1 THE DEVELOPMENT OF PARALLEL MACHINES

As we have seen from the preceding section, the introduction of parallelism into computer implementation occurred almost everywhere where possible. And so there is no simple and single line of development leading to modern parallel processing machines. Before considering this development further, therefore, it is important to recognize that many levels exist where parallelism may be used. Let us summarize the main categories:

i) *Within functional units:* Arithmetic, logical and other operations can be implemented in bit-serial mode, parallel by bit groups, or on whole operands concurrently. Clearly there are limits to what can be done, and there is the question of cost and reliability of the extra hardware involved. However, this category does not generally affect the way in which a problem is formulated, although it does determine the speed of execution. Another area of parallelism in this category is the concurrent access to several interleaved memory units.

ii) *Within processing elements:* At this level, the most obvious form of concurrency is between different operations executing in parallel on different operands. For this to be exploited, the problem formulation (the detailed machine coding in this case) needs to be appropriate to the PE organization. Another kind of concurrency is where there is only a single instruction at one time, but the functional unit to the instruction refers, for example a multiplier, may be able to process a stream of operands in an overlapped or pipelined fashion. Pipelining is a very powerful feature indeed, which we will discuss later. It can give significant improvements in execution speed even for simple sequences of entirely serial code.

iii) *Within uniprocessing computers:* Even with only a single processor, there are many activities that can proceed concurrently. An obvious example already mentioned is memory access, and another is I/O. In some ways, the concurrent operation of input-output is just a detail of system organization, since it is a way of keeping the operational memory full (or to empty it); if main memory were big enough, many of the I/O problems would vanish. However, when several uniprocessors need to inter-communicate in order concurrently to execute different tasks of the same job, then parallel I/O or inter-memory communication is a vital requirement — and a difficult problem to solve.

iv) *Many-processor systems:* An obvious category of concurrency is where a computer system contains several (often many) processors, either sharing main memory or not, and inter-communicating by some means. In the most usual example, the separate processors are used to execute quite independent jobs. This is the well-known multi-processor architecture, and the concurrency here is used not to speed up the execution of individual jobs, but the global throughput of the whole system. Much more interesting from the standpoint of our discussion here is where the different processors are used either to execute separate cooperating tasks within a given job or where even separate code sequences within a given program are executed concurrently. There is a very great difference here between computers where all the processing elements are identical, and operate in lock step, and where they differ.

The way in which all the different levels of con currency are exploited differs from one category to another. In the case of functional parallelism, it is necessary to structure the machine code appropriately, and so the characteristics of any compiler used are of paramount importance. At the other extreme, in multi-processing systems, it is the operating system which does the job of allocating resources among concurrently executing jobs and between processing and I/O. In all other categories, of central importance is the structure of the job itself and so the choice of algorithms used becomes the significant issue. We shall return to this most important question later.

At the present time, several machines exist whose architecture and implementation have been expressly chosen to provide massive parallel-processing capability. We shall describe some of them later. However, in the previous two decades, a number of earlier computer systems were developed where concurrency was a central feature in implementation, and where nearly all the basic ideas on which modern machines are based were tried out. In these machines, it is not always easy to separate out any one feature as being most significant, since several levels of concurrency were often implemented in the same system. Nevertheless, there were perhaps three separate lines of development whose evolution has been more or less distinct, namely pipelining, the use of many functional units within a uniprocessor, and systems of many cooperating processors. And there is one other category which might be mentioned separately, that of special-purpose computing systems. Let us briefly consider these four categories.

1.1 Pipelining

In many computational processes, the total process can be made to take place in a number of discrete steps or segments. For example, the processing of an instruction might be segmented into the separate phases of fetching the instruction, decoding, calculating the operand addresses, fetching the operands and executing the instruction. In **Fig.1** the difference is shown diagrammatically between some unsegmented process **P** and the same process separated into four sequential segments. There is no reason why the total time **T** for an unsegmented process **P** should differ from the sum of the separate segment times $t_1 + t_2 + t_3 + t_4$. The idea of pipelining is simply that if all the segments **S** are implemented by physically separate sub-units, then they can operate together, and several processes may proceed concurrently in an overlapped fashion. **Fig.1(b)** represents thus a processing pipe in which there may be up to four concurrent processes **P** at any given instant.

