CS 33

Machine Programming (4)

Reading Condition Codes

- **SetX instructions**
  - **set single byte (to 1 or 0) based on combinations of condition codes**

SetX	Condition	Description
`sete`	`ZF`	Equal / Zero
`setne`	`~ZF`	Not Equal / Not Zero
`sets`	`SF`	Negative
`setns`	`~SF`	Nonnegative
`setg`	`~(SF^OF)&~ZF`	Greater (Signed)
`setge`	`~(SF^OF)`	Greater or Equal (Signed)
`setl`	`(SF^OF)`	Less (Signed)
`setle`	`(SF^OF)|ZF`	Less or Equal (Signed)
`seta`	`~CF&~ZF`	Above (unsigned)
`setb`	`CF`	Below (unsigned)

Jumping

- **jX instructions**
  - **Jump to different part of program depending on condition codes**

jX	Condition	Description
`jmp`	`1`	Unconditional
`je`	`ZF`	Equal / Zero
`jne`	`~ZF`	Not Equal / Not Zero
`js`	`SF`	Negative
`jns`	`~SF`	Nonnegative
`jg`	`~(SF^OF)&~ZF`	Greater (Signed)
`jge`	`~(SF^OF)`	Greater or Equal (Signed)
`jl`	`(SF^OF)`	Less (Signed)
`jle`	`(SF^OF)|ZF`	Less or Equal (Signed)
`ja`	`~CF&~ZF`	Above (unsigned)
`jb`	`CF`	Below (unsigned)

Conditional-Branch Example

```
int absdiff(int x, int y)
{
    int result;
    if (x > y) {
      result = x-y;
    } else {
      result = y-x;
    }
    return result;
}
```

```
absdiff:
    movl    %esi, %eax
    cmpl    %esi, %edi
    jle     .L6
    subl    %eax, %edi
    movl    %edi, %eax
    jmp     .L7
.L6:
    subl %edi, %eax
.L7:
    ret
```

Body1

Body2a

Body2b

x in %edi

y in %esi

Conditional-Branch Example (Cont.)

```
int goto_ad(int x, int y)
{
  int result;
  if (x <= y) goto Else;
  result = x-y;
  goto Exit;
Else:
  result = y-x;
Exit:
  return result;
}
```

- C allows “goto” as means of transferring control
  – closer to machine-level programming style
- Generally considered bad coding style

```
absdiff:
    movl    %esi, %eax
    cmpl    %esi, %edi
    jle     .L6
    subl    %eax, %edi
    movl    %edi, %eax
    jmp .L7
.L6:
    subl %edi, %eax
.L7:
    ret
```

Body1

Body2a

Body2b

General Conditional-Expression Translation

C Code

```
val = Test ? Then_Expr : Else_Expr;
```

```
val = x>y ? x-y : y-x;
```

- <u>Test</u> is expression returning integer
  - == 0 interpreted as false
  - ≠ 0 interpreted as true
- Create separate code regions for <u>then</u> and <u>else</u> expressions
- Execute appropriate one

Goto Version

```
  nt = !Test;
  if (nt) goto Else;
  val = Then_Expr;
  goto Done;
Else:
  val = Else_Expr;
Done:
  . . .
```

"Do-While" Loop Example

C Code

```
int pcount_do(unsigned x)
{
  int result = 0;
  do {
    result += x & 0x1;
    x >>= 1;
  } while (x);
  return result;
}
```

Goto Version

```
int pcount_do(unsigned x)
{
  int result = 0;
loop:
  result += x & 0x1;
  x >>= 1;
  if (x)
    goto loop;
  return result;
}
```

- **Count number of 1's in argument x ("popcount")**
- **Use conditional branch either to continue looping or to exit loop**

"Do-While" Loop Compilation

Goto Version

```
int pcount_do(unsigned x)  {
  int result = 0;
loop:
  result += x & 0x1;
  x >>= 1;
  if (x)
    goto loop;
  return result;
}
```

Registers:

%edi	x
%eax	result

```
  movl   $0, %eax      #   result = 0
.L2:     # loop:
  movl   %edi, %ecx
  andl   $1, %ecx      #   t = x & 1
  addl   %ecx, %eax    #   result += t
  shrl   %edi          #   x >>= 1
  jne    .L2           #   if !0, goto loop
```

General “Do-While” Translation

C Code

```
do
  Body
  while (Test);
```

Goto Version

```
loop:
  Body
  if (Test)
    goto loop
```

- **Body:**

```
{
   Statement1;
   Statement2;
     …
   Statementn;
}
```

- **Test returns integer**
  - **= 0 interpreted as false**
  - **≠ 0 interpreted as true**

"While" Loop Example

C Code

```
int pcount_while(unsigned x) {
  int result = 0;
  while (x) {
    result += x & 0x1;
    x >>= 1;
  }
  return result;
}
```

Goto Version

```
int pcount_do(unsigned x)  {
  int result = 0;
  if (!x) goto done;
loop:
  result += x & 0x1;
  x >>= 1;
  if (x)
    goto loop;
done:
  return result;
}
```

- **Is this code equivalent to the do-while version?**
  - **must jump out of loop if test fails**

General “While” Translation

While version

```
while (Test)
  Body
```

Do-While Version

```
  if (!Test)
    goto done;
  do
    Body
    while(Test);
done:
```

Goto Version

```
  if (!Test)
    goto done;
loop:
  Body
  if (Test)
    goto loop;
done:
```

"For" Loop Example

C Code

```
#define WSIZE 8*sizeof(int)
int pcount_for(unsigned x) {
  int i;
  int result = 0;
  for (i = 0; i < WSIZE; i++) {
    unsigned mask = 1 << i;
    result += (x & mask) != 0;
  }
  return result;
}
```

- Is this code equivalent to other versions?

