CMPT 295

Unit - Machine-Level Programming

Lecture 22 – Buffer Overflow + Floating-point data & operations

Last lecture

- Manipulation of 2D arrays – in x86-64
  - From x86-64's perspective, a 2D array is a contiguously allocated region of *R* * C * `L` bytes in memory where `L = sizeof(T)` and *T* -> data type of elements stored in array
  - 2D Array layout in memory: Row-Major ordering
  - Memory address of each row `A[i]`: `A + (i * C * L)`
  - Memory address of each element `A[i][j]`:

    `A + (i * C * L) + (j * L)`

    `=> A + (i * C + j) * L`

Today's Menu

- Introduction
  - C program -> assembly code -> machine level code
- Assembly language basics: data, `move` operation
  - Memory addressing modes
- Operation `leaq` and Arithmetic & logical operations
- Conditional Statement – Condition Code + `cmovX`
- Loops
- Function call – Stack
  - Overview of Function Call
  - Memory Layout and Stack - x86-64 instructions and registers
  - Passing control
  - Passing data – Calling Conventions
  - Managing local data
  - Recursion
- Array
- Buffer Overflow
- Floating-point data & operations

Buffer Overflow

C and Stack … so far

- C does not perform any bound checks on arrays

- Local variables in C programs
- **Callee** and **caller** saved registers
- Return addresses

} stored on the stack

- As we saw in Lab 2 and Lab 4, this may lead to **trouble**

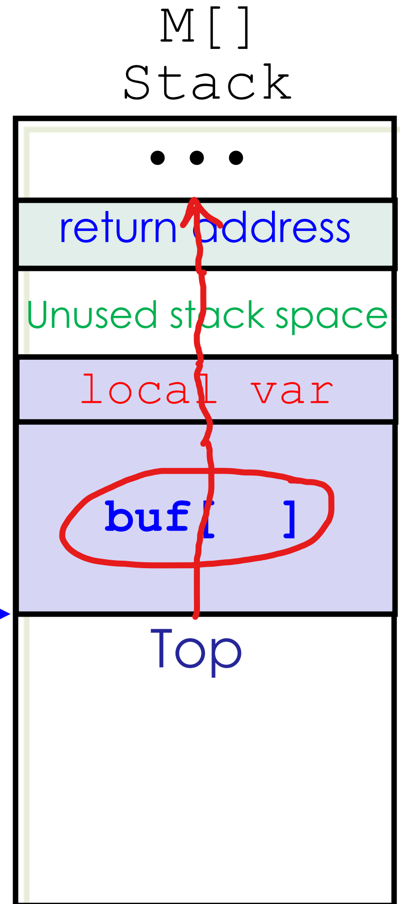
What kind of **trouble**?
-> buffer overflow (overrun)

- If function does not perform *bound-check* when writing to a **local array** …

Here is a an example of a bound-check:
```
if input size <= array size
   write input into array
```

… then it may write more data that the allocated space (to array) can hold, hence overflowing the array -> ***buffer overflow***

- **Effect**: the function may end up writing over, i.e., corrupting, data kept on the stack such as:
  - Value of local variables and registers
  - Return address
- ***Stack smashing***



Demo the **trouble** -> buffer overflow

Why is *buffer overflow* a problem

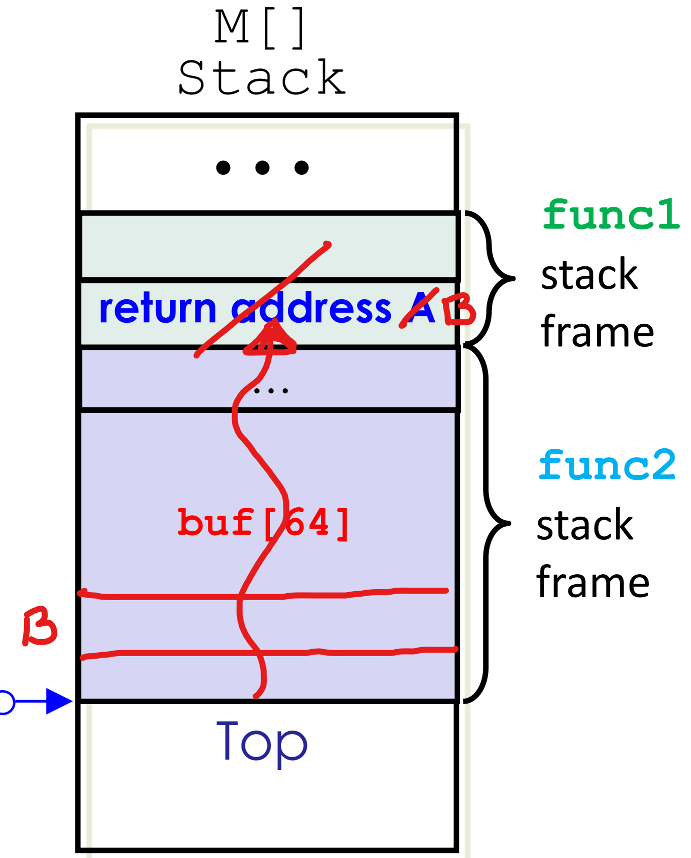
- Corrupted data
- Corrupted return address
  - Which may lead to segmentation fault
    - How?
  - Which also makes a system vulnerable to attacks
    - How?

