Lab3
The Power operator

Ai Nguyen

October 17, 2015

1 power math

The power formula is given by this specification:

$$power : (\mathbb{R} \times \mathbb{N}) \rightarrow \mathbb{R}$$

$$power(r, p) = r^p = \prod_{i=1}^{p} r$$

This can be implemented by multiplying r times itself p times. One algorithm for calculating the power more efficiently than that can be visualized like this:

$$r^p = \overbrace{\underbrace{r \times r \cdots r}_{p \textbf{ div } 2} \times \underbrace{r \times r \cdots r}_{p \textbf{ div } 2} (\times r)}^{p}$$

Here the p multiplications are split into two groups of p **div** 2 multiplications, and, if p is odd, one more multiplication. This is a recursive definition where $r : \mathbb{R}$ and $p : \mathbb{N}$:

$$\begin{cases} 1, & \text{if } p = 0 \\ r \times r^{p-1}, & \text{if } p \text{ is odd} \\ \text{sqr}(r^{p \textbf{ div } 2}), & \text{otherwise} \end{cases}$$

$$\text{where } sqr(x) = x \times x$$

2 power sml

SML

```
fun square x: real = x * x;
square 2.0;
fun odd x = x mod 2 = 1;
odd 3;
odd 4;
fun power r p = if p = 0 then 1.0
                else if odd p then r * power r (p-1)
                else square (power r (p div 2));
power 5.0 3; (* 125 *)
power 2.0 8; (* 256 *)
```

```
debian@debian:~/labs/lab3$ poly < lab3.sml
Poly/ML 5.5.2 Release
val square = fn: real -> real
val it = 4.0: real
val odd = fn: int -> bool
val it = true: bool
val it = false: bool
val power = fn: real -> int -> real
val it = 125.0: real
val it = 256.0: real
```

3 power c without using function composition

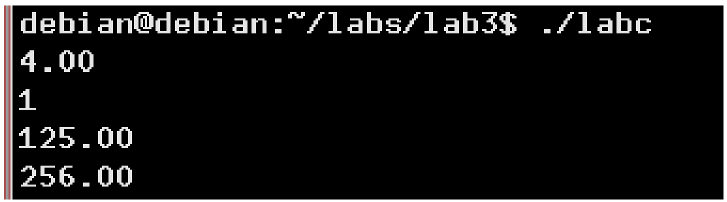
It will help us transition to ASM if we rewrite the C code using only one calculation per line instead of combining them into complex expressions. In fact even the returns are done differently–they are all expected to be in EAX(for integers) and ST(0) for floats.

C source code written to file lab3.c

```
#include <stdio.h>
#include <stdbool.h>
float ST0;
int EAX;
float square(float x) {ST0 = x * x;}
int odd(int x) { EAX = x % 2;}
// return value 0 or 1 in EAX
void power(float r, int p)
{
  if(p!=0) goto first_else;
  ST0 = 1.0;
  goto end_if;
first_else:
  odd(p);
  if(EAX != 1) goto second_else;
  int t = p-1;
  power(r,t);
  ST0 = r * ST0; // ST0 *=r
  goto end_if;
```

C++ source code appended to file lab3.c

```
second_else:
  t = p/2;
  power(r,t);
  square(ST0);
end_if:
  return;
} //result ST0
int main()
{
  power(5.0,3);
  printf("%.2f\n", ST0);
  power(2.0,8);
  printf("%.2f\n", ST0);
}
```



```
debian@debian:~/labs/lab3$ ./labc
4.00
1
125.00
256.00
```

4 power asm

4.1 square function

asm source code written to file lab3.s

```
.text
square:
  push %ebp
  mov %esp,%ebp
  fmuls 8(%ebp)                         # multiply st(0) with the parameter
                                    # st(0) should be equal to 8(%ebp)
  mov %ebp,%esp
  pop %ebp
  ret
```

Here are using 4 offset from esp becase we didn't put ebp on the stack, so the parameter is only 4 bytes above esp.

