HIGH PERFORMANCE COMPUTING FOR SCIENCE AND ENGINEERING

LECTURE 17

Fabian Wermelinger
Harvard University

CS205

Tuesday, April 4th 2023

LAST TIME

- RISC and CISC computers
- Instruction set architecture (ISA): the vocabulary of the computer. Different classes and history.
- Operation counts and comparison with different applications.
- Interpretation of memory addresses.
- Addressing modes and instruction encoding.
- Processor pipelining examples based on hypothetical ISA.

TODAY

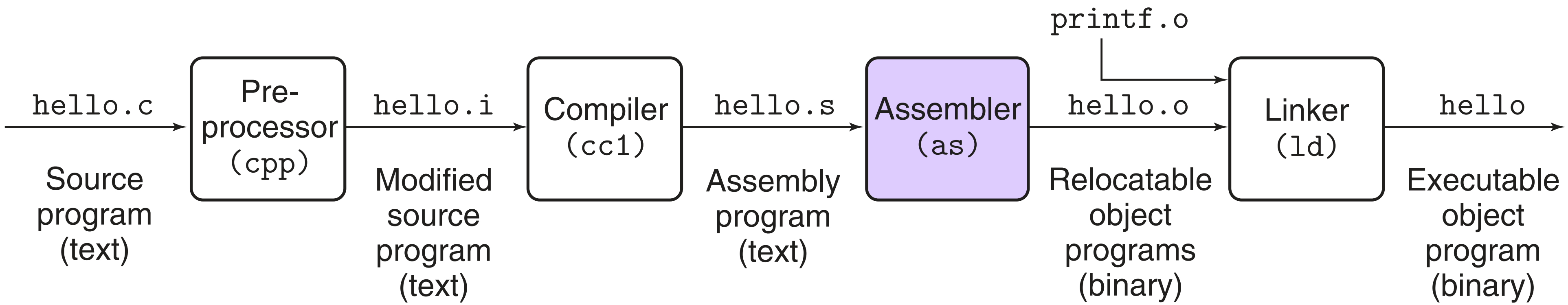
Main topic: ***`x86-64` assembly basics and floating-point operations***

Details:

- Basics of assembly language (closest to machine code, not portable).
- AT&T and Intel syntax.
- Main differences between `IA32` (32-bit) and `x86-64` ISA's.
- General purpose registers.
- `x86-64` operand forms and addressing modes.
- Procedure calls and argument passing
- Instruction-level parallelism and loop-unrolling.
- `x86-64` floating-point operations and media registers.
- Fused multiply-add extensions (FMA)

ASSEMBLY STAGE IN COMPILATION PHASE

Recall: stages of the compiler driver (lecture 2):



- The *assembler* is a program in the pipeline that takes *assembly code* as input and generates *machine code* on output.
- On GNU Linux this program is called as (or sometimes called GAS for GNU assembler).
- The assembly code contains your program *in the vocabulary of the ISA* represented in human readable form.
- The assembler generates (encodes) *binary machine code* based on the rules of the ISA.
- Understanding the basics of assembly is useful to investigate the quality of a program generated by the compiler. *Assuming the compiler always does a good job is not a good assumption.*

ASSEMBLY LANGUAGE

- Writing assembly can be cumbersome and you will not need it most of the time. *The skill to read assembly is useful*.
- As of the previous slide the compiler *usually* does a good job, but if something does not behave as you'd expect (*from the performance perspective*) then only the assembly will tell you the truth about what is going on.
- The *addressing modes* supported by the ISA are revealed in assembly.
- You can identify the architecture class (load-store, register-memory).
- Figure out if the compiler issued FMA instructions where you would expect it to (when targeting peak performance).
- See whether the compiler *vectorized* code (SIMD instructions).
- Knowing assembly is also useful for debugging (lab 6).
- In the extreme case you can write assembly yourself to produce the exact code you want (God mode 🤓).
- ***Assembly code is machine (ISA) specific and not portable!***

BASICS OF x86-64 ASSEMBLY

Assembly syntax:

- There are two flavors of assembly syntax: *Intel* and *AT&T*.
- Assume we have the following function

```c
int f(int x) { return x * x + 1; }
```

AT&T syntax:

```
opname source, destination
```

Assembly code for the function f above:

```
f:
    imull   %edi, %edi
    leal    1(%rdi), %eax
    ret
```

- Registers have a "%" prefix.
- opname's have a *type* suffix.
- Memory address in *parenthesis*.
- Destination can be a source as well.

Intel syntax:

```
opname destination, source
```

Assembly code for the function f above:

```
f:
    imul   edi, edi
    lea    eax, [rdi+1]
    ret
```

- Registers only denoted by name.
- opname's do not have a type indicator.
- Memory address in *square brackets*.
- Destination can be a source as well.

