# Chapter 39

# Parallel operation in the Control Data 6600[1]

*James E. Thornton*

## History

In the summer of 1960, Control Data began a project which culminated October, 1964 in the delivery of the first 6600 Computer. In 1960 it was apparent that brute force circuit performance and parallel operation were the two main approaches to any advanced computer.

This paper presents some of the considerations having to do with the parallel operations in the 6600. A most important and fortunate event coincided with the beginning of the 6600 project. This was the appearance of the high-speed silicon transistor, which survived early difficulties to become the basis for a nice jump in circuit performance.

## System organization

The computing system envisioned in that project, and now called the 6600, paid special attention to two kinds of use, the very large scientific problem and the time sharing of smaller problems. For the large problem, a high-speed floating point central processor with access to a large central memory was obvious. Not so obvious, but important to the 6600 system idea, was the isolation of this central arithmetic from any peripheral activity.

It was from this general line of reasoning that the idea of a multiplicity of peripheral processors was formed (Fig. 1). Ten such peripheral processors have access to the central memory on one side and the peripheral channels on the other. The executive control of the system is always in one of these peripheral processors, with the others operating on assigned peripheral or control tasks. All ten processors have access to twelve input-output channels and may "change hands," monitor channel activity, and perform other related jobs. These processors have access to central memory, and may pursue independent transfers to and from this memory.

Each of the ten peripheral processors contains its own memory for program and buffer areas, thereby isolating and protecting the more critical system control operations in the separate processors. The central processor operates from the central memory with relocating register and file protection for each program in central memory.

## Peripheral and control processors

The peripheral and control processors are housed in one chassis of the main frame. Each processor contains 4096 memory words of 12 bits length. There are 12- and 24-bit instruction formats to provide for direct, indirect, and relative addressing. Instructions provide logical, addition, subtraction, shift, and conditional branching. Instructions also provide single word or block transfers to and from any of twelve peripheral channels, and single word or block transfers to and from central memory. Central memory words of 60 bits length are assembled from five consecutive peripheral words. Each processor has instructions to interrupt the central processor and to monitor the central program address.

To get this much processing power with reasonable economy and space, a time-sharing design was adopted (Fig. 2). This design contains a register "barrel" around which is moving the dynamic information for all ten processors. Such things as program address, accumulator contents, and other pieces of information totalling 52 bits are shifted around the barrel. Each complete trip around requires one major cycle or one thousand nanoseconds. A "slot" in the barrel contains adders, assembly networks, distribution network, and interconnections to perform one step of any peripheral instruction. The time to perform this step or, in other words, the time through the slot, is one minor cycle or one hundred nanoseconds. Each of the ten processors, therefore, is allowed one minor cycle of every ten to perform one of its steps. A peripheral instruction may require one or more of these steps, depending on the kind of instruction.

In effect, the single arithmetic and the single distribution and assembly network are made to appear as ten. Only the memories are kept truly independent. Incidentally, the memory read-write cycle time is equal to one complete trip around the barrel, or one thousand nanoseconds.

[1] *AFIPS Proc. FJCC*, pt. 2 vol. 26, pp. 33–40, 1964.

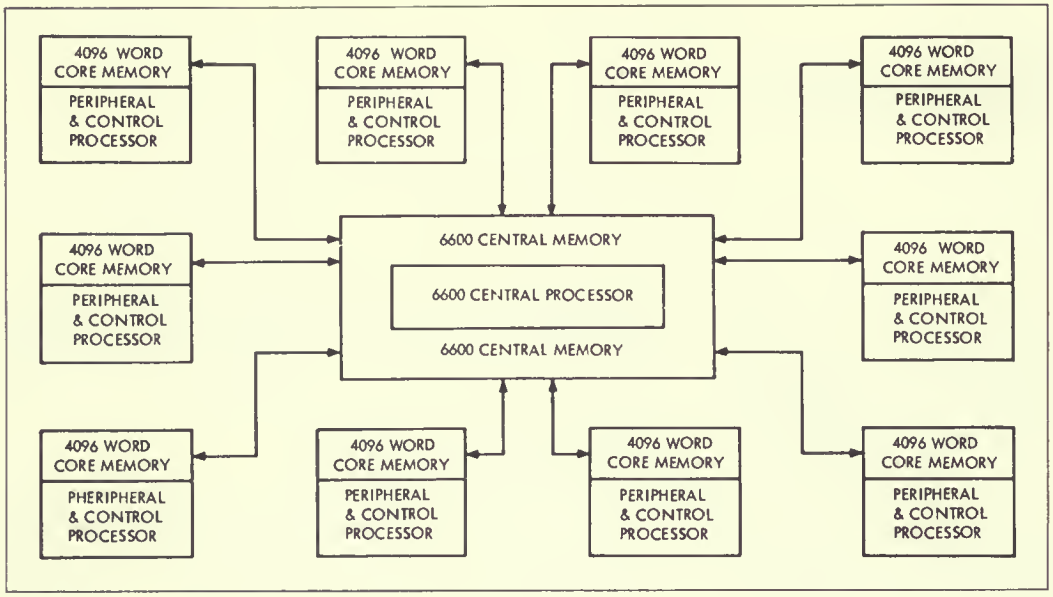


Fig. 1. Control Data 6600.

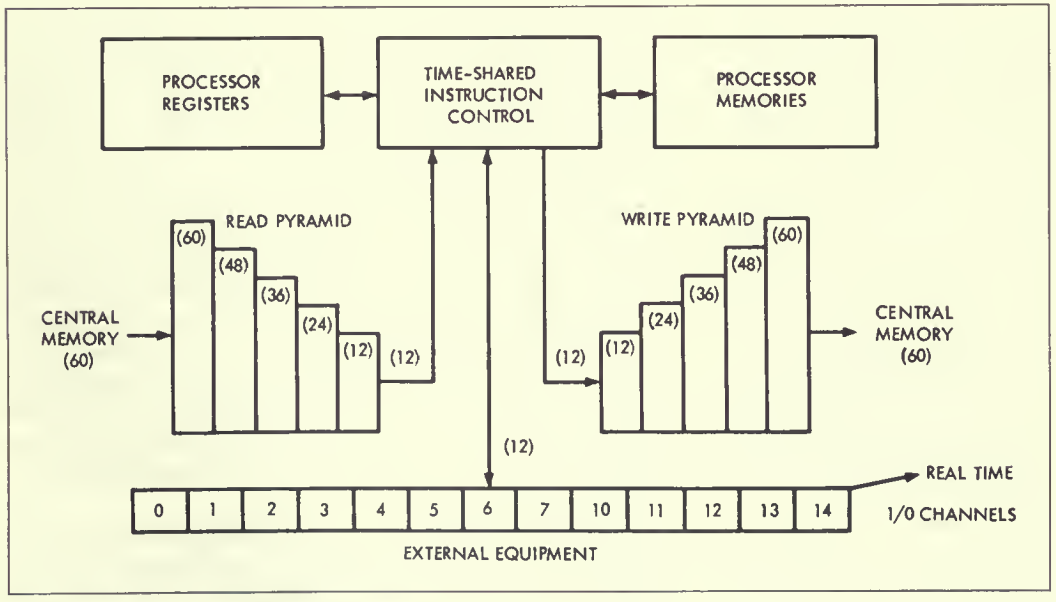


Fig. 2. 6600 peripheral and control processors.

Input-output channels are bi-directional, 12-bit paths. One 12-bit word may move in one direction every major cycle, or 1000 nanoseconds, on each channel. Therefore, a maximum burst rate of 120 million bits per second is possible using all ten peripheral processors. A sustained rate of about 50 million bits per second can be maintained in a practical operating system. Each channel may service several peripheral devices and may interface to other systems, such as satellite computers.

Peripheral and control processors access central memory through an assembly network and a dis-assembly network. Since five peripheral memory references are required to make up one central memory word, a natural assembly network of five levels is used. This allows five references to be "nested" in each network during any major cycle. The central memory is organized in independent banks with the ability to transfer central words every minor cycle. The peripheral processors, therefore, introduce at most about 2% interference at the central memory address control.

A single real time clock, continuously running, is available to all peripheral processors.

### Central processor

The 6600 central processor may be considered the high-speed arithmetic unit of the system (Fig. 3). Its program, operands, and results are held in the central memory. It has no connection to the peripheral processors except through memory and except for two single controls. These are the exchange jump, which starts or interrupts the central processor from a peripheral processor, and the central program address which can be monitored by a peripheral processor.

A key description of the 6600 central processor, as you will see in later discussion, is "parallel by function." This means that a number of arithmetic functions may be performed concurrently. To this end, there are ten functional units within the central

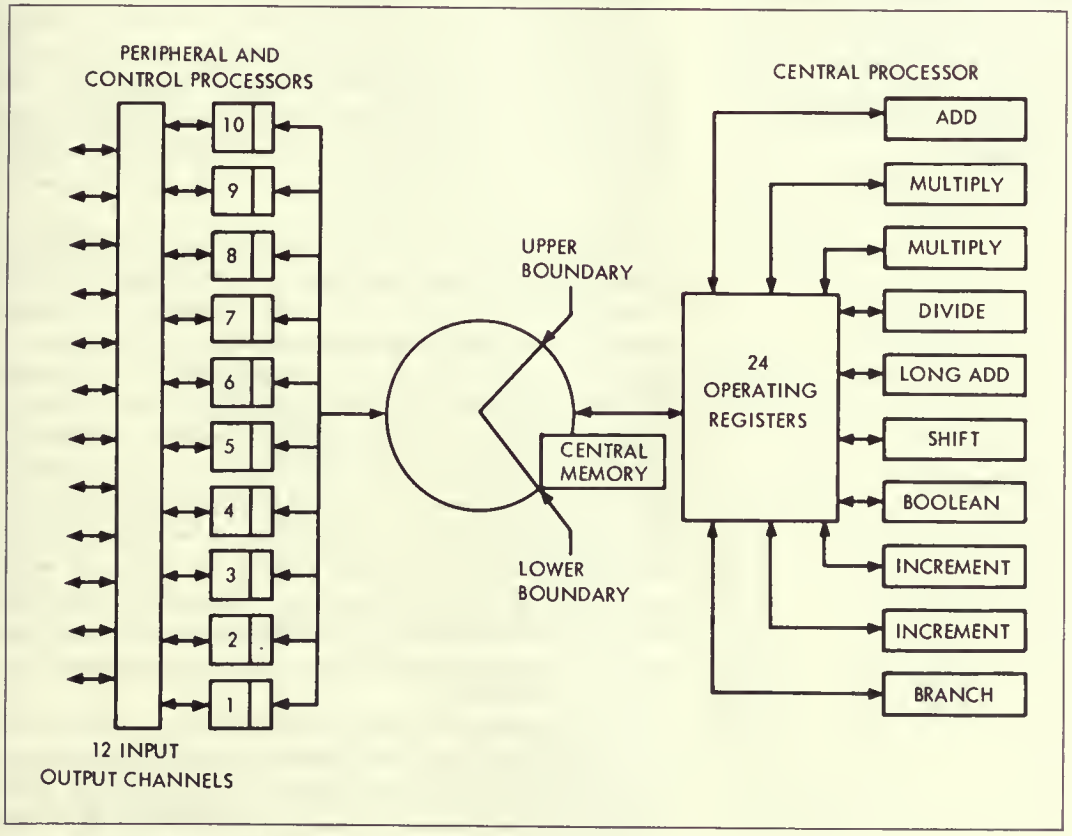


Fig. 3. Block diagram of 6600.

processor. These are the two increment units, floating add unit, fixed add unit, shift unit, two multiply units, divide unit, boolean unit, and branch unit. In a general way, each of these units is a three address unit. As an example, the floating add unit obtains two 60-bit operands from the central registers and produces a 60-bit result which is returned to a register. Information to and from these units is held in the central registers, of which there are twenty-four. Eight of these are considered index registers, are of 18 bits length, and one of which always contains zero. Eight are considered address registers, are of 18 bits length, and serve to address the five read central memory trunks and the two store central memory trunks. Eight are considered floating point registers, are of 60 bits length, and are the only central registers to access central memory during a central program.

In a sense, just as the whole central processor is hidden behind central memory from the peripheral processors, so, too, the ten functional units are hidden behind the central registers from central memory. As a consequence, a considerable instruction efficiency is obtained and an interesting form of concurrency is feasible and practical. The fact that a small number of bits can give meaningful definition to any function makes it possible to develop forms of operand and unit reservations needed for a general scheme of concurrent arithmetic.

Instructions are organized in two formats, a 15-bit format and a 30-bit format, and may be mixed in an instruction word (Fig. 4). As an example, a 15-bit instruction may call for an ADD,

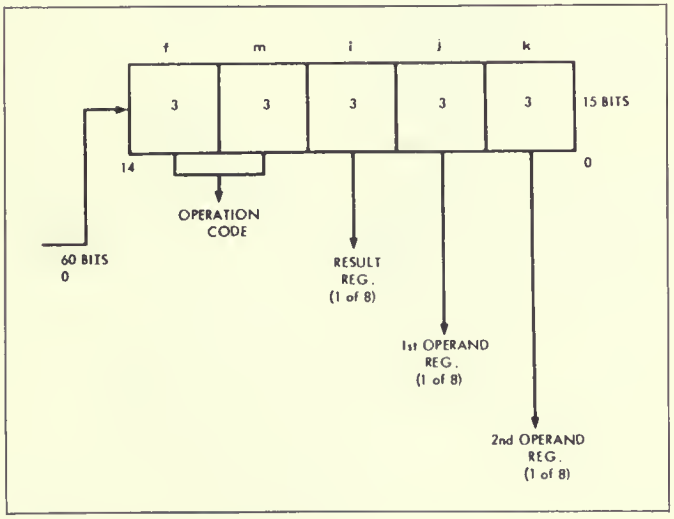


Fig. 4. Fifteen-bit instruction format.

designated by the $f$ and $m$ octal digits, from registers designated by the $j$ and $k$ octal digits, the result going to the register designated by the $i$ octal digit. In this example, the addresses of the three-address, floating add unit are only three bits in length, each address referring to one of the eight floating point registers. The 30-bit format follows this same form but substitutes for the $k$ octal digit an 18-bit constant K which serves as one of the input operands. These two formats provide a highly efficient control of concurrent operations.

As a background, consider the essential difference between a general purpose device and a special device in which high speeds are required. The designer of the special device can generally improve on the traditional general purpose device by introducing some form of concurrency. For example, some activities of a housekeeping nature may be performed separate from the main sequence of operations in separate hardware. The total time to complete a job is then optimized to the main sequence and excludes the housekeeping. The two categories operate concurrently.

It would be, of course, most attractive to provide in a general purpose device some generalized scheme to do the same kind of thing. The organization of the 6600 central processor provides just this kind of scheme. With a multiplicity of functional units, and of operand registers and with a simple and highly efficient addressing system, a generalized queue and reservation scheme is practical. This is called the *scoreboard*.

The scoreboard maintains a running file of each central register, of each functional unit, and of each of the three operand trunks to and from each unit. Typically, the scoreboard file is made up of two-, three-, and four-bit quantities identifying the nature of register and unit usage. As each new instruction is brought up, the conditions at the instant of issuance are set into the scoreboard. A snapshot is taken, so to speak, of the pertinent conditions. If no waiting is required, the execution of the instruction is begun immediately under control of the unit itself. If waiting is required (for example, an input operand may not yet be available in the central registers), the scoreboard controls the delay, and when released, allows the unit to begin its execution. Most important, this activity is accomplished in the scoreboard and the functional unit, and does not necessarily limit later instructions from being brought up and issued.

