# ACA (15CS72) MODULE-4

5.0 Objective

5.1 Introduction

5.2 Multithreading

5..2.1 multiple context processor

5.2.2 multidimensional processor

5.3 Data flow architecture

5.3.1Data flow graph

5.3.2 Static dataflow

5.3.3 Dynamic dataflow

5.4 Self assignment questions

5.5 Reference.

## 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.

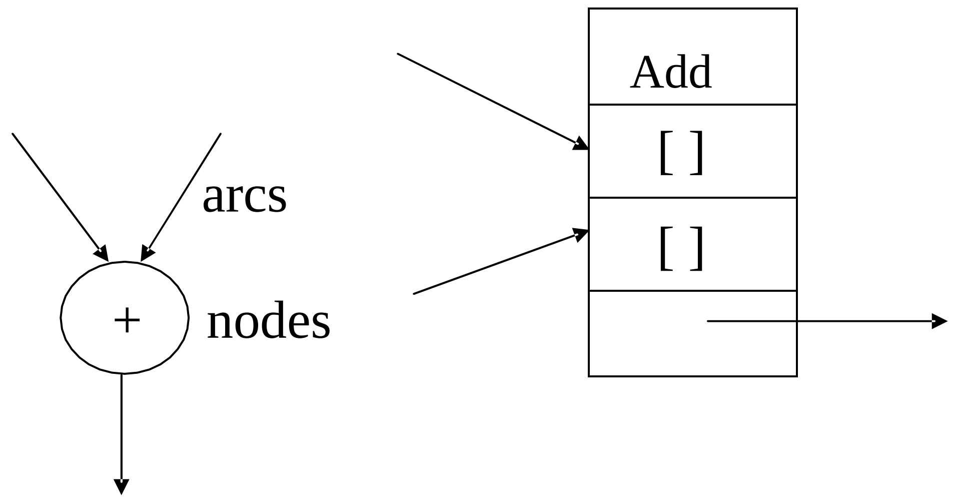
The data driven concept means asynchrony which means that many instructions can be executed simultaneously no PC and global updateable store is required. Information required in a data flow computer are operation packets that are composed of opcode, operand and destinations of its successor instructions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

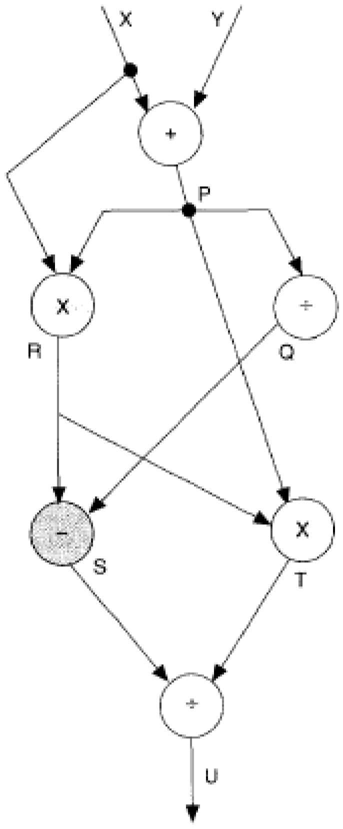
(1,2.3.4,5,6)

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

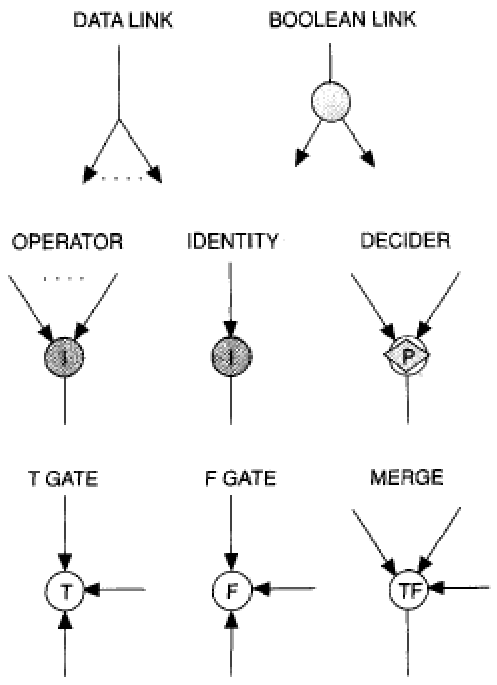
(1,3,5,2,4,6)

