The 801 Minicomputer - An Overview

I. Introduction

This paper attempts to describe the goals, characteristics, and present status of the 801 Experimental Minicomputer project in the Research Division at Yorktown Heights. It is intended as a companion to the documentation which describes the architecture of the system because this latter will not enable a reader to understand what we hope to accomplish, and what is new in our approach. As we develop performance and cost data we will publish them as separate papers.

Let us start with some history. The S/360 (and later the S/370) architecture has provided a compatible interface to software across a product line which varies greatly in cost and performance. (See the excellent analysis of these parameters in the GTA report entitled "Comparative Hardware and Performance Overview" by Manfred Schwengler of the Boeblingen Laboratory.)

While keeping the architecture constant the cost/performance curve is traversed by varying the technologies used and amount of hardware (e.g. number of circuits, size of cache). For a given technology there are practical extrema of performance achievable.

The hardware is made usable for applications by systems programs which offer a higher level of interface by means of control program facilities and compilers from high level languages. This resulting family of systems thus provides a cost/performance curve of "useful" work measured in parameters like thruput, number of transactions per second, response time (as well as less easily measurable parameters like reliability, ease of programming, modifiability).

Compatibility across a product line and, today, compatibility with S/370 (and with MVS, DOS/VS, COBOL, PL/1, IMS, CICS, etc.) is in itself an extremely valuable system characteristic. (Given IBM and users' investment in programs, data, education, etc., it is clearly true today that for most of the product line compatability is, in fact, an essential characteristic.) The question we ask in the 801 project is: What is the price, in cost and in performance which this S/370 compatibility implies? If the price turns out to be very great, (say between one and two orders of magnitude) we ask if there are potential market areas for a product which offers greatly improved cost/performance instead of, or in addition to, a compatible S/370-OS. These are clearly important questions for IBM to investigate even if we decide not to act on the results.

It should be apparent to anyone who has thought about the problem that the answer to the first question is very difficult to determine. Technologies change continuously; CPU performance is useful only if it is coupled with a suitable storage subsystem and I/O configuration and these often overwhelm CPU cost savings; products have differing requirements for checking logic, recoverability in software, maintenance aids, etc.; cost is often dependent on volume, which in turn depends on the potential market areas of the product.

The 801 project was motivated by a number of ideas (to be described in the next section) which potentially can result in this very large cost/performance improvement. We sought a vehicle for implementing these ideas in a way that would make their value apparent while requiring only a modest (about 20 people) project effort. Several constraints are clear:

- We do not have the resources to build more than one prototype. Thus all product line evaluations will have to be paper studies based upon a single implementation.
- We do not have the resources (or the training) to do chip design through EDS. Moreover, since the architecture is new, we need to remain able to make changes quickly and late in the project life. Thus cost and performance of our prototype will not be identical to that of a product even in the same technology. Nevertheless the actual performance of our prototype needs to be impressive in itself - otherwise we might as well write a simulator.

- We do not have the resources to provide all of the OS software facilities, or even to implement a complete language like PL/1 or COBOL. Thus some hand translation of programs will be necessary to make comparisons with S/370 applications. These simplifications, however, will be useful in understanding whether, in fact, all the PL/1/MVS facilities are really needed for those applications for which the 801 is suitable.

Therefore we decided to direct the 801 project initially toward development of the following system:

- A hardware prototype is being built out of vendor logic using only out-of-catalogue dual-in-line packaged chips on a wire-wrapped vendor board. Some checking is incorporated into the prototype, but all cost estimates will allow for additional circuits required for an IBM product. The performance of this technology (Emitter Coupled Logic) is such that an 801 machine cycle (under 10 logic stages) can be completed easily in 60 nanoseconds. (Other technology implementations will be extrapolated from this.) The number of circuits required to build the 801 puts it roughly in the large mini area of machine complexity.

- The storage subsystem for our prototype matches the performance of the CPU and thus is relatively high in cost. Storage subsystem costs for other 801 performance levels must be developed individually.

- The 801 supports standard IBM I/O devices. However, S/370 channels and control units are replaced by 801 adapters and minimal control units. The costs for the 801 units are far less than for S/370 counterparts, but this is offset by a greater burden on the CPU to do this work. How much CPU time will be required to do this function remains to be measured, but must then be subtracted from the computing power available for application programs.

- 801 control program facilities will be provided at a simpler, more basic level than that offered in any S/370 programming system. They will thus be far smaller, cheaper and faster. The object is that they will be adequate for the applications considered as possible 801 markets. Demonstrating this requires actually building and running these applications.

- The 801 will support a large subset of PL/1 initially and, shortly thereafter, FORTRAN and COBOL. The language facilities will be rich enough so that large whole programs can be recompiled from S/370 with only modest hand-translation (of I/O, for instance). The compiler will incorporate many state-of-the-art optimization techniques; early indications are that it will produce better code than any existing language processor.

