

# Impact of Instruction Latency

- Programming languages usually present a model in which one line (or operation) completes before the next starts
  - ♦ This is not what happens in pipelined architectures (= real world)
- Algorithms often have the same feature
  - ♦ After all, often written as pseudo code with these features
- Discussion Questions:
  - ♦ Does this impact the algorithms that are considered?
  - ♦ Would a graphical description encourage more exposure of overlapping operations
    - As in the pipelined sum
  - ♦ Alternatively, should the input format (written by the programmer) and the output format (developed by the compiler) be different to reflect the existence of pipelining
    - Some vectorizing compilers took small steps in this direction

# Architecture, Algorithms, and Programming Models: Instruction Pipelining

William Gropp



## Another Example: Reductions

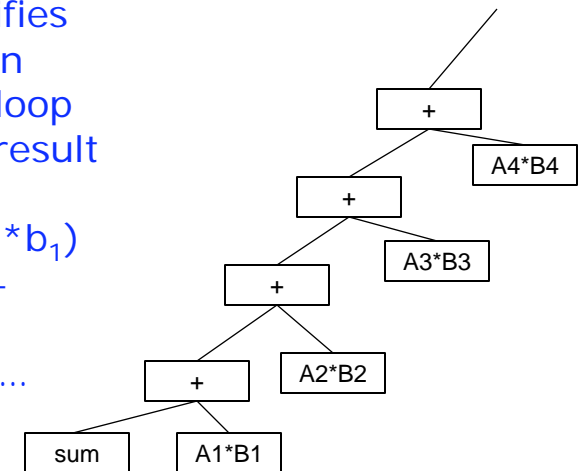
- Consider this code

```
do i=1,n
  sum = sum + a(i) * b(i)
enddo
```
- How fast can this run (assume data already in cache)?
  - ♦ Easy model: L1 Rate (needs 2 8-byte doubles/floating point add/multiply). If 32 GB/sec, then 4GFlops is possible with a 2GHz clock
  - ♦ But it is not that simple...



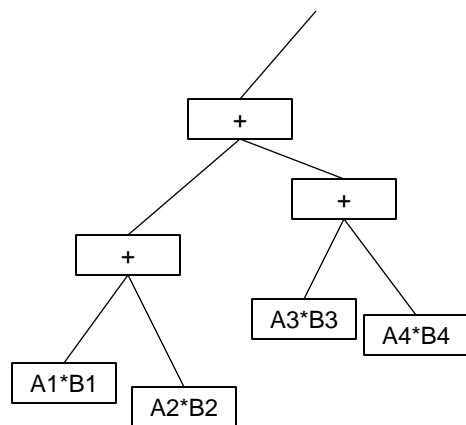
## Operation Order

- Fortran specifies the operation order. The loop defines the result as
$$\text{sum} = (((a_1 * b_1) + (a_2 * b_2)) + (a_3 * b_3)) + (a_4 * b_4) + \dots$$



## Impact of Implied Dependencies

- Assume each add takes 3 cycles to complete (latency)
- Since each add depends on the result of the prior add, only 1 add may be performed every 3 cycles
- Top speed reduced by a factor of 3
- But if the evaluation tree is balanced, there are separate adds (do not involve the results of an immediately prior add), those adds may *overlap*

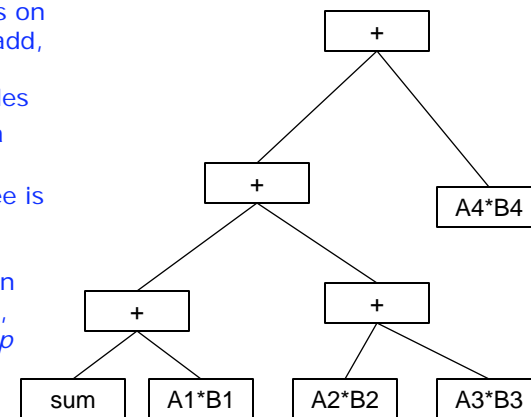


5



## Impact of Implied Dependencies

- Assume each add takes 3 cycles to complete (latency)
- Since each add depends on the result of the prior add, only 1 add may be performed every 3 cycles
- Top speed reduced by a factor of 3
- But if the evaluation tree is balanced, there are separate adds (do not involve the results of an immediately prior add), those adds may *overlap*



