THE FAST PROCESSING OF A SINGLE STREAM OF INSTRUCTIONS

BY

Gilles Garon

Submitted to the School of Graduate Studies in partial fulfilment of the requirements for the degree

of

Master of Applied Science

Department of Electrical Engineering

Faculty of Science and Engineering

University of Ottawa

Ottawa, Ontario

April, 1974

ABSTRACT

In this thesis we investigate the major factors which affect the execution of a single stream of instructions, and develop a multiprocessor architecture to speed-up such an execution. The proposed processor organization integrates the concepts of micromultiprocessing and micro-parallel execution of instructions. This integration, together with an inverse Polish type language, allows a number of closely coupled processors to work concurrently on the execution of a single stream of instructions. After a brief survey of existing fast processing techniques, the thesis develops in detail a basic organization using two sub-processors and describes the microprogram control scheme required to supervise their interaction. The resulting system organization allows hardware detection and automatic execution of parallelism at the instruction level. These basic principles are finally extended to more advanced processors organizations.

ACKNOWLEDGEMENTS

I am most grateful to my supervisor Professor M. Krieger for his guidance and his friendly encouragement throughout the course of this research. I would like to express sincere thanks to Mr. P. DesMarais for the many useful discussions and comments.

I wish to acknowledge the partial financial assistance of IBM Corp. of Canada.

I also wish to extend my very special thanks to Miss S. Zabrzeski for her patience in typing this thesis.

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

Today one of the greatest problems in the area of computer design is to increase the throughput of a machine at relatively low cost. The major factors which affect the throughput are the speed at which the processor performs and the speed at which data can be moved about in the machine (I/O, memory access time etc.).

In a computer, the speed at which the central processor performs can be increased up to a certain point by using more complex logic. As an example, in adders, the speed of propagation of the carry can be increased by bridging methods. On the other hand, the use of more complex logic is generally costly and goes to a point of diminishing the returns of the different operations.

The speed of all computer functions can also be improved by increasing the component speed, but component speed is limited by the state of the art.

Consequently, further increase in computing speed must be achieved by organizational innovations.

The speed at which data can be moved about in a computer mainly depends on the efficiency at which the operating system (or supervisory programs) can direct the flow within the computer. The main solutions to this problem would be to rewrite more efficient operating systems,

have more control programs at microprogram level, and/or reduce the time to run such operating systems by increasing the speed at which the processor performs.

The subject of interest in this thesis is the increase of processing speed through specific processor architectures. In computer design, specific architectural innovations are often used to increase the physical speed limitation of different units. For example, an interleaved Random Access Memory (RAM) will mask the physical speed limitation in accessing the memory by dividing the RAM into sections and thus permit almost simultaneous access to it.

The design of a machine having a better hardware/software relation permits a more efficient processing, and a better sequence of utilization of the resources can then be achieved. For example, a stack-oriented machine allows an orderly and efficient hardware/software relation between the inverse Polish notation (which permits an effective translation of arithmetic expressions to machine language) and the processor operation (storing of partial results, interrupts, etc.).

An increase in processing may also be achieved by ordering the sequence of execution of algorithms. Techniques such as look-ahead, micromultiprocessing and/or parallel processing at different levels permit a better utilization of the resources and decreases the processing

time of a single stream of instructions.

It is the author's belief that the use of a single central processor to do everything in a computer is more than outdated. Special processors are used today for I/O, memory management, etc. The present trends are the incorporation of specialised units into a system (such as I/O processors, processors for data processing, processors for fast arithmetic processing, associative memory processors, etc.) in order to increase the throughput of a machine. Therefore, in this thesis, we will be interested in the design of a specialised unit, a multiprocessor for the fast processing of a single stream of instructions (arithmetic oriented). This will be achieved by:

1. The utilization of a high level machine language which permits a closer hardware/software relation and reduces the processing time of a program (i.e. compiling, assembling etc.), before the actual execution.
2. The design of a processor which will take advantage of its machine language.
3. The implementation of an organization which masks the speed limitation of its different elements.
4. A close communication in a multiprocessor system (microprogram level) which eases the implementation of techniques such as: micromultiprocessing, detection and parallel execution of instructions without having to

analyse the dependencies (operational, procedural, and data dependencies) in a single stream of instructions.

5. The design of a processor that requires smaller overhead by reducing operating system control and also permits a full utilization of the resources.

In Chapter II we shall present a brief survey of fast processing techniques and introduce the background on the research topic. Chapter III gives a frame of work for the design of a fast processor. The design of a two sub-processor unit is presented. In Chapter IV we introduce another technique to detect micro-parallelism and suggest more complex sub-processor organizations. We then conclude this thesis by suggesting further topics of research. Appendices I and II give further details on the modified Polish notation and on the language used. Appendix III shows the relative time of execution used for the micro steps. Finally, Appendix IV gives the listing and the results of a program which was used to evaluate the efficiency of the two sub-processor organization.

CHAPTER II

SURVEY OF FAST PROCESSING TECHNIQUES

The purpose of this chapter is to review some of the major existing techniques (used or still on paper) which permit fast processing.

Originally the main direction to increase the speed of a processor was to include additional hardware into the different units of the processor. Lately the direction of research is to provide speed by different processor organizations, multiprocessing techniques and the use of specialised units.

In this chapter, first the addition of hardware is presented. Next the memory architecture required for a better use of the processor is introduced. Then an orderly presentation of the different processor architectures is given. Finally other processing techniques used to increase the processing speed of a single flow of instructions are briefly described. Note that, in this chapter, we only discuss principles which affect directly the processing speed. Other ideas such as virtual memory, microprogramming, etc., which affect indirectly the processing speed are beyond the scope of this thesis.

II-1 SIMPLE EXTENSIONS (ADDITION OF HARDWARE)

The purpose of this section is to present some of the speed-up techniques that are obtained by simple addition of hardware in the different parts of the computer. Only the principles of these techniques are presented; more detailed information can be obtained from the mentioned references.

1-1 Addition of registers

In a processor, one of the most consuming aspects of instruction execution is the cycle time of the memory references. A straight-forward approach to reduce the number of memory references is to add a small number of additional registers to the processor. These registers are used to store instructions, data, or partial results which are frequently used in the program. The user (or assembler) has to implement the program to introduce operations between the registers as much as possible. Clearly this improvement is program dependent and additional software expenditures are required to take full advantage of the system. There are many examples of multiregister machines [1].

1-2 Multiaddressing

Multiaddressing consists of having more than one address per instruction thus reducing the number of instruction fetches in a program. Multiaddressing may be done in terms

of memory locations or in terms of registers in a multiregister machine. For example, an instruction might contain the two addresses of the operands and the address where the result is to be stored. In such three-address machines, two instruction fetches are avoided when compared to single address.

Memory multiaddress configurations are usually impractical since a long word is needed to specify multiaddress in a RAM. Register multiaddressing is usually more practical since it addresses only a small number of hardware registers and indirect or relative memory multiaddressing can then be done.

A particular example of a two-address-per-instruction machine is the Univac 1108A. This organization permits memory multiaddressing and the word length required is 36 bits, six bits for OP-Code, 15 bits for the first execution address, and 15 bits for the second execution address [1].

1-3 Instruction look-ahead.

Another approach to reduce the number of fetch cycles to the main RAM in a multiregister machine is the instruction look-ahead technique. With a sufficiently large number of high-speed registers, a short sequence of instructions may be stored in the Processor Unit (PU). This enables the capture of short loops in the program which can then be executed at register speed and therefore reduce the number of memory references. By overlapping

the fetch cycle and the execute cycle of instructions with a look-ahead unit, the time consumed for memory reference may be cut in half [2].

An alternate architecture permitting instruction look-ahead is the use of a fast buffer memory (or cache memory) instead of registers. This reduces the cost of a large number of look-ahead registers and the same saving is achieved if block transfer is used.

A particular example of a computer with a look-ahead unit is the Stretch computer [1].

1-4 Overlapping of instructions

When a fast fetching element is included into a computer, the actual execution of certain instructions becomes more time consuming than the instruction and data fetch operations. By permitting an overlap in the execution phase of several instructions, a saving in processing speed can then be achieved. This, in turn, complicates the role of the control unit which has to determine the independent micro-steps of consecutive instructions that can be overlapped. However, execution overlap can be facilitated by programming the computer such that many consecutive instructions are independent. Today in most organizations, such programming effort is left to the user (or the compiler) or is done at the expense of complicated hardware.

An example of a computer using this technique is the Stretch computer [1].

II-2 MEMORY ARCHITECTURE (for fast processing)

In today's computer the limitations of the main RAM are its access time and throughput. They may be masked by: i) providing an intermediate storage between the processor and the main memory. ii) organizing the actual RAM to allow multiple access. There are many different types of memories and generally their structure affects the memory to processor organization and hence the processor architecture. Therefore, before presenting the different types of processor architecture, this section introduces the major memory structures which affect directly the speed of the processor.

2-1 The Buffer (or Cache) memory

A Buffer memory is usually referred to as a high speed memory which is not of sufficient size to satisfy the RAM requirements of a system. In fact the Buffer memory is

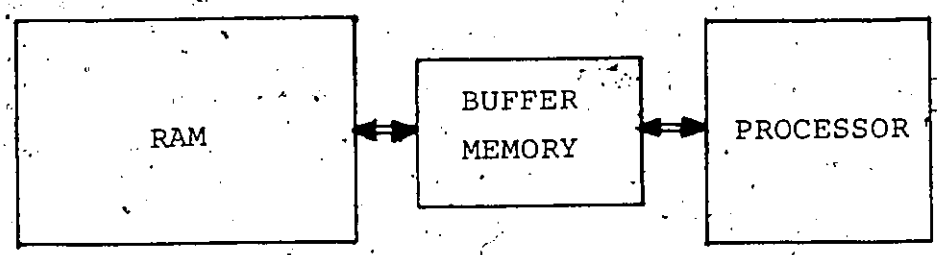


Fig: 2.1 The Buffer Memory

a small memory between the main RAM and the processor which is used to mask the RAM access time to the processor (Fig 2.1). In a system with this type of memory, the programmer or the system programs tries to address the buffer instead of the main RAM in order to save memory access time. The major problems of such a memory is the data transfer between RAM and Buffer memory and the choice of the size of the memory. There exists a strong controversy about the optimal size of such a memory. A large Buffer memory will capture a large part of a program but requires the transfer of large data blocks and high cost of memory. A small buffer avoids the transfer of large data blocks but it will capture less loops in a program. The choice of the size of such a memory is program dependent and depends on the particular application of the system.

A particular example of a buffer (cache) memory is in the IBM 360/85 [3]. In this computer the cache memory provides sufficient storage so that the processor might characteristically behave as if it had 80ns. main storage. An automatic machine algorithm controls the contents of the store dynamically. The store, called buffer or cache memory, is organized into sectors of 1024 bytes, where each sector is, in turn, divided into 16 blocks of 64 bytes each. Each sector is associated with a 1024-byte section in main memory through a sector address register. The system dynamically associates (controlled through an algorithm) the cache sectors with the main store as main memory references are needed by the programs.

2-2 The Scratch-Pad (or working) memory

The S-P memory is a fast small memory which is used as a working place in order to reduce the number of memory references to the main RAM (fig. 2.2). The user (or assembler) has to implement his program to introduce references to the S-P memory as much as possible. Here, as in 2-1, the problem of data transfer between the main RAM and the S-P memory arises. It should be noted that in some organizations the Buffer (or cache) memory is also used as S-P memory.

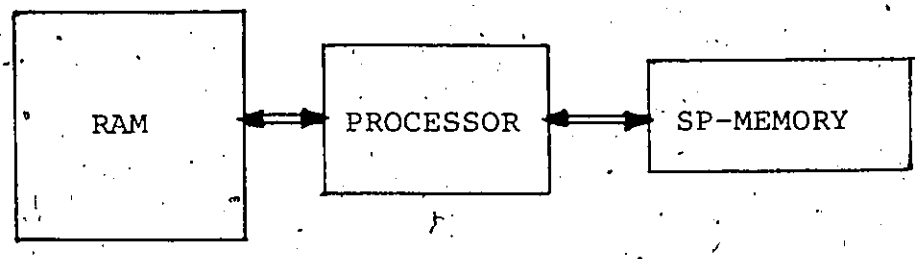


Fig 2-2 The S-P Memory

2-3 Associative Buffer and S-P Memory

All the problems of data transfer in the Buffer and S-P memory may be solved by making these transfers transparent to the programmer, that is organising the memory as associative memory. This can be achieved by keeping track of the history access of the Buffer and S-P memory; words which have not been recently accessed may be replaced automatically. Studies show that, in such organizations, up to 95% of the words requested by the processor have been

found in the fast small memory [2], thus a considerable speed improvement oyer a slow main memory is achieved.

2-4 The Array Memory

An array memory consists of an array of modules (or banks) of memory that can be accessed simultaneously. Each module contains whole data words or parts (bytes) of a data word. This type of memory is mainly used in multi-processor environment such as array processors, parallel processors, etc. Fig 2.3 gives a block diagram of such a memory.

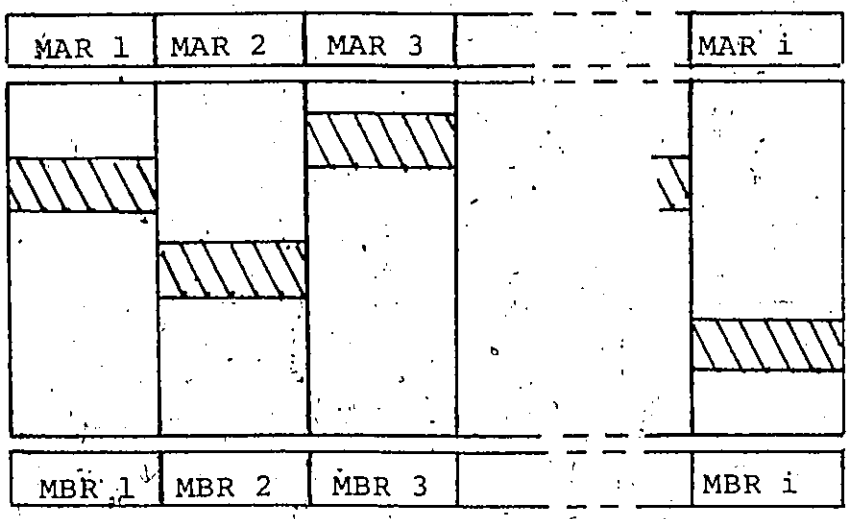


Fig. 2-3 Array memory

2-5 Interleaved Memory

A basic memory organization which does not reduce the access time of a memory but can handle several memory references almost simultaneously is the interleaved memory.

An interleaved memory is a RAM which is divided into several independent banks. (Fig. 2.4).

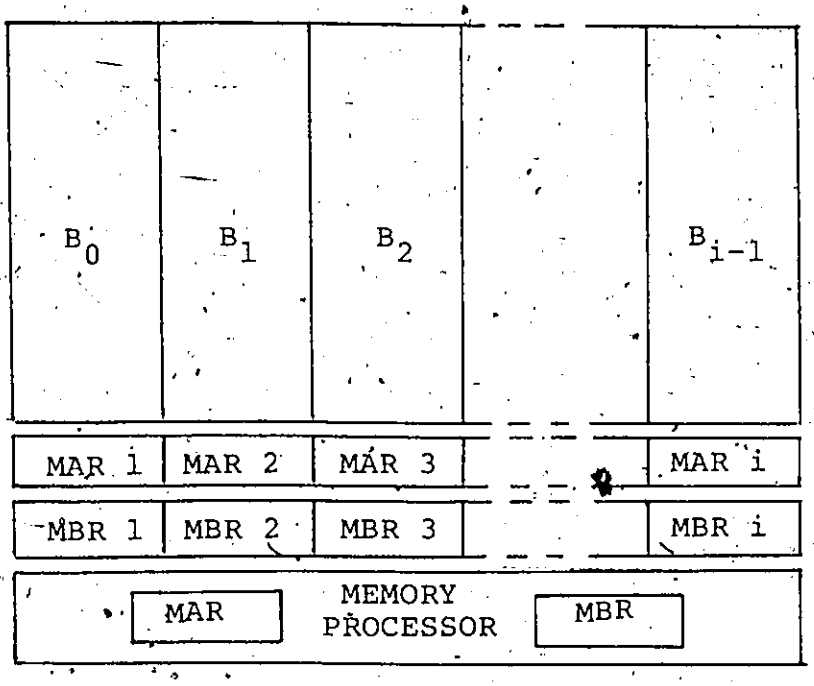


Fig. 2.4 An interleaved memory

The data (or instructions) in these banks is arranged such that the first word of a set of instructions is in B_o, the second in B_1, the i^{th} in B_{i-1} etc. By synchronizing the access time of the different banks, a much higher frequency of memory access can then be obtained. For example, assuming i banks each having a time access of 500t, a word can then be obtained at every $\frac{500}{i}t$. By having more than one MAR and MBR it is then possible to have simultaneous access to the memory (interleaved and array properties).

In today's large computer design, one of the present trends is the use of interleaved memory in order

to improve the frequency of access to the main memory. For example, the IBM 360/60 and 360/70 have two storage units which are interleaved in order to permit a faster frequency of access. Another example of a computer with interleaved memory is the CDC 6600 which has up to 32 banks of 1 μsec core storage of sixty- bit words. That particular organization also permits four memory references to be processed simultaneously.

There are many other organizations of interleaved memories; they are usually a combination of different memory architectures. An example is the interphased memory of Burroughs Corporation.

2.6 Pipelined Memory

A pipelined memory is a memory which provides a stream of words when it is addressed. The memory is composed of different modules (banks) and when one address is furnished to the memory, the words associated with this address (one in each module) are sent sequentially to the MBR. In other words, an address to the memory is the address of a super-word (a part in each module) which is pipelined through the MBR.

Fig 2.5 gives a block diagram of a pipelined memory. For a particular address, each word (or byte) associated with that address is sent to the MBR with a delay Δt equal to the processor clock cycle. This type of organization is useful in pipeline processing of a stream of instructions and data.

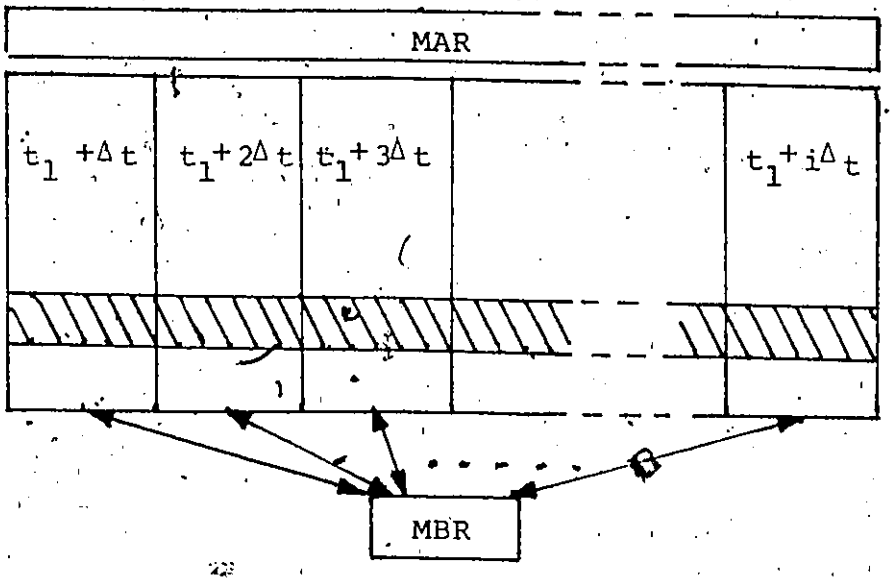


Fig 2.5 A pipelined memory

2.7 Associative memory

An associative memory is one in which a data word is not obtained by supplying an address which specifies a location into the memory. Instead, an identifying descriptor is provided to the memory which is then searched for a matching descriptor. When a match is found, the data word associated with the memory descriptor is then the desired output. The search may be sequential or parallel. A sequential search is usually slow; only the parallel search makes the memory really "associative", but a large amount of combinational logic is then needed.

The organization of the associative memory varies as many data words may be associated with the same descriptor. The descriptor may be stored with the data word or separately. Generally these memories are expensive and the present state of the art limits their size.

2-8 Associative array processor memory

An associative array processor memory is an array memory which is accessed one bit at a time for each processing element (one for each bit) of the memory [4]. Fig 2.6 gives a diagrammatic representation of an associative array processor memory.

