CSL Notebook Entry

To: CSL Notebook

Date: February 3, 1983

From: Anthony West

Location: PARC/CSL

Subject: Extended Dandelion CP - Description

File: [Ivy]<CSL-Notebook>81CSLN-YYYY

XEROX

Attributes: Cedar, Dandelion CP, writable control store, DES, cryptography, SDD, ED, debugging

Abstract: This note discusses the extended Dandelion Central Processor, CPE, from two viewpoints. First, it describes the way in which the CPE differs from the standard CP. Second, it gives hints as to how to debug CPE boards.

Overview

The Extended Dandelion CP, sometimes known as the 16K Control Store CP, is a Dandelion circuit board designed by Bob Garner, and myself with much help from others. The board *replaces* a standard CP board in a Dandelion, and emulates the Standard CP's functions faithfully. In fact, it will run all of the standard microcode and diagnostics without any change.

However, in addition to the standard instruction set, the Extended CP offers a number of additional instructions by which the microprogrammer may access:

1. Extended Control Store - up to 16K words
2. DES Hardware - for high-speed cryptography
3. Parallel Interface Port (planned) - for Stable Storage, etc.

Rationale

The board owes its existence to CSL's desire to run Cedar on Dandelions. As compared to Mesa/Pilot, Cedar imposes a number of additional requirements on the underlying hardware for efficient operation. In particular, for efficiency, Cedar requires:

- Microcode to add IEEE Floating Point instructions to the emulator
- Microcode support for Garbage Collection and Storage Management

The CPE board also provides:

- Hardware support for Cryptography
- Support for an External Parallel Interface port (planned).

Without spending much time on each of these, let us consider them briefly:

IEEE Floating Point

Much of the power of Cedar derives from Cedar Graphics, which makes very extensive use of real arithmetic to prepare images for the screen, or for other devices. In fact, the whole notion of resolution-independence of output or display devices assumes floating point operations (I think).

The standard SDD/Mesa emulator for the Dandelion does not include these instructions (since Star Graphics is, to my knowledge, based on fixed-point operations), and floating-point operations are trapped out to a Mesa implementation.

The standard machine offers 4K words (of 48 bits) of control store. This memory holds the emulator code as well as all of the device tasking code and the fault handlers, etc. The standard emulator has already been "shoehorned" to fit in 4K -- there is nothing else which can be expelled from the control store to admit more code. The solution chosen by CSL was to build a board with a larger control store.

Howard Sturgis (Sturgis.PA) has now finished the first stage of the implementation of the IEEE Floating Point code, and an early version of Cedar is in operation.

Garbage Collection & Storage Management

Another area in which Cedar differs from Mesa is that it offers to programmers extensive assistance with dynamic storage management. In particular, by using reference-counting and garbage-collection techniques, data objects allocated dynamically within procedures are automatically reclaimed by the system when they are no longer needed.

To implement these facilities such that they perform adequately requires some assistance in microcode. Fortunately, the exact functionality of this microcode was determined by a prior microcode implementation on the Dorado. Again, the desire to add more microcode implies a processor with a larger control store is needed.

DES Cryptography Hardware

Most of our efforts in communication protocols and network architecture have concentrated on network system management and providing general internetwork function. Relatively little thought has yet been devoted to the subject of security in our distributed systems. There are two basic kinds of security we are concerned with:

> Support for *secure data transfer.* The distributed system should provide assistance to protect data being transferred between machines from various threats such as unauthorized copying, insertion of data, masquerading, etc.
>
> Support for *secure data storage.* The distributed system provides assistance to permit data to be stored on file servers, etc. in such a way that it is safe against unauthorized copying, etc.

Part of Cedar involves the implementation of secure communication protocols designed to assist with these two requirements, and these protocols assume the existence of hardware support for cryptography. Today, the dominant encryption technology is based on the Federal Data Encryption Standard, DES, proposed by the National Bureau of Standards, and for which LSI devices are available. A characteristic of DES is that an implementation in software, or even microcode, is unacceptably slow.

The extended CP carries optional DES cryptography hardware, accessed by extensions to the microinstruction set. Although some microcode is needed to interface the hardware to the Pilot head, the amount of microcode is much smaller than the other two functions above. Cryptography, then, did not argue *per se* for much more control store -- the rationale was that if CSL was going to engineer a new CP, then advantage should be taken of the opportunity to add DES hardware.