In this manner, it is possible to issue a series of instructions, some related, some not, until no functional units are left free or until a specific register is to be assigned more than one result. With just those two restrictions on issuing (unit free and no double result), several independent chains of instructions may proceed concurrently. Instructions may issue every minor cycle in the

absence of the two restraints. The instruction executions, in comparison, range from three minor cycles for fixed add, 10 minor cycles for floating multiply, to 29 minor cycles for floating divide.

To provide a relatively continuous source of instructions, one buffer register of 60 bits is located at the bottom of an instruction stack capable of holding 32 instructions (Fig. 5). Instruction words from memory enter the bottom register of the stack pushing up the old instruction words. In straight line programs, only the bottom two registers are in use, the bottom being refilled as quickly as memory conflicts allow. In programs which branch back to an instruction in the upper stack registers, no refills are allowed after the branch, thereby holding the program loop completely in the stack. As a result, memory access or memory conflicts are no longer involved, and a considerable speed increase can be had.

Five memory trunks are provided from memory into the central processor to five of the floating point registers (Fig. 6). One address register is assigned to each trunk (and therefore to the floating point register). Any instruction calling for address register result implicitly initiates a memory reference on that trunk. These instructions are handled through the scoreboard and therefore tend to overlap memory access with arithmetic. For example, a new memory word to be loaded in a floating point register can be brought in from memory but may not enter the register until all previous uses of that register are completed. The central registers, therefore, provide all of the data to the ten functional units, and receive all of the unit results. No storage is maintained in any unit.

Central memory is organized in 32 banks of 4096 words. Consecutive addresses call for a different bank; therefore, adjacent addresses in one bank are in reality separated by 32. Addresses may be issued every 100 nanoseconds. A typical central memory information transfer rate is about 250 million bits per second.

As mentioned before, the functional units are hidden behind the registers. Although the units might appear to increase hardware duplication, a pleasant fact emerges from this design. Each unit may be trimmed to perform its function without regard to others. Speed increases are had from this simplified design.

As an example of special functional unit design, the floating multiply accomplishes the coefficient multiplication in nine minor cycles plus one minor cycle to put away the result for a total of 10 minor cycles, or 1000 nanoseconds. The multiply uses layers of carry save adders grouped in two halves. Each half concurrently forms a partial product, and the two partial products finally merge while the long carries propagate. Although this is a fairly large complex of circuits, the resulting device was sufficiently smaller than originally planned to allow two multiply units to be included in the final design.

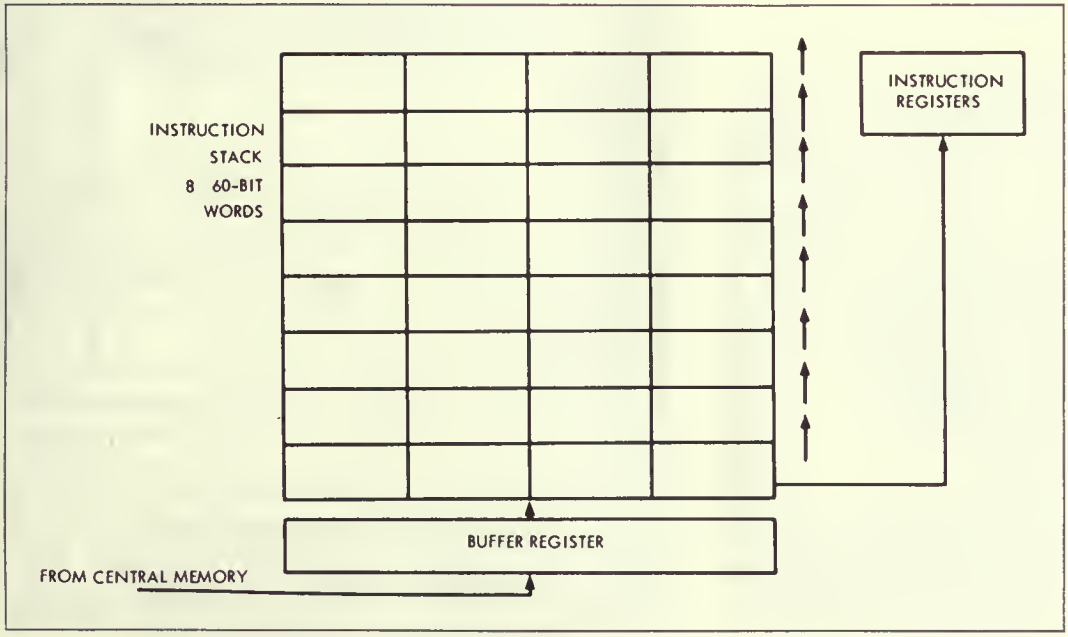


Fig. 5. 6600 instruction stack operation.

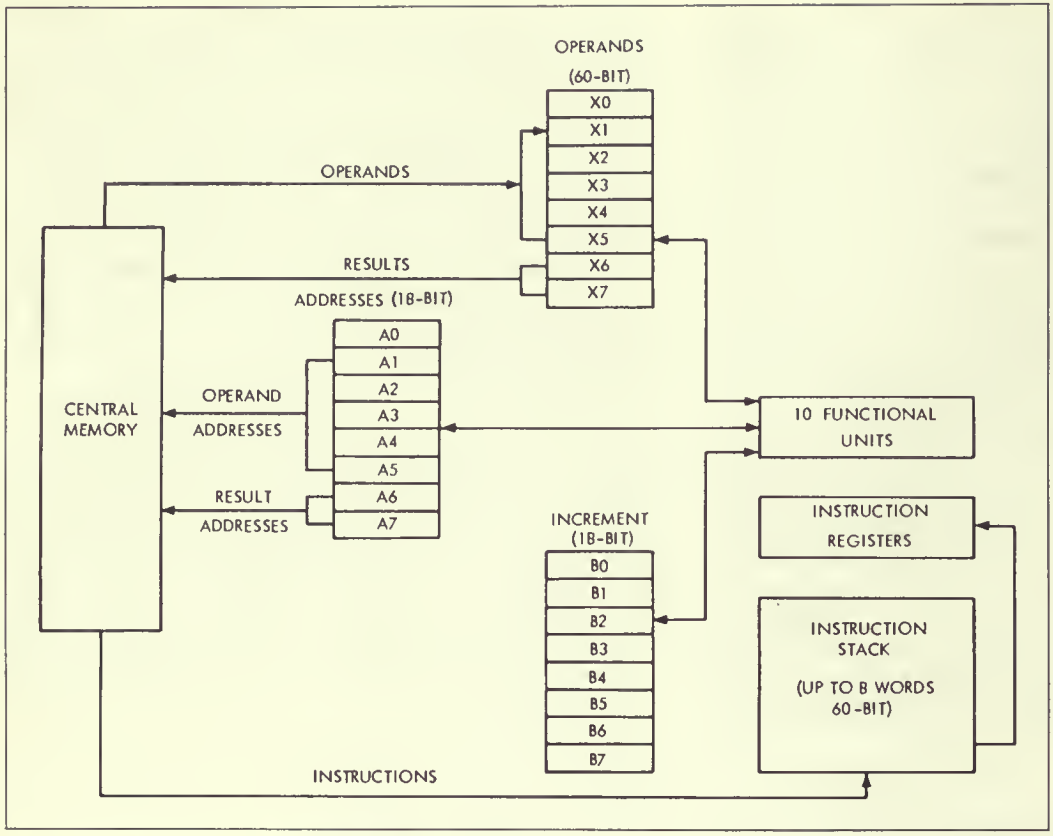


**Fig. 6. Central processor operating registers.**

To sum up the characteristics of the central processor, remember that the broadbrush description is "concurrent operation." In other words, any program operating within the central processor utilizes some of the available concurrency. The program need not be written in a particular way, although centainly some optimization can be done. The specific method of accomplishing this concurrency involves *issuing* as many instructions as possible while handling most of the conflicts during *execution*. Some of the essential requirements for such a scheme include:

1 Many functional units

2 Units with three address properties

3 Many transient registers with many trunks to and from the units

4 A simple and efficient instruction set

### Construction

Circuits in the 6600 computing system use all-transistor logic (Fig. 7). The silicon transistor operates in saturation when switched "on" and averages about five nanoseconds of stage delay. Logic circuits are constructed in a cordwood plug-in module of about $2\frac{1}{2}$ inches by $2\frac{1}{2}$ inches by 0.8 inch. An average of about 50 transistors are contained in these modules.

Memory circuits are constructed in a plug-in module of about six inches by six inches by $2\frac{1}{2}$ inches (Fig. 8). Each memory module contains a coincident current memory of 4096 12-bit words. All read-write drive circuits and bit drive circuits plus address translation are contained in the module. One such module is used for each peripheral processor, and five modules make up one bank of central memory.

Logic modules and memory modules are held in upright hinged chassis in an X shaped cabinet (Fig. 9). Interconnections between modules on the chassis are made with twisted pair transmission

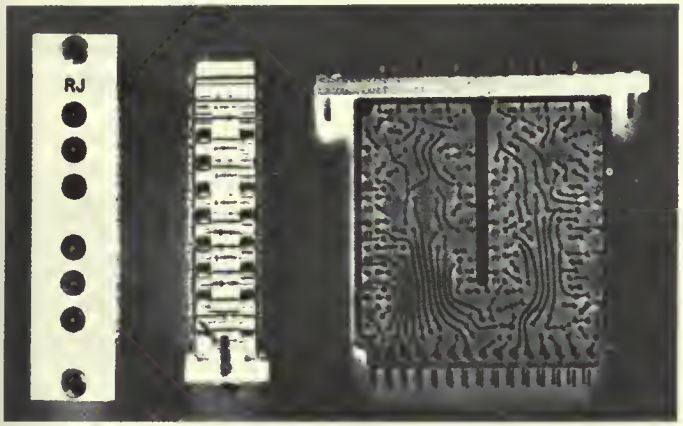


Fig. 7. 6600 printed circuit module.

lines. Interconnections between chassis are made with coaxial cables.

Both maintenance and operation are accomplished at a programmed display console (Fig. 10). More than one of these consoles may be included in a system if desired. Dead start facilities bring

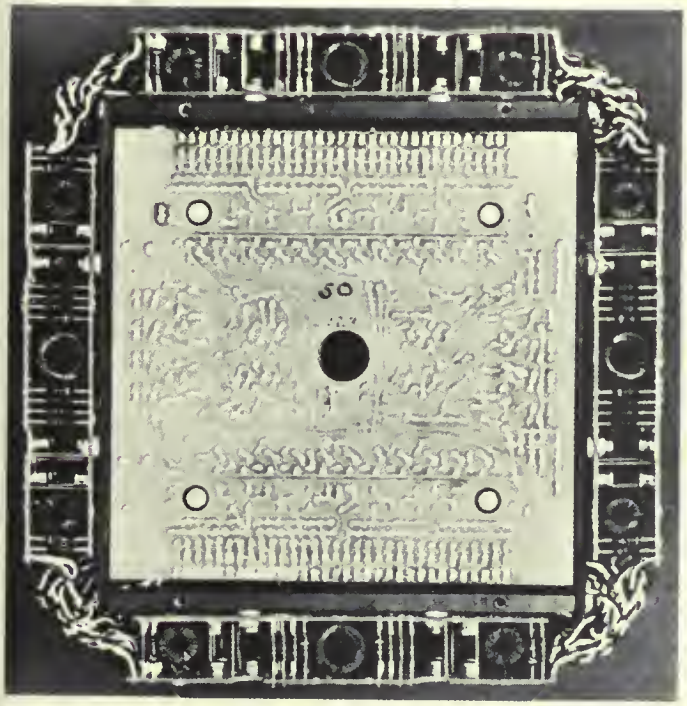

Fig. 8. 6600 memory module.


Fig. 9. 6600 main frame section.

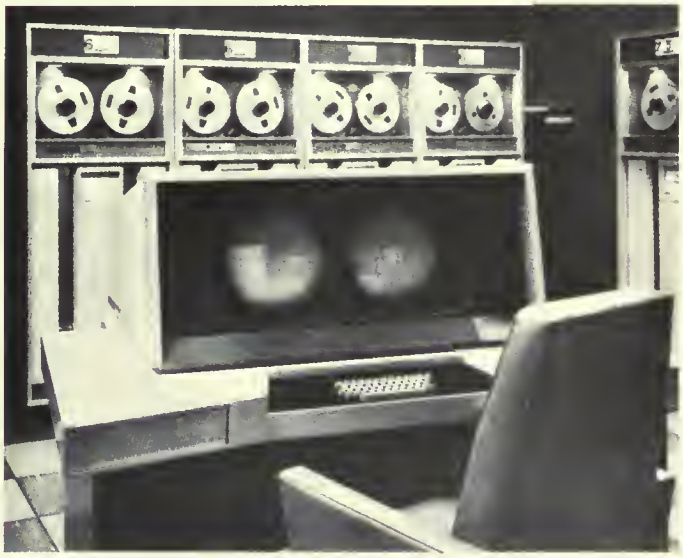

Fig. 10. 6600 display console.

the ten peripheral processors to a condition which allows information to enter from any chosen peripheral device. Such loads normally bring in an operating system which provides a highly sophisticated capability for multiple users, maintenance, and so on.

The 6600 Computer has taken advantage of certain technology advances, but more particularly, logic organization advances which now appear to be quite successful. Control Data is exploring advances in technology upward within the same compatible structure, and identical technology downward, also within the same compatible structure.

