Comments on the 1800 Hardware

The 1800 has a 32K, 16-bit word memory which has a 2 microsecond cycle time, and is addressable as individual words or as even-odd double words. There are 6 16-bit programmable registers: the A-register, the Q-register (an extension of the A-register), 3 index registers, and a program counter. Instructions come in two flavors - short (1 word) and long (2 words) - with one level of indexing and indirect addressing. A priority interrupt scheme and general I/O capabilities are included for data acquisition and control purposes.

The machine is difficult to classify for comparison with other computers -- as a small computer, the classification IBM would most prefer, it is above average, being more powerful, easier to use, and undoubtedly more expensive than many small computers which are often quite primitive (such as the PDP-9). But as a medium-scale computer, the best classification on the basis of cost, the 1800 doesn't quite make it. However, with a few of the modifications suggested below, this machine might outperform the 360/40 and 44 computers, which could be the reason IBM will not make certain key improvements -- they'd rather sell 360's.

Based on the cost of a reasonable configuration with peripherals, the following discussion considers the machine as a medium-scale computer. It was designed explicitly as a data acquisition and control computer, and these comments are based on experience gained by using it in high-energy physics experiments. The intention is to evaluate the hardware design from a user's point of view, examining how well it fulfills its objectives, and suggesting improvements for both this and future machines. Brief comments on the IBM software are inserted where relevant.

I. Interrupts

A highly touted feature of the 1800 is a 24-level priority interrupt scheme. The basic idea is sound, since this arrangement provides an effective hardware queue for demands by the external environment on the computer facility. This enables the computer to always be "alive" to new requests, and to automatically allocate the processor to the most urgent (highest priority) task. However, the claim for rapid response is diminished by the number of instructions and consequent expenditure of processor cycles and memory space needed to save and restore the machine state on each interrupt. The TSX system was measured at 880 microseconds per interrupt as overhead, but this is excessive.
A good interrupt handler might operate in the 200-400 microsecond range, which is still unduly high. To speed this up, there should be hardware that saves all registers as part of the interrupt, and an additional single instruction to restore them, clear the interrupt, and return to the interrupted program when servicing is complete. This is related to re-entrancy problems and is considered more fully in that discussion. For now it suffices to say that interrupts are more costly in time and space than they should be on a computer designed around an interrupt driven scheme.

Interrupt levels can be masked or unmasked by programmed instructions to a hardware mask register. This register is "write only", but the program should also be able to read it in order to determine the current interrupt status. At present there is no way of knowing. This would probably require only a minor modification to the existing hardware. It is often necessary to disable all interrupts for short periods of time, mostly to avoid re-entrancy problems in brief sections of code. For this purpose, hardware enable/disable instructions should be provided along with a "nesting" counter so that, for example, 10 disable instructions would require 10 corresponding enables before the interrupts are actually enabled. These instructions would simultaneously block or unblock all interrupts, and would be in addition to the existing mask instruction which affects individual levels.

Several interrupt sources can be assigned to the same interrupt level so that masking or unmasking a level effectively blocks or unblocks all the triggers on that level. If an interrupt from any of the devices on that level occurs, the individual device must be specifically instructed to reset its interrupt instead of having a general "reset interrupt and clear level" instruction. This makes it impossible to ignore unwanted triggers, since all devices on a level must be known so that they can be explicitly reset. The result is that general interrupt handling requires device-dependent coding, and system hangups occur on interrupts that should just be ignored.

A useful feature is the ability of the program to cause a unique interrupt on any level, called a "program interrupt". This provides the programmer with a convenient driver mechanism consistent with the external trigger scheme, for initiating tasks, switching execution priorities, etc. The only drawback is that the hardware does not "remember" these interrupt triggers if they are unable to be serviced immediately, due either to current operation on a higher level, or to a current "level blocked" setting in the interrupt mask register. Hence higher levels can trigger lower ones (the most common use of this feature) only through contortions within the software operating system.