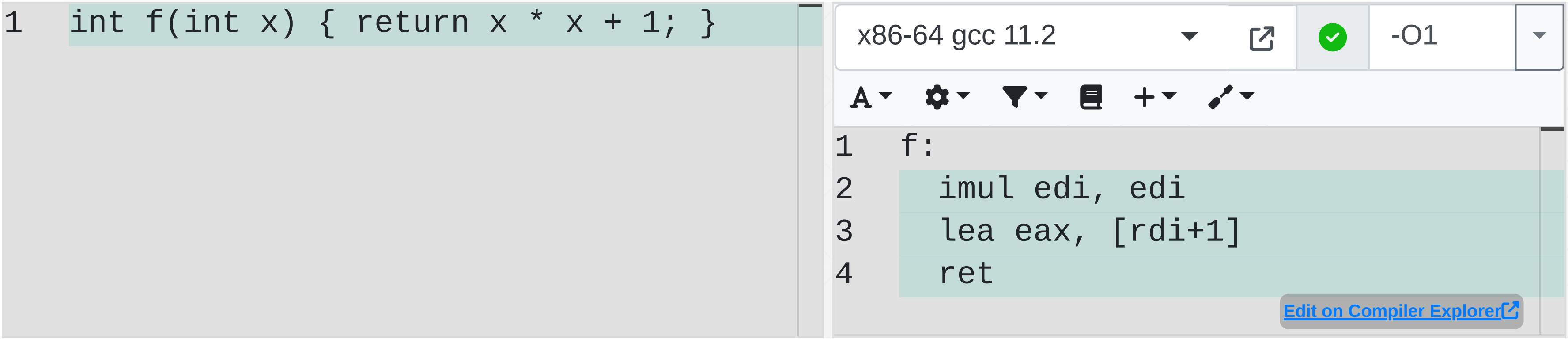
BASICS OF x86-64 ASSEMBLY

Emit assembly using compiler:

- You can emit assembly with GCC or Clang using the `-S` flag.
- The output is a file with "`.s`" suffix you can inspect and modify with an editor.
- You could assemble this file to machine code using the `as` assembler in Linux:

```
$ gcc -S -O1 f.c
$ as f.s
$ file a.out
a.out: ELF 64-bit LSB relocatable, x86-64, version 1 (SYSV), not stripped
```

- A *very* useful and quick alternative to check assembly of a code snippet is https://godbolt.org/ (Compiler Explorer):

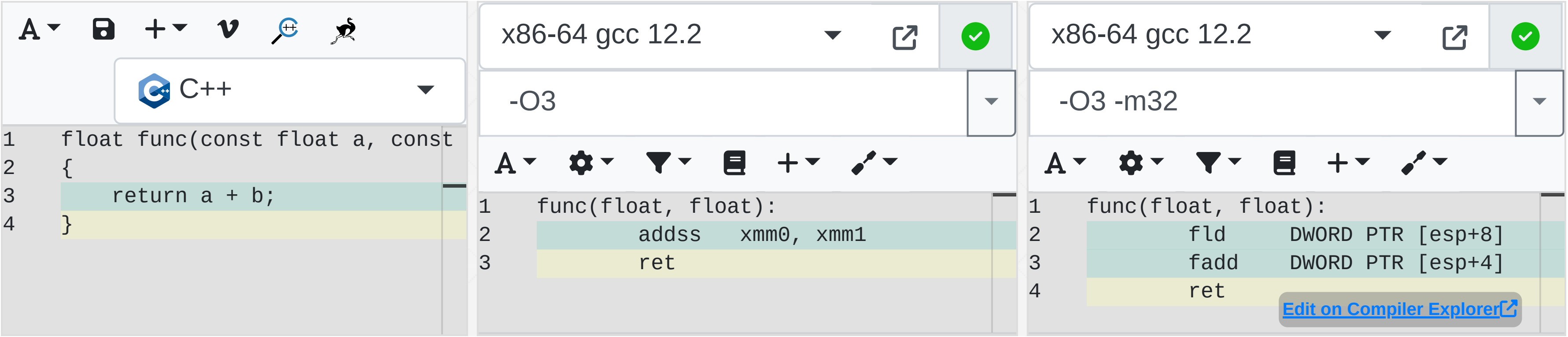


This example: https://godbolt.org/z/YjjvxeTzz

BASICS OF x86-64 ASSEMBLY

Main features of x86-64 compared to IA32 (32-bit):

- *Pointers and `long` integers are 64-bit long*. Integer arithmetic operations support 8, 16, 32 and 64-bit data types.
- The set of general-purpose registers is expanded from 8 to *16*.
- Much of the program state is held in registers rather than on the stack. Integer and pointer procedure arguments (up to 6) are passed via registers. *Some procedures do not need to access the stack at all.*
- Conditional operations are implemented using conditional move instructions when possible, yielding better performance than traditional branching code.
- Floating-point operations are implemented using a register-oriented instruction set, rather than a stack-based approach (see x87 for the floating-point ISA used before).



This example: https://godbolt.org/z/nhevY99da

BASICS OF x86-64 ASSEMBLY

Sizes of standard data types with x86-64:

C declaration	Intel data type	GAS suffix	x86-64 size (bytes)
`char`	Byte	`b`	1
`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
`double`	Double precision	`d`	8
`long double`	Extended precision	`t`	16

You should avoid using `long double` as it is ***not portable*** and uses the legacy x87 floating-point format used in `IA32`. This format is typically not as performant as single and double precision data.

