Lab 1: Oriented with performance measurement

Hung-Wei Tseng

Lab 1

- Performance equation
- Amdahl’s Law
- Programming assignment — get familiar with C/C++

Q2-Q14: CPU Performance Equation

$$Performance = \frac{1}{Execution\ Time}$$

$$Execution\ Time = \frac{Instructions}{Program} \times \frac{Cycles}{Instruction} \times \frac{Seconds}{Cycle}$$

$$ET = IC \times CPI \times CT$$

- IC (Instruction Count)
  - ISA, Compiler, algorithm, programming language, **programmer**
- CPI (Cycles Per Instruction)
  - Machine Implementation, microarchitecture, compiler, application, algorithm, programming language, **programmer**
- Cycle Time (Seconds Per Cycle)
  - Process Technology, microarchitecture, **programmer**

Recap: Performance equation (round 2)

- Consider the following c code snippet and x86 instructions implement the code snippet

C	x86

`for(i = 0; i < count; i++) {`
`    s += a[i];`
`}`
	`.L3:`
	`movslq  (%rdi), %rdx`
	`addq    $4, %rdi`
	`addq    %rdx, %tax`
	`cmpq    %rcx, %rdi`
	`jne     .L3`

Comparing the case where `count` equal to 1,000,000,000 and 2,000,000,000, what factor in performance equation would change?

A. IC
B. CPI
C. CT
D. IC & CPI
E. IC & CT

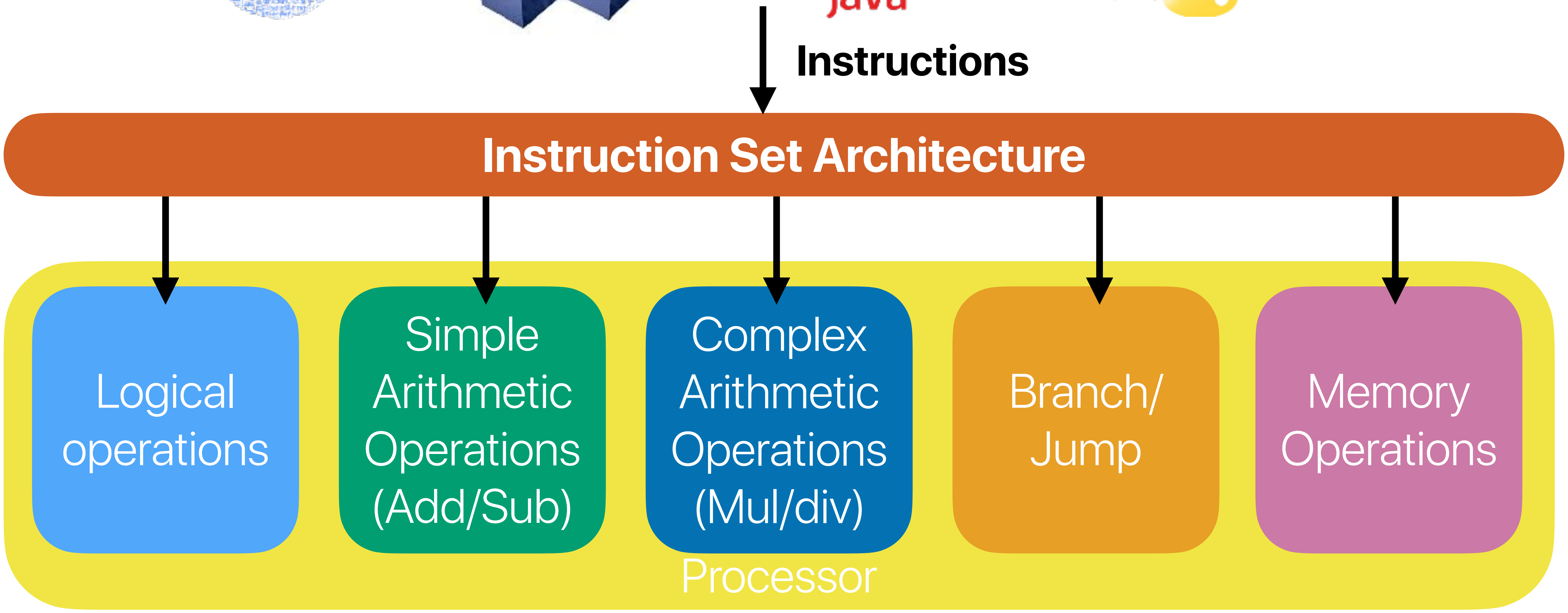
Recap: What does the programmer change?

