Advanced Pipelining and ILP

Instruction-level parallelism (ILP)

The potential overlap among instructions is called **ILP**.

It implies a lack of dependence between instructions.

Earlier we considered pipelines that allow us to overlap **independent** instructions.

Next step is to try and exploit parallelism among *instruction sequences*.

Advanced Pipelining and ILP

Instruction-level parallelism (ILP)

• *Exploiting ILP*

We will discuss methods for exploiting any ILP that may exist in a program.

this can mean:

- Executing instructions *out of order* and
- Other techniques that keep the functional units busy with useful work.

• *Limits to ILP*

We'll also discuss things that limit the amount of ILP that a processor can exploit.

This limitation can come from either:

- The processor (i.e., limited number of functional units) or
- The program (every instruction depends on the previous one).

Advanced Pipelining and ILP

CPI revisited:

Pipeline CPI = Ideal pipeline CPI + Structural stalls + RAW stalls + WAR stalls + WAW stalls + Control stalls

The focus of basic pipelining was on reducing **RAW** and **control** stalls.

more advanced ILP techniques which further reduce RAW and control stalls typically *increase* the importance of dealing with WAR, WAW and structural stalls.

Techniques include

- *Loop unrolling*

  Reduces **control** stalls.

- *Basic pipeline scheduling*

  Reduces **RAW** stalls.

- *Scoreboarding*

  Reduces **RAW** stalls.

ILP Techniques

- *Register renaming*

  Reduces both **WAR** and **WAW** stalls.

- *Dynamic branch prediction*

  Reduces **control** stalls.

- *Issuing multiple instructions per cycle (superscalar)*

  Reduces **ideal CPI** by allowing more than one instruction to start each cycle.

- *Compiler dependence analysis*

  Reduces **ideal CPI** and **data** stalls by having the compiler perform code scheduling.

ILP Techniques

- *Software pipelining and trace scheduling*

  Reduces **ideal CPI** and **data** stalls by using previous executions to tune future executions.

- *Speculation*

  Execute "possible" instructions so that, whichever way the program "really" goes, the CPU will have the proper state.

  This reduces all **data** and **control** stalls.

- *Dynamic memory disambiguation*

  Reduce **RAW** stalls involving memory.

ILP Techniques

What is ILP, and where does it come from?

Basic blocks

A block of code with **no** branches *into* the code except at the start and **no** branches *out* of the code except at the end.

The code inside the average basic block is quite small.

Last chapter, we saw the average dynamic branch frequency in integer programs was about **15%**.

This means that between 6 and 7 instructions are executed between a pair of branches.

It is likely that these instructions depend on one another since the instructions tend to operate on the same data.

Therefore, we must exploit ILP **across multiple basic blocks**.

ILP Techniques

- *Loop-level parallelism*

One of the simplest and most common ways to increase parallelism is to exploit it among iterations of a loop.

In many loops, the iterations of a loop are independent, i.e.

```
for (i = 0; i < 1000; i++) {
   d[i] = x[i] * y[i] + z[i];
}
```

Each iteration can overlap with any other iteration even though individual iterations have few (if any) overlappable instructions.

Many techniques exist for exploiting the ILP in loops.

Some are done *statically* by the compiler (loop unrolling) and some are done *dynamically* by the CPU.

In addition, vector processors can be run very quickly on simple loop operations.

ILP Techniques

- *Pipeline scheduling*

Compiler seeks to separate a *dependent* instruction from the *source* instruction by a distance (in clk cycles) equal to the pipeline latency of the source.

To do this, the compiler must have intimate knowledge of the internal hardware workings.

This is one reason that code written for (say) the Intel 486 may not run optimally on the Pentium — the pipeline latencies have likely changed.

Let's assume the following latencies (for illustration):

Instruction producing result	Instruction using result	Latency in clk 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 (forwarded to store)

ILP Techniques

- *Pipeline scheduling*

Given the knowledge of pipeline latencies, the compiler can reorder the instructions so that stalls are avoided,

```
for (i = 1000;  i > 0;  i = i - 1)
    x[i] = x[i] + c;
```

Assume R1 holds the address of the element in array with highest address. AND x[1] (which has the lowest address) is at address 8 (if not, 1 more comparison would be required). Also assume F2 contains the scaler value, c.

Unscheduled DLX assembly:

```
Loop: LD F0, 0(R1)          ;F0=array element
      ADDD F4, F0, F2       ;add scalar in F2
      SD 0(R1), F4          ;Store result
      SUBI R1, R1, #8       ;Decrement pointer (8 bytes per DW).
      BNEZ R1, Loop         ;branch R1 != zero
```

ILP Techniques

- *Pipeline scheduling*

Unscheduled analysis:

	Clock cycle issued
Loop: LD F0, 0(R1)	1
stall	2
ADDD F4, F0, F2	3
stall	4
stall	5
SD 0(R1), F4	6
SUBI R1, R1, #8	7
stall	8 (since branch reads operand in ID)
BNEZ R1, Loop	9
stall	10 (delayed branch)

When we execute this code, it takes 10 cycles per iteration.

