

Chapter 43

Parallel Operation in the Control Data 6600¹

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, 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

¹AFIPS Proc. FJCC, pt. 2, vol. 26, 1964, pp. 33-40.

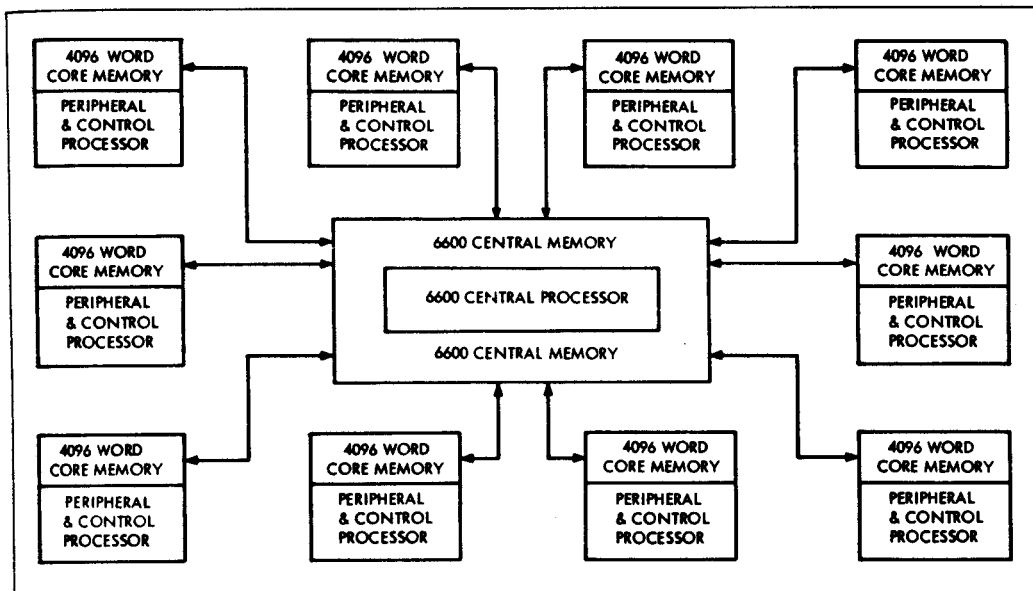


Fig. 1. Control Data 6600.

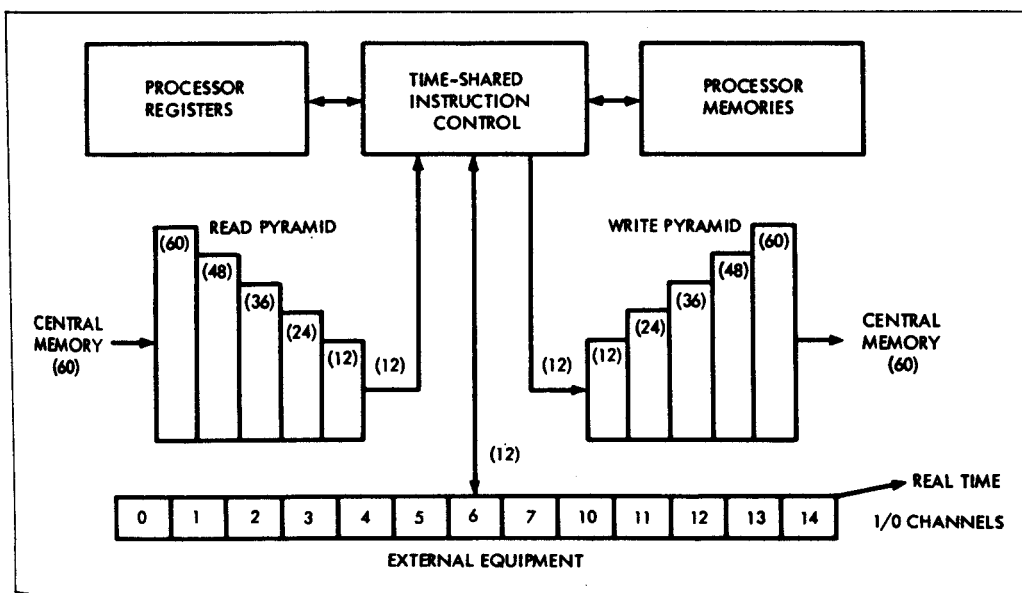


Fig. 2. 6600 peripheral and control processors.