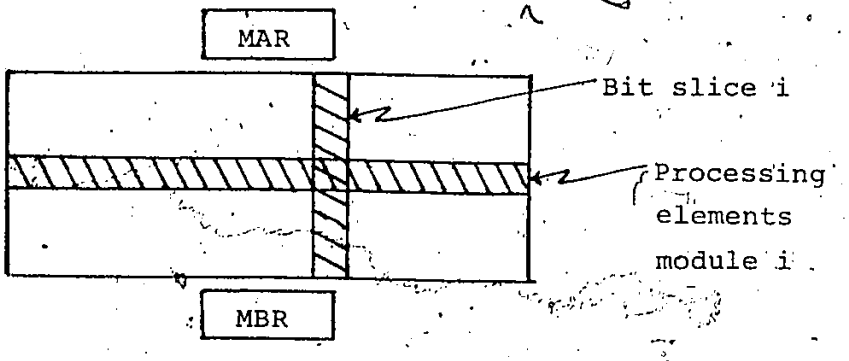


Fig. 2.6 Associative array processor memory

In this memory each bit has logic, ie. a small processing element. The memory is accessed serially and consequently is relatively slow. However, its access could be pipelined by interleaving different associative array processor memories. This type of memory is used, for example, with the associative array processor for data processing on a one bit search basis.

2.9 Orthoganal memory

An orthoganal memory is an associative array processor memory that allows both bit slicing (bit slice memory access) and conventional word access. The word access of the memory permits a more natural I/O than the associative array processor memory.

As in the associative array processor memory each bit of memory has logic. Usually the amount of logic is not very great because of the large number of bits in most memories. Consequently, in such a memory it seems reasonable to time division multiplex the logic hardware over many cells since most memories are slow compared with logic [4].

II-3 PROCESSOR ARCHITECTURE

As mentioned in the introduction, one way to improve processing speed and throughput of a processor is by defining specific processor architectures. There are a number of different ways in which one could categorize present day processors. Here we chose to categorize them in terms of data and instruction flow. Thus from the point of view of an external observer we can consider the shown block diagram and information path (Fig 2.7) between processor and memory. Both the instructions and data can be transmitted to the processor either sequentially or in parallel. In line with this, processors can be classified in four categories [5]: Single Instruction Single Data (SISD), Multiple Instruction Single Data (MISD), Single Instruction Multiple Data (SIMD) and Multiple Instruction Multiple Data (MIMD).

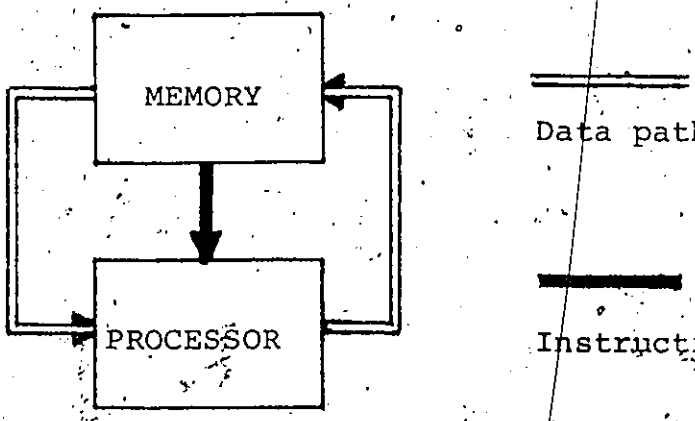


Fig. 2.7. Processor to Memory Interconnection.

Each of these is presented in some detail in the following sections.

3.1 Single Instruction Single Data (or serial) Processor

In this organization there is a single stream of instructions and a single stream of data between processor and memory (Fig 2.8). The instructions are executed sequentially in the processor and the corresponding data is requested synchronously.

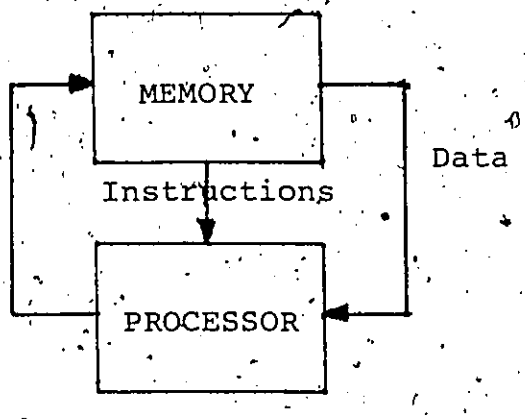
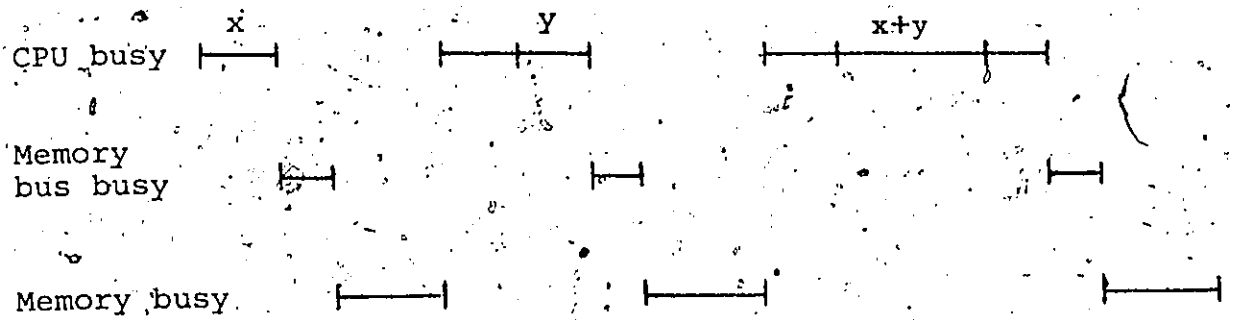


Fig 2.8 The SISD Processor

A typical timing cycle for this organization is as follows (for x = x + y):



processing and micro parallel execution may be used to increase the processing speed of a SISD. They will be discussed in the last part of this chapter.

3.1.1 Bit serial processor

At the most elementary level of serial processor, one bit of an n-bit word is operated on at a given time. There is no concurrency and even the most trivial operation of n-bits requires a time of n. This type of processor was used in the first generation of computers because the cyclic primary memory to which it was connected was fundamentally bit serial. It should be noted that bit-serial processing is today used in LSI microprocessors on a chip.

3.1.2 n-bits serial processor (word serial)

At a second level, the processor and memory could be made to operate on an n-bit parallel basis where the words (n-bits) are available one at a time. This type of processor became common in the second generation when core memories appeared on the market. The word serial processor is the most often used type of processor in present-day computers.

In some special purpose serial processors, the memory used is not homogeneous, having separate memory banks for instructions and data.

3.1.3 The stack processor

Another SISD processor organization is the stack processor. The stack processor is basically a word serial processor in which last-in-first-out stacks (storage area where the last item stored in memory is the first item taken out) are used for the storage and recall of the intermediate results in a sequence of operations. In such

processors the arithmetic is done between the top elements of a stack which is used in the ALU. When the top of the stack is hardware implemented, the instruction operands are available to the processor at register speed. This consequently improves the computing speed of a processor.

The stack organization provides many other advantages such as parameter transfer, efficient evaluation of arithmetic expressions, recursive subroutines, procedure calls, and it enables rapid context switching (establishing interrupts). The stack organization of a SISD processor also permits the implementation of machines designed to support block-structured languages. Such high level machine languages allow more efficient programming, that is they save memory locations and improve processing speed. Another advantage of the stack processor is that it uses zero-address instructions, thus allowing packing of instructions.

Most stack-oriented machines use the inverse Polish notation as code for the arithmetic unit. The notation allows a close software/hardware relationship between the stack structure and the machine language. The inverse Polish notation also permits an efficient translation of high level languages to stack machine language, thereby reducing the processing of a program before the actual execution.

Today most processor designs include some form of stack. The stacks may be implemented through the use

of fast electronic registers and/or various software methods. A particular example of a stack-oriented computer is the HP-3000 [6].

3.2 Multiple Instruction Single Data (or pipeline) processor

A pipeline processor is a processor which has a multiple instruction stream and a single data stream (Fig. 2.9). The processor consists of multiple functional units, each of which is responsible for partial processing of the data stream. The basic instructions are subdivided into a number of intermediate operations for the different stages or units. Each processor stage operates on the data stream, performing its specific part of the instruction. In other words each unit (or stage) is independent and is capable of carrying out only selected tasks on the data stream.

Multiple serial data operations are desirable

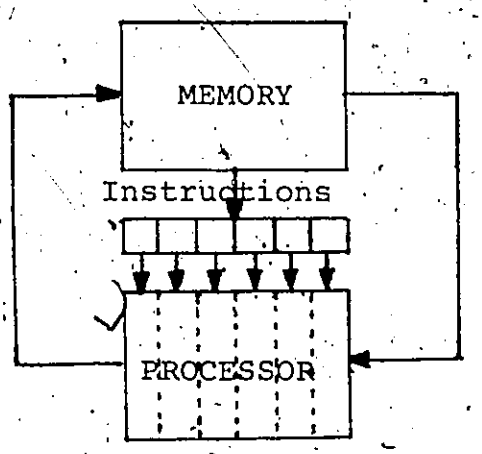


Fig. 2.9. The MISD or pipeline processor

in a pipeline processor so that several independent operations can be carried out at a time on a large number of different data which are in the pipeline. This is mainly to compensate

for the major drawback of this organization, the time Δt which it takes for a specific data to go through the pipeline, that is the time Δt required for the execution of a single complete instruction by a number of different units in the pipeline.

In a program (single stream of instructions) the relationship between the different operations often appears as a dependence of one operation on another for information referring either to the content of a given operand or to which operand should be executed next. The pipeline structure is efficient to execute programs with independent instructions and is very efficient when executing programs having many consecutive instructions specifying the same operation on different arguments. In general, for most programs, the capability to recognize various relationships between instructions is necessary in order to subdivide successive instructions in a series of operations for the different parts of the processor. Consequently an efficient sequencing of a large set of operations is required since the constraints (dependencies) usually depend on the type of operations available, the number of redundant execution units, and the execution time required by each unit for a given operation. Algorithms have been developed to dynamically sequence operations (in a pipeline machine) in order to provide a smooth execution stream while taking advantage of all available parallel executable

resources. These machine constraints are then added to the algorithms, but no algorithm will guarantee an optimal execution sequencing.

Pipeline processors seem to provide a solution for fast processing, but many problems such as sequencing of operations and determining the independence of the consecutive instructions degrade the performance of the processor to a point where such processors are only efficient for certain particular applications.

No actual system is completely designed on the pipeline principle, but the principle is widely used in present-day computers. For example, the CDC-STAR can perform the inner or dot product of two vectors as their elements pass through the pipeline [1]. Another example of the application of the pipeline principle is the overlap of the read-write operations into a memory (interleaved memory) which is becoming widely used in computer design.

3.3 Single Instruction Multiple Data (or parallel)processors

Parallel (SIMD) processors are processors which have a single instruction stream and a multiple data stream. In such processors, the same instruction is executed in a number of separate processors on different data. Fig.2.10 shows the block diagram of the SIMD processor-memory information path.

SIMD processors include array processors, associative processors, associative array processors, and orthogonal processors.

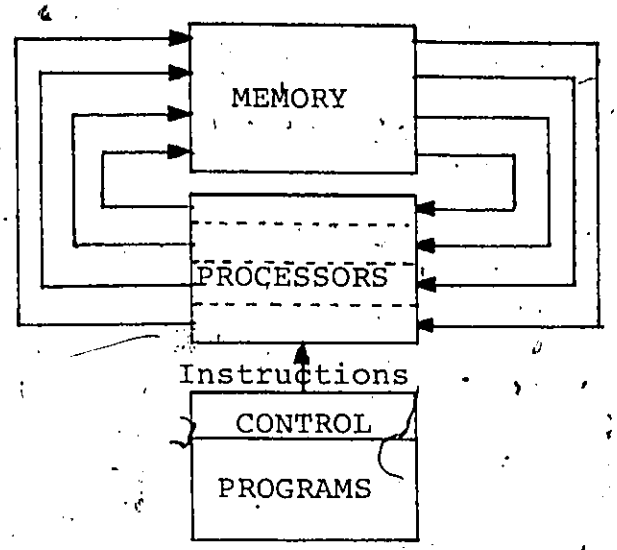


Fig 2.10 SIMD or parallel processor

3.3.1 Array processor

An array processor usually consists of a set of relatively simple serial processor elements, each having its own arithmetic logic unit, its own control unit, etc. These processor elements are usually controlled by a main control unit which does the I/O, inhibit the operation of the processing elements, and control the routing and skewing of instructions. Routing and skewing are used to move data from one processing element to another or allowing one processing element to reference another memory. In array processors, data are processed in parallel. The processors operate on a word or byte basis and usually access blocks of data words for parallel execution in the different processors.

Array processors are very useful to simplify programming. For example, matrix processing may be done

by an array of processing elements with the use of a "COMMON" statement. An example of an array processor is the ILLIAC IV Computer. The array consists of 8×8 processors each connected to its neighbour. The control unit allows a single stream of instructions to operate on many data. The local control permits each element to enable or disable the execution of the common instructions according to local tests.

3.3.2 The Associative Processor

An associative processor is one that operates on data, addressing either by tag or value rather than by memory location. This includes machines that process data in parallel while retaining their associative capability. This type of processor is very efficient for processes which require the use of associative capabilities such as searching through a large data base or the execution of independent instructions on large data stream.

3.3.3 Associative Memory Processor

An associative memory processor is a processor with the above structure but implemented with an associative memory. Types vary from array processors with associative processing element memories to associative memories with enough logic at each bit to perform all arithmetic and logic functions.

At present, because of the high cost of associative memories, these two approaches are suitable for only rather

specialised applications where high hardware efficiency is needed and where data rates dictate this type of structure.

3.3.4 Associative Array Processors

An associative array processor is an associative processor which operates on bit slices of data (ie. on a fixed bit position in a set of data words) instead of on a word basis. The type of processing element used in a basic associative array processor is composed of a 1-bit arithmetic and logic unit and a few flip-flops for control and carry holding.. In fact, an associative array processor can process all capabilities of the array processor with a difference, of course, in its associative capabilities, its addressing technique (bit sliding) and in some cases, a marked difference in speed and efficiency.

Fig 2.11shows a typical example of timing in an 8-bit associative array processor for the addition A+B.

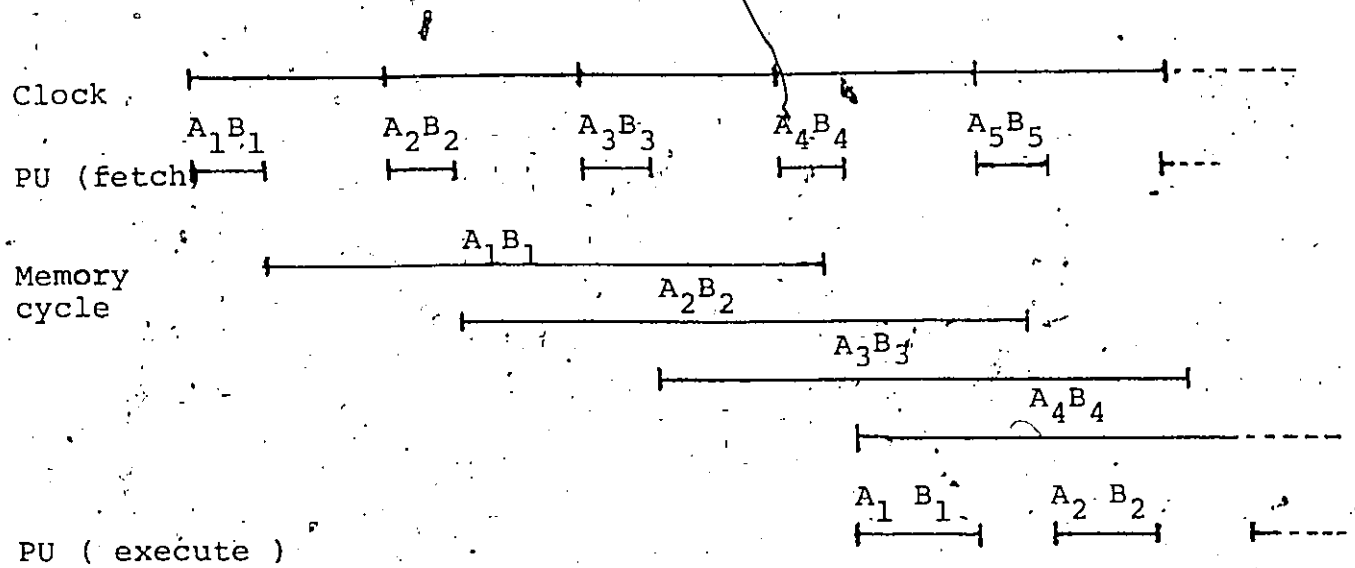


Fig. 2.11 Typical Timing of an Array Processor

It should be noted that memory references are made to successive bit slices and that there are no memory cycle conflicts. However, to increase speed on an associative array processor the designer can use interleaved memories. It should be noted that an associative array processor may be used with an orthoganal memory to allow faster I/O.

3.3.5 Orthogonal Processor

An orthoganal processor is an associative processor which allows both bit slicing and serial access (word access). It is used with an orthogonal memory and in contrast with SIMD processors (Fig.2.10), orthoganal processors have both data and programs in the same memory.

This type of organization is very efficient in programs with extensive branching because the dual-access memory allows test results to be quickly available to the control unit, thereby increasing efficiency. When looping depends on many tests, these tests can easily be performed in the processing element array rather than in the control unit.

Another advantage of this unified organization is that it makes compiling easier. Serial operations, which can be concurrent with parallel ones, can readily operate on data.

3.4 Multiple Instruction Multiple Data Processors (or multiprocessors)

A multiprocessor is a system characterised by a number of independent processors each with its own control

unit. In such an organization (Fig.2.12) m programs or subroutines are running at the same time in the system.

This type of organization is advantageous since the resources of a system (large Random Access Memory, data file, peripheral equipment etc.) which require the heaviest investments can then be shared by the processors. Such a technique of having many processors in a system increases both its reliability and availability. A failed unit only reduces the throughput of the system.

Various types of multiprocessing may be found in existing machines. Particular examples are the CDC 6400 [1] and the B7700 [7].

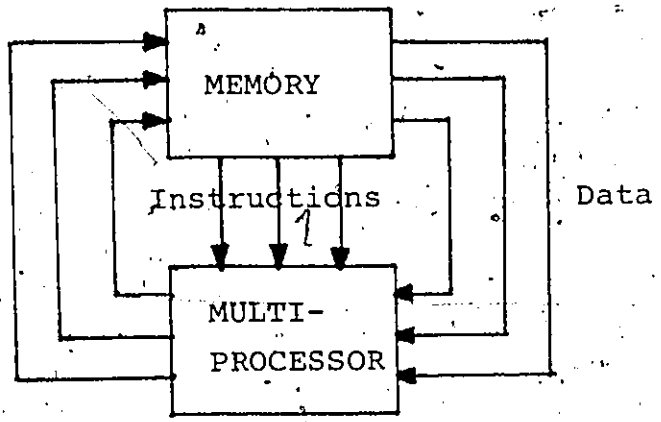


Fig.2.12 MIMD processor (or multiprocessor)

II-4. OTHER PROCESSING TECHNIQUES

In this section, we introduce briefly the major speed-up techniques which are used to improve the processing speed of a program in a multiprocessing environment. More details may be obtained from the mentioned references.

4.1 Parallel Processing

In a multiprocessor system the processing speed of a single stream of instructions can be speeded up by partitioning the job into sections which are substantially independent of one another and which may therefore be executed concurrently, that is in parallel. Such parallel execution provides a fast turn around time in the execution of a single flow of instruction. Furthermore unutilized resources of the multiprocessor can then be applied to other jobs.

There are two main types of parallelism present in a job, that is the macro-parallelism and the micro-parallelism. Macro-parallelism consists of having independent sections, procedures, or subroutines in a job being executed simultaneously by several processing units. Micro-parallelism exploits the relative independence of individual machine instructions in a sequence of consecutive instructions and/or between steps of execution of an instruction.

