Undocumented CPU Behavior:

Analyzing Undocumented Opcodes on Intel x86-64

Catherine Easdon

Why investigate undocumented behavior?

The “golden screwdriver” approach

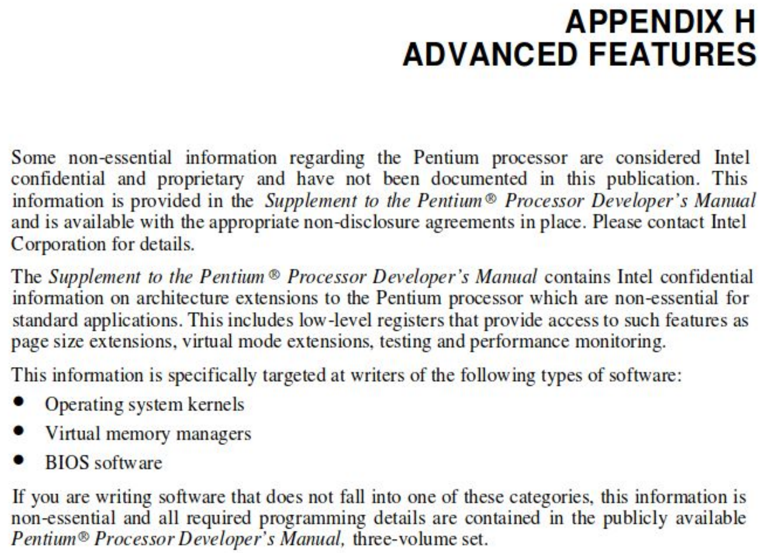
- Intel have confirmed they add undocumented features to general-release chips for key customers

"As far as the etching goes, we have done different things for different customers, and we have put different things into the silicon, such as adding instructions or pins or signals for logic for them. **The difference is that it goes into all of the silicon for that product.** And so the way that you do it is somebody gives you a feature, and they say, 'Hey, can you get this into the product?' You can't do something that takes up a huge amount of die, but you can do an instruction, you can do a signal, you can do a number of things that are logic-related." ~ *Jason Waxman, Intel Cloud Infrastructure Group*

(Source: http://www.theregister.co.uk/2013/05/20/intel_chip_customization/)

Poor documentation

- Intel has a long history of withholding information from their manuals



APPENDIX H
ADVANCED FEATURES

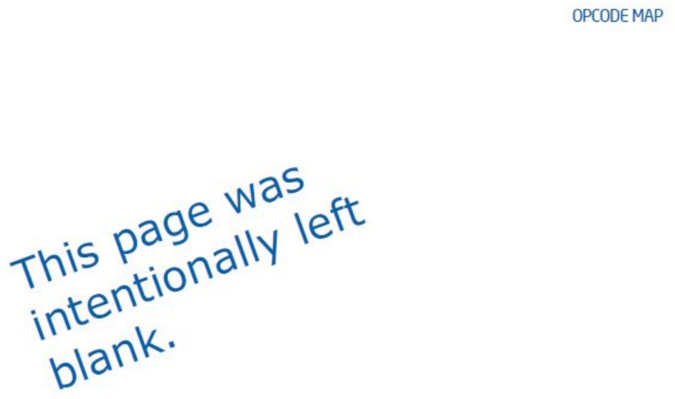
Some non-essential information regarding the Pentium processor are considered Intel confidential and proprietary and have not been documented in this publication. This information is provided in the *Supplement to the Pentium® Processor Developer's Manual* and is available with the appropriate non-disclosure agreements in place. Please contact Intel Corporation for details.

The *Supplement to the Pentium® Processor Developer's Manual* contains Intel confidential information on architecture extensions to the Pentium processor which are non-essential for standard applications. This includes low-level registers that provide access to such features as page size extensions, virtual mode extensions, testing and performance monitoring.

This information is specifically targeted at writers of the following types of software:

- Operating system kernels
- Virtual memory managers
- BIOS software

If you are writing software that does not fall into one of these categories, this information is non-essential and all required programming details are contained in the publicly available *Pentium® Processor Developer's Manual,* three-volume set.



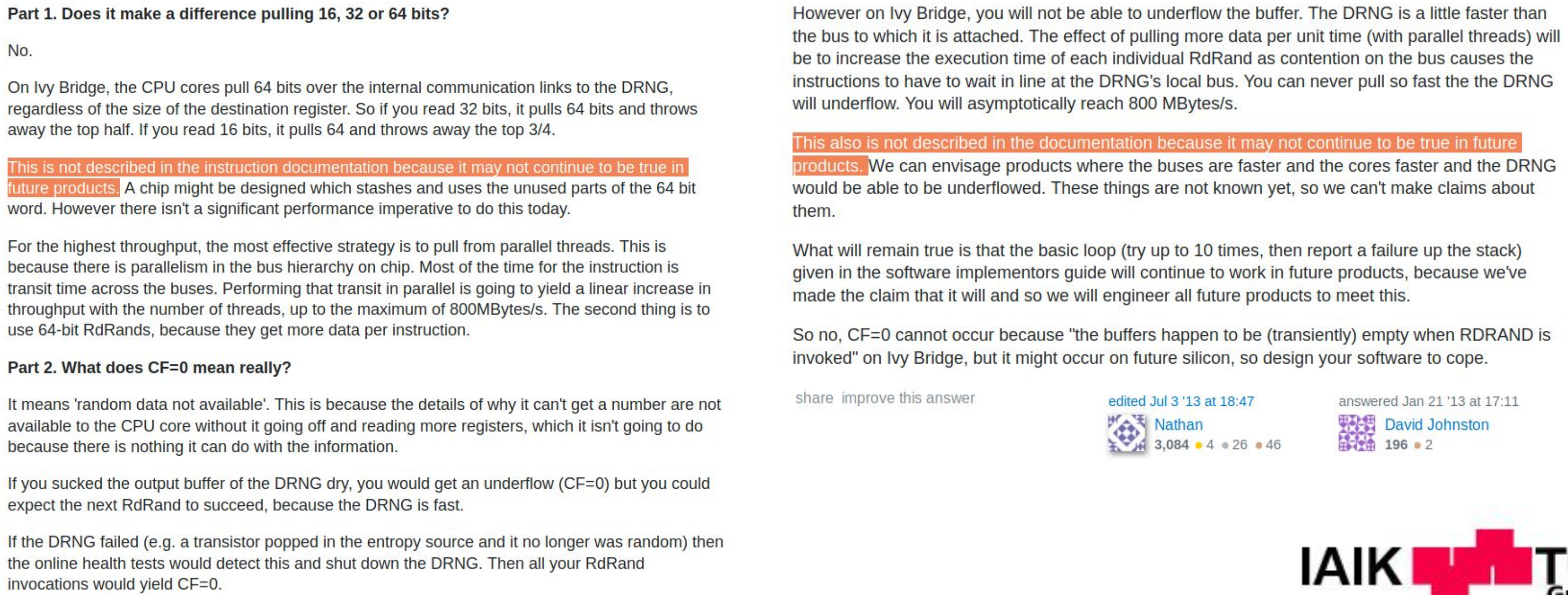
OPCODE MAP

This page was intentionally left blank.

(Source: http://datasheets.chipdb.org/Intel/x86/Pentium/24143004.PDF)

Poor documentation

- Intel has a long history of withholding information from their manuals



Part 1. Does it make a difference pulling 16, 32 or 64 bits?

No.

On Ivy Bridge, the CPU cores pull 64 bits over the internal communication links to the DRNG, regardless of the size of the destination register. So if you read 32 bits, it pulls 64 bits and throws away the top half. If you read 16 bits, it pulls 64 and throws away the top 3/4.

This is not described in the instruction documentation because it may not continue to be true in future products. A chip might be designed which stashes and uses the unused parts of the 64 bit word. However there isn't a significant performance imperative to do this today.

For the highest throughput, the most effective strategy is to pull from parallel threads. This is because there is parallelism in the bus hierarchy on chip. Most of the time for the instruction is transit time across the buses. Performing that transit in parallel is going to yield a linear increase in throughput with the number of threads, up to the maximum of 800MBytes/s. The second thing is to use 64-bit RdRands, because they get more data per instruction.

Part 2. What does CF=0 mean really?

It means 'random data not available'. This is because the details of why it can't get a number are not available to the CPU core without it going off and reading more registers, which it isn't going to do because there is nothing it can do with the information.

If you sucked the output buffer of the DRNG dry, you would get an underflow (CF=0) but you could expect the next RdRand to succeed, because the DRNG is fast.

If the DRNG failed (e.g. a transistor popped in the entropy source and it no longer was random) then the online health tests would detect this and shut down the DRNG. Then all your RdRand invocations would yield CF=0.

However on Ivy Bridge, you will not be able to underflow the buffer. The DRNG is a little faster than the bus to which it is attached. The effect of pulling more data per unit time (with parallel threads) will be to increase the execution time of each individual RdRand as contention on the bus causes the instructions to have to wait in line at the DRNG's local bus. You can never pull so fast the the DRNG will underflow. You will asymptotically reach 800 MBytes/s.

This also is not described in the documentation because it may not continue to be true in future products. We can envisage products where the buses are faster and the cores faster and the DRNG would be able to be underflowed. These things are not known yet, so we can't make claims about them.

What will remain true is that the basic loop (try up to 10 times, then report a failure up the stack) given in the software implementors guide will continue to work in future products, because we've made the claim that it will and so we will engineer all future products to meet this.