This 801 system prototype will attach to a S/370 host. Applications will be developed in the following areas:

- S/370 subsystems (e.g. VTAM/NCP, IMS/3830, Channel/3272).

This will determine the resulting system price performance to be obtained by offloading host functions and replacing channels and control units with an 801. It is attractive only if the 801 can do this job at large factors better MIPS/$ than the host/channel/CU, and if the software facilities support this function well.

- Conventional mini-computer applications (e.g. general purpose interactive computing, real-time applications, on-line data base systems).

Cost and performance for these applications will be extrapolated for various paper design 801 models. (A machine design simulator will enable us to estimate storage and I/O as well as CPU performance for these models.) When the prototype and the simulator are running the applications, we will be able to measure these characteristics quite closely, and a determination can then be made about the 801 product potentials.

II. Technical Innovations

It might be argued that the basic 801 question has already been answered. Minicomputers have long demonstrated the ability to offer large factors of cost/performance improvement over S/370's at the penalty of incompatibility. Moreover, general-purpose software for some of these systems has proven quite reasonable for building many non-trivial applications.

But there are technical innovations in the 801 which lead us to believe that we can significantly exceed the cost/performance improvements of any existing or planned minis. This section describes some of the more important of these innovations.

A) Performance of Register-Register Instructions

Except for microprocessors (with very primitive instruction sets) and super-computers like the 195 or CRAY-I (with no cost constraints), general purpose CPU's are designed with an underlying micro-code processor interpreting micro-programs which reside in control store.

The history of this kind of machine organization can be traced to the early introduction of complex instructions into hardware. This was very advantageous in machines like the 701, 704, and 7090 because of an internal speed to memory access ratio of better than 10:1. Fetching one complex instruction was clearly faster than fetching a sequence of instructions from a subroutine or macro which performed the same function. Early implementations of these complex instructions were in hardware, but it was soon discovered that the I-fetch savings could accrue as well by a subroutine, provided this subroutine was stored in a high speed memory (i.e. control store). Thus, the fundamental machine organization changed to one of a microprocessor interpreting programs stored in ROS or writeable control store. Unfortunately, this generally implied that even the simple instructions were interpreted and so took several machine cycles to execute. The "micro" instruction set used to code these subroutines reflected the limited nature of programs which would execute them, and the relatively small amount of code to be generated. These instruction sets are generally hard to program, and come with little development support (like compilers).

In the last generation of CPU's this I-fetch advantage for complex instructions has in fact disappeared with the advent of high speed memory for caches and instruction buffers. But microcode machine designs persist, largely due to a pernicious circular effect, as follows:

If a subroutine can be fetched as fast as a micro-routine, one might simply use a primitive subset of the CPU instruction set as the underlying architecture on which to program the complex instructions (as has often been done in low end systems). In fact, if one looks at a micro-instruction set it is generally quite easy to find equivalent functions in a subset of the CPU architecture.

But this subset is taken from an architecture which was defined with a microcode implementation in mind and is therefore not easily implemented to execute in one machine cycle, like micro-instructions, especially on lower priced systems. Therefore the complex instruction subroutines are implemented on a lower level architecture and the machine design becomes one of a conventional micro-code variety. This in turn makes the primitive functions slower since they must suffer the burden of interpretation. In fact, on most machines running or planned, even primitive functions like boolean ops,

binary arithmetic, shifts, compares, etc., take several machine cycles to execute.

With the relaxation of compatibility constraints the 801 takes a fundamentally different approach. We define an instruction set which is primitive enough so that, with a number of circuits no larger than many conventional microprocessors, all register-register instructions can execute in at most one machine cycle, and this cycle is at most 10 logic stages. But we define this instruction set by a joint group of programmers and engineers so that it is regular, easy to use, and a superior target for compilers.

For the past ten years or so many instruction traces have been taken for widely different applications. They consistently show a large skew toward frequent execution of a primitive set of instructions. In the mixes used to describe the performance of the next generation of IBM CPU's in both Endicott (E) and POK (H) forty instructions generally exceed the 90 percentile of execution frequency. Reducing the I-time overhead from these instructions by hard-wiring improves the CPU performance considerably.

Surprisingly, even the execution of complex functions is improved. This is because high level interpretation of S/370-like complex instructions introduces overhead through generality. The 801 compiler can recognize special cases and compile faster code sequences. The difference is the same as that of an interpreter versus a compiler. In the case of conventional CPU's the interpreter is written in micro-code and so is somewhat faster, but is generally still significantly slower than compiled code. Consider some examples:

- "Move Character" in S/370 must be prepared to cope with overlap of operands, operands which exceed authorized bounds, operands of different sizes and different alignments, operands ranging in size from one to 256 bytes, etc. Yet frequency traces of use of this instruction show a mean size of 8-10 bytes, little overlap, etc. The micro-interpreter of this instruction *must* check at every execution for these cases even if it is sophisticated enough to do something special about them. An 801 compiler can often easily recognize, for example, that two strings are being concatenated, or that they are small enough so that the move loop should be expanded.

