Machine/Assembler Language
Putting It All Together

Noah Mendelsohn
Tufts University
Email: noah@cs.tufts.edu
Web: http://www.cs.tufts.edu/~noah
Goals for today

- Explore the compilation of realistic programs from C to ASM
- Fill in a few more details on things like structure mapping
### X86-64 / AMD 64 / IA 64 General Purpose Registers

<table>
<thead>
<tr>
<th>Register</th>
<th>63</th>
<th>31</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rax</td>
<td>%eax</td>
<td>%ax</td>
<td>%ah</td>
</tr>
<tr>
<td>%rcx</td>
<td>%ecx</td>
<td>%cx</td>
<td>%ch</td>
</tr>
<tr>
<td>%rdx</td>
<td>%edx</td>
<td>%dx</td>
<td>%dh</td>
</tr>
<tr>
<td>%rbx</td>
<td>%ebx</td>
<td>%bx</td>
<td>%bh</td>
</tr>
<tr>
<td>%rsi</td>
<td>%esi</td>
<td>%si</td>
<td></td>
</tr>
<tr>
<td>%rdi</td>
<td>%edi</td>
<td>%di</td>
<td></td>
</tr>
<tr>
<td>%rsp</td>
<td>%esp</td>
<td>%sp</td>
<td></td>
</tr>
<tr>
<td>%rbp</td>
<td>%ebp</td>
<td>%bp</td>
<td></td>
</tr>
<tr>
<td>%r8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%r9</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%r10</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%r11</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%r12</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%r13</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%r14</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%r15</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Machine code (typical)

- **Simple instructions** – each does small unit of work
- Stored in memory
- *Bitpacked* into **compact binary form**
- Directly executed by transistor/hardware logic*

* We’ll show later that some machines execute user-provided machine code directly, some convert it to an even lower level machine code and execute that.
A simple function in C

```c
int times16(int i)
{
    return i * 16;
}
```

REMEMBER: you can see the assembler code for any C program by running gcc with the –S flag. Do it!!
Addressing modes

%rax                      // contents of rax is data
(%rax)                    // data pointed to by rax
$123,%rax                 // Copy 123 into %rax
0x10(%rax)                // get *(16 + rax)
$0x4089a0(, %rax, 8)       // Global array index
                          // of 8-byte things
(%ebx, %ecx, 2)           // Add base and scaled index
4(%ebx, %ecx, 2)          // index plus offset 4
movl and leal

%rax
(%rax)
$123, %rax
0x10(%rax)
$0x4089a0(, %rax, 8)
(%ebx, %ecx, 2)
4(%ebx, %ecx, 2)

movl (%ebx, %ecx, 1), %edx
leal (%ebx, %ecx, 1), %edx

// contents of rax is data
// data pointed to by rax
// Copy 123 into %rax
// get *(16 + rax)
// Global array index
// of 8-byte things
// Add base and scaled index
// Add base and scaled
// index plus offset 4
// edx <- *(ebx + (ecx * 1))
// edx <- (ebx + (ecx * 1))
movl and leal

movl

Moves the data at the address:

similar to:

char *ebxp;
char edx = ebxp[ecx]

%rax
(%rax)                             // data pointed to by rax
$123,%rax                          // Copy 123 into %rax
0x10(%rax)                         // get *(16 + rax)
$0x4089a0(, %rax, 8)               // Global array index

%ebx, %ecx, 2
4(%ebx, %ecx, 2)                   // Add base and scaled index

movl (%ebx, %ecx, 1), %edx       // edx <- *(ebx + (ecx * 1))
leal (%ebx, %ecx, 1), %edx        // edx <- (ebx + (ecx * 1))
mobl and leal

%rax
(%rax)
$123,%rax
0x10(%rax)
$0x4089a0
(%ebx, %ecx, 1)
4(%ebx, %ecx, 2)

movl (%ebx, %ecx, 1), %edx  // edx <-(ebx + (ecx * 1))
leal (%ebx, %ecx, 1), %edx  // edx <- (ebx + (ecx * 1))


leal

Moves the address itself:

similar to:

char *ebxp;
char *edxp = &(ebxp[ecx]);