"For" Loop Form

General Form

```
for (Init; Test; Update )

    Body
```

```
for (i = 0; i < WSIZE; i++) {
    unsigned mask = 1 << i;
    result += (x & mask) != 0;
}
```

Init

```
i = 0
```

Test

```
i < WSIZE
```

Update

```
i++
```

Body

```
{
  unsigned mask = 1 << i;
  result += (x & mask) != 0;
}
```

"For" Loop → While Loop

For Version

```
for (Init; Test; Update )
      Body
```

While Version

```
Init;
while (Test ) {
      Body
      Update;
}
```

“For” Loop → … → Goto

For Version

```
for (Init; Test; Update )
    Body
```

While Version

```
Init;
while (Test ) {
    Body
    Update;
}
```

```
  Init;
  if (!Test)
    goto done;
  do
    Body
    Update
  while(Test);
done:
```

```
  Init;
  if (!Test)
    goto done;
loop:
  Body
  Update
  if (Test)
    goto loop;
done:
```

"For" Loop Conversion Example

C Code

```
#define WSIZE 8*sizeof(int)
int pcount_for(unsigned x) {
  int i;
  int result = 0;
  for (i = 0; i < WSIZE; i++) {
    unsigned mask = 1 << i;
    result += (x & mask) != 0;
  }
  return result;
}
```

Goto Version

```
int pcount_for_gt(unsigned x) {
  int i;
  int result = 0;
  i = 0;                          Init
  if (!(i < WSIZE))               ! Test
    goto done;
 loop:
  {                                  Body
    unsigned mask = 1 << i;
    result += (x & mask) != 0;
  }
  i++;  Update
  if (i < WSIZE)  Test
    goto loop;
 done:
  return result;
}
```

Initial test can be optimized away

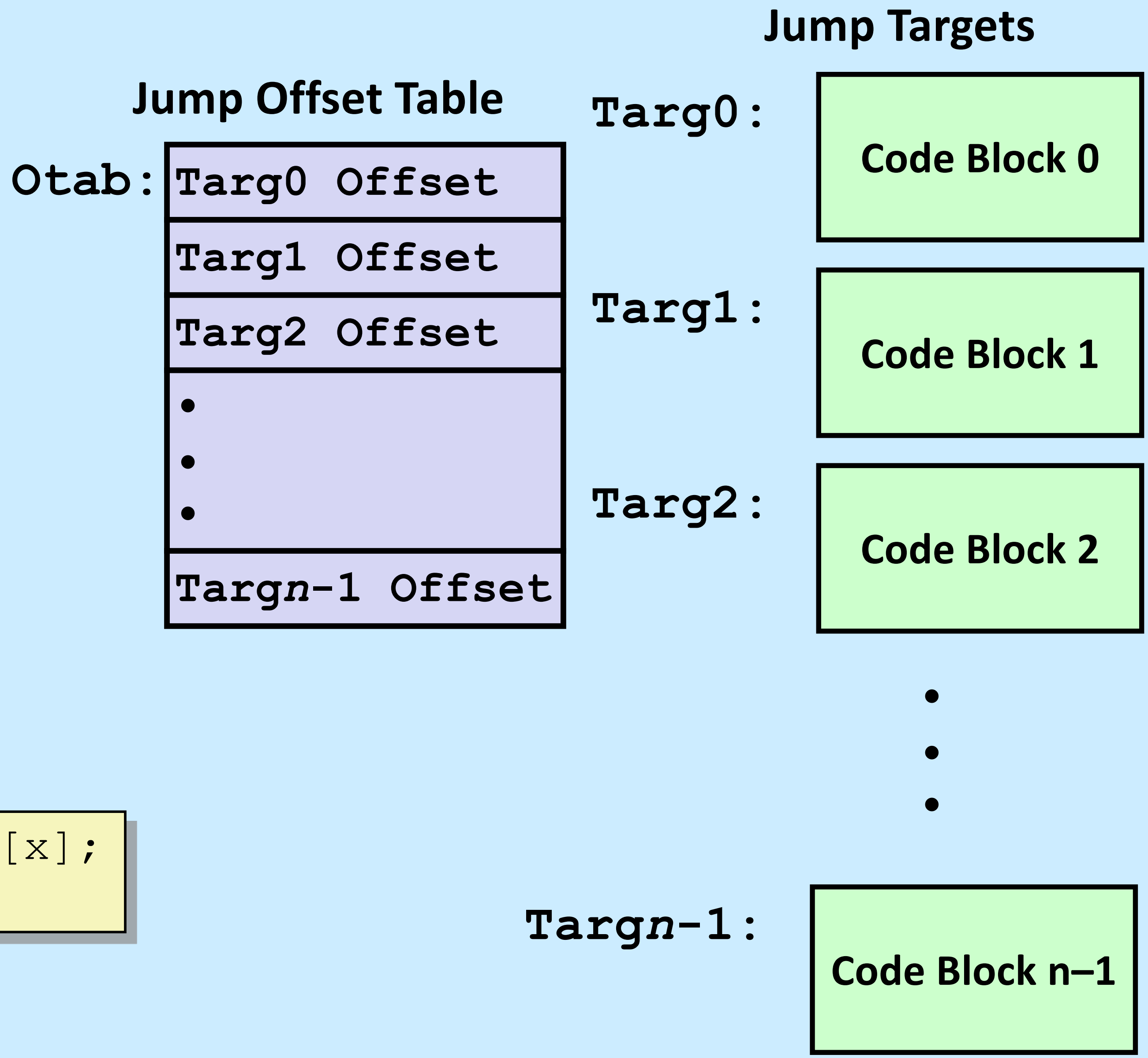
Switch-Statement Example

```
long switch_eg (long m, long d) {
    if (d < 1) return 0;
    switch(m) {
    case 1: case 3: case 5:
    case 7: case 8: case 10:
    case 12:
        if (d > 31) return 0;
        else return 1;
    case 2:
        if (d > 28) return 0;
        else return 1;
    case 4: case 6: case 9:
    case 11:
        if (d > 30) return 0;
        else return 1;
    default:
        return 0;
    }
    return 0;
}
```