4.2 odd function

asm source code appended to file lab3.s

```
odd:
  mov 4(%esp), %eax
  and $1, %eax                      # compare 1 and eax by bit-wise arithmetic
                                    # return 1 to %eax if the number is odd
                                    # or return 0 to %eax if the number is even
  ret
```

4.3 power function

The power function will return the result in $st(0)$, the first parameter is r, a real number, and second is p, an integer to calculate r^p
First, compare p stored at 8(%ebp) with 0, if p equals to 0, push 1 onto floating point stack, $st(0) = 1$ and then return. If p is not equal to 0, jump to first_if.
I found out that the code we did together in class did not work for the power that greater than 4. We will have some weird output -nan. So I fix square and power function a little bit by deleting flds line. The idea is that fld1 already push 1 to floating-point stack, the function basically just modify $st(0)$ by multiplying by r each time.
I think we do not need to use flds to push r onto floating-point stack in the beginning of power function, because it will cause stack overflow if we call power function too many time and we did not pop anything out.

asm source code appended to file lab3.s

```
power:
  push %ebp
  mov %esp, %ebp
  sub $4, %esp                 # room for local varible t

  cmp $0, 8(%ebp)              # p == 0
  jne first_else          # if p !=0 go to first else

  # if p = 0
  fld1                         # push 1 to FP-stack, st(0) = 1
  jmp end_if                   # exit function
```

The first_if is to do calculation if p is odd. If p is odd, decrement p and call power function for r^{p-1}

asm source code appended to file lab3.s

```
first_else:
  push 8(%ebp)                # push p to stack for odd function's parameter
  call odd                    # store 1 to %eax if p is odd
                              # or store 0 to %eax if p is even
  add $4, %esp
  cmp $1, %eax                # eax == 1
  jne second_else             # if p is even, go to second_else

  # if p is odd
  mov 8(%ebp), %eax           # eax = p
  dec %eax                    # eax = p - 1
  mov %eax, -4(%ebp)          # t = eax = p - 1

  # call power function with two parameters r and t = p-1
  push 12(%ebp)                         # r
  push -4(%ebp)                         # t
  call power
  add $8, %esp
  fmul 12(%ebp)                         # st(0) *= r
  jmp end_if
```

The second_if is to do calculation if p is even. If p is even, divide p by 2 and call power function for $r^{p/2}$. We will use bit-shift to divide p.

asm source code appended to file lab3.s

```
second_else:
  mov 8(%ebp), %eax              # eax = p
  shr %eax                       # eax = eax/2, bit-shift right will divide by 2
  mov %eax, -4(%ebp)             # t = p/2

  #call power function with two parameters r and t = p/2
  push 12(%ebp)                        # r
  push -4(%ebp)                        # t
  call power
  add $8, %esp

  #push st(0) to stack to prepare parameter for calling square function
  fsts (%esp)
  call square
  add $4, %esp

end_if:
  mov %ebp, %esp
  pop %ebp
  ret
```

This is another version of power that I wrote without using %ebp, but making use of varible r, p and ST0. It works fine, but the problem is that this power function can just be used for only one set of r and p.

asm source code appended to file lab3.s

```
power_more:
  cmp $0, 4(%esp)
  jne first_else_more
  fld1
  jmp end_if_more
first_else_more:
  push 4(%esp)
  call odd
  add $4, %esp
  cmp $1, %eax
  #jump to second_else if p is even
  jne second_else_more

  sub $1, p
  mov p, %eax
  push r
  push p
  call power_more
  add $8, %esp

  fmuls r
  jmp end_if_more
```

asm source code appended to file lab3.s

```
second_else_more:
  mov p, %eax
  shr %eax
  mov %eax, p

  push r
  push p
  call power_more
  add $8, %esp

  fst ST0
  push ST0
  call square
  add $4, %esp
  jmp end_if_more
end_if_more:
  ret
```

