5/12/7? [illegible]

THE MODEL 500 CENTRAL PROCESSOR

Introduction

The central processor (CPU) of the Model 500 is the system component which is responsible for executing user programs. Certain parts of the operating system also run on the CPU, but the most time-consuming functions of the operating system are relegated to special-purpose processors.

The CPU has been designed with two major ideas in mind:

1) It should match the structure of a powerful high-level language closely, so that code compiled for this language will be nearly as efficient as hand-written machine code. To the extent that this goal is achieved, any incentive to resort to machine language programming is removed, and such well-documented advantages of high-level languages as good documentation and rapid coding are extended to all programming.

2) It should permit the efficient execution of programs written in user-oriented languages such as Basic, Cal and APL. These languages have the desirable property that the programmer does not have to know anything about the implementation; everything that happens is reported to him at the source language level. On most machines this property is paid for by costly software interpretation and error-checking during execution. With properly designed hardware those costs disappear, and programs written in Basic

can execute as efficiently as those in Fortran.

In addition to these major advantages, the Model 500 CPU has other important features which contribute to its efficiency:

- short instructions and powerful addressing modes reduce the size of programs
- all routines are automatically re-entrant, so that a single copy can be shared by many simultaneous users
- subroutines can operate recursively at negligible extra cost
- an orderly instruction set makes fast compilation possible
- all code is relocatable to facilitate incremental compiling
- arrays are accessed through descriptors which automatically check that subscripts fall within the correct bounds
- matrix elements can be referenced without multiplications and without expensive optimization
- part-word fields in tables can be accessed almost as cheaply as full-word quantities
- numerous built-in operations speed the handling of strings in various byte sizes

The next few pages will outline the structure of the CPU to show how these wonders are accomplished.

Registers and Instructions

The Model 500 uses 24-bit words which can hold a single 24-bit twos complement integer, four 6-bit bytes, three 8-bit bytes or two 12-bit bytes. Two words hold a floating point number with an 11-bit exponent ($10^{\pm 300}$) and a 36-bit coefficient (more than 10 decimal digits).

The unique memory organization of the Model 500 permits a large number of memory locations to be accessed as efficiently as the central registers of other machines. As a consequence, the register structure of the CPU has been greatly simplified. There are:

a program counter P;

an accumulator A for general integer and floating-point calculations;

an index register X for address calculations and simple integer arithmetic (add, subtract, multiply by small constant);

two base registers, a <u>local</u> base L and a <u>global</u> base G, whose use is explained in the treatment of addressing below;

a status register S containing indicator bits, a condition code and various mode bits;

a compute-time clock which counts time spent by this user process on the CPU with 10 μs accuracy;

an interval timer used by the operating system.

All registers except A are 24 bits long. The A register is

96 bits long, to accommodate double-precision floating point and four-word data objects.

A full set of arithmetic and logical operations on 24-bit integers is provided: load, store, exchange, add, subtract, multiply, divide, and, or, and exclusive or. A carry indicator and special instructions which sense it permit efficient multiple-precision integer arithmetic. A complete set of shift operations allow a choice of

- single or double-word operations
- arithmetic or logical shifts, or cycles
- shift direction controlled by the sign of the operand

All arithmetic operations set the condition code depending on whether the result is less than, equal to or greater than zero. Six conditional branch instructions permit any combination of these conditions to be tested. Compare instructions allow tests to be made without destroying any registers.

To-memory operations allow incrementing or decrementing any location, or adding the accumulator to memory. These operations also set the condition code, and provide efficient facilities for closing simple loops. A special instruction to increment X and branch can be used for the tightest loops.

Floating point add, subtract, multiply, divide, compare and negate are provided, as well as instructions to fix and float numbers. All floating point arithmetic is normalized and correctly rounded.