## References

AllaR64; ClayB64

## APPENDIX 1 CDC 6400, 6500, 6600 CENTRAL PROCESSOR ISP DESCRIPTION

```
                                Appendix I

        CDC 6400, 6500, 6600 Central Processor ISP Description

Pc State
    P<17:0>                                  Program counter
    X[0:7]<59:0>                             Main arithmetic registers. X[1:5], are implicitly loaded from
    A[0:7]<17:0>                              Mp when A[1:5] are loaded.  X[6:7] are implicitly stored in
                                              Mp when A[6:7] are loaded.
    B[0]<17:0>  := 0                         B registers are general arithmetic registers, and can be used
    B[1:7]<17:0>                              as index registers.
    Run                                      1 if interpreting instructions, not under program control.
    EM<17:0>                                 Exit mode bits
        Address␣out␣of␣range␣mode  := EM<12>
        Operand␣out␣of␣range␣mode  := EM<13>
        Indefinite␣operand␣mode    := EM<14>
 The above description is incomplete in that the above 3 mode's alarm allow conditions to trap Pc at Mp[RA].  Trapping occurs if
 an alarm condition occurs "and" the mode is a one.

Mp State
    Mp[0:777777₈]<59:0>                      main core memory of 2^18 w, (256 kw)
    Ms[0:2015232]<59:0>                      ECS/Extended Core Storage Program can only transfer data between
                                              Mp and Ms.  Program cannot be executed in Ms.
    RA<17:0>                                 reference (or relocation) address register to map a logical Mp'
                                              into physical Mp
    FL<17:0>                                 field length - the bounds register which limits a program's
                                              access to a range of Mp'
    RAECS<59:36>                             reference or relocation register for Ms(Extended Core Storage)
    FLECS<59:36>                             field length for ECS
    Address␣out␣of␣range                     a bit denoting a state when memory mapping is invalid

Memory Mapping Process
 This process maps or relocates a logical program, at location Mp', and Ms',into physical Mp and Ms.
    Mp'[X]  := ((X < FL) →Mp[X + RA]);       logical Mp'
               (X ≥ FL) →(Run ←0; Address␣out␣of␣range ←1))
    Ms'[X]  := ((X < FLECS) →Ms[X]+ RAECS]); logical Ms'
               (X ≥ FLECS) →(Run ←0; Address␣out␣of␣range ←1))
Exchange jump storage allocation map at location, n within Mp:
 The following Mp" array is reserved when Pc state is stored, and switched to another job.  The exchange jump instruction in
 a Peripheral and Control Processor enacts the operation:  (Mp" ← Mp; Mp ← Mp").
    Mp"[n]<53:0>     := P□A[0]□000000₈
    Mp"[n+1]<53:0>   := RA□A[1]□B[1]
    Mp"[n+2]<53:0>   := FL□A[2]□B[2]
    Mp"[n+3]<53:0>   := EM□A[3]□B[3]
    Mp"[n+4]         := RAECS□A[4]□B[4]
    Mp"[n+5]         := FLECS□A[5]□B[5]
    Mp"[n+6]<35:0>   := A[6]□B[6]
    Mp"[n+7]<35:0>   := A[7]□B[7]
    Mp"[n+10₈:n+17₈]:= X[0:7]
```

```
Instruction Format
   instruction<29:0>                                  although 30 bits, most instructions are 15 bits; see
                                                       Instruction Interpretation Process
      fm<5:0>          := instruction<29:24>          operation code or function
      fmi<8:0>         := fm□i                        extended op code
      i<2:0>           := instruction<23:21>          specifies a register or an extension to op code
      J<2:0>           := instruction<20:18>          specifies a register
      k<2:0>           := instruction<17:15>          specifies a register
      jk<5:0>          := j□k                         a shift constant (6 bits)
      K<17:0>          := instruction<17:0>           an 18 bit address size constant
      long␣instruction := ((fm < 10₈) ∨               30 bit instruction
                           (50 ≤ fm < 53) ∨
                           (60 ≤ fm < 63) ∨
                           (70 ≤ fm < 73))
   short␣instruction   := ¬ long instruction          15 bit instruction

Instruction Interpretation Process
 A 15 bit (short) or 30 bit (long) instruction is fetched from Mp'[P]<p x 15 + 15 - 1:p x 15> where p = 3, 2, 1, or 0.  A 30
 bit instruction cannot be stored across word boundaries (or in 2, Mp' locations).
   p<1>₄                                              a pointer to 15 bit quarter word which has instruction
   Run → (instruction<29:15> ←Mp'[P]<(p x 15 + 14):(p x 15)>; next     fetch
      p ←p - 1; next
      (p = 0) ∧ long␣instruction →Run ←0;
      (p ≠ 0) ∧ long␣instruction → (
          instruction<14:0> ←Mp'[P]<(p x 15 + 14):(p x 15)>;
           p ←p - 1); next
      Instruction␣execution; next                                  execute
      (p = 0) → (p ←3; P ←P + 1))

Instruction Set and Instruction Execution Process
 Operand fetches or stores between Mp' and X[i] occur by loading or storing registers A[i].  If (0 < i < 6) a fetch from
 Mp'[A[i]] occurs.  If (i ≥ 6) a store is made to Mp'[A[i]].  The description does not describe Address␣out␣of␣range case,
 which is treated like a null operation.
   Instruction␣execution := (
   Set A[i]/SA
      "SAi Aj + K"  (fm = 50) → (A[i] ←A[j] + K; next Fetch␣Store);
      "SAi Bj + K"  (fm = 51) →(A[i] ←B[j] + K; next Fetch␣Store);
      "SAi Xj + K"  (fm = 52) →(A[i] ←X[j]<17:0> + K; next Fetch␣Store);
      "SAi Xj + Bk" (fm = 53) →(A[i] ←X[j]<17:0> + B[k]; next Fetch␣Store);
      "SAi Aj + Bk" (fm = 54) →(A[i] ←A[j] + B[k]; next Fetch␣Store);
      "SAi Aj - Bk" (fm = 55) →(A[i] ←A[j] - B[k]; next Fetch␣Store);
      "SAi Bj + Bk" (fm = 56) →(A[i] ←B[j] + B[k]; next Fetch␣Store);
      "SAi Bj - Bk" (fm = 57) →(A[i] ←B[j] - B[k]; next Fetch␣Store);
      Fetch␣Store := (
         (0 < i < 6) → (X[i] ←Mp'[A[i]]);                  process to get operand in X or store operand from X when A
         (i ≥ 6) → (Mp'[A[i] ← X[i]))                       is written
   Operations on B and X
   Set B[i]/SBi
      "SBi Aj + K" (fm = 60) → (B[i] ← A[j] + K);
```

```
    "SBi Bj + K" (fm = 61) → (B[i] ← B[j] + K);
    "SBi Xj + K" (fm = 62) → (B[i] ← X[j]<17:0> + K);
    "SBi Xj + Bk" (fm = 63) → (B[i] ← X[j]<17:0> + B[k]);
    "SBi Aj + Bk" (fm = 64) → (B[i] ← A[j] + B[k]);
    "SBi Aj - Bk" (fm = 65) → (B[i] ← A[j] - B[k]);
    "SBi Bj + Bk" (fm = 66) → (B[i] ← B[j] + B[k]);
    "SBi Bj - Bk" (fm = 67) → (B[i] ← B[j] - B[k]);
  Set X[i]/SXi
    "SXi Aj + K" (fm = 70) → (X[i] ← sign_extend(A[j] + K));
    "SXi Bj + K" (fm = 71) → (X[i] ← sign_extend(B[j] + K));
    "SXi Xj + K" (fm = 72) → (X[i] ← sign_extend(X[j] + K));
    "SXi Xj + Bk" (fm = 73) → (X[i] ← sign_extend(X[j] + B[k]));
    "SXi Aj + Bk" (fm = 74) → (X[i] ← sign_extend(A[j] + B[k]));
    "SXi Aj - Bk" (fm = 75) → (X[i] ← sign_extend(A[j] - B[k]));
    "SXi Bj + Bk" (fm = 76) → (X[i] ← sign_extend(B[j] + B[k]));
    "SXi Bj - Bk" (fm = 77) → (X[i] ← sign_extend(B[j] - B[k]));
  Miscellaneous program control
    "PS" (:= fm = 0) → (Run ← 0);                          program stop
    "NO" (:= fm = 46) → ;                                  no operation; pass
  Jump unconditional
    "JP Bi + K" (:= fm = 02) → (P ← B[i] + K; p ← 3);      jump
  Jump on X[j] conditions
    "ZR Xj K" (:= fmi = 030) → ((X[j] = 0) → (P ← K; p ← 3));   zero
    "NZ Xj K" (:= fmi = 031) → ((X[j] ≠ 0) → (P ← K; p ← 3));   non zero
    "PL Xj K" (:= fmi = 032) → ((X[j] ≥ 0) → (P ← K; p ← 3));   plus or position
    "NG Xj K" (:= fmi = 033) → ((X[j] < 0) → (P ← K; p ← 3));   negative
    "IR Xj K" (:= fmi = 034) → (                           out of range constant tests
    ¬((X[j]<59:48>= 3777)∨ (X[j]<59:48> 4000)) →P ← K; p ← 3);
    "OR Xj K" (:= fmi = 035) → (
      (X[j]<59:48>=3777) ∨ (X[j]<59:48>=4000)→ (P ← K; p ← 3));
    "DF Xj K" (:= fmi = 036) → (                           indefinite form constant tests
      (X[j]<59:48>=1777) ∨ (X[j]<59:48>=6000)  → (P ← K; p ← 3));
    "ID Xj K" (:= fmi = 037) → (
      (X[j]<59:48>=1777) ∨ (X[j]<59:48>=6000) → (P ← K; p ← 3));
  Jump on B[i], B[j] comparison
    "EQ Bi Bj K" (:= fm = 04) → ((B[i] = B[j]) → (P ← K; p ← 3));   equal
    "NE Bi Bj K" (:= fm = 05) → ((B[i] ≠ B[j]) → (P ← K; p ← 3));   not equal
    "GE Bi Bj K" (:= fm = 06) → ((B[i] ≥ B[j]) → (P ← K; p ← 3));   greater than or equal
    "LT Bi Bj K" (:= fm = 07) → ((B[i] < B[j]) → (P ← K; p ← 3));   less than
  Subroutine call
    "RJ K" (:= fmi = 010) → (                              return jump
      M[K]<59:30> ← 04₈□00₈□(P + 1)□000000₈; next
      (P ← K + 1; p ← 3));
Reading (REC) and writing (WEC) Mp with Extended Core Storage, subjected to bounds checks, and Ms', Mp' mapping
  "REC Bj + K" (:= fmi = 011) → (                          read extended core
```

```
      Mp'[A[0]:A[0] + B[J] + K-1] ← Ms'[X[0]:X[0] + B[J] + K-1]);
   "WFC BJ + K" (:= fmi = 012) → (                                  writes extended core
      Ms'[X[0]:X[0] + B[J] + K-1] ← Mp'[A[0]:A[0] + B[J] + K-1]);
Fixed Point Arithmetic and Logical operations using X
   "IXI XJ + Xk" (:= fm = 36) → (X[i] ←X[J] + X[k]);                integer sum
   "IXI XJ - Xk" (:= fm = 37) → (X[i] ←X[J] - X[k]);                integer difference
   "CXI Xk" (:= fm = 47)→ (X[i] ←sum_modulo_2(X[k]);                count the number of bits in X[k]
   "BXI XJ" (:= fm = 10₈) → (X[i] ←X[J]);                           transmit
   "BXI XJ * Xk" (:= fm = 11₈) → (X[i] ←X[i] ←X[J] ∧ X[k]);         logical product
   "BXI XJ + Xk" (:= fm = 12) → (X[i] ←X[J] ∨ X[k]);                logical sum
   "BXI XJ - Xk" (:= fm = 13) → (X[i] ←X[J] ⊕ X[k]);                logical difference
   "BXI - Xk" (:= fm = 14) → (X[i] ←¬ X[k]);                        transmit complement
   "BXI - Xk * XJ" (:= fm = 15)→  (X[i] ←X[J] ∧ ¬ X[k]);            logical product and complement
   "BXI - Xk + XJ" (:= fm = 16)→  (X[i] ←X[J] ∨ ¬ X[k]);            logical sum and complement
   "BXI = Xk - XJ" (:= fm = 17)→  (X[i] ←X[J] ⊕ ¬ X[k]);            logical difference and complement
   "LXI Jk" (:= fm = 20) → (X[i] ←X[i] x 2^Jk {rotate});
   "AXI Jk" (:= fm = 21) → (X[i] ←X[i] / 2^Jk);                     arithmetic right shift
   "LXI BJ Xk" (:= fm = 22) → (                                     left shift nominally
      ¬B[J]<17> → X[i] ←X[k] x 2^B[J]<5:0> {rotate};
       B[J]<17> → X[i] ←X[k] / 2^¬ B[J]<10:0>);
   "AXI BJ Xk" (:= fm = 23) → (                                     arithmetic right shift nominally
      ¬B[J]<17> → X[i] ←X[k] / 2^B[J]<10:0>;
       B[J]<17> → X[i] ←X[k] x 2^¬ B[J]<5:0> {rotate});
   "MXI Jk" (:= fm = 43) → (                                        form mask
      X[i]<59:59-Jk+1> ← 2^Jk - 1;
      (Jk = 0) → X[i] ← 0);
Floating Point Arithmetic using X
 Only the least significant (lo) part of arithmetic is stored in Floating DP operations.
   "FXI XJ + Xk" (:= fm = 30) → (X[i] ← X[J] + X[k] {sf});          floating sum
   "FXI XJ - Xk" (:= fm = 31) → (X[i] ← X[J] - X[k] {sf});          floating difference
   "DXI XJ + Xk" (:= fm = 32) → (X[i] ← X[J] + X[k] {ls.df});       floating dp sum
   "DXI XJ - Xk" (:= fm = 33) → (X[i] ← X[J] - X[k] {ls.df});       floating dp difference
   "RXI XJ + Xk" (:= fm = 34) → (
      X[i] ← round(X[J]) + round(X[k]) {sf});
   "RXI XJ - Xk" (:= fm = 35) → (                                   round floating difference
      X[i] ← round(X[J]) - round(X[k]) {sf});
   "FXI XJ * Xk" (:= fm = 40) → (X[i] ← X[J] x X[k] {sf});          floating product
   "RXI XJ * Xk" (:= fm = 41) → (                                   round floating product
      X[i] ← X[J] x X[k] {sf}; next X[i] ← round(X[i]) {sf});
   "DXI XJ * Xk" (:= fm = 42) → (X[i] ← X[J] x X[k] {ls.df});       floating dp product
   "FXI XJ / Xk" (:= fm = 44) → (X[i] ← X[J] / X[k] {sf});          floating divide
   "RXI XJ / Xk" (:= fm = 45) → (X[i] ← round(X[J] / X[k]) {sf}); round floating divide
   "NXI BJ Xk" (:= fm = 24) → (                                     normalize
      X[i] ← normalize(X[k]) {sf};
      B[J] ← normalize_exponent(X[k]) {sf});
```

```
"ZXi Bj Xk" (:= fm = 25) → (                                    round and normalize
   X[i] ← round(X[k]) {sf}; next
   X[i] ← normalize(X[i]) {sf};
   B[j] ← normalize_exponent(X[i]) {sf});
"UXi Bj Xk' (:= fm = 26) → (B[j] ← X[k]<58:48> {si};            unpack
                            X[i] ← X[k]<59,47:0> {si});
"PXi Bj Xk" (:= fm = 27) → (X[k]<58:48> ← B[j] {si};            pack
                            X[k]<59,47:0> ← X[i] {si})
                                  )                             end Instruction_execution
```

## APPENDIX 2 CDC 6400, 6500, 6600, AND 6416 PERIPHERAL AND CONTROL PROCESSORS, PCP, ISP DESCRIPTION

```
                                    Appendix 2

                                CDC 6400, 6500, 6600, and 6416
                        Peripheral and Control Processors/PCP, ISP Description

Pc State
   A<17:0>                                          accumulator
   P<11:0>                                          Program Address Counter

Mp State
   M[0:4095]<11:0>                                  Mp
   M index[0:63]<11:0>:= M[0:63]<11:0>              special arrau in Mp reserved for index register

C('Central) State
   CP␣P<17:0>                                       the main Pc instruction address counter
   CPM[0:777777₈]<59:0>                             the Mp of main C

IO Registers for C('PCP)
   C␣DATA[0:63]<11:0>                               data buffers at peripheral K's
   C␣ACT[0:63]                                      a bit to denote if 1 of the 64 K's is active
   C␣FLG[0:63]                                      denotes a full (or empty) buffer at the K
   C␣FCN[0:63]<11:0>                                function or instruction register at a specific K

Instruction Format
   Ins[0:1]<11:0>                                   instruction
      long␣instruction                              2 w instruction: defined in terms of op codes, see Table, page 503
      short␣instruction := ¬ long␣instruction       1 w instruction
   F<5:0>    := Ins[0]<11:6>                        function or op code
   d<5:0>    := Ins[0]<5:0>
   m<11:0>   := Ins[1]                              address part
   dm<17:0>  := d□m
   i<11:0>   := Ins[1]<11:0>                        indirect bit
   d␣sign<11:0> := (
       ¬d<5> → 0□d;
        d<5> → ¬ d)
   md<11:0> := (
       (d = 0) → m;
       (d ≠ 0) → m + M[d])

Effective Address Calculation Process
   z := ((F<5:3> = 3) → d;
         (F<5:3> = 4) → i;
         (F<5:3> = 5) → md)

Instruction Interpretation Process
   Run → (Ins[0] ← M[P]; P ← P + 1; next                fetch
          long␣instruction → (Ins[1] ← M[P]; P ← P + 1): next
          Instruction␣execution)                        execute
```

