CDC 6000 Hardware

CPU

Our CDC 6500 was basically two CDC 6400 CPUs in one cabinet, sharing memory and I/O. The 6000 CPU was RISC long before it was popular to have a reduced instruction set. The CPU was usually said to have around 74 instructions (the exact number depends on how you count ‘em), but by modern standards the number was less than that. The rough number 74 counts each of 8 addressing modes three times, whereas you could reasonably say that an addressing mode shouldn't be counted as a separate instruction at all. Despite the lean instruction set, there were few complaints about the instruction set missing instructions.

Arithmetics was 1's complement. This was sometimes inconvenient, because a word of all 1's tested as zero, just as did a word of all 0's. Even then, 2's complement (in which a word of all 1's had a value of -1) was more common - and now 2's complement is nearly universal. One of the few advantages of 1's complement arithmetic was that taking the negative of a number involved simply inverting all the bits, whereas in 2's complement, you need to invert all bits and then add 1. A computer science professor once told us that Control Data chose the inferior 1's complement approach because 2's complement was patented, and this web page seems to confirm that.

Central Memory

Central memory (CM) was organized as 60-bit words. There was no byte addressability. If you wanted to store multiple characters in a 60-bit word, you had to shift and mask. Typically, a six-bit character set was used, which meant no lower-case letters. Our site invented a 12-bit character set, which was basically 7-bit ASCII with 5 wasted bits. Other sites used special shift/unshift characters in a 6-bit character set to achieve upper/lower case. The short-lived Cyber 70 Series which followed the 6000 Series added a Compare and Move Unit (CMU) which did complex character handling in hardware. The CMU was not used much, probably due to compatibility concerns. The CMU was such a departure from the 6000's lean and mean instruction set that the CDC engineers must have been relieved to be able to omit it from the next line of computers, the Cyber 170 Series.

CM addresses were 18 bits wide, though I believe that in the original 6000 line, the sign bit had to be zero, limiting addresses to 17 bits. Even without the sign bit problem, though, the amount of addressable central memory was extremely limited by modern standards. A maxed-out 170 Series from around 1980 was limited to 256K words, which in total bits is slightly less than 2 megabytes (using 8-bit bytes purely as a means to compare with modern machines). In the early days, 256K words was more than anyone could afford, but eventually this addressability problem became a real problem. CDC never found a way around it.

The closest there was to a workaround was the Extended Core Storage (ECS) unit. This was auxiliary memory made from the same magnetic cores of which CM was fabricated. (More recent versions of ECS were named ESM, Extended Semiconductor Memory.) ECS was accessible only by block moves to or from CM. I can't remember the address width of ECS, but it was much larger than 18 bits. But not being able to run programs or directly access data from ECS meant it was used mostly to store
operating system tables or to swap programs.

I say “swap” programs because there was no virtual memory on the machine. Memory management was primitive. Each user program had to be allocated a single region of contiguous memory. This region started at the address in the RA (Reference Address) register and went for a certain number of words, as dictated by the contents of the FL (Field Length) register. The CPU hardware always added the contents of the RA register to all address references before the memory access was made; as far as the program was concerned, its first address was always 0. Any attempt to access memory >= FL resulted in a fatal error.

As programs came and went from CM, holes opened up between regions of memory. To place programs optimally in memory, an operating system had to suspend the execution of a program, copy its field length to close up a gap, adjust the RA register to point to the program's new location, and resume execution. On the 6500, it was actually faster to do a block move to ECS and then a block move from ECS than it was to move memory in a tight loop coded with the obvious load and store instructions. This changed with the Cyber 170/750—at least at our site, which retained its old core-based ECS even when it upgraded to the 750.

Incidentally, the CPU enforced access to ECS in much the same way as it did to CM. There were two registers specifying the beginning address and number of words of the single region of ECS to which the CPU had access at any time. At our site, user programs always had an ECS field length of zero. Users weren't allowed access to ECS at all because it was felt that the OS could make better use of that resource.

The 6000 CPU had a load/store architecture: data in memory could be referenced only by load and store instructions. To increment a memory location, then, you had to execute at least three instructions: load from memory, do an add, and store from memory.

Memory access was interleaved. I believe that the 6500's memory was divided into 16 independent banks, so usually the CPU did not have to wait for a memory cycle to complete before starting a new one. I think that the 750 only had 4-way interleave. This sounds like a step down from the 6500. However, it may have been unnecessary to interleave to such a high degree on the more recent 750, since it had semiconductor memory as opposed to the 6500's slower core memory.