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

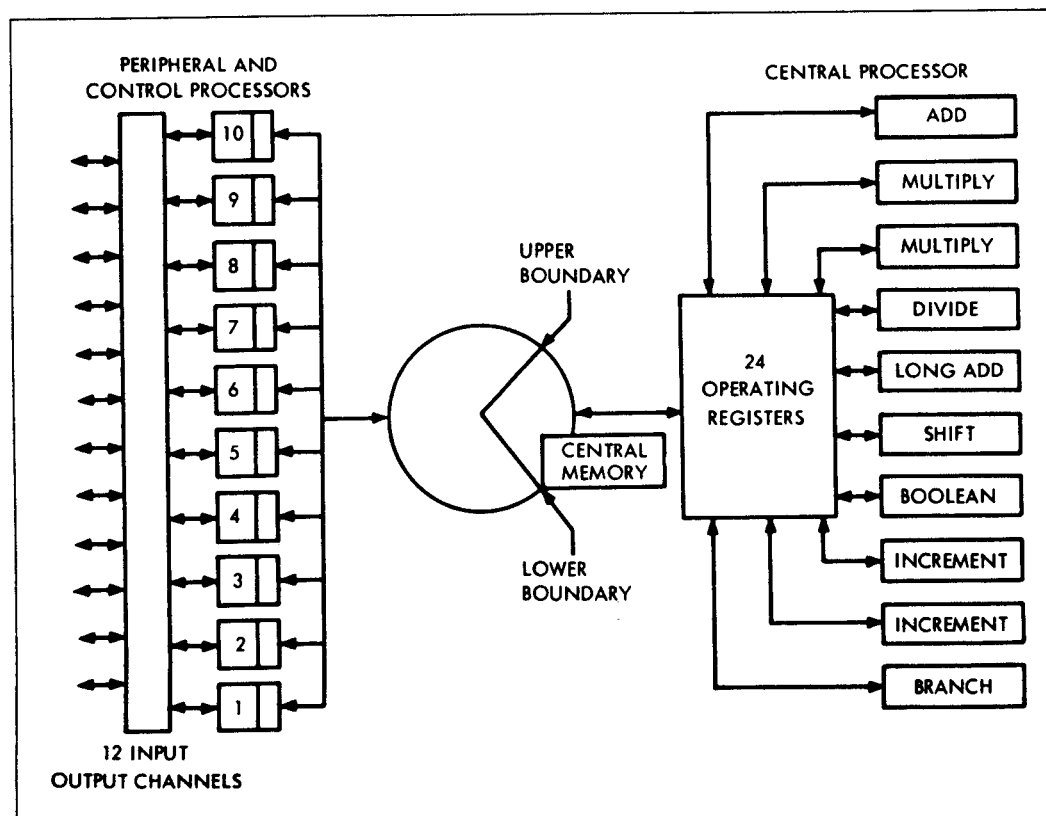


Fig. 3. Block diagram of 6600.

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.

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

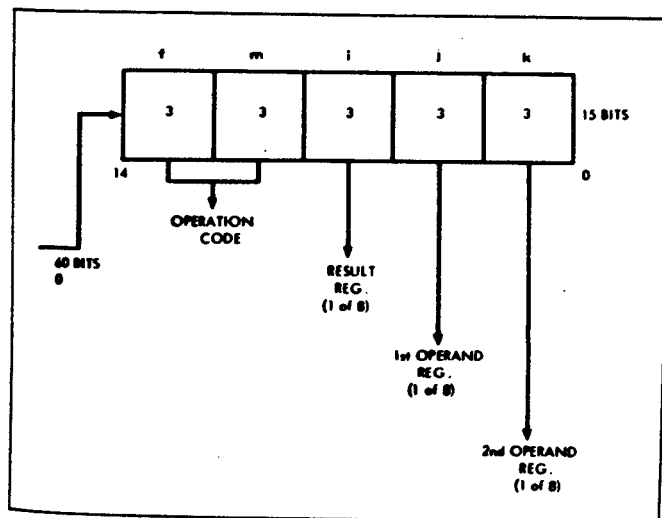


Fig. 4. Fifteen-bit instruction format.

