What's cool about x86 assembly language?

Allan B. Cruse
Emeritus Professor
Computer Science and Mathematics
University of San Francisco
A 'background' story

A few years ago the authors of a book about the Linux kernel asked me to review a first-draft of their revised edition, for 'technical accuracy'

In Chapter 3 they made a sweeping statement which, to me, seemed unlikely to be true:

“page-faults are not allowed in kernel mode”

I realized that I knew enough about Linux, and about CPU behavior, that I could quickly write some x86 assembly language to test this claim
An x86 'introductory overview'

If I explain my idea for the 'test', and show you a few of its details, I can cover the basic topics Professor Pacheco had requested:

-- registers and addressing modes

-- integer (and other) operations

-- instruction encoding

-- some x86 historical notes
Historical note

- The x86 processor is a 'work-in-progress'
  - It has evolved over more than three decades
  - Its features have responded to 'market forces'
  - It has fiercely maintained 'backward compatibility'
- So today's x86 implements an immense legacy
  - Earliest 16-bit real-mode still is x86 'startup' mode
  - It can switch to 16-bit or 32-bit protected-mode
  - It can emulate real-mode within virtual-8086 mode
  - It can now be switched to 64-bit enhanced mode
Design-evolution influences

- Faster
- Cheaper
- Smaller
- Stronger
- Safer
- Cooler

--Yet always keeping 'backward compatibility'
The original 8086 registers

Each of these registers is **16-bits** wide

- AX
- BX
- CX
- DX
- SI
- DI
- BP
- SP
- CS
- DS
- ES
- SS
- FLAGS
- IP

Descriptive names are used for these 'nonorthogonal' registers
The 80386 registers

*Here the 'Extended' registers are 32-bits wide*

For 'backward compatibility', the former 16-bit registers are implemented within these 'Extended' 32-bit registers (i.e., as the least-significant 16-bits in each)
Lab machines use Intel's i7 CPU

Here the 'Revised' registers are 64-bits wide

- RAX
- RBX
- RCX
- RDX
- RSI
- RDI
- RBP
- RSP
- CS
- DS
- ES
- FS
- GS
- SS
- RFLAGS
- RIP

general-purpose registers

addressing registers

segment registers

special-purpose registers

For 'backward compatibility', the former 32-bit registers are implemented within these 'Revised' 64-bit registers (i.e., as the least-significant 32-bits in each)

AND there are 8 additional 64-bit general registers (named R8, R9, R10, ... R15)
Sub-registers

Each general-purpose register can, by using different assembly language names, be treated in programs as having a width of 64-bits, 32-bits, 16-bits, or 8-bits.

**EXAMPLE:** The accumulator register can be called \texttt{AL}, or \texttt{AX}, or \texttt{EAX}, or \texttt{RAX}.

Similarly for the other general-purpose registers (e.g., \texttt{B}, \texttt{C}, and \texttt{D}).
Little-Endian vs Big-Endian

- A signature feature of x86 processors (as well as Intel's various peripheral-device controllers) is its use of the “little-endian” convention when representing multi-byte data-values.

- A programming advantage of x86 “little-endian” data-storage convention is that operands with differing widths can be addressed in a uniform manner, by referencing their earliest address.
Address-Spaces

An unusual feature of the x86 architecture is its use of different address-spaces for accesses to memory-locations versus I/O device-registers.
x86 Privilege Levels

Current Privilege Level (CPL)

Code-segment selector in CS register
Earliest instruction-categories

- Data Transfer
- Control Transfer
- Arithmetical/Logical
- Processor Control
- Bit-manipulation
- String-manipulation

Some useful general rules apply to these categories
Arithmetical/Logical

- ADD, SUB, ADC, SBB, CMP, NEG
- AND, OR, XOR, TEST, NOT
- MUL, IMUL, DIV, IDIV
- INC, DEC
- CBW, CWD, CWDE, CDQ
- AAA, AAS, AAM, AAD, DAA, DAS
Data Transfer

- MOV, XCHG, XLAT, BSWAP
- PUSH, POP, PUSHA, POPA
- LEA, LDS, LES, LSS, LFS, LGS
- LAHF, SAHF, PUSHF, POPF
- IN, OUT
Control Transfer

• JMP, CALL, RET,
• JC/JNC, JS/JNS, JZ/JNZ, JP/JNP, JO/JNO
• JA/JAE, JB/JBE, JL/JLE, JG/JGE
• LOOP, LOOPE/LOOPZ, LOOPNE/LOOPNZ
• JCXZ/JECXZ
• INT, IRET, INTO, INT3
Processor Control

- CLI, STI
- CLD, STD
- CLC, STC, CMC
- NOP, HLT, WAIT, ESC, LOCK
- CS:, DS:, ES:, SS:, FS:, GS:
- INVD, WBINVD, INVLD
Bit-manipulation

