Workshop on Essential Abstractions in GCC

Introduction to Parallelization and Vectorization

GCC Resource Center
(www.cse.iitb.ac.in/grc)

Department of Computer Science and Engineering,
Indian Institute of Technology, Bombay

3 July 2011
Outline

- Transformation for parallel and vector execution
- Data dependence
The Scope of this Tutorial

• What this tutorial does not address
  ▶ Algorithms used for parallelization and vectorization
  ▶ Code or data structures of the parallelization and vectorization pass of GCC
  ▶ Machine level issues related to parallelization and vectorization

• What this tutorial addresses

Basics of Discovering Parallelism using GCC
Part 1

Transformations for Parallel and Vector Execution
A Taxonomy of Parallel Computation

<table>
<thead>
<tr>
<th></th>
<th>Single Program</th>
<th>Multiple Programs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single Data</td>
<td>SPSD</td>
<td>MPSD</td>
</tr>
<tr>
<td>Multiple Data</td>
<td>SPMD</td>
<td>MPMD</td>
</tr>
</tbody>
</table>

Redundant computation for validation of intermediate steps

Essential Abstractions in GCC
GCC Resource Center, IIT Bombay
# A Taxonomy of Parallel Computation

<table>
<thead>
<tr>
<th></th>
<th>Single Program</th>
<th>Multiple Programs</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Single Instruction</td>
<td>Multiple Instructions</td>
</tr>
<tr>
<td>Single Data</td>
<td>SISD</td>
<td>MISD</td>
</tr>
<tr>
<td>Multiple Data</td>
<td>SIMD</td>
<td>MIMD</td>
</tr>
</tbody>
</table>
## A Taxonomy of Parallel Computation

<table>
<thead>
<tr>
<th></th>
<th>Single Program</th>
<th>Multiple Programs</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Single Instruction</strong></td>
<td>SISD</td>
<td>?</td>
</tr>
<tr>
<td><strong>Multiple Instructions</strong></td>
<td></td>
<td>?</td>
</tr>
<tr>
<td><strong>Single Data</strong></td>
<td>SISD</td>
<td>MIMD</td>
</tr>
<tr>
<td><strong>Multiple Data</strong></td>
<td>SIMD</td>
<td>MPMD</td>
</tr>
</tbody>
</table>

Redundant computation for validation of intermediate steps
## A Taxonomy of Parallel Computation

<table>
<thead>
<tr>
<th></th>
<th>Single Program</th>
<th>Multiple Programs</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Single Instruction</td>
<td>Multiple Instructions</td>
</tr>
<tr>
<td>Single Data</td>
<td>SISD</td>
<td>MISD</td>
</tr>
<tr>
<td>Multiple Data</td>
<td>SIMD</td>
<td>MIMD</td>
</tr>
</tbody>
</table>

Transformations performed by a compiler

---

Essential Abstractions in GCC

GCC Resource Center, IIT Bombay
Vectorization: SISD $\Rightarrow$ SIMD

- Parallelism in executing operation on shorter operands (8-bit, 16-bit, 32-bit operands)
- Existing 32 or 64-bit arithmetic units used to perform multiple operations in parallel
  A 64 bit word $\equiv$ a vector of $2 \times (32 \text{ bits})$, $4 \times (16 \text{ bits})$, or $8 \times (8 \text{ bits})$
Example 1

Vectorization (SISD ⇒ SIMD) : Yes
Parallelization (SISD ⇒ MIMD) : Yes

Original Code

```c
int A[N], B[N], i;
for (i=1; i<N; i++)
    A[i] = A[i] + B[i-1];
```
Example 1

Vectorization (SISD $ \Rightarrow $ SIMD) : Yes
Parallelization (SISD $ \Rightarrow $ MIMD) : Yes

Original Code

```c
int A[N], B[N], i;
for (i=1; i<N; i++)
    A[i] = A[i] + B[i-1];
```

Observe reads and writes into a given location

A[0..N]  

B[0..N]  

...
Example 1

Vectorization \text{(SISD} ⇒ \text{SIMD)} : Yes
Parallelization \text{(SISD} ⇒ \text{MIMD)} : Yes

Original Code

```c
int A[N], B[N], i;
for (i=1; i<N; i++)
    A[i] = A[i] + B[i-1];
```

Observe reads and writes into a given location

A[0..N]  B[0..N]
**Example 1**

**Vectorization** (SISD $\Rightarrow$ SIMD) : Yes  
**Parallelization** (SISD $\Rightarrow$ MIMD) : Yes

**Original Code**

```c
int A[N], B[N], i;
for (i=1; i<N; i++)
    A[i] = A[i] + B[i-1];
```

Observe reads and writes into a given location

- **A[0..N]**  
- **B[0..N]**

Iteration # 1
**Example 1**

Vectorization \( (\text{SISD} \Rightarrow \text{SIMD}) \) : Yes  
Parallelization \( (\text{SISD} \Rightarrow \text{MIMD}) \) : Yes

**Original Code**

```c
int A[N], B[N], i;
for (i=1; i<N; i++)
    A[i] = A[i] + B[i-1];
```

Observe reads and writes into a given location...

A[0..N] B[0..N] Iteration # 1 2
**Example 1**

Vectorization (SISD $\Rightarrow$ SIMD) : Yes  
Parallelization (SISD $\Rightarrow$ MIMD) : Yes  

Original Code

```c
int A[N], B[N], i;
for (i=1; i<N; i++)
    A[i] = A[i] + B[i-1];
```

Observe reads and writes into a given location...

<table>
<thead>
<tr>
<th>Iteration #</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>A[0..N]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B[0..N]</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Essential Abstractions in GCC  
GCC Resource Center, IIT Bombay
Example 1

Vectorization (SISD ⇒ SIMD): Yes
Parallelization (SISD ⇒ MIMD): Yes

Original Code

```c
int A[N], B[N], i;
for (i=1; i<N; i++)
  A[i] = A[i] + B[i-1];
```

Observe reads and writes into a given location

A[0..N]

B[0..N]

Iteration # 1 2 3 4

...
Example 1

Vectorization (SISD ⇒ SIMD) : Yes
Parallelization (SISD ⇒ MIMD) : Yes

Original Code

```c
int A[N], B[N], i;
for (i=1; i<N; i++)
    A[i] = A[i] + B[i-1];
```

Observe reads and writes into a given location

![Diagram showing iterations of the code](image-url)
Example 1

Vectorization (SISD $\Rightarrow$ SIMD) : Yes
Parallelization (SISD $\Rightarrow$ MIMD) : Yes

Original Code

```c
int A[N], B[N], i;
for (i=1; i<N; i++)
    A[i] = A[i] + B[i-1];
```

Observe reads and writes into a given location

```
A[0..N]  B[0..N]
```

Iteration #  1  2  3  4  5  6
Example 1

Vectorization \( (\text{SISD} \Rightarrow \text{SIMD}) \) : Yes
Parallelization \( (\text{SISD} \Rightarrow \text{MIMD}) \) : Yes

Original Code

```c
int A[N], B[N], i;
for (i=1; i<N; i++)
    A[i] = A[i] + B[i-1];
```

Observe reads and writes into a given location

```
A[0..N]
B[0..N]
Iteration # 1 2 3 4 5 6 7
```

```c
```

Example 1

Vectorization (SISD $\Rightarrow$ SIMD) : Yes
Parallelization (SISD $\Rightarrow$ MIMD) : Yes

Original Code

```c
int A[N], B[N], i;
for (i=1; i<N; i++)
    A[i] = A[i] + B[i-1];
```

Observe reads and writes into a given location...