Offset Structure

Switch Form

```
switch(x) {
  case val_0:
    Block 0
  case val_1:
    Block 1
    • • •
  case val_n-1:
    Block n–1
}
```



Approximate Translation

```
target = Otab + OTab[x];
goto *target;
```

Assembler Code (1)

```
switch_eg:
        movl    $0, %eax
        testq   %rsi, %rsi
        jle     .L1
        cmpq    $12, %rdi
        ja      .L8
        leaq    .L4(%rip), %rdx
        movslq  (%rdx,%rdi,4), %rax
        addq    %rdx, %rax
        jmp     *%rax
```

```
        .section    .rodata
        .align 4
.L4:
        .long   .L8-.L4
        .long   .L3-.L4
        .long   .L6-.L4
        .long   .L3-.L4
        .long   .L5-.L4
        .long   .L3-.L4
        .long   .L5-.L4
        .long   .L3-.L4
        .long   .L3-.L4
        .long   .L5-.L4
        .long   .L3-.L4
        .long   .L5-.L4
        .long   .L3-.L4
        .text
```

Assembler Code (2)

```
.L3:
        cmpq    $31, %rsi
        setle   %al
        movzbl  %al, %eax
        ret
.L6:
        cmpq    $28, %rsi
        setle   %al
        movzbl  %al, %eax
        ret
.L5:
        cmpq    $30, %rsi
        setle   %al
        movzbl  %al, %eax
        ret
.L8:
        movl    $0, %eax
.L1:
        ret
```

Assembler Code Explanation (1)

```
switch_eg:
        movl    $0, %eax     # return value set to 0
        testq   %rsi, %rsi   # sets cc based on %rsi & %rsi
        jle     .L1          # go to L1, where it returns 0
        cmpq    $12, %rdi
        ja      .L8
        leaq    .L4(%rip), %rdx
        movslq  (%rdx,%rdi,4), %rax
        addq    %rdx, %rax
        jmp     *%rax
```

- **testq %rsi, %rsi**
  - **sets cc based on the contents of %rsi (d)**
  - **jle**
    - **jumps if `(SF^OF)|ZF`**
    - **OF is not set**
    - **jumps if SF or ZF is set (i.e., < 1)**

Assembler Code Explanation (2)

```
switch_eg:
        movl    $0, %eax          # return value set to 0
        testq   %rsi, %rsi        # sets cc based on %rsi & %rsi
        jle     .L1               # go to L1, where it returns 0
        cmpq    $12, %rdi         # %rdi : 12
        ja      .L8               # go to L8 if %rdi > 12 or < 0
        leaq    .L4(%rip), %rdx
        movslq  (%rdx,%rdi,4), %rax
        addq    %rdx, %rax
        jmp     *%rax
```

- **ja .L8**
  - **unsigned comparison, though m is signed!**
  - **jumps if %rdi > 12**
  - **also jumps if %rdi is negative**

Assembler Code Explanation (3)

```
switch_eg:
        movl    $0, %eax
        testq   %rsi, %rsi
        jle     .L1
        cmpq     $12, %rdi
        ja      .L8
        leaq     .L4(%rip), %rdx
        movslq  (%rdx,%rdi,4), %rax
        addq     %rdx, %rax
        jmp      *%rax
```

```
        .section     .rodata
        .align 4
.L4:
        .long    .L8-.L4 # m=0
        .long    .L3-.L4 # m=1
        .long    .L6-.L4 # m=2
        .long    .L3-.L4 # m=3
        .long    .L5-.L4 # m=4
        .long    .L3-.L4 # m=5
        .long    .L5-.L4 # m=6
        .long    .L3-.L4 # m=7
        .long    .L3-.L4 # m=8
        .long    .L5-.L4 # m=9
        .long    .L3-.L4 # m=10
        .long    .L5-.L4 # m=11
        .long    .L3-.L4 # m=12
        .text
```

Assembler Code Explanation (4)

```
switch_eg:
        movl    $0, %eax
        testq   %rsi, %rsi
        jle     .L1
        cmpq     $12, %rdi
        ja      .L8
        leaq    .L4(%rip), %rdx
        movslq  (%rdx,%rdi,4), %rax
        addq    %rdx, %rax
        jmp     *%rax
```

indirect jump

```
        .section    .rodata
        .align 4
.L4:
        .long   .L8-.L4 # m=0
        .long   .L3-.L4 # m=1
        .long   .L6-.L4 # m=2
        .long   .L3-.L4 # m=3
        .long   .L5-.L4 # m=4
        .long   .L3-.L4 # m=5
        .long   .L5-.L4 # m=6
        .long   .L3-.L4 # m=7
        .long   .L3-.L4 # m=8
        .long   .L5-.L4 # m=9
        .long   .L3-.L4 # m=10
        .long   .L5-.L4 # m=11
        .long   .L3-.L4 # m=12
        .text
```

