Quaestiones Informaticae

Vol. 3 No. 1 February, 1984

Quaestiones Informaticae

An official publication of the Computer Society of South Africa and of the South African Institute of Computer Scientists

'n Amptelike tydskrif van die Rekenaarvereniging van Suid-Afrika en van die Suid-Afrikaanse Instituut van Rekenaarwetenskaplikes

Editor: Prof G. Wiechers
Department of Computer Science and Information Systems
University of South Africa
P.O. Box 392, Pretoria 0001

Subscriptions

Annual subscriptions are as follows:

	SA	US	UK
Individuals	R6	$7	£3,0
Institutions	R12	$14	£6,0

Circulation manager
Mr E Anderssen
Department of Computer Science
Rand Afrikaanse Universiteit
P O Box 524
Johannesburg 2000
Tel.: (011) 726-5000

Quaestiones Informaticae is prepared for publication by Thomson Publications South Africa (Pty) Ltd for the Computer Society of South Africa and the South African Institute of Computer Scientists.

A Virtual Multiprocessor Computer Design

Trevor Turton
IBM South Africa

Abstract
This paper describes computer design which would achieve a high instruction throughput rate on a relatively modest amount of hardware. It is based on multiprogramming the CPU at a sub-instruction level. The design will cope well with current computer workloads in on-line environments, where conditional branches and interrupts are frequent. The paper shows that when current high performance computers execute general jobstreams which include on-line work, they require about five machine cycles for each instruction completed, while this design should achieve close to one instruction per cycle. This level of performance can be realized without the large investments in complexity which current high performance computers make in order to reduce the number of cycles per instruction. Furthermore, the design can be made to be program compatible with any current general purpose multi-processor.

1.0 Performance Limitations on Current Computers

General purpose computers require certain functional components and storage elements to perform their tasks. These include registers, storage, instruction decode units, instruction execute units (including adders, multipliers, shifters, etc.) and data channels for communication with I/O devices. On high-performance machines, the functional components tend to be implemented as discrete but interconnected units, in order to achieve the high levels of performance required.

Each instruction that a computer executes must go through a series of these functional components one at a time. For instance to perform an add, the CPU will have to fetch the instruction from storage, decode it, generate the operand adress, fetch the operand, and route the operation via the adder to its destination register. If these operations are perfotmed serially then each individual component will be used only a small fraction of the time and overall systems throughout will be poor.

The following table illustrates the problem. Some common CPUs are shown with their approximate instruction processing rate in millions of instructions per second (MIPS) and major machine cycle times in nanoseconds [5,7]. The average number of cycles per instruction completed is also shown.

Machine	Cycle Time	MIPs	Cycles/Instruction
CDC 6600	100	4	2,5 (scientific)
CDC 6600	100	1	10 (on-line)
IBM 3033 U	57	5	3,5
IBM 370/195	54	2	1,5 (scientific)
IBM 370/195	54	3	6,2 (on-line)
AMDAHL 470V/7	29	5,7	6,0
CDC CYBER 176	27,5	8,1	4,5
IBM 3083 J	26	8	4,8
CDC CYBER 175	25	5,3	7,5
CRAY-1	12,5	140	0,6 (vectors)
CRAY-1	12,5	20	4,0 (scalars)

Each of these machines is capable of decoding and issuing an instruction every cycle, but generally their execution rates fall far short of this rate.

2.0 Current Approaches to Increased Performance

This poor utilization of CPU components is similar to a problem which arose with early computers. Every time the CPU required I/O, it would initiate it and then wait for the operation to complete before proceeding. The I/O unit involved would idle while the CPU handled previous input. This problem was largely overcome by overlapping CPU and I/O operations. I/O buffers were provided in CPU storage to allow input devices to read in data ahead of the program's needs, and to allow program output to stack up and await the services of the output devices. This allowed the CPU, input and output devices to operate concurrently instead of serially, and substantial performance improvements resulted.

2.1 Instruction Fetch and Execute Overlap

This same technique of buffering has been used with considerable success within CPUs. On most medium to high speed machines, instruction fetch and execute are separate and concurrent functions. Instructions are fetched ahead of being needed into instruction buffers, then executed when the execution unit becomes available. Since main store is considerably slower than the CPU's execution units, this approach reduces the amount of time that the CPU has to wait for storage. Execution units have speeded up over the years, and to keep them reasonably busy, several instructions must be pre-fetched, and also any operands they might need to access in storage.

