Uncoupling central processor and storage device speeds

Caxton C. Foster

University of Massachusetts, Amherst, Massachusetts 01002, USA

This paper discusses a method for coupling a very high speed CPU to relatively slow storage devices. The method consists of treating storage references in the same fashion as multi-programming systems treat I/O references.

(Received March 1970)

As is well known, it is presently possible to design and construct Central Processing Units which can accept a new instruction 50 to 100 times per microsecond. It is equally well known that a design which envisioned all of main storage operating at this speed would be prohibitively expensive to build. Anyone attempting to design a computer for use in a utility situation will wish to have high throughput, large main store and good cost-performance. Thus, a designer is faced with the problem of coupling together high speed CPU's and relatively slow speed storage devices.

Two different approaches to this problem have been tried in the past. The first of these has been called a slave memory (Wilkes, 1965). A high speed (comparable to the CPU cycle time), but small, scratch pad is provided. The CPU fetches instructions and operands from this scratch pad, and hardware is provided to trap address faults when the desired data is not present in the scratch pad. Thus, main storage is, in effect, treated as a virtual memory which is paged in and out of the scratch pad. This approach is an outgrowth of Kilburn's pioneering work on the one-level-store of the ATLAS computer (Kilburn, Edwards, Lanigan, and Sumner, 1962) and Arden, Galler, O'Brien, and Westervelt's (1966) design for the 360/67. But the problems of page thrashing (Coffman and Varian, 1968) cannot be ignored, and, despite the fact that Conti, Gibson, and Pitkowski (1968) claim 80% efficiency for the 'cache' of the IBM 360/85, one cannot help but be concerned about a design with two levels (scratch pad-core-drum) of paging, however artful the designers and programmers may be.

The second approach is to break up main storage into a number of independent blocks so that several accesses to storage may be overlapped. The CPU tries to 'look ahead' down the stream of instructions and 'pre-fetch' data and instructions that will be needed in the near future. This method is often combined with a CPU design that separates the logically independent operations of decoding, address calculation, and various kinds of execution. The CDC 6600 and the IBM 360/195 are examples of this approach.

The major problem with this approach is the interlock circuitry required to keep one instruction from treading on another's heels. For example, if a separate multiplier unit is provided (*à la* the 6600) and it is slower than the adder, then the following instruction sequence could easily give erroneous results if there were no interlocks.

1	LDA	JONES
2	MPY	SMITH
3	STA	BROWN
4	LDA	GREEN
5	ADD	BROWN
6	STA	GREEN

If the multiplier should be busy and the adder free, then without interlocks instruction 5 could be executed before instruction 3. Obviously, as the look ahead gets deeper the interlock circuitry gets more complicated. Moreover, given conditional branch instructions, the chances of completing any pre-fetched and partially decoded sequence is reduced and sometimes considerable effort can come to no avail.

The method that we are going to propose in this paper is somewhat like a 6600 without interlock circuitry, but rather more like an overlapped Honeywell 800 with something of the flavour of a Gamma-60 thrown in. Unfortunately, our method requires a special sort of environment in order to be effective, but as we shall see that environment (having many independent tasks ready for execution) will not be at all rare in a computer utility.

Multi-streaming

The CDC-6600 achieves its high speed by storage reference overlap at the cost of complex circuitry. The Honeywell 800 reduces system overhead by interleaving execution of up to eight independent instruction streams but without any attempt at overlap or look ahead. The Gamma-60 (one of the earliest multi-programmed multi-processors) had an elegant scheme for handling requests for use of a processor when that device was already busy. We intend to 'borrow' all three of these ideas.

Consider a computer utility. By its very nature it will be characterised by having many tasks in progress, or waiting for attention, at the same time. Suppose we could arrange to have several tasks, each in independent storage modules. Then we could design a central processor like the H-800 or like the peripheral processing units of a CDC-6000 series, and time-share one fast CPU among the various tasks. The PPU's of the 6000 series are each allotted a time slice at the CPU whether they need it or not. The H-800 overcomes this problem but does not attempt to overlap memory references generated by one program with those generated by another.

The simplest way to both overlap storage references and to assign CPU attention only to those instruction streams able to profit from it, is to allow the *completion* of a storage reference to be the signal that the program which generated that reference needs some CPU time.

