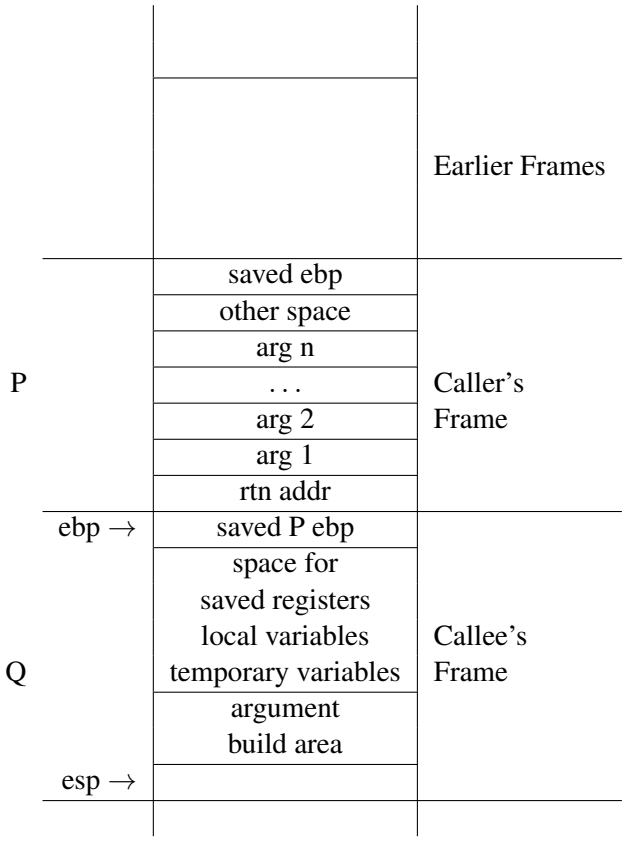
cs281: Computer Organization

Stack Conventions with Subroutines

1 Stack Frame Diagram

Always draw a stack frame while you are writing and/or debugging an assembly program.



1.1 Register Conventions

Since a caller might use registers to hold important values, it needs to know that those values will be there after a call to a subroutine. Similarly, the subroutine needs to know which registers can be used without worry and which ones need to be saved and restored.

The caller must always save:

- eax
- ecx
- edx

The callee must always save:

- ebx

- esi
- edi

Additionally the caller and callee must follow the guidelines about the stack and frame registers, esp and ebp.

1.2 Subroutine Requirements

The caller must always do the following:

- Ensure that the stack pointer points to the return address for the next instruction in the caller's routine. This is typically accomplished automatically with a `call` instruction.
- Save important values in free space on the stack. These are typically values in caller saved registers (eax, ecx and edx).
- Load the callee arguments into the correct location on the stack. They are typically right above the stack pointer (return address) in reverse order. See stack diagram on previous page.
- Call the subroutine (call instruction).

The callee must always do the following:

- Follow the conventions of setting up a stack frame. This means saving the old frame pointer (caller's ebp), setting up the new frame pointer, and clearing local space on the stack for subroutine use. Note these steps may be omitted in the case of a very simple subroutine.
- Access the caller's arguments in the correct location on the stack. These are "read only" values and should not be changed. Note that some of these arguments might be addresses (pointers) to other locations in memory. This is a pass-by-reference situation in which the data (but not the pointer) is intended to be altered by the subroutine. Otherwise the subroutine should only change memory within its own stack frame.
- Save callee saved registers on the stack.
- Do its computation.
- Place the return value in eax.
- Restore the stack and frame pointers (typically a leave instruction).
- Return (a ret instruction). struction).

Consider the following small c program. It uses a subroutine called `loop sum` to compute the sum of numbers $1 + 2 + ... + a$ for an input parameter a.

```
#include <stdio.h>

int loopsum (int);

int main ( void )
{
   int x;
   printf("Enter x: ");
   scanf("%d",&x);

   int s = loopsum(x);

   printf("sum is %d\n",s);
   return 0;
}

int loopsum (int a)
{
   int value = 0;
   while ( a > 0 )
   {
      value = value + a;
      a--;
   }
   return value;
}
```

Here is the partial assembler version with the ”main” routine complete, but the ”loopsum” subroutine incomplete. You are to complete the loopsum subroutine.

```
.text
.LC0:
.string "Enter x: "
.LC1:
.string "%d"
.LC2:
.string "sum is %d\n"
.text
.globl main
main:
pushl %ebp
```

```
movl %esp, %ebp
andl $-16, %esp
subl $32, %esp
movl $.LC0, (%esp)
call printf
leal 24(%esp), %eax
movl %eax, 4(%esp)
movl $.LC1, (%esp)
call __isoc99_scanf
movl 24(%esp), %eax
movl %eax, (%esp)
call loopsum
movl %eax, 28(%esp)
movl 28(%esp), %eax
movl %eax, 4(%esp)
movl $.LC2, (%esp)
call printf
movl $0, %eax
leave
ret

.globl loopsum
loopsum:
```

1. Obtain `loopsum.s` from our class webpage.
2. Complete the `loopsum` subroutine.
3. Set up a full stack frame for the subroutine. Clear 1 32-bit double word of space on the stack for the local variable `value`. Use this stack space to keep track of the local variable. It may be more efficient to store value in a register, but I want you to use local stack space to keep track of this value.
4. Use register `ecx` to load input argument a. Keep this register as the loop counter.
5. Write all the code for the subroutine, follow good loop conventions. Use your text as a resource for reference.
6. Compile with

   ```
   > gcc -m32 -O0 -g loopsum.s
   ```

   You can use ddd to debug your program if necessary.