CS 261
Fall 2018
Mike Lam, Professor

x86-64 Data Structures and Misc. Topics
Topics

• Homogeneous data structures
  – Arrays
  – Nested / multidimensional arrays
• Heterogeneous data structures
  – Structs / records
  – Unions
• Floating-point code
• Misc. notes
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

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

```ml
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 $\text{(base + size * } i\text{)}$

- 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
    - $(base, index, scale)$ → e.g., (%rbx, %rdi, 4) for 32-bit elements
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 \((\text{base} + \text{size}(\text{cols} \times i + j))\)
  - 3D: Address of \((i,j,k)\)th element is \((\text{base} + \text{size}((n_{d1} \times n_{d2}) \times i + n_{d2} \times j + k))\)
• 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;

(struct { x.i = 1; x.j = 2; x.a[0] = 3; x.a[1] = 4; x.p = NULL; (%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`)

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

<table>
<thead>
<tr>
<th>Alignment</th>
<th>Locations</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>i</td>
</tr>
<tr>
<td>2-byte</td>
<td>i</td>
</tr>
<tr>
<td>4-byte</td>
<td>i</td>
</tr>
<tr>
<td>8-byte</td>
<td>i</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Address</th>
<th>0</th>
<th>4</th>
<th>8</th>
<th>12</th>
<th>16</th>
<th>20</th>
<th>24</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-4</td>
<td>None</td>
<td>2-byte</td>
<td>4-byte</td>
<td>8-byte</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4-8</td>
<td>None</td>
<td>2-byte</td>
<td>4-byte</td>
<td>8-byte</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8-12</td>
<td>None</td>
<td>2-byte</td>
<td>4-byte</td>
<td>8-byte</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12-16</td>
<td>None</td>
<td>2-byte</td>
<td>4-byte</td>
<td>8-byte</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16-20</td>
<td>None</td>
<td>2-byte</td>
<td>4-byte</td>
<td>8-byte</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20-24</td>
<td>None</td>
<td>2-byte</td>
<td>4-byte</td>
<td>8-byte</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
• 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)

```c
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!
```
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` = add 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`)
**SSE/AVX**

- **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: Stupid 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!)
Aside: Opcode Suffixes

- We've been assuming that the operand size suffix for opcodes is mandatory
  - E.g., the "l" or "q" in "movl" or "movq"
- Technically, it is only required if it cannot be inferred
  - E.g., \texttt{mov} \%eax, \%edi is not ambiguous
    - We can infer that this is a 32-bit move because of the destination
  - However, \texttt{mov} $2, (%rdx) is ambiguous
    - Is it a 8-bit move? 32 bits? 64 bits?
    - A suffix is required here (e.g., \texttt{movl} $2, (%rdx) for 32 bits)
  - Generally, it is safer always to include the suffix
Aside: Memory Restrictions

- In x86-64, most opcodes have no memory -> memory form
  - I.e., you can't read and write memory in the same instruction
  - Invalid: `movl (%rax), (%rdx)`

- Solution: use a temporary register
  ```
  movl (%rax), %ecx
  movl %ecx, (%rdx)
  ```
Aside: 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>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>nop</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rrmovq rA, rB</td>
<td>2</td>
<td>0</td>
<td>rA</td>
<td>rB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>irmovq V, rB</td>
<td>3</td>
<td>0</td>
<td>F</td>
<td>rB</td>
<td>V</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rrmovq rA, D(rB)</td>
<td>4</td>
<td>0</td>
<td>rA</td>
<td>rB</td>
<td>D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mrmovq D(rB), rA</td>
<td>5</td>
<td>0</td>
<td>rA</td>
<td>rB</td>
<td>D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OPq rA, rB</td>
<td>6</td>
<td>fn</td>
<td>rA</td>
<td>rB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>jXX Dest</td>
<td>7</td>
<td>fn</td>
<td>Dest</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cmovXX rA, rB</td>
<td>2</td>
<td>fn</td>
<td>rA</td>
<td>rB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>call Dest</td>
<td>8</td>
<td>0</td>
<td>Dest</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ret</td>
<td>9</td>
<td>0</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</td>
<td>0</td>
<td>rA</td>
<td>F</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>popq rA</td>
<td>B</td>
<td>0</td>
<td>rA</td>
<td>F</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Operations | Branches | Moves |
--- | --- | --- |
addq 6 | 0 | jmp 7 | 0 | jne 7 | 4 | rrmovq 2 | 0 | cmovne 2 | 4 |
subq 6 | 1 | jle 7 | 1 | jge 7 | 5 | cmovle 2 | 1 | cmovge 2 | 5 |
andq 6 | 2 | jl 7 | 2 | jg 7 | 6 | cmovl 2 | 2 | cmovg 2 | 6 |
xorq 6 | 3 | je 7 | 3 | cmov 2 | 3 |

Number | Register name |
--- | --- |
0 | %rax |
1 | %rcx |
2 | %rdx |
3 | %rpx |
4 | %rsp |
5 | %rbp |
6 | %rsi |
7 | %rdi |
8 | %r8 |
9 | %r9 |
10 | %r10 |
11 | %r11 |
12 | %r12 |
13 | %r13 |
14 | %r14 |

Value | Name | Meaning |
--- | --- | --- |
1 | A0K | Normal operation |
2 | HLT | halt instruction encountered |
3 | ADR | Invalid address encountered |
4 | INS | Invalid instruction encountered |

RF: Program registers

<table>
<thead>
<tr>
<th>%rax</th>
<th>%rsp</th>
<th>%r8</th>
<th>%r12</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rcx</td>
<td>%rbp</td>
<td>%r9</td>
<td>%r13</td>
</tr>
<tr>
<td>%rdx</td>
<td>%rsi</td>
<td>%r10</td>
<td>%r14</td>
</tr>
<tr>
<td>%rpx</td>
<td>%rdi</td>
<td>%r11</td>
<td></td>
</tr>
</tbody>
</table>

CC: Condition codes

<table>
<thead>
<tr>
<th>ZF</th>
<th>SF</th>
<th>OF</th>
</tr>
</thead>
</table>

Stat: Program status

<table>
<thead>
<tr>
<th>PC</th>
</tr>
</thead>
</table>

DMEM: Memory