

CSC 334 – Parallel and Distributed Computing

Instructor: Ms. Muntha Amjad
Lecture# 02: Types of Parallelism

Types of Parallelism

- **Implicit Parallelism:** The compiler, runtime, or system automatically identifies and manages parallelism without requiring the programmer to explicitly define it.
- **Explicit Parallelism:** The programmer directly specifies the parallelism, defining threads, synchronization, and communication.

Explicit Parallelism

- A programming approach where the developer explicitly instructs the system where and how to parallelize operations using dedicated language features like operators, function calls, or directives.
- Since multiple threads are executing concurrently, explicit synchronization mechanisms like barrier or locks are often required to ensure data consistency.
- Examples of explicit parallelism techniques
 - Data parallelism
 - Task parallelism
 - Loop-level parallelism
 - Thread-based parallelism

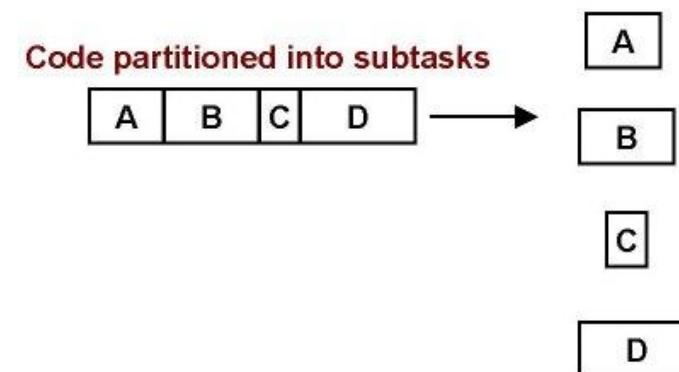
Data Parallelism

- Applying the same operation to large datasets in parallel, often using loops to iterate through data chunks. The same code segment runs concurrently on each processor, but each processor is assigned its own part of the data to work on.
 - For loops define the parallelism.
 - The iterations must be independent of each other.
 - Data parallelism is called "fine grain parallelism" because the computational work is spread into many small subtasks.

Task Parallelism

- As the opposite of data parallelism, task parallelism breaks down a problem into independent tasks that can be executed concurrently on different processors. Different operations are performed on different parts of the data.
 - Task parallelism is called "coarse grain" parallelism because the computational work is spread into just a few subtasks.
 - More code is run in parallel because the parallelism is implemented at a higher level than in data parallelism.
 - Easier to implement and has less overhead than data parallelism.

Example: an image (like color channels) are processed in parallel using separate threads, effectively treating each color channel as a separate task.



Loop-Level Parallelism

- Parallelizing operations within loops where each iteration can be executed independently
- Loop-level parallelism is a specific technique often used to achieve data parallelism, but data parallelism encompasses a wider range of strategies.
- ***Example:***

```
#pragma omp parallel for
for (int i = 0; i < n; i++)
    a[i] = b[i] + c[i];
```

Thread-Level Parallelism

- Deals with the execution of multiple threads within a program to execute different parts of the code simultaneously. The programmer explicitly creates and manages threads that run concurrently.
- Thread-level parallelism is a mechanism for achieving task parallelism by managing threads, but not all task parallelism requires explicit thread management.
- **Example:** A web server using multiple threads to handle incoming requests simultaneously, where each thread processes a single request independently.

Implicit Parallelism

- The compiler, runtime, or system automatically identifies and manages parallelism without requiring the programmer to explicitly define it.
- Examples of implicit parallelism techniques
 - Instruction-Level Parallelism (ILP)
 - Compiler-Driven Parallelism

Instruction Level Parallelism

- Instruction Level Parallelism (ILP) is a set of techniques for **executing multiple instructions at the same time within the same CPU core** (Note: 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 a lot of circuits, but at any given moment, many parts remain idle. This wastes processing power..
- **The solution:** Different parts of the CPU core work on different instructions simultaneously instead of waiting for one instruction to finish before starting the next. If a CPU can handle 10 operations at once, a program could run up to 10 times faster—though, in reality, the speedup is usually lower.

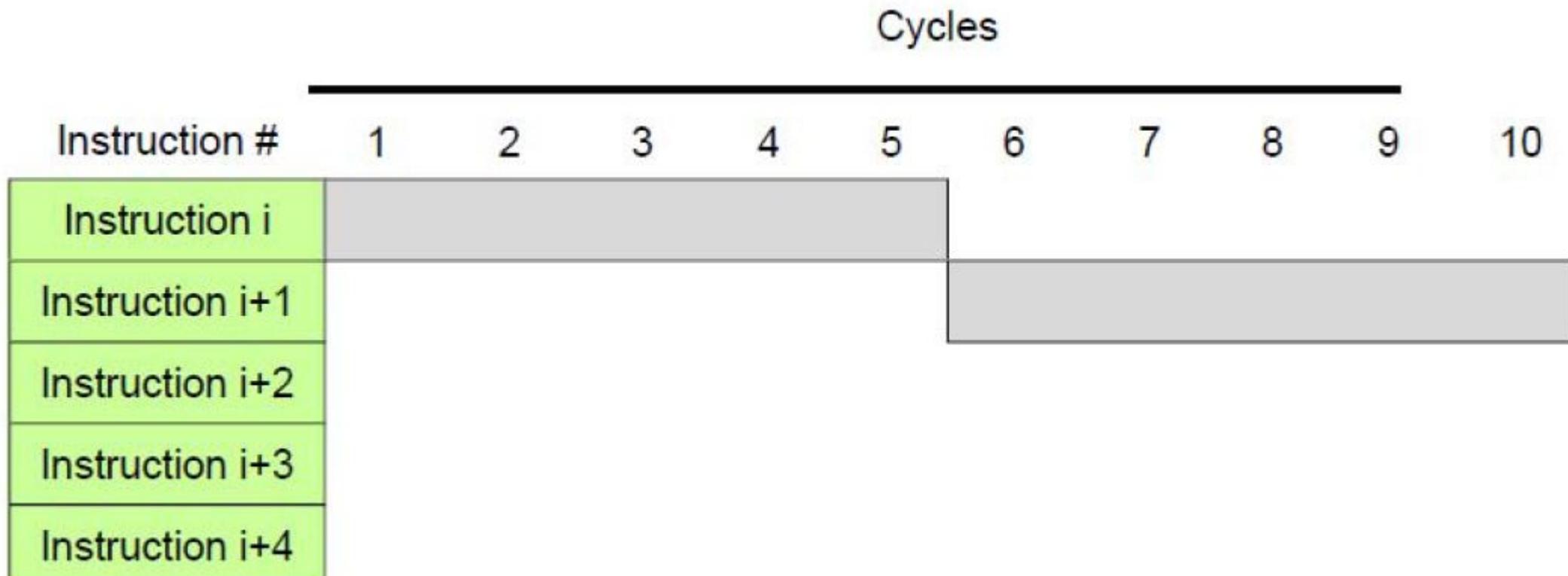
Examples of Instruction Level Parallelism

