UNIT 4

Instruction-Level Parallelism and Dynamic Exploitation

3.1 Instruction-Level Parallelism: Concepts and Challenges:

Instruction-level parallelism (ILP) is the potential overlap the execution of instructions using pipeline concept to improve performance of the system. The various techniques that are used to increase amount of parallelism are reduces the impact of data and control hazards and increases processor ability to exploit parallelism

There are two approaches to exploiting ILP.

1. Static Technique – Software Dependent
2. Dynamic Technique – Hardware Dependent

The static technique is compiler-intensive approach, which have broader adoption in the embedded market than the desktop or server markets, the new IA-64 architecture and Intel's Itanium, use this more static approach.

The dynamic technique is hardware intensive approach, which dominate the desktop and server markets and are used in a wide range of processors, including: the Pentium III and 4, the Althon, the MIPS R10000/12000, the Sun ultraSPARC III, the Power-PC 603, G3, and G4, and the Alpha 21264.

The simplest and most common way to increase the amount of parallelism is loop-level parallelism. Here is a simple example of a loop, which adds two 1000-element arrays, that is completely parallel:

for (i=1;i<=1000; i=i+1) x[i] = x[i] + y[i];

Every iteration of the loop can overlap with any other iteration, although within each loop iteration there is little or no opportunity for overlap.

There are a number of techniques for converting such loop-level parallelism into instruction-level parallelism are basically work by unrolling the loop either statically by the compiler or dynamically by the hardware. An important alternative method for exploiting loop-level parallelism is the use of vector instructions.

Data Dependence and Hazards:

To exploit instruction-level parallelism, determine which instructions can be executed in parallel. If two instructions are parallel, they can execute simultaneously in a pipeline without causing any stalls. If two instructions are dependent they are not parallel and must be executed in order.

There are three different types of dependences: data dependences (also called true data dependences), name dependences, and control dependences.

Data Dependences:

An instruction j is data dependent on instruction i if either of the following holds:

- Instruction i produces a result that may be used by instruction j, or
- Instruction j is data dependent on instruction k, and instruction k is data dependent on instruction i.

The second condition simply states that one instruction is dependent on another if there exists a chain of dependences of the first type between the two instructions. This dependence chain can be as long as the entire program.

For example, consider the following code sequence that increments a vector of values in memory (starting at 0(R1) and with the last element at 8(R2)) by a scalar in register F2:

```
Loop: LW F0,0($S1) ; F0=array element
ADD.D F4,F0,F2 ; add scalar in F2
S.D F4,0($S1) ;store result
DADDUI $S1,$S1,#-8 ;decrement pointer 8 bytes
 BNE $S1,$S2,LOOP ; branch $S1!=zero
```

The data dependences in this code sequence involve both floating point data:

```
Loop: L.D F0,0(R1) ;F0=array element
ADD.D F4,F0,F2 ;add scalar in F2 S.D F4,0(R1) ;store result
and integer data:
        DADDIU R1,R1,-8 ;decrement pointer ;8 bytes (per DW)
        BNE R1,R2,Loop ; branch R1!=zero
```

Both of the above dependent sequences, as shown by the arrows, with each instruction depending on the previous one. The arrows here and in following examples show the order that must be preserved for correct execution. The arrow points from an instruction that must precede the instruction that the arrowhead points to.

If two instructions are data dependent they cannot execute simultaneously or be completely overlapped. The dependence implies that there would be a chain of one or more data hazards between the two instructions. Executing the instructions simultaneously will cause a processor with pipeline interlocks to detect a hazard and stall, thereby reducing or eliminating the overlap. Dependences are a property of programs. Whether a given dependence results in an actual hazard being detected and

whether that hazard actually causes a stall are properties of the pipeline organization. This difference is critical to understanding how instruction-level parallelism can be exploited.

The presence of the dependence indicates the potential for a hazard, but the actual hazard and the length of any stall is a property of the pipeline. The importance of the data dependences is that a dependence (1) indicates the possibility of a hazard, (2) determines the order in which results must be calculated, and (3) sets an upper bound on how much parallelism can possibly be exploited. Such limits are explored in section 3.8.

