Computer Architecture (Practical Class)
Assembly: Stack and Functions - Parameters and Local Variables

Luís Nogueira

Departamento de Engenharia Informática
Instituto Superior de Engenharia do Porto

lmn@isep.ipp.pt

2021/2022

- Many (if not most) functions require some sort of input data
- In x86-64, most of the data passed to procedures take place via registers
- Up to six integral (i.e., integer and pointer) arguments can be passed via registers
- When a function has more than six integral arguments, the other ones are passed on the stack
- This method works equally as well for any Assembly program, even if it wasn't derived from a C program

- The registers are used in a **specified order**, with the name used for a register depending on the size of the data type being passed

Operand size (bits)	Argument number 1	2	3	4	5	6
64	`%rdi`	`%rsi`	`%rdx`	`%rcx`	`%r8`	`%r9`
32	`%edi`	`%esi`	`%edx`	`%ecx`	`%r8d`	`%r9d`
16	`%di`	`%si`	`%dx`	`%cx`	`%r8w`	`%r9w`
8	`%dil`	`%sil`	`%dl`	`%cl`	`%r8b`	`%r9b`

Passing parameters via registers: Example (1/2)

- Consider the following functions:

Function f1

```
long f1(long a, long b, long c){
  return a + b + f2(c);
}
```

Function f2

```
long f2(long c){
  return c + 1;
}
```

- How can you implement an equivalent code in Assembly?

Function f1

```c
long f1(long a, long b, long c){
  return a + b + f2(c);
}
```

Function f2

```c
long f2(long c){
  return c + 1;
}
```

Function f1

```
f1:
  # a in %rdi, b in %rsi, c in %rdx
  # a + b
  addq %rsi, %rdi
  # store %rdi on stack
  pushq %rdi
  # prepare function call f2(c)
  movq %rdx, %rdi
  call f2
  # restore %rdi
  popq %rdi
  # a + b + f2(c)
  addq %rdi, %rax
  ret
```

Function f2

```
f2:
  # c in %rdi
  # c + 1
  incq %rdi
  movq %rdi, %rax
  ret
```

Practice problem

- Consider the following functions:

Function f1

```c
void f1(int a, char b, int *res){
  *res = f2(a,b);
}
```

Function f2

```c
int f2(int a, char b){
  return a * b;
}
```

- Implement an equivalent code in Assembly

Practice problem

Function f1

```c
void f1(int a, char b, int *res){
  *res = f2(a,b);
}
```

Function f2

```c
int f2(int a, char b){
  return a * b;
}
```

Function f1

```
f1:
  # a in %edi, b in %sil
  # *res in %rdx
  # store %rdx on stack
  pushq %rdx
  # f2(a,b)
  call f2
  #restore %rdx
  popq %rdx
  movl %eax, (%rdx)
  ret
```

Function f2

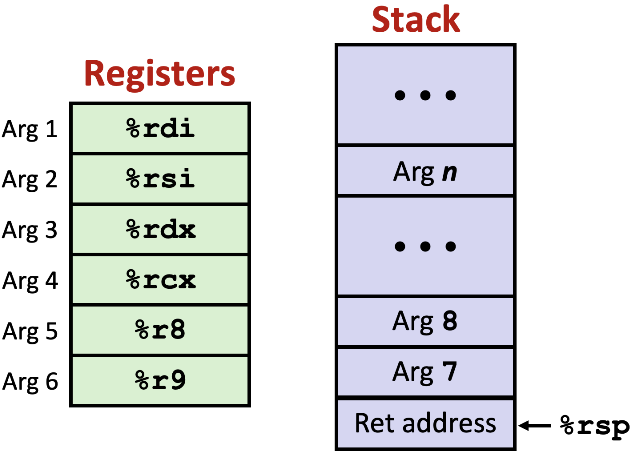
```
f2:
  # a in %edi, b in %sil
  # sign extend b
  movsbl %sil, %eax

  # a * b
  # result in %edx:%eax
  imull %edi
  ret
```