External Parallel Interface Port

Another function Cedar would like to assume of the base hardware of the machines it runs on is the ability to access External Electronic Stable-Storage modules. The reasons for this are discussed in a companion note. Here we had two choices:

Design a *special-purpose* external stable-storage interface

Design a *general-purpose* parallel interface, useable for external stable-storage interfacing, but with much wider applicability.

One criticism of the Dandelion compared with our other machines is that it has lacked a general-purpose parallel interface port, like the printer port on the Alto. Versatec, for one, could have benefited considerably if the machine had offered the ability to interface simple devices easily.

The extended CP will *probably* offer such a general-purpose port accessed by extensions to the microinstruction set. However, unlike the other hardware mentioned above, this has not been implemented yet, and the choices are still being discussed (Dave Boggs).

Extended Control Store - Theory of Operation

The CPE supports up to 16K of control store. The storage array is implemented with 4K by 4 parts instead of 16K by 1 so that the control store can be populated in 4K increments, thus saving money. (The chips are expensive).

The Rams used are Inmos 1420-55, which feature a low-power standby mode so that those 4K banks not in use do not consume much power. The timing is such that IMS 1420-70nS parts would proably also work, and these cost about $12 per chip instead of the $19 which the 1420-55 cost.

The extended control store is addressed by loading a Bank Register. This approach was chosen in preference to expanding the width of the Next-Address fields in the microinstruction because of the existing software investment which would have to be converted, and the extensive hardware changes which this would have necessitated. *Surely* next time sombody designs a machine, they will allow room for control store expansion beyond anyone's wildest dreams!

There is, however, only one bank register, which can only be used by the emulator. The practical result of this is that all of the IO microcode must still reside in Bank 0. When the SwitchProm notices that an IO task is going to run next click, the bank set in the bank register is ignored and bank 0 is coerced by the hardware.

No special action is taken to coerce to bank 0 upon *traps and faults*, however, so trap/fault handler microcode must be replicated in all of the banks you use in the machine.

In practice, this works out OK, since the IO microcode is relatively small, and the emulator is the part of the microcode for which the extended control store is really necessary.

There is one small quirk, however, which is that the IOP cannot load the bank register directly. Thus, loading microcode into the additional banks at boot time is an interesting two-step process. When booting, the IOP first loads up the CP Kernel microcode, which executes as task 7. The Bank register is ignored, and all Control Store data is fetched from bank 0. The Kernel can execute an instruction to load the bank register with the desired bank value without disturbing its own operation.

The IOP then grabs the CP by the throat, and uses task 6's TPC registers to write the control store. When the ScheduleProm sees task 6 is "running", it enables the bank register, so the data written by

the IOP is written into the appropriate bank. Upon completion, the IOP releases the CP, which resumes execution of the Kernel as task 7 in bank 0. The kernel then switches the bank register to the next bank and so on.

At this time, the only way of loading the extended control store with microcode is by using a second Dandelion running the Burdock debugging tool, which supplies the CP Kernel with little overlay programs to do the bank-switching function. Eventually, the IOP Domino microcode will be amended to know about this sequence of events and a new disk/network loader format, thus allowing a machine to boot microcode into its own extended memory, *but this hasn't been done yet.* (Hal Murray & SDD).

So far, we have been using a CPE with all the IO microcode in bank 0, and an extended emulator containing IEEE Floating Point instructions in Bank 1. The emulator did not have to be altered to know about the bank-switching logic, since, once started in bank 1, it never has to change the bank register. Once additional microcode looks like overflowing bank 1, however, (eg. Cedar Garbage collector, etc), then the emulator will have to jump between banks. The best model at this time is not to dispatch instructions across banks, but rather to store extended sequences of operations in the other banks and to switch relatively infrequently, eg. BitBlt, or Garbage Collect.

DES Encryption - Theory of Operation

The idea of the Des option is that the hardware is provided to allow the implementation of a DesBlt operation which runs as an emulator instruction. This is in contrast to the Dorado implementation, where an IO task is used.

The microcode initialises the DES chip, picks up words of plaintext from memory, stuffs them through the DES chip, and stores them back into memory again. Thus, multiple encryptions of the same data can be done sequentially using different keys, and DES can be used both for Ethernet applications, as well as for encrypting data to be stored on disk or diskette.

Detailed Differences between Extended and Standard CP

This section gives you a guided tour of the logic diagrams of the CPE and identifies the places where the design differs from that of the standard CP. Note that design differs relatively little from the standard design, so little effort is devoted to discussing the basic principles of operation of the CP. There is an excellent hardware reference manual for the standard CP (see below for filenames, etc.). Let us scan the drawings page-by-page.