EXAMPLE ASSEMBLY FOR `IA32` AND `x86-64`

Consider the simple function:

```
long int simple_l(long int *xp, long int y)
{
    long int t = *xp + y;
    *xp = t;
    return t;
}
```

With x86 machine code you get (32bit):

```
gcc -O2 -S -masm=intel -m32 simple_l.c
```

```
simple_l:
    mov edx, DWORD PTR 4[esp] ; get the pointer xp
    mov eax, DWORD PTR [edx]  ; dereference *xp
    add eax, DWORD PTR 8[esp] ; add *xp + y
    mov DWORD PTR [edx], eax  ; store t in *xp
    ret                       ; return
```

- Function arguments are fetched relative to the stack pointer `esp`.
- Getting function arguments involves `mov` instructions that move *double words* ***from the stack*** into registers.

With x86-64 machine code you get (64bit):

```
gcc -O2 -S -masm=intel -m64 simple_l.c
```

```
simple_l:
    mov rax, QWORD PTR [rdi] ; dereference *xp
    add rax, rsi             ; add *xp + y
    mov QWORD PTR [rdi], rax ; store t in *xp
    ret                      ; return
```

- Function arguments are fetched directly from registers.
- Move operations take *quad words*.
- Registers have different name.
- The `-m64` flag is default on `x86-64` machines, you usually don't need it.

GENERAL PURPOSE REGISTERS

8 IA32 legacy registers:

63	31		15	8 7	0
rax	eax	ax	ah	al	Return value
rbx	ebx	bx	bh	bl	Callee saved
rcx	ecx	cx	ch	cl	4th arg
rdx	edx	dx	dh	dl	3rd arg
rsi	esi	si		sil	2nd arg
rdi	edi	di		dil	1st arg
rbp	ebp	bp		bpl	Callee saved
rsp	esp	sp		spl	Stack pointer

- The legacy registers for `IA32` (32-bit) start with an "`e`" for extended. The naming `ax`, `bx` and so on is historical *(from 8-bit and 16-bit days)*.
- `x86-64` further extends these registers to a 64-bit width. These registers start with a "`r`".
- These different registers must exist to ensure *backwards compatibility*.
- 64-bit instructions all take 64-bit register operands.

8 registers added in x86-64:

	63	31		15	8 7 0
5th arg	r8	r8d	r8w		r8b
6th arg	r9	r9d	r9w		r9b
Caller saved	r10	r10d	r10w		r10b
Caller saved	r11	r11d	r11w		r11b
Callee saved	r12	r12d	r12w		r12b
Callee saved	r13	r13d	r13w		r13b
Callee saved	r14	r14d	r14w		r14b
Callee saved	r15	r15d	r15w		r15b

- The `x86-64` ISA further introduces 8 additional 64-bit registers `r8` to `r15`.
- For compatibility, these registers are also accessible in the following variations: *double word* ("`d`", 32-bit), *word* ("`w`", 16-bit) and *byte* ("`b`", 8-bit).
- Registers serve different purposes as indicated. Most unique is the ***stack pointer*** `rsp`.
- Other special registers exist that are not general purpose, e.g., the ***instruction pointer*** `rip`.

GENERAL PURPOSE REGISTERS EXAMPLE

- In this example we write a simple assembly code that adds the contents of two general purpose registers together and stores the result back into one of the operands.
- This assembly code adds 1 and 2 together:

```
1  .text
2  .intel_syntax noprefix
3  .global _start
4  _start:
5      mov     rax, 0x1  ; initialize rax with value 1
6      mov     rbx, 0x2  ; initialize rbx with value 2
7      add     rax, rbx  ; add them together and store result in rax
8      ; this exits the program without libc and only works on 64-bit Linux:
9      mov     rax, 60
10     xor     rdi, rdi
11     syscall
```

  Only lines 5-7 are of interest to us. Lines 9-11 exit the program (only works on 64-bit Linux).

- Lines 5 and 6 initialize registers `rax` and `rbx` with values 1 and 2, respectively.
- Line 7 adds the two registers together and stores the result in register `rax`. This example uses Intel syntax.

x86-64 OPERAND FORMS AND ADDRESSING MODES

- In the last lecture we looked at addressing modes for a general ISA format.
- The following are the addressing modes specific for `x86-64`.
- There are *three* types of *operands*:
    1. *immediate*
    2. *register reference*
    3. *memory reference*
- Immediate values are standard C-style literal integers. ***Note:*** AT&T syntax prefixes immediate values with "`$`".
- AT&T prefixes registers with "`%`". Registers are "naked" (without "%") in Intel syntax.
- Memory references in AT&T are within "`()`" and within "`[]`" for Intel.

