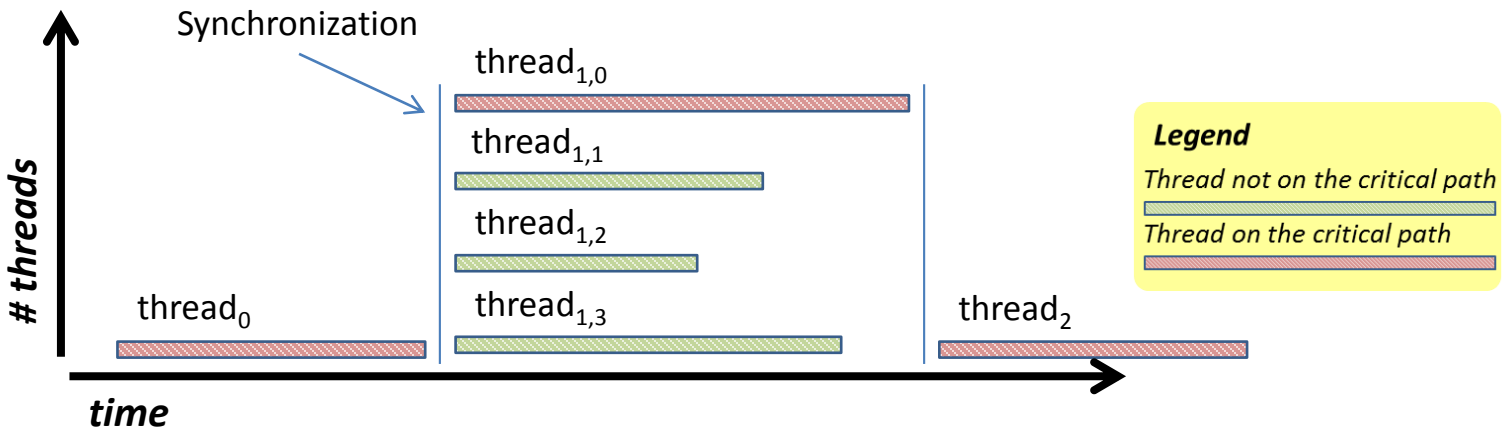
Instruction-Level Parallelism and Automatic Vectorization

Antoine Trouvé
アントワン　トルヴェ
trouve@isit.or.jp
2014/06/23

Execution Time

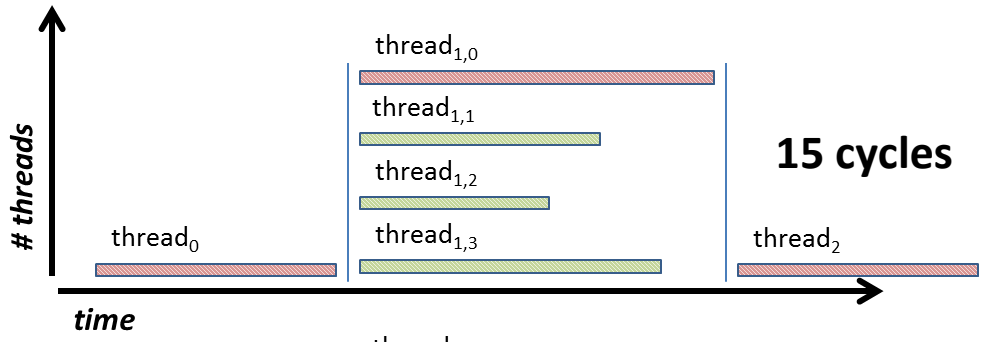
- Reminder: **what is a thread**
  - A sequence of instruction
  - Traditionally programs used to be made of one thread
  - Modern programs use threads in order to parallelize calculations
- Execution time of a multi-thread program
  - Decided by the execution time of the threads on the **critical path**
  - It can be reduced by raising **single thread performances**

$$time = time(thread_0) + \max_i\left(time(thread_{1,i})\right) + time(thread_2)$$



= reduce the execution time

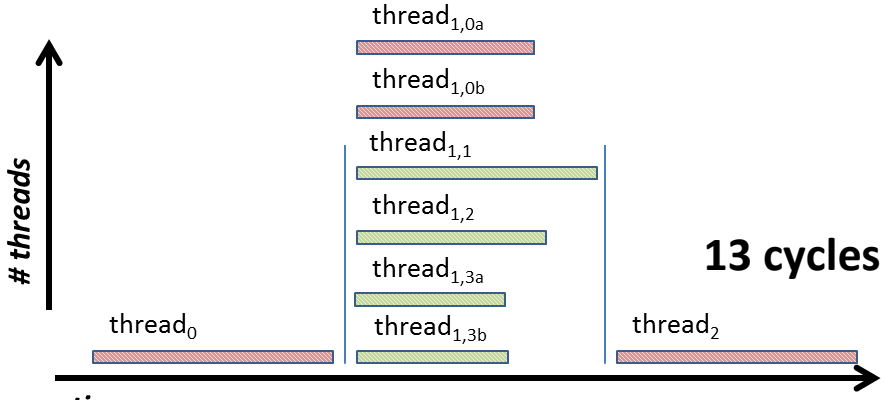
How to Raise Performances ?



Legend

Thread not on the critical path

Thread on the critical path



Method 1: Use more threads

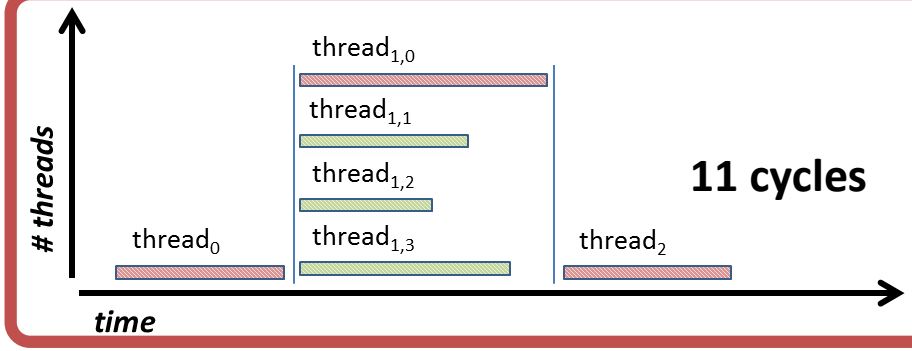
By doing so, we raise the thread-level parallelism.

However, it is not always possible to do.

Not always efficient (Amdahl law)

Moreover, it is hard to do automatically inside the compiler: the programmer should do it himself !

Example: cut thread$_{1,0}$ and thread$_{1,3}$ in half.



Method 2: Raise single-thread performances

Depends on both the hardware and the software.

See next slide…

Example: 1.36 times faster single-thread performances = 1.36 times faster program

About Single-Thread Performances

- What are single-thread performances ?
  - The speed at which a given computing core executes sequential instructions
- How do we calculate it ?

$$\frac{\boldsymbol{instruction}}{\boldsymbol{time}} = \frac{instruction}{cycle} \times \frac{cycle}{time} = \boldsymbol{IPC} \times \boldsymbol{freq}$$

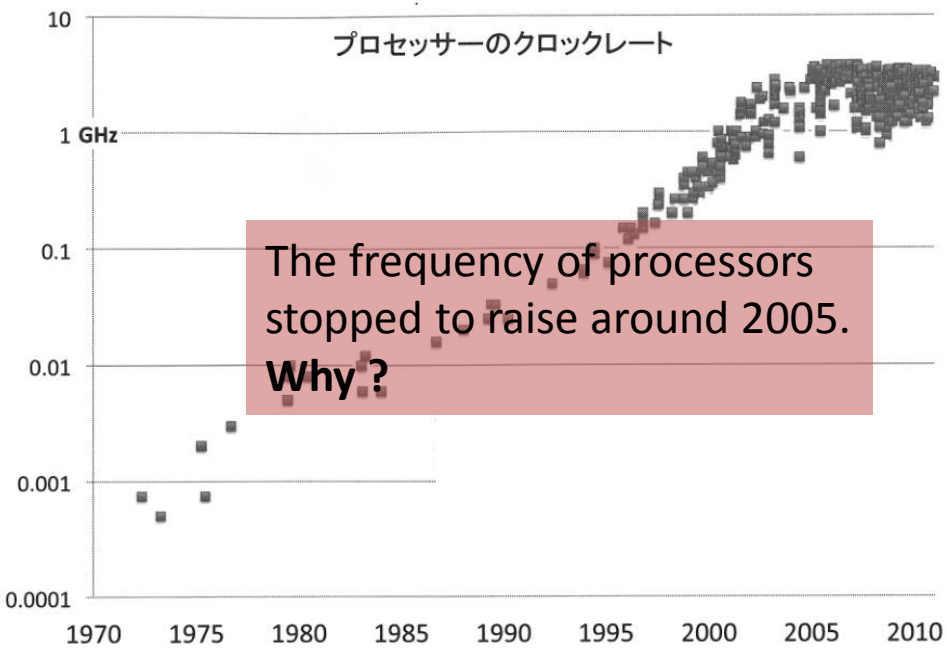
Unit: instruction per second

Instruction per Cycle

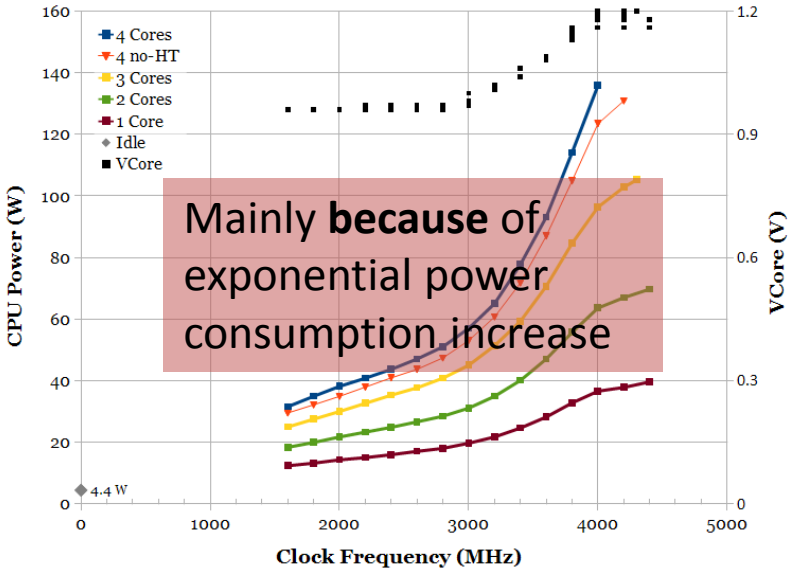
Frequency

Raise Single Threads Performances
① Can we Raise the Frequency ?

$$Single\ Thread\ Performance = IPC \times freq$$



Source: *Hakata Intel Software Conference 2012*



Source: *http://blog.stuffedcow.net/*

Raise Single Threads Performances
① Can we Raise the Frequency ?

Too Much Power !

Too Hot !

$$Single\ Thread\ Performance = IPC \times freq$$

Source: Hakata Intel Softwa… erence 2012

Source: http://… dcow.net/

Raising Single Threads Performances
② Can we Raise the IPC ?

$$Single\ Thread\ Performance = IPC \times freq$$

- Definition of the IPC
  - Instruction Per Cycle
  - The amount of calculation the core is performing per cycle for a given program
- The higher the IPC, the faster
- The IPC is calculated as follows:
  - $IPC = \frac{number\ of\ instructions\ executed}{number\ of\ cycles}$
- **It can be raised by**
  - **Reducing the number of instructions**
  - **Raising the amount of work the processor can do per cycle**

Raising Single Threads Performances
② Can we Raise the IPC ?