- By adding the “sort” in the following code snippet, what the programmer changes in the performance equation to achieve **better** performance?

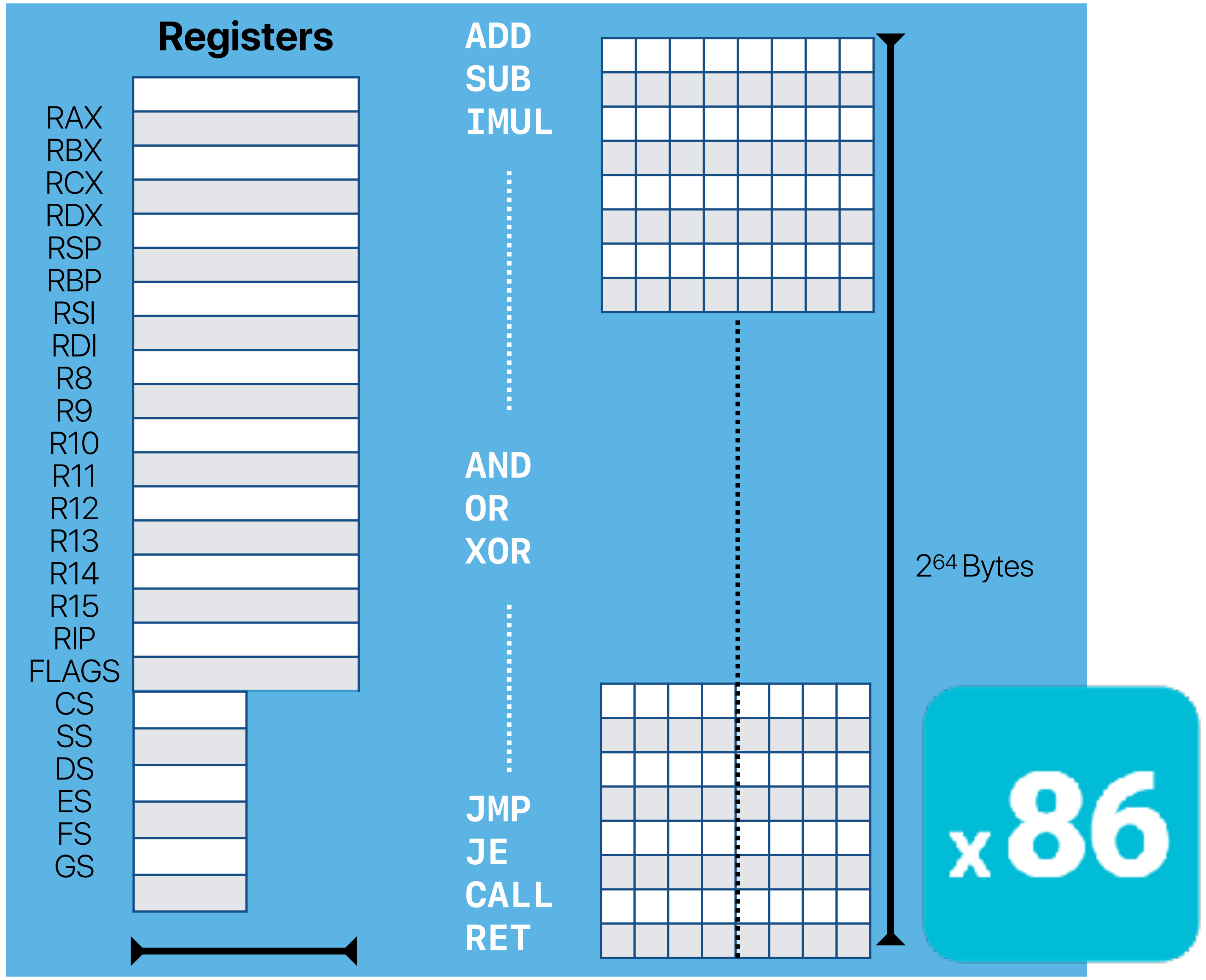
```
std::sort(data, data + arraySize);

for (unsigned c = 0; c < arraySize*1000; ++c) {
        if (data[c%arraySize] >= INT_MAX/2)
            sum ++;
    }
}
```