To the programmer and software system's designer this machine has a high "annoyance factor" because of the many features which almost do what you hope they would, but not quite (such as the programmed interrupts just described). The consequent flaw is seldom fatal but can boggle the mind with its complete irrationality. A second example is the grouping of the $2^4$ interrupt levels. Why $2^4$? Anybody's guess, perhaps a compromise between 16 and 32. Since at most 16 levels can be masked or unmasked at once (1 bit per level in a 16-bit word), how should the $2^4$ be divided between 2 words: 16-8? 12-12? Neither. 14-10??
II. Instruction set

The instruction set is larger than many small computers due to the addition of 3 index registers, which evidently makes this a medium scale machine. A good feature is the method of addressing operands for short instructions relative to the current location counter (or to any other index register, since the location counter is just index 0), thereby eliminating the extra 16-bit address word of the long instructions. Since an address word serves no useful purpose in the computation and usually contains highly redundant information, short instructions increase the code density considerably. In a typical program, 3 out of every 4 instructions will be short. The lack of an address in the instruction has an added advantage in that the assembled code requires no relocation by a loader prior to execution -- it's ready to go as is.

On the other hand, most long instructions waste 25 percent of the storage they occupy since information can be contained in only 24 of the 32 bits required, a very poor code density.

The instructions are basic but the set is incomplete, evidently due to designing by hardware engineers with insufficient regard for programmer use (at the machine level) or higher-level language translators (Fortran). Two glaring voids are the lack of register to register communications, and the inability to index all storage reference instructions. Once information is loaded into a register, it must be used there or dumped back into memory -- it cannot be transferred, exchanged, added, etc. to any other register. This is unfortunate, since most calculation is done in the A-register, but the result is often needed in an index register for addressing or shifting purposes. At least two new instructions -- "load index from A-register" and "load A from index" are desperately needed. Better, however, would be the approach taken in many small computers (and remember this is not a small computer and should therefore be "more powerful" in some sense) to "microprogram" inter-register operations. One of the unused 5-bit op-codes could indicate that the remaining 11 bits should not be treated as the usual format, index, and displacement fields, but rather as bit patterns for gating register-to-register and special single register operations. In addition to the simple transfers mentioned above, other useful register-to-register operations would be add, subtract, and, or, exclusive or, absolute value transfer, negative transfer, compliment (logical and arithmetic) transfer, and exchange. Most of the necessary data paths already exist for address arithmetic, and could be borrowed for this purpose. The op-code set could also be better designed, since the current "load status", and "wait" instructions, each of which requires a full 5-bit op-code, could be included as microprogrammed operations, thus freeing op-codes for more useful instructions (for instance, floating-point arithmetic).

Finally the B-register is currently inaccessible except by shifting from the A-register, and should be included in the register-to-register micro-operations. If nothing else, some means to load and store Q without affecting the A-register should be provided.

The inability to index the address used in storage-to-index and index-to-storage operations (LDX, STX, MDX op-codes) complicates register saving and restoring on interrupts and subroutine calls, and proves fatal to re-entrancy. The use of index registers as base registers for data tables is a useful method of programming because 1) it permits use of short instructions (the full address is not needed, just the relative location in a table), thus saving storage and processor time, 2) it reduces the need for absolute addresses,
thus alleviating both the relocation and re-entrancy problems. 3) it greatly simplifies storage management - subroutine work areas and parameter passing.

On the 1800 this technique is used with great success for operations which can be performed in the A register only, and would greatly increase the power of the machine were it extensible to index register operations as well. Unfortunately, it is not obvious how to extend the hardware in this direction, since there is no room left in short instructions for another index field, and not enough unused on-chip codes remain. Long instructions however do have 8 unused bits which could easily accommodate the extra index field.

For scientific applications the 1800 is badly crippled by the lack of floating-point hardware. The IBM supplied software takes 550 microseconds for a floating add, and probably could not be improved much below 250-300 microseconds by even the trickiest programming. (Other typical timings are multiply-400 microseconds, divide-570 microseconds, square root 5.5 milliseconds.)

In addition these subroutines occupy substantial chunks of memory. Hence floating point calculations in real-time programs are impossible with the data rates normally found in physics experiments. Since floating point (a-la-Fortran) is very desirable in a medium scale computer, the 1800 compares poorly in this respect with other machines in the same price class.