Suppose the CPU cycle time is τ and the cycle time of a storage device is $n\tau$. Then **Fig. 1** shows a primitive multi-streamed device which time-shares the fast CPU among n slow storage devices. Each storage device has a scratch pad associated with it to hold a copy of the contents of the CPU's registers appropriate to its own program. It must also have its own memory address register and memory buffer register. Note especially that the storage devices can be of mixed speeds. Programs stored in faster devices will be executed more rapidly. Also, if we permit one storage device to 'talk to' another (by extending the address field, for instance, to select a device as well as a cell), some of them could be input-output devices such as card readers or line printers or perhaps teletype units. We require enough devices so that their combined average rate of requests for attention will keep the CPU busy. Other than that, we are free to choose as we please.

The scanner should be of a look-ahead type, doing a round-robin scan of the devices, searching for one requiring attention. The multiplexor must have a rather broad highway in order to simultaneously transfer all the CPU register to and from the scratch pads. For this reason alone we should keep the CPU simple with only a few registers.

Perhaps eight registers of 32 bits each arranged somewhat like a PDP-11 would be a good starting point for a design. This should strike a reasonable balance between having too many registers with a high cost for scratch pads and too few registers with a high cost in increased memory references.

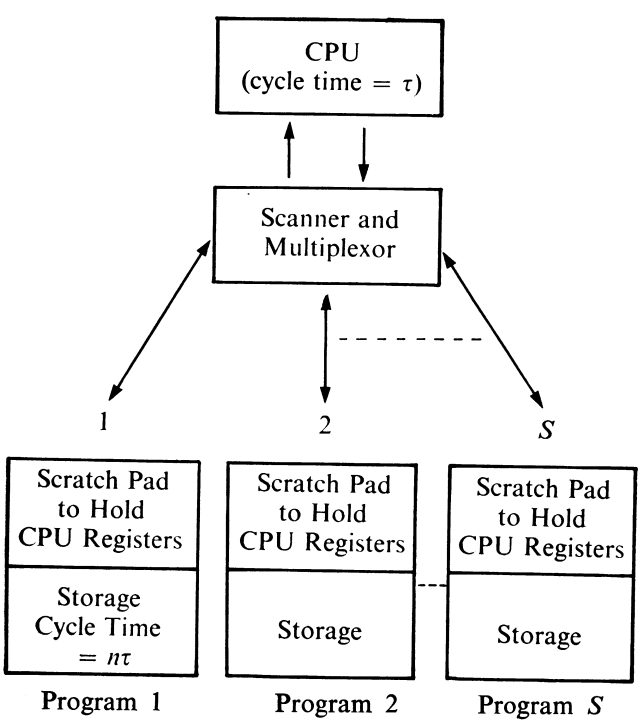

Fig. 1. A primitive multi-streamed architecture

In effect, this design carries the idea of multi-programming to its logical extreme. We treat main storage just like an I/O device and 'interrupt' an instruction stream every time it references store (or any other I/O unit).

In the machine so far described, we have shown how to time-share one fast CPU among several programs—each one running as fast as it can, given the storage device holding it. But we have not yet taken advantage of the law of large numbers as applied to the size of programs. Some programs are small, some are large requiring much storage space. In the primitive machine each program has its own storage device, and it would be impossible to share unused space in one device with a need expressed by another program. The best we could do would be to buy some small and some large storage devices; but this would increase the diversity of equipment installed (and hence the price) and would not be as flexible as the scheme described in the next section.

An integrated multi-stream organisation

The first change we make to the organisation of Fig. 1 is to collect the scratch pads that were associated with each storage device into a common scratch pad called the DORM in which instruction streams sleep when not executing in the CPU. Each instruction stream will be known by the address it occupies in the DORM.

The DORM will need to be wide enough to hold all the CPU registers. If we have eight registers 32 bits long this will require 256 bits per word. At 3 to 4 dollars per bit active integrated circuit memories can be obtained with read or write times of the order of 10 nanoseconds. Although the DORM is logically a single unit it might be profitable to consider constructing it as several physically independent units to achieve access overlap.

We require one read and one write from the DORM every CPU cycle but the read can be destructive and the write, which will be to a different address, can employ separate access circuitry.

Later in this paper we propose to have 256 active instruction streams so this would make the DORM a square array 256 on a side.

The memory address registers and memory buffer registers will stay with the storage device and we must add another register to each storage device called the USER register in which to hold the name of (the DORM address of) the instruction stream for which the device is currently working.

We do this for two reasons. First, there may be more devices than there are instruction streams, and in this case we will save on the total amount of scratch pad required. Secondly, if we are going to share storage among programs, we wish to give the programs an identity independent of which storage device they are currently using.