The implementation of macro-parallel processing is not only limited to an appropriate hardware design. An adequate performance, in such a system, is also determined

by an appropriate software overhead. Allocation of resources, scheduling of tasks, and supervisory strategies must be simplified and the related procedure (algorithm) minimized to compromise only a small portion of the total activity in the system. Another problem with macro-parallelism is the need to develop parallel processing procedures at the language level (addition of special instructions in programs to display parallelism in algorithms) and/or at the supervisory program level, that is programs determining intrinsically parallel procedures in order to take full advantage of a multiprocessing environment in the execution of a program. This problem arises mainly because of the sequential nature of commonly used numerical algorithms, data processing procedures, and general computer programs. The advent of macro-parallelism thus calls for the modification of accepted techniques to expose any inherent parallelism in a job. A major factor for an efficient execution is the interaction between the processors. For example, a close processor communication may cause specified tasks to be executed on the termination of a present instruction in one particular processor, or may cause a number of processors to quit the search for a particular item in a partitioned list when the item has been located by one processor.

An efficient exploitation of micro-parallelism in a single stream of instructions is mainly dependent on how the computer detects and executes m independent instructions out of n instructions.

One approach to parallel processing in SISD processors has been studied by TJADEN & FLYNN [8]. In their paper, they present a simple algorithm which allows simultaneous multiple decoding of a block of instructions to detect micro-parallelism. By organizing the instruction format and the decoder of a SISD processor they readily ascertain the instructions which qualify for parallel execution. Their simulation shows (assuming that the algorithm adds negligible time to the decoding process) an increased efficiency of 86% in the execution rate of a single stream of instructions. This result is based on having a decode stack of size 10 in their organization [8].

In this thesis another approach (which does not involve multiple simultaneous decoding) to the detection and parallel execution of instructions in SISD processors is presented in Chapter III. The method does not guarantee maximum parallel execution but it enables micromultiprocessing and overlapping of instructions to allow fast processing of a SISD stream.

Many other studies on parallel processing such as measurement of parallelism, definition of algorithm, and other structures to detect parallelism can be found in the reference material.

4.2 Micromultiprocessing

Micromultiprocessing is an approach to multiprocessing at the level of very small tasks [9]. A micro multipro-

cessor system consists of processing units sharing the processing of the same instruction stream, where tasks consist of individual instructions. The system configuration developed in [9] and [10] consists of two microprogrammed processing units sharing the same control unit.

There are three alternative techniques for micromultiprocessing that have been developed: leapfrog, pipeline, and graph. In the leapfrog technique each processor unit fetches and executes its instruction. The instructions of a job are executed alternatively by the processors. In the pipeline technique, one processor fetches the instructions while the other executes them. The instructions fetched are stored in an execution stack. When the execution stack is empty both processors fetch instructions; when the stack is full both units execute instructions. The third technique involves the simultaneous participation of all PU's in the execution of each instruction. Fig 2.13 shows a graphical interpretation of these techniques.

Speed up in the execution of a single stream of instructions may be achieved through the idea of micromultiprocessing. Reference [9] shows that the two-processor system gives an increase in processing rate from 1.12 to 1.37 compared to 1 for a uniprocessor system. It should be noted that a multiprocessing environment requires interaction between the processor units because of the dependent nature of the instruction sequence.

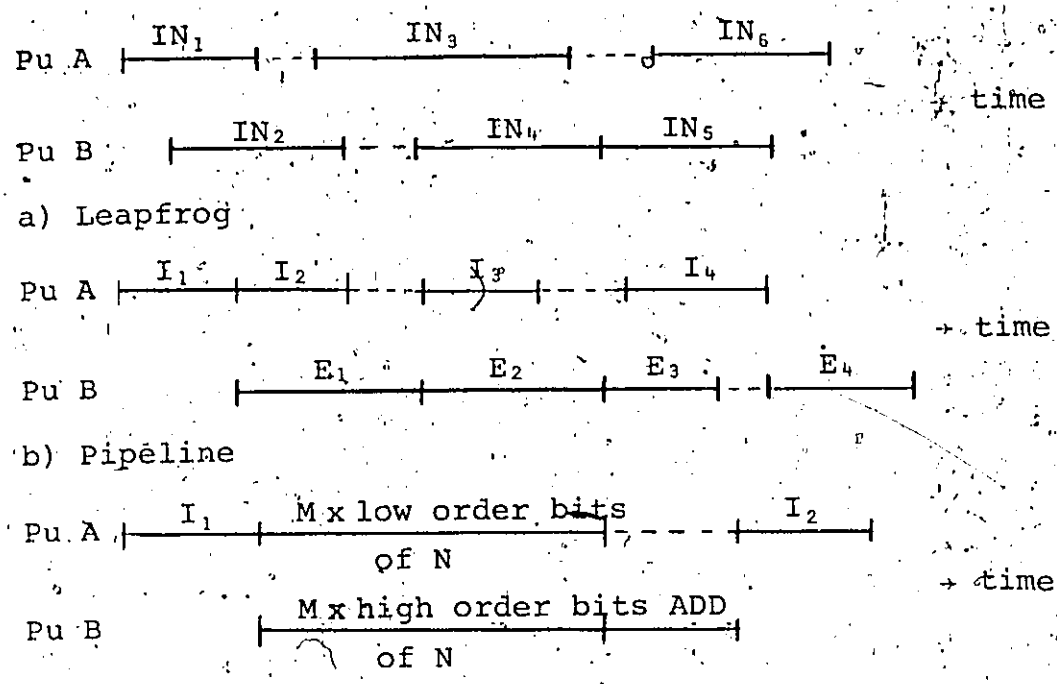


Fig. 2.13 Techniques of Micromultiprocessing.

CHAPTER-III

AN ARCHITECTURE FOR FAST SISD PROCESSING

The subject of this chapter is the design of a processor for the fast processing of a Single Instruction Single Data (SISD) stream. We will present an organization allowing two or more sub-processors to work concurrently at the execution of a single stream of instructions. The idea of micromultiprocessing and micro-parallel processing will be used efficiently by a close hardware/software interaction within the processor. The operation of the two-sub-processor unit is studied in detail and its efficiency is compared with a uniprocessor unit.

III-1 FRAME OF WORK

In today's SISD processor the memory access time, which is a main limitation to fast processing, can be masked by techniques described in Chapter II. Now, if we consider a very fast processing achieved by the collaboration of a number of sub-processors at the execution of a single stream of instructions, the required flow rate can be obtained by a proper memory organization. In this thesis we will assume that the memory can give the flow rate needed and we will organize the processor such that its cycle time becomes independent of the memory access time in order to increase the processing speed of a SISD stream.

A second limitation to fast processing is the speed at which the Arithmetic Logic Unit (ALU) can perform. By a proper distribution of consecutive instructions into a number of sub-processors, each having its own ALU, it then becomes possible to mask the physical speed limits of the different units. Such distribution of consecutive instructions is only possible in a multiprocessor environment having a close interaction (communication) between the different processing units at control level.

In the proposed organization, a particular sub-processor only has to match the speed of its elements (ALU, control, etc.), not the processing speed of the instruction stream. In fact, the processing speed of such a SISD processor

should not depend on the individual processing speed of its sub-processor elements. The idea is to have many sub-processors working concurrently, at their own speed, at the execution of a single stream of instructions. The choice of the particular processing method (micromultiprocessing or microparallel execution) depends on the interaction between successive instructions, and the sequencing of execution is done dynamically by the sub-processors' common control (SPCC). The defined structure also reduces the problem of temporary storage which occurs in most phases of programming in pipeline and parallel processors [4].

The use of a number of processor units for the processing of a single instruction stream provides an efficient solution to fast processing but requires a very strong interaction between the processors. The most common type of communication between a number of processors is done through the use of proper operating systems (or supervisory programs) which are software-oriented. In section III-3 we will introduce a microprogrammed scheme allowing interaction between processors at the control level. This will greatly reduce the software control state (system overhead) in such a system as well as increase its efficiency.

The micromultiprocessing technique was chosen because it reduces the processing time of a single stream of instructions when there is no inherent parallelism. Micromultiprocessing takes advantage of the dependencies

between consecutive instructions and has been studied in [9]. Micro-parallel execution is presently the subject of various types of research and many algorithms and structures have been proposed to detect such parallelism [11, 12]. Measurement of parallelism has been done in different types of Fortran programs [13], in arithmetic expressions [14], in the evaluation of polynomials [15], and in different types of algorithms and instruction streams [16, 17]. Furthermore, the results of these research efforts justify the design of a multiprocessor system which will detect automatically and execute microparallelism in a single instruction stream.

III-2 A PROCESSOR ORGANIZATION FOR FAST SISD PROCESSING

A major limit to the fast processing of a single stream of instructions is the cycle time of the SISD stream processor. The most efficient solution to this problem is to use many processing units to execute the instruction stream. In this section, a basic processor organization for the fast execution of a single stream of instructions is presented. The simplest structure, two sub-processors, is used to simplify the presentation of the subject. The principles discussed here also apply to an n-sub-processor structure.

The basic organization consists of a processor composed of two general (or special-purpose) sub-processors each having its own Arithmetic Logic Unit (ALU), its own control section, and a common section to the sub-processors.

In this particular organization the sub-processors are general-purpose ones. In a more advanced design they may consist of a set of special-purpose units in which, with a slight modification of the Common Control, a particular instruction may be sent to a particular sub-processor in order to speed-up the execution of the instruction stream.

The common block to the sub-processors consists of an Instruction Buffer (IB), a Current Address Register (CAR), a Result Stack (RS), and a control unit. This structure is a very elementary one and in real design it should include other necessary hardware such as stack pointers, stack

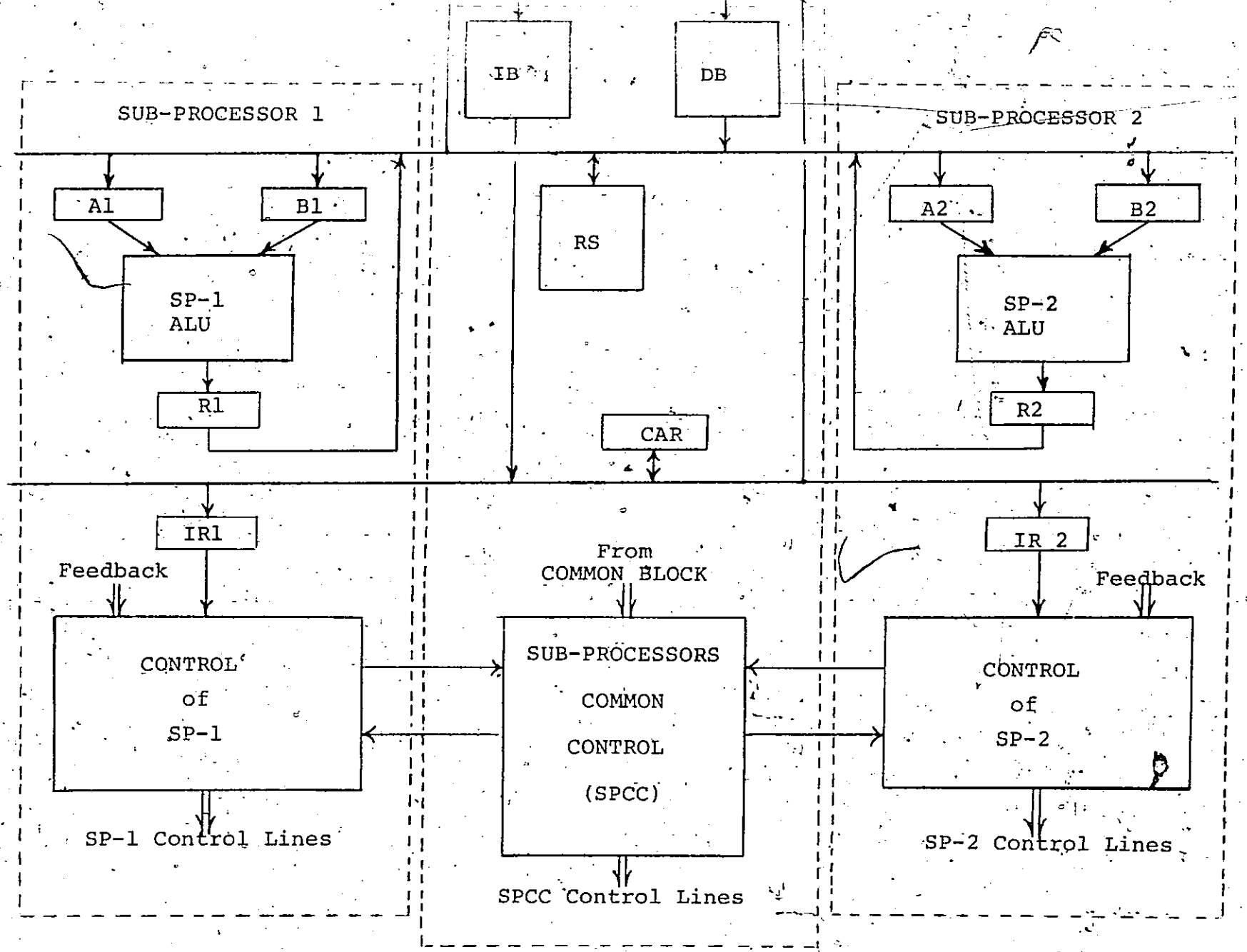


Fig. 3.1 The Basic Organization

limit registers, interrupt registers, interrupt stacks, etc.

For SISD processing the structure of the sub-processors is very simple since the common element takes care of the problem of temporary storage. In a SIMD organization, each sub-processor requires a more complete structure since it has to take care of its own temporary storage. A stack-oriented sub-processor would then be appropriate.

It should be noted here that the I/O is not part of the design and it is assumed to be done by an I/O processor in collaboration with the memory processor.

III-3 SUB-PROCESSOR CONTROL

One of the major cost factors in today's computer technology is the modularity of design of a system. For this reason and for a better independency between the sub-processors it is more advantageous to have a number of independent sub-processors, each having its own control, supervised by a main common control unit, than having a common control unit to do all the control for the sub-processors. Another advantage of having separate control units for the sub-processors is that it permits the addition, to a system, of special-purpose sub-processor units without having to do any major change in the general architecture. Furthermore, distributed control eases the implementation of dynamic microprogramming in the system. For example with the use of look-ahead techniques it is possible to bring in advance a specialized function from secondary storage into a particular sub-processor control unit without having to stop processing in the other sub-processors.

The type of microprogramming used for the sub-processors may be address driven [18] or conventional microprogramming [19, 20]. Address driven microprogramming (ADM) is particularly efficient in this organization since each sub-processor control unit may have its particular control store memory and a common control program store for the sub-processors. Common control programs to the sub-processors

are then stored only once (Fig. 3.2). Further details on address-driven microprogramming can be found in [48].

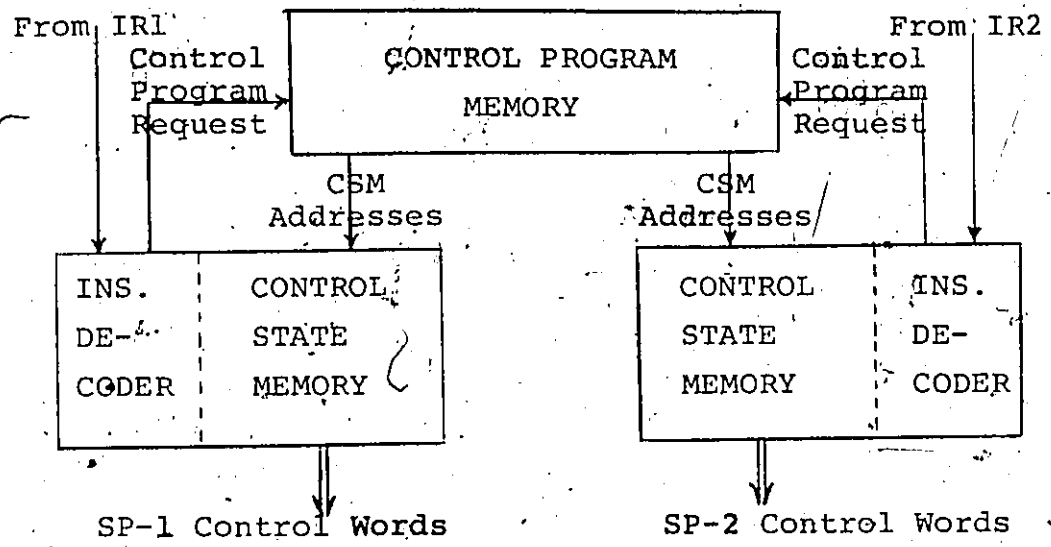


Fig. 3.2 The application of ADM to sub-processor control

In a microprogrammed control unit of a processor, each micro-word consists of a set of bit patterns representing the control state or encoded control state of the processor. The content of the micro-memory address register (MMAR) determines which micro-word has to be executed and may also depend on the process or state. In a multiprocessor system it is possible to control the access of the common resources by allowing access to such resources only on approval of a common control element. The common control (SPCC) element sequences the access of these resources by analysing the control state of each sub-processor at every micro-memory cycle. Furthermore, to permit the fast processing of a single stream of instructions by different sub-

processors, it is necessary to impose interlock conditions on the different sub-processors. These interlock conditions result from the dependencies (procedural, operational, and data) which may occur in successive instructions processed by the system. These interlock conditions can be analysed from the control states of each sub-processor by the SPCC and used to sequence the sub-processors. The SPCC is also used to control data exchange between the sub-processors and can test for failed processing units.

Fig. 3.3 shows a simplified diagram of microprogrammed sub-processor control units with the SPCC. Each control unit sends, at every cycle, one word of information on its state to the SPCC. The set of information words received from the sub-processors constitutes the address of the micro-memory of the SPCC. A proper SPCC word is then fetched and sent back to the individual sub-processor control units. The control words sent to the SPCC may be the individual control words of the sub-processors or a meaningful coded representation of the state of the sub-processors. The feedback from the SPCC allows the control unit of the sub-processors to proceed, to delay the execution in the sub-processors, or to make a modification in the actual execution of a micro-word.