6



## Associativity

- Operations that are associative maybe evaluated in any grouping:
- $(a*b)*c = a*(b*c)$
- Floating point is *not* associative
  - ♦ Round-off error, e.g., can lead to large errors.
- Options available:
  - ♦ Rewrite code to provide an order of evaluation with fewer immediate dependencies
  - ♦ Tell compiler to assume all operations are associative (*all*, in the entire file)
    - Some higher-levels of optimization imply this
    - Without it, latency of pipeline operations severely limits performance
    - But with it, enabling optimization can *change the computed results!*

7



## A Faster Reduce

- Even better is to divide the tree. Consider this code instead
  - ♦ Performance of original version: 530 Mflops
  - ♦ Performance of new version: 1700 Mflops
- (Caveat: The new version has some other advantages)

```

sum1 = 0.0
sum2 = 0.0
sum3 = 0.0
sum4 = 0.0
do i=1,vecSize,4
  sum1 = sum1 + vecA(i)*vecB(i)
  sum2 = sum2 + vecA(i+1) * vecB(i+1)
  sum3 = sum3 + vecA(i+2) * vecB(i+2)
  sum4 = sum4 + vecA(i+3) * vecB(i+3)
enddo
sum = (sum1 + sum2) + (sum3 + sum4)
    
```

Discussion: What is wrong with this approach of manually expressing the ordering?

8



# Dot Product in More Detail

- Consider the following architecture:
  - L1 latency is 3 cycles
  - Floating multiplies take 5 cycles
  - Floating adds take 3 cycles
  - Integer operations take 1 cycle
  - Comparisons and branches take 2 cycles



## Notables

- 18 cycles for 8 operations = 44%
- Rate that loads can be issued controls performance: must load two values before beginning the related multiply



# Example Instruction Schedule

Cycle	Load/Store		Multiply		Add		Instr	
1	La1							
2		Lb1					A++	
3	>a1		La2				B++	
4	Lb2	>b1					A++	
5		La3	>a2	A1*b1			B++	
6	>b2		Lb3				A++	
7	La4	>a3		A2*b2			B++	
8		Lb4	>b3				A++	
9	>a4			=	A3*b3		B++	
10		>b4				+a1b1		
11				A4*b4	=			
12						=	+a2b2	
13					=			
14						+a3b3	=	N-=4
15				=				CMP
16						=	+a4b4	
17								Branch
18						=		



# Example Instruction Schedule: Idle Functional Units

Cycle	Load/Store		Multiply		Add		Instr	
1	La1							
2		Lb1					A++	
3	>a1		La2				B++	
4	Lb2	>b1					A++	
5		La3	>a2	A1*b1			B++	
6	>b2		Lb3				A++	
7	La4	>a3		A2*b2			B++	
8		Lb4	>b3				A++	
9	>a4			=	A3*b3		B++	
10		>b4				+a1b1		
11				A4*b4	=			
12						=	+a2b2	
13					=			
14						+a3b3	=	N-=4
15				=				CMP
16						=	+a4b4	
17								Branch
18						=		



