Instruction Stream and Data Stream

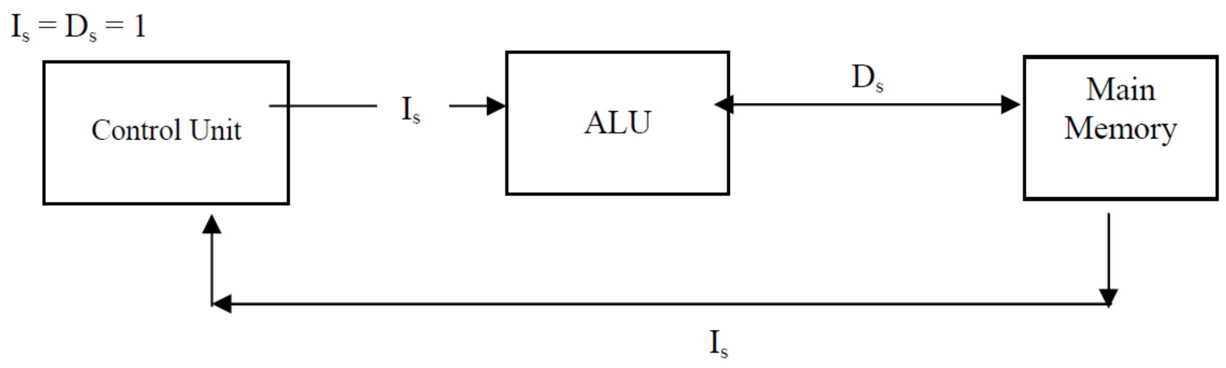
The term 'stream' refers to a sequence or flow of either instructions or data operated on by the computer. In the complete cycle of instruction execution, a flow of instructions from main memory to the CPU is established. This flow of instructions is called **instruction stream.** Similarly, there is a flow of operands between processor and memory bi-directionally. This flow of operands is called **data stream**. These two types of streams are shown in figure. Thus, it can be said that the sequence of instructions executed by CPU forms the Instruction streams and sequence of data (operands) required for execution of instructions from the Data streams.

Flynn's Classification

Flynn's classification is based on multiplicity of instruction streams and data streams observed by the CPU during program execution. Let I_s and D_s are minimum number of streams flowing at any point in the execution, then the computer organization can be categorized as follows:

1. Single Instruction and Single Data (SISD)

In this organization, sequential execution of instructions is performed by one CPU containing a single processing element (PE), i.e., ALU under one control unit as shown in figure. Therefore, SISD machines are conventional serial computers that process only one stream of instructions and one stream of data. This type of computer organization is depicted in the diagram:

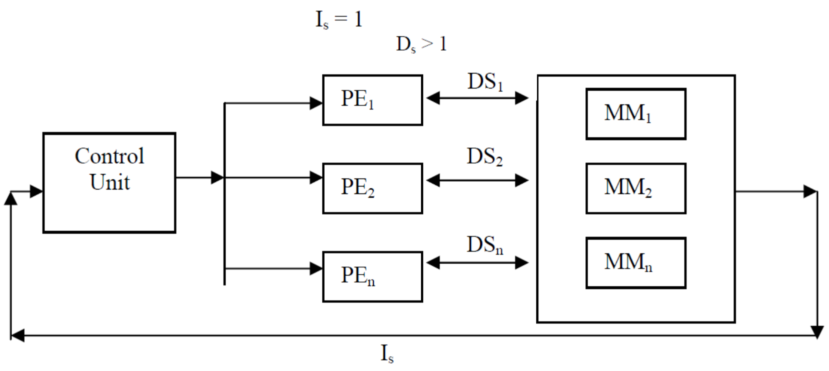


Examples of SISD machines include:

- CDC 6600 which is unpipelined but has multiple functional units.
- CDC 7600 which has a pipelined arithmetic unit.
- Amdhal 470/6 which has pipelined instruction processing.
- Cray-1 which supports vector processing.