The complexity of such a control is mainly dependent on the way that the different control units of the sub-processors are clocked. In this thesis, we will consider

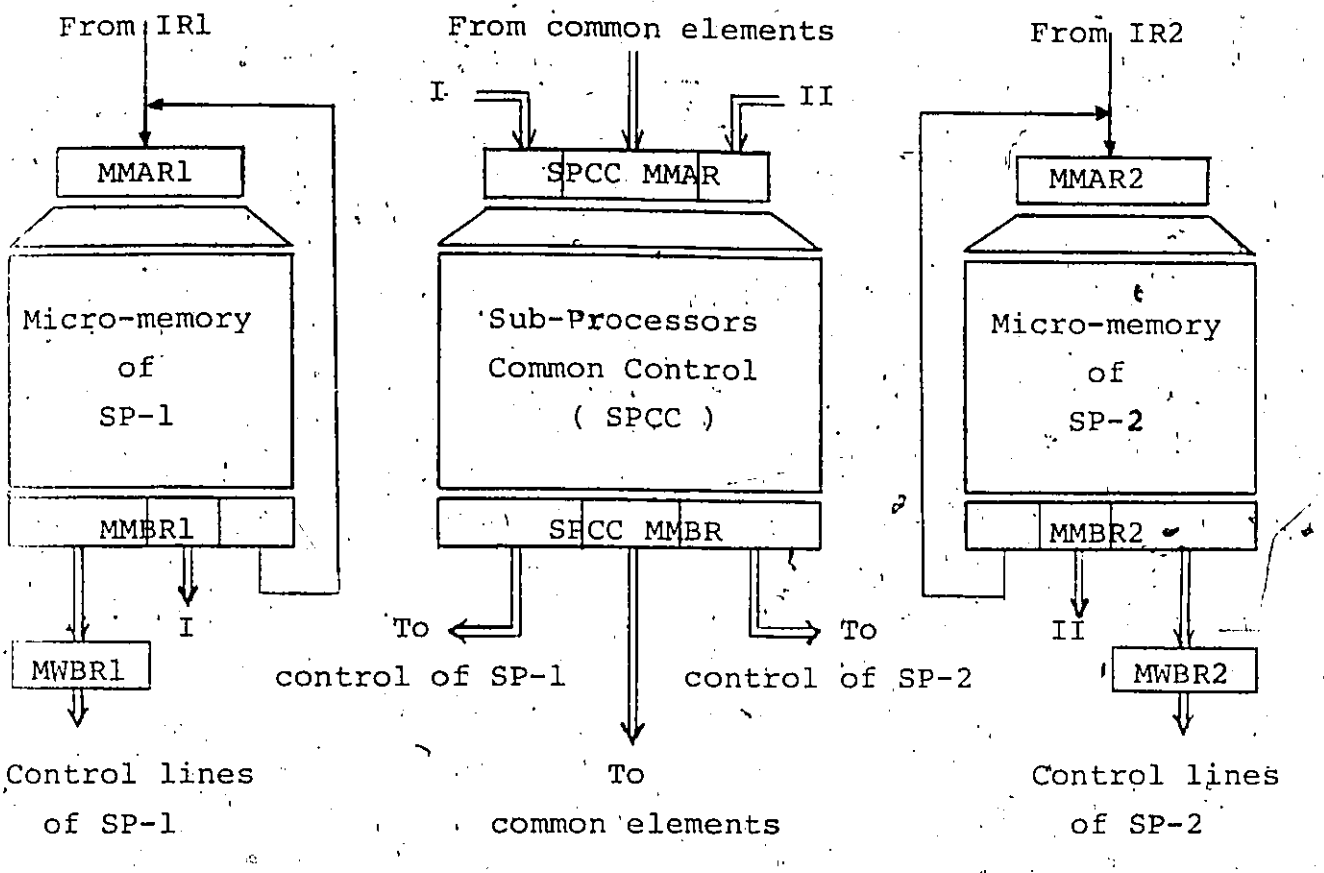


Fig. 3.3 The Control Organization.

only the case where the different control units are clocked simultaneously and have the same cycle time. An asynchronous clocking of the different control units may be more effecient but it increases greatly the complexity of the system. Fig. 3.4 shows the timing diagram of the system of fig. 3.3. First, as shown in fig. 3.4(a) the sub-processors fetch a control word, then information about their states is sent to the SPCC which, in turn, fetches and sends back the proper information to the sub-processor control units in order to allow them to execute, delay, or change the execution of the micro-word in the sub-processor. Fig.3.4(b)

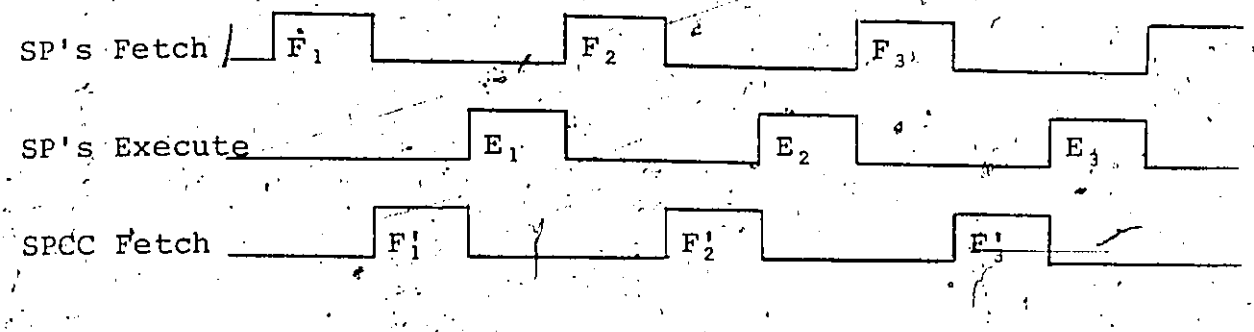


Fig. 3.4 a) Simple SP's control

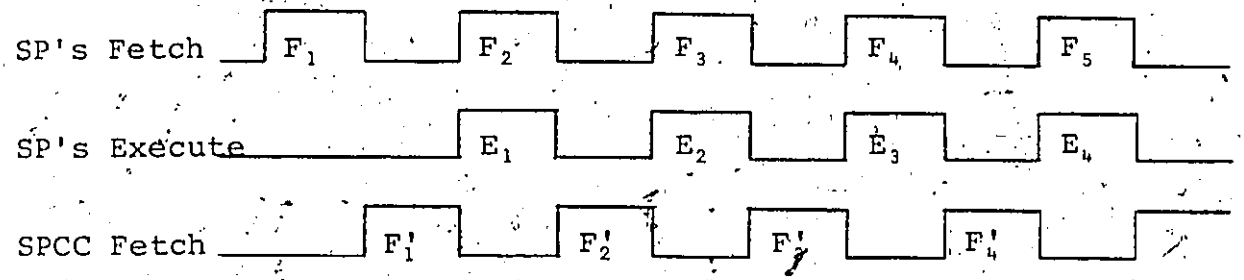


Fig. 3.4 b) Overlapping the fetch and execute cycles in SP's

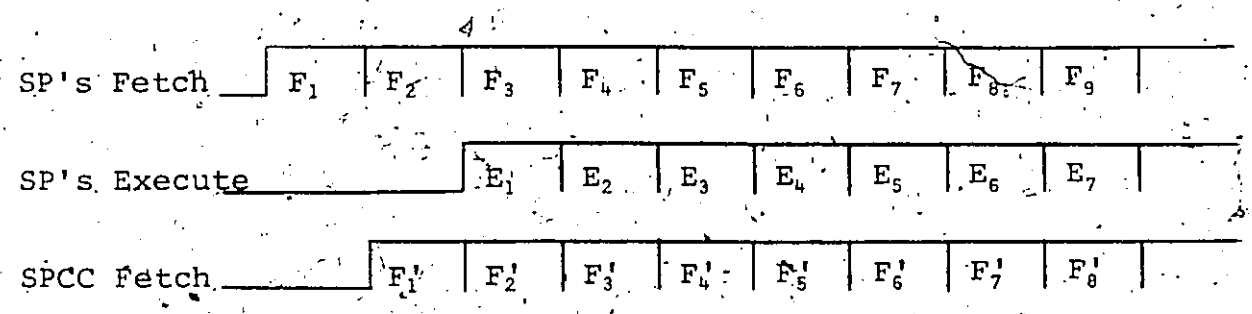


Fig. 3.4 c) SP's control with look-ahead

shows how the fetch and execution cycle can be overlapped in the sub-processors to allow faster control. While a micro-order is executed in a sub-processor, the following one is fetched from its control store. Further speed can be obtained by adding to the sub-processor control units a Micro-Word Buffer Register (MWBR) which allows the control unit to always have a micro-word ready to be executed. However, in such a technique a branching instruction causes a false micro-word to be fetched from the micro-store. This problem can easily be surmounted and such execution is actually done in present-day microprogrammed control units [6]. It should be noted that other types of timing are also possible. Further details on the control part will be given in section III-5.

III-4 DEPENDENCIES AND EXECUTION

In this section the different types of dependencies which must be considered when using the techniques of overlapping, micromultiprocessing, and micro-parallel execution are presented. They arise due to the interaction between instructions and data in SISD and SIMD streams and they directly affect the operation of the sub-processors.

In Chapter II we introduced the techniques of overlapping, micromultiprocessing and micro-parallel execution. The time saved using these techniques and the possible idle times which occur are shown in fig. 3.5 and depend on the interaction between consecutive instructions.

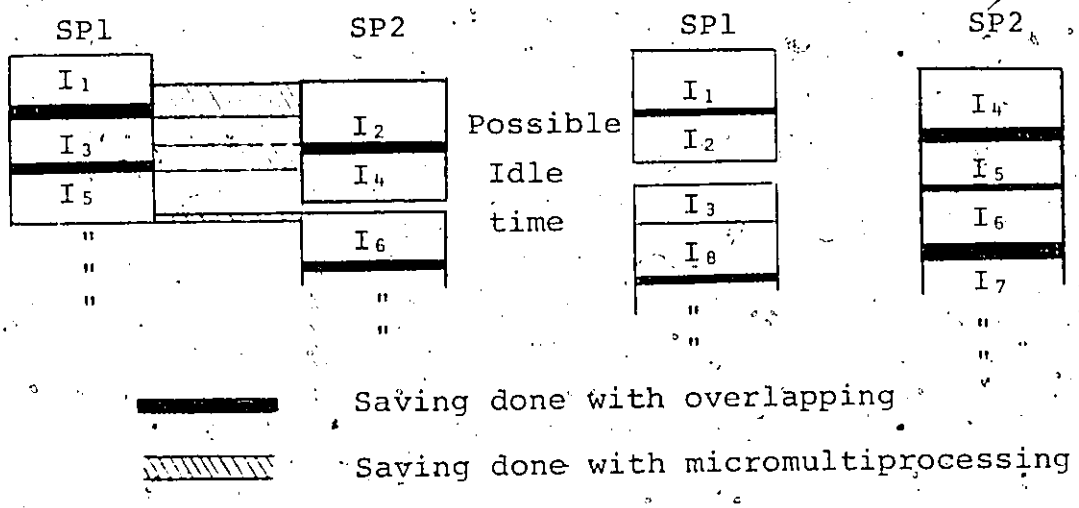


Fig. 3.5 Micromultiprocessing and micro-parallel execution.

Furthermore in the proposed organization the processing techniques used also depend on the dependencies between consecutive instructions in the instruction stream. When no

inherent micro-parallelism is present in the stream of instructions, overlapping and micromultiprocessing are done. When micro-parallelism is detected, the processors operate in parallel and overlapping is done between consecutive instructions. Here it should be noted that the overlapping is done at the control level while the other techniques are used in the execution of the various operations.

The dependencies which arise can be classified into three types: 1) procedural, 2) operational, and 3) data.

A procedural dependency is a dependency in the specification of the instruction sequence. Given a sequence (ascending in time) of m instructions, if I_i is a branch, then for $\{I_1, I_2, \cdots I_i, \cdots I_m\}$ all instructions I_j, $i<j\leq m$, are dependent on I_i. If I_i is an unconditional branch, then the dependencies can be quickly resolved and removed by refetching the correct sequence of instructions into IB. If I_i is a data conditional branch, some idle time occurs until the data test is made before the dependency can be resolved. In section III-6 this problem, where an instruction explicitly specifies the address of the next, will be looked at in more detail.

An operational dependency arises when a resource, common to the sub-processors, associated with the operation specified by an instruction I_i is busy. That is when we have a set of operations $\{OP_1, OP_2, OP_3 \cdots OP_i \cdots OP_k\}$ executed in

parallel by k sub-processors, resources can only be distributed sequentially to the first sub-processor up to the i^{th} one, where i is the number of resources available, $i<k$. Then, all instructions (operations) OP_j; $i<j\leq k$ will be dependent. For example, in our two-sub-processor organization, if two operations executed in parallel require the access to IB, priorities must then be assigned and one processor must wait for servicing. The same problem arises when a set of parallel instructions $\{I_1, I_2 \cdots, I_i, \cdots I_k\}$ are to be distributed into a system of j sub-processors where j, $i<j\leq k$. All instructions $i>j$ will be dependent. For example in distributing the sequence of instructions of the algebraic expression $x=(A+B)(C+D)(E+F)(G+H)$ in a two-sub-processor system, (E+F) and (G+H) cannot be executed concurrently with (A+B) and (C+D) as there are only two sub-processors, therefore, operational dependencies arise. It should be noted that the second type of operational dependencies will not cause any idle time in the sub-processors, but it will prohibit a maximum efficiency in the processing of a single stream of instructions.

The third class of dependencies, data dependencies, arises in a sequence of m instructions when a set of instructions S_i affects (ie. has a sink operand) the source operand of a subsequent set S_k, $i<k\leq m$. For example if S_i is the set of instructions to execute X=A+B, and S_k the set of instructions to execute Y=X+C, then S_i must be completed before

one can execute S_k. In our organization this is taken care of by checking the access to the RS, that is S_k will be executed only after the result of S_i entered the RS.

The different techniques of execution (overlapping, micromultiprocessing, micro-parallel processing) can be executed into an n-sub-processor unit. In the next chapter we will show how it is done. It should be noted that in such an n-sub-processor organization SIMD and MIMD processing is possible with a proper modification in the sub-processors and the common block. This will be discussed in Chapter IV.

III-5 FUNCTIONAL OPERATION

In this section we will show how the two-sub-processor unit operates. In order to do so, the language used by the processor will be presented. Then we will show how the defined architecture can take advantage of the language to detect parallelism in the stream of instructions and how the language can take advantage of the architecture of the processor to implement high level language instructions. Here we will assume that the stream of instructions does not contain any logical or branching instructions. These instructions generally slow down the execution of the instruction stream, and they will be treated separately in another section.

In this organization all the instructions are fetched from the memory according to the (CAR) and are pushed into the IB and the corresponding data into DB by a separate fetching element (or memory processor). This is done continuously in order to supply the required flow of instructions to the processor. The sub-processors consequently fetch all their instructions and data from the IB and DB, respectively. The execution cycle of these instructions is then overlapped (overlapping of non consecutive instructions) and micro-multiprocessed by the sub-processors until micro-parallelism is detected. When such parallelism is detected, the sub-processors work in parallel, overlapping consecutive

instructions. Here it should be noted that the IB and DB may not always contain the right sequence of instructions and data, that is a branching instruction could cause a wrong sequence of instructions and data to be sent to the IB and the DB. In such a case, these instructions and data are eliminated from the IB and the DB, and the right sequence is fetched from the memory by changing the content of the CAR. This is done by the sub-processors common control (SPCC) to which the sub-processors are reporting continuously. Further details on this will be given in section III-6.

For such an organization it is necessary to utilize a language which reduces the problem of temporary storage. Furthermore, studies show [11] that a stack-oriented language minimizes this problem and also permits a more efficient programming. The language proposed for this organization is based on a modified inverse Polish notation which is described in more details in Appendix I. Here we only introduce the parts of the language and the modified Polish notation which are pertinent to explain how the processor operates. The choice of the language is also motivated by the fact that with the postfix Polish notation it becomes easy to translate a high level language to machine code. The structure of the notation also allows an automatic detection of micro-parallelism in a single stream of instructions.

The postfix Polish notation enables the writing of

algebraic or logical expressions without explicitly grouping symbols and without requiring explicit operator precedence conventions. For example, parentheses are necessary as grouping symbols in the expression X=(A+B)/(C-D) to convey the desired interpretation of the expression. In the inverse (postfix) Polish notation the expression would be written AB+CD-/X=.

The proposed language for the processor can be best introduced through an example. In fig. 3.7 a comparison of a single register machine language, a stack machine language, and the proposed language for the expression X=(A+B)/(C-D) is presented. From the example we can see that the proposed language takes less memory locations and instructions than the others. This improvement is achieved

SINGLE REGISTER MACHINE		STACK MACHINE		PROPOSED LANGUAGE	
LOAD	A	PUSH	A	LOAD	A
ADD	B	PUSH	B	ADD	B
STORE	Temporary 1	ADD		LOAD	C
LOAD	C	PUSH	C	SUBTRACT	D
SUBTRACT	D	PUSH	D	SDIVIDE	
STORE	Temporary 2	SUBTRACT		STORE	X
LOAD	Temporary 1	DIVIDE			
DIVIDE	Temporary 2	POP	X		
STORE	X				
16 memory locations 9 instructions		13 memory loc. 8 instructions		11 memory loc. 6 instructions	

Fig. 3.7 A comparison of machine languages

by storing automatically the partial results into a stack (result stack) as they become available, and by introducing instructions which operate on the partial results in the stack. It should be noted that in the inverse Polish notation each operator produces a partial or final result.

By pushing these results automatically into a stack after completion of an operation, we reduce the programming effort, use the processor more efficiently, and save storage. Furthermore, since in the inverse Polish notation an operator not preceded by an operand always implies an operation between partial results, it becomes convenient to define a general operator working on partial results which are stored in the result stack. Such operations can then be specified by an operate instruction (zero address). For example the algebraic expression X=(A+B)(C+D)(E+F)(G+H) can be written as AB+CD+EF+GH+*X= and executed as follows:

```
LOAD    A
ADD     B
            ← First partial result pushed into RS.
LOAD    C
ADD     D
            ← Second partial result pushed into RS.
LOAD    E
ADD     F
            ← Third partial result pushed into RS.
LOAD    G
ADD     H
            ← Fourth partial result pushed into RS.
SMPY        ← All the partial results are multiplied together
              by this instruction.
STORE   X
```

Here the SMPY instruction operates on all partial results in the RS, which represents a very large saving of instructions and memory locations if we compare this program to one written in ordinary stack machine language or register language. In fact, the proposed language reduces greatly

the programming effort and the processing of programs before the actual execution. This is achieved by providing a closer hardware/software relationship in the design of the processor.

As for algebraic expressions, one can use a modified form of Polish notation in order to translate high level logical and branching instructions into machine language for the proposed processor. For example the Fortran statement "If ((A+B).GE.(C*D)) go to N" can be written as AB+CD*>N→ where → means "branch to". The corresponding program is the following:

```
LOAD   A
ADD    B
                ← First partial result into RS.
LOAD   C
MPY    D        ← Second partial result into RS.
JGE    N        ← Partial results into RS are compared and the
                  branching is done to N if the first partial
                  result is greater than or equal to the second.
                  Otherwise the instruction following JGE N is
                  executed.
```

It should be noted that much more complex branching instructions (which are not simple in Fortran) such as "If ((A+B).GT.(C+D).LT.(E-F)) go to N" where (A+B) must satisfy >(C+D) and <(E-F) can then be easily realized. The corresponding modified Polish notation is then AB+CD+>EF-<N→ . More details on the proposed modified inverse Polish notation as well as on the language are given in Appendix I and II.

Using this notation, it becomes very easy to detect parallelism which exists in a stream of instructions. For example if we look at an arbitrary algebraic expression, we notice that everything on the left of an operator is independent of what is on its right until two consecutive operators are met. For example, in AB+CD-*EF-/X=, AB+ is independent of CD- and AB+CD-* is independent of EF-. Using this feature of the notation, it is then easy to detect parallelism existing in an algebraic expression. Parallel execution is then possible, and the sequencing of such parallelism is based upon the fact that partial results must be fed to the RS by the sub-processors in the same sequence that their operands were fetched from IB by the sub-processors. In our example the partial result of AB+ has to be pushed into the RS before the partial result of CD-. Note that the instructions to execute AB+ are fetched from IB before the ones for CD-. More details on how the sub-processors use the RS will be given in the following section.

Parallel execution can also be executed within an instruction which specifies multiple operations on partial results in the RS. If we look at the previous example X=(A+B)(C+D)(E+F)(G+H), the multiplication of the partial results is done by the SMPY instruction which can be executed concurrently by the two sub-processors. This type of micro-parallelism (between stages of execution in

an instruction) is permitted whenever the instruction implies a commutative operation. For example, division is not a commutative operation and a parallel execution of an instruction which divides many partial results in the RS is not permitted.

In the last example, it should be noted that a minimum processing time for the execution of the expression is reached with four sub-processors. In general, for every expression there will be a specific number of sub-processors that will give minimum processing time. Even though the processing time decreases up to a certain point with the number of sub-processors available, the average efficiency of a sub-processor in the system decreases as dependencies arise in the instruction stream. Let T_n be the time required to perform some calculation using n sub-processors, and T_1 be the time required to perform the same calculation by a single processor. We obtain a speed up of the system $S_n = T_1/T_n$ and the average efficiency E_n of a processor as $E_n = S_n/n = T_1/nT_n$ which is smaller than one. Another important factor which affects the efficiency is the number of operations required to compute one expression. As explained by Kurk et al [13], computation time may be saved by performing extra operations. For example, in an n-sub-processor unit the execution of a(b+cde) requires four operations and $T_n = 4$, whereas ab+acde requires five operations and $T_n = 3$. Consequently saving in computation time may be achieved through

a proper compilation of a program.

In this thesis, we will not consider such cases, and we will assume that the sequence of instructions given to the processor is arranged to its minimal form.

The best way to introduce how the sub-processors operate is through an example. Fig. 3.8 shows how the two-sub-processor unit (fig. 3.1) executes an arbitrary algebraic expression. Suppose that sub-processor one starts, it will fetch the first instruction from IB. As soon as

Inverse Polish notation

AB+CD-/X=

1) LOAD A
2) ADD B
3) LOAD C
4) SUB D
5) SDIV
6) STORE X

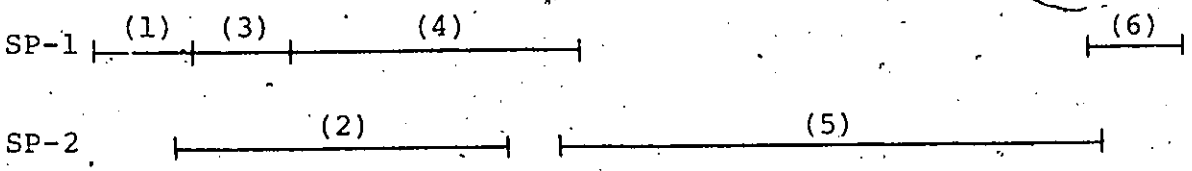


Fig. 3.8 Execution of x=(A+B)/(C-D)

IB is free, sub-processor two will fetch the second instruction from IB while sub-processor one executes its instruction, that is it loads the first data into the opposite processor in register say A 2. Sub-processor two will then get the second data B into its register B2 and add (B2) to (A 2) and push the result into RS while sub-processor one starts executing the third instruction.