Code injection attack

```
void func1(){
   func2();
   // C statement
      at return
      address A
   ...
}
```

```
int func2() {
   char buf[64];
   gets(buf);
   ...
   return ...;
}
```

- An “attacker” could overflow the buffer … array of `char`'s
  - … by inputting a **string** that contains byte representation of ***malicious executable code*** (***exploit code***) instead of legitimate characters
  - The **string** is written to array **`buf`** on stack and overwrites **return address A** with a ***return address*** that points to ***exploit code***
  - When **`func2`** executes **`ret`** instruction, it pops this erroneous ***return address*** onto PC (`%rip`) and jumps to ***exploit code***
  - Microprocessor starts executing the ***exploit code*** at this location



How to protection against such attack

1. Avoid creating overflow vulnerabilities in the code that we write by always checking bounds
   - For example, by calling library functions that limit string lengths
     - "Unsafe" : `gets()`, `strcpy()`, `strcat()`, `sprintf()`, ...
       - These functions can generate a byte sequence without being given any indication of the size of the destination buffer (see next slide)
     - "Safe": `fgets()`

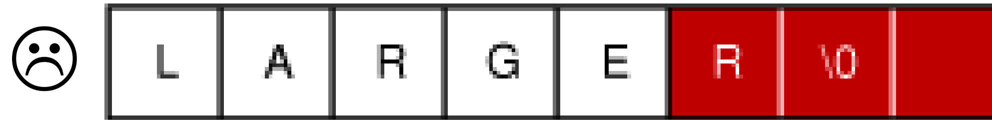
From our Lab 4

```c
void proc1(char *s, int *a, int *b) {
    int v;
    int t;

    t = *a;
    v = proc2(*a, *b);

    sprintf(s, "The result of proc2(%d,%d) is %d.", *a, *b, v);

    *a = *b - 2;
    *b = t;

    return;
}
```

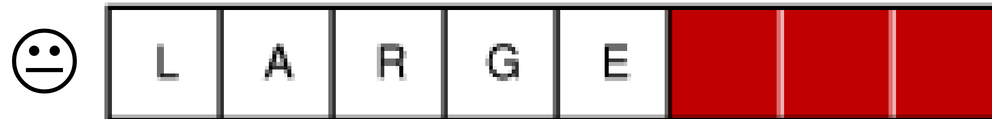
Suggestion from developer.apple.com

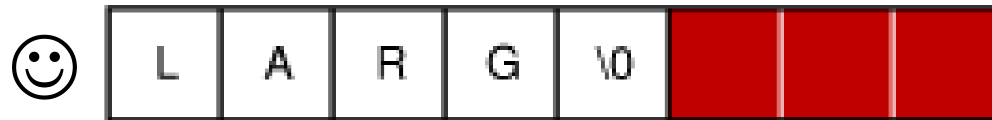
Copies up to **sizeof(destination) -> n** characters from the string pointed to by ***source*** to ***destination***. In a case where the length of ***source*** is less than **n**, the remainder of ***destination*** will be padded with null bytes. In a case where the length of ***source*** is greater than **n**, the ***destination*** will contain a truncated version of ***source***.

```
char destination[5];
char * source = "LARGER";
```



Copies the string pointed to by ***source*** (including the null character) to the ***destination*** and returns it.





Copies up to **sizeof(destination)** - 1 **->** **n** - 1 characters from null-terminated ***source*** to ***destination***, it then "null" terminates ***destination*** and returns the length of ***source***.

