

# **Instruction level Paralism**

**Presented By**

**Dr. Banchhanidhi Dash**

**School of Computer Engineering  
KIIT University**

# Instruction-Level Parallelism

80

- **Instruction-level parallelism (ILP)** is a measure of how many of the operations in a computer program can be performed simultaneously.

Consider the following program:

- For Example:
  1.  $e = a + b$
  2.  $f = c + d$
  3.  $g = e * f$

Here, **Operation 3 depends on the results of operations 1 and 2**, so it cannot be calculated until both of them are completed. As, operations 1 and 2 do not depend on any other operation, so they can be calculated simultaneously.

If each operation is completed in one unit of time then three instructions can be completed in two units of time, giving an ILP of 3/2.

## What is instruction Level Parallelism (ILP)?

- Inherent property of a sequence of instructions, as a result of which some instructions can be allowed to execute in parallel.
- Note that this definition implies parallelism across a sequence of instructions (block). This could be a loop, a conditional, or some other valid sequence of statements.
- There is an upper bound, as to how much parallelism can be achieved, since by definition parallelism is an inherent property of the sequence of instructions.
- We can approach this upper bound via a series of transformations that either expose or allow more ILP to be exposed to later transformations.

## What is instruction Level Parallelism (ILP)?

- Dependencies within a sequence of instructions determine how much ILP is present. Think of this as:

To what degree can we rearrange the instructions without compromising correctness?

Hence →

OUR AIM: Improve performance by exploiting ILP !

## How do we exploit ILP?

- Have a collection of transformations, that operate on or across program blocks, either producing “faster code” or exposing more ILP.

An optimizing compiler does this by iteratively applying a series of transformations!

- Our transformations should rearrange code, from data available statically at compile time and from our knowledge of the underlying hardware.

## How do we exploit ILP?

→ **KEY IDEA:** These transformations do one (or both) of the following, while preserving correctness :

- 1.) Expose more ILP, such that later transformations in the compiler can exploit this exposure of more ILP.
- 2.) Perform a rearrangement of instructions, which results in increased performance (measured by execution time, or some other metric of interest)

## Loop Level Parallelism and Dependence

Q: What is Loop Level Parallelism?

A: ILP that exists as a result of iterating a loop.

Two types of dependencies limit the degree to which Loop Level Parallelism can be exploited.

Two types of dependencies

**Loop Carried**

A dependence, which only applies if a loop is iterated.

**Loop Independent**

A dependence within the body of the loop itself (i.e. within one iteration).

## An Example of Loop Level Dependences

→ Consider the following loop:

```
for (i = 0; i <= 100; i++) {
```

```
    A[i + 1] = A[i] + C[i];           // S1
```

```
    B[i + 1] = B[i] + A[i + 1];           // S2
```

```
}
```

**A Loop Independent Dependence**

N.B. how do we know  $A[i+1]$  and  $A[i+1]$  refer to the same location? In general by performing pointer/index variable analysis from conditions known at compile time.



In the following example code used for swapping the values of two array of length  $n$ , there is a loop-independent dependence ..

```
for (int i = 1; i < n; ++i) {  
    tmp = a[i];  
    a[i] = b[i];  
    b[i] = tmp;  
}
```

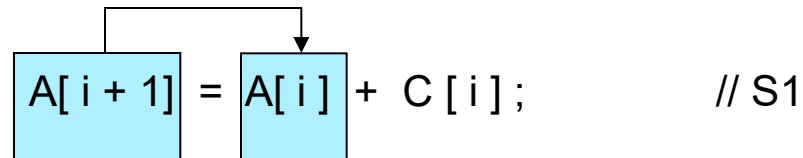
Example of loop carried dependence

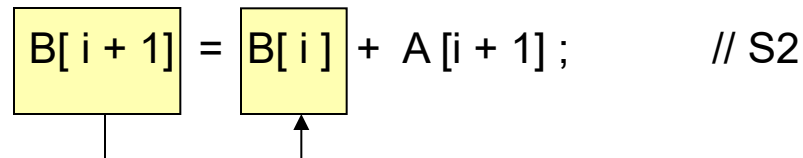
```
for (int i = 1; i < n; ++i) {  
    a[i] = a[i-1] + 1;  
}
```

## An Example of Loop Level Dependences

→ Consider the following loop:

```
for (i = 0; i <= 100; i++) {
```





```
}
```

Two Loop Carried Dependences

- OBSERVATION: A high proportion of loop instructions executed are loop management instructions on the induction variable.
- KEY IDEA: Eliminating this overhead could potentially significantly increase the performance of the loop:
- We'll use the following loop as our example:

```
for (i = 1000 ; i > 0 ; i -- ) {  
    x[ i ] = x[ i ] + constant;  
}
```

## a trivial translation to MIPS

```
for (i = 1000 ; i > 0 ; i-- ) {  
    x[ i ] = x[ i ] + constant;  
}
```

Our example translates into the MIPS assembly code below (**without any scheduling**).