Passing parameters via stack (1/2)

- Assume that procedure P calls procedure Q with n integral arguments, such that $n > 6$
- P, the caller, must copy arguments 1 to 6 into the appropriate registers, and put arguments 7 through n onto the stack, with **argument 7 at the top of the stack**
- When passing parameters on the stack, all data sizes are rounded up to be multiples of eight
- With the arguments in place, procedure P can then execute a *call* instruction to transfer control to procedure Q
    - When the `call` instruction is executed, it places the return address from P onto the top of the stack as well, so Q knows where to return
- Q, the callee, can access its first six arguments via registers and the remaining ones from the stack

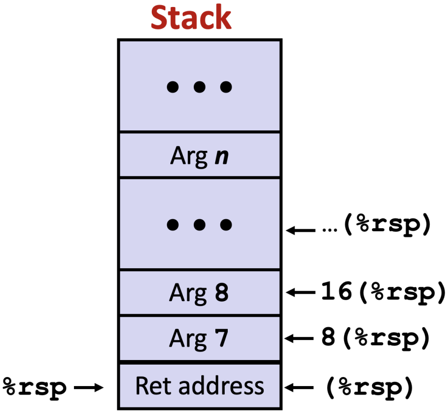
- All these details leave the registers and the stack in the state shown here



Accessing parameters on the stack(1/2)

- The stack pointer (RSP) points to the top of the stack, where the return address is located
  - All of the input parameters for the function are located "underneath" the return address on the stack
- Popping values off of the stack to retrieve the input parameters would cause a problem, as the return address might be lost in the process
- Each parameter can be indirectly accessed via the offset from the RSP register, without having to POP values off of the stack
  - `movq 8(%rsp), %rax` - copies (without popping!) the 7th parameter to the RAX register

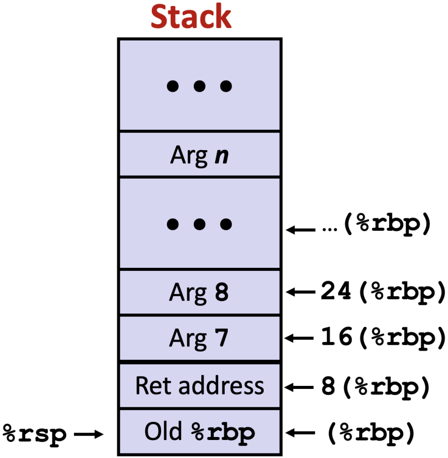
- Using indirect addressing with the RSP register to access the 7+ input parameters



Using RBP as the base pointer (1/2)

- There is an inconvenient problem with this approach! What happens if data is pushed to the stack throughout the callee's code?
- If this happens, it would change the location of the RSP stack pointer and throw off the indirect addressing values for accessing the parameters in the stack
- To avoid this problem, it is common practice to copy the RSP register value to the RBP register when entering the function
- This ensures that there is a register that always contains the correct pointer to the top of the stack when the function is called (a `push` does not changes RBP)
- To avoid corrupting the RBP register on the caller, the callee (before the RSP register value is copied) should also place the RBP register value on the stack (RBP is a callee-saved register) and restore it before returning

- All these details leave the stack in the state shown here



Prologue and Epilogue (1/3)

- This approach has created a standard set of instructions that are found in procedures that make a heavy use of the stack
- Stack space is allocated in **frames**
  - A section of stack used by one procedure call to store context while running, delimited by *%rbp* and *%rsp*
- Procedure code manages stack frames explicitly
  - **Prologue/Setup**: allocate space at start of procedure
  - **Epilogue/Cleanup**: deallocate space before return

- The base pointer *%rbp* being a stable "anchor" to the beginning of the stack frame throughout the execution of a function, is very convenient for manual Assembly coding and for debugging