- $A[0..N]$...
- $B[0..N]$...
- Iteration # 1 2 3 4 5 6 7 8
Example 1

Vectorization (SISD ⇒ SIMD) : Yes
Parallelization (SISD ⇒ MIMD) : Yes

Original Code

```c
int A[N], B[N], i;
for (i=1; i<N; i++)
    A[i] = A[i] + B[i-1];
```

Observe reads and writes into a given location

```
A[0..N]  B[0..N]

Iteration #  1  2  3  4  5  6  7  8  9
```

...
Example 1

Vectorization (SISD ⇒ SIMD) : Yes
Parallelization (SISD ⇒ MIMD) : Yes

Original Code

```c
int A[N], B[N], i;
for (i=1; i<N; i++)
    A[i] = A[i] + B[i-1];
```

Observe reads and writes into a given location

```
A[0..N]
B[0..N]
Iteration #  1  2  3  4  5  6  7  8  9  10
```
Example 1

Vectorization (SISD ⇒ SIMD) : Yes
Parallelization (SISD ⇒ MIMD) : Yes

Original Code

```c
int A[N], B[N], i;
for (i=1; i<N; i++)
    A[i] = A[i] + B[i-1];
```

Observe reads and writes into a given location
Example 1

Vectorization (SISD $\Rightarrow$ SIMD) : Yes
Parallelization (SISD $\Rightarrow$ MIMD) : Yes

Original Code

```c
int A[N], B[N], i;
for (i=1; i<N; i++)
    A[i] = A[i] + B[i-1];
```

Observe reads and writes into a given location

A[0..N]  
B[0..N]  
Iteration # 1 2 3 4 5 6 7 8 9 10 11 12  ...
Example 1

Vectorization (SISD $\Rightarrow$ SIMD) : Yes
Parallelization (SISD $\Rightarrow$ MIMD) : Yes

Original Code

```c
int A[N], B[N], i;
for (i=1; i<N; i++)
    A[i] = A[i] + B[i-1];
```

Vectorized Code

```c
int A[N], B[N], i;
for (i=1; i<N; i=i+4)
```

```
A[0..N]  B[0..N]
```

Iteration #
Example 1

Vectorization (SISD ⇒ SIMD) : Yes
Parallelization (SISD ⇒ MIMD) : Yes

Original Code

```c
int A[N], B[N], i;
for (i=1; i<N; i++)
    A[i] = A[i] + B[i-1];
```

Vectorized Code

```c
int A[N], B[N], i;
for (i=1; i<N; i=i+4)
```

### Vectorization Factor

<table>
<thead>
<tr>
<th>Iteration #</th>
<th>A[0..N]</th>
<th>B[0..N]</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>![Vectorized A]</td>
<td>![Vectorized B]</td>
</tr>
</tbody>
</table>
Example 1

Vectorization \((\text{SISD} \Rightarrow \text{SIMD})\) : Yes
Parallelization \((\text{SISD} \Rightarrow \text{MIMD})\) : Yes

Original Code

```c
int A[N], B[N], i;
for (i=1; i<N; i++)
    A[i] = A[i] + B[i-1];
```

Vectorized Code

```c
int A[N], B[N], i;
for (i=1; i<N; i=i+4)
```

Vectorization Factor

```
A[0..N]  B[0..N]  
Iteration #  1  2  
```
Example 1

Vectorization (SISD $\Rightarrow$ SIMD) : Yes
Parallelization (SISD $\Rightarrow$ MIMD) : Yes

Original Code

```
int A[N], B[N], i;
for (i=1; i<N; i++)
    A[i] = A[i] + B[i-1];
```

Vectorized Code

```
int A[N], B[N], i;
for (i=1; i<N; i=i+4)
```

A[0..N]  

B[0..N]  

Iteration #  

Vectorization Factor

Essential Abstractions in GCC

GCC Resource Center, IIT Bombay
Example 1

Vectorization (SISD ⇒ SIMD) : Yes
Parallelization (SISD ⇒ MIMD) : Yes

Original Code

```c
int A[N], B[N], i;
for (i=1; i<N; i++)
    A[i] = A[i] + B[i-1];
```

Observe reads and writes into a given location
Example 1

Vectorization (SISD $\Rightarrow$ SIMD) : Yes
Parallelization (SISD $\Rightarrow$ MIMD) : Yes

Original Code

int A[N], B[N], i;
for (i=1; i<N; i++)
    A[i] = A[i] + B[i-1];

Observe reads and writes into a given location

A[0..N]  

B[0..N]  

...
Example 1

Vectorization (SISD $\Rightarrow$ SIMD) : Yes
Parallelization (SISD $\Rightarrow$ MIMD) : Yes

Original Code

```c
int A[N], B[N], i;
for (i=1; i<N; i++)
    A[i] = A[i] + B[i-1];
```

Observe reads and writes into a given location...
Example 1

Vectorization (SISD ⇒ SIMD) : Yes
Parallelization (SISD ⇒ MIMD) : Yes

Original Code

```
int A[N], B[N], i;
for (i=1; i<N; i++)
    A[i] = A[i] + B[i-1];
```

Parallelized Code

```
int A[N], B[N], i;
foreach (i=1; i<N; )
    A[i] = A[i] + B[i-1];
```
Example 1: The Moral of the Story

Vectorization (SISD ⇒ SIMD) : Yes
Parallelization (SISD ⇒ MIMD) : Yes

```c
int A[N], B[N], i;
for (i=1; i<N; i++)
    A[i] = A[i] + B[i-1];
```

Observe reads and writes into a given location
Example 1: The Moral of the Story

Vectorization (SISD ⇒ SIMD) : Yes
Parallelization (SISD ⇒ MIMD) : Yes

When the same location is accessed across different iterations, the order of reads and writes must be preserved.

<table>
<thead>
<tr>
<th>Iteration $i$</th>
<th>Iteration $i + k$</th>
<th>Observation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read</td>
<td>Write</td>
<td></td>
</tr>
<tr>
<td>Write</td>
<td>Read</td>
<td></td>
</tr>
<tr>
<td>Write</td>
<td>Write</td>
<td></td>
</tr>
<tr>
<td>Read</td>
<td>Read</td>
<td></td>
</tr>
</tbody>
</table>

Essential Abstractions in GCC
GCC Resource Center, IIT Bombay
Example 1: The Moral of the Story

Vectorization (SISD ⇒ SIMD) : Yes
Parallelization (SISD ⇒ MIMD) : Yes

When the same location is accessed across different iterations, the order of reads and writes must be preserved.

<table>
<thead>
<tr>
<th>Nature of accesses in our example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Iteration $i$</td>
</tr>
<tr>
<td>Read</td>
</tr>
<tr>
<td>Write</td>
</tr>
<tr>
<td>Write</td>
</tr>
<tr>
<td>Read</td>
</tr>
</tbody>
</table>

```
int A[N], B[N], i;
for (i=1; i<N; i++)
    A[i] = A[i] + B[i-1];
```

```
A[0..N]  . . .
B[0..N]  . . .
```
Example 1: The Moral of the Story

Vectorization (SISD ⇒ SIMD) : Yes
Parallelization (SISD ⇒ MIMD) : Yes

When the same location is accessed across different iterations, the order of reads and writes must be preserved.

<table>
<thead>
<tr>
<th>Nature of accesses in our example</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Iteration</strong> $i$</td>
</tr>
<tr>
<td>Read</td>
</tr>
<tr>
<td>Write</td>
</tr>
<tr>
<td>Write</td>
</tr>
<tr>
<td>Read</td>
</tr>
</tbody>
</table>

```
int A[N], B[N], i;
for (i=1; i<N; i++)
    A[i] = A[i] + B[i-1];
```
Example 1: The Moral of the Story

Vectorization (SISD ⇒ SIMD) : Yes
Parallelization (SISD ⇒ MIMD) : Yes

When the same location is accessed across different iterations, the order of reads and writes must be preserved.

