On Implicit Parallelism (with respective compiler roles and coding techniques)

(Muhammad) Amjad Ali
Bahauddin Zakariya University, Multan,
Pakistan.

Serial Computing---to----Parallel

- Serial: Executing instructions one after the other
- Serial: Intrinsic of the von-Neumann Architecture
- Serial: ISA assumes execution of instructions sequential
- But its SLOW------its limiting !
- Limits in building ever faster serial computer
  - Transmission speed
  - Miniaturization
  - Cost
- Ultimately, we need to think some other way: Parallelism
- Needed both in Hardware and Software

What Is Parallelism?

Parallelism is the simultaneous/concurrent use of multiple processing units (either processors or parts of an individual processor) to solve a problem-------
by concurrently solving different parts of the problem

The different parts could be different tasks, or the same task on different pieces of the problem's data.

Luckily, THE UNIVERSE IS PARALLEL

Programmers/developers can recognize/realize/define some one or other form of potential of parallelism in real world problems

So let's break the rule:

- Parallel Execution (not sequential)

- WHERE:
  - At instruction level
  - At thread level
  - At process level

- Is that legal !!
- YES------unless we get caught.
- What's not "get caught" ?
- Program executes correctly

Common Kinds of Parallelism

- Instruction Level Parallelism
  - (Pipelining, Superscalar, Vectorization, etc.)
- Shared Memory Multithreading
  - (OpenMP, Cilk Plus, p-threads)
- Distributed Multiprocessing (for example, MPI, PVM)
- GPU Parallelism (for example, CUDA, OpenCL)
- Hybrid Parallelism
  - Distributed + Shared (for example, MPI + OpenMP)
  - Shared + GPU (for example, OpenMP + CUDA)
  - Distributed + GPU (for example, MPI + CUDA)

Implicit Parallelism

(from programmers perspective)

- Instruction Level Parallelism
  - Pipelining
  - Superscalar
  - Super-pipelining
  - Vectorization
- Automatic Parallelization
  - Shared Memory Multithreading (using ONLY compiler directives----for example with OpenMP, Cilk Plus etc.)

Whatever else is in parallelization is EXPLICIT-------

obviously the programmer has to do MUCH MORE in that.

What Is ILP?

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

(Note that ILP has nothing to do with multicore.)

Basic idea:

Execute several instructions in parallel; i.e., overlap the execution of instructions to run programs faster ("to improve performance").

The problem: A CPU core has lots of circuitry, and at any given time, most of it is idle, which is wasteful.

The solution: Have different parts of the CPU core work on different operations at the same time: If the CPU core has the ability to work on 10 operations at a time, then the program can, in principle, run as much as 10 times as fast (although in practice, not quite so much).

Why You Shouldn't Panic

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

BUT:

Application Programmer needs to be aware of ILP, because how the code is structured affects how much ILP the compiler and the CPU can deliver.

Kinds of ILP

- *Superscalar*: Perform multiple instructions at the same time (for example, simultaneously perform an add, a multiply and a load).
- *Pipeline*: Start performing an operation on one piece of data while finishing the same operation on another piece of data – perform different *stages* of the same INSTRUCTION on different sets of operands at the same time (like an assembly line).
- *Superpipeline*: A combination of superscalar and pipelining – perform multiple pipelined instructions at the same time.
- *Vector*: Load multiple pieces of data into special registers and perform the same instruction on all of them at the same time.

Illusion of Sequentiality

Although the modern CPUs provide immense amount of ILP in different categories, but the user gets the feeling of sequentiality.

So long as everything looks OK to the outside world you can do whatever you want!

- "Outside Appearance" = "Architecture" (ISA)
- "Whatever you want" = "Microarchitecture"

- μArch basically includes everything not explicitly defined in the ISA
  - pipelining, superscalar, caches, branch prediction, etc.

What's an Instruction?

- *Memory*: For example, load a value from a specific address in main memory into a specific register, or store a value from a specific register into a specific address in main memory.
- *Arithmetic*: For example, add two specific registers together and put their sum in a specific register – or subtract, multiply, divide, square root, etc.
- *Logical*: For example, determine whether two registers both contain nonzero values ("AND").
- *Branch*: Jump from one sequence of instructions to another (for example, function call).
- … and so on ….

What is a CPU Cycle or Clock Cycle?

Inside every CPU is a little clock that ticks with a fixed frequency, measured in cycles/sec or Hertz.
(For example, consider a laptop with a 2.2 GHz Core 2 Duo.)

We call each tick of the CPU clock a *clock cycle* or a *cycle*.

So a 2 GHz processor has 2 billion clock cycles per second.

Typically, a primitive operation (for example, add, multiply, divide) takes a fixed number of cycles to execute (assuming no pipelining).

What's the Relevance of Cycles?

Typically, a primitive INSTRUCTION (for example, add, multiply, divide) takes a fixed number of cycles to execute (assuming no pipelining).

- IBM POWER4
  - Multiply or add: 6 cycles (64 bit floating point)
  - Load: 4 cycles from L1 cache
    14 cycles from L2 cache
- Intel Pentium4 EM64T (Core)
  - Multiply: 7 cycles (64 bit floating point)
  - Add, subtract: 5 cycles (64 bit floating point)
  - Divide: 38 cycles (64 bit floating point)
  - Square root: 39 cycles (64 bit floating point)
  - Tangent: 240-300 cycles (64 bit floating point)

Fast and Slow Operations

- **Fast**: sum, add, subtract, multiply
- **Medium**: divide, mod (that is, remainder)
- **Slow**: transcendental functions (sqrt, sin, exp)
- **Incredibly slow**: power x^y for real x and y

On most platforms, divide, mod and transcendental functions are not ***pipelined***, so a code will run faster if most of it is just adds, subtracts and multiplies.

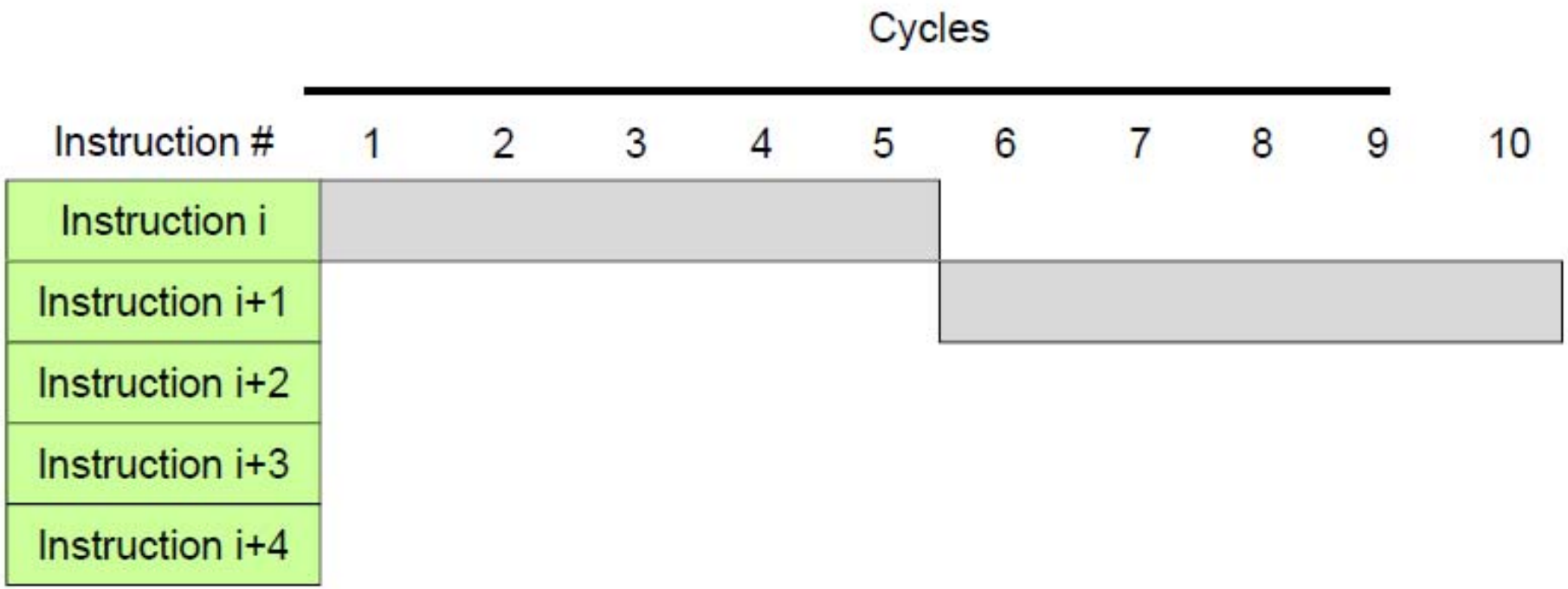
A Conclusion:

- Prefer to replace slow operations with faster ones. For example `(x/y)/z` should be replaced with `x/(y*z)`.
- Sometime it could also be useful to consider a finite series approximation of transcendental function to reduce computation.

5 Stages or Steps in Executing a MIPS instruction

1) **IF**: Instruction Fetch, Increment PC
2) **ID**: Instruction Decode, Read Registers
3) **EX**: (execution)
   - Mem-ref: Calculate Address
   - Arithmatic/logical: Perform Operation
4) **MEM**:
   - Load: Read Data from Memory
   - Store: Write Data to Memory
5) **WB**: Write Data Back to Register

