George Radin

The 801 Minicomputer

This paper provides an overview of an experimental system developed at the IBM Thomas J. Watson Research Center. It consists of a running hardware prototype, a control program, and an optimizing compiler. The basic concepts underlying the system are discussed, as are the performance characteristics of the prototype. In particular, three principles are examined: (1) system orientation towards the pervasive use of high-level language programming and a sophisticated compiler, (2) a primitive instruction set which can be completely hard-wired, and (3) storage hierarchy and I/O organization to enable the CPU to execute an instruction at almost every cycle.

Introduction

In October 1975, a group of about twenty researchers at the IBM Thomas J. Watson Research Center began the design of a minicomputer, a compiler, and a control program whose goal was to achieve significantly better cost/performance for high-level language programs than that attainable by existing systems. The name 801 was chosen because it was the IBM number of the building in which the project resided. (The twenty creative researchers were singularly uninspired namers.) This paper describes the basic design principles and the resulting system components (hardware and software).

Basic concepts

• *Single-cycle implementation*

Probably the major distinguishing characteristic of the 801 architecture is that its instructions are constrained to execute in a single, straightforward, rather primitive machine cycle. A similar general approach has been pursued by a group at the University of California [1].

Complex, high-function instructions, which require several cycles to execute, are conventionally realized by some combination of random logic and microcode. It is often true that implementing a complex function in random logic will result in its execution being significantly faster than if the function were programmed as a sequence of primitive instructions. Examples are floating-point arithmetic and fixed-point multiply. We have no objection to this strategy, provided the frequency of use justifies the cost and, more importantly, provided these complex instructions in no way slow down the primitive instructions.

But it is just this pernicious effect on the primitive instructions that has made us suspicious. Most instruction frequency studies show a sharp skew in favor of high usage of primitive instructions (such as LOAD, STORE, BRANCH, COMPARE, ADD). If the presence of a more complex set adds just one logic level to a ten-level basic machine cycle (e.g., to fetch a microinstruction from ROS), the CPU has been slowed down by 10%. The frequency and performance improvement of the complex functions must first overcome this 10% degradation and then justify the additional cost. If the presence of complex functions results in the CPU exceeding a packaging constraint on some level (e.g., a chip, a board), the performance degradation can be even more substantial.

Often, however, a minicomputer that boasts of a rich set of complex instructions has not spent additional hardware at all, but has simply microprogrammed the functions. These microinstructions are designed to execute in a single cycle and, in that cycle, to set controls most useful for the functions desired. This, however, is exactly the design goal of the 801 primitive instruction set. We question, therefore, the need for a separate set of instructions.

In fact, for "vertical microcode," the benefits claimed are generally not due to the power of the instructions as much as to their residence in a high-speed control store. This amounts to a hardware architect attempting to guess which subroutines, or macros, are most frequently used and assigning high-speed memory to them. It has resulted, for instance, in functions like EXTENDED-PRECISION FLOATING-

 This paper originally appeared in the *Proceedings of the Symposium for Programming Languages and Operating Systems*, published in the *ACM SIGARCH Computer Architecture News*, Vol. 10, No. 2, March 1982; it is republished here, with only slight revisions.

POINT DIVIDE and TRANSLATE AND TEST on System/370 computers residing in high-speed storage, while procedure prologues and the first-level-interrupt handler are in main storage. The 801 CPU gets its instructions from an "instruction cache" which is managed by least-recently-used (LRU) information. Thus, all frequently used functions are very likely to be found in this high-speed storage, exhibiting the performance characteristics of vertical microcode.

Programming complex functions as software procedures or macros rather than in microcode has three advantages:

First, the CPU is interruptible at "microcode" boundaries, hence more responsive. Architectures with complex instructions either restrict interrupts to instruction boundaries, or (as in, for instance, the MOVE CHARACTERS LONG instruction on the System/370) define specific interruptible points. If the instruction must be atomic, the implementation must ensure that it can successfully complete before any observable state is saved. Thus, in the System/370 MOVE CHARACTERS instruction, before the move is started all pages are pretouched (and locked, in an MP system) to guard against a page-fault interrupt occurring after the move has begun. If interruptible points are architected, the state must be such that the instruction is restartable.

The second advantage of programming these functions is that an optimizing compiler can often separate their components, moving some parts out of a loop, commoning others, etc.

Third, it is often possible for parts of a complex instruction to be computed at compile time. Consider, for instance, the System/370 MOVE CHARACTERS instruction once again. Each execution of this instruction must determine the optimal move strategy by examining the lengths of the source and target strings, whether (and in what direction) they overlap, and what their alignment characteristics are. But, for most programming languages, these may all be known at compile time. Consider also a multiply instruction. If one of the operands is a constant, known at compile time, the compiler can often produce more efficient "shift/add" sequences than the general multiply microcode subroutine.