$Single\ T = IPC \times freq$

- Definition of the I
  - Instruction Per C
  - The amount of c orming per cycle for a given program
- The higher the IP
- The ILP is calculat
  - $ILP = \frac{number\ o}{nu}$
- **It can be raised b**
  - Reducing the number of instructions
  - Raising the amount of work the processor can do per cycle

IPC, ILP, ASAP

HARDWARE PEAK IPC AND ILP

Starting Point: the IPC

- Recap from previous slide
  - Instruction Per Cycle
  - The amount of calculation the core is performing per cycle for a given program
  - The higher the IPC, the faster the program
- The IPC is calculated as follows:
  - $IPC = \frac{number\ of\ instructions\ executed}{number\ of\ cycles}$
- For a given hardware and program, the IPC depends on
  - the hardware peak IPC (**IPC_{peak}**)
  - the software Instruction Level Parallelism (**ILP**)
  - the scheduling algorithm used by the core to execute instructions
- The maximum IPC can be defined as:

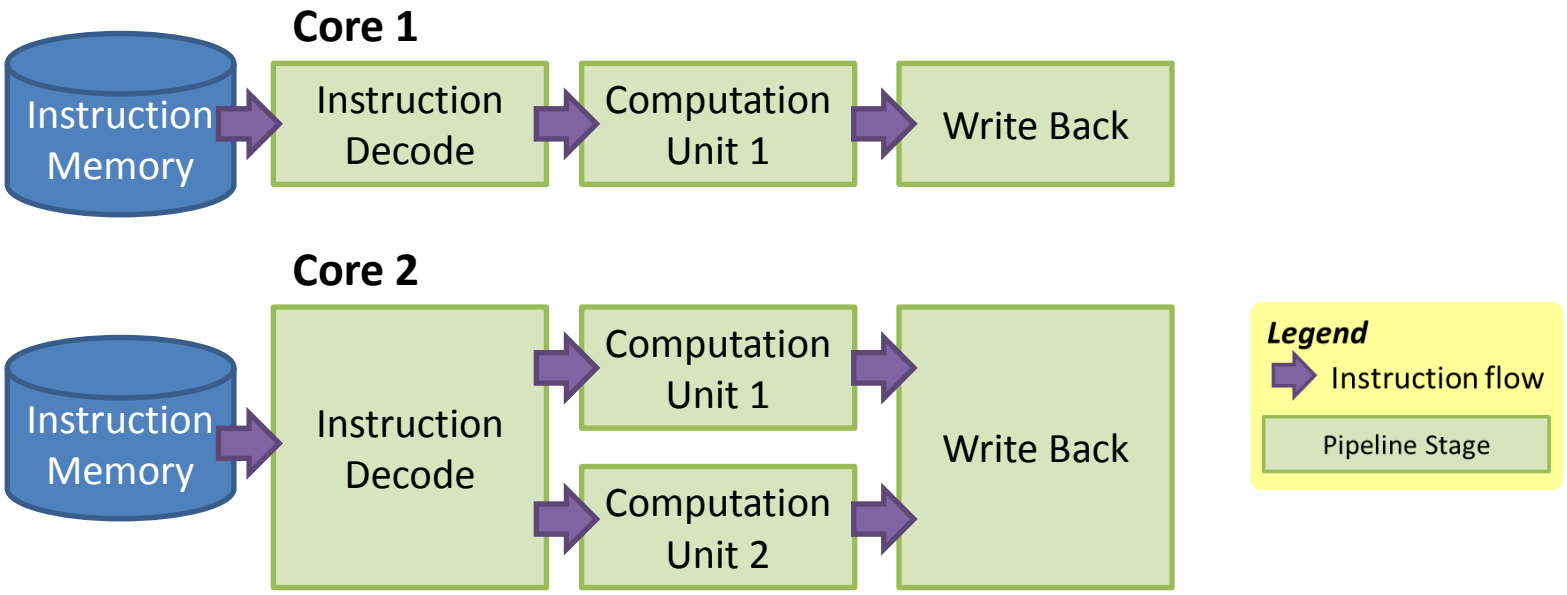
$$\text{IPC} < \min(IPC_{peak}, ILP)$$

Hardware dependent

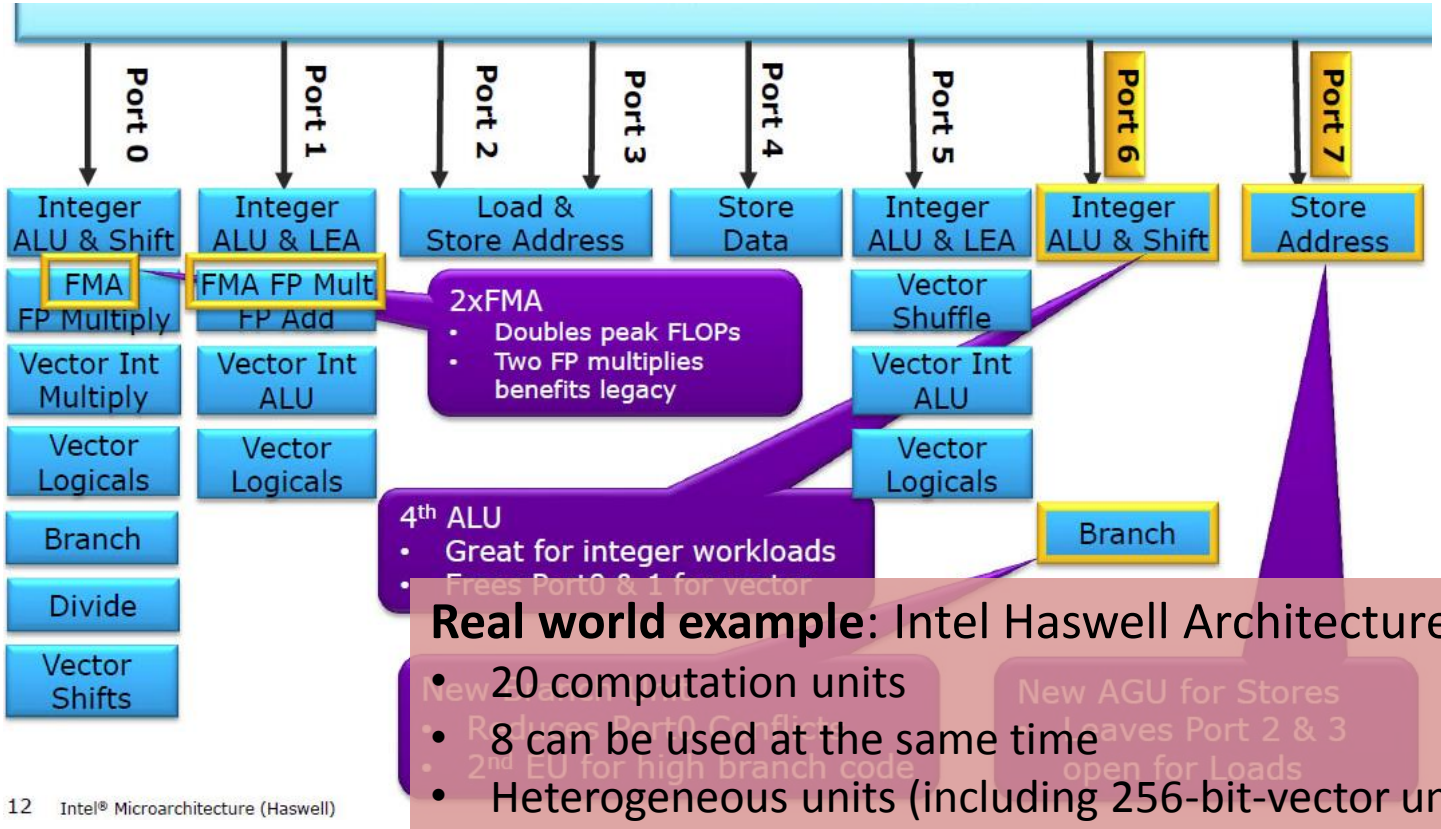
Software dependent

About the Hardware Peak IPC (1/2)

- It is the maximum number of instructions that a computing core can execute per cycle
- It depends on the amount of computation units in the core
- Schematic examples (see figures below)
  - Core 1: $IPC_{peak} = 1$
  - Core 2: $IPC_{peak} = 2$



About the Hardware Peak IPC (1/2)



