3.0 Objective

The main objective of this lesson is to known the basic properties of pipelining, classification of pipeline processors and the required memory support. The main aim this lesson is to learn the how pipelining is implemented in various computer architecture like RISC and CISC etc. How the issues related to limitations of pipelining and are overcame by using superscalar pipeline architecture.

3.1 Introduction

Pipeline is similar to the assembly line in industrial plant. To achieve pipelining one must divide the input process into a sequence of sub tasks and each of which can be executed concurrently with other stages. The various classification or pipeline line processor are arithmetic pipelining, instruction pipelining, processor pipelining have also been briefly discussed. Limitations of pipelining are discussed and shift to Pipeline architecture to

Superscalar architecture is also discussed. Superscalar pipeline organization and design are discussed.

3.2 Linear pipelining

Pipelining is a technique of that decompose 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 3.1, 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

Figure 3.1 Space –time diagram for pipeline

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 t_p be the one clock cycle time.

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

$= k*t_p + (n-1)*t_p = (k+n-1)*t_p$

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

$= n*t_n$

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

$= n*t_n / (n+k-1)*t_p$

if n is very large compared to k than

$= t_n / t_p$

if a process takes same time in both case with pipeline and non pipeline configuration than $t_n = k*t_p$

Thus speed up ratio will $S_k = k*t_p/t_p = k$

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 $n \rightarrow \infty$, $\eta \rightarrow 1$.

$\eta = S_k/k$

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

$W= n/[k*t + (n-1) *t] = \eta/t$

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 3.1. 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 3.2. 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. RISC architecture supports pipelining more than a CISC architecture does. There are three prime disadvantages of pipeline architecture.

1. The first is complexity i.e., to divide the process into dependent subtask
2. Many intermediate registers are required to hold the intermediate information as output of one stage which will be input of next stage. These are not required for single unit circuit thus it is usually constructed entirely as combinational circuit
3. The third disadvantage is its inability to continuously run the pipeline at full speed, i.e. the pipeline stalls for some cycle. There are phenomena called pipeline hazards which disrupt the smooth execution of the pipeline if these hazards are not handled properly they may gave wrong result. Often it is required insert delays in the pipeline flow in order to manage these hazards such delays are called bubbles. Often it is managed by using special hardware techniques while sometime using software techniques such as compiler or code reordering, etc. Various types of pipeline hazards include:
   - structural hazards that happens due to hardware conflicts
   - data hazards that happen due to data dependencies
   - control hazards that happens when there is change in flow of statement like due to branch, jump, or any other control flow changes conditions
   - Exception hazard that happens due to some exception or interrupt occurred while execution in a pipeline system.

3.3 Non linear pipeline

A dynamic pipeline can be reconfigured to perform variable function at different times. The traditional linear pipelines are static pipeline because they used to perform

fixed function. 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.

3.3.1 Reservation Tables and latency analysis

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. (For example, think of X as being a floating square root, and Y as being a floating cosine. A simple floating multiply might occupy just S1 and S2 in sequence.) We could also denote multiple stages being used in parallel, or a stage being drawn out for more than one cycle with these diagrams.

S1	X					X		X
S2		X		X				
S3			X		X		X	

S1	Y				Y	
S2			Y			
S3		Y		Y		Y

We determine the next start time for one or the other of the functions by lining up the diagrams and sliding one with respect to another to see where one can fit into the open slots. Once an X function has been scheduled, another X function can start after 1, 3 or 6 cycles. A Y function can start after 2 or 4 cycles. Once a Y function has been scheduled, another Y function can start after 1, 3 or 5 cycles. An X function can start after 2 or 4 cycles. After two functions have been scheduled, no more can be started until both are complete.

Job Sequencing and Collision Prevention

Initiation the start a single function evaluation collision may occur as two or more initiations attempt to use the same stage at the same time. Thus it is required to properly schedule queued tasks awaiting initiation in order to avoid collisions and to achieve high throughput. We can define collision as:

1. A collision occurs when two tasks are initiated with latency (initiation interval) equal to the column distance between two "X" on some row of the reservation table.
2. The set of column distances F ={l1,l2,…,lr} between all possible pairs of "X" on each row of the reservation table is called the forbidden set of latencies.
3. The collision vector is a binary vector C = (Cn…C2 C1), Where Ci=1 if i belongs to F (set of forbidden latencies) and Ci=0 otherwise.

Some fundamental concepts used in it are**:**

Latency - number of time units between two initiations (any positive integer 1, 2,…)

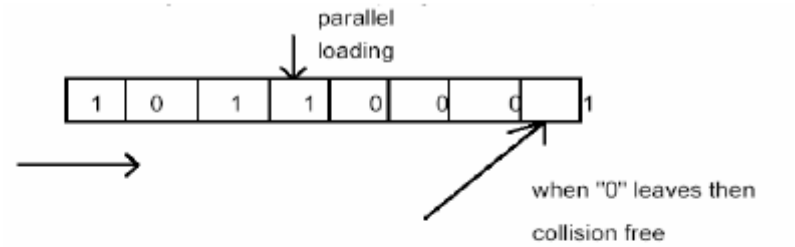
Latency sequence – sequence of latencies between successive initiations

Latency cycle – a latency sequence that repeats itself

Control strategy – the procedure to choose a latency sequence

Greedy strategy – a control strategy that always minimizes the latency between the current initiation and the very last initiation

Example: Let us consider a Reservation Table with the following set of forbidden latencies F and permitted latencies P (complementation of F).



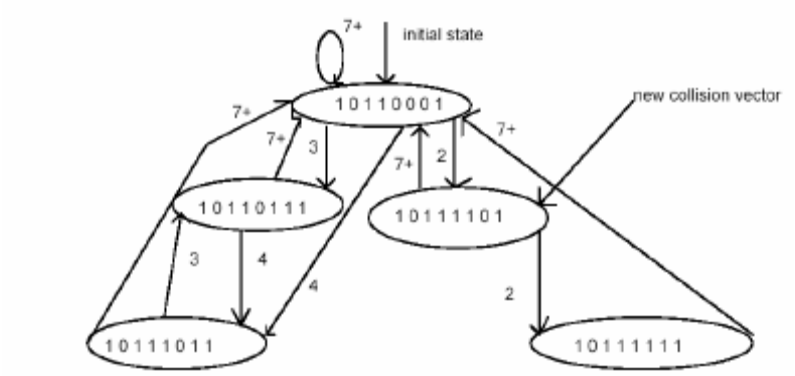
Forbidden list = F = {1,5,6,8}

Collision vector: C={1 0 1 1 0 0 0 1)

8 7 6 5 4 3 2 1

	0	1	2	3	4	5	6	7	8
1	X								X
2		X	X					X	
3				X					
4					X	X			
5							X	X	

1 0 1 1 0 0 0 1
1 0 1 1 0 1 1 1
OR 1 0 1 1
1 0 1 1 1 0 1 1



1 0 1 1 0 0 0 1 initial
(+)0 0 1 0 1 1 0 0 →2 (initial)
1 0 1 1 1 1 0 1

1 0 1 1 0 0 0 1 initial
0 0 0 1 0 1 1 0 →2 (initial)
1 0 1 1 0 1 1 1

1 0 1 1 0 0 0 1 initial
0 0 0 0 1 0 1 1 →4 (initial)

1 0 1 1 0 0 0 1 initial
0 0 0 0 0 0 0 1 →7 (initial)
1 0 1 1 0 0 0 1

It has been observed that

1. The collision vector shows both permitted and forbidden latencies from the same reservation table.

2. One can use n-bit shift register to hold the collision vector for implementing a control strategy for successive task initiations in the pipeline. Upon initiation of the first task, the collision vector is parallel-loaded into the shift register as the initial state. The shift register is then shifted right one bit at a time, entering 0's from the left end. A collision free initiation is allowed at time instant t+k a bit 0 is being shifted at of the register after k shifts from time t.

A **state diagram** is used to characterize the successive initiations of tasks in the pipeline in order to find the shortest latency sequence to optimize the control strategy. A **state** on the diagram is represented by the contents of the shift register after the proper number of shifts is made, which is equal to the latency between the current and next task initiations.

3. The successive collision vectors are used to prevent future task collisions with previously initiated tasks, while the collision vector C is used to prevent possible collisions with the current task. If a collision vector has a "1" in the ith bit (from the right), at time t, then the task sequence should avoid the initiation of a task at time t+i.

4. Closed logs or cycles in the state diagram indicate the steady – state sustainable latency sequence of task initiations without collisions. The **average latency** of a cycle is the sum of its latencies (period) divided by the number of states in the cycle.

5. The throughput of a pipeline is inversely proportional to the reciprocal of the average latency. A latency sequence is called **permissible** if no collisions exist in the successive initiations governed by the given latency sequence.

6. The maximum throughput is achieved by an optimal scheduling strategy that achieves the (MAL) minimum average latency without collisions.

Simple cycles are those latency cycles in which each state appears only once per each iteration of the cycle. A single cycle is a **greedy cycle** if each latency contained in the cycle is the minimal latency (outgoing arc) from a state in the cycle. A good task-initiation sequence should include the greedy cycle.

Procedure to determine the greedy cycles

1. From each of the state diagram, one chooses the arc with the smallest latency label unit; a closed simple cycle can formed.

2. The average latency of any greedy cycle is no greater than the number of latencies in the forbidden set, which equals the number of 1's in the initial collision vector.

3. The average latency of any greedy cycle is always lower-bounded by the

MAL in the collision vector

Two methods for improving dynamic pipeline throughput have been proposed by Davidson and Patel these are

- The reservation of a pipeline can be modified with insertion of non complete delays
- Use of internal buffer at each stage.

Thus high throughput can be achieved by using the modified reservation table yielding a more desirable latency pattern such the each stage is maximum utilized. Any computation can be delayed by inserting a non compute stage.

Reconfigurable pipelines with different function types are more desirable. This requires an extensive resource sharing among different functions. To achieve this one need a more complicated structure of pipeline segments and their interconnection controls like bypass techniques to avoid unwanted stage.

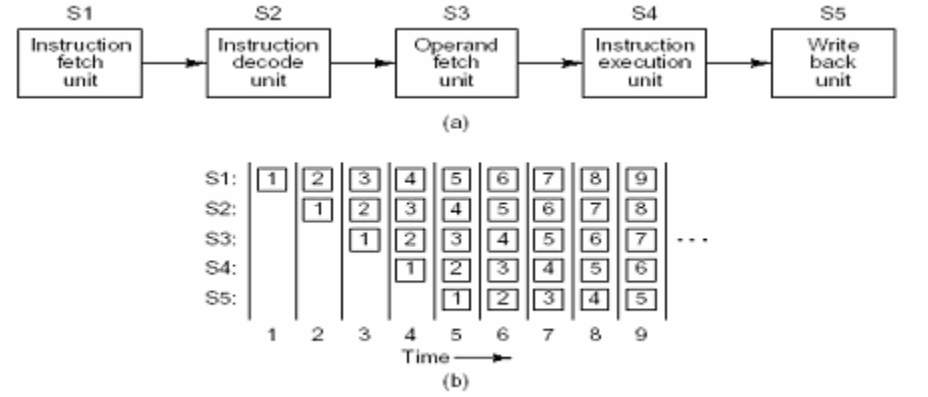
A dynamic pipeline would allow several configurations to be simultaneously present like arithmetic unit performing both addition as well as multiplication at same time. But to achieve this tremendous control overhead and increased interconnection complexity would be expected.

3.4 Design of Instruction pipeline

As we know that in general case, the each instruction to execute in computer undergo following steps:

- Fetch the instruction from the memory.
- Decode the instruction.
- Calculate the effective address.
- Fetch the operands from the memory.
- Execute the instruction (EX).
- Store the result back into memory (WB).

For sake of simplicity we take calculation of the effective address and fetch operand from memory as single segment as operand fetch unit. Thus below figure shows how the instruction cycle in CPU can be processed with five segment instruction pipeline.



While the instruction is decoded (ID) in segment 2 the new instruction is fetched (IF) from segment 1. Similarly in third time cycle when first instruction effective operand is fetch (OF), the 2nd instruction is decoded and the 3rd instruction is fetched. In same manner in fourth clock cycle, and subsequent cycles all subsequent instructions can be fetched and placed in instruction FIFO. Thus up to five different instructions can be processed at the same time. The figure show how the instruction pipeline works, where time is in the horizontal axis and divided into steps of equal duration. Although the major difficulty with instruction pipeline is that different segment may take different time to operate the forth coming information. For example if operand is in register mode require much less time as compared if operand has to be fetched from memory that to with indirect addressing modes. The design of an instruction pipeline will be most effective if the instruction cycle is divided into segments of equal duration. As there can be resource conflict, data dependency, branching, interrupts and other reasons due to pipelining can branch out of normal sequence.

Que 5.3 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.

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.

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)

Cab be 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

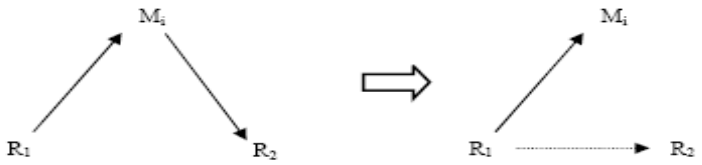
R1 -> (Mi) (fetch)

R2 -> (Mi) (Fetch)

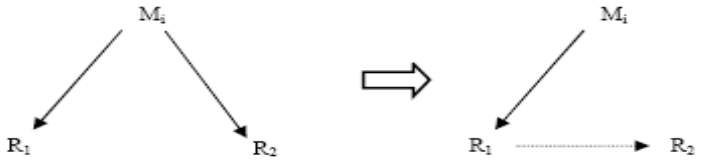
Is being replaced by Only by one memory access

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 by one memory access

Mi -> (R2) (store)

The above steps shows how to apply internal forwarding to simplify a sequence of arithmetic and memory access operations in figure thick arrows for memory accesses and dotted arrows for register transfers

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.

Out of order execution introduces another level of complexity in the control of the pipeline, because it is desirable to preserve the abstraction of in-order issue, even in the presence of exceptions that could flush the pipe at any stage. But we'll defer this to later.

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.

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 achieved either by using scoreboard or with Tomasulo's register tagging algorithm and discussed in superscalar processors

3.5 Arithmetic pipeline

Pipeline arithmetic is used in very high speed computers specially involved in scientific computations a basic principle behind vector processor and array processor. They are used to implement floating – point operations, multiplication of fixed – point numbers and similar computations encountered in computation problems. These computation problems can easily decomposed in suboperations. Arithmetic pipelining is well implemented in the systems involved with repeated calculations such as calculations involved with matrices and vectors. Let us consider a simple vector calculation like

A[i] + b[i] * c[i] for I = 1,2,3,……,8

The above operation can be subdivided into three segment pipeline such each segment has some registers and combinational circuits. Segment 1 load contents of b[i] and c[i] in register R1 and R2 , segment 2 load a[i] content to R3 and multiply content of R1, R2 and store them R4 finally segment 3 add content of R3 and R4 and store in R5 as shown in figure below.

Clock pulse number	Segment 1		Segment 2		Segment 3
	R1	R2	R3	R4	R5
1	B1	C1	-	-	-
2	B2	C2	B1*C1	A1	
3	B3	C3	B2*C2	A2	A1+ B1*C1
4	B4	C4	B3*C3	A3	A2+ B2*C2
5	B5	C5	B4*C4	A4	A3+ B3*C3

6	B6	C6	B5*C5	A5	A4+ B4*C4
7	B7	C7	B6*C6	A6	A5+ B5*C5
8	B8	C8	B7*C7	A7	A6+ B6*C6
9			B8*C8	A8	A7+ B7*C7
10					A8+ B8*C8

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

A = a x 2p

B = b x 2q

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

C = A+ B = c x 2r = d x 2s

Operations performed in the four pipeline stages are specified.

1. Compare the 2 exponents p and q to reveal the larger exponent r =max(p,q) and to determine their difference t =p-q

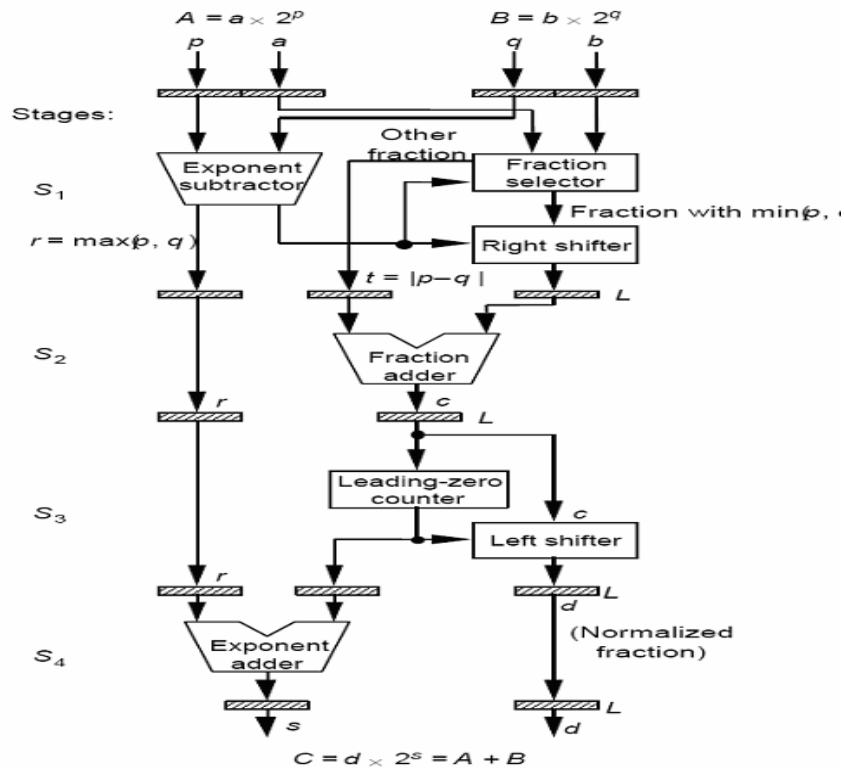
2. Shift right the fraction associated with the smaller exponent by t bits to equalize the two components before fraction addition.

3. Add the preshifted fraction with the other fraction to produce the intermediate sum fraction c where 0 <= c <1.

4. Count the number of leading zeroes, say u, in fraction c and shift left c by u bits to produce the normalized fraction sum d = c x 2u, with a leading bit 1. Update the large exponent s by subtracting s= r – u to produce the output exponent.

The given below is figure show how pipeline can be implemented in floating point addition and subtraction. Segment 1 compare the two exponents this is done using subtraction. Segment2 we chose the larger exponents the one larger exponent as exponent of result also it align the other mantissa by viewing the difference between two and smaller number mantissa should be shifted to right by difference amount. Segment 3 performs addition or subtraction of mantissa while segment 4 normalize the result for that it adjust exponent care must be taken in case of overflow, where we had to shift the mantissa right and increment exponent by one and for underflow the leading zeros of

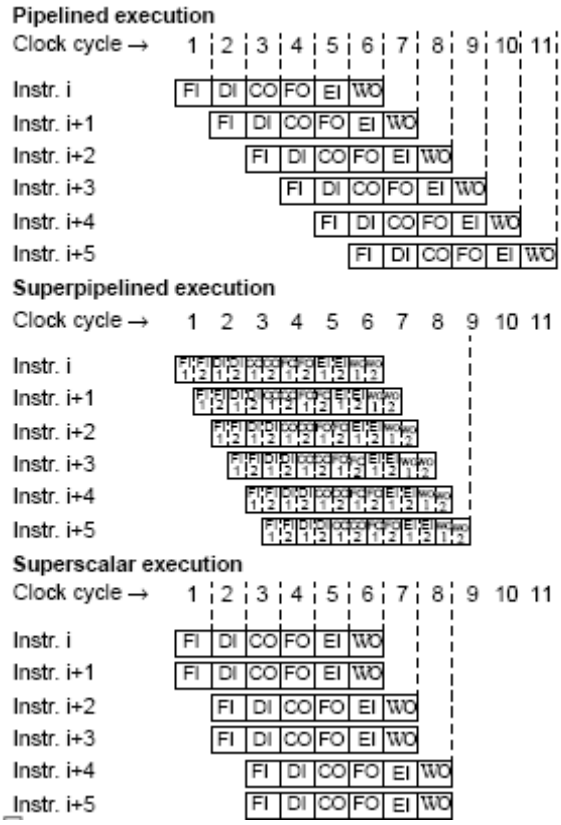
mantissa determines the left shift in mantissa and same number should be subtracted for exponent. Various registers R are used to hold intermediate results.



In order to implement pipelined adder we need extra circuitry but its cost is compensated if we have implement it for large number of floating point numbers. Operations at each stage can be done on different pairs of inputs, e.g. one stage can be comparing the exponents in one pair of operands at the same time another stage is adding the mantissas of a different pair of operands.

3.6 S**uperpipeline 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.



3.6.1 Superpipeline

In order to make processors even faster, various methods of optimizing pipelines have been devised. Superpipelining refers to dividing the pipeline into more steps. The more pipe stages there are, the faster the pipeline is because each stage is then shorter. thus Superpipelining increases 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 Ideally, a pipeline with five stages should be five times faster than a non-pipelined processor (or rather, a pipeline with one stage). The instructions are executed at the speed at which each stage is completed, and each stage takes one fifth of the amount of time that the non-pipelined instruction takes. Thus, a processor with an 8-step pipeline (the MIPS R4000) will be even faster than its 5-step counterpart. The MIPS R4000 chops its pipeline into more pieces by dividing some steps

into two. Instruction fetching, for example, is now done in two stages rather than one. The stages are as shown:

Instruction Fetch (First Half)

Instruction Fetch (Second Half)

Register Fetch

Instruction Execute

Data Cache Access (First Half)

Data Cache Access (Second Half)

Tag Check

Write Back

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.

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.

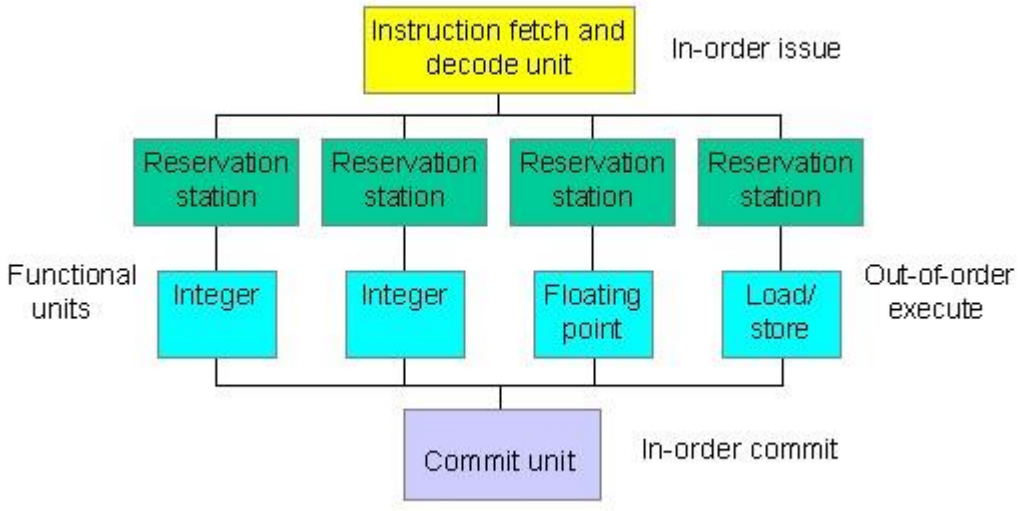
The MIPS R4000 is sometimes called a superpipelined machine, although its 8 stages really only split the I-fetch and D-fetch stages of the pipe and add a Tag Check stage. Nonetheless, the extra stages enable it to operate with higher throughput. The UltraSPARC's 9-stage pipe definitely qualifies it as 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.

3.6.2 Superscalar

A solution to further improve speed is the superscalar architecture. Superscalar pipelining involves multiple pipelines in parallel. Internal components of the processor are replicated so it can launch multiple instructions in some or all of its pipeline stages. The RISC System/6000 has a forked pipeline with different paths for floating-point and integer instructions. If there is a mixture of both types in a program, the processor can keep both forks running simultaneously. Both types of instructions share two initial stages (Instruction Fetch and Instruction Dispatch) before they fork. Often, however, superscalar pipelining refers to multiple copies of all pipeline stages (In terms of laundry, this would mean four washers, four dryers, and four people who fold clothes). Many of today's machines attempt to find two to six instructions that it can execute in every pipeline stage. If some of the instructions are dependent, however, only the first instruction or instructions are issued.

Dynamic pipelines have the capability to schedule around stalls. A dynamic pipeline is divided into three units: the instruction fetch and decode unit, five to ten execute or functional units, and a commit unit. Each execute unit has reservation stations, which act as buffers and hold the operands and operations.



While the functional units have the freedom to execute out of order, the instruction fetch/decode and commit units must operate in-order to maintain simple pipeline behavior. When the instruction is executed and the result is calculated, the commit unit decides when it is safe to store the result. If a stall occurs, the processor can schedule other instructions to be executed until the stall is resolved. This, coupled with the efficiency of multiple units executing instructions simultaneously, makes a dynamic pipeline an attractive alternative

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.