If instruction streams are disassociated from storage devices, then it will be possible for more than one stream to reference the same device and we must provide a mechanism for resolving this interference. Obviously, when two requests for service are outstanding simultaneously, one of them must wait. Preferably this delay in servicing the second request should not tie up the CPU. The hardware provided to accomplish this will be called the 'queueing mechanism' and is derived from that of the Gamma-60.

Further, we should do our best to attempt to minimise memory access conflicts. One way to do this is to interleave storage units on the low order bits of the address. If this interleaving is, say, 16 deep (4 bits) then up to 16 programs could be executing the same subroutine, each program moving one instruction behind the previous one. There may be some delay in getting started if jumps to the subroutine occur too close together, but the programs will sort themselves out (by order of request) and soon be marching along in 'lock step' without mutual interference.

The queueing mechanism

The remaining changes to the organisation of Fig. 1 are to accommodate multiple requests for service from one storage device. They are shown in **Fig. 2**. To each storage device we add a register called LAST and we add a QSTORE scratch pad. The QSTORE has as many words as does the

DORM; one for each instruction stream. Each word of the QSTORE is long enough to hold one memory address, one word of data and the name (number) of an instruction stream plus two more bits.

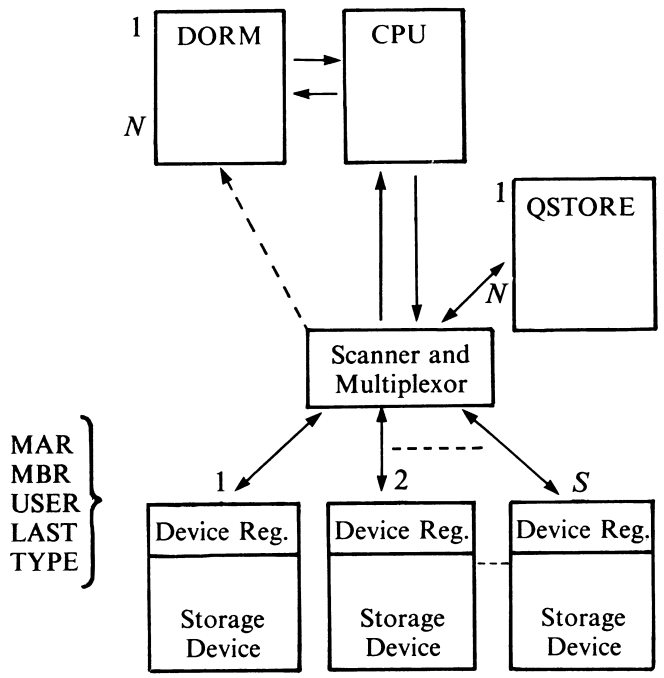

Fig. 2. A more sophisticated multi-stream architecture capable of sharing storage space and common subroutines

If a storage device is idle (has a user number of 0) when an instruction stream requests its service, then the user number, the effective address, the data (if any) and two bits to indicate the 'type of request' are copied into the USER, MAR, MBR and TYPE registers of the device. Simultaneously, the user number is loaded into the LAST register of the device. The TYPE of request is a two bit code indicating (1) instruction fetch, (2) read, (3) write, or (0) do nothing. Finally, the operation is initiated. Meanwhile, the CPU registers have been copied into the users word in the DORM so the CPU is free to go about other work.

Suppose, however, the device is busy (has a non-zero user number). Then the LAST register of the device is examined to discover the name of the instruction stream at the end of the queue waiting for service from this device. Suppose instruction stream J is making the request, and that instruction stream K is named as being at the end of the queue. Then the CPU registers of J's job are placed in J's word in the DORM as usual, but instead of sending the information about J's memory request to the desired device, it is diverted to the K^{th} word of the QSTORE, and J's name is then put into the LAST register of the device. **Fig. 3** shows this process. As more requests arrive, they form a linked list—via the USER field—in the QSTORE of those streams waiting for service from this one device. Since an instruction stream gets interrupted for every storage reference, it can have, at most, one request outstanding and be on, at most, one queue. Thus it needs only one slot in QSTORE to hold the name and data about its immediate successor on that queue. Note that we will make read out from the QSTORE be destructive so that zeros are left in any QSTORE word belonging to an instruction stream without a successor, (not on a queue, or at the end of one).