So no, CF=0 cannot occur because "the buffers happen to be (transiently) empty when RDRAND is invoked" on Ivy Bridge, but it might occur on future silicon, so design your software to cope.

share improve this answer

edited Jul 3 '13 at 18:47
Nathan
3,084 ● 4 ● 26 ● 46

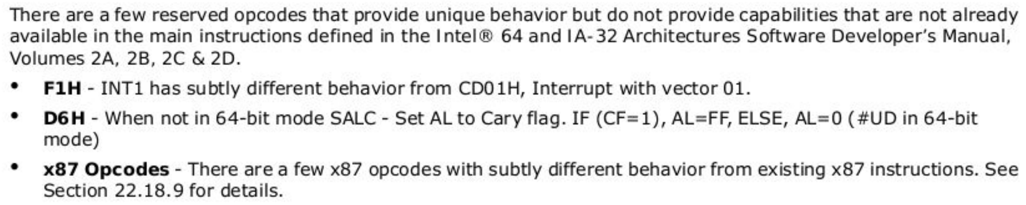
answered Jan 21 '13 at 17:11
David Johnston
196 ● 2

(Source: https://stackoverflow.com/questions/14413839/what-are-the-exhaustion-characteristics-of-rdrand-on-ivy-bridge)

Poor documentation

- Even when the manuals don't withhold information, they are often misleading or inconsistent

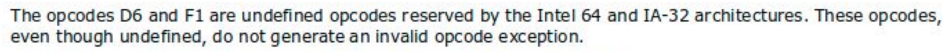
Section 22.15, Intel Developer Manual Vol. 3:



There are a few reserved opcodes that provide unique behavior but do not provide capabilities that are not already available in the main instructions defined in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volumes 2A, 2B, 2C & 2D.

- **F1H** - INT1 has subtly different behavior from CD01H, Interrupt with vector 01.
- **D6H** - When not in 64-bit mode SALC - Set AL to Cary flag. IF (CF=1), AL=FF, ELSE, AL=0 (#UD in 64-bit mode)
- **x87 Opcodes** - There are a few x87 opcodes with subtly different behavior from existing x87 instructions. See Section 22.18.9 for details.

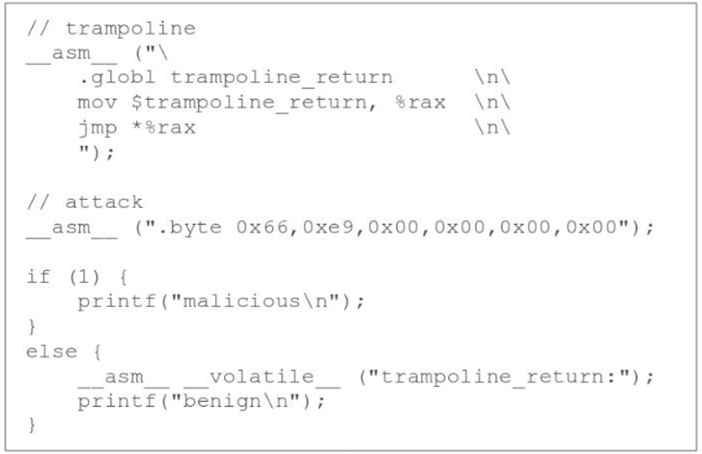
Section 6.15 (#UD exception):



The opcodes D6 and F1 are undefined opcodes reserved by the Intel 64 and IA-32 architectures. These opcodes, even though undefined, do not generate an invalid opcode exception.

Poor documentation leads to vulnerabilities

- In operating systems
  - **POP SS/MOV SS** (May 2018)
    - Developer confusion over #DB handling
    - Load + execute unsigned kernel code on Windows
    - Also affected: Linux, MacOS, FreeBSD...
- In virtual machines / emulators
  - Privilege escalation on a cloud instance!
- In disassemblers
  - “Anti-disassembly”
  - Hide malicious code in plain sight



```
// trampoline
__asm__ ("\
    .globl trampoline_return      \n\
    mov $trampoline_return, %rax  \n\
    jmp *%rax                     \n\
    ");

// attack
__asm__ (".byte 0x66,0xe9,0x00,0x00,0x00,0x00");

if (1) {
    printf("malicious\n");
}
else {
    __asm__ __volatile__ ("trampoline_return:");
    printf("benign\n");
}
```

Figure 7. A malicious program that prints “benign” when run under QEMU, but “malicious” when run on baremetal. The assembly trampoline at the top is copied into low memory, as a target for the mis-emulated jmp instruction, while the jump on baremetal simply falls through to the next instruction.

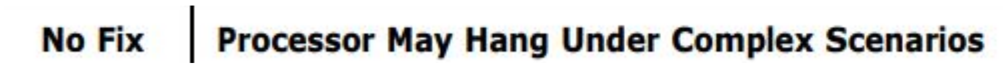
Source: Breaking the x86 ISA, Christopher Domas

Why investigate undocumented opcodes?

- Many undocumented x86 opcodes in the past
    - LOADALL, SALC, INT1 / ICEBP, UDO / UD1…

Why investigate undocumented opcodes?

- Many undocumented x86 opcodes in the past
- “Halt and catch fire” instructions
  - e.g. F00F C7C8 on Pentium
  - Could be used for denial of service attacks
  - “Killer poke”: POKE 62975, 0 (TRS-80 M100); POKE 59458,62 (Commodore PET)



No Fix	Processor May Hang Under Complex Scenarios

IAIK TU Graz

Why investigate undocumented opcodes?

- Many undocumented x86 opcodes in the past
- “Halt and catch fire” instructions
- Instructions which create exploitable side-channels
  - CLFLUSH, PREFETCH…

Why investigate undocumented opcodes?

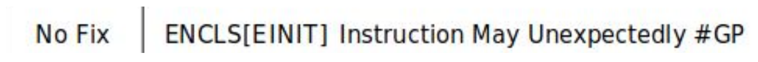
- Many undocumented x86 opcodes in the past
- “Halt and catch fire” instructions
- Instructions which create exploitable side-channels
- Hidden debug mechanisms

Why investigate undocumented opcodes?

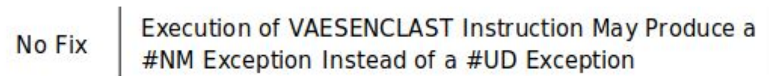
- Many undocumented x86 opcodes in the past
- “Halt and catch fire” instructions
- Instructions which create exploitable side-channels
- Hidden debug mechanisms
- Malicious microcode updates

Why investigate undocumented opcodes?

- Many undocumented x86 opcodes in the past
- “Halt and catch fire” instructions
- Instructions which create exploitable side-channels
- Hidden debug mechanisms
- Malicious microcode updates
- Undocumented behavior - bugs (“errata”)



No Fix	ENCLS[EINIT] Instruction May Unexpectedly #GP



No Fix	Execution of VAESENCLAST Instruction May Produce a #NM Exception Instead of a #UD Exception

Why investigate undocumented opcodes?

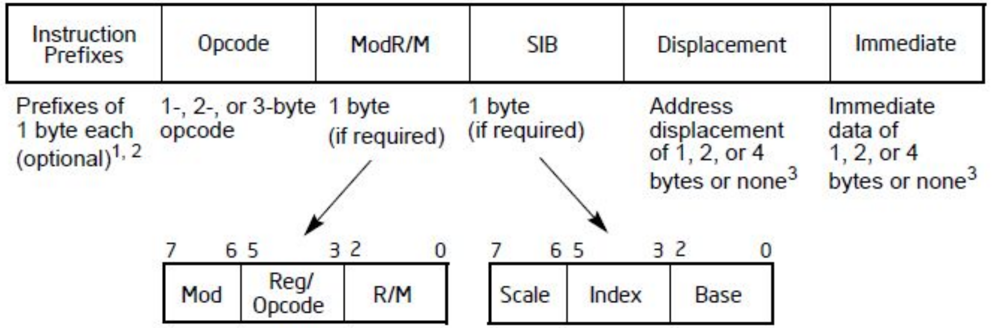
- Many undocumented x86 opcodes in the past
- “Halt and catch fire” instructions
- Instructions which create exploitable side-channels
- Hidden debug mechanisms
- Malicious microcode updates
- Undocumented behavior - bugs (“errata”)
- Capabilities of ultra-privileged modes (Intel ME, SMM…)
    - **Security by obscurity is not enough**

How can we find undocumented opcodes?

How many opcodes are documented?

- No-one knows!
- 1569 XED iclasses (~mnemonics)
- But >30 different encodings for MOV alone…
- 6290 iforms (e.g. ADC_GPRv_IMMb)
- Iforms still don’t account for all variations, e.g. some prefixes

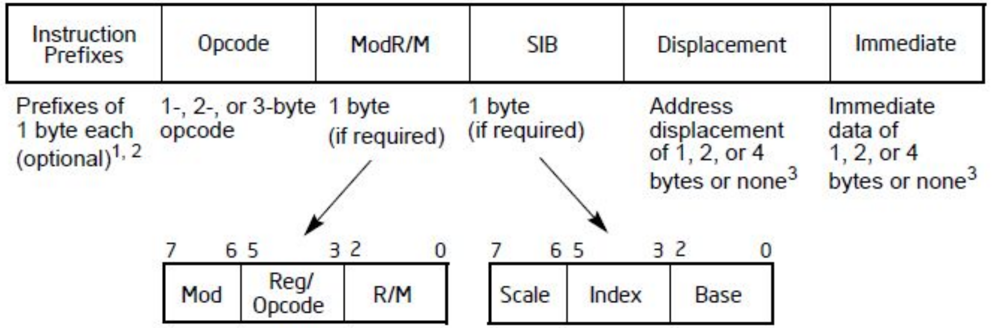
The instruction search space



Opcode != instruction, but max instruction length: 15 bytes

2^15*8 possibilities (~1.33 x 1036)= 1,329,227,995,784,915,872,903,807,060,280,344,576

The instruction search space



- If we test 1 billion instructions a second...it'll only take us 4.21 x 10^{19} years of 24/7/365 testing (that's ~98x the age of the universe)
- And there'll be crashes + processor failures to deal with. Any volunteers?
- So a brute-force search for undocumented (or documented!) instructions is infeasible