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).

3.7 RISC Pipelines

An efficient way to use instruction pipeline is one of characteristic feature of RISC architecture. A RISC processor pipeline operates in much the same way, although the stages in the pipeline are different. As discussed earlier, the length of the pipeline is dependent on the length of the longest step. Because RISC instructions are simpler than those used in pre-RISC processors (now called CISC, or Complex Instruction Set Computer), they are more conducive to pipelining. While CISC instructions varied in length, RISC instructions are all the same length and can be fetched in a single operation.

Ideally, each of the stages in a RISC processor pipeline should take 1 clock cycle so that the processor finishes an instruction each clock cycle and averages one cycle per instruction (CPI). Hence RISC can achieve pipeline segments, just requiring just one clock cycle, while CISC may use many segments in its pipeline, with longest segment requiring two or more clock cycles.

As most RISC data manipulation operations have register to register operations and an instruction cycle has following two phase.

1. I : Instruction fetch
2. E : Execute . Performs an ALU operation with register input and output.

The data transfer instructions in RISC are limited to Load and Store. These instructions use register indirect addressing and require three stages in pipeline

I : Instruction fetch

E: Calculate memory address

D: Memory. Register to memory or memory to register operation.

To prevent conflicts between memory access to fetch an instruction and to load or store operand, most RISC machine use two separate buses with two memories: one or storing the instruction and other for storing data.

Another feature of RISC over CSIC as far as pipelining is considered is compiler support. Instead of designing hardware to handle the data dependencies and branch penalties, RISC relies on efficiency of compiler to detect and minimize the delay encountered with these problems.

A RISC processor pipeline operates in much the same way, although the stages in the pipeline are different. While different processors have different numbers of steps, lets us consider a three segment Instruction pipeline

I: Instruction fetch

A: ALU operation

E: Execute instruction

The I segment fetches the instruction from memory and decode it. The ALU is used for three different functions, it can be data manipulation , effective address calculation for LOAD and STORE operations, or calculation of the branch address for a program control instruction depending on type of instruction. The E segment directs the output of the

ALU to one of three destination i.e., a destination register or effective address to a data memory for loading or storing or the branch address to program counter, depending upon decode instruction.

Multiplying Two Numbers in Memory

Lets consider an example of matrix multiplication here the main memory is divided into locations numbered from (row) 1: (column) 1 to (row) 6: (column) 4. The execution unit is responsible for carrying out all computations. However, the execution unit can only operate on data that has been loaded into one of the six registers (A, B, C, D, E, or F). Let's say we want to find the product of two numbers - one stored in location 2:3 and another stored in location 5:2 - and then store the product back in the location 2:3.

The CISC Approach

The primary goal of CISC architecture is to complete a task in as few lines of assembly as possible. This is achieved by building processor hardware that is capable of understanding and executing a series of operations. For this particular task, a CISC processor would come prepared with a specific instruction (we'll call it "MULT"). When executed, this instruction loads the two values into separate registers, multiplies the operands in the execution unit, and then stores the product in the appropriate register. Thus, the entire task of multiplying two numbers can be completed with one instruction:

```
MULT 2:3, 5:2
```

MULT is what is known as a "complex instruction." It operates directly on the computer's memory banks and does not require the programmer to explicitly call any loading or storing functions. It closely resembles a command in a higher level language. For instance, if we let "a" represent the value of 2:3 and "b" represent the value of 5:2, then this command is identical to the C statement "a = a * b."

One of the primary advantages of this system is that the compiler has to do very little work to translate a high-level language statement into assembly. Because the length of the code is relatively short, very little RAM is required to store instructions. The emphasis is put on building complex instructions directly into the hardware.

The RISC Approach

RISC processors only use simple instructions that can be executed within one clock cycle. Thus, the "MULT" command described above could be divided into three separate

commands: "LOAD," which moves data from the memory bank to a register, "PROD," which finds the product of two operands located within the registers, and "STORE," which moves data from a register to the memory banks. In order to perform the exact series of steps described in the CISC approach, a programmer would need to code four lines of assembly:

```
LOAD A, 2:3
LOAD B, 5:2
PROD A, B
STORE 2:3, A
```

At first, this may seem like a much less efficient way of completing the operation. Because there are more lines of code, more RAM is needed to store the assembly level instructions. The compiler must also perform more work to convert a high-level language statement into code of this form.

CRICS : CISC and RISC Convergence

State of the art processor technology has changed significantly since RISC chips were first introduced in the early '80s. Because a number of advancements (including the ones described on this page) are used by both RISC *and* CISC processors, the lines between the two architectures have begun to blur. In fact, the two architectures almost seem to have adopted the strategies of the other. Because processor speeds have increased, CISC chips are now able to execute more than one instruction within a single clock. This also allows CISC chips to make use of pipelining. With other technological improvements, it is now possible to fit many more transistors on a single chip. This gives RISC processors enough space to incorporate more complicated, CISC-like commands. RISC chips also make use of more complicated hardware, making use of extra function units for superscalar execution. All of these factors have led some groups to argue that we are now in a "post-RISC" era, in which the two styles have become so similar that distinguishing between them is no longer relevant. This era is often called complex reduced instruction set CRISC.

3.8 VLIW Machines

Very Long Instruction Word machines typically have many more functional units that superscalars (and thus the need for longer – 256 to 1024 bits – instructions to provide control for them) usually hundreds of bits long. These machines mostly use

microprogrammed control units with relatively slow clock rates because of the need to use ROM to hold the microcode. Each instruction word essentially carries multiple "short instructions." Each of the "short instructions" are effectively issued at the same time. (This is related to the long words frequently used in microcode.) Compilers for VLIW architectures should optimally try to predict branch outcomes to properly group instructions.

Pipelining in VLIW Processors

Decoding of instructions is easier in VLIW than in superscalars, because each "region" of an instruction word is usually limited as to the type of instruction it can contain. Code density in VLIW is less than in superscalars, because if a "region" of a VLIW word isn't needed in a particular instruction, it must still exist (to be filled with a "no op").

Superscalars can be compatible with scalar processors; this is difficult with VLIW parallel and non-parallel architectures. "Random" parallelism among scalar operations is exploited in VLIW, instead of regular parallelism in a vector or SIMD machine.

The efficiency of the machine is entirely dictated by the success, or "goodness," of the compiler in planning the operations to be placed in the same instruction words.

Different implementations of the same VLIW architecture may not be binary-compatible with each other, resulting in different latencies.

5.7 Summary

1. The job-sequencing problem is equivalent to finding a permissible latency cycle with the MAL in the state diagram.

2. The minimum number of X's in array single row of the reservation table is a lower bound of the MAL.

Pipelining allows several instructions to be executed at the same time, but they have to be in different pipeline stages at a given moment. Superscalar architectures include all features of pipelining but, in addition, there can be several instructions executing simultaneously in the same pipeline stage. They have the ability to initiate multiple instructions during the same clock cycle. There are two typical approaches today, in order to improve performance:

1. Superpipelining
2. Superscalar

VLIW reduces the effort required to detect parallelism using hardware or software techniques.

The main advantage of VLIW architecture is its simplicity in hardware structure and instruction set. Unfortunately, VLIW does require careful analysis of code in order to "compact" the most appropriate "short" instructions into a VLIW word.

3.9 Keywords

pipelining Overlapping the execution of two or more operations. Pipelining is used within processors by *prefetching* instructions on the assumption that no branches are going to preempt their execution; in *vector processors*, in which application of a single operation to the elements of a vector or vectors may be pipelined to decrease the time needed to complete the aggregate operation; and in *multiprocessors* and *multicomputers*, in which a process may send a request for values before it reaches the computation that requires them..

scoreboard A hardware device that maintains the state of machine resources to enable instructions to execute without conflict at the earliest opportunity.

instruction pipelining strategy of allowing more than one instruction to be in some stage of execution at the same time.

3.10 Self assessment questions

1. Explain an asynchronous pipeline model, a synchronous pipeline model and reservation table of a four-stage linear pipeline with appropriate diagrams.
2. Define the following terms with regard to clocking and timing control.

a) Clock cycle and throughput b) Clock skewing c) Speedup factor

3. Describe the speedup factors and the optimal number of pipeline stages for a linear pipeline unit.
4. Explain the features of non-linear pipeline processors with feedforward and feedbackward connections.
5. Explain the pipelined execution of the following instructions with the following instructions:

a) X = Y + Z b) A = B X C

6. What are the possible hazards that can occur between read and write operations in an instruction pipeline?

3.11 References/Suggested readings

Advance Computer architecture: Kai Hwang

Lesson: Cache memory Organization

Lesson No. : 04

4.0 Objective

In this lesson we will discuss about bus that is used for interconnections between different processor. We will discuss about use of cache memory in multiprocessor environment and various addressing scheme used for cache memory. The page replacement policy and performance of cache is also measured. Also we will discuss how shared memory concept is used in multiprocessor. Various issues regarding event ordering specially in case of memory events that deal with shared memory creates

synchronization problem we will also discuss various models designed to overcome these issues.

4.1 Introduction

In the hierarchy memory cache memory are the fastest memory that lies between registers and RAM . It holds recently used data and/or instructions and has a size varying from few kB to several MB.

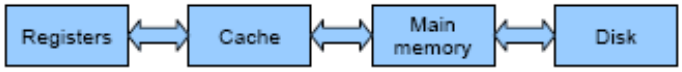


Figure 4.1 Memory structure for a processor

The figure 4.1 shows a cache and main memory structure. A cache consists of C slots and each *slot* in the cache can hold K memory words. Here the main memory with 2^n-1 words i.e., M words with each having a unique n-bit address and cache memory having C*K words where K is the Block size and C are the number of lines. Each word that resides in the cache is a subset of main memory. Since there are more blocks in main memory than number of lines in cache, an individual line cannot be uniquely and permanently dedicated to a particular block. Therefore, each line includes a tag that identifies which particular block of main memory is currently occupying that line of cache. The tag is usually a portion of the main memory address. The cache memory is accessed but by pattern matching on a *tag* stored in the cache.

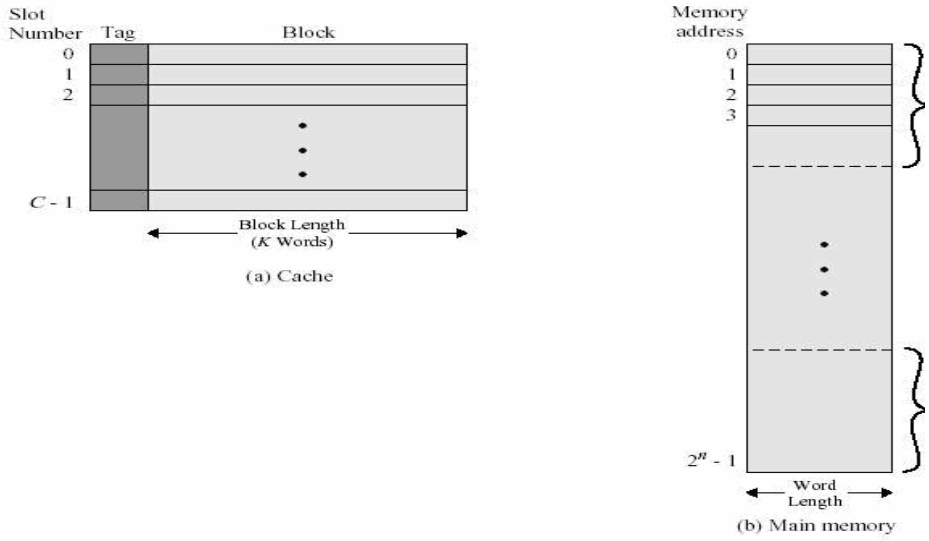


Figure 4.2 Cache / Main memory structure

For the comparison of address generated by CPU the memory controller use some algorithm which determines whether the value currently being addressed in memory is available in the cache. The transformation data from main memory to cache memory is referred as a mapping process. Let us derive an address translation scheme using cache as a linear array of entries, each entry having the following structure as shown in figure 4.3.

A Cache Storage is divided into three fields:

Data - The block of data from memory that is stored in a specific line in the cache

Tag - A small field of length K bits, used for comparison, to check the correct address of data

Valid Bit - A one-bit field that indicates status of data written into the cache.

The N-bit address is produced by the processor to access cache data is divided into three fields:

Tag - A K-bit field that corresponds to the K-bit tag field in each cache entry,

Index - An M-bit field in the middle of the address that points to one cache entry

Byte Offset – L Bits that finds particular data in a line if valid cache is found.

It follows that the length of the virtual address is given by N = K + M + L bits.

Cache Address Translation. As shown in Figure 4.3, we assume that the cache address has length 32 bits. Here, bits 12-31 are occupied by the Tag field, bits 2-11 contain the Index field, and bits 0,1 contain the Offset information. The index points to the line in cache that supposedly contains the data requested by the processor. After the cache line is retrieved, the Tag field in the cache line is compared with the Tag field in the cache address. If the tags do not match, then a cache miss is detected and the comparator outputs a zero value. Otherwise, the comparator outputs a one, which is *and*-ed with the valid bit in the cache row pointed to by the Index field of the cache address. If the valid bit is a one, then the Hit signal output from the *and* gate is a one, and the data in the cached block is sent to the processor. Otherwise a cache miss is registered.

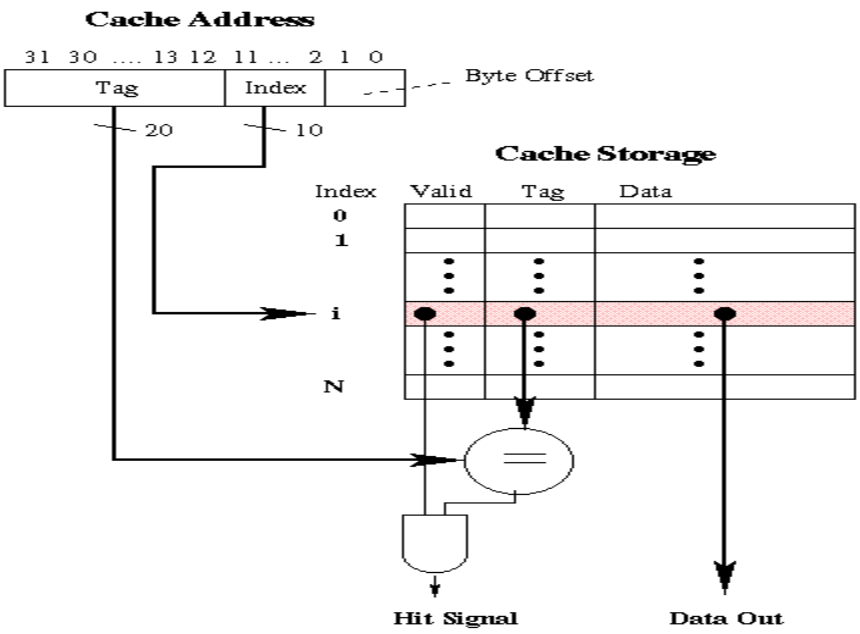


Figure 4.3. Schematic diagram of cache

A cache implements several different policies for retrieving and storing information, one in each of the following categories:

° Fetch policy—determines when information is loaded into the cache.

° Replacement policy—determines what information is purged when space is needed for a new entry.

° Write policy—determines how soon information in the cache is written to lower levels in the memory hierarchy.

4.2 **Cache addressing models**

Most multiprocessor system use private cache associated with different processor.

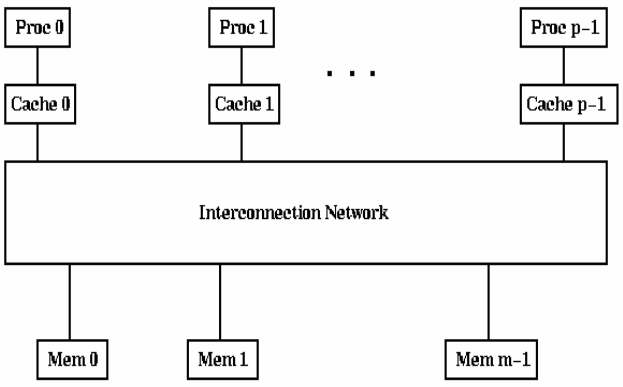


Figure 4.4 A memory hierarchy for a shared memory multiprocessor.

Cache can be addressed either by physical address or virtual address.

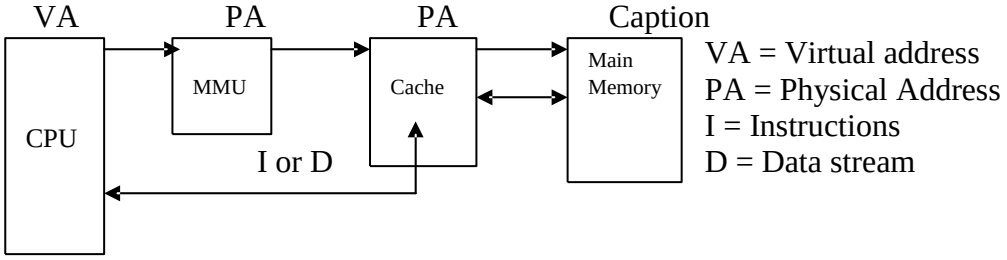
Physical address cache: when cache is addressed by physical address it is called physical address cache. The cache is indexed and tagged with physical address. Cache lookup must occur after address translation in TLB or MMU. No aliasing is allowed so that the address is always uniquely translated without confusion. This provides an advantage that we need no cache flushing, no aliasing problem and fewer cache bugs in OS kernel. The short coming is the slowdown in accessing the cache until the MMU/TLB finishes translating the address.

Advantage of physically addressed caches:

- no cache flushing on a context switch
- no synonym problem (several different virtual addresses can span the same physical addresses : a much better hit ratio between processes)

Disadvantage of physically addressed caches:

- do virtual-to-physical address translation on every access
- increase in hit time because must translate the virtual address before access the cache



4.5 (a) A unified cache accessed by physical address

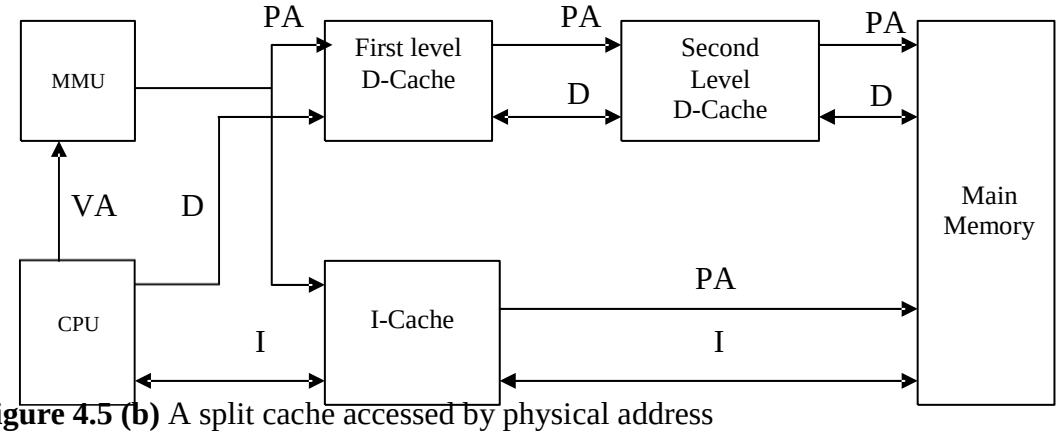


figure 4.5 (b) A split cache accessed by physical address

Virtual Address caches: when a cache is indexed or tagged with virtual address it is called virtual address cache. In this model both cache and MMU translation or validation are done in parallel. The physical address generated by the MMU can be saved in tags for later write back but is not used during the cache lookup operations.

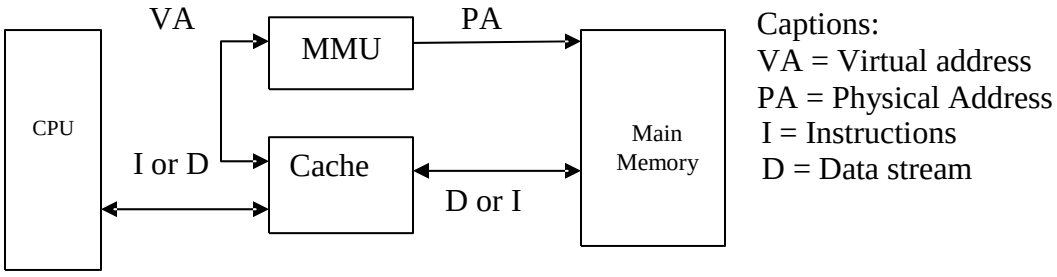
Advantage of virtually-addressed caches

- do address translation only on a cache miss
- faster for hits because no address translation

Disadvantage of virtually-addressed caches

cache flushing on a context switch (example : local data segments will get an erroneous hit for virtual addresses already cached after changing virtual address space, if no cache flushing).

synonym problem (several different virtual addresses cannot span the same physical addresses without being duplicated in cache).



(a) A unified cache accessed by virtual address

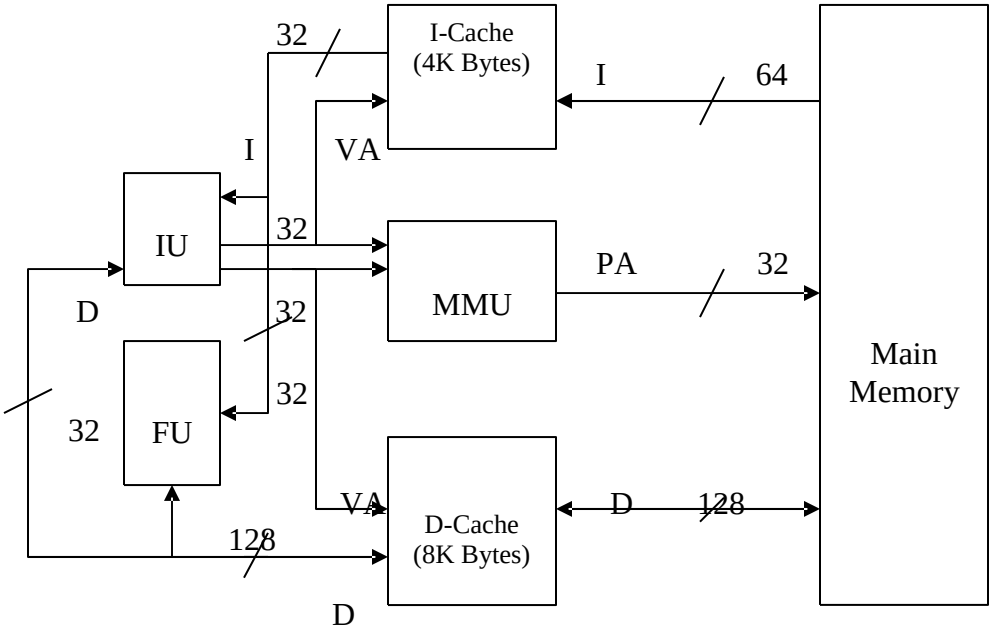


Figure 4.6(b) Virtual address for split cache

Aliasing: The major problem with cache organization in multiprocessor is that multiple virtual addresses can map to a single physical address i.e., different virtual address cache logically addressed data have the same index/tag in the cache. Most processors guarantee that all updates to that single physical address will happen in program order. To deliver on that guarantee, the processor must ensure that only one copy of a physical address resides in the cache at any given time.

4.3 Cache mapping

Caches can be organized according to four different

strategies: ° Direct

° Fully

associative ° Set

associative

° Sectored

4.3.1 Direct-Mapped Caches

The easiest way of organizing a cache memory employs direct mapping that is based on a simple algorithm to map data block i from the main memory into data block j in the cache. There is a one-to-one correspondence between each block of data in the cache and each memory block thus to find a memory block i, then there is one and only one place in the cache where i is stored

If we have 2^n words in main memory and 2^k words in cache memory. In cache memory each word consists of data word and its associated tag. The n-bit memory address is divided into three fields : low order k bits are referred as the index field and used to address a word in the cache. The remaining n-k high-order bits are called the *tag.* The index field is further divided into the *slot* field, which will be used find a particular slot in the cache; and the offset field is used to identify a particular memory word in the slot. When a block is stored in the cache, its *tag* field is stored in the *tag field* of the cache slot.

When CPU generates an address the index field is used to access the cache. The tag field of CPU address is compared with the tag in word read from the cache. If the two tags match, there is a hit and else there is a miss and the required word is read from main memory. Whenever a ``cache miss'' occurs, the cache line will be replaced by a new line

of information from main memory at an address with the same index but with a different tag.

Lets us understand how direct mapping is implemented with following simple example Figure 4.7. The memory is composed of 32 words and accessed by a 5-bit address. Let the address has a 2-bit tag (set) field, a 2-bit slot (line) field and a 1-bit word field. The cache memory holds $2^2 = 4$ lines each having two words. When the processor generates an address, the appropriate line (slot) in the cache is accessed. For example, if the processor generates the 5-bit address 11110_2, line 4 in set 4 is accessed. The memory space is divided into sets and the sets into lines. The Figure 4.7 reveals that there are four possible lines that can occupy cache line 4 lines 4 in set 0, in set 1, in set 2 and set 4. In this example the processor accessed line 4 in set 4. Now “How does the system resolve this issue?"

