RAJIV GANDHI PROUDYOGIKI VISHWAVIDYALAYA, BHOPAL

New Scheme Based On AICTE Flexible Curricula Computer Science & Engineering, VI-Semester

CS-603(A): Advanced Computer Architecture

UNIT III

Topic Covered

Linear pipeline processor, Non-Linear Pipeline Processors, Reservation Table, Instruction pipeline, Superpipeline and Superscalar technique

Linear pipeline processors

Linear pipelining Pipelining is a technique of that decomposes any sequential process into small subprocesses, which are independent of each other so that each subprocess can be executed in a special dedicated segment and all these segments operates concurrently. Thus whole task is partitioned to independent tasks and these subtask are executed by a segment. The result obtained as an output of a segment (after performing all computation in it) is transferred to next segment in pipeline and the final result is obtained after the data have been through all segments. Thus it could understand if take each segment consists of an input register followed by a combinational circuit. This combinational circuit performs the required sub operation and register holds the intermediate result. The output of one combinational circuit is given as input to the next segment. The concept of pipelining in computer organization is analogous to an industrial assembly line. As in industry there different division like manufacturing, packing and delivery division, a product is manufactured by manufacturing division, while it is packed by packing division a new product is manufactured by manufacturing unit. While this product is delivered by delivery unit a third product is manufactured by manufacturing unit and second product has been packed. Thus pipeline results in speeding the overall process. Pipelining can be effectively implemented for systems having following characteristics:

• A system is repeatedly executes a basic function.

• A basic function must be divisible into independent stages such that each stage have minimal overlap.

• The complexity of the stages should be roughly similar.

The pipelining in computer organization is basically flow of information. To understand how it works for the computer system lets consider an process which involves four steps / segment and the process is to be repeated six times. If single steps take t nsec time then time required to complete one process is 4 t nsec and to repeat it 6 times we require 24t nsec. Now let's see how problem works behaves with pipelining concept. This can be illustrated with a space time diagram given below figure, which shows the segment utilization as function of time. Lets us take there are 6 processes to be handled (represented in figure as P1, P2, P3, P4, P5 and P6) and each process is divided into 4 segments (S1, S2, S3, S4). For sake of simplicity we take each segment takes equal time to complete the assigned job i.e., equal to one clock cycle. The horizontal axis displays the time in clock cycles and vertical axis gives the segment number. Initially, process1 is handled by the segment 1. After the first clock segment 2 handles process 1 and segment 1 handles new process P2. Thus first process will take 4 clock cycles and remaining processes will be completed one process each clock cycle. Thus for above example total time required to complete whole job will be 9 clock cycles (with pipeline organization) instead of 24 clock cycles required for non pipeline configuration.

	1	2	3	4	5	6	7	8	9
P1	S1	S2	S3	S4					
P2		S1	S2	S3	S4				
P3			S1	S2	S3	S4			
P4				S1	S2	S3	S4		
P5					S1	S2	S3	S4	
P6						S1	S2	S3	S4

Fig: Space Timing Diagram

Speedup ratio : The speed up ratio is ratio between maximum time taken by non pipeline process over process using pipelining. Thus in general if there are n processes and each process is divided into k segments (subprocesses). The first process will take k segments to complete the processes, but once the pipeline is full that is first process is complete, it will take only one clock period to obtain an output for each process. Thus first process will take k clock cycles and remaining n-1 processes will emerge from the pipe at the one process per clock cycle thus total time taken by remaining process will be (n-1) clock cycle time.

Let tp be the one clock cycle time.

The time taken for n processes having k segments in pipeline configuration will be

= k*tp + (n-1)*tp= (k+n-1)*tp

The time taken for one process is tn thus the time taken to complete n process in non pipeline configuration will be

= n*tn

Thus speed up ratio for one process in non pipeline and pipeline configuration is

= n*tn / (n+k-1)*tp

If n is very large compared to k than

=tn / tp

If a process takes same time in both case with pipeline and non pipeline configuration than

tn = k*tp

Thus speed up ratio will

Sk =k*tp/tp =k

Theoretically maximum speedup ratio will be k where k are the total number of segments in which process is divided.

The following are various limitations due to which any pipeline system cannot operate at its maximum theoretical rate i.e., k (speed up ratio).

a. Different segments take different time to complete there suboperations, and in pipelining clock cycle must be chosen equal to time delay of the segment with maximum propagation time. Thus all other segments have to waste time waiting for next clock cycle. The possible solution for improvement here can if possible subdivide the segment into different stages i.e., increase the number of stages and if segment is not subdivisible than use multiple of resource for segment causing maximum delay so that more than one instruction can be executed in to different resources and overall performance will improve.

b. Additional time delay may be introduced because of extra circuitry or additional software requirement is needed to overcome various hazards, and store the result in the intermediate registers. Such delays are not found in non pipeline circuit.

c. Further pipelining can be of maximum benefit if whole process can be divided into suboperations which are independent to each other. But if there is some resource conflict or data dependency i.e., a instruction depends on the result of pervious instruction which is not yet available than instruction has to wait till result become available or conditional or non conditional branching i.e., the bubbles or time delay is introduced.

Efficiency :

The efficiency of linear pipeline is measured by the percentage of time when processor are busy over total time taken i.e., sum of busy time plus idle time. Thus if n is number of task , k is stage of pipeline and t is clock period then efficiency is given by

$\eta = n/ [k + n -1]$

Thus larger number of task in pipeline more will be pipeline busy hence better will be efficiency. It can be easily seen from expression as $n \rightarrow \infty$, $\eta \rightarrow 1$.

$\eta = Sk/k$

Thus efficiency η of the pipeline is the speedup divided by the number of stages, or one can say actual speed ratio over ideal speed up ratio. In steady stage where $n >> k$, η approaches 1.

Throughput:

The number of task completed by a pipeline per unit time is called throughput, this represents computing power of pipeline. We define throughput as

W= n/[k*t + (n-1) *t] = η/t

In ideal case as η -> 1 the throughout is equal to 1/t that is equal to frequency. Thus maximum throughput is obtained is there is one output per clock pulse.

Que.1.

A non-pipeline system takes 60 ns to process a task. The same task can be processed in six segment pipeline with a clock cycle of 10 ns. Determine the speedup ratio of the pipeline for 100 tasks. What is the maximum speed up that can be achieved?

Soln.

Total time taken by for non pipeline to complete 100 task is

= 100 * 60 = 6000 ns

Total time taken by pipeline configuration to complete 100 task is = (100 + 6 –1) *10 = 1050 ns

Thus speed up ratio will be = 6000 / 1050 = 4.76

The maximum speedup that can be achieved for this process is = 60 / 10 = 6

Thus, if total speed of non pipeline process is same as that of total time taken to complete a process with pipeline than maximum speed up ratio is equal to number of segments.

Que 2.

A non-pipeline system takes 50 ns to process a task. The same task can be processed in a six segment pipeline with a clock cycle of 10 ns. Determine the speedup ratio of the pipeline for 100 tasks. What is the maximum speed up that can be achieved?

Soln.

Total time taken by for non pipeline to complete 100 task is = 100 * 50 = 5000 ns

Total time taken by pipeline configuration to complete 100 task is = (100 + 6 –1) *10 = 1050 ns Thus speed up ratio will be = 5000 / 1050 = 4.76

The maximum speedup that can be achieved for this process is = 50 / 10 = 5

The two areas where pipeline organization is most commonly used are arithmetic pipeline and instruction pipeline. An arithmetic pipeline where different stages of an arithmetic operation are handled along the stages of a pipeline i.e., divides the arithmetic operation into suboperations for execution of pipeline segments. An instruction pipeline operates on a

stream of instructions by overlapping the fetch, decode, and execute phases of the instruction cycle as different stages of pipeline.

Non Linear Pipeline

The transfer of control is non linear. A dynamic pipeline allows feed forward and feedback connections in addition to streamline connection. A dynamic pipelining may initiate tasks from different reservation tables simultaneously to allow multiple numbers of initiations of different functions in the same pipeline.

Difference Between Linear and Non-Linear pipeline:

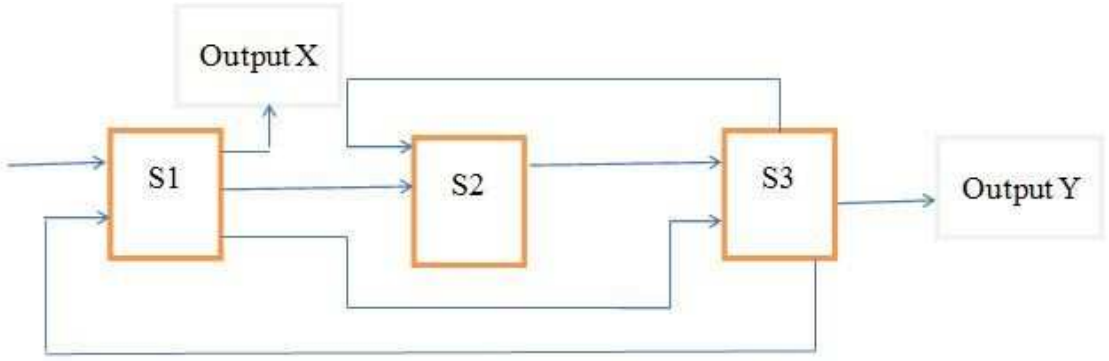
Linear Pipeline	Non-Linear Pipeline
Linear pipeline are static pipeline because they are used to perform fixed functions.	Non-Linear pipeline are dynamic pipeline because they can be reconfigured to perform variable functions at different times.
Linear pipeline allows only streamline connections.	Non-Linear pipeline allows feed-forward and feedback connections in addition to the streamline connection.
It is relatively easy to partition a given function into a sequence of linearly ordered sub functions.	Function partitioning is relatively difficult because the pipeline stages are interconnected with loops in addition to streamline connections.
The Output of the pipeline is produced from the last stage.	The Output of the pipeline is not necessarily produced from the last stage.
The reservation table is trivial in the sense that data flows in linear streamline.	The reservation table is non-trivial in the sense that there is no linear streamline for data

	flows.
Static pipelining is specified by single Reservation table.	Dynamic pipelining is specified by more than one Reservation table.
All initiations to a static pipeline use the same reservation table.	A dynamic pipeline may allow different initiations to follow a mix of reservation tables.

Reservation Table

Reservation tables are used how successive pipeline stages are utilized for a specific evaluation function. These reservation tables show the sequence in which each function utilizes each stage. The rows correspond to pipeline stages and the columns to clock time units. The total number of clock units in the table is called the evaluation time. A reservation table represents the flow of data through the pipeline for one complete evaluation of a given function.

A Three stage Pipeline



Reservation Table: Displays the time space flow of data through the pipeline for one function evaluation.

Time/Stage	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
X						**X**		**X**

S1		X		X				
S2			X		X		X	
S3								

Reservation function for a function x

Latency: The number of time units (clock cycles) between two initiations of a pipeline is the latency between them. Latency values must be non-negative integers.

Collision: When two or more initiations are done at same pipeline stage at the same time will cause a collision. A collision implies resource conflicts between two initiations in the pipeline, so it should be avoided.

Forbidden and Permissible Latency: Latencies that cause collisions are called **forbidden latencies**. (E.g. in above reservation table 2, 4, 5 and 7 are forbidden latencies).

Latencies that do not cause any collision are called **permissible latencies**. (E.g. in above reservation table 1, 3 and 6 are permissible latencies).

Latency Sequence and Latency Cycle: A **Latency Sequence** is a sequence of permissible non-forbidden latencies between successive task initiations.