- **Pipelining**: 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). Example: While adding two numbers, the CPU can start preparing the next addition before finishing the first.
- **Superscalar**: The CPU can execute multiple different instructions at the same time. Example: It can add, multiply, and load data simultaneously.
- **Super-pipelining**: This is a mix of superscalar and pipelining—making pipelining even faster. The CPU breaks tasks into smaller steps, allowing multiple steps to run at the same time.
- **Vectorization**: The CPU performs the same operation on multiple data points at once instead of processing them one by one. Example: Adding two lists of numbers in one step instead of adding each number separately.

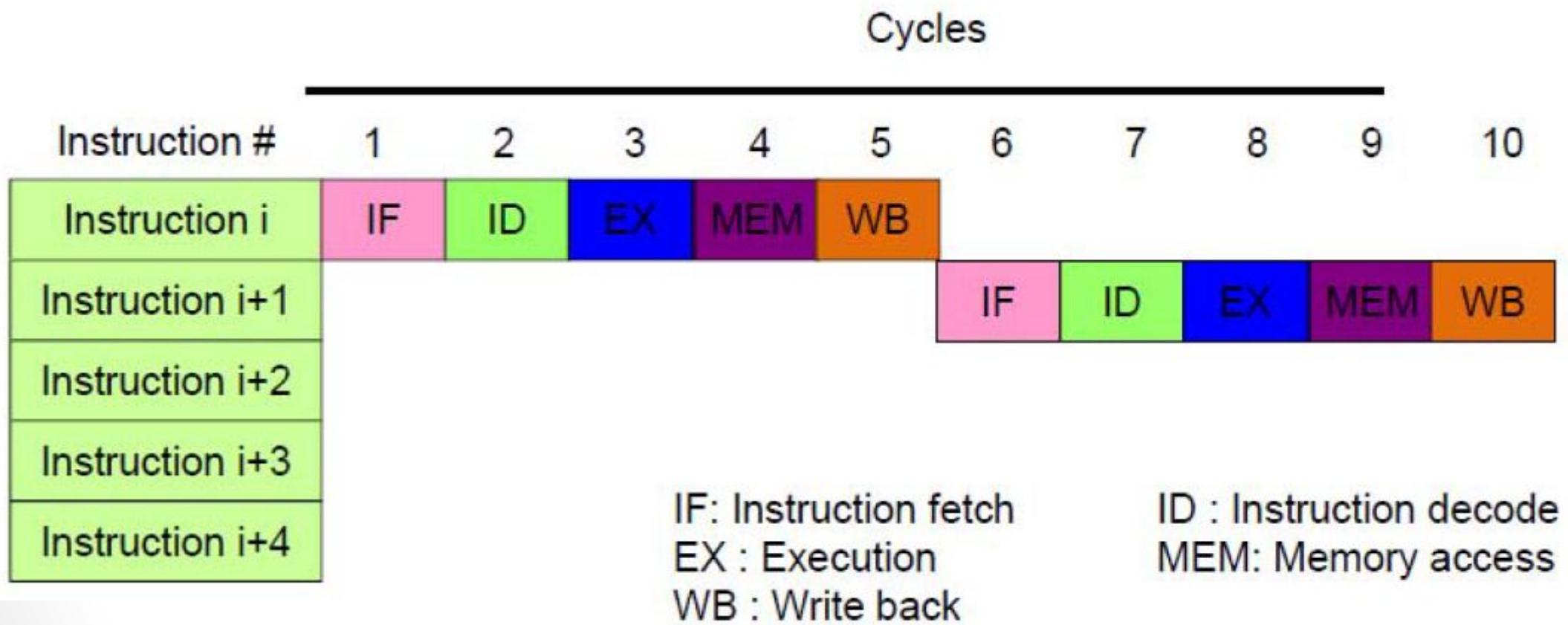
5 Stages in Executing a MIPS instruction

- **IF:** Instruction Fetch, Increment Program Counter
- **ID:** Instruction Decode, Read Registers
- **EX:** Execution
 - Mem-ref: Calculate Address
 - Arithmetic/logical: Perform Operation
- **MEM:**
 - Load: Read Data from Memory
 - Store: Write Data to Memory
- **WB:**
 - Write Data Back to Register