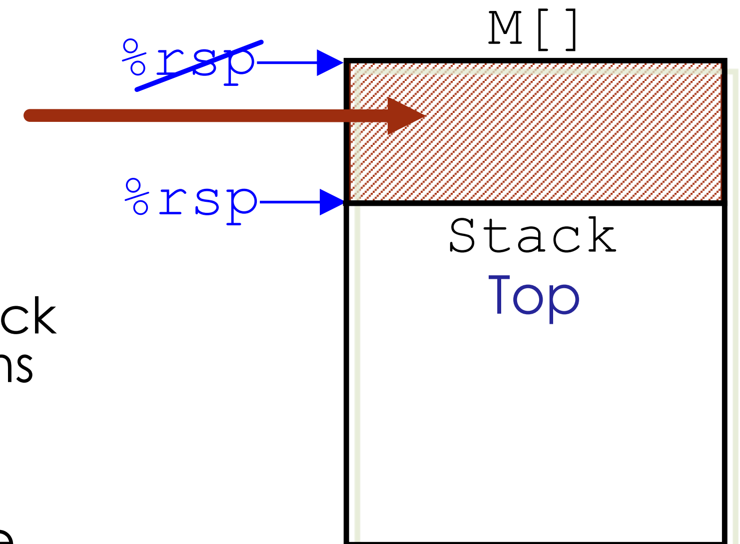
https://linux.die.net/man/3/strlcpy

How to protection against such attack

2. Employ system-level protections

-> Randomized stack offsets

- At start of program, system allocates random amount of space on stack
- Effect: Shifts stack addresses (`%rsp`) for entire program
  - Shifts the memory address of all the stack frames allocated to program's functions when they are called
- Hence, makes it difficult for hackers to predict start of each stack frame (hence where ***exploit code*** may have been inserted) since stack is repositioned each time program executes

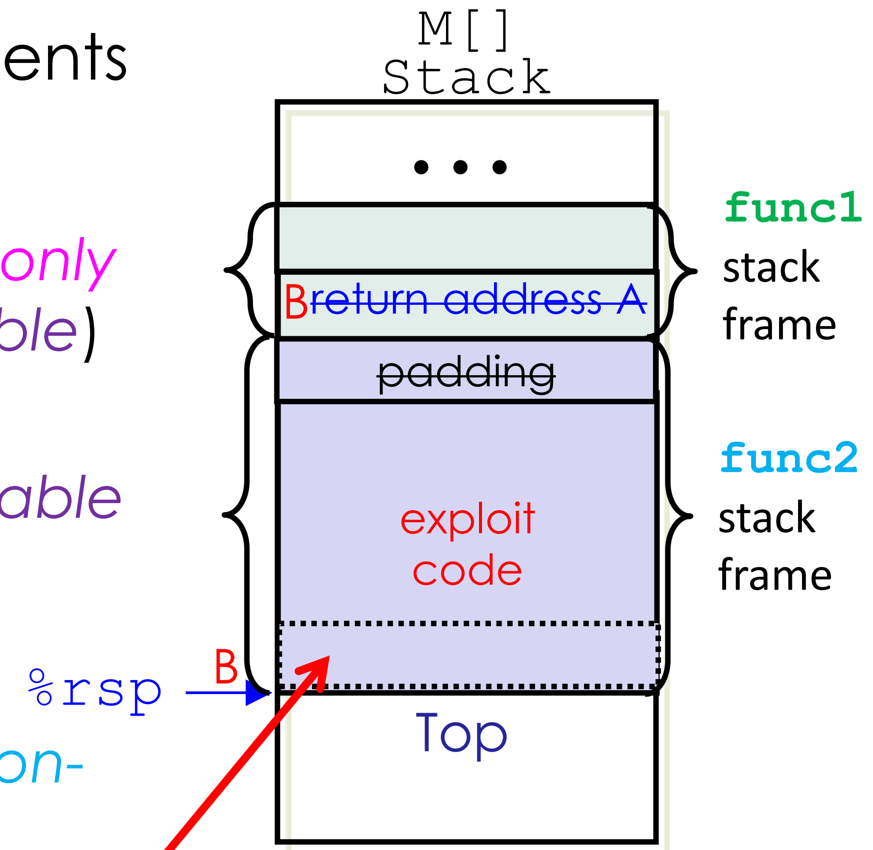


How to protection against such attack

2. Employ system-level protections

   -> Non-executable code segments

- In the old days of x86, memory segments marked as either *read-only* or *writeable* (both implied *readable*) => 2 types of permissions
  - Could execute anything *readable*
- x86-64 has added an explicit *executable* permission
- Stack segment now marked as *non-executable*



How to protection against such attack

3. Compiler (like `gcc`) uses a stack *canary* value
   - History: Starting early 1900's, canaries used in the coal mines to detect gas leaks
   - Push a randomized *canary* value between an array and return address on stack (remember our Lab 4)
   - Before executing a **`ret`** instruction, *canary* value is checked to see if it has been corrupted
     - *If so, failure reported*

```
main:   # main.c from our Lab 4
    endbr64
    pushq %rbp
    …
    subq  $64, %rsp
    movq  %fs:40, %rax
    movq  %rax, 56(%rsp)
    …
    leaq  16(%rsp), %rbp
    …
    movq  56(%rsp), %rax
    xorq  %fs:40, %rax
    jne   .L5
    addq  $64, %rsp
    popq  %rbp
    ret
.L5:
    call  __stack_chk_fail@PLT
```

How to protection against such attack

3. Newest version of our `gcc` compiler (version 8 and up) uses *Control-Flow Enforcement Technology* (CET)
   - Instruction **`endbr64`** (End Branch 64 bit) -> Terminate Indirect Branch in 64 bit
   - Microprocessor tracks indirect branching and ensures that all indirect calls lead to (legal) functions starting with **`endbr64`**
     - If function does -> microprocessor infers that function is safe to execute
     - If function does not -> microprocessor infers that control flow may have been manipulated by some ***exploit code***, i.e., function is unsafe to execute and aborts!