*Implementation*
*The 10 x 52 bits in the barrel for the 10 PCP ISP include:*

A[0:9]<17:0> — *accumulators*

P[0:9]<11:0> — *instruction address counters*

*Temporary Hardware registers (not in the ISP)*

Q[0:9]<11:0> — *low order 6 bits of an instruction or address data*

K[0:9]<5:0> — *six bits hold the operation code. The 3 bits specify the trip count or state of an instruction's interpretation.*

T[0:9]<2:0>

F = $X_8Y_8$ Instruction execution := (

| $X_8$ \ $Y_8$ | 00 | 01 | 02 | 03 | 04 | 05 | 06 | 07 |
|---|---|---|---|---|---|---|---|---|
| 00 | PSN →; *null* | LJM → ( P← md); | RJM → ( M[md] ←P; P← md+1); | UJN → (( | ZJN → ( (A=0) → ( | NJN → ( (A≠0) → ( | PJN → ( ¬ A<17> → ( | MJN → ( A<17> → ( |
| | | | | * | | P← P+d␣sign)); | | |
| 10 | SHN → ( A←Ax2$^{d\text{␣}sign}$ ) * | LMN → ( A←A⊕d); * | LPN → ( A←A∧d); * | SCN → ( A←A∧¬d); * | LDN → ( A←d); * | LCN → ( A←-d); * | ADN→ ( A←A+d); * | SBN → ( A←A-d); * |
| 20 | LDC → ( A← dm): | ADC → ( A←A+dm); | LPC → ( A←A∧dm); | LMC → ( A←A⊕dm); | PSN →; *null* | PSN →; | EXN → ( CP␣P←A); | RPN → ( A←CP␣P); |
| 30 | LDD → ( | ADD → ( | SBD → ( | LMD → ( | STD → ( | RAD → ( | AOD → ( | SOD → ( |
| 40 | LDI → ( | ADI → ( | SBI → ( | LMI → ( | STI → ( | RAI → ( | AOI → ( | SOI → ( |
| 50 | LDM → ( A←M[z]): | ADM → ( A← A+M[z]); | SBM → ( A←A-M[z]); | LMM → ( A ←A⊕M[z]); | STM → ( M[z]← A); | RAM → ( A←A+M[z]; next M[z]←A); | AOM → ( A←M[z]+1; next M[z]←A); | SOM ( A←M[z]-1; next M[z]←A); |
| 60 | CRD → ( M[d:d+5]← CPM[A]); | CRM → ( M[m:m+ 5xM[d]-1]← CPM[A:A+ M[d]-1]); | CWD → ( CPM[A]← M[d:d+5]); | CWM → ( CPM[A:A+ M[d]-1]← M[m:m+ 5xM[d]-1]); | AJM → ( C␣ACT[d]→ ( | IJM → ( ¬ C␣ACT[d]→ ( P←m)); | FJM → ( C␣FLG[d]→ ( | EJM → ( ¬ C␣FLG[d]→ ( |
| 70 | IAN → ( A← C␣DATA[d]): | IAM → ( C␣FLG[d]→ ( M[m:m+A]← C␣DATA[d])); | OAN → ( C␣DATA[d] ←A); | OAM → ( (¬ C␣FLG[d]→ C␣DATA[d] ←M[m:m+A])); | ACN → ( C␣ACT[d] ←1); | DCN → ( C␣ACT[d] ←0); | FAN → ( C␣FCN[d] ←A); | FNC → ( C␣FCN[d] ←m); |

) *end Instruction␣execution*

* *1 word or short␣instruction*

# Chapter 43
# Parallel Operation in the Control Data 6600[1]

*James E. Thornton*

## History

In the summer of 1960, Control Data began a project which culminated October, 1964 in the delivery of the first 6600 Computer. In 1960 it was apparent that brute force circuit performance and parallel operation were the two main approaches to any advanced computer.

This paper presents some of the considerations having to do with the parallel operations in the 6600. A most important and fortunate event coincided with the beginning of the 6600 project. This was the appearance of the high-speed silicon transistor, which survived early difficulties to become the basis for a nice jump in circuit performance.

## System Organization

The computing system envisioned in that project, and now called the 6600, paid special attention to two kinds of use, the very large scientific problem and the time sharing of smaller problems. For the large problem, a high-speed floating point central processor with access to a large central memory was obvious. Not so obvious, but important to the 6600 system idea, was the isolation of this central arithmetic from any peripheral activity.

It was from this general line of reasoning that the idea of a multiplicity of peripheral processors was formed (Fig. 1). Ten such peripheral processors have access to the central memory on one side and the peripheral channels on the other. The executive control of the system is always in one of these peripheral processors, with the others operating on assigned peripheral or control tasks. All ten processors have access to twelve input-output channels and may "change hands," monitor channel activity, and perform other related jobs. These processors have access to central memory, and may pursue independent transfers to and from this memory.

Each of the ten peripheral processors contains its own memory for program and buffer areas, thereby isolating and protecting the more critical system control operations in the separate processors.

[1]*AFIPS Proc. FJCC*, pt. 2, vol. 26, 1964, pp. 33–40.

The central processor operates from the central memory with relocating register and file protection for each program in central memory.

## Peripheral and Control Processors

The peripheral and control processors are housed in one chassis of the main frame. Each processor contains 4096 memory words of 12 bits length. There are 12- and 24-bit instruction formats to provide for direct, indirect, and relative addressing. Instructions provide logical, addition, subtraction, and conditional branching. Instructions also provide single word or block transfers to and from any of twelve peripheral channels, and single word or block transfers to and from central memory. Central memory words of 60 bits length are assembled from five consecutive peripheral words. Each processor has instructions to interrupt the central processor and to monitor the central program address.

To get this much processing power with reasonable economy and space, a time-sharing design was adopted (Fig. 2). This design contains a register "barrel" around which is moving the dynamic information for all ten processors. Such things as program address, accumulator contents, and other pieces of information totalling 52 bits are shifted around the barrel. Each complete trip around requires one major cycle or one thousand nanoseconds. A "slot" in the barrel contains adders, assembly networks, distribution network, and interconnections to perform one step of any peripheral instruction. The time to perform this step or, in other words, the time through the slot, is one minor cycle or one hundred nanoseconds. Each of the ten processors, therefore, is allowed one minor cycle of every ten to perform one of its steps. A peripheral instruction may require one or more of these steps, depending on the kind of instruction.

In effect, the single arithmetic and the single distribution and assembly network are made to appear as ten. Only the memories are kept truly independent. Incidentally, the memory read-write cycle time is equal to one complete trip around the barrel, or one thousand nanoseconds.

Input-output channels are bi-directional, 12-bit paths. One 12-bit word may move in one direction every major cycle, or 1000 nanoseconds, on each channel. Therefore, a maximum burst rate of 120 million bits per second is possible using all ten peripheral processors. A sustained rate of about 50 million bits per second can be maintained in a practical operating system. Each channel may service several peripheral devices and may interface to other systems, such as satellite computers.

Peripheral and control processors access central memory through an assembly network and a dis-assembly network. Since

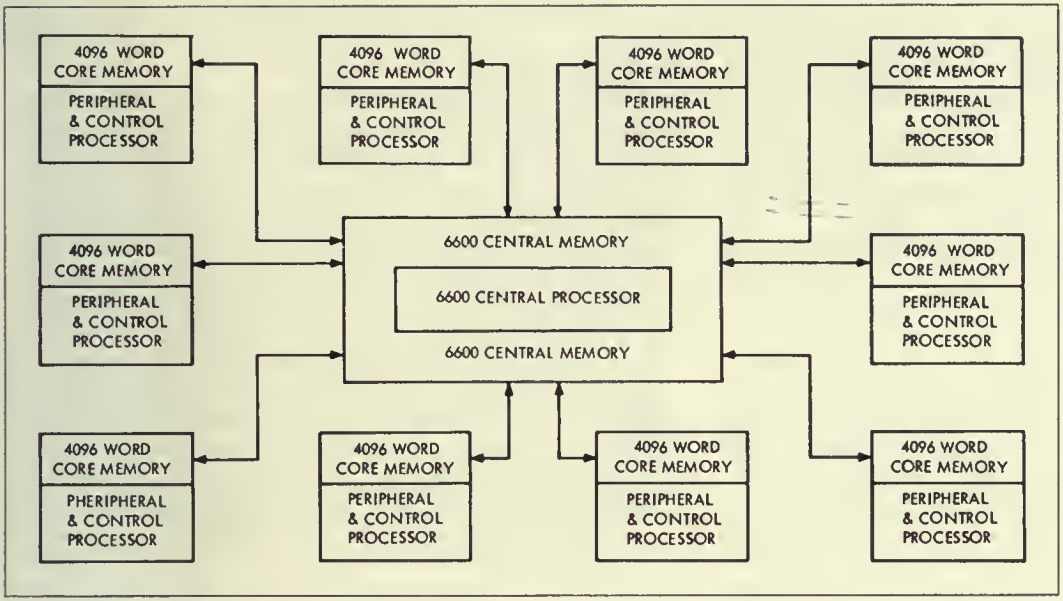


Fig. 1. Control Data 6600.

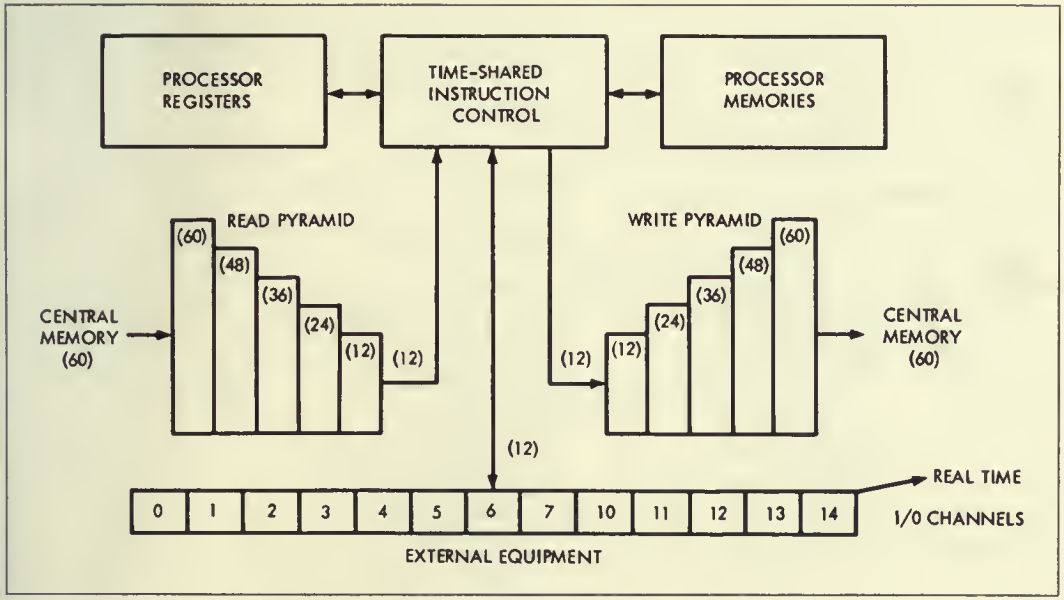


Fig. 2. 6600 peripheral and control processors.

five peripheral memory references are required to make up one central memory word, a natural assembly network of five levels is used. This allows five references to be "nested" in each network during any major cycle. The central memory is organized in independent banks with the ability to transfer central words every minor cycle. The peripheral processors, therefore, introduce at most about 2% interference at the central memory address control.

A single real time clock, continuously running is available to all peripheral processors.

## Central Processor

The 6600 central processor may be considered the high-speed arithmetic unit of the system (Fig. 3). Its program, operands, and results are held in the central memory. It has no connection to the peripheral processors except through memory and except for two single controls. These are the exchange jump, which starts or interrupts the central processor from a peripheral processor, and the central program address which can be monitored by a peripheral processor.

A key description of the 6600 central processor, as you will see in later discussion, is "parallel by function." This means that a number of arithmetic functions may be performed concurrently. To this end, there are ten functional units within the central processor. These are the two increment units, floating add unit, fixed add unit, shift unit, two multiply units, divide unit, boolean unit, and branch unit. In a general way, each of these units is a three address unit. As an example, the floating add unit obtains two 60-bit operands from the central registers and produces a 60 bit result which is returned to a register. Information to and from these units is held in the central registers, of which there are twenty-four. Eight of these are considered index registers, are of 18 bits length, and one of which always contains zero. Eight are considered address registers, are of 18 bits length, and serve to address the five read central memory trunks and the two store central memory trunks. Eight are considered floating point

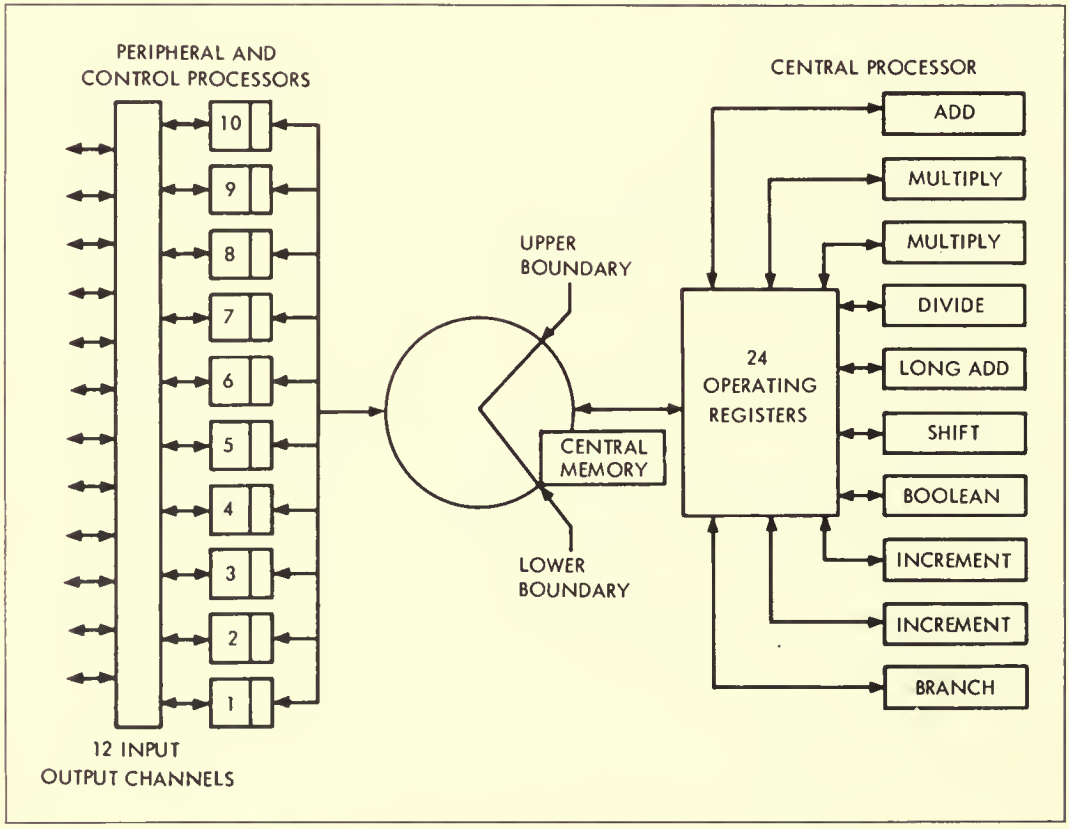


**Fig. 3. Block diagram of 6600.**

registers, are of 60 bits length, and are the only central registers to access central memory during a central program.