Instruction Execution



Instruction Execution



Instruction Execution

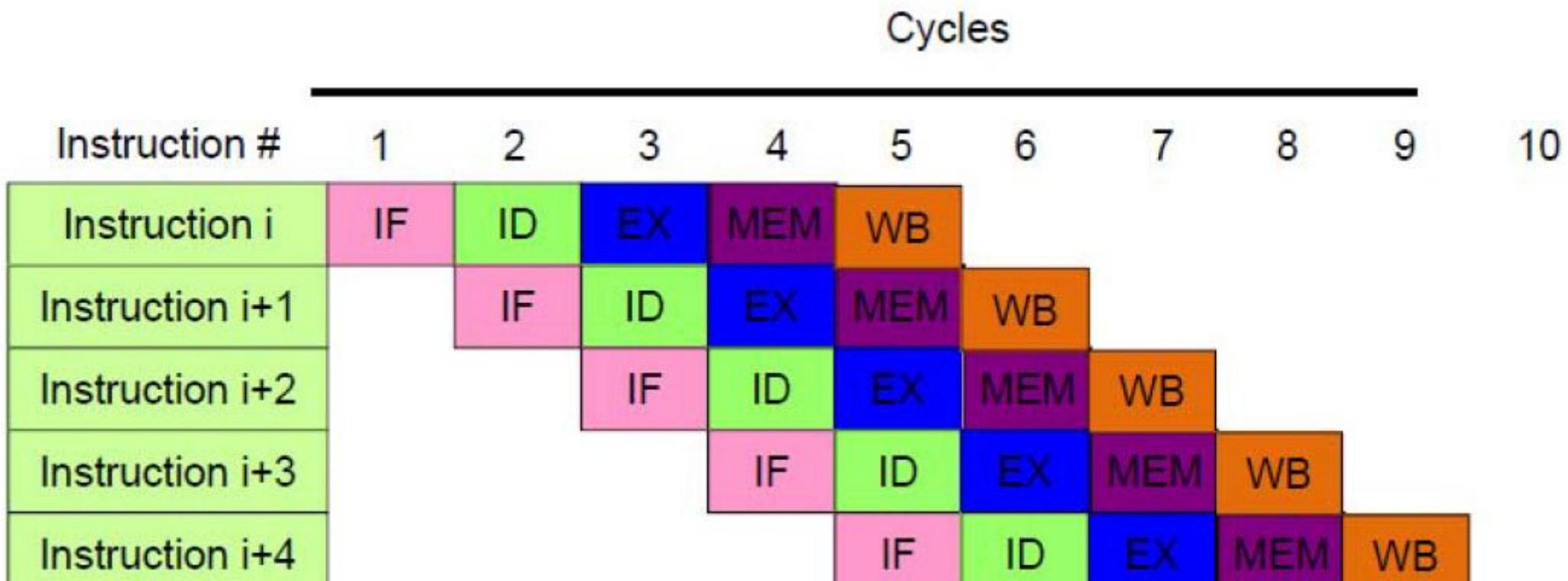
Each instruction starts only after the previous instruction completes.

- Instruction i starts in cycle 1 and completes in cycle 5.
- Instruction $i+1$ starts in cycle 6 and completes in cycle 10.
- Instruction $i+2$ starts in cycle 11, and so on.

There is no pipeline overlapping!

- This means that the next instruction does **not** start until the previous one is fully executed.
- This is different from a **pipelined CPU**, where new instructions start execution before the previous ones finish.

Pipelining Execution



IF: Instruction fetch
EX : Execution
WB : Write back

ID : Instruction decode
MEM: Memory access

Pipelining Execution

Overlapping Execution: Unlike a sequential processor (where each instruction must finish before the next begins), pipelining allows instructions to overlap.

Instruction i begins at cycle 1 and completes at cycle 5.

Instruction $i+1$ starts in cycle 2, enters each stage one cycle later than the previous instruction, and completes at cycle 6.

This pattern continues, filling the pipeline.

