Advanced C:
gcc inline assembly
(and a few words about optimization)

Trammell Hudson
<trammell.hudson@twosigma.com>

Why use inline assembly?


Go Faster!

POLY

Polynomial Evaluation

FORMAT

opcode arg.rx, degree.rw, tbladdr.ab

condition codes

N ⟵ R0 LSS 0;
Z ⟵ R0 EQL 0;
V ⟵ 0;
C ⟵ 0;

exceptions

floating overflow
floating underflow
reserved operand

opcodes

55	POLYF	Polynomial Evaluation F_floating
75	POLYD	Polynomial Evaluation D_floating
55FD	POLYG	Polynomial Evaluation G_floating
75FD	POLYH	Polynomial Evaluation H_floating

DESCRIPTION

The table address operand points to a table of polynomial coefficients. The coefficient of the highest-order term of the polynomial is pointed to by the table address operand. The table is specified with lower-order coefficients stored at increasing addresses. The data type of the coefficients is the same as the data type of the argument operand. The evaluation is carried out by Horner's method, and the contents of R0 (R1′R0 for POLYD and POLYG, R3′R2′R1′R0 for POLYH) are replaced by the result computed is:


THERE IS NO WAY TO GO THAT DIRECTION.
YOU'RE INSIDE BUILDING.
go,south
I DON'T KNOW THAT WORD.
go south
THERE IS NO WAY TO GO THAT DIRECTION.
YOU'RE INSIDE BUILDING.
THERE IS AN EMPTY BOTTLE HERE.
go west
digital VT102
digital
decdatasyst

RDTSC—Read Time-Stamp Counter

Opcode	Instruction	Description
0F 31	RDTSC	Read time-stamp counter into EDX:EAX

Description

This instruction loads the current value of the processor's time-stamp counter into the EDX:EAX registers. The time-stamp counter is contained in a 64-bit MSR. The high-order 32 bits of the MSR are loaded into the EDX register, and the low-order 32 bits are loaded into the EAX register. The processor increments the time-stamp counter MSR every clock cycle and resets it to 0 whenever the processor is reset.

CPUID—CPU Identification

Opcode	Instruction	Description
0F A2	CPUID	EAX ← Processor identification information

Description

This instruction provides processor identification information in registers EAX, EBX, ECX, and EDX. This information identifies Intel as the vendor, gives the family, model, and stepping of processor, feature information, and cache information. An input value loaded into the EAX register determines what information is returned, as shown in Table 3-6.

Table 3-6. Information Returned by CPUID Instruction

Initial EAX Value	Information Provided about the Processor	
0	EAX	Maximum CPUID Input Value (2 for the P6 family processors and 1 for the Pentium® processor and the later versions of Intel486™ processor that

SYSENTER—Fast Transition to System Call Entry Point

Opcode	Instruction	Description
0F, 34	SYSENTER	Transition to System Call Entry Point

Description

The SYSENTER instruction is part of the "Fast System Call" facility introduced on the Pentium® II processor. The SYSENTER instruction is optimized to provide the maximum performance for transitions to protection ring 0 (CPL = 0).

The SYSENTER instruction sets the following registers according to values specified by the operating system in certain model-specific registers.

CS register	set to the value of (SYSENTER_CS_MSR)
EIP register	set to the value of (SYSENTER_EIP_MSR)
SS register	set to the sum of (8 plus the value in SYSENTER_CS_MSR)
ESP register	set to the value of (SYSENTER_ESP_MSR)

The processor does not save user stack or return address information, and does not save any registers.

LGDT/LIDT—Load Global/Interrupt Descriptor Table Register

Opcode	Instruction	Description
0F 01 /2	LGDT *m16&32*	Load *m* into GDTR
0F 01 /3	LIDT *m16&32*	Load *m* into IDTR

Description

These instructions load the values in the source operand into the global descriptor table register (GDTR) or the interrupt descriptor table register (IDTR). The source operand specifies a 6-byte memory location that contains the base address (a linear address) and the limit (size of table in bytes) of the global descriptor table (GDT) or the interrupt descriptor table (IDT). If operand-size attribute is 32 bits, a 16-bit limit (lower two bytes of the 6-byte data operand) and a 32-bit base address (upper four bytes of the data operand) are loaded into the register. If the operand-size attribute is 16 bits, a 16-bit limit (lower two bytes) and a 24-bit base address (third, fourth, and fifth byte) are loaded. Here, the high-order byte of the operand is not used and the high-order byte of the base address in the GDTR or IDTR is filled with zeroes.

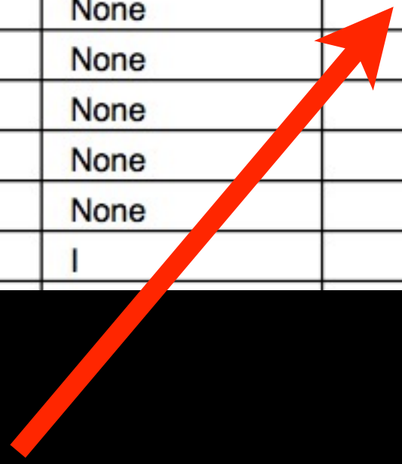
```
extern void bar();
void foo(void)
{
    bar(1,2,3,4,5,6);
}
```

```
# Linux x86-64
_foo:
        pushq   %rbp
        movq    %rsp, %rbp
        movl    $1, %edi
        movl    $2, %esi
        movl    $3, %edx
        movl    $4, %ecx
        movl    $5, %r8d
        movl    $6, %r9d
        popq    %rbp
        jmp     _bar  # TAILCALL
```

```
# Windows PE x86-64
_foo:
        subq    $56, %rsp
        movl    $1, %ecx
        movl    $2, %edx
        movl    $3, %r8d
        movl    $4, %r9d
        movl    $5, 32(%rsp)
        movl    $6, 40(%rsp)
        call    _bar
        nop
        addq    $56, %rsp
        ret
```

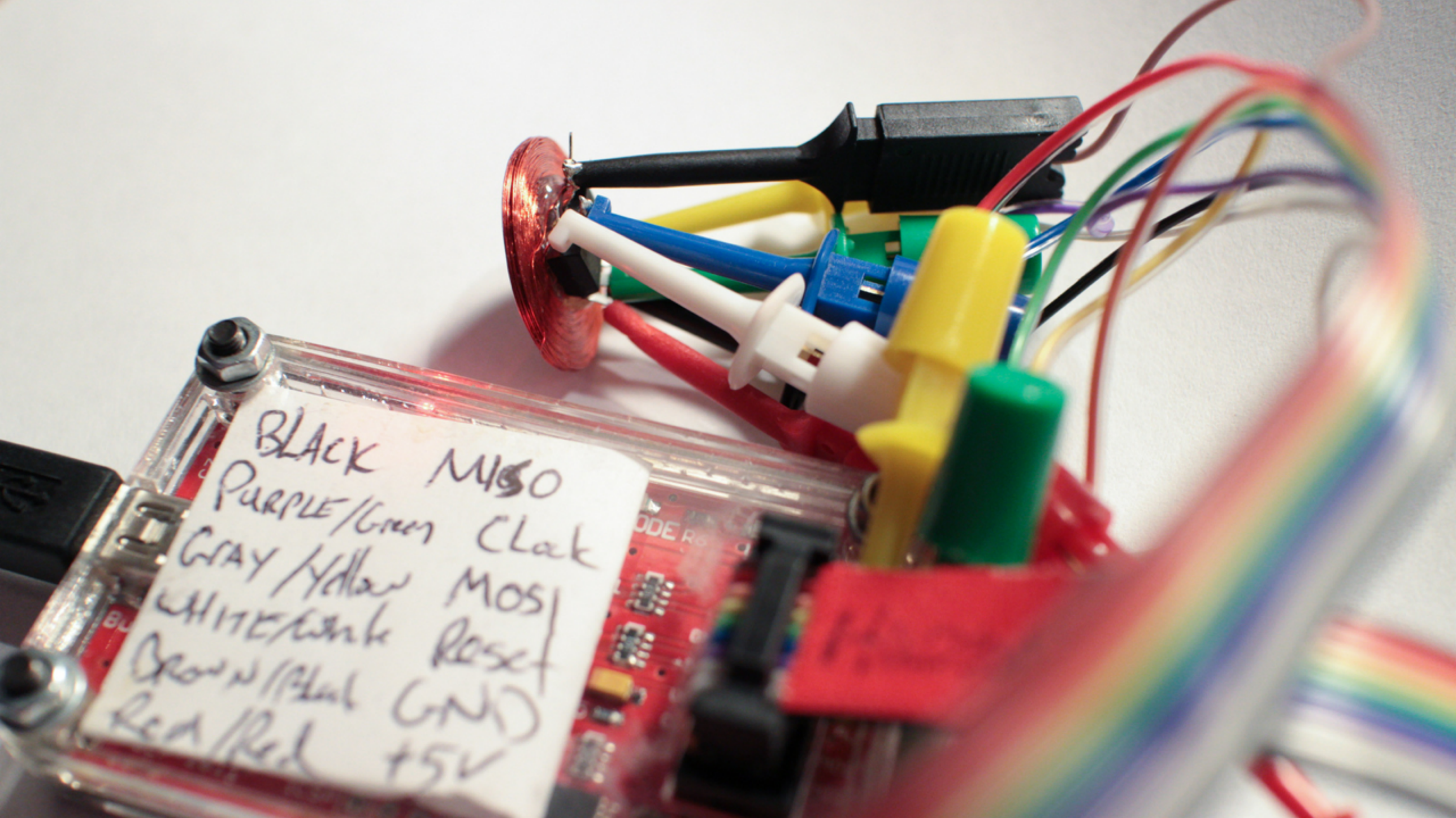
http://en.wikipedia.org/wiki/X86_calling_conventions

Relative Jump	PC ← PC + k + 1	None	2
Indirect Jump to (Z)	PC ← Z	None	2
Relative Subroutine Call	PC ← PC + k + 1	None	3
Indirect Call to (Z)	PC ← Z	None	3
Subroutine Return	PC ← STACK	None	4
Interrupt Return	PC ← STACK	I	4