Moves

Moves the address itself:

similar to:

char *ebxp;
char *edxp = &(ebxp[ecx]);
movl and leal

%rax
(%rax)
$123,%rax
0x10(%rax)
$0x4089a0
(%ebx, %ecx, 2)
4(%ebx, %ecx, 2)
movl (%ebx, %ecx, 1), %edx  // edx <- *(ebx + (ecx * 1))
leal (%ebx, %ecx, 1), %edx  // edx <-  (ebx + (ecx * 1))
movl (%ebx, %ecx, 4), %edx  // edx <- * (ebx + (ecx * 4))
leal (%ebx, %ecx, 4), %edx  // edx <-  (ebx + (ecx * 4))

scale factors support indexing larger types

similar to:

```c
int *ebxp;
int edxp = ebx[p][ecx];
```
Control flow: simple jumps

.L4:

...code here...

j .L4 // jump back to L4
Conditional jumps

.L4:

```assembly
movq (%rdi,%rdx), %rcx
leaq (%rax,%rcx), %rsi
testq ...code here...
cmovg %rsi, %rax
addq $8, %rdx
cmpq %r8, %rdx
jne .L4       // conditional: jump iff %r8 != %rdf
```

This technique is the key to compiling if statements, for loops, while loops, etc.
Challenge: how would this compile?

```c
switch (exp) {
    case 0: SS0;
    case 1: SS1;
    case 2: SS2;
    case 3: SS3:
    default: SSd;
}
```

```c
static code *jump_table[4] =
    { L0, L1, L2, L3 };

t = exp;
if ((unsigned)t < 4) {
    t = jump_table[t];
    goto t;
} else
    goto Ld;
```

L0: SS0;
L1: SS1;
L2: SS2;
L3: SS3:
Ld: SSd;
Lend: ...

And break becomes goto Lend
Calling Functions (Review)
Function calls on Linux/AMD 64

- Caller “pushes” return address on stack
- Where practical, arguments passed in registers
- Exceptions:
  - Structs, etc.
  - Too many
  - What can’t be passed in registers is at known offsets from stack pointer!
- Return values
  - In register, typically %rax for integers and pointers
  - Exception: structures
- Each function gets a stack frame
  - Leaf functions that make no calls may skip setting one up
Arguments/return values in registers

<table>
<thead>
<tr>
<th>Operand Size</th>
<th>Argument Number</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td>64</td>
<td>%rdi</td>
</tr>
<tr>
<td>32</td>
<td>%edi</td>
</tr>
<tr>
<td>16</td>
<td>%di</td>
</tr>
<tr>
<td>8</td>
<td>%dil</td>
</tr>
</tbody>
</table>

Arguments and return values passed in registers when types are suitable and when there aren’t too many

Return values usually in %rax, %eax, etc.

Callee may change these and *most* other registers. However, callee must *not* change %rsp; %rbp, %rbx, %r12 – %r15!

MMX and FP 87 registers used for floating point

Read the specifications for full details!
The stack – general case

Before call

- Arguments
- Return address

After call

- Arguments
- Return address

If callee needs frame

- Arguments
- Return address
- Callee vars
- Args to next call?

sub $0x{framesize},%rsp
A simple function

```c
unsigned int times2(unsigned int a) {
    return a * 2;
}
```

Double the argument

```asm
    leaq (%rdi,%rdi), %rax
    ret
```

...and put it in return value register
The stack – general case

Before call

Must be multiple of 16 when a function is called!

But now it’s not a multiple of 16...

Here too!

Arguments

????

%rsp

Arguments

Return address

%rsp

Arguments

Return address

Callee vars

%rsp

Args to next call?

framesize

sub $0x{framesize},%rsp
The stack – general case

Before call

Must be multiple of 16 when a function is called!