- SHL, SHR, SAL, SAR
- ROL, ROR, RCL, RCR
- BT, BTS, BTR, BTC
- SHLD, SHRD
String Manipulation

- MOVS
- CMPS
- SCAS
- STOS
- LODS
- INS/OUTS
- REP/REPE/REPZ
- REPNE/REPNZ
The FLAGS register

Status Flags automatically get modified by Arithmetic and Logic instructions

Control Flags are only modified by an explicit use of a specific instruction
Assembly Language Statement Format

Example 1: An unlabeled double-operand data-transfer instruction-statement

```plaintext
mov %rdx, %rax  # copies value from RDX into RAX
```

Example 2: An unlabeled single-operand arithmetical instruction-statement

```plaintext
inc %rbx        # increases the value in RBX by +1
```
# x86 Machine-Instruction Format

x86 instructions may be of varying lengths, between 1 and 15 bytes inclusive

<table>
<thead>
<tr>
<th>Legacy Prefixes</th>
<th>REX Prefix</th>
<th>Opcode</th>
<th>ModR/M</th>
<th>SIB</th>
<th>Displacement</th>
<th>Immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Grp 1, Grp 2, Grp 3, Grp 4 (optional)</td>
<td>(optional)</td>
<td>1-, 2-, or 3-byte opcode</td>
<td>1 byte (if required)</td>
<td>1 byte (if required)</td>
<td>Address displacement of 1, 2, or 4 bytes</td>
<td>Immediate data of 1, 2, or 4 bytes or none</td>
</tr>
</tbody>
</table>

**Example:**

xchg %rax, 0x40(%rbx, %rcx, 8)  # exchange register with memory

**Machine-code (hex):**

48 87 44 CB 40  # 5-byte instruction-length

```
01 000 100
11 001 011
1  rax  sib
8  rcx  rbx
```
The One-Byte Opcode-Table

Note: The 0x0F byte is the 'escape' to the Two-Byte Opcode Table.

Note: The bytes 0xD8, 0xD9, ..., 0xDF are 'escapes' to co-processor opcode tables.
CISC string-instruction example

msg: .asciz "Hello, World!"

# for computing the length of a 'null-terminated' character-string

.equ MAXLEN, 65535

_strlen: xor %al, %al  # null-byte in register AL
        mov $msg, %edi   # EDI = string's address
        mov $MAXLEN, %ecx # ECX = maximum length
        cld               # use forward processing
        repne scasb       # scan for the final byte
        sub $msg, %edi    # subtract initial from final

# now EDI will contain the number of bytes scanned
Now, what is a 'page-mapping'?

The CPU only “sees” those portions of the physical memory which are currently “mapped” into its virtual address space.
And, what is a 'task-switch'? 

The CPU only “sees” those portions of the physical memory which are currently “mapped” into its virtual address space.
So, what is a 'page-fault'?

- If the CPU tries to access a memory-address that is not currently “mapped”-- or that it lacks appropriate privilege – this will be detected by the CPU as an illegal memory-reference -- and is known as a 'page-fault exception'

- The CPU will automatically jump to a routine in kernel space, known as an 'exception handler', after first saving a tiny amount of information about the situation that led to this 'exception'

- For more details we need some background...
The kernel's stack

NOTE: The CS register's value includes the 2-bit 'Current Privilege Level' (CPL)
Basic idea for my page-fault tester

# my global variables
unsigned short selector;  # global variable for storing 16-bit value
unsigned long oldisr14;  # global variable for storing 64-bit constant

# my Interrupt Service Routine 'front-end' for handling 'page-fault' exceptions
isr_entry:

    mov 16(%rsp), %rax  # data-transfer from memory to register
    mov %ax, selector   # data-transfer from register to memory
    jmp *oldisr14      # indirect control-transfer via memory

# NOTE: this simple code-fragment ignores a few crucial issues...
Essential issues to confront

• Another 'page-fault' is likely to happen quickly
  • So any saved 'selector' value will get 'overwritten'

• Any interrupt-routine must preserve registers
  • With my 'handler' the RAX register gets 'clobbered'

• All SMP interrupt-routines must be 'reentrant'
  • Multiple CPUs could access 'selector' in parallel
1. use an array of storage-cells

```c
#define MAXNUM 255

unsigned short selector[MAXNUM];  // enough space for lots of 16-bit values
unsigned long oldisr14;           // to store address of the original handler
unsigned long pgfaults = 0;       // keep count of page-fault occurrences

isr_entry:
    mov pgfaults, %rbx              // setup count as array-index in EBX
    mov 16(%rsp), %rax             // copy CS-selector-image into RAX
    mov %ax, selector( , %rbx, 2)  // and save it into the next array cell
    incq pgfaults                  // increment count for the next fault
    jmpq *oldisr14                 // transfer to the normal fault-handler
```
2. Preserve the 'working' registers