2. Single Instruction and Multiple Data (SIMD)

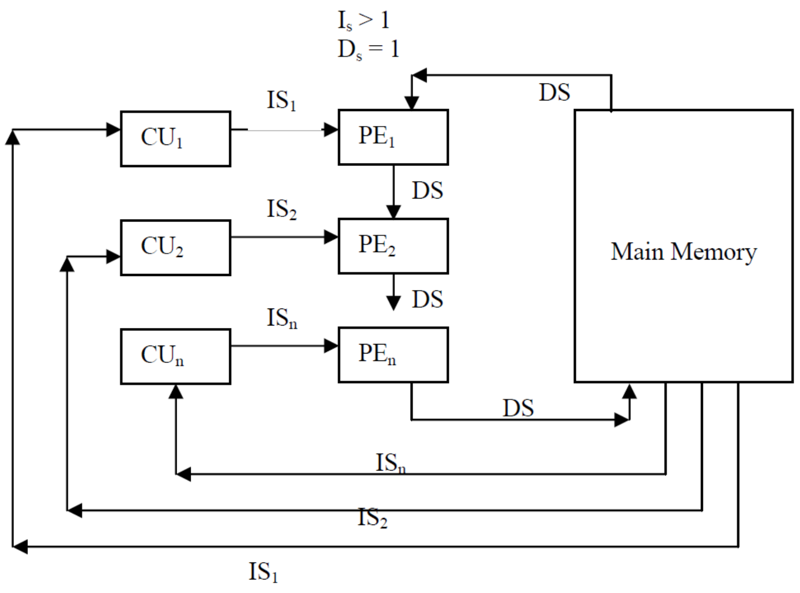
In this organization, multiple processing elements work under the control of a single control unit. It has one instruction and multiple data stream. All the processing elements of this organization receive the same instruction broadcast from the CU. Main memory can also be divided into modules for generating multiple data streams acting as a distributed memory as shown in figure. Therefore, all the processing elements simultaneously execute the same instruction and are said to be 'lock-stepped' together. Each processor takes the data from its own memory and hence it has on distinct data streams. (Some systems also provide a shared global memory for communications.) Every processor must be allowed to complete its instruction before the next instruction is taken for execution. Thus, the execution of instructions is synchronous.



Examples of SIMD organisation: ILLIAC-IV, PEPE, BSP, STARAN, MPP, DAP and CM-1.

3. Multiple Instruction and Single Data (MISD)

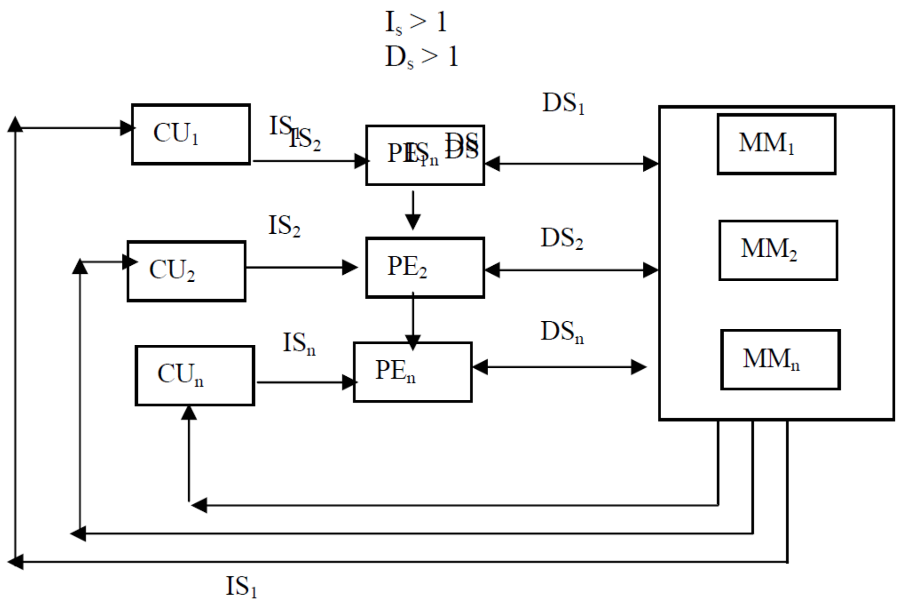
In this organization, multiple processing elements are organized under the control of multiple control units. Each control unit is handling one instruction stream and processed through its corresponding processing element. But each processing element is processing only a single data stream at a time. Therefore, for handling multiple instruction streams and single data stream, multiple control units and multiple processing elements are organized in this classification. All processing elements are interacting with the common shared memory for the organization of single data stream as shown in Figure 6. The only known example of a computer capable of MISD operation is the C.mmp built by Carnegie-Mellon University.