Approach 1: manual targeting

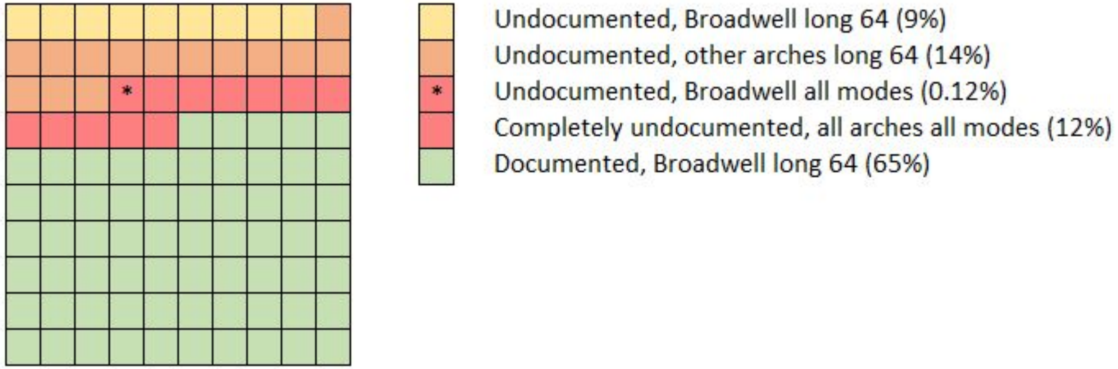
- Requires a deep understanding of the ISA
- Lots of time needed (and there'll always be *just one more opcode* to investigate…)
- Example: Corkami Standard Test by Ange Albertini (for Windows)
  - Aimed at identifying disassembler / OS flaws
  - Exploring misunderstood/undocumented behavior

https://code.google.com/archive/p/corkami/wikis/StandardTest.wiki

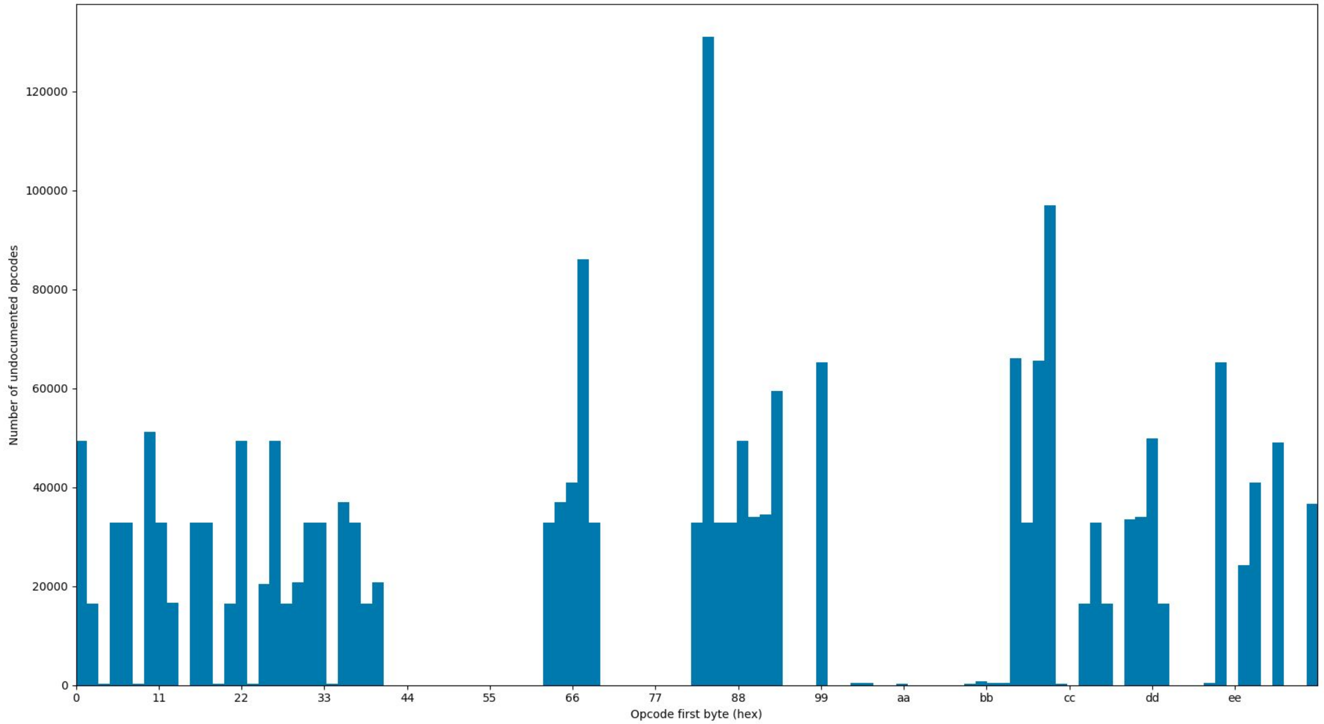
https://www.youtube.com/watch?v=MJvsshovITE (Talk: 'Such a weird processor - messing with x86 opcodes')

Approach 2: opcode search

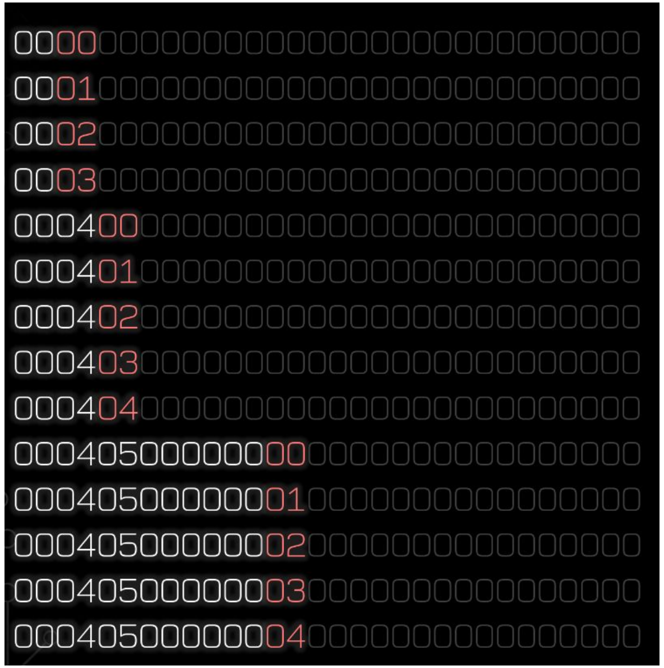
- Just search within 3-byte opcode range
  - 2 ^ (3*8) = 16,777,216. Feasible for brute search
  - BUT **extremely** buggy when executing (stack smashing, chains of seg faults…most likely undocumented jumps!)
  - Seg faults are much harder to handle than illegal opcode exceptions



Approach 2: opcode search



Approach 3: tunneling

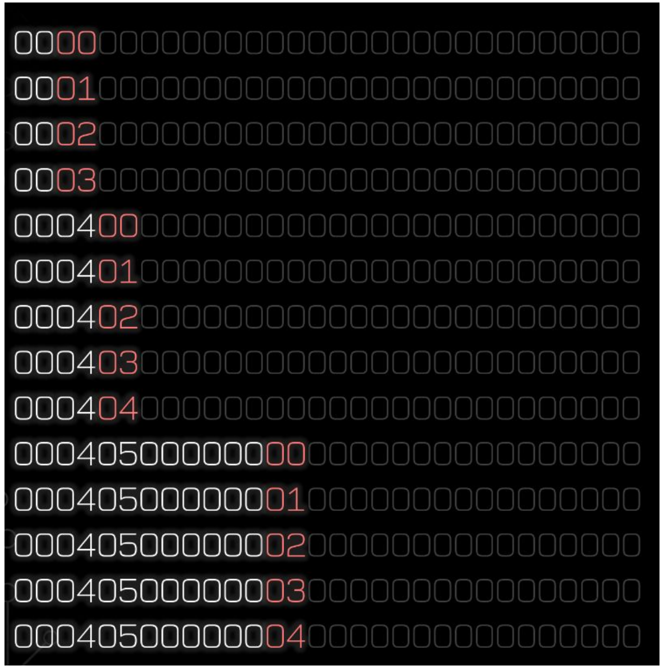


```
000000000000000000000000000000
000100000000000000000000000000
000200000000000000000000000000
000300000000000000000000000000
000400000000000000000000000000
000401000000000000000000000000
000402000000000000000000000000
000403000000000000000000000000
000404000000000000000000000000
000405000000000000000000000000
000405000000010000000000000000
000405000000020000000000000000
000405000000030000000000000000
000405000000040000000000000000
```

- Previous research in this area by Christopher Domas - created Sandsifter tool
- Tunneling algorithm
  - Depth-first search: execute instruction, observe its length, increment last byte, repeat…
    - If length change - start incrementing new last byte
    - If FF, set last byte to 00, start incrementing second to last byte
  - Instruction length determined via page faults (where does the CPU stop decoding?)