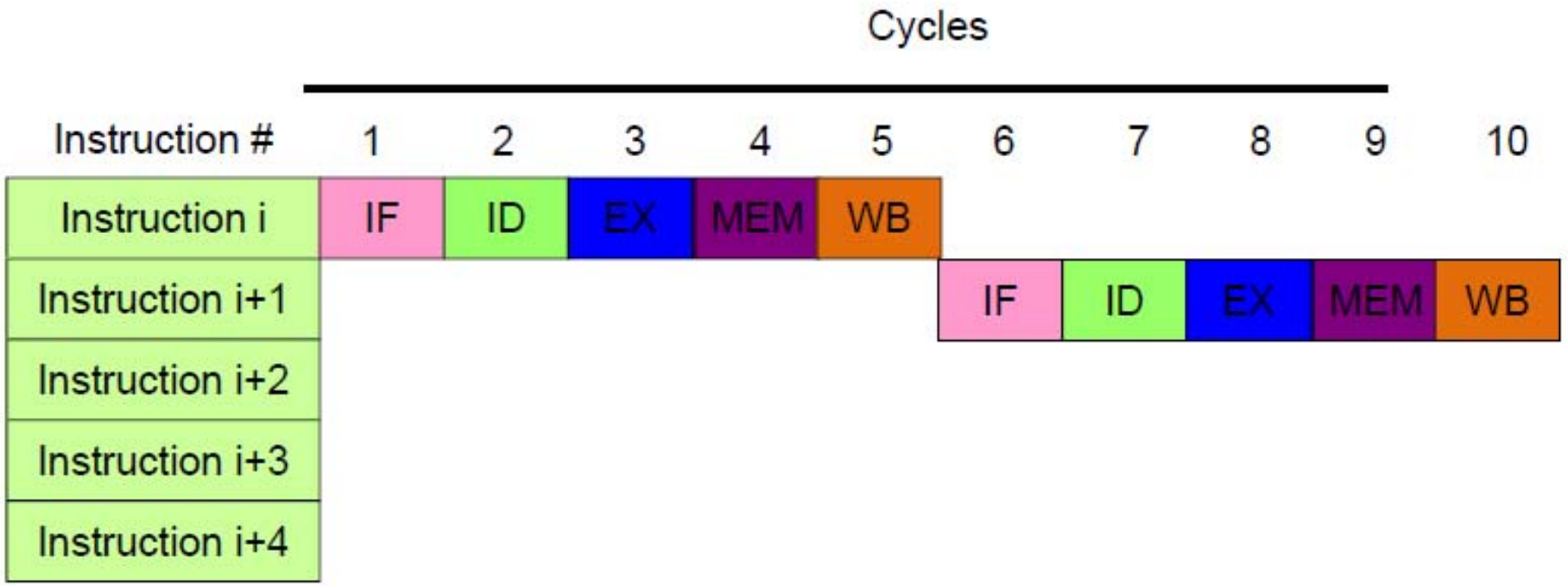
Instruction Execution



Instruction Execution

IF: Instruction fetch
EX : Execution
WB : Write back

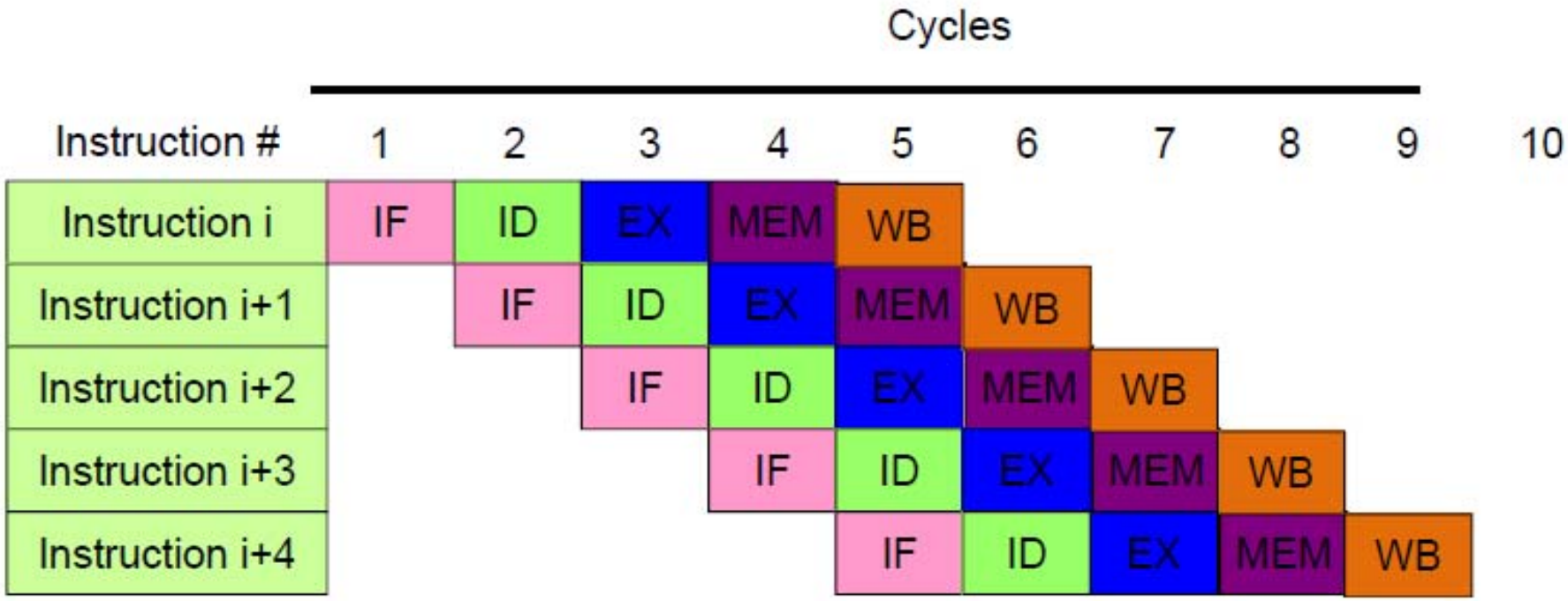
ID : Instruction decode
MEM: Memory access



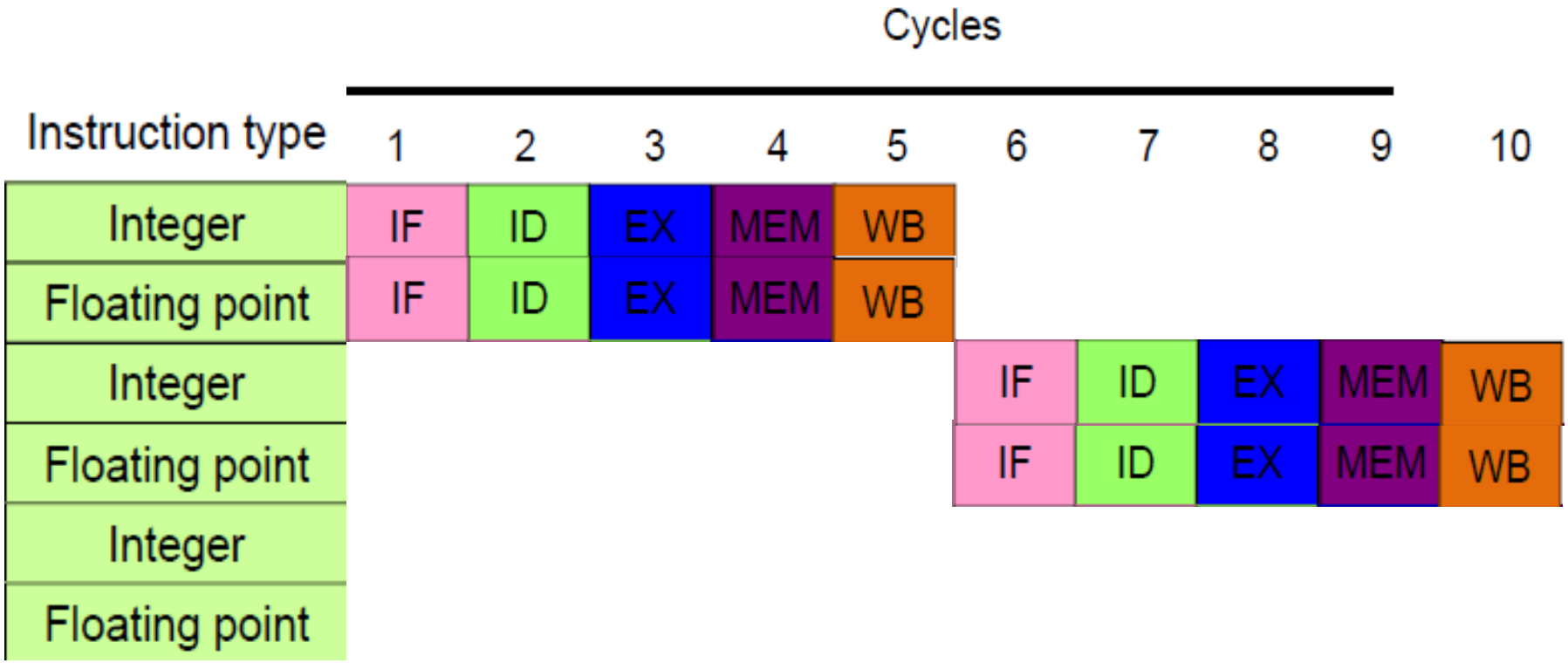
Pipelining Execution

IF: Instruction fetch
EX : Execution
WB : Write back

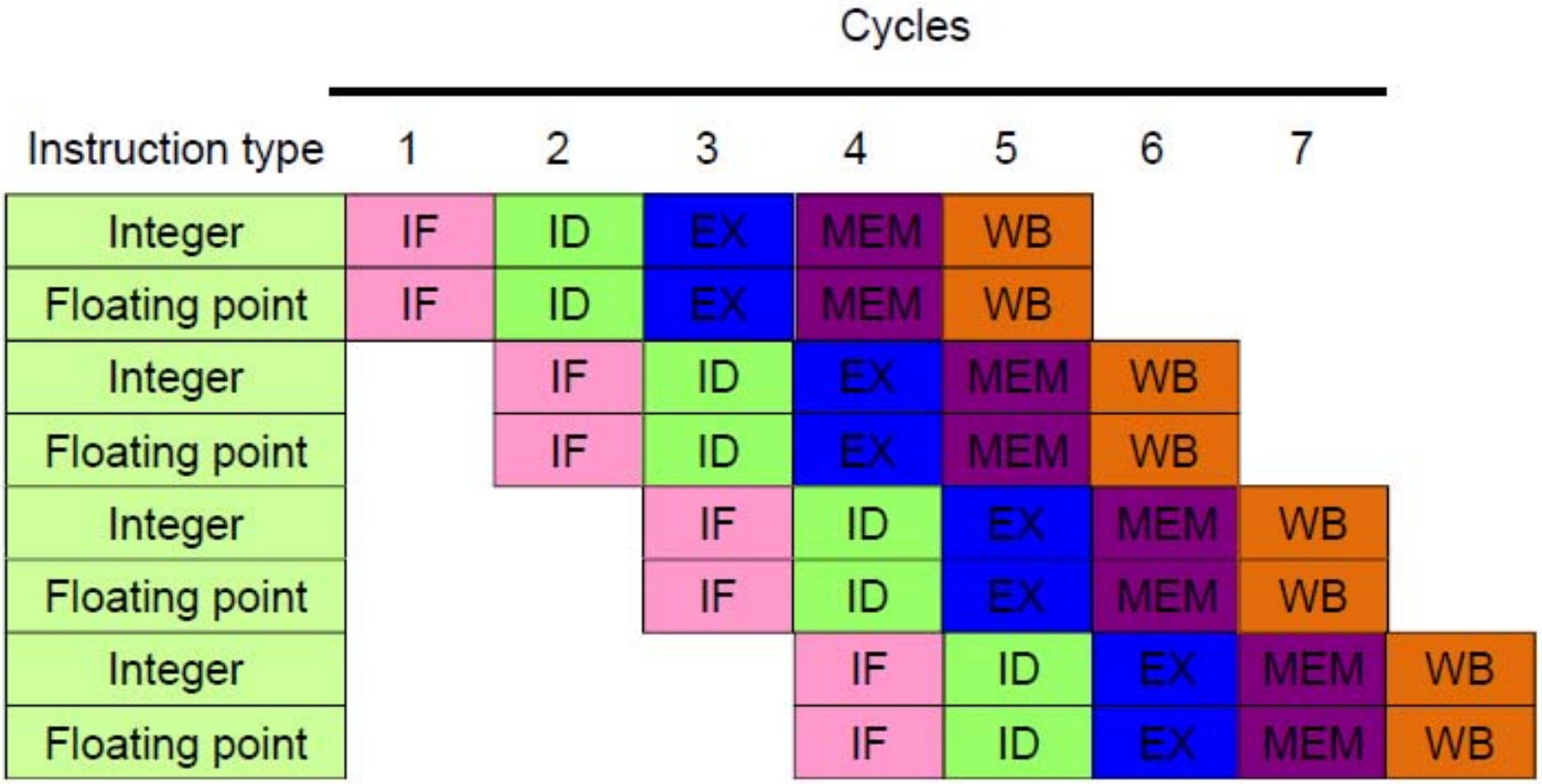
ID : Instruction decode
MEM: Memory access



Superscalar Execution



Super-pipelining : Superscalar + Pipeline



2-issue super-scalar machine

What Is a Vector?

