x86/x64 Assembly + Python = New CPU Architecture
(to rule the world)

Cesare Di Mauro
Appunti Digitali (www.appuntidigitali.it)
EuroPython 2013 – Firenze (Florence)
July 5, 2013
The geek nature

Everyone else work isn’t cool

- Write your own language (and compiler)
- Write your own o.s. (and filesystem, of course)
- Write your own CPU architecture

No matter how much time it takes: it should be done!
Sometimes it pays...

A Dutch invented a wonderful language which actually billions of coders love

A Finnish wrote a brand new Unix o.s. which runs on billions of devices

A Yankee* designed a very low-cost 8-bit microprocessor which was inside billions of devices

* Chuck Peddle
...sometimes it’s just a dream

- A 1024-bit processor
- Works on bit-fields
- 2 memory sources and one memory destination
- Hundreds registers
- Hundreds and hundreds instructions
- Very complicated instructions

Young guys often make mistakes...
A market with a few contenders

Many CPU architectures died or fit just a niche
The market is substantially dominated by two players
Status & new needs

Intel shines on performance
ARM dominates in power efficiency

Mobile devices are every day more powerful
Mobile devices integrates more memory
Mobile devices should consume a few watts
Servers need performance
Servers need efficient cooling
The legacy burden

Intel should support several ISAs: 8086, 80286, 80386, x64
Intel inherits complex instructions (BCD, stack frame, strings, etc.) from 8086/80186
Intel carries old memory models (segmentation)
Intel has many instructions

ARM has several ISAs too: ARM(32), Thumb2, ThumbEE, Jazilla, ARM64
ARM has some complex instructions (load multiple regs)
ARM has many instructions
A new ISA: NEx64T

Trivial instruction decoding (comparable to ARM/Thumb-2)
Very good code density
Excellent performance
Very easy x86/x64 emulation (about the same speed)
Source (assembly) level x86/x64 compatible
32 registers, up to 128 SIMD registers (up to 1024 bits)
Smart and efficient code padding
Instructions “promoted” to 2, 3, or even 4 operands
Scalable from embedded to HPC
Show me the numbers!

A good idea is useless without a proof of goodness

A benchmark must be set-up to show how the new ISA compares to some industry standard

NEx64T is a “rewrite” of x86/x64: they are the reference!

A NEx64T “model” is needed. Here comes Python...
Setting key points for the model

A non-existent ISA is difficult to compare

1) Mapping every x86 or x64 instruction to NEx64T

2) Collect stats about code density

3) Compare decoders (front-ends)

4) NEx64T shares most of the x64 infrastructure (back-end)
An x86 disassembler for Python

diStorm3 library (http://code.google.com/p/distorm/)

- Lightweight, simple, fast, with good docs and support
- Disassembles multiple instructions (buffer-based approach)
- It was a bit bugged. Fixed and reported some issues; others still unfixed
- Supports 8086..80286 (16-bit), 80386+ (32-bit), x64 (64-bit), and latest extensions (AVX too)
Some improvements to diStorm3

Python wrapper lacks some C interface consts and functions. Added them.

Multiple instructions decoding is inefficient when just one instruction is needed.

Wrote new decode and format functions to achieve the task in a simpler and efficient way.

Tweaked the Instruction class to get more useful info.
dislib: PE binaries disassembler

diStorm3 library comes with a basic Microsoft’s PE binary executables disassembler: dislib.py

Shows little PE information

Added more

Disassembles the first instructions (from the entry point).

Needed a huge work to track instructions and disassemble as much as possible of them
## Tracking instructions