Type	Form	Operand value	Name
Immediate	`$Imm`	`Imm`	Immediate
Register	r_a	$R[r_a]$	Register
Memory	`Imm`	`M[Imm]`	Absolute
Memory	(r_a)	$M[R[r_a]]$	Indirect
Memory	$Imm(r_b)$	$M[Imm+R[r_b]]$	Base+displacement
Memory	(r_b,r_i)	$M[R[r_b]+R[r_i]]$	Indexed
Memory	$Imm(r_b,r_i)$	$M[Imm+R[r_b]+R[r_i]]$	Indexed
Memory	$(,r_i,s)$	$M[R[r_i]*s]$	Scaled indexed
Memory	$Imm(,r_i,s)$	$M[Imm+R[r_i]*s]$	Scaled indexed
Memory	(r_b,r_i,s)	$M[R[r_b]+R[r_i]*s]$	Scaled indexed
Memory	$Imm(r_b,r_i,s)$	$M[Imm+R[r_b]+R[r_i]*s]$	Scaled indexed

Bryant and O'Hallaron, 2015

- `R[]` denotes a *register reference* and `M[]` denotes a *memory reference* where `M` is a large array of bytes.
- The argument `x` in `M[x]` is called the *effective address*.
- The scaling factor `s` can be 1, 2, 4 or 8.
- The "Form" column corresponds to AT&T syntax

x86-64 OPERAND FORMS AND ADDRESSING MODES

Recall the previous example:

```
long int simple_l(long int *xp, long int y)
{
    long int t = *xp + y;
    *xp = t;
    return t;
}
```

Intel syntax:

```
simple_l:
    mov rax, QWORD PTR [rdi]
    add rax, rsi
    mov QWORD PTR [rdi], rax
    ret
```

- Memory references in square brackets "`[]`".
- The register `rdi` contains the memory address the pointer `xp` points to.
- Memory references indicate the size of the data, *quad word* in this example.

AT&T syntax:

```
simple_l:
    movq (%rdi), %rax
    addq %rsi, %rax
    movq %rax, (%rdi)
    ret
```

- Memory references in parenthesis "`()`".
- The register `rdi` contains the memory address the pointer `xp` points to.
- The size of the data move is indicated by the instruction name `movq` for *quad word* data.

Note: the difference of the operand order between Intel and AT&T syntax!

PROCEDURE (FUNCTION) CALLS

- Procedures are a key abstraction in software.
- Providing machine-level support for procedures requires one or more of the following mechanisms:
  1. *Passing control:* the instruction pointer must be set to point to the starting address of the called procedure.
  2. *Passing data:* the *caller* must be able to pass arguments to the *callee*.
  3. *Allocating and deallocating memory:* the callee might need to allocate space for *local* variables and must be freed when it returns *(→ stack!)*.
- `x86-64` contains a set of special instructions to implement these mechanisms along with a few *conventions on how to use machine resources*.
- Based on these conventions, it is possible that a function call does not even need the stack to execute (i.e., item 3 above) because all the data movement can be accomplished with registers only!

PROCEDURE CALL EXAMPLE

Consider to following code:

```c
#include <stdio.h>

void hello(const char *msg) {
    printf("Hello, %s!\n", msg);
}

int main(void)
{
    hello("World");
    return 0;
}
```

- The `main` function calls the function `hello`.
- Inside `hello` we call `printf` from the standard library with *two* arguments:
    1. `"Hello, %s!\n"`
    2. `msg`

Part of the assembly code:

```
.LC0: ; label that identifies a constant string
    .string "Hello, %s!\n"
hello: ; label for the hello function
    sub rsp, 8 ; grows the stack by 8 byte
    mov rsi, rdi ; store contents of rdi in rsi
    lea rdi, .LC0[rip] ; load string into rdi
    mov eax, 0 ; zero register eax (32-bit)
    call printf@PLT ; call printf function
    add rsp, 8 ; reduce the stack by 8 byte
    ret ; return from hello
.LC1: ; another constant string
    .string "World"
main: ; label for the main function
    sub rsp, 8 ; grows the stack by 8 byte
    lea rdi, .LC1[rip] ; load string into rdi
    call hello ; call hello function
    mov eax, 0 ; zero register eax (32-bit)
    add rsp, 8 ; reduce the stack by 8 byte
    ret ; return from main
```

- Functions are called using the `call` instruction and the callee returns with the `ret` instruction.
- How are function arguments passed? What about return values?

DATA TRANSFER IN PROCEDURE CALLS

- Most of the function argument passing in `x86-64` is done via *registers*. In the example on the previous slide you have observed that the function arguments to `printf` are passed in registers `rdi` and `rsi`.
- Values are returned in register `rax`.
- The following table provides an overview of which registers are used for function argument passing (*by convention* in `x86-64`):

	Argument number					
Operand size (bits)	**1**	**2**	**3**	**4**	**5**	**6**
64	`rdi`	`rsi`	`rdx`	`rcx`	`r8`	`r9`
32	`edi`	`esi`	`edx`	`ecx`	`r8d`	`r9d`
16	`di`	`si`	`dx`	`cx`	`r8w`	`r9w`
8	`dil`	`sil`	`dl`	`cl`	`r8b`	`r9b`

- Additional arguments must be pushed onto the stack from which the function will read. You can see that functions with more than 6 arguments will create more *overhead* to run on `x86-64` (this is an architectural detail).

BACK TO INSTRUCTION-LEVEL PARALLELISM

In the last lecture we have seen that instruction-level parallelism happens on the hardware. Knowing how this works, you can (help the compiler to) exploit it in your *software* → *you and the compiler are a team!*

- Instruction-level parallelism is dynamically exploited on the hardware:
  - Processor pipelining
  - Out-of-order execution
  - Speculative execution
  - Multiple execution units (***recall:*** *roofline)*

We have talked about these principles last time.