https://github.com/xoreaxeaxeax/sandsifter

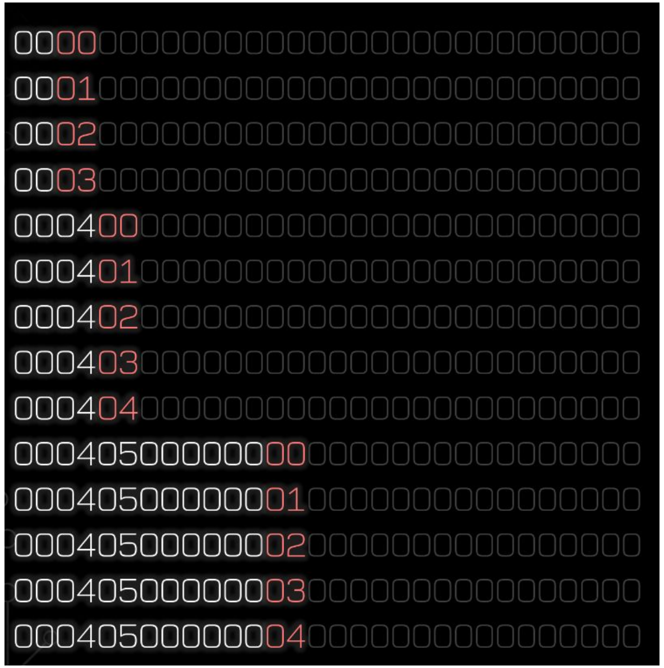
Approach 3: tunneling



```
000000000000000000000000000000
000100000000000000000000000000
000200000000000000000000000000
000300000000000000000000000000
000400000000000000000000000000
000401000000000000000000000000
000402000000000000000000000000
000403000000000000000000000000
000404000000000000000000000000
000405000000000000000000000000
000405000000000100000000000000
000405000000000200000000000000
000405000000000300000000000000
000405000000000400000000000000
```

- Advantages:
  - Reduces search space to ~1,000,000,000 instructions
  - Much more stable (as it is guided by the CPU decoder)
- Flaws:
  - Reduces search space to ~1,000,000,000 instructions! Assumes a length change is the only “interesting” change
  - Assumes the CPU will stop decoding after one instruction...it doesn’t
  - 0000 isn’t padding - it can be decoded as an ADD. Note: 00 isn’t valid.

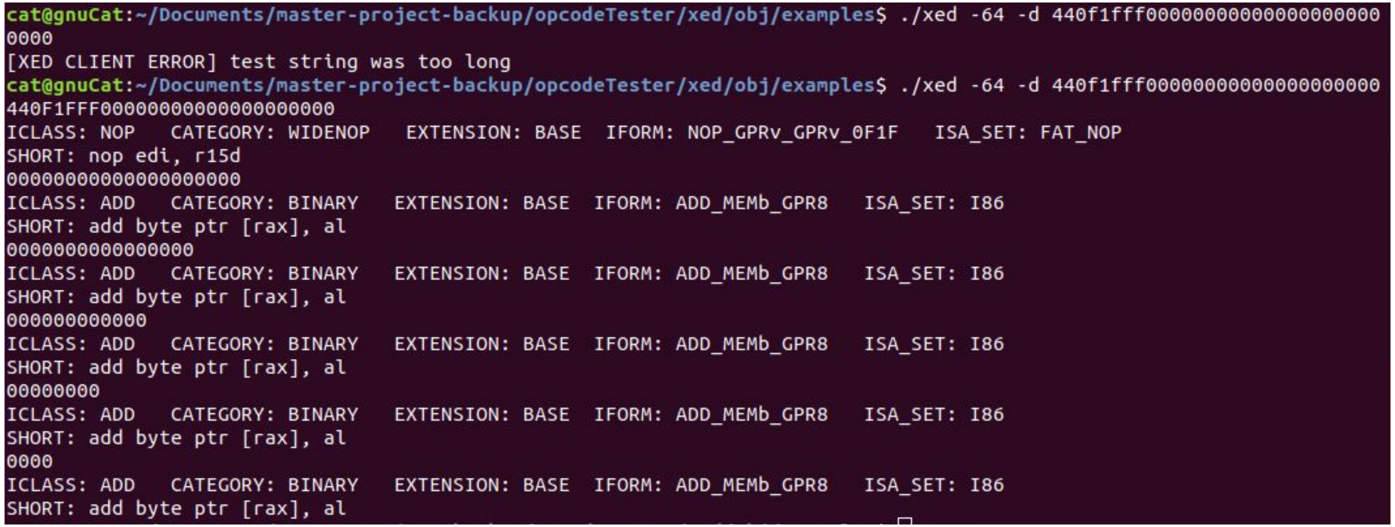
Approach 3: tunneling


```
000000000000000000000000000000
000100000000000000000000000000
000200000000000000000000000000
000300000000000000000000000000
000400000000000000000000000000
000401000000000000000000000000
000402000000000000000000000000
000403000000000000000000000000
000404000000000000000000000000
000405000000000000000000000000
000405000000010000000000000000
000405000000020000000000000000
000405000000030000000000000000
000405000000040000000000000000
```

- Sandsifter is currently the most stable automated approach for detecting undocumented instructions
  - Note: by default Sandsifter also searches for disassembler bugs. Run with --unk flag only (not --dis or --len).
- But Sandsifter has problems:
  - Lots of false positives - many valid but unusual instructions unknown to Capstone disassembler
  - Assumes all instructions which throw SIG_ILL are invalid and can never execute (not true)

Approach 3: tunneling

"If a REX prefix is used when it has no meaning, it is ignored."

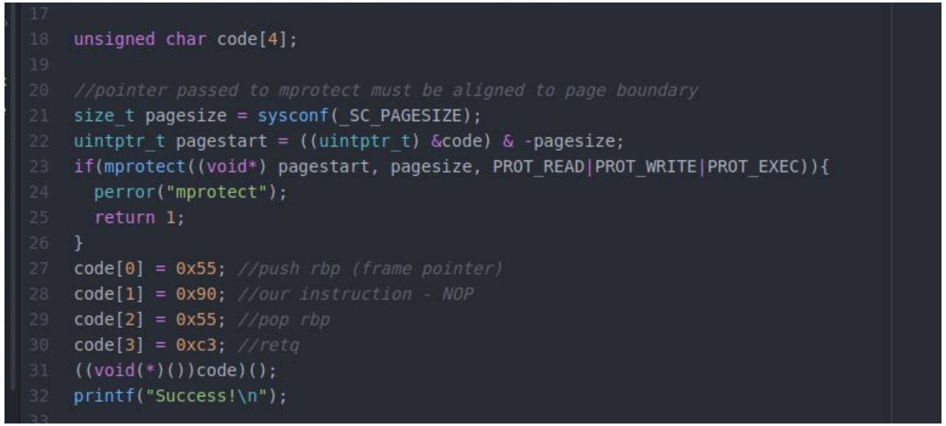


```
cat@gnuCat:~/Documents/master-project-backup/opcodeTester/xed/obj/examples$ ./xed -64 -d 440f1fff00000000000000000000
0000
[XED CLIENT ERROR] test string was too long
cat@gnuCat:~/Documents/master-project-backup/opcodeTester/xed/obj/examples$ ./xed -64 -d 440f1fff00000000000000000000
440F1FFF00000000000000000000
ICLASS: NOP   CATEGORY: WIDENOP   EXTENSION: BASE  IFORM: NOP_GPRv_GPRv_0F1F   ISA_SET: FAT_NOP
SHORT: nop edi, r15d
00000000000000000000
ICLASS: ADD   CATEGORY: BINARY   EXTENSION: BASE  IFORM: ADD_MEMb_GPR8   ISA_SET: I86
SHORT: add byte ptr [rax], al
0000000000000000
ICLASS: ADD   CATEGORY: BINARY   EXTENSION: BASE  IFORM: ADD_MEMb_GPR8   ISA_SET: I86
SHORT: add byte ptr [rax], al
000000000000
ICLASS: ADD   CATEGORY: BINARY   EXTENSION: BASE  IFORM: ADD_MEMb_GPR8   ISA_SET: I86
SHORT: add byte ptr [rax], al
00000000
ICLASS: ADD   CATEGORY: BINARY   EXTENSION: BASE  IFORM: ADD_MEMb_GPR8   ISA_SET: I86
SHORT: add byte ptr [rax], al
0000
ICLASS: ADD   CATEGORY: BINARY   EXTENSION: BASE  IFORM: ADD_MEMb_GPR8   ISA_SET: I86
SHORT: add byte ptr [rax], al
```

How can we execute an undocumented opcode?

Execution in ring 3 (user mode)

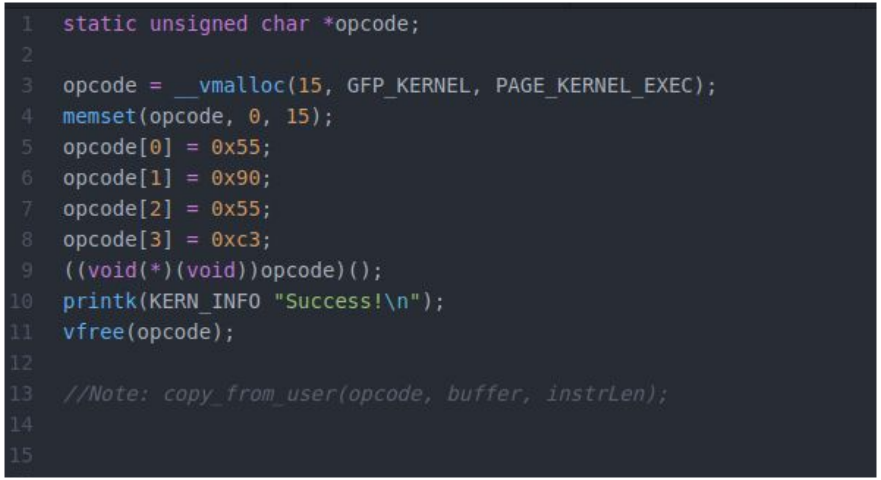
- Unsigned char array of hex instruction bytes + function prologue and epilogue
- **mprotect** to make page containing array executable
  - Must align to start of page boundary
  - Assumes array fits in one page
- Create a function pointer to the array and call it
- Need a **signal handler** if testing undocumented instructions!



```c
unsigned char code[4];

//pointer passed to mprotect must be aligned to page boundary
size_t pagesize = sysconf(_SC_PAGESIZE);
uintptr_t pagestart = ((uintptr_t) &code) & -pagesize;
if(mprotect((void*) pagestart, pagesize, PROT_READ|PROT_WRITE|PROT_EXEC)){
  perror("mprotect");
  return 1;
}
code[0] = 0x55; //push rbp (frame pointer)
code[1] = 0x90; //our instruction - NOP
code[2] = 0x55; //pop rbp
code[3] = 0xc3; //retq
((void(*)())code)();
printf("Success!\n");
```