(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



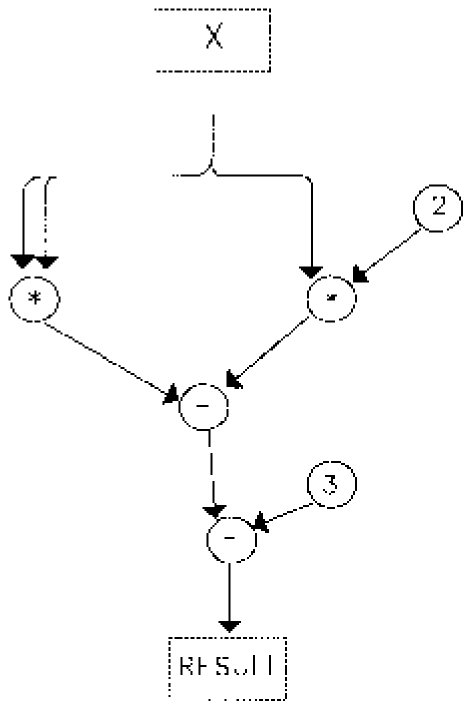
Various notations used to construct a data flow diagram with help of operators (nodes) and links (arcs)

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



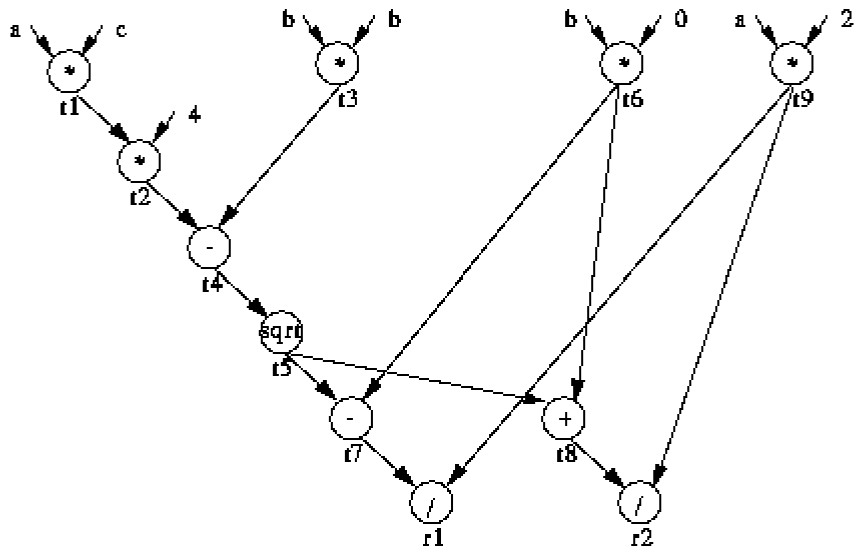
Data flow diagram for the equation $x^2 - 2x + 3$

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)

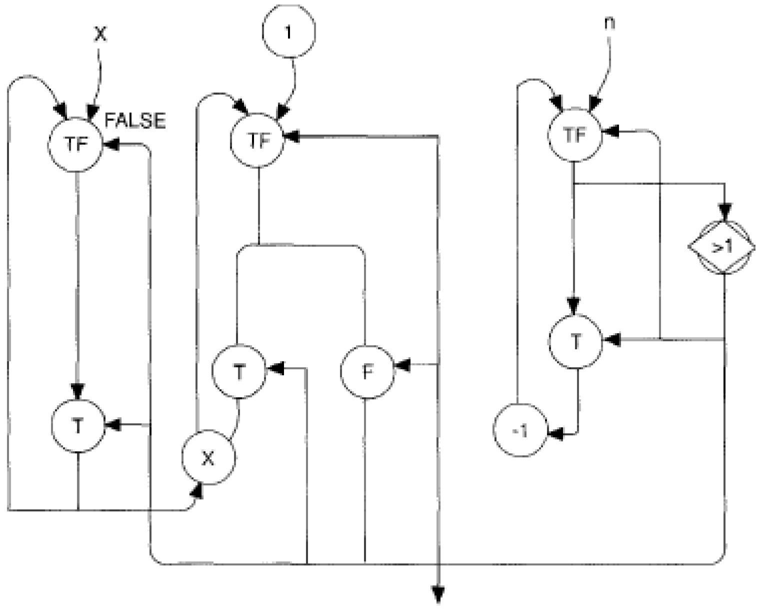
```
{
t1 = a*c;
t2 = 4*t1;
t3 = b*b;
t4 = t3 - t2;
t5 = sqrt( t4);
t6 = -b;
t7 = t6 - t5;
t8 = t7 + t5;
t9 = 2*a;
r1 = t7/t9;
r2 = t8/t9;
}
```

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:

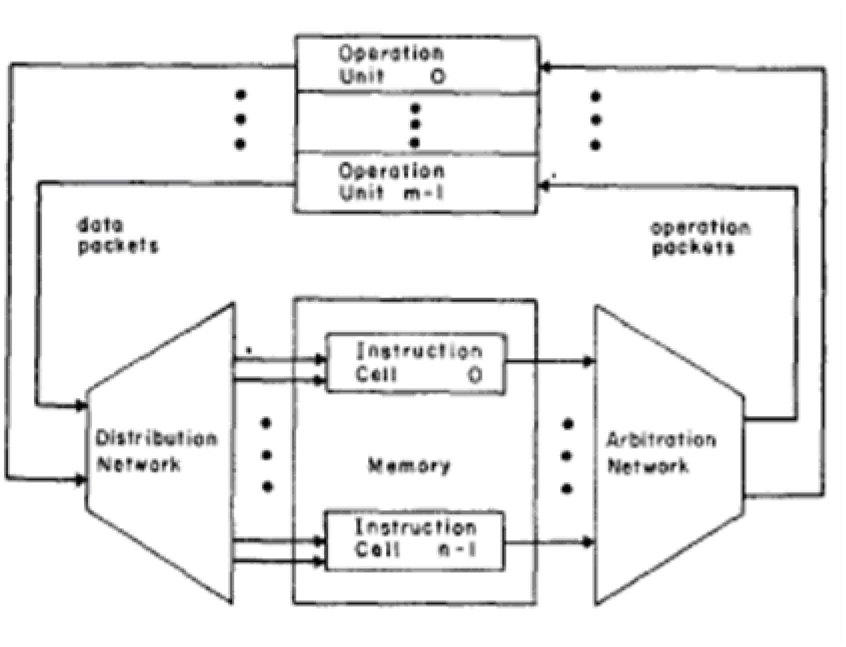
1. The development of efficient data flow languages which are easy to use and to be interpreted by machine hardware
2. The decomposition of programs and the assignment of program modules to data flow processors
3. Controlling and supporting large amounts of interprocessor communication with cost-effective 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
  - RISC approach,
  - complex machine operations,
  - multithreading,
  - large-grain computation,
  - 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

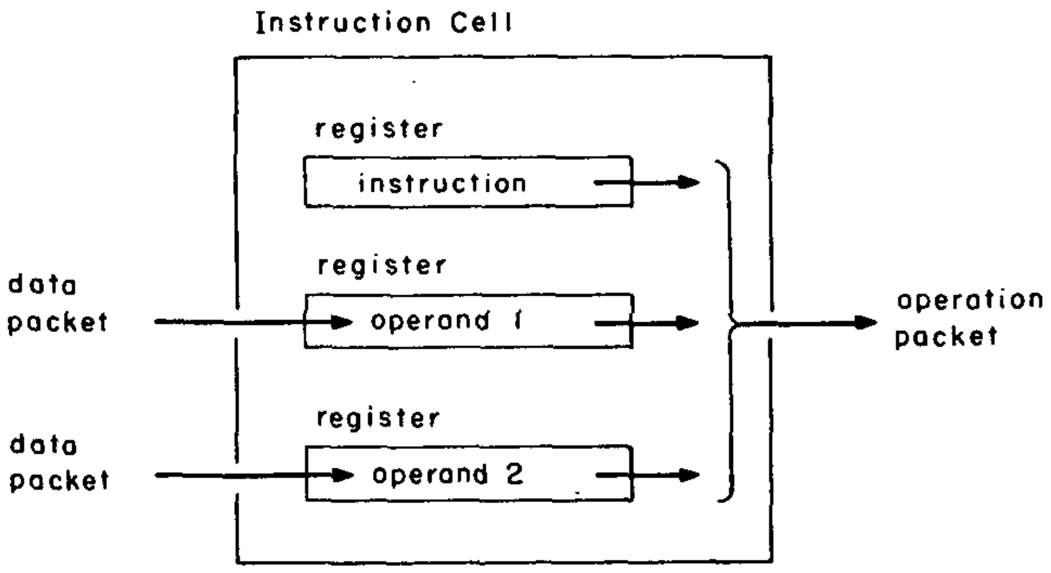
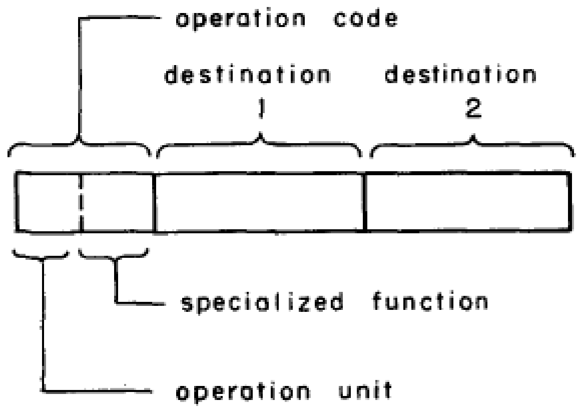


Figure 3. Operation of an Instruction Cell.



