

# Supercomputing and Science



## **An Introduction to High Performance Computing**

---

### **Part III: Instruction-Level Parallelism and Scalar Optimization**

Henry Neeman, Director  
OU Supercomputing Center  
for Education & Research



# Outline

---

- What is Instruction-Level Parallelism?
- Scalar Operation
- Loops
- Pipelining
- Loop Performance
- Superpipelining
- Vectors
- A Real Example



# What Is ILP?

Instruction-Level Parallelism (ILP) is a set of techniques for executing **multiple instructions at the same time within the same CPU.**

The problem: the CPU has lots of circuitry, and at any given time, most of it is idle.

The solution: have different parts of the CPU work on different operations at the same time – if the CPU has the ability to work on 10 operations at a time, then the program can run as much as 10 times as fast.



# Kinds of ILP

- Superscalar: perform multiple operations at the same time
- Pipelining: perform different stages of the same operation on different sets of operands at the same time
- Superpipelining: combination of superscalar and pipelining
- Vector: special collection of registers for performing the same operation on multiple data at the same time



# What's an Instruction?

---

- Load a value from a specific address in main memory into a specific register
- Store a value from a specific register into a specific address in main memory
- Add (subtract, multiply, divide, square root, etc) two specific registers together and put the sum in a specific register
- Determine whether two registers both contain nonzero values ("**AND**")
- Branch to a new part of the program
- ... and so on



# What's a Cycle?

You've heard people talk about having a 500 MHz processor or a 1 GHz processor or whatever. (For example, Henry's laptop has a 700 MHz Pentium III.)

Inside every CPU is a little clock that ticks with a fixed frequency. We call each tick of the CPU clock a **clock cycle** or a **cycle**.

Typically, primitive operations (e.g., add, multiply, divide) each take a fixed number of cycles to execute (before pipelining).



# Scalar Operation

---



---

**DON'T  
PANIC!**





# Scalar Operation

$$z = a * b + c * d$$

How would this statement be executed?

1. Load **a** into register **R0**
2. Load **b** into **R1**
3. Multiply **R2 = R0 \* R1**
4. Load **c** into **R3**
5. Load **d** into **R4**
6. Multiply **R5 = R3 \* R4**
7. Add **R6 = R2 + R5**
8. Store **R6** into **z**



# Does Order Matter?

$$z = a * b + c * d$$

1. Load **a** into R0
2. Load **b** into R1
3. Multiply **R2 = R0 \* R1**
4. Load **c** into R3
5. Load **d** into R4
6. Multiply **R5 = R3 \* R4**
7. Add **R6 = R2 + R5**
8. Store **R6** into **z**

1. Load **d** into R4
2. Load **c** into R3
3. Multiply **R5 = R3 \* R4**
4. Load **a** into R0
5. Load **b** into R1
6. Multiply **R2 = R0 \* R1**
7. Add **R6 = R2 + R5**
8. Store **R6** into **z**

In the cases where order doesn't matter, we say that the operations are independent of one another.



# Superscalar Operation

$$z = a * b + c * d$$

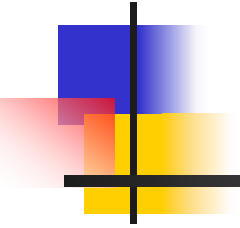
By performing multiple operations at a time,  
we can reduce the execution time.

1. Load **a** into **R0** **AND** load **b** into **R1**
2. Multiply **R2 = R0 \* R1** **AND**  
load **c** into **R3** **AND** load **d** into **R4**
3. Multiply **R5 = R3 \* R4**
4. Add **R6 = R2 + R5**
5. Store **R6** into **z**

So, we go from 8 operations down to 5. Big deal.

# Loops

---





# Loops Are Good

---

Most compilers are very good at optimizing **loops**, and not very good at optimizing other constructs.

```
DO index = 1, length
    dst(index) = src1(index) + src2(index)
END DO !! index = 1, length
```

Why?



# Why Loops Are Good

---

- Loops are very common in many programs.
- So, hardware vendors have designed their products to be able to do loops well.
- Also, it's easier to optimize loops than more arbitrary sequences of instructions: when a program does the same thing over and over, it's easier to predict what's likely to happen next.



---

**DON'T  
PANIC!**



# Superscalar Loops

```
DO i = 1, n  
    z(i) = a(i)*b(i) + c(i)*d(i)  
END DO !! i = 1, n
```

Each of the iterations is completely independent of all of the other iterations; e.g.,

$$z(1) = a(1)*b(1) + c(1)*d(1)$$

has nothing to do with

$$z(2) = a(2)*b(2) + c(2)*d(2)$$

Operations that are independent of each other can be performed in parallel.