ILP Techniques

- *Pipeline scheduling*

Unscheduled/Scheduled code:

```
Loop: LD F0, 0(R1)        ;F0=array element
      ADDD F4, F0, F2     ;add scalar in F2
      SD 0(R1), F4        ;Store result
      SUBI R1, R1, #8     ;Decrement pointer (8 bytes per DW).
      BNEZ R1, Loop       ;branch R1 != zero
```

Scheduled

```
Loop: LD F0, 0(R1)        ;F0=array element
      SUBI R1, R1, #8     ;Decrement pointer (8 bytes per DW).
      ADDD F4, F0, F2     ;add scalar in F2
      stall
      BNEZ R1, Loop       ;branch R1 != zero
      SD 8(R1), F4        ;delay slot filler: altered and interchanged with SUBI.
```

Stall because latency is 2

Execution time is reduced from 10 cycles to 6.

Note that the compiler modified the effective address for SD.

ILP Techniques

- *Pipeline scheduling & loop unrolling*

We knocked the cycle count from 10 down to 6, but we really only did 3 cycles of work:

A load, an add, and a store.

The rest of the loop was **overhead**, i.e., SUBI, BNEZ and a stall.

While it may seem we actually did work by adding to the i index, that's not really work since it didn't produce any results in memory.

We need to get more operations into the loop relative to the number of overhead instructions (branch, loop counter increment/decrement, etc.)

This is done by *loop unrolling*: **replicating** the loop body multiple times and adjusting the loop termination code.

ILP Techniques

Pipeline scheduling & loop unrolling

Loop unrolling involves creating *multiple copies* of the loop body.

It improves scheduling because:

- It eliminates branches.
- It allows instructions from different iterations to be scheduled together, it exposes parallelism.

This allows the CPU to amortize the cost of updating indices across several iterations of the loop.

Loop unrolling also increases register usage.

Basic ILP Techniques

Pipeline scheduling & loop unrolling

Previous code unrolled 4 times and scheduled (assuming R1 was initially a multiple of 32).

```
Loop: LD F0, 0(R1)
      LD F6, -8(R1)
      LD F10, -16(R1)
      LD F14, -24(R1)
      ADDD F4, F0, F2
      ADDD F8, F6, F2
      ADDD F12, F10, F2
      ADDD F16, F14, F2
      SD 0(R1), F4
      SD -8(R1), F8
      SUBI R1, R1, #32
      SD 16(R1), F12
      BNEZ R1, Loop
      SD 8(R1), F16
```

This code has NO stalls !

Execution time is **14** clock cycles or **3.5** clks per element.

This is an improvement over **6** clks for the unrolled scheduled code.

The use of different registers avoids *false dependencies*.
Reordering the instructions avoids stalls.

Note that displacement addressing mode makes it possible to keep the overhead very low. **Assume unrolling k (>4) times, what is # of clocks per element ?**

$3 + 2/k$

For $k = 4$, clocks per element $= 3 + 2/4 = 3.5$
why ?

Assuming code size is not a limiting factor, why not unroll the loop a large numbr of times (say 100 times, i.e., use $k = 100$) to drive down the number of clocks per element toward the lower limit (what is this lower limit ?)

Only the SUBI and BNEZ are loop-overhead instructions, every other xtn is doing useful work (load, add or store). this overhead remains constant. assuming the code can be properly scheduled without any stalls, the cycles required per element

= 3 (to do the useful work: load, add, store) +
2/k (overhead amortized over 4k elements)

Even if code size is not a problem, there should be enough number of registers available to support the execution of the unrolled code on the pipeline without any stalls. The number of GPRs is typically not very large
(at best O(10^2), DLX has 32 Integer and 32 FP GPRs)

Exercise: derive the number of registers required to support k fold unrolling without any stalls. (For this particular example, number of registers is not a limiting factor, but in general it can be). Note that without unrolling, aggressive scheduling is limited by branches so register pressure is typically not an issue.

ILP Techniques

Pipeline scheduling & loop unrolling

In real code, we may not know the upper bound on the loop.(ex: while loops)

We don't know how many times the loop will be executed (for loop but iteration-count passed as an argument at run-time)

Suppose it is **n** and we want to unroll the loop **k** times, Express n as :

n = **floor (**n/k) + n **mod** k (this is similar to basic division X = Q . D + R)

We create *two* consecutive loops:

The first one contains the original code and executes (**n mod k**) times.

The second one contains the unrolled body and executes **n/k** times.

Can the positions of the first (orig) and second (unrolled) loop be swapped ?

If the number of iterations is large, this method saves the CPU lots of time at the cost of larger code size.

Although it was easy for us to recognize these transformations, have a compiler do it and guarantee that the transformed code is correct is **not** trivial.