<table>
<thead>
<tr>
<th>Nature of accesses in our example</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Iteration</strong> $i$</td>
</tr>
<tr>
<td>Read</td>
</tr>
<tr>
<td>Write</td>
</tr>
<tr>
<td>Write</td>
</tr>
<tr>
<td>Read</td>
</tr>
</tbody>
</table>

```c
int A[N], B[N], i;
for (i=1; i<N; i++)
    A[i] = A[i] + B[i-1];
```
Example 1: The Moral of the Story

Vectorization (SISD ⇒ SIMD) : Yes
Parallelization (SISD ⇒ MIMD) : Yes

When the same location is accessed across different iterations, the order of reads and writes must be preserved.

<table>
<thead>
<tr>
<th>Nature of accesses in our example</th>
<th>Iteration $i$</th>
<th>Iteration $i + k$</th>
<th>Observation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read</td>
<td>Write</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>Write</td>
<td>Read</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>Write</td>
<td>Write</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>Read</td>
<td>Read</td>
<td>Does not matter</td>
<td></td>
</tr>
</tbody>
</table>

int A[N], B[N], i;
for (i=1; i<N; i++)
        A[i] = A[i] + B[i-1];
Example 2

Vectorization (SISD $\Rightarrow$ SIMD) : Yes
Parallelization (SISD $\Rightarrow$ MIMD) : No

Original Code

```c
int A[N], B[N], i;
for (i=0; i<N; i++)
```
Example 2

Vectorization (SISD ⇒ SIMD) : Yes
Parallelization (SISD ⇒ MIMD) : No

Original Code

```c
int A[N], B[N], i;
for (i=0; i<N; i++)
```

Observe reads and writes into a given location

A[0..N]  
B[0..N]  

...  
...
Example 2

Vectorization (SISD $\Rightarrow$ SIMD) : Yes
Parallelization (SISD $\Rightarrow$ MIMD) : No

Original Code

```c
int A[N], B[N], i;
for (i=0; i<N; i++)
```

Observe reads and writes into a given location

```
A[0..N]
B[0..N]
Iteration #
```
Example 2

Vectorization (SISD $\Rightarrow$ SIMD) : Yes
Parallelization (SISD $\Rightarrow$ MIMD) : No

Original Code

```c
int A[N], B[N], i;
for (i=0; i<N; i++)
```

Observe reads and writes into a given location

```
A[0..N]
B[0..N]
Iteration # 1
```
Example 2

Vectorization (SISD $\Rightarrow$ SIMD) : Yes
Parallelization (SISD $\Rightarrow$ MIMD) : No

Original Code

```c
int A[N], B[N], i;
for (i=0; i<N; i++)
```

Observe reads and writes into a given location

```
A[0..N] B[0..N]
```

Iteration # 1 2
Example 2

Vectorization (SISD ⇒ SIMD) : Yes
Parallelization (SISD ⇒ MIMD) : No

Original Code

```c
int A[N], B[N], i;
for (i=0; i<N; i++)
```

Observe reads and writes into a given location

A[0..N]  
B[0..N]  

Iteration #  1  2  3  

...  
...
Example 2

Vectorization (SISD $\Rightarrow$ SIMD) : Yes
Parallelization (SISD $\Rightarrow$ MIMD) : No

Original Code

```c
int A[N], B[N], i;
for (i=0; i<N; i++)
```

Observe reads and writes into a given location.
Example 2

Vectorization (SISD $\Rightarrow$ SIMD) : Yes
Parallelization (SISD $\Rightarrow$ MIMD) : No

Original Code

```c
int A[N], B[N], i;
for (i=0; i<N; i++)
```

Observe reads and writes into a given location

```
A[0..N]  B[0..N]  
Iteration # 1 2 3 4 5
...     ...   ...
Example 2

Vectorization (SISD $\Rightarrow$ SIMD) : Yes
Parallelization (SISD $\Rightarrow$ MIMD) : No

Original Code

```c
int A[N], B[N], i;
for (i=0; i<N; i++)
```

Observe reads and writes into a given location

![Diagram of vectorization process]
Example 2

Vectorization (SISD $\Rightarrow$ SIMD) : Yes
Parallelization (SISD $\Rightarrow$ MIMD) : No

Original Code

```c
int A[N], B[N], i;
for (i=0; i<N; i++)
```

Observe reads and writes into a given location

A[0..N]  
B[0..N]

Iteration # 1 2 3 4 5 6 7
Example 2

Vectorization (SISD $\Rightarrow$ SIMD) : Yes
Parallelization (SISD $\Rightarrow$ MIMD) : No

Original Code

```c
int A[N], B[N], i;
for (i=0; i<N; i++)
```

Observe reads and writes into a given location...
Example 2

Vectorization (SISD ⇒ SIMD) : Yes
Parallelization (SISD ⇒ MIMD) : No

Original Code

```c
int A[N], B[N], i;
for (i=0; i<N; i++)
```

Observe reads and writes into a given location

A[0..N]  B[0..N]

Iteration # 1 2 3 4 5 6 7 8 9
**Example 2**

Vectorization (SISD $\Rightarrow$ SIMD) : Yes  
Parallelization (SISD $\Rightarrow$ MIMD) : No

Original Code

```c
int A[N], B[N], i;
for (i=0; i<N; i++)
```

Observe reads and writes into a given location

```
A[0..N]  B[0..N]
```

Iteration # 1 2 3 4 5 6 7 8 9 10
Example 2

Vectorization (SISD $\Rightarrow$ SIMD) : Yes
Parallelization (SISD $\Rightarrow$ MIMD) : No

Original Code

```c
int A[N], B[N], i;
for (i=0; i<N; i++)
```

Observe reads and writes into a given location

```
A[0..N]
B[0..N]
```

Iteration #  1  2  3  4  5  6  7  8  9  10  11
Example 2

Vectorization (SISD ⇒ SIMD) : Yes
Parallelization (SISD ⇒ MIMD) : No

Original Code

```c
int A[N], B[N], i;
for (i=0; i<N; i++)
```

Observe reads and writes into a given location

- A[0..N]
- B[0..N]
- Iteration #: 1 2 3 4 5 6 7 8 9 10 11 12 . . .
Example 2

Vectorization (SISD $\Rightarrow$ SIMD) : Yes
Parallelization (SISD $\Rightarrow$ MIMD) : No

Original Code

```
int A[N], B[N], i;
for (i=0; i<N; i++)
```

- Vector instruction is synchronized: All reads before writes in a given instruction

Diagram:

- $A[0..N]$
- $B[0..N]$
- Iteration #
Example 2

Vectorization (SISD ⇒ SIMD) : Yes
Parallelization (SISD ⇒ MIMD) : No

Original Code

```c
int A[N], B[N], i;
for (i=0; i<N; i++)
```

- Vector instruction is synchronized: All reads before writes in a given instruction
Example 2

Vectorization (SISD ⇒ SIMD) : Yes
Parallelization (SISD ⇒ MIMD) : No

Original Code

```c
int A[N], B[N], i;
for (i=0; i<N; i++)
```

- Vector instruction is synchronized: All reads before writes in a given instruction

---

Essential Abstractions in GCC
GCC Resource Center, IIT Bombay
Example 2

Vectorization (SISD $\Rightarrow$ SIMD) : Yes
Parallelization (SISD $\Rightarrow$ MIMD) : No

Original Code

```c
int A[N], B[N], i;
for (i=0; i<N; i++)
```

- Vector instruction is synchronized: All reads before writes in a given instruction.
Example 2

Vectorization (SISD ⇒ SIMD) : Yes
Parallelization (SISD ⇒ MIMD) : No

Original Code