The major disadvantage to using procedures instead of microcode to implement complex functions occurs when the microinstruction set is defined to permit its operands to be indirectly named by the register name fields in the instruction which is being interpreted. Since, in the 801 and in most conventional architectures, the register numbers are bound into the instructions, a compiler must adopt some specific register-usage convention for the procedure operands and must move the operands to these registers when necessary.

A computer whose instructions all execute very efficiently, however, is attractive only if the number of such instructions required to perform a task is not commensurately larger than that required of a more complex instruction set. The 801 project was concerned only with the execution of programs compiled by our optimizing compiler. Therefore, within the constraints of a primitive data flow, we left the actual definition of the instructions to the compiler writers. The results, discussed later, generally show path lengths (that is, number of instructions executed) about equivalent to those on a System/370 for systems code, and up to 50% longer for commercial and scientific applications (given no hardware floating point).

- *Overlapped storage access*

Instruction mixes for the 801 show that about 30% of instructions go to storage to send or receive data, and between 10% and 20% of instructions are taken branches. Moreover, for many applications, a significant portion of the memory bandwidth is used by I/O. If the CPU is forced to wait many cycles for storage access its internal performance will be wasted.

The second major design goal of the 801 project, therefore, was to organize the storage hierarchy and develop a system architecture to minimize CPU idle time due to storage access. First, it was clear that a cache was required whose access time was consistent with the machine cycle of the CPU. Second, we chose a "store-in-cache" strategy (instead of "storing through" to the backing store) so that the 10% of expected store instructions would not degrade the performance severely. (For instance, if the time to store a word through to the backing store is ten cycles, and 10% of instructions are stores, this will add up to one cycle to each instruction on average depending on the amount of execution overlap.)

But a CPU organization that needs a new instruction at every cycle as well as accessing data every third cycle will still be degraded by a single conventional cache that delivers a word every cycle. Thus, we decided to split the cache into a part containing data and a part containing instructions. In this way we effectively doubled the bandwidth to the cache and allowed asynchronous fetching of instructions and data at the backing store.

Most conventional architectures make this decision difficult because every store of data can be a modification of an instruction, perhaps even the one following the store. Thus, the hardware must ensure that the two caches are properly synchronized, a job that is either expensive or degrading, or both. Even instruction prefetch mechanisms are complex, since the effective address of a store must be compared to the Instruction Address Register.

Historically, as soon as index registers were introduced into computers the frequency of instruction modification fell dramatically until, today, instructions are almost never modified. Therefore, the 801 architecture does not require this hazard detection. Instead it exposes the existence of the split cache to software and provides instructions by which software can synchronize the caches when required. In this system, the only program that modifies instructions is the one that loads programs into memory.

Similarly, in conventional systems in which the existence of a cache is unobservable to the software, I/O must (logically) go through the cache. This is often accomplished in less expensive systems by sending the I/O physically through the cache. The result is that the CPU is idle while the I/O proceeds, and that after an I/O burst the contents of the cache no longer reflect the working set of the process being executed, forcing it back into transient mode. Even in more expensive systems a broadcasting or directory-duplication strategy may result in some performance degradation.

We observed that responsibility for the initiation of I/O in current systems was evolving towards paging supervisors, system I/O managers using fixed-block transfers, and, for low-speed devices, a buffer strategy which moves data between subsystem buffers and user areas. This results in the I/O manager knowing the location and extent of the storage being accessed, and knowing when an I/O transfer is in process. Thus, this software can properly synchronize the caches, and the I/O hardware can transmit directly to and from the backing store. The result of this system approach in our prototype is that, even when half of the memory bandwidth is being used for I/O, the CPU is virtually undegraded.

Notice that in the preceding discussions (and in the earlier discussion of complex instructions) an underlying strategy is being pervasively applied. Namely, wherever there is a system function that is expensive or slow in all its generality, but where software can recognize a frequently occurring degenerate case (or can move the entire function from run time to compile time), that function is moved from hardware to software, resulting in lower cost *and* improved performance.

An interesting example of the application of this strategy concerns managing the cache itself. In the 801 the cache line is 32 bytes and the largest unit of a store is four bytes. In such a cache, whose line size is larger than the unit of a store and in which a "store-in-cache" approach is taken, a store directed at a word which is not in the cache must initiate a fetch of the entire line from the backing store into the cache. This is because, as far as the cache can tell, a load of another word from this line might be requested subsequently. Frequently, however, the store is simply the first store into what to the program is newly acquired space. It could be a new activation on a process stack just pushed on procedure call (e.g., PL/I Automatic); it could be an area obtained by a request to the operating system; or it could be a register save area used by the first-level-interrupt handler. In all of these cases the hardware does not know that no old values from that line will be needed, while to the compiler and supervisor this situation is quite clear. We have defined explicit instructions in the 801 for cache management so that software can reduce these unnecessary loads and stores of cache lines.