From stackoverflow

```
main:
.LFB23:
    .cfi_startproc
    endbr64
    pushq   %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset 6, -16
    movl    $9, %ecx
    movl    $6, %edx
    ...
```

Source: https://stackoverflow.com/questions/56905811/what-does-the-endbr64-instruction-actually-do

Brief overview of
floating-point data and operations

Background

- Once upon a time in the '90's …
  - Use of computer graphics and image processing (multimedia) applications were on the rise
    - Microprocessors (i.e., machine instruction sets) designed to support such applications
    - Idea: speed up microprocessors by executing **s**ingle **i**nstruction on **m**ultiple **d**ata -> **SIMD**
- Since then, microprocessors and their machine instruction sets have evolved …
    - **SSE** (**S**treaming **S**IMD **E**xtensions)
    - **AVX** (**A**dvanced **V**ector E**X**tensions) -> textbook

x86-64 registers and instructions seen so far are referred to as ***integer registers*** and ***integer instructions***

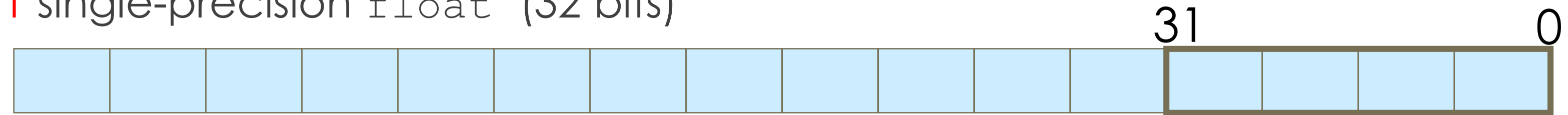
Now we introduce a new set of registers for floating point numbers:

XMM Registers

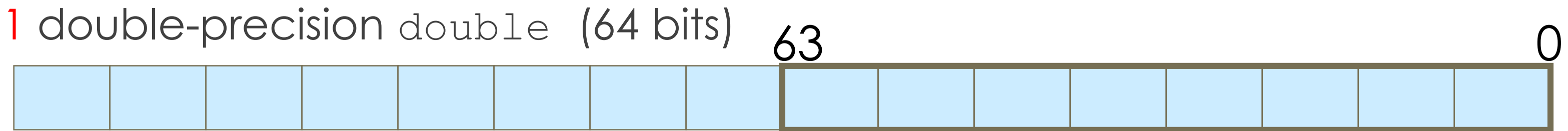
- 16 in total, each 16-byte wide (128 bits), named: `%xmm0`, `%xmm1`, ..., `%xmm15`

Scalar mode

- 1 single-precision `float` (32 bits)

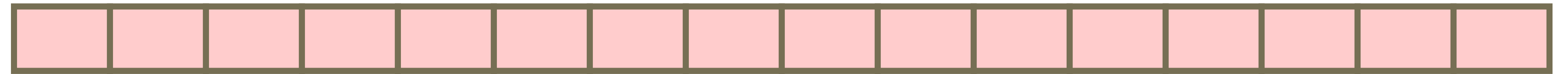


- 1 double-precision `double` (64 bits)

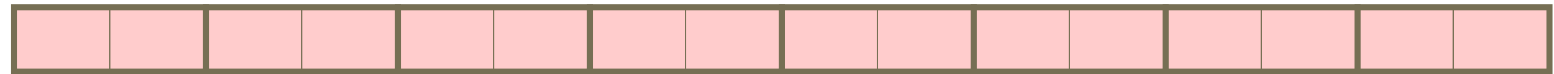


Vector mode (packed data)