In a sense, just as the whole central processor is hidden behind central memory from the peripheral processors, so, too, the ten functional units are hidden behind the central registers from central memory. As a consequence, a considerable instruction efficiency is obtained and an interesting form of concurrency is feasible and practical. The fact that a small number of bits can give meaningful definition to any function makes it possible to develop forms of operand and unit reservations needed for a general scheme of concurrent arithmetic.

Instructions are organized in two formats, a 15-bit format and a 30-bit format, and may be mixed in an instruction word (Fig. 4). As an example, a 15-bit instruction may call for an ADD, designated by the *f* and *m* octal digits, from registers designated by the *j* and *k* octal digits, the result going to the register designated by the *i* octal digit. In this example, the addresses of the three-address, floating add unit are only three bits in length, each address referring to one of the eight floating point registers. The 30-bit format follows this same form but substitutes for the *k* octal digit an 18-bit constant K which serves as one of the input operands. These two formats provide a highly efficient control of concurrent operations.

As a background, consider the essential difference between a general purpose device and a special device in which high speeds are required. The designer of the special device can generally improve on the traditional general purpose device by introducing some form of concurrency. For example, some activities of a housekeeping nature may be performed separate from the main sequence of operations in separate hardware. The total time to complete a job is then optimized to the main sequence and excludes the housekeeping. The two categories operate concurrently.

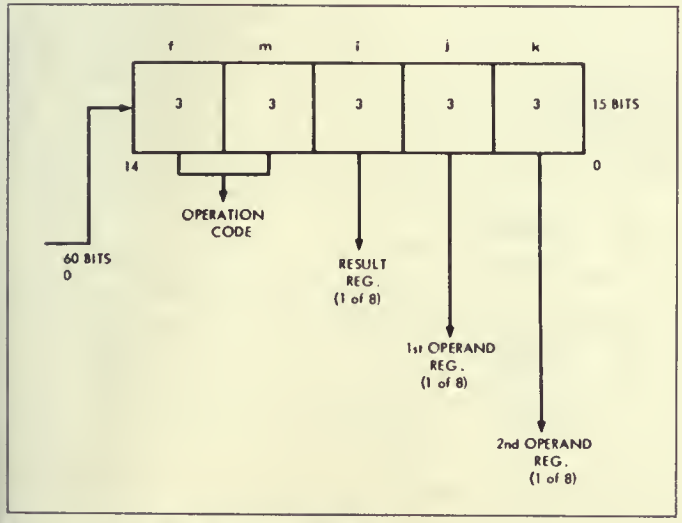


**Fig. 4. Fifteen-bit instruction format.**

It would be, of course, most attractive to provide in a general purpose device some generalized scheme to do the same kind of thing. The organization of the 6600 central processor provides just this kind of scheme. With a multiplicity of functional units, and of operand registers and with a simple and highly efficient addressing system, a generalized queue and reservation scheme is practical. This is called the *scoreboard.*

The scoreboard maintains a running file of each central register, of each functional unit, and of each of the three operand trunks to and from each unit. Typically, the scoreboard file is made up of two-, three-, and four-bit quantities identifying the nature of register and unit usage. As each new instruction is brought up, the conditions at the instant of issuance are set into the scoreboard. A snapshot is taken, so to speak, of the pertinent conditions. If no waiting is required, the execution of the instruction is begun immediately under control of the unit itself. If waiting is required (for example, an input operand may not yet be available in the central registers), the scoreboard controls the delay, and when released, allows the unit to begin its execution. Most important, this activity is accomplished in the scoreboard and the functional unit, and does not necessarily limit later instructions from being brought up and issued.

In this manner, it is possible to issue a series of instructions, some related, some not, until no functional units are left free or until a specific register is to be assigned more than one result. With just those two restrictions on issuing (unit free and no double result), several independent chains of instructions may proceed concurrently. Instructions may issue every minor cycle in the absence of the two restraints. The instruction executions, in comparison, range from three minor cycles for fixed add, 10 minor cycles for floating multiply, to 29 minor cycles for floating divide.

To provide a relatively continuous source of instructions, one buffer register of 60 bits is located at the bottom of an instruction stack capable of holding 32 instructions (Fig. 5). Instruction words from memory enter the bottom register of the stack pushing up the old instruction words. In straight line programs, only the bottom two registers are in use, the bottom being refilled as quickly as memory conflicts allow. In programs which branch back to an instruction in the upper stack registers, no refills are allowed after the branch, thereby holding the program loop completely in the stack. As a result, memory access or memory conflicts are no longer involved, and a considerable speed increase can be had.

Five memory trunks are provided from memory into the central processor to five of the floating point registers (Fig. 6). One address register is assigned to each trunk (and therefore to the

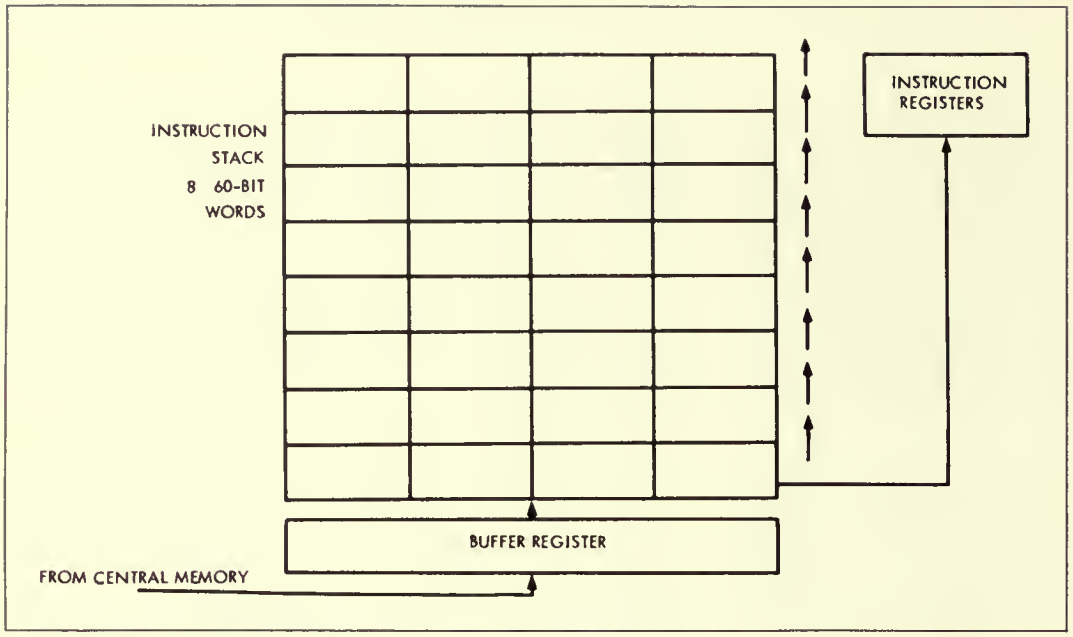


**Fig. 5. 6600 instruction stack operation.**

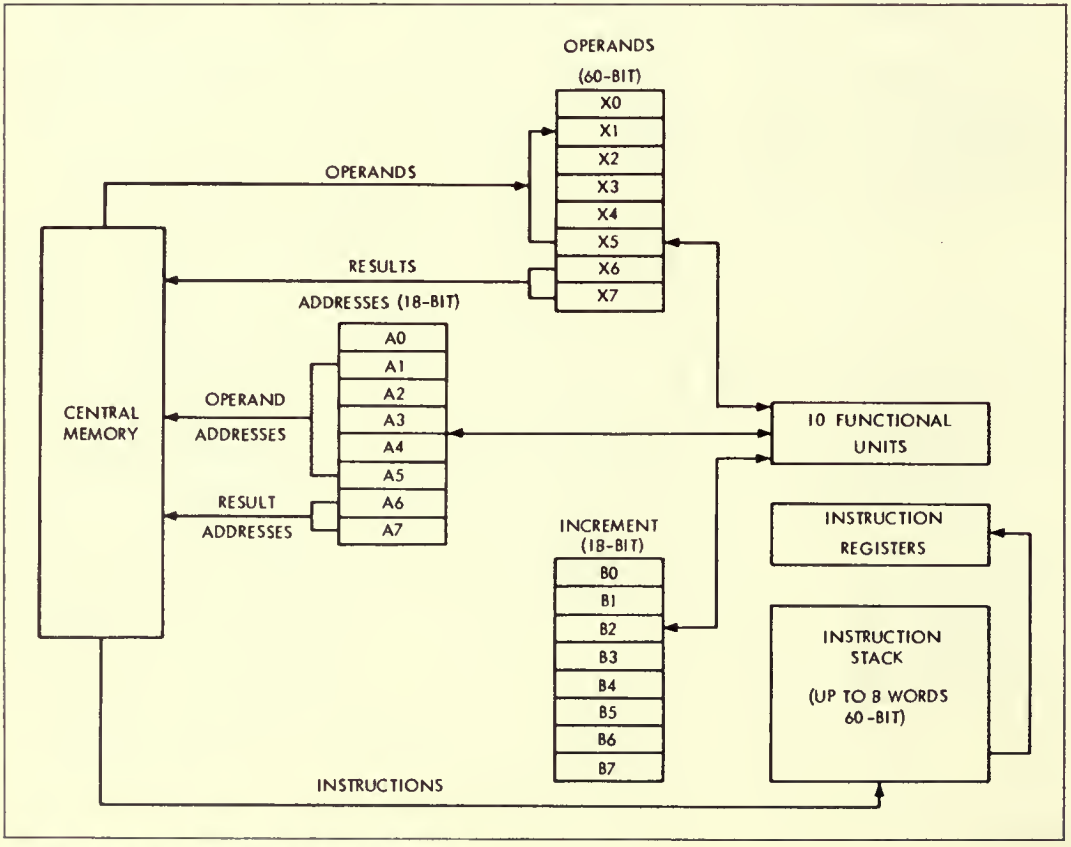


**Fig. 6. Central processor operating registers.**

floating point register). Any instruction calling for address register result implicitly initiates a memory reference on that trunk. These instructions are handled through the scoreboard and therefore tend to overlap memory access with arithmetic. For example, a new memory word to be loaded in a floating point register can be brought in from memory but may not enter the register until all previous uses of that register are completed. The central registers, therefore, provide all of the data to the ten functional units, and receive all of the unit results. No storage is maintained in any unit.

Central memory is organized in 32 banks of 4096 words. Consecutive addresses call for a different bank; therfore, adjacent addresses in one bank are in reality separated by 32. Addresses may be issued every 100 nanoseconds. A typical central memory information transfer rate is about 250 million bits per second.

As mentioned before, the functional units are hidden behind the registers. Although the units might appear to increase hardware duplication, a pleasant fact emerges from this design. Each unit may be trimmed to perform its function without regard to others. Speed increases are had from this simplified design.

As an example of special functional unit design, the floating multiply accomplishes the coefficient multiplication in nine minor cycles plus one minor cycle to put away the result for a total of 10 minor cycles, or 1000 nanoseconds. The multiply uses layers of carry save adders grouped in two halves. Each half concurrently forms a partial product, and the two partial products finally merge while the long carries propagate. Although this is a fairly large complex of circuits, the resulting device was sufficiently smaller than originally planned to allow two multiply units to be included in the final design.

To sum up the characteristics of the central processor, remember that the broadbrush description is "concurrent operation." In other words, any program operating within the central processor utilizes some of the available concurrency. The program need not be written in a particular way, although certainly some optimization can be done. The specific method of accomplishing this concurrency involves *issuing* as many instructions as possible while handling most of the conflicts during *execution.* Some of the essential requirements for such a scheme include:

1 Many functional units

2 Units with three address properties

3 Many transient registers with many trunks to and from the units

4 A simple and efficient instruction set

## Construction

Circuits in the 6600 computing system use all-transistor logic (Fig. 7). The silicon transistor operates in saturation when switched

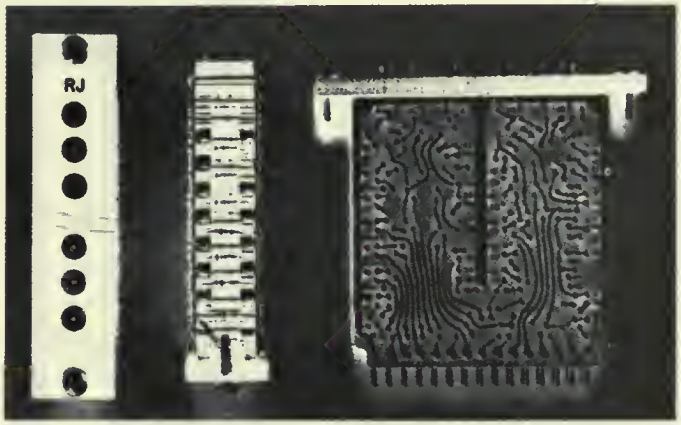


**Fig. 7. 6600 printed circuit module.**

"on" and averages about five nanoseconds of stage delay. Logic circuits are constructed in a cordwood plug-in module of about 2½ inches by 2½ inches by 0.8 inch. An average of about 50 transistors are contained in these modules.

Memory circuits are constructed in a plug-in module of about six inches by six inches by 2½ inches (Fig. 8). Each memory module contains a coincident current memory of 4096 12-bit

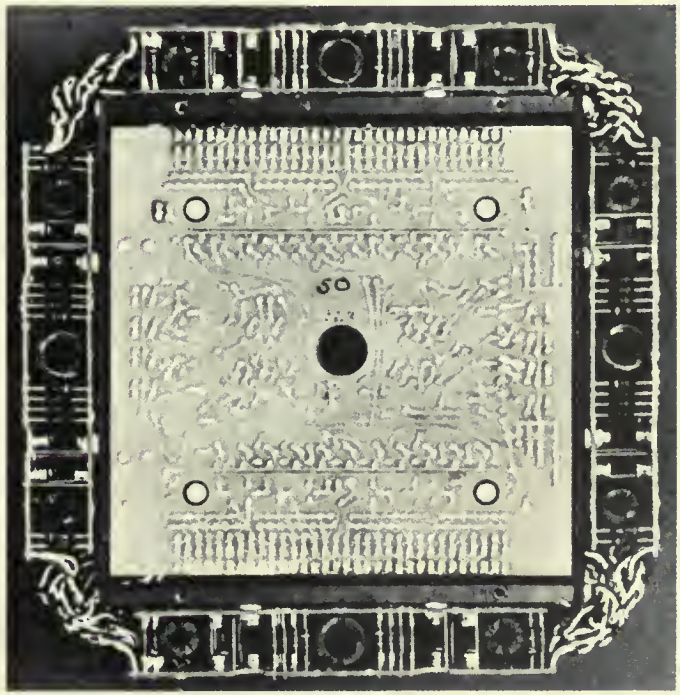

**Fig. 8. 6600 memory module.**


Fig. 9. 6600 main frame section.

words. All read-write drive circuits and bit drive circuits plus address translation are contained in the module. One such module is used for each peripheral processor, and five modules make up one bank of central memory.

Logic modules and memory modules are held in upright hinged chassis in an X shaped cabinet (Fig. 9). Interconnections between modules on the chassis are made with twisted pair transmission lines. Interconnections between chassis are made with coaxial cables.

Both maintenance and operation are accomplished at a programmed display console (Fig. 10). More than one of these consoles may be included in a system if desired. Dead start facilities bring the ten peripheral processors to a condition which allows information to enter from any chosen peripheral device. Such loads normally bring in an operating system which provides a highly sophisticated capability for multiple users, maintenance, and so on.