Assembler Code Explanation (5)

```
switch_eg:
        movl    $0, %eax
        testq   %rsi, %rsi
        jle     .L1
        cmpq     $12, %rdi
        ja      .L8
        leaq     .L4(%rip), %rdx
        movslq   (%rdx,%rdi,4), %rax
        addq     %rdx, %rax
        jmp      *%rax
```

```
        .section     .rodata
        .align 4
.L4:
        .long   .L8-.L4 # m=0
        .long   .L3-.L4 # m=1
        .long   .L6-.L4 # m=2
        .long   .L3-.L4 # m=3
        .long   .L5-.L4 # m=4
        .long   .L3-.L4 # m=5
        .long   .L5-.L4 # m=6
        .long   .L3-.L4 # m=7
        .long   .L3-.L4 # m=8
        .long   .L5-.L4 # m=9
        .long   .L3-.L4 # m=10
        .long   .L5-.L4 # m=11
        .long   .L3-.L4 # m=12
        .text
```

Assembler Code Explanation (6)

```
switch_eg:
        movl    $0, %eax
        testq   %rsi, %rsi
        jle     .L1
        cmpq     $12, %rdi
        ja      .L8
        leaq     .L4(%rip), %rdx
        movslq   (%rdx,%rdi,4), %rax
        addq     %rdx, %rax
        jmp      *%rax
```

```
        .section      .rodata
        .align 4
.L4:
        .long    .L8-.L4 # m=0
        .long    .L3-.L4 # m=1
        .long    .L6-.L4 # m=2
        .long    .L3-.L4 # m=3
        .long    .L5-.L4 # m=4
        .long    .L3-.L4 # m=5
        .long    .L5-.L4 # m=6
        .long    .L3-.L4 # m=7
        .long    .L3-.L4 # m=8
        .long    .L5-.L4 # m=9
        .long    .L3-.L4 # m=10
        .long    .L5-.L4 # m=11
        .long    .L3-.L4 # m=12
        .text
```

Assembler Code Explanation (7)

```
switch_eg:
        movl    $0, %eax
        testq   %rsi, %rsi
        jle     .L1
        cmpq     $12, %rdi
        ja      .L8
        leaq    .L4(%rip), %rdx
        movslq  (%rdx,%rdi,4), %rax
        addq    %rdx, %rax
        jmp     *%rax
```

```
        .section    .rodata
        .align 4
.L4:
        .long   .L8-.L4 # m=0
        .long   .L3-.L4 # m=1
        .long   .L6-.L4 # m=2
        .long   .L3-.L4 # m=3
        .long   .L5-.L4 # m=4
        .long   .L3-.L4 # m=5
        .long   .L5-.L4 # m=6
        .long   .L3-.L4 # m=7
        .long   .L3-.L4 # m=8
        .long   .L5-.L4 # m=9
        .long   .L3-.L4 # m=10
        .long   .L5-.L4 # m=11
        .long   .L3-.L4 # m=12
        .text
```

Switch Statements and Traps

- The code we just looked at was compiled with gcc's O1 flag
  – a moderate amount of "optimization"
- Traps was compiled with the O1 flag
  – no optimization
- O0 often produces easier-to-read (but less efficient) code
  – not so for switch

Gdb and Switch (1)

```
│B+ 0x555555555165 <switch_eg>      mov    $0x0,%eax                                │
│   0x55555555516a <switch_eg+5>    test   %rsi,%rsi                                │
│   0x55555555516d <switch_eg+8>    jle    0x5555555551ab <switch_eg+70>            │
│   0x55555555516f <switch_eg+10>   cmp    $0xc,%rdi                                │
│   0x555555555173 <switch_eg+14>   ja     0x5555555551a6 <switch_eg+65>            │
│   0x555555555175 <switch_eg+16>   lea    0xe88(%rip),%rdx  # 0x555555556004       │
│   0x55555555517c <switch_eg+23>   movslq (%rdx,%rdi,4),%rax                       │
│   0x555555555180 <switch_eg+27>   add    %rdx,%rax                                │
│  >0x555555555183 <switch_eg+30>   jmp    *%rax                                    │
│   0x555555555185 <switch_eg+32>   cmp    $0x1f,%rsi                               │
│   0x555555555189 <switch_eg+36>   setle  %al                                      │
│   0x55555555518c <switch_eg+39>   movzbl %al,%eax                                 │
│   0x55555555518f <switch_eg+42>   ret                                             │
```

```
(gdb) x/14dw $rdx
0x555555556004: -3678   -3711   -3700   -3711
0x555555556014: -3689   -3711   -3689   -3711
0x555555556024: -3711   -3689   -3711   -3689
0x555555556034: -3711   1734439765
```

Gdb and Switch (2)

```
>0x555555555183 <switch_eg+30>   jmp    *%rax
 0x555555555185 <switch_eg+32>   cmp    $0x1f,%rsi
 0x555555555189 <switch_eg+36>   setle  %al
 0x55555555518c <switch_eg+39>   movzbl %al,%eax
 0x55555555518f <switch_eg+42>   ret
 0x555555555190 <switch_eg+43>   cmp    $0x1c,%rsi
 0x555555555194 <switch_eg+47>   setle  %al
 0x555555555197 <switch_eg+50>   movzbl %al,%eax
 0x55555555519a <switch_eg+53>   ret
 0x55555555519b <switch_eg+54>   cmp    $0x1e,%rsi
 0x55555555519f <switch_eg+58>   setle  %al
 0x5555555551a2 <switch_eg+61>   movzbl %al,%eax
 0x5555555551a5 <switch_eg+64>   ret
 0x5555555551a6 <switch_eg+65>   mov    $0x0,%eax
 0x5555555551ab <switch_eg+70>   ret
```