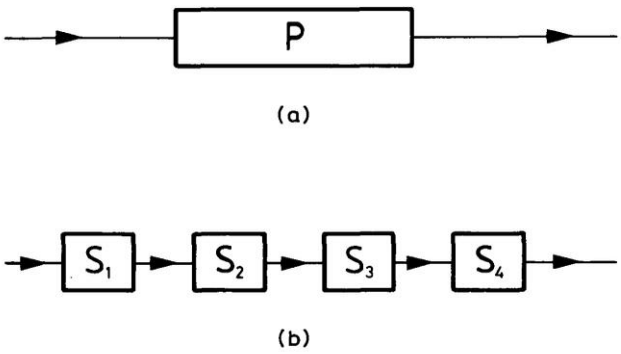

Fig. 1: A process P in usegmented form (a) and suitable for pipelining (b)

The advantage of a pipeline can be seen from **Fig.2** which shows several processes P in concurrent execution. In the first clock period only the first process is executing, but by clock period 5 there are four processes of complete processes from a full pipeline is one for every clock period, a potential improvement of **m** compared with unsegmented processing, where **m** is the number of segments.

The example above is that of pipelining in instruction processing, and one of the earliest implementations of this idea (in rudimentary form) was in the *STRETCH* computer. But pipelining can also be used with even greater effectiveness in arithmetic processes. For example, a floating-point addition may be segmented into, say, four execution phases: compare exponents,

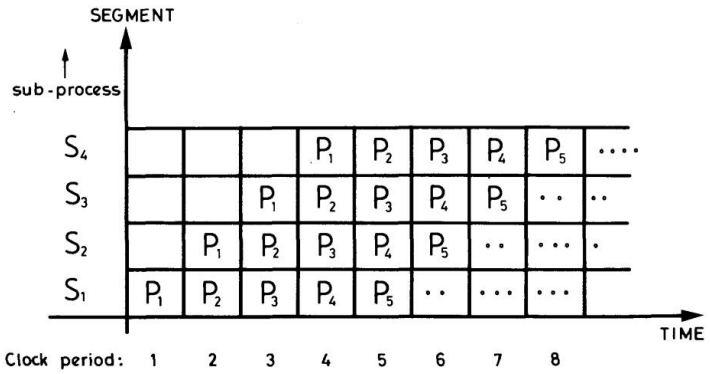

Fig. 2: Timing diagram for a 4-segment pipeline

normalize mantissa, add, normalize result. There is then in principle no limit to the number of operands that can be processed in an overlapped way at the maximum pipeline rate, an encouraging prospect, bringing to mind the alliterative words of Alexander Pope:

"And the smooth stream in smoother numbers flows"

An Essay on Criticism

A machine which used pipelining in this fashion for instruction processing was the Ferranti *ATLAS* computer developed jointly with Manchester University and first delivered in 1963. In this machine, the floating-point addition time of 6 microseconds for a single pair of operands, could be reduced to 1.6 microseconds on average for a long stream.

Pipelining has been employed at more than one level in machines, and in different configurations. In the preceding examples the method has been used either to overlap the processing of successive instructions (in the I-unit) or the execution of successive operations of the same kind on different operands (in a E-unit). Of course, one can have both, and this was probably done for the first time in the *IBM 360/91* in the mid-sixties. However, a very interesting application of pipelining of a somewhat different kind, also used in the *360/91*, is at the micro-system level, for example within a single arithmetic operation. Here, for example in floating-point multiplication or division, the function can be implemented as a successive or iterative execution of a series of microfunctions; in the case of multiplication, the microfunctions might be additions (or subtractions) of partial summands deriving from using one or more bits of the multiplier. Then the same hardware segments are used cyclically to process different parts of operands; but only one instruction is being

executed at any given time. So if, for example in 16-bit multiplications, 4-bit groups are being processed, one might require four cycles through a pipeline of 5 segments to generate the final product. Such a scheme is shown in **Fig.3**, where **S** decodes each 4-bit group and provides 3 appropriately scaled multiplicands to be summed (with proper signs) in a carry-save adder **CSA-1**. In further cycles partial sums of accumulated operands are added in **CSA-2** and **CSA-3**, but at the last cycle (i.e. after 8 minor cycles) the final outputs of **CSA-3** are summed in a carry-propagating adder **A** to give a 32-bit product at the output.