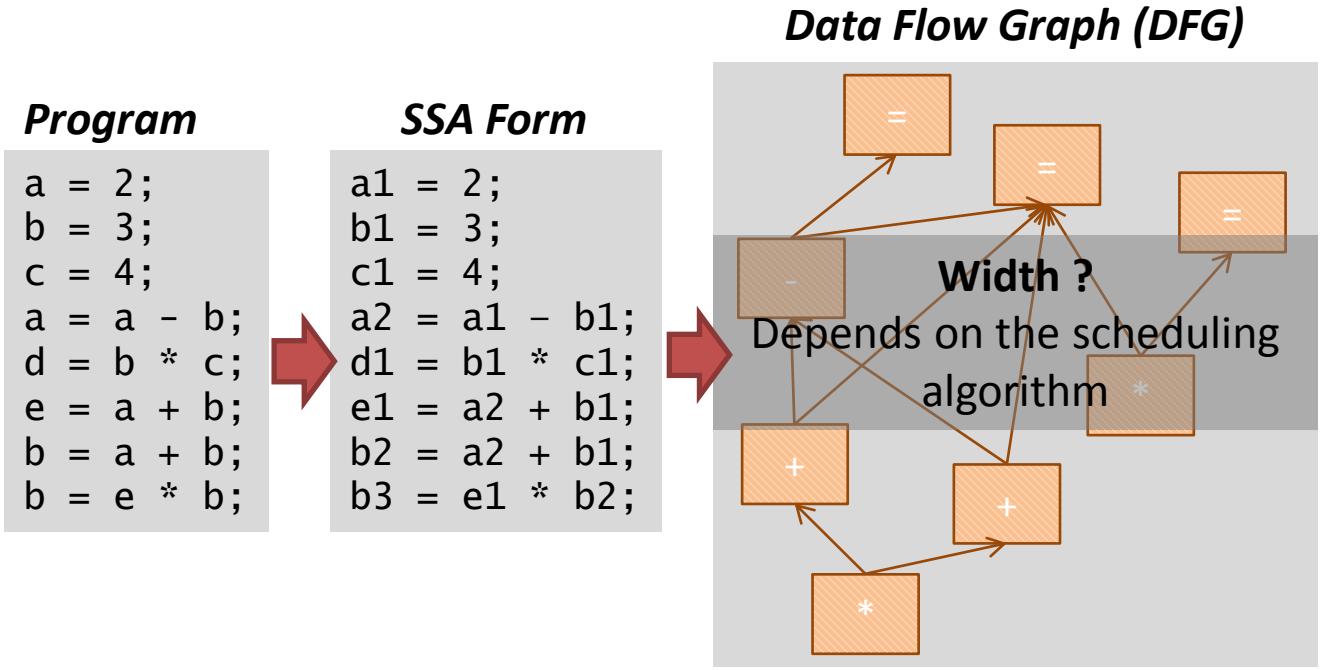
Source: *Intel Software Conference*

Real world example: Intel Haswell Architecture

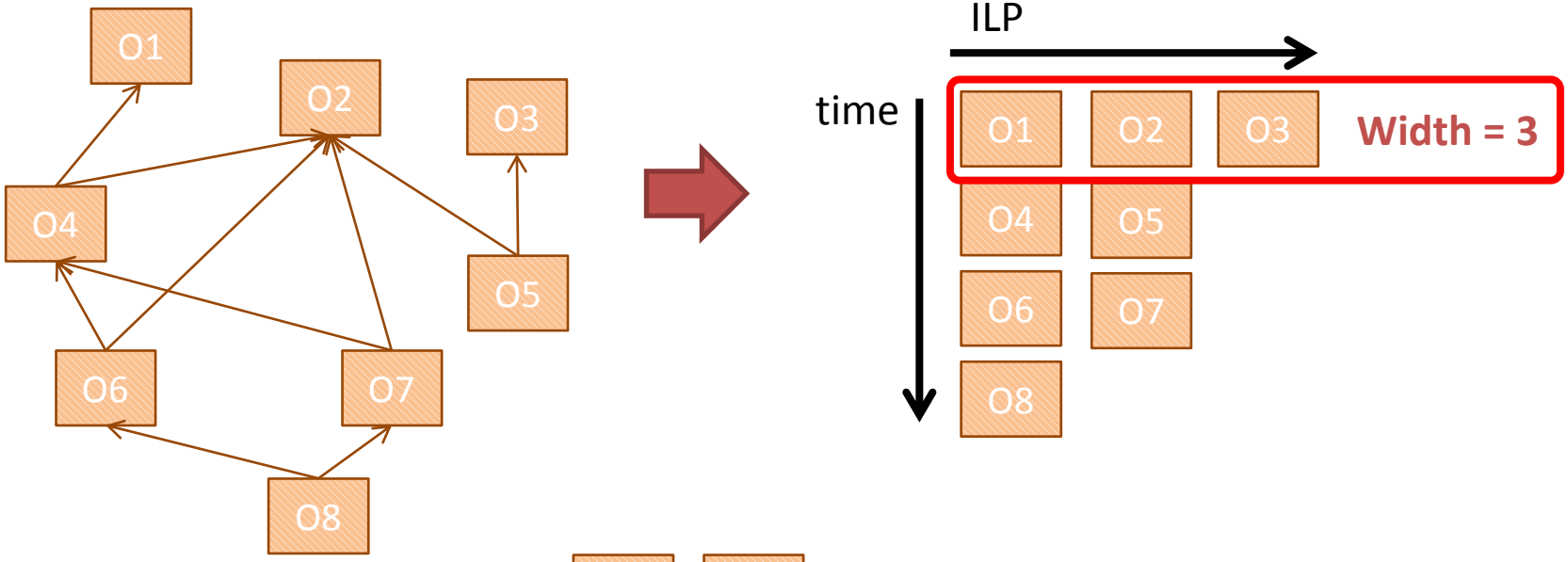
- 20 computation units
- 8 can be used at the same time
- Heterogeneous units (including 256-bit-vector units)
- The Peak IPC depends on the type of instruction
- It is too complex to calculate the Peak IPC, we need to approximate
- Peak IPC = 8 might be a reasonable approximation here (if we consider vector instructions as a single operation)

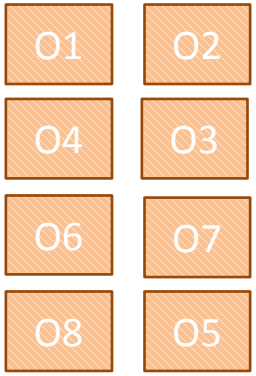
About the ILP (1/2)

- **Definition of ILP**
  - Instruction-Level-Parallelism
  - The maximum number of instructions that can be executed in parallel, as constrained by data dependencies
  - We also use the term **Data-Level-Parallelism**
  - It is a hardware independent metrics
  - The higher the ILP, the more we can expect to reduce the execution time
- **How to calculate it ?**
  1. Generate the data-flow graph of the program
  2. Calculate its width (depends on the scheduling algorithm)



About the ILP (2/2)
Example of ASAP Scheduling





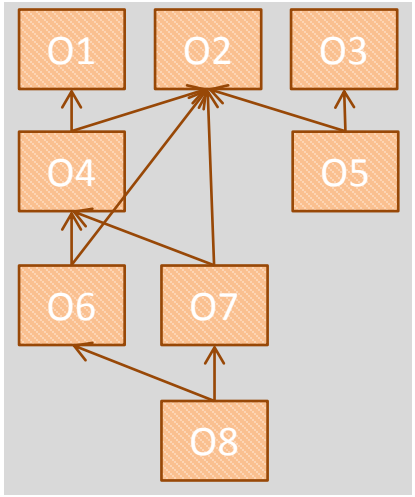
Note:
We can find a scheduling with width=2 and the same execution time for this program

Execution Example (1/2)
Machine with hwIPC = 2

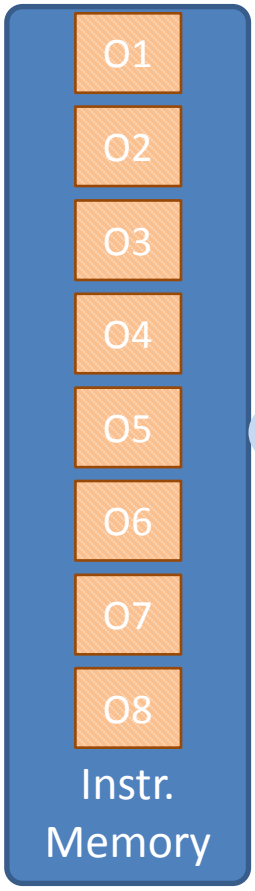
Source Code

```
O1 a1 = 2;
O2 b1 = 3;
O3 c1 = 4;
O4 a2 = a1 - b1;
O5 d1 = b1 * c1;
O6 e1 = a2 + b1;
O7 b2 = a2 + b1;
O8 b3 = e1 * b2;
```

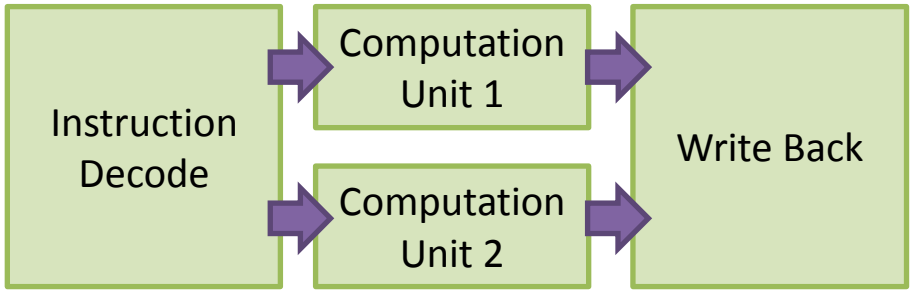
Data-Flow Graph



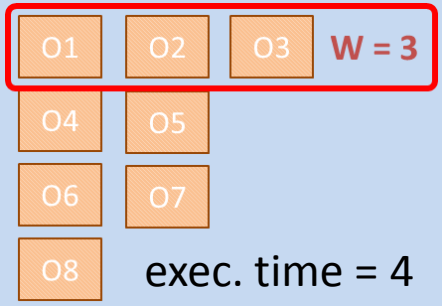
Program
(1 thread)



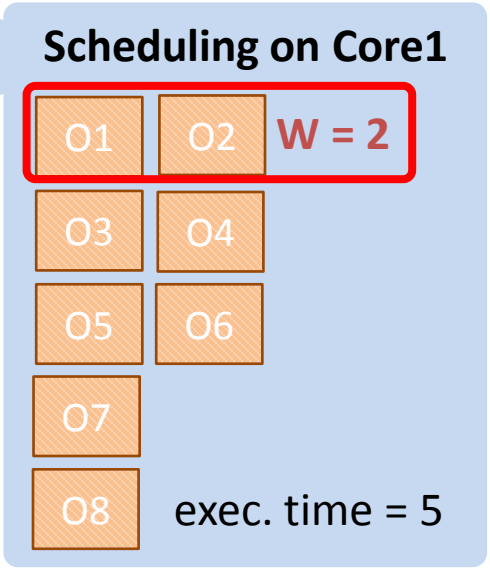
Core 1 (ASAP Scheduling, 2 CUs)



ASAP Scheduling

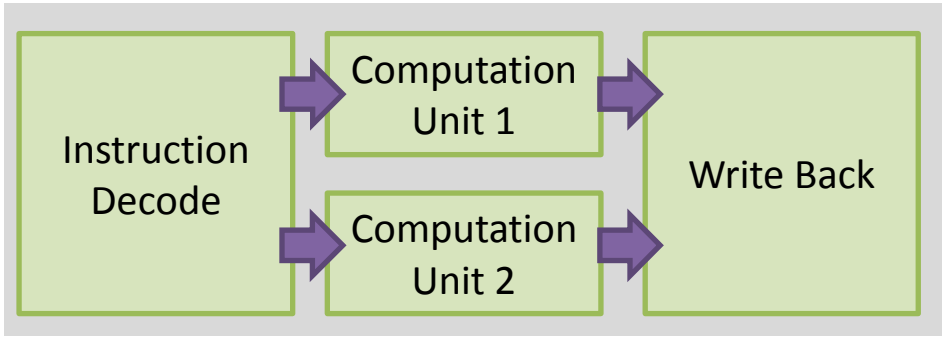


Scheduling on Core1

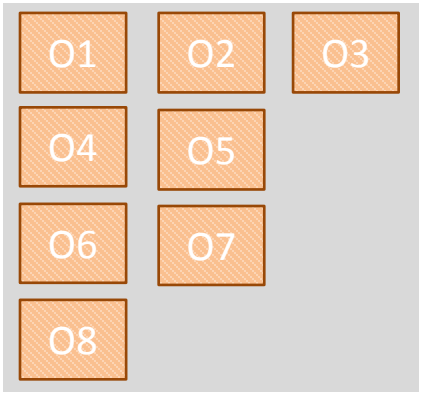


Execution Example (2/2)
Calculation of the IPC

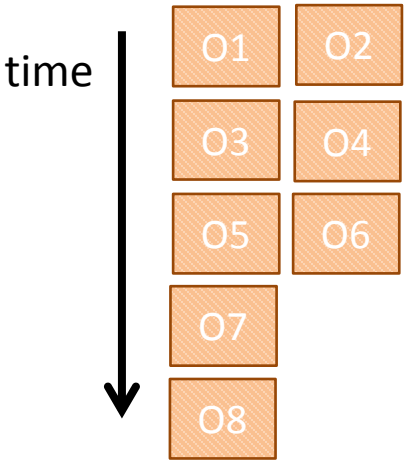
Core 1. hwIPC = 2



Program. ASAP width = 3



Execution on Core 1.



$$ILP = \frac{8\ operations}{5\ cycles} = 1.6 < 2$$

Conclusion on ILP / IPC

- The IPC determines single-thread efficiency
  - IPC = Instruction per Cycle
  - The higher the IPC, the faster
- The IPC is always lower than
  - The hardware Peak IPC
  - The software intrinsic ILP (Instruction-Level Parallelism)
- How to raise the IPC ?
  - Bad: The hardware IPC is fixed
  - **Good: The compiler can raise the software ILP**