Offset -3711

```
(gdb) x/14dw $rdx
0x555555556004: -3678   -3711   -3700   -3711
0x555555556014: -3689   -3711   -3689   -3711
0x555555556024: -3711   -3689   -3711   -3689
0x555555556034: -3711   1734439765
```

Quiz 1

What C code would you compile to get the following assembler code?

```
        movq    $0, %rax
.L2:
        movq    %rax, a(,%rax,8)
        addq    $1, %rax
        cmpq    $10, %rax
        jl      .L2
        ret
```

```
long a[10];
void func() {
  long i;
  for (i=0; i<10; i++)
    a[i]= 1;
}
```

a

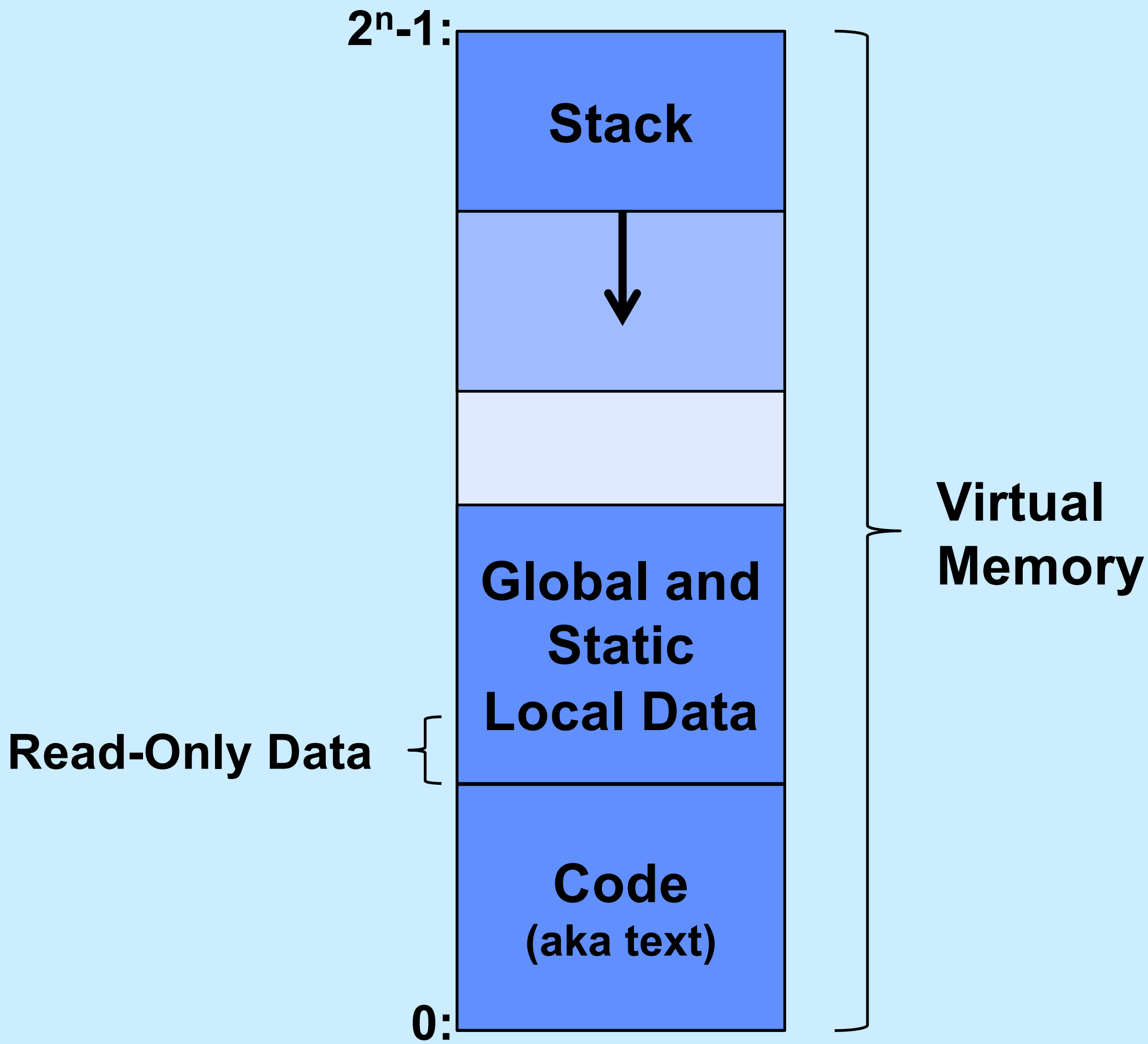
```
long a[10];
void func() {
  long i=0;
  while (i<10)
    a[i]= i++;
}
```

b

```
long a[10];
void func() {
  long i=0;
  switch (i) {
case 0:
    a[i] = 0;
    break;
default:
    a[i] = 10
  }
}
```

c

Digression (Again): Where Stuff Is (Roughly)