#define MAXNUM 255

unsigned short selector[MAXNUM]; # enough space for lots of 16-bit values
unsigned long oldisr14; # to store address of the original handler
unsigned long pgfaults = 0; # keep count of page-fault occurrences

 isr_entry:
    push %rax # save the RAX register-value
    push %rbx # save the RBX register-value
    mov pgfaults, %rbx # setup count as array-index in EBX
    mov 32(%rsp), %rax # copy CS-selector-image into RAX
    mov %ax, selector( , %rbx, 2) # and save it into the next array cell
    incq pgfaults # increment count for the next fault
    pop %rbx # recover the RBX register-value
    pop %rax # recover the RAX register-value
    jmpq *oldisr14 # transfer to the normal fault-handler
Stack after pushing RAX and RBX

The kernel's stack

- RFLAGS
- CS
- RIP
- error-code
- RAX
- RBX

NOTE: The CS register's value includes the 2-bit 'Current Privilege Level' (CPL)
3. Increment 'pgfaults' atomically

#define MAXNUM 255

unsigned short selector[MAXNUM];  # enough space for lots of 16-bit values
unsigned long oldisr14;  # to store address of the original handler
unsigned long pgfaults = 0;  # keep count of page-fault occurrences

isr_entry:
push %rax  # save the RAX register-value
push %rbx  # save the RBX register-value
mov $1, %rbx  # amount of the 'pgfaults' increment
xadd %rbx, pgfaults  # use XADD for SMP-safe update
mov 32(%rsp), %rax  # copy CS-selector-image into RAX
mov %ax, selector(, %rbx, 2)  # and save it into the next array cell
pop %rbx  # recover the RBX register-value
pop %rax  # recover the RAX register-value
jmpq *oldisr14  # transfer to the normal fault-handler
But now a 'new' issue arises

- After a certain number of page-faults occur, our 'selector[ ]' storage-array will overflow!

- Can you think of a way to solve that problem? *(There is more than one way you could do it)*
#define MAXNUM 255  // maximum number of saved selectors
const long max = MAXNUM;
unsigned long oldisr14;
unsigned long pgfaults = 0;
unsigned short selector[MAXNUM];

//--------- INTERRUPT SERVICE ROUTINE ---------
void isr_entry( void );
asm(" .text " ) ;
asm(" .type isr_entry, @function ");
asm("isr_entry: ");
asm(" push %rax ");
asm(" push %rbx ");
asm(" mov pgfaults, %rax ");
asm(" cmp max, %rax ");
asm(" jge bypass ");
asm(" mov $1, %rbx ");
asm(" xadd %rbx, pgfaults ");
asm(" mov 32(%rsp), %rax ");
asm(" mov %ax, selector(, %rbx, 2) ");
asm("bypass: ");
asm(" pop %rbx ");
asm(" pop %rax ");
asm(" jmpq *oldisr14 ");

//---------
Output from './proc/faultcpl' demo

Below are shown the code-segment selectors for the first 255 page-faults:

```
0033 0033 0033 0033 0033 0033 0033 0033 0033 0033 0033 0033 0033 0033 0033 0033
0033 0033 0033 0033 0033 0033 0033 0033 0033 0033 0033 0033 0033 0033 0033 0033
0033 0010 0010 0010 0010 0033 0033 0033 0033 0033 0033 0033 0033 0033 0033 0033
0033 0033 0033 0033 0033 0033 0033 0033 0033 0033 0033 0033 0033 0033 0033 0033
0033 0033 0033 0033 0033 0033 0033 0033 0033 0033 0033 0033 0033 0033 0033 0033
0033 0033 0033 0033 0033 0033 0033 0033 0033 0033 0033 0033 0033 0033 0033 0033
0033 0033 0033 0033 0033 0033 0033 0033 0033 0033 0033 0033 0033 0033 0033 0033
0033 0033 0033 0033 0033 0033 0033 0033 0033 0033 0033 0033 0033 0033 0033 0033
0033 0033 0033 0033 0033 0033 0033 0033 0033 0033 0033 0033 0033 0033 0033 0033
0033 0033 0033 0033 0033 0033 0033 0033 0033 0033 0033 0033 0033 0033 0033 0033
0033 0033 0033 0033 0033 0033 0033 0033 0033 0033 0033 0033 0033 0033 0033 0033
0033 0033 0033 0033 0033 0033 0033 0033 0033 0033 0033 0033 0033 0033 0033 0033
0033 0033 0033 0033 0033 0033 0033 0033 0033 0033 0033 0033 0033 0033 0033 0033
0033 0033 0033 0033 0033 0033 0033 0033 0033 0033 0033 0033 0033 0033 0033 0033
0033 0033 0033 0033 0033 0033 0033 0033 0033 0033 0033 0033 0033 0033 0033 0033
0033 0033 0033 0033 0033 0033 0033 0033 0033 0033 0033 0033 0033 0033 0033 0033
0033 0033 0033 0033 0033 0033 0033 0033 0033 0033 0033 0033 0033 0033 0033 0033
0033 0033 0033 0033 0033 0033 0033 0033 0033 0033 0033 0033 0033 0033 0033 0033
0033 0033 0033 0033 0033 0033 0033 0033 0033 0033 0033 0033 0033 0033 0033 0033
0033 0033 0033 0033 0033 0033 0033 0033 0033 0033 0033 0033 0033 0033 0033 0033
0033 0033 0033 0033 0033 0033 0033 0033 0033 0033 0033 0033 0033 0033 0033 0033
0033 0033 0033 0033 0033 0033 0033 0033 0033 0033 0033 0033 0033 0033 0033 0033
0033 0033 0033 0033 0033 0033 0033 0033 0033 0033 0033 0033 0033 0033 0033 0033
0033 0033 0033 0033 0033 0033 0033 0033 0033 0033 0033 0033 0033 0033 0033 0033
0033 0033 0033 0033 0033 0033 0033 0033 0033 0033 0033 0033 0033 0033 0033 0033
0033 0033 0033 0033 0033 0033 0033 0033 0033 0033 0033 0033 0033 0033 0033 0033
```