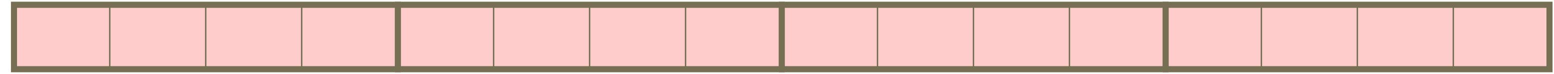
- 16 single-byte integers



- 8 16-bit integers



- 4 32-bit integers

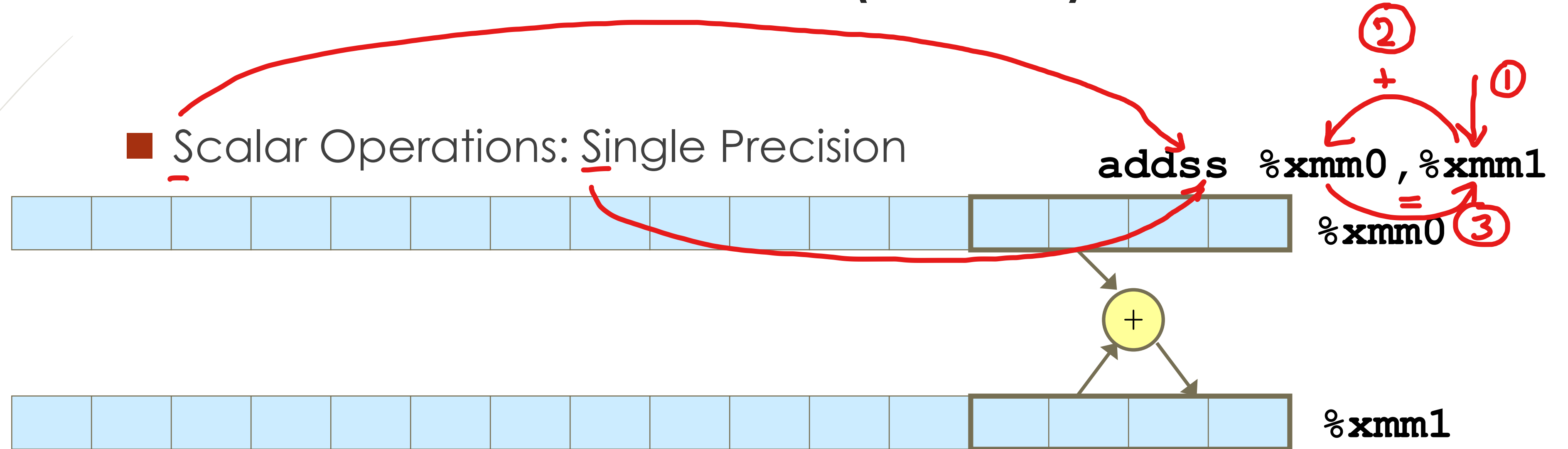


- 4 single-precision `float's`

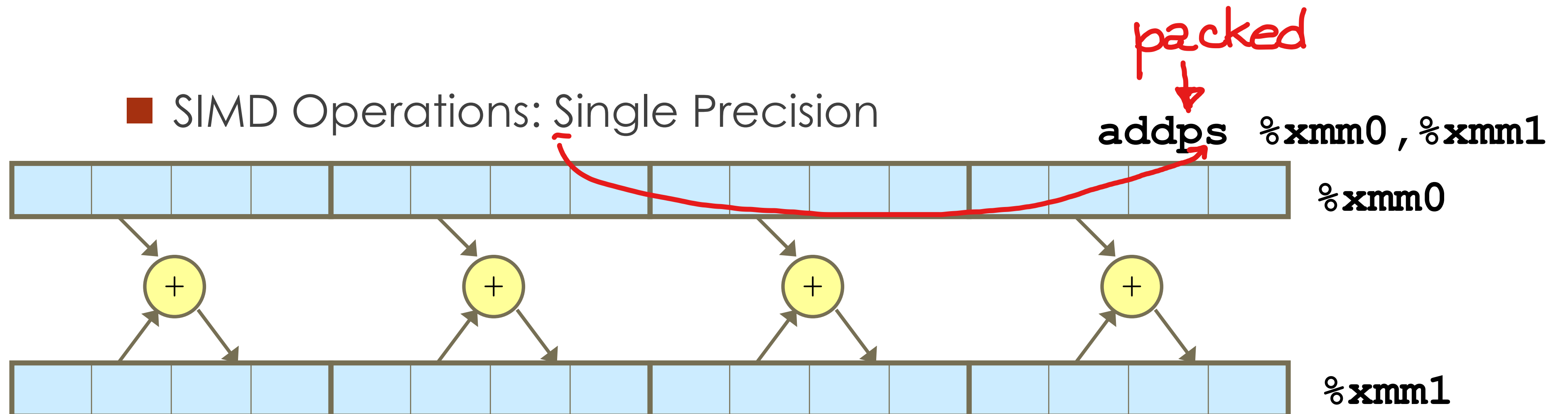
- 2 double-precision `double's`

Scalar versus Vector (SIMD) instructions

- Scalar Operations: Single Precision

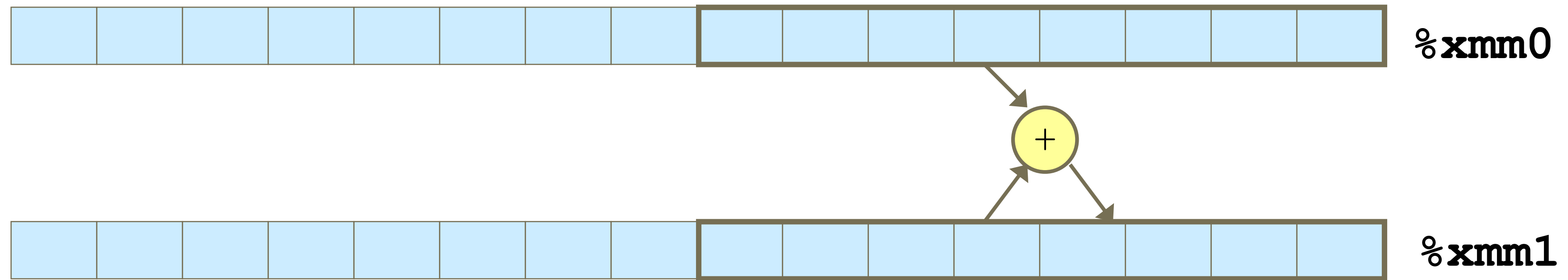


- SIMD Operations: Single Precision

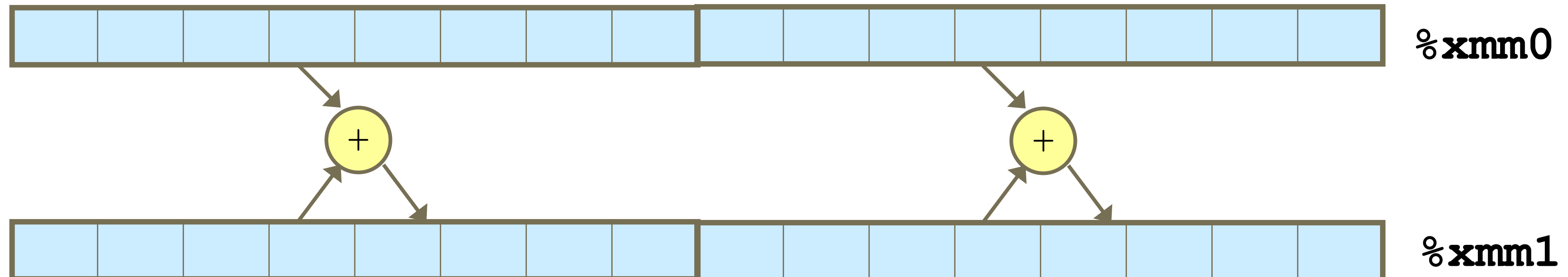


Scalar versus Vector (SIMD) instructions

- Scalar Operations: Double Precision



- SIMD Operations: Double Precision



Data movement instructions

```
float_mov:
# ---------
# float float_mov(float f1,
#                 float *src,
#                 float *dst) {
#   float f2 = *src;
#   *dst = f1;
#   return f2;
# }
# ---------
# f1 in %xmm0, src in %rdi, dst in %rsi
movss  (%rdi), %xmm1   # f2 = *src