A **Latency cycle** is a latency sequence which repeats the same subsequence (cycle) indefinitely.

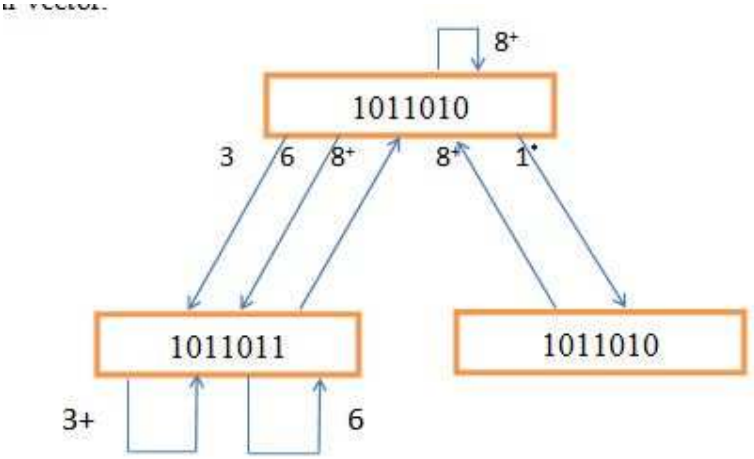
The **Average Latency** of a latency cycle is obtained by dividing the sum of all latencies by the number of latencies along the cycle.

The latency cycle (1, 8) has an average latency of (1+8)/2=4.5.

A **Constant Cycle** is a latency cycle which contains only one latency value. (E.g. Cycles (3) and (6) both are constant cycle).

Collision Vector: The combined set of permissible and forbidden latencies can be easily displayed by a **collision vector,** which is an m-bit (m<=n-1 in a n column reservation table) binary vector C=($C_mC_{m-1}….C_2C_1$). The value of C_i=1 if latency I causes a collision and C_i=0 if latency i is permissible. (E.g. C_x= (1011010)).

State Diagram: Specifies the permissible state transitions among successive initiations based on the collision vector.



The minimum latency edges in the state diagram are marked with asterisk.

Simple Cycle, Greedy Cycle and MAL: A **Simple Cycle** is a latency cycle in which each state appears only once. In above state diagram only (3), (6), (8), (1, 8), (3, 8), and (6, 8) are simple cycles. The cycle(1, 8, 6, 8) is not simple as it travels twice through state (1011010).

A **Greedy Cycle** is a simple cycle whose edges are all made with minimum latencies from their respective starting states. The cycle (1, 8) and (3) are greedy cycles.

MAL (Minimum Average Latency) is the minimum average latency obtained from the greedy cycle. In greedy cycles (1, 8) and (3), the cycle (3) leads to MAL value 3.

Instruction pipeline

An instruction pipeline reads consecutive instruction from memory while previous instruction are being executed in other segment.

Instruction Execution Phase

Consider an instruction pipeline of 4 stages

S1 : Fetch the instruction from the memory (FI).

S2 : Decode the instruction (DA).

S3 : Fetch the operands from the memory (FO).

S4 : Execute the instruction (EX).

Step/Instruction	1	2	3	4	5	6	7	8	9
1	FI	DA	FO	EX					
2		FI	DA	FO	EX				
3			FI	DA	FO	EX			
4				FI	DA	FO	EX		

Fig: Space timing Diagram

Que : Consider a program of 15,000 instructions executed by a linear pipeline processor with a clock rate of 25MHz. The instruction pipeline has five stages and one instruction is issued per clock cycle. Calculate speed up ratio, efficiency and throughput of this pipelined processor?

Soln: Time taken to execute without pipeline is = 15000 * 5* (1/25) microsecs

Time taken with pipeline = (15000 + 5 -1)*(1/ 25) microsecs

Speed up ratio = (15000*5*25) / (15000+ 5 -1)*25 = 4.99

Efficiency = Speed up ratio/ number of segment in pipeline = 4.99/5= 0.99

Throughput = number of task completed in unit time = 0.99 * 25 = 24.9 MIPS

Principles of designing pipeline processor

Buffers are used to speed close up the speed gap between memory access for either instructions or operands. Buffering can avoid unnecessary idling of the processing stages caused by memory access conflicts or by unexpected branching or interrupts. The concepts of busing eliminates the time delay to store and to retrieve intermediate results or to from the registers.

The computer performance can be greatly enhanced if one can eliminate unnecessary memory accesses and combine transitive or multiple fetch-store operations with faster register operations. This is carried by register tagging and forwarding.

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 arithmetic and logic executions in the functional pipes. The instruction or operand buffers provide a continuous supply of instructions or operands to the appropriate pipeline units. Buffering can avoid unnecessary idling of the processing stages caused by memory access conflicts or by unexpected branching or interrupts. Sometimes the entire loop instructions can be stored in the buffer to avoid repeated fetch of the same instructions loop, if the buffer size is sufficiently large. It is very large in the usage of pipeline computers.

Three buffer types are used in various instructions and data types. Instructions are fetched to the instruction fetch buffer before sending them to the instruction unit. After decoding, fixed point and floating point instructions and data are sent to their dedicated buffers. The store address and data buffers are used for continuously storing results back to memory. The storage conflict buffer is used only used when memory

Busing Buffers

The sub function being executed by one stage should be independent of the other sub functions being executed by the remaining stages; otherwise some process in the pipeline must be halted until the dependency is removed. When one instruction waiting to be executed is first to be modified by a future instruction, the execution of this instruction must be suspended until the dependency is released.

Another example is the conflicting use of some registers or memory locations by different segments of a pipeline. These problems cause additional time delays. An efficient internal busing structure is desired to route the resulting stations with minimum time delays.

In the AP 120B or FPS 164 attached processor the busing structure are even more sophisticated. Seven data buses provide multiple data paths. The output of the floating point adder in the AP 120B can be directly routed back to the input of the floating point adder, to the input of the floating point multiplier, to the data pad, or to the data memory. Similar busing is provided for the output of the floating point multiplier. This eliminates the time delay to store and to retrieve intermediate results or to from the registers.

Internal Forwarding and Register Tagging

To enhance the performance of computers with multiple execution pipelines

1. **Internal Forwarding** refers to a short circuit technique for replacing unnecessary memory accesses by register -to-register transfers in a sequence of fetch-arithmetic-store operations

2. **Register Tagging** refers to the use of tagged registers, buffers and reservations stations for exploiting concurrent activities among multiple arithmetic units.

The computer performance can be greatly enhanced if one can eliminate unnecessary memory accesses and combine transitive or multiple fetch-store operations with faster register operations. This concept of internal data forwarding can be explored in three directions. The symbols Mi and Rj to represent the ith word in the memory and jth fetch, store and register-to register transfer. The contents of Mi and Rj are represented by (Mi) and Rj

Store-Fetch Forwarding

The store the n fetch can be replaced by 2 parallel operations, one store and one register transfer.

2 memory accesses

Mi -> (R1) (store)

R2 -> (Mi) (Fetch)

Is being replaced by

Only one memory access

Mi -> (R1) (store)

R2 -> (R1) (register Transfer)

Fetch-Fetch Forwarding

The following fetch operations can be replaced by one fetch and one register transfer. One memory access has been eliminated.

2 memory accesses

R1 -> (Mi) (fetch)

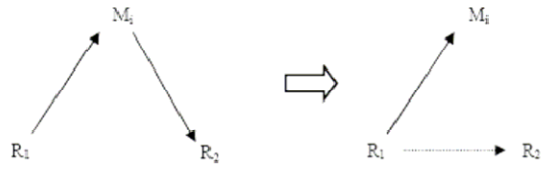
R2 -> (Mi) (Fetch)

Is being replaced by

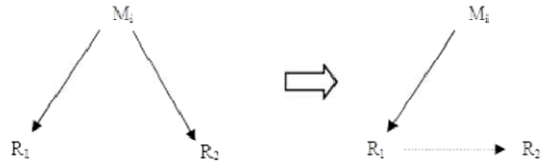
Only one memory access

R1 -> (Mi) (Fetch)

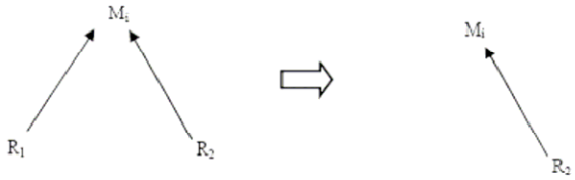
R2 -> (R1) (register Transfer)



Store – Fetch Forwarding



Fetch – Fetch Forwarding



Store-Store overwriting

Store-Store Overwriting

The following two memory updates of the same word can be combined into one; since the second store overwrites the first. 2 memory accesses

Mi -> (R1) (store)

Mi -> (R2) (store)

Is being replaced by

Only one memory access

Mi -> (R2) (store)

Forwarding and Data Hazards

Sometimes it is possible to avoid data hazards by noting that a value that results from one instruction is not needed until a late stage in a following instruction, and sending the data directly from the output of the first functional unit back to the input of the second one (which is sometimes the same unit). In the general case, this would require the output of every functional unit to be connected through switching logic to the input of every functional unit.

Data hazards can take three forms:

Read after write (RAW): Attempting to read a value that hasn't been written yet. This is the most common type, and can be overcome by forwarding.

Write after write (WAW): Writing a value before a preceding write has completed. This can only happen in complex pipes that allow instructions to proceed out of order, or that have multiple write-back stages (mostly CISC), or when we have multiple pipes that can write (superscalar).

Write after read (WAR): Writing a value before a preceding read has completed. These also require a complex pipeline that can sometimes write in an early stage, and read in a later stage. It is also possible when multiple pipelines (superscalar) or out-of-order issue are employed.

The fourth situation, read after read (RAR) does not produce a hazard.

Forwarding does not solve every RAW hazard situation. For example, if a functional unit is merely slow and fails to produce a result that can be forwarded in time, then the pipeline must stall. A simple example is the case of a load, which has a high latency. This is the sort of situation where compiler scheduling of instructions can help, by rearranging independent instructions to fill the delay slots. The processor can also rearrange the instructions at run time, if it has access to a window of prefetched instructions (called a prefetch buffer). It must perform much the same analysis as the compiler to determine which instructions are dependent on each other, but because the window is usually small, the analysis is more limited in scope. The small size of the window is due to the cost of providing a wide enough datapath to predecode multiple instructions at once, and the complexity of the dependence testing logic.

Branch Penalty Hiding

The control hazards due to branches can cause a large part of the pipeline to be flushed, greatly reducing its performance. One way of hiding the branch penalty is to fill the pipe behind the branch with instructions that would be executed whether or not the branch is taken. If we can find the right number of instructions that precede the branch and are independent of the test, then the compiler can move them immediately following the branch and tag them as branch delay filling instructions. The processor can then execute the branch,

and when it determines the appropriate target, the instruction is fetched into the pipeline with no penalty.

Of course, this scheme depends on how the pipeline is designed. It effectively binds part of the pipeline design for a specific implementation to the instruction set architecture. As we've seen before, it is generally a bad idea to bind implementation details to the ISA because we may need to change them later on. For example, if we decide to lengthen the pipeline so that the number of delay slots increases, we have to recompile our code to have it execute efficiently -- we no longer have strict binary compatibility among models of the same "ISA".

The filling of branch delays can be done dynamically in hardware by reordering instructions out of the prefetch buffer. But this leads to other problems. Another way to hide branch penalties is to avoid certain kinds of branches. For example, if we have

IF A < 0

THEN A = -A

we would normally implement this with a nearby branch. However, we could instead use an instruction that performs the arithmetic conditionally (skips the write back if the condition fails). The advantage of this scheme is that, although one pipeline cycle is wasted, we do not have to flush the rest of the pipe (also, for a dynamic branch prediction scheme, we need not put an extra branch into the prediction unit). These are called predicated instructions, and the concept can be extended to other sorts of operations, such as conditional loading of a value from memory.