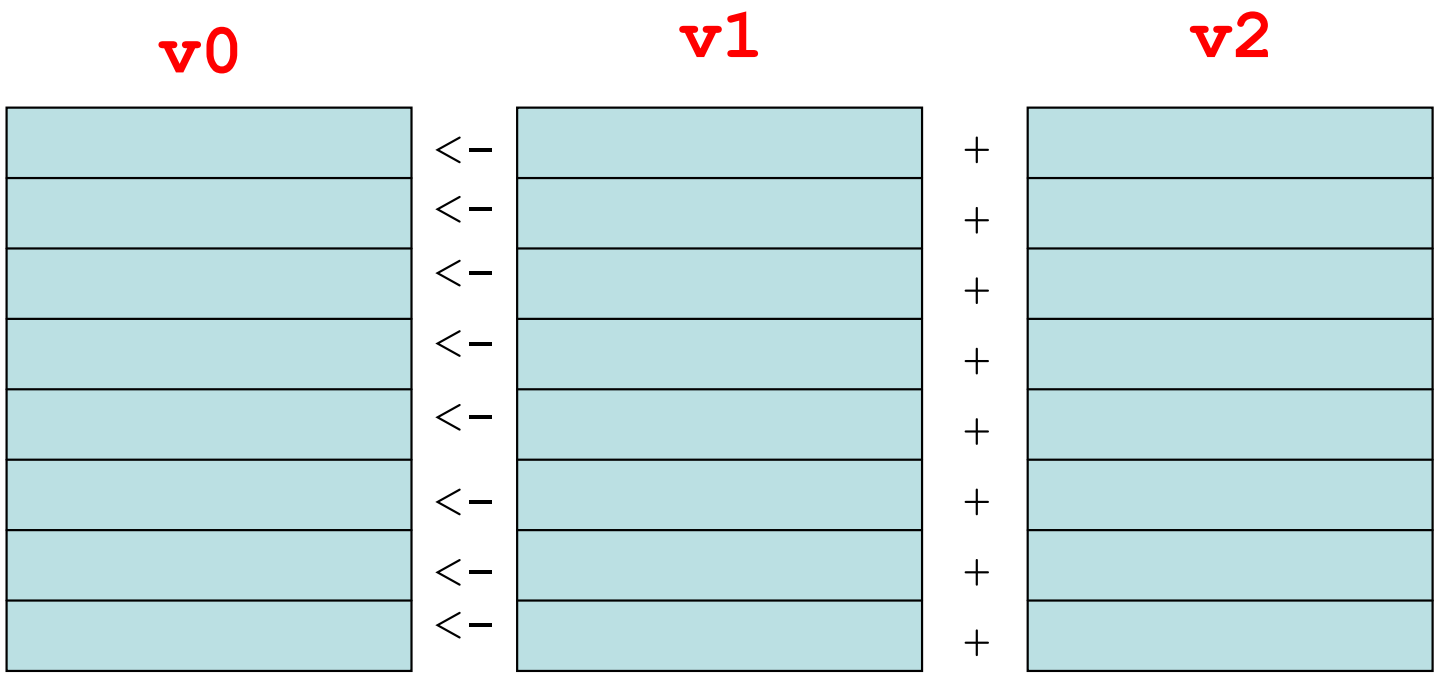
A ***vector*** is a giant register that behaves like a collection of regular registers, except these registers all simultaneously perform the same operation on multiple sets of operands, producing multiple results.

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

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

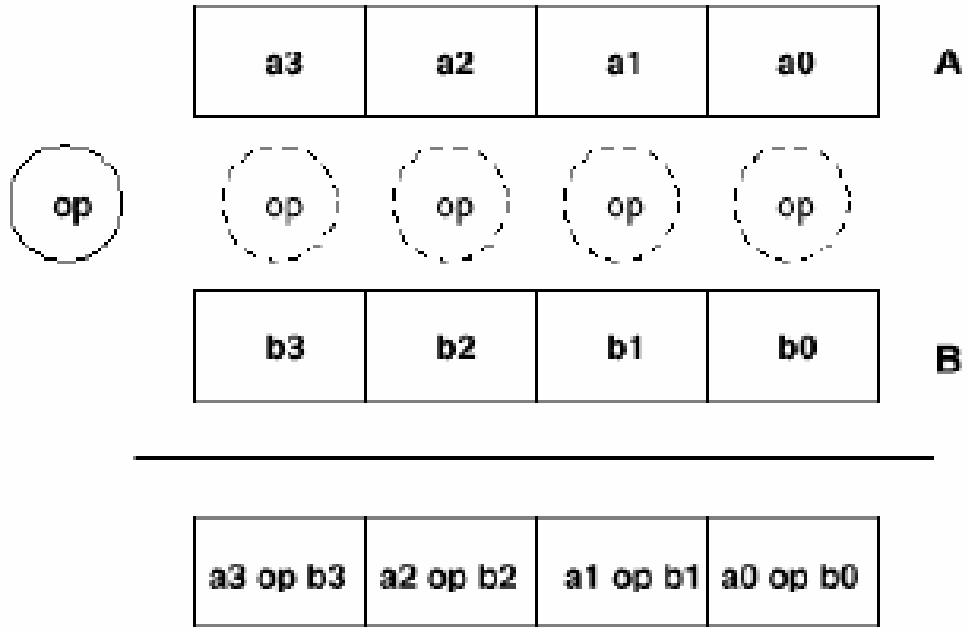
A ***vector instruction*** is an instruction that performs the same operation simultaneously on all of the individual registers of a vector register.

Vector Register



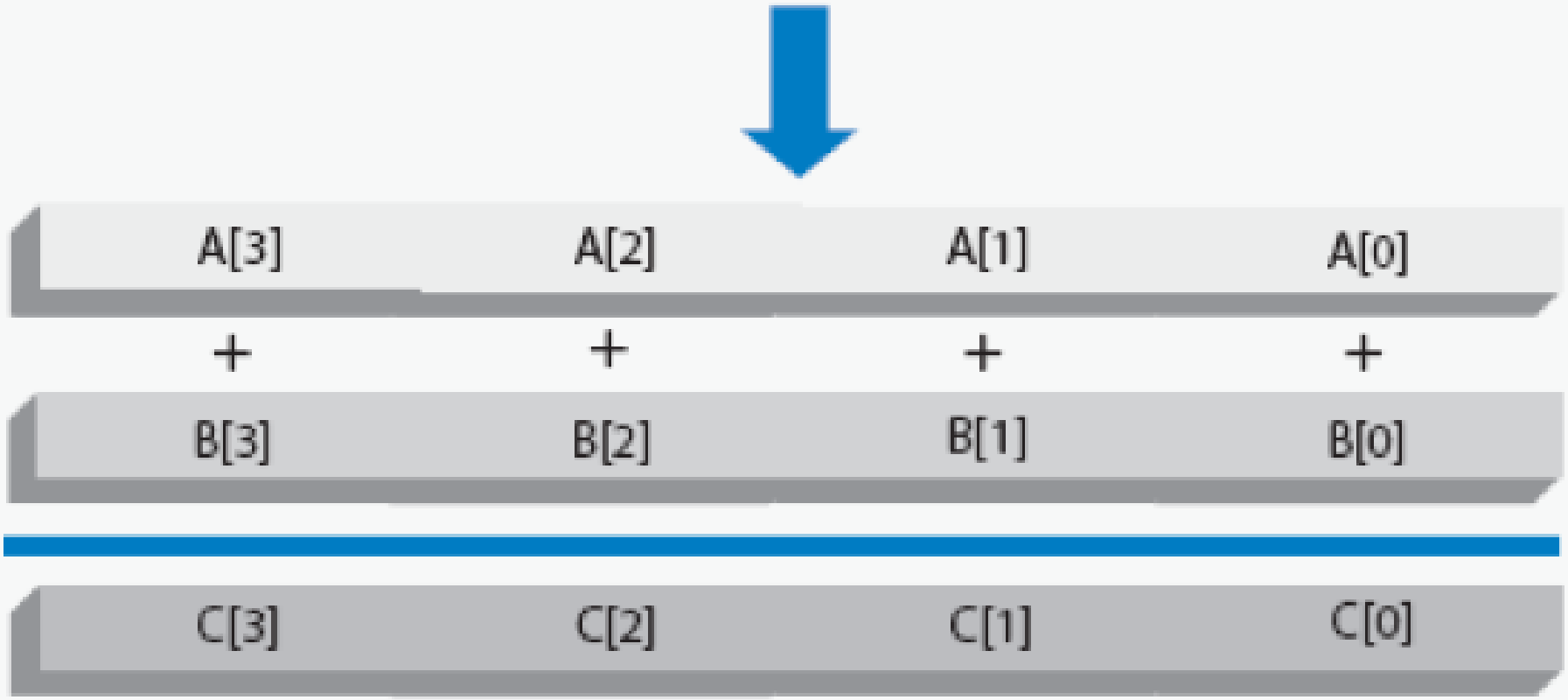

```
v0 <- v1 + v2
```

Single Instruction on Multiple Data (SIMD)



Vectorization Converts Loops

```
for (I=0; I<=MAX; I++)
c [I]=a[I]+b[I];
```



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

In the 1990s, though, they weren't very popular. Why?

Well, vectors aren't used by many commercial codes. So most chip makers didn't bother with vectors.

So, if you wanted vectors, you had to pay a lot of **extra money** for them.

The Pentium III Intel **reintroduced** very small integer vectors (2 operations at a time). The Pentium4 added floating point vector operations, also of size 2.

