**Structures & Alignment**

- **Unaligned Data**
  - Primitive data types require 8 bytes
  - Address must be multiple of 8

- **Aligned Data**
  - Primitive data type requires 8 bytes
  - Address must be multiple of 8
  - Required on some machines, advised on x86-64

**Alignment Principles**

- **Aligned Data**
  - Memory accessed by (aligned) chunks of 4 or 8 bytes (system dependent)
  - Inefficient to load or store datum that spans quad word boundaries
  - Virtual memory trickier when datum spans 2 pages

- **Compiler**
  - Inserts gaps in structure to ensure correct alignment of fields

**Satisfying Alignment with Structures**

- **Within structure:**
  - Must satisfy each element's alignment requirement
  - Overall structure placement
    - Each structure has alignment requirement $K$ (largest alignment of any element)
    - Initial address & structure length must be multiples of $K$

- **Example:**
  - $K = 8$, due to double element

**Meeting Overall Alignment Requirement**

- **For largest alignment requirement $K$**
  - Overall structure must be multiple of $K$

**Arrays of Structures**

- **Overall structure length multiple of $K$**
- **Satisfy alignment requirement for every element

**Accessing Array Elements**

- **Compute array offset $12*\text{idx}$**
- **Align offset $a$ within structure**
- **Assembler gives offset $a$ $B$**

**Saving Space**

- **Put large data types first**

**Specific Cases of Alignment (x86-64)**

- **1 byte:** char, ...
  - no restrictions on address
- **2 bytes:** short, ...
  - lowest 1 bit of address must be 0
- **4 bytes:** int, float, ...
  - lowest 2 bits of address must be 00
- **8 bytes:** double, long, char *
  - lowest 3 bits of address must be 000
- **16 bytes:** long double (GCC on Linux)
  - lowest 4 bits of address must be 0000

**Turning C into Object Code**

- Code in files p1.c p2.c
  - Compiler with command gcc -o p1.o p2.c -o p1
  - Use basic optimisations (-Og) [New in recent versions of GCC]

- **Put resulting binary in file p**
  - text
    - C program (p1.o p2.o)
  - Assembler (gcc or as)
    - text
      - Object program (p1.o p2.o)
    - Linker (gcc or ld)
      - Binary
        - Executable program (p)
Generating x86-64 Assembly

C Code

```c
long t = plus(x, y);
```

Disassembly

```
Disassembled
```

Object Code

```
Code for sumstore
```

Machine Instruction Example

```
```

x86-64 Integer Registers

```
```

movq Operand Combinations

```
```

Origin:

```
```

Some History: IA32 Registers

```
```

Alternate Disassembly

```
```

Some History: IA32 Registers

```
```

Disassembling Object Code

```
```

Alternate Disassembly

```
```

Disassembled

```
```

Some History: IA32 Registers

```
```

Moving Data

```
```

Compiling Into Assembly
Simple Memory Addressing Modes

**Normal (R) Mem[Reg[R]]**
- Register R specifies memory address
- Aha! Pointer dereferencing in C
  ```
  movq (%rcx), %rax
  ```

**Displacement (R) Mem[Reg[R]+D]**
- Register R specifies start of memory region
- Constant displacement D specifies offset
  ```
  movq 8(%rbp), %rdx
  ```

Example of Simple Addressing Modes

### Example Code:
```c
void swap(long *xp, long *yp)
{
    long t0 = *xp;
    long t1 = *yp;
    *xp = t1;
    *yp = t0;
}
```

### Assembly Code:
```assembly
swap:
    movq (%rdi), %rax  # t0 = *xp
    movq (%rsi), %rdx  # t1 = *yp
    movq %rdx, (%rdi)  # *xp = t1
    movq %rax, (%rsi)  # *yp = t0
    ret
```

Understanding Swap()
### Complete Memory Addressing Modes

- **Most General Form**
  - D(Rh,Ri) \( \text{Mem}[(\text{Reg}Rh) \times S + \text{Reg}Ri + D] \)
  - **D**: Constant displacement 1, 2, or 4 bytes.
  - **Rh**: Base register: Any of 16 integer registers.
  - **Ri**: Index register: Any, except for **Rsp**.
  - **S**: Scale: 1, 2, 4, or 8 (why show numbers?)

- **Special Cases**
  - (Rh,Ri) \( \text{Mem}[(\text{Reg}Rh) \times S + \text{Reg}Ri] \)
  - (Rh,Ri) \( \text{Mem}[(\text{Reg}Rh) \times S + \text{Reg}Ri + D] \)
  - (Rh,Ri) \( \text{Mem}[(\text{Reg}Rh) \times S + \text{Reg}Ri] \)

### Address Computation Examples

<table>
<thead>
<tr>
<th>Expression</th>
<th>Address Computation</th>
<th>Address</th>
</tr>
</thead>
</table>
| ```
    leaq (%rdi,%rsi,4)
``` | 0x80(%rsi,%rsi,2), %rdi | 0xf400 |
| ```
    leaq [(%rsi),%rdi]
``` | 2*0xf000 + 0x100, %rdi | 0xf400 |
| ```
    leaq [0x8(%rdx),%rdi]
``` | 0xf000 + 0x80, %rdx | 0xf400 |
| ```
    leaq [0x1e080,%rdx]
``` | 0x0100, %rdx | 0xf400 |