This classification is not popular in commercial machines as the concept of single data streams executing on multiple processors is rarely applied. But for the specialized applications, MISD organization can be very helpful. For example, Real time computers need to be fault tolerant where several processors execute the same data for producing the redundant data. This is also known as N- version programming. All these redundant data are compared as results which should be same; otherwise faulty unit is replaced. Thus MISD machines can be applied to fault tolerant real time computers.

4. Multiple Instructions and Multiple Data (MIMD)

In this organization, multiple processing elements and multiple control units are organized as in MISD. But the difference is that now in this organization multiple instruction streams operate on multiple data streams. Therefore, for handling multiple instruction streams, multiple control units and multiple processing elements are organized such that multiple processing elements are handling multiple data streams from the Main memory as shown in figure. The processors work on their own data with their own instructions. Tasks executed by different processors can start or finish at different times. They are not lock-stepped, as in SIMD computers, but run asynchronously. This classification actually recognizes the parallel computer. That means in the real sense MIMD organization is said to be a Parallel computer. All multiprocessor systems fall under this classification. Examples include; C.mmp, Burroughs D825, Cray-2, S1, Cray X-MP, HEP, Pluribus, IBM 370/168 MP, Univac 1100/80, Tandem/16, IBM 3081/3084, C.m*, BBN Butterfly, Meiko Computing Surface (CS-1), FPS T/40000, iPSC.



Of the classifications discussed above, MIMD organization is the most popular for a parallel computer. In the real sense, parallel computers execute the instructions in MIMD mode.

PARALLEL AND MULTIPROCESSOR ARCHITECTURES

- Since the beginning of computing, scientists have endeavored to make machines solve problems better and faster. Miniaturization technology has resulted in smaller and denser chips. Clocks have become faster but heat and electromagnetic interference limits chip transistor density. On top of these physical limitations, there are also economic limitations.
- Ultimately, we will be left with no feasible way of improving processor performance except to distribute the computational load among several processors. For these reasons, parallelism is becoming increasingly popular.
- Parallelism results in higher throughput, better fault tolerance, and a more attractive price/performance ratio.
- Although parallelism can result in significant speedup, this speedup can never be perfect. We need only recall Amdahl's Law to realize why perfect speedup is not possible. If two processing components run at two different speeds, the slower speed will dominate.
- The additional processors can do nothing but wait until the serial processing is complete because every algorithm has a sequential part that ultimately limits the speedup achievable through a multiprocessor implementation. Using multiple processors on a single task is only one of many different types of parallelism.
- Examples include SIMD machines such as vector, neural, and systolic processors. There are many architectures that allow for multiple or parallel processes, characteristic of all MIMD machines.

Superscalar Processor

The essence of the superscalar approach is the ability to execute instructions independently and concurrently in different pipelines. The concept can be further exploited by allowing instructions to be executed in an order different from the program order. Figure shows the superscalar approach. There are multiple functional units, each of which is implemented as a pipeline, which support parallel execution of several instructions. In this example, two integer, two floating-point, and one memory (either load or store) operations can be executing at the same time.

- A **superscalar processor** is one in which multiple independent instruction pipelines are used.
- Each pipeline consists of multiple stages, so that each pipeline can handle multiple instructions at a time.
- Multiple pipelines introduce a new level of parallelism, enabling multiple streams of instructions to be processed at a time.
- A superscalar processor exploits what is known as **instruction-level parallelism**, which refers to the degree to which the instructions of a program can be executed in parallel.
- *Super pipelining* occurs when a pipeline has stages that require less than half a clock cycle to execute. An internal clock can be added which, when running at double the speed of the external clock, can complete two tasks per external clock cycle.

- A superscalar processor typically fetches multiple instructions at a time and then attempts to find nearby instructions that are independent of one another and can therefore be executed in parallel.
- If the input to one instruction depends on the output of a preceding instruction, then the latter instruction cannot complete execution at the same time or before the former instruction.
- Once such dependencies have been identified, the processor may issue and complete instructions in an order that differs from that of the original machine code.
- The processor may eliminate some unnecessary dependencies by the use of additional registers and the renaming of register references in the original code.