If callee needs frame

If we’ll be calling other functions with no arguments, very common to see

```asm
sub $8, %rsp
```
to re-establish alignment
Getting the details on function call “linkages”

- **Bryant and O’Halloran** has excellent introduction
  - Watch for differences between 32 and 64 bit

- **The official specification:**
  - System V Application Binary Interface: AMD64 Architecture Processor Supplement
  - See especially sections 3.1 and 3.2
  - E.g., You’ll see that `%rbp`, `%rbx`, `%r12` – `%r15` are callee saves. All other GPRs are caller saves. See also Figure 3.4.
Factorial Revisited

```c
int fact(int n)
{
    if (n == 0)
        return 1;
    else
        return n * fact(n - 1);
}
```

```
fact:
  .LFB2:
PUSHQ %RBX
  .LCFI0:
  MOVQ %RDI, %RBX
  MOVL $1, %EAX
  TESTQ %RDI, %RDI
    JE .L4
  LEAQ -1(%RDI), %RDI
  CALL fact
  IMULQ %RBX, %RAX
  .L4:
  POPQ %RBX
  RET
```
An example that integrates what we’ve learned
unsigned long
sum_positives(long arr[], unsigned int length) {
    unsigned long sum = 0;
    unsigned int i;

    for (i = 0; i < length; i++) {
        if (arr[i] > 0)
            sum += arr[i];
    }

    return sum;
}
Sum positive numbers in an array

```c
unsigned long
sum_positives(long arr[],
    unsigned int length)
{
    unsigned long sum = 0;
    unsigned int i;

    for (i = 0; i < length; i++) {
        if (arr[i] > 0)
            sum += arr[i];
    }

    return sum;
}
```

```
sum_positives:
    .LFB0:
        movl $0, %eax
        testl %esi, %esi
        je .L2
        subl $1, %esi
        leaq 8(%rsi,8), %r8
        movl $0, %edx
        .L4:
            movq (%rdi,%rdx), %rcx
            leaq (%rax,%rcx), %rsi
            testq %rcx, %rcx
            cmovg %rsi, %rax
            addq $8, %rdx
            cmpq %r8, %rdx
            jne .L4
        .L2:
            rep
            ret
```
Sum positive numbers in an array

```c
unsigned long
sum_positives(long arr[],
               unsigned int length)
{
    unsigned long sum = 0;
    unsigned int i;

    for (i = 0; i < length; i++) {
        if (arr[i] > 0)
            sum += arr[i];
    }

    return sum;
}
```

```assembly
sum_positives:
.LFB0:
    movl $0, %eax
    testl %esi, %esi
    je .L2
    subl $1, %esi
    leaq 8(%rsi,8), %r8
    movl $0, %edx
.L4:
    movq (%rdi,%rdx), %rcx
    leaq (%rax,%rcx), %rsi
    testq %rcx, %rcx
    cmovg %rsi, %rax
    addq $8, %rdx
    cmpq %r8, %rdx
    jne .L4
.L2:
    rep
    ret
```
Sum positive numbers in an array

unsigned long
sum_positives(long arr[],
               unsigned int length)
{
    unsigned long sum = 0;
    unsigned int i;
    for (i = 0; i < length; i++)
        if (arr[i] > 0)
            sum += arr[i];
    return sum;
}

Looks like sum will be in %eax/%rax
Sum positive numbers in an array

```c
unsigned long
sum_positives(long arr[],
    unsigned int length)
{
    unsigned long sum = 0;
    unsigned int i;

    for (i = 0; i < length; i++) {
        if (arr[i] > 0)    
            sum += arr[i];
    }

    return sum;
}
```

```assembly
sum_positives:
.LFB0:
movl $0, %eax

leaq 8(%rsi,8), %r8
movl $0, %edx

.L4:
movq (%rdi,%rdx), %rcx
leaq (%rax,%rcx), %rsi
testq %rcx, %rcx
cmovg %rsi, %rax
addq $8, %rdx
cmpq %r8, %rdx
jne .L4

.L2:
rep
ret
```
Sum positive numbers in an array

```c
unsigned long
sum_positives(long arr[],
               unsigned int length) {
    unsigned long sum = 0;
    unsigned int i;
    for (i = 0; i < length; i++) {
        if (arr[i] > 0)
            sum += arr[i];
    }
    return sum;
}
```