- While effective addresses in S/370 are defined to require two additions (X+B+D), most uses of RX instructions set either X or D equal to zero. Since the compiler normally is the program that generates addressing sequences it simply calls for one addition most of the time. (In fact, if the operand requested is still in a register as the result of a previous instruction, no address computation code or fetch is executed at all.)

- Multiplication (and division) is often by a constant. A CPU has no choice but to execute general arithmetic sequences for all instructions (fixed, floating, or decimal). A compiler can choose a representation of the constant which minimizes the required adds and shifts. This then tends to make these code sequences approach the performance of hardware multiply and divide.

Continuing this advantage, complex functions which are even higher in level than, say, S/370 instructions, see an even more remarkable performance improvement. It is as though each such function were custom-microprogrammed instead of being programmed using the general high-level primitives of a conventional instruction set. For example, a square root subroutine can be programmed on the 801 to run about as fast as two floating point add's.

And finally, when what the user wants is a very high level interpretive interface (such as APL), the 801 offers the performance of a microcoded implementation with the programming ease of a conventional software interface.

The major technical exposures with the 801 approach are the probability that a sequence of code will be found in high speed storage when needed, and the efficiency of the object code compared to microprograms. Note that an 801 requires no control store (the normal cache is used, and cache-replacement strategies apply) and no local store (normal CPU registers are used and compiler register allocation strategies apply). Thus cost savings are clear - the potential problems are whether these resources, thus shared, can be scheduled without significant performance degradation. This is one of the major areas of investigation for the project.

B) Performance of Register-Storage Instructions

We have seen so far that there is a large potential performance advantage in the 801 approach to operations when the operands are in registers. In fact these instructions execute in at most one machine cycle. However, available instruction mixes consistently indicate that, for the majority of instructions, it is necessary to fetch an operand from storage, or to store a result into storage, or to get a new instruction out of sequence (i.e. branch).

Even when the operands, or new instructions, are found in the cache, and even when the cache access time is commensurate with a machine cycle, it still must take more than one machine cycle to compute a storage address, (or test on a branch condition) and then access the cache. In the 801 prototype the cache access time equals one machine cycle, so when the required word is in the cache these instructions take at most two machine cycles.

Sophisticated (and, thus, expensive) machines use "pipelining" techniques to search ahead in the instruction stream for things that they can safely do while the cache is being accessed. Once again this must be done interpretively each time a sequence of code is being executed. But in fact an optimal sequence of executions, once found, is the same optimal sequence whenever the code executes, and is thus a good candidate for a compile-time activity. The 801 CPU then does not reorder code sequences, or look ahead to schedule dynamically. But it provides a set of primitives so that, when a code sequence has been ordered properly, cache activity is overlapped with CPU execution. These primitives are quite straightforward:

- On Store, the CPU transfers a word to the cache, with its destination address, in one machine cycle. Then, while the cache is storing this word, it proceeds to the next instruction. Thus all Stores are effectively overlapped and take one machine cycle.

- On Load, the CPU computes the effective address and instructs the cache in one machine cycle. It then locks the register which is the target of the Load and attempts to execute the next instruction. If this instruction does not need the locked register it is executed, and again the cache access is overlapped.

(Note that these two facilities enable the CPU to execute ahead, not just one instruction, but as many as it can before interlocking. Thus, in some cases, even cache misses can be effectively overlapped.)

- On Branch, there is, for every type of branch instruction, an alternate form called Branch and Execute. When executing this form the CPU computes the branch target and instructs the cache. Then, while the cache is fetching the new instruction stream into the prefetch buffer, the CPU executes the instruction immediately following the branch instruction. Thus, where the compiler can find an instruction before the branch which does not affect the branch instruction, it generates a Branch and Execute instruction followed by this safe instruction, and the cache access is again overlapped. Our experience has been that these safe instructions are generally easily found (e.g. the increment instruction of a DO loop).

An example may be helpful here. Suppose the compiler is given:

```
A=B+C+D;
GO TO L;
```

to compile.

A straightforward compilation would produce:

```
L  R1, B    Load B into Register 1
L  R2, C    Load C into Register 2
A  R1, R2   Add R1+R2 and store into R1
L  R3, D    Load D into Register 3
A  R1, R3   Add R1+R3 and store into R1
ST R1, A    Store the result into A
B  L        Branch to L
```

But note that, with this sequence:

```
A R1, R2
```

cannot proceed until R2 has been loaded,

```
A R1, R3
```

cannot proceed until R3 has been loaded,

```
B L
```

takes two machine cycles.

Thus the 7 instructions take 10 machine cycles to execute. But if the compiler changes the order of some instructions (and changes the Branch into a Branch and Execute) it can produce this code:

```
L  R1, B
L  R2 C
L  R3 D
A  R1, R2
A  R1, R3
BX L
ST R1, A
```

which now takes only 7 machine cycles.