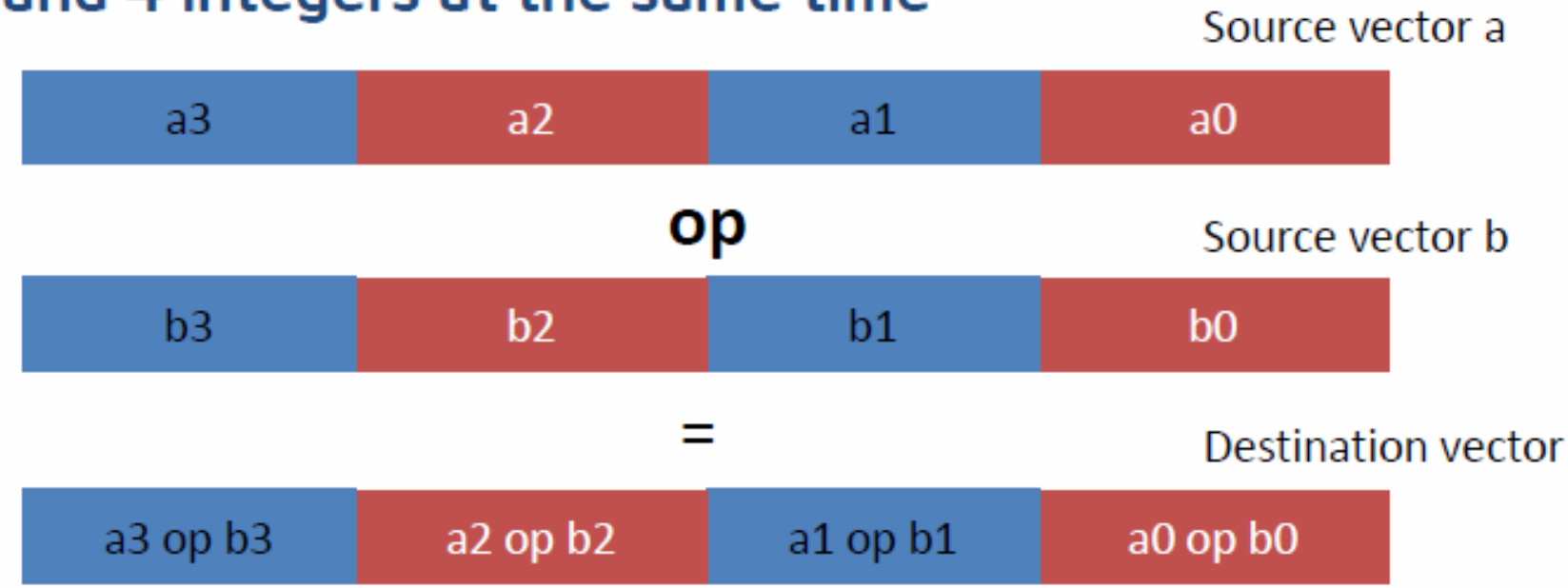
The Core family has doubled the vector size to 4, and

Intel's Sandy Bridge (2011) to 8.

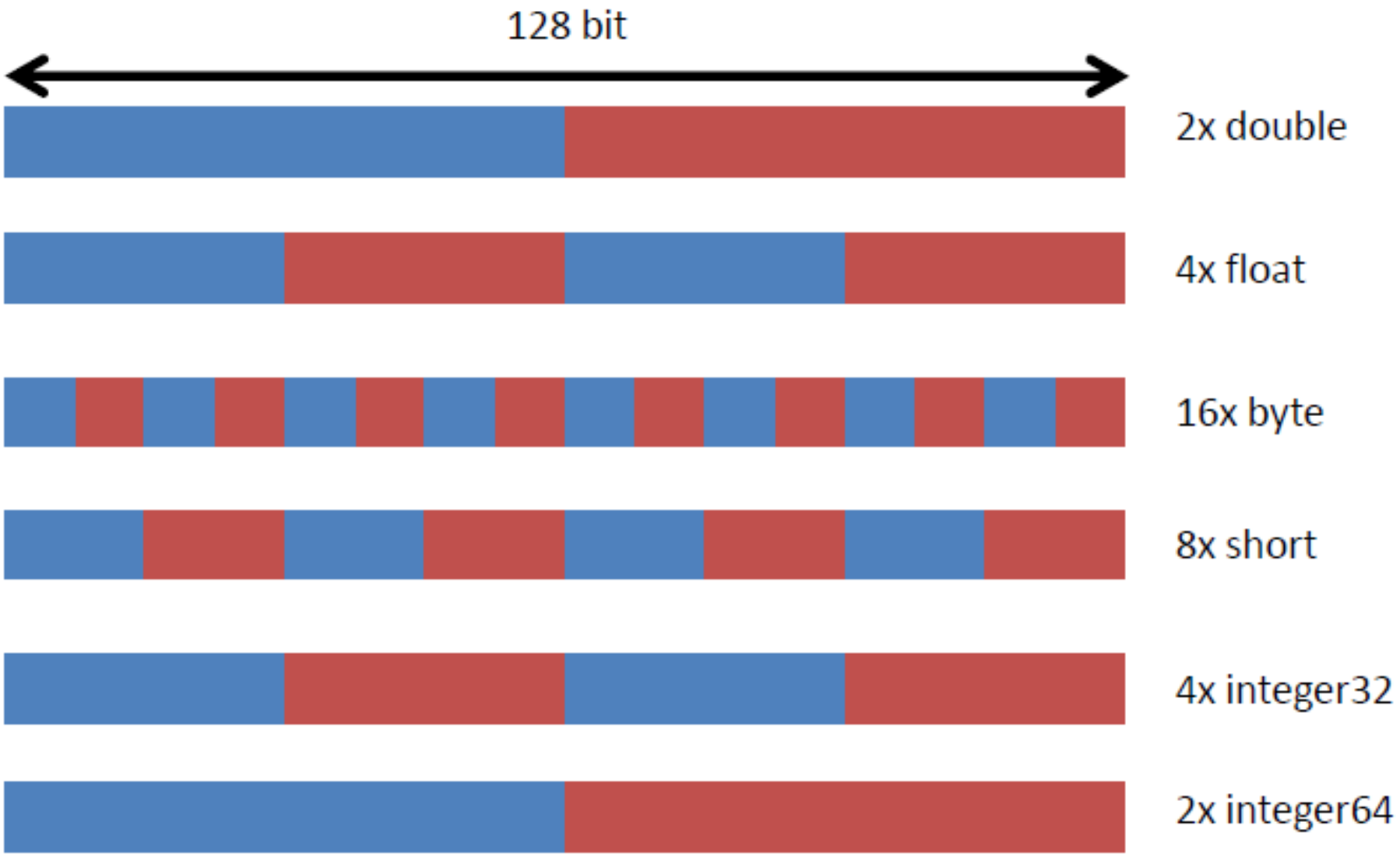
SSE (Streaming SIMD Extensions) and its later versions (also Intel's **AVX**-- Advanced Vector Extension) provide for vectorization.

What is SSE (and related instruction sets)

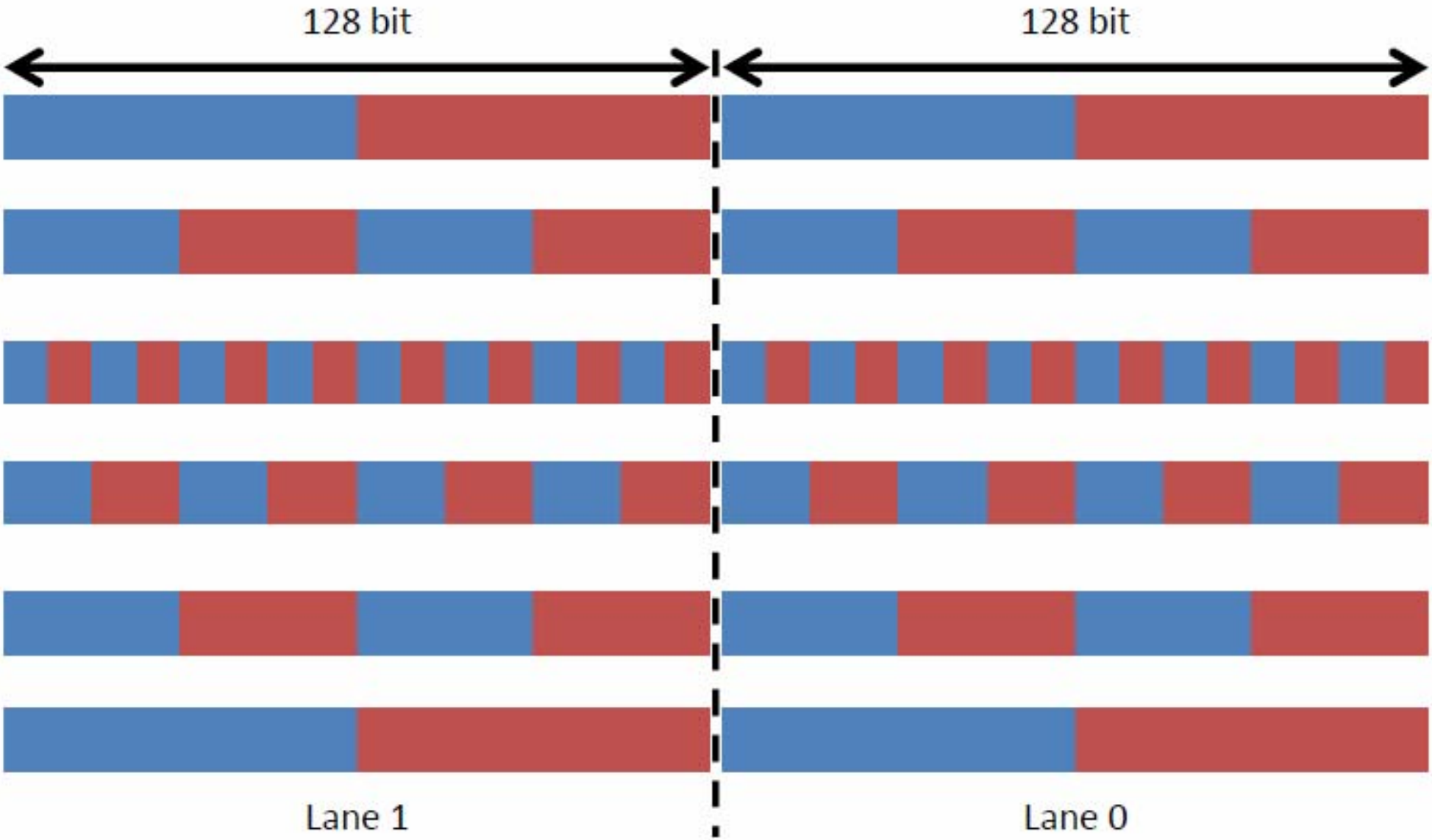
- SSE: Streaming SIMD extension
- SIMD: Single instruction, Multiple Data (Flynn's Taxonomy)
- SSE allows the identical treatment of 2 double, 4 floats and 4 integers at the same time



SSE Data Types



AVX Data Types



The automatic vectorizer (also called the auto-vectorizer) is a component of the Intel® compiler that automatically uses SIMD instructions in

1. the Intel® Streaming SIMD Extensions (Intel® SSE, SSE2, SSE3 and SSE4 Vectorizing Compiler and Media Accelerators),
2. the Supplemental Streaming SIMD Extensions (SSSE3) instruction sets,
3. the Intel® Advanced Vector Extension instruction set.

The vectorizer detects operations in the program that can be done in parallel, and then converts the sequential operations like one SIMD instruction that processes 2, 4, 8 or up to 16 elements in parallel, depending on the data type.