A. $\boxed{\text{CPI}}$
B. IC ← **programmer changes IC as well, but not in the positive direction**
C. CT
D. IC & CPI
E. CPI & CT

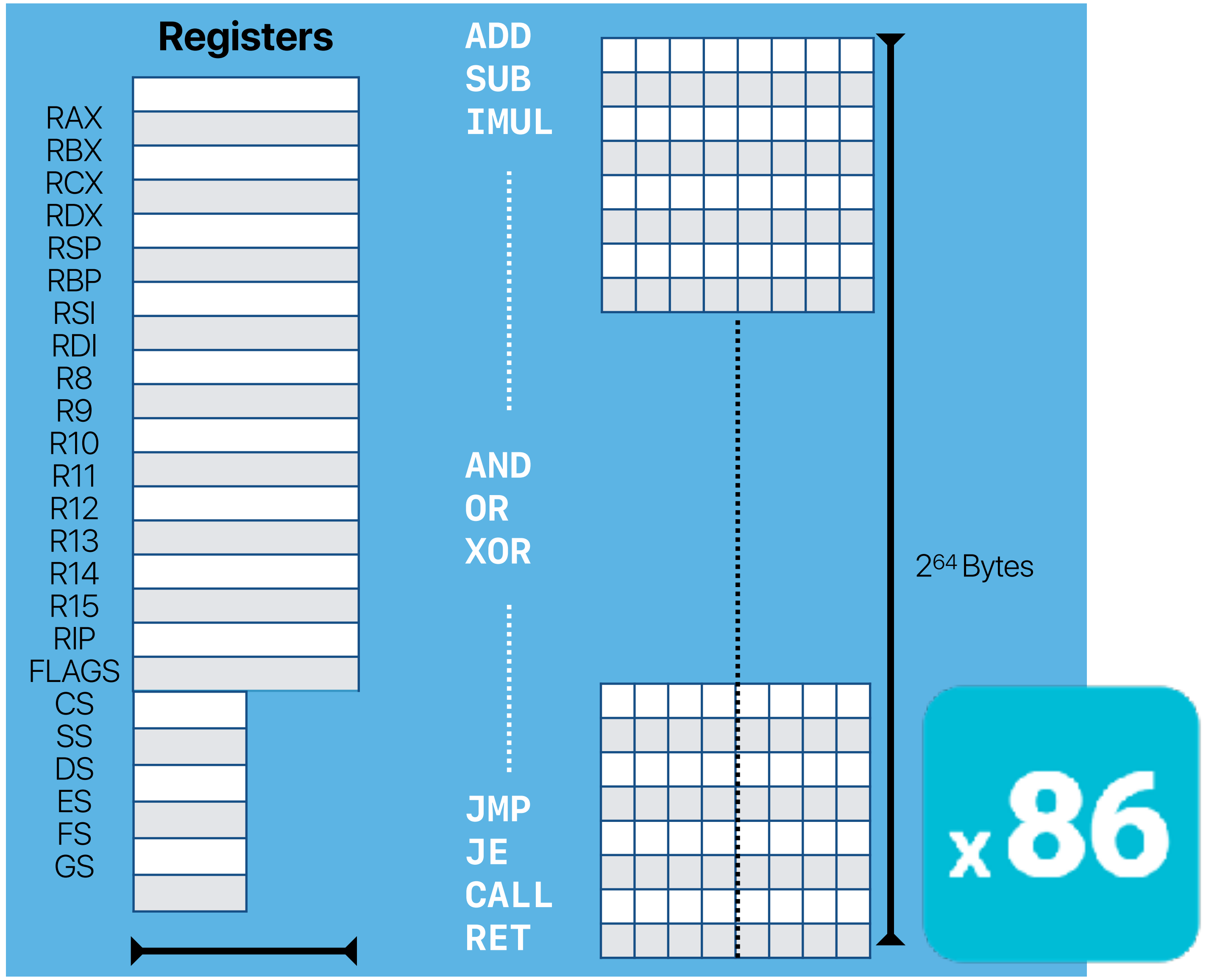
Q8—Q10: Microprocessor — a collection of functional units