One other 801 system strategy leads to more effective use of the cache. Conventional software assumes that its memory is randomly addressable. Because of this assumption, each service program in the supervisor and subsystems has its own local temporary storage. Thus, an application program requesting these services will cause references to many different addresses. In a high-level-language-based system like the 801, control program services are CALLed just like a user's subroutines. The result is that all these service programs get their temporary areas from the same stack, resulting in much reuse of cache lines and, therefore, higher cache hit ratios.

So far we have discussed 801 features that result in overlapped access to the cache between instructions and data, overlapped backing store access among the caches and I/O, less hardware synchronizing among the caches and I/O, and techniques to improve the cache hit ratios. One other aspect of the 801 CPU design and architecture should be described to complete the picture.

Even if almost all instruction and data references are found in the cache, and the cache and backing store are always available to the CPU, a conventional CPU will still often be idle while waiting for a load to complete or for the target of a branch to be fetched. Sophisticated CPUs often keep branch-taken histories or fetch ahead on both paths in order to overcome this idle time. In the 801 project we observed that, with a small number of hardware primitives, software (i.e., the compiler) could reorder programs so that the semantics remained unchanged but the hardware could easily overlap this idle time with useful work.

On load instructions the register that is to be the target of the load is locked by the CPU. The CPU then continues execution of the instruction stream until it reaches an instruction that requires this register, at which time it idles until the load is completed. Thus, if the compiler can find a useful instruction to put after the load that does not require the result of the load, the CPU will not be idle at all while the data cache fetches the requested word. (And if the compiler

can find several such instructions to put after the load, execution of these will even overlap cache miss.)

Similarly for branches, the 801 architecture defines, for every type of branch instruction, an alternate form called BRANCH WITH EXECUTE. (This is similar to the delayed branch in the RISC computer [1].) These instructions have exactly the same semantics as their corresponding branch instructions, except that while the instruction cache is fetching the branch target the CPU executes the instruction that has been placed immediately after the BRANCH WITH EXECUTE instruction. For instance, in the sequence

LOAD R1,A

BNZ L

the CPU would be idle while the instruction cache was fetching L, if the branch was taken. Changing the BRANCH NON-ZERO to a BRANCH NON-ZERO WITH EXECUTE and moving the LOAD instruction results in

BNZX L

LOAD R1,A

which has exactly the same semantics but allows the CPU to execute the LOAD while the instruction cache is fetching the instruction at L. The 801 compiler is able, generally, to convert about 60% of the branches in a program into the execute form.

• *A compiler-based system*

So far we have discussed two major ideas which pervade the 801 system. First, build a CPU that can execute its instructions quickly (i.e., in one relatively short machine cycle), and define these instructions to be a good target for compilation so that resulting path lengths are generally commensurate with those for the same functions on more complex instruction sets (e.g., System/370). Second, define the storage hierarchy architecture, the CPU instructions, the I/O architecture and the software so that the CPU will generally not have to wait for storage access. The third major idea centers about the 801 compiler. A fundamental decision of the 801 project was to base the entire system on its pervasive use. This has resulted in the following system characteristics.

Instruction sets for conventional CPUs have been defined with an implicit assumption that many programmers will use assembly language. This assumption has motivated the definition of complex instructions (such as EDIT AND MARK, TRANSLATE AND TEST) almost as much as has the notion of a fast control store. But, increasingly, programmers do not use assembly language except where optimal performance is essential or machine functions are required that are not reflected in the source language.

The compiler for the 801 has demonstrated that it can produce object code that is close enough to best hand code generally so that assembly language programming is almost never needed for performance. The operating system has isolated those machine-dependent functions not reflected in the language (such as DISABLE, START I/O, DISPATCH) and developed efficient procedures which provide these functions with minimal linkage overhead.

The result is a system in which less than a thousand lines of supervisor code (and some of the "microcode" subroutine implementations of the complex functions) are written in assembly language. This has relieved the 801 architecture of the burden of being easy to program directly. Virtually the only programmers who are concerned with the nature of the architecture are the compiler writers and the "core" supervisor writers. All others see the system only through a high-level language. Because of this, the 801 architects were able to base their decisions solely on the needs of these few programmers and on cost/performance considerations.

Thus, the 801 architecture was defined as that set of run-time operations which

- could not be moved to compile time,
- could not be more efficiently executed by object code produced by a compiler which understood the high-level intent of the program, or
- was to be implemented in random logic more effectively than the equivalent sequence of software instructions.