# Superscalar Loops

```
for (i = 0; i < n; i++) {  
    z[i] = a[i]*b[i] + c[i]*d[i];  
} /* for i */
```

1. Load a[0] into R0 **AND** load b[0] into R1
2. Multiply R2 = R0 \* R1 **AND** load c[0] into R3  
**AND** load d[0] into R4
3. Multiply R5 = R3 \* R4 **AND** load a[1] into R0  
**AND** load b[1] into R1
4. Add R6 = R2 + R5 **AND** load c[1] into R3 **AND**  
load d[1] into R4
5. Store R6 into z[0] **AND** multiply R2 = R0 \* R1 . .

Once this sequence is “in flight,” each iteration adds only 2 operations to the total, not 8.



# Example: Sun UltraSPARC-III

---

4-way Superscalar: can execute up to 4 operations at the same time<sup>[1]</sup>

- 2 integer, memory and/or branch
  - Up to 2 arithmetic or logical operations, and/or
  - 1 memory access (load or store), and/or
  - 1 branch
- 2 floating point (e.g., add, multiply)



# Pipelining

---



# Pipelining

---

Pipelining is like an assembly line or a bucket brigade.

- An operation consists of multiple stages.
- After a set of operands

$$z(i) = a(i) * b(i) + c(i) * d(i)$$

complete a particular stage, they move into the next stage.

- Then, the next set of operands

$$z(i+1) = a(i+1) * b(i+1) + c(i+1) * d(i+1)$$

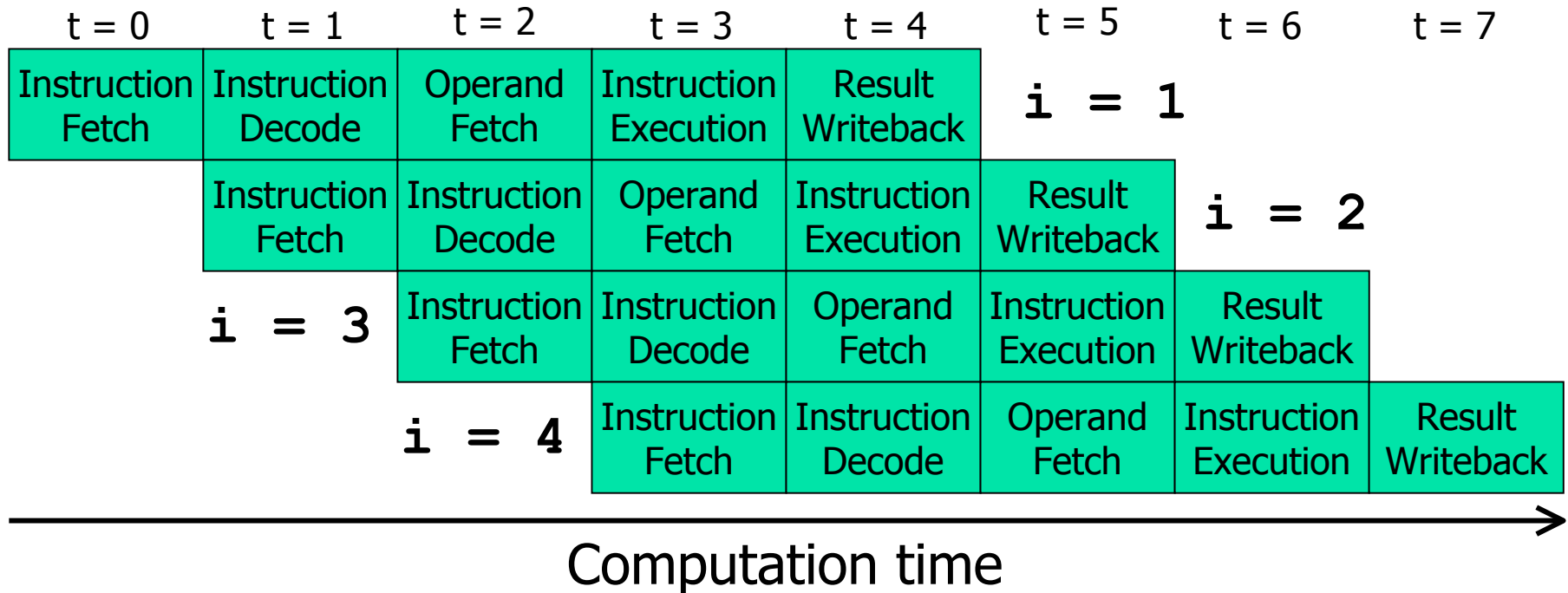
can move into the stage that iteration  $i$  just completed.