Figure 4.7 shows how a direct mapped cache resolves the contention between lines. Each line in the cache memory has a tag or label that identifies which set this particular line belongs to. When the processor accesses line 4, the tag belonging to line 4 in the cache is sent to a comparator. At the same time the set field from the processor is also sent to the comparator. If they are the same, the line in the cache is the desired line and a hit occurs. If they are not the same, a miss occurs and the cache must be updated. Figure 4.17 provides a skeleton structure of a direct mapped cache memory system.

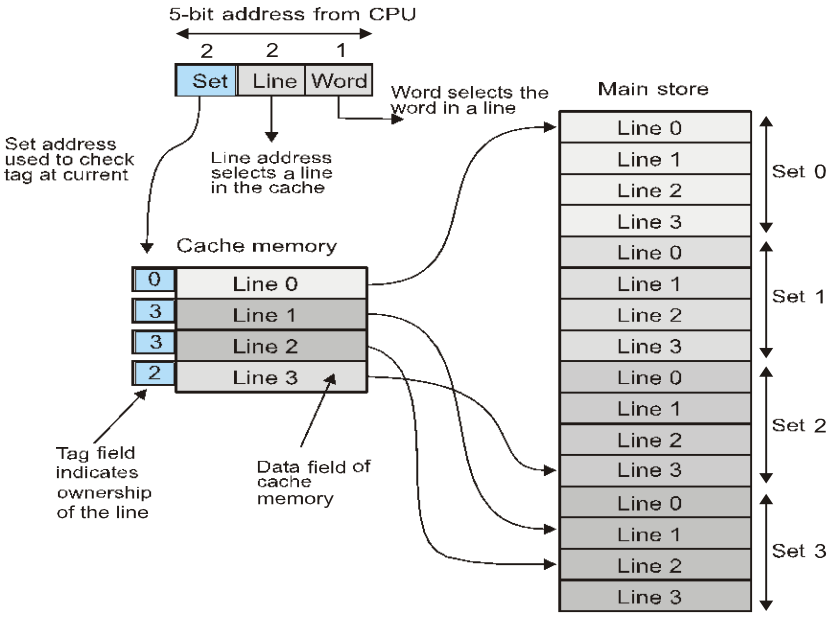


Figure 4.7 Resolving contention between lines in a direct-mapped cache

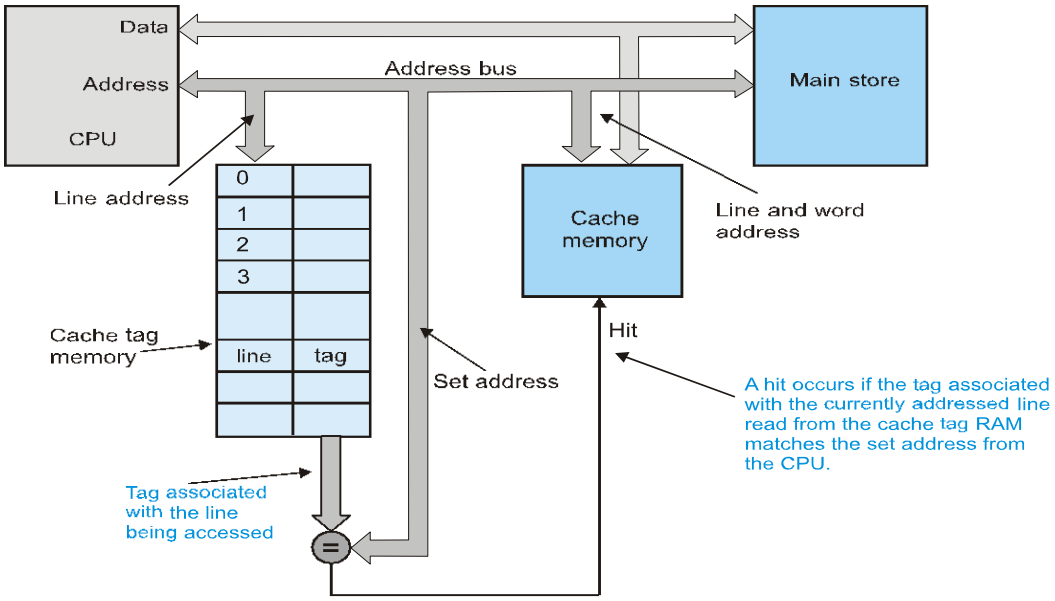


Figure 4.8 Implementation of direct-mapped cache

The advantage of direct mapping are as follows

It's simplicity.

Both the cache memory and the cache tag RAM are widely available devices.

The direct mapped cache requires no complex line replacement algorithm. If line *x* in set *y* is accessed and a miss takes place, line *x* from set *y* in the main store is loaded into the frame for line *x* in the cache memory and the tag set to *y i.e.*,, there is no decision to be taken regarding which line has to be rejected when a new line is to be loaded.

It inherents parallelism. Since the cache memory holding the data and the cache tag RAM are entirely independent, they can both be accessed simultaneously. Once the tag has been matched and a hit has occurred, the data from the cache will also be valid.

The disadvantage of direct mapping are as follows

it is inflexible

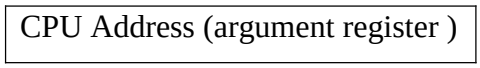
A cache has one restriction *a particular memory address can be mapped into only one cache location also*, all addresses with the same index field are mapped to the same cache location. Consider the following fragment of code:

```
REPEAT
        Get_data
        Compare
UNTIL match OR end_of_data
```

Let the Get data routine and compare routine use two blocks, both these blocks have same index but have different tags are repeated accessed. Consequently, the performance of a direct-mapped cache can be very poor under above circumstances. However, statistical measurements on real programs indicate that the very poor worst-case behavior of direct-mapped caches has no significant impact on their average behavior.

4.3.3 Associative Mapping:

One way of organizing a cache memory which overcomes the limitations of direct mapped cache such that there is no restriction on what data it can contain can be done with associative cache memory. An associative memory is the fastest and most flexible way of cache organization. It stores both the address and the value (data) from main memory in the cache. An associative memory has an *n*-bit input. An address from the processor is divided into three fields: the tag, the line, and the word.The mapping is done with storing tag information in n-bit argument register and comparing it with address tag in each location simultaneously. If the input tag matches a stored tag, the data associated with that location is output. Otherwise the associative memory produces a miss output. Unfortunately, large associative memories are not yet cost-effective. Once the associative cache is full, a new line can be brought in only by overwriting an existing line that requires a suitable line replacement policy. Associative cache memories are efficient because they place no restriction on the data they hold, as permits any location of cache to store any word from main memory.



Address	Data
01101001	10010100
10010001	10101010

Figure 4.9 Associative cache

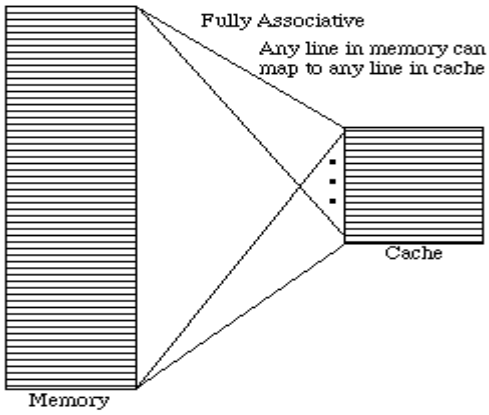


Figure 4.10Associative mapping

All of the comparisons are done simultaneously, so the search is performed very quickly. This type of memory is very expensive, because each memory location must have both a comparator and a storage element. Like the direct mapped cache, the smallest unit of data transferred into and out of the cache is the line. Unlike the direct-mapped cache, there's no relationship between the location of lines in the cache and lines in the main memory.

When the processor generates an address, the word bits select a word location in both the main memory and the cache. The tag resolves which of the lines is actually present. In an associative cache any of the 64K lines in the main store can be located in any of the lines in the cache. Consequently, the associative cache requires a 16-bit tag to identify one of the 2^{16} lines from the main memory. Because the cache's lines are not ordered, the tags are not ordered, it may be anywhere in the cache or it may not be in the cache.

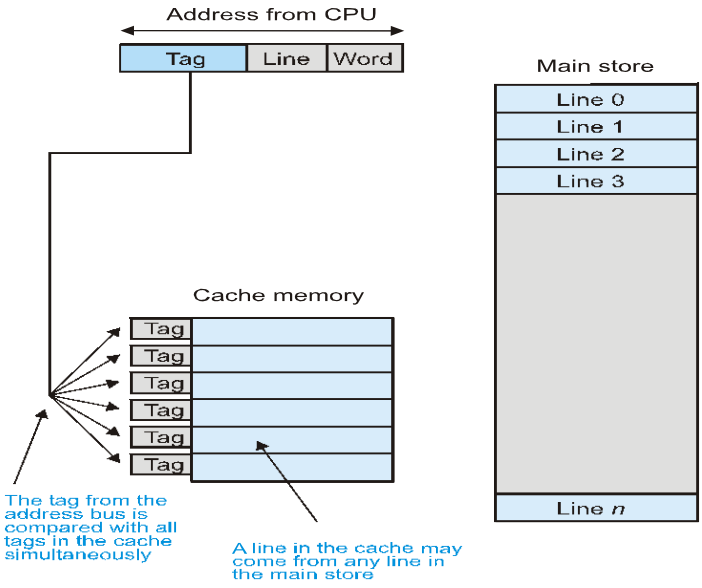


Figure 4.11 Associative-mapped cache

4.3.4 Set associative Mapping:

Most computers use set associative mapping technique as it is a compromise between the direct-mapped cache and the fully associative cache. In a set associative cache memory several direct-mapped caches connected in parallel. Let to find memory block b in the cache, there are n entries in the cache that can contain b we say that this type of cache is called n-*way set associative*. For example, if n = 2, then we have a two-way set associative cache. This is the simplest arrangement and consists of two direct-mapped cache memories. Thus for n parallel sets, a n-way comparison is performed in parallel against all members of the set. Usually $n = 2^k$, for k = 1, 2, 4 are chosen for a set associative cache (k = 0 corresponds to direct mapping). As n is small (typically 2 to 14), the logic required to perform the comparison is not complex. This is a widely used technique in practice (e.g. 80486 uses 4-way, P4 uses 2-way for the instruction cache, 4-way for the data cache).

Figure 4.22 describes the common 4-way set associative cache. When the processor accesses memory, the appropriate line in each of four direct-mapped caches is accessed simultaneously. Since there are four lines, a simple associative match can be used to determine which (if any) of the lines in cache are to supply the data. In figure 4.22 the hit output from each direct-mapped cache is fed to an OR gate which generates a hit if any of the caches generate a hit.

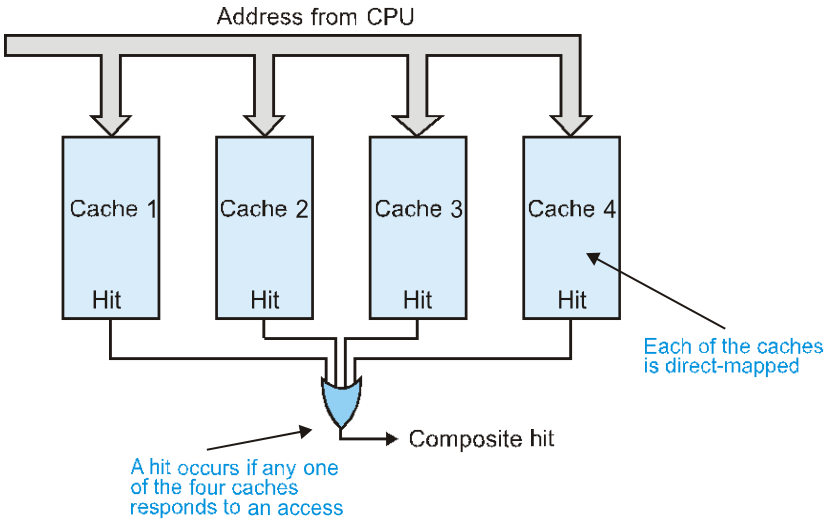


Figure 4.12 Set associative-mapped cache

4.3.4 Sector mapped cache memory

The idea is to partition both the cache and memory into fixed size sectors. Thus in a sectored cache, main memory is partitioned into sectors, each containing several blocks. The cache is partitioned into sector frames, each containing several lines. (The number of lines/sector frame = the number of blocks/sector.) As shown in figure below sector size is of 16 block. Each sector can be mapped to any of the sector frame with full associative at the sector level.

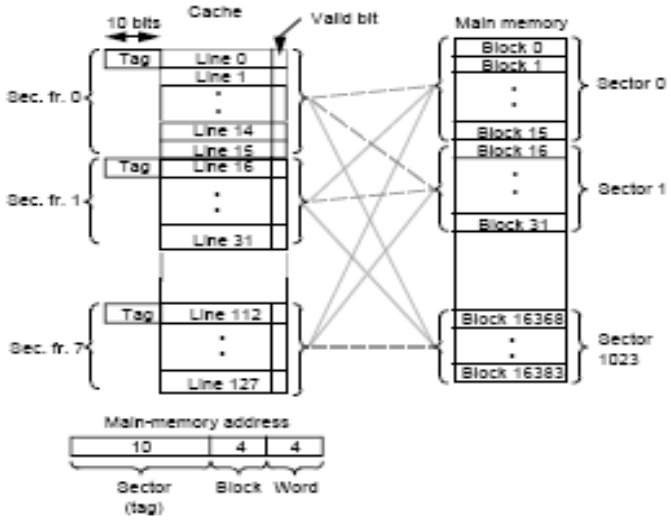


Figure 4.13 Sector mapped memory

Each sector can be placed in any of the available sector frame. The memory requests are destined for blocks not for sectors. This can be filtered out by comparing the sector tag in the memory address with all sector tags using fully associative search.

When block b of a new sector c is brought in,

• it is brought into line b within some sector frame f, and

• the rest of the lines in sector frame f are marked invalid.

Thus, if there are S sector frames, there are S choices of where to place a block.

4.3.5 CACHE performance Issues

As far as the performance of cache is considered the trade off exist among the cache size, set number, block size and memory speed. Important aspect in cache designing with regard to performance are :

a. **the cycle count** : This refers to the number of basic machine cycles needed for cache access, update and coherence control. This count is affected by underlying static or dynamic RAM technology, the cache organization and the cache hit ratios. The write through or write back policy also affect the cycle count. The

cycle count is directly related to the hit ratio, which decreases almost linearly with increasing values of above cache parameters.

b. **Hit ratio:** The processor generates the address of a word to be read and send it to cache controller, if the word is in the cache it generates a Hit signal and also deliver it to the processor. If the data is not found in the cache, then it generates a MISS signal and that data is delivered to the processor from main memory, and simultaneously loaded into the cache. The hit ratio is number of hits divided by total number of CPU references to memory (hits plus misses). When cache size approaches

c. **Effect of Block Size:** With a fixed cache size, cache performance is sensitive to the block size. This block size is determined mainly by the temporal locality in typical program.

d. Effect of set number in set associative number.

4.4 Cache replacement algorithm

When a new block is brought into cache, one of the existing blocks must be replaced. The obvious question arise is which page to be replaced? With direct mapping, the solution is easy as we have not choice. But in other circumstances, we do. The three most commonly used algorithms are *Least Recently Used*, *First in First out* and *Random*.

Random -- The optimal algorithm is called *random replacement*, whereby a location to which a block is to be written in cache is chosen at random from the range of cache indices. The random replacement strategy usually implemented using a random number generator. In a 2-way set associative cache, this can be accomplished with a single modulo 2 random variable obtained, from an internal clock

First in, first out (FIFO) -- here the first value *stored* in the cache is the index position representing value to be replaced. For a 2-way set associative cache, this replacement strategy can be implemented by setting a pointer to the previously loaded word each time a new word is *stored* in the cache; this pointer need only be a single bit.

Least recently used (LRU) -- here the value which was actually used least recently is replaced. In general, it is more likely that the most recently used value will be the one required in the near future. This approach, while not always optimal, is intuitively attractive from the perspective of temporal locality. That is, a given program will likely

not access a page or block that has not been accessed for some time. The LRU replacement algorithm requires that each cache or page table entry have a *timestamp*. This is a combination of date and time that uniquely identifies the entry as having been written at a particular time. Given a timestamp t with each of N entries, LRU merely finds the minimum of the cached timestamps, as

$t_{min} = min\{t_i : i = 1..N\}$.

The cache or page table entry having $t = t_{min}$ is then overwritten with the new entry.

For a 2-way set associative cache, this is readily implemented by setting a special bit called the ``USED" bit for the other word when a value is *accessed* while the corresponding bit for the word which was accessed is reset. The value to be replaced is then the value with the USED bit set. This replacement strategy can be implemented by adding a single USED bit to each cache location. The LRU strategy operates by setting a bit in the other word when a value is *stored* and resetting the corresponding bit for the new word. For an *n*-way set associative cache, this strategy can be implemented by storing a modulo *n* counter with each data word.

4.5 Cache Coherence and Synchronization

4.5.1Cache coherence problem

An important problem that must be addressed in many parallel systems - any system that allows multiple processors to access (potentially) multiple copies of data - is *cache coherence*. The existence of multiple cached copies of data creates the possibility of inconsistency between a cached copy and the shared memory or between cached copies themselves.

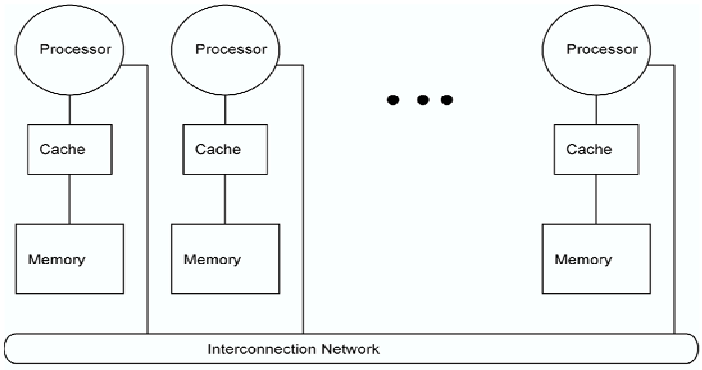


Figure 4.14 cache coherence problem in multiprocessor

There are three common sources of cache inconsistency:

- Inconsistency in data sharing : In a memory hierarchy for a multiprocessor system data inconsistency may occur between adjacent levels or within the same level. The cache inconsistency problem occurs only when multiple private cache are used. Thus it is, the possible that a wrong data being accessed by one processor because another processor has changed it, and not all changes have yet been propagated. Suppose we have two processors, A and B, each of which is dealing with memory word X, and each of which has a cache. If processor A changes X, then the value seen by processor B in *its own cache* will be wrong, *even if processor A also changes the value of X in main memory* (which it - ultimately - should).

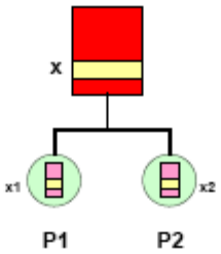


Figure 4.15 Cache coherence problem

In above example initially, x1 = x2 = X = 5.

P1 writes X:=10 using *write-through*.

P2 now reads X and uses its local copy x2, but finds that X is still 5.

Thus P2 does not know that P1 modified X.

Thus the cache inconsistency problem occurs when multiple private cache are used and especially the problem arose by writing the shared variables.

- Process migration(even if jobs are independent): This problem occurs when a process containing shared variable X migrates from process 1 to process2 using the write back cache on the right. Thus another important aspect of coherence is *serialization* of writes - that is, if two processors try to write 'simultaneously', then (i) the writes happen sequentially (and it doesn't really matter who gets to write first - provided we have sensible arbitration); and (ii) *all processors see the writes as occurring in the same order*. That is, if processors A and B both write to X, with A writing first, then any other processors (C, D, E) *all* see the *same* thing.

- DMA I/O – this inconsistency problem occur during the I/O operation that bypass the cache. This problem is present even in a uniprocessor and can be removed by OS cache flushes)

In practice, these issues are managed by a memory bus, which by its very nature ensures write serialization, and also allows us to broadcast invalidation signals (we essentially just put the memory address to be invalidated on the bus). We can add an extra *valid* bit to cache tags to mark then invalid. Typically, we would use a write-back cache, because it has much lower memory bandwidth requirements. Each processor must keep track of which cache blocks are *dirty* - that is, that it has written to - again by adding a bit to the cache tag. If it sees a memory access for a word in a cache block it has marked as dirty, it intervenes and provides the (updated) value. There are numerous other issues to address when considering cache coherence.

One approach to maintaining coherence is to recognize that not every location needs to be shared (and in fact most don't), and simply reserve some space for non-cacheable data such as semaphores, called a coherency domain.

Using a fixed area of memory, however, is very restrictive. Restrictions can be reduced by allowing the MMU to tag segments or pages as non-cacheable. However, that requires the OS, compiler, and programmer to be involved in specifying data that is to be coherently shared. For example, it would be necessary to distinguish between the sharing of semaphores and simple data so that the data can be cached once a processor owns its semaphore, but the semaphore itself should never be cached.

In order to remove this data inconsistency there are a number of approaches based on hardware and software techniques few are given below:

- No caches is used which is not a feasible solution
- Make shared-data non-cacheable this is the simplest software solution but produce low performance if a lot of data is shared
- software flush at strategic times: e.g., after critical sections, this is relatively simple technique but has low performance if synchronization is not frequent
- hardware *cache coherence this can be achieved by making* memory and caches coherent (consistent) with each other, in other words if the memory and other processors see writes then without intervention of the to software

- absolute coherence all copies of each block have same data at all times
- It is not necessary what is required is *appearance of absolute coherence that is done by making* temporary incoherence is OK (e.g., write-back cache)

In general a cache coherence protocols consist of the set of possible states in local caches, the state in shared memory and the state transitions caused by the messages transported through the interconnection network to keep memory coherent. There are basically two kinds of protocols depends on how writes is handled

4.5.2 Snooping Cache Protocol (for bus-based machines);

With a bus interconnection, cache coherence is usually maintained by adopting a "snoopy protocol", where each cache controller "snoops" on the transactions of the other caches and guarantees the validity of the cached data. In a (single-) multi-stage network, however, the unavailability of a system "bus" where transactions are broadcast makes snoopy protocols not useful. Directory based schemes are used in this case.

In case of snooping protocol processors perform some form of *snooping* - that is, keeping track of other processor's memory writes. ALL caches/memories see and react to ALL bus events. The protocol relies on global visibility of requests (ordered broadcast). This allows the processor to make state transitions for its cache-blocks.

Write Invalidate protocol

The states of a cache block copy changes with respect to read, write and replacement operations in the cache. The most common variant of snooping is a *write invalidate protocol*. In the example above, when processor A writes to X, it broadcasts the fact and all other processors with a copy of X in their cache mark it invalid. When another processor (B, say) tries to access X again then there will be a cache miss and either

(i) in the case of a write-through cache the value of X will have been updated (actually, it might not because not enough time may have elapsed for the memory write to complete - but that's another issue); or

(ii) in the case of a write-back cache processor A must spot the read request, and substitute the *correct* value for X.

Figure 6.16 Write back with cache

Figure 6. 17 Write through with cache

An alternative (but less-common) approach is *write broadcast*. This is intuitively a little more obvious - when a cached value is changed, the processor that changed it broadcasts the new value to all other processors. They then update their own cached values. The trouble with this scheme is that it uses up more memory bandwidth. A way to cut this is to observe that many memory words are *not* shared - that is, they will only appear in one cache. If we keep track of which words are shared and which are not, we can reduce the amount of broadcasting necessary. There are two main reasons why more memory bandwidth is used: in an invalidation scheme, only the *first* change to a word requires an invalidation signal to be broadcast, whereas in a write broadcast scheme all changes must be signaled; and in an invalidation scheme only the *first* change to *any* word in a cache block must be signaled, whereas in a write broadcast scheme every word that is written must be signaled. On the other hand, in a write broadcast scheme we do not end up with a cache miss when trying to access a changed word, because the cached copy will have been updated to the correct value.

Processor Activity	Bus Activity	Contents of CPU A's cache	Contents of CPU B's cache	Contents of Memory Location X
				0
CPU A reads X	Cache miss for X	0		0
CPU B reads X	Cache miss for X	0	0	0
CPU A writes 1 to X	Write Broadcast for X	1	1	1
CPU B reads X		1	1	1

Figure 6.18 write back with broadcast

If different processors operate on different data items, these can be cached.

1. Once these items are tagged dirty, all subsequent operations can be performed locally on the cache without generating external traffic.

2. If a data item is read by a number of processors, it transitions to the shared state in the cache and all subsequent read operations become local.

In both cases, the coherence protocol does not add any overhead.

4.5.3Write-through vs. Write-back

In a write-back cache, the snooping logic must also watch for reads that access main memory locations corresponding to dirty locations in the cache (locations that have been changed by the processor but not yet written back).

