UNIT – II
Principles of Pipelining and Vector Processing

Introduction

Pipeline is similar to the assembly line in industrial plant. To achieve pipelining one must divide the input process into a sequence of sub tasks and each of which can be executed concurrently with other stages. The various classification or pipeline line processors are:

- arithmetic pipelining,
- instruction pipelining,
- processor pipelining

Pipelining

Pipelining offers an economical way to realize temporal parallelism in digital computers. To achieve pipelining, one must subdivide the input task into a sequence of subtasks, each of which can be executed by a specialized hardware stage.

- Pipelining is the backbone of vector supercomputers
- Widely used in application-specific machine where high throughput is needed
- Can be incorporated in various machine architectures (SISD, SIMD, MIMD,)

Easy to build a powerful pipeline and waste its power because:

- Data cannot be fed fast enough
- The algorithm does not have inherent concurrency.
- Programmers do not know how to program it efficiently.

Types of Pipelines

- Linear Pipelines
- Non-linear Pipelines
- Single Function Pipelines
- Multifunctional Pipelines
    - Static
    - Dynamic

Principles of Linear Pipelining

A. Basic Principles and Structure

Let T be a task which can be partitioned into K subtasks according to the linear precedence relation:

T= {T1, T2... Tk}; i.e.

a subtask Tj cannot start until {Ti " i < j } are finished. This can be modeled with the linear precedence graph:

A linear pipeline (No Feedback!) can always be constructed to process a succession of subtask with a linear precedence graph.

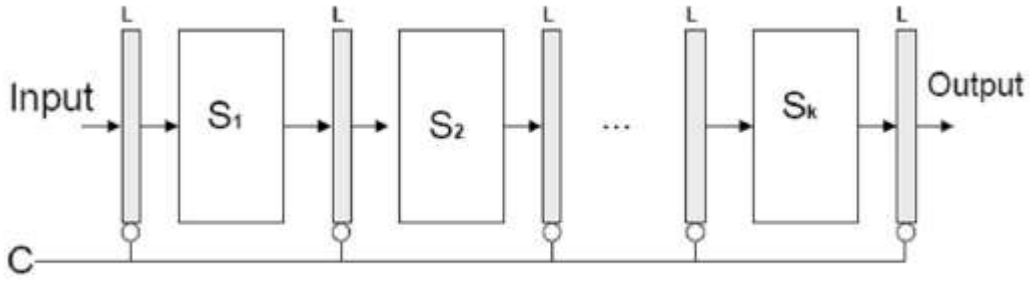


Figure 11.1 Basic Structure and Control of a Linear Pipeline

Processor (L=latch, C=clock, Si=the ith stage.)

- Stages are pure combinational circuits used for processing.
- Latches are fast registers to hold the intermediate data between the stages.
- Informational flow is controlled by a common clock with some clock period “t”, and the pipeline runs at a frequency of 1/t
- t is selected as: t = MAX{ti} + tL = tM + tL where, ti= propagation delay of stage Si tL=latch delay
- Pipeline clock period is controlled by the stage with the max delay.
- Unless the stage delays are balanced, one big and slow stage can slow down the whole pipe

Space-Time Diagrams

Consider a four stage linear pipeline processor and a sequence of tasks.
T1, T2...

Where each task has 4 subtasks (1st subscript if for task, 2nd is for subtask) as follows:

T1=> {T11, T12, T13, T14}

T2=> {T21, T22, T23, T24}

…
T2=> {Tn1, Tn2, Tn3, Tn4}

A space-time diagram can be constructed to illustrate how the overlapping execution of the tasks as follows

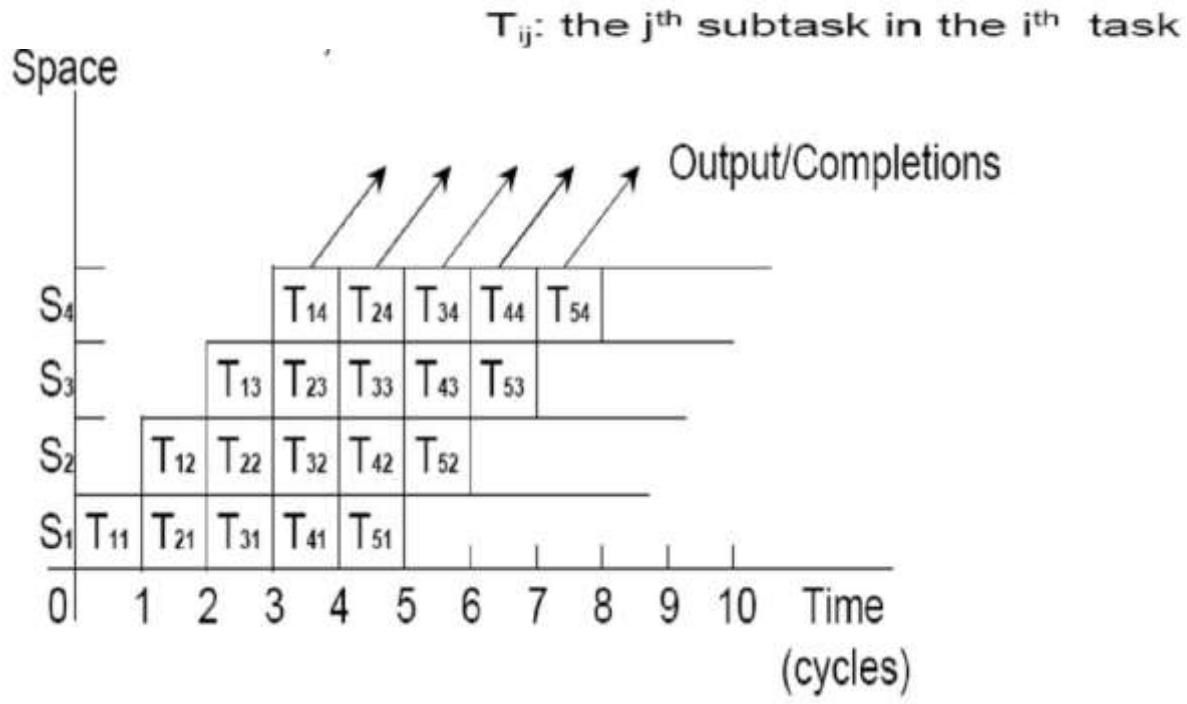


Figure 11.2 Space - Time Diagram

Performance Measure for Linear Pipelined Processors

Speedup (Sk) - the speedup of a k-stage linear pipeline processor (over an equivalent non-pipelined)

To illustrate the operation principles of a pipeline computation, the design of a pipeline floating point adder is given. It is constructed in four stages.

The inputs are:

$A = a \times 2_p$

$B = b \times 2_q$

Where a and b are 2 fractions and p and q are their exponents and here base 2 is assumed.

To compute the sum

$C = A + B = c \times 2_r = d \times 2_s$

Operations performed in the four pipeline stages are specified.

1. Compare the 2 exponents p and q to reveal the larger exponent r =max (p, q) and to determine their difference t =p-q
2. Shift right the fraction associated with the smaller exponent by t bits to equalize the two components before fraction addition.
3. Add the preshifted fraction with the other fraction to produce the intermediate sum fraction c where $0 <= c <1$.
4. Count the number of leading zeroes, say u, in fraction c and shift left c by u bits to produce the normalized fraction sum $d = c \times 2_u$, with a leading bit 1. Update the large exponent s by subtracting s= r – u to produce the output exponent

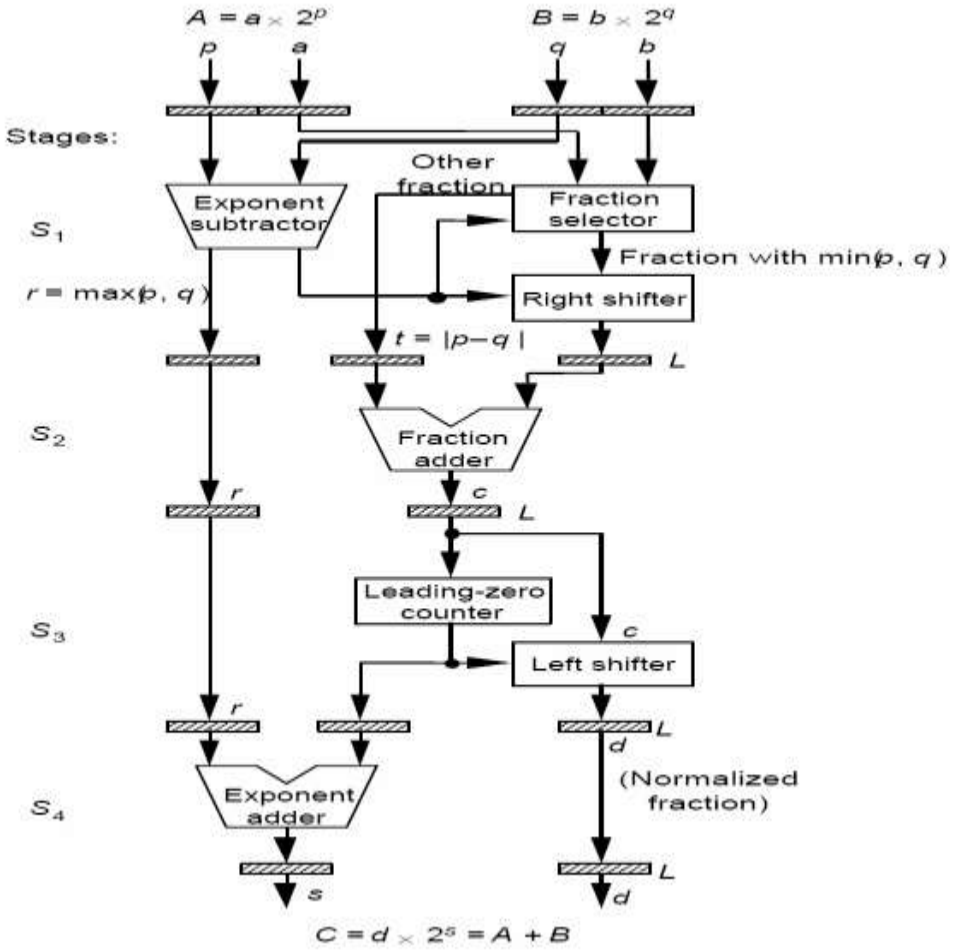


Figure 11.3 A pipelined floating-point adder with four processing stages

Efficiency "E" - the ration of the busy time span over the overall time span

Throughput "W" - is the number of tasks that can be completed, by the pipeline, per unit time
.