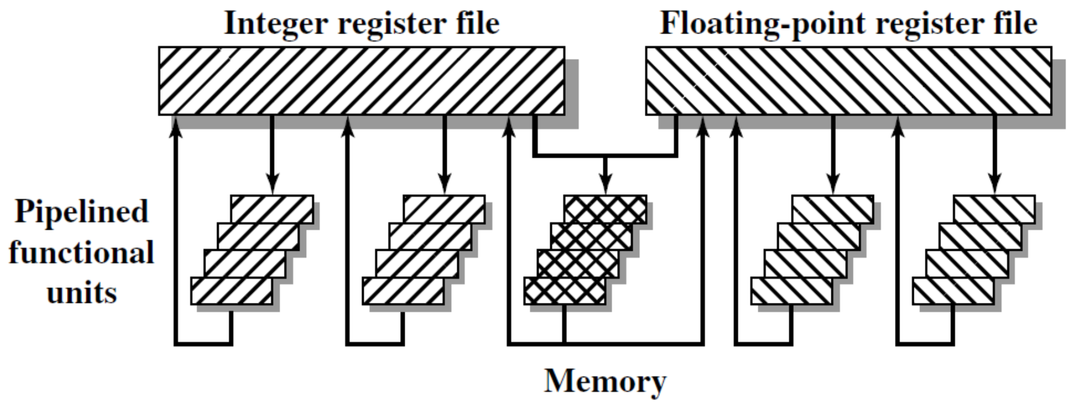


Fig.: General Superscalar Organization

Very Long Instruction Word (VLIW)

- Whereas superscalar processors rely on both the hardware (to arbitrate dependencies) and the compiler (to generate approximate schedules), VLIW processors rely *entirely* on the compiler.
- VLIW processors pack independent instructions into one long instruction, which, in turn, tells the execution units what to do.
- Many argue that because the compiler has a better overall picture of dependencies in the code, this approach results in better performance. However, the compiler cannot have an overall picture of the run-time code so it is compelled to be conservative in its scheduling.
- As a VLIW compiler creates very long instructions, it also arbitrates all dependencies. The instructions, which are fixed at compile time, typically contain four to eight instructions. Because the instructions are fixed, any modification that could affect scheduling of instructions (such as changing memory latency) requires a recompilation of the code, potentially causing a multitude of problems for software vendors. VLIW supporters point out that this technology simplifies the hardware by moving complexity to the compiler. Superscalar supporters counter with the argument that VLIW can, in turn, lead to significant increases in the amount of code generated.

- For example, when program control fields are not used, memory space and bandwidth are wasted. In fact, a typical FORTRAN program explodes to double and sometimes triple its normal size when compiled on a VLIW machine.
- An EPIC architecture holds some advantages over an ordinary VLIW processor. Like VLIW, EPIC bundles its instructions for delivery to various execution units.
- However, unlike VLIW, these bundles need not be the same length. A special delimiter indicates where one bundle ends and another begins.
- Instruction words are pre-fetched by hardware, which identifies and then schedules the bundles in independent groups for parallel execution.
- This is an attempt to overcome the limitations introduced by the compiler's lack of total knowledge of the run-time code. Instructions within bundles may be executed in parallel with no concern for dependencies, and thus no concern for ordering. By most people's definition, EPIC is really VLIW.

Vector Processors

- Often referred to as supercomputers, *vector processors* are specialized, heavily pipelined processors that perform efficient operations on entire vectors and matrices at once. This class of processor is suited for applications that can benefit from a high degree of parallelism, such as weather forecasting, medical diagnoses, and image processing.
- A vector is a fixed-length, one-dimensional array of values, or an ordered series of scalar quantities. Various arithmetic operations are defined over vectors, including addition, subtraction, and multiplication.
- Vector computers are highly pipelined so that arithmetic operations can be overlapped. Each instruction specifies a set of operations to be carried over an entire vector. For example, let's say we want to add vectors V1 and V2 and place the results in V3. On a traditional processor our code would include the following loop:

```
for i = 0 to Vector Length

V3[i] = V1[i] + V2[i];
```

- However, on a vector processor, this code becomes

```
LDV    V1, R1      ;load vector1 into vector register R1

LDV    V2, R2

ADDV   R3, R1, R2

STV    R3, V3      ;store vector register R3 in vector V3
```

- *Vector registers* are specialized registers that can hold several vector elements at one time. The register contents are sent one element at a time to a vector pipeline, and the output from the pipeline is sent back to the vector registers one element at a time.