Note the loop independent dependence in the loop ,i.e. x[ i ] on x[ i ]

---

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

Assume R1 holds the address of the element in array with highest address. and Assume F2 contains the constant.

## 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. For these examples we will assume:

- 1 branch delay slot,

→ Let us assume the following latencies for our pipeline:

INSTRUCTION PRODUCING RESULT	INSTRUCTION USING RESULT	LATENCY (in CC)*
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

→ Also assume that functional units are fully pipelined or replicated, such that one instruction can issue every clock cycle .

→ Assume no structural hazards exist,

\* - CC == Clock Cycles

## issuing our instructions

### Unscheduled LOOP

→ Let us issue the MIPS sequence of instructions obtained:

#### CLOCK CYCLE ISSUED

→ Loop :

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

INSTRUCTION PRODUCING RESULT	INSTRUCTION USING RESULT	LATENCY (in CC)*
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

## issuing our instructions

→ Let us issue the MIPS sequence of instructions obtained:

CLOCK CYCLE ISSUED			
→ Loop :	L.D	F0,0(R1)	1
		stall	2
	ADD.D	F4,F0,F2	3
		stall	4
		stall	5
	S.D	F4,0(R1)	6
	DADDUI	R1,R1,#-8	7
		stall	8
	BNE	R1,R2,Loop	9
		stall	10

→ Each iteration of the loop takes 10 cycles!

→ We can improve performance by rearranging the instructions, in the next slide.

We can push S.D. after BNE, if we alter the offset!

We can push ADDUI between L.D. and ADD.D, since R1 is not used anywhere within the loop body (i.e. it's the induction variable)



## issuing our instructions

→ Here is the **scheduled loop**:

		CLOCK CYCLE ISSUED
→ Loop :	L.D      F0,0(R1)	1
	DADDUI   R1,R1,#-8	2
	ADD.D    F4,F0,F2	3
	stall	4
	BNE      R1,R2,Loop	5
	S.D      F4,8(R1)	6

Here we've decremented R1 before we've stored F4. Hence need an offset of 8!

→ Each iteration now takes 6 cycles

→ This is the best we can achieve because of the inherent dependencies and pipeline latencies!

---

## issuing our instructions

→ Here is the scheduled loop:

CLOCK CYCLE ISSUED		
→ Loop :	L.D      F0,0(R1)	1
	DADDUI   R1,R1,#-8	2
	ADD.D    F4,F0,F2	3
	stall	4
	BNE      R1,R2,Loop	5
	S.D      F4,8(R1)	6

Observe that 3 out of the 6 cycles per loop iteration are due to loop overhead !

## 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: BNE,DADDUI, and stall

We need to get more operations into the loop relative to the number of branch and overhead instructions.

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

## 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.

## STATIC LOOP UNROLLING

- Hence, if we could decrease the loop management overhead, we could increase the performance.
- **SOLUTION : Static Loop Unrolling**
  - Make n copies of the loop body, adjusting the loop terminating conditions and perhaps renaming registers .
  - This results in less loop management overhead, since we effectively merge n iterations into one !
  - This exposes more ILP, since it allows instructions from different iterations to be scheduled together!