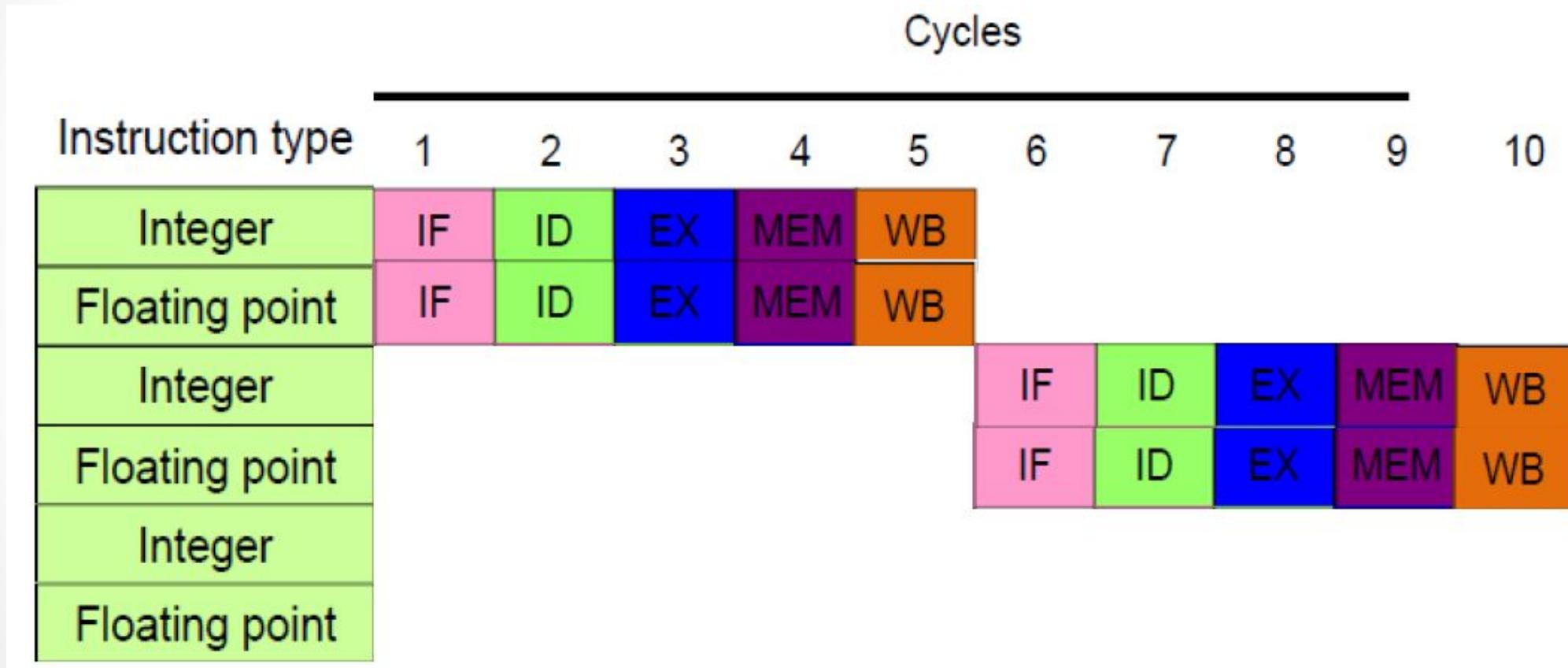
Once the pipeline is full, one instruction completes every cycle.

The previous image showed a pipeline stall or delay between some instructions, leading to inefficiencies.

This image shows a **fully utilized pipeline** with no stalls.

The result: **Higher throughput**, where multiple instructions are being processed at the same time.

Superscalar Execution



Superscalar Execution

Definition: A superscalar processor can fetch, decode, execute, and write back multiple instructions per cycle by using multiple execution units.

Difference from Simple Pipelining: While a pipelined processor processes one instruction per stage at a time, a superscalar processor executes multiple instructions in parallel, as long as they don't depend on each other.

Superscalar Execution

Multiple Execution Pipelines:

- The processor can execute **integer** and **floating-point** instructions **simultaneously**.
- This is possible because the CPU has **separate functional units** for integer and floating-point operations.

Parallel Instruction Execution:

- In cycle 1, an **integer** and a **floating-point** instruction start fetching (IF) **simultaneously**.
- Both instructions go through the pipeline together but use **different execution units**.
- This pattern repeats for subsequent instructions.

Improved Throughput:

- Unlike a standard pipelined execution, where only **one instruction completes per cycle**, superscalar execution allows **multiple instructions to complete per cycle**.
- This **significantly increases performance**.

Functional Units in a CPU

A typical CPU consists of multiple execution units, each specialized for a certain type of operation. These include:

Functional Unit	Purpose
Integer ALU (Arithmetic Logic Unit)	Performs integer arithmetic and logical operations
Floating-Point Unit (FPU)	Handles floating-point arithmetic (addition, multiplication, etc.)
Load/Store Unit	Manages memory operations (reading/writing data from RAM)
Branch Unit	Predicts and handles branch instructions (if-else, loops)
Vector Unit (SIMD)	Executes vector operations (used in multimedia, AI, etc.)

Super-pipelining: Superscalar + Pipeline

Instruction type	Cycles					
	1	2	3	4	5	6
Integer	IF	ID	EX	MEM	WB	
Floating point	IF	ID	EX	MEM	WB	
Integer		IF	ID	EX	MEM	WB
Floating point		IF	ID	EX	MEM	WB
Integer			IF	ID	EX	MEM
Floating point			IF	ID	EX	MEM
Integer				IF	ID	WB
Floating point				IF	ID	WB
Integer					IF	WB
Floating point					IF	WB

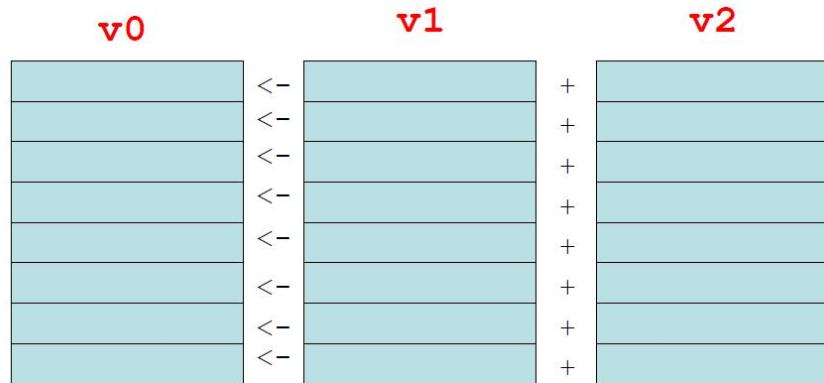
2-issue super-scalar machine

Super-pipelining: Superscalar + Pipeline

- Two instructions are issued per cycle (one integer, one floating-point).
- Each instruction follows a 5-stage pipeline, allowing multiple instructions to execute in parallel.
- Total execution time is reduced significantly by maximizing instruction throughput.