```c
int A[N], B[N], i;
for (i=0; i<N; i++)
```

- Vector instruction is synchronized: All reads before writes in a given instruction
- Read-writes across multiple instructions executing in parallel may not be synchronized
Example 2: The Moral of the Story

Vectorization (SISD ⇒ SIMD) : Yes
Parallelization (SISD ⇒ MIMD) : No

Original Code

```c
int A[N], B[N], i;
for (i=0; i<N; i++)
```

Observe reads and writes into a given location

```
A[0..N]
B[0..N]
...  ...
```
Example 2: The Moral of the Story

Vectorization (SISD ⇒ SIMD) : Yes
Parallelization (SISD ⇒ MIMD) : No

When the same location is accessed across different iterations, the order of reads and writes must be preserved.

<table>
<thead>
<tr>
<th>Nature of accesses in our example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Iteration $i$</td>
</tr>
<tr>
<td>Read</td>
</tr>
<tr>
<td>Write</td>
</tr>
<tr>
<td>Write</td>
</tr>
<tr>
<td>Read</td>
</tr>
</tbody>
</table>

int A[N], B[N], i;
for (i = 0; i < N; i++)
Example 2: The Moral of the Story

Vectorization (SISD $\Rightarrow$ SIMD) : Yes
Parallelization (SISD $\Rightarrow$ MIMD) : No

When the same location is accessed across different iterations, the order of reads and writes must be preserved.

Nature of accesses in our example

<table>
<thead>
<tr>
<th>Iteration $i$</th>
<th>Iteration $i + k$</th>
<th>Observation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read</td>
<td>Write</td>
<td>Yes</td>
</tr>
<tr>
<td>Write</td>
<td>Read</td>
<td></td>
</tr>
<tr>
<td>Write</td>
<td>Write</td>
<td></td>
</tr>
<tr>
<td>Read</td>
<td>Read</td>
<td></td>
</tr>
</tbody>
</table>

Essential Abstractions in GCC

GCC Resource Center, IIT Bombay
Example 2: The Moral of the Story

Vectorization (SISD \Rightarrow SIMD) : Yes
Parallelization (SISD \Rightarrow MIMD) : No

When the same location is accessed across different iterations, the order of reads and writes must be preserved.

| Nature of accesses in our example |
|-----------------|-----------------|----------------|
| Iteration $i$   | Iteration $i + k$ | Observation   |
| Read            | Write           | Yes           |
| Write           | Read            | No            |
| Write           | Write           |               |
| Read            | Read            |               |

Original Code

```c
int A[N], B[N], i;
for (i=0; i<N; i++)
```

```
A[0..N]
```

```
B[0..N]
```
Example 2: The Moral of the Story

Vectorization (SISD ⇒ SIMD) : Yes
Parallelization (SISD ⇒ MIMD) : No

When the same location is accessed across different iterations, the order of reads and writes must be preserved.

<table>
<thead>
<tr>
<th>Nature of accesses in our example</th>
<th>Iteration $i$</th>
<th>Iteration $i + k$</th>
<th>Observation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read</td>
<td>Write</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>Write</td>
<td>Read</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>Write</td>
<td>Write</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>Read</td>
<td>Read</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Original Code

```c
int A[N], B[N], i;
for (i=0; i<N; i++)
```
Example 2: The Moral of the Story

Vectorization (SISD ⇒ SIMD) : Yes
Parallelization (SISD ⇒ MIMD) : No

When the same location is accessed across different iterations, the order of reads and writes must be preserved.

<table>
<thead>
<tr>
<th>Nature of accesses in our example</th>
<th>Iteration $i$</th>
<th>Iteration $i + k$</th>
<th>Observation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read</td>
<td>Write</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>Write</td>
<td>Read</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>Write</td>
<td>Write</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>Read</td>
<td>Read</td>
<td>Does not matter</td>
<td></td>
</tr>
</tbody>
</table>

Original Code

```
int A[N], B[N], i;
for (i=0; i<N; i++)
```

...
Example 3

Vectorization  (SISD ⇒ SIMD) : No
Parallelization (SISD ⇒ MIMD) : No

int A[N], B[N], i;
for (i=0; i<N; i++)
    A[i+1] = A[i] + B[i+1];

Observe reads and writes into a given location
Example 3

Vectorization (SISD ⇒ SIMD) : No
Parallelization (SISD ⇒ MIMD) : No

```c
int A[N], B[N], i;
for (i=0; i<N; i++)
    A[i+1] = A[i] + B[i+1];
```

Observe reads and writes into a given location

```
A[0..N]    B[0..N]
```

...
Example 3

Vectorization (SISD $\Rightarrow$ SIMD) : No
Parallelization (SISD $\Rightarrow$ MIMD) : No

int A[N], B[N], i;
for (i=0; i<N; i++)
    A[i+1] = A[i] + B[i+1];

Observe reads and writes into a given location...
Example 3

Vectorization (SISD $\Rightarrow$ SIMD) : No
Parallelization (SISD $\Rightarrow$ MIMD) : No

```c
int A[N], B[N], i;
for (i=0; i<N; i++)
    A[i+1] = A[i] + B[i+1];
```

Observe reads and writes into a given location

A[0..N]  B[0..N]

Iteration # 1
Example 3

Vectorization (SISD ⇒ SIMD) : No
Parallelization (SISD ⇒ MIMD) : No

```c
int A[N], B[N], i;
for (i=0; i<N; i++)
    A[i+1] = A[i] + B[i+1];
```

Observe reads and writes into a given location
Example 3

Vectorization (SISD $\Rightarrow$ SIMD) : No
Parallelization (SISD $\Rightarrow$ MIMD) : No

```c
int A[N], B[N], i;
for (i=0; i<N; i++)
    A[i+1] = A[i] + B[i+1];
```

Observe reads and writes into a given location
Example 3

Vectorization (SISD ⇒ SIMD) : No
Parallelization (SISD ⇒ MIMD) : No

```
int A[N], B[N], i;
for (i=0; i<N; i++)
    A[i+1] = A[i] + B[i+1];
```

Observe reads and writes into a given location
**Example 3**

Vectorization (SISD $\Rightarrow$ SIMD) : No
Parallelization (SISD $\Rightarrow$ MIMD) : No

```c
int A[N], B[N], i;
for (i=0; i<N; i++)
    A[i+1] = A[i] + B[i+1];
```

Observe reads and writes into a given location
Example 3

Vectorization (SISD $\Rightarrow$ SIMD) : No
Parallelization (SISD $\Rightarrow$ MIMD) : No

```c
int A[N], B[N], i;
for (i=0; i<N; i++)
    A[i+1] = A[i] + B[i+1];
```

Observe reads and writes into a given location

Iteration #

1 2 3 4 5 6
**Example 3**

Vectorization (SISD $\Rightarrow$ SIMD) : No
Parallelization (SISD $\Rightarrow$ MIMD) : No

```c
int A[N], B[N], i;
for (i=0; i<N; i++)
    A[i+1] = A[i] + B[i+1];
```

Observe reads and writes into a given location
Example 3

Vectorization (SISD $\Rightarrow$ SIMD) : No
Parallelization (SISD $\Rightarrow$ MIMD) : No

```c
int A[N], B[N], i;
for (i=0; i<N; i++)
    A[i+1] = A[i] + B[i+1];
```

Observe reads and writes into a given location

![Diagram](image-url)
Example 3

Vectorization (SISD ⇒ SIMD) : No
Parallelization (SISD ⇒ MIMD) : No