Branch Prediction

Branches are the bane of any pipeline, causing a potentially large decrease in performance as we saw earlier. There are several ways to reduce this loss by predicting the action of the branch ahead of time.

Simple static prediction assumes that all branches will be taken or not. The designer decides which way is predicted from instruction trace statistics. Once the choice is made, the compiler can help by properly ordering local jumps. A slightly more complex static branch prediction heuristic is that backward branches are usually taken and forward branches are not (backwards taken, forwards not or BTFN). This assumes that most backward branches are loop returns and that most forward branches are the less likely cases of a conditional branch.

Compiler static prediction involves the use of special branches that indicate the most likely choice (taken or not, or more typically taken or other, since the most predictable branches are

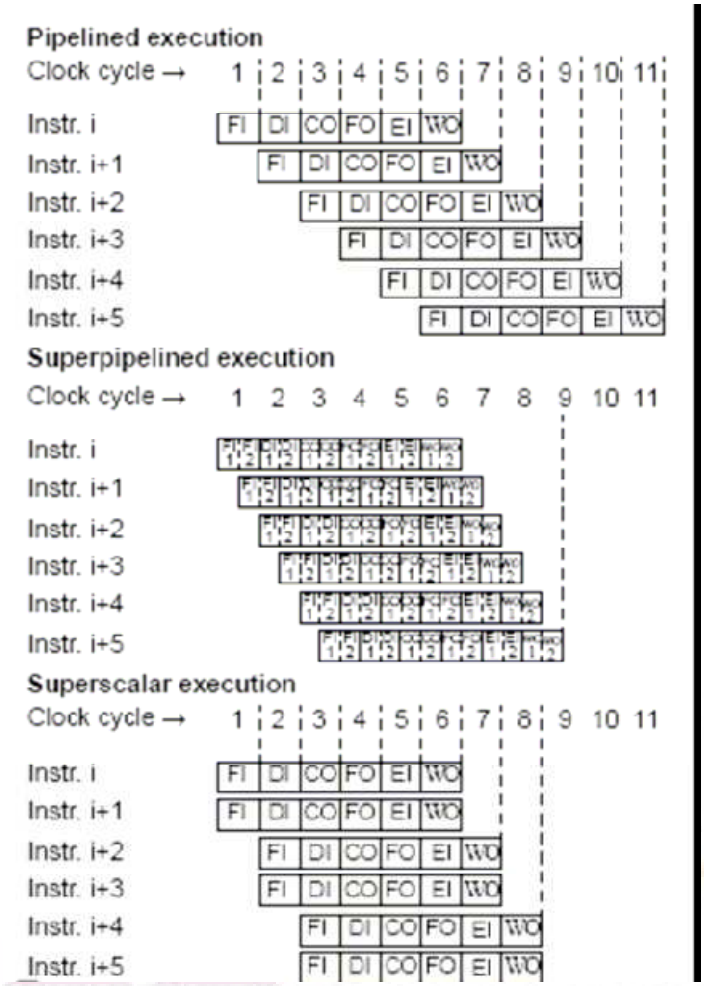
those at the ends of loops that are mostly taken). If the prediction fails in this case, then the usual cancellation of the instructions in the delay slots occurs and a branch penalty results.

Dynamic instruction scheduling

As discussed above the static instruction scheduling can be optimized by compiler the dynamic scheduling is achived either by using scoreboard or with Tomasulo's register tagging algorithm and discussed in superscalar processors.

superpipeline and Superscalar technique

Instruction level parallelism is obtained primarily in two ways in uniprocessors: through pipelining and through keeping multiple functional units busy executing multiple instructions at the same time. When a pipeline is extended in length beyond the normal five or six stages (e.g., I-Fetch, Decode/Dispatch, Execute, D-fetch, Writeback), then it may be called Superpipelined. If a processor executes more than one instruction at a time, it may be called Superscalar. A superscalar architecture is one in which several instructions can be initiated simultaneously and executed independently These two techniques can be combined into a Superscalar pipeline architecture.



Superpipeline

Superpipelining is based on dividing the stages of a pipeline into substages and thus increasing the number of instructions which are supported by the pipeline at a given moment. For example if we divide each stage into two, the clock cycle period t will be reduced to the half, t/2; hence, at the maximum capacity, the pipeline produces a result every t/2 s. For a given architecture and the corresponding instruction set there is an optimal number of pipeline stages; increasing the number of stages over this limit reduces the overall performance. A solution to further improve speed is the superscalar architecture. Given a pipeline stage time T, it may be possible to execute at a higher rate by starting operations at intervals of T/n. This can be accomplished in two ways:

_ Further divide each of the pipeline stages into n substages.

_ Provide n pipelines that are overlapped.

The first approach requires faster logic and the ability to subdivide the stages into segments with uniform latency. It may also require more complex inter-stage interlocking and stall-restart logic.

The second approach could be viewed in a sense as staggered superscalar operation, and has associated with it all of the same requirements except that instructions and data can be fetched with a slight offset in time. In addition, inter-pipeline interlocking is more difficult to manage because of the sub-clock period differences in timing between the pipelines.

Even so, staggered clock pipelines may be necessary with superscalar designs in the future, in order to reduce peak power and corresponding power-supply induced noise. Alternatively, designs may be forced to shift to a balanced mode of circuit operation in which logic transitions are balanced by reverse transitions -- a technique used in the Cray supercomputers that resulted in the computer presenting a pure DC load to the power supply, and greatly reduced noise in the system.

Inevitably, superpipelining is limited by the speed of logic, and the frequency of unpredictable branches. Stage time cannot productively grow shorter than the interstage latch time, and so this is a limit for the number of stages.

The MIPS R4000 is sometimes called a superpipelined machine, although its 8 stages really only split the I-fetch and D-fetch stages of the pipe and add a Tag Check stage. Nonetheless, the extra stages enable it to operate with higher throughput. The UltraSPARC's 9-stage pipe definitely qualifies it as a superpipelined machine, and in fact it is a Super-Super design because of its superscalar issue. The Pentium 4 splits the pipeline into 20 stages to enable increased clock rate. The benefit of such extensive

pipelining is really only gained for very regular applications such as graphics. On more irregular applications, there is little performance advantage.

Superscalar

Superscalar processing has its origins in the Cray-designed CDC supercomputers, in which multiple functional units are kept busy by multiple instructions. The CDC machines could pack as many as 4 instructions in a word at once, and these were fetched together and

dispatched via a pipeline. Given the technology of the time, this configuration was fast enough to keep the functional units busy without outpacing the instruction memory.

Current technology has enabled, and at the same time created the need to issue instructions in parallel. As execution pipelines have approached the limits of speed, parallel execution has been required to improve performance. As this requires greater fetch rates from memory, which hasn't accelerated comparably, it has become necessary to fetch instructions in parallel -- fetching serially and pipelining their dispatch can no longer keep multiple functional units busy. At the same time, the movement of the L1 instruction cache onto the chip has permitted designers to fetch a cache line in parallel with little cost.

In some cases superscalar machines still employ a single fetch-decode-dispatch pipe that drives all of the units. For example, the UltraSPARC splits execution after the third stage of a unified pipeline. However, it is becoming more common to have multiple fetch-decode-dispatch pipes feeding the functional units.

The choice of approach depends on tradeoffs of the average execute time vs. the speed with which instructions can be issued. For example, if execution averages several cycles, and the number of functional units is small, then a single pipe may be able to keep the units utilized. When the number of functional units grows large and/or their execution time approaches the issue time, then multiple issue pipes may be necessary.

Having multiple issue pipes requires

- being able to fetch instructions for that many pipes at once
- inter-pipeline interlocking
- reordering of instructions for multiple interlocked pipelines
- multiple write-back stages
- multiport D-cache and/or register file, and/or functionally split register file

Reordering may be either static (compiler) or dynamic (using hardware lookahead). It can be difficult to combine the two approaches because the compiler may not be able to predict the actions of the hardware reordering mechanism.

Superscalar operation is limited by the number of independent operations that can be extracted from an instruction stream. It has been shown in early studies on simpler processor models, that this is limited, mostly by branches, to a small number (<10, typically about 4).

More recent work has shown that, with speculative execution and aggressive branch prediction, higher levels may be achievable. On certain highly regular codes, the level of parallelism may be quite high (around 50). Of course, such highly regular codes are just as amenable to other forms of parallel processing that can be employed more directly, and are also the exception rather than the rule. Current thinking is that about 6-way instruction level parallelism for a typical program mix may be the natural limit, with 4-way being likely for integer codes. Potential ILP may be three times this, but it will be very difficult to exploit even a majority of this parallelism. Nonetheless, obtaining a factor of 4 to 6 boost in performance is quite significant, especially as processor speeds approach their limits.

Going beyond a single instruction stream and allowing multiple tasks (or threads) to operate at the same time can enable greater system throughput. Because these are naturally independent at the fine-grained level, we can select instructions from different streams to fill pipeline slots that would otherwise go vacant in the case of issuing from a single thread. In turn, this makes it useful to add more functional units. We shall further explore these multithreaded architectures later in the course.

Hardware Support for Superscalar Operation

There are two basic hardware techniques that are used to manage the simultaneous execution of multiple instructions on multiple functional units: Scoreboarding and reservation stations. Scoreboarding originated in the Cray-designed CDC-6600 in 1964, and reservation stations first appeared in the IBM 360/91 in 1967, as designed by Tomasulo.

Scoreboard

A scoreboard is a centralized table that keeps track of the instructions to be performed and the available resources and issues the instructions to the functional units when everything is ready for them to proceed. As the instructions execute, dependences are checked and execution is stalled as necessary to ensure that in-order semantics are preserved. Out of order execution is possible, but is limited by the size of the scoreboard and the execution rules. The scoreboard can be thought of as preceding dispatch, but it also controls execution after the issue. In a scoreboarded system, the results can be forwarded directly to their destination register (as long as there are no write after read hazards, in which case their execution is stalled), rather than having to proceed to a final write-back stage.

In the CDC scoreboard, each register has a matching Result Register Designator that indicates which functional unit will write a result into it. The fact that only one functional unit can be designated for writing to a register at a time ensures that WAW dependences cannot occur. Each functional unit also has a corresponding set of Entry-Operand Register Designators that indicate what register will hold each operand, whether the value is valid (or pending) and if it is pending, what functional unit will produce it (to facilitate forwarding). None of the operands is released to a functional unit until they are all valid, precluding RAW dependences. In addition , the scoreboard stalls any functional unit whose result would write a register that is still listed as an Entry-Operand to a functional unit that is waiting for an operand or is busy, thus avoiding WAR violations. An instruction is only allowed to issue if its specified functional unit is free and its result register is not reserved by another functional unit that has not yet completed. Four Stages of Scoreboard Control

1. Issue—decode instructions & check for structural hazards (ID1) If a functional unit for the instruction is free and no other active instruction has the same destination register (WAW), the scoreboard issues the instruction to the functional unit and updates its internal data structure. If a structural or WAW hazard exists, then the instruction issue stalls, and no further instructions will issue until these hazards are cleared.

2. Read operands—wait until no data hazards, then read operands (ID2) A source operand is available if no earlier issued active instruction is going to write it, or if the register containing the operand is being written by a currently active functional unit. When the source operands are available, the scoreboard tells the functional unit to proceed to read the operands from the registers and begin execution. The scoreboard resolves RAW hazards dynamically in this step, and instructions may be sent into execution out of order.

