Machine Level Programming

History of Intel Processors

- Eveolutionary design: **Backwards compatible** up until `8086` in 1978
- **C**omplex **I**nstruction **S**et **C**omputer (CISC)

RISC vs CISC

- CISC has variable length instructions
- RISC has constant length instructions

1. Intel x86(8086)
    i. IA32 to IA64
    ii. (after x86-64) EM64T(almost same as AMD x86-64)
2. AMD x86-64

C, Assembly, machine code

- ***Architecture***: The parts of a processor design that one needs to understand or write assembly/machine code
    - **ISA(Instruction Set Architecture)**
        - e.g., x86, IA32, Itanium, x86-64, ARM
- ***Microarchitecture***: Implementation of the architecture

form of code:

- Machine Code: the byte-level programs that a processor executes
- Assembly Code: A text representation of machine code

Assembly/Machine Code View

Programmer-Visible State (shown by ISAs)

- PC(Program Counter)
    - Address of next instruction
    - RIP in (x86-64)
- Register file
    - Heavily used program data
- Condition codes
    - store status information about most recent arithmetic or logical op
    - Used for conditional branching
- Memory(external)

Compiling Into Assembly

```
long plus(long x, long y);

void sumstore(long x, long y, long *dest) {
    long t = plus(x, y);
    *dest = t;
}
```

```
3_1.o:     file format elf64-x86-64


Disassembly of section .text:

0000000000000000 <sumstore>:
   0:   f3 0f 1e fa             endbr64
   4:   53                      push   %rbx
   5:   48 89 d3                mov    %rdx,%rbx
   8:   e8 00 00 00 00          call   d <sumstore+0xd>
   d:   48 89 03                mov    %rax,(%rbx)
  10:   5b                      pop    %rbx
  11:   c3                      ret
```

Integer Registers

- In x86-64
  `ax` `bx` `cx` `dx` `si` `di` `sp` `bp` (in 8bytes `r` 4bytes `e`)
  `r8` `r9` `r10` `r11` `r12` `r13` `r14` `r15` (in 4bytes: add `d`)
- In IA32
  `eax` (32bit): 16bit `ax` (`ah`, `al`); origin: accumulate
  `ecx` (32bit): 16bit `cx` (`ch`, `cl`); origin: counter
  `edx` (32bit): 16bit `bx` (`bh`, `bl`); origin: data
  `ebx` (32bit): 16bit `dx` (`dh`, `dl`); origin: base
  `esi` (32bit): 16bit `si` (`sih`, `sil`); origin: source index
  `edi` (32bit): 16bit `di` (`dil`, `dil`); origin: destination index
  `esp` (32bit): 16bit `sp` (`spl`, `spl`); origin: stack pointer
  `ebp` (32bit): 16bit `bp` (`bpl`, `bpl`); origin: base pointer

understanding `movq`

In x86asm, There are three operand types: **immediate, register, memory**

- immediate: constant integer data like `$0x400` `$-533`
- register: one of 16 integer regs
  - e.g., `%rax`, `%r13`
  - but `%rsp` is reserved for special use
- memory: 8 bytes of memory at address given by register
  - e.g., `(%rax)`

movq usage

`movq $src, $dest`

- limit: Cannot do memory-memory transfer with a single instruction(because memory is external device to cpu)

memory addressing modes

`(R)` means `Mem[Reg[R]]`
`D(R)` means `Mem[Reg[R]+D]`, constant displacement `D` specifies offset
`movq 8(%rbp), %rdx`

```c
int swap (long *xp, long *yp) {
    long t0 = *xp;
    long t1 = *yp;
    *xp = t1;
    *yp = t0;
}
```

```
swap:
    movq (%rdi), %rax
    movq (%rsi), %rdx
    movq %rdx, (%rdi)
    movq %rax, (%rsi)
    ret
```

Complete form of memory addressing modes:
`D(Rb, Ri, S)` means `Mem[Reg[Rb] + S*Reg[Ri] + D]`

- `D`: Constant "displacement"

- `Rb` : Base Register
- `Ri` : Index Register
- `S` : Scale Factor(1, 2, 4, or 8)

for example:

`%rdx`	`%rcx`
`0xf000`	`0x0100`

- `0x8(%rdx)` = `0xf008`
- `(%rdx, %rcx)` = `0xf100`
- `(%rdx, %rcx, 4)` = `0xf400`
- `0x80(,%rdx, 2)` = `0x1e080`

