Building a Microcomputer with Associative Virtual Memory

W P Cockshott

Glasgow University Computer Science Dept

ABST RAeT

This report describes the motivation for and design o(the poppy *wm*-puter designed to support persistent programming languages. The com ru ter uses a novel virtual memory architecture to support object addressing.

November 14, 1985

Building a Microcomputer with Associative Virtual Memory

W P Cockshott

Glasgow University Computer Science Dept

1. Persistent Programming

Conventional programming languages like Basic, Pascal or C present the programmer with two broadly different views of memory: program vanables and files. The distinction between them arose because a computer generally has two different types of physical store: RAM and Disks.

Variables are used to store the working data of a program and are held in RAM, whereas files are held on Disk and are used to transmit information from one program to another. As most programmers know to their cost, files are a lot more tedious to program with than variables. It is not just that they are slower than variables, they are also a lot less flexible.

In a language like Pascal, your variables are Typed and you can create new types of data to handle new applications. For graphics work. you can define types for PICTURES, WINDOWS, COLOURS etc, whose properties are optimised to suit the algorithms that they are going to be used in. Your variables can also be structured using type constructors such as arrays, sets, records and pointers. With these facilities you can readily create arbitrarily complex data structures. The freedom that this gives the programmer is even greater in those languages that allow the dynamic creation of data on a heap. In addition, a modern programming language will provide its variables with facilities for information hiding by means of modules and scope rules.

Contrasting this with what we get from files we see that they come a very poor second. They are untyped for a start. When you open a tile you do not know what its internal structure is going to be. It may have been written as a file of records, but you can open it as a tile of characters, so any type rules go out of the window.

Even worse, not all types of data in files. It is easy to build up a binary tree or a linked list on the Pascal heap. If this is then written out to a file of records, then all of the pointer structures get destroyed. If you read the records back in, there is no way of linking them back together again.

To cap it all, we find that files all appear to be "global". A tiJe crcated by one module of a program can be read in any other module. making it very difficult to implement information hiding between modules.

With all these disadvantages why do we put up with files?

It aeema that there are three reasons. The first is just historical inertia. The dUtinction between files and variables has been around so long that it is part of the established paradigm of computing. It requires a conceptuAl revolution to break away from it. Behind this there are ecOnomic and technical reasons. Ram chips are volatile. They do not ItOI'e information in the absence of a power supply, so that disk files tend to be used for information that has to persist over long pcri<x1,In acTc1ition they are more ensive than disks. Small portable compllters do \lie battery backed C OS Ram chips for persistent storage. but for reMOnI the capacity on these is very small. However these WIre problema are by no means insuperable.

1.1. Vil"tul memory

Hardware designers were quick to see the possibilities inherent in rotatinJ etic stor~ devices. Virtual memory usin$ paging technies Was ldUeved by t Manchester University Atlas computer in I 60. ~ allOWI rotating storage devices to be seen as an extension to the on Neumann) random access store.

In the quarter century since that basic advance, software designers have mac1eIUpprjaing~ little use of the potentialities of virtual memory. It ha been eeen m ly as a way of being able to run biger programs. An area of dilk islet aside for page swapping, but this Inherits both the advanta:~ and the disadvantages of conventional RAM. It is viewed as both random access,and volatile. Persistent data is still kept in files.

1.2. Store mapping

Some of the relatively early virtual memory operating systems. like Multicl and the lellgetknown Edinburgh Multi Access System provided an advance, in that the~ allowed files to be mapped into the random accell virtual memory. ersistent data could then be accessed as if it were an array by the normal operations of a proamming language. However, this approach was only supported in certain systems programming languages and never came into wide use. The operating systems still maintained two erate types of store, governed by different conventions. There was t e file store which was public, hierarchically organised and addressed associatively via long symbolic file names. Then there was the virtual memory which was essentially private and organised as an array. One area of store could be partially mapped onto the other, but that presupposed that they were seen as different in the first place.

1.3. PS-algol

From the late '70s there have been breakthroughs in programmin lange design that enable us at last to get away from the 013 file variable paradigm. This has come through the incorporation of the idea of persistence into a number of experimental programming

languages. The first of these were Smalltalk and PS-al§ol. former developed at Xerox PARC and the latter at Edinburgh and t-Andrews universisties. Subse1uentlY the idea has been adopted in a number of other languages: B rom the Mathmatical Centre in Amsterdam. Poly from Cambridge University, and Amber from Bell Labs.

These langes allow variables of any type to persist and have no notion of es in the usual sense.

PS-algol for example supports a persistent heap. Data of any type inc1udin~ procedures can be put on the heap, which will then persist beyond t e time of execution of the program that created the data. The im;;rtant thing that distinguishes a persistent heap is that pointers or re erences to ol:)jects can be made tocr;rsist, so that a linked list or tree used by one .l?program can be accesse weeks later by another. The PS-algol system Itself hides the existence of different store models in the underlying operating system but only at the C05t of considerable software complexity and associated performance overheads. The various teams implementing Smalltalk have also found that although a persistent heap provides a superb programming environment. the complex software needed to sustain it means that you need very fast proces..<;Qrs to get an acceptable performance.