At first it would seem that the simplest way to maintain coherence is to use a write-through policy so that every cache can snoop every write. However, the number of extra writes can easily saturate a bus. The solution to this problem is to use a write-back policy, but that leads to additional problems because there can be multiple writes that do not go to the bus, leading to incoherent data.

One approach is called write-once. In this scheme, the first write is a write-through to signal invalidation to other caches. After that, further writes can occur in write-back mode as long as there is no invalidation. Essentially, the first write takes ownership of the data, and another write from another processor must first deal with the invalidation and may then take ownership. Thus, a cache line has four states:

- Invalid
- Valid unwritten (valid)
- Valid written once (reserved)
- Valid written multiple (dirty)

The last two states indicate ownership. The trouble with this scheme is that if a non-owner frequently accesses an owned shared value, it can slow down to main memory speed or slower, and generate excessive bus traffic because all accesses must be to the owning cache, and the owning cache would have to perform a broadcast on its next write to signal that the line is again invalid.

One solution is to grant ownership to the first processor to write to the location and not allow reading directly from the cache. This eliminates the extra read cycles, but then the cache must write-through all cycles in order to update the copies.

We can change the scheme so that when a write is broadcast, if any other processor has a snoop hit, it signals this back to the owner. Then the owner knows it must write through again. However, if no other processor has a copy (signals snooping), it can proceed to write privately. The processor's cache must then snoop for read accesses from other processors and respond to these with the current data, and by marking the line as snooped. The line can return to private status once a write-through results in a no-snoop response.

One interesting side effect of ownership protocols is that they can sometimes result in a speedup greater than the number of processors because the data resides in faster memory. Thus, other processors gain some speed advantage on misses because instead of fetching from the slower main memory, they get data from another processor's fast cache. However, it takes a fairly unusual pattern of access for this to actually be observed in real system performance.

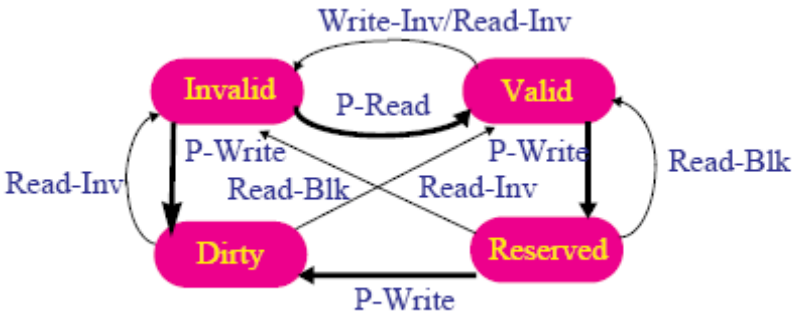


Figure 6.19 write once protocol

Disadvantages:

- If multiple processors read and update the same data item, they generate coherence functions across processors.

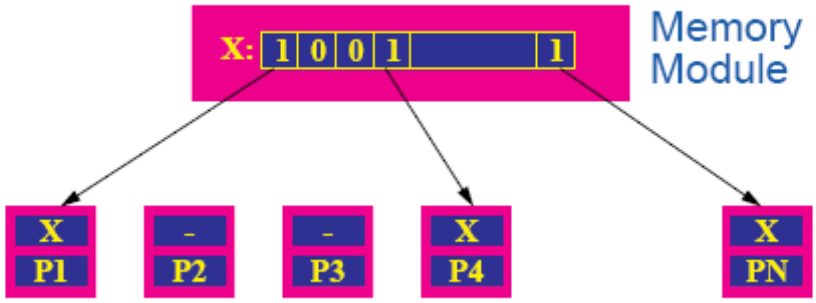
- Since a shared bus has a finite bandwidth, only a constant

Rather than flush the cache completely, hardware can be provided to "snoop" on the bus, watching for writes to main memory locations that are cached.

Another approach is to have the DMA go through the cache, as if the processor is writing it to memory. This results in all valid cache locations. However, any processor cache accesses are stalled during that time, and it clearly does not work well in a multiprocessor, as it would require copies being written to all caches and a protocol for write-back to memory that avoids inconsistency.

4.5.4 Directory-based Protocols

When a multistage network is used to build a large multiprocessor system, the snoopy cache protocols must be modified. Since broadcasting is very expensive in a multistage network, consistency commands are sent only to caches that keep a copy of the block. This leads to *Directory Based protocols*. A directory is maintained that keeps track of the sharing set of each memory block. Thus each bank of main memory can keep a directory of all caches that have copied a particular line (block). When a processor writes to a location in the block, individual messages are sent to any other caches that have copies. Thus the Directory-based protocols selectively send invalidation/update requests to only those caches having copies—the sharing set leading the network traffic limited only to essential updates. Proposed schemes differ in the latency with which memory operations are performed and the implementation cost of maintaining the directory. The memory must keep a bit-vector for each line that has one bit per processor, plus a bit to indicate ownership (in which case there is only one bit set in the processor vector).



.figure 6.20 Directory based protocol

These bitmap entries are sometimes referred to as the presence bits. Only processors that hold a particular block (or are reading it) participate in the state transitions due to coherence operations. Note that there may be other state transitions triggered by processor read, write, or flush (retiring a line from cache) but these transitions can be handled locally with the operation reflected in the presence bits and state in the directory. If different processors operate on distinct data blocks, these blocks become dirty in the respective caches and all operations after the first one can be performed locally.

If multiple processors read (but do not update) a single data block, the data block gets replicated in the caches in the shared state and subsequent reads can happen without triggering any coherence overheads.

Various directory-based protocols differ mainly in how the directory maintains information and what information is stored. Generally speaking the directory may be central or distributed. Contention and long search times are two drawbacks in using a central directory scheme. In a distributed-directory scheme, the information about memory blocks is distributed. Each processor in the system can easily "find out" where to go for "directory information" for a particular memory block. Directory-based protocols fall under one of three categories:

Full-map directories, limited directories, and chained directories.

This full-map protocol is extremely expensive in terms of memory as it store enough data associated with each block in global memory so that every cache in the system can simultaneously store a copy of any block of data.. It thus defeats the purpose of leaving a bus-based architecture.

A limited-map protocol stores a small number of processor ID tags with each line in main memory. The assumption here is that only a few processors share data at one time. If there is a need for more processors to share the data than there are slots provided in the directory, then broadcast is used instead.

Chained directories have the main memory store a pointer to a linked list that is itself stored in the caches. Thus, an access that invalidates other copies goes to memory and then traces a chain of pointers from cache to cache, invalidating along the chain. The actual write operation stalls until the chain has been traversed. Obviously this is a slow process.

Duplicate directories can be expensive to implement, and there is a problem with keeping them consistent when processor and bus accesses are asynchronous. For a write-through cache, consistency is not a problem because the cache has to go out to the bus anyway, precluding any other master from colliding with its access.

But in a write-back cache, care must be taken to stall processor cache writes that change the directory while other masters have access to the main memory.

On the other hand, if the system includes a secondary cache that is inclusive of the primary cache, a copy of the directory already exists. Thus, the snooping logic can use the secondary cache directory to compare with the main memory access, without stalling the processor in the main cache. If a match is found, then the comparison must be passed up to the primary cache, but the number of such stalls is greatly reduced due to the filtering action of the secondary cache comparison.

A variation on this approach that is used with write-back caches is called dirty inclusion, and simply requires that when a primary cache line first becomes dirty, the secondary line is similarly marked. This saves writing through the data, and writing status bits on every write cycle, but still enables the secondary cache to be used by the snooping logic to monitor the main memory accesses. This is especially important for a read-miss, which must be passed to the primary cache to be satisfied.

The previous schemes have all relied heavily on broadcast operations, which are easy to implement on a bus. However, buses are limited in their capacity and thus other structures are required to support sharing for more than a few processors. These structures may support broadcast, but even so, broadcast-based protocols are limited.

The problem is that broadcast is an inherently limited means of communication. It implies a resource that all processors have access to, which means that either they contend to transmit, or they saturate on reception, or they have a factor of N hardware for dealing with the N potential broadcasts.

Snoopy cache protocols are not appropriate for large-scale systems because of the bandwidth consumed by the broadcast operations

In a multistage network, cache coherence is supported by using cache directories to store information on where copies of cache reside.

A cache coherence protocol that does not use broadcast must store the locations of all cached copies of each block of shared data. This list of cached locations whether centralized or distributed is called a cache directory. A directory entry for each block of data contains a number of pointers to specify the locations of copies of the block.

Distributed directory schemes

In scalable architectures, memory is physically distributed across processors. The corresponding presence bits of the blocks are also distributed. Each processor is responsible for maintaining the coherence of its own memory blocks. Since each memory block has an owner its directory location is implicitly known to all processors. When a processor attempts to read a block for the first time, it requests the owner for the block. The owner suitably directs this request based on presence and state information locally available. When a processor writes into a memory block, it propagates an invalidate to the owner, which in turn forwards the invalidate to all processors that have a cached copy of the block. Note that the communication overhead associated with state update messages is not reduced. Distributed directories permit O(p) simultaneous coherence operations, provided the underlying network can sustain the associated state update messages. From this point of view, distributed directories are inherently more scalable than snoopy systems or centralized directory systems. The latency and bandwidth of the network become fundamental performance bottlenecks for such systems.

4.6 Keywords

cache A high-speed memory, local to a single processor , whose data transfers are carried out automatically in hardware. Items are brought into a cache when they are referenced, while any changes to values in a cache are automatically written when they are no longer needed, when the cache becomes full, or when some other process attempts to access them. Also To bring something into a cache.

cache consistency The problem of ensuring that the values associated with a particular variable in the *caches* of several processors are never visibly different.

associative memory: Memory that can be accessed by content rather than by address; content addressable is often used synonymously. An associative memory permits its user to specify part of a pattern or key and retrieve the values associated with that pattern.

direct mapping :A cache that has a set associativity of one so that each item has a unique place in the cache at which it can be stored.

4.7 Summary

In this lesson we had learned how cache memory in multiprocessor is organized and how its address are generated both for physical and virtual address. Various techniques of cache mapping are discussed.

Mapping technique	Advantage	disadvantage
Direct Mapping	Fast lookup (only one comparison needed). Cheap hardware (no associative comparison). Easy to decide	Contention for lines
Fully associative	Minimal contention for lines. Wide variety of replacement algorithms feasible.	The most expensive of all organizations, due to the high cost of associative-comparison hardware.

Set associative mapping trade off advantage and disadvantage of direct and fully associative mapping.

We had discussed about the shared memory organization and how consistency is maintained in it. There are various issues of synchronization and event handling on which various consistency models are designed. Various techniques through which cache coherence is maintained are discussed. Bus based systems are not scalable and not efficient for the processor to snoop and handle the traffic. Directories based system is used in cache coherence for large MPs *Cache coherency protocols maintain exclusive writes in a multiprocessor. Memory consistency policies determine how different processors observe the ordering of reads and writes to memory.* Snoopy caches are typically associated with multiprocessor systems based on broadcast interconnection networks such as a bus or a ring. All processors snoop on (monitor) the bus for transactions. Directory based systems the global memory is augmented with a directory that maintains a bitmap representing cache-blocks and the processors at which they are cached.

4.8 Self assessment questions

1. With diagram, explain the interconnection structures in a generalized multiprocessor system with local memory, private caches, shared memory and shared peripherals.
2. Discuss advantage and disadvantage of various cache mapping techniques
3. Discuss different page replacement polices.
4. Describe the Cache coherence problems in data sharing and in process migration.
5. Draw and explain 2 state-transition graphs for a cache block using write-invalidate snoopy protocols.
6. Explain the Goodman's write-once cache coherence protocol using the write-invalidate policy on write-back caches.
7. Discuss the basic concept of a directory-based cache coherence scheme.
8. Mention and explain the three types of cache directory protocols.

4.9 **References/Suggested readings**

Advance Computer architecture: Kai Hwang

5.0 Objective

In this lesson we will study about advance concepts of improving the performance of multiprocessor. The techniques studied is multithreading , multiple context processor and data flow architecture.

5.1 Introduction

The computers are basically designed for execution of instructions, which are stored as programs in the memory. These instructions are executed sequentially and hence are slow as the next instruction can be executed only after the output of pervious instruction has been obtained. As discussed earlier to improve the speed and through put the concept of parallel processing was introduced. To execute the more than one instruction simultaneously one has to identify the independent instruction which can be passed to separate processors. The parallelism in multiprocessor can be implemented on principle in three ways:

Instruction Level Parallelism

The potential of overlap among instructions is called ***instruction-level parallelism (ILP)*** since the instructions can be evaluated in parallel. 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.

Data Level Parallelsim

The simplest and most common way to increase the amount of parallelism available among instructions is to exploit parallelism among iterations of a loop. This type of parallelism is often called ***loop-level parallelism*** as an example of it vector processor.

Difficult to continue to extract instruction-level parallelism (ILP) or data-level parallelism (DLP) from a single sequential thread of control. Many workloads can make use of thread-level parallelism (TLP)

Thread Level Parallelism

Thread level parallelism (TLP) is the act of running multiple flows of execution of a single process simultaneously. TLP is most often found in applications that need to run independent, unrelated tasks (such as computing, memory accesses, and IO) simultaneously. These types of applications are often found on machines that have a high workload, such as web servers. TLP is a popular ground for current research due to the rising popularity of multi-core and multi-processor systems, which allow for different threads to truly execute in parallel. The TLP can be implemented either through multiprogramming (i.e., run independent sequential jobs) or from multithreaded applications (i.e., run one job faster using parallel threads). Thus Multithreading uses TLP to improve utilization of a single processor

As a designers perspective there are various possible ways in which one can design a system depending on the way we execute the instructions. Four possible ways are

Control flow computers : The next instruction is executed when the last instruction as stored in the program has been executed

Data flow computers An instruction executed when the data (operands) required for executing that instruction is available

Demand driven computers : An instruction is executed when the results of the instruction which is required as input by other instruction is available.

Pattern driven computers : An instruction is executed when we obtain a particular data patterns as output.

5.2 Multi-Threading

In the multithreaded execution model, a program is a collection of partially ordered threads, and a thread consists of a sequence of instructions which are executed in the conventional von Neumann model. Multithreading is the process of executing multiple threads concurrently on a processor. It takes the idea of processes sharing the CPU to a lower level, and allows threads to be switched off and on the processor without any latency. Multithreading processors technology developed by Intel that enables multithreaded software applications to execute threads in parallel on a single multi-core processor instead of processing threads in a linear fashion i.e., thus Multi-Threading, a microprocessor's "core" processor can execute two (rather than one) concurrent streams (or threads) of instructions sent by the operating system. Having two streams of execution units to work on allows more work to be done by the processor during each clock cycle. To the operating system, the multi-Threading microprocessor appears to be two separate processors. It is a feature of Intel's IA-32 processor.

Multithreading demands that the processor be designed to handle multiple contexts simultaneously on a context switching basis. Firstly let’s study the multithread computation model. Let us consider the system where memories are distributed to form global address space. The machine parameter on which machine is analyzed are

a. the latency (L) this include network delay, cache miss penalty, and delay caused by contention in split transaction
b. the number of thread the number of thread that can be interleaved in each processor. A thread is represented by a context consisting a program counter, register set and required context status word.
c. The context switching overhead: this refer to cycle lost in performing context switching in processor. This depends on the switching mechanism and the amount of processor state devoted to maintaining the active thread.
d. The interval between switches: this refer to cycle between switches triggered by remote reference. This inverse of rate of request.

There are a number of ways that multithreading can be implemented, including: fine-grained multithreading, coarse-grained multithreading, and simultaneous multithreading.

Fine-Grained Multithreading

Fine-grained multithreading involves instructions from threads issuing in a round-robin fashion--one instruction from process A, one instruction from process B, another from A, and so on (note that there can be more than two threads). This type of multithreading applies to situations where multiple threads share a single pipeline or are executing on a single-issue CPU.

Coarse-Grained Multithreading

The next type of multithreading is coarse-grained multithreading. Coarse-grained multithreading allows one thread to run until it executes an instruction that causes a latency (cache miss), and then the CPU swaps another thread in while the memory access completes. If a thread doesn't require a memory access, it will continue to run until its time limit is up. As with fine-grained multithreading, this applies multiple threads sharing a single pipeline or executing on a single-issue CPU.

Simultaneous Multithreading (SMT)

Simultaneous multithreading is a refinement on coarse-grained multithreading. The scheduling algorithm allows the active thread to issue as many instructions as it can (up to the issue-width) to keep the functional units busy. If a thread does not have sufficient ILP to do this, other threads may issue instructions to fill the empty slots. SMT only applies to superscalar architectures which are characterized by multiple-issue CPUs. With the advent of multithreaded architectures, dependence management has become easier due to availability of more parallelism. But, the demand for hardware resources has increased. In order for the processor to cater efficiently to multiple threads, it would be useful to consider resource conflicts between instructions from different threads. This need is greater for simultaneous multithreaded processors, since they issue instructions from multiple threads in the same cycle. Similar to the operating system’s interest in maintaining a good job mix, the processor is now interested in maintaining a good mix of instructions. One way to achieve this is for the processor to exploit the choice available during instruction fetch. To aid this, a good thread selection mechanism should be in place. Dependences - data and control - limit the exploitation of instruction level parallelism (ILP) in processors. This is especially so in superscalar processors, where multiple instructions are issued in a single cycle. Hence, a considerable amount of

research has been carried out in the area of dependence management to improve processor performance.

Data dependences are of two types: true and false. False data dependences: anti and output dependences are removed using register renaming, a process of allocating different hardware registers to an architectural register. True data dependences are managed with the help of queues where instructions wait for their operands to become available. The same structure is used to wait for FUs. Control dependences are managed with the help of branch prediction.

Multithreaded processors add another dimension to dependence management by bringing in instruction fetch from multiple threads. The advantage in this approach is that the latencies of true dependences can be covered more effectively. Thus thread-level parallelism is used to make up for lack of instruction-level parallelism.

Simultaneous multithreading (SMT) combines the best features of multithreading and superscalar architectures. Like a superscalar, SMT can exploit instruction-level parallelism in one thread by issuing multiple instructions each cycle. Like a multithreaded processor, it can hide long latency operations by executing instructions from different threads. The difference is that it can do both at the same time, that is, in the same cycle.

The main issue in SMT is effective thread scheduling and selection. While scheduling of threads from the job mix may be handled by the operating system, selection of threads to be fetched is handled at the microarchitecture level. One technique for job scheduling called Symbiotic Job scheduling collects information about different schedules and selects a suitable schedule for different threads.

A number of techniques have been used for thread selection. The ***Icount*** feedback technique gives the highest priority to the threads that have the least number of instructions in the decode, renaming, and queue pipeline stages. Another technique minimizes branch mispredictions by giving priority to threads with the fewest outstanding branches. Yet another technique minimizes load delays by giving priority to threads with the fewest outstanding on-chip cache misses. Of these the Icount technique has been found to give better results.

Costs occurred in implementing Multithreading

• Each thread requires its own user state

– PC

– GPRs

• Also, needs its own system state

– virtual memory page table base register

– exception handling registers

• *Other overheads:*

– Additional cache/TLB conflicts from competing threads

– (or add larger cache/TLB capacity)

– More OS overhead to schedule more threads (where do all

these threads come from?)

5.2.1 Multiple context processor

Multithreaded systems are constructed with multiple context processors. Multiple context processors have been proposed as an architectural technique to mitigate the effects of large memory latency in multiprocessors. It allows multiple instructions to issue into pipeline from each context. This could lead to pipeline hazards, so other safe instructions could be interleaved into the execution. For example the Horizon & Tera the compiler detects such data dependencies and the hardware enforces it by switching to another context if dependency is being detected. This is implemented by inserting into each instruction a field which indicates its minimum number of independent successors over all possible control flows.

Context switching policies.

Switching from one thread to another is performed according to one of the following policies :

l Switching on every instruction: the processor switches from one thread to another every cycle. In other words, it interleaves the instructions from different threads on a cycle-by-cycle basis.

2 Switching on block of instructions: blocks of instructions from different threads are interleaved.

3. Switching on every load: whenever a thread encounters a load instruction, the processor switches to another thread after that load instruction is issued. The context switch is irrespective of whether the data is local or remote.

4. Switching on remote load: processor switches to another thread only when current thread encounters a remote access.

5. Switch on cache miss: This policy correspond the case where a context is preempted when it causes a cache miss.

Multithreaded distributed-memory multiprocessor architectures are composed of a number of (multithreaded) processors, each with its memory, and an interconnection network. The long memory latencies and unpredictable synchronization delays are tolerated by context switching, i.e., by suspending the current thread and switching the processor to another 'ready' thread provided such a thread is available.

Lets assume that the context switching takes place on every load. That is, if the executed instruction issues an operation for accessing either a local or a remote memory location, the execution of the current thread suspends, the thread changes its state to waiting, and another ready thread is selected for execution. When the long latency operation for which a thread was waiting is satisfied, the thread becomes ready and joins the pool of ready threads waiting for execution. The thread that is being executed is said to be executing.

There are two schemes for implementing multiple-context processors. The first scheme switches between contexts only on a cache miss, while the other interleaves the contexts on a cycle-by-cycle basis. Both schemes provide the capability for a single context to fully utilize the pipeline. We show that cycle-by-cycle interleaving of contexts provides a performance advantage over switching contexts only at a cache miss. This advantage results from the context interleaving hiding pipeline dependencies and reducing the context switch cost. In addition, we show that while the implementation of the interleaved scheme is more complex, the complexity is not overwhelming. As pipelines get deeper and operate at lower percentages of peak performance, the performance advantage of the interleaved scheme is likely to justify its additional complexity.

5.2.3 Multidimensional architecture

The architecture of massively parallel processors has evolved from 1-D rings to 2-D and 3-D meshes or tori. The USC orthogonal multiprocessor (OMP) can be extended to

higher dimensions. Here instead of using hierarchical busses or switched network architecture in one dimension, multiprocessor architecture can be extended to a higher dimensionality or multiplicity along each dimension. A example of is Orthogonal multiprocessor (OMP architecture) with n processor simultaneously access n rows or columns of interleaved memory modules. The N*N memory mesh is interleaved in both dimensions. In other words each row is n-way interleaved and so is each column of memory modules. There are 2n logical buses spanning in two orthogonal directions. The memory controller synchronizes the row and column access of shared memory.

5.3 Data flow computers

In this lesson we also will study about data flow model. Data flow machines is an alternative of designing a computer that can store program systems. The aim of designing parallel architecture is to get high performing machines. The designing of new computer is based on following three principles:

- To achieve high performance
- To match technological progress
- To offer better programmability in application areas

Data flow is one of the technique that meet the above requirement and hence are found useful for designing the future supercomputer. Before we study in detail about these data flow computers lets revise the drawbacks of processors based on pipeline architecture.

The major hazards are

- Structural hazards
- **Data** hazards due to
  - true dependences which happens in case of WAR or
  - false dependences also called name dependencies : anti and output dependences (RAW or WAW)
- Control hazards

Among these the Data hazards due to **true dependences** and care is required to avoid it while the **control hazards** can be handled if next instructions in the pipeline to be executed is basically from different contexts Hence if data dependency can be removed the performance of the system will definitely improve. It can removed by one of the followings techniques:

- By renaming the data this will lead to extra burden to complier as this operation is performed by compiler
- By renaming hardware as done in advanced superscalars computers
- By following the single-assignment rule as done in the dataflow **computers**

Data flow computers are based on the principle of data driven computation which is very much different from the von Neumann architecture which is basically based on the control flow while where the data flow architecture is designed on availability of data hence also called data driven computers. There are various types data flow model are static dynamic, VLSI, Hybrid we will discussing about them in this module. The concept of data flow computing was originally developed in 1960's by Karp and Miller. They used a graphical means of representing computations. Later in the early 1970's Dennis and later other developed the computer architectures based on data flow systems. Concept of dataflow computing finds its application in specialized architectures for Digital Signal Processing (DSP) and specialized architectures for demanding computation in the fields of graphics and virtual reality.

Data driven computing and languages

In order to under how Dataflow is different from Control-Flow. Lets see the working of von Neumann architecture which is based on the control flow computing model. Here each program is sequence of instructions which are stored in memory. These a series of addressable instructions store the information about the an operation along with the information about the with memory locations that store the operand or in case of interrupt or some function call it store the address of the location where control has to transferred or in case of conditional transfer it specifies the status bits to be checked and location where the control has to transferred.