3. Execution—operate on operands (EX) The functional unit begins execution upon receiving operands. When the result is ready, it notifies the scoreboard that it has completed execution.

4. Write result—finish execution (WB) Once the scoreboard is aware that the functional unit has completed execution, the scoreboard checks for WAR hazards.

If none, it writes results. If WAR, then it stalls the instruction. Example:

DIVD F0,F2,F4

ADDD F10,F0,**F8**

SUBD **F8**,F8,F14

CDC 6600 scoreboard would stall SUBD until ADDD reads operands

Three Parts of the Scoreboard

1. Instruction status—which of 4 steps the instruction is in

2. Functional unit status—Indicates the state of the functional unit (FU). 9 fields for each functional unit

Busy—Indicates whether the unit is busy or not

Op—Operation to perform in the unit (e.g., + or –)

Fi—Destination register

Fj, Fk—Source-register numbers

Qj, Qk—Functional units producing source registers Fj, Fk

Rj, Rk—Flags indicating when Fj, Fk are ready and not yet read. Set to

No after operands are read.

3. Register result status—Indicates which functional unit will write each register, if one exists. Blank when no pending instructions will write that register

Scoreboard Implications

• provide solution for WAR, WAW hazards

• Solution for WAR – Stall Write in WB to allow Reads to take place; Read registers only during Read Operands stage.

• For WAW, must detect hazard: stall in the Issue stage until other completes

• Need to have multiple instructions in execution phase

• Scoreboard keeps track of dependencies, state or operations

– Monitors every change in the hardware.

– Determines when to read ops, when can execute, when can wb.

– Hazard detection and resolution is centralized.

Reservation Stations The reservation station approach releases instructions directly to a pool of buffers associated with their intended functional units (if more than one unit of a particular type is present, then the units may share a single station). The reservation stations are a distributed resource, rather than being centralized, and can be thought of as following dispatch. A reservation is a record consisting of an instruction and its requirements to execute -- its operands as specified by their sources and destination and bits indicating when valid values are available for the sources. The instruction is released to the functional unit when its requirements are satisfied, but it is important to note that satisfaction doesn't require an

operand to actually be in a register -- it can be forwarded to the reservation station for immediate release or to be buffered (see below) for later release. Thus, the reservation station's influence on execution can be thought of as more implicit and data dependent than the explicit control exercised by the scoreboard.

Tomasulo Algorithm

The hardware dependence resolution technique used **For IBM 360/91 about 3 years after CDC 6600.** Three Stages of Tomasulo Algorithm

1. Issue—get instruction from FP Op Queue

If reservation station free, then issue instruction & send operands (renames registers).

2. Execution—operate on operands (EX)

When both operands ready then execute; if not ready, watch CDB for result

3. Write result—finish execution (WB)

Write on Common Data Bus to all awaiting units; mark reservation station available.

Here the storage of operands resulting from instructions that completed out of order is done through renaming of the registers. There are two mechanisms commonly used for renaming. One is to assign physical registers from a free pool to the logical registers as they are identified in an instruction stream. A lookup table is then used to map the logical register references to their physical assignments. Usually the pool is larger than the logical register set to allow for temporary buffering of results that are computed but not yet ready to write back. Thus, the processor must keep track of a larger set of register names than the instruction set architecture specifies. When the pool is empty, instruction issue stalls.

The other mechanism is to keep the traditional association of logical and physical registers, but then provide additional buffers either associated with the reservation stations or kept in a central location. In either case, each of these "reorder buffers" is associated with a given instruction, and its contents (once computed) can be used in forwarding operations as long as the instruction has not completed.

When an instruction reaches the point that it may complete in a manner that preserves sequential semantics, then its reservation station is freed and its result appears in the logical register that was originally specified. This is done either by renaming the temporary register to be one of the logical registers, or by transferring the contents of the reorder buffer to the appropriate physical register.

Out of Order Issue

To enable out-of-order dispatch of instructions to the pipelines, we must provide at least two reservation stations per pipe that are available for issue at once. An alternative would be to rearrange instructions in the prefetch buffer, but without knowing the status of the pipes, it would be difficult to make such a reordering effective. By providing multiple reservation stations, however, we can continue issuing instructions to pipes, even though an instruction may be stalled while awaiting resources. Then, whatever instruction is ready first can enter the pipe and execute. At the far end of the pipeline, the out-of-order instruction must wait to be retired in the proper order. This necessitates a mechanism for keeping track of the proper order of instructions (note that dependences alone cannot guarantee that instructions will be properly reordered when they complete).

Superscalar-Superpipeline

Of course we may also combine superscalar operation with superpipelining. The result is potentially the product of the speedup factors.

However, it is even more difficult to interlock between parallel pipes that are divided into many stages. Also, the memory subsystem must be able to sustain a level of instruction throughput corresponding to the total throughput of the multiple pipelines -- stretching the processor/memory performance gap even more. Of course, with so many pipes and so many stages, branch penalties become huge, and branch prediction becomes a serious bottleneck (offsetting this somewhat is the potential for speculative branch execution that arises with a large number of pipes).

But the real problem may be in finding the parallelism required to keep all of the pipes and stages busy between branches. Consider that a machine with 12 pipelines of 20 stages must always have access to a window of 240 instructions that are scheduled so as to avoid all hazards, and that the average of 40 branches that would be present in a block of that size are all correctly predicted sufficiently in advance to avoid stalling in the prefetch unit. This is a tall order, even for highly regular code with a good balance of floating point to integer operations, let alone irregular code that is typically unbalanced in its operation mix. As usual, scientific code and image processing with their regular array operations often provide the best performance for a super-super processor. However, a vector unit could more directly address the array processing problems, and what is left to the scalar unit may not benefit nearly so

greatly from all of this complexity. Scalar processing can certainly benefit from ILP in many cases, but probably in patterns that differ from vector processing.

References/Suggested readings

Advance Computer architecture: Kai Hwang

UNIT IV

Topics Covered

Cache coherence, Snoopy protocols, Directory based protocols. Message routing schemes in multicomputer network, deadlock and virtual channel. Vector Processing Principles, Vector instruction types, Vector-access memory schemes. Vector supercomputer architecture, SIMD organization: distributed memory model and shared memory model. Principles of Multithreading: Multithreading Issues and Solutions, Multiple-Context Processors.

4.1 CACHE COHERENCY

Multiple copies of data, spread throughout the caches, lead to a coherence problem among the caches. The copies in the caches are coherent if they all equal the same value. However, if one of the processors writes over the value of one of the copies, then the copy becomes inconsistent because it no longer equals the value of the other copies. If data are allowed to become inconsistent (incoherent), incorrect results will be propagated through the system, leading to incorrect final results. Cache coherence algorithms are needed to maintain a level of consistency throughout the parallel system.

4.1.1 Cache–Memory Coherence

In a single cache system, coherence between memory and the cache is maintained using one of two policies: (1) write-through, and (2) write-back. When a task running on a processor P requests the data in memory location X, for example, the contents of X are copied to the cache, where it is passed on to P. When P updates the value of X in the cache, the other copy in memory also needs to be updated in order to maintain consistency. In write-through, the memory is updated every time the cache is updated, while in write-back, the memory is updated only when the block in the cache is being replaced. Table 4.1 shows the write-through versus write-back policies.

TABLE 4.1 Write-Through vs. Write-Back

		Write-Through		Write-Back	
Serial	Event	Memory	Cache	Memory	Cache
1		X		X	
2	P reads X	X	X	X	X
3	P updates X	X′	X′	X	X′

4.1.2 Cache–Cache Coherence

In multiprocessing system, when a task running on processor P requests the data in global memory location X, for example, the contents of X are copied to processor P's local cache, where it is passed on to P. Now, suppose processor Q also accesses X. What happens if Q wants to write a new value over the old value of X? There are two fundamental cache coherence policies: (1) write-invalidate, and (2) write-update. Write-invalidate maintains consistency by reading from local caches until a write occurs. When any processor updates the value of X through a write, posting a dirty bit for X invalidates all other copies. For example, processor

Q invalidates all other copies of X when it writes a new value into its cache. This sets the dirty bit for X. Q can continue to change X without further notifications to other caches because Q has the only valid copy of X. However, when processor P wants to read X, it must wait until X is updated and the dirty bit is cleared. Write-update maintains consistency by immediately updating all copies in all caches. All dirty bits are set during each write operation. After all copies have been updated, all dirty bits are cleared. Table 4.2 shows the write-update versus write-invalidate

policies.

TABLE 4.2 Write-Update vs. Write-Invalidate

		Write-Update		Write-Invalidate	
Serial	Event	P's Cache	Q's Cache	P's Cache	Q's Cache
1	P reads X	X		X	
2	Q reads X	X	X	X	X
3	Q updates X	X′	X′	INV	X′
4	Q updates X′	X″	X″	INV	X″

4.2 SNOOPY PROTOCOLS

Snoopy protocols are based on watching bus activities and carry out the appropriate coherency commands when necessary. Global memory is moved in blocks, and each block has a state associated with it, which determines what happens to the entire contents of the block. The state of a block might change as a result of the operations Read-Miss, Read-Hit, Write-Miss, and Write-Hit. A cache miss means that the requested block is not in the cache or it is in the cache but has been invalidated. Snooping protocols differ in whether they update or invalidate shared copies in remote caches in case of a write operation. They also differ as to where to obtain the new data in the case of a cache miss. In what follows we go over some examples of snooping protocols that maintain cache coherence.

4.2.1 Write-Invalidate and Write-Through

In this simple protocol the memory is always consistent with the most recently updated cache copy. Multiple processors can read block copies from main memory safely until one processor updates its copy. At this time, all cache copies are invalidated and the memory is updated to remain consistent.

4.3 DIRECTORY BASED PROTOCOLS

Owing to the nature of some interconnection networks and the size of the shared memory system, updating or invalidating caches using snoopy protocols might become unpractical. For example, when a multistage network is used to build a large shared memory system, the broadcasting techniques used in the snoopy protocols becomes very expensive. In such situations, coherence commands need to be sent to only those caches that might be affected

by an update. This is the idea behind directory-based protocols. Cache coherence protocols that somehow store
information on where copies of blocks reside are called directory schemes. A directory is a data structure that maintains information on the processors that share a memory block and on its state. The information maintained in the directory could

4.4 ROUTING IN MESSAGE PASSING NETWORKS

Routing is defined as the techniques used for a message to select a path over the network channels. Formally speaking, routing involves the identification of a set of permissible paths that may be used by a message to reach its destination, and a function, h, that selects one path from the set of permissible paths.

A routing technique is said to be adaptive if, for a given source and destination pair, the path taken by the message depends on network conditions, such as network congestion. Contrary to adaptive routing, a deterministic routing technique, also called oblivious, determines the path using only the source and destination, regardless of the network conditions. Although simple, oblivious routing techniques make inefficient use of the bandwidth available between the source and destination.

Routing techniques can also be classified based on the method used to make the routing decision as centralized (self) or distributed routing. In centralized routing the routing decisions regarding the entire path are made before sending the message. In distributed routing, each node decides by itself which channel should be used to forward the incoming message. Centralized routing requires complete knowledge of the status of the rest of the nodes in the network. Distributed routing requires knowledge of only the status of the neighboring nodes. Examples of the deterministic routing algorithms include the e-cube or dimension order routing used in the mesh and torus multicomputer networks and the XOR routing in the hypercube. The following example illustrates the use of a deterministic routing technique in a hypercube network.

