**INSTRUCTION LEVEL PARALLELISM(ILP)**

All processes use pipelining to overlap the execution of instructions and improve performance. This potential overlap among instructions is called **ILP**, since the instructions can be executed in parallel.

**Loop level parallelism**

* The simplest and most common way to increase the ILP is to exploit parallelism among iterations of a loop. This is often called **loop level parallelism**.

for (i=1; i<=1000; i++)

X[i]=X[i]+Y[i];

* Every iteration of the loop can overlap with the any other iteration.
* There are number of techniques for converting such loop level parallelism into ILP.
* Basically such techniques work by unrolling the loop either statically by the compiler or dynamically by the hardware.

**Unrolling**

for (i=1; i<=1000; i+=4)

{

X[i]+=Y[i];

X[i+1]+=Y[i+1];

X[i+2]+=Y[i+2];

X[i+3]+=Y[i+3];

}

Unrolling simply replicates the loop body multiple times adjusting the loop termination code.

**DATA DEPENDENCIES AND HAZARDS**

* Determining how one instruction depends on another is critical to determining how much parallelism exists in a program, and how that parallelism can be exploited.

There are 3 types of dependencies

1. **True data dependencies**

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

1. Instruction i produces a result that may be used by instruction j

i: a=b+10;

j: c=a/d;

1. Instruction j is data dependent on instruction k, and instruction k is data dependent on instruction i.

i: a=b+10;

k: d=a\*5;

j: f=d-5;

* If two instructions are data dependent they cannot be executed simultaneously or be completely overlapped.
* A data dependency conveys three things
  + - * 1. Possibility of a hazard
        2. The order in which results must be calculated
        3. An upper bound on how much parallelism can possibly be exploited.
      1. **Name dependencies**

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

There are **2 types of name dependence** between an instruction i that precedes instruction j in program order.

1. **Anti-dependence**

Anti-dependence between instruction i and j occurs when instruction j writes a register or memory location that i reads. The original value must be preserved to ensure i reads correct value.

i: a=b+10;

j: b=c\*10;

1. **Output dependencies**

This occurs when i and j write to the same register or memory location. The ordering must be preserved to ensure that value finally written corresponds to j.

i: a=b+10;

j: a=c\*10;

Since a named dependence is not a true dependence, instructions involved in a name dependence can be executed simultaneously or be reordered if the name used in the instructions changed such that the instructions do not conflict.

* + - 1. **Control dependencies**

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.

if p1

{

S1;

}

if p2

{

S2;

}

Here s1 is control dependent on p1 and s2 is control dependent on p2 but not on p1.

In general, there are 2 constraints imposed by control dependencies

1. 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.
2. An instruction that is not control dependent on a branch cannot be moved inside the branch so that its execution is controlled by the branch.

**TWO** Properties that are critical to program correctness are

1. **Exception behavior**

Preserving Exception behavior means that any changes in the ordering of instruction execution must not change how exceptions are raised in the program.

DADDU R2,R3,R4

BEQZ R2,L1

LW R1,0(R2)

L1:

* Result will change if we do not maintain data dependence involving R2.
* If we ignore the control dependence and move the load instruction before the branch, the load instruction may cause memory protection exception.

1. **Data flow**

* The data flow is the actual flow of data values between instructions that produce results and those that consume them.
* Branches make the data flow dynamic.
* It is not sufficient to just maintain the data dependences because an instruction may be data dependent on more than one predecessor.
* Program order will actually deliver data value to an instruction. Program order is ensured by maintaining the control dependence.

DADDU R1,R2,R3

BEQZ R4,L

DSUBU R1,R5,R6

L:….

OR R7,R1,R8

* The value of R1 used by OR instruction depends on whether the branch is taken or not.
* The OR instruction is data dependent on both DADDU and DSUBU instructions, but preserving that order alone is insufficient for correct execution.
* If the branch is not taken, then the value of R1 computed by DSUBU should be used by OR And if the branch is taken, then the value of R1 computed by DADDU should be used by OR.
* By preserving the control dependence of the OR on the branch, we prevent an illegal change to the data flow.

DADDU R1,R2,R3

BEQZ R12,skipnext

DSUBU R4,R5,R6

DADDU R5,R4,R9

skipnext: OR R7,R8,R9

* Sometimes we can determine that violating control dependence cannot affect either the exception behavior or the data flow.
* Suppose we knew that R4 in DSUBU was unused after the instruction labeled skipnext, then changing the value of R4 before the branch would not affect the dataflow since R4 would be dead in the code region after skipnext.

**HAZARDS**

* There are situations called hazards that prevent the next instruction in the instruction stream from executing, during its designated clock cycle.
* Hazards reduce the performance from the ideal speed up gained by pipelining.

**Data hazards**

* A hazard is created whenever there is a dependence between instructions and they are close enough that the overlap during execution would change the order of access to the operand involved in the dependence.
* Because of dependence, we must preserve what is called as program order i.e. the order that instructions would execute in, if executed sequentially one at a time as determined by the original source program.
* Consider two instructions i and j with i preceding j in program order. The possible data hazards are

1. **Read After Write(RAW)**

* j tries to read a source before i writes it so j gets the old incorrect value.

i: a=b+10;

j: c=a/d;

Program order must be preserved to ensure that j receives the value from i.

1. **Write After Write(WAW)**

* 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.

i: a=b+10;

j: a=b\*10;

1. **Write After Read(WAR)**

* j tries to write to destination before it is read by i, so I incorrectly get the new value.

i: a=b+10;

j: b=c\*10;

**Read After Read(RAR)**

* This case is not a hazard

i: a=c+10;

j: b=c/20;

**Register Renaming**

* If you are renaming a register operand it is called **register renaming**.
* Register renaming can be done either statically by a compiler or dynamically by the hardware.

i: a=b+10;

j: b=c\*10;

k: a=b+5;

the above instructions after register renaming:

i: a=b+10;

j: b1=c\*10;

k: a1=b1+5;