Arithmetic & Logical Operations

- `leaq $src, $dst`
  - computing address without memory reference like `p = &x[i]`
  - computing arithmetic expression `x + k * y`
- `addq $src, $dst`
- `subq $src, $dst`
- `imulq $src, $dst`
- `salq $src, $dst`
- `sarq $src, $dst`
- `shrq $src, $dst`
- `xorq $src, $dst`
- `andq $src, $dst`
- `orq $src, $dst`
  all the above operator operates like `dest = dest # src`
- `incq $dest`
- `decq $dest`
- `negq $dest`
- `notq $dest`

Control

Processor State(x86-64, Partial)

- Temporary data(`%rax` , ...)
- Location of runtime stack(`%rsp`)
- Location of current code control point(`%rip` , instruction point)
- Status of recent tests(`CF` , `ZF` , `SF` , `OF`)

Condition Codes

- Single bit registers
  - `CF` Carry flag (for unsigned)
  - `SF` Sign flag (for signed)
  - `ZF` Zero flag
  - `OF` Overflow flag (for signed)

Conditional Codes(Implicit Setting)

Implicit setting is codes are set by arithmetic operations(`addq` , `subq` , `mulq`)
for example: `addq` : `t = a + b`

- `CF` set if carry out from most significant bit or unsigned overflow
- `ZF` set if `t == 0`

- `SF` set if `t < 0` (as signed)
- `OF` set if two's-complement overflow or signed overflow
  `(a > 0 && b > 0 && (a + b) < 0) || (a < 0 && b < 0 &&  (a + b) >= 0)`

The codes are not implictly set by `leaq`, because it is not designed to be used as arithmetic but used as **address calculation**. so it cannot affect to conditional codes.

Conditional Codes(Explicit Setting)

The codes are set explictly by compare instruction.

`cmpq b, a` is computing `a - b` without setting destination.

- `CF` set if carry out from most significant bit or unsigned overflow
- `ZF` set if `a == b` or `a - b == 0`
- `SF` set if `(a - b) < 0` (as signed)
- `OF` set if two's-complement overflow or signed overflow
  `(a > 0 && b > 0 && (a - b) < 0) || (a < 0 && b < 0 &&  (a - b) >= 0)`

And explictly set by test instruction

`testq b, a` is computing `a & b` without setting destination.

Sets condition codes based on value of `a & b` it is useful to have one of the operands be a mask.

- `ZF` set when `a & b == 0`
- `SF` set when `a & b < 0`

Reading Condition Codes

`setX` : set single byte based on combination of condition codes

setX	effect	desc
`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)

it deos not alter remaining bytes of registers. only use 1 byte register(`%al`, `%bl`)

```
cmpq %rsi(y), %rdi(x)  # compare x and y
setg %al         # set when >(greater)
movzbl %al, %eax # move zero extend byte to long
ret
```

Conditional Branches

Jumping

`jX` jump to different part of code depending on condition codes.

jX	condition	desc
`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)

Old Style Conditional Branch

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

```
3_3.o:     file format elf64-x86-64


Disassembly of section .text:

0000000000000000 <absdiff>:
   0:   f3 0f 1e fa             endbr64
   4:   48 39 f7                cmpq   %rsi,%rdi
   7:   7e 07                   jle    10 <absdiff+0x10>
   9:   48 89 f8                movq   %rdi,%rax
   c:   48 29 f0                subq   %rsi,%rax
   f:   c3                      retq
  10:   48 89 f0                movq   %rsi,%rax
  13:   48 29 f8                subq   %rdi,%rax
  16:   c3                      retq
```

expressing with `goto`

```
long absdiff_j(long x, long y) {
  long result;
  int ntest = x <= y;
  if (ntest) goto Else;
  result = x-y;
  goto Done;
Else:
  result = y-x;
Done:
  return result;
}
```

```
gcc: error: unrecognized command-line option ‘-rno-if-conversion’; did you mean ‘-fno-if-conversion’?
```

Conditional Move

But this branchings are very disruptive to instruction flow through pipelines, **Conditional Moves** are highly used because they do not require control transfer.