Why we need vectors

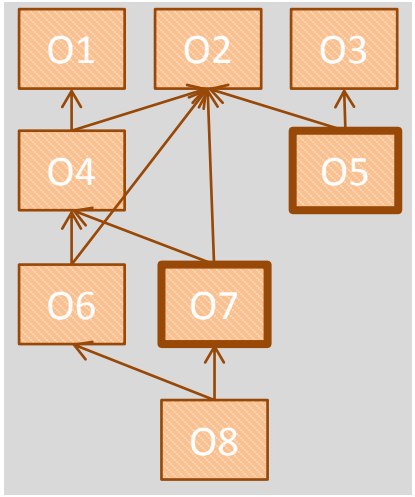
IPC AND OPTIMIZATION

About the Order of Instruction (1/2)

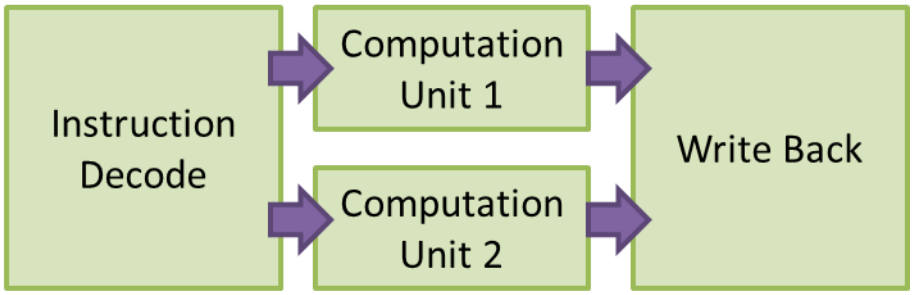
Source Code 1

```
o1 a1 = 2;
o2 b1 = 3;
o3 c1 = 4;
o4 a2 = a1 - b1;
o5 d1 = b1 * c1;
o6 e1 = a2 + b1;
o7 b2 = a2 + b1;
o8 b3 = e1 * b2;
```

Data-Flow Graph



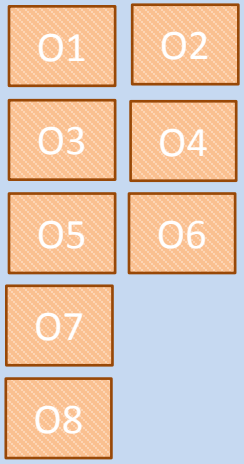
Core 1 (ASAP Scheduling, 2 CUs)



Source Code 2

```
o1 a1 = 2;
o2 b1 = 3;
o3 c1 = 4;
o4 a2 = a1 - b1;
o7 b2 = a2 + b1;
o6 e1 = a2 + b1;
o5 d1 = b1 * c1;
o8 b3 = e1 * b2;
```

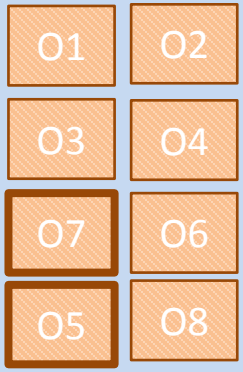
Scheduling on Core1



exec. time = 5

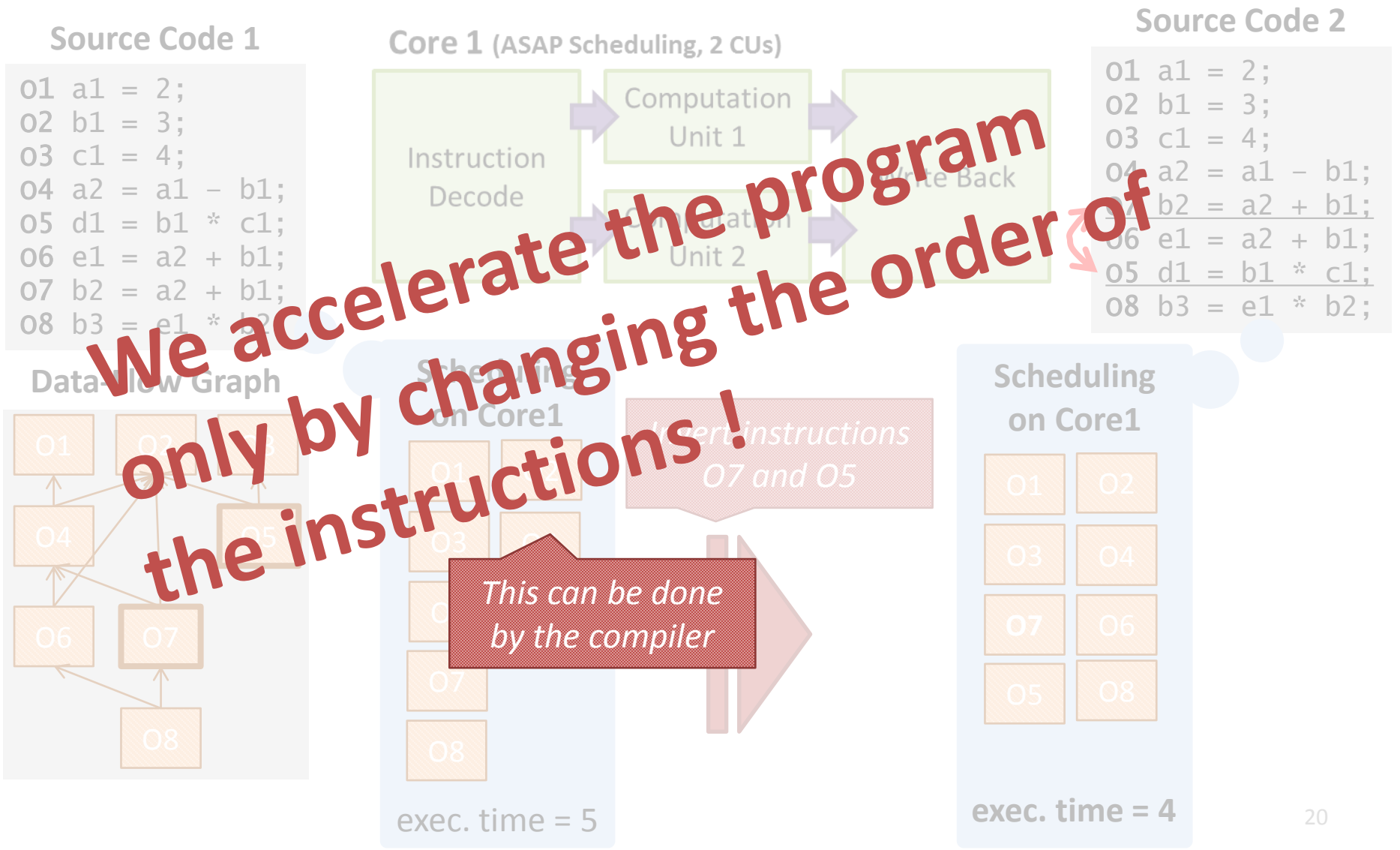
Invert instructions O7 and O5

Scheduling on Core1



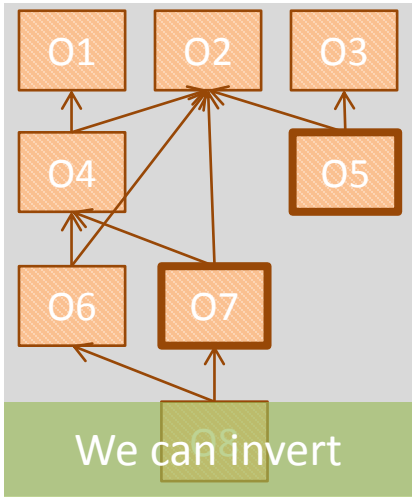
exec. time = 4

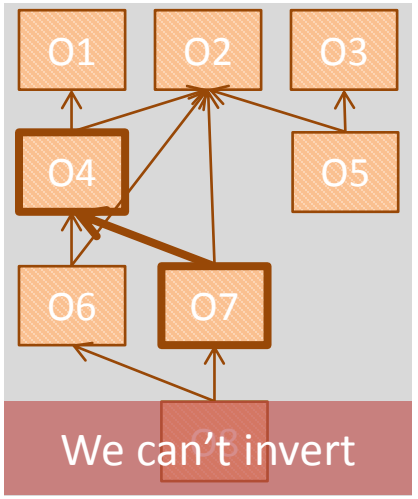
About the Order of Instruction (1/2)



About the Order of Instructions (2/2)

- We can only change the order of instructions that have **no data dependence**
  - A wrong order would break the program
  - This can be checked on the DFG (see figure at right)
- But anyway, we can't expect much speedup from such technique
  - Previous slide: only 20% faster
  - In practice, real improvements are of the same order (10~20%)
- **Other technique to improve IPC: use vector instructions**
  - We can expect speedups up to several times
  - Example of Haswell architecture: up to 8 times





About Vector Instructions

- Processors usually perform scalar operations
  - One operation per computation unit per cycle
  - The notions of operation and instruction are the same
  - **Example:** a = b +c
- But they can also perform vector operations
  - More than one identical operation per computation unit per cycle
  - Width of vectors = number of operation
  - One instruction now performs more than one operation
  - Limitation: all the operations should be the same
  - **Example:** a[1:8] = b[1:8] + c[1:8]

8 operations in 1 instruction !
The ILP is now 8 !

PROGRAM AND OPTIMIZE FOR VECTOR INSTRUCTIONS (SIMD)

Digression: Flynn's Taxonomy (1966)

	Single Instruction	Multiple Instruction
Single Data	**SISD** (Scalar Instructions)	**MISD** (Very Uncommon)
Multiple Data	**SIMD** (Vector Instructions)	**MIMD** (VLIW, superscalar)

SISD: 3 cycles

t=1 a1 = b1 + c1
t=2 a2 = b2 + c2
t=3 a3 = b3 * c3

SISD corresponds to a core with one computation unit

We can execute all the "+" at the same time (but not the "")*

SIMD: 2 cycles

t=1 [a1, a2] = [b1, b2] + [c1, c2]
t=2 a3 = b3 * c3

MIMD: 1 cycle

t=1 a1 = b1 + c1; a2 = b2 + c2; a3 = b3 * c3

We can do everything at the same time, providing we have enough parallel computation units.