Summarizing:

- "Scalar" CPU executes one instruction at a time
  - includes pipelined processors
- "Superscalar" can execute more than one unrelated instructions at a time
  - ADD X + Y, MUL W * Z
- "Vector" CPU executes one instruction at a time, but on vector data
  - X[0:7] + Y[0:7] is one instruction, whereas on a scalar processor, you would need eight

Example: IBM POWER4

8-way Superscalar: can execute up to 8 operations at the same time

- 2 integer arithmetic or logical operations, and
- 2 floating point arithmetic operations, and
- 2 memory access (load or store) operations, and
- 1 branch operation, and
- 1 conditional operation

With Super-pipelining, IBM Power4 can have over 200 different operations “in flight” at the same time

What determines the degree of ILP?

- **dependences**: property of the program
- **hazards**: property of the pipeline

A dependence indicates what program components (statements, loop iterations, etc) may be executed ***ignoring the sequence of events specified by the programmer*** without changing the output.
Program components that are not dependent on each other can be executed in parallel.

Types of Dependences/Dependencies

- Data Dependence (True Dependence)
  - RAW: Read-After-Write
- Name Dependencies
  - WAR: Write-After-Read (Anti-Dependence)
  - WAW: Write-After-Write (Output Dependence)
- Control Dependence
  - When following instructions depend on the outcome of a previous branch/jump

What Is Dependency Analysis?

Dependency analysis describes of how different parts of a program affect one another, and how various parts require other parts in order to operate correctly.

Dependency analysis is one of the major roles of a modern compilers

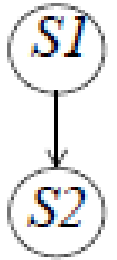
A *data (also a name) dependency* governs how different pieces of data affect each other.

A *control dependency* governs how different sequences of instructions affect each other.

“Data” and “Name” dependencies

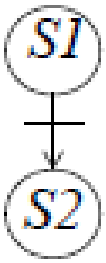
Flow Dependence (True Dependence)

```
S1 X=A+B
S2 C=X+1
```



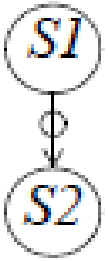
Anti Dependence

```
S1 A=X+B
S2 X=C+D
```



Output Dependence

```
S1 X=A+B
    . . .
S2 X=C+D
```



Control Dependencies

Every program has a well-defined *flow of control* that moves from instruction to instruction.

This flow can be affected by several kinds of operations:

- Branches (if, select case/switch)
- Loops
- Function/subroutine calls
- I/O (typically implemented as calls)

Dependencies affect parallelization!

Branch Dependency (F90)

```
y = 7
IF (x /= 0) THEN
    y = 1.0 / x
END IF
```

Note that `(x /= 0)` means "**x** not equal to zero."

The value of **y** depends on what the condition **(x /= 0)** evaluates to:

- If the condition **(x /= 0)** evaluates to **.TRUE.**, then **y** is set to **1.0 / x**. (1 divided by **x**).
- Otherwise, **y** remains **7**.

Loop Carried Dependency (F90)

```
DO i = 2, length
   a(i) = a(i-1) + b(i)
END DO
```

Here, each iteration of the loop depends on the previous:
iteration **i=3** depends on iteration **i=2**,
iteration **i=4** depends on iteration **i=3**,
iteration **i=5** depends on iteration **i=4**, and so on.

This is sometimes called a *loop carried dependency*.

There is no way to execute iteration **i** until after iteration **i-1** has completed, so this loop can't be parallelized.

Why Do We Care?

- Loops are very common in many programs.
- 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.
- Loops are the favorite control structures of High Performance Computing. Both hardware vendors and compiler writers build for *optimizing* loop performance using instruction-level parallelism (superscalar, pipelining, and vectorization).

Loop carried dependencies affect whether a loop can be parallelized, and how much.

Loop or Branch Dependency? (F90)

Is this a loop carried dependency or a branch dependency?

```
DO i = 1, length
   IF (x(i) /= 0) THEN
      y(i) = 1.0 / x(i)
   END IF
END DO
```

Call Dependency Example (F90)

```
x = 5
y = myfunction(7)
z = 22
```

The flow of the program is interrupted by the call to **myfunction**, which takes the execution to somewhere else in the program.

It's similar to a branch dependency.

I/O Dependency (F90)

```
x = a + b
PRINT *, x
y = c + d
```

Typically, I/O is implemented by hidden subroutine calls, so we can think of this as equivalent to a call dependency.

Hazards

- When two instructions that have one or more dependences between them occur close enough that changing the instruction order will change the outcome of the program
- Not all dependencies lead to hazards!

- If data dependence causes a hazard in pipeline, then the hazard is called a Read After Write (RAW) hazard
- If anti-dependence causes a hazard in the pipeline, then the hazard is called a Write After Read (WAR) hazard
- If anti-dependence caused a hazard in the pipeline, called a Write After Write (WAW) hazard

Limits to pipelining

Hazards prevent next instruction from executing during its designated clock cycle

- Structural hazards: attempt to use the same hardware to do two different things at once
- Data hazards: Instruction depends on result of prior instruction still in the pipeline
- Control hazards: Caused by delay between the fetching of instructions and decisions about changes in control flow (branches and jumps).

ILP and Dependencies, Hazards

- HW/SW must preserve program order: code must give the same results as if instructions were executed sequentially in the original order of the source program
  - Dependences are a property of programs
- The presence of a dependence indicates the potential for a hazard, but the existence of an actual hazard and the length of any stall are properties of the pipeline
- Data dependencies are important. They
  1) Indicate the possibility of a hazard
  2) Determine the order in which results must be calculated
  3) Set upper bounds on how much parallelism can possibly be exploited to speedup a program.
- HW/SW goal: exploit parallelism by preserving program order only where it affects the outcome of the program

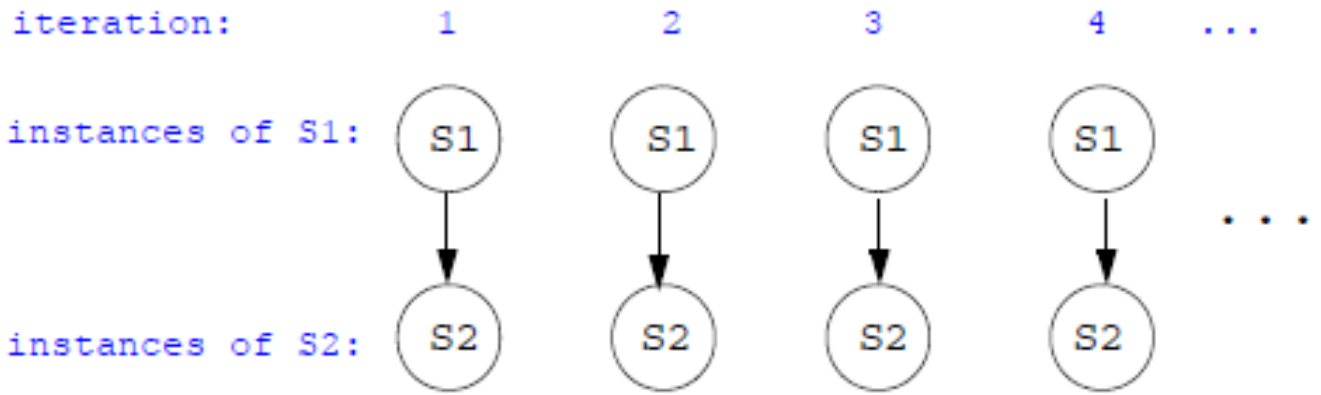
Why Does Order Matter?

- Dependencies can affect whether we can execute a particular part of the program in parallel.
- If we cannot execute that part of the program in parallel, then it'll be SLOW.

Dependences in loops

Dependences in loops are easy to understand if loops are unrolled. Now the dependences are between statement “instances”

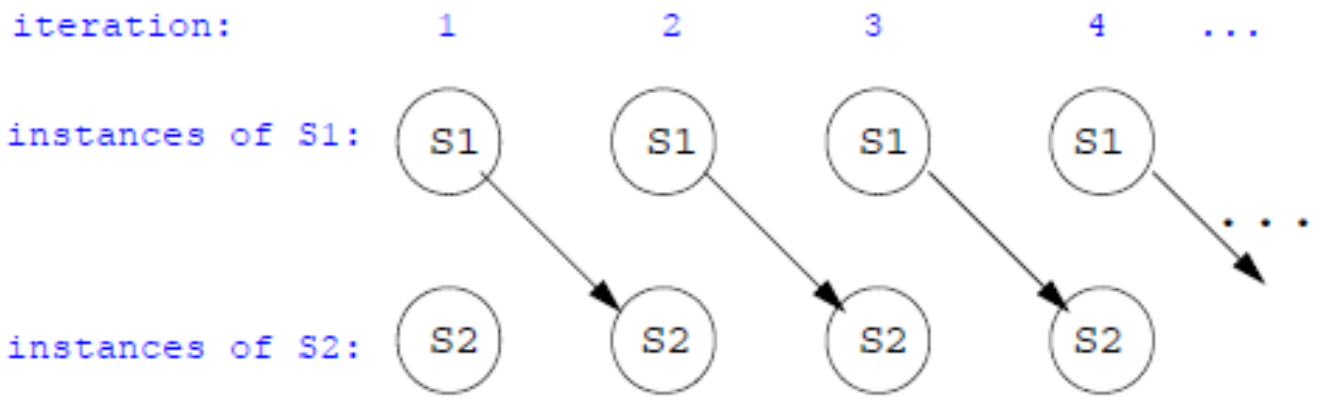
```
        do i=1 to n
S1      a(i)=b(i)+1
S2      c(i)=a(i)+2
```



Dependences in loops

A little more complex example is

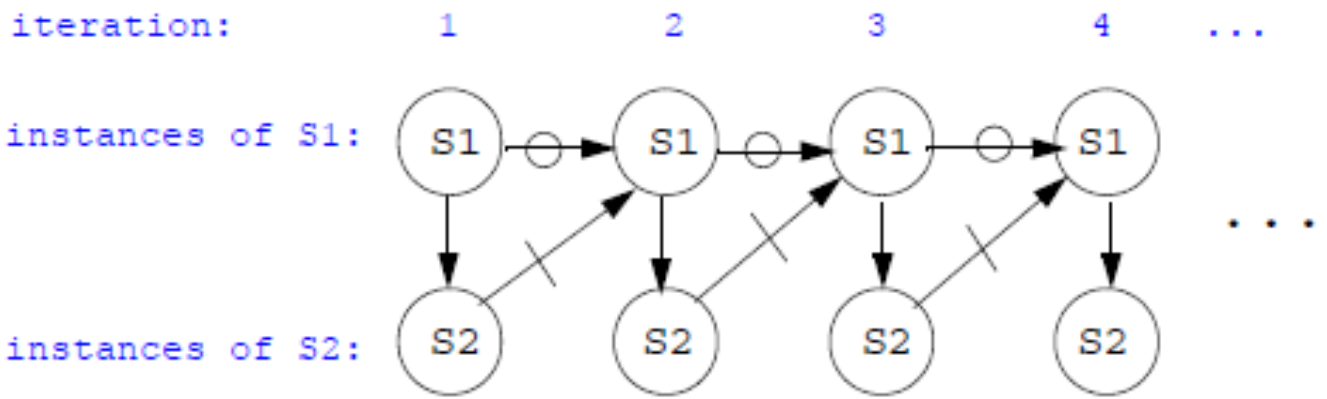
```
      do i=1 to n
S1      a(i)=b(i)+1
S2      c(i)=a(i-1)+2
```