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

```
3_5.o:     file format elf64-x86-64


Disassembly of section .text:

0000000000000000 <absdiff>:
   0:   f3 0f 1e fa             endbr64
   4:   48 89 fa                movq   %rdi,%rdx
   7:   48 89 f0                movq   %rsi,%rax
   a:   48 29 f2                subq   %rsi,%rdx
   d:   48 29 f8                subq   %rdi,%rax
  10:   48 39 f7                cmpq   %rsi,%rdi
  13:   48 0f 4f c2             cmovgq %rdx,%rax
  17:   c3                      retq
```

However, there are several *bad cases* for conditional move.

- expansive computations

```
val = Test(x) ? Hard1(x) : Hard2(x);
```

because both values are get computed. only simple computations are effective for conditional moves.

- risky computations

```
val = p ? *p : 0;
```

both values get computed may have undesiarable effects.

- Computations with side effects

```
val = x > 0 ? x*=7 : x+=3;
```

each expression has side-effect.

Loop

do-while

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

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

```
3_6.o:     file format elf64-x86-64


Disassembly of section .text:

0000000000000000 <pcount_goto>:
   0:   f3 0f 1e fa             endbr64
   4:   b8 00 00 00 00          movl   $0x0,%eax
   9:   48 89 fa                movq   %rdi,%rdx
   c:   83 e2 01                andl   $0x1,%edx
   f:   48 01 d0                addq   %rdx,%rax
  12:   48 d1 ef                shrq   $1,%rdi
  15:   75 f2                   jne    9 <pcount_goto+0x9>
  17:   c3                      retq
```

general do-while translation

`do {`
`  Body`
`} while (Test);`
	`loop:`
	`  Body`
	`  if (Test) goto loop;`

while

general while translation#1

it is called **jump-to-middle translation**, used with `-O0` (or `-Og`) flag.

`while(Test) {`
`  Body`
`}`
	`  goto test;`
	`loop:`
	`  Body`
	`test:`
	`  if (Test)`
	`  goto loop;`
	`done:`

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

```
jmp-to-middle translation

3_7.o:     file format elf64-x86-64


Disassembly of section .text:

0000000000000000 <pcount_while>:
   0:   f3 0f 1e fa             endbr64
   4:   b8 00 00 00 00          movl   $0x0,%eax
   9:   eb 0c                   jmp    17 <pcount_while+0x17>
   b:   48 89 fa                movq   %rdi,%rdx
   e:   83 e2 01                andl   $0x1,%edx
  11:   48 01 d0                addq   %rdx,%rax
  14:   48 d1 ef                shrq   $1,%rdi
  17:   48 85 ff                testq  %rdi,%rdi
  1a:   75 ef                   jne    b <pcount_while+0xb>
  1c:   c3                      retq
```

general while translation#2

while to do-while conversion, used with `-O1` flag.

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

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

```
while to do-while conversion

3_8.o:     file format elf64-x86-64


Disassembly of section .text:

0000000000000000 <pcount_while>:
   0:   f3 0f 1e fa             endbr64
   4:   48 85 ff                testq  %rdi,%rdi
   7:   74 14                   je     1d <pcount_while+0x1d>
   9:   b8 00 00 00 00          movl   $0x0,%eax
   e:   48 89 fa                movq   %rdi,%rdx
  11:   83 e2 01                andl   $0x1,%edx
  14:   48 01 d0                addq   %rdx,%rax
  17:   48 d1 ef                shrq   $1,%rdi
  1a:   75 f2                   jne    e <pcount_while+0xe>
  1c:   c3                      retq
  1d:   b8 00 00 00 00          movl   $0x0,%eax
  22:   c3                      retq
```