1.4. Virtual Memory Micros

Over the last couple of years however, it has become possible to buy microprocessors with virtual memory. It should thus be possible to build a relatively cheap personal computer with hardware support for persistent programming. The rest of this article describes the Poppy. a single board computer with a virtual memory system optimised for persistent programming languages.

What are the requirements of persistent programming that have made it relatively difficult to support with purely software techniques?

Per<>gramming systems, whether PS-algol or Smalltalk. are object . Iiistead of viewing store as a uniform array of bytes. 1hey view it as a collection of obets on a heap. Each object is addressed viii a unique Persistent Identifier PlO). A PlO is valid not just for the ru n of one pr0w,am like anormal store address, but for an indefinite perir>d. Objects Identified in this way may migrate through the store lilvers from RAM to disk and back again.

This means that there are potentially a very large number of PIDs and secondly that since there is no fixed relationship between a PlO and a store location, the underlying addressing mechanism is associative rather than direct.

la o"JII\ arI •••.•• lall ••••••• llke PS-algol or Smalltalk objects may be simple things lik. 1trIa •••• am,. or 1he7""7 be more actiTe objects ca- pable of carrying ou' corn putation In IIMIr owa rlPt. All ••••• ple of thIo might be an abstract object like a dic'ionary that •••• ww* la _lall •••••• lIIto th_ of another. The following program fragment shows how "..sp\ ••••• tILtI _\ of oIIjoct ID PS-algol.

Ulint". _IQ' wIIa\ laterfa •• the dictionary object will present to the ouLsid. w{)rld. For •••••••••••••• !Ia\ 1\ call. ·Structure Classes".

```
     •••••••     lcdo...-,(
      JI'IIIc.trI.,.    )ot.riJIs} tJ'aIII;
      JI'IIIc.trIDJ,I\rIDs}   delille
      )
```

DII ••••• diet.., to be 1ft object that hu two attributes., an ability to m'lp str1ng' !nh) tTin~ •• to ••••••••••••••••••• Itrinp.

a) W. _ lid •••I CleMra10rPwIction that will manufacture dictionaries

```
     lot maII8.dIcI• proe(-:>pntr)
      ••••••
       lIructun   OIlIUtrla&name.<quiv;pntr left,right)
       lot 1IIId:-al1

       lot •• tor.01pIOt( cpntr curr.cell.temp -> potr)

       aurr.-II   .1111: tatp
       '-PC _)      <""rr.cell( name):  bq:in
         . aurr.-II(   Wt)  > enter( curr.cell( left ),temp)
           IIIn.-1I
       end
       temp( name ) > ""rr.cell( name ) : bqin
          eun.-II(   rtcht)  > enter( ""rr.cell( right ),temp )
          eun .••1l
       end
       daflult  : ""ru,ell

        dictionary(
       proe( Cltrla&n  ->string )
       boJin
             let P > head
             whll.  p - -  Dlland  n - -  p( name ) do
             P > It "  < p( name ) then p( left ) ebe p( right)
            It p la cell then p(<quiv) ebe ·unknown"
       ODcI,
       proeUtrlng 11•2)
       hood:•••ter<.11%1 1(Q1s2,nil.nlJ))
        )
     end
```

ThIs deAna make.dict to be • function that creates a diction.try with th~ func1ional d:tribute~ re qulred IDddne and translate 'IIOrds.

3) We now clec:1are cllc:ticmary in the global environment:

```
          let english.to.american -  mate.dictO
```

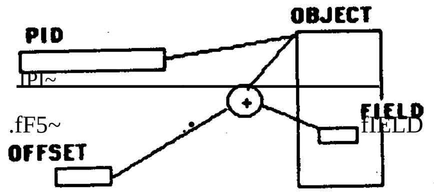
The q1ilh.to.american dictionary wilt then !ast u lone u the e10bal cn\"ironmcnl IL ~ L\. ht- tht~ ICICODds or months.

4) We OID then iDoertand lookup words:

```
          ""gUsh. to.amerlcaD(de6De)(" colour"." color" ,
          ""CUsh. to.amerlcu(cle6De)("   procramme"."  program")
          write   engU.h.to.amerlcan(tr&DlX"cll  vec!",
```

Box 1

Each object in the PS-algo1system has a unique identifier (called a PID or Persistent IDentifier in PS-a1gol) and ad- dres;ing has to be done in two parts, using an object ID. and an offset into an object.



1.5. Persistent Address Spaces

How big is a big address space?

One is accustomed to think pf a 24 bit address space as large and a 32 bit addr~ space as huge. Once you go over to persistent programming, 'things look a bit different. For a start you have to divide the address space up between different users. You could give every user their own 32 bit address space, but this would mean that they could not share data, which would not be very satisfactory. On a large machine yotl can easily have one or two hundred registered users. If you reserved a byte of the address to specify a user number you find that you arC' down to 16 megabytes per user. Remember that this has to hoJd not just transitory data that is used during computations, but alJ the 10ng term data that would normally go into the file store.