Dependences in loops

Even more complex:

```
      do i=1 to n
S1      a=b(i)+1
S2      c(i)=a+2
```



Superscalar Loops (F90)

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

Each of the iterations is **completely independent** of all of the other iterations; for example,

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

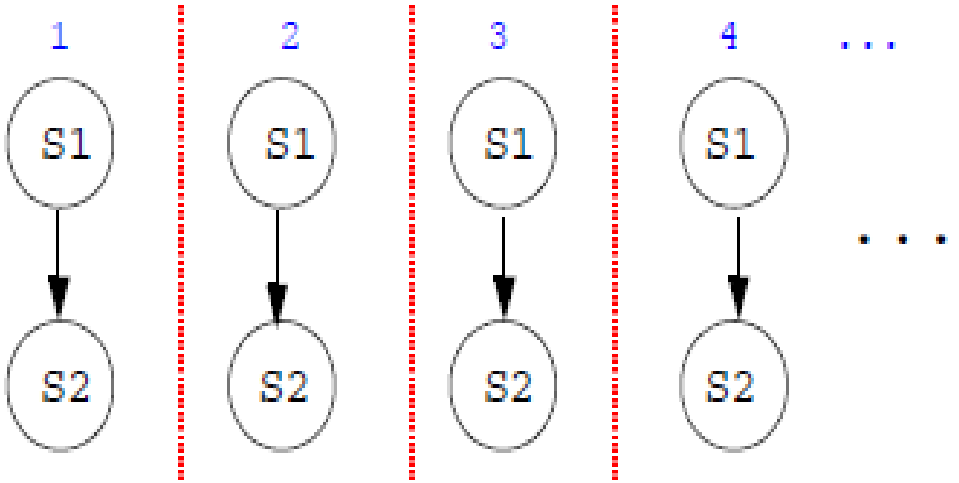
has nothing to do with

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

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

Optimizing with dependences

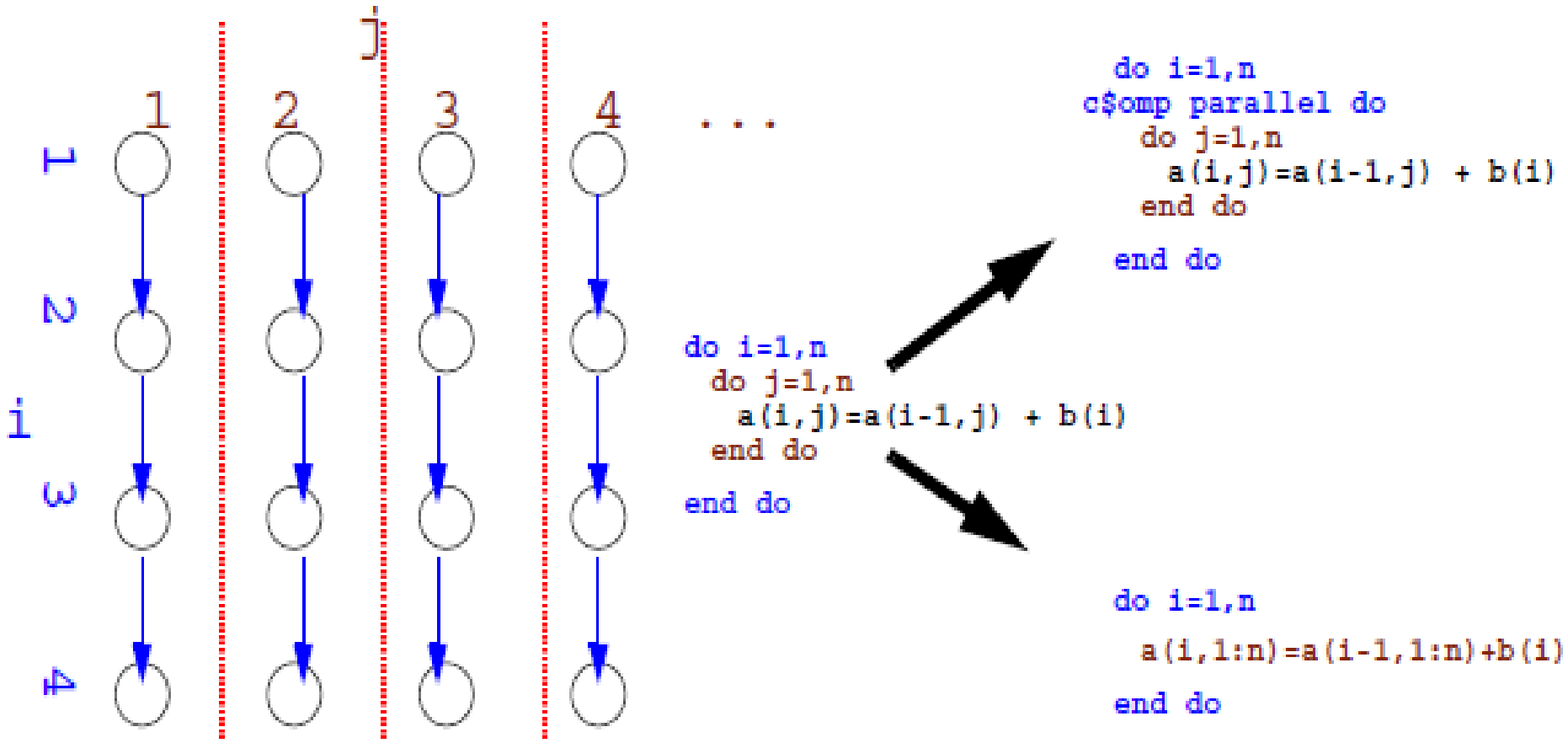
It is valid to parallelize/vectorize a loop if no dependences cross its iteration boundaries:



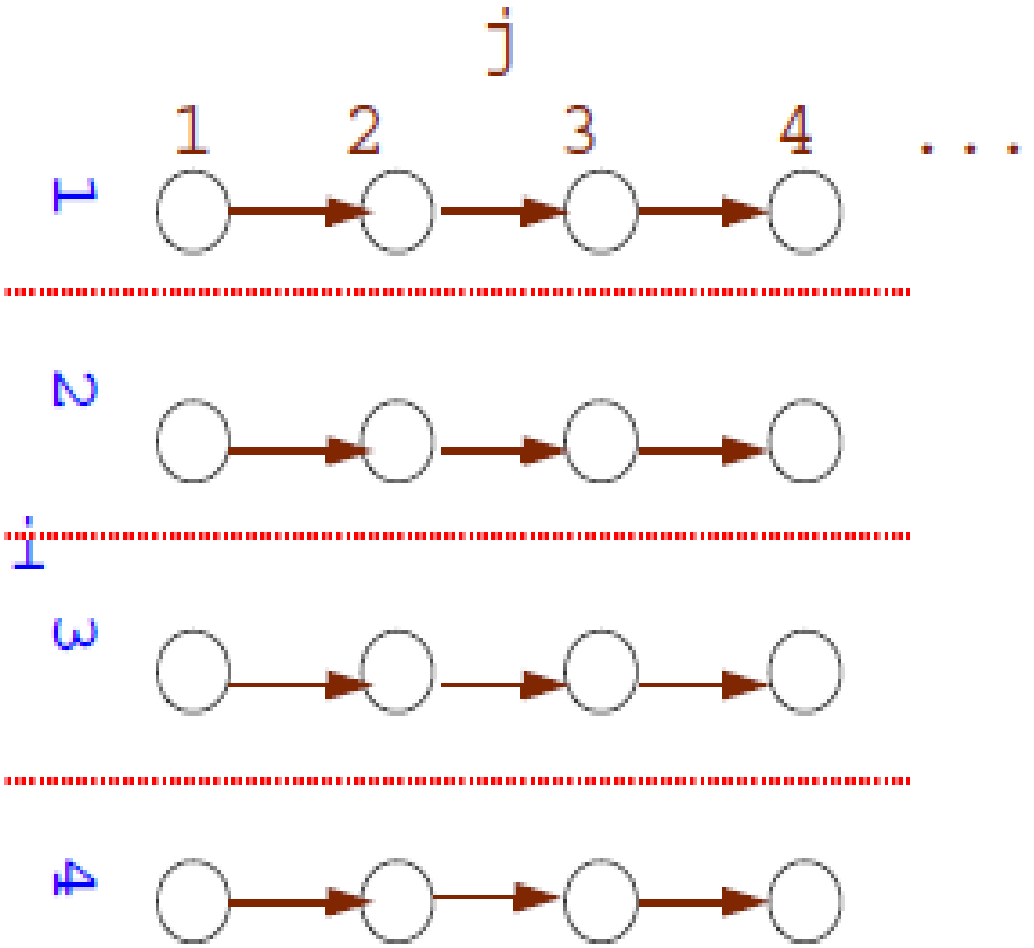

```
do i=1 to n
  S1 a(i)=b(i)+1
  S2 c(i)=a(i)+2
```

```
c$omp parallel do
  do i=1 to n
  S1 a(i)=b(i)+1
  S2 c(i)=a(i)+2
```

```
a(1:n) = b(1:n) + 1
c(1:n) = a(1:n) + 2
```



Note: For vectorization in Fortran 90 loop must not contain other loops inside. This limitation does not apply to CUDA.




```
do i=1,n
   do j=1,n
      a(i,j)=a(i,j-1) + b(i)
   end do
end do
```

```
c$omp parallel do
do i=1,n
   do j=1,n
      a(i,j)=a(i,j-1) + b(i)
   end do
end do
```

Note: Outer loops are typically preferred for parallelization (if we must choose) to reduce overhead.

Tricks That Compilers Play

Scalar Optimizations

- Copy Propagation
- Constant Folding
- Dead Code Removal
- Strength Reduction
- Common Subexpression Elimination
- Variable Renaming
- Loop Optimizations

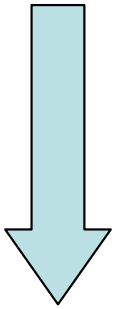
Not every compiler does all of these, so it sometimes can be worth doing these by hand.

Copy Propagation (F90)

Before

```
x = y
z = 1 + x
```

Has data dependency



Compile

After

```
x = y
z = 1 + y
```

No data dependency

Constant Folding (F90)

Before

```
add = 100
aug = 200
sum = add + aug
```

After

```
sum = 300
```

Notice that `sum` is actually the sum of two constants, so the compiler can precalculate it, eliminating the addition that otherwise would be performed at runtime.

Dead Code Removal (F90)

Before

```
var = 5
PRINT *, var
STOP
PRINT *, var * 2
```

After

```
var = 5
PRINT *, var
STOP
```

Since the last statement never executes, the compiler can eliminate it.

Strength Reduction (F90)

Before

```
x = y ** 2.0
a = c /  2.0
```

After

```
x = y * y
a = c * 0.5
```

Raising one value to the power of another, or dividing, is more expensive than multiplying. If the compiler can tell that the power is a small integer, or that the denominator is a constant, it'll use multiplication instead.