Digression: Flynn's Taxonomy (1966)

	Single Instruction	Multiple Instruction
Single Data	SISD (Scalar Instructions)	MISD (Very Uncommon)
Multiple Data	**SIMD** (Vector Instructions)	MIMD (VLIW, superscalar)

SISD: 3 cycles

t=1 a1 = b1 + c1
t=2 a2 = b2 + c2
t=3 a3 = b3 * c3

We can execute all the "+" at the same time (but not the "")*

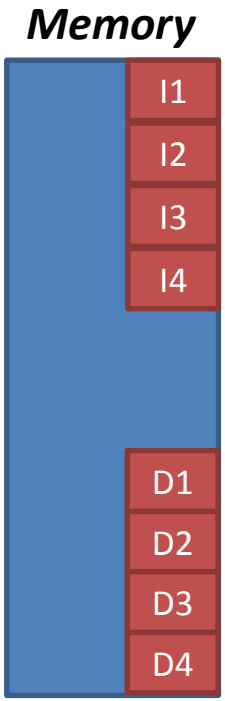
SIMD: 2 cycles

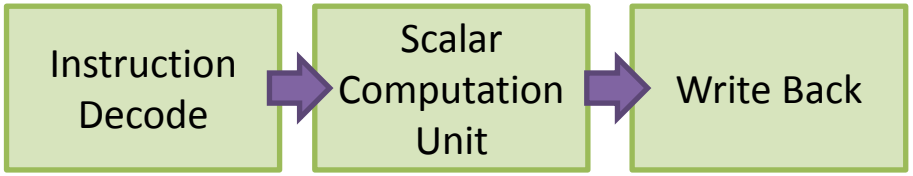
t=1 [a1, a2] = [b1, b2] + [c1, c2]
t=2 a3 = b3 * c3

MIMD: 1 cycle

t=1 a1 = b1 + c1; a2 = b2 + c2; a3 = b3 * c3

Vector vs. Scalar (1/2)
Scalar Instructions





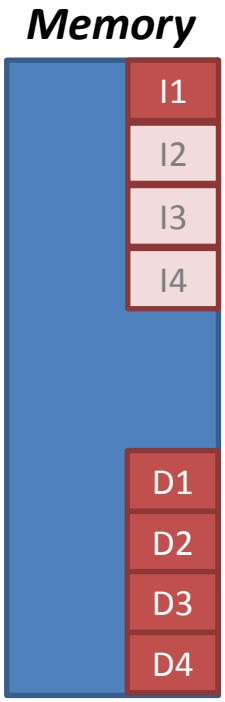
Equivalent Scalar Program

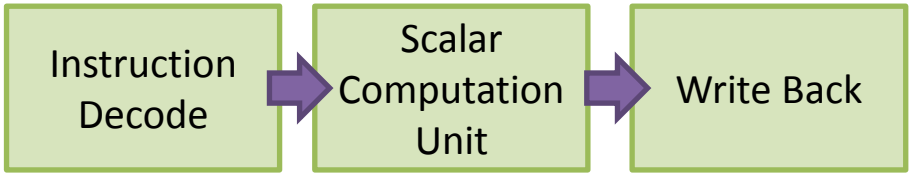
Process D1
Process D2
Process D3
Process D4

Where

- "Process" is an operation (e.g. addition)
- The "Ix" are identical instructions corresponding to each Process
- The "Di" are data (e.g. two numbers for an addition)

Vector vs. Scalar (1/2)
Scalar Instructions





Equivalent SIMD Program

Process D1,D2,D3,D4

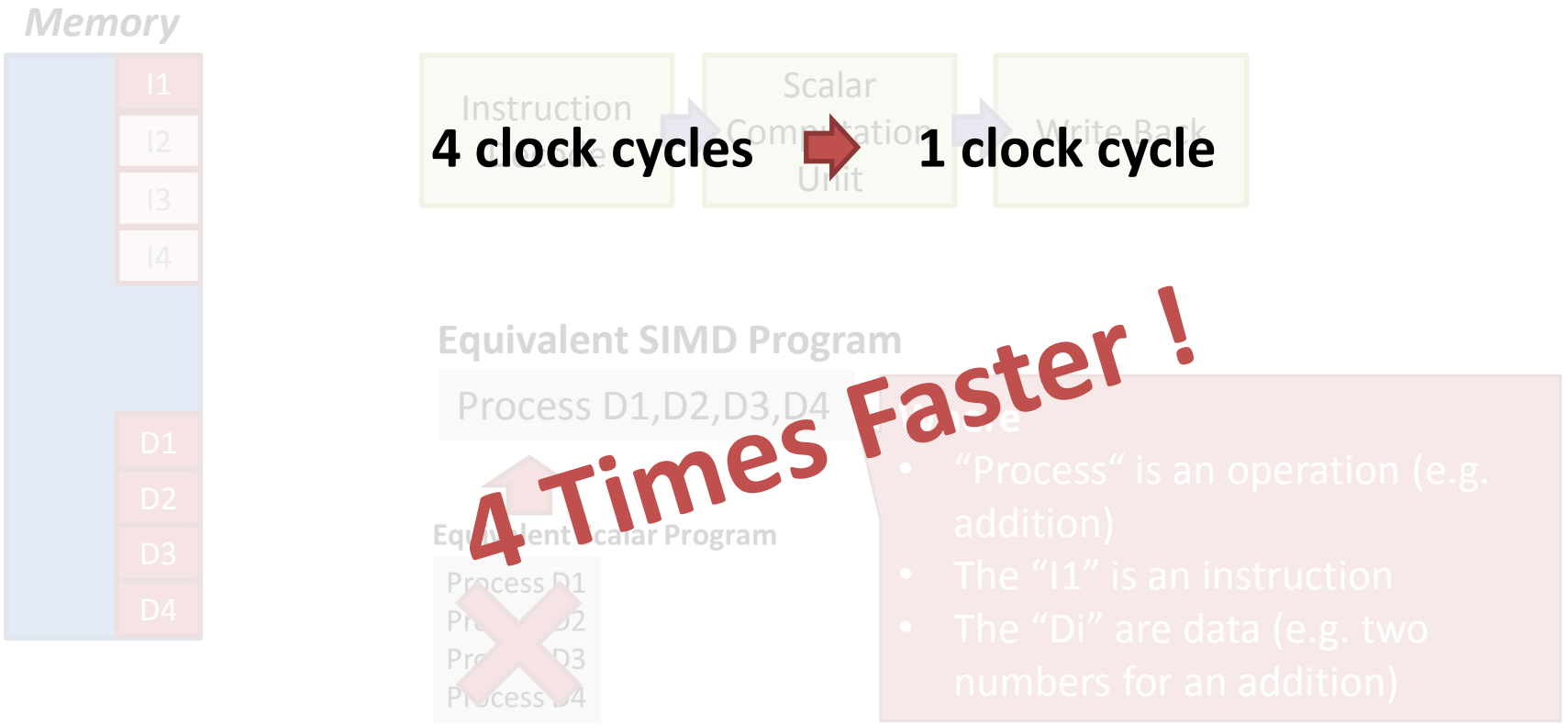
Equivalent Scalar Program

Process D1
Process D2
Process D3
Process D4

Where

- "Process" is an operation (e.g. addition)
- The "I1" is an instruction
- The "Di" are data (e.g. two numbers for an addition)

Vector vs. Scalar (1/2)
Scalar Instructions



How to Program for SIMD Instructions ? (1/3)

$$\begin{pmatrix} a_{0,0} & \cdots & a_{15,0} \\ \vdots & \ddots & \vdots \\ a_{0,15} & \cdots & a_{15,15} \end{pmatrix} = \begin{pmatrix} b_{0,0} & \cdots & b_{15,0} \\ \vdots & \ddots & \vdots \\ b_{0,15} & \cdots & b_{15,15} \end{pmatrix} + \begin{pmatrix} c_{0,0} & \cdots & c_{15,0} \\ \vdots & \ddots & \vdots \\ c_{0,15} & \cdots & c_{15,15} \end{pmatrix}$$

Scalar Program (C)

```
int a[16];
int b[16];
int c[16];
int i;
for (i=0;i<16;i++) {
  a[i] = b[i]+c[i]
}
```

SIMD Program (Pseudo Code)

```
int a[16];
int b[16];
int c[16];
int i;
a = b + c;
```

Yes, but No.
We want to write that, but C does not support such syntax.
(note: other languages like Fortran can)

How to Program for SIMD Instructions ? (2/3)

$$\begin{pmatrix} a_{0,0} & \cdots & a_{15,0} \\ \vdots & \ddots & \vdots \\ a_{0,15} & \cdots & a_{15,15} \end{pmatrix} = \begin{pmatrix} b_{0,0} & \cdots & b_{15,0} \\ \vdots & \ddots & \vdots \\ b_{0,15} & \cdots & b_{15,15} \end{pmatrix} + \begin{pmatrix} c_{0,0} & \cdots & c_{15,0} \\ \vdots & \ddots & \vdots \\ c_{0,15} & \cdots & c_{15,15} \end{pmatrix}$$

Scalar Program (C)

```
int a[16];
int b[16];
int c[16];
int i;
for (i=0;i<16;i++) {
  a[i] = b[i]
}
```

Solution 1:
Insert by hand

```
int a[16];
int b[16];
int c[16];
int i;
ADD16(a,b,c);
```

ADD16(.,.,.):

- Function call to a library provided by the processor vendor (e.g. Intel, ARM)
- Not in the C standard library
- Depend on the processor: the code is not portable anymore

How to Program for SIMD Instructions ? (2/3)

$$\begin{pmatrix} a_{0,0} & \cdots & a_{15,0} \\ \vdots & \ddots & \vdots \\ a_{0,15} & \cdots & a_{15,15} \end{pmatrix} = \begin{pmatrix} b_{0,0} & \cdots & b_{15,0} \\ \vdots & \ddots & \vdots \\ b_{0,15} & \cdots & b_{15,15} \end{pmatrix} + \begin{pmatrix} c_{0,0} & \cdots & c_{15,0} \\ \vdots & \ddots & \vdots \\ c_{0,15} & \cdots & c_{15,15} \end{pmatrix}$$

Scalar Program (C)

```
int a[16];
int b[16];
int c[16];
int i;
for (i=0;i<16;i++) {
  a[i] = b[i] + c[i];
}
```

Solution 1:
Insert by hand

```
int a[16];
int b[16];
int c[16];
int i;
ADD16(a,b,c);
```

ADD16(.,.,.):

- Function call to a lib… processor vendor (e.…
- Not in the C standard library
- Depend on the processor: the code is not portable anymore

Can we ask the compiler to do it for us ?

Benefits

- Save time for the programmer
- Keep the code portable (you can compile and execute the same C program in Intel and ARM processor for example)

How to Program for SIMD Instructions ? (2/3)

$$\begin{pmatrix} a_{0,0} & \cdots & a_{15} \\ \vdots & \ddots & \vdots \\ a_{0,15} & \cdots & a_{15,} \end{pmatrix} \quad \begin{pmatrix} c_{0,0} & \cdots & c_{15,0} \\ \vdots & \ddots & \vdots \\ c_{0,15} & \cdots & c_{15,15} \end{pmatrix}$$

Can we ... to do it

piler

nefits

Save time for the programmer

Keep the code portable (you can compile and execute the same C program in Intel and ARM processor for example)

Scalar Program ... *1:* ... *hand*

```
int a[16];
int b[16];
int c[16];
int i;
for (i=0;i<
  a[i] = b
}
```

processor vendor (e.