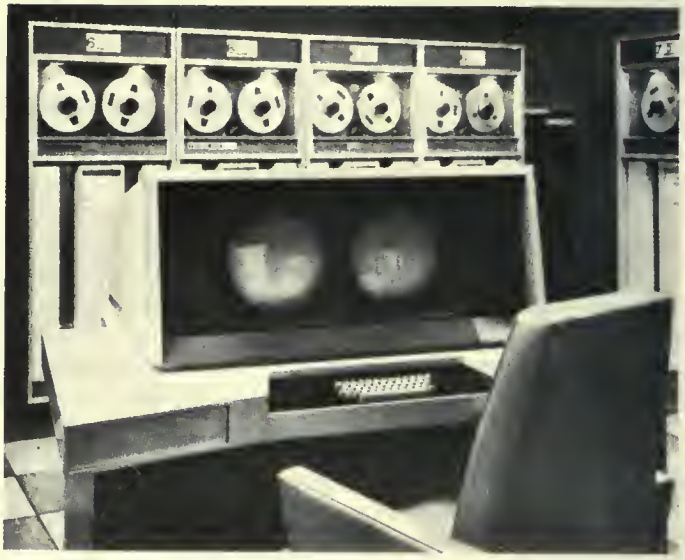

Fig. 10. 6600 display console.

The 6600 Computer has taken advantage of certain technology advances, but more particularly, logic organization advances which now appear to be quite successful. Control Data is exploring advances in technology upward within the same compatible structure, and identical technology downward, also within the same compatible structure.

## References

Allard, Wolf, and Zemlin [1964]; Clayton, Dorff, and Fagen [1964].

## APPENDIX 1 ISP OF CDC 6600 PERIPHERAL AND CONTROL PROCESSOR

```
PC6600(process) :=
    begin

! ISP of the CDC 6600 Peripheral and Control Processor, Barrel distributor,
! and I/O channels.

! Although the 6600 has 10 identical Peripheral and Control processors,
! the ISP for a single processor is shown.  An identifying parameter
! is utilized to specify which of the ten processors is active during
! simulation.

! The CDC 6600 Peripheral and Control processors each possess a
! 4096 word 12 bit local memory.  The ISP shows only one 4096 word
! memory which is used by all the "processors".

**Channel.State**

    CHAN[0:11]<11:0>,                   ! I/O channels

    cact[0:11]<>,                       ! Channel active indicator
    cful[0:11]<>,                       ! Channel full indicator

**Barrel.State**

    A[0:9]<17:0>,                       ! Barrel A registers
    P[0:9]<11:0>,                       ! Barrel P registers
    Q[0:9]<11:0>,                       ! Barrel Q registers
    K[0:9]<8:0>,                        ! Barrel K registers

**PCP.Memory.State**

    M.PCP[0:4095]<11:0>,                ! Only one PCP memory is shown

    read[0:4]<11:0>,                    ! Read pyramid
        c.read<59:0> := read[0:4]<11:0>,

    write[0:4]<11:0>,                   ! Write pyramid
        c.write<59:0> := write[0:4]<11:0>,

**PCP.Instruction.Format**

    pir<23:0>,                          ! PCP Instruction register
        f <  5:0> := pir<23:18>,
        d <  5:0> := pir<17:12>,
        m <11:0> := pir<11:0>,
        dm<17:0> := pir<17:0>,

**Addressing.Calculation**{us}

    index(id<3:0>)<11:0> :=                     ! Indexed addressing
        begin
        DECODE d eql 0 =>
            begin
            0 := (index = m + M.PCP[d];
                  P[id] = P[id] + 1),
            1 :=  index = m
            end
        end,

**Barrel.Execution**

    barrel(main) :=
        begin
        pcp(0) next                             ! Activate processor 0
        pcp(1) next                             ! Activate processor 1
        pcp(2) next                             ! Activate processor 2
        pcp(3) next                             ! Activate processor 3
        pcp(4) next                             ! Activate processor 4
        pcp(5) next                             ! Activate processor 5
        pcp(6) next                             ! Activate processor 6
        pcp(7) next                             ! Activate processor 7
        pcp(8) next                             ! Activate processor 8
        pcp(9) next                             ! Activate processor 9
        RESTART barrel                          ! Do it all again
        end,

**PCP.Execution**{oc}

    pcp(id<3:0>) :=
        begin
        pir<23:12> = M.PCP[P[id]] next
        P[id] = P[id] + 1 next
        m = M.PCP[P[id]];
        K[id]<5:0> = f;
        Q[id] = d next
        DECODE K[id] =>
            begin
    [#00,#24,#25]:=  no.op(),                   ! PSN - Pass
            #14 :=  A[id]  = d,                 ! LDN - Load d into a
            #15 :=  A[id] = #77778(not d),      ! LCN - Load compliment d
            #30 :=  A[id]  = M.PCP[d],          ! LOD - Load (d)
            #34 :=  M.PCP[d] = A[id],           ! STD - Store (d)
            #40 :=  A[id]  = M.PCP[M.PCP[d]],   ! LDI - Load ((d))
            #44 :=  M.PCP[M.PCP[d]] = A[id],    ! STI - Store ((d))
            #20 := (A[id] = dm; P[id]=P[id]+1), ! LDC - Load dm
            #50 :=  A[id] = M.PCP[index(id)],   ! LDM - Load (m + (d))
            #54 :=  M.PCP[index(id)] = A[id],   ! STM - Store (m + (d))
            #16 :=  A[id] = A[id] +{us} d,      ! ADN - Add d
            #17 :=  A[id] = A[id] -{us} d,      ! SBN - Subtract d
            #31 :=  A[id] = A[id]+(us) M.PCP[d],! ADD - Add (d)
            #32 :=  A[id] = A[id]-(us) M.PCP[d],! SBD - Subtract (d)
            #41 :=  A[id] = A[id] +{us}         ! AD1 - Add ((d))
                        M.PCP[M.PCP[d]],
            #42 :=  A[id] = A[id] -{us}         ! SBI - Subtract ((d))
                        M.PCP[M.PCP[d]],
            #21 :=  A[id] = A[id] + dm,         ! ADC - Add dm
            #51 :=  A[id] = A[id] +{us}         ! ADM - Add (m + (d))
                        M.PCP[index(id)],
            #52 :=  A[id] = A[id] -{us}         ! SBM - Subtract (m + (d))
                        M.PCP[index(id)],
            #10 := (DECODE d<5> =>              ! SHN - Shift d
                    begin
                    0 := A[id] slr d,
                    1 := A[id] srO (not d)
                    end),
            #11 :=  A[id]<5:0> = A[id]<5:0>     ! LMN - Logicel difference d
                        xor d,
            #12 :=  A[id] = A[id] and d,        ! LPN - Logical procuct d
            #13 :=  A[id]<5:0> = A[id]<5:0>     ! SCN - Selective clear d
                        and (not d),
            #33 :=  A[id]<11:0>= A[id]<11:0>    ! LMD - Logical difference (d)
                        xor M.PCP[d],
            #43 :=  A[id]<11:0>= A[id]<11:0>    ! LM1 - Logical difference ((d))
                        xor M.PCP[M.PCP[d]],
            #22 :=  A[id] = A[id] and dm,       ! LPC - Logical product dm
            #23 :=  A[id] = A[id] xor dm,       ! LMC - Logical difference dm
            #53 :=  A[id]<11:0> =               ! LMM - Logicel difference
                                                ! (m +(d))
                    A[id]<11:0> xor M.PCP[index(id)],
            #35 :=  M.PCP[d] = A[id] = A[id]    ! RAD - Replace add (d)
                                + M.PCP[d],
            #36 :=  M.PCP[d] = A[id]            ! AOD - Replace add one (d)
                        = M.PCP[d] + 1,
            #37 :=  M.PCP[d] = A[id]            ! SOD - Replace subtract one (d)
                        = M.PCP[d] - 1,
            #45 :=  M.PCP[M.PCP[d]] = A[id]     ! RAI - Replace add ((d))
                        = A[id] + M.PCP[M.PCP[d]],
            #48 :=  M.PCP[M.PCP[d]] = A[id]     ! AD1 - Replace add one ((d))
                        = M.PCP[M.PCP[d]] + 1,
            #47 :=  M.PCP[M.PCP[d]] = A[id]     ! SDI - Replace subtrace one
                        = M.PCP[M.PCP[d]] - 1,  ! ((d))
            #55 :=  M.PCP[index] = A[id]        ! RAM - Replace add (m + (d))
                        = A[id] + M.PCP[index(id)],
            #58 :=  M.PCP[index] = A[id]        ! ADM - Replace add one
                        = M.PCP[index(id)] + 1, ! (m + (d))
            #57 :=  M.PCP[index] = A[id]        ! SDM - Replace subtract one
                        = M.PCP[index(id)] - 1, ! (m + (d))
            #03 := P[id] = (P[id] - 1) + d,     ! UJN - Unconditional jump d
            #04 := IF A[id] eql(us) 0 =>        ! ZJN - Zero Jump d
                        P[id] = (P[id] - 1) + d,
            #05 := IF A[id] neq(us) 0 =>        ! NJN - Nonzero jump d
                        P[id] = (P[id] - 1) + d,
            #06 := IF A[id] geq(us) 0 =>        ! PJN - Plus jump d
                        P[id] = (P[id] - 1) + d,
            #07 := IF A[id] lss(us) 0 =>        ! MJN - Minus jump d
                        P[id] = (P[id] - 1) + d,
            #01 :=  P[id] = index(id),          ! LJM - Long jump to m + (d)
            #02 := (M.PCP[index(id)]            ! RJM - Return jump to m + (d)
                        = P[id] + 2 next
                    P[id] = index + 1),
            #26 := (WAIT (xjf eqv '0) next      ! EXN - Exchange Jump
                    xja = A[id]; xjf = 1),
            #27 :=  A[id] = pc,                 ! RPN - Read program address
            #60 := (c.read = MP[A[id]] next     ! CRD - Central read d = (A)
                    M.PCP[d+0] = read[0] next
                    M.PCP[d+1] = read[1] next
                    M.PCP[d+2] = read[2] next
                    M.PCP[d+3] = read[3] next
                    M.PCP[d+4] = read[4]),
            #61 := (M.PCP[0] = P[id] + 1 next    ! CRM - Central read (d)
                    P[id] = m; Q[id] = d next   !       words from (A) to m
                    CRM0 :=
                        begin
                        c.read = MP[A[id]] next
                        M.PCP[P[id]+0] = read[0] next
                        M.PCP[P[id]+1] = read[1] next
                        M.PCP[P[id]+2] = read[2] next
                        M.PCP[P[id]+3] = read[3] next
                        M.PCP[P[id]+4] = read[4] next
                        P[id] = P[id] + 5;
                        A[id] = A[id] + 1;
                        Q[id] = Q[id] - 1 next
                        IF Q[id] neq 0 => RESTART CRM0
                        end next
                    P[id] = M.PCP[0]),
            #62 := (write[0] = M.PCP[d+0] next   ! CWD - Central write (A) = d
                    write[1] = M.PCP[d+1] next
                    write[2] = M.PCP[d+2] next
                    write[3] = M.PCP[d+3] next
                    write[4] = M.PCP[d+4] next
                    MP[A[id]] = c.write),
            #63 := (M.PCP[0] = P[id] + 1 next    ! CWM - Central write (d)
                    P[id] = m;                  !       to (A) from m
                    Q[id] = d next
                    CWM0 :=
                        begin
                        write[0] = M.PCP[P[id]+0] next
                        write[1] = M.PCP[P[id]+1] next
                        write[2] = M.PCP[P[id]+1] next
                        write[3] = M.PCP[P[id]+1] next
                        write[4] = M.PCP[P[id]+4] next
```

## APPENDIX 1 (cont'd.)

```
                    P[id] = P[id] + 5;
                    A[id] = A[id] + 1;
                    Q[id] = Q[id] - 1 next
                    If Q[id] neq 0 => RESTART CWM0
                    end next
               P[id] = M.PCP[0]),
          #64 := (DECODE cact[d] =>              ! AJM - Jump to m if channel
                    begin                        !       d is active
                    0 := P[id] = P[id] + 1,
                    1 := P[id] = m
                    end),
          #65 := (DECODE cact[d] =>              ! IJM - Jump to m if channel
                    begin                        !       d is inactive
                    0 := P[id] = m,
                    1 := P[id] = P[id] + 1
                    end),
          #66 := (DECODE cful[d] =>              ! FJM - Jump to m if channel
                    begin                        !       d is full
                    0 := P[id] = P[id] + 1,
                    1 := P[id] = m
                    end),
          #67 := (DECODE cful[d] =>              ! EJM - Jump to m if channel
                    begin                        !       d is empty
                    0 := P[id] = m,
                    1 := P[id] = P[id] + 1
                    end),
          #70 :=  A[id] = CHAN[d],               ! IAN - Input A from CHAN d
          #71 := (M.PCP[0] = P[id] next          ! IAM - Input (A) words to m
                  P[id] = m next                 !       from channel d
                  IAM0 :=
                    begin
                    IF A[id] neq 0 => M.PCP[P[id]] = 0 next
                    IF cact[d] =>
                        begin
                        M.PCP[P[id]] = CHAN[d] next
                        P[id] = P[id] + 1; A[id] = A[id] - 1 next
                        IF A[id] neq 0 => RESTART IAM0
                        end
                    end next
                  P[id] = M.PCP[0]),
          #72 :=  CHAN[d] = A[id],               ! OAN - Output from A
                                                 !       on channel d
          #73 := (M.PCP[0] = P[id] + 1 next      ! OAM - Output (A) words
                  P[id] = m next                 !       from m on channel d
                  OAM0 :=
                    begin
                    IF cact[d] and (A[id] neq 0) =>
                        begin
                        CHAN[d] = M.PCP[A[id]] next
                        A[id] = A[id] - 1 next
                        RESTART OAM0
                        end
                    end next
                  P[id] = M.PCP[0]),
          #74 :=  cact[d] = 1,                   ! ACN - Activate channel d
          #75 :=  cact[d] = 0,                   ! DCN - Disconnect channel d
          #76 :=  CHAN[d] = A[id],               ! FAN - Function (A) on CHAN d
          #77 := (CHAN[d] = m;                   ! FNC - Function m on CHAN d
                  P[id] = P[id] + 1),
          end
     end

end          ! End CDC 6600 Peripheral and Control processor
```