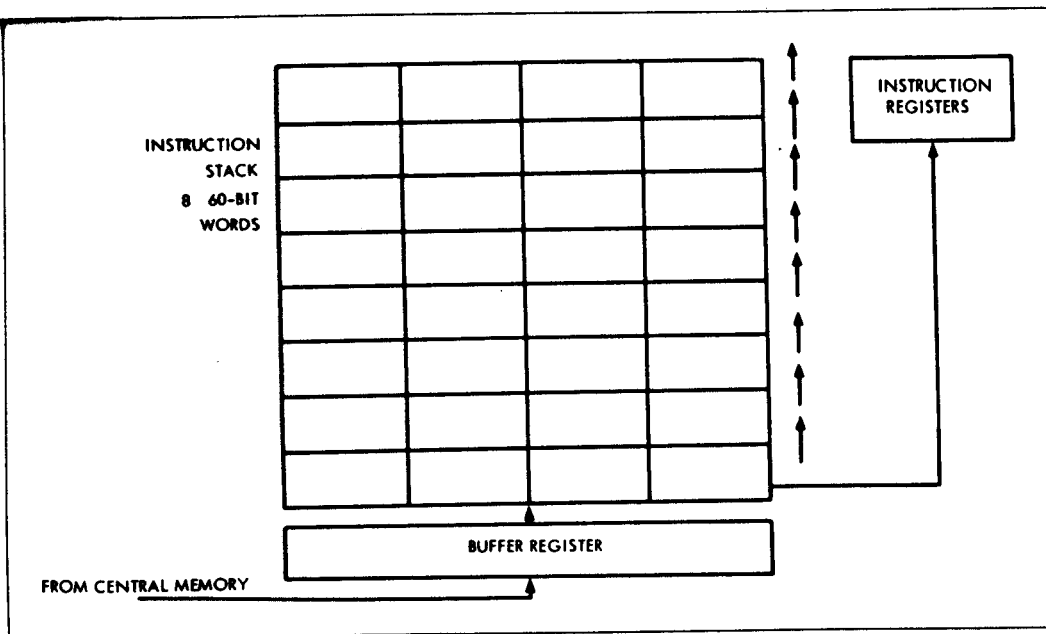


Fig. 5. 6600 instruction stack operation.

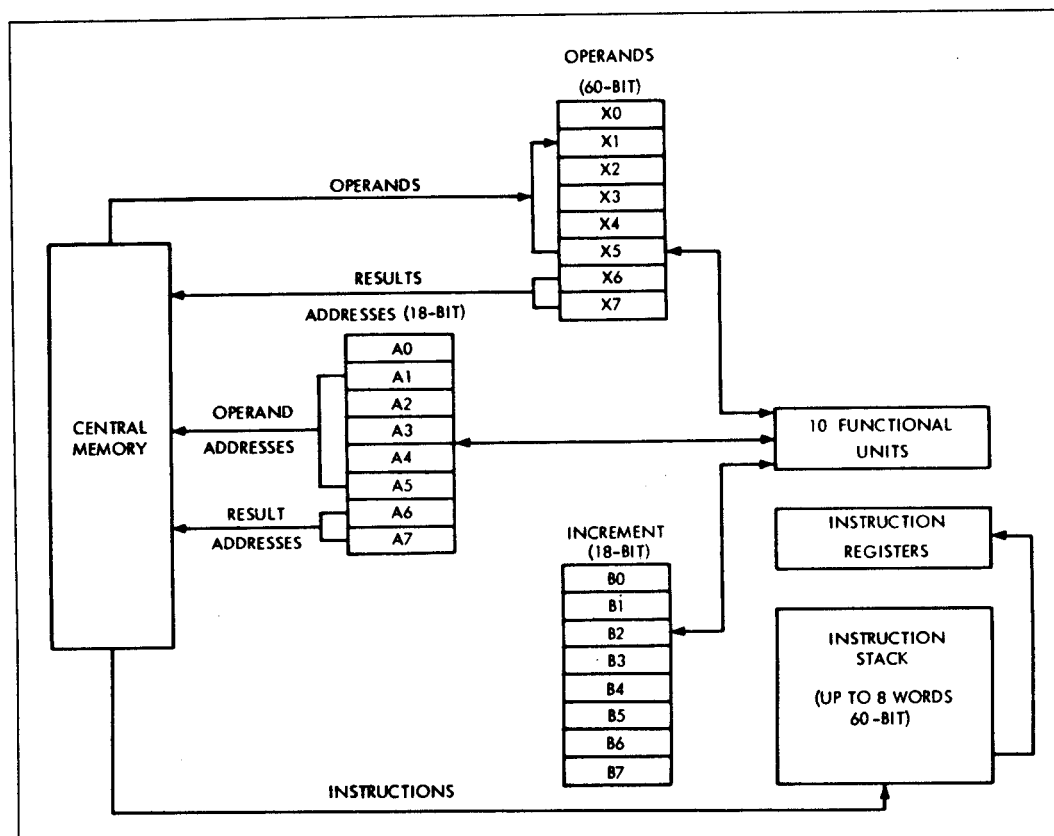


Fig. 6. Central processor operating registers.

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; 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.

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). A silicon transistor operates in saturation when switched

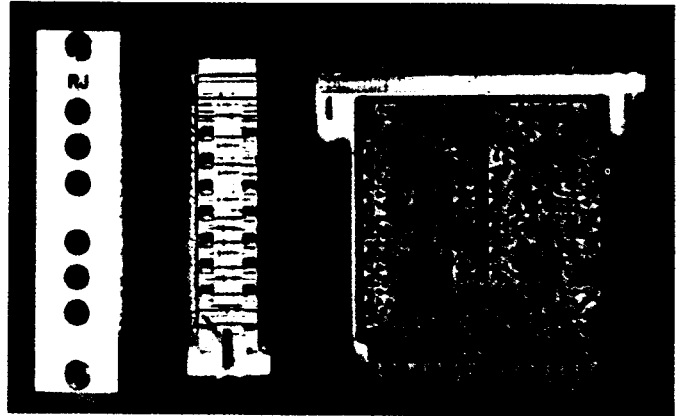


Fig. 7. 6600 printed circuit module.