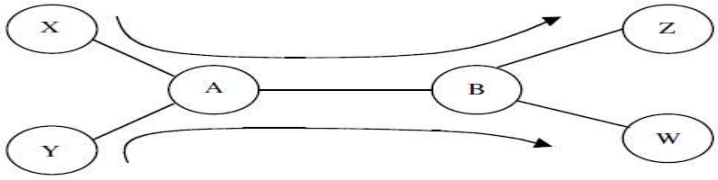
Example 1 Assume that $S = S_5S_4 \ldots S_1S_0$ to be the source node address, and that $D = D_5D_4 \ldots D_1D_0$ is the destination node address in a six-dimensional hypercube message passing system. Let $R = S \oplus D$ be the exclusive OR function executed bitwise for each node in the path. The results of the XOR-ing operation indicate the dimension in which the message should be sent in order to reach the destination. Consider the case whereby $S = 10(001010)$ and $D = 39(100111)$. Then $R = (101101)$; that is, the message has to be sent along dimensions 0, 2, 3, and 5 in order to reach the destination. The order in which these dimensions are traversed is not important. Let us assume that the message will follow the route by traversing the following dimensions 5, 3, 2, and 0. Then the route is totally determined as: $10(001010) \rightarrow 42(101010) \rightarrow 34(100010) \rightarrow 38(100110) \rightarrow 39(100111)$.

An n-dimensional mesh is defined as the interconnection structure that has $K_0 \times K_1 \times \cdots \times K_{n-1}$ nodes, where n is the number of dimensions of the network and K_i is the radix of dimension i. Each node is identified by an *n-coordinate* vector $(x_0, x_1, \ldots, x_{n-1})$, where $0 \leq x_i \leq K_i - 1$. A number of routing techniques have been used for mesh networks. These include *dimension-ordered*, *dimension reversal*, *turn model*, and *message flow model*. In the following, we introduce the dimension-ordered of *X-Y* routing.

Dimension-Ordered (X-Y) Routing : A channel numbering scheme often used in n-dimensional meshes is based on the dimension of channels. In dimension ordered routing, each packet is routed in one dimension at a time, arriving at the proper coordinate in each dimension before proceeding to the next dimension. By enforcing a strict monotonic order on the dimensions traversed, deadlock-free routing is guaranteed. In a two-dimensional mesh, each node is represented by its position (x, y); the packets are first sent along the x-dimension and then along the y-dimension, hence the name X-Y routing. In X-Y routing, messages are first sent along the X-dimension and then along the Y-dimension. In other words, at most one turn is allowed and that turn must be from the X-dimension to the Y-dimension. Let (sx, sy) and (dx, dy) denote the addresses of a source and destination node, respectively. Assume also that (gx, gy) =(dx _ sx, dy _ sy). The X-Y routing can be implemented by placing gx and gy in the first two flits, respectively, of the message. When the first flit arrives at a node, it is decremented or incremented, depending on whether it is greater than 0 or less than 0. If the result is not equal to 0, the message is forwarded in the same direction in which it arrived. If the result equals 0 and the message arrived on the Y-dimension, the message is delivered to the local node. If the result equals 0 and the message arrived on the X-dimension, the flit is discarded and the next flit is examined on arrival. If that flit is 0, the packet is delivered to the local node; otherwise,the packet is forwarded in the Y-dimension.

4.5 Virtual Channels

The principle of virtual channel was introduced in order to allow the design of deadlock-free routing algorithms. Virtual channels provide an inexpensive method to increase the number of logical channels without adding more wires. A number of adaptive routing algorithms are based on the use of virtual channels. A network without virtual channels is composed of single lane streets. Adding virtual channels to an interconnection network is analogous to adding lanes to a street network, thus allowing blocked messages to be passed. In addition to increasing throughput, virtual channels provide an additional degree of freedom in allocating resources to messages in a network. Consider the simple network shown in Figure .



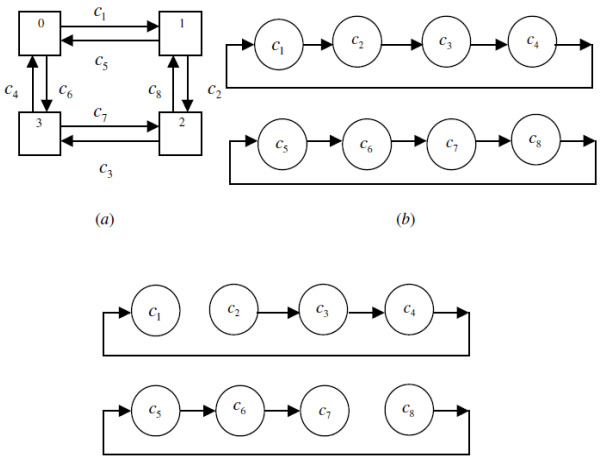
In this case, two paths X-A-B-Z and Y-A-B-W share the common link AB. It is, therefore, required to multiplex link AB between the two paths (two lanes). A provision is also needed such that data sent over the first path (lane) is sent from X to Z and not toW and similarly data sent over the second path (lane) is sent from Y to W and not to Z. This can be achieved if we assume that each physical link is actually divided into a number of unidirectional virtual channels. Each channel can carry data for one virtual circuit (one path). A circuit (path) from one node to another consists of a sequence of channels on the links along the path between the two nodes. When data is sent from node A to node B, then node B will have to determine the circuit associated with the data such that it can decide whether is should route the data to node Z or to node W. One way that can be used to provide such information is to divide the AB link into a fixed number of time slots and statically assign each time slot to a channel. This way, the time slot on which the data arrives identifies the sending channel and therefore can be used to direct the data to the appropriate destination. One of the advantages of the virtual channel concept is deadlock avoidance. This can be done by assigning a few flits per node of buffering. When a packet arrives at a virtual channel, it is put in the buffer and sent along the appropriate time slot.

4.6 Deadlock:

When two messages each hold the resources required by the other in order to move, both messages will be blocked. This is called a deadlock. It is a phenomenon that occurs whenever there exists cyclic dependency for resources. Management of resources in a network is the responsibility of the flow control mechanism used. Resources must be allocated in a manner that avoids deadlock. A straightforward, but inefficient, way to solve the deadlock problem is to allow rerouting (maybe discarding) of the messages participating in a deadlock situation.

Rerouting of messages gives rise to non minimal routing, while discarding messages requires that messages be recovered at the source and retransmitted. This preemptive technique leads to long latency and, therefore, is not used by most message passing networks.

A more common technique is to avoid the occurrence of deadlock. This can be achieved by ordering network resources and requiring that messages request use of these resources in a strict monotonic order. This restricted way for using network resources prevents the occurrence of circular wait, and hence prevents the occurrence of deadlock. The channel dependency graph (CDG) is a technique used to develop a deadlock-free routing algorithm. A CDG is a directed graph D =G(C, E), where the vertex set C consists of all the unidirectional channels in the network and the set of edges E includes all the pairs of connected channels, as defined by the routing algorithm. In other words, if (ci, cj) [E, then ci and cj are, respectively, an input channel and an output channel of a node and the routing algorithm may only route messages from ci to cj . A routing algorithm is deadlock-free if there are no cycles in its CDG. Consider, for example, the 4-node network shown in Figure a. The CDG of the network is shown in Figure b. There are two cycles in the CDG and therefore this network is subject to deadlock. Figure c shows one possible way to avoid the occurrence of deadlock, that is, disallowing messages to be forwarded from channel c1 to c2 and from c7 to c8 .



4.7 Vector Processing Principles

Vector instruction types, memory-access schemes for vector operands, and an overview of supercomputer families are given in this section.

Vector **Instruction** Types

Basic concepts behind vector processing are defined below. Then we discuss major types of vector instructions encountered in a modern vector machine. The intent is to acquaint the reader with the instruction-set architectures of available vector supercomputers.

Vector Processing **Definitions** A *vector* is a set of scalar data items, **all** of the same type, stored in memory. Usually, the vector elements are ordered to have a fixed addressing increment between successive elements called the *stride.* A *vector processor* is an ensemble of hardware resources, including vector registers, functional pipelines, processing elements, and register counters, for performing vector operations.

Vector processing occurs when arithmetic or logical operations are applied to vectors. It is distinguished from scalar processing which operates on one or one pair of data. The conversion from scalar code to vector code is called *vectorization.*

Vector instruction types

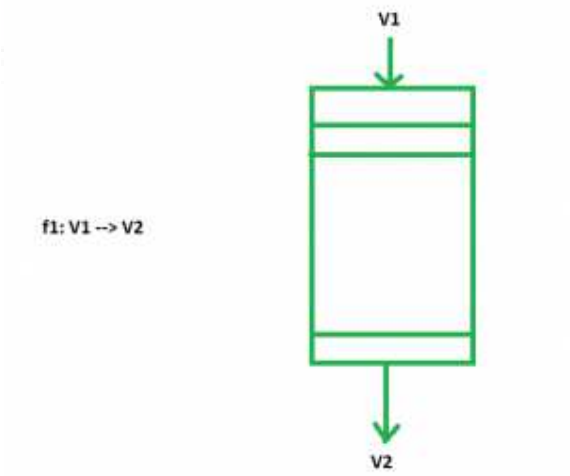
A **Vector operand** contains an ordered set of n elements, where n is called the **length of the ector**. All elements in a vector are same type scalar quantities, which may be a floating point number, an integer, a logical value, or a character.

Four primitive types of vector instructions are:

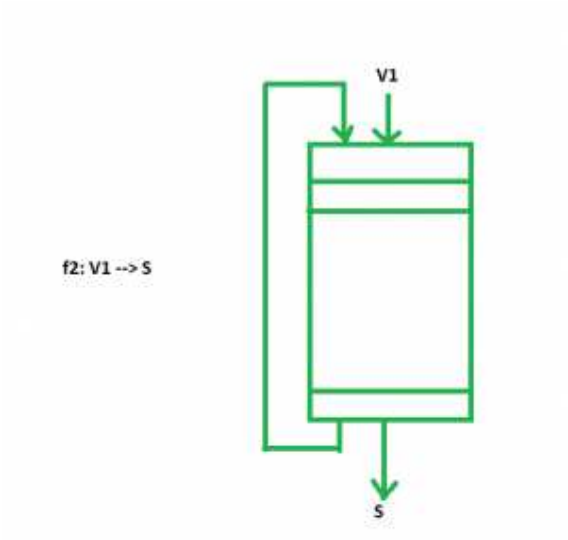
f1 : V --> V
f2 : V --> S
f3 : V x V --> V
f4 : V x S --> V

Where V and S denotes a vector operand and a scalar operand, respectively.