movss  %xmm0, (%rsi)   # *dst = f1
movaps %xmm1, %xmm0    # return value = f2
ret
```

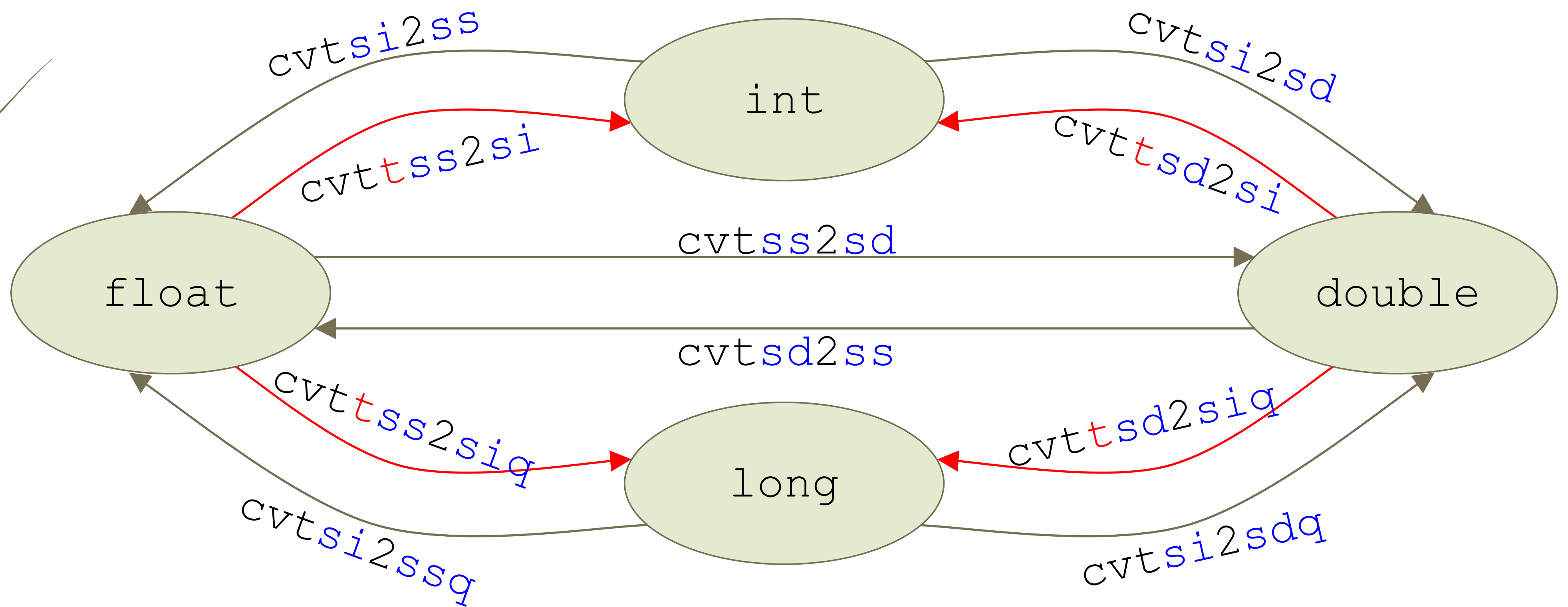
- The instructions we shall look at in this lecture are different than the ones presented in section 3.11 of our textbook – we shall focus on the scalar version of these instructions
- `movss` – move single precision
  - Mem (32 bits) <--> `%xmm`
- `movsd` – move double precision
  - Mem (64 bits) <--> `%xmm`
- First 2 instructions of program: Memory referencing operands (i.e., memory addressing mode operands) specified in the same way as for the integer `mov*` instructions
- `movaps`/`movapd` – move `%xmm` <--> `%xmm`
  - `ap` -> aligned packed