Suppose now that the device shown in Fig. 3 finishes its work for stream K and eventually obtains the attention of the scanner. The K^{th} word is read out of the DORM to load the registers of the CPU and the data (if any) that results from the storage reference is passed along and the device is ready to work for someone else. The contents of the K^{th} word of the QSTORE are fetched and sent to the device registers. If this word was all zeros, then the device falls idle. If not, it begins work for the new user. A detailed discussion of the timing of these various activities is contained in Foster (1970).

Delays due to interference

It is always difficult to estimate the amount of interference that will result in an actual design before that design is built and used; partly because the design of software can strongly influence the results.

Lacking any other knowledge, let us assume that storage addresses are generated at random—each address being equally likely. Then if there are N instruction streams, each of which has generated an address (we ignore the one that is currently executing) and there are S devices, the average number of requests per device will be:

$$\lambda = N/S$$

Given random selection of addresses, the number of devices with i requests will be:

$$D_i = S \cdot \frac{\lambda^i}{i!} e^{-\lambda},$$

and the number that are idle (with zero requests for service) will be:

$$D_0 = Se^{-\lambda}$$

or for $\lambda = 2$ (twice as many streams as storage devices) about 13·5% (e^{-2}) will be idle and 86·5% will be busy.

Let us assume 128 storage devices with an average cycle time of 1 μ second. Since only 86·5% of these will be generating requests for attention by the scanner, we can expect $128 \times .865 = 110$ requests per microsecond, or roughly one every 10 nanoseconds. Providing the CPU can keep up with this request rate (see next section), then the throughput will be 10^8 storage references per second. With 128

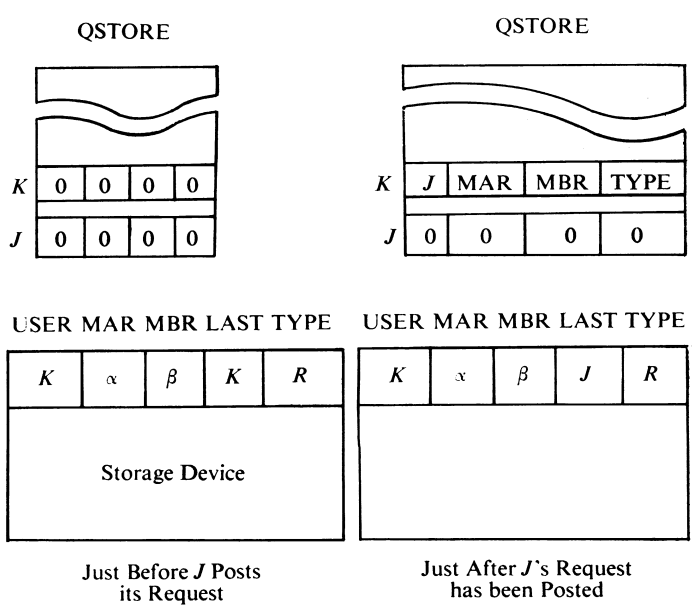

Fig. 3. The process of posting a request by instruction stream J on a device that is busy. The queue then consists of K (being serviced) and J (waiting on the queue). If another instruction stream M comes along, its name will be put into LAST, and information about its request will be stored in word J of the QSTORE

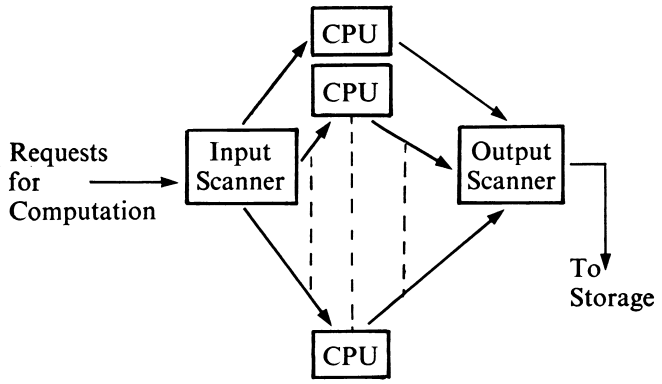

Fig. 4. A parallel organisation for a CPU

devices and a $\lambda = 2$ we will have 256 active instruction streams which must share these 10^8 references. Consequently, the average program will proceed as if it were running alone in a machine with a 2·5 μ second main store, executing 400,000 references or 200,000 instructions per second. One may well question whether this rate could be attained in practice but without a detailed study the assumption of random addressing is probably as reasonable as any other.