## APPENDIX 2 ISP OF THE CDC 6600

```
COC6600(process) :=
    begin

! ISP of the CDC 6600

! Floating point instructions are not described.

! The central processor and central memory are described in this
! ISP.  An auxiliary ISP (PC6800.ISP) describes the peripheral
! processors and control barrel execution.

! The ten functional units are described and allow parallel
! simulation.

! Instructions are processed from an instruction stack.  Instruction
! conflicts are resolved by keeping a "scorecard" containing utilization
! information on all registers and all functional units.
! Reservation control decodes an instruction to determine register
! utilization.  Source and destination registers are allocated
! if they are not being used as destinations of another functional
! unit.  If the required functional unit is free and if both the
! source and destination registers are available, the instruction
! is released to the unit for execution.  If the resources are
! not available, reservation control holds the instruction until
! the resources become available.
! At the completion of execution by a functional unit, tha resourcas
! are released by marking the scorecard.

! Tha following page by page index of tha ISP is provided to aid
! in locating CDC 6600 architectural features.

!         **Central.Memory.State** defines the Central Memory.
!         **Processor.State** defines central processor carriers.
!         **Instruction.Format** defines instruction fields.
!         **Implementation.Declarations** defines ISP related variablas.
!         **Reservation.Control.State** defines variables used by
!         reservation control.  These declarations constitute the
!         resourca allocation "scorecard".
!             Describe the reservation control execution.
!         **Instruction.Fetch** describes tha instruction stack
!         control and instruction fetch processas.
!         **Central.Memory.Access** describes the instruction read
!         and tha register associatad memory access processes.
!         **Exchange.Jump** is the processor interrupt facility.
!         **Instruction.Cycle** is the main instruction processing
!         cycla.  Instruction execution is initiated by issuing
!         the instructions to tha appropriate functional unit.

! Tha functional units are:

!         Branch Unit.
!         Boolaan Unit.
!         Shift Unit.
!         Add Unit.
!         Long Add Unit.
!         Multiply Unit 0.
!         Multiply Unit 1.
!         Divide Unit.
!         Increment Unit 0.
!         Increment Unit 1.

**Central.Memory.State**

    MP[0:4095]<59:0>,                   ! Use only 4k of 60 bit memory

**Processor.State**

    xjp[0:15]<59:0>,                    ! Exchange Jump Packaga
    xja<18:0>,                          ! Exchange Jump Address
    xjf<>,                              ! Exchange Jump Flag

    px<19:0>,                           ! Psaudo program countar
        PC<17:0> := px<19:2>,           ! Program countar
        ilc<1:0> := px<1:0>,            ! Instruction length coun1
        isc<4:0> := px<4:0>,            ! Instruction stack counter
    AREG[0:7]<17:0>,                    ! A ragistars
    BREG[0:7]<17:0>,                    ! B registers
    XREG[0:7]<59:0>,                    ! X registers
    RACM<17:0>,                         ! Raf Address (cantral memory)
    FLCM<17:0>,                         ! Fiald length of program
    RAECS<23:0>,                        ! Reference Address for ECS
    FLECS<23:0>,                        ! Fiald length for ECS
    EM<17:0>,                           ! Program exit mode
    MA<17:0>,                           ! Monitor exchange

**Instruction.Format**

    I<29:0>,                            ! Instruction ragistar
        i0<14:0> := I<29:15>,           ! Short instruction (15 bit)
        i1<14:0> := I<14:0>,            ! Long instruction extansion

        f. <2:0> := I<29:27>,
        m. <2:0> := I<28:24>,
        fm <5:0> := I<29:24>,
        I. <2:0> := i<23:21>,
        j. <2:0> := I<20:18>,
        k. <2:0> := I<17:15>,
        ki<17:0> := I<17:0>,

    is[0:7]<59:0>,                      ! Instruction stack
        ism[0:31]<14:0> := is[0:7]<59:0>,
    ishi<17:0>,                         ! High address limit in stack
    islo<17:0>,                         ! Low  address limit in stack
    isa<2:0>,                           ! Stack insert counter

**Implementation.Declarations**

    stop.bit<>,                         ! Stop flag

    macro not.described := [no.op()],

**Reservation.Control.State**

    abusy[0:7]<>,                       ! A registers busy bits
    arw  [0:7]<>,                       ! A registers read(0)/write(1)
    bbusy[0:7]<>,                       ! B registers busy bits
    brw  [0:7]<>,                       ! B registers read(0)/write(1)
    xbusy[0:7]<>,                       ! X registers busy bits
    xrw  [0:7]<>,                       ! X registers read(0)/write(1)

    fbusy[0:9]<>,                       ! Functional Unit busy bits

                                        ! The following tables are
                                        ! used to deallocata the
                                        ! resource assignments either
                                        ! in the avent of conflict during
                                        ! allocation, or during deallocation
                                        ! at instruction completion.
                                        ! F??U<> indicates usage of tha
                                        ! registers by a unit.
                                        !  1 = used, 0 = not used
    fa [0:9]<2:0>,                      ! Functional Unit A register
    fau[0:9]<>,                         ! A register usage
    fb [0:9]<2:0>,                      ! Functional Unit B register
    fbu[0:9]<>,                         ! B register usage
    fx [0:9]<2:0>,                      ! Functional Unit X register
    fxu[0:9]<>,                         ! X register usage

    unit<3:0>,                          ! Temporary for arith unit numbar

**Reservation.Control**(us)

    source()<> :=                       ! Source register allocation
        begin
        source = 0 next
        DECODE fm =>
            begin
             #01 := IF (i. eql #1) or (i. eql #2) =>
                       (If fbu[unit] and (fb[unit] eql j.) => source = 1 naxt
                        fb[unit] = j.; fbu[unit] = 1;
                        IF (not bbusy[j.]) and (not brw[j.]) =>
                                        source = bbusy[j.] = 1),
[#02,#04:#07,#22]:= (IF fbu[unit] and (fb[unit] eql i.) => source = 1 next
                        fb[unit] = i.; fbu[unit] = 1;
                        IF (not bbusy[i.]) and (not brw[i.]) =>
                           source = bbusy[i.] = 1),
[#04:#07,#23:#27,
 #51,#56:#67,#61,
     #66:#07,#71,
         #70:#77]:= (IF fbu[unit] and (fb[unit] eql j.) => source = 1 next
                        fb[unit] = j.; fbu[unit] = 1;
                        IF (not bbusy[j.]) and (not brw[j.]) =>
                           source = bbusy[j.] = 1),
[#63:#57,#63:#67]:= (IF fbu[unit] and (fb[unit] eql k.) => source = 1 next
                        fb[unit] = k.; fbu[unit] = 1;
                        IF (not bbusy[k.]) and (not bbusy[k.]) =>
                           source = bbusy[k.] = 1),
[#50,#54,#55,#00,
 #64,#65,#70,#74,
             #76]:= (IF fau[unit] and (fa[unit] eql j.) => source = 1 next
                        fa[unit] = j.; fau[unit] = 1;
                        IF (not abusy[j.]) and (not arw[j.]) =>
                           source = bbusy[j.] = 1),
         [#03,#10]:= (IF fxu[unit] and (fx[unit] eql i.) => source = 1 next
                        fx[unit] = i.; fxu[unit] = 1;
                        IF (not xbusy[i.]) and (not xrw[i.]) =>
                           source = xbusy[i.] = 1),
[#11:#13,#16:#17,
 #30:#42,#62,#53,
 #02,#63,#72,#73]:= (IF fxu[unit] and (fx[unit] eql j.) => source = 1 naxt
                        fx[unit] = j.; fxu[unit] = 1;
                        IF (not xbusy[j.]) and (not xrw[j.]) =>
                           source = xbusy[j.] = 1),
[#11:#17,#22:#27,
 #30:#42,#44,#45,
             #47]:= (IF fxu[unit] and (fx[unit] eql k.) => source = 1 next
                        fx[unit] = k.; fxu[unit] = 1;
                        IF (not xbusy[k.]) and (not xrw[k.]) =>
                           source = xbusy[k.] = 1),
       otharwise := source = 1
            and
        end,

    dest()<> :=                 ! Dastination ragistar allocation
        begin
        dest = 0 next
        DECODE fm =>
            begin
         [#10:#45,
#47,#70:#77]:= (fx[unit] = i.; fxu[unit] = 1;
                        IF not xbusy[i.] => dest = xbusy[i.] = xrw[i.] = 1),
         #50:#57 := (fa[unit] = i.; fau[unit] = 1;
                        IF not abusy[i.] => dest = abusy[i.] = arw[i.] = 1),
         [#24:#26,
          #60:#67]:= (fb[unit] = i.; fbu[unit] = 1;
                        IF not bbusy[i.] => dest = bbusy[i.] = brw[i.] = 1),
      otherwise := dest = 1
            and
        end,

    mark :=                             ! Mark stack as invalid
        begin
        islo = ishi = PC
        end,

    dealloc(dunit<3:0>)[critical] :=    ! Daallocate rasources
        begin
```

## APPENDIX 2 (cont'd.)

```
        fbusy[dunit] = 0;
        IF fau[dunit] => (fau[dunit] = abusy[fa[dunit]] = arw[fa[dunit]] = 0);
        IF fbu[dunit] => (fbu[dunit] = bbusy[fb[dunit]] = brw[fa[dunit]] = 0);
        IF fxu[dunit] => (fxu[dunit] = xbusy[fx[dunit]] = xrw[fa[dunit]] = 0)
        end,
    reserv :=
        begin
        unit = 15 next                  ! Mark as "no unit"
        OECODE fm =>
            begin
        #00:#07 := unit = 0,            ! Branch Unit
        #10:#17 := unit = 1,            ! Boolean Unit
  [#20:#27,#43]:= unit = 2,             ! Shift Unit
        #30:#35 := unit = 3,            ! Add Unit
        #36:#37 := unit = 4,            ! Long Add Unit
        #40:#42 := OECODE fbusy[5] =>   ! Multiply Units
                    begin
                    0 := unit = 6,
                    1 := IF not fbusy[6] => unit = 6
                    end,
        #44:#47 := unit = 7,            ! Divide Unit
        #50:#77 := OECODE fbusy[8] =>   ! Increment Units
                    begin
                    0 := unit = 8,
                    1 := IF not fbusy[9] => unit = 9
                    end
                end next
        IF unit neq 15 =>
            begin
            OECODE fbusy[unit] =>
                begin
                0 := OECODE (not dest()) or (not source()) =>
                    begin
                    0 := fbusy[unit] = 1,
                    1 := begin
                         dealloc(unit) next
                         RESTART reserv
                         end
                    end,
                1 := begin
                    WAIT (not fbusy[unit]) next
                    RESTART reserv
                    end
                end
            end
        end,

**Instruction.Fetch**{us}

! Instruction fetch is always from the instruction stack.  If
! the stack is empty (initial power on or branch out of stack),
! or if there are less than three instruction words left in the
! stack, fetch reloads the stack before obtaining an instruction.
! Instructions may be 15 or 30 bits long and aligned on any 15 bit
! boundry.  Fetch obtains 15 bits of an instruction then determines
! if a second 15 bits are required.

    fetch :=
        begin
        IF (PC lss islo) or (PC gtr ishi) => mark() next
        IF (ishi - PC) leq #2 =>
            begin
            islo = PC + ise next
            sfetch :=
                begin
                is[isa] = rni(PC + isa) next
                ishi = PC + ise next
                isa = isa + 1 next
                IF (ishi - PC) lss #7 => RESTART sfetch
                end
            end next
        i0 = ism[isc] next
        px = px + 1 next
        DECODE fm =>                    ! Check for 30 bit instructions
            begin
[#00:#01,#04:#07,
 #30:#37,#50:#52,
 #60:#62,#70:#72]:= (il = ism[isc] next
                     px = px + 1),
       otherwise :=  no.op()
            end
        end,

**Central.Memory.Access**{oc}

! Centeral memory is always accessed indirectly by a user program.
! The Read Next Instruction (RNI) routine is used to load the
! instruction stack.  Touching the A registers 1 through 7 causes
! the corresponding X register to be loaded (A[1:5]) from memory
! or stored (A[6:7]) in memory.

    range(rel<17:0>)<> :=                       ! Address range fault check.
        begin
        range = 0 next
        IF rel geq (FLCM - 1) =>
            begin
            range = 1;                          ! Fault
            DECODE EM<12> =>                    ! Address exit select
                begin
                0 := I = MP[0],                 ! Not selected
                1 := begin
                     MP[RACM]<53:48> = MP[RACM]<53:48> or #010000;
                     MP[RACM]<47:30> = rel + 1 next
                     I = MP[RACM]; PC = 0 next
                     STOP()                     ! Stop the processor
                     end
                end
            end
        end,

    rni(pci<17:0>)<59:0> :=                     ! Read next instruction
        begin
        IF not range(pci) => rni = MP[RACM + pci]
        end,

    aref(reg<2:0>,val<17:0>) :=                 ! A register forced
        begin                                   !   memory access
        AREG[reg] = val next
        range(val) next
        OECODE reg =>
            begin
            #0 := no.op(),
            #1:#5 := (IF range => (XREG[reg] = MP[0] next LEAVE aref) next
                      XREG[reg] = MP[AREG[reg] + RACM]),
            #6:#7 := (IF range => LEAVE aref next
                      MP[AREG[reg] + RACM] = XREG[reg])
            end
        end,

**Exchange.Jump**{us}

! Exchange jump is the central processor's interrupt mechanism.
! Exchange jump is initiated by power on or by one of the ten
! peripheral processors.  All of the central processor's state
! (including all registers) is exchanged with 16 words of centrel
! memory.  The central memory starting address is provided by
! the "interrupting" peripherel processor.  The central memory
! words are formatted such that all of the state can be extracted
! and loaded into the appropriate registers.
! This implementation uses a 16 word holding area (xjp) to format end
! temporarily preserve the old state until the new state is loaded.

    xj :=
        begin
        xjp[00] = PC    @ AREG[0] @ #000000;
        xjp[01] = RACM  @ AREG[1] @ BREG[1];
        xjp[02] = FLCM  @ AREG[2] @ BREG[2];
        xjp[03] = EM    @ AREG[3] @ BREG[3];
        xjp[04] = RAECS @ AREG[4] @ BREG[4];
        xjp[05] = FLECS @ AREG[5] @ BREG[6];
        xjp[06] = MA    @ AREG[6] @ BREG[6];
        xjp[07] =         AREG[7] @ BREG[7];
        xjp[08] = XREG[0];
        xjp[09] = XREG[1];
        xjp[10] = XREG[2];
        xjp[11] = XREG[3];
        xjp[12] = XREG[4];
        xjp[13] = XREG[5];
        xjp[14] = XREG[6];
        xjp[15] = XREG[7] next
        PC    @ AREG[0]         = MP[xja + 00]<53:18>;
        RACM  @ AREG[1] @ BREG[1] = MP[xja + 01];
        FLCM  @ AREG[2] @ BREG[2] = MP[xja + 02];
        EM    @ AREG[3] @ BREG[3] = MP[xja + 03];
        RAECS @ AREG[4] @ BREG[4] = MP[xja + 04];
        FLECS @ AREG[5] @ BREG[5] = MP[xja + 05];
        MA    @ AREG[6] @ BREG[6] = MP[xja + 06];
                AREG[7] @ BREG[7] = MP[xja + 07];
                          XREG[0] = MP[xja + 08];
                          XREG[1] = MP[xja + 09];
                          XREG[2] = MP[xja + 10];
                          XREG[3] = MP[xja + 11];
                          XREG[4] = MP[xja + 12];
                          XREG[5] = MP[xja + 13];
                          XREG[6] = MP[xja + 14];
                          XREG[7] = MP[xja + 15] next
        MP[xja + 00] = xjp[00];
        MP[xja + 01] = xjp[01];
        MP[xja + 02] = xjp[02];
        MP[xja + 03] = xjp[03];
        MP[xja + 04] = xjp[04];
        MP[xja + 05] = xjp[05];
        MP[xja + 06] = xjp[06];
        MP[xja + 07] = xjp[07];
        MP[xja + 08] = xjp[08];
        MP[xja + 09] = xjp[09];
        MP[xja + 10] = xjp[10];
        MP[xja + 11] = xjp[11];
        MP[xja + 12] = xjp[12];
        MP[xja + 13] = xjp[13];
        MP[xja + 14] = xjp[14];
        MP[xja + 15] = xjp[15] next
        xjf = 0
        end,

**Instruction.Cycle**

    start(main) :=
        begin                           ! Initialization
        WAIT (xjf) next                 ! Wait for exchange jump
        stop.bit = 0 next               ! Clear stop bit
        mark() next                     ! Instruction Stack empty
        run :=                          ! Main cycle
            begin
            IF xjf => xj() next         ! Check for exchange jump
            IF stop.bit => RESTART start next
            IF not range =>
                begin
                fetch() next            ! Get an instruction
                reserv()                ! Reservation control
                end next                ! will not return until
            range = 0 next              ! all usage conflicts are
                                        ! resolved.
            exec() next                 ! Issue the instruction
            RESTART run
            end
        end,
```

