LOOP UNROLLING

Instruction-Level Parallelism

- Pipelining ?

Instruction-Level Parallelism

- Pipelining ?

- improve performance.
- overlapping the execution of instructions.

Instruction-Level Parallelism

- Pipelining ?

- improve performance.
- overlapping the execution of instructions.
- This potential overlap among instructions is called instruction-level parallelism (ILP) since the instructions can be evaluated in parallel.

Basic Pipeline Scheduling and Loop Unrolling

- To keep a pipeline full, parallelism among instructions must be exploited by finding sequences of unrelated instructions that can be overlapped in the pipeline.

- To avoid a pipeline stall, a dependent instruction must be separated from the source instruction by a distance in clock cycles equal to the pipeline latency of that source instruction.

- Loop unrolling: replicates the loop body multiple times and adjust the loop termination code.

Running Example

- This code adds a scalar to an array/vector:

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

- Assembly Language:

```
Loop:  L.D      F0, 0 (R1)     ;  load the vector element
       ADD.D    F4, F0, F2     ;  add the scalar in F2
       S.D      0 (R1), F4     ;  store the vector element
       DAADUI   R1, R1, #-8    ;  decrement the pointer by
                               ;  8 bytes (per  DW)
       BNE      R1, R2, Loop   ;  branch R1 != R2
```

- Register R1: contains the last element of array/vector
- Register R2: is precomputed and 8(R2) contains the first element of the array.

An Example

Assume following latency all examples

Instruction Producer	Instruction Consumer	Latency
FP ALU op	FP ALU op	3
FP ALU op	Store Double	2
Load Double	FP ALU op	1
Load Double	Store Double	**0**

- Latency: number of intervening cycles between an instruction that produces a result and instruction that uses the result
- Assume that latency for Integer ops is **zero** and latency for Integer load is **1**

An Example

```
Loop:   L.D       F0, 0 (R1)          1
        STALL                         2
        ADD.D     F4, F0, F2          3
        STALL                         4
        STALL                         5
        S.D       0 (R1), F4          6
        DAADUI    R1, R1, #-8         7
        STALL                         8
        BNE       R1,  R2, Loop       9
```

This requires ***9 Cycles*** per iteration

An Example

Scheduling

```
Loop: L.D       F0, 0 (R1)       1
      DADDUI    R1, R1, #-8      2
      ADD.D     F4, F0, F2       3
      STALL                      4
      STALL                      5
      S.D       8 (R1), F4       6
      BNE       R1, R2 Loop      7
```

This requires ***7 Cycles*** per iteration

An Example

Unrolling

```
1  Loop:    L.D       F0,0(R1)
2           ADD.D     F4,F0,F2
3           S.D       0(R1),F4          ;drop DSUBUI & BNE
4           L.D       F6,-8(R1)
5           ADD.D     F8,F6,F2
6           S.D       -8(R1),F8          ;drop DSUBUI & BNE
7           L.D       F10,-16(R1)
8           ADD.D     F12,F10,F2
9           S.D       -16(R1),F12        ;drop DSUBUI & BNE
10          L.D       F14,-24(R1)
11          ADD.D     F16,F14,F2
12          S.D       -24(R1),F16
13          DADDUI    R1,R1,#-32
14          BNE       R1, R2,LOOP
```

Total Cycles=14+13=27 (L.D ->1 stall, ADD.D-> 2 stall, DADDUI->1 stall).
This requires ***6.75 Cycles*** per iteration (27/4)

Unrolling + Scheduling

An Example

```
Loop :  L.D       F0, 0 (R1)          1
        L.D       F6,   -  8 (R1)     2
        L.D       F10, -16 (R1)       3
        L.D       F14, -24 (R1)       4
        ADD.D     F4, F0, F2          5
        ADD.D     F8, F6, F2          6
        ADD.D     F12, F10, F2        7
        ADD.D     F16, F14, F2        8
        S.D       0 (R1), F4          9
        S.D       -8 (R1), F8         10
        DADDUI    R1, R1, #-32        11
        S.D       16 (R1), F12        12
        S.D       8 (R1), F16         13
        BNEZ      R1, R2 LOOP         14
```

This requires ***3.5 Cycles*** per iteration. (14/4)

Loop unrolling (Contd…)

- Suppose the upper bound is *n*, and we want *k* copies of the loop.
- Don't generate a single unrolled loop!
- Generate a pair of consecutive loops:
  - First executes the original loop (n mod k)
  - Second executes the unrolled loop body (n/k) times
  - For large n, most iterations occur in unrolled loop
- This technique is similar to a technique called Strip mining.

Summary of loop unrolling and scheduling:

- Eliminate the extra test and branch instructions and adjust the loop termination and iteration code.
- Code length increases with additional unrolling. This is issue for embedded processors. This can increase instruction cache miss rates.
- Uses lots of registers. Renaming requires many registers. When registers become scarce: “register pressure”.
- Aggressive unrolling and scheduling and cause a compiler to run out of registers to use for renaming.