4.4 Start program

asm source code appended to file lab3.s

```
.data
r: .float 5.0
p:  .int 3
r2: .float 2.0
p2: .int 8
fmt: .string "%f\n"
fmt2: .string "%i\n"
ST0 : .float 0.0
.text
.globl _start
_start:
```

Testing square function with $r = 5$, expect result will be 25

asm source code appended to file lab3.s

```
flds r
push r
call square
add $4,%esp
#expect result in %st(0)

add $-8,%esp
fstpl (%esp) #push 64-bits st(0) onto the stack and pop st(0)
push $fmt
call printf
add $12, %esp
```

Testing odd function with p = 3 and p = 8, expect result will be 1 and 0

asm source code appended to file lab3.s

```
push p
call odd
add $4,%esp
#expect result in %eax

push %eax
push $fmt2
call printf
add $8, %esp
```

asm source code appended to file lab3.s

```
push p2
call odd
add $4,%esp
#expect result in %eax

push %eax
push $fmt2
call printf
add $8, %esp
```

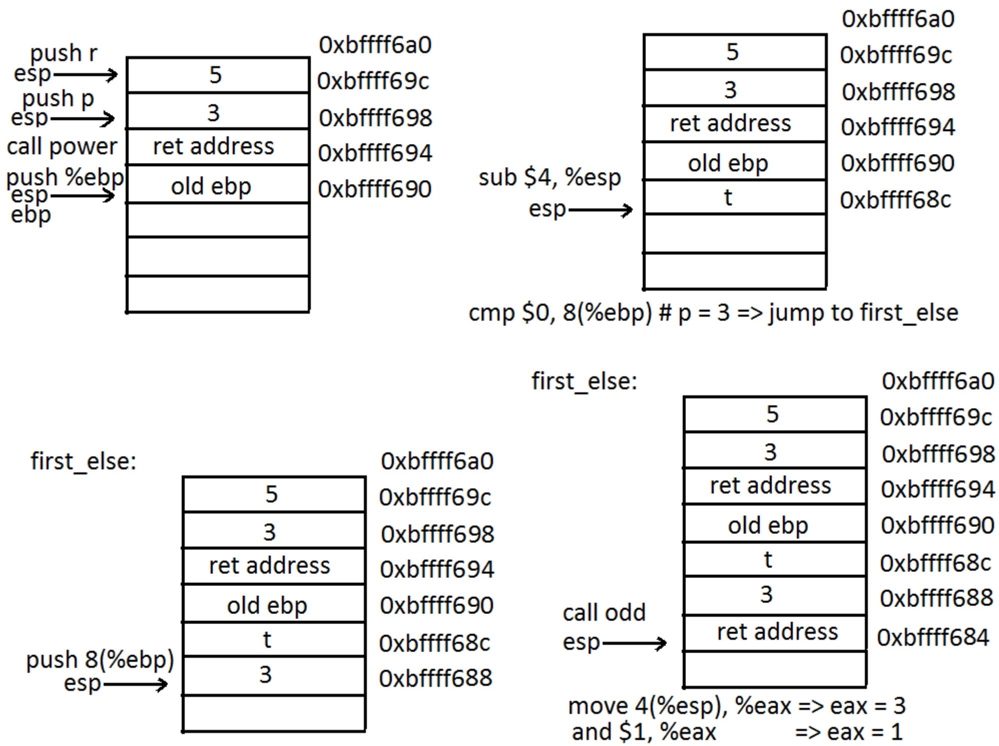
Testing power function with $r = 5$, $p = 3$ and $r = 2$, $p = 8$, expect result will be 125 and 256