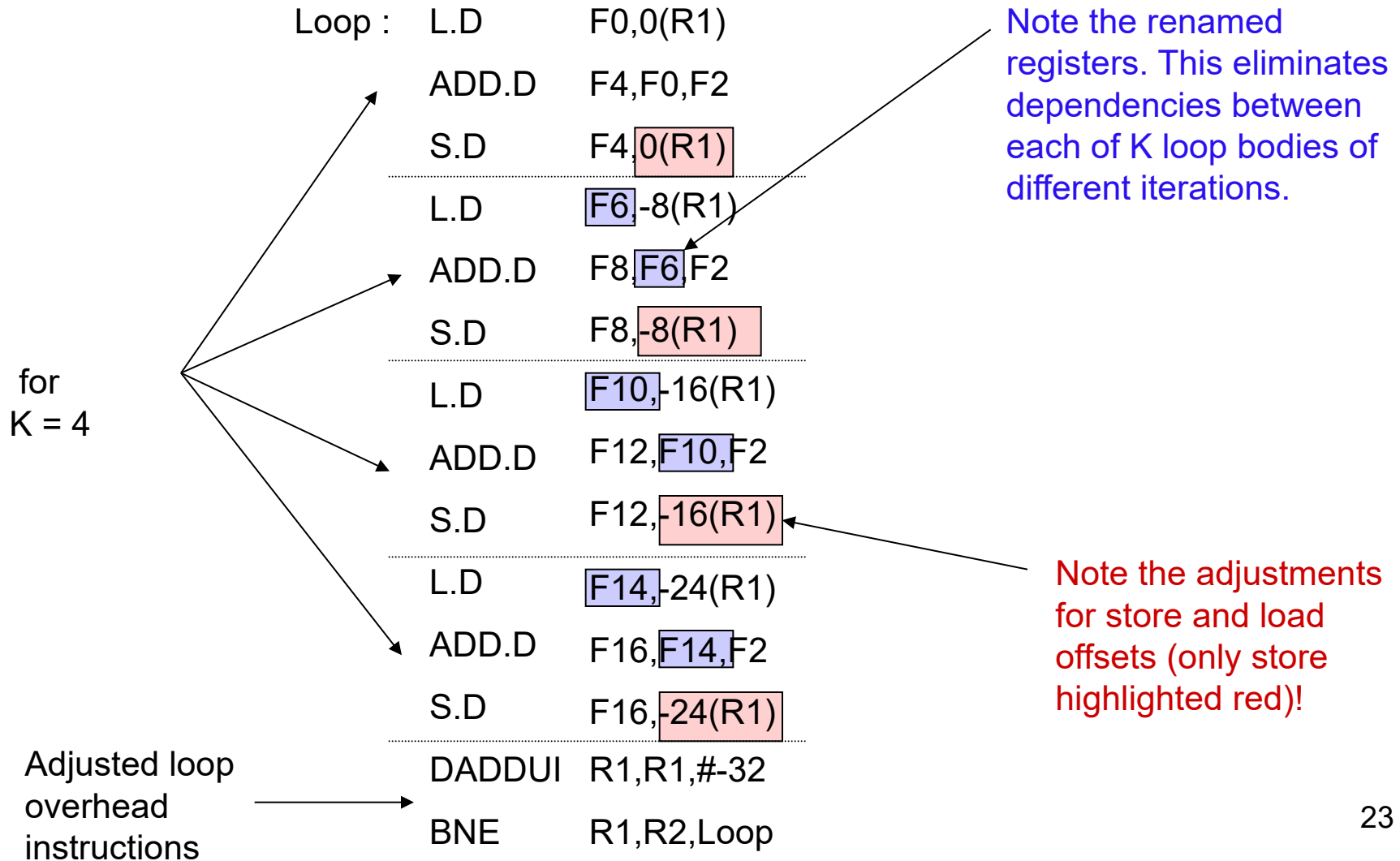
## STATIC LOOP UNROLLING (continued) – issuing our instructions

→ The unrolled loop from the running example with an unroll factor of  $K = 4$  would then be:

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

## STATIC LOOP UNROLLING (continued) – issuing our instructions

→ The unrolled loop from the running example with an unroll factor of K= 4 would then be:



## STATIC LOOP UNROLLING (continued) – issuing our instructions

**scheduled and unrolled loop 4 times**

→ Let's schedule the unrolled loop on our pipeline:

CLOCK CYCLE ISSUED

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	F4,0(R1)	9
	S.D	F8,-8(R1)	10
	DADDUI	R1,R1,#-32	11
	S.D	F12,16(R1)	12
	BNE	R1,R2,Loop	13
	S.D	F16,8(R1);	14



## STATIC LOOP UNROLLING (continued) – issuing our instructions

→ Let's schedule the unrolled loop on our pipeline:

This takes 14 cycles for 1 iteration of the unrolled loop.

Therefore w.r.t. original loop we now have  $14/4 = 3.5$  cycles per iteration.

Previously 6 was the best we could do!

→ We gain an increase in performance, at the expense of extra code and higher register usage/pressure

→ The performance gain on superscalar architectures would be even higher!

			CLOCK CYCLE ISSUED
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	F4,0(R1)	9
	S.D	F8,-8(R1)	10
	DADDUI	R1,R1,#-32	11
	S.D	F12,16(R1)	12
	BNE	R1,R2,Loop	13
	S.D	F16,8(R1);	14

## STATIC LOOP UNROLLING (continued)

**However loop unrolling has some significant complications and disadvantages:**

- Unrolling with an unroll factor of  $K$ , increases the code size by (approximately)  $K$ . This might present a problem,
- Imagine unrolling a loop with a factor  $K=4$ , that is executed a number of times that is not a multiple of four:
  - one would need to provide a copy of the original loop and the unrolled loop,
  - this would increase code size and management overhead significantly,
  - this is a problem, since we usually don't know the upper bound ( $n$ ) on the induction variable (which we took for granted in our example),
  - more formally, the original copy should be included if  $(n \bmod K \neq 0)$ , i.e. number of iterations is not a multiple of the unroll factor

## STATIC LOOP UNROLLING (continued)

**However loop unrolling has some significant complications and disadvantages:**

- We usually *ALSO* need to perform register renaming to reduce dependencies within the unrolled loop. This increases the register pressure!
- The criteria for performing loop unrolling are therefore usually very restrictive!