Name Dependences

The name dependence occurs when two instructions use the same register or memory location, called a name, but there is no flow of data between the instructions associated with that name.

There are two types of name dependences between an instruction i that precedes instruction j in program order:

An antidependence between instruction i and instruction j occurs when instruction j writes a register or memory location that instruction i reads. The original ordering must be preserved to ensure that i reads the correct value.

• An output dependence occurs when instruction i and instruction j write the same register or memory location. The ordering between the instructions must be preserved to ensure that the value finally written corresponds to instruction j.

Both anti-dependences and output dependences are name dependences, as opposed to true data dependences, since there is no value being transmitted between the instructions. Since a name dependence is not a true dependence, instructions involved in a name dependence can execute simultaneously or be reordered, if the name (register number or memory location) used in the instructions is changed so the instructions do not conflict.

This renaming can be more easily done for register operands, where it is called register renaming. Register renaming can be done either statically by a compiler or dynamically by the hardware. Before describing dependences arising from branches, let's examine the relationship between dependences and pipeline data hazards.

Control Dependences:

A control dependence determines the ordering of an instruction, i, with respect to a branch instruction so that the instruction i is executed in correct program order. Every instruction, except for those in the first basic block of the program, is control dependent on some set of branches, and, in general, these control dependences must be preserved to preserve program order. One of the simplest

examples of a control dependence is the dependence of the statements in the "then" part of an if statement on the branch. For example, in the code segment:

```
if p1 { S1;
};
if p2 { S2;
}
```

S1 is control dependent on p1, and S2is control dependent on p2 but not on p1. In general, there are two constraints imposed by control dependences:

An instruction that is control dependent on a branch cannot be moved before the branch so that its execution is **no longer controlled** by the branch. For example, we cannot take an instruction from the then-portion of an if-statement and move it before the if-statement.

2. An instruction that is not control dependent on a branch cannot be moved after the branch so that its execution is **controlled** by the branch. For example, we cannot take a statement before the if-statement and move it into the then-portion.

Control dependence is preserved by two properties in a simple pipeline, **First- Exception Behaviour** Preserving the *exception behavior* means that any changes in the ordering of instruction execution must not change how exceptions are raised in the program.. **Second, - Data flow** The data flow is the actual flow of data values among instructions that produce results and those that consume them

limitations of ILP.

3.8. Studies of the Limitations of ILP

The Hardware Model

An ideal processor is one where all artificial constraints on ILP are removed. The only limits on ILP in such a processor are those imposed by the actual data flows either through registers or memory.

The assumptions made for an ideal or perfect processor are as follows:

1. *Register renaming*—There are an infinite number of virtual registers available and hence all WAW and WAR hazards are avoided and an unbounded number of instructions can begin execution simultaneously.

2. *Branch prediction*—Branch prediction is perfect. All conditional branches are predicted exactly.

3. *Jump prediction*—All jumps (including jump register used for return and computed jumps) are perfectly predicted. When combined with perfect branch prediction, this is equivalent to having a processor with perfect speculation and an unbounded buffer of instructions available for execution.

4. *Memory-address alias analysis*—All memory addresses are known exactly and a load can be moved before a store provided that the addresses are not identical.

Assumptions 2 and 3 eliminate *all* control dependences. Likewise, assumptions 1 and 4 eliminate *all but the true* data dependences. Together, these four assumptions mean that *any* instruction in the of the program's execution can be scheduled on the cycle immediately following the execution of the predecessor on which it depends.

Limitations on the Window Size and Maximum Issue Count

A dynamic processor might be able to more closely match the amount of parallelism uncovered by our ideal processor. consider what the perfect processor must do:

1. Look arbitrarily far ahead to find a set of instructions to issue, predicting all branches perfectly.
2. Rename all register uses to avoid WAR and WAW hazards.
3. Determine whether there are any data dependencies among the instructions in the issue packet; if so, rename accordingly.
4. Determine if any memory dependences exist among the issuing instructions and handle them appropriately.
5. Provide enough replicated functional units to allow all the ready instructions to issue.