The “abstraction”



64-bit



64-bit

Registers

16bit	32bit	64bit	Description	Notes
AX	EAX	RAX	The accumulator register	These can be used more or less interchangeably
BX	EBX	RBX	The base register	
CX	ECX	RCX	The counter	
DX	EDX	RDX	The data register	
SP	ESP	RSP	Stack pointer	
BP	EBP	RBP	Pointer to the base of stack frame	
	Rn	RnD	General purpose registers (8-15)	
SI	ESI	RSI	Source index for string operations	
DI	EDI	RDI	Destination index for string operations	
IP	EIP	RIP	Instruction pointer	
	FLAGS		Condition codes	

Start with this simple loop

```
xorl %eax, %eax
                cmpl  $100, %eax
                jne .L2
                               addl  $1, %eax
for(i = 0; i < 100; i++)
{
  sum += A[i];
}
   .L2: addl  (%ecx,%eax,4), %edx
```

```
      xorl %eax, %eax
.L2:  addl  (%ecx,%eax,4), %edx
      addl  $1, %eax
      cmpl  $100, %eax
      jne .L2
```

Data types in “x86 instructions” vs “C/C++”

C declaration	x86	x86 instruction suffix	x86-64 Size (Bytes)	functional unit
char	Byte	b	1	Integer
short	Word	w	2	
int	Double word	l	4	
unsigned	Double word	l	4	
long int	Quad word	q	8	
unsigned long	Quad word	q	8	
char *	Quad word	q	8	
float	Single precision	s	4	floating point units
double	Double precision	d	8	
long double	Extended precision	t	16	

MOV and addressing modes

- MOV instruction moves data between registers/memory
- MOV instruction has many address modes

instruction	meaning	arithmetic op	memory op
`movl $6, %eax`	`R[eax] = 0x6`	1	0
`movl .L0, %eax`	`R[eax] = .L0`	1	0
`movl %ebx, %eax`	`R[ebx] = R[eax]`	1	0
`movl -4(%ebp), %ebx`	`R[ebx]  = mem[R[ebp]-4]`	2	1
`movl (%ecx,%eax,4), %eax`	`R[eax]  = mem[R[ebx]+R[edx]*4]`	3	1
`movl -4(%ecx,%eax,4), %eax`	`R[eax]  = mem[R[ebx]+R[edx]*4-4]`	4	1
`movl %ebx, -4(%ebp)`	`mem[R[ebp]-4] = R[ebx]`	2	1
`movl $6, -4(%ebp)`	`mem[R[ebp]-4] = 0x6`	2	1

Arithmetic Instructions

- Operands can come from either registers or a memory location

instruction	meaning	arithmetic op	memory op
`subl $16, %esp`	`R[%esp] = R[%esp] - 16`	1	0
`subl %eax, %esp`	`R[%esp] = R[%esp] - R[%eax]`	1	0
`subl -4(%ebx), %eax`	`R[eax] = R[eax] - mem[R[ebx]-4]`	2	1
`subl (%ebx, %edx, 4), %eax`	`R[eax] = R[eax] - mem[R[ebx]+R[edx]*4]`	3	1
`subl -4(%ebx, %edx, 4), %eax`	`R[eax] = R[eax] - mem[R[ebx]+R[edx]*4-4]`	3	1
`subl %eax, -4(%ebx)`	`mem[R[ebx]-4] = mem[R[ebx]-4]-R[eax]`	3	2