Our early experience with code sequences indicates that, on average, an instruction takes about 1.2 machine cycles which, for our prototype, yields about 14 MIPS out of cache.

C) Storage Subsystem Performance

The performance of systems whose CPU's are as fast as the 801 out of cache is heavily dependent on the storage subsystem performance. In our prototype the cache is Snipe (60 n-sec access) and our backing store is Riesling (about 10 times as slow on average). Thus innovations in the storage subsystem approach have good payoff in total system performance.

We seek improvements in two areas:

- improving the Cache Hit Ratio (i.e. the percentage of storage references which are found in the cache),
- improving the time to access a line from the backing store when the cache is missed.

Consider the first. We observe that quite often accesses to the backing store are unnecessary for correct execution; they occur because the hardware cannot guess the intent of the software. Generally this unnecessary overhead falls into two classes:

- The program wants a block of new storage. This can occur when a procedure is called and needs temporary (i.e. AUTOMATIC) storage, when the First Level Interrupt Handler needs a Register Save Area, when an Access Method needs a buffer, when a

program issues a GETMAIN request, etc. What is similar about all of these cases is that the program does not care about the old contents of the storage; it just wants some space. And yet all current (or planned) storage subsystems will, on first reference to such storage, fetch the old line from the backing store to the cache. It will do this because this first reference will be at most to one word in the line (since that is the unit of access between CPU and cache) and the subsystem cannot tell that subsequent requests will not ask for other words in the line before they are newly updated.

- The program no longer needs a block of storage, even though its contents have been modified. This can occur on Return from a procedure when the temporary storage is freed, when a buffer is freed, generally when a program issues FREEMAIN, etc. No current (or planned) storage subsystem can determine that such store-backs of modified lines are not necessary.

The 801 approach to this is consistent with its other strategies. Namely, the hardware provides primitives (in the form of instructions) by which software can give such information to the hardware. In particular, two instructions are defined:

Set Data Cache Line

and

Invalidate Data Cache Line

which are issued by the control program, and generated by the compiler for application programs. These instructions then ensure that such unnecessary backing store accesses are not made. In fact, since the temporary storage needed by a procedure is managed in a stack, and since even supervisor calls in our system are CALL'ed, the backing store will not be accessed for a dispatched process' data unless this data is persistent (i.e. STATIC) or the stack depth gets large compared to the cache size. (Frequent process switching will, of course, reduce this advantage.) Thus the backing store begins to play a role which is similar to that played by secondary storage (i.e. a file space and a paging area). This kind of use of a backing store may be more amenable to incorporating intermediate technologies like bubbles and CCD's than are current systems. (Note that with this strategy, the 801 activity on interrupt is no more than that performed by "priority level interrupt" systems at their high end. They too must store their internal registers in high speed memory (i.e. Register Space). The difference is that the 801 does not dedicate high speed memory for this purpose and thus saves cost. The 801 may find a cache miss on redispatch, but that is not in the response-critical part of the path.)

- The CPU, as in most systems, is fetching instructions from the cache in anticipation of their execution. It has a "prefetch buffer" (in our case, of three words) which it attempts to keep filled. Sometimes filling this buffer results in a cache miss and thus initiates a fetch from the backing store. But it may be that the instructions being thus fetched follow a Branch instruction which is already in the buffer but not yet executed. The 801 prefetch mechanism scans op codes and inhibits such unnecessary backing store fetches. (In fact, while scanning op codes for this purpose it also recognizes and eliminates NO OP's, which thus take zero execution time in the 801.)

Now consider the second way of improving storage subsystem performance, namely making the backing store access faster. The 801 incorporates two features in this area, the first not unique to our system, the second quite new and significant.

- When a word is required from the storage subsystem (either data or instruction) which results in a cache miss, the backing store is accessed for the required line beginning with the needed word. This word is then sent directly to the CPU, bypassing the cache, and the CPU continues execution while the line is being stored

into the cache. Thus, for instance, to complete a Load instruction which results in a cache miss may take as little as 340 nanoseconds (minus whatever instructions can be overlapped with the Load).

- Because the instruction prefetch mechanism is essentially asynchronous with the data fetch mechanism we have found it advantageous to split the cache into two disjoint parts, one for holding instructions, the other data. This results in an ability to access the backing store by each cache independently and in an overlapped manner. It has several other benefits:

  Fetching an instruction line will never require store back, since instruction cache lines are never modified.

  Each cache part is separately 2-way set associative. Thus some of the benefits of 4-way set associativity can be obtained without the cost.

  Separate cache characteristics (e.g. size, depth, line size) and separate replacement algorithms can be employed, taking advantage of the different access patterns of code versus data. (The 801 prototype begins with similar strategies for both caches. Subsequent versions will incorporate changes as improvements are indicated by our simulator.)