```c
int A[N], B[N], i;
for (i=0; i<N; i++)
    A[i+1] = A[i] + B[i+1];
```

Observe reads and writes into a given location

```
A[0..N]
B[0..N]
Iteration #
```

```
... 1 2 3 4 5 6 7 8 9 ...
```
Example 3

Vectorization (SISD ⇒ SIMD) : No
Parallelization (SISD ⇒ MIMD) : No

int A[N], B[N], i;
for (i=0; i<N; i++)
    A[i+1] = A[i] + B[i+1];

Observe reads and writes into a given location
Example 3

Vectorization (SISD ⇒ SIMD) : No
Parallelization (SISD ⇒ MIMD) : No

int A[N], B[N], i;
for (i=0; i<N; i++)
    A[i+1] = A[i] + B[i+1];

Observe reads and writes into a given location
Example 3

Vectorization \( (\text{SISD} \Rightarrow \text{SIMD}) \) : No
Parallelization \( (\text{SISD} \Rightarrow \text{MIMD}) \) : No

```c
int A[N], B[N], i;
for (i=0; i<N; i++)
    A[i+1] = A[i] + B[i+1];
```

Observe reads and writes into a given location.

A\[0..N\]  
B\[0..N\]  
Iteration #: 1 2 3 4 5 6 7 8 9 10 11 12 ···
Example 3

Vectorization (SISD ⇒ SIMD) : No
Parallelization (SISD ⇒ MIMD) : No

int A[N], B[N];
for (i=0; i<N; i++)
  A[i+1] = A[i] + B[i+1];

Nature of accesses in our example

<table>
<thead>
<tr>
<th>Observation</th>
<th>Read</th>
<th>Write</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>No</td>
<td>Write</td>
</tr>
<tr>
<td>2</td>
<td>Write</td>
<td>Read</td>
</tr>
<tr>
<td>3</td>
<td>Write</td>
<td>Write</td>
</tr>
<tr>
<td>4</td>
<td>Read</td>
<td>Read</td>
</tr>
</tbody>
</table>

Essential Abstractions in GCC

GCC Resource Center, IIT Bombay
Example 3

Vectorization (SISD ⇒ SIMD) : No
Parallelization (SISD ⇒ MIMD) : No

<table>
<thead>
<tr>
<th>Nature of accesses in our example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Iteration $i$</td>
</tr>
<tr>
<td>----------------</td>
</tr>
<tr>
<td>Read</td>
</tr>
<tr>
<td>Write</td>
</tr>
<tr>
<td>Write</td>
</tr>
<tr>
<td>Read</td>
</tr>
</tbody>
</table>

int A[N], B[N];
for (i=0; i<N; i++)
    A[i+1] = A[i] + B[i+1];

Nature of accesses in our example

A[0..N]  B[0..N]

Iteration # 1 2 3 4 5 6 7 8 9 10 11 12 . . .
Example 3

Vectorization (SISD ⇒ SIMD) : No
Parallelization (SISD ⇒ MIMD) : No

```
int A[N], B[N], i;
for (i=0; i<N; i++)
  A[i+1] = A[i] + B[i+1];
```

<table>
<thead>
<tr>
<th>Nature of accesses in our example</th>
<th>Observation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Iteration $i$</td>
<td>Iteration $i + k$</td>
</tr>
<tr>
<td>Read</td>
<td>Write</td>
</tr>
<tr>
<td>Write</td>
<td>Read</td>
</tr>
<tr>
<td>Write</td>
<td>Write</td>
</tr>
<tr>
<td>Read</td>
<td>Read</td>
</tr>
</tbody>
</table>

Essential Abstractions in GCC

GCC Resource Center, IIT Bombay
Example 3

Vectorization (SISD ⇒ SIMD) : No
Parallelization (SISD ⇒ MIMD) : No

<table>
<thead>
<tr>
<th>Nature of accesses in our example</th>
<th>Observation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Iteration $i$</td>
<td>Iteration $i + k$</td>
</tr>
<tr>
<td>Read</td>
<td>Write</td>
</tr>
<tr>
<td>Write</td>
<td>Read</td>
</tr>
<tr>
<td>Write</td>
<td>Write</td>
</tr>
<tr>
<td>Read</td>
<td>Read</td>
</tr>
</tbody>
</table>

int A[N], B[N];
for (i=0; i<N; i++)
    A[i+1] = A[i] + B[i+1];
Example 4

Vectorization  (SISD $\Rightarrow$ SIMD)  : No
Parallelization  (SISD $\Rightarrow$ MIMD)  : Yes
Example 4

Vectorization (SISD ⇒ SIMD) : No
Parallelization (SISD ⇒ MIMD) : Yes

• This case is not possible
Example 4

Vectorization  (SISD ⇒ SIMD)  : No
Parallelization (SISD ⇒ MIMD)  : Yes

- This case is not possible
- Vectorization is a limited granularity parallelization
Example 4

Vectorization  \( \text{(SISD} \Rightarrow \text{SIMD}) \)  : No
Parallelization  \( \text{(SISD} \Rightarrow \text{MIMD}) \)  : Yes

- This case is not possible
- Vectorization is a limited granularity parallelization
- If parallelization is possible then vectorization is trivially possible
Data Dependence

Let statements $S_i$ and $S_j$ access memory location $m$ at time instants $t$ and $t + k$

<table>
<thead>
<tr>
<th>Access in $S_i$</th>
<th>Access in $S_j$</th>
<th>Dependence</th>
<th>Notation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read $m$</td>
<td>Write $m$</td>
<td>Anti (or Pseudo)</td>
<td>$S_i \bar{\delta} S_j$</td>
</tr>
<tr>
<td>Write $m$</td>
<td>Read $m$</td>
<td>Flow (or True)</td>
<td>$S_i \delta S_j$</td>
</tr>
<tr>
<td>Write $m$</td>
<td>Write $m$</td>
<td>Output (or Pseudo)</td>
<td>$S_i \delta^O S_j$</td>
</tr>
<tr>
<td>Read $m$</td>
<td>Read $m$</td>
<td>Does not matter</td>
<td></td>
</tr>
</tbody>
</table>

- Pseudo dependences may be eliminated by some transformations
- True dependence prohibits parallel execution of $S_i$ and $S_j$
Consider dependence between statements $S_i$ and $S_j$ in a loop

- **Loop independent dependence.** $t$ and $t + k$ occur in the same iteration of a loop
  - $S_i$ and $S_j$ must be executed sequentially
  - Different iterations of the loop can be parallelized

- **Loop carried dependence.** $t$ and $t + k$ occur in the different iterations of a loop
  - Within an iteration, $S_i$ and $S_j$ can be executed in parallel
  - Different iterations of the loop must be executed sequentially

- $S_i$ and $S_j$ may have both loop carried and loop independent dependences
Dependence in Example 1

• Program

```c
int A[N], B[N], i;
for (i=1; i<N; i++)
    A[i] = A[i] + B[i-1]; /* S1 */
```

• Dependence graph

```
S_1
```

• No loop carried dependence
  Both vectorization and parallelization are possible
Dependence in Example 1

- Program

```c
int A[N], B[N], i;
for (i=1; i<N; i++)
    A[i] = A[i] + B[i-1]; /* S1 */
```

- Dependence graph

- No loop carried dependence
  Both vectorization and parallelization are possible
Dependence in Example 2

- **Program**

```c
int A[N], B[N], i;
for (i=0; i<N; i++)
    A[i] = A[i+1] + B[i]; /* S1 */
```

- **Dependence graph**

  \[ S_1 \xrightarrow{\delta_1} \]

- **Loop carried anti-dependence**
  Parallelization is not possible
  Vectorization is possible since all reads are done before all writes
Dependence in Example 2

- **Program**