Function call and register saving conventions

- Function call convention
  - Integer (and pointer i.e., memory address) arguments passed in *integer* registers
  - Floating point values passed in XMM registers
  - Argument 1 to argument 8 passed in `%xmm0`, `%xmm1`, ..., `%xmm7`
  - Result returned in `%xmm0`
- Register saving convention
  - All XMM registers ***caller-saved***
  - Can use register `%xmm8` ↔ `%xmm15` for managing local data

Data conversion instructions

- Converting between data types: (“**t**” is for “truncate”)



Data manipulation instructions

Arithmetic

- addss/addsd - floating point add
- subss/subsd - … subtract
- mulss/mulsd - … mul
- divss/divsd - … div

Logical

- andps/andpd
- orps/d
- xorps/d
  - **xorpd %xmm0, %xmm0**

    effect **%xmm0 <- 0**

Comparison: ucomiss/d

- Affects only condition codes: CF, ZF
  - use unsigned branches
- If NaN, set all of condition codes: CF, ZF and PF
  - Use jp/jnp to branch on PF

Others

- maxss/maxsd - … max

  For example: maxss %xmm3, %xmm5

  Effect: xmm5 ← max(xmm5, xmm3)
- minss/minsd - … min
- sqrtss/sqrtsd - … square root

Example

```
fadd:
# ---------
# float fadd(float x, float y){
#    return x + y;
# }
# ---------
# x in %xmm0, y in %xmm1
  addss   %xmm1, %xmm0
  ret

dadd:
# ---------
# double dadd(double x, double y){
#    return x + y;
# }
# ---------
# x in %xmm0, y in %xmm1
  addsd   %xmm1, %xmm0
  ret
```

Storing Data in Various Segments of
Memory - Optional

Storing Data in Memory

This material is optional
–> It is for your
learning pleasure!

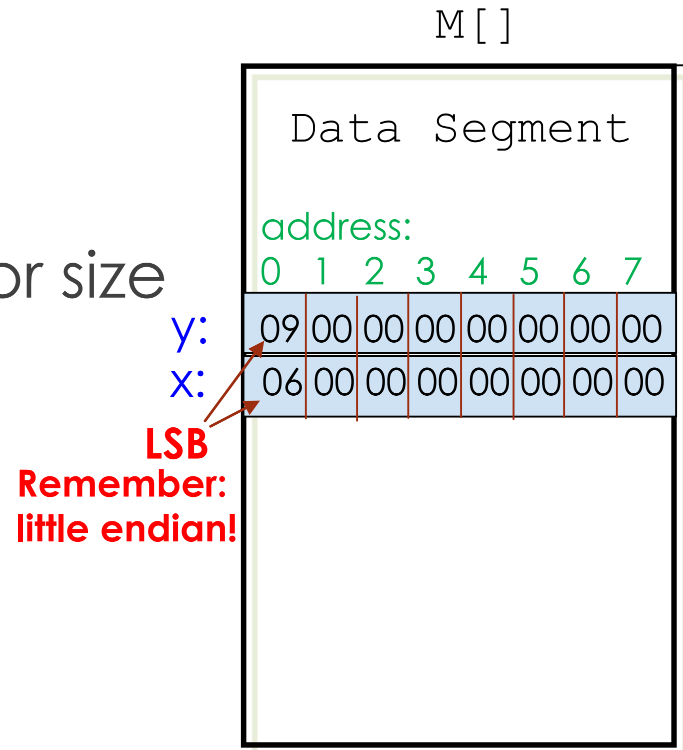
We already
know about
data on stack
and on heap.