Execution in ring 0 (kernel driver)

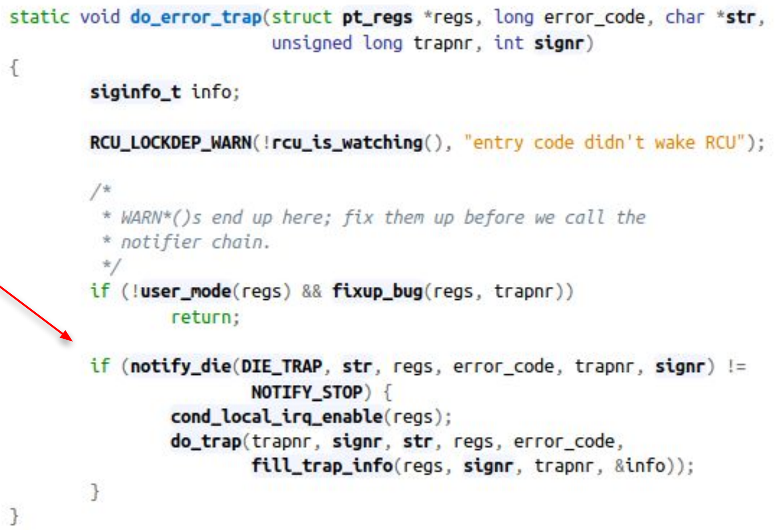
- Similar to user mode using kernel functions
- Exception handling is the hardest part
  - We're not supposed to throw exceptions in the kernel, but most undocumented instructions do fault
  - **Die notifiers** are the kernel equivalent of signal handlers



```c
static unsigned char *opcode;

opcode = __vmalloc(15, GFP_KERNEL, PAGE_KERNEL_EXEC);
memset(opcode, 0, 15);
opcode[0] = 0x55;
opcode[1] = 0x90;
opcode[2] = 0x55;
opcode[3] = 0xc3;
((void(*)(void))opcode)();
printk(KERN_INFO "Success!\n");
vfree(opcode);

//Note: copy_from_user(opcode, buffer, instrLen);


```

Handling exceptions in the kernel driver

- Kernel source digging: do_error_trap calls do_trap_no_signal which calls die. End result: kernel oops and our user process gets killed
- Solution: use a die notifier to return **NOTIFY_STOP**
- But this alone will hang the system!
  - Need to **re-enable interrupts** on this core
  - Need to **move the instruction pointer** past the faulting instruction: (instruction length - 2) if length > 2
- Alternatives: Systemtap, modify IDT
- Important to **minimise messages** in the kernel log



```
static void do_error_trap(struct pt_regs *regs, long error_code, char *str,
                          unsigned long trapnr, int signr)
{
        siginfo_t info;

        RCU_LOCKDEP_WARN(!rcu_is_watching(), "entry code didn't wake RCU");

        /*
         * WARN*()s end up here; fix them up before we call the
         * notifier chain.
         */
        if (!user_mode(regs) && fixup_bug(regs, trapnr))
                return;

        if (notify_die(DIE_TRAP, str, regs, error_code, trapnr, signr) !=
                        NOTIFY_STOP) {
                cond_local_irq_enable(regs);
                do_trap(trapnr, signr, str, regs, error_code,
                        fill_trap_info(regs, signr, trapnr, &info));
        }
}
```

Source: https://elixir.bootlin.com/linux/latest/source/arch/x86/kernel/traps.c
See also:
https://elixir.bootlin.com/linux/latest/source/arch/x86/entry/entry_64.S
https://0xax.gitbooks.io/linux-insides/Interrupts/linux-interrupts-5.html

How can we determine what an undocumented opcode does?

Monitoring opcode execution

- Clock cycles
  - Notoriously difficult to measure accurately
  - Kernel RDTSCP vs. counters
  - More realistic: distinguish between NOPs, simple instructions, and complex instructions
- Performance counters
  - Uops per execution port
  - Floating point operations
  - Memory operations
  - Lots more!

Execution port profiling

- Information sources:
  - Intel Optimization Reference Manual
  - Agner Fog's optimization manuals (3 and 4)
- Microarchitecture-specific
- Needs runtime profiling (all counters have overhead)
- Assume we can saturate all relevant ports for an instruction

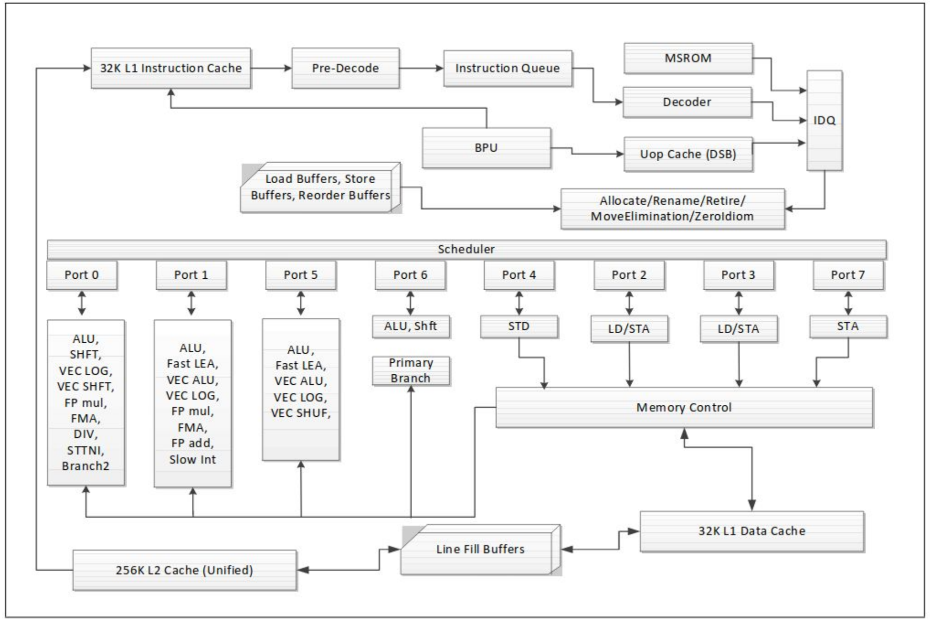

32K L1 Instruction Cache
Pre-Decode
Instruction Queue
MSROM
Decoder
IDQ
BPU
Uop Cache (DSB)
Load Buffers, Store Buffers, Reorder Buffers
Allocate/Rename/Retire/ MoveElimination/ZeroIdiom
Scheduler
Port 0
Port 1
Port 5
Port 6
Port 4
Port 2
Port 3
Port 7
ALU, SHFT, VEC LOG, VEC SHFT, FP mul, FMA, DIV, STTNI, Branch2
ALU, Fast LEA, VEC ALU, VEC LOG, FP mul, FMA, FP add, Slow Int
ALU, Fast LEA, VEC ALU, VEC LOG, VEC SHUF,
ALU, Shft
Primary Branch
STD
LD/STA
LD/STA
STA
Memory Control
32K L1 Data Cache
Line Fill Buffers
256K L2 Cache (Unified)

Figure 2-4. CPU Core Pipeline Functionality of the Haswell Microarchitecture

Execution port profiling

- Combine port uop counters with other counters (memory, FPU, branches…) to make a best guess at functionality
- Surprisingly effective for identifying broad categories (branch, load/store address, division…)
- Program must be locked to a single core (taskset -c 0)