CPU design

We have suggested in the previous section that a CPU speed of one instruction accepted every 10 nanoseconds could be designed. The fastest present designs (CDC-7600 and IBM 360/195) have about a 20-25 nanosecond cycle time. But they must operate with extensive interlocks to prevent undesirable interaction between instructions, whereas we are freed from that necessity by the fact that successive instructions in the same stream can arrive at the CPU no more often than 1 per microsecond (the storage cycle time).

Two approaches to designing a high throughput CPU could be tried. The first of these would be a pipeline design in which independent operations of decoding, etc. would be assigned to separate units which passed on their partial results to the next unit. But there is a limit to how far we can partition instructions, and hence to how much we can profit from pipelining.

The second approach involves parallelism of operation, but without the specialisation of components that is characteristic of the CDC-6600.

Fig. 4 shows the essence of the proposed CPU design. Several, perhaps 12, relatively high-speed (100 nanosecond) CPU's are connected in parallel.

When a request for computation arrives the input scanner looks for an idle CPU and passes the task to it. Some time later, that CPU requests attention of the output scanner and passes its results and registers on to be saved as described previously. Then that CPU indicates to the input scanner that it is ready to accept another job. Since all CPU's are identical, when one goes down, the rest can carry on with a somewhat reduced throughput. Of course, if one of the scanners goes down, the whole system would be paralysed, but these are relatively small in size and could be backed up by duplicate equipment without too great an investment.

In order to achieve high speed within the CPU's we suggest that they might be specialised into three distinct types. The Scanner and Multiplexor can examine the type of memory reference (instruction fetch, read, or write) just completed and select a CPU on this basis.

1. *Fetch* completed. This type of CPU will be the most complicated. The instruction will be decoded and if it is a memory referencing kind, an effective address will be computed. In this case the CPU has done all it can and will initiate the reference and put the instruction stream back to sleep. If the instruction is not memory referencing it will be executed at once and after execution a fetch of the next instruction will be initiated.
2. *Read* completed. This type of CPU will be used when a data reference is finished. The action dictated by the op-code is carried out and the stream is put back to sleep after requesting a fetch of the next instruction.
3. *Write* completed. At the completion of a store operation all that needs to be done is request the next instruction. This type CPU will thus be almost trivially simple.

Whenever indicated these CPU's can be designed on a 'pipeline' basis taking advantage of the independence of successive instructions. This technique would be of particular advantage in Type 1.

Conclusions

The outline of the design of a very high throughput computer is presented. A fast CPU (or combination of CPU's) is coupled to a large number of relatively slow storage devices. The completion of a storage reference is used as the signal that more computation is needed by a program.

The storage devices can be of various speeds. High priority programs could be stored in faster storage while I/O bound programs could be relegated to slower devices.

Acknowledgements

The author wishes to thank Professor Michaelson, Chairman of the Computer Science Department, University of Edinburgh, for many stimulating discussions and suggestions on this design during the year 1967–8 while the author was visiting the University of Edinburgh.

References

ARDEN, B. W., GALLER, B. A., O'BRIEN, T. C., and WESTERVELT, F. H. (1966). Programmed Addressing Structure in a Time-Sharing Environment, *JACM*, Vol. 13, No. 1, pp. 1-16.

COFFMAN, E. G., and VARIAN, L. C. (1968). Further Experimental Data on the Behavior of Programs in a Paging Environment, *CACM*, Vol. 11, No. 7, pp. 471-474.

CONTI, C. M., GIBSON, D. N., and PITKOWSKI, S. H. (1968). Structural Aspects of the System/360 Model 85, I. General Discussion, *IBM Systems J.*, Vol. 7, No. 1, pp. 2-14.

Control Data 6400/6500/6600 Reference Manual, Publication No. 60100000, Palo Alto, California: Control Data Corporation, 1966.

FOSTER, C. C. (1970). *Computer Architecture*. New York: Van Nostrand Reinhold Co.

Gamma 60, A New Concept, *Data Processing* (January-March, 1960).

Honeywell 1800 Programmers' Reference Manual. Wellsely Hills, Mass: Honeywell Electronic Data Processing, 1964.

IBM System/360 Model 195 Functional Characteristics. IBM Systems Reference Library, Form A22-6943-0, Poughkeepsie, N.Y., 1969.

KILBURN, T., EDWARDS, D. B. G., LANIGAN, M. J., and SUMNER, F. H. (1962). One-Level Storage System, *IRE Trans. on Elect. Computers*, EC-11, pp. 223-235.

WILKES, M. V. (1965). Slave Memories and Dynamic Storage Allocation, *IEEE Trans. on Elect. Comp.*, pp. 270-271.