---

**DON'T  
PANIC!**

# Pipelining Example



If each stage takes, say, one CPU cycle, then once the loop gets going, each iteration of the loop only increases the total time by one cycle. So a loop of length 1000 takes only 1004 cycles. <sup>[2]</sup>



# Some Simple Loops

```
DO index = 1, length
  dst(index) = src1(index) + src2(index)
END DO !! index = 1, length
```

```
DO index = 1, length
  dst(index) = src1(index) - src2(index)
END DO !! index = 1, length
```

```
DO index = 1, length
  dst(index) = src1(index) * src2(index)
END DO !! index = 1, length
```

```
DO index = 1, length
  dst(index) = src1(index) / src2(index)
END DO !! index = 1, length
```

```
DO index = 1, length
  sum = sum + src(index)
END DO !! index = 1, length
```

Reduction: convert  
array to scalar



# Slightly Less Simple Loops

```
DO index = 1, length
  dst(index) = src1(index) ** src2(index)
END DO !! index = 1, length
```

```
DO index = 1, length
  dst(index) = MOD(src1(index), src2(index))
END DO !! index = 1, length
```

```
DO index = 1, length
  dst(index) = SQRT(src(index))
END DO !! index = 1, length
```

```
DO index = 1, length
  dst(index) = COS(src(index))
END DO !! index = 1, length
```

```
DO index = 1, length
  dst(index) = EXP(src(index))
END DO !! index = 1, length
```

```
DO index = 1, length
  dst(index) = LOG(src(index))
END DO !! index = 1, length
```