A "split cache" implementation in conventional architectures has serious problems. Since instructions can legally be modified in the data cache and then be branched to, all modifications must be broadcast to the instruction cache, and it must ensure that it invalidates any modified line. But in fact instructions are almost never modified in today's systems. This, then, is another candidate for a function which can be performed much more efficiently in software than repeatedly, on every data modification, in hardware.

In the 801 the data cache does not broadcast changed lines to the instruction cache. Thus these changes will not necessarily be reflected in the next branch to the modified instruction. The 801 provides an instruction, called

Invalidate Instruction Cache Line

which the software must issue to flush the old instruction, and another instruction, called

Store Data Cache Line

to ensure that the modified instruction is reflected into the backing store. (When program modification occurs because of a load from disk, by far the most common case, only the first instruction must be issued.)

Finally, the 801 architecture defines a relocate facility, which we do not plan to implement in the first version of the prototype, which has several important cost/performance innovations:

- Because of our split cache we can naturally support separate virtual memories for instructions and data, thus allowing software strategies in which a single reentrant copy of, say an APL interpreter, executes for many different user areas.

- We do not allow shared pages (which are now less useful because of the previous facility) between virtual memories. (We do, however, provide a convenient facility for switching between data virtual memories.) This restriction provides significant simplifications. Primarily it allows us to run our caches virtual (i.e. the cache is accessed with a virtual address). This results in *no* degradation due to relocate when the line is found in the cache (over 90 percent of the time).

- Conventional relocate systems dedicate

high speed storage for look-aside tables (DLAT's) on Page Tables - storage which is generally the same technology as cache. We access page tables normally through the cache and expect that their high frequency of use will result in an appropriate probability of cache hit. Thus with no additional cost we can approximate the performance of DLAT's. (Our machine design simulator will give us data about the feasibility of this approach. If it proves to be too slow we will consider dedicating part of the cache to this DLAT function.)

D) Alignment

S/370 architecture (and many mini-architectures) do not require that full or half-word operands be aligned on full or half-word boundaries. This introduces complexities (and sometimes performance degradation) in many places in the machine. Every operand may cause two page faults, may need to be checked against two storage protect keys, may cause two cache misses.

The irony of the situation is that good programming style encourages alignment and, in fact, all trace tapes indicate that non-alignment almost never happens. (This is so clear to many that, for instance, the instruction mixes that are used to measure performance of the E machine line assumes 100 percent alignment.)

The 801 has powerful instructions to handle arbitrary bit and character strings even when they are embedded in larger strings with arbitrary alignment. (Bits in the Condition register (called Parity Stack bits) are set to indicate the alignment of these character string operands. Instructions are defined to move bytes depending on the setting of these bits.) But it insists that half, and full-word operands be aligned, and that three byte addresses be stored in the low-order bytes of a full word. Similarly the structure mapping of PL/1 has been modified to provide alignment of elements other than short packed bit strings and single character strings. This approach simplifies the machine at no real loss in function to programmers.

A word should be said here in explanation of the 801 register size (24 bits). We determined early that registers, ALU and Shifter must be wide enough to compute addresses, since this is in fact what binary arithmetic is mostly used for. Mini-architectures that allow 24 or 32 bit addresses but provide only a small subset of operations applicable to such addresses (e.g. no shifts, no compares) will see a severe degradation in performance and usability when they are in fact asked to support data areas greater than 64K. The 801 is a true 16 megabyte address machine. All arithmetic, logic, compare, shift, etc. operations work on 24 bit addresses.

The remaining question is why we did not go to 32 bit registers. Primarily the reason is that a technical case is hard to make for the additional cost. Address arithmetic is already handled. 24 bits are adequate for programming single precision floating point (and 32 bits are inadequate for double precision). What remains is:

- the set of application variables whose range is greater than ± 8 million but less than ± 1 billion.
- the efficiencies of fewer loads and stores for full word or character >2 moves.

In spite of this questionable case it may be advantageous to extend the register size to 32 bits in an 801 product. This will impact very little of the software because the full word alignment constraints already preclude 3 byte packed fields in control blocks. The CPU cost will grow from 7,600 gates to about 10,000 gates. This cost will be assumed in all cost/performance estimates of the 801.

E) Protection

In all systems it is necessary to prevent some programs from accessing some memory areas, or executing some instructions. This

prevention is generally achieved by the following techniques:

- a PSW reflects the Problem State/Supervisor state, and "privileged" instructions are executed only if the machine is in Supervisor state.
- Keys are associated with storage blocks, reflected in cache lines, and matched with keys in the PSW when access or store is attempted.
- Virtual memories provide a gross level of addressing protection.

These facilities are both expensive and inadequate. Their inadequacies are indicated by the large amount of code which is forced to run in key 0 on S/370 (a problem which is being attacked by planned extensions to S/370 architecture). At best they protect a user from other users and prevent him from bringing down the system. Protection facilities which guard against bugs in his own program are often very expensive (e.g. the range check option in PL/1).