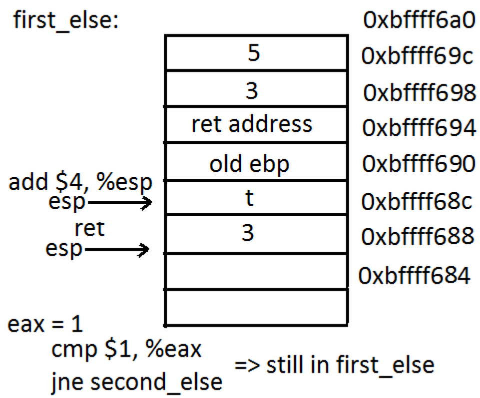
asm source code appended to file lab3.s

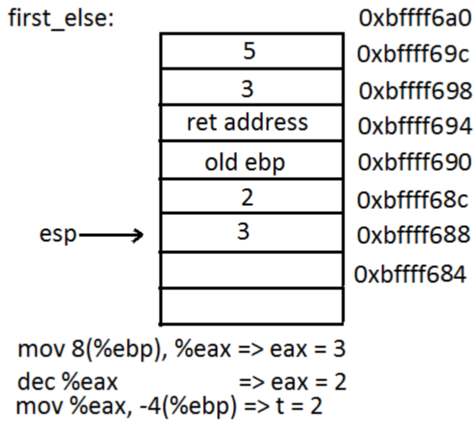
```
push r
push p
call power
add $8,%esp
#expect result in %eax

add $-8,%esp
fstpl (%esp) #push 64-bits st(0) onto the stack and pop st(0)
push $fmt
call printf
add $12, %esp

push r2
push p2
call power
add $8,%esp
#expect result in %eax

add $-8,%esp
fstpl (%esp) #push 64-bits st(0) onto the stack and pop st(0)
push $fmt
call printf
add $12, %esp

mov $1, %eax
mov $0, %ebx
int $0x80
```

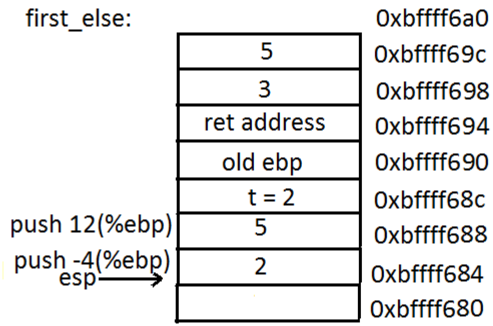
```
debian@debian:~/labs/lab3$ ./lab3asm
25.000000
1
0
125.000000
256.000000
```