Unfortunately this technique cannot be extended indefinitely. Instruction steams contain conditional branches, and only once they are executed does the CPU know which instruction stream to pre-fetch. IBM produced a high speed computer in the early 1960's called the 7030 or Stretch, which attempted high performance by extended instruction pre-fetch. Many jobs were found to have a high conditional branch content, and these prevented the machine from achieving its target throughout.

The IBM 3033 processor design tackles this problem by an improved ability to handle conditional branches [6]. It predicts more accurately the outcome of each branch, and can pre-fetch instructions for up to two alternate branch paths, by having three independent sets of instruction buffers. A major hardware investment is made to try to ensure that there is always a queue of instructions waiting for the execution unit to work on , with their operands pre-fetched. Despite this, and the fact that the execution unit completes most instructions in one or two cycles, te 3033U completes on average only one instruciton every four cycles.

2.2 Cache Store

On larger CPUs, the relative slowness of main store is a major limiting factor on performance. A sophisticated system of storage buffering was developed for the System/360 Model 85 to alleviate this problem. A high speed cache store was insterted between the CPU and main store so that the instruction execution units could operate directly to and from this store, provided it contained the instructions and operands required.

The CPU could not know which areas of main store to cache in advance. It relied on fetching on demand. This resulted in occasional delays, but almost all programs iterate in confined storage areas, so subsequent iterations had their data immediately available in the cache store. In practice a cache only 1 % the size of main store can satisfy more than 97 % of the requests to store. This approach has become an industry standard for machines of one MIPs and more.

2.3 Multiple Instruction Overlap

Other attempts at improving processor throughout were made by the simultaneous execution of several successive instructions.

2.3.1 The CDC 6600

In the mid 1960s CDC developed a new approach to the problem of CPU throughput for their scientific processor, the 6600. This approach was continued in the CDC CYBER 170 series. The 6600 incorporated an instruction preparation unit and nine autonomous execution units, each specialized to handle only a subset of instruction types, e.g. add, multiply, divide. The instruction preparation unit was able to decode and issue one instruction per machine cycle. Each instruction was routed to the execution unit appropriate to its needs. Execution of these instructions ten proceeded simultaneously in each execution unit. A high degree of execution overlap was possible. Most of the execution units were pipelined so that they could accept a new instruction every machine cycle. The execution units had the combined capacity to execute about 7,5 operations per machine cycle (i.e. 75 million per second), even though the supply of operations could never exceed one per cycle.

On selected hand-coded programs the CDC 6600 could approach one instruction completed per machine cycle, but on general job-streams which included on-line work the throughput dropped to one instruction every four to six cycles. This implied that the instruction preparation unit was only 20 % utilized, and the combined execution unit capacity only about 3 % utilized. The reason that this architecture could not maintain a consistent one instruction per machine cycle rate was because in practice successive instructions in a program are not independent. They require operands which are produced by preceding instructions. Unrelated operations have to use the same limited number of machine registers, which forces some serialization. Conditional branches cause potential forks in the instruction stream, which can only be resolved when the branch instruction has completed. This implies that none of the instructions following the conditional branch may be executed till the result of the branch is known.

2.3.2 The IBM 360/195

During the late 1960s IBM developed a new model in their 360 range to cater for high performance scientific applications. This machine, the Model 91 [1], significantly extended the principles evolved for the CDC 6600. Instruction decode, fixed point arithmetic, floating point add/subtract, and floating point multiply/divide were all separate autonomous units. Also separate were main store read/write control and instruction prefetch. Several instructions could be executed simultaneously. The machine did not wait for one instruction to complete before decoding and scheduling the next. Up to sixteen words of instructions could be pre-fetched and executed at the same time.

Performance limitations arose with the Model 91 for the same reason that they did with the CDC 6600. Instruction interdependency forced the model 91 to observe extensive ordering of instruction execution. Much sophisticated circuitry was required to detect these interdependencies and keep track of which operands became available when, and where they were required.

Handling conditional branches also severely impacted the performance of the Model 91. When a branch was met in the instruction steam, the CPU would initiate the instructions following it, but not allow any of them to complete until it knew whether the branch would be taken or not. The Model 91 attempted to reduce this disruption by hedging its bets. It prefetched, decoded and issued instructions past the branch, but also pre-fetched four words of instructions from the branch target to use in case the branch was taken (a two-way Stretch aciton). If the branch were taken then the new instruction stream had to be decoded and issued to the execution units, which idled in the interim.