Note: In Fortran, "`y ** 2.0`" means "y to the power 2."

Common Subexpression Elimination (F90)

Before

```
d = c * (a / b)
e = (a / b) * 2.0
```

After

```
adivb = a / b
d = c * adivb
e = adivb * 2.0
```

The subexpression `(a / b)` occurs in both assignment statements, so there's no point in calculating it twice.

This is typically only worth doing if the common subexpression is expensive to calculate.

Variable Renaming (F90)

Before

```
x = y * z
q = r + x * 2
x = a + b
```

After

```
x0 = y * z
q = r + x0 * 2
x = a + b
```

The original code has an **output dependency**, while the new code **doesn't** – but the final value of **x** is still correct.

Loop Optimizations

- Hoisting Loop Invariant Code
- Unswitching
- Iteration Peeling
- Index Set Splitting
- Loop Interchange
- Unrolling
- Loop Fusion
- Loop Fission

Not every compiler does all of these, so it sometimes can be worth doing some of these by hand.

Hoisting Loop Invariant Code (F90)

Code that doesn't change inside the loop is known as ***loop invariant***. It doesn't need to be calculated over and over.

Before

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

After

```
temp = c * d
DO i = 1, n
  a(i) = b(i) + temp
END DO
e = g(n)
```

Unswitching (F90)

```
DO i = 1, n
  DO j = 2, n
    IF (t(i) > 0) THEN
      a(i,j) = a(i,j) * t(i) + b(j)
    ELSE
      a(i,j) = 0.0
    END IF
  END DO
END DO
```

The condition is `j`-independent.

Before

```
DO i = 1, n
  IF (t(i) > 0) THEN
    DO j = 2, n
      a(i,j) = a(i,j) * t(i) + b(j)
    END DO
  ELSE
    DO j = 2, n
      a(i,j) = 0.0
    END DO
  END IF
END DO
```

So, it can migrate outside the `j` loop.

After

Iteration Peeling (F90)

Before

```
DO i = 1, n
  IF ((i == 1) .OR. (i == n)) THEN
    x(i) = y(i)
  ELSE
    x(i) = y(i + 1) + y(i - 1)
  END IF
END DO
```

We can eliminate the IF by ***peeling*** the weird iterations.

After

```
x(1) = y(1)
DO i = 2, n - 1
  x(i) = y(i + 1) + y(i - 1)
END DO
x(n) = y(n)
```

Index Set Splitting (F90)

```
DO i = 1, n
   a(i) = b(i) + c(i)
   IF (i > 10) THEN
     d(i) = a(i) + b(i - 10)
   END IF
END DO
```

Before

```
DO i = 1, 10
   a(i) = b(i) + c(i)
END DO
DO i = 11, n
   a(i) = b(i) + c(i)
   d(i) = a(i) + b(i - 10)
END DO
```

After

Note that this is a generalization of **peeling**.

Loop Interchange (F90)

Before

```
DO i = 1, ni
  DO j = 1, nj
    a(i,j) = b(i,j)
  END DO
END DO
```

After

```
DO j = 1, nj
  DO i = 1, ni
    a(i,j) = b(i,j)
  END DO
END DO
```

Array elements `a(i,j)` and `a(i+1,j)` are near each other in memory, while `a(i,j+1)` may be far, so it makes sense to make the `i` loop be the inner loop. (This is reversed in C, C++ and Java.)

This technique facilitates efficient exploitation of the phenomenon of **locality of reference.**

Unrolling (F90)

Before

```
DO i = 1, n
   a(i) = a(i)+b(i)
END DO
```

After

```
DO i = 1, n, 4
   a(i)   = a(i)   + b(i)
   a(i+1) = a(i+1) + b(i+1)
   a(i+2) = a(i+2) + b(i+2)
   a(i+3) = a(i+3) + b(i+3)
END DO
```

You generally **shouldn't** unroll by hand.

Why Do Compilers Unroll?

A loop with a lot of operations gets better performance (up to some point), especially if there are lots of arithmetic operations but few main memory loads and stores.

Unrolling creates multiple operations that typically load from the same, or adjacent, cache lines.

So, an unrolled loop has more operations without increasing the memory accesses by much.

Also, unrolling decreases the number of comparisons on the loop counter variable, and the number of branches to the top of the loop.

Loop Fusion (F90)

```
DO i = 1, n
  a(i) = b(i) + 1
END DO

DO i = 1, n
  c(i) = a(i) / 2
END DO

DO i = 1, n
  d(i) = 1 / c(i)
END DO
```

Before

```
DO i = 1, n
  a(i) = b(i) + 1
  c(i) = a(i) / 2
  d(i) = 1 / c(i)
END DO
```

After

As with unrolling, this has fewer branches. It also has fewer total memory references.

Loop Fission (F90)

```
DO i = 1, n
  a(i) = b(i) + 1
  c(i) = a(i) / 2
  d(i) = 1 / c(i)
END DO
```

Before

```
DO i = 1, n
  a(i) = b(i) + 1
END DO
```

After

```
DO i = 1, n
  c(i) = a(i) / 2
END DO
```

```
DO i = 1, n
  d(i) = 1 / c(i)
END DO
```

Fission reduces the cache footprint and the number of operations per iteration.

To Fuse or to Fizz?

The question of when to perform fusion versus when to perform fission, like many many optimization questions, is highly dependent on the application, the platform and a lot of other issues that get very, very complicated.

Compilers don’t always make the right choices.

That’s why it’s important to examine the actual behavior of the executable.

Inlining (F90)

Before

```
DO i = 1, n
   a(i) = func(i)
END DO
…
REAL FUNCTION func(x)
   …
   func = x * 3
END FUNCTION func
```

After

```
DO i = 1, n
   a(i) = i * 3
END DO
```

When a function or subroutine is ***inlined***, its contents are transferred directly into the calling routine, eliminating the overhead of making the call.

Tasks of the compiler

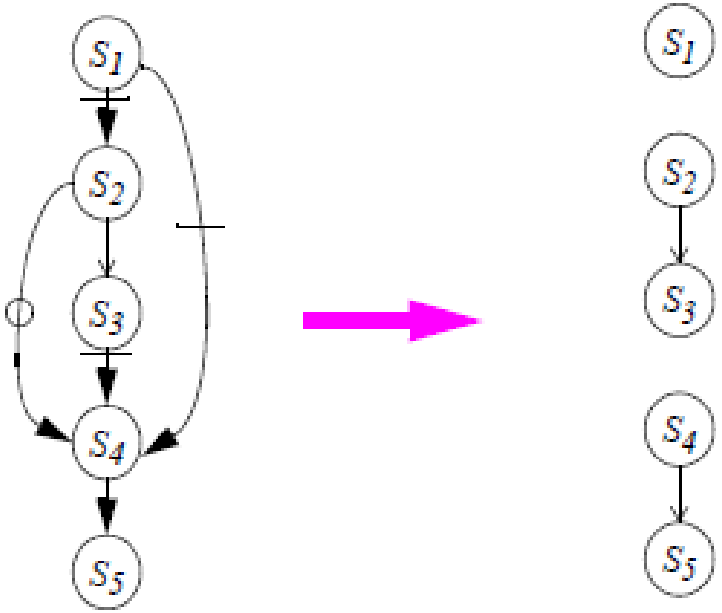
- It transform program to remove dependences.
- Use dependence to reorganize computation for parallelism and locality.

However, manual transformation by hand can sometimes be more useful.

Transform programs: renaming

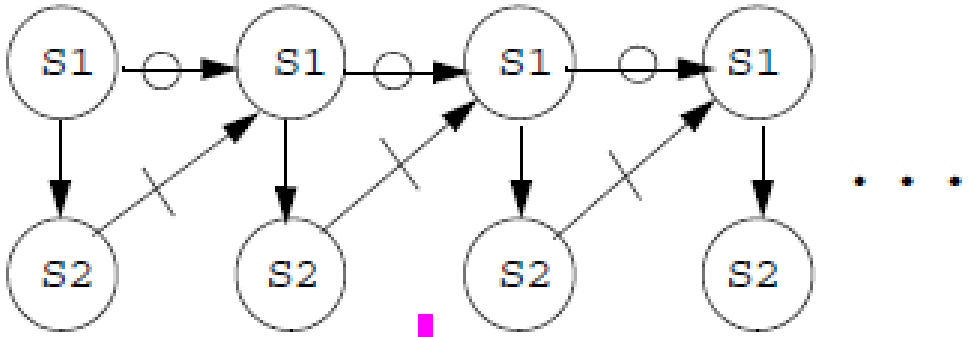
```
S1 A=X+B

S2 X=Y+1

S3 C=X+B

S4 X=Z+B

S5 D=X+1
```

```
S1 A=X+B

S2 X1=Y+1

S3 C=X1+B

S4 X2=Z+B

S5 D=X2+1
```

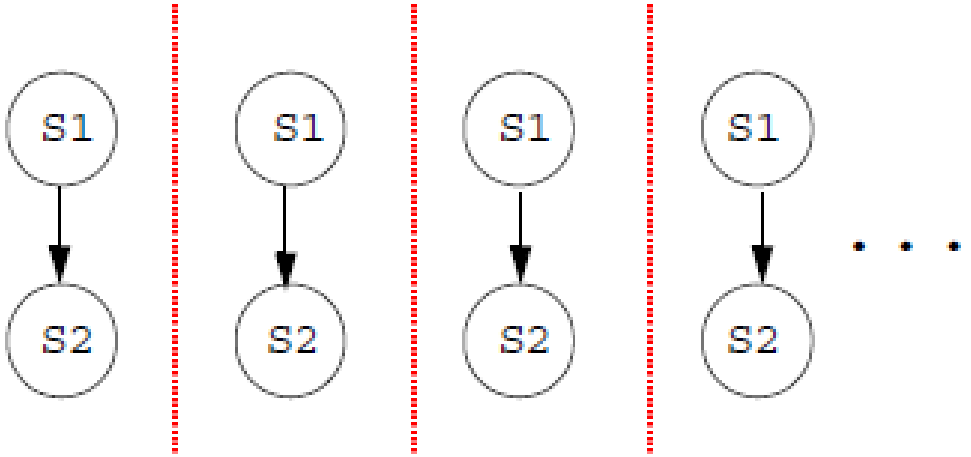


Transform programs: scalar expansion

```
    do i=1,n
S1      a=b(I)+1
S2      c(I)=a+d(I)
    end do
```




```
    do i=1,n
S1      a1(i)=b(i)+1
S2      c(i)=A1(i)+d(i)
    end do
    a=a1(n)
```



Transform programs: scalar expansion

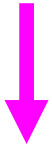
- The rule is simple: it is valid to expand a scalar if it is always assigned before it is used within the body of the loop and if either
  - its final value at the end of the loop is known, or
  - it is never used after the loop.