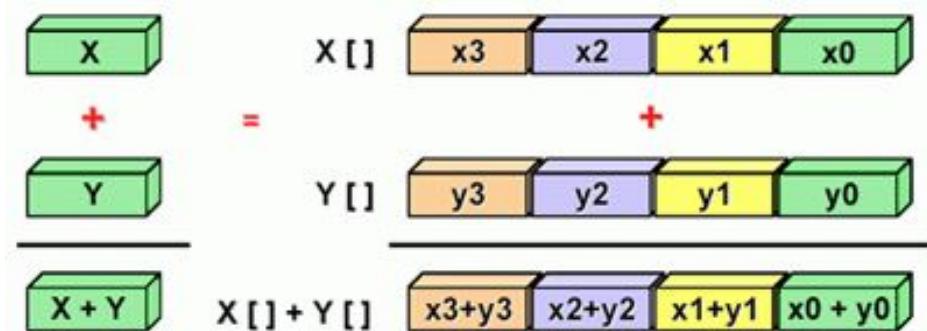
Vectorization

- A **vector register** is a register that's made up of many individual registers that simultaneously perform the same operation on multiple sets of operands, producing multiple results. In a sense, vectors are like operation-specific cache.
- A **vector instruction** is an instruction that performs the same operation simultaneously on all individual registers of a vector register.



v0 <- v1 + v2

```
for (i=0; i <=MAX; i++)  
    Z[i] = X[i] + Y[i];
```



What determines the degree of ILP?

- **Dependencies**: 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 Dependencies

- **Data Dependence:** When one instruction needs data from a previous instruction before it can execute. The second instruction must wait for the first to finish.
(True Dependence)
 - RAW: Read-After-Write

```
a = 5; // First instruction writes to 'a'  
b = a + 2; // Second instruction reads 'a' (Read-After-Write - RAW)
```

- **Named Dependencies:** **Write-After-Read (WAR or Anti-Dependance):** Happens when an instruction writes to a variable that a previous instruction is still reading. The write to 'y' must wait for the read to finish.

```
x = y + 3; // Read 'y'  
y = 10; // Write to 'y' (WAR issue)
```

Types of Dependencies

Write-After-Write (WAW or Output Dependence): Occurs when two instructions write to the same variable. The second write cannot execute before the first completes.

```
a = 5;  
a = 10; // This write must wait for the first one
```

- **Control Dependence:**

- When an instruction depends on a decision (branch/jump) from a previous instruction.
- The second instruction must wait to see if the first condition is true.

```
if (x > 0) { // Condition must be checked first  
    y = x + 2; // Only runs if the condition is true  
}
```

What is Dependency Analysis?

- It examines how different parts of a program depend on each other.
 - Ensures that parts of the program execute in the correct order.
 - Prevents incorrect results due to improper instruction execution.
 - A key role of modern compilers, which optimize code execution while respecting dependencies.
 - A **data dependency** describes how different pieces of data affect each other.
 - A **control dependency** describes how instruction sequences affect each other.
-
- **Why is this Important?**
 - Compilers analyze dependencies to reorder instructions for better performance.
 - Parallel execution (in multi-core systems) needs to handle dependencies correctly.
 - Avoids incorrect calculations due to premature execution of dependent instructions.

Data Dependence and Hazards

- If two instructions are data dependent, they cannot execute simultaneously, be completely overlapped or execute in out-of-order
- Instruction **S2** is data dependent (aka true dependence) on Instruction **S1**

S1: **x** = A + B
S2: C = **x** + 1



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

Loop Carried Dependency

```
FOR i = 2 to MAX  
  a[i] = a[i-1] + b[i]  
END FOR
```

- 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.
- 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.

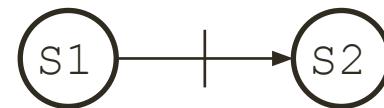
ILP and Data Dependencies, Hazards

- HW/SW must preserve **program order**: code must give the same results as if instructions were executed sequentially in original order of the source program
- Importance of the data dependencies:
 1. Indicates the possibility of a hazard
 2. Determines order in which results must be calculated
 3. Sets an upper bound on how much parallelism can possibly be exploited
- **Goal**: Exploit parallelism by preserving program order **only where it affects the outcome of the program**

Named Dependence and Hazards

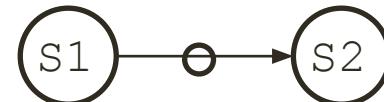
- **Name Dependence:** When two instructions use the same variable name (register or memory location), but there is no flow of data between them associated with that variable name.
- **Anti-Dependence:** Instruction **S2** writes operand before Instruction **S1** reads it

S1: $A = \textcolor{red}{X} + B$
S2: $\textcolor{red}{X} = C + D$



- **Output-Dependence:** Instruction **S2** writes operand before Instruction **S1** writes it

S1: $X = A + B$
...
S2: $X = C + D$



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

Control Dependencies

- Every program has a well-defined flow of control that moves from instruction to instruction. Every instruction is control dependent on some set of branches (if condition, switch case, function calls, I/O), and, in general, these control dependencies must be preserved to preserve program order.

```
IF p1 THEN  
    S1  
IF p2 THEN  
    S2
```

- S1 is control dependent on p1, and S2 is control dependent on p2 but not on p1.
- Control dependence need not be preserved
 - willing to execute instructions that should not have been executed, thereby violating the control dependences, if can do so without affecting correctness of the program
- Speculative Execution

Speculation

- Speculative parallelism helps computers run faster by guessing the outcome of decisions (branches) before knowing for sure. If the guess is right, the program runs faster. If it's wrong, the incorrect work is discarded, and the correct path is executed.
- **How it works?**
 - **Speculation** → The processor fetches, issues, and executes instructions as if its guesses about branches (like if-else conditions) were always correct.
 - **Dynamic Scheduling** → The system fetches and issues instructions but doesn't execute them until it's sure they're needed.
- Essentially a data flow execution model: Operations execute as soon as their operands are available

Speculation

- Types of predictors
 - Branch Prediction: Guessing which path (if/else) the program will take.
 - Value Prediction: Predicting future values based on past data
 - Prefetching (memory access pattern prediction): Predicting which memory locations will be needed soon and loading them early.
- Problems & Inefficiencies
 - If the guess is incorrect, extra work needs to be undone.
 - The system must clear out incorrect guesses, wasting time.
 - Making guesses and fixing wrong ones uses more energy.

