Q. Why do assembly programmers need to know how to swim?
A. Because they work below C level!

x86-64 Miscellaneous Topics
Topics

• Homogeneous data structures
  – Arrays
  – Nested / multidimensional arrays

• Heterogeneous data structures
  – Structs / records
  – Unions

• Floating-point code
Arrays

- An array is simply a block of memory (*bits*)
  - Fixed-sized *homogeneous* elements of a particular type (*context*)
  - Contiguous layout
  - Fixed length (not stored as part of the array!)

```c
int32_t stuff[3];

3 elements
each element is 4 bytes wide
total size is 3 * 4 = 12 bytes
```

```c
stuff[0] = 7
stuff[1] = 7
stuff[2] = 7
```

```assembly
movq $0x600100, %rbx
movl $7, (%rbx)
movl $7, 4(%rbx)
movl $7, 8(%rbx)
```
Arrays and pointers

- Array name is essentially a pointer to first element (base)
  - The $i$th element is at address $(\text{base} + \text{size} \times i)$
- C pointer arithmetic uses intervals of the element width
  - No need to explicitly multiply by size in C
  - “stuff+0” or “stuff” is the address of the first element
  - “stuff+1” is the address of the second element
  - “stuff+2” is the address of the third element
- Indexing = pointer arithmetic plus dereferencing
  - “stuff[i]” means “*(stuff + i)”
  - In assembly, use the scaled index addressing mode
    - $(\text{base, index, scale}) \rightarrow$ e.g., (%rbx, %rdi, 4) for 32-bit elements
Fill in the blank to correctly translate the following C code into x86-64:

```c
int64_t data[10];

for (int i = 0; i < 10; i++) {
    data[i] = 0;
}
```

```
movq $0x600100, %rbx
movq $0, %rdx
jmp L2
L1:
    movq $0, %rdx
    incq %rdx
L2:
    cmpq $10, $rdx
    jl L1
```
Fill in the blank to correctly translate the following C code into x86-64:

```c
int64_t data[10];

for (int i = 0; i < 10; i++) {
    data[i] = 0;
}
```
Nested / multidimensional arrays

- Generalizes cleanly to multiple dimensions
  - Think of the elements of outer dimensions as being arrays of inner dimensions
  - “Row-major” order: outer dimension specified first
  - E.g., “int16_t grid[4][3]” is a 4-element array of 3-element arrays of 16-bit integers
  - 2D: Address of \((i,j)\)th element is \((base + size(cols \times i + j))\)
  - 3D: Address of \((i,j,k)\)th element is \((base + size((n_{d1} \times n_{d2}) \times i + n_{d2} \times j + k))\)
C structs are also just regions of memory

- “Structured” heterogeneous regions—they’re split into fields
- Contiguous layout (w/ occasional gaps for alignment)
- Offset of each field can be determined by the compiler
- Sometimes called “records” generally

```c
struct {
    int i;
    int j;
    int a[2];
    int *p;
} x;
```

```assembly
(%rbx = &x and %rdi = 1)
    movl $1, (%rbx)
    movl $2, 4(%rbx)
    movl $3, 8(%rbx)
    movl $4, 8(%rbx, %rdi, 4)
    movq $0, 16(%rbx)
```

<table>
<thead>
<tr>
<th>Offset</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>i</td>
</tr>
<tr>
<td>4</td>
<td>j</td>
</tr>
<tr>
<td>8</td>
<td>a[0]</td>
</tr>
<tr>
<td>16</td>
<td>a[1]</td>
</tr>
<tr>
<td>24</td>
<td>p</td>
</tr>
</tbody>
</table>
**Alignment**

- **Alignment restrictions** require addresses be $n$-divisible
  - E.g., 4-byte alignment means all addresses must be divisible by 4
  - Specified using an assembler directive
  - Improves memory performance if the hardware matches
  - Can be avoided in C using “attribute (packed)” (as in elf.h)

```c
struct {
    int i;
    char c;
    int j;
} rec;
```

![Diagram showing alignment options]

- None
- 2-byte
- 4-byte
- 8-byte
C **unions** are also just regions of memory

- Can store one “thing”, but it could be multiple sizes depending on what kind of “thing” it currently is (so context is even more important!)
- All “fields” start at offset zero
- Generally a bad idea! (circumvents the type system in C)
- Can be used to do OOP in C (i.e., polymorphism)

typedef enum { CHAR, INT, FLOAT } objtype_t;

typedef struct {
    objtype_t type;
    union {
        char c;
        int i;
        float f;
    } data;
} obj_t;

obj_t foo;
foo.type = INT;
foo.data.i = 65;
printf("%c", foo.data.c);  ← VALID!
Aside: Enums

• **Enumerations** are types where all values are listed
  - Declared in C using `enum` keyword
  - In C, the actual values are stored as integers
  - Can assign integer values if desired
  - Primary advantage: named constants

```c
typedef enum {
    MON = 1, TUE, WED, THU, FRI, SAT, SUN
} day_t;

// essentially the same as: int midterm_day = 3;
day_t midterm_day = WED;
```
Floating-point code