The 801 system takes the following position:

Most application programs, and many system programs, are written in high level languages. It is a characteristic of these languages (even PL/S) that programs are inherently almost completely safe, and can be made completely safe with a little effort by the compiler.

For instance,

- Privileged instructions, such as Start I/O, can be generated only for programs whose authority is asserted in installation commands. Thus no run time state checking is ever necessary.
- Branches are accomplished by high level control flow statements where the branch target is always correctly generated by the compiler (e.g. GO TO, DO, IF THEN ELSE, CALL)
- Data references to constants, scalars, and array elements with constant indices are always safe.

References to array elements with expression indices (e.g. A(I+J)) must be checked at run time. To accomplish this efficiently the 801 hardware provides

Compare and Trap

instructions which compare two values and interrupt on unequal, or on high.

PL/1 Pointers remain a protection problem. To resolve it we restrict pointers to offsets in Areas. This, in fact, makes programs more well structured and constrains the function very little. It allows us, however, to use the same Compare and Trap strategy as for arrays.

The result is a set of language processors which, with a very modest run time overhead, and no hardware cost, produce programs which are guaranteed to be safe not only for others, but will check many internal bugs as well. (The extent of this cost will be determined as the compiler implementation proceeds, but it does not appear to be at all significant.) The checking will ensure that all indices of arrays are within the ranges specified for each dimension, that all "based" variables are within the areas in which they are asserted to be found, and that all references to Controlled variables refer to existing allocated versions.

Since the applications program is written in a high level language, the compiler can control invocations of control program services. Thus SVC-type of interrupts are unnecessary. These services are invoked simply by Branch and Link; they use the caller's stack and participate in optimization of the program.

The remaining requirement is that the compilers be efficient enough so that all application programs and most systems

programs can be written in a high level language. We ensure this by:

- employing many novel compiler optimization techniques to yield excellent code (see below),
- subsetting the version of PL/1 used for our systems programmers to that set which is most useful, but which eliminates many of the causes of inefficiencies,
- providing, as Built-In-Functions in this language, the 801 register-register instructions so that, where necessary, the language can be used as a "safe assembly language".

F) Compiler Optimization

Much of the preceding discussion describes a consistent strategy of system design. Namely, it observes that many functions which are normally performed by hardware (at the expense of cost or performance, or both) are in fact suitable for compile-time analysis. This results in great potential efficiencies, comparable to the efficiencies of compiled programs versus interpreted programs. Examples are:

- Special cases of complex functions can be detected at compile time and specific code sequences generated, such as for MVC, XOC, multiply by a constant.
- Code sequences can be scheduled at compile time for maximum overlapped execution thereby eliminating the need for sophisticated pipeline machine organizations.
- Protection can almost always be analyzed at compile time and Compare-and-Trap instructions inserted where this analysis fails.

To make this approach feasible, however, compilers must produce programs which are efficient enough for almost all applications. In addition to providing the "safe assembly language" escape valve described previously, the 801 compiler (and its successors for other languages) incorporates many new or improved optimization techniques which have been invented in this Laboratory, chiefly by John Cocke and Frances Allen. (While results of their application are still sparse, early test programs show remarkable performance and space improvements.) Finally, the 801 compiler takes great care in generating good code for "systems-like" kernels such as bit string handling and logical expressions, areas which are often poorly handled in existing compilers.

The innovative optimization techniques mentioned above act as transformations on an intermediate level of language (IL) between source and machine. At this level the instructions are generally 801 instructions, but the number of registers is as large as necessary. Front-end processors translate source language programs into equivalent programs in IL. (Thus, to produce compilers for other source languages, only new front-end processors are required.)

The optimizations are implemented as functions which transform IL programs into IL programs. Thus any of them can be eliminated, modified, or added, as we get experience about their value. They rely upon a compiler phase which provides control flow and data flow graphs of the program.

The optimizations include:

common subexpression elimination

code movement from more to less frequently executed areas of a program

conversion of multiplies in loops to adds (strength reduction)

reuse of registers (subsumption)

the elimination of code that produces results that are not used elsewhere (dead code elimination)

evaluation at compile time of

expressions whose terms are constant (constant propagation).

All of these techniques have been used in previous compilers, e.g. Fortran H and PL/I optimizer. The 801 compiler produces better results because these optimizations are applied pervasively to a machine language level text. While previous compilers have optimized source language constructs such as (A*B) and the multiplies and adds involved in subscript computations, the 801 compiler also optimizes:

Loads of source variables

Loads of constants

Loads of addresses used to reference parameters, external and controlled variables

Adds of constants to bases, where the displacement to the variable is greater than that provided in the instruction (64K)

Adds of base and index where the instruction has a base, but not an index register.

Loads of descriptors

Code associated with building argument lists.

