Introduction to Optimizing Compilers

Antoine Trouvé
アントワン　トルヴェ
2014/06/23

まず、自己紹介

名　姓

Antoine | Trouvé
アントワン | トルヴェ

2006年
フランスのボルドー大学で修士取得
（「ENSEIRB」グランゼコール）

2007年~2014年
九州先端科学研究所（ももち浜＠福岡市）
研究員

2011年
九州大学で博士取得

現在
九州大学の助教

Contents of the Presentation

Introduction

入門

to Optimizing Compilers

最適化
コンパイラ

Today's Objective

Accelerate computer programs

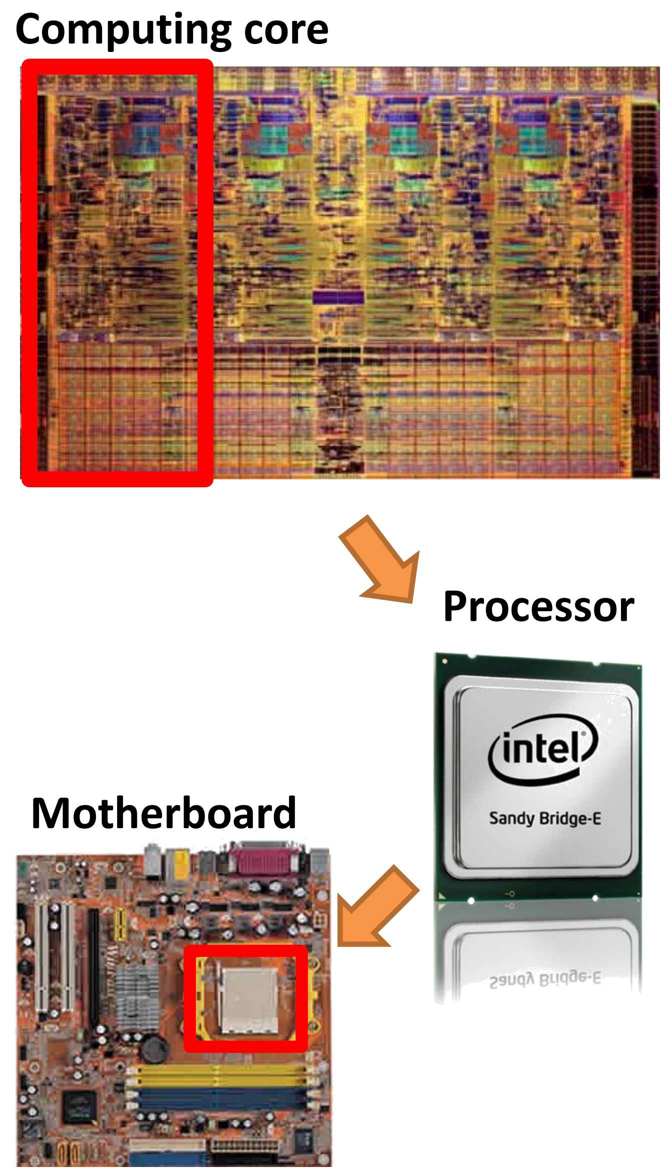
Outline

- **Introduction**
  - Core / Threads
  - Single thread and parallel performances
- **Introduction to programming language**
  - The compilation flow
  - Quick history of programming languages
  - Quick taxonomy of programming languages
- **What is a compiler ?**
  - We need a translator
  - Difference between a compiler and an assembler
- **Introduction to Optimizations**
  - Introduction to the intermediate representation (IR)
  - The front / middle / back ends
  - Example of optimizations
- **Internal representation of programs**
  - The control flow graph (CFG)
  - The data-flow graph (DFG)
  - The static single assignment form (SSA)
  - The function-call graph
- **Example of Optimizations**
  - Example 1: constant propagation
  - Example 2: function inlining
  - Example 3: combination
- **Loop optimization**
  - Definition of loops
  - Example of nest interchange
- **Conclusion**

INTRODUCTION TO CORE, THREAD AND SINGLE-THREAD PERFORMANCE

Computing Core

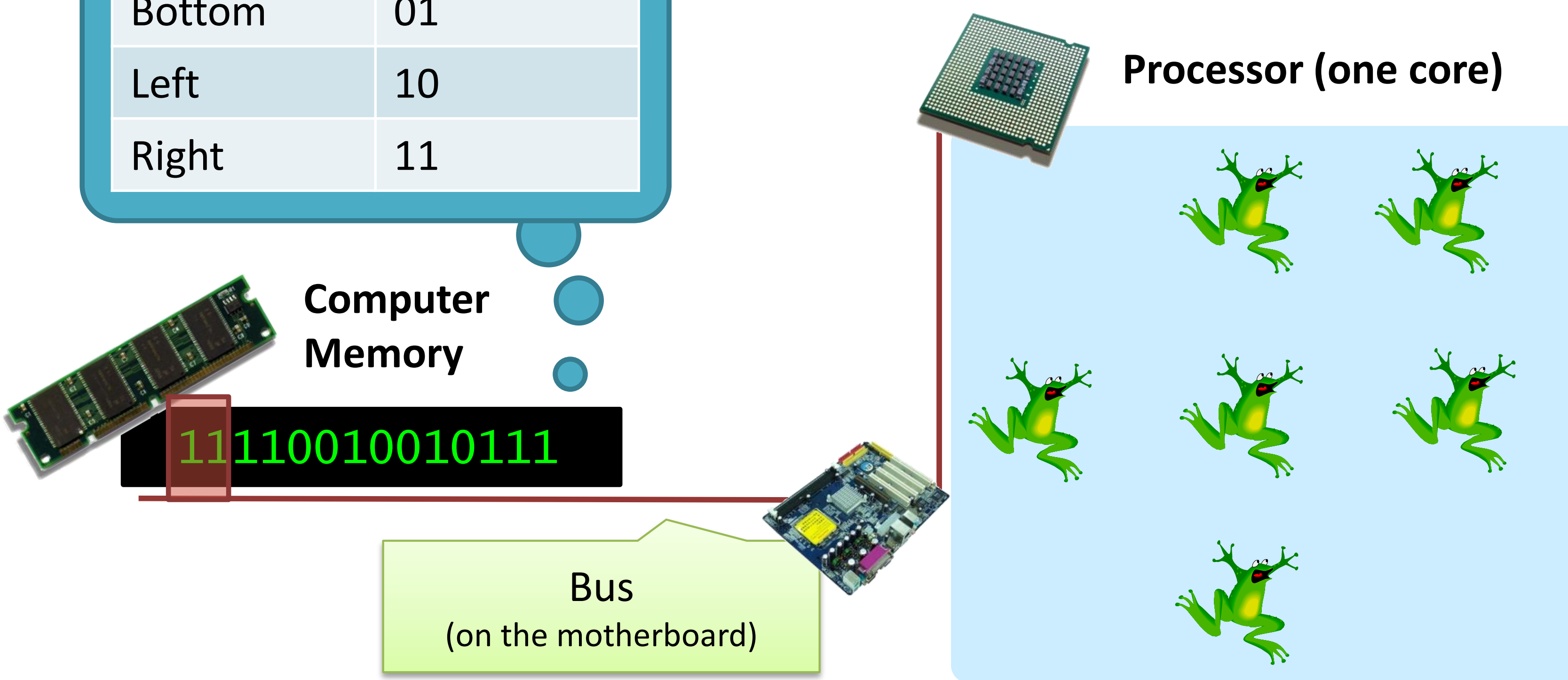
- Computer programs are executed on processors
- Processors are made of one or more computing cores
- A computing core executes a sequence of machine instructions
  - Traditionally, one core executes one sequence of machine instructions
  - Exception of Intel Hyper-Threading: one core executes two sequences of instructions
- The list of instructions that a core understands is called the Instruction-Set Architecture
  - Examples of ISAs
    - x86 (Intel 32 bit), x86-64 (Intel 64 bits)
    - MMX (early Intel multi-media extension
    - ARM v7, the most (only ?) used ISA in smartphones
  - One core may understand more than one ISA
    - Example of Intel Haswell (latest Intel architecture): x86, x86-64, MMX, SSE, SSE2, SSE3, AVX, AVX2 …



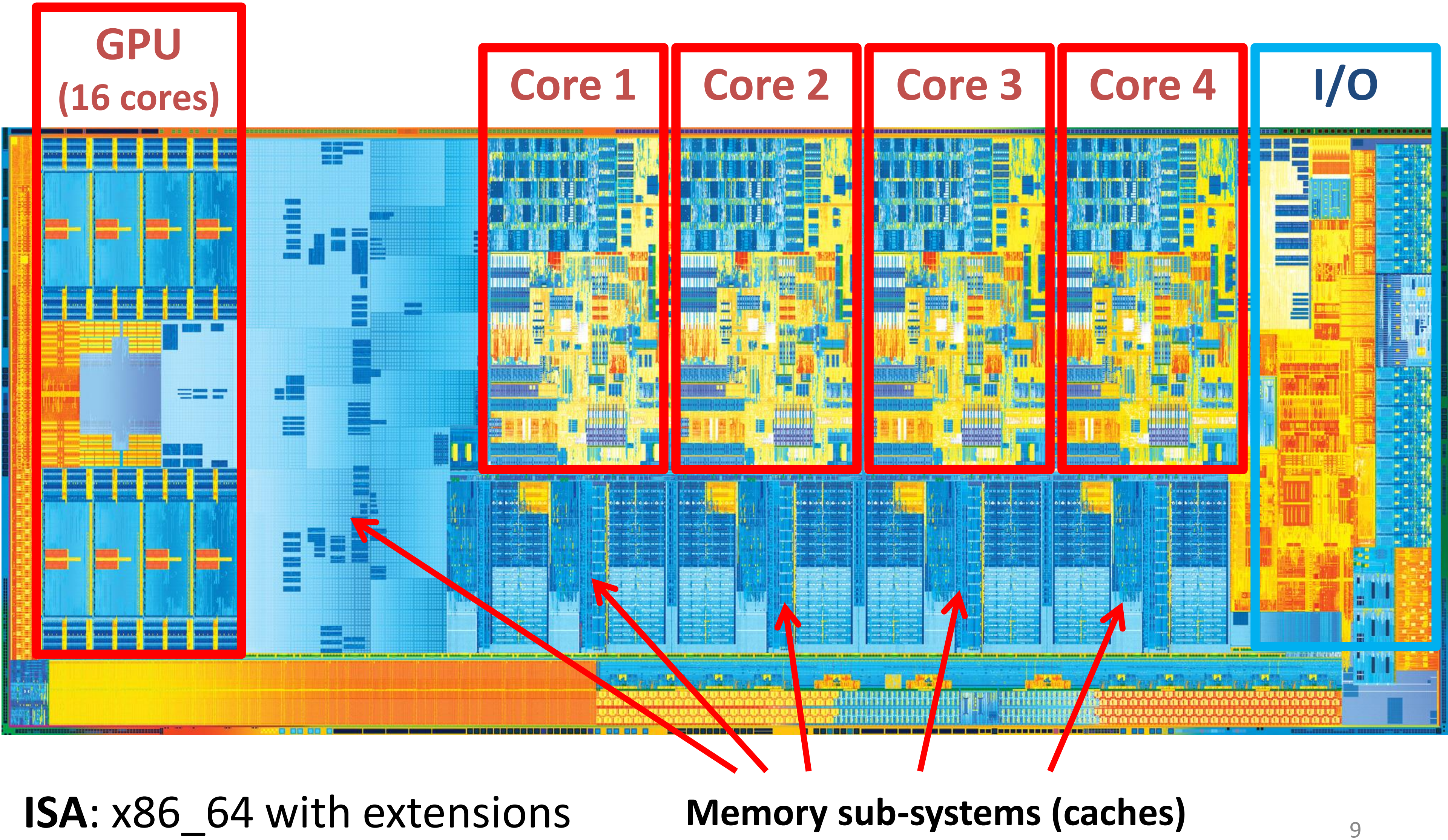
Processing Core: Example

Definition of the ISA

Instruction	Encoding
Top	00
Bottom	01
Left	10
Right	11



Processor: Picture of Intel Ivy Bridge



ISA: x86_64 with extensions

What is a Computing Thread ?

- A computing thread is a sequence of machine instructions
  - The instructions are executed one after the other
  - The execution order might vary depending on the architecture of the processor (e.g. out of order execution)
- A computer program is made of one or more threads
  - One thread: **single-thread programming**
  - >1 thread: **multi-thread programming**
- Threads allow to parallelize computations
  - We can expect programs to run **faster** (see next slide)
  - But one needs more than one computing core (hardware overhead)
- Multiple thread can also run on a single core
  - Threads are cut smaller sequence and scheduled by the operating system
  - Few acceleration to expect unless the program is often waiting for I/Os

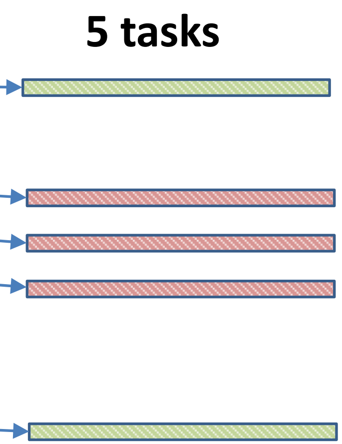
Single Thread and Parallel Performances

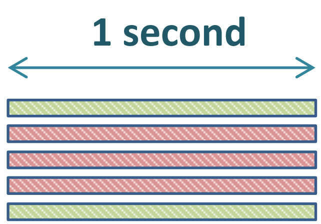
On an example first (1/2)

```
k = scanf(“%d”)
sum = 0

for(i=      0 to   N/6) M[i]=M[i] % k
for(i=  N/3+1 to 2*N/3) M[i]=M[i] % k
for(i=2*N/3+1 to     N) M[i]=M[i] % k

for(i=0 to N) sum+=M[i]
print(“%d”, sum)
```