# Loop Performance

---

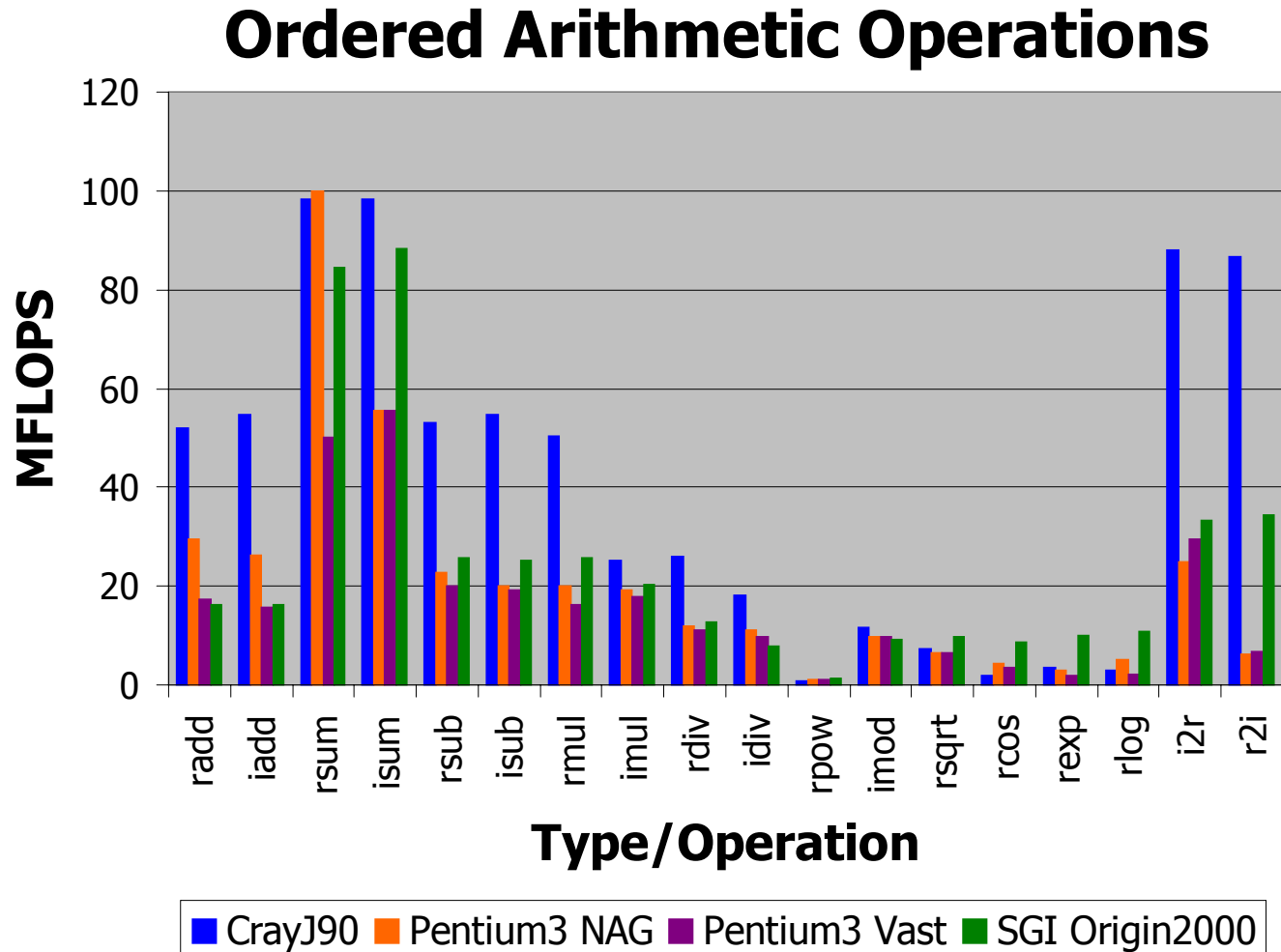


# Performance Characteristics

---

- Different operations take different amounts of time.
- Different processors types have different performance characteristics, but there are some characteristics that many platforms have in common.
- Different compilers, even on the same hardware, perform differently.
- On some processors, floating point and integer speeds are similar, while on others they differ.

# Arithmetic Operation Speeds





# What Can Prevent Pipelining?

---

Certain events make it very hard (maybe even impossible) for compilers to pipeline a loop, such as:

- array elements accessed in random order
- loop body too complicated
- **IF** statements inside the loop (on some platforms)
- premature loop exits
- function/subroutine calls
- I/O



# How Do They Kill Pipelining?

---

- **Random access order:** ordered array access is common, so pipelining hardware and compilers tend to be designed under the assumption that most loops will be ordered. Also, the pipeline will constantly stall because data will come from main memory, not cache.
- **Complicated loop body:** compiler gets too overwhelmed and can't figure out how to schedule the instructions.



# How Do They Kill Pipelining?

- **IF** statements in the loop: on some platforms (but not all), the pipelines need to perform exactly the same operations over and over; **IF** statements make that impossible. **However**, many CPUs can now perform speculative execution: both branches of the **IF** statement are executed while the condition is being evaluated, but only one of the results is retained (the one associated with the condition's value).



# How Do They Kill Pipelining?

- **Function/subroutine calls** interrupt the flow of the program even more than **IF** statements. They can take execution to a completely different part of the program, and pipelines aren't set up to handle that.
- **Loop exits** are similar.
- **I/O**: typically, I/O is handled in subroutines (above). Also, I/O instructions can take control of the program away from the CPU (they can give control to I/O devices).