It might at first seem surprising that compiler writers would not want powerful high-level instructions. But in fact these instructions are often hard to use since the compiler must find those cases which exactly fit the architected construct. Code selection becomes not just finding the fewest instructions, but the right instructions. And when these instructions name operands in storage instead of in registers, code selection depends upon the results of register allocation.

The 801 approach to protection is strongly based upon this compiler intermediary between users and the hardware. Conventional systems expect application programmers, and certainly subsystem programmers, to use assembly language or other languages in which it is possible to subvert the system (either deliberately or accidentally). Thus, hardware facilities are required to properly isolate these users. The most popular examples of these facilities are storage protect keys, multiple virtual address spaces, and supervisor state. These facilities are often costly and sometimes degrade performance. But what is more important is that they are often inadequate. Since even 16 different keys are insufficient for unique assignment, for instance, different users are sometimes given the same key or the system limits the

number of active users. Also, because the key disciplines are only two-level, many subsystems are forced to run with full addressing capability.

If, however, users are constrained to a properly defined source language, and their programs are processed by an intelligent compiler and run on an operating system that understands the addressing strategies of the compiler, it is possible to provide better protection at less cost. The 801 system, therefore, is based upon the assumption that certain critical components of the compiler are correct, and that all programs executing on the system (except for a small supervisor core) have been compiled by this compiler. The system will guarantee

- that all references to data (scalars, arrays, structures, areas) really do point to that data, and that the extents of the references are included in the extents of the data,
- that a reference to dynamically allocated-and-freed data is made only between an allocation and a free,
- that all branches are to labels, and all calls are to proper entry points in procedures,
- that the extents of all arguments to a procedure match the extents of their corresponding parameters, so that the protection persists across calls, and
- that all declarations of global (external) variables in separately compiled procedures have consistent extents.

This checking is often done at compile time, link-edit time, or program-fetch time, but, when necessary, trap instructions are introduced into the object code to check at run time. The resulting increase in path length due to this run-time checking is generally less than 10% because this code is optimized along with the rest of the program [2].

Notice that this is not a "strongly typed" approach to checking. Overlays of one data type on another are permitted, provided the domains are not exceeded. But our experience in running code conventionally on the System/370 and then on the 801 with this checking has shown that many program bugs are discovered and that, more importantly, they tend to be the kinds of bugs that elude normal component test procedures.

It was noted earlier that, because the operating system was also written in the 801's high-level language and compiled by the 801 compiler, its service programs were simply CALLed like any external procedure, resulting in better cache behavior. An even more important consequence of this design, however, is that the checking of matches between arguments and parameters is performed at the time a program is loaded into memory and linked to the supervisor. This results in efficient calls to supervisor services, especially when compared to conventional overhead. It means also that the compiler-generated "traceback" mechanism continues into the operating system, so that when an error occurs the entire symbolic call chain can be displayed.

The linkage between procedures on the 801 is another example of a consistent machine design based on a system used solely via a high-level language. We wanted applications on the 801 to be programmed using good programming style. This implies a large number of procedures and many calls. In particular, it implies that very short procedures can be freely written and invoked. Thus, for these short procedures, the linkage must be minimal.

The 801 procedure linkage attempts to keep arguments in registers where possible. It also expects some register values to be destroyed across a CALL. The result is that a procedure call can be as cheap as a BRANCH AND LINK instruction when the called procedure can execute entirely out of available registers. As more complex functions are required they increase the overhead for linkage incrementally.

Finally, the pervasive use of a high-level language and compiler has given the project great freedom to change. The architecture has undergone several drastic changes and countless minor ones. The linkage conventions, storage mapping strategies, and run-time library have similarly been changed as experience provided new insights. In almost every case the cost of the change was limited to recompilations.

This ability to preserve source code, thus limiting the impact of change, can have significant long-range impact on systems. New technologies (and packaging) often offer great performance and cost benefits if they can be exploited with architecture changes.

System components

- *The programming language*

The source language for the 801 system is called PL.8. It was defined to be an appropriate language for writing systems programs and to produce optimized code with the checking described previously.

PL.8 began as an almost-compatible subset of PL/I, so that the PL.8 compiler was initially compiled by the PL/I Optimizer. It contains, for instance, the PL/I storage classes, functions, floating-point variables, varying character strings, arrays with adjustable extents, the structured control primitives of PL/I, the string-handling built-in functions, etc. It differs from PL/I in its interpretation of bit strings as binary numbers, in its binary arithmetic (which simply reflects the arithmetic of the 801 hardware) and in some language additions borrowed from Pascal. It does not contain full PL/I ON conditions, multiple entry points, or the ability to

develop absolute pointers to Automatic or Static storage. Relative pointers, called Offsets, can be developed only to Areas. This discipline has several advantages:

- All program and data areas can be moved freely by the system, since absolute addresses are never stored in user-addressable data structures.
- Any arithmetic data type can be used as an offset (relative pointer) and all arithmetic operations can be freely performed, since the extent checks are made on every use.
- A store, using a computed offset, can only affect other data in that particular area. Thus, the locations whose values could have been changed by this store are significantly limited. This enhances the power of the optimization algorithms.
- It leads to better structured, more easily readable programs.

