Vector and Array Processors

Back to the SIMD architecture: now is the point where need to talk about an interesting feature of SIMD processing.
Back to the SIMD architecture: now is the point where need to talk about an interesting feature of SIMD processing

The main feature of SIMD is that all processors are doing the same thing...
Back to the SIMD architecture: now is the point where need to talk about an interesting feature of SIMD processing.

The main feature of SIMD is that all processors are doing the same thing.

... so how can conditionals work?
Back to the SIMD architecture: now is the point where need to talk about an interesting feature of SIMD processing.

The main feature of SIMD is that all processors are doing the same thing...

... so how can conditionals work?

Here is an example, written using a fictional SIMD C
Vector and Array Processors

Suppose we have a `get_proc()` function ("get processor number") that returns the index of the processor:

```c
int me;
me = get_proc();
...
```

This allows us to distinguish between processors; the value of `me` is different on each processor.
Vector and Array Processors

Suppose we have a `get_proc()` function (“get processor number”) that returns the index of the processor:

```c
int me;
me = get_proc();
...
```

This allows us to distinguish between processors; the value of `me` is different on each processor.

We could use `me` to index into a vector, so each processor operates on a different element:

```c
v[me] = (v[me - 1] + v[me + 1])/2.0;
```
So what does this code do?

```c
int me, n;

me = get_proc();

if (me > 512) {
    n = 1;
}
else {
    n = -1;
}
```
Instinctively you think it sets $n$ in processors above 512 to 1 and in the other processors $n$ is set to -1.
Vector and Array Processors

Instinctively you think it sets $n$ in processors above 512 to 1 and in the other processors $n$ is set to -1.

And this is what it does do.
Vector and Array Processors

Instinctively you think it sets $n$ in processors above 512 to 1 and in the other processors $n$ is set to -1.

And this is what it does do.

But a SIMD machine executes the same code in all processors, so how can it execute the $n = 1$ assignment on some and the $n = -1$ assignment on others?
Vector and Array Processors

It doesn’t: at any point in time each processor is executing the same instruction
Vector and Array Processors

It doesn’t: at any point in time each processor is executing the same instruction

or doing nothing at all
It doesn’t: at any point in time each processor is executing the same instruction

or doing nothing at all

Processors can be inhibited, meaning not participating in the current instruction
Vector and Array Processors

It doesn’t: at any point in time each processor is executing the same instruction

or doing nothing at all

Processors can be inhibited, meaning not participating in the current instruction

There is a per-processor inhibit flag to say whether this processor is on or off
Vector and Array Processors

It doesn’t: at any point in time each processor is executing the same instruction

or doing nothing at all

Processors can be inhibited, meaning not participating in the current instruction

There is a per-processor inhibit flag to say whether this processor is on or off

This is how we get different code paths on different processors
Vector and Array Processors

We must modify our description of SIMD machines:

*Each processor either executes the same instruction as the others; or does nothing at all*
Vector and Array Processors

Returning to the code

```java
if (me > 512) {
    n = 1;
}
else {
    n = -1;
}
```

This is executed as follows:

- All processors execute the test in the `if`.
- In those processors for which the test fails, the inhibit flag is set.
- All processors move to the `n = 1`; the inhibited processors do nothing while the others execute the assignment.
Vector and Array Processors

Returning to the code

```java
if (me > 512) {
    n = 1;
}
else {
    n = -1;
}
```

This is executed as follows:
Vector and Array Processors

Returning to the code

```java
if (me > 512) {
    n = 1;
} else {
    n = -1;
}
```

This is executed as follows:

- All processors execute the test in the if
Vector and Array Processors

Returning to the code

```java
if (me > 512) {
    n = 1;
} else {
    n = -1;
}
```

This is executed as follows:

- All processors execute the test in the `if`
- In those processors for which the test fails, the inhibit flag is set
Vector and Array Processors

Returning to the code

```java
if (me > 512) {
    n = 1;
}
else {
    n = -1;
}
```

This is executed as follows:

- All processors execute the test in the `if`
- In those processors for which the test fails, the inhibit flag is set
- All processors move to the `n = 1`; the inhibited processors do nothing while the others execute the assignment
Vector and Array Processors

- All processors move to the `else`; the inhibit flag is inverted
Vector and Array Processors

- All processors move to the `else`; the inhibit flag is inverted
- All processors move to the `n = -1`; the inhibited processors do nothing while the others execute the assignment
Vector and Array Processors

- All processors move to the `else`; the inhibit flag is inverted
- All processors move to the \( n = -1 \); the inhibited processors do nothing while the others execute the assignment
- The inhibit flag is cleared
Vector and Array Processors

- All processors move to the `else`; the inhibit flag is inverted
- All processors move to the \( n = -1 \); the inhibited processors do nothing while the others execute the assignment
- The inhibit flag is cleared
- All processors move on to after the `if`
Vector and Array Processors

- All processors move to the `else`; the inhibit flag is inverted
- All processors move to the `n = -1`; the inhibited processors do nothing while the others execute the assignment
- The inhibit flag is cleared
- All processors move on to after the `if`

Both branches of an `if` always taken by all processors!
### Vector and Array Processors