Limits to Pipelining

- Pipelining increases CPU efficiency by overlapping instruction execution, but it has limitations due to hazards. Hazards prevent the next instruction from executing in its designated clock cycle.
- **Structural hazards:** Occur when hardware resources are insufficient, meaning two or more instructions require the same hardware at the same time.
- **Data hazards:** Occur when an instruction depends on the result of a previous instruction that hasn't finished executing.
- **Control hazards:** Occur when the CPU encounters a branch (conditional or jump) and doesn't know the correct next instruction to fetch.

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**.

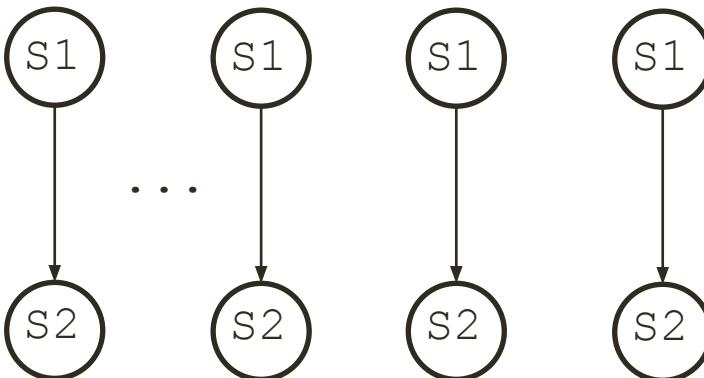
Dependencies in Loops

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

```
FOR i = 1 to n  
S1      a[i] = b[i] + 1  
S2      c[i] = a[i] + 2
```

Iteration: 1 2 3 4 ...

Instances of S1:



Instances of S2:

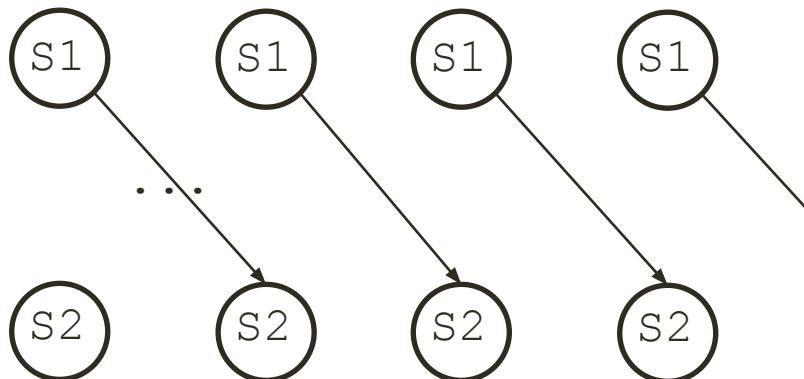
Dependencies in Loops

- A little more complex example

```
FOR i = 1 to n
S1      a[i] = b[i] + 1
S2      c[i] = a[i-1] + 2
```

Iteration: 1 2 3 4 ...

Instances of S1:



Instances of S2:

Dependencies in Loops

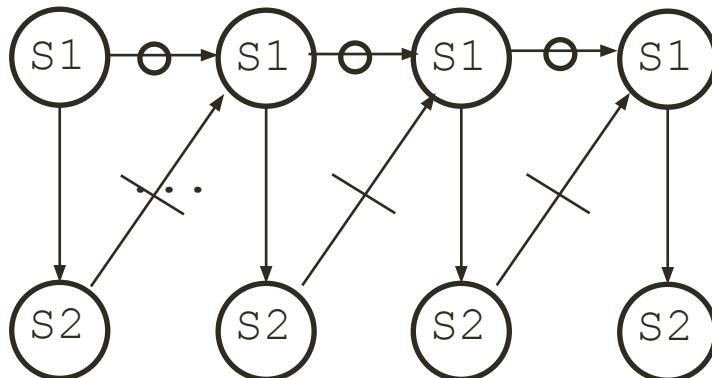
- Even more complex example

```
FOR i = 1 to n
S1      a = b[i] + 1
S2      c[i] = a + 2
```

Iteration:

1 2 3 4 ...