The Systgem/360 Model 195 was essentially a combination of the Models 85 and 91, having both a memory cache and concurrent istruction execution. The Model 195 execution units had enough power to execute 70 million instructions per second (MIPS), but the instruction preparation unit could issue only 18 MIPS. On selected scientific applications the machine achieved 14 MIPS, but on general jobstreams this dropped substantially. When running the Airline Control Program (ACP) which supports on-line transaction processing, its throughput dropped to about 3 MIPS [3,4]. This means that at most 20 % of the total execution unit power could be exploited, and at worst, 4 %.

The problems encountered with conditional branching described above have become more severe with the passing of time. Computer workloads have become more and more on-line and demand driven. This has resulted in an increasing percentage of conditional branch execution, and the attempts by modern computers to achieve concurrent execution of many instructions have become increasingly ineffective.

2.4 Parallel and Pipeline (Array) Processing

Other high-speed computers are designed to carry out instructions over complete arrays of numbers at a time. Examples are ILLIAC V, CDC STAR and CRAY-1. These devices are well sujited to applications involving array operations, but are not cost-effective when normal scalar oriented workloads are involved. For example, the CRAY-1 processing a scalar workload completes about one instruction every four cycles (20 MIPS), while it has enough execution power to complete 12 operations every machine cycle (960 MIPS). In this instance only 2 % of its total execution power is exploited. However given a suitable program of vector arithmetic it can get multiple execution units busy concurrently, and complete 1,75 operations per cycle, i.e. 140 MIPS [7].

2.5 Multiprocessing

The conventional approach to multiprocessing is to connect more than one CPU to a common memory. Since both cost and performance go up together, no significant change in the performance/cost ratio results. Multiprocessing in this way is often employed to improve systems availability by redundancy of all critical systems components.

3.0 Performance Limitations for Future Computers

It is impossible to know what new problems will beset computer designers in the future, but some extrapolation of current problems can be done. Demands for ever higher switching speeds will continue. Component sizes will reduce. The reduction of inter-circuit signal propagation delays will become increasingly important, and circuit packaging densities will increase. Heat dissipation will necessitate liquid cooling, and ultimately the adoption of technologies which generate far less heat per circuit, such as Josephson Junctions.

There is today a disparity in the speeds of logic circuits and main store subsystems in all large computers. Execution units are much faster than main store. As execution units become faster, so the depth of multiprogramming of the machines will have to increase. This is because the I/O devices which the programs communicate with are mechanical devices, and their response times have not improved at the same rate as electronic technology has. Unfortunately, this disparity is likely to worsen rather than improve.

Each concurrently active program requires a working set of real storage. Hence the amount of real storage required by a large scale processor will tend to increase linearly with processor speed. As the amount of main store increases, so its physical size must increase. This will result in increased signal propagation delays. So as computer speeds improve, the performance disparity between execution units and main store will worsen. This statement would hold true even if the components used to implement main store had the same switching speed as those in the execution units. It is not economic to do this today, and will become increasingly less economic as the relative size of main store increases in the future.

Cache store is currently used to mask the disparity between

the speed of execution units and main store. A well-organized cache store just 1 % of the size of main store can service 97,5% of all references to main store. As main store sizes grow, cache sizes will have to grow too, and keeping the performance of cache at the same level as the execution units will become increasingly difficult. Furthermore, the cost of a cache "miss" will become more severe as main stores grow and get slower. If the executive units of an IBM 3083 were coupled to the economic 64k bit chips used in IBM 4300 main store, then a 2,5 % cache miss rate would slow the machine down by about 50 %. If we extrapolate to a CPU with execution circuitry ten times faster, the same cache miss rate would slow this machine down by 600 %.

4.0 Proposed Approach — The Virtual Multiprocessor (VMP)

We have discussed the problems encountered in achieving high performance on a uniprocessor CPU. These CPUs are able to decode and issue one instruction per major machine cycle, and have enough power in the supporting execution hardware to process about four to ten times as many instructions as they can issue. However, when confronted with an on-line workload, they complete about one instruction every four major machine cycles. This is far short of their one instruction per cycle issue rate, and the net utilization of their execution units is plainly poor — often only about 4 %.