# What If No Pipelining?

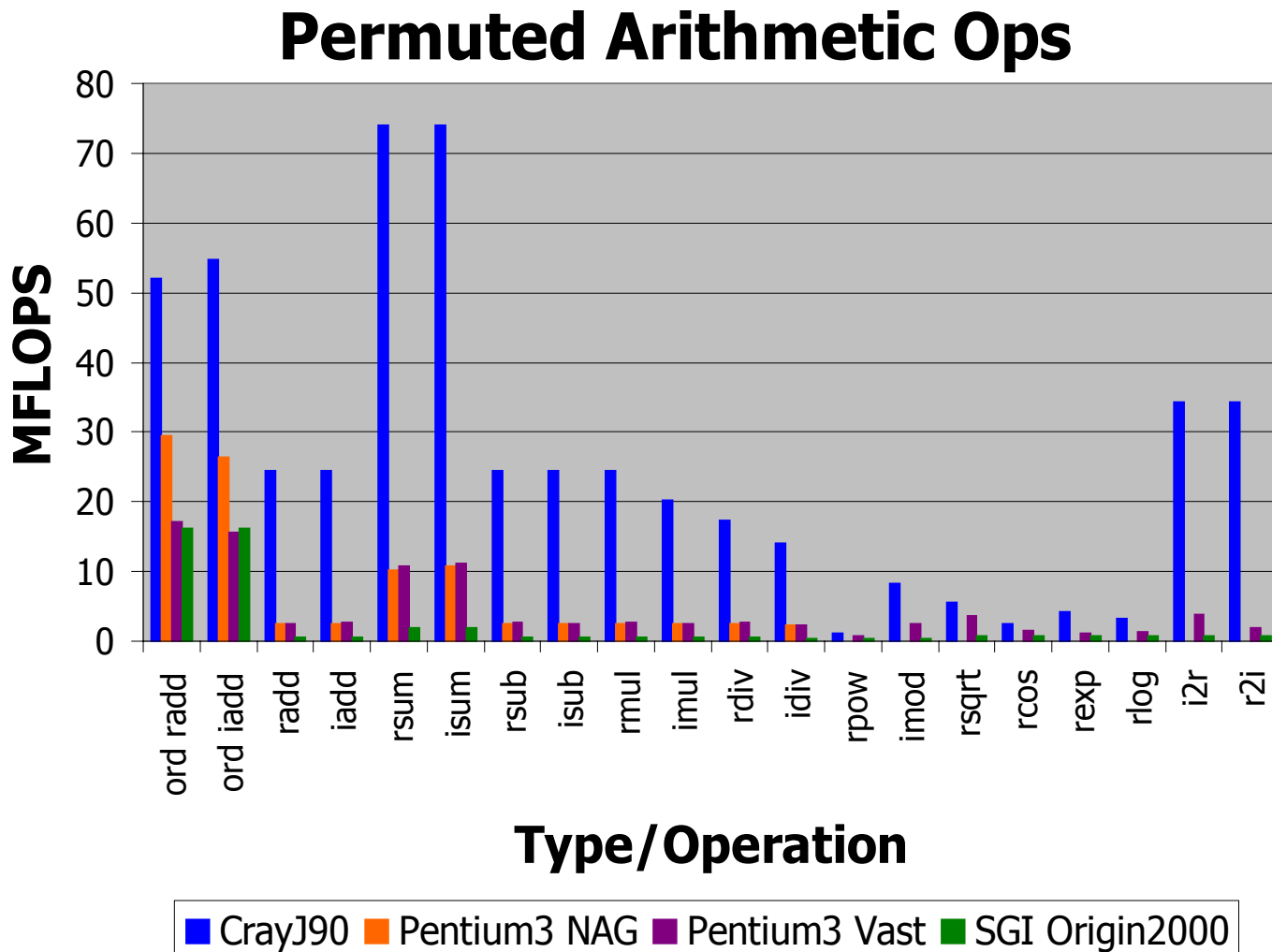
---

**SLOW!**

(on most platforms)



# Randomly Permuted Loops





# Superpipelining

---



# Superpipelining

---

Superpipelining is a combination of superscalar and pipelining.

So, a superpipeline is a collection of multiple pipelines that can operate simultaneously.

In other words, several different operations can execute simultaneously, and each of these operations can be broken into stages, each of which is filled all the time.

So you can get multiple operations per cycle.

For example, a Compaq Alpha 21264 can have up to 80 operations “in flight” at once.<sup>[3]</sup>



# More Ops At a Time

---

- If you put more operations into the code for a loop, you'll get better performance:
  - more operations can execute at a time (use more pipelines), and
  - you get better register/cache reuse.
- On most platforms, there's a limit to how many operations you can put in a loop to increase performance, but that limit varies among platforms, and can be quite large.



# Some Complicated Loops

```
DO index = 1, length
  dst(index) = src1(index) + 5.0 * src2(index)
END DO !! index = 1, length
```

madd:  
multiply then add  
(2 ops)

```
dot = 0
DO index = 1, length
  dot = dot + src1(index) * src2(index)
END DO !! index = 1, length
```

dot product  
(2 ops)

```
DO index = 1, length
  dst(index) = src1(index) * src2(index) + &
  & src3(index) * src4(index)
END DO !! index = 1, length
```

from our  
example  
(3 ops)

```
DO index = 1, length
  diff12 = src1(index) - src2(index)
  diff34 = src3(index) - src4(index)
  dst(index) = SQRT(diff12 * diff12 + diff34 * diff34)
END DO !! index = 1, length
```

Euclidean distance  
(6 ops)