The optimizing compiler

There have been about seven programmers in the compiler group since the project began. A running compiler was completed after about two years. Since then the group has been involved with language extensions, new optimization techniques, debugging, and usability aids. It should be noted, however, that for about twenty years the Computer Sciences department at Yorktown Heights has been working on compiler algorithms, many of which were simply incorporated into this compiler.

The PL.8 compiler adopts two strategies which lead to its excellent object code. The first is a strategy which translates, in the most straightforward, inefficient (but correct) manner, from PL.8 source language to an intermediate language (IL). This translation has as its only objective the production of semantically correct object code. It seeks almost no special cases, so that it is relatively easy to debug. Moreover, the intermediate language which is its target is at a very low level, almost at that of the real 801 machine.

The next phase of the compiler develops flow graphs of the program as described in [3], and, using these graphs, performs a series of conventional optimization algorithms, such as

- common sub-expression elimination,
- moving code out of loops,
- eliminating dead code, and
- strength reduction.

Each of these algorithms transforms an IL program into a semantically equivalent, but more efficient, IL program. Thus, these procedures can be (and are) called repetitively and in any order. While these procedures are quite sophisticated, since each of them acts on the entire program and on all programs, a bug in one of them is very easily observed.

The power of this approach is not only in the optimizing power of the algorithms but in the fact that they are applied to such a low-level IL. Conventional global optimizing compilers perform their transformations at a much higher level of text, primarily because they were designed to run in relatively small-size memory. Thus, they can often not do much more than convert one program to another which could have been written by a more careful programmer. The PL.8 compiler, on the other hand, applies its optimization algorithms to addressing code, domain checking code, procedure linkage code, etc.

The second compiler strategy which is different from conventional compilers is our approach to register allocation [4, 5]. The IL, like that of most compilers, assumes an arbitrarily large number of registers. In fact, the result of each different computation in the program is assigned a different (symbolic) register. The job for register allocation is simply to assign real registers to these symbolic registers. Conventional approaches use some subset of the real registers for special purposes (e.g., pointers to the stack, to the code, to the parameter list). The remaining set is assigned locally within a statement, or at best a basic block (e.g., a loop). Between these assignments, results which are to be preserved are temporarily stored and variables are redundantly loaded.

The 801 approach observes that the register-assignment problem is equivalent to the graph-coloring problem, where each symbolic register is a node and the real registers are different colors. If two symbolic registers have the property that there is at least one point in the program where both their values must be retained, we model that property on the graph as a vertex between the two nodes. Thus, the register-allocation problem is equivalent to the problem of coloring the graph so that no two nodes connected by a vertex are colored with the same crayon.

This global approach has proven very effective. Surprisingly many procedures "color" so that no store/load sequences are necessary to keep results in storage temporarily. (At present the compiler "colors" only computations. There is, however, no technical reason why local variables could not also be "colored," and we intend to do this eventually.) When it does fail, other algorithms which use this graph information are employed to decide what to store. Because of this ability of the compiler to effectively utilize a large number of registers, we decided to implement 32 general-purpose registers in the hardware.

The compiler also accepts Pascal programs, producing compatible object code so that PL.8 and Pascal procedures can freely call one another. It also produces efficient object code for the System/370, thus providing source code portability.

Instructions and operands

Instruction formats and data representations are areas which saw significant change as the project evolved. This section describes the current version of the architecture. The kind of instruction and operand set requested by the compiler developers turned out, fortunately, to be precisely one which made hardware implementation easier. The overriding theme was regularity. For instance,

- All operands must be aligned on boundaries consistent with their size (i.e., halfwords on halfword boundaries, words on word boundaries). All instructions are fullwords on fullword boundaries. (This results in an increase in program size over two- and four-byte formats, but the larger format allows us to define more powerful instructions resulting in shorter path lengths.) Since the 801 was designed for a cache/main store/hard disk hierarchy and virtual-memory addressing, the consequence of larger programs is limited to more disk space and larger working sets (i.e., penalties in cache-hit-ratio and page-fault frequencies).

  With this alignment constraint, the hardware is greatly simplified. Each data or instruction access can cause at most one cache miss or one page fault. The caches must access at most one aligned word. Instruction prefetch mechanisms can easily find op codes if they are searching for branches. Instruction alignment and data alignment are unnecessary. Instruction Length Count fields (as in the System/370 PSW) are unnecessary and software can always backtrack instructions. Moreover, for data, traces show that misaligned operands rarely appear, and when they do are often the result of poor programming style.