For most users at present 16 megabytes would be enough. but for some it is already a bit tight and we know that with the developmen t of computing the amount of store used has risen inexorably. If you start holding digitised images or digitised voice online then you are going to need a lot more memory. A 32 bit address space is likely to look rather tight for persistent programming within the the next decade or so. But just how big an adaressing system does persistent programming demand?

In languages that use a heap, a great number of objects are cn'iHrd that last only a short while. If each object were given a new object nUlTlhn. object numbers will be used up faster than the accumulation "I f'U-sistent objects would justify. Broadly there are two solutions to thts. either you make the object space so big that you never run out. or)'pu build in a garbage collector that is able to recover unused obJCct numbers.

The first solution seems to be the better. Once you go to billions of objects, conventional garbage collection techniques are likely to be too expensive. A pointer following garbage collector would be far too slow when working on a heap of this size. A reference count one would impose an overhead cost on every pointer assignment and upon every stack retraction. It is likely that on very large heaps it will be necessary to use incremental garbage collections based upon local information to tidy up local regions of tne global heap. Such techniques will hold onto some things that could be discarded, but provided that we are not going to run out of object numbers, this would be supportable. Infrequent1y used objects would migrate onto optical archive store. Some objects in archive store will never be reused because they are actually unreach-able though the system does not know it, but it is in the nature of archive stores to hold a lot junk that nobody is going to ever look. at again.

How big does your object address space have to be so as not to TU n out of object numbers?

Back of an envelope calculations indicate that a 48 bit object num ber is lik.ely to be enough. A computer creating half a million objects per second would take over 10 years to run out of object numbers, by which point it is likely to have been scrapped. In the end, in order to allow room to expand the address space accross a network, I chose a 64 bit PID (see box 2).

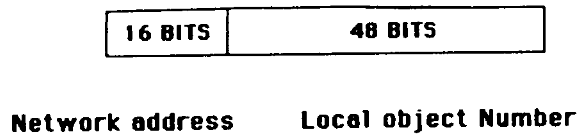
1.6. What chip to choose

There are now several microprocessors supporting virtual memory: the Intel 286 used in the IBM PC/AT, the Motorola 68020, the [ntel 432 and the National Semiconductor 32000 series. Which one is most suit-able for an associative, object oriented virtual memory?

Examination of all of these architectures, led me to the conclLLion that only the NatSemi 32000 series met all of the requirements. At first this may seem paradoxical, as the 32000 series (like the 68000 series) hiL~

Box 2

An obvious extension to the idea of a large persisten t add res.~ space is to allow the address space to be distributed over several machines. This requires a further extension of the addres.~spacr, If we allow for a local area network with 64000 machines, we find that a 64 bit key would uniquely identify every object ever cTrated "n any machine on the network.



the classic Von Neumann view of memory as a uniform array of words. whereas the Intel machines are explicitly object oriented. Both of them have serious limitations. The 286 allows far too few objects: each user only has access to 8192 object descriptors, this comes from it being ba~ cally a 16 bit machine. The intel 432 is a more serious proposition. Again it was rejected as having too small an address space. Other disad-vantages of the 432 were its low speed and the lack. of affordab1c com-pilers for it.

The National Semiconductor 32016 turned out to be nexib1c enough I-"r some quite simple additional hardware to turn its flat, paged, virtual memory into an associative object oriented virtual memory.

1.7. Programmers view of Poppy

To the machine level programmer, the Poppy's store is divided into 3 parts:

1 Machine registers make up the fastest store, it has 8 floating point registers, 8 general purpose registers and 8 special purpose registers.

2 Temporary values that will not fit into registers are stored on a stack.

3 Data of longer duration is held on a vast heap, made up of a practi-cally inexaustible number of named objects.

The Poppy instruction set is an extended version of the NS32000 archi-tecture. Certain of the machines addressing modes to enable them 10 dereference 64 bit object keys rather than 24 bit virtual add res,<;(':-;.

The basic NS32000 architecture supports a 24 bit virtual addr{'s.~space, which was well described in the April 1983 edition of Byte. Most addressing modes on the Poppy still operate using this space.

1 Register

The general purpose and floating point registers serve iL~the operands in Register address mode.

2 Register Relative

A general purpose rer~ contains a 24 bit virtual address. to which a displacement IS added to derive the effective address.

3 Memory Space

This is similar to the Register Relative address mode but uses one of the dedicated high level language regiaters PC,sP,FP,sB pIus an offset to get to the operand. These register again all contain 24 bit pointers.

4 Object Dereference

This is the new addressing mode added by the PSm to the basic NS32000. It allows a 64 Dit PID in memory to be derefenced. An offset is added to the start of the object refered to by the pointer to obtain the operand, it is explained in box 3.