Classification of Pipeline Processors

Arithmetic Pipelining

The arithmetic and logic units of a computer can be segmentized for pipeline operations in various data formats. Well known arithmetic pipeline examples are

- Star 100
- The eight stage pipes used in TI-ASC

    - 14 pipeline stages used in Cray-1
    - 26 stages per pipe in the cyber-205

Instruction Pipelining

The execution of a stream of instructions can be pipelined by overlapping the execution of the current instruction with fetch, decode and operand fetch of subsequent instructions. This technique is known as instruction look-ahead.

Fig:

Processor pipelining

This refers to pipelining processing of the same data stream by a cascade of processors each of which processes a specific task. The data stream passes the first processor with results stored in memory block which is also accessible by the second processor. The second processor then passes the refined results to the third and so on.

Fig:

The principle pipeline classification schemes are:

Unification Vs Multifunction pipelines

A pipeline with fixed and dedicated function such as floating adder is called unifuncitonal pipeline. Eg : Cray-1

A multifunction pipe may perform different functions, either at different times or at the same time, by interconnecting different subsets of stages in the pipeline.

E.g.: TI-ASC

Static Vs Dynamic Pipeline

A static pipeline has only one functional configuration at a time.

A dynamic pipeline permits several functional configurations to exist simultaneously.

Scalar Vs Vector Pipelines

A scalar pipeline processes a sequence of scalar operands under the control of DO loop.

A vector pipeline is designed to handle vector instructions over vector operand

INSTRUCTION AND ARITHMETIC PIPELINES

Before studying various pipeline design techniques and examining vector pr'ocessing requirements, we need to understand how instructions can be overlap: executed, and how repeated arithmetic computations can be done with pipe lining. Instruction pipelining is illustrated with the designs in the IBM 360,

Arithmetic pipelining will be studied in detail with four design examples multiple-number addition, floating-point addition, multiplication, and division . Finally, multifunction-pipeline designs and array pipelining for-matrix arith will be introduced.

Design of Pipelined Instruction Units

We will study the instruction pipeline in the IBM System/360 Model 91 as a ing example. The IBM 360/91 incorporates a high degree of pipelining in instruction preprocessing and instruction execution. It is a 32-bit machine s designed for scientific computations in either fixed-point or floating-point formats. Multiple pipeline functional units are built into the system to allow arithmetic computations in either data format.

PRINCIPLES OF PIPELINING AND VECTOR PROCESSING

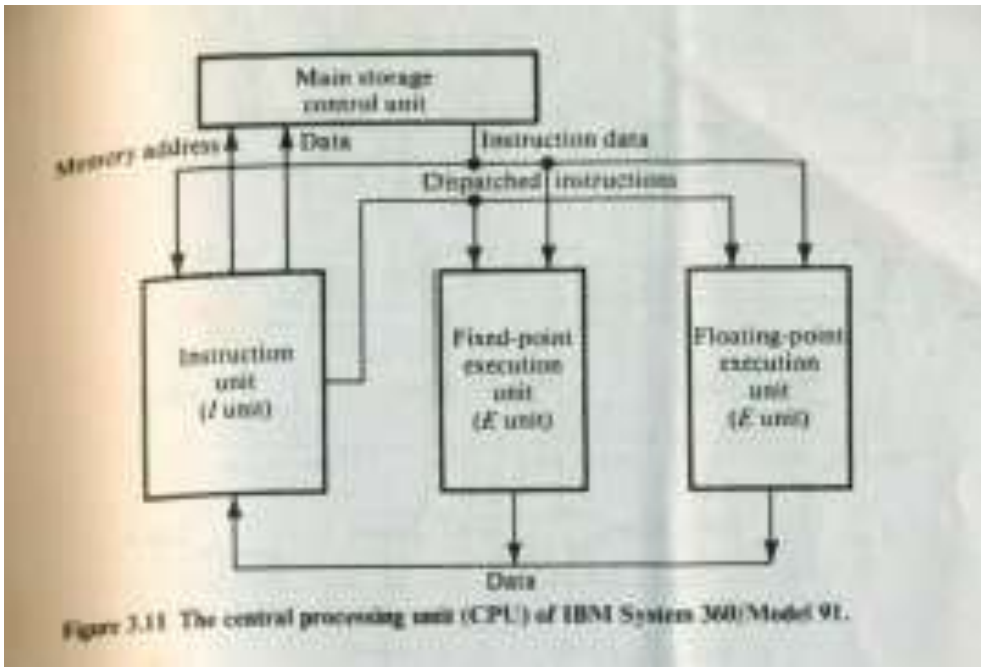


Figure 3.11 The central processing unit (CPU) of IBM System 360/Model 91.

Figure 3.1 1 The central processing unit (CPU) of IBM System 360/Model 91.

A block diagram of the CPU in the IBM 360/91 is depicted in Figure . It consists of four major parts: the *main storage control unit,* the *instruction unit,* the *fixed-point execution unit,* and the *floating-point execution unit.* The instruction unit (I unit) is pipelined with a clock period of 60 ns. This CPU is designed to issueinstructions at a burst rate of one instruction per clock cycle, and the performanceof the two execution units (E units) should support this rate. The storage control unit supervises information exchange between the CPU and the mainmemory major functions of the 1 unit, including instruction fetch, decode, and deliveryto the appropriate E unit, operand address calculation and operand fetch.

The two E units are responsible for the fixed-point and floating-point arithmetic logicoperations needed in the execution phase. '

From memory access to instruction decode and execution, the CPU is fully pipelined across the four units shown in Figure . Concurrency among successive instructions in the Model 91 is illustrated in Figure It is desirable to overlay separate instruction functions to the greatest possible degree. The shaded boxes correspond to circuit functions and the thin lines between them refer to delays scaused by memory access. Obviously, memory accesses for fetching Instructions or operands take much longer time than the delays of functional circuitary..Following the delay caused by the initial filling of the pipeline, the execuon; results will begin emerging at the rate of one per 60 ns.

For the processing of a typical floating-point storage-to-register instruction, the functional segmentation of the pipeline in Figure along with the clock time divisions. The basic time cycle accommodates the pipelining of most hardware functions. However, the memory and many execution functions require flanc:ttale number of pipeline cycles. In general, these storage and execution decOd~nsrequire a large portion of time cycles, as revealed in Figure . After lng, two parallel sequences of operation may be initiated: one for operand

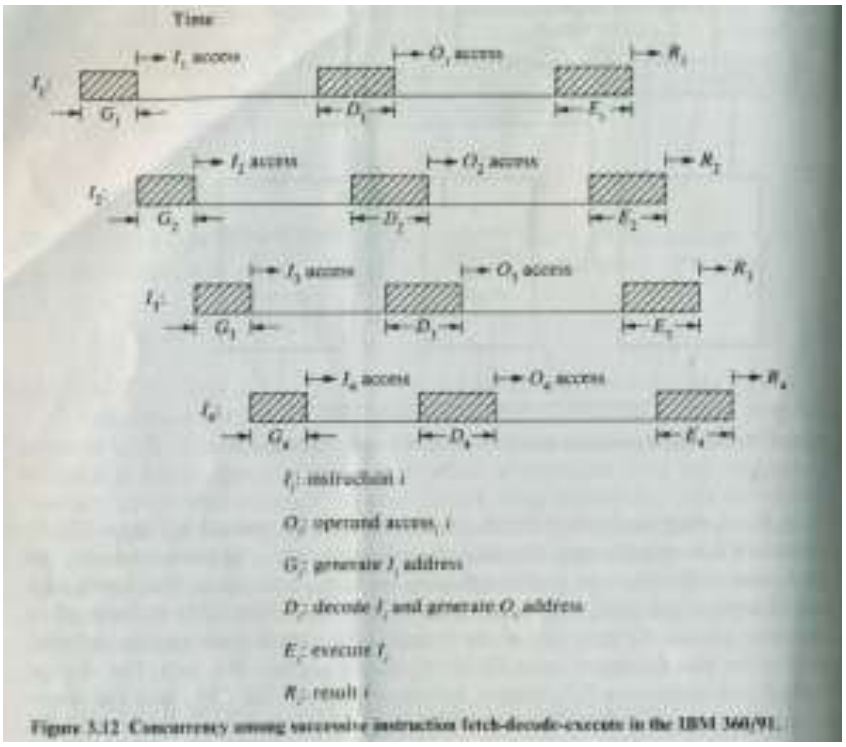
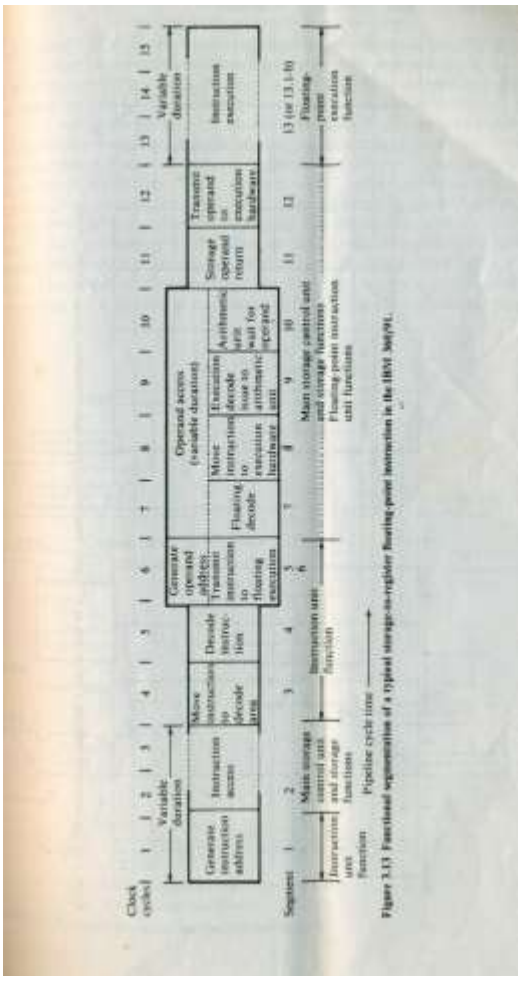