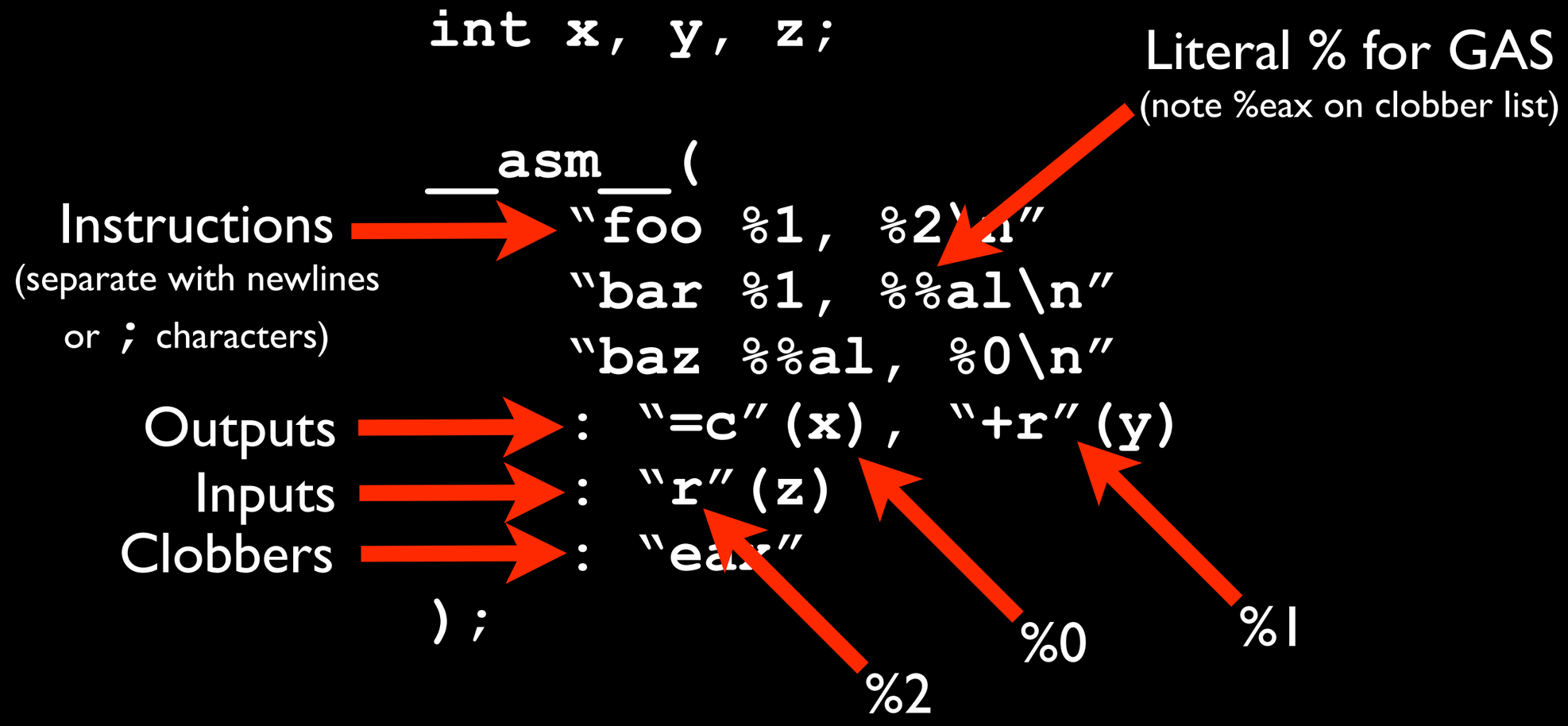


```
{
    if (clock_ticks & 1)
        asm("nop");

    switch(clock_ticks/2)
    {
    case 4: asm("rjmp .+1");
    case 3: asm("rjmp .+1");
    case 2: asm("rjmp .+1");
    case 1: asm("rjmp .+1");
    case 0: break;
    }
}
```

http://www.nycresistor.com/2012/12/27/rfid-multipass/


BLACK MISO
PURPLE/Green CLK
GRAY/Yellow MOSI
WHITE/Blue Reset
Brown/Black GND
Red/Red +5V

Enough talking!
Let’s write some code.


int x, y, z;
__asm__(
"foo %1, %2\n"
"bar %1, %%al\n"
"baz %%al, %0\n"
: "=c"(x), "+r"(y)
: "r"(z)
: "eax"
);
Literal % for GAS
(note %eax on clobber list)
Instructions
(separate with newlines
or ; characters)
Outputs
Inputs
Clobbers
%0
%1
%2

intel

Input in a register or memory

BSR—Bit Scan Reverse

Opcode	Instruction	Description
0F BD	BSR *r16,r/m16*	Bit scan reverse on *r/m16*
0F BD	BSR *r32,r/m32*	Bit scan reverse on *r/m32*

Output to a register

Actual bytes written to executable

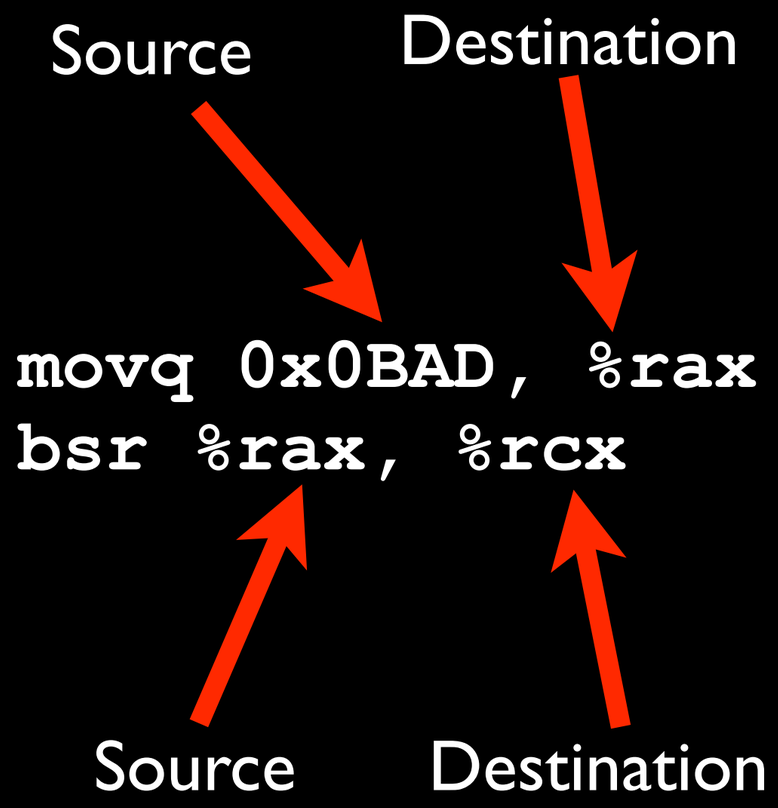
Description

This instruction searches the source operand (second operand) for the most significant set bit (1 bit). If a most significant 1 bit is found, its bit index is stored in the destination operand (first operand). The source operand can be a register or a memory location; the destination operand is a register. The bit index is an unsigned offset from bit 0 of the source operand. If the contents source operand are 0, the contents of the destination operand is undefined.

Operation

Pseudocode

```
IF SRC = 0
    THEN
        ZF ← 1;
        DEST is undefined;
    ELSE
        ZF ← 0;
        temp ← OperandSize – 1;
    WHILE Bit(SRC, temp) = 0
    DO
        temp ← temp – 1;
        DEST ← temp;
    OD;
FI;
```




```
%rax <- 0000101110101101
%rcx <- 12
```

Bit #12

```
static inline unsigned
log2i(
        uint64_t x
)
{
        uint64_t rc;
        asm(
            "bsr %1, %0"
            : "=r" (rc)
            : "r"  (x)
        );
        return rc
}
```

Nothing is clobbered, so it is not included.

Output(s)

Input(s)

http://gcc.gnu.org/onlinedocs/gcc/Simple-Constraints.html

```
“r”  Any general register
“i”  Immediate integer (compile time constant)
“F”  Floating point value (compile time)
“m”  Memory reference
“o”  Memory operand, near offset
“p”  Any valid memory address
```

intel

BSR—Bit Scan Reverse

Opcode	Instruction	Description
0F BD	BSR *r16,r/m16*	Bit scan reverse on *r/m16*
0F BD	BSR *r32,r/m32*	Bit scan reverse on *r/m32*

Description

This instruction searches the source operand (second operand) for the most significant set bit (1 bit). If a most significant 1 bit is found, its bit index is stored in the destination operand (first operand). The source operand can be a register or a memory location; the destination operand is a register. The bit index is an unsigned offset from bit 0 of the source operand. If the contents source operand are 0, the contents of the destination operand is undefined.

Undefined!