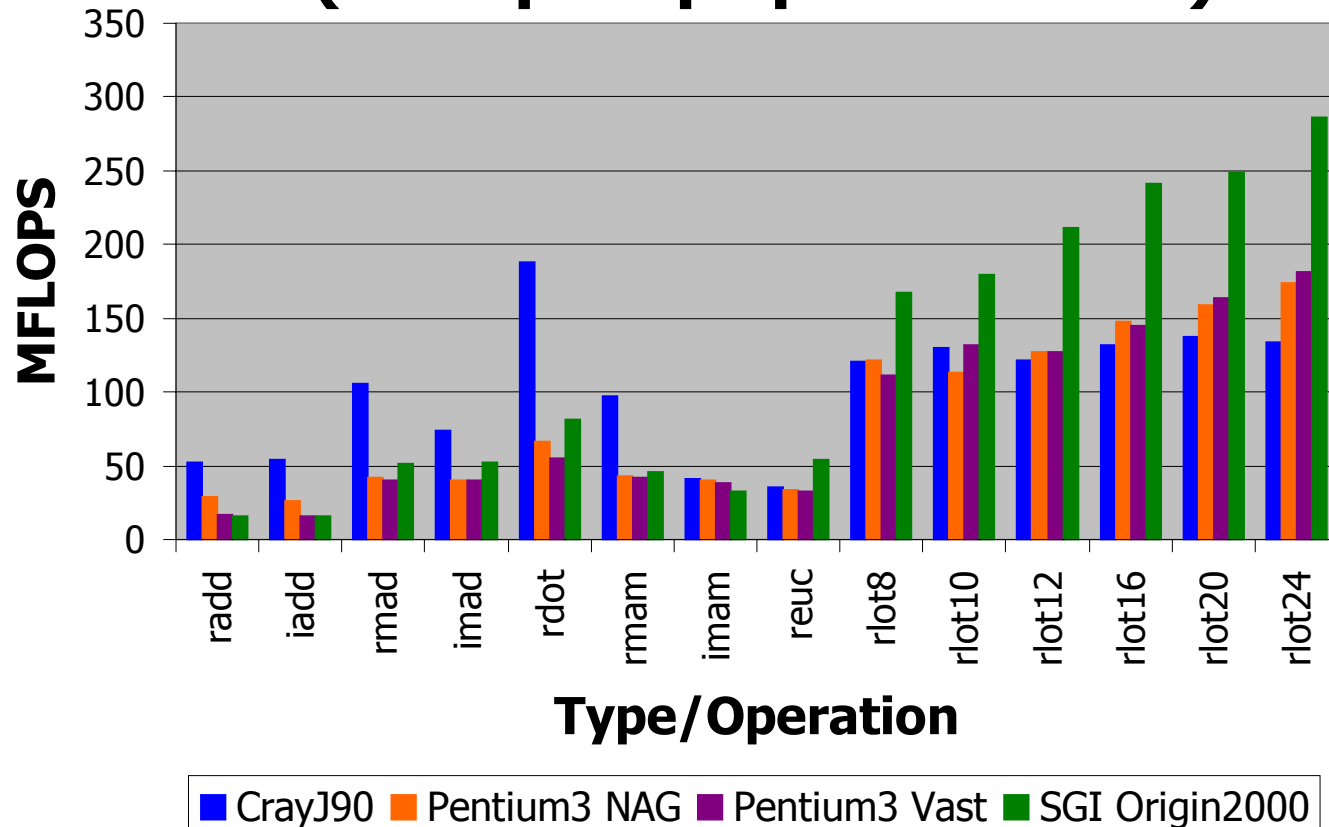
# A Very Complicated Loop

```
lot = 0.0
DO index = 1, length
    lot = lot +
        &      src1(index) * src2(index) +      &
        &      src3(index) * src4(index) +      &
        &      (src1(index) + src2(index)) *      &
        &      (src3(index) + src4(index)) *      &
        &      (src1(index) - src2(index)) *      &
        &      (src3(index) - src4(index)) *      &
        &      (src1(index) - src3(index) +      &
        &      src2(index) - src4(index)) *      &
        &      (src1(index) + src3(index) -      &
        &      src2(index) + src4(index)) +      &
        &      (src1(index) * src3(index)) +      &
        &      (src2(index) * src4(index))
END DO !! index = 1, length
```

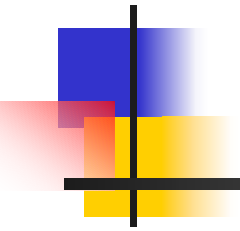
24 arithmetic ops per iteration  
4 memory/cache loads per iteration

# Multiple Ops Per Iteration

## Ordered Arithmetic Operations (multiple ops per iteration)



# Vectors







# What Is a Vector?

---

A vector is a collection of registers that act together to perform the same operation on multiple operands.