"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

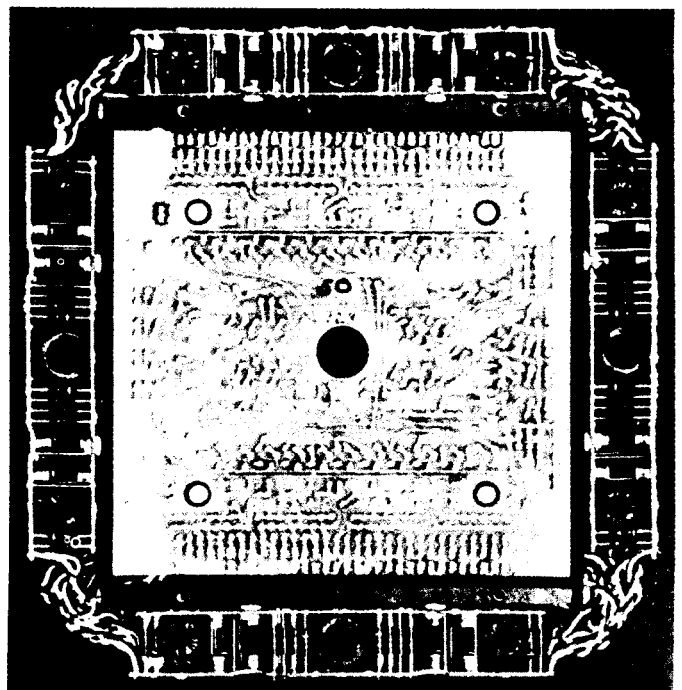


Fig. 8. 6600 memory module.

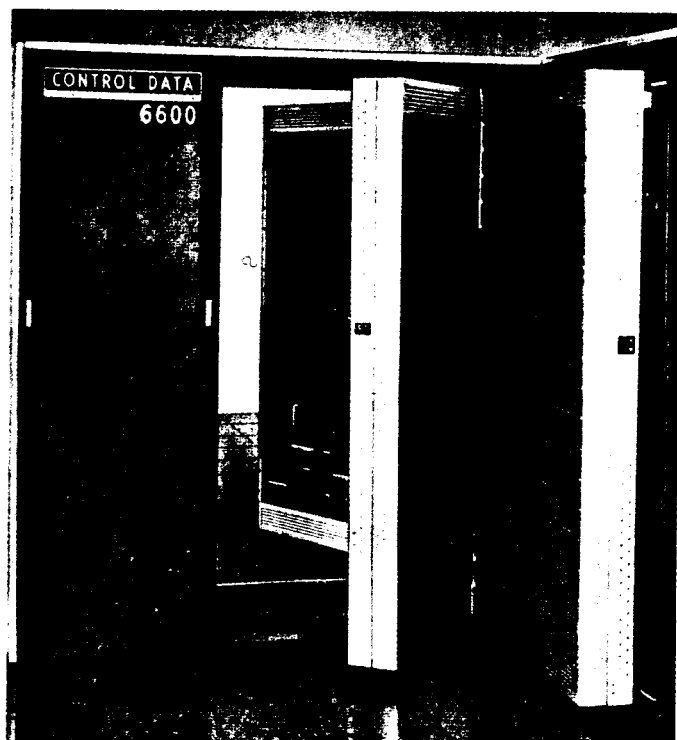


Fig. 9. 6600 main frame section.

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

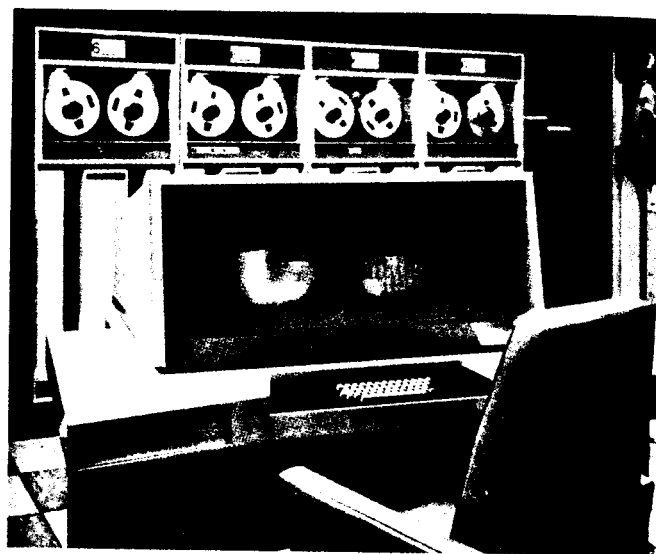


Fig. 10. 6600 display console.

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.