- These registers are, therefore, FIFO queues capable of holding many values. Vector processors generally have several of these registers.
- The instruction set for a vector processor contains instructions for loading these registers, performing operations on the elements within the registers, and storing the vector data back to memory.
- Vector processors are often divided into two categories according to how the instructions access their operands.
- *Register-register vector processors* require that all operations use registers as source and destination operands.
- *Memory-memory vector processors* allow operands from memory to be routed directly to the arithmetic unit. The results of the operation are then streamed back to memory.
- Register-to-register processors are at a disadvantage in that long vectors must be broken into fixed length segments that are small enough to fit into the registers.
- However, memory-to-memory machines have a large startup time due to memory latency. (Startup time is the time between initializing the instruction and the first result emerging from the pipeline.) After the pipeline is full, however, this disadvantage disappears.
- Vector instructions are efficient for two reasons.
- First, the machine fetches significantly fewer instructions, which means there is less decoding, control unit overhead, and memory bandwidth usage.
- Second, the processor knows it will have a continuous source of data and can begin pre-fetching corresponding pairs of values.
- If interleaved memory is used, one pair can arrive per clock cycle.
- The most famous vector processors are the Cray series of supercomputers.

Shared Memory Multiprocessors

Multiprocessors are classified by how memory is organized. Tightly coupled systems use the same memory and are thus known as *shared memory processors*. This does not mean all processors must share one large memory. Each processor could have a local memory, but it must be shared with other processors. It is also possible that local caches could be used with a single global memory. These three ideas are illustrated in figure.

The global shared memory was divided equally among the processors. If a processor generated an address, it would first check its local memory. If the address was not found in local memory, it was passed on to a controller. The controller would try to locate the address within the processors that occupied the subtree for which it was the root. If the required address still could not be located, the request was passed up the tree until the data was found or the system ran out of places to look.

Shared memory MIMD machines can be divided into two categories according to how they synchronize their memory operations.

Uniform Memory Access **(*UMA*)**

In *Uniform Memory Access* (*UMA*) systems, all memory accesses take the same amount of time. A UMA machine has one pool of shared memory that is connected to a group of processors through a bus or switch network. All processors have equal access to memory, according to the established protocol of the interconnection network. As the number of processors increases, a switched interconnection network (requiring 2^nconnections) quickly becomes very expensive in a UMA machine. Bus based UMA systems saturate when the bandwidth of the bus becomes insufficient for the number of processors in the system. Multistage networks run into wiring constraints and significant latency if the number of processors becomes very large. Thus the scalability of UMA machines is limited by the properties of interconnection networks. Symmetric multiprocessors are well-known UMA architectures. Specific examples of UMA machines include *Sun's Ultra Enterprise, IBM's iSeries and pSeries servers*, the *Hewlett-Packard 900*, and *DEC's AlphaServer*.