- Not in the C standard library
- Depend on the processor: the code is not portable anymore

How to Program for SIMD Instructions ? (3/4)

$$\begin{pmatrix} a_{0,0} & \cdots & a_{15,0} \\ \vdots & \ddots & \vdots \\ a_{0,15} & \cdots & a_{15,15} \end{pmatrix} = \begin{pmatrix} b_{0,0} & \cdots & b_{15,0} \\ \vdots & \ddots & \vdots \\ b_{0,15} & \cdots & b_{15,15} \end{pmatrix} + \begin{pmatrix} c_{0,0} & \cdots & c_{15,0} \\ \vdots & \ddots & \vdots \\ c_{0,15} & \cdots & c_{15,15} \end{pmatrix}$$

Scalar Program (C)

```
int a[16];
int b[16];
int c[16];
int i;
for (i=0;i<16;i++) {
  a[i] = b[i]+c[i];
}
```

Solution 2: Unroll to increase ILP and trust Basic Block Vectorization

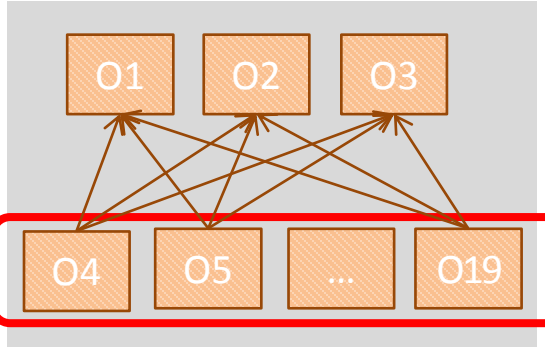
```
int a[16];
int b[16];
int c[16];
a[0] = b[0] + c[0];
a[1] = b[1] + c[1];
…
a[15] = b[15] + c[15];
```

Automatic Basic-Block Vectorization

Unrolled Program

```
O1  int a[16];
O2  int b[16];
O3  int c[16];
O4  a[0] = b[0] + c[0];
O5  a[1] = b[1] + c[1];
O6 ~ O18 …
O19 a[15] = b[15] + c[15];
```

Data Flow Graph



Pseudo-Code

```
int a[16];
int b[16];
int c[16];
ADD16(a,b,c)
```

16 Identical Operations

(Loop Unrolling)

Loop unrolling might be done:
- by hand (very tedious)
- by the compiler (automatically)

Loop unrolling is not mandatory but it increases the success of the transformation (by increasing the ILP in the body of the loop).

Automatic Basic Block Vectorization
(from the DFG)

Identical operations with no dependency between each other inside a basic block are "**vectorized**".

How to Program for SIMD Instructions ? (4/4)

$$\begin{pmatrix} a_{0,0} & \cdots & a_{15,0} \\ \vdots & \ddots & \vdots \\ a_{0,15} & \cdots & a_{15,15} \end{pmatrix} = \begin{pmatrix} b_{0,0} & \cdots & b_{15,0} \\ \vdots & \ddots & \vdots \\ b_{0,15} & \cdots & b_{15,15} \end{pmatrix} + \begin{pmatrix} c_{0,0} & \cdots & c_{15,0} \\ \vdots & \ddots & \vdots \\ c_{0,15} & \cdots & c_{15,15} \end{pmatrix}$$

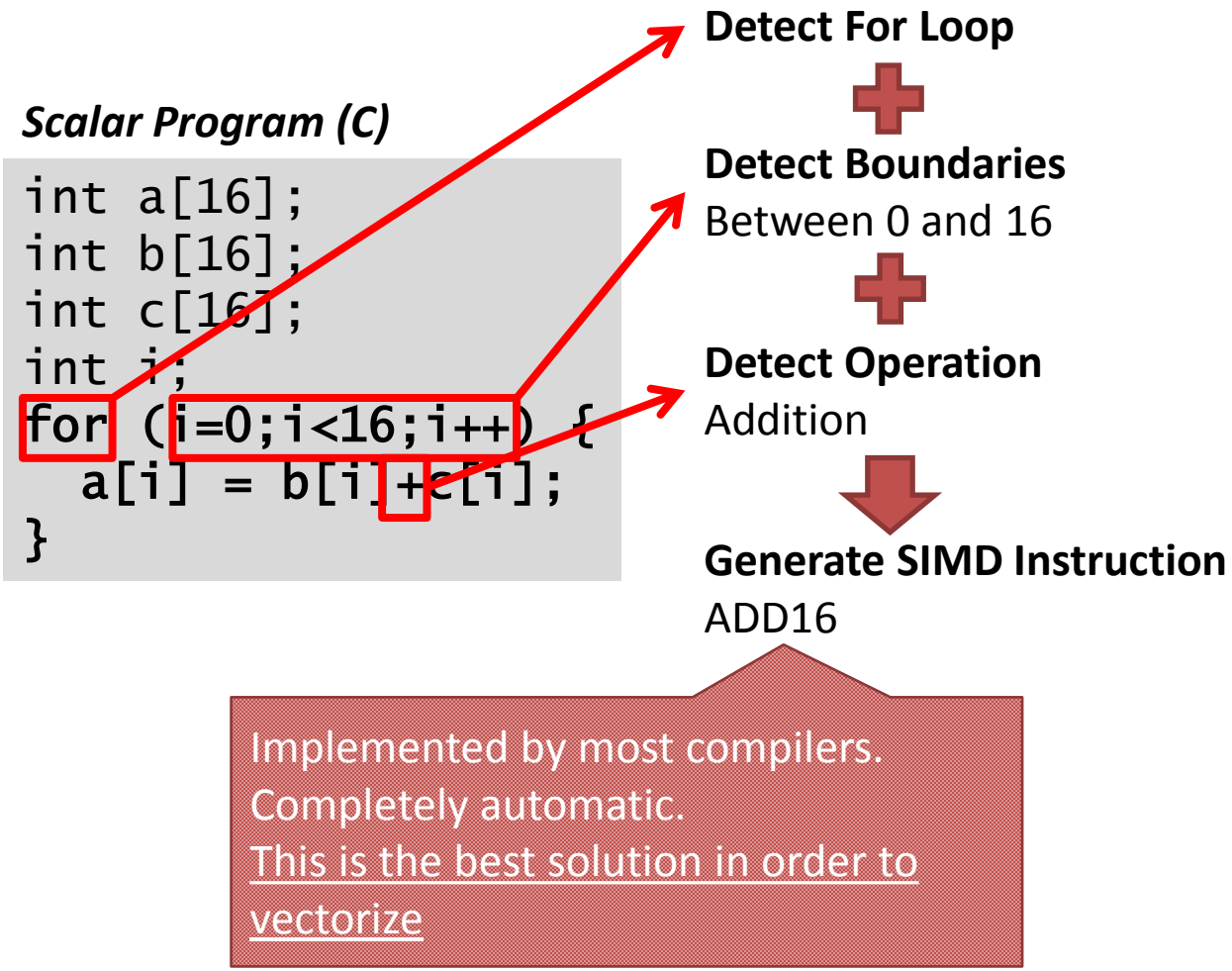
Scalar Program (C)

```
int a[16];
int b[16];
int c[16];
int i;
for (i=0;i<16;i++) {
  a[i] = b[i]+c[i];
}
```

Solution 3:

Trust Loop-Vectorization

Automatic Loop Vectorization



Conclusion on Programing for Vector Instructions

- Vector instructions
  - SIMD programing model from Flynn's taxonomy
  - Single Instruction Multiple Data
- Easy and cheap way to accelerate programs
  - Example of Haswell architecture: 256 bits vectors, that is, 8 simultaneous operations on 32-bit data (words or floating point)
- Usually generated by the compiler
  - Automatic basic block vectorization
  - **Automatic loop vectorization**

You thought it was easy huh ???

AUTOMATIC VECTORIZATION: ISSUES AND CHALLENGES

Why we can't always Vectorize

- Vectorization is very powerful
- However, it is not always possible to vectorize
- **First situation:**
  - The code is vectorizable, but the compiler fails to vectorize it
  - Solution 1: insert vector function calls by hand
  - Solution 2: modify the code
- **Second situation:**
  - The algorithm is not vectorizable
  - Nothing can be done
- **Third situation:**
  - The code can be vectorized, but it is slower with vectors

First Challenge

The code should be "Simple"

First or Second Situation: the code is sometimes vectorizable if modified

```
for (i=0;i<16;i++) {
  a[i] = b[i]+c[i];
}
```

OK.
Example of previous section

```
int function(int M) {
  for (i=0;i<M;i++) {
    a[i] = b[i]+c[i];
  }
}
```

Dangerous.
The compiler needs to know the value of M.
Some compilers still generate vectors by adding conditions before the loop
Can be solved by the compiler with constant propagation and function inlining.

```
int M;
for (i=0;i<cos(M);i++) {
  a[i] = b[i]+c[i];
}
```

Bad.
The boundaries of the for loop are complex.
Most compilers will fail to vectorize this code.

First Challenge

The code should be "Simple"

```
for (i=0;i<16;i++) {
  a[i] = b[i]+c[i]
}
```

OK.
Example of previous section

```
int function(int M) {
  for (i=0;i<M;i++) {
    a[i] = b[i]+c[i]
  }
}
```

Dangerous
The c[illegible]eeds to know the value of M.
Some compilers still generate vectors by adding conditions before the loop
Can be solved by the compiler with constant

```
int M;
for (i=0;i<M;i++) {
  a[i] = b[i]+c[i]
}
```

[illegible]nplex.
[illegible]is
code.

There are many more examples.

It depends on the compiler.
Intel Compiler is the smartest.
GCC is not too bad either.
LLVM still lags behind.

Second Challenge

The loop should not contain complex operations

First Situation: the code is vectorizable if modified

```
for (i=0;i<16;i++) {
  a[i] = b[i]+c[i];
  NON VECTORIZABLE CODE
}
```

Bad.
It contains non-vectorizable code (e.g. function call)

```
for (i=0;i<16;i++) {
  a[i] = b[i]+c[i];
}
for (i=0;i<16;i++) {
  NON VECTORIZABLE CODE
}
```

Solution.
Cut the loop in two loops to isolate the non-vectorizable code.
This is called **loop fission**.
The compiler usually don't do it automatically.

Third Challenge

The loop should not contain branches (1/2)

First or Second Situation: the code is sometimes vectorizable if modified

```
bool c;
for (i=0;i<16;i++) {
  if(c) a[i] = b[i]+c[i];
  else  a[i] = b[i]-c[i];
}
```

Bad.
The loop body contains a "**if**" statement.
"If" statements are a special example of non-vectorizable code.

```
if(c) {
  for (i=0;i<16;i++) {
    a[i] = b[i]+c[i];
  }
}
else {
  for (i=0;i<16;i++) {
    a[i] = b[i]-c[i];
  }
}
```

Solution.
Put the branch outside the loop.
This is called loop unswitching.
The compiler often does it automatically.
It is not possible to do if the condition depends on loop variables (see next slide)

Fourth Challenge

The loop should not contain branches (2/2)

First or Second Situation: the code is sometimes vectorizable if modified

```
for (i=0;i<16;i++) {
  if(i<8) a[i] = b[i]+c[i];
  else    a[i] = b[i]-c[i];
}
```

Bad.
It contains non-vectorizable code (e.g. function call)

```
for (i=0;i<8;i++) {
  a[i] = b[i]+c[i];
}
for (i=8;i<16;i++) {
  a[i] = b[i]-c[i];
}
```

Solution.
Cut the loop in two loops (one per branch).
This is another kind of **loop fission**.
The compiler never does it automatically.

This is however an easy example; such transformation is often impossible to apply.

```
for (i=0;i<16;i++) {
  if(i*4 % 8 == 0) a[i] = b[i]+c[i];
  else             a[i] = b[i]-c[i];
}
```

Fifth Challenge

Loop-carried Dependencies

```
for (i=1;i<16;i++) {
  a[i] = a[i-1]+c[i];
}
```

Bad (second situation)
The calculation depends on the result of a previous loop iteration, therefore we can't vectorize

```
for (i=0;i<15;i++) {
  a[i] = a[i+1]+c[i];
}
```

Dangerous (first situation).
The calculation does not depend on the result of other calculation, but it is similar to such codes.
Many compilers will fail to vectorize this code.