It should be noted here that micro-parallelism has been detected (at the second load instruction) but

more micromultiprocessing than parallel execution has been done since there is only a short sequence of micro-paralellism involved in the expression. Furthermore, for such cases where there exists only a short sequence of parallelism in an expression, parallel execution requires the same processing time as a micromultiprocessed execution.

Now consider the example of fig. 3.9. In this example, (C+D-E) is executed concurrently with (A*B). In

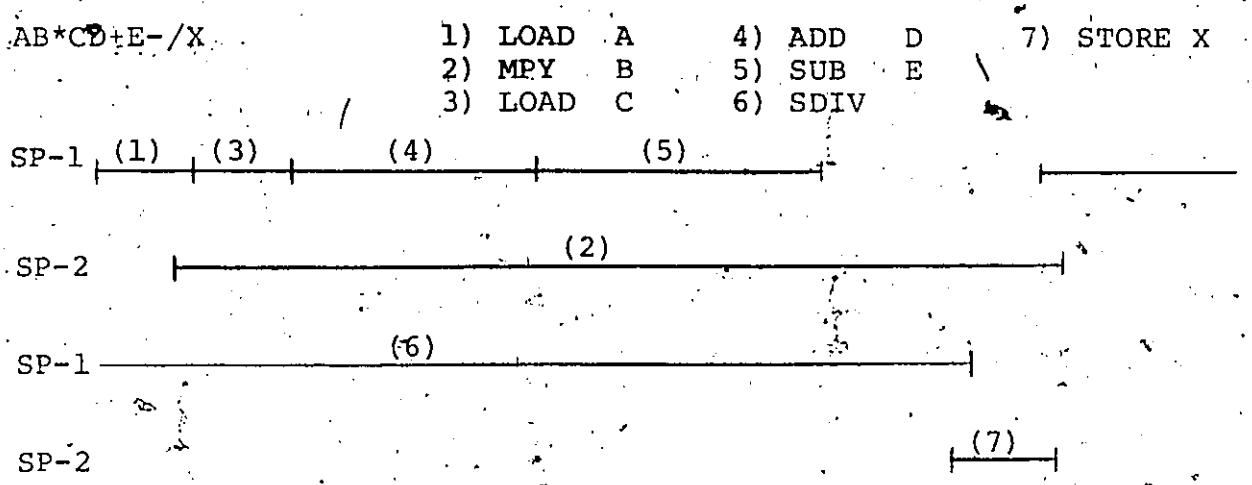


Fig. 3.9 Execution of X=(A*B)/(C+D-E).

fact, the type of processing utilized is not predetermined. As soon as they are free the sub-processors simply fetch their instructions from IB and try to execute them as soon as they are allowed to by the SPCC. Parallel execution only occurs when a particular sub-processor is busy executing a long instruction or the sequence of instructions fetched from IB by one of the sub-processors happens to belong to a particular branch of a stream where parallelism is present. In the next chapter, another method which per-

mits more parallel execution is presented, but it requires two result stacks instead of one in order to avoid the confusion of partial results.

III-6 MORE DETAILS ON THE PROCESSOR OPERATION

In the preceding section we introduced the principle of operation of the basic organization and used only approximate timing diagrams. Here we will consider micro-step by micro-step execution in the processor. This section will also include a sequencing scheme for the control units of the sub-processors (SP).

To simplify the presentation of this section we will assume that each micro control word of the sub-processors does not contain more than one micro-step. However, more than one micro-word may be required to execute a particular micro-step. The second assumption made here is that the access time of the control memories (micro-memories of the sub-processors) do not add any appreciable delay to the execution of the micro-steps of the instructions.

In an actual design, more than one micro-step may be included in one micro-word. This is done to reduce the number of fetches to the micro-memory of the control unit and/or to reduce the idle time that would occur when the actual execution time of a micro-step is much smaller than the micro-memory access time.

The number of micro-steps included in a micro-word actually depends on the size of the micro word (#of bits), on the type of microprogramming (direct or encoded) used

for the design of the control unit, and on the number of micro-steps which are allowed to be executed by the same micro-word. In this thesis we shall not consider the problem of having to choose between direct or encoded microprogramming in the design of a control unit. The principles of sub-processor control introduced in this thesis are applicable in a direct type of microprogramming as well as in an encoded one. Further detail on these microprogramming techniques and their respective advantages may be found in [20].

A detailed description of the operation of the proposed processor can best be introduced through an example. Fig. 3.10 shows a listing of all the micro-steps required to execute an arbitrary algebraic expression, namely X=(A*B)+(C*D). In postfix Polish notation this is written as AB*CD*+X=. The figure also includes the relative time of execution of each micro-step (which are given in detail in Appendix III). It should be noted that this sequential listing of micro-steps represents how the arithmetic expression would be executed by a uniprocessor operating with the proposed language and the proposed structure. The total execution time in such a uniprocessor would then be 348t (time units).

Fig. 3.11 shows how the same program is executed by the two sub-processor unit. Here we assumed that sub-processor one started the execution. As mentioned earlier, we can see that each micro-step is executed as soon as possible by the sub-processors. The delay occurring between

INSTRUCTION		MICRO-STEP		RELATIVE TIME
LOAD	A	(1)	IR←(IB)	2t
		(2)	A ←(DB)	2t
MPY	B	(3)	IR←(IB)	2t
		(4)	B ←(DB)	2t
		(5)	R ←(A)*(B)	150t
		(6)	RS←(R)	2t
LOAD	C	(7)	IR←(IB)	2t
		(8)	A ←(DB)	2t
MPY	D	(9)	IR←(IB)	2t
		(10)	B ←(DB)	2t
		(11)	R ←(A)*(B)	150t
		(12)	RS←(R)	2t
SADD		(13)	IR←(IB)	2t
		(14)	A ←(RS)	2t
		(15)	B ←(RS)	2t
		(16)	R ←(A+B)	20t
		(17)	RS←(R)	2t
TOTAL				348t

Fig. 3.10 Execution of X=(A*B)+(C*D) by a single processor

SP1		SP2		EXECUTION TIME SP1	SP2	TOTAL
(1)	IR1←(IB)			2t		2t
(2)	A2 ←(DB)	(3)	IR2←(IB)	2t	2t	4t
(7)	IR1←(IB)	(4)	B2 ←(DB)	2t	2t	-6t
(8)	A1 ←(DB)	(5)	R2 ←(A2)*(B2)	2t	150t	156t
(9)	IR1←(IB)		↑	2t	↑	↑
(10)	B1 ←(DB)		↓	2t	↓	↓
(11)	R1 ←(A1)*(B1)			150t		
	↑	(13)	IR2←(IB)		2t	158t
	↓	(6)	RS ←(R2)		2t	160t
(12)	RS ←(R1)			2t		162t
		(14)	A2 ←(RS)		2t	164t
		(15)	B2 ←(RS)		2t	166t
		(16)	R2 ←(A2)+(B2)		20t	186t
		(17)	RS ←(R2)		2t	188t

Total time of execution=188t

Fig. 3.11 Execution of X=(A*B)+(C*D) by two sub-processors

the execution of micro-steps within an instruction is due to procedural, operational, and data dependencies. Fig. 3.12 shows how the instructions are executed by the sub-processors.

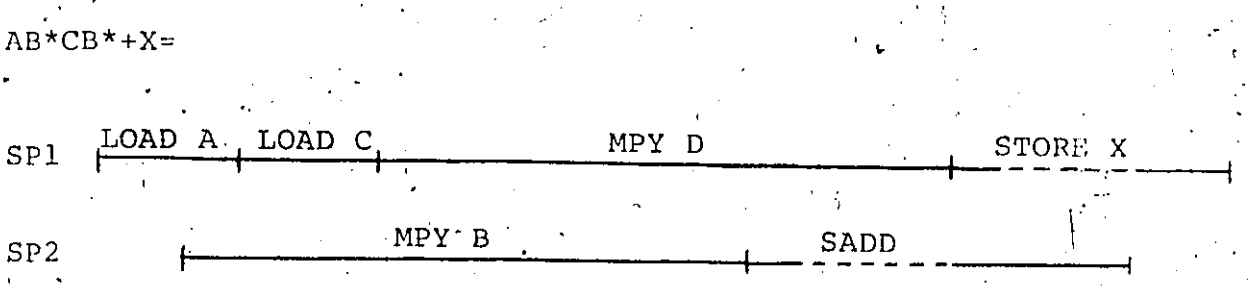


Fig. 3.12 Execution of X=(A*B)+(C*D) by two sub-processors.

The processing time of the algebraic expression by two sub-processors is 188t. This represents a speed up S=348t/188t=1.91 which is achieved without preprocessing the instruction stream to detect parallelism. Furthermore, it should be noted that we compared the processing speed of the two sub-processor unit to a uniprocessor having the same language as code for its ALU, and we did not include the speed up which is achieved with the language proposed for the system. For example a conventional stack machine language would have required eight instructions instead of the six required with the chosen language.

Appendix IV shows the algorithm and gives the

listing of a program which has been written to show the efficiency of the two sub-processor system over the uni-processor system for execution of algebraic expressions. Without taking into account the speed-up achieved with the language, the average speed-up obtained is approximately 1.51 for 250 arbitrary arithmetic expressions.

Now we shall consider how the actual execution of the micro-steps is done at the control unit level. Fig. 3.13 shows the timing diagram of the control units for the execution of X= (A*B)+(C*D) . On this timing diagram, it was assumed that the access time of the micro-memories did not add any delay to the actual execution of the micro-step. It should be noted that operational dependencies arise at the execution of micro-steps one and three. $F_{1,3}$ is done by the SPCC and the execution is only allowed for micro-step one in SP-1 (E_1). At the next SPCC cycle, micro-steps two and three are analysed and since there are no dependencies these micro-steps are executed concurrently (E_2 and E_3). Operational dependencies also occur at the execution of micro-steps 12 and 14.

To allow such a sequencing in the two-sub-processor unit of fig. 3.1, five bits have to be sent from each sub-processor to the SPCC at every micro-memory cycle. Three are used to direct the access to the common elements, one bit to tell the SPCC if a sub-processor is idle or executing, and the last bit is used to control the state of a flip flop

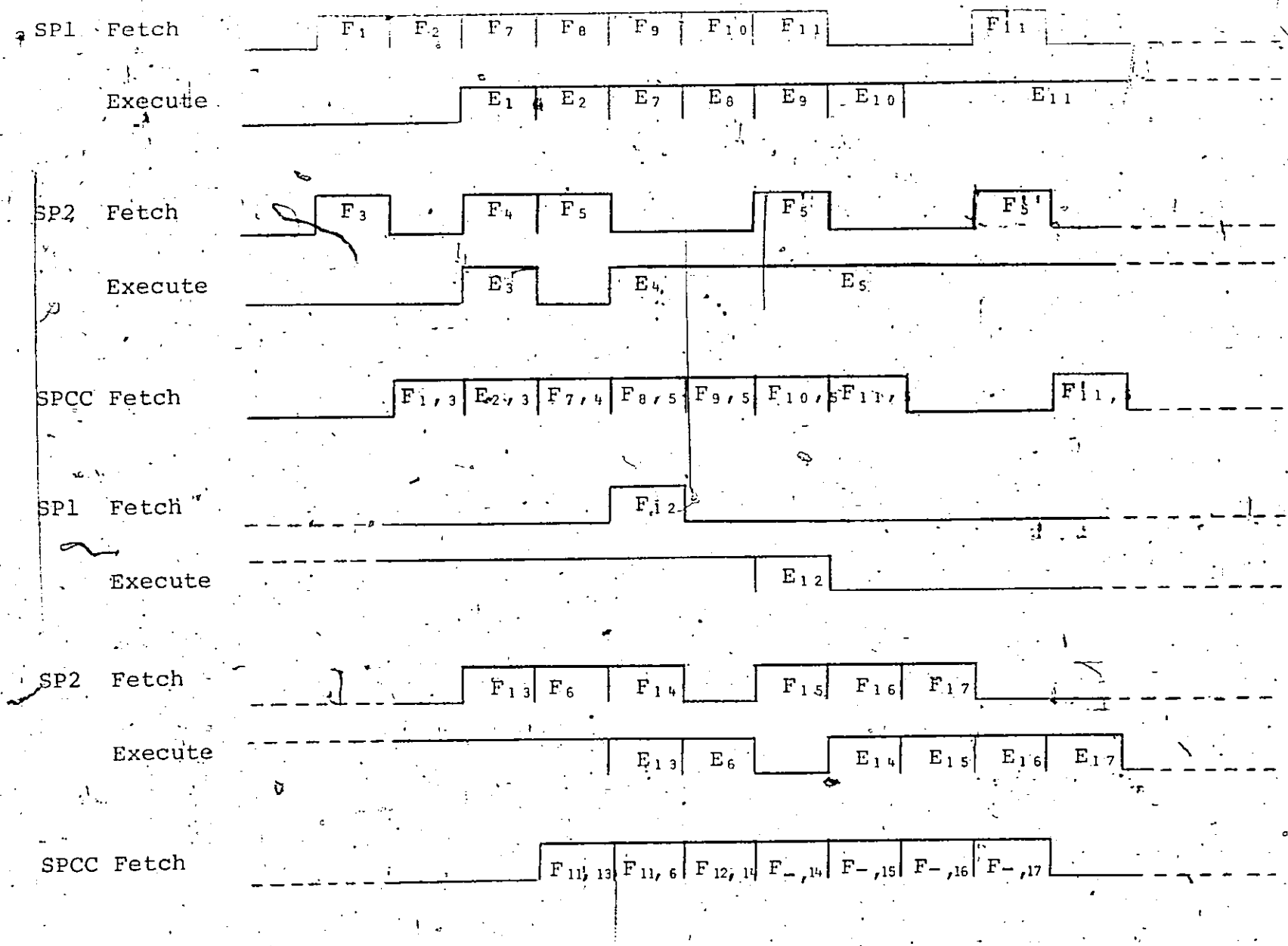


Fig. 3.13 Control timing for X=(A*B)+(C*D)

which will allow the partial results to enter the RS in the right sequence. Another bit must be used to inform the SPCC if there is any data in the RS in order to resolve data dependencies.

As an example we defined in table 3.1 the control codes used to direct the access of the common elements. Operational dependencies can then be resolved by the SPCC by checking these codes. For example the SPCC receives at time t 001 as a code from both sub-processors. This will imply an operational dependency. Therefore, at the SPCC address composed by these codes, we insert a control word which

ELEMENT	CODE
No request	000
IB	001
DB	010
CAR	011
RS(Fetch)	100
RS(Store)	101
Spare	110
	111

Table 3.1 Control Codes

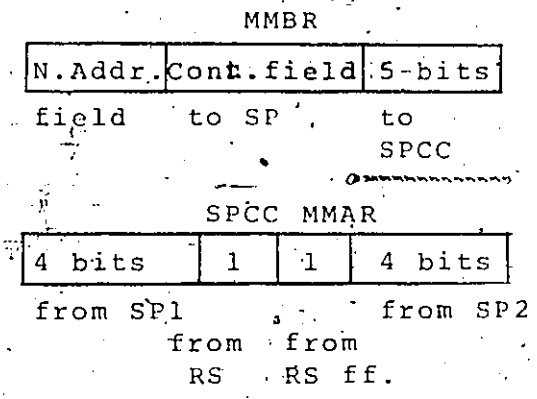


Fig. 3.14 Format of MMBR and CMMAR

gives the priority of operation to a particular unit, that is delay the execution in one sub-processor.

In this organization the partial results must be fed to the RS in the same order that the LOAD instructions associated with these operations are fetched from the IB. For example in executing X=(A*B)/(C+D), the result of (A*B)

must enter the RS before that of (C+D) in order to avoid confusion of partial results. This is done by the sub-processor which sends a one bit signal to change the state of a flip flop in the SPCC each time a LOAD instruction is executed. The state of this flip flop is sent to the SPCC MMAR and this will determine the address of an appropriate CPCC word which will, in the last example, not allow the result of (C+D) to enter the RS before that of (A*B). Fig. 3.15 shows how X=(A*B)/(C+D) will be executed.

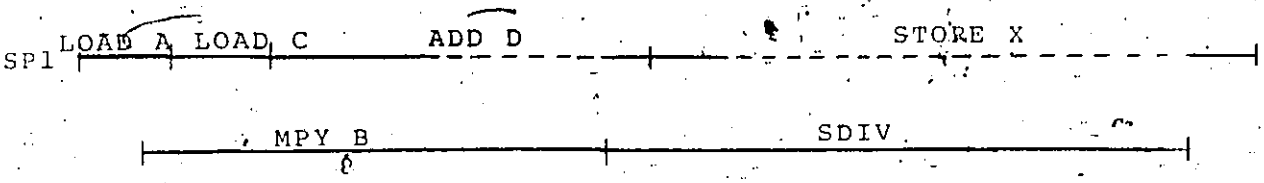


Fig. 3.15 Execution of X=(A*B)/(C+D)

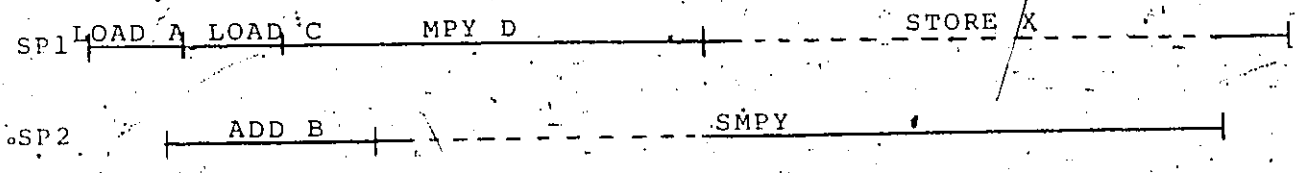


Fig. 3.16 Execution of X=(A+B)(C*D)

As mentioned earlier, one bit must be sent from the RS to the SPCC in order to resolve data dependencies. This can be seen from the example of fig. 3.16 which shows how X=(A+B)(C*D) is executed. In this example the execution of the SMPY instruction is delayed until the result of (C*D) is known. As the result of (C*D) enters the RS, SP1 will fetch it and will continue the execution of its instruction.

INSTRUCTION		MICRO-STEP		RELATIVE TIME
LOAD	A	(1)	IR←(IB)	2t
		(2)	A ←(DB)	2t
ADD	B	(3)	IR←(IB)	2t
		(4)	B ←(DB)	2t
		(5)	R ←(A)+(B)	20t
		(6)	A ←(R)	2t
SUB	C	(7)	IR←(IB)	2t
		(8)	B ←(DB)	2t
		(9)	R ←(A)+(B)	20t
		(10)	A ←(R)	2t
DIV	D	(11)	IR←(IB)	2t
		(12)	B ←(DB)	2t
		(13)	R ←(B)/(A)	300t
		(14)	RS←(R)	2t
LOAD	F	(15)	IR←(IB)	2t
		(16)	A ←(DB)	2t
ADD	G	(17)	IR←(IB)	2t
		(18)	B ←(DB)	2t
		(19)	R ←(A)+(B)	20t
		(20)	A ←(R)	2t
MPY	C	(21)	IR←(IB)	2t
		(22)	B ←(DB)	2t
		(23)	R ←(A)*(B)	150t
		(24)	RS←(R)	2t
SADD		(25)	IR←(IB)	2t
		(26)	A ←(RS)	2t
		(27)	B ←(RS)	2t
		(28)	R ←(A)+(B)	20t
		(29)	RS←(R)	2t
TOTAL				576t

Fig 3.17 Execution of X=[(A+B-C)/D]+[(F+G)*C] by a single processor.