Some lesser points worth mentioning are: (a) the add-to-memory is much better than the usual variety, since all numbers in the range [-128, +127] can be added to any memory cell, rather than the usual addends 1 or 2. Unfortunately the memory address in this instruction cannot be indexed and cannot be indirect. (b) the instructions for subroutine entry, similar to those found in all small and medium computers, are bad for re-entrant code, as is discussed later. (c) the storage protection scheme is better than most, but is not usable in the IBM software by application programs. (d) very annoying is the requirement that all double-word operands be located on an even-address storage boundary. This benefits only one person -- the engineer who designed the addressing hardware -- while causing errors and wasting storage for everyone else. (e) the shift instructions don't work correctly. The shift count can be either the displacement field in the instruction, or the value in an index register, but not both. Most other machines are designed to add the displacement field to the index value, using the sum as the number of shift counts, since this is frequently the easiest way to perform shifting operations from the programmer's point of view. (f) the instruction set lacks on "execute" instructions, which is common in most other machines. This would prove useful in many applications, especially for saving and restoring registers on interrupts and in re-entrant routines (provided that the address of the "execute" could be both indexed and indirect). This would not solve all the problems in this area (see next section), but might make them a little less painful. More importantly, this instruction could be added to the existing hardware with little apparent redesigning. (g) the I/O scheme is well-designed from a programmer's point of view. The availability of nine high speed data channels having direct "cycle-stealing" access to any location of memory (independent of CPU operations or registers) is a very powerful feature. The channels are cheap selector-type having no programmable controls, but since all devices (except typewriters and plotters) have been specifically designed for attachment to them, there is no great need for more control. A multiplexor-type channel however would be useful. (see section IV on I/O).
III. General Overview

With an explanation of interrupts and instructions as background, it is appropriate to discuss the 1800 in terms of how it operates as a whole and how consistent it is within itself.

The natural organization of a data acquisition system consists of a group of semi-independent tasks, each handling one particular job in the total system, and each driven by unique demands from the external equipment on the computer. This lends itself quite easily to the priority interrupt scheme, where the most urgent demands trigger the highest priority interrupts. By attaching a task to each interrupt, it will operate on a unique priority level, independently of tasks on other levels.

In order to perform their intended functions, tasks require many resources of the system, such as subroutines, I/O devices, data storage, etc. These resources will be used on various levels, giving rise to the problem of re-entrancy -- if subroutine in operation on a low level gets interrupted by a higher level, and the task of that higher level then calls the same subroutine, extreme care must be taken in the subroutine to keep the intermediate results and return addresses separate for the two independent tasks using it. This problem is found only in a multi-tasking situation, since ordinarily a subroutine will completely finish its operation on one job before it can be called by another.

The essential limitation on re-entrant routines is where they may store temporary values, parameters, and return addresses. Fixed locations cannot be used, since the values for a low level call would be overwritten by a higher level call. A scheme must be implemented whereby the storage for a subroutine's temporaries is a function of the level on which the routine is operating. An alternative solution is to block all interrupts at entry to a subroutine and unblock them at exit. This allows the subroutine to use fixed cells for temporary storage, but is not an acceptable solution in general because it destroys the rapid response to external demands which is essential in a data acquisition system. It can be used only for very short sections of code.

A machine with a priority interrupt scheme should logically provide a mechanism for re-entrant routines. Unfortunately the 1800 hardware was designed with very little consideration of this vital problem -- there is no simple re-entrant method possible to handle this problem.

Several techniques are well known. The most elegant (and the easiest to use) requires a hardware stack and instructions which operate on values in the stack. When an interrupt occurs, all registers are saved by pushing them onto the stack. A subroutine uses the stack for all temporary storage, for passing parameters, and for saving the return address. Of course the instruction set must be completely oriented toward a stacking mechanism for this to be of any use. There is little hope of ever seeing a stack on an IBM machine.