Instances of S1:



Instances of S2:

Branch Dependency

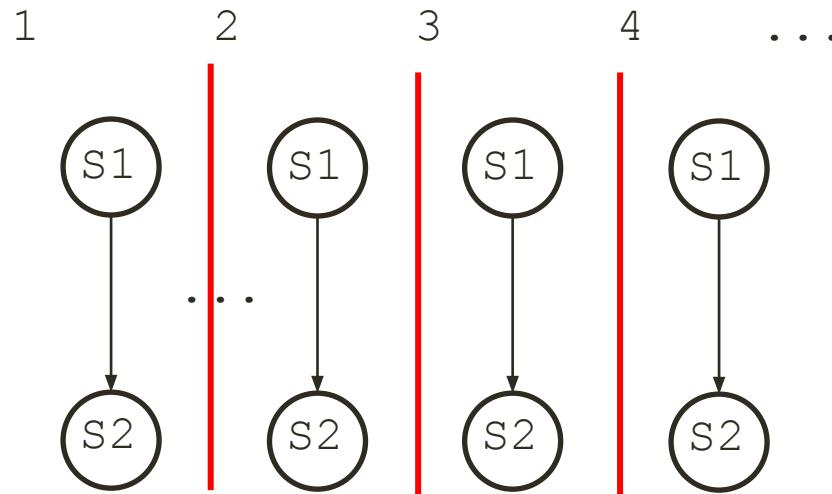
```
y = 7  
IF (x != 0) THEN  
    y = 1.0/x  
END IF
```

- The value of **y** depends on what the condition ($x \neq 0$) evaluates to:
 - If the condition evaluates to TRUE, then **y** is set to $1.0/x$
 - Otherwise, **y** remains 7

Optimizing with Dependencies

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

```
FOR i = 1 to n
S1      a[i] = b[i] + 1
S2      c[i] = a[i] + 2
```



Loop or Branch Dependency?

- Is this a **loop carried dependency** or a **branch dependency**?

```
FOR i = 1 to MAX
  IF (x[i] != 0)
    y[i] = 1.0/x[i]
  END IF
END FOR
```

Loop or Branch Dependency?

- Is this a **loop carried dependency** or a **branch dependency**?

```
FOR i = 1 to MAX
  IF (x[i] != 0)
    y[i] = 1.0/x[i]
  END IF
END FOR
```

Answer:

- The given code has a branch dependency due to the conditional (IF $x[i] \neq 0$).
- However, it does not have a loop-carried dependency since each iteration is independent.

Compiler-Driven Parallelism

- Tricks that compilers play for automatic parallelization
 - Scalar optimizations
 - Loop-level optimizations

Scalar Optimizations

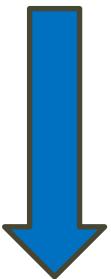
- Copy Propagation
- Constant Folding
- Dead Code Removal
- Strength Reduction
- Common Subexpression Elimination
- Variable Renaming
- Not every compiler does all these, so it sometimes can be worth doing these by hands

Copy Propagation

Before

$$\begin{aligned} \mathbf{x} &= \mathbf{y} \\ z &= 1 + \mathbf{x} \end{aligned}$$

Has data dependency



Compile

After

$$\begin{aligned} \mathbf{x} &= \mathbf{y} \\ z &= 1 + \mathbf{y} \end{aligned}$$

No data dependency

Constant Folding

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

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

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.

Common Subexpression Elimination

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

Before

```
x = y * z  
q = r + x * 2.0  
x = a + b
```

After

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

- The original code has **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
- Inlining
- Not every compiler does all these, so it sometimes can be worth doing these by hands

Hoisting Loop Invariant Code

- The code that does not change inside the loop is known as **loop invariant**. It doesn't need to be calculated over and over.

Before

```
FOR i = 1 to n  
a[i] = b[i] + c * d  
e = g(n)  
END FOR
```

After

```
temp = c * d  
FOR i = 1 to n  
a[i] = b[i] + temp  
END FOR  
e = g(n)
```

Unswitching

```
FOR i = 1 to n
  FOR j = 2 to 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 FOR
END FOR
```

The condition is
j-independent

Before

```
FOR i = 1 to n
  IF (t(i) > 0) THEN
    FOR j = 2 to n
      a[i][j] = a[i][j] * t(i) + b(j)
    END FOR
  ELSE
    FOR j = 2 to n
      a[i][j] = 0.0
    END FOR
  END IF
END FOR
```

So, it can migrate
outside the j loop

(55)

After

Iteration Peeling

```
FOR i = 1 to n
  IF ((i == 1) OR (i == n)) THEN
    x[i] = y[i]
  ELSE
    x[i] = y[I + 1] + y[i - 1]
  END IF
END FOR
```

Before

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

```
x(1) = y(1)
FOR i = 2 to n-1
  x[i] = y[I + 1] + y[i - 1]
END FOR
x(n) = y(n)
```

After

Index Set Splitting

```
FOR i = 1 to n  
  a[i] = b[i] + c[i]  
  IF ((i > 10) THEN  
    d[i] = a[i] + b[i - 10]  
  END IF  
END FOR
```

Before

```
FOR i = 1 to 10  
  a[i] = b[i] + c[i]  
END FOR  
  
FOR i = 11 to n  
  a[i] = b[i] + c[i]  
  d[i] = a[i] + b[i - 10]  
END FOR
```

After

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

Loop Interchange

Before

```
FOR i = 1 to nj  
  FOR j = 1 to ni  
    a[i][j] = b[i][j]  
  END FOR  
END FOR
```

After

```
FOR i = 1 to ni  
  FOR j = 1 to nj  
    a[i][j] = b[i][j]  
  END FOR  
END FOR
```

- The array elements $a[i][j]$ and $a[i][j+1]$ are near each other in memory, while $a[i+1][j]$ may be far, so it makes sense to make the i loop be the outer loop.
- This technique facilitates efficient exploitation of the phenomenon of **locality of reference**.