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1000015a4</td>
<td>4883ec28</td>
<td>SUB RSP, 0x28</td>
</tr>
<tr>
<td>0x1000015a8</td>
<td>e873020000</td>
<td>CALL 0x100001820</td>
</tr>
<tr>
<td>0x1000015ad</td>
<td>4883c428</td>
<td>ADD RSP, 0x28</td>
</tr>
<tr>
<td>0x1000015b1</td>
<td>e9d2fcfff</td>
<td>JMP 0x100001288</td>
</tr>
<tr>
<td>0x100001820</td>
<td>48895c2418</td>
<td>MOV [RSP+0x18], RBX</td>
</tr>
<tr>
<td>0x100001825</td>
<td>57</td>
<td>PUSH RDI</td>
</tr>
<tr>
<td>0x100001826</td>
<td>4883ec20</td>
<td>SUB RSP, 0x20</td>
</tr>
<tr>
<td>0x10000182a</td>
<td>488b05df080000</td>
<td>MOV RAX, [RIP+0x8df]</td>
</tr>
<tr>
<td>0x100001831</td>
<td>488364243000</td>
<td>AND QWORD [RSP+0x30], 0x0</td>
</tr>
<tr>
<td>0x100001837</td>
<td>48bf32a2df2d992b0000</td>
<td>MOV RDI, 0x2b992ddfa232</td>
</tr>
<tr>
<td>0x100001841</td>
<td>483bc7</td>
<td>CMP RAX, RDI</td>
</tr>
<tr>
<td>0x100001844</td>
<td>740c</td>
<td>JZ 0x100001852</td>
</tr>
<tr>
<td>0x100001846</td>
<td>48f7d0</td>
<td>NOT RAX</td>
</tr>
<tr>
<td>0x100001849</td>
<td>488905c8080000</td>
<td>MOV [RIP+0x8c8], RAX</td>
</tr>
<tr>
<td>0x100001850</td>
<td>eb76</td>
<td>JMP 0x1000018c8</td>
</tr>
<tr>
<td>0x100001288</td>
<td>488bc4</td>
<td>MOV RAX, RSP</td>
</tr>
</tbody>
</table>
Collecting addresses

Array to mark already disassembled instructions bytes

A queue (FIFO) of addresses to disassemble

When an instruction is disassembled -> mark its bytes

When a jump / call is found -> put address (arg) in queue

Jumps/ rets/ ints/ illegal instructions/ etc. stops disassembly
Collecting instructions stats

Instruction length

Mnemonic

Arguments

collections.Counter

Constant

Addressing mode
Translating x86/x64 to NEx64T

A new module (nex64t.py) written from scratch. About 3K lines

Converts the addressing mode to one of the available addressing modes

Converts constant to internal, packed format (if possible) to reduce space

Packs opcode, addressing mode, and constant
Comparing ISAs stats - 32 bits

Adobe Photoshop CS6 public beta

Total Instructions: 1746569

<table>
<thead>
<tr>
<th>Class</th>
<th>Count</th>
<th>%</th>
<th>Avg sz</th>
<th>NE/x64T</th>
<th>Diff</th>
<th>Diff %</th>
</tr>
</thead>
<tbody>
<tr>
<td>INTEGER</td>
<td>1631136</td>
<td>93.39</td>
<td>3.2</td>
<td>3.4</td>
<td>0.2</td>
<td>+5.6%</td>
</tr>
<tr>
<td>FPU</td>
<td>114521</td>
<td>6.56</td>
<td>3.2</td>
<td>3.6</td>
<td>0.4</td>
<td>+13.9%</td>
</tr>
<tr>
<td>SSE</td>
<td>912</td>
<td>0.05</td>
<td>4.0</td>
<td>4.7</td>
<td>0.6</td>
<td>+16.1%</td>
</tr>
</tbody>
</table>

Size: 5634556  NE/x64T Size: 5982402  Diff: 347846

Global result: +6.2%
Comparing ISAs stats - 64 bits

Adobe Photoshop CS6 public beta

Total Instructions: 1737331

<table>
<thead>
<tr>
<th>Class</th>
<th>Count</th>
<th>%</th>
<th>Avg sz</th>
<th>NEx64T</th>
<th>Diff</th>
<th>Diff %</th>
</tr>
</thead>
<tbody>
<tr>
<td>INTEGER</td>
<td>1638505</td>
<td>94.31</td>
<td>4.3</td>
<td>3.5</td>
<td>-0.8</td>
<td>-17.8%</td>
</tr>
<tr>
<td>SSE</td>
<td>93942</td>
<td>5.41</td>
<td>5.2</td>
<td>4.5</td>
<td>-0.7</td>
<td>-12.9%</td>
</tr>
<tr>
<td>FPU</td>
<td>4884</td>
<td>0.28</td>
<td>3.1</td>
<td>3.2</td>
<td>0.0</td>
<td>+1.1%</td>
</tr>
</tbody>
</table>

Size: 7556180  NEx64T Size: 6239790  Diff: -1316390

Global result: **-17.4%**
Conclusions

Python was a fundamental tool to develop NEx64T

Rapidly built a working model

Let experiment MANY ideas

Two new ISA versions completed; developing a third one

Quickly get hands on numbers -> comparing with real world