Our program: 5 tasks of 1 second

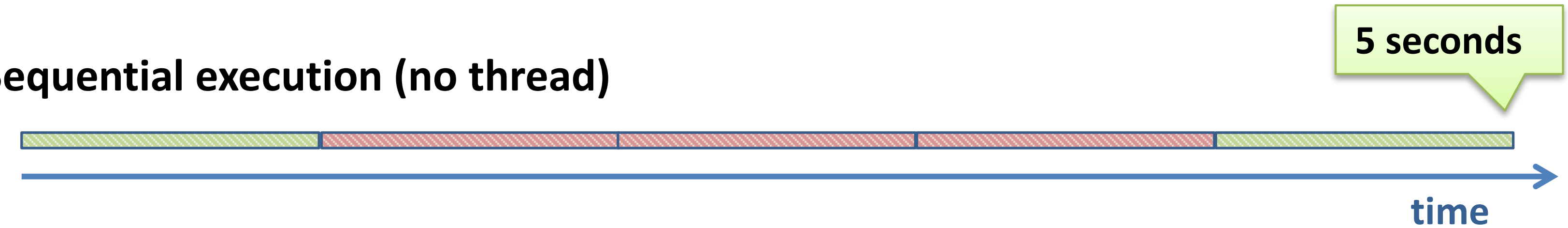
Green: can not execute in parallel

Red: can execute in parallel

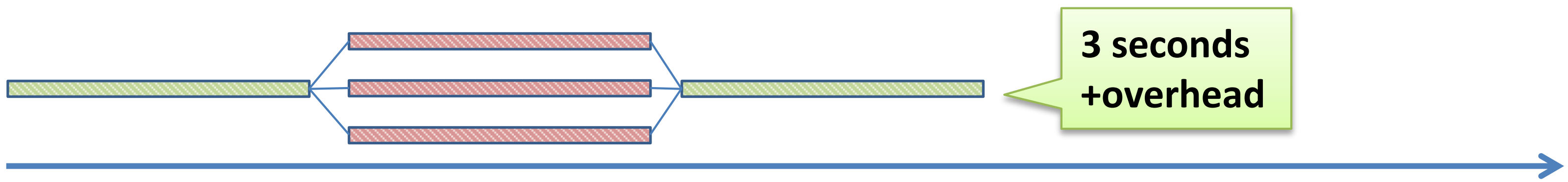
Single Thread and Parallel Performances

On an example (2/2)

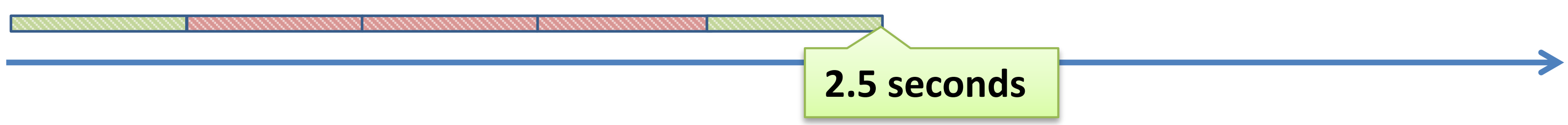
① **Sequential execution (no thread)**



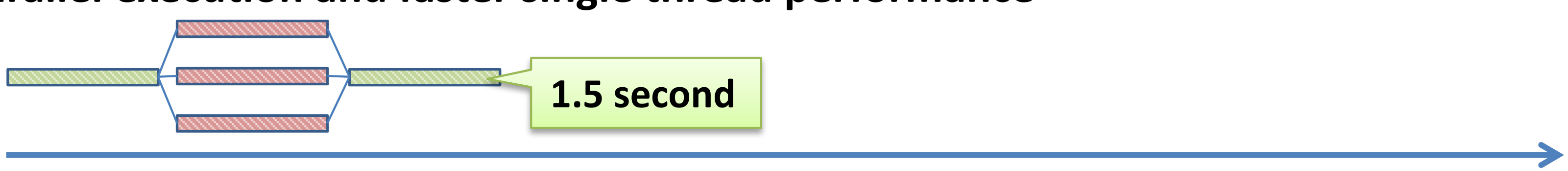
② **Parallel execution (with threads)**



③ **Sequential with 2 times faster single thread performance**

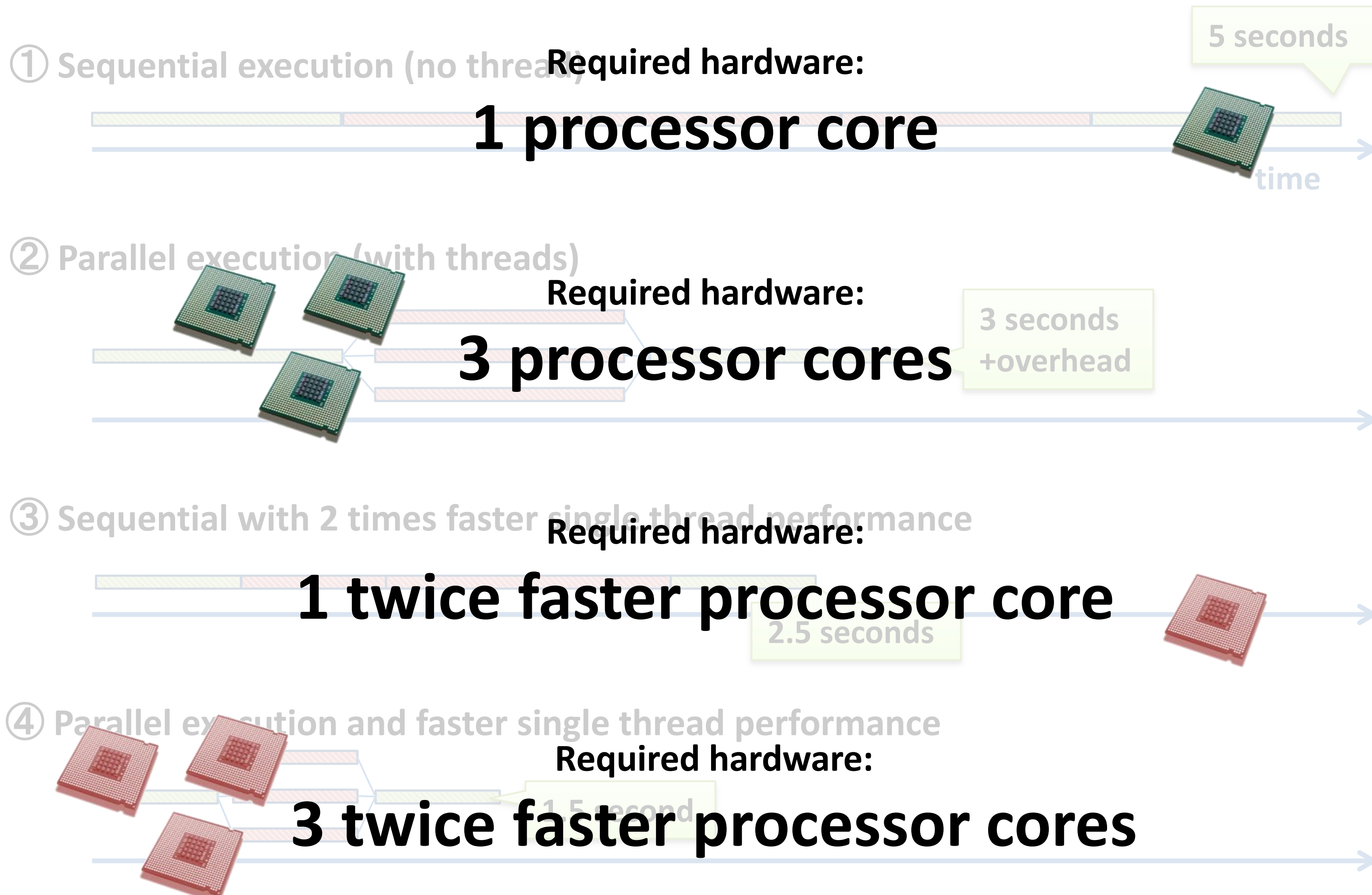


④ **Parallel execution and faster single thread performance**



Single Thread and Parallel Performances

On an example (2/2)



Required hardware:

1 processor core

Required hardware:

3 processor cores

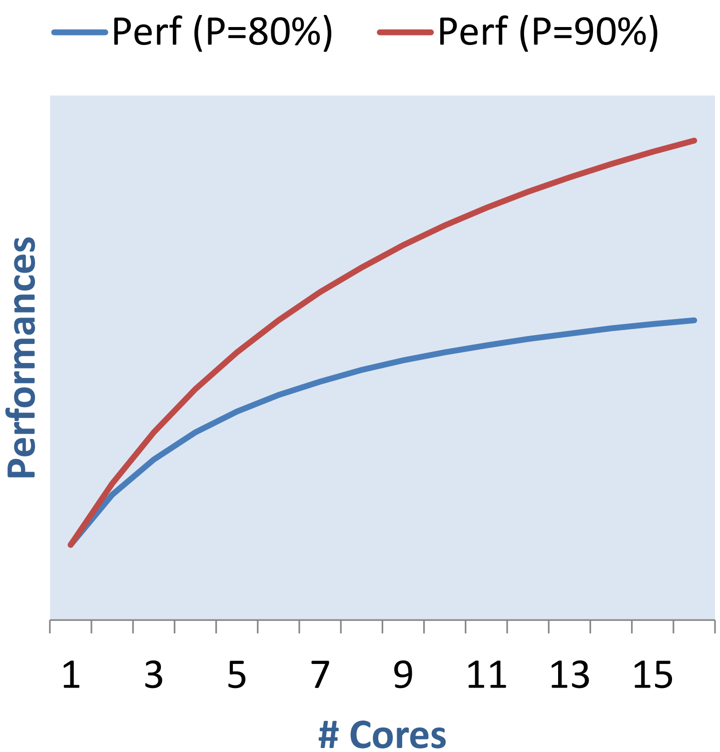
Required hardware:

1 twice faster processor core

Required hardware:

3 twice faster processor cores

Amdahl's Law

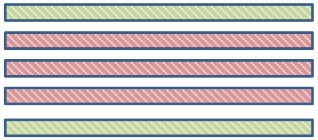


$$S(N) = \frac{1}{(1-P) + P/N}$$

N: number of cores
S(N): Speedup by using N cores
P: part of the program that you can parallelize

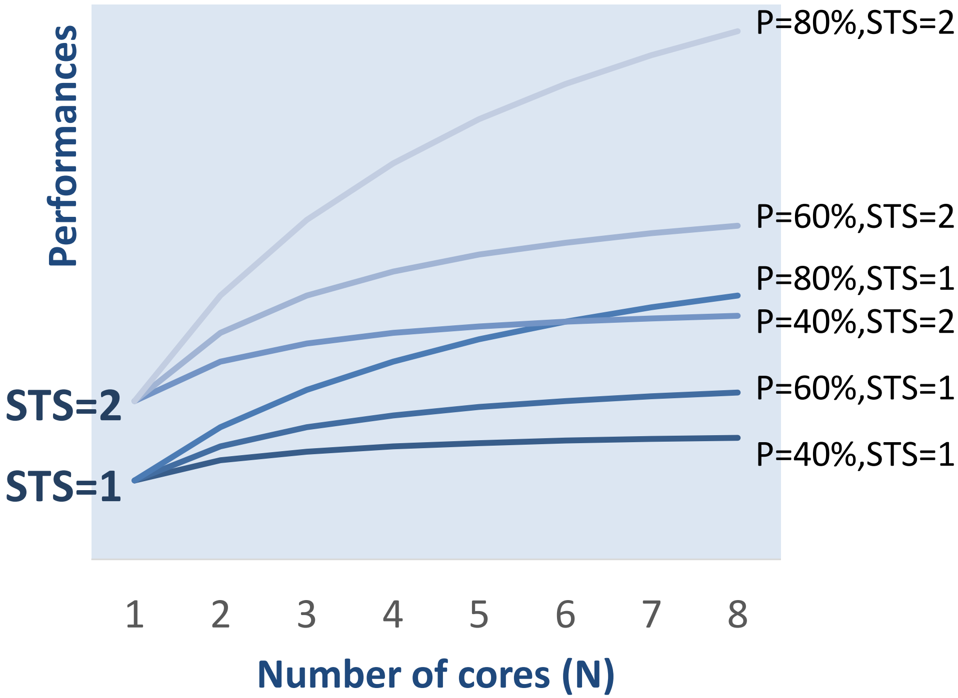
Gene Amdahl
1922 ~ (92 y.o.)
Formulated while at IBM

* picture: courtesy of Wikipedia



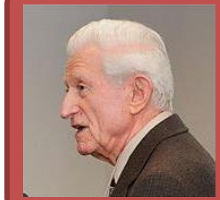
Previous example:
N=3, **P=3/5=60% ⇒ S(N)=1.67** times faster
(compared to N=1 and P=0)

Amdahl's Law and Single-thread Performance



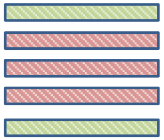
$$S(N) = \frac{STS}{(1-P) + P/N}$$

N: number of cores
S(N): Speedup by using N cores
P: part of the program that you can parallelize
STS: Single-thread speedup