If length argument is zero, just leave
Sum positive numbers in an array

```c
unsigned long
sum_positives(long arr[],
               unsigned int length)
{
    unsigned long sum = 0;
    unsigned int i;

    for (i = 0; i < length; i++) {
        if (arr[i] > 0)
            sum += arr[i];
    }

    return sum;
}
```

We’re testing the value of `%rcx`, but why???
Sum positive numbers in an array

```c
unsigned long sum_positives(long arr[],
    unsigned int length)
{
    unsigned long sum = 0;
    unsigned int i;

    for (i = 0; i < length; i++) {
        if (arr[i] > 0)
            sum += arr[i];
    }

    return sum;
}
```

We're testing the value of `%rcx`, could that be our `arr[i] > 0` test? Yes!
Sum positive numbers in an array

```c
unsigned long
sum_positives(long arr[],
    unsigned int length)
{
    unsigned long sum = 0;
    unsigned int i;
    for (i = 0; i < length; i++) {
        if (arr[i] > 0)
            sum += arr[i];
    }
    return sum;
}
```

```assembly
sum_positives:
    .LFB0:
        movl $0, %eax
        testl %esi, %esi
        je .L2
        subl $1, %esi
        leaq 8(,%rsi,8), %r8
        movl $0, %edx
        .L4:
            movq (%rdi,%rdx), %rcx
            leaq (%rax,%rcx), %rsi
            testq %rcx, %rcx
            cmovg %rsi, %rax
            addq $8, %rdx
            cmpq %r8, %rdx
            jne .L4
        .L2:
            rep
            ret
```
Sum positive numbers in an array

```c
unsigned long
sum_positives(long arr[],
               unsigned int length)
{
    unsigned long sum = 0;
    unsigned int i;

    for (i = 0; i < length; i++) {
        if (arr[i] > 0)
            sum += arr[i];
    }

    return sum;
}
```

Hmm. We’re conditionally updating our result (%rax) with $rsi...
...we must have computed the sum there!

```
sum_positives:
   .LFB0:
   movl $0, %eax
   testl %esi, %esi
   je .L2
   subl $1, %esi
   leaq 8(,%rsi,8), %r8
   movl $0, %edx
   .L4:
   movq (%rdi,%rdx), %rcx
   leaq (%rax,%rcx), %rsi
   testq %rcx, %rcx
   cmovg %rsi, %rax
   addq $8, %rdx
   cmpq %r8, %rdx
   jne .L4
   .L2:
   rep
   ret
```
Sum positive numbers in an array

```c
unsigned long sum_positives(long arr[],
        unsigned int length)
{
    unsigned long sum = 0;
    unsigned int i;

    for (i = 0; i < length; i++) {
        if (arr[i] > 0)
            sum += arr[i];
    }

    return sum;
}
```

```assembly
sum_positives: 
.LFB0:
    movl $0, %eax
    testl %esi, %esi
    je .L2
    subl $1, %esi
    leaq 8(,%rsi,8), %r8
    movl $0, %edx
    .L4:
    movq (%rdi,%rdx), %rcx
    leaq (%rax,%rcx), %rsi
    testq %rcx, %rcx
    cmovq %rsi, %rax
    addq $8, %rdx
    cmpq %r8, %rdx
    jne .L4
    .L2:
    rep
    ret
```
Sum positive numbers in an array

```c
unsigned long sum_positives(long arr[], unsigned int length)
{
    unsigned long sum = 0;
    unsigned int i;
    for (i = 0; i < length; i++) {
        if (arr[i] > 0)
            sum += arr[i];
    }
    return sum;
}
```