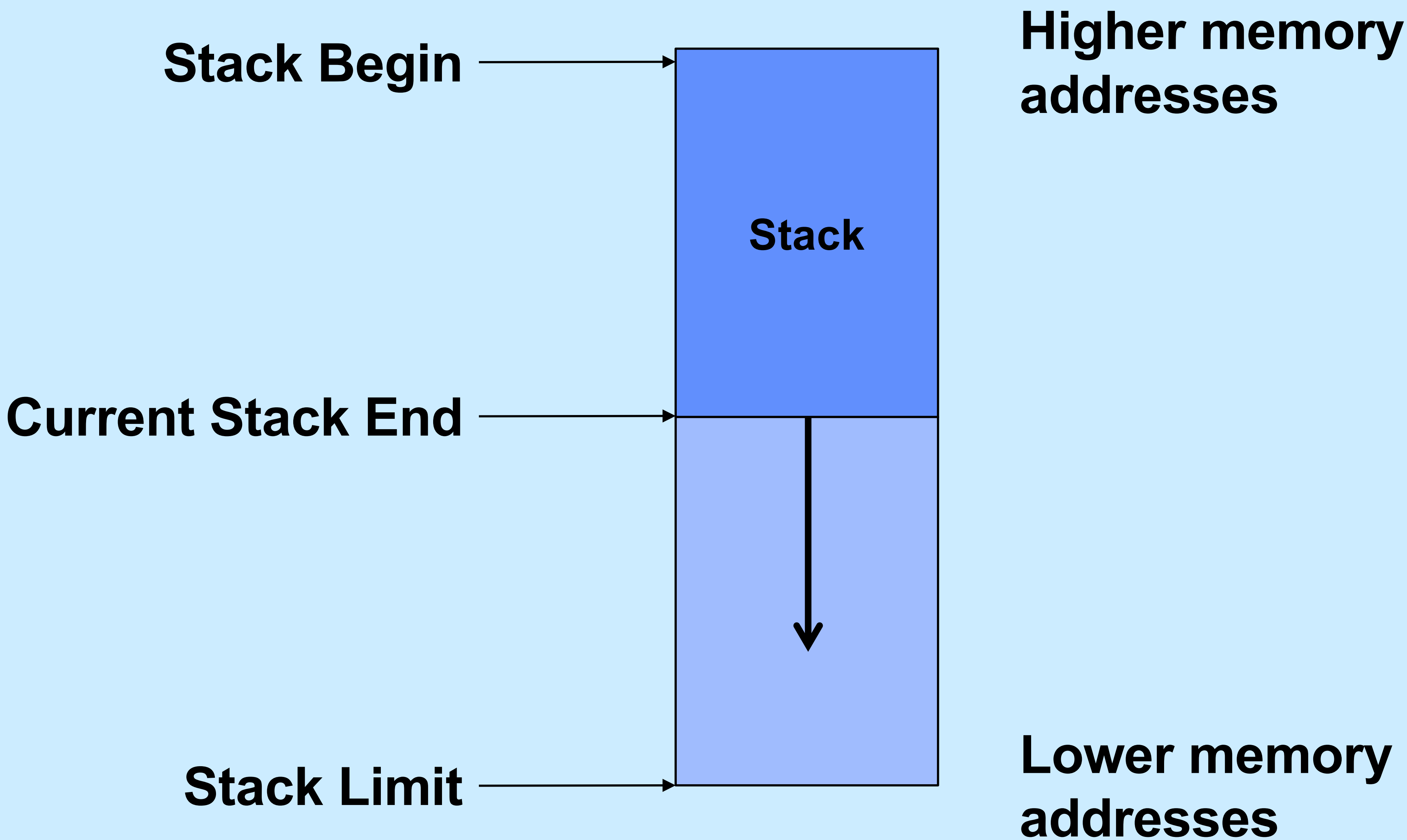
Function Call and Return

- Function A calls function B
- Function B calls function C

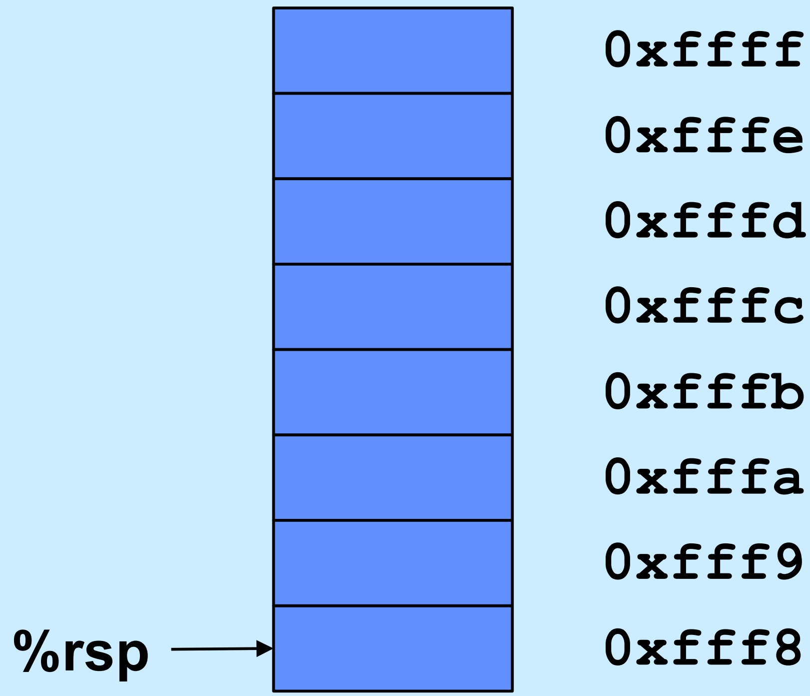
... several million instructions later

- C returns
  – how does it know to return to B?
- B returns
  – how does it know to return to A?