Operation

```
IF SRC = 0
    THEN
        ZF ← 1;
        DEST is undefined;
    ELSE
        ZF ← 0;
        temp ← OperandSize – 1;
    WHILE Bit(SRC, temp) = 0
    DO
        temp ← temp – 1;
        DEST ← temp;
    OD;
FI;
```

```
static inline unsigned
log2i(
        uint64_t x
)
{
        uint64_t rc = 0;
        if (x)
            asm(
                "bsr %1, %0"
                : "=r"(rc)
                : "r" (x)
            );
        return rc;
}
```

```
static inline unsigned
log2i(
        uint64_t x
)
{
        uint64_t rc = 0;
#ifdef __x86_64__
        if (x)
            asm(
                "bsr %1, %0"
                : "=r"(rc)
                : "r" (x)
            );
#else
        while (x >>= 1)
                rc++;
#endif
        return rc;
}
```

```
gcc -O3 -W -Wall -S log2i.c
```

```
gcc -O3 -W -Wall -c log2i.c
objdump -D log2i.o
```

```
<_log2i>:
   0:   55                    push   %rbp
   1:   31 c0                 xor    %eax,%eax
   3:   48 85 ff              test   %rdi,%rdi
   6:   48 89 e5              mov    %rsp,%rbp
   9:   74 04                 je     f <_log2i+0xf>
   b:   48 0f bd c7           bsr    %rdi,%rax
   f:   c9                    leaveq
  10:   c3                    retq
```

Offset (file)

Bytes

Instructions

http://gcc.gnu.org/onlinedocs/gcc/Other-Builtins.html

```
-- Built-in Function: int __builtin_clz (unsigned int x)
     Returns the number of leading 0-bits in X, starting at
the most significant bit position.  If X is 0, the result is
undefined.
```

```
unsigned
log2i(
        uint64_t x
)
{
        if (x)
                return 64 - __builtin_clzll(x);
        return 0;
}
```

intel

RDTSC—Read Time-Stamp Counter

Opcode	Instruction	Description
0F 31	RDTSC	Read time-stamp counter into EDX:EAX

Description

This instruction loads the current value of the processor's time-stamp counter into the EDX:EAX registers. The time-stamp counter is contained in a 64-bit MSR. The high-order 32 bits of the MSR are loaded into the EDX register, and the low-order 32 bits are loaded into the EAX register. The processor increments the time-stamp counter MSR every clock cycle and resets it to 0 whenever the processor is reset.

http://gcc.gnu.org/onlinedocs/gcc/Machine-Constraints.html

```
“a”   %al, %ax, %eax or %rax
“b”   %bl, %bx, %ebx or %rbx
“c”   %cl, %bx, %ecx or %rcx
“d”   %dl, %dx, %edx or %rdx
“q”   Any of a, b, c, d (in 32-bit mode)
“R”   Any of a, b, c, d, si, di, bp, sp
“I”   Integer 0..31
“J”   Integer 0..63
```

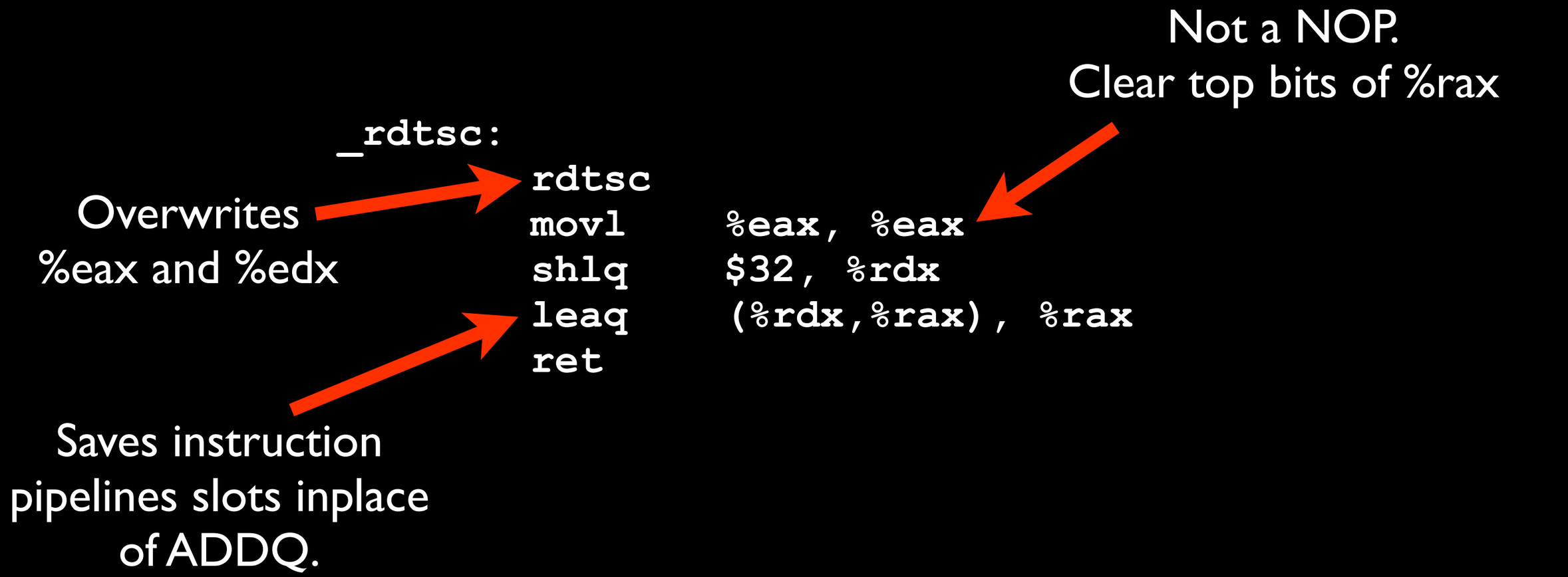
```
uint64_t
rdtsc(void)
{
        unsigned int tickl;
        unsigned int tickh;

        asm(
          "rdtsc"
          : "=a"(tickl), "=d"(tickh)
        );

        return ((uint64_t)tickh << 32) | tickl;
}
```

No inputs,
No clobbers,
Only outputs

tickl in %eax

tickh in %edx

shift exceeds 32 bit width


Not a NOP.
Clear top bits of %rax
_rdtsc:
rdtsc
movl %eax, %eax
shlq $32, %rdx
leaq (%rdx,%rax), %rax
ret
Overwrites
%eax and %edx
Saves instruction
pipelines slots inplace
of ADDQ.

CPUID—CPU Identification

Opcode	Instruction	Description
0F A2	CPUID	EAX ← Processor identification information

Description

This instruction provides processor iden
EDX. This information identifies Intel
processor, feature information, and ca
register determines what information is

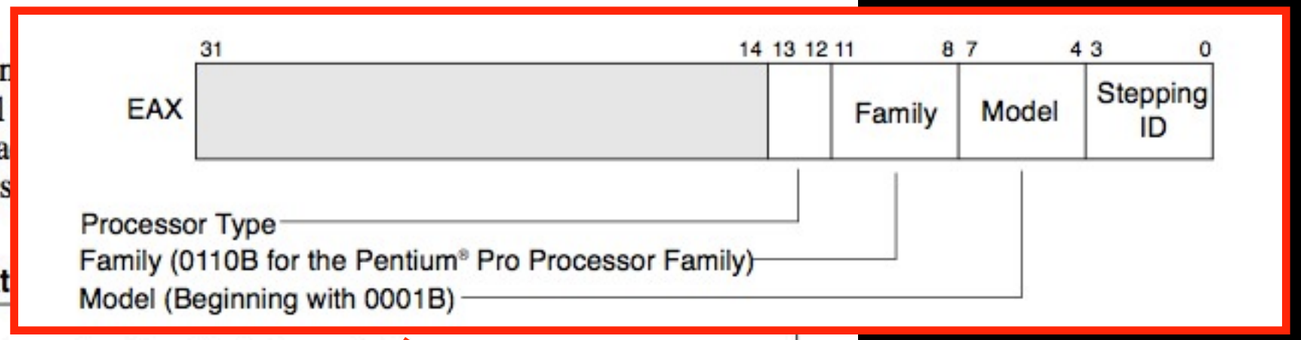
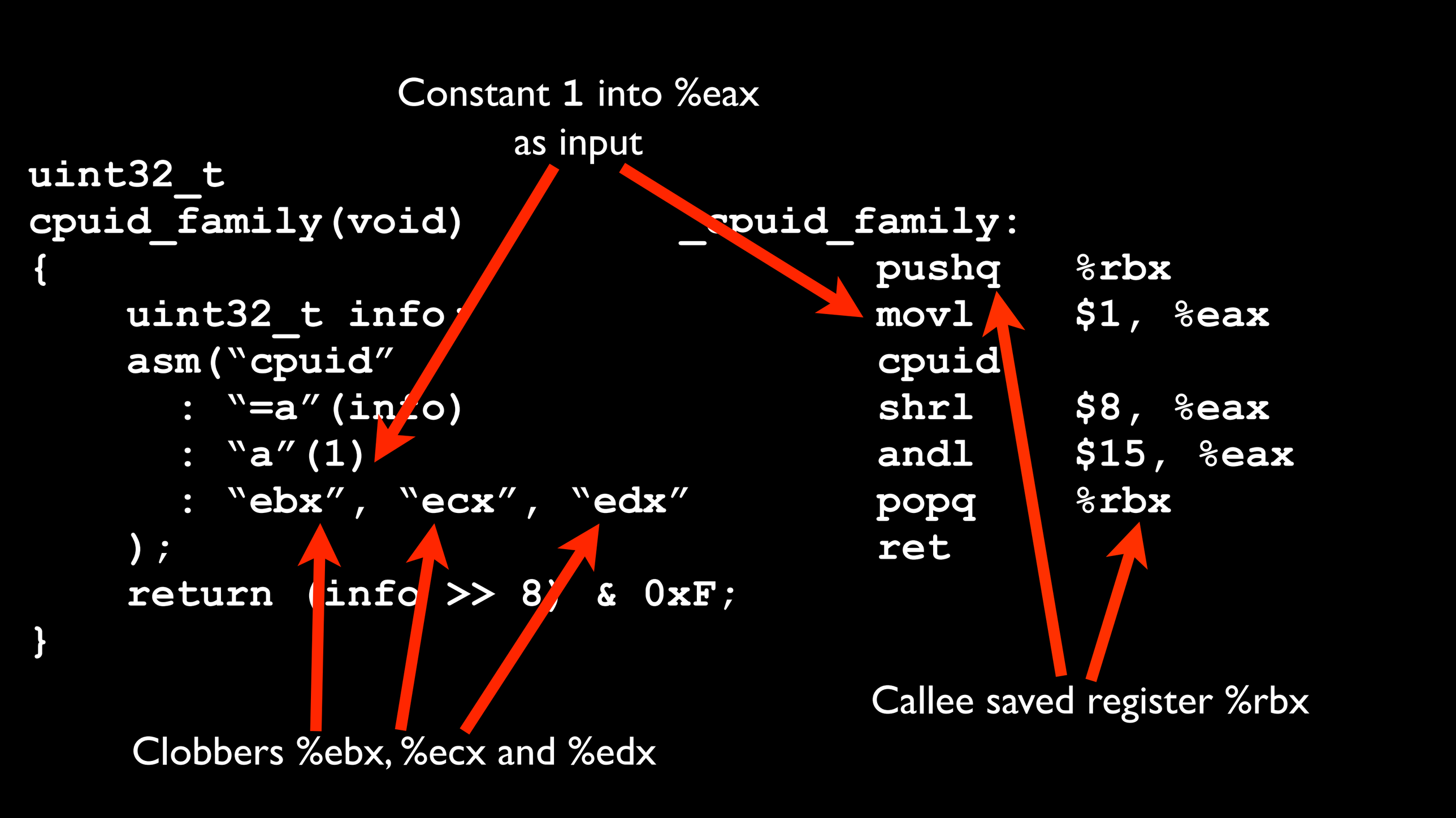