The floating accumulator maintains 10 extra bits of precision to minimize round-off error in polynomial evaluation and other common computations. All floating point operations check for illegal operands and trap to a user-provided routine; this feature makes it easy to detect references to variables which have never been initialized. Overflow and underflow also trap to user-provided routines, leaving results which are correct except for the sign of the exponent. Divide by zero has its own trap, as does an attempt to fix a number too big to be represented as an integer. A special instruction fixes a number and loads the resulting integer into X; this is important for languages like Basic which use floating-point subscripts.

The very powerful function ~~and~~ instruction is described in a separate section. There is also a simple branch which leaves the return link in X for special purpose subroutine linkages.

The addressing structure provides for character pointers which can address any 6, 8 or 12-bit byte. Instructions are provided to increment such pointers by n bytes, to find the number of bytes in a string (i.e., between two character pointers), and to advance to the next or previous byte of a string, with a test for exhaustion of the string.

A non-addressable instruction copies a block of words to a new place in memory, ensuring that the block will not be overwritten during the operation. A variant copies a constant into a block of words. Other instructions count the number of one bits in the accumulator and the number of leading zeros or ones.

Addressing

The CPU has an address space of 2^{18} 24-bit words, divided into three <u>rings</u> as follows:

$0-377777_8$	user ring	2[illegible]	user programs
$400000-577777_8$	utility ring	2^{16}	less privileged system routines
$600000-777777_8$	monitor ring	2^{16}	highly privileged system routines

The hardware protection allows programs executing in a given ring to reference only that and lower rings. This arrangement allows system routines convenient access to the memory of user programs but ensures that the system is protected.

The addressing system divides naturally into two parts: addressing from instructions and indirect addressing. The main goals of the instruction addressing are

1) To allow 2^{18} words to be addressed from a 24-bit word which also must hold a 7-bit operation code, leaving 17 bits for the address, mode bits and index register selection;
2) To provide good facilities for accessing arrays and referencing structured data;
3) To permit subroutines to have local storage in a convenient way;
4) To make all programs automatically relocatable;
5) To allow immediate addressing for small constants, so that space and memory references can be saved.

There are four CPU registers which can be used to hold full-size addresses. Each of these has a specific purpose, and general indexing is done from memory locations in a manner described below.

- the global base register G addresses storage which is common to a large group of routines which are compiled or at least loaded together. It rarely or never changes during the execution of a program.
- the local base register L addresses storage which belongs to a single function or subroutine. To facilitate its use for this purpose, it is given special treatment by the function call and return operations, so that no extra instructions need to be executed to keep it pointing to the local storage of the current function.
- the X register is used for indexing calculations too complex to be handled by the special memory indexing modes.
- the P counter points to the instruction currently being executed.

Instruction addressing allows locations near each of these registers to be addressed conveniently. In particular, there is

1) direct and indirect addressing from G to $G+2^{14}-1$ (+16k)
2) direct and indirect addressing from L to $L+2^{11}-1$ (+2k)
3) direct addressing from $X-2^{7}+1$ to $X+2^{14}-1$ (-128, +16k)
4) direct and indirect addressing from $P-2^{11}+1$ to $P+2^{11}-1$ ($\pm$2k)

In addition, there is

5) Immediate addressing for operands in the range -2^{11} to $2^{11}-1$ ($\pm$1k)

or $X-2^{11}$ to $X+2^{11}-1$ ($X\pm$1k)

In addition, there are four addressing modes which take two operands from the instruction word, one of 8 bits and the other of 6. Two of these allow 64 words pointed to by a memory location to be addressed.

6) direct and indirect addressing from Q-32 to Q+31, where Q is the contents of some location from G to G+127 or L to L+127.

These modes are used for access to structured data. For example, if T is a pointer to a table entry and happens to be stored within 128 words of L or G, then any word of the table entry can be accessed with this mode, provided the entry has 32 or fewer words.

The remaining two modes are for array or part-word access. They allow an array descriptor or field descriptor (see the discussion of indirection) and an index to be specified, provided both are stored in memory close enough to L or G. The index quantity is used to select an array element, or to provide a base address for the part-word reference.