- Data on stack memory (on stack frame of function)
  - Temporarily use and recycle
  - Lasts through life of function call
- Data on heap
  - Temporarily use and recycle
  - Lasts until memory is "free'ed"
- Data in fixed memory, i.e., Data segment
  - Statically allocated data
    - e.g., global variables, static variables, string constants
  - Lasts while program executes

What does this
type of data
look like?

This material is optional
-> It is for your
learning pleasure!

Data stored in Data Segment



- Declared using a label & a directive for size
  - label is a **memory address**
  - size: `.byte`, `.word`, `.long`, `.quad` (1, 2, 4, 8)
  - initial value
- **Example 1:** C:

```
long x = 6;
long y = 9;
void main {
    . . .
}
```

x86-64:

```
x: .quad 6 # 0x0000000000000006
y: .quad 9 # 0x0000000000000009
```

Data stored in Data Segment – **Example 2**

```c
#define N 6

int A[N] = {12, 34, 56, 78, -90, 1};

void main () {

    printf("The total is %d.\n", sum_array(A,N));

    return;
}
```

```
main:
.LFB38:
    .cfi_startproc
    subq    $8, %rsp
    .cfi_def_cfa_offset 16
    movl    $6, %esi
    movl    $A, %edi
    call    sum_array
    movl    $.LC0, %esi
    movl    %eax, %edx
    movl    $1, %edi
    xorl    %eax, %eax
    addq    $8, %rsp
    .cfi_def_cfa_offset 8
    jmp __printf_chk

     . . .
A:
     .long   12       # or .long 12, 34, 56, 78, -90, 1
     .long   34
     .long   56
     .long   78
     .long   -90
     .long   1
     .ident  "GCC: (Ubuntu 7.3.0-21ubuntu1~16.04) 7.3.0"
     .section    .note.GNU-stack,"",@progbits
```

This material is optional –> It is for your learning pleasure!

Data stored on Stack – **Example 1**

```
void main( int argc, char * argv ) {

    int A[ ] = {12, 34, 56, 78, -90, 1};

    printf("The total is %d.\n", sum_array(A,N));

    return;

}
```

```
main:
.LFB38:
        .cfi_startproc
        subq    $40, %rsp
        .cfi_def_cfa_offset 48
        movl    $6, %esi
        movq    %fs:40, %rax
        movq    %rax, 24(%rsp)
        xorl    %eax, %eax
        movabsq $146028888076, %rax
        movq    %rsp, %rdi
        movq    %rax, (%rsp)
        movabsq $335007449144, %rax
        movq    %rax, 8(%rsp)
        movabsq $8589934502, %rax
        movq    %rax, 16(%rsp)
        call    sum_array
        movl    $.LC0, %esi
        movl    %eax, %edx
        movl    $1, %edi
        xorl    %eax, %eax
        call    __printf_chk
        movq    24(%rsp), %rax
        xorq    %fs:40, %rax
        jne .L5
        addq    $40, %rsp
        .cfi_remember_state
        .cfi_def_cfa_offset 8
        ret
```

How does this large # end up representing 12 and 34:

- Express **$146028888076** in binary
- Transform binary to hex => **0x000000220000000c**
- Read hex's LSB (32 bits) (**0000000c**) as a decimal => **12**
- Read hex's MSB (32 bits) (**00000022**) as a decimal => **34**
- Repeat for other 2 operands of **movabsq** instructions

This material is optional –> It is for your learning pleasure!

Summary - 1

- What is a buffer overflow
  - When function writes more data in array than array can hold on stack
  - Effect: data kept on the stack (value of other local variables and registers, return address) may be corrupted

  -> *Stack smashing*
- Why buffer overflow spells **trouble** -> it creates vulnerability
  - Allowing hacker attacks
- How to protect system against such attacks
  1. Avoid creating overflow vulnerabilities in the code that we write
     - By always checking bounds and calling "safe" library functions that consider size of array
  2. Employ system-level protections
     - Randomized initial stack pointer and non-executable code segments
  3. Use compiler (like `gcc`) security features:
     - Stack "canary" value and **`endbr64`** instruction

Summary - 2

- Floating point data and operations
  - Data held and manipulated in XMM registers
  - Assembly language instructions similar to *integer* assembly language instructions we have seen so far

Next Lecture

Start a new unit …

- Instruction Set Architecture (ISA)