```c
int A[N], B[N], i;
for (i=0; i<N; i++)
    A[i] = A[i+1] + B[i]; /* S1 */
```

- **Dependence graph**

  Dependence due to the outermost loop

- **Loop carried anti-dependence**
  Parallelization is not possible
  Vectorization is possible since all reads are done before all writes
Dependence in Example 3

- Program

```c
int A[N], B[N], i;
for (i=0; i<N; i++)
    A[i+1] = A[i] + B[i+1]; /* S1 */
```

- Dependence graph

- Loop carried flow-dependence
  Neither parallelization not vectorization is possible
for (i=0, i<4; i++)
    for (j=0; j<4; j++)
    {
        a[i+1][j] = a[i][j] + 2;
    }

<table>
<thead>
<tr>
<th>Iteration Vector</th>
<th>Index Vector</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>LHS</td>
</tr>
<tr>
<td>0,0</td>
<td>1,0</td>
</tr>
<tr>
<td>0,1</td>
<td>1,1</td>
</tr>
<tr>
<td>0,2</td>
<td>1,2</td>
</tr>
<tr>
<td>0,3</td>
<td>1,3</td>
</tr>
<tr>
<td>1,0</td>
<td>2,0</td>
</tr>
<tr>
<td>1,1</td>
<td>2,1</td>
</tr>
<tr>
<td>1,2</td>
<td>2,2</td>
</tr>
<tr>
<td>1,3</td>
<td>2,3</td>
</tr>
<tr>
<td>2,0</td>
<td>3,0</td>
</tr>
<tr>
<td>2,1</td>
<td>3,1</td>
</tr>
<tr>
<td>2,2</td>
<td>3,2</td>
</tr>
<tr>
<td>2,3</td>
<td>3,3</td>
</tr>
<tr>
<td>3,0</td>
<td>4,0</td>
</tr>
<tr>
<td>3,1</td>
<td>4,1</td>
</tr>
<tr>
<td>3,2</td>
<td>4,2</td>
</tr>
<tr>
<td>3,3</td>
<td>4,3</td>
</tr>
</tbody>
</table>
Iteration Vectors and Index Vectors: Example 1

for (i=0, i<4; i++)
    for (j=0; j<4; j++)
    {
        a[i+1][j] = a[i][j] + 2;
    }

Loop carried dependence exists if

- there are two distinct iteration vectors such that
- the index vectors of LHS and RHS are identical

<table>
<thead>
<tr>
<th>Iteration Vector</th>
<th>Index Vector</th>
</tr>
</thead>
<tbody>
<tr>
<td>LHS</td>
<td>RHS</td>
</tr>
<tr>
<td>0,0</td>
<td>1,0</td>
</tr>
<tr>
<td>0,1</td>
<td>1,1</td>
</tr>
<tr>
<td>0,2</td>
<td>1,2</td>
</tr>
<tr>
<td>0,3</td>
<td>1,3</td>
</tr>
<tr>
<td>1,0</td>
<td>2,0</td>
</tr>
<tr>
<td>1,1</td>
<td>2,1</td>
</tr>
<tr>
<td>1,2</td>
<td>2,2</td>
</tr>
<tr>
<td>1,3</td>
<td>2,3</td>
</tr>
<tr>
<td>2,0</td>
<td>3,0</td>
</tr>
<tr>
<td>2,1</td>
<td>3,1</td>
</tr>
<tr>
<td>2,2</td>
<td>3,2</td>
</tr>
<tr>
<td>2,3</td>
<td>3,3</td>
</tr>
<tr>
<td>3,0</td>
<td>4,0</td>
</tr>
<tr>
<td>3,1</td>
<td>4,1</td>
</tr>
<tr>
<td>3,2</td>
<td>4,2</td>
</tr>
<tr>
<td>3,3</td>
<td>4,3</td>
</tr>
</tbody>
</table>
## Iteration Vectors and Index Vectors: Example 1

```plaintext
for (i=0, i<4; i++)
    for (j=0; j<4; j++)
    {
        a[i+1][j] = a[i][j] + 2;
    }
```

Loop carried dependence exists if

- there are two distinct iteration vectors such that
- the index vectors of LHS and RHS are identical

### Conclusion: Dependence exists

<table>
<thead>
<tr>
<th>Iteration Vector</th>
<th>Index Vector</th>
</tr>
</thead>
<tbody>
<tr>
<td>LHS</td>
<td>RHS</td>
</tr>
<tr>
<td>0, 0</td>
<td>1, 0</td>
</tr>
<tr>
<td>0, 1</td>
<td>1, 1</td>
</tr>
<tr>
<td>0, 2</td>
<td>1, 2</td>
</tr>
<tr>
<td>0, 3</td>
<td>1, 3</td>
</tr>
<tr>
<td>1, 0</td>
<td>2, 0</td>
</tr>
<tr>
<td>1, 1</td>
<td>2, 1</td>
</tr>
<tr>
<td>1, 2</td>
<td>2, 2</td>
</tr>
<tr>
<td>1, 3</td>
<td>2, 3</td>
</tr>
<tr>
<td>2, 0</td>
<td>3, 0</td>
</tr>
<tr>
<td>2, 1</td>
<td>3, 1</td>
</tr>
<tr>
<td>2, 2</td>
<td>3, 2</td>
</tr>
<tr>
<td>2, 3</td>
<td>3, 3</td>
</tr>
<tr>
<td>3, 0</td>
<td>4, 0</td>
</tr>
<tr>
<td>3, 1</td>
<td>4, 1</td>
</tr>
<tr>
<td>3, 2</td>
<td>4, 2</td>
</tr>
<tr>
<td>3, 3</td>
<td>4, 3</td>
</tr>
</tbody>
</table>
Iteration Vectors and Index Vectors: Example 1

for (i=0, i<4; i++)
    for (j=0; j<4; j++)
    {
        a[i+1][j] = a[i][j] + 2;
    }

Loop carried dependence exists if
- there are two distinct iteration vectors such that
- the index vectors of LHS and RHS are identical

Conclusion: Dependence exists
### Iteration Vectors and Index Vectors: Example 1

```c
for (i=0, i<4; i++)
    for (j=0; j<4; j++)
    {
        a[i+1][j] = a[i][j] + 2;
    }
```

<table>
<thead>
<tr>
<th>Iteration Vector</th>
<th>Index Vector</th>
<th>LHS</th>
<th>RHS</th>
</tr>
</thead>
<tbody>
<tr>
<td>0,0</td>
<td>1,0</td>
<td>0,0</td>
<td>0,0</td>
</tr>
<tr>
<td>0,1</td>
<td>1,1</td>
<td>0,1</td>
<td>0,1</td>
</tr>
<tr>
<td>0,2</td>
<td>1,2</td>
<td>0,2</td>
<td>0,2</td>
</tr>
<tr>
<td>0,3</td>
<td>1,3</td>
<td>0,3</td>
<td>0,3</td>
</tr>
<tr>
<td>1,0</td>
<td>2,0</td>
<td>1,0</td>
<td>1,0</td>
</tr>
<tr>
<td>1,1</td>
<td>2,1</td>
<td>1,1</td>
<td>1,1</td>
</tr>
<tr>
<td>1,2</td>
<td>2,2</td>
<td>1,2</td>
<td>1,2</td>
</tr>
<tr>
<td>1,3</td>
<td>2,3</td>
<td>1,3</td>
<td>1,3</td>
</tr>
<tr>
<td>2,0</td>
<td>3,0</td>
<td>2,0</td>
<td>2,0</td>
</tr>
<tr>
<td>2,1</td>
<td>3,1</td>
<td>2,1</td>
<td>2,1</td>
</tr>
<tr>
<td>2,2</td>
<td>3,2</td>
<td>2,2</td>
<td>2,2</td>
</tr>
<tr>
<td>2,3</td>
<td>3,3</td>
<td>2,3</td>
<td>2,3</td>
</tr>
<tr>
<td>3,0</td>
<td>4,0</td>
<td>3,0</td>
<td>3,0</td>
</tr>
<tr>
<td>3,1</td>
<td>4,1</td>
<td>3,1</td>
<td>3,1</td>
</tr>
<tr>
<td>3,2</td>
<td>4,2</td>
<td>3,2</td>
<td>3,2</td>
</tr>
<tr>
<td>3,3</td>
<td>4,3</td>
<td>3,3</td>
<td>3,3</td>
</tr>
</tbody>
</table>
```