Gene Amdahl
1922 ~ (92 y.o.)
Formulated while at IBM

* picture: courtesy of Wikipedia

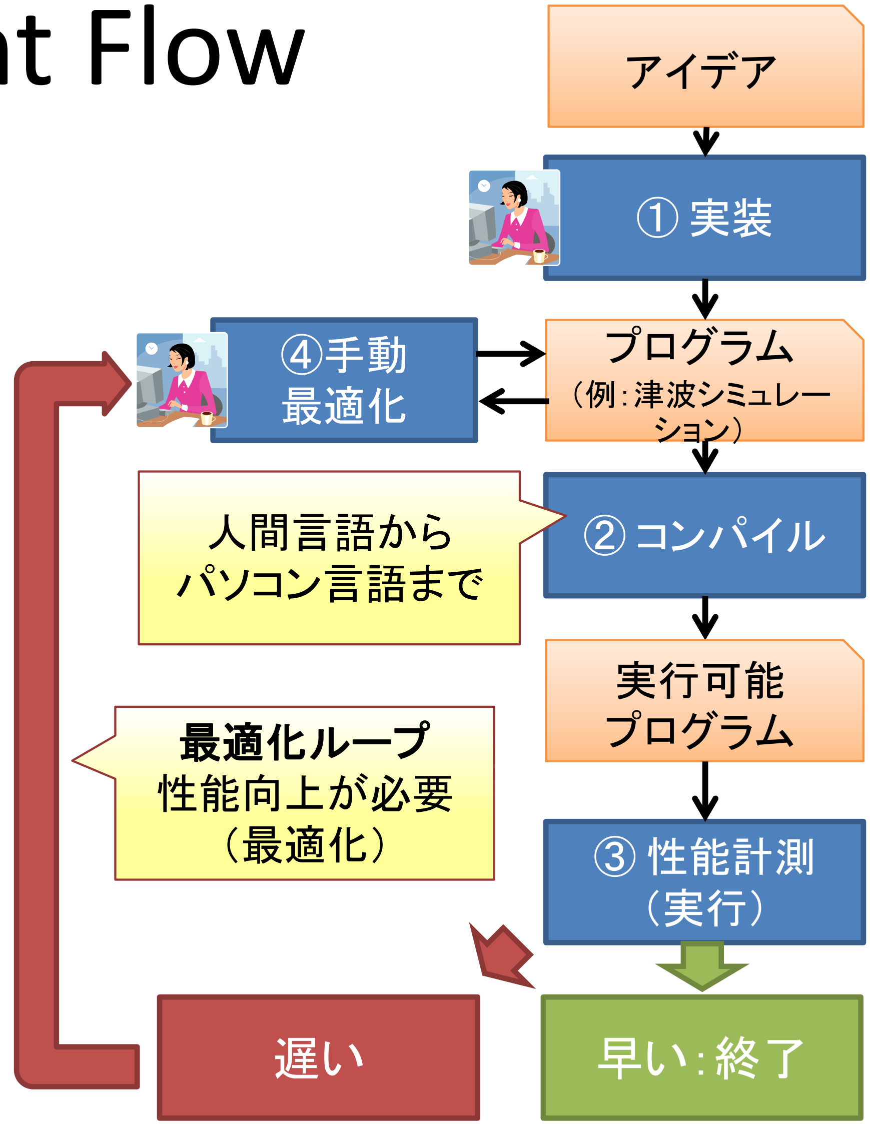


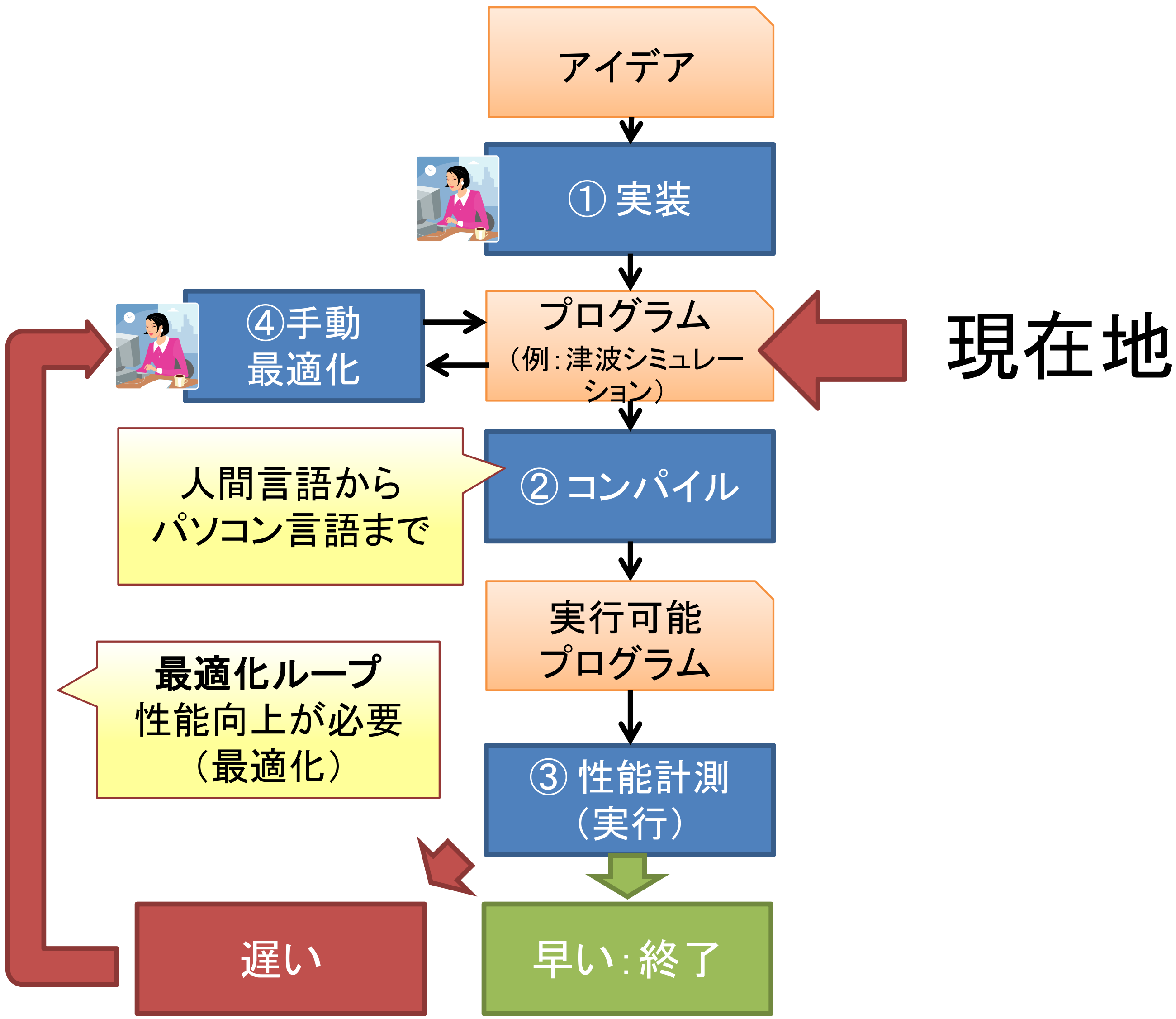
Previous example:
N=3, **P=3/5=60%,STS=2** ⇒ **S(N)=3.32** times faster
(compared to N=1, P=0 and STS=1)

INTRODUCTION TO PROGRAMMING LANGUAGES

The Development Flow

- Everything starts with an **<u>idea</u>**
- The programmer implements the idea in a <u>programming language</u>
- The programming language is compiled in <u>machine code</u>
- The machine code is executed on the <u>processor</u>
- The programmer repeats the flow until the program is fast enough




アイデア
① 実装
④手動
最適化
プログラム
（例：津波シミュレーション）
現在地
人間言語から
パソコン言語まで
② コンパイル
実行可能
プログラム
最適化ループ
性能向上が必要
（最適化）
③ 性能計測
（実行）
遅い
早い：終了

Very Quick History of Prog. Lang. (1/2)
Early times

1940's: machine code (first generation of prog. Lang.)

- Programming using binary code directly
- <u>Example of the frog</u>: 11110010010111

But binary has low productivity

- Too complex for human being: error prone
- Very hard to write large programs

1950's: assembly language (second generation of prog. lang.)

- Instead of writing "1" and "0", people write "add" or "sub"
- <u>Example of the frog</u>: 右;右;上;左;下;下;右

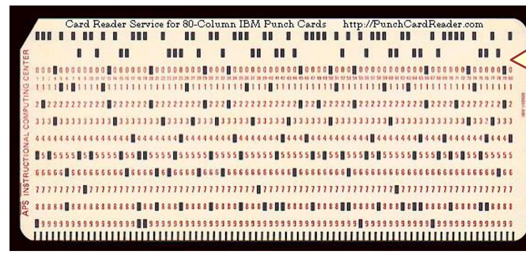
Productivity is better than binary, but it could be better

- Quick fix: people use "macro assembly instructions": instead of writing 右;右 we can write ２回右
- No real "high level language" yet

Very Quick History of Prog. Lang. (2/2) Toward modern languages

End of 1950: Apparition of first programming languages (third generation of prog. lang.)

- Fortran: scientific calculations
- Cobol: data processing
- Lisp: functional language



Written not as text files, but as **punch cards**

1969-1973: C language

- Created in Bell laboratories (USA) to implement the first UNIX OS
- The most used language right now
- Meant for system programming, but used for everything now (unfortunately)

1983: C++ language (object-oriented language)

- Extension of C to support object-oriented programming
- Widely popular now

1996: Java (virtual machines and just-in-time compilation)

- Resembles C++, but abstracts memory allocations
- Originality: the Java compiler compiles in bytecode, not machine code

Example of Languages

Binary Machine Code

```
09 2e 73 65 63 74 69 6f 6e 09 5f 5f 54 45 58 54
2c 5f 5f 74 65 78 74 2c 72 65 67 75 6c 61 72 2c
70 75 72 65 5f 69 6e 73 74 72 75 63 74 69 6f 6e
73 0a 09 2e 67 6c 6f 62 6c 09 5f 6d 61 69 6e 0a
09 2e 61 6c 69 67 6e 09 34 2c 20 30 78 39 30 0a
5f 6d 61 69 6e 3a 20 20 20 20 20 20 20 20 20 20
20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20
20 20 20 20 20 20 20 20 23 23 20 40 6d 61 69 6e
0a 09 2e 63 66 69 5f 73 74 61 72 74 70 72 6f 63
0a 23 23 20 42 42 23 30 3a 0a 09 70 75 73 68 71
09 25 72 62 70 0a 4c 74 6d 70 32 3a 0a 09 2e 63
66 69 5f 64 65 66 5f 63 66 61 5f 6f 66 66 73 65
74 20 31 36 0a 4c 74 6d 70 33 3a 0a 09 2e 63 66
69 5f 6f 66 66 73 65 74 20 25 72 62 70 2c 20 2d
31 36 0a 09 6d 6f 76 71 09 25 72 73 70 2c 20 25
72 62 70 0a 4c 74 6d 70 34 3a 0a 09 2e 63 66 69
5f 64 65 66 5f 63 66 61 5f 72 65 67 69 73 74 65
72 20 25 72 62 70 0a 09 6d 6f 76 6c 09 24 30 2c
20 2d 34 28 25 72 62 70 29 0a 09 6d 6f 76 6c 09
25 65 64 69 2c 20 2d 38 28 25 72 62 70 29 0a 09
6d 6f 76 71 09 25 72 73 69 2c 20 2d 31 36 28 25
72 62 70 29 0a 09 63 6d 70 6c 09 24 31 2c 20 2d
38 28 25 72 62 70 29 0a 09 6a 67 09 4c 42 42 30
5f 32 0a 23 23 20 42 42 23 31 3a 0a 09 6d 6f 76
6c 09 24 2d 31 2c 20 2d 34 28 25 72 62 70 29 0a
09 6a 6d 70 09 4c 42 42 30 5f 33 0a 4c 42 42 30
5f 32 3a 0a 09 6d 6f 76 71 09 2d 31 36 28 25 72
62 70 29 2c 20 25 72 61 78 0a 09 6d 6f 76 71 09
38 28 25 72 61 78 29 2c 20 25 72 61 78 0a 09 6d
6f 76 73 62 6c 09 28 25 72 61 78 29 2c 20 25 65
63 78 0a 09 61 64 64 6c 09 24 33 2c 20 25 65 63
78 0a 09 6d 6f 76 6c 09 25 65 63 78 2c 20 2d 34
28 25 72 62 70 29 0a 4c 42 42 30 5f 33 3a 0a 09
6d 6f 76 6c 09 2d 34 28 25 72 62 70 29 2c 20 25
65 61 78 0a 09 70 6f 70 71 09 25 72 62 70 0a 09
72 65 74 0a 09 2e 63 66 69 5f 65 6e 64 70 72 6f
63 0a 0a 0a 2e 73 75 62 73 65 63 74 69 6f 6e 73
5f 76 69 61 5f 73 79 6d 62 6f 6c 73 0a
```

Assembly (Intel x86-64)

```
_main:                          ## @main
  .cfi_startproc
## BB#0:
  pushq           %rbp
Ltmp2:
  .cfi_def_cfa_offset 16
Ltmp3:
  .cfi_offset %rbp, -16
  movq %rsp, %rbp
Ltmp4:
  .cfi_def_cfa_register %rbp
  movl $0, -4(%rbp)
  movl $3, -8(%rbp)
  movl $42, -12(%rbp)
  movl $67, -16(%rbp)
  movl $0, -20(%rbp)
LBB0_1:
  movl -20(%rbp), %eax
  cmpl -16(%rbp), %eax
  jge             LBB0_4
## BB#2:
  movl -8(%rbp), %eax
  imull -8(%rbp), %eax
  cltd
  idivl -12(%rbp)
  movl %edx, -8(%rbp)
## BB#3:
  movl -20(%rbp), %eax
  addl $1, %eax
  movl %eax, -20(%rbp)
  jmp             LBB0_1
LBB0_4:
  movl -8(%rbp), %eax
  popq %rbp
  ret
  .cfi_endproc
```

C

```
int main() {
  int seed = 3, k=42, N=67;
  for(int i=0; i<N; i++) seed = seed * seed % k;
  return seed;
}
```

ruby

```
seed = 3; k = 42; N = 67
(0…N).each { |x| seed = seed * seed % k }
```

There are many Paradigms to Classify Programming Languages

Memory model
Von-Neuman ?
NUMA ?

Programming model
Object-oriented ?
Functional ?

Typing model
Strongly typed ?
Weakly typed ?
Non-typed ?

Threading model
Single-thread ?
Explicit threads ?

Runtime
No runtime ?
Virtual machine ?

Memory allocation model
Explicit allocation ?
Automatic reference counting ?
With garbage collector ?

Compilation model
Statically compiled ?
Just-in-time compiled ?
Interpreted (not compiled) ?

NUMA: Non-uniform memory accesses

There are many Paradigms to Classify Programming Languages

Memory model
Von-Neuman ?
NUMA ?

Programming model
Object-oriented ?
Functional ?

Typing model
Strongly typed ?
Weakly typed ?
Non-typed ?

Threading model
Single thread ?
Explicit threads ?

Runtime
No runtime ?
Virtual machine ?

Most language are multi-paradigm

Memory allocation mode
Explicit allocation ?
Automatic reference countin
With garbage collector ?

ilation model
ally compiled ?
-time compiled ?
reted (not compiled) ?

NUMA: Non-uniform memory accesses

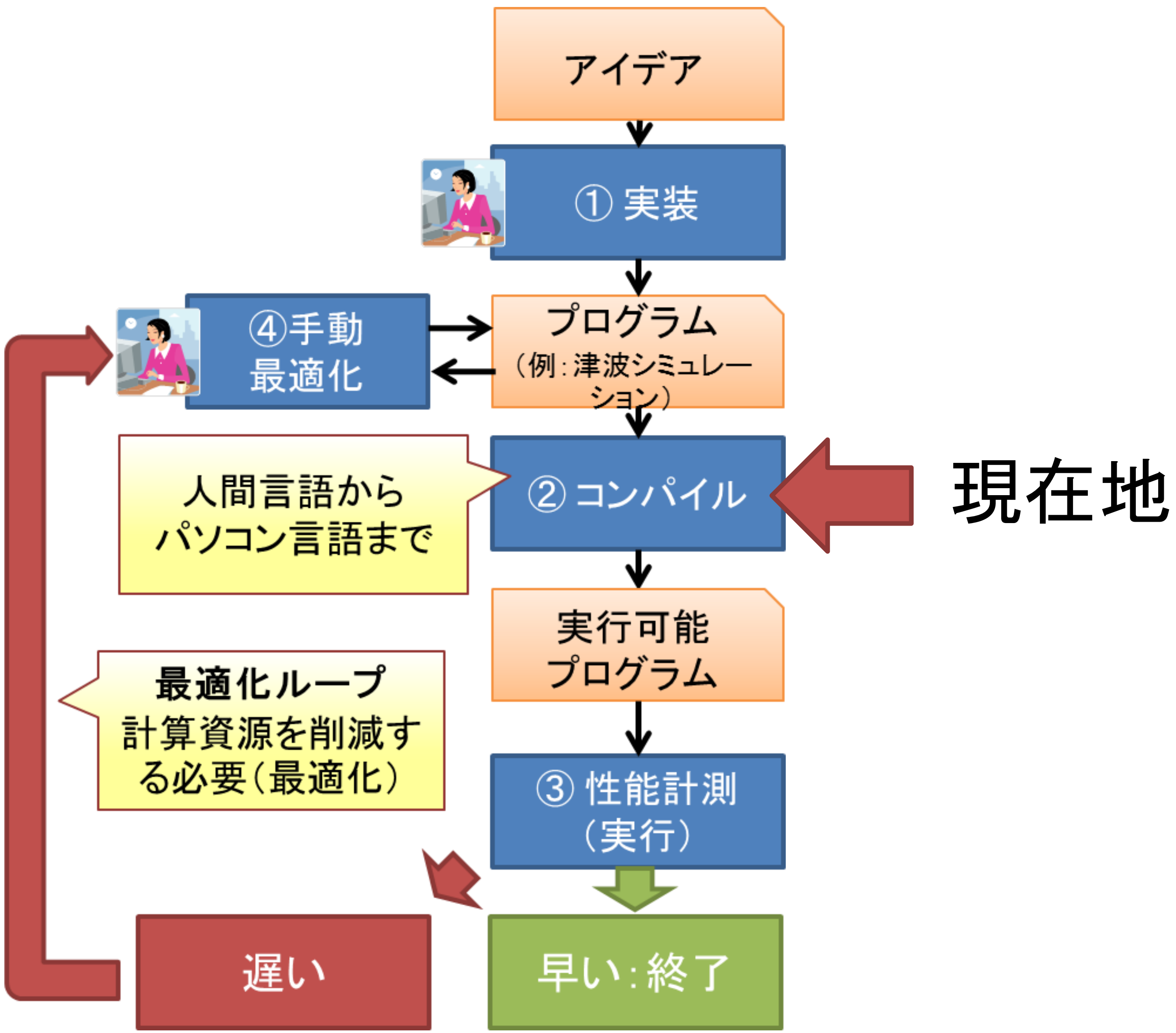
WHAT IS A COMPILER ?