What is needed is an architecture where the CPU can process various types of workload including on-line work, and still get close to the target of completing one instruction per major machine cycle, without requiring engineering overkill in the execution units. Their total power should more nearly approximate to the effective instruction throughput rate. Likewise the extensive instruction pre-fetch of machines such as the IBM 3033, with its complex logic when conditional branches and interrupts are encountererd, entails hardware complexity and cost which would be better avoided.

A better approach is to accept the fact that each individual instruction takes a relatively long time to complete, and achieve the required throughput by executing instructions from many independent programs concurrently. To do this, the computing system has to have enough resources to execute several programs simultaneously. It must emulate multiple independent conventional CPUs.

The benefit of this approach is that all the instructions being worked on at any moment will belong to separate programs. No effort need be invested in controlling the sequence in which they execute. If one particular program is held up by, for example, a cache fault, the execution units can proceed with the processing of the other programs, and so keep themselves well utilized, and overall system throughput high. Similarly, when a program encounters a conditional branch or interrupt, only it will be affected. None of the other programs need be delayed while this conditon is being resolved. The proposed system need have less execution unit capacity than current high-performance uniprocessors, but these units can achieve far better utilization, and greater throughput should result.

Another benefit can be gained by executing several programs concurrently. Individual programs tend to concentrate on only a subset of instruction types, such as floating point arithmetic, or character manipulation. This results in a subset of the execution logic being heavily used while the rest idles. This problem can be avoided by running a mix of program types on the machine.

The proposed approach can be likened to the IBM VM/370 operating suystem which simulates many independent compuyters on one uniprocessor, so that each user appears to have his own system. We propose that the CPU be multiprogrammed at the sub-instruction level, to emulate multiple virtual CPUs. Hence the name of the system — the Virtual Multiprocessor, or VMP.

4.1 General Operation of the VMP

VMP consists of several interconnected components, each performing specialized functions. The system emulates multiple CPUs, each of which executes a separate program one instruction at a time. The instructions undergoing execution move from one component of VMP to the next, obtaining the services they need to complete execution. As each instruction completes, it triggers the fetch, decode and issue of the next instruction of its program. No attempt is made to pre-fetch instructions or operands for any of the virtual CPUs. None of the functional components of VMP is dedicated to any particular program. They are shared by all programs on a demand basis.

The idea of multiprogramming a CPU at the sub-instruction level is similar to the principle of multiprogramming a byte multiplexor I/O channel, as is common on many computers. The channel executes many channel programs concurrently, one for each active I/O device attached to it. It multiplexes itself from one channel program to the next to meet the demands of the devices, resuming channel commands (instructions) partway through their execution. VMP follows the same principles, except that it contains multiple functional units, all of which may be busy on different programs.

The functional components that VMP requires are:

- Instruction Fetch, Decode and Issue unit
- Integer Arithmetic and Logic unit
- Floating Point Arithmetic unit
- Store-to-Store unit, for instructions with multi-word operands
- Main Store Control unit
- I/O Channels

When an instruction is decoded, its opcode is converted to a string of bits, and an instruction parcel is issued by the Instruction Fetch and Decode unit. This parcel contains:

- a series of routing and control bits — the routing bits determine the path of the parcel through the network, and the control bits determine the operation carried out by each component that the parcel passes through
- a tag to identify which virtual CPU the parcel belongs to
- the program interrupt mask and condition code bits of the associated virtual CPU, since most components read or modify these
- the rest of the parcel, which changes as it gets operated on, e.g.:
  - — in the initial parcel issued by Instruction Fetch, the instruction minus its opcode
  - — after address arithmetic, a storage address
  - — after Main Store Control operand fetch, an operand

Every instruction parcel issued progresses independently through the network of components in the sequence determined by its routing bits. The components operate on the parcels to carry out the services selected by their control bits. The parcels are modified by each component before emerging. Routing and control bits are stripped off as they are acted upon, to expose the remaining routing and control bits. Once an instruction has received all the services it needs to complete, its routing bits will take it back to the Instruction Fetch and Decode unit, so the next instruction of that program can be issued.

A component may modify the routing and control bits of an instruction parcel to alter its pre-planned routing. For example, a Floating Add instruction obtaining its operand from Main Store Control may cause a storage fetch violation. In this case Main Store Control would reroute the parcel back to Instruction Fetch, with control bits reset to direct a program check interrupt, whereas the original routing would have taken the parcel to the Floating Point unit.