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. Initial version by Gary Leive (ca. 1978)

! 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]<0>,          ! Channel active indicator
cful[0:11]<0>,          ! 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 < 8:0> := pir<23:18>,
d < 8:0> := 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 eq 0 =>
begin
0 := begin
index = m + M.PCP[d];
P[id] = P[id] + 1
end,
1 := index = m
end
end,

**Barrel.Execution**

barrel(main) :=
begin
! Activate processor 0
pcp(0) next
! Activate processor 1
pcp(1) next
! Activate processor 2
pcp(2) next
! Activate processor 3
pcp(3) next
! Activate processor 4
pcp(4) next
! Activate processor 5
pcp(5) next
! Activate processor 6
pcp(6) next
! Activate processor 7
pcp(7) next
! Activate processor 8
pcp(8) next
! Activate processor 9
pcp(9) next
! Do it all again
RESTART barrel
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, #26 := no.op(), ! PSN - Pass
! LDW - Load d into a
#14 := A[id] = d, ! LCN - Load complement d
#15 := A[id] = #7777(not d), ! LDD - Load (d)
#30 := A[id] = M.PCP[d], ! STD - Store (d)
#34 := M.PCP[d] = A[id], ! LDI - Load ((d))
#40 := A[id] = M.PCP[M.PCP[d]], ! STI - Store ((d))
#44 := M.PCP[M.PCP[d]] = A[id], ! LDC - Load dm
#20 := begin
A[id] = dm; P[id]=P[id]+1
end,
#50 := A[id] = M.PCP[index(id)], ! LDW - Load (m + (d))
#54 := M.PCP[index(id)] = A[id], ! STM - Store (m + (d))
#16 := A[id] = A[id] + (us) d, ! ADM - Add d
#17 := A[id] = A[id] - (us) d, ! SBW - 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), ! ADI - 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), ! SBW - Subtract (m + (d))
M.PCP[index(id)],
#10 := DECODE d<6> => ! SHW - Shift d
begin
0 := A[id] = A[id] slr d,
1 := A[id] = A[id] srl (not d)
end,
#11 := A[id]<5:0> = A[id]<5:0>, ! LMN - Logical difference d
xor d,
#12 := A[id] = A[id] and d, ! LPN - Logical product 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>, ! LMI - 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 - Logical difference
A[id]<11:0> xor M.PCP[index(id)], ! (m + (d))
#35 := M.PCP[d] = A[id] + M.PCP[d], ! 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]],
#46 := M.PCP[M.PCP[d]] = A[id], ! AOI - Replace add one ((d))
= M.PCP[M.PCP[d]] + 1,
#47 := M.PCP[M.PCP[d]] = A[id], ! SOI - Replace subtract 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)],
#56 := M.PCP[index] = A[id], ! AOM - Replace add one
= M.PCP[index(id)] + 1, ! (m + (d))
#57 := M.PCP[index] = A[id], ! SOM - 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] eq (us) 0 => ! ZJN - Zero Jump d
P[id] = (P[id] - 1) + d, ! NJN - Nonzero jump d
#05 := IF A[id] neq (us) 0 => ! PJN - Plus jump d
P[id] = (P[id] - 1) + d, ! MJN - Minus jump d
#07 := IF A[id] lss (us) 0 => ! LJM - Long jump to m + (d)
P[id] = index(id),
#01 := P[id] = index(id), ! RJM - Return jump to m + (d)
begin
M.PCP[index(id)] = P[id] + 2 next
P[id] = index + 1
end,
#26 := begin
WAIT (xjf eqv '0) next ! EXN - Exchange Jump
xja = A[id]; xjf = 1
end,
#27 := A[id] = pc, ! RPN - Read program address
#60 := begin
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] next
end,
#61 := begin
M.PCP[0] = P[id] + 1 next ! CRM - Central read (d)
P[id] = m; Q[id] = d next ! words from (A) to m
CRMD :=
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 CRMD
end next
P[id] = M.PCP[0]
end,
#62 := begin
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
end,
#63 := begin
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

```


APPENDIX 1 (cont'd.)

```

write[2] = M.PCP[P[id]+1] next
write[3] = M.PCP[P[id]+1] next
write[4] = M.PCP[P[id]+4] next
P[id] = P[id] + 6;
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]
end.
#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 := begin
  M.PCP[0] = P[id] + 1 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 ! clear next

```

