

Data Dependence

- Loop: L.D F0,0(R1)
- ADD.D F4,F0,F2
- S.D F4,0(R1)
- DADDUI R1,R1,#-8
- BNE R1,R2,Loop

Compiler Techniques for Exposing ILP

- Pipeline scheduling
 - Separate dependent instruction from the source instruction by the pipeline latency of the source instruction
- Example:


```
for (i=999; i>=0; i=i-1)
    x[i] = x[i] + s;
```

No dependence
between iterations
MIPS code?

Instruction producing result	Instruction using result	Latency in clock cycles
FP ALU op	Another FP ALU op	3
FP ALU op	Store double	2
Load double	FP ALU op	1
Load double	Store double	0

Assembly Code

```
for(i=1000;i>0;i--)
    x[i]=x[i]+A
```

- | | | | |
|-------|--------|------------|---|
| Loop: | L.D | F0,0(R1) | 1 |
| | ADD.D | F4,F0,F2 | 2 |
| | S.D | F4,0(R1) | 3 |
| | DADDUI | R1,R1,#-8 | 4 |
| | BNE | R1,R2,Loop | 5 |

Loop Unrolling/Pipeline Scheduling

- Pipeline schedule the unrolled loop:

```
Loop:  L.D   F0,0(R1)
      L.D   F6,-8(R1)
      L.D   F10,-16(R1)
      L.D   F14,-24(R1)
      ADD.D F4,F0,F2
      ADD.D F8,F6,F2
      ADD.D F12,F10,F2
      ADD.D F16,F14,F2
      S.D   F4,0(R1)
      S.D   F8,-8(R1)
      DADDUI R1,R1,#-32
      S.D   F12,16(R1)
      S.D   F16,8(R1)
      BNE  R1,R2,Loop
```

Loop iterations are independent



Strip Mining

- Unknown number of loop iterations?
 - Number of iterations = n
 - Goal: make k copies of the loop body
 - Generate pair of loops:
 - First executes $n \bmod k$ times
 - Second executes n / k times
 - "Strip mining"



Loop Level Parallelism

- Loop-Level Parallelism (LLP) analysis focuses on whether data accesses in later iterations of a loop are data dependent on data values produced in earlier iterations and possibly making loop iterations independent.

```
For(i=0;i<100;i++)
  x[i]=x[i]+A;
```

- the computation in each iteration is independent of the previous iterations and the loop is thus parallel. The use of $X[i]$ twice is within a single iteration.
 - Thus loop iterations are parallel (or independent from each other).

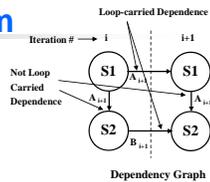


Loop Level Parallelsim

- **Loop-carried Dependence:** A data dependence between different loop iterations (data produced in earlier iteration used in a later one).
- LLP analysis is important in software optimizations such as loop unrolling since it usually requires loop iterations to be independent.
- LLP analysis is normally done at the source code level or close to it since assembly language and target machine code generation introduces loop-carried name dependence in the registers used for addressing and incrementing.
- Instruction level parallelism (ILP) analysis, on the other hand, is usually done when instructions are generated by the compiler

Loop Level Parallism

```
for (i=1; i<=100; i=i+1) {
    A[i+1] = A[i] + C[i]; /* S1 */
    B[i+1] = B[i] + A[i+1]; /* S2 */
}
```



- S2 uses the value A[i+1], computed by S1 in the same iteration. This data dependence is within the same iteration (not a loop-carried dependence).
 - = does not prevent loop iteration parallelism.
- S1 uses a value computed by S1 in an earlier iteration, since iteration i computes A[i+1] read in iteration i+1 (loop-carried dependence, prevents parallelism). The same applies for S2 for B[i] and B[i+1]
 - = These two dependencies are loop-carried spanning more than one iteration preventing loop parallelism.