Design Automation Facilities

This design was done using ED/SDD's design automation facilities, so that, as far as possible, the design would be compatible with the rest of the Dandelion. In addition, mostly Xerox/ED qualified components were used, thus making it easier for ED to take over the manufacture of the board some day. You shouild realize, however, that ED's versions of Sil, Analyze, and Route differ from those used in PARC in the following very important ways:

1. You can define font 4 macros in Sil *which will not be expanded by Analyze.* This means you can define macros for components which are private to the page they occur on. This is how I have defined the special variants of the S240 and S241, as well as the IMS 1420 Rams, and the AmZ8068 DES chip. *Only macros 0-9 fall into this class* -- others will be expanded as in CSL's Analyze.

2. ED's version of Route is a snapshot of CSL's Route as it was about 18 months ago. This means that many new features added by Ed McCreight since, such as explicit groups of nets (foo!1, foo!2... foo!50) do not work anything like as well, and it is not possible to produce a PC form of the wirelist which any of the standard PCB companies can accept. I made an attempt to cause Inouye.ES, which controls ED's Route, to re-adopt CSL's latest Route, but this has not been done yet.

3. There is one important addition which ED have put into their Route which CSL's Route does not have. This is the ability to define the pin row/column table by filling out a template form and then "compiling" that into a compressed form of the data structure. This can then be pre-loaded with a canonical board characterisation using ED Route's /i switch. *I use this feature in this design* -- I used ED's board description facilities to describe the high-density board I used. Further documentation is available on this facility from Inouye.ES, and the relevent files I used are listed at the end.

sCPE00

You should use Alto DF file software if you want to do anything involving the design. Either type

```
Bringover /a [Indigo]<Dandelion>CPE>DfFiles>sCPE-W.df
```

to retrieve all the ED/SDD Design Automation software together with the drawings, etc, or type

```
Bringover /a [Indigo]<Dandelion>CPE>DfFiles>Proms.df
```

to retrieve all the files to work on the proms on the board.

Very important: If you change anything, make sure you flush the updates out back to Indigo using SModel *or you will really regret it.*

sCPE01, sCPE02

The board used for the design is a high-density Dandelion stitchweld board made by Interconnection Technology, part number 596P55884A. ITI cunningly make two different boards with this same part number, *so you have to specify which you want.*

Type 1 boards have many decoupling capacitors soldered in the board at the expense of stitchweld pins in those locations. Type 2 boards are filled with pins, and very few capacitors. The type 2 boards are the ones you want for this and all future Dandelion hardware work.

Notice that there are 18 spare IC locations, should you wish to add something else. Some of these may be consumed by the parallel interface option.

As described above, I used ED's board description tools (BuildRb.Run) to create a new template describing the pin configuration of the board. I then also amended ED's canonical board characterisation to match. Using both of these things, the ED Route's /i switch pre-loads the template before routing the board.

The reason that rows 20 to 24 are interspersed with rows 1 to 19 is historical -- the idea was to be able to take a design layed out for the original low-density board, and be able to build it on the high-density board and have it end up looking similar. Note that placing 40-pin chips requires using a different row-numbering scheme. The 5 40-pin chips are placed in e32, g32, e33, g33, and c33.

sCPE03

First, some information about the Proms. The proms used on the CPE are mostly identical to those used on the standard CP. There are some exceptions:

1. The SwitchProm used is Revision R -- it contains additional logic for the Bank select feature, not present in the standard CP.

2. The Des option requires 3 proms, two for the control of the Master (input) port, and one for the control of the Slave (output) port. These don't exist at all in the standard CP.

Second, the lower half of the page shows the font4 macros used in the rest of the drawings. As noted above, only ED's version of Analyze will not expand these macros -- *CSL's Analyze won't work with this design.* There is an additional Analyze dictionary containing the descriptions of these components -- this should be prepended to the standard SDD/ED dictionary chain.

sCPE04

This design uses the newer 2901C ALU's instead of the slower 2901B-1's used in the standard design. This gives us more timing margin for little increase in cost. Since we have added more loads on various busses, this extra margin is welcome. 2901B-1's would probably work, however.

sCPE05 - same as standard CP

sCPE06

Early stitchweld versions of the standard CP didn't have the S241's on the outputs of the F93422 SU registers. It was found that the F93422 devices did not have enough oomph to drive the X-bus sufficiently crisply, so the buffers were added when the standard board went to etch. I have added them too, and, as in the standard board, this means that the F93422's can be permanently enabled.

sCPE07 to sCPE11 - same as standard CP

sCPE12

This page contains several differences from the standard CP:

1. An additional S138 (b23) has been added to provide the additional fZNorm decoding needed by the DES option. There are two further decodes left over, but I expect these to be used by the parallel interface option.