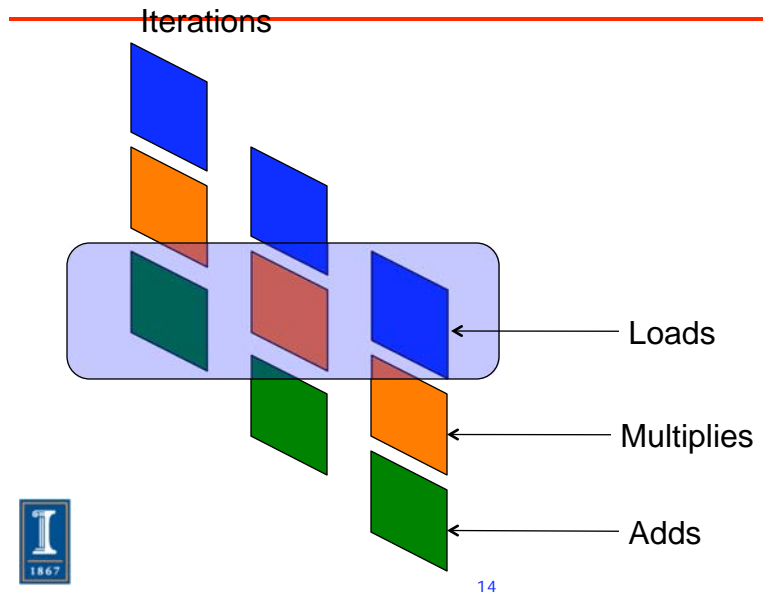
# Skewing

- The dependencies between the load, multiply, and add operations mean that there is much wasted time.
- We can *unroll* the loop to provide more operations
- We can *skew* the operations across loops to hide the latency of the operations



13

# Unroll and Skew



14

## Example Instruction Schedule: Unroll by 3 and Skew

Cycle	Load/Store		Multiply		Add	Instr
1	La3		A2*b2		S1+r1	
2		Lb3				
3	=		La1		=	
4	Lb1	=				
5			=	r2=	A3*b3	
6	=	La2				S2+r2
7			Lb2	A1*b1		
8		=			=	N-=4
9			=	r3=		cmp
10					S3+r3	
11				R1=		branch
12					=	



15

## Comments

- 12 cycles and 6 operations = 50%
- Additional unrolling and skewing can further increase the operations completed per cycle
- But adding additional steps requires additional registers to store the intermediate results
  - ◆ Registers are an important resource for hiding latency; a good architecture will provide sufficient registers for the task



16

## Double Loads

- Because only one data item can be loaded per cycle, we can bound the performance at 1 op/cycle
- If we could load 2 items/cycle, that bound become 2 op/cycle
- Many architectures provide such an operation (quad word load in POWER, e.g.)
- But these double loads come with restrictions



17

## Example Instruction Schedule

Cycle	Load/Store	Multiply	Add	Instr
1	La1,2			
2	Lb1,2			A+=2
3	>a1,2	La3,4		B+ =2
4	Lb3,4	>b1,2		A+=2
5		>a3,4	A1*b1	B+=2
6	>b3,4		A2*b2	
7			A3*b3	
8			A4*b4	
9		=		
10		=	+a1b1	
11		=	+a2b2	
12		=	=	+a3b3 N-=4
13			+a4b4	= CMP
14				
15			=	Branch



18

## Comments

- 15 cycles (instead of 18) for 8 ops
- This version does not use skewing, which would reduce total cycle count (hiding latency)



19

## Example Use of Double Loads

- Sparse matrix-vector multiplies look like dot products
  - ◆ Need to load 2 data items before performing 2 floating-point operations
  - ◆ To exploit double load instructions, organize data structure to satisfy alignment rules
    - For example, add zeros to make all values come in pairs
- A package that uses this approach is the Watson Sparse Matrix Package:
- <http://www-users.cs.umn.edu/~agupta/wsmp.html>



20

# Summary

---

- Latency in instructions adds additional performance problems unless the latency is hidden
  - ◆ Transformation to hide latency can change the order of evaluation as specified by the language, leading to different results at different levels of optimization, and *wrong* results if the order of evaluation was critical
    - *Compiler flags to change the language for an entire file are a dangerous and blunt-instrument approach*
    - *Question: Is there a better way?*

21



# Summary Con't

---

- Special instructions to better match data motion to compute operations (such as double load) can significantly improve performance
  - ◆ But these require
    - *Consecutive data items*
    - *Aligned data*
  - ◆ Achieving these can require ensuring aligned data and may require *changes to the data structures*, as in the Watson Sparse Matrix Package
- Language Questions
  - ◆ Can you think of a data type that is a natural, 16-byte type?
  - ◆ Should there be language support for similar types

22