Figure 3.12 Concurrency among successive instruction fetch-decode-execute in the IBM 360/91.

access and the other for the setup of operands to be transmitted to an assi execution station in the selected arthmetic unit. The effective memory access f must match the speeds of the pipeline stages.

Because of the time disparities between various instruction types, the Model 91 utilizes the organizational techniques of memory interleaving, parallel arithmetic functions, data buffering, and internal forwarding to overcome the speed gap problems. The depth of interleaving is a function of the memory cycle time the CPU storage request rate, and the desired effective-access time. The Model 91 chooses a depth of 16 for interleaving 400 nsjcycle storage modules to sat" an effective access time of 60 ns. We will examine pipeline arithmetic and data buffering techniques in subsequent sections.

Concurrent arithmetic executions are facilitated in the Model 91 by using two separate units for fixed-point execution and floating-point execution. This permits Instructions of the two classes to be executed in parallel. As long as no cross data dependencies exist, the execution does not necessarily flow in the sequence in which the instructions are programmed. Within the floating-point E unit an *add unit* and a *mllitiply/divide unit* which can operate in parallel. Furthmore, pipelining is practiced within arithmetic units, as will be described in .

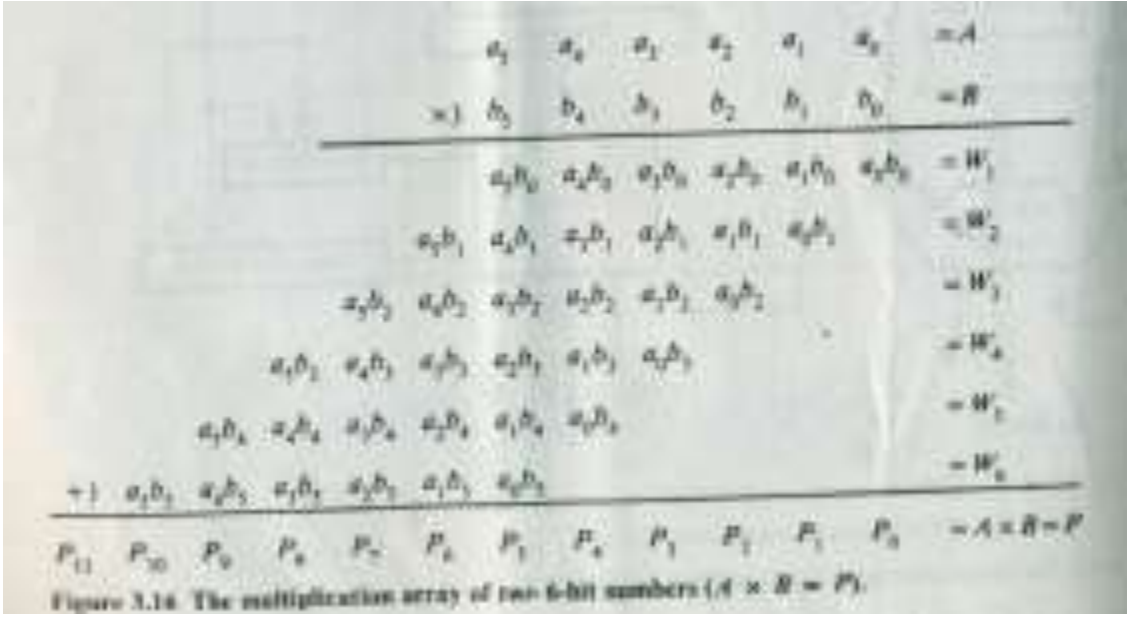


When a branch or interrupt occurs, the pipeline will lose many cycles to handle the out-of-sequence operations. Techniquesto overcome this difficulty include instruction prefetch, proper buffering, special branch handling, and optimized task scheduling. We wi1l study these techniques in and check their applications in real-life system designs

Arithmetic Pipeline Design Examples

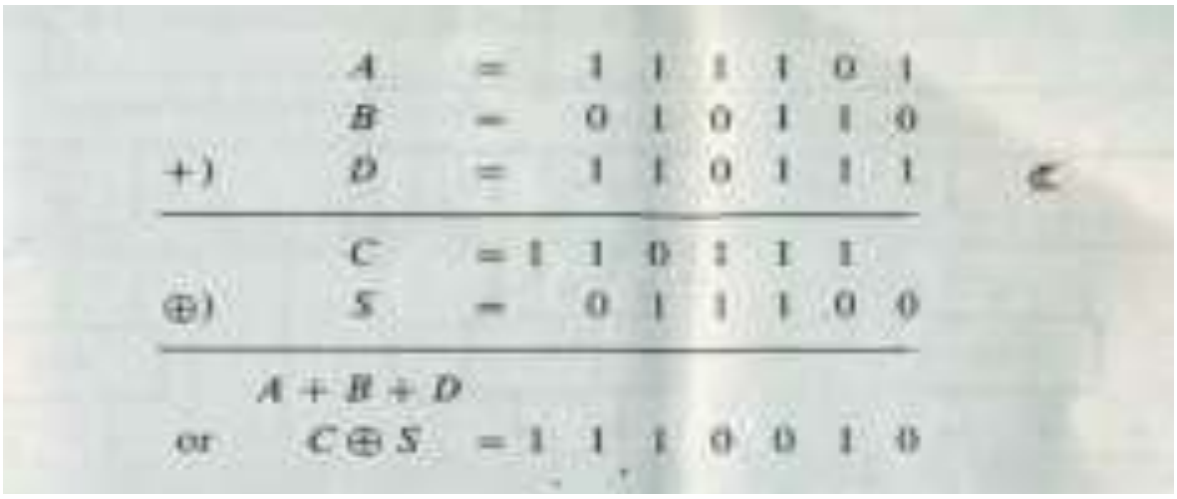
Static and unifunction arithmetic pipelines are introduced in' this section with design examples. We will study the pipeline design of Wallace trees for multiplenumber addition, which can be applied to designing pipeline multipliers and dividers. Then we will review the arithmetic pipeline designs in the IBM 360/9for high-speed floating-point addition, multiplication, and division. The method *convergence division* will be introduced, since it has been widely applied in man' commercial computers.

Traditionally, the multiplication of two fixed-point numbers is done by repeated add-shift operations, using an *arithmetic logic unit* (ALU) which has built-iadd and shift functions. The



number of add-shift operations required is proportional to the operand width. This sequential execution makes the multiplication a very slow process. By examining the multiplication array of two nurnbers Figure , it is clear that the multiplication process is equivalent to the addition of multiple copies of shifted multiplicands, such as the six shown in Figure .