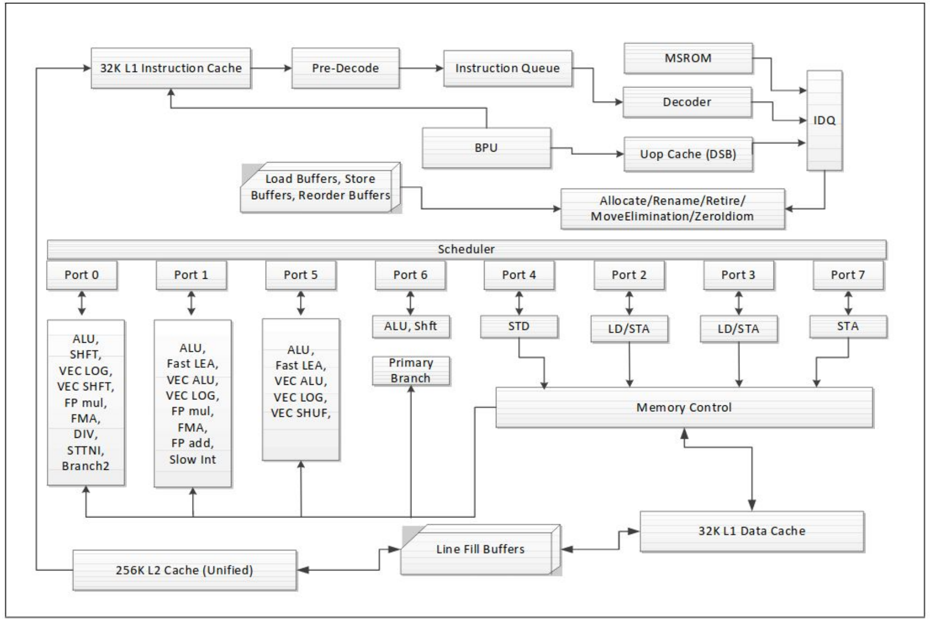
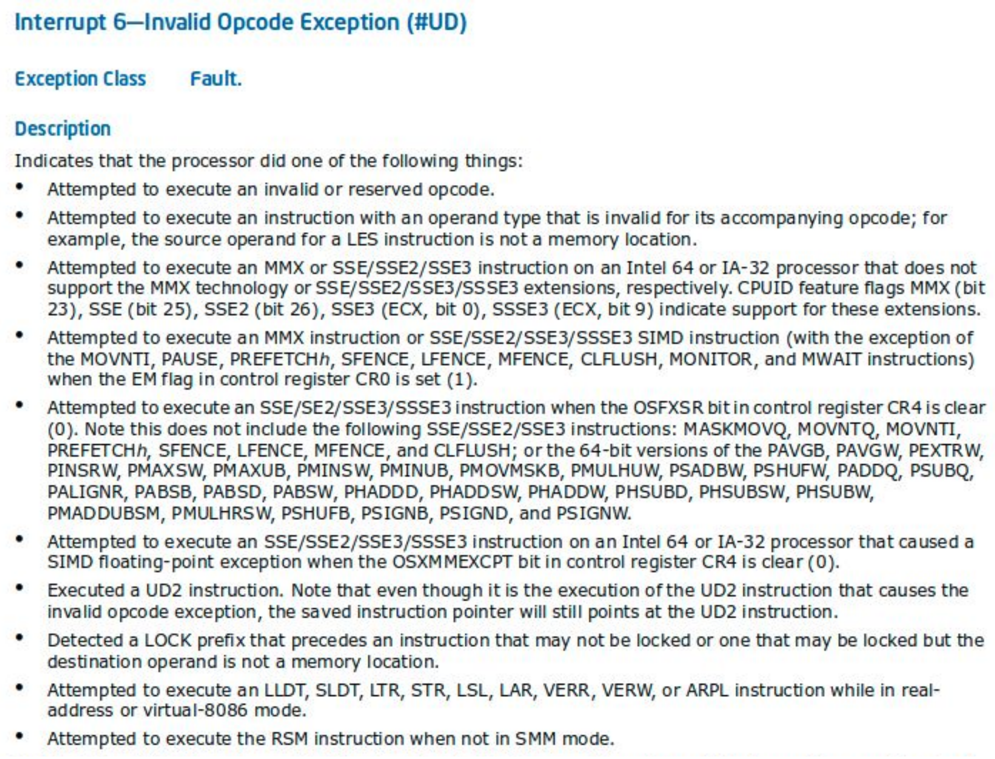

32K L1 Instruction Cache
Pre-Decode
Instruction Queue
MSROM
Decoder
IDQ
BPU
Uop Cache (DSB)
Load Buffers, Store Buffers, Reorder Buffers
Allocate/Rename/Retire/ MoveElimination/ZeroIdiom
Scheduler
Port 0
Port 1
Port 5
Port 6
Port 4
Port 2
Port 3
Port 7
ALU, SHFT, VEC LOG, VEC SHFT, FP mul, FMA, DIV, STTNI, Branch2
ALU, Fast LEA, VEC ALU, VEC LOG, FP mul, FMA, FP add, Slow Int
ALU, Fast LEA, VEC ALU, VEC LOG, VEC SHUF,
ALU, Shft
Primary Branch
STD
LD/STA
LD/STA
STA
Memory Control
32K L1 Data Cache
Line Fill Buffers
256K L2 Cache (Unified)

Figure 2-4. CPU Core Pipeline Functionality of the Haswell Microarchitecture

Can we learn anything about faulting instructions?



Interrupt 6—Invalid Opcode Exception (#UD)

Exception Class **Fault.**

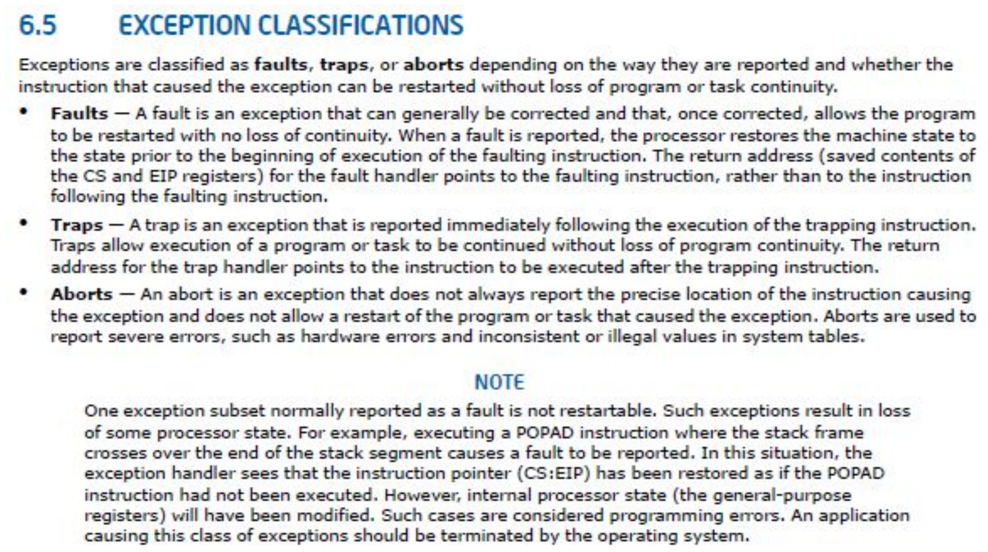
Description

Indicates that the processor did one of the following things:

- Attempted to execute an invalid or reserved opcode.
- Attempted to execute an instruction with an operand type that is invalid for its accompanying opcode; for example, the source operand for a LES instruction is not a memory location.
- Attempted to execute an MMX or SSE/SSE2/SSE3 instruction on an Intel 64 or IA-32 processor that does not support the MMX technology or SSE/SSE2/SSE3/SSSE3 extensions, respectively. CPUID feature flags MMX (bit 23), SSE (bit 25), SSE2 (bit 26), SSE3 (ECX, bit 0), SSSE3 (ECX, bit 9) indicate support for these extensions.
- Attempted to execute an MMX instruction or SSE/SSE2/SSE3/SSSE3 SIMD instruction (with the exception of the MOVNTI, PAUSE, PREFETCH*h*, SFENCE, LFENCE, MFENCE, CLFLUSH, MONITOR, and MWAIT instructions) when the EM flag in control register CR0 is set (1).
- Attempted to execute an SSE/SE2/SSE3/SSSE3 instruction when the OSFXSR bit in control register CR4 is clear (0). Note this does not include the following SSE/SSE2/SSE3 instructions: MASKMOVQ, MOVNTQ, MOVNTI, PREFETCH*h*, SFENCE, LFENCE, MFENCE, and CLFLUSH; or the 64-bit versions of the PAVGB, PAVGW, PEXTRW, PINSRW, PMAXSW, PMAXUB, PMINSW, PMINUB, PMOVMSKB, PMULHUW, PSADBW, PSHUFW, PADDQ, PSUBQ, PALIGNR, PABSB, PABSD, PABSW, PHADDD, PHADDSW, PHADDW, PHSUBD, PHSUBSW, PHSUBW, PMADDUBSM, PMULHRSW, PSHUFB, PSIGNB, PSIGND, and PSIGNW.
- Attempted to execute an SSE/SSE2/SSE3/SSSE3 instruction on an Intel 64 or IA-32 processor that caused a SIMD floating-point exception when the OSXMMEXCPT bit in control register CR4 is clear (0).
- Executed a UD2 instruction. Note that even though it is the execution of the UD2 instruction that causes the invalid opcode exception, the saved instruction pointer will still points at the UD2 instruction.
- Detected a LOCK prefix that precedes an instruction that may not be locked or one that may be locked but the destination operand is not a memory location.
- Attempted to execute an LLDT, SLDT, LTR, STR, LSL, LAR, VERR, VERW, or ARPL instruction while in real-address or virtual-8086 mode.
- Attempted to execute the RSM instruction when not in SMM mode.

(Source: Intel Developer Manual Vol. 3, section 6.15)

Can we learn anything about faulting instructions?



6.5 EXCEPTION CLASSIFICATIONS

Exceptions are classified as **faults**, **traps**, or **aborts** depending on the way they are reported and whether the instruction that caused the exception can be restarted without loss of program or task continuity.

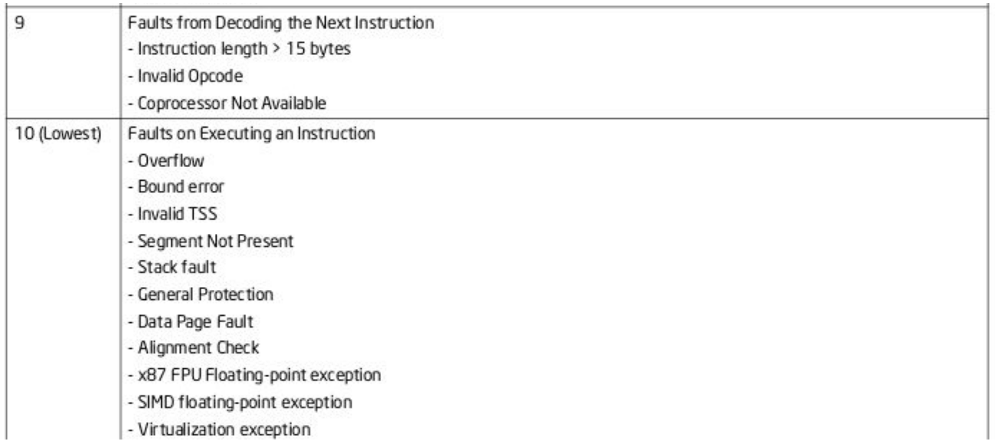
- **Faults** — A fault is an exception that can generally be corrected and that, once corrected, allows the program to be restarted with no loss of continuity. When a fault is reported, the processor restores the machine state to the state prior to the beginning of execution of the faulting instruction. The return address (saved contents of the CS and EIP registers) for the fault handler points to the faulting instruction, rather than to the instruction following the faulting instruction.
- **Traps** — A trap is an exception that is reported immediately following the execution of the trapping instruction. Traps allow execution of a program or task to be continued without loss of program continuity. The return address for the trap handler points to the instruction to be executed after the trapping instruction.
- **Aborts** — An abort is an exception that does not always report the precise location of the instruction causing the exception and does not allow a restart of the program or task that caused the exception. Aborts are used to report severe errors, such as hardware errors and inconsistent or illegal values in system tables.