SP1	SP2	EXEC. TIME	OPR1	OPR2	EX1	EX2	BITS to RS S1	BITS to RS S2	RS FF	RS FF	DEPENDENCY
(1) IR1←(IB)	(3) IR2←(IB)	2t	001	001	1	0	1	0	0	0	Opr.
(2) A2 ←(DB)	(3) IR2←(IB)	4t	010	001	1	1	0	0	0	1	
(7) IR1←(IB)	(4) B2 ←(DB)	6t	001	010	1	1	0	0	0	1	
(8) A1 ←(DB)	(5) R2 ←(A2)+(B2)	26t	010	000	1	1	0	0	0	1	
(9) R1 ←(B1)-(A1)	(11) IR2←(IB)	28t	000	001	0	1	0	0	0	1	Data
(9) R1 ←(B1)-(A1)	(6) B1 ←(R2)	30t	000	000	0	1	0	0	0	1	Data
(9) R1 ←(B1)-(A1)	(12) A2 ←(DB)	50t	000	010	1	1	0	0	0	1	
(15) IR1←(IB)	(13) R2 ←(B2)/(A2)	52t	001	000	1	0	1	0	0	1	Data
(10) B2 ←(R1)	(13) R2 ←(B2)/(A2)	54t	000	000	1	0	0	0	0	1	Data
(16) A1 ←(DB)	(13) R2 ←(B2)/(A2)	354t	010	000	1	1	0	0	0	1	
(17) IR1←(IB)			001		1	1	0	0	0	1	
(18) B1 ←(DB)			010		1	1	0	0	0	1	
(19) R1 ←(A1)+(B1)			000								
(21) IR1←(IB)			001								
(20) A1 ←(R1)			000								
(22) B1 ←(DB)			010								
(23) R1 ←(A1)*(B1)			000								
(25) IR1←(IB)			001								
(24) RS ←(R1)			101				1	0	0		
(26) A1 ←(RS)			100				0	0	1		Data
(27) B1 ←(RS)			100		0	1	0	0	0		Data
			100	000	0	1	0	0	0		Data
	(XX) IR2←(IB)	356t	100	001	0	1	0	0	0		Data
	(14) RS ←(R2)	358t	100	101	0	1	0	0	0		Data
(27) B1 ←(RS)		360t	100	-	1	-	0	-	1	0	
(28) R1 ←(A1)+(B1)		380t	000	-	1	-	0	-	0	0	
(29) RS ←(R1)		382t	101	-	1	-	0	-	0	0	
		-	-	-	-	-	-	-	1	-	

Total execution time =382t

- = Next set of instructions

Fig. 3.18 Execution of X={(A+B-C)/D}+{(F+G)*C} by 2 sub-processors.

Therefore, in our organization ten bits enter the CMMAR. Fig. 3.14 gives the required format for the sub-processors MMBR's and the CMMAR.

We shall conclude this section by another example showing how the control resolves operational and data dependencies. Fig. 3.17 shows the micro-steps required to execute X={(A+B-C)/D}+{(F+G)*C}. Fig. 3.18 shows how these micro-steps are then executed by the two-sub-processor organization defined in this section. From fig. 3.18 we observe the following:

i) At the execution of micro-steps one and three operational dependencies are resolved by having a proper control word at the SPCC address where OPR1=OPR2=001.

ii) At the execution of micro-step (9), the control of SP1 stops the execution in the sub-processor since all the required data is not available. The processing will continue at the execution of micro-step (6) when SP1 receives the required data.

iii) At micro-step (27) the RS bit=0 indicating that no data is present in the RS. The execution of the micro-step is delayed until a data enters the RS, that is set the RS bit to one.

iv) The state of the RS flip flop indicates which processor has the priority of access to the RS. Here, a one indicates that the SP2 has priority and 0 SP1.

It should also be noted that before the content of

the R registers are stored, the next instruction is fetched from the IB and decoded. This is done in order to know where the result has to be stored. If the new instruction is a LOAD instruction, the content of the R registers must go to the RS, otherwise it goes either to the A or B registers. Examples of this occur at micro-steps (11), (15), (17), (21), (24) and (XX). Here XX represents the first micro-step of the next sequence of instructions.

In this section we have shown how a microprogrammed scheme can be used to sequence the operation of two sub-processors which contribute to the execution of the same instruction stream. The design presented was very simple. However, in actual design more information will have to be sent between the SPCC and the sub-processors. Here we only intend to introduce the principle of having more than one processor executing the same instruction stream in closely coupled sub-processors (at the control level) where the technique of micromultiprocessing and micro-parallel execution are used concurrently.

Further details such as how much information must be sent between the sub-processor and the SPCC, how this information should be coded, the writing of required microprograms etc., depends on the particular design intended.

III-7 EXECUTION OF BRANCHING INSTRUCTIONS

The purpose of this section is to show how branching instructions are executed in the proposed organization. Branching instructions may be sub-divided into two types, unconditional and conditional branch. The unconditional branch does not present any problems since, in filling the IB, it can be taken care of by adding a partial decoder to the IB which will detect the branching and push the correct sequence into the buffer. In this section, we shall therefore consider only conditional branching.

As mentioned in section III-4, conditional branching can only be resolved when the data test is made by the processor. In the simplest type of CPU, the processor can fetch from the memory the right sequence of instructions only after the conditional jump has been resolved. This, consequently, reduces the processing speed of a stream of instructions by a time equal to the memory access time.

In the proposed organization, all instructions are pushed into the IB and the corresponding data into the DB. When a conditional jump occurs in the stream of instructions and the range of the jump is inside the IB, it is only required to delete instructions below the instruction required by the processor. This, then, avoids the fetching of the required instruction from the memory and, consequently, does not add any appreciable delay to the execution

in the processor. However, when the jump is to an instruction which is outside the IB, the IB must be cleared and the right sequence fetched from the memory. In such a case this will involve a delay in the processing of the instruction stream equal to the access time of the memory.

Some speed-up can be achieved by bypassing conditional jumps, that is, by having more than one IB and DB, many tentative computational paths may be maintained simultaneously. However, to bypass N consecutive conditional branches, 2^N IB and DB would then be required. Furthermore, only a small number of conditional jumps can then be bypassed in an actual design. As mentioned in [12] the frequency of occurrence of conditional branching instructions in programs varies from 0.1 to 0.2. Therefore, for an IB and DB of size N, approximately $2 \cdot 2^{2N}$ bypass stacks would take care of quite a large number of conditional branchings.

The problem of conditional branching in actual programming is beyond the scope of this thesis. Further references on this may be found in [16, 17].

CHAPTER IV

OTHER CLOSELY COUPLED PROCESSOR ORGANIZATIONS FOR FAST PROCESSING

The purpose of this chapter is to introduce some additional fast processing techniques which can be implemented through a close communication (at control level) in a multiprocessor environment. In the last chapter we presented an organization which can automatically detect and execute microparallelism in a single stream of instructions. In this chapter, we will first introduce an alternative technique to detect automatically micro-parallelism in a single stream of instructions. Then, we will indicate how SIMD can be executed in closely coupled processors. Finally we will show how more than one program can be executed simultaneously in an organization of more than two sub-processors.

IV-1 DETECTION OF PARALLELISM WITH A PARTIAL DECODER

In section III-5 we presented a method to detect micro-parallelism which was based upon the structure of the inverse Polish notation. As mentioned in that section micro-parallelism was not always executed even if it was present. In this section we will modify the previous organization in order to execute more micro-parallelism in a stream of instructions.

An alternative way to detect micro-parallelism is to associate a partial decoder with the IB. The partial decoder is only used to detect LOAD instructions. By adding an extra bit to the IB (flag bit) which is set to one to indicate the LOAD instructions, it is now possible to send a block of instructions (steps which can be executed in parallel) to the sub-processors for parallel execution. Here, it should be noted that the use of a partial decoder can be avoided by letting the compiler associate the flag bit with the LOAD instructions.

In this organization, two small buffers must be associated with each sub-processor in order to store the block of instructions and the corresponding data which are sent to the sub-processors. Fig. 4.1 shows the modified system. The operation is as follows: the first sub-processor fetches an instruction from the IB, then it looks for a flag bit. If a flag bit is set to one there is another LOAD instruction in the IB, all the instructions

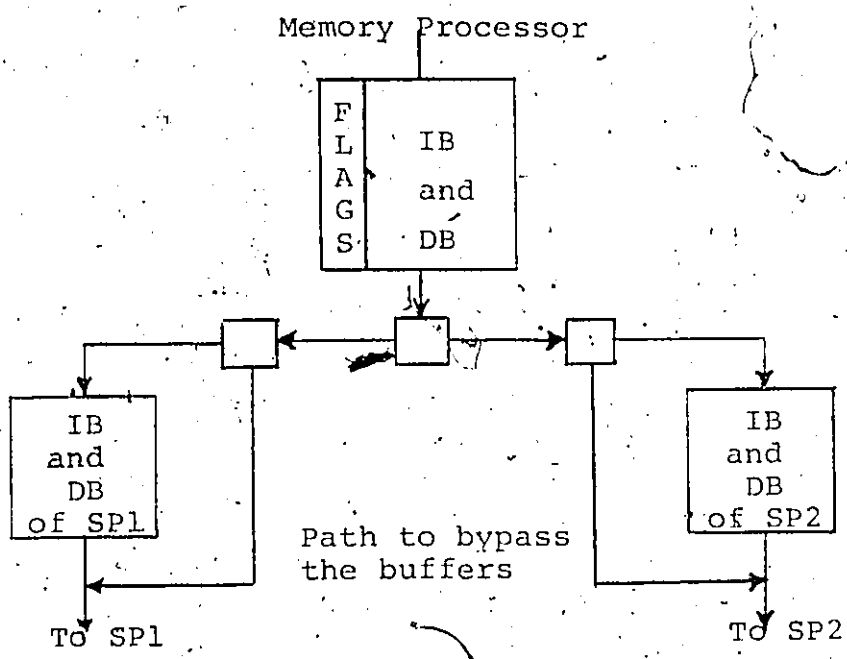


Fig. 4.1 Parallel detection using a partial decoder.

and the corresponding data up to that LOAD instruction are sent to the buffers (instruction and data) of that sub-processor. Then the second sub-processor fetches the next instruction available in the IB, that is the second LOAD instruction. The second sub-processor then looks for a flag bit and if there is one set to 1, it does the same thing as with the first one. However, when there is no flag bit set to one in the IB after the first sub-processor fetched its instruction, this indicates that there is no inherent micro-parallelism inside the IB. Micro-multiprocessing is then done as in the organization presented in the previous chapter. In such a case the buffers of the sub-processors are not used.

Fig. 4.2 shows how the arithmetic expression of

fig. 3.8 would be executed by two sub-processors using a partial decoder to detect micro-parallelism. As in the previous organization, priority must be assigned to order the use of the RS.

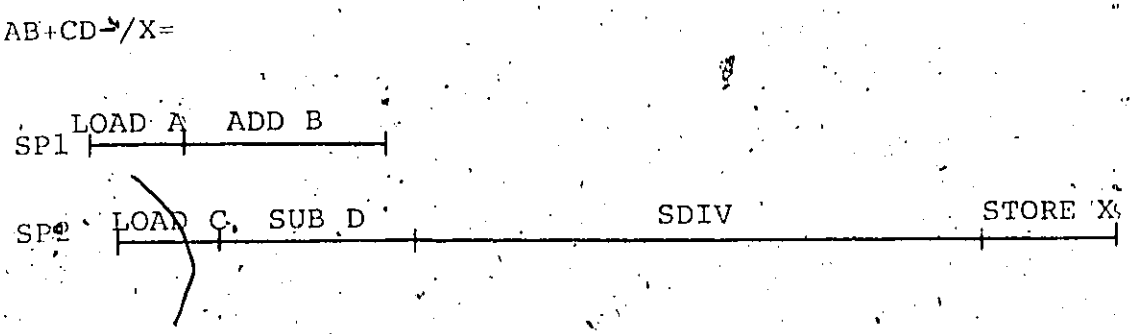


Fig. 4.3 Execution of X= (A+B) / (C-D)

The only disadvantage of this method is that it uses more hardware than the previous one introduced in Chapter III. Its advantages are that more micro-parallelism can be executed when independent consecutive arithmetic expressions are to be executed by the sub-processors. Two Result Blocks are also required in order to avoid confusion of partial results. Fig. 4.3 shows how two consecutive arithmetic expressions, namely X= (A+B) (C+D) and Y= (F+G) (H+I), are executed by the organization described in chapter III. Fig. 4.4 shows how this is now executed when the detection of micro-parallelism is done with a partial decoder (or look ahead) and when two RS are used. From fig. 4.4 we see the need for having two RS's for such an execution. In that organization, the execution of the first arithmetic expression uses one RS and the execution of the second, the other

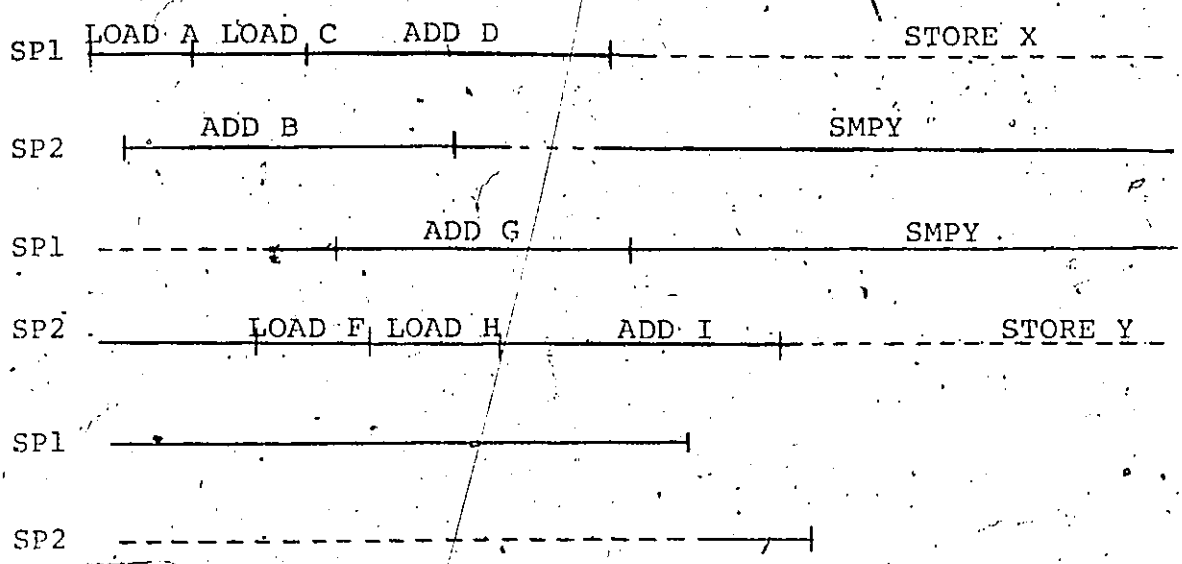


Fig. 4.3 Execution by organization of Chapter III

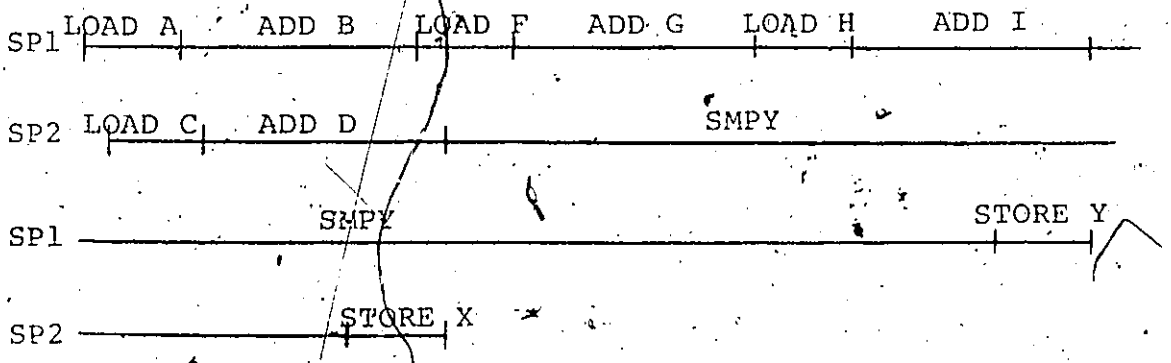


Fig. 4.4 Execution with look ahead

RS. Furthermore, in order to indicate to a sub-processor that it is starting another arithmetic expression and that another RS must be used (at the point marked by an arrow) it is necessary in such an organization to differentiate the first LOAD instruction from the others. This can be easily done at compilation time, that is, the first LOAD instruction of an arithmetic expression would then be a different instruction than the subsequent LOAD instructions in the expression (e.g. a PUSH xx instruction may be used).

The times of execution of the example of fig. 4.3 and 4.4 by the two organizations are respectively ≈380t and ≈260t. Consequently for that particular example, the second organization (with look ahead) gives a speed-up of 1.46 over the organization of Chapter III.

Detection of parallelism with partial decoders can also be extended to an n sub-processor unit. Then n RS are required.

IV-2 SINGLE INSTRUCTION MULTIPLE DATA PROCESSING

SIMD processing can easily be executed by extending a SISD sub-processor organization (Chapter III). Two data streams can be handled simultaneously by having one of the sub-processors slaved to the other and having two RS's and two DB's. However, SIMD processing would inhibit parallel processing and micromultiprocessing. SIMD processing can be allowed on a special instruction directing the SPCC to make the second sub-processor execute the same operations as in the first processor on another data stream.

To maintain the possibility of micromultiprocessing and parallel execution for SIMD processing a more specialised unit must then be designed. By associating with each sub-processor a slaved ALU and an RS common to these ALU's (Fig. 4.5), SIMD can then be executed with micromultiprocessing and parallel processing. Such an organization would

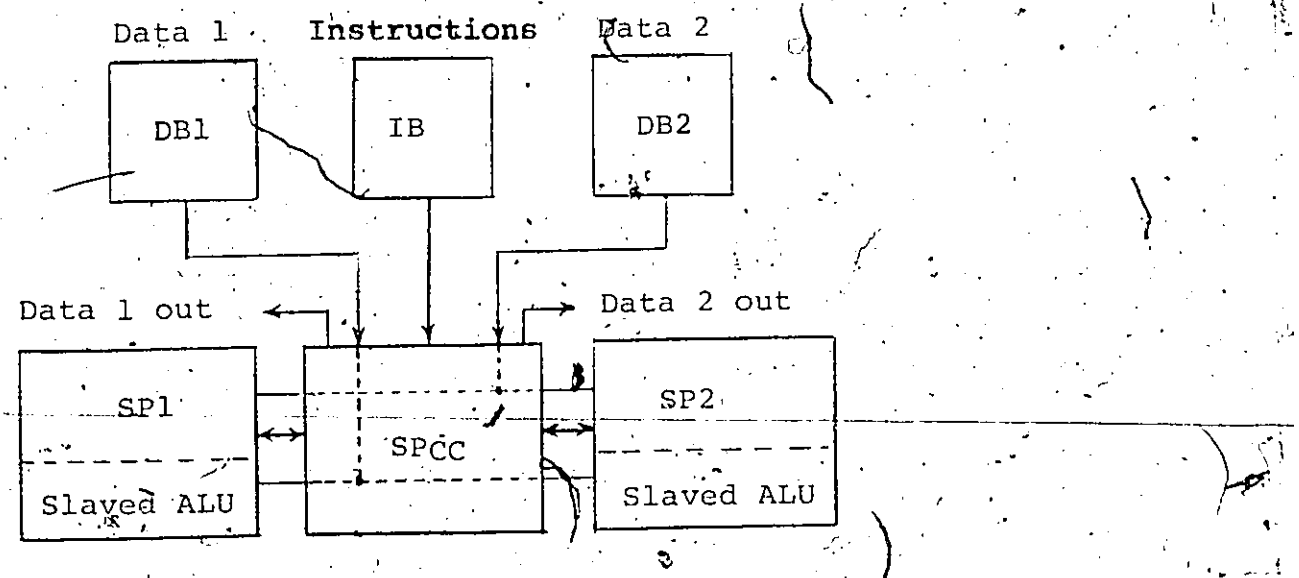


Fig. 4.5 An architecture allowing parallel processing and micromultiprocessing in SIMD executions.

operate as the one studied in Chapter III, but on two data streams. It should be noted that in order to use the look-ahead technique to detect micro-parallelism (Section IV.1) two RS's would then have to be associated with the ALU's slaved to the sub-processors.