- Given four-byte instructions, other benefits accrue. Register fields in instructions are made five bits long so that the 801 can name 32 registers. (This aspect of 801 architecture makes it feasible to use the 801 to emulate other architectures which have 16 general-purpose registers, since 16 additional 801 registers are still available for emulator use.)

  Four-byte instructions also allow the target register of every instruction to be named explicitly so that the input operands need not be destroyed. This facility is applied pervasively, as in "Shift Reg A Left by contents of Reg B and Store Result in Reg C." This feature of the architecture simplifies register allocation and eliminates many MOVE REGISTER instructions.

- The 801 is a true 32-bit architecture, not a 16-bit architecture with extended registers. Addresses are 32 bits long; arithmetic is 32-bit two's complement; logical and shift instructions deal with 32-bit words (and can shift distances up to 32). A useful way to reduce path length (and cache misses) is to define a rich set of immediate fields, but of course it is impossible to encode a general 32-bit constant to fit into an immediate field in a four-byte instruction. The 801 defines the following subsets of such constants which meet most requirements:

  - A 16-bit immediate field for arithmetic and address calculation (D field) which is interpreted as a two's-complement signed integer. (Thus, the constants $\pm 2^{15}$ can be represented immediately.)
  - A 16-bit logical constant. Each logical operation has two immediate forms—upper and lower, so that in at most two instructions (cycles) logical operations can be performed using a 32-bit logical constant.
  - An 11-bit encoding of a Mask (i.e., a substring of ones surrounded by zeros or zeros surrounded by ones). Thus, for shift, insert, and isolate operations the substring can be defined immediately.
  - A 16-bit immediate field for branch-target calculation (D field) which is interpreted as a signed two's-complement offset from the address of the current instruction. (Thus, a relative branch to and from anywhere within a 32K-byte procedure can be specified immediately.)
  - A 26-bit immediate field specifying an offset from the address of the current instruction or an absolute address, so that branches between procedures, to supervisor services, or to "microcode subroutines" can be specified without having to establish addressability.

- LOAD and STORE instructions are available in every combination of the following options:

  - LOAD or STORE.
  - Character, halfword, sign-extended halfword, or fullword.
  - Base + Index, or Base + Displacement effective address calculation. (Usage statistics for System/370 show low use for the full B + X + D form. Thus, a three-input adder did not seem warranted.)
  - Store the effective address back into the base register (i.e., "autoincrement") or not.

- Branches are available with the following branch-target specifications:

  - Absolute 26-bit address,
  - Instruction Address Register + Displacement (signed 16- or 26-bit word offset), or
  - Register + Register,

  BRANCH AND LINK forms are defined normally. But conditional branches are defined not only based upon the state of the Condition Register but on the presence or absence of a one in any bit position in any register. [This allows the TEST UNDER MASK – BRANCH CONDI-

TION sequence in System/370 to be executed in one machine cycle (and no storage references) if the bit is already in a register. Again, the power of global register allocation makes this more probable.]

- There are COMPARE AND TRAP instructions defined which allow the System/370 COMPARE – BRANCH CONDITION sequence to be executed in one machine cycle for those cases where the test is for an infrequently encountered exception condition. These instructions are used to implement the run-time extent checking discussed earlier.

- Arithmetic is 32-bit two's complement. There are special instructions defined to allow MAX, MIN, and decimal add and subtract to be coded efficiently. There are also two instructions defined (MULTIPLY STEP and DIVIDE STEP) to allow two 32-bit words to be multiplied in 16 cycles (yielding a 64-bit product) and a 64-bit dividend to be divided by a 32-bit divisor in 32 cycles (yielding a 32-bit quotient and a 32-bit remainder).

- The 801 has a rich set of shift and insert instructions. These were developed to make device-controller "microcode," emulator "microcode," and systems code very efficient. The functions, all available in one machine cycle, are as follows:
  - Ring-shift a register up to 31 positions (specified in another register or in an immediate field).
  - Using a mask (in another register or in an immediate field), merge this shifted word with all zeros (i.e., isolate the field) or with any other register (i.e., merge), or with the result of the previous shift (i.e., long shift),
  - Store this back into any other register or into storage (i.e., move character string).

  (This last facility allows misaligned source and target character string moves to execute as fast as two characters per cycle.)

Interrupts and I/O

I/O in the 801 prototype is controlled by a set of adapters which attach to the CPU and memory by two buses. The External Bus attaches the adapters to the CPU. It is used by software to send commands and receive status, by means of synchronous READ and WRITE instructions. Data are transmitted between the adapters and the 801 backing store through the MIO (Memory-I/O) bus. (As described previously, it is the responsibility of the software to synchronize the caches.)