<table>
<thead>
<tr>
<th>Proc</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>...</th>
<th>513</th>
<th>514</th>
<th>515</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>inhibit</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>...</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>n</td>
<td>n</td>
<td>n</td>
<td>n</td>
<td>...</td>
<td>n</td>
<td>n</td>
<td>n</td>
<td>n</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>...</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Vector and Array Processors

<table>
<thead>
<tr>
<th>Proc</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>...</th>
<th>513</th>
<th>514</th>
<th>515</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>inhibit</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>...</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>...</td>
</tr>
<tr>
<td>if (me &gt; 512)</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>...</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>...</td>
</tr>
</tbody>
</table>
Vector and Array Processors

<table>
<thead>
<tr>
<th>Proc</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>...</th>
<th>513</th>
<th>514</th>
<th>515</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>inhibit</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>...</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>n = 1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>...</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>...</td>
</tr>
</tbody>
</table>

n = 1
Vector and Array Processors

<table>
<thead>
<tr>
<th>Proc</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>...</th>
<th>513</th>
<th>514</th>
<th>515</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>inhibit</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>...</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>else</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>...</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Vector and Array Processors

| Proc | 0 | 1 | 2 | ... | 513 | 514 | 515 | ...
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>inhibit</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
</tbody>
</table>

\[ n = -1 \quad -1 \quad -1 \quad -1 \quad \ldots \quad 1 \quad 1 \quad 1 \quad 1 \quad \ldots \]
<table>
<thead>
<tr>
<th>Proc</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>...</th>
<th>513</th>
<th>514</th>
<th>515</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>inhibit</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>...</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>n</td>
<td>n</td>
<td>n</td>
<td>n</td>
<td>...</td>
<td>n</td>
<td>n</td>
<td>n</td>
<td>n</td>
</tr>
<tr>
<td>after</td>
<td>−1</td>
<td>−1</td>
<td>−1</td>
<td>...</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Vector and Array Processors

The time taken for an if is the sum of the times of both branches.
The time taken for an if is the sum of the times of both branches

Quite different from sequential code
Vector and Array Processors

The time taken for an if is the sum of the times of both branches

Quite different from sequential code

Reality is a little more complicated: think about nested if s
Vector and Array Processors

The time taken for an if is the sum of the times of both branches

Quite different from sequential code

Reality is a little more complicated: think about nested if's

There is actually a stack of inhibit flags!
Vector and Array Processors

The time taken for an *if* is the sum of the times of both branches

Quite different from sequential code

Reality is a little more complicated: think about nested *if*s

There is actually a *stack* of inhibit flags!

Exercise. Think this through for yourself!
Vector and Array Processors

This seems like poor use of our processors if lots of them are inhibited

if (me > 512) foo();
else bar();

is not good code: all of foo must be executed before bar can start, so there is a large amount of serialisation
Vector and Array Processors

This seems like poor use of our processors if lots of them are inhibited

True, so SIMD code should be written to minimise conditional branches
Vector and Array Processors

This seems like poor use of our processors if lots of them are inhibited

True, so SIMD code should be written to minimise conditional branches

But with thousands of CPUs, processing power is cheap, so inhibiting some of them is not as bad as it seems, as long as it is not overdone
Vector and Array Processors

This seems like poor use of our processors if lots of them are inhibited

True, so SIMD code should be written to minimise conditional branches

But with thousands of CPUs, processing power is cheap, so inhibiting some of them is not as bad as it seems, as long as it is not overdone

```
if (me > 512) foo();
else bar();
```

is not good code: all of foo must be executed before bar can start, so there is a large amount of serialisation
Vector and Array Processors

Inhibition applies to all conditional code, like loops:

```c
int i, n;
...
for (i = 0; i < n; i++) {
    ...
}
```
Vector and Array Processors

Inhibition applies to all conditional code, like loops:

```c
int i, n;
...
for (i = 0; i < n; i++) {
    ...
}
```

All processors start the loop
Inhibition applies to all conditional code, like loops:

```c
int i, n;
...
for (i = 0; i < n; i++) {
    ...
}
```

All processors start the loop

As $i$ increases, some processors pass their exit test and are inhibited; other processors continue executing; all processors continue looping
Inhibition applies to all conditional code, like loops:

```c
int i, n;
...
for (i = 0; i < n; i++) {
    ...
}
```

All processors start the loop

As \( i \) increases, some processors pass their exit test and are inhibited; other processors continue executing; *all processors continue looping*

Note no processor starts executing after the loop until *all processors* have exited
Loops must wait until all processors have completed: they take time the maximum of the individual processors.
Loops must wait until all processors have completed: they take time the maximum of the individual processors

SIMD loops are most efficient when all the loops are of the same size
Vector and Array Processors

Loops must wait until all processors have completed: they take time the maximum of the individual processors

SIMD loops are most efficient when all the loops are of the same size

Similarly for all conditional constructs: if there is a choice all processors will take all the choices, but some are appropriately inhibited
Connection Machines had a lightbulb per processor: initially they set it so the light was on when the processor was active...
Vector and Array Processors

Connection Machines had a lightbulb per processor: initially they set it so the light was on when the processor was active. After a while they fixed it so the light was on when the processor was inhibited...
Vector and Array Processors

Connection Machines had a lightbulb per processor: initially they set it so the light was on when the processor was active.

After a while they fixed it so the light was on when the processor was inhibited.

We shall return to SIMD programming with CUDA, later, when we talk about parallel languages.