- In your software, you can further help to exploit these principles with techniques such as:
  - Loop-level parallelism: *loop-unrolling*. The compiler attempts to do this automatically when the `-O3` flag is present (see `man gcc` for what is in `-O3`).
  - *Code fusion:* break/reduce dependency chains among instructions. For example, two small functions may be merged into one large function.

BACK TO INSTRUCTION-LEVEL PARALLELISM

Consider the reduction kernel: $s = \sum_{i=1}^{n} x_i$

Naive implementation:

```c
float reduce(const float *x, const int n)
{
    float sum = 0.0;
    for (int i = 0; i < n; ++i) {
        sum += x[i];
    }
    return sum;
}
```

Assembly: `gcc -O2 -S`

```asm
reduce:
.L3: ; loop label
    addss xmm0, DWORD PTR [rdi] ; add to sum
    add   rdi, 4 ; add 4 bytes to address of x
    cmp   rdi, rax ; check if exit loop
    jne   .L3 ; if not continue at .L3
    ret ; done
```

Manual 4-fold unroll:

```c
float reduce(const float *x, const int n)
{
    float sum0 = 0.0;
    float sum1 = 0.0;
    float sum2 = 0.0;
    float sum3 = 0.0;
    for (int i = 0; i < n; i += 4) {
        sum0 += x[i + 0];
        sum1 += x[i + 1];
        sum2 += x[i + 2];
        sum3 += x[i + 3];
    }
    return sum0 + sum1 + sum2 + sum3;
}
```

Assembly: `gcc -O2 -S`

```asm
reduce: ; 4-fold unroll: 4x more addss instructions
.L3: ; loop label
    addss xmm1, DWORD PTR [rdi]
    addss xmm0, DWORD PTR 4[rdi]
    add   rdi, 16 ; advance address of x by 16 byte
    addss xmm3, DWORD PTR -8[rdi]
    addss xmm2, DWORD PTR -4[rdi]
    cmp   rax, rdi ; check if exit loop
    jne   .L3 ; if not continue at .L3
    addss xmm0, xmm1 ; reduce partial sum
    addss xmm0, xmm3 ; reduce partial sum
    addss xmm0, xmm2 ; reduce partial sum
    ret ; done
```

BACK TO INSTRUCTION-LEVEL PARALLELISM

Consider the reduction kernel: $s = \sum_{i=1}^{n} x_i$

- GCC also supports `#pragma GCC unroll n` where n specifies how many time the loop should be unrolled.
- To tell GCC that it should unroll loops in certain functions, you can identify them with `__attribute__((optimize("unroll-loops")))`. See https://gcc.gnu.org/onlinedocs/gcc/Function-Attributes.html.

GCC specific optimization:

```c
float reduce(const float *x, const int n)
{
    float sum = 0.0;
#pragma GCC unroll 8 // 8-fold unroll
    for (int i = 0; i < n; ++i) {
        sum += x[i];
    }
    return sum;
}
```

The loop is unrolled 8-times. You have reduced the loop overhead by executing fewer `cmp` instructions. ***Can you identify something else in the assembly that may not be ideal?***

Assembly: `gcc -O2 -S`

```asm
reduce:
.L3:
    addss xmm0, DWORD PTR [rdi]
    add   rdi, 32
    addss xmm0, DWORD PTR -28[rdi]
    addss xmm0, DWORD PTR -24[rdi]
    addss xmm0, DWORD PTR -20[rdi]
    addss xmm0, DWORD PTR -16[rdi]
    addss xmm0, DWORD PTR -12[rdi]
    addss xmm0, DWORD PTR -8[rdi]
    addss xmm0, DWORD PTR -4[rdi]
    cmp   rdi, rdx
    jne   .L3
    ret
```

BACK TO INSTRUCTION-LEVEL PARALLELISM

Consider the reduction kernel: $s = \sum_{i=1}^{n} x_i$

GCC specific optimization:

```c
float reduce(const float *x, const int n)
{
    float sum0 = 0.0;
    float sum1 = 0.0;
#pragma GCC unroll 4 // 4-fold unroll
    for (int i = 0; i < n; i += 2) {
        // times 2-fold manual unroll
        sum0 += x[i + 0];
        sum1 += x[i + 1];
    }
    return sum0 + sum1;
}
```

- The two summation variables will allocate two separate registers.
- This code can better exploit architectures with multiple execution units for floating-point `addss` instructions.