Programmers use programming language

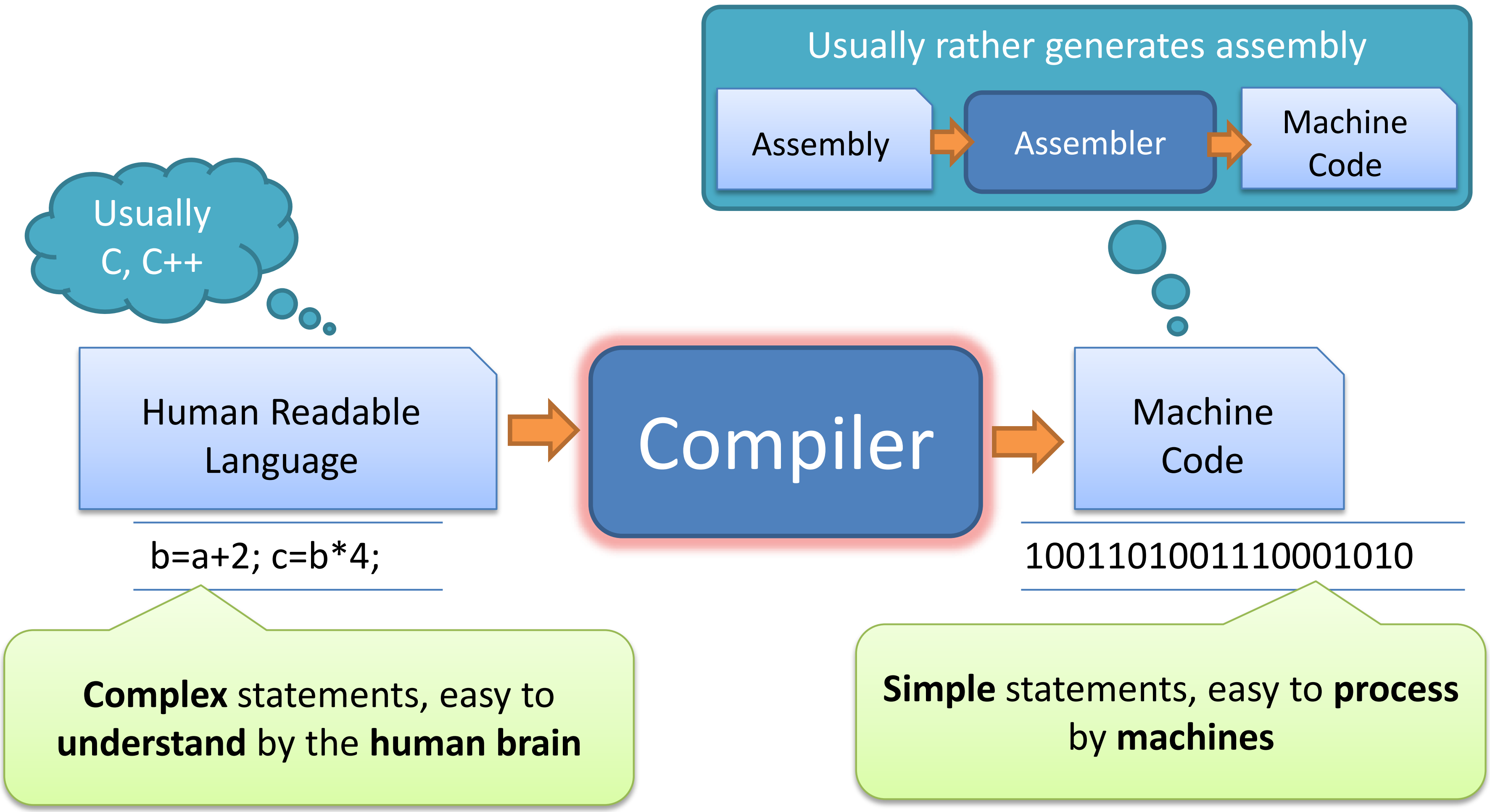
but

Processors only understand machine code

We need a translator: the compiler


アイデア
① 実装
④手動
最適化
プログラム
（例：津波シミュレーション）
② コンパイル
現在地
人間言語から
パソコン言語まで
実行可能
プログラム
最適化ループ
計算資源を削減する必要（最適化）
③ 性能計測
（実行）
遅い
早い：終了

Input / Output of the Compiler



Example of program

you shall go two times right

you shall go top

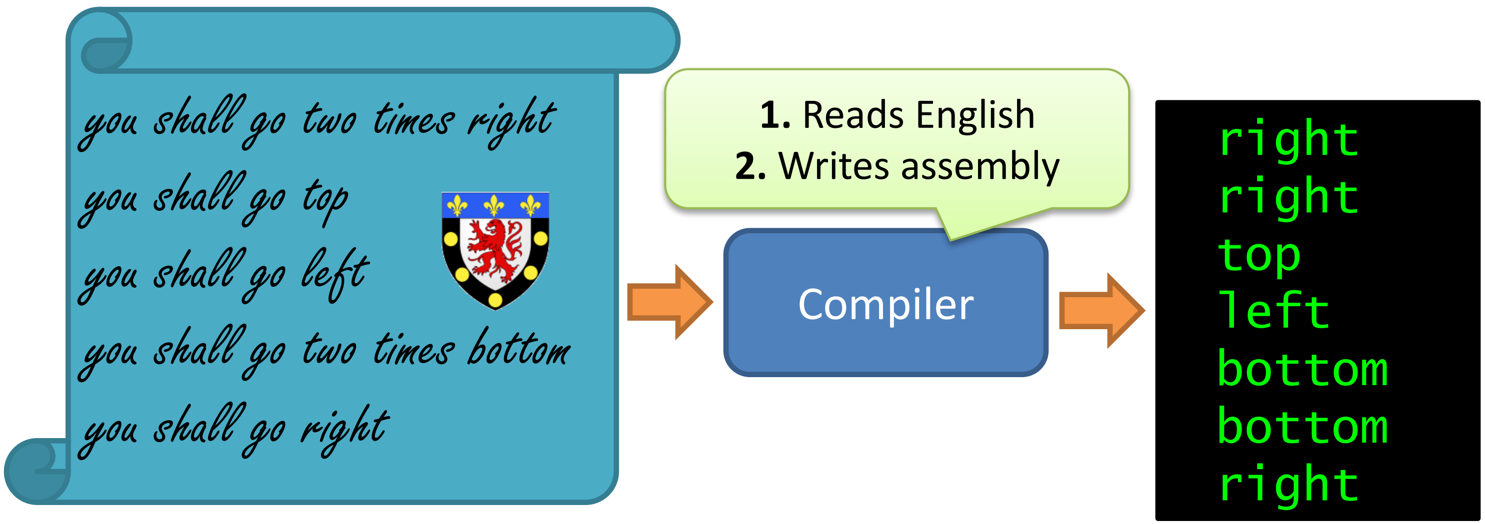
you shall go left

you shall go two times bottom

you shall go right

The example of the frog of slide 8

Example of assembly



Programming language
(human readable language)

Assembly

Example of machine code

(Usually) One assembly instruction per processor instruction

(modern assembly language feature "pseudo" or "macro" instructions that correspond to more than one processor instruction)

Definition of the ISA

Instruction	Encoding
Top	00
Bottom	01
Left	10
Right	11

```
right
right
top
left
bottom
bottom
right
```

Assembly

Assembler

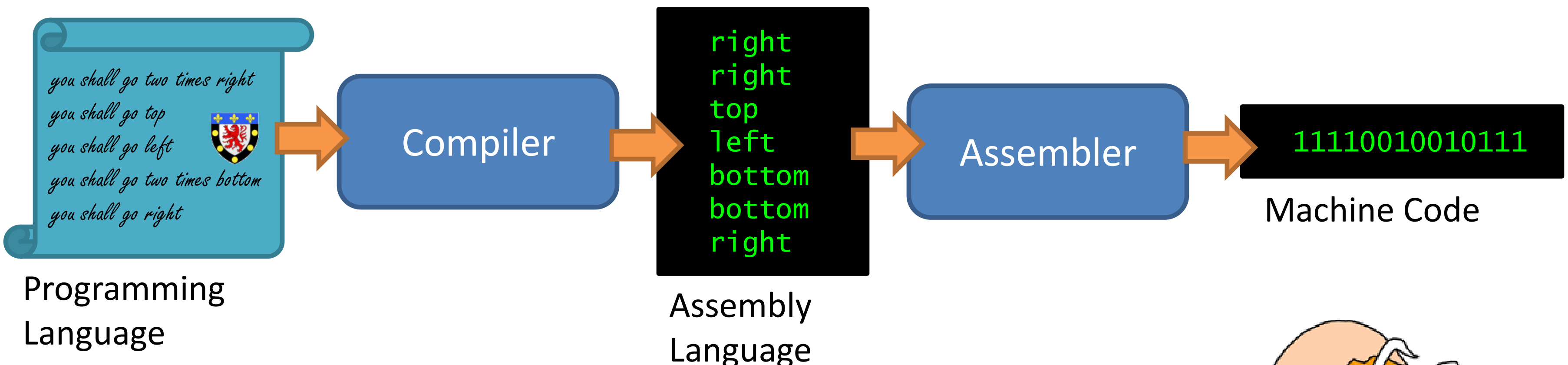
```
11110010010111
```

Machine Code

Assembly and Machine Code are equivalent.

Each processor architecture come with its dedicated assembly and machine code languages.

Sum-up: the Compilation Flow



Popular Compilers

Intel Compiler (icc)

Microsoft Compiler (Visual Studio)

GNU C Compiler (gcc)

LLVM

But the compiler is far more than just a translator…

It can **optimize** programs

COMPILER OPTIMIZATION

There are unnecessary moves in this program

Can you find them ?

you shall go two times right

you shall go top

you shall go left

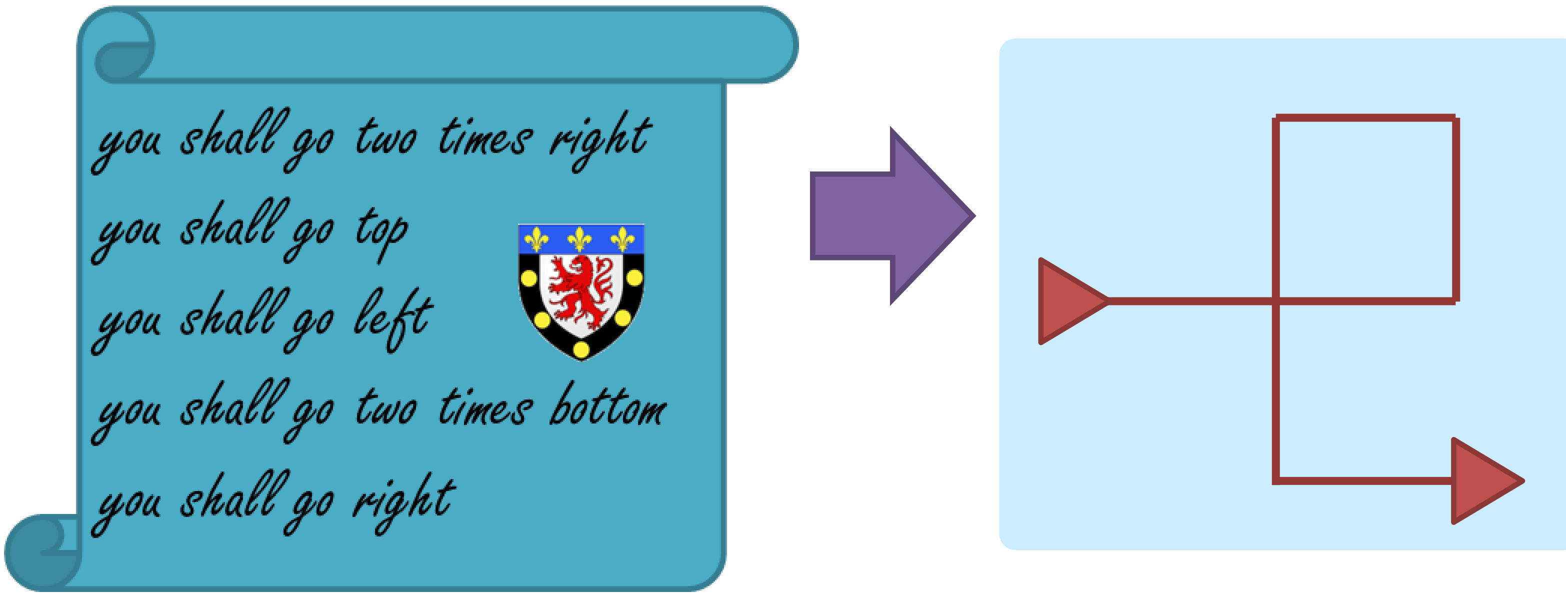
you shall go two times bottom

you shall go right

Hard to answer from the text of the program: people tend to use **graphical representation**