Multiple-number addition can be realized with a multilevel tree adder. conventional *carry propagation adder* (CPA) adds two input numbers, say *A* and B, to produce one output number, called the sum *A* + B. A *carry-save ad*, the *sum vector* S and the *carry vector* C. Mathematically, we have *A* + *B* + D =S $ C, where + is arithmetic addition and $ is bitwise exclusive-or operation

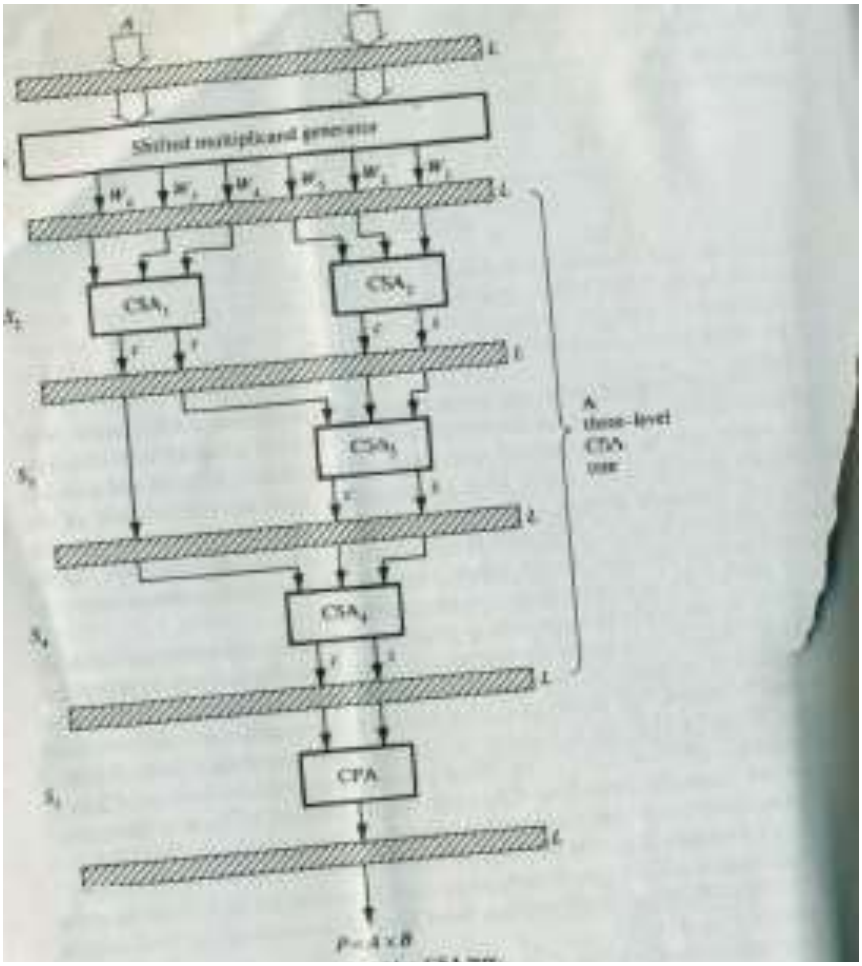


A carry-propagate adder can be implemented with a cascade offull adders with the carry-out of a lower stage connected to the carry-in of a higher stage. A carrysave adder can be implemented with a set of full adders with all the carry-interminals serving as the input lines for the third input number *D*, and the carry-out terminals serving as the output lines for the carry vector C. In other words, the carry lines of all full adders are not interconnected in a carry-sa ve adder.

For the present purpose, we can simply view a CPA as a two-to-one number converter and a CSA as a *three-to-two* number converter. .

Now we are ready to show how to use a number of CSAs for multiple-number addition. This, in turn, serves the purpose of pipeline multiplication. This pipeline is designed to multiply two 6-bit numbers, as illustrated in Figure . There are fivepipeline stages. The first stage is for the generation of all 6 x 6 = 36 immediate product terms *{ai hi* Ia :s:; i :s:; 5 and a :s:; *j* :s:; 5}, which form the six rows of shifted multiplicands *{Wi* Ii = 1, 2, ... , 6}. The six numbers are then fed into two CSAs in the second stage. In total, four CSAs are interconnected to form a three-level *carry-save adder tree* (from stage two to stage four in the pipeline). This CSA tree merges six numbers into two numbers: the sum vector S and the carry vector C.

The final stage is a CPA (carry lookahead may be embedded in it, if the operand ~ngth is long) which adds the two numbers Sand C to produce the final output, product *P* = *A* x B..



Key design problems of pipeline processors are studied in this section. We begin with a review of various instruction-prefetch and branch-control strategies for designing pipelined instruction units. Data-buffering and busing structures are presented for smoothing pipelined operations to avoid congestion. We will study Internaldata-forwarding and register-tagging techniques by examining instructiondependence relations. The detection and resolution of logic hazards in pipelines Willbe described. Principles of job sequencing in a pipeline will be studied with re.servationtables to avoid collisions in utilizing pipeline resources. Finally, we :1I consider the problems of designing dynamic pipelines and the necessary ystem supports for pipeline reconfigurations.

Instruction Prefetch and Branch Handling

From 'the viewpoi.nt 0f over Iappe d iinstructio.n executio.n sequencin.g for pipelined Processessing , the instruction mixes in typical computer programs can be classified 4 types, as shown in Table . The arithmetic load operations constitute 60 pe rcent of a typi.cal computer program. These are mainly data manipulation operations which require one or two operand fetches. The execution of different arithmetic operations requires a different number of pipeline cycles. The store-type operation does not require a fetch operand, but memory access is needed to store the data. The branch-type operation corresponds to an unconditional jump There are two possible paths for a conditional branch operation. The *yes* path requires the calculation of the new address being branched to, whereas the *no* proceeds to the next sequential instruction in the program. The arithmetic-load and store instructions do not alter the sequential execution order of the program

The branch instructions (25 percent in typical programs) may alter the programcounter (PC) in order to jump to a program location other than the next instruction. Different types ofinstructions require different cycle allocations. The branch types of instructions will cause some damaging effects on the pipeline performa

Some functions, like interrupt and branch, produce damaging effects on performance of pipeline computers. When instruction *1* is being executed the occurrence of an interrupt postpones the execution of instruction until the-interrupting request has been serviced. Generally, there are two types of interrupts. *Precise interrupts* are caused by illegal operation codes found in instructions,

which can be detected during the decoding stage. The other *imprecise interrupts,* is caused by defaults from storage, address, and execution functions.

Since decoding is usually the first stage of an instruction pipeline, an interrupt on instruction *I* prohibits instruction *1* + 1 from entering the pipeline. those instructions preceding instruction *1* that have not yet emerged from theline continue to run until the pipeline is drained. Then the interrupt routine serviced. An imprecise interrupt occurs usually when the instruction is halfway through the pipeline and subsequent instructions are already admitted in pipeline. When an interrupt of this kind occurs, no new instructions are allotted in to enter the pipeline, but all the incompleted instructions inside the pipeline, Whether they precede or follow the interrupted instruction, will be completed before the th processing unit is switched to service the interrupt.

In the Star-I 00 system, the pipelines are dedicated to vector-oriented arithtmetic operations. In order to handle interrupts during the execution of a vector ins truction, special interrupt buffer areas are needed to hold addresses, delimiters, field lengths etc., that are needed to restart the vector instructions after an Interrupt. For the Cray-I computer, the interrupt system is built around an *exchange* pac*kage.* To change tasks, it is necessary to save the current processor state and load a new processor state. The CraY-1 does this semiautomatic ally when an interrupt occurs or when a program encounters an *exit* instruction. Under such Circumstances, the Cray- I saves the eight scalar registers, the eight address registers, the program counter, and the monitor flags. These are packed into 16 words and Swapped with a block whose address is specified by a hardware exchange address register. However, the exchange package does not contain all the hardware state information, so software *interrupt handlers* must save the rest of the states. "The rest" includes 5 I2 words of vector registers, I28 words of intermediate registers, a vector mask, and a real-time clock.

The effect of branching on pipeline performance is described below by a linear instruction pipeline consisting of five segments: *instruction fetch, decode, operand fetch, execute,* and *store results.* Possible memory conflicts between overlapped fetches are ignored and a sufficiently large cache memory (instructiondata buffers) is used in the following analysis.)

As illustrated in Figure, the instruction pipeline executes a stream of instructions continuously in an overlapped fashion if branch-type instructions do not appear. Under such circumstances,(Once the pipeline is filled up with seq uential instructions (nonbranch type), the

pipeline completes the execution of one Instruction per a fixed latency (usually one or two clock periods

On the other hand, a branch instruction entering the pipeline may be halfway downthe pipe (such as a "successful" conditional branch instruction) before a br anch decision is made. This will cause the program counter to be loaded with new address to which the program should be directed, making all prefetched Instructions (either in the cache memory or already in the pipeline) useless. The next Ins.truction cannot be initiated until the completion of the current branch instruction cycle

.This causes extra time delays in order to drain the pipeline, as

In. Figure The overlapped action is suspended and the pipeline must be drained at the end of the branch cycle. The continuous flow of instructions in the pipeline temporarily interrupted because of the presence of a branch instruction In general, the higher the percentage of branch-type instructions in a program,the slower a program will run on a pipeline processor