- **x87**: extension of x86 for floating-point arithmetic
  - Originally for the 8087 floating-point co-processor
  - Adds new floating-point "stack" registers \( ST(0) - ST(7) \)
    - 80-bit **extended double** format (15 exponent and 63 significand bits)
  - Push/pop with **FLD** and **FST** instructions
  - Arithmetic: **FADD**, **FMUL**, **FSQRT**, etc.
  - Largely deprecated now in favor of new SIMD architectures
Floating-point code

- **Single-Instruction, Multiple-Data (SIMD)**
  - Performs the same operation on multiple pairs of elements
  - Also known as **vector** instructions
- **Various floating-point SIMD instruction sets**
  - MMX, **SSE**, **SSE2**, SSE3, SSE4, SSE5, **AVX**, **AVX2**
  - 16 new extra-wide XMM (128-bit) or YMM (256-bit) registers for holding multiple elements
    - Floating-point arguments passed in %xmm0-%xmm7
    - Return value in %xmm0
    - All registers are caller-saved
Floating-point code

- **SSE** (Streaming SIMD Extensions)
  - 128-bit XMM registers
    - Can store two 64-bit doubles or four 32-bit floats
  - New instructions for movement and arithmetic
    - General form: \(<op><s|p><s|d>\)
    - \(<s|p>: s=scalar\ (single\ data)\ p=packed\ (multiple\ data)\)
    - \(<s|d>: s=single\ (32-bit)\ d=double\ (64-bit)\)
    - E.g., “addsd” = add scalar 64-bit doubles
    - E.g., “mulps = multiply packed 32-bit floats
- **AVX** (Advanced Vector Extensions)
  - 256-bit YMM registers
    - Can store four 64-bit doubles or eight 32-bit floats
  - Similar instructions as SSE (but with “v” prefix, e.g., vmulps)
• **Movement**
  - `movss` / `movsd`
  - `movaps` / `movapd`

• **Conversion**
  - `cvtsi2ss` / `cvtsi2sd`
  - `cvtss2si` / `cvtss2sd`
  - `cvtss2sd` / `cvtsd2ss`

• **Arithmetic**
  - `addss` / `addsd`
  - `addps` / `addpd`
  - `...` (sub, mul, div, max, min, sqrt)
  - `andps` / `andpd`
  - `xorps` / `xorpd`

• **Comparison**
  - `ucomiss` / `ucomisd`

(AVX has "v___" opcodes)
Bitwise operations in SSE/AVX

- Assembly instructions provide low-level access to floating-point numbers
  - Some numeric operations can be done more efficiently with simple bitwise operations

- AKA: Floating-Point Hacks™
  - Set to zero (value XOR value)
  - Absolute value (value AND 0x7fffffff)
  - Additive inverse (value XOR 0x80000000)

- Lesson: Information = Bits + Context
  - (even if it wasn't the intended context!)
Y86-64 ISA

<table>
<thead>
<tr>
<th>Byte</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>halt</td>
<td>00</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>nop</td>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rmovq rA, rB</td>
<td>20 rA rB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>irmovq V, rB</td>
<td>30 F rB V</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rmovq rA, D(rB)</td>
<td>40 rA rB D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mrmovq D(rB), rA</td>
<td>50 rA rB D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OPq rA, rB</td>
<td>6 fn rA rB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>jXX Dest</td>
<td>7 fn Dest</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cmovXX rA, rB</td>
<td>2 fn rA rB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>call Dest</td>
<td>8 0 Dest</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ret</td>
<td>9 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pushq rA</td>
<td>A 0 rA F</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>popq rA</td>
<td>B 0 rA F</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Not in CS:APP:

Itotrap id

C id

<table>
<thead>
<tr>
<th>Number</th>
<th>Register name</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>%rax</td>
</tr>
<tr>
<td>1</td>
<td>%rcx</td>
</tr>
<tr>
<td>2</td>
<td>%rdx</td>
</tr>
<tr>
<td>3</td>
<td>%rbx</td>
</tr>
<tr>
<td>4</td>
<td>%rsp</td>
</tr>
<tr>
<td>5</td>
<td>%rbp</td>
</tr>
<tr>
<td>6</td>
<td>%rsi</td>
</tr>
<tr>
<td>7</td>
<td>%rdi</td>
</tr>
<tr>
<td>8</td>
<td>%r8</td>
</tr>
<tr>
<td>9</td>
<td>%r9</td>
</tr>
<tr>
<td>10</td>
<td>%r10</td>
</tr>
<tr>
<td>11</td>
<td>%r11</td>
</tr>
<tr>
<td>12</td>
<td>%r12</td>
</tr>
<tr>
<td>13</td>
<td>%r13</td>
</tr>
<tr>
<td>14</td>
<td>%r14</td>
</tr>
</tbody>
</table>

Value | Name       | Meaning
------|------------|
1      | AOK        | Normal operation
2      | HLT        | halt instruction encountered
3      | ADR        | Invalid address encountered
4      | INS        | Invalid instruction encountered

RF: Program registers

%rax | %rsp | %r8 | %r12
%rcx | %rbp | %r9 | %r13
%rdx | %rsi | %r10 | %r14
%rbx | %rdi | %r11

CC: Condition codes

ZF SF OF

DMEM: Memory

PC

Stat: Program status

[Blank]