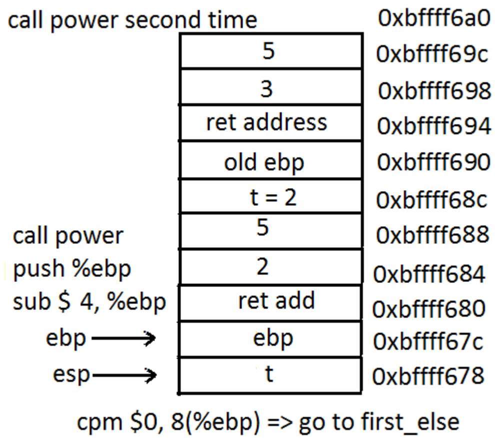
5 Stack Frame

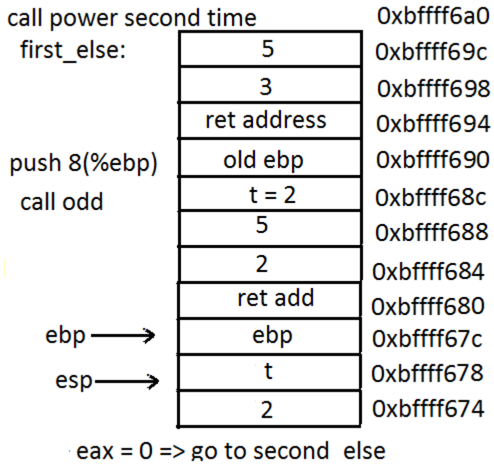


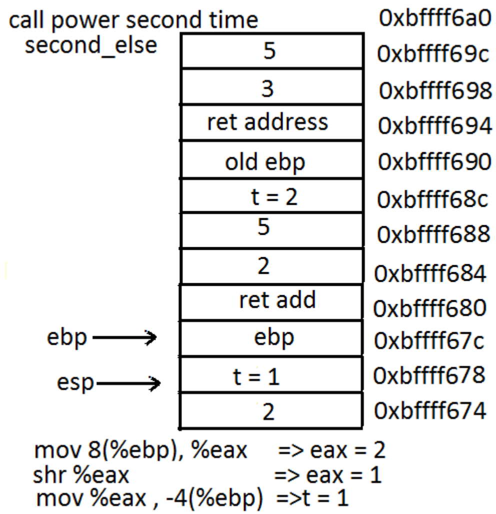

first_else:
0xbffff6a0
5
0xbffff69c
3
0xbffff698
ret address
0xbffff694
old ebp
0xbffff690
add $4, %esp
esp
t
0xbffff68c
ret
esp
3
0xbffff688
0xbffff684
eax = 1
cmp $1, %eax
jne second_else
=> still in first_else

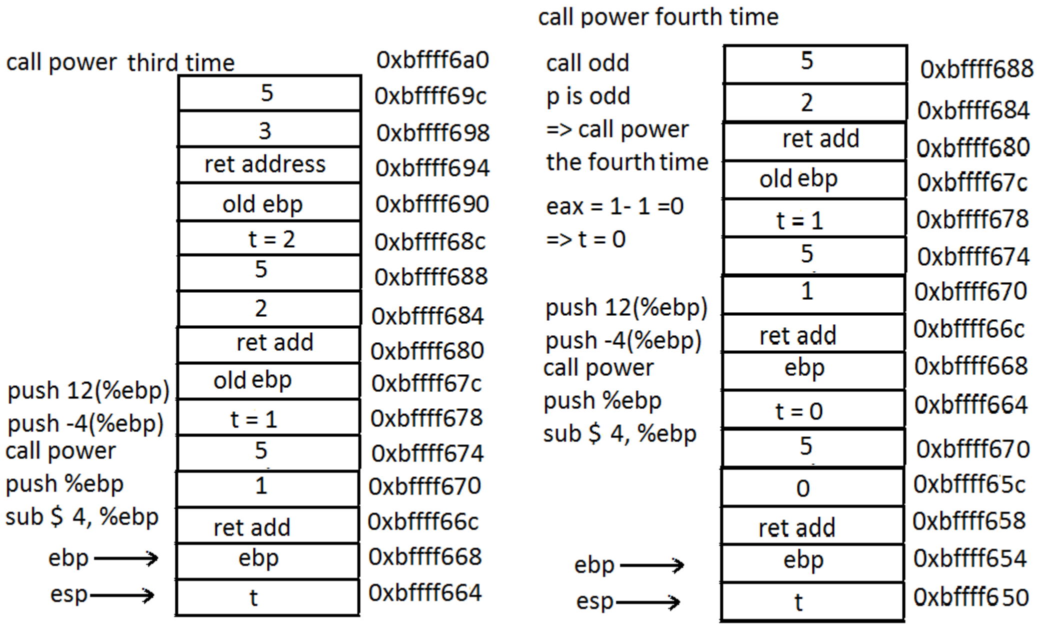

first_else:
0xbffff6a0
5
0xbffff69c
3
0xbffff698
ret address
0xbffff694
old ebp
0xbffff690
2
0xbffff68c
esp
3
0xbffff688
0xbffff684
mov 8(%ebp), %eax => eax = 3
dec %eax => eax = 2
mov %eax, -4(%ebp) => t = 2