**Registers**

In addition to the obvious program counter (P register), the 6000 Series had 24 user-accessible CPU registers. There were 3 types of registers, 8 of each type: A, B, and X. Registers of each type were numbered 0-7.

X registers were 60 bits wide and were general-purpose data registers. Most instructions operated only on X registers.

“A” registers were 18-bit address registers with a strange relationship to X registers: loading a value (let's call it m) into any register A1 - A5 would cause the CPU to load the correspondingly-numbered X register from memory location m. Loading A6 or A7 with m would cause the correspondingly-numbered X register to be stored at that location. This was the only way that data could be moved between any register and memory.

A0 was a pretty worthless register. I believe that by convention, code generated by FORTRAN kept a pointer to the beginning of the current subroutine in A0, to aid in subroutine traceback in cause an
error occurred. Similarly, X0 was not too useful, as it could neither be loaded from or stored to memory directly. However, it was moderately useful for holding intermediate results.

The B registers were index registers that could also be used for light-duty arithmetic. B registers tended to not get used a whole lot because

- They were only 18 bits wide.
- The arithmetic you could do on them was limited to addition and subtraction.
- You couldn't load or store B registers directly to or from memory. Instead, you had to go through an X register and move the contents to or from a B register.

B0 was hardwired to 0. Any attempt to set B0 was ignored by the CPU. In fact, on some CPUs, it was faster to execute a 30-bit instruction to load B0 with a constant than it was to execute two consecutive no-ops (which were 15-bit instructions). Therefore, if you had to “force upper” by 30 or more bits, it made sense to use a 30-bit load into B0. Fortunately, the assembler did force uppers automatically when necessary, so programmers were generally isolated from those details.

Many programmers felt that CDC should also have hardwired B1 to 1, since there was no increment or decrement instruction. Since there was no register hardwired to 1, many assembly language programs started with “SB1 1”, the instruction to load a 1 into B1.

### Instruction Set

Instructions in the CPU were 15 or 30 bits. The 30-bit instructions contained an 18-bit constant. Usually this was an address, but it could also be used as an arbitrary 18-bit integer. From the point of view of the instruction decoder, each 60-bit word was divided into four 15-bit instruction parcels. While up to four instructions could be packed into a 60-bit word, instructions could not be broken across word boundaries. If you needed to execute a 30-bit instruction and the current position was 45 bits into a word, you had to fill out the word with a no-op and start the 30-bit instruction at the beginning of the next word. I suspect that the 6000 Series made heavier use of its no-op instruction (46000 octal) than nearly any other machine. No-ops were also necessary to pad out a word if the next instruction was to be the target of a branch. Branches could be done only to whole-word boundaries. The act of inserting no-ops to word-align the next instruction was calling doing a “force-upper”.

There was no condition code register in the 6000 Series. Instructions that did conditional branches actually did the test and then branched on the result. This, of course, is in contrast to many architectures such as the Intel x86, which uses a condition code register that stores the result of the last arithmetic operation. When I learned about condition code registers years after first learning the 6000 architecture, I was shocked. Having a single condition code register seemed to me to be a significant potential bottleneck. It would make execution of multiple instructions simultaneously very difficult. I still think that having a single condition code register is stupid, but I must admit that the Intel Pentium Pro and successors, for instance, are pretty darned fast anyway.

The instruction set included integer (I), logical (B), and floating-point (F) instructions. The assembler syntax was different than most assemblers. There were very few different mnemonics; differentiation amongst instructions was done largely by operators. Arithmetic instructions were mostly three-address; that is, an operation was performed on two registers, with the result going to a third register. (Remember that the 6000’s load/store architecture precluded working with memory-based operands.) For instance, to add two integers in X1 and X5 and place the result in X6, you did:
A floating-point multiplication of X3 and X7, with the result going to X0, would be:

| IX6 | X1+X5 |

An Exclusive Or of X6 and X1, with the result going to X6, would be:

| FX0 | X3*X7 |

Initially, there was no integer multiply instruction. Integer multiply was added to the instruction set pretty early in the game, though, when CDC engineers figured out a way of using existing floating-point hardware to implement the integer multiply. The downside of this clever move was that the integer multiply could multiply only numbers that could fit into the 48-bit mantissa field of a 60-bit register. If your integers were bigger than 48 bits, you'd get unexpected results.