With these modes the following constructs from well-known languages can be compiled into short, fast code:

Fortran:	A(I) = B(J)	2 instructions
	A(I,J+2) = B(J,I-5)	4 instructions
PL/I:	(P→MAN).AGE = A(I)	2 instructions

Indirect addressing provides a number of additional facilities. Some of these remove the size limitations enforced on instruction addressing, and others allow access to bytes and part-word fields and to arrays. The first kind of indirect word allows:

1) direct or indirect addressing of any word in memory, with an optional trap on stores
2) direct and indirect addressing from L to L+2^{14}-1 (+16k)
3) direct addressing of any amount relative to X, with an optional trap on stores
4) direct and indirect addressing from R-2^{14}+1 to R+2^{14}-1 ($\pm$ 16k), where R is the address of the indirect word

In addition to the option to trap stores for absolute and indexed addresses, it is possible to trap all references for any of the modes. This is very convenient for monitoring references to, or changes in, any variable or array.

A second kind of indirect word points to any 6, 8 or 12 bit byte in memory. Such a byte may be used as the operand of any machine instruction which normally works on a full word. Convenient instructions exist for incrementing and decrementing these character pointers.

A third kind of indirect word, called a field, specifies any group of 1 to 24 bits in any word within $\pm$ 512 of the current indexing quantity (which may be a pointer stored in memory; see above). The specified bits, with optional sign extension, may also be used as the operand of any machine instruction which

normally works on a full word. Typically a field is created for each component of a structure or record which occupies less than a full machine word.

Finally, there is an array indirect word which checks the indexing quantity for being at least Ø or 1 and less than an upper bound which may be anything from Ø to 128k. It then multiplies the index by anything from 1 to 63 and adds it to an 18-bit base address for the array. There are options to trap on any reference or on stores only. This type of indirection provides automatic subscript bounds checking, and permits access to arrays of complex numbers or of structured data without any additional complications. Multi-dimensional arrays are normally handled by marginal indexing: a 5 x 5 matrix has a vector of 5 descriptors for the 5 row vectors, and two indexing operations are required. This is slightly more expensive than the code which would be generated by a good compiler on a conventional machine for a matrix element in a loop. Of course there is nothing to prevent a good compiler from making the improvement; a cheaper and faster compiler will do much better than it would on the conventional machine.

A final comment on addressing is perhaps worthwhile. The reader may have concluded that very few programmers could hope to learn all the addressing machinery well enough to use it effectively, and he is of course right. Most of the modes were put there not for programmers, but for compilers. Almost every common construct in a higher-level algebraic or system programming language (Fortran, PL/I, Jovial) has a straightforward and efficient representation in the addressing hardware of the CPU.

Function Calls

The most elaborate feature of the Model 500 CPU is its provisions for function calls. There is a single instruction which performs essentially all the operations required by a function or subroutine call in any reasonable higher-level language, including rather complete error-checking. The reason for expending so much effort on this aspect of the machine is that properly written programs consist of many small routines which are constantly calling each other. It is therefore very important to handle each call efficiently and completely, without imposing any requirement for long CALL-SAVE-RETURN sequences which waste both time and space.

One characteristic of the CPU which greatly simplifies the problem is the absence of central registers. Since the memory and processor architecture encourage the use of memory locations for bases and indexing quantities, there is no reason to preserve A or X across a function call. The global base is normally unchanged, and only the local base L requires special handling. It must be set up at each call to point to the local storage for the new routine, and the old value must be saved so that it can be restored upon return. A function descriptor thus consists of two quantities, one specifying the new value of P (the location of the function) and the other the new value of L. The return link is stored in exactly the same form, with the old values of P and L.

A very important feature is the ability to allocate storage for the function being entered on a stack. When this is done the function descriptor tells how many words are needed, and the hardware automatically fetches and updates a pointer to the current top of stack. There is a bounds check for stack overflow, which causes a trap if it fails. A user-provided trap routine can provide more stack space if necessary. The return descriptor is set up in such a way that the stack pointer will be restored on return. An additional complication arises when a transfer exits from several functions at once (a 'non-local goto' in PL/I); in this case it is necessary to reestablish L and the stack pointer. In the important case of an exit label, passed as a parameter, everything is taken care of automatically by the hardware. Since the details are rather involved, they are omitted here.