5 Immediate

The operand is encoded within the instruction.

6 Absolute

An absolute addrelll within the 24 bit virtual address space is speci.1l.ed.

7 Top of stack.

The operand is Pushed (Poped) onto (from) the top of the stack.

8 Indexing

Any of the addrelll modes other than the Register or Immediate may be further qualified by an index. Indexing has the effect of calculating an "aI'ective address· (in virtual memory) and then mu\-tipying one of the general purpose registers by 1,2,4,or8 and then adding it to the total to get the final Effective address of the operand. The indexing option thus allows for arrays of bytes, short integers, integers, and PIDs or reals.

1.8. Poppy Hardware architecture

The Poppy is a single board virtual memory computer based upon the National Semicondutor 32000 series chipset, with a custom additional memory management unit. The overall logical structure of the processor is shown in the following diagram:

1.8.1. Processor

The ProcelllOris made up of 4 National Semiconductor chips. a timer. a CPU, a memory management unit or MMU and a floating point unit. These are connected via a 16 bit multiplexed private address/data bus. In normal operation the CPU outputs a virtual address onto the

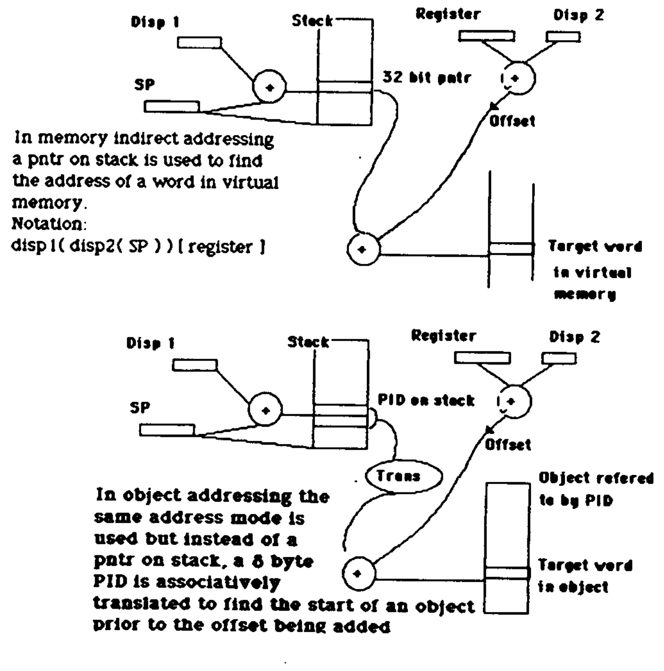
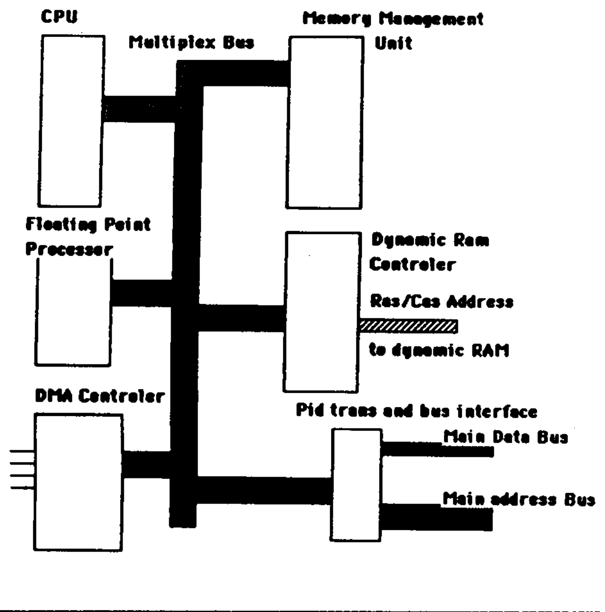

Disp 1
Stack
Register
Disp 2
SP
32 bit pntr
Offset
In memory indirect addressing a pntr on stack is used to find the address of a word in virtual memory.
Notation:
disp1(disp2(SP)) [register]
Target word in virtual memory
Disp 1
Stack
Register
Disp 2
SP
PID on stack
Offset
Trans
Object refered to by PID
In object addressing the same address mode is used but instead of a pntr on stack, a 8 byte PID is associatively translated to find the start of an object prior to the offset being added
Target word in object

Diagram on object addressing mode

Box 4



Processor Block Diagram

multiplexed bus during the first time interval of a store cycle. This is latched into the MMU which tranlates it into a physical addres'<;which is output on the next store cycle.

1.8.2. Memory

The physical address is latched and output to the demultiplexed address bus to which the· memory is connected. This memory is made up of both nonvolatile RAM and more conventional volatile Dynamic RAM. The nonvolatile RAM is made of high speed CMOS static ram with lithium battery backup to provide 10 year data retention. The dynamic ram can be made up of either 64k or 256k chips. To save board area the dynamic RAM IS mounted on 256k or 1 Meg SIPs, enabling either 512k bytes or 2 Mega bytes to fit into an area of 4.8" by 2.2". This way of mounting chips looks like becoming a a standard for dynamic rams since it gives about 4 times the density of conventional OIls. Data COIII-ming from or going to the memory travels along the demultipJexed data bus.