4.2 Registers to Support Virtual CPUs

VMP must provide a distinct set of control and data registers for each virtual processor that it supports. For example, to support the System/370 architecture:

- a Program Status Word (PSW) — instruction address and state flags
- an Instruction Buffer
- 16 General Purpose Registers
- 4 Floating Point Registers

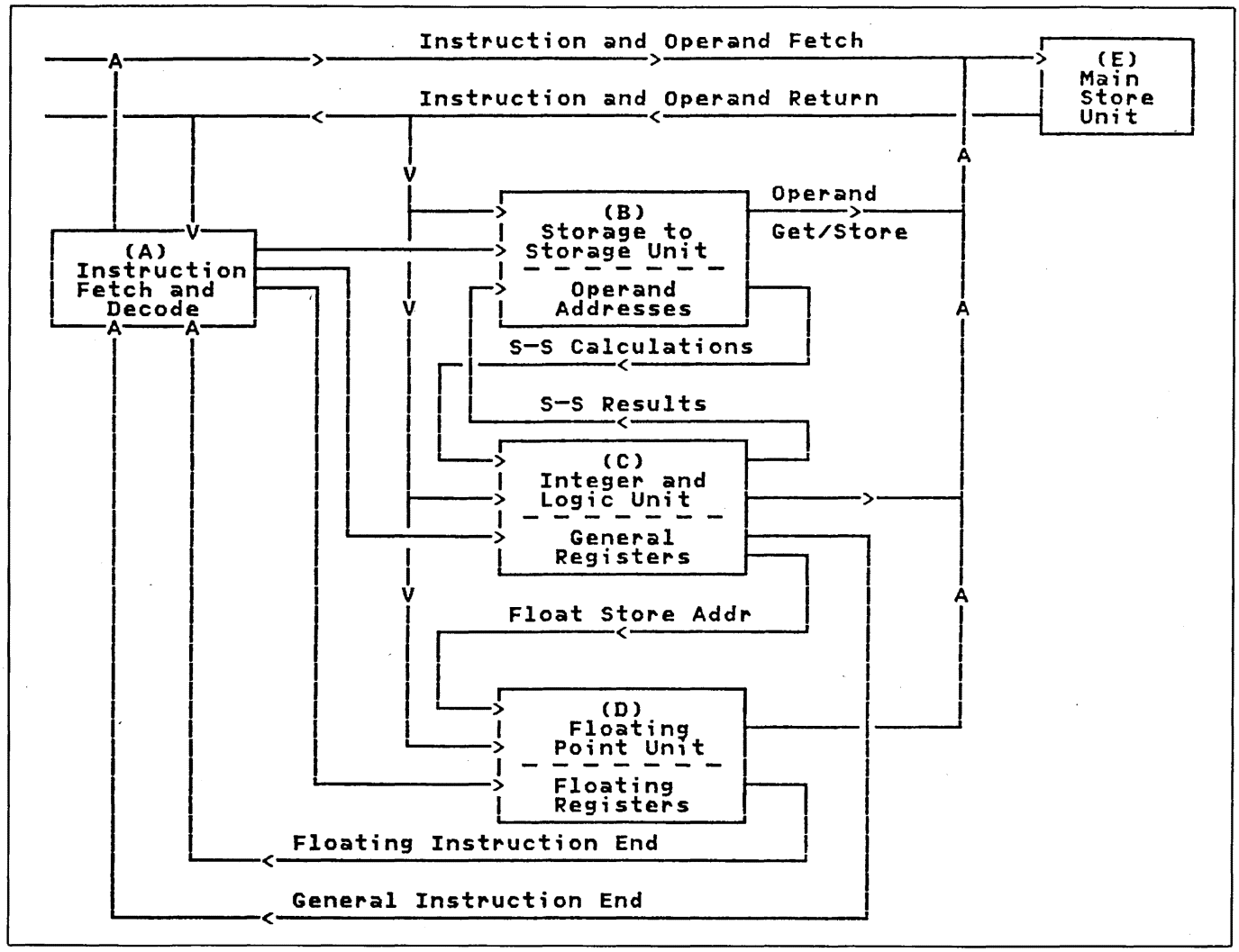

FIGURE 1. Organization of major VMP components

- 16 Control Registers
- 2 Store-to-Store Operand Address Registers
- 1 Store-to-Store Operand Length Register
- 1 Word buffer for Store-to-Store Operands

The concept of VMP is that no major component can accept more than one parcel per cycle, and that those components which are heavily used such as main store control, and instruction fetch and decode, should be able to accept a new parcel every machine cycle. Should they require longer than one cycle to service a request, pipelining and storage interleaving is assumed. Since each component has multiple possible predecessors, it may be presented with more than one parcel per cycle. It will only be able to accept one, so the other parcels will have to wait. This implies that each major component may have to buffer parcels it has completed, before being able to issue them. The components will be able to continuje accepting and processing new parcels as long as their output buffers are not all full, or committed to work in progress in their pipelines.

