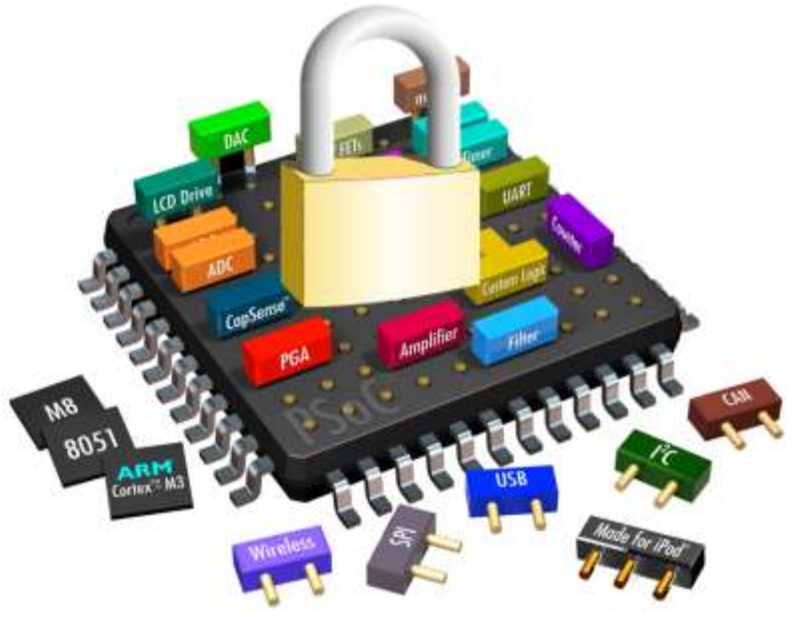
KOREA UNIVERSITY

Advanced Computer Architecture

Instruction-Level Parallelism and Its Exploitation



KOREA UNIVERSITY

Understanding Instruction-Level Parallelism

All processors since about 1985 use pipelining to overlap the execution of instructions and improve performance. This potential overlap among instructions is called *instruction-level parallelism* (ILP), since the instructions can be evaluated in parallel.

Dynamic Approach

Relies on hardware to discover and exploit parallelism dynamically at runtime

Dominates desktop and server markets (Intel Core series)

Static Approach

Relies on software technology to find parallelism at compile time

Common in personal mobile devices where energy efficiency is key (ARM Cortex-A8)

Despite enormous efforts, aggressive compiler-based approaches have not been successful outside of scientific applications. Understanding ILP limitations remains important in balancing ILP and thread-level parallelism.

Loop-Level Parallelism

The amount of parallelism available within a *basic block* (straight-line code sequence with no branches except at entry/exit) is quite small. For typical MIPS programs, the average dynamic branch frequency is between 15% and 25%, meaning only three to six instructions execute between branches.

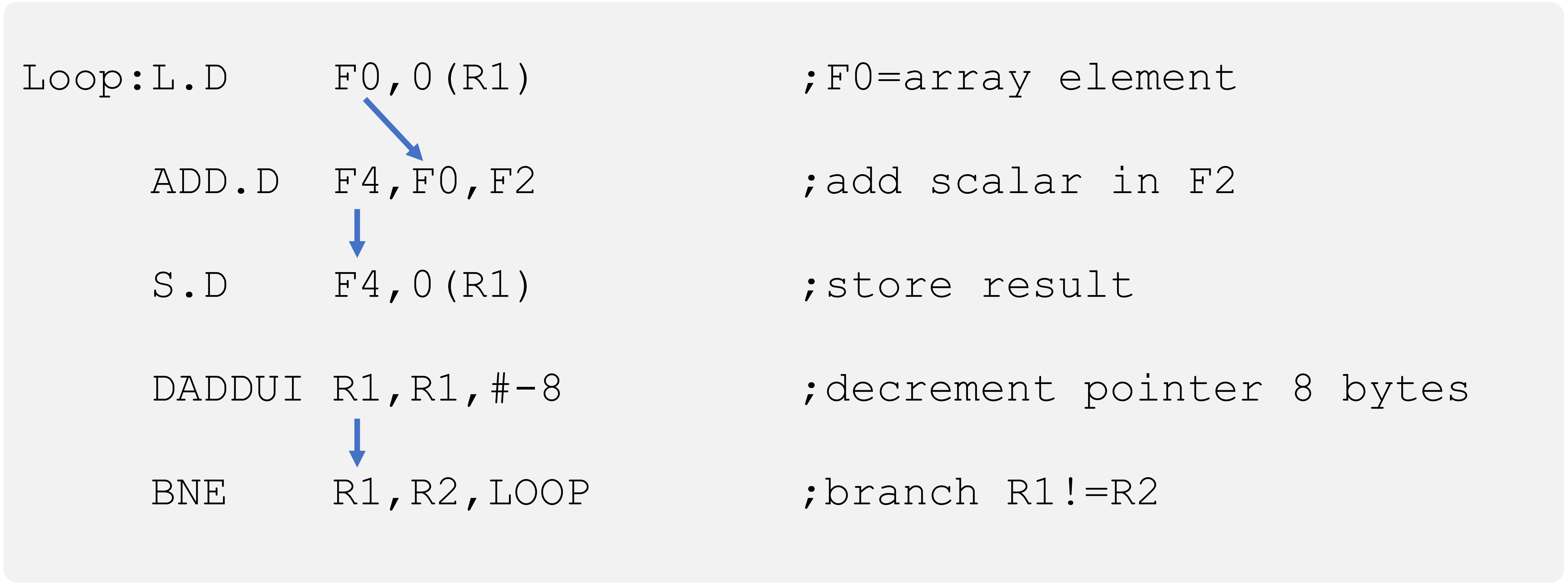
The simplest way to increase ILP is to exploit parallelism among iterations of a loop, called *loop-level parallelism*.

```
for (i=0; i<=999; i=i+1)
    x[i] = x[i] + y[i];
```