2. An additional signal, Bank←', has been added to the fYNorm-High decoder. This is the signal which causes the control-store bank register to be loaded on the next clock. Note that the Dandelion hardware manual is incorrect -- Bank← is fY=0D, and not an fZ as stated. The signal replaces a signal ClrIOPReq', which is on the standard CP but is not used.

sCPE13 to sCPE16 - same as standard CP

sCPE17

On this page is the revised logic for the SwitchProm. An additional signal has been added, IOBank. The SwitchProm contains the logic which, looking at the task which is currently running (Ct) and the task which will run next (Nt), decides whether there will be a task switch or not.

There is only one Bank Register, and only the emulator, task 0, can use it. The purpose of IOBank, therefore, is to force all Control Store addressing associated with IO tasks to come from bank 0. This coercion is done on the top of page 27 -- see below.

One useful hint: The meaning of the Nt, Ct, and Pt, lines changes depending on which cycle of a click you are looking at them in. The table at the top of the page tells you what these signals mean for c1, c2, and c3. As you can see, the most useful signals to look at with a logic analyzer are Nt, since, in c1, Nt tells you which task will run next click, in c2, which task ran last click, and, in c3, which task is running this click.

sCPE18 - same as standard CP

sCPE19

I just reorganised the page. The only point worth making is that I went to some trouble to receive the clock signals right near the edge-connector they come in on, and then fanned them out to the middle of the board for redistribution, as we did for typical Dorado boards.

sCPE20

The logic on this page is completely new. Data coming (top) from the IOP is buffered near the edge-connector, and distributed to two basic places:

1. To the LS240 buffers. The control store in the CPE is implemented with 4K by 4 static Rams where Data Input and Data Output share bidirectional I/O pins. In the standard design, the 4K by 1 Rams have separate Input and Output pins. This means that I have to tristate the IOP data coming in when the CPE is running, and enable the LS240's when I want to write the control store. The control store is loaded one byte at a time, depending on which of the 6 CSWE signals are activated.

2. To the LS374 and LS244, which supply both Nt and NIAX to the TPC registers on page 16. Look back to page 16 for a second. The TPC and TC registers hold the saved next-instruction address and task condition codes when one task is suspended at the end of a click and another resumed. You see that, by setting NIAX appropriately, the IOP can write data into the TPC registers at the address given by Nt when the signal SwTAddr disables the S374. This means that the IOP can seize control of the TPC Registers, and this is how it writes the control store. The IOP stops the CPU, sets Nt to be task 6 (not used by anythong else), and stores a control store address in the TPC registers. This address percolates through the 25S09 Mux/Latch on page 27 and addresses the control store. The IOP then supplies the data for each byte of the control store, one at a time, and enables the appropriate CSWE signal. Reading works similarly, except that the data read back is sent back to the IOP via the 8:1 MUX on page 26.

What this all means is that, unless the IOP can read and write the TPC registers, it cannot load the control store, so the CP is dead. In debugging, get the path from the IOP, through the TPC registers, and back to the IOP through the IOP Mux working first.

sCPE21 to sCPE24

These pages show the new control store. The 4K by 4 Inmos Ram was chosed over a 16K by 1 part to permit the control store to be populated incrementally in 4K chunks, as required. When making these boards, the first 4K should probably be soldered in, the next 4K may or may not be socketed, depending on whether SDD want to use this board to replace standard boards in the field, and the other 8K should be socketed to allow for incremental expansion. The chips cost between $13 and $20, so this is a way of cutting the cost.

sCPE25

This page is essentially unchanged compared to the standard CP. The only difference is that I added an extra S374 to latch the output of the parity checkers which is situated near the CSIntProm. This could possibly be an LS374. CSParErr is simply the XOR of CSPar[0..5].

sCPE26

In order to provide a path for the IOP to read back the bank register, the path provided for the IOP to read back the TC bits, TC[0..3] has been disconnected. This path was never used, neither by the standard microcode, not by the diagnostics. Therefore, the operation of the software is unaffected.