NOTE

One exception subset normally reported as a fault is not restartable. Such exceptions result in loss of some processor state. For example, executing a POPAD instruction where the stack frame crosses over the end of the stack segment causes a fault to be reported. In this situation, the exception handler sees that the instruction pointer (CS:EIP) has been restored as if the POPAD instruction had not been executed. However, internal processor state (the general-purpose registers) will have been modified. Such cases are considered programming errors. An application causing this class of exceptions should be terminated by the operating system.

So...we can learn nothing, as the machine state is **entirely** restored to the pre-execution state? What about that subset of exceptions which do result in loss of execution state?

Can we learn anything about faulting instructions?

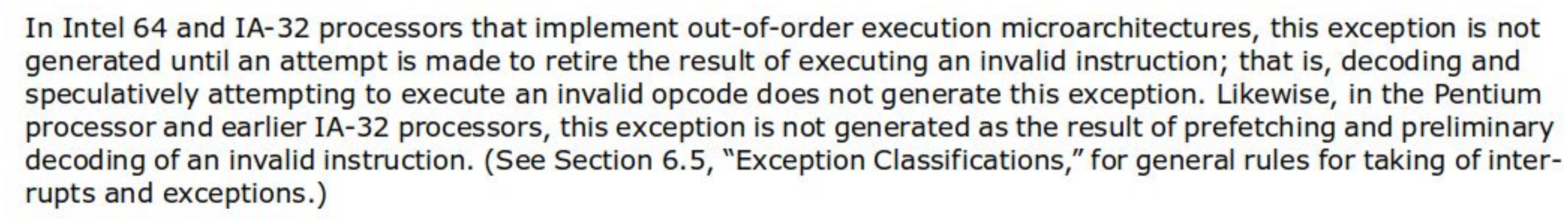


9	Faults from Decoding the Next Instruction - Instruction length > 15 bytes - Invalid Opcode - Coprocessor Not Available
10 (Lowest)	Faults on Executing an Instruction - Overflow - Bound error - Invalid TSS - Segment Not Present - Stack fault - General Protection - Data Page Fault - Alignment Check - x87 FPU Floating-point exception - SIMD floating-point exception - Virtualization exception

Source: Table 6-2 'Priority among Simultaneous Exceptions and Interrupts', Intel Developer Manual Vol. 3

Wait...if #UD is a fault from *decoding the next instruction*, does the instruction even execute at all?

Can we learn anything about faulting instructions?


In Intel 64 and IA-32 processors that implement out-of-order execution microarchitectures, this exception is not generated until an attempt is made to retire the result of executing an invalid instruction; that is, decoding and speculatively attempting to execute an invalid opcode does not generate this exception. Likewise, in the Pentium processor and earlier IA-32 processors, this exception is not generated as the result of prefetching and preliminary decoding of an invalid instruction. (See Section 6.5, "Exception Classifications," for general rules for taking of interrupts and exceptions.)

(Source: Intel Developer Manual Vol. 3, section 6.15)

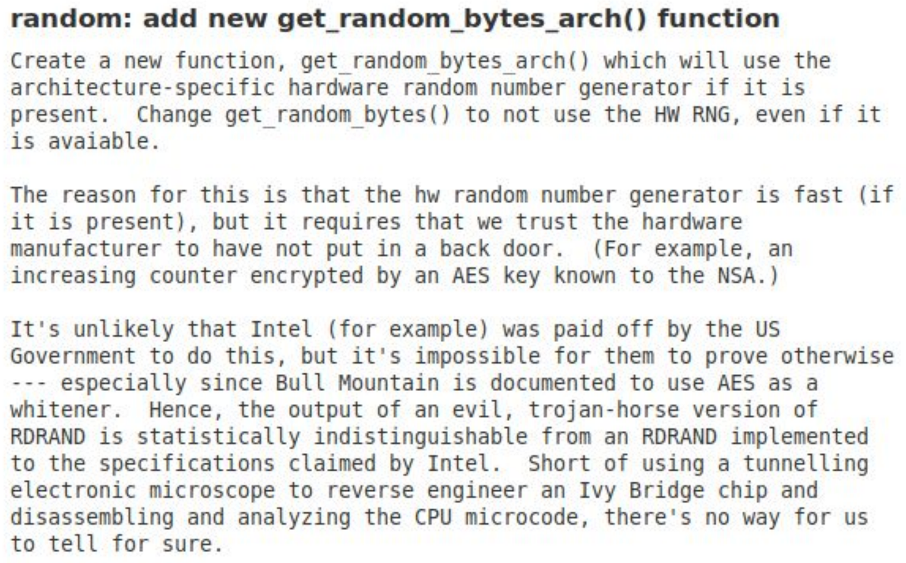
Apparently, yes it can be *speculatively* executed.

Did someone say Spectre?...We can also target it with performance counters!

Do #UD instructions leave microarchitectural traces behind?

Can we defend against *unknown* undocumented behavior?

Defending random.c from RDRAND



random: add new get_random_bytes_arch() function

```
Create a new function, get_random_bytes_arch() which will use the
architecture-specific hardware random number generator if it is
present.  Change get_random_bytes() to not use the HW RNG, even if it
is avaiable.

The reason for this is that the hw random number generator is fast (if
it is present), but it requires that we trust the hardware
manufacturer to have not put in a back door.  (For example, an
increasing counter encrypted by an AES key known to the NSA.)

It's unlikely that Intel (for example) was paid off by the US
Government to do this, but it's impossible for them to prove otherwise
--- especially since Bull Mountain is documented to use AES as a
whitener.  Hence, the output of an evil, trojan-horse version of
RDRAND is statistically indistinguishable from an RDRAND implemented
to the specifications claimed by Intel.  Short of using a tunnelling
electronic microscope to reverse engineer an Ivy Bridge chip and
disassembling and analyzing the CPU microcode, there's no way for us
to tell for sure.
```

Defending random.c from RDRAND

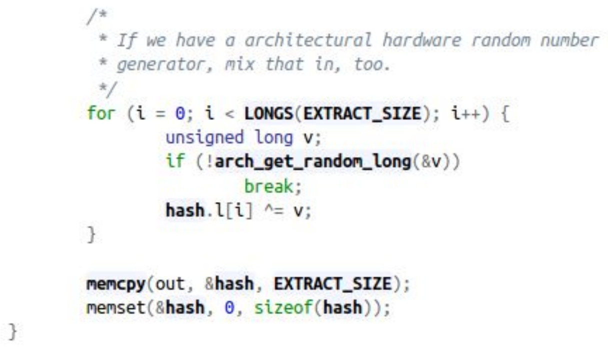
random: mix in architectural randomness in extract_buf() [July 2012]

“Mix in any architectural randomness in extract_buf() instead of xfer_secondary_buf(). This allows us to mix in more architectural randomness, and it also makes xfer_secondary_buf() faster, moving a tiny bit of additional CPU overhead to process which is extracting the randomness.

[Commit description modified by tytso to remove an extended advertisement for the RDRAND instruction.]

Signed-off-by: H. Peter Anvin <hpa@linux.intel.com>
Acked-by: Ingo Molnar <mingo@kernel.org>
Cc: DJ Johnston <dj.johnston@intel.com>
Signed-off-by: Theodore Ts'o <tytso@mit.edu>
Cc: stable@vger.kernel.org”

For full code, see:
https://github.com/torvalds/linux/blob/master/drivers/char/random.c
https://elixir.bootlin.com/linux/v3.0.41/source/drivers/char/random.c



```
        /*
         * If we have a architectural hardware random number
         * generator, mix that in, too.
         */
        for (i = 0; i < LONGS(EXTRACT_SIZE); i++) {
                unsigned long v;
                if (!arch_get_random_long(&v))
                        break;
                hash.l[i] ^= v;
        }

        memcpy(out, &hash, EXTRACT_SIZE);
        memset(&hash, 0, sizeof(hash));
}
```

What’s wrong here?

- HWRNG output added **after** all mixing - so it controls the final “random” output
- Imagine HWRNG is compromised via malicious microcode update
- Microcode has access to our hash.l[i] value
- It can output a value v which XORs to 0

Defending random.c from RDRAND

random: mix in architectural randomness in extract_buf() [July 2012]

“Mix in any architectural randomness in extract_buf() instead of xfer_secondary_buf(). This allows us to mix in more architectural randomness, and it also makes xfer_secondary_buf() faster, moving a tiny bit of additional CPU overhead to process which is extracting the randomness.

[Commit description modified by tytso to remove an extended advertisement for the RDRAND instruction.]