Traps for range and existence

Intermediate results of code generation, including conversions and string calculations.

For some time there has been speculation that exposing such low level operations would result in greatly optimized code. It also simplifies code generation, by permitting the generation of sloppy or repetitive code, which is systematically improved by optimization. This improves reliability, since code generation has long been recognized as one of the most error prone areas of compilation whereas formal optimization has been very reliable.

There have been two main arguments against exposing low level operations to optimization. First the time to optimize a program has tended to increase exponentially as the program increases in size. Low level text was assumed to imply very long compile times. However the Allen-Cocke interval analysis can be implemented very efficiently. It is likely that optimization will result in no net increase in compile time, as the final compilation phases will have less data to process. The other barrier to low level optimization has been the concern that it could actually be unprofitable to optimize. For example, the load of a variable might be moved out of a loop but if a register could not be reserved for the entire time the variable was "alive" there would be additional stores and loads beyond what would have occurred if the load had not been moved. A good register allocator is required if low level optimization is to be effective.

The register allocator of the 801 compiler allocates registers by coloring. The technique is called "coloring" because it is identical to the "map coloring" problem of graph theory. As long as the symbolic registers can be reduced to the 16 that exist on the 801, coloring yields optimum results for register allocation. Preliminary results indicate that many small procedures, some large ones and the vast majority of inner loops do color in under 16 registers. For procedures that do not color in under 16 registers, a very effective partitioning mechanism has been developed. Registers will have to be saved and restored at partition boundaries but the preliminary evidence is that this will not be frequent.

The use of low level text for optimization combined with register allocation via coloring provides a simple, formal and uniform approach for many aspects of compilation, that have traditionally been handled in a complex, ad hoc way. The result is a simpler compiler that produces better object code.

G) I/O Subsystem

The 801 I/O subsystem is as incompatible with S/370 as is the central CPU architecture. It will, however, support conventional devices and TP protocols. The object is to achieve significant improvements in cost/performance and in flexibility as a consequence of relaxing compatibility requirements with existing channels and control units. The following is a list of some of the major areas of innovation.

- Many systems save cost by executing channel or control unit functions on the CPU engine (e.g. integrated channels, cycle-steal adapters). The 801, since its CPU has such good cost/performance characteristics, pursues this strategy very aggressively. Hardware requirements (in 801 adapters and control units needed to support I/O devices) are less costly than in existing systems. Much of this function can be executed directly on the 801. (The adapter and control unit architecture is flexible enough to allow specific systems to decide not to adopt this strategy but instead to build intelligence outboard.)

- Unlike existing systems, however, the 801 processor is its own "microprocessor". Thus the channel and control unit "emulation" programs are written to run on the same instruction set as all other software. This has two advantages:

  1) All software development tools (e.g. compilers, debugging aids) and control program facilities (e.g. interrupt handler, storage manager) are now available for programming these functions. Changes and new device support are more quickly and cheaply implemented.

  2) Even when they run on the same engine, because they run at different architecture levels, the linkage between software and these emulator programs are awkward and inefficient. For instance, invoking a S/370 integrated channel program requires Start I/O (with CCW's, CSW's and CAW's) and interrupts on return. (When cost/performance of the CPU requires executing the controller function outboard, as NCP on the 3705, this linkage is even more inefficient.) In the 801 I/O programs are invoked synchronously by CALL (Branch and Link at run time) and terminated by RETURN (Branch back). They use the caller's stack for temporaries and are subject to the same optimization as other procedures. Thus we see a progression from stand-alone channel and outboard control units to a CPU whose cost/performance allows these functions to be performed on the CPU engine, and finally to the 801 where these functions are performed on the same instruction set as users' procedures with very efficient linkage between them.

- Conventional control units, channels and some access methods support multiple length blocks on DASD even though the thrust of such devices at all levels of cost/performance is towards fixed blocks of a single standard size. They also provide outboard search logic even though more efficient indexing is possible in software. The 801 at the outset supports the Picollo device with control unit, adapter and software restricted to directly addressed fixed length blocks. This greatly simplifies the resulting system while providing, to the user, the same level of functions.

- The 801 has two busses on which adapters are attached: a high speed parallel bus for programmed I/O, and a

very high speed memory bus for Direct Memory Attachment (DMA) adapters. The former is used for control and status for all devices, and for data as well for low data rate devices. The 801 instructions I/O READ and I/O WRITE synchronously transmit 2 bytes between the CPU's registers and an adapter in less than two machine cycles. (DMA adapters also attach to this parallel bus, for transmitting control and status information.)

To attach control units at great distances and for little cost, connections between adapters and control units are via serial links. (The 801 project is pursuing (in addition to conventional coaxial cable implementation) a laser-driven optical link for this purpose because it has excellent potential cost characteristics at long distances, better noise immunity, better error rates, etc.)