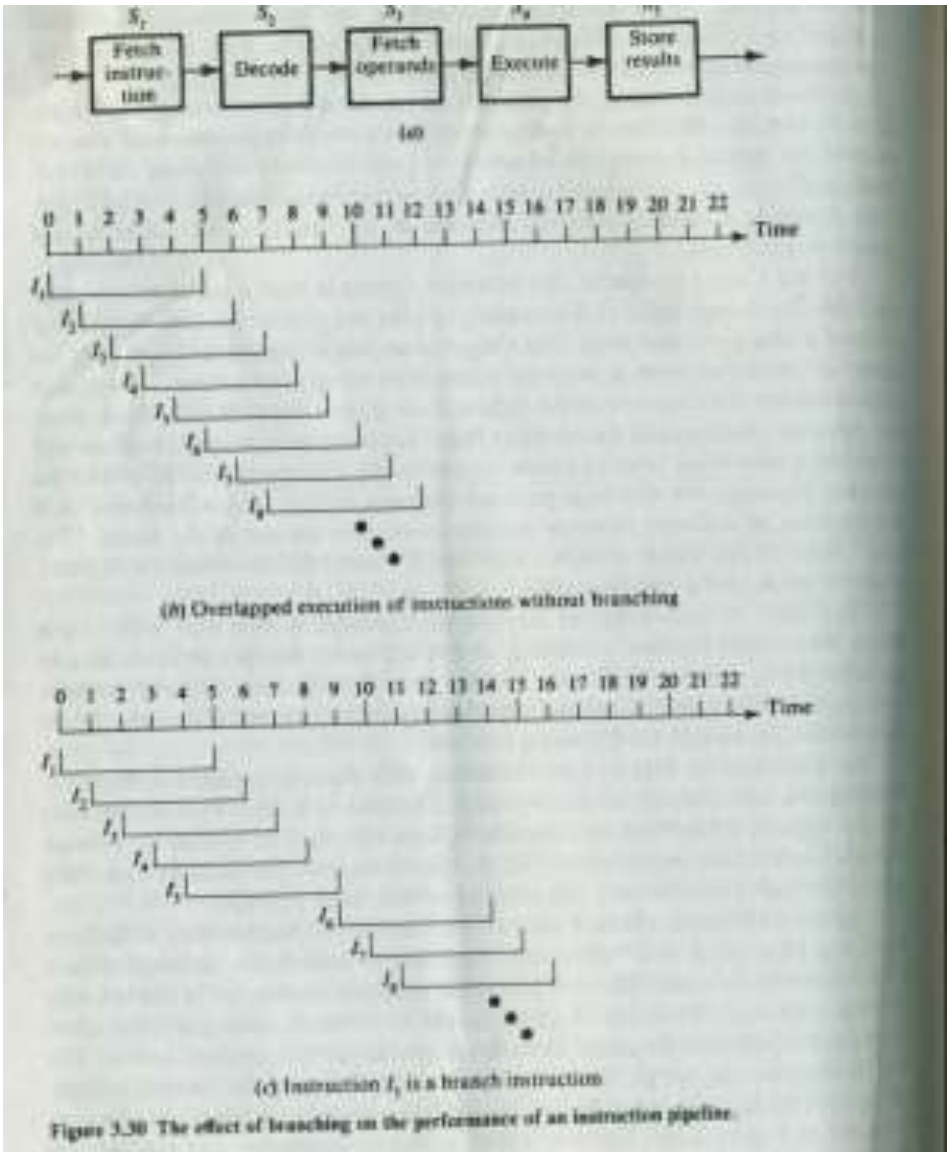
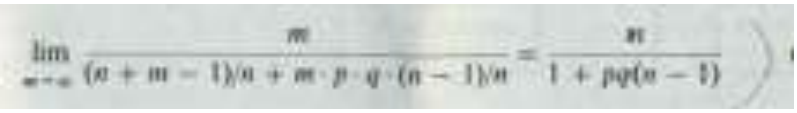


Figure 3.30 The effect of branching on the performance of an instruction pipeline

if a branch instruction does not occur the performance would be one instruction per each pipeline cycle. Let *p* be probability of a conditional branch instruction in a typical program and *q* be the probability that a branch is successful Suppose that there are *m* instructions waiting to through the pipeline. The number of instructions that cause successful branches equals *m . p . q.* Since *(n* - I*)/n* extra time delay is needed for each successful branch instruction, the total instruction cycles required to process these *m* instructions equal (l/n)(11 + m - I) + *(m· p- q)(n - I)/n.* As *m* becomes very large lthe performance of the instruction pipeline is measured by the average number ofmstructions executed per instruction cycle:



When $p = 0$ (no branch instructions encountered), the above measure reduces to n instructions per n pipeline clocks, which is ideal. In reality, the above ratio is always less than n. For example, with $n = 5$, $p = 20$ percent, and $q = 60$ percent, we have the performance of instructions per instruction cycle (or 5 pipeline cycles),which is less than the ideal execution rate of 5 instructions per 5 pipeline cycles. In other words, an average of 35.2 percent cycles may be wasted because of branching. In order to cope with the damaging effects of branch instructions, various mechanisms have been developed in pipeline computers . I unit in the IBM 360/91 was described (Figure). Formally, a prefetching strategy can be stated as follows:

The prefetch of instructions is modeled in Figure . The memory is assumed to be interleaved and can accept requests at one per cycle. All requests require T cycles to return from memory. There are two prefetch buffers of sizes sand t instruction words. The s-size buffer holds instructions fetched during the sequential part of a run. When a *branch* is successful, the entire buffer is invalidated. The other buffer holds instructions fetched from the target of a conditional branch. When a conditional branch is resolved and determined to be unsuccessful, the contents of this buffer are invalidated. The decoder requests instruction words at a maximum rate of one per r cycles. If the instruction requested by the decoder is available in the sequential buffer for sequential instructions, or is in the target buffer if a conditional branch has iust been resolved. and is successful, it enters the decoder with zero delay.

Otherwise, the decoder is Idle until the instruction returns from memory . for *jump* instructions, all decoded instructions enter the execution, Where E units are required to complete execution. If the decoded instruction is an unconditional branch, the instruction word at the target of the *jump*

is requested immediately by the decoder and decoding ceases until the target instruction returns from the memory. The pipeline will see the full memory latency time T,since there was no opportunity for target prefetching.

If the decoded Instruction is a *conditional branch,* sequential prefetching is enters the during the E cycles it is being executed. The instruction simultaneously enters solved execution npipeline, but no more instructions are decoded until the branch is resolved at the end of E units. Instructions are prefetched from the target memory address of the conditional branch instruction. Requests for *t* target instructions are issued at the rate of one per cycle. Once the branch is resolved, target prefetching becomes unnecessary.

If the branch is successful, the target instruction stream becomes the sequentialstream, and instructions are requested every *r* time units from this stream.Execution of this new stream begins when the target of the branch returns memory, or whenever E units have elapsed, whichever is later. If the branch is unsuccessful, instruction requests are initiated every r units of time following branch resolution and continue until the next branch or jump is decoded. Instruction prefetching reduces the damaging effects of branching

Data Buffering and Busing Structures

The processing speeds of pipeline segments are usually unequal. Consider the example pipeline in Figure with three segments having delays T1,T2 and T3.

If *T, = T3* = Tand *T2 = 3T,* obviously segment S2 is the bottle-neck. The throughput of the pipeline is inversely proportional to the bottleneck.

Therefore, it is desirable to remove the bottleneck which causes the unneces, sary congestion. One obvious method is to subdivide the bottleneck. Figure shows two different subdivisions of segment S2' The throughput is increased

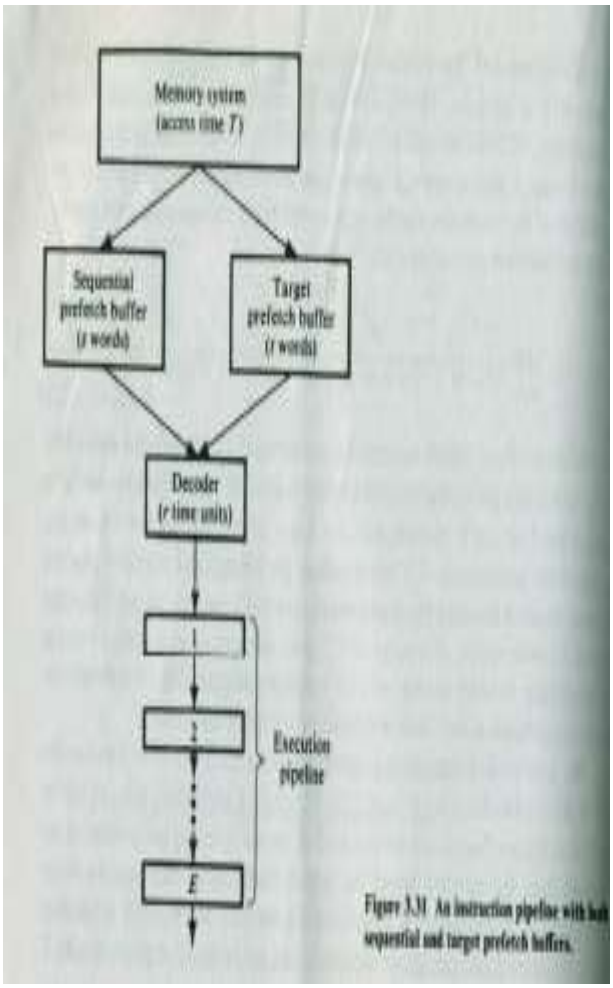
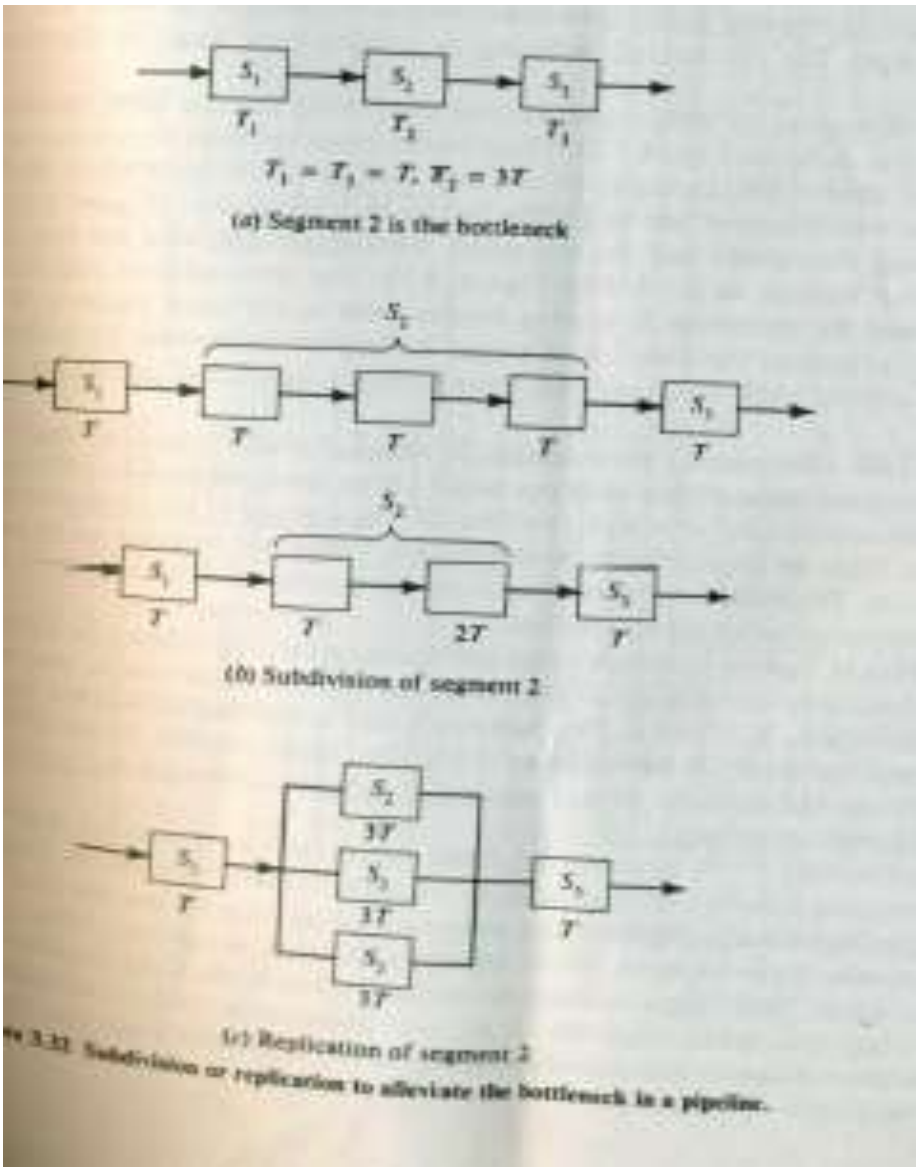