Branch instructions

- x86 use condition codes for branches
  - Arithmetic instruction sets the flags
  - Example:

    `cmp %eax, %ebx` #computes %eax-%ebx, sets the flag

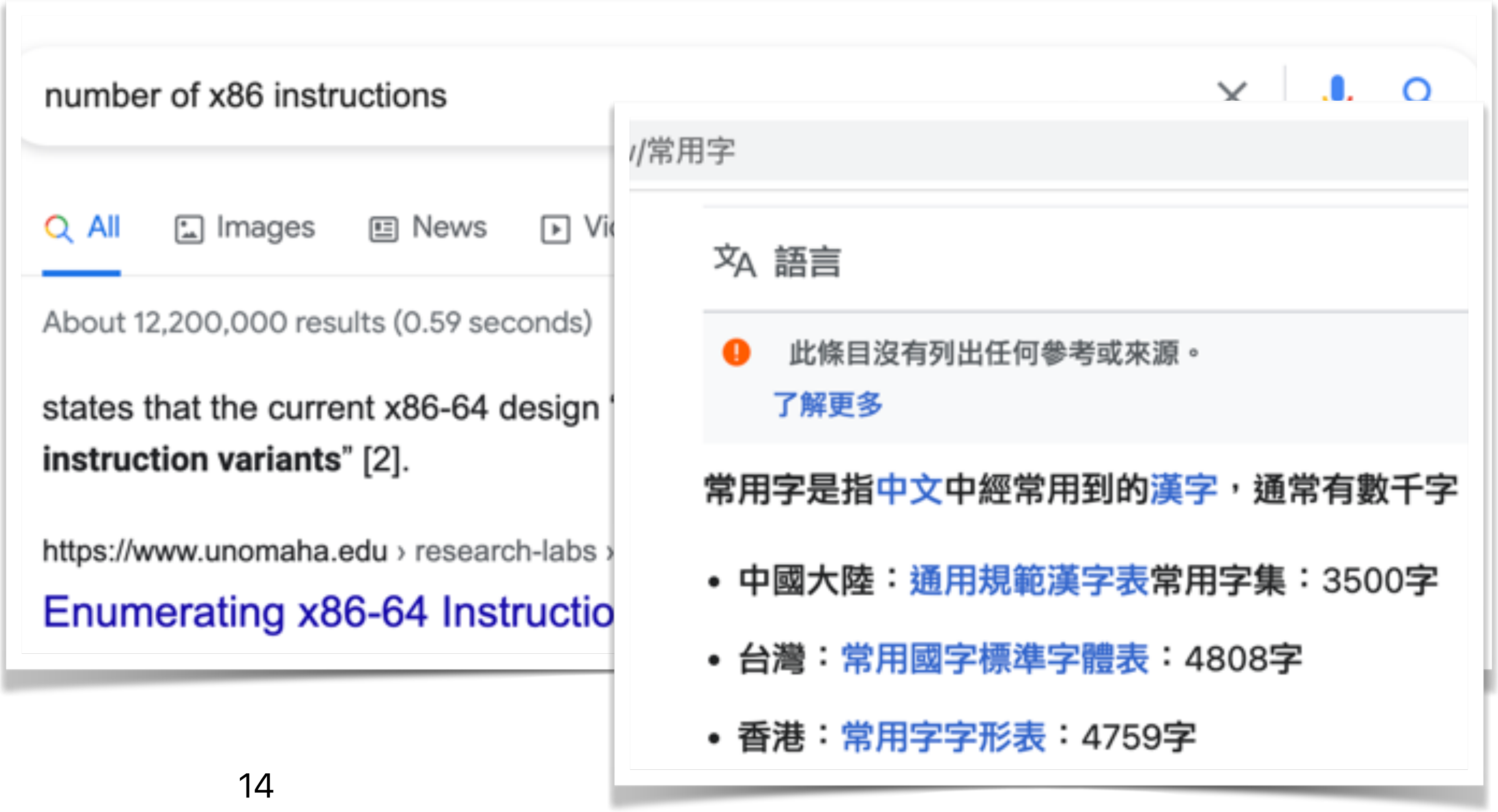
    `je <location>` #jump to location if equal flag is set
- Unconditional branches
  - Example:

    `jmp <location>` #jump to location

Looking at these complex examples

```
  movl -4(%ecx,%eax,4), %eax: R[eax] = mem[R[ebx]+R[edx]*4-4]
subl -4(%ebx, %edx, 4), %eax: R[eax] = R[eax]
                                     - mem[R[ebx]+R[edx]*4-4]
```

- Lots of commonalities
- Repetitive operations
  - memory accesses
  - multiplications
  - subtractions



Can we rewrite it?

```
subl -4(%ebx, %edx, 4), %eax: R[eax] = R[eax]
                                      - mem[R[ebx]+R[edx]*4-4]
```

Version A

```
subl -4(%ebx, %edx, 4), %eax
```

Version B

```
movl %edx, %r8d
shl %r8d, $2            // %r8d=%r8d*4
subl %r8d, $4           // %r8d=%r8d-4
addl %r8d, %rbx         // %r8d=%r8d+%rbx
movl %r8d, 0(%r8d)      // %r8d=mem[%r8d]
subl %rax, %r8d         // %rax=%rax-%r8d
```

Translate from C to Assembly

- gcc: gcc [options] [src_file]
  - compile to binary
    - gcc -o foo foo.c
  - compile to assembly (assembly in foo.s)
    - gcc -S foo.c
  - compile with debugging message
    - gcc -g -S foo.c
  - optimization
    - gcc -On -S foo.c
      - n from 0 to 3 (0 is no optimization)
- We're skipping the detail for now — You SHOULD HAVE experienced a lot of these during Lab 2/3

Q11—Q14: Recap — what programmer changed?

A

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

B

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

$O(n^2)$	**Complexity**	$O(n^2)$
Same	**Instruction Count?**	**Same**
Same	**Clock Rate**	**Same**
Better	**CPI**	**Worse**

Q10: Recap: Revisited the demo with compiler optimizations!

- gcc has different optimization levels.
  - -O0 — no optimizations
  - -O3 — typically the best-performing optimization

A

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

B

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

Start the presentation to see live content. Still no live content? Install the app or get help at **PollEv.com/app**

Q15—Q17: Lessons learned from Amdahl's Law

$$Speedup_{enhanced}(f, s) = \frac{1}{(1-f) + \frac{f}{s}}$$

f is the fraction of "execution time" — neither of the IC, CPI or CT

- Corollary #1: Maximum speedup
- Corollary #2: Make the common case fast
  - Common case changes all the time
- Corollary #3: Optimization is a moving target
- Corollary #4: Exploiting more parallelism from a program is the key to performance gain in modern architectures
- Corollary #5: Single-core performance still matters

$$Speedup_{max}(f, \infty) = \frac{1}{(1-f)}$$

$$Speedup_{max}(f_1, \infty) = \frac{1}{(1-f_1)}$$

$$Speedup_{max}(f_2, \infty) = \frac{1}{(1-f_2)}$$

$$Speedup_{max}(f_3, \infty) = \frac{1}{(1-f_3)}$$

$$Speedup_{max}(f_4, \infty) = \frac{1}{(1-f_4)}$$

$$Speedup_{parallel}(f_{parallelizable}, \infty) = \frac{1}{(1-f_{parallelizable})}$$

$$Speedup_{parallel}(f_{parallelizable}, \infty) = \frac{1}{(1-f_{parallelizable})}$$