1.8.3. PID Translate Unit

Between the muItiplexed address/data bus and the demultiplcxed buses is the PlO translate unit (PTU). This unit supports the new ob'4l addressing mode. Logically it is a 512 word associative cache Il1cmory mapping PIDs to virtual addresses.

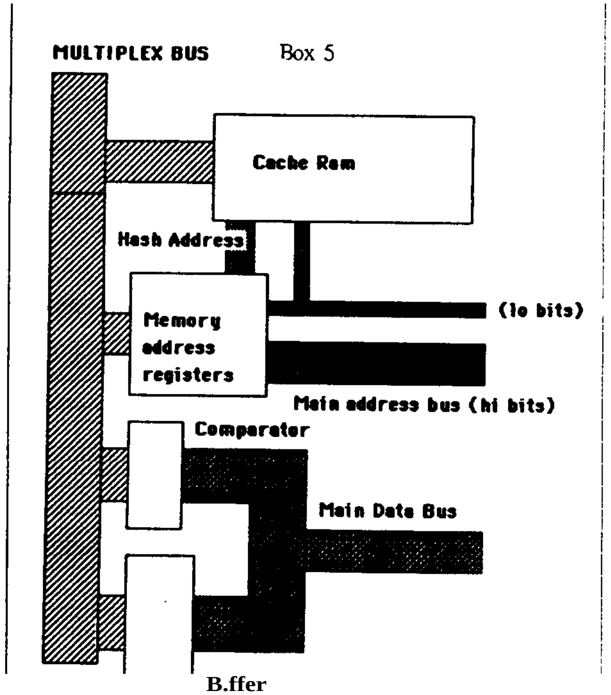


Diagram of the logical structure of the PTU

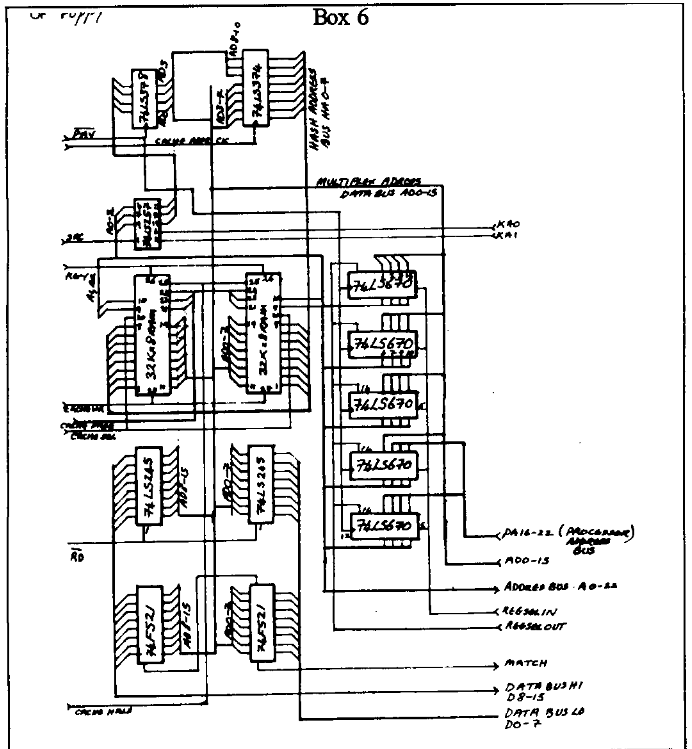
When the Processor uses the Object Indirect mode of addressing. the CPU thinks that it is just trying to fetch a 32 bit pointer from memory as per the NS32000 memory relative addressig mode. Instead of allowing this to happen, the PTU traps this, and fetches not a 32 bit pointer but a 64 bit PID from memory. It looks this up in the associative cache and returns the virtual address corresponding to the PlO. The proces,<;or "thinks" it is using the old memory relative addressing mode. when in fact there is an associative operation going on behind the scenes.

How does this associative memory work?

A true associative memory chip is a set of entries each made up of M bits of tag and N bits of data. When the chip is presented with an M bit pattern on its inputs, it simultaneously compares this with the tag fields of all the entries and if one of them matches, the correspondin$ N bits of data are returned. Because of technical difficulties in producmg such chips they are not readily available, and practical associative memories in applications like Mini or Mainframe cache memories have to be built out of conventional RAM chips.

These work on a modification of the hashing techniques used for software table lookup applications. Two banks of fast static RAM are used, the tag RAM which is M bits wide and the N bit wide data RAM. The input pattern is hashed to provide an address in the tag RAM and then the tag at that address is compared with the input tag, if they match then the data at the corresponding address in the data ram is returned.