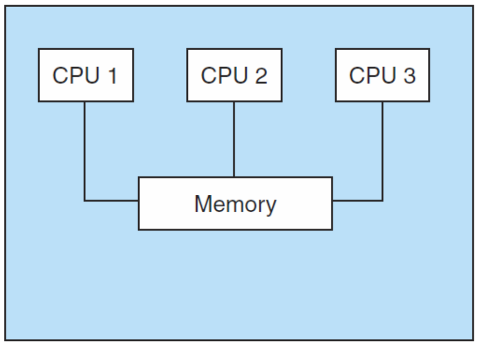


Fig.1: global shared memory

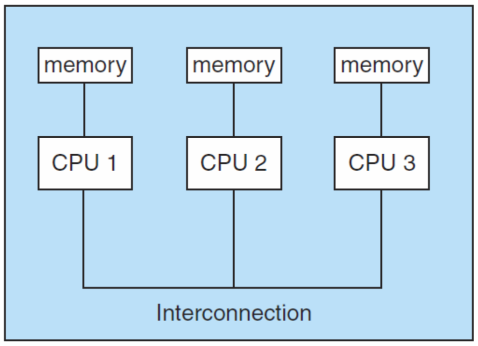


Fig.2: distributed shared memory

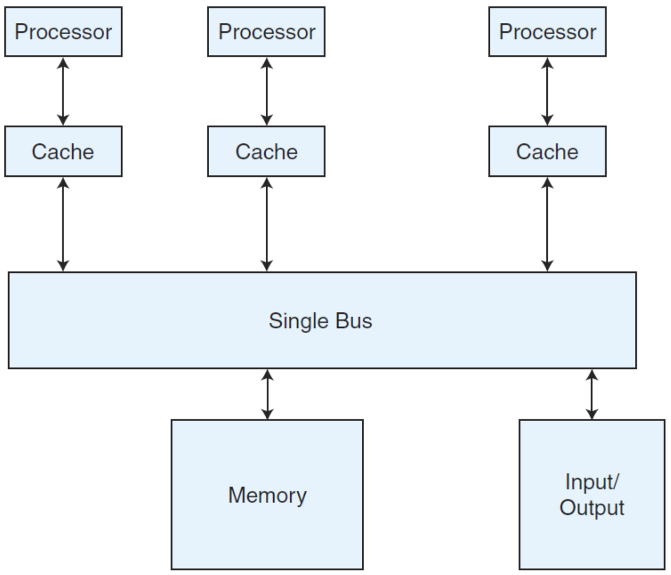


Fig.3: global shared memory with separate cache at processors

***Non-uniform memory access* (*NUMA*)**

Non-uniform memory access (*NUMA*) machines get around the problems inherent in UMA architectures by providing each processor with its own piece of memory. The memory space of the machine is distributed across all of the processors, but the processors see this memory as a contiguous addressable entity. Although NUMA memory constitutes a single addressable entity, its distributed nature is not completely transparent. Nearby memory takes less time to read than memory that is further away. Thus, memory access time is inconsistent across the address space of the machine.

NUMA machines are prone to *cache coherence* problems. To reduce memory access time, each NUMA processor maintains a private cache. However, when a processor modifies a data element that is in its local cache, other copies of the data become inconsistent.

Specially designed hardware units known as *snoopy cache controllers* monitor all caches on the system. They implement the system's cache consistency protocol. NUMA machines that employ snoopy caches and maintain cache consistency are referred to as *CC-NUMA* (*cache coherent NUMA*) architectures.

Dataflow Computing

The architecture of a dataflow machine consists of a number of processing elements that must communicate with each other. Each processing element has an *enabling unit* that sequentially accepts tokens and stores them in memory. If the node to which this token is addressed fires, the input tokens are extracted from memory and are combined with the node itself to form an executable packet. The processing element's *functional unit* computes any necessary output values and combines them with destination addresses to form more tokens. These tokens are then sent back to the enabling unit, at which time they may enable other nodes. In a tagged-token machine, the enabling unit is split into two separate stages: the *matching unit* and the *fetching unit*. The matching unit stores tokens in its memory and determines whether an instance of a given destination node has been enabled. In tagged architectures, there must be a match of both the destination node address and the tag. When all matching tokens for a node are available, they are sent to the fetching unit, which combines these tokens with a copy of the node into an executable packet, which is then passed on to the functional unit.

We can understand the computation sequence of a dataflow computer by examining its *data flow graph*. In a data flow graph, nodes represent instructions, and arcs indicate data dependencies between instructions. Data flows through this graph in the form of *data tokens*. When an instruction has all of the data tokens it needs, the node *fires*. When a node fires, it consumes the data tokens, performs the required operation, and places the resulting data token on an output arc. This idea is illustrated in figure.

The data flow graph shown in figure is an example of a *static* dataflow architecture in which the tokens flow through the graph in a staged pipelined fashion. In *dynamic* dataflow architectures, tokens are tagged with context information and are stored in a memory. During every clock cycle, memory is searched for the set of tokens necessary for a node to fire. Nodes fire only when they find a complete set of input tokens within the same context.