Figure 3.31 An instruction pipeline with both sequential and target prefetch buffers.



the bottleneck in parallel is another way to smooth congestions, as depicted in Figure . The control and synchronization of tasks in parallel segments are much more complex than those for cascaded segments.

Data and instruction buffers Another method to smooth the traffic flow in a pipeline

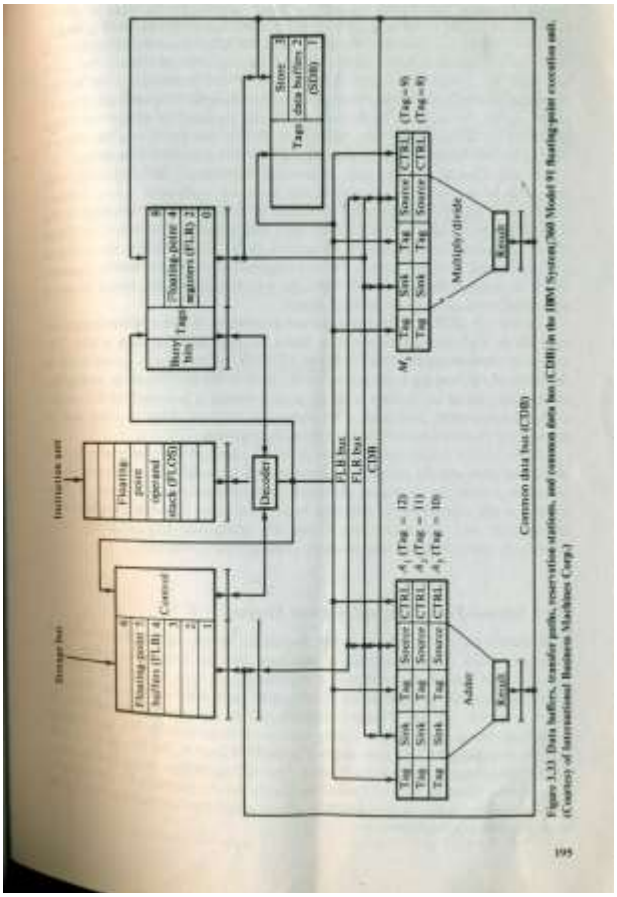
is to use buffers to close up the speed gap between the memory accesses for either instructions or operands and the arithmetic logic executions in the functional pipes. The instruction or operand buffers provide a continuous supply of instructions operands to the appropriate pipeline units. Buffering can avoid unnecessary idling of the processing stages caused by memory-access conflicts or by unexpect branching or interrupts. Sometimes the entire loop's instructions can be stored in the buffer to avoid repeated fetch of the same instruction loop. if the buffer size ' sufficiently large. The amount of buffering is usually very large in pipelipeline computers.

.. Instructions are first fetched to the instruction-fetch buffers (64 bits each) before sending them to the instruction unit (Figure)

After decoding, fixed-point and floating-point instructions and data are sent their dedicated buffers, as labeled in Figure . The store-address and data buffers are used for continuously storing results back to the main memory, have already explained the function of target buffers for instruction prefetching The storage-conflict buffers are used only when memory-access conflicts are taking place.

In the STAR-l 00 system, a 64-word (of 128 bits each) buffer is used to ternperorarily a hold the input data stream until operands are properly aligned. In additonthere is an instruction buffer which provides for the storage of thirty-twOinstructions. Eight 64-bit words in the instruction buffer will be filled up by memory fetch. The buffer supplies a continuous stream of instructions executed, despite memory-access conflicts.

In the TI-ASC system, two eight-word buffers are utilized to balance the stream of instructions from the memory to the execution unit. A *memory buffer unit* three double buffers, X, Y, and Z. Two buffers (X and Y) are used to hold the'operands and the third (Z buffer) is used for the output results. These greatly alleviate the problem of mismatched bandwiclths between the memory and arithmetic operations.



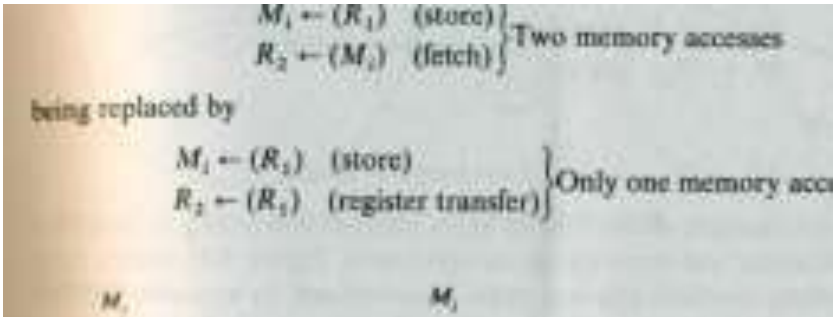
I

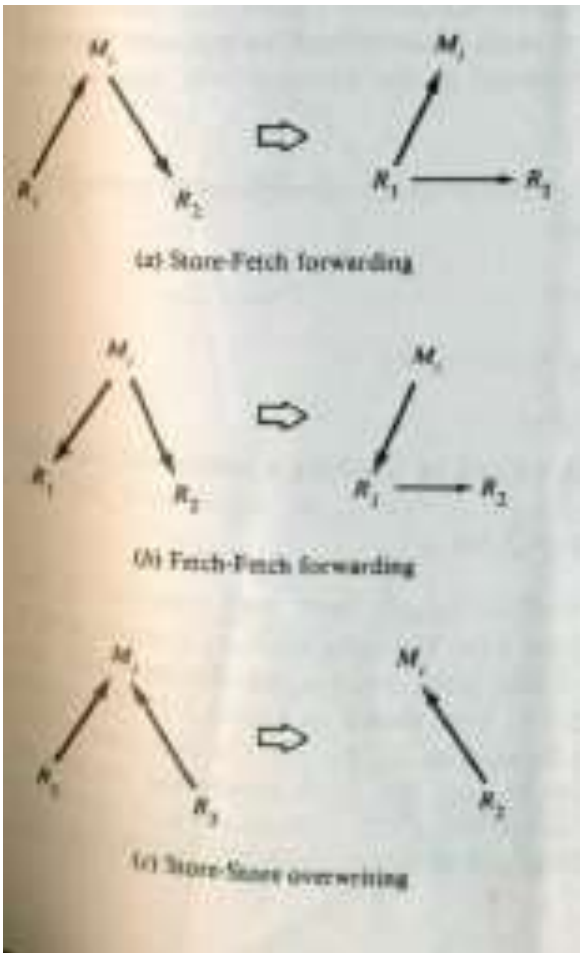
Internal forwarding and register tagging

Internal forwarding refers"short-circuit" technique for replacing unnecessary memory accesses by re .to-register transfers in a sequence of fetch-arithmetic-store operations. *Register taggin g* refers to the use of tagged registers, buffers, and reservation stations are exploiting concurrent activities among multiple arithmetic units. We will show how these techniques have been applied in the IBM System/360 Model 91, has multiple execution units with common data buffers and data paths. The computer performance can be greatly enhanced if one can elirninate unnecessary memory accesses and combine some transitive or multiplefetch-store operations with faster register operations. This concept of internal dataforwarding can be explored in three directions, as illustrated in Figure.