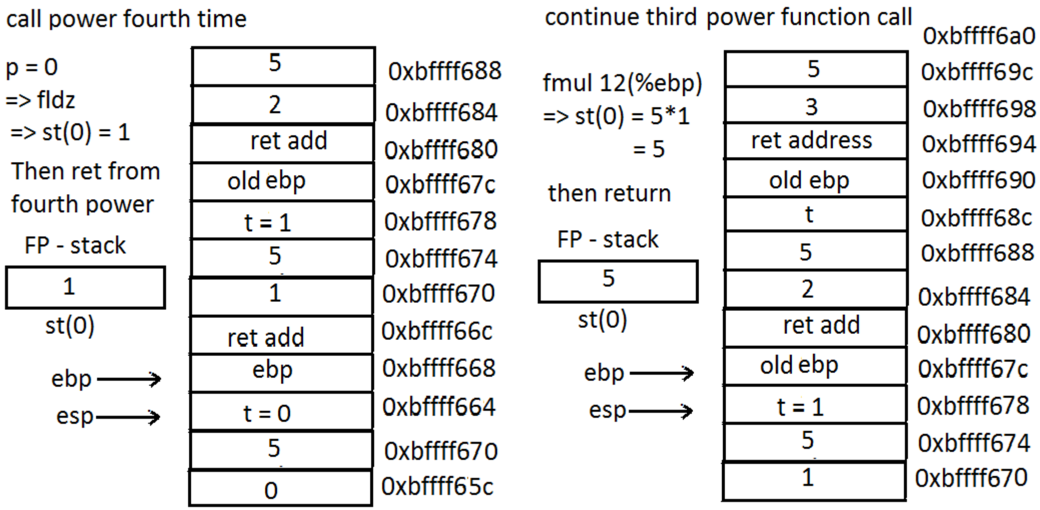

first_else:
0xbffff6a0
5
0xbffff69c
3
0xbffff698
ret address
0xbffff694
old ebp
0xbffff690
t = 2
0xbffff68c
push 12(%ebp)
5
0xbffff688
push -4(%ebp)
esp
2
0xbffff684
0xbffff680


call power second time
0xbffff6a0
5
0xbffff69c
3
0xbffff698
ret address
0xbffff694
old ebp
0xbffff690
t = 2
0xbffff68c
call power
5
0xbffff688
push %ebp
2
0xbffff684
sub $ 4, %ebp
ret add
0xbffff680
ebp
ebp
0xbffff67c
esp
t
0xbffff678
cpm $0, 8(%ebp) => go to first_else

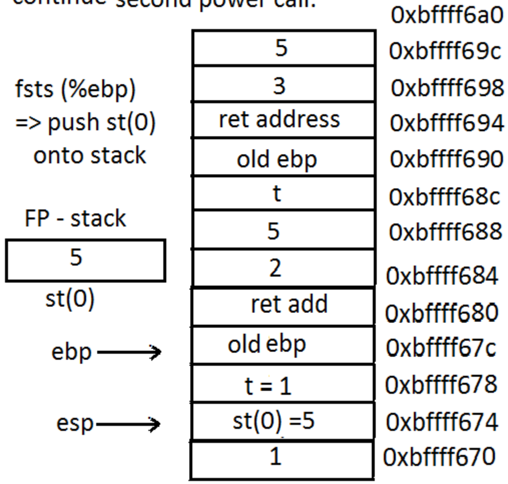

call power second time
first_else:
push 8(%ebp)
call odd
ebp
esp
5
3
ret address
old ebp
t = 2
5
2
ret add
ebp
t
2
0xbffff6a0
0xbffff69c
0xbffff698
0xbffff694
0xbffff690
0xbffff68c
0xbffff688
0xbffff684
0xbffff680
0xbffff67c
0xbffff678
0xbffff674
eax = 0 => go to second else


call power second time
second_else
ebp
esp
5
3
ret address
old ebp
t = 2
5
2
ret add
ebp
t = 1
2
0xbffff6a0
0xbffff69c
0xbffff698
0xbffff694
0xbffff690
0xbffff68c
0xbffff688
0xbffff684
0xbffff680
0xbffff67c
0xbffff678
0xbffff674
mov 8(%ebp), %eax => eax = 2
shr %eax => eax = 1
mov %eax , -4(%ebp) =>t = 1