Each instruction undergoing execution includes a tag to identify the virtual CPU that is "executing" it. This tag is used to select the set of registers which correspondend to that virtual CPU.

4.3 Organization of the Major Components of VMP

A possible layout of the major components of VMP and their connections is shown in Figure 1. Each major component contains the logic to provide some instruction service, and the registers associated with that service, e.g.:

- The Instruction Fetch and Decode unit includes the Instruction Address Register of each virtual CPU
- The Floating Point arithmetic unit includes the Floating Point registers of each virtual CPU
- The Integer and Logic unit performs integer arithmetic, address arithmetic, logical operations and general instructions, and includes the General Registers of each virtual CPU, plus an operand buffer
- The Store-to-Store unit manages instructions with multiple-word operands, and includes Operand Address and Length Registers for each virtual CPU, which it updates as the instruction proceeds
- The Main Control Unit manages main store and the high speed cache store

These major components are interconnected by data paths which allow the movement of instructions from one component to another so that each instruction can get the services that it needs. For some operations the amount of data in a parcel may exceed the width of the data paths of VMP, for example a store operation with an operand and its address. When this occurs, transmission of the instruction parcel takes place over two or more consecutive cycles.

4.4 Execution of Store-to-Store Instructions

Many architectures include instructions which process multi-word operands, such as multiple register loads and stores, and character vector moves and compares. These are handled in VMP by the Store-to-Store unit (SSU). This unit uses the Integer and Logic ujnit to calculate operand starting addresses, then uses the same unit iteratively to process the multi-word operands one word at a time.

The SSU reconstitutes the routing and control bits of the instruction parcel on each iteration to ensure that the parcel keeps recirculating through the functional units till its operands have been completely processed. It contains two operand address registers and a length register for each virtual CPU, and an ad-

der to update them as the operands are processed. The Integer and Logic unit contains an operand buffer for each virtual CPU to support the processing of multi-word operands.

5.0 Performance Constraints and Proposed Solutions for VMP

We have seen in Section 1 that current high performance computers execute on average only one instruction every three to six machine cycles, despite large investments in circuitry to try to speed them up. The performance goal for VMP is to approach one instruction completion per cycle. If we can get within 80 % of this goal, then with the 26 nanosceond (ns) cycle of an IBM 3083, VMP would process 30 million instructions per second (MIPS), and with the 12,5 ns cycle of CRAY-1, 64 MIPS. The goal imposes constraints on the design of VMP which are investigated below.

5.1 Major Cycle Time

The design engineer has a certain amount of freedom in choosing the duration of the major cycle of a computer. The longer the cycle, the more things can be done during it, and hence fewer cycles are needed to complete an instruction. The shorter the cycle, the less can be done during each, and hence more are needed to complete an instruction. For example, Amdahl chose to use two machine cycles on the 470V/7 to fetch and decode an instruction, while the contemporary IBM 3033U does the same job in one cycle which is twice as long, but the overall throughput of the two machines is very similar. Thus the duration of the cycle does not have the immediate impact on machine speed that one might expect.

This factor is very important in the design of VMP. Since it cannot complete more than one instruction per major cycle, the cycle time of VMP should be designed to a low value. This will have the consequence that less work can be completed during each cycle, and so each instruction will need more cycles to complete. This effect can be allowed for by increasing the depth of multiprogramming on VMP — i.e. by increasing the number of virtual processors emulated.

For example, with the 57 ns cycle time of the 3033U, VMP could not execute more than 17,5 MIPS, while with the 29 ns cycle time of the 470V/7 it could achieve almost double this throughput. This despite the fact that the speed of the underlying technologies of these two machines is similar.

5.2 Connection of VMP Components

The components of VMP cannot be connected via a common bus. Although this leads to a very simple design, the average instruction would need to be transported three to four times. Each transport would take a full cycle of bus time, so overall VMP throughput would be limited to one instruction per three or four cycles, which is far below the performance target. For this reason, separate bussing is needed between the components which communicate with each other.