All four bank register bits can be read back by the IOP.

sCPE26

The lower half of this page is the same as the standard CP, the upper half is new.

The CPE has a single 4-bit Bank register which can be loaded from the Y-bus when the Bank←' signal is activated (see page sCPE12). An LS161 counter is used to hold the bank number since this is the only part I could find which has a suitable asynchronous reset, useful for coercing the bank back to 0 when the IOP is resetting the CPE. The implication of this is that, when the IOP boots the CPE, unless the microcode loaded uses Bank← instructions, or, for some new reason, the IOP is required to read the TC bits, the machine cannot be distinguished from a standard CP. *It will run all of the standard microcode and diagnostics without any change.*

The bottom two bits of the Bank number are decoded into 4 bank select signals which are supplied to the 25S09. If IOBank is not active, the value of the bank number in the bank register is used to address the corresponding quadrant of the control store *on the next clock - the 25S09 contains a register.* Otherwise, if IOBank *is* active, the bank selected will be bank 0.

sCPE27

This page is essentially the same as in the standard CP. The addresses from the 25S09's on the previous page pass through some series-damping resistors before being fanned out to the control store Ram array. I have found that 15 ohm series resistors reduce over/undershoot to an acceptable level on the stitchweld board, but the values should be reconsidered when converting to etch. Replace one of the resistors by a variable resistor, put the CPE in a tight loop, and, with the scope on the corresponding CSA line, adjust the resistor between 10 and 30 Ohms until a suitable compromise between reduced over/undershoot and increased rise/fall times is reached.

Theory is fine -- you can probably calculate all this using transmission line theory, but why waste the time. Measure it.

You can install a Beckmann 1K ResNet Dip chip, or discrete resistors for the pullups, as you wish.

sCPE29

The standard CP has a 16-pin socket where you can plug in a logic analyzer and debug the software. I decided that I would like an external connecter, so that you can do this without having to yank the board out or stop the machine.

I chose a 37-way Cannon D-Type female, so that not only the CSA lines, but also many other interesting signals could be viewed at the same time. The Garage, or whoever makes these boards, should make up a number of cables which connect a logic analyzer into this connecter in various interesting ways, so that debugging is simplified. The connector carries all the interesting signals to permit a history buffer and debugging logic as used in the Dicentra to be added.

sCPE30

This page contains all the spare gates, etc. Since I have tons of room left on the board, I did not try too hard to reduce the design.

sCPE31 to sCPE36 - DES Option

These pages contain the DES logic. On page 31, data arriving on the Y-bus is latched in an LS374 before falling into the Master port of the DES chip. The Slave port output is latched by another S374 before being enabled onto the X-Bus whenever an instruction containing XBus←Des' is active.

Note that the upper byte of the X-Bus is set to 0, thus simplifying the speed-critical inner loop of the DES microcode, which has to reassemble the bytes into words again.

The DES chip runs on a 411nS clock -- ie. one DES clock per Dandelion click. The 374's are needed to permit the CP to write and read the DES registers in any cycle, and not only in, say, c2.

There are logically two different paths that data written to the DES chip might take:

1. Data written by the CP in c1 or c3 is written into the DES chip with MDS, the master port data strobe. This is the normal case.

2. Data written by the CP in c2 is written into the DES chip with MAS, the master port address strobe, and, depending on the values of MP1 and MP2 at the time, serves to select one of the various command or mode registers inside the chip.

Programming information: If you write microcode to write data to the DES chip in c1, it will be transferred into the DES chip in the immediately following c2, meaning that you can write a byte every click. If you write data in c2 or c3, it will only be transferred into the DES chip in the c2 of the *next* click, meaning that you can only write *every other* click. Similarly on reading data from the DES chip. If you read the data from the buffer in c1, you can read another byte in the next c1. If you read in c2 or c3, then you must let an idle click go by before reading another byte.

Note that there is no way to examine the slave port flag or the master port flag status indicators, so you have to write the microcode to conform with the chip's specifications.

Further, if the microcode ever gets out of sync, you are in trouble. Therefore, before doing anytning, *always* reset the DES chip with a software reset, and read a byte out of the output buffer to reset it to the empty state.

The master port of the chip is never read, and the slave port never written - which would correspond more closely with the way AMD intended the chip to be used.