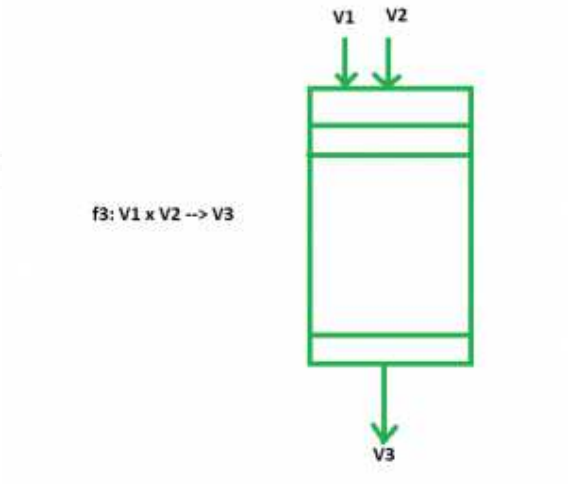
The instructions, f1 and f2 are unary operations and f3 and f4 are binary operations. The **VCOM (vector complement)**, which complements each complement of the vector, is an f1 operation. The pipe lined implementation of f1 operation is shown in the figure:



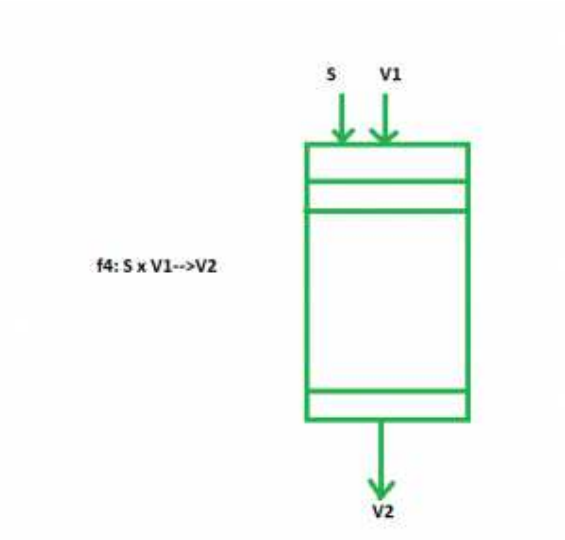
The **VMAX (vector maximum)**, which finds the maximum scalar quantity from all the complements in the vector, is an f2 operation. The pipe lined implementation of f2 operation is shown in the figure:



The **VMPL (vector multiply)**, which multiply the respective scalar components of two vector operands and produces another product vector, is an f3 operation. The pipe lined implementation of f3 operation is shown in the figure:



The **SVP (scalar vector product)**, which multiply one constant value to each component of the vector, is f4 operation. The pipe lined implementation of f4 operation is shown in the figure:



The Inputs are given as scalar components in the pipeline. Apart from these basic types of instructions, some special instructions may be used to facilitate the manipulation of vector data.

4.8 Vector-Access Memory Schemes

The flow of vector operands between the main memory and vector registers is usually pipelined with multiple access paths.

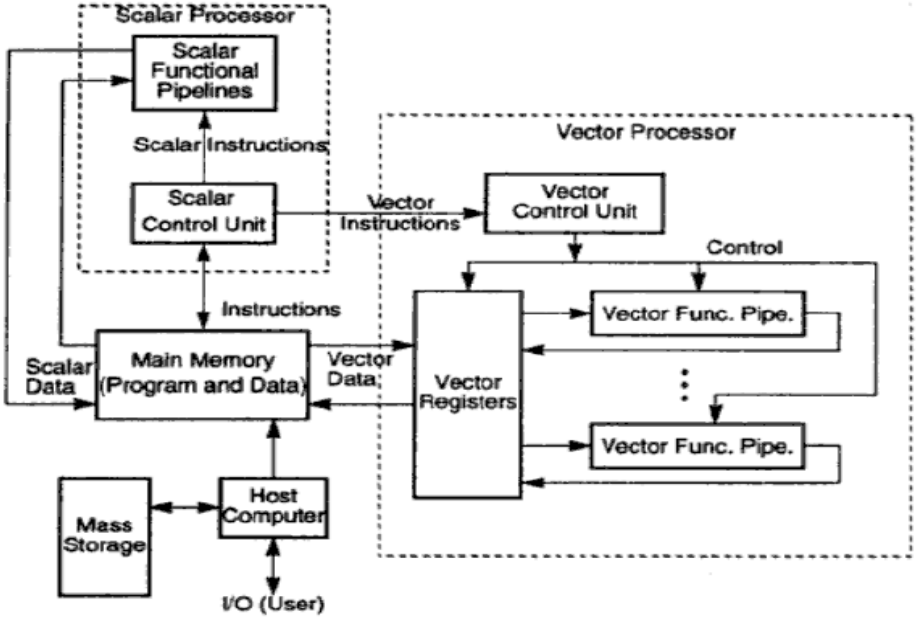
a) C-Access Memory Organization: **The** m-way low-order interleaved memory structure allows m memory words to be accessed concurrently in an overlapped manner. This *concurrent access* has been called *C-access* . The access cycles in different memory modules are staggered. The low-order *a* bits select the modules, and the high-order b bits select the word within each module, where $m = 2^a$ and $a + b =$ **n** is the address length.

b) **S-Access Memory Organization :** The low-order interleaved memory can be rearranged to allow *simultaneous access,* or *S-access,* . In this case, all memory modules are accessed simultaneously in a synchronized manner. Again the high-order (n — *a)* bits select the same offset word from each module. At the end of each memory cycle m = 2^a consecutive words are latched.

c) **S-Access Memory Organization :** A memory organization in which the C-access and S-access are combined is called *C/S-access.* In this scheme n access buses are used with m interleaved memory modules attached to each bus. The m modules on each bus arc m-way interleaved to allow C-access. The n buses operate in parallel to allow S-access. In each memory cycle, at most m • n words are fetched if the n buses are fully used with pipelined memory accesses. The C/S-access memory is suitable for use in vector multiprocessor configurations. It provides parallel pipelined access of a vector data set with high bandwidth. A special *vector cache* design is needed within each processor in order to guarantee smooth data movement between the memory and multiple vector processors.

4.9 Vector Supercomputers

A vector computer is often built on top of a scalar processor. As shown in figure the vector processor is attached to the scalar processor as an optional feature. Program and data are first loaded into the main memory through a host computer. All instructions are first decoded by the scalar control unit- If the decoded instruction is a scalar operation or a program control operation, it will be directly executed by the scalar processor using the scalar functional pipelines. If the instruction is decoded as a vector operation, it will be sent to the vector control unit. This control unit will supervise the flow of vector data between the main memory and vector functional pipelines. The vector data flow is coordinated by the control unit. A number of vector functional pipelines may be built into a vector processor.



Vector Processor Models Figure above shows a *register-to-register* architecture. Vector registers are used to hold the vector operands, intermediate and final vector results. The vector functional pipelines retrieve operands from and put results into the vector registers. AH vector registers are programmable in user instructions. Each vector register is equipped with a component counter which keeps track of the component registers used in successive pipeline cycles.

4.10 SIMD Supercomputers

In Fig. , we have shown an abstract model of SIMD computers having a single instruction stream over multiple data streams. SIMD computer is specified by a 5-tuple:

$M = (N,C,J,M,R)$

where

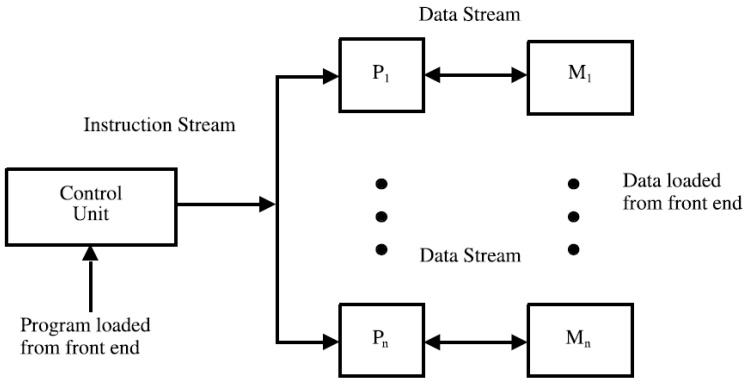
(1) N is the number of *processing elements* (PEs) in the machine. For example, the IUiac IV has 64 PEs and the Connection Machine CM-2 uses 65,536 PEs.

(2) *C* is the set of instructions directly executed by the *control unit* (CU), including scalar and program flow control instructions.

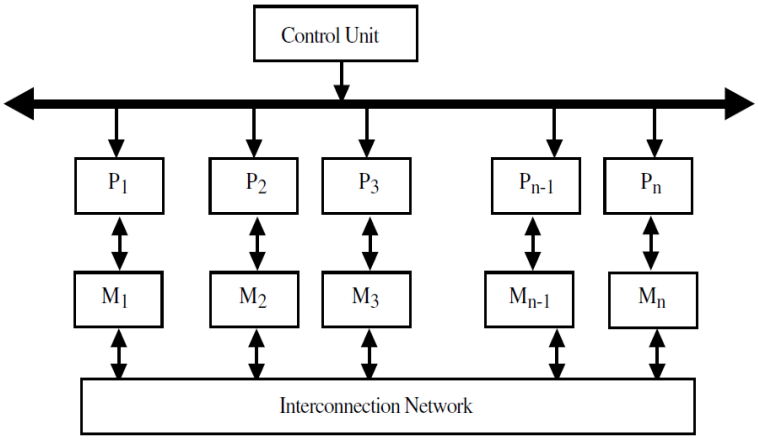
(3) / i s the set of instructions broadcast by the CU to all PEs for parallel execution. These include arithmetic, logic, data routing, masking, and other local operations executed by each active PE over data within that PE.

(4) *M* is the set of masking schemes, where each mask partitions the set of PEs into enabled and disabled subsets.

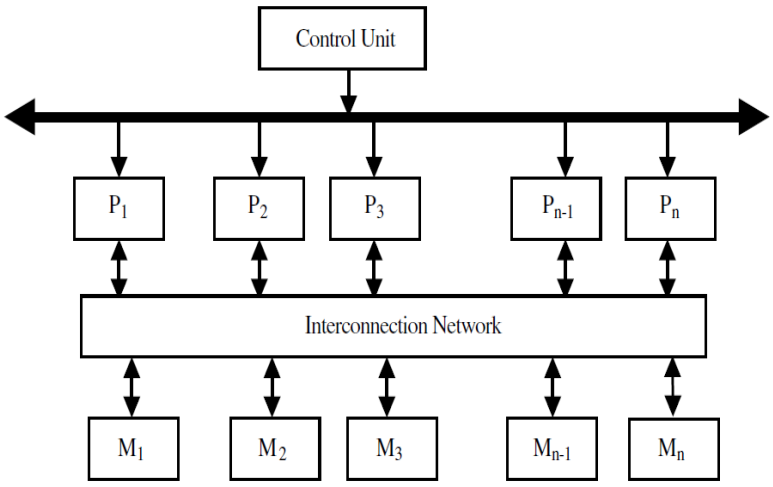
(5) *R* is the set of data-routing functions, specifying various patterns to be set up in the interconnection network for inter-PE communications



There are two main configurations that have been used in SIMD machine . In the first scheme, each processor has its own local memory. Processors can communicate with each other through the interconnection network. If the interconnection network does not provide direct connection between a given pair of processors, then this pair can exchange data via an intermediate processor.



SIMD 1 Scheme : Distributed Memory Model



SIMD 2 Scheme : Shared Memory Model

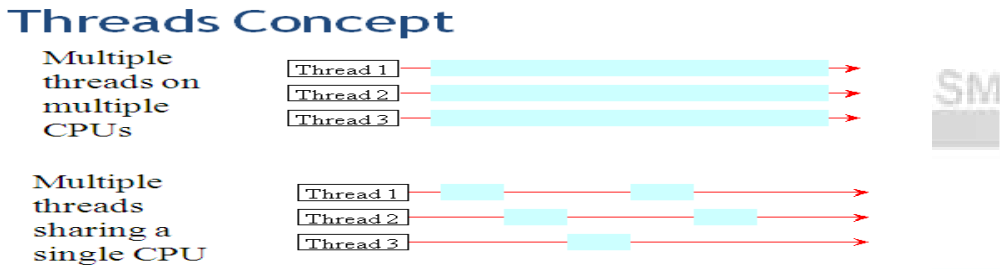
4.11 Thread

An important aspect of modern operating systems is their support for threads within processes. A thread, sometimes called a lightweight process, is a basic unit of processor utilization. It runs sequentially on a processor and is interruptible so that the processor can switch to other threads. A process does nothing if it has no threads in it, and a thread must be in exactly one process. A thread is different from the traditional or heavy-weight process, which is equivalent to a task with one thread. Context switching among peer threads is relatively inexpensive, compared with

context switching among heavy-weight processes.