Amdahl's Law on Multiple Optimizations

- We can apply Amdahl's law for multiple optimizations
- These optimizations must be dis-joint!
  - If optimization #1 and optimization #2 are dis-joint:

f_{Opt1}	f_{Opt2}	$1\text{-}f_{Opt1}\text{-}f_{Opt2}$

$$Speedup_{enhanced}(f_{Opt1}, f_{Opt2}, s_{Opt1}, s_{Opt2}) = \frac{1}{(1 - f_{Opt1} - f_{Opt2}) + \frac{f\_Opt1}{s\_Opt1} + \frac{f\_Opt2}{s\_Opt2}}$$

  - If optimization #1 and optimization #2 are not dis-joint:

$f_{OnlyOpt1}$	$f_{OnlyOpt2}$	$f_{BothOpt1Opt2}$	$1\text{-}f_{OnlyOpt1}\text{-}f_{OnlyOpt2}\text{-}f_{BothOpt1Opt2}$

$$Speedup_{enhanced}(f_{OnlyOpt1}, f_{OnlyOpt2}, f_{BothOpt1Opt2}, s_{OnlyOpt1}, s_{OnlyOpt2}, s_{BothOpt1Opt2})$$
$$= \frac{1}{(1 - f_{OnlyOpt1} - f_{OnlyOpt2} - f_{BothOpt1Opt2}) + + \frac{f\_BothOpt1Opt2}{s\_BothOpt1Opt2} + \frac{f\_OnlyOpt1}{s\_OnlyOpt1} + \frac{f\_OnlyOpt2}{s\_OnlyOpt2}}$$

Remember the demo program today

```
fp = fopen(argv[1],"rb");
fread(cpu_idata,sizeof(float),numElements,fp);
fclose(fp);

qsort(cpu_odata, numElements, sizeof(float), compare);
```

- We applied the following optimizations
  - GPU to **only** accelerate `qsort`
  - Storing data to SSD to **only** accelerate `fread`

Amdahl's Law on Multiple Optimizations

- We can apply Amdahl's law for multiple optimizations
- These optimizations must be dis-joint!
  - If optimization #1 and optimization #2 are dis-joint:

f_{Opt1}	f_{Opt2}	$1-f_{Opt1}-f_{Opt2}$

$$Speedup_{enhanced}(f_{Opt1}, f_{Opt2}, s_{Opt1}, s_{Opt2}) = \frac{1}{(1 - f_{Opt1} - f_{Opt2}) + \frac{f\_Opt1}{s\_Opt1} + \frac{f\_Opt2}{s\_Opt2}}$$

- We applied the following optimizations
  - Opt1 — GPU
    — **only** accelerate `qsort` — $\mathbf{f_{opt1}}$
  - Opt2 — Storing data to SSD
    — **only** accelerate `fread` — $\mathbf{f_{opt2}}$

Programming assignment

Why C/C++ programming?

- The only pathway to performance programming

Lab 1: the main function & basic I/O

```
#include <fstream>
#include <iostream>

int main(int argc, char *argv[])
{
   std::ofstream ofs ("hello.txt", std::ofstream::out);
   ofs << "Hello CSE142L!\n";
   ofs.close();
   std::cout << "Execution Complete" << std::endl;
   return 0;
}
```

Hints to Lab 1 PA

- You need to manipulate the argv array
- You need to find out how to convert "ASCII" based characters into integers

Problems in Lab 1

Turn in the lab

- Notebook — PDF generated from going through lab.ipydb datahub
    - https://www.gradescope.com/courses/564383/assignments/3005902/
- Programming assignment
    - https://www.gradescope.com/courses/564383/assignments/3005904

Announcement

- Lab 1 due 8/17 midnight through gradescope
- Lab 2 will be released next Tuesday
- Find the "right" staff and the "right time" to ask questions
  - If an office hour is for 142L, we don't address 142 issues there. Likewise for 142 hours
  - The TA cannot help CSE142L. Tutors do not help 142. That's not their duties
  - Please review the first lecture regarding the slides for "I need help"

Computer
Science &
Engineering

142L

つづく

How my “C code” becomes a “program”