Rather than support integrated and complex (multi-level) interrupt hardware, the 801 again moves to software functions that can be performed more efficiently by programming. Software on systems that provide, say, eight interrupt levels often find this number inadequate as a distinguisher of interrupt handlers. Thus, a software first-level-interrupt handler is programmed on top of the hardware, increasing the real time to respond. Moreover, the requirement to support eight sets of registers results in these being stored in some fast memory rather than in logic on-chip. This results in a slower machine cycle. If the real-time responsiveness of a system is measured realistically, it must include not only the time to get to an interrupt handler but the time to process the interrupt, which clearly depends on the length of the machine cycle. Thus, in a practical sense the 801 is a good real-time system.

Interrupt determination and priority handling is packaged outboard of the CPU chips in a special unit called the external interrupt controller (along with the system clocks, timers, and adapter locks). (This packaging decision allows other versions of 801 systems to choose different interrupt strategies without impacting the CPU design.) In this controller, there are (logically) two bit vectors. The first, the Interrupt Request Vector (IRV) contains a bit for each device which may wish to interrupt the CPU (plus one each for the clocks, timers, and the CPU itself for simulating external interrupts). These bits are tied by lines to the devices.

The second vector, called the Interrupt Mask Vector (IMV) contains a bit corresponding to each bit in the IRV. The IMV is loaded by software in the CPU. It dynamically establishes the priority levels of the interrupt requesters. If there is a one in a position in the IRV corresponding to a one in the corresponding position of the IMV, and the 801 CPU is enabled for interrupt, the CPU is interrupted.

On interrupt, the CPU becomes disabled and unrelocated and begins executing the first-level-interrupt handler (FLIH) in lower memory. The FLIH stores the interrupted state, reads the IRV, and determines the requester. Using this position number, it sends a new IMV (reflecting the priority of the requester) and branches to the interrupt handler for that requester, which executes enabled and relocated. Path lengths for the FLIH are less than 100 instructions (and can be reduced for a subclass of fast-response interrupts), and less than 150 instructions for the dispatcher (when the interrupt handler completes).

Internal bus

We have, so far, described a CPU that must have the following (logical) buses to storage:

- a command bus to describe the function requested,
- an address bus,
- a source data bus for stores, and
- a target data bus for loads.

Table 1 Performance comparison: System/370-168 and 801, for a Heap Sort programmed in PL.8.

CPU	*In inner loop*				
	Code size (bytes)	*No. of instructions*	*Data refs.*	*Cycles*	*Cycles/ inst.*
System/370-168	236	33	8	56	1.7
801	240	28	6	31	1.1

We observed that other functions might be implemented outboard of the CPU and could attach to the CPU via these same buses (e.g., floating point). Therefore, we exposed these buses in an 801 instruction, called INTERNAL BUS OPERATION (IBO). This instruction has operands to name the following:

- the bus unit being requested,
- the command,
- the two operands (B,D or B,X) which will be added to produce the output on the address bus,
- the source register, and
- the target register, if needed,

and three flags:

- privileged command or not,
- target register required or not, and
- address bus sent back to Base register or not.

Having defined this generic instruction, we gave bus-unit names to the instruction and data caches, the external-interrupt controller, the timer, and the relocate controller, and assigned the IBO op code to all instructions directed to these units.

Prototype hardware

A hardware prototype has been built for an early version of the 801 architecture, out of MECL 10K DIPs (Motorola Emitter Current Logic dual in-line packages). It runs at 1.1 cycles per instruction. (This number must be taken as an out-of-cache performance figure because the applications which currently run show hit ratios at close to 100% after the initial cache load.) We do not yet have multiple-user measurements.

The register file is capable of reading out any three and writing back any two registers within a single cycle. Thus, the CPU is pipelined as follows.

The first level of the pipeline decodes the instruction, reads two registers into the ALU, executes the ALU, and either latches the result or, for LOAD or STORE instructions, sends the computed address to the cache. On a STORE instruction, the data word is also fetched from the register file and sent to the cache.

The second level of the pipeline sends the latched result through the shifter, sets the condition register bits, and stores the result back into a register. During this cycle also, if a word has been received from the cache as the result of a load instruction, it is loaded into the register.

(The hardware monitors register names to bypass the load when the result is being immediately used.)

The cache is designed so that, on a miss, the requested word is sent directly to the CPU, thus reducing lockout while the cache line is being filled.

Table 2 Performance comparison: randomly selected modules on PL.8 compiler. (Note: Relative numbers are the ratios of 801 parameters to System/370 parameters.)