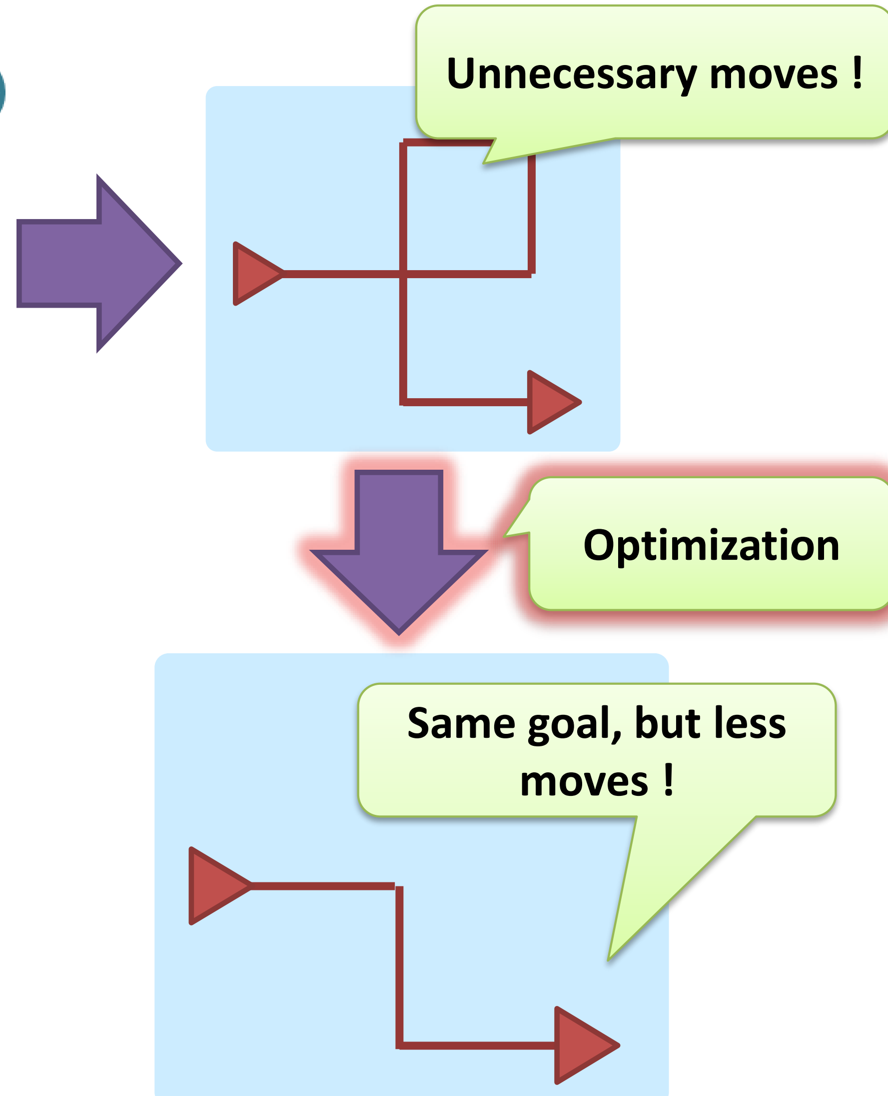
The compiler is the same !

The Intermediate Representation



- The IR is the way the compiler represents program internally
- It expresses the important properties of the program for further analysis
- In particular, it eases **optimization**

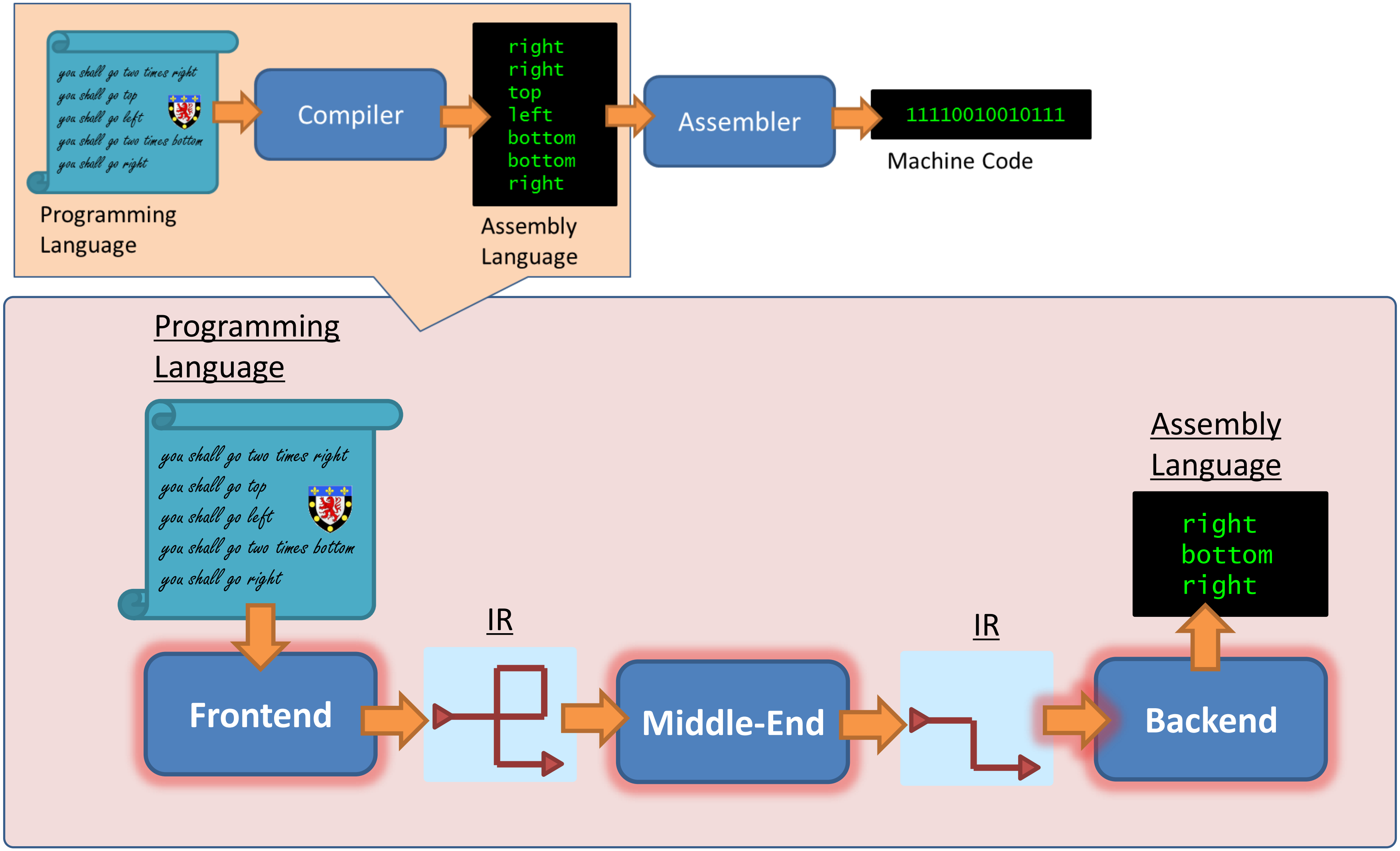
Example of Optimization



Optimization is about Removing Unnecessary Calculations

But, without changing the result of the program

Front / Middle / Back-end (1/2)



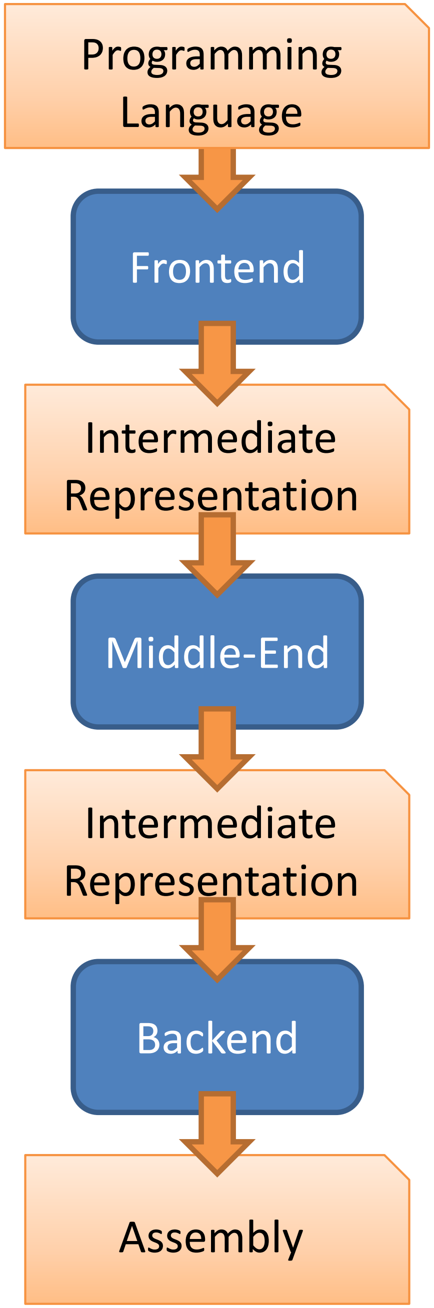
Front / Middle / Back-end (1/2)

- **Frontend**
  - **Input:** Programming language
  - **Output:** Intermediate representation
  - **Key steps:** lexing, parsing
  - Often uses another IR inside for: the abstract syntax tree (AST)

Also called **H**igh **L**evel **L**anguage

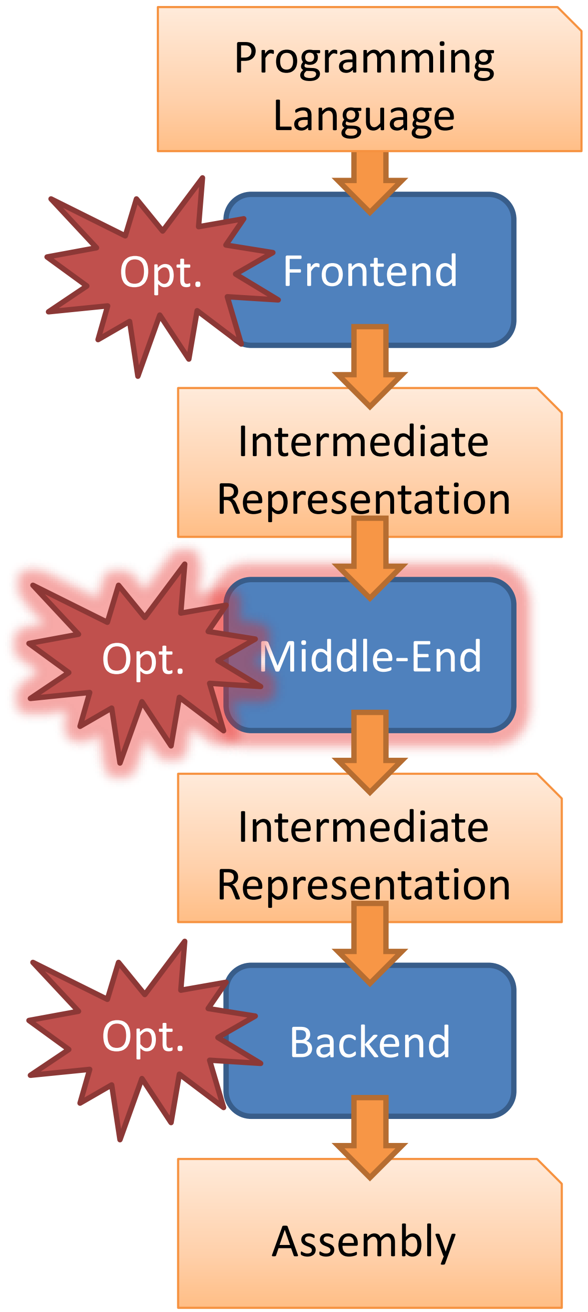
Grammar, language theory

- **Backend**
  - **Input:** Intermediate representation
  - **Output:** Assembly
  - **Key steps:** instruction selection and register allocation
- **Middle-end**
  - **Input:** Intermediate representation
  - **Output:** Intermediate representation
  - **Key steps:** many kinds of optimizations !
- **Intermediate representation (IR)**
  - Stored in memory, but can also be saved in files
  - Every compiler has its own IR (gcc, LLVM …)



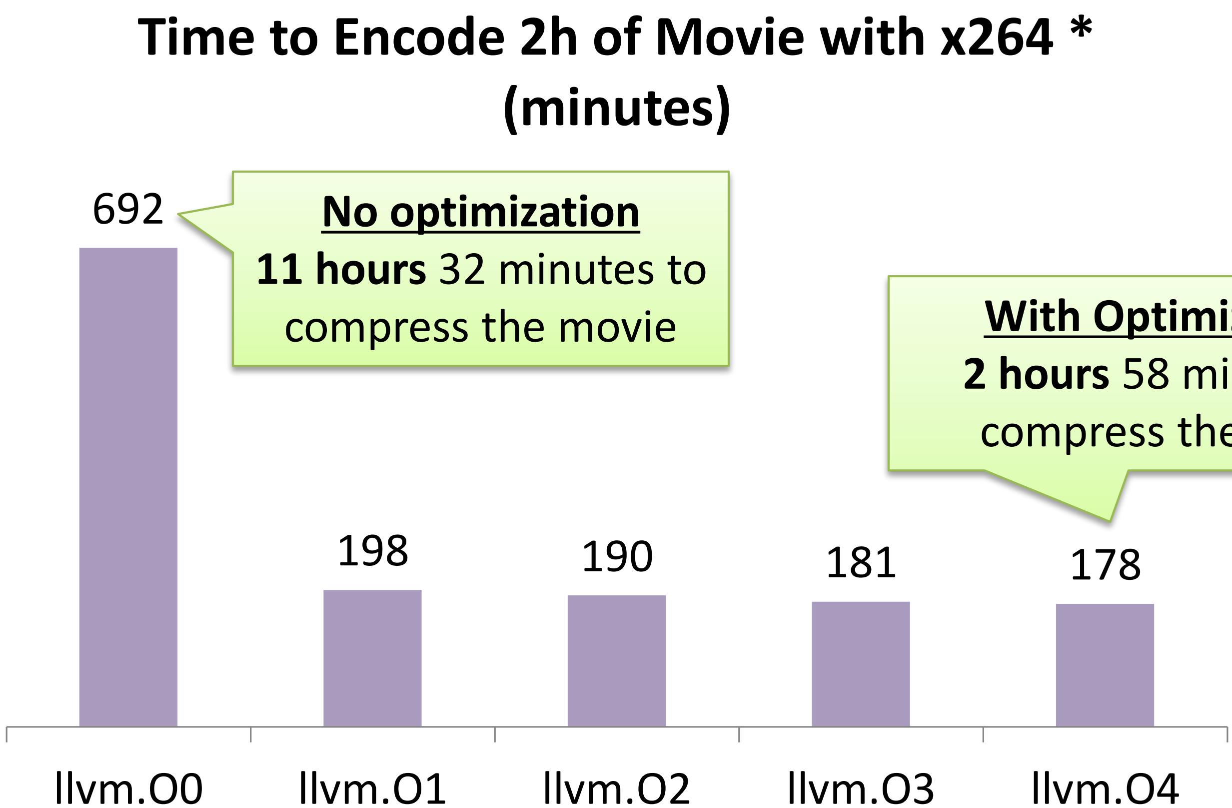
Optimizations are carried at every compilation stage

