Instruction Level Parallelism (ILP)

or
Declaration of Independence

What is ILP?

- The characteristic of a program that certain instructions are ***independent***, and can potentially be ***executed in parallel***.
- Any mechanism that creates, identifies, or exploits the independence of instructions, allowing them to be executed in parallel.

- Why do we want/need ILP?
  - In a superscalar architecture?
  - What about a scalar architecture?

Where do we find ILP?

- In basic blocks?
  - 15-20% of (dynamic) instructions are branches in typical code
- Across basic blocks?
  - how?

```
for (i=1; i<=1000; i++)
    x[i] = x[i] * s
```


How do we expose ILP?

- by moving instructions around.
- How??
  - software
  - hardware

Exposing ILP in software

- instruction scheduling (changes ILP within a basic block)
- loop unrolling (allows ILP across iterations by putting instructions from multiple iterations in the same basic block)
- Others (trace scheduling, software pipelining) we'll talk about later…

A sample loop

```
Loop:   LD      F0,0(R1)        ;F0=array element, R1=X[]
        MULD    F4,F0,F2        ;multiply scalar in F2
        SD      F4, 0(R1)       ;store result
        ADDI    R1,R1,8         ;increment pointer 8B (DW)
        SEQ     R3, R1, R2      ;R2 = &X[1001]
        BNEZ    R3,Loop         ;branch R3!=zero
        NOP                     ;delayed branch slot
```

Where are the dependencies and stalls?

Operation	***Latency (stalls)***
FP Mult	***6 (5)***
LD	***2 (1)***
Int ALU	***1 (0)***

Instruction Scheduling

```
Loop:   LD      F0,0(R1)
        MULD    F4,F0,F2
        SD      0(R1),F4
        ADDI    R1,R1,8
        SEQ     R3, R1, R2
        BNEZ    R3,Loop
        NOP
```

```
Loop:   LD      F0,0(R1)
        ADDI    R1,R1,8
        MULD    F4,F0,F2
        SEQ     R3, R1, R2
        BNEZ    R3,Loop
        SD      -8(R1),F4
```


Loop Unrolling

```
Loop:   LD      F0,0(R1)
        ADDI    R1,R1,8
        MULD    F4,F0,F2
        SEQ     R3, R1, R2
        BNEZ    R3,Loop
        SD      -8(R1),F4
```

```
Loop:   LD      F0,0(R1)
        ADDI    R1,R1,8
        MULD    F4,F0,F2
        SEQ     R3, R1, R2
        BNEZ    R3,Loop
        SD      -8(R1),F4
        LD      F0,0(R1)
        ADDI    R1,R1,8
        MULD    F4,F0,F2
        SEQ     R3, R1, R2
        BNEZ    R3,Loop
        SD      -8(R1),F4
```

Loop Unrolling

```
Loop:   LD      F0,0(R1)
        ADDI    R1,R1,8
        MULD    F4,F0,F2
        SEQ     R3, R1, R2
        BNEZ    R3,Loop
        SD      -8(R1),F4
```

```
Loop:   LD      F0,0(R1)
        ADDI    R1,R1,8
        MULD    F4,F0,F2
        SEQ     R3, R1, R2
        BNEZ    R3,Loop
        SD      -8(R1),F4
        LD      F0,0(R1)
        ADDI    R1,R1,8
        MULD    F4,F0,F2
        SEQ     R3, R1, R2
        BNEZ    R3,Loop
        SD      -8(R1),F4
```


Loop Unrolling

```
Loop:   LD      F0,0(R1)
        ADDI    R1,R1,8
        MULD    F4,F0,F2
        SEQ     R3, R1, R2
        BNEZ    R3,Loop
        SD      -8(R1),F4
```

```
Loop:   LD      F0,0(R1)
        MULD    F4,F0,F2
        SD      0(R1),F4
        LD      F0,8(R1)
        ADDI    R1,R1,16
        MULD    F4,F0,F2
        SEQ     R3, R1, R2
        BNEZ    R3,Loop
        SD      -8(R1),F4
```