- To avoid an unnecessary increase in the use of memory we could do one of two things:
  - Stripmine the loop to vectorize with short vectors
  - Propagate or privatize instead of expand

Stripmining and Scalar expansion

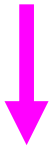
It is a simple transformation and always valid:

```
     DO I=1,r*q
S1:    A1(I)=B(I)+1
S2:    C(I)=A1(I)+D(I)
     END DO
     A=A1(N)
```



Stripmine

```
     DO M=1,r*q,q
      DO I=M,M+q-1
S1:      A1(I)=B(I)+1
S2:      C(I)=A1(I)+D(I)
      END DO
     END DO
    A=A1(N)
```



Vectorize inner loop

```
     DO M=1,r*q,q
S1:      A1(M:M+q-1)=B(M:M+q-1)+1
S2:      C(M:M+q-1)=A1(M:M+q-1)+D(M:M+q-1)
     END DO
     A=A1(N)
```

Algorithm replacement

Some program patterns occur frequently in programs.
Programmers should replace those with their parallel version.
e.g.

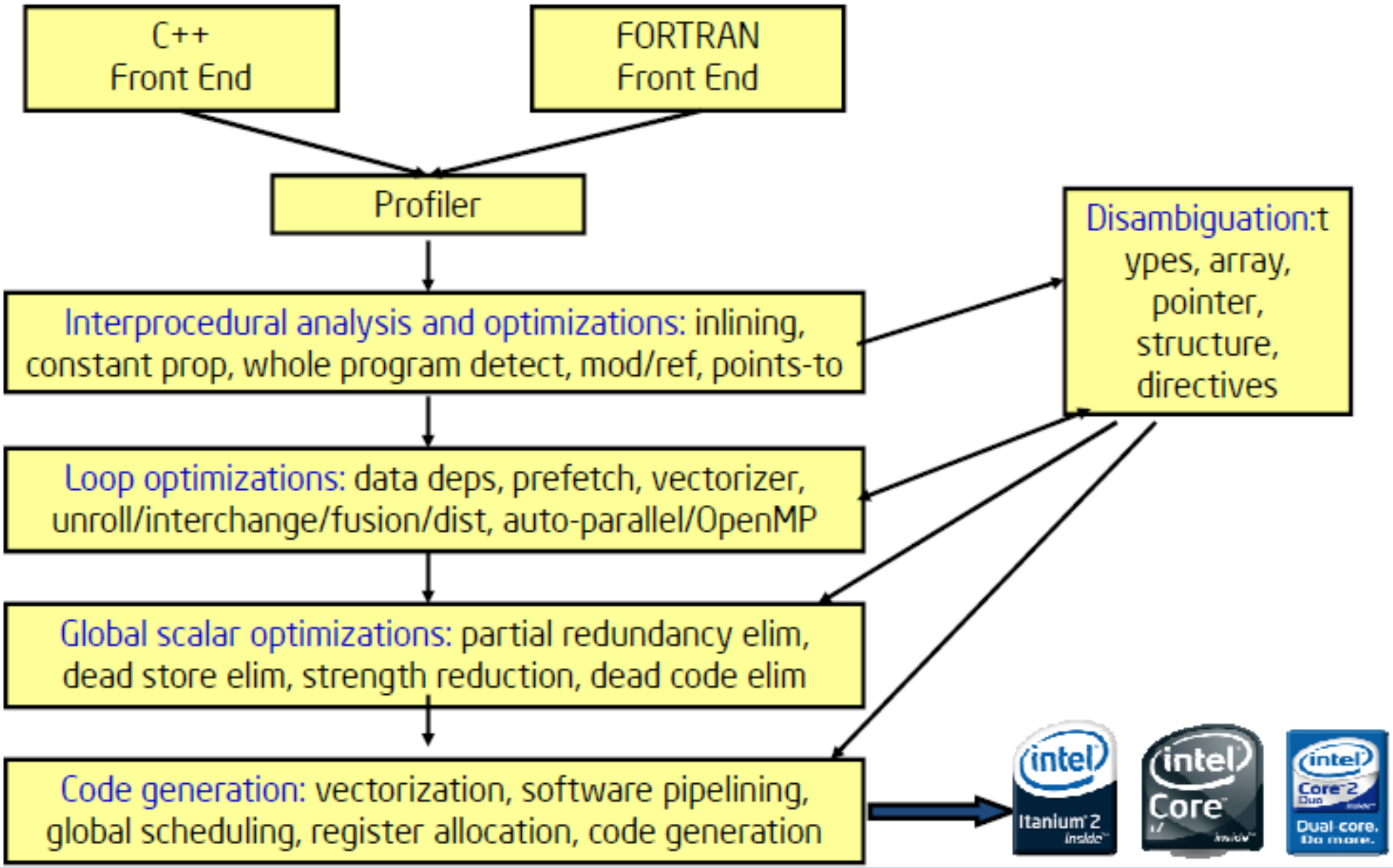
```
DO I=1,N
  A(I)=A(I-1)+B(I)
END DO
```

```
A(1:N)=REC1N(B(1:N),A(0),N)
```

```
X=A(1)
DO I=2,N
 IF(X.GT.A(I))X=A(I)
END DO
```

```
X=MIN(A(1:N))
```

Intel® Compiler Architecture



A few General Switches

Functionality	Linux
Disable optimization	-O0
Optimize for speed (no code size increase), no SWP	-O1
Optimize for speed (default)	-O2
High-level optimizer (e.g. loop unroll), -ftz (for Itanium)	-O3
Vectorization for x86, -xSSE2 is default	<many options>
Aggressive optimizations (e.g. -ipo, -O3, -no-prec-div, -static -xHost for x86 Linux*)	-fast
Create symbols for debugging	-g
Generate assembly files	-S
Optimization report generation	-opt-report
OpenMP support	-openmp
Automatic parallelization for OpenMP* threading	-parallel

Architecture Specific Switches

Functionality	Linux
Optimize for current machine	-xHOST
Generate SSE v1 code	-xSSE1
Generate SSE v2 code (may also emit SSE v1 code)	-xSSE2
Generate SSE v3 code (may also emit SSE v1 and v2 code)	-xSSE3
Generate SSE v3 code for Atom-based processors	-xSSE_ATOM
Generate SSSE v3 code (may also emit SSE v1, v2, and v3 code)	-xSSSE3
Generate SSE4.1 code (may also emit (S)SSE v1, v2, and v3 code)	-xSSE4.1
Generate SSE4.2 code (may also emit (S)SSE v1, v2, v3, and v4 code)	-xSSE4.2
Generate AVX code	-xAVX

CEAN: C/C++ Extensions for Array Notation

Add array notation to C/C++ similar to what was done for Fortran language in Fortran90 to express data parallel (e.g. SSE-) code explicitly

Sample:

```
// Traditional:
for (i=0; i < M-K; i++)
{
    s = 0;
    for (j = 0; j < K; j++)
        s += x[i+j] * c[j]
    y[i] = s;
}
```

```
// CEAN Version - outer loop
// "vectorized"

y[0:M-K] = 0;

for (j = 0; j < K; j++)
    y[0:M-K]+= x[j:M-K]*c[j];
```

- Nothing totally new; many similar initiatives in the past 30 years
  - E.g.: "Vector C" for Control Data 205, +25 years ago
- CEAN requires switch –farray-notation in 12.0 Compiler

CEAN Array Sections

- Array sections
  - Specification per dimension is lower bound : count [:stride]
  - This is different from Fortran lower bound : upper bound : [stride]
  - Samples:

```
A[:]     // All of vector A
B[2:6]   // Elements 2 to 7 of vector B
C[:][5]  // Column 5 of matrix C
D[0:3:2] // Elements 0,2,4 of vector D
```

- Basic, data parallel operations on array sections
  - Support for most C/C++ arithmetic and logic operators like “+”, “/”, ”<“, “&&”, “+=“, …
  - The shape of the sections must be identical, scalars are expanded implicitly
  - Sample:

```
( a[0:s]+b[5:s] )* pi  // pi * {a[i]+b[i+5], (i=0;i<s;i++)}
```

CEAN Assignment

Assignment evaluates LHS and assigns values to RHS in parallel

- The shapes of the RHS and LHS array section must be the same
- Scalar is expanded automatically
- Conceptually, RHS is evaluated completely before LHS
  - Compiler ensures semantic in code being generated
  - This can be a critical performance issue (temporary array variable)
- Samples:

```
a[:][:] = b[:][2][:] + c;
e[:] = d;              // scalar expansion
e[:] = b[:][1][:];     // error, shapes different
a[:][:] = e[:];        // error, shapes different
a[b[0:s]] = c[:]       // scatter operation
c[0:s] = a[b[:]]       // gather operation
```

CEAN Assignment

Assignment evaluates LHS and assigns values to RHS in parallel

- The shapes of the RHS and LHS array section must be the same
- Scalar is expanded automatically
- Conceptually, RHS is evaluated completely before LHS
  - Compiler ensures semantic in code being generated
  - This can be a critical performance issue (temporary array variable)
- Samples:

```
a[:][:] = b[:][2][:] + c;
e[:] = d;              // scalar expansion
e[:] = b[:][1][:];     // error, shapes different
a[:][:] = e[:];        // error, shapes different
a[b[0:s]] = c[:]       // scatter operation
c[0:s] = a[b[:]]       // gather operation
```

CEAN - Some Advanced Features

- Functions can take array sections as arguments and can return sections
  - Any assumption on order (side effects) are a mistake
  - Compiler may generate calls to vectorized library functions
  - Examples:

```
a[:] = pow(b[:], c);   // b[:]**c
a[:] = pow(c, b[:]);   // c**b[:]
a[:] = foo(b[:])       // user defined
```

- Reductions combine elements in an array section into a single value using pre-defined operators or a user function
  - Pre-defined, e.g. _sec_reduce_{add, mul, min, ...}

```
sum = __sec_reduce_add(a[:]*b[:]);    // dot product
res = __sec_reduce(fn, a[:], 0);      // apply function fn
```

Final Message:

- Implicit parallelism is becoming increasingly valuable with the advancement in hardware and compilers capabilities
- Try to get maximum of what a compiler can do for you automatically (LEARN compiler flags/options/directives). But keep ensured whether your BULL (the compiler) is really pulling your cart FAST (not just dancing), even that to the RIGHT direction.
- Although a number of compiler/coding techniques have been discussed---but profiling and analyzing the benefit of a specific technique is a MUST TO DO.
- While optimizing the code, use profiling to find out HOT SPOTS or Bottlenecks and then focus on those, one by one, also ensuring correctness. The 10-90 rule often works. Algorithm transformation could be considered, also.

Bibliography/Sources:

- Why Computer Architecture Matters, By Cosmin Pancratov, Jacob M. Kurzer, Kelly A. Shaw, and Matthew L. Trawick. An IEEE 3 paper series.
- Material from lecture series Supercomputing in Plain English by Henry Neeman.
- Material from lecture on Implicit Parallelism at 2009 Summer School on Mulicore Programming at UPSRC Illinois.
- Material from Intel website.
- Material from MIT Open Courseware, course 6172.
- Lecture slides of Shawn Brown on Optimization and Profiling at an ICTP HPC school in 2009.
- Introduction to Parallel Computing, LLNL workshop material.

That's all folks

Thank you for your attention

Questions ?