Observe that 10 of these page-faults occurred while executing in kernel-mode
So what's the “coolest” thing?

- Knowing x86 assembly language gives you the power to investigate all capabilities of the CPUs being used in today's most common platforms.
- It's like having Galileo's telescope: you can see whether 'conventional wisdom' is correct or not.
- And you can investigate new x86 features as soon as they are implemented, without waiting until software tools or programming languages have been developed to offer support for them.
Model Specific Registers (MSR)

- As enhancements get added to the x86 CPU, additional registers are needed for controls.
- So Intel devised a scheme for accessing up to 4-billion so-called MSRs by means of just two privileged instructions: `rdmsr` and `wrmsr`.

**EXAMPLE:**  # reading from a 64-bit MSR

```
mov $0x19C, %ecx # load MSR's ID-number into ECX
rdmsr            # read the Model Specific Register
mov %eax, msr_lo # EAX holds least-significant 32-bits
mov %edx, msr_hi # EDX holds most-significant 32-bits
```
Intel-x86 Processor Modulation

This is a fairly recent Intel-x86 enhancement that allows monitoring and controlling the temperature inside the CPU, to prevent circuit-damage due to 'overheating'.

Figure 14-5. Processor Modulation Through Stop-Clock Mechanism
Intel-x86 Thermal Status register

Figure 14-12. IA32_THERM_STATUS Register
Table 14-1. On-Demand Clock Modulation Duty Cycle Field Encoding

<table>
<thead>
<tr>
<th>Duty Cycle Field Encoding</th>
<th>Duty Cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td>000B</td>
<td>Reserved</td>
</tr>
<tr>
<td>001B</td>
<td>12.5% (Default)</td>
</tr>
<tr>
<td>010B</td>
<td>25.0%</td>
</tr>
<tr>
<td>011B</td>
<td>37.5%</td>
</tr>
<tr>
<td>100B</td>
<td>50.0%</td>
</tr>
<tr>
<td>101B</td>
<td>63.5%</td>
</tr>
<tr>
<td>110B</td>
<td>75%</td>
</tr>
<tr>
<td>111B</td>
<td>87.5%</td>
</tr>
</tbody>
</table>
IA32_THERM_INTERRUPT

Figure 14-13. IA32_THERM_INTERRUPT Register
Our 'celsius.c' example

- We wrote this kernel module to let users view the value in a cpu's THERM_STATUS register.
- Its 'Digital Readout' field will show how far the processor's current temperature is below what Intel regards as its maximum (called 'Tj Max').
- The processor shuts down if this difference is 0.
- These temperatures are in degrees-Celsius.
- (Note: AMD's cpus employ a different scheme)
'/proc/celsius'

- Our 'celsius.c' module creates this pseudo-file, which a user can access with this command:
  
  $ cat /proc/celsius

- Here's what the screen-output would look like:

  4 CPUs detected
  
  cpu1  thermal_status = 882E0000  54-degrees (celsius)
  cpu2  thermal_status = 882C0000  56-degrees (celsius)
  cpu3  thermal_status = 88280000  60-degrees (celsius)
  cpu4  thermal_status = 882E0000  54-degrees (celsius)

Note: This demo was run on a machine with an Intel Core 2 Quad processor.
Demos and Questions

Speaker's website: <http://cs.usfcs.edu/~cruse/>