Lecture #33 – April 30, 2004

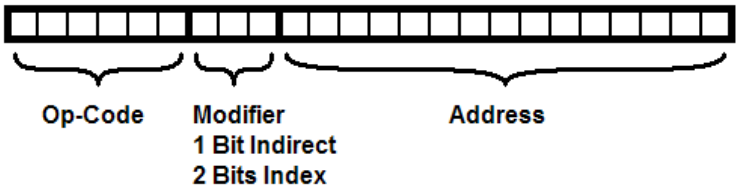
The CDC-3300 and 6000 Series

Continuing on in "Bizarre Architectures Week" we look at two machines from the 1960s, both strongly influenced by Seymour Cray before he left Control Data Corporation (CDC) to form his own company. Both machines were very powerful computers, and were considered to be supercomputers in their day. By today's standards they are very underpowered. Part of the lesson of examining these two machines is that neither observes the "standard" design rules implicit in essentially all modern machines. Modern processors all use a word that is eight bits or an even power of eight bits in length, all use two's complement binary arithmetic with occasional BCD augmentation, and except for some machines that still use EBCDIC nearly all use ASCII or the Unicode superset of ASCII as their character set. While these design rules are implicit today, early machines were designed with very different criteria. Only as alternatives were explored and discarded over time did we converge on the current approaches.

The CDC-3300

For example, the CDC-3300 was a 24-bit architecture, using *one's complement* binary arithmetic, with two accumulators labeled A and Q, and two or three banks of 32K of 24-bit magnetic core memory. Accumulator A was the main accumulator, and Q was a "quotient register" partially dedicated to handling multiplications and divisions. In addition, there were four 15-bit ***index registers*** used for memory offsets into arrays (actually, there were only three physical index registers R1, R2, and R3; "register" R0 didn't actually exist and therefore always had the value zero).

Instructions observed a relatively fixed format: 6 bits for Op-Code, 3 bits for the modifier field (containing 1 bit for indirect memory references and 2 bits to specify the index register), and 15 bits for the address. Any instruction could reference any location in memory, and the index registers were large enough to treat the entire 32K address space in any memory bank as one large array.



An effective address is formed by first extracting the 15-bit address field from the instruction, and then adding to that address field the contents of the specified index register. This is why R0 is always zero; an absolute address uses R0 by default. The address so formed is the final effective address of the desired operand only if the Indirect bit is 0, but if the Indirect bit equals 1 the *contents of memory* at the specified address are used as the final address of the desired operand.

Instructions sequences to add or subtract two numbers used the A accumulator most of the time, but they could also use the Q register independently, or in a 48-bit mode in combination with accumulator A. Thus, adding two numbers for the high-level expression `C := A + B` would take one of the following three forms:

24-Bit using A	**24-Bit using Q**	**48-Bit using AQ**
`LDA A`	`LDQ A`	`LDAQ A`
`ADA B`	`ADQ B`	`ADAQ B`
`STA C`	`STA C`	`STAQ C`

Arithmetic used one's complement binary, so negating a number was as simple as inverting all of the bit values. While this representation has exactly the same number of negative values as positive values, it also generates the unfortunate side effect that both 000000000000000000000000 and 111111111111111111111111 are representations for zero. The arithmetic unit had to detect the "negative zero" case and convert it to "positive zero" appropriately.

Characters were six bits in length, not eight, and so each 24-bit word could contain exactly four characters. Letters were limited to upper-case only, as there were not enough bit patterns in 6 bits ($2^6 = 64$ values) to represent both upper and lower case letters and still leave room for anything else. Once you force 26 upper case letters, 26 lower case letters, and 10 digits into the character set, you have just enough space left for the blank and the period!

The 3300 could execute around one million instructions per second and was considered a supercomputer in 1965. One was installed at Oregon State University, and until the late 1970s ran a homegrown operating system called OS3 (Oregon State Open Shop Operating System). At its peak it could handle up to about 80 simultaneous users, all editing, compiling, and running BASIC or FORTRAN programs via 10-character-per-second Teletype™ printing terminals.

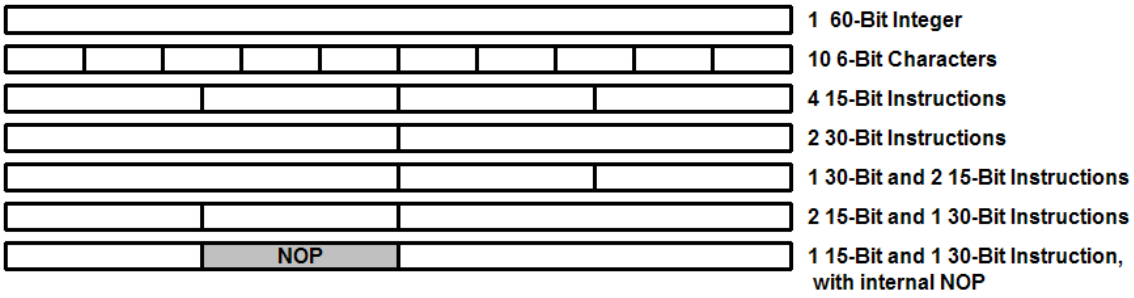
The CDC-6000 Series

The Control Data Corporation 6000 series consisted of the 6400, 6500, 6600, and 6700 models, which evolved into the Cyber 7000 series. Only a few dozen such machines were ever constructed. They were designed as heavy-duty, one-program-at-a-time "number-crunchers" for scientific processing but were forced into service in many installations as general-purpose time-sharing machines, an application for which they were particularly ill suited.