Assembly: `gcc -O2 -S`

```asm
reduce:
.L3:
    lea    rax, 8[rdi]
    addss xmm0, DWORD PTR [rdi]
    addss xmm1, DWORD PTR 4[rdi]
    addss xmm0, DWORD PTR 8[rdi]
    addss xmm1, DWORD PTR 12[rdi]
    addss xmm0, DWORD PTR 16[rdi]
    addss xmm1, DWORD PTR 20[rdi]
    lea    rdi, 24[rax]
    addss xmm0, DWORD PTR 16[rax]
    addss xmm1, DWORD PTR 20[rax]
    cmp    rdx, rdi
    jne    .L3
    addss xmm0, xmm1 ; reduce partial sums
    ret
```

FLOATING-POINT OPERATIONS

- In the previous slide you may have noticed that the compiler used *different registers* for the reduction operation.
- *In `x86-64` the general purpose registers are used for integer and pointers*.
- *Floating-point data is treated differently*. The following aspects affect ISA's with floating-point support:
  - How floating-point values are stored and accessed. This is typically via some form of registers.
  - The instructions that operate on floating-point data.
  - The conventions used for passing floating-point values as arguments to functions and for returning them as results.
  - The conventions for how registers are preserved during function calls. For example, with some registers designated as *caller saved* and others as *callee saved*. If a register is *caller saved*, then it can be *overwritten* in the callee without creating a backup.

HISTORY OF FLOATING-POINT ON `x86-64`

- The `IA32` and `x86-64` instruction sets introduced ***extensions*** intended for *multimedia* and *graphics* support starting in 1997.
- The instructions in these extensions focused on allowing to operate on *multiple data* in parallel, applying the same instruction to each data element.
- In Flynn's taxonomy we have identified this form of parallel computer as ***S**ingle-**I**nstruction-**M**ultiple-**D**ata*, short *SIMD*.
- The data these instructions focus on are integer or floating-point and exploit *data-level parallelism* (DLP).
- Old Cray vector computers already operated on *data vectors* to perform computations *data-parallel*. These machines were mostly used for numerical simulations. See https://www.hpe.com/us/en/compute/hpc/cray.html for some history and https://en.wikipedia.org/wiki/Cray-1 for more on Cray-1.
- We say a code is *vectorized* when it is capable to process *vectors of data*. All modern CPU architectures support this form of parallelism. See also https://en.wikipedia.org/wiki/Vector_processor.

HOW IMPORTANT IS SIMD?

- SIMD is a ***very powerful*** form of parallelism and can deliver *significant performance improvements.*
- It is *difficult* to exploit SIMD efficiently on CPUs (it is slightly easier on GPU architectures due to the SIMT execution model).
- ***Not all algorithms map well to this form of data-level parallelism!***

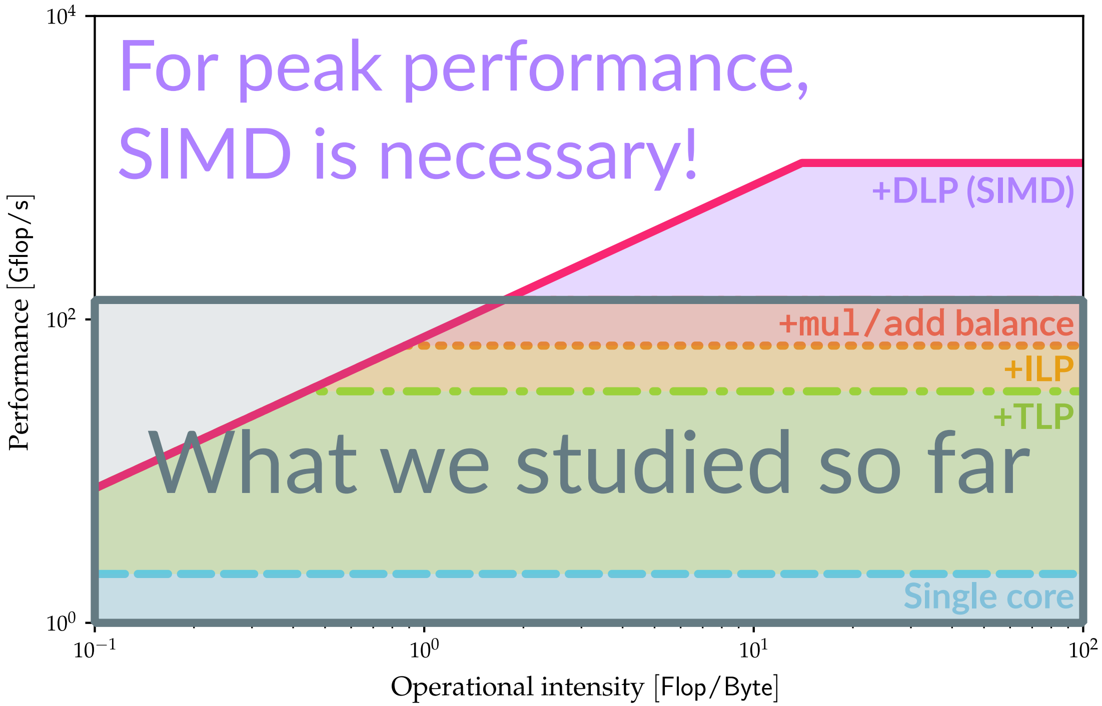
Recall the roofline:

Leiserson et al., 2020
(first reading)

Table 1. Speedups from performance engineering a program that multiplies two 4096-by-4096 matrices. Each version represents a successive refinement of the original Python code. "Running time" is the running time of the version. "GFLOPS" is the billions of 64-bit floating-point operations per second that the version executes. "Absolute speedup" is time relative to Python, and "relative speedup," which we show with an additional digit of precision, is time relative to the preceding line. "Fraction of peak" is GFLOPS relative to the computer's peak 835 GFLOPS. See Methods for more details.