Table 3-6. Informat

Initial EAX Value	Information Provided about the rocessor	
0	EAX	Maximum CPUID Input Value (2 for the P family processors and 1 for the Pentium® processor and the later versi s of Intel486™ processor that support the CPUID instruction).
	EBX	"Genu"
	ECX	"ntel"
	EDX	"inel"
1	EAX	Version Information (Type, Family, Model, and Stepping ID)
	EBX	Reserved
	ECX	Reserved
	EDX	Feature Information
2	EAX	Cache and TLB Information
	EBX	Cache and TLB Information
	ECX	Cache and TLB Information
	EDX	Cache and TLB Information

Overwrites %ebx, %ecx and %edx


Constant 1 into %eax
as input
uint32_t
cpuid_family(void)
{
uint32_t info;
asm("cpuid"
: "=a"(info)
: "a"(1)
: "ebx", "ecx", "edx"
);
return (info >> 8) & 0xF;
}
_cpuid_family:
pushq %rbx
movl $1, %eax
cpuid
shrl $8, %eax
andl $15, %eax
popq %rbx
ret
Clobbers %ebx, %ecx and %edx
Callee saved register %rbx

```
extern void bar();
void foo(void)
{
    bar(1,2,3,4,5,6);
}
```

Six register arguments:
di, si, dx, cx, 8d, 9d
then spill onto the stack.
Callee saves bx, r12-r15
-fomit-frame-pointer

```
# Linux x86-64
_foo:
        movl    $1, %edi
        movl    $2, %esi
        movl    $3, %edx
        movl    $4, %ecx
        movl    $5, %r8d
        movl    $6, %r9d
        jmp     _bar
        # TAILCALL
```

Four register arguments:
cx, dx, 8d, 9d
then spill onto the stack.
Callee saves bx, di, si, r12-r15

```
# Windows PE x86-64
_foo:
        subq    $56, %rsp
        movl    $1, %ecx
        movl    $2, %edx
        movl    $3, %r8d
        movl    $4, %r9d
        movl    $5, 32(%rsp)
        movl    $6, 40(%rsp)
        call    _bar
        nop
        addq    $56, %rsp
        ret
```

All args use the stack.
Callee saves bx, di, si

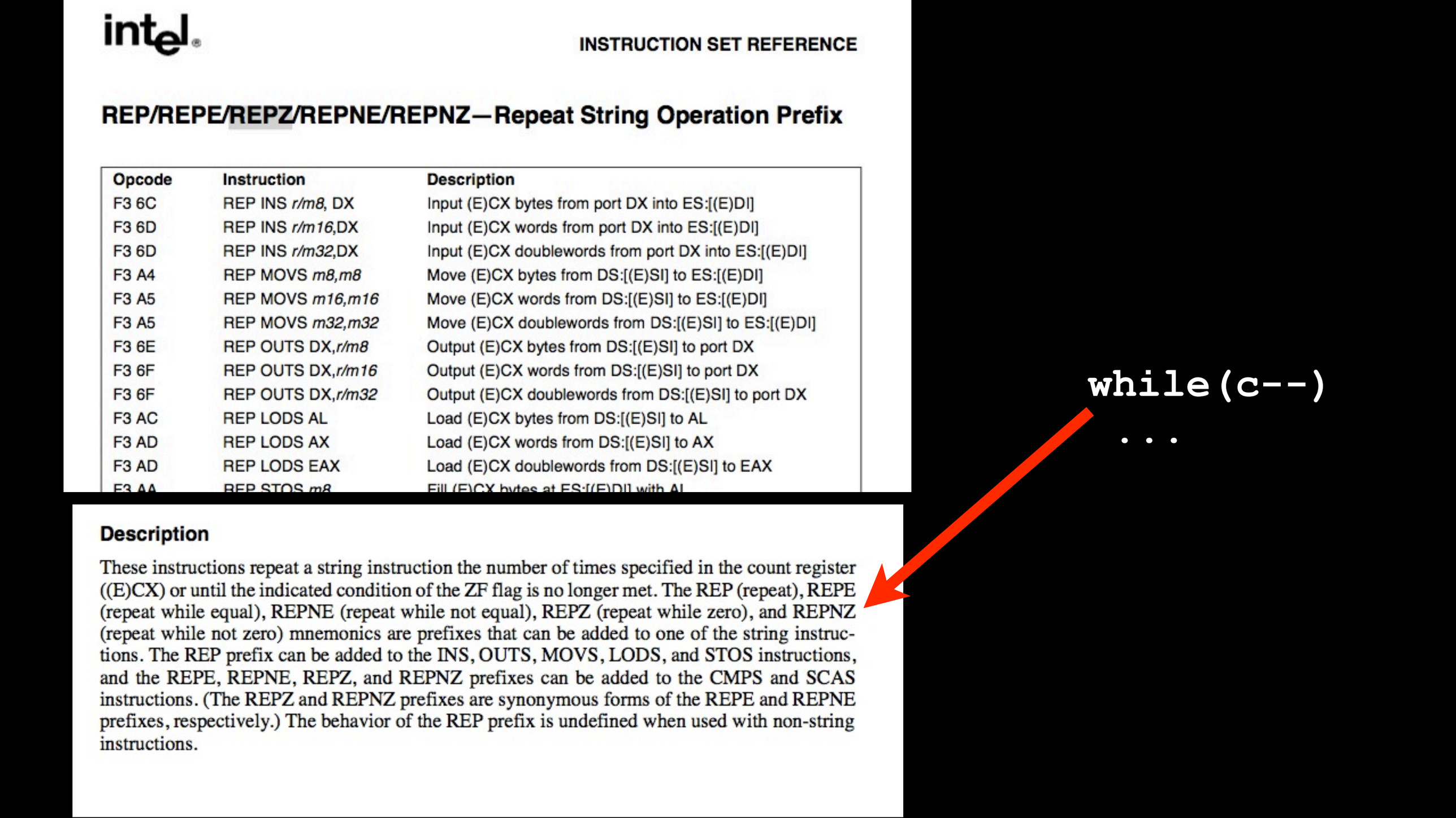
```
# i386 32-bit
_foo:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $40, %esp
        movl    $6, 20(%esp)
        movl    $5, 16(%esp)
        movl    $4, 12(%esp)
        movl    $3, 8(%esp)
        movl    $2, 4(%esp)
        movl    $1, (%esp)
        call    L_bar$stub
        leave
        ret
```

REP/REPE/REPZ/REPNE/REPNZ—Repeat String Operation Prefix

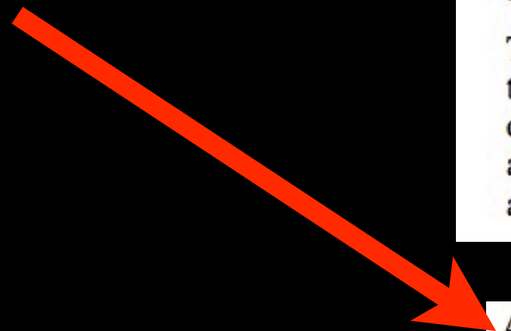
Opcode	Instruction	Description
F3 6C	REP INS *r/m8*, DX	Input (E)CX bytes from port DX into ES:[(E)DI]
F3 6D	REP INS *r/m16*,DX	Input (E)CX words from port DX into ES:[(E)DI]
F3 6D	REP INS *r/m32*,DX	Input (E)CX doublewords from port DX into ES:[(E)DI]
F3 A4	REP MOVS *m8,m8*	Move (E)CX bytes from DS:[(E)SI] to ES:[(E)DI]
F3 A5	REP MOVS *m16,m16*	Move (E)CX words from DS:[(E)SI] to ES:[(E)DI]
F3 A5	REP MOVS *m32,m32*	Move (E)CX doublewords from DS:[(E)SI] to ES:[(E)DI]
F3 6E	REP OUTS DX,*r/m8*	Output (E)CX bytes from DS:[(E)SI] to port DX
F3 6F	REP OUTS DX,*r/m16*	Output (E)CX words from DS:[(E)SI] to port DX
F3 6F	REP OUTS DX,*r/m32*	Output (E)CX doublewords from DS:[(E)SI] to port DX
F3 AC	REP LODS AL	Load (E)CX bytes from DS:[(E)SI] to AL
F3 AD	REP LODS AX	Load (E)CX words from DS:[(E)SI] to AX
F3 AD	REP LODS EAX	Load (E)CX doublewords from DS:[(E)SI] to EAX
F3 AA	REP STOS *m8*	Fill (E)CX bytes at ES:[(E)DI] with AL

Description

These instructions repeat a string instruction the number of times specified in the count register ((E)CX) or until the indicated condition of the ZF flag is no longer met. The REP (repeat), REPE (repeat while equal), REPNE (repeat while not equal), REPZ (repeat while zero), and REPNZ (repeat while not zero) mnemonics are prefixes that can be added to one of the string instructions. The REP prefix can be added to the INS, OUTS, MOVS, LODS, and STOS instructions, and the REPE, REPNE, REPZ, and REPNZ prefixes can be added to the CMPS and SCAS instructions. (The REPZ and REPNZ prefixes are synonymous forms of the REPE and REPNE prefixes, respectively.) The behavior of the REP prefix is undefined when used with non-string instructions.



```
*di++ = a;
```



intel

STOS/STOSB/STOSW/STOSD—Store String

Opcode	Instruction	Description
AA	STOS m8	Store AL at address ES:(E)DI
AB	STOS m16	Store AX at address ES:(E)DI
AB	STOS m32	Store EAX at address ES:(E)DI
AA	STOSB	Store AL at address ES:(E)DI
AB	STOSW	Store AX at address ES:(E)DI
AB	STOSD	Store EAX at address ES:(E)DI

Description

These instructions store a byte, word, or doubleword from the AL, AX, or EAX register, respectively, into the destination operand. The destination operand is a memory location, the address of which is read from either the ES:EDI or the ES:DI registers (depending on the address-size attribute of the instruction, 32 or 16, respectively). The ES segment cannot be overridden with a segment override prefix.

