computer Systems and Technology Conference, London, IEE Conference Publication 121, 1974, pp. 42-48.
3. A.Norton and M.Scire, CERN, Geneva - Private communication. See also Reference 1.
4. See for example:
T. Lingjaerde, "A fast Microprogrammable Processor", CERN DD/75/17.
P.F.Kunz, "The LASS Hardware Processor", Nuclear Instrum. Methods 135 (1976), p. 435.
T.F.Droege, I.Gaines, K.J.Turner, "The M7 - a high speed digital processor for second level trigger selections" IEEE Transactions on Nuclear Science, Vol. NS-25, (1978), p. 698.
C.MacLean, G.MacPherson and P.Wilde, "Special Purpose Computing Hardware for Processing Data from Experiments", Proc. 2nd. ISPRA Nuclear Electronics Symposium, Stresa, May 20-23, 1975, pp. 307-313.
5. See for example:
J.B.Lindsay, C.Millerin, J.C.Tarle, H.Verweij and H.Wendler, "A fast and flexible data acquisition system for multiwire proportional chambers and other detectors" Nuclear Instrum. Methods, 156 (1978) pp. 329-333.
6. "CAMAC - A Modular Instrumentation System for Data Handling", EUR 4100e, Revised version 1972, available from Commission of the European Communities, D.G. XIII - C.I.D. 29, rue Aldringen. Luxembourg.
CAMAC is also identically specified by IEEE Standard 583.
7. B.Pfister and S.Horvath, Abteilung fur industrielle Forschung des Institutes fur Technische Physik an der E.T.H. Zurich, Private Communication.
8. R.D.Russell, edited by T.C.Streater, "PL-11: A Programming

Language for the DEC PDP-11 Computer with addendum (version 5).", CERN 74-24 Rev.

9. M.R.Barbacci, G.E.Barnes, R.G.Cattell and D.P.Siewiorek, "The ISPS Computer Description Language", Computer Science Department, Technical Report, Carnegie Mellon University, August 1977.

M.R.Barbacci and A.W.Nagle, "An ISPS Simulator", Computer Science Department, Technical Report, Carnegie Mellon University, March 1978.

THE FRACTION (F) OF POTENTIAL TRIGGERS WHICH CAN BE RECORDED IS

$$F = \frac{1}{1 + R\tau}$$

WHERE R IS THE POTENTIAL TRIGGER RATE AND τ IS THE DEADTIME IMPOSED BY THE READOUT OF THE EVENT

FIG.1 DEADTIME LOSSES

FIG. 2 - SYSTEM CONFIGURATION

FIG. 3 — MICE CONFIGURATION

FIG. 4 - MICE CPU

FIG. 5 - MICRO-INSTRUCTION PIPELINE SCHEME

Notes: Source and Destination operand fetching is not shown.

Ej : Execute target instruction j
 Dfj : Decode instruction j and fetch micro-instruction
 to execute target instruction j
 Fj : Fetch target instruction j
 PFj : Prepare fetch address of target instruction j
 Wj : Write result of instruction j in target memory

FIGURE 6 - TARGET INSTRUCTION PIPELINE SCHEME

FIGURE 7 - TYPICAL MICROPROGRAM SEQUENCE