Box 6



The implementation of the PTU

This is the technique used in the Poppy, except that for reasons of economy the tags and the data are held on the same chips. The PTU architecture is shown in Box 5 . It sits between the processor's private multiplexed address/data bus and the main address and data busses going to memory. It is made up of:

Buffer

Between the databus and the private address/data bus are a 16 hit bidirectional buffer and a 16 bit comparator connected in parallel. This buffer can be opened to allow the passage of data to and from the private bus. The comparator detects whether the contents of the two buses is the same.

The buffer is implemented wit 7415245 s and the comparator with a pair of 74f52l s.

Memory-Address-Registers

The private bus is connected to the address bus via two memory address rel!;isters, the Data Address Register and the Pid Address Register. These are implemented by using 2 of the 4 registers in a set of 7415670s.

Cache

The cache is made up of 8192 by 16 bit words of CMOS static RAM.

Cache-Address-Register

The low order 3 address bit of the cache come from the main address bus, but the mid order bits come from a separate 8 bit Hash register, and the high order bits come from the control unit.

Control-Unit

This unit monitors the CPU bus transactions and steers da tiI between the various busses and registers.

During a normal memory fetch the address is latched into the Data Address Register which is then output onto the address bus. During an object indirect instruction, the following sequence of events takes place:

1 The CPU outputs the virtual address of a PID on the stack, simultaneously the Status Monitoring unit spots that this is an Object Indirect addressing mode and puts the

2 The MMU translates this to a physical address

3 The PIU senses that this is a PID fetch and takes hold of the bLL~ and latches the physical address of the PID into the Pid Add ress Register.

4 The memory returns the first 16 bits of the PID, which pass through the bus buffers and a hashing function is performed to

produce an 8 bit result. This is latched into the Hash Address Register.

5 The bus buffers are disabled and the PID cache outputs the first 16 bits of the PID found in the cache onto the private bus.

6 This is compared with the corresponding 16 bits of the PID being read from main memory.

7 If they are equal the next 16 bits of the PID in memory and the PID in the cache are compared, and so on until all 64 bits of the PID from memory are shown to correspond to the pid in the cache.

8 If the two PIDs differed, then an address fault interupt is generated, otherwise the cache returns to the CPU the virtual address at which the object corresponding to the PID is located. The processor then adds the contents of an index register to the virtual address to find the field within object that the instruction wanL:~

1.8.4. Input Output

The board has two 10 interfaces. There are iSBX bus sockets and there is the BBC tube. Both of these are proprietary interfaces. The two of them are under the control of DMA and interrupt controler chips, the NSC 16203 and NSC 16202 respectively. The NSC 16203 provides 4 DMA channels and the NSC 16202 provides either 8 or 16 interrupt channels.

The iSBX bus is a simple 10 bus developed by Intel for their multi bus series of boards. It allows small daughter boards to be mounted on a CPU board. Each of these boards may contain one or two 10 devices. Several manufacturers produce these boards and the interface was chosen for its simplicity and the cheapness of the boards. Examples of the sort of functions that are available on iSBX boards are serial interfaces, parallel interfaces, floppy disk controlers, SASI interfaces and graphics boards. .

There are two iSBX bus sockets on the board. Each looks like a set of 16 memory mapped byte wide registers. Eaeh socket also has associated with it a DMA channel and an interrupt channel.

The Tube is a roprietary high speed 10 interface using custom VLSI chips develope for the BBC microcomputer, manufactured by Acorn Computers. Along with carefully designed software protocols it allows the Poppy to defe$ate all terminal, network, and disk 10 to the BBC microcomputer. High level commands are passed to the BBC machine which then executes the 10 operation concurrently with the Poppy, allowing the Poppy to continue with computation. The tube is connected to a DMA and an interrupt channel on the Poppy.

The BBC micro can control floppy disks and winchester disks and provides a bit mapped graphics display, with the option of a mouse interface.

1.8.5. Database Assist Hardware

Two additional pieces of circuitry have been included to speed up database searches on the persistent store. One of these is the PF474 string search processor described in the November 1984 edition of Byte. The other is a special hashing unit that is intended to be used for computing the indices of relational databases in persistent store ...

1.9. Storage management

Storage management software on the Poppy has to cooperatc with the hardware in presenting the programmer with a view of single. large. object addressed store within which all distinctions between differen ı storage media and the geographical location of these media are effaced. The storage system can be viewed from 3 levels:

- Object store
- Paged virtual memory
- Physical store made up of:
  - Non volatile RAM
  - Volatile RAM
  - Rotating store

1.9.1. Object store

The object store is made up of up to 2 to the power of 64 distinct objects. In principle each of these should be able to contain from 1 to 4 gigabytes of data. However for reasons to do with the restriction of the acfdressin~ of early models of the National Semiconductor 16032 series. the practJ.callimit for the size of each object is somewhat under 8 megabytes.

