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 disassambler 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

```
0x1000015a4 (4) 4883ec28             SUB RSP, 0x28
0x1000015a8 (5) e873020000           CALL 0x100001820
0x1000015ad (4) 4883c428             ADD RSP, 0x28
0x1000015b1 (5) e9d2fcffff           JMP 0x100001288

0x100001820 (5) 48895c2418           MOV [RSP+0x18], RBX
0x100001825 (1) 57                   PUSH RDI
0x100001826 (4) 4883ec20             SUB RSP, 0x20
0x10000182a (7) 488b05df080000       MOV RAX, [RIP+0x8df]
0x100001831 (6) 488364243000         AND QWORD [RSP+0x30], 0x0
0x100001837 (a) 48bf32a2df2d992b0000 MOV RDI, 0x2b992ddfa232
0x100001841 (3) 483bc7               CMP RAX, RDI
0x100001844 (2) 740c                 JZ 0x100001852
0x100001846 (3) 48f7d0               NOT RAX
0x100001849 (7) 488905c8080000       MOV [RIP+0x8c8], RAX
0x100001850 (2) eb76                 JMP 0x1000018c8

0x100001288 (3) 488bc4               MOV RAX, RSP
```

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

```
0x1000015a4 (4) 4883ec28          SUB RSP, 0x28
0x1000015a8 (5) e873020000        CALL 0x100001820
0x1000015ad (4) 4883c428          ADD RSP, 0x28
0x1000015b1 (5) e9d2fcffff        JMP 0x100001288

0x100001820 (5) 48895c2418        MOV [RSP+0x18], RBX
```

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

Class          Count          %  Avg sz  NEx64T    Diff
INTEGER      1631136      93.39     3.2     3.4     0.2  +5.6%
FPU           114521       6.56     3.2     3.6     0.4 +13.9%
SSE              912       0.05     4.0     4.7     0.6 +16.1%

Size: 5634556  NEx64T Size: 5982402  Diff: 347846

                    Global result: +6.2%
```

Comparing ISAs stats - 64 bits

Adobe Photoshop CS6 public beta

```
Total Instructions: 1737331


Class          Count          %  Avg sz  NEx64T    Diff
INTEGER      1638505      94.31     4.3     3.5    -0.8 -17.8%
SSE            93942       5.41     5.2     4.5    -0.7 -12.9%
FPU             4884       0.28     3.1     3.2     0.0  +1.1%


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