After the byte, word, or doubleword is transferred from the AL, AX, or EAX register to the memory location, the (E)DI register is incremented or decremented automatically according to the setting of the DF flag in the EFLAGS register. (If the DF flag is 0, the (E)DI register is incremented; if the DF flag is 1, the (E)DI register is decremented.) The (E)DI register is incremented or decremented by one for byte operations, by two for word operations, or by four for doubleword operations.

```
void
memset_slow(
     void * dest,
     uint8_t fill_value,
     uint64_t count
){
#ifdef __x86_64__
     asm(
        "cld;"
        "rep;"
        "stosq;"
     : /* no output registers */
     : "c" (count), "a" (fill_value), "D" (dest)
     : "ecx", "edi"
#else
     uint8_t * ptr = dest;
     while(count--)
       *ptr++ = fill_value;
#endif
}
```

Clear direction flag
stosq will increment
(not required in x86-64)

Clobbers -- unknown values

Now for something a little more complicated...

http://gcc.gnu.org/onlinedocs/gcc/Extended-Asm.html

```
void
memcpy_slow(
        void * dest,
        const void * src,
        uint64_t count
) {
#ifdef __x86_64__
        uint64_t i = 0;
        __asm__ __volatile__(
                "1:\n"
                "movb (%[src],%[i]), %%al;"
                "movb %%al, (%[dest],%[i]);"
                "inc %[i];"
                "loop 1b;"
                : [i]"+r"(i), "+c"(count)
                : [dest]"r"(dest), [src]"r"(src)
                : "eax", "ecx", "memory"
        );
#else
        uint8_t * a = dest;
        const uint8_t * b = src;

        while (count--)
                *a++ = *b++;
#endif
}
```

if (--cx != 0)
goto label 1

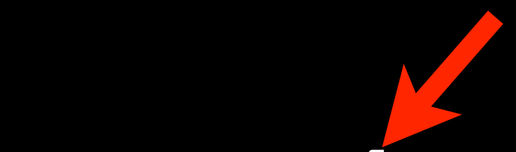
Temp register a must be in clobber list

Count must be in cx since loop uses it.

Named args instead of positional args.

Tell gcc that memory has been modified

Why memcpy_slow() ?



```
<_memcpy_slow>:
   0:   55                      push   %rbp
   1:   48 89 e5                mov    %rsp,%rbp
   4:   48 89 d1                mov    %rdx,%rcx
   7:   31 d2                   xor    %edx,%edx
   9:   8a 04 16                mov    (%rsi,%rdx,1),%al
   c:   88 04 17                mov    %al,(%rdi,%rdx,1)
   f:   48 ff c2                inc    %rdx
  12:   e2 f5                   loop   9 <_memcpy_slow+0x9>
  14:   5d                      pop    %rbp
  15:   c3                      retq
```

```
 #define kShort     255                 // for nonzero memset(), too short for
commpage


        .text
        .globl _memset
        .align 2
_memset:                                // void *memset(void *b, int c, size_t len);
        movl   8(%esp),%eax             // get 1-byte pattern
        movl   12(%esp),%edx            // get length
        andl   $0xFF,%eax               // (c==0) ?
        jnz    LNonzero                 // not a bzero

        movl   %edx,8(%esp)             // put count where bzero() expects it
        jmp           _bzero                    // enter _bzero


        // Handle memset of a nonzero value.

LNonzero:
        pushl  %edi                     // save a few nonvolatiles
        pushl  %esi
        movl   %eax,%esi                // replicate byte in %al into all bytes
        movl   12(%esp),%edi            // point to operand
        shll   $8,%esi
        orl    %esi,%eax
        movl   %eax,%esi
        shll   $16,%esi
        orl    %esi,%eax                // now %eax has "c" in all 4 bytes
        cmpl   $(kShort),%edx                 // is operand too short for SSE?
        ja     LCallCommpage            // no

// Nonzero memset() too short to call commpage.
//      %eax = replicated 4-byte pattern
//      %edi = ptr
//      %edx = length (<= kShort)

        cmpl   $16,%edx                 // long enough to word align?
        jge    3f                       // yes
        test   %edx,%edx                // length==0?
        jz     6f
1:
        movb   %al,(%edi)               // pack in a byte
        inc    %edi
        dec    %edx
        jnz    1b
        jmp    6f
2:
        movb   %al,(%edi)               // pack in a byte
        inc    %edi
        dec    %edx
3:
        test   $3,%edi                        // is ptr doubleword aligned?
        jnz    2b                       // no
        movl   %edx,%ecx                // copy length
        shrl   $2,%edx                        // #doublewords to store
4:
        movl   %eax,(%edi)              // store aligned doubleword
        addl   $4,%edi
        dec    %edx
        jnz    4b
        andl   $3,%ecx                        // any leftover bytes?
        jz     6f                       // no
5:
        movb   %al,(%edi)               // pack in a byte
        inc    %edi
        dec    %ecx
        jnz    5b
6:
```

```
// Handle memset of a 16-byte pattern.

        .globl _memset_pattern16
        .align 2, 0x90
_memset_pattern16:                      // void memset_pattern16(void *b, const void *c16,
size_t len);
        pushl  %edi
        pushl  %esi
        movl   20(%esp),%edx            // get length
        movl   16(%esp),%esi            // get ptr to 16-byte pattern
        movl   12(%esp),%edi            // point to operand
        movdqu (%esi),%xmm0             // load the pattern
        jmp    LAlignPtr

// Handle memset of an 8-byte pattern.

        .globl _memset_pattern8
        .align 2, 0x90
_memset_pattern8:                       // void memset_pattern8(void *b, const void *c8,
size_t len);
        pushl  %edi
        pushl  %esi
        movl   20(%esp),%edx            // get length
        movl   16(%esp),%esi            // get ptr to 8-byte pattern
        movl   12(%esp),%edi            // point to operand
        movq   (%esi),%xmm0             // load pattern into low 8 bytes
        punpcklqdq %xmm0,%xmm0                  // replicate into all 16
        jmp    LAlignPtr

// Handle memset of a 4-byte pattern.

        .globl _memset_pattern4
        .align 2, 0x90
_memset_pattern4:                       // void memset_pattern4(void *b, const void *c4,
size_t len);
        pushl  %edi
        pushl  %esi
        movl   20(%esp),%edx            // get length
        movl   16(%esp),%esi            // get ptr to 4-byte pattern
        movl   12(%esp),%edi            // point to operand
        movd   (%esi),%xmm0             // load pattern into low 4 bytes
        pshufd $0x0,%xmm0,%xmm0 // replicate the 4 bytes across the vector


// Align ptr if necessary.  We must rotate the pattern right for each byte we
// store while aligning the ptr.  Since there is no rotate instruction in SSE3,
// we have to synthesize the rotates.
//      %edi = ptr
//      %edx = length
//      %xmm0 = pattern

LAlignPtr:                              // NB: can drop down to here!
        cmpl   $100,%edx                // long enough to bother aligning ptr?
        movl   %edi,%ecx                // copy ptr
        jb     LReady                   // not long enough
        negl   %ecx
        andl   $15,%ecx                 // get #bytes to align ptr
        jz     LReady                   // already aligned
        subl   %ecx,%edx                // adjust length

        test   $1,%cl                   // 1-byte store required?
        movd   %xmm0,%eax               // get 4 low bytes in %eax
        jz     2f                       // no
        movdqa %xmm0,%xmm1              // copy pattern so we can shift in both directions
        movb   %al,(%edi)               // pack in the low-order byte
        psrldq $1,%xmm0                 // shift pattern right 1 byte
        inc    %edi
        pslldq $15,%xmm1                // shift pattern left 15 bytes
```

This is what memcpy() really looks like!

Things memcpy() needs to worry about:

* Inlining *
* Alignment *
* Prefetching *
* Loop unrolling *
* Cache pollution *
* Bulk data transfers *

There is no doubt that the grail of efficiency leads to abuse. Programmers waste enormous amounts of time thinking about, or worrying about, the speed of noncritical parts of their programs, and these attempts at efficiency actually have a strong negative impact when debugging and maintenance are considered. We *should* forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil.

Knuth, 1974 "Structured Programming With goto statements"

The First Law of Program Optimization: Don’t do it

Yet we should not pass up our opportunities in that critical 3%. A good programmer will not be lulled into complacency by such reasoning, he will be wise to look carefully at the critical code; but only *after* that code has been identified. It is often a mistake to make a priori judgments about what parts of a program are really critical, since the universal experience of programmers who have been using measurement tools has been that their intuitive guesses fail. After work-

Knuth, 1974 "Structured Programming With goto statements"

The Second Law of Program Optimization: Don’t do it, yet

(For experts only)

What else, other than profiling?