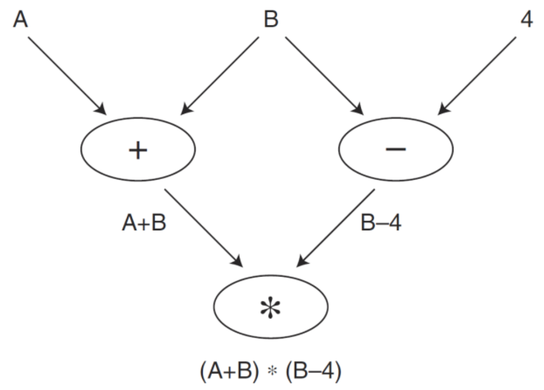


Figure: Data flow Graph computing N=(A+B)*(B-4)

Because data drives processing on dataflow systems, dataflow multiprocessors do not suffer from the contention and cache coherency problems that are so vexing to designers of control driven multiprocessors. It is interesting to note that von Neumann, whose name is given to the von Neumann bottleneck in traditional computer architectures, studied the possibility of data-driven architectures similar in nature to dataflow machines.

Neural Networks

Traditional architectures are quite good at fast arithmetic and executing deterministic programs. However, they are not so good at massively parallel applications, fault tolerance, or adapting to changing circumstances. Neural networks, on the other hand, are useful in dynamic situations where we can't formulate an exact algorithmic solution, and where processing is based on an accumulation of previous behavior.

A neural network consists of processing elements (PEs), which multiply inputs by various sets of weights, yielding a single output value. The actual computation involved is deceptively easy; the true power of a neural network comes from the parallel processing of the interconnected PEs and the adaptive nature of the sets of weights. Weights and thresholds must be changed to compensate for the error. The network's *learning algorithm* is the set of rules that governs how these changes are to be made.

Neural network computers are composed of a large number of simple processing elements that individually handle one piece of a much larger problem. The simplest example of a neural net is the *perceptron*, a single trainable neuron. A perceptron produces a Boolean output based on the values that it receives from several inputs. A perceptron is trainable because its threshold and input weights are modifiable. Figure shows a perceptron with inputs x_1, x_2, ... x_n, which can be Boolean or real values. Z is the Boolean output. The w_i's represent the weights of the edges and are real values. T is the real-valued threshold. In this example, the output Z is true (1) if the net input, $w_1x_1 + w_2x_2 + \ldots + w_nx_n$, is greater than the threshold T. Otherwise, Z is zero.

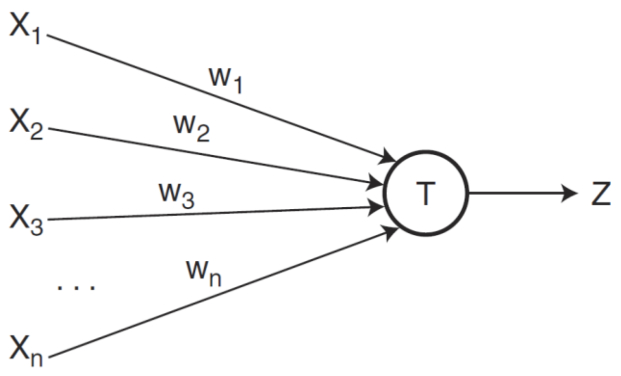


Figure : A perceptron

A perceptron produces outputs for specific inputs according to how it is trained. If the training is done correctly, we should be able to give it any input and get reasonably correct output. The perceptron should be able to determine a reasonable output even if it has never before seen a particular set of inputs. The “reasonableness” of the output depends on how well the perceptron is trained.

Perceptrons are trained by use of either supervised or unsupervised learning algorithms.

Supervised learning assumes prior knowledge of correct results, which are fed to the neural network during the training phase. While it is learning, the neural network is told whether its final state is correct. If the output is incorrect, the network modifies input weights to produce the desired results.

Unsupervised learning does not provide the correct output to the network during training. The network adapts solely in response to its inputs, learning to recognize patterns and structure in the input sets.

The best way to train a neural network is to compile a wide range of examples that exhibit the very characteristics in which you are interested. A neural network is only as good as the training data, so great care must be taken to select a sufficient number of correct examples.

Neural networks are known by many different names, including *connectionist systems*, *adaptive systems*, and *parallel distributed processing systems*. These systems are particularly powerful when a large database of previous examples can be used by the neural net to learn from prior experiences. They have been used successfully in a multitude of real world applications including quality control, weather forecasting, financial and economic forecasting, speech and pattern recognition, oil and gas exploration, health care cost reduction, bankruptcy prediction, machine diagnostics, securities trading, and targeted marketing. It is important to note that each of these neural nets has been specifically designed for a certain task, so we cannot take a neural network designed for weather forecasting and expect it to do a good job for economic forecasting.