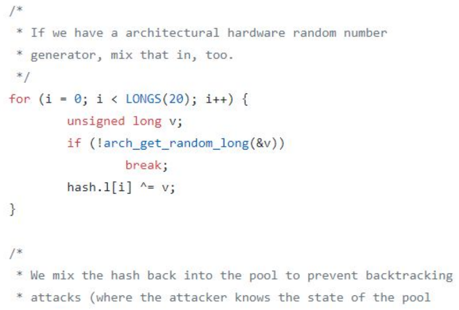
Signed-off-by: H. Peter Anvin <hpa@linux.intel.com>
Acked-by: Ingo Molnar <mingo@kernel.org>
Cc: DJ Johnston <dj.johnston@intel.com>
Signed-off-by: Theodore Ts'o <tytso@mit.edu>
Cc: stable@vger.kernel.org”

For full code, see:
https://github.com/torvalds/linux/blob/master/drivers/char/random.c
https://elixir.bootlin.com/linux/v3.0.41/source/drivers/char/random.c

random: mix in architectural randomness earlier in extract_buf() [September 2013]

“Previously if CPU chip had a built-in random number generator (i.e., RDRAND on newer x86 chips), we mixed it in at the very end of extract_buf() using an XOR operation.

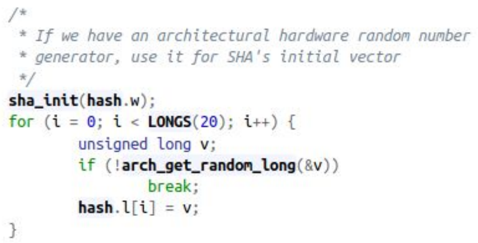
We now mix it in right after the calculate a hash across the entire pool. This has the advantage that any contribution of entropy from the CPU's HWRNG will get mixed back into the pool. In addition, it means that if the HWRNG has any defects (either accidentally or maliciously introduced), this will be mitigated via the non-linear transform of the SHA-1 hash function before we hand out generated output.
Signed-off-by: "Theodore Ts'o" <tytso@mit.edu>”



```
/*
 * If we have a architectural hardware random number
 * generator, mix that in, too.
 */
for (i = 0; i < LONGS(20); i++) {
        unsigned long v;
        if (!arch_get_random_long(&v))
                break;
        hash.l[i] ^= v;
}

/*
 * We mix the hash back into the pool to prevent backtracking
 * attacks (where the attacker knows the state of the pool
```

random: use the architectural HWRNG for the SHA's IV in extract_buf() [December 2013]

“To help assuage the fears of those who think the NSA can introduce a massive hack into the instruction decode and out of order execution engine in the CPU without hundreds of Intel engineers knowing about it (only one of which woud need to have the conscience and courage of
Edward Snowden to spill the beans to the public), use the HWRNG to initialize the SHA starting value, instead of xor'ing it in afterwards.
Signed-off-by: "Theodore Ts'o" <tytso@mit.edu>”



```
/*
 * If we have an architectural hardware random number
 * generator, use it for SHA's initial vector
 */
sha_init(hash.w);
for (i = 0; i < LONGS(20); i++) {
        unsigned long v;
        if (!arch_get_random_long(&v))
                break;
        hash.l[i] = v;
}
```

Opcode Tester

Opcode Tester

- Concept: automate CPU analysis as far as possible
- Execute and analyze instructions in ring 3 and ring 0
  - Command-line tool
  - Input: Sandsifter log file (or similarly formatted instruction list)
  - Filter Sandsifter false positives with XED
  - Clock cycles, functionality analysis
  - Stable ring 0 (kernel driver) #UD handling
    - Test in ring 0 to see if an instruction is valid but privileged
    - Execute **500,000+** illegal instructions in the kernel...And nothing explodes!
    - Segfaults are harder to handle - but only crash the program, not the OS
- Lots more potential for development

https://github.com/cattius/opcodetester

Where next?

Unanswered questions...

- What might we find looking for undocumented instructions in:
    - SGX
    - SMM
    - Other machine modes
    - ME (separate coprocessor)
    - Non-Intel processors

Unanswered questions...

- Why do instructions which *normally* throw #UD sometimes throw #GP instead? Is this a hardware bug, and could it be exploited?

Unanswered questions...

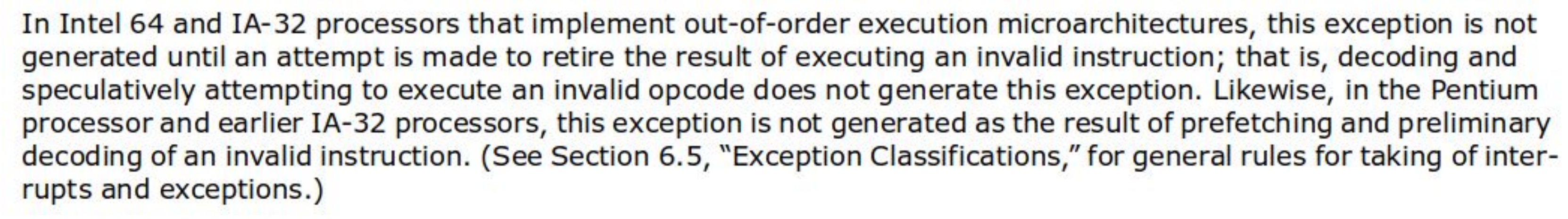
- How can we feasibly test for undocumented behavior which depends on ‘password’ register values?

```
EDI=9C5A203A
```

activates 4 debug MSRs on AMD K7

Unanswered questions...

- Recall: do speculatively executed #UD instructions leave microarchitectural traces?



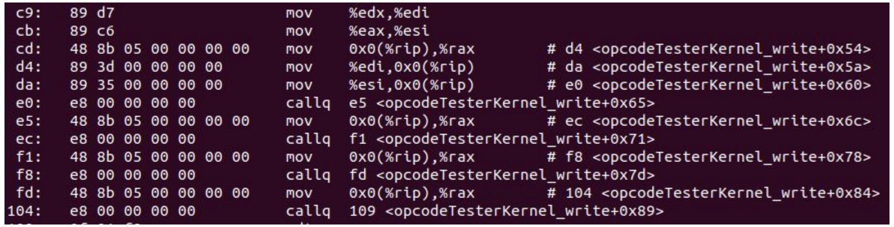
In Intel 64 and IA-32 processors that implement out-of-order execution microarchitectures, this exception is not generated until an attempt is made to retire the result of executing an invalid instruction; that is, decoding and speculatively attempting to execute an invalid opcode does not generate this exception. Likewise, in the Pentium processor and earlier IA-32 processors, this exception is not generated as the result of prefetching and preliminary decoding of an invalid instruction. (See Section 6.5, "Exception Classifications," for general rules for taking of interrupts and exceptions.)

(Source: Intel Developer Manual Vol. 3, section 6.15)

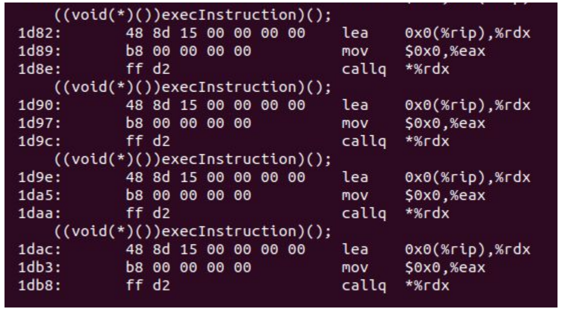
Thank you for listening!
Any questions?

Bonus: incrementing IP - when does an exception occur?

- User program:
  - LEA: Copy RIP + 0 offset to RDX
  - MOV: Set EAX to 0
  - CALL: Push return address (current RIP value) onto stack and jump to absolute address (value of RDX)
  - CALL E8 rel64 calls near with displacement relative to next instruction (so call next instruction, basically)
- Kernel program:
  - Mov RIP to RDX and then CALL the line itself
  - CALL FF calls near absolute indirect address given in register



```
 c9:   89 d7                   mov    %edx,%edi
 cb:   89 c6                   mov    %eax,%esi
 cd:   48 8b 05 00 00 00 00    mov    0x0(%rip),%rax        # d4 <opcodeTesterKernel_write+0x54>
 d4:   89 3d 00 00 00 00       mov    %edi,0x0(%rip)        # da <opcodeTesterKernel_write+0x5a>
 da:   89 35 00 00 00 00       mov    %esi,0x0(%rip)        # e0 <opcodeTesterKernel_write+0x60>
 e0:   e8 00 00 00 00          callq  e5 <opcodeTesterKernel_write+0x65>
 e5:   48 8b 05 00 00 00 00    mov    0x0(%rip),%rax        # ec <opcodeTesterKernel_write+0x6c>
 ec:   e8 00 00 00 00          callq  f1 <opcodeTesterKernel_write+0x71>
 f1:   48 8b 05 00 00 00 00    mov    0x0(%rip),%rax        # f8 <opcodeTesterKernel_write+0x78>
 f8:   e8 00 00 00 00          callq  fd <opcodeTesterKernel_write+0x7d>
 fd:   48 8b 05 00 00 00 00    mov    0x0(%rip),%rax        # 104 <opcodeTesterKernel_write+0x84>
104:   e8 00 00 00 00          callq  109 <opcodeTesterKernel_write+0x89>
```



```
    ((void(*)())execInstruction)();
1d82:       48 8d 15 00 00 00 00    lea    0x0(%rip),%rdx
1d89:       b8 00 00 00 00          mov    $0x0,%eax
1d8e:       ff d2                   callq  *%rdx
    ((void(*)())execInstruction)();
1d90:       48 8d 15 00 00 00 00    lea    0x0(%rip),%rdx
1d97:       b8 00 00 00 00          mov    $0x0,%eax
1d9c:       ff d2                   callq  *%rdx
    ((void(*)())execInstruction)();
1d9e:       48 8d 15 00 00 00 00    lea    0x0(%rip),%rdx
1da5:       b8 00 00 00 00          mov    $0x0,%eax
1daa:       ff d2                   callq  *%rdx
    ((void(*)())execInstruction)();
1dac:       48 8d 15 00 00 00 00    lea    0x0(%rip),%rdx
1db3:       b8 00 00 00 00          mov    $0x0,%eax
1db8:       ff d2                   callq  *%rdx
```