- In the front-end
  - The transformations from HLL to IR should be of high quality
  - Several optimizations are done at AST level
  - AST is often referred to as a “high-level IR”
- In the backend
  - Performance are influenced by the instruction selection and register allocation
- **In the middle-end**
  - **Our focus today**



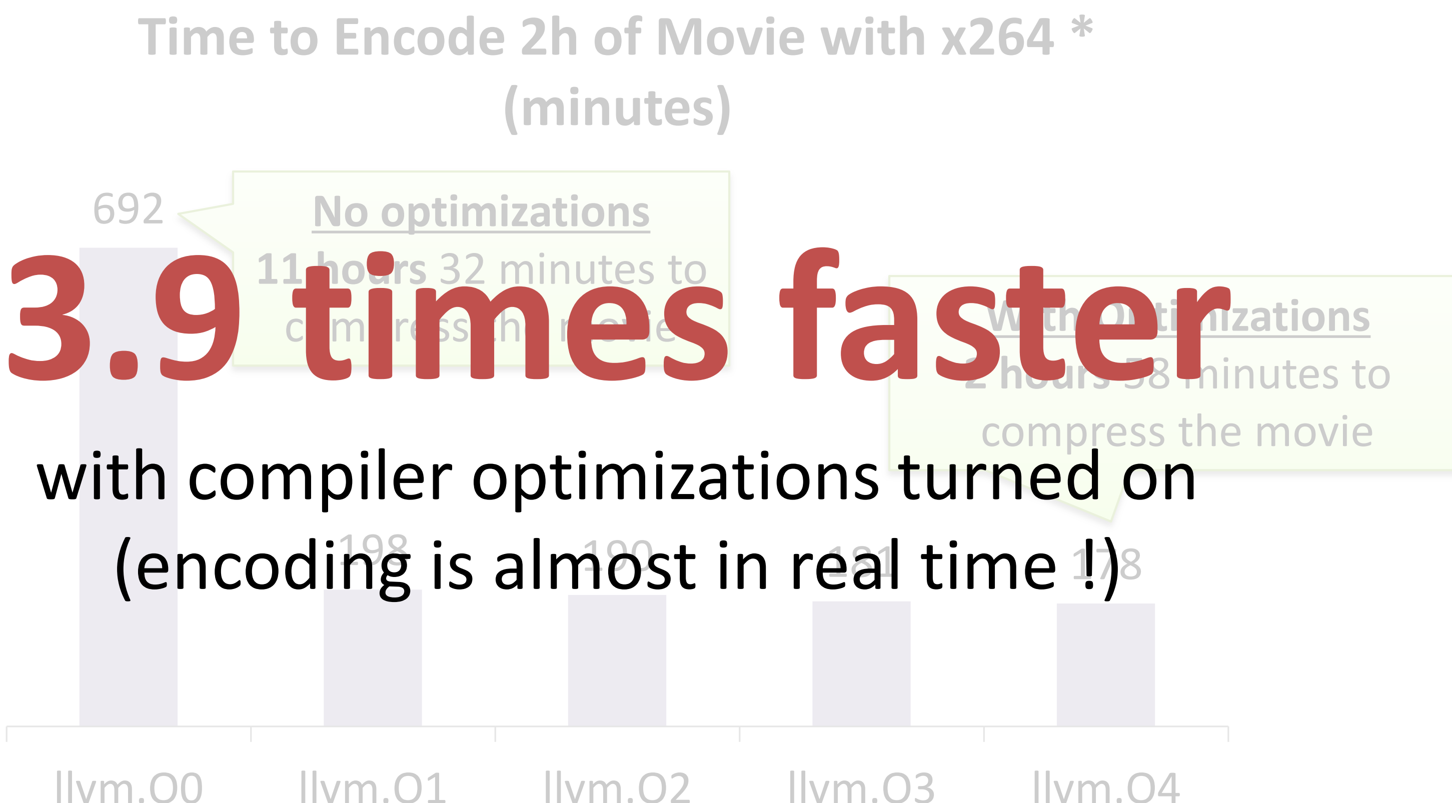
HLL: High level language / **IR:** Intermediate representation
AST: Abstract syntax tree

Speedup Video Compression with Optimization (real example)



*30 fps, cif (352x288), main profile, extrapolated from video “bridge close”
machine: Intel Core2Duo@2.26GHz, 8GB DDR3, MacOS X 10.7.4

Speedup Video Compression with Optimization (real example)



3.9 times faster

with compiler optimizations turned on
(encoding is almost in real time !)

*30 fps, cif (352x288), main profile, extrapolated from video “bridge close”
machine: Intel Core2Duo@2.26GHz, 8GB DDR3, MacOS X 10.7.4

Effect of Optimizations on **Power Consumption**

- The K supercomputer dissipates 15MW
- Let us consider a program that requires 1h to run
  - You need 15MWH to run it
- Let us say you are able to 3.9 times with optimization
  - You need 15/3.9=3.8MWH to run it
  - **You saved 11.2MWH**, that is, the power consumption of **15 apartments (a small mansion) during one month !**
- All we had to do is to set the correct optimization option to the compiler

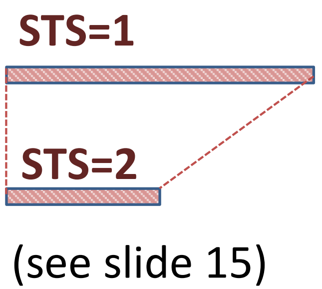
MWH: Mega Watt per Hour (メガワット時)

What kind of Optimizations are carried-out by Compilers ?

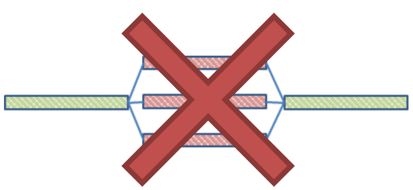
There are may optimization techniques !
(LLVM: more than 50 !)

Carried-out optimization depend on the compiler and the target processor

- Compilers mainly optimize **single-thread performance**
  - Remove unnecessary computations
  - Improve the use of cache to reduce access latency
  - Reduce memory accesses by using processor registers
  - Take advantage of ISA extension (especially SIMD)



- Compilers are very **bad at thread parallelization**
  - It is the responsibility of the programmer to parallelize the code



Practical Example: Remove Unnecessary Calculations

Example of C program: transformation to capital letters for string **str** of length **N**

```
for(i=0; i<strlen(str); i++) {
  str[i] += ‘A’-’a’;
}
```

strlen ‘A’-’a’

computations = N^2 + N + N = **o(N^2)**

+=

First optimization*

```
int N=strlen(str);
for(i=0; i<N; i++) {
  str[i] += ‘A’-’a’;
}
```

strlen ‘A’-’a’

computations = N + N + N = **3N**

+=

Second optimization*

```
int N=strlen(str);
int delta=‘A’-’s’;
for(i=0; i<N; i++) {
  str[i] += delta;
}
```

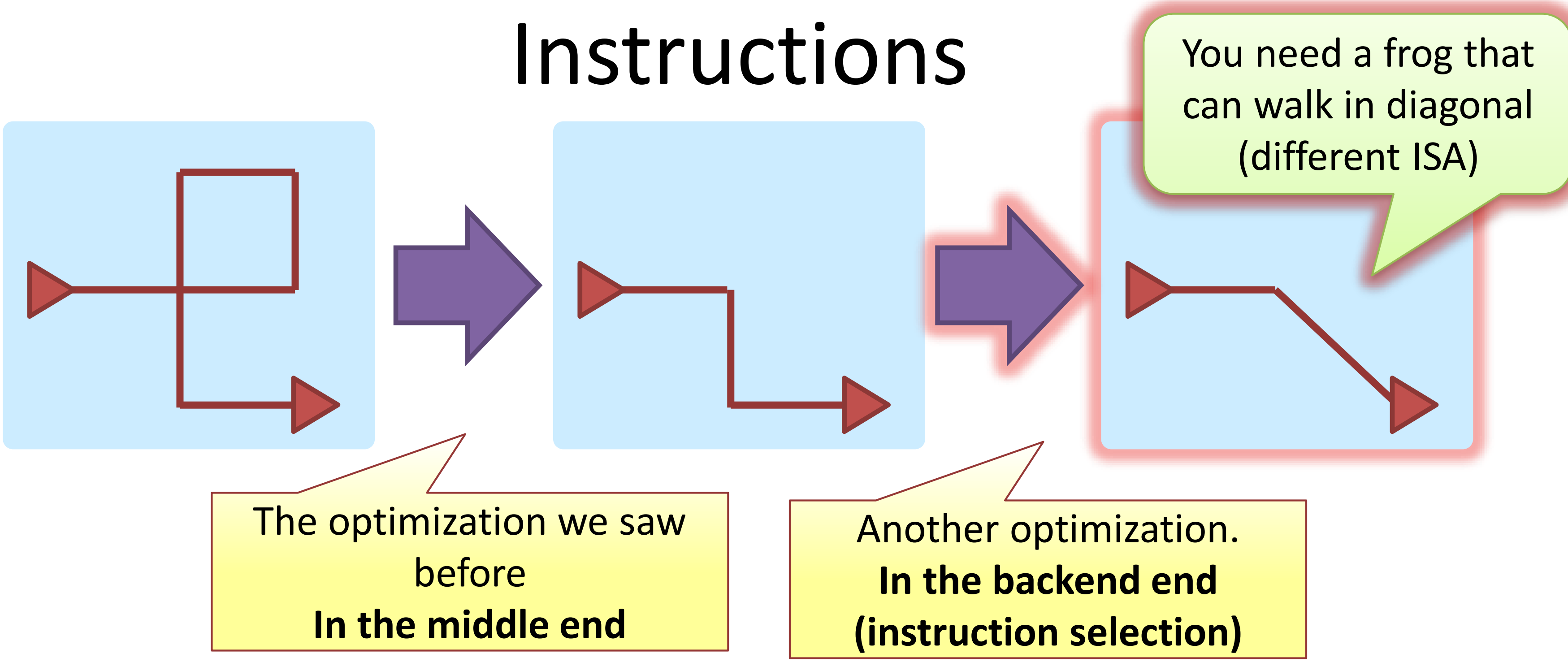
strlen ‘A’-’a’

computations = N + 1 + N = **2N+1**

+=

*Type of optimization: loop-invariant removal

Frog example: Better Use of Processor Instructions

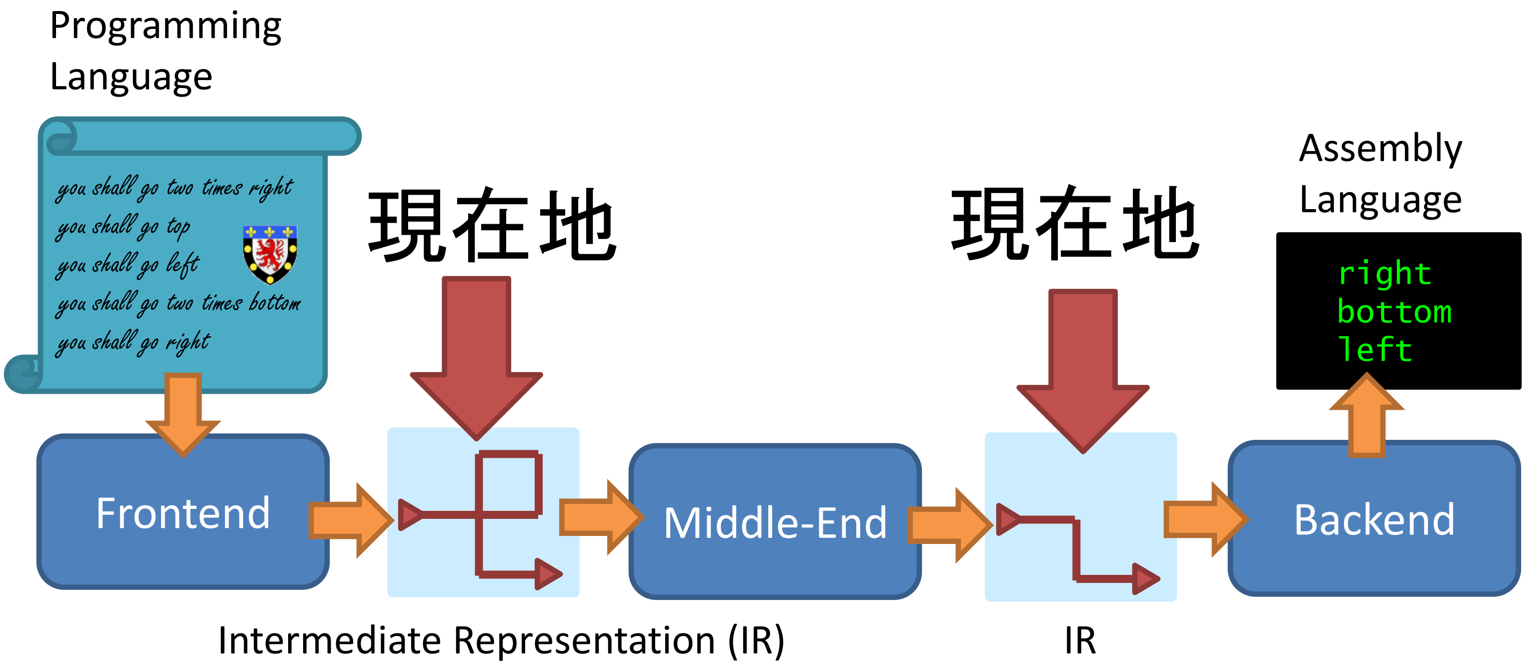


Common example in real, modern processors:

- **Compound instructions**:
  - MAC: perform multiplication and addition
  - Memory access + arithmetic (common in Intel Processors)
- **Vector instructions** (see next lecture)

ISA: Instruction set architecture (see slide 7) / **MAC**: Multiplication and Accumulation

CONTROL AND DATA-FLOW GRAPHS



What kind of IR compilers use for real ?