Concurrency among processes can be exploited because threads in different processes may execute concurrently. Moreover, multiple threads within the same process can be assigned to different processors. Threads can be used to send and receive messages while other operations within a task continue. For example, a task might have many threads waiting to receive and process request messages. When a message is received by one thread, it is processed by this thread concurrently with other threads processing other messages. Java has support for multithreading built in the language. Java threads are usually mapped to real operating system threads if the underlying operating system supports multithreads. Thus, applications written in

Java can be executed in parallel on a multiprocessor environment.



UNIT V

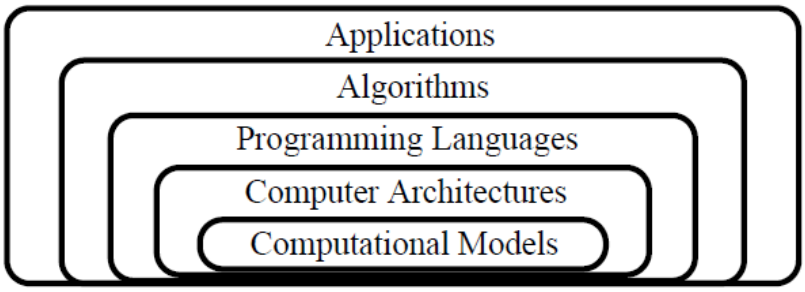
TOPICS COVERED

Parallel Programming Models, Shared-Variable Model, Message-Passing Model, Data-Parallel Model, Object-Oriented Model, Functional and Logic Models, Parallel Languages and Compilers, Language Features for Parallelism, Parallel Programming Environment, Software Tools and Environments.

5.1 Parallel Programming Models

Many computer designs achieve higher performance by operating various parts of the computer system concurrently. Even Charles Babbage, considered the Father of Computing, utilized parallelism in the design of his Analytical Engine. His Analytical Engine was designed in the 1820s and 1830s and was to be a mechanical device operated by hand crank. The arithmetic unit was designed to perform 50-digit calculations at the following speeds: add or subtract in one second; multiply or divide in one minute. To achieve such high speeds with mechanical parts, Babbage devised, after years of work, a parallel addition algorithm with anticipatory carry logic. [Kuck, 1978]. At a higher level of parallelism, the Analytical Engine performed indexing arithmetic on an index register for loop counting in parallel, with its arithmetic unit [Kuck, 1978]. Unfortunately, Babbage was never able to build his Analytical Engine because technology took over one hundred years to advance to the point where his design could be implemented.

Research in parallel algorithms has been in two different arenas: scientific computing and the theory of algorithms. Researchers in scientific computing are interested in the design and analysis
of numerical algorithms . Researchers in the theory of algorithms have developed theoretical models of parallelism and analyzed the behavior of parallel algorithms based on these models. In order to deal with this massive amount of subject material, we need a way to organize it.



A computational model attempts to define and clarify one of the many ways we view computation. By proposing an abstract machine or a mathematical model, a researcher tries to characterize the essence of one way of computation to form a theoretical basis for other research.

5.2 Shared-Variable Model

A shared-memory multiprocessor can be modeled as a complete graph, in which every node is connected to every other node, as shown in Fig. 2 for $p = 9$. In the 2D mesh of Fig. 1, Processor 0 can send/receive data directly to/from P1 and P 3 . However, it has to go through an intermediary to send/receive data to/from P 4 , say. In a shared-memory multiprocessor, every piece of data is directly accessible to every processor (we assume that each processor can simultaneously send/receive data over all of its $p - 1$ links). The diameter $D = 1$ of a complete graph is an indicator of this direct access. The node degree $d = p - 1$, on the other hand, indicates that such an architecture would be quite costly to implement if no restriction is placed on data accesses.

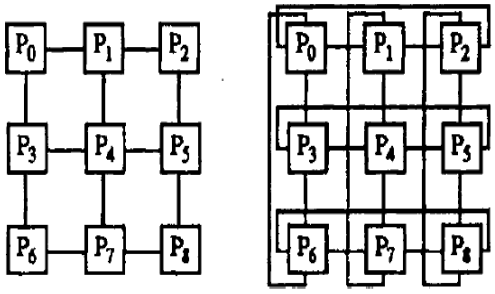


FIGURE 1. A 2D mesh of nine processors and its torus variant.

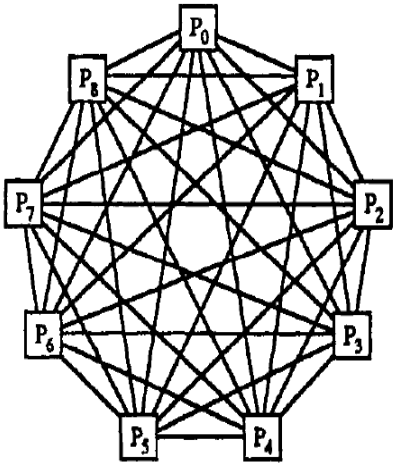


Figure 2 : A shared-variable architecture modeled as a complete graph.

5.3 Message-Passing Model

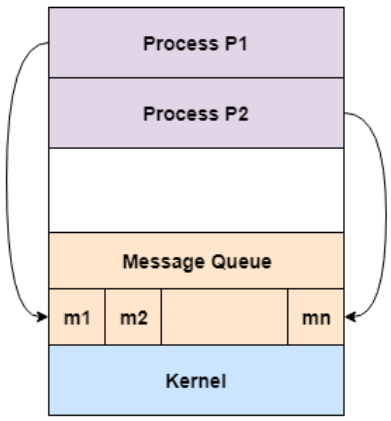
In the message-passing library approach to parallel programming, a collection of processes executes programs written in a standard sequential language augmented with calls to a library of functions for sending and receiving messages. In this chapter, we introduce the key

concepts of message-passing programming and show how designs developed using the techniques discussed in Part I can be adapted for message-passing execution. For concreteness, we base our presentation on the Message Passing Interface (MPI), the de facto message-passing standard. However, the basic techniques discussed are applicable to other such systems, including p4, PVM, Express, and PARMACS.

MPI is a complex system. In its entirety, it comprises 129 functions, many of which have numerous parameters or variants. As our goal is to convey the essential concepts of message-passing programming, not to provide a comprehensive MPI reference manual, we focus here on a set of 24 functions and ignore some of the more esoteric features. These 24 functions provide more than adequate support for a wide range of applications.

Message passing model allows multiple processes to read and write data to the message queue without being connected to each other. Messages are stored on the queue until their recipient retrieves them. Message queues are quite useful for interprocess communication and are used by most operating systems.

A diagram that demonstrates message passing model of process communication is given as follows



Message Passing Model

n the above diagram, both the processes P1 and P2 can access the message queue and store and retrieve data.

Advantages of Message Passing Model

Some of the advantages of message passing model are given as follows −

- The message passing model is much easier to implement than the shared memory model.
- It is easier to build parallel hardware using message passing model as it is quite tolerant of higher communication latencies.

Disadvantage of Message Passing Model

The message passing model has slower communication than the shared memory model because the connection setup takes time.

5.4 Object-Oriented Model

Object-oriented development offers a different model from the traditional software development approach, which is based on functions and procedures. In simplified terms, object-oriented systems development is a way to develop software by building self-contained modules or objects that can be easily replaced, modified, and reused.

In an object-oriented environment,

software is a collection of discrete objects that encapsulate their data as well as the functionality to model real-world "objects."

An object orientation yields important benefits to the practice of software construction

Each object has attributes (data) and methods (functions).

Objects are grouped into classes; in object-oriented terms, we discover and describe the classes involved in the problem domain.

everything is an object and each object is responsible for itself.

Object-oriented methods enable us to create sets of objects that works together synergistically to produce software that better model their problem domains than similar systems produced by traditional techniques. The systems are easier to adapt to changing requirements, easier to maintain, more robust, and promote greater de-sign and code reuse. Object-oriented development allows us to create modules of functionality. Once objects are defined, it can be taken for granted that they will perform their desired functions and you can seal them off in

your mind like black boxes. Your attention as a programmer shifts to what they do rather than how they do it. Here are some reasons why object orientation works

In an object-oriented environment, software is a collection of discrete objects that encapsulate their data and the functionality to model real-world "objects." Once objects are defined, you can take it for granted that they will perform their desired functions and so seal them off in your mind like black boxes. Your attention as a programmer shifts to what they do rather than how they do it. The object-oriented life cycle encourages a view of the world as a system of cooperative and collaborating gents. An object orientation produces systems that are easier to evolve, more flexible, more robust, and more reusable than a top-down structure approach.

An object orientation

Allows working at a higher level of abstraction.

Provides a seamless transition among different phases of software development.

Encourages good development practices.

Promotes reusability.

5.5 Data Parallel Models

Parallel processing has been developed as an effective technology in modern computers to meet the demand for higher performance, lower cost and accurate results in real-life applications. Concurrent events are common in today's computers due to the practice of multiprogramming, multiprocessing, or multicomputing.

Modern computers have powerful and extensive software packages. To analyze the development of the performance of computers, first we have to understand the basic development of hardware and software.

- **Computer Development Milestones** − There is two major stages of development of computer - **mechanical** or **electromechanical** parts. Modern computers evolved after the introduction of electronic components. High mobility electrons in electronic computers replaced the operational parts in mechanical computers. For information transmission, electric signal which travels almost at the speed of a light replaced mechanical gears or levers.

- **Elements of Modern computers** − A modern computer system consists of computer hardware, instruction sets, application programs, system software and user interface.

he computing problems are categorized as numerical computing, logical reasoning, and transaction processing. Some complex problems may need the combination of all the three processing modes.

- **Evolution of Computer Architecture** − In last four decades, computer architecture has gone through revolutionary changes. We started with Von Neumann architecture and now we have multicomputers and multiprocessors.
- **Performance of a computer system** − Performance of a computer system depends both on machine capability and program behavior. Machine capability can be improved with better hardware technology, advanced architectural features and efficient resource management. Program behavior is unpredictable as it is dependent on application and run-time conditions

Multiprocessors and Multicomputers

In this section, we will discuss two types of parallel computers −

- Multiprocessors
- Multicomputers

Shared-Memory Multicomputers

Three most common shared memory multiprocessors models are −

Uniform Memory Access (UMA)

In this model, all the processors share the physical memory uniformly. All the processors have equal access time to all the memory words. Each processor may have a private cache memory. Same rule is followed for peripheral devices.

When all the processors have equal access to all the peripheral devices, the system is called a **symmetric multiprocessor**. When only one or a few processors can access the peripheral devices, the system is called an **asymmetric multiprocessor**.

Distributed - Memory Multicomputers − A distributed memory multicomputer system consists of multiple computers, known as nodes, inter-connected by message passing network. Each node acts as an autonomous computer having a processor, a local memory and sometimes I/O devices. In this case, all local memories are private and are accessible only to the local processors. This is why, the traditional machines are called **no-remote-memory-access (NORMA)** machines.

Parallel Random-Access Machines

Sheperdson and Sturgis (1963) modeled the conventional Uniprocessor computers as random-access-machines (RAM). Fortune and Wyllie (1978) developed a parallel random-access-machine (PRAM) model for modeling an idealized parallel computer with zero memory access overhead and synchronization.