The Runtime Stack

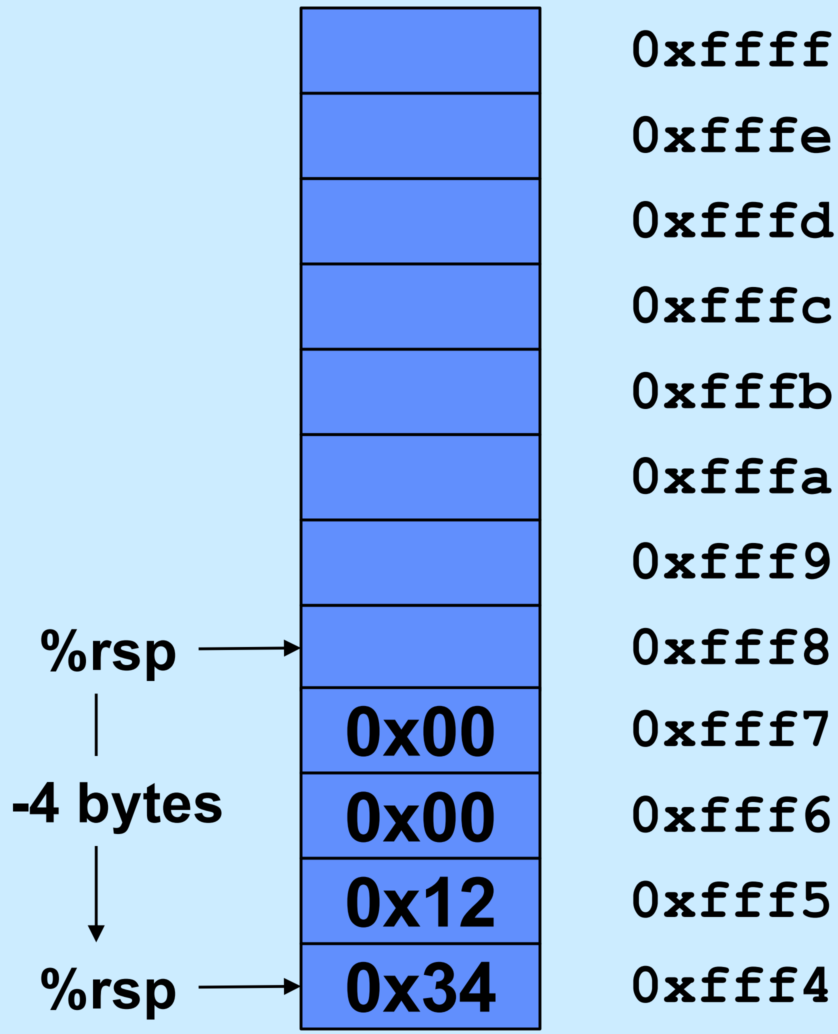


Stack Operations



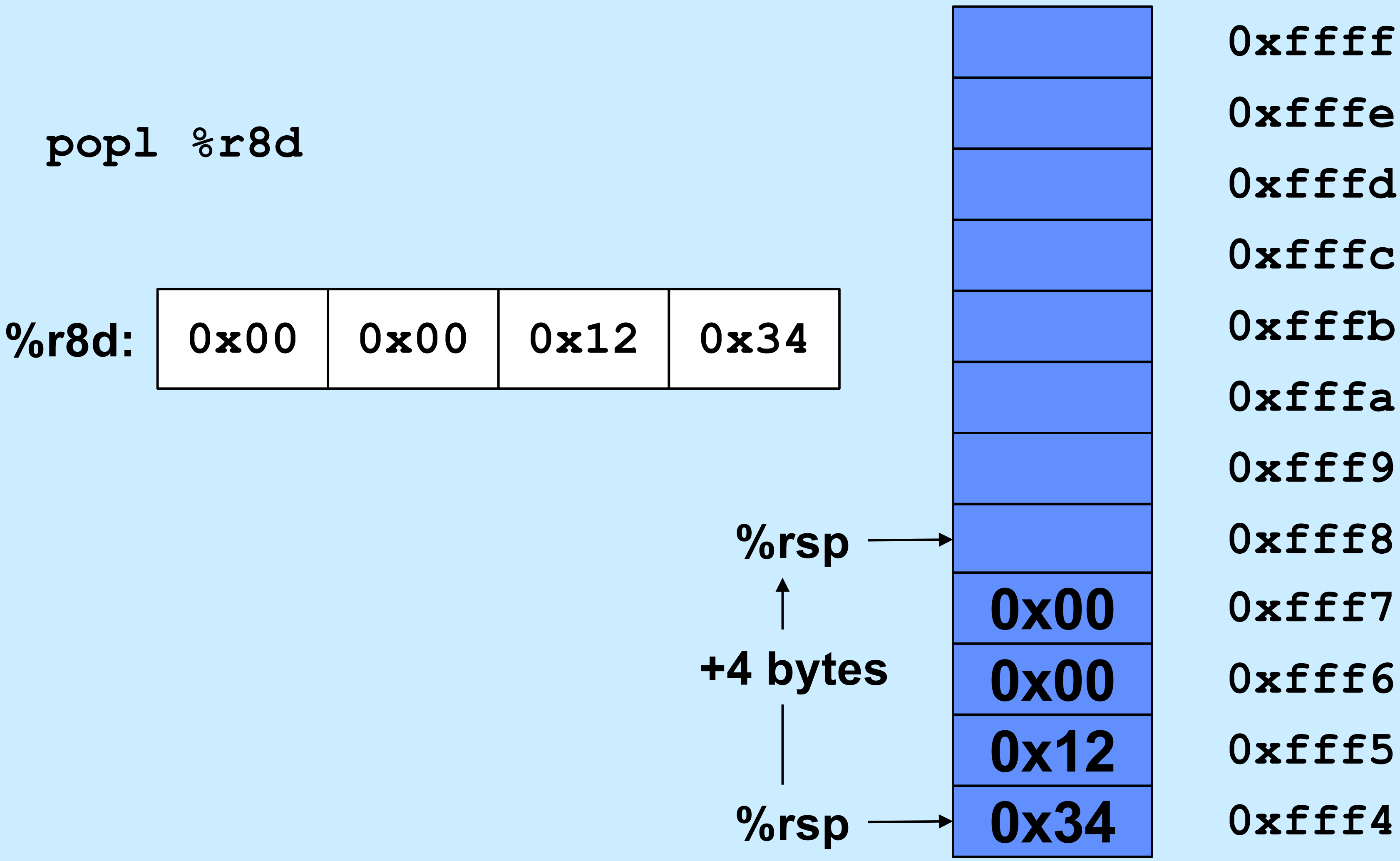
Push

`pushl $0x1234`



Pop

```
popl %r8d
```



%r8d:
0x00
0x00
0x12
0x34
%rsp
+4 bytes
%rsp
0x00
0x00
0x12
0x34
0xffff
0xfffe
0xfffd
0xfffc
0xfffb
0xfffa
0xfff9
0xfff8
0xfff7
0xfff6
0xfff5
0xfff4

Call and Return

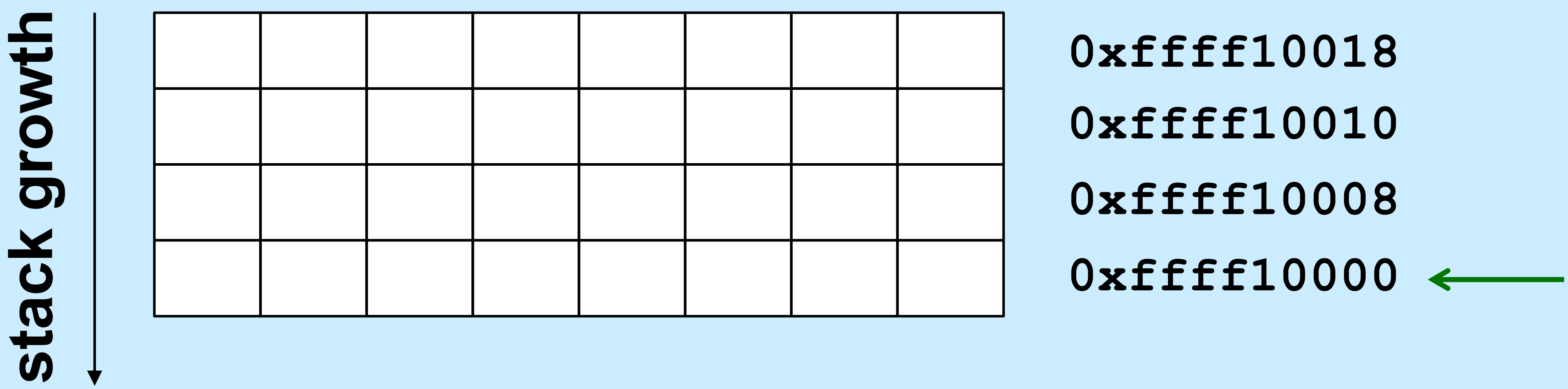
```
0x2000: func:
    ... ...
0x2200: movq $6, %rax
0x2203: ret
```

```
0x1000: call func
0x1004: addq $3, %rax
```

Call and Return

```
0x2000:  func:
   ...   ...
0x2200:  movq $6, %rax
0x2203:  ret
```

```
0x1000:  call func
0x1004:  addq $3, %rax
```



								%rax
00	00	00	00	00	00	10	00	%rip
00	00	00	0f	ff	f1	00	00	%rsp

Call and Return

```
0x2000:  func:
   ...   ...
0x2200:  movq $6, %rax
0x2203:  ret
```

```
0x1000:  call func
0x1004:  addq $3, %rax
```

stack growth

								0xffff10018