Register Renaming

```
Loop:   LD      F0,0(R1)
        ADDI    R1,R1,8
        MULD    F4,F0,F2
        SEQ     R3, R1, R2
        BNEZ    R3,Loop
        SD      -8(R1),F4
```

```
Loop:   LD      F0,0(R1)
        MULD    F4,F0,F2
        SD      0(R1),F4
        LD      F10,8(R1)
        ADDI    R1,R1,16
        MULD    F14,F10,F2
        SEQ     R3, R1, R2
        BNEZ    R3,Loop
        SD      -8(R1),F14
```


Register Renaming

```
Loop:   LD      F0,0(R1)
        ADDI    R1,R1,8
        MULD    F4,F0,F2
        SEQ     R3, R1, R2
        BNEZ    R3,Loop
        SD      -8(R1),F4
```

```
Loop:   LD      F0,0(R1)
        LD      F10,8(R1)
        MULD    F4,F0,F2
        MULD    F14,F10,F2
        ADDI    R1,R1,16
        SEQ     R3, R1, R2
        SD      0(R1),F4
        BNEZ    R3,Loop
        SD      -8(R1),F14
```

Compiler Perspectives on Code Movement

- Remember: *dependencies* are a property of code, whether or not it is a HW *hazard* depends on the given pipeline.
- Compiler must respect (*True*) Data dependencies (RAW)
  - Easy to determine for registers (fixed names)
  - Hard for memory:
    - Does 100(R4) = 20(R6)?
    - From different loop iterations, does 20(R6) = 20(R6)?
- False dependences (WAR and WAW) can sometimes be overcome.

Compiler Perspectives on Code Movement

- Compilers must also preserve *control dependence*
- Example

```
if (c1)
      I1;
if (c2)
      I2;
```

I1 is control dependent on c1 and I2 is control dependent on c2 but not on c1.

Compiler Perspectives on Code Movement

- Two (obvious) constraints on control dependences:
  - 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.
  - An instruction that is not *control dependent* on a branch cannot be moved to *after* the branch so that its execution is controlled by the branch.
- Control dependencies relaxed to get parallelism; as long as we get same effect if preserve order of exceptions and data flow

Code Motion

- Can be done in SW or HW
- Why SW?
- Why HW?

- Also, like software, we'd like the following capabilities in our hardware code motion.
  - Ability to move instructions across branches
  - Ability to overcome (or ignore) false dependences
  - Both easier in hardware

HW Schemes: Instruction Parallelism

- Why in HW at run time?
  - Works when can't know dependence until run time
    - Variable latency
    - Control dependent data dependence
  - Can schedule differently every time through the code.
  - Compiler simpler
  - Code for one machine runs well on another
- Key idea: Allow instructions behind stall to proceed

```
DIVD  F0,F2,F4
ADDD  F10,F0,F8
SUBD  F12,F8,F14
```

  - Enables out-of-order execution => out-of-order completion

First HW ILP Technique: Out-of-order Issue/Dynamic Scheduling

- Problem -- need to get stalled instructions out of the ID stage, so that subsequent instructions can begin execution.
- Must separate detection of structural hazards from detection of data hazards
- Must split ID operation into two:
  - Issue (decode, check for structural hazards)
  - Read operands (read operands when NO DATA HAZARDS)
  - Otherwise, one stalled (for data) instruction would cause all others to back up behind the ID stage.
- i.e., must be able to issue even when a data hazard exists
- instructions *issue* in-order, but proceed to EX out-of-order

Dynamic Scheduling by hand

	in-order	out-of-order
DIVD F0,F2,F4 (10 cycles)		
ADDD F10, F0, F8 (4 cycles)		
SUBD F12, F8, F14 (4 cycles)		
ADDD F20,F2,F3		
MULTD F13,F12,F2 (6 cycles)		
ADDD F4,F1,F3		
ADDD F5,F4,F13		

(assume several FP ADD units)

Key Points

- You can find, create, and exploit Instruction Level Parallelism in SW or HW
- Loop level parallelism is usually easiest to see
- Dependencies exist in a program, and become hazards if HW cannot resolve
- SW dependencies/compiler sophistication determine if compiler can/should unroll loops
- SW code motion is limited by lack of runtime knowledge of dependencies (esp. memory), latencies (esp. memory), and control flow.