Obviously, this analysis is quite complicated. For example, to determine whether *n* issuing instructions have any register dependences among them, assuming all instructions are register-register and the total number of registers is unbounded, requires

$$2n-2+2n-4+\ldots\ldots..+2 = 2\Sigma_{i=1}^{n-1} i = [2\ (n-1)n]/2 = n^2-n$$

comparisons. Thus, to detect dependences among the next 2000 instructions—the default size we assume in several figures—requires almost *four million* comparisons! Even issuing only 50 instructions requires 2450 comparisons. This cost obviously limits the number of instructions that can be considered for issue at once.

In existing and near-term processors, the costs are not quite so high, since we need only detect dependence pairs and the limited number of registers allows different solutions. Furthermore, in a real processor, issue occurs in-order and dependent instructions are handled by a renaming process that accommodates dependent renaming in one clock. Once instructions are issued, the detection of dependences is handled in a distributed fashion by the reservation stations or scoreboard.

The set of instructions that are examined for simultaneous execution is called the *window*. Each instruction in the window must be kept in the processor and the number of comparisons required every clock is equal to the maximum completion rate times the window size times the number of operands per

instruction (today typically 6 x 80 x 2= 960), since every pending instruction must look at every completing instruction for either of its operands. Thus, the total window size is limited by the required storage, the comparisons, and a limited issue rate, which makes larger window less helpful

The window size directly limits the number of instructions that begin execution in a given cycle.

The Effects of Realistic Branch and Jump Prediction :

Our ideal processor assumes that branches can be perfectly predicted: The outcome of any branch in the program is known before the first instruction is executed. Our data is for several different branch-prediction schemes varying from perfect to no predictor. We assume a separate predictor is used for jumps. Jump predictors are important primarily with the most accurate branch predictors, since the branch frequency is higher and the accuracy of the branch predictors dominates

FLYNN'S CLASSIFICATION

This classification was first studied and proposed by Michael Flynn in 1972. Flynn did not consider the machine architecture for classification of parallel computers; he introduced the concept of instruction and data streams for categorizing of computers. All the computers classified by Flynn are not parallel computers, but to grasp the concept of parallel computers, it is necessary to understand all types of Flynn's classification. Since, this classification is based on instruction and data streams, first we need to understand how the instruction cycle works.

2.3.1 Instruction Cycle

The instruction cycle consists of a sequence of steps needed for the execution of an instruction in a program. A typical instruction in a program is composed of two parts: Opcode and Operand. The Operand part specifies the data on which the specified operation is to be done. (See Figure 1). The Operand part is divided into two parts: addressing mode and the Operand.

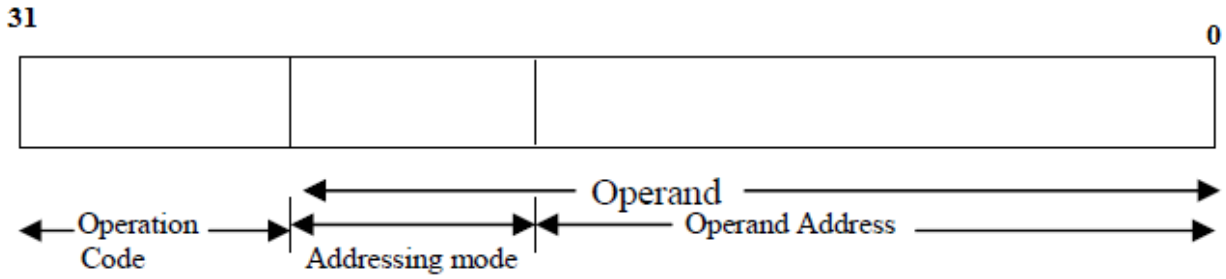

Figure 1: Opcode and Operand

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 3.