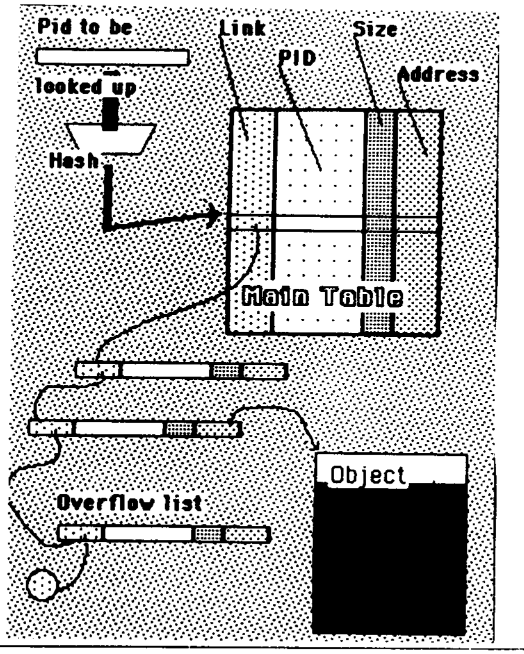
Each object is mapped onto the paged virtual memory of the processor as it is used, in a way analogous to the mapping of virtual pages onto physical pages in a conventional virtual memory system. On occasions objects have to be unmapped from the virtual memory of the 16032 in order to prevent the virtual memory becoming too full.

At run time the PTU will translate PIDs to virtual addresses provided that the appropriate PID/address pairs are loaded into the cache. If the PID is not found in the cache a PID/missing interrupt is generated and a software procedure is invoked to locate the object refered to by the PID and load the PID/address pair into the cache. The PID which caused the address fault has had its physical address stored in the PID Address Register. Indirection on this register enables the value of the PID itself to De located.

1.9.2. Pidlam

This software procedure uses a data structure termed the PIDLA M which is short for Persistent IDentifier to Local Address Map. The PIDLAM is a hash table in virtual memory with the structure shown in Box 7. Entries are found by Hashing a PID and then if necessary.

Box 7



The Pidlam

chasing down an overflow chain until an entry with the same pid IS found.

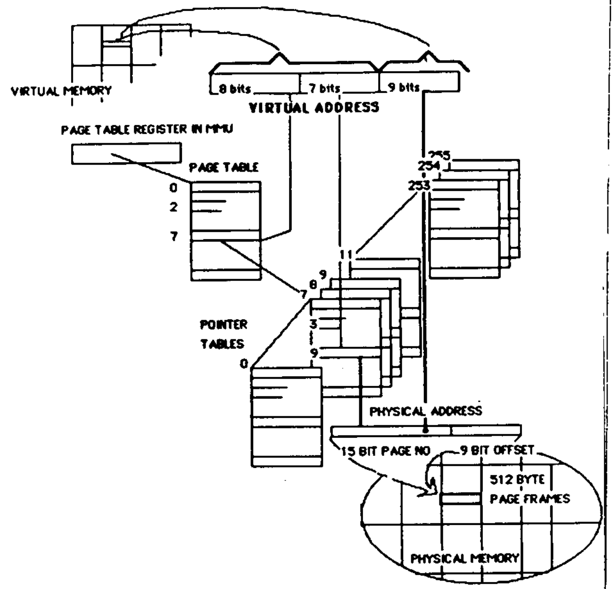
There is an entry in the PIDLAM for each object currently mapped onto virtual memory. Therefore, if an object is resident in virtual memory, the PID/missing interrupt can be met by searching the PIDLAM. Otherwise the object must be taken from rotating store and mapped or moved into virtual memory.

1.9.3. Paged Store

The paed virtual memory is accessed via two level page tables as shown In Box 8. The 24 bit virtual address of the NS16032 provides 16 megabytes of virtual address space.

For reasons of efficiency, we divide objects into two great cla."es - the pad and the non-paged. Large objects are paged, small ObjcCl~are no\. This distinction arises from a desire to make the best wish 0f the two types of virtual memory technology on the POPPY. We have paging hardware and object addressing hardware.

Box 8



Pal"ed Virtual Memorv on the POPPY

The object addressing hardware maps objects to virtual addre&es. the paging hardware maps virtual pages to physical pages. The simplest way to use these would be to allocate a range of vitual addreses for a heap. and map a working set of objects onto this heap. When an objc'ct w'as addressed and found not to be present, it would be copied from disk into the heap. This is what happens with existing software impleJl1(,11 tations of PS-algol.

This approach has a the drawbacks that if we are dealing with very large objects then we may be faced with the overhead of bringing in a lot more data than we actual need. If we t alter one word of a [0000 element array, the whole array is still COptedin to the heap.

To overcome this, we chose to COpy small objects onto the heap. but to MAP large objects onto to virtual memory.

A mapped object need not all be resident in physical memory, instead it occupies a range of virtual pages individual members of which are brought into physical memory on demand. Non-paged objects are copied into a heap on demand. To the extent that the heap into which they are coppied is itself paged, then they also need not be physically resident. However, it does seem reasonable to keep the virtual size of the heap sufficiently small as to ensure that most of the heap is likely to remain in in physical RAM Otherwise, we would be faced with having two swappmg mechanisms competing with one another.

