Today

• Function calls (wrap up)

Assignments

• HW7 out
• Exam 2 out
Register parameters

<table>
<thead>
<tr>
<th>operand size (bits)</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>64</td>
<td>%rdi</td>
<td>%rsi</td>
<td>%rdx</td>
<td>%rcx</td>
<td>%r8</td>
<td>%r9</td>
</tr>
<tr>
<td>32</td>
<td>%edi</td>
<td>%esi</td>
<td>%edx</td>
<td>%ecx</td>
<td>%r8d</td>
<td>%r9d</td>
</tr>
<tr>
<td>16</td>
<td>%di</td>
<td>%si</td>
<td>%dx</td>
<td>%cx</td>
<td>%r8w</td>
<td>%r9w</td>
</tr>
<tr>
<td>8</td>
<td>%dil</td>
<td>%sil</td>
<td>%dl</td>
<td>%cl</td>
<td>%r8b</td>
<td>%r9b</td>
</tr>
</tbody>
</table>

Callee-Saved registers

%rbx, %rbp, %r12–%r15

- caller can assume **not** overwritten during a function call
- thus, when P calls Q, Q must **preserve** these registers

Caller-Saved registers

All registers, except %rbx, %rbp, %r12–%r15, and %rsp

- thus, functions can modify them as needed
- if caller needs to maintain them, must save them, then restore after call
The “Base Pointer” Register Convention

The register %rbp is callee saved (called function maintains)

- by convention, used to hold the bottom of the stack frame for the function
- previous %rbp pushed onto stack at beginning of function body
- and previous %rbp returned by popping just before return
- helps with stack alignment
- helps deallocate the stack (can just move to %rsp)

fun1:

```
pushq %rbp
movq %rsp, %rbp  # %rbp is stack bottom (to deallocate stack)
...
callq fun2  # since %rbp pushed, stack aligned (less other pushes)
...
popq %rbp  # restore %rbp just prior to returning
retq
```

Note that using %rbp in this way is just a convention ...

- the gcc compiler follows this by default (optimizes compile time by default)
- but optimizes it out (when execution speed optimization via -O flag)
Stack Frames

Each function call has an associated "stack frame"

- a program's "runtime stack" consists of all the active stack frames
- each stack frame contains:
  - callee saved registers
  - caller saved registers (if it calls another function)
  - local variables stored on stack
  - arguments passed to functions it calls (as needed)
  - return address (if it called a function)
Mixing C and Assembly: Calling Functions

A simple sub3 function in assembly (sub3.s):

```
.global sub3
.text

sub3:           movq  %rdi, %rax
   subq  %rsi, %rax
   subq  %rdx, %rax
   retq
```

A simple C program that calls sub3 (callsub3.c):

```
#include <stdio.h>

long sub3(long x, long y, long z);

int main()
{
   printf("5-2-1: %ld\n", sub3(5, 2, 1));
   printf("5-1-2: %ld\n", sub3(5, 1, 2));
   printf("2-5-1: %ld\n", sub3(2, 5, 1));
}
```

To compile:

```
gcc callsub3.c sub3.s -o callsub3
```

Running the program:

```
./callsub3
5-2-1:  2
5-1-2:  2
2-5-1: -4
```
And again with stack params:

In sub8.s:

```assembly
.global sub8
.text

sub8:    movq  %rdi, %rax  # rax = a
         subq  %rsi, %rax  # rax -= b
         subq  %rdx, %rax  # rax -= c
         subq  %rcx, %rax  # rax -= d
         subq  %r8, %rax   # rax -= e
         subq  %r9, %rax   # rax -= f
         subq  8(%rsp), %rax # rax -= g
         subq  16(%rsp), %rax # rax -= h
         retq
```

In callsub8.c:

```c
#include <stdio.h>

long sub8(long a, long b, long c, long d, long e, long f, long g, long h);

int main()
{
    printf("%ld\n", sub8(29, 1, 2, 3, 4, 5, 6, 7));
}
```

Compiling and running:

```bash
gcc callsub8.c sub8.s -o callsub8
./callsub8
1
```
Recursive Functions ...

The “classic” factorial example:

// programmer-friendly form:

int factorial(int n)
{
    if (n <= 1)
        return 1;
    return n * factorial(n - 1);
}

// assembly-friendlier form:

int factorial(int n)
{
    if (n <= 1)
        return 1;
    int r = factorial(n - 1);
    r = r * n;
    return r;
}

As an assembly function (with C driver)

• In factorial.s:

    .global factorial
    .text
    factorial:  cmp   $1, %edi
                jnle  body  # !(n <= 1)
                mov   $1, %eax  # n <= 1
                ret
    body:      push  %rdi  # save n
                dec   %edi  # n - 1
                call  factorial
                pop   %rdi
                imul  %edi, %eax  # r = r * n
                ret

• In callfactorial.c:

    #include <stdio.h>
    int main()
    {
        for (int i = 0; i < 13; ++i)
            printf("factorial(%d) = %d\n", i, factorial(i));
    }
• compiling and running:

```bash
gcc -g callfactorial.c factorial.s -o callfactorial
./callfactorial
factorial(0) = 1
factorial(1) = 1
factorial(2) = 2
factorial(3) = 6
factorial(4) = 24
factorial(5) = 120
factorial(6) = 720
factorial(7) = 5040
factorial(8) = 40320
factorial(9) = 362880
factorial(10) = 3628800
factorial(11) = 39916800
factorial(12) = 479001600
```

Note that could also use push and pop for saving %edi

• instead of “allocating/deallocating” the stack (for alignment)

• Q: Which would be faster and why?
Functions with array arguments

Array variables are just pointers (so memory address to first array value is passed)

Driver program (callsumarray.c):

```c
#include <stdio.h>
long sum_array(long array[], long n);
int main()
{
  long xs[] = {1, 2, 3, 4, 5, 6};
  int n = 6;
  printf("%ld \n", sum_array(xs, n));
}
```

Assembly function (sumarray.s):

```
.global sum_array
.text
# rdi holds address of first array element
# rsi holds length of array
sum_array:
  xorq %rax, %rax # default result is zero
loop:
  cmpq $1, %rsi # loop rsi times
  jl done
  addq (%rdi), %rax # add value at array address
  addq $8, %rdi # go to next array value
  decq %rsi
  jmp loop
done: repz retq # repz is a nop here
```

Note on repz retq ...

- first AMD 64 architectures didn’t support branch prediction to a return
- the repz instruction here is just a nop instruction
- but gcc uses repz instead of nop for efficiency