Version	Implementation	Running time (s)	GFLOPS	Absolute speedup	Relative speedup	Fraction of peak (%)
1	Python	25,552.48	0.005	1	–	0.00
2	Java	2,372.68	0.058	11	10.8	0.01
3	C	542.67	0.253	47	4.4	0.03
4	Parallel loops	69.80	1.969	366	7.8	0.24
5	Parallel divide and conquer	3.80	36.180	6,727	18.4	4.33
6	plus vectorization	1.10	124.914	23,224	3.5	14.96
7	plus AVX intrinsics	0.41	337.812	62,806	2.7	40.45

About 10x speedup with SIMD (depends on architecture).



OVERVIEW OF x86 SIMD EXTENSIONS

	Vector register width	*x86-64, 64-bit*	*IA32, 32-bit*	Microarchitecture	*Time*
				386, 486, Pentium	1985
Multimedia Extensions (MMX)	64-bit (integers)		MMX	Pentium MMX	1997
Streaming SIMD Extensions (SSE)	128-bit		SSE	Pentium III	1999
			SSE2	Pentium 4	
			SSE3	Pentium 4E	
		SSE4		Pentium 4F	
				Core 2 Duo	2006
				Core i7 (Nehalem)	2008
Advanxed Vector Extensions (AVX)	256-bit	AVX		Sandy Bridge	2011
		AVX2		Haswell, Broadwell	2013
	512-bit	AVX512		Skylake	2015
				Coffe Lake	
				Cannon Lake	2019

Run `lscpu` to get a listing of what your processor supports.

MEDIA REGISTERS

- There are special media registers for floating-point data.
- Starting with SSE2 (Pentium 4 in 2000), the media instructions also perform operations for *scalar* floating-point data using the low-order 32 or 64-bit of the `xmm` or `ymm` registers.
- Floating-point computation in `x86-64` is based on SSE or AVX.
- Memory references are specified in the same way as for the general purpose registers discussed earlier. All of the different combinations of displacement, base register, index register and scaling factor are supported.

Reduction example from earlier today:

```
1 reduce:
2 .L3: ; loop label
3     addss xmm0, DWORD PTR [rdi] ; add to sum (scalar!)
4     add   rdi, 4 ; add 4 bytes to address of x
5     cmp   rdi, rax ; check if exit loop
6     jne   .L3 ; if not continue at .L3
7     ret ; done
```

Floating-point operation in line 3, the rest is integer arithmetic.

16 floating point registers in `x86-64`:

	255	127 0
1st FP arg/ return value	ymm0 (256-bit)	xmm0 (128-bit)
2nd FP arg	ymm1	xmm1
3rd FP arg	ymm2	xmm2
4th FP arg	ymm3	xmm3
5th FP arg	ymm4	xmm4
6th FP arg	ymm5	xmm5
7th FP arg	ymm6	xmm6
8th FP arg	ymm7	xmm7
Caller saved	ymm8	xmm8
Caller saved	ymm9	xmm9
Caller saved	ymm10	xmm10
Caller saved	ymm11	xmm11
Caller saved	ymm12	xmm12
Caller saved	ymm13	xmm13
Caller saved	ymm14	xmm14
Caller saved	ymm15	xmm15

PROCEDURES WITH FLOATING-POINT ARGUMENTS

- With `x86-64`, the `xmm` registers are used for *passing* floating-point arguments to functions and for *returning* floating-point values from them.
- Up to *eight* floating-point arguments can be passed in registers `xmm0` to `xmm7`. These registers are used in the order the arguments appear in the function signature.
- Additional floating-point arguments can be passed on the stack.
- A function that *returns* a floating-point value does so using register `xmm0`.
- All `xmm`/`ymm` registers are *caller saved*. The callee may *overwrite* any of these registers without first saving it.

Example:

```
double f(double *pd, double a0, double a1, double a2,
         double a3, double a4, double a5, double a6,
         double a7, double a8)
{
    return pd[0] +
           a0 + a1 + a2 + a3 +
           a4 + a5 + a6 + a7 +
           a8;
}
```

`gcc -O2 -S -masm=intel`

```
f:
    addsd   xmm0, QWORD PTR [rdi] ; xmm0 = a0 + pd[0]
    movapd  xmm8, xmm0 ; xmm8 = xmm0
    movsd   xmm0, QWORD PTR 8[rsp] ; xmm0 = a8 (stack)
    addsd   xmm8, xmm1 ; xmm8 += a1
    addsd   xmm8, xmm2 ; xmm8 += a2
    addsd   xmm8, xmm3 ; xmm8 += a3
    addsd   xmm8, xmm4 ; xmm8 += a4
    addsd   xmm8, xmm5 ; xmm8 += a5
    addsd   xmm8, xmm6 ; xmm8 += a6
    addsd   xmm8, xmm7 ; xmm8 += a7
    addsd   xmm0, xmm8 ; xmm0 += xmm8 (return xmm0)
    ret
```

FLOATING-POINT ARITHMETIC OPERATIONS

- Floating-point arithmetic is implemented in SSE/AVX/AVX2 in `x86-64`. Single precision (32-bit) and double precision (64-bit) operations for *scalar* and *packed* data are supported. *Scalar* → *single data element* and *packed* → *multiple data elements (SIMD)*.
- Each of these instructions has either *one or two source operands* and *one destination*. The second source operand can either be a `xmm` register *or* a memory location, the first source and destination must be `xmm` registers (Intel).