Keep code simple.

```
double
prod(
   const double * a,
   const double * b,
   const unsigned n
)
{
  double p = 0;
  unsigned i;
  for (i = 0 ; i < n ; i++)
     p += a[i] * b[i];

  return p;
}
```

```
double
prod_hand_unrolled(
   const double * a,
   const double * b,
   const unsigned n
)
{
   double p = 0;
   unsigned i;
   for (i = 0 ; i < n ; i += 4)
   {
     p += a[i+0] * b[i+0];
     p += a[i+1] * b[i+1];
     p += a[i+2] * b[i+2];
     p += a[i+3] * b[i+3];
   }

   return p;
}
```

```
gcc-mp-4.5 \
  -ftree-vectorizer-verbose=2 \
  -ftree-vectorize \
  -ffast-math \
  -funroll-loops \
  -msse2 \
  -m64 \
  -std=gnu99 \
  -O3 -W -Wall \
  -c unroll.c

unroll.c:9: note: LOOP VECTORIZED.
unroll.c:2: note: vectorized 1 loops in function.

unroll.c:31: note: vectorized 0 loops in function.
```

```
 0:    push   %rbp
 1:    test   %edx,%edx
 3:    xorpd  %xmm0,%xmm0
 7:    mov    %rsp,%rbp
 a:    je     4ff <_prod1+0x4ff>
10:    mov    %edx,%r10d
13:    shr    %r10d
16:    cmp    $0x4,%edx
19:    lea    (%r10,%r10,1),%r9d
1d:    jbe    501 <_prod1+0x501>
23:    test   %r9d,%r9d
26:    je     501 <_prod1+0x501>
2c:    movsd  (%rsi),%xmm1
30:    lea    -0x1(%r10),%r8d
34:    movsd  (%rdi),%xmm2
38:    mov    $0x1,%ecx
3d:    movhpd 0x8(%rsi),%xmm1
42:    and    $0x7,%r8d
46:    cmp    $0x1,%r10d
4a:    movhpd 0x8(%rdi),%xmm2
4f:    mov    $0x10,%eax
54:    mulpd  %xmm2,%xmm1
58:    movapd %xmm1,%xmm0
5c:    jbe    34f <_prod1+0x34f>
62:    test   %r8d,%r8d
65:    je     1e1 <_prod1+0x1e1>
6b:    cmp    $0x1,%r8d
6f:    je     1aa <_prod1+0x1aa>
75:    cmp    $0x2,%r8d
79:    je     17a <_prod1+0x17a>
7f:    cmp    $0x3,%r8d
83:    je     14e <_prod1+0x14e>
89:    cmp    $0x4,%r8d
8d:    je     125 <_prod1+0x125>
93:    cmp    $0x5,%r8d
97:    je     f5 <_prod1+0xf5>
99:    cmp    $0x6,%r8d
9d:    je     c5 <_prod1+0xc5>
9f:    movsd  0x10(%rsi),%xmm0
a4:    mov    $0x2,%ecx
a9:    movsd  0x10(%rdi),%xmm7
ae:    mov    $0x20,%eax
b3:    movhpd 0x18(%rsi),%xmm0
b8:    movhpd 0x18(%rdi),%xmm7
bd:    mulpd  %xmm7,%xmm0
c1:    addpd  %xmm1,%xmm0
c5:    movsd  (%rsi,%rax,1),%xmm11
cb:    add    $0x1,%ecx
ce:    movsd  (%rdi,%rax,1),%xmm10
d4:    movhpd 0x8(%rsi,%rax,1),%xmm11
db:    movhpd 0x8(%rdi,%rax,1),%xmm10
e2:    add    $0x10,%rax
e6:    movapd %xmm11,%xmm8
eb:    mulpd  %xmm10,%xmm8
f0:    addpd  %xmm8,%xmm0
f5:    movsd  (%rsi,%rax,1),%xmm15
fb:    add    $0x1,%ecx
fe:    movsd  (%rdi,%rax,1),%xmm14
104:   movhpd 0x8(%rsi,%rax,1),%xmm15
10b:   movhpd 0x8(%rdi,%rax,1),%xmm14
112:   add    $0x10,%rax
116:   movapd %xmm15,%xmm12
11b:   mulpd  %xmm14,%xmm12
120:   addpd  %xmm12,%xmm0
125:   movsd  (%rsi,%rax,1),%xmm5
12a:   add    $0x1,%ecx
12d:   movsd  (%rdi,%rax,1),%xmm4
132:   movhpd 0x8(%rsi,%rax,1),%xmm5
138:   movhpd 0x8(%rdi,%rax,1),%xmm4
13e:   add    $0x10,%rax
142:   movapd %xmm5,%xmm1
146:   mulpd  %xmm4,%xmm1
14a:   addpd  %xmm1,%xmm0
14e:   movsd  (%rsi,%rax,1),%xmm8
154:   add    $0x1,%ecx
157:   movsd  (%rdi,%rax,1),%xmm7
15c:   movhpd 0x8(%rsi,%rax,1),%xmm8
163:   movhpd 0x8(%rdi,%rax,1),%xmm7
169:   add    $0x10,%rax
16d:   movapd %xmm8,%xmm6
172:   mulpd  %xmm7,%xmm6
176:   addpd  %xmm6,%xmm0
17a:   movsd  (%rsi,%rax,1),%xmm12
180:   add    $0x1,%ecx
183:   movsd  (%rdi,%rax,1),%xmm11
189:   movhpd 0x8(%rsi,%rax,1),%xmm12
190:   movhpd 0x8(%rdi,%rax,1),%xmm11
197:   add    $0x10,%rax
19b:   movapd %xmm12,%xmm9
1a0:   mulpd  %xmm11,%xmm9
1a5:   addpd  %xmm9,%xmm0
1aa:   movsd  (%rsi,%rax,1),%xmm1
1af:   add    $0x1,%ecx
1b2:   movsd  (%rdi,%rax,1),%xmm15
1b8:   movhpd 0x8(%rsi,%rax,1),%xmm1
1be:   movhpd 0x8(%rdi,%rax,1),%xmm15
1c5:   add    $0x10,%rax
1c9:   cmp    %ecx,%r10d
1cc:   movapd %xmm1,%xmm13
1d1:   mulpd  %xmm15,%xmm13
1d6:   addpd  %xmm13,%xmm0
1db:   jbe    34f <_prod1+0x34f>
1e1:   movsd  (%rsi,%rax,1),%xmm5
1e6:   lea    0x10(%rax),%r8
1ea:   movsd  (%rdi,%rax,1),%xmm4
1ef:   lea    0x20(%rax),%r11
1f3:   movhpd 0x8(%rsi,%rax,1),%xmm5
1f9:   movsd  0x10(%rsi,%rax,1),%xmm15
200:   movhpd 0x8(%rdi,%rax,1),%xmm4
206:   movsd  0x10(%rdi,%rax,1),%xmm14
20d:   movapd %xmm5,%xmm1
211:   movhpd 0x8(%rsi,%r8,1),%xmm15
218:   movhpd 0x8(%rdi,%r8,1),%xmm14
21f:   movsd  0x20(%rsi,%rax,1),%xmm11
226:   mulpd  %xmm4,%xmm1
22a:   lea    0x30(%rax),%r8
22e:   movapd %xmm15,%xmm12
233:   add    $0x8,%ecx
236:   movhpd 0x8(%rsi,%r11,1),%xmm11
23d:   movsd  0x20(%rdi,%rax,1),%xmm10
244:   mulpd  %xmm14,%xmm12
249:   movhpd 0x8(%rdi,%r11,1),%xmm10
250:   lea    0x40(%rax),%r11
254:   addpd  %xmm1,%xmm0
258:   movapd %xmm11,%xmm8
25d:   movsd  0x30(%rsi,%rax,1),%xmm7
263:   mulpd  %xmm10,%xmm8
268:   movhpd 0x8(%rsi,%r8,1),%xmm7
26f:   movsd  0x30(%rdi,%rax,1),%xmm6
275:   addpd  %xmm12,%xmm0
27a:   movapd %xmm7,%xmm5
27e:   movhpd 0x8(%rdi,%r8,1),%xmm6
285:   lea    0x50(%rax),%r8
289:   movsd  0x40(%rsi,%rax,1),%xmm4
28f:   mulpd  %xmm6,%xmm5
293:   movhpd 0x8(%rsi,%r11,1),%xmm4
29a:   addpd  %xmm8,%xmm0
29f:   movsd  0x40(%rdi,%rax,1),%xmm1
2a5:   movapd %xmm4,%xmm15
2aa:   movhpd 0x8(%rdi,%r11,1),%xmm1
2b1:   movsd  0x50(%rsi,%rax,1),%xmm14
2b8:   movsd  0x50(%rdi,%rax,1),%xmm13
2bf:   lea    0x60(%rax),%r11
2c3:   mulpd  %xmm1,%xmm15
2c8:   movhpd 0x8(%rsi,%r8,1),%xmm14
2cf:   addpd  %xmm5,%xmm0
2d3:   movhpd 0x8(%rdi,%r8,1),%xmm13
2da:   lea    0x70(%rax),%r8
2de:   movapd %xmm14,%xmm11
2e3:   movsd  0x60(%rsi,%rax,1),%xmm10
2ea:   movsd  0x60(%rdi,%rax,1),%xmm9
2f1:   mulpd  %xmm13,%xmm11
2f6:   movhpd 0x8(%rsi,%r11,1),%xmm10
2fd:   addpd  %xmm15,%xmm0
302:   movapd %xmm10,%xmm7
307:   movhpd 0x8(%rdi,%r11,1),%xmm9
30e:   movsd  0x70(%rsi,%rax,1),%xmm6
314:   movsd  0x70(%rdi,%rax,1),%xmm5
31a:   sub    $0xffffffffffffff80,%rax
31e:   mulpd  %xmm9,%xmm7
323:   movhpd 0x8(%rsi,%r8,1),%xmm6
32a:   addpd  %xmm11,%xmm0
32f:   cmp    %ecx,%r10d
332:   movapd %xmm6,%xmm4
336:   movhpd 0x8(%rdi,%r8,1),%xmm5
33d:   mulpd  %xmm5,%xmm4
341:   addpd  %xmm7,%xmm0
345:   addpd  %xmm4,%xmm0
349:   ja     1e1 <_prod1+0x1e1>
34f:   cmp    %r9d,%edx
352:   haddpd %xmm0,%xmm0
356:   je     4ff <_prod1+0x4ff>
35c:   mov    %r9d,%eax
35f:   mov    %r9d,%ecx
362:   movsd  (%rsi,%rax,8),%xmm3
367:   not    %ecx
369:   add    %edx,%ecx
36b:   mulsd  (%rdi,%rax,8),%xmm3
370:   lea    0x1(%r9),%eax
374:   and    $0x7,%ecx
377:   cmp    %eax,%edx
379:   addsd  %xmm3,%xmm0
37d:   jbe    4ff <_prod1+0x4ff>
383:   test   %ecx,%ecx
385:   je     453 <_prod1+0x453>
38b:   cmp    $0x1,%ecx
38e:   je     435 <_prod1+0x435>
394:   cmp    $0x2,%ecx
397:   je     41f <_prod1+0x41f>
39d:   cmp    $0x3,%ecx
3a0:   je     409 <_prod1+0x409>
3a2:   cmp    $0x4,%ecx
3a5:   je     3f6 <_prod1+0x3f6>
3a7:   cmp    $0x5,%ecx
3aa:   je     3e0 <_prod1+0x3e0>
3ac:   cmp    $0x6,%ecx
3af:   nop
3b0:   je     3c9 <_prod1+0x3c9>
3b2:   mov    %eax,%eax
3b4:   movsd  (%rsi,%rax,8),%xmm14
3ba:   mulsd  (%rdi,%rax,8),%xmm14
3c0:   lea    0x2(%r9),%eax
3c4:   addsd  %xmm14,%xmm0
3c9:   mov    %eax,%r11d
3cc:   add    $0x1,%eax
3cf:   movsd  (%rsi,%r11,8),%xmm15
3d5:   mulsd  (%rdi,%r11,8),%xmm15
3db:   addsd  %xmm15,%xmm0
3e0:   mov    %eax,%r8d
3e3:   add    $0x1,%eax
3e6:   movsd  (%rsi,%r8,8),%xmm1
3ec:   mulsd  (%rdi,%r8,8),%xmm1
3f2:   addsd  %xmm1,%xmm0
3f6:   mov    %eax,%ecx
3f8:   add    $0x1,%eax
3fb:   movsd  (%rsi,%rcx,8),%xmm2
400:   mulsd  (%rdi,%rcx,8),%xmm2
405:   addsd  %xmm2,%xmm0
409:   mov    %eax,%r9d
40c:   add    $0x1,%eax
40f:   movsd  (%rsi,%r9,8),%xmm4
415:   mulsd  (%rdi,%r9,8),%xmm4
41b:   addsd  %xmm4,%xmm0
41f:   mov    %eax,%r10d
422:   add    $0x1,%eax
425:   movsd  (%rsi,%r10,8),%xmm5
42b:   mulsd  (%rdi,%r10,8),%xmm5
431:   addsd  %xmm5,%xmm0
435:   mov    %eax,%r11d
438:   add    $0x1,%eax
43b:   movsd  (%rsi,%r11,8),%xmm3
441:   cmp    %eax,%edx
443:   mulsd  (%rdi,%r11,8),%xmm3
449:   addsd  %xmm3,%xmm0
44d:   jbe    4ff <_prod1+0x4ff>
453:   mov    %eax,%r10d
456:   lea    0x1(%rax),%r9d
45a:   lea    0x2(%rax),%ecx
45d:   movsd  (%rsi,%r10,8),%xmm13
463:   lea    0x3(%rax),%r8d
467:   movsd  (%rsi,%r9,8),%xmm12
46d:   lea    0x4(%rax),%r11d
471:   mulsd  (%rdi,%r10,8),%xmm13
477:   lea    0x5(%rax),%r10d
47b:   mulsd  (%rdi,%r9,8),%xmm12
481:   lea    0x6(%rax),%r9d
485:   movsd  (%rsi,%rcx,8),%xmm11
48b:   movsd  (%rsi,%r8,8),%xmm10
491:   mulsd  (%rdi,%rcx,8),%xmm11
497:   lea    0x7(%rax),%ecx
49a:   add    $0x8,%eax
49d:   mulsd  (%rdi,%r8,8),%xmm10
4a3:   cmp    %eax,%edx
4a5:   addsd  %xmm13,%xmm0
4aa:   movsd  (%rsi,%r11,8),%xmm9
4b0:   movsd  (%rsi,%r10,8),%xmm8
4b6:   mulsd  (%rdi,%r11,8),%xmm9
4bc:   mulsd  (%rdi,%r10,8),%xmm8
4c2:   addsd  %xmm12,%xmm0
4c7:   movsd  (%rsi,%r9,8),%xmm7
4cd:   movsd  (%rsi,%rcx,8),%xmm6
4d2:   mulsd  (%rdi,%r9,8),%xmm7
4d8:   mulsd  (%rdi,%rcx,8),%xmm6
4dd:   addsd  %xmm11,%xmm0
4e2:   addsd  %xmm10,%xmm0
4e7:   addsd  %xmm9,%xmm0
4ec:   addsd  %xmm8,%xmm0
4f1:   addsd  %xmm7,%xmm0
4f5:   addsd  %xmm6,%xmm0
4f9:   ja     453 <_prod1+0x453>
4ff:   leaveq
500:   retq
501:   xor    %r9d,%r9d
504:   xorpd  %xmm0,%xmm0
508:   jmpq   35c <_prod1+0x35c>
50d:   nopl   (%rax)
```