### Address Computation Instruction

- **1leaq Src, Dest**
  - Src = address mode expression
  - Dest = address denoted by expression

- **Uses**
  - Computing addresses without a memory reference
  - E.g., translation of \( p = ax[i] \)
  - Computing arithmetic expressions of the form \( x + k \cdot y \)
  - \( k = 1, 2, 4, \text{or } 8 \)

### Some Arithmetic Operations

- **Two Operand Instructions**
  - ```
      addq Src,Dest
      dest = dest + src
      src = src
      also called add
    ```
  - ```
      subq Src, Dest
      dest = dest - src
      src = src
      also called sub
    ```
  - ```
      imulq Src, Dest
      dest = dest \times src
      src = src
      also called mul
      arithmetic
    ```
  - ```
      shrq Src, Dest
      dest = dest >> src
      src = src
      logical
    ```
  - ```
      andq Src, Dest
      dest = dest \& src
      src = src
      also called and
      logical
    ```
  - ```
      orq Src, Dest
      dest = dest | src
      src = src
      also called or
      logical
    ```
  - ```
      notq Dest
      dest = dest
      also called not
      logical
    ```

- **One Operand Instructions**
  - ```
      incq Dest
      dest = dest + 1
    ```
  - ```
      decq Dest
      dest = dest - 1
    ```
  - ```
      negq Dest
      dest = dest
      also called neg
      arithmetic
    ```
  - ```
      shrq Dest
      dest = dest >> src
      src = src
      logical
    ```
  - ```
      andq Dest
      dest = dest \& src
      src = src
      also called and
      logical
    ```
  - ```
      orq Dest
      dest = dest | src
      src = src
      also called or
      logical
    ```
  - ```
      notq Dest
      dest = dest
      also called not
      logical
    ```

### Understanding Arithmetic Expression Example

### Arithmetic Expression Example

### Address Computation Instruction

### Some Arithmetic Operations

### Understanding Arithmetic Expression Example

### Processor State (x86-64, Partial)

- **Information about currently executing program**
  - Temporary data
  - Location of runtime stack
  - Location of current code control point
  - Status of recent tests
  - Current stack top

### Registers

- [Register Name]
Condition Codes (Implicit Setting)

- Single bit registers
  - CF: Carry Flag (for unsigned)
  - SF: Sign Flag (for signed)
  - ZF: Zero Flag
  - OF: Overflow Flag (for signed)

- Implicitly set (think of it as side effect) by arithmetic operations
  - Example: `addq Src, Dest ↔ t = a + b`
    - CF set if carry out from most significant bit (unsigned overflow)
    - ZF set if t == 0
    - SF set if t < 0 (as signed)
    - OF set if two’s-complement (signed) overflow
      \[ (a>0 && b>0 && t<0) || (a<0 && b<0 && t>0) \]
    - Not set by `leaq` instruction

Condition Codes (Explicit Setting: Compare)

- Explicit Setting by Compare Instruction
  - `cmpq` (16x2, 32x1)
  - Compare `b,a` like computing `a - b` without setting destination
    - CF set if carry out from most significant bit (used for unsigned comparisons)
    - ZF set if `a == b`
    - SF set if `(a-b) < 0` (as signed)
    - OF set if two’s-complement (signed) overflow
      \[ (a>0 && b<0 && (a-b)<0) || (a<0 && b>0 && (a-b)>0) \]

Condition Codes (Explicit Setting: Test)

- Explicit Setting by Test Instruction
  - `testq` (16x2, 32x1)
  - Like computing `a & b` without setting destination
    - ZF set when `a & b == 0`
    - SF set when `a & b < 0`

Reading Condition Codes

- **SetX Instructions**
  - Set low-order byte of destination to 0 or 1 based on combinations of condition codes
  - Does not alter remaining 7 bytes

- **SetX Instructions (Cont.)**
  - One of addressable byte registers
    - Typically use `movzbl` to finish job
      - 32-bit instructions also set upper 32 bits to 0

- **Jumping**
  - JX Instructions
    - Jump to different part of code depending on condition codes

<table>
<thead>
<tr>
<th>X</th>
<th>Condition Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Equal or Zero</td>
</tr>
<tr>
<td>1</td>
<td>Not Equal or Zero</td>
</tr>
<tr>
<td>2</td>
<td>Greater or Equal</td>
</tr>
<tr>
<td>3</td>
<td>Less or Equal</td>
</tr>
<tr>
<td>4</td>
<td>Greater</td>
</tr>
<tr>
<td>5</td>
<td>Less</td>
</tr>
<tr>
<td>6</td>
<td>Signed (unsigned)</td>
</tr>
<tr>
<td>7</td>
<td>Signed</td>
</tr>
</tbody>
</table>

Example:

```
cmpq %rsi, %rdi   # Compare x:y
setg %al          # Set when >
movzbl %al, %rax  # Zero rest of true ext
```