A second method consists of assigning each interrupt level a block of storage pointed to by an address in a base (index) register. If all addressing is done relative to this base register, values for different levels will be automatically kept in separate storage locations. This scheme has inherent
limitations which can prove fatal. The block of storage reserved for each
level must be of fixed size, thus limiting the amount of temporary storage
available to subroutines. More serious is the requirement that subroutines
use fixed portions of the block, assigned so that cells of one subroutine will
not overlap those of another. To make this assignment requires knowledge of
the storage requirements of all subroutines, and defeats the concept of sub-
routines as self-contained semi-independent units. It also makes impossible
automatic storage assignment such as that needed in FORTRAN. A third method
involves a set of registers for each interrupt level, but is only found on
larger machines due to the cost involved.

The 1800 TSX software system attempts to implement the second technique
using an index or the base register, but in addition to the aforementioned
limitations inherent in the scheme, also runs into the following difficulties
with the hardware: 1) index register instructions (LDX, STX, MDX) cannot refer
to indexed storage addresses. Therefore index registers cannot be used unless
they do not require loading, storing, or modification with operands in storage
(i.e. all indexing instructions must have "immediate" operands); 2) lack of
communication between the A-register and the index registers requires that
addresses and shift counts computed in the A-register be stored and then loaded
into index registers. This runs into problem 1); 3) since double indexing is
not possible, variables addressed relative to a base register cannot be
indexed arrays. This makes searches, block transfers, vector and matrix operations
extremely difficult; 4) indirect addressing is allowed to a depth of only
1 level, and the indirect address itself cannot be indexed. Therefore parameter
addresses passed to subroutines cannot be relative to a base register, unless
a lot of address computation is performed that is irrelevant to the main
computing job; 5) the subroutine entry mechanism is all wrong. The BSI
instruction, used for all subroutine calls, stores the current location in the
effective address. Unless the subroutine immediately a) blocks interrupts
(bad) or b) loads the stored return location into a register (thereby destroying
any information contained in that register), the subroutine cannot be re-entrant,
since the next call will overwrite the return address. The subroutine call
instruction should either push the return address into a stack, or load it
directly into a register without the necessity of going through storage.

These problems are found in all small computers and most medium ones.
In this respect it is perhaps unfair to single out the 1800, but it does serve
as a typical example of an outmoded, but still very commonly found, design
philosophy -- namely to get together a group of hardware engineers, set a maxi-
mum hardware cost figure, and let them design a machine. Then after it's complete,
give it to a programmer to develop software, which will now cost 2 or 3 times
as much as it would have if a few necessary features were incorporated into
the hardware. Both engineers and programmers should design a computer, since
the cleverest engineering schemes are often totally useless to a programmer.
This is especially true when dealing with higher level languages -- the 1800
hardware failings have resulted in a large, extremely slow FORTRAN which is
useless in most real-time applications. Furthermore the compiler writers
realized that re-entrancy with 1800 hardware is impossible in the general case,
and did not even attempt to make FORTRAN programs re-entrant.
IV. Input/Output

I/O on the 1800 is divided into two classes -- "process I/O" for data acquisition and control devices, and "DP I/O" for the data processing I/O gear usually found on computers (tapes, card readers, disks, printers, plotters, etc.).

A. DP I/O

The peripherals attached to the 1800 are far too slow for the computer mainframe, a situation true in most computer systems, but in this case the speed differences are extreme. Since most devices (card reader, printer, tapes, disks) operate on data channels having direct cycle-stealing access to any location in memory, a great deal of asynchronous operation is theoretically possible. The advantage is negated by the slow devices on the channels, since the CPU will have a new transmission ready to go long before a previous one is complete, and thus ends up waiting for the device anyway. The situation cannot be remedied without undue expense, since many of the peripherals have been designed specifically for the 1800 and cannot be replaced by faster devices because of incompatible interface requirements.

A further flaw with DP I/O device interfacing is that each device requires its own information coding for characters. The 1443 printer code differs from the typewriter output code, which differs from the typewriter input code, and none of these is the standard EBCDIC code. The conversion between character representation is a nuisance to the programmer, wasting both time and space, and causing errors, inconveniences, and a proliferation of code-dependent routines. Since the devices themselves are slow and transmissions can be overlapped with conversion, wasted time is not as significant as wasted space, since the conversion routines require large tables, and devices cannot share buffers because of the incompatible codes. Use of a single internal coding (say EBCDIC) with hardware decoding as part of the transmission to the individual devices would be a substantial improvement.