Instructions and Graphs

Let us start with some **definitions**

Sequential instructions

Control flow instructions

Basic Block

Data dependencies

Taxonomy of Instructions

- **Def. 1: Sequential instructions**
  - Are executed in the same order as they are written
  - Actually perform computations
  - Examples: load, add …
- **Def. 2: Control flow instructions**
  - Allow to jump between different locations of a sequence
  - Blue arrows
  - Examples: branch, jump, exceptions …

C Language

```
int a = 1;
int b = 2;
if (a<b) {
  b = a;
} else {
  a = b;
}
```

Control flow

IR instructions

```
  a = 1
  b = 2
  if a<b goto L2
L1:
  b = a
  goto L3
L2:
  a = b
  goto L3
L3:
```

Def. 3: The Basic Block

- A basic block is a sequence of instructions that are always executed together
- A basic block only contains <u>sequential instructions</u> and often ends with one <u>control flow instruction</u>
- Example:

```
  a = 1
  b = 2
  if a<b goto L2
```

Basic Block 1

```
L1:
  b = a
  goto L3
```

Basic Block 2

```
L2:
  a = b
  goto L3
```

Basic Block 3

```
L3:
```

Basic Block 4

Def. 4: Data Dependences

- For all instructions, we can define
  - **The input set**: the set of the data that the instructions need to be executed
  - **The output set**: the set of data generated by the instructions
- The inclusion between the input and output sets determines the type of data dependencies
  - **See next slide**
- Examples:

a=1

Input: variable a
Output: variable a

int a=1

Input: nothing
Output: variable a

a=b+1

Input: variables a and b
Output: variable a

if(a>3)

Input: variables a
Output: nothing

Types of Data Dependences

I1: a = 2
I2: a = 3
I3: b = a + 1
I4: c = a + 2
I5: a = b + a
I6: d = 6

Example Program

Read after write
One instruction reads the value written by another
<u>Example</u>: I3 and I2

Write after write
Two instructions write in the same memory location or register
It is important to keep the order of writes
<u>Examples</u>: I1 and I2

Read after read
Two instructions read in the same location
Often not a problem
<u>Example</u>: I3 and I4

Write after read
One instruction reads a value before it is written by another instructions
<u>Example</u>: I5 and I4

What is the Big Deal with Data Dependencies ?

- You can change the result of a program if you break a dependency
- **Example**: break a read-after-write

Input of I4:
b=4

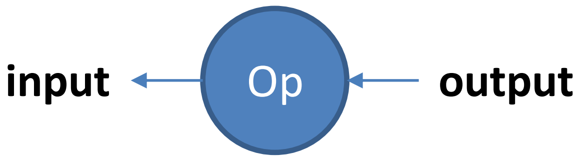
I1: a = 2
I2: a = 3
I3: b = a + 1
I4: c = a + 2
I5: a = b + a
I6: d = 6

I1: a = 2
I3: b = a + 1
I2: a = 3
I4: c = a + 2
I5: a = b + a
I6: d = 6

Input of I4:
b=3

- The compiler often needs to move calculation to optimize
- It needs to analyze dependencies to determine when it can (and can't) move calculation around
- Constraint: the compiler should not change the output of the program

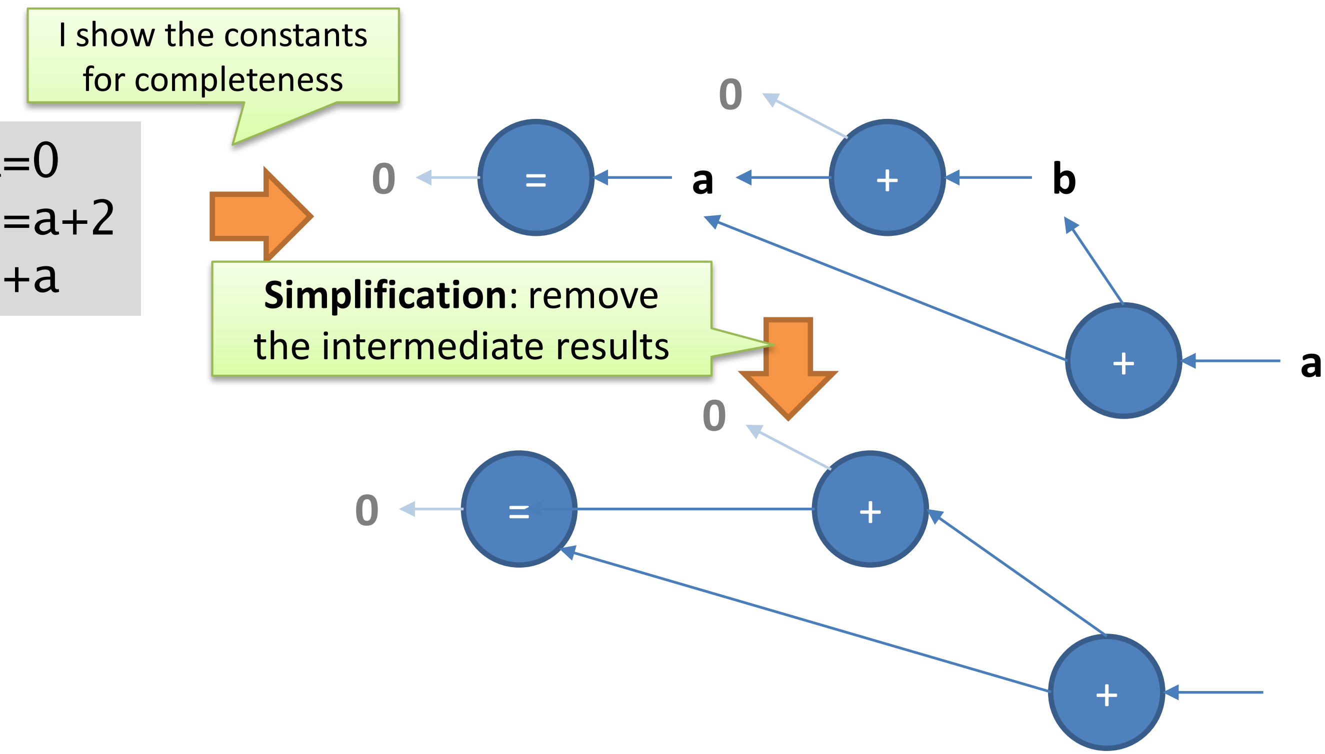
Graphical Representation of Data Dependencies



Expresses **dependency**: arrow
- From operation to input
- From output to operation

Example:

```
int a=0
int b=a+2
a = b+a
```



Def. 5: The SSA Form

- Issue with the IR of previous slides
  - Variables with the same names contain different data
  - It is hard to understand the dependency between instructions
  - Example:
    - I4 reads **a**, but it does not depend on I1
    - The **a** of I1 and the **a** of I2 are actually different things
- Solution: use Static Single Assignment Form (SSA)
  - Variables can only be assigned once
  - Variables with the same names represent the same data
  - Makes it easier to understand data dependencies
  - Developed in the 1980s
  - Now all IR are in SSA form
- Note: SSA introduce an instruction called “PHI” to solve name conflicts during branches
  - I won’t detail this today

```
I1: a = 2
I2: a = 3
I3: b = a + 1
I4: c = a + 2
I5: a = b + a
I6: d = 6
```

```
I1: a1 = 2
I2: a2 = 3
I3: b1 = a2 + 1
I4: c1 = a2 + 2
I5: a3 = b1 + a2
I6: d1 = 6
```

The 3 important graphs that define the IR

Control Flow Graph

Data Flow Graph

Function Call Graph

Graph 1: The Control-Flow Graph (CFG)

- Graph (V,E) where
  - V is the set of basic blocks of the program
  - E represents the execution order of basic blocks
- Example:

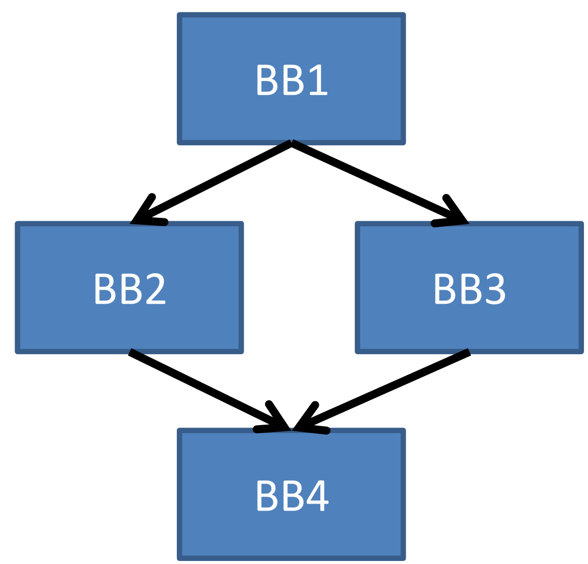
```
   a = 1
   b = 2
   if a<b goto L2
L1:
   b = a
   goto L3
L2:
   a = b
   goto L3
L3:
```

Basic Block 1

Basic Block 2

Basic Block 3

Basic Block 4



Graph 2: The Data Flow Graph (DFG)

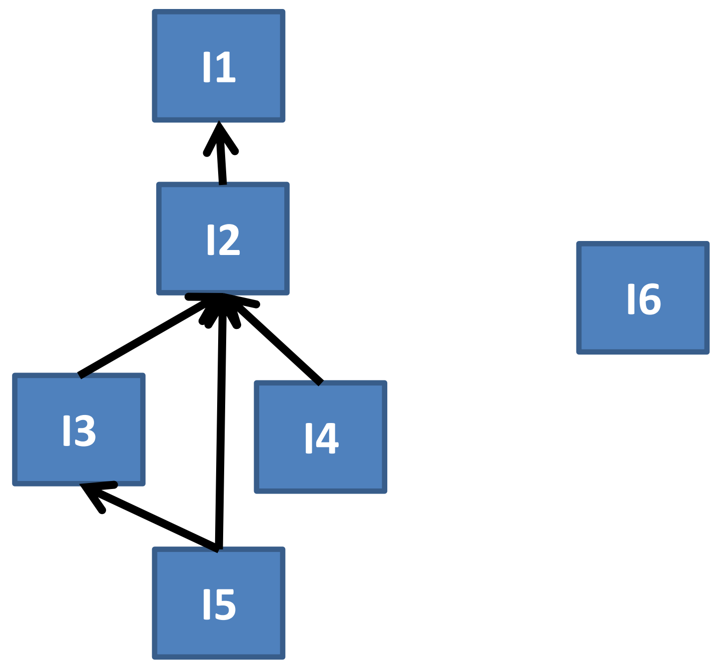
- Data dependences can be graphically displayed
- Definition of the data flow graph DFG = (V,E)
  - V: the set of instructions
  - E: the RAW and WAW dependencies
- Example:

Program

I1: a = 2
I2: a = 3
I3: b = a + 1
I4: c = a + 2
I5: a = b + a
I6: d = 6

SSA Form

I1: a1 = 2
I2: a2 = 3
I3: b1 = a2 + 1
I4: c1 = a2 + 2
I5: a3 = b1 + a2
I6: d1 = 6



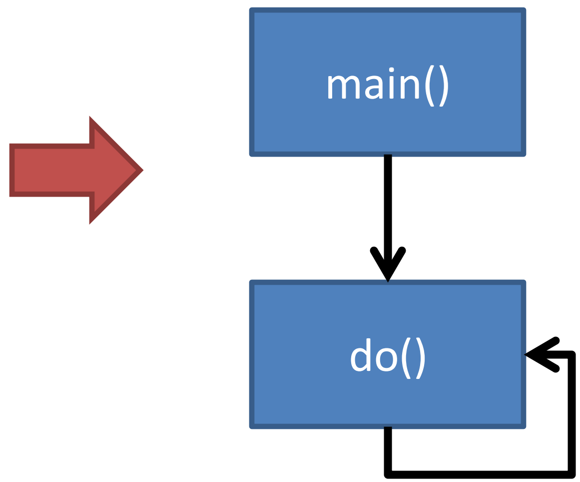
Nobody says **FCG**

Graph 3: The Function Call Graph (~~FCG~~)

- The third representation used by compilers is the function call graph
  - Graph (V,E)
  - V are functions of the program
  - E symbolize function calls
- Optimizations that involve the function-call graph is are called Inter-Procedural Optimizations (IPO)
  - Early compiler did not feature any IPA
- Example:

```
int main() {
  return do(6)
}
int do(int x) {
  if(x!=0) {
    do(i-1)
  } else {
    return 1;
  }
}
```

C Language



Put Everything Together

- Compilers analyze program using IR
  - IR: Intermediate Representation
  - More expressive that text-form: contains semantic information
- The IR consists of
  - Operations and data types
  - Control flow graph and function call graph: express the order of execution
  - Data flow graph: expresses the dependency between instructions
- The SSA representation
  - SSA: Single Static Assignment
  - Makes data dependency analysis easier

EXAMPLE OF OPTIMIZATIONS

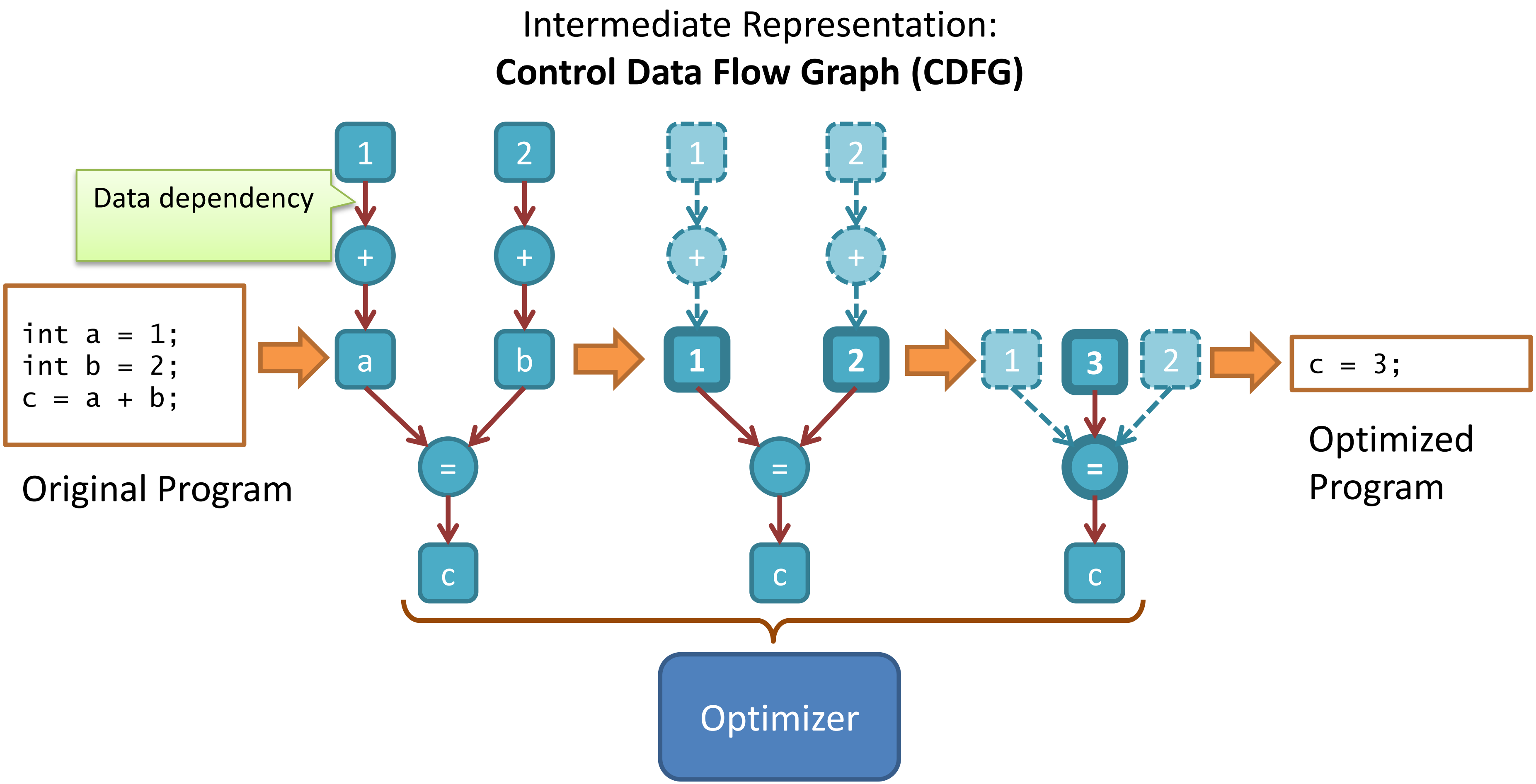
Overview of Optimizations

- Optimizations may change
  - The control flow graph
  - The data flow graph (without breaking dependencies)
  - The function call graph
- Some optimizations are always efficient
- Some other are double edged
  - **Depending on the program / target processor an optimization can actually reduce performance**
- Current compilers almost only optimize single-thread, Von Neumann programs
  - Because most language follow this paradigms
  - Especially, there exist few efficient optimization for threaded programs
- Compilers for other architecture (e.g. GPU) exist, but they provide with very few optimization