The next instruction to be executed depends on what happened during the execution of the current instruction. Thus accordingly the address of next instruction to be executed is transferred to PC. And on next clock pulse the instruction is executed, the operands are fetched from the desired memory location as required in the instruction. Here the instruction is also executed even if some of its operands are not available yet (e.g. uninitialized). The fetching of data and instruction from memory becomes bottleneck in

exploiting the parallelism to its maximum possible utility. The key features of control flow model are

- Data is passed between instructions via reference to shared memory cells
- Flow of control is implicitly sequential but special control operators can be used for explicit parallelism
- Program counter are used to sequence the execution of instruction in centralized control environment

However the data driven model accept the execution of any instruction only on availability of the operand. Data flow programs are represented by directed graphs which show the flow of data between instructions. Each instruction consists of an operator, one or two operands and one or more destinations to which the result is to be transferred. The key features of data driven model are as follows:

- Intermediate results as well as final result are passed directly as data token between instruction.
- There is no concept of shared data storage as used in traditional computers
- In contrast to control driven computers where the program has complete control over the instruction sequencing here the data driven computer the program sequencing is constrained only by data dependency among the instructions.
- Instructions are examined to check the operand availability and if functional unit and operand both are available the instruction is immediately executed.

As the fetching of data every time from memory which is part of instruction cycle of von Neumann model is overcome by transferring the available data the bottleneck in exploiting parallelism are missing or we can say parallelism is better implemented in data driven system. This is because there is no concept of shared memory cells and one can say that data flow diagram are free from side effects as in data driven computers the operands are directly transferred as token value instead of address variable as in case of control flow model. There is always a chance of side effect as the change of memory words in case of control flow computers.

The data driven concept means asynchrony which means that many instructions can be executed simultaneously no PC and global updateable store is required. Information required in a data flow computer are operation packets that are composed of opcode,

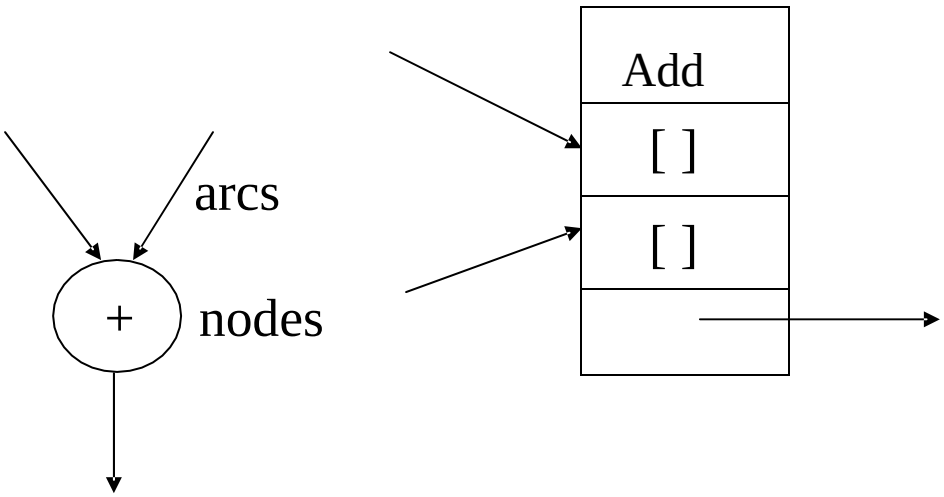
operand and destinations of its successor instructions and data token which is formed with a result value and its destinations. Many of these packets are passed among various resource sections in a data flow machine. One basic rules involved in computation are in data flow computer are :

- *Enabling rule* which states that a**n instruction is enabled (i.e. executable) if all operands are available however in control flow computer as in case of** von Neumann model, an instruction is enabled if it is pointed to by PC.
- The *computational rule* or *firing rule* , specifies when an enabled instruction is actually executed. Thus when a**n instruction is fired (i.e. executed) when it becomes enabled** and effect of firing of an instruction is the consumption of its input **data** (operands) and generation of output **data** (results).

Data Flow graph

Data flow computing as required to implement the parallelism hence it is required to analysis the data dependency . Data flow computational model uses directed graph G = (V ,E), which is also called as data dependency graph or DataFlow Graph (DFG). An important characteristic of dataflow graph is its ability to detect parallelism of computation by finding various types of dependency among the data. This graph consists of nodes that represent the operations (opcode) and an arc connects the two node and it indicates how the data flow between these nodes or we can say arcs are pointers for forwarding the data tokens. DFG is used for the description of behavior of data driven computer. Vertex v _ V is an actor, a directed edge e _ E describes precedence relationships of source actor to sink actor and is guarantee of proper execution of the dataflow program. This assures proper order of instructions execution with contemporaneous parallel execution of instructions. Tokens are used to indicate presence of data in DFG. Actor in dataflow program can be executed only in case there is a presence of a requisite number of data values (tokens) on input edges of an actor. When firing an actor execution, the defined number of tokens from input edges is consumed and defined number of tokens is produced to the output edges.

The figure below represent a data flow graph which is basically is a directed graph consist of arcs (edges) which represent data flow, and nodes, which represent operations.



These graphs demonstrate the data dependency among the instructions. In data flow computers the machine level program is represented by data flow graphs.

In conventional computer the only focus while designing program is for assignment of control flow. To implement the parallel computing in this architecture if we need many processing elements (electronic chips like ALU) working in parallel simultaneously. Now designing a prospect of programming for each chip individually becomes unthinkable. Researchers have designed various computer architects based on the von Neumann principle i.e., to create a single large machine from many processors like Illiac IV, Cmmp, etc. The major problem for implementing implicit parallelism in these machine (based on von Neumann architecture) is

- (Centralized) sequential control
- Shared mernory cells

Data flow languages make a clean break from the von Neumann framework, giving a new definition to concurrent programming languages. They manage to make optimal use of the implicit parallelism in a program. Consider the following segment:

l . P = X + Y (waits for availability of input value for X and Y)

2. Q = P *I* Y (as P is required input it must waits for instruction 1 to finish)

3. R = X * P (as P is required input it must waits for instruction 1 to finish)

4. **S** = R - Q(as R and Q are required as input it must waits for instruction 2 and 3 to finish)

5. T = R * P (as R is required input it must waits for instruction 3 to finish)

6. U = S *I* T (as S and T are required as input it must waits for instruction 4 and 5 to finish)

Permissible computation sequences of the above program for the conventional von Neumann machine are

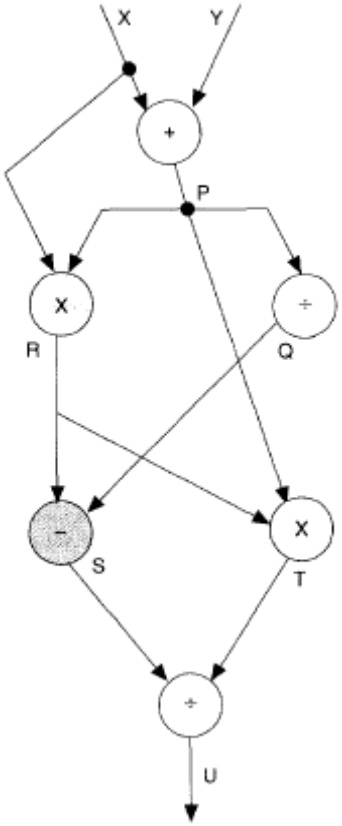
(1,2.3.4,5,6)

(1,3,2,5,4,6)

(1,3,5,2,4,6)

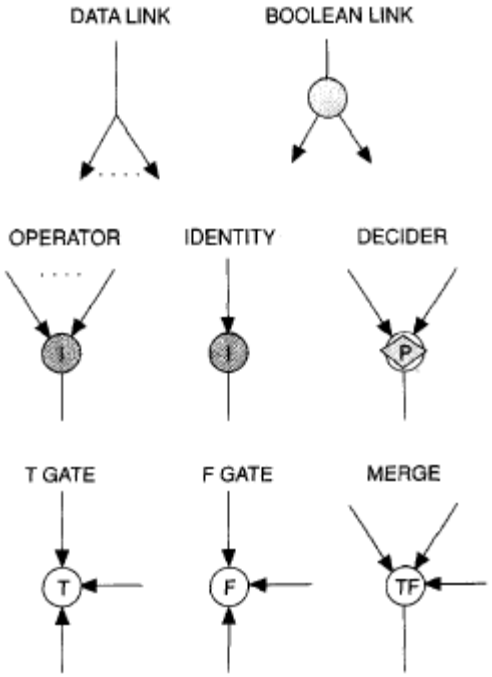
(1,2,3,5,4,6) and

(1,3,2,4.5,6)



On parallel computer it is possible to perform these 6 operations in three steps by performing 2,3 instruction simultaneously and 4,5 also simultaneously. Thus sequence of instruction can be [1, (2,3) and (4,5)} The above program is shown as data flow graph. A

dataflow program is a graph, where nodes represent operations and edges represent data paths



Various notations used to construct a data flow diagram with help of operators (nodes) and links (arcs)

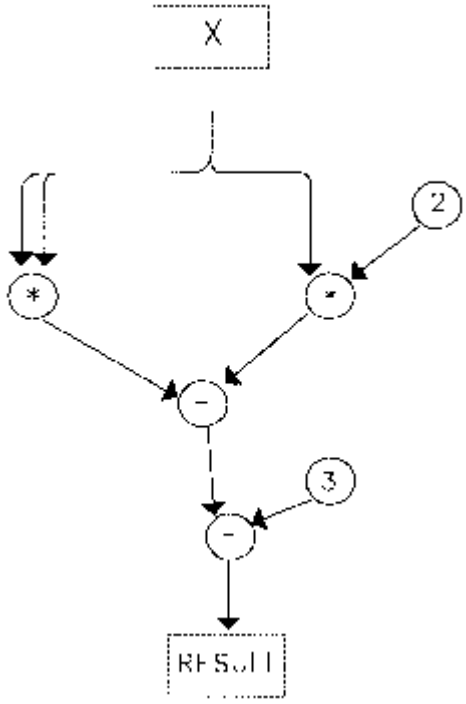
The above Figure show various commonly used symbols in a data flow graph. Data links are used to transmit all types of data whether it is integer or float except for Boolean values, for which special links are used as shown in the figure. Any operator is stored in node and has two or more input and one output except for the identity operator that has one input arc and it transfer the value of data token unchanged. For conditional and iterative computations deciders, gates and merge operators are used in data flow graphs. A decider requires a value from each of its input arcs and test the condition and according to the condition it satisfies it transmit a truth value. Control tokens bearing boolean values control the flow of data tokens by means of the gates namely *T* gates, the ***F*** gates, and the merge operators where the ***T*** gate will transmit a data token from its input arc to its output arc if the value on its control input is true. It will absorb a data token from its data input arc and place nothing on its output ARC IF IT RECIEVES A False value. The F gate also have similar behavior except now the control test for false condition. **A** merge

operator has ***T*** and *F* input arcs and a truth-value control arc. When a true value is received on its control arc, the data token on the *T* input is transmitted.

The token on the other unused input arc **is** discarded. Similary the false input is passed to the output when the control arc is false.

As said earlier in data flow graphs the tokens flow through the graph. When a node receives the tokens from the incoming edge it will execute and put the result as tokens on its output edges. Unlike control flow computer there is no predetermined sequence of the execution of a data flow computer rather here the data drives the order of execution. Once a node is activated and operation stored in its node is performed, this process s also called "fired" and the output of the operation is passed along the arc to waiting node. This process is repeated until all of the nodes are fired and the final result is created. The parallelism is implemented as simultaneously more than one node can be fired.

Lets see the data flow diagram for an equation x^2 -2x + 3



Data flow diagram for the equation $x^2 - 2x + 3$

Lets take another example of implementing a simple problem of finding the root of a quadratic equation (algorithm assumes real roots) using the data flow graph. For calculating the roots function quad(a,b,c) performs the following steps:

quad(a, b, c)

{

t1 = a*c;

t2 = 4*t1;

t3 = b*b;

t4 = t3 - t2;

t5 = sqrt(t4);

t6 = -b;

t7 = t6 - t5;

t8 = t7 + t5;

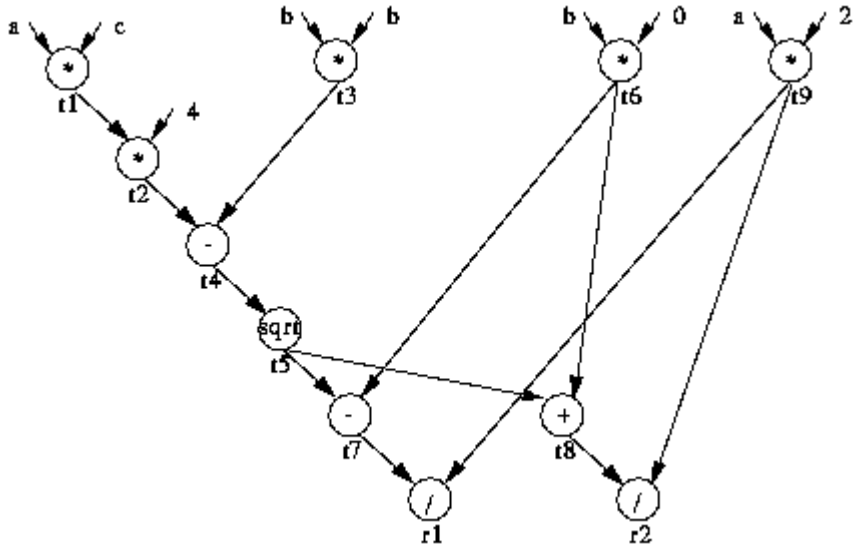
t9 = 2*a;

r1 = t7/t9;

r2 = t8/t9;

}

In the control flow computer this algorithm is implemented line by line. In order to implement it through data flow computr one should first note the dependancies between each operation. For example t2 can not be computed before t1, but t3 could be computed before t1 or t2.



Lets consider example of iterative computation z = *x" and represent it* by the data flow graph Figure 5. 3. using the symbols shown in Fig.2. The input reuired are for inputs ***x,n:*** Variable used are y***,i***

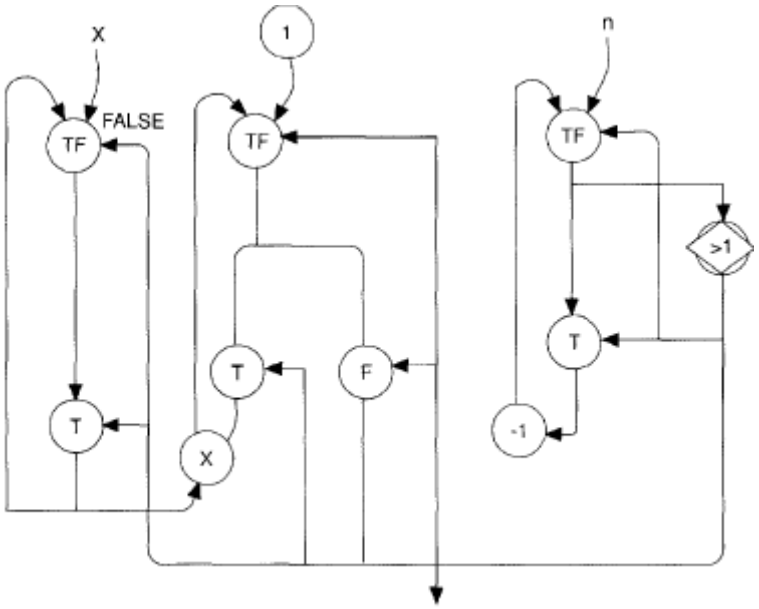
y= 1 ***:i=n***

while i>0 do

begin y= y*x , i= i-1 end

z=y

output z



The computation involve successive calculation of loop variable values i.e., y and I and these value will pass through the links and test the condition. The initial values of the control arcs are labeled false to initiate computation. The result z will be obtained when the decider's output is false.

Two important characteristics of dataflow graphs are

- *Functionality*: The evaluation of a dataflow graph is equivalent to evaluation of the corresponding mathematical function on the same input **data**.
- *Composability*: Dataflow graphs can be combined to form new graphs.

Major design issues in implementing Data flow computers

Although the data flow computers as far as theoretical aspect is considered is proved to very good and appears it should generate the desired results but when it comes toward the practical realization of a data flow computer, we identify below a number of important technical problems that remain to be solved:

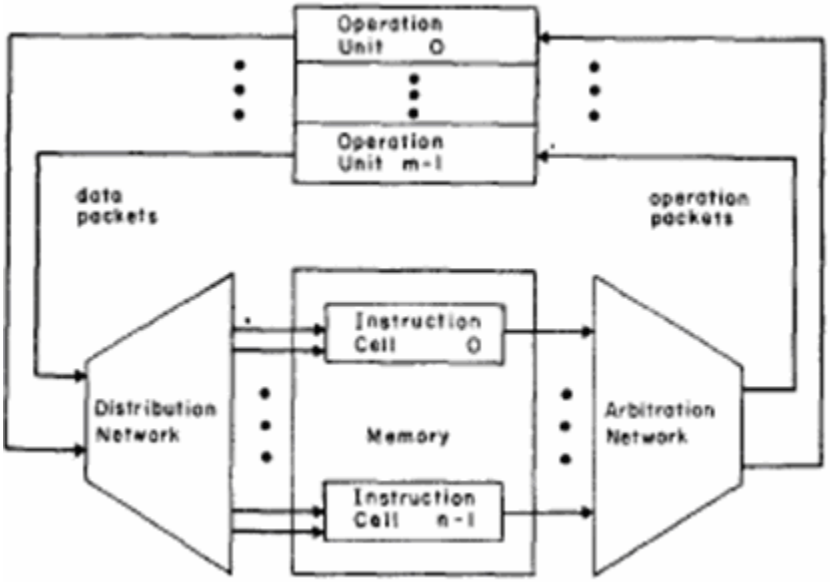
1. The development of efficient data flow languages which are easy to use and to be interpreted by machine hardware
2. The decomposition of programs and the assignment of program modules to data flow processors
3. Controlling and supporting large amounts of interprocessor communication with cost-effective packet-switched networks
4. Developing intelligent data-driven mechanisms for either static or dynamic data flow machines
5. Efficient handling of complex data structures, such as arrays, in a data flow environment
6. Developing a memory hierarchy and memory allocation schemes for supporting data flow computations
7. A large need for user acquaintance of functional data flow languages, software supports, data flow compiling, and new programming methodologies
8. Performance evaluation of data flow hardware in a large variety of application domains, especially in the scientific areas

Disadvantage of dataflow model

- **Data flow programs tends to waste lot of memory space for increased code length due to single assignment rule and excessive copying of data array.**
- **The data driven at instruction level cause excessive pipeline overhead per instruction which may destroy the benefits of parallelism specially in case where program involve the iterative computing.**
- **When data flow computer become large with high number of instruction cells and processing elements, the packet switched network used becomes cost prohibitive to the entire system.**
- **Data** hazards due to
    - ③ true dependences □ dataflow principle
    - ③ name (false) dependences □ not present due to single assignment rule in dataflow languages
- Control hazards □ transformed into **data** dependences

Data Flow Computer architecture

The data flow computer architecture can be classified as pure data flow computers and hybrid data flow computers. Earlier the researchers designed pure data flow computers based on data flow computation principles later researchers observed the shortcoming of pure data flow computer and combine the principle of conventional computer and data flow computer to design hybrid data flow computers



- The Pure dataflow **computers are further classified as the** :
  - static,
  - **dynamic**
  - Very Large Scale Integration (VSLI) Dataflow
  - and the explicit token store architecture.
- Hybrid dataflow **computers**:

These computers are designed by augmenting the dataflow computation model with control-**flow** mechanisms, such as

- ③ RISC approach,
- ③ complex machine operations,
- ③ multithreading,
- ③ large-grain computation,
- ③ etc.

Let begin the study about the Pure Dataflow computer. The basic principle of any Dataflow computer is data driven and hence it executes a program by receiving, processing and sending out *token.*

These token consist of some ***data*** and a *tag*. These tags are used for representing all types of dependences between instructions. Thus dependencies are handled by translating them into *tag matching* and *tag transformation*. The processing unit is composed of two parts matching unit that is used for matching the tokens and execution unit used for actual implementation of instruction. When the processing element gets a token the matching unit perform the matching operation and when a set of *matched tokens* the processing begins by execution unit. The type of operation to be performed by the instruction has to be fetched from the instruction store which is stored as the tag information. This information contains details about

- what operation has be performed on the **data**
- how to transform the tags.

The *matching unit* and the execution unit are connected through an asynchronous pipeline, with queues added between the stages. To perform fast token matching some form of fast associative memories are used. The various possible solution for the associative memory used to support token matching are.

- a real memory with associative access,
- a simulated memory based on hashing,
- or a direct matched memory.

Jack deniss and his associates at MIT have pioneered the area of data flow research and they came forward with two models called Dennis machine and Arvind machine. The Dennis machine has static architecture while Arvind used tagged token and colored activities and was designed for dynamic architecture.

There are variety of static, dynamic and also hybrid dataflow computing models.

In static model, there is possibility to place only one token on the edge at the same time. When firing an actor, no token is allowed on the output edge of an actor. It is called static model because token arms are not labeled and control tokens must be used to acknowledge the proper timing in the transferring data token from one node to another.

Disadvantage of the static model is impossibility to use dynamic forms of parallelism, such a loops and recursive parallelism. Computer with static dataflow computer architecture was designed by Denis and Misunas.

Dynamic model of dataflow computer architecture allows placing of more than one token on the edge at the same time. To allow implementation of this feature of the architecture, the concept of tagging of tokens was used. Each token is tagged and the tag identifies conceptual position of token in the token flow i.e., the label attached in each tag uniquely identify the context in which particular token is used. For firing an actor execution, a condition must be fulfilled that on each input edge of an actor the token with the same tag must be identified. After firing of an actor, those tokens are consumed and predefined amount of tokens is produced to the output edges of an actor.

There is no condition for firing an actor that no tokens must be on output edge of an actor. The architecture of dynamic dataflow computer was first introduced at Massachusetts Institute of technology (MIT) as a Tagged Token Dataflow Architecture.

Both static and dymnaic data flow architecture have a pipelined ring structure with ring having four resource sections

The memories used for storing the instruction

The processors unit that form the task force for parallel execution of enabled instruction

The routing network the routing network is used to pass the result data token to their destined instruction

The input output unit serves as an interface between data flow computer and outside world.

Hybrid dataflow architecture is a combination of control flow and data flow computation control mechanisms. Research in the field of computing with dataflow control of computation is predominantly limited to research laboratories where software simulations or hardware prototypes of dataflow computers are built.

Dataflow computers have yet a little impact in commercial computing, especially because of problematic design of optimal communication architecture and control of computing process. Although for example in 1985 Nippon Electronics Corporation (NEC) commercializes first dataflow processor μpd7281.

Static Dataflow

The static architecture was proposed by Dennis and Misunas [1975]. The static data flow computer data tokens are assumed to move along the arcs of the data flow program graph to the operator nodes. The nodal operations gets executed only when all its input are present at the input arc. Data flow graph used in the Dennis machine must follow the static execution rule that only one token is allowed to exist on any arc at any given time, otherwise successive sets of tokens cannot be distinguished thus instead of FIFO design of string token at arc is replace by simple design where the arc can hold at most one data token. This is called static because here tokens are not labeled and control token are used for acknowledgement purpose so that proper timing in the transferring data tokens from node to node can take place. Here the complete program is loaded into memory before execution begins. Same storage space is used for storing both the instructions as well as data. In order to implement this, acknowledge arcs are implicitly added to the dataflow graph that go in the opposite direction to each existing arc and carry an acknowledgment token Some example of static data flow computers are MIT Static Dataflow, DDM1 Utah Data Driven, LAU System, TI Distributed Data Processor, NEC Image Pipelined Processor

The graph itself is stored in the computer as a collection of activity *templates*, such that each template represents a node of the graph. The template as shown in the figure below holds opcode specifying operation to be performed by the node; a memory space to hold the value of the data token i.e., address of operand on each input arc, with a presence flag for each one; and a list of destination addresses for the output tokens referring to the operand slots in sub-sequent activity templates that need to receive the result value.

The instruction stored in memory cell is represented as in figure below

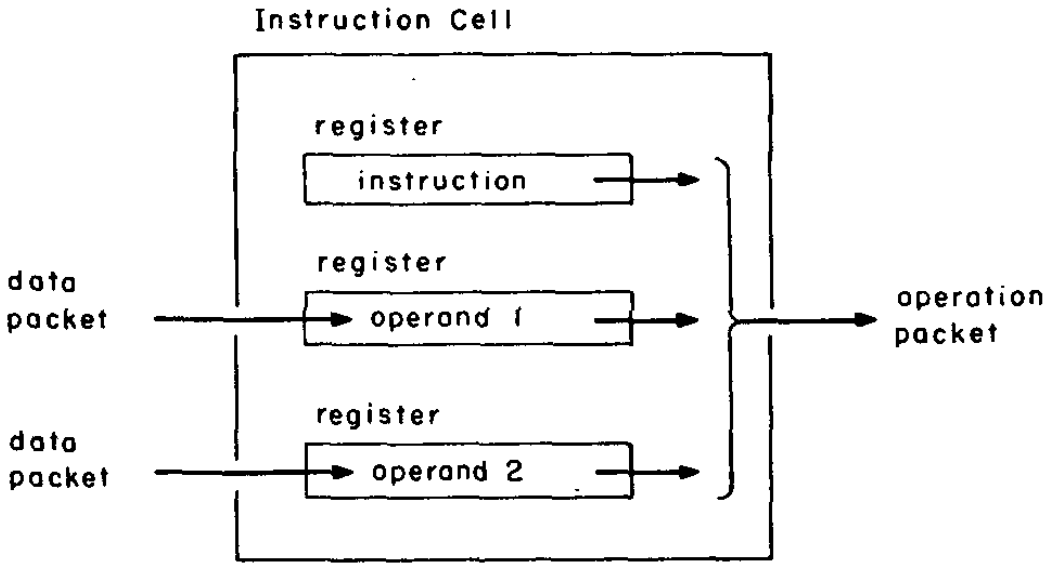
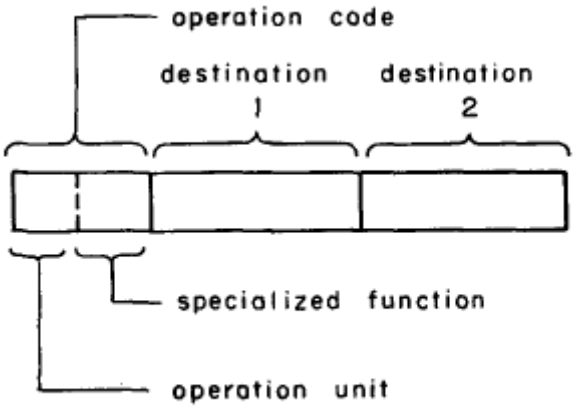


Figure 3. Operation of an Instruction Cell.