Unrolling:

```
FOR i = 1 to n  
  a[i] = a[i] + b[i]  
END FOR
```

Before

```
FOR 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 FOR
```

After

- 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

```
FOR i = 1 to n  
  a[i] = b[i] + 1  
END FOR
```

```
FOR i = 1 to n  
  c[i] = a[i] /2  
END FOR
```

```
FOR i = 1 to n  
  d[i] = 1 / c[i]  
END FOR
```

```
FOR i = 1, n  
  a[i] = b[i] + 1  
  c[i] = a[i] / 2  
  d[i] = 1 / c[i]  
END FOR
```

Before

After

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

Loop Fission

```
FOR i = 1, n  
  a[i] = b[i] + 1  
  c[i] = a[i] / 2  
  d[i] = 1 / c[i]  
END FOR
```

Before

```
FOR i = 1 to n  
  a[i] = b[i] + 1  
END FOR
```

```
FOR i = 1 to n  
  c[i] = a[i] /2  
END FOR
```

After

```
FOR i = 1 to n  
  d[i] = 1 / c[i]  
END FOR
```

- 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

Before

```
FOR i = 1 to n  
a[i] = func(i)  
END FOR
```

```
int func(x)  
func = x * 3  
RETURN func
```

After

```
FOR i = 1 to n  
a[i] = i * 3  
END FOR
```

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

Task of the Compiler

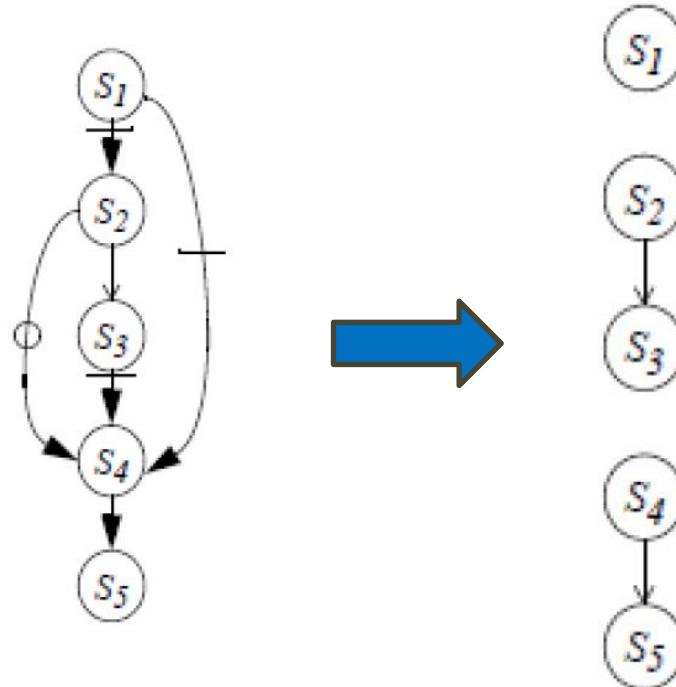
- It transforms program to remove dependencies
- Use dependence to reorganize computation for parallelism and locality.
- However, manual transformation by hand can sometimes be more useful.

Transforming Programs: Renaming

```
S1: A = X + B  
S2: X = Y + 1  
S3: C = X + B  
S4: X = Z + B  
S5: D = X + 1
```



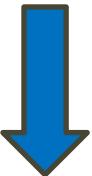
```
S1: A = X + B  
S2: X1 = Y + 1  
S3: C = X1 + B  
S4: X2 = Z + B  
S5: D = X2 + 1
```



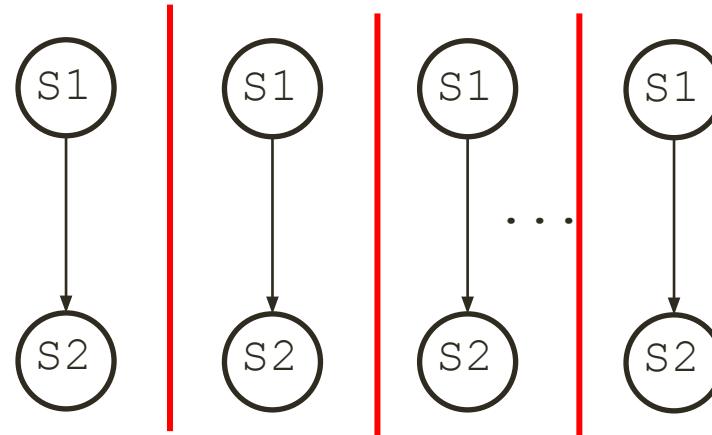
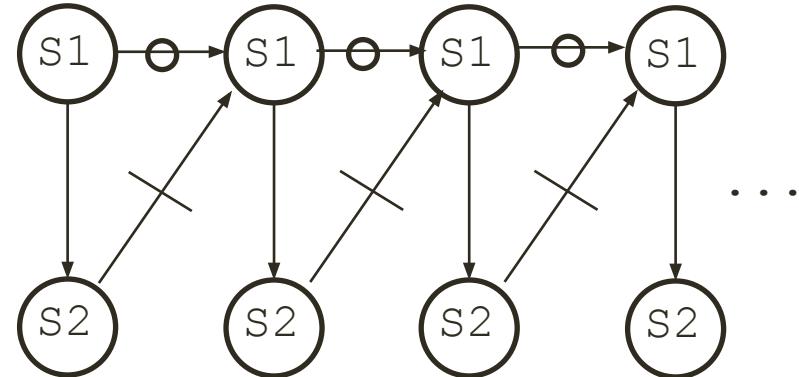
Transform Programs: Scalar Expansion

Scalar expansion is an optimization technique that transforms a loop containing a **scalar dependency** into an equivalent loop using an **array version of the scalar variable**. This eliminates dependencies and improves parallelization.

```
FOR i = 1 to n  
S1      a = b[i] + 1  
S2      c[i] = a + d[i]  
END FOR
```



```
FOR i = 1 to n  
S1      a1[i] = b[i] + 1  
S2      c[i] = a1[i] + d[i]  
END FOR  
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.

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.