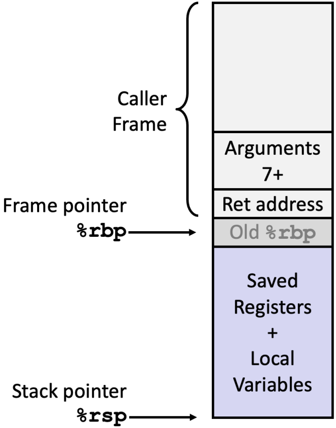
Function template with Prologue and Epilogue

```
function:
   # prologue
   pushq %rbp         # save the original value of RBP
   movq  %rsp, %rbp   # copy the current stack pointer to RBP

   # function body
   ...

   #epilogue
   movq  %rbp, %rsp   # retrieve the original RSP value
   popq  %rbp         # restore the original RBP value
   ret
```

- The prologue saves the caller's value for *%rbp* into the callee's stack frame and the current stack pointer in *%rbp*
- This adjusted value of *%rbp* is the callee's frame pointer
  - The callee will not change this value until just before it returns
  - The frame pointer provides a stable reference point for local variables and caller arguments
- The epilogue ensures that the caller's *%rsp* and *%rbp* are restored, which deallocates all callee's reserved temporary space



Cleaning out the stack (1/2)

- Before the function (with more than 6 arguments) is called, the caller places the 7+ required input values in the stack
- When the called function returns, those values are still on the stack (since the callee function accessed them without popping them off of the stack)
- The caller should remove the old input values from the stack to get the stack back to where it was before the function call
  - This is particularly important if other values are going to be popped out of the stack after the callee's return
  - Ensures that the *ret* instruction at the end of the caller finds the proper return value on top of the stack
- While you can use the POP instruction to do this, you can also just move the RSP stack pointer back to the original location before the function call

- Adding back the size of the data elements pushed onto the stack using the ADD instruction clears all the pushed data with a single instruction

Cleaning the Stack after Return

```
function:
  ...

  pushq %rax      # push 8th parameter onto stack
  pushq %rbx      # push 7th parameter onto stack
  call utilfunc
  addq $16, %rsp # get the stack back to where it was before the function call

  ...
  ret
```

Passing parameters via stack: Example (1/3)

- Consider the following functions:

Function f1

```
long f1(long a, long b, long c,
        long d, long e, long f){
  return a + f2(a*b,b,c,d,e,f,10,-20);
}
```

Function f2

```
long f2(long a, long b, long c,
        long d, long e, long f,
        long g, long h){
  /* do someting with parameters */
  ...
  return h/g;
}
```

- How can you implement an equivalent code in Assembly?

Passing Parameters

```
#long f1(a,b,c,d,e,f)
f1:
  # a in %rdi, b in %rsi, c in %rdx
  # d in %rcx, e in %r8, f in %r9

  pushq %rdi                  # save %rdi on stack

  movq %rdi, %rax             # a * b
  imulq %rsi
                              # prepare function call
  movq %rax, %rdi             # a*b in %rdi
  pushq $-20                  # 8th parameter
  pushq $10                   # 7th paramerer
  call f2                     # f2(a*b,b,c,d,e,f,10,-20);
  addq $16, %rsp              # clear 7th and 8th parameters

  popq %rdi                   # restore %rdi
  addq %rdi, %rax             # a + f2(...)
  ret
```

Accessing Parameters

```
#long f2(a,b,c,d,e,f,g,h)
f2:
  pushq %rbp                #prologue
  movq %rsp, %rbp

  # a in %rdi, b in %rsi, c in %rdx
  # d in %rcx, e in %r8, f in %r9
  # g in 16(%rbp), h in 24(%rbp)

  pushq %rbx               # save %rbx

  # do something with parameters
  ...
  movq 16(%rbp), %rbx      # get g from stack
  movq 24(%rbp), %rax      # get h from stack
  cqto                     # h/g
  idivq %rbx

  popq %rbx                # restore %rbx

  movq %rbp, %rsp          # epilogue
  popq %rbp
  ret
```