Processing elements receive the operation packets as the following form:

Opcode	Operands	Destinations

The advantage of this approach is that operands can only be affected by one selected node at a time. On the other hand, complex data structures, or even simple arrays could not reasonably be carried in the instruction and hence cannot be handles in the mechanism.

The resulting packet or token consist only of a value and a destination address and it has the following form:

Value	Destination

The output from an instruction cell generated when all of the input packets (tokens) have been received. Thus Static dataflow has the following firing rules:

1) Nodes are fire when all input tokens is released and the previous output token have been consumed.
2) Input tokens are then removed and new output tokens are generated.

The major drawback of this scheme is if different tokens are destined for the same destination data flow computer cannot be distinguished between them. However Static dataflow overcome this problem by allowing at most one token on any one arc which e*xtends the basic firing rule* as follows:

- **An enabled node is fired if there is no token on any of its output arcs**.

This rule allow pipeline computations and loops but does not allow the computation that involve the code sharing and recursion.

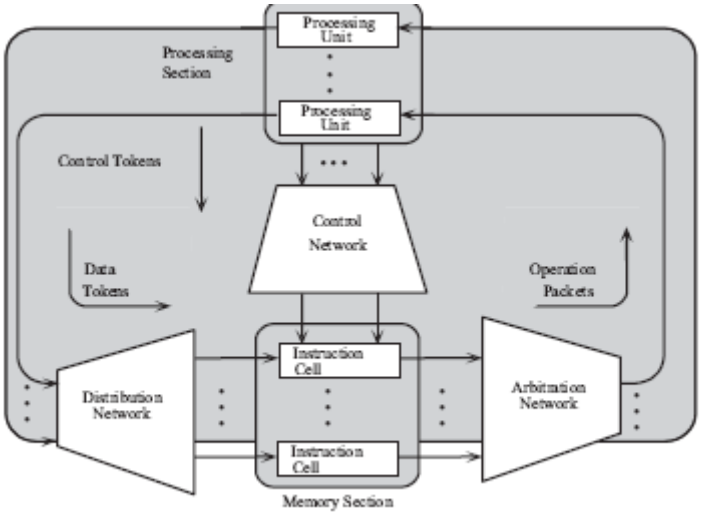
The static data flow adopts a handshaking acknowledgement mechanism which can take the form of special control tokens set from processors once they respond to a fired node. In order to implement this, acknowledge arcs are implicitly added to the dataflow graph that go in the opposite direction to each existing arc and carry an acknowledgment token. Thus additional *acknowledge signals* (tokens), travel along additional arcs from consuming to producing nodes. As acknowledgement concept is used we can redefine the firing rule in its original form:

- **A node is fired at the moment when it becomes enabled**.

Some example of dynamic dataflow computers are Manchester Dataflow, MIT Tagged Token, CSIRAC II , NTT Dataflow Processor Array, Distributed Data Driven Processor, Stateless Dataflow Architecture , SIGMA-1, Parallel Inference Machine (1984) (17)

Case study of MIT Static dataflow computer

The static dataflow mechanism was the first one to receive attention for hardware realization at MIT. MIT Static Dataflow Machine



It consist of five major sections connected by channels through which information is sent in the form of discrete tokens (packet):

- Memory section consist of instruction cells which hold instructions and their operands. The memory section is a collection of memory cells, each cell composed of three memory words that represent an instruction template. The first word of each instruction cell contains op-code and destination address(es), and the next two words represent the operands
- Processing section consists of processing units that units perform functional operations on data tokens . It consist of many pipelined functional units, which perform the operations, form the result packet(s), and send the result token(s) to the memory section.
- Arbitration network delivers operation packets from the memory section to the processing section. Its purpose is to establish a smooth flow of enabled instructions (i.e., instruction packet) from the memory section to the processing section. An instruction packet contains the corresponding op-code, operand value(s), and destination address(es).

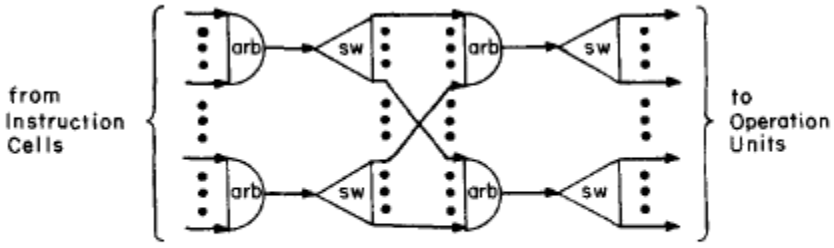


Figure 5. Structure of the Arbitration Network.

- Control network delivers a control token from the processing section to the memory section. The control network reduces the load on the distribution network by transferring the Boolean tokens and the acknowledgement signals from the processing section to the memory section.
- Distribution network delivers data tokens from the processing section to the memory section.

Instruction stored in the memory section are enabled for execution by the arrival of their operands in data token from the distributed network and control token from the control network. The instruction together with data and control are sent as operation packets to the processing section through arbitration network. The results of the instruction are sent through the distribution network and the control network to the memory section where they become input data for the other instruction.

Deficiencies of static dataflow

- Consecutive iterations of a loop can only be pipelined In certain cases, the single-token-per-arc limitation means that a second loop iteration cannot begin executing until the present loop has completed its execution
- The additional acknowledgment arcs increase data traffic by a factor of 1.5 to 2 in the system, without benefiting the computation. This is because here a node has to wait for acknowledgment tokens to arrive before it can execute again as a result , the time between two successive firings of a node increases.
- Lack of support for programming constructs that are essential to modern programming language
    - no procedure calls,

- no recursion.

Advantage:

The static architecture's main strength is that it is very simple it does not require a data structure like queue or stack to hold the list of tokens as only one token is allowed at a node. The static architecture is quickly able to detect whether or not a node is fireable. Additionally, it means that memory can be allocated for each arc at compile-time as each arc will only ever hold 0 or 1 data token. This implies that there is no need to create complex hardware for managing queues of data tokens: each arc can be assigned to a particular piece of memory store.

Dynamic Dataflow

In Dynamic machine data tokens are tagged (labeled or colored) to allow multiple tokens to appear simultaneously on any input arc of an operator. No control tokens are needed to acknowledge the transfer of data tokens among the instructions. The tagging is achieve by attaching a label with each token which uniquely identifies the context of that particular token. This dynamically tagged data flow model suggests that maximum parallelism is exploited from the program graph. However here the matching of token tags (labels or colors) is performed to merge them for instructions requiring more than one operand token. Thus the *dynamic model*, it exposed to an additional parallelism by allowing multiple invocations of a subgraph that is for implementation of an iterative loop by performing dynamically unfolding of the iterative loop. While this is the conceptual view of the tagged token model, in reality only one copy of the graph is kept in memory and tags are used to distinguish between tokens that belong to each invocation. A general format for instruction has opcode, the number of constants stored in instruction and number of destination for the result token. Each destination is identified by four fields namely the destination address, the input port at the destination instruction, number of token needed to enable the destination and the assignment function used in selecting processing element for the execution of destination instruction. The dynamic architecture has following characteristic different from static architecture. Here Program nodes can be instantiated at run time unlike in static architecture where it is loaded in the beginning. Also in dynamic architecture Several instances of an data packet are enabled and also Separate storage space used for instructions and data

Dynamic dataflow refer to a system in which the dataflow graph being executed is not fixed and can be altered through such actions as code sharing and recursion. Tags could be attached to the packets to identify tokens with particular computations.

Dynamic dataflow has the following firing rules:

1) A node fires when all input tokens with the same tag appear.
2) More than one token is allowed on each arc and previous output tokens need not be consumed before the node can be fired again.

The dynamic architecture requires storage space for the unmatched tokens. First in first out token queue for storing the tokens is not suitable. A tag contains a unique subgraph invocation ID, as well as an iteration ID if the subgraph is a loop. These pieces of information, taken together, are commonly known as the *color* of the token However no acknowledgement mechanism is required. The term "coloring" is used for the token labeling operations and tokens with the same color belong together.

Iteration level	Activation name	Index

Each field will hold a number. Iteration level identifies the particular activation for loop body, activation name represents the particular function call and index describe the particular element of an array.

Thus instead of the single-token-per-arc rule of the static model, the dynamic model represents each arc as a large queue that can contain any number of tokens, each with a different tag. In this scenario, a given node is said to be fireable whenever the same tag is found in a data token on each input arc. It is important to note that, because the data tokens are not ordered in the tagged-token model, processing of tokens does not necessarily proceed in the same order as they entered the system. However, the tags ensure that the tokens do not conflict, so this does not cause a problem. The tags themselves are generated by the system. Tokens being processed in a given invocation of a subgraph are given the unique invocation ID of that subgraph. Their iteration ID is set to zero. When the token reaches the end of the loop and is being fed back into the top of the loop, a special control operator increments the iteration ID. Whenever a token finally leaves the loop, another control operator sets its iteration

ID back to zero.

A hardware architecture based on the dynamic model is necessarily more complex than the static architecture . Additional units are required to form tokens and match tags. More memory is also required to store the extra tokens that will build up on the arcs. The key advantage of the tagged-token model is that it can take full advantage of pipelining effects and can even execute separate loop iterations simultaneously. It can also execute out-of-order, bypassing any tokens that require complex execution and that delay the rest of the computation. It has been shown that this model offers the maximum possible parallelism in any dataflow interpreter.

- Each loop iteration or subprogram invocation should be able to execute in parallel as a separate instance of a reentrant subgraph.
- The replication is only conceptual.
- Each *token* has a *tag*:
  - address of the instruction for which the particular **data** value is destined
  - and context information
- Each arc can be viewed as a bag that may contain an arbitrary number of tokens with different tags.
- The *enabling and firing rule* is now:

A node is enabled and fired as soon as tokens with identical tags are present on all input arcs.

Advantages and Deficiencies of **Dynamic** Dataflow

Dynamically tagged data flow model suggest the maximum parallelism can be exploited from the program graph,

- Major advantage of the dynamic data flow computers is its better performance as compared with static data flow computer as this architecture allows existence of multiple tokens on each arc which thereby lead to unfold iterative program leading to more parallelism.

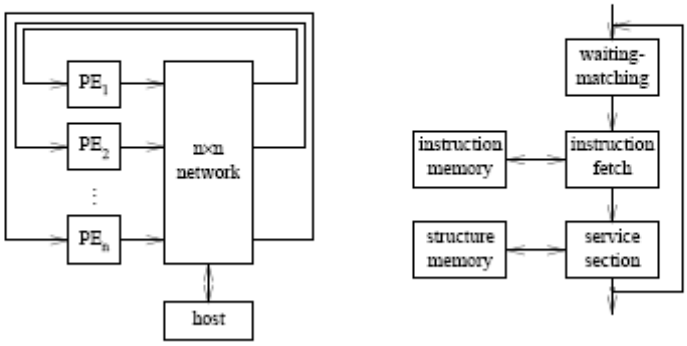
Deficiencies of dynamic dataflow computers

- efficient implementation of the *matching unit* that collects tokens with matching tags.

- *Associative memory* would be ideal.
- Unfortunately, it is not cost-effective since the amount of memory needed to store tokens waiting for a match tends to be very large.
- As a result, all existing machines use some form of *hashing* techniques that are typically not as fast as associative memory.

- bad single thread performance (when not enough workload is present)
- dyadic instructions lead to pipeline bubbles when first operand tokens arrive
- no instruction locality no use of registers

MIT Dynamic Data-Flow Architecture



The main disadvantage of the tagged token model is the extra overhead required to match tags on tokens, instead of simply their presence or absence. More memory is also required and, due to the quantity of data being stored, an associative memory is not practical. Thus, memory access is not as fast as it could be . Nevertheless, the tagged-token model does seem to offer advantages over the static model. A number of computers using this model have been built and studied.

Case study of Dynamic Data Flow Computers

Three dynamic data flow projects are introduced below. In dynamic machines, data tokens are tagged (labeled or colored) to allow multiple tokens to appear simultaneously on any input are of an operator node. No control tokens are needed to acknowledge the transfer of data tokens among instructions. Instead, the matching of token tags (labes or colors) is performed to merge them for instructions requiring more than one operand token. Therefore, additional hardware is needed to attach tags onto data tokens and to

perform tag matching. We shall present the Arvind machine. These machine was designed with following objectives:

1) Modularity: The machine should be constructed from
only a few different component types, regularly interconnected, but internally these components will probably be quite complex (e.g., a processor).

2) Reliability and Fault- Tolerance: Components should be pooled, so removal of a failed component may lower speed and capacity but not the ability to complete a computation.

The development of the Irvine data flow machine was motivated by the desire to exploit the potential of VLSI and to provide a high-level, highly concurrent program organization. This project originated at the University of California at Irvine and now continues at the Massachusetts Institute of Technology by Arvind and his associates. The architectecture of the original Irvine machine is conceptually shown in Figure 10.15. The ID programming language was developed for this machine. This machine has not been built; but extensive simulation studies have been performed on its projected performance.

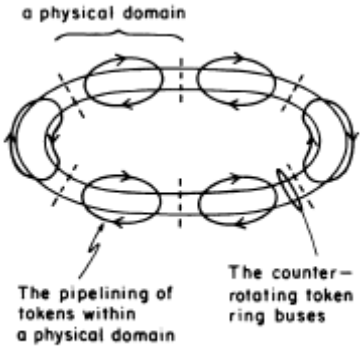


Fig. 7. Physical domains operating concurrently.

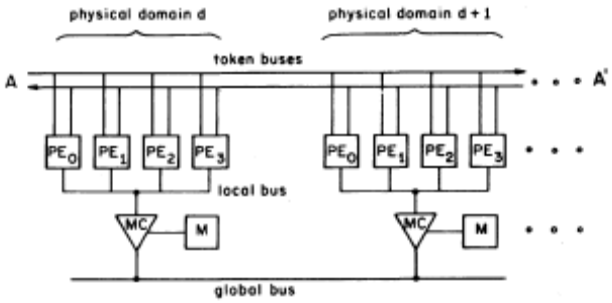


Fig. 5. A ring domain.

The Irvine machine was proposed to consist of multiple PE clusters. All PE clusters (physical domains) can operate concurrently. Here a PE organized as
a pipelined processor. Each box in the figure is a unit that performs work on one item at a time drawn from FIFO input queue(s).

The physical domains are interconnected by two system buses. The token bus is a pair of bidirectional shift-register rings. Each ring is partitioned into as many slots as there are PEs and each slot is either empty or holds one data token. Obviously, the token rings are used to transfer tagged tokens among the PEs.

Each cluster of PEs (four PEs per cluster, as shown in Figure 10.15) shares a local memory through a local bus and a memory controller. A global bus is used to transfer data structures among the local memories. Each PE must accept all tokens that are sent to it and sort those tokens into groups by activity name. When all input tokens for an activity have arrived (through tag matching), the PE must execute that activity. The U-interpreter can help implement interative or procedure computation by mapping the loop or procedure instances into the PE clusters for parallel executions

The Arvind machine at MIT is modified from the Irvine machine, but still based on the ID Language. Instead of using token rings, the Arvind machine has chosen to use an N x N packet switch network for inter-PE communications as demonstrated in Figure 10.16a. The machine consists of N PEs, where each PE is a complete computer with an instruction set, a memory, tag-matching hardware, etc. Activities are divided among the PEs according to a mapping from tags to PE numbers. Each PE uses a statistically chosen assignment function to determine the destination PE number.

5.4 Keywords

context switching Saving the state of one process and replacing it with that of another that is *time sharing* the same processor. If little time is required to switch contexts, *processor overloading* can be an effective way to hide *latency* in a *message passing system*

data flow graph (1) machine language for a data flow computer; (2) result of data flow analysis.

dataflow A model of parallel computing in which programs are represented as *dependence graphs* and each operation is automatically *blocked* until the values on which

it depends are available. The parallel functional and parallel logic programming models are very similar to the dataflow model.

thread a lightweight or small granularity process.

5.5 Summary

The ***Multithreading*** paradigm has become more popular as efforts to further exploit instruction level parallelism have stalled since the late-1990s. This allowed the concept of *Throughput Computing* to re-emerge to prominence from the more specialized field of transaction processing:

- Even though it is very difficult to further speed up a single thread or single program, most computer systems are actually multi-tasking among multiple threads or programs.
- Techniques that would allow speed up of the overall system throughput of all tasks would be a meaningful performance gain.

The two major techniques for *throughput computing* are multiprocessing and multithreading.

Advantages :

- If a thread gets a lot of cache misses, the other thread(s) can continue, taking advantage of the unused computing resources, which thus can lead to faster overall execution, as these resources would have been idle if only a single thread was executed.
- If a thread can not use all the computing resources of the CPU (because instructions depend on each other's result), running another thread permits to not leave these idle.
- If several threads work on the same set of data, they can actually share their cache, leading to better cache usage or synchronization on its values.

We had studied how multihtreading improves the perfromance of procesor. We also dicussed various techniques by which we can implement multiple contest processors. Dataflow has had a fairly long development time, starting from 1960’s with a few groups studying the technique without it is becoming widespread in commercial use.

In dataflow architecture the flow of computation is not instructions flow driven, like it is in control flow architecture. There is no concept of program counter implemented in this

architecture. Control of computation is realized by data flow. Instruction is executed immediately in condition there are all operands of this instruction presented. When executed, instruction produces output operands, which are input operands for other instructions. The most important drawback was compitability issue. Compitability with existing system inhibit the introduction of a radically different computer system requiring a different style of programming and different programming languages.

5.5 Self assignment questions

1. What is cocnept of thread? How use of multithread can improve the computer performance.

2. What is difference between control flow and data flow computer

3. What are static dataflow computer

4. Explain working of dynamic data flow computer

5.6 reference.

Advance computer architecture by Kai HWang

Self assceesed questions

1.

.

6.0 Objective

In this lesson we will about various types of concurrent processor. To study vector processor how pipelining is implemented in vector processor through the instruction format, functional unit. To provides a general overview of the architecture of a vector computer which includes an introduction to vectors and vector arithmetic, a discussion of performance measurements used to evaluate this type of machine. Various models for memory organization for the vector processor are also discussed. We will also study about multiple instruction issue machine which include VLIW, EPIC etc .

6.1 Introduction

The Concurrent Processors must be able to execute multiple instructions at the same time. Concurrent processors must be able to make simultaneous accesses to memory and to simultaneously execute multiple operations. Concurrent processors depend on sophisticated compilers to detect various types of instruction level parallelism that exist within a program. They are classified as

- Vector processors
- SIMD and small clustered MIMD
- Multiple instruction issue machines

    Superscalar (run time schedule)

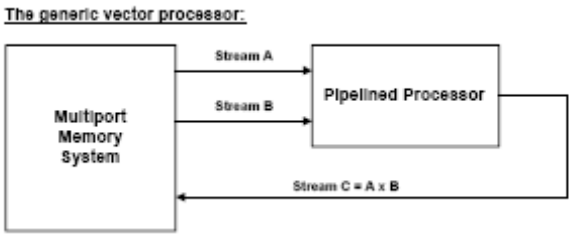
    VLIW (compile time schedule)

    EPIC

    Hybrids

A Vector processor is a processor that can operate on an entire vector in one instruction. The operands to the instructions are complete vectors instead of one element.

Vector processors reduce the fetch and decode bandwidth as the numbers of instructions fetched are less.



They also exploit data parallelism in large scientific and multimedia applications. Based on how the operands are fetched, vector processors can be divided into two categories - in memory-memory architecture operands are directly streamed to the functional units from the memory and results are written back to memory as the vector operation proceeds. In vector-register architecture, operands are read into vector registers from which they are fed to the functional units and results of operations are written to vector registers.

Many performance optimization schemes are used in vector processors. Memory banks are used to reduce load/store latency. Strip mining is used to generate code so that vector operation is possible for vector operands whose size is less than or greater than the size of vector registers.

Various techniques are used for fast accessing these include

— Vector chaining - the equivalent of forwarding in vector processors - is used in case of data dependency among vector instructions.

— Special scatter and gather instructions are provided to efficiently operate on sparse matrices.

Instruction set has been designed with the property that all vector arithmetic instructions only allow element N of one vector register to take part in operations with element N from other vector registers. This dramatically simplifies the construction of a highly parallel vector unit, which can be structured as multiple parallel lanes. As with a traffic highway, we can increase the peak throughput of a vector unit by adding more lanes. Adding multiple lanes is a popular technique to improve vector performance as it requires little increase in control complexity and does not require changes to existing machine code. The reason behind the declining popularity of vector processors is their cost as compared to multiprocessors and superscalar processors. The reasons behind high cost of vector processors are

• Vector processors do not use commodity parts. Since they sell very few copies, design cost dominates overall cost.

• Vector processors need high speed on-chip memories which are expensive.

• It is difficult to package the processors with such high speed. In the past, vector manufactures have employed expensive designs for this.

• There have been few architectural innovations compared to superscalar processors to improve performance keeping the cost low.

Vector processing has the following semantic advantages.

• Programs size is small as it requires less number of instructions. Vector instructions also hide many branches by executing a loop in one instruction.

• Vector memory access has no wastage like cache access. Every data item requested by the processor is actually used.

• Once a vector instruction starts operating, only the functional unit(FU) and the register buses feeding it need to be powered. Fetch unit, de-code unit, ROB etc can be powered off. This reduces the power usage.

6.2 Vector processor

The vector computer or **vector processor** is a machine designed to efficiently handle arithmetic operations on elements of arrays, called *vectors*. Such machines are especially useful in high-performance scientific computing, where matrix and vector arithmetic are quite common. The Cray Y-MP and the Convex C3880 are two examples of vector processors used today.

Vectors and vector arithmetic

A *vector*, *v*, is a list of elements

v = (v1, v2, v3, ..., vn),

transposed. The *length* of a vector is defined as the number of elements in that vector; so the length of *v* is *n*. As far as a vector to a computer program, we declare it as an 1-D array. In Fortran, we declare *v* by the statement

```
DIMENSION  V(N)
```

where N is an integer variable holding the value of the length of the vector.

Arithmetic operations may be performed on vectors. Two vectors are added by adding corresponding elements:

s = x + y = (x1+y1, x2+y2, ..., xn+yn).

In Fortran, vector addition could be performed by the following code

```
DO  I=1,N
  S(I) = X(I) + Y(I)
ENDDO
```

where *s* is the vector representing the final sum and S, X, and Y have been declared as arrays of dimension N. This operation is sometimes called *elementwise* addition. Similarly, the subtraction of two vectors, *x – y*, is an elementwise operation.

The stages of a floating-point operation

Consider the steps or stages involved in a floating-point addition on a sequential machine with IEEE arithmetic hardware: *s = x + y*.

- [A:] The exponents of the two floating-point numbers to be added are compared to find the number with the smallest magnitude.
- [B:] The significand of the number with the smaller magnitude is shifted so that the exponents of the two numbers agree.
- [C:] The significands are added.
- [D:] The result of the addition is normalized.
- [E:] Checks are made to see if any floating-point exceptions occurred during the addition, such as overflow.
- [F:] Rounding occurs.

Step	**A**	**B**	**C**	**D**	**E**	**F**
x	0.1234E4	0.12340E4				
y	-0.5678E3	-0.05678E4				
s			0.066620E4	0.66620E3	0.66620E3	0.6662E3

Figure 6.2 : An example showing the stages of a floating-point addition: $s = x + y$.

Figure 6.2 shows the step-by-step example of such an addition. The numbers to be added are `x = 1234.00` and `y = -567.8`. Now consider this scalar addition performed on all the elements of a pair of vectors (arrays) of length `n`. Each of the six stages needs to be executed for every pair of elements. If each stage of the execution takes `t` units of time, then each addition takes `6*t` units of time (not counting the time required to fetch and decode the instruction itself or to fetch the two operands). So the number of time units required to add all the elements of the two vectors in a serial fashion would be `Ts = 6*n*t`. These execution stages are shown in figure 6.3 with respect to time.