Loop carried dependence exists if

- there are two distinct iteration vectors such that
- the index vectors of LHS and RHS are identical

**Conclusion:** Dependence exists
for (i=0, i<4; i++)
    for (j=0; j<4; j++)
    {
        a[i][j] = a[i][j] + 2;
    }

<table>
<thead>
<tr>
<th>Iteration Vector</th>
<th>Index Vector</th>
<th>LHS</th>
<th>RHS</th>
</tr>
</thead>
<tbody>
<tr>
<td>0,0</td>
<td>0,0</td>
<td>0,0</td>
<td>0,0</td>
</tr>
<tr>
<td>0,1</td>
<td>0,1</td>
<td>0,1</td>
<td>0,1</td>
</tr>
<tr>
<td>0,2</td>
<td>0,2</td>
<td>0,2</td>
<td>0,2</td>
</tr>
<tr>
<td>0,3</td>
<td>0,3</td>
<td>0,3</td>
<td>0,3</td>
</tr>
<tr>
<td>1,0</td>
<td>1,0</td>
<td>1,0</td>
<td>1,0</td>
</tr>
<tr>
<td>1,1</td>
<td>1,1</td>
<td>1,1</td>
<td>1,1</td>
</tr>
<tr>
<td>1,2</td>
<td>1,2</td>
<td>1,2</td>
<td>1,2</td>
</tr>
<tr>
<td>1,3</td>
<td>1,3</td>
<td>1,3</td>
<td>1,3</td>
</tr>
<tr>
<td>2,0</td>
<td>2,0</td>
<td>2,0</td>
<td>2,0</td>
</tr>
<tr>
<td>2,1</td>
<td>2,1</td>
<td>2,1</td>
<td>2,1</td>
</tr>
<tr>
<td>2,2</td>
<td>2,2</td>
<td>2,2</td>
<td>2,2</td>
</tr>
<tr>
<td>2,3</td>
<td>2,3</td>
<td>2,3</td>
<td>2,3</td>
</tr>
<tr>
<td>3,0</td>
<td>3,0</td>
<td>3,0</td>
<td>3,0</td>
</tr>
<tr>
<td>3,1</td>
<td>3,1</td>
<td>3,1</td>
<td>3,1</td>
</tr>
<tr>
<td>3,2</td>
<td>3,2</td>
<td>3,2</td>
<td>3,2</td>
</tr>
<tr>
<td>3,3</td>
<td>3,3</td>
<td>3,3</td>
<td>3,3</td>
</tr>
</tbody>
</table>
Iteration Vectors and Index Vectors: Example 2

```
for (i=0, i<4; i++)
    for (j=0; j<4; j++)
        { 
            a[i][j] = a[i][j] + 2;
        }
```

Loop carried dependence exists if
- there are two distinct iteration vectors such that
- the index vectors of LHS and RHS are identical

<table>
<thead>
<tr>
<th>Iteration Vector</th>
<th>Index Vector</th>
</tr>
</thead>
<tbody>
<tr>
<td>LHS</td>
<td>RHS</td>
</tr>
<tr>
<td>0,0</td>
<td>0,0</td>
</tr>
<tr>
<td>0,1</td>
<td>0,1</td>
</tr>
<tr>
<td>0,2</td>
<td>0,2</td>
</tr>
<tr>
<td>0,3</td>
<td>0,3</td>
</tr>
<tr>
<td>1,0</td>
<td>1,0</td>
</tr>
<tr>
<td>1,1</td>
<td>1,1</td>
</tr>
<tr>
<td>1,2</td>
<td>1,2</td>
</tr>
<tr>
<td>1,3</td>
<td>1,3</td>
</tr>
<tr>
<td>2,0</td>
<td>2,0</td>
</tr>
<tr>
<td>2,1</td>
<td>2,1</td>
</tr>
<tr>
<td>2,2</td>
<td>2,2</td>
</tr>
<tr>
<td>2,3</td>
<td>2,3</td>
</tr>
<tr>
<td>3,0</td>
<td>3,0</td>
</tr>
<tr>
<td>3,1</td>
<td>3,1</td>
</tr>
<tr>
<td>3,2</td>
<td>3,2</td>
</tr>
<tr>
<td>3,3</td>
<td>3,3</td>
</tr>
</tbody>
</table>
Iteration Vectors and Index Vectors: Example 2

for (i=0, i<4; i++)
    for (j=0; j<4; j++)
    {
        a[i][j] = a[i][j] + 2;
    }

Loop carried dependence exists if

- there are two distinct iteration vectors such that
- the index vectors of LHS and RHS are identical

Conclusion: No dependence
## Example 4: Dependence

### Program to swap arrays

```c
for (i=0; i<N; i++)
{
    T = A[i];  /* S1 */
    A[i] = B[i]; /* S2 */
    B[i] = T;  /* S3 */
}
```

### Dependence Graph

```
\begin{tikzpicture}
    \node (s1) at (0,0) {$S_1$};
    \node (s2) at (2,0) {$S_2$};
    \node (s3) at (-2,0) {$S_3$};
    \draw[->] (s1) to [bend left] node [above] {$\delta_1^{\text{O}}$} (s2);
    \draw[->] (s2) to [bend right] node [below] {$\delta_\infty$} (s1);
    \draw[->] (s3) to [bend right] node [below] {$\bar{\delta}_\infty$} (s1);
    \draw[->] (s3) to [bend left] node [above] {$\bar{\delta}_1$} (s2);
    \draw[->] (s1) to [bend left] node [below] {$\bar{\delta}_\infty$} (s3);
    \end{tikzpicture}
```
Example 4: Dependence

Program to swap arrays

```
for (i=0; i<N; i++)
{
    T = A[i];  /* S1 */
    A[i] = B[i];  /* S2 */
    B[i] = T;  /* S3 */
}
```

Dependence Graph

Loop independent anti dependence due to A[i]
Example 4: Dependence

Program to swap arrays

```
for (i=0; i<N; i++)
{
    T = A[i]; /* S1 */
    A[i] = B[i]; /* S2 */
    B[i] = T; /* S3 */
}
```

Dependence Graph

Loop independent anti dependence due to B[i]
**Example 4: Dependence**

Program to swap arrays

```
for (i=0; i<N; i++)
{
    T = A[i]; /* S1 */
    A[i] = B[i]; /* S2 */
    B[i] = T; /* S3 */
}
```

Loop independent flow dependence due to T
Example 4: Dependence

Program to swap arrays

```
for (i=0; i<N; i++)
{
    T = A[i];    /* S1 */
    A[i] = B[i]; /* S2 */
    B[i] = T;    /* S3 */
}
```

Loop carried anti dependence due to \( T \)
Example 4: Dependence

Program to swap arrays

```
for (i=0; i<N; i++)
{
    T = A[i]; /* S1 */
    A[i] = B[i]; /* S2 */
    B[i] = T; /* S3 */
}
```