You'd think that 60-bit floating-point numbers (1 sign bit, 11-bit exponent including bias, 48-bit bit-normalized mantissa) would be large enough to satisfy anyone. Nope: the 6000 instruction set, lean as it was, did include double precision instructions for addition, subtraction, and multiplication. They operated on 60-bit quantities, just as single precision numbers; the only difference is that the double precision instructions returned a floating point number with the 48 least-significant bits, rather than the 48 most-significant bits. So, double precision operations—especially multiplication and division—required several instructions to produce the final 120-bit result. Double precision numbers were just two single precision numbers back-to-back, with the second exponent being essentially redundant. It was a waste of 12 bits, but you still got 96 bits of precision.

You can tell that floating point was important to CDC when you consider that there was separate rounding versions of the single precision operations. These were rarely used, for some reason. The non-rounding versions needed to be in the instruction set because they were required for double-precision work. The mnemonic for double precision operations was D (as in DX7 X2*X3) and for rounded operations was R.

Another instruction that is surprising to find in such a lean instruction set was Population Count. This instruction counted the number of 1 bits in a word. CX6 X2, for instance, would count the number of bits in X2 and place the result in X6. This was the slowest instruction on most 6000 machines. Rumor had it that the instruction was implemented at the request of the National Security Agency for use in cryptanalysis.

For more details, see the [CDC 6000 CPU Instruction Set](https://codex.retro1.org/cdc:6600.hardware).

### Peripheral Processors

In addition to one or two CPUs, all CDC 6000-style machines at least 10 peripheral processors (PPs). As the name implies, these were simple built-in computers with architectures oriented toward doing I/O. However, in practice, much of the operating system was also implemented in PPs.

Each PP had 4096 words of 12 bits each. These 12-bit units were often referred to as bytes, to distinguish them from the 60-bit CPU words. However, I found the terminology a bit misleading, as some people referred to the 6-bit characters used on the CDC systems as bytes. Each PP had its own
4096-byte memory. There was no way to directly access another PP's memory, though you could have two PPs talk to each other over an I/O channel.

The original 6000 machines had one bank of 10 PPs. Later machines typically had two banks of 12 for a total of 24 PPs. The fact that on some machines, PPs were divided into multiple banks was a hardware implementation issue and was not programmer-visible.

PPs had instruction sets that were reminiscent of the PDP-8 or the later Motorola 6800. There was only one data register, an 18-bit A register, and there was a 12-bit instruction pointer P. Instead of index registers and the like, the architecture provided easy access to the first 100 octal (that's 100B in CDC talk) memory locations. These were called "direct cells". Many PP instructions had a 6-bit field that referred to a direct cell, and used direct cells in much the way you'd use real registers. There was also a Q register, which was used internally and was not generally programmer-visible. I think it was used to manage multiple-word transfers to/from central memory.

One little-mentioned fact regarding peripheral processors is that they were virtual processors. A single physical arithmetic and logic unit did the actual work for 10 (or on later machines, 12) PPs. A set of 10 PPs would consists of 10 4096-byte memories and 10 sets of registers. A single ALU would service each PP in a round-robin fashion in an arrangement referred to as a “barrel”. Early 6000 machines had one PP ALU implementing 10 PPs; later machines had two implementing 12 each for a total of 24 PPs.

PPs could read and write any central memory location at will; hence, you didn't want users to be able to run their own PP programs. The simple Reference Address + Field Length memory protection enforced on the CPU did not apply to PPs. The need to read/write CM was the reason for the PP’s A register being 18 bits long. As you can imagine, computing 18-bit CM addresses on a 12-bit machine was tedious. Systems programmers had powerful assembly language macros the ease the task.

PPs did I/O by attaching to channels and performing input or output 12 bits at a time. A 6000 CPU did not have I/O instructions (though the later 7600 did). Thus, PPs were utterly crucial for running a system.

For a CPU program to do input, it would have to get a PP program to read data from a peripheral device into PP memory, and then turn around and write the data into central memory. Central memory could be written only in units of 60-bit CM words, so PPs transferred data to/from CM 5 bytes (5*12=60) at a time.

## Console Display

CDC mainframes were operated from a proprietary console that included a keyboard, a deadstart (reboot) button, and two displays. On the 6000 Series, the two displays were identical round CRTs adjacent to each other; on the Cyber 700 Series, there was one larger CRT that normally displayed the two logically distinct screens. There was a rocker switch on the 700 console to select the left screen, the right screen, or both. Displaying just the left or right screen resulted in larger characters. But the CRT was so big that we always left the display set to show both screens on the tube.