for loop form

```
for (init; test; update) {
  Body
}
```

for-to-while conversion

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

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

```
#include <stddef.h>
#define WSIZE 8 * sizeof(int)

long pcount_for(unsigned long x) {
  size_t i;
  long result = 0;
  for (i = 0; i < WSIZE; i++) {
    unsigned bit = (x >> i) & 0x1;
    result += bit;
  }
  return result;
}
```

```
#include <stddef.h>
#define WSIZE 8 * sizeof(int)
long pcount_for(unsigned long x) {
  size_t i;
  long result = 0;
  i = 0;
  while(i < WSIZE) {
    unsigned bit = (x >> i) & 0x1;
    result += bit;
    i++;
  }
  return result;
}
```

```
3_9.o:     file format elf64-x86-64


Disassembly of section .text:

0000000000000000 <pcount_for>:
   0:   f3 0f 1e fa             endbr64
   4:   45 31 c0                xorl   %r8d,%r8d
   7:   31 c9                   xorl   %ecx,%ecx
   9:   0f 1f 80 00 00 00 00    nopl   0x0(%rax)
  10:   48 89 f8                movq   %rdi,%rax
  13:   48 d3 e8                shrq   %cl,%rax
  16:   48 83 c1 01             addq   $0x1,%rcx
  1a:   83 e0 01                andl   $0x1,%eax
  1d:   49 01 c0                addq   %rax,%r8
  20:   48 83 f9 20             cmpq   $0x20,%rcx
  24:   75 ea                   jne    10 <pcount_for+0x10>
  26:   4c 89 c0                movq   %r8,%rax
  29:   c3                      retq
```

```
3_10.o:     file format elf64-x86-64


Disassembly of section .text:

0000000000000000 <pcount_for>:
   0:   f3 0f 1e fa             endbr64
   4:   45 31 c0                xorl   %r8d,%r8d
   7:   31 c9                   xorl   %ecx,%ecx
   9:   0f 1f 80 00 00 00 00    nopl   0x0(%rax)
  10:   48 89 f8                movq   %rdi,%rax
  13:   48 d3 e8                shrq   %cl,%rax
  16:   48 83 c1 01             addq   $0x1,%rcx
  1a:   83 e0 01                andl   $0x1,%eax
  1d:   49 01 c0                addq   %rax,%r8
  20:   48 83 f9 20             cmpq   $0x20,%rcx
  24:   75 ea                   jne    10 <pcount_for+0x10>
  26:   4c 89 c0                movq   %r8,%rax
  29:   c3                      retq
```

for to do-while conversion, initial test can be optimized away.

Switch

Jump Table Structure

Switch form

```c
long switch_eg (long x, long y, long z) {
  long w = 1;
  switch(x) {
  case 1:
    w = y*z;
    break;
  case 2:
    w = y/z;
    /* Fall Through */
  case 3:
    w += z;
    break;
  case 5:
  case 6:
    w -= z;
    break;
  case 7:
    w *= z;
    break;
  default:
    w = 2;
  }
  return w;
}
```

```
        .file   "fovuk830h_code_chunk.gcc"
        .text
        .globl  switch_eg
        .type   switch_eg, @function
switch_eg:
        endbr64
        movq    %rdx, %rcx
        cmpq    $7, %rdi
        ja      .L9
        leaq    .L4(%rip), %rdx
        movslq  (%rdx,%rdi,4), %rax
        addq    %rdx, %rax
        notrack jmp     *%rax
        .section        .rodata
        .align 4
        .align 4
.L4:
        .long   .L9-.L4
        .long   .L8-.L4
        .long   .L7-.L4
        .long   .L10-.L4
        .long   .L9-.L4
        .long   .L5-.L4
        .long   .L5-.L4
        .long   .L3-.L4
        .text
.L3:
        movq    %rcx, %rax
        ret
.L8:
        movq    %rcx, %rax
        imulq   %rsi, %rax
        ret
.L7:
        movq    %rsi, %rax
        cqto
        idivq   %rcx
.L6:
        addq    %rcx, %rax
        ret
.L10:
        movl    $1, %eax
        jmp     .L6
.L5:
        movl    $1, %eax
        subq    %rcx, %rax
        ret
.L9:
        movl    $2, %eax
        ret
        .size   switch_eg, .-switch_eg
        .ident  "GCC: (Ubuntu 11.5.0-2ubuntu1) 11.5.0"
        .section        .note.GNU-stack,"",@progbits
        .section        .note.gnu.property,"a"
        .align 8
        .long   1f - 0f
        .long   4f - 1f
        .long   5
0:
        .string "GNU"
1:
        .align 8
        .long   0xc0000002
        .long   3f - 2f
2:
        .long   0x3
3:
        .align 8
4:
```

Procedures {#procedures }

Mechanisms in Procedures

- **Passing control**
  - to beginning of procedure code
  - back to return point
- **Passing data**
  - procedure arguments
  - return value
- **Memory management**
  - allocate during procedure execution
  - deallocate upon return

this mechanisms are all implemented with machine instructions. **x86-64 implementation** of a procedure used only those mechanisms required.