More on Next Slides…

Loop-Carried Dependencies (1/3)

Original Program

```
for (i=1;i<5;i++) {
  a[i] = a[i-1]+c[i];
}
```

Correct Result

a=[1,3,5,7,9]

Unrolled Program

```
a[1] = a[0]+c[1];
a[2] = a[1]+c[2];
a[3] = a[2]+c[3];
a[4] = a[3]+c[4];
```

State of the Memory

a=[1,1,1,1,1], c=[2,2,2,2,2]
a=[1,3,1,1,1]
a=[1,3,5,1,1]
a=[1,3,5,7,1]
a=[1,3,5,7,9]

Vectorized Program

```
a[1:4] = a[0:3]+c[1:4];
```

State of the Memory

a=[1,1,1,1,1], c=[2,2,2,2,2]
a=[1,3,3,3,3]

The result is wrong !

Loop-Carried Dependencies (1/3)

Original Program

```
for (i=1;i<5;i++) {
  a[i] = a[i-1]+c[i];
}
```

Correct Result

a=[1,3,5,7,9]

The vectorized program does not return the correct result.

This is because the calculation are not performed in the correct order.

Such program cannot be vectorized

(second situation)

Unrolled Program

```
a[1] = a[0]+c[1];
a[2] = a[1]+c[2];
a[3] = a[2]+c[3];
a[4] = a[3]+c[4];
```

State of the Memory

a=[1,1,1,1,1], c=[2,2,2,2,2]
a=[1,3,1,1,1]
a=[1,3,5,1,1]
a=[1,3,5,7,1]
a=[1,3,5,7,9]

Vectorized Program

```
a[1:4] = a[0:3]+c[1:4];
```

State of the Memory

a=[1,1,1,1,1], c=[2,2,2,2,2]
a=[1,3,3,3,3]

Loop-Carried Dependencies (2/3)

We have a “+” instead of a “-” before

Original Program

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

Correct Result

a=[3,3,3,3,1]

Unrolled Program

```
a[0] = a[1]+c[0];
a[1] = a[2]+c[1];
a[2] = a[3]+c[2];
a[3] = a[4]+c[3];
```

State of the Memory

```
a=[1,1,1,1,1], c=[2,2,2,2,2]
a=[3,1,1,1,1]
a=[3,3,1,1,1]
a=[3,3,3,1,1]
a=[3,3,3,3,1]
```

Vectorized Program

```
a[0:3] = a[1:4]+c[0:3];
```

State of the Memory

```
a=[1,1,1,1,1], c=[2,2,2,2,2]
a=[3,3,3,3,1]
```

The result is correct !

Loop-Carried Dependencies (3/3)

The Loop-carried-Dependency Graph (1/2)

- Expresses the data dependencies between iteration of a loop
  - Same kind of dependency as for the data-flow graph: RAW (Read After Write), WAR (Write After Read), RAR (Read After Read), WAW (Write After Read)
  - Similar to the DFG of the unrolled loop
- Example with some loops of the previous slides:

It is always OK to ignore it

RAR	RAW
WAR	WAW

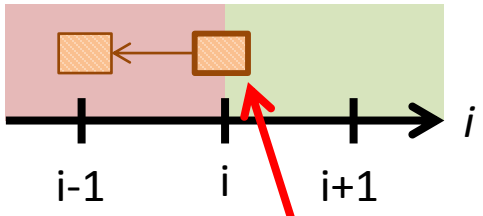
WAW does not prevent vectorization by itself, but the ordering of SIMD instruction is then important. **(anyway WAW is very rare)**

Legend
Cannot vectorize
Can vectorize
Dangerous

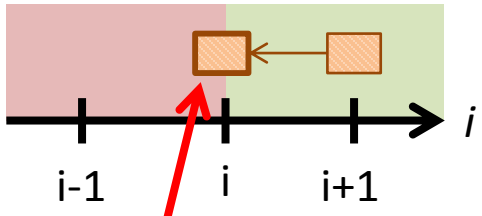
```
for (i=0;i<16;i++) {
  a[i] = a[i-1]+c[i];
}
```

```
for (i=0;i<16;i++) {
  a[i] = a[i+1]+c[i];
}
```

RAW Dependency
(Read After Write)



WAR Dependency
(Write After Read)



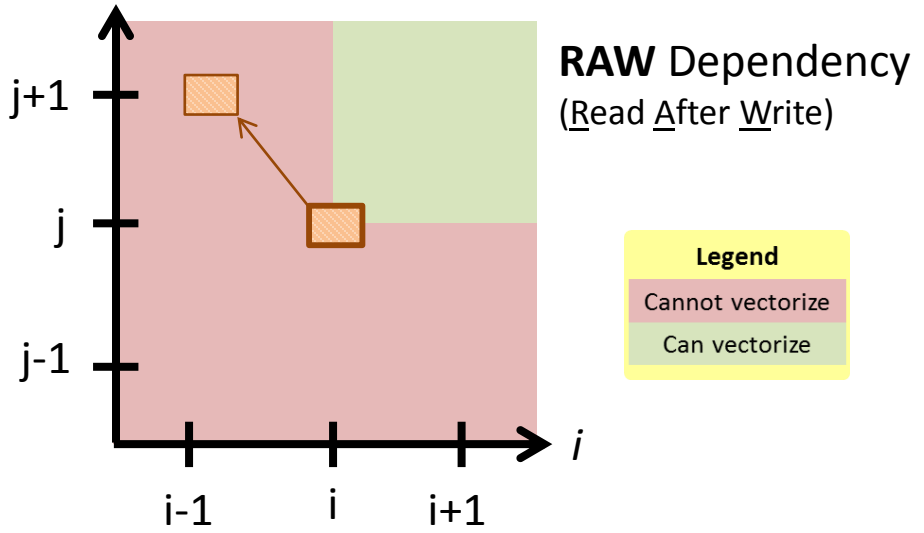
The reference node is the node that reads

Loop-Carried Dependencies (3/3)

The Loop-carried-Dependency Graph (2/2)

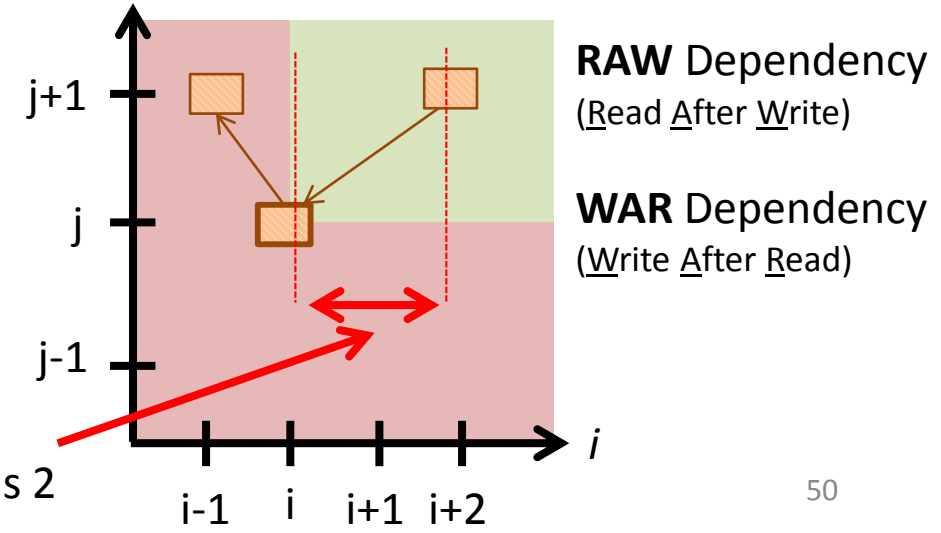
Example in 2D

```
for (i=0;i<16;i++) {
  for (j=0; j<16; j++)
    a[i][j] = a[i-1][j+1]+c[i];
  }
}
```



A slightly more complex example

```
for (i=0;i<16;i++) {
  for (j=0; j<16; j++)
    a[i][j]=
        a[i-1][j+1]+a[i+2][j+1];
  }
}
```



Conclusion on Vectorization Challenges

- Vectorization is done automatically by the compiler
  - Automatic basic block vectorization
  - Loop vectorization
  - Note: another powerful technique exists: software pipelining
- But sometimes, even though we can vectorize, the compiler fails to vectorize because:
  - The code is too “complex” (the meaning of “complex” depends on the compiler)
  - The code contains non-vectorizable code
  - The code contains branches
- In many situations, the code can be vectorizable by modifying the code by hand
- However, some code is not vectorizable
  - The code contains branches that depends on value calculated inside the loop
  - The code contains WAR loop-carried dependency
- Remains a third situation:
  - The code is vectorizable, but run slower with SIMD instructions
  - **See next section…**

VECTORS AND MEMORY ACCESSES

The Memory Wall: Slide form Prof. Inoue

依然として聳え立つ3つの壁

シングル　マルチ　メニー

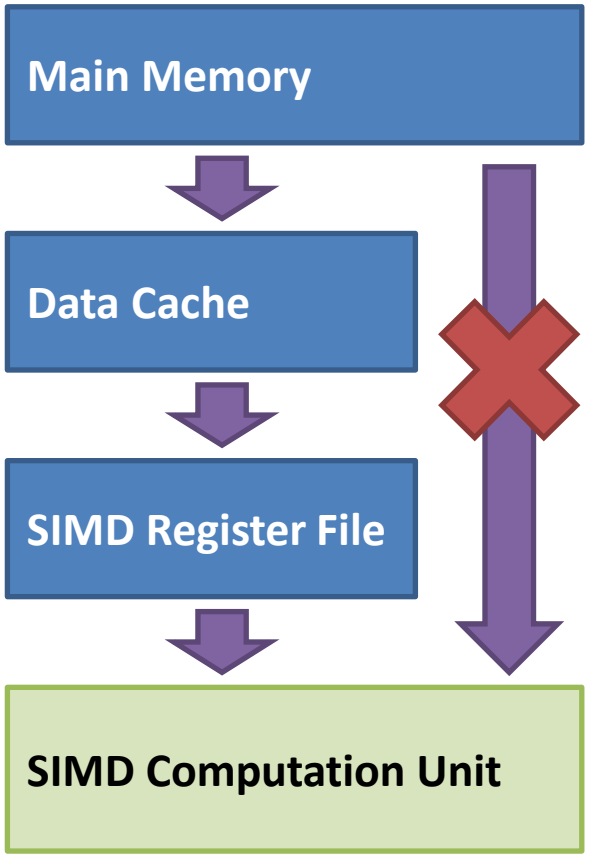
First Wall: the ILP Wall
Programs often don't exhibit high ILP.
Can be partially addressed using SIMD

Second Wall: the frequency Wall
We can't raise the frequency because it consumes too much power.
No solution (see slide 5)