Every iteration of this loop can overlap with any other iteration. We can convert loop-level parallelism into instruction-level parallelism by unrolling the loop either statically by the compiler or dynamically by the hardware.

An important alternative method for exploiting loop-level parallelism is using SIMD in both vector processors and GPUs, which can dramatically reduce instruction count by processing multiple data items in parallel.

Data Dependences



```
Loop:L.D     F0,0(R1)              ;F0=array element
     ADD.D   F4,F0,F2              ;add scalar in F2
     S.D     F4,0(R1)              ;store result
     DADDUI  R1,R1,#-8             ;decrement pointer 8 bytes
     BNE     R1,R2,LOOP            ;branch R1!=R2
```

Name Dependences

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

Antidependence

```
Read X
Write X
```

Output Dependence

```
Write X
Write X
```

- Because a name dependence is not a true dependence, instructions involved in a name dependence can execute simultaneously or be reordered, if the name (register number or memory location) used in the instructions is changed so the instructions do not conflict.

Data Hazards

A hazard exists when there is a name or data dependence between instructions close enough that their overlap during execution would change the order of access to the operand. We must preserve program order where it affects program outcome.

Suppose instruction i is executed before instruction j.

RAW (Read After Write)

j tries to read a source before i writes it, getting the old value

Most common hazard, corresponds to true data dependence

WAW (Write After Write)

j tries to write an operand before i writes it

Corresponds to output dependence

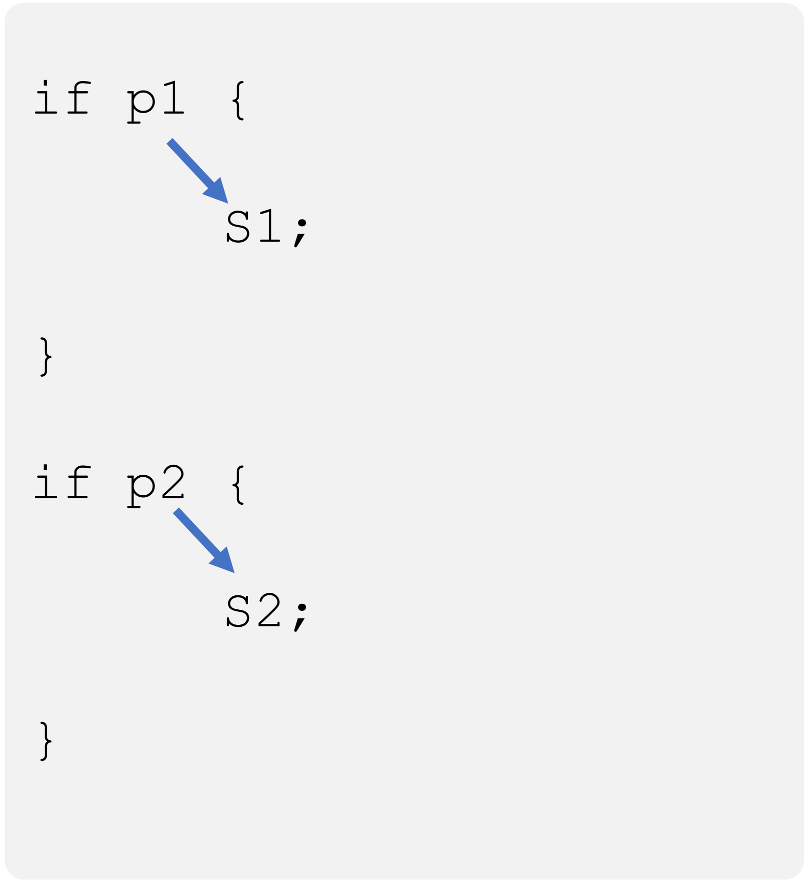
WAR (Write After Read)

j tries to write a destination before it's read by i

Arises from antidependence (name dependence)

Control Dependences

- A control dependence determines the ordering of an instruction, i, with respect to a branch instruction so that instruction i is executed in correct program order and only when it should be.



```
if p1 {

        S1;

}

if p2 {

        S2;

}
```