								0xffff10010
								0xffff10008
								0xffff10000
00	00	00	00	00	00	10	04	0xffff0fff8

								%rax
00	00	00	00	00	00	20	00	%rip
00	00	00	0f	ff	f0	ff	f8	%rsp

Call and Return

```
0x2000:  func:
   ...   ...
0x2200:  movq $6, %rax
0x2203:  ret
```

```
0x1000:  call func
0x1004:  addq $3, %rax
```

stack growth

								0xffff10018
								0xffff10010
								0xffff10008
								0xffff10000
00	00	00	00	00	00	10	04	0xffff0fff8

00	00	00	00	00	00	00	06	%rax
00	00	00	00	00	00	22	03	%rip
00	00	00	0f	ff	f0	ff	f8	%rsp

Call and Return

```
0x2000:  func:
   ...   ...
0x2200:  movq $6, %rax
0x2203:  ret
```

```
   0x1000:  call func
→  0x1004:  addq $3, %rax
```

stack growth

								0xffff10018
								0xffff10010
								0xffff10008
								0xffff10000 ←
00	00	00	00	00	00	10	04	0xffff0fff8

00	00	00	00	00	00	00	06	%rax
00	00	00	00	00	00	10	04	%rip
00	00	00	0f	ff	f1	00	00	%rsp