The symbols *M,* and *R,* to represent the ith word in the memory and the jth registerin the CPU. We use arrows to specify data-moving operations such as fetch,store, and register-to-register transfer. The contents of *M,* and *R,* are represented by (Mi) and *(Rj),* respectively.

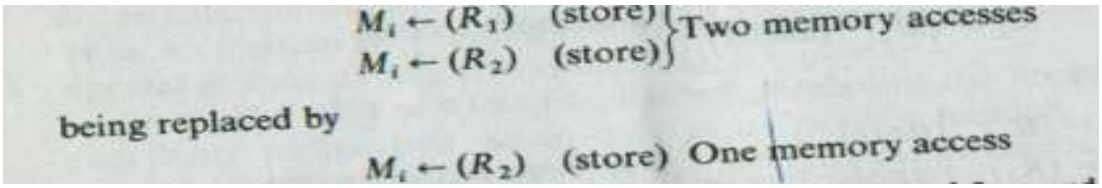
Store-fetch forwarding The following sequence of the two operations store-thenfetchcan be replaced by two parallel operations, one store and one register transfer, as shown in Figure





Fetch-fetch forwardiDgThe following two fetch operations can be replaced by one fetch and one registertransfer, as shown in Figure *3.34b.* Again one memory access has been eliminated:

Store-store overwriting The following two memory updates (stores) of the s wordcan be combined into one, since the second store overwri the first:



sequence of arithmetic and memory-access operations. Figure 3.35 depicts t simplification steps, in which adjacent steps are combined to minimize mem references. Nodes in the graph correspond to the memory cells, registers. adder, or a multiplier

. Hazard Detection and Resolution

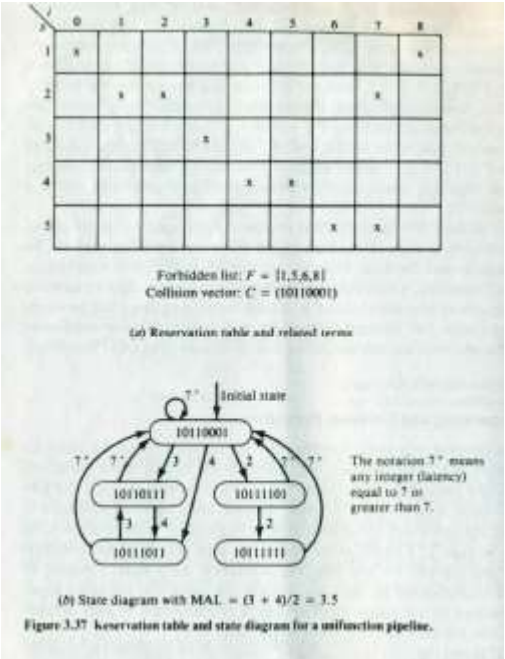


Figure 3.37 Reservation table and state diagram for a unifunction pipeline.

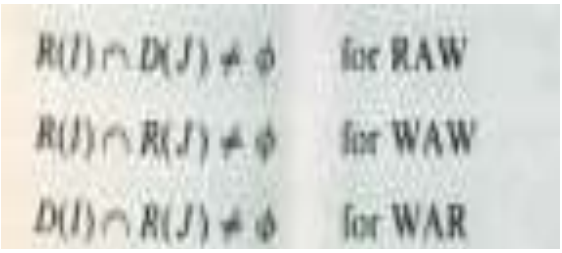
Pipeline hazards are caused by resource-usage conflicts among various instruct tions in the pipeline. Such hazards are triggered by interinstruction dependencies. In this section, we characterize various hazard conditions. Hazard-detection are known as *data-dependent hazards.* Methods to cope with such hazards are needed in any type of look ahead processors for either synchronous or asynchronous-multiprocessing systems. Another type of hazard Is dueto a job scheduling problem .

When successive instructions overlap their fetch, decode and execution through a pipeline processor, interstruction dependencies may .arise to prevent the sequential data flow in the

pipeline. For example, an instruction may depend on the results of a previous instruction. Until the completion of the previous instruction, the present instruction cannot be initiated into the pipeline. In other instances, two stages of a pipeline may need to update the same memory location.

Hazards of this sort, if not properly detected and resolved, could result in an *interlock* situation in the pipeline or produce unreliable results by overwriting.There are three classes of data-dependent hazards, according to various data update patterns: *write after read* (WAR) hazards, *read after write* (RAW) hazards, and *write after write* (WAW) hazards. Note that *read-after-read* does not pose a problem, because nothing is changed.

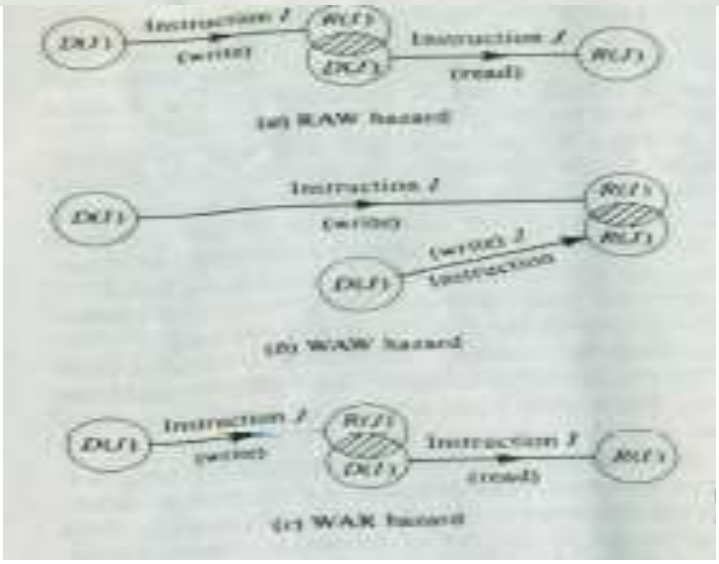
We use *resource objects* to refer to working registers, memory locations, and special flags. The contents of these resource objects are called *data objects.* Each instruction can be considered a mapping from a set of data objects to a set of data objects. The *domain D(I)* of an instruction 1is the set of resource objects whose data objects may affect the execution of instruction 1. The *range R(I)* of an instruction *I* isthe set of resource objects whose data objects may be modified by the execution of instruction 1. Obviously, the operands to be used in an instruction execution are retrieved (read) from its domain, and the results will be stored (written) in its range. In what follows, we consider the execution of the two instructions 1and *J* in a program. Instruction *J* appears after instruction *I* in the program. There may be none or other instructions between instructions *I* and *J.* The latency between the two instructions is a very subtle matter. Instruction *J* may enter the execution before or after the completion of the execution of instruction 1. The improper timing and data dependencies may create some hazardous situations, as shown in figure.RAW hazard between the two instructions 1 and *J* may occur when *J* may attempts to read some data object that has been modified by 1. A WAR hazard may occur When *J.* attempts to modify some data object that is read by *I.* A WAW hazard may occur if both *I* and *J* attempt to modify the same data object. Formally,necessary conditions for these hazards are given below



Possibly hazards for the four types of instructions are listed in the table.

Table 3.3 Possible hazards for various instruction types

Instruction J (second)	Instruction I (first): Arithmetic and load type	Store type	Branch type	Conditional branch type
Arithmetic and load type	RAW WAW WAR	RAW WAR	WAR	WAR
Store type	RAW WAR	WAW		
Branch type	RAW		WAW	WAW
Conditional branch type	RAW		WAW	WAW



Hazard detection in the Instruction-fetch stage of a pipeline processor by comparing the domain and range of the incoming instruction with those of the instructions being processed in the pipe. Another approach is to allow the incomIng instruction through the pipe and distribute the detection toal1 the potcntial pipeline stages. This distributed approach offers better flexibility to all the potential pipeline stages

Once a hazard is detected, the system should resolve the interlock situation consider the instruction sequence {.. , *I*, *I* + 1,.,., *J*, *J* + I, ... } in which a hazardhas been detected between the current instruction *J* and a previous instruction I .A straightforward approach is to stop the pipe and to suspend the execution of 'instructions *J*, *J* + 1, *J* + 2, ... , until the instruction *I* has passed the point of resource conflict. A more sophisticated approach is to suspend only instruction J and continue the flow of instructions *J* + I, *J* + 2, ... , down the pipe. Of course, the potential hazards due to the suspension of *J* should be continuously checked as instructions *J* + I, *J* + 2, ' ..

move ahead of *J.* Multilevel hazard detection may be encountered, requiring much more complex control mechanisms to resolve a stack of hazards.

In order to avoid RAW hazards, IBM engineers developed a *short-circuiting* approach which gives a copy of the data object to be written directly to the instruction waiting to read the data. This concept was generalized into a technique, known as *data forwarding,* which forwards multiple copies of the data to as many waiting instructions as may wish to read it. A data-forwarding chain can be established in some cases. The internal-forwarding and register-tagging techniques presented in the section should be helpful in resolving logic hazards in pipelines.

Dynamic Pipelines and Reconfigurability

A dynamic pipeline may initiate tasks from different reservation tables simultan eously to allow multiple numbers of initiations of different functions in the pipeline. Two methods for improving the throughput of dynamic pipeline processors have been proposed by Davidson and Patel (1978). The reservation pipeline can be modified with the insertion of noncompute delays or with the internal buffers at each stage. The utilization of the stages and, hence, the through put of the pipe can be greatly advanced with a modified reservation table yielding a more desirable latency pattern.