In a sense, vectors are like operation-specific cache.

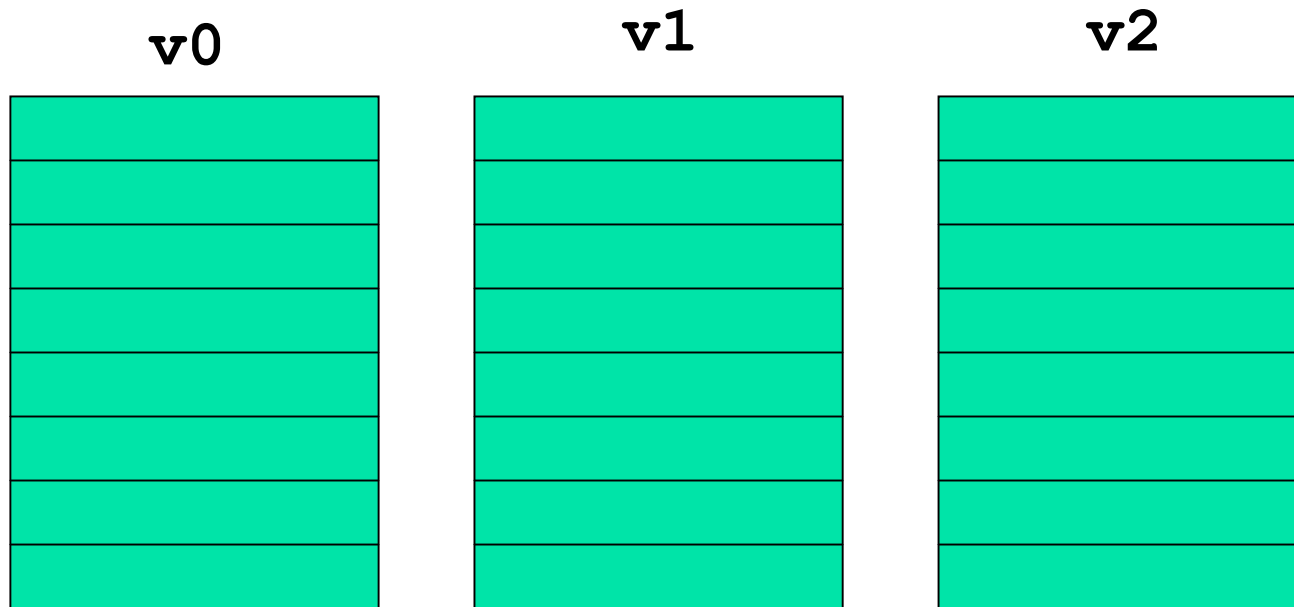
A vector register is a register that's actually made up of many individual registers.

A vector instruction is an instruction that operates on all of the individual registers of a vector register.



# Vector Register

---



$$\mathbf{v2} = \mathbf{v0} + \mathbf{v1}$$



# Vectors Are Expensive

---

Vectors were very popular in the 1980s, because they're very fast, often faster than pipelines.

Today, though, they're very unpopular. Why?

Well, vectors aren't used by most commercial codes (e.g., MS Word). So most chip makers don't bother with vectors.

So, if you want vectors, you have to pay a lot of extra money for them.



# The Return of Vectors

---

Vectors are making a comeback in a very specific context: graphics hardware.

It turns out that speed is incredibly important in computer graphics, because you want to render millions of objects (typically tiny triangles) per second.

So some graphics hardware has vector registers and vector operations.