Just what was displayed on the screens was completely under program control; see DSD: Operator console and Console Commands. The console was the only device in the world whose native character set was Display Code. The number of displayable characters was about 48: the 26 upper-case letters, the 10 digits, and a few punctuation symbols. Since Display Code was a 6-bit character set, this left...
about 16 characters left over. These characters were used to implement a very simple graphics mode. In graphics mode, the only operation you could perform was to place a dot on the screen.

**Console Oddities**

The console display had two characteristics that would surprise most modern-day developers. For one thing, characters were drawn “caligraphically”. That is, unlike televisions and most modern CRTs, the screen was not scanned left-to-right and top-to-bottom. Instead, the beam was moved around to and fro in response to commands received on the I/O channel. When the console was told to draw a character, it moved the electron beam around the same way a human would move a pencil. It drew fully-formed characters, not characters made of dots.

Secondly, the console had no memory of what it had just drawn. Characters and graphics stayed on the screen only as long as the persistence of the phosphor. For a screen to stay displayed, the controlling peripheral processor had to send the same display commands again and again, constantly. This meant that in practice, you needed to have a PP completely dedicated to driving the console. Even with a dedicated PP, character-only screens typically flickered somewhat and screens containing graphics flickered extensively.

The need to constantly refresh the screen had some advantages. Areas of the screen could be made brighter, or could be made to blink, simply by refreshing them more or less often. The usual PP program that drove the console used this feature in an innovative way: When the operator had typed enough characters to uniquely identify a command, a rippling effect was created by varying the intensity of successive characters over time.

**Console Graphics**

Drawing characters and dots on the screen was done like this.

First, you would select dot mode, or character mode; if character mode, the size (small, medium, large – 8, 16, or 32 dots high).

Next you'd send a sequence of 12 bit data words. There are three possibilities:

1. data < 6000 octal: two characters, first the upper 6 bits, then the lower 6 bits, drawn at current position. Not meaningful in dot mode; I'm not sure what would happen if you tried.
2. data = 6xxx: set X coordinate to xxx.
3. data = 7yyy: set Y coordinate to yyy. In dot mode, draw a dot at the resulting X/Y. (A dot is simply the character ".")

The fact that the only graphics-mode operation was to draw a dot meant that graphics performance was very poor, and graphics were barely used at all.

**Keyboard input**

The console had a limited keyboard containing the 26 alphabetic characters, 10 digits, and a very few punctuation characters, not much more than . , + - ( ) The + and - keys were used like PageUp and PageDown keys. There were also two special keys named “Left Blank” and “Right Blank”. By
convention, Left Blank was used to clear the current keyboard entry and any error messages. Right Blank was used to advance the left screen display sequence established by the DSD SET command.

The hardware interface to the keyboard was quite simplistic and required polling by the controlling PP. Each key generates a 6 bit code in the range 1 to 62 octal. The PP would look for keystrokes by selecting input mode, then reading from the display channel. This would produce a 12 bit data word. If zero, it means no key is depressed. Otherwise, the PP would see the OR of the keycodes for the currently depressed keys. (So if you press A and B both, you get 0003 – the keycode for C.) Usually this was a nuisance, but in some games it was used to good advantage.

**Deadstart Panel**

The 6000 machines did not have a PC-style ROM BIOS, much less a separate mainframe control processor as do modern big machines. When the machine was booted, all it had was whatever very short program could be entered on the deadstart panel. The deadstart panel contained rows of 12 on/off switches, each row corresponding to one PP word. There were 12 rows on the 6500 panel, and 16 on the 750 panel. When a machine was deadstarted (booted), the contents of the deadstart panel were read into the memory of peripheral processor 0, starting at location 1. Control was then given to PP 0. There was a switch to control which of two PPs was to be number zero. This allowed you to boot even if one PP went bad.

The deadstart panel was hidden away inside the machine. It wasn't often necessary to change the deadstart program that had been toggled in. CDC's design of the panel, with enough switches for an entire small program, made deadstarting much easier than on, say, some DEC PDP models. On those machines, you had to toggle in a bootup program one word at a time and press “Enter” to enter the word. With CDC, you just set it and forgot it. (Though some diagnostic programs run by the CDC customer engineers may have required a different deadstart panel configuration.) It was bad news if the switches themselves went bad. The 6500 switches were large toggle switches that snapped into place with a satisfying click, but the 750 switches were small flimsy ones, and once a switch went bad on us. Ouch!