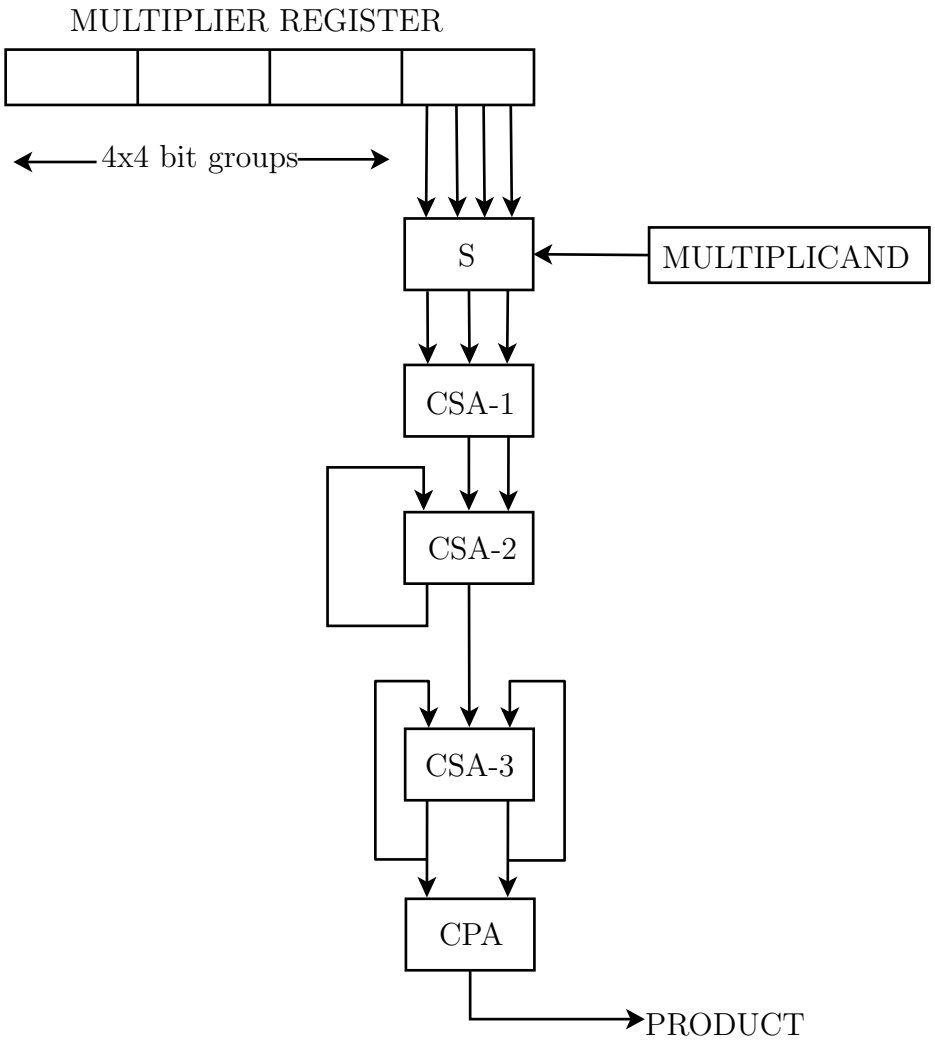

Fig. 3: An iterative pipeline multiplier

We should remark here that the iterative scheme of **Fig.3** for pipeline multiplication is a development of an early proposal by *Wallace*, tu use carry-save adders interconnected in a binary tree to perform all additions in successive levels until only two numbers remain; these are then added finally in a carry-propagating adder. The essential feature which enables pipelining to proceed here is that the output of each level should not change during the period of each minor cycle for which this output is needed as an input to the successive level. This feature is provided by *"latching"* the output of each **CSA**, a technique first introduced by *Earle*.

A further remark to make is that iterative pipelining within a single arithmetic function is not restricted to the simple operations of addition and multiplication. There is an elegant extension of the technique to the evaluation of more complicate functions, such as exponential, logarithm, ratio and square root. This is based on the iterative co-transformations of a number pair (x, y) such that some chosen bivariate function $F(x, y)$ remains invariant; in this process x is directed towards x_n, where y_n in the required result. The pipelining technique has been employed here to implement a functional unit of two segments, one for successive evaluation of x, the other for y, the process continuing iteratively until convergence (usually when $x_n = 1$). For example, the *360/91* used such a scheme for division.

So far in our discussion of pipelining, we have considered machines where pipelining has been used to speed up arithmetic operations on sequences of operands not necessarily associated with one another. Moreover, the pipeline units themselves have been auxiliary to non-pipelined functional units, they have been fixed in the operations they can perform and also, generally, of fixed execution time. They are termed static uni-functional pipelines.

More recently, actually since about 1970, a number of machines have appeared where pipelining is a central rather than auxiliary feature of the architecture, and where sometimes the pipelines can be used to implement a range of different functions. Perhaps the best example of such an early implementation for general purposes is the *STAR-100* computer, manufactured by *CDC* and first delivered in 1974.