What is Instruction-Level Parallelism?

The potential overlap among instruction is called instruction-level parallelism (ILP).

There are two approaches to exploiting ILP:
- Dynamic – depends on hardware to locate the parallelism,
- Static – more software oriented.
Informal data dependence definition – 1st approach

<table>
<thead>
<tr>
<th>Statement</th>
<th>Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1:</td>
<td>A = 1.0 ;</td>
</tr>
<tr>
<td>S2:</td>
<td>B = A + 3.14</td>
</tr>
<tr>
<td>S3:</td>
<td>A = 1/3 * ( C - D )</td>
</tr>
<tr>
<td>S4:</td>
<td>A = ( B * 3.8 ) / 2.71</td>
</tr>
</tbody>
</table>

- Let’s consider statement S2. The value of ‘A’ used here must be the value assigned to ‘A’ by S1. An interchange between S1 and S2 would result in S2 accessing the old value of ‘A’. That kind of dependence is called **true dependence** from S1 to S2 and it is also know as a **flow dependence** between S1 and S2.

- Now, consider statements S2 and S3. Here, we can also find that an interchange of S2 and S3, would result on S2 accessing a wrong value for ‘A’ (not the value assigned in S1 but in S3). It is called **anti dependence** from S2 to S3.

- Finally, the relation between S3 and S4 is called **output dependence**. As you can see here, the textual order must be followed, if we want to guarantee that the value for ‘A’ used after S4 is the right one.
Formal data dependence definition

Data dependence is a partial order or precedence relation on the statements of a program. Statement T depends on statement S, denoted $S \triangleright T$, if there exists an instance $S'$ of S, an instance $T'$ of T, and a memory location M, such that:

1. both $S'$ and $T'$ reference M, and at least one reference is a writer;
2. in sequential execution of the program, $S'$ is executed before $T'$;
3. in the same execution, M is not written between the times $S'$ finishes and the time $T'$ starts.

There are three types of dependences based on the references to M:
- Flow dependence: $S \triangleright T$ if S writes M and then T reads M
- Anti dependence: $S \triangleright a T$ if S reads M and then T writes M
- Output dependence: $S \triangleright o T$ if S writes M and then T writes M again
“Practical” data dependence definition

Computing the exact dependence relations can be very time consuming or even impossible. To solve that problem we can approximate the data dependence relations using \textit{IN/OUT} sets and execution order.

- The \textit{IN} set (input set) is a set of input items of statement whose values are fetched in the statement.
- The \textit{OUT} set (output set) is a set of output items of statement, whose values are changed in the statement.
- \textit{Execution order} is denoted by the symbol "\(\Theta\)", \(Si \Theta Sj\) means that statement \(Si\) can be executed before statement \(Sj\).

Then we can defined

- \textit{Flow dependence:} If \(S1 \rightarrow S2\) then \(S1 \Theta S2\) and \(\text{OUT}(S1) \cap \text{IN}(S2) \neq 0\).
- \textit{Anti dependence:} If \(S1 \leftarrow S2\) then \(S1 \Theta S2\) and \(\text{IN}(S1) \cap \text{OUT}(S2) \neq 0\).
- \textit{Output dependence:} If \(S1 \rightarrow S2\) then \(S1 \Theta S2\) and \(\text{OUT}(S1) \cap \text{OUT}(S2) \neq 0\).
There are three different types of dependences:

- Data dependences
- Name dependences
- Control dependences

An instruction \( j \) is data dependent on instruction \( i \) if either of the following holds:
- Instruction \( i \) produced 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 \( i \).
Name dependence

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 the name.

There 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 value finally written corresponds to instruction $j$. 
A hazard is created whenever there is a dependence between instructions, and they are close enough that the overlap caused by pipelining, or other reordering of instruction, would change the order of access to the operand involved in the dependence.

The goal of parallelising techniques is to exploit parallelism by preserving program order only where it effects the outcome of the program.