The second major job done by the function call instruction is argument or parameter passing. As part of this process the actual and formal arguments are checked for agreement in number and type. This checking can be suppressed if desired. Each formal argument specifies whether the address (as in Fortran) or the value (as in most system programming) should be passed, When an array or label is passed by value, any relative address in it is converted to absolute so that it will still work in the new environment. For a label there is in addition an option to attach the current value of L, so that the stack will be unwound properly when it is used to return control.

Since return is handled exactly like call except that no return descriptor is stored, any number of values can be returned without difficulty. Since the return descriptor is stored in the first two words pointed to by the new L, all programs are automatically re-entrant and programs which have stacked local storage are automatically recursive.

A bit in the operation code, when set, causes the remaining 6 bits to be interpreted as a call of one of 64 programmer-designated functions. The address field is then free to pass the first argument, so that one word is saved for each call of the 64 most common functions in a program.

System Calls, Traps and Interrupts

A sharp distinction is drawn in the Model 500 among three essentially different kinds of control transfer which are usually lumped together:

1) System calls, or requests to the operating system for some kind of service. These are treated exactly like ordinary subroutine calls, except that since a higher ring is being entered the call must address a designated entry point. A low-level program is thus prevented from calling a high-level one except in a legal way. Any addresses passed on a system call are checked for legality in the context of the caller. This means, for example, that if the monitor is passed an array it can store into that array without fear of damaging itself. Note also that the two system levels or rings allow a small bulk of highly sensitive code to be protected from a larger volume of less critical code.

2) Traps occur synchronously with program execution as a result of some untoward occurrence triggered by the program. They are further divided into two sub-categories:

2a) User traps, which send control directly to a user-provided routine with slighly less overhead than a function call. These result from unusual events which may be anticipated, and

perhaps frequent, so that a user may wish to take efficient corrective action and continue. Examples of user traps are

- array subscript out of bounds
- floating overflow or underflow, divide by zero, fix overflow, undefined floating-point
- indirect addressing reference or store trap
- stack overflow
- function call error such as type mismatch

The trap always leaves things set up so that execution can continue, and no information is destroyed. A user trap always leaves control in the ring in which it occurs.

2b) System traps, which the user is not equipped to handle. These always give control to the monitor. Typical system traps are

- reference to page not in core, or non-existent
- attempt to store into read-only page
- attempt to execute privileged or undefined instruction
- more than 16 levels of indirection

The action taken is defined by the operating system, and may involve returning control to a user trap routine if that is appropriate. It is expected, however, that system action will normally be required and that the user will rarely wish to get control.

3) Interrupts, or requests for processor service which come from outside the program being executed. All cases in which it is necessary to switch the CPU from one program to a completely independent one require the entire processor state to be saved and restored and are handled in a single uniform fashion. No distinction is therefore made between changing user programs and servicing a tape unit. The processor-switching operation is ordered by an independent processor which keeps track of the priorities of the various competitors for the CPU; it takes about 20 μs, in addition to the time required to reload the map for the new process.

Memory Mapping

The 128k of addressable memory available to a Model 500 user program is organized into 64 pages, each containing 2048 words. An additional 64 pages are available to the monitor and utility levels of the system. Each program can have its own set of 64 addressable pages, or of course fewer for programs which do not need this much memory. A program's pages can be private to that program or shared with other programs in any combination desired. Of course each program establishes the contents of its memory independently of any other program. The total amount of addressable memory for n user programs is n*128k words.

The CPU contains hardware registers which establish the physical location in core of each page of memory belonging to the running program. No time is added to program execution by the mapping process which these registers perform.

When a program starts to run, the hardware mapping registers are cleared. As each new page is referenced, the CPU automatically loads the appropriate mapping register. This process takes 5 to 10 μs, a cost which is incurred only once for each page touched by the program. When the CPU switches to another program, all the registers are cleared and must be reloaded.