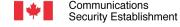
# Floating point and vectorized operations

Getting to know some more registers

© Government of Canada

This document is the property of the Government of Canada. It shall not be altered, distributed beyond its intended audience, produced, reproduced or published, in whole or in any substantial part thereof, without the express permission of CSE.





Historically the 8087 was used as the "floating point unit". This is legacy.

We will be using SSE SIMD operations to do floating point operations and learn about SIMD at the same time.





For those of you who are afraid about backward compatibility. Know that SSE was introduced in 1999 (P3). Safe to assume most Intel based computer still in service supports these...



# **SSE = Streaming SIMD Extensions**

# How about we start by learning what SIMD is about...



### Single Instruction Multiple Data (SIMD)

- Used when the same operation is to be applied across a large data set
- Commonly used in math heavy scenarios
  - As part of reverse engineering task (in malware analysis for example). It's "fairly safe" to assume that code making heavy use of SIMD instruction is likely either encoding or cryptographic code.
- Although we now have "SIMD" instructions, the idea is far from new
  - Example:
    - mov eax, 0x41424344; eax = "ABCD"
    - xor eax, 0x56565656 ; A = A ^ 0x56, B = B ^ 0x56, C = C ^ 0x56, D = D ^ 0x56
- SIMD instructions provides hardware level support for common edge cases
- SSE provide 16 128 bits registers (XMM0 XMM15)
- AVX extends these registers to 256 bits (name changes to YMM0 YMM15) and even to 512 bits with AVX2





#### Quick example using integer arithmetics

```
adder:
        push rbp
        mov rbp, rsp
        xor rax, rax
    loopAdder:
        add qword [rdi + rax], 0x01
        add rax, 0x08
        cmp rax, rsi
        jnz loopAdder
        pop rbp
        ret
```

fadder implements vectorized addition even if the code is larger, the vectorized version runs at about twice the speed of the non vectorized code. However, the vectorized code uses more memory.

```
fadder:
        push rbp
        mov rbp, rsp
        sub rsp,
        mov rax, \theta
        mov [rbp - 8], rax
        mov [rbp - 0x10], rax
        movaps xmm1, [rbp - 0x10]
        xor rax, rax
    fastLoop:
        movdqa xmm0, [rdi + rax]
        paddq xmm0, xmm1
        movdqa [rdi + rax], xmm0
        add rax, 0x10
        cmp rax, rsi
        jnz fastLoop
        add rsp, 0x10
        pop rbp
        ret
```

## Floating point operations (non vectorized)

Instruction	Description	
MOVSS	Move a single precision scalar into an xmm register	
MOVSD	Move a double precision scalar into an xmm register	
CMPSS / CMPSD	Compare. This is similar to CMP you are used to. With a twist.	
ADDSS / ADDSD	Addition. This is similar to ADD you are used to.	
SUBSS / SUBSD	Subtraction. This is similar to SUB you are used to.	
MULSS / MULSD	Multiplication. Some differences! Single destination register.	
DIVSS / DIVSD	Division. Some differences! Single destination register.	



#### **Example**

Bottom line here: a whole instruction set is available for SSE and the associated floating point operations.

```
section .data
    align 16
        x: dq 3.5
        y: dq 3.1
    form:
        db "%f", 0xa, 0x00
section .text
extern printf
global main
main:
    push rbp
    mov rbp, rsp
    movsd xmm0, [x]
    movsd xmm1, [y]
    addsd xmm0, xmm1
    movsd [x], xmm0
    lea rdi, [form]
    mov rsi, [x]
    call printf
    pop rbp
    ret
```



Communications

#### **Using CMP with scalar floating points**

Instruction	Condition	Туре	Example
CMP	EQ	SS or SD	CMPEQSS xmm0, xmm1 (compare to see if equal)
	LT		CMPLTSS xmm0, xmm1 (compare to see if less than)
	LE		CMPLESS xmm0, xmm1 (compare to see if less of equal)
	NEQ		CMPNEQSS xmm0, xmm1 (compare to see if not equal)
	NLT		CMPNLTSS xmm0, xmm1 (Not less than)
	NLE		CMPNLESS xmm0, xmm1 (Not less or equal)



#### **Example**

Writing compare code using SSE can feel a bit strange.

```
LINCLASSIFIED
section .data
    align 16
        x: dq 2.0
        y: dq 6.5
    form:
        db "%f", 0x0a, 0
section .text
extern printf
global main
main:
    push rbp
    mov rbp, rsp
again:
    movsd xmm0, [x]
    movsd xmm1, [y]
    mulsd xmm0, xmm1
    cmpnlesd xmm0, xmm1 ; Result is an endless loop
    jge again
    pop rbp
    ret
```

## Let's write some code