5.3 Implementation of Register Store

The simplest way of implementing registers in VMP would be as a single high speed storage array. This would however significantly limit the performance of the machine. Each instruction requires access to the instruction counter for readout and update, and most will also require access to registers for address arithmetic, and operands. A single storage array would be able to service only one such request per cycle, and so would limit throughout to one instruction every three to five cycles.

The approach proposed for VMP is to segregate the registers by type, and incorporate each type within the functional unit which references it most frequently. This would have several advantages:

- Different register types could be accessed simultaneously by the various functional units which contain them
- Transmission of data between functional units would be minimized, thus reducing the contention for data paths
- Functional Units usually operate on an internal cycle which is much faster than that of the machine as a whole [1]. Registers integrated within functional units would be physically adjacent to the circuits which access them, and could be accessed within the minor cycle of the functional unit.

5.4 VMP Cache Store Management

The speed of the cache store in current high-performance computers is crucial to their performance. The cache must respond to requests in about two major machine cycles, or machine throughout is badly impacted. As machines get bigger and faster in the future it will become increasingly difficult and expensive to maintain this level of performance.

With the VMP design the speed of cache store is much less critical. The cache must be able to accept a new request on each machine cycle. This will be achieved by pipe-lining its logic. It need however only respond several cycles later. This will slow the execution of individual programs, but overall throughput can be kept high by increasing the depth of multiprogramming.

If VMP were provided with a single cache store for both instruction and data references then this woujld impose a performance bottle-neck, as illustrated by the following assumptions taken from [1]

- eight byte storage doubleword
- average instruction 0,45 doublewords long
- 80 % of instructions reference an operand in Main Store

Then for each instruction executed there will be 0,45 Main Store references for instruction fetch, and 0,8 for operand reference, totalling 1,25. A single cache store unit could handle only one storage reference per cycle which would limit the instruction execution rate to 0,8 instructions per cycle.

This limitation could be overcme by introducing a second cache store to operate in parallel with the first — for example, a small cache for instruction fetch. If this secondary cache satisfied only 60 % of the instruction fetch requests, then the reference rate to the primary cache would drop below one per machine cycle, eliminating the bottle-nbeck. John von Neumann notwithstanding, stores into the instruction stream are uncommon on modern computers, and indeed illegal on many. The secondary cache may require a mechanism to detect such occurrances, but it need not be particularly elaborate.

In keeping with the philosophy of VMP, the cache store should not pre-fetch data from store as most current machines do, but only fetch on demand. As Belady has shown, this would greatly reduce the amount of data fetched from main store [2].

5.5 VMP Main Store Management

The Main Store Control Unit for VMP can follow a conventional design of multiple independent interleaved storage units which operate in parallel. the speed of main store is nowhere near as critical in the VMP design as it is for current high-speed machines. A cache fault on a current machine brings it to its knees. Almost everything stops until the needed data has been fetched from main store. With the VMP design however, an individual program may suffer a cache fault and consequent delay with no impact at all on any of the other programs. They can keep on running, and hence keep the functional units of VMP busy and its throughput high. This factor is important with the technology available today, and will become more so in the future as main store becomes slower relative to the execution units that it serves.

5.6 Number of Virtual CPUs Emulated

The number of CPUs that VMP emulates determines the depth of multiprogramming that its components support. This number should somewhat exceed the number of major machine cycles that the average instruction takes to complete in VMP. For a machine using components similar to those of a

System/360 Model 195 in speed, ten virtual CPUs should suffice.

6.0 Software Support of VMP

Most operating systems for current large scale general purpose computers support multiprocessor configurations. To them, VMP will appear to be just another multiprocessor. The depth of multiprocessing required to fully exploit the VMP architecture may however exceed the targets for which these operating systems were designed. Performance bottle-necks may arise because of this.

6.1 Accounting for Machine Time

There is a very strong requirement for computers to account for the time that they spend executing problem programs. Since VMP executes many problem programs simultaneously, and they compete with one another for resources, the amount of time that they spend to execute a fixed amount of work will vary each time that they run.

To provide an invariant measure of CPU work done, VMP could count the number of instructions executed by each program. If desired, different weightings could be attached to different types of instruction to account for their relative speeds. Alternatively, VMP could calculate the path length executed by each program as it runs. This requires less work than counting instructions, since VMP need only do arithmetic wen a program branched or was interrupted.