`x86-64` scalar floating-point arithmetic operators (latencies given in cycles):

Instruction				Latency single/double	
Single	**Double**	**Operation**	**Description**	**Haswell**	**Broadwell**
`vaddss`	`vaddsd`	`dst ← src2 + src1`	Floating-point add	3/3	3/3
`vsubss`	`vsubsd`	`dst ← src2 - src1`	Floating-point subtract	3/3	3/3
`vmulss`	`vmulsd`	`dst ← src2 * src1`	Floating-point multiply	5/5	3/5
`vdivss`	`vdivsd`	`dst ← src2 / src1`	Floating-point divide	<13/14-20	<11/<14
`vmaxss`	`vmaxsd`	`dst ← max(src2, src1)`	Floating-point maximum	3/3	3/3
`vminss`	`vminsd`	`dst ← min(src2, src1)`	Floating-point minimum	3/3	3/3
`vsqrtss`	`vsqrtsd`	`dst ← √src1`	Floating-point square root	13/20	13/20

FUSED MULTIPLY-ADD

- Fused Multiply-add (FMA) instructions are a further extension in the `x86-64` instruction set.
- They allow to perform a *multiply* and *add* in the same cycle (*fused*).
- Using these instructions requires a balanced multiply/add operation mix in your compute kernel → *to reach peak performance, use of FMA instructions is necessary*.
- FMA comes in two forms: FMA4 and FMA3. They are not compatible and have 4 and 3 operands, respectively.
- The basic operation is `d = round(a * b + c)`, where in FMA4 registers a, b, c and d can all be different and in FMA3 register d must either be a, b or c.
- Intel uses FMA3 in the following form: `a = round(a * b + c)` where a and b must be `xmm/ymm` registers and c can be a `xmm/ymm` register or a memory location. Let a denote operand 1, b operand 2 and c operand 3. The following permutations are available:
  - ***132***: `a = round(a * c + b)` ***Example instruction:*** `vfmadd132ss`
  - ***213***: `a = round(b * a + c)` ***Example instruction:*** `vfmadd213ss`
  - ***231***: `a = round(b * c + a)` ***Example instruction:*** `vfmadd231ss`
- See the Intel intrinsics guide: https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html

FUSED MULTIPLY-ADD

*x86-64 **scalar** FMA examples (latencies given in cycles):*

Instruction			Latency single/double	
Single	Double	Operation	Haswell	Broadwell
vfmadd132ss	vfmadd132sd	dst ← dst * src3 + src2	5/5	5/5
vfnmadd231ss	vfnmadd231sd	dst ← -(src2 * src3) + dst	5/5	5/5
vfmsub132ss	vfmsub132sd	dst ← dst * src3 - src2	5/5	5/5
vfnmsub213ss	vfnmsub213sd	dst ← -(src2 * dst) - src3	5/5	5/5

Consider the simple kernel:

```
double fma(double a, double b, double c) { return a * b + c; }
```

gcc -O1 -S -mfma:

```
fma: ; xmm0 = a, xmm1 = b, xmm2 = c
    vmulsd  xmm0, xmm0, xmm1 ; multiply
    vaddsd  xmm0, xmm0, xmm2 ; then add
    ret
```

- You need the `-mfma` flag to enable FMA instructions.
- Still, -O1 or less *does not* issue FMA instructions.

gcc -O2 -S -mfma:

```
fma: ; xmm0 = a, xmm1 = b, xmm2 = c
    ; xmm0 = xmm0 * xmm1 + xmm2
    vfmadd132sd xmm0, xmm2, xmm1 ; op1, op2, op3
    ret
```

- Here the compiler has issued a FMA instruction.
- A high performance kernel needs many FMA's in *packed form (SIMD → next lecture).*

RECAP

- Understanding the basics of (`x86-64`) assembly language is important for program analysis.
- There are 16 general purpose registers in `x86-64`.
- The *stack pointer* is stored in register `rsp` and the *base pointer* in `rbp`. The *instruction pointer* is stored in the special register `rip`.
- The first 6 function arguments are passed in registers. A return value is passed in register `rax`.
- SIMD is important if peak performance is an issue.
- Floating-point operations are available through SSE and AVX in `x86` architectures.
- Special media registers are used for floating point operations.
- Fused multiply-add instructions are further important if peak performance is an issue. Exploiting FMA depends on the algorithm however.

Further reading:

- Chapter 4 in "*Computer Architecture*",
  J. Hennessy and D. Patterson, Morgan Kaufmann 2019
- "*Intel© 64 and IA-32 Architectures Software Developer's Manual*",
  Volume 2, Instruction Set Reference, pdf