Loop Level parallelism

- for(i=0; i<=100; i++)
- A[i] = A[i] + B[i]; /* S1 */
- B[i+1] = C[i] + D[i]; /* S2 */
- S1 uses the value calculated by S2 in the previous iteration (loop carried dependence)
- The dependence is not circular, S2 does not depend on S1 in the previous iteration

GCD test

- A simple and **sufficient** test for absence can be found.
- If a loop dependence exists, then

$$\text{GCD}(c, a) \text{ divides } (d - b)$$

GCD Test -- Example

```
for(i=1; i<=100; i=i+1) {  
    x[2*i+3] = x[2*i] * 5.0;  
}
```

a = 2 b = 3 c = 2 d = 0

GCD(a, c) = 2

d - b = -3

2 does not divide -3 ⇒ No dependence is not possible.

5,7,9,11,13,15,17,19,21,23,....

4,6,8,10,12,14,16,18,20,22,....

Dependence Analysis

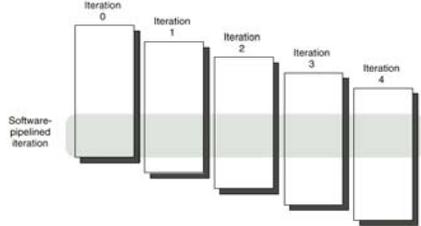
- Dependence analysis is a very important tool for exploiting LLP, it can not be used in these situations
- Objects are referenced using pointers
- Array indexing using another array $a[b[i]]$
- Dependence may exist for some values of input, but in reality the input never takes these values.
- When we want to know more than the possibility of dependence (which write causes it?)
- Dependence analysis across procedure boundaries

Dependence Analysis

- Sometimes, *points-to* analysis might help.
- We might be able to answer *simpler* questions, or get some hints.
- Do 2 pointers point to the same list?
- Type information
- Information derived when the object was allocated
- Pointer assignments

Software Pipelines

- Software pipelined loop chooses instructions from different loop iterations, thus separating the dependent instructions within one iteration of the original loop



Software Pipelines

```

Loop:  L.D   F0,0(R1)
       ADD.D F4,F0,F2
       S.D   F4,0(R1)
       DADDUI R1,R1,#-8
       BNE
    
```

Before: Unrolled 3 times

```

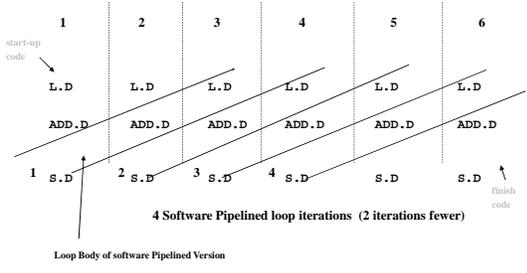
1 L.D   F0,0(R1)
2 ADD.D F4,F0,F2
3 S.D   F4,0(R1)
4 L.D   F0,-8(R1)
5 ADD.D F4,F0,F2
6 S.D   F4,-8(R1)
7 L.D   F0,-16(R1)
8 ADD.D F4,F0,F2
9 S.D   F4,-16(R1)
10 DADDUI R1,R1,#-24
11 BNE   R1,R2,LOOP
    
```

After: Software Pipelined Version

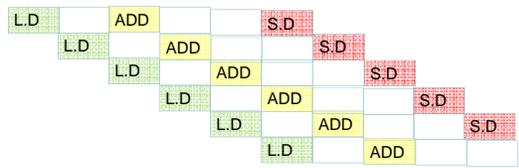
```

1 L.D   F0,0(R1)
2 ADD.D F4,F0,F2
3 S.D   F4,0(R1)
4 L.D   F0,-8(R1)
5 S.D   F4,0(R1) ;Stores M[i]
6 ADD.D F4,F0,F2 ;Adds to M[i-1]
7 L.D   F0,-16(R1);Loads M[i-2]
8 DADDUI R1,R1,#-8
9 BNE   R1,R2,LOOP
10 S.D   F4,0(R1)
11 ADD.D F4,F0,F2
12 S.D   F4,-8(R1)
    
```

Software Pipelines



Software pipelines



```
Loop: L.D F0,0(R1)
      ADD.D F4, F0, F2
      S.D F4,0(R1)
      DADD R1,R1,#-8
      BNE R1,R2,Loop
```

```
Loop: S.D F4,16(R1)
      ADD.D F4, F0, F2
      L.D F0,0(R1)
      DADD R1,R1,#-8
      BNE R1,R2,Loop
```