```

IF cact[d] =>
begin
  M.PCP[P[id]] = CHAN[d] next ! input
  P[id] = P[id] + 1;          ! increment address
  A[id] = A[id] - 1 next;     ! decrement count
  IF A[id] neq 0 => RESTART IAM0
end
end next
P[id] = M.PCP[0]              ! restore program counter
end.
#72 := CHAN[d] = A[id],      ! OAN - Output from A
begin                        ! on channel d
#73 := begin
  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[P[id]] next ! output
  P[id] = P[id] + 1;          ! increment address
  A[id] = A[id] - 1 next;     ! decrement count
  RESTART OAM0
end
end next
P[id] = M.PCP[0]              ! Restore program counter
end.
#74 := cact[d] = 1,          ! ACM - Activate channel d
#75 := cact[d] = 0,          ! DCN - Disconnect channel d
#76 := CHAN[d] = A[id],      ! FAN - Function (A) on CHAN d
#77 := begin
  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

```

CDC6600(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 (PC6600.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, the resources
! are released by marking the scorecard.

! The following page by page index of the 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 variables.
! **Reservation.Control.State** defines variables used by
! reservation control. These declarations constitute the
! resource allocation "scorecard".
! Describe the reservation control execution.
! **Instruction.Fetch** describes the instruction stack
! control and instruction fetch processes.
! **Central.Memory.Access** describes the instruction read
! and the register associated memory access processes.
! **Exchange.Jump** is the processor interrupt facility.
! **Instruction.Cycle** is the main instruction processing
! cycle. Instruction execution is initiated by issuing
! the instructions to the appropriate functional unit.

! The functional units are:
!
!   Branch Unit.
!   Boolean 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]<69:0>. ! Use only 4k of 60 bit memory

! **Processor.State**
! Exchange Jump Package
xjp[0:15]<69:0>.
! Exchange Jump Address
xja[16:0>.
! Exchange Jump Flag
xjf<0>.
! Pseudo program counter
px[19:0>.
! Program counter
pc[17:0> := px[19:2>.
! Instruction length count
ilc[1:0> := px[1:0>.
! Instruction stack counter
isc[4:0> := px[4:0>.
! A registers
AREG[0:7]<17:0>.
! B registers
BREG[0:7]<17:0>.
! X registers
XREG[0:7]<69:0>.
! Ref Address (central memory)
RACM[17:0>.
! Field length of program
FLCM[17:0>.
! Reference Address for ECS
RAECS[23:0>.
! Field length for ECS
FLECS[23:0>.
! Program exit mode
EMC[17:0>.
! Monitor exchange
MAC[17:0>.

! **Instruction.Format**
! Instruction register
I[29:0>.
! Short instruction (15 bit)
is[14:0> := I[29:15>.
! Long instruction extension
l[14:0> := I[14:0>.

! f[2:0> := I[29:27>.
! m[2:0> := I[28:24>.
! fm[5:0> := I[29:24>.
! l[2:0> := I[23:21>.
! j[2:0> := I[20:18>.
! k[2:0> := I[17:15>.
! kl[17:0> := I[17:0>.

! Instruction stack
is[0:7]<69:0>.
! High address limit in stack
ish[17:0>.
! Low address limit in stack
islo[17:0>.
! Stack insert counter
isa[2:0>.

! **Implementation.Declarations**
stop.bit<0>. ! Stop flag