This is looking a lot like our for loop index increment & test.
Sum positive numbers in an array

```c
unsigned long
sum_positives(long arr[],
               unsigned int length)
{
    unsigned long sum = 0;
    unsigned int i;

    for (i = 0; i < length; i++) {
        if (arr[i] > 0)
            sum += arr[i];
    }

    return sum;
}
```

But why are we adding 8 to the index variable???
Sum positive numbers in an array

```c
unsigned long
sum_positives(long arr[],
               unsigned int length)
{
    unsigned long sum = 0;
    unsigned int i;

    for (i = 0; i < length; i++) {
        if (arr[i] > 0)
            sum += arr[i];
    }

    return sum;
}
```

Address just past array!! (unclear why compiler bothers to subtract one and then add 8)
Sum positive numbers in an array

```c
unsigned long
sum_positives(long arr[],
               unsigned int length)
{
    unsigned long sum = 0;
    unsigned int i;

    for (i = 0; i < length; i++) {
        if (arr[i] > 0)
            sum += arr[i];
    }

    return sum;
}
```

But...what is %rdx?

```
sum_positives:
    .LFB0:
    movl    $0, %eax
    testl   %esi, %esi
    je      .L2
    subl    $1, %esi
    leaq    8(%rsi,8), %r8
    movl    $0, %edx
    .L4:
    movq    (%rdi,%rdx), %rcx
    leaq    (%rax,%rcx), %rsi
    testq   %rcx, %rcx
    cmovg   %rsi, %rax
    addq    $8, %rdx
    cmpq    %r8, %rdx
    jne     .L4
    .L2:
    rep
    ret
```
unsigned long
sum_positives(long arr[],
    unsigned int length)
{
    unsigned long sum = 0;
    unsigned int i;

    for (i = 0; i < length; i++) {
        if (arr[i] > 0)
            sum += arr[i];
    }
    return sum;
}

sum_positives:
.LFB0:
    movl $0, %eax
    testl %esi, %esi
    je .L2
    subl $1, %esi
    leaq 8(%rsi,8), %r8
    movl $0, %edx
    .L4:
        movq (%rdi,%rdx), %rcx
        leaq (%rax,%rcx), %rsi
        testq %rcx, %rcx
        cmovg %rsi, %rax
        addq $8, %rdx
        cmpq %r8, %rdx
        jne .L4
    .L2:
        rep
    ret

Offset to the next array element...we never actually compute variable i!
How C Language Data is Represented (Review)
## Sizes and alignments

<table>
<thead>
<tr>
<th>C Type</th>
<th>Width</th>
<th>Alignment</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>short</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>int</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>float</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>pointer, long, long long</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>double</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>long double</td>
<td>10</td>
<td>16</td>
</tr>
</tbody>
</table>

Why align to “natural sizes”?  

- Memory systems delivers aligned to cache blocks  
- CPU loads aligned words or blocks from cache  
- Data that spans boundaries needs two accesses!  
- *Naturally* aligned data never unnecessarily crosses a boundary  
- Intel/AMD alignment rules  
  - Pre AMD 64: unaligned data allowed but slower  
  - AMD 64: data *must* be aligned as shown on previous slides

http://www.cs.tufts.edu/comp/40/readings/amd64-abi.pdf#page=13
Alignment of C Language Data

- **Structures**
  - Wasted space used to “pad” between struct members if needed to maintain alignment
  - Therefore: when practical, put big members of larger primitive types ahead of smaller ones!
  - Structures sized for worst-case member
  - Arrays of structures will have properly aligned members

- **malloc return values always multiple of 16**
  - `structp = (struct s *)malloc(sizeof(struct s))` is always OK

- **Argument area on stack is 16 byte aligned**
  - `%rsp - 8` is multiple of 16 on entry to functions
  - Extra space “wasted” in caller frame as needed
  - Therefore: called function knows how to maintain alignment
Summary

- C code compiled to assembler
- Data moved to registers for manipulation
- Conditional and jump instructions for control flow
- Stack used for function calls
- Compilers play all sorts of tricks when compiling code
- Simple C data types map directly to *properly aligned* machine types in memory and registers