On page 32, there is a summary of the DES chip data sheet. The device has two funny problems:

1. AMD parameter 44a shows that MDS and SDS have *maximum* active-low times. This means that when the CP is stopped by the Wait signal (either some clicks are skipped because the CP is trying to access the display memory whilst the screen is being refreshed, or the IOP has stopped the CP, possibly as a result of a breakpoint), these signals have to be in the inactive state, or data will be lost. It is for this reason that the DES clock's rising edge coincides with c2, which would be the first clock to be skipped when Wait goes active.

2. MDS and SDS have funny *hold times* relative to DesClk falling, see AMD parameter 45 and 46. This necessitates bringing down DesClk a little before these signals, which is done by the S00 in the reset path to the S74 clock generator at the bottom of the page.

On page 33, the two finite-state machines which control the master and slave ports are shown. They are both reset by IOPReset, which jams them both into state 0. IOPReset is synchronised with DesClk because the AMD spec requires it.

The Master Port FSM looks at the current state, which cycle we are in, and what the CP is doing (Des←YBus'). Depending on which cycle Des←YBus is activated in, either MAS or MDS will be activated in the next possible DesClk.

Similarly, whenever the Slave Port FSM sees that SFLG indicates that there is data to be read from the Slave Port, it reads the byte and loads it into the S374 for the CP to read at its leisure. Only after the CP has read this data will the next byte be read by the FSM out of the slave port.

If the CP tries to access the chip at the wrong times (Des←YBus is activated before the previous write has completed, or XBus←Des is activated when there is no data ready yet), then DesMpError or DesSpError are set. These signals are OR'ed together and appear on the external debugging connector for information. Once set, the signals remain set until the FSM's are reset by IOPReset.

The state diagrams for these two FSM's are shown on the next page, with the timing on page 35.

Page 36 shows some folklore about pipelining, but this is not known to be definitely correct.

Debugging the Extended CP

1. Turn the machine on.

2. Get out a scope, and check that all the clocks and critical cycle signals are OK. In particular, turn to page sCPE19 and scope the AlwaysClk's on e23-2, e23-4, and e23-6. Check also Cycle 1, 2, and 3, on b19-2, b19-4, b19-6.

3. Now be an optimist and go for bust -- try to run the standard Sunlight diagnostic. If this works, proceed to step 11 below.

4. Sunlight doesn't run, so load Burdock's CPKernel Tester. Start with trying to get the data paths from the IOP to and from the TPC registers to run. If these paths don't work, you can't test anything else. Additional buttons are provided which put the IOP in a loop reading or writing the TPC registers so that you can put a scope on the selects, strobes and data paths and make sure they are doing the right thing. Also, the tester will tell you what data it is sending and what comes back, so you can often deduce whether there is a data or address bit stuck, or whether the strobes work at all.

5. When the TPC registers test out OK, test bank 0 of the control store. All subsequent operations depend on Burdock being able to load little CP microcode test programs, so, again, you have to get past this hurdle first. Again, you are told what data is sent and returned, so you may be able to deduce what's happening.

6. You can now load microcode. Poke the FlapXY button (which loads a little test program) and look at all the X-bus and Y-bus bits to make sure that they have reasonable square-wayes on them. This means that the main CP data paths are clear.

7. Push the button to flap all the control store data bits (except Mem and EP), and make sure (with the scope) that all is well.

8. Poke the JumpNext button, which loads control store with microcode that jumps to the next location. When the IOP sets the CP off, it should rip around the control store at full speed. With a scope, make sure that successive CS address lines carry square waves which double in frequency from line to line..

9. Try Sunlight again. If it runs, skip to step 11 below.

10. Load the Kernel by pressing the Boot button on the CP Panel. Then test the U registers and the R registers. Once again, these have to work first before the rest will run.

11. You are close. Load the CS Bank Tester, and after Booting the kernel (see 10), try the bank switching logic. Make sure the IOP manages to write and read the bank register OK.

12. Test the extra control store banks.

13. You should now be able to get through Sunlight to Othello. Try booting Cedar and see what happens.

If any of these steps leave you stuck, look at the selects. If you are *really* stuck, Hal Murray has a special version of Sunlight which sends out a little more debugging information to Burdock. By looking at this information, you can generally intuit what Sunlight is trying to test, and often, what exotic thing is failing.