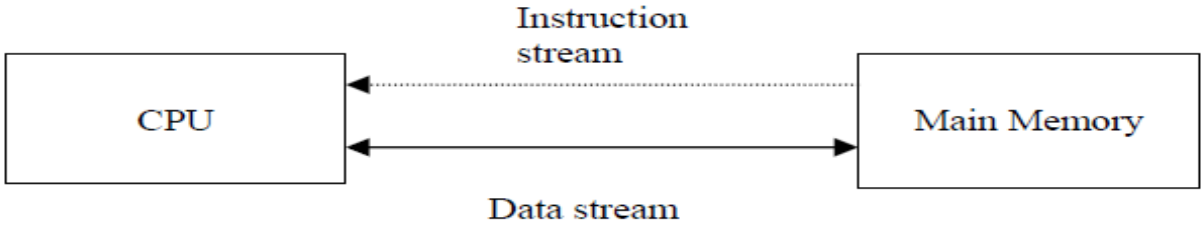

Figure 3: Instruction and data stream

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 organisation can be categorized as follows:

1) **Single Instruction and Single Data stream (SISD)**

In this organisation, 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 4. Therefore, SISD machines are conventional serial computers that process only one stream of instructions and one stream of data. This type of computer organisation is depicted in the diagram

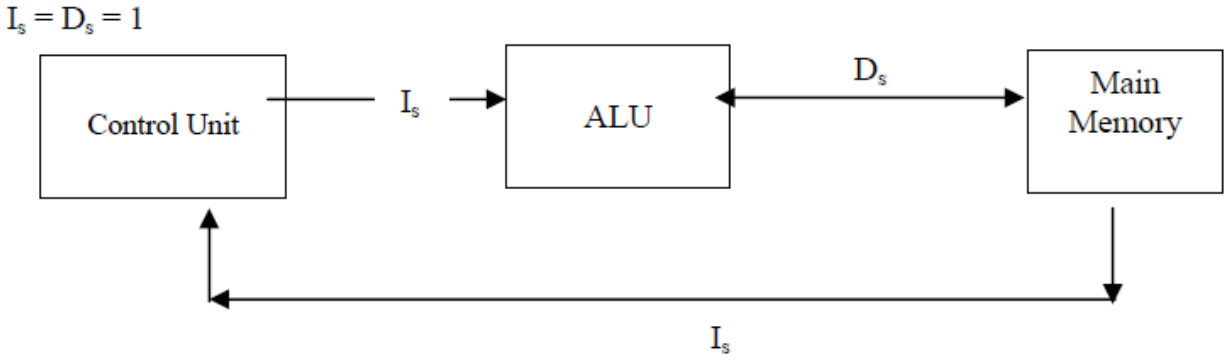

Figure 4: SISD Organisation

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 stream (SIMD)**

In this organisation, 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 5. 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 are ILLIAC-IV, PEPE, BSP, STARAN, MPP, DAP and the Connection Machine (CM-1).

This type of computer organisation is denoted as

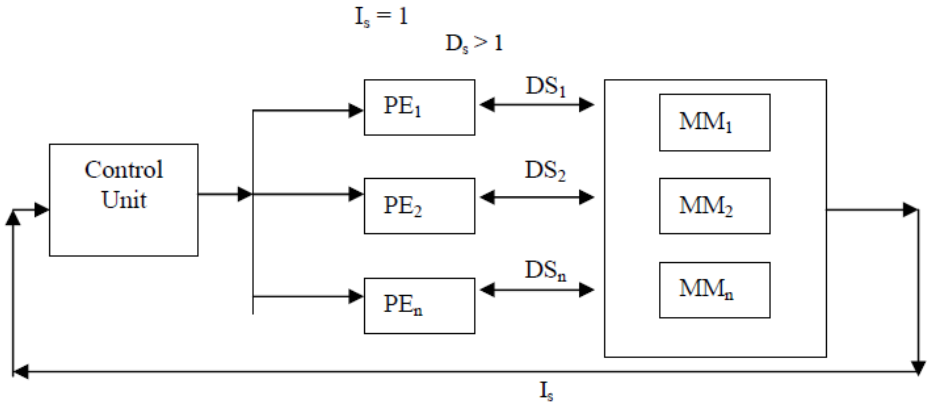

Figure 5: SIMD Organisation

3) **Multiple Instruction and Single Data stream (MISD)**

In this organization, multiple processing elements are organised 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 organised in this classification. All processing elements are interacting with the common shared memory for the organisation 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 type of computer organisation is denoted as:

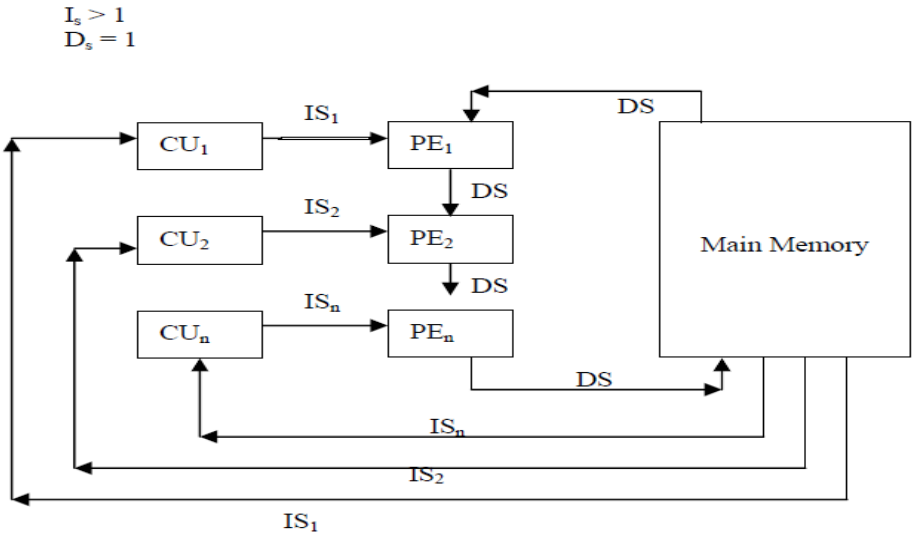

Figure 6: MISD Organisation

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 organisation 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 Instruction and Multiple Data stream (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 7. 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 organisation 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.

This type of computer organisation is denoted as:

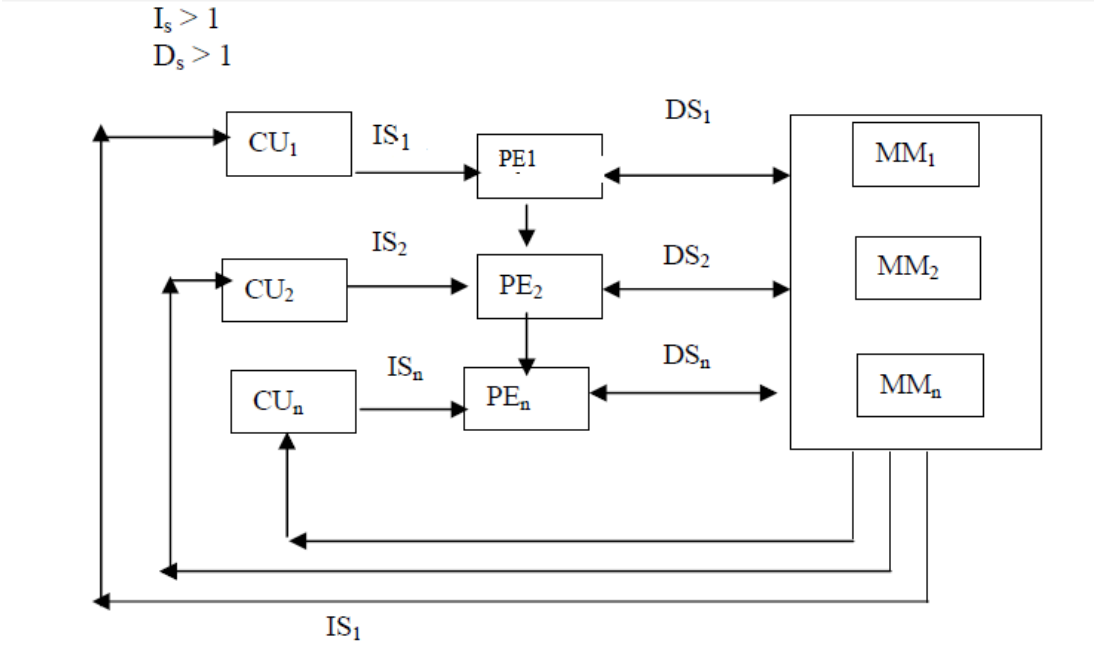

Figure 7: MIMD Organisation