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.

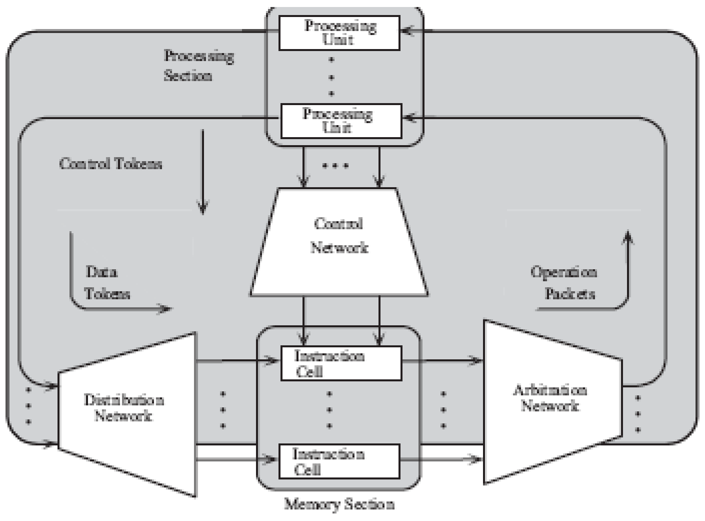
The static data flow adopts a handshaking acknowledgement mechanism which can take the form of special control tokens set from processors once they respond to a fired node. In order to implement this, acknowledge arcs are implicitly added to the dataflow graph that go in the opposite direction to each 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).

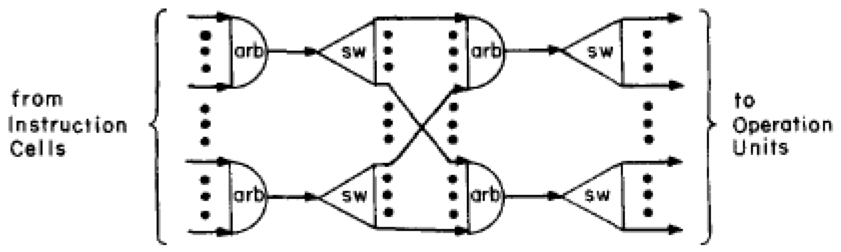


Figure 5. Structure of the Arbitration Network.

- 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,

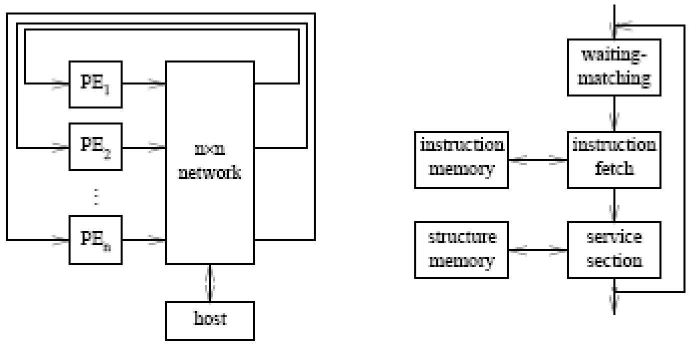
- Major advantage of the dynamic data flow computers is its better performance as compared with static data flow computer as this architecture allows existence of multiple tokens on each arc which thereby lead to unfold iterative program leading to more parallelism.

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.

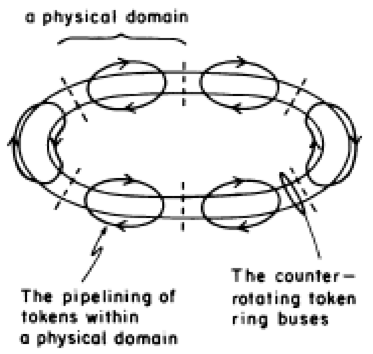

a physical domain
The pipelining of tokens within a physical domain
The counter-rotating token ring buses

Fig. 7.   Physical domains operating concurrently.

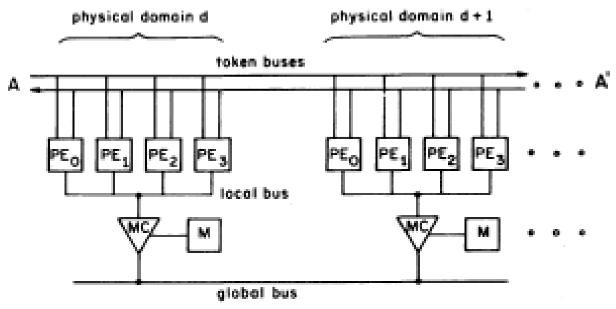


Fig. 5.   A ring domain.

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.

.