References

[1] "The IBM System/360 Model 91 : Machine Philosophy", IBM Journal of R & D 11 : No 1 (Jan 1967).

[2] A.J. Belady, "Virtual Storage Systems", IBM Systems Journal Vol 13 No 3 (Jan 1967).

[3] M.J. Flynn et al, "A Multiple Instruction Stream Processor with Shared Resources", Parallel Processor Systems, Technologies and Applications.

[4] M.J. Flynn, "Toward More Efficient Computer Organizations", Spring Joint Computer Conference 1972.

[5] T. Henkel, "IBM Main Frames and the Plug Compatibles", Computerworld 13 July 1981.

[6] IBM Corporation, "3033 Functional Characteristics", Form Number GA22-7060.

[7] R.M. Russel, "The CRAY-1 Computer System", Communications of the ACM Vol 21 No. 1 January 1978.

Notes for Contributors

The purpose of this Journal will be to publish original papers in any field of computing. Papers submitted may be research articles, review articles, exploratory articles of general interest to readers of the Journal. The preferred languages of the Journal will be the congress languages of IFIP although papers in other languages will not be precluded.

Manuscripts should be submitted in triplicate to: Prof. G. Wiechers at:

Department of Computer Science
University of South Africa
P.O. Box 392
Pretoria 0001
South Africa

Form of manuscript

Manuscripts should be in double-space typing on one side only of sheets of A4 size with wide margins. The original ribbon copy of the typed manuscript should be submitted. Authors should write concisely.

The first page should include the article title (which should be brief), the author's name, and the affiliation and address. Each paper must be accompanied by a summary of less than 200 words which will be printed immediately below the title at the beginning of the paper, together with an appropriate key word list and a list of relevant Computing Review categories.

Tables and figures

Illustrations and tables should not be included in the text, although the author should indicate the desired location of each in the printed text. Tables should be typed on separate sheets and should be numbered consecutively and titled.

Illustrations should also be supplied on separate sheets, and each should be clearly identified on the back in pencil with the Author's name and figure number. Original line drawings (not photoprints) should be submitted and should include all relevant details. Drawings, etc., should be submitted and should include all relevant details. Drawings, etc., should be about twice the final size required and lettering must be clear and "open" and sufficiently large to permit the necessary reduction of size in block-making.

Where photographs are submitted, glossy bromide prints are required. If words or numbers are to appear on a photograph, two prints should be sent, the lettering being clearly indicated on one print only. Computer programs or output should be given on clear original printouts and preferably not on lined paper so that they can be reproduced photographically.

Figure legends should be typed on a separate sheet and placed at the end of the manuscript.

Symbols

Mathematical and other symbols may be either handwritten or typewritten. Greek letters and unusual symbols should be identified in the margin. Distinction should be made between capital and lower case letters between the letter O and zero; between the letter l, the number one and prime; between K and kappa.

References

References should be listed at the end of the manuscript in alphabetical order of author's name, and cited in the text by number in square brackets. Journal references should be arranged thus:

1. ASHCROFT, E. and MANNA, Z. (1972). The Translation of 'GOTO' Programs to 'WHILE' Programs, in *Proceedings of IFIP Congress 71,* North-Holland, Amsterdam, 250-255.
2. BÖHM, C. and JACOPINI, G. (1966). Flow Diagrams, Turing Machines and Languages with only Two Formation Rules, *Comm. ACM,* **9**, 366-371.
3. GINSBURG, S. (1966). *Mathematical Theory of context-free Languages,* McGraw Hill, New York.

Proofs and reprints

Galley proofs will be sent to the author to ensure that the papers have been correctly set up in type and not for the addition of new material or amendment of texts. Excessive alterations may have to be disallowed or the cost charged against the author. Corrected galley proofs, together with the original typescript, must be returned to the editor within three days to minimize the risk of the author's contribution having to be held over to a later issue.

Fifty reprints of each article will be supplied free of charge. Additional copies may be purchased on a reprint order form which will accompany the proofs.

Only original papers will be accepted, and copyright in published papers will be vested in the publisher.

Letters

A section of "Letters to the Editor" (each limited to about 500 words) will provide a forum for discussion of recent problems.

Hierdie notas is ook in Afrikaans verkrygbaar.

Quaestiones Informaticae

Contents/Inhoud

*Presented at the Third South African Computer Symposium held on 14th to 16th September, 1983.

LITHO