A Switch adapter allows direct serial links between adapters and different control units at different times, thus allowing a great many control units to share few adapters. This strategy of direct one-to-one attachment (via the switch) simplifies the transfer of control sequences. Unlike S/370 chaining, critical transfer time does not depend on the reselection sequence of the channel.

The Direct Memory Attachment adapter is used to send data between the 801 backing store and high-data rate devices such as disks, displays, and S/370's. The memory bus accommodates a data rate of up to 90 megabytes, of which up to 45 megabytes is available to the I/O subsystem. Each DMA can support a data rate up to 5 megabytes.

A special adapter, called the Interrupt adapter, attaches to the parallel bus for transmitting control and status, but also attaches to the CPU via interrupt-raising lines. It is the only adapter that can raise an external interrupt in the CPU. All others attach to the Interrupt adapter and, when they wish to interrupt the CPU, set a bit in an interrupt-pending vector (one bit position per adapter plus others which are associated directly with devices). This vector is AND'ed with an Enable mask vector sent to the adapter by software in the CPU. If there is a resulting logical one, and the CPU is enabled, the CPU is interrupted. (There are two other bit positions in this vector, one for an interval timer, and one for a CPU software-generated interrupt (PCI).) The First-level interrupt handler reads the vector (by IOR instructions) and, since the vector has been ordered by priority of adapter, executes a COUNT LEADING ZEROS instruction which gives it the branch address to the adapter handling code. This ordering is, of course, a software option. The hardware allows any association between bit positions and adapters/devices. (Our First Level Interrupt handler, which stores all registers, sets the interrupted process inactive, turns off its timer and branches to the required adapter handler takes about 50 instructions (i.e. about 3 microseconds), and can be greatly improved where the intent is cycle-stealing (i.e. the interrupted process will continue when the handler completes). Thus the software can perform all the functions of a hardware priority interrupt scheme (and several additional ones) for as many levels as there are adapters with no hardware expense, no dedicated Register Space, and with speed comparable to hardware.)

This adapter and attachment architecture permits great flexibility of I/O configurations. (Even the Interrupt adapter can be customized.) We define the interface between CPU and adapter in the 801 Architecture document so that other adapters can be built, and we permit in our control program the substitution of adapter handling programs.

- Notice that the DMA transmits to and from the backing store, not the cache. This simplifies the transmission and eliminates the degradation due to cache broadcasting. In effect, the CPU is not measurably degraded due to I/O activity of the DMA. However, just as in the case of our two independent caches, software must ensure that buffers are properly synchronized with their use as data (or instruction) lines by the caches. The issuing of these cache managing instructions is confined

to our access method programs and does not currently appear to degrade performance unduly, but this is another area which we intend to measure carefully.

H) Operating System Facilities

The innovations in the 801 operating system do not come from any startling new function. In fact, the attempt is to provide a basic set of facilities, any of which can be replaced, and which concentrate on minimizing path lengths and providing acceptable response for large numbers of interactive display terminals. The major new approach in the supervisor is to work closely with the compiler in providing protection between users and for the supervisor itself.

Our prototype system will drive 3277 displays and attach to a 168 VM host. It will support Picollo disks and provide a compatible UC-interface adapter to support 3790 peripherals. With this configuration and an efficient, reliable multi-tasking, memory management control program, we will, in the second phase of the 801 project (1978) develop distributed applications and investigate the enhancements to the operating system needed for these applications.

III. Conclusions

In some sense the 801 appears to be rushing in the opposite direction to the conventional wisdom of this field. Namely, everyone else is busily moving software into hardware and we are clearly moving hardware into software. Rather than consuming the projected cheaper, faster hardware, we are engaged in an effort to save circuits, cut path lengths and reduce functions at every level of the normal system hierarchy. There are three comments which must be made about this anachronistic set of objectives.

1) It all depends on the absolute numbers, not on generalities. If the radically different direction we are pursuing leads to system cost/performance improvements of 10 - 30 percent we might as well go back to the technologists and seek that kind of improvement from chip density, new cooling methods, or cheaper memories. If we can demonstrate one to two orders of magnitude of cost/performance improvement (while keeping source language programs relatively unchanged) then there may well be some product potential in the 801 approach.

2) Less function does not necessarily mean harder to use. For instance, the 801 principles of operation are far easier to read and learn than S/370 (or many mini architectures). There is not as much there but it's all regular and straightforward (mostly anyway). The functions that we have subset from PL/1 have not impeded the development of our own programs at all. (Where we found that some missing function was useful we put it back in.) In fact, after a dozen years of experience, being able to eliminate a lot of marginally useful, complicated language constructs makes the language easier to understand and use. Our Control Program Facilities interface, because of its limited function, will similarly make it more accessible.

3) "Moving functions from software to hardware" today generally really means "moving functions from software to microcode". It should be clear by now that we have, in fact, moved *all* software into microcode. We are simply making this feasible by making the development of efficient, safe microprograms as easy (or easier) than software today.