# A Real Example

---



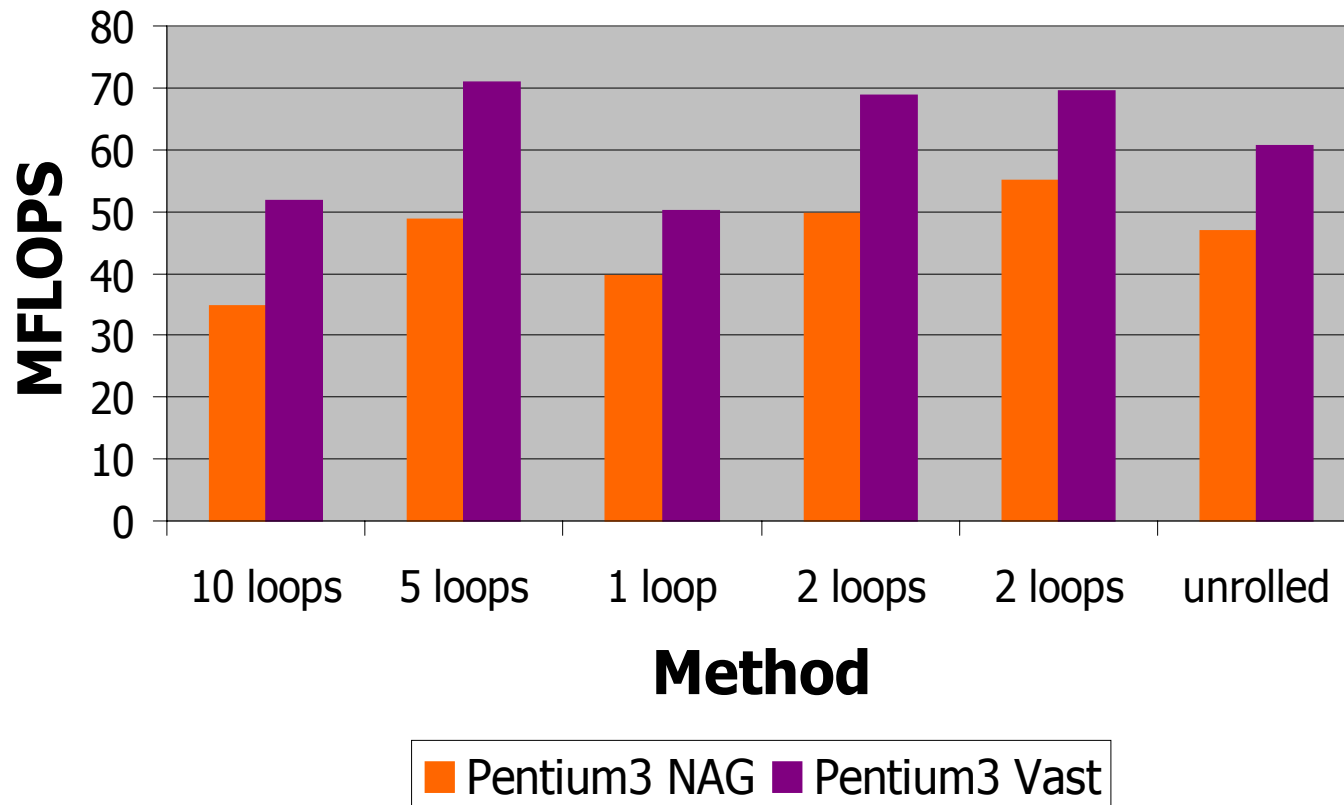
# A Real Example<sup>[4]</sup>

```
DO k=2,nz-1
  DO j=2,ny-1
    DO i=2,nx-1
      tem1(i,j,k) = u(i,j,k,2)*(u(i+1,j,k,2)-u(i-1,j,k,2))*dxinv2
      tem2(i,j,k) = v(i,j,k,2)*(u(i,j+1,k,2)-u(i,j-1,k,2))*dyinv2
      tem3(i,j,k) = w(i,j,k,2)*(u(i,j,k+1,2)-u(i,j,k-1,2))*dzinv2
    END DO
  END DO
END DO
DO k=2,nz-1
  DO j=2,ny-1
    DO i=2,nx-1
      u(i,j,k,3) = u(i,j,k,1) -      &
&      dtbig2*(tem1(i,j,k)+tem2(i,j,k)+tem3(i,j,k))
    END DO
  END DO
END DO

. . .
```

# Real Example Performance

## Performance By Method





---

**DON'T  
PANIC!**





# Why You Shouldn't Panic

---

In general, the compiler and the CPU will do most of the heavy lifting for instruction-level parallelism.

**BUT:**

You need to be aware of ILP, because how your code is structured affects how much ILP the compiler and the CPU can give you.



# Next Time

---

## **Part IV: Dependency Analysis and Stupid Compiler Tricks**



# References

---

- [1] Ruud van der Pas, "The UltraSPARC-III Microprocessor: Architecture Overview," 2001, p. 23.
- [2] Kevin Dowd and Charles Severance, *High Performance Computing*, 2<sup>nd</sup> ed. O'Reilly, 1998, p. 16.
- [3] "Alpha 21264 Processor" (internal Compaq report), page 2.
- [4] Code courtesy of Dan Weber, 2001.