Time:	***t***	**2 *t***	**3 *t***	**4 *t***	**5 *t***	**6 *t***	**7 *t***	**8 *t***
Step								
A	*x1* + *y1*						*x2* + *y2*	

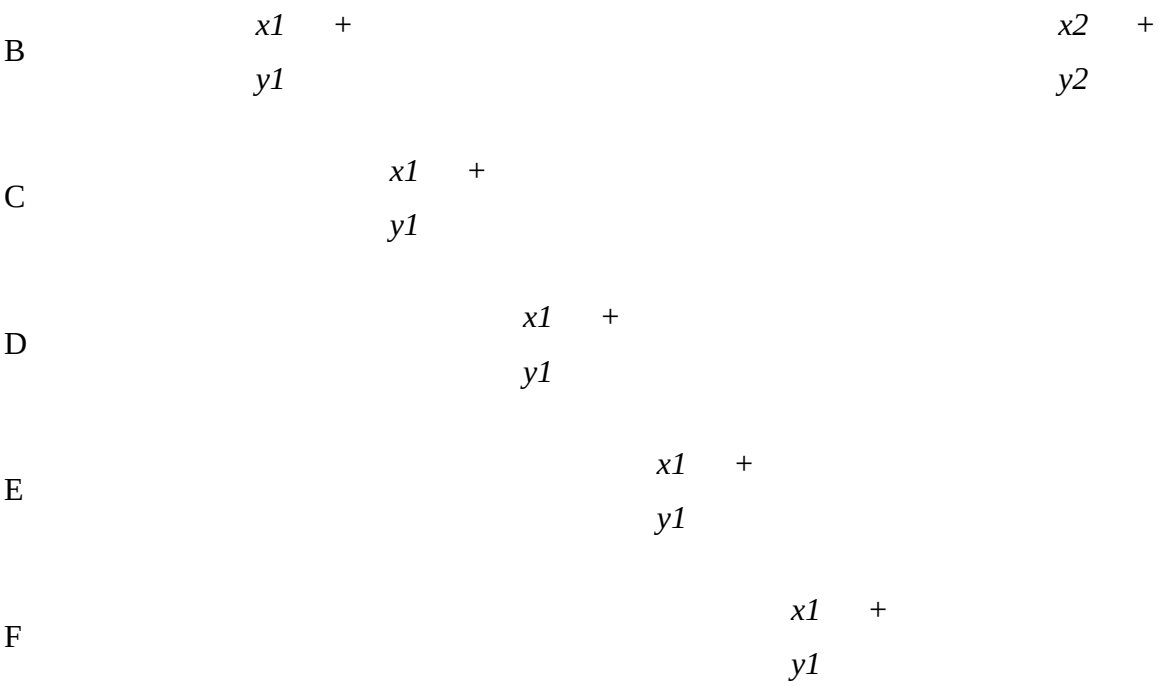
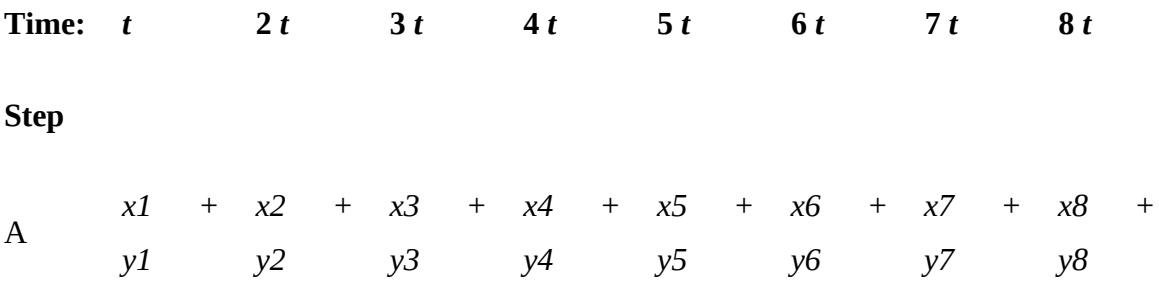


Figure 6.3 : Scalar floating-point addition of vector elements.

An arithmetic pipeline

Suppose the addition operation described in the last subsection is pipelined; that is, one of the six stages of the addition for a pair of elements is performed at each stage in the pipeline. Each stage of the pipeline has a separate arithmetic unit designed for the operation to be performed at that stage. Once stage A has been completed for the first pair of elements, these elements can be moved to the next stage (B) while the second pair of elements moves into the first stage (A). Again each stage takes t units of time. Thus, the flow through the pipeline can be viewed as shown in figure 6.4



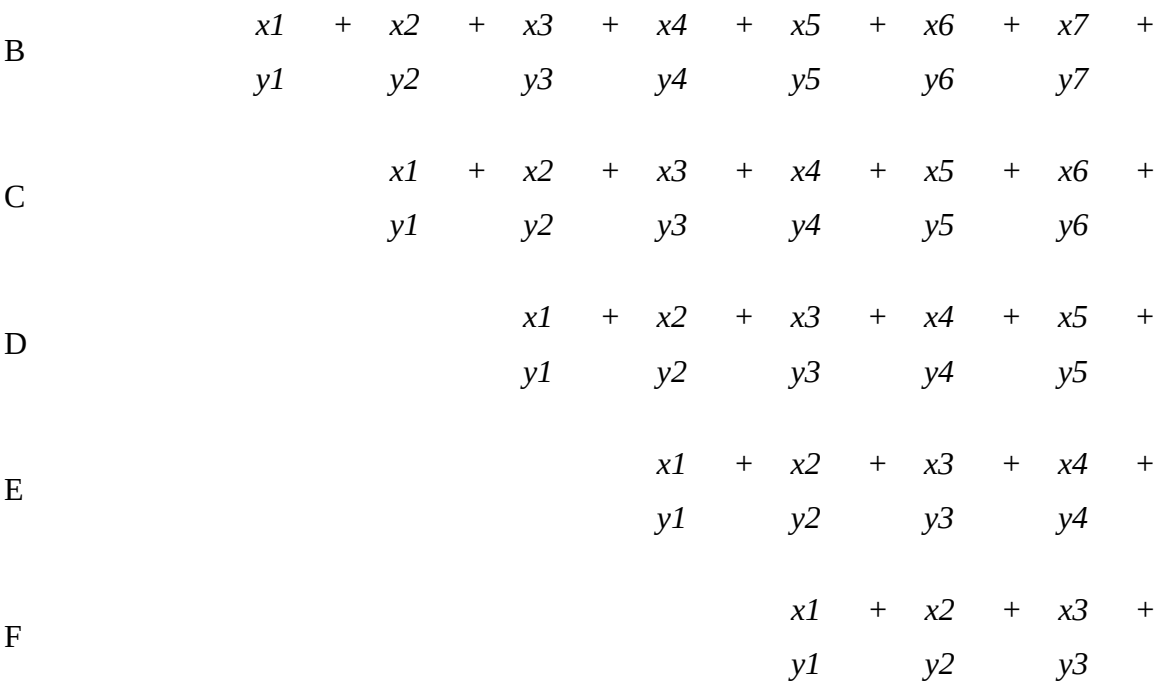


Figure 6.4: Pipelined floating-point addition of vector elements.

Observe that it still takes `6*t` units of time to complete the sum of the first pair of elements, but that the sum of the next pair is ready in only `t` more units of time. And this pattern continues for each succeeding pair. This means that the time, `Tp`, to do the pipelined addition of two vectors of length `n` is

`Tp = 6*t + (n-1)*t = (n + 5)*t`.

The first `6*t` units of time are required to *fill the pipeline* and to obtain the first result. After the last result, `xn + yn`, is completed, the pipeline is emptied out or *flushed*.

Comparing the equations for `Ts` and `Tp`, it is clear that

`(n + 5)*t < 6*n*t`, for `n > 1`.

Thus, this pipelined version of addition is faster than the serial version by almost a factor of the number of stages in the pipeline. This is an example of what makes vector processing more efficient than scalar processing. For large `n`, the pipelined addition for this sample pipeline is about six times faster than scalar addition.

6.2.1 Vector Functional unit

Vector Processing Requirements

A vector operand contains an ordered set of n elements, where n is called the length of the vector. Each element in a vector is a scalar quantity, which may be a floating point number, an integer, a logical value or a character. A vector processor consists of a scalar processor and a vector unit, which could be thought of as an independent functional unit capable of efficient vector operations.

Vector Hardware

Vector computers have hardware to perform the vector operations efficiently. Operands can not be used directly from memory but rather are loaded into registers and are put back in registers after the operation. Vector hardware has the special ability to overlap or pipeline operand processing.

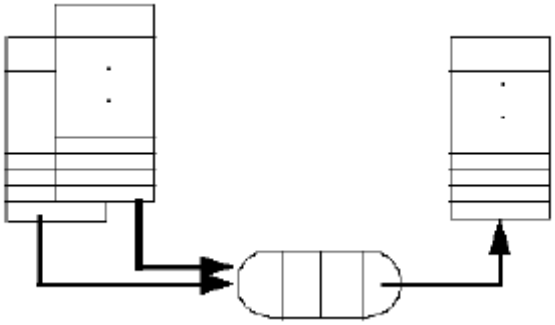


Figure 6.5 Vector Hardware

Vector functional units pipelined, fully segmented each stage of the pipeline performs a step of the function on different operand(s) once pipeline is full, a new result is produced each clock period (cp).

Pipelining

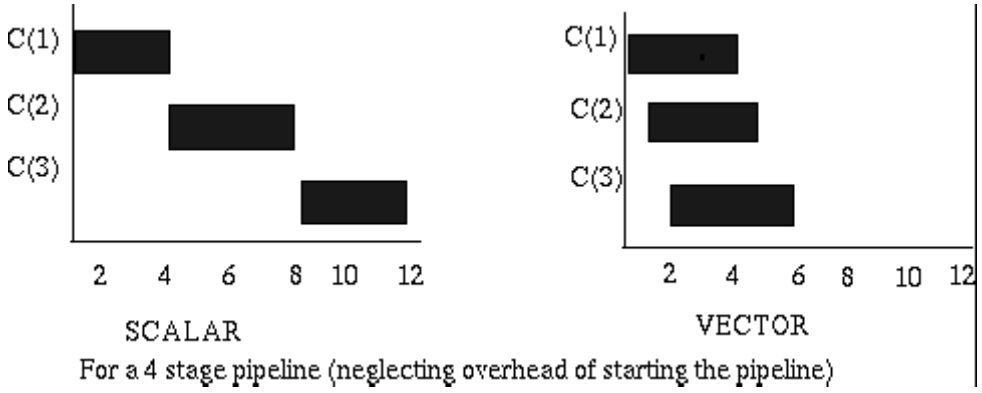
The pipeline is divided up into individual segments, each of which is completely independent and involves no hardware sharing. This means that the machine can be working on separate operands at the same time. This ability enables it to produce one result per clock period as soon as the pipeline is full. The same instruction is obeyed repeatedly using the pipeline technique so the vector processor processes all the elements of a vector in exactly the same way. The pipeline segments arithmetic operation such as floating point multiply into stages passing the output of one stage to the next stage as input. The next pair of operands may enter the pipeline after the first stage has processed

the previous pair of operands. The processing of a number of operands may be carried out simultaneously.

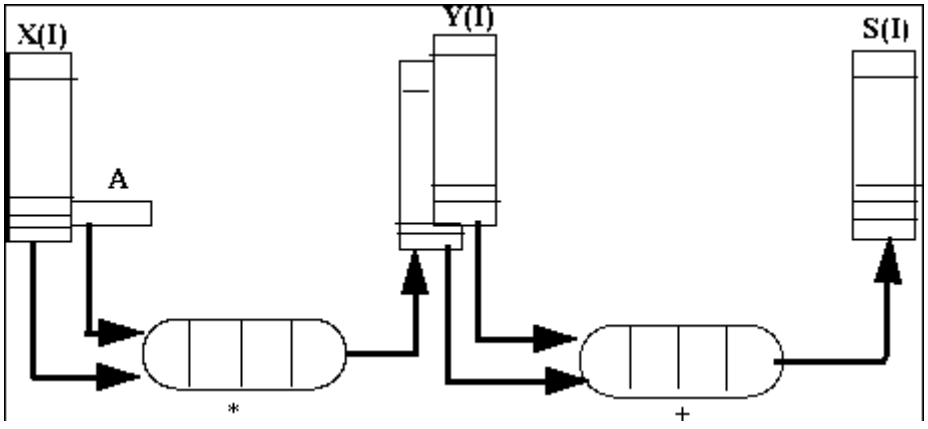
The loading of a vector register is itself a pipelined operation, with the ability to load one element each clock period after some initial startup overhead.

Chaining

Theoretical speedup depends on the number of segments in the pipeline so there is a direct relationship between the number of stages in the pipeline you can keep full and the performance of the code. The size of the pipeline can be increased by chaining thus the Cray combines more than one pipeline to increase its effective size. Chaining means that the result from a pipeline can be used as an operand in a second pipeline as illustrated in the next diagram



For a 4 stage pipeline (neglecting overhead of starting the pipeline)



S(I) = A * X(I) + Y(I)

Figure Pipeline Chaining

This example shows how two pipelines can be chained together to form an effectively single pipeline containing more segments. The output from the first segment is fed directly into the second set of segments thus giving a resultant effective pipeline length of 8. Speedup (over scalar code) is dependent on the number of stages in the pipeline. Chaining increases the number of stages

Most vector architectures have more than one pipeline; they may also contain different types of pipelines. Some vector architectures provide greater efficiency by allowing the output of one pipeline to be *chained* directly into another pipeline. This feature is called *chaining* and eliminates the need to store the result of the first pipeline before sending it into the second pipeline. Figure 14.5 demonstrates the use of chaining in the computation of a *saxpy* vector operation:

a*x + y,

where *x* and *y* are vectors and **a** is a scalar constant.

Vector Chaining used to compute a*x + y

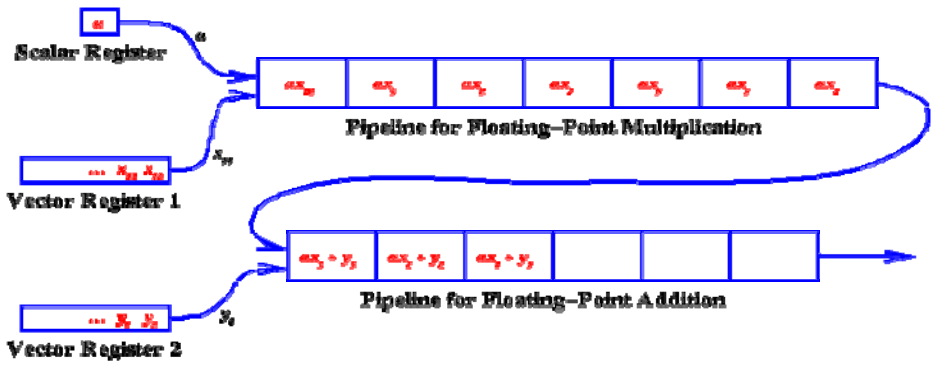


Figure 6.9 Vector chaining used to compute a scalar value **a** times a vector *x*, adding the elements the resultant vector to the elements of a second vector *y* (of the same length).

Chaining can double the number of floating-point operations that are done in x units of time. Once both the multiplication and addition pipelines have been filled, one floating-point multiplication and one floating-point addition (a total of two floating-point operations) are completed every x time units. Conceptually, it is possible to chain more than two functional units together, providing an even greater speedup. However this is rarely (if ever) done due to difficult timing problems.

6.2.2 Vector instruction /operation

Vector Instructions

The ISA of a scalar processor is augmented with vector instructions of the following types:

Vector-vector instructions:

f1: Vi -> Vj (e.g. MOVE Va, Vb)

f2: Vj x Vk -> Vi (e.g. ADD Va, Vb, Vc)

Vector-scalar instructions:

f3: s x Vi -> Vj (e.g. ADD R1, Va, Vb)

Vector-memory instructions:

f4: M -> V (e.g. Vector Load)

f5: V -> M (e.g. Vector Store)

Vector reduction instructions:

f6: V -> s (e.g. ADD V, s)

f7: Vi x Vj ->s (e.g. DOT Va, Vb, s)

Scatter and gather operations

Sometimes, only certain elements of a vector are needed in a computation. Most vector processors are equipped to pick out the appropriate elements (a *gather* operation) and put them together into a vector or a vector register. If the elements to be used are in a regularly-spaced pattern, the spacing between the elements to be gathered is called the *stride*. For example, if the elements

```
x1, x5, x9, x13, ..., x[4*floor((n-1)/4)+1]
```

are to be extracted from the vector

```
( x1, x2, x3, x4, x5, x6, ..., xn )
```

for some vector operation, we say the stride is equal to `4`. A *scatter* operation reformats the output vector so that the elements are spaced correctly. Scatter and gather operations may also be used with irregularly-spaced data.

f8: M x Va -> Vb (e.g. gather)

f9: Va x Vb -> M (e.g. scatter)

Gather and scatter are used to process sparse matrices/vectors. The gather operation, uses a base address and a set of indices to access from memory "few" of the elements of a

large vector into one of the vector registers. The scatter operation does the opposite. The masking operations allows conditional execution of an instruction based on a "masking" register.

Masking instructions:

fa: Va x Vm->Vb (e.g. MMOVE V1, V2, V3)

Gather and scatter are used to process sparse matrices/vectors. The gather operation, uses a base address and a set of indices to access from memory "few" of the elements of a large vector into one of the vector registers. The scatter operation does the opposite. The masking operation allows conditional execution of an instruction based on a "masking" register.

- A Boolean vector can be generated as a result of comparing two vectors, and can be used as a masking vector for enabling and disabling component operations in a vector instruction.
- A compress instruction will shorten a vector under the control of a masking of vector.
- A merge instruction combines two vectors under the control of a masking vector.

In general machine operation suitable for pipelining should have the following properties:

- Identical Processes (or functions) are repeatedly invoked many times, each of which can be subdivided into subprocesses (or sub functions)
- Successive Operands are fed through the pipeline segments and require as few buffers and local controls as possible.
- Operations executed by distinct pipelines should be able to share expensive resources, such as memories and buses in the system.
- The **operation code** must be specified in order to select the functional unit or to reconfigure a multifunctional unit to perform the specified operation.
- For a memory reference instruction, the **base addresses** are needed for both source operands and result vectors. If the operands and results are located in the vector register file, the designated vector registers must be specified.
- The **address increment** between the elements must be specified.
- The **address offset** relative to the base address should be specified. Using the base address and the offset the relative effective address can be calculated.

- The **Vector length** is needed to determine the termination of a vector instruction.
- The Relative Vector/Scalar Performance and Amdahl Law

The major hurdle for designing a vector unit is to ensure that the flow of data from memory to the vector unit will not pose a bottleneck. In particular, for a vector unit to be effective, the memory must be able to deliver one datum per clock cycle. This is usually achieved using pipelining using the C-access memory organization (concurrent access) or the S-access memory organization (simultaneous access), or a combination thereof.

Vector-register vector processors

If a vector processor contains vector registers, the elements of the vector are read from memory directly into the vector register by a *load vector* operation. The vector result of a vector operation is put into a vector register before it is stored back in memory by a *store vector* operation; this permits it to be used in another computation without needing to be reread, and it allows the store to be overlapped by other operations. On these machines, all arithmetic or logical vector operations are register-register operations; that is, they are only performed on vectors that are already in the vector registers. For this reason, these machines are called *vector-register* vector processors.

Memory-memory vector processors

Another type of vector processor allows the vector operands to be fetched directly from memory to the different vector pipelines and the results to be written directly to memory; these are called *memory-memory* vector processors. Because the elements of the vector need to come from memory instead of a register, it takes a little longer to get a vector operation started; this is due partly to the cost of a memory access. One example of a *memory-memory* vector processor is the CDC Cyber 205.

Because of the ability to overlap memory accesses and the possible reuse of vector processors, vector-register vector processors are usually more efficient than memory-memory vector processors. However as the length of the vectors in a computation increase, this difference in efficiency between the two types of architectures is diminished. In fact, the memory-memory vector processors may prove more efficient if the vectors are long enough. Nevertheless, experience has shown that shorter vectors are more commonly used.

Comparison - Vector and Scalar Operations

A scalar operation works on only one pair of operands from the S register and returns the result to another S register whereas a vector operation can work on 64 pairs of operands together to produce 64 results executing only one instruction. Computational efficiency is achieved by processing each element of a vector identically eg initializing all the elements of a vector to zero.

A vector instruction provides iterative processing of successive vector register elements by obtaining the operands from the first element of one or more V registers and delivering the result to another V register. Successive operand pairs are transmitted to a functional unit in each clock period so that the first result emerges after the start up time of the functional unit and successive results appear each clock cycle.

Vector overhead is larger than scalar overhead, one reason being the vector length which has to be computed to determine how many vector registers are going to be needed (i.e., the number of elements divided by 64).

Each vector register can hold up to 64 words so vectors can only be processed in 64 element segments. This is important when it comes to programming as one situation to be avoided is where the number of elements to be processed exceeds the register capacity by a small amount e.g., a vector length of 65. What happens in this case is that the first 64 elements are processed from one register, the 65th element must then be processed using a separate register, after the first 64 elements have been processed. The functional unit will process this element in a time equal to the start up time instead of one clock cycle hence reducing the computational efficiency.

There is a sharp decrease in performance at each point where the vector length spills over into a new register.

The Cray can receive a result by a vector register and retransmit it as an operand to a subsequent operation in the same clock period. In other words a register may be both a result and an operand register which allows the chaining of two or more vector operations together as seen earlier. In this way two or more results may be produced per clock cycle.

Parallelism is also possible as the functional units can operate concurrently and two or more units may be co-operating at once. This combined with chaining, using the result of one functional unit as the input of another, leads to very high processing speeds.

Scalar and vector processing examples

```
DO 10 I = 1, 3
JJ(I) = KK(I)+LL(I)
10 CONTINUE
```

A generic vector processor

Vector registers

Some vector computers, such as the Cray Y-MP, contain *vector registers*. A general purpose or a floating-point register holds a single value; vector registers contain several elements of a vector at one time. For example, the Cray Y-MP vector registers contain 64 elements while the Cray C90 vector registers hold 128 elements. The contents of these registers may be sent to (or received from) a vector pipeline one element at a time.

Scalar registers

Scalar registers behave like general purpose or floating-point registers; they hold a single value. However, these registers are configured so that they may be used by a vector pipeline; the value in the register is read once every `tau` units of time and put into the pipeline, just as a vector element is released from the vector pipeline. This allows the elements of a vector to be operated on by a scalar. To compute

```
y = 2.5 * x,
```

the `2.5` is stored in a scalar register and fed into the vector multiplication pipeline every `tau` units of time in order to be multiplied by each element of `x` to produce `y`.

6.2.4 Vector computing performance

For typical vector architectures, the value of `tau` (the time to complete one pipeline stage) is equivalent to one clock cycle of the machine On some machines, it may be equal to two or more clock cycles.. Once a pipeline like the one shown in figure 3 has been filled, it generates one result for each `t` units of time, that is, for each clock cycle. This means the hardware performs one floating-point operation per clock cycle.

Let `k` represent the number of `t` time units the same sequential operation would take (or the number of stages in the pipeline). Then the time to execute that sequential operation on a vector of length `n` is

```
Ts = k*n*t,
```

and the time to perform the pipelined version is

$Tp = k*t + (n-1)*t = (n + k - 1)*t$.

Again for $n > 1$, $Ts > Tp$.

A *startup time* is also required; this is the time needed to get the operation going. In a sequential machine, there may some overhead required to set up a loop to repeat the same floating-point operation for an entire vector; the elements of the vector also need to be fetched from memory. If we let Ss be the number of t time units for the sequential startup time, then Ts must include this time:

$Ts = (Ss + k*n)*t$.

In a pipelined machine, the flow from the vector registers or from memory to the pipeline needs to be started; call this time quantity Sp. Another overhead cost, $k*t$ time units, is the time needed to initially fill the pipeline. Hence, Tp must include the startup time for the pipelined operation; thus,

$Tp = (Sp + k)*t + (n - 1)*t$

or

$Tp = (Sp + k + n - 1)*t$.

As the length of the vector gets larger (as n goes to infinity), the startup time becomes negligible in both cases. This means that

$Ts \dashrightarrow k*n*t$

while

$Tp \dashrightarrow n*t$.

Thus, for large n, Ts is k times larger than Tp.

There are a number of other terms to describe the performance of vector processors or vector computers. The following list introduces some of these:

- Rn: For a vector processor, the number of Mflops obtainable for a vector of length n.
- $R\_infinity$: The asymptotic number of Mflops for a given vector computer as the length of the vectors gets large. This means that the startup time would be completely negligible. When the vectors are very long, there should be a result from the pipeline at every tau units of time or every clock cycle. So the number

of floating-point operations that can be completed in one second is *1.0/tau*; dividing this result by one million produces the result in Mflops.