```

```

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

! **Reservation.Control.State**
! A registers busy bits
arw[0:7]<0>.
! A registers read(0)/write(1)
bbusy[0:7]<0>.
! B registers busy bits
brw[0:7]<0>.
! B registers read(0)/write(1)
xbusy[0:7]<0>.
! X registers busy bits
xrw[0:7]<0>.
! X registers read(0)/write(1)

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

! The following tables are
! used to deallocate the
! resource assignments either
! in the event of conflict during
! allocation, or during deallocation
! at instruction completion.
! ???U<> indicates usage of the
! registers by a unit.
! 1 = used, 0 = not used
! Functional Unit A register
! A register usage
! Functional Unit B register
! B register usage
! Functional Unit X register
! X register usage

unit[3:0>. ! Temporary for arith unit number

! **Reservation.Control**{us}
! Source register allocation
source[0> :=
begin
source = 0 next
DECODE fm =>
begin
#01 := IF (i. eq1 #1) or (i. eq1 #2) =>
! IF fbu[unit] and (fb[unit] eq1 j.) => source = 1 next
! 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] eq1 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,#57,#61,
#66,#67,#71,
#76,#77] := (IF fbu[unit] and (fb[unit] eq1 j.) => source = 1 next
! fb[unit] = j.; fbu[unit] = 1;
! IF (not bbusy[j.]) and (not brw[j.]) =>
! source = bbusy[j.] = 1.
[#53,#57,#63,#67] := (IF fbu[unit] and (fb[unit] eq1 k.) => source = 1 next
! fb[unit] = k.; fbu[unit] = 1;
! IF (not bbusy[k.]) and (not brw[k.]) =>
! source = bbusy[k.] = 1.
[#50,#54,#55,#60,
#64,#65,#70,#74,
#75] := (IF fau[unit] and (fa[unit] eq1 j.) => source = 1 next
! fa[unit] = j.; fau[unit] = 1;
! IF (not abusy[j.]) and (not arw[j.]) =>
! source = abusy[j.] = 1.
[#03,#10] := (IF fau[unit] and (fa[unit] eq1 i.) => source = 1 next
! fa[unit] = i.; fau[unit] = 1;
! IF (not xbusy[i.]) and (not xrw[i.]) =>
! source = xbusy[i.] = 1.
[#11,#13,#15,#17,
#30,#42,#52,#53,
#62,#63,#72,#73] := (IF fxu[unit] and (fx[unit] eq1 j.) => source = 1 next
! 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,#46,
#47] := (IF fxu[unit] and (fx[unit] eq1 k.) => source = 1 next
! fx[unit] = k.; fxu[unit] = 1;
! IF (not xbusy[k.]) and (not xrw[k.]) =>
! source = xbusy[k.] = 1.
otherwise := source = 1
end,
end,
end,
! Destination register allocation
dest[0> :=
begin
dest = 0 next
DECODE fm =>
begin
[#10,#45,
#47,#70,#77] := (fx[unit] = i.; fau[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
end,
end,
end,
! Mark stack as invalid
mark :=
begin
islo = ishi = PC
end,
end,
! Deallocate resources
dealloc(dunit[3:0>){critical} :=
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[fb[dunit]] = 0);
If ffx[dunit] => (ffx[dunit] = xbusy[fx[dunit]] = xrw[fx[dunit]] = 0);
end.
reserv :=
begin
unit = 15 next
DECODE 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 := DECODE fbusy[5] => ! Multiply Units
begin
0 := unit = 5,
1 := If not fbusy[6] => unit = 6
end.
#44:#47 := unit = 7,      ! Divide Unit
#50:#77 := DECODE fbusy[8] => ! Increment Units
begin
0 := unit = 8,
1 := If not fbusy[9] => unit = 9
end
end next
If unit neq 15 =>
begin
DECODE fbusy[unit] =>
begin
0 := DECODE (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
! boundary. 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 + isa next
sfetch :=
begin
is[isa] = rni(PC + isa) next
ishi = PC + isa 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 =>
begin
! Check for 30 bit instructions
[#00:#01, #04:#07,
#30:#37, #50:#52,
#60:#62, #70:#72] := (i1 = ism[isc] next
px = px + 1).
otherwise := no.op()
end
end.

**Central.Memory.Access**(oc)
! Central 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>)<> :=
begin
! Address range fault check.
range = 0 next
If rel geq (FLCM - 1) =>
begin
range = 1;
DECODE EM<12> =>
begin
0 := 1 = MP[0],
1 := begin
MP[RACM]<53:48> = MP[RACM]<53:48> or #010000;
MP[RACM]<47:30> = rel + 1 next
1 = MP[RACM]; PC = 0 next
STOP()
end
end
end.

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

aref(reg<2:0>,val<17:0>) :=
begin
! A register forced
! memory access
AREG[reg] = val next
range(val) next
DECODE 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 central
! memory. The central memory starting address is provided by
! the "interrupting" peripheral 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 and
! 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[5];
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
stop.bit = 0 next
mark() next
run :=
begin
! Main cycle
If xjf => xj() next
! Check for exchange jump
If stop.bit => RESTART start next
If not range =>
begin
fetch() next
reserv()
end next
range = 0 next
! all usage conflicts are
! resolved.
! Issue the instruction
exec() next
RESTART run
end
end.

```

APPENDIX 2 (cont'd.)

```

-- exec :=
begin
  DECODE unit =>
  begin
    0 := BRANCH.UNIT(1);
    1 := BOOLEAN.UNIT(1);
    2 := SHIFT.UNIT(1);
    3 := ADD.UNIT(1);
    4 := LONG.ADD.UNIT(1);
    5 := MULTIPLY.UNIT.0(1);
    6 := MULTIPLY.UNIT.1(1);
    7 := DIVIDE.UNIT(1);
    8 := INCREMENT.UNIT.0(1);
    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] = #0400@PC+1)<17:0>#0000000000 next
      PC = k1 + 1; mark(i);
      #02?? := JP := (PC = k1 + BREG[i.]; mark(i));
      #030 := ZR := IF XREG[j.] eq 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> eq(us) #3777) or
        (XREG[j.]<59:48> eq(us) #4000)) => PC = k1;
      #035 := OR := IF
        (XREG[j.]<59:48> eq(us) #3777) or
        (XREG[j.]<59:48> eq(us) #4000) => PC = k1;
      #036 := DF := IF not ((XREG[j.]<59:48> eq(us) #1777) or
        (XREG[j.]<59:48> eq(us) #6000)) => PC = k1;
      #037 := ID := IF
        (XREG[j.]<59:48> eq(us) #1777) or
        (XREG[j.]<59:48> eq(us) #6000) => PC = k1;
      #04?? := EQ := IF BREG[i.] eq(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) lshi) or (PC gtr(us) lshi) => mark(i) 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 =>
      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[j.] sllr jk; ! LX1
        #21 := XREG[i.] = XREG[j.] srlr jk; ! AX1
        #22 := DECODE BREG[j.]<17> => ! LX1
        begin
          0 := XREG[i.] = XREG[k.] sllr BREG[j.]<5:0>;
          1 := DECODE (not BREG[j.]<10:0>) eq1 '00000 =>
          begin
            0 := XREG[i.] = 0;
            1 := XREG[i.] = XREG[k.] srlr (not BREG[j.]<10:0>)
          end
        end
        #23 := DECODE BREG[j.]<17> => ! AX1
        begin
          0 := DECODE BREG[j.]<10:0> eq1 '00000 =>
          begin
            0 := XREG[i.] = 0;
            1 := XREG[i.] = XREG[k.] srlr BREG[j.]
          end
          1 := XREG[i.] = XREG[k.] sllr (not BREG[j.]<5:0>)
        end
        #24 := not.described; ! MX1
        #25 := not.described; ! ZX1
        #26 := begin
          XREG[i.] <= XREG[k.]<59> @ XREG[k.]<47:0>;
          BREG[j.] <= #2000 - (us) XREG[k.]<58:48>
        end;
        #27 := begin
          XREG[i.]<47:0> = XREG[k.]<47:0>;
          XREG[i.]<59> = XREG[k.]<59>;
          DECODE XREG[k.]<59> =>
          begin
            0 := XREG[i.]<58:48> = not BREG[j.]<10>
              @ BREG[j.]<9:0>;
            1 := XREG[i.]<58:48> = BREG[j.]<10>
              @ not BREG[j.]<9:0>
          end
        end
        #43 := begin
          XREG[i.] = 0 next
          XREG[i.]<59> = (jk neq 0) next
          XREG[i.] = XREG[i.] srlr (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; ! FX1 -> (Xj + Xk)
        #31 := not.described; ! FX1 -> (Xj - Xk)
        #32 := not.described; ! DX1 -> (Xj + Xk)
        #33 := not.described; ! DX1 -> (Xj - Xk)
        #34 := not.described; ! RX1 -> (Xj + Xk)
        #35 := not.described; ! RX1 -> (Xj - Xk)
      end next
      dealloc(3)
    end
  end.

  **Long.Add.Unit**
  LONG.ADD.UNIT(i<29:0>){process: critical} :=
  begin
    **Long.Add.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>;

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

    ladd(main) :=
    begin
      DECODE fm =>
      begin
        #36 := XREG[i.] = XREG[j.] + XREG[k.]; ! IX1 -> Xj + Xk
        #37 := XREG[i.] = XREG[j.] - XREG[k.]; ! IX1 -> 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.Declarations**