In addition to the need for larger capacity disk storage and faster card readers and printers, the 1800 also needs a CRT display. The usefulness of a scope for operator interaction in a real-time data acquisition system is beyond question, so that the lack of this component puts the 1800 at a severe disadvantage in comparison with other data acquisition machines on which display scopes are available. An elaborate scope such as the IBM 2250 is not necessary; one with horizontal-vertical vector generation and simple point plotting that operates on a data channel from a buffer in 1800 memory would be sufficient. Character generation, omni-directional vectors, external memory buffers, and light-pen input, while useful in many instances, are not essential.

B. Process I/O

The process I/O consists of process interrupts, digital input and output, and analog input and output.

The process interrupts are groups of 16 electrical terminals arranged so that each terminal, independently of the others, will trigger a unique interrupt whenever the proper signal is received. The assignment of triggers to priority levels is flexible, with any number (up to 256) allowed on a single level.
For a programmer, this is a simple, well designed, easy to use scheme.

The basic unit in the digital I/O is called a group, each group consisting of a 16-bit data bus on the back panel of the computer to which 16 electrical lines from external devices are attached for transmitting data signals. Any number of these groups can be ordered from IBM, and they are a very convenient method of attaching data acquisition devices to the computer. The program identifies each group by a unique "group address" and can therefore route data to or from "individual devices without address decode and recognize functions being incorporated into all external devices. The scheme is general, and fairly useable, but has serious drawbacks -- the hardware gives no indication to the individual devices when they have been selected by the program to transmit data. The program can sometimes supply this "select" signal via a separate register, but this is not always possible, and requires much system bookkeeping to avoid errors. The simplest solution -- add a seventeenth bit to each group which will be a select line and signals the device when to send or receive data from the 16 other lines in that group. This is especially necessary if the next improvement is implemented.

Currently all the input (or output) groups operate as a single input (or output) device on one data channel. Therefore if a transmission is in progress to one group, no other groups can be used until the transmission is complete. This is a real bottleneck and could be alleviated by designing a multiplexor channel in which each group was a separate device operating independently of and asynchronously to all other groups. In this setup, a "device select" line for each group is mandatory. There are rumors that this multiplexing type of digital input-output is an RPQ available from IBM, but nothing appears in print about it.

There should also be provided a means for the external device to signal a "transmission terminated" to the channel, and another signal to inform the device when the channel has finished its transmission. Currently the device must send or receive exactly the number of words expected by the 1300 program -- if there are too few, the 1300 will wait forever to complete the transmission, and if there are too many, the 1300 will not accept the extra words, nor will it give any indication to the device that the transmission has ended.

All these problems result from the fact that the hardware needs more synchronization and control for entire transmissions (as opposed to the control of individual words in a transmission, which is handled satisfactorily now.). This also holds true for analog input and output, since the design is analogous. Cost was probably the primary factor influencing the current "part-way" hardware design, but the user pays anyway in increased software, overhead time, and inability to utilize completely the hardware that does exist.

The programmer also lacks sufficient information about a data channel transmission. He cannot obtain any information from the channel -- not even a simple busy/not busy indication -- only the devices can be tested for status. However neither the digital I/O nor the analog I/O provides sufficient status about their operations. At least the word-count and channel data address should be accessible, so that the program could compute how far an incomplete transmission progressed. The channels are relatively inexpensive, but then they are really just high-speed data buses with very little programmable control. The DP I/O devices individually make up for some of these deficiencies by having their own word count registers which are available to the program in some
instances (2401 tape drive, for example). Either the process I/O devices should be redesigned to incorporate similar hardware, or the channels should be rebuilt to conform with standard practices.

A final comment about attaching home-built equipment to the process I/O terminals. The electrical specifications seem to have been chosen at random -- the levels on inputs differ from those on outputs, and data lines have different specifications than ready-sync lines. Further, many levels are opposite of those standard in the industry (i.e., negative-to-ground instead of ground-to-positive). These inconsistencies lead to many headaches for engineers designing the external equipment and result in unnecessary confusion. The designers of the process I/O should have given more thought to developing a consistent set of electrical requirements.