Dependence Graph

Loop carried output dependence due to T
Example 4: Dependence

Program to swap arrays

```c
for (i=0; i<N; i++)
{
    T = A[i]; /* S1 */
    A[i] = B[i]; /* S2 */
    B[i] = T; /* S3 */
}
```

Dependence Graph

The diagram shows the dependence graph for the program. The nodes represent the statements, and the arrows indicate the dependences between them. The graph illustrates the loop independent anti dependence due to `A[i]`.
Tutorial Problem for Discovering Dependence

Draw the dependence graph for the following program 
(Earlier program modified to swap 2-dimensional arrays)

```c
for (i=0; i<N; i++)
{
    for (j=0; j<N; j++)
    {
        T = A[i][j];       /* S1 */
        A[i][j] = B[i][j]; /* S2 */
        B[i][j] = T;       /* S3 */
    }
}
```
Data Dependence Theorem

There exists a dependence from statement $S_1$ to statement $S_2$ in common nest of loops if and only if there exist two iteration vectors $i$ and $j$ for the nest, such that

1. $i < j$ or $i = j$ and there exists a path from $S_1$ to $S_2$ in the body of the loop,
2. statement $S_1$ accesses memory location $M$ on iteration $i$ and statement $S_2$ accesses location $M$ on iteration $j$, and
3. one of these accesses is a write access.
Anti Dependence and Vectorization

Read lexicographically precedes Write

```c
int A[N], B[N], C[N], i;
for (i=0; i<N; i++) {
    C[i] = A[i+2];
    A[i] = B[i];
}
```
Anti Dependence and Vectorization

Read lexicographically precedes Write

```
int A[N], B[N], C[N], i;
for (i=0; i<N; i++) {
    C[i] = A[i+2];
    A[i] = B[i];
}
```

```
int A[N], B[N], C[N], i;
for (i=0; i<N; i=i+4) {
    C[i:i+3] = A[i+2:i+5];
    A[i:i+3] = B[i:i+3];
}
```
Anti Dependence and Vectorization

Write lexicographically precedes Read

```c
int A[N], B[N], C[N], i;
for (i=0; i<N; i++) {
    A[i] = B[i];
    C[i] = A[i+2];
}
```
Anti Dependence and Vectorization

**Write lexicographically precedes Read**

```c
int A[N], B[N], C[N], i;
for (i=0; i<N; i++) {
    A[i] = B[i];
    C[i] = A[i+2];
}
```

```c
int A[N], B[N], C[N], i;
for (i=0; i<N; i++) {
    C[i] = A[i+2];
    A[i] = B[i];
}
```
Anti Dependence and Vectorization

Write lexicographically precedes Read

```c
int A[N], B[N], C[N], i;
for (i=0; i<N; i++) {
    A[i] = B[i];
    C[i] = A[i+2];
}
```

```c
int A[N], B[N], C[N], i;
for (i=0; i<N; i++) {
    C[i] = A[i+2];
    A[i] = B[i];
}
```

```c
int A[N], B[N], C[N], i;
for (i=0; i<N; i=i+4) {
    C[i:i+3] = A[i+2:i+5];
    A[i:i+3] = B[i:i+3];
}
```
True Dependence and Vectorization

Write lexicographically precedes Read

```c
int A[N], B[N], C[N], i;
for (i=0; i<N; i++) {
    A[i+2] = C[i];
    B[i] = A[i];
}
```
True Dependence and Vectorization

Write lexicographically precedes Read

```c
int A[N], B[N], C[N], i;
for (i=0; i<N; i++) {
    A[i+2] = C[i];
    B[i] = A[i];
}
```

```c
int A[N], B[N], C[N], i;
for (i=0; i<N; i=i+4) {
    A[i+2:i+5] = C[i:i+3];
    B[i:i+3] = A[i:i+3];
}
```
Conjunction of Dependences and Vectorization

Anti Dependence and True Dependence

```c
int A[N], i;
for (i=0; i<N; i++) {
    A[i] = A[i+2];
}
```
Conjunction of Dependences and Vectorization

Anti Dependence and True Dependence

```c
int A[N], i;
for (i=0; i<N; i++) {
    A[i] = A[i+2];
}
```

```c
int A[N], i, temp;
for (i=0; i<N; i++) {
    temp = A[i+2];
    A[i] = temp;
}
```
Conjunction of Dependences and Vectorization

Anti Dependence and True Dependence

```
int A[N], i;
for (i=0; i<N; i++) {
    A[i] = A[i+2];
}
```

```
int A[N], i, temp;
for (i=0; i<N; i++) {
    temp = A[i+2];
    A[i] = temp;
}
```

```
int A[N], T[N], i;
for (i=0; i<N; i++) {
    T[i] = A[i+2];
    A[i] = T[i];
}
```
**Conjunction of Dependences and Vectorization**

**Anti Dependence and True Dependence**

```c
int A[N], i;
for (i=0; i<N; i++) {
    A[i] = A[i+2];
}
```

```c
int A[N], i, temp;
for (i=0; i<N; i++) {
    temp = A[i+2];
    A[i] = temp;
}
```

```c
int A[N], T[N], i;
for (i=0; i<N; i=i+4) {
    T[i:i+3] = A[i+2:i+5];
    A[i:i+3] = T[i:i+3];
}
```

```c
int A[N], T[N], i;
for (i=0; i<N; i++) {
    T[i] = A[i+2];
    A[i] = T[i];
}
```
Conjunction of Dependences and Vectorization

True Dependence and Anti Dependence

```c
int A[N], B[N], i;
for (i=0; i<N; i++) {
    A[i] = B[i];
    B[i+2] = A[i+1];
}
```
Conjunction of Dependences and Vectorization

True Dependence and Anti Dependence

```
int A[N], B[N], i;
for (i=0; i<N; i++) {
    A[i] = B[i];
    B[i+2] = A[i+1];
}
```

```
int A[N], B[N], i;
for (i=0; i<N; i++) {
    B[i+2] = A[i+1];
    A[i] = B[i];
}
```
Conjunction of Dependences and Vectorization

True Dependence and Anti Dependence

```c
int A[N], B[N], i;
for (i=0; i<N; i++) {
    A[i] = B[i];
    B[i+2] = A[i+1];
}
```

```c
int A[N], B[N], i;
for (i=0; i<N; i++) {
    B[i+2] = A[i+1];
    A[i] = B[i];
}
```

```c
int A[N], B[N], i;
for (i=0; i<N; i=i+4) {
    B[i+2:i+5] = A[i+1:i+4];
    A[i:i+3] = B[i:i+3];
}
```
Cyclic Dependency and Vectorization

Cyclic True Dependence

```c
int A[N], B[N], i;
for (i=0; i<N; i++) {
    B[i+2] = A[i];
    A[i+1] = B[i];
}
```
Cyclic Dependency and Vectorization

Cyclic True Dependence

```c
int A[N], B[N], i;
for (i=0; i<N; i++) {
    B[i+2] = A[i];
    A[i+1] = B[i];
}
```

Cyclic Anti Dependence

```c
int A[N], B[N], i;
for (i=0; i<N; i++) {
    B[i] = A[i+1];
    A[i] = B[i+2];
}
```
Cyclic Dependency and Vectorization

Cyclic True Dependence

```c
int A[N], B[N], i;
for (i=0; i<N; i++) {
    B[i+2] = A[i];
    A[i+1] = B[i];
}
```

Cyclic Anti Dependence

```c
int A[N], B[N], i;
for (i=0; i<N; i++) {
    B[i] = A[i+1];
    A[i] = B[i+2];
}
```

Rescheduling of statements will not break the cyclic dependency - cannot vectorize
Last but not the least . . .

Thank You!