- *n_1/2*: The length, *n*, of a vector such that *Rn* is equal to *R_infinity / 2*. Again for very large vectors, there should be a result from the pipeline at every *tau* units of time. So, *n_1/2* represents the vector length needed to get a result at every *2*tau* units of time or every two clock cycles.
- *n_v*: The length, *n*, of a vector such that performing a vector operation on the *n* elements of that vector is more efficient than executing the *n* scalar operations instead.

Vector Computer Performance

Performance Characteristics	Year	Clock Cycle (nsec)	Peak Perf (Mflops)	R_infinity (x * y) (Mflops)	n_1/2 (x * y)
Cray-1	1976	12.5	160	22	18
CDC Cyber 205	1980	20.0	100	50	86
Cray X-MP	1983	9.5	70	53	
210 ... with 4 Procs	---	---	---	---	
840 Cray-2	1985	4.1	56	83	
488 ... with 4 Procs	---	---	---	---	
1951 IBM 3090	1985	18.5	54	high 20's	
108 ... with 8 Procs	---	---	---	---	
432 ETA 10	1986	10.5	---	---	
1250 ... with 8 Procs	---	---	---	---	
10,000 Alliant FS/8	1986	170.0	1	151	
6 ... with 8 Procs	---	---	47	1	23
Cray C90	1990	4.2	---	---	
952 ... with Procs	---	---	---	650	
			15,238	---	---
Convex C3880	---	---	960		

Performance Characteristics	Year	Clock Cycle (nsec)	Peak Perf (Mflops)	R_infinity (x * y) (Mflops)	n_1/2 (x * y)
Cray 3-128	1993	2.1	948	---	---
... with 4 Procs	---	---	3972	---	---

Table 1: Performance characteristics of vector processing computers using 64-bit floating-point numbers. The expression *(x * y)* refers to the element wise multiplication of two vectors, *x* and *y*

Table 1 provides some performance characteristics for some of the vector computers discussed later in this section. The values of *R_infinity* and *n_1/2* are for the elementwise multiplication of two vectors.

The pipeline vector computers can be divided into 2 architectural configurations according to where the operands are received in a vector processor. They are :

- Memory -to- memory Architecture, in which source operands, intermediate and final results are retrieved directly from the main memory.
- Register-to-register architecture, in which operands and results are retrieved indirectly from the main memory through the use of large number of vector or scalar registers.

Pipelined Vector Processing Methods

Vector computations are often involved in processing large arrays of data. By ordering successive computations in the array, the vector array processing can be classified into three types :

Horizontal Processing, in which vector computations are performed horizontally from left to right in row fashion.

Vertical processing, in which vector computations are carried out vertically from top to bottom in column fashion.

Vector looping, in which segmented vector loop computations are performed from left to right and top to bottom in a combined horizontal and vertical method.

A simple vector summation computation illustrate these vector processing methods

Let { ai for 1 <= i<= n) be n scalar contstants, Xj = (X1j,X2j…… Xmj)T for j = 1,2,3 ….n be n column vectors and Yj = (Y1j,Y2j…… Ym)T be a column vector of m components. The computation to be performed is

Y = ai.x1 + a2.x2 + …. an.xn

Y1 = Z11 + Z12 + …..Z1n

Y2 = Z21 + Z22 + …..Z2n

.

.

. Ym = Zm1 +Zm2+…..Zmn

Horizontal Vector Processing

In this method all components of the vector y are calculated in sequential order, yi for i = 1,2,….m. Each summation involving n-1 additions must be completed before switching to the evaluation the next summation.

Vertical Vector Processing :

The sequence of additions in this method are, compute the partial sum sequentially through the pipeline (in row wise z11+z12…)

Computer the partial sum in the column format repeatedly.

Vector Looping Method:

It combines the horizontal and vertical approaches into a block approach.

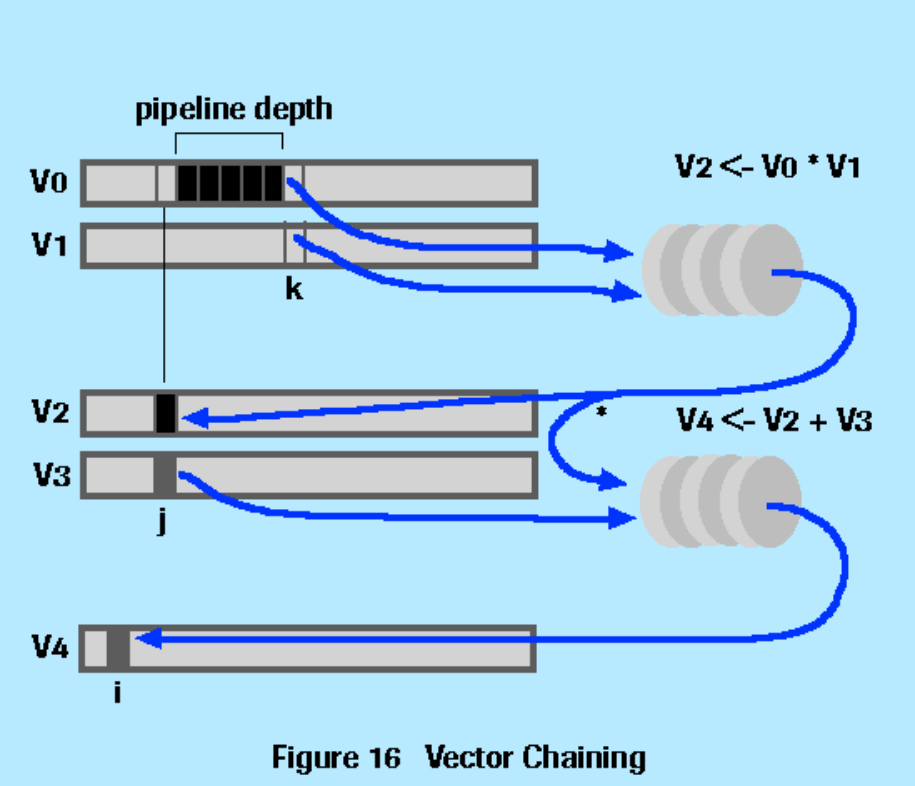


Figure 16 Vector Chaining

The Relative Vector/Scalar Performance and Amdahl Law

Let r be the vector/scalar speed ratio and f be the vectorization ratio. For example, if the time it takes to add a vector of 64 integers using the scalar unit is 10 times the time it takes to do it using the vector unit, then r = 10. Moreover, if the total number of operations in a program is 100 and only 10 of these are scalar (after vectorization), then f=90 (i.e. 90% of the work is done by the vector unit). It follows that the achievable speedup is:

Time without the vector unit

Time with the vector unit

For our example, assuming that it takes one unit of time to execute one scalar operation, this ratio will be:

100x1

10x1 + 90x0.1 = 100/19 (approx 5).

In general, the speedup is:

$$\frac{r}{(1-f)r + f}$$

So even if the performance of the vector unit is extremely high (r = oo) we get a speedup less than 1/(1-f), which suggests that the ratio f is crucial to performace since it poses a limit on the attainable speedup. This ratio depends on the efficiency of the compilation, etc... This also suggests that a scalar unit with a mediocre performance (even if coupled with the fastest vector unit), will yield mediocre speedup.

Strip-mining

If a vector to be processed has a length greater than that of the vector registers, then strip-mining is used, whereby the original vector is divided into equal size segments (equal to the size of the vector registers) and these segments are processed in sequence. The process of strip-mining is usually performed by the compiler but in some architectures (like the Fujitsu VP series) it could be done by the hardware.

Compound Vector Processing

A sequence of vector operation may be bundled into a "compound" vector function (CVF), which could be executed as one operation (without having to store intermediate results in register vectors, etc..) using a technique called chaining, which is an extension of bypassing (used in scalar pipelines). The purpose of "discovering" CVFs is to explore opportunities for concurrent processing of linked vector operations.

Notice that the number of available vector registers and functional units imposes limitations on how many CVFs can be executed simulataneously (e.g. Cray 1 CVP of SAXPY code leads to a speedup of 5/3. The X-MP results in a speadup of 5).

6.3 Vector memory

Interleaved memory banks

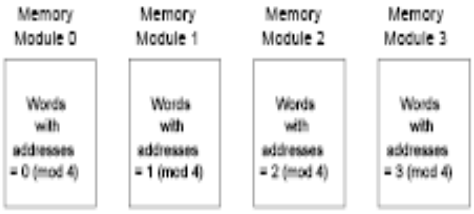
To allow faster access to vector elements stored in memory, the memory of a vector processor is often divided into *memory banks*. *Interleaved* memory banks associate successive memory addresses with successive banks cyclically; thus word 0 is stored in bank 0, word 1 is in bank 1, ..., word $n-1$ is in bank $n-1$, word n is in bank 0, word $n+1$ is in bank 1, ..., etc., where n is the number of memory banks. As with many other computer architectural features, n is usually a power of 2:

$n = 2^k$,

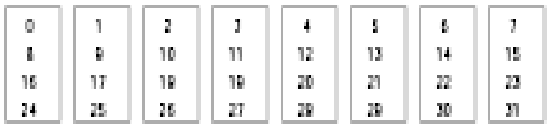
where $k = 1, 2, 3,$ or 4.

One memory access (load or store) of a data value in a memory bank takes several clock cycles to complete. Each memory bank allows only one data value to be read or stored in a single memory access, but more than one memory bank may be accessed at the same time. When the elements of a vector stored in an interleaved memory are read into a vector register, the reads are staggered across the memory banks so that one vector element is read from a bank per clock cycle. If one memory access takes n clock cycles, then n elements of a vector may be fetched at a cost of one memory access; this is n times faster than the same number of memory accesses to a single bank.

The figure below is an interleaved memory as it can be seen it places consecutive words of memory in different memory modules:



Since a read or write to one module can be started before a read/write to another module finishes, reads/writes can be overlapped. Only the leading bits of the address are used to determine the address within the module. The least-significant bits (in the diagram above, the two least-significant bits) determine the memory module. Thus, by loading a single address into the memory-address register (MAR) and saying “read” or “write”, the processor can read/write *M* words of memory. We say that memory is *M*-way interleaved. *Low-order interleaving* distributes the addresses so that consecutive addresses are located within consecutive modules. For example, for 8-way interleaving:



The Low end machine use the interleaved memory

• Memory banks take turns being connect to bus

• Interleaved memory access improves available bandwidth and may reduce latency for concurrent accesses.

High end machine use the multiple concurrent banks

• Might use crossbar switch (instead of bus, not instead of VDS) to connect several memory banks to the VDS simultaneously

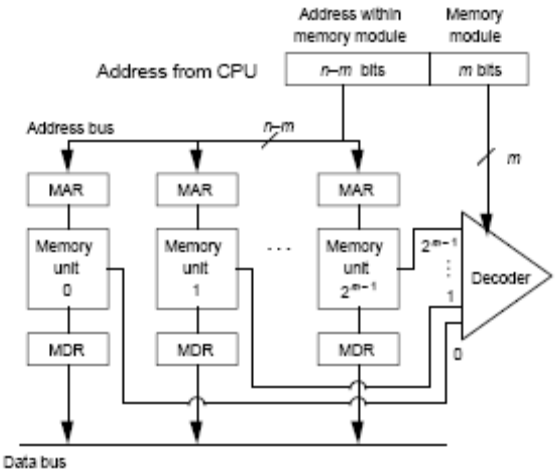
• Might be interleaved and assume different subsets of banks connected each clock

Interleaved-memory designs: Interleaved memory divides an address into two portions: one selects the module, and the other selects an address within the module.

Each module has a separate MAR and a separate MDR.

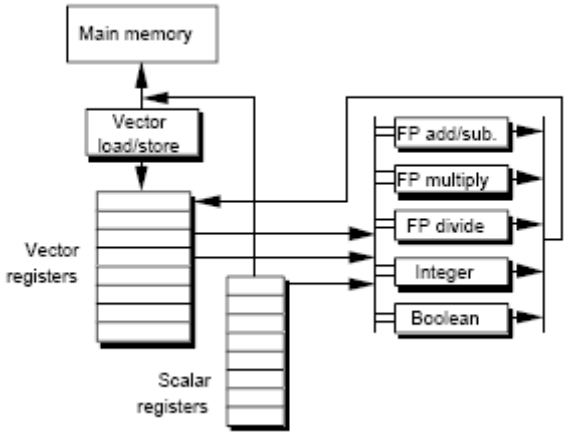
· When an address is presented, a decoder determines which MAR should be loaded with this address. It uses the low-order m — log2M bits to decide this.

· The high-order $n–m$ bits are actually loaded into the MAR. They select the proper location within the module.



An alternative to feeding a vector processor directly from external storage is to provide a hierarchical memory system similar to cache memory. Memory on the processor chip is called ***register storage*** rather than L1 cache, and is managed directly by the programmer rather than automatically by the hardware.

A vector processor with high-speed register storage:



The vector registers are large – 64 to 256 floating point numbers each. 256 floating point numbers at 64 bits each times 8 registers is equivalent to a 16k byte internal data cache.

6.3.1 Vector Memory Modeling

In vector processor when vector operate the parallel execution the memory access can be overlapped with vector execution the problem arise if the memory cannot keep up with vector execution rate.

Gamma (©) **ℜinomial model**

This model request is based on the principal to use vector request buffer to bypass waiting requests. An associated issue is the degree of bypassing or out-of-order requests that a source can make to the memory system. Suppose a conflict arises: a request is directed to a busy module. How many subsequent requests can the source make before it must wait? Assume each of *s* access ports to memory has a buffer of size *T BE / s* (Fig 7.19). This buffer holds requests (element addresses) to memory that are being held due to a conflict. For each source, the degree of bypassing is defined as the allowable number of requests waiting before stalling of subsequent requests occurs.

From a modeling point of view, this is different from the simple binomial or the δ-binomial models. The basis difference is that the queue awaiting service from a module is larger by an amount ϒ, where ϒ is the man queue size of bypassed requests awaiting service. Note that the average queue size (ϒ) is always less than or equal to the buffer size:

$$\gamma \leq TBF / s,$$

Since r cannot exceed the size of the physically implemented buffer. (Although, depending on the organization of the *TBF*, one source buffer could "borrow" from another)

With or without request bypassing, there ids a buffer between the *s* request sources and the *m* memory modules (Figure 7.19). This must be large enough to accommodate denied requests (no bypassing) i.e.:

Buffer = $TBF.\ mQ_c$

Where *Qc* is the expected number of denied requests per module, and *m* is the number of modules. The $m\ .\ Qc = n - B$, as discussed in chapter 6. If we allow bypassing, we will require additional buffer entries and additional control. Typically, an entry could include:

- Request source id.
- Request source tag (i.e., VR number).
- Module id.
- Address for request to a module
- Entry priority id (assuming more than one request can be bypassed).

While some optimization is possible, it is clear that large bypassed request buffers can be complex.

7.3.3 **Gamma(ɣ)-Binomial Model**

We now develop the ɣ -binomial model of bypassed vector memory behavior. Assume that each vector sources issues a request each cycle ($\delta = 1$), and that each physical requestor in the vector processor has the same buffer capacity and characteristic. If the vector processor can make *s* requests per cycle, and there are *t cycles per Tc*, we have:

Total requests per $Tc = t\ .\ s = n.$

This is the same as our *n* requests per *Tc* in the simple binominal model, but the situation in the vector processor is more complex. We assume that each of the sources s makes a request each cycle and *each of its* ɣ *-buffered requests* also makes a request.

Depending on the buffer control, these buffer requests are made only implicitly. The controller "knows" when a target module will be free and therefore schedules the actual request for that time. From a memory modeling point of view, this is equivalent to the buffer requesting service each cycle until the module is free.

Thus, we now have:

Total requests per $Tc = \underline{t \cdot s + t \cdot s \cdot \lambda}$

$= \underline{t \cdot s (1 + \lambda)}$

$= \underline{n(1 + \lambda)}$

Vector computation model not as compelling as it once was

• Multi-issue, latency-tolerant architectures reduce cost of loop overhead

– Instruction concurrency is available, and can substitute for data concurrency

• Improved compiler technology reduces value of programmer using vectors to give hints to hardware

– Improved algorithms to exploit cache

– Smart pre-fetching hardware, cache bypass, latency tolerance

• Commodity networked computing can often achieve comparable performance to a supercomputer

– Single-chip CPUs now have very high clock rates

– Improved infrastructure for parallel computing makes it accessible

But, desktop CPUs can benefit from supercomputer tricks

• Strided prefetching to reduce latency and better use memory bandwidth

• Selective bypassing of cache to avoid cache pollution

• Intel i860 was an experiment in this direction; but it was a poor compiler target

6.4 Multiple issue machines

The alternative to vector processors is multiple-issue machine. There are two broad classes of multiple-issue machines: statically scheduled and dynamically scheduled. In principle, these two classes are quite similar. Dependencies among groups of instructions are evaluated, and groups found to be independent are simultaneously dispatched to multiple execution units. For statically scheduled processors, this detection process is done by the compiler, and Instructions are assembled into *instruction packets*, which are decoded and executed at run time. For dynamically scheduled processors, the detection of independent instructions may also be done at compiler time and the code suitably arranged to optimize execution patterns, but the ultimate selection of instructions (to be executed or dispatched) is done by the hardware in the decoder at run time. In principle, the dynamically scheduled processor may have an instruction representation and form

that is indistinguishable from slower pipeline processors. Statically scheduled processors must have some additional information either implicitly or explicitly indicating instruction packet boundaries.

The extensive use of register ports provides simultaneous access to data as required by a VLIW processor. This suggests the register set as a processor bottleneck. Dynamic multiple-issue processors usually use multiple buses connecting the register set and functional units, and each bus services multiple functional units. This may limit the maximum degree of concurrency, but it can also significantly reduce the required number of register ports.

6.4.1 Very Long Instruction Words

Another approach to the parallelism problem is to exploit instruction level parallelism by having the compiler create bundles of instructions that take advantage of the chip's known functional units. For instance, if the processor is capable of executing 2 ALU operations, 1 load/store operation, and one multiply operation simultaneously, the compiler can do its best to arrange the instructions in such a way that groups consisting of all these elements will be formed. Together, the group will be issued as a very long instruction.

This technique is not as popular as superscalar because of the high dependency on compiler support, and the initial lack thereof. VLIW avoids the chip complexity issues that are present in superscalar, but it is hindered by the fact that if there is no compiler capable of efficiently created very long instructions, the architecture is basically useless. The VLIW technique is probably most useful in certain implementations of high-performance computers where the types of programs that will be executed are known in advance and that extensive compiler support is not needed.

VLIW Machines

As superscalar machines become more complex, the difficulties of scheduling instruction issue become more complex. The on-chip hardware devoted to resolving dependencies and deciding on instruction issue is growing as a proportion of the total. In some ways, the situation is reminiscent of the trend towards more complex CISC processors - eventually leading to the radical change to RISC machines.

Another way of looking at superscalar machines is as dynamic instruction schedulers - the hardware decides on the fly which instructions to execute in parallel, out of order, etc. An alternative approach would be to get the compiler to do it beforehand - that is, to *statically* schedule execution. This is the basic concept behind *Very Long Instruction Word*, or VLIW machines.

VLIW machines have, as you may guess, very long instruction words - in which a number of 'traditional' instructions can be packed. (Actually for more recent examples, this is arguably not really true but it's a convenient mental model for now.) For example, suppose we have a processor which has two integer operation units; a floating point unit; a load/store unit; and a branch unit. An 'instruction' for such a machine would consist of [up to] two integer operations, a floating point operation, a load or store, and a branch. It is the compilers responsibility to find the appropriate operations, and pack them together into a very long instruction - which the hardware can execute simultaneously without worrying about dependencies (because the compiler has already considered them).

Pros and Cons

VLIW has both advantages and disadvantages. The main advantage is the saving in hardware - the compiler now decides what can be executed in parallel, and the hardware just does it. There is no need to check for dependencies or decide on scheduling - the compiler has already resolved these issues. (Actually, as we shall see, this may not be entirely true either.) This means that much more hardware can be devoted to useful computation, bigger on-chip caches etc., meaning faster processors.

Not surprisingly, there are also disadvantages.

- **Compilers.** First, obviously compilers will be harder to build. In fact, to get the best out of current, dynamically scheduled superscalar processors it is necessary for compilers to do a fair bit of code rearranging to 'second guess' the hardware, so this technology is already developing. It is observed that building good compilers for VLIW is non-trival.
- **Code Bigger.** Secondly, programs will get bigger. If there are not enough instructions that can be done in parallel to fill all the available slots in an instruction (which will be the case most of the time). There will consequently be empty slots in instructions. It is likely that the majority of instructions, in typical

applications, will have empty code slots, meaning wasted space and bigger code. (It may well be the case that to ensure that all scheduling problems are resolved at compiler time, we will need to put in some *completely empty* instructions.) Memory and disk space is cheap - however, memory bandwidth is not. Even with the large and efficient caches, we would prefer not to have to fetch large, half-empty instructions.

- **One Stalls, all Stall.** Unfortunately, it is not possible at compile time to identify all possible sources of pipeline stalls and their durations. For example, suppose a memory access causes a *cache miss*, leading to a longer than expected stall. If other, parallel, functional units are allowed to continue operating, sources of data dependency may *dynamically* emerge. For example, consider two operations which have an output dependency. The original scheduling by the compiler would ensure that there is no consequent WAW hazard. However, if one stalls and the other 'runs ahead', the dependency may turn into a WAW hazard. In order to get the compiler to do *all* dependency resolution, it is required to stall *all* pipeline elements together. This is another performance problem.
- **Hardware Shows Through** A significant issue is the break in the barrier between architecture and implementation which has existed since the IBM 360 in the early/mid 60s. It will be necessary for compilers to know exactly what the capabilities of the processor are - for example, how many functional units are there?
- VLIW instruction sets are not backward compatible between implementations. As wider implementations (more execution units) are built, the instruction set for the wider machines is not backward compatible with older, narrower implementations.
- Load responses from a memory hierarchy which includes CPU caches and DRAM do not give a deterministic delay of when the load response returns to the processor. This makes static scheduling of load instructions by the compiler very difficult.

6.4.2 Moving beyond VLIW

EPIC architectures add several features to get around the deficiencies of VLIW:

- Each group of multiple software instructions is called a *bundle*. Each of the bundles has information indicating if this set of operations is depended upon by the subsequent bundle. With this capability, future implementations can be built to issue multiple bundles in parallel. The dependency information is calculated by the compiler, so the hardware does not have to perform operand dependency checking.
- A *speculative* load instruction is used as a type of data prefetch. This prefetch increases the chances for a primary cache hit for normal loads.
- A check load instruction also aids speculative loads by checking that a load was not dependent on a previous store.

The *EPIC* architecture also includes a *grab-bag* of architectural concepts to increase *ILP*:

- Predicated execution is used to decrease the occurrence of branches and to increase the speculative execution of instructions. In this feature, branch conditions are converted to predicate registers which are used to kill results of executed instructions from the side of the branch which is not taken.
- Delayed exceptions (using a Not-A-Thing bit within the general purpose registers) also allow more speculative execution past possible exceptions.
- Very large architectural register files avoid the need for register renaming.
- Multi-way branch instructions

The IA-64 architecture also added register_rotation - a digital signal processing concept useful for loop unrolling and software pipelining.

6.5 Summary

Vector supercomputers are not viable due to cost reason, but vector instruction set architecture is still useful. Vector supercomputers are adapting commodity technology like SMT to improve their price-performance. Superscalar microprocessor designs have begun to absorb some of the techniques made popular in earlier vector computer systems (Ex - Intel MMX extension). Vector processors are useful for embedded and multimedia applications which require low power, small code size and high performance.

Vector Processor vs Multiple Issue processor

Advantage of Vector Processor

— good Sp on large scientific problems

— mature compiler technology.

Disadvantage of **Vector Processor**

— limited to regular data and control structures

— Vector Registers and buffers

— memory BW

Advantage of multiple issue processor

— general-purpose

— good Sp on small problems

— developing compiler technology

Advantage of multiple issue processor

— instruction decoder H/W

— large D cache

— inefficient use of multiple ALUs

6.6 Keywords

vector an ordered list of items in a computer's memory. A simple vector is defined as having a starting address, a length, and a stride. An indirect address vector is defined as having a relative base address and a vector of values to be applied as indices to the base.

vector processor A computer designed to apply arithmetic operations to long vectors or arrays. Most vector processors rely heavily on *pipelining* to achieve high performance.

vector register a storage device that acts as an intermediate memory between a computer's functional units and main memory

interleaved memory memory divide into a number of modules or banks that can be accessed simultaneously.

VLIW Very Long Instruction Word; the use of extremely long instructions (256 bits or more) in a computer to improve its ability to chain operations together.

6.7 Self assessment Question

1. What are vector?
2. Why vector processors popular in scientific calculations
3. Drawback of vector processor
4. Drawback of VILW processor
5. Write problems in implementing VILW processor?

6.8 Reference:

Advance computer architecture by Kai Hwang

Computer Architecture by Michael J. Flynn