Major Targets for Optimizations

- **Calculations**
  - Reduce the amount of calculations
  - Use the computations units of the processor more efficiently (e.g. SIMD units)
- **Flow / Order of execution**
  - Change the order of execution to allow better single-thread parallelism (SIMD, out-of-order execution)
- **Data**
  - Change the order the program access to the memory
  - Often try to take advantage of caches (If any)

Example 1: Constant Propagation



Example 2: Function Inlining

```
int inc(int x) {
  return x + 1;
}
int do(int x) {
  if (x!=0) {
    return inc(x-1);
  } else {
    return x;
  }
}
int main() {
  return do(6);
}
```

```
int inc(int x) {
  return x + 1;
}
int do(int x) {
  if (x!=0) {
    return x-1+1;
  } else {
    return x;
  }
}
int main() {
  return do(6);
}
```

Removes the call to acc()
Saves execution time:

- a function call requires several control-flow instructions
- these instructions disappear

Very efficient, especially in object-oriented languages where programmers often implement small methods

Increases the size of programs. May negatively affect power consumption and instruction cache usage on some processors (especially embedded)

Example 3: The Power of Combination

Optimizations are even more powerful when combined !

```
int inc(int x) {
  return x + 1;
}
int do(int x) {
  if (x!=0) {
    return inc(x-1);
  } else {
    a = inc(x);
    return a – 1;
  }
}
int main() {
  return do(6);
}
```

Inlining

```
int inc(int x) {
  return x + 1;
}
int do(int x) {
  if (x!=0) {
    return x-1+1;
  } else {
    a = x+1;
    return a-1;
  }
}
int main() {
  return do(6);
}
```

Constant Propagation

```
int inc(int x) {
  return x + 1;
}
int do(int x) {
  return x;
}
int main() {
  return do(6);
}
```

Inlining

```
int inc(int x) {
  return x + 1;
}
int do(int x) {
  return x;
}
int main() {
  return 6;
}
```

Only one instruction left !

Combination in a Real Compiler

```
$> opt [...] -O1 -debug-pass=Structure
Pass Arguments:  [...]
Target Library Information
Target Data Layout
No Alias Analysis (always returns 'may' alias)
Type-Based Alias Analysis
Basic Alias Analysis (stateless AA impl)
  ModulePass Manager
    Global Variable Optimizerl loops
    Interprocedural Sparse Conditional Constant Propagation
    Dead Argument Elimination
    FunctionPass Manager Code Motion
      Combine redundant instructions
      Simplify the CFG
    Basic CallGraph Construction
    Call Graph SCC Pass Manager
      Remove unused exception handling info
      Inliner for always_inline functions
      Deduce function attributes
      FunctionPass Manager
        Scalar Replacement of Aggregates (SSAUp)
        Dominator Tree Construction
        [...]
        Loop Pass Manager
          Canonicalize natural loops
          [...]
        Combine redundant instructions
        Scalar Evolution Analysis
        [...]
    Strip Unused Function Prototypes
    FunctionPass Manager
      Preliminary module verification
      Dominator Tree Construction
      Module Verifier
    Bitcode Writer
```

Option “O1” of LLVM

About <u>40 optimizations</u>

With many repetitions

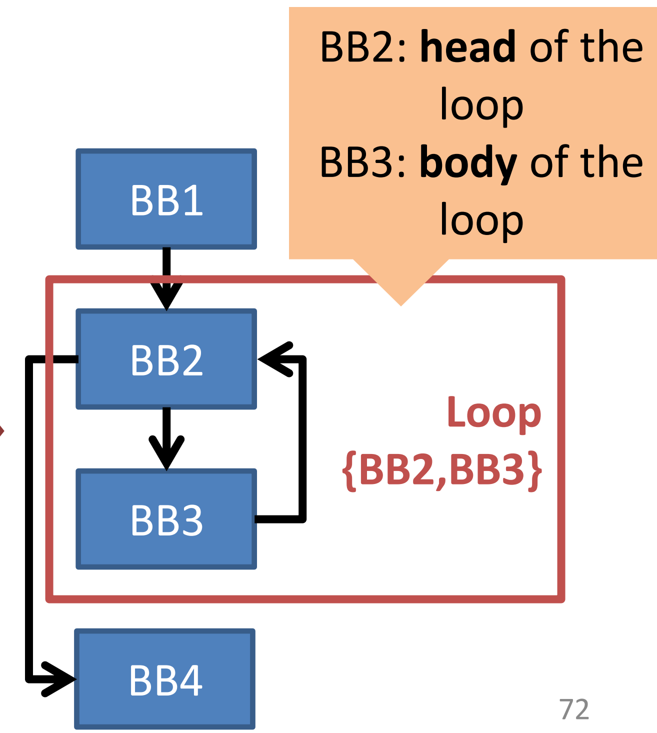
LOOP OPTIMIZATIONS

What is a Loop

- A loop is a piece of code that may be executed several times
- It corresponds to a cycle in the data flow graph (DFG)
- In compilation we consider the following constraints:
  - a single entry point
  - we also often only allow a single exit point
- Example:

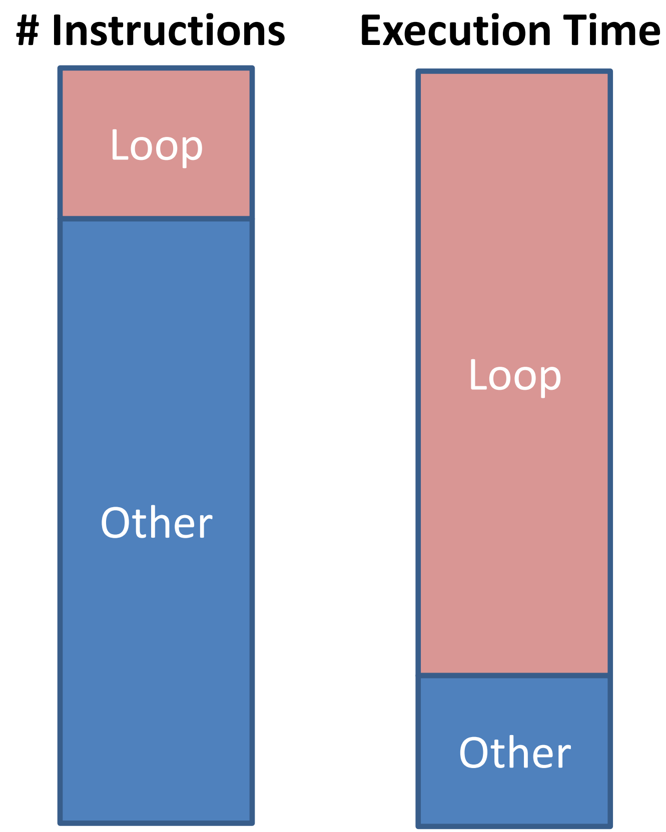
```
for(i=0; i<MAX; i++) {
  a[i]++;
}
```

```
  i=0                      BB1
L1:                        BB2
  if i>=MAX goto L3
L2:                        BB3
   a1 = load a + 1
   a2 = a1 + 1
   store a2, a+1
   i = i - 1
   goto L1
L3:                        BB4
```



Why are Loops Important ?

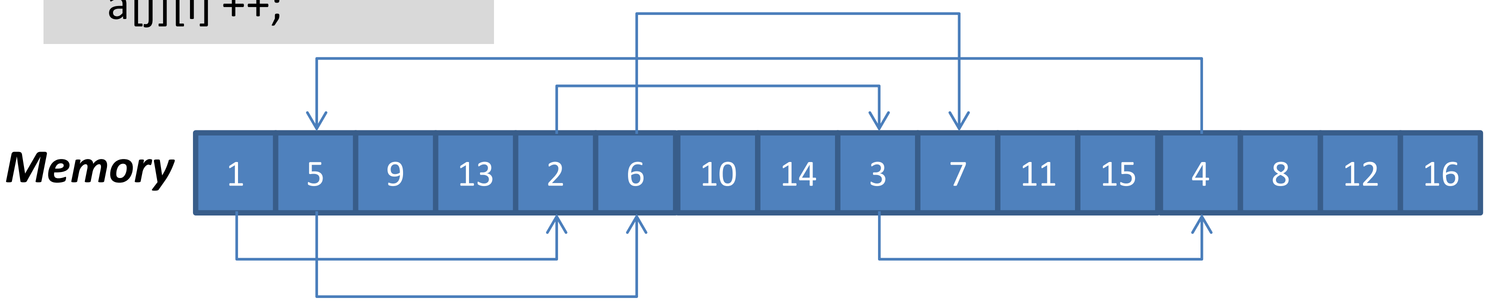
- Rule of “80% / 20%”
  – Loops usually count for **20%** of the code of a program
  – But programs usually spend more than **80%** of their times in loops
- Example:
  – Let us consider that we divide by two the execution of a given piece of code
  – Case 1: the code is outside a loop
    - total time = 20% / 2 + 80% = 90% of the original program
  – Case 2: the code is inside a loop
    - total time = 20% + 80% / 2 = 60% of the original program !



Example of Loop Optimization
Nest Interchange

```
int a[4][4];
for (int i=0; i<4; i++)
  for (int j=0; j<4; j++)
    a[j][i] ++;
```

Accesses are not sequential
Processor caches are not designed to handle such cases
All memory accesses will miss !




```
int a[4][4];
for (int j=0; j<4; j++)
  for (int i=0; i<4; i++)
    a[j][i] ++;
```

If we swap the “for”, the access pattern becomes sequential
This is the best access pattern for caches.
We miss only when we reach a new cache line
On my computer: 5 times faster !

CONCLUSION

Why Should I Study Optimizing Compilers ?

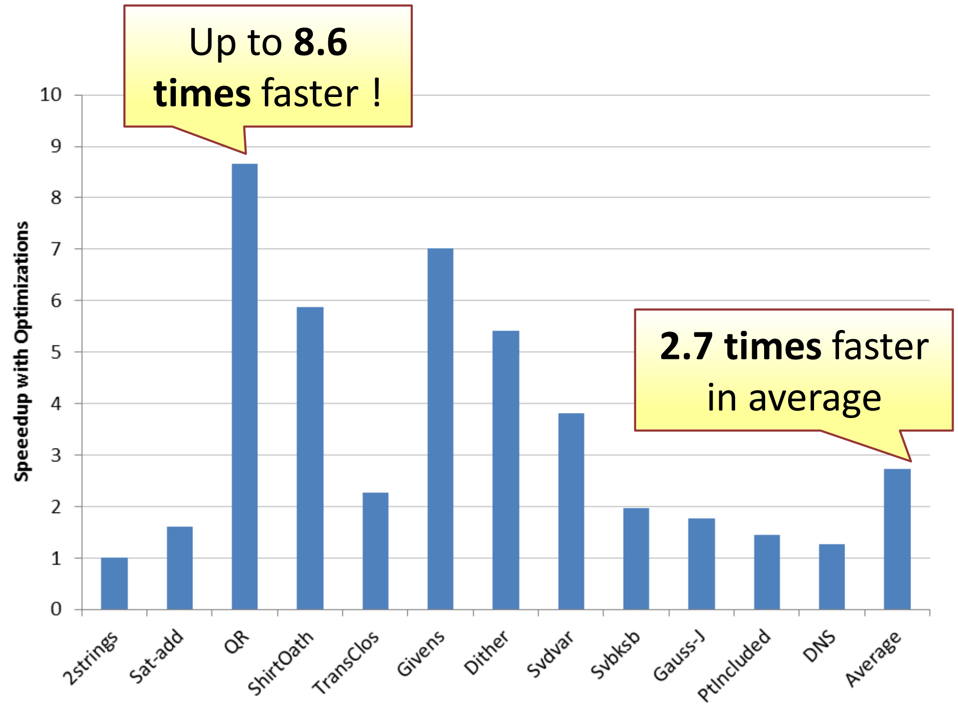
- If you don’t program, you don’t need
- Otherwise, it is important to understand
  - **What kind of code your compiler expects**
  - **What kind of code your processor is designed for**
- It is often possible to reduce the execution time by several times with simple code modifications !

Two Levers for Optimizations

- Compiler options
  - O2, O3
  - Vectorization options
- Code transformations (by hand)
  - **First objective:** Change the order and nature of operations by hand
  - **Second objective:** Make it easier for the compiler to optimize

Example of Loop Optimizations

- Loop unrolling
- Loop fusion
- Loop fission
- Loop collapsing
- Loop unroll and jam
- Polyhedral Optimizations



Data from INRIA laboratory, France
Optimizations are made automatically by some research algorithms.

Conclusion

- Computer programs are written in language that the processor doesn’t understand
  - The **compiler** does the translation
- But a compiler is more than just a translator
  - It produces fast code
  - To do so, it carries out **optimizations**
- The compiler uses powerful **internal representation** to analyze the code
  - Data dependency analysis
  - Control flow analysis
- Optimizations are often double-edged
  - They may reduce performances if misused
  - Optimizations should be tailored to the target processor
- The most important optimization targets are loops
  - **Rule of 80% / 20%**
  - We can expect several times performance increase !
- In practice optimization is a fine mix of
  - manual-tuning
  - compiler options setting

THANK YOU VERY MUCH