Are you going to write that by hand?

For more than one architecture?

With a test suite?

And benchmark it?

http://ridiculousfish.com/blog/posts/will-it-optimize.html

Will it optimize?
(gcc edition)

```
unsigned uhalve_it(unsigned x)
{
   return x / 2;
}

int halve_it(int x)
{
   return x / 2;
}

int double_it(int x)
{
   return x * 2;
}

int ten_it(int x)
{
   return x * 10;
}
```

```
_uhalve_it:
        movl    %edi, %eax
        shrl    %eax
        ret

_halve_it:
        movl    %edi, %eax
        shrl    $31, %eax
        addl    %edi, %eax
        sarl    %eax
        ret

_double_it:
        leal    (%rdi,%rdi), %eax
        ret

_ten_it:
        leal    (%rdi,%rdi,4), %edi
        leal    (%rdi,%rdi), %eax
        ret
```

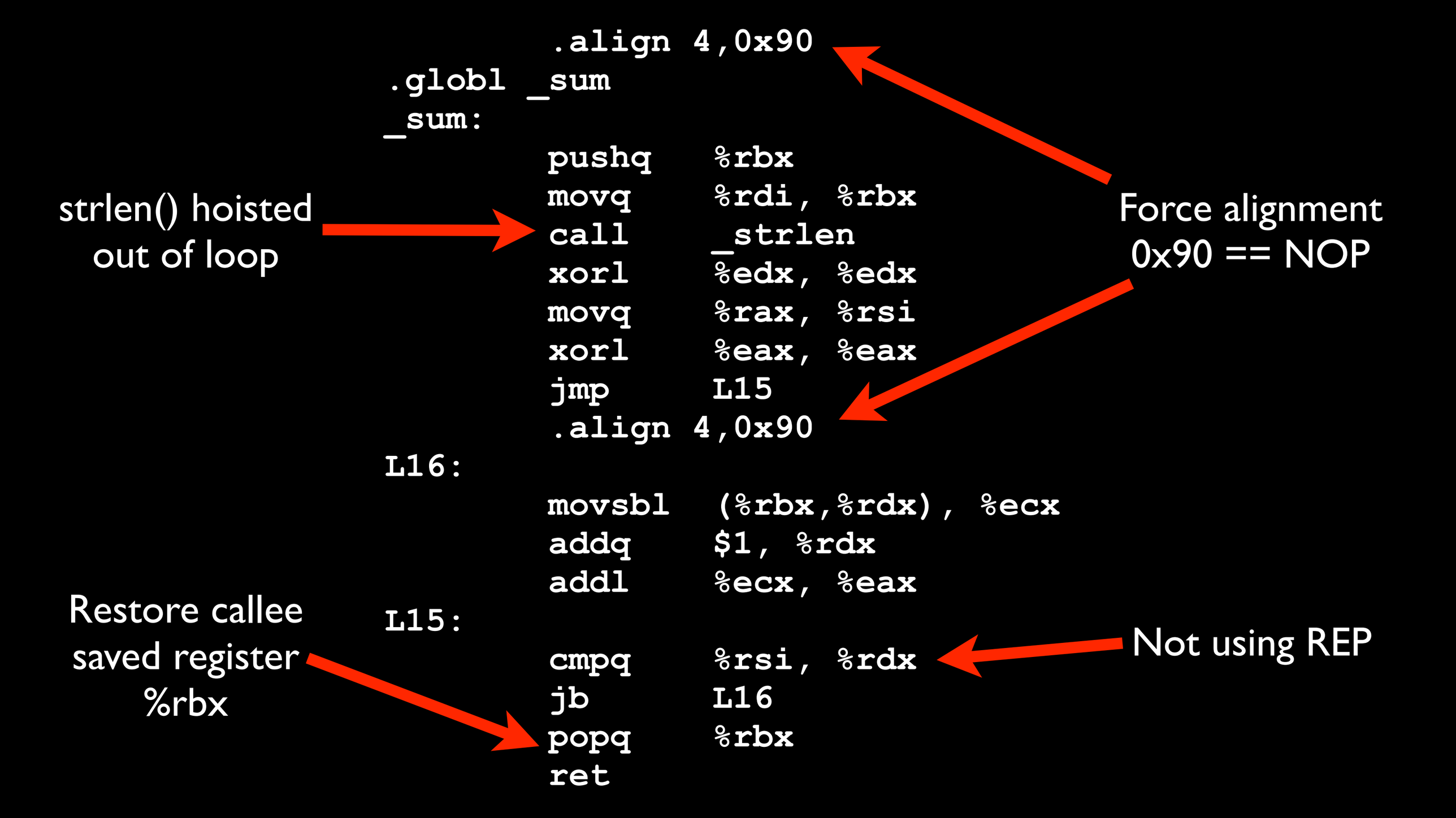
```
int decimate_it(int x)
{
   return x / 10;
}
```

```
_decimate_it:
        movl    %edi, %eax
        movl    0x66666667, %edx
        sarl    $31, %edi
        imull   %edx
        sarl    $2, %edx
        subl    %edi, %edx
        movl    %edx, %eax
        ret
```

http://www.jjj.de/fxt/ or “Hacker’s Delight” by Henry Warren for proof of correctness.

```
unsigned sum(
   const unsigned char *s
)
{
   unsigned result = 0;
   size_t i;
   for (i=0; i < strlen(s); i++)
      result += s[i];
   return result;
}
```

```
gcc-mp-4.5 \
 -fomit-frame-pointer \
 -O3 \
 -S \
 sum.c
```