IV-3 A MULTIPROCESSING ORGANIZATION

The method of detection of parallelism presented in Chapter III limits the execution of parallelism in a stream of instructions. That is if the processor were implemented with more than two sub-processors, more micro-multiprocessing than parallel execution would be done. In a system of more than two sub-processors it is advantageous to use the micro-parallelism detection method presented in Section IV-1. A study [12] shows that the amount of parallelism in programs depends on the look ahead done in a program. Furthermore, in realizing say a four-sub-processor unit the length of the IB can be chosen such that a four-sub-processor system becomes efficient to execute parallelism. However, when there is no inherent parallelism in a stream of instructions, the four sub-processors would have idle times. Fig. 4.6 shows how micromultiprocessing would be done in a four-sub-processor unit. From the diagram it

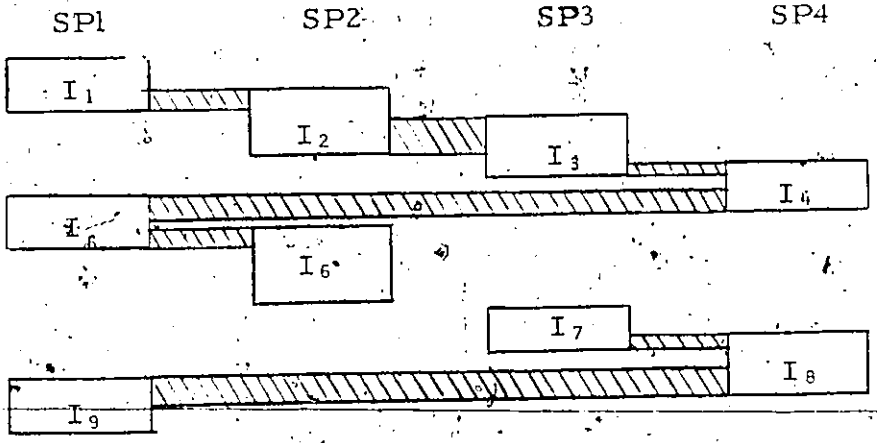


Fig. 4.6 Micromultiprocessing in four sub-processors.

can be seen that micromultiprocessing with four sub-processors

does not give any more speed-up than with a two-sub-processor unit. This only complicates the role of the common control unit. In fact, maximum efficiency with micromultiprocessing can be achieved with two sub-processors.

One way to reduce the idle time in the sub-processors in an organization of more than two sub-processors is to allow a second program to be executed simultaneously with the first one, i.e. the main program. To execute micromultiprocessing the main program can be restricted to use only two sub-processors. The other sub-processors can be used to execute the second or background program. As micro-parallelism appears in the main program, the SPCC reduces or stops the execution of the background program. For example, in a four-sub-processor unit, parallel execution of the main program may require only three sub-processors, thus one sub-processor would then be available to execute the background program. This would also reduce the idle time of the sub-processors when dependencies arise in the execution of the main program. Until these dependencies are resolved, the sub-processors can work on the second program.

Simultaneous execution of two programs in a processor requires a duplication of the IB, DB, RS's, etc., in order to simplify the interrupt of the second program. Such techniques can also be extended to an n-sub-processor unit where a number of programs can run simultaneously in

the system. The greatest problem in designing such an organization would be to design the SPCC (write its micro-program) that would be required to control and sequence the sub-processors. The control unit of a particular sub-processor would not have to be more complicated than the one for the two-sub-processor organization since a particular sub-processor does not have to be associated to a particular program. In such an organization the sub-processors would only be directed to execute a job by the SPCC which would have to associate these sub-processors to a particular program.

CHAPTER V

CONCLUSION

In this thesis we developed an organization that involves a number of closely coupled sub-processors working concurrently on the execution of a single stream of instructions. As shown in the literature survey of Chapter II recent reported techmiques to increase the speed of execution of a single stream of instructions involve either micro-multiprocessing or parallel execution. In this thesis, by proposing both a new processor organization and a suitable language, we introduced how these ideas could be used concurrently to process a single instruction stream.

In Chapter III a two-sub-processor organization was considered in detail. It was shown how a microprogram control scheme can be used to control and sequence the sub-processors in order to allow micromultiprocessing and parallel execution. A common control was used to resolve procedural, operational, and data dependencies which arise when doing such processing. We also showed how the basic unit (two sub-processors) operates.

In Chapter IV we presented a more efficient structure to detect micro parallelism. We also introduced other structures which can be realized with closely coupled sub-processors, allowing SIMD and MIMD processing.

More details on the language and the modified

Polish notation introduced in Chapter III are given in Appendices I and II, respectively. In Appendix IV we include the listing of a program which was written to show the speed-up that can be achieved by the two-sub-processor system (described in Chapter III) over a uniprocessor. The appendix also includes comparative results obtained with the program. As mentioned earlier these data do not include the saving achieved with the language proposed for the processor. Furthermore, it should be noted that the speed-up is gained without having to preprocess the instruction stream in order to detect micro-parallelism.

In developing the required processor organizations we made assumptions with respect to the availability of instructions and data. This topic in itself requires further research to resolve the following problems:

i) The relative size of the buffers (IB and DB) which are used in the proposed organizations must be investigated. These buffers must be long enough to capture as many loops as possible in a program, but, on the other hand, long buffers require a longer delay in processing when they have to be rewritten. A solution to this problem would be to allow one or more instructions to bypass the buffer after it has been erased. However, this can be done only at the cost of a more complex system.

ii) Determine the number of buffers required to bypass the conditional jumps which would be inside the IB. This

would require an analysis of the speed-up achieved by by-passing different numbers of conditional jumps versus cost of the system in order to obtain an efficient design.

iii). Ascertain the size of an output buffer which can be used to temporarily store the results outputted by the processor.

These topics were not investigated here primarily because they are highly program dependent and their study becomes much more meaningful when one considers a particular application. For general purpose computers, the best that one can do is to consider statistics over a large number of different types of programs and adjust the number and the size of buffers according to results obtained.

Another topic of research would be to determine the number of sub-processors required to give maximum efficiency for the execution of a single stream of instructions. Even though we showed that for micromultiprocessing two sub-processors can give maximum efficiency, the choice of the number of sub-processors for a system should depend on the degree of parallelism (number of independent instructions, sets of instructions, etc.) present in a stream of N instructions, where N is the size of the Instruction Buffer. The number of sub-processors required in an organization would then be dependent on the particular application of the processor.

In this thesis we only considered basic sub-processors

that were identical. As mentioned in Chapter IV, the sub-processors could be special-purpose ones. In such a system, one would then have to investigate the design of a common control which could distribute a particular instruction to a particular sub-processor unit.

Many other research topics such as determining how much preprocessing should be done before the actual execution, how multiprogramming and multiprocessing would be implemented in such a system, and how I/O should be handled, still need to be investigated.

Concerning the efficiency of the system, it should be investigated how the processing speed is affected by its cost. The factors which have to be considered are the degree of parallelism in programs, the size of the buffers for look-ahead, the number of processors in the system, the cost of an individual sub-processor unit. It is the author's belief that the number of sub-processors required for a good performance is relatively small for the execution of general programs, but may become quite large for certain special purpose applications. However, with the advent of LSI technology, micro-processors on a chip are actually designed and it is expected that in the near future special purpose microprocessors will become available. Even though in most present-day computers multiprocessors are mainly used for multiprogramming and multiprocessing of programs at the task level, it is the author's belief that, with

the advent of LSI micromultiprocessors, the use of a number of closely coupled sub-processors will greatly increase the speed of a single instruction stream. A scheme similar to the one developed in this thesis would then prove most advantageous.

APPENDIX I

A Modified Polish Notation

The Polish notation was developed by the Polish logician J. Lukasieviez, and provides for the specification of an arithmetic expression parenthesis-free form. There are several forms of Polish notation. In this thesis we use a modified Postfix (trailing operator), or inverse Polish notation, in order to take advantage of the structure of the processor; that is, we transform the zero-address inverse Polish notation (operators do not have an address) into one including one-address codes.

In writing an arithmetic expression in this notation, the first and second character must be an operand (or letter). The other characters may then be either operands or operators. The first character or operand is associated with a LOAD instruction. The second character is then associated with a LOAD instruction if it is followed by another operand or to the following operator if it is followed by an operator. Similarily, each of the following operands is then associated with a LOAD instruction if followed by another operand, or with the operator if followed by an operator. For example, X = (A+B*C) may be written as ABC*+X = ; then, A would be associated with a LOAD instruction, B with another LOAD instruction, and C with the multiplication operator. Every time a LOAD instruction is thus met, the part of the expression preceding the LOAD instruction is then independent of what comes after it until two consecutive operators are met. Therefore, every time a load instruction is met, the result (partial result) of the previous part is sent (in order) in a stack S_1. Every time two consecutive operators are met, the second operator, OP_2, is associated with a zero-address instruction which then operates on all the partial results of S_1,

and stores the result in a stack S_2. Stack, S_2, receives the data of all OP_2's. All these data can then be operated on by the third operator, OP_3, of a sequence of 3 operators in a row, and so on. For example, in executing the following Polish expression

AB+CD-*EF+GH+/-X = the partial results of AB+ and CD- are pushed into S_1, the OP_2 (*) operates on the partial results in S_1, and the result of this is then sent to S_2. Then the partial results of EF+ and GH+ are sent into S_1, and OP_2(/), operates on them and pushes the result into S_2. OP_3(-) will then execute the subtraction of the 2 results in S_2 and send the result in S_3. S_3 will then contain the final answer.

If an operator OP_i does not find at least 2 operands in S_{i-1}, OP_i uses S_i as a source of operands. The operation OP_i is then executed on all the partial results (operands) in S_i. If S_i does not contain any operand, S_{i+1} is then used, and so on. The following expression illustrates this:- AB+CD-*EF-/X = . Here the partial results of AB+ and CD- are pushed into S_1. OP_2(*) then operates on these data and the result is sent into S_2. EF- is then executed and the result pushed into S_1. As OP_2(/) is to be executed, only one operand is present in S_1, and the second operand is then fetched from S_2. The last rule of execution in this notation is that, given a sequence L_1, L_2, L_3------L_n of consecutive LOAD instructions, the first LOAD is pushed into S_1. When the second LOAD L_2 comes in, L_1 is transferred into S_2 and L_2 is put into S_1. When L_3 comes in, L_1 is moved into S_3, L_2 into S_2, and L_1 into S_1, and so on. This is done to avoid confusion in associating the operands with the operators.

For example in ABCD+/*X = , A, B and C are associated with LOAD instructions and D with the add operator (+). When the expression is to be executed, A is in S_2 and B in S_1. The operation CD+ is then done and the result pushed into S_1. OP_2(/) then divides the two results stored in S_1 and pushes the result into S_2. OP_3(*) will then multiply the operands stored in S_2 together and will put the answer into S_3.

In the processor architecture presented in chapter III of this thesis, we use only one stack (RS) for partial results. Therefore we must limit the number of partial results in the stack that a zero-address operator operates on and restrict access to the stack to the sub-processor which made the last LOAD instruction. One way to reduce the number of operands that a zero-address instruction can operate on is to allow it to operate on only two operands stored in the RS. A second possibility is to use the address field of such instructions to specify the number of operands in RS that the operator operates on. For example, the expression AB+CD-*EF+GH+/-X = is then executed as follows (assuming that the address part on non-addressing instructions specifies the number of operands in RS which can be operated on). First the result of AB+ will be pushed into the stack followed by the result of CD-. The operation will then multiply the contents of the stack together and put the answer back into the stack. Then the results of EF+ and GH+ are pushed into the stack. With the address field of the non-addressing OP_2(/) operation set to 2 at compilation time, the instruction then divides only the first 2 elements off the top of the stack. The result is then pushed back into the stack and the OP_3(-) operation operates on the 2 results left in the stack.

It should be noted that when using this notation it is possible to reduce the number of operators required to specify an arithmetic expression. For example, to execute (A+B)(C+D)E+F)G+H), this can be written as AB+CD+EF+GH+* instead of AB+CD+EF+GH+***. When using more than one stack the OP_2(*) operation multiplies automatically all the partial results, which here will be in S_1, together. In the one-stack case, the OP_2(*) operation then has to specify that the operation is implied 3 times.

The sub-processor organization presented in chapter III uses only one stack. However, an organization with more than one stack is possible, and will allow more parallelism to be detected and executed automatically in a stream of instructions.

Branching Instructions

In order to allow an easy translation of H.L. branching instructions into machine language for the proposed sub-processor organization, we introduce the following symbols in the notations:

>	greater than
<	less than
≥	greater or equal
≤	smaller or equal
=	equal
→	branch to

As two partial results are available, a comparison can then be done and "X →" means branch to X if the condition is satisfied. The following example shows how Fortran IF statements can then be written in

this Polish notation.

1. Logical IF

   IF((A+B).GT.(C+D))GOTO N

   AB+CD+>N→

2. Arithmetic IF

   IF((A+B)*(C+D)) X_1, X_2, X_3

   AB+CD+*$X_1X_2X_3$ →

3. Computed GOTO

   GOTO (X_1, X_2, X_3 -----X_n) I

   I → $X_1X_2X_3$------X_n

APPENDIX II

The Language

The language used in this processor is a machine language derived directly from the modified Polish notation presented in this thesis. Since the processor can be adapted to any one-address machine language, we introduce here only the instructions which are particular to the processor.

As stated in Chapter 3, every time a LOAD instruction is met by a sub-processor, the previous partial result in the arithmetic expression is stored automatically into the RS. For example, X = (A+B)(C+D) is written AB+CD+*X = in inverse Polish notation and the corresponding instructions are LOAD A, ADD B, LOAD C, ADD D, SMPY, STORE X. Here, when one processor executes LOAD C, it sends a signal to the control unit indicating that the partial result (A+B) must be stored in the RS. Therefore it then becomes advantageous to develop instructions which will operate on these partial results. Table II.1 gives a list of proposed instructions which operate on the RS stack. It should be noted that here we only give some operations on RS. In fact, all the stack operations on it can be implemented here. Furthermore by having an address field specifying the number of operands on which the instruction operates, more complex instructions are then possible.

SADD	2 operands on top of the stack are added together.
SSUB	Second operand is subtracted from the top of the stack.
SMPY	2 operands on top of the stack are multiplied together.
SDIV	Top of the stack is divided by second operand.
JNES	2 operands on top of stack are compared. If they are not equal, the next memory location gives the address to branch to. If the condition is not satisfied, the instruction at the second memory location is executed.

JEQS	Same as JNES. Here the branching is done where the elements are equal.
JGTS	Same as JNES. Here the branching is done when the first operand is greater than the second.
JLTS	Same as JNES. Branching is done when the first operand is less than the second.
JGES	Same as JNES. Branching is done when the first operand is greater than or equal to the second.
JLES	Same as JNES. Branching is done when the first operand is smaller than or equal to the second.
JMPS	Jump to one of the 3 next addresses according to the sign of the operand (+, -, 0) on top of the stack.
JMPS XX	Branch according to the XX memory location following this instruction.

Table II-1 Proposed instructions.

The following examples show how the branching instructions of Appendix I in Polish notation are written in the proposed language:

1. AB+CD+>N→

   LOAD A, ADD B, LOAD C, ADD D, JGTS,N,

2. AB+CD+*$X_1X_2X_3$→

   LOAD A, ADD B, LOAD C, ADD D, SMPY, JMPS, X_1, X_2, X_3

3. I→$X_1X_2X_3$---X_n

   JMPS I, X_1, X_2, X_3

As the language used in a processor depends on the particular design (architecture), we judged it not necessary here to define a formal set of instructions. The intent was to show the type of instruction that this particular design allows.

APPENDIX III

RELATIVE TIMES (OF EXECUTION)

In this thesis we use the relative times of execution listed below. These figures correspond with a good approximation of execution times in a 36-bit computer. As these relative times vary greatly with the architecture and the electronic technology of the computer, in the actual evaluation of the two-sub-processor unit we worked with a number of different sets of relative time figures, as given in Appendix IV.

OPERATION	TIME UNITS (t)
Register and Stack transfer	2t
Addition (of two data)	20t
Subtraction (of two data)	20t
Multiplication (of two data)	150t
Division (of two data)	300t

Here t may be considered as an unit delay depending on the electronic technologie used in the computer. That is it may vary anywhere between 2ns to 30ns.