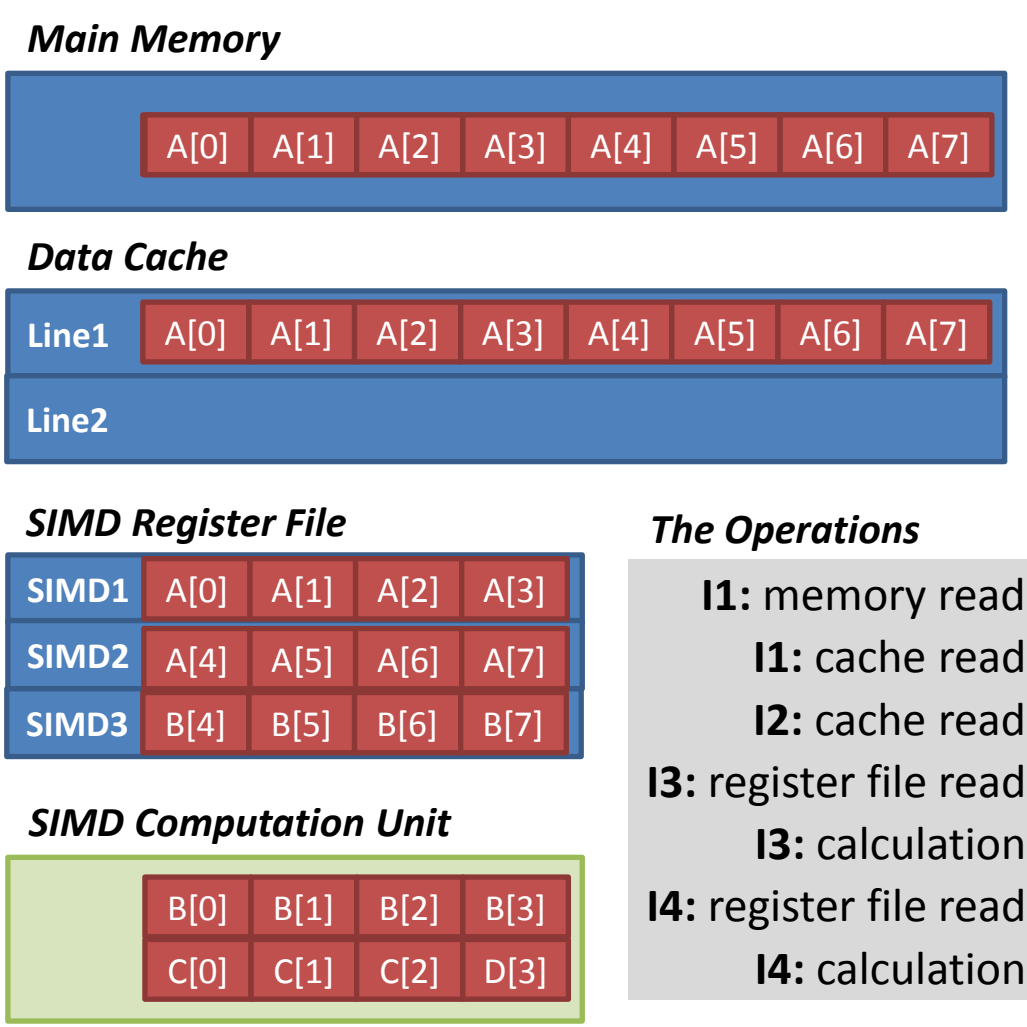
Third Wall: the memory Wall
Cores cannot access data as fast as they compute on them.
Also happens with SIMD!

Break the Memory Wall: The Memory Hierarchy (1/2)

- The SIMD computation units do not access the main memory directly
  - Its latency is too high (>100 cycles)
- They operate on a dedicated **register file**
  - Temporal locality: accelerate neighbor calculations on the same data
  - Example of Intel Haswell: 16 registers of 256 bits each (=4Kb)
- They use the same **data caches** as the scalar computation units
  - Temporal locality: accelerate neighbor calculations on the same data
  - Spatial locality: accelerate neighbor calculations on neighbor data
  - Example of Intel Haswell: 32Kb (Level 1) + 256Kb (Level 2) per core
- Point of view of the ISA
  - Data are explicitly loaded from the main memory to the SIMD register file
  - The cache is not visible to the ISA



Break the Memory Wall: The Memory Hierarchy (2/2)



The Program

I1 Load A[0:3] to register file SIMD1

I2 Load A[4:7] to register file SIMD2

I3 Calculate SIMD3 = SIMD1 + SIMD2

I4 Calculate SIMD4 = SIMD3 + SIMD2

The Operations

I1: memory read

I1: cache read

I2: cache read

I3: register file read

I3: calculation

I4: register file read

I4: calculation

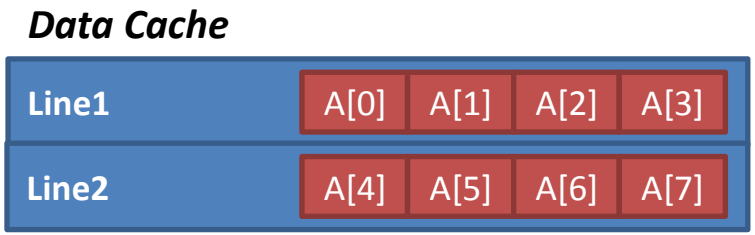
We read a whole cache line instead of only the data (spatial locality of cache)

The second instruction does not need to access the memory (spatial locality of cache)

The second calculation does not need to read the memory or the cache (temporal locality of register file)

Limitations of the Memory Hierarchy Aligned Accesses (1/3)

- Most SIMD core architectures only allow to read vectors from cache aligned with cache lines
  - The vector should start at the beginning of a cache line
  - Example: Fujitsu SPARC64 XII fx (K Computer)
- Other work with non-aligned data, but slower
  - Read non-aligned data requires many cycles
  - Example: Intel Haswell
- The same limitation exists when data is read from main memory



Aligned Access

```
SIMD1 = A[0:3]
```

1 clock cycle

Unaligned Access

```
SIMD1 = A[2:5]
```

A least 3 clock cycles:
cycle 1: read A[2:3]
cycle 2: read A[4:5]
cycle 3: concatenate A[2:3] and A[4:5] in SIMD1

Limitations of the Memory Hierarchy Aligned Accesses (1/3)

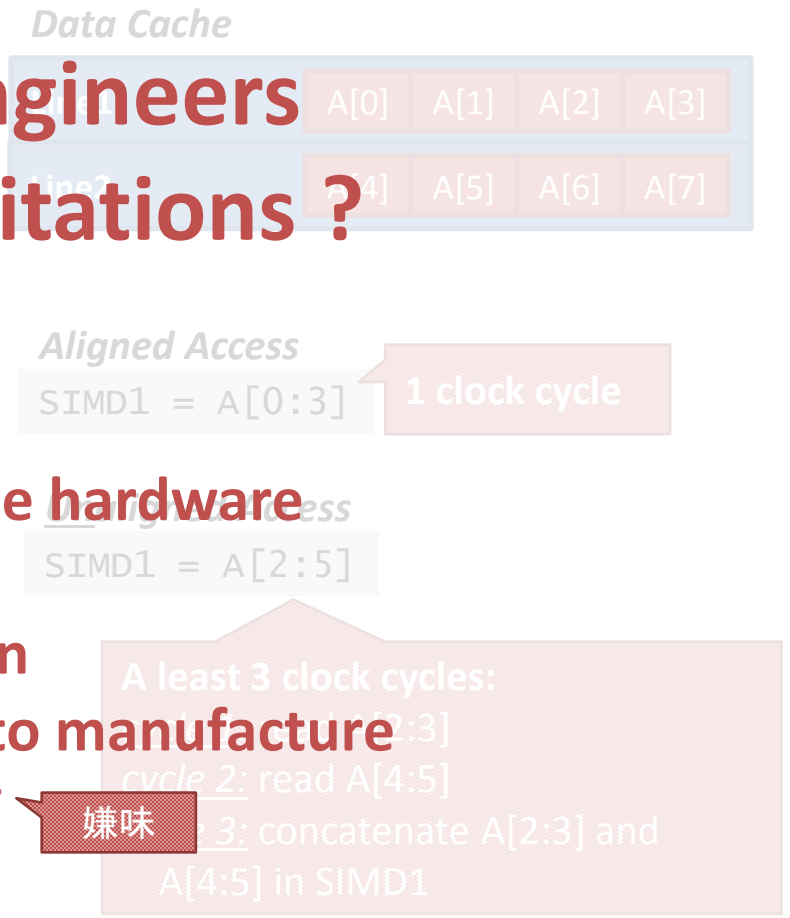
- Most SIMD core architectures only allow to read vectors from cache aligned with cache lines
  - The vector should start at the beginning of a cache line
  - Example: Fujitsu SPARC64 XII fx (K Computer)
- Other work with non-aligned data, but slower
  - Read non-aligned data requires many cycles
  - Example: Intel Haswell
- The same limitation exists when data is read from main memory

Why hardware engineers included such limitations ?

Because without them the hardware would have been:

- **too expensive to design**
- **too hard (impossible) to manufacture**
- **… and too easy to program for**

嫌味



Limitations of the Memory Hierarchy Aligned Accesses (2/3)

Example 1
We consider a cache line of **128 bits** (4 integers)

```
int my_function(){
  int a[32];
  int i;
  for (i=0; i<32; i++) a[i]++;
}
```

This should be slow
"a" is in the stack, which is likely not to be aligned

```
int my_function(){
  static int a[32];
  int i;
  for (i=0; i<32; i++) a[i]++;
}
```

This may be slow
"a" is not in the stack, but we don't know if it will be aligned or not
(note: e can also declare "a" global)

This will be fast (loop vectorization)
We force alignment with the attribute "aligned" (gcc only, other compilers may use different keywords)

Solution: "align" gcc attribute

```
int my_function(){
  static int a[32] __attribute__((aligned(0x1000)));
  int i;
  for (i=0; i<32; i++) a[i]++;
}
```

Limitations of the Memory Hierarchy
Aligned Accesses (3/3)

Example 2
We still consider a cache line of **128 bits** (4 integers)

This will be slow
The address of "a[1][0]" is not aligned !

```
int my_function(){
  static int a[32][10] __attribute__((aligned(0x1000)));
  int i,j;
  for (i=0; i<10; i++)
    for (j=0; j<32; i++)
      a[i][j]++;
}
```

Note:
&a[0][0] = 0 % 128 bits
&a[1][0] = **64** % 128 bits
&a[2][0] = 0 % 128 bits
&a[3][0] = **64** % 128 bits
(it should be 0 everywhere)

This will be fast
Now the address of "a[1][0]" is aligned.

Solution: array padding

```
int my_function(){
  static int a[32][12] __attribute__((aligned(0x1000)));
  int i,j;
  for (i=0; i<10; i++)
    for (j=0; j<32; i++)
      a[i][j]++;
}
```

Note:
&a[0][0] = 0 % 128 bits
&a[1][0] = 0 % 128 bits
&a[2][0] = 0 % 128 bits
&a[3][0] = 0 % 128 bits

Today's Conclusion

- Compiler can accelerate single-thread performance by raising the ILP of programs
  - Instruction Level Parallelism
- The most efficient method to do so is to use vector operations
  - Also called SIMD instructions (Flynn's taxonomy)
  - Only possible if the target core architecture supports it
- The compiler is able to generate SIMD instructions from normal C code
  - Two techniques today: basic block vectorization and loop vectorization
  - Only if the algorithm allows it
  - Sometimes we need to change to C program so that the compiler can better understand it and generate vectors
- Still, even with SIMD we hit the memory wall
  - We cannot access the memory as fast as we do SIMD calculation
  - Modern architecture use cache and register files
  - But those have many limitations that we need to understand
- **Not everything has been said**
  - Another automatic vectorization method: loop pipelining
  - Another restriction to automatic vectorization: memory aliasing (you may want to check the keyword "restrict" of C)

Any questions ?

THANK YOU VERY MUCH