Location of Files

Logic Diagrams:	[Indigo]<Dandelion>CPE>Archive>sCPE-W.press
CP Reference Manual (3 parts):	[Indigo]<Dandelion>CPE>Docs>DLionManual1.press
	[Indigo]<Dandelion>CPE>Docs>DLionManual2.press
	[Indigo]<Dandelion>CPE>Docs>DLionManual3.press
This document:	[Indigo]<Dandelion>CPE>Docs>CPE.bravo and .press
Prom sources etc.:	[Indigo]<Dandelion>CPE>Proms>*
Past Laurel mail file messages:	[Indigo]<Dandelion>CPE>Mail>DLionCS.mail

// BankSwitching.doc, HGM, 27-Apr-83 18:17:32

CSL is building DLion CP boards with up to 16K of control store. This is a quick description of how to
** use the new features.

Aside from the extra RAM chips, the heard of the change a new 4 bit Bank register. It is used when fet
**ching microcode for the Emulator task and when the IOP is writing or reading control store via Task 6
**. IO tasks must all fit into the first bank.

The Bank register is loaded from the Ybus with the Bank← function. Bank← is FYNorm 0D. Beware the DLi
**on hardware manual lists it as FZ 04. The change allows FZ to be used to specify a constant.

Mass doesn't yet provide much help for programmers who need to switch banks. You have to assemble each
** bank of microcode separately, and specify the transfer locations by hand.

If you want to get to another bank, you need to write code like the following:

In Mumble.df:
```
        Set[LandingSpot, nnnn];
```

In the microcode used to build the first bank:
```
        xxx, Bank ← n,                  c1;
        xxx,                            c2;
        xxx, GOTOABS[LandingSpot],      c3;
```

In the microcode used to build the other bank:
```
        xxx,                            c1, at [LandingSpot];
```

Yes, there is an extra stage of pipeline delay beyond what you would expect. (It's not in the Dicentra
**.)

The Bank← needs to be in c1 to avoid confusion in case an IO task runs in the middle of the bank switch
**ing sequence. Actually, you can switch banks on any different cycle if you will are willing to repli
**cate a few instructions in both banks and force both copys to have the same location.

Currently, the only way to get code into the second bank is with a DLion Burdock command file. Try som
**ething like:

```
Boot[];
SetBank[0];
Load[IOStuff];
SetBank[1];
LoadMore[Emulator];
....
```

Internally, Burdock uses 16bit CS addresses. Since Mass only generates 12 bit addresses, Burdock puts
**the bank number saved by the Setbank command into the appropiate high order bits when loading microco
**de and inserting symbols into it's symbol table. When you Start (or store into a TPC), Burdock will
**automatically switche banks if necessary.

Remember, only the emulator can run outside bank 0.

// DESChip.doc, HGM, 27-Apr-83 20:24:01

The 16K control store CP board also contains a DES chip. It's an AMD and/or Zilog 8068. (AMD also cal
**ls it a 9518.)

The DES chip is clocked once per click so clicks and clocks are interchangeable in the following discus
**sion.

DES works with 56 bit keys and data blocks of 8 (8 bit) bytes. The 8th bit in each key byte is used fo
**r parity. The encryption algorithm is used in 2 modes. ECB (Electronic Code Book) takes 8 data byte
**s and 56 key bits, scrambles things, and produces 8 result bytes. CBC (Cypher Block Chaining) uses t
**he same basic scrambling algorithim but it also includes a 64 bit vector that is XORed with the data
**before encryption. After encrypting each block, the encrypted data is saved for use as the next 64
**bit vector. (You do things in a slightly differnet order when decrypting.)

To encrypt things, you set the desired mode, load the key, and then (with the left hand) start feeding
**data into the chip and, after a slight delay, (with the right hand) start extracting the encrypted da
**ta. It takes two hands because there is a long pipeline in the chip, and you must overlap things to
**keep the chip busy.

If you keep the pipeline full, the chip will encrypt an 8 byte block in 18 clocks. That works out to 8
**.6 megabits/second. Ignoring the startup problem, if you want to keep the chip busy you have 18 clic
**ks to read 4 words from memory, store 8 bytes into the chip, extract 8 bytes from the chip, and write
** 4 words to memory. That's not trivial, you obviously can't do things in that order, but it's not ve
**ry hard. (You will probably miss a beat when crossing page boundries and/or when IO tasks steal cycl
**es.)

We use the chip in the "Dual-Port Configuration, Multiplexed Control". There is no way to insert a mas
**ter key through the auxiliary port.