```

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.

**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(5)
  end.

  **Divide.Unit**
  DIVIDE.UNIT.1(i<29:0>){process: critical} :=
  begin
    **Divide.Declarations**
    fm <5:0> := i<29:24>.
    m. <2:0> := i<28:24>.
    i. <2:0> := i<23:21>.
    j. <2:0> := i<20:18>.
    k. <2:0> := i<17:16>.
    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
      end next
      xcmt = 0;
      XREG[i.] = 0 next
      CXi :=
      begin
        XREG[i.] = XREG[i.] + {us} XREG[k.]<0> next
        XREG[k.] = XREG[k.] srr 1;
        xcmt = xcmt + 1 next
        If xcmt lss {us} 80 => RESTART CXi.
      end
    end next
    dealloc(7)
  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<28:24>.
    i. <2:0> := i<23:21>.
    j. <2:0> := i<20:18>.
    k. <2:0> := i<17:16>.
    k1<17:0> := i<17:0>.

    **Increment.0.Execution**{oc}
    incr0(main) :=
    begin
      DECODE fm =>
      begin
        #50 := SA1                ! Increment
      end next
      #50:#57 := SA1
      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.,BREG[j.]) + BREG[k.].
        #6 := aref(i.,AREG[j.]) - BREG[k.].
        #7 := aref(i.,BREG[j.]) - BREG[k.].
      end
    end.

    #50:#57 := SB1
    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.] = BREG[j.] + BREG[k.].
      #6 := BREG[i.] = BREG[j.] - BREG[k.].
      #7 := BREG[i.] = BREG[j.] - BREG[k.].
    end
  end.

  #70:#77 := SX1
  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.] <= BREG[j.] + BREG[k.].
    #6 := XREG[i.] <= BREG[j.] - BREG[k.].
    #7 := XREG[i.] <= BREG[j.] - BREG[k.].
  end
end.

end next
dealloc(8)
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<28:24>.
  i. <2:0> := i<23:21>.
  j. <2:0> := i<20:18>.
  k. <2:0> := i<17:16>.
  k1<17:0> := i<17:0>.

  **Increment.1.Execution**{oc}
  incr1(main) :=
  begin
    DECODE fm =>
    begin
      #50:#57 := SA1                ! Increment
    end next
    #50:#57 := SA1
    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.,BREG[j.]) + BREG[k.].
      #6 := aref(i.,AREG[j.]) - BREG[k.].
      #7 := aref(i.,BREG[j.]) - BREG[k.].
    end
  end.

  #50:#57 := SB1
  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.] = BREG[j.] + BREG[k.].
    #6 := BREG[i.] = BREG[j.] - BREG[k.].
    #7 := BREG[i.] = BREG[j.] - BREG[k.].
  end
end.

  #70:#77 := SX1
  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.] <= BREG[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 [PC8600.isp].

! Peripheral Processor Description

end ! End CDC 6600