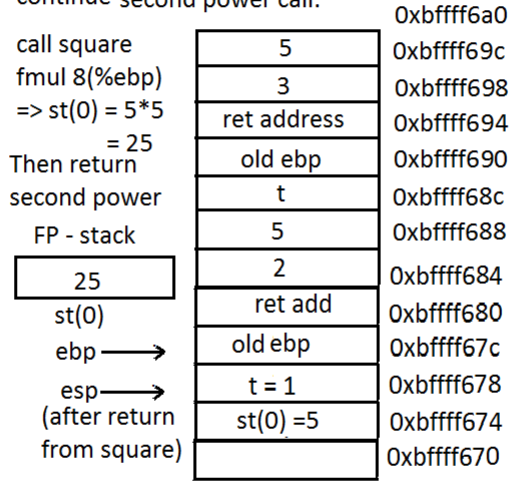

call power third time
5
3
ret address
old ebp
t = 2
5
2
ret add
old ebp
t = 1
5
1
ret add
ebp
t
0xbffff6a0
0xbffff69c
0xbffff698
0xbffff694
0xbffff690
0xbffff68c
0xbffff688
0xbffff684
0xbffff680
0xbffff67c
0xbffff678
0xbffff674
0xbffff670
0xbffff66c
0xbffff668
0xbffff664
push 12(%ebp)
push -4(%ebp)
call power
push %ebp
sub $ 4, %ebp
ebp
esp
call power fourth time
call odd
p is odd
=> call power
the fourth time
eax = 1- 1 =0
=> t = 0
push 12(%ebp)
push -4(%ebp)
call power
push %ebp
sub $ 4, %ebp
ebp
esp
5
2
ret add
old ebp
t = 1
5
1
ret add
ebp
t = 0
5
0
ret add
ebp
t
0xbffff688
0xbffff684
0xbffff680
0xbffff67c
0xbffff678
0xbffff674
0xbffff670
0xbffff66c
0xbffff668
0xbffff664
0xbffff670
0xbffff65c
0xbffff658
0xbffff654
0xbffff650


call power fourth time
p = 0
=> fldz
=> st(0) = 1
Then ret from
fourth power
FP - stack
1
st(0)
ebp
esp
5
2
ret add
old ebp
t = 1
5
1
ret add
ebp
t = 0
5
0
0xbffff688
0xbffff684
0xbffff680
0xbffff67c
0xbffff678
0xbffff674
0xbffff670
0xbffff66c
0xbffff668
0xbffff664
0xbffff670
0xbffff65c
continue third power function call
fmul 12(%ebp)
=> st(0) = 5*1
= 5
then return
FP - stack
5
st(0)
ebp
esp
5
3
ret address
old ebp
t
5
2
ret add
old ebp
t = 1
5
1
0xbffff6a0
0xbffff69c
0xbffff698
0xbffff694
0xbffff690
0xbffff68c
0xbffff688
0xbffff684
0xbffff680
0xbffff67c
0xbffff678
0xbffff674
0xbffff670

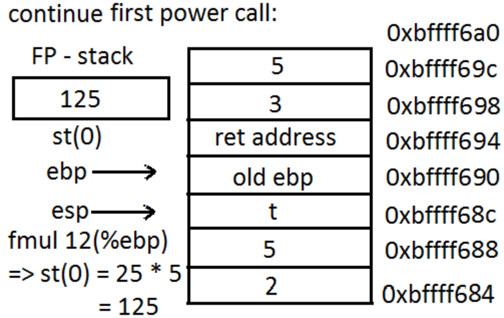
continue second power call:



continue second power call:



continue first power call:



Then we finish calling power function for 5^3