Stack Structure

x86-64 Stack

Region of memory managed with *stack discipline*. It grows toward lower addresses. `%rsp` contains lowest stack address(address of top element).

`pushq $src`

- fetches operand at src
- decrement `%rsp` by 8
- write operand at address given by `%rsp`

`popq $dest`

- read value at address given by `%rsp`
- increment `%rsp` by 8
- store value at dest(must be register)

Procedure Control Flow

```
long mult2(long x, long y) {
  long t = x * y;
  return t;
}

void multstore(long x, long y, long *dest) {
  long t = mult2(x, y);
  *dest = t;
}
```

```
3_12.o:     file format elf64-x86-64


Disassembly of section .text:

0000000000000000 <mult2>:
   0:   f3 0f 1e fa             endbr64
   4:   48 89 f8                movq   %rdi,%rax
   7:   48 0f af c6             imulq  %rsi,%rax
   b:   c3                      retq

000000000000000c <multstore>:
   c:   f3 0f 1e fa             endbr64
  10:   53                      pushq  %rbx
  11:   48 89 d3                movq   %rdx,%rbx
  14:   e8 00 00 00 00          callq  19 <multstore+0xd>
  19:   48 89 03                movq   %rax,(%rbx)
  1c:   5b                      popq   %rbx
  1d:   c3                      retq
```

Procedure call `call label`

- push return address on stack
- jmp to label
  Return address:
- Address of the next instruction right after call
  Procedure return: `ret`

Procedure Data Flow

- registers
  - first 6 args: `%rdi`, `%rsi`, `%rdx`, `%rcx`, `%r8`, `%r9`
  - return value: `rax`
- stack

for example with above example

```
3_12.o:     file format elf64-x86-64


Disassembly of section .text:

0000000000000000 <mult2>:
   0:   f3 0f 1e fa             endbr64
   4:   48 89 f8                movq   %rdi,%rax
   7:   48 0f af c6             imulq  %rsi,%rax
   b:   c3                      retq

000000000000000c <multstore>:
   c:   f3 0f 1e fa             endbr64
  10:   53                      pushq  %rbx
  11:   48 89 d3                movq   %rdx,%rbx
  14:   e8 00 00 00 00          callq  19 <multstore+0xd>
  19:   48 89 03                movq   %rax,(%rbx)
  1c:   5b                      popq   %rbx
  1d:   c3                      retq
```

- with above `mult2` variable `t` is already stored in `%rax`
- so `movq %rax,(%rbx)` where `%rbx` is `long*dest`

Managing local data

Stack-Based Languages

In languages that support recursion

- Code must be "reentrant", which means multiple simultaneous instantiations of single procedure.
- Need some place to store ***state*** of each instantiation: (**args**, **local variables**, **return pointer**)

In order to get this, **stack discipline** is used. state for given procedure needed for limited time(from called to return): Calle returns before caller does.
Stack allocated in **frames**, state for single procdure instantiation.
When function is called, a new stack frame is created above stack top. And then when the function is returned, a corresponding frame is popped. and return to previous call state.