## APPENDIX 2 (cont'd.)

```
    exec :=                             ! The instruction is issued
        begin                           ! to the appropiate execution
        DECODE unit =>                  ! unit.
            begin
            0 := BRANCH.UNIT(I),
            1 := BOOLEAN.UNIT(1),
            2 := SHIFT.UNIT(I),
            3 := ADD.UNIT(I),
            4 := LONG.ADD.UNIT(1),
            5 := MULTIPLY.UNIT.0(I),
            6 := MULTIPLY.UNIT.1(I),
            7 := DIVIDE.UNIT(I),
            8 := INCREMENT.UNIT.0(I),
            9 := INCREMENT.UNIT.1(1)
            end
        end,
! The remainder of the ISP describes the ten arithmetic processing
! units.  These units will function in parallel much as they do
! in the real CDC 6600.

! Note that floating point instructions are decoded but this ISP
! does not describe their actual execution.

**Branch.Unit**

BRANCH.UNIT(i<29:0>)[process; critical] :=
        begin

**Branch.Declarations**

        fm <5:0> := i<29:24>,
        i. <2:0> := i<23:21>,
        j. <2:0> := i<20:18>,
        k. <2:0> := i<17:15>,

    **Branch.Execution**{oc}

    branch[main] :=
        begin
        DECODE fm @ i. =>
            begin
            #00?? := PS  := stop.bit = 1,
            #010 := RJ  := (MP[k1+RACM] = #04008(PC+1)<17:0>8#0000000000 next
                           PC = k1 + 1; mark()),
            #022? := JP  := (PC = k1 + BREG[i.]; mark()),
            #030 := ZR  :=  IF XREG[j.] eql 0 => PC = k1,
            #031 := NZ  :=  IF XREG[j.] neq 0 => PC = k1,
            #032 := PL  :=  IF not XREG[j.]<59> => PC = k1,
            #033 := NG  :=  IF XREG[j.]<59> => PC = k1,
            #034 := IR  :=  IF not ((XREG[j.]<59:48> eql{us} #3777) or
                               (XREG[j.]<59:48> eql{us} #4000)) => PC = k1,
            #035 := OR. :=  IF      (XREG[j.]<59:48> eql{us} #3777) or
                               (XREG[j.]<59:48> eql{us} #4000)  => PC = k1,
            #03B := OF  :=  IF not ((XREG[j.]<59:48> eql{us} #1777) or
                               (XREG[j.]<59:48> eql{us} #6000)) => PC = k1,
            #037 := ID  :=  IF      (XREG[j.]<59:48> eql{us} #1777) or
                               (XREG[j.]<59:48> eql{us} #6000)  => PC = k1,
            #04?? := EQ  :=  IF BREG[i.] eql{us} BREG[j.] => PC = k1,
            #05?? := NE  :=  IF BREG[i.] neq{us} BREG[j.] => PC = k1,
            #06?? := GE  :=  IF BREG[i.] geq{us} BREG[j.] => PC = k1,
            #07?? := LT  :=  IF BREG[i.] lss{us} BREG[j.] => PC = k1
            end next
        IF (PC lss{us} islo) or (PC gtr{us} ishi) => mark() next
        dealloc(0)
        end
    end,
**Boolean.Unit**

BOOLEAN.UNIT(i<29:0>)[process; critical] :=
    begin

**Boolean.Declarations**

        fm <5:0> := i<29:24>,
        i. <2:0> := i<23:21>,
        j. <2:0> := i<20:18>,
        k. <2:0> := i<17:15>,

    **Boolean.Execution**{us}

    boolean[main] :=
        begin
        DECODE fm =>                    ! All instructions are "BXi"
            begin
            #10 := XREG[i.] = XREG[j.],
            #11 := XREG[i.] = XREG[j.] and XREG[k.],
            #12 := XREG[i.] = XREG[j.] or  XREG[k.],
            #13 := XREG[i.] = XREG[j.] xor XREG[k.],
            #14 := XREG[i.] = not XREG[k.],
            #15 := XREG[i.] = XREG[j.] and (not XREG[k.]),
            #16 := XREG[i.] = XREG[j.] or  (not XREG[k.]),
            #17 := XREG[i.] = XREG[j.] xor (not XREG[k.])
            end next
        dealloc(1)
        end
    end,
**Shift.Unit**

SHIFT.UNIT(i<29:0>)[process; critical] :=
    begin

**Shift.Declarations**

        fm <5:0> := i<29:24>,
        i. <2:0> := i<23:21>,
        j. <2:0> := i<20:18>,
        k. <2:0> := i<17:15>,
        jk <5:0> := i<20:15>,

**Shift.Execution**{us}

shift[main] :=
    begin
    DECODE fm =>
        begin
        #20 := XREG[i.] = XREG[i.] slr jk,          ! LXi
        #21 := XREG[i.] = XREG[i.] srr jk,          ! AXi
        #22 := DECODE BREG[j.]<17> =>               ! LXi
                begin
                0 := XREG[i.] = XREG[k.] slr BREG[j.]<5:0>,
                1 := DECODE (not BREG[j.]<10:6>) eql '00000 =>
                        begin
                        0 := XREG[i.] = 0,
                        1 := XREG[i.]
                             = XREG[k.] srd (not BREG[j.]<10:0>)
                        end
                end,
        #23 := DECODE BREG[j.]<17> =>               ! AXi
                begin
                0 := DECODE BREG[j.]<10:6> eql '00000 =>
                        begin
                        0 := XREG[i.] = 0,
                        1 := XREG[i.] = XREG[k.] srd BREG[j.]
                        end,
                1 := XREG[i.] = XREG[k.] slr (not BREG[j.]<5:0>)
                end,
        #24 := not.described,                       ! NXi
        #25 := not.described,                       ! ZXi
        #26 := begin                                ! UXi
               XREG[i.] <= XREG[k.]<59> @ XREG[k.]<47:0>;
               BREG[j.] <= #2000 -(us) XREG[k.]<58:48>
               end,
        #27 := begin                                ! PXi
               XREG[i.]<47:0> = XREG[k.]<47:0>;
               XREG[i.]<69> = XREG[k.]<59>;
               DECODE XREG[k.]<59> =>
                   begin
                   0 := XREG[i.]<58:48> = not BREG[j.]<10>
                                            @ BREG[j.]<9:0>,
                   1 := XREG[i.]<68:48> = BREG[j.]<10>
                                            @ not BREG[j.]<9:0>
                   end
               end,
        #43 := begin                                ! MXi
               XREG[i.] = 0 next
               XREG[i.]<59> = (jk neq 0) next
               XREG[i.] = XREG[i.] srd (jk -{us} 1)
               end
        end next
    dealloc(2)
    end
  end.

**Add.Unit**

ADD.UNIT(i<29:0>)[process; critical] :=
    begin

**Add.Declarations**

    **Add.Execution**{oc}

    add[main] :=
        begin
        DECODE fm =>
            begin
            #30 := not.described,                   ! FXi -> (Xj + Xk)
            #31 := not.described,                   ! FXi -> (Xj - Xk)
            #32 := not.described,                   ! DXi -> (Xj + Xk)
            #33 := not.described,                   ! DXi -> (Xj - Xk)
            #34 := not.described,                   ! RXi -> (Xj + Xk)
            #35 := not.described                    ! RXi -> (Xj - Xk)
            end next
        dealloc(3)
        end
    end,

**Long.Add.Unit**

LONG.ADD.UNIT(i<29:0>)[process; critical] :=
    begin

**Long.Add.Declarations**

        fm <6:0> := i<29:24>,
        i. <2:0> := i<23:21>,
        j. <2:0> := i<20:18>,
        k. <2:0> := i<17:15>,

    **Long.Add.Execution**{oc}

    ladd[main] :=
        begin
        DECODE fm =>
            begin
            #36 := XREG[i.] = XREG[j.] + XREG[k.],      ! IXi -> Xj + Xk
            #37 := XREG[i.] = XREG[j.] - XREG[k.],      ! IXi -> Xj - Xk
      otherwise := no.op()
            end next
        dealloc(4)
        end
    end,

**Multiply.Unit.0**

MULTIPLY.UNIT.0(i<29:0>)[process; critical] :=
    begin

**Multiply.0.Decleretions**
```

## APPENDIX 2 (cont'd.)

```
        fm <5:0> := i<29:24>,

    **Multiply.0.Execution**[oc]

    mpy0[main] :=
        begin
        DECODE fm =>
            begin
            #40 := not.described,                ! FXi -> Xj * Xk
            #41 := not.described,                ! RXi -> Xj * Xk
            #42 := not.described                 ! DXi -> Xj * Xk
            end next
        dealloc(5)
        end
    end,

**Multiply.Unit.1**

MULTIPLY.UNIT.1(i<29:0>)[process; critical] :=
    begin

**Multiply.1.Declarations**

        fm <5:0> := i<29:24>,

    **Multiply.1.Execution**[oc]

    mpy1[main] :=
        begin
        DECODE fm =>
            begin
            #40 := not.described,                ! FXi -> Xj * Xk
            #41 := not.described,                ! RXi -> Xj * Xk
            #42 := not.described                 ! DXi -> Xj * Xk
            end next
        dealloc(8)
        end
    end,

**Divide.Unit**

DIVIDE.UNIT(i<29:0>)[process; critical] :=
    begin

**Divide.Declarations**

        fm <5:0> := i<29:24>,
        m. <2:0> := i<26:24>,
        i. <2:0> := i<23:21>,
        j. <2:0> := i<20:18>,
        k. <2:0> := i<17:15>,
        k1<17:0> := i<17:0>,
        xcnt<5:0>,                           ! Counter for CXi

    **Divide.Execution**[oc]

    div[main] :=
        begin
        DECODE fm =>
            begin
            #44 := not.described,                ! FXi -> Xi = Xj / Xk
            #45 := not.described,                ! RXi -> Xi = Xj / Xk
            #46 := NO  := no.op(),
            #47 := CXi
                := begin
                   xcnt = 0;
                   XREG[i.] = 0 next
                   CXi. :=
                      begin
                      XREG[i.] = XREG[i.] +{us} XREG[k.]<0> next
                      XREG[k.] = XREG[k.] srr 1;
                      xcnt = xcnt + 1 next
                      IF xcnt lss{us} 60 => RESTART CXi.
                      end
                   end
            end next
        dealloc(7)
        end
    end,

**Increment.Unit.0**

INCREMENT.UNIT.0(i<29:0>)[process; critical] :=
    begin

**Increment.0.Declarations**

        fm <5:0> := i<29:24>,
        m. <2:0> := i<26:24>,
        i. <2:0> := i<23:21>,
        j. <2:0> := i<20:18>,
        k. <2:0> := i<17:15>,
        k1<17:0> := i<17:0>,

    **Increment.0.Execution**[oc]

    incr0[main] :=
        begin
        DECODE fm =>
            begin
        #50:#57 := SAi                           ! Increment
                := begin
                   DECODE m. =>
                      begin
                      #0 := aref(i.,AREG[j.] + k1),
                      #1 := aref(i.,BREG[j.] + k1),
                      #2 := aref(i.,XREG[j.]<17:0> + k1),
                      #3 := aref(i.,XREG[j.]<17:0> + BREG[k.]),
                      #4 := aref(i.,AREG[j.] + BREG[k.]),
                      #5 := aref(i.,AREG[j.] - BREG[k.]),
                      #6 := aref(i.,BREG[j.] + BREG[k.]),
                      #7 := aref(i.,BREG[j.] - BREG[k.])
                      end
                   end,
        #60:#67 := SBi
                := begin
                   DECODE m. =>
                      begin
                      #0 := BREG[i.] = AREG[j.] + k1,
                      #1 := BREG[i.] = BREG[j.] + k1,
                      #2 := BREG[i.] = XREG[j.]<17:0> + k1,
                      #3 := BREG[i.] = XREG[j.]<17:0> + BREG[k.],
                      #4 := BREG[i.] = AREG[j.] + BREG[k.],
                      #5 := BREG[i.] = AREG[j.] - BREG[k.],
                      #6 := BREG[i.] = BREG[j.] + BREG[k.],
                      #7 := BREG[i.] = BREG[j.] - BREG[k.]
                      end
                   end,
        #70:#77 := SXi
                := begin
                   DECODE m. =>
                      begin
                      #0 := XREG[i.] <= AREG[j.] + k1,
                      #1 := XREG[i.] <= BREG[j.] + k1,
                      #2 := XREG[i.] <= XREG[j.]<17:0> + k1,
                      #3 := XREG[i.] <= XREG[j.]<17:0> + BREG[k.],
                      #4 := XREG[i.] <= AREG[j.] + BREG[k.],
                      #5 := XREG[i.] <= AREG[j.] - BREG[k.],
                      #6 := XREG[i.] <= BREG[j.] + BREG[k.],
                      #7 := XREG[i.] <= BREG[j.] - BREG[k.]
                      end
                   end
            end next
        dealloc(8)
        end
    end,

**Increment.Unit.1**

INCREMENT.UNIT.1(i<29:0>)[process; critical] :=
    begin

**Increment.1.Declarations**

        fm <5:0> := i<29:24>,
        m. <2:0> := i<26:24>,
        i. <2:0> := i<23:21>,
        j. <2:0> := i<20:18>,
        k. <2:0> := i<17:15>,
        k1<17:0> := i<17:0>,

    **Increment.1.Execution**[oc]

    incr1[main] :=
        begin
        DECODE fm =>
            begin
        #50:#57 := SAi                           ! Increment
                := begin
                   DECODE m. =>
                      begin
                      #0 := aref(i.,AREG[j.] + k1),
                      #1 := aref(i.,BREG[j.] + k1),
                      #2 := aref(i.,XREG[j.]<17:0> + k1),
                      #3 := aref(i.,XREG[j.]<17:0> + BREG[k.]),
                      #4 := aref(i.,AREG[j.] + BREG[k.]),
                      #5 := aref(i.,AREG[j.] - BREG[k.]),
                      #6 := aref(i.,BREG[j.] + BREG[k.]),
                      #7 := aref(i.,BREG[j.] - BREG[k.])
                      end
                   end,
        #60:#67 := SBi
                := begin
                   DECODE m. =>
                      begin
                      #0 := BREG[i.] = AREG[j.] + k1,
                      #1 := BREG[i.] = BREG[j.] + k1,
                      #2 := BREG[i.] = XREG[j.]<17:0> + k1,
                      #3 := BREG[i.] = XREG[j.]<17:0> + BREG[k.],
                      #4 := BREG[i.] = AREG[j.] + BREG[k.],
                      #5 := BREG[i.] = AREG[j.] - BREG[k.],
                      #6 := BREG[i.] = BREG[j.] + BREG[k.],
                      #7 := BREG[i.] = BREG[j.] - BREG[k.]
                      end
                   end,
        #70:#77 := SXi
                := begin
                   DECODE m. =>
                      begin
                      #0 := XREG[i.] <= AREG[j.] + k1,
                      #1 := XREG[i.] <= BREG[j.] + k1,
                      #2 := XREG[i.] <= XREG[j.]<17:0> + k1,
                      #3 := XREG[i.] <= XREG[j.]<17:0> + BREG[k.],
                      #4 := XREG[i.] <= AREG[j.] + BREG[k.],
                      #5 := XREG[i.] <= AREG[j.] - BREG[k.],
                      #6 := XREG[i.] <= BREG[j.] + BREG[k.],
                      #7 := XREG[i.] <= BREG[j.] - BREG[k.]
                      end
                   end
            end next
        dealloc(9)
        end
    end,

REQUIRE.ISP |PC6600.isp|.       ! Peripheral Processor Description

end             ! End CDC 6600
```