It is assumed that any co utation step can be delayed by inserting a non compute stage. We consider first a unifunction pipeline. A *constant latency cycle* is a cycle with only one latency. A latency between two tasks is said to be allowablif these two tasks do not collide in the pipeline. Consequently, a cycle is allowable in a pipeline if all the latencies in the cycle are allowable.

. However, an allowable cycle with the MAL d.oes not necessarily imply100percentutilizationof the pipe e1inewhere utilization is measured as the percentage of time the busiest stage piP ains busy. When a latency cycle results in a 100 percent utilization of at least one of the pipeline stages, the periodic latency sequence is called a *perfect cycle.*

, pipelines es with perfect .cyc1es can be better utilized than those with nonperfect initiation cycles.

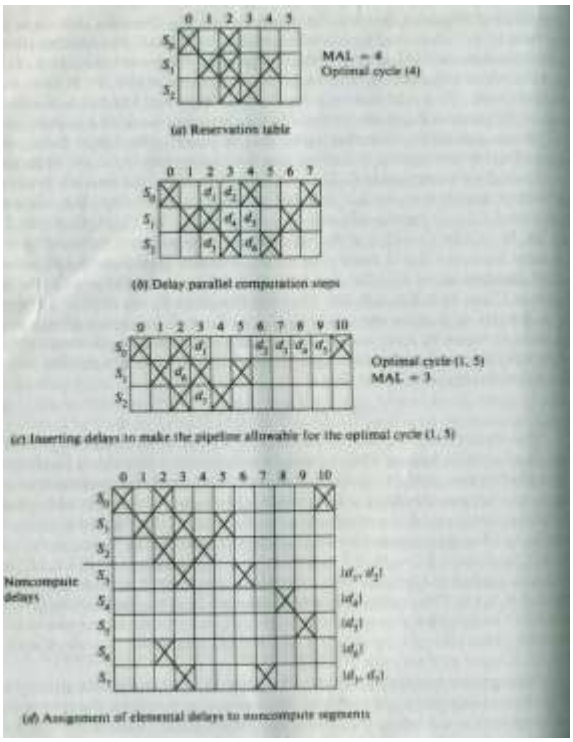
Consider a latency cycle C. The set *Gc* of all possible time Intervals between initiations derived from cycle C is called an *initiation interval set.* For example,

Gc = {4, 8,I2, ... } for C = (4), and *c;* = {2, 3, 5, 7, 9, 10, 12, 14, 15, 17, 19,21,

22 24,26, ... } for C = (2, 3, 2, 5). Note that the interval is not restricted to two adjacent initiations. Let Gc(mod *p)* be the set formed by taking mod *p* equivalents of all clements of set *Gc*· For the cycle (2,3,2,5) with period *p* = 12, the set G (mod 12) = {O,2, 3, 5, 7,9, IO}. *{* A latency cycle C with a period *p* and an initiation interval set Gc is allowable in a pipeline with a forbidden latency set *F* if, and only if,

This means that there will be no collision if none of the initiation intervals equals a forbidden latency . Thus, a constant cycle *(I)* with a period *p = I* is allowed for a pipeline processor I , and only if, *I* does not divide any forbidden latency in the set *F.* Another way of looking at the problem is to choose a reservation table whose fo.rbiddenlatency set *F* is a subset of the set *GcCmod p).* Then the latency cycle C={2,3,4,5} and *Gc(mod* 12) = {I, 4, 6, 8, II}, so C can be apphed to a pipeline with a forbidden latency set *F* equal to any subset Of{ l, 4, 6, 8, 11}. This condition is very effective to check the applicability (allowth I Ity) of an initiation sequence (or a cycle) to a given pipeline, or one can modify th: reser vationtable of a pipeline to yield a forbidden list which is confined within set Gc{mod *p),* if the cycle C is fixed..

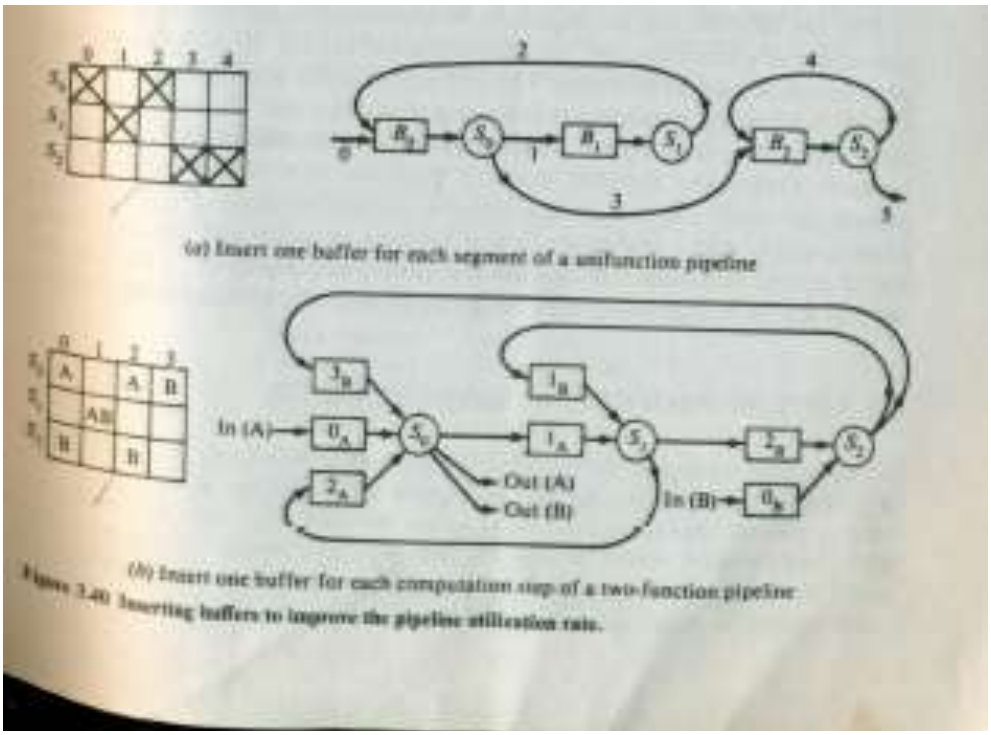
Adding noncompute stages to a pipeline can make it allowable for a given cycle. The effect of delaying *d* before the step being delayed. Each *d* indicates one unit of Delay called an *elemental delay.* It is assumed that all steps in a column must complete any steps in the next column are executed. In Figure , the effect of delaying the step in row 0 and column 2 by two time units and the step in row 2 and column 2 The elemental 'ac OUtPUt ' 3 reqUIre the use of the additional delays *d4, d5* and *d 6* to make all the outputs sImultaneously available in column 2 of the original reservation table.



row of the table. Thus by adding elemental delays, a unifunction pipeline can at be fully utilized through the use of a cycle that has a constant latency equal to the maximum number of marks in any row of the reservation table. The maximum achievable throughput of that pipeline is thereby attained. On the other hand, an arbitrary cycle, a pipeline can be made allowable by delaying some steps. reservation table of Figure *a* can be made allowable with respect to optimal cycle (1, 5) by adding some elemental delays. The resulting table is shown in figure.. Once a modified table is obtained, it is necessary to assign the mental delays to noncompute stages. Noncompute stages may be shared by various elemental delays. Figure shows the modified reservation table after introduction. Of noncom pute stages *S3, S4, S5,S6 ,S7*

The task arrivals In a pipeline processor may be periodic for a program with per loops. If we assume that each task can only occupy one stage at a time, no pa:rallel computations can be done within a single task. Such an assumption stems *on* the practical difficulties encountered in implementing a priority scheme in volving parallel computations of a task. Once some buffers are provided internally the task-scheduling problem can be greatly simplified. Whenever two or more tasks are trying to use the same stage, only one of the tasks is allowed to use the stage, while the rest wait in the buffers according to some priority schemes.
There are two different implementations of internal buffers in a pipeline: The

first uses one buffer for each stage (Figure *3.40a),* and the second uses one buffer per computation step (Figure *3.40b).* For one buffer per stage, two priority schemes, *Fl FO-global* and *LIFO-global,* can be used. In the FIFO-global scheme, a task has priority over all tasks initiated later. In the LIFO-global scheme, a task has priority over all tasks initiated earlier. For one buffer per computation step, multiple buffers may be used in each segment with the following priorities: MPF: most processed first; LPF: least processed first; LWRF: least work remaining first; and MWRF: most work remaining first.



(a) Insert one buffer for each segment of a unifunction pipeline

(b) Insert one buffer for each computation step of a two-function pipeline

Figure 3.40 Inserting buffers to improve the pipeline utilization rate.

. To achieve this, a more complicated structure of pipeline segments and their interconnection controls is needed. Bypass techniques can be used to avoid unwanted attempts to usetheoperands fetched for preceding instructions. To alleviate this problem, one solution has each instructionactivate a number of consecutive stages down the pipeline which satisfy its need. A dynamic pipeline would allow several configurations to be simultaneously present. For example, a dynamic-pipeline arithmetic unit could perform addition and multiplication at the same time. Tremendous control overhead and increasing interconnection complexity would be expected. None of the existing pipeline processor s has achieved this dynamic capability. Most

commercial pipelines static. In TI-ASC, the desired control allows different instructions to assume diferent data paths through the arithmetic pipeline at different times. All path-cont: information is stored in a *read-ollly memory* (ROM), which can be accessed at the initiation of an instruction.