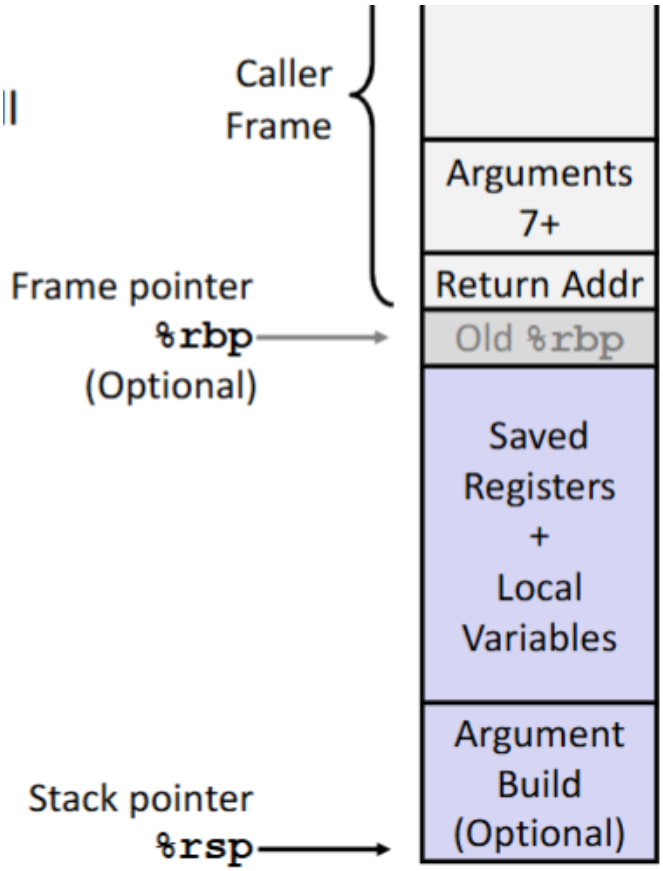
Stack Frame

is consist of **return information**, **local storage(if needed)** and **temporary space(if needed)**.

- `%rbp` frame pointer
- `%rsp` stack pointer

Space allocated when enter procedure, "set-up" code and includes push by `call`.
Deallocated when return, "finish" code and includes pop by `ret`.

x86-64/Linux Stack Frame



- Arguments
- Local variables
- Old `rbp`

Register Saving Conventions

When calling function, the temporary value of registers could be removed by called function, it could be trouble. So there are **conventions** to save the registers value.

When procedure `yoo` calls `who` : `yoo` is `caller` , `who` is `callee`

- Caller saves temporary values in its frame before the call.
- Callee saves saves temporary values in its frame before using and restores them before returning to caller.

x86-64 Linux Register Usage

`%rbx` , `%r12` , `%r13` , `%r14` , `%r15`

- Callee-saved
- Callee must save & restore

`%rbp`

- Callee-saved
- Callee must save & restore
- May be used as frame pointer by callee
- Can mix & match

`%rsp`

- Special form of callee-saved
- Restored to original value upon exit from procedure

EX

- for compile w/o *stack canary*, add option `-fno-stack-protector`

```c
long incr(long *p, long val) {
  long x = *p;
  long y = x + val;
  *p = y;
  return x;
}
long call_incr() {
  long v1 = 15213;
  long v2 = incr(&v1, 3000);
  return v1 + v2;
}
```

```
3_13.o:     file format elf64-x86-64


Disassembly of section .text:

0000000000000000 <incr>:
   0:   f3 0f 1e fa             endbr64
   4:   48 8b 07                movq   (%rdi),%rax
   7:   48 01 c6                addq   %rax,%rsi
   a:   48 89 37                movq   %rsi,(%rdi)
   d:   c3                      retq

000000000000000e <call_incr>:
   e:   f3 0f 1e fa             endbr64
  12:   48 83 ec 10             subq   $0x10,%rsp
  16:   48 c7 44 24 08 6d 3b    movq   $0x3b6d,0x8(%rsp)
  1d:   00 00
  1f:   48 8d 7c 24 08          leaq   0x8(%rsp),%rdi
  24:   be b8 0b 00 00          movl   $0xbb8,%esi
  29:   e8 00 00 00 00          callq  2e <call_incr+0x20>
  2e:   48 03 44 24 08          addq   0x8(%rsp),%rax
  33:   48 83 c4 10             addq   $0x10,%rsp
  37:   c3                      retq
```

Recursive Function

```c
long pcount_r(unsigned long x) {
  if (x == 0) {
    return 0;
  } else {
    return (x & 1) + pcount_r(x >> 1);
  }
}
```

```
3_14.o:     file format elf64-x86-64


Disassembly of section .text:

0000000000000000 <pcount_r>:
   0:   f3 0f 1e fa             endbr64
   4:   b8 00 00 00 00          movl   $0x0,%eax
   9:   48 85 ff                testq  %rdi,%rdi
   c:   75 01                   jne    f <pcount_r+0xf>
   e:   c3                      retq
   f:   53                      pushq  %rbx
  10:   48 89 fb                movq   %rdi,%rbx
  13:   48 d1 ef                shrq   $1,%rdi
  16:   e8 00 00 00 00          callq  1b <pcount_r+0x1b>
  1b:   83 e3 01                andl   $0x1,%ebx
  1e:   48 01 d8                addq   %rbx,%rax
  21:   5b                      popq   %rbx
  22:   c3                      retq
```

Recursion is not a special function.

- Stack frames mean that each function call has private storage.
- Register saving conventions prevent one function call from corrupting another's data. *unless the explictly corrupting like buffer overflow*
- Stack discipline follows call/return pattern LIFO