Local storage on the stack

- At times, local data must be stored in memory. Common cases of this include:
  - There are not enough registers to hold all of the local data
  - The address operator '&' is applied to a local variable, and hence we must be able to generate an address for it
  - Some of the local variables are arrays or structures and hence must be accessed by array or structure references
- When looking for an easy location for working storage for data elements within a function, the stack again comes to the rescue

Reserving space for local variables (1/3)

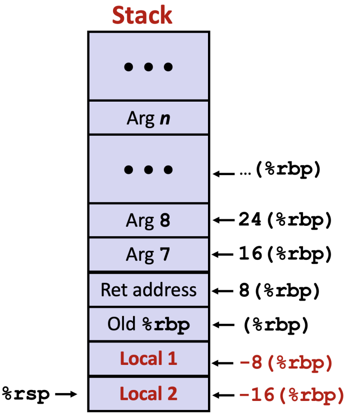
- The idea is to allocate storage for local variables when entering the function and release it before returning
- This space should be reserved after setting the RBP register to point to the top of the stack without affecting how the function's parameters are accessed
- Typically, a function allocates space on the stack frame by decrementing the stack pointer (%rsp)

Reserving Space for Local Variable

```
function:
   # prologue
   pushq %rbp
   movq  %rsp, %rbp
   subq  $16, %rsp    # reserve 16 bytes for local variables
                      # this space can be used, for e.g., as either
                      # four 32-bit values or two 64-bit values
   ...
```

Reserving space for local variables (2/3)

- By keeping %rbp pointing to the initial position throughout the execution of the function, we can refer local variables and parameters with fixed offsets relative to %rbp



Reserving space for local variables (3/3)

- Now, if any data is pushed onto the stack, it will be placed after the local variables
- Local variables are then preserved and they can still be accessed via RBP
- When the epilogue of the function is reached and the RSP register is set back to its original value, the local variables will be lost from the stack
- They will not be directly accessible using the RSP or RBP registers from the calling function (thus the term "local variables")

Using local variables: Example (1/3)

- Consider the following functions:

Function swap_add

```
long swap_add(long *xp, long *yp){
  long x = *xp;
  long y = *yp;
  *xp = y;
  *yp = x;
  return x + y;
}
```

Function f1

```
long f1(){
  long arg1 = 100;
  long arg2 = 200;
  long sum = swap_add(&arg1,&arg2);
  long diff = arg1 - arg2;
  return sum * diff;
}
```

- How can you implement an equivalent code in Assembly?

Using local variables: Example (2/3)

Function swap_add

```
#long swap_add(long *xp, long *yp)
swap_add:
   movq (%rdi), %rax     # get x
   movq (%rsi), %rcx     # get y
   movq %rcx, (%rdi)     # *xp = y
   movq %rax, (%rsi)     # *yp = x
   addq %rcx, %rax       # x + y
   ret
```

Using local variables: Example (3/3)

Function f1

```
#long f1()
f1:
  pushq %rbp                  # prologue
  movq  %rsp, %rbp
  subq $16, %rsp              # allocate 16 bytes for local variables

  movq $100, -8(%rbp)         # arg1 = 100;
  movq $200, -16(%rbp)        # arg2 = 200;

                              # prepare function call
  leaq -8(%rbp), %rdi         # compute &arg1 as first argument
  leaq -16(%rbp), %rsi        # compute &arg2 as second argument
  call swap_add               # call swap_add(&arg1,&arg2)

  movq -8(%rbp), %rdx         # get arg1
  subq -16(%rbp), %rdx        # diff = arg1 - arg2
  imulq %rdx                  # sum * diff

  movq  %rbp, %rsp            # epilogue (deallocates 16 bytes)
  popq  %rbp
  ret
```