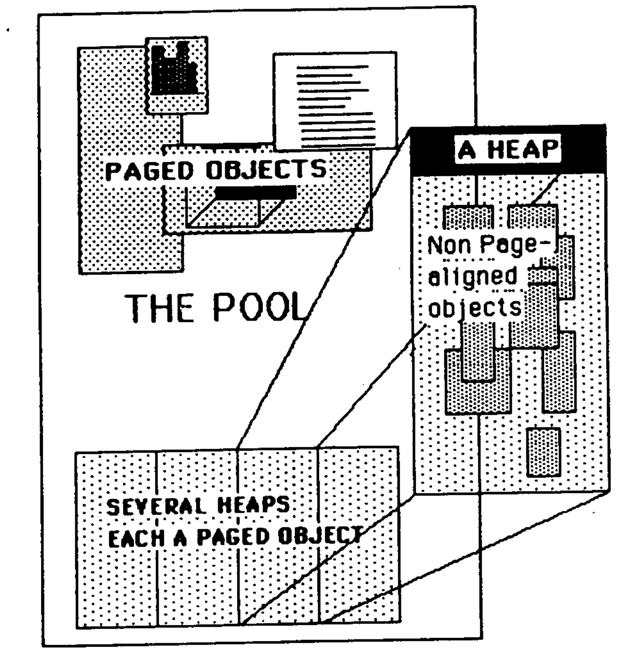
Only one copy of each object is ever present so that all transactions that can access the object work on the same copy. The definition of the loca· tion of an object in virtual memory is handled by the PIDLAM (Per. sistent ID to L:.ocal Address Map).

1.10. Making Object Oriented Languages really rtm

At least initially, the Poppy will be a single language machine, the software will consist of an interactive PS-algol complIer and its run time support. Any data declared at the outermost level of the system. whatever its type will persist indefinitely.

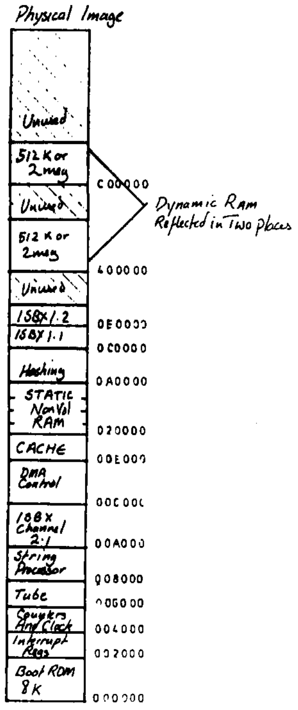
For some time PS-algol has been running on the ICL/Three Rivers Perq computer, which is a workstation in the same general class as the Xerox star. Because this lacks the necessryassociative addressing hardware. the language runs slowly. The experience of Smalltalk implementations too is tl1at it is difficult to get a satisfactory performance unless you have a very high powered machine like the Dorado (which costs tens of thousands of dollars). Hopefully, the type of simple associative hardware used on the Poppy should enable these sophisticated languages to be run fast on the next generation of cheap personal computers.

Box 9



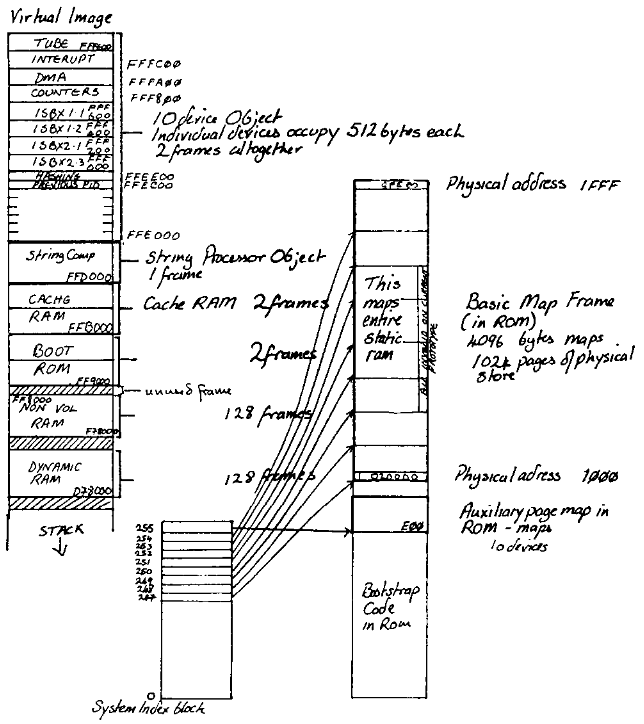
The Pool

BOX 10



Poppy Physical Memory Map

BOX 11



Poppy Virtual Memory map and Standard Page Tables
This shows how the uhYSical resources are maured onto virtual