Section 4: A Program Example

Simple C program example

- Consider the following simple C program, with two functions. It illustrates the X86 conventions for parameter passing, return value, and use of caller and callee save registers.
- First, main:

```
long sum(long count, long *array);
int main() {
    static long array[4] = {10, 12, 15, 19};
    long count= 4;                  /* number of array elements  */
    long result;
    result = sum(count, array);
    printf(“The sum of the array is %i\n”, result);
}
```

Function sum()

- Now, sum():

```
long sum(long count, long *array) {
    long result = 0;
     long i;
    for (i = 0; i < count; i++) {
        result = result + array[i];
    }
    return(result);
}
```

Now, the assembly language . . .

The next slide shows X86 assembler directives to set up space in memory for:

1. the static array,
2. output, and
3. the stack

Now, the assembly language . . .

```
.file “sumprog.s”
# Assembler directives to allocate storage for static array
.section
.rodata
printf_line:
.string “The sum of the array is %i\n”
.data
.align 8            # insure that we are starting on an 8-byte boundary
array:              # this is a LABEL
   .quad 10
   .quad 12
   .quad 15
   .quad 19
.globl main
         .type main, @function
```

Now, main()

```
.text
main:
    pushq %rbp                  # save caller's %rbp
    movq  %rsp, %rbp                      # copy %rsp to %rbp so our stack frame  is ready to use

    movq  $array, %rsi                    # set %rsi (2nd parameter) to point to start of array
    movq $4, %rdi                         # set %rdi (1st parameter) to count = 4
                                # (i.e. caller saved registers) since we aren't using %rsi or %rdi
                                # values or the value in any other caller saved registers,
                                # we don't have to push them
    call sum

    movq %rax, %rsi                       # Write return value to 2nd parameter
    movq $printf_line, %rdi               # Write string literal to 1st parameter

    movq $0, %rax
    call printf
    leave
    ret
.size main, .-main
```

Finally, sum()

```
.globl sum
        .type sum, @function
sum:
   pushq    %rbp                 #save caller's rbp
   movq %rsp, %rbp               #set function's frame pointer
                                 # register %rdi contains count (1st parameter)
                                 # register %rsi contains address to array (2nd parameter)
   movq $0, %rax                 # initialize sum to 0, by putting 0 in %rax,
                                 # it's where return value
                                 # needs to be when we return
loop:                            # loop to sum values in array
   decq   %rdi                   # decrement number of remaining elements by 1
   jl exit                                 # jump out of loop if no elements remaining
   addq (%rsi,%rdi,8), %rax # add element to sum
jmp     loop                     # jump to top of loop
exit:                            # sum already in register %rax so ready to return
   leave
   ret                           #return to caller's code at return address
.size sum, .-sum
```

Finally, sum() (modified)

```
sum:
    pushq    %rbp                          #save caller's rbp
    movq %rsp, %rbp                        #set function's frame pointer
                                           # register %rdi contains count (1st parameter)
                                           # register %rsi contains address to array (2nd parameter)
    movq $0, %rax                          # initialize sum to 0, by putting 0 in %rax,
                                           # it's where return value
                                           # needs to be when we return
    pushq %rax
    pushq %rdi                             #so what happens if I decide to add a printf call
    pushq %rsi                             # in the middle of this code?
    movq $printf_literal1, %rdi            # It changes a fundamental assumption about registers.
    movq $0, %rax                          # There is a better way to do this.
    call printf                            # the performance hit of this code if it was in the loop
    popq %rsi                              # would be bad – and there is a better way to do it.
    popq %rdi                              # what is the better way?
    popq %rax
loop:                                      # loop to sum values in array
    decq   %rdi                            # decrement number of remaining elements by 1
    jl exit                                # jump out of loop if no elements remaining
    addq (%rsi,%rdi,8), %rax               # add element to sum
jmp     loop                               # jump to top of loop
exit:                                      # sum already in register %rax so ready to return
    leave
    ret                                    #return to caller's code at return address
.size sum, .-sum
```