.align 4,0x90
.globl _sum
_sum:
pushq %rbx
movq %rdi, %rbx
call _strlen
xorl %edx, %edx
movq %rax, %rsi
xorl %eax, %eax
jmp L15
.align 4,0x90
L16:
movsbl (%rbx,%rdx), %ecx
addq $1, %rdx
addl %ecx, %eax
L15:
cmpq %rsi, %rdx
jb L16
popq %rbx
ret
strlen() hoisted out of loop
Force alignment 0x90 == NOP
Restore callee saved register %rbx
Not using REP

```
int factorial(int x)
{
   if (x > 1)
     return x * factorial(x-1);
   else
     return 1;
}
```

```
gcc-mp-4.5 \
 -fomit-frame-pointer \
 -O1 \
 -S \
 factorial.c
```

```
_factorial:
        pushq   %rbx
        movl    %edi, %ebx
        movl    $1, %eax
        cmpl    $1, %edi
        jle     L2
        leal    -1(%rbx), %edi
        call    _factorial
        imull   %ebx, %eax
L2:
        popq    %rbx
        ret
```

```
int factorial(int x)
{
   if (x > 1)
     return x * factorial(x-1);
   else
     return 1;
}
```

```
gcc-mp-4.5 \
 -fomit-frame-pointer \
 -O3 \
 -S \
 factorial.c
```

Now with optimization!

```
_factorial:
    cmpl    $1, %edi
    movl    $1, %eax
    jle     L2
    leal    -1(%rdi), %ecx
    movl    %ecx, %edx
    shrl    $2, %edx
    cmpl    $6, %ecx
    leal    0(,%rdx,4), %esi
    jbe     L8
    testl    %esi, %esi
    je      L8
    leal    -2(%rdi), %eax
    movl    %edi, -12(%rsp)
    movdqa  LC1(%rip), %xmm4
    movl    %eax, -8(%rsp)
    leal    -3(%rdi), %eax
    movd    -8(%rsp), %xmm1
    movl    %eax, -4(%rsp)
    xorl    %eax, %eax
    movd    -4(%rsp), %xmm0
    punpckldq %xmm0, %xmm1
    movd    -12(%rsp), %xmm0
    movl    %ecx, -12(%rsp)
    movd    -12(%rsp), %xmm2
    punpckldq %xmm2, %xmm0
    punpcklqdq %xmm1, %xmm0
    movdqa  LC0(%rip), %xmm1
    .align 4,0x90
```

```
L4:
    movdqa  %xmm1, %xmm3
    psrldq  $4, %xmm1
    addl    $1, %eax
    movdqa  %xmm0, %xmm2
    cmpl    %eax, %edx
    pmuludq %xmm0, %xmm3
    paddd   %xmm4, %xmm0
    psrldq  $4, %xmm2
    pmuludq %xmm1, %xmm2
    pshufd  $8, %xmm3, %xmm1
    pshufd  $8, %xmm2, %xmm2
    punpckldq %xmm2, %xmm1
    ja      L4
    movdqa  %xmm0, %xmm2
    subl    %edi, %eax
    cmpl    %esi, %ecx
    psrldq  $8, %xmm2
    movdqa  %xmm2, %xmm3
    psrldq  $4, %xmm2
    pmuludq %xmm1, %xmm0
    psrldq  $4, %xmm1
    pshufd  $8, %xmm0, %xmm0
    pmuludq %xmm2, %xmm1
    pshufd  $8, %xmm1, %xmm1
    punpckldq %xmm1, %xmm0
    movdqa  %xmm0, %xmm1
    psrldq  $4, %xmm1
    movdqa  %xmm1, %xmm2
```

```
    psrldq  $4, %xmm1
    pmuludq %xmm0, %xmm2
    psrldq  $4, %xmm0
    pmuludq %xmm0, %xmm1
    pshufd  $8, %xmm2, %xmm0
    pshufd  $8, %xmm1, %xmm1
    punpckldq %xmm1, %xmm0
    movd    %xmm0, -12(%rsp)
    movl    -12(%rsp), %eax
    je      L13
    .align 4,0x90
L9:
    imull   %edi, %eax
    subl    $1, %edi
    cmpl    $1, %edi
    jg      L9
L2:
    rep
    ret
L8:
    movl    $1, %eax
    jmp     L9
L13:
    .p2align 4,,6
    ret
```

Vectorized! Recursion eliminated!

Integer overflow?

```
unsigned uoverflow(void)
{
        unsigned x = 0;
        while (++x > 0)
                ;
        return x;
}
```

```
_uoverflow:
        xorl    %eax, %eax
        ret
```

```
int overflow(void)
{
        int x = 0;
        while (++x > 0)
                ;
        return x;
}
```

```
_overflow:
L2:
        jmp     L2
```

Optimizing dispatch tables?

```
static void foo(void) { printf("foo\n"); }
static void bar(void) { printf("bar\n"); }
static void baz(void) { printf("baz\n"); }

typedef void (*thunk_t)(void);
static const thunk_t handlers[] = { foo, bar, baz };

static void dispatch(int x)
{
        handlers[x]();
}

void start_it(void)
{
        dispatch(0);
}
```

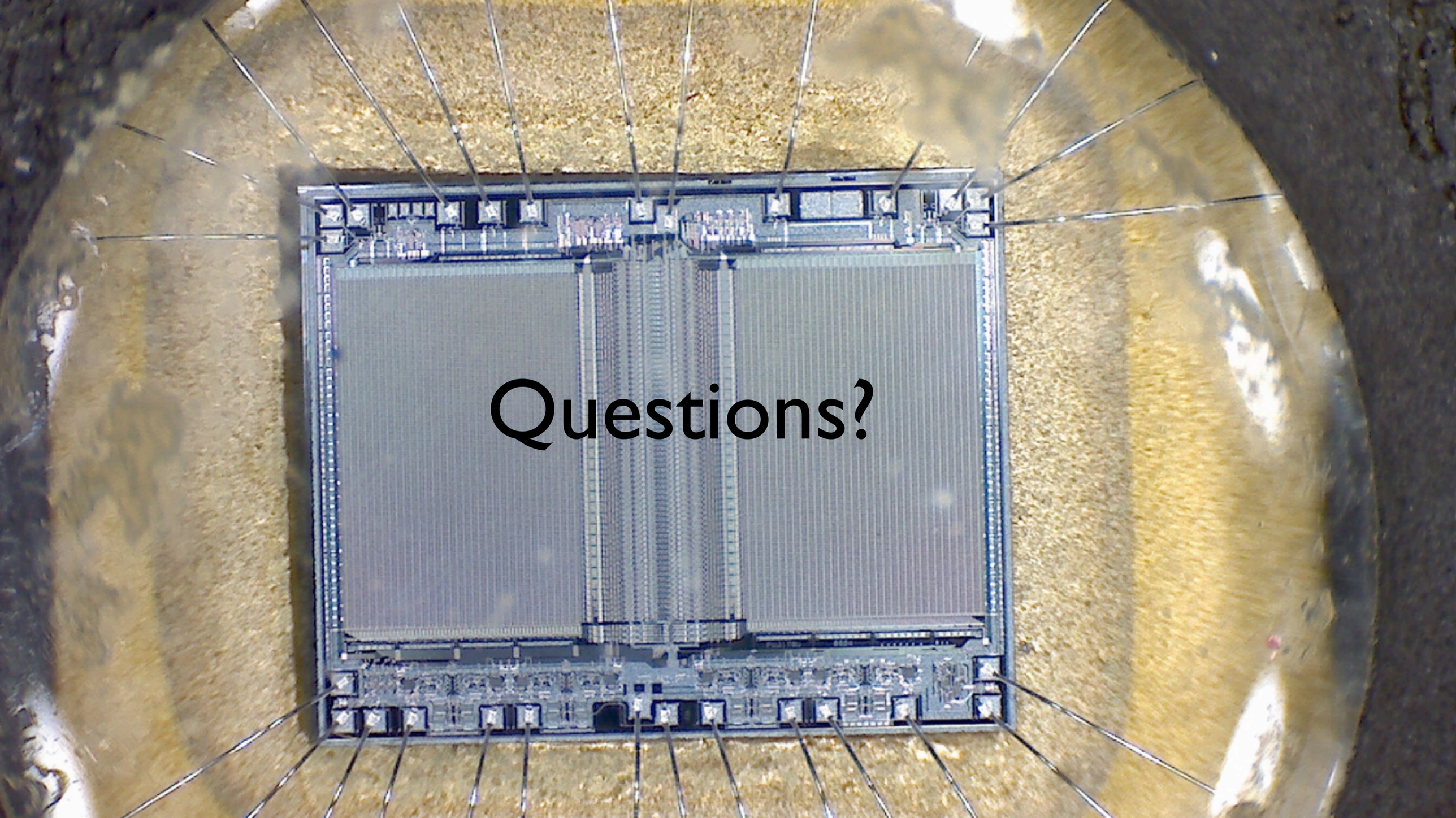
```
_start_it:
        leaq    LC2(%rip), %rdi
        jmp     _puts
        .data
LC2:
        .ascii "foo\0"
```

```
thunk_t handlers[] = {
        foo,
        bar,
        baz,
};
```

```
_start_it:
        movq    _handlers(%rip), %rax
        jmp     *%rax
```

For some reason we all (especially me) had a mental block about optimization, namely that we always regarded it as a behind-the-scenes activity, to be done in the machine language, which the programmer isn't supposed to know.

Knuth, 1974 "Structured Programming With goto statements"


Questions?