The specs don't quite describe the way we actually use the chip. They expected it to be used to interf
**ace to a disk or tape drive. That way all the encrypted data would be on the slave port, and you wou
**ld encrypt by storing clear text into the master port and extracting encrypted data from the slave po
**rt. Decryption would reverse the direction of data flow and store encrypted data into the slave port
** and read the clear text from teh master port. We do it differently. On the DLion, you always store
** into the master port, and read from the slave port. Encryption/deryption is selected by using the u
**sing appropiate mode bits.

The DES chip adds 3 function decodes to a DLion:

DESCtl ← is FZ 09. It must be issued in c2. DESCtl does an address strobe cycle to the chip using
**the data from the Y bus as the address. This determines where bytes stored into the master port will
** go. 0 is the data register, 2 is the command register, and 6 is the mode register.

DESMP ← (DES Master Port) is also FZ 09. It stores a byte of data from the Y bus into the master por
**t. It is valid in c1 or c3. If it is used in c3, it must not be used in the following click.

← DESSP (DES Slave Port) is FZNorm 0A. (Beware, Mass can't yet process reading things with FZNorm de
**codes, so the existing boards have blue wired ←DESSP to be the same as TIData.) It reads the a byte
**from the slave port onto the X bus. The high byte is cleared. It can be given in any cycle, but if
**it is used in c2 or c3, it must not be used in the following click.

There is no way to read any status bits. In particular you can tell if the chip gets a key parity err
**or.

Interesting Mode register bits:
- 00H decrypts in ECB mode.
- 02H decrypts in CBC mode.
- 14H encrypts in ECB mode.
- 16H encrypts in CBC mode.

Interesting Command bytes:
- 00H resets the chip.
- 11H loads the encryption key.
- 12H loads the decryption key.
- 84H loads the initial vector for CBC decryption.
- 85H loads the initial vector for CBC encryption.
- C0H starts crunching bytes, encrypting or decrypting as setup by the mode.

After setting the mode and/or storing a command, you need to wait 6 clocks.

After writing the 8th byte of an input block, you have to wait 6 clocks before storing any more data.
**Similarly after reading the 8th byte of an output block, you also have to wait 8 clocks before readin
**g the first byte of the next block.

The programmer is responsible for meeting the timing constraints. The idea it to count cycles. There **isn't time enough to read a status bit. Besides, there isn't any hardware to read them.

Unfortunately, there is no clean way to reset the DES logic. It's possible to get a byte trapped in th **e output buffer. The initialization code should read a byte (even if there isn't one available) and **discard it.

The state machines used to control the input and output to/from the DES chip have an extra bit that may ** be helpful for debugging microcode. It comes on when a byte is read/written before the hardware has ** emptied/refilled the buffer. (Unfortunately, the initialization sequence described above will norma **lly set this bit.)

Working microcode, Mesa test programs, and a variant of Mass that knows about the new function decodes **are stored on [Idun]<Murray>DES>.

// LoadingExtrabanks.doc, HGM, 27-Apr-83 18:00:11

The Bank← function is FYNorm 0D. (Beware the DLion hardware manual lists it as FZ 04. The change allo
**ws FZ to specify a constant.) The Bank register is loaded from the Y bus.

Mass doesn't yet provide much help for programmers who need to get to another bank. You have to assemb
**le each bank of microcode separately, and specify the transfer locations by hand.

If you want to get to another bank, you need to write code like the following:

```
In Mumble.df:
        Set[LandingSpot, nnnn];

In the microcode used to build the first bank:
        xxx, Bank ← n,                  c1;
        xxx,                            c2;
        xxx, GOTOABS[LandingSpot],      c3;

In the microcode used to build the other bank:
        xxx,                            c1, at [LandingSpot];
```

Yes, there is an extra stage of pipeline delay beyond what you would expect. (It's not in the Dicentra
**.)

The Bank← needs to be in c1 to avoid confusion in case an IO task runs while bank switching. (Actually
**, you could switch banks on a different cycle if you will are willing to replicate a few instructions
** in both banks and force them to have the same locations.)

Currently, the only way to get code into the second bank is with a DLion Burdock command file. Try som
**ething like:

```
Boot[];
SetBank[0];
Load[IOStuff];
SetBank[1];
LoadMore[Emulator];
....
```

Internally, Burdock uses 16bit CS addresses. Since Mass only generates 12 bit addresses, Burdock puts
**the bank number saved by the Setbank command into the appropiate high order bits when loading microco
**de and inserting symbols into it's symbol table. When you Start (or store into a TPC), Burdock will
**automatically switche banks if necessary.

Remember, only the emulator can run outside bank 0.