An N-processor PRAM has a shared memory unit. This shared memory can be centralized or distributed among the processors. These processors operate on a synchronized read-memory, write-memory and compute cycle. So, these models specify how concurrent read and write operations are handled.

Following are the possible memory update operations −

- **Exclusive read (ER)** − In this method, in each cycle only one processor is allowed to read from any memory location.
- **Exclusive write (EW)** − In this method, at least one processor is allowed to write into a memory location at a time.
- **Concurrent read (CR)** − It allows multiple processors to read the same information from the same memory location in the same cycle.
- **Concurrent write (CW)** − It allows simultaneous write operations to the same memory location. To avoid write conflict some policies are set up.

5.6 Functional Programming

Functional programming languages are specially designed to handle symbolic computation and list processing applications. Functional programming is based on mathematical functions. Some of the popular functional programming languages include: Lisp, Python, Erlang, Haskell, Clojure, etc.

Functional programming languages are categorized into two groups, i.e. −

- **Pure Functional Languages** − These types of functional languages support only the functional paradigms. For example − Haskell.
- **Impure Functional Languages** − These types of functional languages support the functional paradigms and imperative style programming. For example − LISP.

Functional Programming – Characteristics

The most prominent characteristics of functional programming are as follows −

- Functional programming languages are designed on the concept of mathematical functions that use conditional expressions and recursion to perform computation.
- Functional programming supports **higher-order functions** and **lazy evaluation** features.
- Functional programming languages don't support flow Controls like loop statements and conditional statements like If-Else and Switch Statements. They directly use the functions and functional calls.
- Like OOP, functional programming languages support popular concepts such as Abstraction, Encapsulation, Inheritance, and Polymorphism.

Functional Programming – Advantages

Functional programming offers the following advantages −

- **Bugs-Free Code** − Functional programming does not support **state**, so there are no side-effect results and we can write error-free codes.

- **Efficient Parallel Programming** − Functional programming languages have NO Mutable state, so there are no state-change issues. One can program "Functions" to work parallel as "instructions". Such codes support easy reusability and testability.
- **Efficiency** − Functional programs consist of independent units that can run concurrently. As a result, such programs are more efficient.
- **Supports Nested Functions** − Functional programming supports Nested Functions.
- **Lazy Evaluation** − Functional programming supports Lazy Functional Constructs like Lazy Lists, Lazy Maps, etc.

As a downside, functional programming requires a large memory space. As it does not have state, you need to create new objects every time to perform actions.

Functional Programming is used in situations where we have to perform lots of different operations on the same set of data.

- Lisp is used for artificial intelligence applications like Machine learning, language processing, Modeling of speech and vision, etc.
- Embedded Lisp interpreters add programmability to some systems like Emacs.

5.7 Logic programming

Logic programming is a computer programming paradigm in which program statements express facts and rules about problems within a system of formal logic. Rules are written as logical clauses with a head and a body; for instance, "H is true if B1, B2, and B3 are true." Facts are expressed similar to rules, but without a body; for instance, "H is true."

Some logic programming languages, such as Datalog and ASP (Answer Set Programming), are purely declarative. They allow for statements about what the program should accomplish, with no explicit step-by-step instructions on how to do so. Others, such as Prolog, are a combination of declarative and imperative. They may also include procedural statements, such as "To solve H, solve B1, B2, and B3."

Languages used for logic programming

- Absys
- ALF (algebraic logic functional programming language).
- Algorithmic program debugging
- Alice
- Alma-0
- ASP (Answer Set Programming)
- CHIP
- Ciao
- CLACL

Logic programming is a programming paradigm which is largely based on formal logic. Any program written in a logic programming language is a set of sentences in logical form, expressing facts and rules about some problem domain. Major logic programming language families include Prolog, answer set programming (ASP) and Datalog. In all of these languages, rules are written in the form of *clauses*:

$H :- B_1, \ldots, B_n.$

and are read declaratively as logical implications:

H if B_1 and … and B_n.

H is called the *head* of the rule and $B_1, \ldots, B_n$ is called the *body*. Facts are rules that have no body, and are written in the simplified form:

$H.$

In the simplest case in which $H, B_1, \ldots, B_n$ are all atomic formulae, these clauses are called definite clauses or Horn clauses. However, there are many extensions of this simple case, the most important one being the case in which conditions in the body of a clause can also be negations of atomic formulas. Logic programming languages that include this extension have the knowledge representation capabilities of a non-monotonic logic.

In ASP and Datalog, logic programs have only a declarative reading, and their execution is performed by means of a proof procedure or model generator whose behaviour is not meant to be controlled by the programmer. However, in the Prolog family of languages, logic programs also have a procedural interpretation as goal-reduction procedures:

to solve H, solve B_1, and ... and solve B_n.

Consider the following clause as an example:

fallible(X) :- human(X).

5.8 Parallel Languages and Compilers

Linda is a parallel programming environment that uses generative communication. Gelernter describes generative communication and argues that it is sufficiently different from the other three basic mechanisms of concurrent programming (monitors, message passing, and remote operations) as to make it a fourth model. It differs from the other models in requiring that messages be added in tuple form to an environment called tuple space where they exist independently until a process chooses to remove them. While Linda is best suited to building distributed data structure programs involving many worker processes attacking the structure simultaneously, it also works well with more traditional methods of employing parallelism. Furthermore, because it is a high-level programming tool, Linda can model both the shared-memory as well as message-passing style of programming regardless of the underlying architecture. We describe three compatible implementations of Linda--two completed, and one still in the development stage. The two completed implementations are built on the p4 parallel programming system and take advantage of shared-memory architectures and message-passing systems, respectively. We plan to build the third implementation on p5, the successor to p4.

5.9 Language Features for Parallelism and Parallel Programming Environment

1. Introduction : The environment f or parallel computers is much more demanding than that for sequential computers The programming environment is a collection of software tools and system software support To break this hardware/software barrier , we need a parallel software environment Which provide better tools for user to implement parallelism and to debug programs

2. LANGUAGE FEATURES Features are idealized for general purpose application. Some of the features are identified with existing language /compiler development. Language features are classified into six categories

3. CATEGORIES Optimization Availability features. features. Synchronization /communication Control of parallelism Data parallelism Process features management features features.

4. OPTIMIZATION FEATURES This features converting sequentially coded programs into parallel forms. The purpose is to match the software with the hardware Parallelism in the target machine. Automated parallelizer(alliant FX fortan) Semiautomed Interractive parallelizer(programers intrection) restructure support(static analyser , run time static , data flow graph,)

5. AVAILABILTY FEATURES Feature enhance user friendliness . Make the language portable to a larger class of parallel computer Expand the applicability of software libraries Scalabilty – scalable to the number of processors available and independent of hardware topology Compatibility-compatible with establishment sequential Portability –portable to shared memory multiprocessors , message passing multicomputers , or both

6. SYNC/COMMUN FEATURES Single assignment Remote languages producer call Data flow languages such as ID Send /receive for message passing Barriers ,mailbox , semaphores , monitors

7. CONTROL OF PARALLELISM Coarse ,medium, or fine grains Explicit Global Take versus implicit parallelism parallelism in the entire program spilt parallelism Shared task queue

8. DATA PARALLELISM FEATURES Used to specify how data are accessed and distributed in either SIMD and MIMD computers Run- time automatic decomposition Mapping specification Virtual Direct processor support access to shared data SPMD(single program multiple data)

9. PROCESS MANAG FEATURES Needed to support the efficient creation of parallel processes , implementation of multithreading or multitasking. Dynamic process creation at run time Light weight processes(threads)- compare to UNIX(heavyweight)processes Replicated work Partitioned networks Automatic load balancing

10. COMPILERS Using It's high level language in source code become a necessity in modern computer ROLE OF COMPILER Remove the burden of program optimization and code generation from the programmer.

11. THREE PHASES OF COMPILER FLOW ANALYSIS OPTIMIZATION CODE GENERATION

12. FLOW ANALYSIS Program flow pattern in order to determine data and control dependence in the source code Flow analysis is conducted at different execution levels on different parallel computers Instruction level parallelism is exploited in super scalar or VLSI processors , loop level in SIMD , vector, Task level in multiprocessors , multicomputer , or a network workstation

13. OPTIMIZATION The transformation of user programs in order to explore the hardware capabilities as much as possible Transformation can be conducted at the loop level , locality level , or prefetching level The ultimate goal of PO is to maximize the speed of code execution It involves minimization of code length and of memory accesses and the exploitation Sometimes should be conducted at the algorithmic level and must involves the programmer

14. CODE GENERATION Code generation usually involves transformation from one representation to another ,called an intermediate form Even more demanding because parallel constructs must be included Code generation closely tied to instruction scheduling policies used Optimized to encourage a high degree of parallelism Parallel code generation is very different fo different computer classes they are software and hardware scheudled

5.10 Parallel Programming Tools and Environments

HPF

- HPF Forum
- High Performance Fortran Resource List
- CMFotran
- High Performance Fortran / Fortran 90D Compiler at Syracuse
- IBM HPF
- The D System
- PARADIGM
- Excalibur
- Fotran Parallel Programming Systems

Message Passing

- MPI
- PVM
- BLACS
- PICL
- Interprocessor Collective Communications Library (iCC)
- Pade

Active Messages

- CMMD and Active Messages

Fortran

- FX
- Fortran M (for lack of a better place)

Parallel C

- Split-C
- CILK
- Jade
- CID
- AC

Parallel C++

- CC++ Programming Language
- HPC++
- pC++
- Mentat
- ORCA
- uC++

Compiler Infrastructure

- SUIF
- Titanium

Regular Data Structures

- PBLAS
- Global Arrays

Irregular Data Structure Libraries

- PETSc 2.0 for MPI
- Multipol

- LPARX
- KELP

Process Structure

- BLACS Process Grids

Numeric Libraries

- **LAPACK** - Linear Algebra PACKage for high performance workstations and shared memory parallel computers. The **LAPACK Manual** is available on-line. (This and much other useful numerical software is available on Netlib.)
- **ScaLAPACK** - Scalable Linear Algebra PACKage for high performance distributed memory parallel computers
- **Templates for the Solution of Linear Systems**: Building Blocks for Iterative Methods is a hyper-text book on iterative methods for solving systems of linear equations.
- IML++ (Iterative Methods Library) v. 1.2a
- CMSSL

Scheduling Tools

- LSF
- Prospero

Debugging Support

- Prism
- P2D2
- Total View
- Mantis

Performance Analysis Tools

- ACTS

- AIMS
- Pablo
- Paradyn
- Delphi - Integrated, Language-Directed Performance Prediction, Measurement, and Analysis

Environment

- PARL
- PPP

Distributed Shared Virtual Memory

- CRL
- KOAN
- Locust
- Mirage
- CDS
- Quarks
- Midway
- Treadmarks
- Munin
- Simple COMA

Clusters

- UCB NOW
- Mosix
- Condor
- Codine

Wide-area Computing

- Legion

- NetSolve
- Globus

Less Traditional Languages

- NESL
- Modula2*
- Impala
- LINDA
- CODE Visual programming
- SISAL

BIBILIOGRAPHY

1. Kai Hwang, “Advanced computer architecture”, TMH.
2. J.P.Hayes, “computer Architecture and organization”; MGH.
3. V.Rajaranam & C.S.R.Murthy, “Parallel computer”; PHI Learning.
4. Kain,”Advance Computer Architecture: -A System Design Approach”, PHI Learning
5. M.J Flynn, “Computer Architecture, Pipelined and Parallel Processor Design”; Narosa Publishing.
6. Hwang and Briggs, “Computer Architecture and Parallel Processing”; MGH.
7.Tutorialspoint.org