Module (In order of increasing size)	*Relative code size*	*Dynamic comparisons*	
		Relative instructions executed	*Relative data storage references*
FIND	1.02	0.91	0.60
SEARCHV	0.93	0.83	0.38
LOAD S	0.83	0.91	0.43
P2_EXTS	1.00	1.00	0.57
SORT_S1	0.86	0.78	0.59
PM_ADD1	0.86	0.96	0.63
ELMISS	0.87	0.86	0.69
PM_GKV	0.92	0.76	0.46
P5DBG	0.98	0.81	0.52
DESCRPT	0.86	0.75	0.42
ENTADD	0.79	0.76	0.42
Total	0.90	0.80	0.50

Performance comparisons

Tables 1 and 2 show some early performance comparisons. Since the compiler produces object code for the System/370 as well as the 801, these comparisons are possible for the same source programs and the same compiler. We use the number of cycles in the inner loops and the number of storage references in the inner loops to approximate dynamic performance.

Table 1 shows results for an in-memory sort procedure. Table 2 shows the results for randomly selected modules from the compiler itself. Note that as the modules get larger the power of global register allocation results in fewer storage references. Note also that, in spite of the fact that the

801 contains no complex instructions, the 801 modules contain fewer instructions and fewer instruction executions. This is because the complex instructions are generally very infrequent, whereas the 801 has a more powerful set of primitive instructions.

Conclusions

While we do not have nearly enough measurements to draw hard conclusions, the 801 group has developed a set of intuitive principles which seem to hold consistently.

At least in low-to-mid-range processor complexity, a general-purpose, register-oriented instruction set can be at least as good as any special vertical microcode set. Thus, there should be only one hard-wired instruction set, and it should be directly available to the compiler.

A good global register allocator can effectively use a large number of general-purpose registers. Therefore, all the registers which the CPU can afford to build in hardware should be directly and simultaneously addressable. Stack machines, machines that hide some of the registers to improve CALL performance, and multiple-interrupt-level machines all seem to make poorer use of the available registers.

Protection is far more effectively provided at a level where the source language program is understood.

It is easy to design and build a fast, cheap CPU, and it will become more so as VLSI evolves. The harder problem is to develop software, architecture, and hardware which do not keep the CPU idling due to storage access.

Acknowledgments

The seminal idea for the 801 and many subsequent concepts are due to John Cocke. The list of contributors has grown too large to list here. The following people were with the project from the beginning and were responsible for most of the design and implementation: Hardware: Frank Carrubba, manager; Paul Stuckert, Norman Kreitzer, Richard Freitas, and Kenneth Case. Software: Marc Auslander, manager; Compiler: Martin Hopkins, manager; Richard Goldberg, Peter Oden, Philip Owens, Peter Markstein, and Gregory Chaitin; Control Program: Richard Oehler, manager; Albert Chang. Joel Birnbaum was the first manager of the project and later a constant supporter. Bill Worley also contributed significantly through the years.

References

1. D. A. Patterson and C. H. Séquin, "RISC-I: A Reduced Instruction Set VLSI Computer" *Proceedings of the Eighth Annual Symposium on Computer Architecture*, May, 1981.
2. V. Markstein, J. Cocke, and P. Markstein, "Optimization of Range Checking," *Research Report RC-8456*, IBM Thomas J. Watson Research Center, Yorktown Heights, NY, 1980.
3. J. Cocke and P. W. Markstein, "Measurement of Program Improvement Algorithms," *IFIP 80 Proceedings, Information Processing*, S. H. Lavington, Ed., North-Holland Publishing Co., Amsterdam, 1980, pp. 221–228.
4. G. J. Chaitin, M. A. Auslander, A. K. Chandra, J. Cocke, M. E. Hopkins, and P. W. Markstein, "Register Allocation via Coloring," *Computer Languages* (British) **6**, 47–57 (1981).
5. G. J. Chaitin, "Register Allocation and Spilling via Coloring," *Research Report RC-9124*, IBM Thomas J. Watson Research Center, Yorktown Heights, NY, 1981.

Received May 6, 1982; revised November 11, 1982

George Radin *IBM System Products Division, 44 South Broadway, White Plains, New York 10601.* Mr. Radin is an IBM Fellow and director of architecture responsible for the definition of advanced architectures for System Products Division processors. He joined IBM in 1963 at the New York Programming Center. Since then he has worked on PL/I language definition in New York, as senior manager at the Thomas J. Watson Research Center, Yorktown Heights, New York, as manager of advanced system architecture and design in Poughkeepsie, New York, and as a member of the Corporate Technical Committee, Armonk, New York. Before joining IBM, he was manager of the computer center at the New York University College of Engineering. Mr. Radin received his B.A. in 1951 from Brooklyn College, New York, in English literature, his M.A. in 1954 from Columbia University, New York, in English literature, and his M.S. in 1958 from New York University in mathematics.