The basic architecture of the machines consisted of a central processing core surrounded by ten (or twenty) ***peripheral processing units*** (PPUs). The CPU did all the computational "heavy lifting" but was not capable of performing any input/output on its own. The PPUs were small independent computers similar in architecture to the PDP-8 (12-bit arithmetic, 4K of memory) that could access the main memory at the same time as the CPU. The PPUs would leave input values in main memory for the CPU to find, and would extract for output any values left in main memory by the CPU.

The CPU was a one's complement, 60-bit architecture, using multiple registers (more on that later) and 256K 60-bit words of primary memory. Instructions were either 15 bits or 30 bits in length, depending on whether or not they referenced memory. The 15-bit instructions were very similar to those on the ARM in that they specified two operand registers and a result register. The 30-bit instructions contained space to specify two registers and a single 18-bit memory address. A 60-bit memory word could therefore contain four 15-bit instructions, two 30-bit instructions, two 15-bit instructions and one 30-bit instruction, or one 30-bit instruction and two 15-bit instructions. You could not specify a 30-bit instruction to be between two 15-bit instructions, however, and this occasionally required the insertion of a 15-bit `NOP` (No Operation) instruction into the code to align the 30-bit instruction with the proper half of the word.

Similarly, a 60-bit word could hold one long integer, one floating point number, or ten 6-bit characters (not exactly the same character set as on the 3300, but subject to the same upper-case-only constraints). The 18-bit addresses could reference any 60-bit word in the 256K address space.



There were eight 60-bit X registers for computations, along with eight 18-bit A registers for addresses and eight 18-bit B registers for array offsets. The A and X registers operated in concert for loading values from or storing values into memory; setting A0 through A5 to any address value loaded the corresponding X register from memory at the specified address, and setting A6 or A7 to any address value stored the corresponding X register into memory. This may seem a bit odd, but it allowed arithmetic on addresses to be separate from arithmetic on data. Incrementing A3 by 1 several times, for example, stepped through an array in memory by successively creating new addresses into A3, each with the side effect of loading the corresponding memory location into X3. Memory reference instructions were always 30 bits in length, each containing a 6-bit Op-Code, two 3-bit register specifiers, and an 18-bit address.

The B registers were used as index registers, where the value in a B register was added to the address in an A register to obtain the final effective address. For absolute addressing purposes B0 always contained zero (similar to the situation on the 3300). Arithmetic operations were always performed between X registers, similar to the situation on the ARM. These instructions were always 15 bits in length, each containing a 6-bit Op-Code and three 3-bit register specifiers (two source registers and one destination, similar to the three-address instructions on the ARM).

Our high-level code for adding two numbers together, `C := A + B`, would look as follows on the 6000 series (if I remember the Op-Codes correctly):

```
SA1 A     Set A1 to A's address, load  X1 with Memory[A1]
SA2 B     Set A2 to B's address, load  X2 with Memory[A2]
RX6 X1+X2 Compute X6 := X1 + X2
SA6 C     Set A6 to C's address, store X6 into Memory[A6]
```

This instruction sequence takes up only two words of memory. The first word contains the two 30-bit load instructions. The second word contains the 15-bit computation instruction and the 30-bit store instruction, but in order to align the 30-bit instruction properly the assembler must insert a 15-bit `NOP` instruction between the `RX6` and the `SA6` instructions. Notice that the instruction sequence has two loads and one store; this is why three times as many of the A registers caused loads to happen rather than stores.

ASCII-based terminal equipment and primitive batch-mode word processors were becoming popular towards the end of the lifetime of the 6000 series; support for lower case characters was implemented by prefixing each lower case letter with an up-carat character. Native upper case only files were unaffected by this approach, but mixed case text files were roughly twice as large. For example, the phrase "`This is a test.`" would be stored as "`T^H^I^S ^I^S ^A ^T^E^S^T.`" in the file. The PPUs would have to translate between this form and ASCII on both input and output.

It was expected that once a program started running that it would run to completion without interruption, so there was no initial support for timesharing or switching between different programs. Many early 6000 series machines were ***batch*** only; reading in decks of punched cards one at a time, compiling and running the associated program, printing the results, and flushing the program from memory. The timesharing approach that was taken in the Kronos and NOS operating systems was to have the PPUs switch from one program to another by saving the *entire* program memory and the contents of all registers to disk, called a ***rollout*** of the program. Once one program was rolled out another was rolled in, its register values were restored, and it started running. As each program's time slice expired, determined by the PPUs, it was rolled out to disk and the next program rolled in for its time slice. This approach is dreadfully inefficient, but it worked well enough and the machines were fast enough that timesharing operating systems would support a few dozen simultaneous users.

Conclusions

For their day, these machines were among the most powerful computers on the planet. At the time numerical problems were paramount, so these machines were designed to support high-speed computations at the expense of text or string processing. You can get a feeling for this from the decision to use a 6-bit character set without any native support for lower case characters.

Both 24-bit words on the 3300 (48-bits in double precision) and 60-bit words on the 6000 series seem very strange sizes to us today, but they were certainly adequate for the problems they were solving. What seems most strange to me today is the choice of one's complement arithmetic instead of two's complement.

Despite their peculiarities, the 3300 and the 6000 series were relatively clean architectures (contrast them with the Intel 80x86 line), and the 6000 series had some early elements of what we consider today to be RISC designs. While none of these machines are in operation today, as far as I know, they still have some lessons for us in alternative approaches to successful machine design.