APPENDIX IV

EVALUATION OF THE EFFICIENCY OF THE TWO SUB-PROCESSOR SYSTEM

The program listed in this appendix accepts a valid arithmetic expression in Polish notation and gives execution times of the expression by two sub-processors and an uniprocessor. Here we neglected register transfers between the different operations as they are negligible compared to the actual operations and can be almost be masked when using two sub-processors (Chapter III section 6). The program is in PDP-8 assembler language.

```
.PALD
*OUT-S:PAR
*
*IN-S:PAR
*
*OPT-T

                        *252
0252  0226               226
0253  0024               24
                        *255
0255  0024               24
                        *257
0257  0454               454
                        *100
0100  1200  TYPE,       STYPE
0101  1206  READ,       SREAD
0102  1214  HEAD,       SEAD
0103  1223  OUT,        SOJT
0104  0000  T1,         0
0105  0000  ACC,        0
0106  0000  F1,         0
0107  0000  F2,         0
0110  0000  OPR2,       0
0111  0000  B1,         0
0112  0000  SIGN,       0
0113  0000  TEST,       0
0114  0000  PAR1,       0
0115  0000  T2,         0
0116  0000  FG,         0
0117  0000  ACC2,       0
0120  0000  CAR,        0
0121  0000  OPD,        0
0122  0000  PAR,        0
0123  0000  FLG,        0
0124  0000  BAR,        0
```

```
                        *200
0200  7300               CLA CLL
0201  3105               DCA ACC
0202  3117               DCA ACC2
0203  3120               DCA CAR
0204  3115               DCA T2
0205  3106               DCA F1
0206  3107               DCA F2
0207  3111               DCA B1
0210  3112               DCA SIGN
0211  3113               DCA TEST
0212  3114               DCA PAR1
0213  3116               DCA FG
0214  3121               DCA OPD
0215  3124               DCA BAR
0216  3104               DCA T1
0217  3122               DCA PAR
0220  3123               DCA FLG
0221  5777               JMP START
0377  0400
                        *400
0400  1377  START,      TAD (300
0401  4500               JMS I TYPE
0402  7300  INP,        CLA CLL
0403  4501               JMS I READ        / INPUT
0404  3104               DCA T1
0405  1104               TAD T1
0406  1376               TAD (7477
0407  7500               SMA
0410  5212               JMP .+2
0411  5220               JMP OPRD          /INPUT IS AN OPERAND
0412  7300               CLA CLL
0413  1375               TAD (7445
0414  1104               TAD T1
0415  7500               SMA
0416  5774               JMP 200           / ERROR IN INPUT
0417  5252               JMP CARR          /INPUT OK
0420  7300  OPRD,        CLA CLL
0421  1104               TAD T1
0422  1373               TAD (7526
0423  7450               SNA
0424  5256               JMP OPRR          /INPUT IS AN OPERAND(*)
0425  7300               CLA CLL
0426  1104               TAD T1
0427  1372               TAD (7525
0430  7450               SNA
0431  5256               JMP OPRR          /INPUT IS AN OPERAND (+)
0432  7300               CLA CLL
0433  1104               TAD T1
0434  1371               TAD (7523
0435  7450               SNA
0436  5256               JMP OPRR          /INPUT IS AN OPERAND (-)
0437  7300               CLA CLL
0440  1104               TAD T1
0441  1370               TAD (7521
```

```
0442  7450          SNA
0443  5256          JMP OPRR          /INPUT IS AN OPERAND (/)
0444  7300          CLA CLL
0445  1104          TAD T1
0446  1367          TAD (7503
0447  7450          SNA
0450  5766          JMP END /INPUT IS =,EXIT PROGRAM
0451  5774          JMP 200
0452  7300  CARR,    CLA CLL /INPUT IS A LETTER
0453  3121          DCA OPD
0454  2120          ISZ CAR
0455  5202          JMP INP
0456  7300  OPRR,    CLA CLL           / INPUT IS AN OPERAND
0457  1115          TAD T2
0460  1504          TAD I T1
0461  3115          DCA T2
0462  2121          ISZ OPD
0463  1120          TAD CAR
0464  1365          TAD (7777
0465  7440          SZA
0466  5336          JMP CARN1         / CAR NOT = 1
0467  7300  C1,     CLA CLL           / CAR =1
0470  1111          TAD B1
0471  1365          TAD (7777
0472  7450          SNA
0473  5301          JMP C2
0474  7300          CLA CLL
0475  1122          TAD PAR
0476  1365          TAD (7777
0477  7550          SPA SNA
0500  5305          JMP C3
0501  7300  C2,     CLA CLL           / B1 =1
0502  1117          TAD ACC2
0503  7440          SZA
0504  5320          JMP C5            / ACC2 NOT=0
0505  7300  C3,     CLA CLL           / ACC2=0
0506  1105          TAD ACC
0507  1504          TAD I T1
0510  3105          DCA ACC
0511  1504          TAD I T1
0512  7040          CMA
0513  7001          IAC
0514  3117          DCA ACC2
0515  7300  C4,     CLA CLL
0516  3120          DCA CAR
0517  5764          JMP XIN
0520  7300  C5,     CLA CLL           / ACC2 NOT = 0
0521  1117          TAD ACC2
0522  1504          TAD I T1
0523  3117          DCA ACC2
0524  1117          TAD ACC2
0525  7550          SPA SZA
0526  5315          JMP C4            / ACC2<0
```

```
0527  7300          CLA CLL
0530  1105          TAD ACC
0531  1117          TAD ACC2
0532  3105          DCA ACC
0533  3117          DCA ACC2
0534  3111          DCA B1
0535  5315          JMP C4
0536  7300  CARN1,   CLA CLL          / CAR NOT = 1
0537  1120          TAD CAR
0540  7450          SNA
0541  5763          JMP CARO         / CAR NOT = 0
0542  7300          CLA CLL
0543  1120          TAD CAR
0544  7040          CMA
0545  1121          TAD OPD
0546  7001          IAC
0547  7001          IAC
0550  7440          SZA
0551  5354          JMP C6
0552  2122          ISZ PAR
0553  5301          JMP C2
0554  7300  C6,     CLA CLL
0555  1120          TAD CAR
0556  1365          TAD (7777
0557  3124          DCA BAR
0560  1104          TAD T1
0561  3112          DCA SIGN
0562  5301          JMP C2,
0563  0600
0564  0732
0565  7777
0566  1054
0567  7503
0570  7521
0571  7523
0572  7525
0573  7526
0574  0200
0575  7445
0576  7477
0577  0300
                    *600
0600  7300  CARO,   CLA CLL
0601  1124          TAD BAR
0602  7440          SZA
0603  5777          JMP BNO
0604  1122          TAD PAR
0605  7440          SZA
0606  5214          JMP D1
0607  1114          TAD PAR1
0610  3122          DCA PAR
0611  3114  D0,     DCA PAR1
0612  3111          DCA B1
0613  5226          JMP PIMP
0614  7300  D1,     CLA CLL
```

```
0615  1122          TAD PAR
0616  1376          TAD (7777
0617  7440          SZA
0620  5226          JMP PIMP
0621  7300          CLA CLL
0622  1122          TAD PAR
0623  1114          TAD PAR1
0624  3122          DCA PAR
0625  5211          JMP D0
0626  7300  PIMP,    CLA CLL
0627  1122          TAD PAR
0630  7040          CMA
0631  1375          TAD (3
0632  3253          DCA TESS
0633  1104          TAD T1
0634  1374          TAD (7521
0635  7450          SNA
0636  5773          JMP E1          / OPERAND IS /
0637  7300          CLA CLL
0640  1122          TAD PAR
0641  0372          AND (0001
0642  7450          SNA
0643  5254          JMP EVEN
0644  7300  D2,     CLA CLL         / ODD
0645  1122          TAD PAR
0646  7001          IAC
0647  7010          RAR
0650  1376          TAD (7777
0651  3122          DCA PAR
0652  5261          JMP COMPL
0653  0000  TESS,    0
0654  7300  EVEN,    CLA CLL
0655  1122          TAD PAR
0656  7010          RAR
0657  1376          TAD (7777
0660  3122          DCA PAR
0661  7300  COMPL,   CLA CLL
0662  1122          TAD PAR
0663  7040          CMA
0664  3113          DCA TEST
0665  3122          DCA PAR
0666  2114          ISZ PAR1
0667  7300  C8,      CLA CLL
0670  1106          TAD F1
0671  7440          SZA
0672  5304          JMP E2
0673  1253          TAD TESS
0674  7450          SNA
0675  5304          JMP .+7
0676  7300          CLA CLL
0677  1115          TAD T2
0700  1504          TAD I T1
```

```
0701  2253          ISZ TESS
0702  5300          JMP .-2
0703  3115          DCA T2
0704  7300  E2,     CLA CLL
0705  1111          TAD B1
0706  1376          TAD (7777
0707  7440          SZA
0710  5314          JMP D3
0711  1117          TAD ACC2
0712  7440          SZA
0713  5343          JMP D5
0714  7300  D3,     CLA CLL
0715  1105          TAD ACC
0716  1504          TAD I T1
0717  2113          ISZ TEST
0720  5316          JMP .-2
0721  3105          DCA ACC
0722  1504          TAD I T1
0723  7040          CMA
0724  7001          IAC
0725  3117          DCA ACC2
0726  7300  D4,     CLA CLL
0727  7001          IAC
0730  3111          DCA B1
0731  5771          JMP INP
0732  7300  XIN,    CLA CLL
0733  1107          TAD F2
0734  1376          TAD (7777
0735  7440          SZA
0736  5771          JMP INP
0737  3107          DCA F2
0740  1110          TAD OPR2
0741  3104          DCA T1
0742  5770          JMP C1
0743  7300  D5,     CLA CLL          / ACC2,NOT=0
0744  1117          TAD ACC2
0745  1504          TAD I T1
0746  2113          ISZ TEST
0747  5345          JMP .-2
0750  3117          DCA ACC2
0751  1117          TAD ACC2
0752  7550          SPA SZA
0753  5326          JMP D4
0754  7300          CLA CLL
0755  1105          TAD ACC
0756  1117          TAD ACC2
0757  3105          DCA ACC
0760  3117          DCA ACC2
0761  5326          JMP D4
0770  0467
0771  0402
0772  0001
0773  1000
0774  7521
0775  0003
0776  7777
0777  1014
```

```
                    *1000
1000  7300  E1,     CLA CLL
1001  1122          TAD PAR
1002  1377          TAD 7776
1003  3122          DCA PAR
1004  1106          TAD F1
1005  7450          SNA
1006  5776          JMP COMPL
1007  7300          CLA CLL
1010  1122          TAD PAR
1011  7001          IAC
1012  3122          DCA PAR
1013  5776          JMP COMPL
1014  7300  BNO,     CLA CLL          / BAR NOT=0
1015  1104          TAD T1
1016  7040          CMA
1017  7001          IAC
1020  1112          TAD SIGN
1021  7440          SZA
1022  5234          JMP D6
1023  2106          ISZ F1           / OPR =SIGN
1024  7300          CLA CLL
1025  1124          TAD BAR
1026  7040          CMA
1027  7001          IAC
1030  1121          TAD OPD
1031  7440          SZA
1032  5775          JMP INP
1033  5250          JMP D7
1034  7300  D6,     CLA CLL
1035  1106          TAD F1
1036  7450          SNA
1037  5774          JMP C1
1040  7300          CLA CLL
1041  7001          IAC
1042  3107          DCA F2
1043  1104          TAD T1
1044  3110          DCA OPR2
1045  1112          TAD SIGN
1046  3104          DCA T1
1047  5774          JMP C1
1050  7300  D7,     CLA CLL
1051  1106          TAD F1
1052  3122          DCA PAR
1053  5773          JMP PIMP
1054  7300  END,    CLA CLL
1055  2116          ISZ FG
1056  7300          CLA CLL
1057  1105          TAD ACC
1060  0372          AND (7000
1061  7006          RTL
1062  7006          RTL
1063  4503          JMS I OUT
1064  7300          CLA CLL
1065  1105          TAD ACC
```

```
1066  0371          AND (0700
1067  7012          RTR
1070  7012          RTR
1071  7012          RTR
1072  4503          JMS I OUT
1073  7300          CLA CLL
1074  1105          TAD ACC
1075  0370          AND (0070
1076  7012          RTR
1077  7010          RAR
1100  4503          JMS I OUT
1101  7300          CLA CLL
1102  1105          TAD ACC
1103  0367          AND (0007
1104  4503          JMS I OUT
1105  7300          CLA CLL
1106  1116          TAD FG
1107  7040          CMA
1110  7001          IAC
1111  7001          IAC
1112  7440          SZA
1113  5322          JMP NST
1114  1366          TAD (254
1115  4500          JMS I TYPE
1116  7300          CLA CLL
1117  1115          TAD T2
1120  3105          DCA ACC
1121  5254          JMP END
1122  4502  NST,     JMS I HEAD
1123  5765          JMP 200
1165  0200
1166  0254
1167  0007
1170  0070
1171  0700
1172  7000
1173  0626
1174  0467
1175  0402
1176  0661
1177  7776
                    *1200
1200  0000  STYPE,  0
1201  6041          TSF
1202  5201          JMP .-1
1203  6046          TLS
1204  7300          CLA CLL
1205  5600          JMP I STYPE
1206  0000  SREAD,  0
1207  6031          KSF
1210  5207          JMP .-1
1211  6036          KRB
1212  6046          TLS
1213  5606          JMP I SREAD
```

```
1214  0000  SEAD,   0
1215  7300          CLA CLL
1216  1377          TAD (215
1217  4500          JMS I TYPE
1220  1376          TAD (212
1221  4500          JMS I TYPE
1222  5614          JMP I SEAD
1223  0000  SOUT,   0
1224  1375          TAD (260
1225  4500          JMS I TYPE
1226  5623          JMP I SOUT
1375  0260
1376  0212
1377  0215

ACC     0105
ACC2    0117
BAR     0124
BNO     1014
B1      0111
CAR     0120
CARN1   0536
CARO    0600
CARR    0452
COMPL   0661
C1      0467
C2      0501
C3      0505
C4      0515
C5      0520
C6      0554
C8      0667
D0      0611
D1      0614
D2      0644
D3      0714
D4      0726
D5      0743
D6      1034
D7      1050
END     1054
EVEN    0654
E1      1000
E2      0704
FG      0116
FLG     0123
F1      0106
F2      0107
HEAD    0102
INP     0402
NST     1122
OPD     0121
OPRD    0420
OPRR    0456
OPR2    0110
```

```
OUT     0103
PAR     0122
PAR1    0114
PIMP    0626
READ    0101
SEAD    1214
SIGN    0112
SOUT    1223
SREAD   1206
START   0400
STYPE   1200
TESS    0653
TEST    0113
TYPE    0100
T1      0104
T2      0115
XIN     0732
```

EFFICIENCY OF THE TWO SUB-PROCESSOR SYSTEM

The speed-up achieved when using two sub-processors instead of one to execute arithmetic expressions is approximatively 1.508. The next page shows an example of output of the program. For these expressions the average speed-up is 1.48 . The execution time of the different operations was then changed and the same expressions fed to the program (p111). The resulting speed-up becomes 1.49 .

```
OUT     0103
PAR     0122
PARI    0114
PIMP    0626
READ    0101
SEAD    1214
SIGN    0112
SOUT    1223
SREAD   1206
START   0400
STYPE   1200
TESS    0653
TEST    0113
TYPE    0100
TI      0104
T2      0115
XIN     0732
```

EFFICIENCY OF THE TWO SUB-PROCESSOR SYSTEM

The speed-up achieved when using two sub-processors instead of one to execute arithmetic expressions is approximatively 1.503. The next page shows an example of output of the program. For these expressions the average speed-up is 1.48 . The execution time of the different operations was then changed and the same expressions fed to the program (p111). The resulting speed-up becomes 1.49 .

Execution time of operations: + = 20 t, - = 20 t, * = 150t, / = 300 t. (Outputs are given in octal)

```
@AB+CD++X=0050,0074
@AB+CD+*X=0252,0276
@AB*CD*+X=0252,0500
@ABCD+++X=0050,0074
@ABCD***X=0454,0702
@ABCD//*X=1356,1356
@ABCD/*/X=1356,1356
@AB+CD/+X=0500,0524
@AB-CD-/AB*EF*/+X=1200,1700
@AB+C*D/AB+CD+*+X=1426,1452
@AB-CD-/EF*GH*-X=0524,1224
@AB+CD-FG*AB+*X=0726,1224
@ABC//AB/X=1130,1604
@AB-CD-EF+GH+*X=0524,1022
@AB*CD*/AB+C*D+*X=0726,1452
@AB-C+D/EF+CD**X=1200,1452
@AB*AB-C+EF*GH**X=1200,1654
@AB/AB//AB*CD**-X=1402,2532
@AB-CD+/EF+GH/+*X=0776,1476
@ABCDE++++EF*GH**X=0550,1022
@AB*CD-E*/X=0726,1154
@AB*CD-EF-*X=0702,0752
@AB*CD*EF*GH**X=1130,2032
@ABCDE****X=0702,1130
@AB*CD*E*+X=0500,0726
@AB-C*EF-G*/X=0752,1200
@AB-CD-*EF+G+HI*-X=0346,0620
@AB*AB*C*DE/FG-*X=1402,2304
@AB/CD/EF/GH/+X=1200,2354
@AB*CD/EF//X=2260,2506
@AB+C-D/AC**X=0752,1200
@AB*CD-/AC*AF*-FG--X=0776,1522
@AB+CD+/AB*F-G++X=0524,1046
@AB/CD/EF/GH*IJ**X=2032,3410
@AB-C/EF+CD*E-+X=0550,1046
@AB/AC//AC-AC--AB*DF*--X=1320,2450
@AB+CD+/AB+CD-E*G++X=0500,1072
@AB/CD+E-F/D+AB*CD*/X=3004,3504
@AB+CD-EF/GH**AB+CD+**X=1452,2400
@AB+CD--AB+CD---X=0120,0214
@AB/CD/EF/GH/IJ/KL/*X=2506,4766
@AB+C+D-FG*X*AB**X=1200,1452
@AB/CD+F-*AB-CD-+*X=1130,1274
@AB-CD-*AB-CD+/-X=0574,1046
@AB+CD-/AB+CD-/AB+CD+*X=1022,1546
@AB/CD/+AB+CD*/FG/HI++-X-X=1700,2722
@AB*CD**AB*C*D*+X=0726,1630
@AB+CD/EF*G*X=0702,1154
@AB+CD+EF++X=0120,0144
@AB+C+D+AB/C/D//AB*C*+X=1700,3054
```

Execution time of operations: + = 25 t, - = 25 t, * = 125t

/ = 300 t. (Outputs are given in octal)

```
@AB+CD+*X=0062,0113
@AB+CD+*X=0226,0257
@AB*CD*+X=0226,0423
@ABCD+++X=0062,0113
@ABCD***X=0372,0567
@ABCD//*X=1325,1325
@ABCD/*/X=1325,1325
@AB+CD/+X=0505,0536
@AB-CD-/AB*EF*/+X=1130,1635
@AB+C*D/AB+CD+*+X=1325,1356
@AB-CD-/EF*GF*-X=0505,1161
@AB+CD-FG*AB+*X=0620,1077
@ABC//AB/X=1130,1604
@AB-CD-EF+GH+*X=0454,0733
@AB*CD*/AB+C*D++X=0702,1356
@AB-C+C/EF+CD**X=1130,1356
@AB*AB-C+EF*GH**X=1046,1440
@AB/AB//AB*CD**-X=1274,2424
@AB-CD+/EF+GH/+*X=0764,1471
@ABCDE++++EF*GH**X=0505,0733
@AB*CD-E*/X=0702,1077
@AB*CD-EF-*X=0567,0651
@AB*CD*EF*GH**X=0764,1553
@ABCDE****X=0567,0764
@AB*CD*E*+X=0423,0620
@AB-C*EF-G*/X=0733,1130
@AB-CD-*EF+G+HI+-X=0226,0423
@AB*AB*C*DE/FG-*X=1274,2063
@AB/CD/EF/GH/+X=1212,2373
@AB*CD/EF//X=2260,2455
@AB+C-D/AC**X=0733,1130
@AB*CD-/AC*AF*-FG--X=0733,1440
@AB+CD+/AB*F-G+*X=0536,1046
@AB/CD/EF/GH*IJ**X=1717,3162
@AB-C/EF+CD*E-+X=0567,1046
@AB/AC//AC-AC--AB*DF*--X=1274,2424
@AB+CD+/AB+CD-E*G*+X=0505,1077
@AB/CD+E-F/D+AB*CD*/X=2734,3441
@AB+CD-EF/GH**AB+CD+**X=1356,2176
@AB+CD--AB+CD---X=0144,0257
@AB/CD/EF/GH/IJ/KL/*X=2373,4571
@AB+C+D-FG*X*AB**X=1046,1274
@AB/CD+F-*AB-CD-+*X=1046,1243
@AB-CD-*AB-CD+/-X=0620,1046
@AB+CD-/AB+CD-/AB+CD+*X=1015,1553
@AB/CD/+AB+CD*/FG/HI++-X-X=1717,2734
@AB*CD**AB*C*D*+X=0620,1407
@AB+CD/EF*G*X=0651,1077
@AB+CD+EF++X=0144,0175
@AB+C+D+AB/C/D//AB*C*+X=1717,3016
```

REFERENCES

1. Bell and Newell, Computer Structures: Reading and Examples, Computer Science Series, Mc Graw-Hill, 1971

2. F.J. Hill and G.R. Peterson, Digital Systems: Hardware Organization and Design, John Wiley & Son, 1973

3. H. Lorin, Parallelism in Hardware and Software, Prentice Hall, 1972

4. L.C. Higbie, "Supercomputer Architecture", Computer, vol.6, no.12, pp.48-58, Dec.1973

5. M.J. Flynn, "Some Computer Organizations and Their Effectiveness", IEEE Trans. Comput., vol.C-21, no.9, pp.948,960, Sept.1972

6. Hewlett Packard, HP 3000 Reference Manual, 1971

7. Burroughs Corp., B 7700 Reference Manual,

8. G.S. Tjaten and M.J. Flynn, "Detection and Parallel Execution of Independent Instructions", IEEE Trans. Comput., vol.C-19, no.10, pp.889-895, Oct.1970

9. J.L. Rosenfeld and R.D. Villani, "Micromultiprocessing:"An Approach to Multiprocessing at the Level of Very Small Tasks", IEEE Trans. Comput., vlo C-22, no.2, pp.149-153, Feb. 1973

10. J.M. Kurtzberg and R.D. Villani, "A Balanced Pipelining Approach to Multiprocessing on an Instruction Stream Level", IEEE Trans. Comput., vol.C-22, no.2, pp.143-148, Feb. 1973

11. C.V. Ramamoorthy, J.H. Park and H.F. Li, "Compilation Techniques for Recognition of Parallel Processable Task in Arithmetic Expressions", IEEE Trans. Comput., vol.C-22, no.11, pp.986-998, Nov. 1973

12. M.J. Flynn and A. Podvin, "Shared Ressource Multiprocessing", Computer, pp.20-28, Apr.1972

13. D.J. Kuck et al.,"Measurement of Parallelism in Ordinary Fortran Programs", Computer, pp.37-46, 1974

14. R. Brent, D. Kuck and K. Maruyama, "The Parallel Evaluation of Arithmetic Expressions Without Divisions", IEEE Trans. Comput., vol.C-22, no.1, Jan. 1973

15. K. Maruyama, "On the Parallel Evaluation of Polynomials", IEEE Trans. Comput., vol.C-22, no.1, pp. 2-5, Jan.1973

16. E.M. Riseman and C.C. Foster, "The inhibition of Parallelism by Conditional Jumps", IEEE Trans. Comput., vol.C-21, no.12, pp.1405-1411, Dec.1972

17. C.C. Foster and E.M. Riseman, "Percolation of Codes to Enhace Parallel Dispatching and Execution", IEEE Trans. Comput., vol.C-21, no.12, pp.1411-1415, Dec. 1972

18. G. Garon and M. Krieger, "Address Driven Microprogramming" Tech. Report 73-19, Dept. of Elec. Eng. University of Ottawa, 1973

19. M.J. Flynn, and R.S. Rosin, "Microprogramming: An Introduction and a Viewpoint", IEEE Trans. Comput. vol.C-20. no.7, pp.727-731, July 1971

20. S.S. Husson, Microprogramming Principles and Practices, Englewood Cliffs, N.J.: Prentice-Hall, 1970

VITA

Name: Gilles Garon

Place of Birth: Rimouski, Quebec, Canada

Date of Birth: Feb. 8, 1949.

Education: High-School Paul-Hubert, Rimouski

University U. of Ottawa

B.A.Sc. (E.E.),1973