Detecting and avoiding hazards ensures that necessary program order is preserved.
Data hazards II

There are three classes of data hazards. Let’s consider two instructions $i$ and $j$, with $i$ occurring before $j$ in program order. The possible data hazards are:

- **RAW (read after write)** – $j$ tries to read a source before $i$ writes it, so $j$ incorrectly gets the old value,

- **WAW (write after write)** – $j$ tries to write an operand before it is written by $i$. The writes end up being performed in the wrong order, leaving the value written by $i$ rather than the value written by $j$ in the destination,

- **WAR (write after read)** – $j$ tries to write a destination before it is read by $i$, so $i$ incorrectly gets the new value.
Control Dependences I

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 and only when it should be.

- Let’s consider the following code segment
  ```java
  if p1 {
    S1;
  }
  if p2 {
    S2;
  }
  ```

- S1 is control dependent on p1, and S2 is control dependent on p2 but not on p1
Control Dependences II

There are two constraints imposed by control dependence:

- As 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.

- 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.
Preserving control dependence is a simple way to preserve program order, however it is not the critical property that must be preserved.

There are two other properties critical to program correctness, normally preserved by maintaining both data and control dependences:

- exception behavior – any changes in the ordering of instruction execution must not change how exceptions are raised in the program
- data flow – is a actual flow of data values among the instructions that produce results and those that consume them
**Exception behavior**

Reordering of instruction execution must not cause any new exception in the program

```
DADDU    R2,R3,R4
BEQZ     R2,L1
LW       R1,0(R2)
L1:...........
```

- We need to maintain the data dependence involving R2
- We cannot move load instruction before the branch
Data flow

It is not sufficient to just maintain data dependences because an instruction may be data dependent on more than one predecessor

DADDU R1,R2,R3  
BRQZ R4,L1  
DSUBU R1,R5,R6  
L1: ........  
OR R7,R1,R8

- The value of R1 used by OR instruction depends on whether the branch is taken or not
Control Dependences IV

Sometimes we can determine that violating the control dependence cannot effect either the exemption behavior or the data flow

DADDU      R1,R2,R3
BRQZ         R12, skipnext
DSUBU      R4,R5,R6
DADDU      R5,R4,R9
skipnext OR             R7,R8,R9
An example

DIV.D F0,F2,F4
ADD.D F6,F0,F8
SUB.D F8,F10,F14
MUL.D F6,F10,F8

Executing SUB before ADD, which is waiting for DIV yielding to a WAR hazard → renaming
The analysis of loop-level parallelism focuses on determining whether data accesses in later iterations are dependent on data values produced in earlier iterations – such a dependence is called a loop-carried dependence.

Let’s consider the following loop:

```plaintext
for (l=1000; l>0; l=l-1)
    x[l] = x[l] + s;
```

There is dependence between two uses of `x[l]` – it is not loop carried.

There is loop carried dependence between successive uses of `l` in different iterations – it involves an induction variable.
Consider a loop like this one

```c
for(l=1;l<=100;l=l+1)
```

Assume that A,B,C are distinct, nonoverlapping arrays.

There are two different dependences in above example

- S1 uses a value computed by S1 in an earlier iteration (iteration l computes A[l+1], which is read in iteration l+1), the same for S2 – loop carried,
- S2 uses the value A[l+1] computed by S1 in the same iteration - data dependence
Often loop-carried dependences are in the form of a recurrence.

\[
\text{For } (i=6; i<=100; i=i+1) \\
\{
Y[i] = Y[i-5] + Y[i]; 
\}
\]

The recurrence is when a variable is defined based on the value of that variable in an earlier iteration.

On the iteration \( i \), the loop references element \( i - 1 \). The loop is said to have a dependence distance of 5.

Then we have unroll the loop as a sequence of five statements that have no dependences – much more ILP.
Limitation of dependence analysis

There are two limitations that affect our ability to do accurate dependence analysis for large programs:

- Limitation in the analysis algorithms – analysis for pointers is essentially impossible for programs that use pointers in arbitrary fashion, by doing arithmetic on pointers (limited by the lack of applicability),

- Limitation in analysis behavior across procedure boundaries → interprocedural analysis (makes analysis much more difficult)